Lab01_QuantumCircuits_MIGUEL_ANGEL_NAVARRO_ARENAS

April 20, 2022

Lab 1 Quantum Circuits

MIGUEL ÁNGEL NAVARRO ARENAS

Prerequisite - Qiskit basics - Ch.1.2 The Atoms of Computation

Other relevant materials - Access IBM Quantum Systems - IBM Quantum Systems Configuration - Transpile - IBM Quantum account - Quantum Circuits

[3]:
```
from qiskit import *
from qiskit.visualization import plot_histogram
import numpy as np
```

Part 1: Classical logic gates with quantum circuits

Goal

```
<p style=" padding: 0px 0px 10px 10px;
          font-size:16px;">Create quantum circuit functions that can compute the XOR, AND, NAN
```

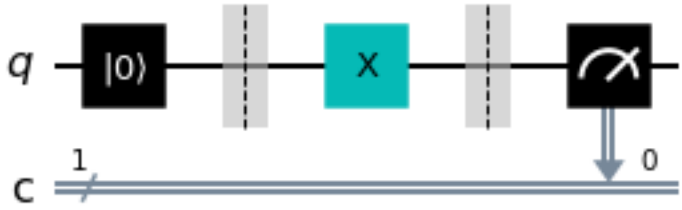
An implementation of the `NOT` gate is provided as an example.

[4]:
```
def NOT(inp):
    """An NOT gate.

    Parameters:
        inp (str): Input, encoded in qubit 0.

    Returns:
        QuantumCircuit: Output NOT circuit.
        str: Output value measured from qubit 0.
    """

    qc = QuantumCircuit(1, 1) # A quantum circuit with a single qubit and a␣
↪single classical bit
    qc.reset(0)

    # We encode '0' as the qubit state |0⟩, and '1' as |1⟩
    # Since the qubit is initially |0⟩, we don't need to do anything for an␣
↪input of '0'
```

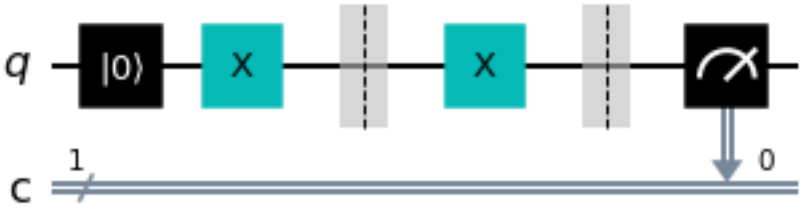
```
    # For an input of '1', we do an x to rotate the |0> to |1
    if inp=='1':
        qc.x(0)

    # barrier between input state and gate operation
    qc.barrier()

    # Now we've encoded the input, we can do a NOT on it using x
    qc.x(0)

    #barrier between gate operation and measurement
    qc.barrier()

    # Finally, we extract the |0>/|1> output of the qubit and encode it in the␣
↪bit c[0]
    qc.measure(0,0)
    qc.draw('mpl')

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to␣
↪get it
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

[5]:
```
## Test the function
for inp in ['0', '1']:
    qc, out = NOT(inp)
    print('NOT with input',inp,'gives output',out)
    display(qc.draw())
    print('\n')
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning:
scipy._lib.messagestream.MessageStream size changed, may indicate binary
incompatibility. Expected 56 from C header, got 64 from PyObject

NOT with input 0 gives output 1
```




```
NOT with input 1 gives output 0
```



XOR gate

Takes two binary strings as input and gives one as output.

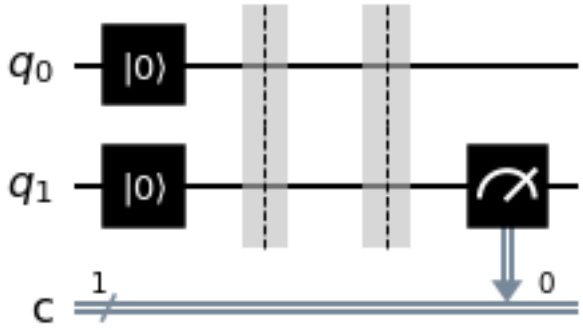
The output is '0' when the inputs are equal and '1' otherwise.

[6]:
```
def XOR(inp1,inp2):
    """An XOR gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 1.
    """
```

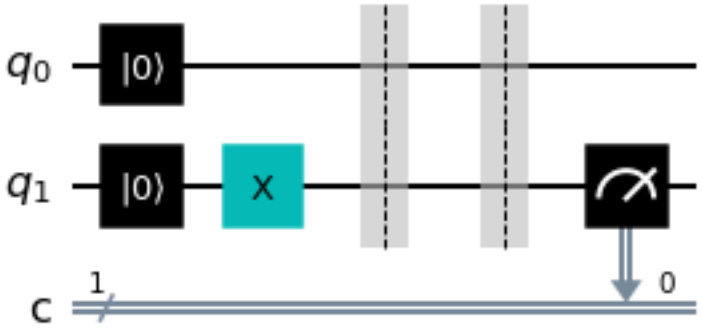
```
    qc = QuantumCircuit(2, 1)
    qc.reset(range(2))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    # barrier between input state and gate operation
    qc.barrier()

    # this is where your program for quantum XOR gate goes









    # barrier between input state and gate operation
    qc.barrier()

    qc.measure(1,0) # output from qubit 1 is measured

    #We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    #Since the output will be deterministic, we can use just a single shot to␣
 ↪get it
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

[7]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = XOR(inp1, inp2)
        print('XOR with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

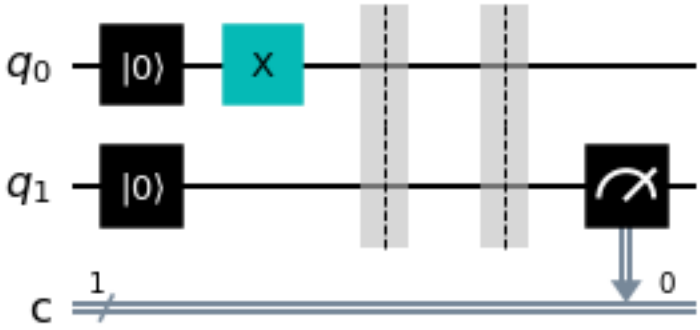
```
XOR with inputs 0 0 gives output 0
```



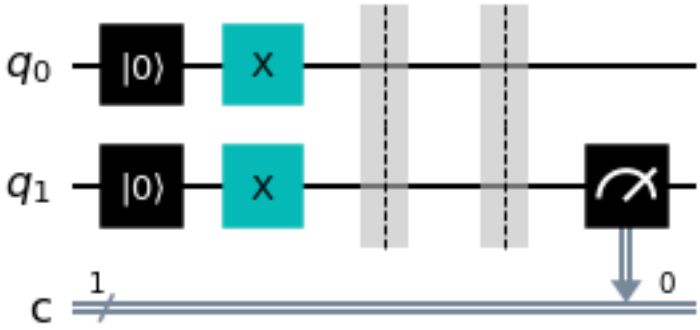

```
XOR with inputs 0 1 gives output 1
```




```
XOR with inputs 1 0 gives output 0
```




```
XOR with inputs 1 1 gives output 1
```



AND gate

Takes two binary strings as input and gives one as output.

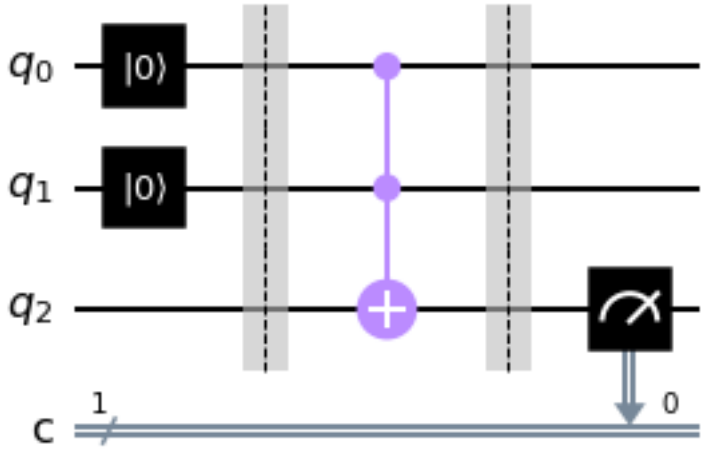
The output is `'1'` only when both the inputs are `'1'`.

[8]:
```
def AND(inp1,inp2):
    """An AND gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
```

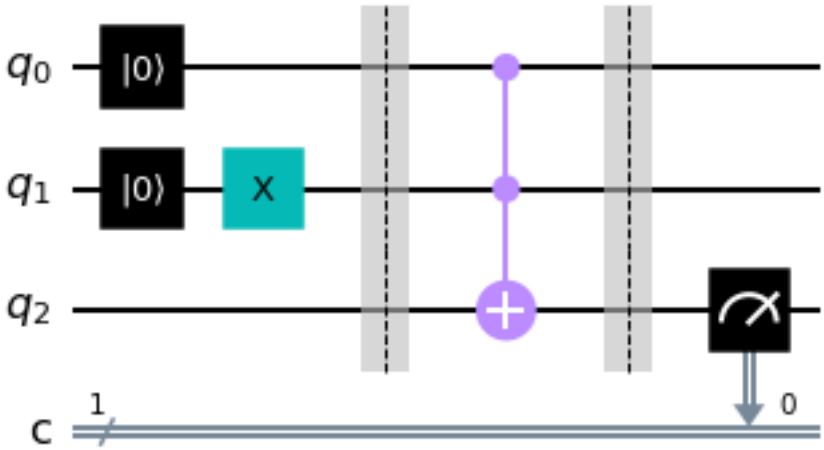
```
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 2.
    """
    qc = QuantumCircuit(3, 1)
    qc.reset(range(2))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    # this is where your program for quantum AND gate goes

    qc.ccx(0,1,2)









    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to␣
 ↪get it
    job = backend.run(qc, shots=1, memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

[9]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = AND(inp1, inp2)
        print('AND with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

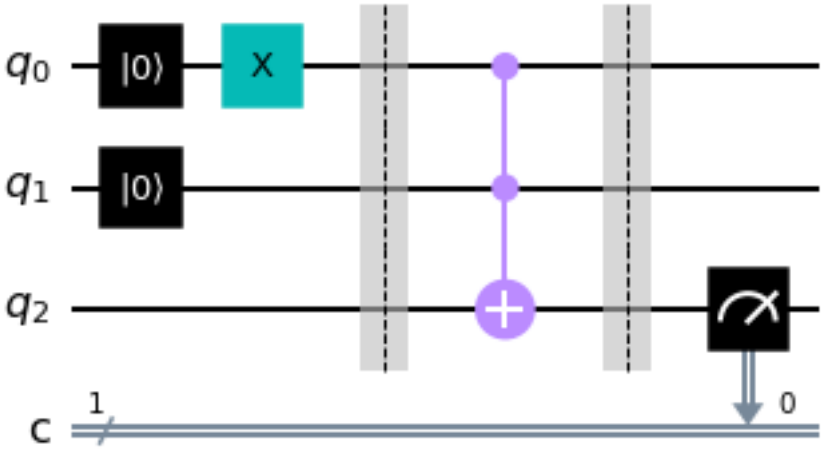
```
AND with inputs 0 0 gives output 0
```



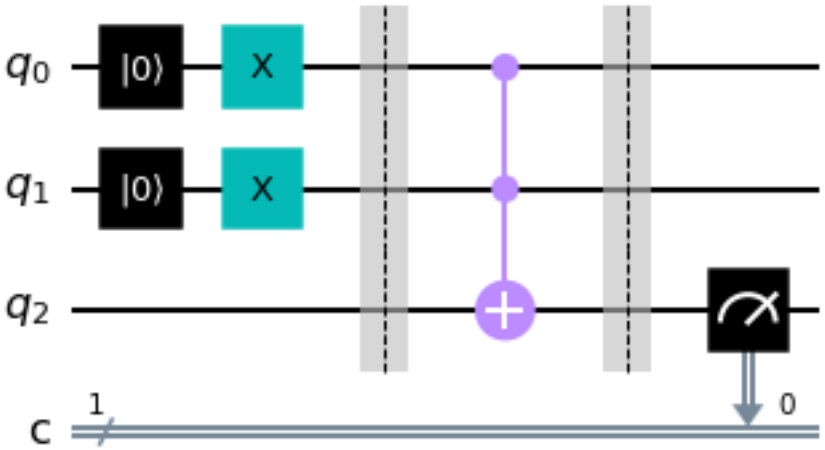

```
AND with inputs 0 1 gives output 0
```




```
AND with inputs 1 0 gives output 0
```




```
AND with inputs 1 1 gives output 1
```



NAND gate

Takes two binary strings as input and gives one as output.

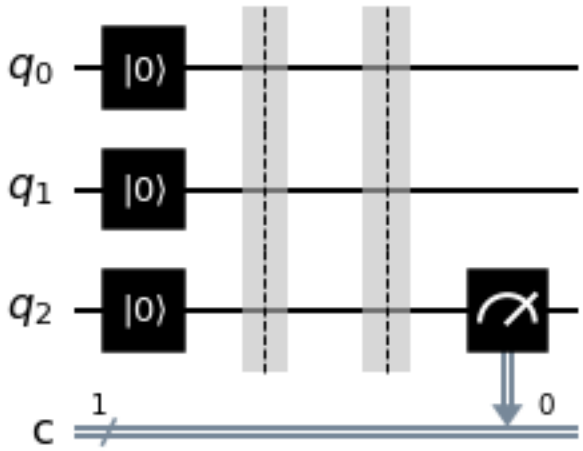
The output is '0' only when both the inputs are '1'.

[10]:
```
def NAND(inp1,inp2):
    """An NAND gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output NAND circuit.
        str: Output value measured from qubit 2.
    """
    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    # this is where your program for quantum NAND gate goes









    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to␣
 ↪get it
    job = backend.run(qc,shots=1,memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

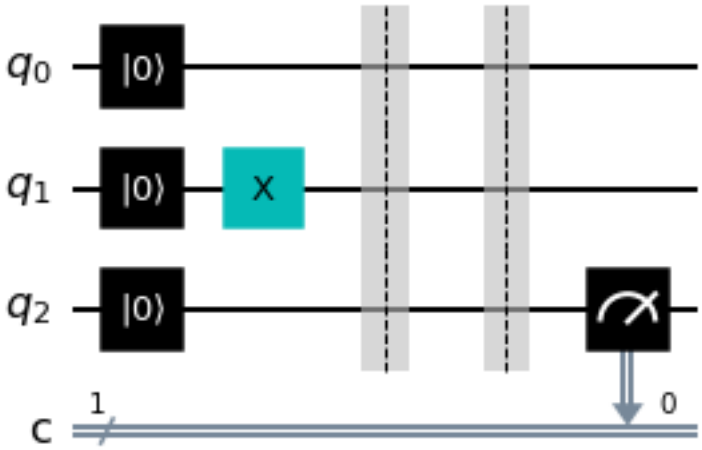
[11]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = NAND(inp1, inp2)
```

```
        print('NAND with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

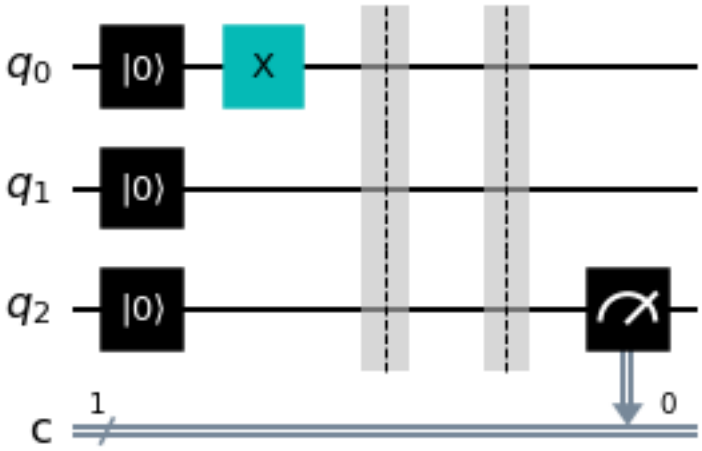
```
NAND with inputs 0 0 gives output 0
```



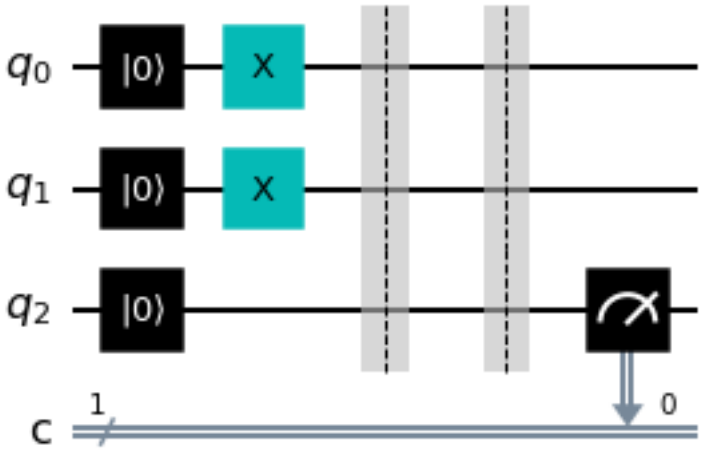

```
NAND with inputs 0 1 gives output 0
```



```
NAND with inputs 1 0 gives output 0
```




```
NAND with inputs 1 1 gives output 0
```



OR gate

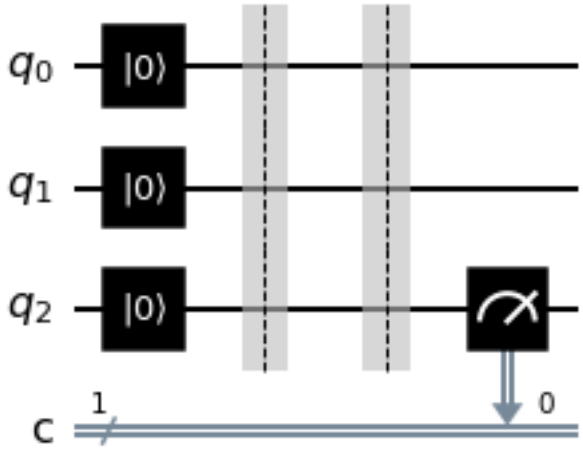
Takes two binary strings as input and gives one as output.

The output is '1' if either input is '1'.

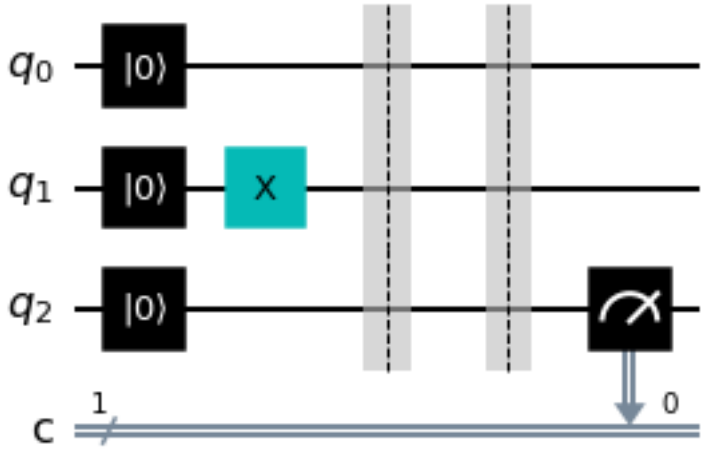
[12]:
```
def OR(inp1,inp2):
    """An OR gate.

    Parameters:
        inpt1 (str): Input 1, encoded in qubit 0.
        inpt2 (str): Input 2, encoded in qubit 1.

    Returns:
        QuantumCircuit: Output XOR circuit.
        str: Output value measured from qubit 2.
    """

    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()

    # this is where your program for quantum OR gate goes









    qc.barrier()
    qc.measure(2, 0) # output from qubit 2 is measured

    # We'll run the program on a simulator
    backend = Aer.get_backend('aer_simulator')
    # Since the output will be deterministic, we can use just a single shot to␣
↪get it
    job = backend.run(qc,shots=1,memory=True)
    output = job.result().get_memory()[0]

    return qc, output
```

[13]:
```
## Test the function
for inp1 in ['0', '1']:
    for inp2 in ['0', '1']:
        qc, output = OR(inp1, inp2)
        print('OR with inputs',inp1,inp2,'gives output',output)
        display(qc.draw())
        print('\n')
```

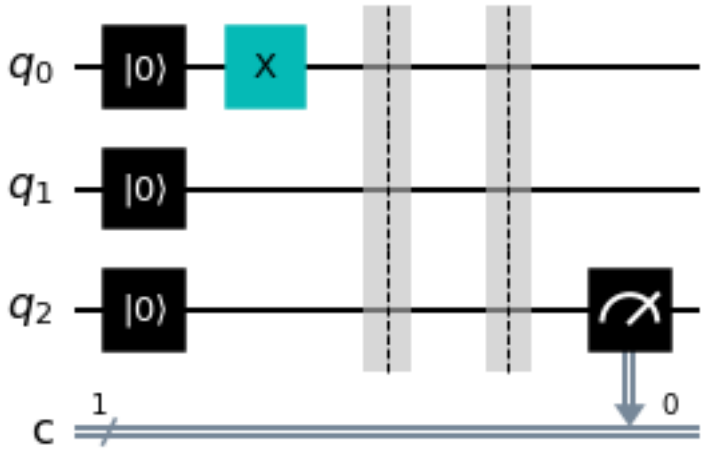
```
OR with inputs 0 0 gives output 0
```



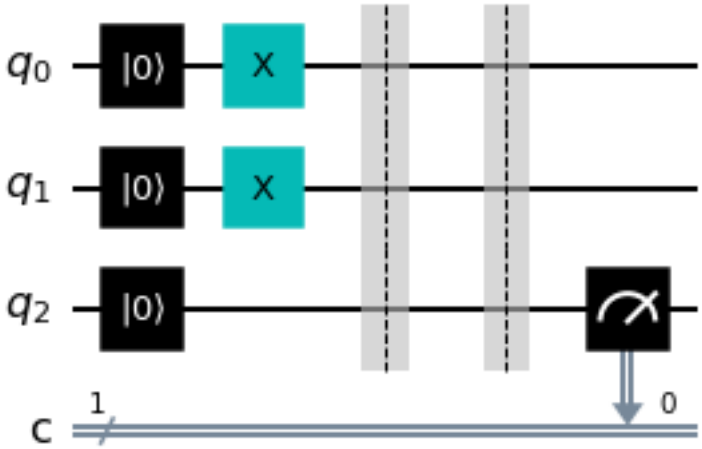

```
OR with inputs 0 1 gives output 0
```




```
OR with inputs 1 0 gives output 0
```




```
OR with inputs 1 1 gives output 0
```



Part 2: AND gate on Quantum Computer

Goal

```
<p style=" padding: 0px 0px 10px 10px;
          font-size:16px;">Execute AND gate on a real quantum system and learn how the noise p
```

In Part 1 you made an `AND` gate from quantum gates, and executed it on the simulator. Here in Part 2 you will do it again, but instead run the circuits on a real quantum computer. When using a real quantum system, one thing you should keep in mind is that present day quantum computers are not fault tolerant; they are noisy.

The 'noise' in a quantum system is the collective effects of all the things that should not happen, but nevertheless do. Noise results in outputs are not always what we would expect. There is noise associated with all processes in a quantum circuit: preparing the initial state, applying gates, and qubit measurement. For the gates, noise levels can vary between different gates and between different qubits. `cx` gates are typically more noisy than any single qubit gate.

Here we will use the quantum systems from the IBM Quantum Experience. If you do not have access, you can do so here.

Now that you are ready to use the real quantum computer, let's begin.

Step 1. Choosing a device

First load the account from the credentials saved on disk by running the following cell:

[15]:
```
from qiskit import IBMQ
IBMQ.
 ↪save_account('079e1cfd631de2886a50ebc04ee5d9823d9d1574b6a56392a322ea6a36a640dabf994a2661878
```

```
IBMQ.load_account()
```

```
configrc.store_credentials:WARNING:2022-04-20 17:51:24,147: Credentials already
present. Set overwrite=True to overwrite.
ibmqfactory.load_account:WARNING:2022-04-20 17:51:24,365: Credentials are
already in use. The existing account in the session will be replaced.
```

[15]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

After your account is loaded, you can see the list of providers that you have access to by running the cell below. Each provider offers different systems for use. For open users, there is typically only one provider `ibm-q/open/main`:

[16]:
```
IBMQ.providers()
```

[16]:
```
[<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>]
```

Let us grab the provider using `get_provider`. The command, `provider.backends( )` shows you the list of backends that are available to you from the selected provider.

[17]:
```
provider = IBMQ.get_provider('ibm-q')
provider.backends()
```

[17]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_bogota') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

Among these options, you may pick one of the systems to run your circuits on. All except the `ibmq_qasm_simulator` all are real quantum computers that you can use. The differences among these systems resides in the number of qubits, their connectivity, and the system error rates.

Upon executing the following cell you will be presented with a widget that displays all of the information about your choice of the backend. You can obtain information that you need by clicking on the tabs. For example, backend status, number of qubits and the connectivity are under `configuration` tab, where as the `Error Map` tab will reveal the latest noise information for the system.

[18]:
```
import qiskit.tools.jupyter

backend_ex = provider.get_backend('ibmq_lima')
backend_ex
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pa
```

[18]:
```
<IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>
```

For our AND gate circuit, we need a backend with three or more qubits, which is true for all the real systems except for `ibmq_armonk`. Below is an example of how to filter backends, where we filter for number of qubits, and remove simulators:

[19]:
```
backends = provider.backends(filters = lambda x:x.configuration().n_qubits >= 2␣
 ↪and not x.configuration().simulator
                              and x.status().operational==True)
backends
```

[19]:
```
[<IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_bogota') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

One convenient way to choose a system is using the `least_busy` function to get the backend with the lowest number of jobs in queue. The downside is that the result might have relatively poor accuracy because, not surprisingly, the lowest error rate systems are the most popular.

[20]:
```
from qiskit.providers.ibmq import least_busy
backend = least_busy(provider.backends(filters=lambda x: x.configuration().
 ↪n_qubits >= 2 and
```

```
                                        not x.configuration().simulator and x.
↪status().operational==True))
backend
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pa
```

[20]:
```
<IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>
```

Real quantum computers need to be recalibrated regularly, and the fidelity of a specific qubit or gate can change over time. Therefore, which system would produce results with less error can vary.

In this exercise, we select one of the IBM Quantum systems: `ibmq_quito`.

[21]:
```
# run this cell
backend = provider.get_backend('ibmq_quito')
```

Step 2. Define AND function for a real device

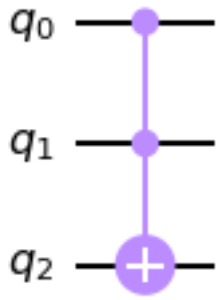
We now define the AND function. We choose 8192 as the number of shots, the maximum number of shots for open IBM systems, to reduce the variance in the final result. Related information is well explained here.

Qiskit Transpiler

It is important to know that when running a circuit on a real quantum computer, circuits typically need to be transpiled for the backend that you select so that the circuit contains only those gates that the quantum computer can actually perform. Primarily this involves the addition of swap gates so that two-qubit gates in the circuit map to those pairs of qubits on the device that can actually perform these gates. The following cell shows the AND gate represented as a Toffoli gate decomposed into single- and two-qubit gates, which are the only types of gate that can be run on IBM hardware.

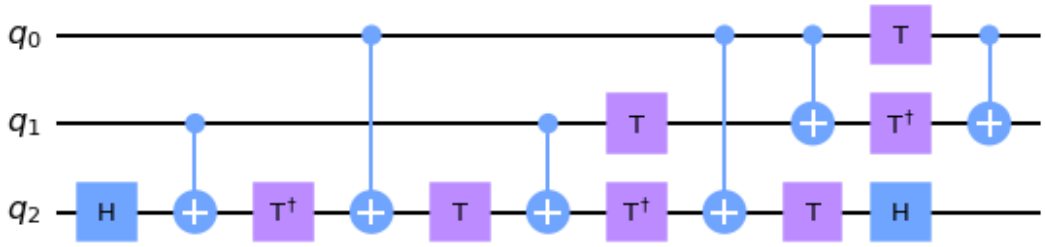
[22]:
```
qc_and = QuantumCircuit(3)
qc_and.ccx(0,1,2)
print('AND gate')
display(qc_and.draw())
print('\n\nTranspiled AND gate with all the required connectivity')
qc_and.decompose().draw()
```

```
AND gate
```



Transpiled AND gate with all the required connectivity

[22]:



In addition, there are often optimizations that the transpiler can perform that reduce the overall gate count, and thus total length of the input circuits. Note that the addition of swaps to match the device topology, and optimizations for reducing the length of a circuit are at odds with each other. In what follows we will make use of initial_layout that allows us to pick the qubits on a device used for the computation and optimization_level, an argument that allows selecting from internal defaults for circuit swap mapping and optimization methods to perform.

You can learn more about transpile function in depth here.

Let's modify AND function in Part1 properly for the real system with the transpile step included.

[23]:
```
from qiskit.tools.monitor import job_monitor
```

[24]:
```
# run the cell to define AND gate for real quantum system

def AND(inp1, inp2, backend, layout):

    qc = QuantumCircuit(3, 1)
    qc.reset(range(3))

    if inp1=='1':
        qc.x(0)
    if inp2=='1':
        qc.x(1)

    qc.barrier()
    qc.ccx(0, 1, 2)
    qc.barrier()
    qc.measure(2, 0)

    qc_trans = transpile(qc, backend, initial_layout=layout,␣
↪optimization_level=3)
    job = backend.run(qc_trans, shots=8192)
    print(job.job_id())
    job_monitor(job)

    output = job.result().get_counts()

    return qc_trans, output
```

When you submit jobs to quantum systems, job_monitor will start tracking where your submitted job is in the pipeline.

First, examine ibmq_quito through the widget by running the cell below.

[26]:
```
backend
```

VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pa

[26]: <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open', project='main')>

Determine three qubit initial layout considering the error map and assign it to the list variable layout2.

[28]:
```
layout = (3,2,1)
```

Describe the reason for your choice of initial layout.

your answer: Because they're the three nodes with less error rate when we examine the error map window. Also, we select the third one because as the first because I think that it will be better to have less error rate in the target qubit.

Execute AND gate on ibmq_quito by running the cell below.

[29]:
```
output_all = []
qc_trans_all = []
prob_all = []

worst = 1
best = 0
for input1 in ['0','1']:
    for input2 in ['0','1']:
        qc_trans, output = AND(input1, input2, backend, layout)

        output_all.append(output)
        qc_trans_all.append(qc_trans)

        prob = output[str(int( input1=='1' and input2=='1' ))]/8192
        prob_all.append(prob)

        print('\nProbability of correct answer for inputs',input1,input2)
        print('{:.2f}'.format(prob) )
        print('--------------------------------')

        worst = min(worst,prob)
        best = max(best, prob)

print('')
print('\nThe highest of these probabilities was {:.2f}'.format(best))
print('The lowest of these probabilities was {:.2f}'.format(worst))
```

```
6260495172e2114d34b79ca1
Job Status: job has successfully run

Probability of correct answer for inputs 0 0
0.93
--------------------------------
626049793ba1990133565aad
Job Status: job has successfully run

Probability of correct answer for inputs 0 1
0.93
--------------------------------
626049dea28f430c16a009dc
Job Status: job has successfully run

Probability of correct answer for inputs 1 0
0.87
--------------------------------
626049f2a28f435e2ca009dd
```

```
Job Status: job has successfully run

Probability of correct answer for inputs 1 1
0.84
---------------------------------


The highest of these probabilities was 0.93
The lowest of these probabilities was 0.84
```

Step 3. Interpret the result

There are several quantities that distinguish the circuits. Chief among them is the **circuit depth**. Circuit depth is defined in detail here (See the Supplementary Information and click the Quantum Circuit Properties tab). Circuit depth is proportional to the number of gates in a circuit, and loosely corresponds to the runtime of the circuit on hardware. Therefore, circuit depth is an easy to compute metric that can be used to estimate the fidelity of an executed circuit.

A second important value is the number of **nonlocal** (multi-qubit) **gates** in a circuit. On IBM Quantum systems, the only nonlocal gate that can physically be performed is the CNOT gate. Recall that CNOT gates are the most expensive gates to perform, and thus the total number of these gates also serves as a good benchmark for the accuracy of the final output.

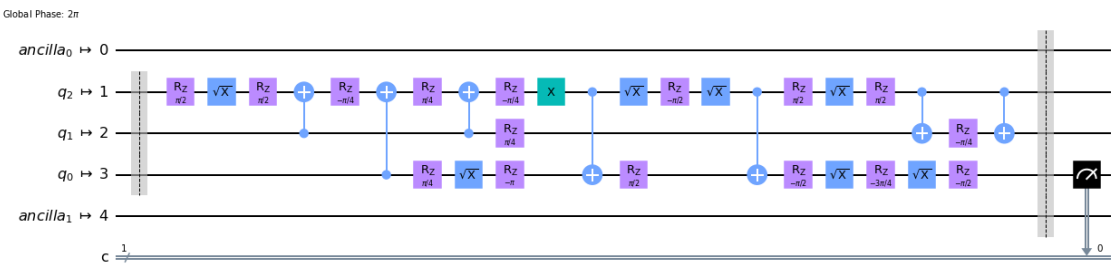
Circuit depth and result accuracy

Running the cells below will display the four transpiled AND gate circuit diagrams with the corresponding inputs that executed on ibm_lagos and their circuit depths with the success probability for producing correct answer.

[30]:
```
print('Transpiled AND gate circuit for ibmq_vigo with input 0 0')
print('\nThe circuit depth : {}'.format (qc_trans_all[0].depth()))
print('# of nonlocal gates : {}'.format (qc_trans_all[0].num_nonlocal_gates()))
print('Probability of correct answer : {:.2f}'.format(prob_all[0]) )
qc_trans_all[0].draw('mpl')
```

```
Transpiled AND gate circuit for ibmq_vigo with input 0 0

The circuit depth : 22
# of nonlocal gates : 7
Probability of correct answer : 0.93
```

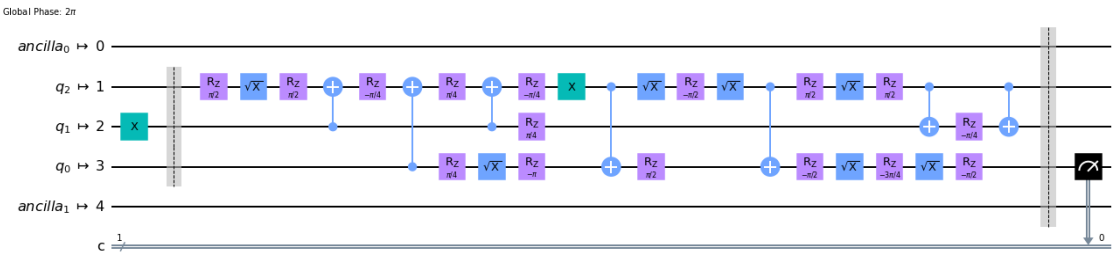
[30]:



[31]:
```
print('Transpiled AND gate circuit for ibmq_vigo with input 0 1')
print('\nThe circuit depth : {}'.format (qc_trans_all[1].depth()))
print('# of nonlocal gates : {}'.format (qc_trans_all[1].num_nonlocal_gates()))
print('Probability of correct answer : {:.2f}'.format(prob_all[1]) )
qc_trans_all[1].draw('mpl')
```

```
Transpiled AND gate circuit for ibmq_vigo with input 0 1

The circuit depth : 23
# of nonlocal gates : 7
Probability of correct answer : 0.93
```

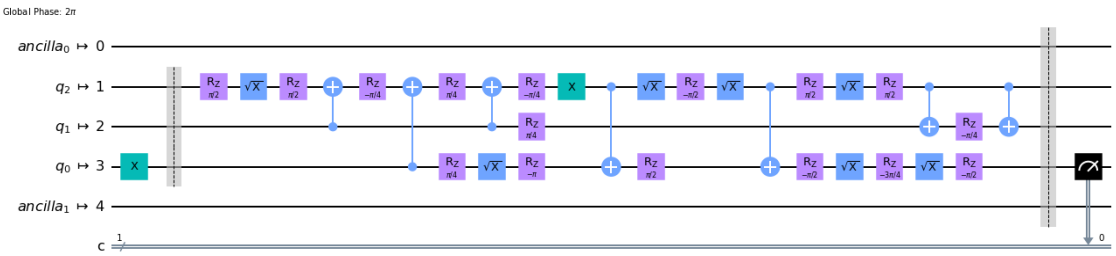
[31]:



[32]:
```
print('Transpiled AND gate circuit for ibmq_vigo with input 1 0')
print('\nThe circuit depth : {}'.format (qc_trans_all[2].depth()))
print('# of nonlocal gates : {}'.format (qc_trans_all[2].num_nonlocal_gates()))
print('Probability of correct answer : {:.2f}'.format(prob_all[2]) )
qc_trans_all[2].draw('mpl')
```

```
Transpiled AND gate circuit for ibmq_vigo with input 1 0

The circuit depth : 23
# of nonlocal gates : 7
Probability of correct answer : 0.87
```

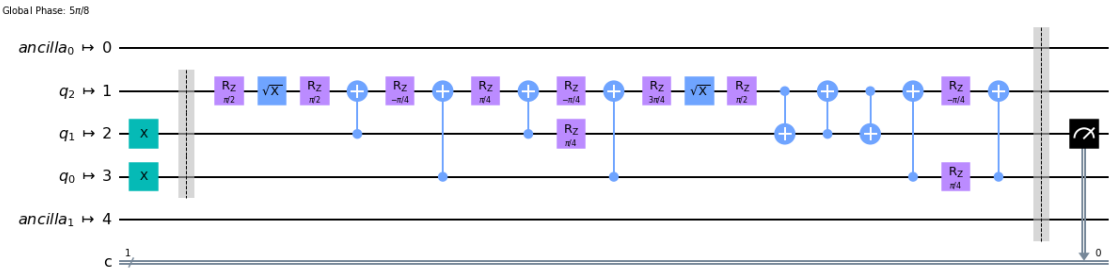
[32]:



[33]:
```
print('Transpiled AND gate circuit for ibmq_vigo with input 1 1')
print('\nThe circuit depth : {}'.format (qc_trans_all[3].depth()))
print('# of nonlocal gates : {}'.format (qc_trans_all[3].num_nonlocal_gates()))
print('Probability of correct answer : {:.2f}'.format(prob_all[3]) )
qc_trans_all[3].draw('mpl')
```

```
Transpiled AND gate circuit for ibmq_vigo with input 1 1

The circuit depth : 21
# of nonlocal gates : 9
Probability of correct answer : 0.84
```

[33]:



Explain reason for the dissimilarity of the circuits. Describe the relations between the property of the circuit and the accuracy of the outcomes.

your answer: Normalmente, aquellos circuitos que tengan menor profundidad tendrán peores resultados en la probabilidad de respuesta correcta. Esto sucede porque vamos acumulando el error, es decir, cuanto más profundo, más probabilidad de error tenemos. También nos ocurre (en el caso de entrada 1-1) que, tenemos la profundidad más pequeña pero el error más grande, esto sucede porque es el circuito que mayor número de puertas CNOT utiliza. El uso de puertas CNOT también agrava el error. Por tanto, podemos obsevar que los mejores resultados los obtenemos con entradas 0-0 y 0-1, que aunque no tengan la menor profundidad, si que son las que tienen menor número de puertas CNOT (7 en ambos casos). El caso de 1-0 tenemos profundidad 23 y un uso de también 7 puertas, por tanto obtenemos un peor resultado que en los dos anteriores (debido a la profundidad mayor). En el caso de 1-1 tenemos profundidad 21 y 9 puertas y es en el que peor resultado obtenemos. Como vemos en el apartado anterior en la parte de error map, vemos que el error de las puertas CNOT es alto, por tanto, podemos decir que este agrava mucho más la probabilidad de respuesta correcta que la profundidad que tenga el circuito.

[ ]:

[ ]: