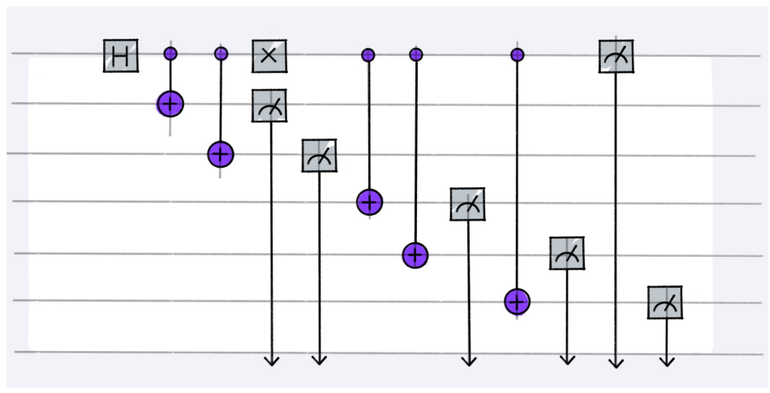
Primera Escuela de Computación Cuántica

Autores: Claudia Zendejas-Morales\ Institución: Qiskit IBM Quantum & UNAM\ Correo: clausia@ciencias.unam.mx

También puede consultar la última versión de este notebook en nuestro repositorio y los videos de clase en nuestro canal de Youtube.

Circuitos Cuánticos y sus Compuertas

Laboratorio Computacional



Outline

1. Introducción

1.A. Computación cuántica, ¿cómo la uso?

La computación cuántica es una tecnología de rápido crecimiento que aprovecha las leyes de la mecánica cuántica (como la superposición, la interferencia y el entrelazamiento) para resolver problemas demasiado complejos para las computadoras clásicas. Los dispositivos que realizan cálculos cuánticos se conocen como computadoras cuánticas.

De manera análoga a la computación clásica, podemos crear **algoritmos cuánticos** que se ejecutan en estas nuevas computadoras. Actualmente una manera popular de implementar estos algoritmos es con **circuitos cuánticos**, los cuales necesitan **compuertas cuánticas** para definirlos.

Veamos cómo crear estos circuitos cuánticos, cómo ejecutarlos en un simulador y en una computadora cuántica real.

1.B. ¿En dónde estamos actualmente?

Las computadoras cuánticas que tenemos hoy en día son demasiado pequeñas e inestables para dar una ventaja sobre las computadoras tradicionales.

En un nivel muy simple, hay dos factores que limitan el tamaño de los problemas que pueden resolver las computadoras cuánticas de hoy en día.

El primero es la cantidad de datos que se pueden almacenar y por lo tanto trabajar con estos, que normalmente medimos en qubits. Un **qubit** es un *bit cuántico*. Si no tenemos suficientes qubits, simplemente no podemos almacenar ni operar problemas por encima de cierto tamaño.

El segundo es la tasa de error de nuestra computadora cuántica; crear computadoras cuánticas es un proceso delicado. Las computadoras cuánticas que tenemos ahora son ruidosas, lo que significa que a menudo se equivocan e introducen "*ruido*". El ruido es información inútil que es difícil de distinguir de la información útil.

Por ejemplo, la estimación para factorizar un número muy grande (por ejemplo de 617 dígitos) en menos de un día asume una cantidad de ~20 millones de qubits ruidosos. En el momento de escribir esto (verano 2022), IBM ha creado una computadora cuántica de 65 qubits y tiene como objetivo crear un sistema con más de 1000 qubits para 2023. Hay otros algoritmos que creemos que nos darán una ventaja cuántica mucho antes de este hito.

Actualmente, por ejemplo, existen empresas que están invirtiendo en algorimos cuánticos y computación cuántica con el fin de aprovechar esa ventaja.

Afortunadamente, cualquier interesado, puede ejecutar y crear algoritmos cuánticos propios en miras de aprender a usar esta nueva tecnología.

2. Qubits

Al igual que en las computadoras digitales estándar (computadoras clásicas) la unidad fundamental son los **bits**, en las computadoras cuánticas tenemos a los **qubits**, una extensión del bit a la mecánica cuántica.

Una definición más precisa de **qubit** sería:

> Un qubit es un sistema mecánico cuántico de dos estados (o dos niveles) que pertenece a un espacio de Hilbert bidimensional ($\mathbb{C}^2$).

Un espacio de Hilbert es un espacio vectorial que define un producto interior. El espacio vectorial ocupado para definir a los qubits utiliza el campo de los números complejos $\mathbb{C}$ y como deseamos que sea bidimensional, es que se expresa como $\mathbb{C}^2$, es decir, necesitamos 2 números complejos para describir a un **qubit**.

Recordatorio: Números complejos

Un número complejo $\mathbb{C}$ es un elemento de un sistema numérico que amplía los números reales con un elemento específico denotado con i, llamado **unidad imaginaria** y satisface la ecuación

$$i^2 = -1$$

todo número complejo se puede expresar en la forma

$$z = a + bi$$

donde a y b son números reales, teniendo así una *parte real* (a) y otra *parte imaginaria* (b).

Así $a, b \in \mathbb{R}$, mientras que $z \in \mathbb{C}$.

2.A. Representación de los qubits: Matemáticamente

Para representar a los qubits de manera matemática, se utiliza la **notación de Dirac**, también conoida como notación *bra-ket*.

El estado $|0\rangle$ se pronuncia "**ket 0**", y el estado $|1\rangle$ es "**ket 1**". Esto dos estados (o qubits) son ortogonales entre sí y forman una base $\{|0\rangle, |1\rangle\}$, llamada **base computacional**, debido a que abarcan el espacio vectorial lineal bidimensional (espacio de Hilbert, $\mathbb{C}^2$) del qubit.

La notación de Dirac es una manera corta de escribir **vectores**. Hemos dicho que un qubit es un estado de dos niveles, y para representar qué tanto tenemos de cada nivel, necesitamos un vector de dos dimensiones. La manera más general de escribir un estado cuántico (un quibit) es entonces:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

donde $\alpha, \beta \in \mathbb{C}$ y el ket $|\psi\rangle$ es la forma corta de escribir el estado de dos niveles (el vector).

En el caso de los elementos de la base computacional $\{|0\rangle, |1\rangle\}$, representan a los *vectores de estado* siguientes:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad\qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Entonces, un qubit, es una superposición de los estados base, es decir, un qubit puede ser expresado como la combinación lineal de $|0\rangle$ y $|1\rangle$:

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

donde α y β son amplitudes de probabilidad. Cuando medimos este qubit, en la base computacional, la probabiliadad de obtener el estado $|0\rangle$ es de $|\alpha|^2$ y la probabilidad de obtener el estado $|1\rangle$ es de $|\beta|^2$. Debido a que los cuadrados de los valores absolutos de las amplitudes equivalen a probabilidades, se deduce que α y β deben estar restringidos de acuerdo con el segundo axioma de la teoría de la probabilidad mediante la ecuación:

$$|\alpha|^2 + |\beta|^2 = 1$$

Nota: Dada esta restrición, notemos que la longitud de los vectores siempre será igual a 1.

Ejemplo:

$$|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle = \begin{pmatrix} \frac{\sqrt{3}}{2} \\ \\ \frac{1}{2} \end{pmatrix}$$

La probabilidad de observar el estado $|0\rangle$ es de $\left|\frac{\sqrt{3}}{2}\right|^2 = \frac{3}{4}$, que es 75%.

La probabilidad de observar el estado $|1\rangle$ es de $\left|\frac{1}{2}\right|^2 = \frac{1}{4}$, que es 25%.

La suma de las probabilidades para cada caso: $\frac{3}{4} + \frac{1}{4} = 1$, que es 100%.

Recordatorio: Base de un espacio vectorial

Un conjunto B de vectores en un espacio vectorial V se llama **base** si cada elemento de V puede escribirse de manera única como una combinación lineal finita de

elementos de B.

De manera equivalente, un conjunto B es una base si sus elementos son linealmente independientes y cada elemento de V es una combinación lineal de elementos de B. En otras palabras, una base es un conjunto generador linealmente independiente.

Otras bases para los qubits

La base más común es la previamente mencionada, la base computacional, definida por los vectores: $\{|0\rangle, |1\rangle\}$.

Pero podemos usar cualesquiera otros dos vectores para definir una base para los qubits, siempre y cuando, sean linealmente independientes, pertenezcan al espacio vectorial, en este caso el espacio de $\mathbb{C}^2$ (dos números complejos).

Sin embargo, hay bases que sirven para entablar una comunicación entre quienes hacemos comptación cuántica, y las bases con elementos ortogonales son preferidas.

Otra base común es $\{|+\rangle, |-\rangle\}$, donde la definición de estos vectores es,

$$|+\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \qquad\qquad |-\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}$$

Estos estados también pueden ser escritos en términos de la base computacional como,

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

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

Notemos que los coeficientes de ambas bases, son números reales, deben pertenecer a los complejos, pero en particular ahora tenemos que $z = a + 0i$, por lo que queda solo la parte real (siguen siendo números dentro del campo $\mathbb{C}$).

Entonces es fácil realizar una gráfica de estos cuatro vectores:

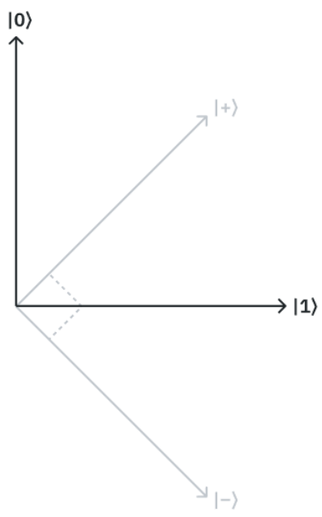


Imagen obtenida del Libro de Texto de Qiskit

2.B. Representación de los qubits: Esfera de Bloch

¿Qué pasa si las amplitudes de los qubits ya no son solo números reales?

Entonces serán números complejos, como se mencionó anteriormente, es decir $b \neq 0$, teniendo así tanto la parte real como la imaginaria: $z = a + bi$.

Debido a que un qubit se escribe,

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

con $\alpha, \beta \in \mathbb{C}$, es decir, necesita dos números complejos, y a su vez cada número complejo necesita de 2 números reales para poder escribirlo, entonces, necesitamos en total 4 números reales para representar a un qubit.

$$\alpha = a + bi$$
$$\beta = c + di$$

donde $a, b, c, d \in \mathbb{R}$.

Podría parecer a primera vista que se tienen 4 grados de libertad para un qubit, sin embargo, un grado de libertad deja de ser *libre* debido a la restricción de normalización: $|\alpha|^2 + |\beta|^2 = 1$. Tenemos ahora 3 grados de libertad.

Entonces, utilizando una *transformación* de coordenadas conveniente, se puede eliminar ese grado de libertad, es decir, lograr la conversión de $\mathbb{R}^4 \rightarrow \mathbb{R}^3$. Una elección posible son las coordenadas de Hopf, que manda el espacio de 4 dimensiones a la superficie de una esfera unitaria en 3 dimensiones.

Después de considerar que para un qubit, una *fase global* del estado no tiene consecuencias físicamente observables, se obtiene la **esfera de Bloch** (más detalles aquí).

> La esfera de Bloch es una representación geométrica de un sistema mecánico cuántico de dos niveles (un quibit).

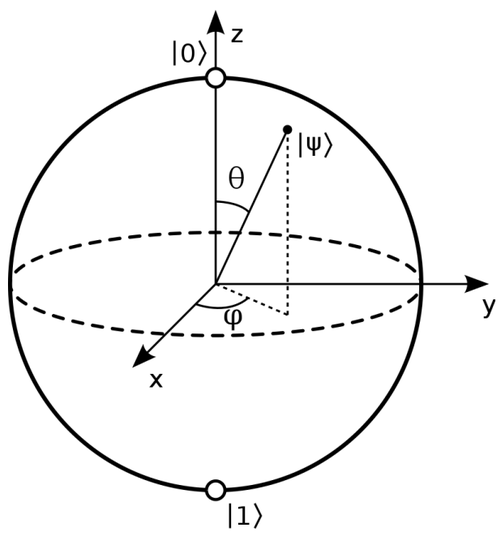


Imagen obtenida de Wikipedia

Entonces, el estado cuántico, en términos de la esfera de Bloch, se escribe como,

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

donde

$$0 \leq \theta \leq \pi$$
$$0 \leq \varphi < 2\pi$$

Entonces, la transformación es,

$$\alpha = \cos\left(\frac{\theta}{2}\right)$$
$$\beta = e^{i\varphi}\sin\left(\frac{\theta}{2}\right)$$

Notemos que bajo esta transformación, $\alpha \in \mathbb{R}$ y $\beta \in \mathbb{C}$.

Nota: Los parámetros θ y φ, respectivamente pueden ser reinterpretados en coordenadas esféricas como la colatitud con respecto al eje z y la longitud con respecto al eje x. Es decir, $(x, y, z) = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$ con $r = 1$.

En la esfera de Bloch, los puntos antipodales corresponden a un par de vectores de estado mutuamente ortogonales. El *polo norte* y el *polo sur* de la esfera de Bloch generalmente se eligen para que correspondan a los vectores base estándar $|0\rangle$ y $|1\rangle$ (canónicos), respectivamente, que a su vez podría corresponder, por ejemplo, a los estados de 'espín arriba' y 'espín abajo' de un electrón, a veces escritos como: $|\uparrow\rangle$ y $|\downarrow\rangle$. Sin embargo, esta **elección es arbitraria**.

Entonces, los estados $|0\rangle$ y $|1\rangle$ en los polos norte y sur, son **espacialmente antiparalelos** (lo que observamos en la esfera de Bloch), pero **en el espacio de Hilbert son ortogonales**.

En efecto, es un poco confuso tener dos nociones diferentes de "ortogonal", una para el espacio físico y otra para el espacio de Hilbert, pero esto viene del hecho de tener dos espacios diferentes en los que debemos pensar.

Otras bases en la esfera Bloch

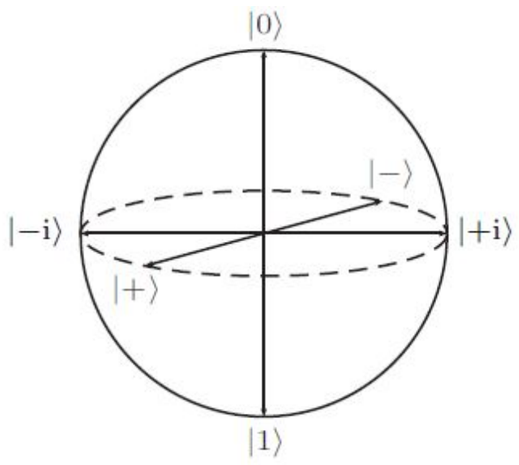


Imagen obtenida de Wikipedia

La base formada por $\{|0\rangle, |1\rangle\}$ son puntos antipodales sobre el eje z. La base formada por $\{|+\rangle, |-\rangle\}$ son puntos antipodales sobre el eje x. Entonces podemos esperar que exista una base que esté situada en el eje y, y así es, dicha base está formada por los estados $\{|+i\rangle, |-i\rangle\}$, que en términos de la base computacional se escriben,

$$|+i\rangle = \frac{1}{\sqrt{2}}\Big(|0\rangle + i|1\rangle\Big) = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{pmatrix}$$
$$|-i\rangle = \frac{1}{\sqrt{2}}\Big(|0\rangle - i|1\rangle\Big) = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{i}{\sqrt{2}} \end{pmatrix}$$

Nota: Estas tres bases son las más comunes, pero no son las únicas, mientras un par de estados, o vectores, sirvan para generar el espacio, pueden formar una base es decir, que no sean linealmente dependientes en el espacio de Hilbert.

Ejemplo: Veamos que los elementos de la base $\{|+i\rangle, |-i\rangle\}$ efectivamente son ortogonales.

Dos vectores son ortogonales cuando su producto interno es igual a cero, en notación de Dirac, se escribe como un **braket**, es decir: $\langle\phi|\psi\rangle$, en donde el bra $\langle\phi|$ indica el complejo conjugado transpuesto del vector $|\phi\rangle$, entonces tenemos que

$$\langle +i|-i\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{pmatrix}^* \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} \end{pmatrix} = \left(\frac{1}{\sqrt{2}}\right)\left(\frac{1}{\sqrt{2}}\right) + \left(\frac{-i}{\sqrt{2}}\right)\left(\frac{-i}{\sqrt{2}}\right) = \frac{1}{2} - \frac{1}{2} = 0$$

2.C. Representación de los qubits: Qiskit

En Qiskit podemos crear qubits con la clase `QuantumRegister` como sigue:

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

qubit1 = QuantumRegister(1) # el parámetro indica cuántos qubits queremos

qubit2 = QuantumRegister(1, 'qreg') # se puede indicar un nombre al registro cuántico (parámetro opcional)
```

In [2]:
```
print(qubit1) # al no especificar el nombre, le asigna uno con una numeración consecutiva
print(qubit2)
```

```
QuantumRegister(1, 'q0')
QuantumRegister(1, 'qreg')
```

La base que usa **Qiskit**, es la **base computacional**: $\{|0\rangle, |1\rangle\}$.

3. Circuito Cuántico

Debemos tener en dónde usar o colocar los qubits, para eso tenemos al **circuito cuántico**.

> Un circuito cuántico es un modelo de computación cuántica, similar a los circuitos clásicos, en los que una computación es una secuencia de compuertas cuánticas, mediciones, inicializaciones de qubits a valores conocidos y posiblemente otras acciones.

El conjunto mínimo de acciones que un circuito debe poder realizar en los qubits para permitir la computación cuántica se conoce como criterio de DiVincenzo.

Este tipo de circuitos se escriben de manera en que el eje horizontal representa al tiempo, comenzando en el lado izquierdo y terminando en el derecho. Las

líneas horizontales son qubits, las líneas dobles representan bits clásicos. Los elementos que estan conectados por estas líneas son operaciones realizadas en los qubits, como mediciones o compuertas.

Veamos cómo incluir en un circuito uno de los qubits que creamos anteriormente:

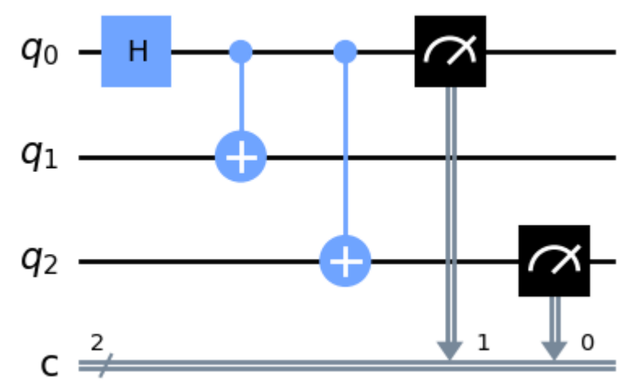
In [3]:
```
from qiskit import QuantumCircuit

circuit = QuantumCircuit(qubit1) # crear un circuito cuántico con un qubit

circuit.draw('mpl') # mostramos la representación gráfica del circuito
```

Out[3]:

En el siguiente ejemplo se define un circuito cuántico con 3 qubits, los cuales no tienen que ser creados necesariamente usando la clase `QuantumRegister`, sino que basta con indicar la cantidad de qubits que deseamos tenga el circuito, también se definen 2 bits clásicos (líneas dobles), y se aplican 3 compuertas. Veremos que efecto tiene cada una más adelante.

In [4]:
```
circ = QuantumCircuit(3, 2) # circuito con 3 qubits y 2 bits clásicos

circ.h(0)           # aplicar compuerta H al qubit 0
circ.cx(0, 1)       # aplicar compuerta CNOT a los qubits 0 y 1
circ.cx(0, 2)       # aplicar compuerta CNOT a los qubits 0 y 2
circ.measure(0, 1)  # medir el qubit 0 en el bit clásico 1
circ.measure(2, 0)  # medir el qubit 2 en el bit clásico 0

circ.draw('mpl')    # mostrar el circuito
```

Out[4]:



Como se mencionó, el tiempo ocurre de izquierda a derecha, entonces la compuerta H es la que se aplica primero, y después la compuerta $CNOT_{0,1}$ y por último la compuerta $CNOT_{0,2}$, al final ocurren las mediciones.

Un importante punto a resaltar, es que *todos los qubits comienzan en el estado* $|0\rangle$.

3.A. Orden de los qubits

En el circuito anterior se tienen 3 qubits: q_0, q_1 y q_2. Dado el subíndice podemos saber cual de ellos es el primero, el segundo y el tercero. Pero al momento de hacer la relación con las expresiones matemáticas, debemos tomar en cuenta que, un estado de tres qubits tendrá el siguiente orden (respecto a la convención de Qiskit),

$$|q_2\, q_1\, q_0\rangle = |000\rangle$$

Más detalle más adelante.

4. Compuerta Cuántica

Una vez que tenemos qubits en un circuito cuántico, lo siguiente es manipularlos, para lograr un objetivo, implementar algoritmos cuánticos. Esta manipulación de hace con las **compuertas cuánticas**.

> Las compuertas cuánticas son operadores unitarios y se describen como matrices unitarias en relación con alguna base. Usualmente se usa la base computacional.

Las compuertas cuánticas son los componentes básicos de los circuitos cuánticos, como lo son las compuertas lógicas clásicas para los circuitos digitales convencionales. A diferencia de muchas compuertas lógicas clásicas, las compuertas lógicas cuánticas son **reversibles**.

Recordatorio: Matriz Unitaria

En álgebra lineal, una matriz cuadrada compleja U es unitaria si su transpuesta conjugada U^* es también su inversa, es decir,

$$U^*U = UU^* = UU^{-1} = I$$

donde I es la matriz identidad.

En mecánica cuántica, la transpuesta conjugada se conoce como el adjunto hermitiano de una matriz y se denota con una daga (†), por lo que la ecuación anterior se escribe,

$$U^\dagger U = UU^\dagger = I$$

Las matrices unitarias tienen una importancia significativa en la mecánica cuántica porque conservan normas y, por lo tanto, amplitudes de probabilidad.

4.A. Representación de las compuertas: matemáticamente

Una compuerta cuántica que actúa sobre n qubits, esta representada por una matriz unitaria de $2^n \times 2^n$. Los vectores que representan estados cuánticos de dos niveles (qubits) sobre los que actúan las compuertas son vectores unitarios en 2^n dimensiones.

Las compuertas cuánticas más comunes operan en espacios vectoriales de uno o dos qubits, al igual que las compuertas lógicas clásicas comunes operan en uno o dos bits.

La acción de una compuerta en un estado cuántico específico se encuentra multiplicando el vector $|\psi_1\rangle$ que representa el estado, por la matriz U que representa la compuerta. El resultado es un nuevo estado cuántico $|\psi_2\rangle$:

$$U|\psi_1\rangle = |\psi_2\rangle$$

Cuando tenemos un qubit, $n = 1$, el vector que lo representa tiene dos elementos y una compuerta que opera sobre un qubit es una matriz de 2×2, obtener el estado resultante de aplicar la compuerta será como sigue,

$$U|\psi_1\rangle = \begin{pmatrix} u_{11} & u_{12} \end{pmatrix} \begin{pmatrix} \alpha \end{pmatrix} = \begin{pmatrix} \alpha\, u_{11} + \beta\, u_{12} \end{pmatrix} = \begin{pmatrix} \alpha' \end{pmatrix} = |\psi_2\rangle$$

$$U|\psi_1\rangle = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha\, u_{11} + \beta\, u_{12} \\ \alpha\, u_{21} + \beta\, u_{22} \end{pmatrix} = \begin{pmatrix} \beta' \\ \beta' \end{pmatrix} = |\psi_2\rangle$$

4.B. Compuertas de un sólo qubit

4.B.a. Compuertas de Pauli

Las **compuertas de Pauli** son las tres matrices de Pauli ($\sigma_x, \sigma_y, \sigma_z$) y actúan sobre un solo quibit.

Las compuertas X, Y, Z, respectivamente, equivalen a una rotación de π radianes (180°) al rededor del eje x, y, z en la *esfera de Bloch*.

Compuerta X

Esta compuerta es equivalente a la compuerta de negación (NOT) en las computadoras clásicas.

También es llamada *bit-flip* ya que mapea los estados base como sigue:

$$X|0\rangle = |1\rangle$$
$$X|1\rangle = |0\rangle$$

Su representación matricial es,

$$X = NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Ejemplo: Efecto de la compuerta X sobre el estado $|1\rangle$

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

En Qiskit aplicamos la compuerta X con el método `x()` :

In [5]:
```
circ1 = QuantumCircuit(1) # circuito con 1 qubit

circ1.x(0)                # aplicar compuerta X al (único) qubit 0

circ1.draw('mpl')         # mostrar el circuito
```

Out[5]:

Obtener el vector de estado con Qiskit

Con la clase `Statevector` de Qiskit, se puede obtener el estado de un circuito en el punto de se desee:

In [6]:
```
from qiskit.quantum_info import Statevector
from qiskit.visualization import array_to_latex


circ1 = QuantumCircuit(1)    # circuito con 1 qubit

psi_0 = Statevector(circ1)   # estado justo después de crear el circuito

circ1.x(0)                   # aplicar compuerta X al (único) qubit 0

psi_1 = Statevector(circ1)   # estado después aplicar la compuerta X

display(circ1.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0.data))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de X:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1.data))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))
```


```
👉 Estado inicial:
➤ en forma de vector:
```

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

```
➤ en forma de ket:
```

$$|0\rangle$$

```
👉 Estado después de X:
➤ en forma de vector:
```

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

```
➤ en forma de ket:
```

$$|1\rangle$$

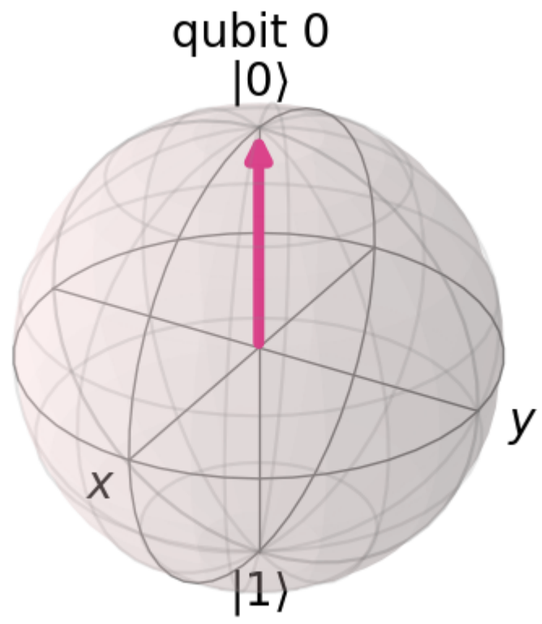
Visualización en la esfera de Bloch con Qiskit

Con la función `plot_bloch_multivector()` podemos visualizar un qubit, indicando el vector de estado.

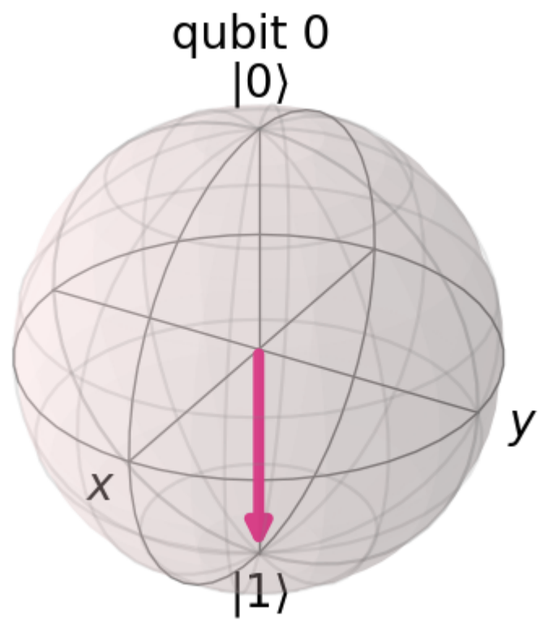
In [7]:
```
from qiskit.visualization import plot_bloch_multivector


print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de X:")
display(plot_bloch_multivector(psi_1))
```

```
display(plot_bloch_multivector(psi_1))
```

```
Estado inicial:
```




```
Estado después de X:
```



Compuerta Y

Esta compuerta mapea los estados base como sigue:

$$Y|0\rangle = i|1\rangle$$

$$Y|1\rangle = -i|0\rangle$$

Su representación matricial es,

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

Ejemplo: Efecto de la compuerta Y sobre el estado $|+\rangle$
Con notación de Dirac:

$$Y|+\rangle = Y\left(\frac{1}{\sqrt{2}}\Big(|0\rangle + |1\rangle\Big)\right) = \frac{1}{\sqrt{2}}\Big(Y|0\rangle + Y|1\rangle\Big) = \frac{1}{\sqrt{2}}\Big(i|1\rangle - i|0\rangle\Big) = \frac{-i}{\sqrt{2}}\Big(|0\rangle - |1\rangle\Big) = -i|-\rangle$$

Con notación matricial:

$$Y|+\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{-i}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{pmatrix} = -i\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix} = -i|-\rangle$$

En Qiskit aplicamos la compuerta Y con el método `y()` :

In [8]:
```
circ2 = QuantumCircuit(1)    # circuito con 1 qubit

psi_0 = Statevector(circ2)   # estado justo después de crear el circuito

circ2.y(0)                   # aplicar compuerta Y al (único) qubit 0

psi_1 = Statevector(circ2)   # estado después aplicar la compuerta Y

display(circ2.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de Y:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))
```


```
👉 Estado inicial:
➤ en forma de vector:
```

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

```
➤ en forma de ket:
```

$$|0\rangle$$

```
👉 Estado después de Y:
➤ en forma de vector:
```

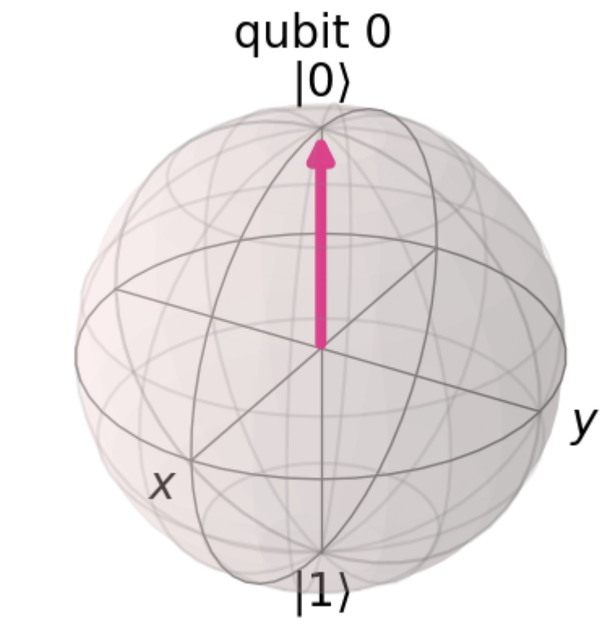
$$\begin{bmatrix} 0 & i \end{bmatrix}$$

➤ en forma de ket:

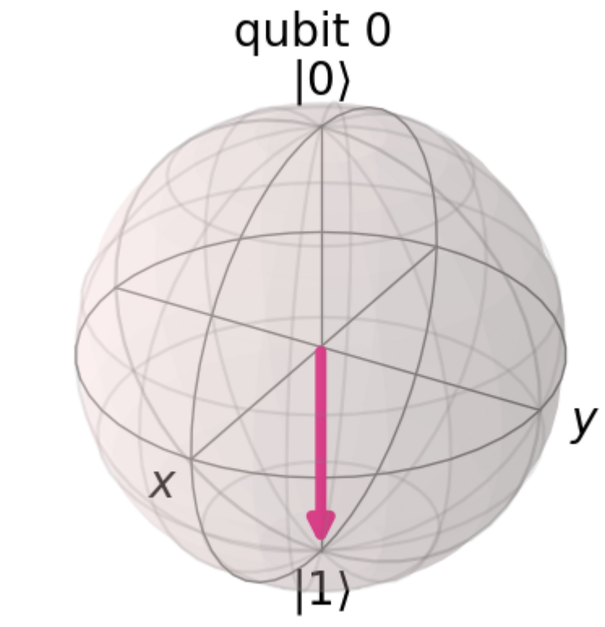
$$i|1\rangle$$

In [9]:
```
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de Y:")
display(plot_bloch_multivector(psi_1))
```

🔵 Estado inicial:



🔵 Estado después de Y:



¡Se ve igual que en el caso de la compuerta X!

¿Por qué?

No se pueden representar las fases globales en la esfera de Bloch.

Compuerta Z

Esta compuerta mapea los estados base como sigue:

$$Z|0\rangle = |0\rangle$$
$$Z|1\rangle = -|1\rangle$$

Su representación matricial es,

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

En Qiskit aplicamos la compuerta Z con el método `z()` :

In [10]:
```
circ3 = QuantumCircuit(1)    # circuito con 1 qubit

circ3.x(0)                   # obtener el estado |1>

psi_0 = Statevector(circ3)   # estado justo después de crear el circuito e iniciar el qubit en |1>

circ3.z(0)                   # aplicar compuerta Z al (único) qubit en la posición 0

psi_1 = Statevector(circ3)   # estado después aplicar la compuerta Z

display(circ3.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de Z:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de Z:")
display(plot_bloch_multivector(psi_1))
```

👉 Estado inicial:
➤ en forma de vector:

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

➤ en forma de ket:

$$|1\rangle$$

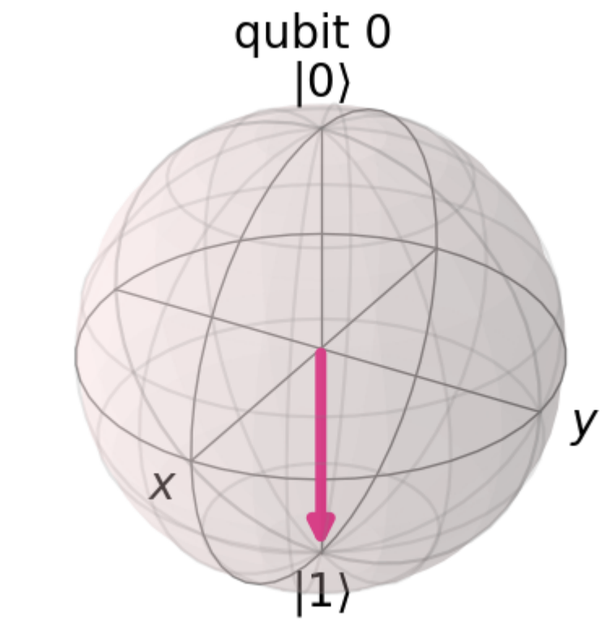
👉 Estado después de Z:
➤ en forma de vector:

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

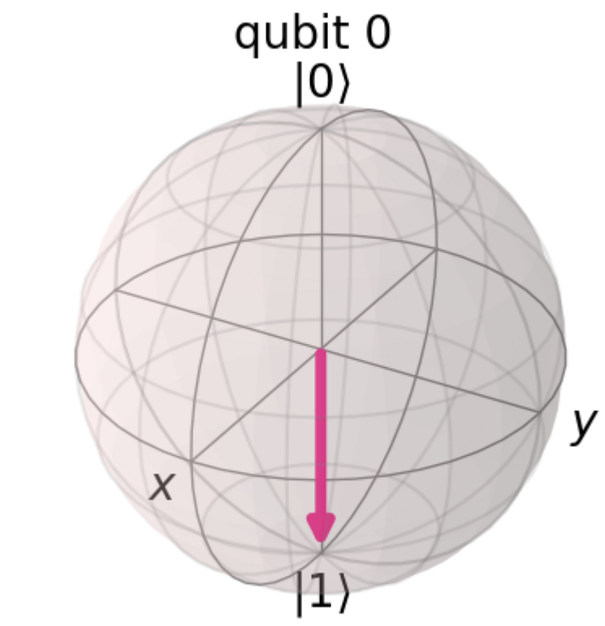
➤ en forma de ket:

$$-|1\rangle$$

🔵 Estado inicial:



🔵 Estado después de Z:



Matriz de transición

Si conocemos las trancisiones de las compuertas, podemos "armar" la matriz correspondiente:

Transiciones	Matriz
Compuerta X: $\|0\rangle \rightarrow \|1\rangle$ $\|1\rangle \rightarrow \|0\rangle$	$\begin{array}{c\|cc} \hookleftarrow & \|0\rangle & \|1\rangle \\ \hline \|0\rangle & 0 & 1 \\ \|1\rangle & 1 & 0 \end{array}$
Compuerta Y: $\|0\rangle \rightarrow i\|1\rangle$ $\|1\rangle \rightarrow -i\|0\rangle$	$\begin{array}{c\|cc} \hookleftarrow & \|0\rangle & \|1\rangle \\ \hline \|0\rangle & 0 & -i \\ \|1\rangle & i & 0 \end{array}$
Compuerta Z: $\|0\rangle \rightarrow \|0\rangle$ $\|1\rangle \rightarrow -\|1\rangle$	$\begin{array}{c\|cc} \hookleftarrow & \|0\rangle & \|1\rangle \\ \hline \|0\rangle & 1 & 0 \\ \|1\rangle & 0 & -1 \end{array}$

4.B.b. Rotaciones con otros ángulos

Las compuertas de Pauli son rotaciones de π radianes, pero podemos hacer que el estado cuántico rote con otros ángulos, para ello, tenemos las compuertas de rotaciones, alrededor de los ejes, definidas como sigue:

$$R_x(\phi) = \begin{pmatrix} \cos\left(\frac{\phi}{2}\right) & -i\sin\left(\frac{\phi}{2}\right) \\ -i\sin\left(\frac{\phi}{2}\right) & \cos\left(\frac{\phi}{2}\right) \end{pmatrix} = \cos\left(\frac{\phi}{2}\right) I - i\sin\left(\frac{\phi}{2}\right) X$$

$$R_y(\phi) = \begin{pmatrix} \cos\left(\frac{\phi}{2}\right) & -\sin\left(\frac{\phi}{2}\right) \\ \sin\left(\frac{\phi}{2}\right) & \cos\left(\frac{\phi}{2}\right) \end{pmatrix} = \cos\left(\frac{\phi}{2}\right) I - i\sin\left(\frac{\phi}{2}\right) Y$$

$$R_z(\phi) = \begin{pmatrix} e^{-i\frac{\phi}{2}} & 0 \\ 0 & e^{i\frac{\phi}{2}} \end{pmatrix} = \cos\left(\frac{\phi}{2}\right) I - i\sin\left(\frac{\phi}{2}\right) Z$$

En Qiskit existen los métodos `rx()`, `ry()` y `rz()` para lograr las rotaciones con diferentes ángulos.

En el siguiente código rotaremos el estado cuántico primero $\pi/2$ radianes al rededor dej eje y, lo que lo dejará sobre el eje $+x$, después lo rotaremos $\pi/4$ radianes alrededor del eje z y finalmente π radianes alrededor del eje x. En forma matemática, expresamos estos pasos como:

$$|\psi_3\rangle = R_x\left(\pi\right) R_z\left(\frac{\pi}{4}\right) R_y\left(\frac{\pi}{2}\right) |0\rangle$$

Notemos que el orden al multiplicar las matrices ocurre de derecha a izquierda, es decir, se multiplica primero la matriz que está más cerca del estado $|0\rangle$, después la que sigue a la izquierda y por último la que aparece al principio en la lectura de la expresión.

Los pasos son entonces:

$$|\psi_0\rangle = |0\rangle \qquad = |0\rangle$$

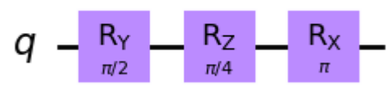
$$|\psi_1\rangle = R_y\left(\tfrac{\pi}{2}\right)|\psi_0\rangle \qquad = R_y\left(\tfrac{\pi}{2}\right)|0\rangle$$

$$|\psi_2\rangle = R_z\left(\tfrac{\pi}{4}\right)|\psi_1\rangle \qquad = R_z\left(\tfrac{\pi}{4}\right)R_y\left(\tfrac{\pi}{2}\right)|0\rangle$$

$$|\psi_3\rangle = R_x\left(\pi\right)|\psi_2\rangle \qquad = R_x\left(\pi\right)R_z\left(\tfrac{\pi}{4}\right)R_y\left(\tfrac{\pi}{2}\right)|0\rangle$$

In [11]:

