QCP7010 : Introduction to Quantum Information Laboratory

Report on Part-A questions
Major Exam

Name:	**Thirumalai M**
Roll Number:	**M23IQT008**
Program:	**M.Tech- Quantum Technologies**

Chapter 1

Question A

1.1 A

Implement the following and find its output in IBM qiskit or any similar open source platform

1.1.1 i

C-NOT- Gate input- 00

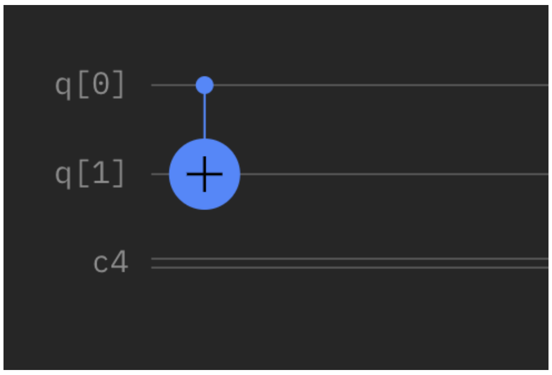

Figure 1.1: circuit diagram, 00

output for C-NOT - 00

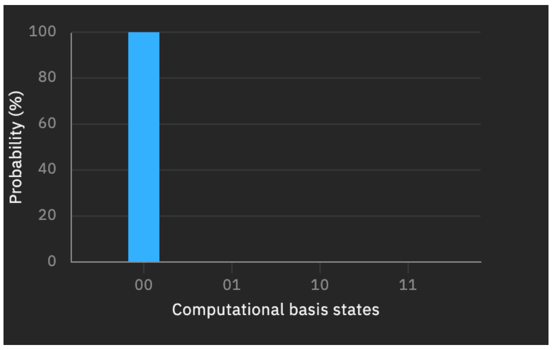

Figure 1.2: Output for C-NOT 00

input- 01

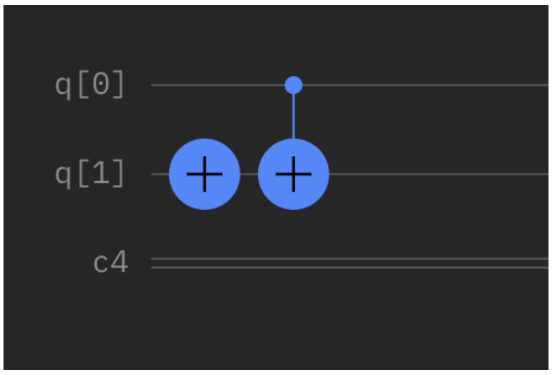

Figure 1.3: circuit diagram, 01

output for C-NOT - 01

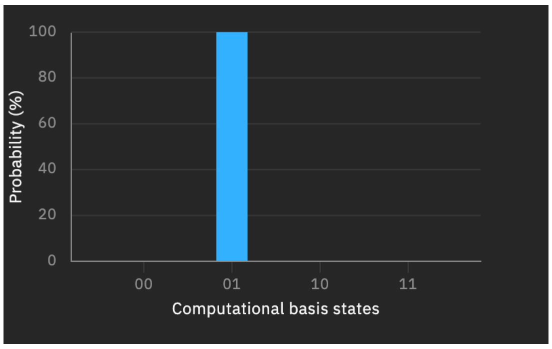

Figure 1.4: Output for C-NOT 01

input- 10

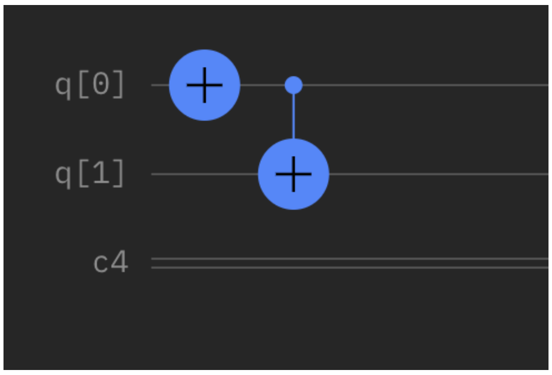

Figure 1.5: circuit diagram, 10

output for C-NOT - 10

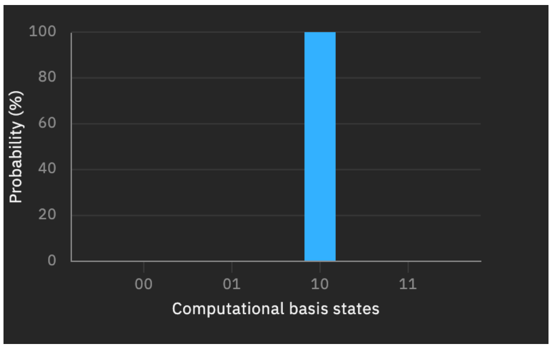

Figure 1.6: Output for C-NOT 10

input- 11

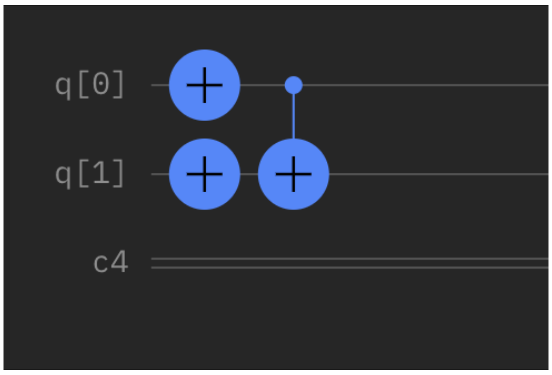

Figure 1.7: circuit diagram, 11

output for C-NOT - 11

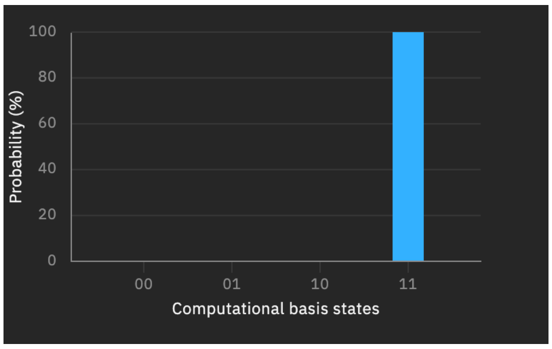

Figure 1.8: Output for C-NOT 11

1.1.2 ii

T Gate

input- 11

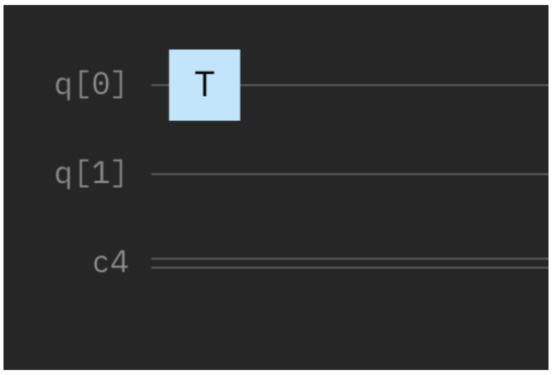

Figure 1.9: circuit diagram, for T Gate

1.1.3 iii

Bell State of $\frac{1}{\sqrt{2}}(|00> + |11>)$

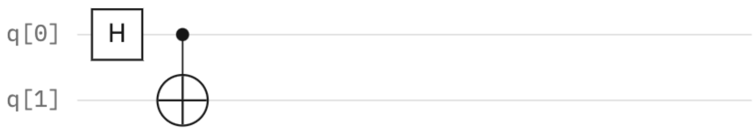

Figure 1.10: IBM Composer of Bell state $\frac{1}{\sqrt{2}}(|00> + |11>)$

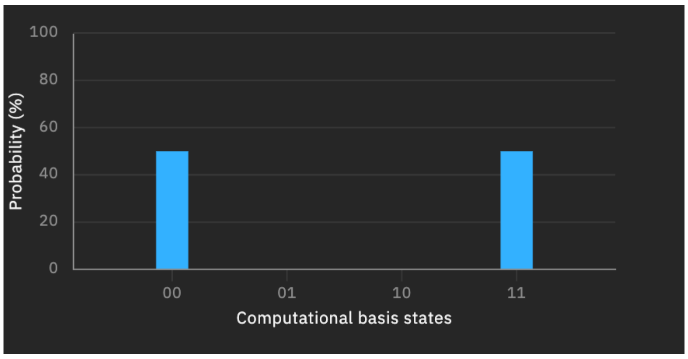

Figure 1.11: Probablity Distribution

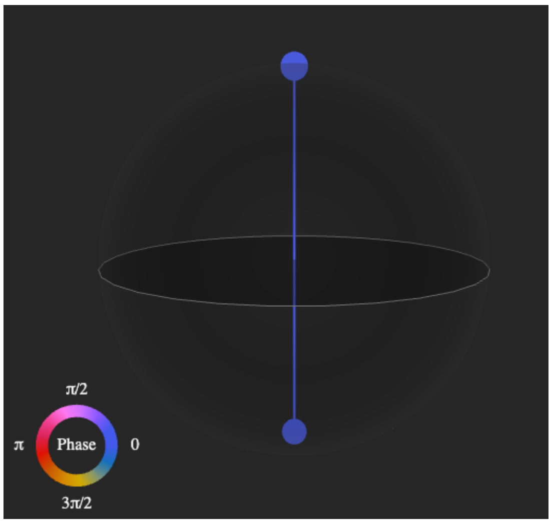

Figure 1.12: Representation in Bloch Sphere

Bell State of $\frac{1}{\sqrt{2}}(|00> -|11>)$

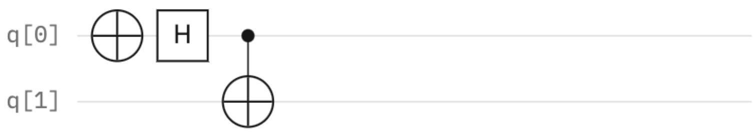

Figure 1.13: IBM Composer of Bell state $\frac{1}{\sqrt{2}}(|00> -|11>)$

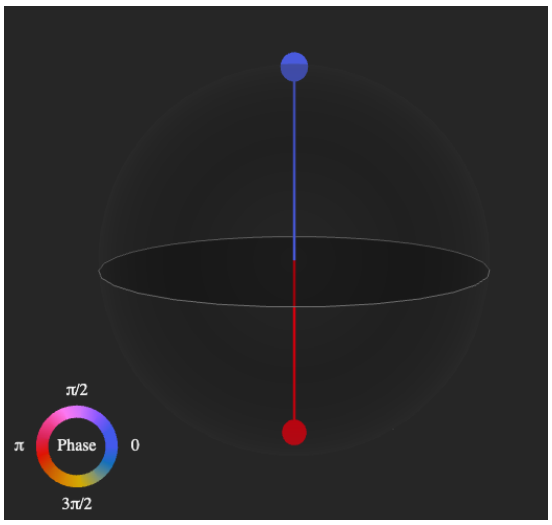

Figure 1.14: Representation in Bloch Sphere

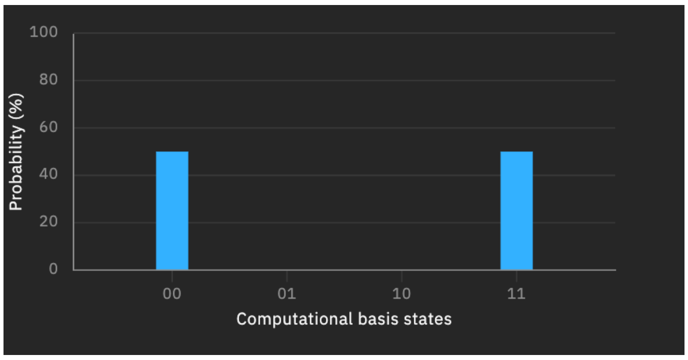

Figure 1.15: Probablity Distribution

Bell State of $\frac{1}{\sqrt{2}}(|01>+|10>)$

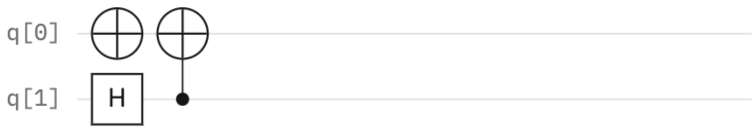

Figure 1.16: IBM Composer of Bell state $\frac{1}{\sqrt{2}}(|01>+|10>)$

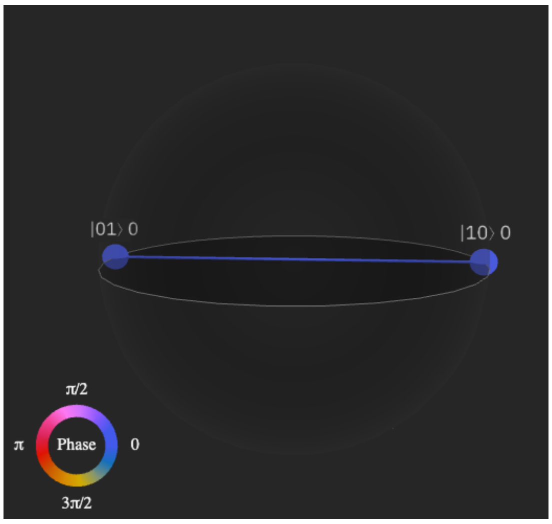

Figure 1.17: Representation in Bloch Sphere

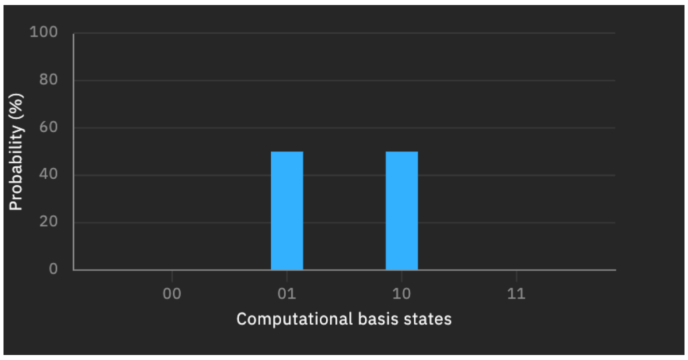

Figure 1.18: Probablity Distribution

Bell State of $\frac{1}{\sqrt{2}}(|01> -|10>)$

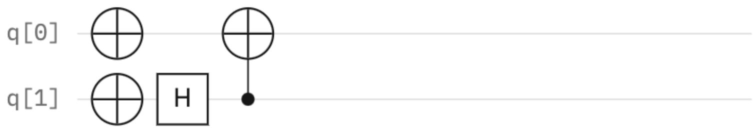

Figure 1.19: IBM Composer of Bell state

$$\frac{1}{\sqrt{2}}(|01> -|10>)$$

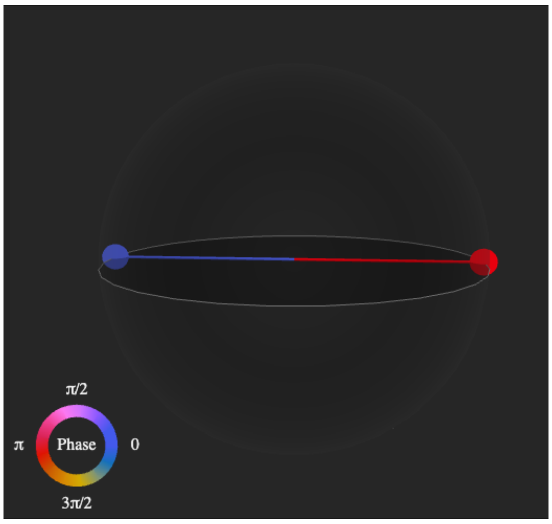

Figure 1.20: Representation in Bloch Sphere

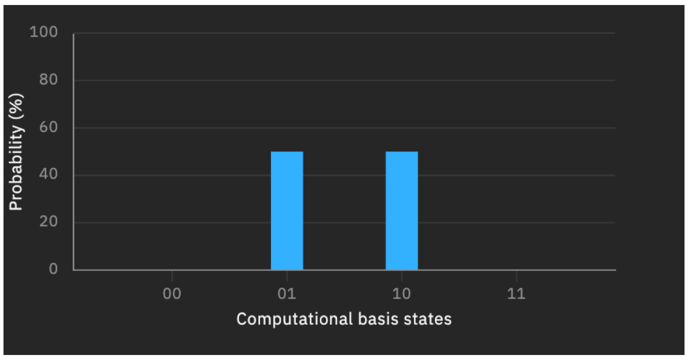

Figure 1.21: Probablity Distribution

1.1.4 i

GHZ state

GHZ State - $\frac{1}{\sqrt{2}}(|000> + |111>)$

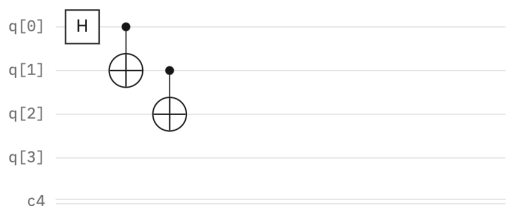

Figure 1.22: IBM Composer representation of GHZ state

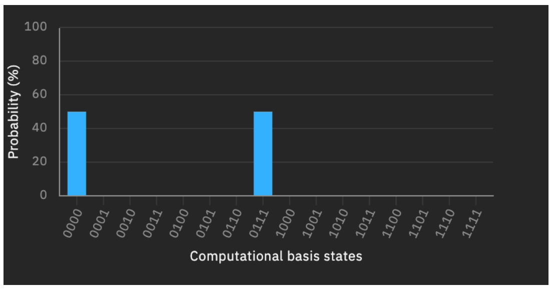

Figure 1.23: GHZ state Probablity distribution

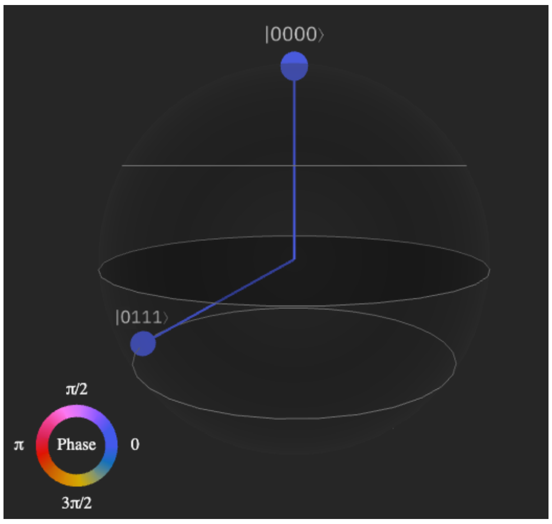

Figure 1.24: GHZ state Bloch Sphere representation

Chapter 2

Question B

Find the output of the Given Quantum Circuit and share your findings.

2.1 Deutsch-Jozsa Algorithm

2.2 Fundamentals of Deutsch-Jozsa Algorithm

The Deutsch-Jozsa technique enables the determination of whether a given Boolean function is constant or balanced. Let us consider a binary function that accepts input values of 0 and 1, and produces output values of 0 or 1. A function is deemed to be constant if and only if all of its outputs are either 0 or 1. A function is considered balanced when exactly half of its inputs are zeroes and exactly half of its inputs are ones.

2.2.1 Working Principles

The Deutsch-Jozsa algorithm encompasses several distinct implementations, although with a common 5-step procedure. The initial stage involves the preparation of quantum registers and input states. During the second stage, the qubits are arranged in accordance with the Hadamard Basis. This implies that every qubit possesses an equal probability of being in the state 0 or 1, each with a probability of 50%. The third stage involves the utilisation of an oracle, which encompasses the encoding of a function that establishes the nature of the outputs, whether they are constant or balanced, through the utilisation of quantum entanglement. The fourth phase involves the reversion of the qubits to the measurement basis for the purpose of determining the solution. In the final stage, the qubits undergo measurement in order to get the answer.

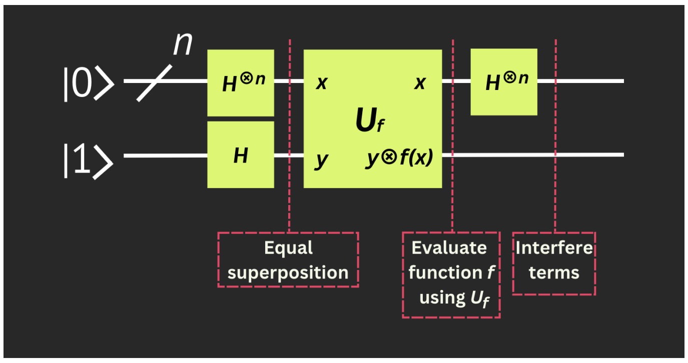

Figure 2.1: circuit diagram, credits classiq

There is minimal variation in the implementation methodologies between the initial and subsequent stages, with the primary distinction lying in the execution of the oracle. One often employed approach in implementing the oracle involves utilising a CX gate to initialise a secondary quantum register, which consists of a solitary ancilla qubit commonly known as a "work qubit." The standard initialization for qubits is typically set to the state 0. However, in this particular scenario, the ancilla qubit has been initialised to the state 1. Subsequently, the oracle employs a technique known as phase kickback to evaluate the nature of the function, determining whether it is constant or balanced. In the event that the function is balanced, the phenomenon of phase kickback will introduce a negative phase to precisely half of the quantum states.

2.2.2 Code

```
# useful additional packages
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# importing Qiskit
import qiskit
from qiskit import BasicAer, IBMQ
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister,
execute
from qiskit.compiler import transpile
from qiskit.tools.monitor import job_monitor

# import basic plot tools
from qiskit.tools.visualization import plot_histogram


n=13

# Choose a type of oracle at random. With probability half it is constant,
# and with the same probability it is balanced
oracleType, oracleValue = np.random.randint(2), np.random.randint(2)

if oracleType == 0:
    print("The oracle returns a constant value ", oracleValue)
else:
    print("The oracle returns a balanced function")
    a = np.random.randint(1,2**n) # this is a hidden parameter for
    balanced oracle.


# Creating registers
# n qubits for querying the oracle and one qubit for storing the answer
qr = QuantumRegister(n+1) #all qubits are initialized to zero
# for recording the measurement on the first register
cr = ClassicalRegister(n)
```

```
circuitName = "DeutschJozsa"
djCircuit = QuantumCircuit(qr, cr)

# Create the superposition of all input queries in the first register by
applying the Hadamard gate to each qubit.
for i in range(n):
    djCircuit.h(qr[i])

# Flip the second register and apply the Hadamard gate.
djCircuit.x(qr[n])
djCircuit.h(qr[n])

# Apply barrier to mark the beginning of the oracle
djCircuit.barrier()

if oracleType == 0:#If the oracleType is "0", the oracle returns
oracleValue for all input.
    if oracleValue == 1:
        djCircuit.x(qr[n])
    else:
        djCircuit.id(qr[n])
else: # Otherwise, it returns the inner product of the input with a (non-zero
bitstring)
    for i in range(n):
        if (a & (1 << i)):
            djCircuit.cx(qr[i], qr[n])
# Apply barrier to mark the end of the oracle
djCircuit.barrier()

# Apply Hadamard gates after querying the oracle
for i in range(n):
    djCircuit.h(qr[i])

# Measurement
djCircuit.barrier()
for i in range(n):
    djCircuit.measure(qr[i], cr[i])


Output :

The oracle returns a constant value  0


#draw the circuit
djCircuit.draw(output='mpl',scale=0.5)
```

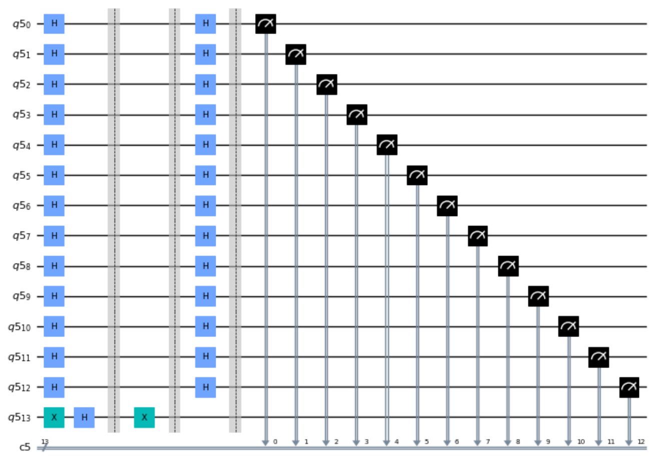

Figure 2.2: circuit for Algorithm

```
backend = BasicAer.get_backend('qasm_simulator')
shots = 5000
job = execute(djCircuit, backend=backend, shots=shots)
results = job.result()
answer = results.get_counts()

plot_histogram(answer)
```

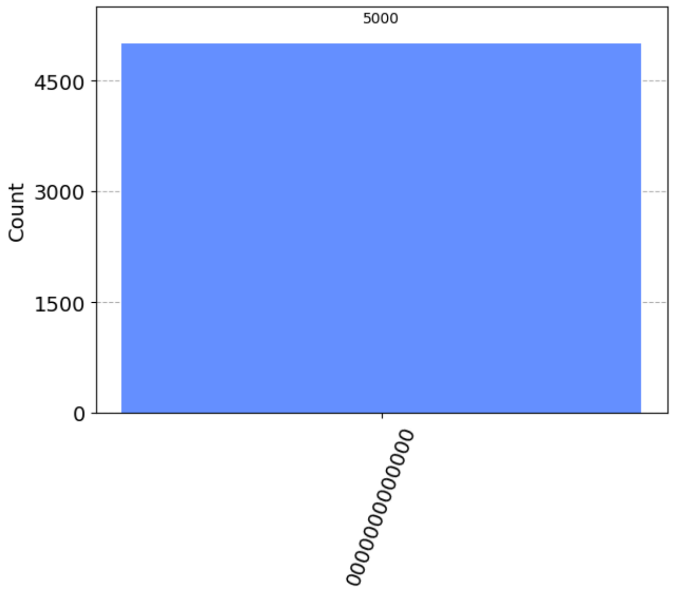

Figure 2.3: Probablity Distribution

2.3 Conclusion

The Deutsch-Jozsa method has no known real-world applications, but that was never its primary intent. The method was created to be simple for quantum computers yet complex for classical systems to solve. Nonetheless, it demonstrates that quantum computers may be able to address challenges more effectively than their traditional counterparts.

Chapter 3

Question C

Implement of CHSH violation IBM-Qiskit

3.1 CHSH inequality

The CHSH inequality, named after the authors Clauser, Horne, Shimony, and Holt, is used to establish Bell's theorem (1969) empirically. This theorem states that local hidden variable theories cannot account for some entanglement implications in quantum mechanics. The violation of the CHSH inequality is used to demonstrate that quantum mechanics and local hidden-variable theories are incompatible. This is a crucial experiment for comprehending the fundamentals of quantum physics.

For this experiment, we will create an entangled pair on which we measure each qubit on two different bases. We will label the bases for the first qubit A and a and the bases for the second qubit as B and b. This allows us to compute S_1

$$S_1 = A(B - b) + a(B + b)$$

Each observable is either +1 or -1. Clearly, one of the terms B+b or B-b must be 0,and the other must be +2 or -2. Therefore S_1 must satisfy the inequality

$$| < S_1 > | =<= 2$$

If quantum mechanics can be described by local hidden variable theories, the previous inequalities must hold true. However, as is demonstrated in code, these inequalities can be violated in a quantum computer. Therefore, quantum mechanics is not compatible with local hidden variable theories.

3.2 Code

```
import numpy as np

from qiskit import QuantumCircuit
from qiskit.circuit import Parameter
from qiskit.quantum_info import SparsePauliOp

from qiskit_ibm_runtime import QiskitRuntimeService, Estimator

import matplotlib.pyplot as plt
import matplotlib.ticker as tck
```

```
service = QiskitRuntimeService()
backend = service.get_backend("ibm_brisbane")


theta = Parameter("$\\theta$")

chsh_circuit_no_meas = QuantumCircuit(2)
chsh_circuit_no_meas.h(0)
chsh_circuit_no_meas.cx(0, 1)
chsh_circuit_no_meas.ry(theta, 0)
chsh_circuit_no_meas.draw("mpl")
```

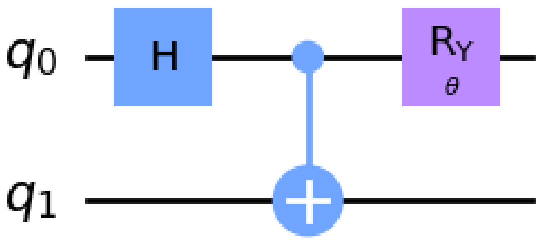

Figure 3.1: circuit diagram

```
number_of_phases = 21
phases = np.linspace(0, 2 * np.pi, number_of_phases)
# Phases need to be expressed as list of lists in order to work
individual_phases = [[ph] for ph in phases]

ZZ = SparsePauliOp.from_list([("ZZ", 1)])
ZX = SparsePauliOp.from_list([("ZX", 1)])
XZ = SparsePauliOp.from_list([("XZ", 1)])
XX = SparsePauliOp.from_list([("XX", 1)])

ops = [ZZ, ZX, XZ, XX]
num_ops = len(ops)


batch_circuits = [chsh_circuit_no_meas] * number_of_phases * num_ops
batch_ops = [op for op in ops for _ in individual_phases]

estimator = Estimator(backend)

batch_expvals = (
    estimator.run(
        batch_circuits, batch_ops, parameter_values=individual_phases *
        num_ops, shots=int(1e4)
    )
    .result()
    .values
)

ZZ_expval, ZX_expval, XZ_expval, XX_expval = [
    batch_expvals[kk * number_of_phases : (kk + 1) *
    number_of_phases] for kk in range(num_ops)
]



# <CHSH1> = <AB> - <Ab> + <aB> + <ab>
chsh1_est = ZZ_expval - ZX_expval + XZ_expval + XX_expval

# <CHSH2> = <AB> + <Ab> - <aB> + <ab>
chsh2_est = ZZ_expval + ZX_expval - XZ_expval + XX_expval


fig, ax = plt.subplots(figsize=(10, 6))
# results from hardware
ax.plot(phases / np.pi, chsh1_est, "o-", label="CHSH1", zorder=3)
ax.plot(phases / np.pi, chsh2_est, "o-", label="CHSH2", zorder=3)
# classical bound +-2
ax.axhline(y=2, color="0.9", linestyle="--")
ax.axhline(y=-2, color="0.9", linestyle="--")
# quantum bound, +-2 2
ax.axhline(y=np.sqrt(2) * 2, color="0.9", linestyle="-.")
ax.axhline(y=-np.sqrt(2) * 2, color="0.9", linestyle="-.")
ax.fill_between(phases / np.pi, 2, 2 * np.sqrt(2), color="0.6", alpha=0.7)
ax.fill_between(phases / np.pi, -2, -2 * np.sqrt(2), color="0.6", alpha=0.7)
# set x tick labels to the unit of pi
ax.xaxis.set_major_formatter(tck.FormatStrFormatter("%g $\pi$"))
ax.xaxis.set_major_locator(tck.MultipleLocator(base=0.5))
# set title, labels, and legend
plt.title(f"Violation of CHSH Inequality ({backend.name})")
plt.xlabel("Theta")
plt.ylabel("CHSH witness")
plt.legend();
```

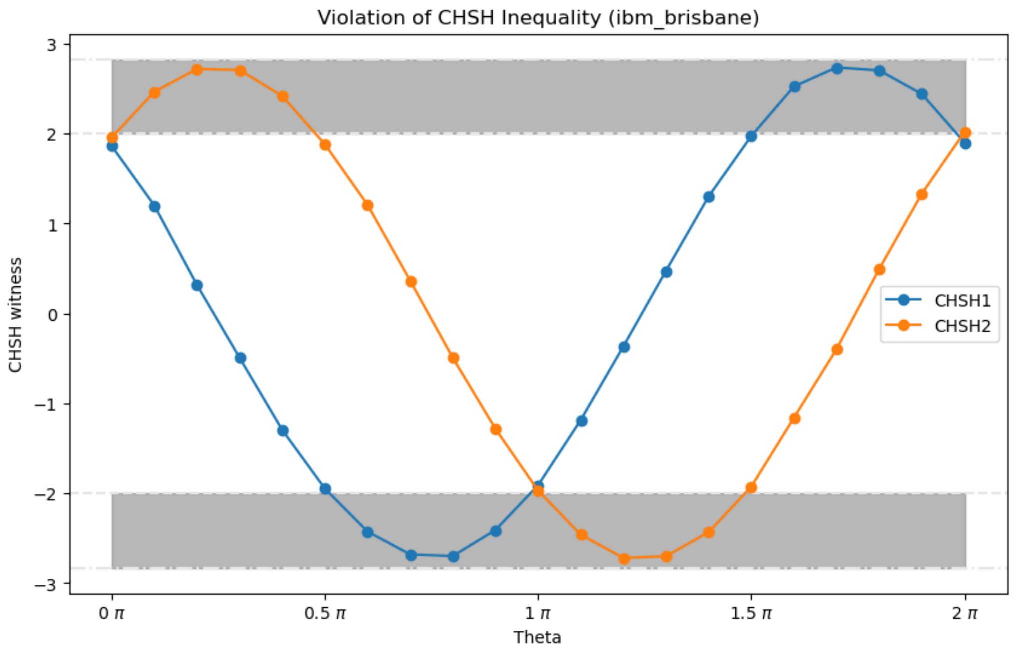

Figure 3.2: Violation of Bell inequality