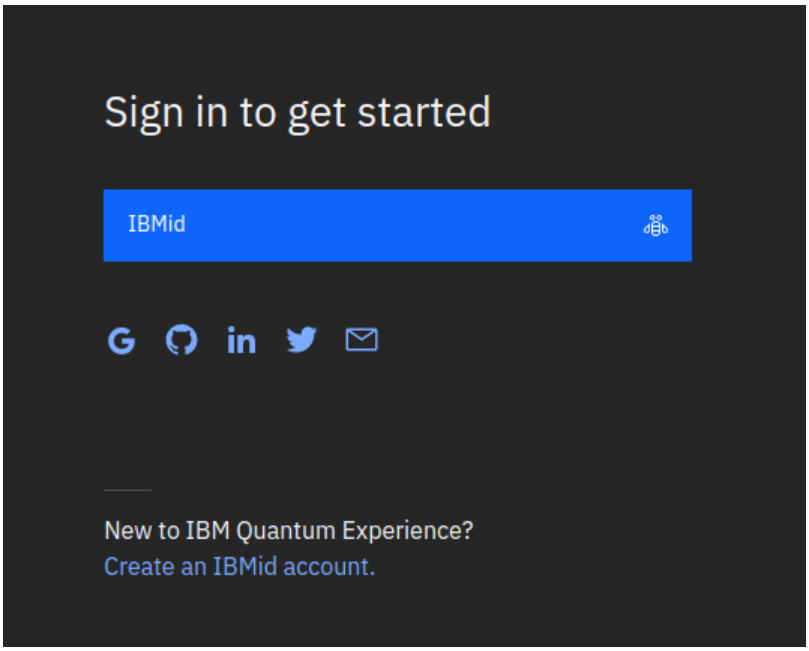
Hands on Quantum Computing Using Qiskit

What we hope to achieve through out this series is to provide you with the needed tools to create your own quantum algorithm, run it on a classical simulator and most importantly, run it on a real quantum computer!

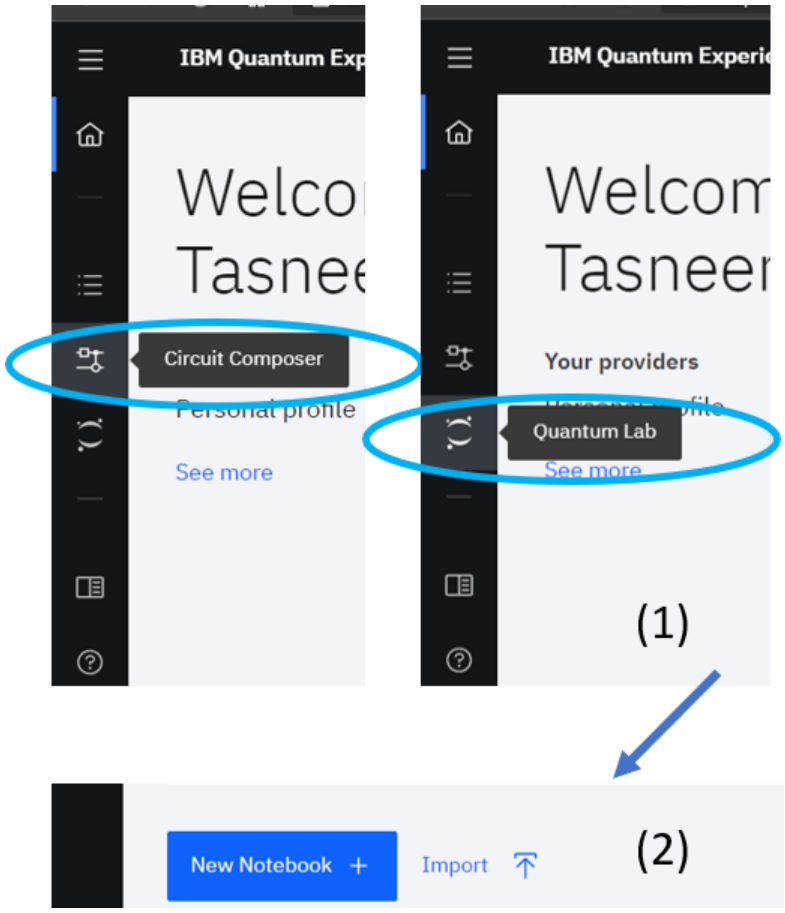
Creating an Account on IBM Experience Website

To create an account on IBM Experience Website, you need to follow these steps: 1) Go to the quantum computing website: https://quantum-computing.ibm.com/ (https://quantum-computing.ibm.com/) 2) hit on "Create an IBMid account"



3) fill the required fields and create your own account!

Once you login to your account, you'll be directed to a page where on the left side you have two important options as seen below: (1) Circuit Composer: Allows you to create quantum circuits graphically (2) Quantum Lab: Here you can create a jupyter notebook online, by clicking on "New Notebook +", which is the option we are using here.



Note: There is no need to install Python/ Qiskit on your computer for using this platform.

Getting Started with Qiskit- 01-Building a Quantum Circuit and Performing Measurements

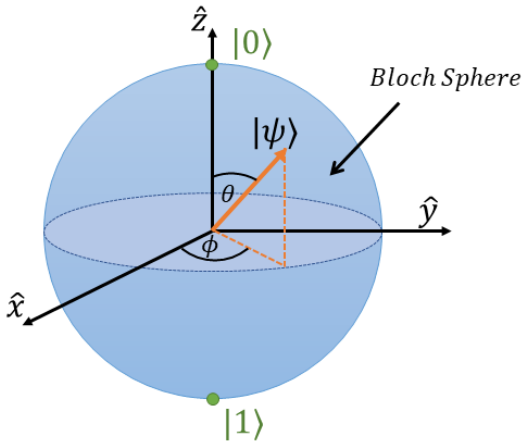
Qubits in Qiskit

Classical Bit

Two values:

- 0
- 1

Qubit- Quantum Bit



General Qubit state: $|\psi\rangle = a|0\rangle + b|1\rangle$

$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} - Computational\ Basis$

In case of a multiple-qubit system, $q_1, q_2, q_3, .. q_n$, the general state will be

$$|\psi\rangle = |q_n q_{n-1} \cdots q_3 q_2 q_1\rangle = \sum_{i=1}^{2^n} a_i |i\rangle$$

where $|i\rangle$ is a computational basis vector, eg. $|010010 \cdots 1\rangle$ and $a_i's$ are complex coefficients.

Qubits ordering in qiskit: Note that the last qubit is the leftmost qubit and the first one is the rightmost qubit in qiskit.

Example: 2-qubit system:

$$|\psi\rangle = |q_2 q_1\rangle = a_1|00\rangle + a_2|01\rangle + a_3|10\rangle + a_4|11\rangle = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix}$$

Building a quantum circuit using qiskit

Two fundamental steps you need to do when programming a quantum computer/ algorithm are building the quantum circuit needed in your algorithm that includes initializing the qubits and applying the gates included in the quantum circuit, and secondly perform the needed measurements that will be used to solve the problem your algorithm is trying to solve, and store this information in classical bits. If so, then the two main things we'll focus on in this section are: 1) Building a quantum circuit- In this section we'll see how to build a quantum circuit using qiskit.

2) Measureing the final results and executing the algorithm- Next section.

Let's see how a 3-qubit quantum circuit can be built using qiskit $\rightarrow$

How many registers will be needed? We need three quantum registers for the three qubits and three classical registers for the classical bits in which we store the measurement results of each one of the qubits. To create these registers we type:

In [1]:
```
%matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

```
/opt/conda/lib/python3.7/site-packages/qiskit/providers/ibmq/ibmqfactory.py:192: UserWarning: Timestamps in IBMQ back
end properties, jobs, and job results are all now in local time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
```

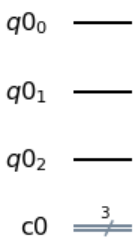
In [2]:
```
from qiskit import QuantumRegister,ClassicalRegister
q = QuantumRegister(3)
#This line creates 3 quantum registers
c = ClassicalRegister(3)
#This line creates 3 classical registers
```

By default, the three qubits will be initialized in the $|0\rangle$ state. To create the quantum circuit, we'll call it qc1 all we need to do is

In [3]:
```
from qiskit import QuantumCircuit
qc1=QuantumCircuit(q,c)
```

It is possible as well to easily draw the circuit we've just created using:

In [4]:
```
qc1.draw('mpl')
```

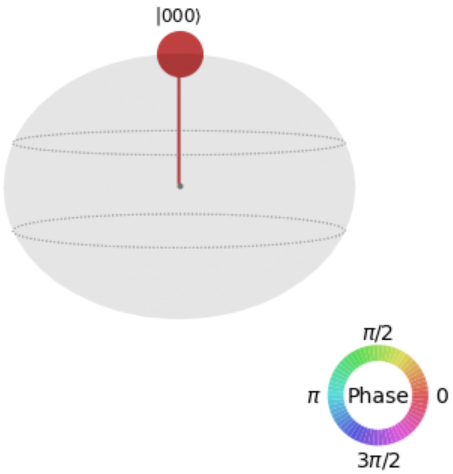


Let"s visualize the obtained state on the Q-Sphere using the methods: 1. Statevector.from_instruction that returns the output state of a quantum circuit.

2. plot_state_qsphere: This method returns a plot of the representation of the state on the qsphere.

(This part can be done with circuits that do not include measurements).

In [5]:
```
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_state_qsphere

state= Statevector.from_instruction(qc1)
plot_state_qsphere(state)
```



The color of the points here represents the phase while the size as will be more clear later represent the amplitude of the coreesponding corresponding basis state (a_i).

Measurements

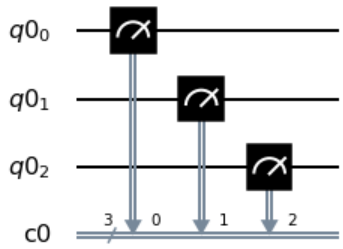
Now let's add the measurements part of the qubits and store the results in the classical bits. This can be done using:

In [6]:
```
qc1.measure(q,c)
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x7f404f36fa10>
```

Now together with the measurements part, the final quantum circuit will look like:

In [8]:
```
qc1.draw()
```



Getting Started with Qiskit- 02-Simulating the Quantum Circuit

Now that we have built our quantum circuit, we wish to simulate it and find out what the measurement results are. This can be done either classicaly, meaning simulating the quantum circuit on a regular classical computer, or on a real quantum computer. Here we'll focus on simulating the circuit classically and will leave executing the code on a real quantum computer for the second part of this series.

To simulate the circuit classically, we will see here how we can run it on the 'qasm-simulator' and on the 'statevector simulator' that can be imported from the Qiskit Aer package (there is also the 'unitary simulator' that we're not considering here).

The main difference between running the circuit on the 'qasm simulator' and on the 'statevector simulator' is that using the statevector simulator we compute the accurate final qubits quantum state without performing measurements (\finite number of measurements) at the end, and hence it does not take into account statistical noise. The 'qasm simulator' on the other hand, mimics an actual quantum computer as it performs a finite number of measurements at the end and therefore takes into account the statatistical noise accompanied with the readout of the results.

To see the difference and how each one works, let's try executing our first quantum circuit using both simulators:

Statevector Simulator

In [ ]:
```
#########      Statevector Simulator       ##########
#1 Gives the final state vector of the qubits (in the computational basis), see example below  (write the one we are e
xpecting all qubits in state 0)
```

In [9]:
```
#2 To use this simulator we first set the backend on which we are executing to be statevector backend,
#and then we execute the result of the execution.. This is done in this way:
from qiskit import Aer
from qiskit import execute
backend = Aer.get_backend('statevector_simulator')
#3 Next we execute the circuit, and save the (results that includes the data of the final result)
result=execute(qc1, backend).result()
```

If we print 'result' we obtain:

In [10]:
```
print(result)
```

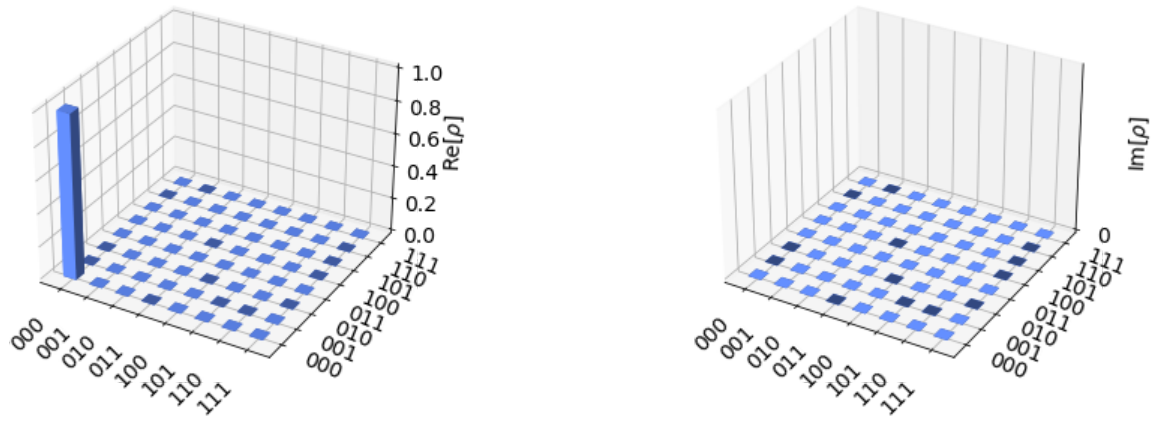
```
<qiskit.result.result.Result object at 0x7f404794b490>
```

So, to extract the statevector from the data, we use the method: get_statevector(first_qc)

In [11]:
```
final_state=result.get_statevector(qc1)
print('final_state:',final_state)
```

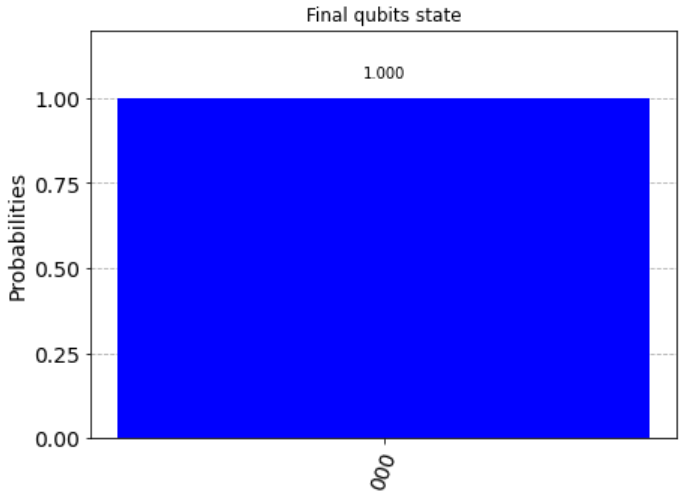
```
final_state: [1.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j]
```

In [12]:
```
#To print the density matrix of the obtained result:
from qiskit.visualization import plot_state_city
plot_state_city(final_state)
```



In [13]:
```
#or we can plot a histogram, and this is easily done using the method: result.get_counts(first_qc)
from qiskit.visualization import plot_histogram
final_state_counts=result.get_counts(qc1)
#final_state_counts includes dictionary: {'qubits_state0':number_of_counts_of-state0,'qubits_state1':probability to measure qubits_state1,...etc'}
print('final_state_counts:',final_state_counts)
plot_histogram(final_state_counts,color='blue', title="Final qubits state")
```

```
final_state_counts: {'000': 1}
```



Which is exactly what we expect to have, all the qubits are in state 0!

Note: As you may have noticed, we did not use the classical bits, and that is because this way does not include measurements.. We'll see how are they used when executing the circuit on the 'qasm simulator'.

Qasm Simulator

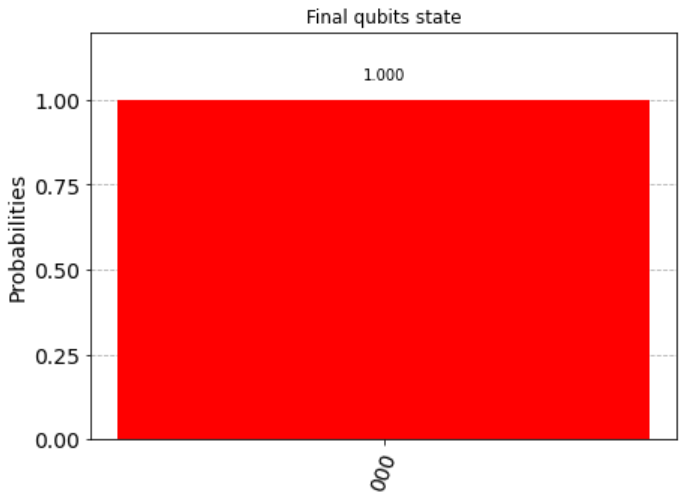
In [14]:
```
#########      Qasm Simulator       ##########
# As before, we first set the backend to Qasm-simulator
from qiskit import Aer
from qiskit import execute
backend = Aer.get_backend('qasm_simulator')
```

In [ ]:
```
# In this case, we'll need first to do the measurements, and store the results in classical bits...Why?
measure=qc1.measure(q,c)
```

In [15]:
```
# Now let's execute the circuit, obtain the result, and show the final result using the visualization methods we've se
en before
resultqasm = execute(qc1, backend, shots=1024).result()
# if you don't include # of shots, by default this variable will be set to 1024
```

In [16]:
```
#or we can plot a histogram, and this is easily done using the method: result.get_counts(first_qc)
from qiskit.visualization import plot_histogram
final_state_counts=resultqasm.get_counts(qc1)
#final_state_counts includes dictionary: {'qubits_state0':probability to measure qubits_state0,'qubits_state1':probabi
lity to measure qubits_state1,...etc'}
print('final_state_counts:',final_state_counts)
plot_histogram(final_state_counts,color='red', title="Final qubits state")
```

```
final_state_counts: {'000': 1024}
```



Real Device Simulator

To run the circuit on a real IBM quantum device, we need to first load our IBM_account. If we are working here, there is no need to copy it, we can just type

In [17]:
```
IBMQ.load_account()
```

```
/opt/conda/lib/python3.7/site-packages/qiskit/providers/ibmq/ibmqfactory.py:192: UserWarning: Timestamps in IBMQ back
end properties, jobs, and job results are all now in local time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
ibmqfactory.load_account:WARNING:2020-10-12 08:47:51,983: Credentials are already in use. The existing account in the
session will be replaced.


<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

However, if you are working on some other platform you'll need to specify your API token. You can find your API token here: https://quantum-computing.ibm.com/account (https://quantum-computing.ibm.com/account).

Now to make this faster, it is desired to pick the device which has as much less jobs. To check how busy the devices are, we can use

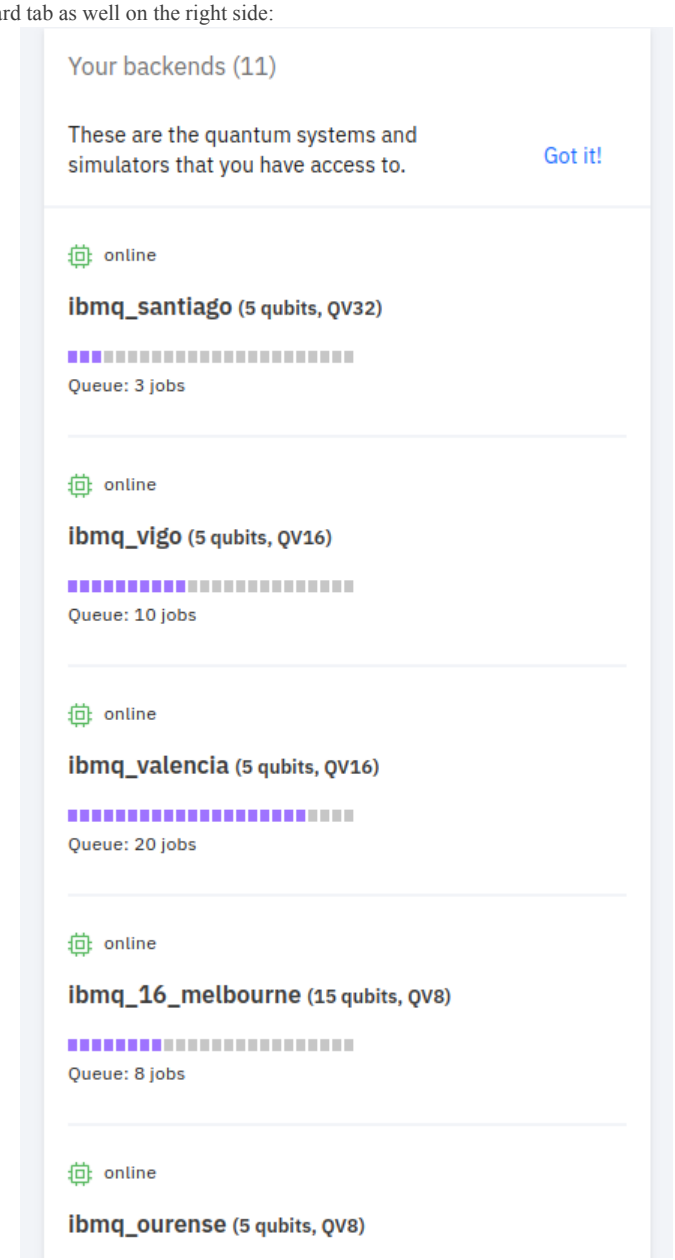
In [18]:
```
provider.backends()
```

```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_valencia') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_athens') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

In [19]:
```
from qiskit import IBMQ
provider = IBMQ.get_provider(hub='ibm-q-education', group='technion-Weinste', project='Quantum-Computin')
backend = provider.get_backend('ibmq_rome')
```

In [ ]:
```
from qiskit.providers.ibmq import least_busy
provider = IBMQ.load_account()
backend = least_busy(provider.backends(filters=lambda b: b.configuration().n_qubits >= 3 and
                                       not b.configuration().simulator and b.status().operational==True))
```

This list can is shown also in the Dashboard tab as well on the right side:



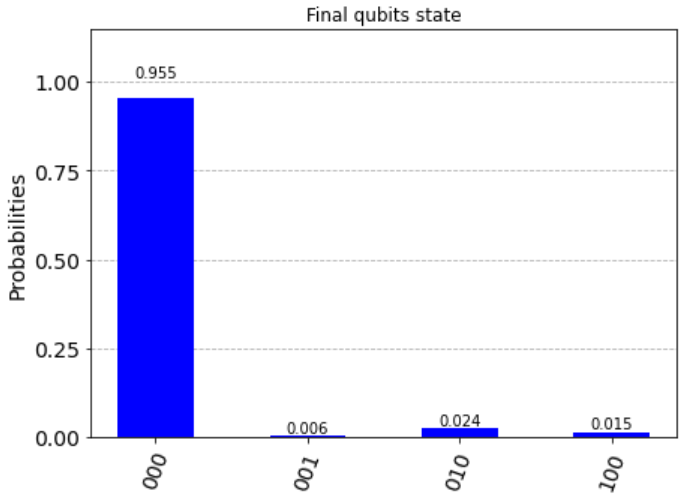
Now similar to the qasm simulator we do the following to run our circuit and obtaining the final state:

In [20]:
```
from qiskit.tools.monitor import job_monitor
job=execute(qc1, backend, shots=1024)
job_monitor(job)
final_state_counts = job.result().get_counts(qc1)
```

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

In [22]:
```
print('final_state_counts:',final_state_counts)
plot_histogram(final_state_counts,color='blue', title="Final qubits state")
```

```
final_state_counts: {'000': 978, '001': 6, '010': 25, '100': 15}
```



Getting Started with Qiskit- 03-Manipulating the qubits using Single-Qubit gates and Two-Qubit gates

Now that we have seen how we can create a circuit using qiskit, and execute it, let's try to do the circuit more intersting than the previous example! to do this, we need to manipulate the qubits first using quantum gates.. let's do this together..

Entangling two-qubit gates, such as CNOT gate and the single-qubit gates form the building blocks for most of the quantum algorithms. In this section, we will take the previous circuit, see how we can act on the qubits with various single-qubit gates and entangle them using CNOT gate, by the end, we will measure the final state, which is now not trivial as we had previously. Also, we can create our own unitaries!

Single- qubit gates- Pauli operators

Pauli Gates

Pauli X- gate:

The Pauli X-gate is the quantum analog to the classical Not gate $\rightarrow$ maps $|1\rangle$ to $|0\rangle$ and vise versa. For a state which is a general superposition of $|1\rangle$ to $|0\rangle$, X-gate is operation is equivalent to a rotation around the X-axis of the Bloch sphere by angle π. Matrix representation expressed in the computational basis

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

To obtain the state $|001\rangle$ in the previous example, we just need to act with Pauli-X on the first qubit:

In [ ]:
```
qc2=QuantumCircuit(3)
qc2.x(0)

qc2.draw()
```

Now let's check the final result

In [ ]:
```
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_state_qsphere

state= Statevector.from_instruction(qc2)
plot_state_qsphere(state)
```

Pauli Y and Z- gates:

Pauli Y-gate (Z gate) are rotate the quantum stae around the Y-axis (Z-axis) of the Bloch sphere by by angle π. So,

1. when Y-gate acts on $|0\rangle$ it maps it to $i|1\rangle$ and when it acts on $|1\rangle$ it maps it to $-i|0\rangle$. Matrix representation expressed in the computational basis
$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix},$$
2. when Z-gate acts on $|0\rangle$ it returns $|0\rangle$ and when it acts on $|1\rangle$ it returns $-|1\rangle$. Matrix representation expressed in the computational basis
$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

To act on the second qubit with z-gate and the third qubit with y gate we type:

In [ ]:
```
qc2.z(1)
qc2.y(2)

qc2.draw()
```

What is the final state now? .. let's check

In [ ]:
```
state= Statevector.from_instruction(qc2)
print(state)
```

How does these operations change the Bloch Sphere? Let's visualize the final result on the Bloch sphere or the qsphere.

In [ ]:
```
from qiskit.visualization import plot_bloch_multivector
plot_bloch_multivector(state)
```

In [ ]:
```
plot_state_qsphere(state)
```

So the final state is $i|101\rangle$

Homework #1:

1) Create a quantum circuit that maps the 3-qubit state vector $|\psi\rangle = \frac{1}{\sqrt{2}}(|010\rangle + |001\rangle)$ to the state $|\psi\rangle = \frac{1}{\sqrt{2}}(|111\rangle + |100\rangle)$ using only the quantum gates: Z,X,Y. Draw the obtained circuit using qiskit.

2) a. Draw the quantum circuit that performs the following operation on 2-qubit state:

$$X \otimes Y$$

.

on the state $|\psi\rangle = |10\rangle$.

b. Compute the final result of this operation both on a statevector simulator and on the qasm simulator (with #shots=1024).

Hadamard gate - H

When the Hadamard gate acts on $|0\rangle$ it maps it to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, and when it acts on $|1\rangle$ it maps it to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, where $\frac{|0\rangle\pm|1\rangle}{\sqrt{2}}$ are the eigenvectors $|+\rangle$ and $|-\rangle$ of Pauli-X, and hence it transforms one basis to another. Matrix representation expressed in the computational basis

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

Let's see how it acts on one qubit in the $|1\rangle$ state:

In [ ]:
```
qc3=QuantumCircuit(1)
qc3.x(0)
qc3.h(0)
qc3.draw()
```

In [ ]:
```
from qiskit.visualization import plot_bloch_multivector
final_state=Statevector.from_instruction(qc3)
plot_state_qsphere(final_state)
```

Exercise: What operation performs a "NOT" gate on the basis states $|+\rangle$ and $|-\rangle$. Write a code showing this.

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

#first: on |+>
qc_ex=QuantumCircuit(1)
qc_ex.h(0)
qc_ex.z(0)

display(qc_ex.draw())
stateplus= Statevector.from_instruction(qc_ex)
plot_state_qsphere(stateplus)
```

In [ ]:
```
#second: on |->
qc_ex=QuantumCircuit(1)
qc_ex.x(0)
qc_ex.h(0)
qc_ex.z(0)
display(qc_ex.draw())
stateminus= Statevector.from_instruction(qc_ex)
plot_state_qsphere(stateminus)
```

Exercise: How can we obtain X gate from Z and H? Implement X gate on qiskit using only Z and H gates and show that the obtained unitary of the circuit is equal to X. Use the backend 'unitary_simulator' which gives the unitary which is obtained by multiplying all the gates in the quantum circuit.

unitary = execute(qc,backend).result().get_unitary()

Answer: X=HZH, code:

In [ ]:
```
from qiskit import QuantumCircuit

qc=QuantumCircuit(1)
qc.h(0)
qc.z(0)
qc.h(0)

backend = Aer.get_backend('unitary_simulator')
unitary = execute(qc,backend).result().get_unitary()
print(unitary)
```

Single-qubit rotation gates:

Phase Shift gate R_ϕ

Phase shift gate is a parametrized gate that depends on one paramerer, ϕ, you need to specify. When R_ϕ acts on $|0\rangle$ it leaves it unchanged, while when it acts on $|1\rangle$ it maps it to $|1\rangle e^{i\phi}$.Matrix representation expressed in the computational basis

$$R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix},$$

In qiskit this gate can be implemented using: qc.p(ϕ, qubit_index).

Note: This gate is equivalent to R_z gate, which performs a rotation about the z -axis, up to some phase- $R_\phi(\phi) = e^{i\theta/2} R_z(\phi)$.

Special cases:

- $R_{\pi/2}$ is known as the S gate which is also called $\sqrt{Z}$ gate - In qiskit: eg. qc.s(qubit*index). -$R{\pi/4}isknownasthe$T$ gate- - In qiskit: eg. qc.t(qubit_index).
- R_π is the Pauli-Z gate- In qiskit: eg. qc.z(qubit_index).

Let's start from the state $|+\rangle$ and see the action of p for various $\phi's$ on the Bloch Sphere:

In [ ]:
```
import numpy as np
from qiskit.visualization import plot_bloch_multivector
phi=[0,np.pi/3,np.pi/2,0.66*np.pi,np.pi]
qc4=QuantumCircuit(1)
qc4.h(0)
qc4.p(phi[0],0)

display(qc4.draw())
backend = Aer.get_backend('statevector_simulator')
result=execute(qc4, backend).result()
final_state=result.get_statevector(qc4)
print('phase:',phi[0],'final_state:',final_state)
plot_bloch_multivector(final_state)
```

In [ ]:
```
qc4=QuantumCircuit(1)
qc4.h(0)
qc4.p(phi[1],0)

display(qc4.draw())

final_state=execute(qc4, backend).result().get_statevector(qc4)
print('phase:',phi[1],'final_state:',final_state)
plot_bloch_multivector(final_state)
```

In [ ]:
```
qc4=QuantumCircuit(1)
qc4.h(0)
qc4.p(phi[2],0)

display(qc4.draw())

final_state=execute(qc4, backend).result().get_statevector(qc4)
print('phase:',phi[2],'final_state:',final_state)
plot_bloch_multivector(final_state)
```

In [ ]:
```
qc4=QuantumCircuit(1)
qc4.h(0)
qc4.p(phi[4],0)

display(qc4.draw())


final_state=execute(qc4, backend).result().get_statevector(qc4)
print('phase:',phi[1],'final_state:',final_state)
plot_bloch_multivector(final_state)
```

What should we change in the code above to do a rotation about X axis starting from the state $|0\rangle$?

Answer: $h-p-h$ is equivalent to a rotation around the $x-axis$.

$U3, U2, U1, R_x, R_y, R_z$

The U3 gate Qiskit provides performs a generic rotation. This is a parametrized gate where you need to specify three angles, the Euler angles: θ, ϕ and λ, and its matrix representation is

$$U3(\theta,\phi,\lambda) = \begin{pmatrix} cos(\theta/2) & -e^{i\lambda}sin(\theta/2) \\ e^{i\phi}sin(\theta/2) & e^{i(\phi+\lambda)}cos(\theta/2) \end{pmatrix},$$

specific cases of $U3$:

- $U2$ gate- $U2 = U3(\pi/2, \phi, \lambda)$.
- $U1$ gate- same as the phase shift gate, $U1 = U3(0, 0, \lambda) = e^{i\lambda/2}R_z = R_\phi$. In qiskit R_z can be implemented by: qc.rz(qubit_index). -R_x gate -rotation about x axis- $R_x = U3(\theta, -\pi/2, \pi/2)$. In qiskit can be implemented also by: qc.rx(qubit_index) -R_y gate- rotation about y axis- $R_y = U3(\theta, 0, 0)$. In qiskit can be implemented also by: qc.ry(qubit_index)

$U3$ can be implemented by: qc.u3(θ, ϕ, λ, qubit_index).

Let's see a quick example: Rotation about the y-axis

In [ ]:
```
import numpy as np
qc5=QuantumCircuit(1)
qc5.u3(np.pi/3,0,0,0)
#qc4.ry(np.pi/3,0)

display(qc5.draw())

backend = Aer.get_backend('statevector_simulator')
result=execute(qc5, backend).result()
final_state=result.get_statevector(qc5)
plot_bloch_multivector(final_state)
```

Multiple- qubit gates

The controlled NOT (CNOT /CX) gate

The CNOT gate acts on two qubits and it flips the second qubit that we call the target qubit if the first qubit that we call the control qubit is in state $|1\rangle$. (Classical analog- XOR). More generally, it acts as follows

$$CNOT(a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle) = a|00\rangle + b|11\rangle + c|10\rangle + d|01\rangle.$$

Matrix representation:

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

Implementation in qiskit: qc.cx(control_qubit, target_qubit)

Let's see how we can entangle two qubits using CNOT, lets check what happens when CNOT acts on $|-1\rangle$:

In [ ]:
```
qc6=QuantumCircuit(2)
qc6.x(0)
qc6.x(1)
qc6.h(1)

display(qc6.draw())

backend = Aer.get_backend('statevector_simulator')
result=execute(qc6, backend).result()
initial_state=result.get_statevector(qc6)
print('initial_state:',final_state)
qc6.cx(0,1)
backend = Aer.get_backend('statevector_simulator')
result=execute(qc6, backend).result()
final_state=result.get_statevector(qc6)
print('final_state:',final_state)
```

Other useful controlled operations provided by qiskit:

- Controlled- Y gate: $CY = I \otimes |0\rangle\langle 0| + Y \otimes |1\rangle\langle 1|$. Implementation: qc.cy(control_qubit, target_qubit).
- Controlled- Z gate: $CZ = I \otimes |0\rangle\langle 0| + Y \otimes |1\rangle\langle 1|$. Implementation: qc.cz(control_qubit, target_qubit).
- Controlled- H gate: $CH = I \otimes |0\rangle\langle 0| + H \otimes |1\rangle\langle 1|$. Implementation: qc.ch(control_qubit, target_qubit).
- Controlled- $U3$ gate: $CU3 = I \otimes |0\rangle\langle 0| + U3 \otimes |1\rangle\langle 1|$. Implementation: qc.cu3(θ, ϕ, λ, control_qubit, target_qubit).
- Controlled- R_x gate: $CR_x = I \otimes |0\rangle\langle 0| + R_x \otimes |1\rangle\langle 1|$. Implementation: qc.crx(θ, control_qubit, target_qubit, rotation phase). Similarly, qc.cry(θ, control_qubit, target_qubit, rotation phase) performs a controlled rotaion about the y axis and qc.crz(θ, control_qubit, target_qubit, rotation phase) performs a controlled rotaion about the z axis.

Let's see how can we implement $CR_z|-1\rangle$:

In [ ]:
```
qc7=QuantumCircuit(2)
qc7.x(0) #remove for checking the case |- 0>
qc7.x(1)
qc7.h(1)

display(qc7.draw())

backend = Aer.get_backend('statevector_simulator')
result=execute(qc7, backend).result()
initial_state=result.get_statevector(qc7)
print('initial_state:',final_state)

qc7.crz(np.pi/2,0,1)
backend = Aer.get_backend('statevector_simulator')
result=execute(qc7, backend).result()
final_state=result.get_statevector(qc7)
print('final_state:',final_state)
plot_bloch_multivector(final_state)
```

Class Exercises

1-

a. Consider the following operation that acts on two qubits:

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

. What does this unitary do?

b. Show that:

$$(H \otimes I)U_A(\theta = \pi)(H \otimes I)|ab\rangle = |a \oplus b\rangle|b\rangle,$$

where $\oplus$ is addition modulo 2 and $a, b = 0, 1$. Which gate performs this operation $|ab\rangle - > |a \oplus b\rangle|b\rangle$?

c. Show this is true using qiskit.

d. Replace U_A by CX gate. How does this change the operation:

$$(H \otimes I)U_A(H \otimes I)|ab\rangle = ?.$$

Show the implementation of this operation using qiskit.

Answer:

a. This unitary is equivalent to controlled-R_ϕ that acts on two qubits where it acts with R_ϕ on the target qubit (second qubit) if the first qubit is 1 and does nothing if the first qubit is 0.

b. $U_A(\theta = \pi)$ is equal to CZ and the right hand side is what we obtain when acting with CNOT gate. Hence, we are showing here that $(H \otimes I)CZ(H \otimes I) = CNOT$.

$$\begin{aligned}(H \otimes I)CZ(H \otimes I)|ab\rangle &= \frac{1}{\sqrt{2}}(H \otimes I)CZ\left(|0\rangle + (-1)^a|1\rangle\right)|b\rangle \\ &= (H \otimes I)\left(|0\rangle + (-1)^{a+b}|1\rangle\right)|b\rangle = |a \oplus b\rangle|b\rangle\end{aligned}$$

Where in the last step we used that H is the inverse of H and that $H|a\rangle = |0\rangle + (-1)^a|1\rangle$.

c. Code using qiskit:

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

def circuitresult(initial_circuit):
    initial_circuit.h(1)
    initial_circuit.cz(0,1)
    initial_circuit.h(1)
    final_state=Statevector.from_instruction(initial_circuit)
    return final_state

qc1=QuantumCircuit(2)
qc1.x(0)
qc1.x(1)
print('result for initial state |11>:', circuitresult(qc1))

qc2=QuantumCircuit(2)
qc2.x(0)
print('result for initial state |01>:', circuitresult(qc2))

qc3=QuantumCircuit(2)
qc3.x(1)
print('result for initial state |10>:', circuitresult(qc3))

qc4=QuantumCircuit(2)
print('result for initial state |00>:', circuitresult(qc4))
```

d. For the case $U_A = CX$ gate:

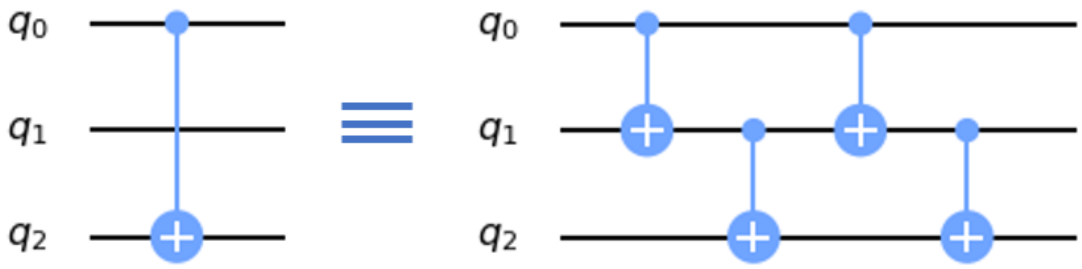
$$
\begin{aligned}
(H\otimes I)U_A(H\otimes I)|ab\rangle &= \frac{1}{\sqrt{2}}(H\otimes I)CX\left(|0\rangle+(-1)^a|1\rangle\right)|b\rangle \\
&= \begin{matrix}\frac{1}{\sqrt{2}}(H\otimes I)\left(|1\rangle+(-1)^a|0\rangle\right) & for\ |b\rangle=|1\rangle \\ \frac{1}{\sqrt{2}}(H\otimes I)\left(|0\rangle+(-1)^a|1\rangle\right) & for\ |b\rangle=|0\rangle\end{matrix} \\
&= \frac{1}{\sqrt{2}}(-1)^{ab}(H\otimes I)\left(|0\rangle+(-1)^b|1\rangle\right) = (-1)^{ab}|ab\rangle
\end{aligned}
$$

This is the same result obtained by acting with CZ. Implementation in qiskit:

In [ ]:
```
def circuitresult(initial_circuit):
    initial_circuit.h(1)
    initial_circuit.cx(0,1)
    initial_circuit.h(1)
    final_state=Statevector.from_instruction(initial_circuit)
    return final_state

qc1=QuantumCircuit(2)
qc1.x(0)
qc1.x(1)
print('result for initial state |11>:', circuitresult(qc1))

qc2=QuantumCircuit(2)
qc2.x(0)
print('result for initial state |01>:', circuitresult(qc2))

qc3=QuantumCircuit(2)
qc3.x(1)
print('result for initial state |10>:', circuitresult(qc3))

qc4=QuantumCircuit(2)
print('result for initial state |00>:', circuitresult(qc4))
```

2-

Show using the unitary simulator in qiskit that the following two circuits are equivalent:



Using this, we can distribute CNOT that acts on two qubits using a third qubit without acting on the original two qubits similtaneously.

In [ ]:
```
from qiskit import QuantumCircuit
backend = Aer.get_backend('unitary_simulator')

LHS_circuit=QuantumCircuit(3)
LHS_circuit.cx(0,2)
LHS_unitary = execute(LHS_circuit,backend).result().get_unitary()
RHS_circuit=QuantumCircuit(3)
RHS_circuit.cx(0,1)
RHS_circuit.cx(1,2)
RHS_circuit.cx(0,1)
RHS_circuit.cx(1,2)
RHS_unitary = execute(RHS_circuit,backend).result().get_unitary()
print('RHS_unitary:',RHS_unitary,'\n')
print('LHS_unitary:',LHS_unitary,'\n')
print('difference:',RHS_unitary-LHS_unitary)
```

Swap gate

The swap gate swaps two qubits. Matrix representation (in the computational basis):

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

Implementation in qiskit: qc.swap(first_qubit, second_qubit)

For example, when it acts on $|0+\rangle$:

In [ ]:
```
qc8=QuantumCircuit(2)
qc8.h(0)
qc8.swap(0,1)

display(qc8.draw())

backend = Aer.get_backend('statevector_simulator')
result=execute(qc8, backend).result()
final_state=result.get_statevector(qc8)
print('final_state:',final_state)
```

Which is the state $|+0\rangle$.

Creating an arbitrary gate

The gates available in qiskit are only a small part of quantum gates one can apply in quantum circuits. For example, to create the following gate

$$A=\frac{1}{\sqrt{2}}(\sigma_x+\sigma_y)=\begin{pmatrix}0 & (1-i)/\sqrt(2)\\ (1+i)/\sqrt(2) & 0\end{pmatrix},$$

we can use the Operator method as follows:

In [ ]:
```
from qiskit.quantum_info.operators import Operator
from qiskit.extensions import RXGate
qc9=QuantumCircuit(1)
unitary=[[0,(1-1j)/np.sqrt(2)],[(1+1j)/np.sqrt(2),0]]
Op=Operator(unitary)
print(Op)
qc9.append(Op,[0])
qc9.draw('mpl')
```

Examples- Quantum Protocols: Superdense coding and Simon's Algorithm

1. Superdense coding

Both Dense coding and teleportation protocols make use of entangled states. In the theory lectures you've met the Bell states,the most widely used entangled states. The two-qubits Bell states are given by:

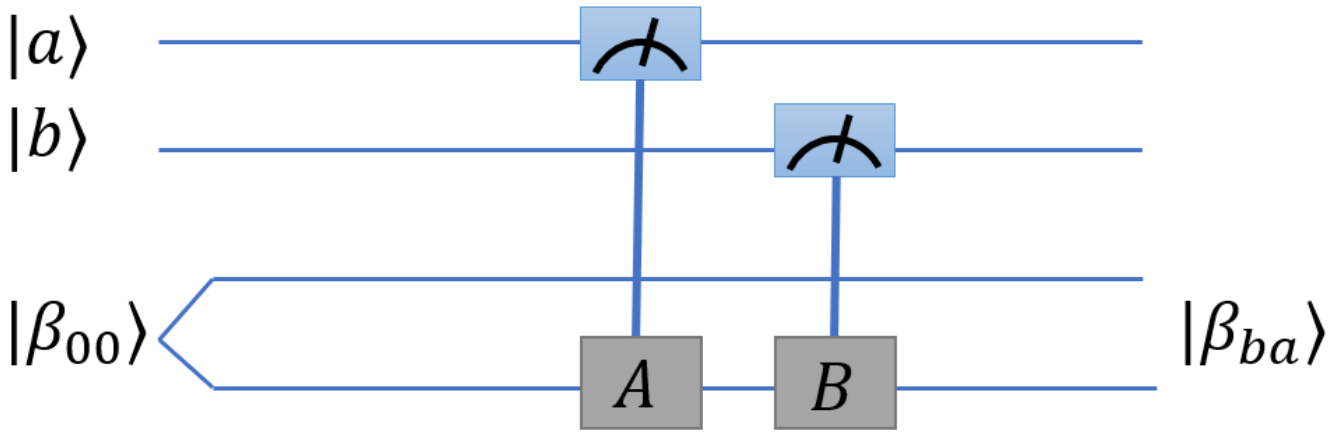
$$|\beta_{ba}\rangle=\frac{1}{\sqrt{2}}(|b0\rangle+(-1)^a|\bar{b}1\rangle.$$

How can we obtain such states from $|ba\rangle$ using a simple circuit? remember that $H|a\rangle=|0\rangle+(-1)^a|1\rangle$.

This looks very similar to the operation of the Hadamard on the second qubit. To obtain the Bell states, we just need to add a CX gate, where the target qubit is the second one. In qiskit we do the following:

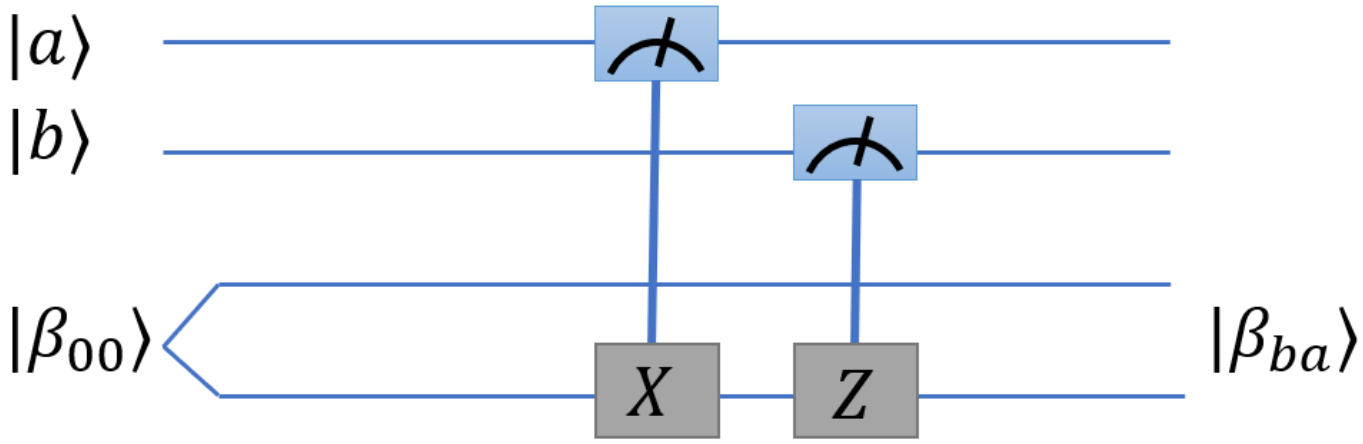
In [ ]:
```
###### Creating Bell_States #######
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_state_qsphere

def Bell_state(a,b):
    qc=QuantumCircuit(2)
    for i in range(a):## a can be either 0 or 1, if 0 this loop is not executed.
        qc.x(0)
    for i in range(b):
        qc.x(1)
    qc.h(0)
    qc.cx(0,1)
    return qc
#for_example:
qc1=Bell_state(0,1)
state= Statevector.from_instruction(qc)
plot_state_qsphere(state)
```

We are interested now to encode two classical bits in one of the Bell states. So, let's say we have two bits a,b then we wish to first create a qubit that encodes these two qubits: $|a\rangle$ and $|b\rangle$. Then, we want to create a circuit that transforms the compuational basis states $|ba\rangle$ into states in the Bell basis. This is done using what is called: "the dense coding protocol". Here, we have a circuit composed of four qubits, two qubits that encode the classical information, and two qubits of the Bell state that are initialized in the Bell state $|\beta_{00}\rangle$. To do this we perform the following circuit:



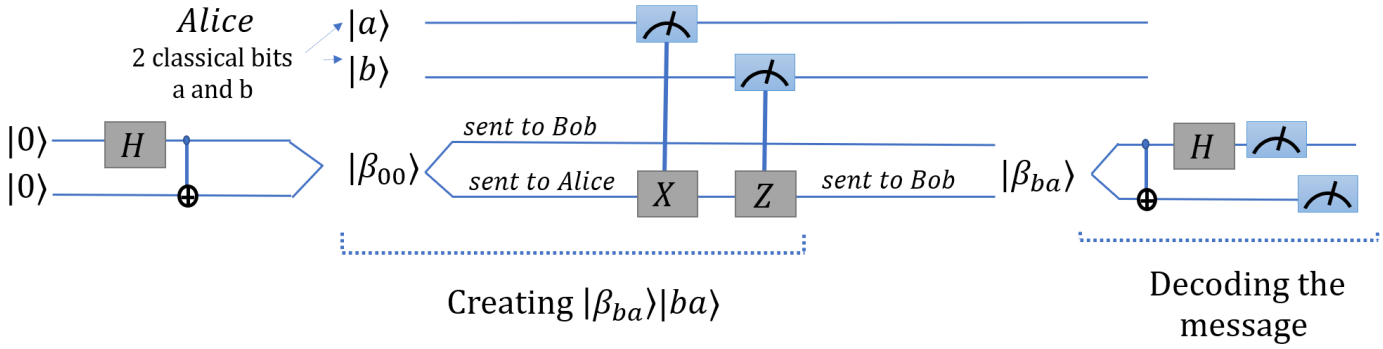
What should A and B be to get the transformation: $(|00\rangle + |11\rangle)|ba\rangle \rightarrow (|b0\rangle + (-1)^a)|\bar{b}1\rangle|ba\rangle\rangle$?

So, up to some phase, this can be performed using the above circuit when choosing: $A = X$ and $B = Z$. Hence, the dense coding circuit becomes:



Now, we have the following problem:

Alice has the two classical bits, and wants to send her classical to Bob using qubits. How can she do so using the dense protocol above? This can be done using the following circuit:



So this circuit performs the transformtion $|0\rangle|0\rangle|b\rangle|a\rangle \rightarrow |b\rangle|a\rangle|b\rangle|a\rangle$. In the first step, the Bell state $|\beta_{00}\rangle$ is created, one of the qubits is sent to Alice and the other one to Bob. In the next step, Alice performs the circuit we described above which changes the entangled state $|\beta_{00}\rangle$ to $|\beta_{ba}\rangle$ depending on the classical bits that Alice has, a and b, which are the message she wants to pass to Bob. Now, that the Bell state corresponding to Alice's bits is created, Alice sends the qubit she has to Bob so he can decode the message from the full state. Hence, Bob now has the state $|\beta_{ba}\rangle$, the original qubit he already had and this that Alice kept. Using these two qubits, Bob can extract the message encoded in the full Bell state. To do so, he simply maps the state into the original computational basis using the inverse circuit of this used for mapping the computational basis into Bell states.

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

def superdense(a,b):
    qc=QuantumCircuit(4,2)
    for i in range(a):## a can be either 0 or 1, if 0 this loop is not executed.
        qc.x(0)
    for i in range(b):
        qc.x(1)
    ####Prebaring |beta_0>
    qc.h(2)
    qc.cx(2,3)
    qc.barrier()

    ####encoding the bits in a bell state
    qc.cx(1,3)
    qc.cz(0,3)
    qc.barrier()

    ###decoding- mapping the bell state to the computational basis and measuring the result
    qc.cx(2,3)
    qc.h(2)
    qc.barrier()
    qc.measure(2,0)
    qc.measure(3,1)
    return qc
#for_example:
qc=superdense(1,0)
qc.draw()
```

In [ ]:
```
backend = Aer.get_backend('qasm_simulator')
job_sim = execute(qc, backend, shots=1024)
sim_result = job_sim.result()

measurement_result = sim_result.get_counts(qc)
print(measurement_result)
plot_histogram(measurement_result)
```