```
import numpy as np

circ4 = QuantumCircuit(1)    # circuito con 1 qubit
psi_0 = Statevector(circ4)   # estado justo después de crear el circuito
circ4.ry(np.pi/2, 0)         # rotar pi/2 alrededor del eje y
psi_1 = Statevector(circ4)   # estado después de la primera rotación
circ4.rz(np.pi/4, 0)         # rotar pi/4 alrededor del eje z
psi_2 = Statevector(circ4)   # estado después de la segunda rotación
circ4.rx(np.pi, 0)           # rotar pi alrededor del eje x
psi_3 = Statevector(circ4)   # estado después de la tercera rotación
display(circ4.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después rotación de pi/2 alrededor del eje y:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))
print()
print("👉 Estado después rotación de pi/4 alrededor del eje z:")
print("➤ en forma de vector:")
display(array_to_latex(psi_2))
print("➤ en forma de ket:")
display(psi_2.draw('latex'))
print()
print("👉 Estado después rotación de pi alrededor del eje x:")
print("➤ en forma de vector:")
display(array_to_latex(psi_3))
print("➤ en forma de ket:")
display(psi_3.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después rotación de pi/2 alrededor del eje y:")
display(plot_bloch_multivector(psi_1))

print("🔵 Estado después rotación de pi/4 alrededor del eje z:")
display(plot_bloch_multivector(psi_2))

print("🔵 Estado después rotación de pi alrededor del eje x:")
display(plot_bloch_multivector(psi_3))
```



👉 Estado inicial:
➤ en forma de vector:

$$[1 \quad 0]$$

➤ en forma de ket:

$$|0\rangle$$

👉 Estado después rotación de pi/2 alrededor del eje y:
➤ en forma de vector:

$$\left[\tfrac{1}{\sqrt{2}} \quad \tfrac{1}{\sqrt{2}}\right]$$

➤ en forma de ket:

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

👉 Estado después rotación de pi/4 alrededor del eje z:
➤ en forma de vector:

$$[0.65328 - 0.2706i \quad 0.65328 + 0.2706i]$$

➤ en forma de ket:

$$(\sqrt{\frac{\sqrt{2}}{8} + \frac{1}{4}} - i\sqrt{\frac{1}{4} - \frac{\sqrt{2}}{8}})|0\rangle + (\sqrt{\frac{\sqrt{2}}{8} + \frac{1}{4}} + i\sqrt{\frac{1}{4} - \frac{\sqrt{2}}{8}})|1\rangle$$

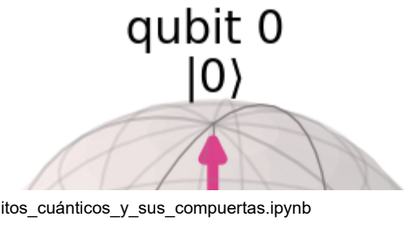
👉 Estado después rotación de pi alrededor del eje x:
➤ en forma de vector:

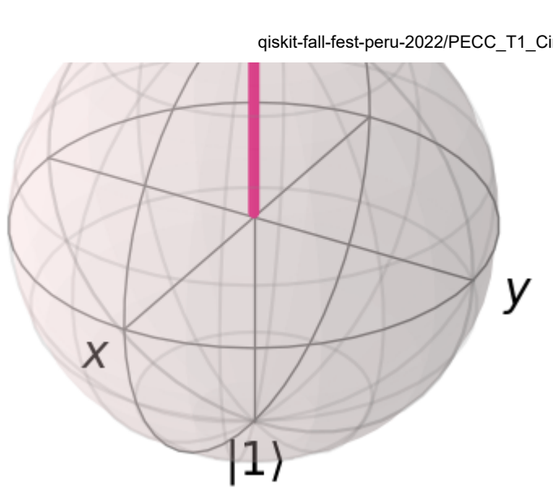
$$[0.2706 - 0.65328i \quad -0.2706 - 0.65328i]$$

➤ en forma de ket:

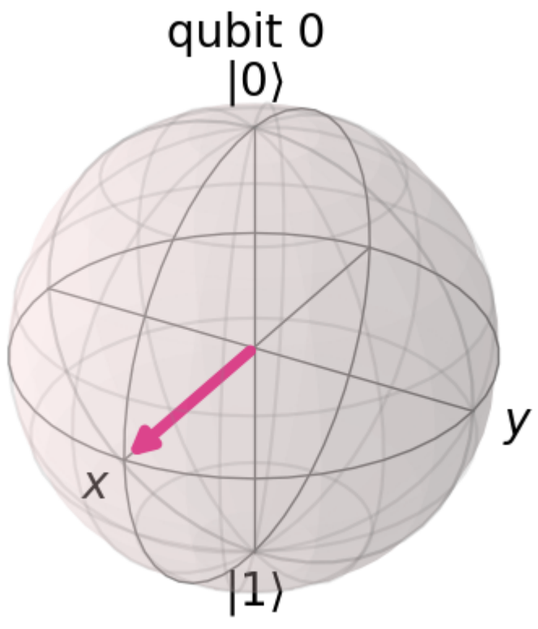
$$(\sqrt{\frac{1}{4} - \frac{\sqrt{2}}{8}} - i\sqrt{\frac{\sqrt{2}}{8} + \frac{1}{4}})|0\rangle + (-\sqrt{\frac{1}{4} - \frac{\sqrt{2}}{8}} - i\sqrt{\frac{\sqrt{2}}{8} + \frac{1}{4}})|1\rangle$$

🔵 Estado inicial:

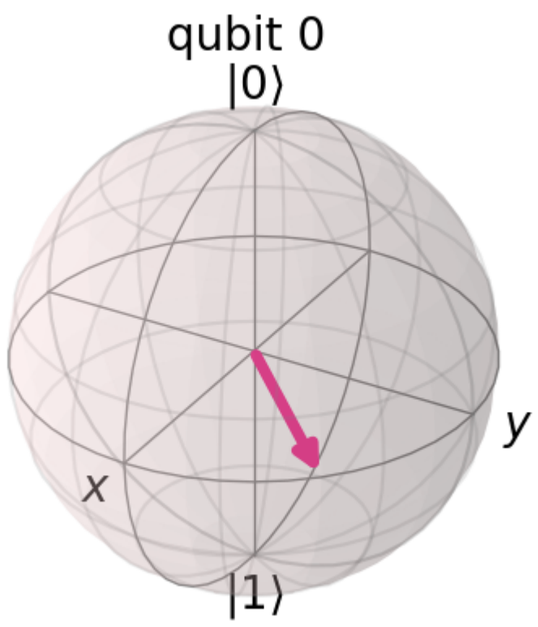




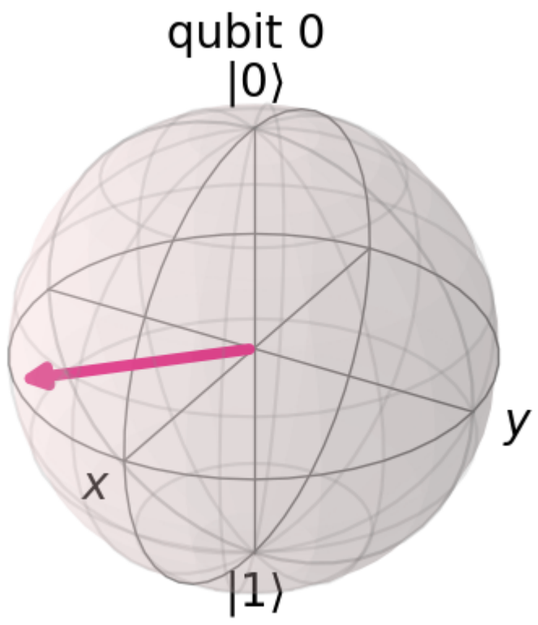
Estado después rotación de pi/2 alrededor del eje y:



Estado después rotación de pi/4 alrededor del eje z:



Estado después rotación de pi alrededor del eje x:



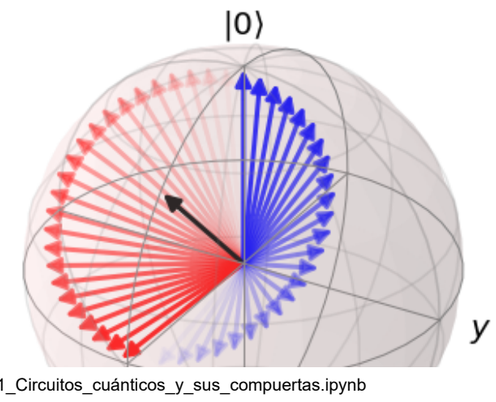
4.B.c. Compuerta Hadamard

La compuerta Hadamard crea un estado de superposición de la base computacional $\{|0\rangle, |1\rangle\}$, entonces mapea los estados como sigue,

$$H|0\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = |+\rangle$$

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

La compuerta Hadamard también es una rotación, pero alrededor del eje $\frac{(\hat{x}+\hat{z})}{\sqrt{2}}$, es decir, el eje en el plano xz con 45° del estado $|0\rangle$



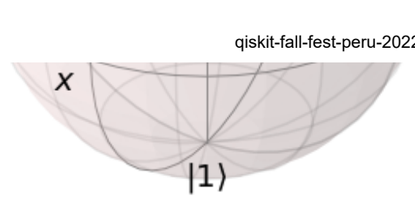


Imagen obtenida de Physics StackExchange

Su representación matricial es,

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

Ejemplo: Efecto de la compuerta H sobre el estado $|-\rangle$

Con notación de Dirac:

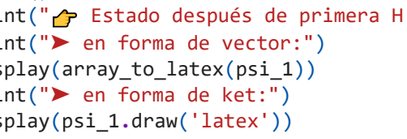
$$\begin{aligned} H|-\rangle &= H\left(\frac{1}{\sqrt{2}}\Big(|0\rangle - |1\rangle\Big)\right) = \frac{1}{\sqrt{2}}\Big(H|0\rangle - H|1\rangle\Big) \\ &= \frac{1}{\sqrt{2}}\left[\left(\frac{1}{\sqrt{2}}\Big(|0\rangle + |1\rangle\Big)\right) - \left(\frac{1}{\sqrt{2}}\Big(|0\rangle - |1\rangle\Big)\right)\right] \\ &= \frac{1}{2}\Big(|0\rangle + |1\rangle - |0\rangle + |1\rangle\Big) = \frac{1}{2}\Big(2|1\rangle\Big) = |1\rangle \end{aligned}$$

Con notación matricial:

$$H|-\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}\begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1-1 \\ 1+1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle$$

En Qiskit aplicamos la compuerta H con el método `h()` :

In [12]:
```
circ5 = QuantumCircuit(1)    # circuito con 1 qubit

psi_0 = Statevector(circ5)   # estado justo después de crear el circuito

circ5.h(0)                   # apliquemos H al estado |0>, quedamos en |+>

psi_1 = Statevector(circ5)   # estado justo después de aplicar H

circ5.z(0)                   # aplicar compuerta Z al (único) qubit 0, quedamos en el estado |->

psi_2 = Statevector(circ5)   # estado después aplicar la compuerta Z

circ5.h(0)                   # aplicar compuerta H de nuevo, quedamos en el estado |1>

psi_3 = Statevector(circ5)   # estado después aplicar la compuerta Z

display(circ5.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de primera H:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))
print()
print("👉 Estado después de Z:")
print("➤ en forma de vector:")
display(array_to_latex(psi_2))
print("➤ en forma de ket:")
display(psi_2.draw('latex'))
print()
print("👉 Estado después de segunda H:")
print("➤ en forma de vector:")
display(array_to_latex(psi_3))
print("➤ en forma de ket:")
display(psi_3.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de primera H:")
display(plot_bloch_multivector(psi_1))

print("🔵 Estado después de Z:")
display(plot_bloch_multivector(psi_2))

print("🔵 Estado después de segunda H:")
display(plot_bloch_multivector(psi_3))
```



👉 Estado inicial:
➤ en forma de vector:

$$[1 \quad 0]$$

➤ en forma de ket:

$$|0\rangle$$

👉 Estado después de primera H:
➤ en forma de vector:

$$\left[\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}\right]$$

➤ en forma de ket:

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

👉 Estado después de Z:
➤ en forma de vector:

$$\left[\frac{1}{\sqrt{2}} \quad -\frac{1}{\sqrt{2}}\right]$$

➤ en forma de ket:

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

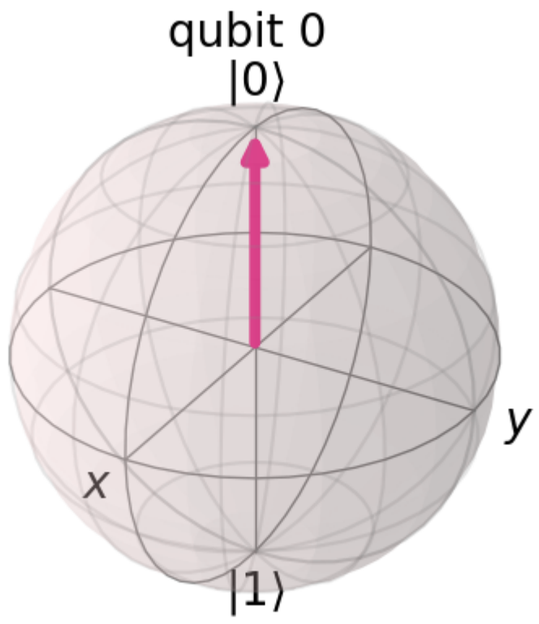
👉 Estado después de segunda H:
➤ en forma de vector:

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

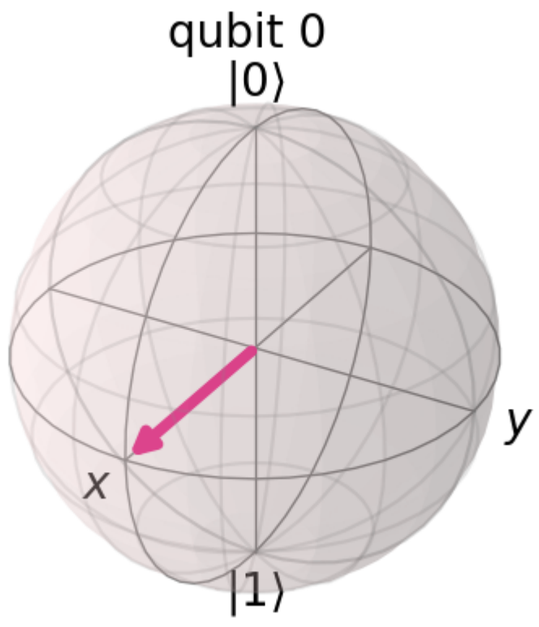
➤ en forma de ket:

$$|1\rangle$$

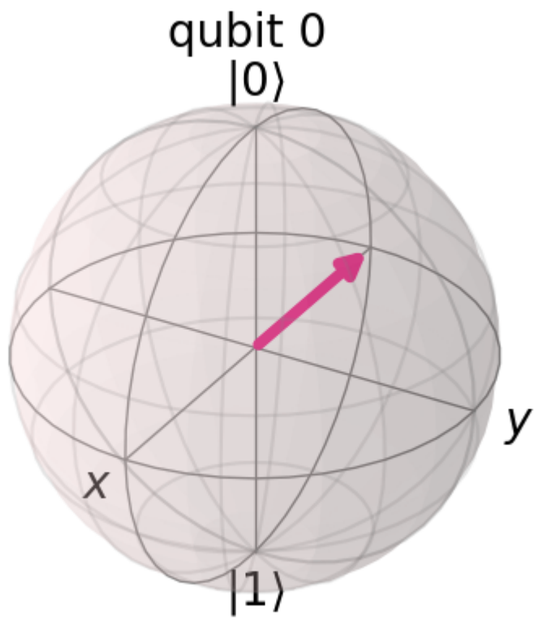
🔵 Estado inicial:



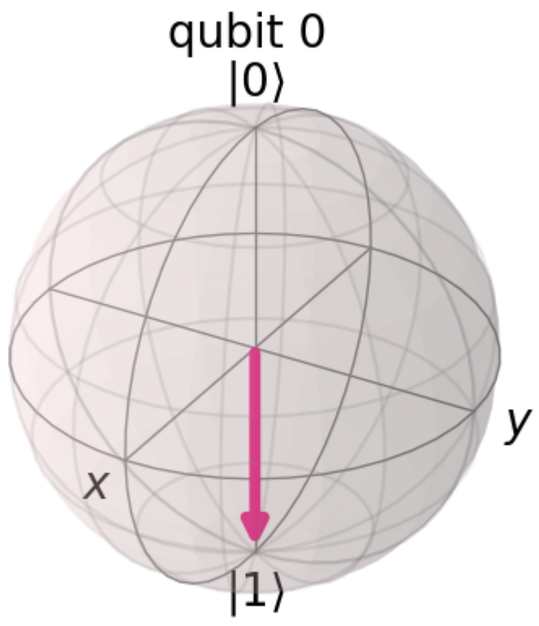
🔵 Estado después de primera H:



🔵 Estado después de Z:



🔵 Estado después de segunda H:



Notemos que

$$HZH = X$$

4.B.d. Compuertas de cambio de fase

Las compuertas de cambio de fase forman una familia de compuertas de un solo q-bit q, e mapean los estados base como sig. e

Las compuertas de cambio de fase forman una familia de compuertas de un solo qubit que mapean los estados base como sigue,

$$P(\phi)\,|0\rangle = |0\rangle$$

$$P(\phi)\,|1\rangle = e^{i\phi}|1\rangle$$

La probabilidad de medir $|0\rangle$ o $|1\rangle$ no cambia después de aplicar esta compuerta, sin embargo, modifica la fase del estado cuántico. Esto es equivalente a trazar un círculo horizontal (una línea de latitud), o una rotación a lo largo del eje z en la esfera de Bloch por ϕ radianes. La compuerta de cambio de fase está representada por la matriz:

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

donde ϕ es el cambio de fase con periodo 2π.

Notemos que la compuerta $P(\phi)$ no es hermitiana (excepto para todos los $\phi = n\pi,\ n \in \mathbb{Z}$).

Existen compuertas con su propio nombre para algunos casos particulres del valor de ϕ.

Compuertas S y $S^\dagger$

Es una rotación alrededor del eje z por $\pi/2$ radianes, entonces

$$S = \sqrt{Z} = P\left(\frac{\pi}{2}\right)$$

La compuerta $S^\dagger$ es el conjugado de la compuerta S, entonces,

$$S^\dagger = P\left(-\frac{\pi}{2}\right)$$

Sus matrices son,

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad\qquad S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$$

Compuertas T y $T^\dagger$

Es una rotación alrededor del eje z por $\pi/4$ radianes, entonces

$$T = \sqrt{S} = \sqrt[4]{Z} = P\left(\frac{\pi}{4}\right)$$

$$T^\dagger = P\left(-\frac{\pi}{4}\right)$$

Sus matrices son,

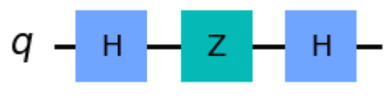
$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \qquad\qquad T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & e^{-i\frac{\pi}{4}} \end{pmatrix}$$

En Qiskit tenemos a nuestra disposición métodos para aplicar las compuertas de cambio de fase: `p()`, `s()`, `sdg()`, `t()`, `tdg()`.

Veamos el efecto de estas compuertas, pero debido a que representan rotaciones alrededor del eje z, entonces partiremos del estado $|+\rangle$, para que sean notorias la rotaciones.

In [13]:

```
circ6 = QuantumCircuit(1)    # circuito con 1 qubit

circ6.h(0)                   # nos movemos al estado |+>

psi_0 = Statevector(circ6)   # estado justo después de crear el circuito y posicionarse en |+>

circ6.s(0)                   # aplicar compuerta S, rotando pi/2

psi_1 = Statevector(circ6)   # estado justo después de aplicar P

circ6.tdg(0)                 # aplicar compuerta T† rotanfo -pi/4

psi_2 = Statevector(circ6)   # estado después aplicar la compuerta S

circ6.p(2*np.pi/3, 0)        # aplicar compuerta P, rotando 2pi/3

psi_3 = Statevector(circ6)   # estado después aplicar la compuerta T†

display(circ5.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial |+>:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de S:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))
print()
print("👉 Estado después de T†:")
print("➤ en forma de vector:")
display(array_to_latex(psi_2))
print("➤ en forma de ket:")
display(psi_2.draw('latex'))
print()
print("👉 Estado después de P(2*pi/3):")
print("➤ en forma de vector:")
display(array_to_latex(psi_3))
print("➤ en forma de ket:")
display(psi_3.draw('latex'))

print()
print("🔵 Estado inicial |+>:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de S:")
display(plot_bloch_multivector(psi_1))

print("🔵 Estado después de T†:")
display(plot_bloch_multivector(psi_2))

print("🔵 Estado después de P(2*pi/3):")
display(plot_bloch_multivector(psi_3))
```




```
👉 Estado inicial |+>:
➤ en forma de vector:
⎡ 1      1 ⎤
```

$$\left[\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}\right]$$

➤ en forma de ket:

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

👉 Estado después de S:
➤ en forma de vector:

$$\left[\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}}i\right]$$

➤ en forma de ket:

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

👉 Estado después de T†:
➤ en forma de vector:

$$\left[\frac{1}{\sqrt{2}} \quad \frac{1}{2}(1+i)\right]$$

➤ en forma de ket:

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

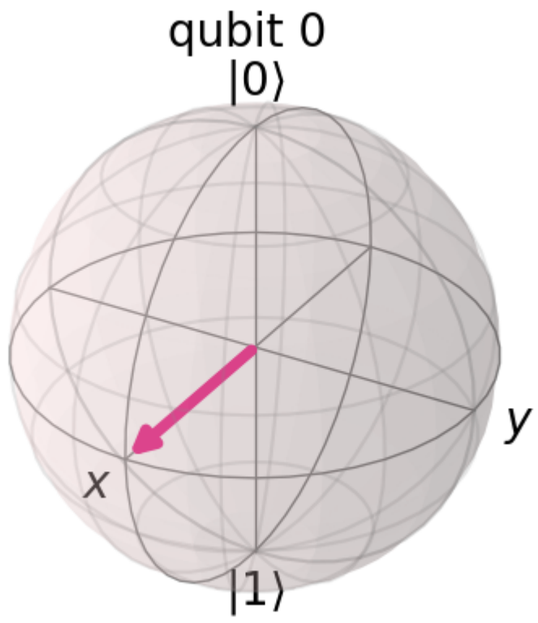
👉 Estado después de P(2*pi/3):
➤ en forma de vector:

$$\left[\frac{1}{\sqrt{2}} \quad -0.68301 + 0.18301i\right]$$

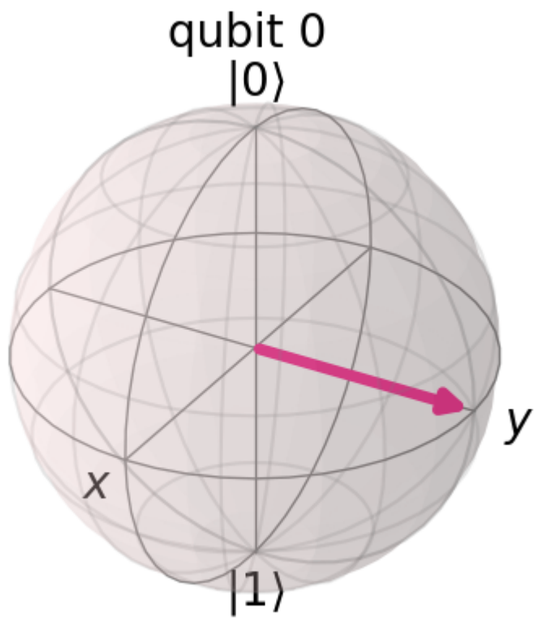
➤ en forma de ket:

$$\frac{\sqrt{2}}{2}|0\rangle + (-0.683012701892219 + 0.183012701892219i)|1\rangle$$

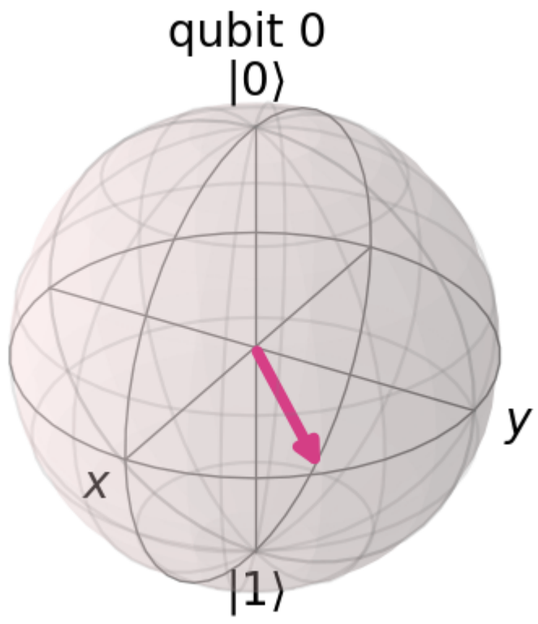
🔵 Estado inicial |+>:



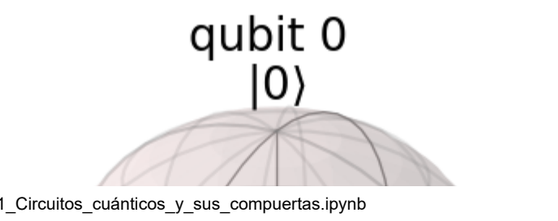
🔵 Estado después de S:

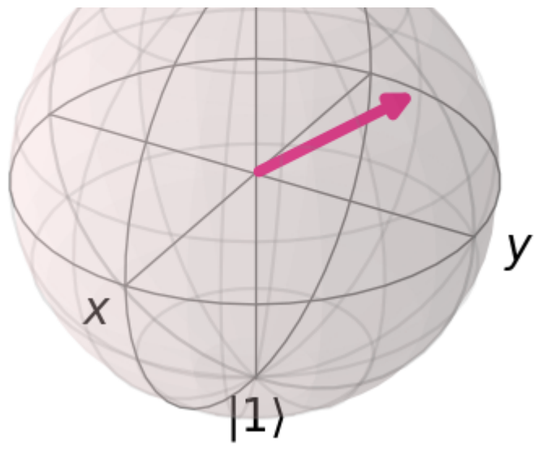


🔵 Estado después de T†:



🔵 Estado después de P(2*pi/3):





4.B.e. Compuerta de un qubit más general

La compuerta que generaliza todas las compuertas de un solo qubit es la conocida como $U3$ y está definida como:

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

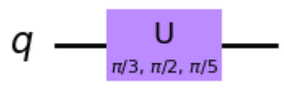
ejemplos de equivalencias:

$$U3\left(\theta, -\frac{\pi}{2}, \frac{\pi}{2}\right) = RX(\theta)$$

$$U3\left(\theta, 0, 0\right) = RY(\theta)$$

En Qiskit podemos usar el método `u()` para aplicar esta compuerta:

In [14]:
```
circ_u = QuantumCircuit(1)               # circuito con 1 qubit

psi_0 = Statevector(circ_u)              # estado justo después de crear el circuito e iniciar el qubit en |1>

circ_u.u(np.pi/3, np.pi/2, np.pi/5, 0)   # aplicar compuerta U3 al (único) qubit en la posición 0

psi_1 = Statevector(circ_u)              # estado después aplicar la compuerta U3

display(circ_u.draw('mpl'))              # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de U3(pi/3, pi/8, pi/5):")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de U3(pi/3, pi/8, pi/5):")
display(plot_bloch_multivector(psi_1))
```



👉 Estado inicial:
➤ en forma de vector:

$$[1 \quad 0]$$

➤ en forma de ket:

$$|0\rangle$$

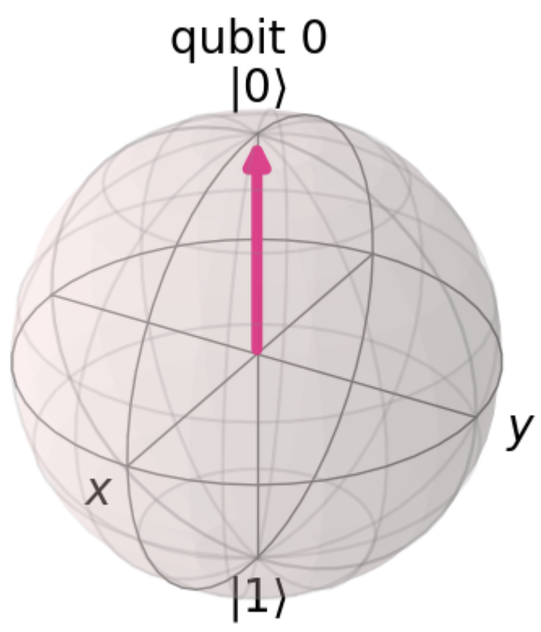
👉 Estado después de U3(pi/3, pi/8, pi/5):
➤ en forma de vector:

$$\left[\sqrt{\tfrac{3}{4}} \quad \tfrac{1}{2}i\right]$$

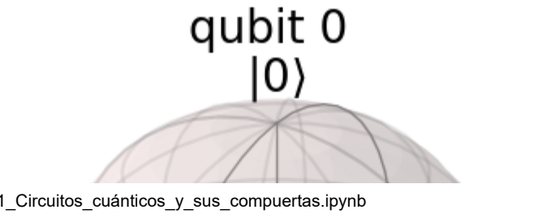
➤ en forma de ket:

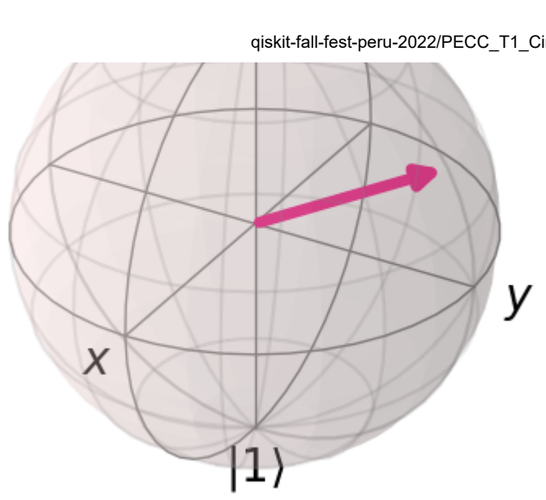
$$\frac{\sqrt{3}}{2}|0\rangle + \frac{i}{2}|1\rangle$$

🔵 Estado inicial:



🔵 Estado después de U3(pi/3, pi/8, pi/5):





4.C. Estados multi-qubit

La dimensión del espacio de Hilbert para n qubits está dado por 2^n. Para obtenerlo se usa el **producto tensorial** (o *producto de Kronecker*).

Recordatorio: Producto tensorial

El producto tensorial, denotado por $\otimes$, se puede aplicar a vectores y matrices (en general a tensores). Actúa como sigue,

$$\vec{a} \otimes \vec{b} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \\ a_2 \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a_1 b_1 \\ a_1 b_2 \\ a_2 b_1 \\ a_2 b_2 \end{pmatrix}$$

$$C \otimes D = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \otimes \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} = \begin{pmatrix} c_{11} \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} & c_{12} \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} \\ c_{21} \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} & c_{22} \begin{pmatrix} d_{11} & d_{12} \\ d_{21} & d_{22} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} c_{11}d_{11} & c_{11}d_{12} & c_{12}d_{11} & c_{12}d_{12} \\ c_{11}d_{21} & c_{11}d_{22} & c_{12}d_{21} & c_{12}d_{22} \\ c_{21}d_{11} & c_{21}d_{12} & c_{22}d_{11} & c_{22}d_{12} \\ c_{21}d_{21} & c_{21}d_{22} & c_{22}d_{21} & c_{22}d_{22} \end{pmatrix}$$

El espacio de 2 qubits

El espacio de Hilebert de un estado cuántico formado por dos qubits, $n = 2$, tendrá una dimensión de $2^2 = 4$. Por ejemplo, el estado formado por dos qubits, cada uno en $|0\rangle$ es,

$$|0\rangle \otimes |0\rangle = |00\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 0 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Ahora, considerando todas las posibles combinaciones de los estados base para cada uno de los dos qubits, tenemos que,

$$|0\rangle \otimes |0\rangle = |00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|0\rangle \otimes |1\rangle = |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

$$|1\rangle \otimes |0\rangle = |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

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

Notemos que debido a que estamos usando la *base computacional* para los qubits, podemos tener una relación directa con la representación de los números decimales en binario; es decir, el estado $|00\rangle$ tiene un 1 en la posición 0 del vector de estado de cuatro elementos, el estado $|10\rangle$ tienen un 1 en la posición 2 (contando desde cero), pues se cumple que $10_{binario} = 2_{decimal}$.

El espacio de 3 qubits

Para el caso de $n = 3$, tenemos que la dimensión es $2^3 = 8$, entonces los vectores de estado para este espacio de Hilbert, en la base computacional, serían:

$$|000\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |001\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |010\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |011\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|100\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |101\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |110\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |111\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

4.D. Compuertas de un qubit en estados multi-qubit

Las compuertas de un solo qubit que actúan en estados multi-qubit, deben tener la dimension correcta para poder efectuar la multiplicación matriz por vector.

Si por ejemplo tenemos esto: $H|q_1 q_0\rangle$, no se puede saber a cual de los dos qubits se le debe aplicar la compuerta H si a $|q_1\rangle$ o a $|q_0\rangle$, entonces se debe ser explícito. Si lo que se desea es aplicar H al $|q_0\rangle$ se debe escribir:

$$I \otimes H \, |q_1 q_0\rangle$$

Lo que implica que se aplicará la compuerta *identidad* al $|q_1\rangle$, es decir, no será afectado.

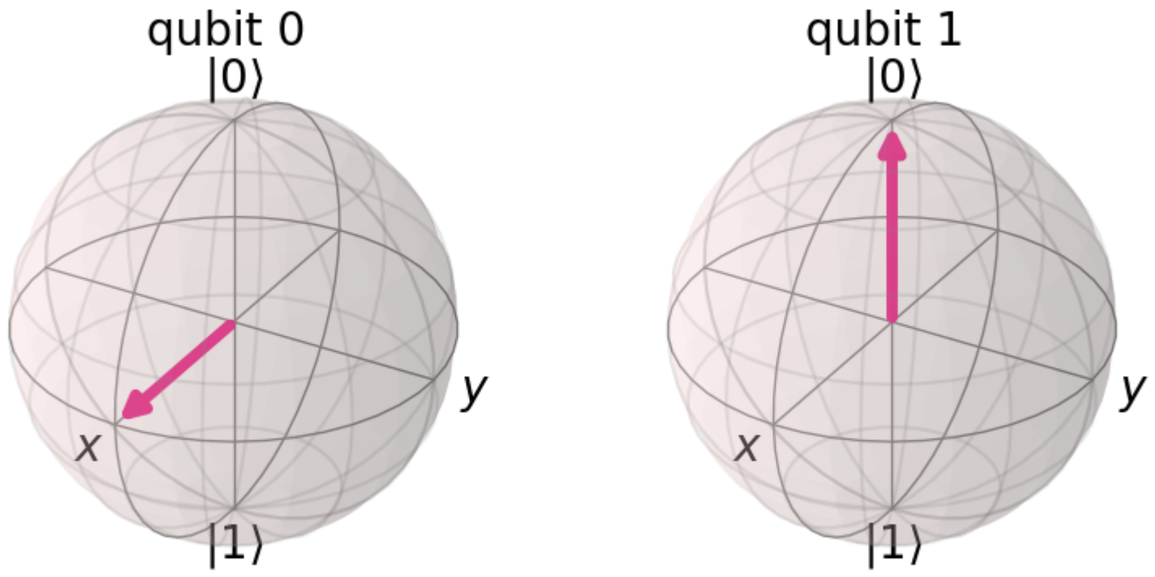
La compuerta $I \otimes H$ a aplicar al estado de 2 qubits es:

$$I \otimes H = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} & 0 \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \\ 0 \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} & 1 \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

En Qiskit podemos usar la clase `Operator` para obtener el operador equivalente a las compuertas agregadas a un circuito, veamos le ejemplo anterior:

In [15]:
```
from qiskit.quantum_info import Operator

circuit = QuantumCircuit(2) # circuito con 2 qubits

circuit.h(0)    # Hadamard en q0
circuit.id(1)   # Identidad en q1

display(circuit.draw('mpl'))

op = Operator(circuit)
display(array_to_latex(op))

psi = Statevector(circuit)

display(plot_bloch_multivector(psi))
```

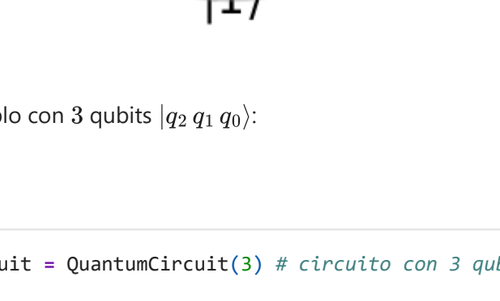
$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$



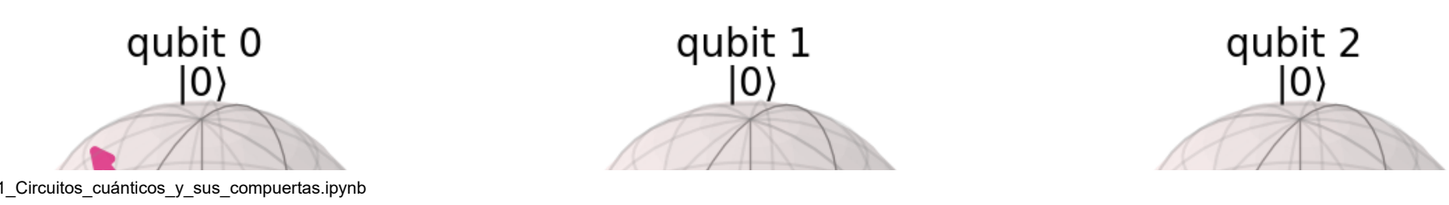
Ejemplo con 3 qubits $|q_2\, q_1\, q_0\rangle$:

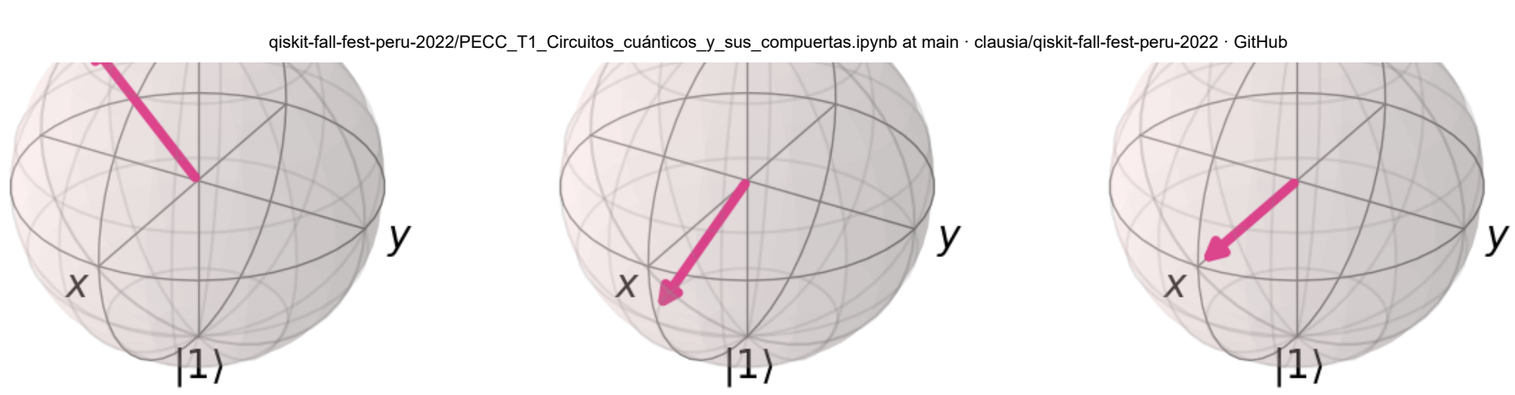
$$H \otimes R_y(3\pi/5) \otimes R_x(\pi/4)\ |000\rangle$$

In [16]:
```
circuit = QuantumCircuit(3) # circuito con 3 qubits

circuit.rx(np.pi/4, 0)      # Rx de pi/4  al q0
circuit.ry(3*np.pi/5, 1)    # Ry de 3pi/5 al q1
circuit.h(2)                # Hadamard al q2

display(circuit.draw('mpl'))

op = Operator(circuit)
display(array_to_latex(op))

psi = Statevector(circuit)

print()
display(plot_bloch_multivector(psi))
```



$$\begin{bmatrix} 0.38399 & -0.15905i & -0.52852 & 0.21892i & 0.38399 & -0.15905i & -0.52852 & 0.21892i \\ -0.15905i & 0.38399 & 0.21892i & -0.52852 & -0.15905i & 0.38399 & 0.21892i & -0.52852 \\ 0.52852 & -0.21892i & 0.38399 & -0.15905i & 0.52852 & -0.21892i & 0.38399 & -0.15905i \\ -0.21892i & 0.52852 & -0.15905i & 0.38399 & -0.21892i & 0.52852 & -0.15905i & 0.38399 \\ 0.38399 & -0.15905i & -0.52852 & 0.21892i & -0.38399 & 0.15905i & 0.52852 & -0.21892i \\ -0.15905i & 0.38399 & 0.21892i & -0.52852 & 0.15905i & -0.38399 & -0.21892i & 0.52852 \\ 0.52852 & -0.21892i & 0.38399 & -0.15905i & -0.52852 & 0.21892i & -0.38399 & 0.15905i \\ -0.21892i & 0.52852 & -0.15905i & 0.38399 & 0.21892i & -0.52852 & 0.15905i & -0.38399 \end{bmatrix}$$





Nota: Si se aplica la misma compuerta a todos los qubits de un estado, se puede indicar con un subíndice, como por ejemplo, la compuerta $H_2 = H \otimes H$ es aplicada a dos qubits: $H_2 \left|00\right\rangle = H \otimes H \left|0\right\rangle \otimes \left|0\right\rangle$ y de la misma manera, se tiene que,

$$H_n \underbrace{\left|0..0\right\rangle}_{n\ qubits} = \underbrace{H \otimes \cdots \otimes H}_{n\ Hadamards} \underbrace{\left|0..0\right\rangle}_{n\ qubits}$$

Notaciones usadas:

$$\underbrace{H \otimes \cdots \otimes H}_{n\ veces} = \bigotimes_{1}^{n} H = H^{\otimes n} = H_n$$

4.E. Compuertas de dos qubits

Ahora que ya sabemos cómo son los estados multi-qubit, podemos revisar las compuertas que son aplicadas a mas de un qubit. Primero veamos algunas compuertas que necesitan dos qubits para operar.

4.E.a. Compuerta $CNOT$ o CX

Se trata de una compuerta controlada, actúa sobre dos qubits, en donde uno de ellos sirve como control y el otro como el objetivo. Se aplicará una operación NOT o X en el qubit objetivo, solo cuando el qubit control sea $\left|1\right\rangle$ y si no lo es, entonces el qubit objetivo permance sin cambio, entonces, si el qubit de la izquierda ($\left|q_1\right\rangle$) es el control y el de la derecha ($\left|q_0\right\rangle$) es el objetivo, en el estado $\left|q_1 q_0\right\rangle$, las transiciones son:

$$CNOT_{1,0}|00\rangle = |00\rangle$$
$$CNOT_{1,0}|01\rangle = |01\rangle$$
$$CNOT_{1,0}|10\rangle = |11\rangle$$
$$CNOT_{1,0}|11\rangle = |10\rangle$$

donde los subíndices en $CNOT$ indican el qubit control y el objetivo, en ese orden.

Dado esto podríamos tener otra versión del $CNOT$ en donde el qubit control sea q_0 y el objetivo sea q_1, las transiciones para este caso serían,

$$CNOT_{0,1}|00\rangle = |00\rangle$$
$$CNOT_{0,1}|01\rangle = |11\rangle$$
$$CNOT_{0,1}|10\rangle = |10\rangle$$
$$CNOT_{0,1}|11\rangle = |01\rangle$$

Conociendo la definición de la compuerta y las transiciones, podemos obtener la matriz que representa a la operación $CNOT$:

Transiciones

Compuerta $CNOT_{1,0}$:

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

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

$|10\rangle \rightarrow |11\rangle$

$|11\rangle \rightarrow |10\rangle$

Matriz

$\hookleftarrow$	$\|00\rangle$	$\|01\rangle$	$\|10\rangle$	$\|11\rangle$
$\|00\rangle$	1	0	0	0
$\|01\rangle$	0	1	0	0
$\|10\rangle$	0	0	0	1
$\|11\rangle$	0	0	1	0

Compuerta $CNOT_{0,1}$:

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

$|01\rangle \rightarrow |11\rangle$

$|10\rangle \rightarrow |10\rangle$

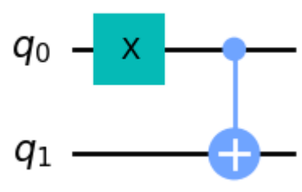
$|11\rangle \rightarrow |01\rangle$

$\hookleftarrow$	$\|00\rangle$	$\|01\rangle$	$\|10\rangle$	$\|11\rangle$
$\|00\rangle$	1	0	0	0
$\|01\rangle$	0	0	0	1
$\|10\rangle$	0	0	1	0
$\|11\rangle$	0	1	0	0

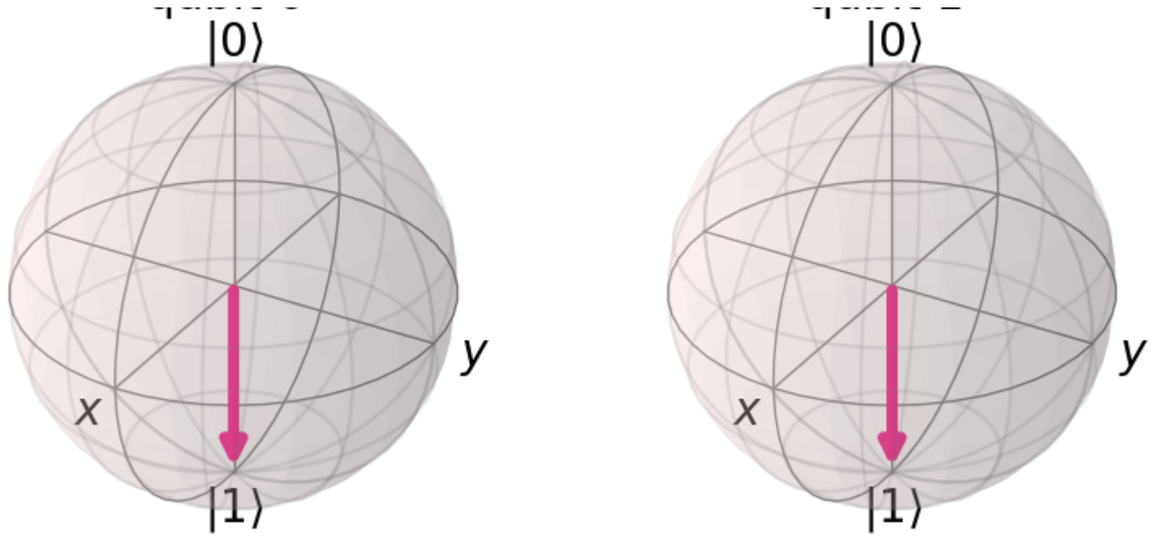
En Qiskit se cuenta con el método `cx(c, t)`, que recibe el qubit de control como primer parámetro y el qubit objetivo como segundo parámetro.

In [17]:
```
circ7 = QuantumCircuit(2)   # circuito con 2 qubits

circ7.x(0)                  # ponemos el control en |1> para que la compuerta sea aplicada

circ7.cx(0, 1)              # CNOT con q0 como control y q1 como objetivo

print("Este es el caso para CNOT(0,1)|01> = |11>")

display(circ7.draw('mpl'))

op = Operator(circ7)
display(array_to_latex(op))

psi = Statevector(circ7)

print()
display(plot_bloch_multivector(psi))
```

```
Este es el caso para CNOT(0,1)|01> = |11>
```



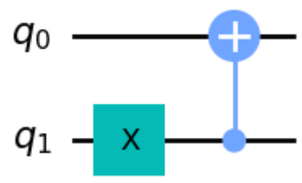
$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$



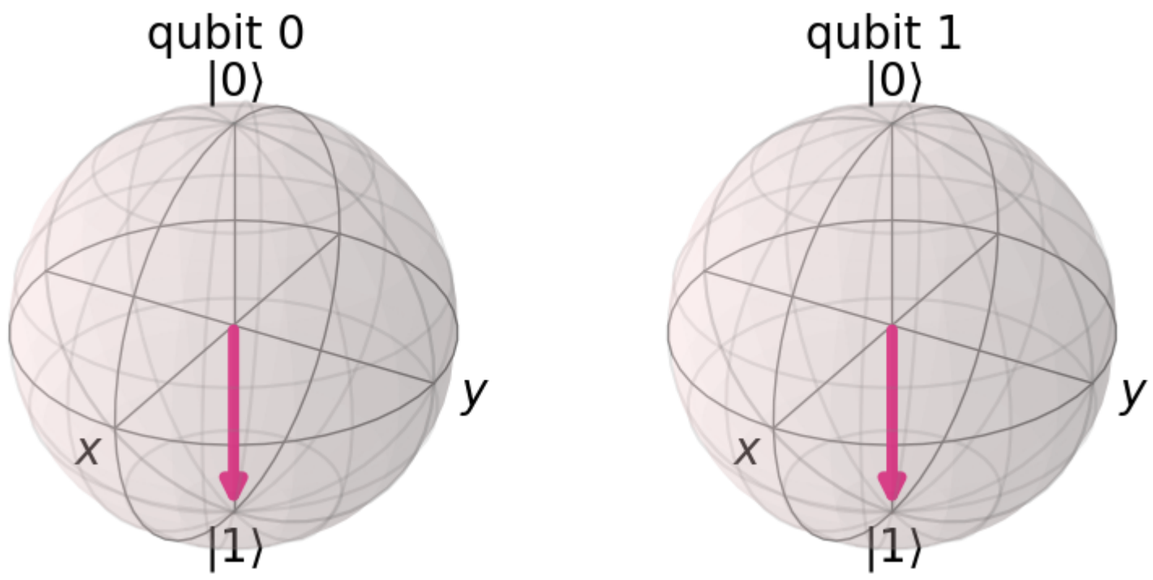
In [18]:

```
circ8 = QuantumCircuit(2)   # circuito con 2 qubits

circ8.x(1)                  # ponemos el control en |1> para que la compuerta sea aplicada

circ8.cx(1, 0)              # CNOT con q1 como control y q0 como objetivo

print("Este es el caso para CNOT(1,0)|10> = |11>")

display(circ8.draw('mpl'))

op = Operator(circ8)
display(array_to_latex(op))

psi = Statevector(circ8)

print()
display(plot_bloch_multivector(psi))
```

```
Este es el caso para CNOT(1,0)|10> = |11>
```



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



4.E.b. Compuertas controladas

Así como podemos controlar si aplicar la compuerta X de acuerdo al estado de un qubit control, podemos controlar cualquier otra compuerta. Si tenemos la compuerta U, su versión controlada sería CU, y podemos decidir cual qubit es el control y cual es el objetivo (como para el caso de $CNOT$).

Una opeeración unitaria en general la podemos escribir como,

$$U = \begin{pmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{pmatrix}$$

Entonces su versión controlada CU, si el control es el qubit de la izquierda ($|q_1\rangle$), tendría las transiciones:

$CU_{1,0}|00\rangle = |00\rangle$

$CU_{1,0}|01\rangle = |01\rangle$

$CU_{1,0}|10\rangle = |1\rangle \otimes U|0\rangle = |1\rangle \otimes (u_{11}|0\rangle + u_{21}|1\rangle)$

$CU_{1,0}|11\rangle = |1\rangle \otimes U|0\rangle = |1\rangle \otimes (u_{12}|0\rangle + u_{22}|1\rangle)$

Su forma matricial es,

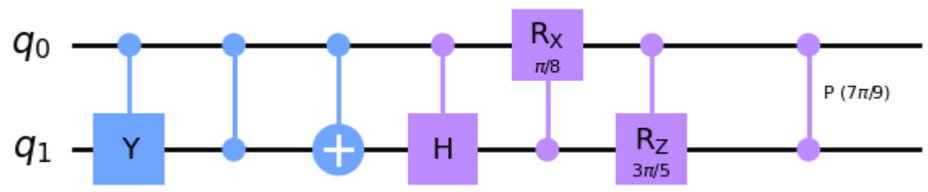
$$CU = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{11} & u_{12} \\ 0 & 0 & u_{21} & u_{22} \end{pmatrix}$$

En Qiskit existen varias compuertas controladas predefinidas, en el siguiente código se usarán algunas a modo de ejemplo:

In [19]:

```
circ9 = QuantumCircuit(2)    # circuito con 2 qubits

circ9.cy(0, 1)               # Y controlada, qubit control es q0, y objetivo es q1
circ9.cz(1, 0)               # Z controlada, qubit control es q1, y objetivo es q0
circ9.cx(0, 1)               # X controlada, qubit control es q0, y objetivo es q1

circ9.ch(0, 1)               # H controlada, qubit control es q0, y objetivo es q1

circ9.crx(np.pi/8, 1, 0)     # Rx controlada, ángulo de pi/8, qubit control es q1, y objetivo es q0
circ9.crz(3*np.pi/5, 0, 1)   # Rz controlada, ángulo de 3pi/5, qubit control es q0, y objetivo es q1
```

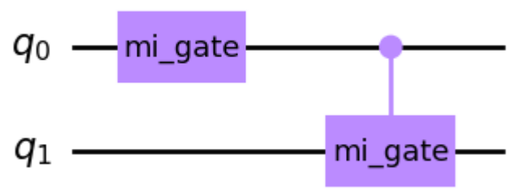
```
circ9.cp(7*np.pi/9, 1, 0)    # P controlada, ángulo de 7pi/9, qubit control es q1, y objetivo es q0

display(circ9.draw('mpl'))
```



En Qiskit también podemos hacer controlada una compuerta particular, en general se usa esta funcionalidad para los casos en los que la compuerta es personalizada, ya que Qiskit incluye la version controlada de la mayoría de las compuertas de un qubit.

Creamos una compuerta personalizada con la clase `UnitaryGate` , a la que le debemos indicar la matriz que define a la operación, recordando que dicha matriz debe ser unitaria. Veamos un ejemplo:

In [20]:
```
from qiskit.extensions import UnitaryGate

matrix = [[1j, 0],
          [0 , 1,]]

mi_gate = UnitaryGate(matrix, 'mi_gate')

circ10 = QuantumCircuit(2)      # circuito con 2 qubits

circ10.append(mi_gate, [0])     # le agregamos al circuito la compuerta personalizada

mi_gate_controlada = mi_gate.control(1)   # hacemos la version controlada de mi_gate

circ10.append(mi_gate_controlada, [0, 1]) # se indican los qubits, primero el control y luego el objetivo

display(circ10.draw('mpl'))
```



4.E.c. Compuerta $SWAP$

Esta compuerta es la única compuerta de dos qubts que no representa una operación controlada. $SWAP$ simplemente intercambia dos qubits. Es muy útil cuando se tiene restrición fisica en las conexiones entre qubits.

Sus transiciones son entonces,

$$SWAP\,|00\rangle = |00\rangle$$
$$SWAP\,|01\rangle = |10\rangle$$
$$SWAP\,|10\rangle = |01\rangle$$
$$SWAP\,|11\rangle = |11\rangle$$

Su representación matricial es,

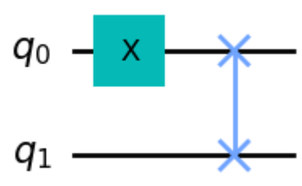
$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La compuerta $SWAP$ puede descomponerse como,

$$SWAP = \frac{I \otimes I + X \otimes X + Y \otimes Y + Z \otimes Z}{2}$$

En Qsikit la podemos usar con `swap()` como sigue:

In [21]:
```
circ11 = QuantumCircuit(2)    # circuito con 2 qubit

circ11.x(0)                   # obtener el estado |01>

psi_0 = Statevector(circ11)   # estado justo después de crear el circuito e iniciar el qubit en |01>

circ11.swap(0, 1)             # aplicar compuerta SWAP a los dos quibits existentes

psi_1 = Statevector(circ11)   # estado después aplicar la compuerta Z

display(circ11.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de SWAP:")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de SWAP:")
display(plot_bloch_multivector(psi_1))
```



👉 Estado inicial:

👉 Estado inicial:
➤ en forma de vector:

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

➤ en forma de ket:

$$|01\rangle$$

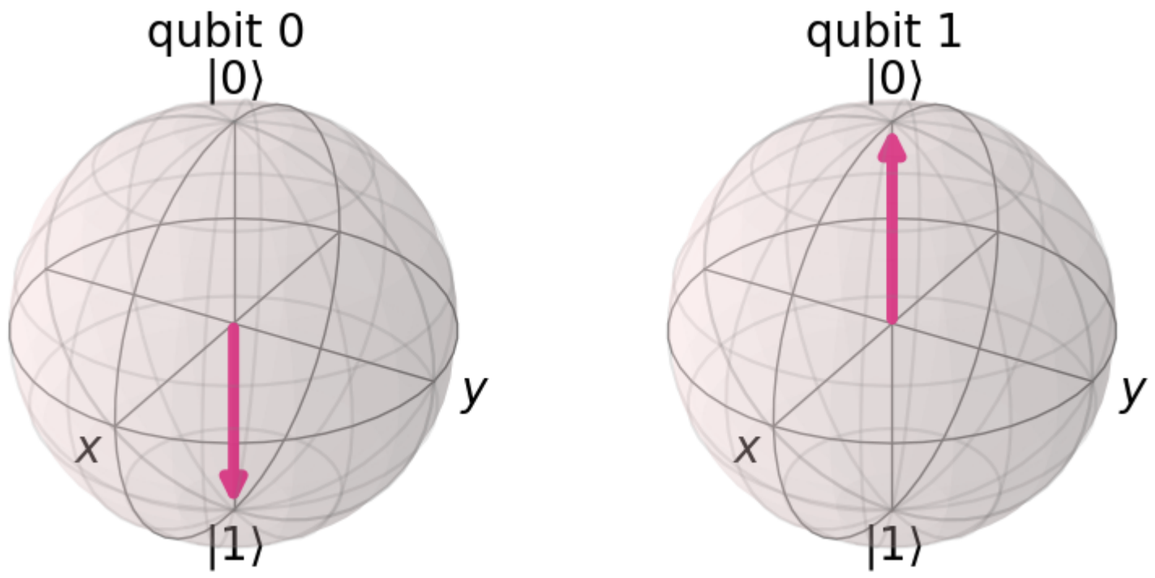
👉 Estado después de SWAP:
➤ en forma de vector:

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

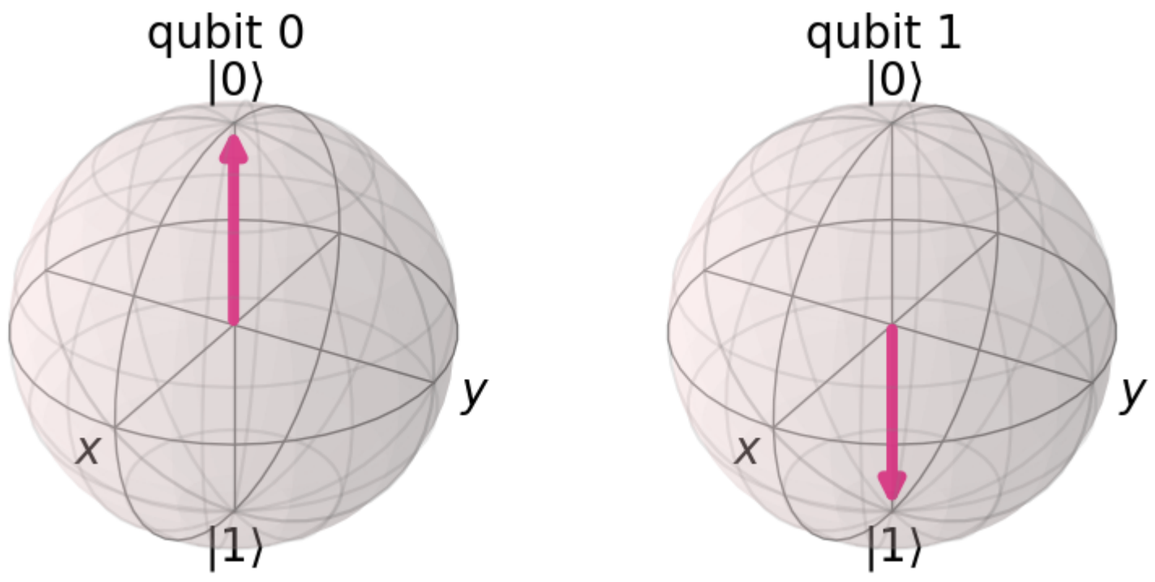
➤ en forma de ket:

$$|10\rangle$$

🔵 Estado inicial:



🔵 Estado después de SWAP:



4.F. Compuertas de más qubits

Las compuertas pueden estar definidas para n cantidad de qubits, siempre y cuando su matriz sea unitaria. Las compuertas de más de dos qubits suelen ser compuertas con más de un control.

4.F.a. Compuerta Toffoli , $CCNOT$ o CCX

Es una compuerta de 3 qubits. Aplicará un NOT si los dos qubits de control están en el estado $|1\rangle$, en caso contrario no aplicará ninguna operación.

Su forma matricial para el caso de que los dos qubits de la izquierda sean los controles, y el de más a la derecha sea el objetivo es como sigue,

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

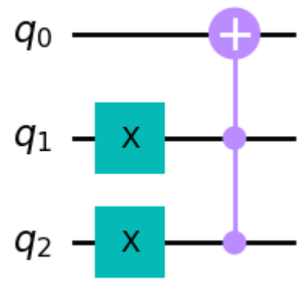
Entonces sus transiciones son,

$$CCX\,|000\rangle = |000\rangle$$
$$CCX\,|001\rangle = |001\rangle$$
$$CCX\,|010\rangle = |010\rangle$$
$$CCX\,|011\rangle = |011\rangle$$
$$CCX\,|100\rangle = |100\rangle$$
$$CCX\,|101\rangle = |101\rangle$$
$$CCX\,|110\rangle = |111\rangle$$
$$CCX\,|111\rangle = |110\rangle$$

In [22]:

```
circ12 = QuantumCircuit(3)    # circuito con 3 qubit

circ12.x(1)                   # obtener el estado |010>
circ12.x(2)                   # obtener el estado |110>

psi_0 = Statevector(circ12)   # estado justo después de crear el circuito e iniciar el qubit en |110>

circ12.ccx(1, 2, 0)           # aplicar compuerta CCX: control1, control2, objetivo

psi_1 = Statevector(circ12)   # estado después aplicar la compuerta Z

display(circ12.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
```

```
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de CCX(1, 2, 0):")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de CCX(1, 2, 0):")
display(plot_bloch_multivector(psi_1))
```




```
👉 Estado inicial:
➤ en forma de vector:
```

$$[0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0]$$

```
➤ en forma de ket:
```

$$|110\rangle$$

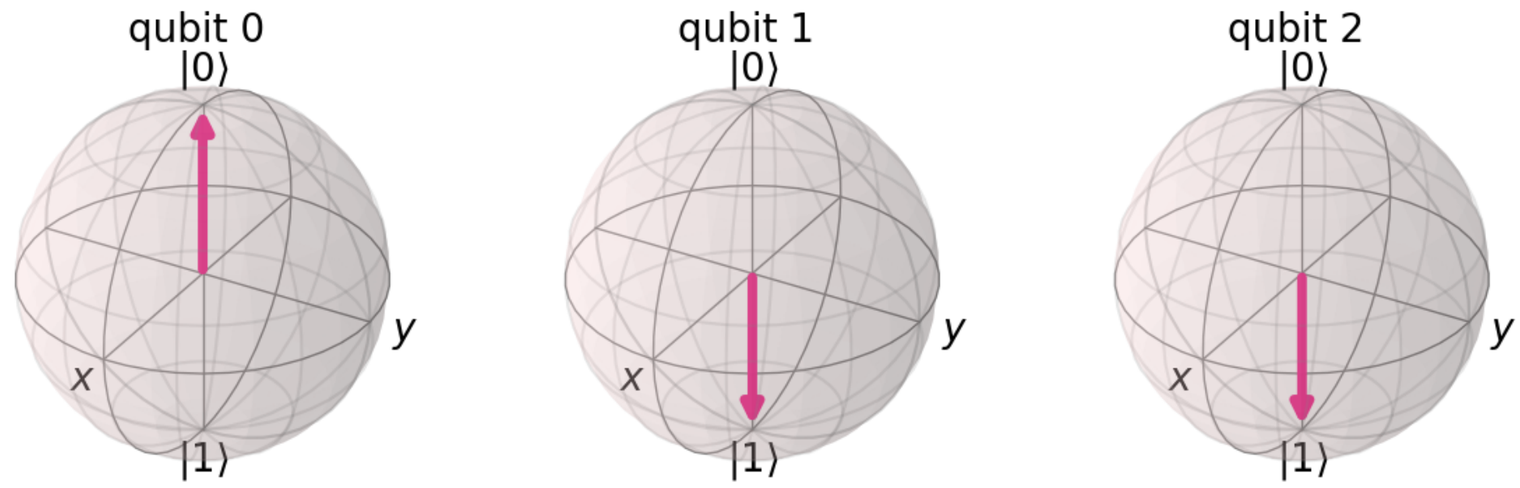
```
👉 Estado después de CCX(1, 2, 0):
➤ en forma de vector:
```

$$[0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1]$$

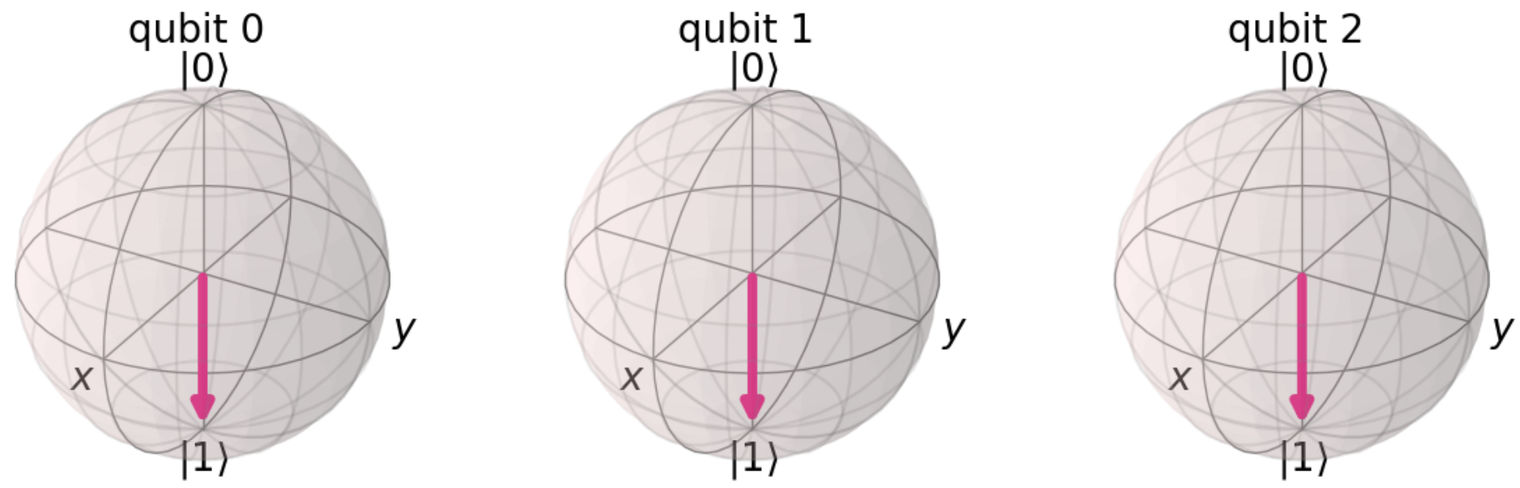
```
➤ en forma de ket:
```

$$|111\rangle$$

```
🔵 Estado inicial:
```




```
🔵 Estado después de CCX(1, 2, 0):
```



4.F.b. Compuerta Fredkin o $CSWAP$

Esta compuerta tiene un qubit de control, y si está en el estado $|1\rangle$, entonces se efectúa un $SWAP$ a los dos quibits objetivo. La forma matricial es como sigue:

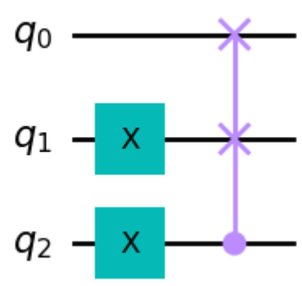
$$CSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Entonces sus transiciones son,

$$CSWAP|000\rangle = |000\rangle$$
$$CSWAP|001\rangle = |001\rangle$$
$$CSWAP|010\rangle = |010\rangle$$
$$CSWAP|011\rangle = |011\rangle$$
$$CSWAP|100\rangle = |100\rangle$$
$$CSWAP|101\rangle = |110\rangle$$
$$CSWAP|110\rangle = |101\rangle$$
$$CSWAP|111\rangle = |111\rangle$$

In [23]:
```
circ13 = QuantumCircuit(3)    # circuito con 3 qubit
```

```
circ13 = QuantumCircuit(3)    # circuito con 3 qubit

circ13.x(2)                   # obtener el estado |100>
circ13.x(1)                   # obtener el estado |110>

psi_0 = Statevector(circ13)   # estado justo después de crear el circuito e iniciar el qubit en |110>

circ13.cswap(2, 0, 1)         # aplicar compuerta CCX: control1, objetivo1, objetivo2

psi_1 = Statevector(circ13)   # estado después aplicar la compuerta Z

display(circ13.draw('mpl'))   # mostrar el circuito

print("👉 Estado inicial:")
print("➤ en forma de vector:")
display(array_to_latex(psi_0))
print("➤ en forma de ket:")
display(psi_0.draw('latex'))
print()
print("👉 Estado después de SWAP(2, 0, 1):")
print("➤ en forma de vector:")
display(array_to_latex(psi_1))
print("➤ en forma de ket:")
display(psi_1.draw('latex'))

print()
print("🔵 Estado inicial:")
display(plot_bloch_multivector(psi_0))

print("🔵 Estado después de SWAP(2, 0, 1):")
display(plot_bloch_multivector(psi_1))
```




```
👉 Estado inicial:
➤ en forma de vector:
```

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

```
➤ en forma de ket:
```

$$|110\rangle$$

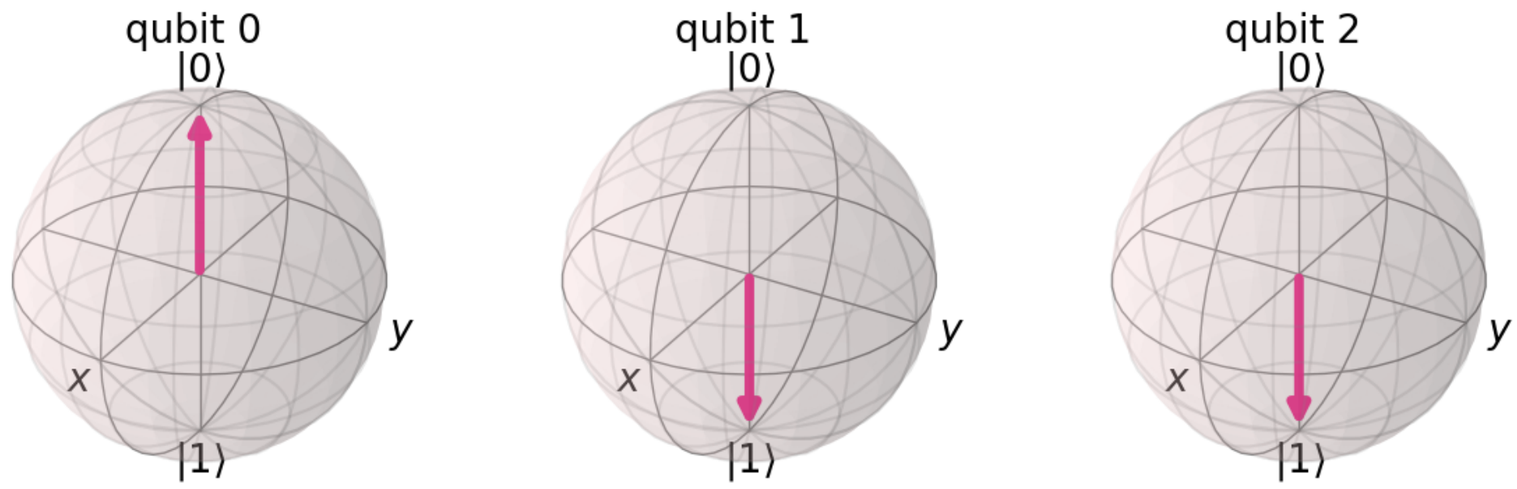
```
👉 Estado después de SWAP(2, 0, 1):
➤ en forma de vector:
```

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

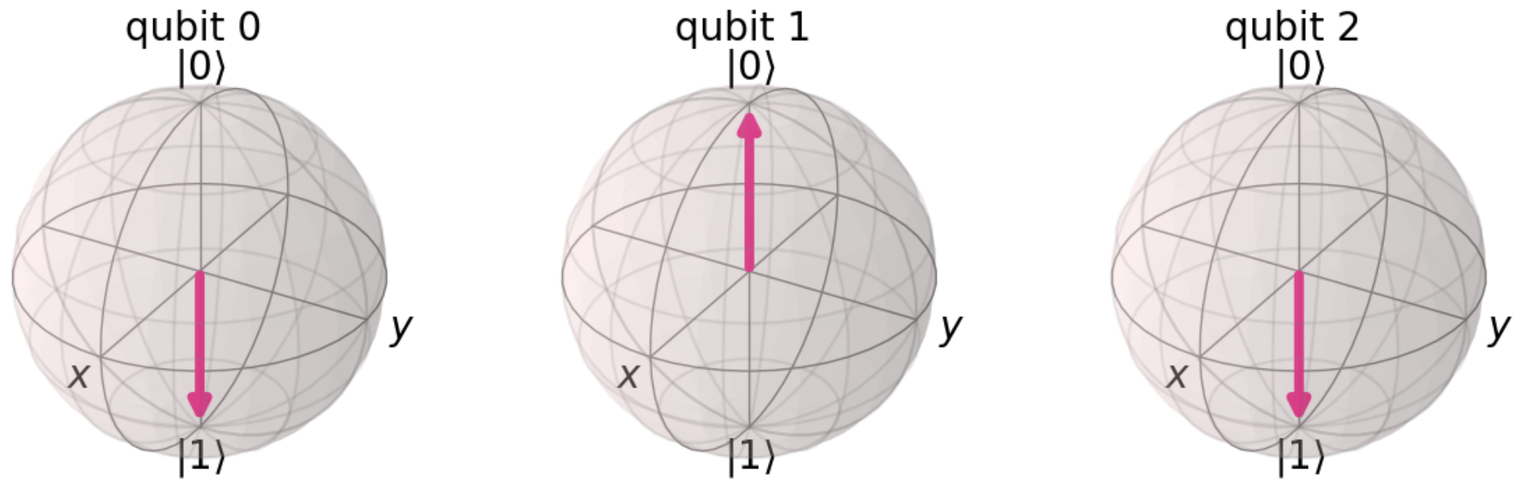
```
➤ en forma de ket:
```

$$|101\rangle$$

```
🔵 Estado inicial:
```




```
🔵 Estado después de SWAP(2, 0, 1):
```



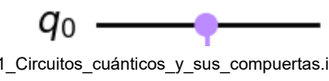
4.F.c. Compuerta controlada personalizada

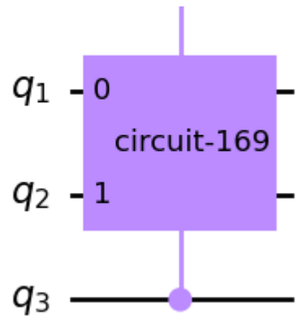
Compuerta personalizada en Qiskit, con dos controles y dos objetivos:

In [24]:

```
# Crear una comperta a partir de un circuito
qc1 = QuantumCircuit(2)
# estos dos quits serán los objetivos
qc1.x(0)
qc1.h(1)
custom = qc1.to_gate().control(2) # se indica que debe ser controlada con dos qubits

# Aplicar esa compuerta a otro circuito
qc2 = QuantumCircuit(4)
qc2.append(custom, [0, 3, 1, 2]) # primeros 2 son control, el resto a los que se aplica la compuerta
qc2.draw('mpl')
```

Out[24]:





5. Estados de Bell: Entrelazamiento

Existen estados particulares con su propio nombre, tal es el caso de los estados de Bell. Son relevantes porque una vez creados dichos estados, se tiene que los dos qubits que los conforman están entrelazados. Representan los ejemplos más simples del entrelazamiento cuántico.

Que los qubits estén entralazados significa que el estado no puede ser separado en el producto tensorial de dos qubits como $|a\rangle \otimes |b\rangle$.

Conozcamos los 4 estados de Bell:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}\Big(|00\rangle + |11\rangle\Big) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}\Big(|00\rangle - |11\rangle\Big) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\-1\end{pmatrix}$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}\Big(|01\rangle + |10\rangle\Big) = \frac{1}{\sqrt{2}}\begin{pmatrix}0\\1\\1\\0\end{pmatrix}$$

$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}\Big(|01\rangle - |10\rangle\Big) = \frac{1}{\sqrt{2}}\begin{pmatrix}0\\1\\-1\\0\end{pmatrix}$$

Ejemplo: Confirmar que el estado $|\Phi^+\rangle$ no puede ser separado como $|a\rangle \otimes |b\rangle$.
Supongamos que si se puede, entonces se debe cumplir que,

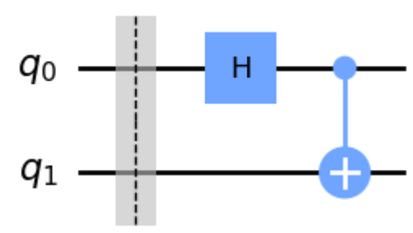
$$|a\rangle \otimes |b\rangle = \begin{pmatrix}a_1\\a_2\end{pmatrix} \otimes \begin{pmatrix}b_1\\b_2\end{pmatrix} = \begin{pmatrix}a_1b_1\\a_1b_2\\a_2b_1\\a_2b_2\end{pmatrix} = \begin{pmatrix}1\\0\\0\\1\end{pmatrix}$$

Entonces $a_1b_2 = 0$, lo que implica que $a_1 = 0$ o que $b_2 = 0$, si $a_1 = 0$ entonces no puede cumplirse que $a_1b_1 = 1$, y si en cambio sucede que $b_2 = 0$, entonces hay contradicción con $a_2b_2 = 1$.

Por lo tanto no es posible expresar el estado $|\Phi^+\rangle$ como el producto tensorial de dos qubits separados ($|a\rangle \otimes |b\rangle$).

Veamos como generar los estados de Bell en Qiskit:

In [25]:
```
circ_bell_1 = QuantumCircuit(2)    # circuito con 2 qubits

circ_bell_1.barrier()
circ_bell_1.h(0)                   # aplicar compuerta H al qubit 0
circ_bell_1.cx(0,1)                # aplicar compuerta CNOT

psi_bell_1 = Statevector(circ_bell_1)   # estado final

display(circ_bell_1.draw('mpl'))   # mostrar el circuito

print("👉 Estado de Bell 1:")
print("➤ en forma de vector:")
display(array_to_latex(psi_bell_1))
print("➤ en forma de ket:")
display(psi_bell_1.draw('latex'))
```



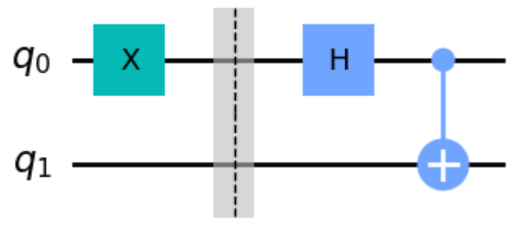
👉 Estado de Bell 1:
➤ en forma de vector:

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

➤ en forma de ket:

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

In [26]:
```
circ_bell_2 = QuantumCircuit(2)    # circuito con 2 qubits

circ_bell_2.x(0)                   # aplicar compuerta X al qubit 0
circ_bell_2.barrier()
circ_bell_2.h(0)                   # aplicar compuerta H al qubit 0
circ_bell_2.cx(0,1)                # aplicar compuerta CNOT

psi_bell_2 = Statevector(circ_bell_2)   # estado final

display(circ_bell_2.draw('mpl'))   # mostrar el circuito

print("👉 Estado de Bell 2:")
print("➤ en forma de vector:")
display(array_to_latex(psi_bell_2))
print("➤ en forma de ket:")
display(psi_bell_2.draw('latex'))
```



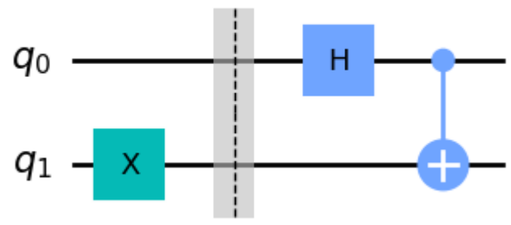
👉 Estado de Bell 2:
➤ en forma de vector:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

➤ en forma de ket:

$$\frac{\sqrt{2}}{2}|00\rangle - \frac{\sqrt{2}}{2}|11\rangle$$

In [27]:
```
circ_bell_3 = QuantumCircuit(2)    # circuito con 2 qubits

circ_bell_3.x(1)                   # aplicar compuerta X al qubit 1
circ_bell_3.barrier()
circ_bell_3.h(0)                   # aplicar compuerta H al qubit 0
circ_bell_3.cx(0,1)                # aplicar compuerta CNOT

psi_bell_3 = Statevector(circ_bell_3)   # estado final

display(circ_bell_3.draw('mpl'))   # mostrar el circuito

print("👉 Estado de Bell 2:")
print("➤ en forma de vector:")
display(array_to_latex(psi_bell_3))
print("➤ en forma de ket:")
display(psi_bell_3.draw('latex'))
```



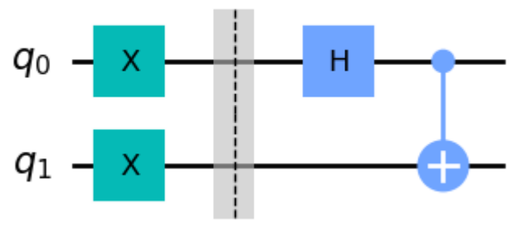
👉 Estado de Bell 2:
➤ en forma de vector:

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

➤ en forma de ket:

$$\frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

In [28]:
```
circ_bell_4 = QuantumCircuit(2)    # circuito con 2 qubits

circ_bell_4.x(0)                   # aplicar compuerta X al qubit 0
circ_bell_4.x(1)                   # aplicar compuerta X al qubit 1
circ_bell_4.barrier()
circ_bell_4.h(0)                   # aplicar compuerta H al qubit 0
circ_bell_4.cx(0,1)                # aplicar compuerta CNOT

psi_bell_4 = Statevector(circ_bell_4)   # estado final

display(circ_bell_4.draw('mpl'))   # mostrar el circuito

print("👉 Estado de Bell 2:")
print("➤ en forma de vector:")
display(array_to_latex(psi_bell_4))
print("➤ en forma de ket:")
display(psi_bell_4.draw('latex'))
```



👉 Estado de Bell 2:
➤ en forma de vector:

$$\begin{bmatrix} 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

➤ en forma de ket:

$$-\frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

6. Simular Circuitos Cuánticos

Ya sabemos crear cicuitos y manipular sus qubits. Lo que sigue es ejecutar dichos circuitos y obtener la ventajas de la parte cuántica, es decir, permitir que las probabiliadades entren en juego al momento de observar los resultados.

Hasta ahora, hemos mostrado información sobre el estado final (e intermedios) suponiendo un circuito ideal y sin forzar un colapso del resutado, hemos visto el estado como si solo efectuaramos las operaciones matemáticas, comprobando que las mismas funcionan, pero no hemos hecho nada cuántico.

Sin embargo, un experimento real termina midiendo cada qubit (normalmente sobre la base computacional $\{|0\rangle, |1\rangle\}$). Sin medición, no podemos obtener información sobre el estado. **Las mediciones hacen que el sistema cuántico colapse en bits clásicos**.

Supongamos el siguiente estado de 3 qubits, conocido como estado GHZ,

$$|\psi\rangle = \frac{1}{\sqrt{2}}\Big(|000\rangle + |111\rangle\Big)$$

Denotemos la cadena de bits resultane como abc. Recordemos que, bajo el etiquetado de qubits utilizado por Qiskit ($|q_2 q_1 q_0\rangle$), a correspondería al resultado en el qubit 2, b al resultado en el qubit 1, y c al resultado en qubit 0.

Ahora, la probabilidad de obtener un resultado abc viene dada por,

$$Pr(abc) = |\langle abc|\psi\rangle|^2$$

En particular para el estado GHZ, la probabilidad de obtener 000 o 111 es $\frac{1}{2}$.

Ejemplo: Calcular la probabilidad de medir 111 en el estado $|\psi\rangle = \frac{1}{\sqrt{2}}\Big(|000\rangle + |111\rangle\Big)$.

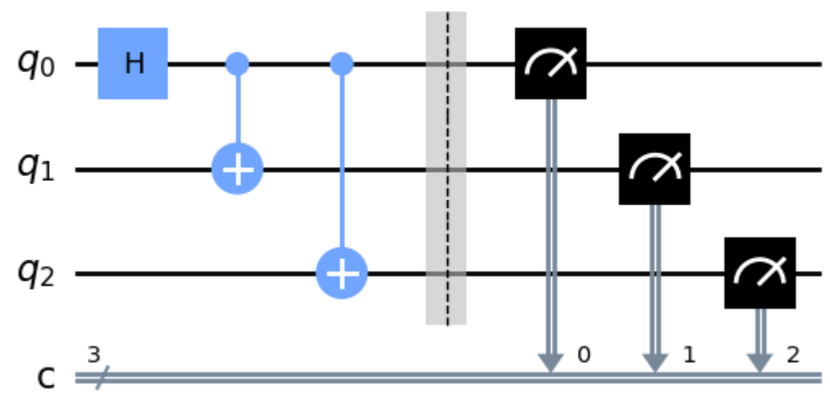
Utilizamos la ecuación indicada para calcular la probabilidad,

$$\begin{aligned}
\Pr(111) &= |\langle 111|\psi\rangle|^2 \\
&= \left|\left\langle 111\right| \frac{1}{\sqrt{2}}\Big(|000\rangle + |111\rangle\Big)\right\rangle\Big|^2 \\
&= \left|\frac{1}{\sqrt{2}}\langle 111\Big|\Big(|000\rangle + |111\rangle\Big)\rangle\right|^2 \\
&= \left|\frac{1}{\sqrt{2}}\Big(\langle 111|000\rangle + \langle 111|111\rangle\Big)\right|^2 \\
&= \left|\frac{1}{\sqrt{2}}\Big(0 + 1\Big)\right|^2 \\
&= \left|\frac{1}{\sqrt{2}}\right|^2 \\
&= \frac{1}{2}
\end{aligned}$$

Para poder simular un circuito, es necesario indicar que qubits queremos medir y elegir en dónde queremos guardar la medición, serán bits clásicos y necesitamos uno por cada qubit a medir.

In [29]:
```
circ_ghz = QuantumCircuit(3, 3)     # circuito con 3 qubits y 3 bits clásicos

circ_ghz.h(0)                       # aplicar compuerta H al qubit 0
circ_ghz.cx(0, 1)                   # aplicar compuerta CNOT a los qubits 0 y 1
circ_ghz.cx(0, 2)                   # aplicar compuerta CNOT a los qubits 0 y 2
circ_ghz.barrier()
circ_ghz.measure([0,1,2], [0,1,2])  # medir los 3 qubits en los 3 bits clásicos

circ_ghz.draw('mpl')                # mostrar el circuito
```

Out[29]:



Para simular este circuito, usamos el `aer_simulator` de Qiskit Aer. Cada ejecución de este circuito producirá la cadena de bits 000 o 111.

Entonces, si ejecutamos este circuito una sola vez, obtendremos la salida 000 o 111, con probabilidad de $\frac{1}{2}$. Es decir, al medir el estado cuántico, estamos obligando a que colapse en una de las dos opciones que lo componen.

In [75]:
```
from qiskit import execute, Aer

simulator = Aer.get_backend('aer_simulator')

job = execute(circ_ghz, simulator, shots=1)              # ejecutar el circuito una sola vez
counts = job.result().get_counts(circ_ghz)               # obtener los resultados de la ejecución
print(counts)
```

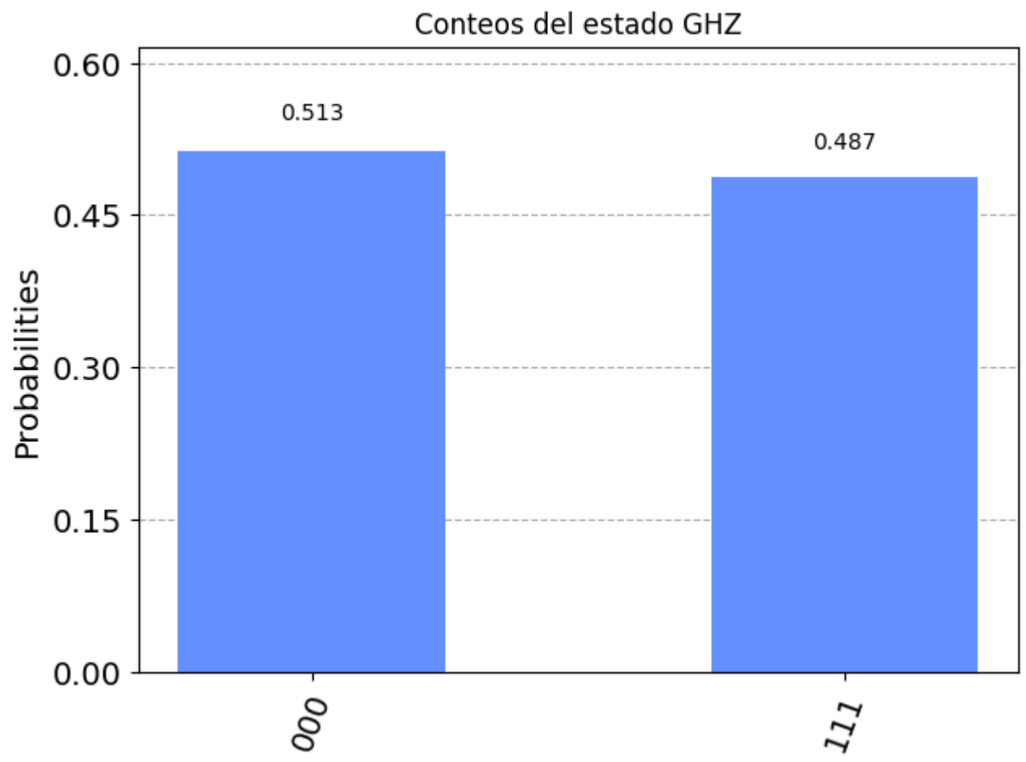
```
{'000': 1}
```

Para **generar estadísticas** sobre la distribución de las cadenas de bits (para, por ejemplo, estimar $Pr(000)$), necesitamos repetir el circuito muchas veces. El número de veces que se repite el circuito se puede especificar en la función `execute`, mediante la palabra clave `shots`.

In [31]:
```
from qiskit.tools.visualization import plot_histogram

job = execute(circ_ghz, simulator, shots=1000)           # ejecutar el circuito 1000 veces
counts = job.result().get_counts(circ_ghz)               # obtener los resultados de la ejecución
print(counts)

plot_histogram(counts, title='Conteos del estado GHZ')   # mostrar los resultados en forma de histograma
```

```
{'111': 487, '000': 513}
```

Out[31]:



Las probabilidades de salida estimadas $Pr(000)$ y $Pr(111)$ se calculan tomando los conteos agregados y dividiendo entre el número de iteraciones (cantidad de veces que el circuito fue repetido).

El comportamiento predeterminado del backend `AerSimulator` es imitar la ejecución de un dispositivo real. Al ejecutar un `QuantumCircuit` que contiene mediciones, devolverá un diccionario de conteos que contiene los valores finales de cualquier registro clásico del circuito.

contiene mediciones, devolverá un diccionario de conteos que contiene los valores finales de cualquier registro clásico del circuito.

Existen diferentes **backends locales** de simulación, podemos consultarlos con la siguiente instrucción:

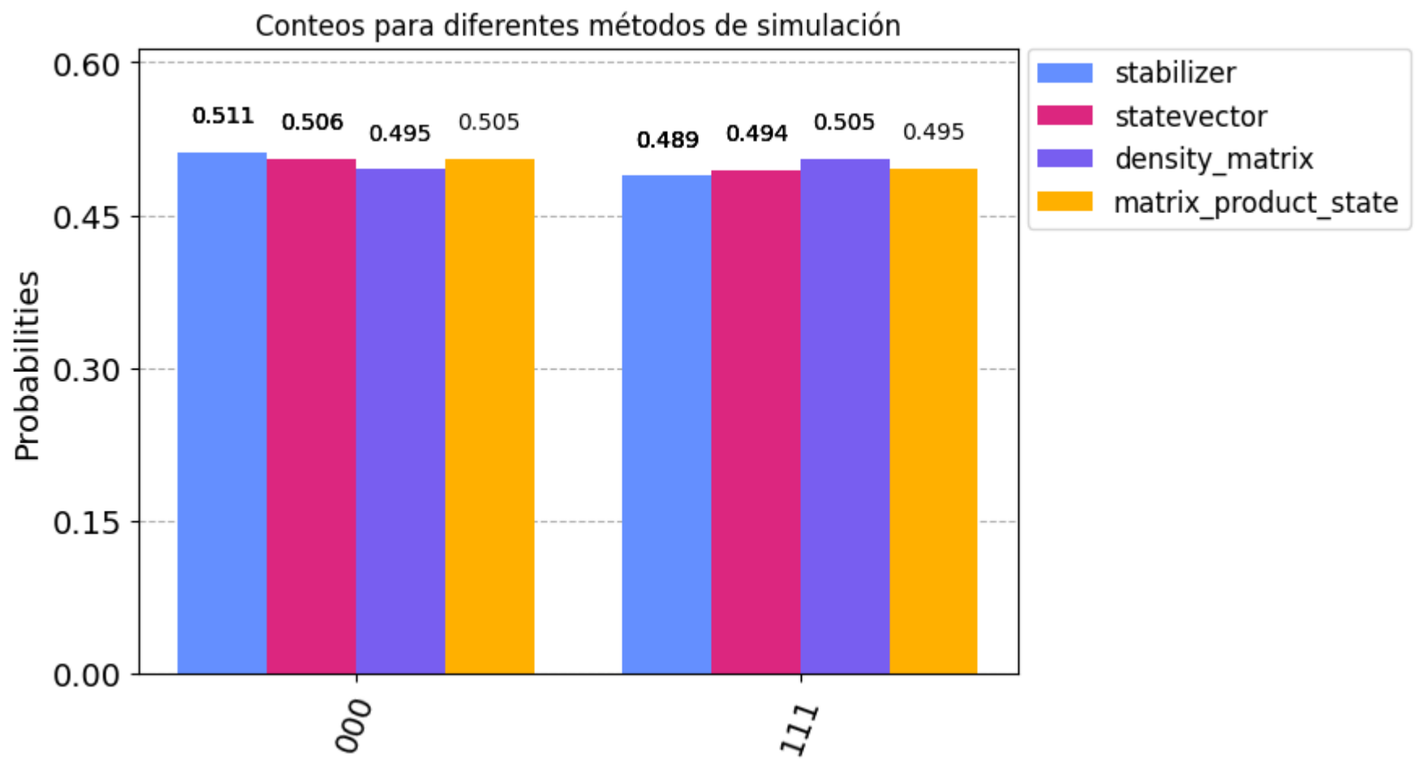
In [32]:
```
Aer.backends()
```

Out[32]:
```
[AerSimulator('aer_simulator'),
 AerSimulator('aer_simulator_statevector'),
 AerSimulator('aer_simulator_density_matrix'),
 AerSimulator('aer_simulator_stabilizer'),
 AerSimulator('aer_simulator_matrix_product_state'),
 AerSimulator('aer_simulator_extended_stabilizer'),
 AerSimulator('aer_simulator_unitary'),
 AerSimulator('aer_simulator_superop'),
 QasmSimulator('qasm_simulator'),
 StatevectorSimulator('statevector_simulator'),
 UnitarySimulator('unitary_simulator'),
 PulseSimulator('pulse_simulator')]
```

Cuando se simulan **circuitos cuánticos ideales**, al cambiar el simulador no debería cambiar el resultado de la simulación (aparte de las variaciones habituales de las probabilidades de muestreo para los resultados de medición).

In [33]:
```
# incrementar la cantidad de 'shots' reduce la variación en el muestreo
shots = 10000

# Método de simulación: Stabilizer
sim_stabilizer = Aer.get_backend('aer_simulator_stabilizer')
job_stabilizer = sim_stabilizer.run(circ_ghz, shots=shots)   #también se puede usar el método 'run' del backend
counts_stabilizer = job_stabilizer.result().get_counts(0)

# Método de simulación: Statevector
sim_statevector = Aer.get_backend('aer_simulator_statevector')
job_statevector = sim_statevector.run(circ_ghz, shots=shots)
counts_statevector = job_statevector.result().get_counts(0)

# Método de simulación: Density Matrix
sim_density = Aer.get_backend('aer_simulator_density_matrix')
job_density = sim_density.run(circ_ghz, shots=shots)
counts_density = job_density.result().get_counts(0)

# Método de simulación: Matrix Product State
sim_mps = Aer.get_backend('aer_simulator_matrix_product_state')
job_mps = sim_mps.run(circ_ghz, shots=shots)
counts_mps = job_mps.result().get_counts(0)

plot_histogram([counts_stabilizer, counts_statevector, counts_density, counts_mps],
               title='Conteos para diferentes métodos de simulación',
               legend=['stabilizer', 'statevector',
                       'density_matrix', 'matrix_product_state'])
```

Out[33]:



La diferencia al usar uno u otro simulador, será notoria cuando, por ejemplo, se simulen circuitos que modelen ruido, o que posean un conjunto específico de instrucciones. Pera detalles sobre cada método de simulación se puede consultar la documentación de la clase `AerSimulator` . En este tutorial se puede aprender cómo obtener los detalles de os simuladores.

7. Ejecutar Circuitos en Computadoras Cuánticas Reales

Para poder usar los dispositivos cuánticos en la nube, es necesario contar con una cuenta en la plataforma IBM Quantum.

Una vez que se ha creado la cuenta y se ha iniciado sesión, en la págna principal se debe generar un **Token**, el cual está formado por una cadena alfanumérica, la misma plataforma te permite copiar esta cadena (es muy larga), una vez que esté generada, se debe copiar y pegar en la celda de abajo sustituyendo `TOKEN` y luego se debe ejecutar el método `save_account()` .

Nota: no es necesario que se ejecute `save_account()` cada vez, con que se haga una vez será suficiente, incluso cuando pase mucho tiempo entre ejecuciones, el *token* quedará guardado en disco duro. A menos que por alguna razón se regenere el *token* entonces si será necesario volerlo a guardar.

Lo que si se debe hacer cada vez que se inicia un contexto, por ejemplo abrir un nuevo notebook, es cargar la cuenta con ayuda del método `IBMQ.load_account()` .

El método `IBMQ.providers()` muestra la lista de proveedores a los que se tiene acceso, por ejemplo, con una cuenta pública, solo se verá enlistado el proveedor *ibm-q/open/main*.

Nota: Tenemos tanto simuladores como computadoras cuánticas reales a nuestra dsposición en la nube.

In [34]:
```
from qiskit import IBMQ

#IBMQ.save_account('TOKEN')  # guardar tu cuenta en tu disco duro

IBMQ.load_account()         # cargar cuenta desde el disco duro

IBMQ.providers()            # listar todos los proveedores disponibles (para tu cuenta)
```

Out[34]:
```
[<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>,
 <AccountProvider for IBMQ(hub='q-summer-school', group='main', project='project-2')>]
```

7.A. Dispositivos en la nube

Existen diferentes **backends en la nube**, podemos consultarlos con la ayuda de algún *provider* que se tenga asignado:

In [35]:
```
provider = IBMQ.get_provider(hub='ibm-q')                # obtenemos un proveedor
```

In [36]:
```
provider.backends()
```

Out[36]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

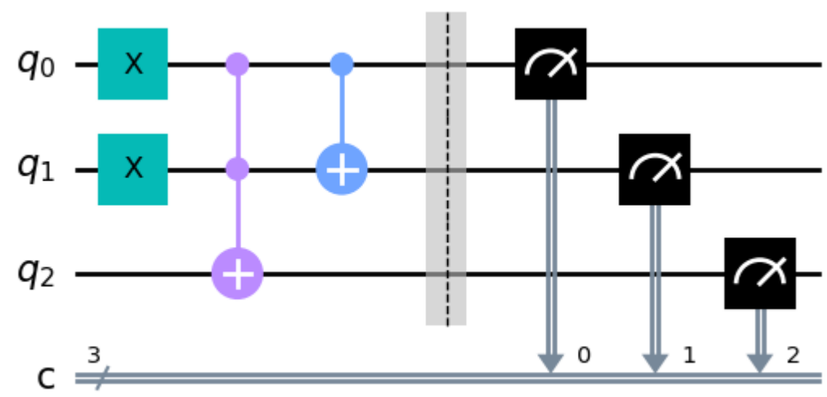
7.A.a. Simuladores en la nube

Para ejecutar experimentos en un dispositivo real, lo primero es cargar la cuenta, como lo hicimos en la sección anterior, posteriormente debemos obtener el *proveedor* que deseamos usar, y de éste obtener un *backend* a donde enviaremos nuestro circuito cuántico.

El código para efectuar la ejecución es similar a como cuando se realiza con simuladores locales, es decir, usando el método `backend.run()` al cual le enviamos como parámetro el circuito que deseamos sea ejecutado en el dispositivo.

Veamos un ejemplo:

In [37]:
```
provider = IBMQ.get_provider(hub='ibm-q')                # obtenemos un proveedor

backend = provider.get_backend('ibmq_qasm_simulator')    # obtenemos un backend, en este ejemplo estamos usando
                                                         # un simulador, pero uno que se encuentra en la nube,
                                                         # es decir, la ejecución no ocurrirá en nuestra máquina local
                                                         # esto para obtener un resultado de manera rápida,
                                                         # más adelante veremos cómo ejecutar en un dispositivo real

circuit = QuantumCircuit(3, 3)                           # creamos un circuito de 3 qubits y 3 registros clásicos
circuit.x(0)                                             # aplicamos algunas compuertas
circuit.x(1)
circuit.ccx(0, 1, 2)
circuit.cx(0, 1)
circuit.barrier()
circuit.measure([0, 1, 2], [0, 1, 2])                    # agregamos mediciones a los 3 qubits

display(circuit.draw('mpl'))                             # mostramos el circuito

job = backend.run(circuit)                               # mandamos ejecutar el circuito al backend en la nube,
                                                         # recuerda, este aún es un simulador, pero no local
                                                         # esta línea de código puede tardarse varios segundos, o
                                                         # incluso minutos dependiendo de la saturación de la red

result = job.result()                                    # la ejecución nos regresa un 'job'

counts = result.get_counts()                             # al que le podemos pedir los conteos

print(counts)                                            # imprimimos los conteos
```




```
{'101': 4000}
```

Los simuladores en la nube tienen la ventaja de que no consumen recursos en nuestra máquina local y, a su vez, al estar en los servidores de IBM, tienen grandes recursos para ejecutar nuestros circuitos.

Por ejemplo, el `ibmq_qasm_simulator` que acabamos de usar, tiene 32 qubits, lo que significa que es capaz de manejar 2^{32} números complejos, y considerando que para representar un número complejo necesitamos 2 números reales, que en Python son tipos de datos flotantes, los cuales ocupan 64 bits, lo cual implica un uso de memoria de

$$64\,\texttt{bits}\cdot 2\,\texttt{números reales}\cdot 2^{32}\,\texttt{números complejos} = 64\,\texttt{Gigabytes}$$

Que es una cantidad de memoria RAM con la que pocas computadoras personales cuentan, además, esta es la cantidad de memoria **solo** para representar los quibits, sigue siendo necesaria más memoria para el sistema operativo, para los cálculos con los qubits, para la representación de las compuertas cuánticas, entre otros recursos necesarios para que funcione la computadora clásica (es clásica pues se trata de un simulador).

Por lo que considera el uso de estos simuladores en la nube, ya que pueden ser de gran ayuda al momento de ejecutar circuitos cuánticos grandes que se lleguen a necesitar al resolver problemas complejos.

7.A.b. Dispositivos reales en la nube

Ahora bien, con nuestro proveedor también tenemos accceso a computadoras cuánticas reales y para saber cuales son, podemos aplicar un filtro y obtener así solamente los dispositivos reales:

In [38]:
```
# obtenemos todos los backends que no sean simuladores y que estén en línea

provider.backends(simulator=False, operational=True)
```

Out[38]:
```
[<IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

Los filtros pueden ser más específicos, como en este ejemplo:

In [39]:
```
# obtenemos todos los backends con más de 3 qubits, que no sean simuladores y que estén en línea

provider.backends(filters=lambda x: x.configuration().n_qubits >= 3
                                    and not x.configuration().simulator
                                    and x.status().operational==True)
```

Out[39]:
```
[<IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='main')>,
```

```
<IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='main')>,
<IBMQBackend('ibm_nairobi') from IBMQ(hub='ibm-q', group='open', project='main')>,
<IBMQBackend('ibm_oslo') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

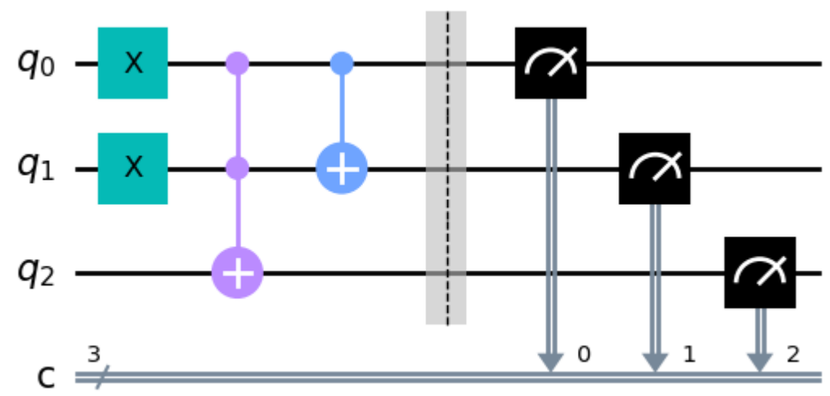
Esos dispositivos enlistados están al alcance de todos aquellos que tengan una cuenta en la plataforma, por lo que usualmente hay que formarse para que un experimento sea ejecutado, esta formación cambia constatemente, si no es muy importante en qué dispositivo se desea ejecutar un circuito, simplemente se debe elegir un *backend*, pero si más bien lo que se desea es que se ejecute lo antes posible, se puede preguntar por el **dispositivo menos ocupado**, con la función `least_busy` .

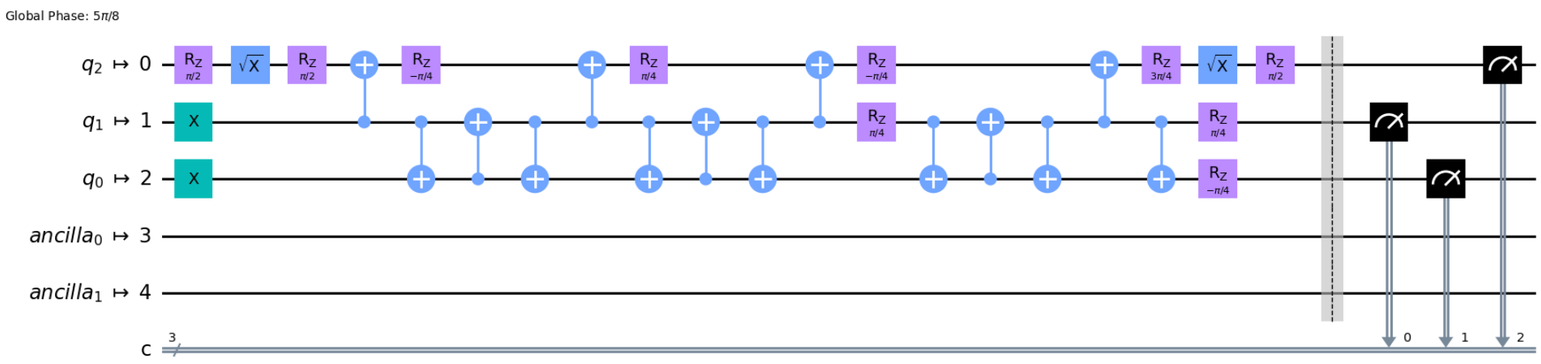
In [40]:
```
from qiskit.providers.ibmq import least_busy

devices = provider.backends(simulator=False, operational=True)

least_busy_device = least_busy(devices)                # de la lista de dispositivos, averiguar cual es el menos ocupado

least_busy_device
```

Out[40]:
```
<IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>
```

La ejecución en un dispositivo real es muy similar a la ejecución en un dispositivo de simulación, con la salvedad de que se debe ejecutar una transpilación previa, debido a que los dispositivos reales solamente aceptan cierto conjunto de compuertas, esto se logra pasando el circuito por la función `transpile()` . En el siguiente ejemplo se puede ver la ejecución en un dispositivo real, **puede tardar valrios minutos**, dependiendo del tamaño de la fila de espera.

In [41]:
```
from qiskit.compiler import transpile


circuit = QuantumCircuit(3, 3)                         # creamos un circuito de 3 qubits y 3 registros clásicos
circuit.x(0)                                           # aplicamos algunas compuertas
circuit.x(1)
circuit.ccx(0, 1, 2)
circuit.cx(0, 1)
circuit.barrier()
circuit.measure([0, 1, 2], [0, 1, 2])                  # agregamos mediciones a los 3 qubits

display(circuit.draw('mpl'))                           # mostramos el circuito


circuit = transpile(circuit, least_busy_device)        # transpilamos el circuito para el backend en específico


display(circuit.draw('mpl'))                           # mostramos el circuito transpilado

job = least_busy_device.run(circuit)                   # mandamos ejecutar el circuito al backend en la nube
                                                       # guardado en la variable 'least_busy_device'
                                                       # esta línea de código puede tardarse hasta varios minutos
                                                       # dependiendo de la cantidad de 'jobs' encolados


result = job.result()                                  # la ejecución nos regresa un 'job'

counts = result.get_counts()                           # al que le podemos pedir los conteos

print(counts)                                          # imprimimos los conteos
```






```
{'000': 75, '001': 253, '010': 129, '011': 107, '100': 163, '101': 3116, '110': 69, '111': 88}
```

Podemos averiguar el estado del backend real con:

In [42]:
```
least_busy_device.status()                             # su estado actual
```

Out[42]:
```
<qiskit.providers.models.backendstatus.BackendStatus object at 0x00000205277E4C70>
```
name: ibmq_belem
version: 1.0.50, **pending jobs**: 22
status: active

7.B. Jobs

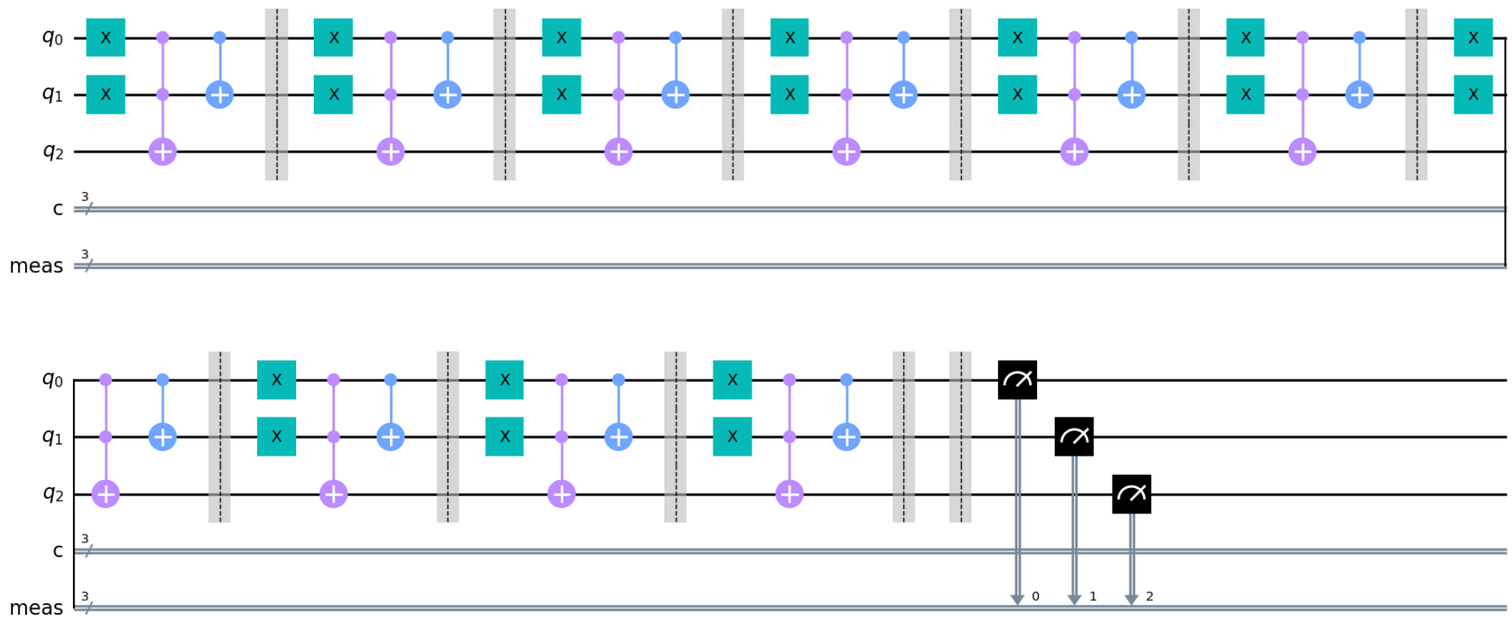
Qiskit provee una serie de funcionalidades para conocer qué esta sucediendo con los backends y con las ejecuciones, esto con la ayuda del objeto `job` .

Las instancias de `job` se pueden considerar como el "ticket" para una ejecución enviada. Con estos se puede averiguar el estado de ejecución en un momento dado (por ejemplo, si el trabajo está formado en la fila, en ejecución o ha fallado) y también permiten el control del mismo `job` .

Veamos un ejemplo:

In [43]:
```
# creamos un circuito para probar

circuit = QuantumCircuit(3, 3)
for i in range(10):
    circuit.x(0)
    circuit.x(1)
    circuit.ccx(0, 1, 2)
    circuit.cx(0, 1)
    circuit.barrier()

circuit.measure_all()

display(circuit.draw('mpl'))
```

```
display(circuit.draw('mpl'))
```



In [44]:

```
from qiskit.providers.jobstatus import JobStatus
from time import sleep
from IPython.display import clear_output


circuit = transpile(circuit, least_busy_device)              # transpilamos el circuito para el backend en específico

job = least_busy_device.run(circuit)                         # creamos el 'job' al solicitar la ejecución del circuito

print("Identificador del job:\t", job.job_id())              # obtenemos el identificador del 'job'

print("Job creado el:\t\t", job.creation_date())             # preguntamos la fecha de creacion del 'job'
print()

tt = 0                                                       # tiempo transcurrido (en segundos)

job_status = job.status()                                    # preguntamos por el estatus del 'job'

while job_status.name not in ["DONE", "CANCELLED", "ERROR"]:
    try:
        job_status = job.status()                            # preguntamos por el estatus del 'job' nuevamente
        if job_status is JobStatus.RUNNING:
            print("\rEl job sigue ejecutándose, tt = " + str(tt) + " s", end="")
            tt += 1
            sleep(1)
    except IBMApiError as ex:
        print("Algo malo sucedió!: {}".format(ex))


print()
print("\n\t>>> El job ha terminado su ejecución")
print("\n\n")

print("Resultado:\t", job.result().get_counts())              # resultado de la ejecución en el dispositivo real
```

```
Identificador del job:   634efc6192d4045cbdd93900
Job creado el:           2022-10-18 14:20:00.682000-05:00

El job sigue ejecutándose, tt = 8 s

        >>> El job ha terminado su ejecución


Resultado:       {'000 000': 418, '010 000': 638, '011 000': 295, '100 000': 519, '101 000': 313, '110 000': 1211, '111 000': 295, '0
01 000': 311}
```

8. Conclusiones

Esto ha sido una introducción a lo que se puede hacer con los circuitos cuánticos, en especial usando Qiskit, sin embago, este SDK es mucho más poderoso, por lo que se recomienda visitar los Tutoriales de Qiskit (los cuales están traducidos al Español), así como el Qiskit Textbook (próximante traducido al Español).

9. Referencias

- Qiskit Documentation. Consultado en https://qiskit.org/documentation/
- Qiskit Textbook. Consultado en https://qiskit.org/learn/
- Qiskit Tutorials. Consultado en https://qiskit.org/documentation/tutorials.html
- QWold's Bronze tutorial. Consultado en https://gitlab.com/qworld/bronze-qiskit
- QWold's Silver tutorial. Consultado en https://gitlab.com/qworld/silver
- Nielsen, M., & Chuang, I. (2010). Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511976667
- Kaye P., Laflamme R., & Mosca M. (2007). An Introduction to Quantum Computing. Oxford: Oxford University Press. ISBN: 9780198570493

Donaciones

Puedes donar una vez en el siguiente enlace (Ko-Fi):

Click en la imagen. Buy me a coffee