In [1]:
```python
# imports:
import numpy as np
from qiskit import IBMQ, Aer
from qiskit import QuantumCircuit, assemble, transpile
```

Deutsch-Jozsa

Author: Bruna Shinohara de Mendonça. Last updated: 10 May 2022.

This might not be you first rodeo with the concept of Oracle [1], but here we will find it in a different context.

Consider that we have a function that, given an input (the qubits in the register), it may return:

I - All measurements in state $|0\rangle$ or in state $|1\rangle$;

II - Half of the measurements are in state $|0\rangle$ and half in state $|1\rangle$.

Function I is said to be **constant** and function II is said to be **balanced**.

Going back to classical computation: how could we differentiate these functions? In that situation, we may need to check half of the inputs and one more: this would be the "worst case" scenario, i.e., the situation where all the states in the first half are either zero or one, so we would need to check one more state to make sure it is (or isn't constant). In other words, for a number n of inputs, the worst case scenario requires

$$\text{Shots}_{\text{Classical}} = n/2 + 1$$

Let's get quantum?

The neat thing about Deutsch-Jozsa algorithm is that calling the function only one time is enough to tell its type! We don't even have a "worst" case, since we always have

$$\text{Shots}_{\text{Quantum}} = 1$$

How?

Here is the algorithm for a generic number of input qubits. We need two registers: one will store the n qubits that will be used as input, initialized in $|0\rangle$. The other will store a $|1\rangle$ state, that will work as an auxiliary qubit.

Thus, our first step is:

1. Prepare two quantum registers as instructed above:

$$|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle$$

2. Apply Hadamard gates to all qubits, including the auxiliary one. We may write the obtained state as:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}|x\rangle\left(|0\rangle - |1\rangle\right)$$

   Note the appearance of a misterious $|x\rangle$ state. This generic way of writing it is intentional, since this state is only necessary for the implementation of the oracle, but it does not intefere in the final state.
3. Apply the quantum oracle. Here, it will act as $|x\rangle|y\rangle$ to $|x\rangle|y\oplus f(x)\rangle$, $\oplus$ being the operation of addition modulo 2 [2]:

$$\begin{aligned}|\psi_2\rangle &= \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}|x\rangle(|f(x)\rangle - |1\oplus f(x)\rangle)\\ &= \frac{1}{\sqrt{2^{n+1}}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle)\end{aligned}$$

4. We now apply Hadamard gates to all qubit in the first register:

$$\begin{aligned}|\psi_3\rangle &= \frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}\left[\sum_{y=0}^{2^n-1}(-1)^{x\cdot y}|y\rangle\right]\\ &= \frac{1}{2^n}\sum_{y=0}^{2^n-1}\left[\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\right]|y\rangle\end{aligned}$$

   where $x\cdot y = x_0y_0\oplus x_1y_1\oplus\ldots\oplus x_{n-1}y_{n-1}$. The $|x\rangle$ state is already omitted from the equations, as it does not interfere at all after this point.
5. Measure the first register.
   We will have two possible outputs when evaluating the probability of the final state to be in $|0\rangle^{\otimes n}$. This is given by

$$P(|0\rangle^{\otimes n}) = |\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}(-1)^{x\cdot y}\langle 0^{\otimes n}|y\rangle|^2$$

   Note that $\langle 0^{\otimes n}|y\rangle = 1$ if $|y\rangle = |0^{\otimes n}\rangle$ (and zero otherwise). When $\langle 0^{\otimes n}|y\rangle = 1$, $(-1)^{f(x)}(-1)^{x\cdot y} = (-1)^{f(x)}$ Therefore, our probability reduces to

$$P(|0\rangle^{\otimes n}) = |\frac{1}{2^n}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|^2$$

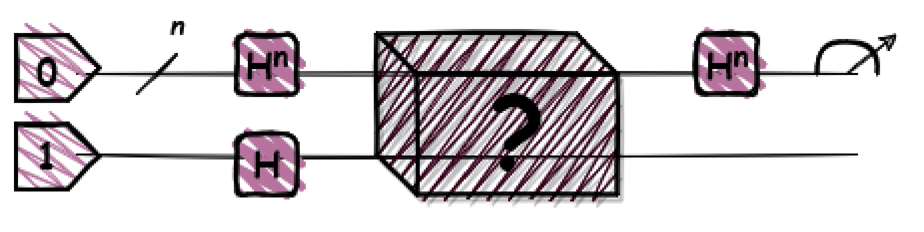
   Now, if our f(x) is balanced, the equal number of zeros and ones will cancel the terms in the summation. Something like:

$$P(|0\rangle^{\otimes n}) = |\frac{1}{2^n}(-1)^0 + (-1)^1 + (-1)^0 + (-1)^1 + \ldots|^2 = 0$$

   However, if our f(x) is constant, the sum would lead to:

$$P(|0\rangle^{\otimes n}) = |\frac{1}{2^n}(-1)^{f(x)}2^n|^2 = 1$$

   since $|(-1)^0|^2 = |(-1)^1|^2 = 1$. Here is a sketch of the circuit:



Example: two-qubit register

How about we implement Deutsch-Jozsa algorithm in Qiskit? Let us start simple, $n = 2$.

I: Create initial states of the registers

In this case, we initialize by creating $|0,0\rangle\otimes|1\rangle$.

In [2]:
```python
n=2
dj_circuit = QuantumCircuit(n+1, n)
```

note that we use `n+1` qubits, since we need on more qubit to use as auxiliary.

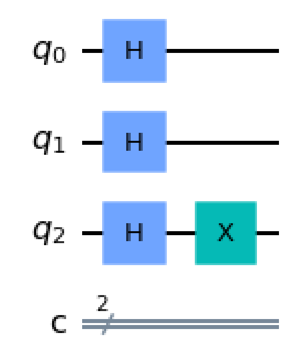
II: Add Hadamard gates

In [3]:
```python
bits = []

for qubit in range(n+1):
    dj_circuit.h(qubit)

# Put the n qubit in state |->
dj_circuit.x(n)
dj_circuit.draw(output = 'mpl')
```

Out[3]:



Here, we also use a X gate to obtain a $|-\rangle$ state, as seen in the original algorithm.

III: Call the oracle

The oracle function may be constructed in different ways, depending on whether you are aiming for a balanced or constant function. Here, we use an oracle that can be both, depending on the parameter `"bits"`.

This oracle is inspired by this QHack 2022 problem [3] - feel free to try it out!

In [4]:
```python
def oracle(bits):
    for i in bits:
        return dj_circuit.x(i)
```

How exactly this oracle works? We will see this in a second with an example, hold on ;)

IV: Apply Hadamard to qubits in the first register:

In [5]:
```python
for qubit in range(n):
    dj_circuit.h(qubit)
```

Finally,

V: Do the measurement

In [6]:
```python
for i in range(n):
    dj_circuit.measure(i, i)
```

All together now:

In [7]:
```python
# I: Create state |00>x|1>

n=2

dj_circuit = QuantumCircuit(n+1, n)

def full_circuit(bits):

    # II: Apply Hadamard to all
    for qubit in range(n+1):
        dj_circuit.h(qubit)

    # Put the n qubit in state |->
    dj_circuit.x(n)
    dj_circuit.h(n)
    dj_circuit.draw()


    # III: Call Oracle

    oracle(bits)

    # IV: Apply Hadamard to all qubits in the first register

    for qubit in range(n):
        dj_circuit.h(qubit)

    # V: Measure first register
    for i in range(n):
        measurements = dj_circuit.measure(i, i)

    # get results

    aer_sim = Aer.get_backend('aer_simulator')
    qobj = assemble(dj_circuit, aer_sim)
    results = aer_sim.run(qobj).result()
    answer = results.get_counts()

    return list(answer)[0]
```

We can now create a Python function to tell us whether the funcion is balanced or constant:

In [8]:
```python
def oracle_type(bits):
    sample = full_circuit(bits)
    ans = ''
    if sample[0]=='0' and sample[1]=='0':
        ans = "constant"
    else:
        ans = "balanced"
    return print(ans)
```

Let's call the function, having [1,1] as an input:

In [9]:
```python
bits1 = [1,1]
oracle_type(bits1)
```

```
constant
```

Why is it constant? Let's take a look at our oracle's definition:

```python
def oracle(bits):
    for i in bits:
        return dj_circuit.x(i)
```

Therefore, in this example, our function is producing

```python
dj_circuit.x(1)
dj_circuit.x(1)
```

However, our note that CNOT is reversible (you may check it yourself by multiplying the corresponding matrices!). Thus, applying it twice does not change anything.

Some final questions

1. How to construct different types of oracles? You can find different approaches for both balanced and constant types. Be creative!
2. How to generalize our code for more qubits?

References

[1]: Grover's algorthm is covered here: https://qiskit.org/textbook/ch-algorithms/grover.html

[2]: Modular arithmetic: https://en.wikipedia.org/wiki/Modular_arithmetic

[3]: QHACK problem:
https://github.com/XanaduAI/QHack/blob/master/Coding_Challenges/algorithms_100_DeutschJozsa_template/deutsch_jozsa_template.py