# IT437 lab3 by Vivek Vittal Biragoni(211AI041)

In [ ]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit_aer import AerSimulator

# qiskit-ibmq-provider has been deprecated.
# Please see the Migration Guides in https://ibm.biz/provider_migration_guide for m
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, O

# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")

# Invoke a primitive inside a session. For more details see https://qiskit.org/docu
# with Session(backend=service.backend("ibmq_qasm_simulator")):
#     result = Sampler().run(circuits).result()
```

In [ ]:
```
!pip install qiskit
```

In [2]:
```
# Useful additional packages
import matplotlib.pyplot as plt
import numpy as np
from math import pi
```

In [3]:
```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, transpile
from qiskit.tools.visualization import circuit_drawer
from qiskit.quantum_info import state_fidelity
from qiskit import BasicAer
from qiskit.visualization import plot_histogram, plot_bloch_multivector, visualize_

backend_unitary = BasicAer.get_backend('unitary_simulator')
backend_qasm = BasicAer.get_backend('qasm_simulator')
backend_statevector = BasicAer.get_backend('statevector_simulator')
```

A qubit is like a tiny computer that can do special things that normal computers can't. It can be in two states, called "0" and "1", but it can also be in both states at the same time. This might seem strange, but it's actually really important for solving certain problems.

A qubit state is like a special recipe that tells us how much of "0" and "1" it has. We write it like this: $\left|\psi\right\rangle = \alpha\left|0\right\rangle + \beta \left|1\right\rangle$. $\alpha$ and $\beta$ are like the ingredients in the recipe, but instead of being regular numbers, they're something called "complex numbers". We don't need to worry too much about that for now.

When we look at the qubit to see what state it's in, it will either be in state "0" with a probability of $|\alpha|^2$, or in state "1" with a probability of $|\beta|^2$. The sum of these

probabilities is always 1, because the qubit has to be in one of the two states.

We can also draw the qubit as a little ball called the Bloch sphere. This helps us understand how it works. Different points on the surface of the ball correspond to different qubit states.

Finally, we can do special things to qubits, like turn them from one state into another. We call these things "quantum gates". To do this, we use a special kind of math called "matrix multiplication". Each gate is like a recipe for changing a qubit state in a particular way.

# Single-Qubit Gates

The single-qubit gates available are:

- U gate
- P gate
- Identity gate
- Pauli gates
- Clifford gates
- $C3$ gates
- Standard rotation gates

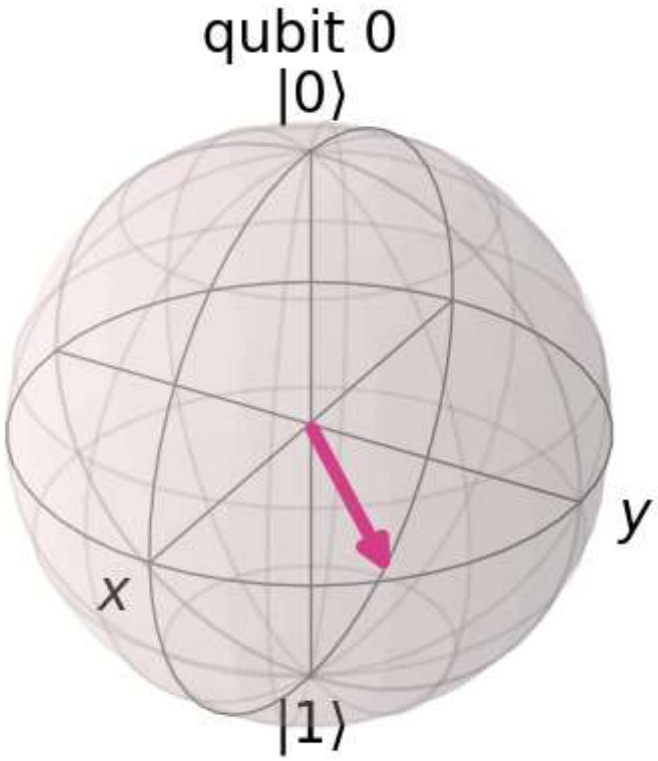
In [4]:
```
q = QuantumRegister(1)
```

# The U gate

In [16]:
```
#Implementation of the U gate.
qc = QuantumCircuit(q)
qc.u(pi/2,pi/4,pi/8,q)
print(qc.draw())
plot_bloch_multivector(qc)
```

```
Figure(287.093x117.056)
```

Out[16]:




This code creates a quantum circuit object named qc with a quantum register q. The circuit applies a single-qubit gate called the U gate with specific parameters (pi/2, pi/4, pi/8) to the qubit in the quantum register. The qc.draw() function visualizes the circuit.

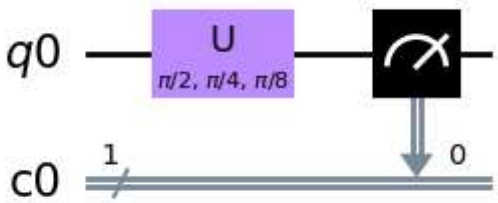
In [6]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[6]:
```
array([[ 0.707+0.j   , -0.653-0.271j],
       [ 0.5  +0.5j  ,  0.271+0.653j]])
```

This code runs a quantum circuit object qc on a backend that supports unitary simulation, which produces a job object. The job.result().get_unitary(qc, decimals=3) function then retrieves the unitary matrix representing the quantum circuit, with a specified level of precision.

In [8]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.u(pi/2,pi/4,pi/8,0)
qc.measure(q,c)
qc.draw()
```

Out[8]:

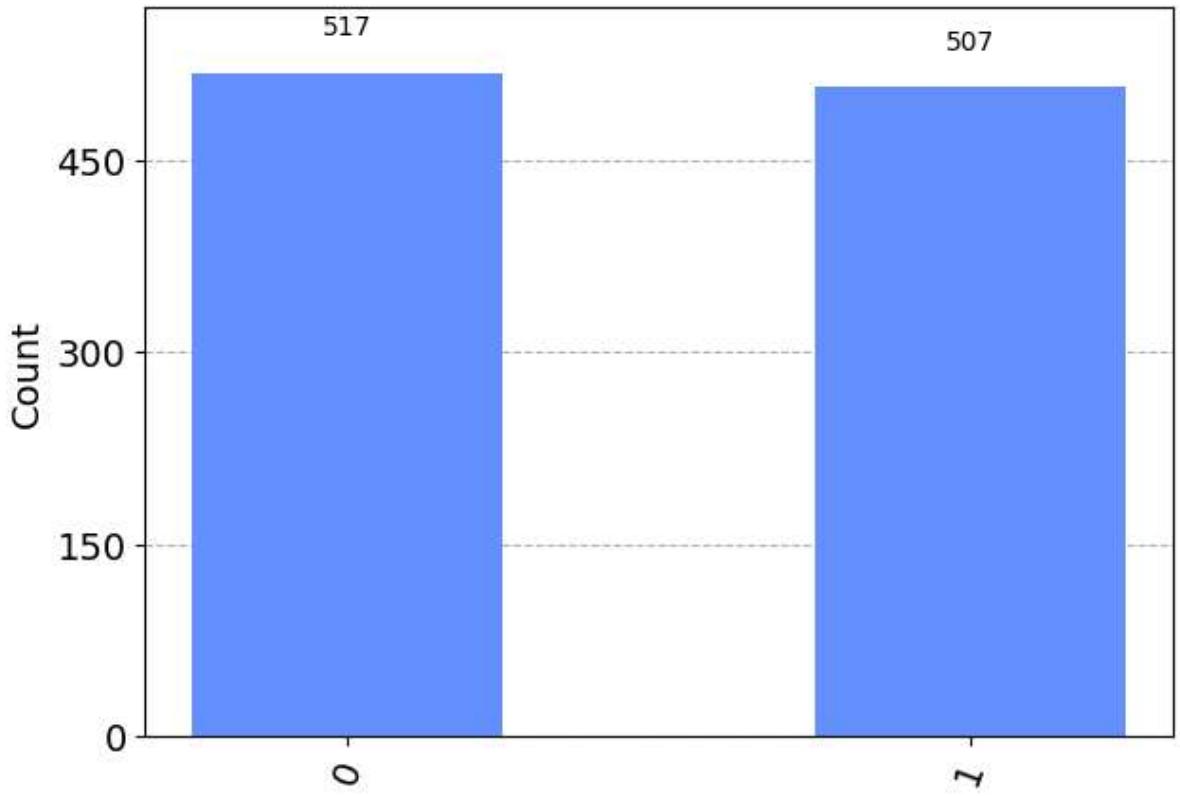



This code creates a quantum circuit object named qc with a quantum register q and a classical register c. The circuit applies a single-qubit gate called the U gate with specific parameters (pi/2, pi/4, pi/8) to the qubit in the quantum register, and then measures the state of that qubit and stores the measurement result in the classical register. The qc.draw() function visualizes the circuit.

In [9]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'1': 507, '0': 517}
```

Out[9]:

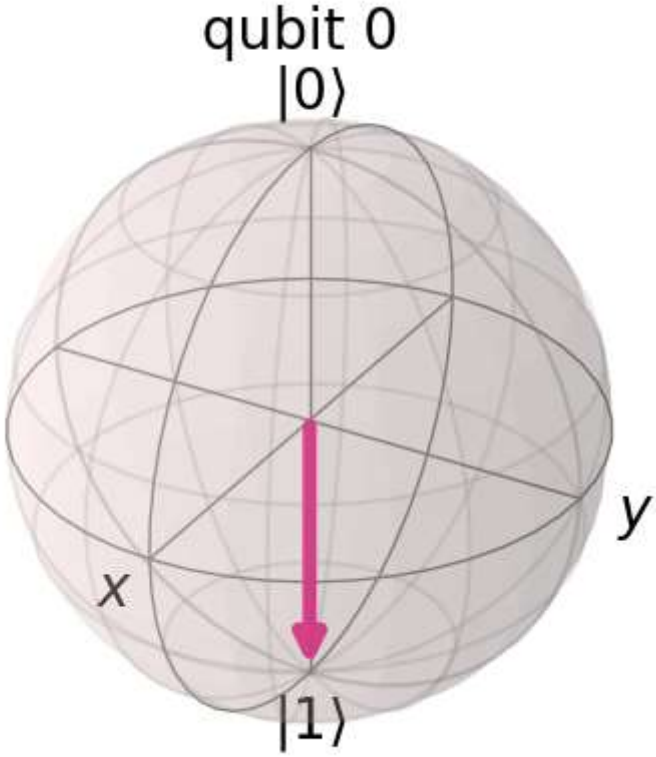



This code is running a quantum circuit qc on a backend called backend_qasm and obtaining the counts of all possible measurement outcomes. Then, it is displaying a histogram plot of those counts.

In [10]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

```
[0.        +0.j         0.70710678+0.70710678j]
```

Out[10]:




This code is running a quantum circuit qc on a backend called backend_statevector and obtaining the state vector of the resulting quantum state. Then, it is displaying a Bloch sphere plot of that state vector.

understanding of u gate the u gate is a type of gate that we can use to manipulate qubits in a certain way. It's called the u gate because "u" stands for "universal," which means that we can use it to do lots of different things to a qubit.

For example, we can use the u gate to change the phase of a qubit. This means that if a qubit is in one state, the u gate can change it to a different state. We can also use the u gate to rotate the state of a qubit in a certain direction.

# The P gate

## P gate

The $p(\lambda)= u(0, 0, \lambda)$ gate has the matrix form

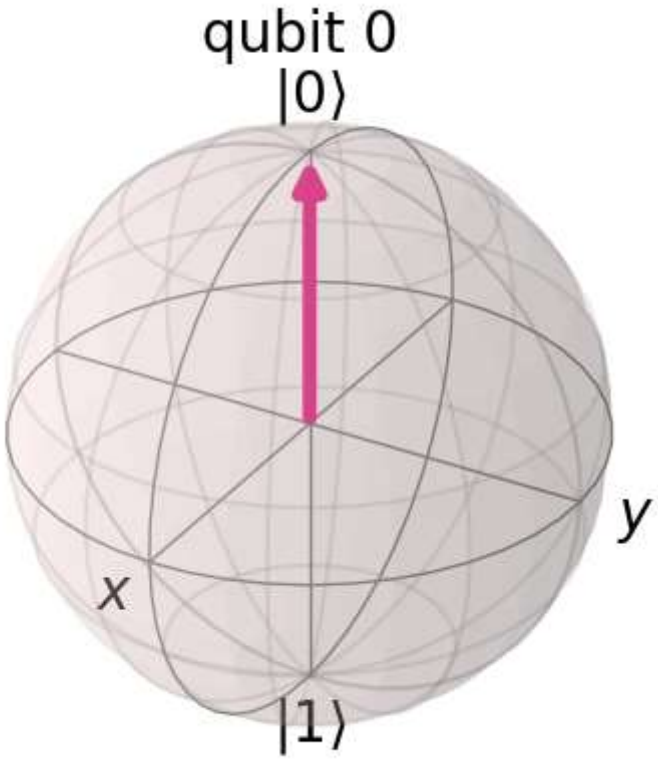
$$ p(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i \lambda} \end{pmatrix}, $$

which is useful as it allows us to apply a quantum phase.

In [17]:
```
#Implementation of P gate
qc = QuantumCircuit(q)
qc.p(pi/4,q)
print(qc.draw())
plot_bloch_multivector(qc)
```

```
Figure(203.481x117.056)
```

Out[17]:




In [13]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[13]:
```
array([[1.+0.j, 0.+0.j],
       [0.+0.j, 0.+1.j]])
```

The circuit starts by creating a QuantumCircuit object called "qc" with a single qubit "q" as input. The code then applies a gate called "p" to the qubit "q", with an angle of pi/2. The "p" gate is a single-qubit gate that performs a phase shift of the qubit's state by the given angle.

After defining the circuit, the code uses a simulator backend called "backend_unitary" to simulate the quantum circuit and compute its resulting unitary matrix. The "unitary" is a mathematical object that describes how the quantum circuit transforms the state of the qubit.

Finally, the code prints out the resulting unitary matrix with a precision of 3 decimal places. This matrix represents the complete transformation that the quantum circuit performs on the qubit, which can be used to understand how the circuit affects the qubit's state.

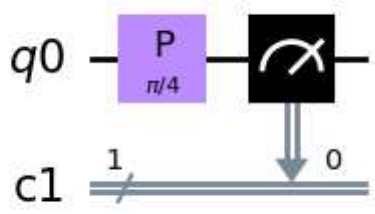
the P gate is a type of gate that we can use to manipulate qubits in a certain way. It's called the P gate because "P" stands for "phase shift," which means that it can change the phase of a qubit's state.

The P gate takes an angle as input, and when we apply the gate to a qubit, it rotates the qubit's state by that angle around the Z-axis. The Z-axis is one of the three axes in 3D space, and the P gate's rotation around this axis changes the probability of measuring a certain state for the qubit.

For example, if a qubit is initially in the state |0⟩ (which means it's more likely to be measured as a 0), applying a P gate with an angle of pi/2 would rotate the qubit's state to the state (|0⟩ + i|1⟩)/sqrt(2), which means it's equally likely to be measured as a 0 or a 1.

In [18]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.p(pi/4,0)
qc.measure(q,c)
qc.draw()
```

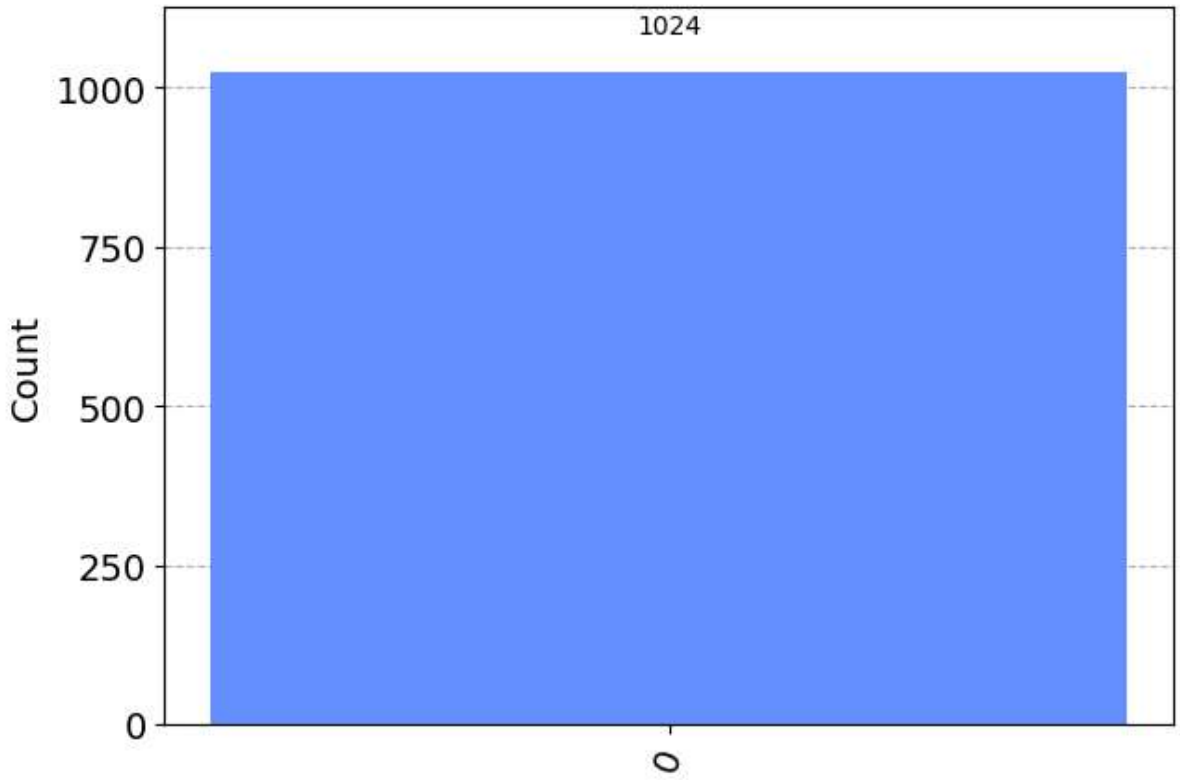
Out[18]:




In [19]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

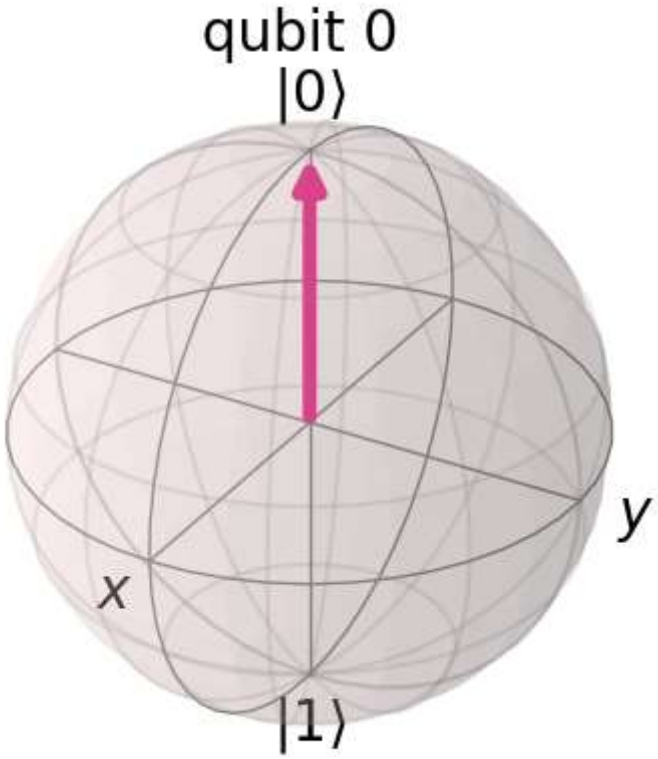
Out[19]:




In [20]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[20]:

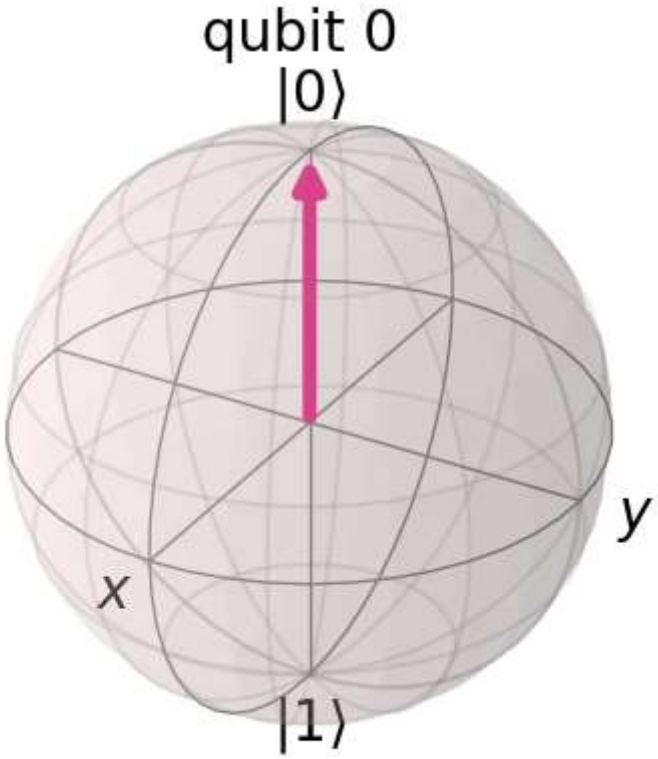


# the I gate

In [22]:
```
#Implementation of the identity gate.
qc = QuantumCircuit(q)
qc.id(q)
print(qc.draw())
plot_bloch_multivector(qc)
```

```
Figure(203.481x117.056)
```

Out[22]:

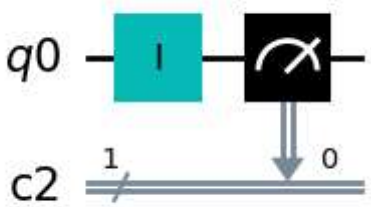



In [23]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[23]:
```
array([[1.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j]])
```

In [24]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.id(0)
qc.measure(q,c)
qc.draw()
```

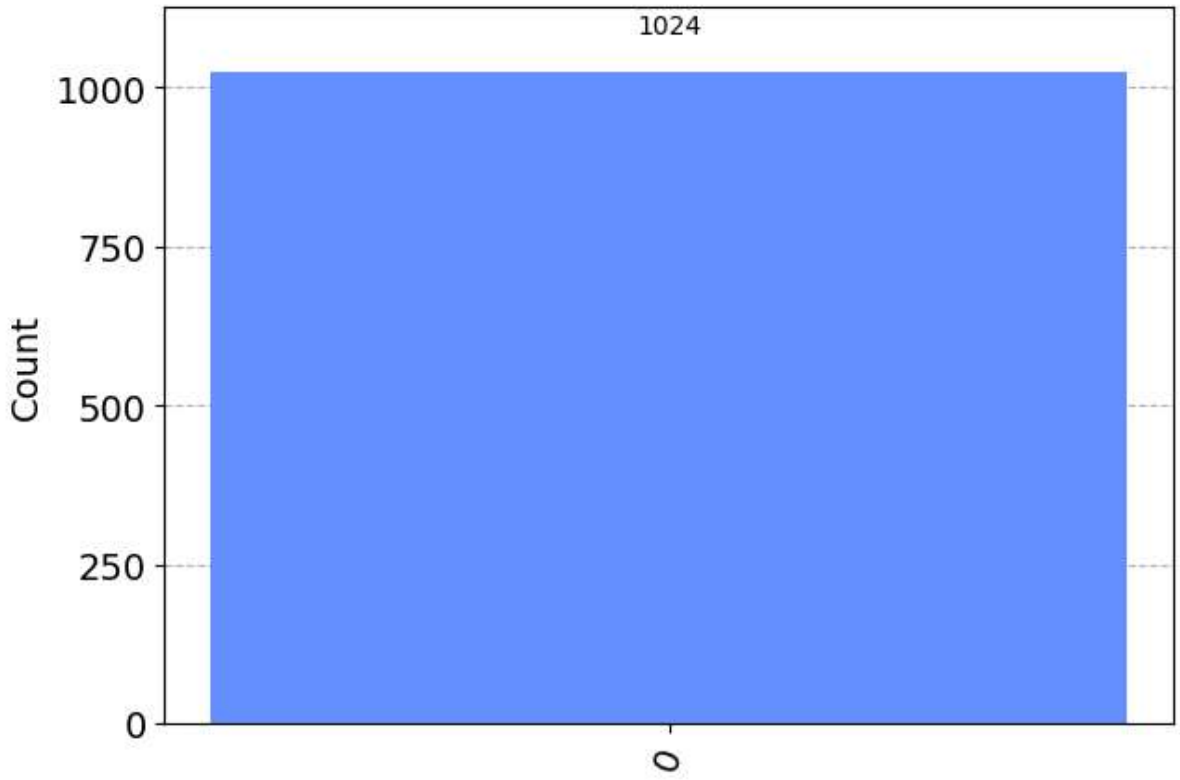
Out[24]:




In [25]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

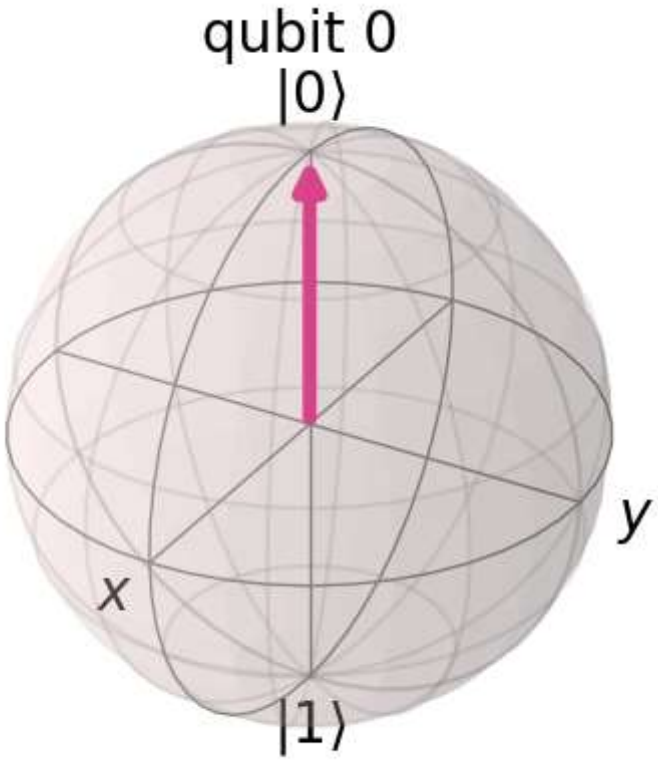
Out[25]:




In [26]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[26]:




the identity gate is a basic quantum gate that does not change the quantum state of a qubit. Mathematically, it is represented by the identity matrix, which is a square matrix with ones on the diagonal and zeros elsewhere.

The identity gate is sometimes called the "do nothing" gate because it leaves the qubit in the same state it was in before the gate was applied. However, the identity gate can still be useful in quantum circuits as a placeholder or to help align different parts of a circuit.

In essence, the identity gate is a no-op or a null operation, and it plays a role similar to the NOP (no-operation) instruction in classical computing.

## the X:bit-flip gate

In [27]:
```
#Implementation of the X:bit-flip gate
qc = QuantumCircuit(q)
qc.x(q)
qc.draw()
```

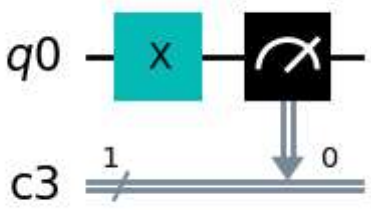
Out[27]:

In [28]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[28]:
```
array([[0.+0.j, 1.+0.j],
       [1.+0.j, 0.+0.j]])
```

In [29]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.x(0)
qc.measure(q,c)
qc.draw()
```

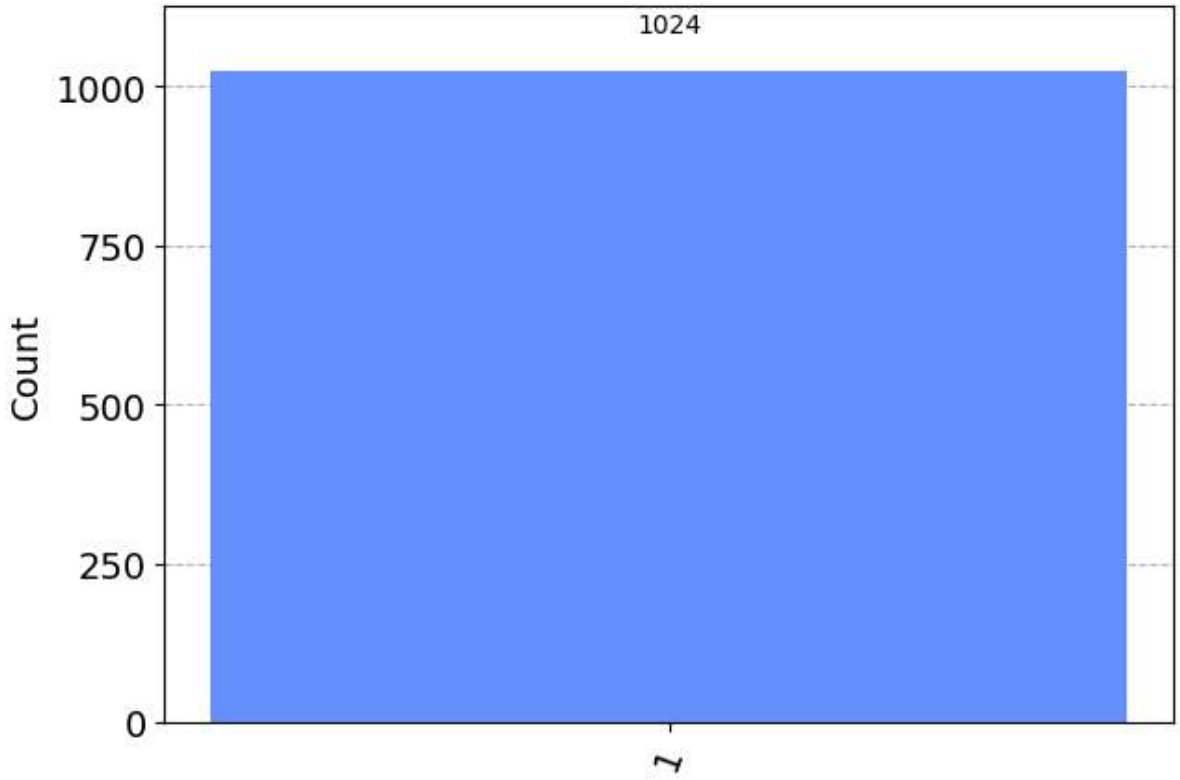
Out[29]:




In [30]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'1': 1024}
```

Out[30]:

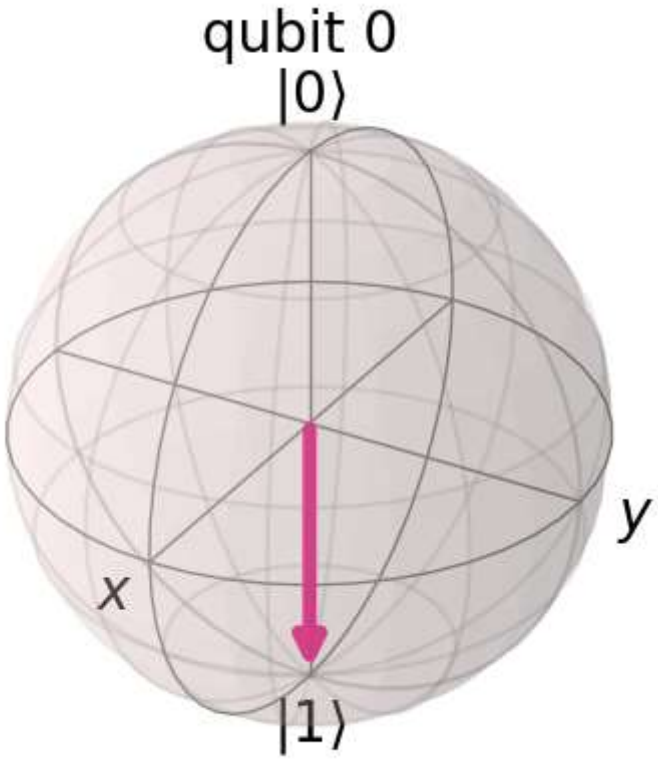



In [31]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
```

```
print(sv)
plot_bloch_multivector(sv)
```

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

Out[31]:




the X gate is a fundamental quantum logic gate that performs a bit-flip operation on a single qubit. The X gate switches the quantum state of a qubit from |0⟩ to |1⟩, or vice versa. Mathematically, the X gate corresponds to the Pauli-X matrix, which is a 2x2 matrix that flips the probability amplitudes of the qubit's state vector.

The X gate is important in quantum computing because it is a fundamental gate that can be used to construct more complex quantum circuits. It is also used in various quantum algorithms, such as the Deutsch-Jozsa algorithm and the Grover's search algorithm.

The X gate can also be applied to multiple qubits at once, in which case it performs a bit-flip operation on each qubit independently. This operation is known as the controlled-X (or CNOT) gate and is a key building block in quantum circuits for quantum error correction and quantum teleportation.

# Y:bit and phase-flip gate.

In [32]:
```
#Implementation of the Y:bit and phase-flip gate.
qc = QuantumCircuit(q)
qc.y(q)
qc.draw()
```

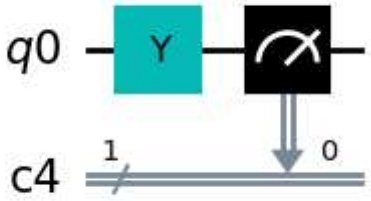
Out[32]:


In [33]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[33]:
```
array([[ 0.+0.j, -0.-1.j],
       [ 0.+1.j,  0.+0.j]])
```

In [34]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.y(0)
qc.measure(q,c)
qc.draw()
```

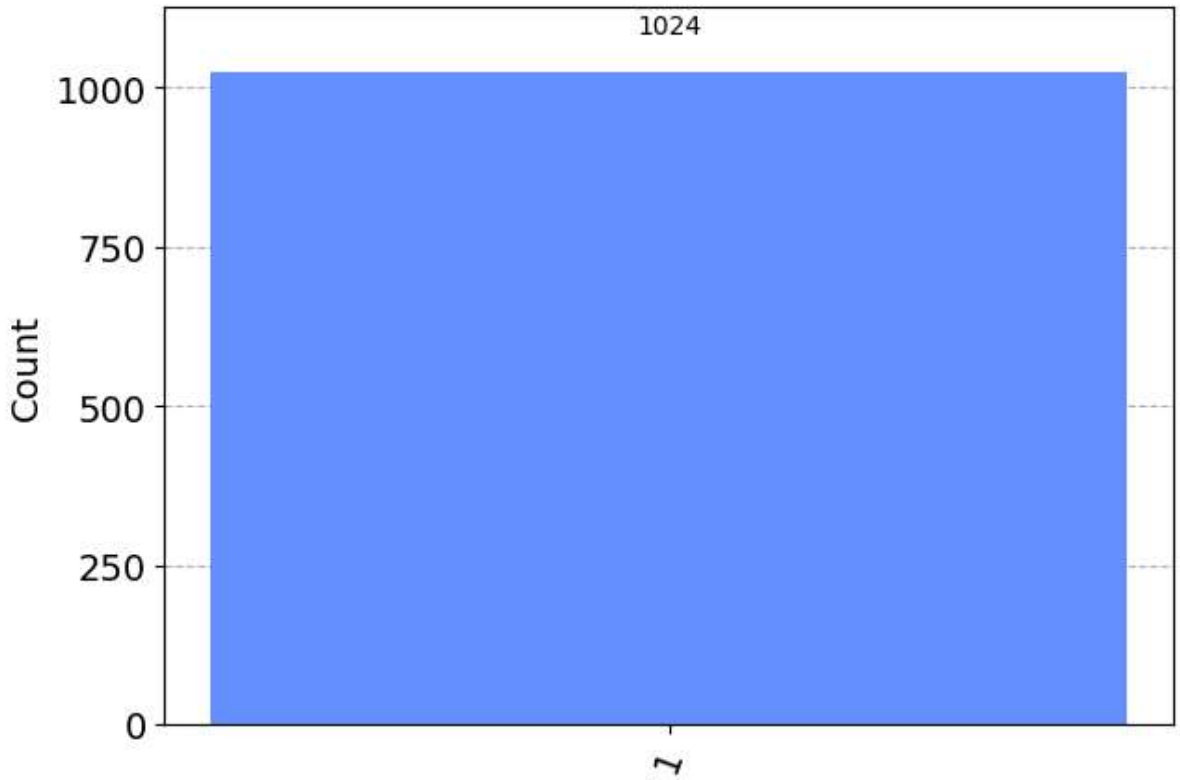
Out[34]:




In [35]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'1': 1024}
```

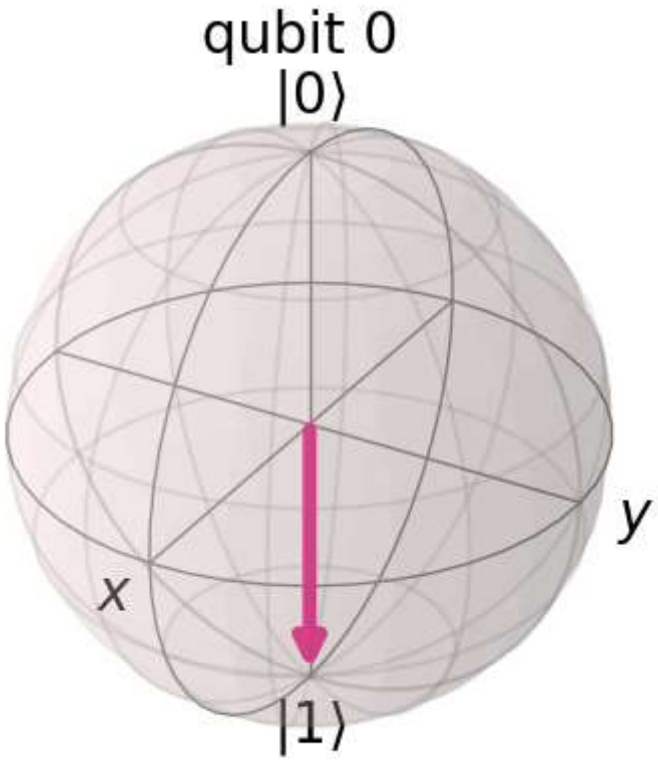
Out[35]:




In [36]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

```
[0.000000e+00+0.j 6.123234e-17+1.j]
```

Out[36]:



the Y gate is a fundamental quantum logic gate that performs a phase-flip operation on a single qubit. The Y gate rotates the quantum state of a qubit by 180 degrees around the y-axis of the Bloch sphere. Mathematically, the Y gate corresponds to the Pauli-Y matrix, which is a 2x2 matrix that flips the sign of the qubit's imaginary probability amplitudes.

The Y gate is important in quantum computing because it, together with the X gate, can generate any single-qubit gate via rotations around the x, y, and z axes of the Bloch sphere. It is also used in various quantum algorithms, such as the quantum Fourier transform.

The phase-flip gate, on the other hand, is a generalization of the Y gate that performs a phase shift operation on a qubit. The phase-flip gate rotates the quantum state of a qubit by a certain angle around the z-axis of the Bloch sphere, without changing its probability amplitudes. Mathematically, the phase-flip gate corresponds to a matrix that adds a phase factor to the qubit's state vector.

The phase-flip gate is useful in quantum circuits for creating superposition states and implementing quantum algorithms, such as the phase estimation algorithm and the quantum phase estimation algorithm. It is also a component of more complex quantum gates, such as the Toffoli gate and the controlled phase gate.

# the Z phase-flip gate.

In [38]:
```
#Implementation of the Z phase-flip gate.
qc = QuantumCircuit(q)
qc.z(q)
qc.draw()
```

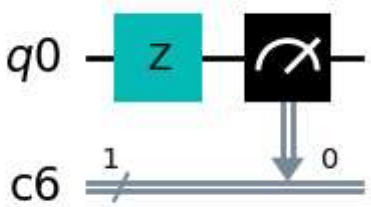
Out[38]:


In [39]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[39]:
```
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j, -1.+0.j]])
```

In [43]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.z(0)
qc.measure(q,c)
qc.draw()
```

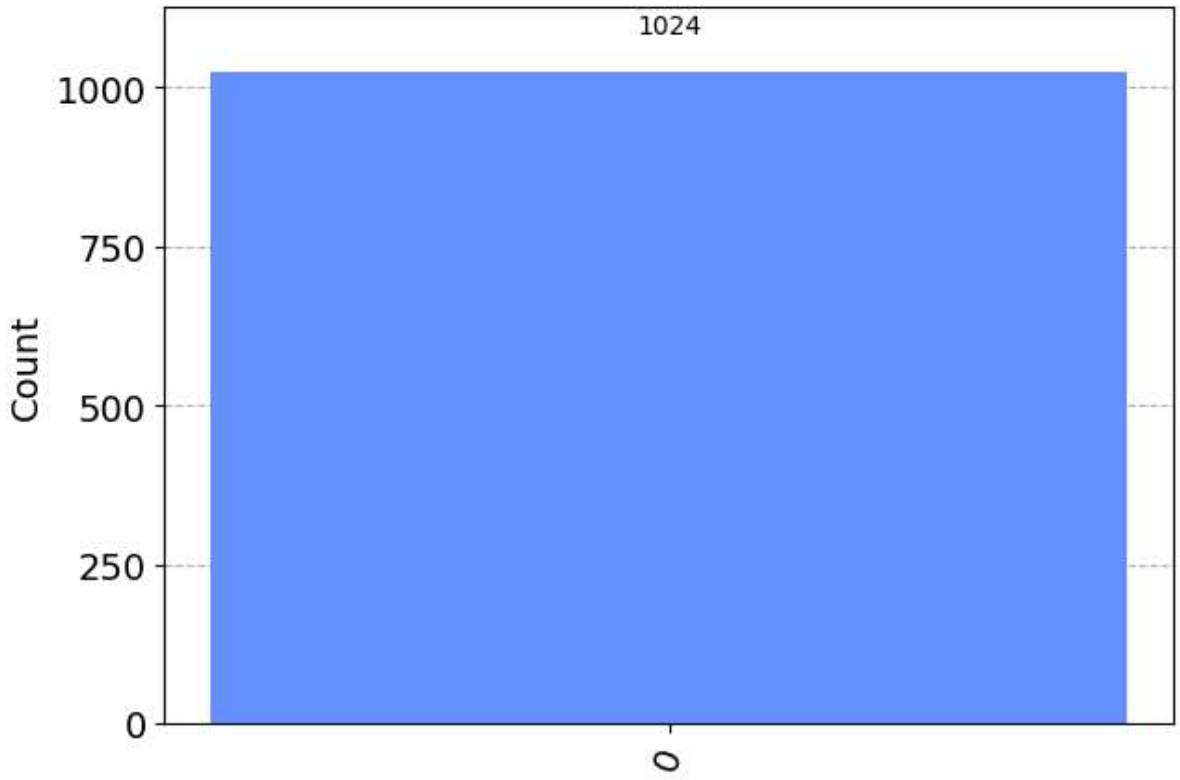
Out[43]:




In [41]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

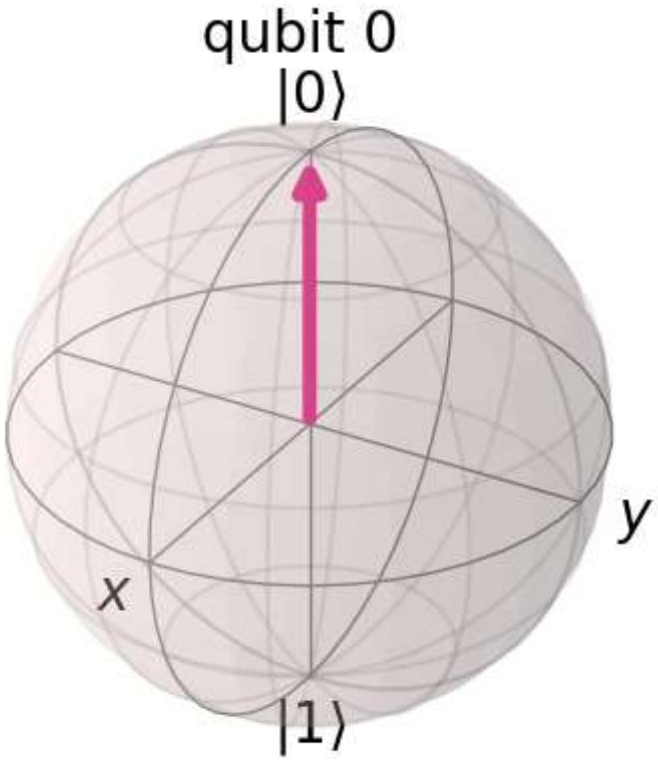
Out[41]:




In [42]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[42]:




the Z gate, also known as the phase-flip gate, is a fundamental quantum logic gate that applies a phase shift of 180 degrees (π radians) to a qubit's quantum state. Mathematically, the Z gate corresponds to the Pauli-Z matrix, which is a 2x2 matrix that changes the sign of the qubit's |1⟩ state.

The Z gate is important in quantum computing because it, along with the X and Y gates, can generate any single-qubit gate via rotations around the x, y, and z axes of the Bloch sphere. It is also used in various quantum algorithms, such as the quantum phase estimation algorithm and the quantum amplitude amplification algorithm.

The Z gate is also a component of more complex quantum gates, such as the controlled phase gate, which is a two-qubit gate that applies a phase shift to the target qubit's state based on the control qubit's state. The controlled phase gate is important for implementing quantum algorithms like the quantum Fourier transform and for constructing quantum error-correcting codes.

## the Hadamard gate.

In [44]:
```
#Implementation of the Hadamard gate.
qc = QuantumCircuit(q)
qc.h(q)
qc.draw()
```

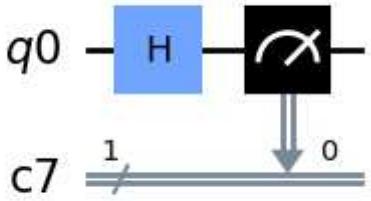
Out[44]:


In [45]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[45]:
```
array([[ 0.707+0.j,  0.707-0.j],
       [ 0.707+0.j, -0.707+0.j]])
```

In [46]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.h(0)
qc.measure(q,c)
qc.draw()
```

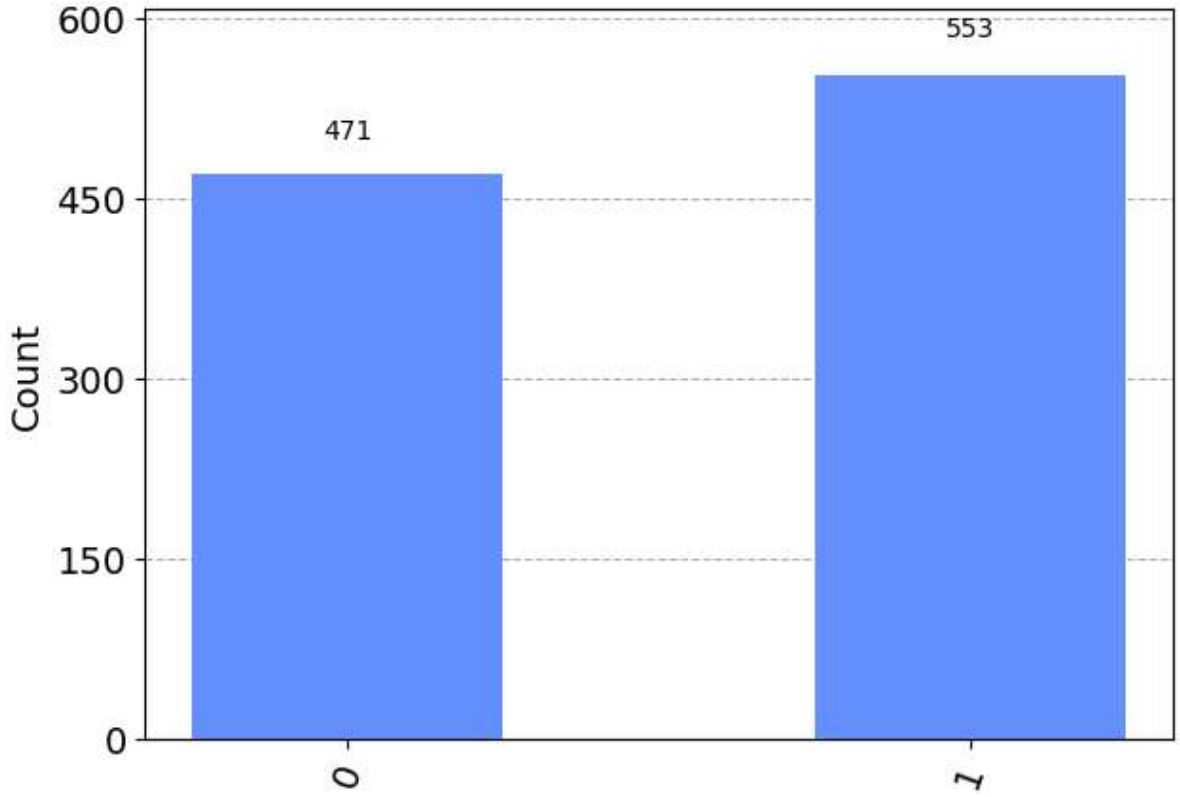
Out[46]:




In [47]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 471, '1': 553}
```

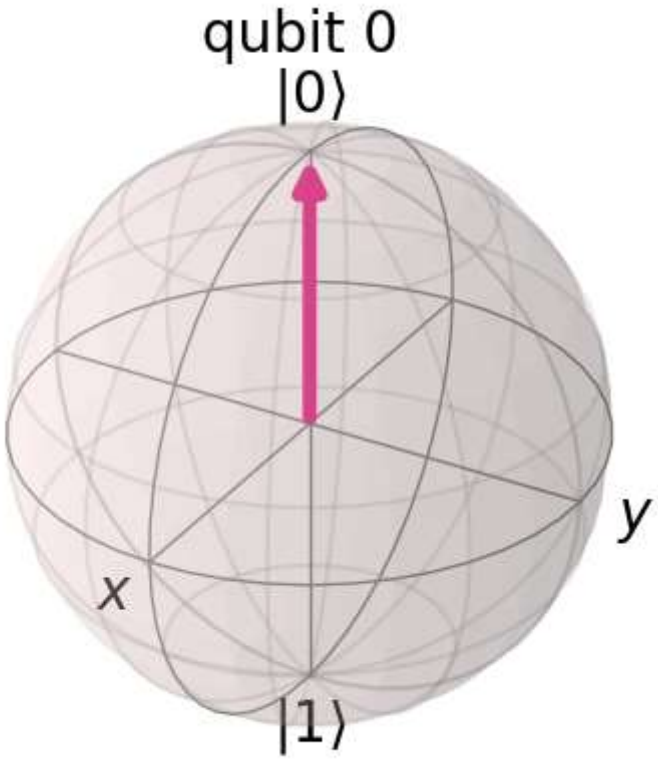
Out[47]:




In [48]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[48]:



The Hadamard gate is a basic gate in quantum computing that is used to put a qubit (quantum bit) into a superposition state. When a qubit is in a superposition state, it can represent both 0 and 1 at the same time, allowing for parallel computation and increased efficiency.

The Hadamard gate takes a single qubit as input and applies a transformation to it that maps the basis states |0⟩ and |1⟩ to equal superpositions of these states. Specifically, it maps |0⟩ to (|0⟩+|1⟩)/√2 and |1⟩ to (|0⟩-|1⟩)/√2.

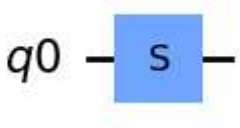
In other words, if you start with a qubit in the state |0⟩ and apply a Hadamard gate to it, you get a qubit in the state (|0⟩+|1⟩)/√2, which is a superposition of |0⟩ and |1⟩. Similarly, if you start with a qubit in the state |1⟩ and apply a Hadamard gate to it, you get a qubit in the state (|0⟩-|1⟩)/√2, which is another superposition of |0⟩ and |1⟩.

The Hadamard gate is often used in quantum algorithms, such as the famous quantum algorithm for database search (Grover's algorithm), as well as in quantum error correction and quantum cryptography.

# The S gate

In [50]:
```
#Implementation of S gate,
qc = QuantumCircuit(q)
qc.s(q)
qc.draw()
```

Out[50]:

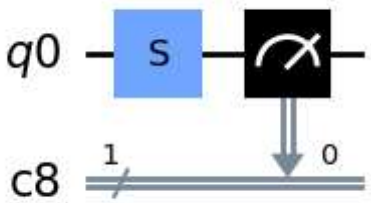



In [51]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[51]:
```
array([[1.+0.j, 0.+0.j],
       [0.+0.j, 0.+1.j]])
```

In [52]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.s(0)
qc.measure(q,c)
qc.draw()
```

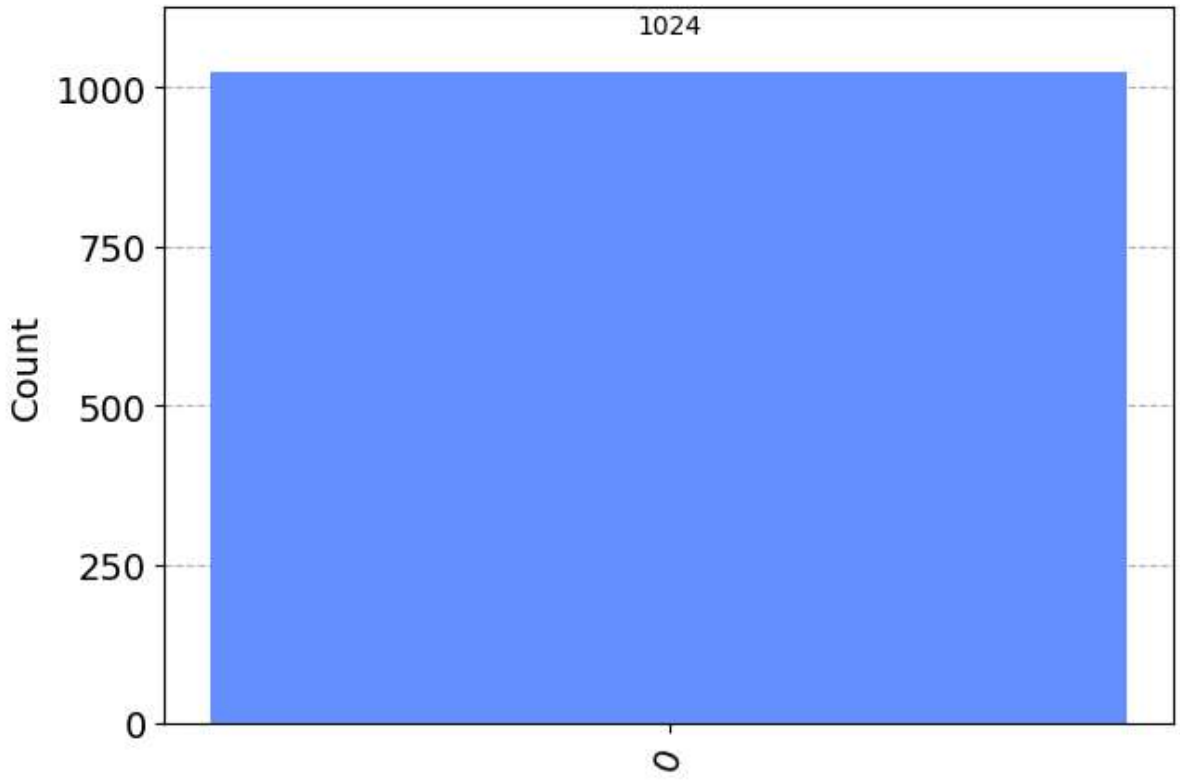
Out[52]:




In [53]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

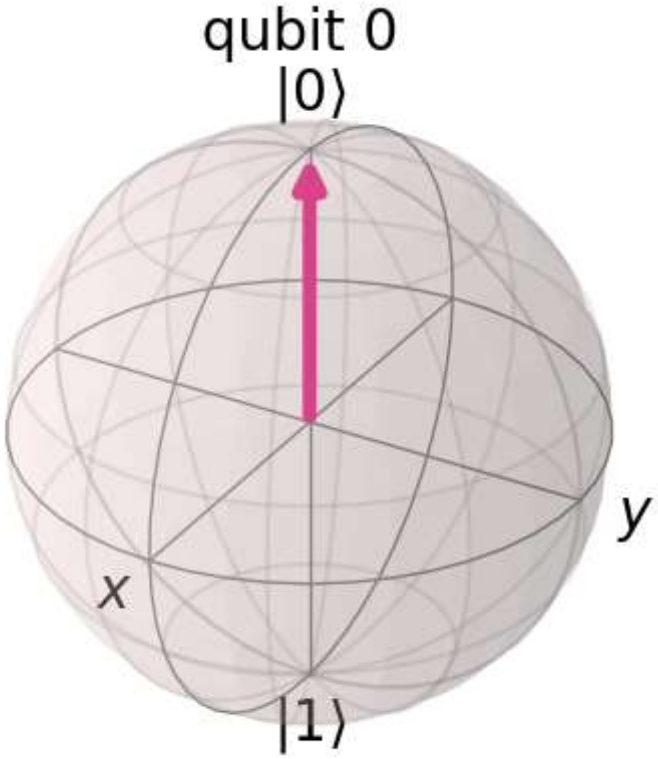
Out[53]:




In [54]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[54]:




The S gate, also known as the phase gate, is a basic gate in quantum computing that introduces a phase shift of 90 degrees to a qubit. It is a single-qubit gate that is represented by the matrix:

S gate = [1 0; 0 i]

where i is the imaginary unit.

The S gate operates on a single qubit and maps the basis state |0⟩ to itself, while mapping the basis state |1⟩ to a state with a phase shift of 90 degrees, i.e., the state |1⟩ is multiplied by i. Thus, the action of the S gate on a qubit can be written as:

S(|0⟩) = |0⟩ S(|1⟩) = i|1⟩

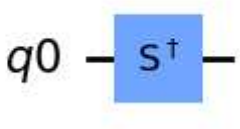
The S gate is a commonly used gate in quantum algorithms, such as quantum phase estimation, which is used to determine the eigenvalues of a unitary operator. It is also used in quantum error correction and in constructing more complex quantum gates.

It is worth noting that the S gate is related to the T gate, which introduces a phase shift of 45 degrees and is used in many quantum algorithms as well. The T gate is essentially the square of the S gate, and can be obtained by applying the S gate twice.

## the S-dagger gate

In [56]:
```
#Implementation of the S-dagger gate
qc = QuantumCircuit(q)
qc.sdg(q)
qc.draw()
```

Out[56]:

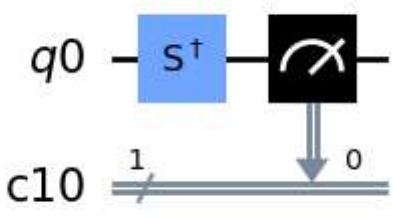



In [57]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[57]:
```
array([[1.+0.j, 0.+0.j],
       [0.+0.j, 0.-1.j]])
```

In [58]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.sdg(0)
qc.measure(q,c)
qc.draw()
```

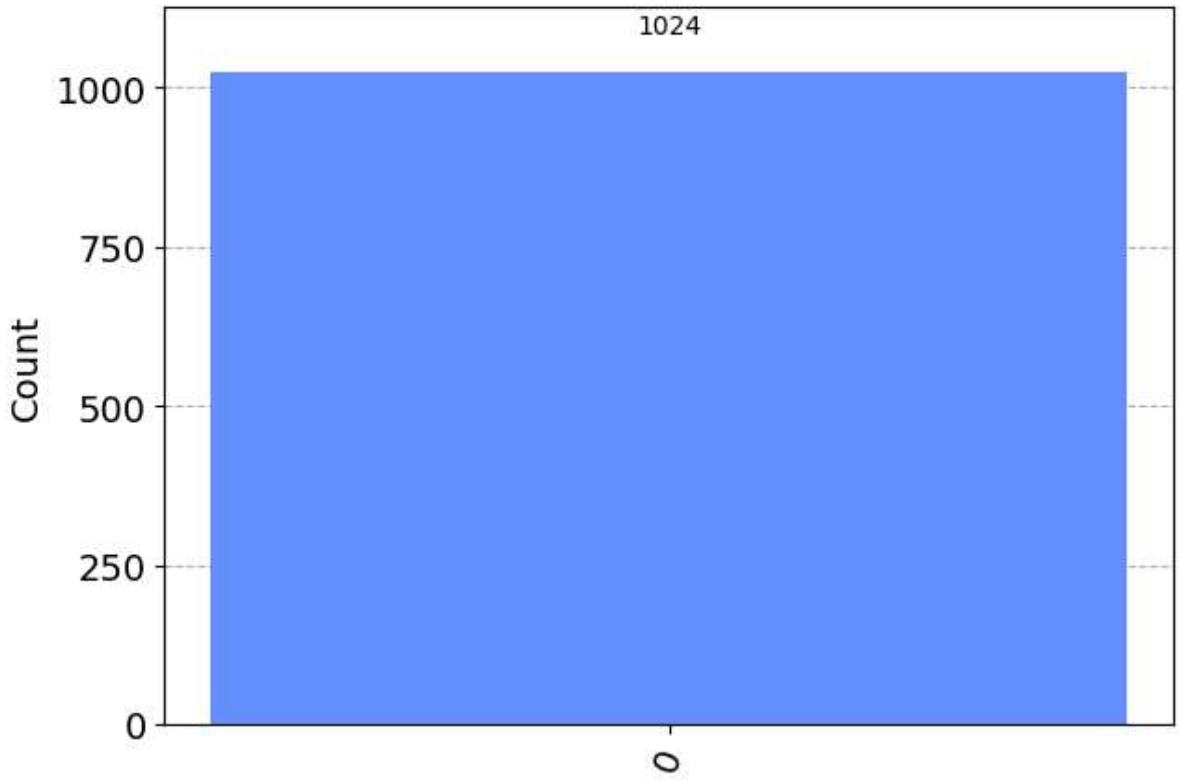
Out[58]:




In [59]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

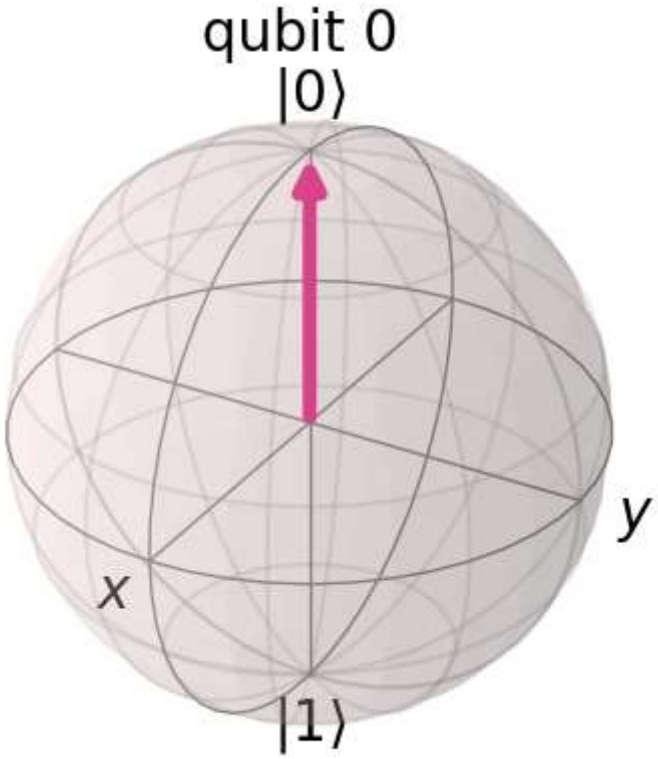
Out[59]:




In [60]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[60]:



The S-dagger gate, also known as the inverse phase gate, is the inverse of the S gate in quantum computing. It is a single-qubit gate that introduces a phase shift of -90 degrees to a qubit, and is represented by the matrix:

S-dagger gate = [1 0; 0 -i]

where i is the imaginary unit.

The S-dagger gate operates on a single qubit and maps the basis state |0⟩ to itself, while mapping the basis state |1⟩ to a state with a phase shift of -90 degrees, i.e., the state |1⟩ is multiplied by -i. Thus, the action of the S-dagger gate on a qubit can be written as:

S-dagger(|0⟩) = |0⟩ S-dagger(|1⟩) = -i|1⟩

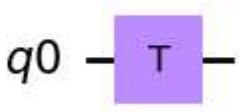
The S-dagger gate is the inverse of the S gate, in the sense that applying the S-dagger gate followed by the S gate (or vice versa) results in the identity operation. This property makes the S-dagger gate useful in quantum algorithms such as quantum phase estimation, where it can be used to "undo" the effect of the S gate.

The S-dagger gate is also used in quantum error correction and in constructing more complex quantum gates.

# the T gate.

In [61]:
```
#Implementation of the T gate.
qc = QuantumCircuit(q)
qc.t(q)
qc.draw()
```

Out[61]:

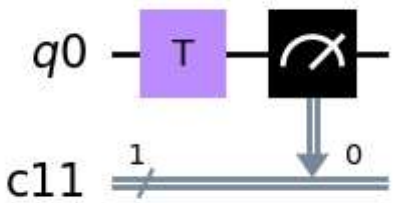



In [62]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[62]:
```
array([[1.   +0.j   , 0.   +0.j   ],
       [0.   +0.j   , 0.707+0.707j]])
```

In [63]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.t(0)
qc.measure(q,c)
qc.draw()
```

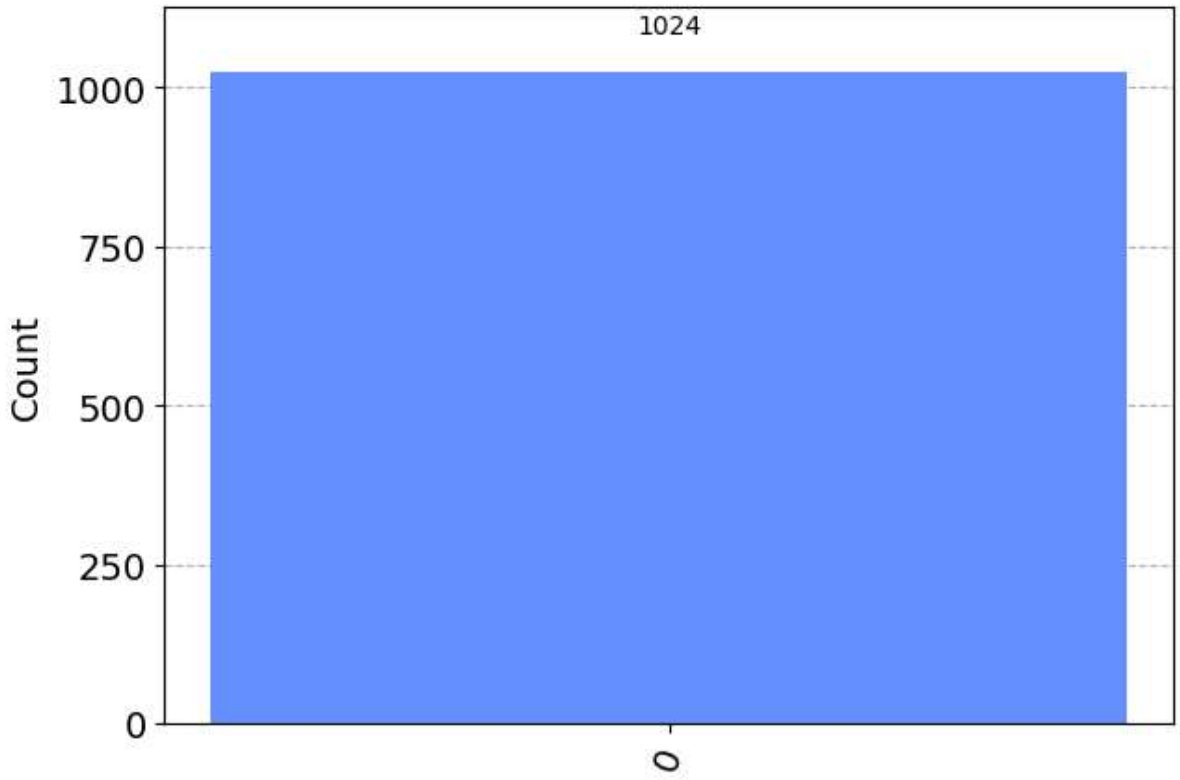
Out[63]:




In [64]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

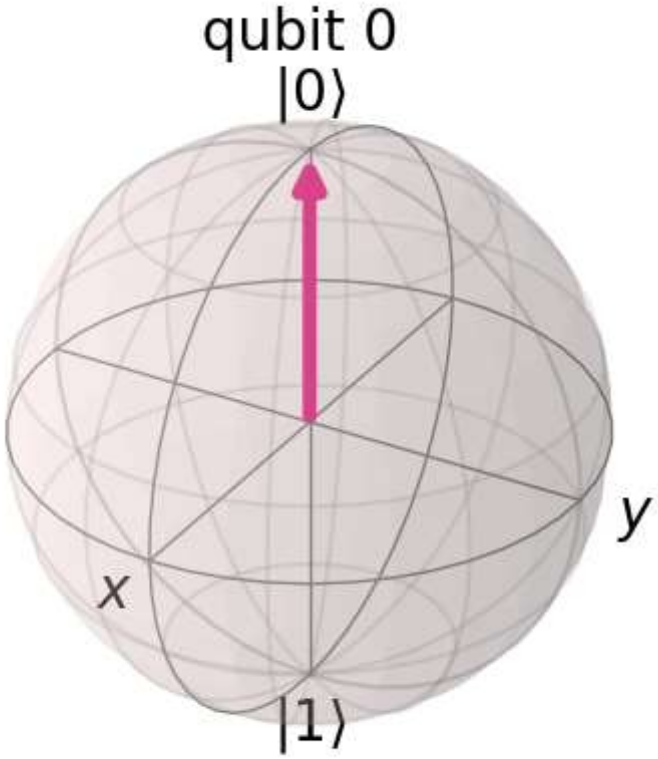
Out[64]:




In [65]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[65]:




The T gate is a single-qubit gate in quantum computing that introduces a phase shift of 45 degrees to a qubit. It is represented by the matrix:

T gate = [1 0; 0 e^(iπ/4)]

where e^(iπ/4) is a complex number that represents a phase shift of 45 degrees.

The T gate operates on a single qubit and maps the basis state |0⟩ to itself, while mapping the basis state |1⟩ to a state with a phase shift of 45 degrees. Thus, the action of the T gate on a qubit can be written as:

T(|0⟩) = |0⟩ T(|1⟩) = e^(iπ/4)|1⟩

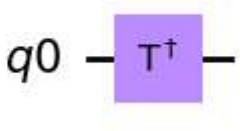
The T gate is related to the S gate, which introduces a phase shift of 90 degrees. In fact, the T gate is simply the square root of the S gate, meaning that applying the T gate twice gives the same result as applying the S gate once. The T gate is also related to the S-dagger gate, which introduces a phase shift of -90 degrees, in the sense that applying the T gate followed by the T-dagger gate (or vice versa) results in the identity operation.

The T gate is used in many quantum algorithms, such as quantum phase estimation and quantum Fourier transform, as well as in quantum error correction and in constructing more complex quantum gates.

## the T dagger gate.

In [66]:
```
#Implementation of the T dagger gate.
qc = QuantumCircuit(q)
qc.tdg(q)
qc.draw()
```

Out[66]:

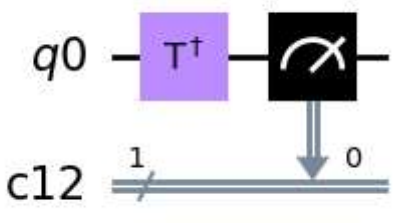



In [67]:
```
job = backend_unitary.run(transpile(qc, backend_unitary))
job.result().get_unitary(qc, decimals=3)
```

Out[67]:
```
array([[1.   +0.j   , 0.   +0.j   ],
       [0.   +0.j   , 0.707-0.707j]])
```

In [69]:
```
c = ClassicalRegister(1)
qc = QuantumCircuit(q,c)
qc.tdg(0)
qc.measure(q,c)
qc.draw()
```

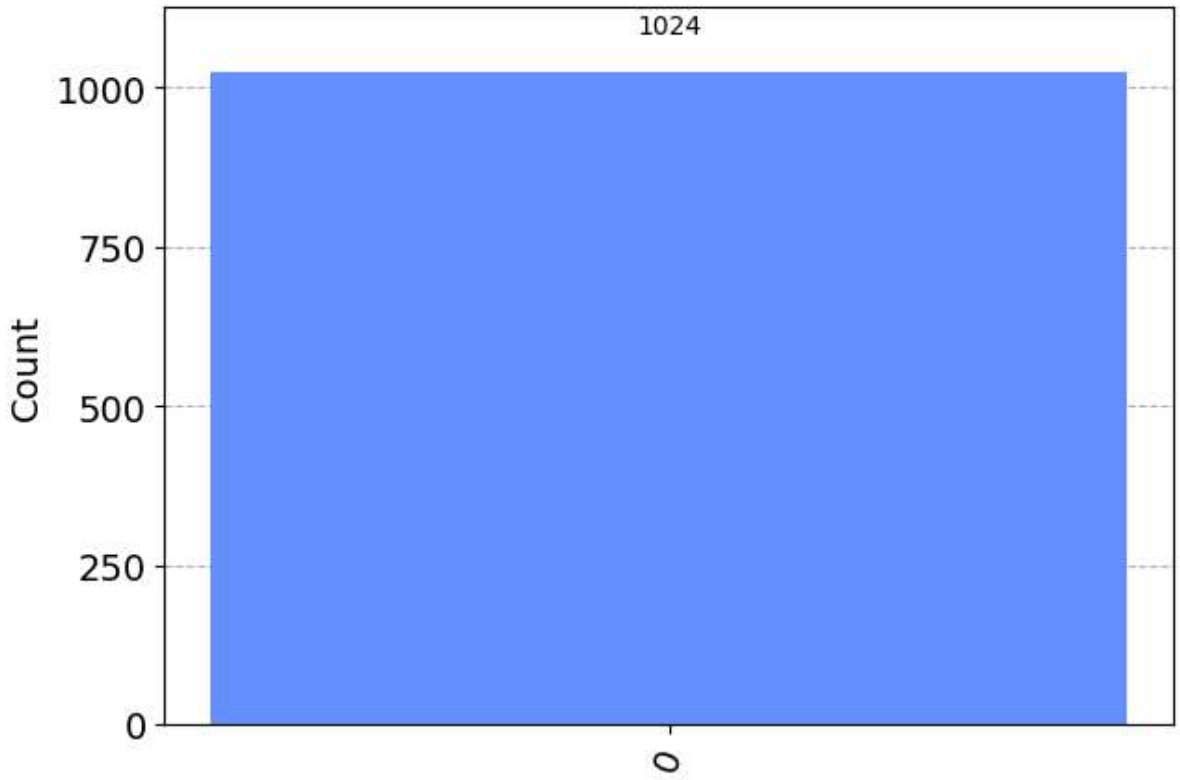
Out[69]:




In [70]:
```
job = backend_qasm.run(transpile(qc, backend_qasm))
counts=job.result().get_counts()
print(counts)
plot_histogram(counts)
```

```
{'0': 1024}
```

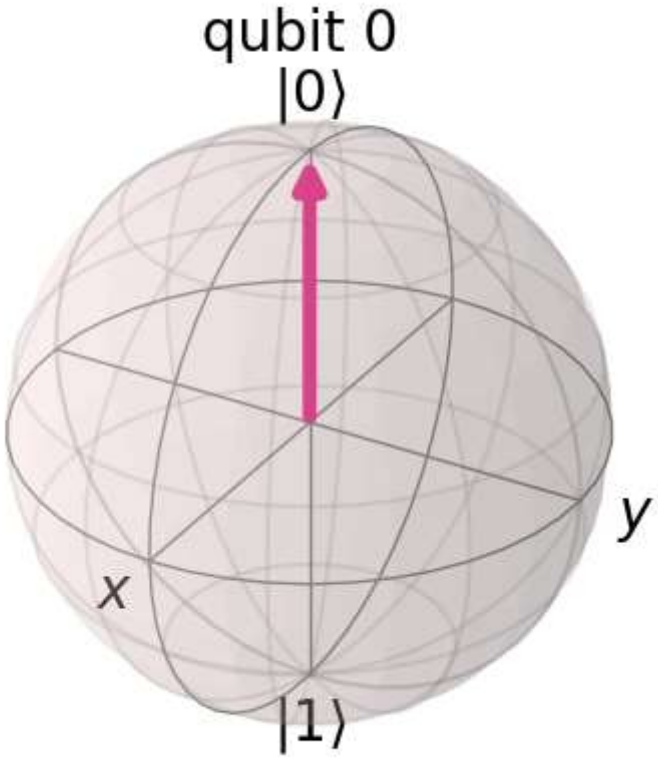
Out[70]:




In [71]:
```
job = backend_statevector.run(transpile(qc, backend_statevector))
sv=job.result().get_statevector()
print(sv)
plot_bloch_multivector(sv)
```

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

Out[71]:



The T-dagger gate, also known as the inverse T gate, is the inverse of the T gate in quantum computing. It is a single-qubit gate that introduces a phase shift of -45 degrees to a qubit, and is represented by the matrix:

T-dagger gate = [1 0; 0 e^(-iπ/4)]

where e^(-iπ/4) is a complex number that represents a phase shift of -45 degrees.

The T-dagger gate operates on a single qubit and maps the basis state |0⟩ to itself, while mapping the basis state |1⟩ to a state with a phase shift of -45 degrees. Thus, the action of the T-dagger gate on a qubit can be written as:

T-dagger(|0⟩) = |0⟩ T-dagger(|1⟩) = e^(-iπ/4)|1⟩

The T-dagger gate is the inverse of the T gate, in the sense that applying the T-dagger gate followed by the T gate (or vice versa) results in the identity operation. This property makes the T-dagger gate useful in quantum algorithms such as quantum phase estimation, where it can be used to "undo" the effect of the T gate.

The T-dagger gate is also used in quantum error correction and in constructing more complex quantum gates.

In [ ]: