# LS_QIQC_assignment_Week_1_22B3936

July 24, 2023

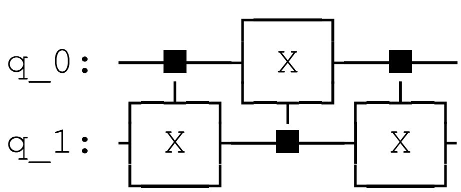
[1]:
```python
# Learner's Space: Quantum Computing Assignment_Week-1
# NAME: Sravan K Suresh
# Roll no: 22B3936
```

[2]:
```python
from qiskit import QuantumCircuit, Aer, execute, transpile, assemble
import qiskit.quantum_info as qi
from qiskit.quantum_info import Statevector, Operator
from qiskit.visualization import array_to_latex, circuit_drawer, plot_bloch_multivector, plot_histogram
import numpy as np
import matplotlib.pyplot as plt
```

[3]:
```python
# Q.1a] Code up the circuit to swap the states of two qubits. You should have seen the circuit
# already in QCQI/Qiskit Textbook
```

[4]:
```python
# Create a quantum circuit with two qubits
qc_swap = QuantumCircuit(2)

# Apply the swap operation using CNOT gates
qc_swap.cx(0, 1)  # Controlled-NOT gate with qubit 0 as control and qubit 1 as target
qc_swap.cx(1, 0)  # Controlled-NOT gate with qubit 1 as control and qubit 0 as target
qc_swap.cx(0, 1)  # Controlled-NOT gate with qubit 0 as control and qubit 1 as target

# The first CNOT swaps qubit 0 into qubit 1 if qubit 0 is in state |1⟩,
# and the second CNOT undoes this operation, bringing the state of qubit 1 back to qubit 0.
# Finally, the third CNOT swaps the state of qubit 1 back into qubit 0.

# Visualize the circuit
qc_swap.draw()
```

[4]:



[5]:
```
# Execute the circuit on a simulator
simulator = Aer.get_backend('statevector_simulator')
result = execute(qc_swap, simulator).result()
statevector = result.get_statevector()

# State input to circuit- Computational basis state { |00⟩ } (hence I expect␣
↪|00⟩ = [1, 0, 0, 0] to be returned)

# Print the final statevector after the swap
print("Final statevector after the swap:")
print(statevector)
```

```
Final statevector after the swap:
Statevector([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
            dims=(2, 2))
```

[6]:
```
# Trial run with qubits |10⟩, expected output: |01⟩ (Qubits swapped)
# Define the ket vector for the input state |psi⟩ = a|00⟩ + b|01⟩ + c|10⟩ + d|11⟩
a = 0
b = 0
c = 1
d = 0

input_ket_vector = [a, b, c, d]  # Ket vector [a, b, c, d]

# Create a quantum circuit with the desired number of qubits
num_qubits = 2
qc_swap = QuantumCircuit(num_qubits)

# Feed the input state to the quantum circuit using the initialize method
qc_swap.initialize(input_ket_vector, range(num_qubits))

qc_swap.cx(0, 1)  # Controlled-NOT gate with qubit 0 as control and qubit 1 as␣
↪target
qc_swap.cx(1, 0)  # Controlled-NOT gate with qubit 1 as control and qubit 0 as␣
↪target
qc_swap.cx(0, 1)  # Controlled-NOT gate with qubit 0 as control and qubit 1 as␣
↪target

# Measure the qubits (if necessary) to get the measurement outcomes
qc_swap.measure_all()

# Simulate the circuit
```

```
simulator = Aer.get_backend('qasm_simulator')
job = execute(qc_swap, simulator, shots=1024)  # You can adjust the number of␣
↪shots as needed
result = job.result()

# Now that I've set the registers of input qubits to this circuit as |10⟩ (as c␣
↪= 1),
# I expect the output by this ckt to be |01⟩

# Get the counts from the result
counts = result.get_counts(qc_swap)
print(counts)
```

```
{'01': 1024}
```

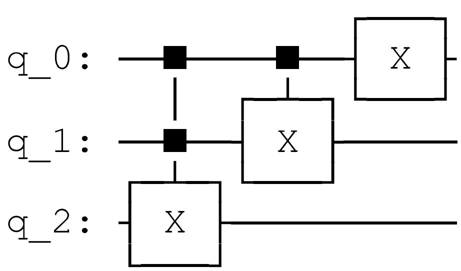
[7]:

```
# Q.1b] Given a three digit binary number abc, code up a circuit to increment␣
↪the number by 1
# (mod 8) The result should be stored in-place, i.e., in the same qubits that␣
↪are used for the
# inputs. The inputs will be of the form |a⟩  |b⟩  |c⟩ where a, b, c are each␣
↪either 0 or 1
```

[8]:

```
# Create a quantum circuit with two qubits
qc_mod8 = QuantumCircuit(3)

# Apply the increment by 1 (mod 8) operation using gates Pauli-X, Controlled-NOT␣
↪and Controlled-controlled-NOT (Toffoli)

qc_mod8.ccx(0, 1, 2)  # Toffoli gate with qubit 0 and qubit 1 as control and␣
↪qubit 2 as target
qc_mod8.cx(0, 1)  # Controlled-NOT gate with qubit 1 as control and qubit 0 as␣
↪target
qc_mod8.x(0)  # X gate on qubit 0

# Visualize the circuit
qc_mod8.draw()
```

[8]:




[9]:

```
# Execute the circuit on a simulator
simulator = Aer.get_backend('statevector_simulator')
```

```
result = execute(qc_mod8, simulator).result()
statevector = result.get_statevector()

# State input to circuit- Computational basis state { |000⟩ }
# (hence I expect the output to be |001⟩ = [0, 1, 0, 0, 0, 0, 0, 0, 0] to be␣
↪returned)

# Print the final statevector after the swap
print("Final statevector after the increment:")
print(statevector)
```

```
Final statevector after the increment:
Statevector([0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j,
             0.+0.j],
            dims=(2, 2, 2))
```

[10]:

```
# Trial run with qubits |111⟩, expected output: |000⟩
# Define the ket vector for the input state |psi⟩ = a|000⟩ + b|001⟩ + c|010⟩ +␣
↪d|011⟩ + e|100⟩ + f|101⟩ + g|110⟩ + h|111⟩
a = 0
b = 0
c = 0
d = 0
e = 0
f = 0
g = 0
h = 1

input_ket_vector = [a, b, c, d, e, f, g, h]  # Ket vector [a, b, c, d, e, f, g,␣
↪h]

# Create a quantum circuit with the desired number of qubits
num_qubits = 3
qc_mod8 = QuantumCircuit(num_qubits)

# Feed the input state to the quantum circuit using the initialize method
qc_mod8.initialize(input_ket_vector, range(num_qubits))

qc_mod8.ccx(0, 1, 2)  # Toffoli gate with qubit 0 and qubit 1 as control and␣
↪qubit 2 as target
qc_mod8.cx(0, 1)  # Controlled-NOT gate with qubit 1 as control and qubit 0 as␣
↪target
qc_mod8.x(0)  # X gate on qubit 0

# Measure the qubits (if necessary) to get the measurement outcomes
qc_mod8.measure_all()
```

```
# Simulate the circuit
simulator = Aer.get_backend('qasm_simulator')
job = execute(qc_mod8, simulator, shots=1024)  # You can adjust the number of␣
↪shots as needed
result = job.result()

# Get the counts from the result
counts = result.get_counts(qc_mod8)
print(counts)
```

```
{'000': 1024}
```

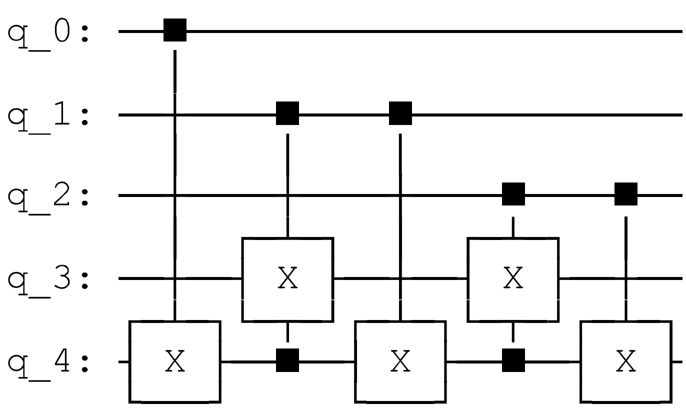
[11]:

```
# Q.1d] The Hamming Weight of a binary number is the number of 1s in its binary␣
↪representa}tion.
# For a binary number with 3 bits, construct a circuit that takes |x⟩ |0⟩ to |x⟩␣
↪|w(x)⟩ where
# w(x) is the Hamming weight of (x).
```

[12]:

```
# Create a quantum circuit
qc_HammingW = QuantumCircuit(5)

qc_HammingW.cx(0, 4)
qc_HammingW.ccx(1, 4, 3)
qc_HammingW.cx(1, 4)
qc_HammingW.ccx(2, 4, 3)
qc_HammingW.cx(2, 4)

# Visualize the circuit
qc_HammingW.draw()

# Here, q_3 and q_4 ought to be universally 0 (I think so)
```

[12]:




[13]:

```
# Q.2] Implement either the Deutsch-Josza or Bernstein Vazirani algorithms (try␣
↪not to copy Qiskit
# Textbook, please :))
```

[14]:
```
# Define the Bernstein-Vazirani algorithm function
def BV_algo(secret_string):
    # Calculate the number of qubits required
    n_qubits = len(secret_string)

    # Create a quantum circuit with n_qubits plus one ancillary qubit and a␣
↪classical output bit
    circuit = QuantumCircuit(n_qubits + 1, n_qubits)

    # Apply Hadamard gate to all qubits
    circuit.h(range(n_qubits))

    # Apply X and H gate to the ancillary qubit
    circuit.x(n_qubits)
    circuit.h(n_qubits)

    # Apply the secret string function to the quantum circuit
    for qubit in range(n_qubits):
        if secret_string[qubit] == '1':
            circuit.cx(qubit, n_qubits)

    # Apply Hadamard gate to the first n_qubits
    circuit.h(range(n_qubits))

    # Measure the first n_qubits
    circuit.measure(range(n_qubits), range(n_qubits))

    return circuit

# Trial Run (example: "101010")
secret_string = "101010"

# Create the quantum circuit for the given secret string
circuit = BV_algo(secret_string)

# Simulate the quantum circuit using the Aer simulator
simulator = Aer.get_backend('qasm_simulator')
job = execute(circuit, simulator, shots=1)

# Get the result
result = job.result()
counts = result.get_counts(circuit)

# Print the result
print("The secret string is:", secret_string[::-1])
print("Measurement result:", list(counts.keys())[0][::-1])
```

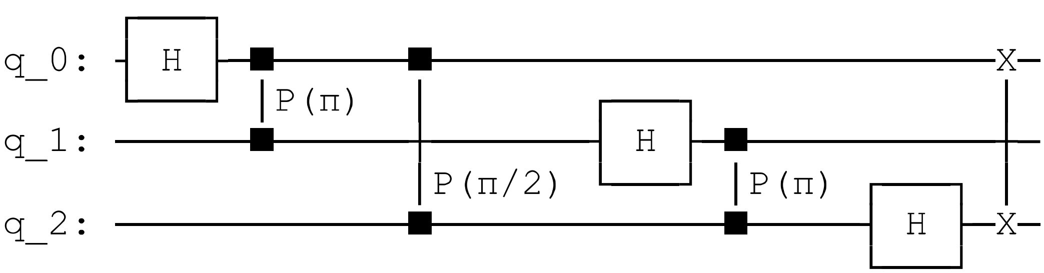
```
The secret string is: 010101
```

```
Measurement result: 101010
```

[15]:
```
# Q.4] Implement the following in qiskit as well (Implementation always helps␣
↪you understand it
# better)
# • Quantum Fourier Transform
# • Quantum Phase Estimation
# • Shor’s Algorithm
# • Grover’s Search Algorithm
```

[16]:
```
# Function to create the Quantum Fourier Transform (QFT) circuit
def qft(n):
    circuit = QuantumCircuit(n)

    for qubit in range(n):
        circuit.h(qubit)
        for controlled_qubit in range(qubit+1, n):
            angle = 2 * np.pi / (2 ** (controlled_qubit - qubit))
            circuit.cp(angle, controlled_qubit, qubit)

    # Swap the qubits
    for i in range(n // 2):
        circuit.swap(i, n - i - 1)

    return circuit

# Number of qubits
n_qubits = 3

# Here I've used n_qubits = 3, similarly n = 4, 5, ...... can be simulated by␣
↪changing 'n' here

# Create the Quantum Circuit for QFT
qc = qft(n_qubits)

# Visualize the circuit
qc.draw()
```

[16]:



[17]:
```
# I am left with Q.3] and 3 parts of Q.4] which I will complete soon.
# I am not able to complete it right now as I am down with high fever :(
```