Implementation of Quantum Algorithms

Raimondas Čiegis

Department of Mathematical Modelling, email: rc@vgtu.lt

September 12, 2026

Let's start from the example given in a previous Lecture 3:

a circuit for turning $|00\rangle$ into an entangled state.

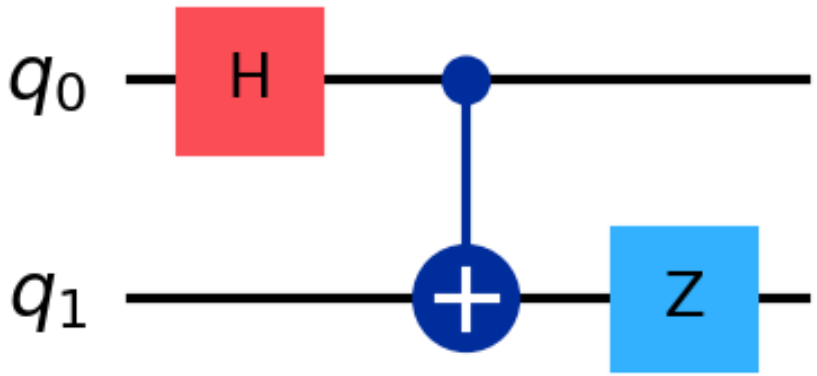


Fig1. Simple circuit for turning $|00\rangle$ into an entangled state

Qiskit

Qiskit is an open-source software development kit created by IBM

for writing, optimizing, and running quantum programs on simulators and real quantum hardware.

Qiskit's qubit ordering convention for circuits

The topmost qubit in a circuit diagram has index 0 and corresponds to the rightmost position in a tuple of qubits (or in a string, Cartesian product) corresponding to this tuple.

The second-from-top qubit has index 1 and corresponds to the position second-from-right in a tuple, and so on, down to the bottommost qubit, which has the highest index, and corresponds to the leftmost position in a tuple.

In particular, Qiskit's default names for the qubits in an n-qubit circuit are represented by the n-tuple $(q_{n-1}, ..., q_0)$ with q_0 being the qubit on the top and q_{n-1} on the bottom in quantum circuit diagrams.

In Qiskit's convention, higher qubit indices are more significant.

In many textbooks, controlled gates are presented with the assumption of more significant qubits as control, which in the case of $(q_0, q_1)^T$ would be q_1.

Let's remind You step-by-step the algorithm to get an entangled state of two qubits.

We will use ordering of qubits assumed in Qiskit tool kit.

Let's remind You step-by-step the algorithm to get an entangled state of two qubits.

We will use ordering of qubits assumed in Qiskit tool kit.

First apply the Hadamard gate to the top qubit q_0:

$$|A_1\rangle = I \otimes H\,|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle),$$

then apply $CNOT$ to both qubits with the top qubit acting as the control one

$$|A_2\rangle = CNOT\,|A_1\rangle = \frac{1}{\sqrt{2}}\big(\,|00\rangle + |11\rangle\,\big).$$

and then apply Z gate to the circuit bottom qubit q_1:

$$|A\rangle = Z \otimes I\;|A_2\rangle = \frac{1}{\sqrt{2}}\big(\,|00\rangle - |11\rangle\,\big).$$

Qiskit Program

```
from qiskit import QuantumCircuit, QuantumRegister
qubits = QuantumRegister(2, name = "q")
q0, q1 = qubits
circuit = QuantumCircuit(qubits)
circuit.h(q0)
circuit.cx(q0, q1)
circuit.z(q1)
circuit.draw(output="mpl", style="iqp",
filename='circuitEPR.png')
```

Now we want to run this circuit on a Qiskit simulator.

Now we want to run this circuit on a Qiskit simulator.

A convenient way to run simulations is to apply these general tools:

AerSimulator, Aer

Now we want to run this circuit on a Qiskit simulator.

A convenient way to run simulations is to apply these general tools:

AerSimulator, Aer

```
from qiskit import QuantumCircuit
from qiskit_aer import AerSimulator, Aer
qc = QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0,1)    # control qubit q_0 for |q_1, q_0⟩
qc.z(1)
qc.barrier()    # Used to split parts of the circuit
```

qc.measure([0, 1], [0, 1]) # maps quantum register indices to classical register bits: mapping qubit 0 to classical bit 0 and qubit 1 to classical bit 1.

```
sim = Aer.get_backend('qasm_simulator')
result = sim.run(qc, shots=512).result()
answer = result.get_counts()
print(answer)
```

qc.measure([0, 1], [0, 1]) # maps quantum register indices to classical register bits: mapping qubit 0 to classical bit 0 and qubit 1 to classical bit 1.

sim = Aer.get_backend('qasm_simulator')

result = sim.run(qc, shots=512).result()

answer = result.get_counts()

print(answer)

How measure Works

First list [0, 1]: Specifies the target quantum bits (qubit 0 and qubit 1).

Second list [0, 1]: Specifies the destination classical bits where the measurement outcomes (0 or 1) are saved.

Correspondence: Index-by-index pairing means qubit[0] goes to clbit[0] and qubit[1] goes to clbit[1].

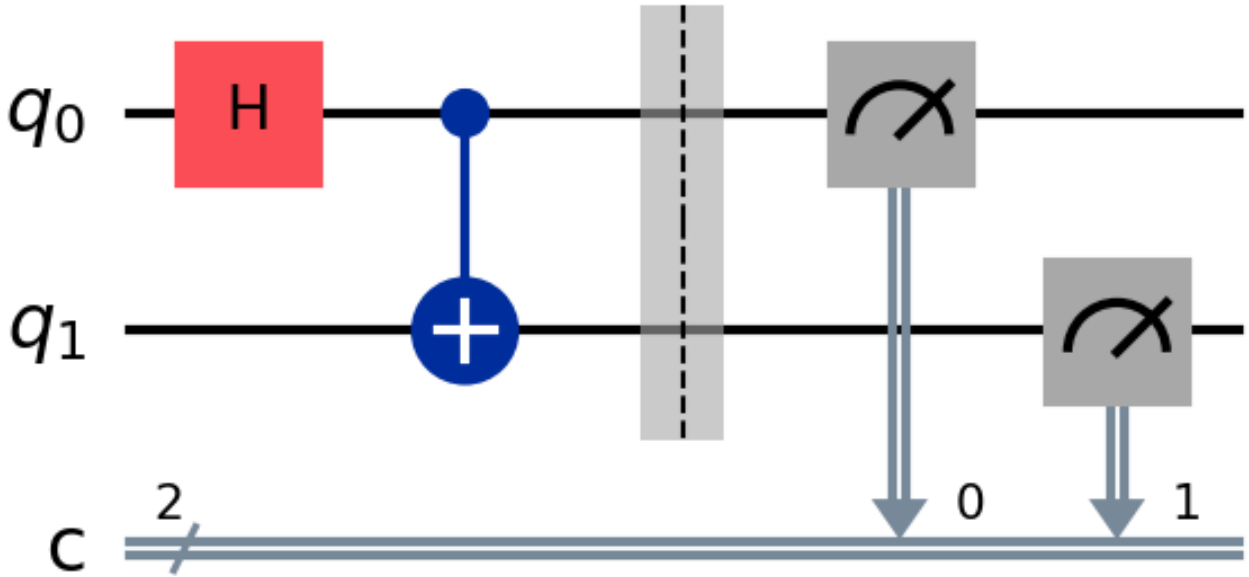


Fig2. A simple start circuit with two qubits.

We run this circuit on a simulator

```
result = sim.run(qc, shots=512).result()
```

We run this circuit on a simulator

```
result = sim.run(qc, shots=512).result()
```

```
{'11': 264, '00': 248}
```

Teleportation

```
# Required imports

from qiskit import QuantumCircuit, QuantumRegister,
ClassicalRegister

from qiskit.quantum_info import Statevector, Operator

qubit = QuantumRegister(1, "Q")
ebit0 = QuantumRegister(1, "A")
ebit1 = QuantumRegister(1, "B")
a = ClassicalRegister(1, "a")
b = ClassicalRegister(1, "b")

protocol = QuantumCircuit(qubit, ebit0, ebit1, a, b)
```

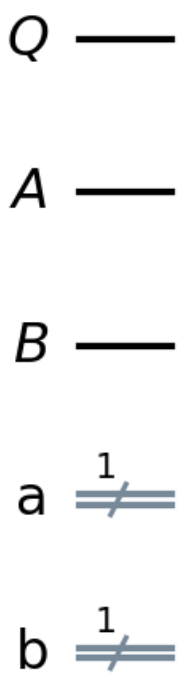


Fig1. All registers – initial information.

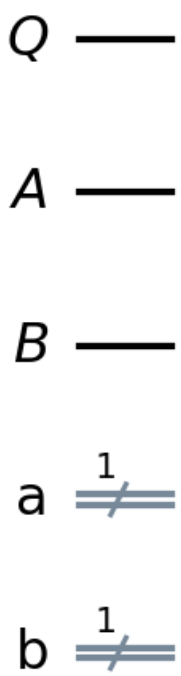


Fig1. All registers – initial information.

The order of three qubits

$$|BAQ\rangle\,.$$

protocol.h(ebit0)

protocol.cx(ebit0, ebit1)

protocol.barrier()

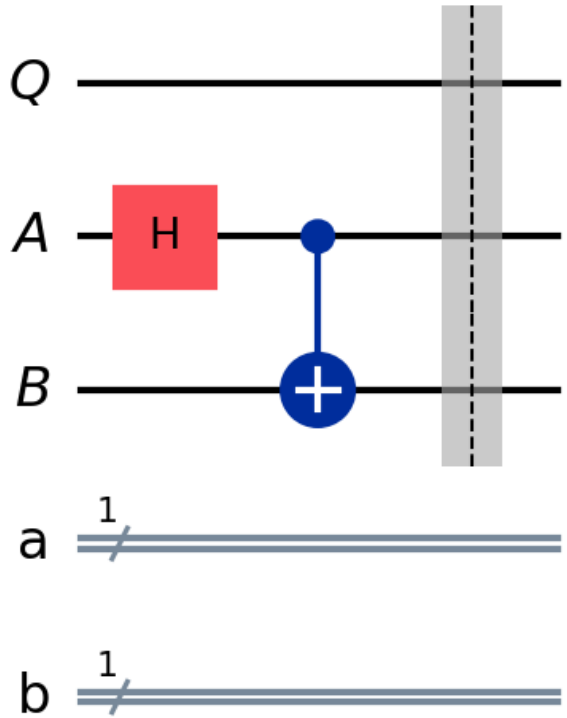


Fig2. The first modification step – generate EPR state of two basic qubits.

Alice modifies her two qubits Q and A (ebit0)

protocol.cx(Q, ebit0)

protocol.h(Q)

protocol.barrier()

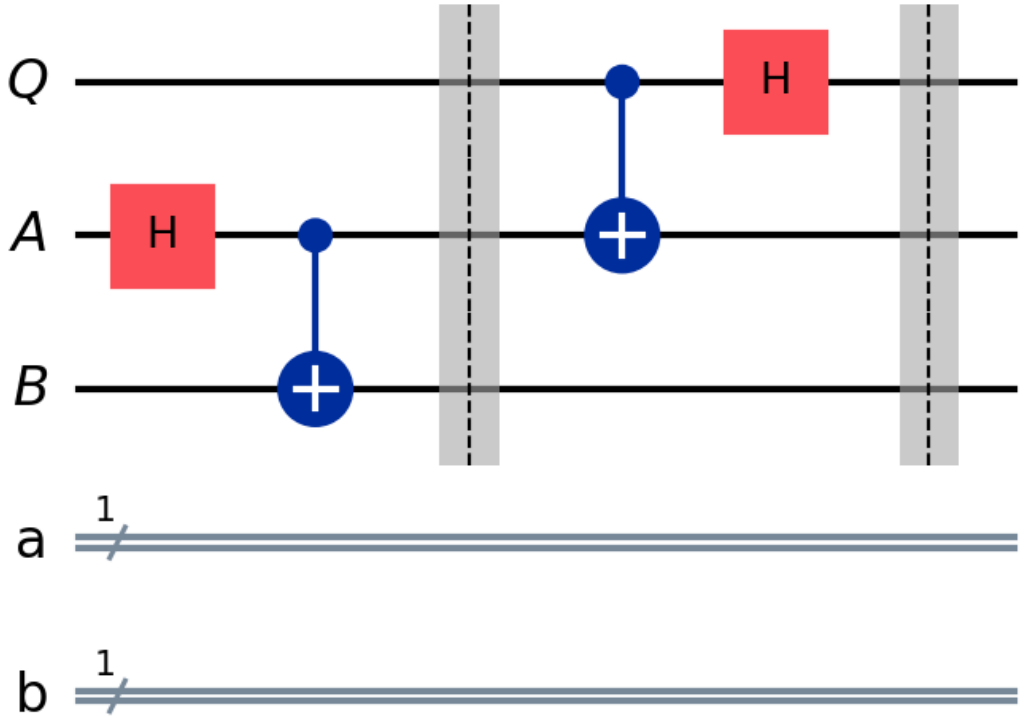
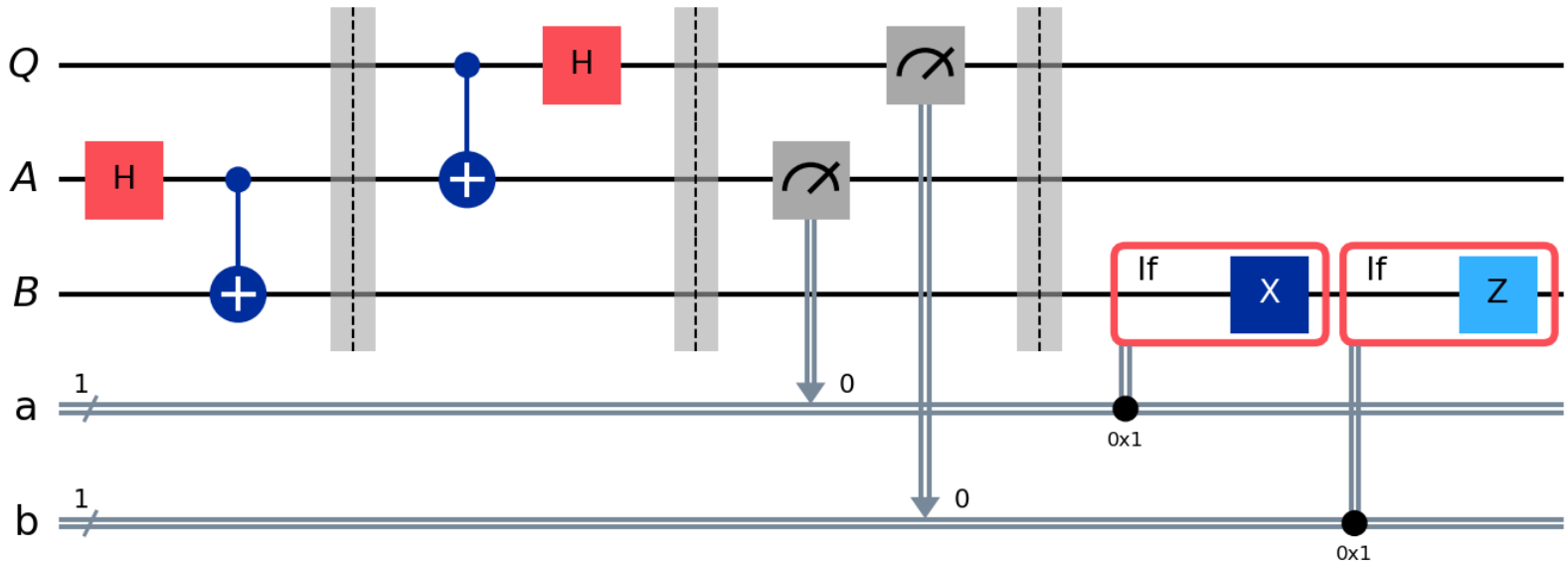


Fig3. The circuit.

```
#Alice measures and sends classical bits to Bob
protocol.measure(ebit0, a)
protocol.measure(qubit, b)
protocol.barrier()
#Bob uses the classical bits to conditionally apply gates
with protocol.if_test((a, 1)):
protocol.x(ebit1)
with protocol.if_test((b, 1)):
protocol.z(ebit1)
```


Q
A
B
a
b
H
H
1
1
0
0
If
X
If
Z
0x1
0x1

Simulation of a real test.

```
random_gate = UGate(
theta=random.random() * 2 * pi,
phi=random.random() * 2 * pi,
lam=random.random() * 2 * pi,
)

print("random_gate: ", random_gate)
print(random_gate.to_matrix())
```

Simulation of a real test.

```
random_gate = UGate(
theta=random.random() * 2 * pi,
phi=random.random() * 2 * pi,
lam=random.random() * 2 * pi,
)

print("random_gate: ", random_gate)
print(random_gate.to_matrix())

# Create a new circuit including the same bits and qubits used in
# the teleportation protocol.

test = QuantumCircuit(qubit, ebit0, ebit1, a, b)

# Start with the randomly selected gate on Q
test.append(random_gate, qubit)
test.barrier()
```

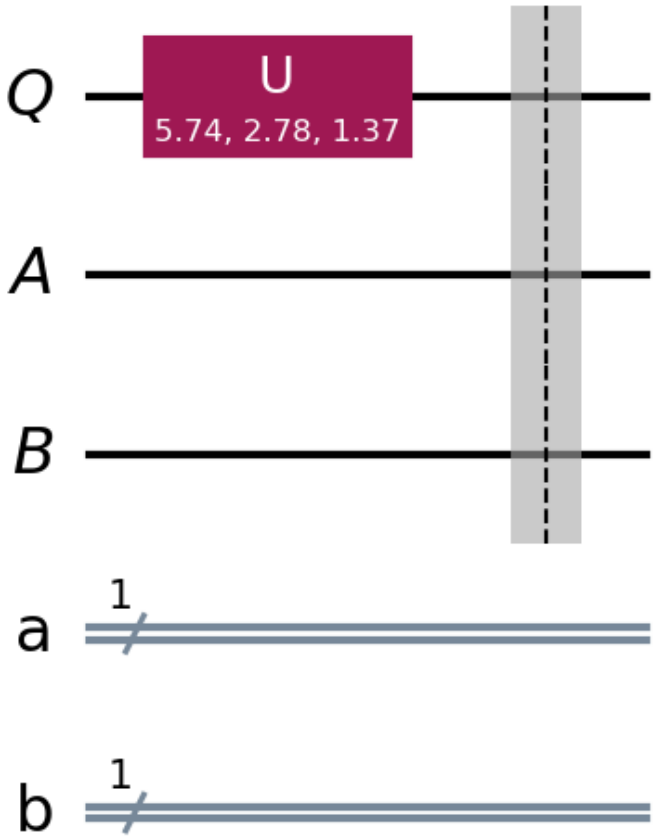


Fig 5. U gate in the test circuit.

```
# Append the entire teleportation protocol from above.
test = test.compose(protocol)
test.barrier()
test.draw(output="mpl", style="iqp", filename='testEntire.png')
```

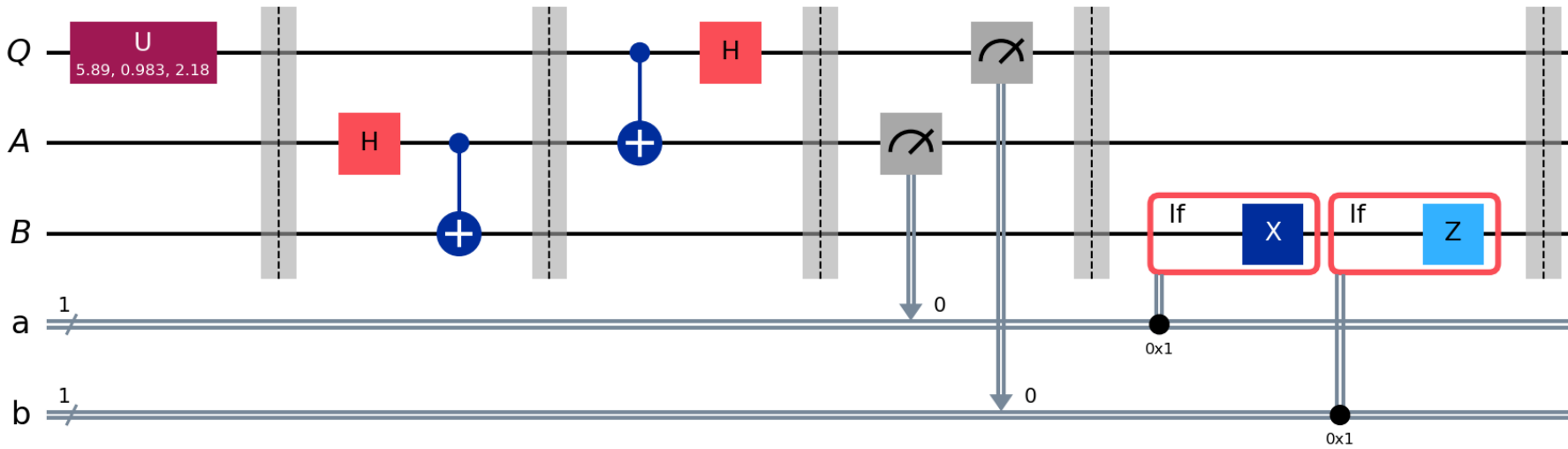


Fig 6. The entire test circuit.

Finally, apply the inverse of the random unitary to B and measure.

```
test.append(random_gate.inverse(), ebit1)

test.draw(output="mpl", style="iqp", filename='testInverse.png')
```

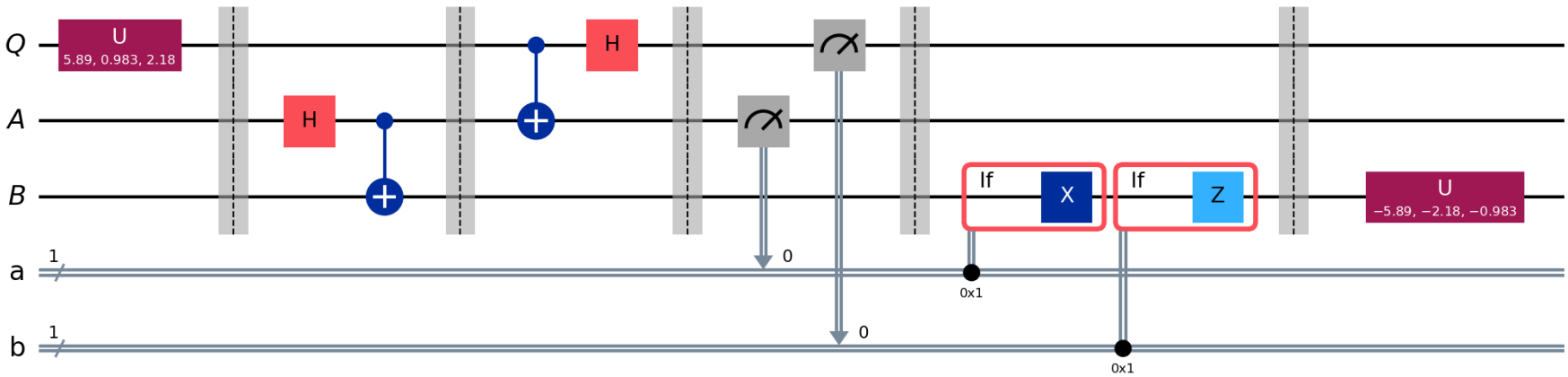


Fig 6. B append the inverse of random unitary.

```
result = ClassicalRegister(1, "Result")

test.add_register(result)

test.measure(ebit1, result)

test.draw(output="mpl", style="iqp",
filename='testMeasure.png')
```

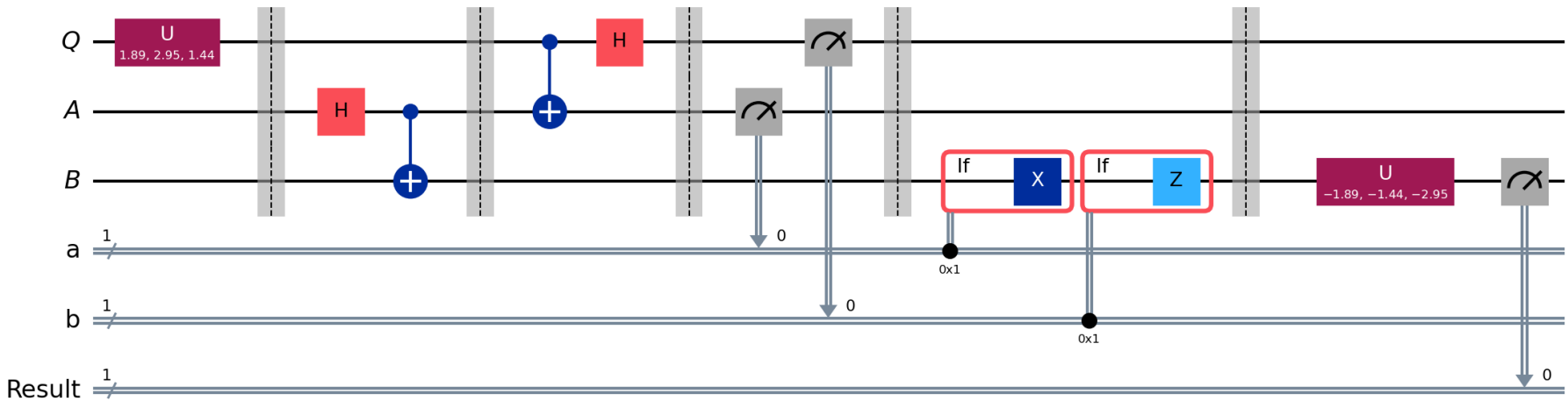


Fig 7. B measured the obtained result.

```
from qiskit_aer import AerSimulator

result = AerSimulator().run(test, shots=512).result()

# the first bit 0 means that the data was transmitted correctly
statistics = result.get_counts()

print(statistics)
```

```
from qiskit_aer import AerSimulator

result = AerSimulator().run(test, shots=512).result()

# the first bit 0 means that the data was transmitted correctly
statistics = result.get_counts()

print(statistics)
```

```
{ '0 1 0': 149, '0 0 0': 107, '0 0 1': 128, '0 1 1': 128 }
```