Practica4_APCC

May 8, 2021

1 La QFT y sumadores cuánticos

Mario Quiñones Pérez

2 Forest

[1]:
```
from pyquil import get_qc, Program
from pyquil.api import QVMConnection
from pyquil.gates import *
from pyquil import latex
from pyquil.api import local_forest_runtime
from pyquil import get_qc, Program
```

Para la creación de circuitos se usará la función latex.to_latex(Program()), pero al crear esta función un string que pasar a TexWorks, solo se usará una vez como ejemplo para no ensuciar la memoria.

2.1 Algoritmo de Deutsch

El problema trata de que se nos da un circuito (Oráculo) que implementa una función booleana de un bit y hay que determinar si la función es constante(mismo valor para todas las entradas) o balanceada (1 en una entrada y 0 en la otra).

En el escenario clásico, necesitaríamos consultar esta caja negra dos veces, para calcular ambos valores de la función midiendo f(0) y luego f(1).

En el escenario cuántico, podemos hacerlo con solo una evaluación, en superposición, y utilizando el entrelazamiento y al final interferencia.

Este circuito calcula, de forma reversible (algo que se debe cumplir para todo circuito cuántico), una cierta función f (en este caso, de una sola entrada)

```
Si la función es constante, mediremos "0"
Si la función es balanceada mediremos "1"
```

El algoritmo de Deutsch explota un fenómeno de interferencia similar al encontrado en algunos experimentos físicos. Por ejemplo el experimento de la doble rendija o Interferómetro de Mach-Zehnder.

Creamos los siguientes oraculos para la prueba de el algoritmo de Deutsch

[2]:
```
def NOT(p):
    p += X(1)
```

[3]:
```
def CNOTs(p):
    p += CNOT(0,1)
```

[4]:
```
def NOTCNOT(p):
    p += X(0)
    p += CNOT(0,1)
    p += X(0)
```

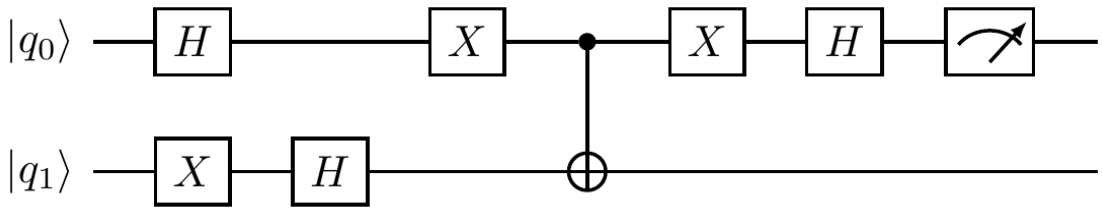
Creamos el circuito para la comprobación de un oráculo, en este añadimos un qubit auxiliar que inicializamos a |1> y pondremos ambos eb superposición pasandolos al estado |+> y |-> con las puertas Hadamard. Pasaremos estos qubits por el oráculo y tras este paso deshacemos la superposición y medimos el qubit q0 que es el que nos interesa

Este será la única parte de la memoria en la que se mostrará el código generado por la función latex.to_latex(p).

[5]:
```
p = Program()
ro = p.declare('ro', 'BIT', 1)

p += X(1)
p += H(0)
p += H(1)

p += FENCE()

NOTCNOT(p)

p += FENCE()

p += H(0)
p += MEASURE(0, ro[0])

print(latex.to_latex(p))
print(p)
```

```
\documentclass[convert={density=300,outext=.png}]{standalone}
\usepackage[margin=1in]{geometry}
\usepackage{tikz}
\usetikzlibrary{quantikz}
\begin{document}
\begin{tikzcd}
\lstick{\ket{q_{0}}} & \gate{H} & \gate{X} & \ctrl{1} & \gate{X} & \gate{H} &
\meter{} & \qw \\
\lstick{\ket{q_{1}}} & \gate{X} & \gate{H} & \targ{} & \qw & \qw & \qw & \qw
\end{tikzcd}
\end{document}
```

```
DECLARE ro BIT[1]
X 1
H 0
H 1
FENCE
X 0
CNOT 0 1
X 0
FENCE
H 0
MEASURE 0 ro[0]
```



2.1.1 Simulaciones

Vemos que para la simulación con un oraculo balanceado como "NOTCNOT" la salida sale 1 mientras que para un oraculo constante como la identidad la salida sale siempre 0

[11]:
```
p = Program()
ro = p.declare('ro', 'BIT', 1)

p += X(1)
p += H(0)
p += H(1)


p += H(0)
p += MEASURE(0, ro[0])


qc = get_qc('2q-qvm')
executable = qc.compile(p)
result = qc.run(executable)

print(result)
```

```
[[0]]
```

[12]:
```
p = Program()
ro = p.declare('ro', 'BIT', 1)
```

```
p += X(1)
p += H(0)
p += H(1)


NOTCNOT(p)


p += H(0)
p += MEASURE(0, ro[0])


qc = get_qc('2q-qvm')
executable = qc.compile(p)
result = qc.run(executable)

print(result)
```

```
[[1]]
```

2.2 Algoritmo de Deutsch-Jozsa

El algoritmo Deutsch-Jozsa resuelve un tipo de problema llamados de consulta, oráculo o de promesa.

Se nos da una función booleana f de más de un parámetro y se nos promete que es constante o balanceado (0 para la mitad de las entradas y 1 para el resto). Tenemos que averiguar cual de las dos es llamando a la función el menor número de veces posible.

Mientras que con un algoritmo determinista clásico necesitamos (en el peor caso) 2(n-1) + 1 llamadas a f, con el algoritmo cuántico Deutsch-Jozsa es suficiente evaluarlo una vez.

1. Creamos el estado | (0^n) 1 >
2. Usamos H para crear superposición
3. Aplicamos el oráculo
4. Aplicamos de nuevo H a los n primeros qbits
5. Medimos los n primeros qbits

Si la función es constante, obtendremos 0 y si es balanceada, obtendremos una cadena distinta.

Utilizaremos la función prepare para preparar un circuito a partir de una cantidad de qubits cualesquiera “n” en un circuito “p”. Se añadirá un qubit auxiliar inicializado a uno y se pondrán todos en superposición.

[8]:
```
def prepare(p, n):
    p += X(n)

    for i in range(n + 1):
        p += H(i)
```

```
    return p
```

Utilizaremos la función end quitar los n primeros qubits del circuito de superposición y añadir mediciones a todos los qubits menos al último.

[9]:
```
def end(p, n, ro):

    for i in range(n):
        p += H(i)

    lista = list(range(n))

    for i, q in enumerate(lista):
        p += MEASURE(q, ro[i])
```

He creado una serie de oráculos basándose en ejercicios dados y aquellos ya mostrados en clase.

[10]:
```
def ejercicio1(p, n):
    for i in range(n):
        p += CNOT(i, n)
        p += X(i)
        p += CNOT(i, n)
```

[11]:
```
def ejercicio2(p, n):
        p += CNOT(2, n)
        p += CNOT(1, n)
        p += CCNOT(0, 1, n)
```

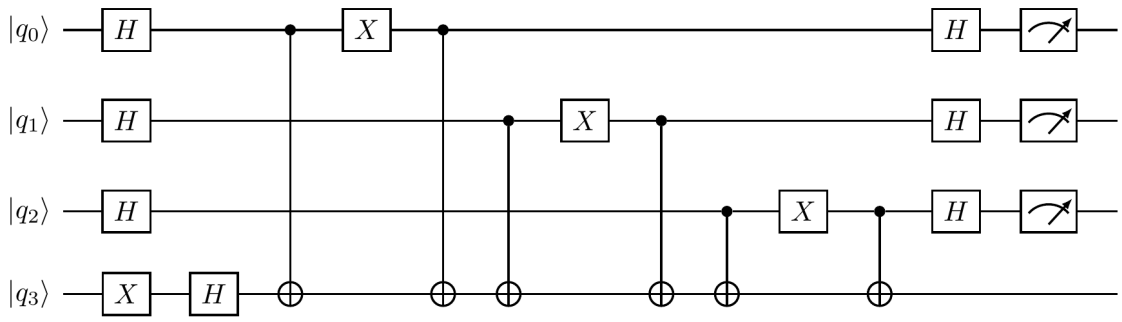
[12]:
```
def cnots(p, n):
    for i in range(n):
        p += CNOT(i, n)
```

[13]:
```
def Rcnots(p, n):
    for i in range(n):
        p += CNOT((n-1) - i,n)
```

[14]:
```
n = 3
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

p += FENCE()

ejercicio1(p, n)

p += FENCE()
```

```
end(p, n, ro)

print(p)
```

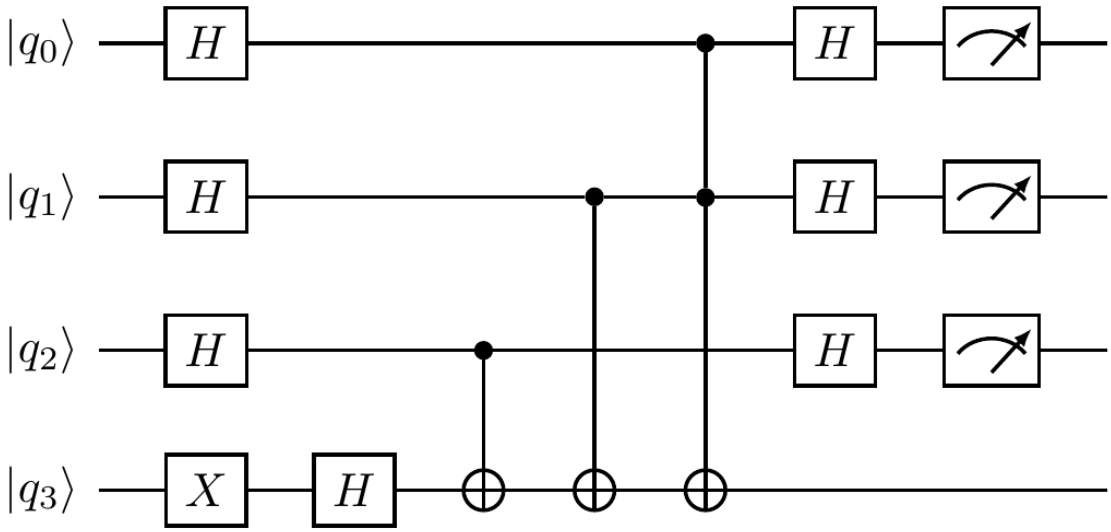
```
DECLARE ro BIT[3]
X 3
H 0
H 1
H 2
H 3
FENCE
CNOT 0 3
X 0
CNOT 0 3
CNOT 1 3
X 1
CNOT 1 3
CNOT 2 3
X 2
CNOT 2 3
FENCE
H 0
H 1
H 2
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
```



[15]:
```
n = 3
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

p += FENCE()
```

```
ejercicio2(p, n)

p += FENCE()

end(p, n, ro)

print(p)
```

```
DECLARE ro BIT[3]
X 3
H 0
H 1
H 2
H 3
FENCE
CNOT 2 3
CNOT 1 3
CCNOT 0 1 3
FENCE
H 0
H 1
H 2
MEASURE 0 ro[0]
MEASURE 1 ro[1]
MEASURE 2 ro[2]
```



2.2.1 Simulaciones

Como podemos comprobar en las siguientes ejecuciones, las funciones cuya salida para todos los qubits es siempre |0> son constantes, mientras que aquellas cuya salida es distinta de este valor son balanceadas.

[16]:
```
n = 4
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

ejercicio1(p, n)

end(p, n, ro)

qc = get_qc('5q-qvm')
executable = qc.compile(p)
result = qc.run(executable)


print(result)
```

```
[[0 0 0 0]]
```

[17]:
```
n = 4
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

ejercicio2(p, n)

end(p, n, ro)

qc = get_qc('5q-qvm')
executable = qc.compile(p)
result = qc.run(executable)


print(result)
```

```
[[1 0 1 0]]
```

[18]:
```
n = 4
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

cnots(p, n)

end(p, n, ro)
```

```
qc = get_qc('5q-qvm')
executable = qc.compile(p)
result = qc.run(executable)


print(result)
```

```
[[1 1 1 1]]
```

[27]:
```
n = 4
p = Program()
ro = p.declare('ro', 'BIT', n)

prepare(p, n)

Rcnots(p, n)

end(p, n, ro)

qc = get_qc('5q-qvm')
executable = qc.compile(p)
result = qc.run(executable)


print(result)
```

```
[[1 1 1 1]]
```

3 Qiskit

Ahora realizaremos el mismo procedimiento pero con qiskit. Se evitarán repetición de explicaciones para hacer más amena la memoria

[1]:
```
from qiskit import QuantumCircuit
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram
from qiskit.extensions import Initialize
import numpy as np
```

3.1 Algoritmo de Deutsch

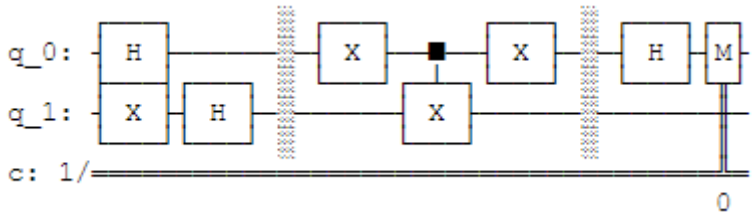
[2]:
```
def I(qc):
    return
```

[3]:
```
def CNOT(qc):
    qc.cx(0,1)
```

[4]:
```
def NOT(qc):
    qc.x(1)
```

[5]:
```
def NOTCNOT(qc):
    qc.x(0)
    qc.cx(0,1)
    qc.x(0)
```

[6]:
```
qc = QuantumCircuit(2,1)
qc.x(1)
qc.h(0)
qc.h(1)

qc.barrier()

NOTCNOT(qc)

qc.barrier()

qc.h(0)
qc.measure([0], [0])
qc.draw()
```

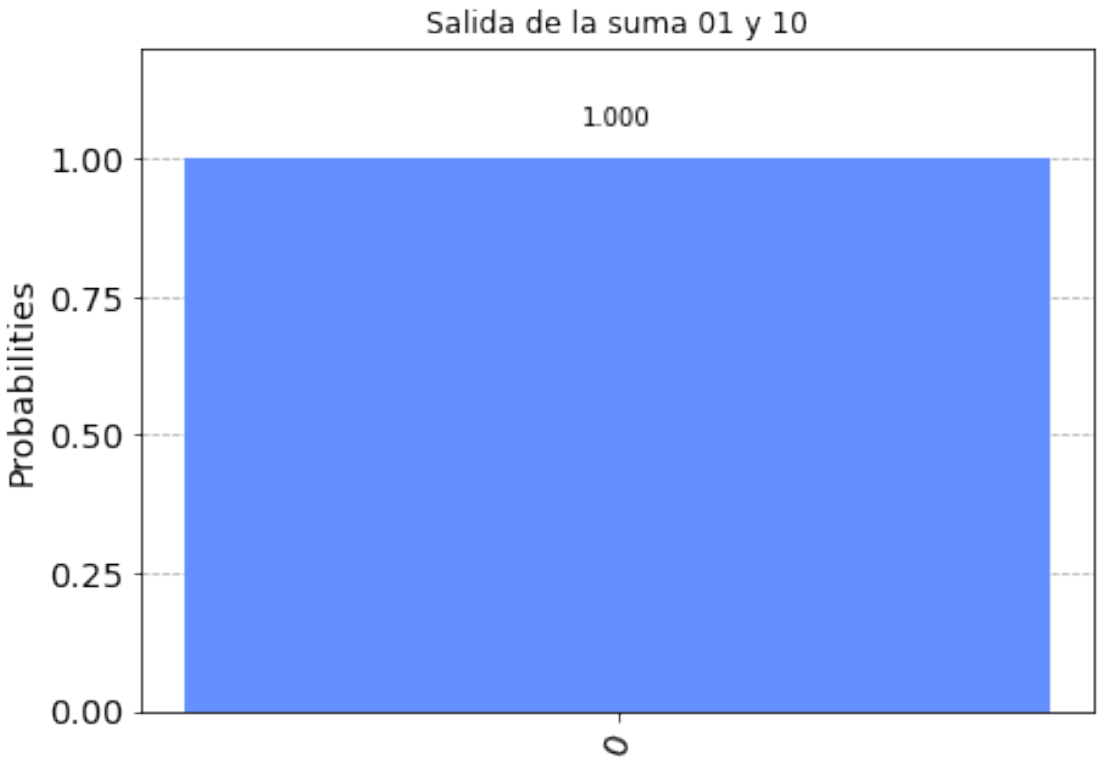


3.1.1 Simulaciones

[7]:
```
qc = QuantumCircuit(2,1)
qc.x(1)
qc.h(0)
qc.h(1)

qc.barrier()

NOT(qc)

qc.barrier()

qc.h(0)
qc.measure([0], [0])
```

```
# Simulamos el circuito 1000 veces y mostramos el resultado en un histograma
qasm_sim = Aer.get_backend('qasm_simulator')
qc = transpile(qc, qasm_sim)
qobj = assemble(qc)


result = qasm_sim.run(qc).result()

counts = result.get_counts(qc)
plot_histogram(counts, title='Salida de la suma 01 y 10')
```

[7]:



3.2 Algoritmo de Deutsch-Jozsa

[8]:
```
def prepare(n):
    qc = QuantumCircuit(n + 1,n)
    qc.x(n)

    for i in range(n + 1):
        qc.h(i)

    return qc
```

[9]:
```
def end(qc, n):

    for i in range(n):
        qc.h(i)

    lista = list(range(n))
    qc.measure(lista,lista[::-1])
```

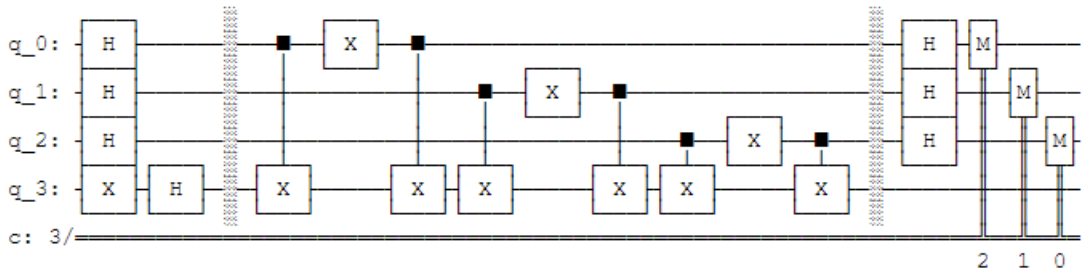
[10]:
```
def ejercicio1(qc, n):
    for i in range(n):
        qc.cx(i,n)
        qc.x(i)
        qc.cx(i,n)
```

[11]:
```
def cnots(qc, n):
    for i in range(n):
        qc.cx(i,n)
```

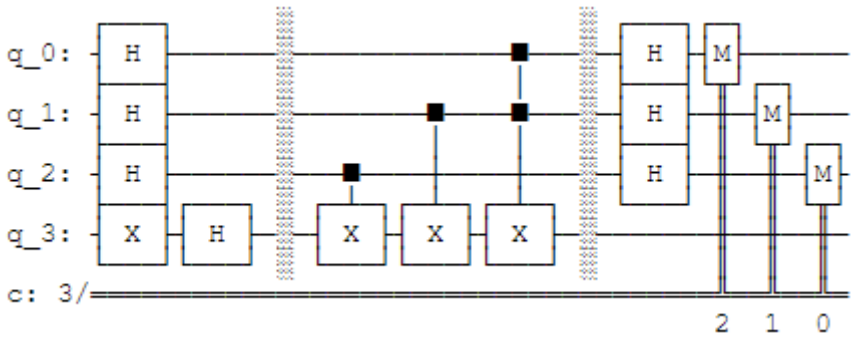
[12]:
```
def Rcnots(qc, n):
    for i in range(n):
        qc.cx((n-1) - i,n)
```

[13]:
```
def ejercicio2(qc, n):
        qc.cx(2,n)
        qc.cx(1,n)
        qc.ccx(0,1,n)
```

[14]:
```
n = 3

qc = prepare(n)

qc.barrier()

ejercicio1(qc, n)

qc.barrier()

end(qc, n)

qc.draw()
```



[15]:
```
n = 3

qc = prepare(n)

qc.barrier()

ejercicio2(qc, n)

qc.barrier()

end(qc, n)

qc.draw()
```



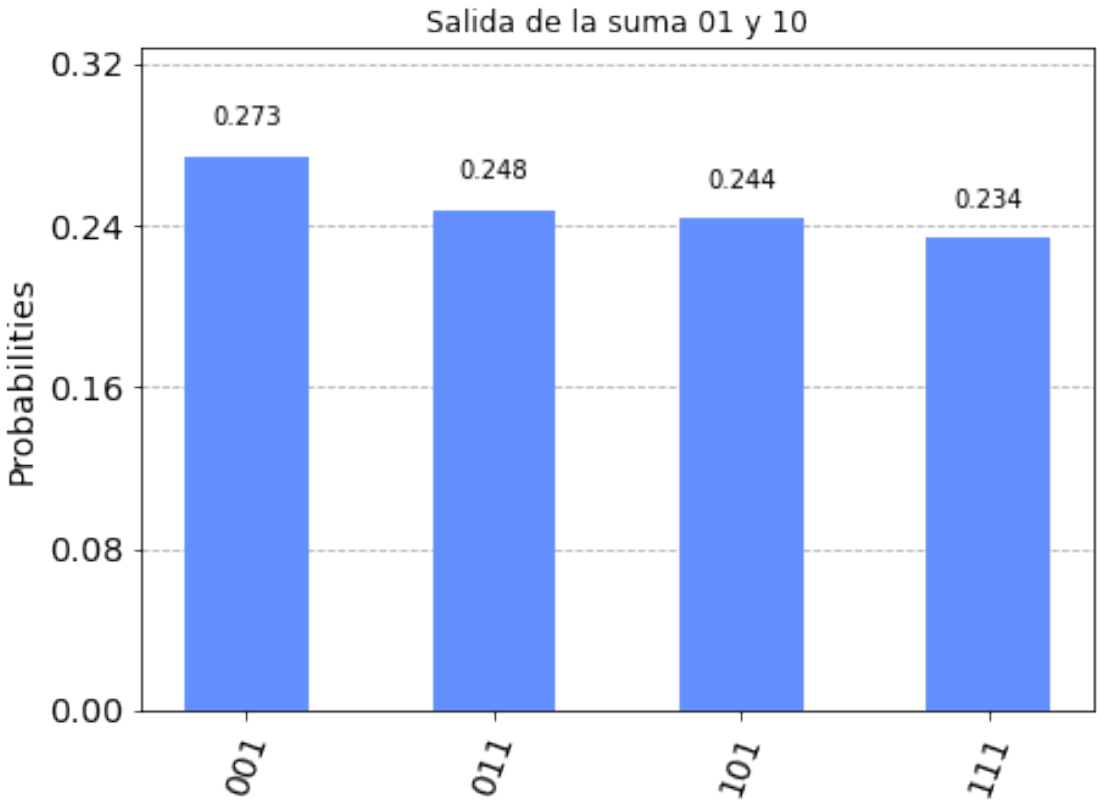
3.2.1 Simulaciones

Solo en aquellos oraculos cuya salida es constante su salida es siempre |0>

[16]:
```
n = 3

qc = prepare(n)

qc.barrier()

ejercicio2(qc, n)

qc.barrier()
```

```
end(qc, n)

# Simulamos el circuito 1000 veces y mostramos el resultado en un histograma
qasm_sim = Aer.get_backend('qasm_simulator')
qc = transpile(qc, qasm_sim)
qobj = assemble(qc)


result = qasm_sim.run(qc).result()

counts = result.get_counts(qc)
plot_histogram(counts, title='Salida de la suma 01 y 10')
```

[16]:

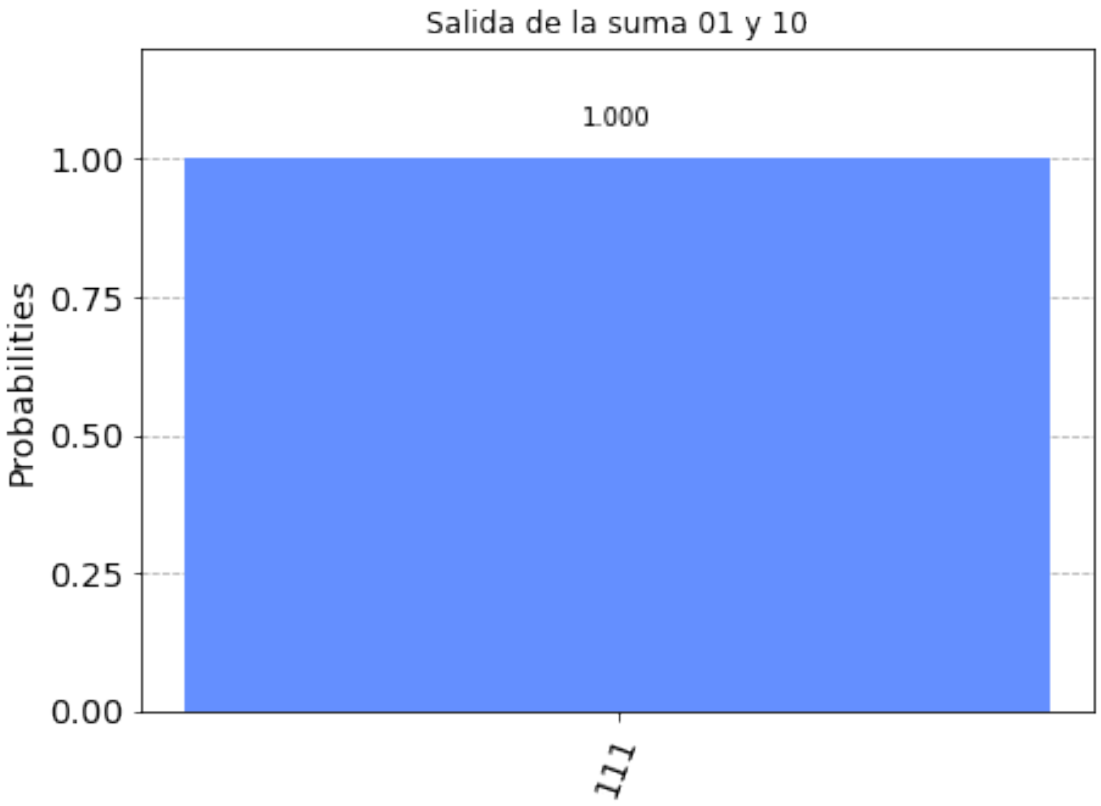


[17]:

```
n = 3

qc = prepare(n)

qc.barrier()

cnots(qc, n)
```

```
qc.barrier()

end(qc, n)

# Simulamos el circuito 1000 veces y mostramos el resultado en un histograma
qasm_sim = Aer.get_backend('qasm_simulator')
qc = transpile(qc, qasm_sim)
qobj = assemble(qc)


result = qasm_sim.run(qc).result()

counts = result.get_counts(qc)
plot_histogram(counts, title='Salida de la suma 01 y 10')
```

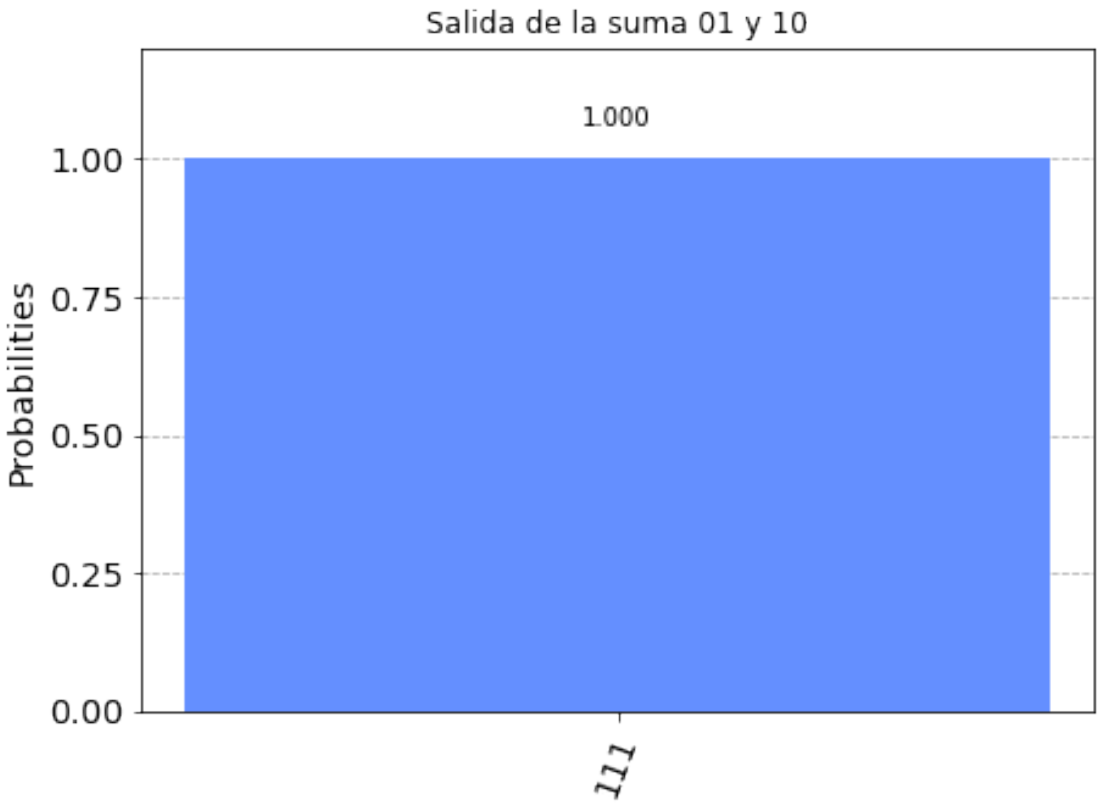
[17]:



[18]:
```
n = 3

qc = prepare(n)

qc.barrier()
```

```
Rcnots(qc, n)

qc.barrier()

end(qc, n)

# Simulamos el circuito 1000 veces y mostramos el resultado en un histograma
qasm_sim = Aer.get_backend('qasm_simulator')
qc = transpile(qc, qasm_sim)
qobj = assemble(qc)


result = qasm_sim.run(qc).result()

counts = result.get_counts(qc)
plot_histogram(counts, title='Salida de la suma 01 y 10')
```

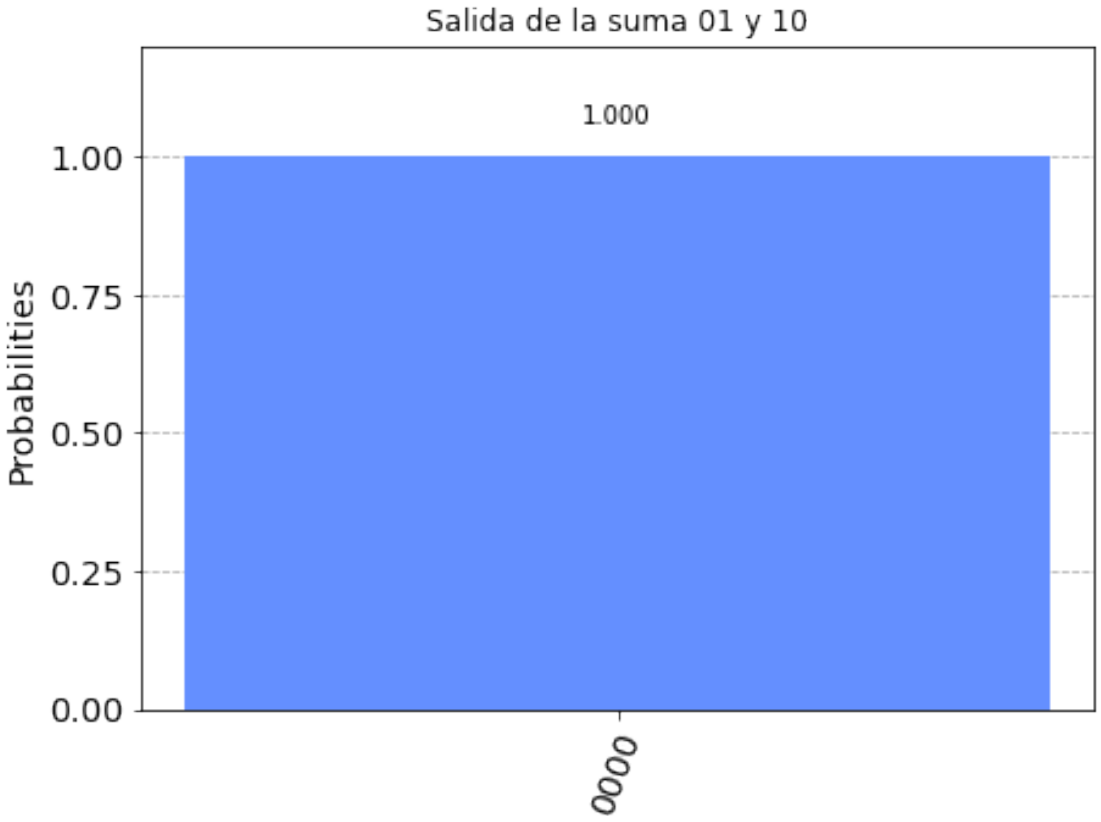
[18]:



[19]:
```
n = 4

qc = prepare(n)
```

```
qc.barrier()

ejercicio1(qc, n)

qc.barrier()

end(qc, n)

# Simulamos el circuito 1000 veces y mostramos el resultado en un histograma
qasm_sim = Aer.get_backend('qasm_simulator')
qc = transpile(qc, qasm_sim)
qobj = assemble(qc)


result = qasm_sim.run(qc).result()

counts = result.get_counts(qc)
plot_histogram(counts, title='Salida de la suma 01 y 10')
```

[19]:



3.2.2 Ejecuciones en backends reales

Como podemos comprobar en estas simulaciones echas en backends reales, para un orácuo constante como ejercicio1() vemos que las salidas son mayoritariamente (66.7%) |0000>, aunque haya errores debido al ruido. Por otro lado en un oraculo no constante como Rcnots() este no es el caso habiendo |0000> un 0.5% de las veces y dominando la salida |1111>

[92]:
```
n = 4
qc = prepare(n)
qc.barrier()
ejercicio1(qc, n)
qc.barrier()
end(qc, n)
```

[93]:
```
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy
IBMQ.
 ↪save_account('55dbb1b5e08af7c7c6ec803a2770842c2f2e9b16b557e0a3bf6e5025a41316a162a1655901146
 ↪overwrite=True)
shots = 1024

# Load local account information
IBMQ.load_account()
# Get the least busy backend
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().
 ↪n_qubits >= 2
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
print("least busy backend: ", backend)
# Run our circuit
t_qc = transpile(qc, backend, optimization_level=3)
qobj = assemble(t_qc)
job = backend.run(qobj)
```

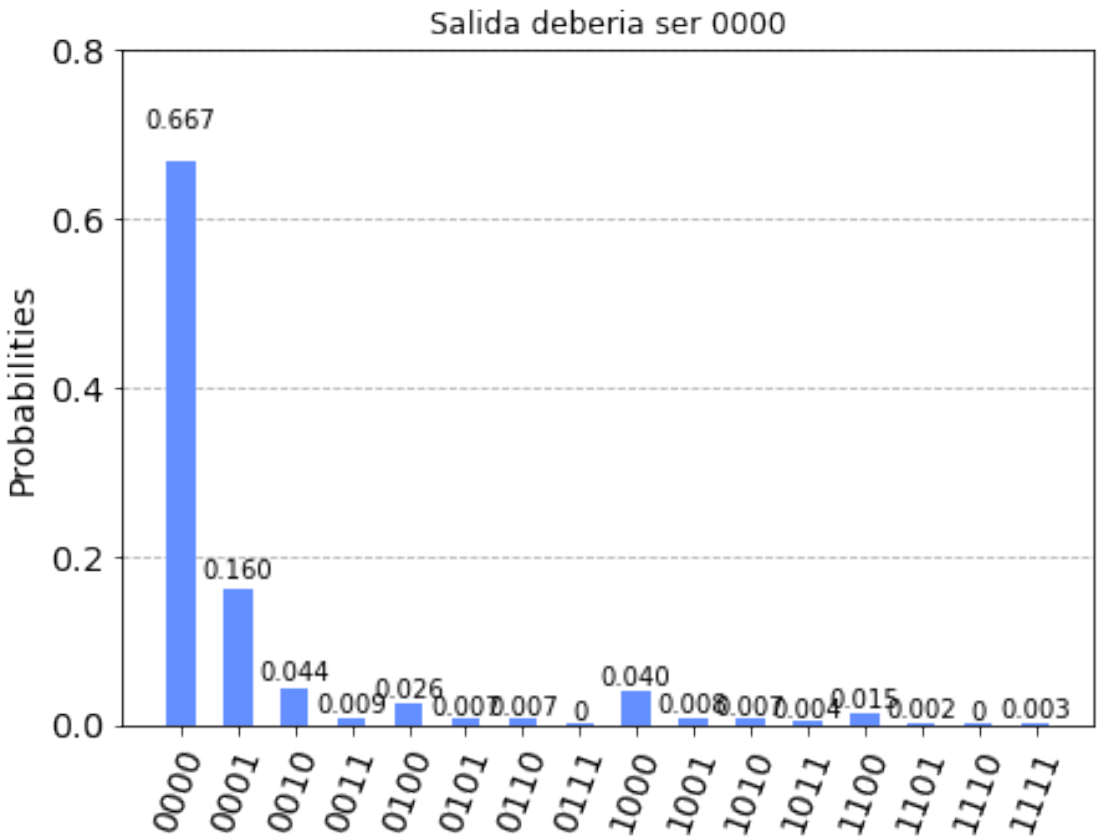
```
least busy backend:  ibmq_quito
```

[94]:
```
# Monitoring our job
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

```
Job Status: job has successfully run
```

[95]:
```
# Plotting our result
result = job.result()
plot_histogram(result.get_counts(qc), title='Salida deberia ser 0000')
```

[95]:



[83]:
```
n = 4
qc = prepare(n)
qc.barrier()
Rcnots(qc, n)
qc.barrier()
end(qc, n)
```

[84]:
```
# Load local account information
IBMQ.load_account()
# Get the least busy backend
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().
 ↪n_qubits >= 2
                                       and not x.configuration().simulator
                                       and x.status().operational==True))
print("least busy backend: ", backend)
# Run our circuit
t_qc = transpile(qc, backend, optimization_level=3)
qobj = assemble(t_qc)
job = backend.run(qobj)
```

```
least busy backend:  ibmq_quito
```

[85]:
```
# Monitoring our job
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

```
Job Status: job has successfully run
```

[88]:
```
# Plotting our result
result = job.result()
plot_histogram(result.get_counts(qc))
```

[88]: