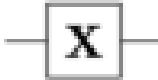

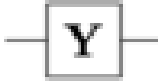




```
In [1]: %matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.tools.monitor import job_monitor
from qiskit.visualization import *
from math import pi
from qiskit.visualization import plot_bloch_multivector
from qiskit.visualization import plot_state_qsphere
from qiskit.quantum_info import Statevector
from IPython.core.display import Image, display # for web image to upload in jupyter
```

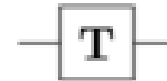
```
In [2]: display(Image('https://upload.wikimedia.org/wikipedia/commons/thumb/e/e0/Quantum_Logic_Gates.png/500px-Quantum_Logic_Gates.png', width=800, unconfined=True))
```

Operator	Gate(s)
Pauli-X (X)	 
Pauli-Y (Y)	
Pauli-Z (Z)	
Hadamard (H)	

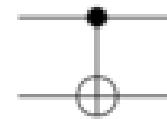
Phase (S, P)



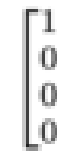
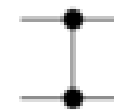
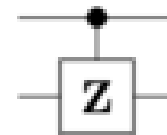
$\pi/8$  (T)



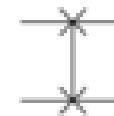
Controlled Not  
(CNOT, CX)



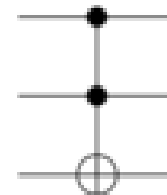
Controlled Z (CZ)



SWAP



Toffoli  
(CCNOT,  
CCX, TOFF)



In classical computers, information is represented as the binary digits 0 or 1. These are called bits. In fact, every classical computer translates these bits into the human readable information on your electronic device. The word document you read or video you watch is encoded in the computer binary language in terms of these 1's and 0's.

## Single qubits

**Quantum bits**, called qubits, are similar to bits in that there are two measurable states called the 0 and 1 states. However, unlike classical bits, qubits can also be in a superposition state of these 0 and 1 states. The state of a qubit is enclosed in the right half of an angled bracket, called the "ket". A qubit  $|\psi\rangle$ , could be in a  $|0\rangle$  or  $|1\rangle$  state which is a superposition of both  $|0\rangle$  and  $|1\rangle$ . This is written as

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where  $\alpha$  and  $\beta$  are called the amplitudes of the states.

Amplitudes are very important because they tell us the probability of finding the particle in that specific state when performing a measurement. The probability of measuring the particle in state  $|0\rangle$  is  $\alpha^2$ , and the probability of measuring the particle in state  $|1\rangle$  is  $\beta^2$ . The matrix representation of qubits are

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

</font>

# 1. Pauli X-gate

The  $X$  gate has the matrix

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

and this is the Pauli X matrix, named after Wolfgang Pauli. It has the property that

$$\sigma_x|0\rangle = |1\rangle \quad \text{and} \quad \sigma_x|1\rangle = |0\rangle$$

It "flips" between  $|0\rangle$  and  $|1\rangle$ .

For  $|\psi\rangle = a|0\rangle + b|1\rangle$  in  $\mathbb{C}^2$ ,

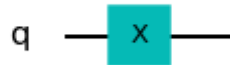
$$X|\psi\rangle = b|0\rangle + a|1\rangle$$

The Pauli-X gate is a single qubit operation gate that is similar to the classical not gate. It takes a value and flips it to the opposite one. It maps  $|0\rangle$  to  $|1\rangle$  and  $|1\rangle$  to  $|0\rangle$ .

Here we assume the initialization to be Zero which is zero by default

```
In [3]: qcx= QuantumCircuit(1)
        qcx.x(0)
        qcx.draw('mpl')
```

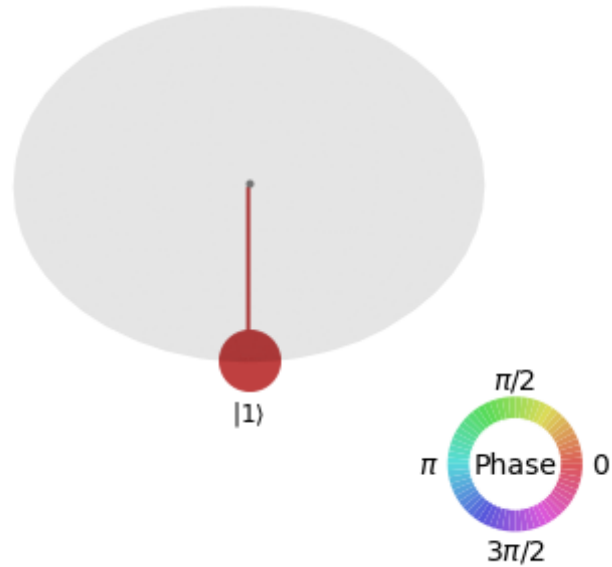
Out[3]:



```
In [4]: sv = Statevector.from_label('0')
        new_sv = sv.evolve(qcx)
```

```
plot_state_qsphere(new_sv.data)
```

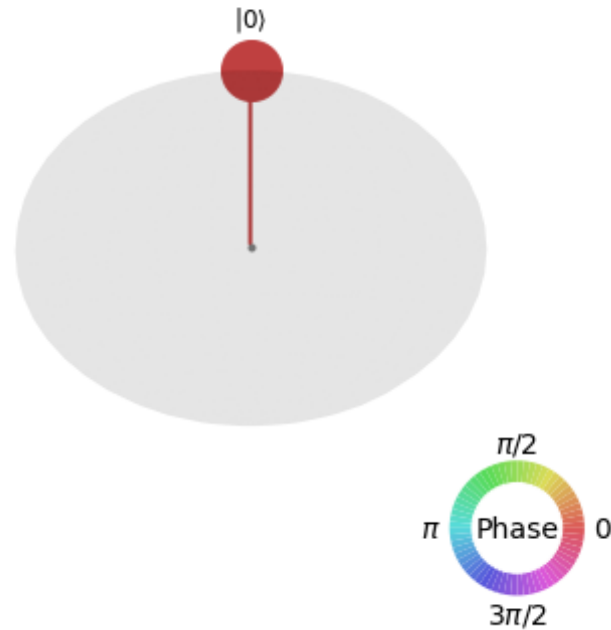
Out[4]:



Here we initialize the circuit to be ket one

```
In [5]: sv = Statevector.from_label('1')  
new_sv = sv.evolve(qcx)  
plot_state_qsphere(new_sv.data)
```

Out[5]:



## 2. Pauli Y-gate

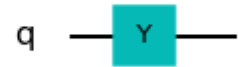
The Y gate has the matrix

$$\sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = i \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

and this is the Pauli Y matrix. The Pauli-Y gate is a single qubit operation. It maps  $|0\rangle$  to  $-i|1\rangle$  and  $|1\rangle$  to  $i|0\rangle$ . It equates to a rotation around the Y-axis of the Bloch sphere by  $\pi$  radians.

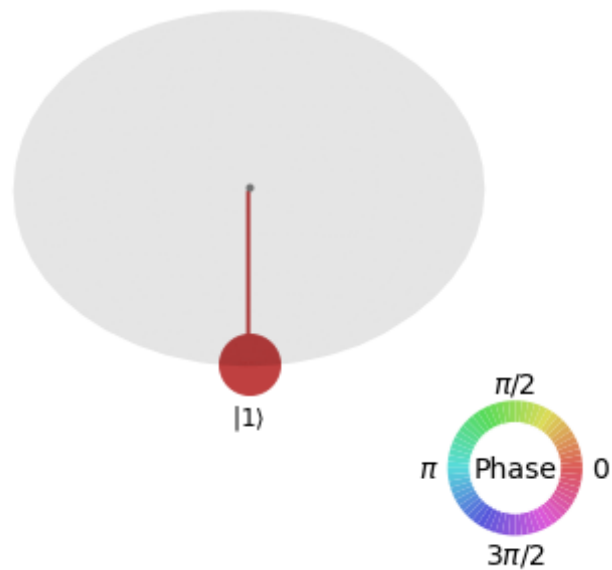
```
In [6]: qcy = QuantumCircuit(1)
qcy.y(0)
qcy.draw('mpl')
```

Out[6]:



```
In [7]: sv = Statevector.from_label('0')
new_sv = sv.evolve(qcy)
plot_state_qsphere(new_sv.data)
```

Out[7]:



### 3. Pauli Z-gate

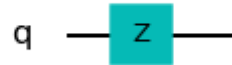
The Z gate has the matrix

$$\sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

and this is the Pauli Z matrix. It rotates qubit states by  $\pi$  around the  $z$  axis on the Bloch sphere. The Pauli-Z gate is a single qubit operation. It maps  $|1\rangle$  to  $-|1\rangle$  and it leaves  $|0\rangle$  unchanged. It equates to a rotation around the Z-axis of the Bloch sphere by pi radians.

```
In [8]: qcz = QuantumCircuit(1)
        qcz.z(0)
        qcz.draw('mpl')
```

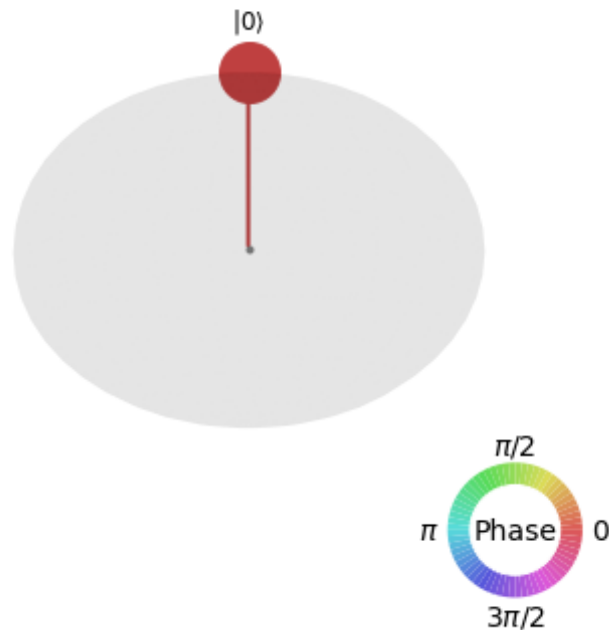
Out[8]:



```
In [9]: sv = Statevector.from_label('0')
        new_sv = sv.evolve(qcz)
        plot_state_qsphere(new_sv.data)
```

Out[9]:





## 4. Hadamard gate

The Hadamard gate (H-gate) is a fundamental quantum gate. It allows us to move away from the poles of the Bloch sphere and create a superposition of  $|0\rangle$  and  $|1\rangle$ . It has the matrix:

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

We can see that this performs the transformations below:

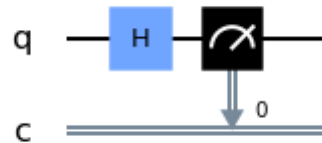
$$H|0\rangle = |+\rangle$$

$$H|1\rangle = |-\rangle$$

**Hadamard gates gives the output of superposition of both states with equal probabilities**

```
In [39]: qc = QuantumCircuit(1,1)
          qc.h(0)
          qc.measure(0,0)
          qc.draw("mpl")
```

Out[39]:

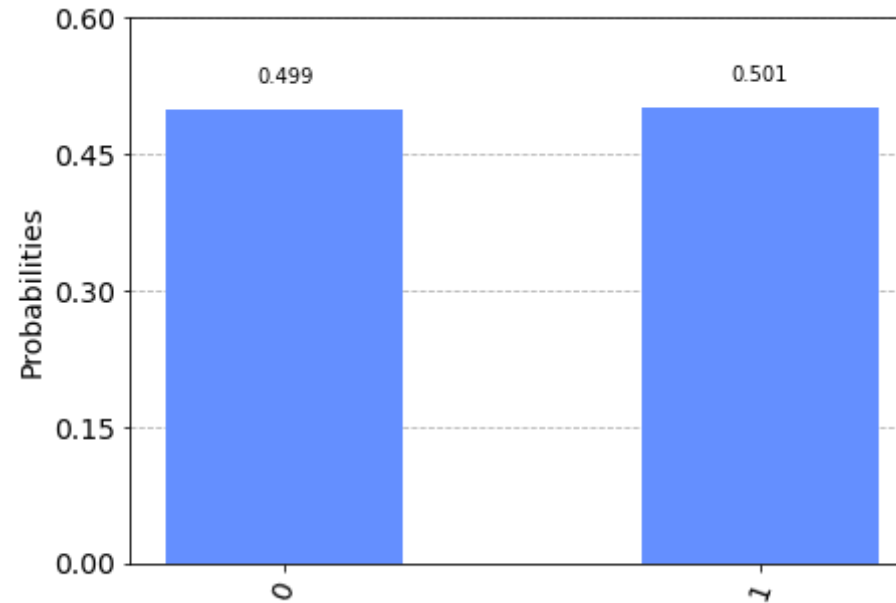


```
In [40]: def run_circuit(qc2):
          backend = Aer.get_backend('qasm_simulator') # we choose
              the simulator as our backend
          result = execute(qc, backend, shots = 1000).result() # w
              e run the simulation
          counts = result.get_counts() # we get the counts
          return counts

          counts = run_circuit(qc)
          print(counts)
          plot_histogram(counts) # let us plot a histogram to see the
              possible outcomes and corresponding probabilities
```

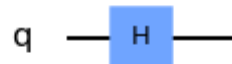
```
{'0': 499, '1': 501}
```

Out[40]:



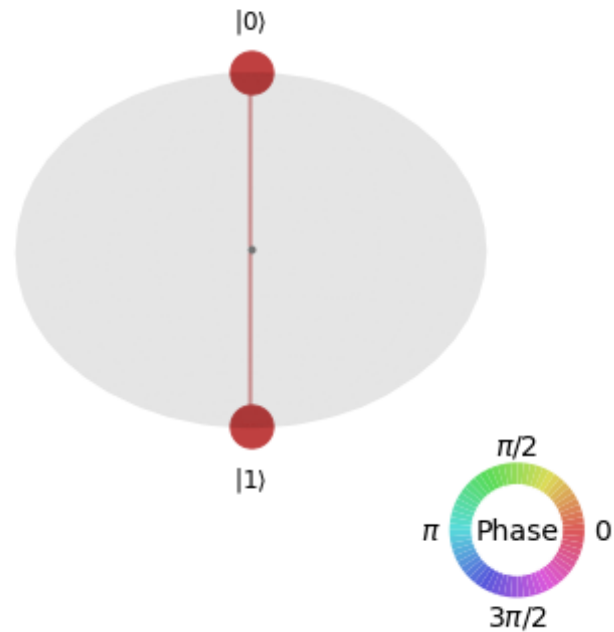
```
In [12]: qch = QuantumCircuit(1)
qch.h(0)
qch.draw('mpl')
```

Out[12]:



```
In [13]: sv = Statevector.from_label('0')
new_sv = sv.evolve(qch)
plot_state_qsphere(new_sv.data)
```

Out[13]:



We have the state

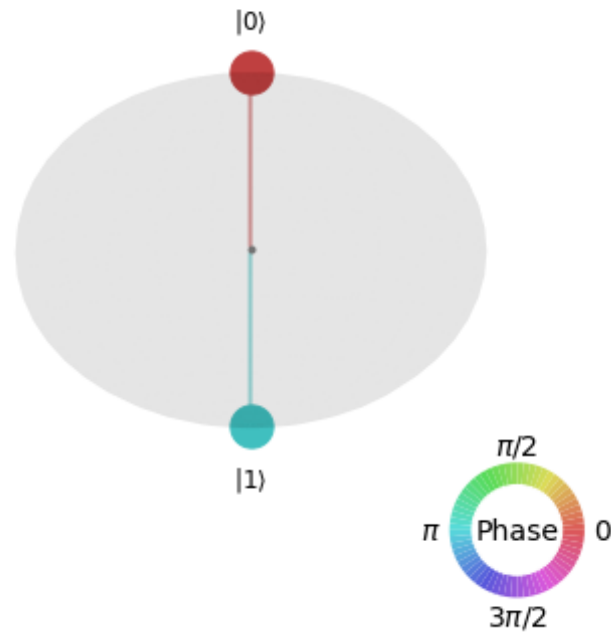
$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

which is termed as ket plus state as

$$H|0\rangle = |+\rangle$$

```
In [14]: sv = Statevector.from_label('1')
          new_sv = sv.evolve(qch)
          plot_state_qsphere(new_sv.data)
```

Out[14]:



This time, the bottom circle, corresponding to the basis state  $|1\rangle$  has a different color corresponding to the phase of  $\phi = \pi$ . This is because the coefficient of  $|1\rangle$  in the state

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

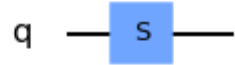
is  $-1$ , which is equal to  $e^{i\pi}$ . The given state is termed as ket minus as

$$H|1\rangle = |-\rangle$$

## 5. S or P Gate

```
In [15]: qcp = QuantumCircuit(1)
         qcp.s(0)
         qcp.draw('mpl')
```

Out[15]:



```
In [16]: sv = Statevector.from_label('0')
         new_sv = sv.evolve(qcp)
         new_sv
         #plot_state_qsphere(new_sv.data)
```

Out[16]: Statevector([1.+0.j, 0.+0.j],  
dims=(2,))

```
In [17]: sv = Statevector.from_label('1')
         new_sv = sv.evolve(qcp)
         plot_state_qsphere(new_sv.data)
         new_sv
```

Out[17]: Statevector([0.+0.j, 0.+1.j],  
dims=(2,))

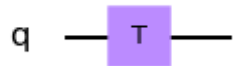
Here P gate do nothing on ket zero but it changes phases by 90 degree on ket one

## 6. T-gate

The T-gate is a very commonly used gate, it is an  $R_\phi$ -gate with  $\phi=\pi/4$

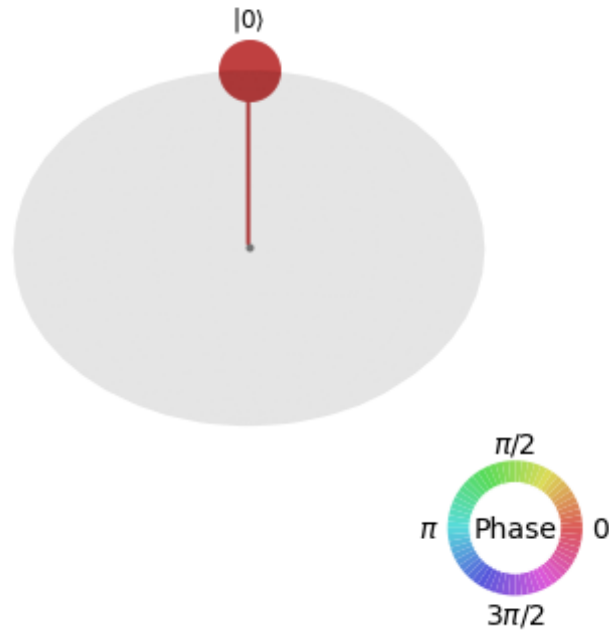
```
In [18]: qct = QuantumCircuit(1)
          qct.t(0)
          qct.draw('mpl')
```

Out[18]:



```
In [19]: sv = Statevector.from_label('0')
          new_sv = sv.evolve(qct)
          plot_state_qsphere(new_sv.data)
```

Out[19]:



```
In [20]: sv = Statevector.from_label('1')
new_sv = sv.evolve(qct)
new_sv
#plot_state_qsphere(new_sv.data)
```

```
Out[20]: Statevector([0.          +0.j          , 0.70710678+0.70710678j],
                    dims=(2,))
```

## Multiqubit gates



# 1. Controlled Gates

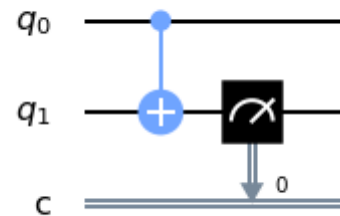
The controlled gate is a 2 or more qubit operation, where 1 or more qubits act as a control for some operation on a qubit. (eg: cx, cy, cz gates)

## 1(a) Controlled x gate

The controlled-x gate acts on 2 qubits and performs the NOT operation on the second qubit only when the first qubit is  $|1\rangle$ .

```
In [21]: qccx=QuantumCircuit(2,1)
qccx.cx(0,1)
qccx.measure(1,0)
qccx.draw("mpl")
```

Out[21]:



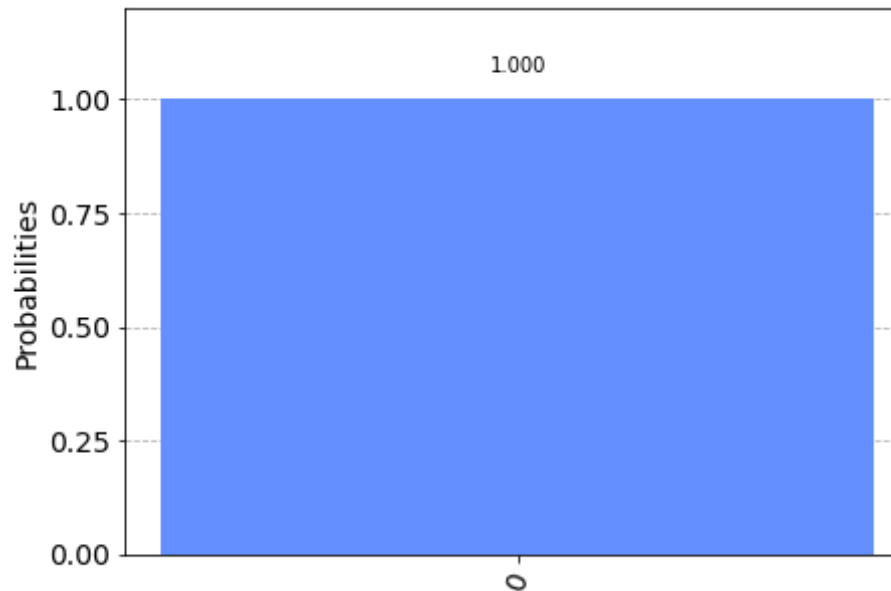
```
In [22]: def run_circuit(qccx):
```

```
backend = Aer.get_backend('qasm_simulator') # we choose
the simulator as our backend
result = execute(qccx, backend, shots = 1000).result() #
we run the simulation
counts = result.get_counts() # we get the counts
return counts
```

```
counts = run_circuit(qccx)
print(counts)
plot_histogram(counts)
```

```
{'0': 1000}
```

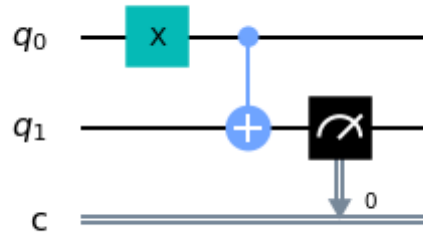
Out[22]:



Here the probability of second qubit which is zero is still zero when controlled is zero.

```
In [23]: qccx=QuantumCircuit(2,1)
qccx.x(0)
qccx.cx(0,1)
qccx.measure(1,0)
qccx.draw("mpl")
```

Out[23]:



**Here in the above circuit we make first qubit one applying X gate**

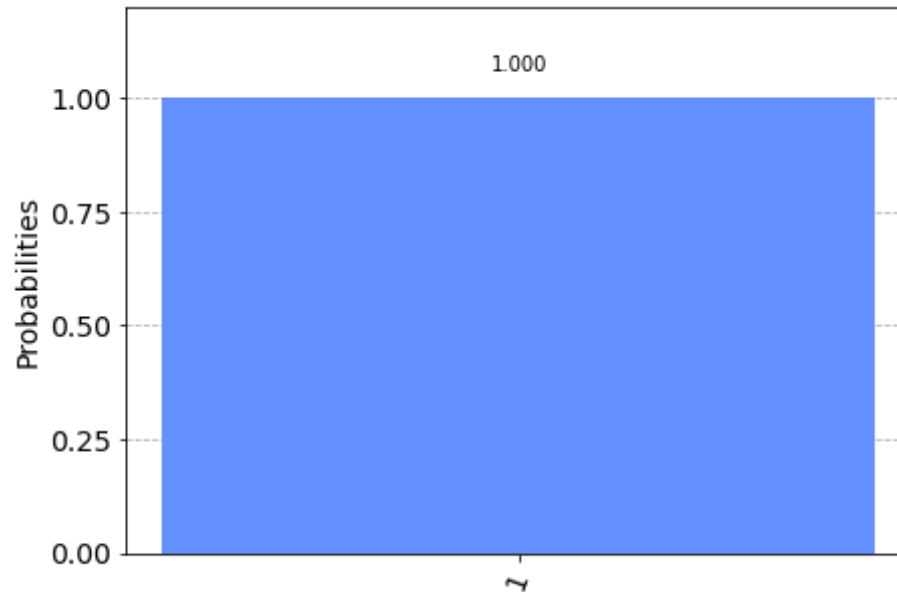
```
In [24]: def run_circuit(qccx):
    backend = Aer.get_backend('qasm_simulator') # we
    choose the simulator as our backend
    result = execute(qccx, backend, shots = 1000).res
    ult() # we run the simulation
    counts = result.get_counts() # we get the counts
    return counts

counts = run_circuit(qccx)
```

```
print(counts)
plot_histogram(counts)
```

```
{'1': 1000}
```

Out[24]:

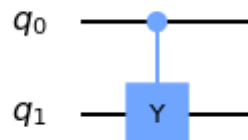


Here the second qubit which is zero at first has converted to 1 when controlled is one

## 1(b) Controlled y gate

```
In [25]: qccy=QuantumCircuit(2)
qccy.cy(0,1)
qccy.draw('mpl')
```

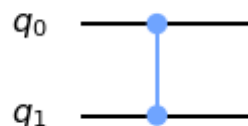
Out[25]:



## 1(c) Controlled z gate

```
In [26]: qccz=QuantumCircuit(2)
qccz.cz(0,1)
qccz.draw('mpl')
```

Out[26]:



## 1(d) Controlled Not Gate

```
In [27]: qccn=QuantumCircuit(2)
qccn.cnot(0,1)
qccn.draw('mpl')
```

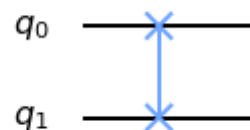
Out[27]:



## 2.SWAP Gate

```
In [28]: qcs=QuantumCircuit(2)
          qcs.swap(0,1)
          qcs.draw('mpl')
```

Out[28]:



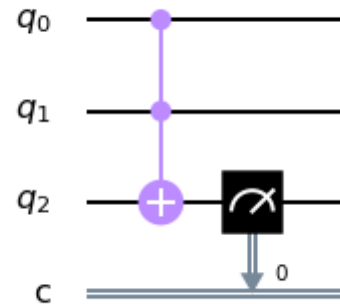
## 3.Toffoli Gate or CCX Gate

It is also known as the "controlled-controlled-not" gate, which describes its action. It has 3-bit inputs and outputs; if the first two bits are both set to 1, it inverts the third bit, otherwise all bits stay the same.

```
In [29]: qc = QuantumCircuit(3,1)
```

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

Out[29]:

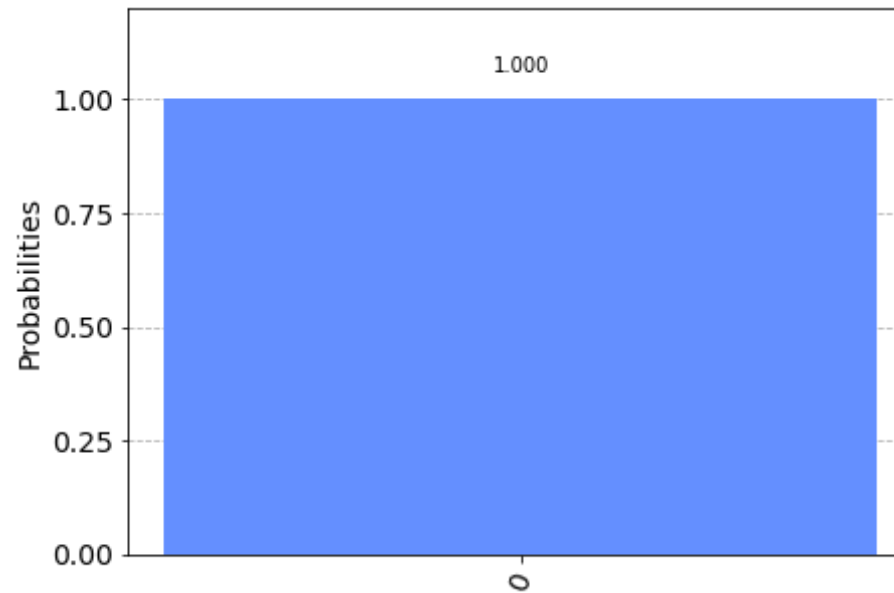


```
In [30]: def run_circuit(qc):
          backend = Aer.get_backend('qasm_simulator') # we choose
              the simulator as our backend
          result = execute(qc, backend, shots = 1000).result() # w
              e run the simulation
          counts = result.get_counts() # we get the counts
          return counts

counts = run_circuit(qc)
print(counts)
plot_histogram(counts)

{'0': 1000}
```

Out[30]:

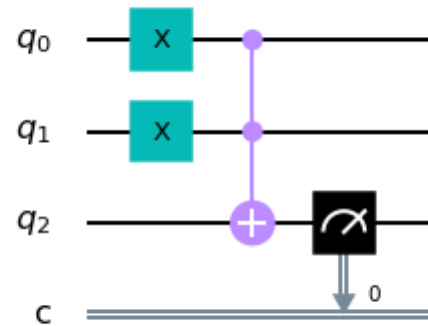


since both qubits 0 and 1 are at zero state so the output of qubit 2 is zero

```
In [31]: qc = QuantumCircuit(3,1)
          qc.x(0)
          qc.x(1)
          qc.ccx(0,1,2)
          qc.measure(2,0)
          qc.draw('mpl')
```

Out[31]:





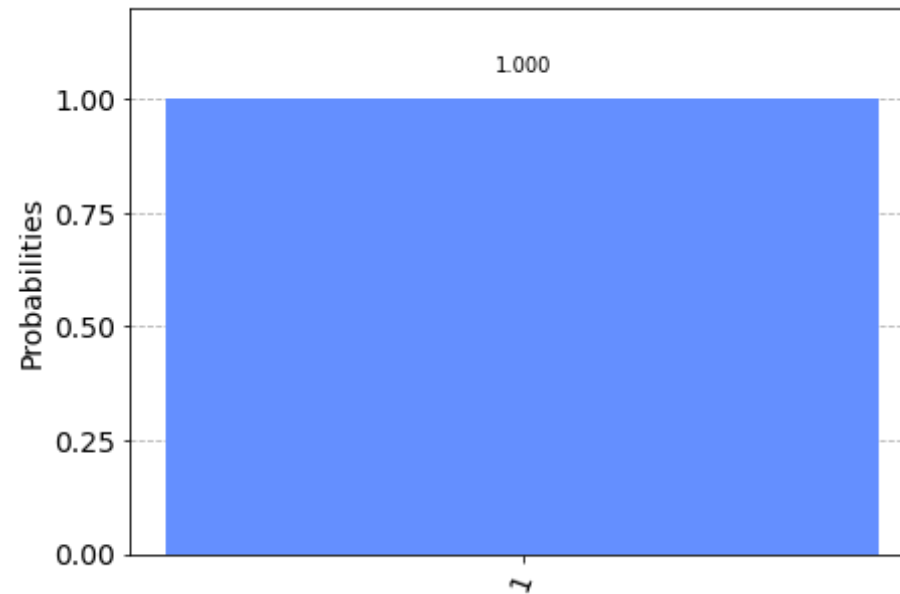
Here we make the 0 and 1 qubits to 1 using bit flip gate i.e x gate

```
In [32]: def run_circuit(qc):
    backend = Aer.get_backend('qasm_simulator') # we choose
    the simulator as our backend
    result = execute(qc, backend, shots = 1000).result() # w
    e run the simulation
    counts = result.get_counts() # we get the counts
    return counts

counts = run_circuit(qc)
print(counts)
plot_histogram(counts)

{'1': 1000}
```

Out[32]:



Here, the output of qubits 2 is changed from the input(0) when both the qubits 0 and 1 is set to 1

In [ ]: