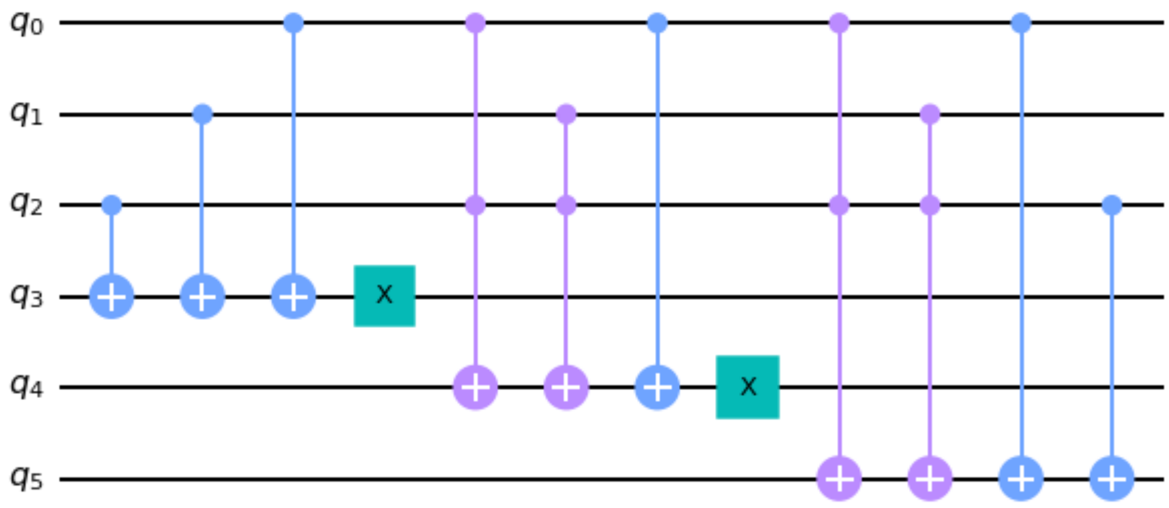
In [2]:
```
import numpy as np

# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestream.MessageStream size changed, m
ay indicate binary incompatibility. Expected 56 from C header, got 64 from PyObject
```

In [3]:
```
from qiskit import *
```

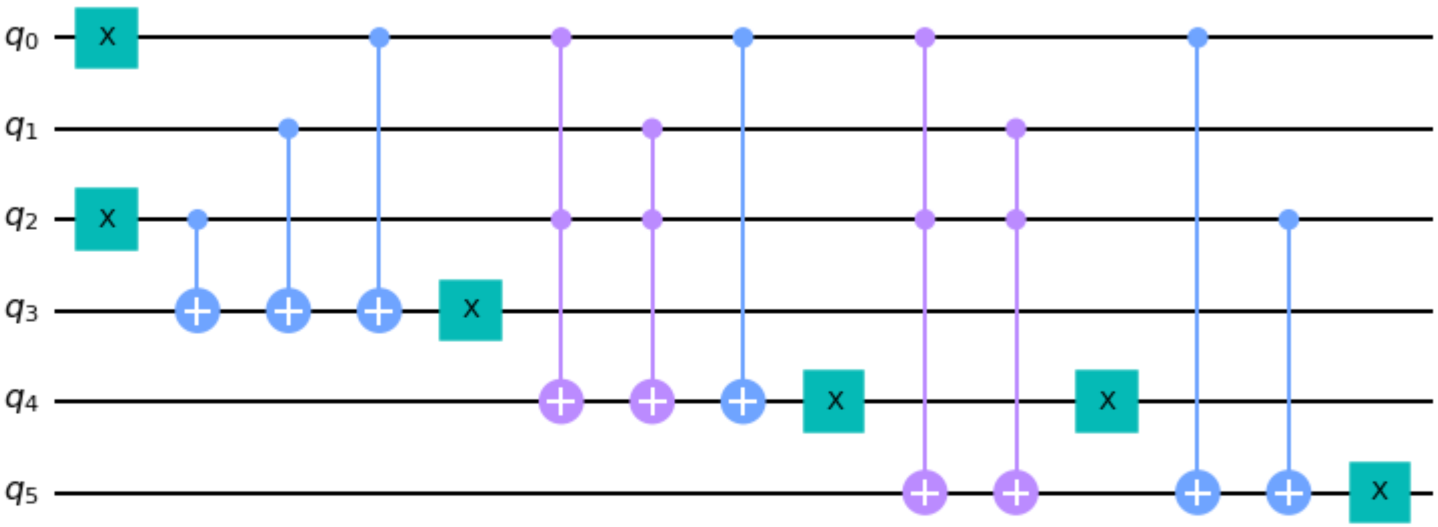
In [4]:
```
from qiskit import *
#Let P be a permutation which takes(0 6 5 7)(1 3 4)(2)
# so its ANF form is with inputs x, y and z is y1=(x1+x2+x3)', y2=(x1x3+x2x3+x1)',
#y3=x1x3+x2x3+x1+x3
#Public permutation
q = QuantumRegister(6, 'q')


P=QuantumCircuit(q)

P.cx(q[2], q[3])
P.cx(q[1], q[3])
P.cx(q[0], q[3])
P.x(q[3])#y1 =(x1+x2+x3)'

P.ccx(q[0], q[2], q[4])
P.ccx(q[1], q[2], q[4])
P.cx(q[0], q[4])
P.x(q[4]) #y2=(x1x3+x2x3+x)'

P.ccx(q[0], q[2], q[5])
P.ccx(q[1], q[2], q[5])
P.cx(q[0], q[5])
P.cx(q[2], q[5]) #y3=x1x3+x2x3+x1+x3




P.draw()
```

Out[4]:



In [5]:
```
from qiskit import *
#Let P be a permutation which takes(0 6 5 7)(1 3 4)(2)
# so its ANF form is with inputs x, y and z is y1=(x1+x2+x3)', y2=(x1x3+x2x3+x1)',
#y3=x1x3+x2x3+x1+x3
#Public permutation
#Even-Mansour cipher

#E(x)=P(x+k1)+k2
#k1=(1 0 1), k2 = (0 1 1)
#where k1 and k2 are our secret keys

q = QuantumRegister(6, 'q')

E=QuantumCircuit(q)
E.x(q[0])
E.x(q[2]) # k1=(1 0 1)

E.cx(q[2], q[3])
E.cx(q[1], q[3])
E.cx(q[0], q[3])
E.x(q[3])#y1 =(x1+x2+x3)'

E.ccx(q[0], q[2], q[4])
E.ccx(q[1], q[2], q[4])
E.cx(q[0], q[4])
E.x(q[4]) #y2=(x1x3+x2x3+x)'

E.ccx(q[0], q[2], q[5])
E.ccx(q[1], q[2], q[5])
E.cx(q[0], q[5])
E.cx(q[2], q[5]) #y3=x1x3+x2x3+x1+x3

E.x(q[4])
E.x(q[5])#k2 = (0 1 1)


E.draw()
```

Out[5]:

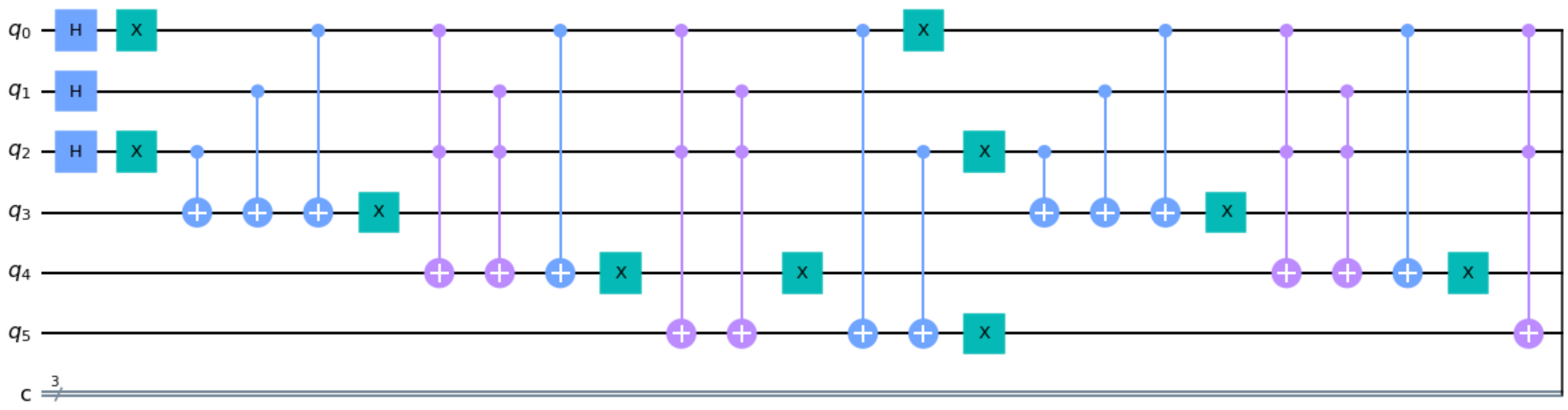


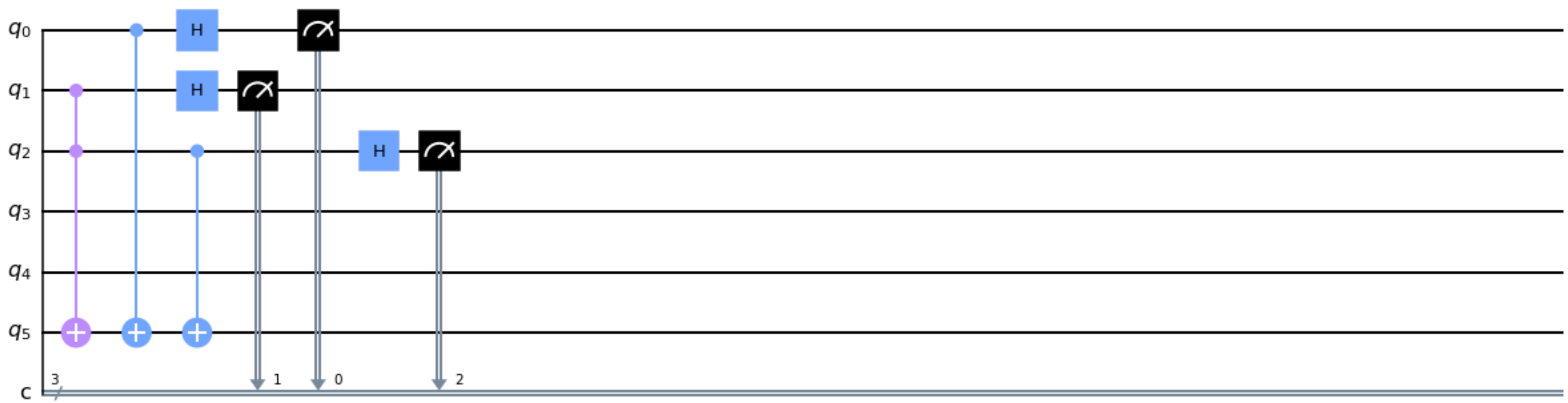
In [ ]:

In [12]:
```
q = QuantumRegister(6, 'q')

M = ClassicalRegister(3, 'c')
EM_k_r=QuantumCircuit(q, M)

for i in range(3):
    EM_k_r.h(q[i])

#simon's function
    #encryption oracle
EM_k_r.x(q[0])
EM_k_r.x(q[2]) # k1=(1 0 1)

EM_k_r.cx(q[2], q[3])
EM_k_r.cx(q[1], q[3])
EM_k_r.cx(q[0], q[3])
EM_k_r.x(q[3])#y1 =(x1+x2+x3)'

EM_k_r.ccx(q[0], q[2], q[4])
EM_k_r.ccx(q[1], q[2], q[4])
EM_k_r.cx(q[0], q[4])
EM_k_r.x(q[4]) #y2=(x1x3+x2x3+x)'

EM_k_r.ccx(q[0], q[2], q[5])
EM_k_r.ccx(q[1], q[2], q[5])
EM_k_r.cx(q[0], q[5])
EM_k_r.cx(q[2], q[5]) #y3=x1x3+x2x3+x1+x3

EM_k_r.x(q[0])
EM_k_r.x(q[2])
EM_k_r.x(q[4])
EM_k_r.x(q[5])#k2 = (0 1 1)

#passing on the output of encryption oracle to pulic permutation

EM_k_r.cx(q[2], q[3])
EM_k_r.cx(q[1], q[3])
EM_k_r.cx(q[0], q[3])
EM_k_r.x(q[3])
```

```
EM_k_r.ccx(q[0], q[2], q[4])
EM_k_r.ccx(q[1], q[2], q[4])
EM_k_r.cx(q[0], q[4])
EM_k_r.x(q[4])

EM_k_r.ccx(q[0], q[2], q[5])
EM_k_r.ccx(q[1], q[2], q[5])
EM_k_r.cx(q[0], q[5])
EM_k_r.cx(q[2], q[5])

for i in range(3):
    EM_k_r.h(q[i])
    EM_k_r.measure(q[i], M[i])
    #EM_k_r.measure(q[3+i], M[3+i])



EM_k_r.draw()
```

Out[12]:





In [14]:
```
backend = Aer.get_backend('qasm_simulator')
qjob = execute(EM_k_r, backend, shots=1000)
counts = qjob.result().get_counts()
print(counts)
```

```
{'000': 511, '111': 489}
```

In [ ]:

```
```

In [ ]:

```
```