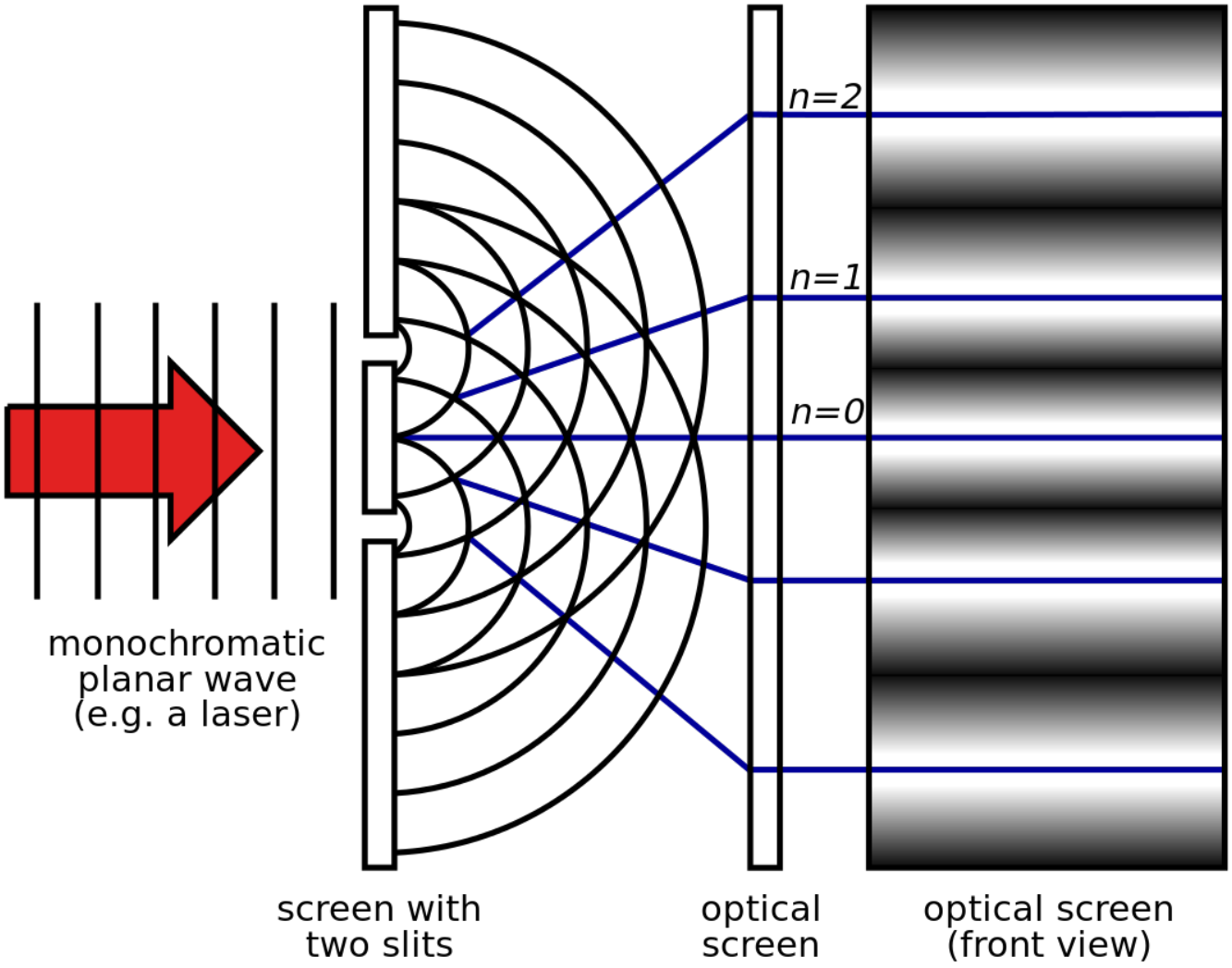
Open Source Quantum Computing

Sean Dague
Software Engineer - IBM Q
sean.dague@ibm.com
@sdague

Quantum Mechanics - Double Slit Experiment



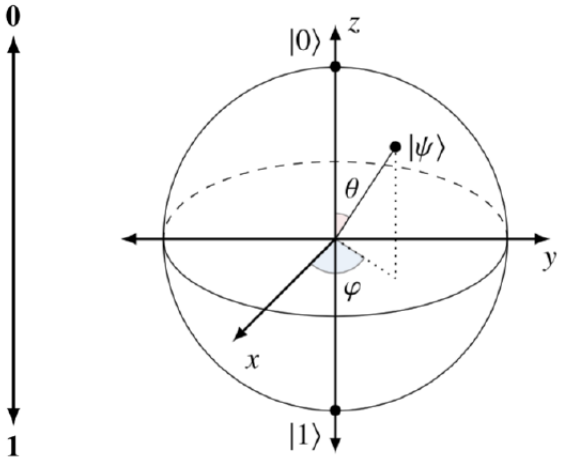
Wikimedia commons Two-Slit_Experiment_Light (https://upload.wikimedia.org/wikipedia/commons/thumb/8/8b/Two-Slit_Experiment_Light.svg/2000px-Two-Slit_Experiment_Light.svg.png)

Bits and qubits

A classical bit can be **0** or **1**.

A quantum bit, or *qubit*, can take on those values but can represent a combination of **0** and **1** while we are computing.

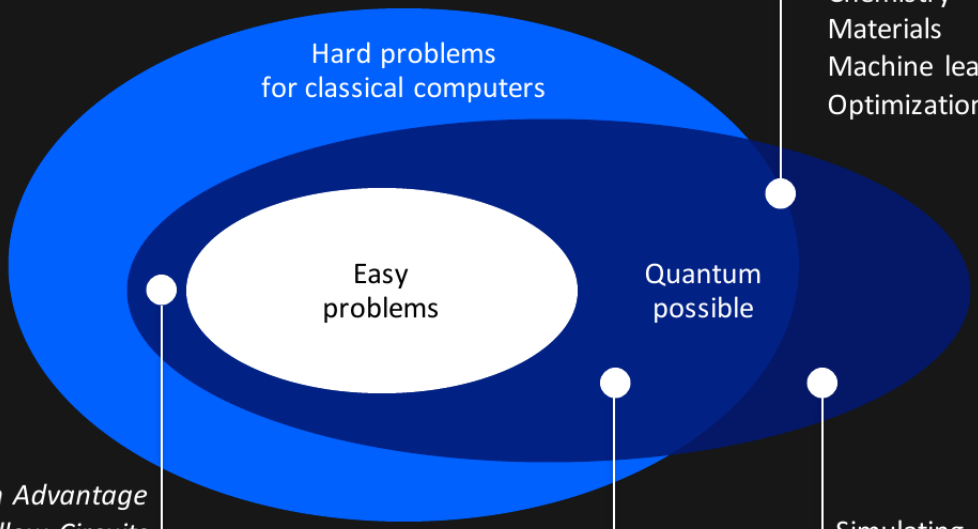
When we measure a qubit, it becomes **0** or **1** based on probability.



classical bit quantum bit analogy

Hard versus easy problems

Quantum computing may provide a new path to solve some of the hardest or most memory intensive problems in business and science.



Applications

Chemistry
Materials
Machine learning
Optimization

Quantum Advantage with Shallow Circuits
IBM 2018

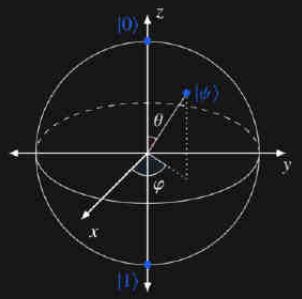
Factoring

Simulating quantum mechanics

 #IBMQ

Quantum computing uses essential ideas from quantum mechanics

Superposition



Superposition is creating a quantum state that is a combination of |0> and |1>

$$a\,|0\rangle + b\,|1\rangle$$

where

a and b are complex numbers

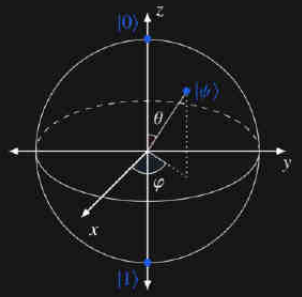
$|a|^2 + |b|^2 = 1$

Note that if a and b are non-zero, then the qubit's state contains both |0> and |1>.

This is what people mean when they say that a qubit can be "0 and 1 at the same time."

Quantum computing uses essential ideas from quantum mechanics

Measurement



Measurement is forcing the qubit's state

$$a\ |0\rangle + b\ |1\rangle$$

to |0> or |1> by observing it, where

$|a|^2$ is the probability we will get |0> when we measure

$|b|^2$ is the probability we will get |1> when we measure

Examples

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

has an equal probability of becoming |0> or |1>.

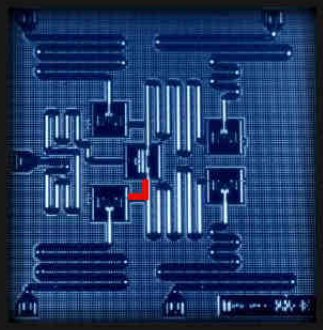
$$\frac{\sqrt{3}}{2}\ |0\rangle - \frac{1}{2}\ i\ |1\rangle$$

has a 75% chance of becoming |0>.

 #IBMQ

Quantum computing uses essential ideas from quantum mechanics

Entanglement



With two qubits we get combinations like

$$a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle$$

where

$|01\rangle$ means the first qubit is $|0\rangle$ and the second is $|1\rangle$

a, b, c, and d are complex numbers

$|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$

If two or more of the a, b, c, and d are non-zero, and we cannot separate the qubits, they are entangled with perfect correlation and are no longer independent.

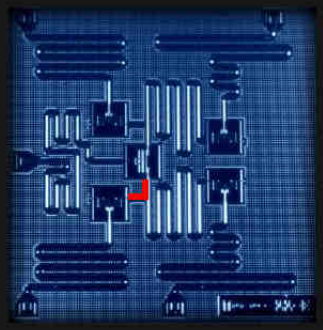
Examples

$\frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|01\rangle$ *not* entangled

$\frac{\sqrt{2}}{2}|01\rangle - \frac{\sqrt{2}}{2}|10\rangle$, $\frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$ entangled

Quantum computing uses essential ideas from quantum mechanics

Entanglement



We can write

$$\frac{\sqrt{2}}{2}|00> + \frac{\sqrt{2}}{2}|01>$$

as

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

but we cannot write

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

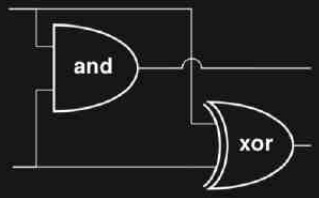
as the combination of two single qubit states.

They are entangled!

Once you measure the first qubit, the second is uniquely determined.

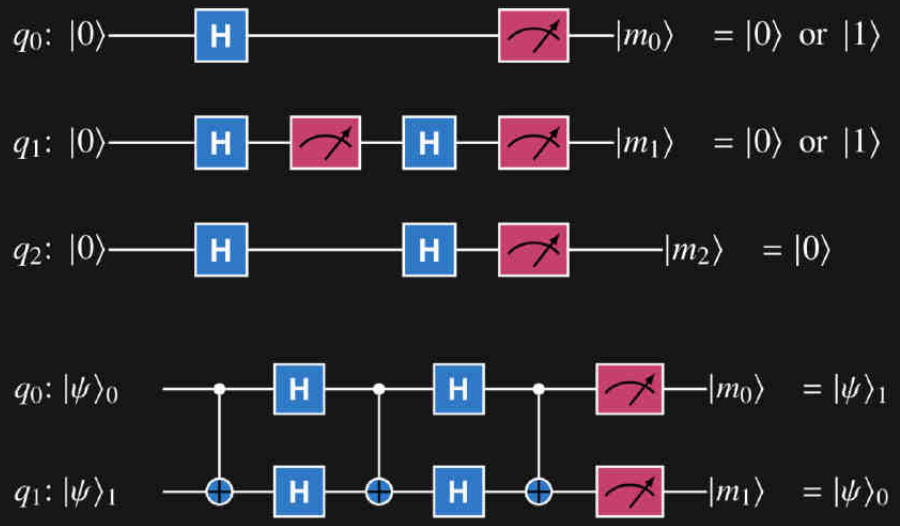
Quantum computing uses essential ideas from quantum mechanics

Gates / Operations



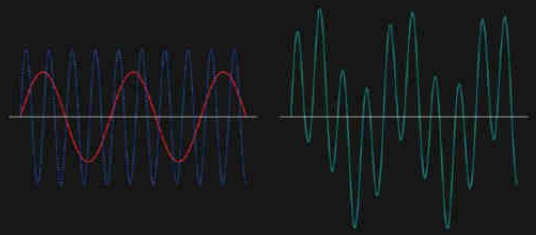
Classical logical circuits use operations like and, or, not, nand, and xor. We also call these gates.

Quantum circuits use reversible gates that change the quantum states of one, two , or more qubits.

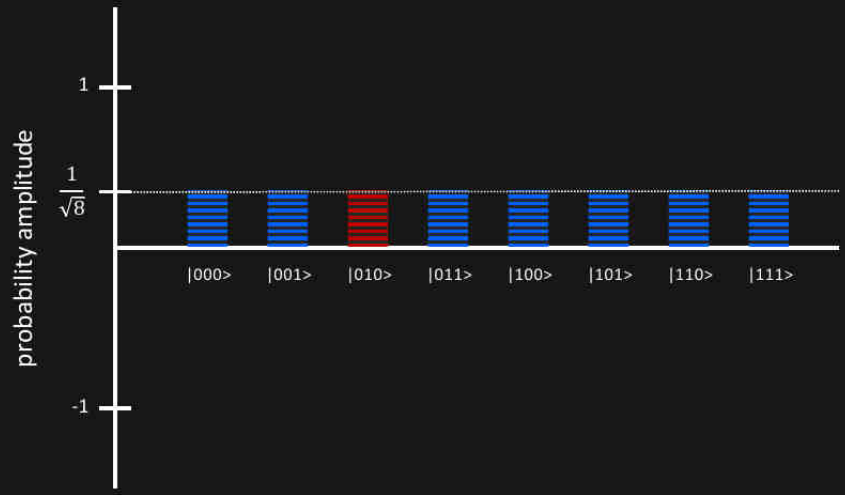


Quantum computing uses essential ideas from quantum mechanics

Interference



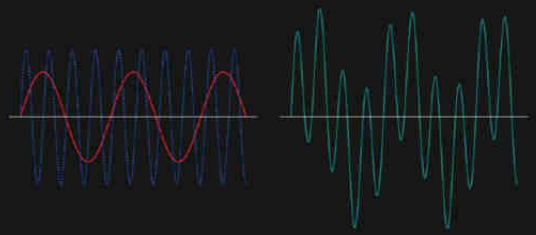
Interference allows us to increase the probability of getting the right answer and decrease the chance of getting the wrong one.



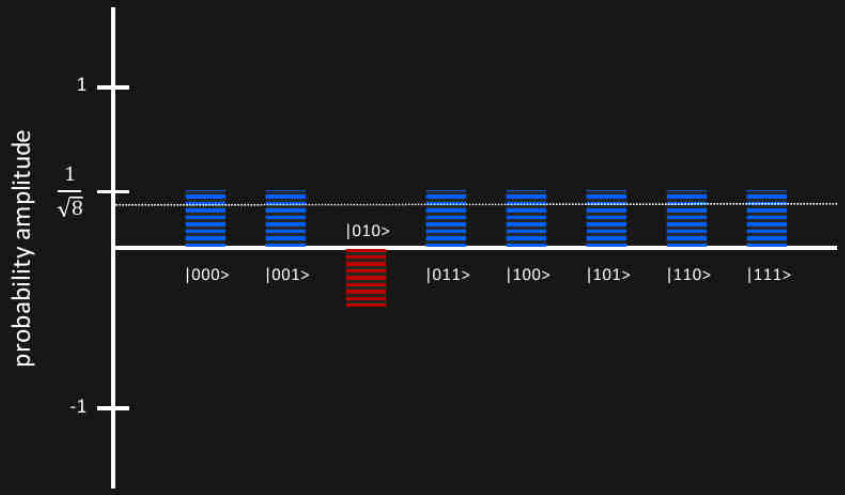
 #IBMQ

Quantum computing uses essential ideas from quantum mechanics

Interference



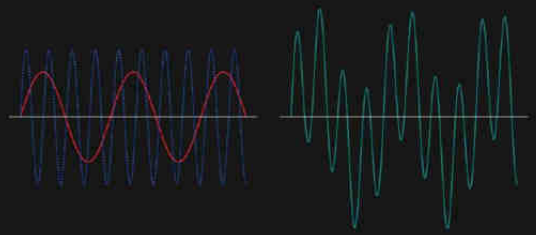
Interference allows us to increase the probability of getting the right answer and decrease the chance of getting the wrong one.



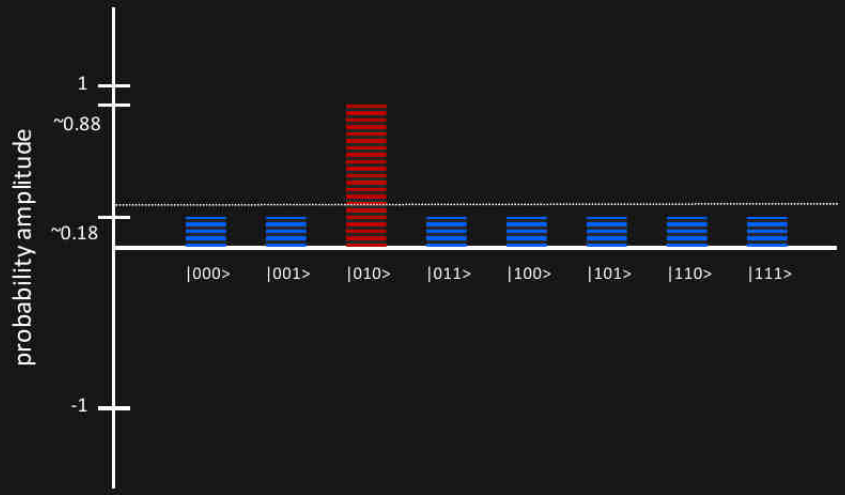
 #IBMQ

Quantum computing uses essential ideas from quantum mechanics

Interference



Interference allows us to increase the probability of getting the right answer and decrease the chance of getting the wrong one.



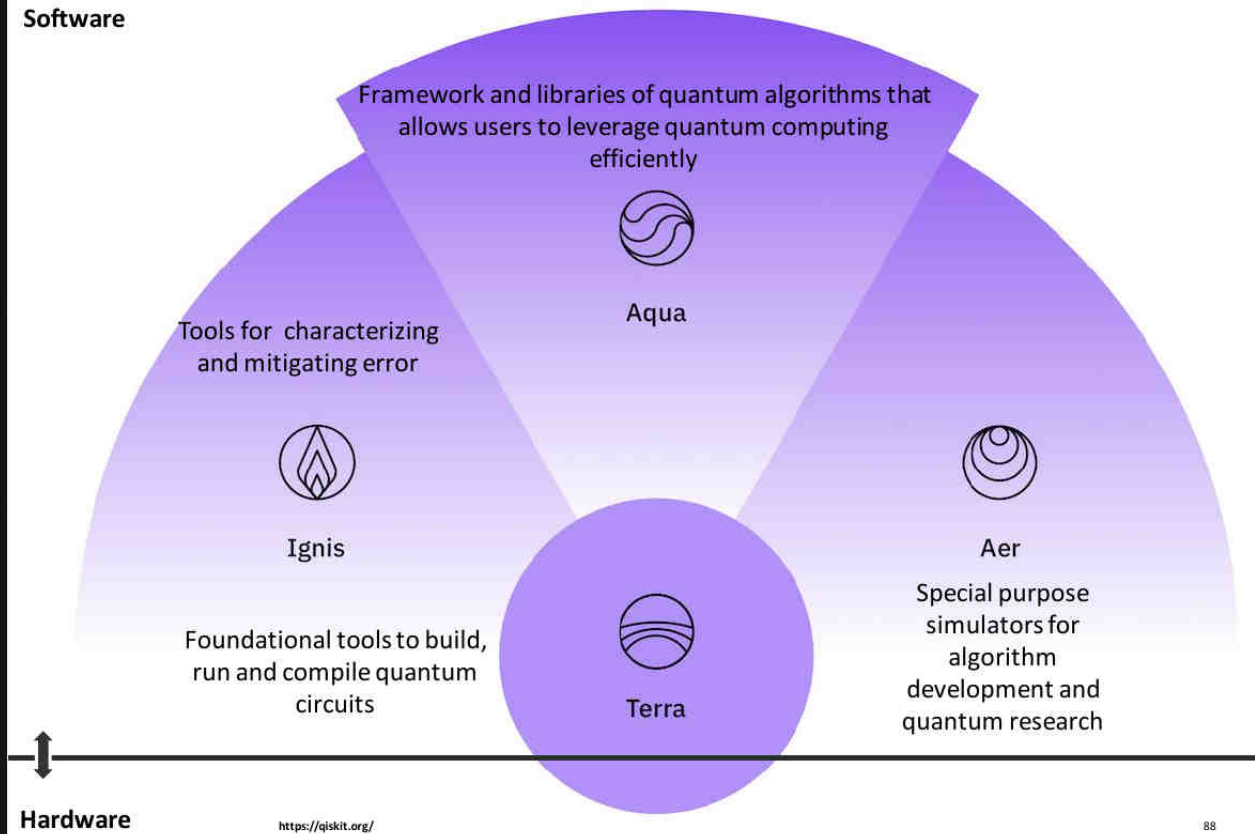
 #IBMQ

Let's do some Quantum Computing!

Qiskit Software Stack

Brings quantum computing out of the laboratory and into the hands of developers

- Open Source (Apache 2.0)
- Written in Python 3
- Modular and extendible



 #IBMQ

https://qiskit.org/

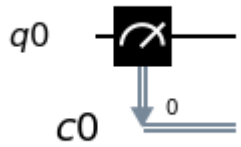
Let's run some Qiskit

In [1]:
```
import qiskit
from qiskit.tools.jupyter import *
%matplotlib inline
# Allocate 2 simulators, we'll use these later
SIM = qiskit.BasicAer.get_backend('qasm_simulator')
SV_SIM = qiskit.BasicAer.get_backend('statevector_simulator')
qiskit.__version__
```

Out[1]:
```
'0.9.0'
```

In [2]:
```
# Define the Quantum and Classical Registers
q = qiskit.QuantumRegister(1)
c = qiskit.ClassicalRegister(1)

# Build the circuit
single_q_measurement = qiskit.QuantumCircuit(q, c)
single_q_measurement.measure(q, c);

# Draw the circuit
single_q_measurement.draw(output='mpl')
```

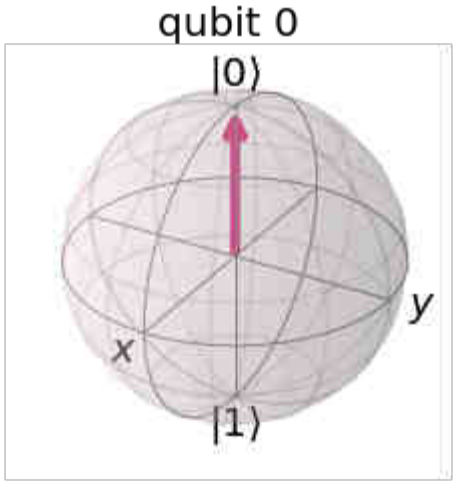
Out[2]:



Single Qubit Results

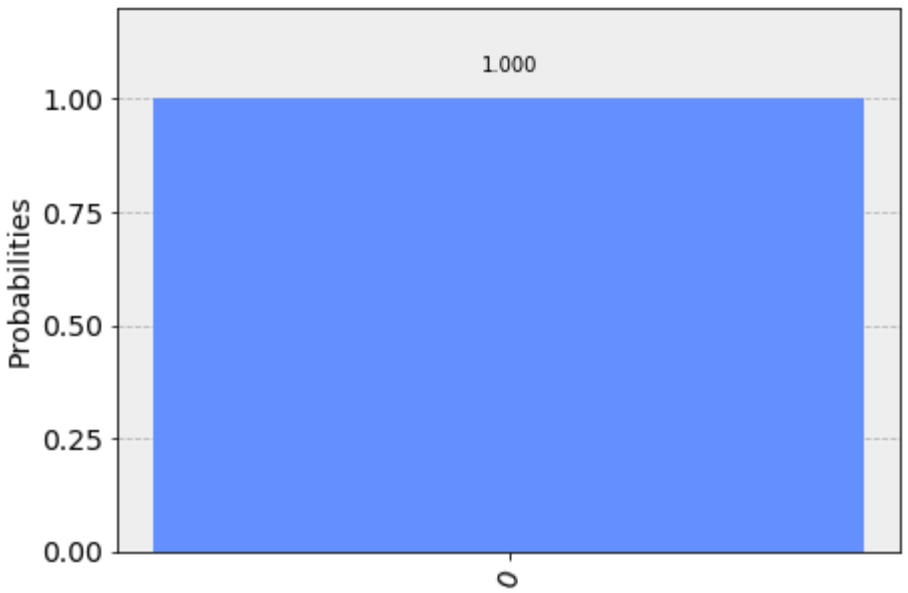
In [3]:
```
result = qiskit.execute(single_q_measurement, SV_SIM).result()
qiskit.tools.visualization.plot_bloch_multivector(result.get_statevector())
```

Out[3]:



In [4]:
```
qiskit.tools.visualization.plot_histogram(result.get_counts(single_q_measuremen
t))
```

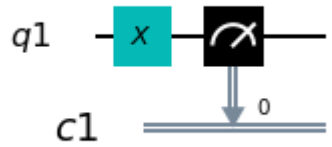
Out[4]:



Quantum Not Gate

In [5]:
```
# Create a Quantum Register with 1 qubit (wire).
qr = qiskit.QuantumRegister(1)

# Create a Classical Register with 1 bit (double wire).
cr = qiskit.ClassicalRegister(1)

# Create a Quantum Circuit from the quantum and classical registers
circ = qiskit.QuantumCircuit(qr, cr)

# Place an X gate on the qubit wire. The registers are zero-indexed.
circ.x(qr[0])

# Measure the qubit into the classical register
circ.measure(qr, cr)

# Draw the circuit
circ.draw(output='mpl')
```

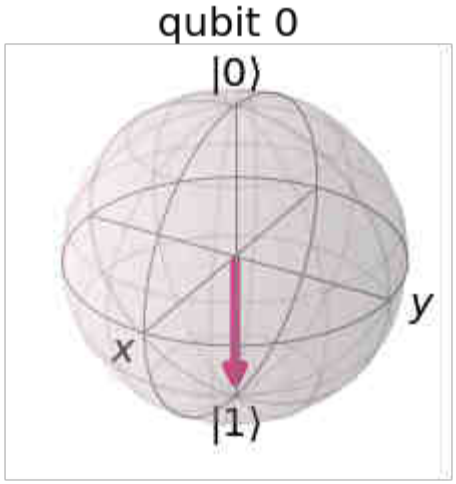
Out[5]:



Quantum Not Gate - Results

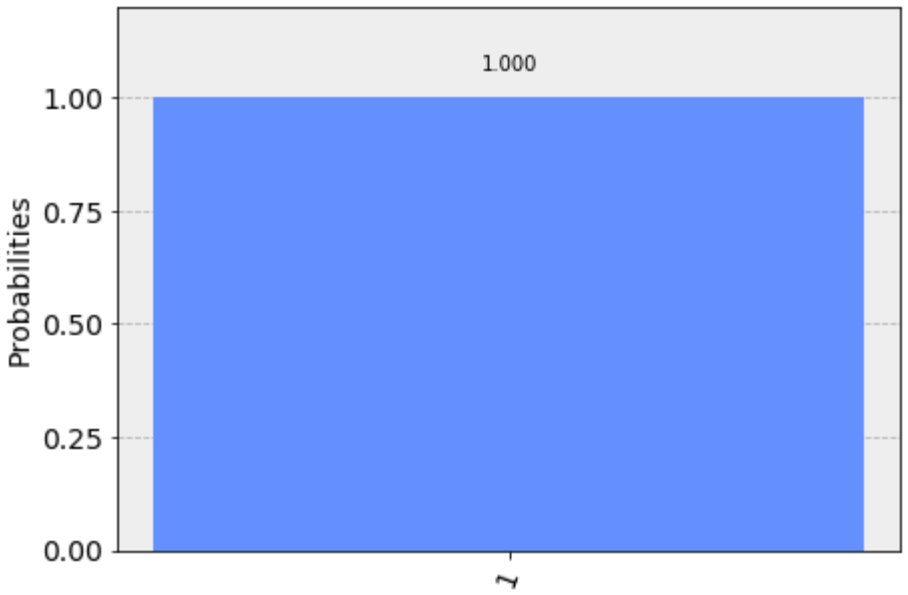
In [6]:
```
result = qiskit.execute(circ, SV_SIM).result()
qiskit.tools.visualization.plot_bloch_multivector(result.get_statevector())
```

Out[6]:



In [7]:
```
qiskit.tools.visualization.plot_histogram(result.get_counts(circ))
```

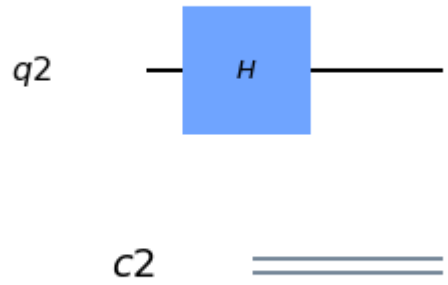
Out[7]:



Super Position

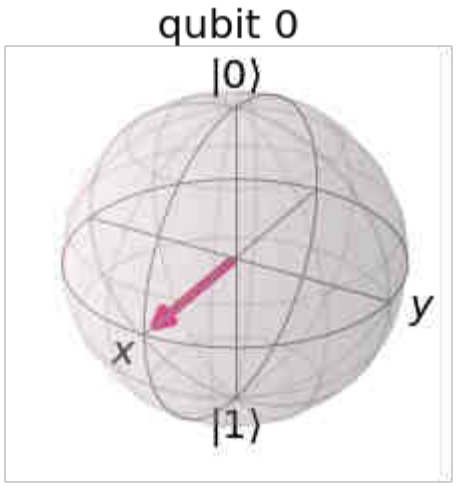
In [8]:
```
# Create a Quantum Register with 1 qubit (wire).
qr = qiskit.QuantumRegister(1)

# Create a Classical Register with 1 bit (double wire).
cr = qiskit.ClassicalRegister(1)

# Create a Quantum Circuit from the quantum and classical registers
circ = qiskit.QuantumCircuit(qr, cr)

# Place an Hadamard gate on the qubit wire
circ.h(qr[0])

# Measure the qubit into the classical register
# circ.measure(qr, cr)

# Draw the circuit
circ.draw(output='mpl', scale=1.5)
```

Out[8]:



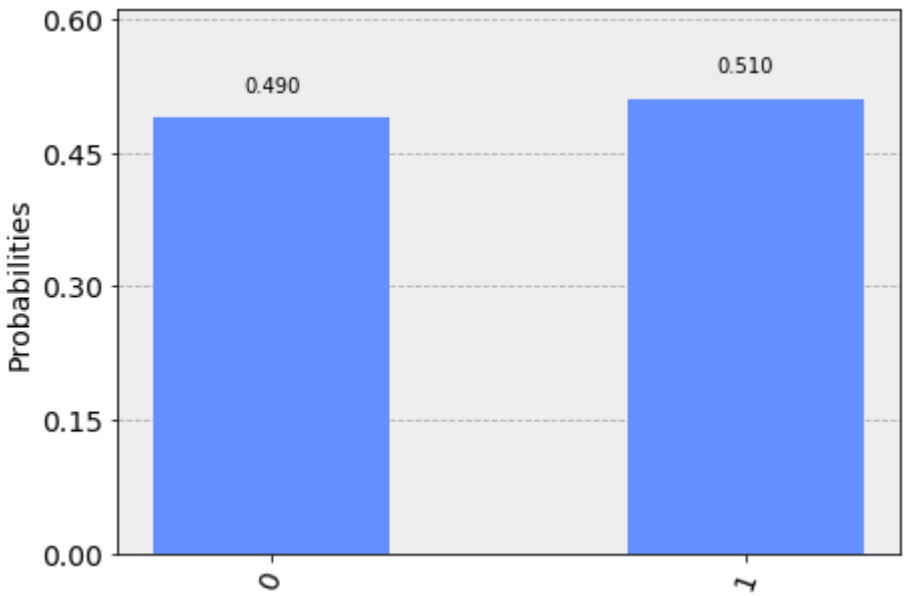
In [9]:
```
result = qiskit.execute(circ, SV_SIM).result()
qiskit.tools.visualization.plot_bloch_multivector(result.get_statevector())
```

Out[9]:



In [10]:
```
# Measure the qubit into the classical register
circ.measure(qr, cr)
# Rerun the simulation on the QASM simulator, 100 times
result = qiskit.execute(circ, SIM, shots=100).result()
qiskit.tools.visualization.plot_histogram(result.get_counts(circ))
```

Out[10]:



H gates are reversable

In [ ]:
```
# Build a circuit with 1 qubit and 1 classical bit
qr = qiskit.QuantumRegister(1)
cr = qiskit.ClassicalRegister(1)
qc = qiskit.QuantumCircuit(qr, cr)
# Apply 2 Hadamard gates on the circuit and then measure
qc.h(qr[0])
qc.h(qr[0])
qc.measure(qr, cr)
# Draw the circuit
qc.draw(output='mpl')
```

In [ ]:
```
result = qiskit.execute(qc, SIM, shots=100).result()
qiskit.tools.visualization.plot_histogram(result.get_counts(qc))
```

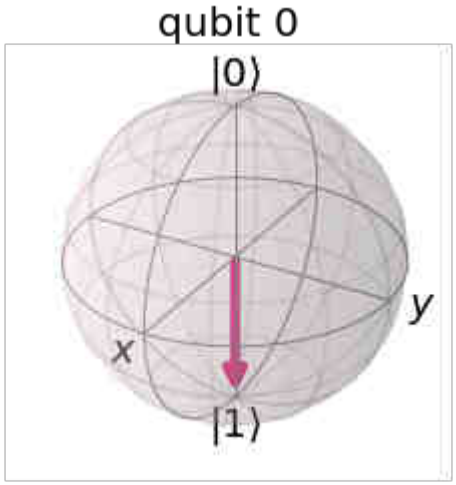
Other Gates - Y

The above shows the end state after applying the Pauli Y gate (which like the Pauli X/Quantum Not gate) can be thought of a 180 degree rotation around the Y axis. So when starting from the $|0\rangle$ basis state the result is the $|1\rangle$ basis state.

However, many gates rotate the phase. To visualize any change in phase you'll want to put the qubit in superposition with a hadamard gate first. For example:

In [11]:
```
qr = qiskit.QuantumRegister(1)
cr = qiskit.ClassicalRegister(1)
circ = qiskit.QuantumCircuit(qr, cr)
circ.y(qr)
result = qiskit.execute(circ, SV_SIM).result()
qiskit.tools.visualization.plot_bloch_multivector(result.get_statevector())
```

Out[11]:

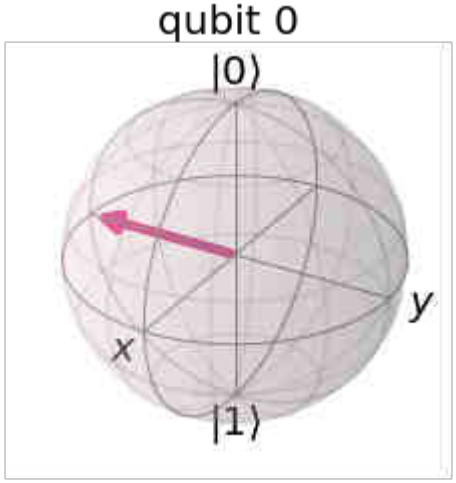


Other Gates - $S^\dagger$

This example shows the output of the $S^\dagger$ (or, conjugate of $\sqrt{Z}$ phase) gate on the bloch sphere for a qubit in superposition from a Hadamard gate. Since it is a phase gate it has no noticeable effect on a qubit in the $|0\rangle$ basis state:

In [12]:
```
qr = qiskit.QuantumRegister(1)
cr = qiskit.ClassicalRegister(1)
circ = qiskit.QuantumCircuit(qr, cr)
circ.h(qr)
circ.sdg(qr)
result = qiskit.execute(circ, SV_SIM).result()

qiskit.tools.visualization.plot_bloch_multivector(result.get_statevector())
```

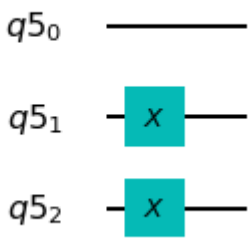
Out[12]:



Multi Qubit Circuits and Gates

In [13]:
```
# Create a Quantum Register with 3 qubits
qr = qiskit.QuantumRegister(3)

# Create a Quantum Circuit from the quantum register. Because we're going to us
e
# the statevector_simulator, we won't measure the circuit or need classical reg
isters.
circ = qiskit.QuantumCircuit(qr)

# Place an X gate on the 2nd and 3rd wires. The topmost wire is index 0.
circ.x(qr[1])
circ.x(qr[2])

# Draw the circuit
circ.draw(output='mpl')
```

Out[13]:

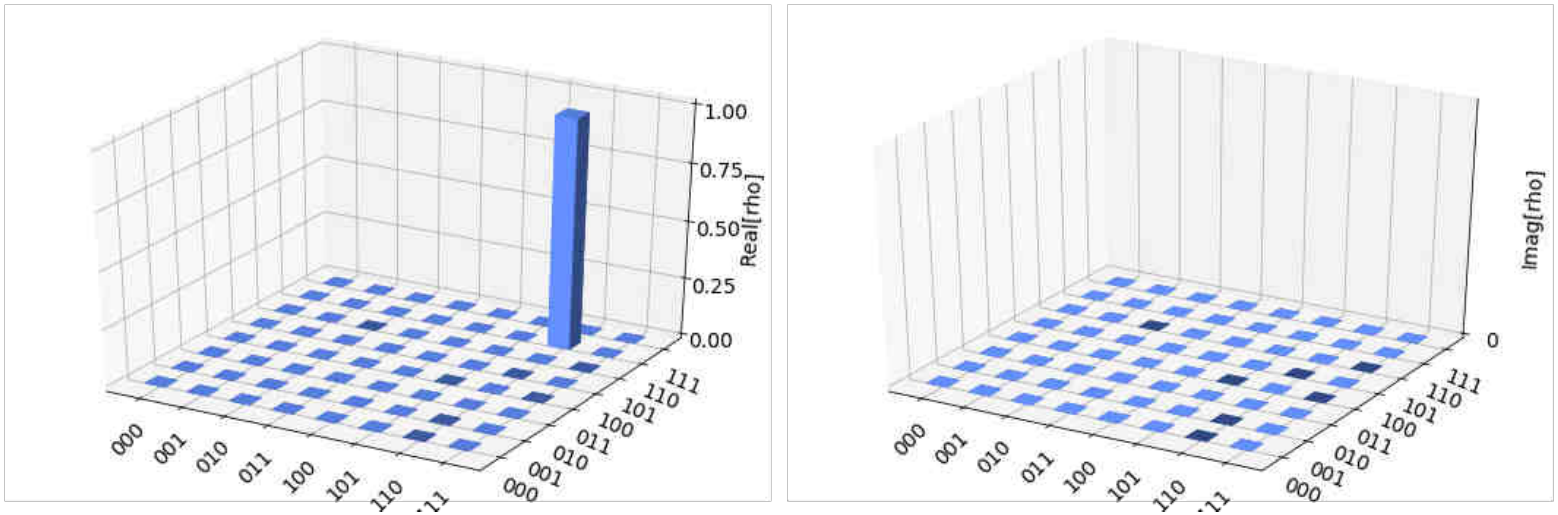


In [14]:

```
# Execute the circuit on the state vector simulator
job_sim = qiskit.execute(circ, SV_SIM)

# Grab the results from the job.
result_sim = job_sim.result()

# Obtain the state vector for the quantum circuit
quantum_state = result_sim.get_statevector(circ, decimals=3)
```

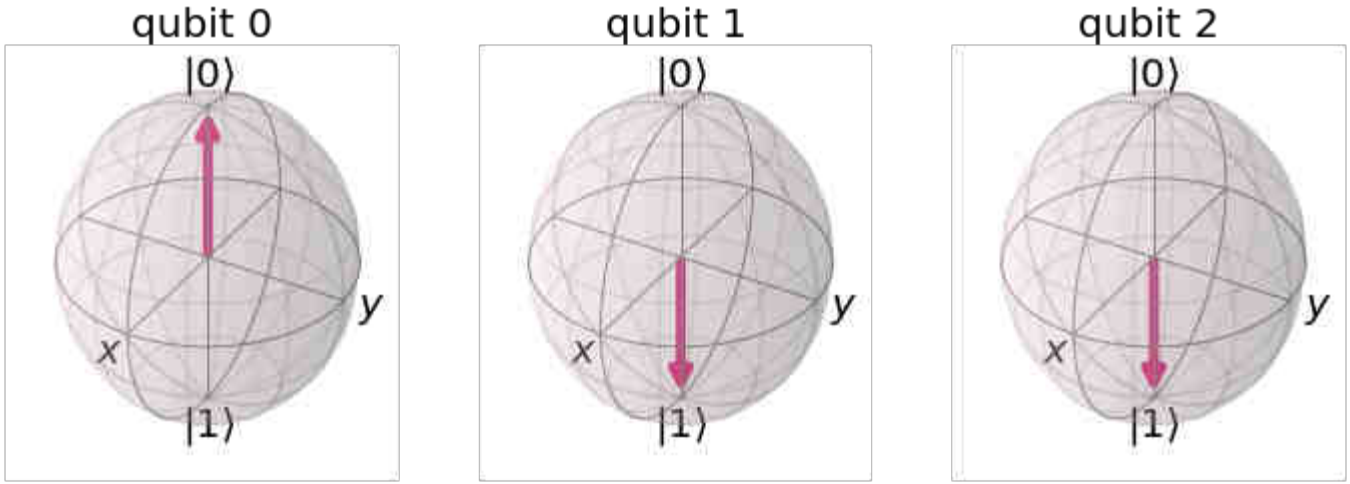
In [15]:
```
qiskit.tools.visualization.plot_state_city(quantum_state)
```

Out[15]:



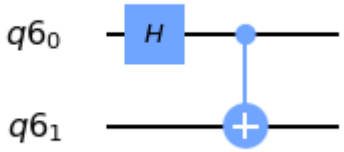
In [16]:
```
qiskit.tools.visualization.plot_bloch_multivector(quantum_state)
```

Out[16]:



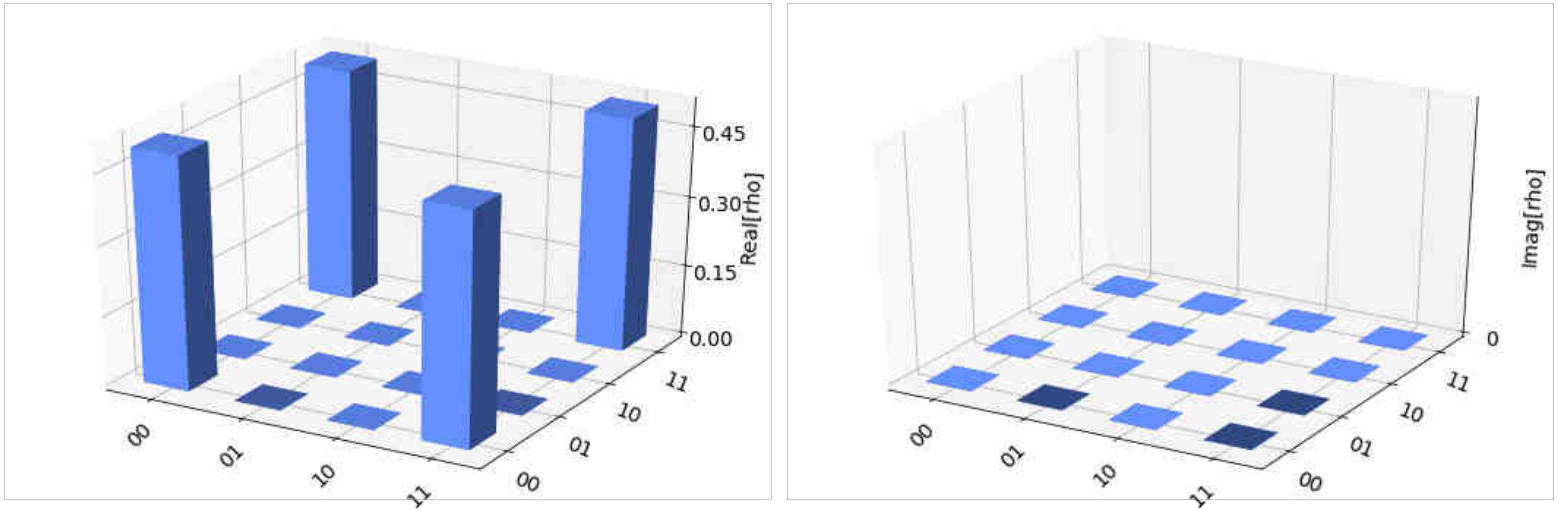
In [17]:
```
# Creating registers
q2 = qiskit.QuantumRegister(2)
c1 = qiskit.ClassicalRegister(1)
c2 = qiskit.ClassicalRegister(2)

# quantum circuit to make an entangled bell state
bell = qiskit.QuantumCircuit(q2)
bell.h(q2[0])
bell.cx(q2[0], q2[1])
bell.draw(output="mpl")
```

Out[17]:

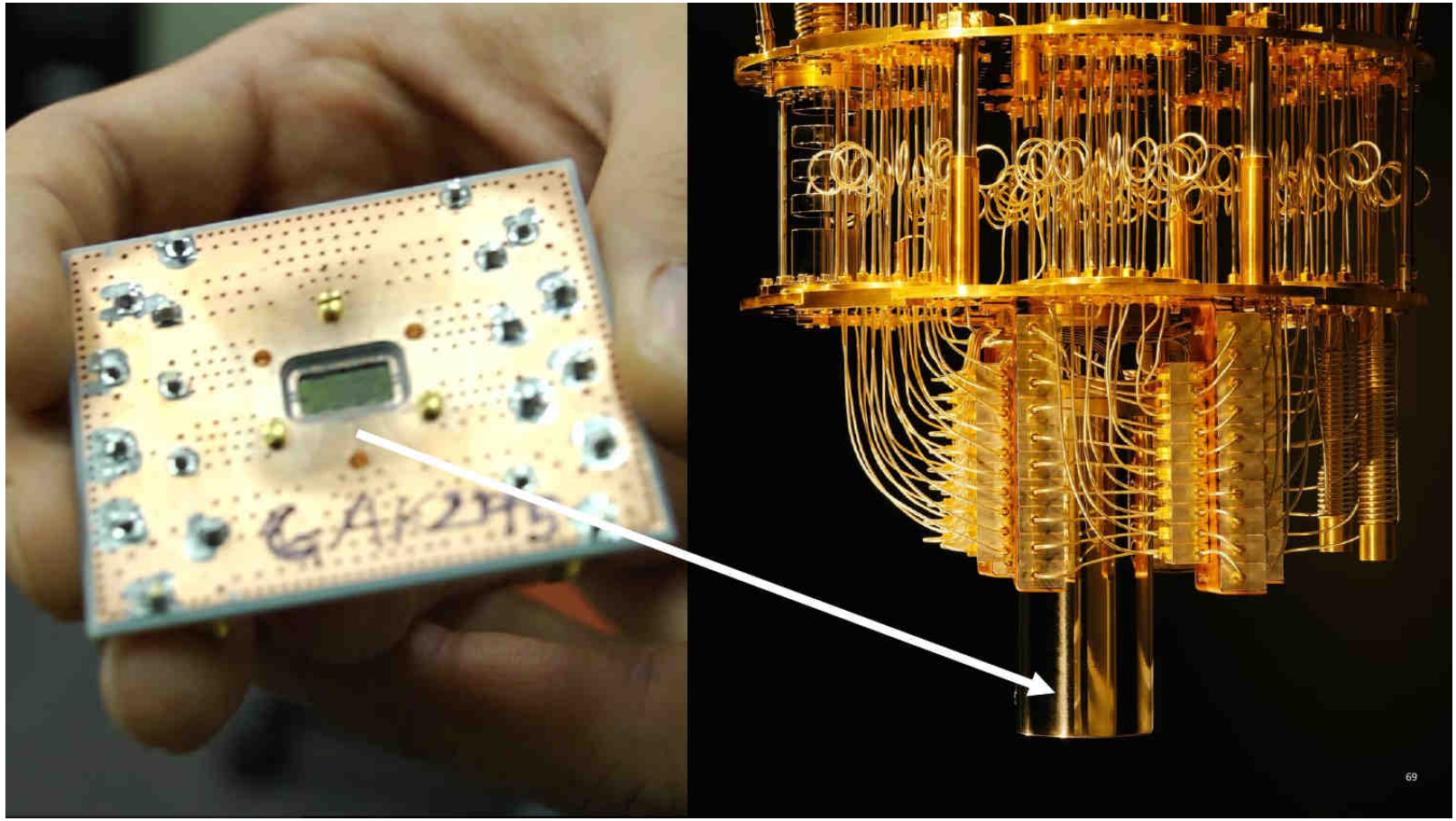


In [18]:
```
result_sv = qiskit.execute(bell, SV_SIM).result()
qiskit.tools.visualization.plot_state_city(result_sv.get_statevector())
```

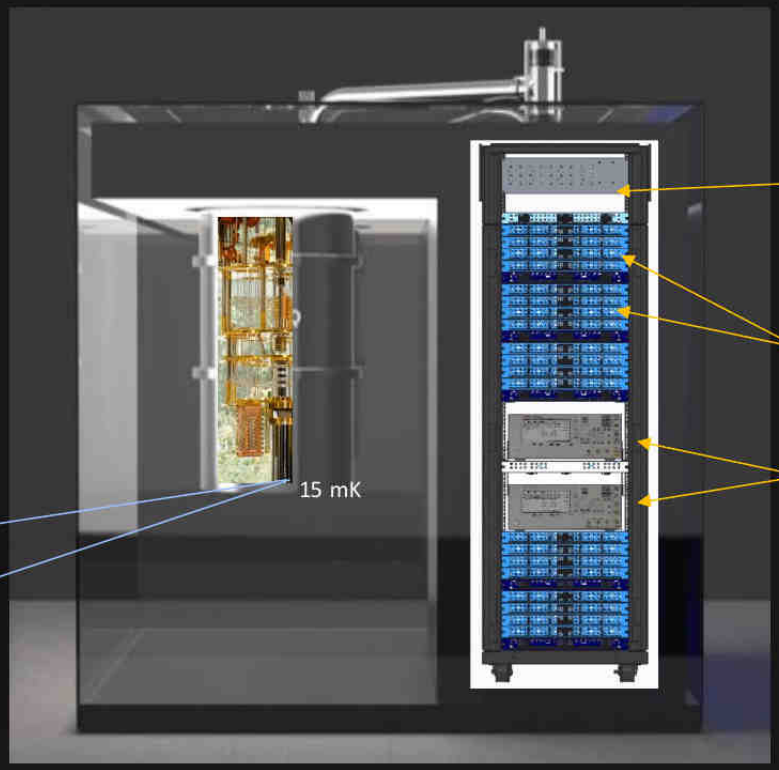
Out[18]:



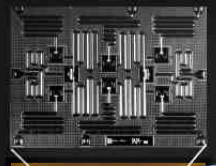
Real Quantum Devices


69

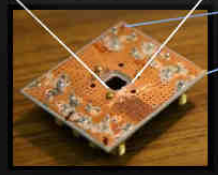
Inside an IBM Q quantum computing system



Qubits on c hip



Circuit board



Analog microwave components and signal digitization

FPGA based microwave pulse shaping control units

Stable microwave sources

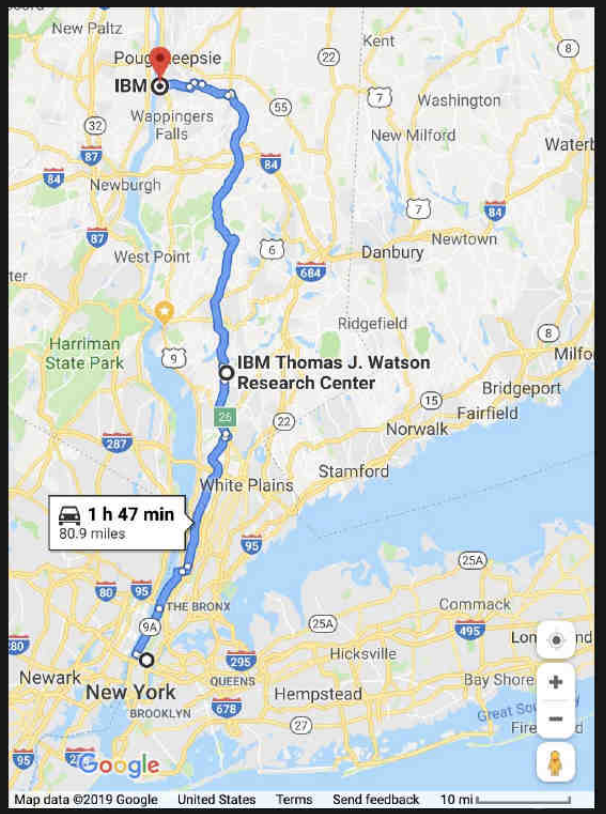
 #IBMQ

IBM News Room Press Tools

IBM Opens Quantum Computation Center in New York; Brings World's Largest Fleet of Quantum Computing Systems Online, Unveils New 53-Qubit Quantum System for Broad Use

Fleet of ten quantum systems benefit from the industry's most powerful hardware with improved stability to propel quantum computing forward

ARMONK, N.Y., Sept. 18, 2019 /PRNewswire/ -- Today, IBM (NYSE: IBM) announced the opening of the IBM Quantum Computation Center in New York State. The new center expands the world's largest fleet of quantum computing systems for commercial and research activity that exist beyond the confines of experimental lab environments. The IBM Quantum Computation Center will support the growing needs of a community of over 150,000 registered users and nearly 80 commercial clients, academic institutions and research laboratories to advance quantum computing and explore practical applications.



Run a Quantum Algorithm on a Real Device!

In [19]:
```
# useful additional packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from qiskit import BasicAer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute
from qiskit.compiler import transpile
from qiskit.tools.monitor import job_monitor
from qiskit.tools.visualization import plot_histogram

qiskit.IBMQ.load_account()
# After loading credentials we query the backends
%qiskit_backend_overview
```

In [20]:
```
IBMQ.get_backend('ibmq_valencia')
```

```
/home/sdague/code/quantum/opensource-quantum-presentation/.venv3/lib/python3.7
/site-packages/qiskit/providers/ibmq/ibmqfactory.py:595: DeprecationWarning: I
BMQ.get_backend() is being deprecated. Please use IBMQ.get_provider() to retri
eve a provider and AccountProvider.get_backend("name") to retrieve a backend.
  DeprecationWarning)
```

Out[20]:
```
<IBMQBackend('ibmq_valencia') from IBMQ(hub='ibm-q-dev', group='cloud', projec
t='main')>
```

Deutsch Jozsa Algorithm

The Deutsch-Jozsa algorithm (http://rspa.royalsocietypublishing.org/content/439/1907/553) is one of the earliest examples demonstrating the power of quantum computers. The algorithm deals with guessing the type of a hidden Boolean function given as an oracle. The Boolean function is promised to be either balanced, i.e., whose values are 0 on half of its inputs, or constant, i.e., whose values are the same on all inputs.

Classically, in the best case, two queries to the oracle can determine if the hidden Boolean function is balanced; in the worst case, at least half of the inputs must be queried to determine if the hidden Boolean function is constant for all inputs. On the other hand, the Deutsch-Jozsa algorithm can determine the Boolean function with one quantum query.

The algorithm in this notebook follows that in Cleve et al. 1997 (https://arxiv.org/pdf/quant-ph/9708016.pdf). We assume that the input to the oracle f is a n-bit string. Namely, for each $x \in \{0,1\}^n$, the value of $f(x)$ is promised to be either constant, i.e., the same for all x, or balanced, i.e., exactly half of the n-bit string whose $f(x) = 0$.

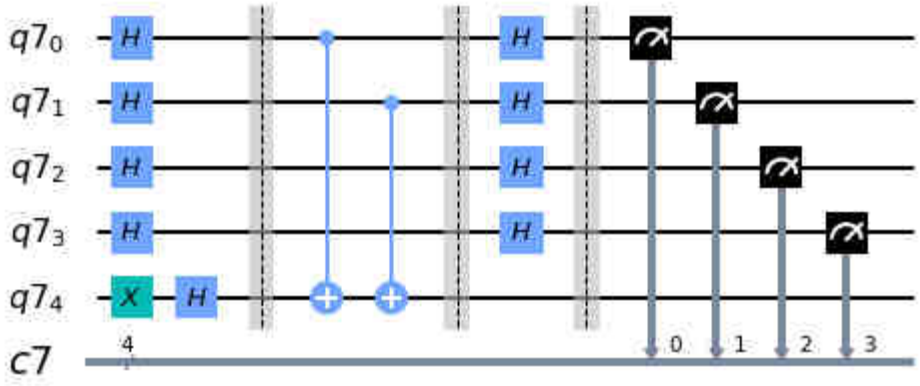
In [21]:
```
n = 4 # the length of the first register for querying the oracle
```

In [22]:

```
# Choose a type of oracle at random. With probability half it is constant,
# and with the same probability it is balanced
oracleType, oracleValue = 1, np.random.randint(2)

if oracleType == 0:
    print("The oracle returns a constant value ", oracleValue)
else:
    print("The oracle returns a balanced function")
    a = np.random.randint(1,2**n) # this is a hidden parameter for balanced ora
cle.

# Creating registers
# n qubits for querying the oracle and one qubit for storing the answer
qr = QuantumRegister(n+1) #all qubits are initialized to zero
# for recording the measurement on the first register
cr = ClassicalRegister(n)

circuitName = "DeutschJozsa"
djCircuit = QuantumCircuit(qr, cr)

# Create the superposition of all input queries in the first register by applyi
ng the Hadamard gate to each qubit.
for i in range(n):
    djCircuit.h(qr[i])

# Flip the second register and apply the Hadamard gate.
djCircuit.x(qr[n])
djCircuit.h(qr[n])

# Apply barrier to mark the beginning of the oracle
djCircuit.barrier()

if oracleType == 0:#If the oracleType is "0", the oracle returns oracleValue fo
r all input.
    if oracleValue == 1:
        djCircuit.x(qr[n])
    else:
        djCircuit.iden(qr[n])
else: # Otherwise, it returns the inner product of the input with a (non-zero b
itstring)
```

In [23]:
```
djCircuit.draw(output="mpl", scale=0.5)
```

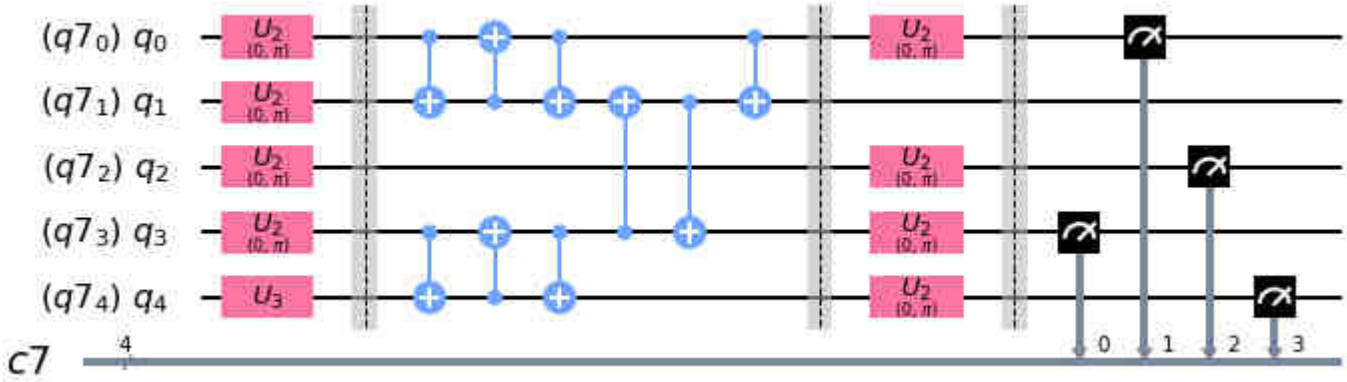
Out[23]:



In [24]:
```
backend = IBMQ.get_backend('ibmq_valencia')
djCompiled = transpile(djCircuit, backend=backend, optimization_level=1)
djCompiled.draw(output='mpl',scale=0.5)
```

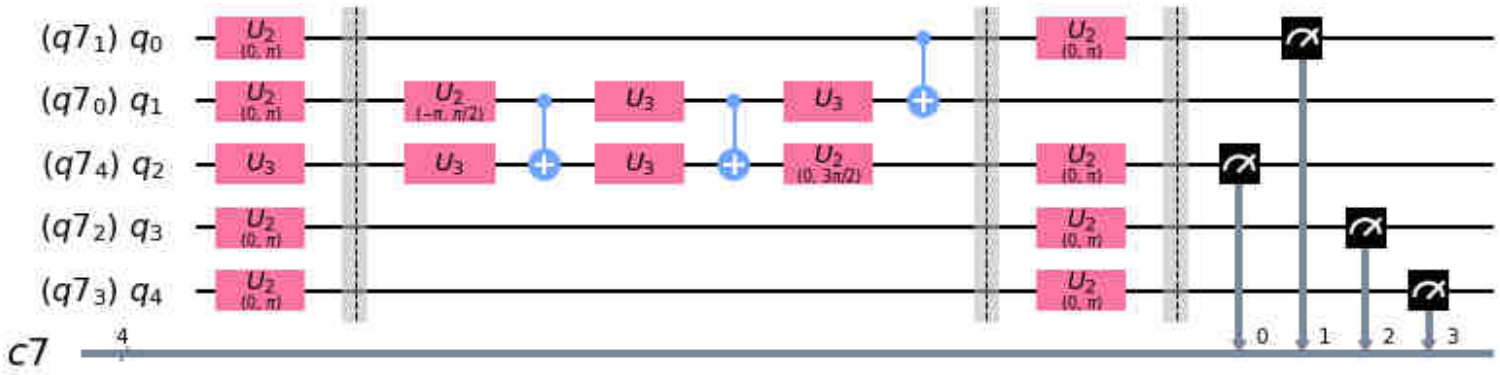
```
/home/sdague/code/quantum/opensource-quantum-presentation/.venv3/lib/python3.7
/site-packages/qiskit/providers/ibmq/ibmqfactory.py:595: DeprecationWarning: I
BMQ.get_backend() is being deprecated. Please use IBMQ.get_provider() to retri
eve a provider and AccountProvider.get_backend("name") to retrieve a backend.
  DeprecationWarning)
```

Out[24]:



In [27]:
```
djCompiled = transpile(djCircuit, backend=backend, optimization_level=3)
djCompiled.draw(output='mpl',scale=0.5)
```

Out[27]:

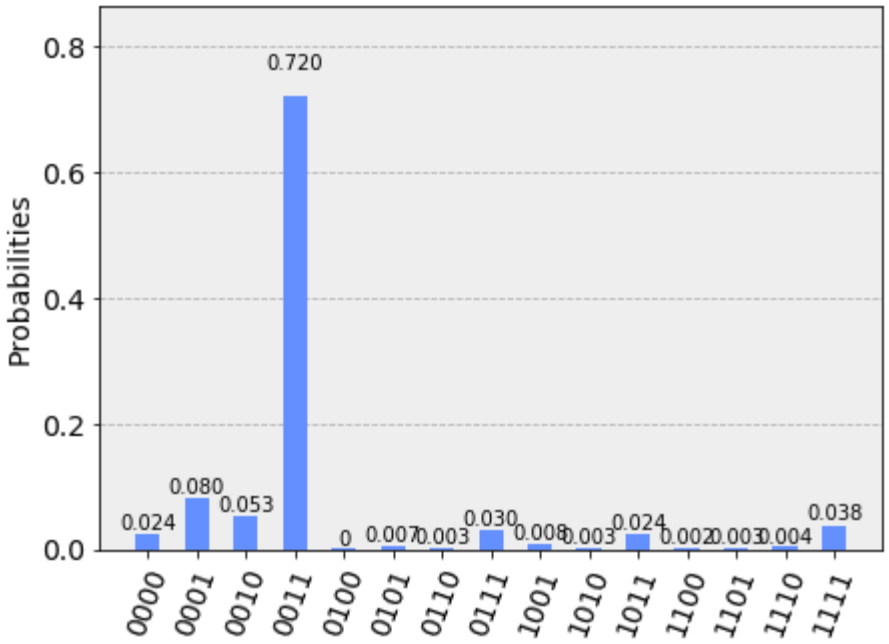


In [28]:
```
job = execute(djCompiled, backend=backend, shots=1024)
job_monitor(job)
```

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

In [29]:
```
results = job.result()
answer = results.get_counts()
plot_histogram(answer)
```

Out[29]:



Diving In

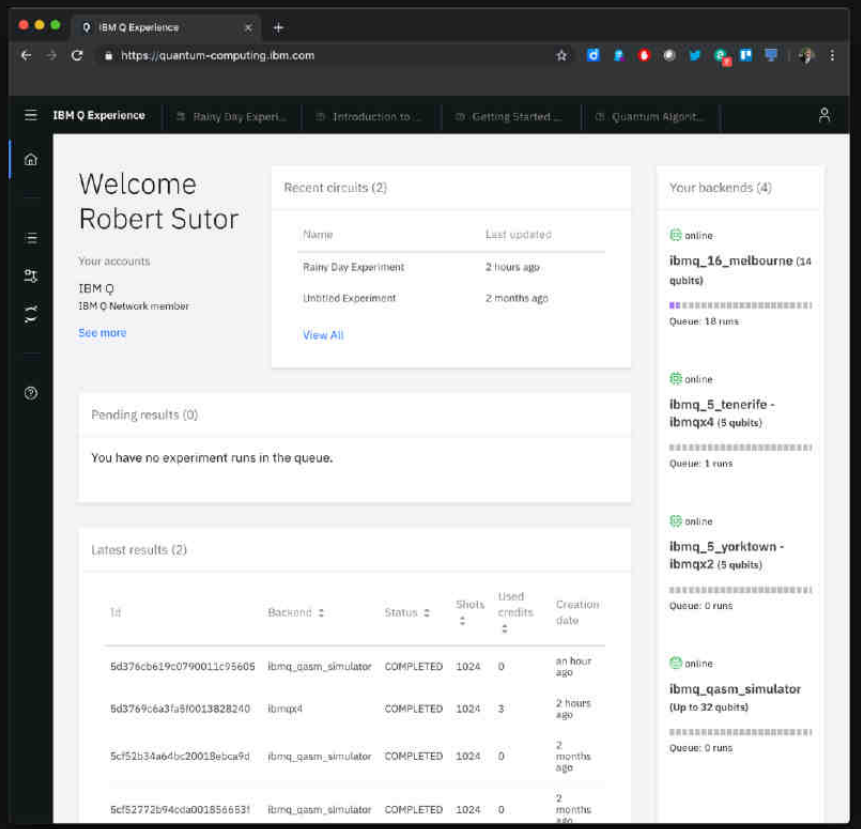
Powerful programming tools for powerful hardware

Open Access

Available to public users to promote the growth of the community and the use of IBM systems.

Premium Access

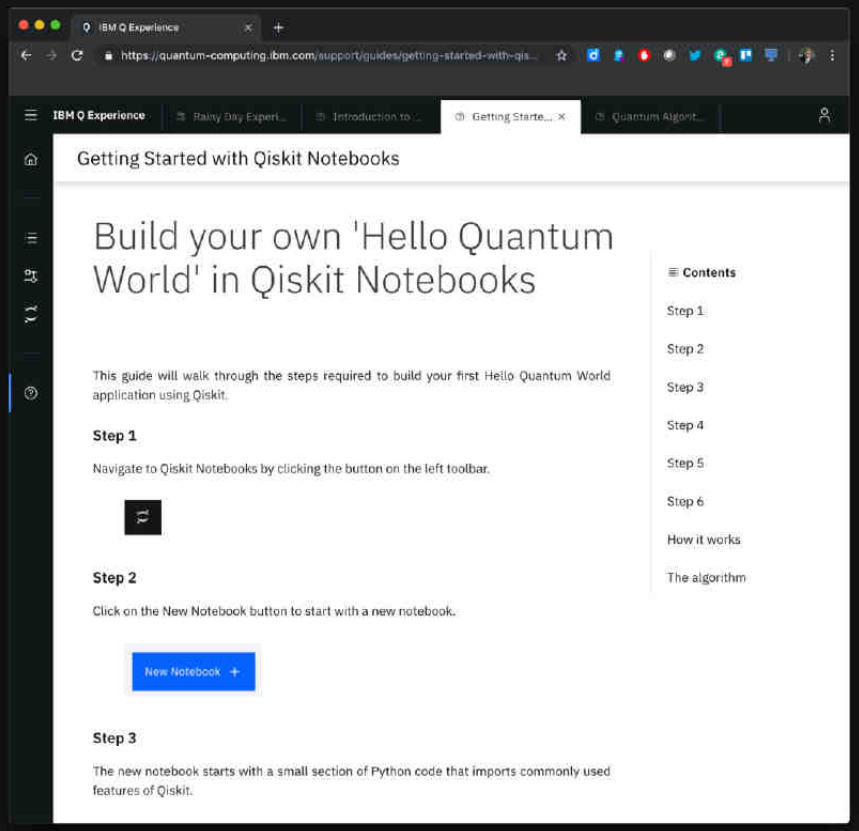
Available to the IBM Q Network to provide differentiated capability to develop for quantum.



 #IBMQ

Powerful programming tools for powerful hardware

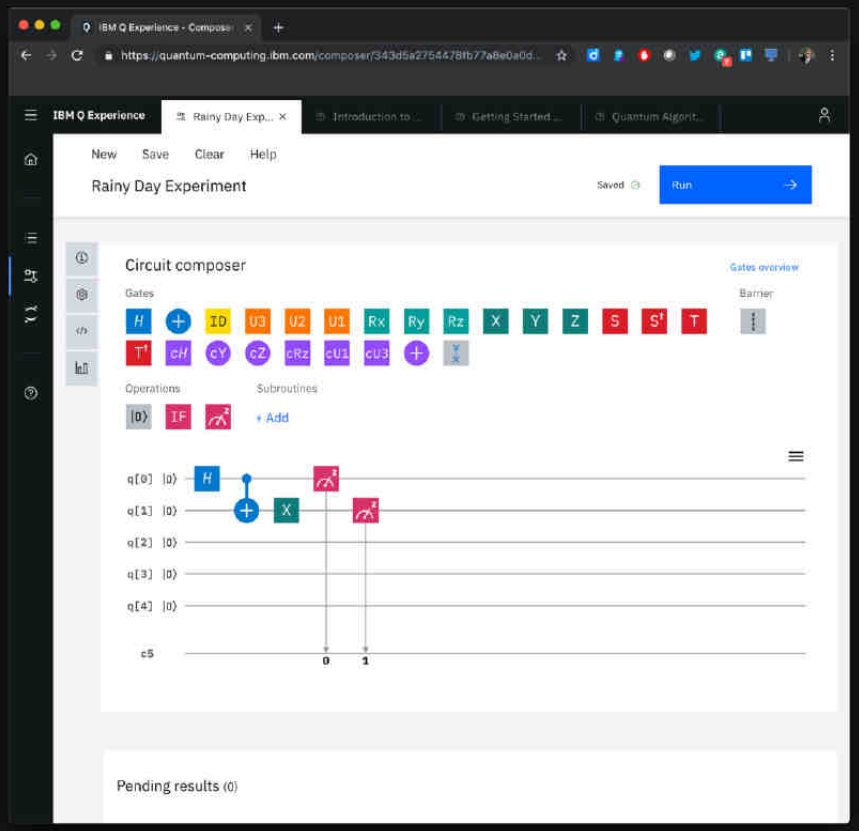
Full documentation shows you how to get started programming the IBM Q quantum computing system.



 #IBMQ

Powerful programming tools for powerful hardware

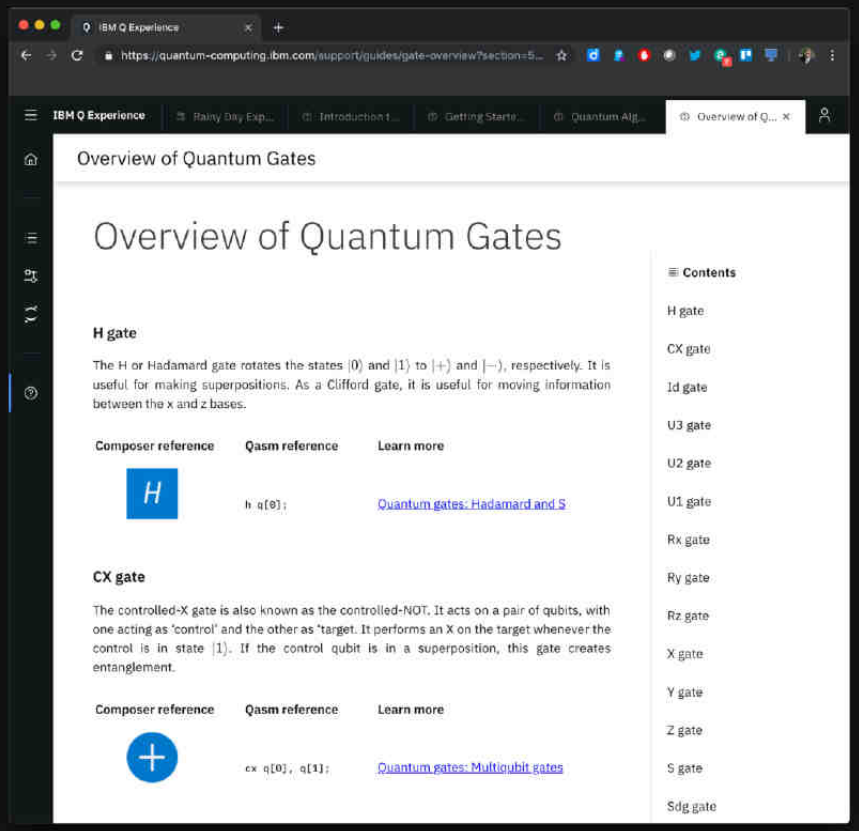
Graphically build, debug, and run quantum circuits on IBM Q hardware systems and simulators.



 #IBMQ

Powerful programming tools for powerful hardware

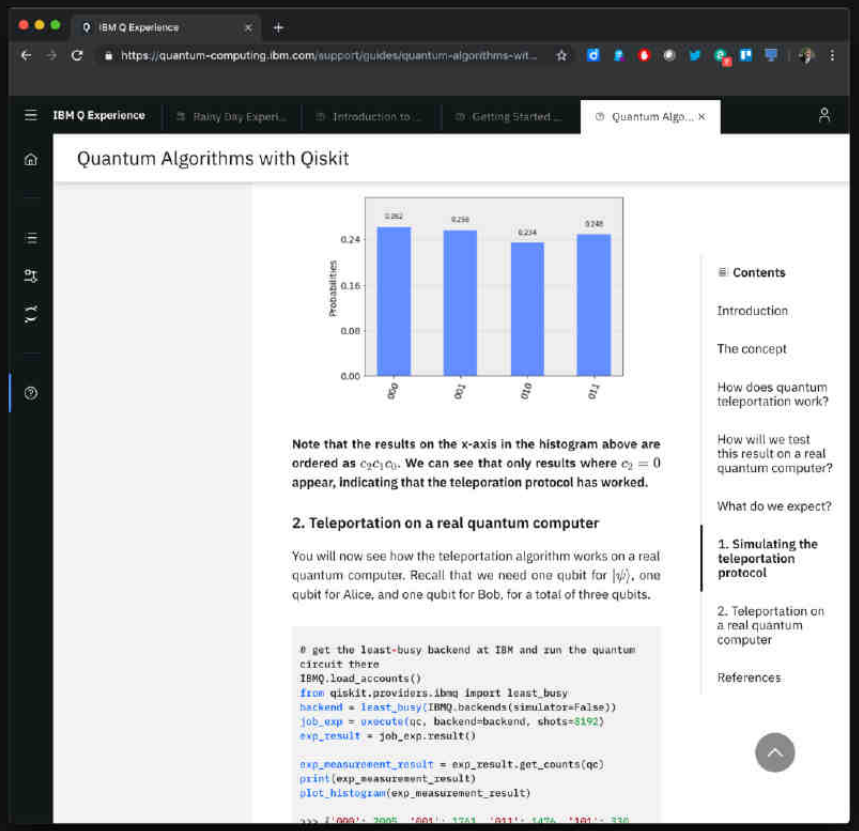
Use the composer to interactively learn the fundamentals of quantum computing.



 #IBMQ

Powerful programming tools for powerful hardware

Programming with Qiskit is seamlessly integrated into the platform – work is saved in the cloud and tools are automatically updated.



 #IBMQ

Thank you!

Questions?

- Contact: sean.dague@ibm.com
- Twitter: @sdague
- IBM Quantum Experience: quantum-computing.ibm.com (https://quantum-computing.ibm.com)