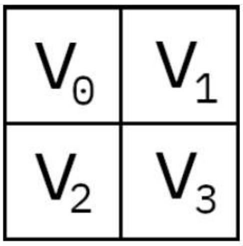
In [5]:

```python
#initialization
import matplotlib.pyplot as plt
import numpy as np

# importing Qiskit
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.providers.ibmq import least_busy

# import basic plot tools
from qiskit.visualization import plot_histogram
from qiskit_textbook.problems import grover_problem_oracle
```

In [20]:

```python
def diffuser(nqubits):
    qc = QuantumCircuit(nqubits)
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(qubit)
    # Apply transformation |00..0> -> |11..1> (X-gates)
    for qubit in range(nqubits):
        qc.x(qubit)
    # Do multi-controlled-Z gate
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)  # multi-controlled-toffoli
    qc.h(nqubits-1)
    # Apply transformation |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)
    # Apply transformation |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)
    # We will return the diffuser as a gate
    U_s = qc.to_gate()
    U_s.name = "U$_s$"
    return U_s
```

Turning the Problem into a Circuit



we want our circuit to output a solution to this sudoku.

Note that, while this approach of using Grover's algorithm to solve this problem is not practical (you can probably find the solution in your head!), the purpose of this example is to demonstrate the conversion of classical decision problems into oracles for Grover's algorithm.

5.1 Turning the Problem into a Circuit

We want to create an oracle that will help us solve this problem, and we will start by creating a circuit that identifies a correct solution. Similar to how we created a classical adder using quantum circuits in *The Atoms of Computation*, we simply need to create a *classical* function on a quantum circuit that checks whether the state of our variable bits is a valid solution.

Since we need to check down both columns and across both rows, there are 4 conditions we need to check:

```
v0 ≠ v1   # check along top row
v2 ≠ v3   # check along bottom row
v0 ≠ v2   # check down left column
v1 ≠ v3   # check down right column
```

In [6]:

```python
clause_list = [[0,1],
               [0,2],
               [1,3],
               [2,3]]
```

In [14]:

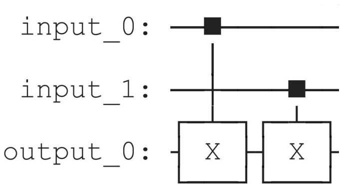
```python
#To check these clauses computationally, we will use the XOR gate (we came across th
def XOR(qc, a, b, output):
    qc.cx(a, output)
    qc.cx(b, output)
```

Convince yourself that the output0 bit in the circuit below will only be flipped if input0 ≠ input1:

In [15]:

```python
# We will use separate registers to name the bits
in_qubits = QuantumRegister(2, name='input')
out_qubit = QuantumRegister(1, name='output')
qc = QuantumCircuit(in_qubits, out_qubit)
XOR(qc, in_qubits[0], in_qubits[1], out_qubit)
qc.draw()
```

Out[15]:



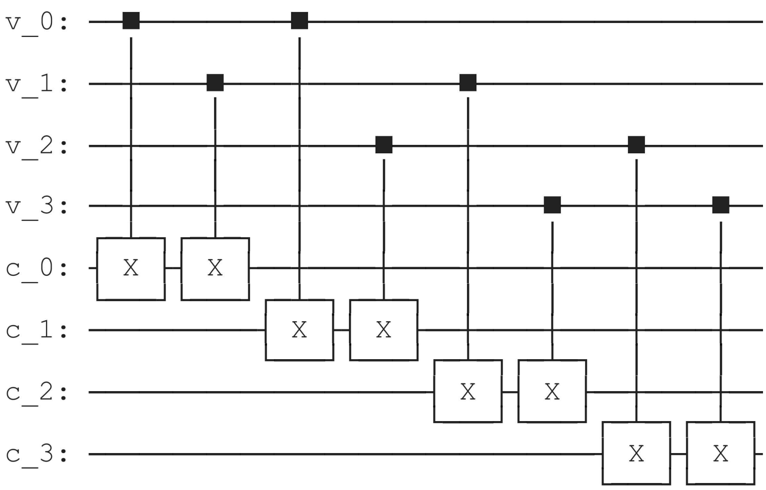
This circuit checks whether input0 == input1 and stores the output to output0. To check each clause, we repeat this circuit for each pairing in clause_list and store the output to a new bit

In [16]:

```python
# Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v')  # variable bits
clause_qubits = QuantumRegister(4, name='c')  # bits to store clause-checks
```

```python
# Create quantum circuit
qc = QuantumCircuit(var_qubits, clause_qubits)

# Use XOR gate to check each clause
i = 0
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

qc.draw()
```

Out[16]:

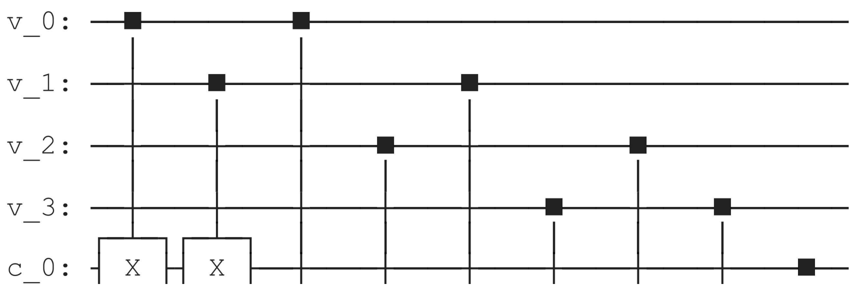


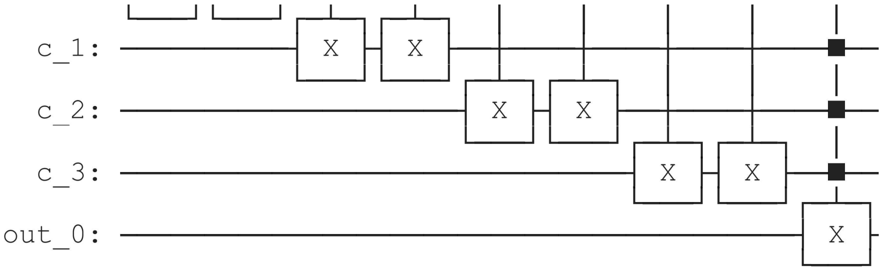
The final state of the bits c0, c1, c2, c3 will only all be 1 in the case that the assignments of v0, v1, v2, v3 are a solution to the sudoku. To complete our checking circuit, we want a single bit to be 1 if (and only if) all the clauses are satisfied, this way we can look at just one bit to see if our assignment is a solution. We can do this using a multi-controlled-Toffoli-gate:

In [17]:

```python
# Create separate registers to name bits
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit)

# Compute clauses
i = 0
for clause in clause_list:
    XOR(qc, clause[0], clause[1], clause_qubits[i])
    i += 1

# Flip 'output' bit if all clauses are satisfied
qc.mct(clause_qubits, output_qubit)

qc.draw()
```

Out[17]:





The circuit above takes as input an initial assignment of the bits v0, v1, v2 and v3, and all other bits should be initialized to 0. After running the circuit, the state of the out0 bit tells us if this assignment is a solution or not; out0 = 0 means the assignment is not a solution, and out0 = 1 means the assignment is a solution.

Uncomputing, and Completing the Oracle

- One register which stores our sudoku variables (we'll say $x = v_3, v_2, v_1, v_0$)
- One register that stores our clauses (this starts in the state $|0000\rangle$ which we'll abbreviate to $|0\rangle$)
- And one qubit ($|\text{out}_0\rangle$) that we've been using to store the output of our checking circuit.

To create an oracle, we need our circuit (U_ω) to perform the transformation:

$$U_\omega|x\rangle|0\rangle|\text{out}_0\rangle = |x\rangle|0\rangle|\text{out}_0 \oplus f(x)\rangle$$

If we set the `out0` qubit to the superposition state $|{-}\rangle$ we have:

$$\begin{aligned} U_\omega|x\rangle|0\rangle|{-}\rangle &= U_\omega|x\rangle|0\rangle \otimes \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= |x\rangle|0\rangle \otimes \tfrac{1}{\sqrt{2}}(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) \end{aligned}$$

If $f(x) = 0$, then we have the state:

$$\begin{aligned} &= |x\rangle|0\rangle \otimes \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= |x\rangle|0\rangle|{-}\rangle \end{aligned}$$

(i.e. no change). But if $f(x) = 1$ (i.e. $x = \omega$), we introduce a negative phase to the $|{-}\rangle$ qubit:

$$\begin{aligned} &= \quad |x\rangle|0\rangle \otimes \tfrac{1}{\sqrt{2}}(|1\rangle - |0\rangle) \\ &= \quad |x\rangle|0\rangle \otimes -\tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= -|x\rangle|0\rangle|{-}\rangle \end{aligned}$$

This is a functioning oracle that uses two auxiliary registers in the state $|0\rangle|{-}\rangle$:

$$U_\omega|x\rangle|0\rangle|{-}\rangle = \begin{cases} \;\;|x\rangle|0\rangle|{-}\rangle & \text{for } x \neq \omega \\ -|x\rangle|0\rangle|{-}\rangle & \text{for } x = \omega \end{cases}$$

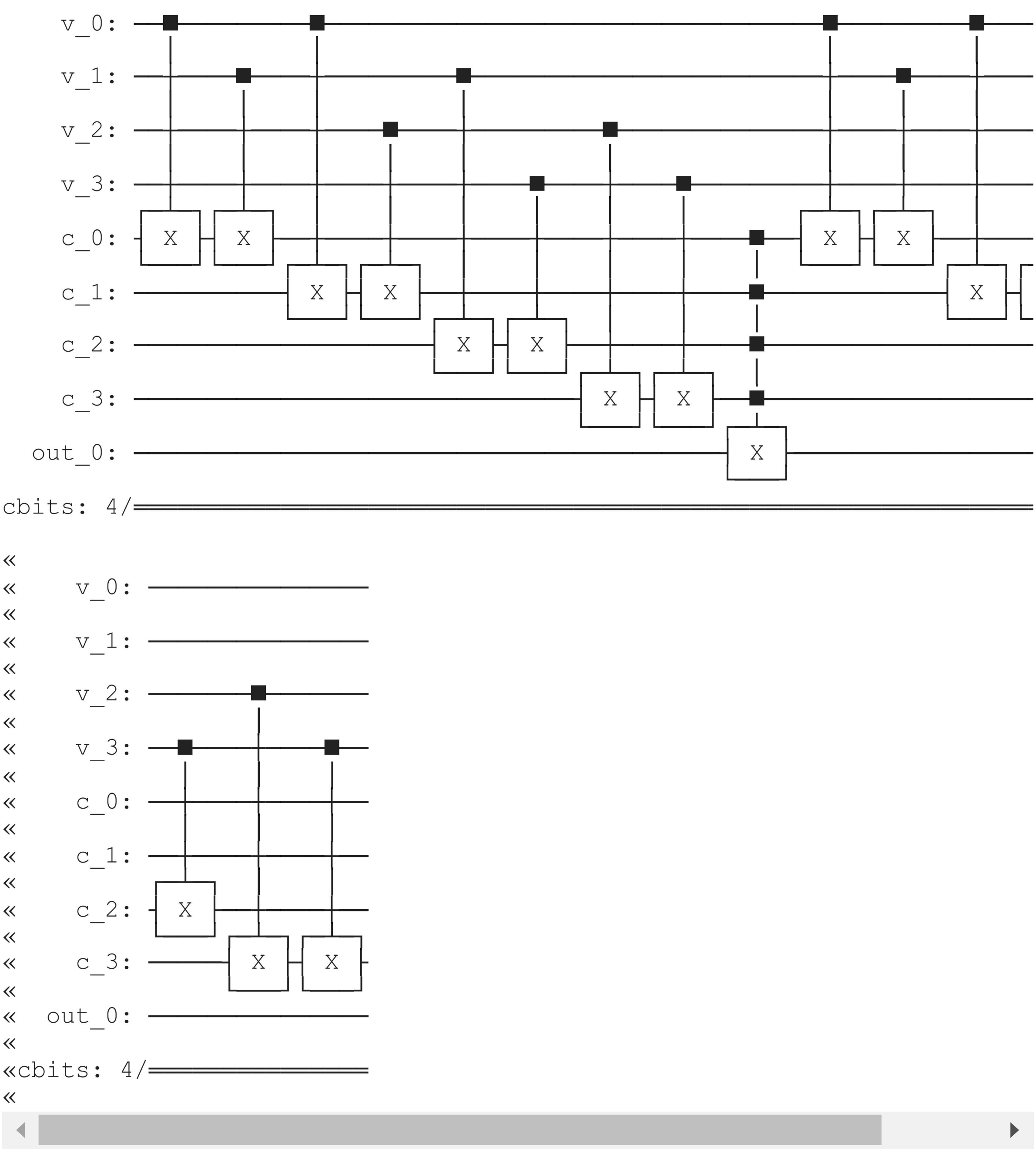
To adapt our checking circuit into a Grover oracle, we need to guarantee the bits in the second register (`c`) are always returned to the state $|0000\rangle$ after the computation. To do this, we simply repeat the part of the circuit that computes the clauses which guarantees `c0 = c1 = c2 = c3 = 0` after our circuit has run. We call this step *'uncomputation'*.

In [18]:

```
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
cbits = ClassicalRegister(4, name='cbits')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

def sudoku_oracle(qc, clause_list, clause_qubits):
```

```
    # Compute clauses
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

    # Flip 'output' bit if all clauses are satisfied
    qc.mct(clause_qubits, output_qubit)

    # Uncompute clauses to reset clause-checking bits to 0
    i = 0
    for clause in clause_list:
        XOR(qc, clause[0], clause[1], clause_qubits[i])
        i += 1

sudoku_oracle(qc, clause_list, clause_qubits)
qc.draw()
```

Out[18]:

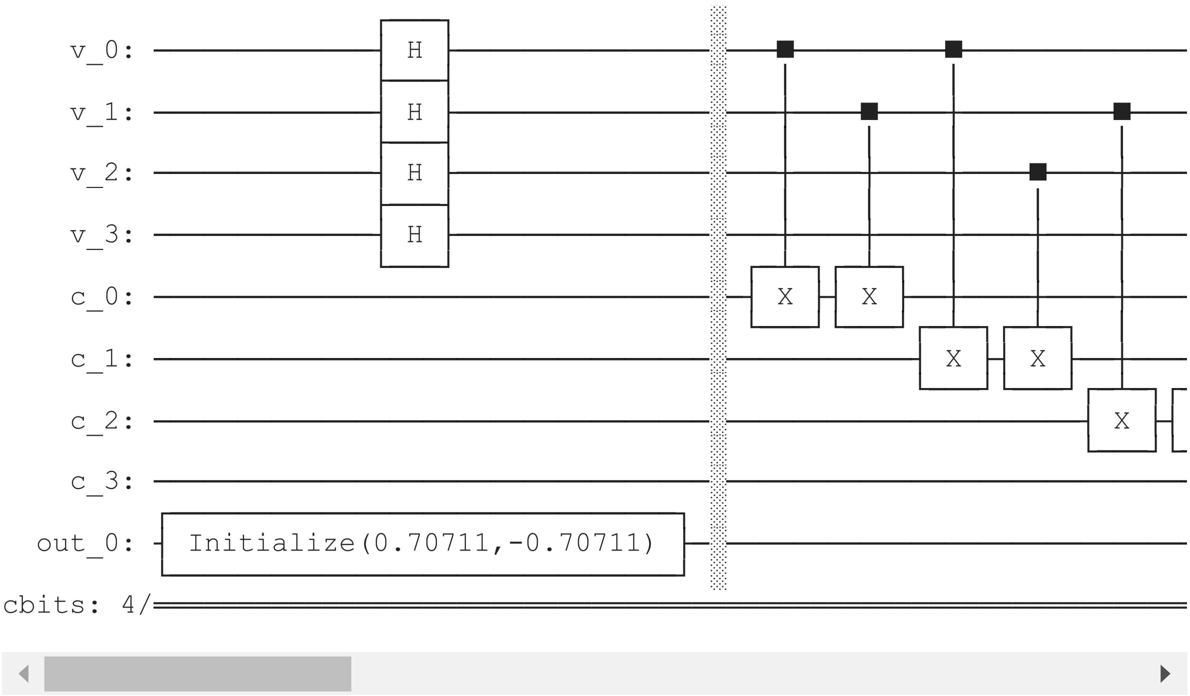


The full Algorithm

In [21]:

```python
var_qubits = QuantumRegister(4, name='v')
clause_qubits = QuantumRegister(4, name='c')
output_qubit = QuantumRegister(1, name='out')
cbits = ClassicalRegister(4, name='cbits')
qc = QuantumCircuit(var_qubits, clause_qubits, output_qubit, cbits)

# Initialize 'out0' in state |->
qc.initialize([1, -1]/np.sqrt(2), output_qubit)

# Initialize qubits in state |s>
qc.h(var_qubits)
qc.barrier()  # for visual separation

## First Iteration
# Apply our oracle
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier()  # for visual separation
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

## Second Iteration
sudoku_oracle(qc, clause_list, clause_qubits)
qc.barrier()  # for visual separation
# Apply our diffuser
qc.append(diffuser(4), [0,1,2,3])

# Measure the variable qubits
qc.measure(var_qubits, cbits)

qc.draw(fold=-1)
```

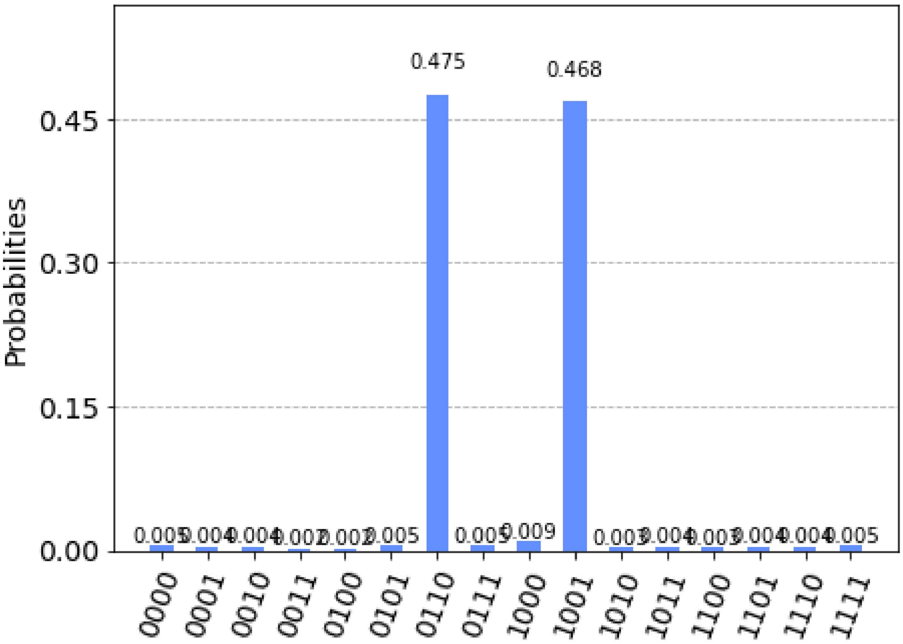
Out[21]:



In [23]:

```python
# Simulate and plot results
aer_simulator = Aer.get_backend('aer_simulator')
transpiled_qc = transpile(qc, aer_simulator)
qobj = assemble(transpiled_qc)
result = aer_simulator.run(qobj).result()
plot_histogram(result.get_counts())
```

Out[23]:



There are two bit strings with a much higher probability of measurement than any of the others, `0110` and `1001`. These correspond to the assignments:

```
v0 = 0
v1 = 1
v2 = 1
v3 = 0
```

and

```
v0 = 1
v1 = 0
v2 = 0
v3 = 1
```

which are the two solutions to our sudoku! The aim of this section is to show how we can create Grover oracles from real problems. While this specific problem is trivial, the process can be applied (allowing large enough circuits) to any decision problem. To recap, the steps are:

1. Create a reversible classical circuit that identifies a correct solution
2. Use phase kickback and uncomputation to turn this circuit into an oracle
3. Use Grover's algorithm to solve this oracle

In [ ]: