In [1]:

```python
# Do the necessary imports
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector, array_to_la
from qiskit.extensions import Initialize
from qiskit.ignis.verification import marginal_counts
from qiskit.quantum_info import random_statevector
```

1. Circuit

In [2]:

```python
## SETUP
# Protocol uses 3 qubits and 2 classical bits in 2 different registers

qr = QuantumRegister(3, name="q")    # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical bits
crx = ClassicalRegister(1, name="crx") # in 2 different registers
teleportation_circuit = QuantumCircuit(qr, crz, crx)
```

Step 1

A third party, Telamon, creates an entangled pair of qubits and gives one to Bob and one to Alice.

The pair Telamon creates is a special pair called a Bell pair. In quantum circuit language, the way to create a Bell pair between two qubits is to first transfer one of them to the X-basis ($|+\rangle$ and $|-\rangle$) using a Hadamard gate, and then to apply a CNOT gate onto the other qubit controlled by the one in the X-basis.

In [3]:

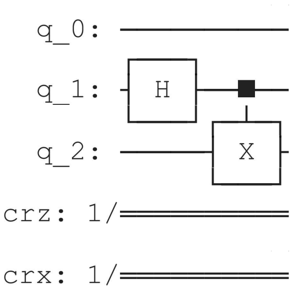
```python
def create_bell_pair(qc, a, b):
    """Creates a bell pair in qc using qubits a & b"""
    qc.h(a) # Put qubit a into state |+>
    qc.cx(a,b) # CNOT with a as control and b as target
```

Step 1

In [4]:

```python
# In our case, Telamon entangles qubits q1 and q2
# Let's apply this to our circuit:
create_bell_pair(teleportation_circuit, 1, 2)
# And view the circuit so far:
teleportation_circuit.draw()
```

Out[4]:



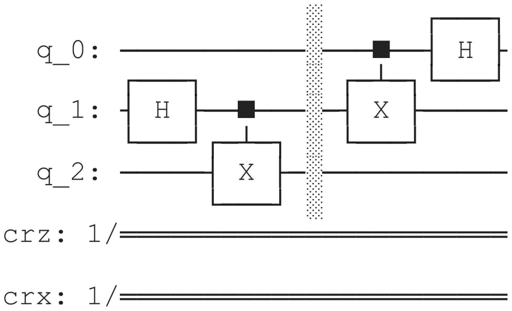
Step 2

Alice applies a CNOT gate to q_1, controlled by $|\psi\rangle$ (the qubit she is trying to send Bob). Then Alice applies a Hadamard gate to $|\psi\rangle$. In our quantum circuit, the qubit ($|\psi\rangle$) Alice is trying to send is q_0:

In [5]:
```python
def alice_gates(qc, psi, a):
    qc.cx(psi, a)
    qc.h(psi)
```

In [6]:
```python
## STEP 2
teleportation_circuit.barrier() # Use barrier to separate steps
alice_gates(teleportation_circuit, 0, 1)
teleportation_circuit.draw()
```

Out[6]:



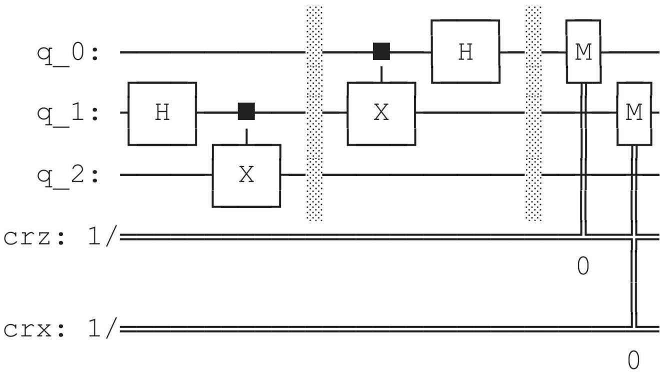
Step 3

Next, Alice applies a measurement to both qubits that she owns, q_1 and $|\psi\rangle$, and stores this result in two classical bits. She then sends these two bits to Bob.

In [7]:
```python
def measure_and_send(qc, a, b):
    """Measures qubits a & b and 'sends' the results to Bob"""
    qc.barrier()
    qc.measure(a,0)
    qc.measure(b,1)
```

In [8]:
```python
## STEP 3
measure_and_send(teleportation_circuit, 0 ,1)
teleportation_circuit.draw()
```

Out[8]:



Step 4

Bob, who already has the qubit q_2, then applies the following gates depending on the state of the classical bits:

00 $\rightarrow$ Do nothing

01 $\rightarrow$ Apply X gate

10 $\rightarrow$ Apply Z gate

11 $\rightarrow$ Apply ZX gate

(*Note that this transfer of information is purely classical.*)

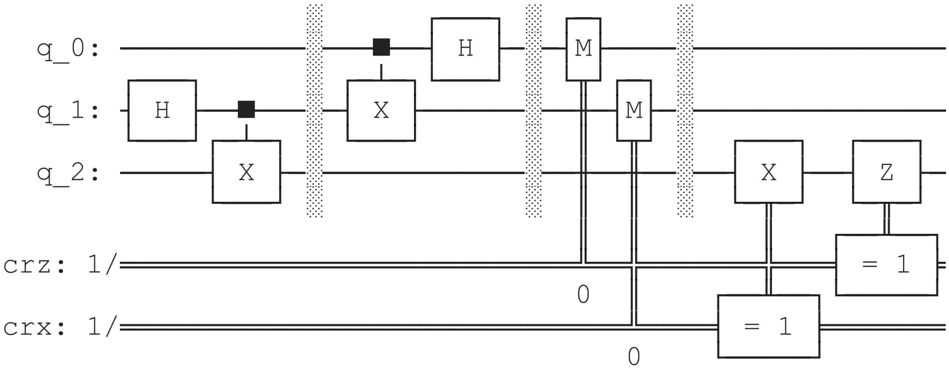
In [9]:

```
# This function takes a QuantumCircuit (qc), integer (qubit)
# and ClassicalRegisters (crz & crx) to decide which gates to apply
def bob_gates(qc, qubit, crz, crx):
    # Here we use c_if to control our gates with a classical
    # bit instead of a qubit
    qc.x(qubit).c_if(crx, 1) # Apply gates if the registers
    qc.z(qubit).c_if(crz, 1) # are in the state '1'
```

In [10]:

```
## STEP 4
teleportation_circuit.barrier() # Use barrier to separate steps
bob_gates(teleportation_circuit, 2, crz, crx)
teleportation_circuit.draw()
```

Out[10]:



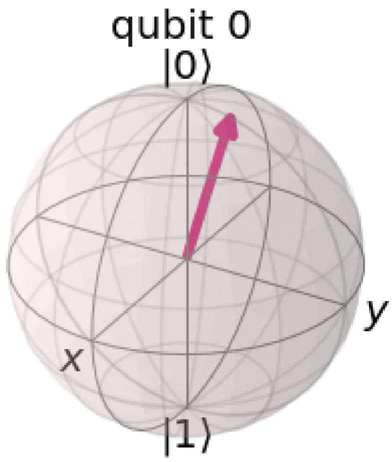
2. Simulating the Ciruit

In [11]:

```
# Create random 1-qubit state
psi = random_statevector(2)

# Display it nicely
display(array_to_latex(psi, prefix="|\\psi\\rangle ="))
# Show it on a Bloch sphere
plot_bloch_multivector(psi)
```

$$|\psi\rangle = \begin{bmatrix} 0.22667 + 0.86743i & -0.04133 - 0.441i \end{bmatrix}$$

Out[11]:



In [12]:

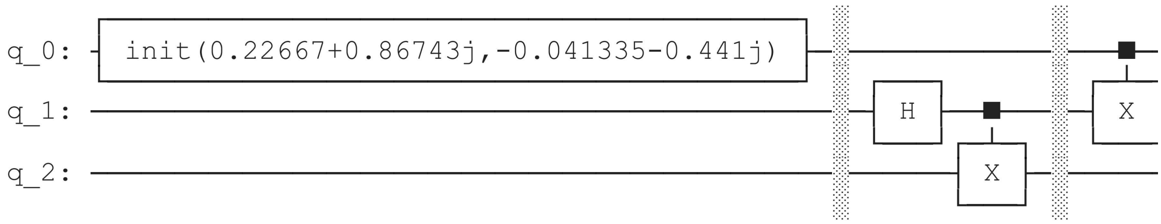
```
init_gate = Initialize(psi)
init_gate.label = "init"
```

(Initialize is technically not a gate since it contains a reset operation, and so is not reversible. We call it an 'instruction' instead). If the quantum teleportation circuit works, then at the end of the circuit the qubit |q2⟩will be in this state. We will check this using the statevector simulator

In [13]:

```
## SETUP
qr = QuantumRegister(3, name="q")    # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's q0
qc.append(init_gate, [0])
qc.barrier()

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)

## STEP 3
# Alice then sends her classical bits to Bob
measure_and_send(qc, 0, 1)

## STEP 4
# Bob decodes qubits
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw()
```

Out[13]:

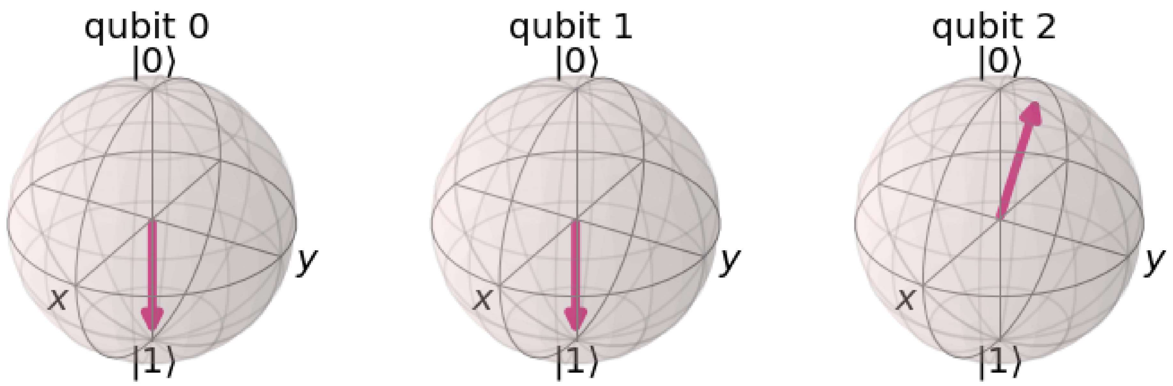


```
crz: 1/════════════════════════════════════════════════════════════════

crx: 1/════════════════════════════════════════════════════════════════

«
«   q_0: ┤M├───────────────────────
«         ╥ ┌─┐
«   q_1: ─╫─┤M├────────────────────
«         ║ └╥┘ ┌───┐    ┌───┐
«   q_2: ─╫──╫──┤ X ├────┤ Z ├─
«         ║  ║  └─┬─┘    └─┬─┘
«         ║  ║    │     ┌──┴──┐
«crz: 1/═╩══╬════╪═════╡ = 1 ╞
«        0   ║    │     └─────┘
«crx: 1/════╩═╡ = 1 ╞══════════
«            0  └─────┘
```

In [14]:

```python
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector)
```

Out[14]:



We can see below, using the statevector obtained from the aer simulator, that the state of $|q_2\rangle$ is the same as the state $|\psi\rangle$ we created above, while the states of $|q_0\rangle$ and $|q_1\rangle$ have been collapsed to either $|0\rangle$ or $|1\rangle$. The state $|\psi\rangle$ has been teleported from qubit 0 to qubit 2.

3. Verifying

In [15]:

```python
inverse_init_gate = init_gate.gates_to_uncompute()
```

In [16]:

```python
## SETUP
qr = QuantumRegister(3, name="q")    # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's q0
qc.append(init_gate, [0])
qc.barrier()

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
```

```python
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)

## STEP 3
# Alice then sends her classical bits to Bob
measure_and_send(qc, 0, 1)

## STEP 4
# Bob decodes qubits
bob_gates(qc, 2, crz, crx)

## STEP 5
# reverse the initialization process
qc.append(inverse_init_gate, [2])

# Display the circuit
qc.draw()
```

Out[16]:

```
      ┌───────────────────────────────────┐ ░           ░    
 q_0: ┤ init(0.22667+0.86743j,-0.041335-0.441j) ├─░───────────░───■──
      └───────────────────────────────────┘ ░ ┌───┐     ░ ┌─┴─┐
 q_1: ──────────────────────────────────────░─┤ H ├──■──░─┤ X ├
                                             ░ └───┘┌─┴─┐ ░ └───┘
 q_2: ──────────────────────────────────────░──────┤ X ├─░──────
                                             ░      └───┘ ░      
crz: 1/═════════════════════════════════════════════════════════
                                                                 
crx: 1/═════════════════════════════════════════════════════════
                                                                 
«
«   q_0: ┤M├────────────────────────────────────
«         ╥ ┌─┐
«   q_1: ─╫─┤M├─────────────────────────────────
«         ║ └╥┘ ┌───┐   ┌───┐  ┌──────────────┐
«   q_2: ─╫──╫──┤ X ├───┤ Z ├──┤ disentangler ├─
«         ║  ║  └─╥─┘   └─╥─┘  └──────────────┘
«         ║  ║    ║    ┌──╨──┐
«crz: 1/══╩══╬════╬════╡ = 1 ╞══════════════════
«         0  ║ ┌──╨──┐ └─────┘
«crx: 1/═════╩═╡ = 1 ╞══════════════════════════
«            0 └─────┘
```

In [17]:

```python
# Need to add a new ClassicalRegister
# to see the result
cr_result = ClassicalRegister(1)
qc.add_register(cr_result)
qc.measure(2,2)
qc.draw()
```

Out[17]:

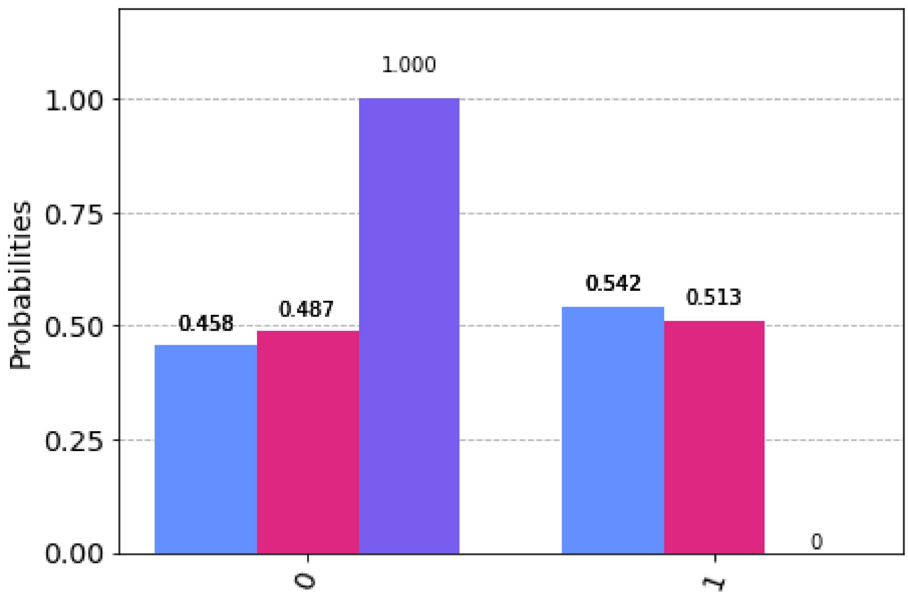
```
      ┌───────────────────────────────────┐ ░           ░    
 q_0: ┤ init(0.22667+0.86743j,-0.041335-0.441j) ├─░───────────░───■──
      └───────────────────────────────────┘ ░ ┌───┐     ░ ┌─┴─┐
 q_1: ──────────────────────────────────────░─┤ H ├──■──░─┤ X ├
                                             ░ └───┘┌─┴─┐ ░ └───┘
 q_2: ──────────────────────────────────────░──────┤ X ├─░──────
                                             ░      └───┘ ░      
crz: 1/═════════════════════════════════════════════════════════
                                                                 
crx: 1/═════════════════════════════════════════════════════════
```

```
   c0: 1/═══════════════════════════════════════════════════════════
                                                                    
«                                                                   
«   q_0: ┤M├─────────────────────────────────────────────────
«         ║┌─┐                                                
«   q_1: ─╫┤M├───────────────────────────────────────────────
«         ║└╥┘ ┌───┐   ┌───┐ ┌──────────────┐┌─┐           
«   q_2: ─╫─╫──┤ X ├───┤ Z ├─┤ disentangler ├┤M├
«         ║ ║  └─╥─┘   └─╥─┘ └──────────────┘└╥┘
«         ║ ║    ║    ┌──╨──┐                  ║ 
«crz: 1/══╩═╬════╬════╡ = 1 ╞══════════════════╬═
«         0 ║    ║    └─────┘                  ║ 
«           ║ ┌──╨──┐                          ║ 
«crx: 1/════╩═╡ = 1 ╞══════════════════════════╬═
«           0 └─────┘                          ║ 
« c0: 1/═══════════════════════════════════════╩═
«                                              0 
```

In [18]:

```python
t_qc = transpile(qc, sim)
t_qc.save_statevector()
counts = sim.run(t_qc).result().get_counts()
qubit_counts = [marginal_counts(counts, [qubit]) for qubit in range(3)]
plot_histogram(qubit_counts)
```

Out[18]:



We can see we have a 100% chance of measuring q2 (the purple bar in the histogram) in the state |0⟩. This is the expected result, and indicates the teleportation protocol has worked properly.

In [ ]: