Programming quantum computers - Quantum teleportation

Initial settings

Loading necessary libraries:

In [1]:
```
import qiskit
from qiskit.visualization import plot_bloch_multivector, plot_histogram, plot_circuit_layout
from math import sin, cos, pi, sqrt
from random import random, seed
import numpy as np
from qiskit import IBMQ
from qiskit.providers.aer import AerSimulator
import matplotlib.pyplot as plt
from qiskit.transpiler.passes import RemoveBarriers
```

To be able to access IBM quantum devices, one needs to initialize it. For the first time one needs to use `save_account()` command with your IBMQ token that can be obtained on the IBMQ page. Later on using `load_account()` is used for the initialization.

In [2]:
```
# IBMQ.save_account("TOKEN")
IBMQ.load_account()
```

Out[2]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

Here are some global definitions that we will use.

In [3]:
```
IBM_DEVICE = "ibm_oslo"
SHOTS = 5000
USE_REPO = True  # If repository is used, it will use JOB_IDS to download data
                 # This works only for the user that performed the computations
                 # When using the file with new account, use the False flag
                 # that will rewrite the JOB_IDS
# JOB_IDS = {"standard":"62b8cd895a1cb09bb0774b0a", "no_cif":"62b8ce680133391af0dddfb8",
#            "no_int_measure":"62b8d044a8fe82a1902c361c", "bad_mapping":"62b8d0af6a05a54cd48d
#            "best":"62b8d10740f154a84cafbe5e"}
JOB_IDS = {'standard': '62b973b1013339a3d8dde252',
           'no_cif': '62b973da40f1544890afc0e3',
           'no_int_measure': '62b973f7bd18a2fa9b327688',
           'bad_mapping': '62b9741c013339604adde253',
           'best': '62b9753c40f154d1ddafc0ea'}

# Device to be simulated
DEVICE = IBMQ.get_provider().get_backend(IBM_DEVICE)

# Extracted simulator for the device
DEVICE_SIM = AerSimulator.from_backend(DEVICE)

# Perfect simulator
IDEAL_SIM = qiskit.Aer.get_backend('qasm_simulator')
```

Bits and pieces we will use

Our tasks will be very specific - we will design our experiments so that the preferred outcome will be always `0`. To this end we design all our helper functions.

Experiments classes

The following class just simplifies manipulation of experiments. How to perform an experiment is also shown later.

In [4]:
```
class Experiment:
    """
    Experiment manipulation class.

    Attributes:
     - name: name of the experiment
     - circuit: circuit to be executed
     - shots: number of shots in  the execution
     - ideal: NumPy array of success probabilities (measuring 0) using ideal simulator
     - sim: NumPy array of success probabilities (measuring 0) using device simulator [genera
     - result: NumPy array of success probabilities (measuring 0) [or None]
     - job_id: IBMQ job ID for data retrieval [or None]

    Methods:
     - get_device_results: (i) if job_id is None, experiment is submitted to the device, else
                           (ii) if results are None, job is retrieved from the device
     - status: prints status of the experiment
    """
    def __init__(self, name, circuit, shots=SHOTS):
        self.name = name
        self.circuit = circuit
        self.shots = shots

        # job can be submitted with (usually up to 100) circuits [depends on the device]
        job = qiskit.execute([self.circuit] * 100, DEVICE_SIM, shots=self.shots)
        self.sim = Experiment._extract_bobs_success(job.result().get_counts())

        job = qiskit.execute([self.circuit] * 100, IDEAL_SIM, shots=self.shots)
        self.ideal = Experiment._extract_bobs_success(job.result().get_counts())

        self.result = None
        self.job_id = JOB_IDS.get(self.name, None) if USE_REPO else None

    # Extracts success rate (of measuring 0 on Bob's qubit)
    @staticmethod
    def _extract_bobs_success(counts_lst, pos=0):
        succ = []
        for counts in counts_lst:
            bob_counts = {"0":0, "1":0}
            for key in counts:
                bob_key = key[pos]
                bob_counts[bob_key] += counts[key]
            succ.append(bob_counts["0"] / (bob_counts["0"] + bob_counts["1"]))
        return np.array(succ)

    # (i) if job_id is None, experiment is submitted to the device, else
    # (ii) if results are None, job is retrieved from the device
    def get_device_results(self):
        if self.job_id is None:
            job = qiskit.execute([self.circuit] * 100, DEVICE, shots=SHOTS)
            self.job_id = job.job_id()
            JOB_IDS[self.name] = self.job_id
            print(f"Job submitted to {IBM_DEVICE} under ID {job.job_id()}")
            return

        if self.result is None:
            job = DEVICE.retrieve_job(self.job_id)
            self.result = Experiment._extract_bobs_success(job.result().get_counts())
            print(f"Precomputed job successfully downloaded from repository")

    # Prints status of the experiment
    def status(self, old=True):
```

```
        print(f"EXPERIMENT: {self.name}")
        print(f"Ideal success rate: {100 * self.ideal.mean():.2f} ± {100 * self.ideal.std():.
        print(f"Simulated device: {100 * self.sim.mean():.2f} ± {100 * self.sim.std():.2f} %"
        if self.result is not None:
            print(f"Actual IBMQ device ({IBM_DEVICE}): {100 * self.result.mean():.2f} ± {100
        else:
            print(f"IBMQ runs not available.")
        depth = self.circuit.depth()
        ops = self.circuit.count_ops()
        double = ops.get("cx", 0) + ops.get("cz", 0)
        meas = ops["measure"]
        single = sum(ops[x] for x in ops if x not in ["barrier", "measure", "cx", "cz"])
        print(f"The circuit has depth {depth}, contains {single} one-qubit operations, {doubl
```

In [5]:
```
class Experiments:
    """
    Class for the collection of experiments

    Attributes:
     - experiments: dictionary of experiments [name:experiment]

    Methods:
     - add: adds experiment if not in experiments
     - remove: removes experiment from experiments by name
     - get: returns experiment by name
     - status: prints statuses of all experiments
    """
    def __init__(self):
        self.experiments = {}

    def add(self, exp):
        assert exp.name not in self.experiments.keys()
        self.experiments[exp.name] = exp

    def remove(self, name):
        if name in self.experiments.keys():
            del self.experiments[name]

    def get(self, name):
        return self.experiments.get(name, None)

    def status(self):
        for e in self.experiments.values():
            e.status()
            print()
```

Random state preparator

We want to prepare a function that will return a Qiskit gate for random state preparation. We shall use U_3 unitary with randomly chosen angles θ, ϕ and λ. As we intend to test the correctness of the teleportation, the function will return both U and $U^\dagger$.

In [6]:
```
def random_gate(s=None):
    seed(s)

    # Choose the angles randomly
    theta = 2*pi*random()
    phi = pi*random()
    lam = pi*random()

    # Composing the gate U
    c = qiskit.QuantumCircuit(1)
    c.u(theta, phi, lam, 0)

    # Composing the gate U^\dagger
    c_dag = qiskit.QuantumCircuit(1)
```

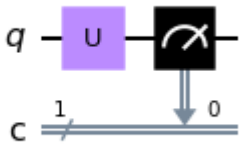
```
    c_dag.u(-theta, -lam, -phi, 0)

    seed()
    return c.to_gate(label="U"), c_dag.to_gate(label="U+")
```

Simple experiments to see how Qiskit works

The circuit runs on a qubit register initialized to state $|0\rangle$. Then we perform the computation by using various quantum gates (similarly as in the classical computation). To get the results we perform measurements on chosen qubits. Measurements are in general stochastic producing in each run either a `0` or a `1` for each qubit.

In [7]:
```
u, u_dag = random_gate()

# Define the circuit
c = qiskit.QuantumCircuit(1, 1)
c.append(u, [0])

# Append measurements
c.measure(0, 0)

c.draw(output="mpl")
```

Out[7]:



Circuits are submitted as jobs to given backends (devices) and these are run several times to get the statistics. One can submit also a list of circuits.

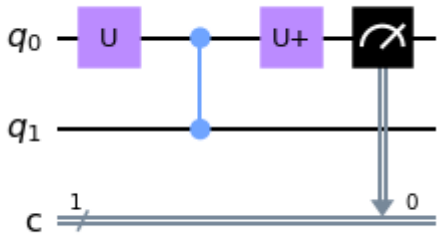
In [8]:
```
job = qiskit.execute(c, IDEAL_SIM, shots=SHOTS) # Alternative is run method for backends
counts = job.result().get_counts()
print(counts)
```

```
{'0': 4994, '1': 6}
```

Let us see what happens in a slightly more involved case. We add one `CZ` gate and use $U^\dagger$ to return the first qubit to its original state - if we have somewhere state $|\psi\rangle$ prepared by U, then applying $U^\dagger$ on it shall return it to state $|0\rangle$ and so we can measure the success in this case by the probability of getting outcome `0`.

In [9]:
```
# Define the circuit
c = qiskit.QuantumCircuit(2, 1)
c.append(u, [0])
c.cz(0, 1)
c.append(u_dag, [0])

# Append measurements
c.measure(0, 0)

c.draw(output="mpl")
```

Out[9]:



Each specific device uses its own gates set that it can perform. And so if we submit the circuit for the computation, it undergoes many steps before it is fed to the quantum device:

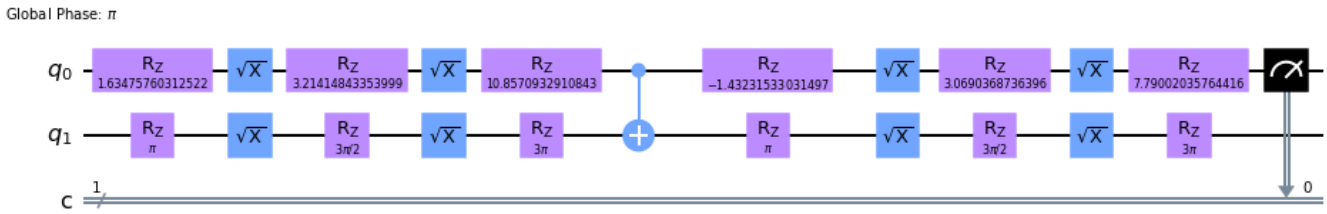
- **routing:** fitting the circuit to chosen device
- **decomposition:** replacing gates with the native set
- **optimization:** reducing the complexity of fitted circuit

These steps are collectively called *transpilation*, but the process is more complex than that c.f. this link.

One of the most important ways of estimating complexity of the circuit is to look at Let's see what happens to the circuit when we decompose it.

In [10]:
```
c_decomposed = qiskit.transpile(c, basis_gates=["cx", "id", "rz", "sx", "x"], optimization_le
c_decomposed.draw(output="mpl")
```

Out[10]:



Running the circuit on ideal simulator gives following results:

In [11]:
```
job = qiskit.execute(c_decomposed, IDEAL_SIM, shots=SHOTS)
counts = job.result().get_counts()
print(counts)
```

```
{'0': 5000}
```

We can make use of the `Experiment` class where we will automatically extract success rates not only for the ideal simulator, but also for simulated device (with errors) and possibly with submission to actual device.

In [12]:
```
e = Experiment("test", c_decomposed)
```

In [13]:
```
e.status()
```

```
EXPERIMENT: test
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 99.02 ± 0.15 %
IBMQ runs not available.
The circuit has depth 12, contains 20 one-qubit operations, 1 two-qubit operations and 1 meas
urements.
```

Teleportation

Teleportation is one of the basic quantum tasks where a state is "teleported" from one qubit to another just by using entanglement and classical communication. Before immersing in the implementation, let us

initialize the `Experiments` class and the seed for random number generation in the `random_gate`, so that the results will be consistent.

In [14]:
```
rnd_seed = 42
```

In [15]:
```
exp = Experiments()
```

Standard circuit

Standard teleportation circuit consists of Alice with two qubits and Bob having single qubit. Before Alice wants to send her state to Bob, the two of them make a maximally entangled Bell state between one of Alice's qubits and Bob's qubit. Alice and Bob are then free to go as far as they want. When Alice wants to send her qubit, she performs a specific (Bell) measurement on her two qubits - one with the unknown state and the other one, which shares entanglement with Bob's qubit. Alice will get two bits of information that she sends to Bob. Bob, after receiving the two bits (which are completely random) adjusts his state and in the end he is supposed to get the state that Alice was sending. Alice's qubits are in a random state of the measurement basis.

The circuit thus consists of four parts:

1. Creating entanglement
2. Preparation of the teleported state
3. Bell measurement on Alice's qubits
4. Correction of Bob's state based on Alice's result

We will implement also additional step that we discussed above - as expect the final Bob's state to be prepared by unitary U, we will "undo" it by $U^\dagger$ and the measurement should be in state `0`. And so the fifth part of the circuit is:

1. Rotating the state to the measurement basis and measure

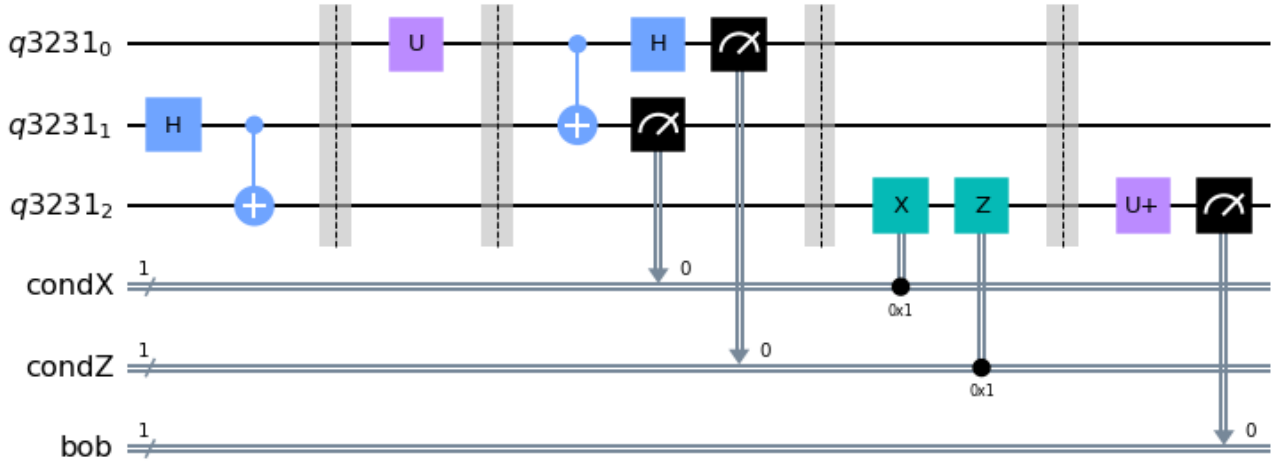
In [16]:
```
def teleport_standard(u, u_dag):
    qreg = qiskit.QuantumRegister(3)

    # We will use three separate classical registers (bits)
    # for the two bits of information Alice sends to Bob
    # and the third for potential measurement of Bob
    cregx = qiskit.ClassicalRegister(1, name="condX")
    cregz = qiskit.ClassicalRegister(1, name="condZ")
    cregbob = qiskit.ClassicalRegister(1, name="bob")

    teleport = qiskit.QuantumCircuit(qreg, cregx, cregz, cregbob)

    # Step 1: Creating entanglement
    teleport.h(1)
    teleport.cx(1, 2)
    teleport.barrier()

    # Step 2: Preparation of the telported state
    # (random state will be passed in the argument of the function)
    teleport.append(u, [0])
    teleport.barrier()

    # Step 3: Bell measurement
    teleport.cx(0, 1)
    teleport.h(0)
    teleport.measure(0, cregz[0])
    teleport.measure(1, cregx[0])
    teleport.barrier()
```

```
    # Step 4: Correcting state on Bob;s side
    teleport.x(2).c_if(cregx, 1)
    teleport.z(2).c_if(cregz, 1)
    teleport.barrier()

    # Step 5: Bob's measurement
    # (If everything is correct, only 0s are expected)
    teleport.append(u_dag, [2])
    teleport.measure(2, cregbob[0])

    return teleport
```

Let's draw the full teleportation circuit with some random state for Alice to send.

In [17]:
```
u, u_dag = random_gate(rnd_seed)
c_standard = teleport_standard(u, u_dag)
c_standard.draw(output="mpl")
```

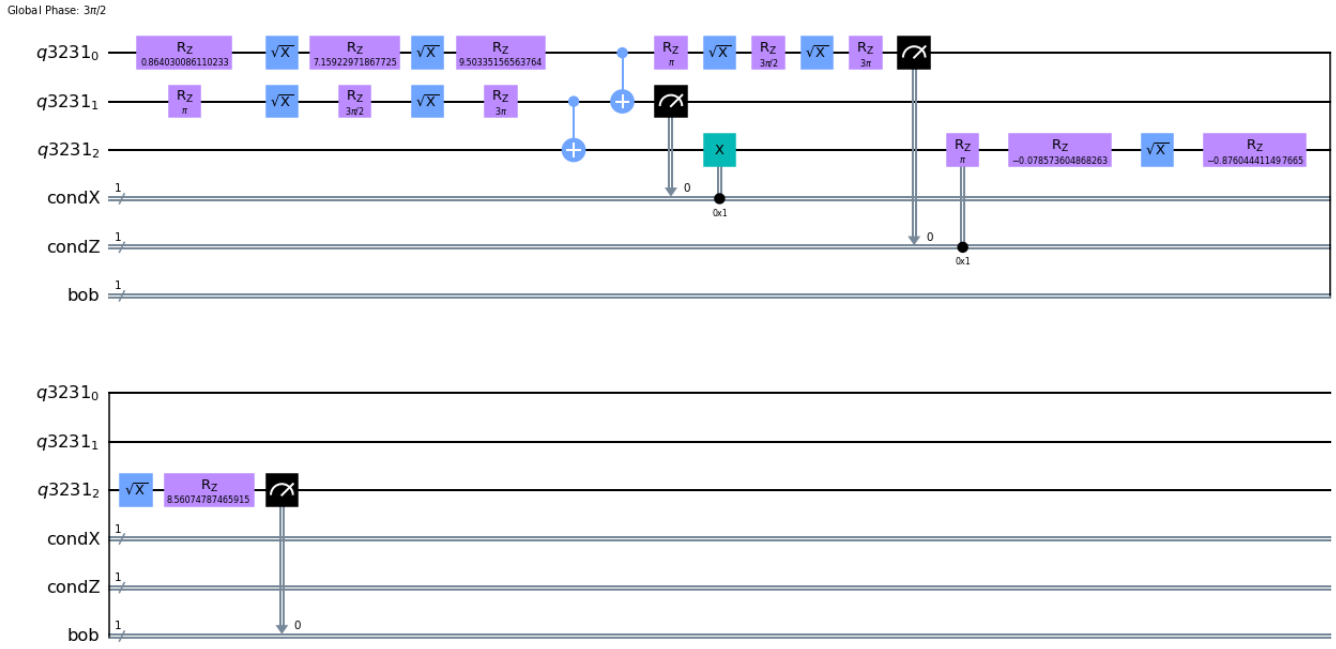
Out[17]:



Barriers in the picture have both visual fuinction, but also practical. Therefore, we will remove them before using the circuit. We will also decompose the circuit to the native base set of IBM quantum devices and for now we will leave optimization and routing to the standard setting.

In [18]:
```
exp.add(Experiment("standard", qiskit.transpile(RemoveBarriers()(c_standard),
                                                basis_gates=["cx", "id", "rz", "sx", "x"],
                                                optimization_level=0)))
exp.experiments["standard"].circuit.draw(output="mpl")
```

Out[18]:



Let us see what the quantum computation on real device gives:

In [19]:
```
exp.get("standard").get_device_results()
```

```
---------------------------------------------------------------------------
IBMQJobFailureError                       Traceback (most recent call last)
~\AppData\Local\Temp\1/ipykernel_21860/893574494.py in <module>
----> 1 exp.get("standard").get_device_results()

~\AppData\Local\Temp\1/ipykernel_21860/1700584260.py in get_device_results(self)
     56         if self.result is None:
     57             job = DEVICE.retrieve_job(self.job_id)
---> 58             self.result = Experiment._extract_bobs_success(job.result().get_counts())
     59             print(f"Precomputed job successfully downloaded from repository")
     60

~\Miniconda3\envs\qiskit\lib\site-packages\qiskit\providers\ibmq\job\ibmqjob.py in result(sel
f, timeout, wait, partial, refresh)
    288                 else:
    289                     error_message = ": " + error_message
--> 290                 raise IBMQJobFailureError(
    291                     'Unable to retrieve result for job {}. Job has failed{}'.format(
    292                         self.job_id(), error_message))

IBMQJobFailureError: 'Unable to retrieve result for job 62b973b1013339a3d8dde252. Job has fai
led: Instruction bfunc is not supported. Error code: 7001.'
```

Running this circuit on IBM quantum device fails (situation on 27/06/2022) as it does not support classical feedback. It has been just few months since IBMQ started supporting intermediate measurements on some of the devices. So now we can use the fact that the quantum bit that was measured will be in the same (classical) state as was the classical outcome of the measurement; we can do the measurements, but will use the corresponding qubit as the control.

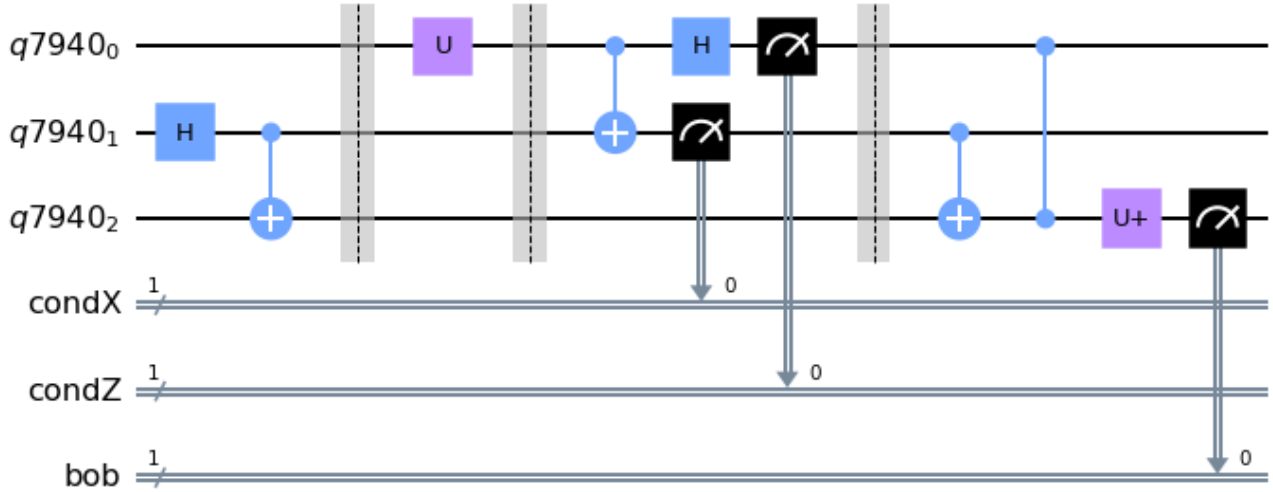
Removing the classical control

In [20]:
```
def teleport_no_cif(u, u_dag):
    qreg = qiskit.QuantumRegister(3)

    # We will use three separate classical registers (bits)
    # for the two bits of information Alice sends to Bob
    # and the third for potential measurement of Bob
    cregx = qiskit.ClassicalRegister(1, name="condX")
    cregz = qiskit.ClassicalRegister(1, name="condZ")
    cregbob = qiskit.ClassicalRegister(1, name="bob")

    teleport = qiskit.QuantumCircuit(qreg, cregx, cregz, cregbob)

    # Step 1: Creating entanglement
    teleport.h(1)
    teleport.cx(1, 2)
    teleport.barrier()

    # Step 2: Preparation of the telported state
    # (random state will be passed in the argument of the function)
    teleport.append(u, [0])
    teleport.barrier()

    # Step 3: Bell measurement
    teleport.cx(0, 1)
    teleport.h(0)
    teleport.measure(0, cregz[0])
    teleport.measure(1, cregx[0])
    teleport.barrier()

    # Step 4: Correcting state on Bob;s side
```

```
    teleport.cx(1, 2)
    teleport.cz(0, 2)

    # Step 5: Bob's measurement
    # (If everything is correct, only 0s are expected)
    teleport.append(u_dag, [2])
    teleport.measure(2, cregbob[0])

    return teleport
```

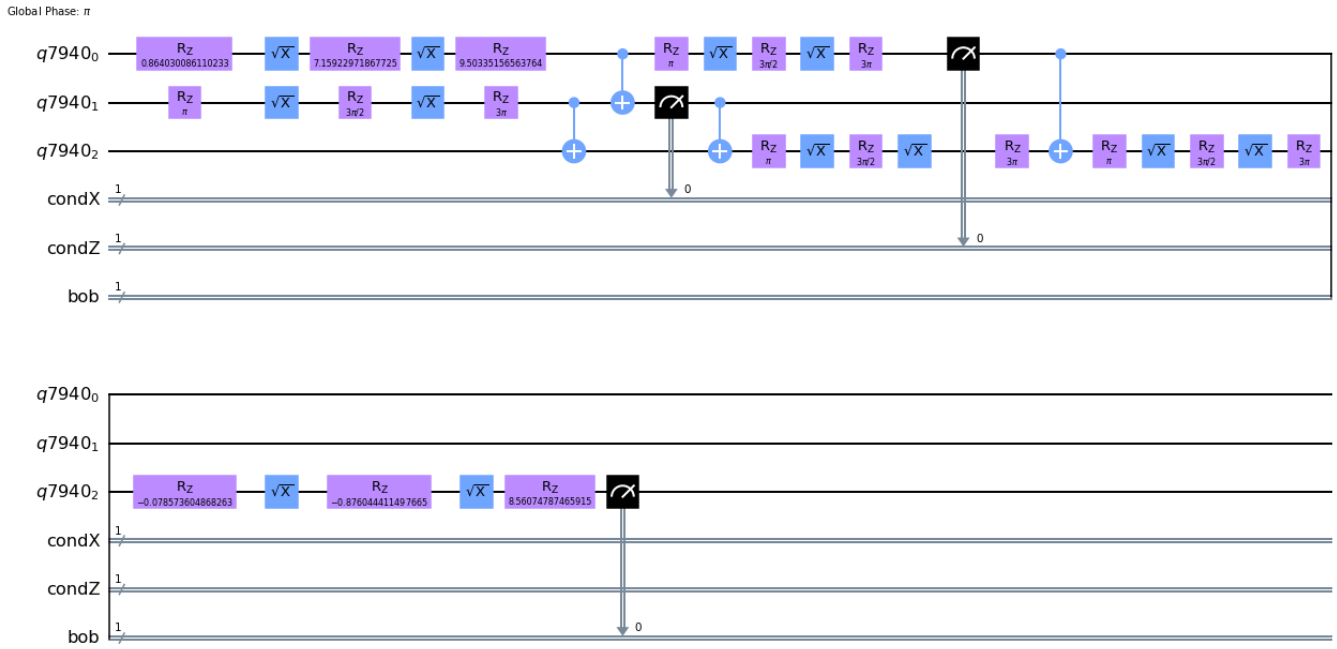
In [21]:
```
c_no_cif = teleport_no_cif(u, u_dag)
c_no_cif.draw(output="mpl")
```

Out[21]:



In [22]:
```
exp.add(Experiment("no_cif", qiskit.transpile(RemoveBarriers()(c_no_cif),
                                              basis_gates=["cx", "id", "rz", "sx", "x"],
                                              optimization_level=0)))
exp.experiments["no_cif"].circuit.draw(output="mpl")
```

Out[22]:



In [23]:
```
exp.get("no_cif").get_device_results()
```

```
Precomputed job successfully downloaded from repository
```

The experiment in this case was successful and we can now compare the results. The circuit implementable on QPUs is now longer and is suspected to give worse results than the standard circuit would. We see that the estimate is very optimistic, as the actual device gives a success rate by 10% worse than the simulation.

In [24]:
```
exp.status()
```

```
EXPERIMENT: standard
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 97.24 ± 0.21 %
IBMQ runs not available.
The circuit has depth 20, contains 22 one-qubit operations, 2 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_cif
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.40 ± 0.53 %
Actual IBMQ device (ibm_oslo): 85.85 ± 1.61 %
The circuit has depth 26, contains 30 one-qubit operations, 4 two-qubit operations and 3 meas
urements.
```

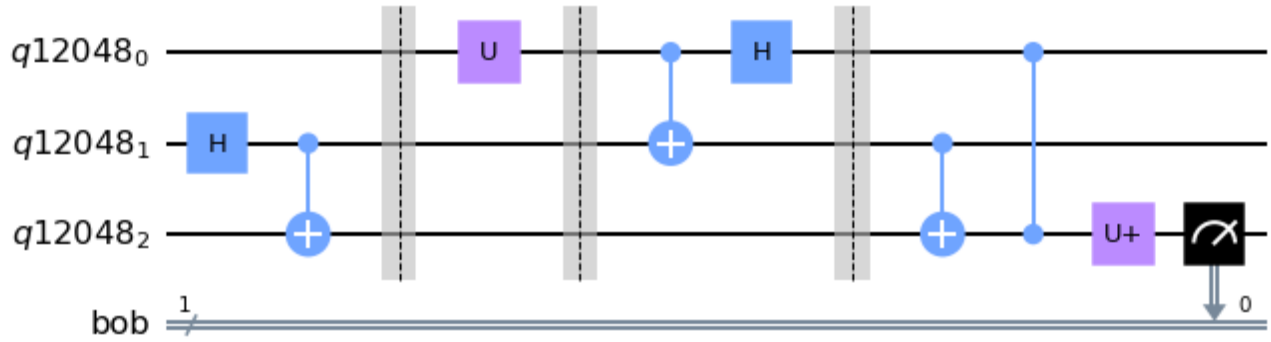
Without intermediate measurements

We can simplify the circuit even further also it no longer corresponds to the strict conditions of the definition above. The classical communication is replaced by quantum feedback; one can show that the measurements are irrelevant.

In [25]:
```
def teleport_no_int_measure(u, u_dag):
    qreg = qiskit.QuantumRegister(3)

    # We will use three separate classical registers (bits)
    # for the two bits of information Alice sends to Bob
    # and the third for potential measurement of Bob
    cregbob = qiskit.ClassicalRegister(1, name="bob")

    teleport = qiskit.QuantumCircuit(qreg, cregbob)

    # Step 1: Creating entanglement
    teleport.h(1)
    teleport.cx(1, 2)
    teleport.barrier()

    # Step 2: Preparation of the telported state
    # (random state will be passed in the argument of the function)
    teleport.append(u, [0])
    teleport.barrier()

    # Step 3: Bell "measurement"
    teleport.cx(0, 1)
    teleport.h(0)
    teleport.barrier()

    # Step 4: Correcting state on Bob;s side
    teleport.cx(1, 2)
    teleport.cz(0, 2)

    # Step 5: Bob's measurement
    # (If everything is correct, only 0s are expected)
    teleport.append(u_dag, [2])
    teleport.measure(2, cregbob[0])

    return teleport
```

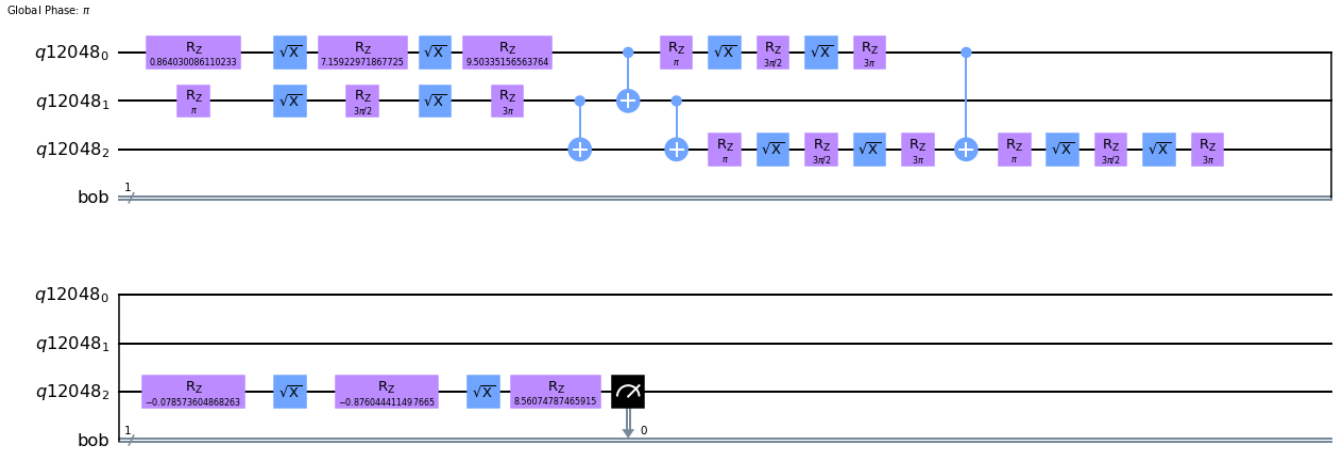
In [26]:
```
c_no_int_measure = teleport_no_int_measure(u, u_dag)
c_no_int_measure.draw(output="mpl")
```

Out[26]:



In [27]:
```
exp.add(Experiment("no_int_measure", qiskit.transpile(RemoveBarriers()(c_no_int_measure),
                                                      basis_gates=["cx", "id", "rz", "sx", "x
                                                      optimization_level=0)))
exp.experiments["no_int_measure"].circuit.draw(output="mpl")
```

Out[27]:



In [28]:
```
exp.get("no_int_measure").get_device_results()
```

```
Precomputed job successfully downloaded from repository
```

In [29]:
```
exp.status()
```

```
EXPERIMENT: standard
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 97.24 ± 0.21 %
IBMQ runs not available.
The circuit has depth 20, contains 22 one-qubit operations, 2 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_cif
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.40 ± 0.53 %
Actual IBMQ device (ibm_oslo): 85.85 ± 1.61 %
The circuit has depth 26, contains 30 one-qubit operations, 4 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_int_measure
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.35 ± 0.50 %
Actual IBMQ device (ibm_oslo): 86.74 ± 1.34 %
The circuit has depth 25, contains 30 one-qubit operations, 4 two-qubit operations and 1 meas
urements.
```

We can see that the removal of measurements did not change anything. This need not be the case in general, as on some QPUs the measurements are much longer than gates and the decoherence effects become more pronounced. On `ibm_oslo` we have following:

In [30]:
```
p = DEVICE.properties()
```

```
single = p.gates[21]
print(f"Gate {single.gate} has gate length {single.parameters[-1].value:.1f} {single.paramete
double = p.gates[39]
print(f"Gate {double.gate} has gate length {double.parameters[-1].value:.1f} {double.paramete
reset = p.gates[-1]
print(f"Gate {reset.gate} has gate length {reset.parameters[-1].value:.1f} {reset.parameters[
measure = p.readout_length(6) * 1e9
print(f"Measurement has length {measure:.1f} ns")
```

```
Gate x has gate length 35.6 ns
Gate cx has gate length 412.4 ns
Gate reset has gate length 960.0 ns
Measurement has length 910.2 ns
```

But on `ibmq_manila` we would probably get worse results as there we have:

In [31]:
```
p = IBMQ.get_provider().get_backend("ibmq_manila").properties()
single = p.gates[16]
print(f"Gate {single.gate} has gate length {single.parameters[-1].value:.1f} {single.paramete
double = p.gates[21]
print(f"Gate {double.gate} has gate length {double.parameters[-1].value:.1f} {double.paramete
reset = p.gates[-1]
print(f"Gate {reset.gate} has gate length {reset.parameters[-1].value:.1f} {reset.parameters[
measure = p.readout_length(0) * 1e9
print(f"Measurement has length {measure:.1f} ns")
```

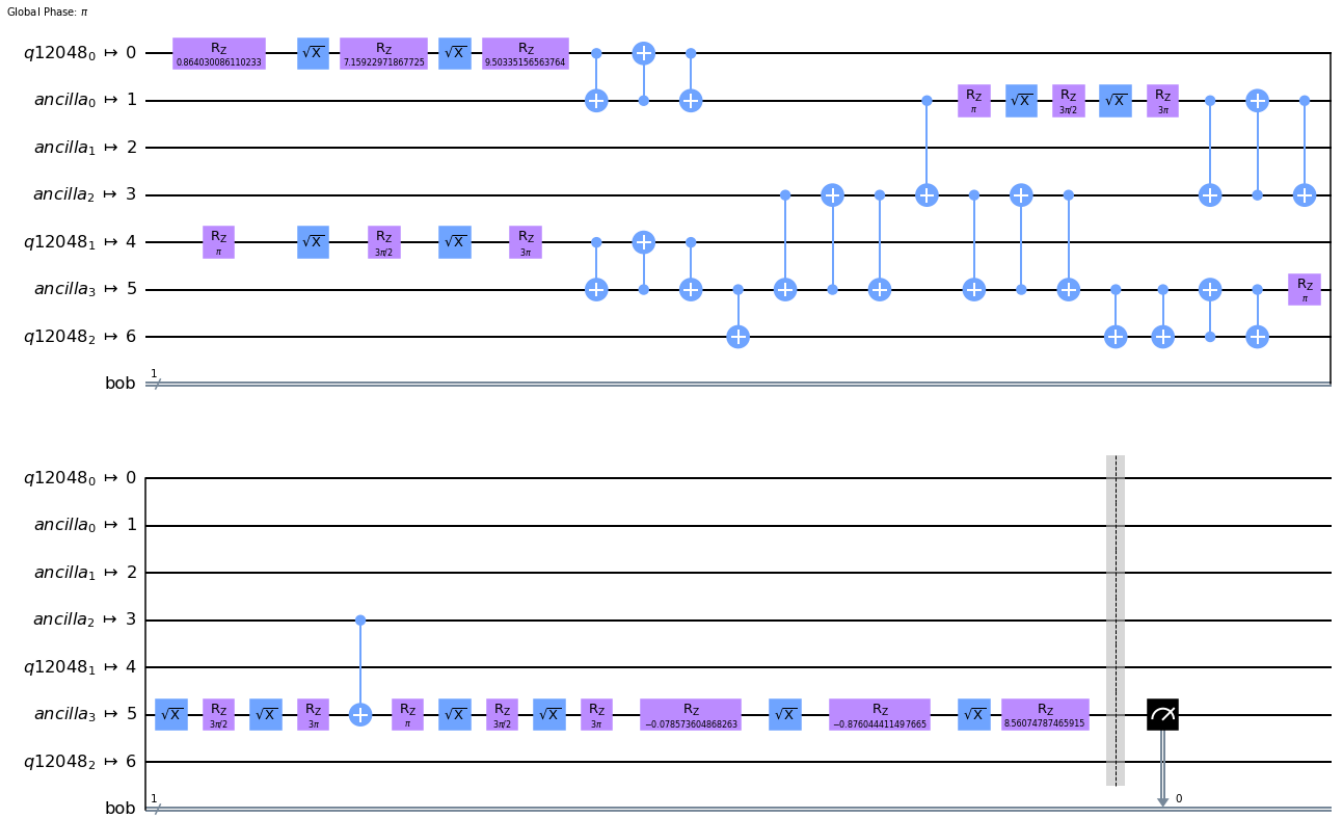
```
Gate x has gate length 35.6 ns
Gate cx has gate length 334.2 ns
Gate reset has gate length 5514.7 ns
Measurement has length 5351.1 ns
```

Including architecture and bad mapping

Part of transpilation is also **routing.** That is a way of mapping logical qubits to physical qubits. It makes a lot of difference. See for example this mapping: `qr[0]-> QB0` , `qr[1]-> QB4` , `qr[2] -> QB6`

In [32]:
```
c_bad_mapping = qiskit.transpile(RemoveBarriers()(c_no_int_measure), backend=DEVICE,
                                 optimization_level=0, initial_layout=[0, 4, 6])
c_bad_mapping.draw(output="mpl")
```

Out[32]:



In [33]:
```
exp.add(Experiment("bad_mapping", c_bad_mapping))
```

In [34]:
```
exp.get("bad_mapping").get_device_results()
```

```
Precomputed job successfully downloaded from repository
```

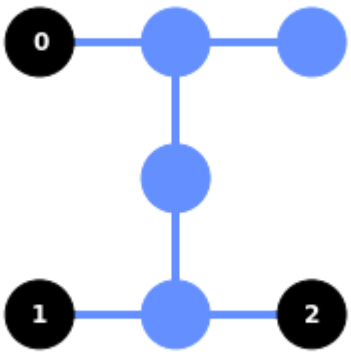
In [35]:
```
exp.status()
```

```
EXPERIMENT: standard
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 97.24 ± 0.21 %
IBMQ runs not available.
The circuit has depth 20, contains 22 one-qubit operations, 2 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_cif
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.40 ± 0.53 %
Actual IBMQ device (ibm_oslo): 85.85 ± 1.61 %
The circuit has depth 26, contains 30 one-qubit operations, 4 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_int_measure
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.35 ± 0.50 %
Actual IBMQ device (ibm_oslo): 86.74 ± 1.34 %
The circuit has depth 25, contains 30 one-qubit operations, 4 two-qubit operations and 1 meas
urements.

EXPERIMENT: bad_mapping
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 91.67 ± 0.36 %
Actual IBMQ device (ibm_oslo): 80.46 ± 0.59 %
The circuit has depth 37, contains 30 one-qubit operations, 22 two-qubit operations and 1 mea
surements.
```

These results are much worse since the routing is very bad. We made use of following qubits:

In [36]:
```
plot_circuit_layout(c_bad_mapping, DEVICE)
```

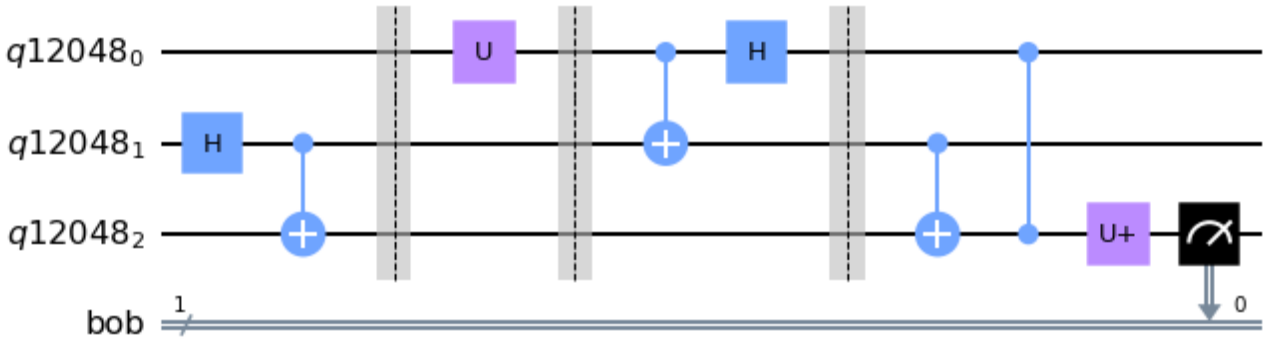
Out[36]:



Routing is very important and can make a big difference. Problematic is, however, if the original circuit has CNOT "loops":

In [37]:
```
c_no_int_measure.draw(output="mpl")
```

Out[37]:

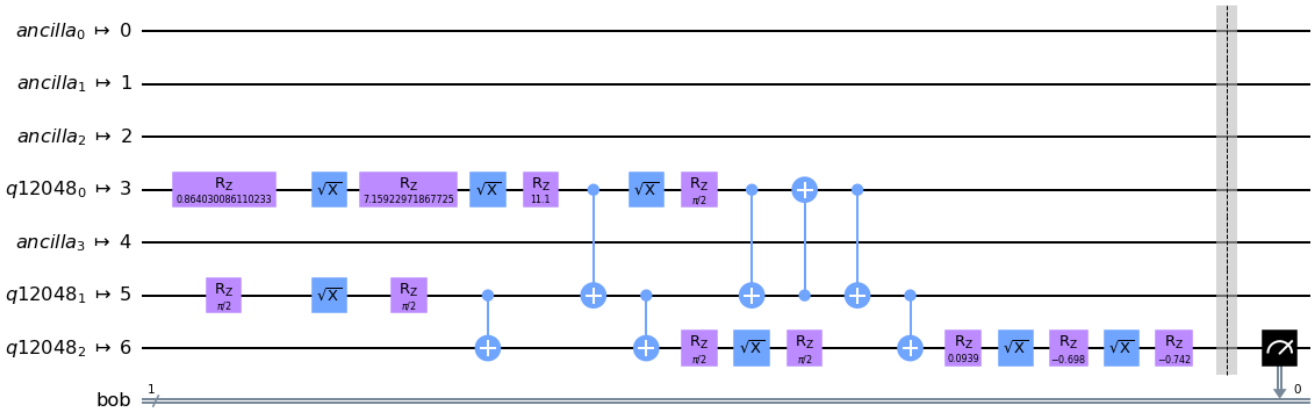


Good mapping and optimization

Finally, after routing and decomposing, one can use various methods to optimize the circuit (single qubit gates might be joined, etc.). Let us have a look at the best automatic optimization in Qiskit (it is probabiistic).

In [38]:
```
c_best = qiskit.transpile(RemoveBarriers()(c_no_int_measure), backend=DEVICE,
                          optimization_level=3)
c_best.draw(output="mpl")
```

Out[38]:



In [39]:
```
exp.add(Experiment("best", c_best))
```

In [40]:
```
exp.get("best").get_device_results()
```

```
Precomputed job successfully downloaded from repository
```

Wrap-up

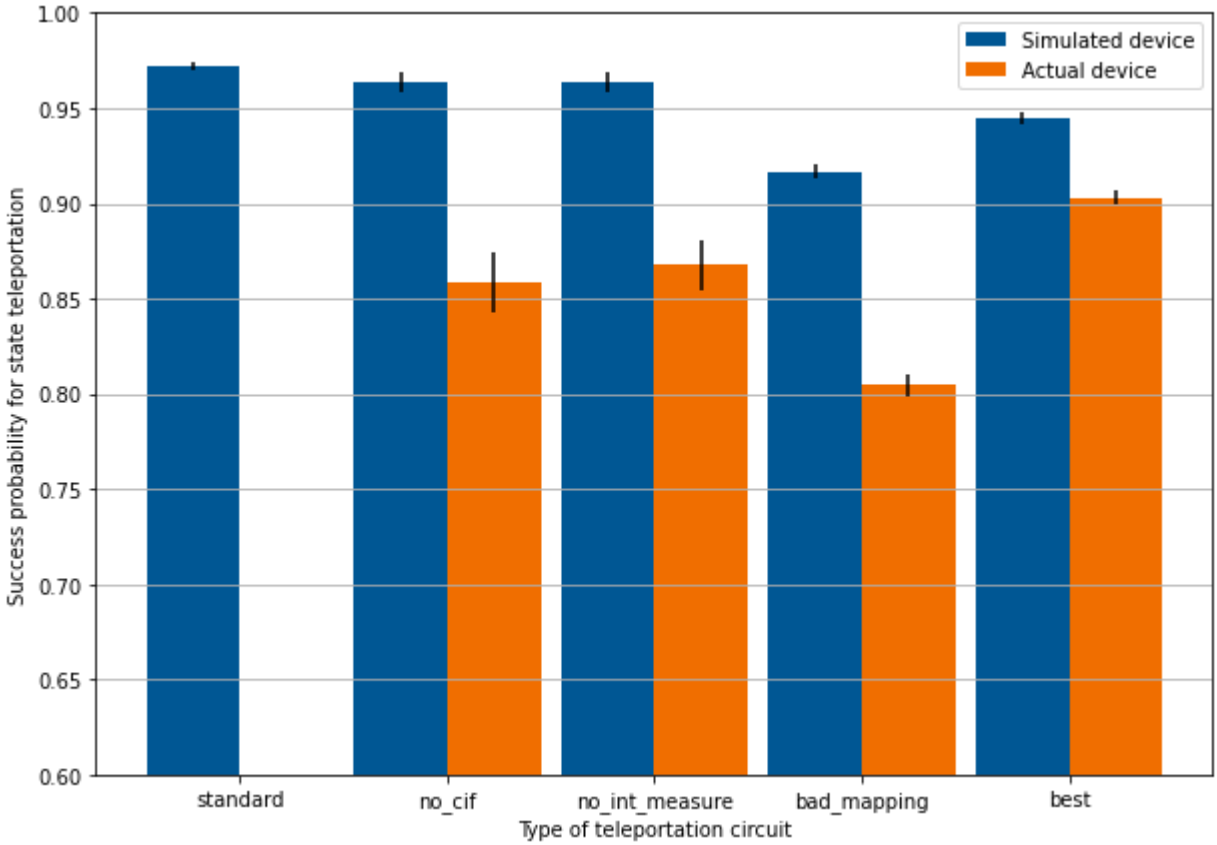
Let us have a look at all the data we produced:

In [41]:
```
exp.status()
```

```
EXPERIMENT: standard
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 97.24 ± 0.21 %
IBMQ runs not available.
The circuit has depth 20, contains 22 one-qubit operations, 2 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_cif
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.40 ± 0.53 %
Actual IBMQ device (ibm_oslo): 85.85 ± 1.61 %
The circuit has depth 26, contains 30 one-qubit operations, 4 two-qubit operations and 3 meas
urements.

EXPERIMENT: no_int_measure
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 96.35 ± 0.50 %
Actual IBMQ device (ibm_oslo): 86.74 ± 1.34 %
The circuit has depth 25, contains 30 one-qubit operations, 4 two-qubit operations and 1 meas
urements.

EXPERIMENT: bad_mapping
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 91.67 ± 0.36 %
Actual IBMQ device (ibm_oslo): 80.46 ± 0.59 %
The circuit has depth 37, contains 30 one-qubit operations, 22 two-qubit operations and 1 mea
surements.

EXPERIMENT: best
Ideal success rate: 100.00 ± 0.00 %
Simulated device: 94.45 ± 0.31 %
Actual IBMQ device (ibm_oslo): 90.29 ± 0.37 %
The circuit has depth 18, contains 18 one-qubit operations, 7 two-qubit operations and 1 meas
urements.
```

In [42]:

```python
def plot(exp):
    xlabels = list(exp.experiments.keys())
    x = np.array(range(len(xlabels)))

    sim = [exp.get(x).sim.mean() for x in xlabels]
    sim_err = [exp.get(x).sim.std() for x in xlabels]
    res = [exp.get(x).result.mean() if exp.get(x).result is not None else 0 for x in xlabels]
    res_err = [exp.get(x).result.std() if exp.get(x).result is not None else 0 for x in xlabe

    fig, ax = plt.subplots(1, 1, figsize=(10, 7))

    width = 0.45
    ax.bar(x - width / 2, sim, yerr=sim_err,
           width=width, color="#005794", label="Simulated device")
    ax.bar(x + width / 2, res, yerr=res_err,
           width=width, color="#f06e00", label="Actual device")

    ax.set_xticks(x, xlabels)
    ax.set_ylim((0.6, 1))
    ax.legend()
    ax.yaxis.grid(True)
    ax.set_xlabel("Type of teleportation circuit")
    ax.set_ylabel("Success probability for state teleportation")

    plt.show()

plot(exp)
```



Programing quantum computers workflow

1. Identify problem to be solved
2. Choose suitable platform and vendor
3. Prepare quantum program
4. Transpile
5. Submit

Points 4.-5. are sometimes called quantum compilation.

Transpilation:

- **routing:** fitting the circuit to chosen device
- **decomposition:** replacing gates with the native set
- **optimization:** reducing the complexity of fitted circuit

We could see, that routing and optimization play a crucial role in current devices and probably will play for a long time. Just like GPU computing, we can do computations with pre-configured approaches (which will get better over time), but if we will want to get the most out of the QPU, we will have to consider QPU's architecture, quality and (potentially) speed.

In [43]:
```
# If you did your own coalculations, replace the JOB_IDS by the following
# dictionary to have the results accessible
JOB_IDS
```

Out[43]:
```
{'standard': '62b973b1013339a3d8dde252',
 'no_cif': '62b973da40f1544890afc0e3',
 'no_int_measure': '62b973f7bd18a2fa9b327688',
 'bad_mapping': '62b9741c013339604adde253',
 'best': '62b9753c40f154d1ddafc0ea'}
```

In [ ]:
```
```