

Qiskit 4 - Bernstein Vazirani

A 4-bit secret bit string is held by a quantum oracle. We are to build a quantum circuit that are able to retrieve the information of that secret string. This will be achieved by the following steps:

1. Define quantum and classical registers.
2. Construct a black-box/oracle and define the secret bitstring.
3. Perform a state-preparation and add the oracle to the circuit.
4. Add a change of basis and measurement.
5. Execute the circuit and visualize the result.
6. Run in cloud

1 - Define registers

In [78]:

```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import Aer, execute

from qiskit.tools.visualization import circuit_drawer, plot_histogram

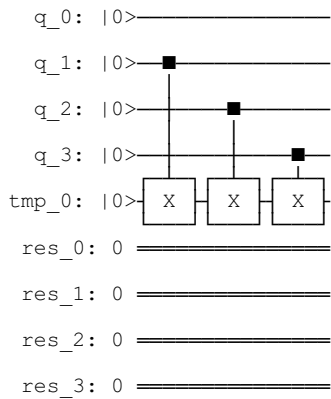
q = QuantumRegister(4, 'q')
tmp = QuantumRegister(1, 'tmp')
res = ClassicalRegister(4, 'res')
```

2 - Construct the oracle

In [79]:

```
secret = 14 # 1110
oracle = QuantumCircuit(q, tmp, res)
for i in range(len(q)):
    if (secret & (1 << i)):
        oracle.cx(q[i], tmp[0])
circuit_drawer(oracle)
```

Out[79]:



3 - State preperation

```
bv = QuantumCircuit(q, tmp, res)
bv.x(tmp) # flip the tmp qubit
bv.barrier(q) # for a nicer diagram
bv.h(q) # add a full layer of hadamard gates to the q qubits
bv.h(tmp) # and add a hadamard to tmp
bv.barrier()
bv += oracle # add the oracle to the bv circuit
bv.barrier()
circuit_drawer(bv)
```

Quantum circuit diagram showing the evolution of a 4-qubit system. The qubits are labeled q_0 , q_1 , q_2 , q_3 , and tmp_0 . The initial state is $|0\rangle$ for all qubits. The circuit consists of the following operations:

- Stage 1: q_0 and tmp_0 both have an X gate.
- Stage 2: q_0 , q_1 , q_2 , and q_3 each have an H gate.
- Stage 3: A CNOT gate from q_1 to tmp_0 .
- Stage 4: A CNOT gate from q_2 to tmp_0 .
- Stage 5: A CNOT gate from q_3 to tmp_0 .

The final state of tmp_0 is measured into res_0 , res_1 , res_2 , and res_3 .

In [81]:

```

bv.h(q) # full layer of hadamard
bv.h(tmp) # hadamard on tmp qubit which is then discarded
bv.measure(q, res) # measure the q qubits
circuit_drawer(bv)

```

The diagram illustrates a quantum circuit for a quantum walk algorithm. It features five qubits at the top: q_0 , q_1 , q_2 , q_3 , and tmp_0 . Each of these qubits starts in the $|0\rangle$ state. The circuit is divided into stages by vertical dashed lines.

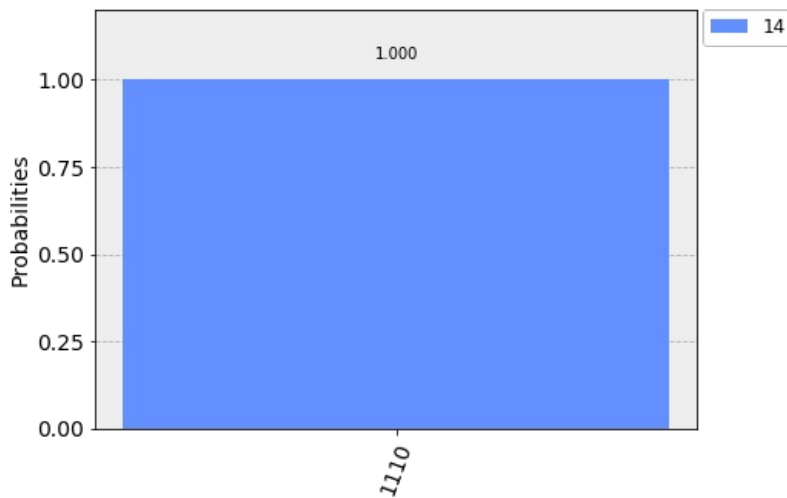
- Initial State:** All qubits are initialized to $|0\rangle$.
- Stage 1:** A Hadamard (H) gate is applied to each of the four qubits (q_0, q_1, q_2, q_3). The tmp_0 qubit has an 'X' gate.
- Stage 2:** Controlled operations are performed between the four qubits and tmp_0 . Specifically, there are controlled-X gates from q_1 to tmp_0 , q_2 to tmp_0 , and q_3 to tmp_0 .
- Stage 3:** Another set of Hadamard (H) gates is applied to q_0, q_1, q_2, q_3 . The tmp_0 qubit remains unchanged.
- Measurement Stage:** Measurements (M) are performed on all five qubits ($q_0, q_1, q_2, q_3, \text{tmp}_0$) simultaneously.
- Classical Registers:** Below the qubits, there are four classical registers labeled $\text{res}_0, \text{res}_1, \text{res}_2,$ and res_3 , each initialized to 0. These registers store the results of the measurements performed on $q_0, q_1, q_2,$ and q_3 respectively.

5 - Execute

In [82]:

```
sim = Aer.get_backend('qasm_simulator')
job = execute(bv, sim, shots=1000)
count = job.result().get_counts()
plot_histogram(count, legend=[str(secret)])
```

Out[82]:



We can easily change up the number of qubits and handle larger bitstrings, we also get correct results when we vary the secret string. Key parts here though is the query speed up, and the insight that if you are in a position where you can construct a quantum oracle on your problem, it is nice. Now lets have a look at Grover.

Also, keep in mind that whole mechanism with having a 'spare' qubit going in to your oracle/blackbox/unitary that you can dump information and operations on as per whatever requirements. Then, it seems, one simply discard it upon measurement. Why? It is clear that you can obtain perhaps 'enough' information by reading off the qubits corresponding to the length of the bit string that your trying to obtain...

6 - Run in cloud

In [83]:

```
from qiskit import IBMQ
IBMQ.save_account('YOUR_API_KEY')
```

In [87]:

```
IBMQ.load_account()
```

Out[87]:

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

In [88]:

```
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
```

Out[88]:

```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_vigo') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_london') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_burlington') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_essex') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

In [89]:

```
backend = provider.get_backend('ibmq_essex')
```

In [90]:

```
job_cloud = execute(bv, backend, shots=1000)
```

In [91]:

```
count_cloud = job_cloud.result().get_counts()
print(count_cloud)
```

```
{'0010': 65, '1100': 55, '1101': 39, '0000': 65, '1000': 23, '1111': 166, '1010': 50, '0011': 45, '0111': 41, '1001': 19, '0101': 25, '0100': 31, '1110': 238, '0110': 46, '1011': 34, '0001': 58}
```

In [92]:

```
plot_histogram(count_cloud, legend=['14, 1110, quantum computed'])
```

Out[92]:

