In [1]:

```python
from qiskit import QuantumCircuit

def deutsch_function(case: int):
    """
    Generate a valid Deutsch function as a `QuantumCircuit`.
    """
    if case not in [1, 2, 3, 4]:
        raise ValueError("`case` must be 1, 2, 3, or 4.")

    f = QuantumCircuit(2)
    if case in [2, 3]:
        f.cx(0, 1)
    if case in [3, 4]:
        f.x(1)
    return f
```

In [2]:

```python
display(deutsch_function(3).draw())
```

```
                
q_0: ──■───────
     ┌─┴─┐┌───┐
q_1: ┤ X ├┤ X ├
     └───┘└───┘
```

In [3]:

```python
def compile_circuit(function: QuantumCircuit):
    """
    Compiles a circuit for use in Deutsch's algorithm.
    """
    n = function.num_qubits - 1
    qc = QuantumCircuit(n + 1, n)

    qc.x(n)
    qc.h(range(n + 1))

    qc.barrier()
    qc.compose(function, inplace=True)
    qc.barrier()

    qc.h(range(n))
    qc.measure(range(n), range(n))

    return qc
```

In [4]:

```python
display(compile_circuit(deutsch_function(3)).draw())
```

```
     ┌───┐      ░            ░ ┌───┐┌─┐
q_0: ┤ H ├──────░───■────────░─┤ H ├┤M├
     ├───┤┌───┐ ░ ┌─┴─┐┌───┐ ░ └───┘└╥┘
q_1: ┤ X ├┤ H ├─░─┤ X ├┤ X ├─░───────╫─
     └───┘└───┘ ░ └───┘└───┘ ░       ║ 
c: 1/════════════════════════════════╩═
                                     0 
```

In [5]:

```python
from qiskit_aer import AerSimulator

def deutsch_algorithm(function: QuantumCircuit):
    """
    Determine if a Deutsch function is constant or balanced.
    """
    qc = compile_circuit(function)

    result = AerSimulator().run(qc, shots=1, memory=True).result
    measurements = result.get_memory()
    if measurements[0] == "0":
        return "constant"
    return "balanced"
```

In [6]:

```python
f = deutsch_function(3)
display(f.draw())
display(deutsch_algorithm(f))
```

```
q_0: ──■──────────
     ┌─┴─┐┌───┐
q_1: ┤ X ├┤ X ├
     └───┘└───┘
```

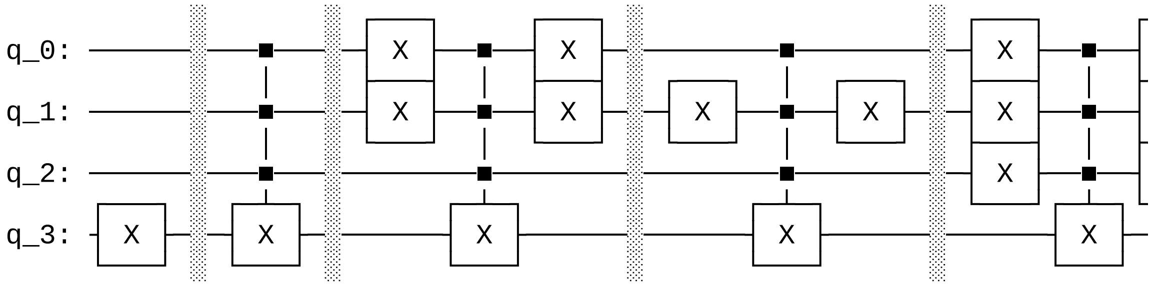
```
'balanced'
```

In [7]:

```
from qiskit import QuantumCircuit
import numpy as np

def dj_function(num_qubits):
    """
    Create a random Deutsch-Jozsa function.
    """

    qc = QuantumCircuit(num_qubits + 1)
    if np.random.randint(0, 2):
        # Flip output qubit with 50% chance
        qc.x(num_qubits)
    if np.random.randint(0, 2):
        # return constant circuit with 50% chance
        return qc

    # next, choose half the possible input states
    on_states = np.random.choice(
        range(2**num_qubits),  # numbers to sample from
        2**num_qubits // 2,  # number of samples
        replace=False,  # makes sure states are only sampled once
    )

    def add_cx(qc, bit_string):
        for qubit, bit in enumerate(reversed(bit_string)):
            if bit == "1":
                qc.x(qubit)
        return qc

    for state in on_states:
        qc.barrier()  # Barriers are added to help visualize how
        qc = add_cx(qc, f"{state:0b}")
        qc.mcx(list(range(num_qubits)), num_qubits)
        qc = add_cx(qc, f"{state:0b}")

    qc.barrier()

    return qc
```

In [9]:

```
display(dj_function(3).draw())
```



In [10]:

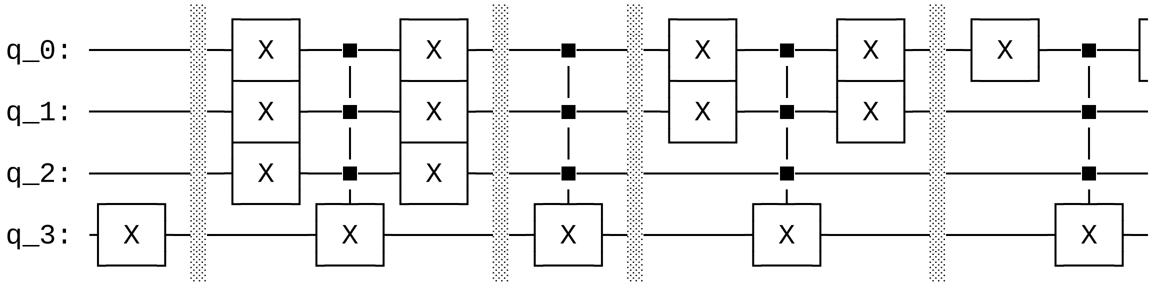
```python
def compile_circuit(function: QuantumCircuit):
    """
    Compiles a circuit for use in the Deutsch-Jozsa algorithm.
    """
    n = function.num_qubits - 1
    qc = QuantumCircuit(n + 1, n)
    qc.x(n)
    qc.h(range(n + 1))
    qc.compose(function, inplace=True)
    qc.h(range(n))
    qc.measure(range(n), range(n))
    return qc
```

In [11]:

```python
from qiskit_aer import AerSimulator

def dj_algorithm(function: QuantumCircuit):
    """
    Determine if a Deutsch-Jozsa function is constant or balance
    """
    qc = compile_circuit(function)

    result = AerSimulator().run(qc, shots=1, memory=True).result
    measurements = result.get_memory()
    if "1" in measurements[0]:
        return "balanced"
    return "constant"
```

In [13]:

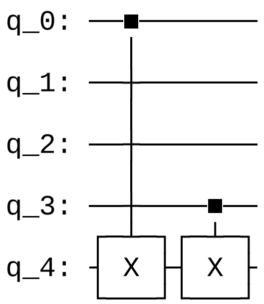
```python
f = dj_function(3)
display(f.draw())
display(dj_algorithm(f))
```



```
'balanced'
```

In [14]:

```python
def bv_function(s):
    """
    Create a Bernstein-Vazirani function from a string of 1s and
    """
    qc = QuantumCircuit(len(s) + 1)
    for index, bit in enumerate(reversed(s)):
        if bit == "1":
            qc.cx(index, len(s))
    return qc

display(bv_function("1001").draw())
```



In [15]:

```python
def bv_algorithm(function: QuantumCircuit):
    qc = compile_circuit(function)
    result = AerSimulator().run(qc, shots=1, memory=True).result
    return result.get_memory()[0]

display(bv_algorithm(bv_function("1001")))
```

```
'1001'
```

In [16]:

```python
# import random
import qiskit.quantum_info as qi
from qiskit import QuantumCircuit
import numpy as np

def simon_function(s: str):
    """
    Create a QuantumCircuit implementing a query gate for Simon p
    """
    # Our quantum circuit has 2n qubits for n = len(s)
    n = len(s)
    qc = QuantumCircuit(2 * n)

    # Define a random permutation of all n bit strings. This perm
    pi = np.random.permutation(2**n)

    # Now we'll define a query gate explicitly. The idea is to fi
    # is a simple function that satisfies the promise, and then w
    # permutation pi. This gives us a random function satisfying

    query_gate = np.zeros((4**n, 4**n))
    for x in range(2**n):
        for y in range(2**n):
            z = y ^ pi[min(x, x ^ int(s, 2))]
            query_gate[x + 2**n * z, x + 2**n * y] = 1

    # Our circuit has just this one query gate
    qc.unitary(query_gate, range(2 * n))
    return qc
```

In [17]:

```python
from qiskit_aer import AerSimulator
from qiskit import ClassicalRegister

def simon_measurements(problem: QuantumCircuit, k: int):
    """
    Quantum part of Simon's algorithm. Given a `QuantumCircuit` t
    implements f, get `k` measurements to be post-processed later
    """
    n = problem.num_qubits // 2

    qc = QuantumCircuit(2 * n, n)
    qc.h(range(n))
    qc.compose(problem, inplace=True)
    qc.h(range(n))
    qc.measure(range(n), range(n))

    result = AerSimulator().run(qc, shots=k, memory=True).result(
    return result.get_memory()
```

In [18]:

```
display(simon_measurements(simon_function("11011"),k=12))
```

```
['11111',
 '01101',
 '00100',
 '00000',
 '00000',
 '00100',
 '10001',
 '11111',
 '00011',
 '10010',
 '10001',
 '11000']
```

In [20]:

```
import numpy as np
import galois

def simon_algorithm(problem: QuantumCircuit):
    """
    Given a `QuantumCircuit` that implements a query gate for Si
    """

    # Quantum part: run the circuit defined previously k times a
    # Replace +10 by +r for any nonnegative integer r depending

    measurements = simon_measurements(problem, k=problem.num_qub
    print("Measurement results:")
    display(measurements)

    # Classical post-processing:

    # 1. Convert measurements of form '11101' to 2D-array of inte
    matrix = np.array([list(bitstring) for bitstring in measureme

    # 2. Interpret matrix as using arithmetic mod 2, and find nul
    null_space = galois.GF(2)(matrix).null_space()
    print("Null space:")
    display(null_space)

    # 3. Convert back to a string
    print("Guess for hidden string s:")
    if len(null_space) == 0:
        # No non-trivial solution; `s` is all-zeros
        return "0" * len(measurements[0])
    return "".join(np.array(null_space[0]).astype(str))
```

In [21]:

```
display(simon_algorithm(simon_function("10011")))
```

```
Measurement results:

['11010',
 '01100',
 '01000',
 '00111',
 '01100',
 '10110',
 '00111',
 '00100',
 '10101',
 '00000',
 '10010',
 '01000',
 '00000',
 '01011',
 '10001']

Null space:

GF([[1, 0, 0, 1, 1]], order=2)

Guess for hidden string s:

'10011'
```

In [ ]:

```

```