QCQT-QY3 Quantum Algorithms and Quantum Information 2024-2025
Professor Karafyllidis Ioannis
Topalidis Dimitrios
Assignment 1

The Bernstein-Vazirani quantum algorithm takes as input a function (oracle) that implements the mapping:

$$f : \{0,1\}^n \to \{0,1\}$$

For the sake of this exercise, we choose the oracle to be:

$$f(x) = x\,s = x_1 s_1 \oplus x_2 s_2 \oplus \cdots \oplus x_n s_n (\mod 2)$$

which gives the product of a hidden number **s**, for which only the nubmer of digits is known, with a number **x**. The function is supposed that can be only balanced or constant.
Then the goal of the algorithm is to find **s** with only one computation, regardless of its number of digits.

For this assignment, we are asked to implement the Bernstein-Varizani quantum algorithm using Qiskit and then execute it on a quantum computer, for the case of:

$$s = 1010$$

My implementation takes into account the general case and can work for any given bitstring and oracle.
In our example though, we will test the implementation with the abovementioned oracle and the bitstring $s = 1010$ as required by the exercise

The main implementation is found in the method **bernstein_varizani(oracle,n)**
This method takes two arguments:

- **oracle**, which is a quantum circuit in Qiskit, which implements the oracle.
- **n**, which is the known number of digits of the hidden number

Then, we have the following steps:

1. We initialize a quantum circuit with $\mathbf{n+1}$ qubits, where **n** is the number of the digits of the hidden number and **n** classical bits.
2. We apply a Hadamard gate to all qubits except from the ancilla qubit.
3. We apply an Pauli X gate to the ancilla qubit, in order to prepate its state to $|1\rangle$, and then apply a Hadamard gate separately.
4. We append the oracle circuit.
5. We apply again a Hadamard gate to all qubits except from the ancilla qubit.
6. We measure all qubits except of the ancilla qubit, in the respective quantum register

```
def bernstein_varizani(oracle, n):


    #Step 1
    qc = QuantumCircuit(n + 1, n)

    #Step 2
    qc.h(np.arange(0, n, 1))
```

```
    #Step 3
#Preparation step: We need the ancilla qubit to be prepared to |1>.
#                  There in no built-in way in Qiskit to achieve this, since all qubits are initialized
     in state |0>.
#                  For this reason, we are applying a Pauli X gate to the ancilla qubit, so we flip it'
    s state from |0> to |1>
    qc.x(n)
    qc.h(n)


    #Step 4
    qc.barrier()
    qc.compose(oracle, qubits=range(n + 1), inplace=True)
    qc.barrier()


    #Step 5
    qc.h(np.arange(0, n, 1))
    qc.barrier()


    #Step 6
    qc.measure(np.arange(0, n, 1),np.arange(0, n, 1))

    return qc
```

Having implemented the Bernstein-Varizani algorithm, we need now to create the oracle required by the assignment, as follows:

```
def assignment1_oracle(s):
    n = len(s)
    qc = QuantumCircuit(n+1)

    for i, bit in enumerate(reversed(s)):
        if bit == "1":
            qc.cx(i, n)
    return qc
```

We now have to "build" our circuit, by calling the **bernstein_varizani** function and passing the parameters of this example. We can also draw it, to have a nice visual representation.

```
    s = "1010"
    o = assignment1_oracle(s)
    qc = bernstein_verizani(o, len(s))
    qc.draw('mpl')
```

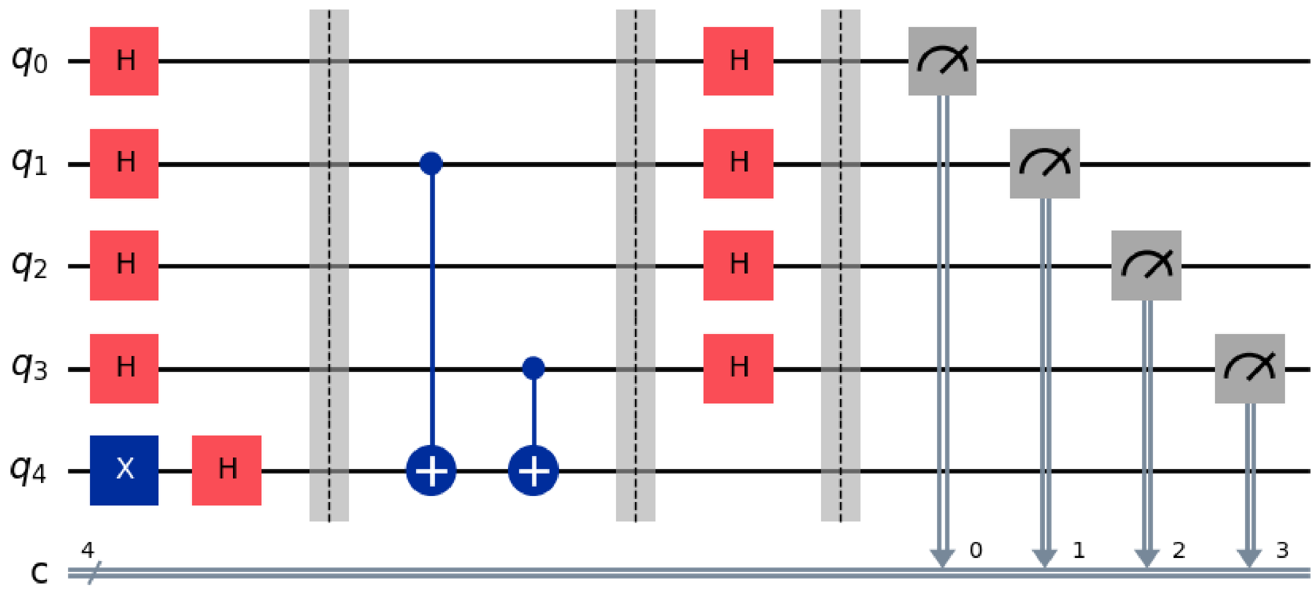

Figure 1: Visual representation of the Bernstein Varizani algorithm circuit implementation in Qiskit, for the example of the assignment

We then proceed to transpile our quantum circuit and simulate it **without noise**, using the **StatevectorSampler** and to plot the results.

```
#Simulation using StatevectorSampler
pm = generate_preset_pass_manager(optimization_level=1)
transpiled_qc = pm.run(qc)
statevectorSampler = StatevectorSampler()
job = statevectorSampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)
```

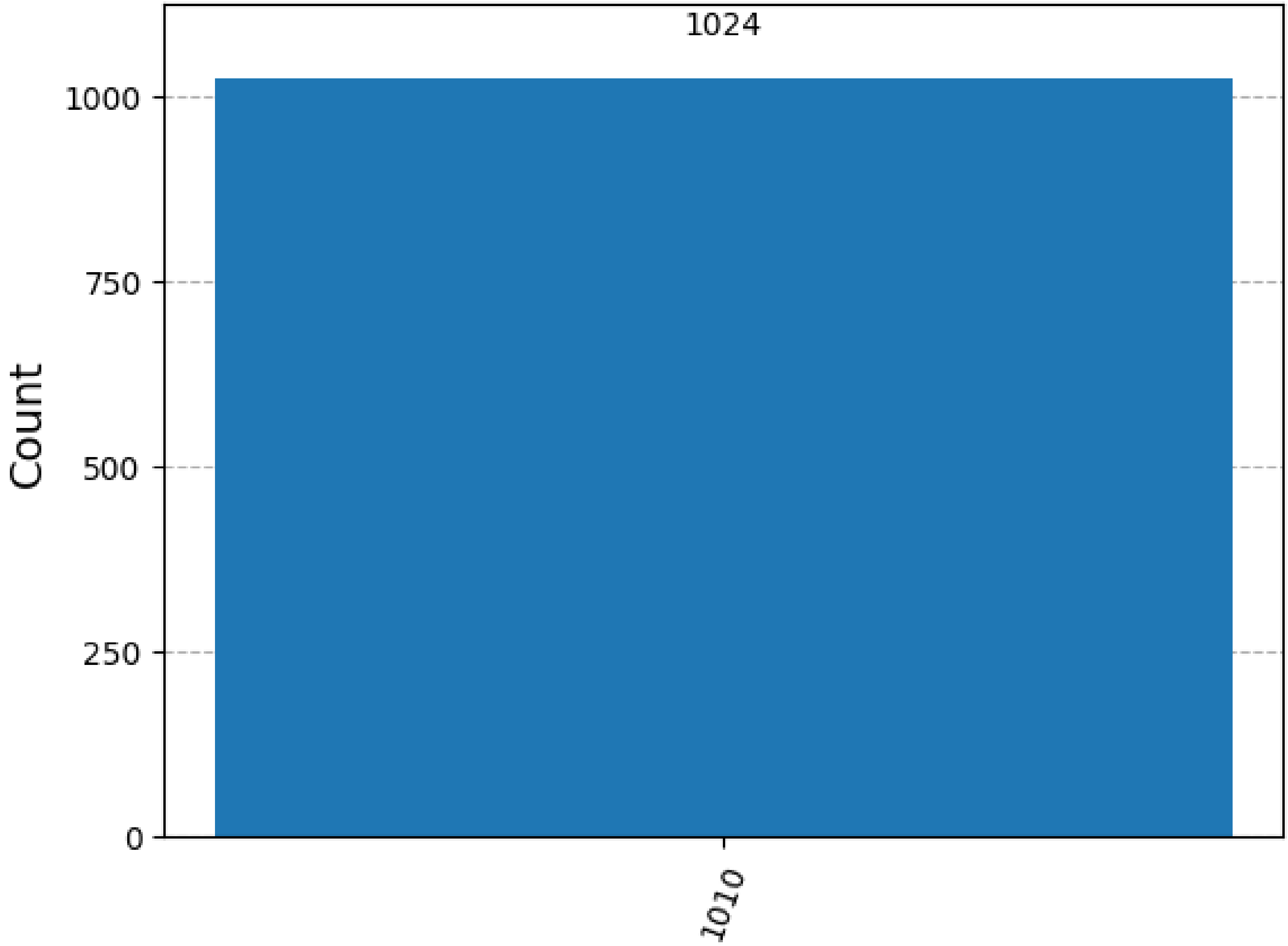

Figure 2: Simulation result using StatevectorSampler

Also, we transpile the quantum circuit and simulate its execution using the fake backend **FakeAlmadenV2**, but with **SamplerV2** this time:

```
#Simulation using SamplerV2
backend = FakeAlmadenV2()
pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
transpiled_qc = pm.run(qc)
sampler = SamplerV2(mode=backend)
job = sampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)
```

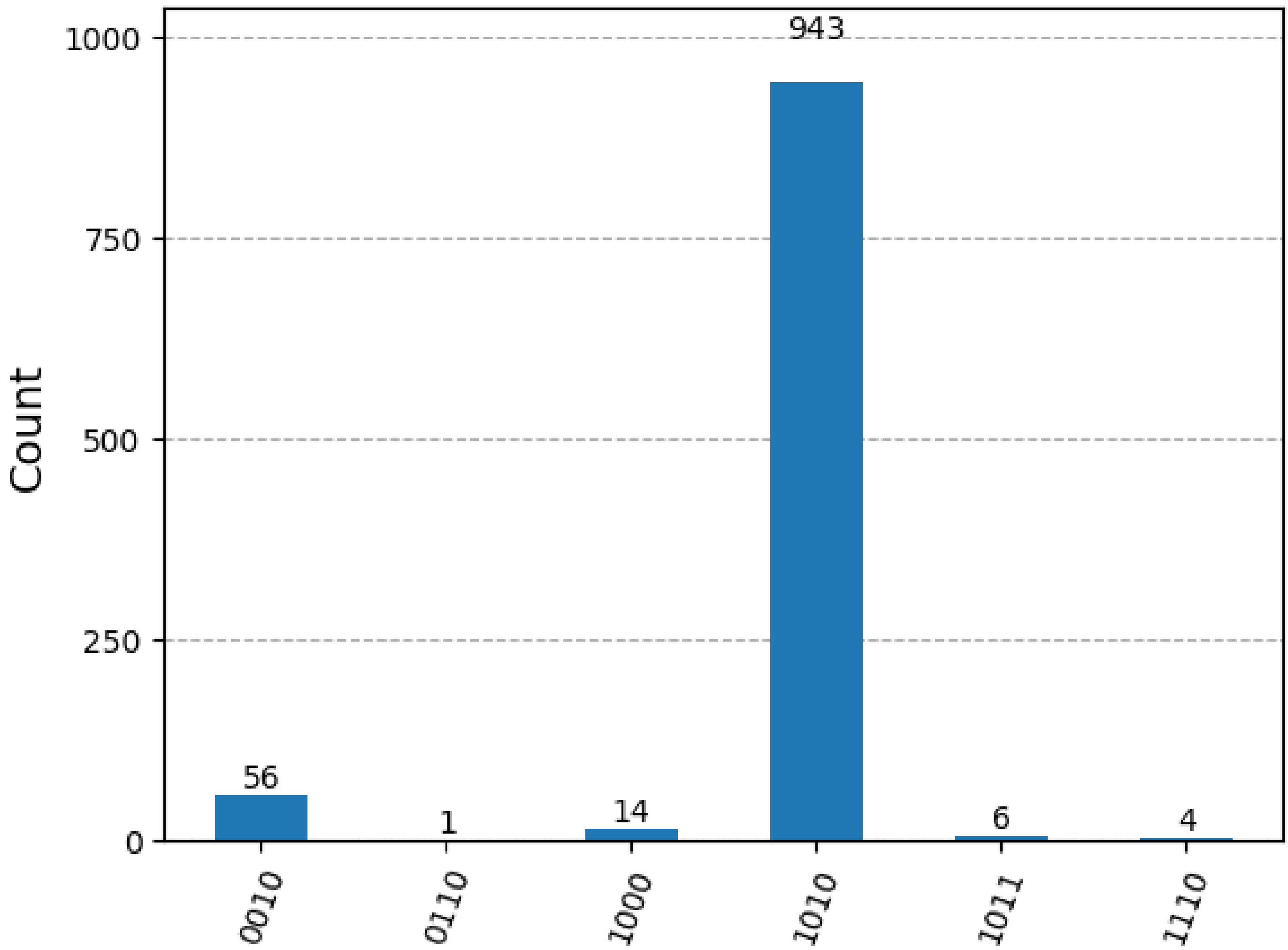

Figure 3: Simulation result using SamplerV2

Finally, we execute the Bernstein-Varizani algorithm in a real IBM Quantum Computer:

```
#Execution on a real quantum computer
service = QiskitRuntimeService(
    channel='ibm_quantum',
    token='<IBM_QUANTUM_TOKEN>'
)
shots=1024

backend = service.least_busy(operational=True, simulator=False)
pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
transpiled_qc = pm.run(qc)
sampler = SamplerV2(backend)
job = sampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)
```

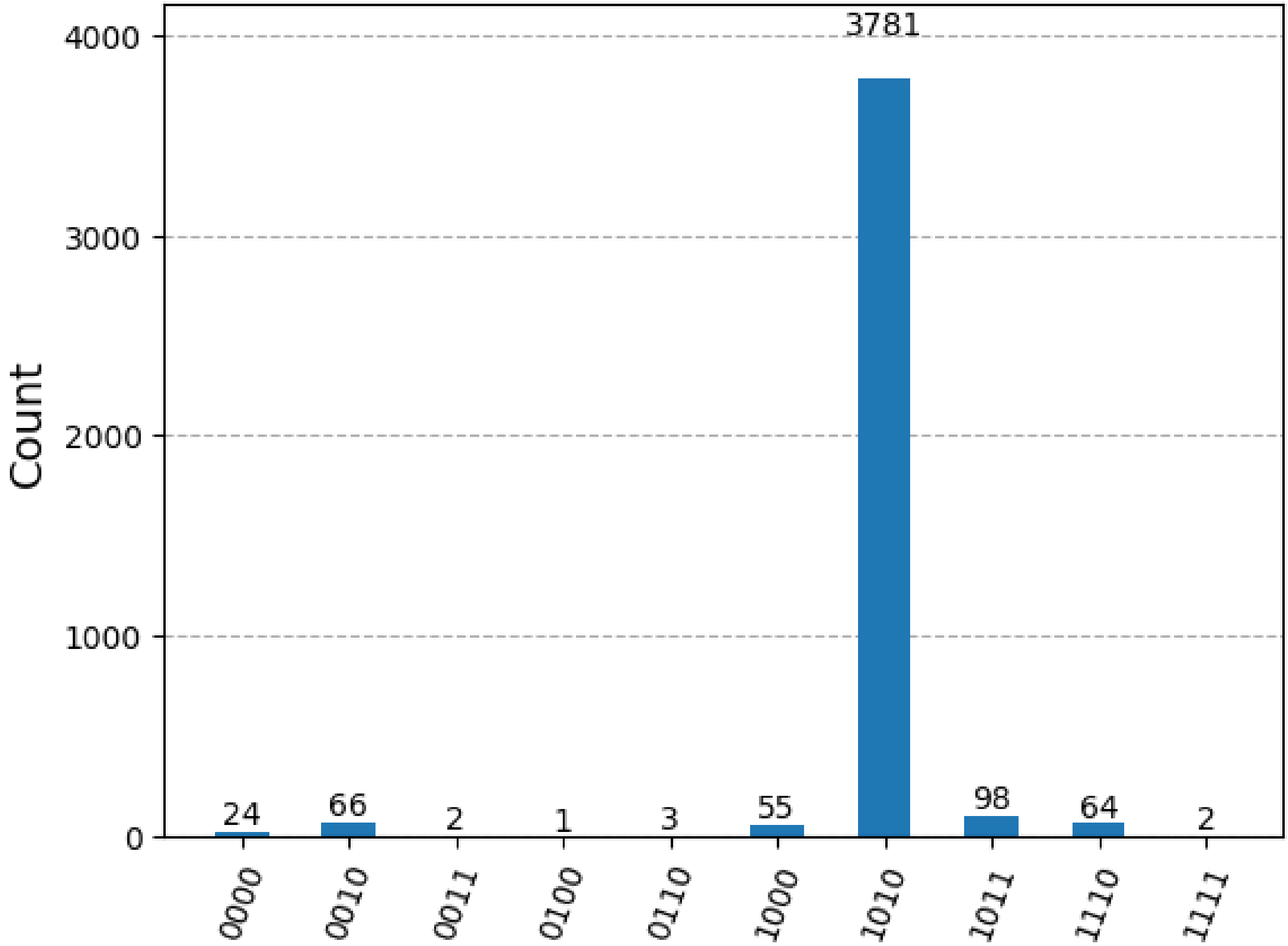

Figure 4: Result after execution on a real quantum computer

You will find all of the code used for this exercise in this repository: `https://github.com/topalidisd-qcqt-duth/qy3-assignment-1`
You can also find all of the code below:

```
from qiskit import *
import numpy as np
from qiskit_ibm_runtime.fake_provider import FakeAlmadenV2
from qiskit.visualization import plot_histogram
from qiskit_ibm_runtime import QiskitRuntimeService, SamplerV2
from qiskit.primitives import StatevectorSampler
from qiskit.transpiler import generate_preset_pass_manager

def assignment1_oracle(s):
    n = len(s)
    qc = QuantumCircuit(n+1)

    for i, bit in enumerate(reversed(s)):
        if bit == "1":
            qc.cx(i, n)
    return qc

def bernstein_varizani(oracle, n):


    #Step 1
    qc = QuantumCircuit(n + 1, n)

    #Step 2
    qc.h(np.arange(0, n, 1))



    #Step 3
#Preparation step: We need the ancilla qubit to be prepared to |1>.
#                  There in no built-in way in Qiskit to achieve this, since all qubits are initialized
     in state |0>.
#                  For this reason, we are applying a Pauli X gate to the ancilla qubit, so we flip it'
    s state from |0> to |1>
    qc.x(n)
    qc.h(n)


    #Step 4
    qc.barrier()
    qc.compose(oracle, qubits=range(n + 1), inplace=True)
    qc.barrier()


    #Step 5
    qc.h(np.arange(0, n, 1))
    qc.barrier()


    #Step 6
    qc.measure(np.arange(0, n, 1),np.arange(0, n, 1))

    return qc

#Example required by the assignment
s = "1010"
o = assignment1_oracle(s)
qc = bernstein_varizani(o, len(s))
qc.draw('mpl')


#Simulation using StatevectorSampler
pm = generate_preset_pass_manager(optimization_level=1)
transpiled_qc = pm.run(qc)
statevectorSampler = StatevectorSampler()
job = statevectorSampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)
```

```
#Simulation using SamplerV2
backend = FakeAlmadenV2()
pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
transpiled_qc = pm.run(qc)
sampler = SamplerV2(mode=backend)
job = sampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)


#Execution on a real quantum computer
service = QiskitRuntimeService(
    channel='ibm_quantum',
    token='<IBM_QUANTUM_TOKEN>'
)
shots=1024

backend = service.least_busy(operational=True, simulator=False)
pm = generate_preset_pass_manager(backend=backend, optimization_level=1)
transpiled_qc = pm.run(qc)
sampler = SamplerV2(backend)
job = sampler.run([transpiled_qc])
pub_result = job.result()[0]
counts = pub_result.data.c.get_counts()
plot_histogram(counts)
```