Trabalho Prático

Grupo 7

Filipe Barbosa a77252

Hugo Ferreira a78555

Descrição

Neste trabalho prático é pretentido descobrir um s tal que $s \equiv N \; mod \; 8$,sendo N o número de grupo atribuido, usando um algoritmo quântico. O algoritmo quântico que vai ser usado é o algoritmo $Grover$. Os possíveis casos vão ser $000, 001, 010, 011, 100, 101, 110, 111$. Como no nosso caso $N = 7$, depois de converter para binário vamos ter que o resultado pretendido vai ser 111.

Algoritmo de Grover

Para a construção do algoritmo de Grover vamos ter as seguintes etapas:

- Inicialização
- Oráculo
- Amplificação

Inicialização

Vamos inicializar os estados do sistema com a mesma amplitude.

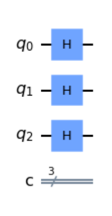
$$\sum_{x_i} |x_i\rangle$$

Esta inicialização é feita através da sobreposição dos $qubits$ aplicando um gate de Hadamard a cada um.

In [1]:
```
from qiskit import *
%matplotlib inline
from qiskit.tools.visualization import *
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy
```

In [2]:
```
n=3

cr = ClassicalRegister(n, 'c')
qr = QuantumRegister(n, 'q')

qc = QuantumCircuit(qr, cr)
qc.h(qr)
qc.draw('mpl')
```

Out[2]:

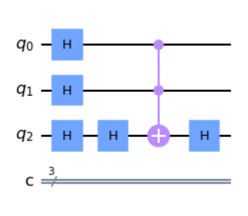


Oráculo

Vamos criar o oráculo que vai ser responsável por mudar a fase para o nosso número em binário pretendido.
Como no nosso caso o número pretendido é o 111 não vamos precisar aplicar nenhum gate-X e simplesmente vamos aplicar o um gate-CCZ que pode ser construido em $qiskit$ através da sequência gate-H, gate-CCX e gate-H.

In [3]:
```
def oracle(circuit,q):
    circuit.h(q[2])
    circuit.ccx(q[0],q[1],q[2])
    circuit.h(q[2])

oracle(qc, qr)
qc.draw(output='mpl')
```

Out[3]:



Através do statevector a seguir representado podemos verificar que o oráculo está a mudar de fase nos qubits pretendidos.

In [4]:
```
backend_state = Aer.get_backend('statevector_simulator')
```

In [5]:
```
result = execute(qc, backend_state).result()
qstate = result.get_statevector(qc)
```

Como o último elemento da matriz é o que está negativo, e esse elemento representa o 111 temos que o oracle está correto.

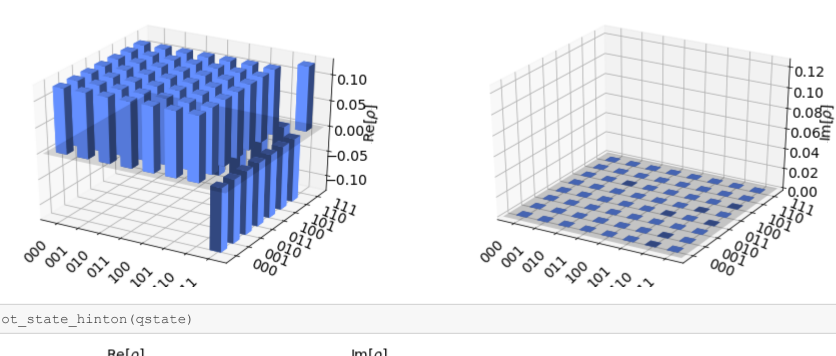
In [6]:
```
qstate
```

Out[6]:
```
array([ 0.35355339+0.00000000e+00j,  0.35355339+0.00000000e+00j,
        0.35355339+0.00000000e+00j,  0.35355339-4.32978028e-17j,
        0.35355339+0.00000000e+00j,  0.35355339+0.00000000e+00j,
        0.35355339+0.00000000e+00j, -0.35355339+4.32978028e-17j])
```

Apresentamos em seguida o statevector em forma de gráficos para melhor visualização.

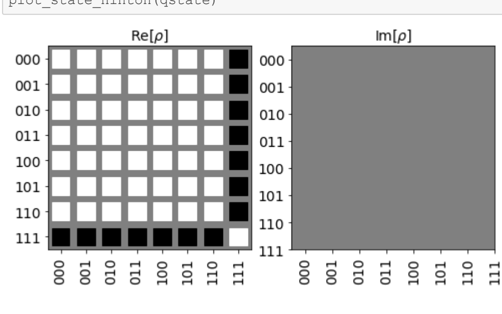
In [7]:
```
plot_state_city(qstate)
```

Out[7]:



In [8]:
```
plot_state_hinton(qstate)
```

Out[8]:



Amplificação

Nesta fase vamos criar um diffuser que nos vai permitir aumentar a amplitude dos qubits de forma a que aumente a probabilidade de obtermos o resultado que pretendemos. Para isso é necessário inverter e aumentar a amplitude da componente pretendidade em torno da média das amplitudes e simultaneamente reduzir das restantes componentes.

In [9]:
```
def diffuser(circuit,qr):
    circuit.h(qr)
    circuit.x(qr)

    circuit.h(qr[2])
    circuit.ccx(qr[0],qr[1],qr[2])
    circuit.h(qr[2])

    circuit.x(qr)
    circuit.h(qr)
```

Agora vamos criar um $QuantumCircuit$ que vai conter o nosso oráculo e $diffuser$ aproximadamente $\sqrt{2^n}$. O espectável seria fazer 3 iterações, usando $round(m.sqrt(2**n))$, mas obtivemos melhores resultados fazendo duas iterações, por isso usamos $m.floor(m.sqrt(2**n))$.

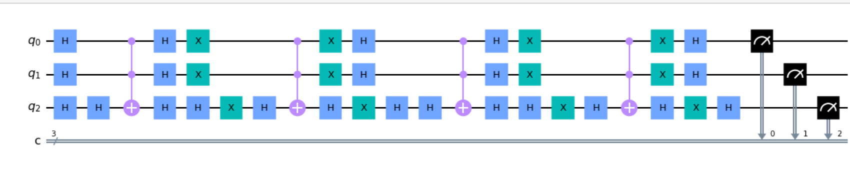
In [10]:
```
import math as m

times= m.floor(m.sqrt(2**n))
print(times)
```

```
2
```

Fazemos no fim a medição que consiste na projeção do estado quântico de cada um dos qubits para um bit clássico. Ao ser medido, cada qubit colapsa para um dos estados possíveis, não sendo mais possível obter o estado quântico inicial.

In [11]:
```
qc_Grover= QuantumCircuit(qr,cr)

qc_Grover.h(qr)

for t in range(times):
    oracle(qc_Grover,qr)

    diffuser(qc_Grover,qr)

qc_Grover.measure(qr,cr)

qc_Grover.draw(output='mpl')
```

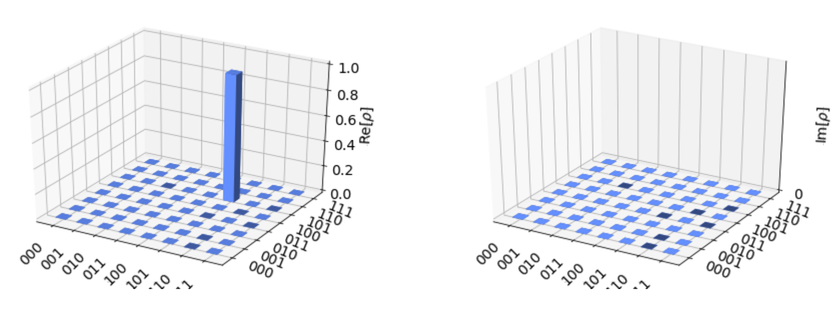
Out[11]:



Temos então que o estado que queremos é o que tem a maior probabilidade.

In [12]:
```
result = execute(qc_Grover, backend_state).result()
qstate = result.get_statevector(qc_Grover)
plot_state_city(qstate)
```

Out[12]:

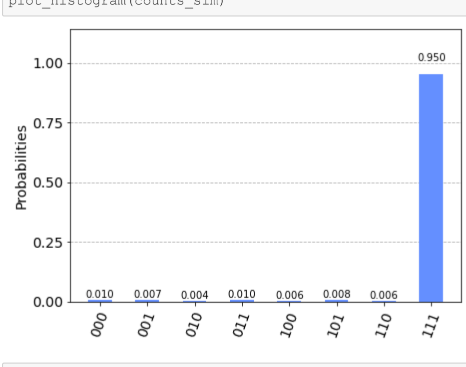


Simulação

In [13]:
```
backend = Aer.get_backend("qasm_simulator")
```

In [14]:
```
shots=1024
result = execute(qc_Grover, backend, shots=shots).result()
counts_sim = result.get_counts(qc_Grover)
plot_histogram(counts_sim)
```

Out[14]:



In [15]:
```
qc_Grover.depth()
```

Out[15]:
```
22
```

Simulação com noise

In [16]:
```
provider = IBMQ.load_account()
provider.backends()
```

Out[16]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_athens') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_quito') 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')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open', project='main')>]
```

In [17]:
```
import qiskit.tools.jupyter

%qiskit_backend_overview
```

In [18]:
```
from qiskit.tools.monitor import backend_overview, backend_monitor

backend_overview()
```

```
ibmq_manila                 ibmq_quito                  ibmq_belem
-----------                 ----------                  ----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 13            Pending Jobs: 14            Pending Jobs: 9
Least busy:   False         Least busy:   False         Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      163.8         Avg. T1:      80.3          Avg. T1:      75.9
Avg. T2:      68.2          Avg. T2:      71.2          Avg. T2:      75.6



ibmq_lima                   ibmq_santiago               ibmq_athens
---------                   -------------               -----------
Num. Qubits:  5             Num. Qubits:  5             Num. Qubits:  5
Pending Jobs: 3             Pending Jobs: 0             Pending Jobs: 5
Least busy:   False         Least busy:   True          Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      62.4          Avg. T1:      110.7         Avg. T1:      96.0
Avg. T2:      50.6          Avg. T2:      100.7         Avg. T2:      104.1



ibmq_armonk                 ibmq_16_melbourne           ibmqx2
-----------                 -----------------           ------
Num. Qubits:  1             Num. Qubits:  15            Num. Qubits:  5
Pending Jobs: 1             Pending Jobs: 18            Pending Jobs: 1
Least busy:   False         Least busy:   False         Least busy:   False
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      169.7         Avg. T1:      54.6          Avg. T1:      58.6
Avg. T2:      261.0         Avg. T2:      53.1          Avg. T2:      35.8
```

Escolhemos o computador com menos erro.

In [19]:
```
backend_device = provider.get_backend('ibmq_santiago')
print("Running on: ", backend_device)
```

```
Running on:  ibmq_santiago
```

In [20]:
```
backend_device
```

Out[20]:
```
<IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open', project='main')>
```

In [21]:
```
backend_monitor(backend_device)
```

```
ibmq_santiago
=============
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 0
    backend_version: 1.3.22
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    online_date: 2020-06-03 04:00:00+00:00
    qubit_lo_range: [[4.333429063663208e+18, 5.333429063663208e+18], [4.123616111887707e+18, 5.123616
111887708e+18], [4.3205355364195937e+18, 5.3205355364195937e+18], [4.2423110151676165e+18, 5.242311015
167617e+18], [4.3163299835615 23e+18, 5.316329983561523e+18]]
    n_uchannels: 8
    meas_kernels: ['hw_qmfk']
    backend_name: ibmq_santiago
    dtm: 0.2222222222222222
    meas_levels: [1, 2]
    input_allowed: ['job']
    meas_map: [[0, 1, 2, 3, 4]]
    rep_times: [0.001]
    hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In this case, the system
includes higher energy states, i.e. not just |0> and |1>. The Pauli operators are generalized via the
following set of transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv b^\\dagg
er_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-} \\rightarrow b$,\n\n$\\sigma
_{i}^X \\rightarrow b^\\dagger_{i} + b_{i}$.\n\nQubits are coupled through resonator buses. The provi
ded Hamiltonian has been projected into the zero excitation subspace of the resonator buses leading t
o an effective qubit-qubit flip-flop interaction. The qubit resonance frequencies in the Hamiltonian
are the cavity dressed frequencies and not exactly what is returned by the backend defaults, which al
so includes the dressing due to the qubit-qubit interactions.\n\nQuantities are returned in angular f
requencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system Hamiltonian information is avai
lable to the public, missing values have been replaced with 0.\n', 'h_latex': '\\begin{align} \\mathc
al{H}/\\hbar = & \\sum_{i=0}^{4}\\left(\\frac{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\D
elta_{i}}{2}(O_i^2-O_i)+\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ & + J_{0,1}(\\sigma_{0}^{+}\\si
gma_{1}^{-}+\\sigma_{0}^{-}\\sigma_{1}^{+}) + J_{3,4}(\\sigma_{3}^{+}\\sigma_{4}^{-}+\\sigma_{3}^{-}
\\sigma_{4}^{+}) + J_{2,3}(\\sigma_{2}^{+}\\sigma_{3}^{-}+\\sigma_{2}^{-}\\sigma_{3}^{+}) + J_{1,2}
(\\sigma_{1}^{+}\\sigma_{2}^{-}+\\sigma_{1}^{-}\\sigma_{2}^{+}) \\\\ & + \\Omega_{d,0}(U_{0}^{(0,1)}
(t))\\sigma_{0}^{X} + \\Omega_{d,1}(U_{1}^{(1,0)}(t)+U_{2}^{(1,2)}(t))\\sigma_{1}^{X} \\\\ & + \\Omeg
a_{d,2}(U_{3}^{(2,1)}(t)+U_{4}^{(2,3)}(t))\\sigma_{2}^{X} + \\Omega_{d,3}(U_{6}^{(3,4)}(t)+U_{5}^{(3,
2)}(t))\\sigma_{3}^{X} \\\\ & + \\Omega_{d,4}(U_{7}^{(4,3)}(t))\\sigma_{4}^{X} \\\\ \\end{align}', 'h
_str': ['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]', '_SUM[i,0,4,delta{i}/2*O{i}*O{i}]', '_SUM[i,0,4,-delta{i}/
2*O{i}]', '_SUM[i,0,4,omegad{i}*X{i}||D{i}]', 'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1', 'jq3q4*Sp3*Sm4', 'jq3
q4*Sm3*Sp4', 'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3', 'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2', 'omegad1*X0||U0', 'o
megad0*X1||U1', 'omegad2*X1||U2', 'omegad1*X2||U3', 'omegad3*X2||U4', 'omegad4*X3||U6', 'omegad2*X3||
U5', 'omegad3*X4||U7'], 'osc': {}, 'qub': {'0': 3, '1': 3, '2': 3, '3': 3, '4': 3}, 'vars': {'delta
0': -2.1481278490714906, 'delta1': -2.0623435150768743, 'delta2': -2.1429828509850863, 'delta3': -2.1
37118237032298, 'delta4': -2.154596484455155, 'jq0q1': 0.007378105608801839, 'jq1q2': 0.0072687006787
58498, 'jq2q3': 0.007255936195908655, 'jq3q4': 0.006881064755295536, 'omegad0': 1.0104658648016156,
'omegad1': 0.985750533807121, 'omegad2': 1.0014620722905927, 'omegad3': 1.0069291377711964, 'omegad
4': 1.0007250287472587, 'wq0': 30.369330476103457, 'wq1': 29.051036820251653, 'wq2': 30.2883180551686
56, 'wq3': 29.796818892577082, 'wq4': 30.261893787242265}}
    conditional: False
    rep_delay_range: [0.0, 500.0]
    conditional_latency: []
    max_shots: 8192
    pulse_num_channels: 9
    open_pulse: False
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    max_experiments: 75
    processor_type: {'family': 'Falcon', 'revision': 4, 'segment': 'L'}
    credits_required: True
    supported_instructions: ['u2', 'measure', 'u1', 'delay', 'acquire', 'reset', 'sx', 'shiftf', 'i
d', 'u3', 'setf', 'x', 'cx', 'play', 'rz']
    dt: 0.2222222222222222
    allow_object_storage: True
    qubit_channel_mapping: [['d0', 'm0', 'u0', 'u1'], ['u2', 'u1', 'u0', 'm1', 'u3', 'd1'], ['u2', 'u
5', 'u3', 'u4', 'd2', 'm2'], ['m3', 'u5', 'd3', 'u7', 'u4', 'u6'], ['d4', 'u7', 'm4', 'u6']]
    pulse_num_qubits: 3
    discriminators: ['hw_qmfk', 'quadratic_discriminator', 'linear_discriminator']
    default_rep_delay: 250.0
    dynamic_reprate_enabled: True
    uchannels_enabled: True
    multi_meas_enabled: True
    url: None
    description: 5 qubit device
    sample_name: family: Falcon, revision: 4, segment: L
    coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4, 3]]
    acquisition_latency: []
    n_registers: 1
    channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire', 'type': 'acquire'}, 'a
cquire1': {'operates': {'qubits': [1]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire2': {'opera
tes': {'qubits': [2]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits':
[3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates': {'qubits': [4]}, 'purpose':
'acquire', 'type': 'acquire'}, 'd0': {'operates': {'qubits': [0]}, 'purpose': 'drive', 'type': 'driv
e'}, 'd1': {'operates': {'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates': {'q
ubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates': {'qubits': [3]}, 'purpose': 'd
rive', 'type': 'drive'}, 'd4': {'operates': {'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm
0': {'operates': {'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates': {'qubi
ts': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates': {'qubits': [2]}, 'purpose':
'measure', 'type': 'measure'}, 'm3': {'operates': {'qubits': [3]}, 'purpose': 'measure', 'type': 'mea
sure'}, 'm4': {'operates': {'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operate
s': {'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1': {'operates': {'qubit
s': [1, 0]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u2': {'operates': {'qubits': [1, 2]},
'purpose': 'cross-resonance', 'type': 'control'}, 'u3': {'operates': {'qubits': [2, 1]}, 'purpose':
'cross-resonance', 'type': 'control'}, 'u4': {'operates': {'qubits': [2, 3]}, 'purpose': 'cross-reson
ance', 'type': 'control'}, 'u5': {'operates': {'qubits': [3, 2]}, 'purpose': 'cross-resonance', 'typ
e': 'control'}, 'u6': {'operates': {'qubits': [3, 4]}, 'purpose': 'cross-resonance', 'type': 'contro
l'}, 'u7': {'operates': {'qubits': [4, 3]}, 'purpose': 'cross-resonance', 'type': 'control'}}
    u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 0, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0
j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}], [{'q': 2, 'scale': (1+0j)}], [{'q':
4, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}]]
    quantum_volume: 32
    allow_q_object: True
    memory: True
    meas_lo_range: [[6.952624018e+18, 7.952624018e+18], [6.701014434e+18, 7.701014434e+18], [6.837332
258e+18, 7.837332258e+18], [6.901770712e+18, 7.901770712e+18], [6.775814414e+18, 7.775814414e+18]]

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
-----------------------------------------------------------------------
    Q0 / 4.83343 GHz / 135.71038 us / 104.86319 us / 0.00000 / 0.00022 / 0.00022 / 0.01860
    Q1 / 4.62362 GHz / 88.97324 us / 95.24654 us / 0.00000 / 0.00019 / 0.00019 / 0.01100
    Q2 / 4.82054 GHz / 95.30146 us / 83.43965 us / 0.00000 / 0.00025 / 0.00025 / 0.00950
    Q3 / 4.74231 GHz / 137.77549 us / 92.50995 us / 0.00000 / 0.00021 / 0.00021 / 0.00580
    Q4 / 4.81633 GHz / 95.62009 us / 127.30675 us / 0.00000 / 0.00058 / 0.00058 / 0.02190

Multi-Qubit Gates [Name / Type / Gate Error]
---------------------------------------------
    cx4_3 / cx / 0.00619
    cx3_4 / cx / 0.00619
    cx2_3 / cx / 0.00676
    cx3_2 / cx / 0.00676
    cx2_1 / cx / 0.00562
    cx1_2 / cx / 0.00562
    cx0_1 / cx / 0.00651
    cx1_0 / cx / 0.00651
```

In [22]:
```
from qiskit.providers.aer.noise import NoiseModel
```

In [23]:
```
coupling_map = backend_device.configuration().coupling_map
```

In [24]:
```
noise_model = NoiseModel.from_backend(backend_device)
print(noise_model)
```

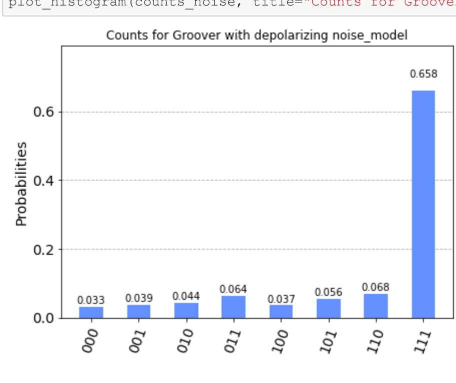
```
NoiseModel:
  Basis gates: ['cx', 'id', 'reset', 'rz', 'sx', 'x']
  Instructions with noise: ['measure', 'sx', 'cx', 'reset', 'id', 'x']
  Qubits with noise: [0, 1, 2, 3, 4]
  Specific qubit errors: [('id', [0]), ('id', [1]), ('id', [2]), ('id', [3]), ('id', [4]), ('sx',
[0]), ('sx', [1]), ('sx', [2]), ('sx', [3]), ('sx', [4]), ('x', [0]), ('x', [1]), ('x', [2]), ('x',
[3]), ('x', [4]), ('cx', [4, 3]), ('cx', [3, 4]), ('cx', [2, 3]), ('cx', [3, 2]), ('cx', [2, 1]), ('c
x', [1, 2]), ('cx', [0, 1]), ('cx', [1, 0]), ('reset', [0]), ('reset', [1]), ('reset', [2]), ('rese
t', [3]), ('reset', [4]), ('measure', [0]), ('measure', [1]), ('measure', [2]), ('measure', [3]), ('m
easure', [4])]
```

In [25]:
```
basis_gates = noise_model.basis_gates
print(basis_gates)
```

```
['cx', 'id', 'reset', 'rz', 'sx', 'x']
```

In [26]:
```
result_noise = execute(qc_Grover, backend,
noise_model=noise_model,
coupling_map=coupling_map,
basis_gates=basis_gates).result()
counts_noise = result_noise.get_counts(qc_Grover)
plot_histogram(counts_noise, title="Counts for Groover with depolarizing noise_model")
```

Out[26]:



Simulação num computador quântico

In [27]:
```
%qiskit_job_watcher
```

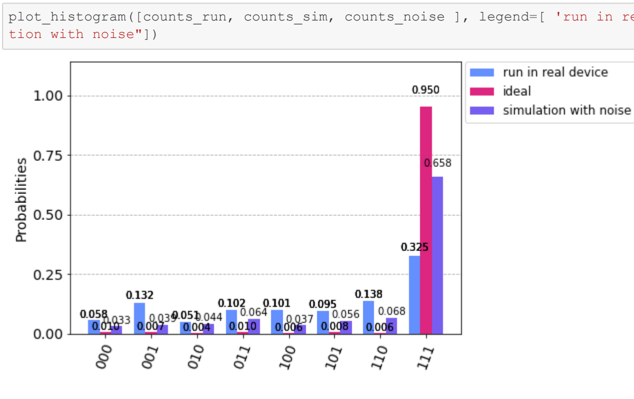
In [28]:
```
job_r = execute(qc_Grover, backend_device, shots=shots)

jobID_r = job_r.job_id()

print('JOB ID: {}'.format(jobID_r))
```

```
JOB ID: 60bcd49cb454d093beaa7409
```

In [29]:
```
job_get=backend_device.retrieve_job("60bcd49cb454d093beaa7409")

result_r = job_get.result()
counts_run = result_r.get_counts(qc_Grover)
```

In [30]:
```
plot_histogram([counts_run, counts_sim, counts_noise ], legend=[ 'run in real device', 'ideal', "simula
tion with noise"])
```

Out[30]:

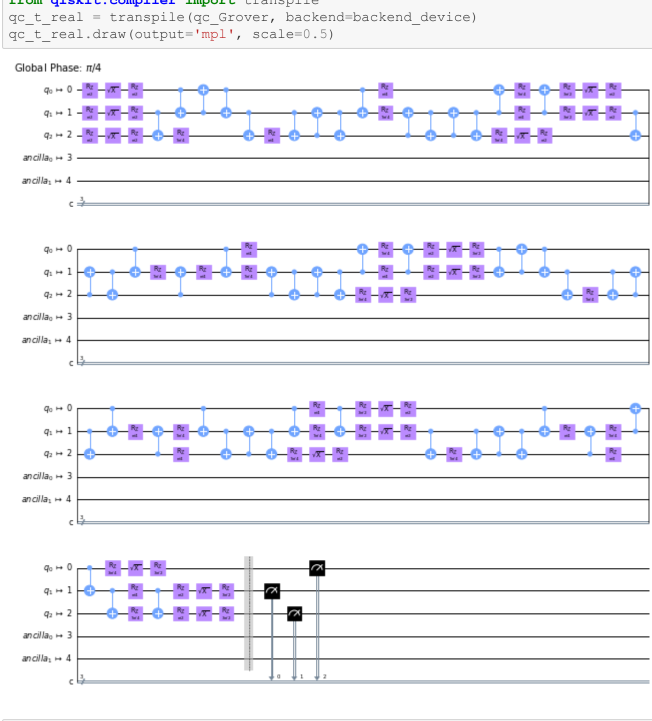


Optimisation

Decidimos fazer uma optimização de nível 3.

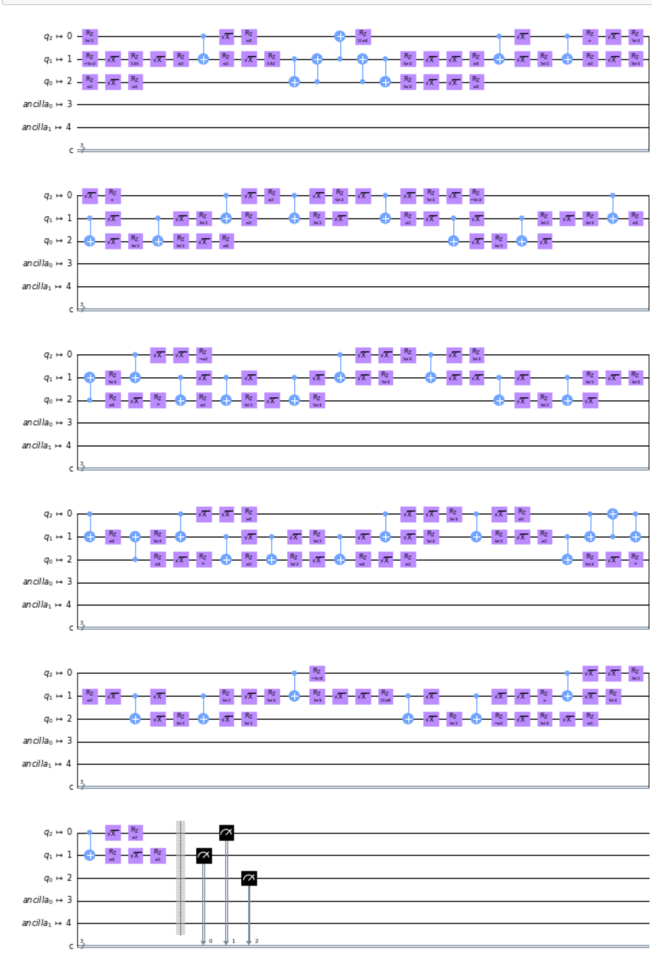
In [31]:
```
from qiskit.compiler import transpile
qc_t_real = transpile(qc_Grover, backend=backend_device)
qc_t_real.draw(output='mpl', scale=0.5)
```

Out[31]:



In [32]:
```
qc_optimized_3 = transpile(qc_Grover, backend=backend_device,optimization_level=3)
qc_optimized_3.draw(output='mpl', scale=0.5)
```

Out[32]:



In [33]:
```
qc_Grover.depth()
```

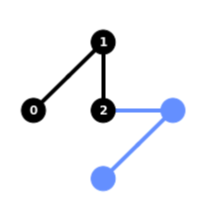
Out[33]:
```
22
```

In [34]:
```
qc_t_real.depth()
```

Out[34]:
```
83
```

In [35]:
```
from qiskit.visualization import plot_circuit_layout
plot_circuit_layout(qc_t_real, backend_device)
```

Out[35]:

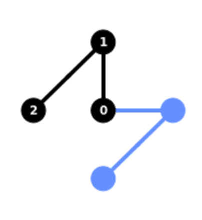


In [36]:
```
qc_optimized_3.depth()
```

Out[36]:
```
130
```

In [37]:
```
from qiskit.visualization import plot_circuit_layout
plot_circuit_layout(qc_optimized_3, backend_device)
```

Out[37]:



In [38]:
```
job_exp = execute(qc_optimized_3, backend_device, shots = shots)
jobID = job_exp.job_id()
print('JOB ID: {}'.format(jobID))
job_exp.result().get_counts(qc_optimized_3)
```

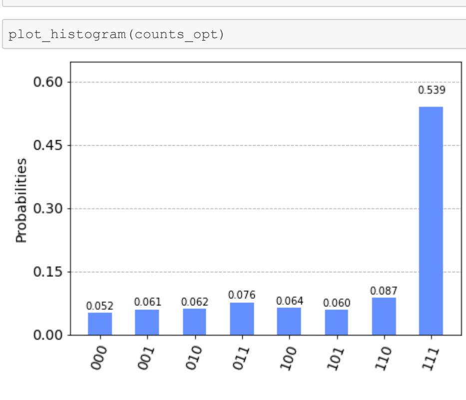
```
JOB ID: 60bcd4c8fe8ff1433b29f5dd
```

Out[38]:
```
{'000': 53,
 '001': 62,
 '010': 63,
 '011': 78,
 '100': 66,
 '101': 61,
 '110': 89,
 '111': 552}
```

In [39]:
```
job_get_o =backend_device.retrieve_job("60bcd4c8fe8ff1433b29f5dd")
result_opt = job_get_o.result(timeout=3600, wait=5)
counts_opt = result_opt.get_counts(qc_optimized_3)
```

In [40]:
```
plot_histogram(counts_opt)
```

Out[40]:



Ignis

Vamos agora usar o Ignis para a mitigação de erros.

In [41]:
```
from qiskit.ignis.mitigation.measurement import (complete_meas_cal, tensored_meas_cal,
                                                 CompleteMeasFitter, TensoredMeasFitter)
```

Começamos por gerar a lista dos circuitos de calibração de medição.
Como estamos a medir 3 qubits vamos precisar de $2^3 = 8$ circuitos de calibração.

In [42]:
```
qr1 = QuantumRegister(n)
meas_calibs, state_labels = complete_meas_cal(qubit_list=[0,1,2], qr=qr, circlabel='mcal')
```

In [43]:
```
state_labels
```

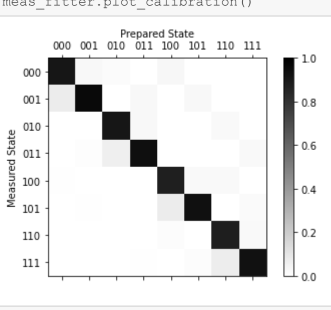
Out[43]:
```
['000', '001', '010', '011', '100', '101', '110', '111']
```

In [44]:
```
job_ignis = execute(meas_calibs, backend=backend_device, shots=shots)
jobID_run_ignis = job_ignis.job_id()
print('JOB ID: {}'.format(jobID_run_ignis))
```

```
JOB ID: 60bcd50025cc6e360365d32e
```

In [45]:
```
job_get=backend_device.retrieve_job("60bcd50025cc6e360365d32e")
cal_results = job_get.result()
```

In [46]:
```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')
meas_fitter.plot_calibration()
```



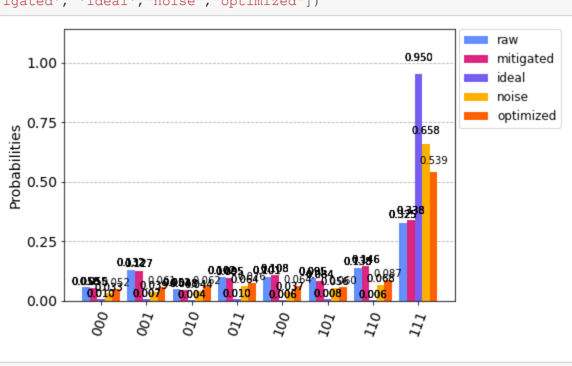
In [47]:
```
print("Average Measurement Fidelity: %f" % meas_fitter.readout_fidelity())
```

```
Average Measurement Fidelity: 0.917114
```

In [48]:
```
meas_filter = meas_fitter.filter
# Results with mitigation
mitigated_results = meas_filter.apply(result_r)
mitigated_counts = mitigated_results.get_counts()
```

In [49]:
```
plot_histogram([counts_run, mitigated_counts, counts_sim, counts_noise,counts_opt], legend=['raw', 'mitigated', 'ideal',"noise","optimized"])
```

Out[49]:



In [ ]: