Programming a Quantum Computer with Qiskit - IBM SDK

December 6, 2022

1 Programming a Quantum Computer with Qiskit - IBM SDK

1.1 Task 1 : Fundamentals of Quantum Computation

[1]:
```
#importing basic libraries...

import numpy as np
import math as mth
```

1.1.1 1.1 Getting started : Basic Arithmatic Operations with Complex Numbers

Complex numbers are always of the form:

$$\alpha = a + bi, \quad where\ a, b \in \mathbb{R} \tag{1}$$

- In python the imaginary parts have a syntax for a real number joined with the character j.
- For example, the the comple number iota, 'i' is used as 1j.
- In the output, there is an auto-representation of the compelx number in its standard notation.
- basic mathematical operations like, multiplication and addition just passes through like their real counterparts.

[2]:
```
#basic preliminary operations...

print("Iota square is: ", 1j*1j) #...........basic complex multiplication and␣
↪their auto representation.

z1 = 4 + 8j
z2 = 5 - 2j

print("The real part of the complex number z1 is: ", np.real(z1)) #.....getting␣
↪the real part of a complex number.
print("The imaginary part of the complex number z2 is: ", np.imag(z2)) #getting␣
↪the imaginary part of a complex no.
```

```
Iota square is:  (-1+0j)
The real part of the complex number z1 is:  4.0
The imaginary part of the complex number z2 is:  -2.0
```

1.1.2 1.2 Complex conjugation

- The conjugate of a complex number z is often denotes as z^* or $\bar{z}$, in standard literature.
- In complex conjugation, we basically get the reflection image of a complex number with respect to the imaginary axis in the complex plane, i.e., the Y-axis.
- In simple terms, the sign of the imaginary part is reversed.

[3]:
```
#complex conjugation...

print("The conjugate of z2 is: ", np.conj(z2))
```

```
The conjugate of z2 is:  (5+2j)
```

1.1.3 1.3 Norms or Absolute Values

$$||z|| = \sqrt{zz^*} = \sqrt{|z|^2}, \tag{2}$$
$$||w|| = \sqrt{ww^*} = \sqrt{|w|^2}, \tag{3}$$

[4]:
```
print("The absolute value of z1 is: ", np.abs(z1))
```

```
The absolute value of z1 is:  8.94427190999916
```

[5]:
```
print("The absolute value of z2 is: ", np.abs(z2))
```

```
The absolute value of z2 is:  5.385164807134504
```

1.1.4 1.4 Row Vectors, Column Vectors, and Bra-Ket Notation

$$\text{Column Vector: } \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \qquad \text{Row Vector: } \begin{pmatrix} a_1, & a_2, & \cdots, & a_n \end{pmatrix} \tag{4}$$

[6]:
```
row_vec = np.array([3, z1, z2])
```

[7]:
```
row_vec #here, we can observe a uniform conversion of all other numbers in␣
↪complex notation.
```

[7]:
```
array([3.+0.j, 4.+8.j, 5.-2.j])
```

[8]:
```
col_vec = np.array([[2], [z2], [z1]])
```

[9]:
```
col_vec
```

[9]:
```
array([[2.+0.j],
       [5.-2.j],
       [4.+8.j]])
```

Row vectors in quantum mechanics are also called **bra-vectors**, and are denoted as follows:

$$\langle A| = \begin{pmatrix} a_1, & a_2, \cdots, & a_n \end{pmatrix} \tag{5}$$

Column vectors are also called **ket-vectors** in quantum mechanics and are denoted as follows:

$$|B\rangle = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{6}$$

In general, if we have a column vector, i.e. a ket-vector:

$$|A\rangle = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \tag{7}$$

the corresponding bra-vector:

$$\langle A| = \begin{pmatrix} a_1^*, & a_2^*, & \cdots, & a_n^* \end{pmatrix} \tag{8}$$

1.1.5 1.5 Inner Product

$$\langle A| = \begin{pmatrix} a_1, & a_2, & \cdots, & a_n \end{pmatrix}, \qquad |B\rangle = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{9}$$

Taking the inner product of $\langle A|$ and $|B\rangle$ gives the following:

$$\begin{aligned} \langle A|B\rangle &= \begin{pmatrix} a_1, & a_2, & \cdots, & a_n \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} && (10) \\ &= a_1 b_1 + a_2 b_2 + \cdots + a_n b_n && (11) \\ &= \sum_{i=1}^{n} a_i b_i && (12) \end{aligned}$$

[10]:
```
# Define the 4x1 matrix version of a column vector:
A = np.array([[4], [2], [z2], [z1]])
```

[11]:
```
# Define B as a 1x4 matrix:
B = np.array([3, 6, z2, z1])
```

[12]:
```
# Compute <B|A>:
# we can compute the iner product using the dot function provided by numpy:

BA = np.dot(B,A) # remark: The inner product AB is different from BA, and also␣
↪might not always lake sense.
print("The the inner product of B and A are: ", BA)
```

```
The the inner product of B and A are:  [-3.+44.j]
```

1.1.6 1.6 Matrices

$$M = \begin{pmatrix} 2-i & -3 \\ -5i & 2 \end{pmatrix} \tag{13}$$

[13]:
```
M = np.array([[2-1j, -3], [-5j, 2]])
M
```

[13]:
```
array([[ 2.-1.j, -3.+0.j],
       [-0.-5.j,  2.+0.j]])
```

[14]:
```
# we also have the matrix method to generate a matrix using numpy:

N = np.matrix([[2-1j, -3], [-5j, 2]])
N #.........it gives the exact same result as the array method, when a matrix␣
↪result is intended.
```

[14]:
```
matrix([[ 2.-1.j, -3.+0.j],
        [-0.-5.j,  2.+0.j]])
```

Hermitian conjugates are given by taking the conjugate transpose of the matrix

[15]:
```
N.H #remark: matrix created using array method will not support the hermitian␣
↪operator .H
```

[15]:
```
matrix([[ 2.+1.j, -0.+5.j],
        [-3.-0.j,  2.-0.j]])
```

1.1.7 1.7 Tensor Products of Matrices

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \otimes \begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} a\begin{pmatrix} x & y \\ z & w \end{pmatrix} & b\begin{pmatrix} x & y \\ z & w \end{pmatrix} \\ c\begin{pmatrix} x & y \\ z & w \end{pmatrix} & d\begin{pmatrix} x & y \\ z & w \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ax & ay & bx & by \\ az & aw & bz & bw \\ cx & cy & dx & dy \\ cz & cw & dz & dw \end{pmatrix} \tag{14}$$

[16]:
```
np.kron(M,M) #remark: the tensor product operator .kron works for both .array␣
↪and .matrix matrices.
```

[16]:
```
array([[  3. -4.j,  -6. +3.j,  -6. +3.j,   9. -0.j],
       [ -5.-10.j,   4. -2.j,   0.+15.j,  -6. +0.j],
       [ -5.-10.j,   0.+15.j,   4. -2.j,  -6. +0.j],
       [-25. +0.j,   0.-10.j,   0.-10.j,   4. +0.j]])
```

1.2 Task 2 : Qubits, Bloch Sphere and Basis States

[17]:
```
import qiskit
qiskit.__qiskit_version__
```

[17]:
```
{'qiskit-terra': '0.22.2', 'qiskit-aer': '0.11.1', 'qiskit-ignis': None,
'qiskit-ibmq-provider': '0.19.2', 'qiskit': '0.39.2', 'qiskit-nature': None,
'qiskit-finance': None, 'qiskit-optimization': None, 'qiskit-machine-learning':
None}
```

1.2.1 2.1 Qubits

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad \text{where } \sqrt{\langle\psi|\psi\rangle} = 1. \tag{15}$$

- Think of Qubit as an Electron:

$$\text{spin-up}: \ |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{16}$$

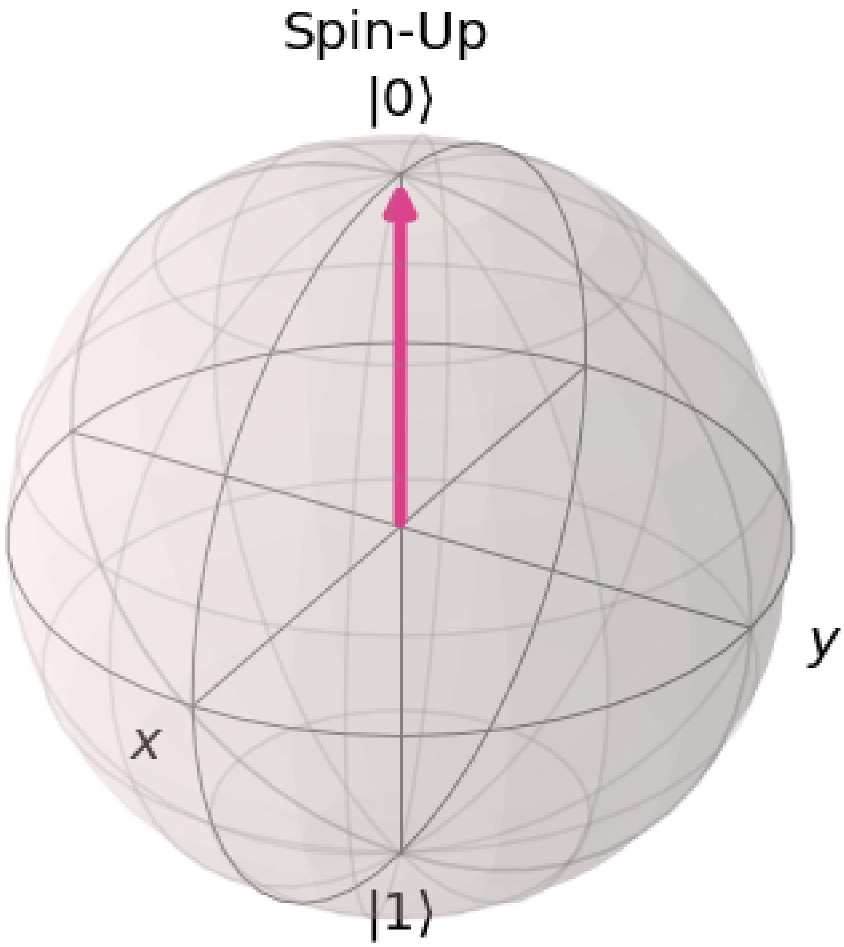
$$\text{spin-down}: \ |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{17}$$

[18]:
```
from qiskit import *
```

Another representation is via **Bloch Sphere** :

[19]:
```
from qiskit.visualization import plot_bloch_vector
plot_bloch_vector([0,0,1], title='Spin-Up')
```

[19]:



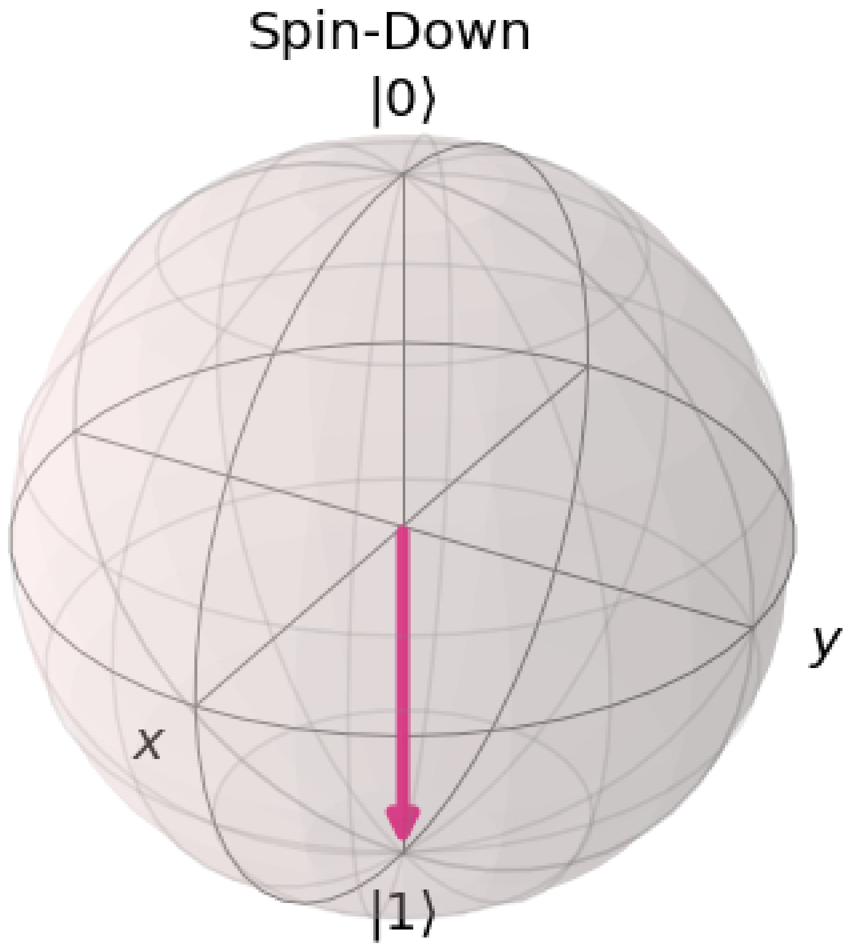
1.2.2 2.2 Spin + / -

$$
\text{spin} + : \ |+\rangle = \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \tag{18}
$$

$$
\text{spin - : } \ |-\rangle = \begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \tag{19}
$$

[20]:
```
plot_bloch_vector([0,0,-1], title='Spin-Down')
```

[20]:



1.2.3 2.3 Basis States

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{20}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{21}$$

Preapring other states from Basis States:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \tag{22}$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \tag{23}$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \tag{24}$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \tag{25}$$

[21]:
```
ket_zero = np.array([[1], [0]])
ket_one = np.array([[0], [1]])
```

[22]:
```
ket_zero_one = np.kron(ket_zero, ket_one)
ket_zero_one
```

[22]:
```
array([[0],
       [1],
       [0],
       [0]])
```

1.3 Task 3 : Quantum Gates and Quantum Circuits

[23]:
```
from qiskit import *
from qiskit.visualization import plot_bloch_multivector
import pylatexenc
```

1.3.1 3.1 Pauli Matrices

$$I = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}, \quad X = \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}, \quad Y = \begin{pmatrix}0 & i\\-i & 0\end{pmatrix}, \quad Z = \begin{pmatrix}1 & 0\\0 & -1\end{pmatrix} \tag{26}$$

1.3.2 3.2 X-gate

The X-gate is represented by the Pauli-X matrix:

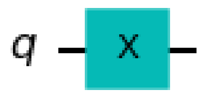
$$X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

Effect a gate has on a qubit:

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

[24]:
```
# Let's do an X-gate on a |0> qubit

qc = QuantumCircuit(1)
qc.x(0)
qc.draw('mpl')
```

[24]:



[25]:
```
# Let's see the result

backend = Aer.get_backend('statevector_simulator')
out = execute(qc, backend).result().get_statevector()
print(out)
```

```
Statevector([0.+0.j, 1.+0.j],
            dims=(2,))
```

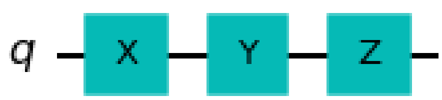
1.3.3 3.3 Z & Y-Gate

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

$$Y = -i|0\rangle\langle 1| + i|1\rangle\langle 0| \qquad Z = |0\rangle\langle 0| - |1\rangle\langle 1|$$

[26]:
```
# Do Y-gate on qubit 0:
qc.y(0)
# Do Z-gate on qubit 0:
qc.z(0)

qc.draw('mpl')
```

[26]:



1.3.4 3.4 Hadamard Gate

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

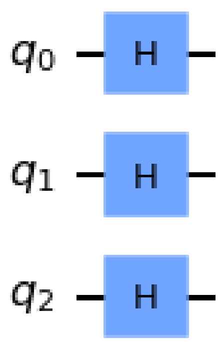
We can see that this performs the transformations below:

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

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

[27]:
```python
# we create circuit with three qubit:
qc = QuantumCircuit(3)

# then apply H-gate to each qubit:
for qubit in range(3):
    qc.h(qubit)

# See the circuit:
qc.draw('mpl')
```

[27]:



1.3.5 3.5 Identity Gate

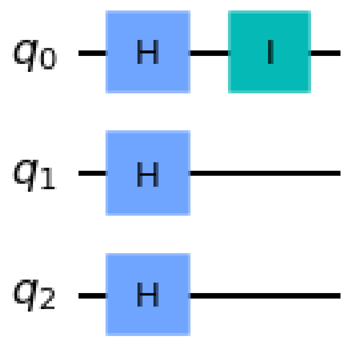
$$I = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$

$$I = XX$$

[28]:

```
qc.i(0) # we basically added an extra identity gate to the zero-cubit from our␣
↪exisitng quantum circuit.
qc.draw('mpl')
```

[28]:



** Other Gates are: S-gate , T-gate, U-gate, which have notations and properties of their own but mathematically can be expresse in similar way.

1.4 Task 4 : Multiple Qubits, Entanglement

1.4.1 4.1 Multiple Qubits

The state of two qubits :

$$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$

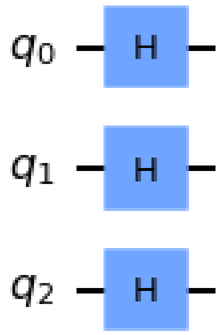
[29]:

```
from qiskit import *
```

[30]:

```
qc = QuantumCircuit(3) # we again make a quantum circuit with three qubits.
# Apply H-gate to each qubit:
for qubit in range(3):
    qc.h(qubit)
# See the circuit:
qc.draw('mpl')
```

[30]:



Each qubit is in the state $|+\rangle$, so we should see the vector:

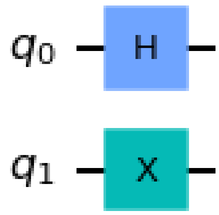
$$|{+}{+}{+}\rangle = \frac{1}{\sqrt{8}}\begin{bmatrix}1\\1\\1\\1\\1\\1\\1\\1\end{bmatrix}$$

[31]:
```
# Let's see the result
backend = Aer.get_backend('statevector_simulator')
out = execute(qc, backend).result().get_statevector()
print(out)
```

```
Statevector([0.35355339+0.j, 0.35355339+0.j, 0.35355339+0.j,
             0.35355339+0.j, 0.35355339+0.j, 0.35355339+0.j,
             0.35355339+0.j, 0.35355339+0.j],
            dims=(2, 2, 2))
```

[32]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.draw('mpl')
```

[32]:



$$X|q_1\rangle \otimes H|q_0\rangle = (X \otimes H)|q_1 q_0\rangle$$

The operation looks like this:

$$X \otimes H = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \tfrac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

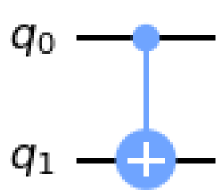
Which we can then apply to our 4D statevector $|q_1 q_0\rangle$. You will often see the clearer notation:

$$X \otimes H = \begin{bmatrix} 0 & H \\ H & 0 \end{bmatrix}$$

1.4.2 4.2 C-Not Gate

[33]:
```
# we create circuit with two qubit
qc = QuantumCircuit(2)
# Apply CNOT
qc.cx(0, 1)
# See the circuit:
qc.draw('mpl')
```

[33]:



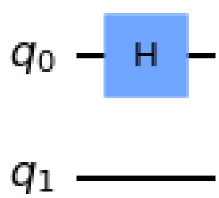
Classical truth table of C-Not gate:

Input (t,c)	Output (t,c)
00	00
01	11
10	10
11	01

1.4.3 4.3 Entanglement

[34]:
```
#create two qubit circuit
qc = QuantumCircuit(2)
# Apply H-gate to the first:
qc.h(0)
qc.draw('mpl')
```

[34]:



[35]:
```
# Let's see the result:
backend = Aer.get_backend('statevector_simulator')
final_state = execute(qc, backend).result().get_statevector()
print(final_state)
```

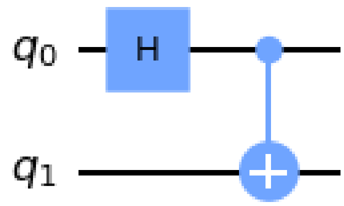
```
Statevector([0.70710678+0.j, 0.70710678+0.j, 0.        +0.j,
             0.        +0.j],
            dims=(2, 2))
```

Quantum System Sate is:

$$|0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$$

[36]:
```
qc = QuantumCircuit(2)
# Apply H-gate to the first:
qc.h(0)
# Apply a CNOT:
qc.cx(0, 1)
qc.draw('mpl')
```

[36]:



[37]:
```
# Let's see the result:
backend = Aer.get_backend('statevector_simulator')
final_result = execute(qc, backend).result().get_statevector()
print(final_result)
```

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

We see we have this final state (**Bell State**):

$$\mathrm{CNOT}|0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

1.5 Task 5 : Bernstein-Vazirani Algorithm

A black-box function f, which takes as input a string of bits (x), and returns either 0 or 1, that is:

$$f(\{x_0, x_1, x_2, ...\}) \rightarrow 0 \text{ or } 1 \text{ where } x_n \text{ is } 0 \text{ or } 1$$

The function is guaranteed to return the bitwise product of the input with some string, s.

In other words, given an input x, $f(x) = s \cdot x \,(\bmod 2) = \ x_0 * s_0 + x_1 * s_1 + x_2 * s_2 + ... \ \bmod 2$

The quantum Bernstein-Vazirani Oracle:

1. Initialise the inputs qubits to the $|0\rangle^{\otimes n}$ state, and output qubit to $|-\rangle$.
2. Apply Hadamard gates to the input register
3. Query the oracle
4. Apply Hadamard gates to the input register
5. Measure

1.5.1 5.1 Example Two Qubits:

The register of two qubits is initialized to zero:

$$|\psi_0\rangle = |00\rangle$$

Apply a Hadamard gate to both qubits:

$$|\psi_1\rangle = \frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

For the string $s = 11$, the quantum oracle performs the operation:

$$|x\rangle \xrightarrow{f_s} (-1)^{x\cdot 11}|x\rangle.$$

$$|\psi_2\rangle = \frac{1}{2}\left((-1)^{00\cdot 11}|00\rangle + (-1)^{01\cdot 11}|01\rangle + (-1)^{10\cdot 11}|10\rangle + (-1)^{11\cdot 11}|11\rangle\right)$$

$$|\psi_2\rangle = \frac{1}{2}\left(|00\rangle - |01\rangle - |10\rangle + |11\rangle\right)$$

Apply a Hadamard gate to both qubits:

$$|\psi_3\rangle = |11\rangle$$

Measure to find the secret string $s = 11$

[38]:
```
from qiskit import *
%matplotlib inline
from qiskit.tools.visualization import plot_histogram
```

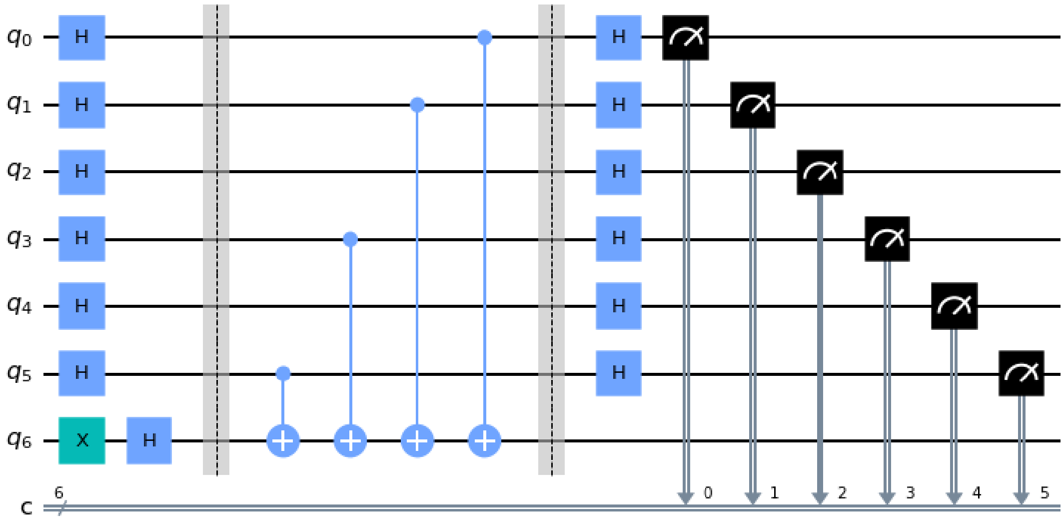
[39]:
```
s = 101011
```

[40]:
```
qc = QuantumCircuit(6+1, 6)
qc.h([0, 1, 2, 3, 4, 5]) #..........Step 1.

qc.x(6)  #.........................Step 2.
qc.h(6)
qc.barrier()

qc.cx(5, 6)  #.....................Step 3.
qc.cx(3, 6)
qc.cx(1, 6)
qc.cx(0, 6)

qc.barrier()  #....................Step 4.

qc.h([0, 1, 2, 3, 4, 5])  #.........Step 5.

qc.measure([0, 1, 2, 3, 4, 5], [0, 1, 2, 3, 4, 5]) #.........Step 6.
```

[40]: <qiskit.circuit.instructionset.InstructionSet at 0x1317fb010>

[41]:
```
qc.draw('mpl')
```

[41]:



[42]:
```
#Let's check the result:

simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, backend=simulator, shots=1).result()
counts = result.get_counts()
print(counts)
```

```
{'101011': 1}
```

Here, we can verify that, we got the string **s** as **101011**, which is in accordance with the input given.