BB84 Key Distribution Protocol in Qiskit

In Part 1 of this assignment, you learnt about a simple quantum key distribution protocol (called BB84, after C. H. Bennett and G. Brassard) for securely generating random bitstrings which can then be used to encrypt information to securely share it between parties. Implement this protocol in Q# using the template given below.

In [2]:
```python
from qiskit import QuantumCircuit, Aer
from qiskit.quantum_info import Statevector
from qiskit.primitives import Sampler
from qiskit.visualization import plot_histogram
import random
```

Part 1: Warm Up

Task 1.1

1. Prepare a qubit in the $|1\rangle$ state
2. Create a barrier
3. Measure the qubit in the Z-basis

What is the probability of observing `0`?

In [76]:
```python
qc = QuantumCircuit(1,1)

# ANSWER
qc.x(0)
qc.barrier(0)
qc.measure(0,0)
qc.draw()
```

Out[76]:
```
q: ┤ X ├░┤M├
c: 1/═══════╩═
            0
```

In [87]:
```python
# RUN ON A SIMULATOR
sampler = Sampler()
sampler.set_options(shots=1024)
results = sampler.run(qc).result()

results
#0% observing 0
```

Out[87]: 1

Task 1.2

1. Prepare a qubit in the $|1\rangle$ state
2. Apply a Hadamard gate
3. Measure in the X-basis

What is the probability of observing 0? Run the experiment for a 1024 shots on any simulator.

In [19]:
```python
qc = QuantumCircuit(1,1)

# ANSWER
qc.x(0)
qc.h(0)
qc.measure(0,0)
qc.draw()
```

Out[19]:
```
q: ┤ X ├┤ H ├┤M├
c: 1/═════════╩═
              0
```

In [20]:
```python
# RUN ON A SIMULATOR
sampler = Sampler()
sampler.set_options(shots=1024)
results = sampler.run(qc).result()

results
#48.5% observing 0
```

Out[20]: `SamplerResult(quasi_dists=[{0: 0.4853515625, 1: 0.5146484375}], metadata=[{'shots': 1024}])`

Task 1.3

1. Prepare a qubit in the $|+\rangle$ state
2. Measure in the Z-basis

What is the probability of measuring 0? What is the probability of measuring 1?

Solution

In [22]:
```python
qc = QuantumCircuit(1,1)

# ANSWER
qc.h(0)
qc.measure(0,0)
qc.draw()
# RUN ON A SIMULATOR
sampler = Sampler()
sampler.set_options(shots=1024)
results = sampler.run(qc).result()

results
```

Out[22]: `SamplerResult(quasi_dists=[{0: 0.515625, 1: 0.484375}], metadata=[{'shots': 1024}])`

Part 2: Protocol Example

The protocol makes use of the fact that measuring a qubit can change its state. If Alice sends Bob a qubit, and an eavesdropper (Eve) tries to measure it before Bob does, there is a chance that Eve's measurement will change the state of the qubit and Bob will not receive the qubit state Alice sent.

Task 2.1

Alice prepares a qubit in the state $|+\rangle$ (0 in the X-basis), and Bob measures it in the X-basis. Write the quantum circuit corresponding to this. What is the probability of measuring `0`?

Solution

In [35]:
```python
qc = QuantumCircuit(2,1)
qc.h(0)
qc.cx(0,1)
qc.measure(0,0)
qc.draw()
```

Out[35]:
```
q_0: ┤ H ├──■──┤M├
q_1: ──────┤ X ├─╫─
c: 1/════════════╩═
                 0
```

In [33]:
```python
sampler = Sampler()
sampler.set_options(shots=1024)
results = sampler.run(qc).result()

results
```

Out[33]: `SamplerResult(quasi_dists=[{0: 0.5009765625, 1: 0.4990234375}], metadata=[{'shots': 1024}])`

If Eve tries to measure this qubit in the Z-basis before it reaches Bob, she will change the qubit's state from $|+\rangle$ to either $|0\rangle$ or $|1\rangle$, and Bob is no longer certain to measure `0`. Bob now has a 50% chance of measuring `1`, and if he does, he and Alice will know there is something wrong with their channel.

The quantum key distribution protocol involves repeating this process enough times that an eavesdropper has a negligible chance of getting away with this interception.

Part 3: BB84 in Qiskit

Task 3.1: Random Bits

Implement an operation that returns a random bit. This will be used in further operations.

Summary

Returns a random bit (0 or 1) with equal probability.

Output

A random integer in {0, 1}

In [47]:
```python
def RandomBit():
    random_int = random.choice([0,1])
    return random_int

RandomBit()
```

Out[47]: 0

Task 3.2: Alice encodes a random bitstring

The first step of this protocol is for Alice to generate random bitstrings and initializing `n` single qubit QuantumCircuits randomly to $|0\rangle, |1\rangle, |+\rangle \text{ or } |-\rangle$.

Your task is to do the following:

1. Create a list of circuits with one qubit each representing Alice's qubits.
2. Generate two lists of random integers `bits` and `bases`
3. If bits[i] is 0, prepare the qubit in $|0\rangle$ state. If bits[i] is 1, prepare the qubit in the $|1\rangle$ state.
4. If bases[i] is 0, encode the qubit in the Z-basis
5. If bases[i] is 1, encode the qubit in the X-basis

Note that $|0\rangle$ and $|1\rangle$ are the basis vectors in the Z-basis and $|+\rangle$ and $|-\rangle$ are the basis vectors in the X-basis. Therefore, we can say that Alice is encoding the random bitstring S_A either in the Z-basis ($|0\rangle$ and $|1\rangle$) or the X-basis ($|+\rangle$ and $|-\rangle$).

Summary

This operation prepares each of the qubits in the input array in one of the $|0\rangle, |1\rangle, |+\rangle \text{ or } |-\rangle$. states randomly.

Input *n*: Number of bits to encode

Returns

1. `message`: A list of QuantumCircuits with the prepared qubits
2. `bits`: The chosen bits
3. `bases`: The chosen bases

In [169…
```python
def alice_encoding(n):
    quantumCircuits = []
    randomBits = []
    randomBases = []
    for num in range(n):
        randomBits.append(RandomBit())
        randomBases.append(RandomBit())

    for num in range(n):
        qc = QuantumCircuit(1,1)
        if randomBits[num] != 0:
            qc.x(0)
        if randomBases[num] != 0:
            qc.h(0)
        quantumCircuits.append(qc)
    return quantumCircuits, randomBits, randomBases
```

Out[169…
```
[<qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121344890>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121347210>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121345810>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121344790>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121347090>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121347250>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121347e50>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121311150>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x11feaa4d0>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x121345f90>]
```

Task 3.3: Bob measures the circuits in the encoded message in some random bases (Z or X)

In the next step of the protocol, Bob receives the encoded message from Alice and randomly selects bases to measure in.

Summary

Measures each qubit in a randomly chosen basis, X or Z, and returns the chosen bases and the measurement results.

Input `encoded_message`: A list of `QuantumCircuit` objects

Returns

1. `decoded_message`: A list of integers $\in \{0, 1\}$ based on measurement outcomes of the `encoded_message`
2. `measured_bases`: The list of random bases Bob chose to measure in. `0` indicates Z-basis, `1` indicates X-basis

In [170…
```python
def bob_decoding(encoded_message):
    measured_bases = []
    decoded_message = []

    for qubit in range(len(encoded_message)):
        random_basis = RandomBit()
        measured_bases.append(random_basis)
        qc = encoded_message[qubit]
        if random_basis != 0:
            qc.h(0)
        qc.measure(0,0)
        #run on sim
        sampler = Sampler()
        sampler.set_options(shots=1024)
        results = sampler.run(qc).result()
        decoded_message.append(results.quasi_dists[0])

    return measured_bases, decoded_message
```

Out[170…
```
[{1: 1.0},
 {0: 0.4912109375, 1: 0.5087890625},
 {0: 1.0},
 {0: 1.0},
 {0: 0.498046875, 1: 0.501953125},
 {0: 1.0},
 {0: 0.494140625, 1: 0.505859375},
 {1: 1.0},
 {0: 1.0},
 {0: 0.517578125, 1: 0.482421875}]
```

Task 3.4: Putting it all together

So now we just need to call our operations in the right order to perform key distribution. The length of generated keys is probabilistic, since we do not know how many times the sender and the receiver will choose the same basis. Our BB84 operation takes an argument N which specifies the number of protocol iterations we run. The length of our key will be between 0 and N, and on average it will be $\frac{1}{2}N$.

Summary

This operation acts as the intermediary exchanging both classical and quantum information between parties. N controls the number of qubits transmitted, so our final key will be of length less than or equal to N.

Input

`n`: The number of qubits to attempt key distribution with.

Returns

1. The preparation bases the sender (Alice) used.
2. The measurement bases the receiver (Bob) used.
3. The bits that the sender encoded.
4. The bits that were measured by the receiver.
5. Alice's key that the Alice made after comparing bases.
6. Bob's key from measurements of of the quantum circuits

In [176…
```python
def bb84_protocol(n):
    alice_qc, alice_randomBits, alice_randomBases = alice_encoding(n)
    bob_measured_bases, bob_decoded_message = bob_decoding(alice_qc)
    alice_key = []
    bob_key = []
    for index in range(n):
        if alice_randomBases[index] == bob_measured_bases[index]:
            alice_key.append(alice_randomBases[index])
        if len(bob_decoded_message[index]) == 1:
            bob_key.append(list(bob_decoded_message[index].keys())[0])
            bob_decoded_message[index] = list(bob_decoded_message[index].keys())[0]

    return alice_randomBases, bob_measured_bases, alice_randomBits, bob_decoded_message, alice_key, bob_key
```

Check your work

In [178…
```python
def formatOutput(basesS, basesR, bitS, bitR, key1, key2):
    keyCopy = key2.copy()
    same = "|"
    basisSentChar = "|"
    basisRecChar = "|"
    bitSent = "|"
    bitRec = "|"
    keyS = "|"
    keyR = "|"
    stateSent = "|"
    for i in range(len(basesR)):
        bitSent += ' 1 |' if bitS[i] == 1 else ' 0 |'
        bitRec += ' 1 |' if bitR[i] == 1 else ' 0 |'
        same += ' y |' if basesR[i] == basesS[i] else ' n |'
        keyS += ' {} |'.format(bitS[i]) if basesR[i] == basesS[i] else "   |"
        keyR += ' {} |'.format(key2.pop(0)) if basesR[i] == basesS[i] else "   |"
        basisRecChar += ' Z |' if basesR[i] == 0 else ' X |'
        if basesS[i] == 0:
            stateSent += "|0>|" if bitS[i] == 0 else "|1>|"
            basisSentChar += " Z |"
        else:
            stateSent += "|+>|" if bitS[i] == 0 else "|->|"
            basisSentChar += " X |"

    print("Let's compare this to the selected bases, and the transmitted qubit states")
    print("Alice's bases were:       {}".format(basisSentChar))
    print("Bob's bases were:         {}".format(basisRecChar))
    print("Both bases match (yes/no):{}".format(same))
    print("Alice's encoded bit:      {}".format(bitSent))
    print("The states sent were:     {}".format(stateSent))
    print("Bob measured:             {}".format(bitRec))
    print("Notice how the key is formed by the bits where bases are equal")
    print("Bob's key:                {}".format(keyR))
    print("Alice's key:              {}".format(keyS))
    print("The key that was generated was {}\n".format(keyCopy))
(basesS, basesR, bitS, bitR, alice_key, bob_key) = bb84_protocol(10)
formatOutput(basesS, basesR, bitS, bitR, alice_key, bob_key)
```

```
Let's compare this to the selected bases, and the transmitted qubit states
Alice's bases were:       | X | X | Z | X | X | Z | Z | X | X | X |
Bob's bases were:         | X | X | Z | Z | Z | Z | X | Z | X | Z |
Both bases match (yes/no):| y | y | y | n | n | y | n | n | y | n |
Alice's encoded bit:      | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
The states sent were:     ||->||+>||0>||+>||->||0>||0>||+>||+>||->|
Bob measured:             | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
Notice how the key is formed by the bits where bases are equal
Bob's key:                | 1 | 0 | 0 |   |   | 0 |   |   | 0 |   |
Alice's key:              | 1 | 0 | 0 |   |   | 0 |   |   | 0 |   |
The key that was generated was [1, 0, 0, 0, 0]
```

In [ ]:

In [ ]: