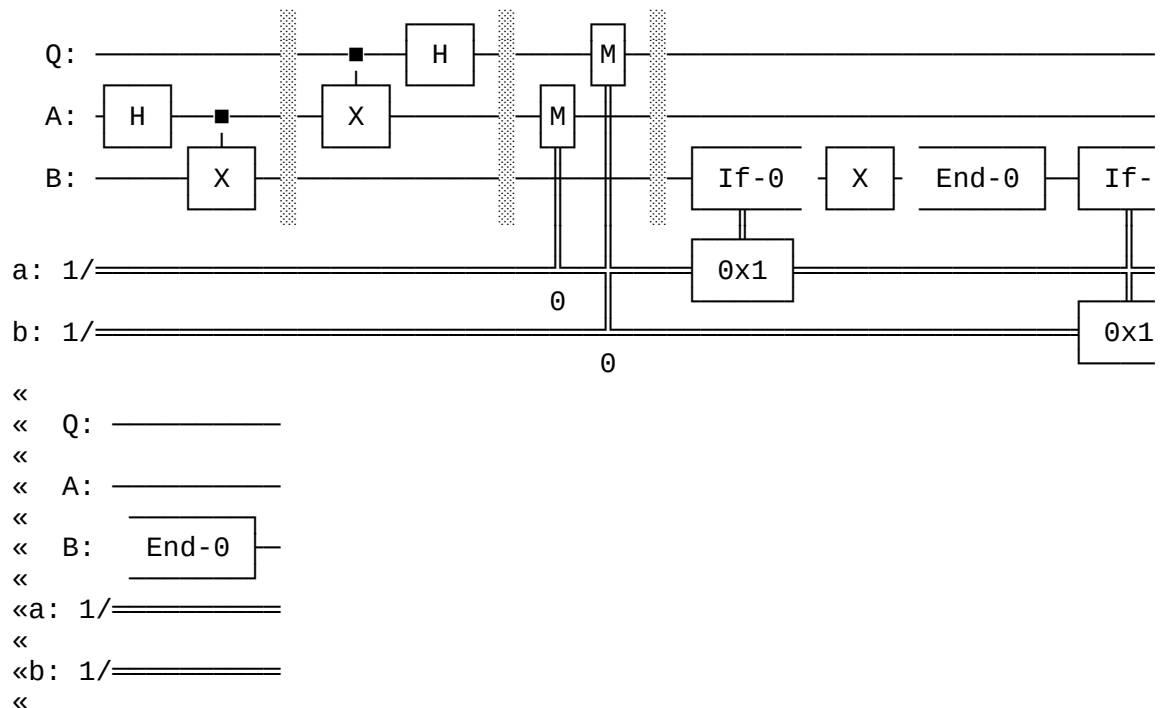


```
In [2]: 1 # Required imports
        2
        3 from qiskit import QuantumCircuit, QuantumRegister, ClassicalReg:
        4 from qiskit_aer import AerSimulator
        5 from qiskit.visualization import plot_histogram
        6 from qiskit.result import marginal_distribution
        7 from qiskit.circuit.library import UGate
        8 from numpy import pi, random
```

```

In [3]: 1 qubit = QuantumRegister(1, "Q")
        2 ebit0 = QuantumRegister(1, "A")
        3 ebit1 = QuantumRegister(1, "B")
        4 a = ClassicalRegister(1, "a")
        5 b = ClassicalRegister(1, "b")
        6
        7 protocol = QuantumCircuit(qubit, ebit0, ebit1, a, b)
        8
        9 # Prepare ebit used for teleportation
       10 protocol.h(ebit0)
       11 protocol.cx(ebit0, ebit1)
       12 protocol.barrier()
       13
       14 # Alice's operations
       15 protocol.cx(qubit, ebit0)
       16 protocol.h(qubit)
       17 protocol.barrier()
       18
       19 # Alice measures and sends classical bits to Bob
       20 protocol.measure(ebit0, a)
       21 protocol.measure(qubit, b)
       22 protocol.barrier()
       23
       24 # Bob uses the classical bits to conditionally apply gates
       25 with protocol.if_test((a, 1)):
       26     protocol.x(ebit1)
       27 with protocol.if_test((b, 1)):
       28     protocol.z(ebit1)
       29
       30 display(protocol.draw())

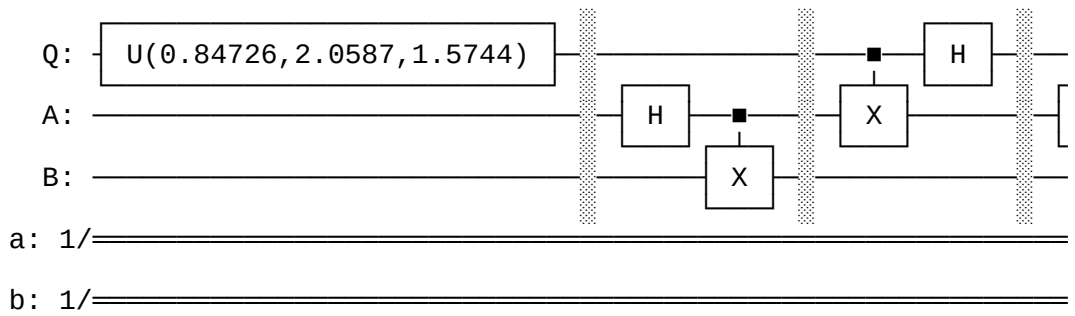
```



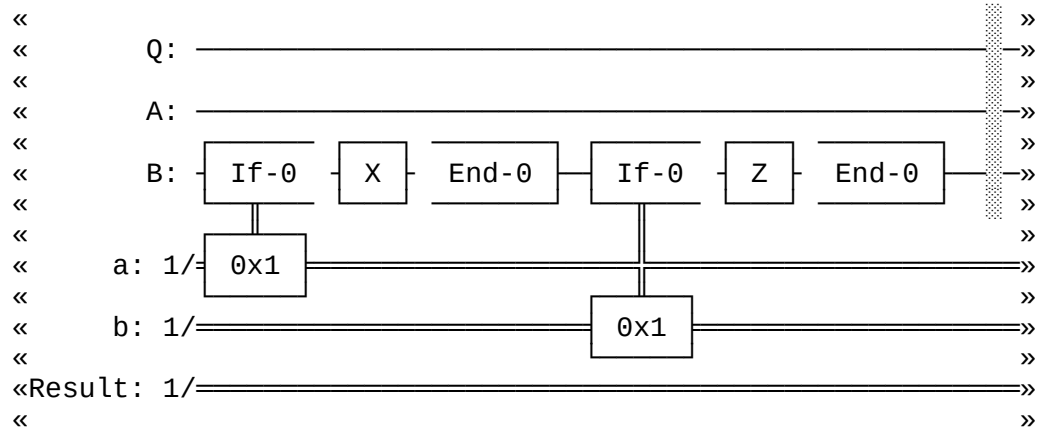
```
In [4]: 1 random_gate = UGate(
        2     theta=random.random() * 2 * pi,
        3     phi=random.random() * 2 * pi,
        4     lam=random.random() * 2 * pi,
        5 )
        6
        7 display(random_gate.to_matrix())
```

```
array([[ 0.91160233+0.j          ,  0.00149082-0.41107051j],
       [-0.19269079+0.36311355j, -0.80369121-0.43023162j]])
```

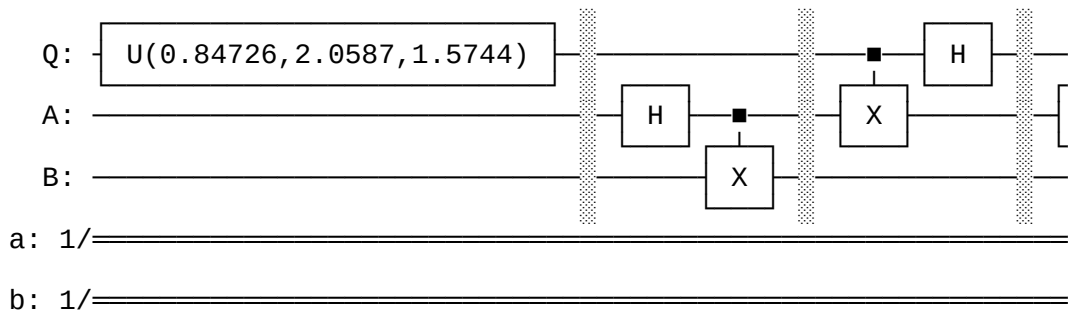
```
In [5]: 1 # Create a new circuit including the same bits and qubits used in
2 # teleportation protocol.
3
4 test = QuantumCircuit(qubit, ebit0, ebit1, a, b)
5
6 # Start with the randomly selected gate on Q
7
8 test.append(random_gate, qubit)
9 test.barrier()
10
11 # Append the entire teleportation protocol from above.
12
13 test = test.compose(protocol)
14 test.barrier()
15
16 # Finally, apply the inverse of the random unitary to B and measu
17
18 test.append(random_gate.inverse(), ebit1)
19
20 result = ClassicalRegister(1, "Result")
21 test.add_register(result)
22 test.measure(ebit1, result)
23
24 display(test.draw())
```



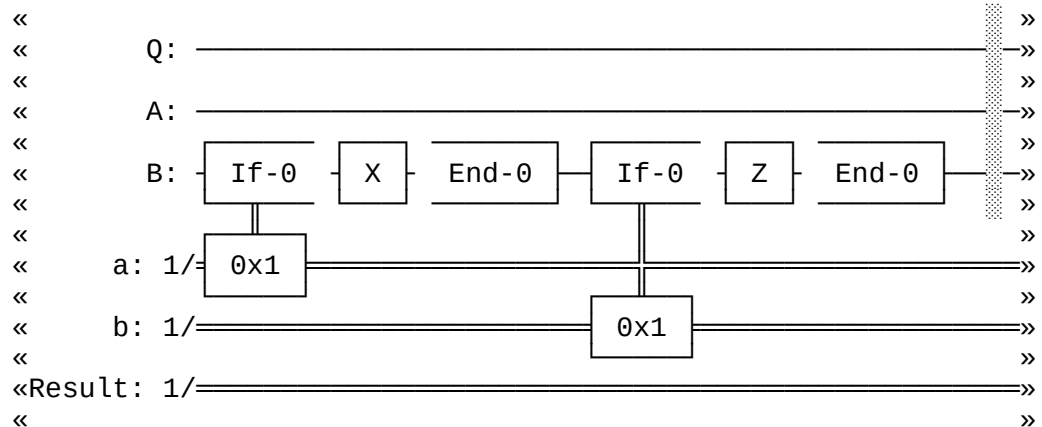
Result: 1/



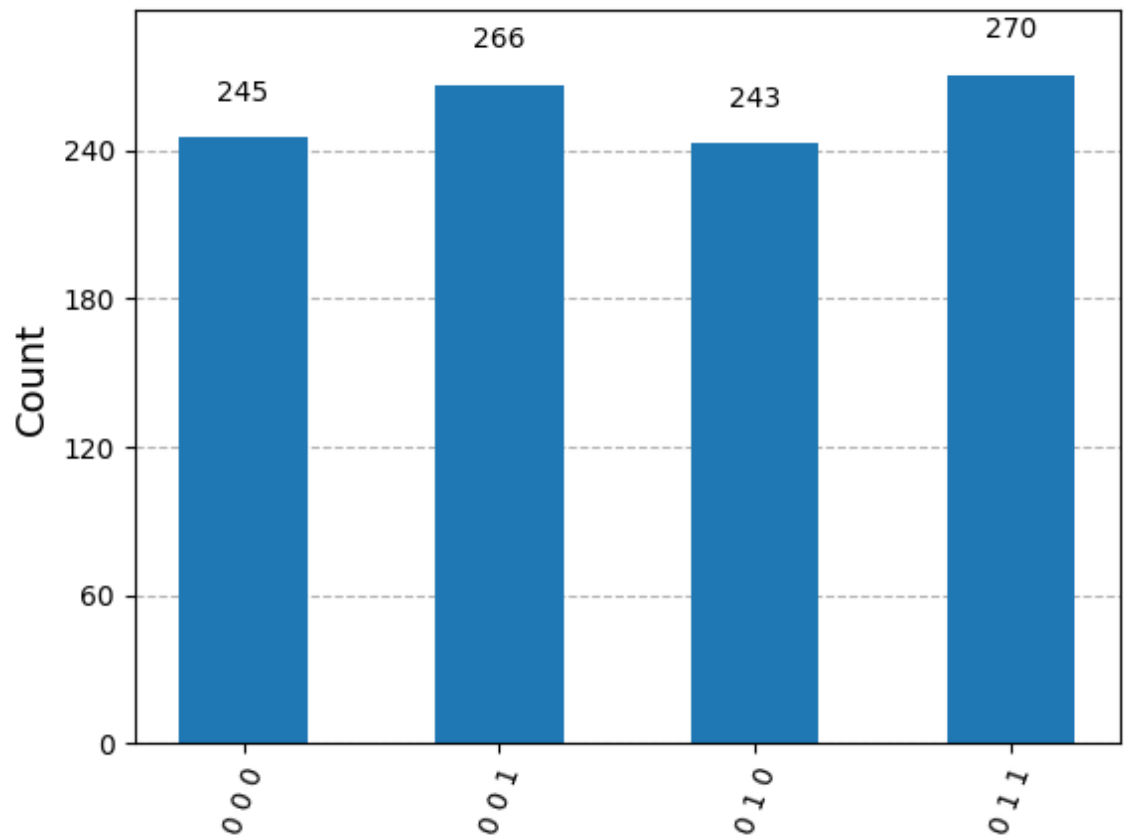
```
In [6]: 1 # Create a new circuit including the same bits and qubits used in
2 # teleportation protocol.
3
4 test = QuantumCircuit(qubit, ebit0, ebit1, a, b)
5
6 # Start with the randomly selected gate on Q
7
8 test.append(random_gate, qubit)
9 test.barrier()
10
11 # Append the entire teleportation protocol from above.
12
13 test = test.compose(protocol)
14 test.barrier()
15
16 # Finally, apply the inverse of the random unitary to B and measu
17
18 test.append(random_gate.inverse(), ebit1)
19
20 result = ClassicalRegister(1, "Result")
21 test.add_register(result)
22 test.measure(ebit1, result)
23
24 display(test.draw())
```



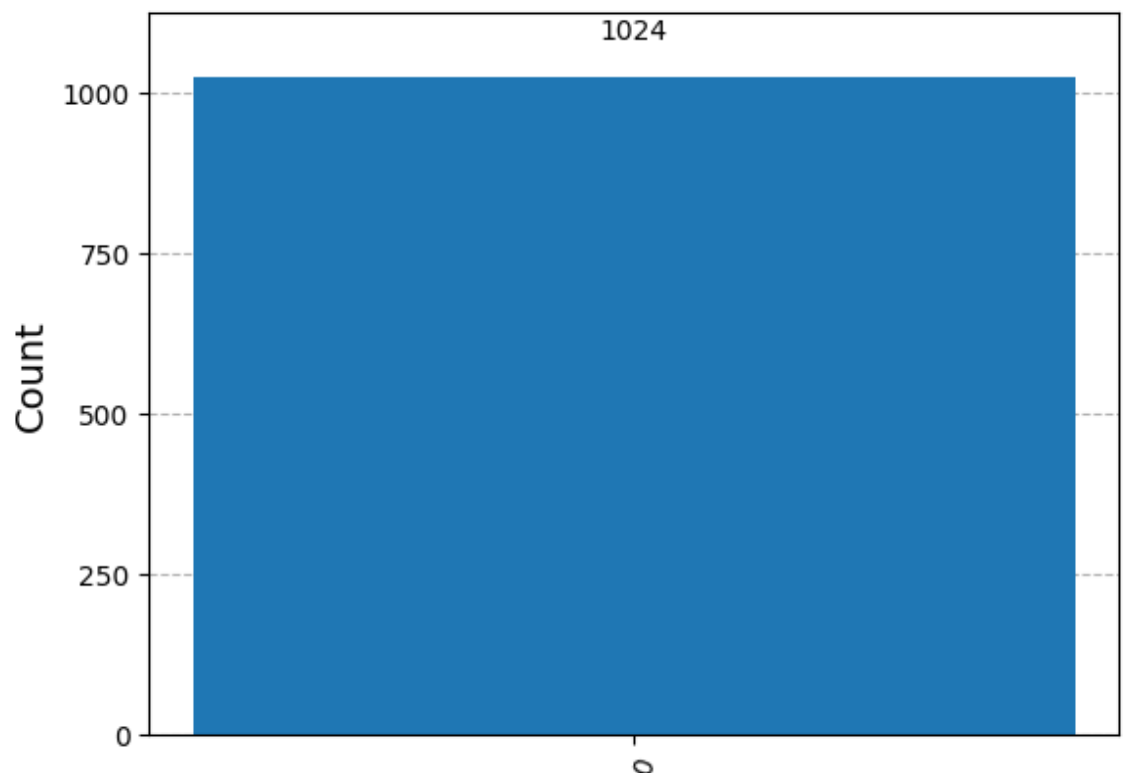
Result: 1/



```
In [7]: 1 result = AerSimulator().run(test).result()
        2 statistics = result.get_counts()
        3 display(plot_histogram(statistics))
```



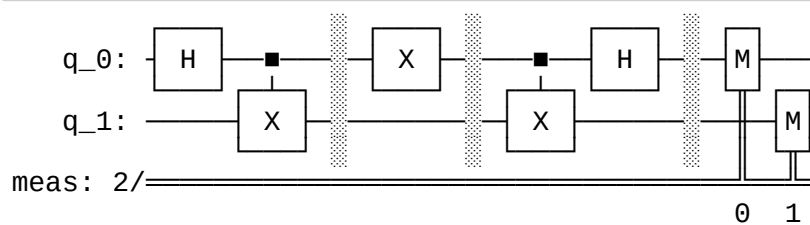
```
In [8]: 1 filtered_statistics = marginal_distribution(statistics, [2])
        2 display(plot_histogram(filtered_statistics))
```




```
In [9]: 1 # Required imports
        2
        3 from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegi
        4 from qiskit_aer.primitives import Sampler
        5 from qiskit_aer import AerSimulator
        6 from qiskit.visualization import plot_histogram
```

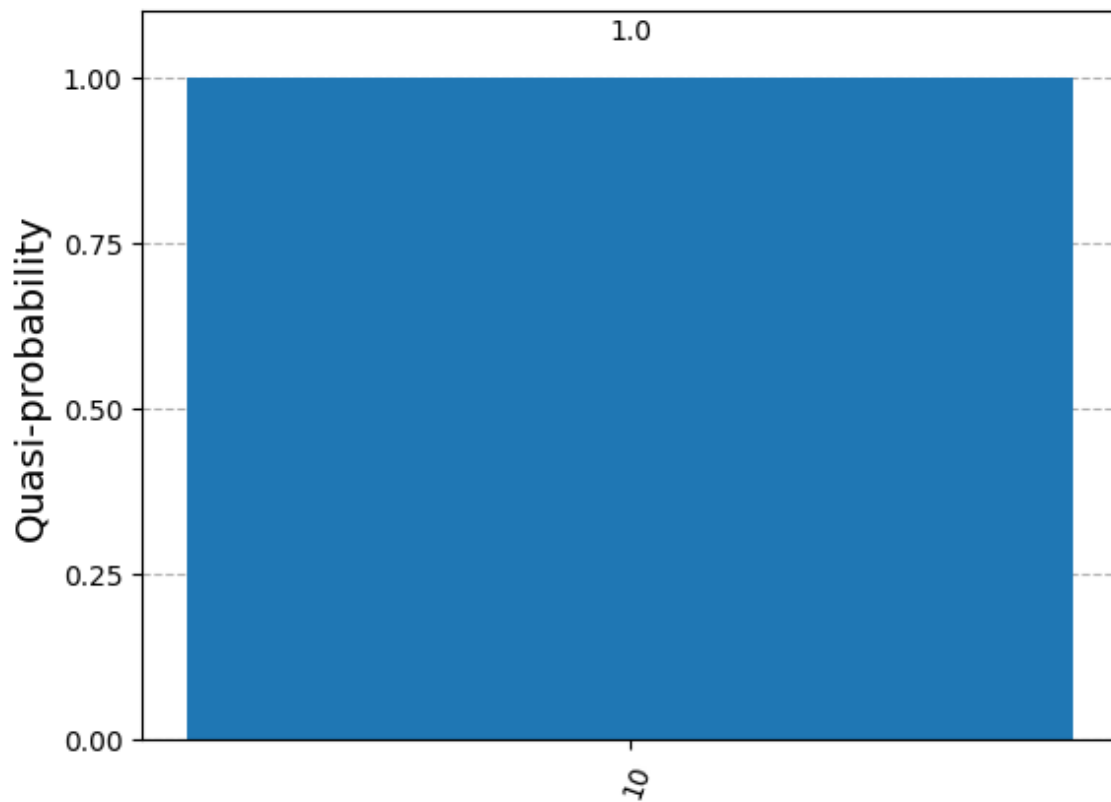
```
In [10]: 1 c = "1"
        2 d = "0"
```

```
In [11]: 1 protocol = QuantumCircuit(2)
        2
        3 # Prepare ebit used for superdense coding
        4 protocol.h(0)
        5 protocol.cx(0, 1)
        6 protocol.barrier()
        7
        8 # Alice's operations
        9 if d == "1":
        10     protocol.z(0)
        11 if c == "1":
        12     protocol.x(0)
        13 protocol.barrier()
        14
        15 # Bob's actions
        16 protocol.cx(0, 1)
        17 protocol.h(0)
        18 protocol.measure_all()
        19
        20 display(protocol.draw())
```



```
In [12]: 1 result = Sampler().run(protocol).result()
2 statistics = result.quasi_dists[0].binary_probabilities()
3
4 for outcome, frequency in statistics.items():
5     print(f"Measured {outcome} with frequency {frequency}")
6
7 display(plot_histogram(statistics))
```

Measured 10 with frequency 1.0



```
In [13]: 1 rbg = QuantumRegister(1, "randomizer")
2 ebit0 = QuantumRegister(1, "A")
3 ebit1 = QuantumRegister(1, "B")
4
5 Alice_c = ClassicalRegister(1, "Alice c")
6 Alice_d = ClassicalRegister(1, "Alice d")
7
8 test = QuantumCircuit(rbg, ebit0, ebit1, Alice_d, Alice_c)
9
10 # Initialize the ebit
11 test.h(ebit0)
12 test.cx(ebit0, ebit1)
13 test.barrier()
14
15 # Use the 'randomizer' qubit twice to generate Alice's bits c and d
16 test.h(rbg)
17 test.measure(rbg, Alice_c)
18 test.h(rbg)
19 test.measure(rbg, Alice_d)
20 test.barrier()
21
22 # Now the protocol runs, starting with Alice's actions, which depend
23 # on her bits.
24 with test.if_test((Alice_d, 1), label="Z"):
25     test.z(ebit0)
26 with test.if_test((Alice_c, 1), label="X"):
27     test.x(ebit0)
28 test.barrier()
29
30 # Bob's actions
31 test.cx(ebit0, ebit1)
32 test.h(ebit0)
33 test.barrier()
34
35 Bob_c = ClassicalRegister(1, "Bob c")
36 Bob_d = ClassicalRegister(1, "Bob d")
37 test.add_register(Bob_d)
38 test.add_register(Bob_c)
39 test.measure(ebit0, Bob_d)
40 test.measure(ebit1, Bob_c)
41
42 display(test.draw())
```



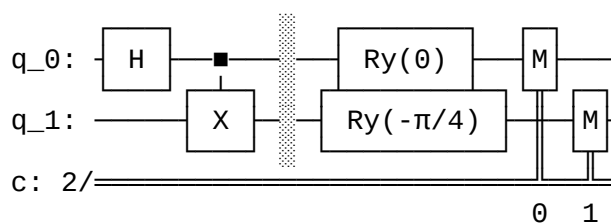
```
In [15]: 1 # Required imports
          2
          3 from qiskit import QuantumCircuit
          4 from qiskit_aer.primitives import Sampler
          5 from numpy import pi
          6 from numpy.random import randint
```

```
In [16]: 1 def chsh_game(strategy):
          2     """Plays the CHSH game
          3     Args:
          4         strategy (callable): A function that takes two bits (as
          5         returns two bits (also as `int`s). The strategy must
          6         rules of the CHSH game.
          7     Returns:
          8         int: 1 for a win, 0 for a loss.
          9     """
         10     # Referee chooses x and y randomly
         11     x, y = randint(0, 2), randint(0, 2)
         12
         13     # Use strategy to choose a and b
         14     a, b = strategy(x, y)
         15
         16     # Referee decides if Alice and Bob win or lose
         17     if (a != b) == (x & y):
         18         return 1 # Win
         19     return 0 # Lose
```

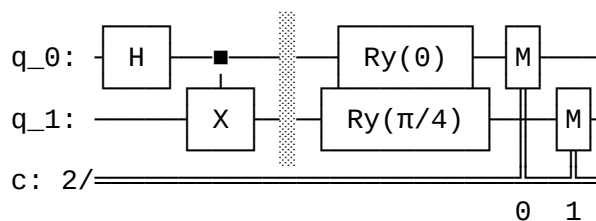
```
In [17]: 1 def chsh_circuit(x, y):
2         """Creates a `QuantumCircuit` that implements the best CHSH s
3         Args:
4             x (int): Alice's bit (must be 0 or 1)
5             y (int): Bob's bit (must be 0 or 1)
6         Returns:
7             QuantumCircuit: Circuit that, when run, returns Alice and
8                 answer bits.
9         """
10        qc = QuantumCircuit(2, 2)
11        qc.h(0)
12        qc.cx(0, 1)
13        qc.barrier()
14
15        # Alice
16        if x == 0:
17            qc.ry(0, 0)
18        else:
19            qc.ry(-pi / 2, 0)
20        qc.measure(0, 0)
21
22        # Bob
23        if y == 0:
24            qc.ry(-pi / 4, 1)
25        else:
26            qc.ry(pi / 4, 1)
27        qc.measure(1, 1)
28
29        return qc
```

```
In [18]: 1 # Draw the four possible circuits
2
3 print("(x,y) = (0,0)")
4 display(chsh_circuit(0, 0).draw())
5
6 print("(x,y) = (0,1)")
7 display(chsh_circuit(0, 1).draw())
8
9 print("(x,y) = (1,0)")
10 display(chsh_circuit(1, 0).draw())
11
12 print("(x,y) = (1,1)")
13 display(chsh_circuit(1, 1).draw())
```

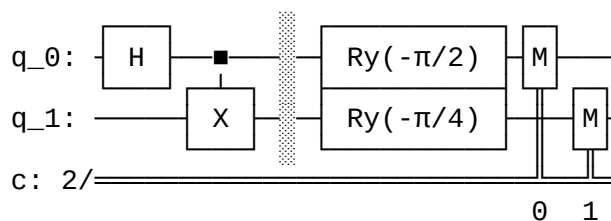
(x,y) = (0,0)



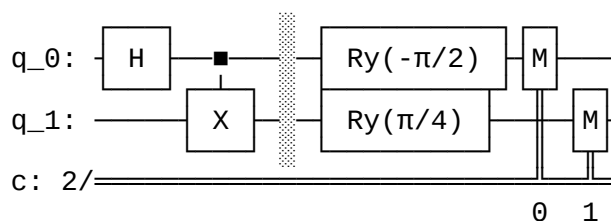
(x,y) = (0,1)



(x,y) = (1,0)



(x,y) = (1,1)



```
In [19]: 1 sampler = Sampler()
2
3
4 def quantum_strategy(x, y):
5     """Carry out the best strategy for the CHSH game.
6     Args:
7         x (int): Alice's bit (must be 0 or 1)
8         y (int): Bob's bit (must be 0 or 1)
9     Returns:
10        (int, int): Alice and Bob's answer bits (respectively)
11        """
12        # `shots=1` runs the circuit once
13        result = sampler.run(chsh_circuit(x, y), shots=1).result()
14        statistics = result.quasi_dists[0].binary_probabilities()
15        bits = list(statistics.keys())[0]
16        a, b = bits[0], bits[1]
17        return a, b
```

```
In [20]: 1 NUM_GAMES = 1000
2 TOTAL_SCORE = 0
3
4 for _ in range(NUM_GAMES):
5     TOTAL_SCORE += chsh_game(quantum_strategy)
6
7 print("Fraction of games won:", TOTAL_SCORE / NUM_GAMES)
```

Fraction of games won: 0.857

```
In [21]: 1 def classical_strategy(x, y):
2     """An optimal classical strategy for the CHSH game
3     Args:
4         x (int): Alice's bit (must be 0 or 1)
5         y (int): Bob's bit (must be 0 or 1)
6     Returns:
7         (int, int): Alice and Bob's answer bits (respectively)
8         """
9     # Alice's answer
10    if x == 0:
11        a = 0
12    elif x == 1:
13        a = 1
14
15    # Bob's answer
16    if y == 0:
17        b = 1
18    elif y == 1:
19        b = 0
20
21    return a, b
```


In [22]:

```
1 NUM_GAMES = 1000
2 TOTAL_SCORE = 0
3
4 for _ in range(NUM_GAMES):
5     TOTAL_SCORE += chsh_game(classical_strategy)
6
7 print("Fraction of games won:", TOTAL_SCORE / NUM_GAMES)
```

Fraction of games won: 0.743

In []:

1