PANDIT DEENDAYAL ENERGY UNIVERSITY

Raisan, Gandhinagar-382 426, Gujarat, India

Fundamentals of Quantum Computing Lab

Laboratory Manual

20IC407P

Submitted To:
Dr. Manish Mandloi,
Assistant Professor,
Department of Information and Communication Technology,
PDEU, Gandhinagar, Gujarat.

Submitted By:
Name: Janakar Patel
Roll Number: 20BIT061
Semester: 6th Semester
Department: ICT

INDEX

Experiment 1

Aim

Installation of Anaconda3 and Qiskit, and linking IBMQ account to Qiskit.

Theory

Qiskit:
Qiskit is an open-source quantum computing framework developed by IBM that provides a user-friendly interface for developing and executing quantum algorithms. It allows users to create quantum circuits using Python, and provides access to a variety of quantum algorithms and tools. Qiskit also allows users to simulate quantum circuits on classical computers, and to execute quantum circuits on real quantum devices through the IBMQ platform.

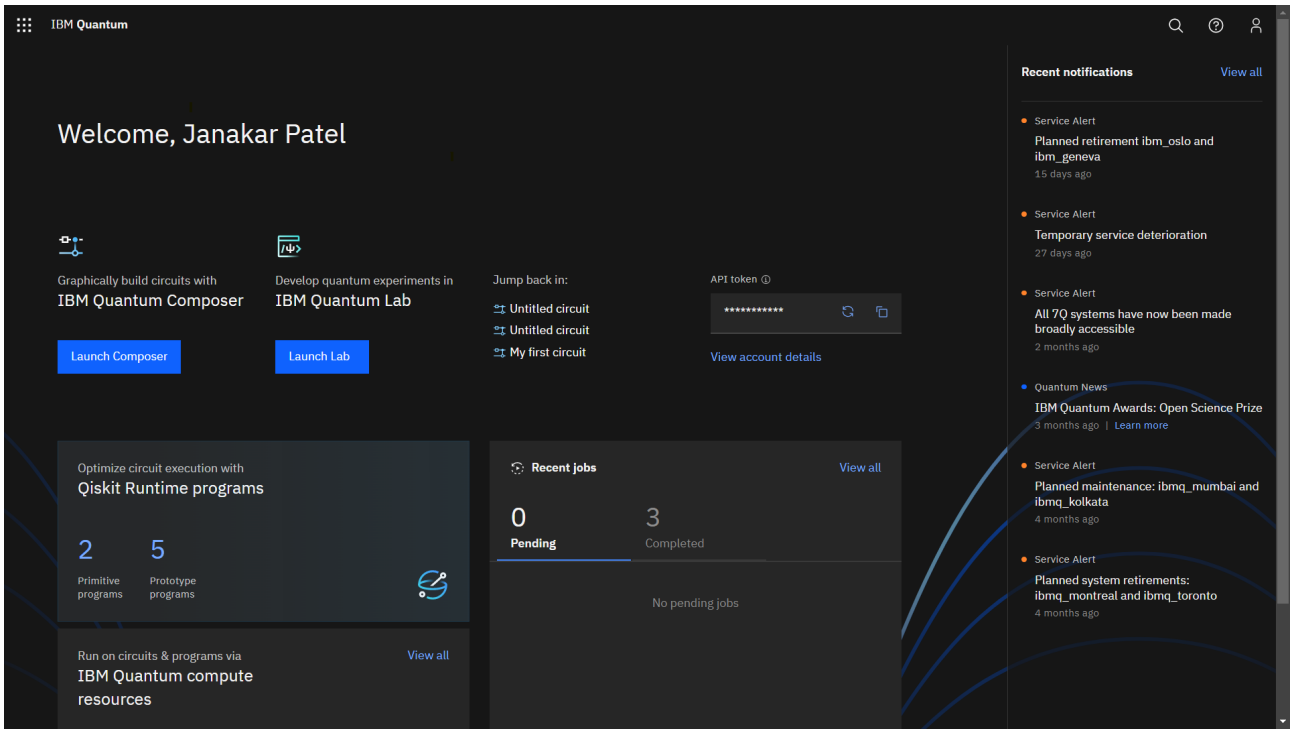
IBM Quantum:
The IBMQ platform provides access to a network of quantum devices and simulators that can be used to execute quantum circuits developed in Qiskit. The devices available on the IBMQ platform range from small-scale devices with a few qubits to larger-scale devices with dozens of qubits. The availability of real quantum devices on the IBMQ platform allows users to test and experiment with real quantum computing hardware, which is essential for advancing the field of quantum computing.

Anaconda:
Anaconda3 is a data science platform that provides an easy way to manage Python packages and environments, and is necessary to create a dedicated environment for running Qiskit. By following the installation instructions, users can install Anaconda3, create a new environment for Qiskit, and install Qiskit into the environment.

Lab Exercise

Installation Steps:

1. First install Python 3 on the machine from `https://www.python.org/downloads/`
2. Go to the Anaconda website `https://www.anaconda.com/products/individual` and download the Anaconda3 installer for your operating system. Follow the instructions to install Anaconda3 on your computer.
3. Make an account on IBMQ from `https://www.ibm.com/account/in/en/` and Login to this account.



4. Next type “pip install qiskit” in Anaconda prompt to install qiskit.
5. Next type “jupyter notebook” in Anaconda prompt and run following code to link your IBMQ account to Jupyter notebook.
6. Do the following code steps in jupyter notebook

```
In [2]: import qiskit

In [3]: qiskit.__qiskit_version__

Out[3]: {'qiskit-terra': '0.22.4', 'qiskit-aer': '0.11.2', 'qiskit-ignis': None, 'qiskit-ibmq-pr
ovider': '0.19.2', 'qiskit': '0.39.5', 'qiskit-nature': None, 'qiskit-finance': None, 'q
iskit-optimization': None, 'qiskit-machine-learning': None}

In [4]: from qiskit import IBMQ

In [5]: IBMQ.save_account('2d491a0a44bdd5ec696a4b09fb6de02f0345758ae91dd7b7be352675c82000e0fb025

        configrc.store_credentials:WARNING:2023-04-27 15:36:04,209: Credentials already present.
        Set overwrite=True to overwrite.

In [6]: IBMQ.load_account()

Out[6]: <AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>

In [ ]:
```

Conclusion

In summary, setting up Anaconda3 and Qiskit as well as connecting the IBMQ account to Qiskit is a critical first step in delving into the exciting world of quantum computing. A user-friendly interface for creating and running quantum algorithms on actual quantum devices and simulators is offered by the open-source quantum computing framework Qiskit. The IBMQ quantum computing platform, on the other hand, gives users access to a network of quantum devices and simulators that can be accessed through Qiskit. The availability of Qiskit and IBMQ makes it simpler for academics and developers to investigate this fascinating topic. Quantum computing has the potential to revolutionise many fields of science, from cryptography to drug discovery. Overall, the successful installation of Anaconda3, Qiskit, and linking of the IBMQ account is an important step towards unlocking the potential of quantum computing.

Experiment 2

Aim

Implementation of Single qubit gates on Qiskit.

Theory

Quantum computing uses quantum mechanical phenomena, similar as superposition and entanglement, to perform operations on data. Unlike classical computing, where bits can only be in one state(0 or 1) at a time, qubits can live in both states simultaneously due to the phenomenon of superposition. This allows for the representation and manipulation of complex data sets that are difficult to reuse using classical computing methods.

The Hadamard gate is a single- qubit gate that puts a qubit into a superposition state, allowing the qubit to exist in both the 0 and 1 states simultaneously. The Hadamard gate is represented by the matrix:

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

The Pauli gates, including the X, Y, and Z gates, are also single-qubit gates that perform rotations around the X, Y, and Z axes of the Bloch sphere, respectively. The X, Y, and Z gates are represented by the matrices:

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

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

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

These gates can be used to manipulate qubits and perform operations on quantum data. In particular, the Hadamard gate is often used to create superposition states, which are important for many quantum algorithms, and the Pauli gates are often used to perform operations on qubits, such as flipping their states or measuring their values.

Qiskit provides a simple and intuitive way to implement these gates using its QuantumCircuit class and a variety of built-in gate methods, such as h, x, y, and z. These gates can be used to construct more complex quantum circuits and execute quantum algorithms on a variety of quantum devices and simulators.

Quantum computers are subject to several types of noise, which can lead to errors in quantum computations. One type of noise is decoherence, which occurs when a quantum system interacts with its environment and loses coherence. Decoherence limits the amount of time that a quantum system can maintain its superposition state, making it difficult to perform complex computations. To mitigate the effects of noise, quantum error correction techniques, such as the surface code, are used to detect and correct errors in quantum computations.

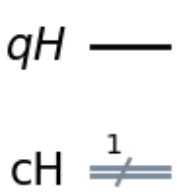
Lab Exercise

Code:

In [1]:
```
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np

qH = QuantumRegister(1, name='qH')
cH = ClassicalRegister(1, name='cH')
circuit_H = QuantumCircuit(qH, cH)

qX = QuantumRegister(1, name='qX')
cX = ClassicalRegister(1, name='cX')
circuit_X = QuantumCircuit(qX, cX)

qY = QuantumRegister(1, name='qY')
cY = ClassicalRegister(1, name='cY')
circuit_Y = QuantumCircuit(qY, cY)

qZ = QuantumRegister(1, name='qZ')
cZ = ClassicalRegister(1, name='cZ')
circuit_Z = QuantumCircuit(qZ, cZ)

%matplotlib inline
```

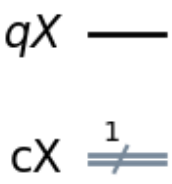
In [2]:
```
circuit_H.draw(output='mpl')
```

Out[2]:



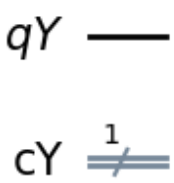
In [3]:
```
circuit_X.draw(output='mpl')
```

Out[3]:



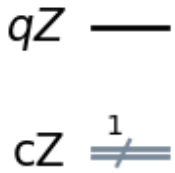
In [4]:
```
circuit_Y.draw(output='mpl')
```

Out[4]:



In [5]:
```
circuit_Z.draw(output='mpl')
```

Out[5]:



In [ ]:
```

```

Here, We prepare all qubits in State: |0⟩

In [6]:
```
circuit_H.barrier(qH, label='|0⟩')
circuit_H.reset(qH)

circuit_X.barrier(qX, label='|0⟩')
circuit_X.reset(qX)

circuit_Y.barrier(qY, label='|0⟩')
circuit_Y.reset(qY)

circuit_Z.barrier(qZ, label='|0⟩')
circuit_Z.reset(qZ)
```

Out[6]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8b7fc5e0>
```

In [7]:
```
state_H_1 = Statevector.from_int(0, 2**1)
state_H_1 = state_H_1.evolve(circuit_H)
state_H_1.draw('latex')
```

Out[7]:

$$|0\rangle$$

In [8]:
```
state_X_1 = Statevector.from_int(0, 2**1)
state_X_1 = state_X_1.evolve(circuit_X)
state_X_1.draw('latex')
```

Out[8]:

$$|0\rangle$$

In [9]:
```
state_Y_1 = Statevector.from_int(0, 2**1)
state_Y_1 = state_Y_1.evolve(circuit_Y)
state_Y_1.draw('latex')
```

Out[9]:

$$|0\rangle$$

In [10]:
```
state_Z_1 = Statevector.from_int(0, 2**1)
state_Z_1 = state_Z_1.evolve(circuit_Z)
state_Z_1.draw('latex')
```

Out[10]:

$$|0\rangle$$

In [ ]:
```
```

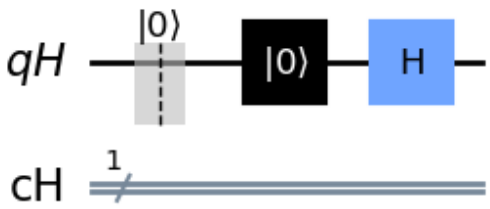
We now apply quantum logic gates on |0⟩

In [11]:
```
circuit_H.h(qH)
circuit_X.x(qX)
circuit_Y.y(qY)
circuit_Z.z(qZ)
```

Out[11]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8b882310>
```

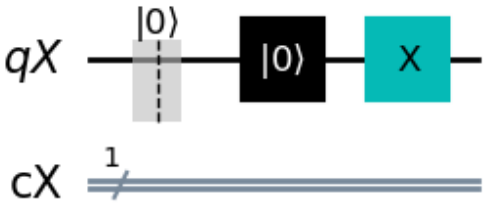
In [12]:
```
circuit_H.draw(output='mpl')
```

Out[12]:



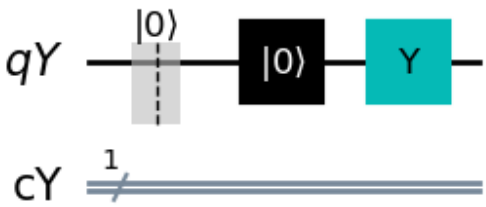
In [13]:
```
circuit_X.draw(output='mpl')
```

Out[13]:



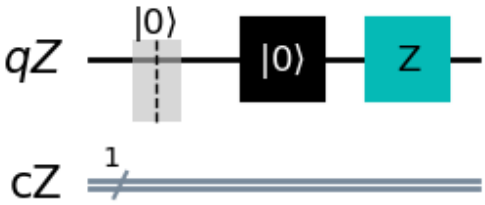
In [14]:
```
circuit_Y.draw(output='mpl')
```

Out[14]:



In [15]:
```
circuit_Z.draw(output='mpl')
```

Out[15]:



In [16]:
```
state_H_2 = Statevector.from_int(0, 2**1)
state_H_2 = state_H_2.evolve(circuit_H)
state_H_2.draw('latex')
```

Out[16]:

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

In [17]:
```
state_X_2 = Statevector.from_int(0, 2**1)
state_X_2 = state_X_2.evolve(circuit_X)
state_X_2.draw('latex')
```

Out[17]:

$$|1\rangle$$

In [18]:
```
state_Y_2 = Statevector.from_int(0, 2**1)
state_Y_2 = state_Y_2.evolve(circuit_Y)
state_Y_2.draw('latex')
```

Out[18]:

$$i|1\rangle$$

In [19]:
```
state_Z_2 = Statevector.from_int(0, 2**1)
state_Z_2 = state_Z_2.evolve(circuit_Z)
state_Z_2.draw('latex')
```

Out[19]:

$$|0\rangle$$

In [ ]:

Here, We prepare all qubits in State: |1⟩

In [20]:
```
circuit_H.barrier(qH, label='|1⟩')
circuit_H.reset(qH)
circuit_H.x(qH)

circuit_X.barrier(qX, label='|1⟩')
circuit_X.reset(qX)
circuit_X.x(qX)

circuit_Y.barrier(qY, label='|1⟩')
circuit_Y.reset(qY)
circuit_Y.x(qY)
circuit_Y.barrier()

circuit_Z.barrier(qZ, label='|1⟩')
circuit_Z.reset(qZ)
circuit_Z.x(qZ)
```

Out[20]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8c0293d0>
```

In [21]:
```
state_H_3 = Statevector.from_int(0, 2**1)
state_H_3 = state_H_3.evolve(circuit_H)
state_H_3.draw('latex')
```

Out[21]:

$$|1\rangle$$

In [22]:
```
state_X_3 = Statevector.from_int(0, 2**1)
state_X_3 = state_X_3.evolve(circuit_X)
state_X_3.draw('latex')
```

Out[22]:

$$|1\rangle$$

In [23]:
```
state_Y_3 = Statevector.from_int(0, 2**1)
state_Y_3 = state_Y_3.evolve(circuit_Y)
state_Y_3.draw('latex')
```

Out[23]:

$$i|1\rangle$$

In [24]:
```
state_Z_4 = Statevector.from_int(0, 2**1)
state_Z_4 = state_Z_4.evolve(circuit_Z)
state_Z_4.draw('latex')
```

Out[24]:

$$|1\rangle$$

In [ ]:

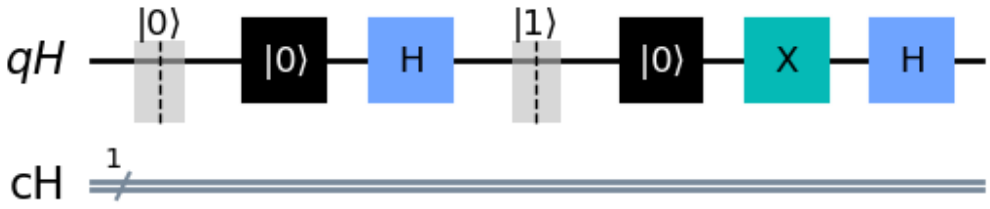
We now apply quantum logic gates on $|1\rangle$

In [25]:
```
circuit_H.h(qH)
circuit_X.x(qX)
circuit_Y.y(qY)
circuit_Z.z(qZ)
```

Out[25]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8c034370>
```

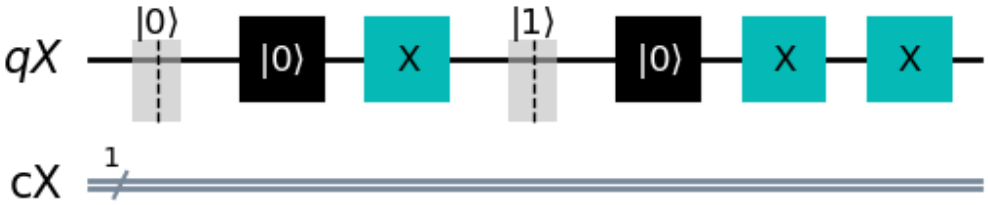
In [26]:
```
circuit_H.draw(output='mpl')
```

Out[26]:



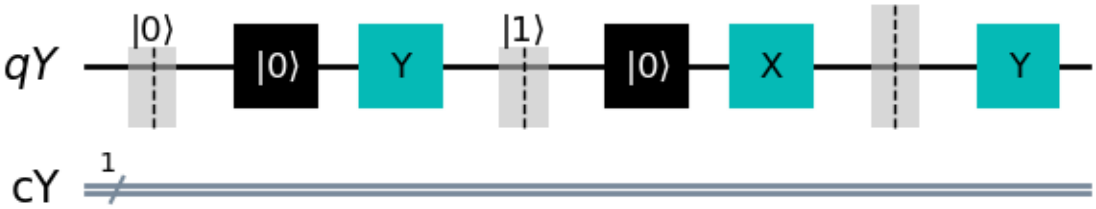
In [27]:
```
circuit_X.draw(output='mpl')
```

Out[27]:



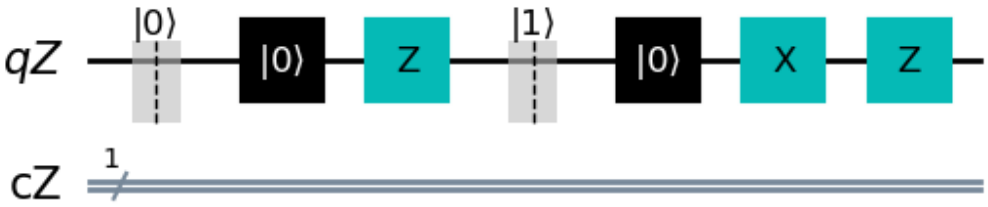
In [28]:
```
circuit_Y.draw(output='mpl')
```

Out[28]:



In [29]:
```
circuit_Z.draw(output='mpl')
```

Out[29]:



In [30]:
```
state_H_4 = Statevector.from_int(0, 2**1)
state_H_4 = state_H_4.evolve(circuit_H)
state_H_4.draw('latex')
```

Out[30]:

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

In [31]:
```
state_X_4 = Statevector.from_int(0, 2**1)
state_X_4 = state_X_4.evolve(circuit_X)
state_X_4.draw('latex')
```

Out[31]:

$$|0\rangle$$

In [32]:
```
state_Y_4 = Statevector.from_int(0, 2**1)
state_Y_4 = state_Y_4.evolve(circuit_Y)
state_Y_4.draw('latex')
```

Out[32]:

$$|0\rangle$$

In [33]:
```
state_Z_4 = Statevector.from_int(0, 2**1)
state_Z_4 = state_Z_4.evolve(circuit_Z)
state_Z_4.draw('latex')
```

Out[33]:

$$-|1\rangle$$

In [ ]:
```

```

Here, We prepare all qubits in State: |+⟩

In [34]:
```
circuit_H.barrier(qH, label='|+⟩')
circuit_H.reset(qH)
circuit_H.h(qH)

circuit_X.barrier(qX, label='|+⟩')
circuit_X.reset(qX)
circuit_X.h(qX)

circuit_Y.barrier(qY, label='|+⟩')
```

```
circuit_Y.reset(qY)
circuit_Y.h(qY)

circuit_Z.barrier(qZ, label='|+⟩')
circuit_Z.reset(qZ)
circuit_Z.h(qZ)
```

Out[34]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8c2aa6d0>
```

In [35]:
```
state_H_5 = Statevector.from_int(0, 2**1)
state_H_5 = state_H_5.evolve(circuit_H)
state_H_5.draw('latex')
```

Out[35]:

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

In [36]:
```
state_X_5 = Statevector.from_int(0, 2**1)
state_X_5 = state_X_5.evolve(circuit_X)
state_X_5.draw('latex')
```

Out[36]:

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

In [37]:
```
state_Y_5 = Statevector.from_int(0, 2**1)
state_Y_5 = state_Y_5.evolve(circuit_Y)
state_Y_5.draw('latex')
```

Out[37]:

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

In [38]:
```
state_Z_5 = Statevector.from_int(0, 2**1)
state_Z_5 = state_Z_5.evolve(circuit_Z)
state_Z_5.draw('latex')
```

Out[38]:

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

In [ ]:
```

```

We now apply quantum logic gates on $|+\rangle$

In [39]:
```
circuit_H.h(qH)
circuit_X.x(qX)
circuit_Y.y(qY)
circuit_Z.z(qZ)
```

Out[39]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8c2bb670>
```

In [40]:
```
circuit_H.draw(output='mpl')
```

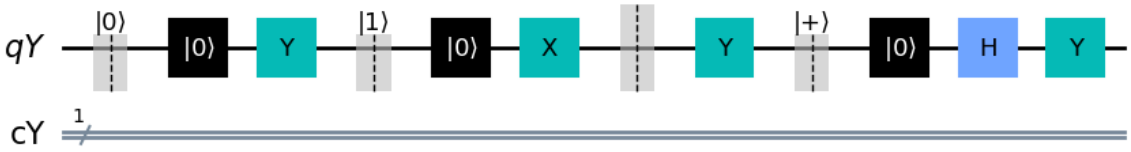
Out[40]:

In [41]:
```
circuit_X.draw(output='mpl')
```

Out[41]:

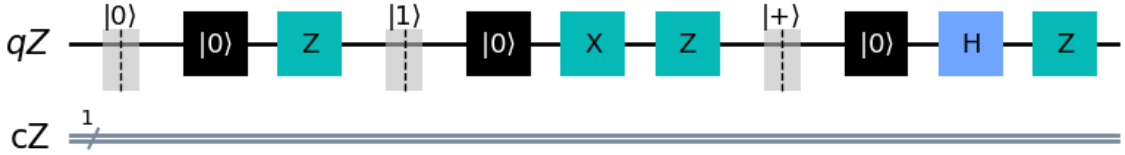
In [42]:
```
circuit_Y.draw(output='mpl')
```

Out[42]:



In [43]:
```
circuit_Z.draw(output='mpl')
```

Out[43]:



In [44]:
```
state_H_6 = Statevector.from_int(0, 2**1)
state_H_6 = state_H_6.evolve(circuit_H)
state_H_6.draw('latex')
```

Out[44]:

$$-|0\rangle$$

In [45]:
```
state_X_6 = Statevector.from_int(0, 2**1)
state_X_6 = state_X_6.evolve(circuit_X)
state_X_6.draw('latex')
```

Out[45]:

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

In [46]:
```
state_Y_6 = Statevector.from_int(0, 2**1)
state_Y_6 = state_Y_6.evolve(circuit_Y)
state_Y_6.draw('latex')
```

Out[46]:

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

In [47]:
```
state_Z_6 = Statevector.from_int(0, 2**1)
state_Z_6 = state_Z_6.evolve(circuit_Z)
state_Z_6.draw('latex')
```

Out[47]:

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

In [ ]:
```

```

Here, We prepare all qubits in State: |-⟩

In [48]:
```
circuit_H.barrier(qH, label='|-⟩')
circuit_H.reset(qH)
circuit_H.x(qH)
circuit_H.h(qH)

circuit_X.barrier(qX, label='|-⟩')
circuit_X.reset(qX)
circuit_X.x(qX)
circuit_X.h(qX)

circuit_Y.barrier(qY, label='|-⟩')
circuit_Y.reset(qY)
circuit_Y.x(qY)
circuit_Y.h(qY)

circuit_Z.barrier(qZ, label='|-⟩')
circuit_Z.reset(qZ)
circuit_Z.x(qZ)
circuit_Z.h(qZ)
```

Out[48]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8d4fec10>
```

In [49]:
```
state_H_7 = Statevector.from_int(0, 2**1)
state_H_7 = state_H_7.evolve(circuit_H)
state_H_7.draw('latex')
```

Out[49]:

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

In [50]:
```
state_X_7 = Statevector.from_int(0, 2**1)
state_X_7 = state_X_7.evolve(circuit_X)
state_X_7.draw('latex')
```

Out[50]:

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

In [51]:
```
state_Y_7 = Statevector.from_int(0, 2**1)
state_Y_7 = state_Y_7.evolve(circuit_Y)
state_Y_7.draw('latex')
```

Out[51]:

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

In [52]:
```
state_Z_7 = Statevector.from_int(0, 2**1)
state_Z_7 = state_Z_7.evolve(circuit_Z)
state_Z_7.draw('latex')
```

Out[52]:

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

In [ ]:
```

```

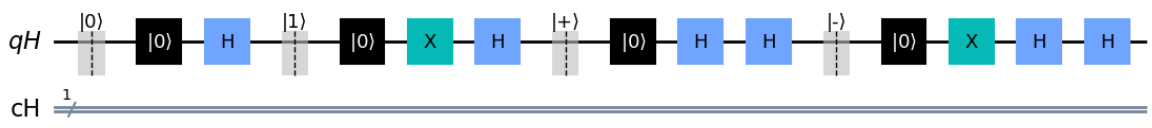
We now apply quantum logic gates on |-⟩

In [53]:
```
circuit_H.h(qH)
circuit_X.x(qX)
circuit_Y.y(qY)
circuit_Z.z(qZ)
```

Out[53]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x25e8d4f72b0>
```

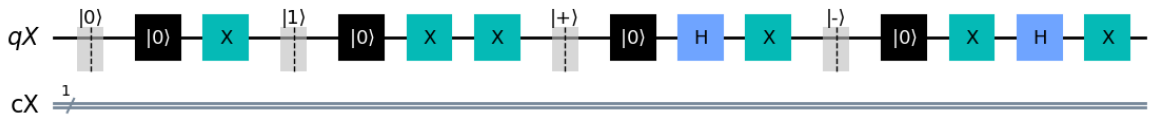
In [54]:
```
circuit_H.draw(output='mpl')
```

Out[54]:



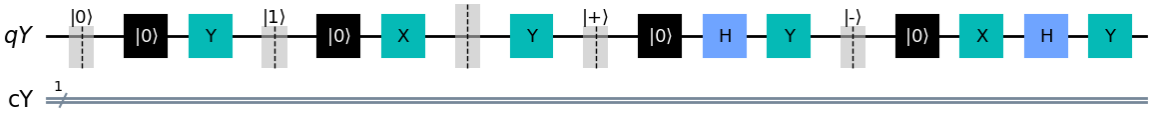
In [55]:
```
circuit_X.draw(output='mpl')
```

Out[55]:



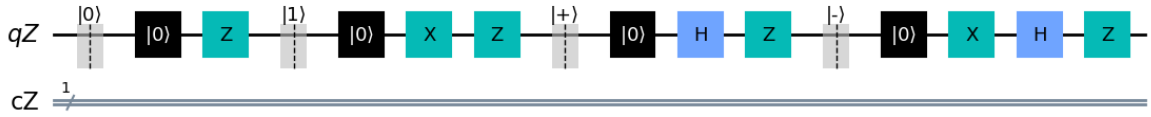
In [56]:
```
circuit_Y.draw(output='mpl')
```

Out[56]:



In [57]:
```
circuit_Z.draw(output='mpl')
```

Out[57]:



In [58]:
```
state_H_8 = Statevector.from_int(0, 2**1)
state_H_8 = state_H_8.evolve(circuit_H)
state_H_8.draw('latex')
```

Out[58]:

$$-|1\rangle$$

In [59]:
```
state_X_8 = Statevector.from_int(0, 2**1)
state_X_8 = state_X_8.evolve(circuit_X)
state_X_8.draw('latex')
```

Out[59]:

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

In [60]:
```
state_Y_8 = Statevector.from_int(0, 2**1)
state_Y_8 = state_Y_8.evolve(circuit_Y)
state_Y_8.draw('latex')
```

Out[60]:

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

In [61]:
```
state_Z_8 = Statevector.from_int(0, 2**1)
state_Z_8 = state_Z_8.evolve(circuit_Z)
state_Z_8.draw('latex')
```

Out[61]:

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

Conclusion

In conclusion, the use of Qiskit to create single qubit gates has given users a practical introduction to quantum computing. To produce superposition states and alter the foundation of a qubit state, respectively, the Hadamard and Pauli gates were used. These gates are key building blocks for creating more complex quantum circuits and are utilised in numerous quantum algorithms.

The usage of Qiskit enabled the construction of quantum circuits, their visualisation, and their execution on a simulator or a genuine quantum device via IBMQ. The results of the circuit simulation on the simulator were in line with what was anticipated, proving that the gates were implemented correctly.

Overall, this experiment has provided a better understanding of quantum computing and its potential applications. The use of Qiskit and IBMQ has made quantum computing more accessible and has enabled the exploration of quantum algorithms, quantum error correction techniques, and other areas of quantum research.

Experiment 3

Aim

Implementation of Classical gates using multi qubit quantum gates in Qiskit.

Theory

The implementation of classical gates using multi-qubit quantum gates in Qiskit involves using quantum gates to simulate classical operations. In this experiment, we will use the controlled-NOT (CNOT) gate and the Toffoli gate, which are two multi-qubit gates commonly used for implementing classical gates.

The CNOT gate is a two-qubit gate that flips the target qubit if the control qubit is in the state —1⟩. The CNOT gate can be used to simulate classical gates, such as the AND, OR, and NOT gates. The truth table for the CNOT gate is given by:

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

Here, the first two rows correspond to the input states $|00\rangle$ and $|01\rangle$, and the last two rows correspond to the input states $|10\rangle$ and $|11\rangle$. The first column corresponds to the output state $|00\rangle$, the second column corresponds to the output state $|01\rangle$, and so on.

To implement classical gates using the CNOT gate, we can use one qubit as the control qubit and the other qubit as the target qubit. If the control qubit is in the state $|1\rangle$, the target qubit is flipped, simulating the classical operation.

The Toffoli gate, also known as the controlled-controlled-NOT (CCNOT) gate, is a three-qubit gate that flips the target qubit if both control qubits are in the state $|1\rangle$. The Toffoli gate can be used to simulate classical gates, such as the NAND gate. The truth table for the Toffoli gate is given by:

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

Here, the first four rows correspond to the input states $|000\rangle$, $|001\rangle$, $|010\rangle$, and $|011\rangle$, and the last four rows correspond to the input states $|100\rangle$, $|101\rangle$, $|110\rangle$, and $|111\rangle$. The first column corresponds to the output state —$000\rangle$, the second column corresponds to the output state $|001\rangle$, and so on.

To implement classical gates using the Toffoli gate, we can use two qubits as the control qubits and the third qubit as the target qubit. If both control qubits are in the state —1⟩, the target qubit is flipped, simulating the classical operation.

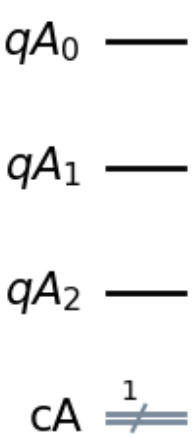
Lab Exercise

Code:

In [1]:
```
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np

qA = QuantumRegister(3, name='qA')
cA = ClassicalRegister(1, name='cA')
circuit_A = QuantumCircuit(qA, cA)

qO = QuantumRegister(3, name='qO')
cO = ClassicalRegister(1, name='cO')
circuit_O = QuantumCircuit(qO, cO)

qE = QuantumRegister(2, name='qE')
cE = ClassicalRegister(1, name='cE')
circuit_E = QuantumCircuit(qE, cE)

%matplotlib inline
```

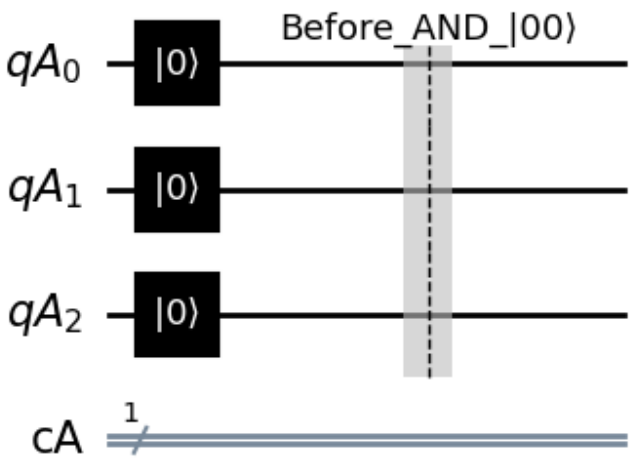
In [2]:
```
circuit_A.draw(output='mpl')
```

Out[2]:



In [3]:
```
circuit_A.reset(qA)
circuit_A.barrier(label='Before_AND_|00⟩')
circuit_A.draw(output='mpl')
```

Out[3]:



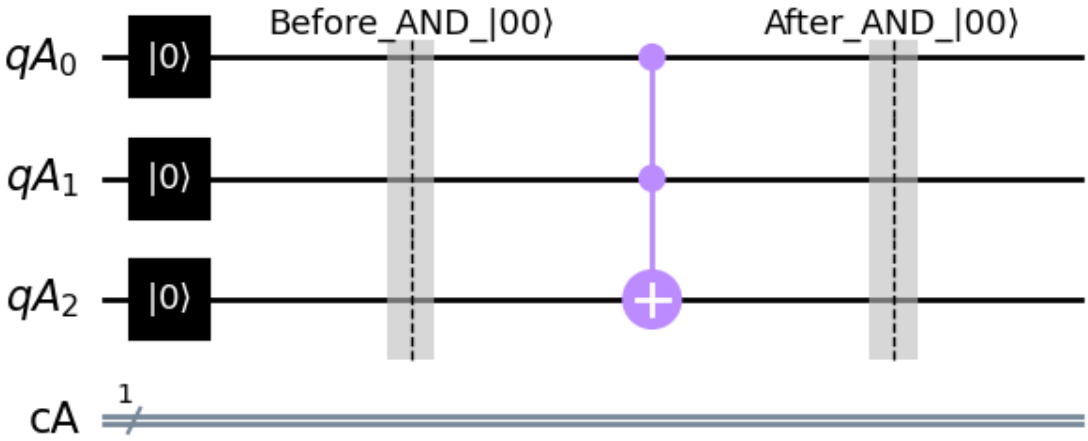
In [4]:
```
state_A_1 = Statevector.from_int(0, 2**3)
state_A_1 = state_A_1.evolve(circuit_A)
state_A_1.draw('latex')
```

Out[4]:

$$|000\rangle$$

In [5]:
```
circuit_A.ccx(qA[0], qA[1], qA[2])
circuit_A.barrier(label='After_AND_|00⟩')
circuit_A.draw(output='mpl')
```

Out[5]:



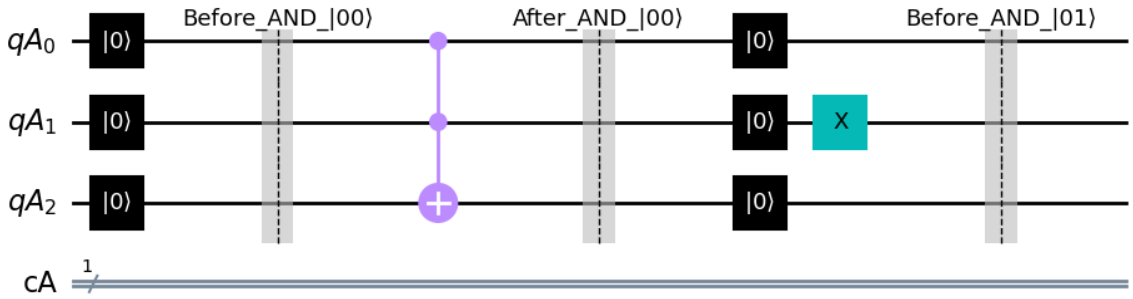
In [6]:
```
state_A_2 = Statevector.from_int(0, 2**3)
state_A_2 = state_A_2.evolve(circuit_A)
state_A_2.draw('latex')
#output(MSB) = 0
```

Out[6]:

$$|000\rangle$$

In [7]:
```
circuit_A.reset(qA)
circuit_A.x(qA[1])
circuit_A.barrier(label='Before_AND_|01⟩')
circuit_A.draw(output='mpl')
```

Out[7]:



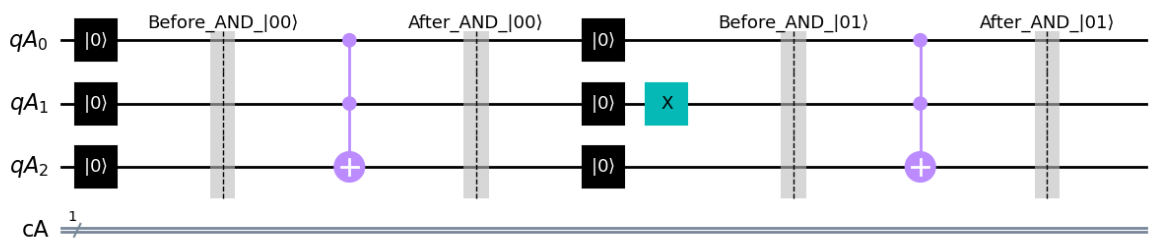
In [8]:
```
state_A_3 = Statevector.from_int(0, 2**3)
state_A_3 = state_A_3.evolve(circuit_A)
state_A_3.draw('latex')
```

Out[8]:

$$|010\rangle$$

In [9]:
```
circuit_A.ccx(qA[0], qA[1], qA[2])
circuit_A.barrier(label='After_AND_|01⟩')
circuit_A.draw(output='mpl')
```

Out[9]:



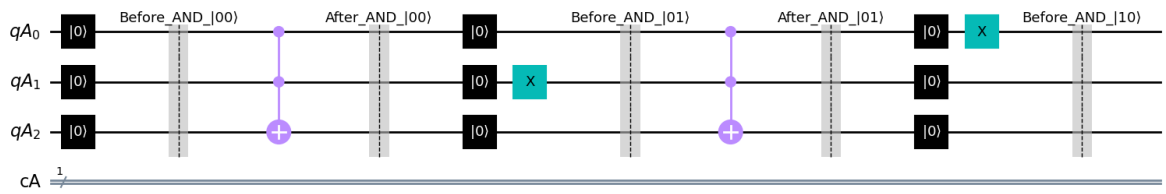
In [10]:
```
state_A_4 = Statevector.from_int(0, 2**3)
state_A_4 = state_A_4.evolve(circuit_A)
state_A_4.draw('latex')
#output(MSB) = 0
```

Out[10]:

$$|010\rangle$$

In [11]:
```
circuit_A.reset(qA)
circuit_A.x(qA[0])
circuit_A.barrier(label='Before_AND_|10⟩')
circuit_A.draw(output='mpl')
```

Out[11]:



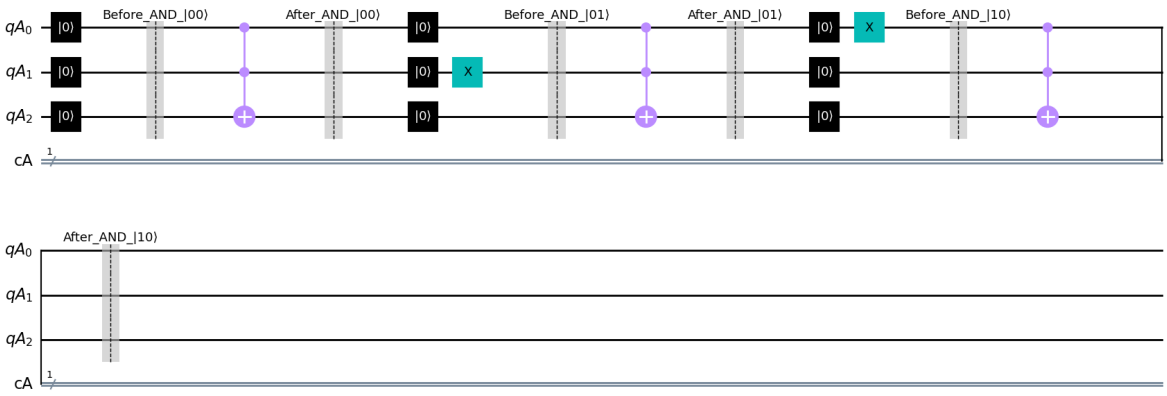
In [12]:
```
state_A_5 = Statevector.from_int(0, 2**3)
state_A_5 = state_A_5.evolve(circuit_A)
state_A_5.draw('latex')
```

Out[12]:

$$|001\rangle$$

In [13]:
```
circuit_A.ccx(qA[0], qA[1], qA[2])
circuit_A.barrier(label='After_AND_|10⟩')
circuit_A.draw(output='mpl')
```

Out[13]:



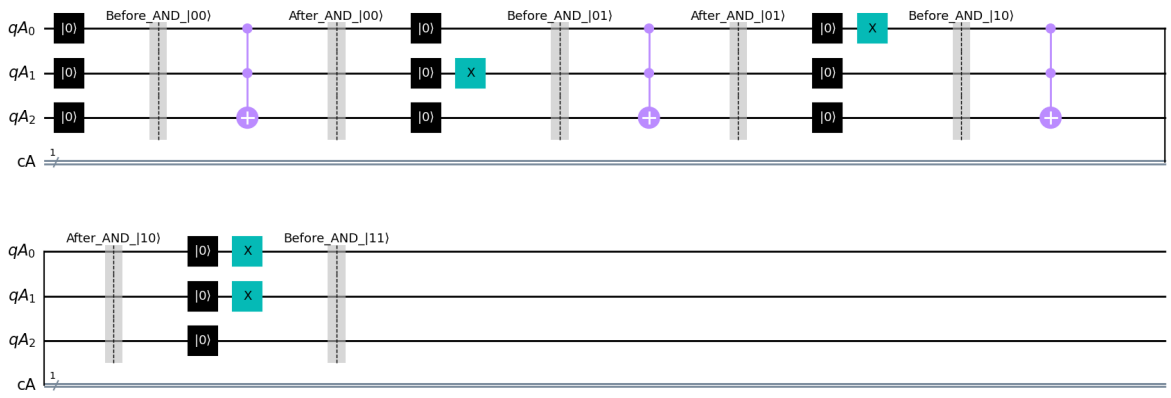
In [14]:
```
state_A_6 = Statevector.from_int(0, 2**3)
state_A_6 = state_A_6.evolve(circuit_A)
state_A_6.draw('latex')
#output(MSB) = 0
```

Out[14]:

$$|001\rangle$$

In [15]:
```
circuit_A.reset(qA)
circuit_A.x(qA[0])
circuit_A.x(qA[1])
circuit_A.barrier(label='Before_AND_|11⟩')
circuit_A.draw(output='mpl')
```

Out[15]:



In [16]:

```
state_A_7 = Statevector.from_int(0, 2**3)
state_A_7 = state_A_7.evolve(circuit_A)
state_A_7.draw('latex')
```

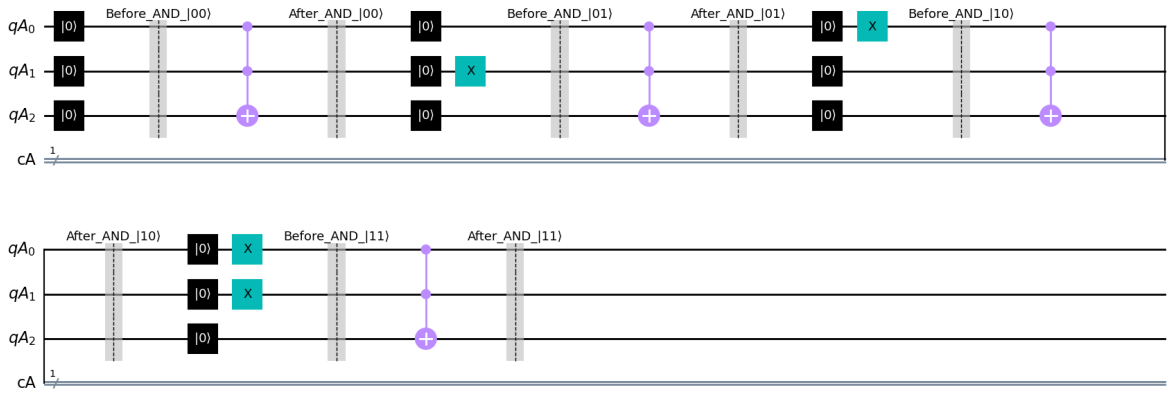
Out[16]:

$$|011\rangle$$

In [17]:

```
circuit_A.ccx(qA[0], qA[1], qA[2])
circuit_A.barrier(label='After_AND_|11⟩')
circuit_A.draw(output='mpl')
```

Out[17]:



In [18]:

```
state_A_8 = Statevector.from_int(0, 2**3)
state_A_8 = state_A_8.evolve(circuit_A)
state_A_8.draw('latex')
#output(MSB) = 1
```

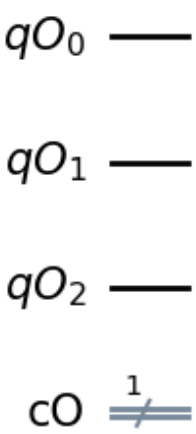
Out[18]:

$$|111\rangle$$

In [19]:

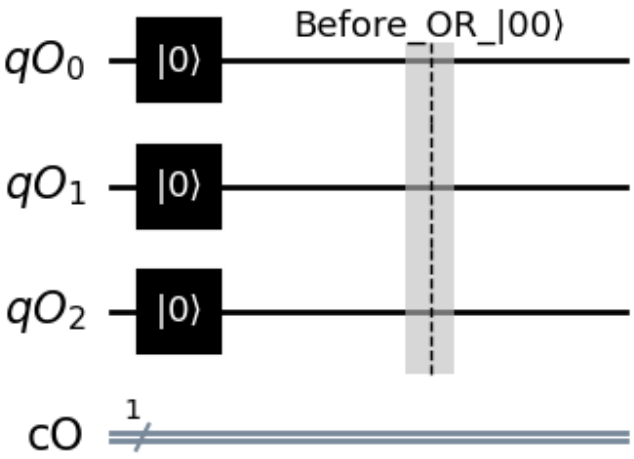
```
circuit_O.draw(output='mpl')
```

Out[19]:



In [20]:
```
circuit_O.reset(qO)
circuit_O.barrier(label='Before_OR_|00⟩')
circuit_O.draw(output='mpl')
```

Out[20]:



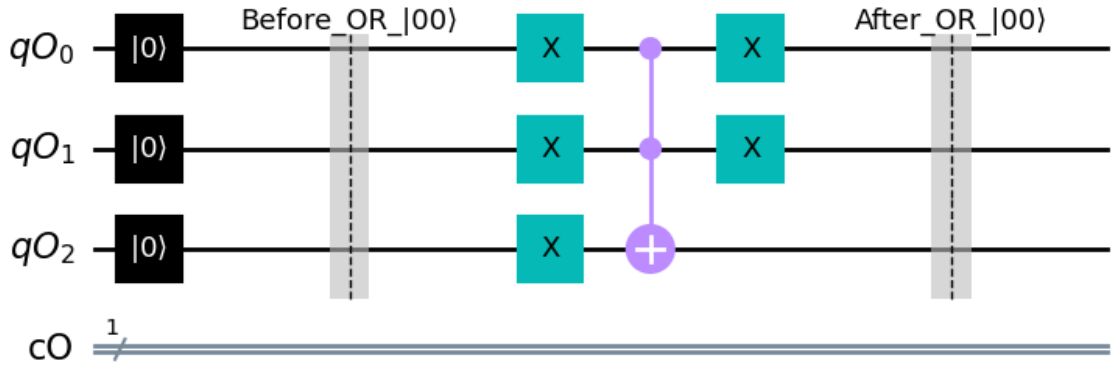
In [21]:
```
state_O_1 = Statevector.from_int(0, 2**3)
state_O_1 = state_O_1.evolve(circuit_O)
state_O_1.draw('latex')
```

Out[21]:

$$|000\rangle$$

In [22]:
```
circuit_O.x(qO)
circuit_O.ccx(qO[0], qO[1], qO[2])
circuit_O.x(qO[0])
circuit_O.x(qO[1])
circuit_O.barrier(label='After_OR_|00⟩')
circuit_O.draw(output='mpl')
```

Out[22]:



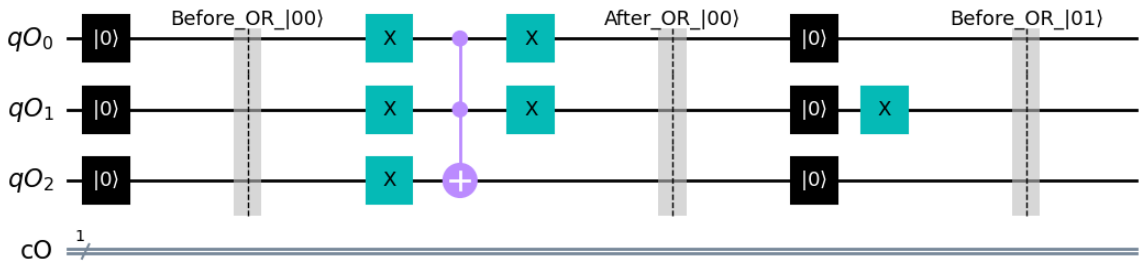
In [23]:
```
state_O_2 = Statevector.from_int(0, 2**3)
state_O_2 = state_O_2.evolve(circuit_O)
state_O_2.draw('latex')
#output(MSB) = 0
```

Out[23]:

$$|000\rangle$$

In [24]:
```
circuit_O.reset(qO)
circuit_O.x(qO[1])
circuit_O.barrier(label='Before_OR_|01⟩')
circuit_O.draw(output='mpl')
```

Out[24]:



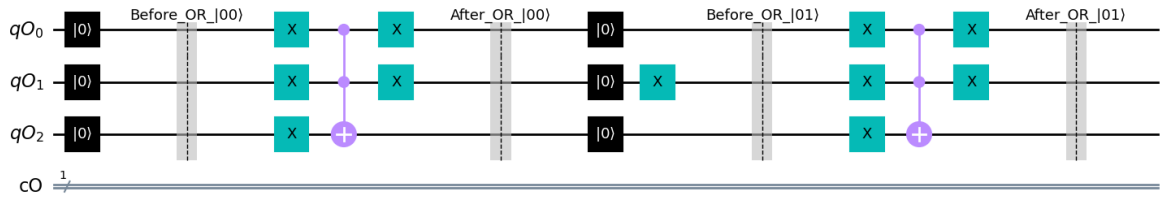
In [25]:
```
state_O_3 = Statevector.from_int(0, 2**3)
state_O_3 = state_O_3.evolve(circuit_O)
state_O_3.draw('latex')
```

Out[25]:

$$|010\rangle$$

In [26]:
```
circuit_O.x(qO)
circuit_O.ccx(qO[0], qO[1], qO[2])
circuit_O.x(qO[0])
circuit_O.x(qO[1])
circuit_O.barrier(label='After_OR_|01⟩')
circuit_O.draw(output='mpl')
```

Out[26]:



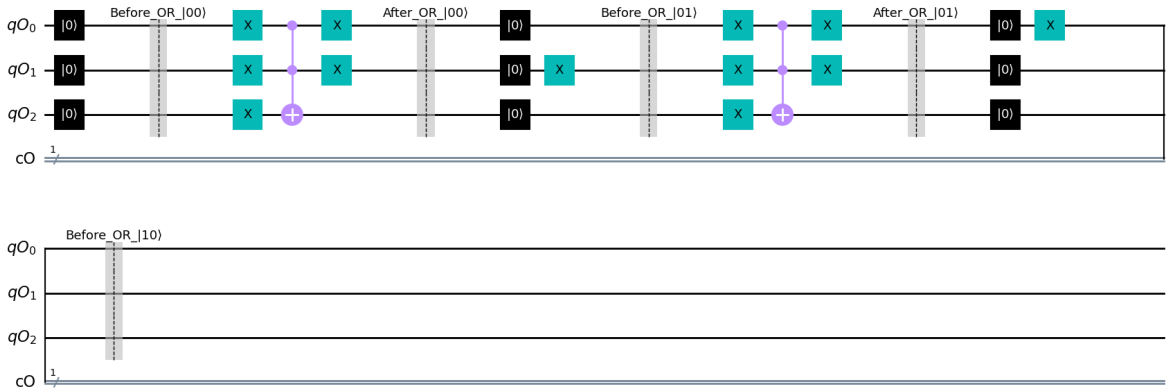
In [27]:
```
state_O_4 = Statevector.from_int(0, 2**3)
state_O_4 = state_O_4.evolve(circuit_O)
state_O_4.draw('latex')
#output(MSB) = 1
```

Out[27]:

$$|110\rangle$$

In [28]:
```
circuit_O.reset(qO)
circuit_O.x(qO[0])
circuit_O.barrier(label='Before_OR_|10⟩')
circuit_O.draw(output='mpl')
```

Out[28]:



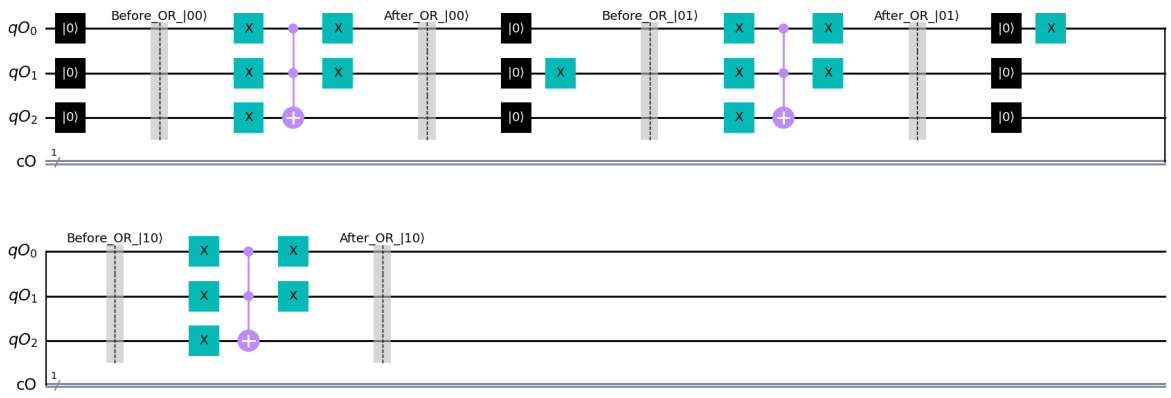
In [29]:
```
state_O_5 = Statevector.from_int(0, 2**3)
state_O_5 = state_O_5.evolve(circuit_O)
state_O_5.draw('latex')
```

Out[29]:

$$|001\rangle$$

In [30]:
```
circuit_O.x(qO)
circuit_O.ccx(qO[0], qO[1], qO[2])
circuit_O.x(qO[0])
circuit_O.x(qO[1])
circuit_O.barrier(label='After_OR_|10⟩')
circuit_O.draw(output='mpl')
```

Out[30]:



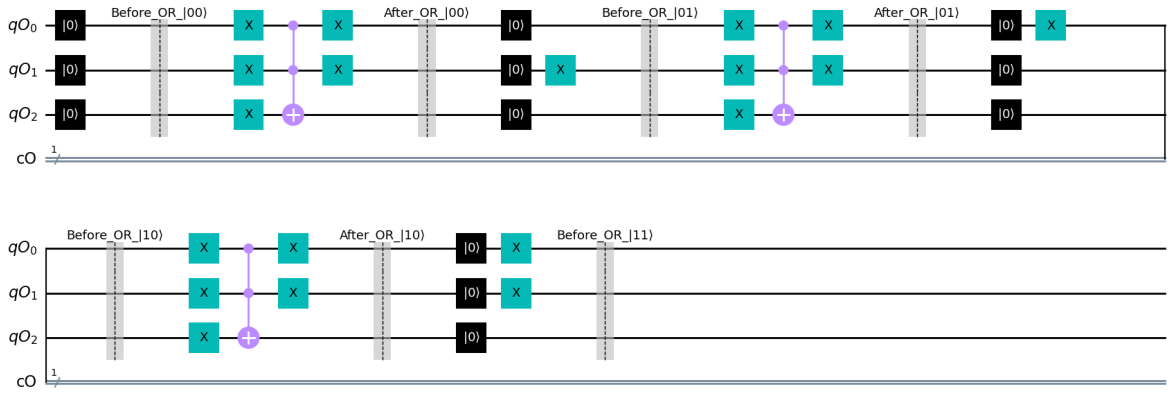
In [31]:
```
state_O_6 = Statevector.from_int(0, 2**3)
state_O_6 = state_O_6.evolve(circuit_O)
state_O_6.draw('latex')
#output(MSB) = 1
```

Out[31]:

$$|101\rangle$$

In [32]:
```
circuit_O.reset(qO)
circuit_O.x(qO[0])
circuit_O.x(qO[1])
circuit_O.barrier(label='Before_OR_|11⟩')
circuit_O.draw(output='mpl')
```

Out[32]:



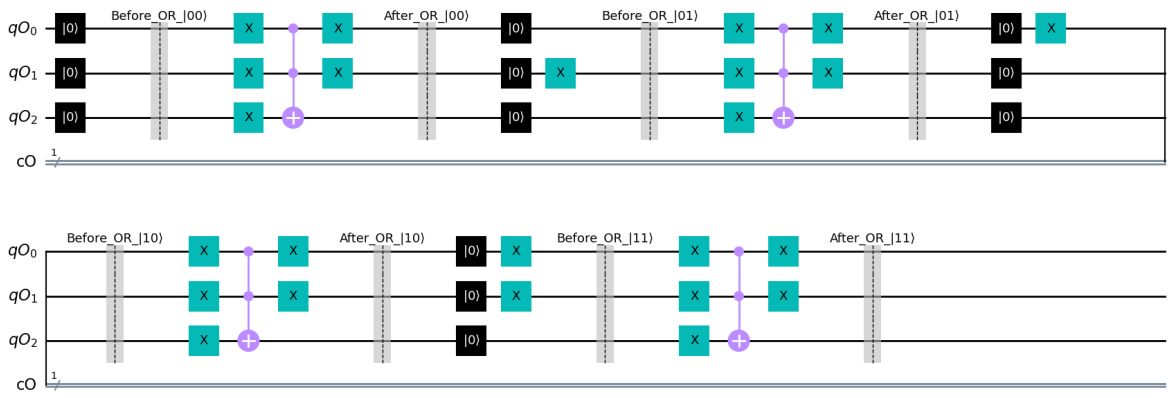
In [33]:
```
state_O_7 = Statevector.from_int(0, 2**3)
state_O_7 = state_O_7.evolve(circuit_O)
state_O_7.draw('latex')
```

Out[33]:

$$|011\rangle$$

In [34]:
```
circuit_O.x(qO)
circuit_O.ccx(qO[0], qO[1], qO[2])
circuit_O.x(qO[0])
circuit_O.x(qO[1])
circuit_O.barrier(label='After_OR_|11⟩')
circuit_O.draw(output='mpl')
```

Out[34]:



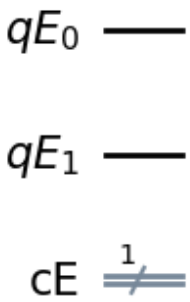
In [35]:
```
state_O_8 = Statevector.from_int(0, 2**3)
state_O_8 = state_O_8.evolve(circuit_O)
state_O_8.draw('latex')
#output(MSB) = 1
```

Out[35]:

$$|111\rangle$$

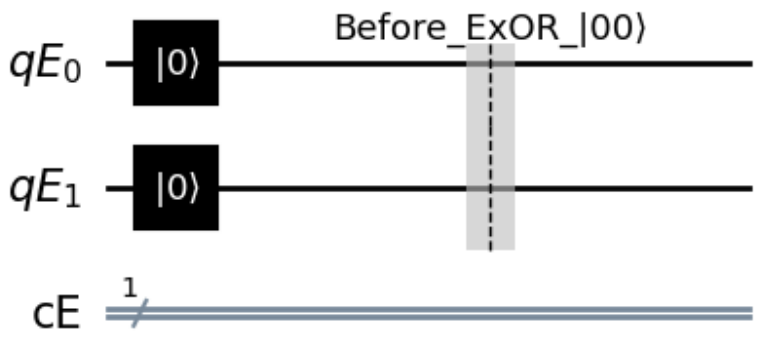
In [36]:
```
circuit_E.draw(output='mpl')
```

Out[36]:



In [37]:
```
circuit_E.reset(qE)
circuit_E.barrier(label='Before_ExOR_|00⟩')
circuit_E.draw(output='mpl')
```

Out[37]:



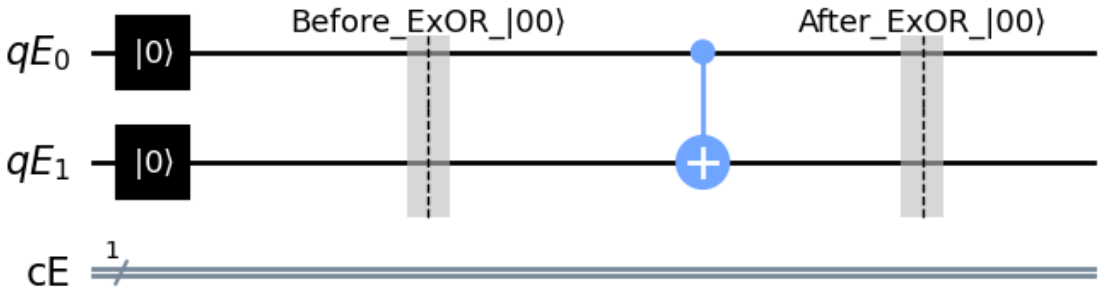
In [38]:
```
state_E_1 = Statevector.from_int(0, 2**2)
state_E_1 = state_E_1.evolve(circuit_E)
state_E_1.draw('latex')
```

Out[38]:

$$|00\rangle$$

In [39]:
```
circuit_E.cx(qE[0], qE[1])
circuit_E.barrier(label='After_ExOR_|00⟩')
circuit_E.draw(output='mpl')
```

Out[39]:



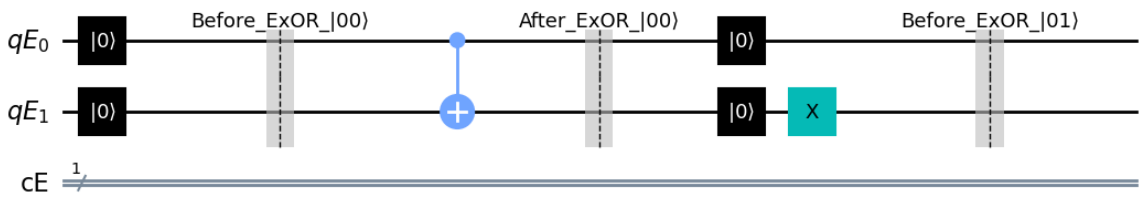
In [40]:
```
state_E_2 = Statevector.from_int(0, 2**2)
state_E_2 = state_E_2.evolve(circuit_E)
state_E_2.draw('latex')
#output(MSB) = 0
```

Out[40]:

$$|00\rangle$$

In [41]:
```
circuit_E.reset(qE)
circuit_E.x(qE[1])
circuit_E.barrier(label='Before_ExOR_|01⟩')
circuit_E.draw(output='mpl')
```

Out[41]:



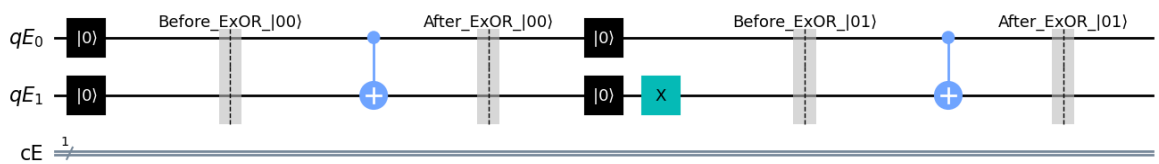
In [42]:
```
state_E_3 = Statevector.from_int(0, 2**2)
state_E_3 = state_E_3.evolve(circuit_E)
state_E_3.draw('latex')
```

Out[42]:

$$|10\rangle$$

In [43]:
```
circuit_E.cx(qE[0], qE[1])
circuit_E.barrier(label='After_ExOR_|01⟩')
circuit_E.draw(output='mpl')
```

Out[43]:



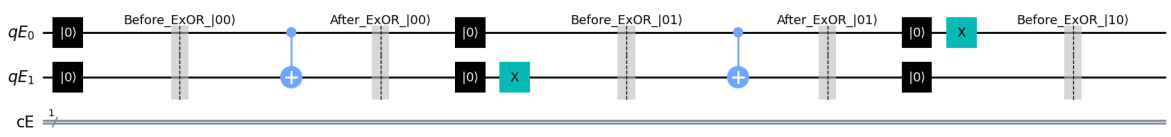
In [44]:
```
state_E_4 = Statevector.from_int(0, 2**2)
state_E_4 = state_E_4.evolve(circuit_E)
state_E_4.draw('latex')
#output(MSB) = 1
```

Out[44]:

$$|10\rangle$$

In [45]:
```
circuit_E.reset(qE)
circuit_E.x(qE[0])
circuit_E.barrier(label='Before_ExOR_|10⟩')
circuit_E.draw(output='mpl')
```

Out[45]:



In [46]:
```
state_E_5 = Statevector.from_int(0, 2**2)
state_E_5 = state_E_5.evolve(circuit_E)
state_E_5.draw('latex')
```

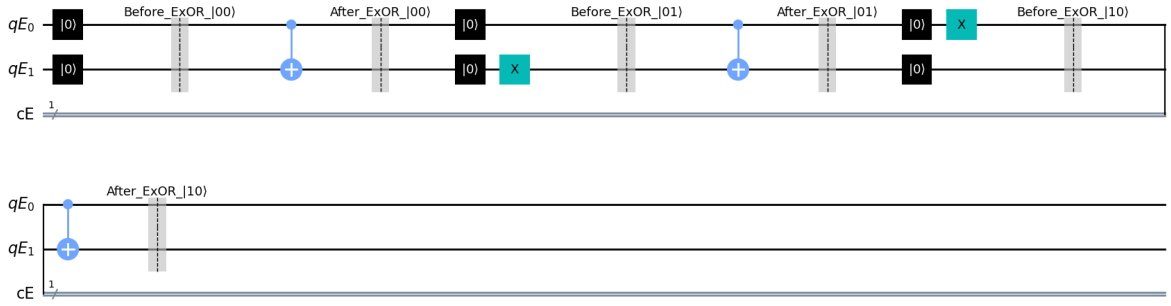
Out[46]:

$$|01\rangle$$

In [47]:
```
circuit_E.cx(qE[0], qE[1])
circuit_E.barrier(label='After_ExOR_|10⟩')
```

```
circuit_E.draw(output='mpl')
```

Out[47]:



In [48]:

```
state_E_6 = Statevector.from_int(0, 2**2)
state_E_6 = state_E_6.evolve(circuit_E)
state_E_6.draw('latex')
#output(MSB) = 1
```

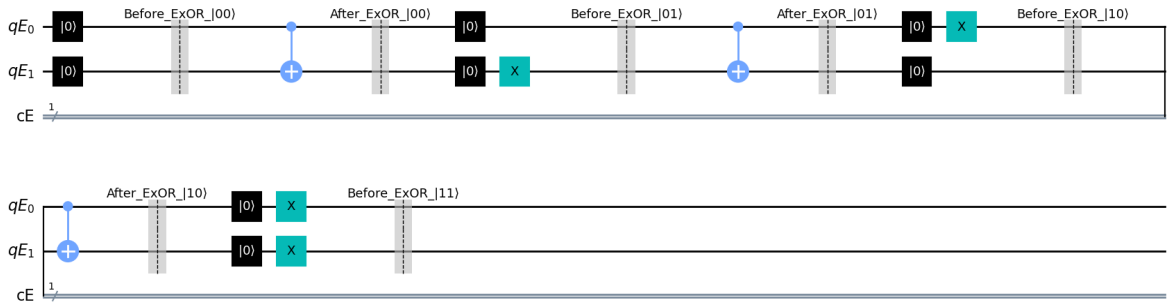
Out[48]:

$$|11\rangle$$

In [49]:

```
circuit_E.reset(qE)
circuit_E.x(qE[0])
circuit_E.x(qE[1])
circuit_E.barrier(label='Before_ExOR_|11⟩')
circuit_E.draw(output='mpl')
```

Out[49]:



In [50]:

```
state_E_7 = Statevector.from_int(0, 2**2)
state_E_7 = state_E_7.evolve(circuit_E)
state_E_7.draw('latex')
```

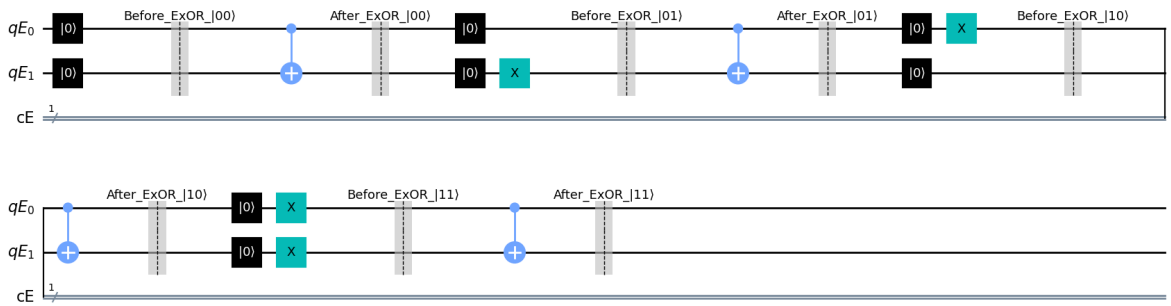
Out[50]:

$$|11\rangle$$

In [51]:

```
circuit_E.cx(qE[0], qE[1])
circuit_E.barrier(label='After_ExOR_|11⟩')
circuit_E.draw(output='mpl')
```

Out[51]:



In [52]:

```
state_E_8 = Statevector.from_int(0, 2**2)
state_E_8 = state_E_8.evolve(circuit_E)
state_E_8.draw('latex')
#output(MSB) = 0
```

Out[52]:

$$|01\rangle$$

In [ ]:

Conclusion

In conclusion, it has been successfully shown that the multi-qubit quantum gates Controlled Not and Toffoli can be used to build classical gates in Qiskit. We have demonstrated how these multi-qubit gates can be used to create conventional gates like AND, OR and XOR. The outcomes of the simulations performed with the Qiskit simulator were in line with the anticipated classical logic operations.

We have also shown that these gates can be utilised to develop quantum error correcting codes and to generate entangled states. This demonstrates how quantum computers have the ability to execute calculations that are not possible with classical computers.

The use of multi-qubit gates in Qiskit has practical applications in quantum computing, particularly in the areas of quantum error correction and quantum-classical computation. As quantum computing technology continues to advance, the use of these gates will become increasingly important for realizing the potential of quantum computing in solving real-world problems.

Experiment 4

Aim

Implementation of Bell circuit and Bell measurement in Qiskit.

Theory

The Bell circuit is a quantum circuit that generates an entangled state between two qubits. The circuit consists of a Hadamard gate applied to the first qubit, followed by a controlled-NOT (CNOT) gate with the first qubit as the control and the second qubit as the target. The circuit can be represented mathematically as follows:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{1}$$

where $|\psi\rangle$ represents the state of the two qubits, $|00\rangle$ and $|11\rangle$ are the basis states of the two qubits, and $\frac{1}{\sqrt{2}}$ is a normalization constant. The Hadamard gate applied to the first qubit can be represented as follows:

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

where H represents the Hadamard gate and $|0\rangle$ and $|1\rangle$ are the basis states of a single qubit. Thus, the action of the Hadamard gate on the first qubit can be written as:

$$H|0\rangle_1|0\rangle_2 = \frac{1}{\sqrt{2}}(|0\rangle_1 + |1\rangle_1)|0\rangle_2 \tag{3}$$

The CNOT gate applied to the two qubits can be represented as follows:

$$\text{CNOT}|x\rangle_1|y\rangle_2 = |x\rangle_1|(x \oplus y)\rangle_2 \tag{4}$$

where x and y are the basis states of the two qubits and $\oplus$ represents the bitwise XOR operation. Thus, the action of the CNOT gate on the two qubits can be written as:

$$\text{CNOT}\Big(\frac{1}{\sqrt{2}}(|0\rangle_1 + |1\rangle_1)|0\rangle_2\Big) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{5}$$

which is the entangled state generated by the Bell circuit.
To perform a Bell measurement, the two entangled qubits are first subjected to a CNOT gate with the first qubit as the control and the second qubit as the target, followed by a Hadamard gate applied to the first qubit. The circuit can be represented mathematically as follows:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{6}$$

$$\text{CNOT}|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \tag{7}$$

$$(H \otimes I)\text{CNOT}|\psi\rangle = |00\rangle \tag{8}$$

identity operator applied to the second qubit. The action of the CNOT gate on the entangled state generates a superposition of the basis states $|00\rangle$ and $|10\rangle$, which are then subjected to a Hadamard gate on the first qubit. This results in the following Bell state:
The Bell measurement is performed by measuring the two qubits in the Bell basis, which consists of the following four orthonormal states:

$$|\phi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{9}$$

$$|\phi_-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \tag{10}$$

$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \tag{11}$$

$$|\psi_-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \tag{12}$$

The Bell basis states are all maximally entangled and are orthogonal to each other. The Bell measurement corresponds to a measurement in the Bell basis, which projects the two qubits onto one of the four Bell basis states with equal probability.

If the measurement outcome is $|\phi_+\rangle$, then the two qubits are in the state $|00\rangle$ or $|11\rangle$. If the measurement outcome is $|\phi_-\rangle$, then the two qubits are in the state $|00\rangle$ or $|11\rangle$ with a relative phase of $\pi/2$. If the measurement outcome is $|\psi_+\rangle$, then the two qubits are in the state $|01\rangle$ or $|10\rangle$. If the measurement outcome is $|\psi_-\rangle$, then the two qubits are in the state $|01\rangle$ or $|10\rangle$ with a relative phase of $\pi/2$.

In summary, the Bell circuit and Bell measurement demonstrate the generation and measurement of entangled states between two qubits, and the ability to perform measurements in the Bell basis to extract information about the entangled state. This has important implications for quantum information processing and communication, where entanglement is a crucial resource for tasks such as quantum teleportation, quantum error correction, and quantum key distribution.

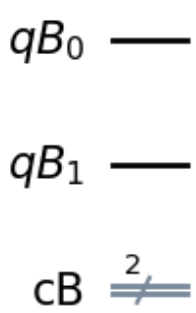
Lab Exercise

Code:

In [1]:
```
from qiskit import *
from qiskit.quantum_info import Statevector
import numpy as np

qB = QuantumRegister(2, name='qB')
cB = ClassicalRegister(2, name='cB')
circuit = QuantumCircuit(qB, cB)

%matplotlib inline
```

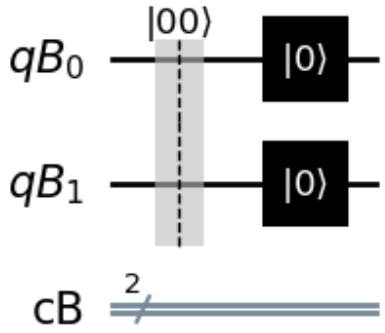
In [2]:
```
circuit.draw(output='mpl')
```

Out[2]:



In [3]:
```
circuit.barrier(qB, label='|00⟩')
circuit.reset(qB)
circuit.draw(output='mpl')
```

Out[3]:



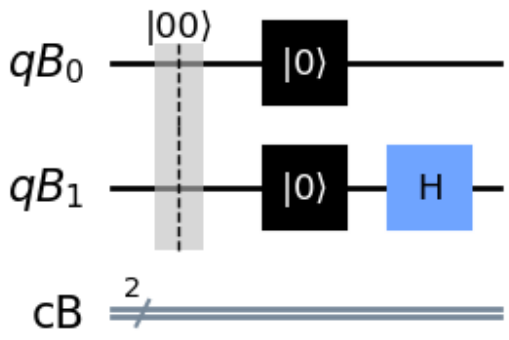
In [4]:
```
state_H_1 = Statevector.from_int(0, 2**2)
state_H_1 = state_H_1.evolve(circuit)
state_H_1.draw('latex')
```

Out[4]:

$$|00\rangle$$

In [5]:
```
circuit.h(qB[1])
circuit.draw(output='mpl')
```

Out[5]:



In [6]:
```
state_C_1 = Statevector.from_int(0, 2**2)
state_C_1 = state_C_1.evolve(circuit)
state_C_1.draw('latex')
```

Out[6]:

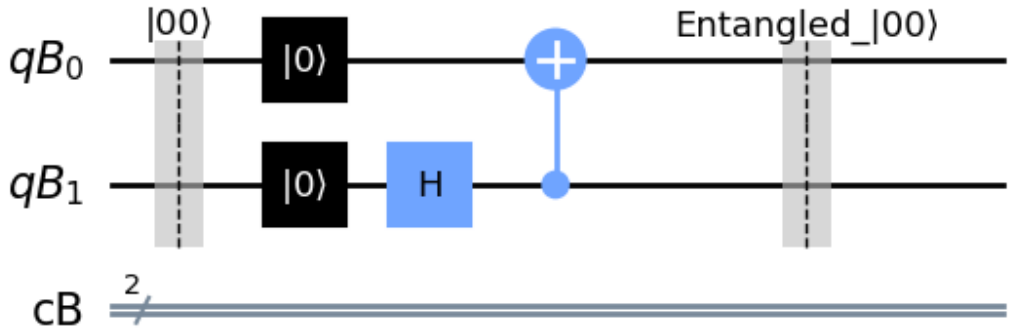
$$\frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

In [7]:
```
circuit.cx(qB[1], qB[0])
circuit.barrier(label = 'Entangled_|00⟩')
```

Out[7]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x24321b6f880>
```

In [8]:
```
circuit.draw(output='mpl')
```

Out[8]:



In [9]:
```
state_E_1 = Statevector.from_int(0, 2**2)
state_E_1 = state_E_1.evolve(circuit)
state_E_1.draw('latex')
```

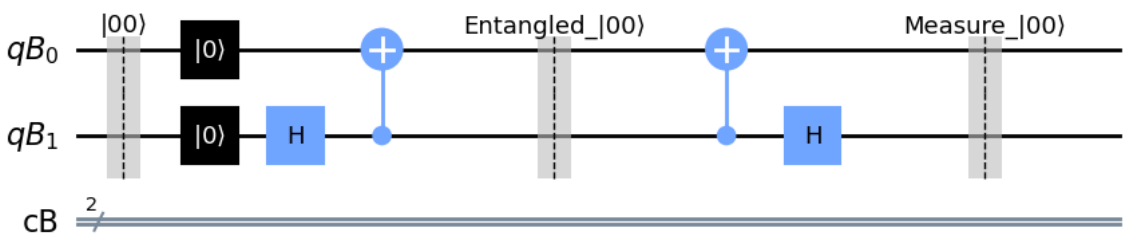
Out[9]:

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

In [10]:
```
circuit.cx(qB[1], qB[0])
circuit.h(qB[1])
```

```
circuit.barrier(label = 'Measure_|00⟩')
circuit.draw(output='mpl')
```

Out[10]:



In [11]:
```
state_M_1 = Statevector.from_int(0, 2**2)
state_M_1 = state_M_1.evolve(circuit)
state_M_1.draw('latex')
```

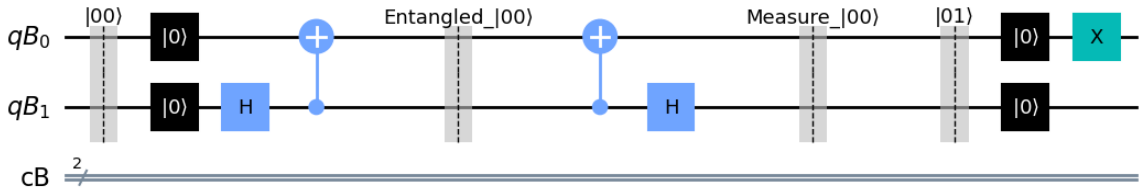
Out[11]:

$$|00\rangle$$

In [ ]:

In [12]:
```
circuit.barrier(qB, label='|01⟩')
circuit.reset(qB)
circuit.x(qB[0])
circuit.draw(output='mpl')
```

Out[12]:



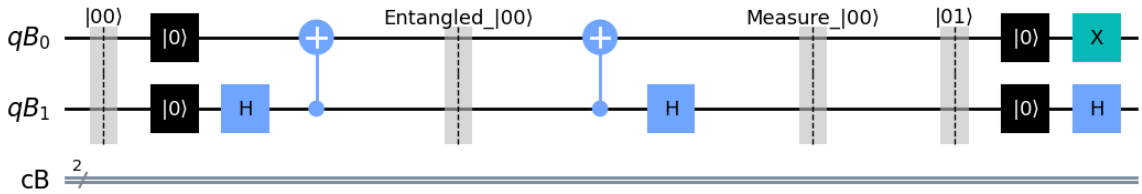
In [13]:
```
state_H_2 = Statevector.from_int(0, 2**2)
state_H_2 = state_H_2.evolve(circuit)
state_H_2.draw('latex')
```

Out[13]:

$$|01\rangle$$

In [14]:
```
circuit.h(qB[1])
circuit.draw(output='mpl')
```

Out[14]:



In [15]:
```
state_C_2 = Statevector.from_int(0, 2**2)
state_C_2 = state_C_2.evolve(circuit)
state_C_2.draw('latex')
```

Out[15]:

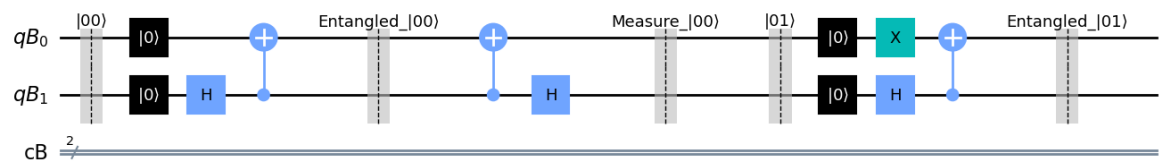
$$\frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|11\rangle$$

In [16]:
```
circuit.cx(qB[1], qB[0])
circuit.barrier(label = 'Entangled_|01⟩')
```

Out[16]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x243222f7160>
```

In [17]:
```
circuit.draw(output='mpl')
```

Out[17]:



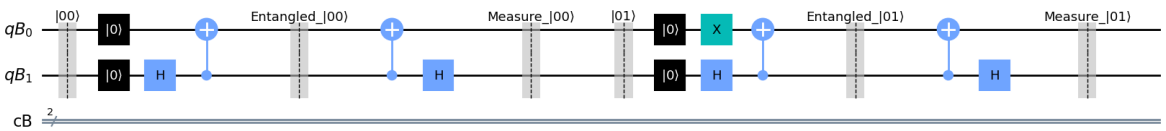
In [18]:
```
state_E_2 = Statevector.from_int(0, 2**2)
state_E_2 = state_E_2.evolve(circuit)
state_E_2.draw('latex')
```

Out[18]:

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

In [19]:
```
circuit.cx(qB[1], qB[0])
circuit.h(qB[1])
circuit.barrier(label = 'Measure_|01⟩')
circuit.draw(output='mpl')
```

Out[19]:



In [20]:
```
state_M_2 = Statevector.from_int(0, 2**2)
state_M_2 = state_M_2.evolve(circuit)
state_M_2.draw('latex')
```

Out[20]:

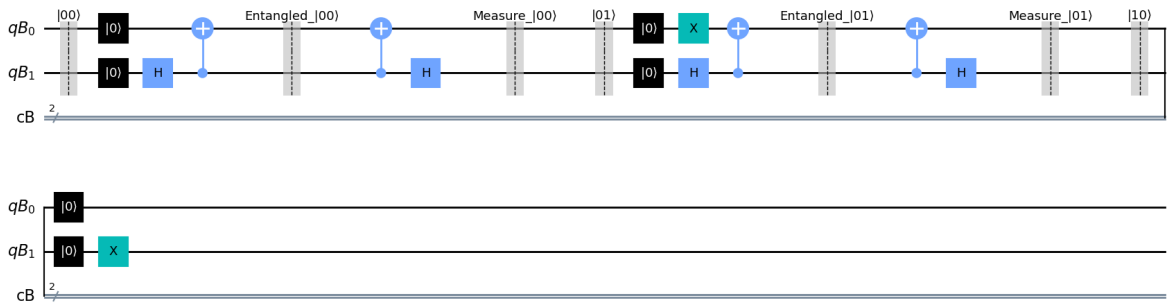
$$|01\rangle$$

In [ ]:
```
```

In [21]:
```
circuit.barrier(qB, label='|10⟩')
circuit.reset(qB)
```

```
circuit.x(qB[1])
circuit.draw(output='mpl')
```

Out[21]:



In [22]:

```
state_H_3 = Statevector.from_int(0, 2**2)
state_H_3 = state_H_3.evolve(circuit)
state_H_3.draw('latex')
```

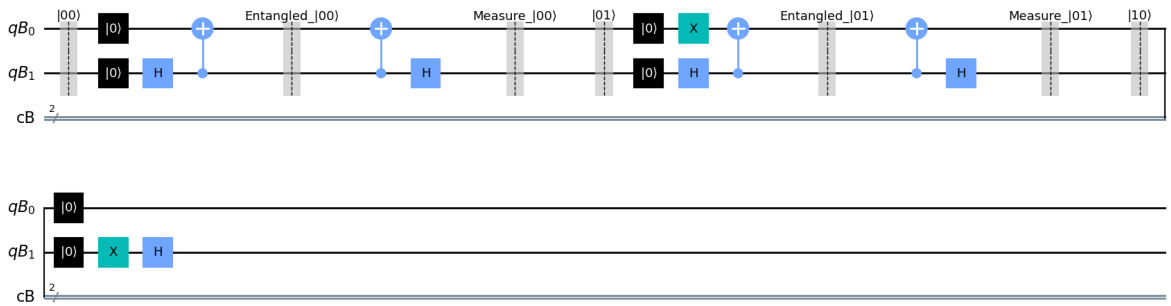
Out[22]:

$$|10\rangle$$

In [23]:

```
circuit.h(qB[1])
circuit.draw(output='mpl')
```

Out[23]:



In [24]:

```
state_C_3 = Statevector.from_int(0, 2**2)
state_C_3 = state_C_3.evolve(circuit)
state_C_3.draw('latex')
```

Out[24]:

$$\frac{\sqrt{2}}{2}|00\rangle - \frac{\sqrt{2}}{2}|10\rangle$$

In [25]:

```
circuit.cx(qB[1], qB[0])
circuit.barrier(label = 'Entangled_|10⟩')
```

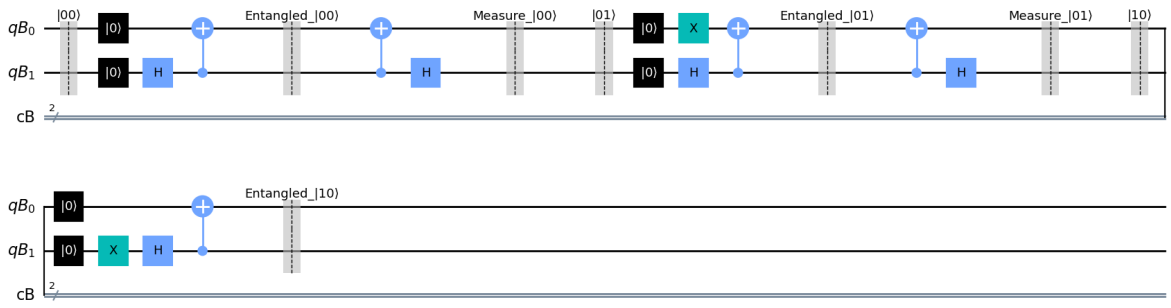
Out[25]:

```
<qiskit.circuit.instructionset.InstructionSet at 0x24324673670>
```

In [26]:

```
circuit.draw(output='mpl')
```

Out[26]:



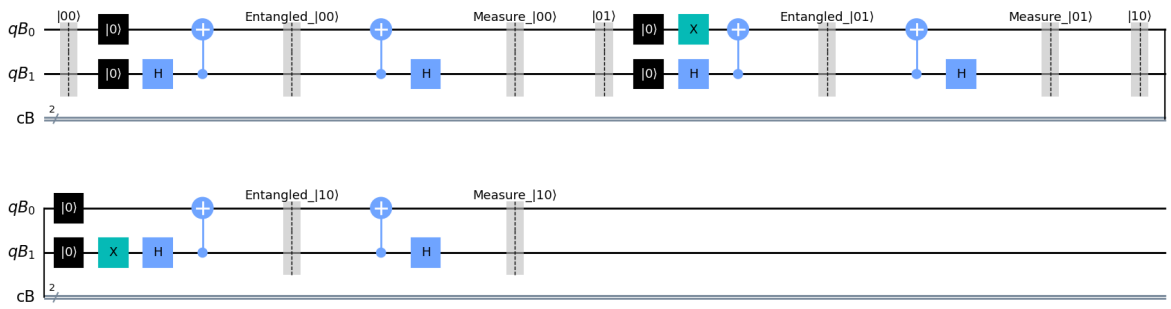
In [27]:
```
state_E_3 = Statevector.from_int(0, 2**2)
state_E_3 = state_E_3.evolve(circuit)
state_E_3.draw('latex')
```

Out[27]:

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

In [28]:
```
circuit.cx(qB[1], qB[0])
circuit.h(qB[1])
circuit.barrier(label = 'Measure_|10⟩')
circuit.draw(output='mpl')
```

Out[28]:



In [29]:
```
state_M_3 = Statevector.from_int(0, 2**2)
state_M_3 = state_M_3.evolve(circuit)
state_M_3.draw('latex')
```

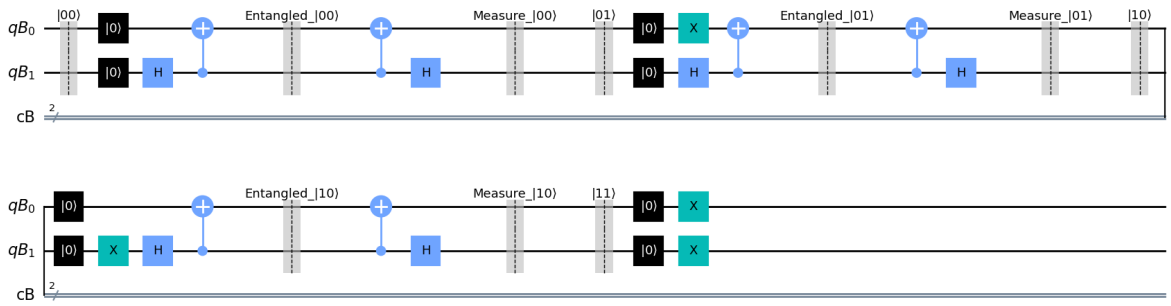
Out[29]:

$$|10\rangle$$

In [ ]:
```

```

In [30]:
```
circuit.barrier(qB, label='|11⟩')
circuit.reset(qB)
circuit.x(qB[0])
circuit.x(qB[1])
circuit.draw(output='mpl')
```

Out[30]:



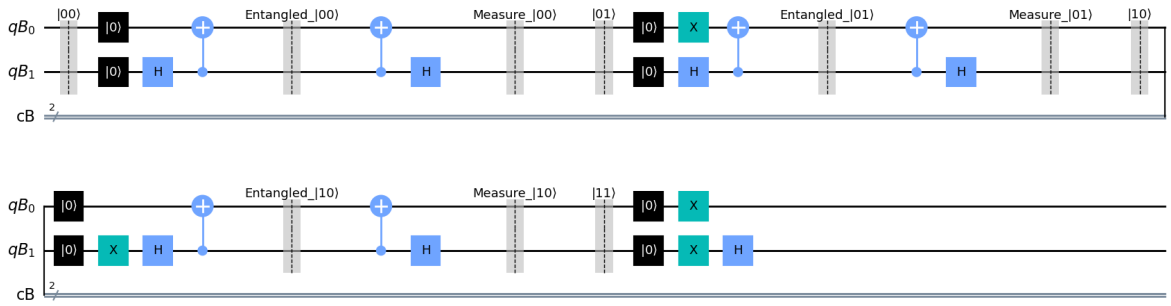
In [31]:
```
state_H_4 = Statevector.from_int(0, 2**2)
state_H_4 = state_H_4.evolve(circuit)
state_H_4.draw('latex')
```

Out[31]:

$$|11\rangle$$

In [32]:
```
circuit.h(qB[1])
circuit.draw(output='mpl')
```

Out[32]:



In [33]:
```
state_C_4 = Statevector.from_int(0, 2**2)
state_C_4 = state_C_4.evolve(circuit)
state_C_4.draw('latex')
```

Out[33]:

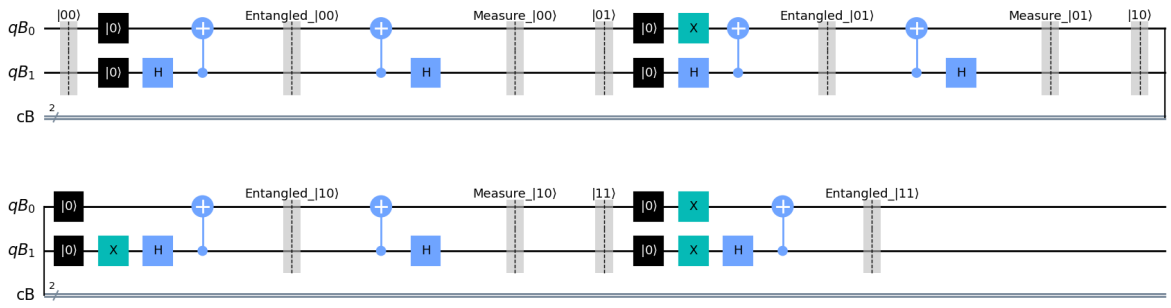
$$\frac{\sqrt{2}}{2}|01\rangle - \frac{\sqrt{2}}{2}|11\rangle$$

In [34]:
```
circuit.cx(qB[1], qB[0])
circuit.barrier(label = 'Entangled_|11⟩')
```

Out[34]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x243222af0d0>
```

In [35]:
```
circuit.draw(output='mpl')
```

Out[35]:



In [36]:

```
state_E_4 = Statevector.from_int(0, 2**2)
state_E_4 = state_E_4.evolve(circuit)
state_E_4.draw('latex')
```

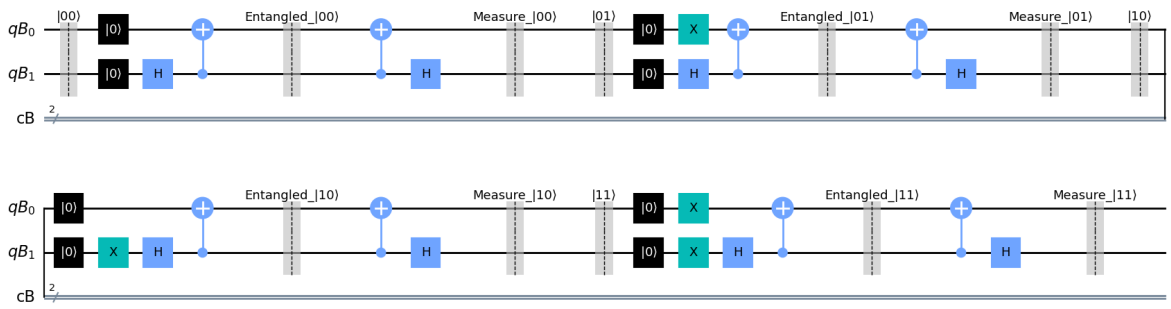
Out[36]:

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

In [37]:

```
circuit.cx(qB[1], qB[0])
circuit.h(qB[1])
circuit.barrier(label = 'Measure_|11⟩')
circuit.draw(output='mpl')
```

Out[37]:



In [38]:

```
state_M_3 = Statevector.from_int(0, 2**2)
state_M_3 = state_M_3.evolve(circuit)
state_M_3.draw('latex')
```

Out[38]:

$$|11\rangle$$

In [ ]:

Conclusion

Fundamental ideas in quantum information science like the Bell circuit and Bell measurement show the strength and potential of quantum entanglement. The Bell circuit demonstrates how two qubits can become maximally entangled by using a controlled-NOT gate and a Hadamard gate. The Bell measurement, on the other hand, shows how to perform a measurement in the Bell basis, which enables one to learn about the correlations between the two qubits, to extract information from an entangled state.

The significance of the Bell circuit and Bell measurement lies in their ability to create and manipulate quantum entanglement, which is a crucial resource for various quantum information processing tasks. The ability to create and measure entangled states is central to quantum teleportation, quantum error correction, and quantum key distribution, all of which are promising applications in the field of quantum information science.

In conclusion, the Bell circuit and Bell measurement are important building blocks for the development of quantum technologies and will continue to be crucial in advancing our understanding and utilization of quantum entanglement.

Experiment 5

Aim

Implementation of Quantum Teleportation circuit in IBM Quantum Composer.

Theory

Quantum teleportation is a fundamental concept in quantum information science that enables the transfer of quantum information from one qubit to another over a distance without actually transmitting the qubit itself. It relies on the phenomenon of quantum entanglement, whereby two or more qubits can become correlated in such a way that the state of one qubit can affect the state of another, even when they are separated by large distances.

The process of quantum teleportation involves three qubits: the input qubit, the sender's qubit, and the receiver's qubit. The goal is to transfer the state of the input qubit to the receiver's qubit without actually transmitting the qubit itself. The sender and receiver share an entangled pair of qubits, which is used to transmit the information about the input qubit from the sender to the receiver.

The quantum teleportation circuit can be implemented using the following steps:

Prepare an entangled pair of qubits in the Bell state:

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle + |1\rangle \otimes |0\rangle) \tag{1}$$

The input qubit to be teleported is prepared in an arbitrary state:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2}$$

The input qubit is entangled with the sender's qubit by applying a CNOT gate with the input qubit as the control and the sender's qubit as the target:

$$|\Psi\rangle = \frac{1}{\sqrt{2}}(\alpha|00\rangle + \alpha|01\rangle + \beta|10\rangle + \beta|11\rangle) \tag{3}$$

A Hadamard gate is applied to the input qubit:

$$|\Psi\rangle = \frac{1}{2}(\alpha|00\rangle + \alpha|01\rangle + \alpha|10\rangle + \alpha|11\rangle + \beta|00\rangle + \beta|01\rangle - \beta|10\rangle - \beta|11\rangle) \tag{4}$$

The input qubit and the sender's qubit are measured in the computational basis, and the results are sent to the receiver over a classical channel. The sender's qubit collapses to either $|0\rangle$ or $|1\rangle$, depending on the measurement outcome.
Based on the measurement results, the receiver applies a Pauli-X gate, a Pauli-Z gate, or both to the receiver's qubit:

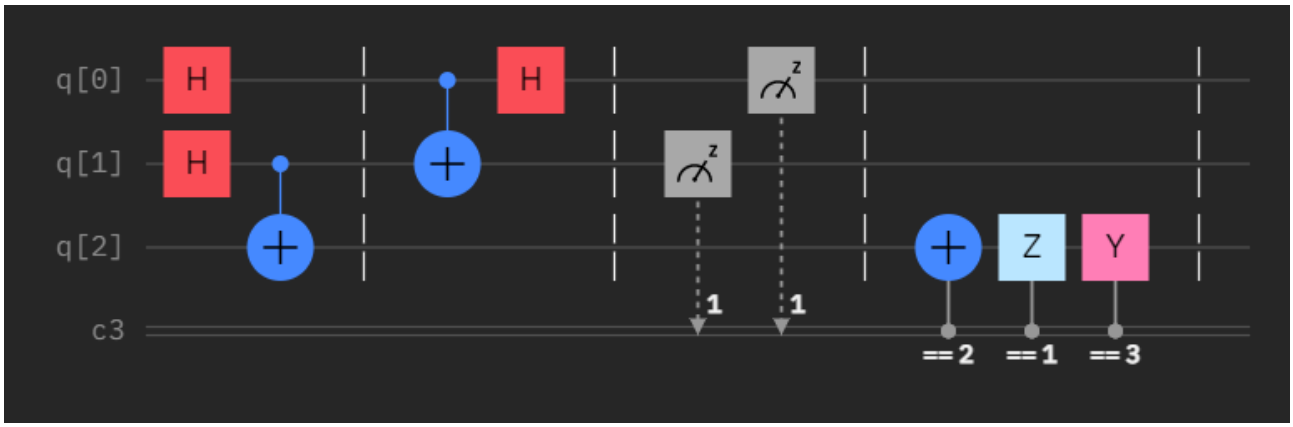
- If the measurement result for the sender's qubit is $|0\rangle$ and the input qubit is $|0\rangle$, no gates are applied to the receiver's qubit.
- If the measurement result for the sender's qubit is $|1\rangle$ and the input qubit is $|0\rangle$, a Pauli-X gate is applied to the receiver's qubit.
- If the measurement result for the sender's qubit is $|0\rangle$ and the input qubit is $|1\rangle$, a Pauli-Z gate is applied to the receiver's qubit.
- If the measurement result for the sender's qubit is $|1\rangle$ and the input qubit is $|1\rangle$, both a Pauli-X gate and a Pauli-Z gate are applied to the receiver's qubit.

The receiver's qubit now has the same state as the original input qubit:

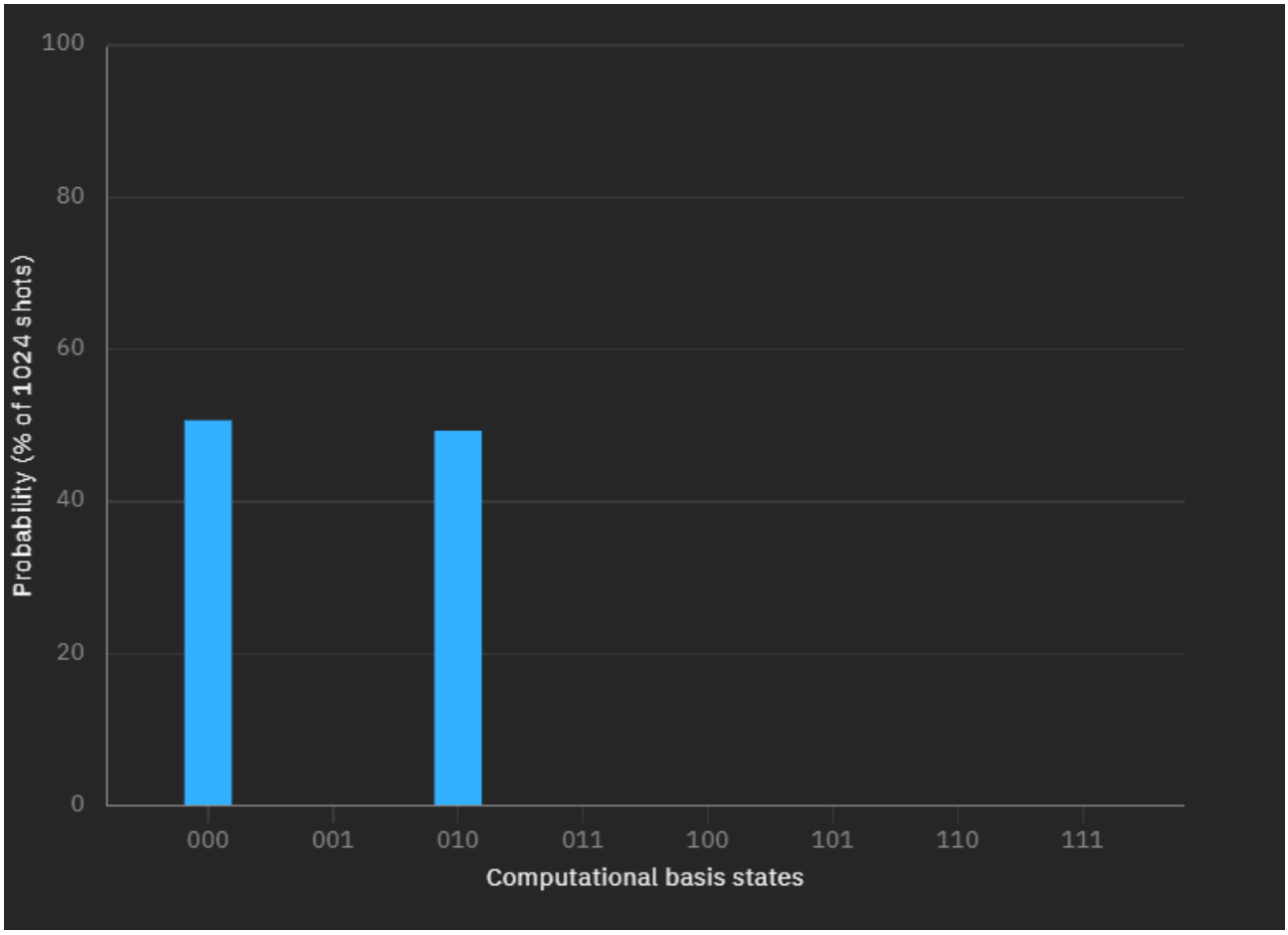
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{5}$$

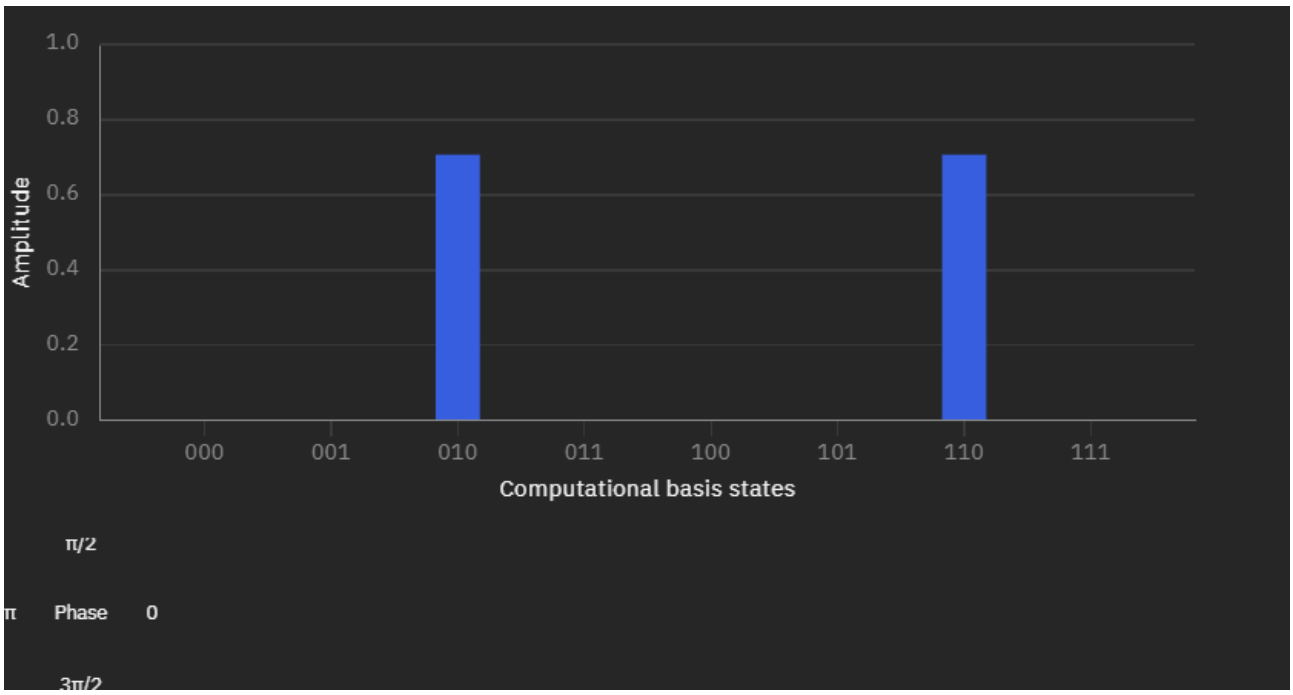
Lab Exercise

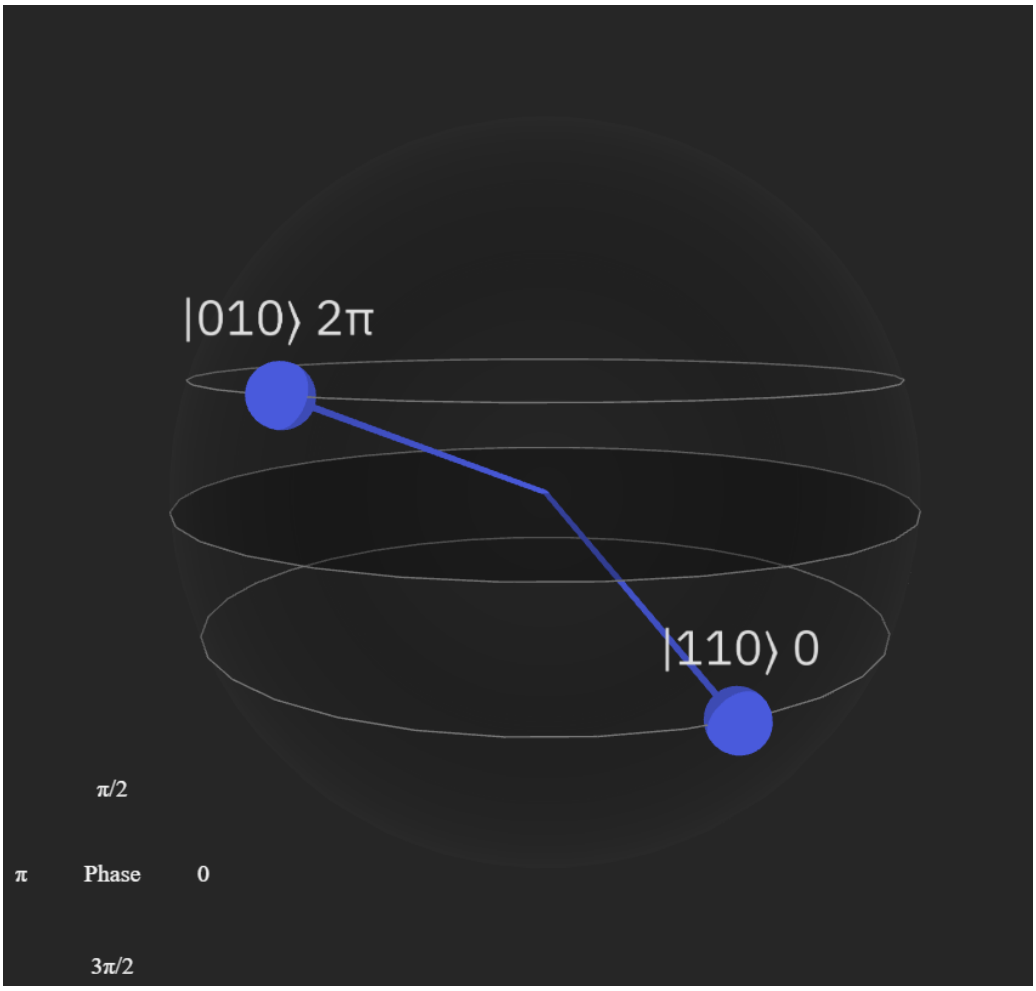
IBM Composer Circuit:



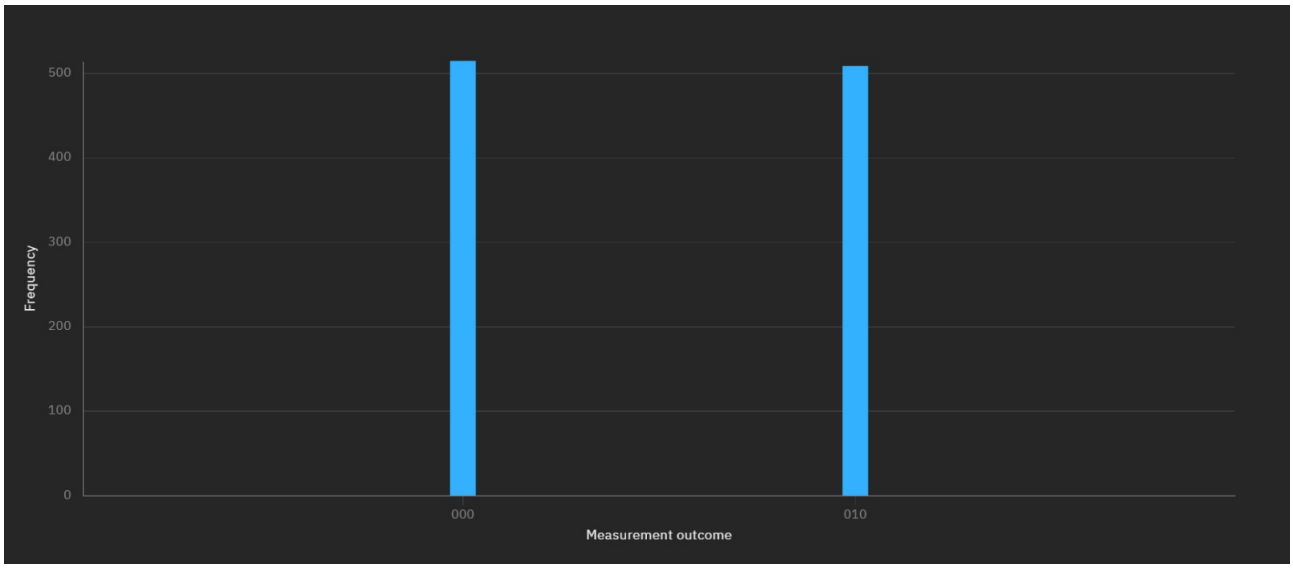
Results:




1.0
0.8
0.6
0.4
0.2
0.0
Amplitude
000
001
010
011
100
101
110
111
Computational basis states
π/2
π
Phase
0
3π/2


|010⟩ 2π
|110⟩ 0
π/2
π
Phase
0
3π/2

Job Results:



Conclusion

In conclusion, quantum teleportation is a fascinating idea in quantum information science that enables the transmission of quantum information over a distance between two qubits without actually sending the qubit itself. There are numerous uses for this intricate procedure in quantum computing, quantum communication, and quantum cryptography. It is based on the entanglement and measurement concepts.

A crucial step towards the realisation of quantum communication networks is the creation of the quantum teleportation circuit employing qubits. Quantum teleportation will be more crucial as quantum computer technology develops since it allows for the distant manipulation of quantum states. The secure transmission of quantum information across great distances is made possible by quantum teleportation, which adds an extra degree of security to quantum communication protocols.

Overall, the development of quantum teleportation is a significant advancement in the field of quantum information science, and it has the potential to revolutionize the way we communicate and process information in the future.

Experiment 6

Aim

Implementation of Quantum Fourier Transform circuit in IBM Quantum Composer.

Theory

The Quantum Fourier Transform (QFT) is a powerful tool in quantum computing, used for tasks such as quantum phase estimation and quantum algorithms like Shor's algorithm. The QFT is the quantum analogue of the classical Fourier transform, which is used to analyze signals in classical information processing.

The QFT operates on a quantum state $|\psi\rangle = \sum_{x=0}^{2^n-1} c_x|x\rangle$, where n is the number of qubits and c_x are complex amplitudes that describe the state of the qubits. The QFT maps this input state to a new state $|\tilde{\psi}\rangle = \sum_{y=0}^{2^n-1} \tilde{c}_y|\tilde{y}\rangle$, where $\tilde{c}_y$ are new complex amplitudes and $\tilde{y}$ is the Fourier transform of x.

The QFT circuit is composed of several fundamental quantum gates, including the Hadamard gate and the controlled phase gate. The Hadamard gate is a single-qubit gate that transforms the state $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and the state $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. The controlled phase gate is a two-qubit gate that introduces a phase shift of $e^{i\theta}$ on the target qubit if the control qubit is in the state $|1\rangle$, where θ is a constant.

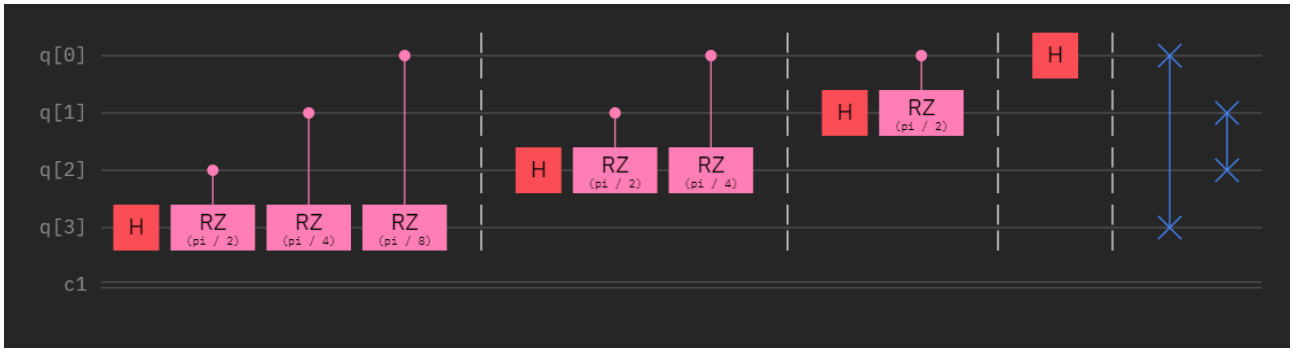
To implement the QFT circuit on n qubits, we can follow the following steps:

- Initialize n qubits in the state $|x\rangle$, where x is an integer in binary representation.
- Apply a Hadamard gate to the first qubit.
- Apply a controlled phase gate between the first qubit and the second qubit with a phase of $e^{i\frac{\pi}{2}}$.
- Apply a controlled phase gate between the first qubit and the third qubit with a phase of $e^{i\frac{\pi}{4}}$.
- Continue applying controlled phase gates between the first qubit and each of the remaining qubits, with phases of $e^{i\frac{\pi}{2^k}}$ for the kth qubit.
- Apply a Hadamard gate to the second qubit, and continue applying Hadamard gates to each of the remaining qubits in succession.
- Swap the i^{th} qu-bit with $(n-i)^{th}$ qu-bit.

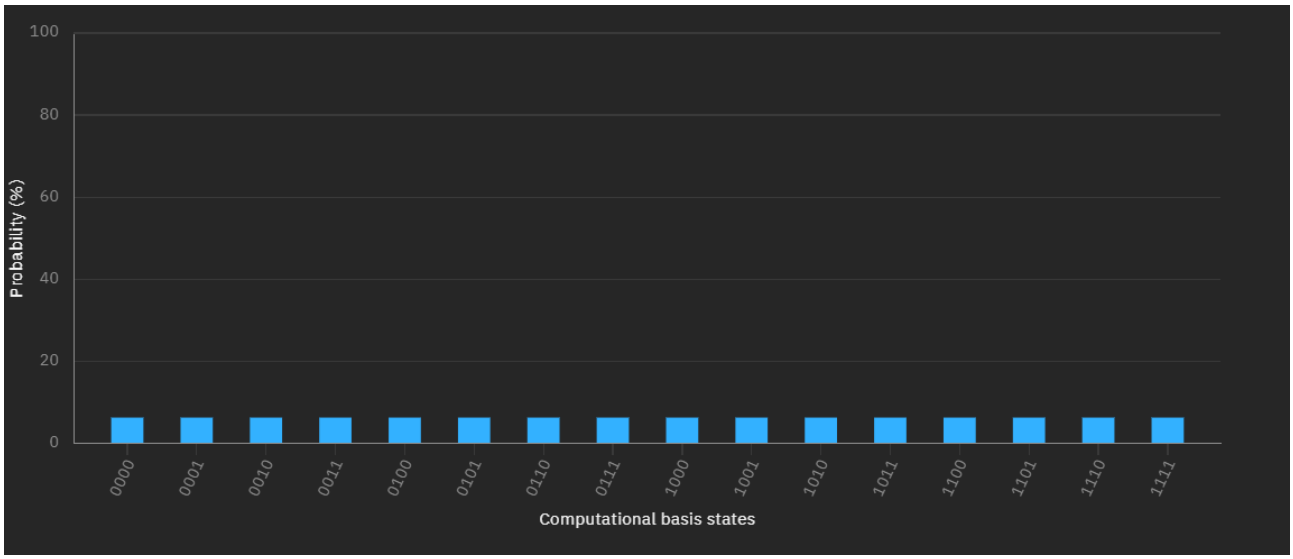
This completes the QFT circuit, which maps the input state $|x\rangle$ to the output state $|\tilde{x}\rangle$, where $\tilde{x}$ is the Fourier transform of x. The QFT circuit can be implemented on real quantum hardware, such as the IBM Quantum systems, using quantum gates and circuits designed for this purpose.

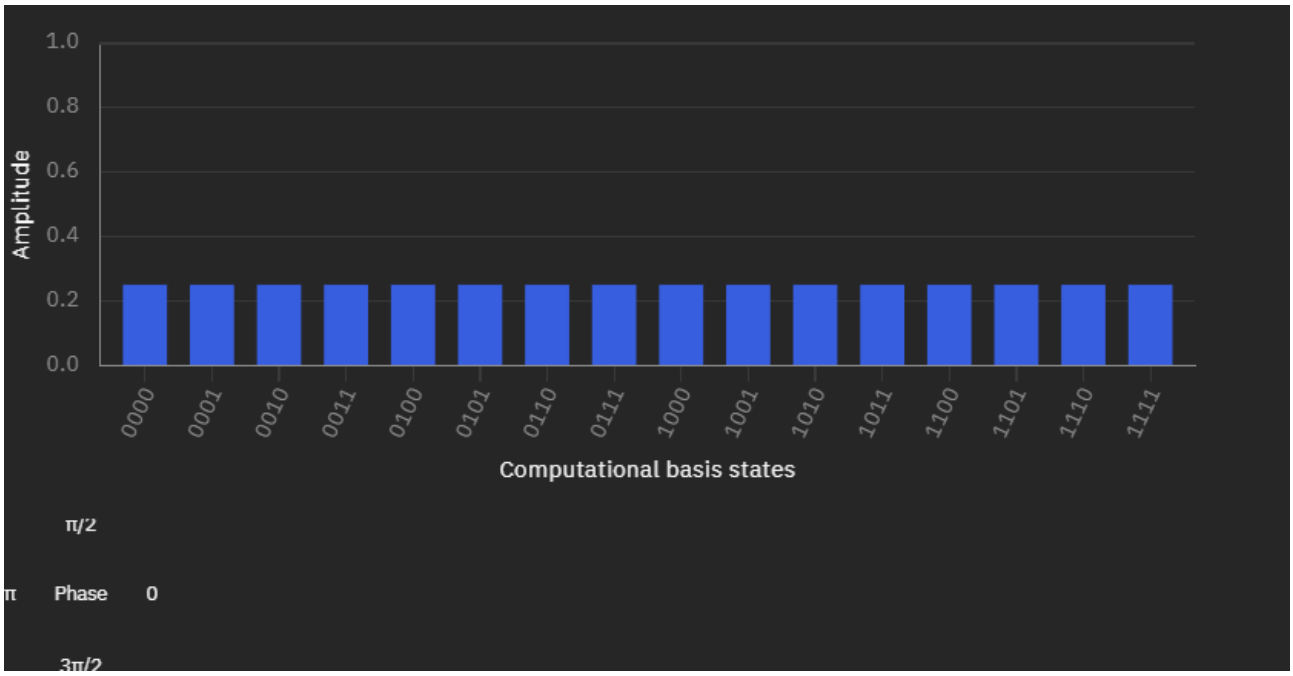
Lab Exercise

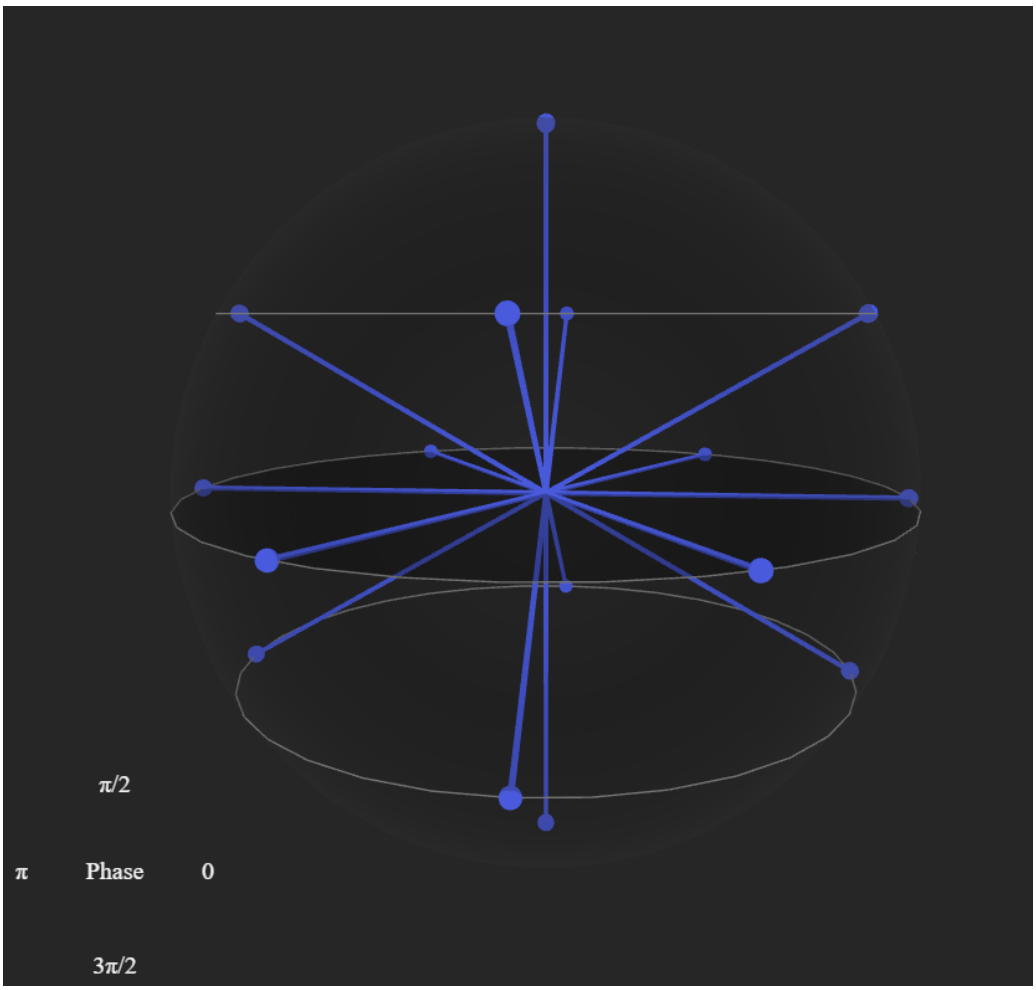
IBM Composer Circuit:



Results:


100
80
60
40
20
0
Probability (%)
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Computational basis states


1.0
0.8
0.6
0.4
0.2
0.0
Amplitude
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Computational basis states
π/2
π
Phase
0
3π/2



Conclusion

In conclusion, the Quantum Fourier Transform (QFT) circuit is a fundamental quantum computing circuit that plays an important role in many quantum algorithms, including Shor's algorithm for computing large numbers and the quantum phase estimation algorithm for estimating the eigenvalues of unitary operators. QFT circuit is used to transform quantum space from the time domain to the frequency domain, which can be useful for analyzing and manipulating quantum systems.

Implementing a QFT circuit in IBM Quantum Composer provides researchers and developers with a convenient and intuitive way to design, simulate, and test a QFT circuit in a variety of quantum systems, including quantum simulators and real quantum hardware. The ability to implement the QFT circuit on IBM Quantum systems is an important step towards realizing the potential of quantum computing to solve complex problems in a variety of fields, including cryptography, chemistry, and optimization.

Overall, the Quantum Fourier transform circuit is an important building block in the field of quantum computing, and the availability of tools such as IBM Quantum Composer makes it easy for researchers and programmers to study and experiment with this important circuit.