IBM Qiskit Developer Exam Prep

Table of contents

> Introduction
> Single qubit gates
> Execution and visualization of state vectors and circuit outputs
> Multi-qubit gates
> Phase kickback
> Non-unitary operations
> Analogies between classical and quantum logic
> Performing operations among quantum circuits
> Bell States
> GHZ States
> Getting circuit properties
> Plotting states in the Bloch Sphere
> Reading and writing in a QASM file
> Fidelity
> Operators
> Density operators
> Getting real quantum computers properties

Introduction

back to the top

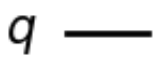
First, we need to import the libraries

In [57]:
```
import qiskit
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister,
                   execute, IBMQ
from qiskit.tools.visualization import circuit_drawer
```

Let's start by constructing the simplest quantum circuit

In [63]:
```
qc = QuantumCircuit(1)
qc.draw(output = 'mpl')
```

Out[63]:



Available output styles are `'mpl'`, `'text'`, `'latex'` and `'latex_source'`, which output a `matplotlib.Figure`, `TextDrawer` and `PIL.image` object, respectively.

Additional customizations are available like `plot_barriers = False` if we do not want barriers to be drawn or `reverse_bits = True` if we want to alter the order of qubits.

In [58]:
```
quantum_register = QuantumRegister(2, 'q', bits = None)
classical_register = ClassicalRegister(2, 'c', bits = None)

qc = QuantumCircuit(quantum_register, classical_register)
qc.draw(output = 'text', filename = 'circuit.pdf')
```

Out[58]:
```
q_0:

q_1:

c: 2/
```

An alternative function for circuit drawing is `circuit_drawer`, which works idntically to the `draw()` method, except it need a `QuantumCircuit` as argument.

In [64]:
```
circuit_drawer(qc, output = 'mpl')
```

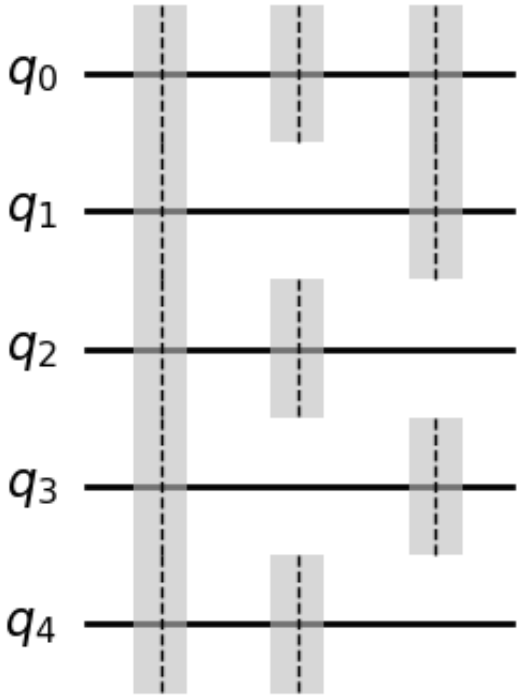
Out[64]:

Barriers can be added to any desired qubit(s) on the circuit if clarity is desired (specially handy in complex algorithms where the number of gates is extensive).

BE CAREFUL: Barriers also serve as circuit restrictors, i.e., if a barrier is placed between two gates that would otherwise simplify (like $XX = I$), such simplification cannot be performed.

In [65]:
```
qc = QuantumCircuit(5)
qc.barrier()
qc.barrier(range(0,5,2))
qc.barrier(0,1,3)
qc.draw('mpl')
```

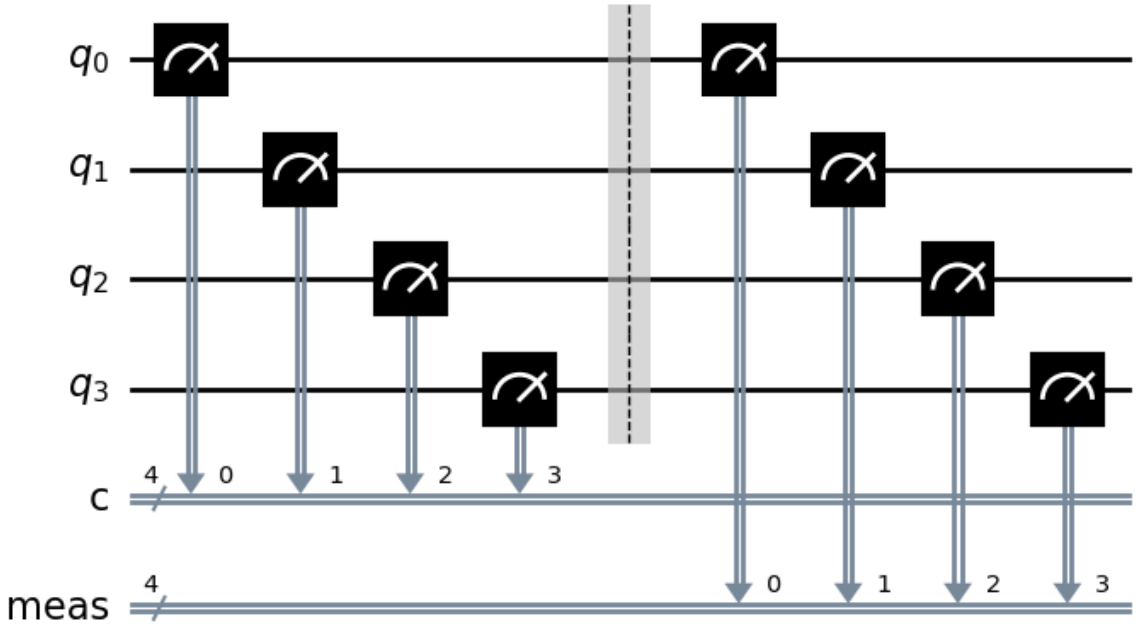
Out[65]:



Measurements of qubits can be done using either `measure_all(inplace=True, add_bits=True)` or `measure(qubit, cbit)`. The former, apart from measuring all qubits, adds a barrier by default before measurement is performed, while the latter does not. For example, `qc.measure([0,1],[0,1])` measures qubits q_0 and q_1 and stores their values into the classical bits c_0, c_1. On the other side, `measure_all()` creates by default a new register containing measurement bits. If one wishes to store the result of the measurements on the already exiting classical bits, set `add_bits` to `False`.

In [62]:
```
qc = QuantumCircuit(4,4)
qc.measure([0,1,2,3], range(4))
qc.measure_all()
qc.draw('mpl')
```

Out[62]:



In the exam they may ask what Qiskit version are we using. To simply get Qiskit's version,

In [13]:
```
qiskit.__version__
```

Out[13]:
```
'0.45.2'
```

To get more details, like the versions of each module,

In [2]:
```
qiskit.__qiskit_version__
```

Out[2]:
```
{'qiskit': '0.45.2', 'qiskit-aer': '0.12.0', 'qiskit-ignis': None, 'qiskit-ibmq-provider': '0.20.2', 'qiskit-nature': None, 'qiskit-finance': None, 'qiskit-optimization': None, 'qiskit-machine-learning': None}
```

We can also display this information in a table syle as follows

In [3]:
```
import qiskit.tools.jupyter
%qiskit_version_table
%qiskit_copyright
```

Version Information

Software	Version
`qiskit`	0.45.2
`qiskit_aer`	0.12.0
	System information
Python version	3.9.13
Python compiler	Clang 12.0.0
Python build	main, Aug 25 2022 18:29:29
OS	Darwin
CPUs	2
Memory (Gb)	8.0
	Wed Jan 31 11:17:57 2024 CET

Single qubit gates

back to the top

Now let's get into single qubit gates. First, we import the libraries needed for this. It is important to note that all single qubit and multi qubit gates can be implemented through methods of the `QuantumCircuit` class so no special libraries need to be imported for this. Later on we will see other ways of adding gates to our circuits.

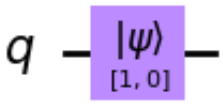
In [4]:
```
import qiskit
import numpy as np
from qiskit import QuantumCircuit, Aer, ClassicalRegister, QuantumRegiste
from math import pi, sqrt
```

SIDENOTE: Qubits on Qiskit are ordered from back to front, i.e., $|q_n \dots q_3 q_2 q_1 q_0\rangle$.

By default, qubits are initialized to $|0\rangle$ in Qiskit. Nonetheless, one may initialize qubits in a particular state vector $[a, b]$ in the basis $\{|0\rangle, |1\rangle\}$ using the `initialize(params, qubits = None, normalize = False)` method. This method takes a `Statevector` object as input `params`, which can be created in different ways:

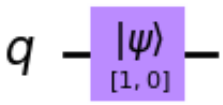
In [3]:
```
qc = QuantumCircuit(1)

state_vector = [1,0]
qc.initialize(state_vector, 0)

qc.draw('mpl')
```

Out[3]:



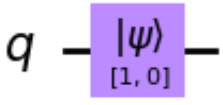
In [55]:
```
from qiskit.quantum_info import Statevector

qc = QuantumCircuit(1)

state_vector = Statevector.from_label('0')
qc.initialize(state_vector, 0)

qc.draw('mpl')
```

Out[55]:



In [3]:
```
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector

qc = QuantumCircuit(1)

state_vector = Statevector.from_int(0, 2**1) # the first argument is
# the index of the computational basis state.
# 0 -> |0>, 1 -> |1>
# the second is the dimension of the system
qc.initialize(state_vector, 0)

qc.draw('mpl')
```

Out[3]:



State preparation can be also achieved through the `prepare_state(state, qubits = None, label = None, normalize = False)` method.

Pauli-X, Y, Z gate

The X gate, also called bit-flip or NOT gate, transforms $|0\rangle \rightarrow |1\rangle$ and $|1\rangle \rightarrow |0\rangle$. Equates to a rotation of π around the X axis of the Bloch sphere (BS). Equivalently, Y performs a rotation of π around the Y axis of the BS, mapping $|0\rangle \rightarrow i|1\rangle$ and $|1\rangle \rightarrow -i|0\rangle$. Finally, the Z gate leaves $|0\rangle$ unchanged while mapping $|1\rangle \rightarrow -|1\rangle$ (that is why it is also called the phase-flip gate).

Their matrix forms can be written in the computational basis as follows (using the conventional Pauli matrices expressions)

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

NOTE: Notice how the Y gate acts as a combination of X and Z gate (due to the fact that $Y = iXZ$) i.e., it first applies a bit-flip, adds a phase of i to both states and flips it (adds a minus sign) if the original state was $|1\rangle$

Hadamard Gate

Maps $|0\rangle \rightarrow |+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $|1\rangle \rightarrow |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$. It basically creates an equal superposition of the computational basis states $\{|0\rangle, |1\rangle\}$. In the BS, it performs a rotation of π around the axis $(\mathbf{x} + \mathbf{z})/\sqrt{2}$. In matrix form,

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

NOTE: This gate can be seen as a basis change from $\{|0\rangle, |1\rangle\}$ to $\{|+\rangle, |-\rangle\}$. In can be easily seen that in this basis, X and Z gates play the reverse role, i.e., Z flips $|+\rangle$ to $|-\rangle$ and X leaves $|+\rangle$ unchanged while mapping $|-\rangle$ to - $|-\rangle$. This allows the gate simplifications

$$HXH = Z \qquad HZH = X$$

NOTE: It might be useful to remind that $H = (X + Z)/\sqrt{2}$

The three Pauli gates and the Hadamard gate are said to be **involutory** since they are their own inverse. More formally, a matrix A is said to be involutory if and only if $A^2 = I$.

S and $S^\dagger$ Gate

It performs a rotation of $\pi/2$ around the $\mathbf{z}$-axis. As a result, it leaves $|0\rangle$ unchanged while mapping $|1\rangle \rightarrow i|1\rangle$. In matrix form,

$$S = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

NOTE: The intermediate step used in the equality above is achieved using Euler's formula, $e^{i\phi} = \cos\phi + i\sin\phi$.

Similarly, the $S^\dagger$ gate performs a $-\pi/2$ rotation around the $\mathbf{z}$-axis and so,

$$S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & e^{-i\pi/2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$$

More generally, any gate of the form

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

is called a phase gate, since it adds a phase kickback of $e^{i\theta}$ to the $|1\rangle$ state (i.e., maps $|1\rangle \rightarrow e^{i\theta}|1\rangle$). Notice how $P(\theta = \pi) = Z$, $P(\theta = \pi/2) = S = \sqrt{Z}$ and $P(\theta = \pi/4) = T$.

T and $T^\dagger$ Gate

Similar to the S gate, it performs a rotation of $\pi/4$ around the $\mathbf{z}$-axis. As a result, it leaves $|0\rangle$ unchanged while mapping $|1\rangle \rightarrow e^{i\pi/4}|1\rangle$. In matrix form,

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{pmatrix}$$

Note that $S = T^2$. $T^\dagger$ works for T in the same way $S^\dagger$ works for S,i.e. it performs the reverse rotation.

Rotation gates (R_x, R_y, R_z)

Pauli matrices give rise to the rotation operators around the $\mathbf{x}, \mathbf{y}, \mathbf{z}$ axes. Since they arise from exponentiating these matrices, it is important to first note that

$$e^{iAx} = \cos(x)1 + i\sin(x)A$$

for any matrix A that satisfies $A^2 = 1$. Along this line one finds the matrix expressions for R_x, R_y, R_z

$$R_x(\theta) \equiv e^{-i\theta X/2} = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

$$R_y(\theta) \equiv e^{-i\theta Y/2} = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

$$R_z(\theta) \equiv e^{-i\theta Z/2} = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

U-Gate

It is the most general out of all single-qubit gates and is parametrized by the three Euler angles θ, ϕ, λ. Its matrix representation is

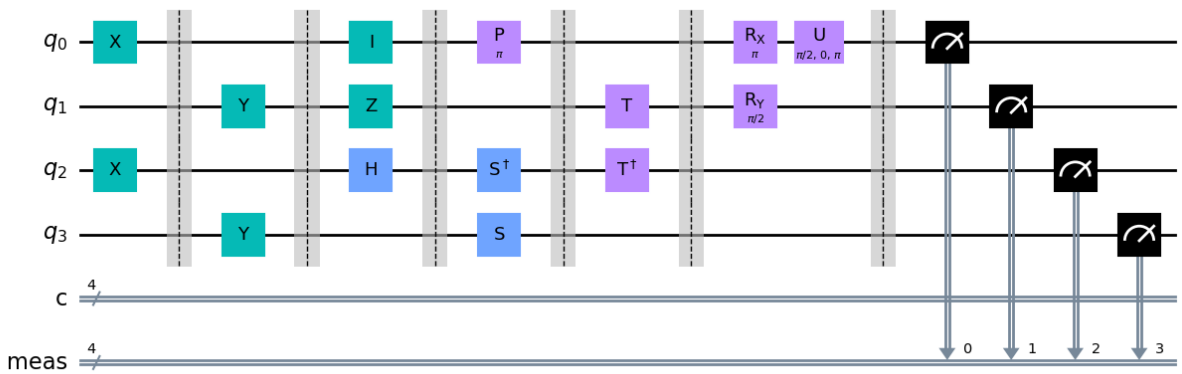
$$U(\theta, \phi, \lambda) = \begin{pmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\cos\frac{\theta}{2} \end{pmatrix}$$

Every single qubit gate can be written as an U-Gate for certain values of θ, ϕ, λ. For example, $U(\pi/2, 0, \pi) = H$, $U(0, 0, \lambda) = P(\lambda)$. However, it is not commonly used due to the difficulty in reading it.

The following snippet of code implements all single qubit gates we have seen until now. One should get familiar with both their methods and gate icons.

In [4]:
```
from math import pi

qc = QuantumCircuit(4,4)
qc.x([0,2])
qc.barrier()
qc.y([1,3])
qc.barrier()
qc.z(1)
qc.h(2)
qc.id(0) # Applies the Identity gate (i.e., leaves the qubit state unchan
qc.barrier()
qc.s(3)
qc.p(pi,0)
qc.sdg(2)
qc.barrier()
qc.t(1)
qc.tdg(2)
qc.barrier()
qc.rx(pi,0)
qc.ry(pi/2,1)
qc.u(pi/2, 0, pi, 0) # U always takes parameters first as input and then
qc.measure_all()
qc.draw('mpl')
```

Out[4]:



Apart from their own methods like `QuantumCircuit.sdg` or `QuantumCircuit.x`, quantum gates can added to a circuit more generally using the `append(instruction, qargs = None, cargs = None)` method.

In [7]:
```
from qiskit.circuit.library import XGate

qc = QuantumCircuit(1)
qc.append(XGate(), [0])
qc.draw('mpl')
```

Out[7]:

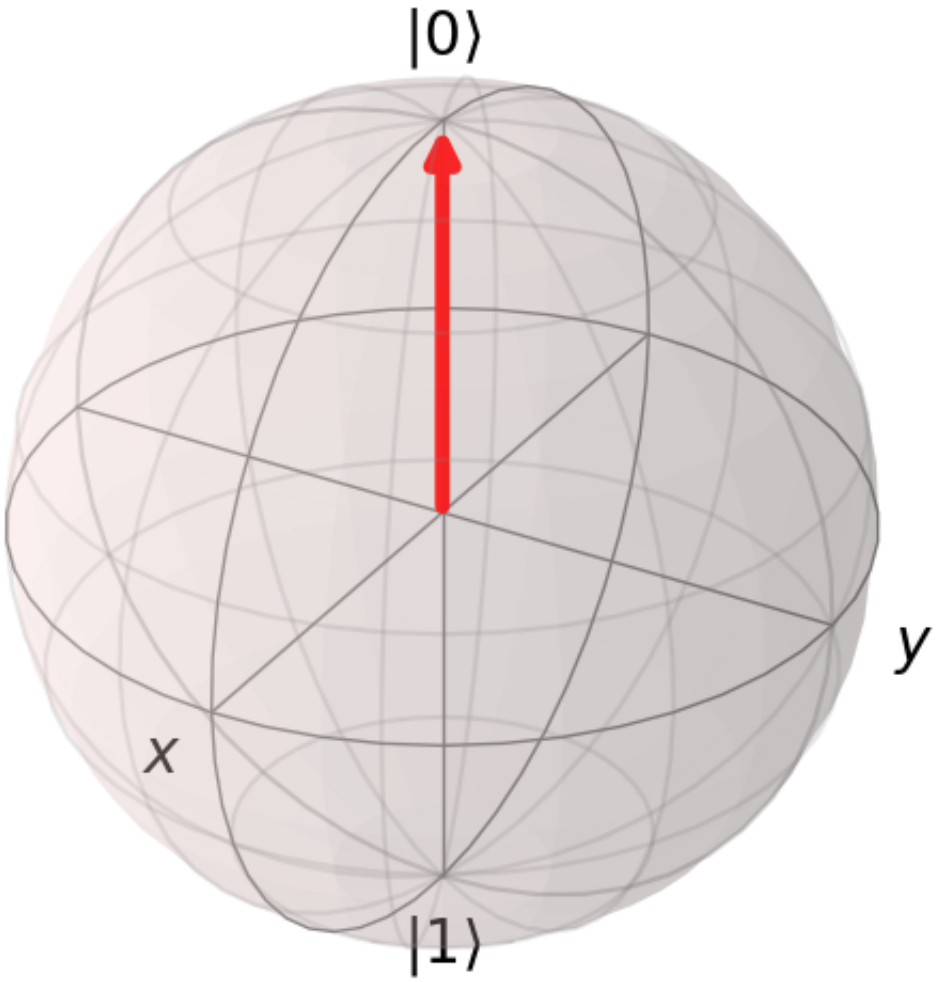
Execution and visualization of state vectors and circuit outputs

back to the top

If we are interested in seeeing how these gates change the orientation of our states in the Bloch sphere (not asked in the exam but might be helpful), we can run the following lines

In [21]:
```
from qiskit.visualization import visualize_transition

qc = QuantumCircuit(1)
qc.h(0)
qc.z(0)
qc.h(0)
visualize_transition(qc, trace = True) # Setting trace to True allows to
# see the trajectory followed by the state.
```

Out[21]:



It is worth noting that if we wanted to visualize the effects of a phase gate on a qubit, Qiskit would raise an error. This is due to the fact that global phases do not affect the state so there is no transition to visualize.

For all the following visualizations we will need to import the libraries below:

In [50]:
```
from qiskit import Aer, QuantumCircuit, execute, QuantumRegister,
                    BasicAer
from qiskit.visualization import plot_bloch_multivector,
                                 plot_state_city,
                                 plot_histogram,
                                 plot_state_qsphere,
                                 plot_state_hinton,
                                 plot_state_paulivec,
                                 visualize_transition
```

In order to run our code several times we need to invoke a backend. This can be achieved through `Aer` or `BasicAer`. The main difference between those two providers is that the former runs our code in C++ (faster) and the latter in Phyton.

In [25]:
```
Aer.backends()
```

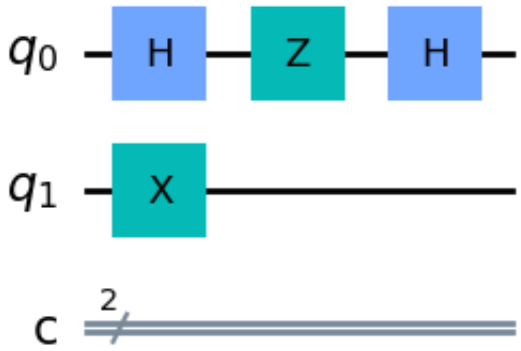
Out[25]:
```
[AerSimulator('aer_simulator'),
 AerSimulator('aer_simulator_statevector'),
 AerSimulator('aer_simulator_density_matrix'),
 AerSimulator('aer_simulator_stabilizer'),
 AerSimulator('aer_simulator_matrix_product_state'),
 AerSimulator('aer_simulator_extended_stabilizer'),
 AerSimulator('aer_simulator_unitary'),
 AerSimulator('aer_simulator_superop'),
 QasmSimulator('qasm_simulator'),
 StatevectorSimulator('statevector_simulator'),
 UnitarySimulator('unitary_simulator'),
 PulseSimulator('pulse_simulator')]
```

'qasm_simulator' may be used for 2D histograms visualization, 'statevector_simulator' for 3D representations and 'unitary_simulator' for matrix representations.

Let's say we want to run the following circuit in our backend and get the resulting state

In [51]:
```
qc = QuantumCircuit(2,2)
qc.h(0)
qc.z(0)
qc.h(0)
qc.x(1)
# qc.measure([0,1],[0,1])
qc.draw('mpl')
```

Out[51]:



To do so, the most common option is through the `execute` function. Let's get into the main inputs this function takes since it can be asked in the exam.

```
execute( experiments, backend, coupling_map = None,
optimization_level = None, shots = 1024, memory = False,
memory_slots = None, rep_time = None)
```

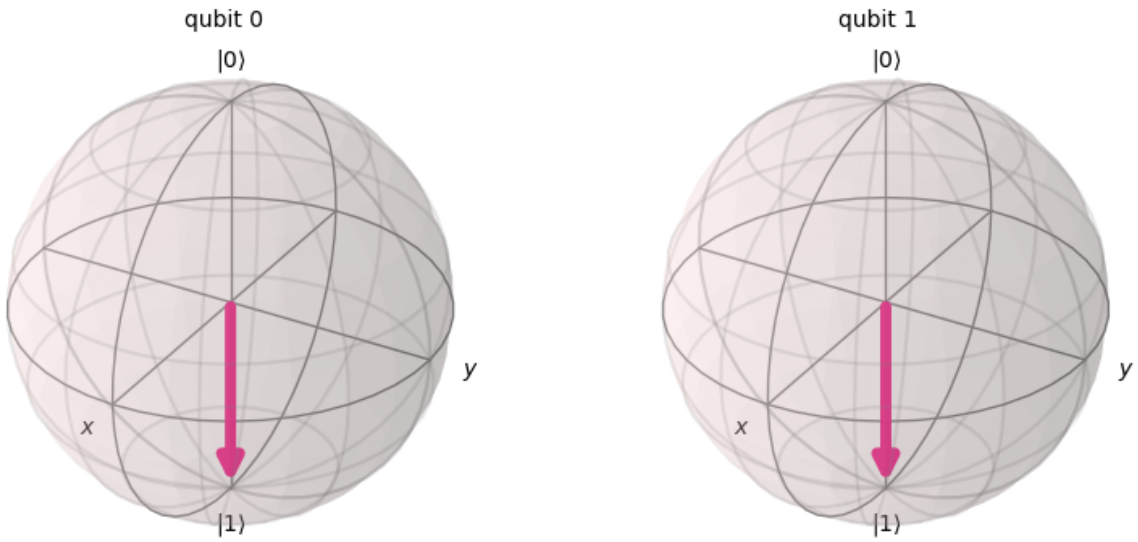
- **experiments** (QuantumCircuit) - circuit to execute.
- **backend** (BaseBackend or Backend) - backend to execute circuits on. Transpiler options are automatically grabbed from backend.configuration() and backend.properties().
- **coupling_map** (CouplingMap or list) - coupling map to target in mapping.
- **optimization_level** (int) - how much optimization to perform on circuits. Higher levels mean more optimized circuits at the expense of longer transpilation time.
- **shots** (int) - number of repetitions of each circuit. By default is 1024.
- **memory** (bool) - if True, per-shot measurement bitstrings are returned as well (provided the backend supports it).
- **memory_slots** (int) - Number of classical memory slots used in this job.
- **rep_time** (int) - time per program execution in seconds. Must be from the list provided by the backend (backend.configuration().rep_times).

This returns a `BaseJob`, from which we can get the result of the experiment using the `result()` method. Additionally, we can get the status of the job using the `status()` method or even the `done()` method to check whether the job has succesfully run.

In [52]:
```
simulator = Aer.get_backend('statevector_simulator')

result = execute(qc, backend = simulator, shots = 1024).result()

statevector = result.get_statevector(qc, decimals = 3)

print('\n The resulting quantum state is:', statevector)

plot_bloch_multivector(statevector)
```

```
 The resulting quantum state is: Statevector([0.+0.j, 0.+0.j, 0.+0.j, 1.+
0.j],
            dims=(2, 2))
```

Out[52]:



For a cleaner visualization of the statevector, we can use in Jupyter Notebook the `array_to_latex` function

In [56]:
```
from qiskit.visualization import array_to_latex

display(array_to_latex(result, prefix="\\text{Statevector} = "))
```

$$\text{Statevector} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

As expected, the final result is $|11\rangle = |1\rangle \otimes |1\rangle$. In matrix form, and using Kronecker's tensor product,

$$|11\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ 1 \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

which is precisely the statevector we got as a result of the simulation.

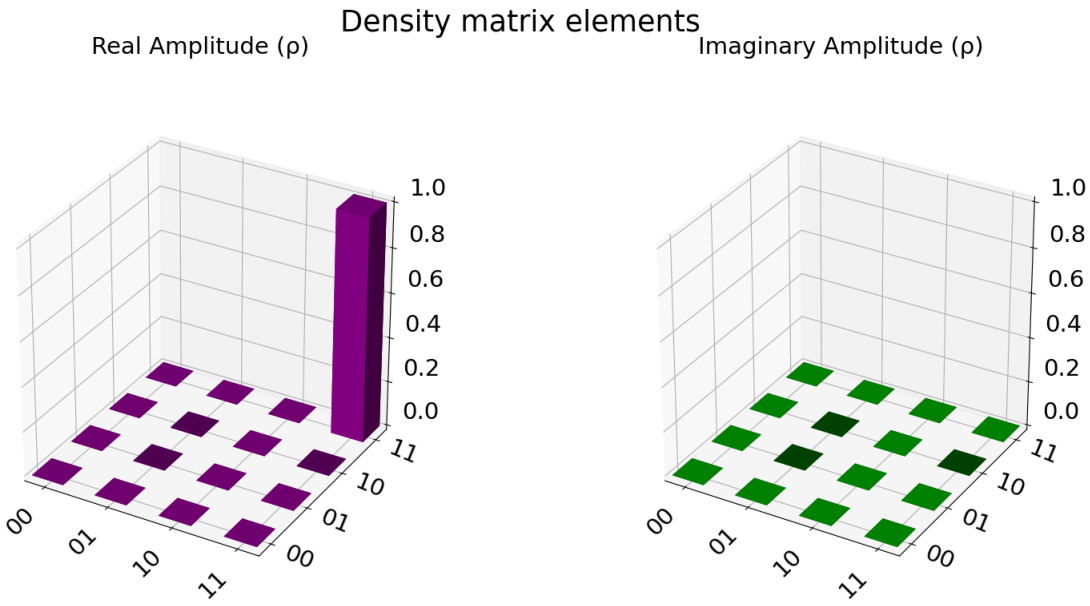
Once a backend has been selected, one may use `Backend.run(qob, backend_properties = None)` method as an alternative to the `execute` function. This method returns a result object aswell. Note that for the `run` method to work, we must first transpile the circuit.

In [ ]:
```
circuit = transpile(qc, simulator)
result = simulator.run(circuit).result()
```

We can also get a 3D representation of the matrix elements of the density operator ρ of the final state (see Density Operators), which in this case is $\rho = |11\rangle\langle 11|$, as well as a representation of the final state in the q-sphere.

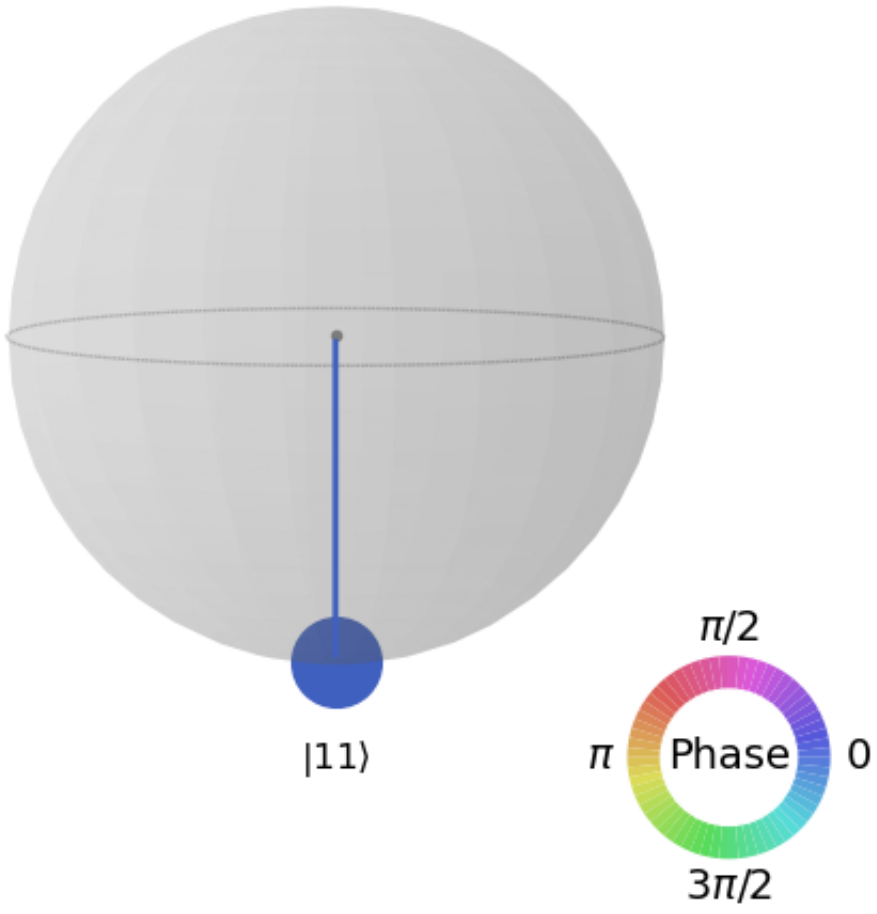
In [53]:
```
plot_state_city(qc, color = ['purple', 'green'],
                title = 'Density matrix elements')
```

Out[53]:



In [41]:
```
plot_state_qsphere(qc)
```

Out[41]:



We can get the unitary matrix that represents the circuit using the 'unitary_simulator'

In [54]:
```
backend = Aer.get_backend('unitary_simulator')

job = execute(qc, backend) # the number of shots is 1024 by default.

unitary = job.result().get_unitary(qc, decimals = 3)

display(unitary)
```

```
Operator([[ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
          [ 0.+0.j,  0.+0.j,  1.+0.j, -0.-0.j],
          [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
          [ 1.+0.j, -0.-0.j,  0.+0.j,  0.+0.j]],
         input_dims=(2, 2), output_dims=(2, 2))
```

In [55]:
```
from qiskit.visualization import array_to_latex

array_to_latex(unitary, prefix = '\\text{circuit} \\rightarrow \n')
```

Out[55]:

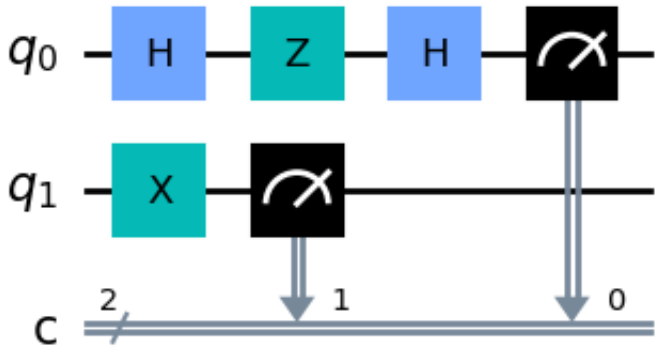
$$\text{circuit} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Note that in Qiskit notation, the circuit operations read as $X \left|q_1\right\rangle \otimes HZH \left|q_0\right\rangle$ and so the matrix representing the circuit is $X \otimes HZH$.

Finally, if we want to repeat our experiment a few times and get an estimate of the probabilities for the different outcomes, we can plot an histogram using `qasm_simulator`. Note that for results to be obtained, and for the histogram to be plotted, a measurement has to be added to the circuit.

In [51]:
```
qc = QuantumCircuit(2,2)
qc.h(0)
qc.z(0)
qc.h(0)
qc.x(1)
qc.measure([0,1],[0,1])
qc.draw('mpl')
```

Out[51]:

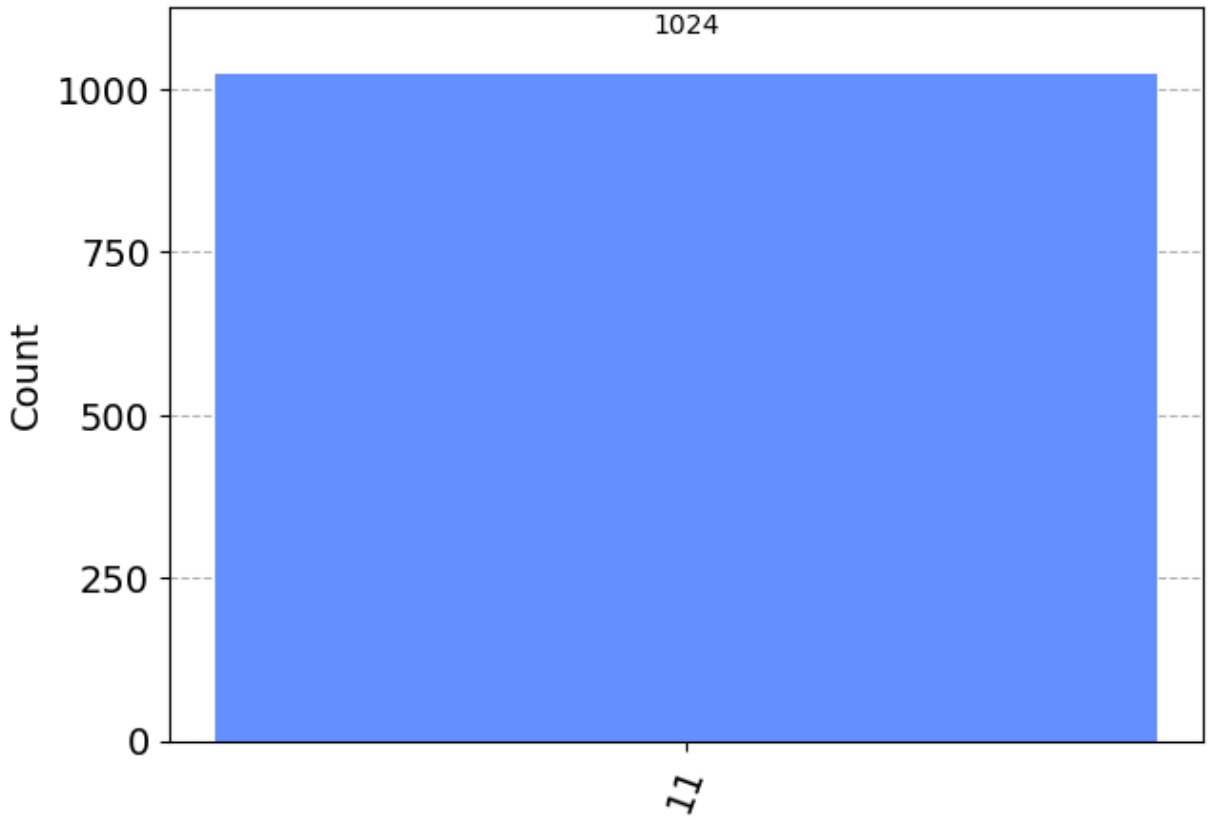


In [52]:

```
backend = Aer.get_backend('qasm_simulator')

job = execute(qc, backend, shots = 1024)

result = job.result()

counts = result.get_counts(qc)

print('\n The resulting state is:', counts)

plot_histogram(counts)
```

```
 The resulting state is: {'11': 1024}
```

Out[52]:



As seen, we got '11' as a result 1024 times out of the 1024 shots performed, i.e., with probability 1.

References: Udemy course, Nielsen and Chuang, quantumcomputinguk, the ultimate guide to quantum computing certification with qiskit, qiskit documentation, qiskit textbook (github repository)

Multi-qubit gates

back to the top

CNOT gate

Controlled gates act on 2 or more qubits, where one or more act as the control for some operation. $CNOT$ gates (also known as controlled Pauli-X or CX gates) act on two qubits and perform the logical NOT operation (i.e., flipping the bit) on the second qubit only when the first qubit is $|1\rangle$, otherwise it leaves it unchanged.

In the computational basis for two qubits $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ the $CNOT$ is represented as the following matrix

$$CNOT = |0\rangle\langle 0| \otimes 1 + |1\rangle\langle 1| \otimes X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Maps states $|a, b\rangle \rightarrow |a, a \oplus b\rangle$.

Remember that since Qiskit orders qubits in a different way, the matrix representation that it would yield would be

$$CNOT = 1 \otimes |0\rangle\langle 0| + X \otimes |1\rangle\langle 1| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Controlled gates

In general, if U is a gate that operates on a single qubit

$$U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix},$$

the controlled-U gate is a gate that operates on two qubits in such a way that the first acts as control and the second as target, mapping

$$|00\rangle \rightarrow |00\rangle$$

$$|01\rangle \rightarrow |01\rangle$$

$$|10\rangle \rightarrow |1\rangle \otimes U\,|0\rangle \rightarrow |1\rangle \otimes (u_{00}\,|0\rangle + u_{01}\,|1\rangle$$

$$|11\rangle \rightarrow |1\rangle \otimes U\,|1\rangle \rightarrow |1\rangle \otimes (u_{10}\,|0\rangle + u_{11}\,|1\rangle$$

In matrix form,

$$CU = |0\rangle\langle 0| \otimes 1 + |1\rangle\langle 1| \otimes U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{pmatrix}$$

In Qiskit,

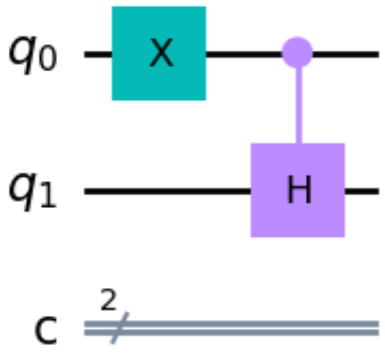
$$CU = 1 \otimes |0\rangle\langle 0| + U \otimes |1\rangle\langle 1| = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & u_{00} & 0 & u_{01} \\ 0 & 0 & 1 & 0 \\ 0 & u_{10} & 0 & u_{11} \end{pmatrix}$$

When U is X, Y, Z, H, we get controlled-X (CX), controlled-Y (CY), controlled-Z (CZ), and controlled-Hadamard (CH), respectively. We can also obtain a controlled phase gate (CP) using P as U.

Let's visualize the effect of a simple circuit implementing a CH:

In [5]:
```
qc = QuantumCircuit(2,2)
qc.x(0)
qc.ch(0,1) # cx, cy, cz, ch for other controlled gates.
# First input is control and second, target.
qc.draw('mpl')
```

Out[5]:

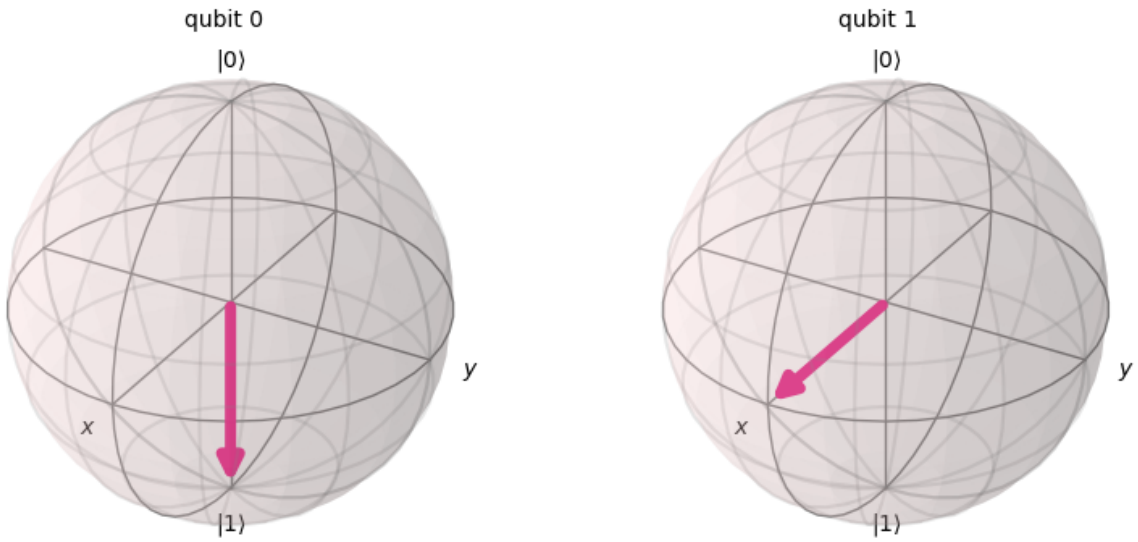


In [14]:

```
simulator = Aer.get_backend('statevector_simulator')

result = execute(qc, backend = simulator).result()

statevector = result.get_statevector(qc, decimals = 3)

print('\n The resulting quantum state is:', statevector)

plot_bloch_multivector(statevector)
```

```
 The resulting quantum state is: Statevector([0.   +0.j, 0.707-0.j, 0.
+0.j, 0.707+0.j],
            dims=(2, 2))
```

Out[14]:

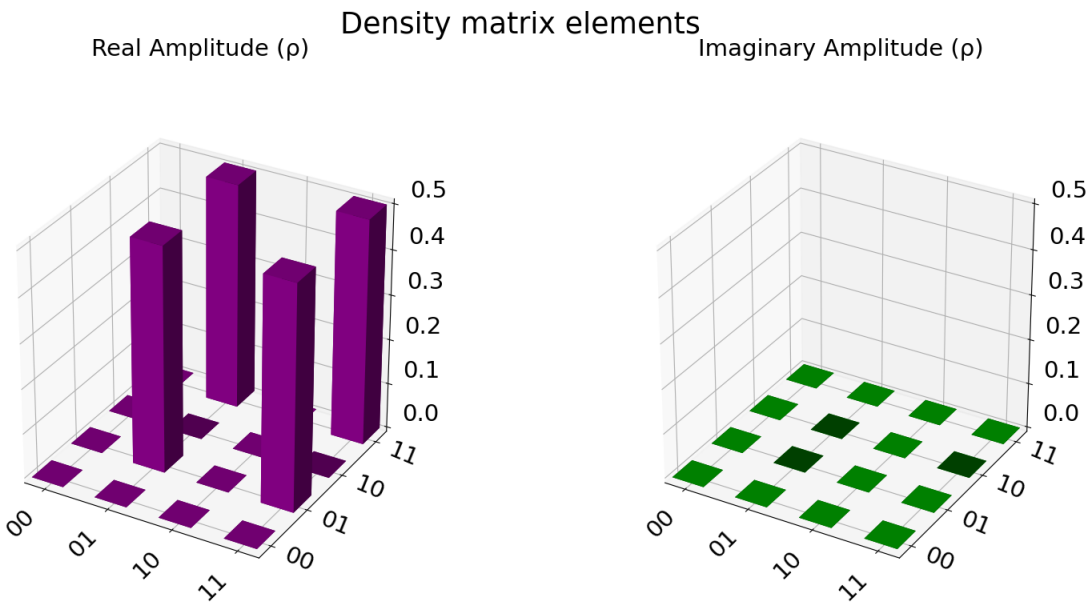


Remember that Qiskit orders qubits in a cascading fashion; whereas for us the resulting state is $|q_0 q_1\rangle = |1\rangle \otimes |+\rangle = (|10\rangle + |11\rangle)/\sqrt{2}$, Qiskit orders this state as $|q_1 q_0\rangle = (|01\rangle + |11\rangle)/\sqrt{2}$, which may be expressed in the computational basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, in matrix form as $(0, 1, 0, 1)$ (up to a global phase).

For the density matrix, which involves the outer product of the result, i.e. $\rho = |+1\rangle \langle +1|$ we would get a 4×4 matrix of zeros with a 0.5 on the $\rho_{11}, \rho_{13}, \rho_{31}, \rho_{33}$ positions.

In [15]:
```
plot_state_city(qc, color = ['purple', 'green'],
                title = 'Density matrix elements')
```

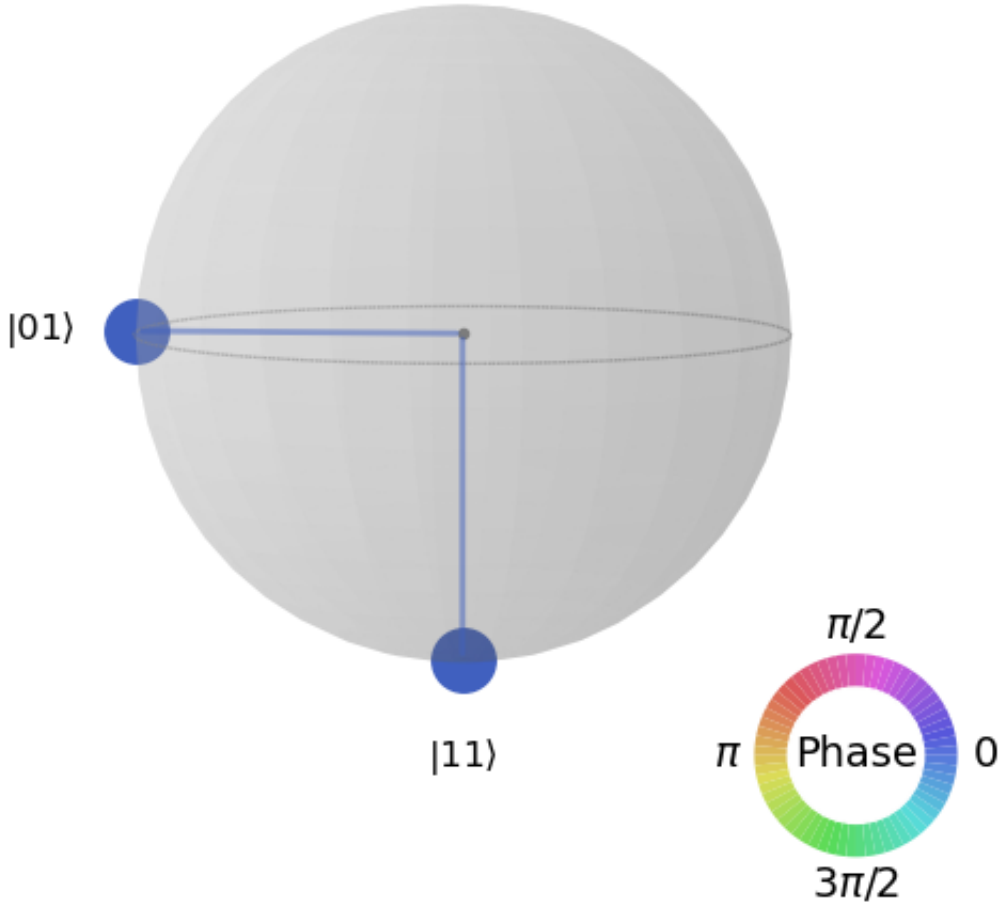
Out[15]:



Finally, the q-sphere would point to both $|01\rangle$ and $|11\rangle$

In [16]:
```
plot_state_qsphere(qc)
```

Out[16]:

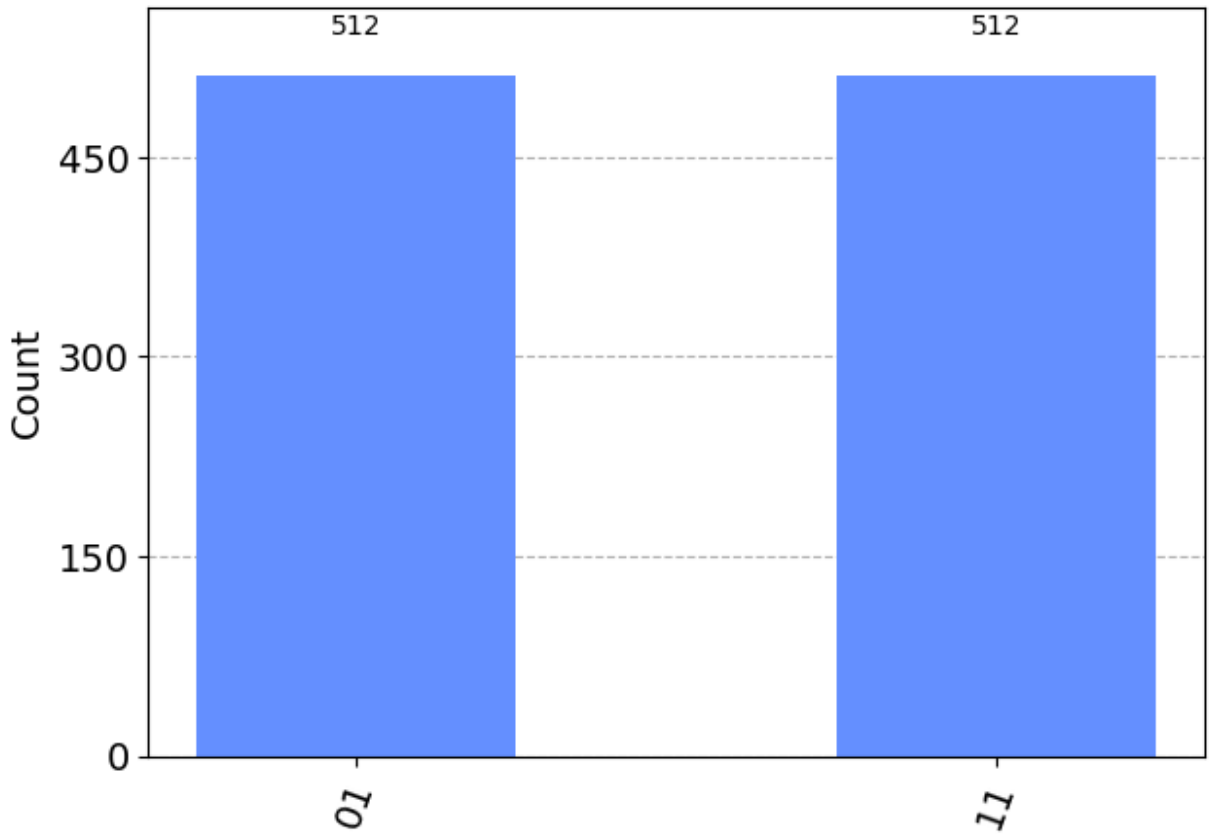


And if we were to measure these two qubits and repeat the experiment several times, onee would expect to get '01' half of the time and '11' on the other.

In [18]:
```
qc = QuantumCircuit(2)

qc.x(0)

qc.ch(0,1)

qc.measure_all()

backend = Aer.get_backend('qasm_simulator')

job = execute(qc, backend, shots = 1024)

result = job.result()

counts = result.get_counts(qc)

print('\n The resulting state is:', counts)

plot_histogram(counts)
```

```
 The resulting state is: {'11': 512, '01': 512}
```

Out[18]:



SWAP gate

It swaps the state of two qubits. It is equivalent to performing the sequence $CNOT_{12}CNOT_{21}CNOT_{12}$, where the first subscript indicates the control and the second, the target.

$$|a,b\rangle \rightarrow |a,a\oplus b\rangle \rightarrow |a\oplus(a\oplus b),a\oplus b\rangle = |b,a\oplus b\rangle \rightarrow |b,(a\oplus b)\oplus b\rangle = |b,a\rangle$$

In matrix form,

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

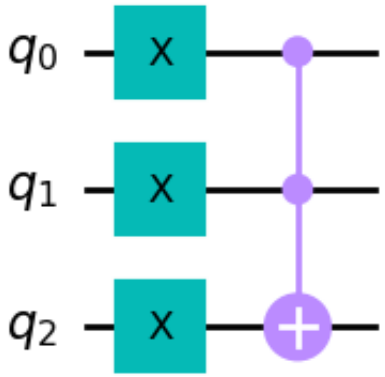
Toffoli, CCX or CCNOT gate

It is a three qubit gate that applies a Pauli-X (or NOT) if the first two bits are in the state $|1\rangle$, else it does nothing. Its matrix representation can be readily obtained noticing that $CCNOT = |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes I \otimes X$, where the product gates are single qubit ones (resulting in a 8×8 matrix).

Let's take a look at how Qiskit plots three-qubits information with a simple circuit.

In [24]:
```
qc = QuantumCircuit(3)
qc.x(0)
qc.x(1)
qc.x(2)
qc.ccx(0,1,2) # ccx takes three inputs: first two are the controls
              # and the last one is the target.
qc.draw('mpl')
```

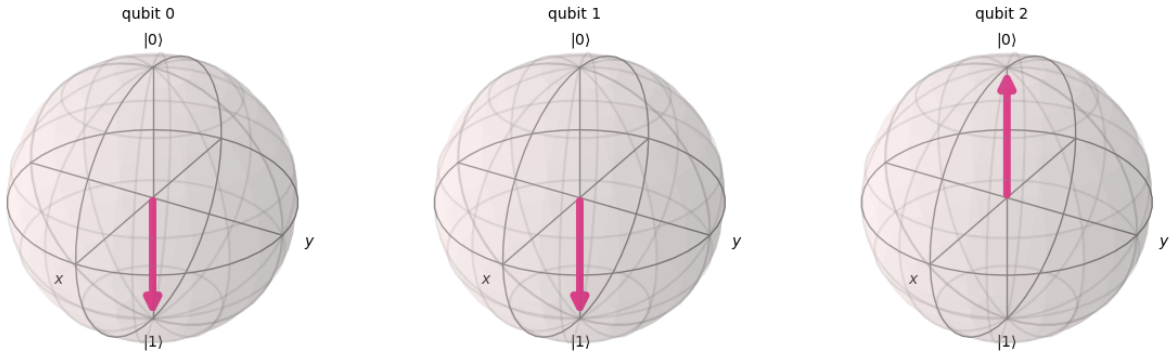
Out[24]:



In [25]:
```
simulator = Aer.get_backend('statevector_simulator')

result = execute(qc, backend = simulator).result()

statevector = result.get_statevector(qc, decimals = 3)

print('\n The resulting quantum state is:', statevector)

plot_bloch_multivector(statevector)
```

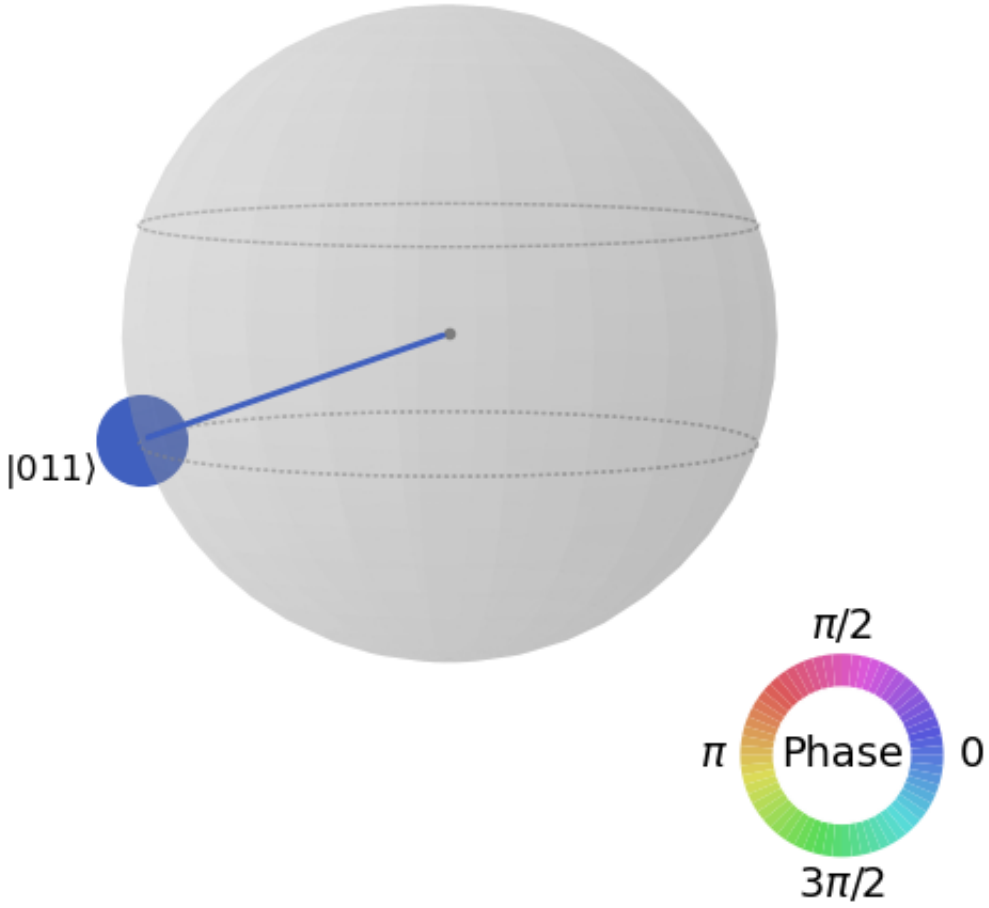
```
 The resulting quantum state is: Statevector([0.+0.j, 0.+0.j, 0.+0.j, 1.+
0.j, 0.+0.j, 0.+0.j, 0.+0.j,
             0.+0.j],
            dims=(2, 2, 2))
```

Out[25]:



In [26]:
```
plot_state_qsphere(qc)
```

Out[26]:



CSWAP or Fredkin gate

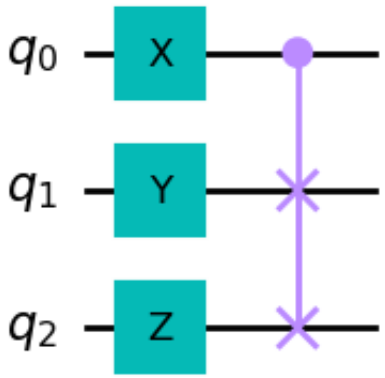
It is a three-qubit gate that exchanges the states of two target qubits if the control is in the state $|1\rangle$, otherwise it does nothing.

In the computational basis,

$$CSWAP = |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes SWAP$$

In [6]:
```
qc = QuantumCircuit(3)
qc.x(0)
qc.y(1)
qc.z(2)
qc.cswap(0,1,2) # can also be implemented with fredkin() method
qc.draw('mpl')
```

Out[6]:

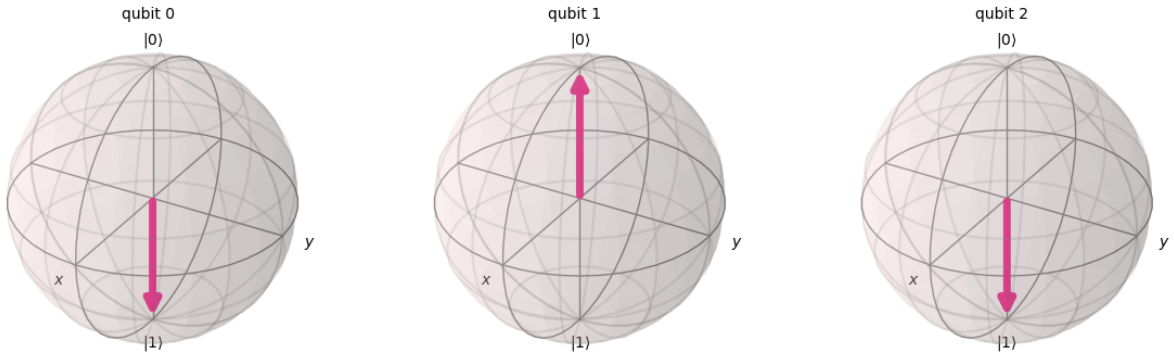


In [3]:

```
simulator = Aer.get_backend('statevector_simulator')

result = execute(qc, backend = simulator).result()

statevector = result.get_statevector(qc, decimals = 3)

print('\n The resulting quantum state is:', statevector)

plot_bloch_multivector(statevector)
```

```
 The resulting quantum state is: Statevector([ 0.-0.j,  0.-0.j,  0.+0.j,
0.+0.j,  0.+0.j,  0.+1.j, -0.+0.j,
             -0.+0.j],
            dims=(2, 2, 2))
```

Out[3]:



CNOT, CZ (CPHASE), SWAP and CSWAP are the only involutory multi-qubit gates.

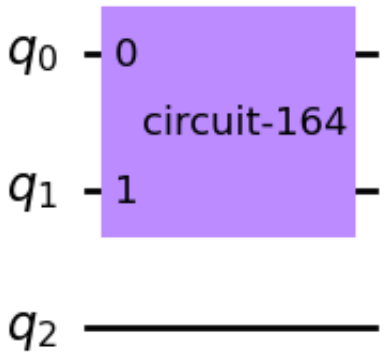
All involutory gates are also hermitian

Custom gates

We can 'pack' several gates into a custom gate that may be saved to add later to another circuit. For example, we can merge together `h(0)` and `cx(0,1)` as a `bell_gate` (see Bell_states).

In [15]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)

bell_gate = qc.to_gate()

qc2 = QuantumCircuit(3)
qc2.append(bell_gate,[0,1])
qc2.draw('mpl')
```

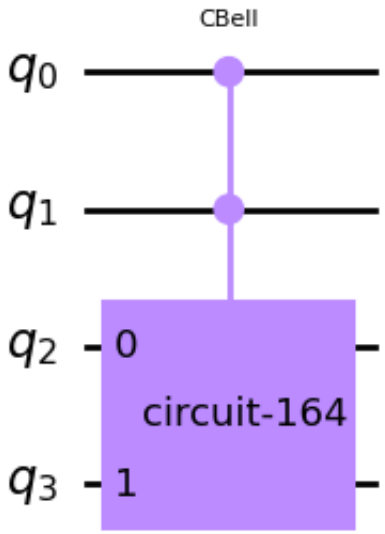
Out[15]:



Additionally, we can convert any `Gate` into a control gate using the `control` method and even customize the number of control qubits.

In [23]:
```
cbell = bell_gate.control(num_ctrl_qubits = 2, label = 'CBell')

qc3 = QuantumCircuit(4)
qc3.append(cbell, [0,1,2,3]) # 0,1 are the control and 2,3 the target.
qc3.draw('mpl')
```

Out[23]:



Non-unitary operations

back to the top

These include measurements, reset of qubits and conditional operations.

Measurements

`measure(qubit, cbit)` - measures a quantum bit in the Z basis into a classical bit. Inputs can be given as *int*, *slice*, *Quantum* or *ClassicalRegisters* or *list*. For example, all the `measure` statements below are equivalent.

In [1]:
```
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

qc.measure(q,c)
#
qc.measure(q[0],c[0])
qc.measure(q[1],c[1])
#
qc.measure([0,1],[0,1])
#
qc.measure(0,0)
qc.measure(1,1)
#
qc.measure(-1,-1)
qc.measure(-2,-2)
```

Out[1]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x7fa2d3f1aa60>
```

Reset

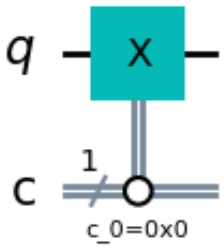
The `reset(qubit)` simply resets the qubit to its default state. Qubit(s) can be inputted in the same formats available for the `measure()` method.

Conditionals

Allows us to perform operations on qubits depending on the value of the classical bits. For example,

In [5]:
```
qc = QuantumCircuit(1,1)
qc.x(0).c_if(0,0) #applies X to q_0 if c_0 = 0
qc.draw('mpl')
```

Out[5]:



Phase kickback

back to the top

Any unitary operator U can be written in its eigenbasis $\{|\psi_j\rangle\}$ as

$$U = \sum_k e^{i\phi_k} |\psi_k\rangle \langle\psi_k|$$

where the ϕ_k satisfy $U|\psi_k\rangle = e^{i\phi_k}|\psi_k\rangle$, for all k. For example, the X gate can be written in its eigenbasis $\{|+\rangle, |-\rangle\}$ as

$$X = e^{i\cdot 0}|+\rangle\langle+| + e^{i\pi}|-\rangle\langle-| = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

This phase is typically a global one that has no observable effects. Consider now the controlled-X operation. We know that if the control qubit is on either $|0\rangle$ or $|1\rangle$, CX simply adds a global phase

$$CX|-0\rangle = |-0\rangle$$

$$CX|-1\rangle = X|-\rangle \otimes |1\rangle = -|-\rangle \otimes |-0\rangle = -|-1\rangle$$

Now, if the control qubit is in a superposition of these two, the situation is different,

$$CX\ |-+\rangle = \frac{1}{\sqrt{2}}(CNOT|-0\rangle + CNOT|-1\rangle) =$$

$$= \frac{1}{\sqrt{2}}(|-0\rangle + (X \otimes I)\ |-1\rangle) =$$

$$= \frac{1}{\sqrt{2}}(|-0\rangle - |-1\rangle) =$$

$$= |-\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) =$$

$$= |--\rangle$$

.

Here, the control qubit started in the $|+\rangle$ state and, due to the phase factor addeed to the target qubit, ended up in the $|-\rangle$ state, all while leaving the target qubit unchanged.

This can be done with any arbitrary unitary gate. As a more general case, consider the P-gate

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

which, as we already know, maps $|1\rangle \rightarrow e^{i\theta}|1\rangle$. Now we define the controlled-P gate, which in textbook notation is

$$CP(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix}$$

Setting our control qubit in a superposed state $|+\rangle$ and applying this gate to the state $|1\rangle$ we get

$$CP(\theta)\ |1+\rangle = \frac{1}{\sqrt{2}} CP(\theta)\ (|10\rangle + |11\rangle) = \frac{1}{\sqrt{2}} \Big(|10\rangle + e^{i\theta}\ |11\rangle \Big) =$$

$$= |1\rangle \otimes \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i\theta}\ |1\rangle \Big)$$

.

This process, where the eigenvalue of the gate, which is in principle being applied to one qubit, is 'kicked back' into a different one via a controlled operation is known as phase kickback.

Analogies between classical and quantum logic

back to the top

Buffer

Output has the same value as input. Its truth table then is

Input	Output
0	0
1	1

This can be done in a quantum circuit with an identity gate I.

NOT (Inverter)

Inverts the input, so the output is $\bar{A}$ or $\neg A$.

Input	Output
0	1
1	0

This is precisely what the Pauli-X gate does.

AND

Returns a 1 if and only if both inputs hold a value of 1. Else, it returns a 0. The output in boolean logic is $A \cdot B$ or $A \wedge B$.

Input 1 (A)	Input 2 (B)	Output
0	0	0
0	1	0
1	0	0
1	1	1

This can be achieved with a Toffoli gate with A and B as controls and the thrid one as target initialized in the $|0\rangle$ state. In a general case, the Toffoli gate performs the mapping $|a, b, c\rangle \rightarrow |a, b, c \oplus (a \wedge b)\rangle$ so it works as a combination of XOR and AND operations. Setting $c = 0$ allows the implementation of only the AND operation.

NAND (NOT AND)

Works in the exact opposite way as the AND gate. As result, it can be easily implemented on a QC by applying an X gate (NOT operation) to the thrid (output) qubit after the Toffoli gate.

Input 1 (A)	Input 2 (B)	Output
0	0	1
0	1	1
1	0	1
1	1	0

XOR (exclusive OR)

It returns a 1 if both inputs are different. Else, it returns a zero. Logically, it outputs $A \oplus B$.

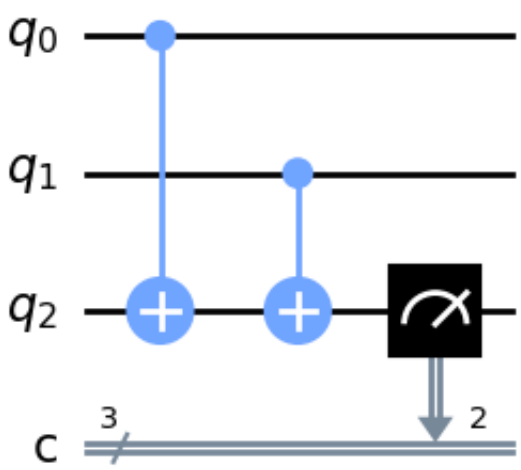
Input 1 (A)	Input 2 (B)	Output
0	0	0
0	1	1
1	0	1
1	1	0

If we read this table as the two input states and the after-operation target state, this is precisely what a $CNOT$ does, in fact it maps $|a, b\rangle \rightarrow |a, a \oplus b\rangle$, where $\oplus$ is the classical XOR operation, representing a mod 2 sum.

If we wanted to implement this classical gate as a three-qubit circuit, we would need to add an extra $CNOT$ to it (see circuit below).

In [3]:
```
import qiskit
from qiskit import QuantumCircuit

qc = QuantumCircuit(3,3)
qc.cx(0,2)
qc.cx(1,2)
qc.measure(2,2)
qc.draw('mpl')
```

Out[3]:



OR

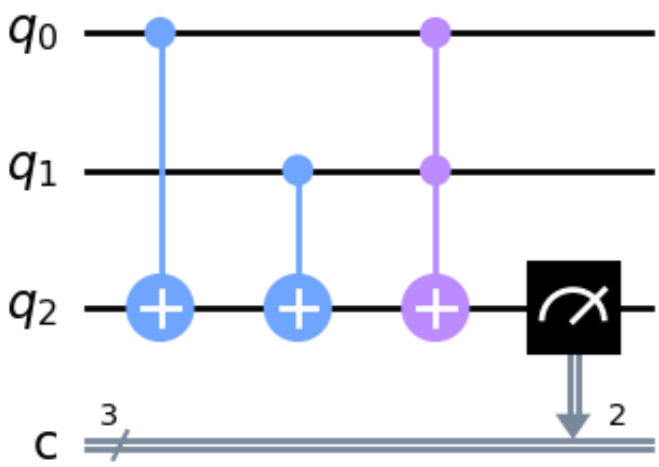
It returns a zero if and only if both inputs are zero. Else, it returns a one. In boolean logic it outputs $A + B$ or $A \vee B$.

Input 1 (A)	Input 2 (B)	Output
0	0	0
0	1	1
1	0	1
1	1	1

This truth table is similar to the XOR one, except for the last case. As a result, we can use the same circuit as above but we need to flip the last qubit at the end if both inputs are 1. This can be achieved with a Toffoli gate.

In [6]:
```
qc = QuantumCircuit(3,3)

qc.cx(0,2)
qc.cx(1,2)
qc.ccx(0,1,2)
qc.measure(2,2)
qc.draw('mpl')
```

Out[6]:



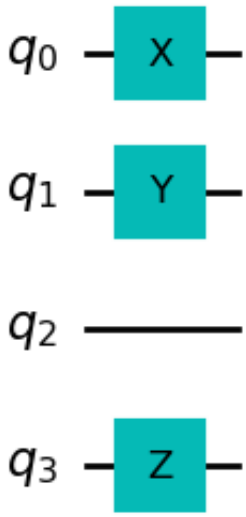
Performing operations among quantum circuits

back to the top

To combine circuits, use the `compose` method. It is important to note that by default, this method does not modify the original circuits being composed but rather creates a new one. If one wishes to modify the circuit on which we are calling this method, set `inplace = True`.

In [8]:
```
qc_a = QuantumCircuit(4)
qc_a.x(0)

qc_b = QuantumCircuit(2)
qc_b.y(0)
qc_b.z(1)

# compose qubits (0, 1) of qc_a to qubits (1, 3) of qc_b respectively

combined = qc_a.compose(qc_b, qubits=[1, 3])
combined.draw("mpl")
```

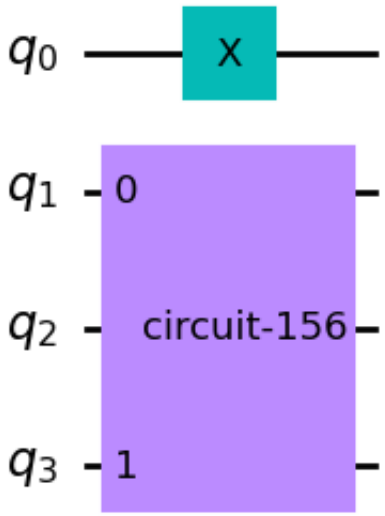
Out[8]:



One can also convert a `QuantumCircuit` into an `instruction` object, to keep circuits organized, with the `to_instruction` method and then append it to another circuits. For example, with the circuits above we can achieve the same output by running the following lines

In [11]:
```
instruction = qc_b.to_instruction()
qc_a.append(instruction, [1,3])
qc_a.draw('mpl')
```

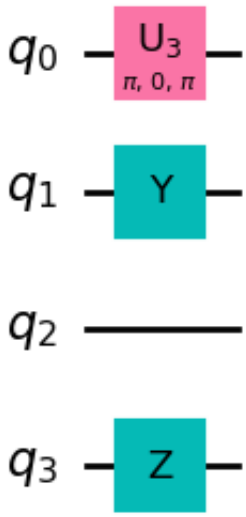
Out[11]:



Now imagine we only got this circuit and had no clue about the construction of it. Maybe it would be helpful to understand what is inside the purple box. To do so, one may use the `decompose()` method, which expands each instruction into its definition. This method is specially useful in custom gates.

In [12]:
```
qc_a.decompose().draw('mpl')
```

Out[12]:



Bell states

back to the top

They are four specific maximally entangled quantum states of two qubits. The result of a measurement of a single qubit in a Bell state is indeterminate unless such measurement is done in the $\mathbf{z}$-basis. In this particular case, the result of measuring the second qubit is guaranteed to yield the same value (for ϕ states) or the opposite value (for Ψ states). This implies that the measurement outcomes are correlated.

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

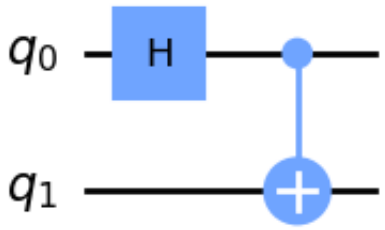
$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

NOTE: Maximally entanglement here refers to the fact that the reduced density operator of the one subsystem is maximally mixed ($\rho_0 = \rho_1 = I/2$).

Although there are different ways of creating a Bell pair, the simplest one involves a combination of an H and $CNOT$ gate.

In [25]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.draw('mpl')
```

Out[25]:

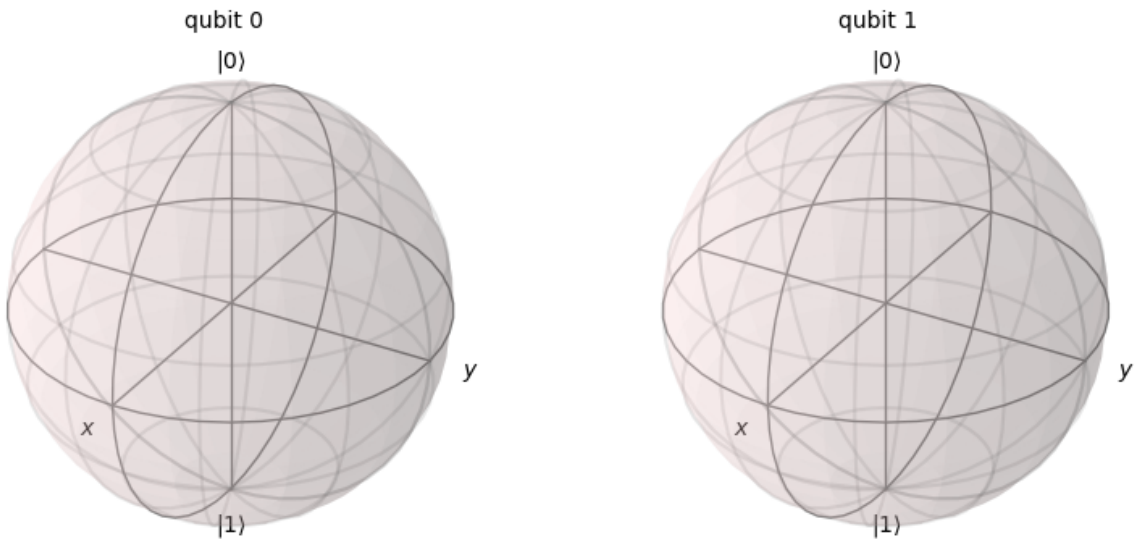


A Bloch sphere representation of a Bell pair is not possible due to the entangled nature of it.

In [7]:
```
simulator = Aer.get_backend('statevector_simulator')

result = execute(qc, backend = simulator).result()

statevector = result.get_statevector(qc, decimals = 3)

print('\n The resulting quantum state is:', statevector)

plot_bloch_multivector(statevector)
```

```
 The resulting quantum state is: Statevector([0.707+0.j, 0.   +0.j, 0.
+0.j, 0.707+0.j],
            dims=(2, 2))
```

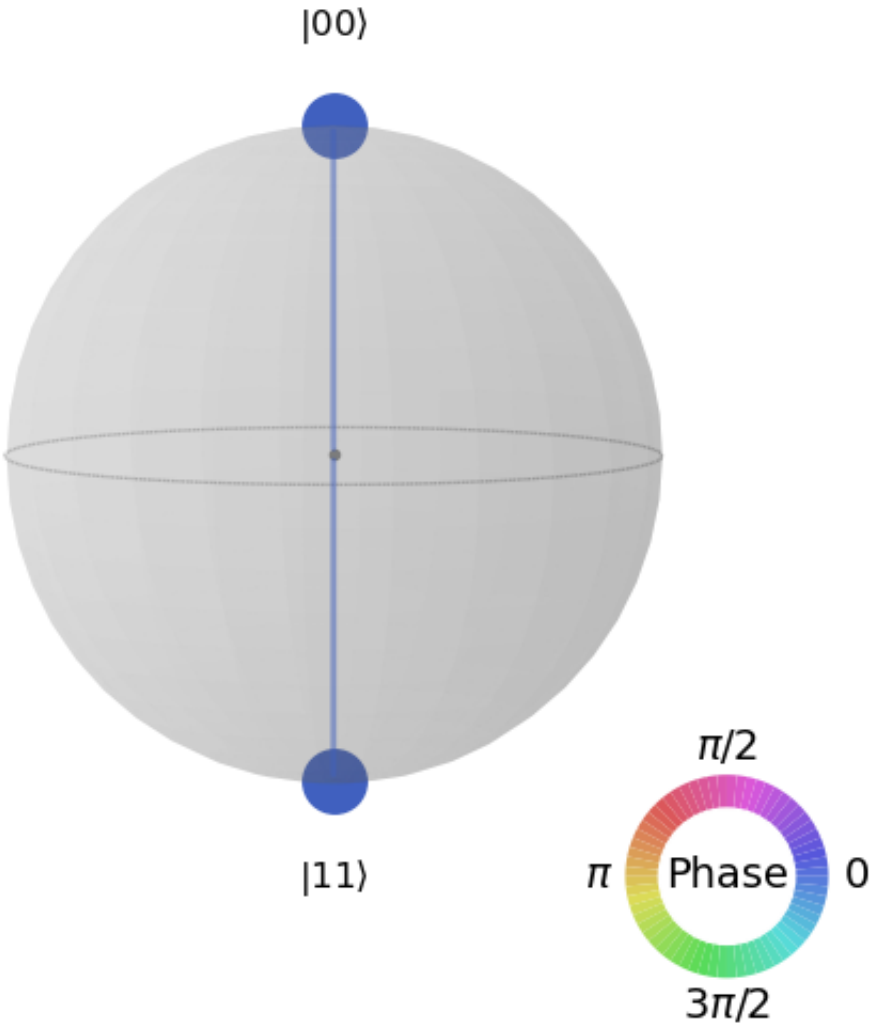
Out[7]:



Although one can still get a qsphere representation

In [8]:
```
plot_state_qsphere(qc)
```

Out[8]:



We can also get the resulting state from the circuit and display it in `latex` format

In [26]:
```
from qiskit.quantum_info import Statevector

phi_plus = Statevector.from_instruction(qc)
phi_plus.draw('latex', prefix = '|\\Phi^{+} \\rangle = ')
```

Out[26]:

$$|\Phi^+\rangle = \frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$$

GHZ state

back to the top

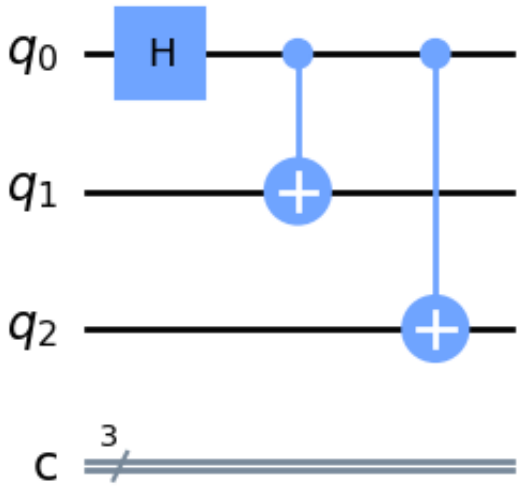
A Greenberg-Horne-Zeilinger (GHZ) state is an entangled state that involves at least three qubits. For three qubits,

$$|GHZ\rangle = \frac{|000\rangle + |111\rangle}{\sqrt{2}}$$

This can be easily obtained in Qiskit adding an extra $CNOT$ to a Bell entanglement circuit

In [9]:
```
qc = QuantumCircuit(3,3)
qc.h(0)
qc.cx(0,1)
qc.cx(0,2)

qc.draw('mpl')
```

Out[9]:



In [10]:
```
Statevector.from_instruction(qc).draw('latex')
```

Out[10]:

$$\frac{\sqrt{2}}{2}|000\rangle + \frac{\sqrt{2}}{2}|111\rangle$$

Getting circuit properties

back to the top

Depth

Is a measure of how many layers of quantum gates, executed in parallel, it takes to complete the computation defined by the circuit. Mathematically, it is defined as the longest path in a directed acyclic graph (DAG). Although still hard to visualize by just looking at the circuit, the following link might be helpful https://docs.quantum.ibm.com/api/qiskit/circuit. Barriers do not count towards computing a circuit's depth but measurements do.

Size and number of gates.

It is the total number of instructions in a circuit. Again, barriers do not count towards computing a circuit's size but measurements still do. We can also get the number and type of the gates in a circuit using `count_ops()` method. Note in the example down below that this method also counts the number of barriers.

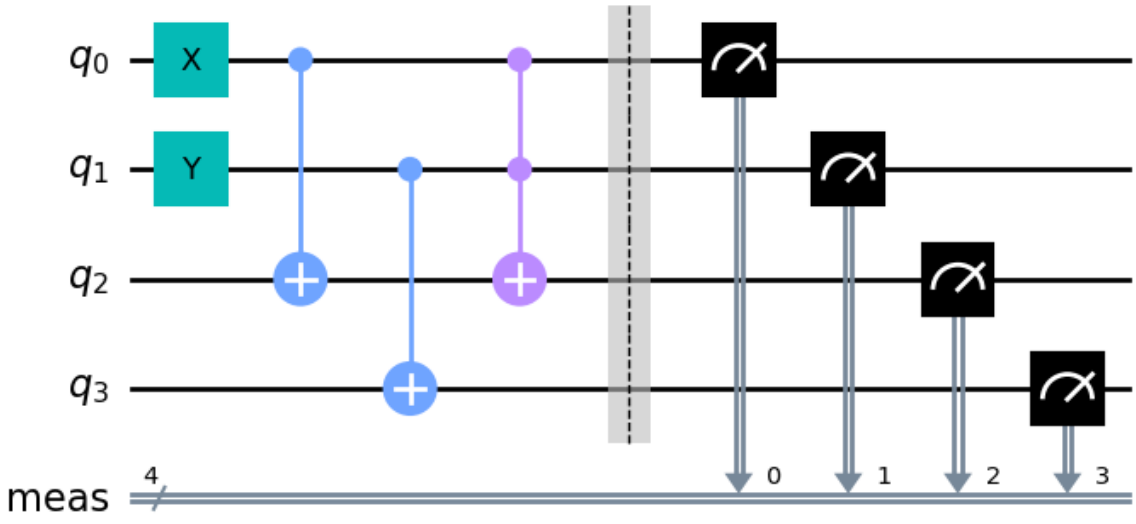
Number of qubits and width

For a quantum circuit composed from just qubits, the circuit width is equal to the number of qubits. However, for more complicated circuits with classical registers, this equivalence breaks down. In the example down below, classical registers are needed to perform the measurements, therefore the circuit's width increases from 4 to 8 in this last layer. In short, width returns the number of qubits, classical bits and measurement registers involved in our quantum circuit.

In [24]:
```
qc = QuantumCircuit(4)
qc.x(0)
qc.y(1)
qc.cx(0,2)
qc.cx(1,3)
qc.ccx(0,1,2)
qc.measure_all()

depth = qc.depth()
size = qc.size()
width = qc.width()
num_qubits = qc.num_qubits
count_ops = qc.count_ops()

print('\n The circuit depth is:', depth)
print('\n The circuit size is:', size)
print('\n The circuit width is:', width)
print('\n The number of qubits is:', num_qubits)
print('\n The number and type of gates is:', count_ops)


qc.draw('mpl')
```

```
 The circuit depth is: 4

 The circuit size is: 9

 The circuit width is: 8

 The number of qubits is: 4

 The number of gates is: OrderedDict([('measure', 4), ('cx', 2), ('x',
1), ('y', 1), ('ccx', 1), ('barrier', 1)])
```

Out[24]:



Refs: Nielsen and Chuang, IBM Quantum Documentation, IBM Qiskit Textbook, Wikipedia, Udemy course.

Plotting states in the Bloch sphere

back to the top

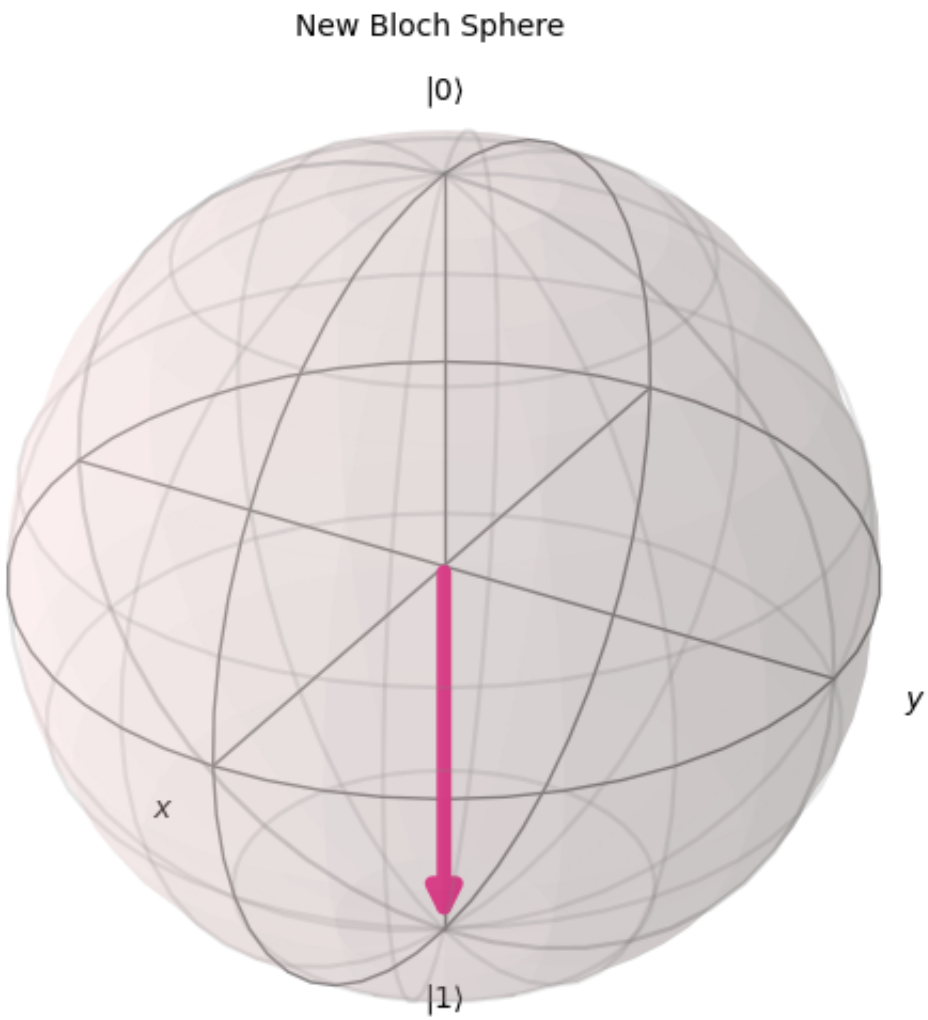
The command `plot_bloch_vector` can be used to plot a Bloch vector in Bloch sphere with its specified coordinates, that can be given in both cartesian and spherical systems. The general syntax is as follows:

```
plot_bloch_vector(bloch, title = '', ax = None, figsize = None,
coord_type = 'cartesian', font_size = None)
```

- **bloch**(*list*) - array of three elements $[x, y, z]$ (cartesian) or $[r, \theta, \varphi]$ (spherical in radians). θ is the inclination angle from the z direction and φ is the azimuth from the x direction.
- **title**(*str*) - a string that represents the plot title.
- **ax**(*matplotlib.axes.Axes*) - an Axes to use for rendering the bloch sphere.
- **figsize**(*tuple*) - figure size in inches. Has no effect if passing `ax`.
- **coord_type**(*str*) - a string that specifies coordinate type for bloch (Cartesian or spherical), default is Cartesian.
- **font_size** (*float*) - font size

In [28]:
```
from qiskit.visualization import plot_bloch_vector

plot_bloch_vector([0,0,-1], title="New Bloch Sphere")
```

Out[28]:



Note that since Bloch sphere has radius unity, every state vector with norm 1 will touch the surface of the sphere, while the rest (with norm greater or lower than 1) will lay outside or inside it.

Refs: course, plot_bloch vector in IBM Quantum Documentation

Reading and writing in a QASM file

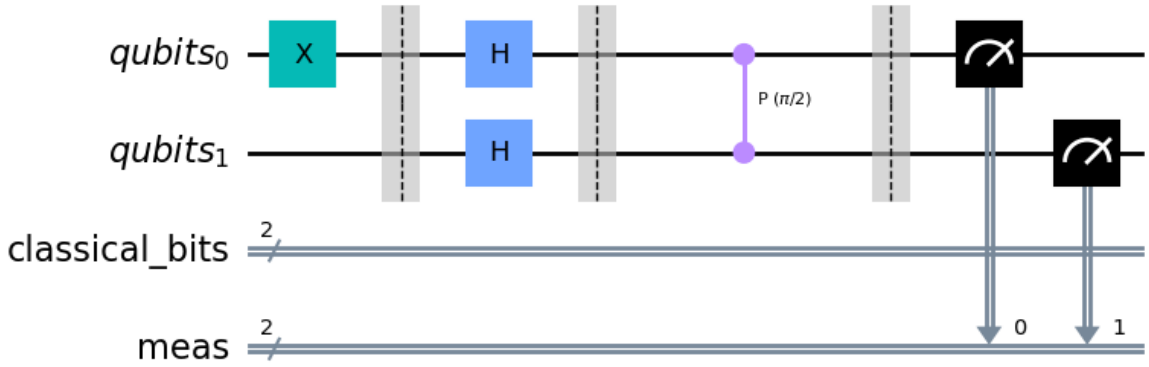
back to the top

QASM is a quantum computing programming language more formally called OpenQASM, which stands for Open Quantum Assembly language. Everything we write in our code in Qiskit is internally converted into QASM. We can export directly our coded circuit into a QASM file using the `qasm()` method

In [40]:
```
quantum_register = QuantumRegister(2, 'qubits')
classical_register = ClassicalRegister(2, 'classical_bits')
qc = QuantumCircuit(quantum_register, classical_register)
qc.x(0)
qc.barrier()
qc.h([0,1])
qc.barrier()
qc.cp(pi/2,0,1)
qc.measure_all()

str = qc.qasm()

print(str)  # qc.qasm() returns a str

qc.draw('mpl')
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg qubits[2];
creg classical_bits[2];
creg meas[2];
x qubits[0];
barrier qubits[0],qubits[1];
h qubits[0];
h qubits[1];
barrier qubits[0],qubits[1];
cp(pi/2) qubits[0],qubits[1];
barrier qubits[0],qubits[1];
measure qubits[0] -> meas[0];
measure qubits[1] -> meas[1];
```

Out[40]:

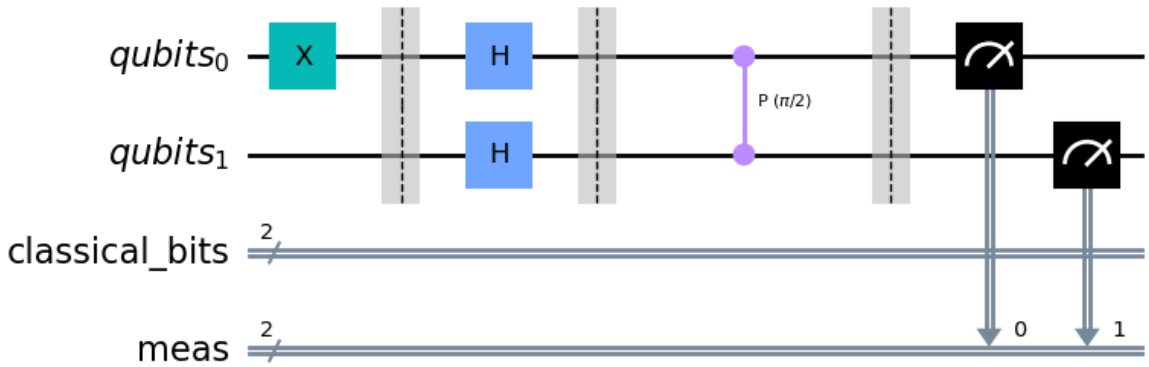


Additionally, one can import a QASM string or file and use it to build a quantum circuit.

In [42]:

```
# Loading from a QASM string

qc_1 = QuantumCircuit.from_qasm_str(str)

qc_1.draw('mpl')

# If we wanted to load such str from a text file,

# qc = QuantumCircuit.from_qasm_file(filename)
```

Out[42]:



Fidelity

back to the top

It is a measure of the distance between two quantum states. It expresses the probability that one state will pass a test to identify as the other.

The fidelity of states represented by density operators ρ, σ is given by

$$F(\rho, \sigma) = \operatorname{tr}\sqrt{\rho^{1/2}\sigma\rho^{1/2}}$$

For any ρ, σ, $0 \leq F(\rho, \sigma) \leq 1$, $F(\rho, \rho) = 1$ and $F(\rho, \sigma) = 0$ if the states are completely different, i.e. orthogonal. In particular, for pure states, $\rho = |\psi_\rho\rangle \langle\psi_\rho|$ and $\sigma = |\psi_\sigma\rangle \langle\psi_\sigma|$, fidelity simplifies to

$$F(\rho, \sigma) = |\langle\psi_\rho|\psi_\sigma\rangle|^2$$

In this last expression, one realises that fidelity may be interpreted as the probability of finding the state $|\psi_\rho\rangle$ when measuring $|\psi_\sigma\rangle$ in a basis containing $|\psi_\rho\rangle$.

If ρ and σ are both qubit states, ρ and σ are represented by two-dimensional matrices and

$$F(\rho, \sigma) = \operatorname{tr}(\rho\sigma) + 2\sqrt{\det(\rho)\det(\sigma)}.$$

Additionally, if ρ (or σ) is pure, the above expression simplifies further to $F(\rho, \sigma) = \operatorname{tr}(\rho\sigma)$ since $\det(\rho) = 0$ for pure states.

Now regarding quantum computing in particular, fidelity might be used as a measure of noise affecting your hardware, as it can measure how close you can expect your quantum code to execute compared to when running it on a physical quantum hardware. In this sense, Qiskit offers various methods for measuring fidelity,

- **State fidelity**: distance between two statevectors or density operators.
- **Process fidelity**: measures the fidelity of a quantum channel or operator.
- **Average gate fidelity**: measure the fidelity of multiple gates within a quantum channel.

State fidelity

Let's first see that two circuits using the same gates output the same state, i.e. fidelity equals 1.

In [6]:
```
from qiskit.quantum_info import state_fidelity

qc1 = QuantumCircuit(2)
qc1.x(0)
qc1.x(1)

qc2 = QuantumCircuit(2)
qc2.x([0,1])

simulator = Aer.get_backend('statevector_simulator')
result1 = execute(qc1, backend = simulator).result()
output_state_1 = result1.get_statevector(qc1)

result2 = execute(qc2, backend = simulator).result()
output_state_2 = result2.get_statevector(qc2)

fidelity = state_fidelity(output_state_1, output_state_2)
print('\n Fidelity: ', fidelity)
```

```
Fidelity:  1.0
```

Now imagine that instead of applying one X gate to each qubit in the second circuit, we only apply it to one of them, let's say the first one. In this case, only q_0 would flip and the resulting state would be $|q_1 q_0\rangle = |01\rangle$ which is orthogonal to the previous one $|q_1 q_0\rangle = |11\rangle$, so one expects the fidelity to be zero.

In [7]:
```
qc3 = QuantumCircuit(2)
qc3.x(0)

result3 = execute(qc3, backend = simulator).result()
output_state_3 = result3.get_statevector(qc3)

fidelity = state_fidelity(output_state_1, output_state_3)
print('Fidelity: ', fidelity)
```

```
Fidelity:  0.0
```

Finally, consider the following case.

In [11]:
```
qc1 = QuantumCircuit(2)
qc1.h([0,1])

qc2 = QuantumCircuit(2)
qc2.h(0)

result1 = execute(qc1, backend = simulator).result()
out_state_1 = result1.get_statevector(qc1)

result2 = execute(qc2, backend = simulator).result()
out_state_2 = result2.get_statevector(qc2)

fidelity = state_fidelity(out_state_1, out_state_2)
print('Fidelity: ', fidelity)
```

```
Fidelity:  0.5000000000000001
```

In this case, the first circuit puts both qubits into superposition $|q_1 q_0\rangle = |00\rangle \rightarrow |++\rangle$ while the second one only puts q_0 into it, $|q_1 q_0\rangle = |00\rangle \rightarrow |0+\rangle$. As a result, we have a 50% chance of obtaining the first state while measuring the second on the X basis, since

$$|0\rangle = \frac{|+\rangle + |-\rangle}{\sqrt{2}},$$

so the state fidelity is expected to be 0.5.

Process or gate fidelity and average gate fidelity

Consider we want to compare the operator representing our quantum circuit with its ideal operator. In this case, one would expect a process fidelity of 1.

In [22]:
```
from qiskit.quantum_info import process_fidelity, average_gate_fidelity,
                                Operator, Pauli

# Let's first get the operator from the circuit

qc = QuantumCircuit(1)
qc.h(0)
operator_1 = Operator(qc)


# Or we can get it using the unitary_simulator

backend = Aer.get_backend('unitary_simulator')
job = execute(qc, backend)
operator_2 = job.result().get_unitary(qc, decimals = 3)


# Now the ideal operator using the from_label() method

ideal_operator = Operator.from_label('H')

# And now let's compare both

fidelity_1 = process_fidelity(operator_1, ideal_operator)
fidelity_2 = process_fidelity(operator_2, ideal_operator)

av_fidelity_1 = average_gate_fidelity(operator_1, ideal_operator)
av_fidelity_2 = average_gate_fidelity(operator_2, ideal_operator)


print('Process fidelities are: ', fidelity_1, fidelity_2)
print('Average gate fidelities are: ', av_fidelity_1, av_fidelity_2)
```

```
Input channel is not TP. Tr_2[Choi] - I has non-zero eigenvalues: [-0.000
302 -0.000302]
Process fidelities are:  0.9999999999999996 0.9996979999999996
Average gate fidelities are:  0.9999999999999997 0.9997986666666664
```

Apart from comparing gates, process fidelity can be used to check circuit identities, such as the ones we saw in the single qubit gates section. For example, let's check that $HXH = Z$

In [8]:
```
qc1 = QuantumCircuit(1)
qc1.h(0)
qc1.x(0)
qc1.h(0)

qc2 = QuantumCircuit(1)
qc2.z(0)

print('Average fidelity: ', average_gate_fidelity(qc1, qc2))
print('Process fidelity: ', process_fidelity(qc1, qc2))
```

```
Average fidelity:  0.9999999999999997
Process fidelity:  0.9999999999999996
```

Operators

back to the top

The `Operator` class is used to represent operators acting on a quantum system.

Creating operators

They can be created directly giving a list with the rows representing them in matrix form

In [25]:
```
import numpy as np
from qiskit.quantum_info import Operator

X = Operator([[0,1],[1,0]])
print(X)
```

```
Operator([[0.+0.j, 1.+0.j],
          [1.+0.j, 0.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

One can also construct `Operator` from other objects like `Pauli`, `Gate` or `QuantumCircuit`

In [26]:
```
from qiskit.quantum_info import Pauli

pauliX = Pauli('X')
X = Operator(pauliX)
print(X)
```

```
Operator([[0.+0.j, 1.+0.j],
          [1.+0.j, 0.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

In [27]:
```
from qiskit.extensions import XGate

gate = XGate()
X = Operator(gate)
print(X)
```

```
Operator([[0.+0.j, 1.+0.j],
          [1.+0.j, 0.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

In [31]:
```
from qiskit import QuantumCircuit

qc = QuantumCircuit(1)
qc.x(0)
X = Operator(qc)
print(X)
```

```
Operator([[0.+0.j, 1.+0.j],
          [1.+0.j, 0.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

Random unitary or hermitian `Operator` can be created using the `qiskit.quantum_info.random_unitary(dims)` or `qiskit.quantum_info.random_hermitian(dims)`, respectively.

We can also convert those `Operator` objects into Numpy arrays using the `Operator.data` property

In [32]:
```
X.data
```

Out[32]:
```
array([[0.+0.j, 1.+0.j],
       [1.+0.j, 0.+0.j]])
```

Operators can be also combined, either via tensor product, inner product or linear combinations.

In [40]:
```
A = Operator(Pauli('I'))
B = Operator(Pauli('X'))

A.tensor(B)
```

```
Operator([[0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
          [1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
          [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j]],
         input_dims=(2, 2), output_dims=(2, 2))
```

Note that the gate $A \otimes B$ in Qiskit notation is in fact $A_1 \otimes B_0$, i.e., B acts on q_0 while A does on q_1. This is due to Qiskit's ordering of qubits. The reverse order operation $B \otimes A = B_1 \otimes A_0$ can be achieved using the `expand()` method.

In [41]:
```
A.expand(B)
```

```
Operator([[0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
          [0.+0.j, 0.+0.j, 0.+0.j, 1.+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]],
         input_dims=(2, 2), output_dims=(2, 2))
```

For regular product operators like XZX we can use the `compose()` method. If we want to put B in front of A we can set `front = True` inside it.

In [43]:
```
A = Operator(Pauli('X'))
B = Operator(Pauli('Z'))

A.compose(B).compose(A)
```

```
Operator([[-1.+0.j,  0.+0.j],
          [ 0.+0.j,  1.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

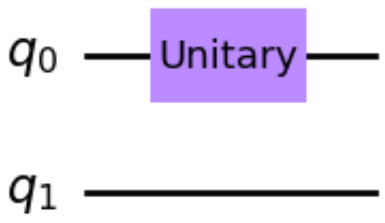
In [44]:
```
A + B - 3*A
```

```
Operator([[ 1.+0.j, -2.+0.j],
          [-2.+0.j, -1.+0.j]],
         input_dims=(2,), output_dims=(2,))
```

`Operator` objects can be appended into `QuantumCircuit` through the `append()` method. It converts the `Operator` into a `UnitaryGate` object before adding it to the circuit. Since trying to append a non-unitary gate to a `QuantumCircuit` will raise an exception, the method `Operator.is_unitary()` may be used to check beforehand whether it is in fact unitary or not.

In [52]:
```
qc = QuantumCircuit(2)
qc.append(X, [0]) # Need to wrap the qubit index in square brackets.
qc.draw('mpl')

# To see the contents of this unitary, we can use the
# QuantumCircuit.decompose() method and then draw it.
```

Out[52]:



Finally, we saw in the previous section that two gates can be compared to see if they are approximately equal using fidelity. Operators also implement a equality method that can be used to achieve this same goal.

In [45]:
```
A = Operator(Pauli('X'))
B = Operator(XGate())

A == B
```

Out[45]:
```
True
```

IMPORTANT: This method checks that each matrix element representing the operators is approximately equal, so two unitaries that differ a global phase will NOT be considered equal. This is not the case with process fidelity, where those two unitaries would actually yield fidelity 1.

In [50]:
```
A = Operator(Pauli('X'))
B = np.exp(1j*np.pi) * A

print(A == B)
print(process_fidelity(A, B))
```

```
False
1.0
```

Density operators

back to the top

It is an alternative way of expressing quantum states. For pure states,

$$\rho \equiv |\psi\rangle \langle\psi|$$

.

For mixed states consisting of an ensamble of n states each with probability of ocurrence p_j,

$$\rho \equiv \sum_{j=1}^{n} p_j \, |\psi_j\rangle \langle\psi_j|$$

.

These operators, as with statevectors, can be represented in the computational basis (or any basis) as matrices.

In Qiskit, we can get the `DensityMatrix` representing the density operator of the resulting state of a circuit directly from it. For example, from the Bell pair circuit:

In [29]:
```
from qiskit.quantum_info import DensityMatrix

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)

rho = DensityMatrix.from_instruction(qc)
rho.draw('latex', prefix = '\\rho =')
```

Out[29]:

$$\rho = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

One may also construct density operators from their original statevectors

In [32]:
```
state = Statevector.from_instruction(qc)

state.to_operator().draw('latex', prefix = '\\rho =')
```

Out[32]:

$$\rho = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

And reversely, transform the density operator into a statevector

In [34]:
```
state = rho.to_statevector()

state.draw('latex', prefix = '| \\psi \\rangle =')
```

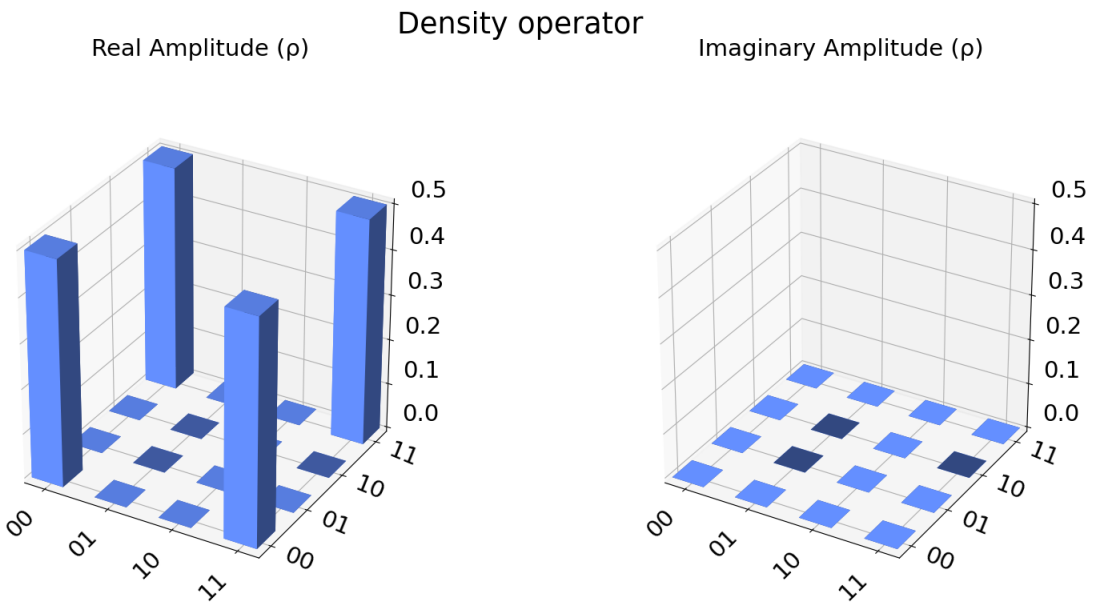
Out[34]:

$$|\psi\rangle = \frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$$

We can also visualize it via a cityscape plot

In [30]:
```
from qiskit.visualization import plot_state_city

plot_state_city(rho, title = 'Density operator')
```

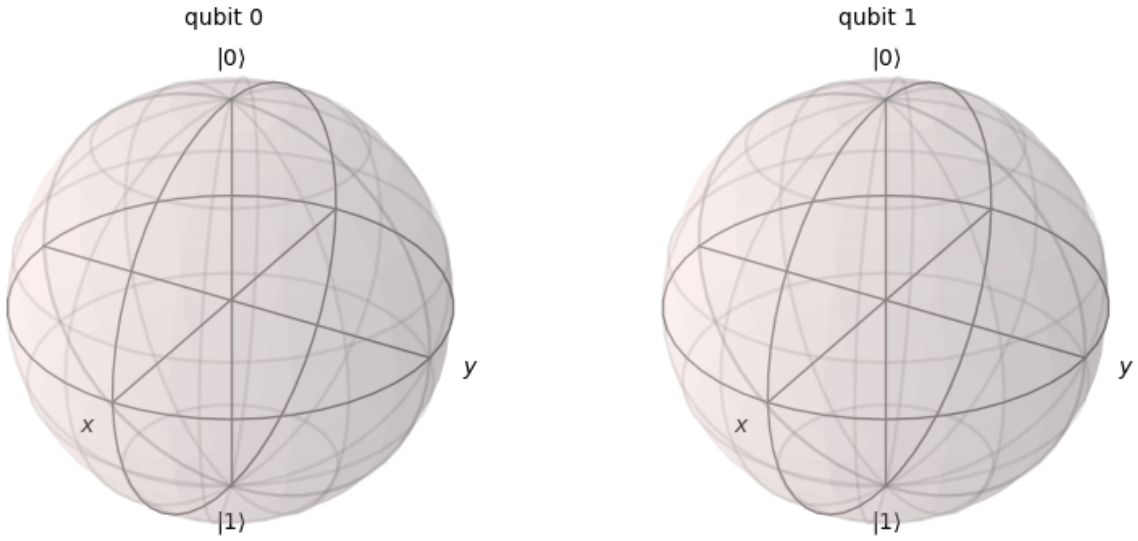
Out[30]:



Or even in the Bloch sphere

In [43]:
```
from qiskit.visualization import plot_bloch_multivector

plot_bloch_multivector(rho.data)
```

Out[43]:



Another way to create a density matrix is manually; inputing the states as arrays and doing the tensor product between them. For example, consider the state $|00\rangle$

In [40]:
```
state_1 = Statevector([1,0])
state_2 = Statevector([1,0])

state_1.tensor(state_2).to_operator().draw('latex', prefix = '\\rho =')
```

Out[40]:

$$\rho = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Or even simpler, by just inputing the matrix elements as a numpy array and then convert it into DensityMatrix

In [42]:
```
import numpy as np

rho_matrix = np.array([[1/2, 0, 0, 1/2],
                       [0, 0, 0, 0],
                       [0, 0, 0, 0],
                       [1/2, 0, 0, 1/2]])

rho = DensityMatrix(rho_matrix)
rho.draw('latex', prefix = '\\rho =')
```

Out[42]:

$$\rho = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

Finally, for more complicated states like $\rho = \frac{1}{3}|1\rangle\langle 1| + \frac{2}{3}|+\rangle\langle +|$ one may use the `from.label()` method.

In [44]:
```
rho = 1/3*DensityMatrix.from_label('1') +
      2/3 * DensityMatrix.from_label('+')
rho.draw('latex', prefix = '\\rho =')
```

Out[44]:

$$\rho = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{2}{3} \end{bmatrix}$$

Getting real quantum computer properties

back to the top

Apart from simulators, we can also run our code on real quantum computers provided by IBM. First we need to log into our IBMQ account using or API token

In [1]:
```
from qiskit import IBMQ

IBMQ.enable_account('<insert token>')
```

```
/var/folders/8_/j_18s_f53db_5lmjt_zj2hc80000gn/T/ipykernel_1675/1132892266.py:3: DeprecationWarning: The package qiskit.providers.ibmq is being deprecated. Please see https://ibm.biz/provider_migration_guide to get instructions on how to migrate to qiskit-ibm-provider (https://github.com/Qiskit/qiskit-ibm-provider) and qiskit-ibm-runtime (https://github.com/Qiskit/qiskit-ibm-runtime).
  IBMQ.enable_account('a16d511f4ae77312f0f17395b97c07fa48cb905a2739b8820e188a62d9d8778e13e70fc7fa3d2858f5cb4730f1872e0e0ed90eb7bc82474c51c2626391919e4d')
/var/folders/8_/j_18s_f53db_5lmjt_zj2hc80000gn/T/ipykernel_1675/1132892266.py:3: DeprecationWarning: The qiskit.IBMQ entrypoint and the qiskit-ibmq-provider package (accessible from 'qiskit.providers.ibmq`) are deprecated and will be removed in a future release. Instead you should use the qiskit-ibm-provider package which is accessible from 'qiskit_ibm_provider'. You can install it with 'pip install qiskit_ibm_provider'. Just replace 'qiskit.IBMQ' with 'qiskit_ibm_provider.IBMProvider'
  IBMQ.enable_account('a16d511f4ae77312f0f17395b97c07fa48cb905a2739b8820e188a62d9d8778e13e70fc7fa3d2858f5cb4730f1872e0e0ed90eb7bc82474c51c2626391919e4d')
```

Out[1]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

Now we can retrieve a list with all available quantum computers

In [14]:
```
providers = IBMQ.get_provider(hub ='ibm-q')
providers.backends()
```

Out[14]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_brisbane') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_kyoto') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_osaka') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

Now, how do we choose among them? Maybe we want to decide based on number of qubits, pending jobs or on whether they are actual quantum devices or simply simulators. To display that information, we can run the following loop

In [15]:
```
for backend in providers.backends():

    try:
        qubit_count = len(backend.properties().qubits)

    except:
        qubit_count = 'simulated'

    print(f'{backend.name()}: has {backend.status().pending_jobs} pending
```

```
ibmq_qasm_simulator: has 0 pending jobs and simulated qubits
simulator_statevector: has 0 pending jobs and simulated qubits
simulator_mps: has 0 pending jobs and simulated qubits
simulator_extended_stabilizer: has 0 pending jobs and simulated qubits
simulator_stabilizer: has 0 pending jobs and simulated qubits
ibm_brisbane: has 177 pending jobs and 127 qubits
ibm_kyoto: has 704 pending jobs and 127 qubits
ibm_osaka: has 46 pending jobs and 127 qubits
```

Let's run a simple circuit on one of these devices

In [ ]:
```
from qiskit import QuantumCircuit, execute

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
qc.measure_all()

# Note that this circuit creates a Bell pair in the state phi+

backend = providers.get_backend('ibm_osaka')

job = execute(qc, backend)
result = job.result()
counts = result.get_counts(qc)

print('\n The resulting state is:', counts)
plot_histogram(counts)
```

If we want to monitor the status of a job, we can pop these lines right after defining the job in the previous code

In [10]:
```
from qiskit.tools import job_monitor

# ...
# job = execute(qc, backend)

job_monitor(job)
job.status()
```

```
Job Status: job has successfully run
```

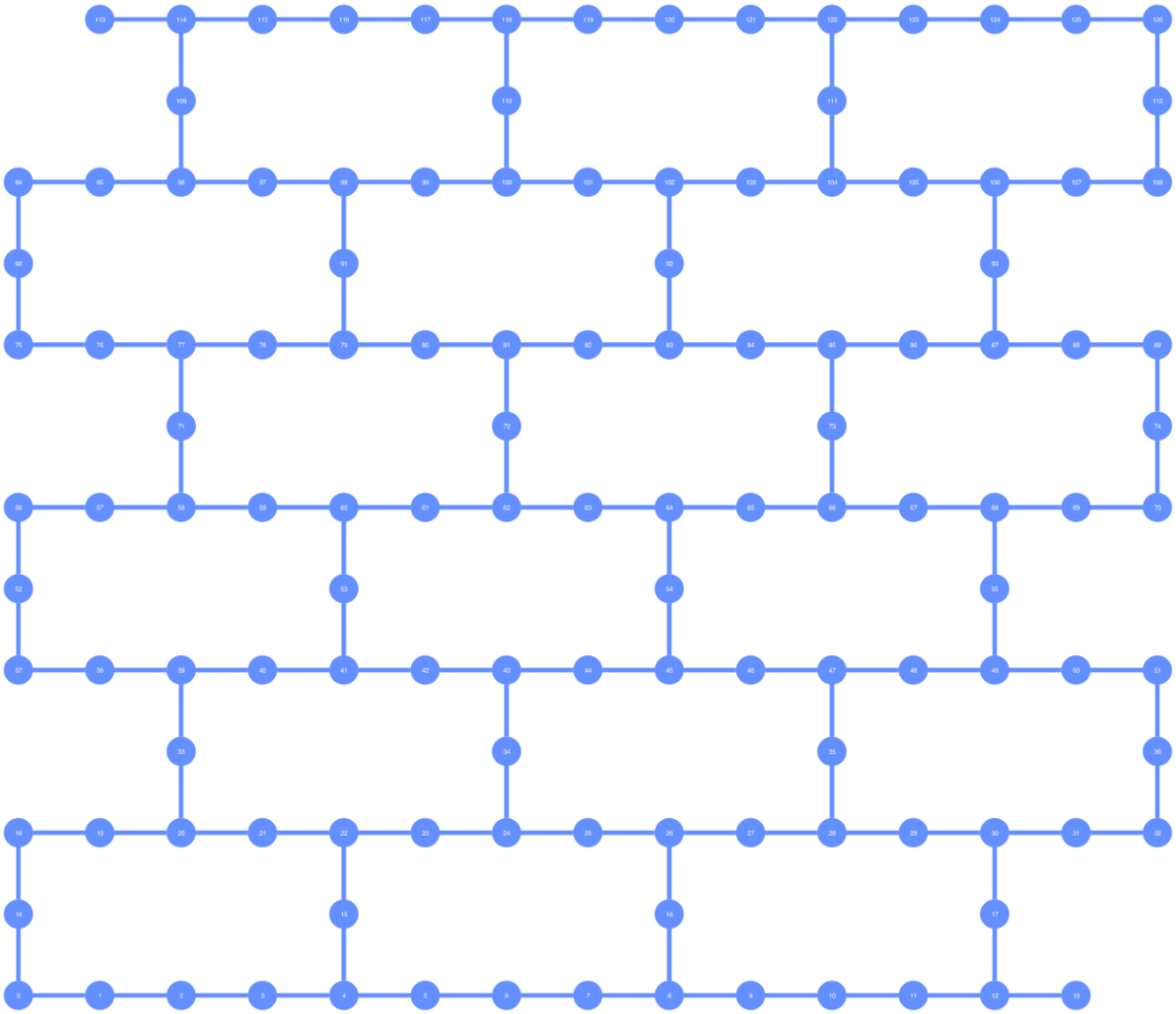
Out[10]:
```
<JobStatus.DONE: 'job has successfully run'>
```

Plotting gate maps and error rates

We can plot the way physical qubits are interconnected in our real quantum device (gate map), as well as the error rate expected on qubit readouts and gates (error map). To do this, first we need to connect to an actual backend using our IBMQ account.

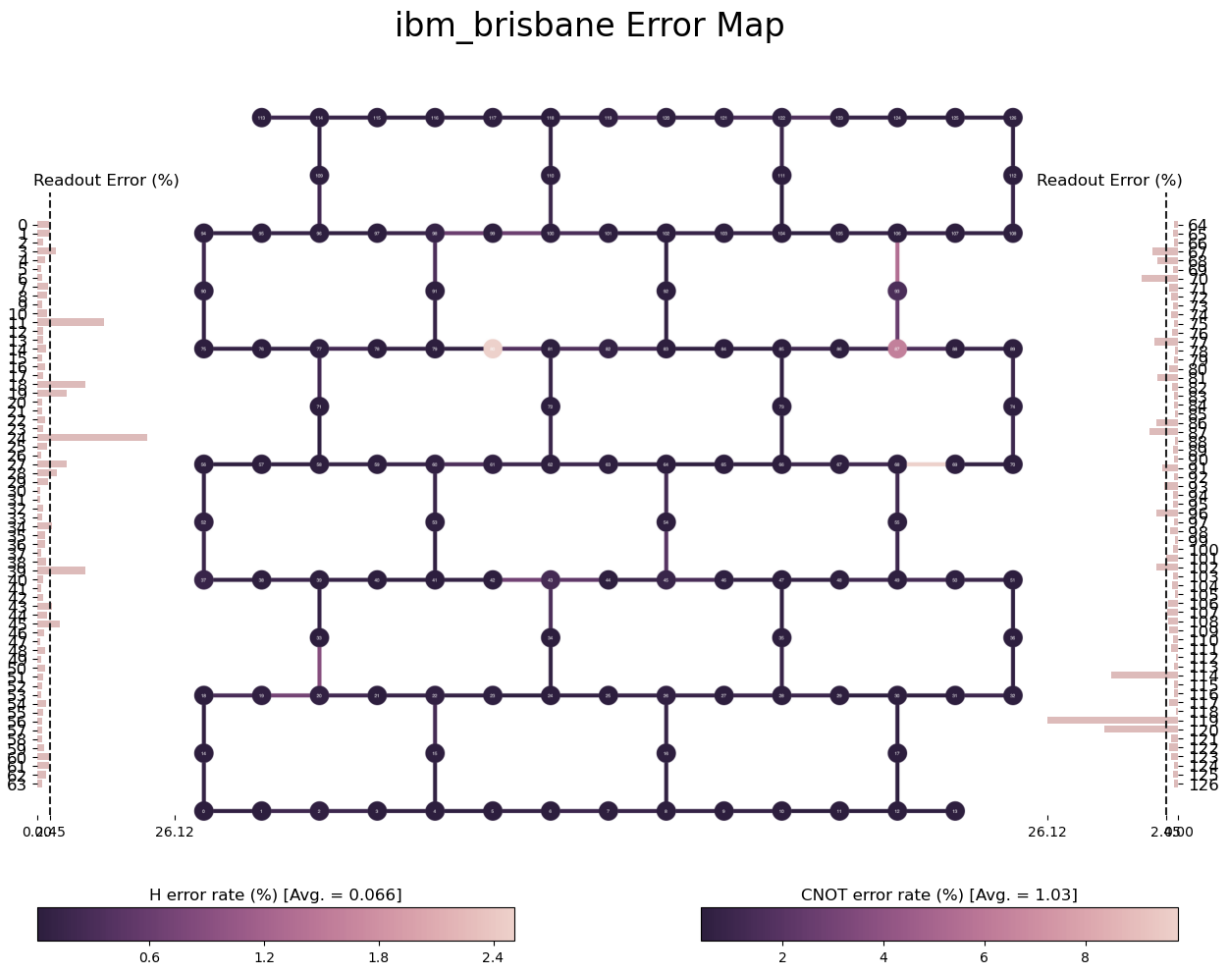
In [2]:
```
from qiskit.visualization import plot_gate_map, plot_error_map

provider = IBMQ.get_provider(hub = 'ibm-q')
backend = provider.backends(simulator = False)[0] # Restricts the
# backends to real devices only and selects the first one on the list.

plot_gate_map(backend)
```

Out[2]:



In [3]:
```
plot_error_map(backend)
```

Out[3]:



Coupling maps

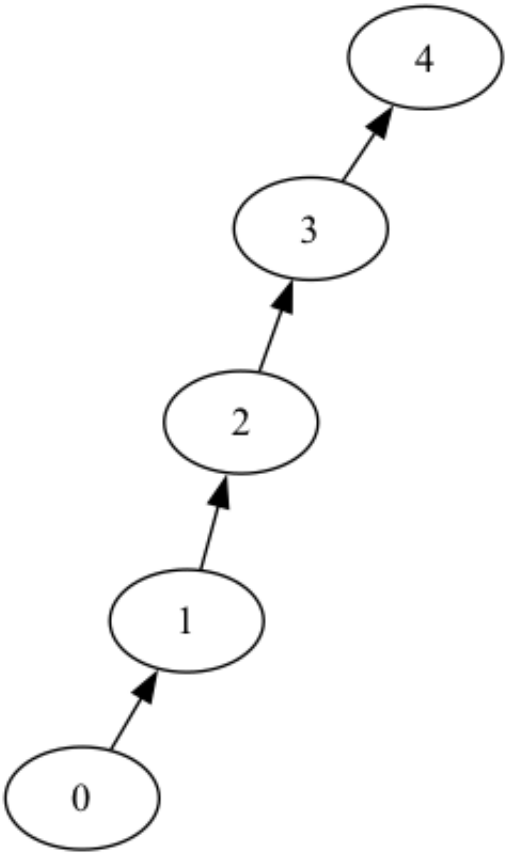
They are directed graphs that represent the constraints in connectivity of qubits in a specific backend. The nodes represent the physical qubits and the directed edges correspond to permitted CNOT gates, with source and destination corresponding to control and target qubits.

These are used by the compilere to know the limitations in connectivity when mapping a circuit to a backend. As such, they can be added in the `execute` function setting `coupling_map = cmap`, where `cmap` is our custom coupling map.

In [3]:

```
from qiskit.transpiler import CouplingMap

cmap = CouplingMap()

for i in range(4):
    cmap.add_edge(i, i+1)

cmap.draw()
```

Out[3]:



Sometimes we can get asked about how can one get quick information about real quantum devices available in Qiskit. To do this,

In [ ]:

```
import qiskit.tools.jupyter
%qiskit_backend_overview
```