Without Interception

In [1]:

```
from qiskit import QuantumCircuit, Aer, transpile, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from numpy.random import randint
import numpy as np
print("Imports Successful")
```

```
Imports Successful
```

Let’s first see how the protocol works when no one is listening in, then we can see how Alice and Bob are able to detect an eavesdropper. As always, let's start by importing everything we need:

To generate pseudo-random keys, we will use the randint function from numpy. To make sure you can reproduce the results on this page, we will set the seed to 0:

In [2]:

```
np.random.seed(seed=0)
```

In [3]:

```
# We will call the length of Alice's initial message n. In this example, Alice will

n = 100
```

Step 1

Alice generates her random set of bits:

In [4]:

```
np.random.seed(seed=0)
n = 100
## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)
print(alice_bits)
```

```
[0 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 0
 1 1 0 0 1 0 1 1 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1 1 0 1 0 1 0 0 0 0 0 1 1 0 0
 0 1 1 0 1 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0 1 0 0 1 1 0]
```

At the moment, the set of bits 'alice_bits' is only known to Alice. We will keep track of what information is only known to Alice, what information is only known to Bob, and what has been sent over Eve's channel in a table like this:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		

Step 2

Alice chooses to encode each bit on qubit in the X or Z-basis at random, and stores the choice for each qubit in alice_bases. In this case, a 0 means "prepare in the Z-basis", and a 1 means "prepare in the X-basis":

In [5]:

```
## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
print(alice_bases)
```

```
[1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 0 1 1 0 0 1 0
 0 0 0 1 1 0 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 0
 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0]
```

Alice also keeps this knowledge private:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		

The function `encode_message` below, creates a list of `QuantumCircuits`, each representing a single qubit in Alice's message:

In [6]:

```
def encode_message(bits, bases):
    message = []
    for i in range(n):
        qc = QuantumCircuit(1,1)
        if bases[i] == 0: # Prepare qubit in Z-basis
            if bits[i] == 0:
                pass
            else:
                qc.x(0)
        else: # Prepare qubit in X-basis
            if bits[i] == 0:
                qc.h(0)
            else:
                qc.x(0)
                qc.h(0)
        qc.barrier()
        message.append(qc)
    return message
```

In [7]:

```
message = encode_message(alice_bits, alice_bases)
```

We can see that the first bit in alices_bits is 0, and the basis she encodes this in is the X-basis (represented by 1):

In [8]:

```
print('bit = %i' % alice_bits[0])
print('basis = %i' % alice_bases[0])
```

```
bit = 0
basis = 1
```

And if we view the first circuit in message (representing the first qubit in Alice's message), we can verify that Alice has prepared a qubit in the state $|+\rangle$:

In [9]:

```
message[0].draw()
```

Out[9]:

```
        ┌───┐ ░ 
q_0: ─┤ H ├─░─
        └───┘ ░ 
c: 1/═════════
```

This message of qubits is then sent to Bob over Eve's quantum channel:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message

Step 3

Bob then measures each qubit in the X or Z-basis at random and stores this information:

In [10]:
```
## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
print(bob_bases)
```

```
[1 0 1 0 0 1 1 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0 0 1 1 1 1 0 0 0 1 1
 0 1 0 0 1 0 1 1 1 1 0 0 0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 0 1 1 0 1 1 1 1 1 0
 0 0 1 0 1 0 1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 0 0 0 1 1]
```

`bob_bases` stores Bob's choice for which basis he measures each qubit in.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases

Below, the function `measure_message` applies the corresponding measurement and simulates the result of measuring each qubit. We store the measurement results in `bob_results`.

In [11]:
```
def measure_message(message, bases):
    backend = Aer.get_backend('aer_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Z-basis
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in X-basis
            message[q].h(0)
            message[q].measure(0,0)
        aer_sim = Aer.get_backend('aer_simulator')
        qobj = assemble(message[q], shots=1, memory=True)
        result = aer_sim.run(qobj).result()
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements
```

In [12]:
```
bob_results = measure_message(message, bob_bases)
```

In [13]:
```
message[0].draw()
```

Out[13]:
```
     ┌───┐ ░ ┌───┐┌─┐
q_0: ┤ H ├─░─┤ H ├┤M├
     └───┘ ░ └───┘└╥┘
```

```
     └───┘ ░ └───┘ └╥┘
c: 1/═══════════════╩═
                    0 
```

Since Bob has by chance chosen to measure in the same basis Alice encoded the qubit in, Bob is guaranteed to get the result 0. For the 6th qubit (shown below), Bob's random choice of measurement is not the same as Alice's, and Bob's result has only a 50% chance of matching Alices'.

In [14]:

```python
message[6].draw()
```

Out[14]:

```
     ┌───┐ ░ ┌───┐┌─┐
q_0: ┤ X ├─░─┤ H ├┤M├
     └───┘ ░ └───┘└╥┘
c: 1/══════════════╩═
                   0 
```

In [15]:

```python
print(bob_results)
```

```
[0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0,
1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0,
1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1,
0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0]
```

Bob keeps his results private.

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results

Step 4

After this, Alice reveals (through Eve's channel) which qubits were encoded in which basis:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases

And Bob reveals which basis he measured each qubit in:

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	

If Bob happened to measure a bit in the same basis Alice prepared it in, this means the entry in bob_results will match the corresponding entry in alice_bits, and they can use that bit as part of their key. If they measured in different bases, Bob's result is random, and they both throw that entry away. Here is a function remove_garbage that does this for us:

In [16]:

```
def remove_garbage(a_bases, b_bases, bits):
    good_bits = []
    for q in range(n):
        if a_bases[q] == b_bases[q]:
            # If both used the same basis, add
            # this to the list of 'good' bits
            good_bits.append(bits[q])
    return good_bits
```

In [17]:

```
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
print(alice_key)
```

```
[0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0]
```

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		

In [18]:

```
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
print(bob_key)
```

```
[0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0]
```

Alice's Knowledge	Over Eve's Channel	Bob's Knowledge
alice_bits		
alice_bases		
message	message	message
		bob_bases
		bob_results
	alice_bases	alice_bases
bob_bases	bob_bases	
alice_key		bob_key

Step 5

Finally, Bob and Alice compare a random selection of the bits in their keys to make sure the protocol has worked correctly:

In [19]:

```
def sample_bits(bits, selection):
    sample = []
    for i in selection:
        # use np.mod to make sure the
        # bit we sample is always in
        # the list range
        i = np.mod(i, len(bits))
        # pop(i) removes the element of the
        # list at index 'i'
        sample.append(bits.pop(i))
    return sample
```

Alice and Bob both broadcast these publicly, and remove them from their keys as they are no longer secret:

In [20]:
```
## Step 5
sample_size = 15
bit_selection = randint(n, size=sample_size)

bob_sample = sample_bits(bob_key, bit_selection)
print("  bob_sample = " + str(bob_sample))
alice_sample = sample_bits(alice_key, bit_selection)
print("alice_sample = "+ str(alice_sample))
```

```
  bob_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
alice_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

If the protocol has worked correctly without interference, their samples should match:

In [21]:
```
bob_sample == alice_sample
```

Out[21]:
```
True
```

In [22]:
```
print(bob_key)
print(alice_key)
print("key length = %i" % len(alice_key))
```

```
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 0, 0]
[1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1,
1, 0, 1, 0, 0]
key length = 33
```

With Interception

Let's now see how Alice and Bob can tell if Eve has been trying to listen in on their quantum message. We repeat the same steps as without interference, but before Bob receives his qubits, Eve will try and extract some information from them. Let's set a different seed so we get a specific set of reproducible 'random' results:

In [23]:
```
np.random.seed(seed=3)
```

Step 1

In [24]:
```
#Alice generates her set of random bits:
alice_bits = randint(2, size=n)
print(alice_bits)
```

```
[0 0 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1
 0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 1 0 1 0 0 1 1 0 1 0 0 0 0 0 1 1 0 1 1]
```

Step 2

Alice encodes these in the Z and X-bases at random, and sends these to Bob through Eve's quantum channel:

In [25]:
```
## Step 2
```

```
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)
print(alice_bases)
```

```
[1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0
 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 0 0 1 1 1 1
 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 1 1]
```

Interception

Oh no! Eve intercepts the message as it passes through her channel. She tries to measure the qubits in a random selection of bases, in the same way Bob will later.

In [26]:
```
## Interception!!
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
print(intercepted_message)
```

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0,
0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0]
```

We can see the case of qubit 0 below; Eve's random choice of basis is not the same as Alice's, and this will change the qubit state from $|+\rangle$, to a random state in the Z-basis, with 50% probability of $|0\rangle$ or $|1\rangle$:

Step 3

Eve then passes on the qubits to Bob, who measures them at random. In this case, Bob chose (by chance) to measure in the same basis Alice prepared the qubit in. Without interception, Bob would be guaranteed to measure 0, but because Eve tried to read the message he now has a 50% chance of measuring 1 instead.

In [27]:
```
## Step 3
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
```

Step 4

In [28]:
```
## Step 4
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
```

Step 5

In [29]:
```
## Step 5
sample_size = 15
bit_selection = randint(n, size=sample_size)
bob_sample = sample_bits(bob_key, bit_selection)
print("  bob_sample = " + str(bob_sample))
alice_sample = sample_bits(alice_key, bit_selection)
print("alice_sample = "+ str(alice_sample))
```

```
  bob_sample = [1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]
alice_sample = [1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

In [30]:
```
bob_sample == alice_sample
```

Out[30]:
```
False
```

Oh no! Bob's key and Alice's key do not match. We know this is because Eve tried to read the message between steps 2 and 3, and changed the qubits' states. For all Alice and Bob know, this could be due to noise in the channel, but either way they must throw away all their results and try again- Eve's interception attempt has failed.

In [ ]:
```

```