Trabalho Prático Interação e Concorrência 21/22

- Tiago Carriço A91695
- Inês Presa A90355

In [1]:
```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, Aer, execute, IBMQ
from qiskit.tools.visualization import plot_histogram, visualize_transition
from qiskit.tools import job_monitor
import matplotlib.pyplot as plt
import numpy as np
```

Fórmula booleana utilizada

De modo a obter uma fórmula com uma só solução, partiu-se da equação 1 apresentada no enunciado, acrecentando novas cláusulas.

$$f = (\neg v_1 \lor \neg v_2 \lor \neg v_3) \land (v_1 \lor \neg v_2 \lor v_3) \land (v_1 \lor v_2 \lor \neg v_3) \land (v_1 \lor \neg v_2 \lor \neg v_3) \land (\neg v_1 \lor v_2 \lor v_3) \land (v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor \neg v_3)$$

$$c_1 = (\neg v_1 \lor \neg v_2 \lor \neg v_3)$$

$$c_2 = (v_1 \lor \neg v_2 \lor v_3)$$

$$c_3 = (v_1 \lor v_2 \lor \neg v_3)$$

$$c_4 = (v_1 \lor \neg v_2 \lor \neg v_3)$$

$$c_5 = (\neg v_1 \lor v_2 \lor v_3)$$

$$c_6 = (\neg v_1 \lor \neg v_2 \lor v_3)$$

$$c_7 = (v_1 \lor v_2 \lor v_3)$$

v_1	v_2	v_3	c_1	c_2	c_3	c_4	c_5	c_6	c_7	f
0	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	0	1	1	1	1	0
0	1	0	1	0	1	1	1	1	1	0
0	1	1	1	1	1	0	1	1	1	0
1	0	0	1	1	1	1	0	1	1	0
1	0	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	0	1	0
1	1	1	0	1	1	1	1	1	1	0

Simulação

- `execute_circuit(qc)` - executa o circuito e devolve os resultados obtidos

In [2]:
```
def execute_circuit(qc, shots=1024, decimal=False):
    #define backend
    device = Aer.get_backend('qasm_simulator')
    #get counts
    counts = device.run(qc, shots=shots).result().get_counts()

    if decimal:
        counts = dict((int(a[::-1],2),b) for (a,b) in counts.items())
    else:
        counts = dict((a[::-1],b) for (a,b) in counts.items())

    return counts
```

- `init(flips)` - inicializa o circuito quântico com o número de qubits necessários (3 bits inciais + 1 ancilla por cada cláusula + 1 ancilla final) e coloca todos estes qubits em superposição uniforme ($|+\rangle$ nos bits iniciais e $|-\rangle$ nas ancillas)

In [4]:
```
def init(flips):
    qr = QuantumRegister(3)
    ancilla = QuantumRegister(len(flips) + 1)
    cr = ClassicalRegister(3)
    qc = QuantumCircuit(qr, ancilla, cr)

    qc.h(qr)
    qc.x(ancilla)
    qc.h(ancilla)
    qc.barrier()
    return qc, qr, ancilla, cr
```

- `diffusion_operator(qr, ancilla)` - aplica o operador de difusão a um circuito quântico e devolve esse circuito

In [5]:
```
def diffusion_operator(qr, ancilla):
    qc = QuantumCircuit(qr,ancilla)

    qc.h(qr)
    qc.x(qr)
    qc.h(qr[-1])
    # controlos sao todos menos o ultimo qubit e o target é o ultimo
    qc.mcx(qr[:-1],qr[-1])
    qc.h(qr[-1])
    qc.x(qr)
    qc.h(qr)

    qc.barrier()
    return qc
```

- `oracle(qr, ancilla, flips)` - Uma vez que não é possível utilizar o operador da disjunção em qubits, começou-se por reescrever a fórmula da seguinte forma:

$$f = \neg(v_1 \wedge v_2 \wedge v_3) \wedge \neg(\neg v_1 \wedge v_2 \wedge \neg v_3) \wedge \neg(\neg v_1 \wedge \neg v_2 \wedge v_3) \wedge \neg(\neg v_1 \wedge v_2 \wedge v_3) \wedge \neg(v_1 \wedge \neg v_2 \wedge \neg v_3) \wedge \neg(\neg v_1 \wedge \neg v_2 \wedge \neg v_3) \wedge \neg(v_1 \wedge \neg v_2 \wedge v_3)$$

O argumento flips representa a fórmula, indicando para cada cláusula se esta deve ou não ser negada. Com isto, aplica-se, em cada cláusula, a gate x aos qubits que devem ser negados, a gate mcx aos 3 qubits originais e a gate x à ancilla da cláusula. Finalmente, aplica-se a gate mcx às ancillas de cada cláusula, usando a última ancilla como target.

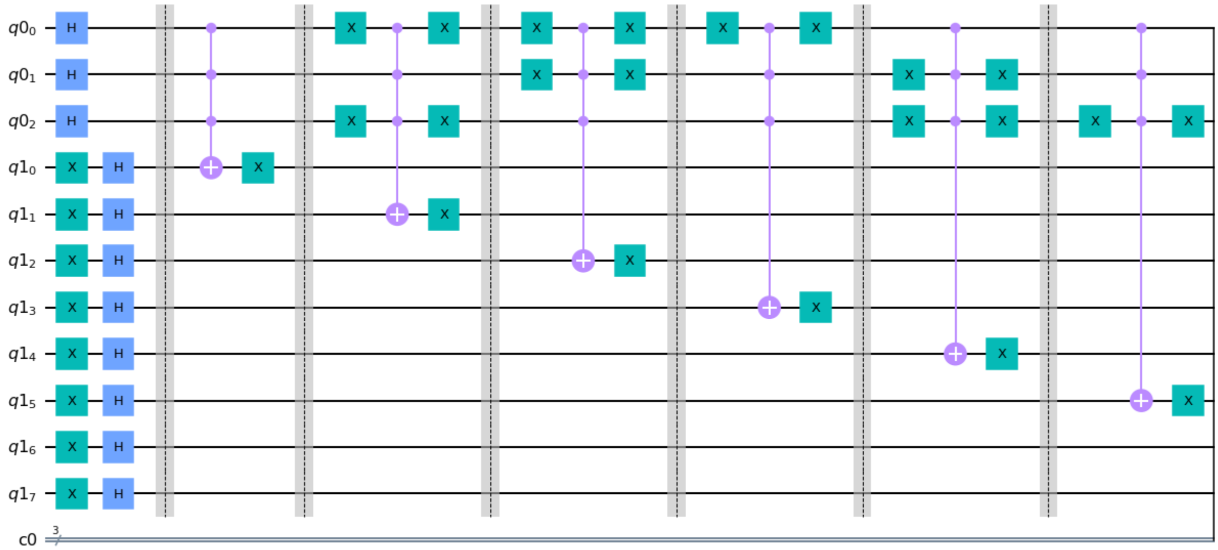
In [6]:
```
def oracle(qr, ancilla, flips):
    qc = QuantumCircuit(qr, ancilla)
    clauses = len(flips)

    for k, flip in enumerate(flips):
        for i in range(3):
            if not flip[i]:
                qc.x(i)

        qc.mcx(qr, ancilla[k])
        qc.x(ancilla[k])

        for i in range(3):
            if not flip[i]:
                qc.x(i)

        qc.barrier()
    qc.mcx(ancilla[:-1],ancilla[-1])
    qc.barrier()
    return qc
```

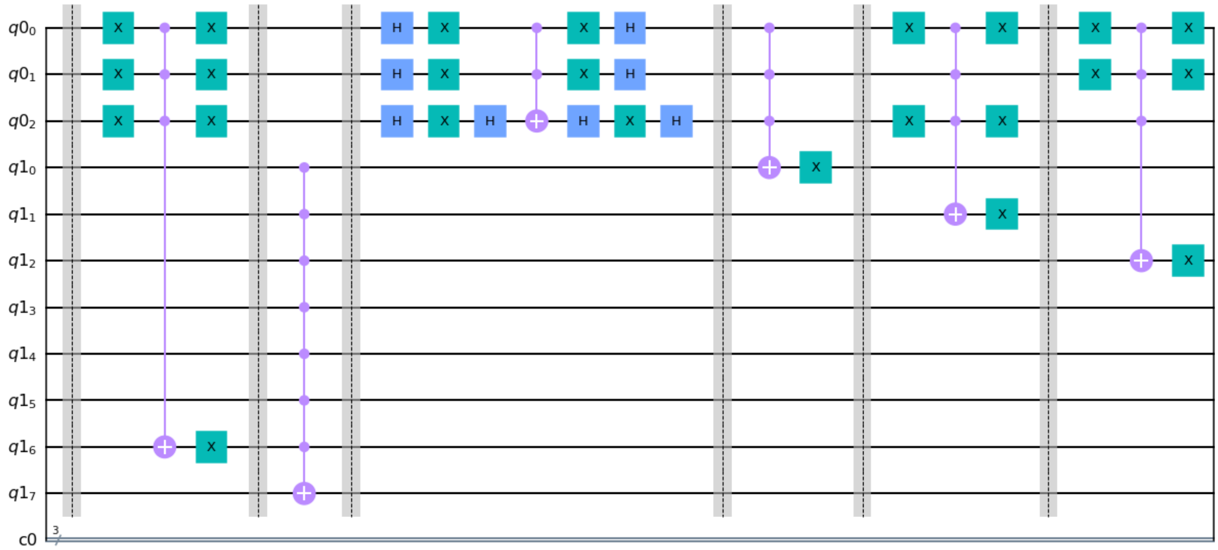
- `grover(qc, qr, ancilla, oracle)` - Calcula o número de iterações do algortimo e aplica, em cada iteração, o oráculo e o operador de difusão

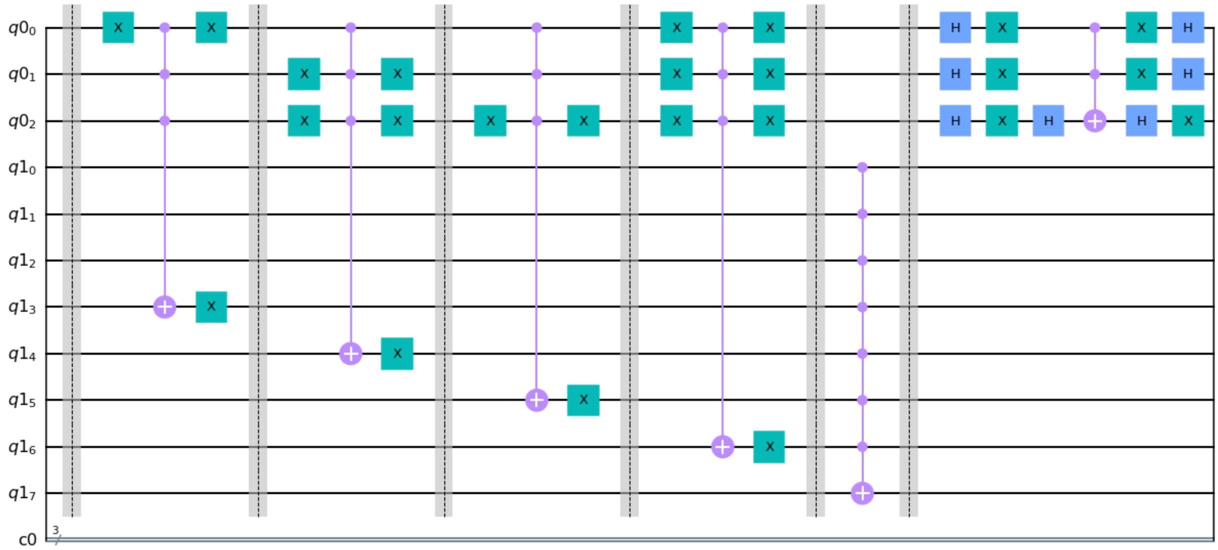
In [7]:
```
def grover(qc, qr, ancilla, oracle):
    elements = 2**3

    iterations = int(np.floor(np.pi/4 * np.sqrt(elements)))

    for j in range(iterations):
        qc = qc.compose(oracle(qr, ancilla, flips))
        qc = qc.compose(diffusion_operator(qr, ancilla))
    return qc
```

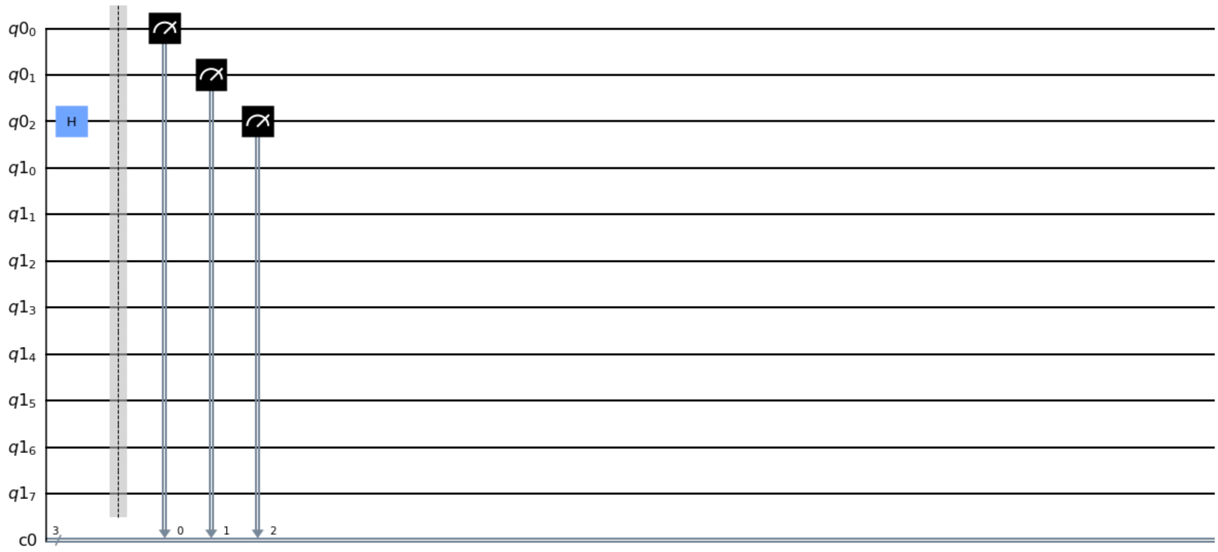
In [8]:
```python
flips = [[True, True, True],
         [False, True, False],
         [False, False, True],
         [False, True, True],
         [True, False, False],
         [True, True, False],
         [False, False, False]]

qc, qr, ancilla, cr = init(flips)
qc = grover(qc, qr, ancilla, oracle)
qc.measure(qr, cr)
qc.draw(output="mpl")
```

Out[8]:



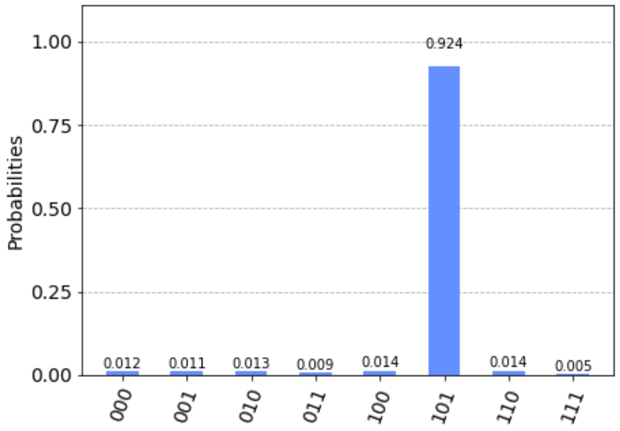






```
In [9]: plot_histogram(execute_circuit(qc, 1024))
```

Out[9]:



Como esperado, a solução da fórmula obtida foi 101.

IBM

Uma vez que os computadores quânticos disponibilizados pela IBM têm um limite de qubits inferior ao necessário (11 qubits), utilizou-se um simulador para o efeito.

```
In [10]: #IBMQ.active_account()
         #IBMQ.save_account(ibm_token, overwrite=True)
         IBMQ.load_account()

Out[10]: <AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>

In [11]: provider = IBMQ.get_provider()
         provider.backends(simulator=True, operational=True)

Out[11]: [<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
          <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open', project='main')>,
          <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open', project='main')>,
          <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>,
          <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open', project='main')>]

In [14]: backend = provider.get_backend("simulator_mps")
         backend.configuration().n_qubits

Out[14]: 100

In [15]: job = execute(qc, backend)

In [16]: job_monitor(job)

         Job Status: job has successfully run
```

In [17]:
```
result = job.result()
counts = result.get_counts()
plot_histogram(counts)
```

Out[17]: