An introduction to implementing quantum algorithms (a Qiskit tutorial)

Lindon Roberts, University of Melbourne

MathQuEST Day, UWA, 30/11/26

(Note: this cell has some hidden LaTeX code to define bra-ket notation, double-click to see)

1. Installation

Make sure you have installed Qiskit using the instructions here:
https://quantum.cloud.ibm.com/docs/en/qiskit/install-qiskit

Starting from a fresh virtual environment, you should install the relevant Qiskit libraries (main SDK, IBM runtime, Aer simulator) via:

```
pip install qiskit
pip install qiskit-ibm-runtime
pip install qiskit-aer
```

Some other dependencies that are needed for this tutorial are:

```
pip install jupyter
pip install matplotlib
pip install pylatexenc
```

After this, you can start Jupyter, to run this notebook or create your own:

```
jupyter notebook
```

In [1]:
```
# Load key modules for circuit construction
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
from qiskit.quantum_info import Statevector
from qiskit.circuit.library import QFTGate  # for QPE in Shor's algorithm exampl

# Runtime tools
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit_ibm_runtime.executor_sampler import Sampler
from qiskit_ibm_runtime import RuntimeDecoder, RuntimeEncoder

# Load visualization tools
from qiskit.visualization import plot_histogram

# Misc tools
```

```
from fractions import Fraction
import json
from math import floor, gcd, log, sqrt
import numpy as np
```

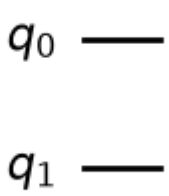
2. Creating Quantum circuits

Recall: an n-qubit quantum register is a unit vector in $\mathbb{C}^{2^n}$, and a quantum circuit implements a sequence of unitary operations to a given vector.

We can create a quantum circuit on, for example, $n = 2$ qubits via:

In [2]:
```
qc = QuantumCircuit(2)
qc.draw("mpl")
```

Out[2]:



In [3]:
```
# Get basic data about the circuit
print(f"Circuit operates on {len(qc.qubits)} qubits")
print(f"Circuit has gate depth {qc.depth()}")
```

```
Circuit operates on 2 qubits
Circuit has gate depth 0
```

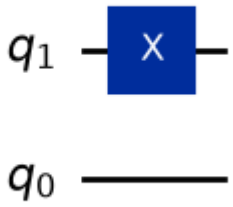
2.1 Single qubit operations

There are many common quantum gates (unitary matrices) that are available in Qiskit, such as the Pauli gates, which all operate on a single qubit: $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$ and $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. We also have the Hadamard gate $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

For an n-qubit system, we can apply these gates to one of the qubits, which acts as a tensor product, e.g. $I \otimes \cdots \otimes I \otimes X \otimes I \otimes \cdots \otimes I$.

In [4]:
```
# Add an X gate to the second (index 1) qubit
qc.x(1)
qc.draw("mpl", reverse_bits=True)
```

Out[4]:



Warning: Qiskit uses 'little Endian' representation of binary strings (least significant digit first), which is the opposite of how we normally write binary integers (i.e. q_0 operates on the 2^0 binary digit, q_1 operates on the 2^1 binary digit). To match with the intuition written below, we plot all circuits with the bit orders reversed.

For more details, see: https://quantum.cloud.ibm.com/docs/en/qiskit/bit-ordering

2.2 Multi-qubit operations

Some common gates operate on > 1 qubits simultaneously. For example, the CX (aka CNOT) gate has *control* and *target* bits. It can be uniquely described by its actions on the coordinate vectors in $\mathbb{C}^4$:

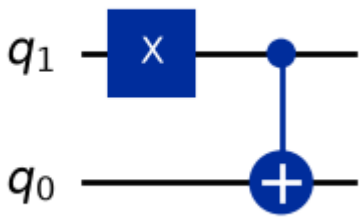
- If the control bit is $|0\rangle = (1,0)^T$, it does nothing to the control or target bits (i.e. $CX\,|00\rangle = |00\rangle$ and $CX\,|01\rangle = |01\rangle$)
- If the control bit is $|1\rangle = (0,1)^T$, it does nothing to the control bit, but flips the target bit, $|0\rangle \leftrightarrow |1\rangle$ (i.e. $CX\,|10\rangle = |11\rangle$ and $CX\,|11\rangle = |10\rangle$)

(recall: $|10\rangle = |1\rangle \otimes |0\rangle = (0,1)^T \otimes (1,0)^T = (0,0,1,0)^T \in \mathbb{C}^4$, i.e. $|10\rangle$ corresponds to the $[10]_2 = 2$nd, zero-indexed, coordinate vector in big Endian notation)

For example, if the control bit is the first qubit and the target bit is the second qubit, we get $CX_{12} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.

In [5]:
```
# Add a CX gate
qc.cx(1, 0)  # cx(control_bit, target_bit)
qc.draw("mpl", reverse_bits=True)
```

Out[5]:



2.3 Applying a quantum circuit

Qiskit allows us to construct a given quantum state in $\mathbb{C}^{2^n}$ and see what happens under a given circuit.

First, let's apply the above circuit to $|00\rangle = (1,0,0,0)^T$. We can check $(X \otimes I)\,|00\rangle = (0,0,1,0)^T = |10\rangle$ (flips first bit), and $CX_{12}\,|10\rangle = |11\rangle = (0,0,0,1)^T$.

In Qiskit, we can print a state in several ways, e.g. as a vector in $\mathbb{C}^{2^n}$, or as a dictionary of probabilities of seeing any given binary string after measurement.

In [6]:
```
input_state = Statevector.from_label("00")
print("Input state =", input_state.data)
print("Input state -> measurement probabilities:", input_state.probabilities_dic
output_state = input_state.evolve(qc)
print("Ouput state =", output_state.data)
```

```
Input state = [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
Input state -> measurement probabilities: {np.str_('00'): np.float64(1.0)}
Ouput state = [0.+0.j 0.+0.j 0.+0.j 1.+0.j]
```

Now, let's apply the circuit (let's call it Q) to a mixed state $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle - \frac{i}{\sqrt{2}}|11\rangle = (\frac{1}{\sqrt{2}},0,0,-\frac{i}{\sqrt{2}})^T$. We saw above that $Q\,|00\rangle = |11\rangle$, and we can similarly compute that $(X \otimes I)\,|11\rangle = |01\rangle$ and $CX_{12}\,|01\rangle = |11\rangle$, so $Q\,|11\rangle = |01\rangle$.

So, we expect $Q\,|\psi\rangle = -\frac{i}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle = (0,-\frac{i}{\sqrt{2}},0,\frac{1}{\sqrt{2}})^T$.

In [7]:
```
# Apply the circuit to the input vector [1/sqrt(2), 0, 0, -i/sqrt(2)]
input_state = Statevector([1/sqrt(2), 0, 0, -1j/sqrt(2)])
print("Input state =", input_state.data)
output_state = input_state.evolve(qc)
print("Ouput state =", output_state.data)
```

```
Input state = [ 0.70710678+0.j          0.        +0.j          0.        +0.j
 -0.        -0.70710678j]
Ouput state = [0.        +0.j         0.        -0.70710678j 0.        +0.j
 0.70710678+0.j        ]
```

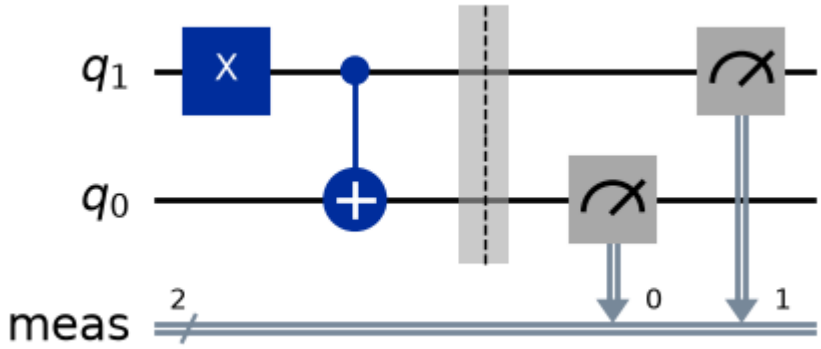
2.4 Measurement

Once we have created a quantum state, we need to measure the state. If we measure the k-th qubit of a state $|\psi\rangle = \sum_{j\in\{0,1\}^n} \alpha_j\,|j\rangle \in \mathbb{C}^{2^n}$, we get 0 with probability $\sum_{j\in\{0,1\}^n;j_k=0} |\alpha_j|^2$ and 1 with probability $\sum_{j\in\{0,1\}^n;j_k=1} |\alpha_j|^2$. The state $|\psi\rangle$ is then modified so that the kth qubit is deterministically 0 or 1, per the measurement (i.e. we only have a sum over $j \in \{0,1\}^n$ with $j_k = 0$ or $j_k = 1$), with the relevant subset of coefficients α_j renormalized so we still have a unit vector.

At any point in a circuit, we can measure either some/all qubits, and save the results into a classical register.

In [8]:
```
# Measure all qubits in our circuit
# (this automatically adds classical registers to the circuit, as needed)
qc.measure_all()
qc.draw("mpl", reverse_bits=True)
```

Out[8]:



3. Running a quantum circuit

Once we have constructed a circuit, we have to run the associated program (usually starting from an initial state $|00\cdots0\rangle$. Because of the randomness in quantum measurements, we usually need to run the circuit multiple times to understand the distribution.

The steps in running a circuit in Qiskit are:

1. Build the circuit (as above)
2. *Transpile* the circuit: create an equivalent quantum circuit that is suitable for running on particular quantum hardware
3. Run the circuit a given number of times (*shots*) to get a probability distribution of output states

The transpilation step is the hardest, and Qiskit seems to provide a bunch of functionality to customize this process. However, this is probably not so relevant to us. Instead, they provide a built-in function that generates a *pass manager* (≈ compiler object) that can be used to transpile your circuit for the relevant hardware:
https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.generate_preset_pass_ma

In [9]:
```
# Create a pass manager suitable for a given hardware
aer_sim = AerSimulator()  # <-- use Qiskit Aer simulator as the 'hardware' backe
pass_manager = generate_preset_pass_manager(backend=aer_sim, optimization_level=
qc_transpiled = pass_manager.run(qc)  # <-- create transpiled version of our cir
```

In [10]:
```
# Run the circuit
num_shots = 2048
sampler = Sampler(mode=aer_sim)
job = sampler.run([qc_transpiled], shots=num_shots)
```

In [11]:
```
# Extract the details of the run
result = job.result()
```

In [12]:
```
# Metadata associated with the experiment (e.g. runtime)
result[0].metadata
```

Out[12]:
```
{'circuit_metadata': {},
 'executor': {'shots': 2048,
  'circuit_metadata': {},
  'simulator_metadata': {'time_taken_parameter_binding': 1.7811e-05,
   'time_taken_execute': 0.054346171,
   'omp_enabled': True,
   'max_gpu_memory_mb': 0,
   'max_memory_mb': 7760,
   'parallel_experiments': 1}}}
```

In [13]:
```
# Extract the actual shot results
# Note: 'meas' is the default name of the classical/output register
answer = result[0].data['meas']
print(f"Results are measurements of {answer.num_bits} qubits")
print(f"Results comprised of {answer.num_shots} shots")
print("Results as a NumPy array (bitstring -> int):")
print(answer.array)
```

```
Results are measurements of 2 qubits
Results comprised of 2048 shots
Results as a NumPy array (bitstring -> int):
[[3]
 [3]
 [3]
 ...
 [3]
 [3]
 [3]]
```

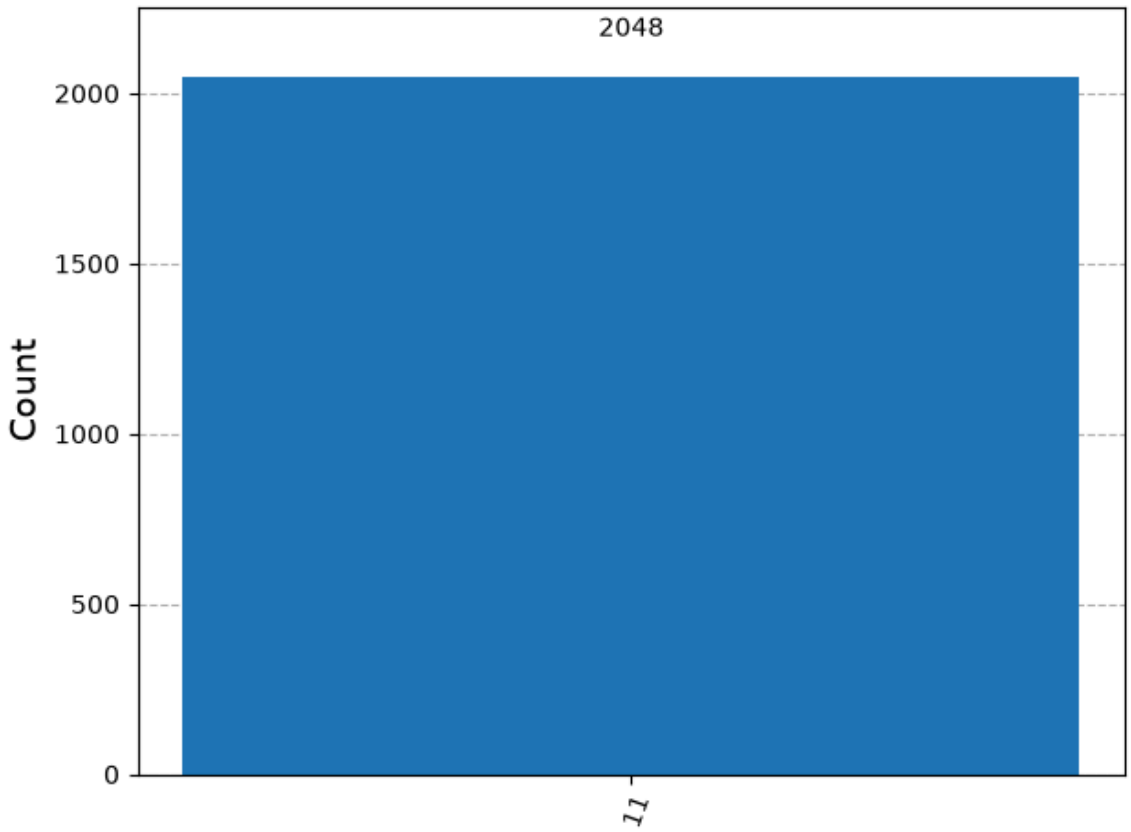
Just printing out an array of integers is not so helpful. There are some useful built-in routines to summarize the probability distribution.

In [14]:
```
# Get the count distribution over bitstrings:
answer.get_counts()
```

Out[14]:
```
{'11': 2048}
```

In [15]:
```
# Plot a histogram of the bitstring distribution
plot_histogram(answer.get_counts())
```

Out[15]:



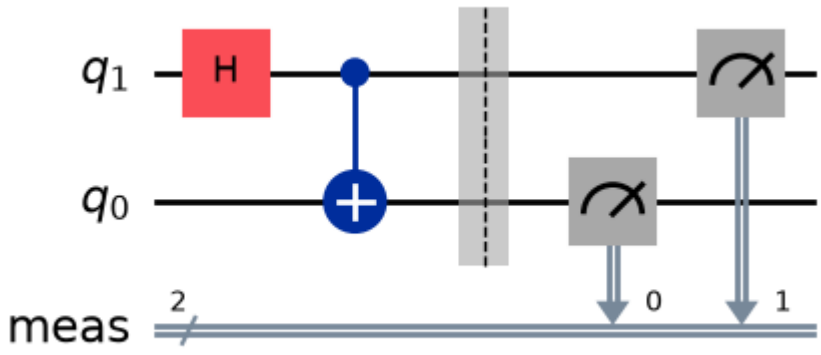
4. Complete (basic) example

Let's put all this together: we build and run a simple 2-qubit circuit that, starting from $|00\rangle$, produces an entangled state with a 50/50 distribution over $|00\rangle$ and $|11\rangle$.

If CX_{12} is the same controlled X-gate above and H is the 2-dimensional Hadamard matrix, our circuit comes from: $CX_{12}(H \otimes I)\,|00\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$.

In [16]:
```
# Build the circuit
qc_ex = QuantumCircuit(2)
qc_ex.h(1)
qc_ex.cx(1, 0)
qc_ex.measure_all()
qc_ex.draw("mpl", reverse_bits=True)
```

Out[16]:

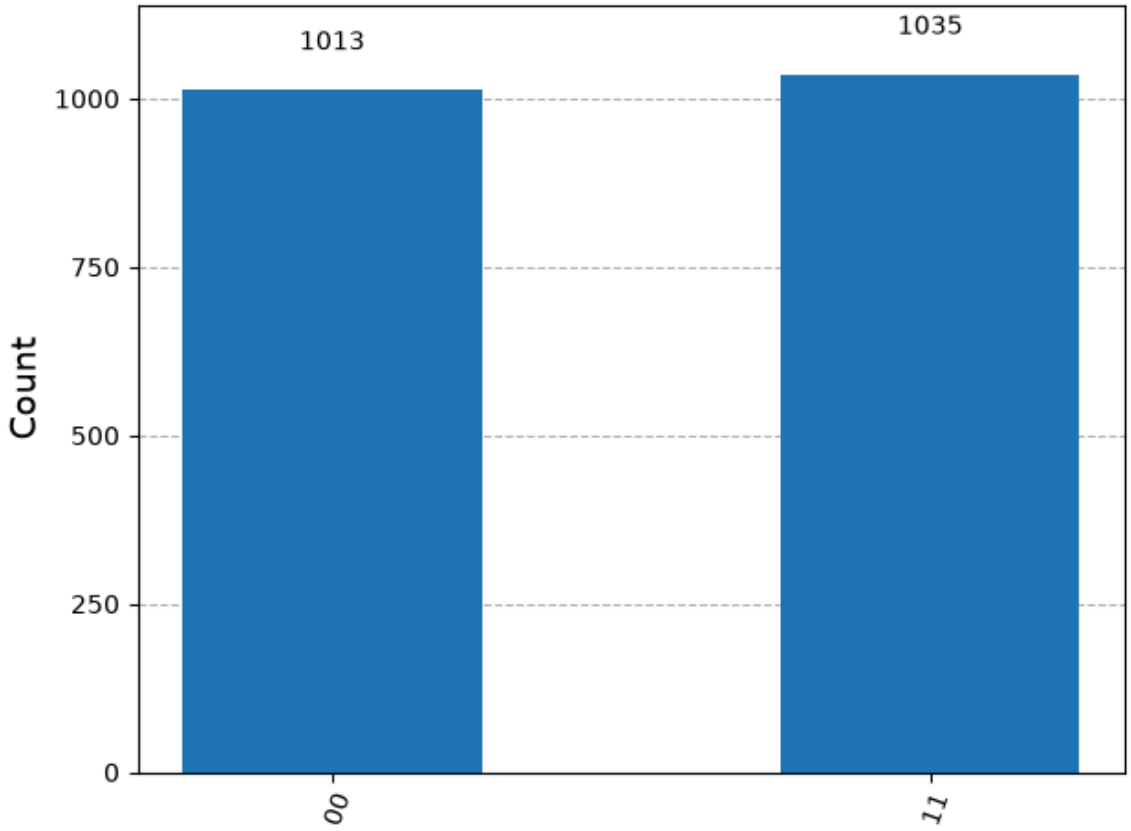


In [17]:
```
# Transpile the circuit
aer_sim = AerSimulator()  # <-- use Qiskit Aer simulator as the 'hardware' backe
pass_manager = generate_preset_pass_manager(backend=aer_sim, optimization_level=
qc_ex_t = pass_manager.run(qc_ex)  # <-- create transpiled version of our circui
```

In [18]:
```
# Run the circuit
num_shots = 2048
sampler = Sampler(mode=aer_sim)
job = sampler.run([qc_ex_t], shots=num_shots)
```

In [19]:
```
# Plot a histogram of the results: analytically, we expect a 50/50 split of stat
result = job.result()
answer = result[0].data['meas']
plot_histogram(answer.get_counts())
```

Out[19]:



5. Larger Example: Shor's Algorithm

To provide a more interesting example, we will follow this tutorial, and implement Shor's algorithm to compute the factorization $15 = 3 \times 5$.

This particular example yields a particularly small quantum circuit, so is good for tutorial purposes. It was first implemented on quantum hardware in 2001 (Vanderspyen et al., *Nature*, 2001). By comparison, factoring $N = 21$ on quantum hardware took much longer (Amico, Saleem & Kumph, *Physical Review A*, 2019).

Basic Approach

To find a nontrivial factor of N, we look for an integer x with $x^2 = 1 \mod N$ but $x \neq \pm 1 \mod N$: then, $(x+1)(x-1)$ is a multiple of N, so we can write $N = pq$ for

$p = \gcd(x-1, N)$ and $q = \gcd(x+1, N)$.

The full factorization process is:

1. Pick $a \in \{2, \ldots, N-1\}$ coprime to N (otherwise we've found a nontrivial factor).
2. Find the order of a modulo N: minimal r such that $a^r = 1 \mod N$. If r is odd, try a different a.
3. Set $x = a^{r/2} \mod N$. If $x = \pm 1 \mod N$, try a different a.
4. We have found a valid x, so return the factorization $N = pq$ with $p = \gcd(x-1, N)$ and $q = \gcd(x+1, N)$.

We will try this for $N = 15$ and $a = 2$ (with true order $r = 4$, giving $x = 4$ and hence we get the factorization $p = 3$ and $q = 5$).

Quantum Acceleration

The hardest part is step 2, which we accelerate using Quantum Phase Estimation: given a unitary matrix U and eigenstate $|\psi\rangle$, QPE computes the associated eigenvalue (which will be a complex scalar of unit magnitude, since U is unitary).

For Shor's algorithm, we apply QPE to the n-qubit unitary operator M defined by $M|j\rangle = |aj \mod N\rangle$ for $j \in \{0, \ldots, 2^n - 1\}$ (for our given selection of a). Since $M^r|1\rangle = |1\rangle$, an eigenstate of M is $|\psi_j\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{2\pi i j k/r} \left|a^k\right\rangle$ with associated eigenvalue $\lambda_j = e^{2\pi i \theta_j}$ for phase $\theta_j = j/r$. Thus, we can use QPE to get the order via $r = j/\theta_j$.

There are two problems:

1. Our computed θ_j will just be an estimate (a binary fraction of given precision). The larger the QPE circuit, the more digits of accuracy we get for θ_j, so for sufficient precision we can compute r to error < 0.5 (so rounding to integer gives the correct value).
2. We can't construct an eigenstate $|\psi_j\rangle$ without knowing r. Instead, we use the eigenstate $|\psi\rangle = |1\rangle$, which is a superposition of all eigenstates, $|1\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |\psi_j\rangle$. QPE will simultaneously estimate all periods $\theta_j = j/r$, and the measurement process will select one of these $\theta_j = j/r$ at random. We collect many measurements (each with a different j) and use these to find r.

We will use $m = 8$ qubits for QPE.

Implementation

For an n-qubit QPE implementation, we requires controlled gates for $U, U^2, \ldots, U^{2^{m-1}}$. In our case, $U = M$, so we need controlled gates for $M_{a^{2^k}}$ defined by $M_{a^{2^k}}|j\rangle = \left|a^{2^k} \mod N\right\rangle$ ($j = 0, \ldots, 2^n - 1$) for $k = 0, \ldots, m-1$.

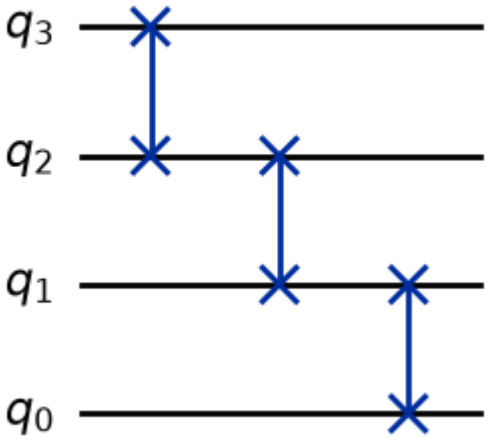
In our case ($N = 15$, $a = 2$, $m = 8$), we are lucky that $M_{a^{2^k}} = I$ for $k = 2, \ldots, 7$, so we only need to implement M_2 and M_4. In general, this is not the case, and more circuits need to be implemented.

For a general factorization problem, we could use the `UnitaryGate` Qiskit function to automatically construct a circuit from a given $M_{a^{2^k}}$. However, for this specific problem, we can derive M_2 and M_4 by considering how they operate on the binary representation of each state (and writing this in terms of bit swaps).

In [20]:
```
# Plot M2
num_qubits_N = 4  # number of qubits needed to represent j=0,...,N-1 for N=15

M2mod15 = QuantumCircuit(num_qubits_N)
M2mod15.swap(2, 3)
M2mod15.swap(1, 2)
M2mod15.swap(0, 1)
M2mod15.draw("mpl", reverse_bits=True)
```

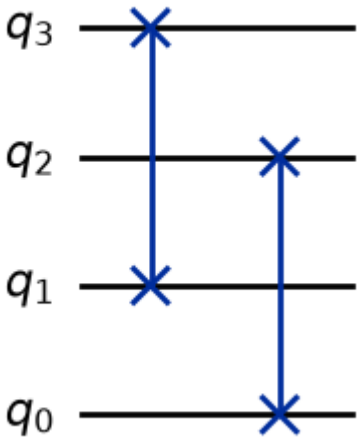
Out[20]:



In [21]:
```
# Plot M4
num_qubits_N = 4  # number of qubits needed to represent j=0,...,N-1 for N=15

M4mod15 = QuantumCircuit(num_qubits_N)
M4mod15.swap(1, 3)
M4mod15.swap(0, 2)
M4mod15.draw("mpl", reverse_bits=True)
```

Out[21]:



For QPE, we need to have *controlled* versions of M_2 and M_4 (i.e. if a control qubit is $|0\rangle$, do nothing, otherwise apply the operator to some target bits). We will also need to turn the operator into a `Gate` (a generic unitary operator object), to allow it to be composed with other operations.

In [22]:
```
# Implemented controlled M2 and M4, as gates
num_qubits_N = 4  # number of qubits needed to represent j=0,...,N-1 for N=15

def controlled_M2mod15():
    # Construct M2mod15 as above
    U = QuantumCircuit(num_qubits_N)
    U.swap(2, 3)
    U.swap(1, 2)
    U.swap(0, 1)
    U.name = "M_2"

    # Convert to a Gate object
    U = U.to_gate()

    # Make controlled variant
    c_U = U.control()
    return c_U

def controlled_M4mod15():
    # Construct M4mod15 as above
    U = QuantumCircuit(num_qubits_N)
    U.swap(1, 3)
    U.swap(0, 2)
    U.name = "M_4"

    # Convert to a Gate object
    U = U.to_gate()

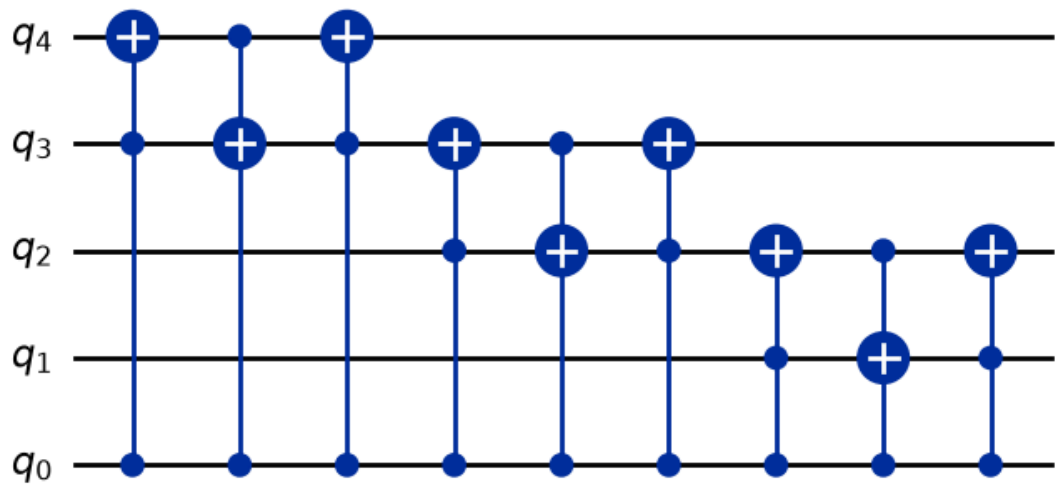
    # Make controlled variant
    c_U = U.control()
    return c_U
```

In [23]:
```
# Plot controlled-M2
controlled_M2 = controlled_M2mod15()

# Add it to a circuit and plot
circ2 = QuantumCircuit(num_qubits_N + 1)
circ2.compose(controlled_M2, inplace=True)

# Just drawing circ2 isn't interesting, it just has an "M2" block
# We need to decompose it into its constituents
#circ2.draw("mpl", reverse_bits=True)
circ2.decompose().draw("mpl", reverse_bits=True)
# Note: each SWAP gate is decomposed into 3xCX gates
#       (controlled SWAP -> 3x CCX gates)
```

Out[23]:

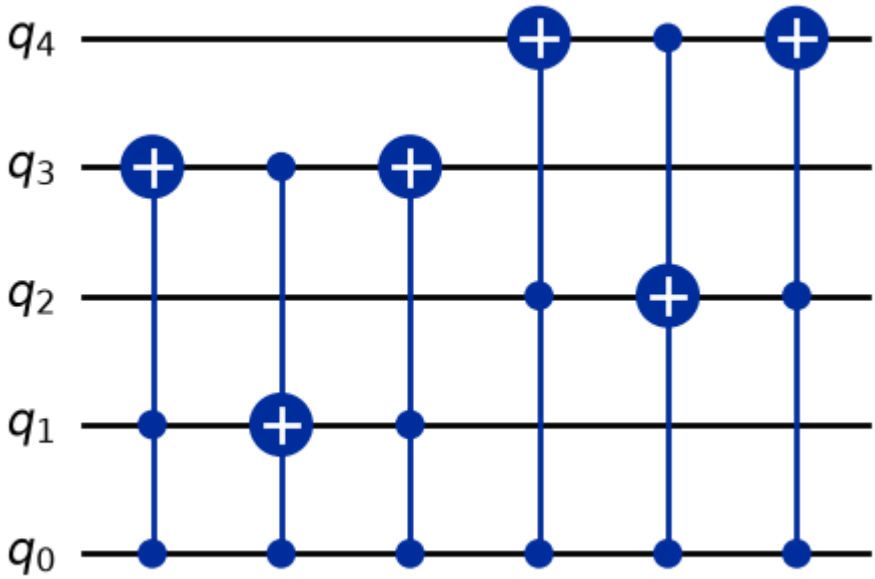


In [24]:
```
# Plot controlled-M4
controlled_M4 = controlled_M4mod15()

# Add it to a circuit and plot
circ4 = QuantumCircuit(num_qubits_N + 1)
circ4.compose(controlled_M4, inplace=True)

# Just drawing circ4 isn't interesting, it just has an "M4" block
# We need to decompose it into its constituents
#circ4.draw("mpl", reverse_bits=True)
circ4.decompose().draw("mpl", reverse_bits=True)
```

Out[24]:



Now, we construct the full QPE circuit, which we will use for order estimation

In [25]:
```
# Helper routine for constructing M operators:
# Compute a^{2^k} (mod N) by repeated squaring
def a2kmodN(a, k, N):
    for _ in range(k):
```

```
        a = int(np.mod(a**2, N))
    return a

# Target values
N = 15
a = 2

# Number of qubits
num_target = floor(log(N - 1, 2)) + 1  # for M2, M4, ...
num_control = 2 * num_target  # for precision of periods theta_j

# List of M operators in order
k_list = range(num_control)
b_list = [a2kmodN(2, k, 15) for k in k_list]

# Initialize the registers
control = QuantumRegister(num_control, name="C")  # for applying M2, M4, ...
target = QuantumRegister(num_target, name="T")  # for storing theta_j
output = ClassicalRegister(num_control, name="out")  # classical outputs
circuit = QuantumCircuit(control, target, output)  # combine into a circuit

# Initialize the target register to the state |1>
circuit.x(num_control)

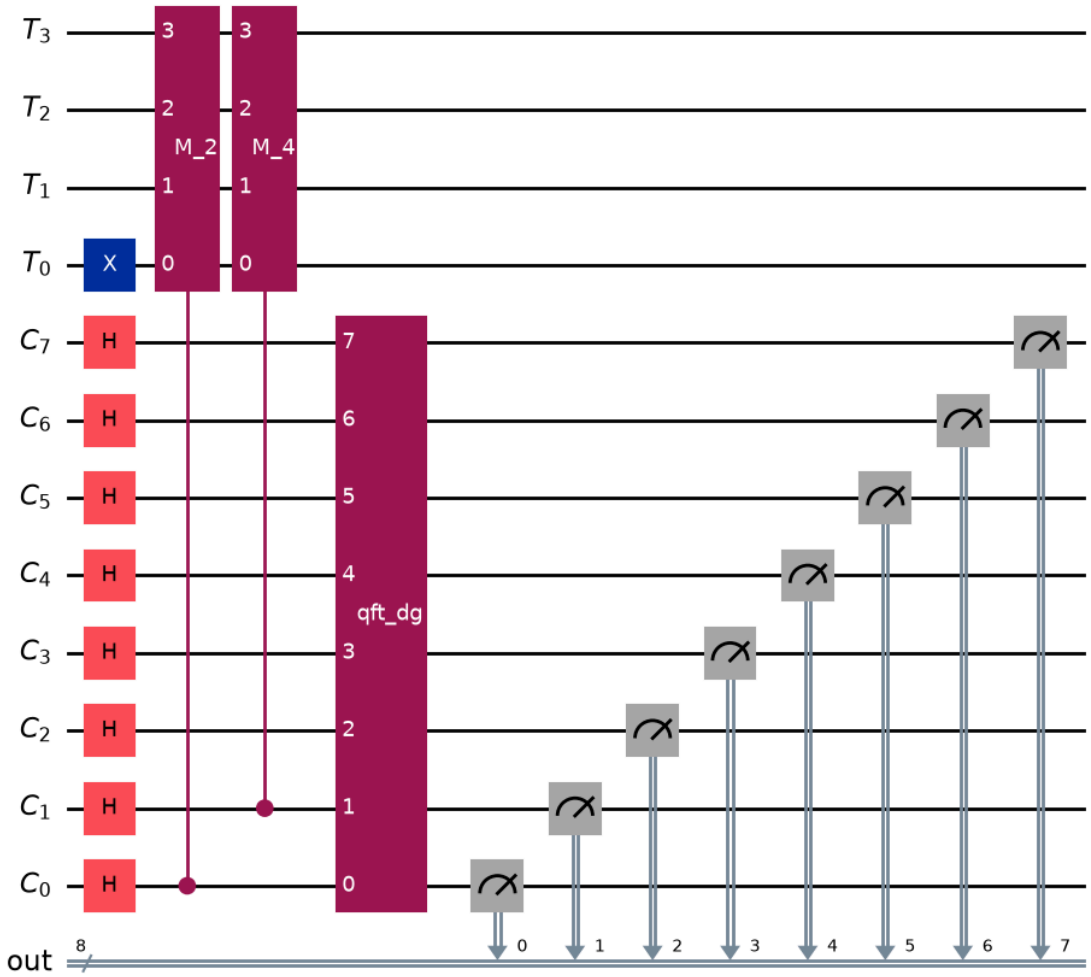
# Build QPE circuit:
for k, qubit in enumerate(control):
    circuit.h(k)
    b = b_list[k]
    if b == 2:  # M2
        circuit.compose(
            controlled_M2mod15(), qubits=[qubit] + list(target), inplace=True
        )
    elif b == 4:  # M4
        circuit.compose(
            controlled_M4mod15(), qubits=[qubit] + list(target), inplace=True
        )
    else:
        continue  # M_{2^k} is the identity operator

# Apply the inverse QFT to the control register
circuit.compose(QFTGate(num_control).inverse(), qubits=control, inplace=True)

# Measure the control register
circuit.measure(control, output)

# Draw the QPE circuit (fold=-1 allows for overflow in large circuits)
circuit.draw("mpl", fold=-1, reverse_bits=True)
```

Out[25]:



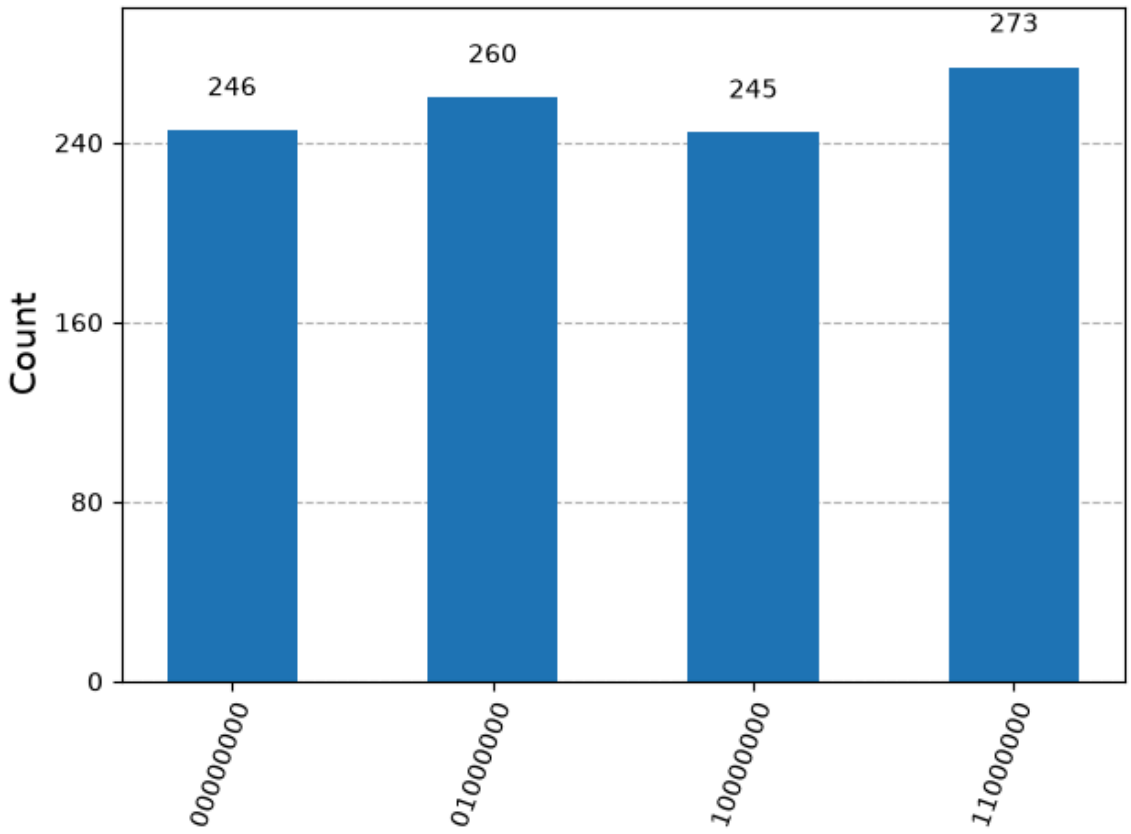
In [26]:
```
# Transpile and run the QPE circuit
aer_sim = AerSimulator()
pass_manager = generate_preset_pass_manager(backend=aer_sim, optimization_level=
transpiled_circuit = pass_manager.run(circuit)

num_shots = 1024
sampler = Sampler(mode=aer_sim)
job = sampler.run([transpiled_circuit], shots=num_shots)
```

In [27]:
```
# Plot a histogram of the results
result = job.result()
answer = result[0].data['out']  # <-- name of classical register
plot_histogram(answer.get_counts())

# Note: IBM tutorial running on actual quantum hardware has much more noise
# So, they filter out the low-frequency results
```

Out[27]:



Now, we switch back to the classical setting to compute the order r and hence get our factorization

In [28]:

```
# Classical post-processing of QPE
counts = answer.get_counts()

for bitstring in counts.keys():
    int_value = int(bitstring, 2)  # convert bitstring to integer in 0, ..., 2^{
    print(f"Testing output bitstring {bitstring}_2 = {int_value}")
    theta_est = int_value / (2**num_control)  # convert bitstring to binary frac
    print(f" - Estimated phase theta = {theta_est}")

    # Write theta_est as optimal fraction with denominator <= N
    # Since theta_est = j/r, the denominator should be r
    frac = Fraction(theta_est).limit_denominator(N)
    r = frac.denominator  # order = r
    print(f" - Estimated order r = {r}")

    # Check if a^r = 1 mod N (necessary condition for valid order)
    if pow(a, r, N) == 1:
        print(" - a^r == 1 mod N, possibly correct order")
        if r % 2 == 0:
            x = pow(a, r // 2, N)
            p = gcd(x - 1, N)
            q = gcd(x + 1, N)
            if p > 1 or q > 1:
                print(f"**** Factorization found (p={p}, q={q} from x={x}) ****"
        else:
            print(" - a^r == 1 mod N, possibly correct order, but r odd [skippin
    else:
        print(" - a^r != 1 mod N, incorrect order estimate")
    print("")
```

```
Testing output bitstring 10000000_2 = 128
 - Estimated phase theta = 0.5
 - Estimated order r = 2
 - a^r != 1 mod N, incorrect order estimate

Testing output bitstring 01000000_2 = 64
 - Estimated phase theta = 0.25
 - Estimated order r = 4
 - a^r == 1 mod N, possibly correct order
**** Factorization found (p=3, q=5 from x=4) ****

Testing output bitstring 11000000_2 = 192
 - Estimated phase theta = 0.75
 - Estimated order r = 4
 - a^r == 1 mod N, possibly correct order
**** Factorization found (p=3, q=5 from x=4) ****

Testing output bitstring 00000000_2 = 0
 - Estimated phase theta = 0.0
 - Estimated order r = 1
 - a^r != 1 mod N, incorrect order estimate
```

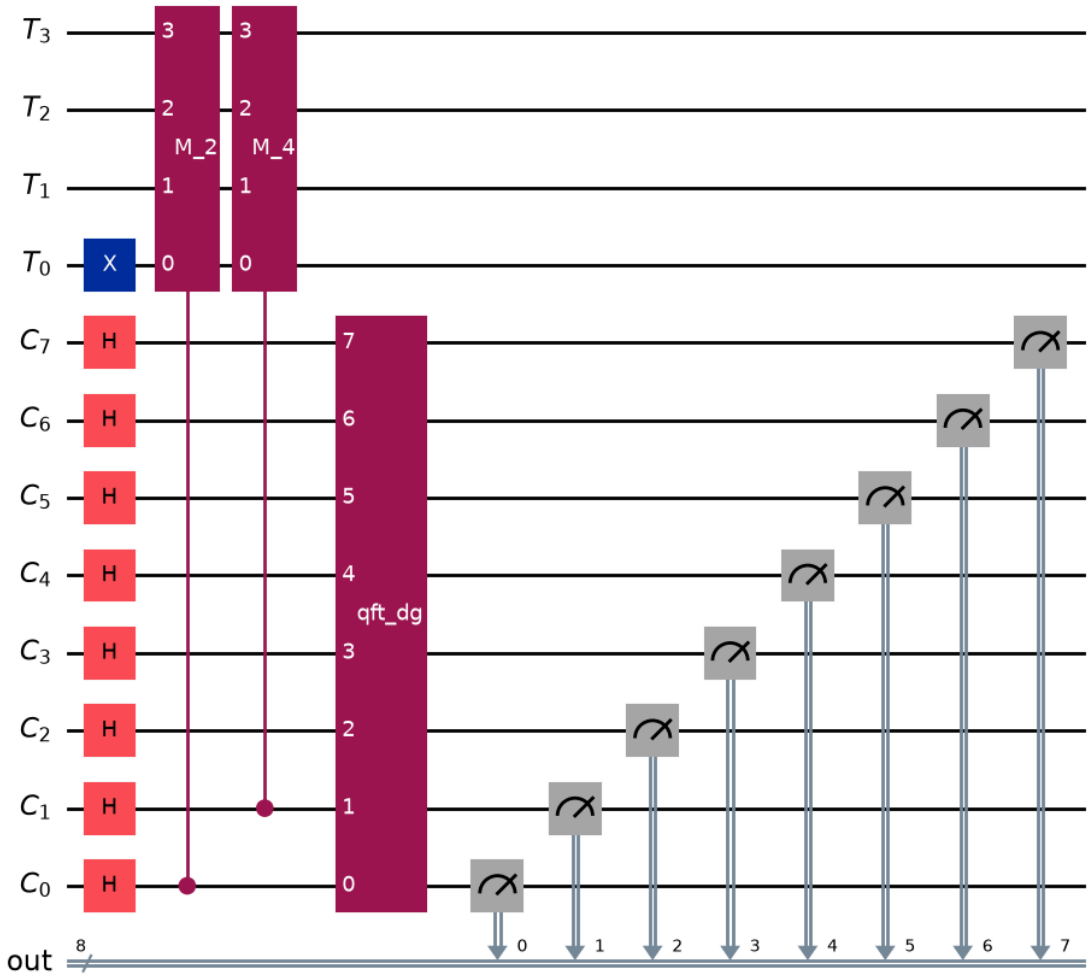
6. Running on Quantum Hardware

The above code runs using the local/classical Qiskit Aer simulator. With an IBM account, we can run a circuit on actual Quantum hardware. We demonstrate this using the simple 2-qubit example above (where we expect an equal distribution over $|00\rangle$ and $|11\rangle$ measurements).

Note: Aer can be configured to add realistic noise in the simulations, see here.

In [29]:
```
# Draw the QPE circuit, to make sure it's still loaded in memory
circuit.draw("mpl", fold=-1, reverse_bits=True)
```

Out[29]:



In [30]:
```
# Lindon's login details (private)
from qhub_login import API_KEY, CRN

# Save a copy of login details to ~/.qiskit/qiskit-ibm.json
#QiskitRuntimeService.save_account(channel='ibm_cloud', instance=CRN, token=API_

# Login to IBM server
#service = QiskitRuntimeService(channel='ibm_cloud', instance=CRN, token=API_KEY
service = QiskitRuntimeService(channel='ibm_cloud')  # <-- use saved login
backend = service.least_busy(operational=True, simulator=False)
```

In [31]:
```
print(backend)
```

```
<IBMBackend('ibm_fez')>
```

In [32]:
```
# For comparison, let's use two simulators: ideal/noise-free, and mimicking this
noisy_simulator = AerSimulator.from_backend(backend)
ideal_simulator = AerSimulator()

# Samplers
hardware_sampler = Sampler(mode=backend)
noisy_sampler = Sampler(mode=noisy_simulator)
ideal_sampler = Sampler(mode=ideal_simulator)
```

In [33]:
```
# Transpile the circuit for this particular hardware
pm = generate_preset_pass_manager(target=backend.target, optimization_level=2)
transpiled_circuit = pm.run(circuit)
```

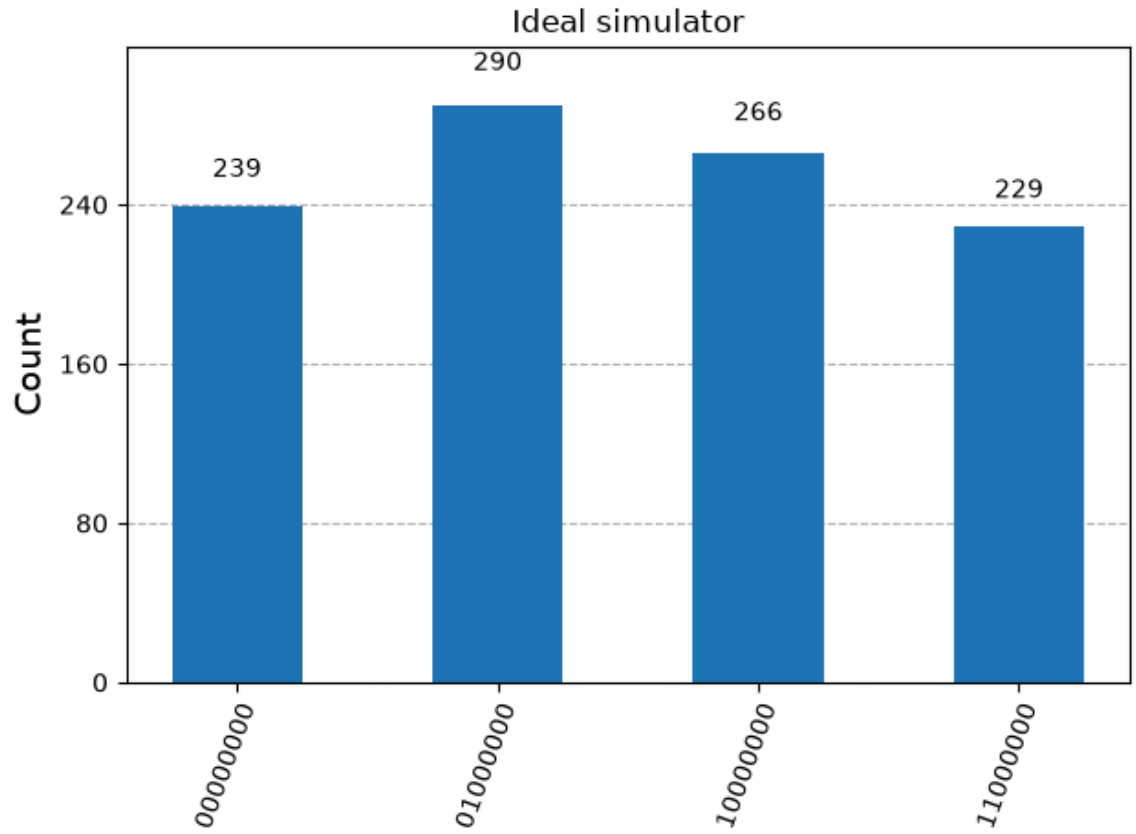
```
# Since different hardware backends have different low-level gates implemented
# we can inspect the actual size of our final, transpiled circuit
print(f"Operator counts: {transpiled_circuit.count_ops()}")
```

```
Operator counts: OrderedDict({'sx': 493, 'rz': 335, 'cz': 260, 'measure': 8, 'x': 1})
```

In [34]:
```
# Run the circuit
num_shots = 1024
ideal_job = ideal_sampler.run([transpiled_circuit], shots=num_shots)
noisy_job = noisy_sampler.run([transpiled_circuit], shots=num_shots)
#hardware_job = hardware_sampler.run([transpiled_circuit], shots=num_shots)  # <
## Save hardware results to file to avoid re-computing each time (this is slow,
#with open("hardware_job_results.json", "w") as file:
#  json.dump(hardware_job.result(), file, cls=RuntimeEncoder)
```

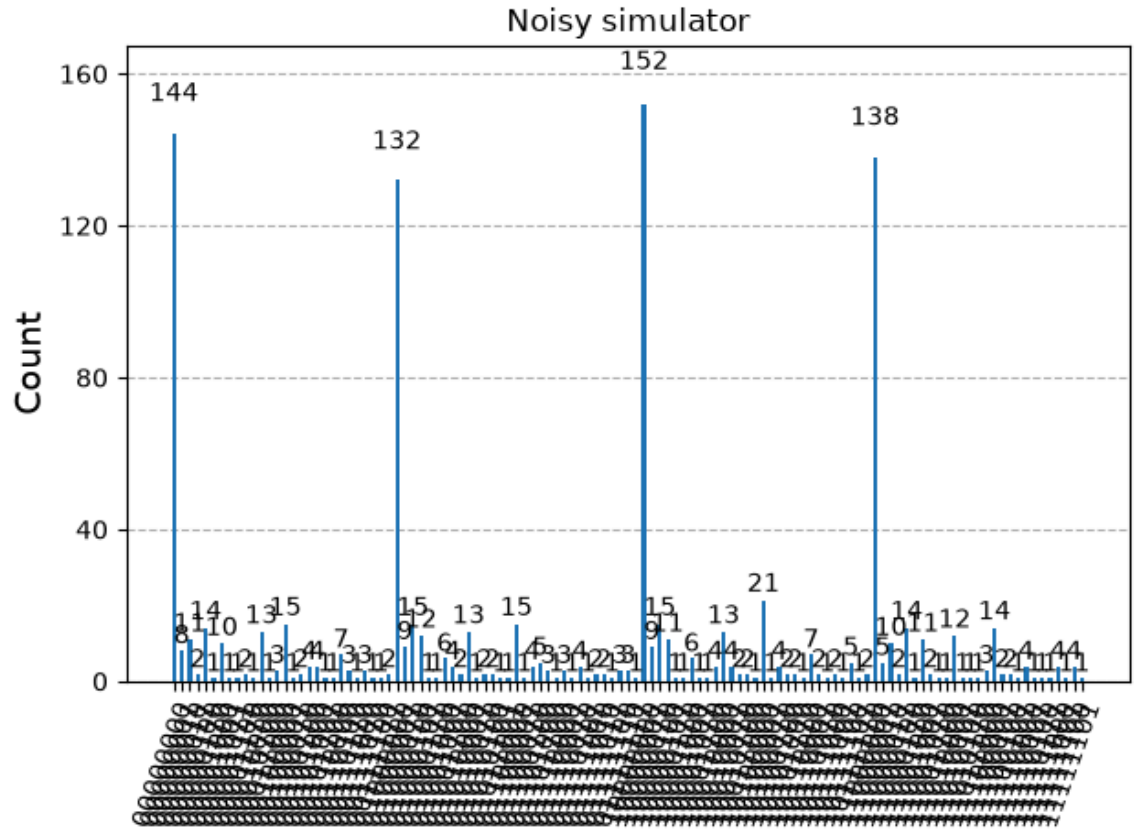
In [35]:
```
ideal_result = ideal_job.result()
ideal_answer = ideal_result[0].data['out']
plot_histogram(ideal_answer.get_counts(), title='Ideal simulator')
```

Out[35]:



In [36]:
```
noisy_result = noisy_job.result()
noisy_answer = noisy_result[0].data['out']
plot_histogram(noisy_answer.get_counts(), title='Noisy simulator')
```

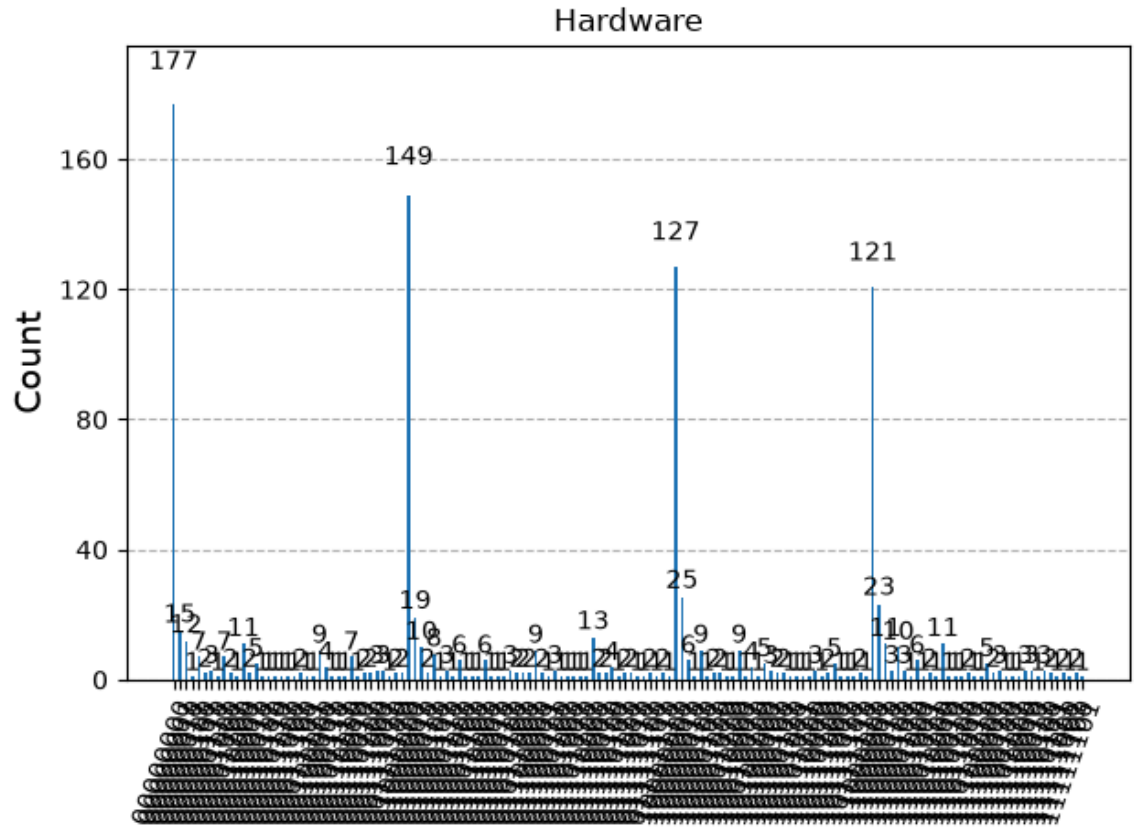
Out[36]:



In [37]:

```
#hardware_result = hardware_job.result()  # <-- load results directly after runn
with open("hardware_job_results.json", "r") as file:
  hardware_result = json.load(file, cls=RuntimeDecoder)
hardware_answer = hardware_result[0].data['out']
plot_histogram(hardware_answer.get_counts(), title='Hardware')
```

Out[37]:



In [38]:
```
hardware_runtime = hardware_result.metadata['execution']['execution_spans'].dura
print(f"Running the simulation on hardware took {hardware_runtime} seconds")
```

```
Running the simulation on hardware took 1.329909 seconds
```

In [39]:
```
# For noisy runs, we need to threshold the counts
# The IBM tutorial recommends ignoring outputs with count < 0.5 * max_count
def threshold_counts(counts):
    threshold = np.max(list(counts.values())) / 2
    counts_keep = {}
    for key, value in counts.items():
        if value > threshold:
            counts_keep[key] = value
    return dict(sorted(counts_keep.items()))  # sort by key

ideal_counts = threshold_counts(ideal_answer.get_counts())
noisy_counts = threshold_counts(noisy_answer.get_counts())
hardware_counts = threshold_counts(hardware_answer.get_counts())

print("Ideal counts =", ideal_counts, f" (kept {sum(ideal_counts.values())} shot
print("Noisy counts =", noisy_counts, f" (kept {sum(noisy_counts.values())} shot
print("Hardware counts =", hardware_counts, f" (kept {sum(hardware_counts.values
```

```
Ideal counts = {'00000000': 239, '01000000': 290, '10000000': 266, '11000000': 22
9}  (kept 1024 shots)
Noisy counts = {'00000000': 144, '01000000': 132, '10000000': 152, '11000000': 13
8}  (kept 566 shots)
Hardware counts = {'00000000': 177, '01000000': 149, '10000000': 127, '11000000':
121}  (kept 574 shots)
```