Practica2_Memoria

March 28, 2021

Mario Quiñones Pérez

1 Teleportación Cuántica

1.1 Creación del circuito y simulacion en condiciones ideales

Este circuito esta diseñado para pasar una informacion quantica concreta de un lado a otro. Por ejemplo madar un qubit |psi> = a|0>+B|1>, lo que significaria pasar información sobre a y B.

Como existe el teorema de que no se puede copiarla información exacta de un estado cuantico desconocido, llamado teorema de la no clonación, no podemos simplemente generar una copia de un qubit y mandarla, solo de un estado clásico y no de superposiciones.

Entonces usaremos dos bits y un par entrelazado de qubits para pasar el estado, lo que llamamos teleportación cuantica. Recive este nombre ya que al final qubit inicial ,llamemosle q0 pasará a no tener el estado |psi> y a su vez tendra o estado |0> o |1> y el qubit final o al que queremos pasarle el estado acabará valiendo |psi>

1.1.1 Quiskit

En este apartado se expone como implementar un circuito de teleportacion cuantica en qiskit y a partir de el poder enviar un qubit de q0 a q2 ayudandonos de q1.

[1]:
```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.extensions import Initialize
from qiskit_textbook.tools import random_state, array_to_latex
```

En esta primera parte creamos los diferentes qubits y bits que utilizaremos a lo largo del circuito y daremos nombre al circuito que vamos a crear

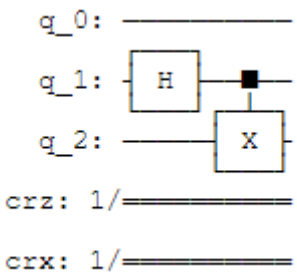
[2]:
```
qr = QuantumRegister(3, name="q")
crz = ClassicalRegister(1, name="crz")
crx = ClassicalRegister(1, name="crx")
teleportation_circuit = QuantumCircuit(qr, crz, crx)
```

Esta funcion sera usada par crear un par de bell el cual se utilizara para entrelazar dos qubits

[3]:
```
def create_bell_pair(qc, a, b):
    qc.h(a) # Aplicamos una Hadamard a 'a'
    qc.cx(a,b) # Aplicamos una cnot a 'b'
               # usando 'a' como qubit de control
```

Y entrelazamos q1 y q2 con la funcion anterior.

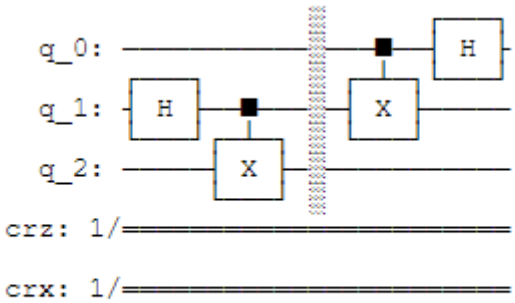
[4]:
```
create_bell_pair(teleportation_circuit, 1, 2)
```



Esta funcion sirve para definir lo que el qubit q0 hará para prepararse para el envio realizando una especie de par de bell invertido con q2

[5]:
```
def alice_gates(qc, psi, a):
    qc.cx(psi, a)
    qc.h(psi)
```

[6]:
```
teleportation_circuit.barrier()
alice_gates(teleportation_circuit, 0, 1)
```

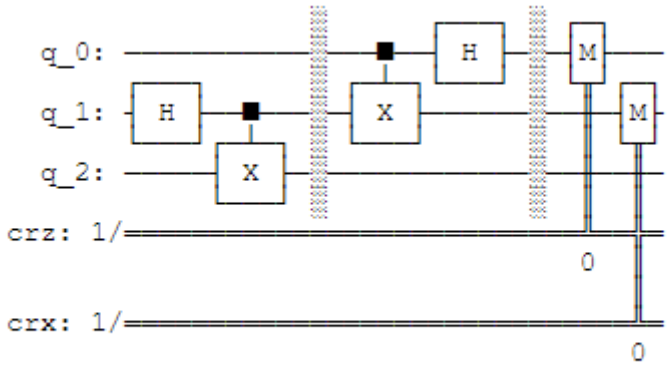


Se miden los resultados de q0 y q1 y se envian para que q2 los trate para poder luego hacer los cambios necesarios para obtener el qubit inicial en q0.

[7]:
```
def measure_and_send(qc, a, b):
    qc.barrier()
    qc.measure(a,0)
```

```
    qc.measure(b,1)
```

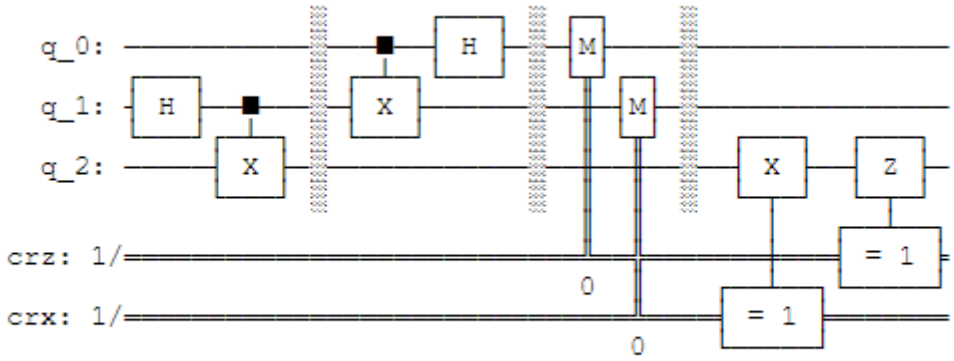
[8]:
```
measure_and_send(teleportation_circuit, 0, 1)
```



El qubit q2 despues de obtener la medicion de los dos qubits realiza una serie de puertas segun los valores de los bits que recive. En concreto si el segundo bit es un uno realiza una puerta X y si el primer bit recivido es un uno realiza una puerta Z sobre q2

[9]:
```
def bob_gates(qc, qubit, crz, crx):
    qc.x(qubit).c_if(crx, 1)
    qc.z(qubit).c_if(crz, 1)
```

[10]:
```
teleportation_circuit.barrier()
bob_gates(teleportation_circuit, 2, crz, crx)
```

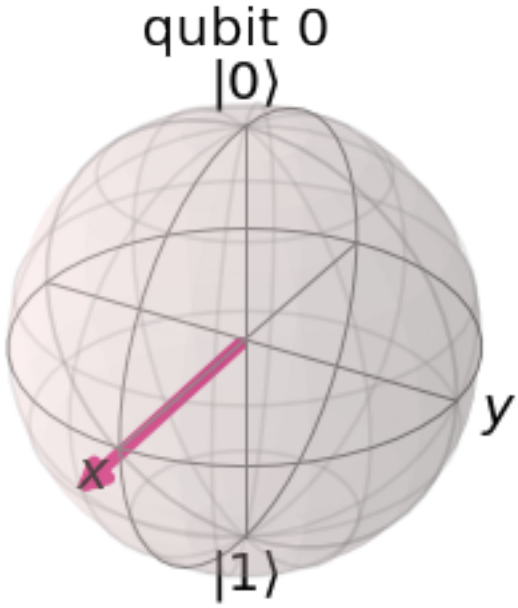


Creamos un qubit aleatorio que llamaremos psy y mostramos su esfera de bloch para poder comprobar si el circuito realiza bien la teleportacion cuantica

[11]:
```
psi = random_state(1)

array_to_latex(psi, pretext="|\\psi\\rangle =")

plot_bloch_multivector(psi)
```

$$|\psi\rangle = \begin{bmatrix} 0.04779 + 0.50948i \\ 0.44452 + 0.73522i \end{bmatrix}$$

[11]:



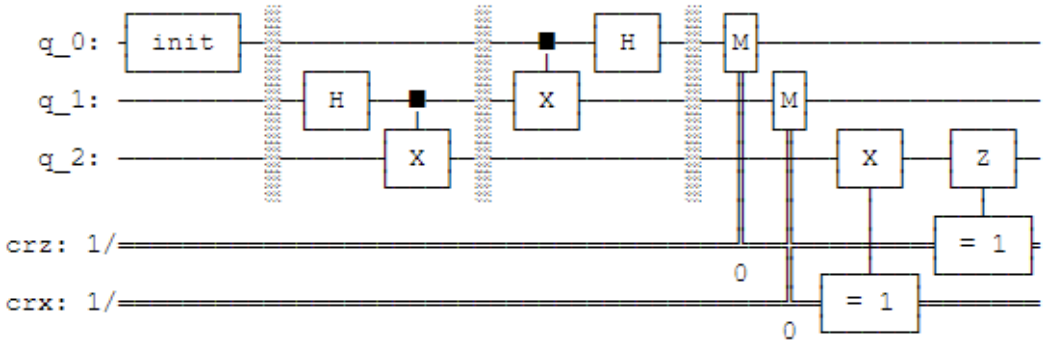
Creamos una puerta init que realiza la inicializacion de qbit a el valor aleatorio que hemos creado anteriormente

[12]:
```
init_gate = Initialize(psi)
init_gate.label = "init"
```

Creamos el nuevo circuito añadiendo la puerta de inicialización

[13]:
```
qr = QuantumRegister(3, name="q")
crz = ClassicalRegister(1, name="crz")
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)


# Inicializamos el qubit q0
qc.append(init_gate, [0])
qc.barrier()
```

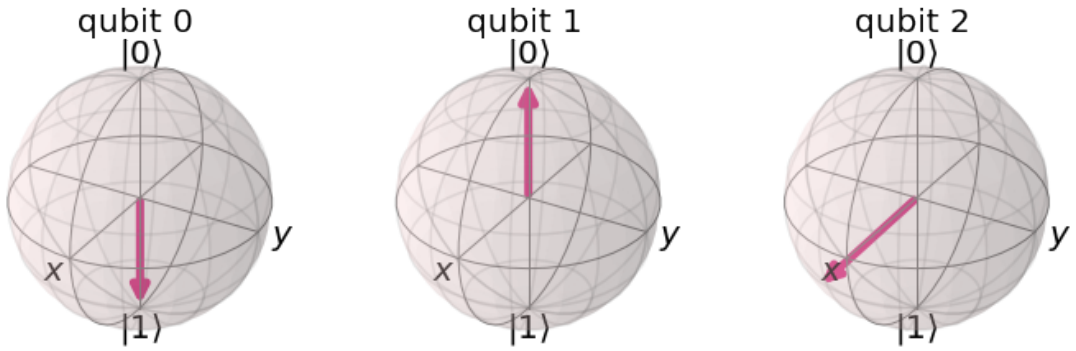
```
# Entrelazamos q1 y q2
create_bell_pair(qc, 1, 2)
qc.barrier()

# Enviamos q1 al emisor y q2 al receptor de el estado inicial aleatorio
alice_gates(qc, 0, 1)


# El emisor envia sus bits clasicos obtenidos de q0 y q1 al receptor
measure_and_send(qc, 0, 1)

# Se decodifics el mensaje en el receptor
bob_gates(qc, 2, crz, crx)
```



Simulamos el circuito y comprovamos que el valor de q2 es identico al valor inicila aleatorio de q0 y que q1 vuelcan en |0> o |1>. Hemos comprovado que el circuito a transportado el qubit psi de q0 a q2

[16]:
```
sv_sim = Aer.get_backend('statevector_simulator')
qobj = assemble(qc)
out_vector = sv_sim.run(qobj).result().get_statevector()
plot_bloch_multivector(out_vector)
```

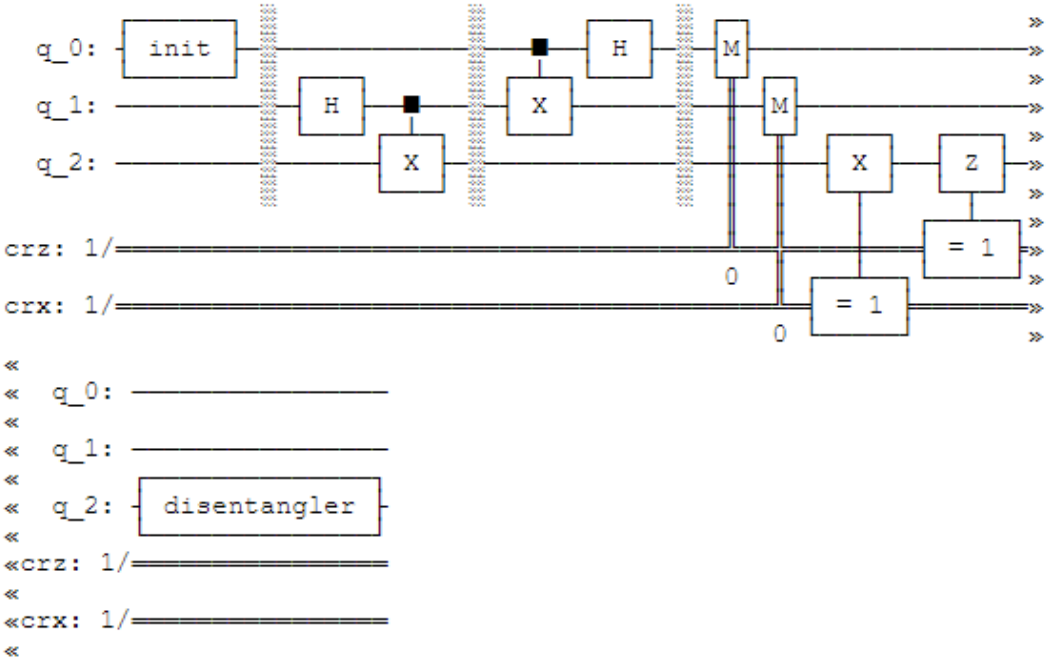
[16]:



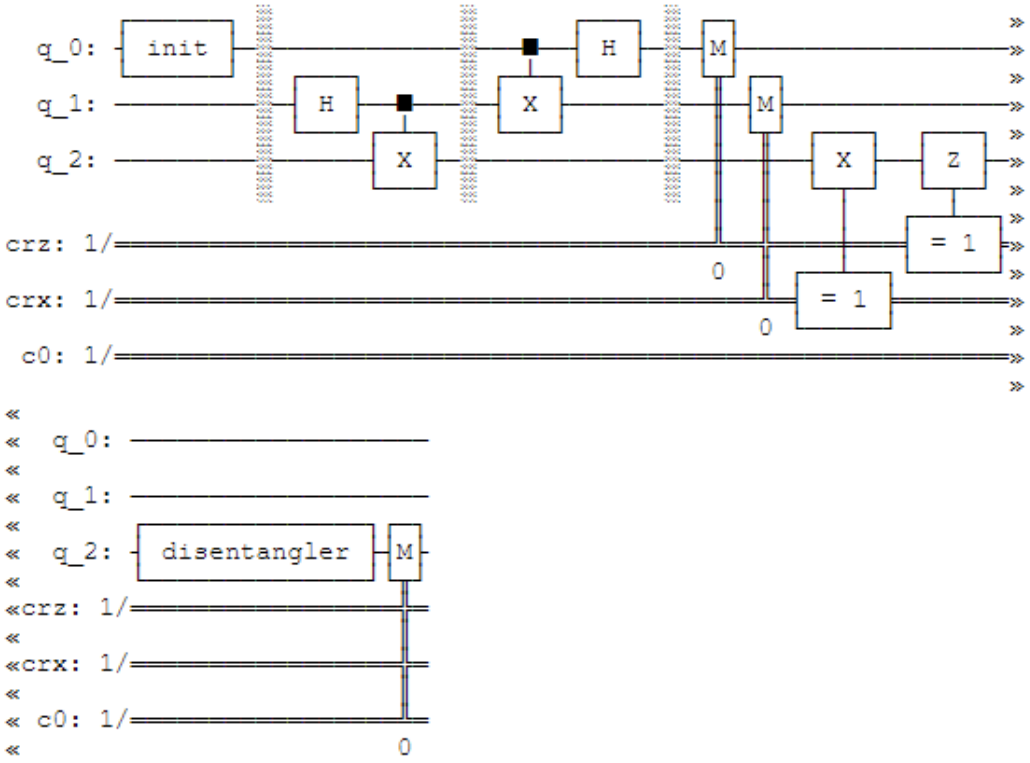
Creamos la puerta inversa de la inicializadora

[17]:
```
inverse_init_gate = init_gate.gates_to_uncompute()
```

[18]:
```
qc.append(inverse_init_gate, [2])
```

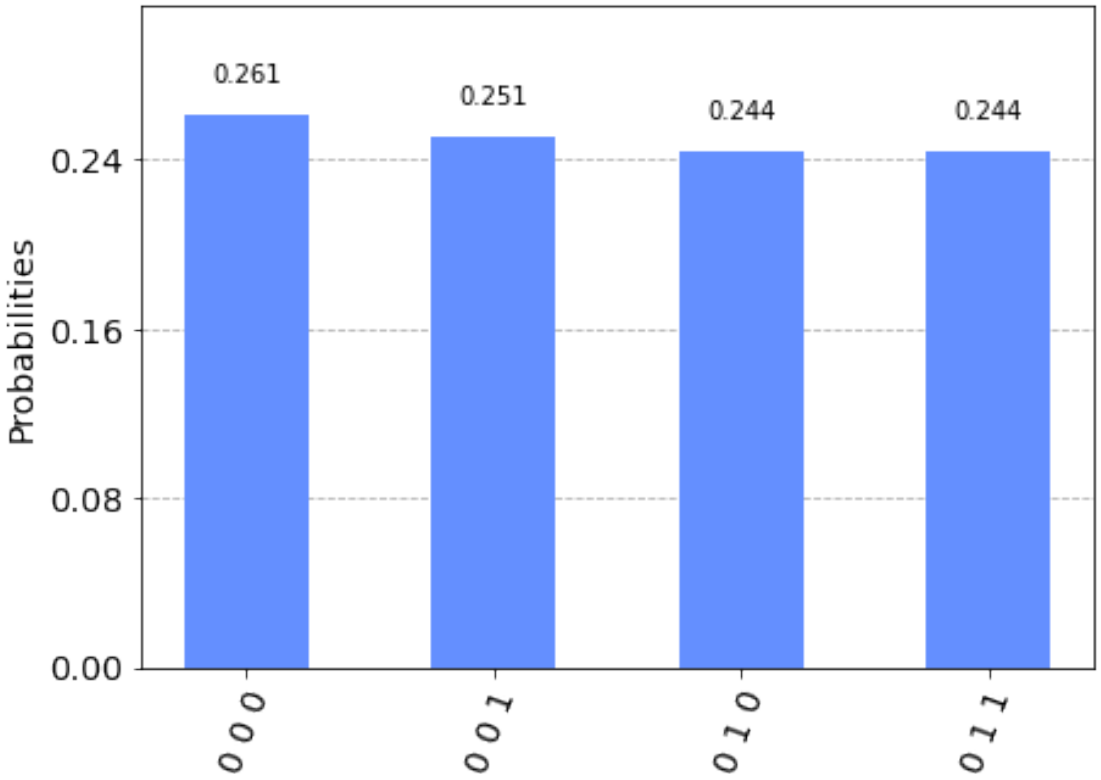


[19]:
```
cr_result = ClassicalRegister(1)
qc.add_register(cr_result)
qc.measure(2,2)
```



[20]:
```
qasm_sim = Aer.get_backend('qasm_simulator')
t_qc = transpile(qc, qasm_sim)
qobj = assemble(t_qc)
counts = qasm_sim.run(qobj).result().get_counts()
plot_histogram(counts)
```

[20]:



1.1.2 Cirq

En este apartado aparece como se puede construir un circuito en cirq que sirve para provar la teleportación cuantica, en la que pasamos un qubit de q0 a q2 utilizando q1 de mediador.

[ ]:
```
import cirq
from cirq import Simulator
```

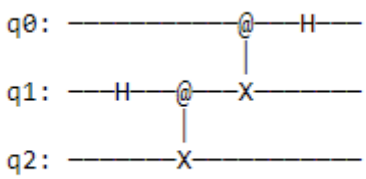
Añadimos el entrelazamiento entre los qubits q1 y q2

[82]:
```
q0 = cirq.NamedQubit('q0')
q1 = cirq.NamedQubit('q1')
q2 = cirq.NamedQubit('q2')

circ = cirq.Circuit(cirq.H(q1), cirq.CNOT(q1, q2))
```

```
q1: ---H---@---
           |
q2: -------X---
```

Añadimos las puertas que preparan los qubits q1 y q0 para que q2 los utilicen para crear de nuevo el estado |psi> original

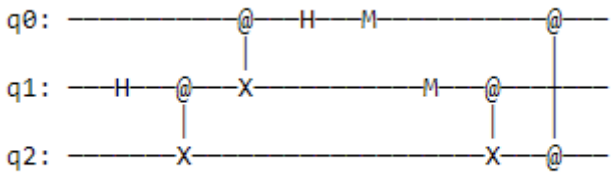
[83]:
```
circ.append(cirq.Moment([cirq.CNOT(q0, q1)]))
circ.append(cirq.H(q0))
```



Se añaden las puertas que utilizando los valores de q0 y q1 transforman q2 en el estado original que se queria enviar

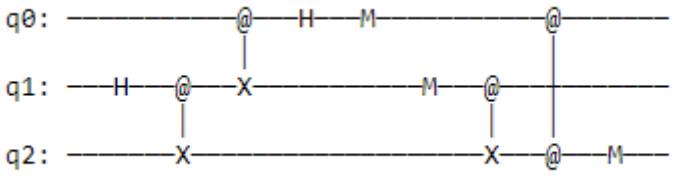
[85]:
```
def bob_gates(qc, a, b, c):
    qc.append(cirq.CNOT(b, c))
    qc.append(cirq.CZ(a,c))
```

[86]:
```
bob_gates(circ, q0, q1, q2)
```



Se mide el ultimo qubit para comprovar su valor y comprovar el correcto envio

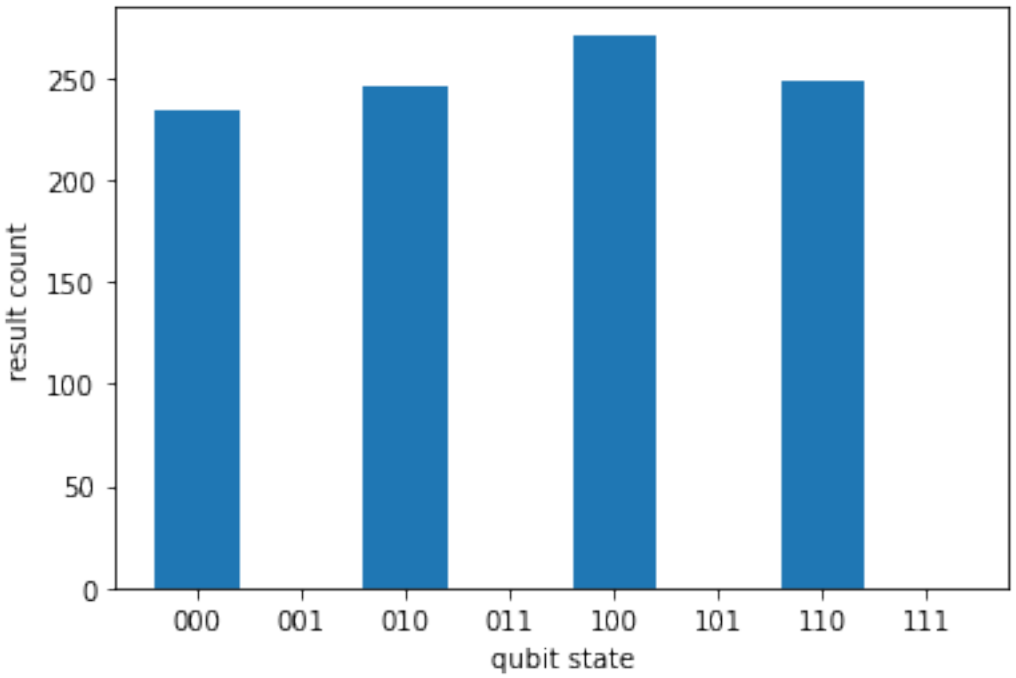
[87]:
```
circ.append(cirq.Moment([cirq.measure(q2)]))
```



Se hace la simulacion 1000 veces del circuito para comprovar resultados

[88]:
```
from cirq import Simulator
simulator = Simulator()
result = simulator.run(circ, repetitions=1000)
print('resultados para q2: ',result.histogram(key='q2'))
counts = cirq.plot_state_histogram(result)
```

```
resultados para q2:  Counter({0: 1000})
```



Ya que los qubits se inicializan a |0> el estado final del qubit q2 siempre es 0 como podemos comprobar en los resultados para q2.

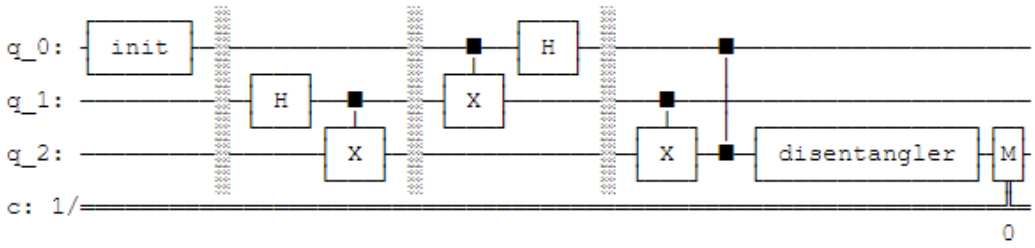
Ademas podemos observar en el histograma que el valor de q2 representados por el tercer bit siempre es cero, aunque q1 y q2 pueden variar, pero estos valores no son de vital importancia.

1.2 Simulacion en condiciones no ideales

Como en computadores cuanticos reales solo se puede medir al final del cicuito entonces tenemos que cambiar ligeramente el circuito para que las puertas que se realizaran a q2 no dependan de las mediciones en bits clasicos de q0 y q1 sino de q0 y q1 como tales. Para eso la puerta X dependera de q1 en vez de su conversion en bit y la Z dependera del qubit q0 como tal tambien.

[21]:
```
def new_bob_gates(qc, a, b, c):
    qc.cx(b, c)
    qc.cz(a, c)
```

[22]:
```
qc = QuantumCircuit(3,1)

qc.append(init_gate, [0])
qc.barrier()

create_bell_pair(qc, 1, 2)
qc.barrier()

alice_gates(qc, 0, 1)
qc.barrier()


# En vez de enviar sus dos bits clasicos, el
# receptor utiliza los qubits q0 y q1 directamente.
new_bob_gates(qc, 0, 1, 2)


qc.append(inverse_init_gate, [2])

#vemos el estado de q2 que es el que nos interesa
qc.measure(2,0)
```



1.2.1 Ejecucion con modelos de ruido

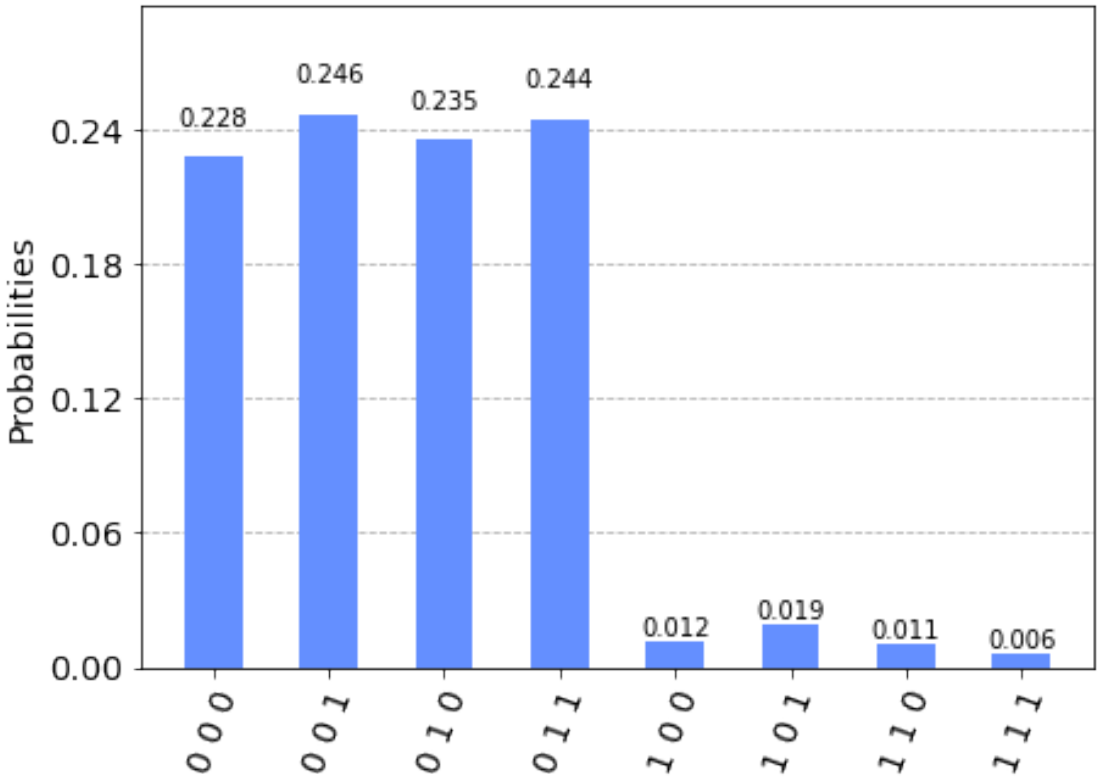
Utilizamos es mismo circuito que vamos utilizar para backends reales para que los resultados tengan una mayor realicion entre ellos y puedan ser comentadas sus similitudes.

