1. Initialization

In [1]:

```
# Importing everything
from qiskit import QuantumCircuit
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram
```

We saw that to create an entangled pair, we needed to do a H-gate followed by a CNOT. Let's create a function that takes a QuantumCircuit and entangles the qubits with indices a and b.

2. Circuit

In [2]:

```
def create_bell_pair():
    """
    Returns:
        QuantumCircuit: Circuit that produces a Bell pair
    """
    qc = QuantumCircuit(2)
    qc.h(1)
    qc.cx(1, 0)
    return qc
```

Next we need to encode our message. We saw that there were four possible messages we could send: 00, 10, 01 or 11. Let's create a function that takes this message and applies the appropriate gates for us:

Encoding Rules for Superdense Coding (Alice protocol):

Intended Message	Applied Gate	Resulting State ($\cdot\frac{1}{\sqrt{2}}$)
00	I	$\|00\rangle + \|11\rangle$
01	X	$\|10\rangle + \|01\rangle$
10	Z	$\|00\rangle - \|11\rangle$
11	ZX	$-\|10\rangle + \|01\rangle$

In [5]:

```
def encode_message(qc, qubit, msg):
    """Encodes a two-bit message on qc using the superdense coding protocol
    Args:
        qc (QuantumCircuit): Circuit to encode message on
        qubit (int): Which qubit to add the gate to
        msg (str): Two-bit message to send
    Returns:
        QuantumCircuit: Circuit that, when decoded, will produce msg
    Raises:
        ValueError if msg is wrong length or contains invalid characters
    """
    if len(msg) != 2 or not set(msg).issubset({"0","1"}):
        raise ValueError(f"message '{msg}' is invalid")
    if msg[1] == "1":
        qc.x(qubit)
    if msg[0] == "1":
```

```
        qc.z(qubit)
    return qc
```

Bob Receives ($\cdot \frac{1}{\sqrt{2}}$)	After CNOT-gate ($\cdot \frac{1}{\sqrt{2}}$)	After H-gate
$\|00\rangle + \|11\rangle$	$\|00\rangle + \|10\rangle$	$\|00\rangle$
$\|10\rangle + \|01\rangle$	$\|11\rangle + \|01\rangle$	$\|01\rangle$
$\|00\rangle - \|11\rangle$	$\|00\rangle - \|10\rangle$	$\|10\rangle$
$-\|10\rangle + \|01\rangle$	$-\|11\rangle + \|01\rangle$	$\|11\rangle$

In [6]:

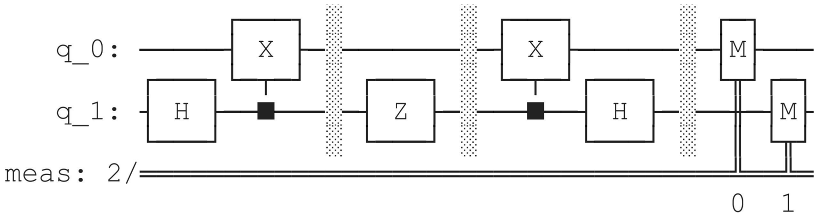
```
def decode_message(qc):
    qc.cx(1, 0)
    qc.h(1)
    return qc
```

Finally, we can put this together to complete our protocol.

In [7]:

```
# Charlie creates the entangled pair between Alice and Bob
qc = create_bell_pair()

# We'll add a barrier for visual separation
qc.barrier()

# At this point, qubit 0 goes to Alice and qubit 1 goes to Bob

# Next, Alice encodes her message onto qubit 1. In this case,
# we want to send the message '10'. You can try changing this
# value and see how it affects the circuit
message = '10'
qc = encode_message(qc, 1, message)
qc.barrier()
# Alice then sends her qubit to Bob.

# After recieving qubit 0, Bob applies the recovery protocol:
qc = decode_message(qc)

# Finally, Bob measures his qubits to read Alice's message
qc.measure_all()

# Draw our output
qc.draw()
```

Out[7]:



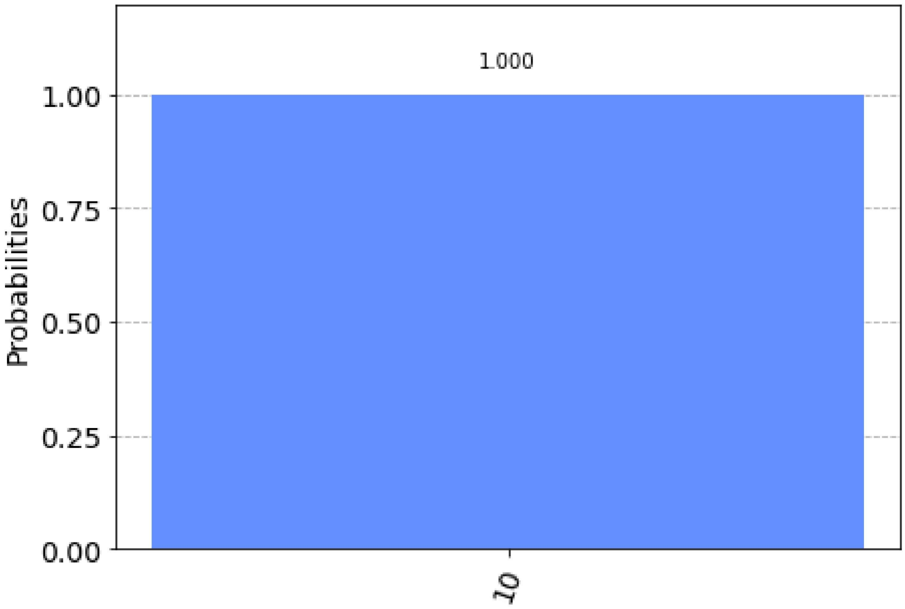
3. Results(Simulator)

In [8]:

```
aer_sim = Aer.get_backend('aer_simulator')
qobj = assemble(qc)
result = aer_sim.run(qobj).result()
counts = result.get_counts(qc)
print(counts)
plot_histogram(counts)
```

```
{'10': 1024}
```

Out[8]:



Our simulator simulates a perfect quantum computer. We can see that, without errors, we get a 100% chance of measuring the correct message.

4. Results(Real Device)

In [9]:

```
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy
shots = 1024

# Load local account information
IBMQ.load_account()
# Get the least busy backend
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
print("least busy backend: ", backend)
# Run our circuit
t_qc = transpile(qc, backend, optimization_level=3)
job = backend.run(t_qc)
```

```
least busy backend:  ibmq_lima
```

In [10]:

```
# Monitoring our job
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

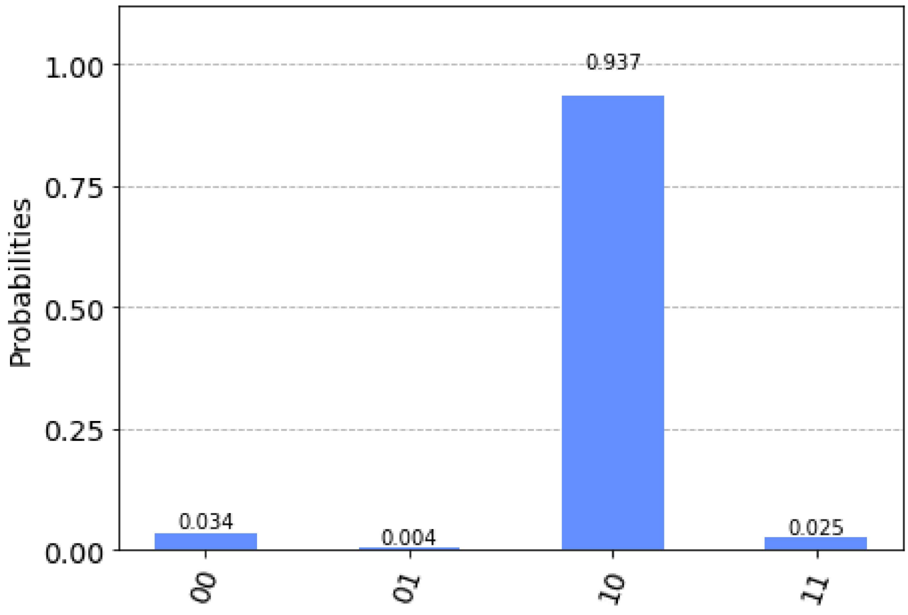
```
Job Status: job has successfully run
```

In [11]:

```
# Plotting our result
```

```python
result = job.result()
plot_histogram(result.get_counts(qc))
```

Out[11]:



As we see that there are a few results from the other three states when run in a real quantum computer. These are due to errors in the gates and qubit decoherence.

In [12]:

```python
correct_results = result.get_counts(qc)[message]
accuracy = (correct_results/shots)*100
print(f"Accuracy = {accuracy:.2f}%")
```

```
Accuracy = 93.65%
```

In [ ]: