COE 530
Quantum Computers and Architecture

Lecture 13

Qubit Mapping and Reuse

References:

Ch 6 Quantum Computing Systems

Li, Gushu, Yufei Ding, and Yuan Xie. "Tackling the qubit mapping problem for NISQ-era quantum devices." In Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, pp. 1001-1014. 2019.

Qubit Mapping

- Quantum circuits assume all-to-all connectivity between qubits
- However, real NISQ hardware has limited physical connectivity
- Two-qubit gates (e.g., CNOT) can only act on physically adjacent qubits

- **Objective:** devise an initial mapping of logical qubits to physical qubits and insert proper SWAP gates to ensure that the result circuit satisfies the constraints imposed by two-qubit gate interactions

- A good qubit mapping strives to minimize the number of SWAP operations

- **Why It Matters for NISQ**
  - Every extra gate increases circuit depth and error accumulation
  - Poor mapping can increase gate count by 2-10x on real devices

Connectivity Constraints

- QPU Topologies
- Grid / Square Lattice
  - Google Sycamore; ~4 neighbors
- Heavy-Hex
  - IBM Eagle/Heron; degree-3 (sparse)
- Trapped-Ion
  - Dynamically reconfigurable zones
  - All-to-all within zone; shuttle between
- Neutral Atom
  - Blockade radius defines connectivity
  - Atoms can be physically rearranged

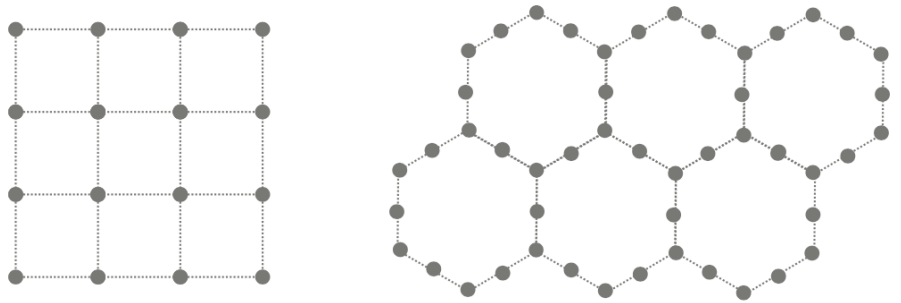


Figure 6.10: Different connectivity graph for superconducting devices. **Left:** 2D square lattice. **Right:** 2D (heavy) hexagonal lattice. The choice of connectivity graphs is typically based on hardware constraints such as wiring bandwidth and noises of circuit components.

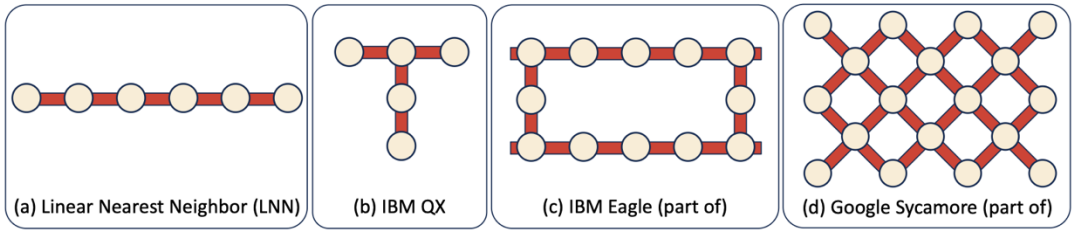


Fig. 4. Common structures of superconductivity devices and their corresponding connectivity graphs.

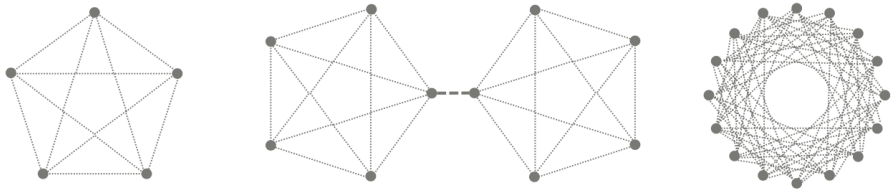


Figure 6.11: Different connectivity graph for trapped ion devices. **Left:** Complete (Clique) connectivity for small number of ions in one trap. **Center:** Weakly connected cliques for multiple traps. **Right:** A long chain of ions in one trap in a tape-like architecture.

Device connectivity graph (device topology)

- Qubit mapping can be analyzed using the device connectivity graph

- A graph $G = (V, E)$ describe physical connectivity of qubits
  - V = set of physical qubits, each node (vertex) is a qubit
  - E = pairs that can execute 2-qubit gates
  - Two qubits can interact only if they are adjacent in the graph
  - Qubits can be moved around using SWAP gates
  - A $CNOT(q_i, q_j)$ can only execute if (q_i, q_j) is an edge in E
  - If not adjacent → must route qubit states via SWAP operations
  - Sparser topologies → more SWAPs needed

Program interaction graph

- graph $G = (V, E)$, where V is a set of logical qubits present in the computation, and E is a set of two-qubit interaction gates contained in the program (e.g., CNOT gates).
- By analyzing this graph, we can perform an optimized mapping, which assigns a physical location for each logical qubit $q \in V$

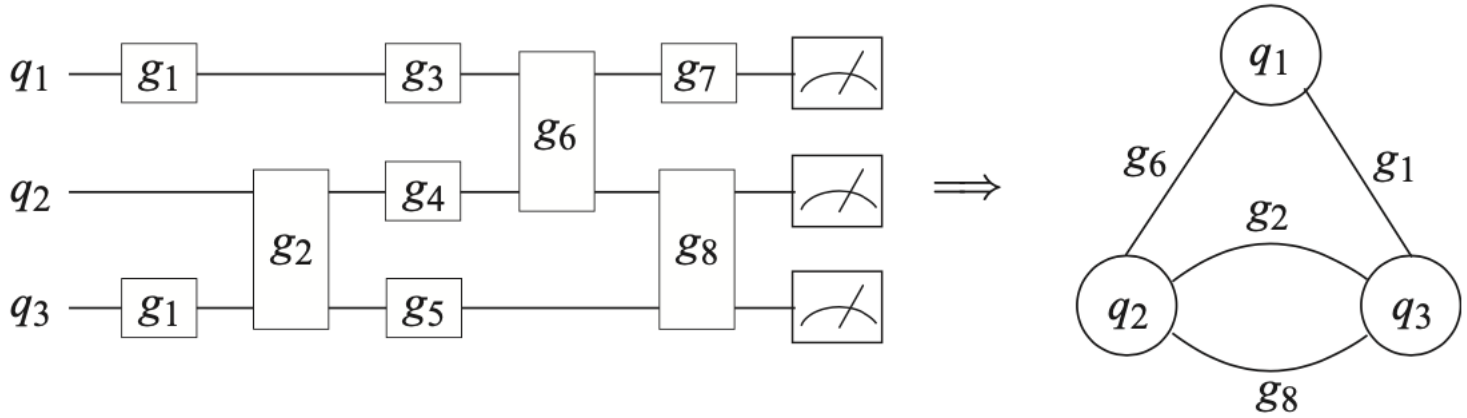


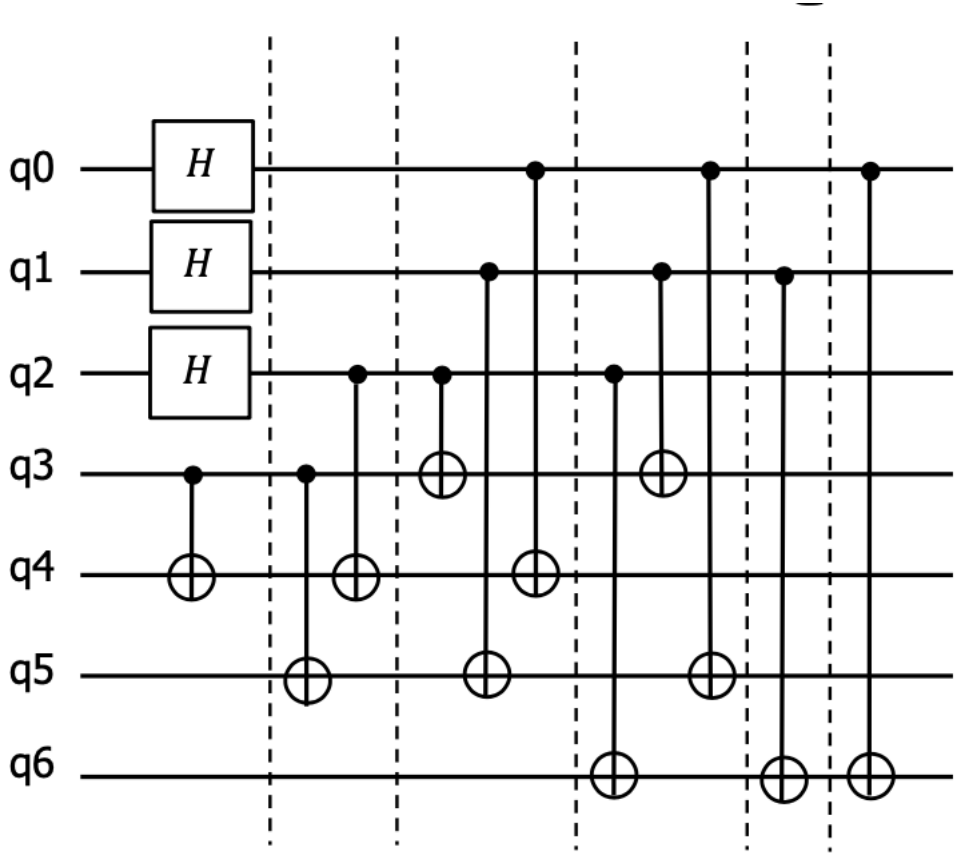
Figure 6.12: The program interaction graph for an example quantum circuit.

Difference between the two graphs

	Device Connectivity Graph	Program Interaction Graph
Represents	Physical hardware topology	Circuit's qubit-pair interactions
Nodes	Physical qubits	Logical qubits
Edges	Directly executable 2Q gates	Required 2Q gate interactions
Nature	Fixed per device	Varies per program
Typically	Sparse (nearest-neighbor)	Can be dense

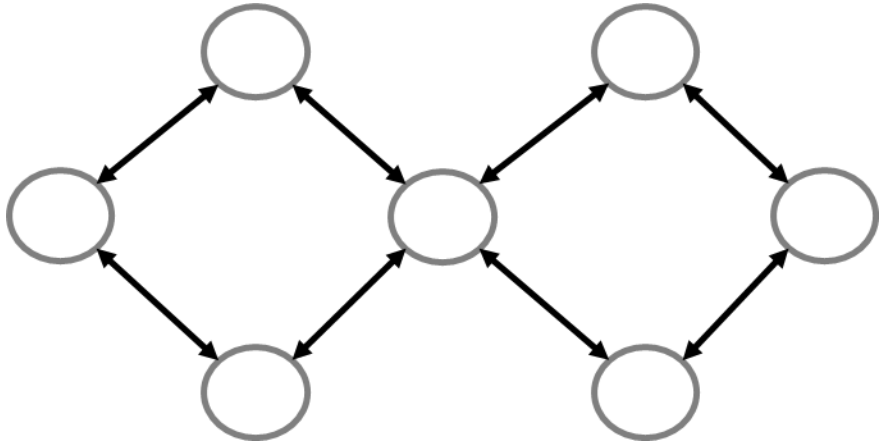
Motivating Example

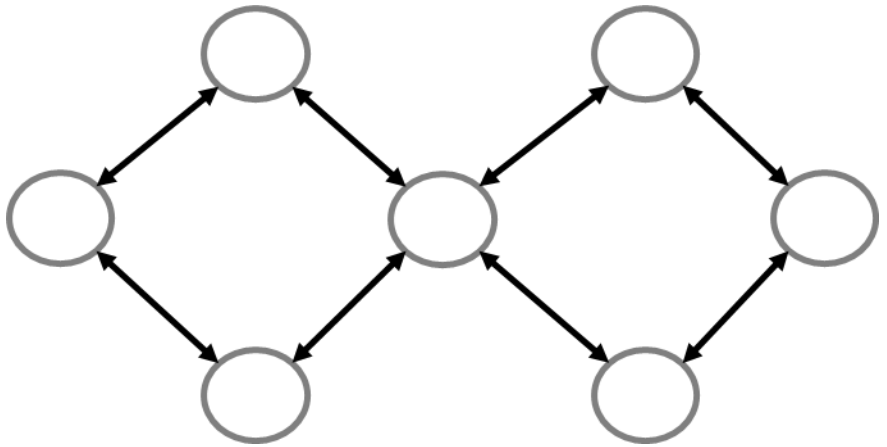
- Before executing the circuits, each logical qubit needs to be assigned to a physical qubits




```
h q0
h q1
h q2
cnot q3,q4
cnot q3,q5
cnot q2,q3
cnot q2,q4
cnot q2,q6
cnot q1,q3
cnot q1,q5
cnot q1,q6
cnot q0,q4
cnot q0,q5
cnot q0,q6
```

Program interaction graph





SWAP Insertion & Qubit Routing

- **Qubit routing:** Insert SWAP gates to move qubit states so required gates can execute
- **Goal:** Minimize the number of added SWAPs (each SWAP = 3 CNOTs of overhead)

- The Routing Process
  - Step 1: Pick the next two-qubit gate from the graph
  - Step 2: Check if the mapped physical qubits are adjacent
  - Step 3: If not adjacent → insert SWAPs along the shortest path
  - Step 4: Execute the gate, update the mapping, proceed to next gate

- **Challenge:** Choosing WHERE to SWAP affects ALL future gates!

- We want to automate this solution

Qubit Mapping and Routing in Other Quantum Hardware

- In superconducting qubits -> SWAP chain
- In trapped ions→ shuttling
- In neutral atoms -> atom rearrangements using optizal tweazers
- Beyond the course (and my) knowledge

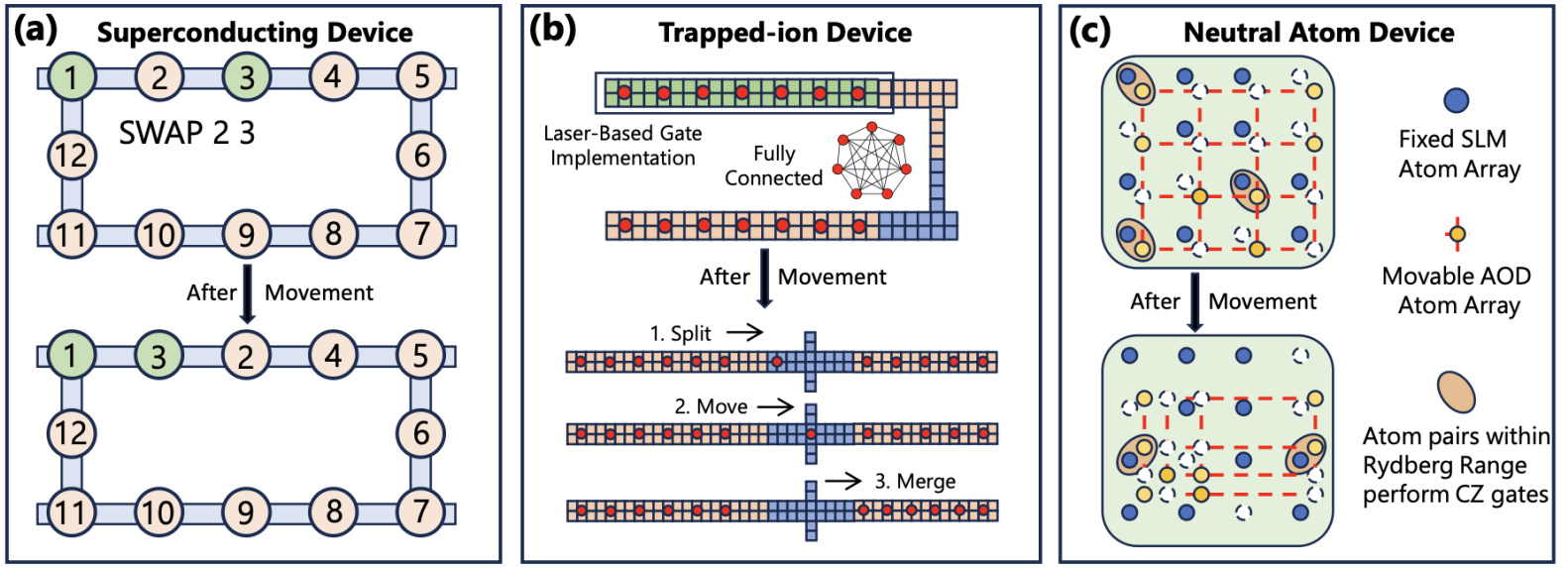


Fig. 1. Overview of the three mainstream quantum hardware architectures and their graph illustrations: (a) superconducting devices, (b) trapped-ion devices, and (c) neutral atom devices.

Qubit Routing Cost

Cost Model

- Model overhead as **weighted edges** on the device connectivity graph
- Weighted distance from q_i to q_j = cost of moving between locations
- Longer swap chains: more steps, more error accumulation, more likely to overlap

Two-Step Mapping Approach

  - **Step 1 — Allocate:** Assign logical qubits to physical locations
  - **Step 2 — Route:** Resolve long-distance interactions via SWAP insertion
  - Note: Mapping and gate scheduling are **tightly intertwined** — best optimized together

- Qubit routing cost can be computed with
  - Edge distance: the Euclidean distance between the physical locations of each endpoint of each edge in the mapped interaction graph -> latency
  - Edge Density Uniformity: When two edges in the mapped interaction graph are very close to each other, they are more likely to intersect and cause congestion
  - Edge Crossings Minimization: an edge crossing in a mapping as two pairs of endpoints that intersect in their geodesic paths, once their endpoint qubits have been mapped

Formalizing the Qubit Mapping Problem

Formal Definitions

- **Initial Mapping (π):** An injective function $L \rightarrow P$ mapping logical qubits to physical ones
- **Dynamic Mapping:** π is updated during execution as SWAPs permute qubit positions
- **Coupling Graph (G_p):** $G_p = (V, E)$ where V = physical qubits, E = allowed 2-qubit interactions
- **Objective:** Find π and a SWAP sequence that minimizes total gate overhead

NP-Completeness of the QMP

- QMP reduces to **subgraph isomorphism** (a known NP-complete problem)
- Even finding the **optimal initial mapping alone** is NP-hard
- Joint optimization of mapping + routing = combinatorial explosion

Implications

- **Exact solvers (SAT/SMT)** guarantee optimality but only scale to ~20-30 qubits
- **Heuristics (SABRE)** are necessary for NISQ-scale (100-1000+ qubits) circuits
- This NP-completeness motivates the entire heuristic vs. exact methods debate

Toy Example

- Assume 4-qubit device model (Figure 3 (b))
- Two-qubit gates are allowed on the following physical qubit pairs:
  - $\{Q1, Q2\}, \{Q2, Q4\}, \{Q4, Q3\}, \{Q3, Q1\}$
- and not allowed on
  - $\{Q1, Q4\}, \{Q2, Q3\}$
- Now suppose we have a small quantum circuit of six CNOT gates to be executed on this 4-qubit device (Figure 3 (c))
- Initial logical-to-physical qubits mapping is

$$\{q1 \rightarrow Q1, q2 \rightarrow Q2, q3 \rightarrow Q3, q4 \rightarrow Q4\}$$

- To execute the remaining CNOTs, Q_1 and Q_2 are swapped

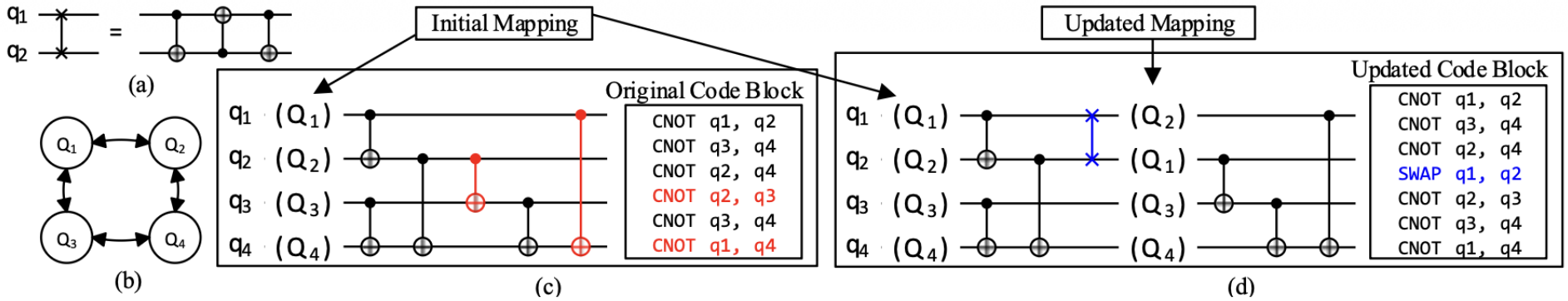


Figure 3. (a) SWAP Gate Decomposition, (b) Physical Qubit Coupling Graph Example, (c) Original Quantum Circuit, (d) Updated Hardware-Compliant Quantum Circuit

DAG Representation of Quantum Circuits

What is a DAG?

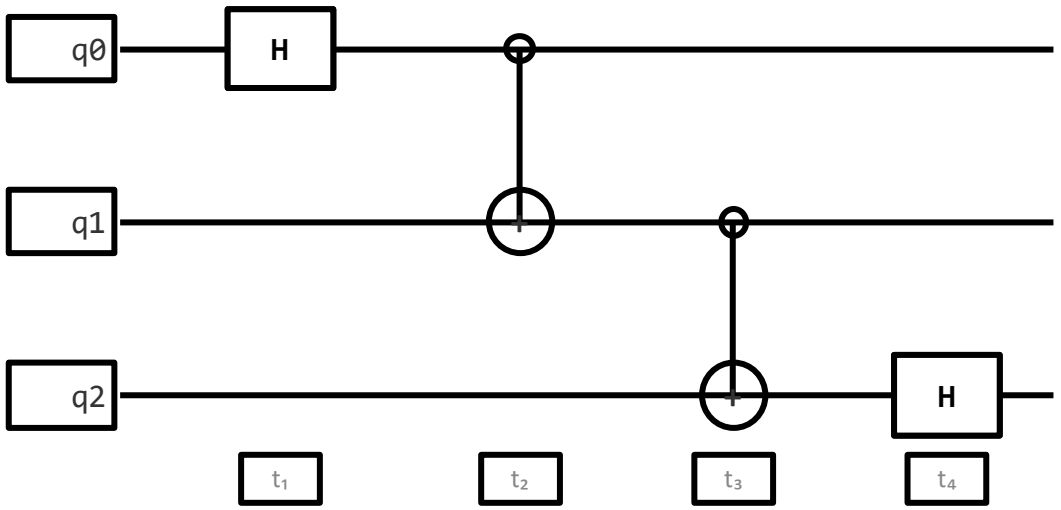
- A **Directed Acyclic Graph (DAG)** captures the dependency structure of a quantum circuit
- **Nodes** represent quantum gates (operations)
- **Edges** represent data dependencies (qubit wire connections)
- **Direction** encodes temporal ordering (gate A must execute before gate B)

Why Use a DAG for Mapping?

- **Front layer:** Gates with no unresolved dependencies → ready to execute
- **Scheduling freedom:** Independent gates can be reordered without changing semantics
- **Look-ahead:** Routing algorithms inspect future layers to make smarter SWAP decisions
- **Critical path:** Longest chain in the DAG determines minimum circuit depth

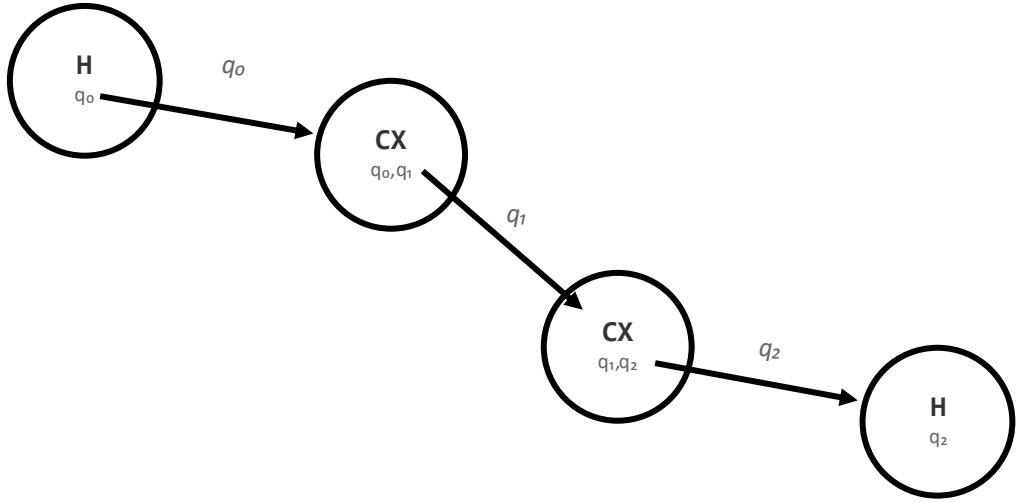
Constructing a DAG from a Quantum Circuit

Quantum Circuit



construct

DAG Representation



Construction Steps

1 **Create a node for each gate**
Each gate in the circuit becomes a node in the DAG. Label with gate type and qubit operands.

2 **Add directed edges**
Draw edge from gate A to gate B if they share a qubit and B comes after A with no gate on that qubit between them.

3 **Result: a DAG**
Edges encode data dependencies. Independent gates (no shared qubits, no path) can be scheduled in parallel.

More Efficient DAG Construction

- Neglect one-qubit gate in DAGs -> executed locally on one qubit without bringing dependencies on other qubits
- A two-qubit gate $CNOT\ (q_i\ , q_j)$can be executed only when all the previous two-qubit gates on q_i or q_j have been executed
- Traverse the entire quantum circuit and construct a DAG to represent execution dependencies with complexity $O(g)$, where g is the number of gates

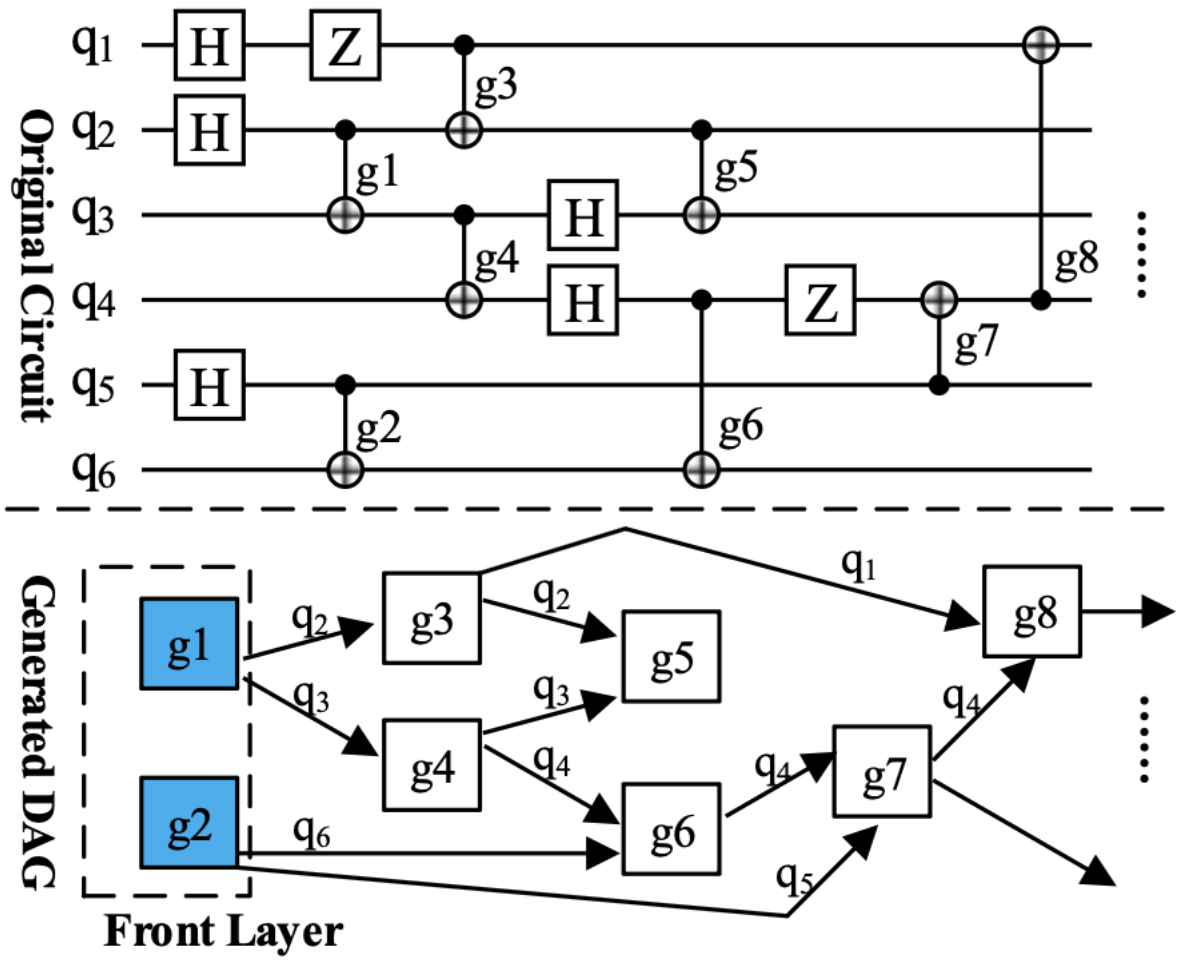


Figure 4. Example of DAG Generation and Front Layer Initialization.

SABRE: SWAP-based Bidirectional Heuristic Search Algorithm

- Most widely used routing heuristic (default in Qiskit)
- Key Strategies
  - Look-ahead Heuristic: Score SWAPs by distance reduction in the front layer + upcoming gates
  - Decay Effect: Penalize frequently swapped qubits to prevent hot spots and balance parallelism
  - Reverse Traversal: Run mapper backward through reversed DAG to refine initial placement
- Algorithm Flow
  - Build circuit DAG → identify front layer → score candidate SWAPs
  - Apply best SWAP → execute resolved gates → update DAG → repeat
  - Final mapping from forward pass seeds the backward pass; iterate to converge
  - Polynomial time → scales to 1000+ qubits, but no optimality guarantee

Heuristics vs. Exact Methods

Heuristic Approach: SABRE

- Greedy, stochastic SWAP selection guided by distance-based cost function
- **Speed:** Polynomial time; handles 1000+ qubit circuits in seconds
- **Quality:** Near-optimal for most practical circuits, but **no optimality guarantee**

Exact Approach: SMT/SAT Solvers

- Encode mapping + routing as a **Satisfiability Modulo Theories** (SMT) formula
- **Speed:** Exponential worst-case; practical limit ~20-30 qubits
- **Quality: Provably optimal** SWAP count for the given constraints

When to Use Each?

- **SABRE:** Production compilation, large circuits, rapid iteration
- **SMT:** Benchmarking, small critical subroutines, verifying heuristic quality
- **Hybrid:** Use SMT on critical subcircuits, SABRE on the rest

Optimization Passes & Pre-Mapping

Assertion-Based Pre-Optimization

- Use **pre- and post-conditions** on gates to eliminate redundancies before mapping
- Example: Remove SWAPs between qubits in identical states (provably no-ops)
- Fewer gates **before mapping** = simpler routing problem downstream

Gate Merging

- Combine consecutive **single-qubit rotations** (Rx, Ry, Rz) into one unitary
- Reduces the number of operations the physical mapper must schedule
- Especially effective for variational circuits (VQE, QAOA) with many rotation layers

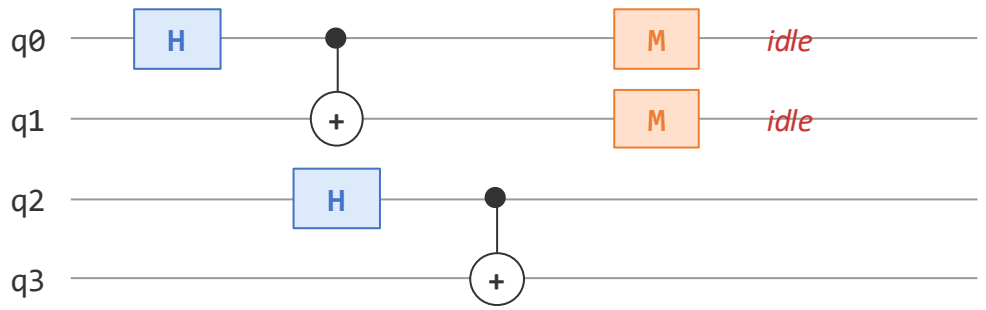
Subroutine Boundary Optimization

- Optimizing **across subroutine boundaries** can reduce total qubit width
- Fewer active qubits = **less constrained mapping** (easier to fit on the coupling graph)
- Trade-off: cross-boundary optimization breaks modularity of circuit design

Qubit Reuse

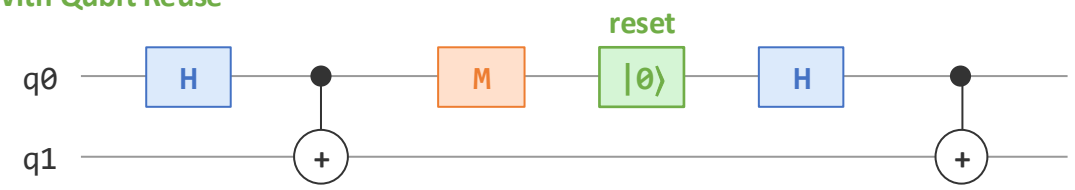
- After a qubit is measured or is no longer needed, it can be reset to $|0\rangle$ and reused as a fresh qubit for a different part of the circuit.
- Why Reuse Qubits?
  - Fewer physical qubits: reduces hardware requirements by recycling measured or unneeded qubits.
  - Reduced decoherence: Idle qubits accumulate errors. Reuse eliminates idle time by keeping qubits active
  - Enables larger algorithms: Allows running circuits that would otherwise exceed the device's qubit count

✗ Without Qubit Reuse



4 qubits needed

✓ With Qubit Reuse



2 qubits needed!

Uncomputation

- Uncomputation reverses an intermediate computation to return ancilla qubits back to $|0\rangle$, disentangling them so they can be safely discarded or reused
- This is critical for maintaining coherence in reversible quantum computation
- Similar to Garbage Collection in Java!
- How It Works
  1. Compute (U): Apply U to compute an intermediate result in the ancilla qubit
  2. Use result (V): Gate V uses the ancilla to modify the output qubit
  3. Uncompute ($U^\dagger$): Apply $U^\dagger$ (inverse of U) to return ancilla to $|0\rangle$

- Key Insight: Since quantum gates are unitary (reversible), any computation U can always be undone by applying $U^\dagger$. This restores the ancilla to $|0\rangle$, disentangling it from the rest of the system

Uncomputation - Example

