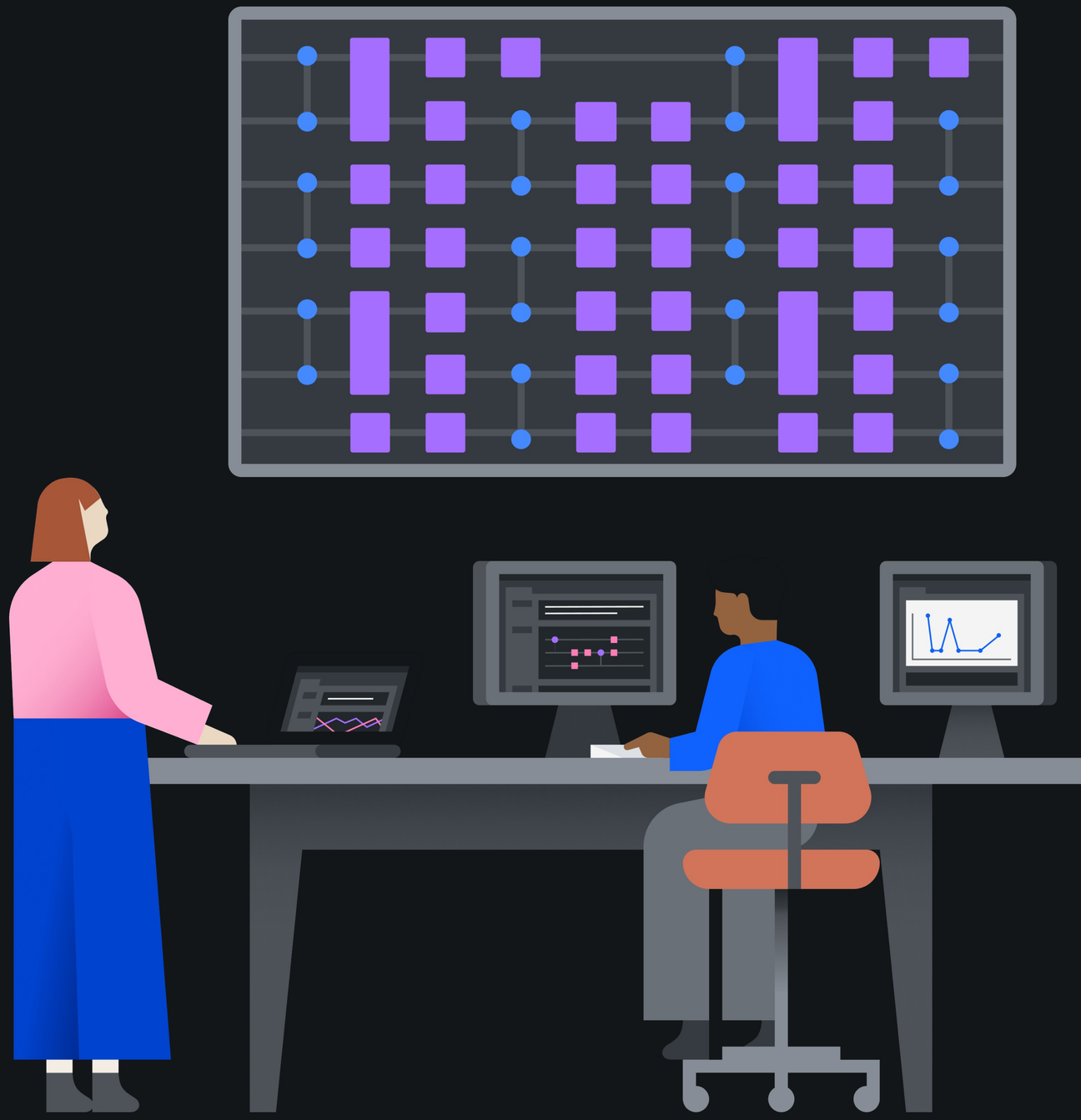
Qiskit Global Summer School 2024

Quantum Circuit Compilation with Qiskit

Matthew Treinish
Senior Software Engineer
IBM

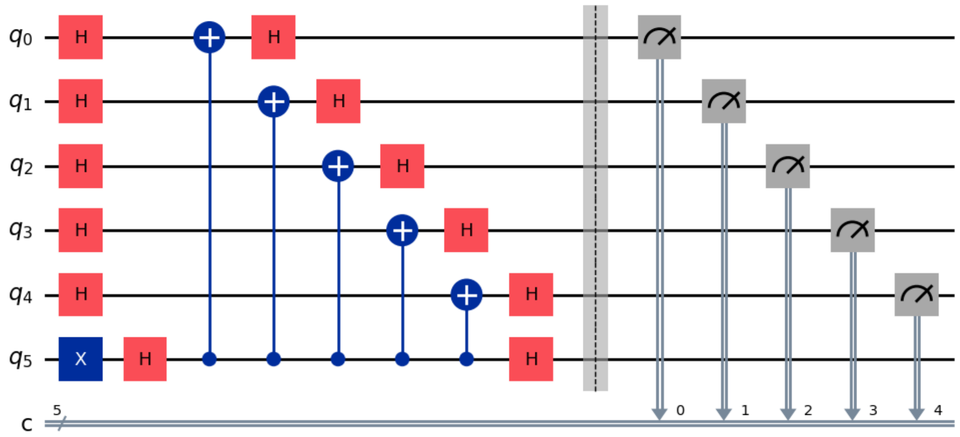


Note on terminology

- In Qiskit the quantum circuit compiler is called the "transpiler"
- This is a historical name and doesn't accurately represent its capabilities or function
- Typically, when you look up the term "transpiler" it refers to a source-to-source compiler. Such as a compiler that translates Python to Rust.
- I'll use "compile"/"compiler" and "transpile"/"transpiler" interchangeably during this lecture; I'm referring to the same thing.

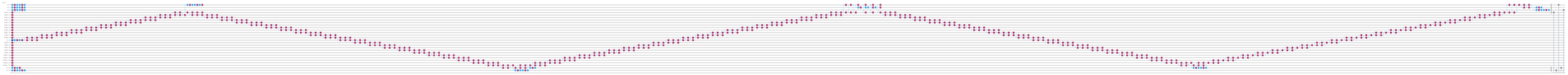
Why do we need quantum circuit compilation?

- Despite programming at a low level there is still abstractions in our representation that can't be executed by hardware
- There are numerous constraints with hardware that need to be accounted for
- The compiler's job is to take the high-level abstract circuit and output an optimized circuit that is capable of running on a given quantum computer

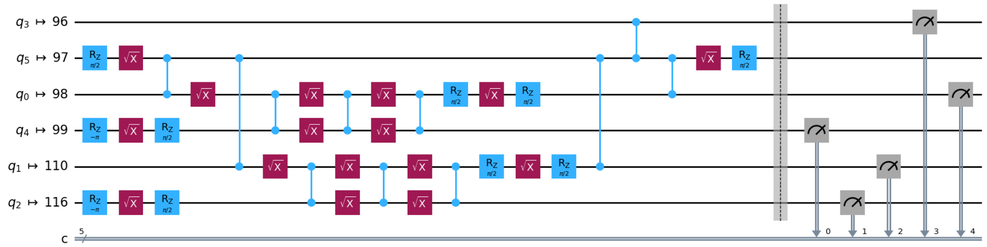


Why do we need quantum circuit compilation?

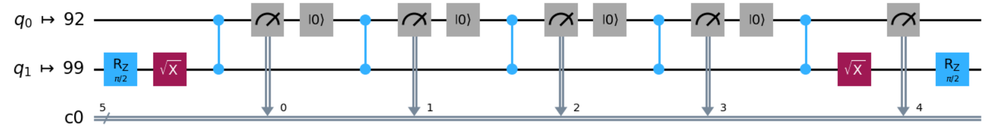
Bad compilation:



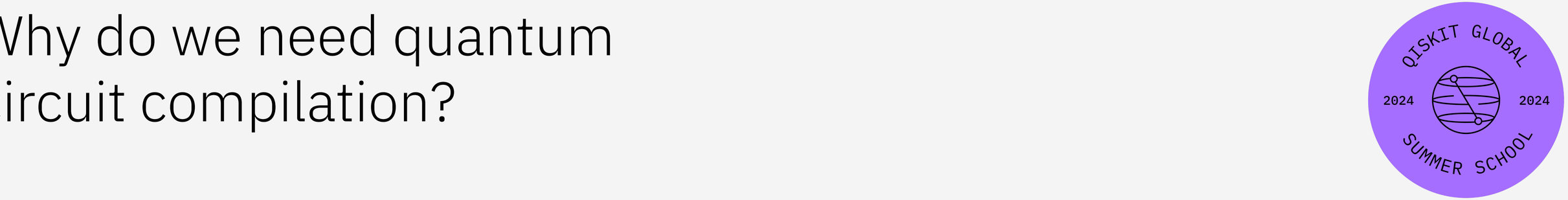
Good compilation:



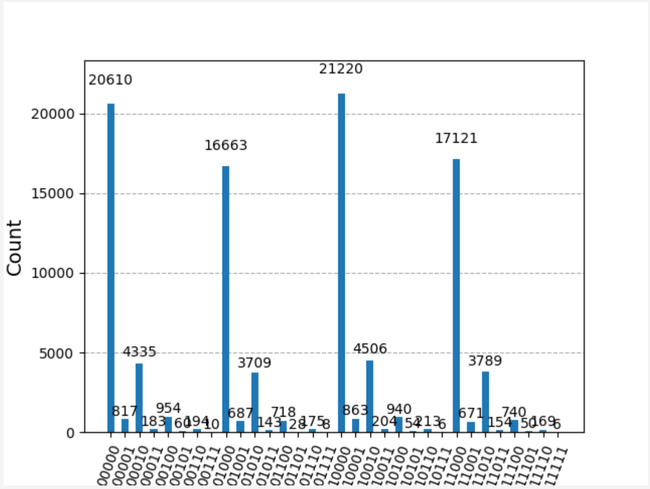
Better compilation:



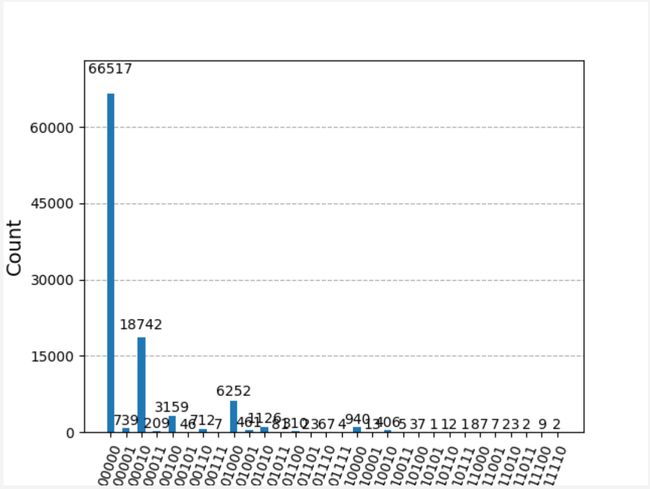
Why do we need quantum circuit compilation?