[23]:
```
from qiskit import QuantumCircuit, execute
from qiskit import IBMQ, Aer
from qiskit.visualization import plot_histogram
from qiskit.providers.aer.noise import NoiseModel
from qiskit.compiler import transpile, assemble, schedule
IBMQ.save_account('[redacted]
                  '[redacted]
                  '[redacted]
                  '[redacted]', overwrite=True)
provider = IBMQ.load_account()
```

[24]:
```
# Construir un modelo de ruido a partir de las características de un backend␣
 ↪real
backend = provider.get_backend('ibmq_belem')
noise_model = NoiseModel.from_backend(backend)

# Obtener el mapa de interconexión de los qubits
coupling_map = backend.configuration().coupling_map

# Obtener las características de las puertas básicas
basis_gates = noise_model.basis_gates

######################
# Crear circuito     #
######################

# Perform a noise simulation
result = execute(t_qc, Aer.get_backend('qasm_simulator'),
                 coupling_map=coupling_map,
                 basis_gates=basis_gates,
                 noise_model=noise_model).result()

######################
# Mostrar resultados #
######################

counts = result.get_counts(t_qc)
print("\nNúmero de ocurrencias:",counts)
plot_histogram(counts)
```

```
Número de ocurrencias: {'0 0 0': 233, '0 0 1': 252, '0 1 0': 241, '0 1 1': 250,
'1 0 0': 12, '1 0 1': 19, '1 1 0': 11, '1 1 1': 6}
```

[24]:



1.2.2 Ejecucion en backends reales

Mandamos a ejecutar el nuevo circuito a un ordenador cuantico real y comprabamos los resultados

[82]:
```
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
```

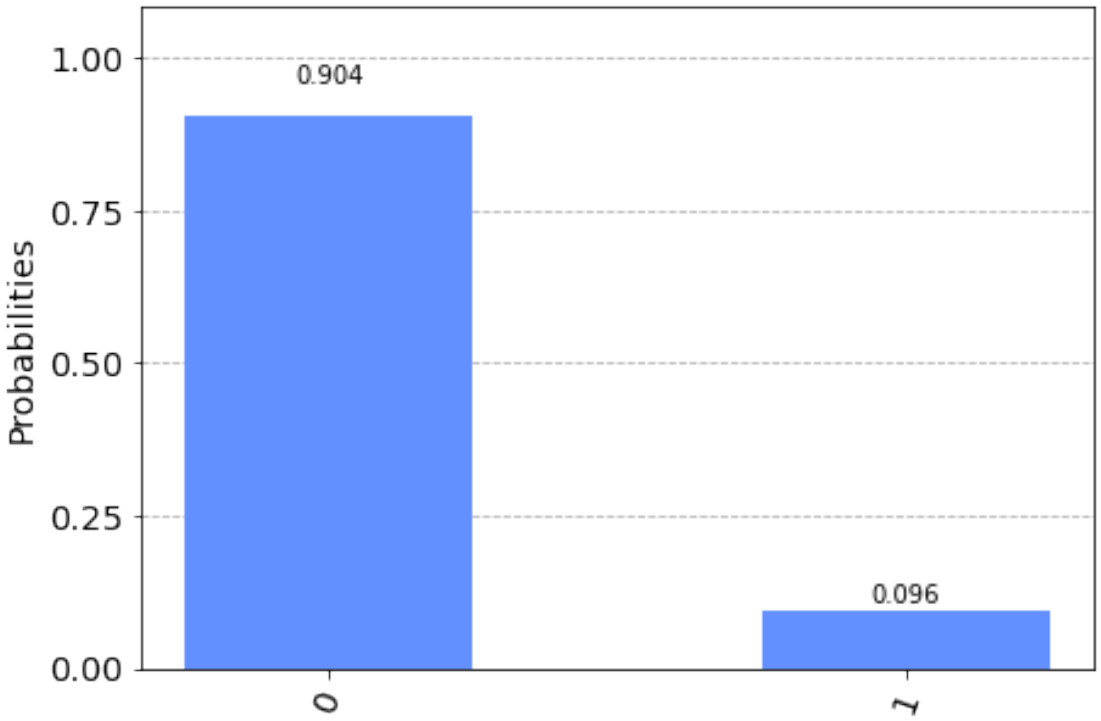
[39]:
```
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
backend = least_busy(provider.backends
                     (filters=lambda b:
                      b.configuration().n_qubits >= 3 and
                      not b.configuration().simulator and
                      b.status().operational==True))
t_qc = transpile(qc, backend, optimization_level=3)
qobj = assemble(t_qc)
job = backend.run(qobj)
job_monitor(job)
```

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

[84]:
```
exp_result = job.result()
exp_counts = exp_result.get_counts(qc)
print(exp_counts)
plot_histogram(exp_counts)
```

```
{'0': 926, '1': 98}
```

[84]:



[25]:
```
print(f"El error experimental es de:
      {(exp_counts['1']/sum(exp_counts.values())*100):.3f}%")
```

```
El error experimental es de : 9.6%
```

Vemos que aunque el error no es nulo la tasa de fallos del experimento en menor del 10% que aunq para ordenadores que deberian trabajar con miles de operaciones aun es mucho es bastante impresionante.

1.2.3 Conclusiones

Como ya vimos en la practica anterior los resultados no son siempre los esperados debido a la existencia de ruido por lo que no siempre funcionara el circuito aunque como vemos en el ultimo apartado de una ejecucion en backend real esta tasa de fallo es muy pequeña.

En cuanto a las diferencias entre la simulacion con ruido y la ejecucion en un ordenador cuantico podemos observar que en el backend real existe una probabilidad mas grande de fallo ya que la suma de las probabilidades de los

estados fallidos es 0.052 en la simulación y en el backend real es de 0.096 , que es casi el doble del medido en la simulación, aunque esto se puede deber solo al azar.

2 Codificación superdensa

2.1 Diferencias entre Teleportación cuántica y Codificación superdensa

Teleportación cuántica es el proceso por el cual un estado (|psi>) puede ser transmitido de un lugar a otro usando dos bits clasicos y un par Bell. La codificación superdensa trata de un proceso que permite enviar dos bits clasicos atraves de un solo qubit de comunicación.

2.1.1 Quiskit

En este apartado se expone como implementar un circuito de codificación superdensa en qiskit y a partir de el poder enviar dos bits cualesquiera ayudandonos de dicho circuito.

[11]:
```
from qiskit import QuantumCircuit
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram
```

Creamos el par de Bell para entrelazar los qubits entre si

[2]:
```
def create_bell_pair(qc, a, b):
    qc.h(a) # Aplicamos una Hadamard a 'a'
    qc.cx(a,b) # Aplicamos una cnot a 'b'
               # usando 'a' como qubit de control
```

Esta funcion pasa recive el mensaje que se quiere enviar y aplica las puertas necesarias al qubit dado (que en este caso sera el qubit al que se aplica le hadamard en el par de Bell al inicio del circuito) para que este pueda transmitir el mensaje de dos bits.

[3]:
```
def encode_message(qc, qubit, msg):
    if msg == "00":
        pass    # Nada para mandar 00
    elif msg == "10":
        qc.x(qubit) # Puerta X para mandar 10
    elif msg == "01":
        qc.z(qubit) # Puerta Z para mandar 01
    elif msg == "11":
        qc.z(qubit) # Puerta Z para mandar 11
        qc.x(qubit) # sequida de una puerta X
    else:
        print("Invalid Message: Sending '00'")
```

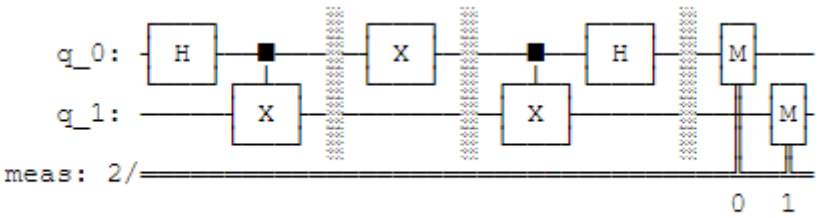
Esta función sera la encargada de una vez llegado el qubit que contiene la información de ambos bits enviados y utilizando el segundo qubit del par de bell

entrelazado al principio de circuito, recrear el mensaje de dos bits clasicos que se querian enviar.

[4]:
```
def decode_message(qc, a, b):
    qc.cx(a,b)
    qc.h(a)
```

Creamos el circuito usando todas las funciones anteriores

[5]:
```
qc = QuantumCircuit(2)


# Primero, creamos un par entrelazado de qubits
create_bell_pair(qc, 0, 1)
qc.barrier()

# Uno va a emisor y otro al receptor de los dos bits clasicos

# Ahora codificamos el mensaje en q0, en este caso hemos utilizado el
# mensaje '10' , lo que nos es indiferente ya que cualquier mensaje
# de dos podría ser mandado
message = "10"
encode_message(qc, 0, message)
qc.barrier()
# Enviamos el qubit al receptor

# Decodificamos el qubit que nos han
# mandado con la información del mensaje
decode_message(qc, 0, 1)

qc.measure_all()
```



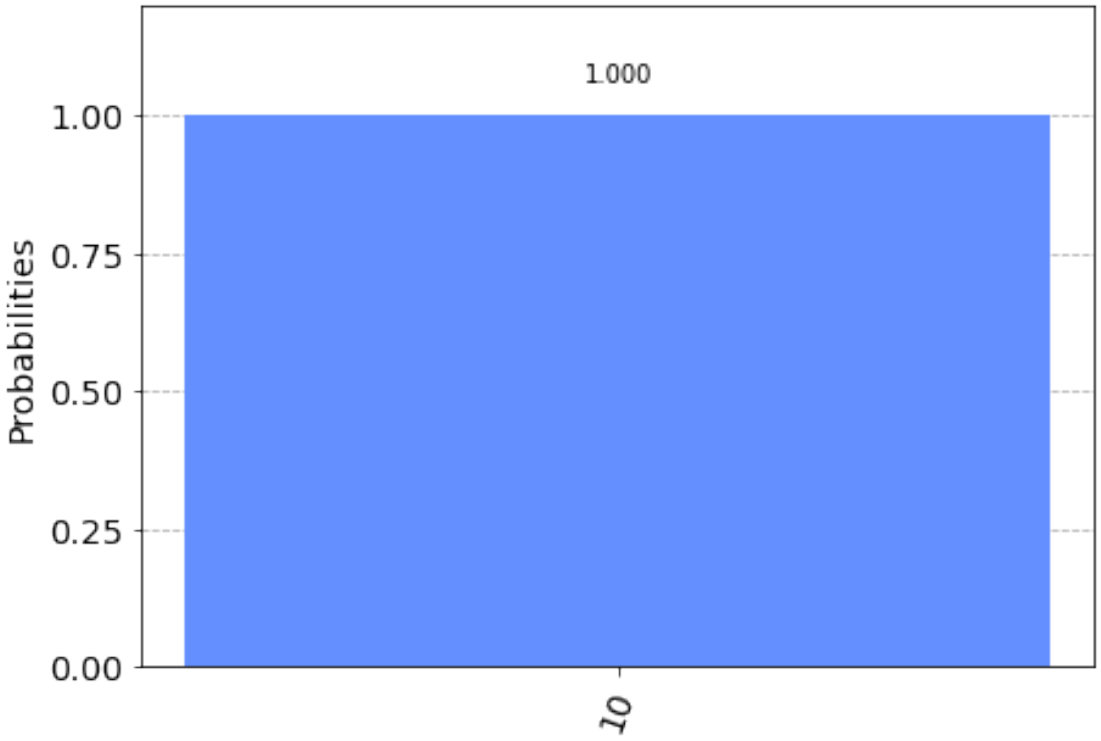
Como podemos observar en la siguiente simulacion, los datos que nos salen al medir son siempre iguales al mensaje enviado, y esto pasa siempre ya que lo estamos midiendo en condiciones ideales

[6]:
```
qasm_sim = Aer.get_backend('qasm_simulator')
qobj = assemble(qc)
result = qasm_sim.run(qobj).result()
```

```
counts = result.get_counts(qc)
print(counts)
plot_histogram(counts)
```

```
{'10': 1024}
```

[6]:



2.1.2 Cirq

En este apartado aparece como se puede construir un circuito en cirq que sirve para utilizar codificación superdensa, en la que pasamos un dos bits clasicos de un emisor a un receptor utilizando unicamente un qubit, para lo que necitaremos ademas un qubit adicional que este entrelazado con este primero para poder decodificar el mensaje en el receptor.

[21]:
```
import cirq
from cirq import Simulator
```

Creamos una función que codifique el mensaje de dos bits en un solo qubit a parti de aplicar puertas X o Z

[22]:
```
def encode_message(qc, qubit, msg):
    if msg == "00":
        pass    # Nada para mandar 00
    elif msg == "10":
```

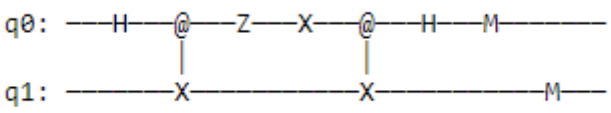
```
        qc.append(cirq.Moment([cirq.X(qubit)]))
        # Puerta X para mandar 10
    elif msg == "01":
        qc.append(cirq.Moment([cirq.Z(qubit)]))
        # Puerta Z para mandar 01
    elif msg == "11":
        qc.append(cirq.Moment([cirq.Z(qubit)]))
        # Puerta Z para mandar 11
        qc.append(cirq.Moment([cirq.X(qubit)]))
        # sequida de una puerta X
    else:
        print("Invalid Message: Sending '00'")
```

Y cremos una funcion que decodifique dicho mensaje que nos han mandado con un solo qubit

[23]:
```
def decode_message(qc, a, b):
    circ.append(cirq.CNOT(a, b))
    qc.append(cirq.H(a))
```

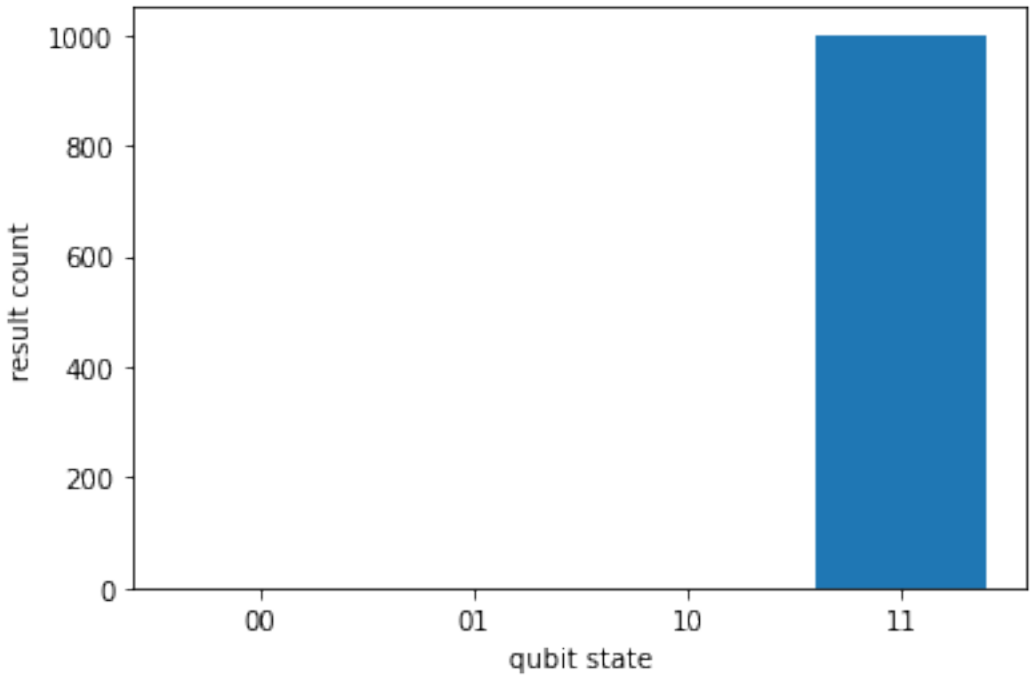
Cremos el circuito que se encargara de enviar mensajes de dos bits atraves de un solo qubit

[24]:
```
q0 = cirq.NamedQubit('q0')
q1 = cirq.NamedQubit('q1')
q2 = cirq.NamedQubit('q2')

# Añadimos el entrelazamiento entre los qubits q0 y q1
circ = cirq.Circuit(cirq.H(q0), cirq.CNOT(q0, q1))

# Utilizamos como mensaje 11 y lo codificamos usandos la funcion creada
message = "11"
encode_message(circ, q0, message)

# El receptor recive q0 y a traves de este y q1 que esta
# entrelazado con q0 desde antes de la codificación
# decodifica el mensaje
decode_message(circ, q0, q1)

# Se miden los resultados
circ.append(cirq.measure(q0))
circ.append(cirq.Moment([cirq.measure(q1)]))
```



Tal y como vemos en la simulación la unica salida en las 1000 iteraciones es `11' que es el mensaje que queriamos transmitir, por tanto en condiciones ideales se demuestra que el circuito funciona tal y como debería.

[25]:
```
simulator = Simulator()
result = simulator.run(circ, repetitions=1000)
counts = cirq.plot_state_histogram(result)
```



2.2 Simulacion en condiciones no ideales

2.2.1 Ejecucion con modelos de ruido

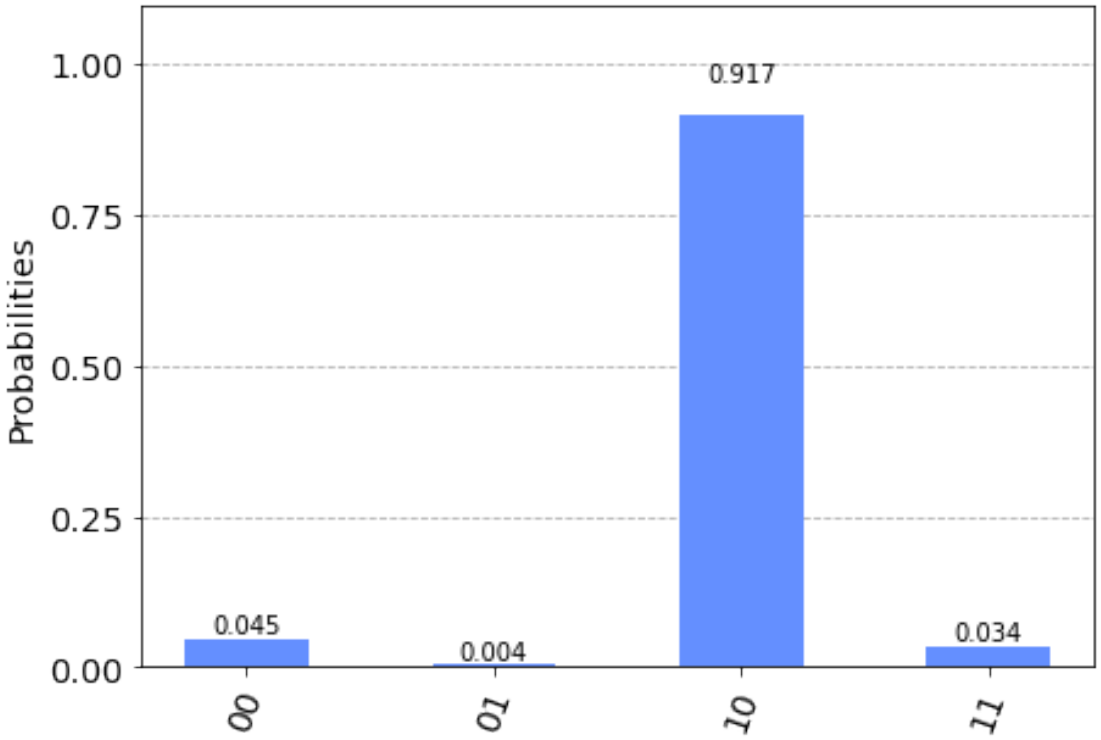
[7]:
```
from qiskit import QuantumCircuit, execute
from qiskit import IBMQ, Aer
from qiskit.visualization import plot_histogram
from qiskit.providers.aer.noise import NoiseModel
from qiskit.compiler import transpile, assemble, schedule
IBMQ.save_account('55dbb1b5e08af7c7c6ec803a2770842c2f2e9b1
                  '6b557e0a3bf6e5025a41316a162a165590114633
                  '1db9eaf4861ecc9cd69479420d0cb77bafcea5f4
                  '3b10c18e3', overwrite=True)
provider = IBMQ.load_account()
```

[10]:
```
# Construir un modelo de ruido a partir de las características de un backend␣
 ↪real
```

```
backend = provider.get_backend('ibmq_belem')
noise_model = NoiseModel.from_backend(backend)

# Obtener el mapa de interconexión de los qubits
coupling_map = backend.configuration().coupling_map

# Obtener las características de las puertas básicas
basis_gates = noise_model.basis_gates

######################
# Crear circuito     #
######################

# Perform a noise simulation
result = execute(qc, Aer.get_backend('qasm_simulator'),
                 coupling_map=coupling_map,
                 basis_gates=basis_gates,
                 noise_model=noise_model).result()

######################
# Mostrar resultados #
######################

counts = result.get_counts(qc)
print("\nNúmero de ocurrencias:",counts)
plot_histogram(counts)
```

```
Número de ocurrencias: {'00': 46, '01': 4, '10': 939, '11': 35}
```

[10]:



Como podemos ver aunque `10' (el mensaje que queremos enviar) no es siempre la salida, la probabilidad de que ocurra es lo suficientemente mayor como para considerarse un exito, teniendo un fallo de solo el 8,5%.

2.2.2 Ejecucion en backends reales

[14]:
```
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy
shots = 1024

IBMQ.load_account()
# Get the least busy backend
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends
                     (filters=lambda x:
                      x.configuration().n_qubits >= 2
                      and not x.configuration().simulator
                      and x.status().operational==True))
print("least busy backend: ", backend)

t_qc = transpile(qc, backend, optimization_level=3)
qobj = assemble(t_qc)
job = backend.run(qobj)
```

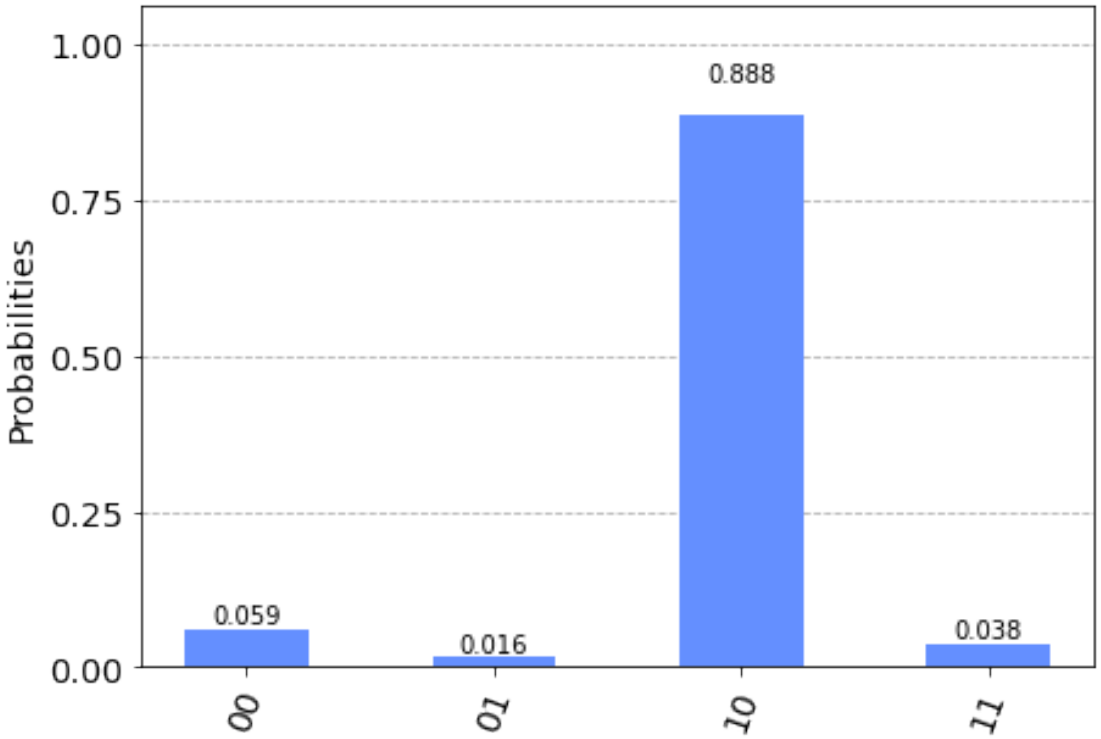
```
least busy backend:  ibmq_belem
```

[18]:
```
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

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

[19]:
```
result = job.result()
plot_histogram(result.get_counts(qc))
```

[19]:



[20]:
```
correct_results = result.get_counts(qc)[message]
accuracy = (correct_results/shots)*100
print(f"Accuracy = {accuracy:.2f}%")
```

```
Accuracy = 88.77%
```

Vemos que el error no es nulo, la tasa de fallos del experimento es de un 11,23% que aunq para ordenadores que deberian trabajar con miles de operaciones aun es mucho es suficiente como para considerarlo un exito.

2.2.3 Conclusiones

Como ya vimos en la practica anterior los resultados no son siempre los esperados debido a la existencia de ruido por lo que no siempre funcionara el circuito,

lo que podemos ver en el ultimo apartado de una ejecucion en backend real ya que esta tasa de fallo supera el 10%.

En cuanto a las diferencias entre la simulacion con ruido y la ejecucion en un ordenador cuantico podemos observar que en el backend real existe una probabilidad mas grande de fallo ya que la suma de las probabilidades de los estados fallidos es 0.085 en la simulación y en el backend real es de 0.1123, que aun no siendo el doble como ocurria en la teleportación cuantica, esta tasa de fallos es muy grande para un ordenador.

3 Diferencias entre Qiskit y Cirq

Entre qiskit y cirq se pueden observar una serie de diferencias, aunque estas no afectan al resultado del cirquito.

Estas diferencias son principalmente la manera de añadir puertas con append en cirq y utilizando la propia funcion .'puerta' en qiskit (por ejemplo .h(qubit) aplica la hadamard a qubit).

Otra diferencia significativa son las barreras , mientras que en qiskit existe una funcion que crea una barrera que separa y ordena el circuito de forma sencilla, en cirq estas barreras se tienen que hacer de manera manual utilizando la funcion cirq.Moment dentro de los appends lo que complica un poco mas el codigo y la legibilidad del mismo.

Por ultimo , existe una diferencia leve entre ambos debido a sus representaciones graficas de tanto circuitos como graficas. En qiskit los circuitos estan mejor representados y se puede dibujar hasta los bits clasicos mientras que en cirq solo se puede representar los qubits utilizados y la puertas de manera menos visual. Ademas en las graficas de los histogramas se puede ver que los bits que representan cada salida están invertidos uno respecto al otro (en qiskit sales en el orden q2,q1,q0 y en cirq aparecen como q0,q1,q2)