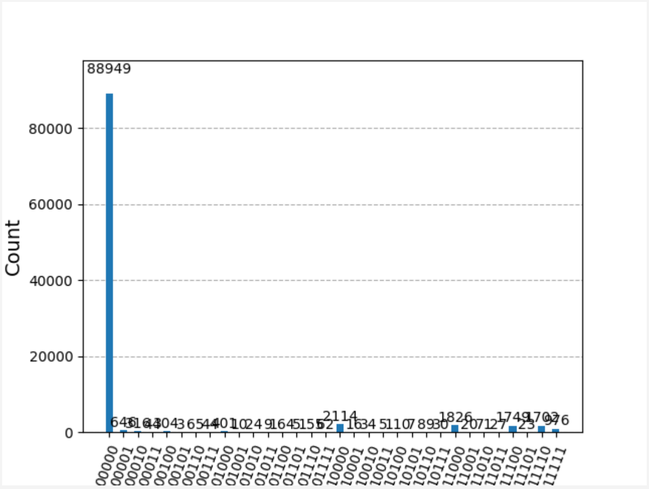
Bad compilation:



Good compilation:



Better compilation:



Modelling Hardware constraints

Operations

Each Quantum Computer only supports a set of operations

Typically, hardware supports a universal gate set that can be used to construct any unitary operation

All qubits do not necessarily need to support the same set of operations.

Operations supported on different generations of IBM Hardware:

IBM Heron Systems:

CZ, Rz, sqrt(X) , X

IBM Eagle Systems:

ECR, Rz, sqrt(X) , X

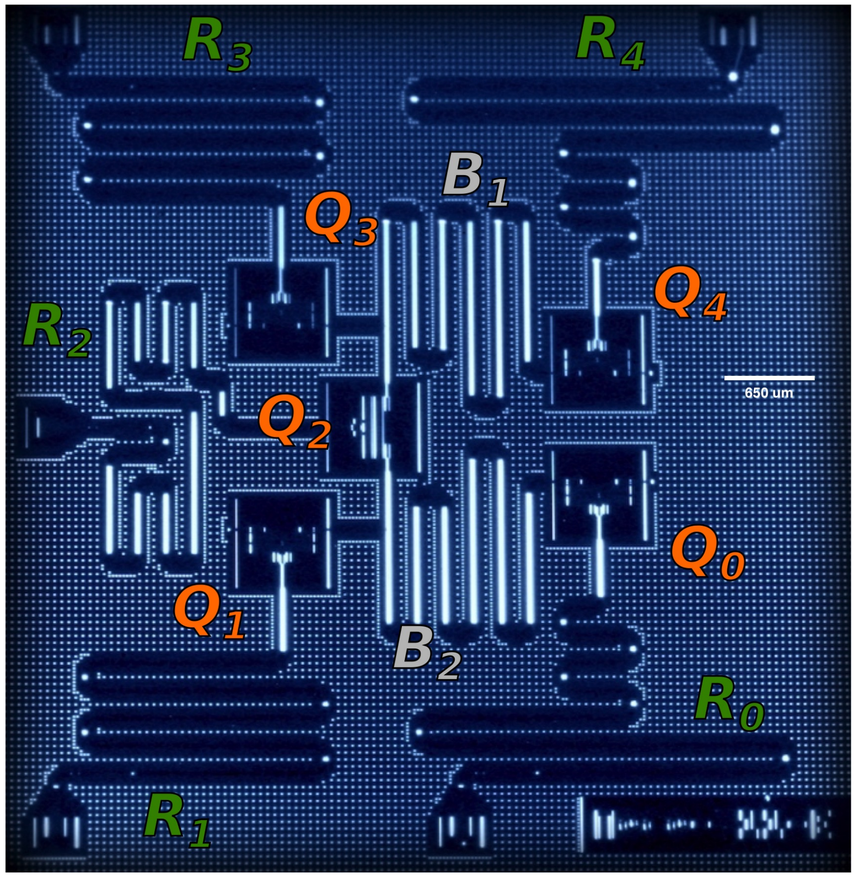
IBM Falcon Systems(retired):

ECR, CX, Rz, sqrt(X) , X

Qubit connectivity

- Qubits may have limited connectivity
- For IBM's super conducting hardware this is limited to qubits that have physical connectivity.
- For multi-qubit gates this means we can only run these operations where there is connectivity between qubits.

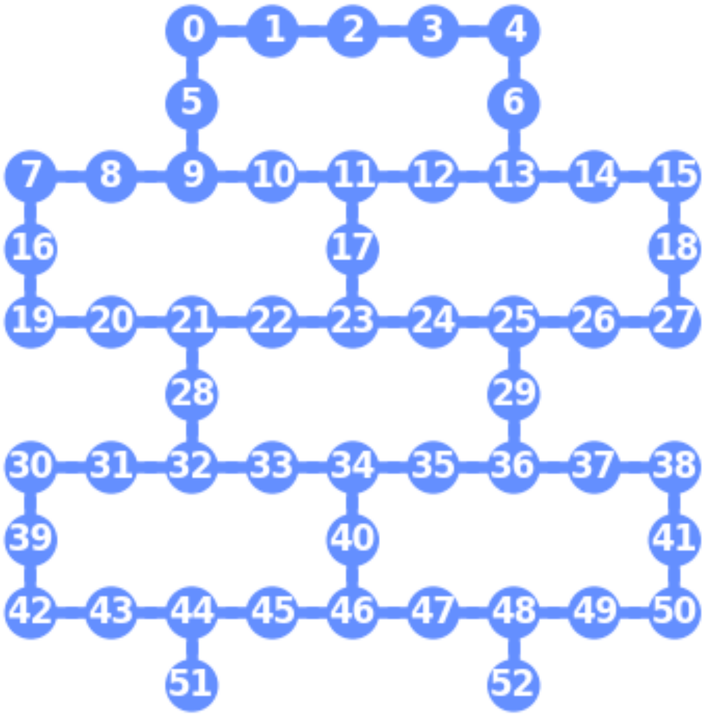
IBM Yorktown 5 qubits



Qubit connectivity

- Qubits may have limited connectivity
- For IBM's super conducting hardware this is limited to qubits that have physical connectivity.
- For multi-qubit gates this means we can only run these operations where there is connectivity between qubits.

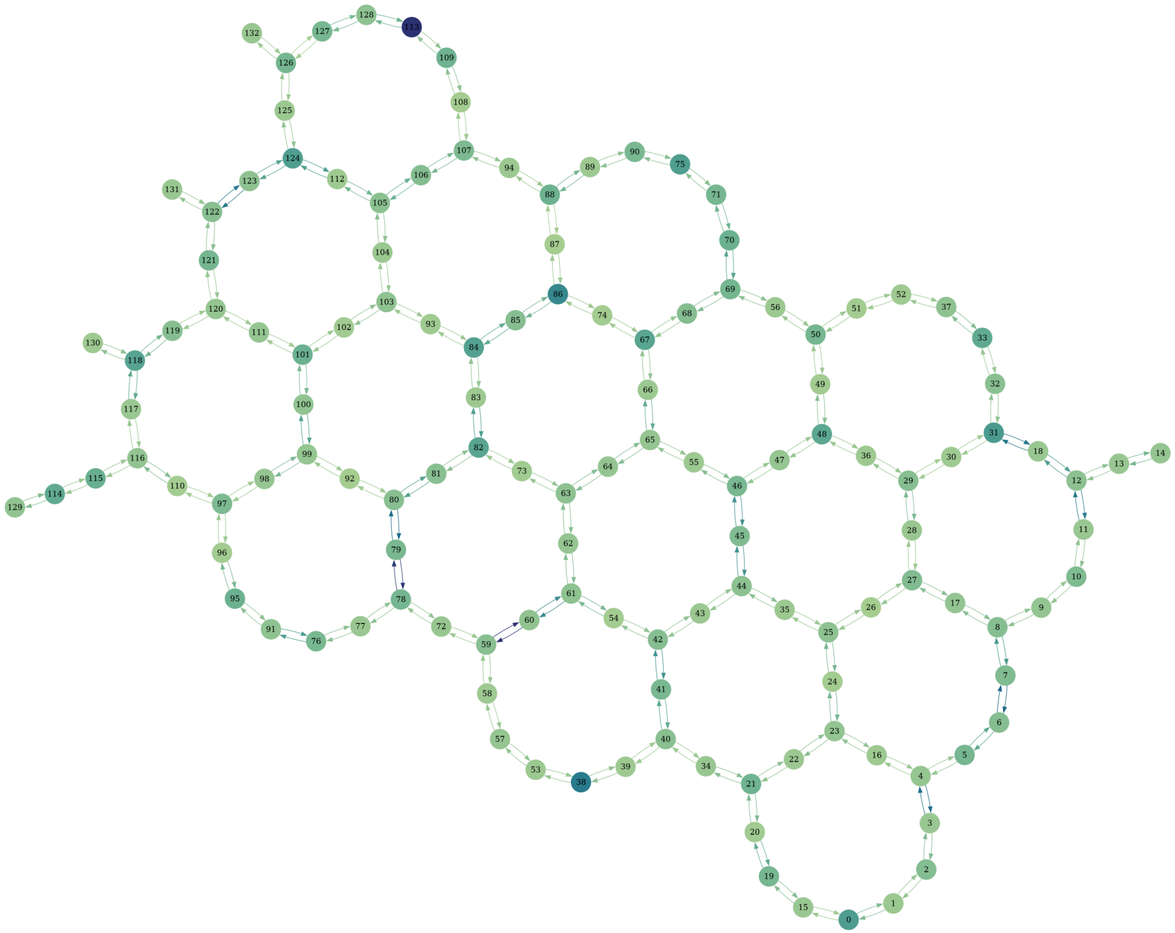
IBM Rochester (retired) 53 qubits



Qubit connectivity

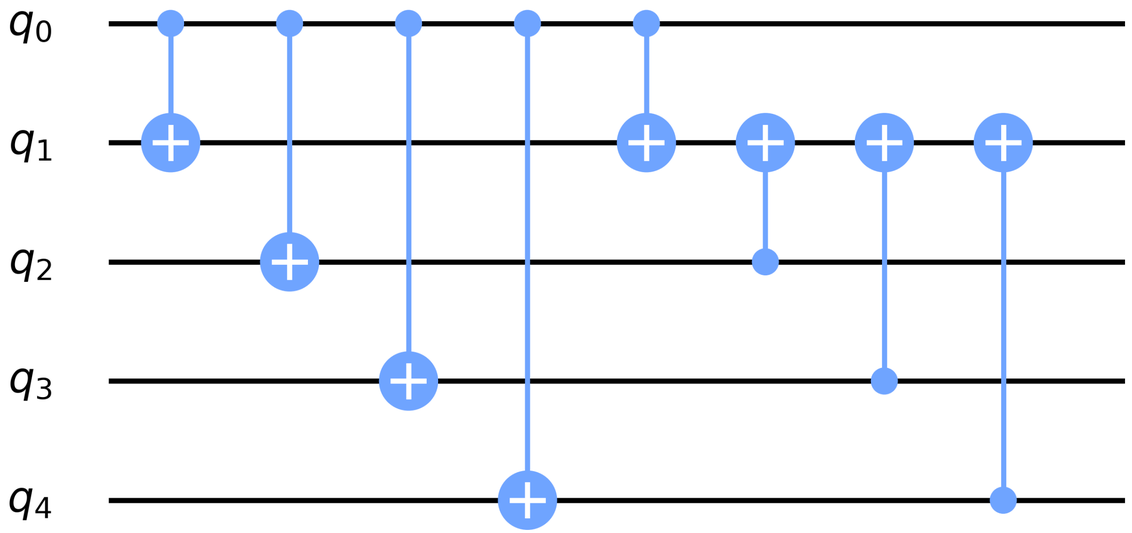
- Qubits may have limited connectivity
- For IBM's super conducting hardware this is limited to qubits that have physical connectivity.
- For multi-qubit gates this means we can only run these operations where there is connectivity between qubits.

IBM Torino 133 qubits



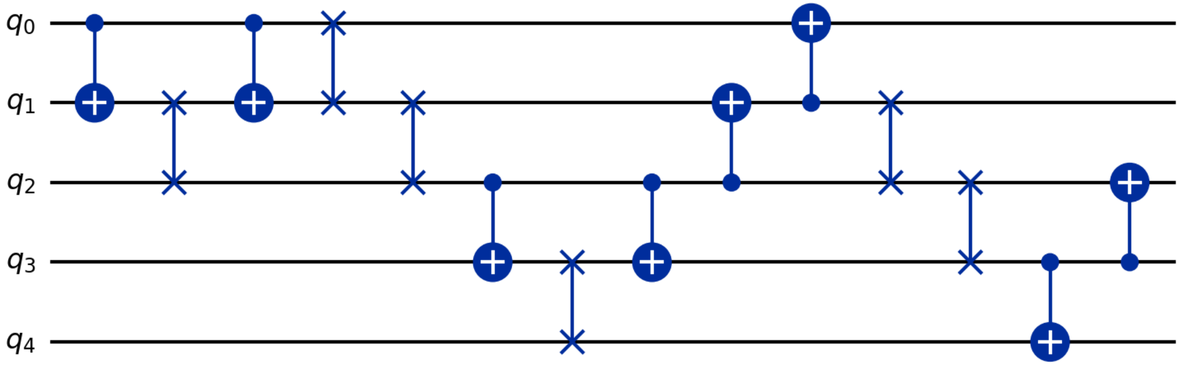
Not enough Connectivity

- Connectivity is not always sufficient to fully map the abstract circuit
- The compiler will use SWAP gates (or other techniques) to move state between qubits
- This is potentially expensive to do, so the compiler's job is to minimize this as much as it can



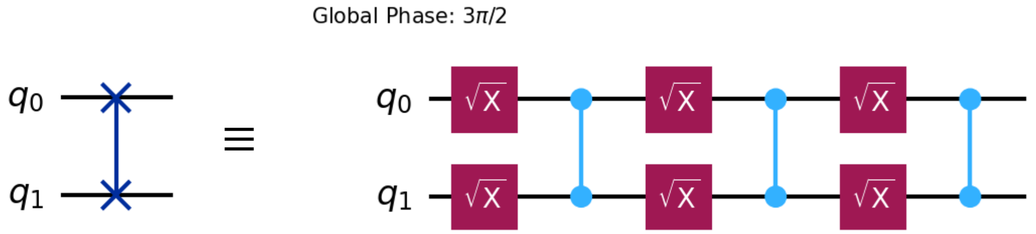
Not enough Connectivity

- Connectivity is not always sufficient to fully map the abstract circuit
- The compiler will use SWAP gates (or other techniques) to move state between qubits
- This is potentially expensive to do, so the compiler's job is to minimize this as much as it can



Not enough Connectivity

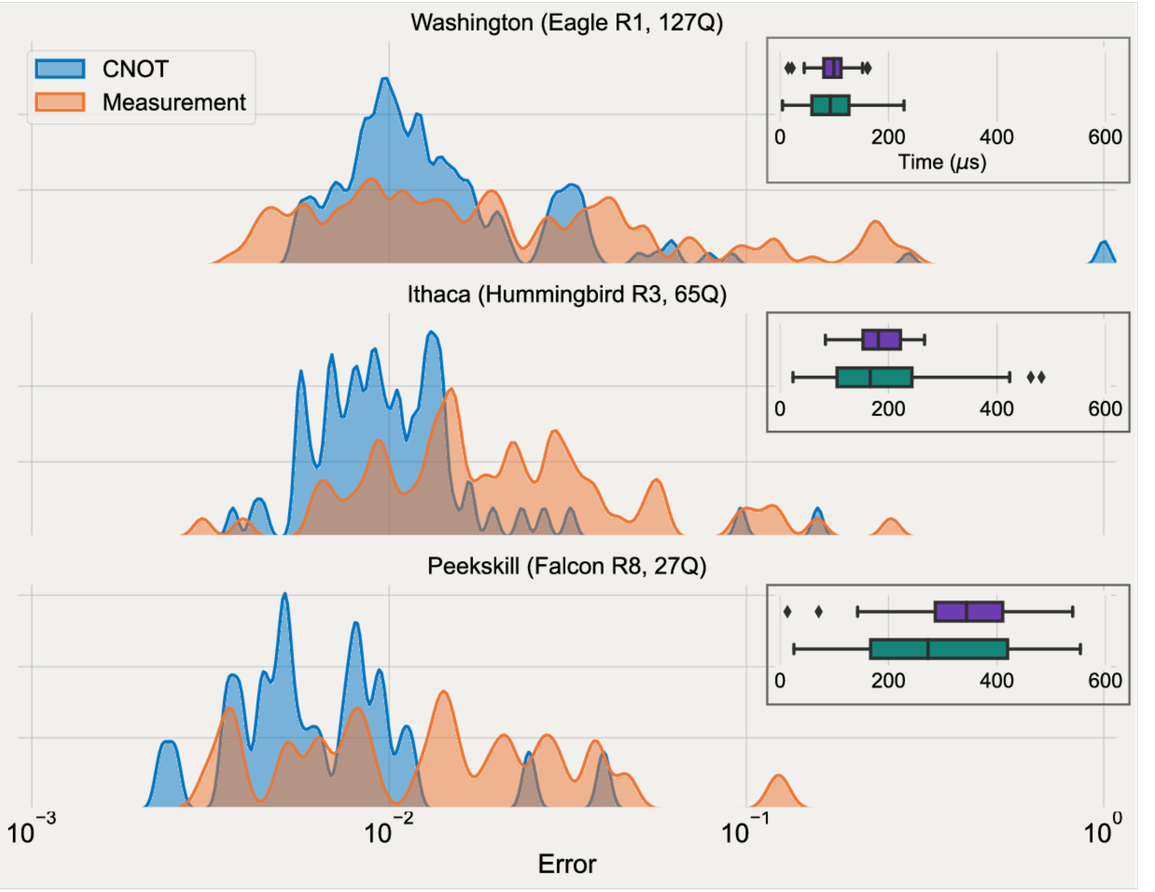
- Connectivity is not always sufficient to fully map the abstract circuit
- The compiler will use SWAP gates (or other techniques) to move state between qubits
- This is potentially expensive to do, so the compiler's job is to minimize this as much as it can



Noise

Every operation is a potential source of noise or errors

- Gate Errors
  - Single qubit Errors
  - Multiqubit Errors
- Decoherence
  - T1: Energy Relaxation, the time for a qubit at $|1\rangle$ to decay to ground state $|0\rangle$
  - T2: dephasing of a qubit in superposition state
- Readout Error



Target

- The constraints of a given compilation target is modeled by the `Target` class in Qiskit.
- The target contains a list of every instruction (operation/gate on qubits), the properties associated with those instructions. Typically error rate, duration, and a pulse calibration.
- Additional properties about qubits and other device characteristics use for compilation are contained within the target.

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.Target

```
print(backend.target)
```

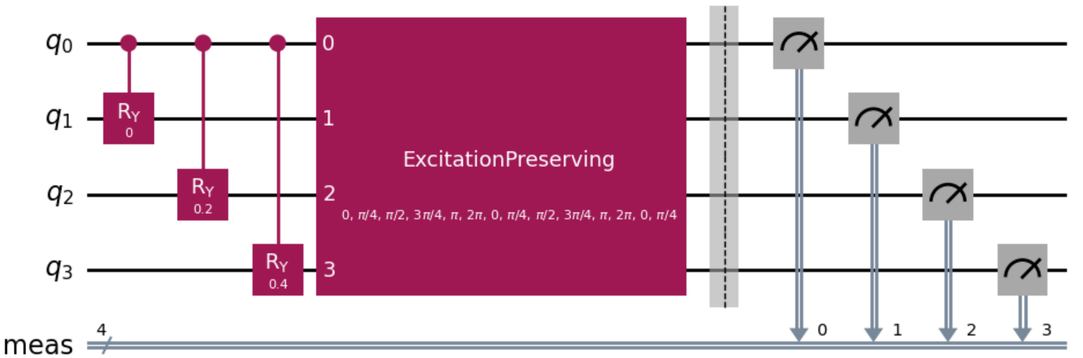
```
Target
Number of qubits: 5
Instructions:
        cx
                (0, 1):
                        Duration: 4.124444444444444e-07 sec.
                        Error Rate: 0.011112942844963669
                        With pulse schedule calibration
                (1, 0):
                        Duration: 3.7688888888888884e-07 sec.
                        Error Rate: 0.011112942844963669
                        With pulse schedule calibration
                (1, 2):
                        Duration: 2.702222222222222e-07 sec.
                        Error Rate: 0.007959389035724018
                        With pulse schedule calibration
                (2, 1):
                        Duration: 3.0577777777777775e-07 sec.
                        Error Rate: 0.007959389035724018
                        With pulse schedule calibration
                (2, 3):
                        Duration: 2.8444444444444443e-07 sec.
                        Error Rate: 0.025418806013513956
                        With pulse schedule calibration
                (3, 2):
                        Duration: 3.2e-07 sec.
                        Error Rate: 0.025418806013513956
                        With pulse schedule calibration
                (3, 4):
                        Duration: 4.195555555555555e-07 sec.
                        Error Rate: 0.018081475298807576
                        With pulse schedule calibration
                (4, 3):
                        Duration: 3.84e-07 sec.
                        Error Rate: 0.018081475298807576
                        With pulse schedule calibration
        sx
                (0,):
                        Duration: 3.5555555555555554e-08 sec.
                        Error Rate: 0.00037753800551696055
                        With pulse schedule calibration
```

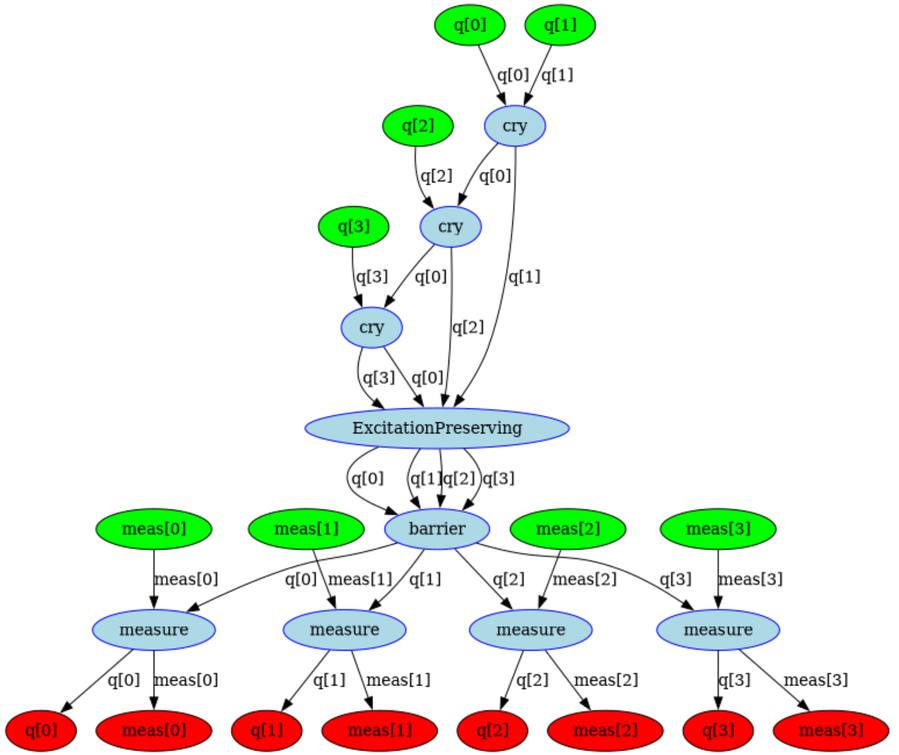
Transpiler Internals

DAG Representation

Inside the transpiler Qiskit represents the circuit as a Directed Acyclic Graph (DAG). The graph tracks the flow of data from the start of the circuit with input nodes (green in the visualization) through operations (blue in the visualization) to output nodes that represent the end of the circuit (red in the visualization). The edges are used to track bits, both quantum and classical, in addition to other classical variables.

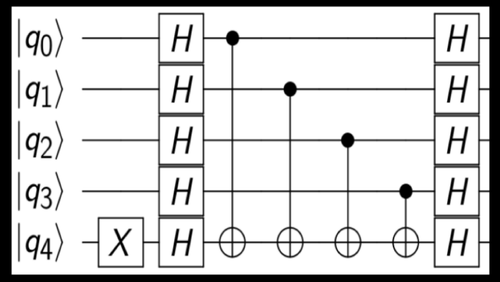
This representation makes it simpler to do analysis of the circuit and see dependency between operations. It also makes doing in place mutation and substitution of operations simpler.





Transpiler Architecture

The complexity of scheduling is solely in the pass manager, to keep passes simple



Pass 1 — Unroll to basis

Pass 2 — Cancel redundant gates

Pass 3 — Map to device by swapping

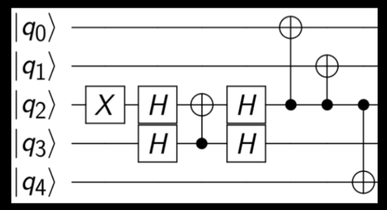
Pass 4 — Cancel redundant gates

Each pass does one small, well-defined task.

Pass Manager

Global Property Set

- Depth
- Commuting Sets
- Layout
- Fixed Point Condition



The pass manager maintains a global context about the circuit as it runs through the pipleine

Pass Manager Stages

Qiskit's compilation pipeline is broken up into discrete stages. These stages execute linearly, and each perform a logical transformation as we progressively transform the circuit towards an output that's optimized for execution on a given target. By default, Qiskit uses 6 stages for its preset pass managers but if you're constructing a pass manager manually you can create your own stages.

https://docs.quantum.ibm.com/transpile/transpiler-stages

Pass Manager Stages

- Init stage: Initial passes that operate on abstract circuit. This stage is responsible for any logical optimizations and decomposing larger gates into terms of 1 and 2 qubit gates to enable the next stage.
- Layout stage: The stage responsible for mapping circuit qubits to qubits in the target.
- Routing stage: This stage makes up for a lack of connectivity and inserts Swap gates or other operations to move state between qubits.
- Translation stage: This stage is responsible for transforming all circuit operations into ones supported by the target.
- Optimization stage: This stage is responsible for performing any optimizations, especially after earlier stages likely inserted extra operations.
- Scheduling stage: The final optional stage is for any scheduling of the circuit to account for the timing of operations in the circuit.

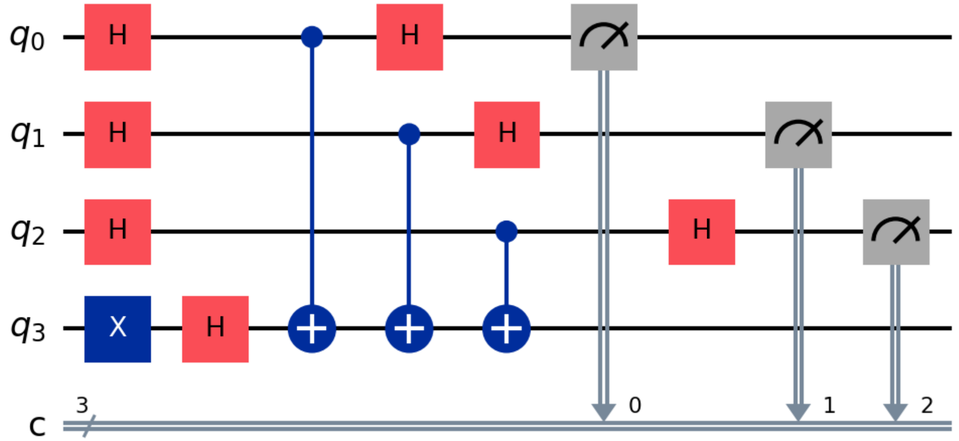
Pass Manager Pluggability

The pass managers are pluggable and extensible so that you can replace the implementation of just a stage on the fly. You can build a custom PassManager object and substitute it for a stage or modify a stage in place. This lets you reuse a prebuilt pipeline and customize a small component

You can install/publish external Python packages that advertise

https://docs.quantum.ibm.com/transpile/transpiler-plugins

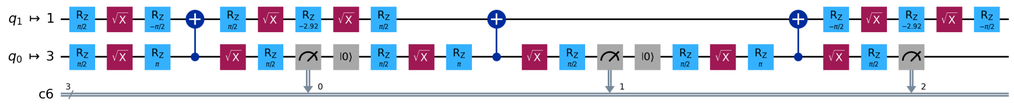
```
pip install qiskit-qubit-reuse
```




```
transpile(qc, backend, init_method="qubit_reuse")
```

or

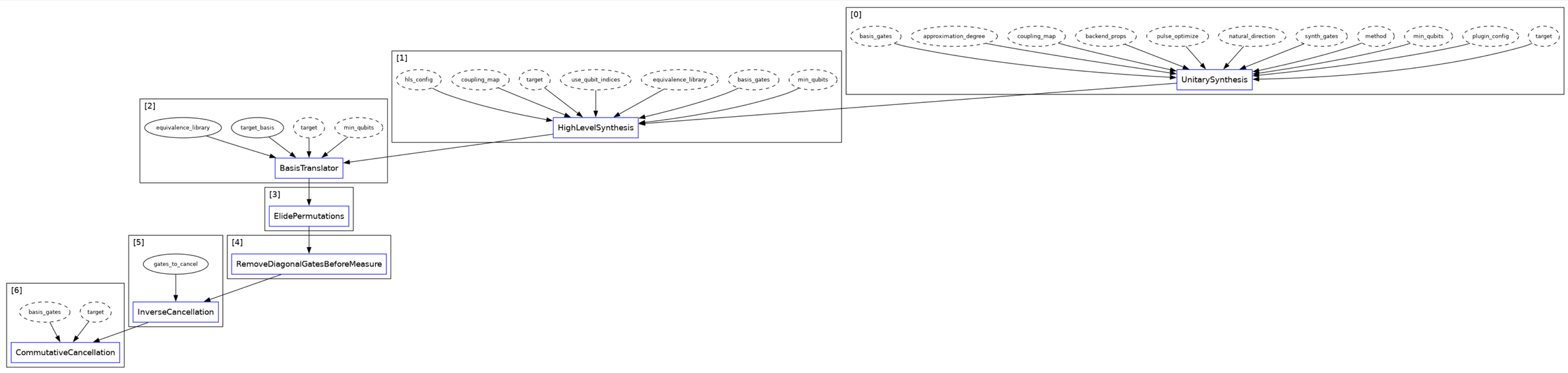
```
pm = generate_preset_pass_manager(2, backend, init_method="qubit_reuse")
pm.run(qc)
```



Preset Pass Managers

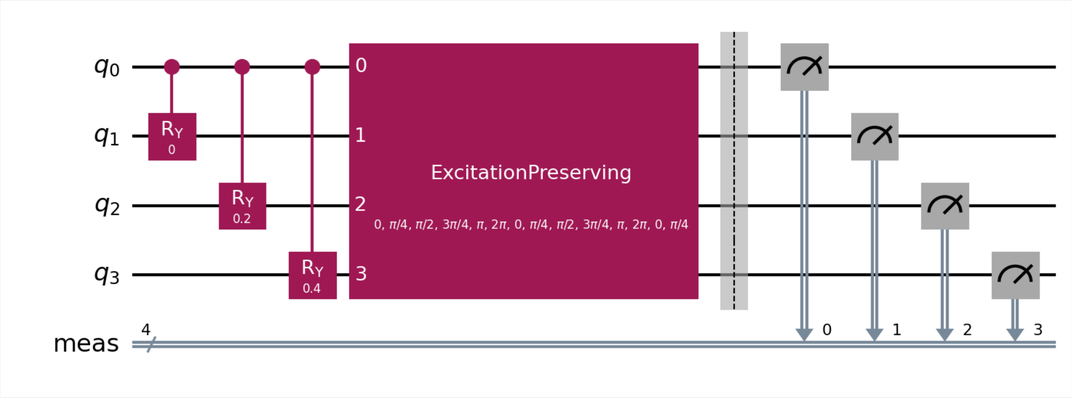
Init stage

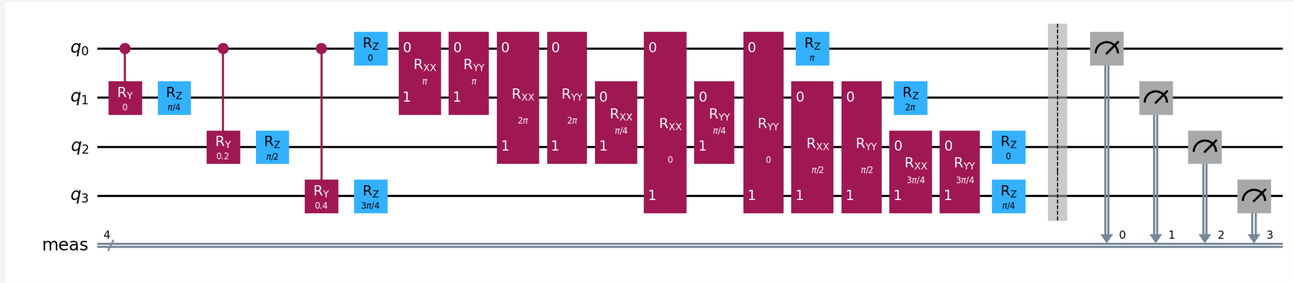
- Stage is broken into two phases:
  - Decompose larger (>=3 qubit) gates to 1 and 2 qubit gates
  - Logical optimizations
- At the end of the stage, we are able to run layout (which models the circuit as a graph)



Init stage

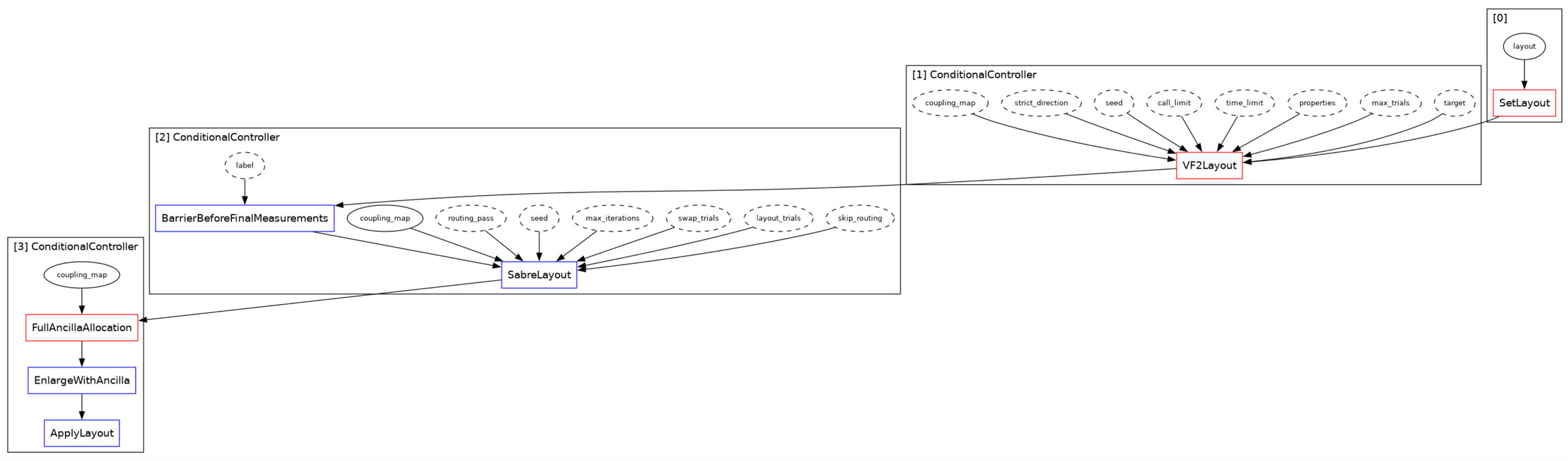
- Stage is broken into two phases:
  - Decompose larger (>=3 qubit) gates to 1 and 2 qubit gates
  - Logical optimizations
- At the end of the stage, we are able to run layout (which models the circuit as a graph)



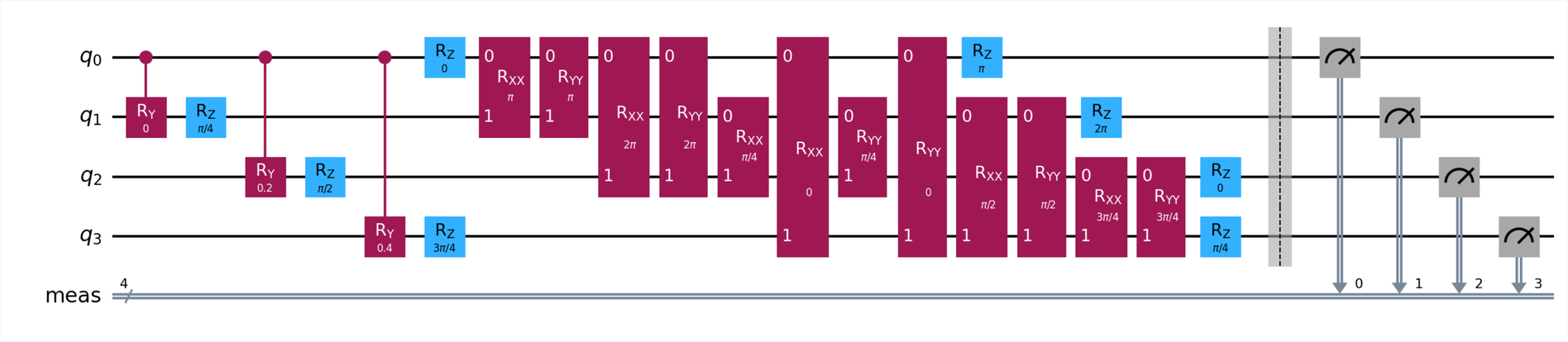


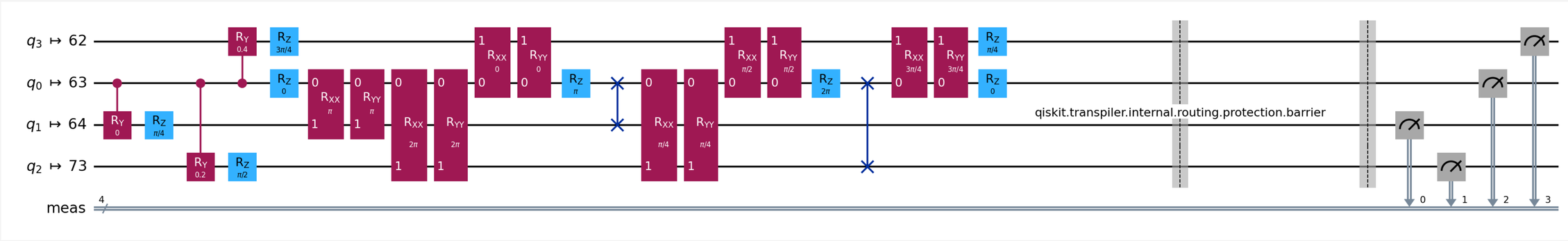
Layout stage

- This stage takes the virtual qubits from abstract circuit and maps them to a qubit as defined in the target
- Layout is of critical importance because not all qubits perform the same.
- This stage typically tries multiple techniques to find the best layout. In Qiskit we use two passes by default: `VF2Layout` and `Sabre`
- A poor layout can induce more Swaps because of the connectivity



Layout stage





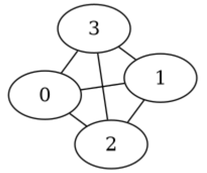
VF2Layout

This pass creates a graph from the 2 qubit interactions in the circuit and tries to find an isomorphic subgraph of the connectivity graph of the target. If a subgraph is found than that indicates a layout which does not require swaps can be used. If a layout is found, then the pass will search for additional layouts and pick the one with the lowest predicted error rate.

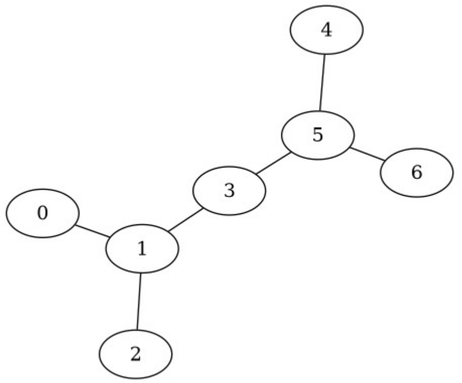
It leverages the rustworkx's library's VF2 algorithm implementation to efficiently search for an isomorphism.

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.VF2Layout

Interaction Graph



Connectivity Graph



Sabre

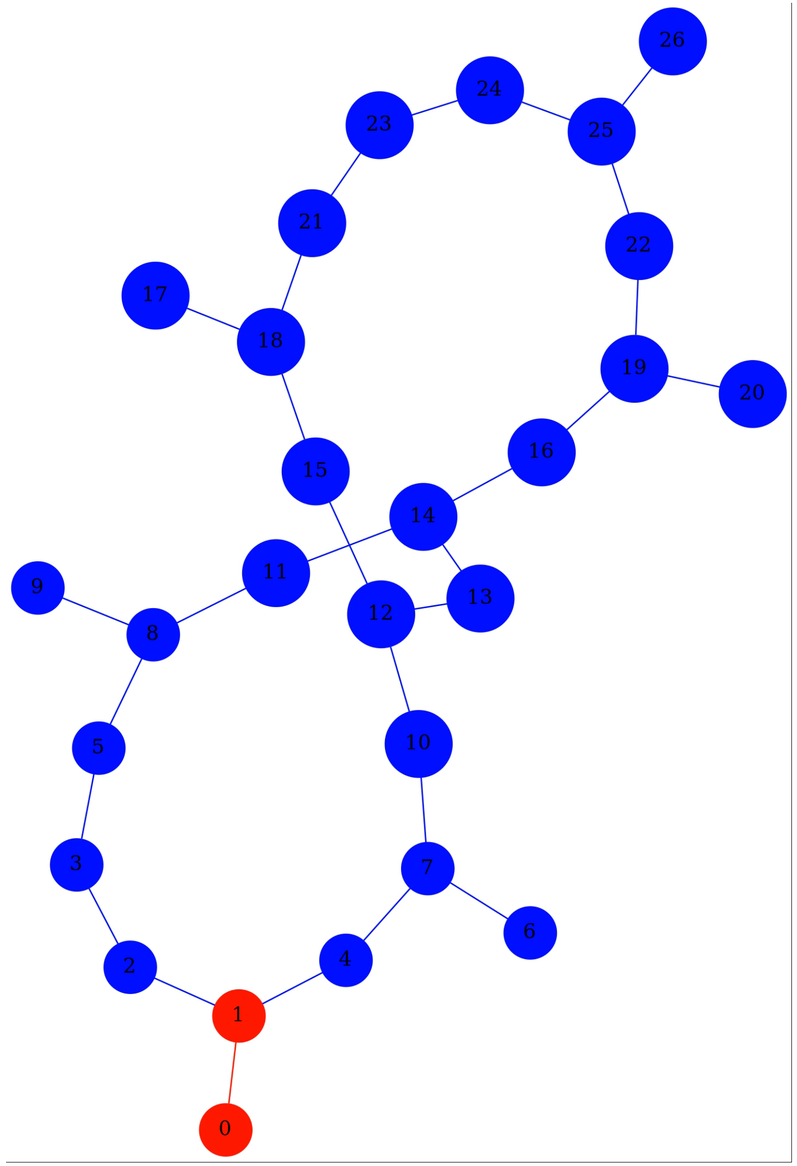
If `VF2Layout` does not find a layout then we run `SabreLayout` pass. This pass is based on the SABRE (SWAP-based BidiREctional) algorithm originally detailed in:

https://arxiv.org/pdf/1809.02573.pdf.

The SABRE algorithm works by starting with an initial random guess, then running a routing algorithm on it to insert swap gates and instead of inserting a swap it swaps the qubits in the layout. It fully "routes" the circuit then reverses the edge direction in the dag and repeats. This is performed multiple times, doing this will minimize the number of swaps needed.

The implementation in Qiskit has significantly changed and improved on the algorithm from the original paper.

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.SabreLayout



Sabre

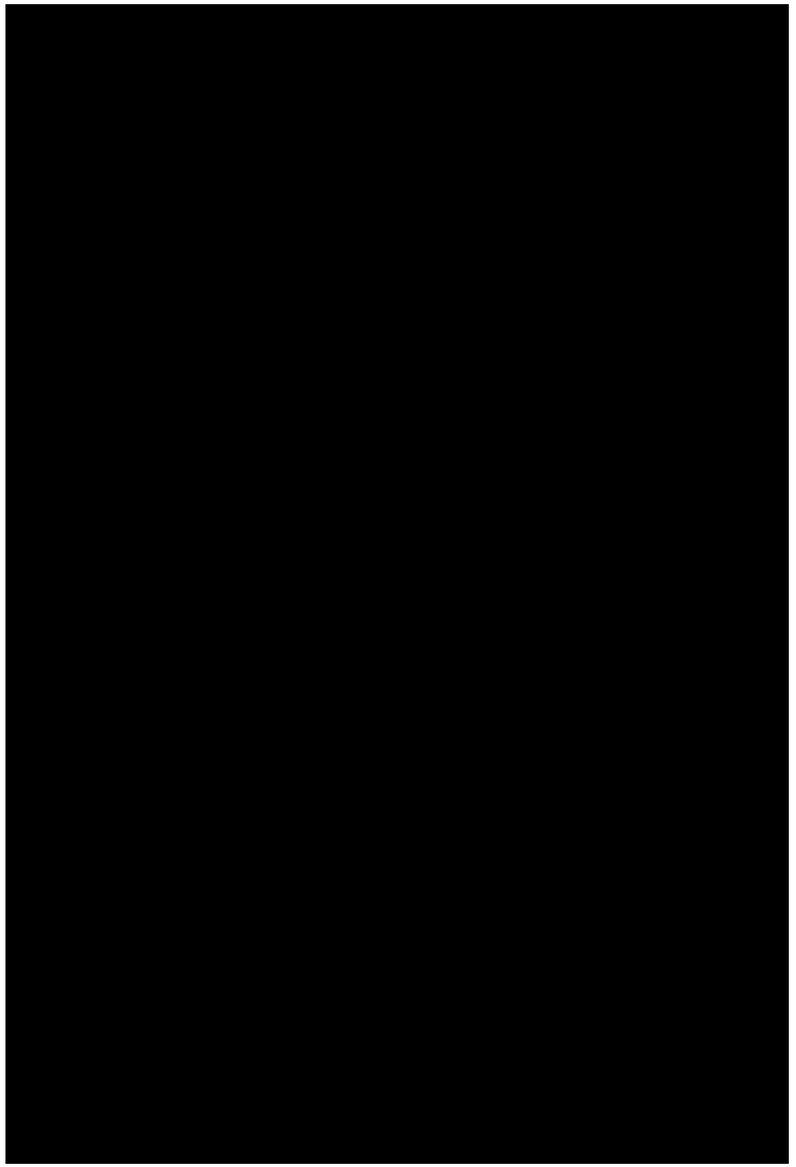
If `VF2Layout` does not find a layout then we run `SabreLayout` pass. This pass is based on the SABRE (SWAP-based BidiREctional) algorithm originally detailed in:

https://arxiv.org/pdf/1809.02573.pdf.

The SABRE algorithm works by starting with an initial random guess, then running a routing algorithm on it to insert swap gates and instead of inserting a swap it swaps the qubits in the layout. It fully "routes" the circuit then reverses the edge direction in the dag and repeats. This is performed multiple times, doing this will minimize the number of swaps needed.

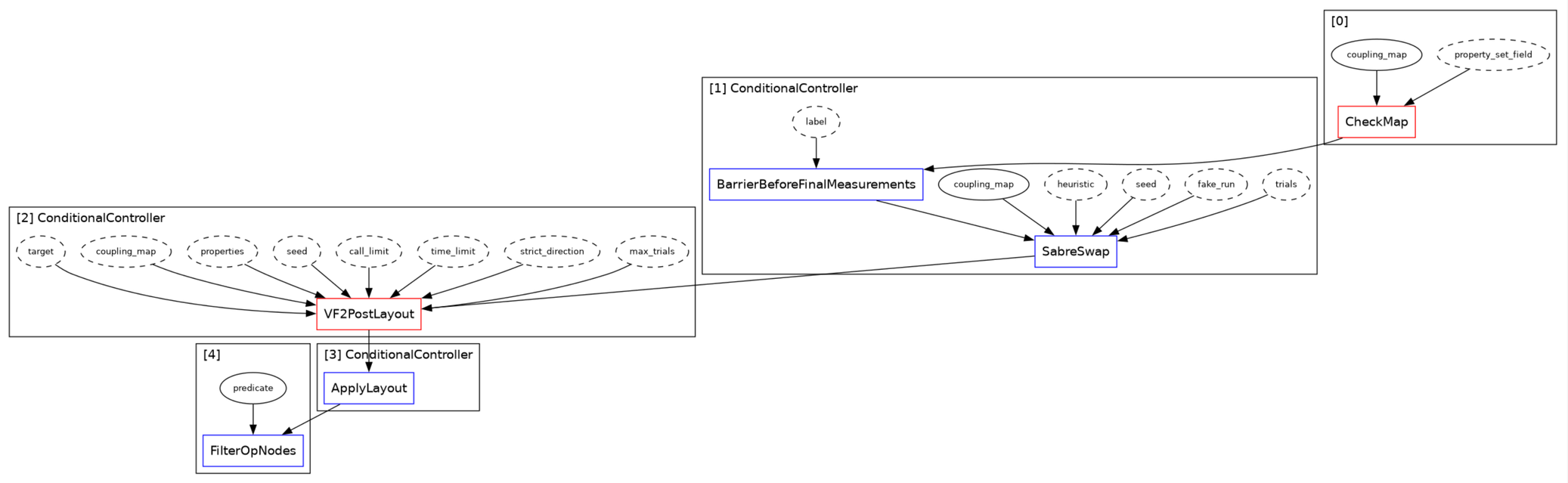
The implementation in Qiskit has significantly changed and improved on the algorithm from the original paper.

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.SabreLayout



Routing stage

- This stage takes the circuit with a layout set and inserts Swap gates where necessary.
- The default pass for this is `SabreSwap`.
- If running with defaults however this pass is skipped for efficiency, and it's run as part of `SabreLayout` in the Layout stage.
- We re-run layout with `VF2PostLayout` after swap insertion to search for a better layout



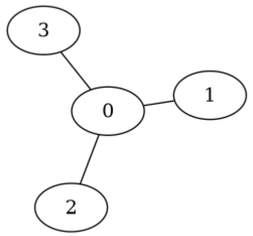
VF2PostLayout

Just as with VF2Layout this pass creates a graph from the 2 qubit interactions in the circuit and tries to find an isomorphic subgraph of the connectivity graph of the target. The difference is by running after routing we know that there is at least one isomorphic subgraph. This pass then searches for other layouts and picks the one found with the lowest predicted error rate.

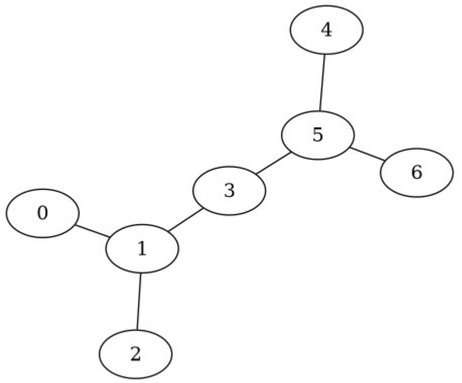
https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.010327

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.VF2PostLayout

Interaction Graph



Connectivity Graph

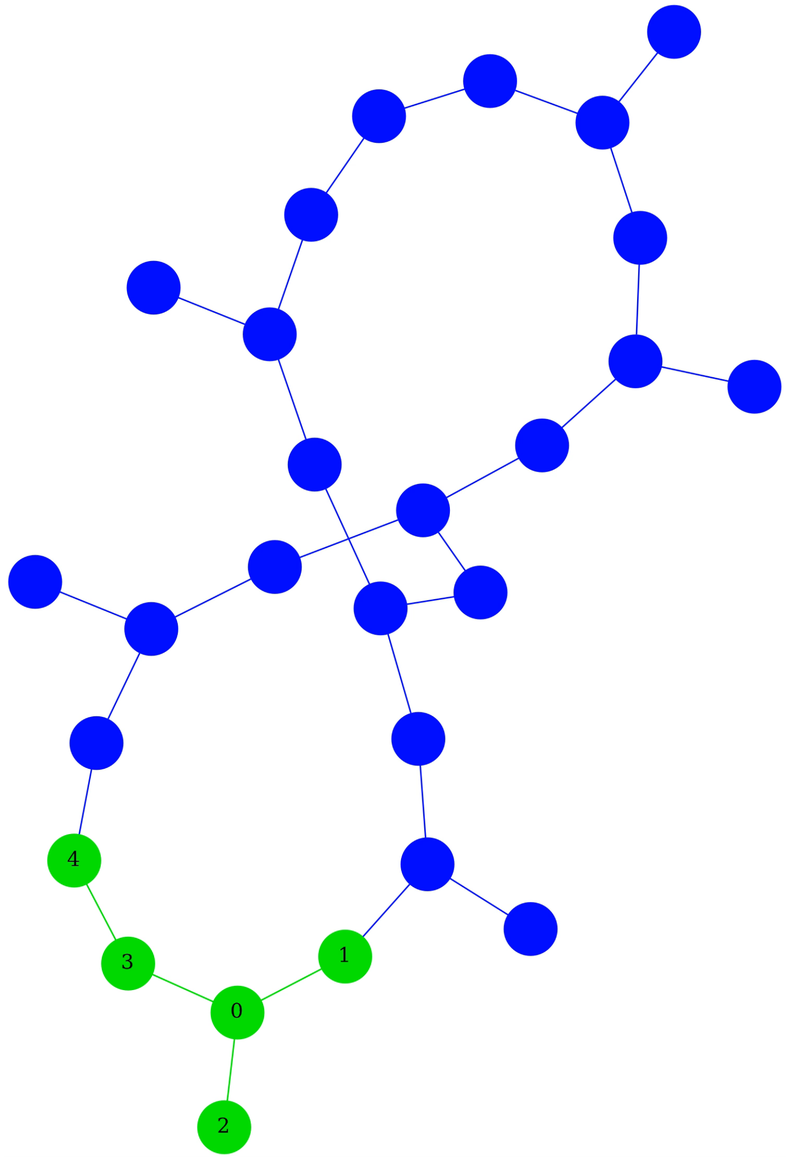


VF2PostLayout

Just as with VF2Layout this pass creates a graph from the 2 qubit interactions in the circuit and tries to find an isomorphic subgraph of the connectivity graph of the target. The difference is by running after routing we know that there is at least one isomorphic subgraph. This pass then searches for other layouts and picks the one found with the lowest predicted error rate.

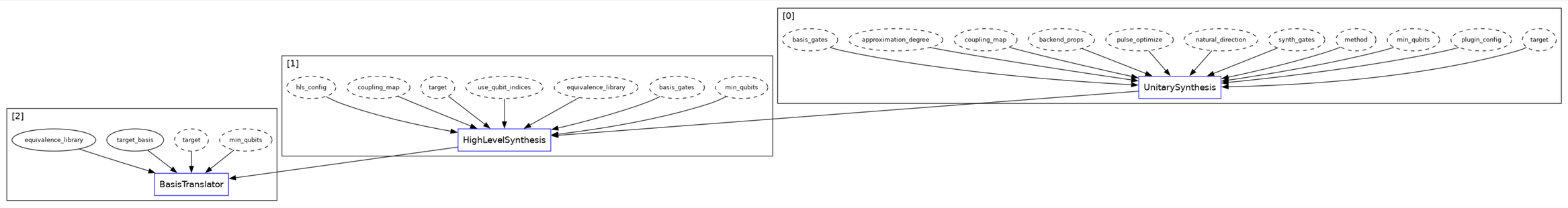
https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.010327

https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.VF2PostLayout



Translation stage

- This stage takes the routed circuit which meets the connectivity constraints of the target and is responsible for making sure that all the operations are supported on the target
- This stage runs the same passes we used in the first phase of the init stage but for all operations instead of just those with >= 3 qubits

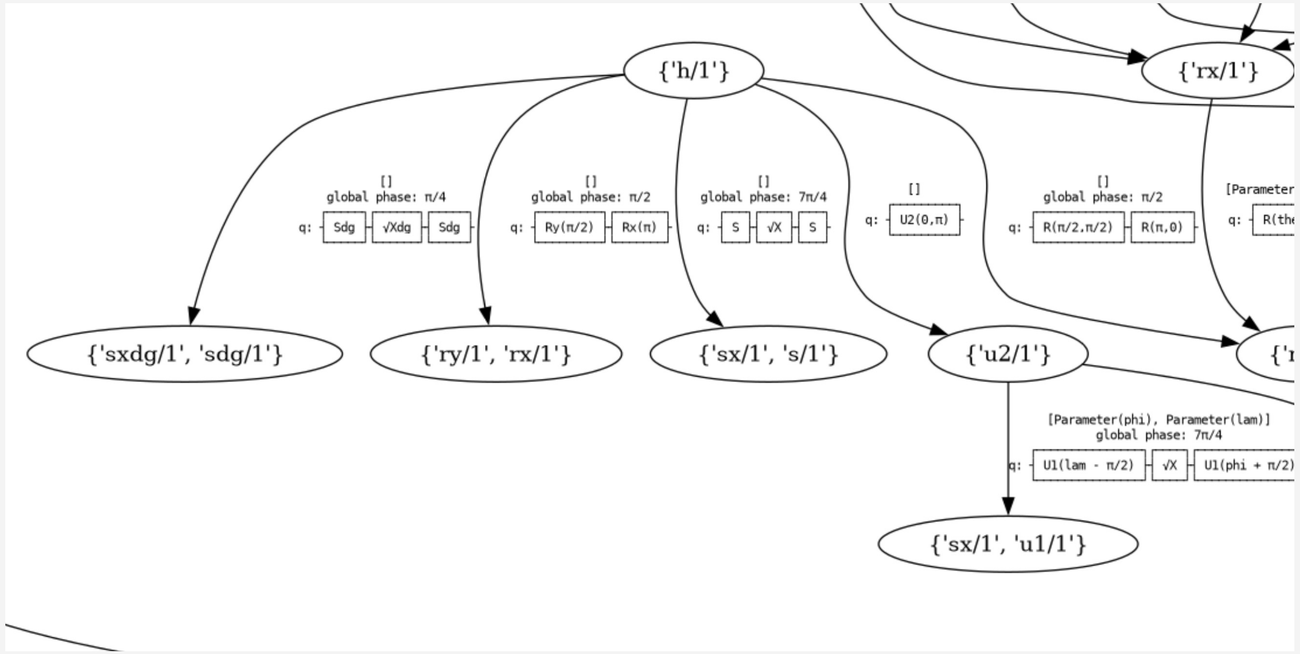


Basis Translator

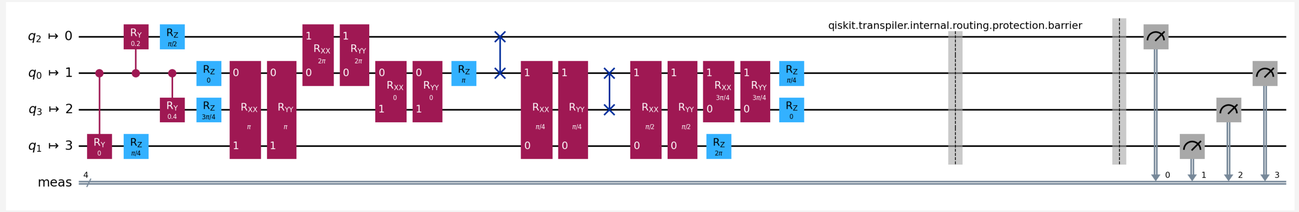
- The basis translator uses an equivalence library that builds a graph of equivalent circuits to a gate
- Dijkstra's algorithm is used to find a path from a given gate to gates in the target
- The gates in the circuits are recursively substituted along the path to build an equivalent circuit that conforms to the target
- Each gate is then replaced in the DAG by that equivalent circuit

Basis Translator

- The basis translator uses an equivalence library that builds a graph of equivalent circuits to a gate
- Dijkstra's algorithm is used to find a path from a given gate to gates in the target
- The gates in the circuits are recursively substituted along the path to build an equivalent circuit that conforms to the target
- Each gate is then replaced in the DAG by that equivalent circuit

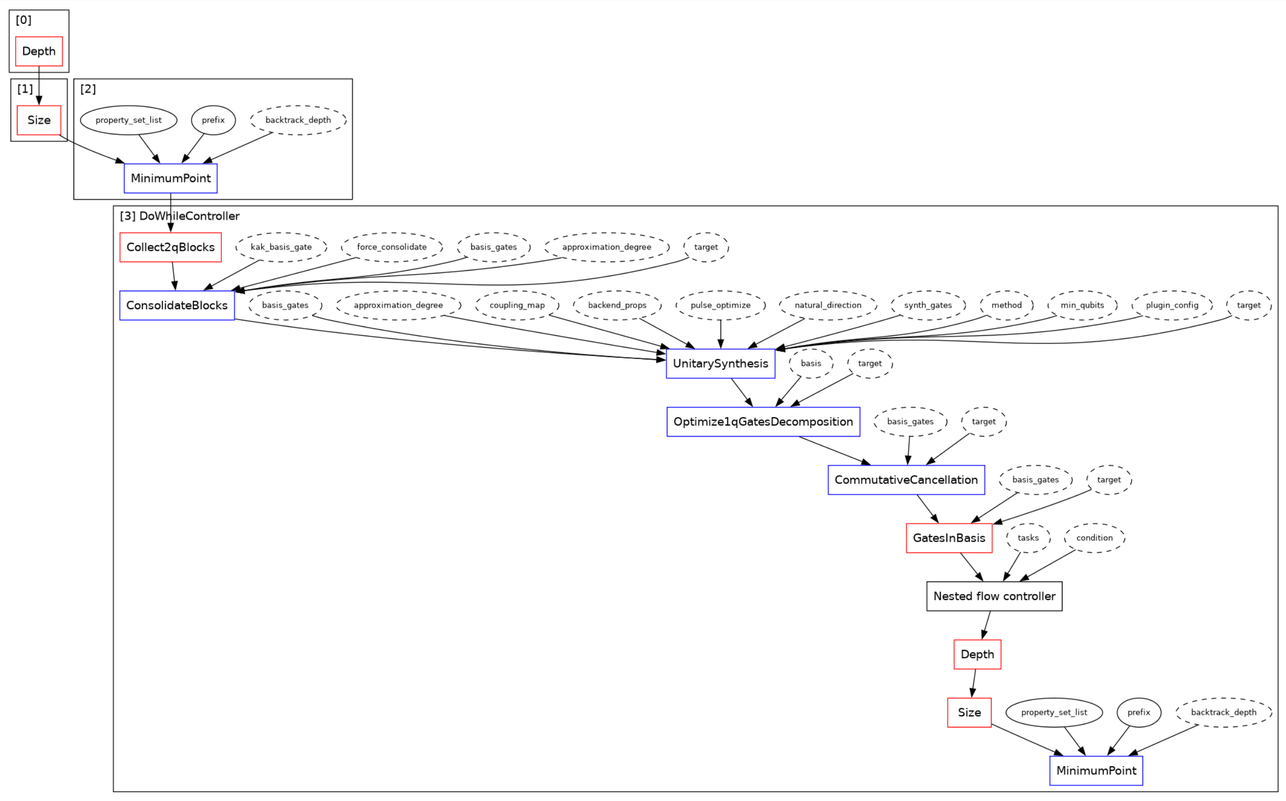


Basis Translator



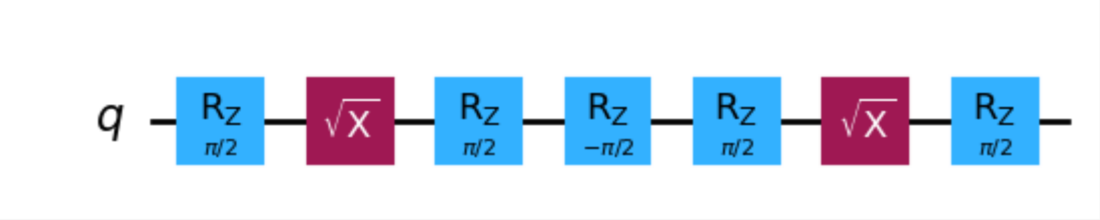
Optimization stage

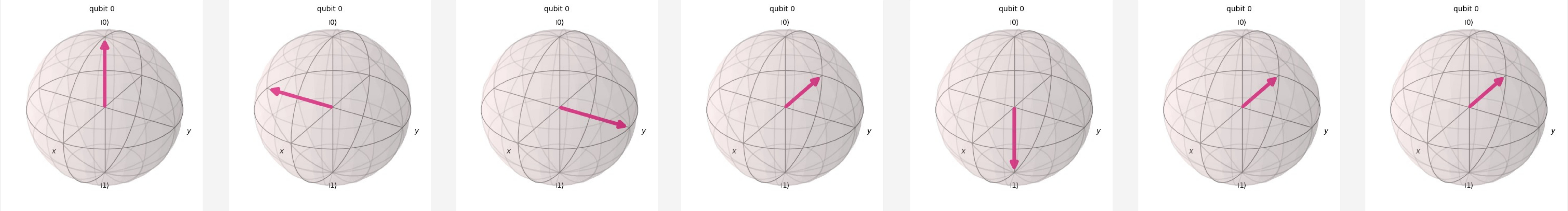
- This stage takes the mapped circuit and optimizes it to reduce unnecessary operations
- This stage runs in a loop to repeatedly run the optimization passes until a minimum point in depth and size is found



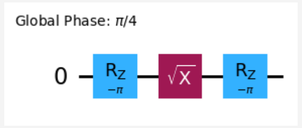
1 Qubit Unitary Peephole Optimization

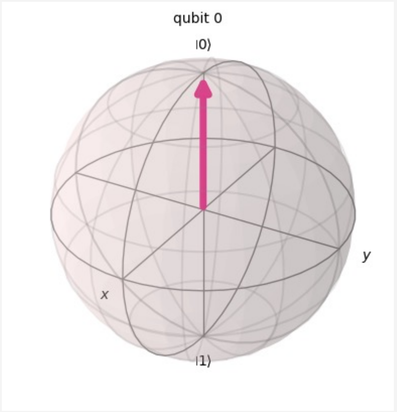
– This pass looks for sequences of single qubit gates

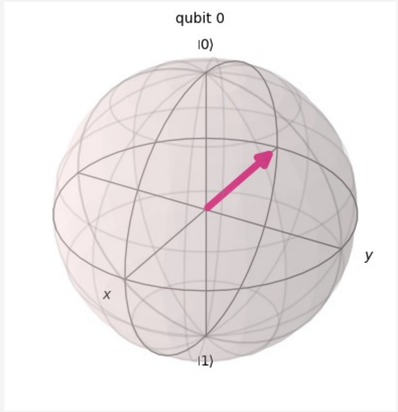




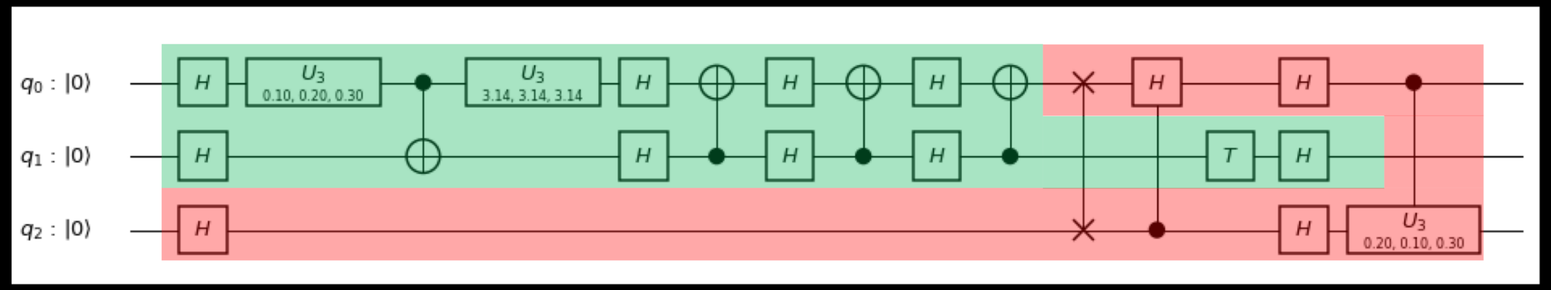
1 Qubit Unitary Peephole Optimization



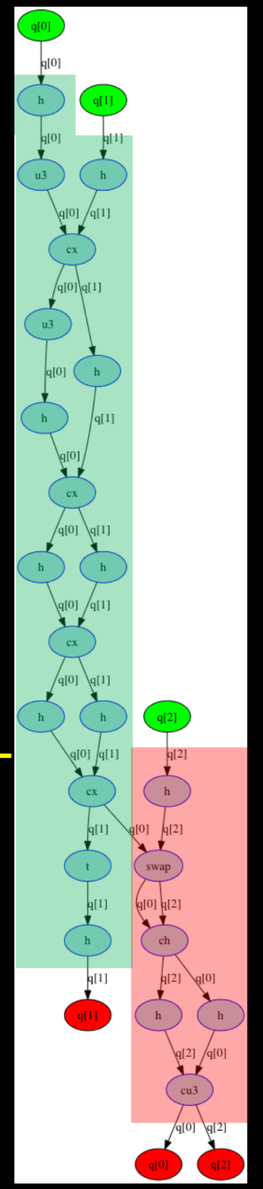




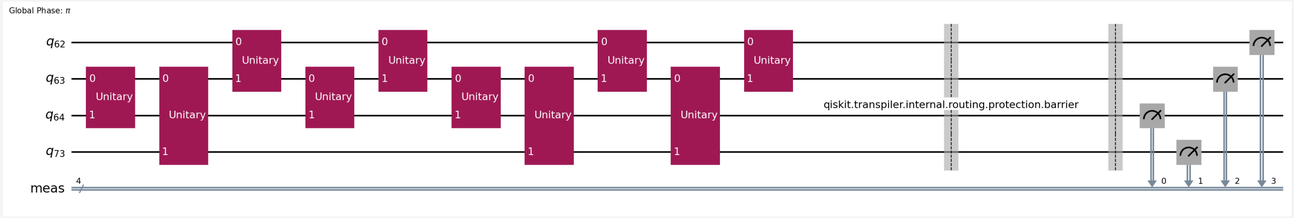
2-qubit Block Collection

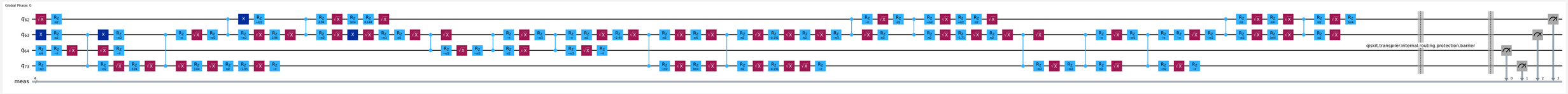


Efficient graph traversal on the DAG.
For each CNOT, collect ancestors and predecessors until a branch.



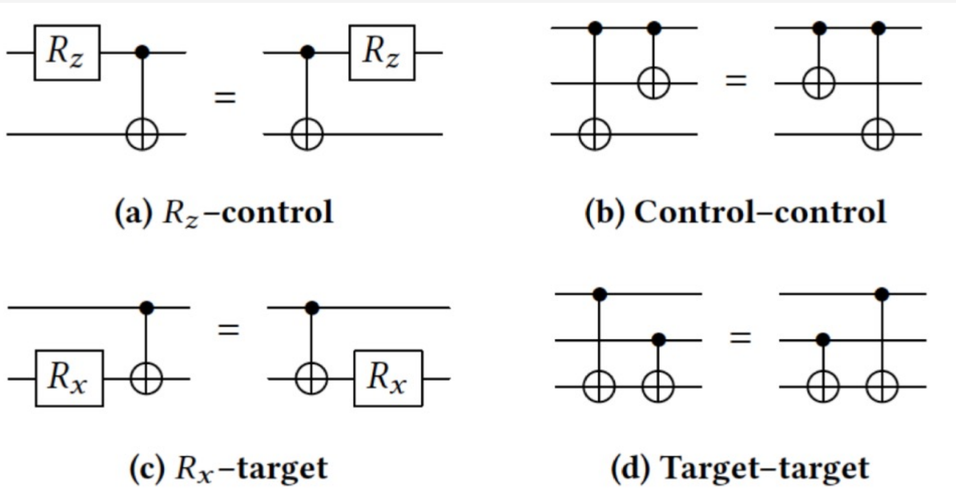
2 Qubit Unitary Peephole Optimization



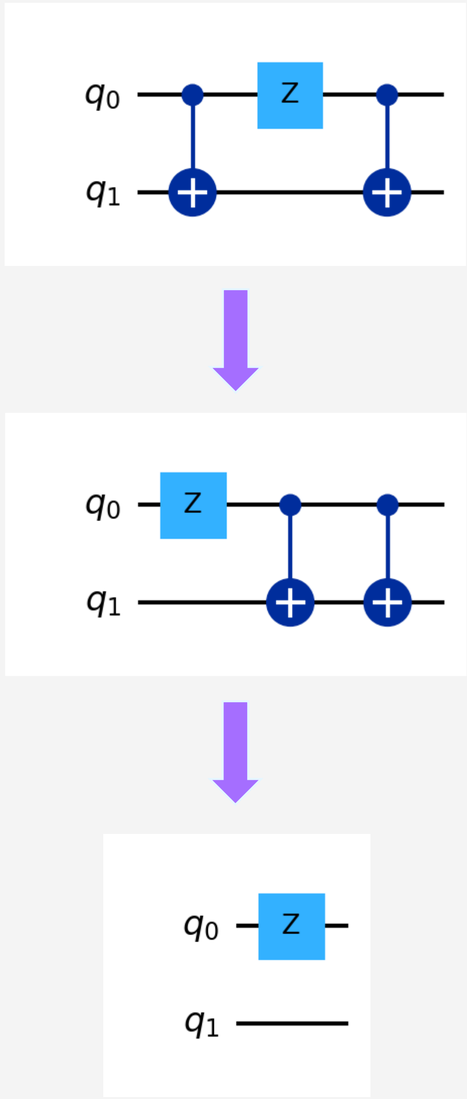


Commutative Cancellation

Examples of commuting gates:

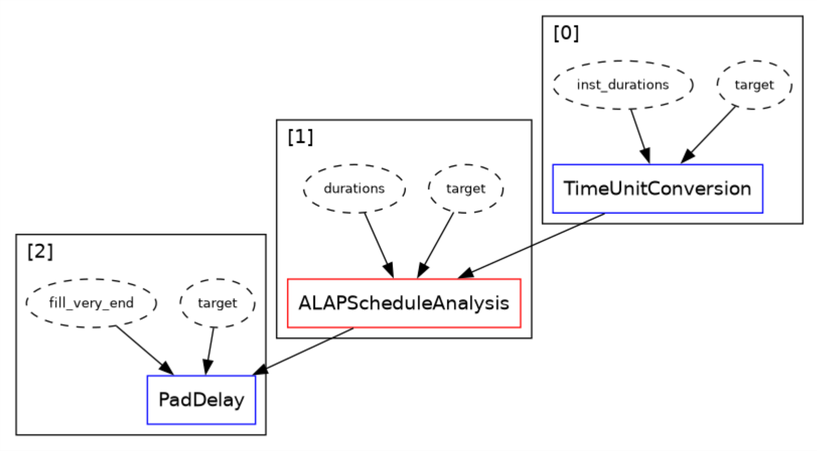


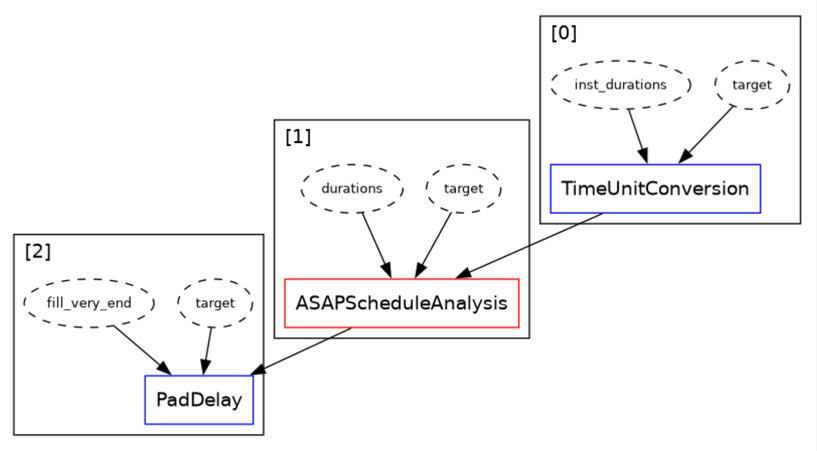
Two consecutive gates are commutative ⇔
They can be exchanged without changing what they compute.



Scheduling stage

- This stage takes the mapped and optimized circuit and analyzes the runtime of the operations in the circuit and identify idle periods on each qubit
- It will then insert explicit operations to account for all the time in the circuit





Scheduling stage

- This stage takes the mapped and optimized circuit and analyzes the runtime of the operations in the circuit and identify idle periods on each qubit
- It will then insert explicit operations to account for all the time in the circuit