An Introduction to Qisit and Quantum Software Development

Author : Gwonhak Lee (gwonhak@gmail.com)

---

1. Qiskit 소개

Qiskit 은 python 기반 오픈소스 양자 SDK(Software Development Kit)로써, 양자회로 구현을 위한 요소, 회로 최적화 툴과 시뮬레이터와 양자 프로세서 인터페이스 등을 포함하고 있습니다. Qiskit은 크게 다가지 구성요소로 이루어져 있습니다.

(클릭하시면 각 요소에 해당하는 github repo를 확인하실 수 있습니다.)

Terra	회로구성, 최적화 등 Qiskit의 기본 구성요소
Aer	고전 컴퓨터에서 양자회로 시뮬레이션을 위한 요소
Ignis	양자 하드웨어 검증과 Error Correction을 위한 툴을 제공
Aqua	양자 알고리즘 응용 패키지 (화학, 금융, 기계학습, 최적화)
IBMQ Provider	IBMQ 양자 프로세서 활용을 위한 인터페이스

Qiskit Aqua는 0.9.0 버전 이후로 domain 별로 분화되었습니다.

- qiskit-finance,
- qiskit-machine-learning,
- qiskit-nature,
- qiskit-optimization

2. Table of Contents

본 튜토리얼은 다음과 같은 순서로 준비되었습니다.

3. Resources

- qiskit tutorial (https://qiskit.org/documentation/tutorials.html)
- qiskit slack channel (for QnA) (https://qiskit.slack.com)
- qiskit github (https://github.com/Qiskit)

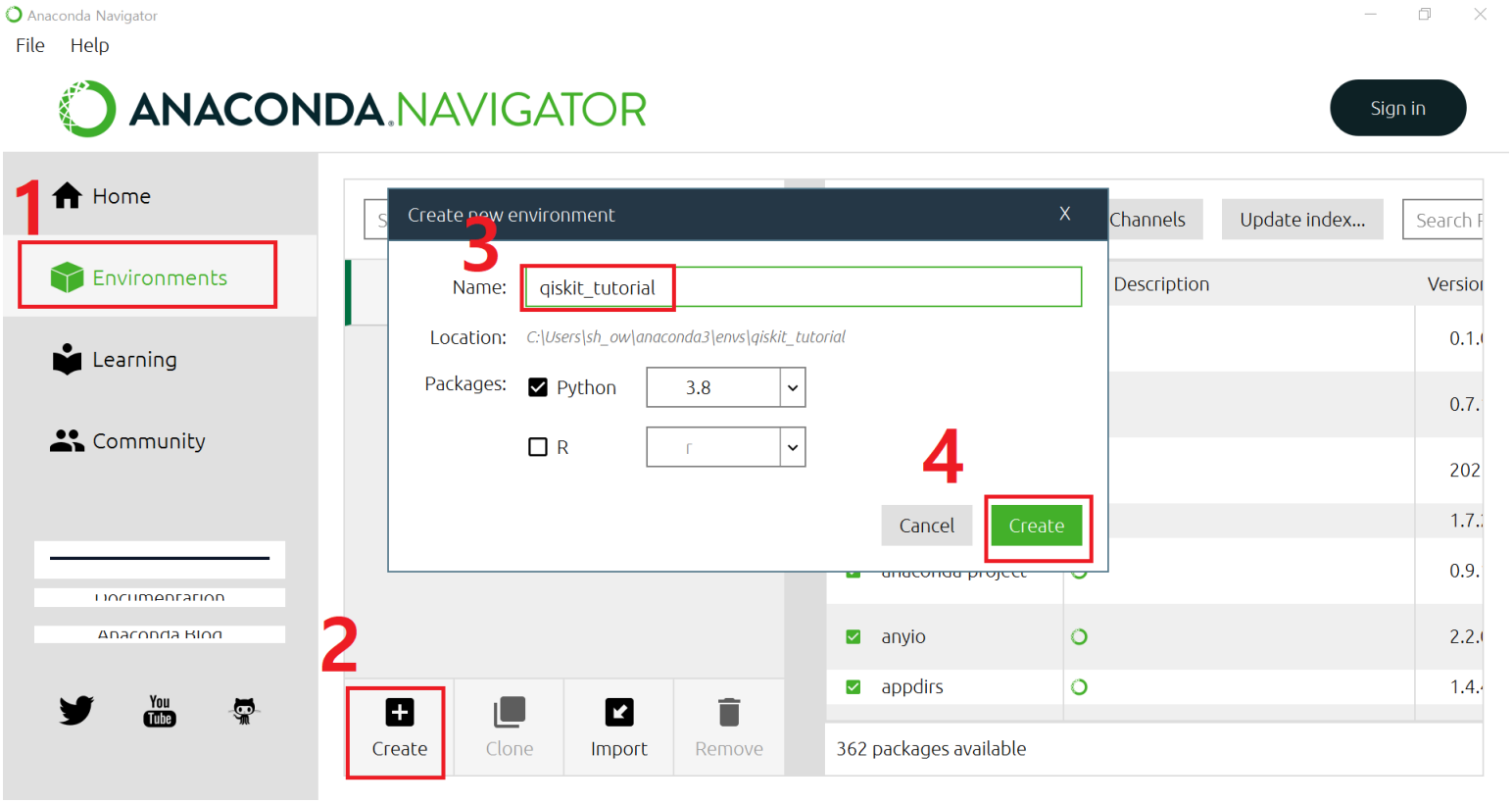
1. Installation

1-1. Install Anaconda

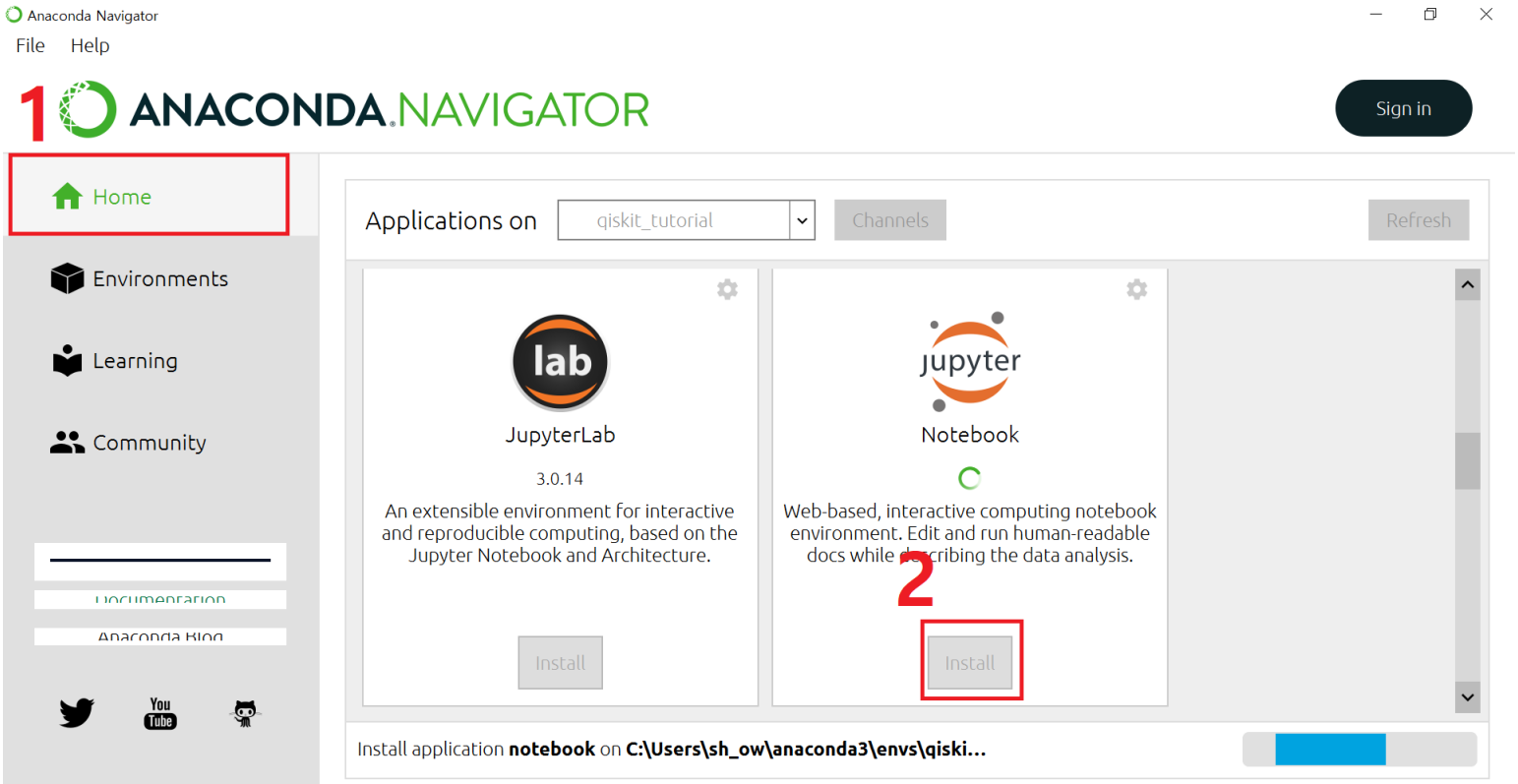
(https://www.anaconda.com/distributuion)

1-2. Setup Virtual Environment

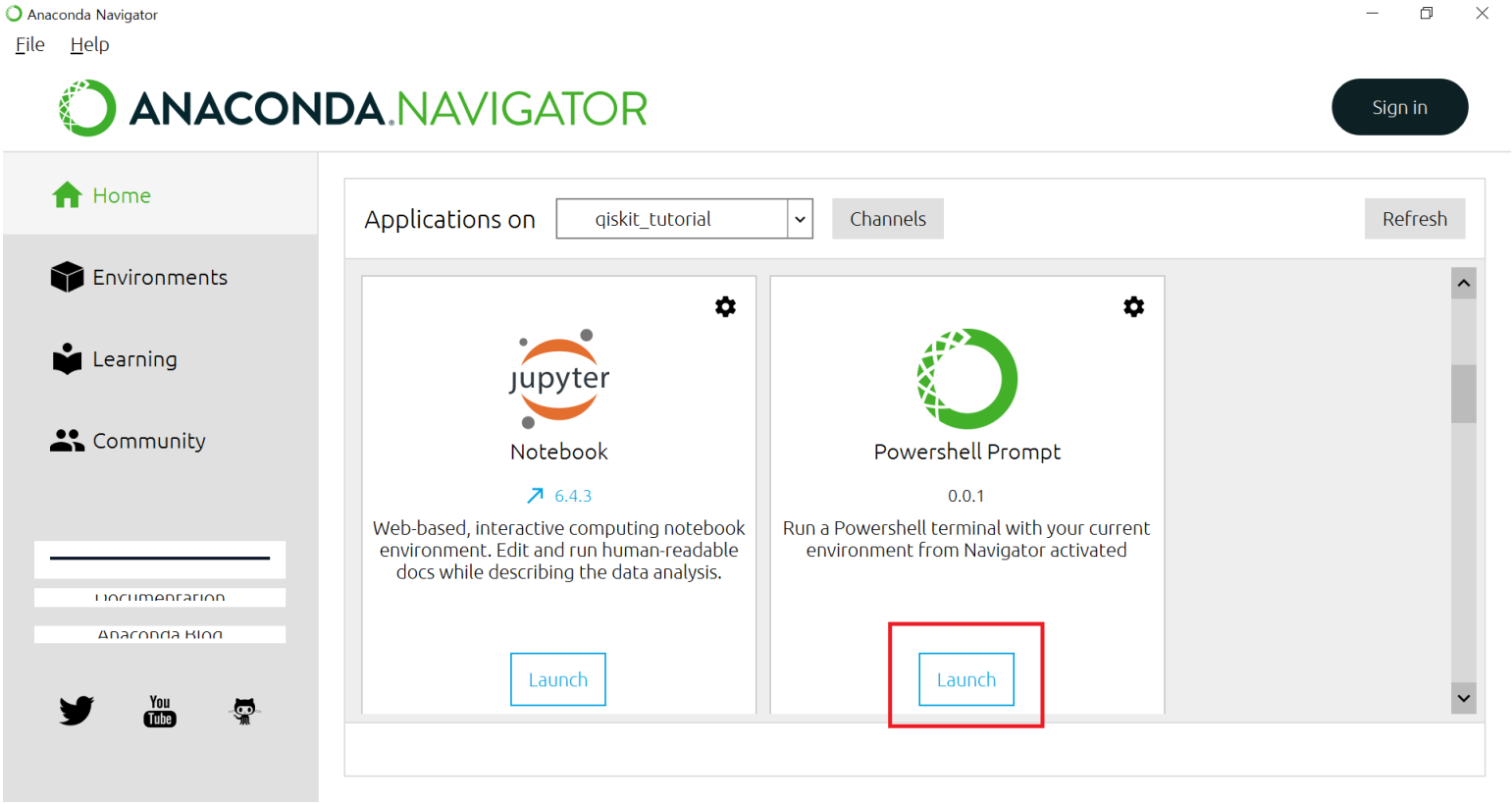
Open anaconda navigator



1-3. Install Jupyter Notebook



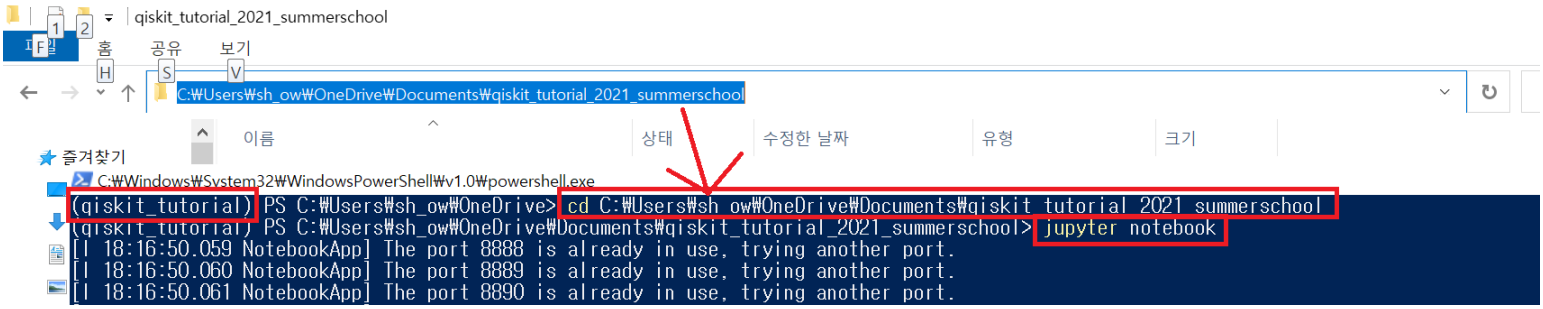
1-4. Launch Powershell



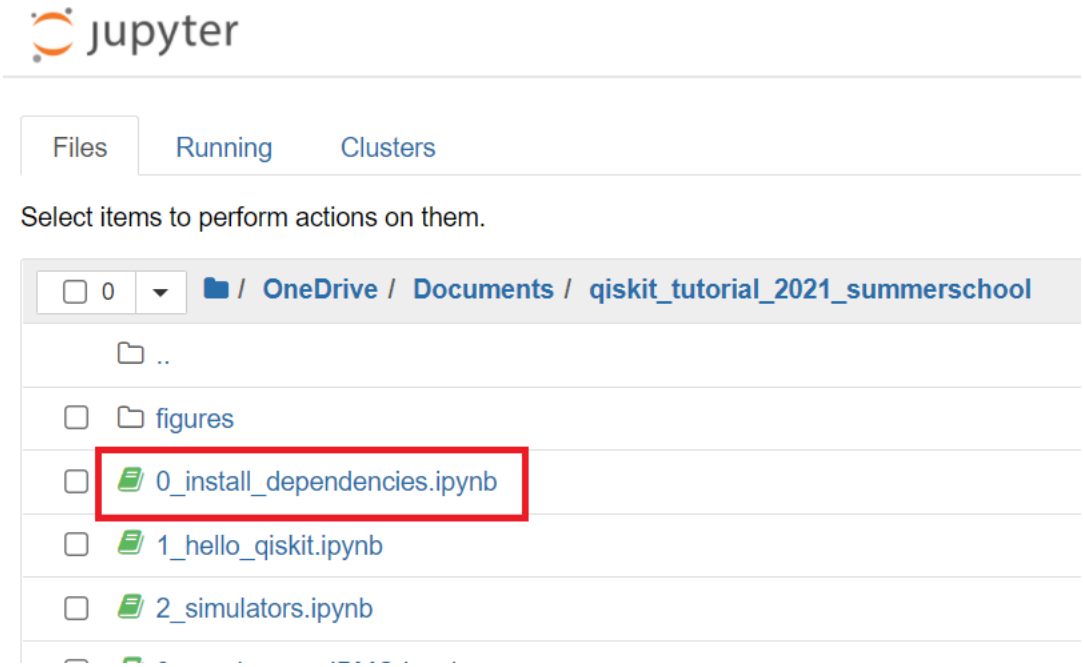
1-5. Launch jupyter notebook

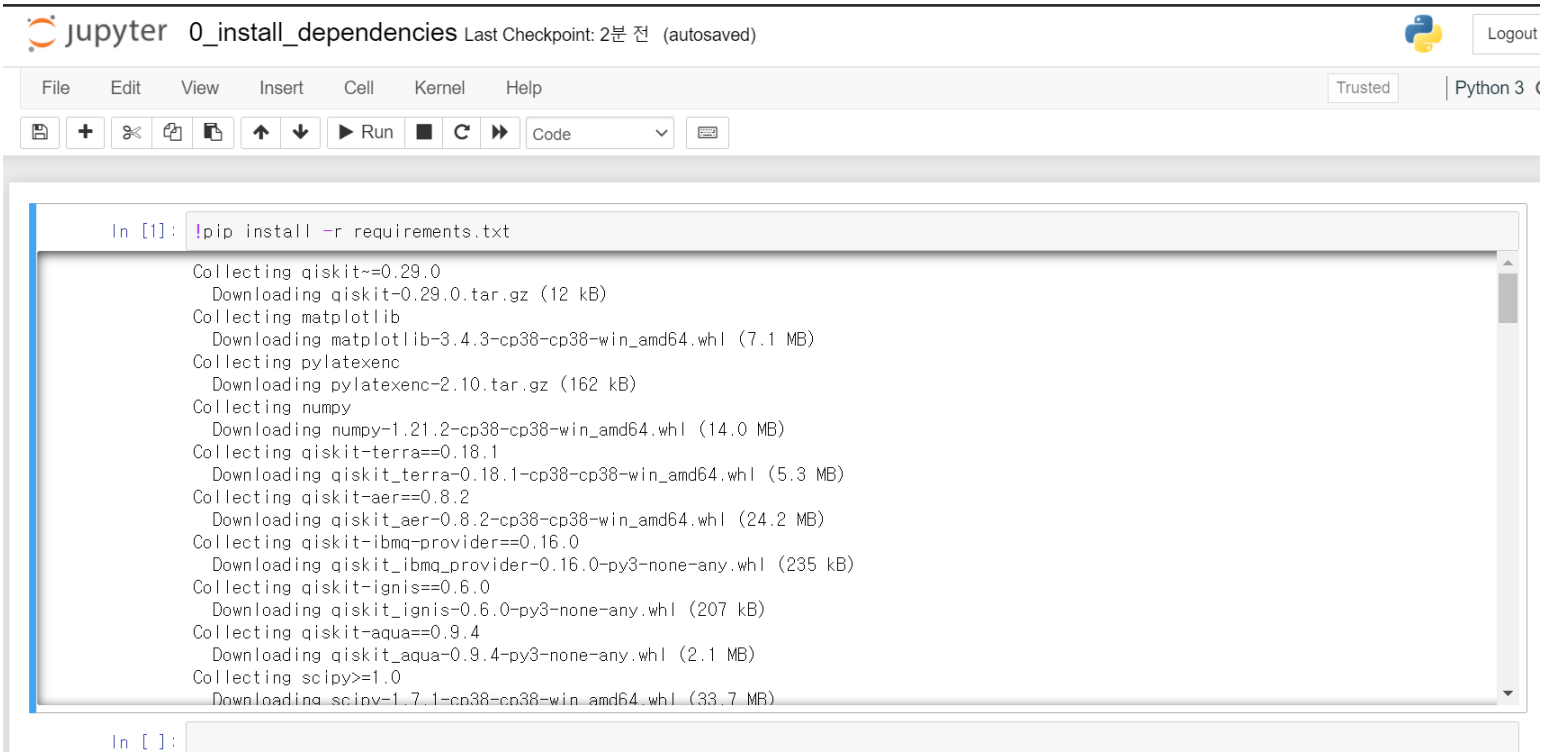
1. 튜토리얼 파일이 있는 곳으로 이동 (윈도우 탐색기)
2. 주소 복사
3. Powershell에 아래 명령어 차례대로 입력

```
cd <Copied Addr>
jupyter notebook
```



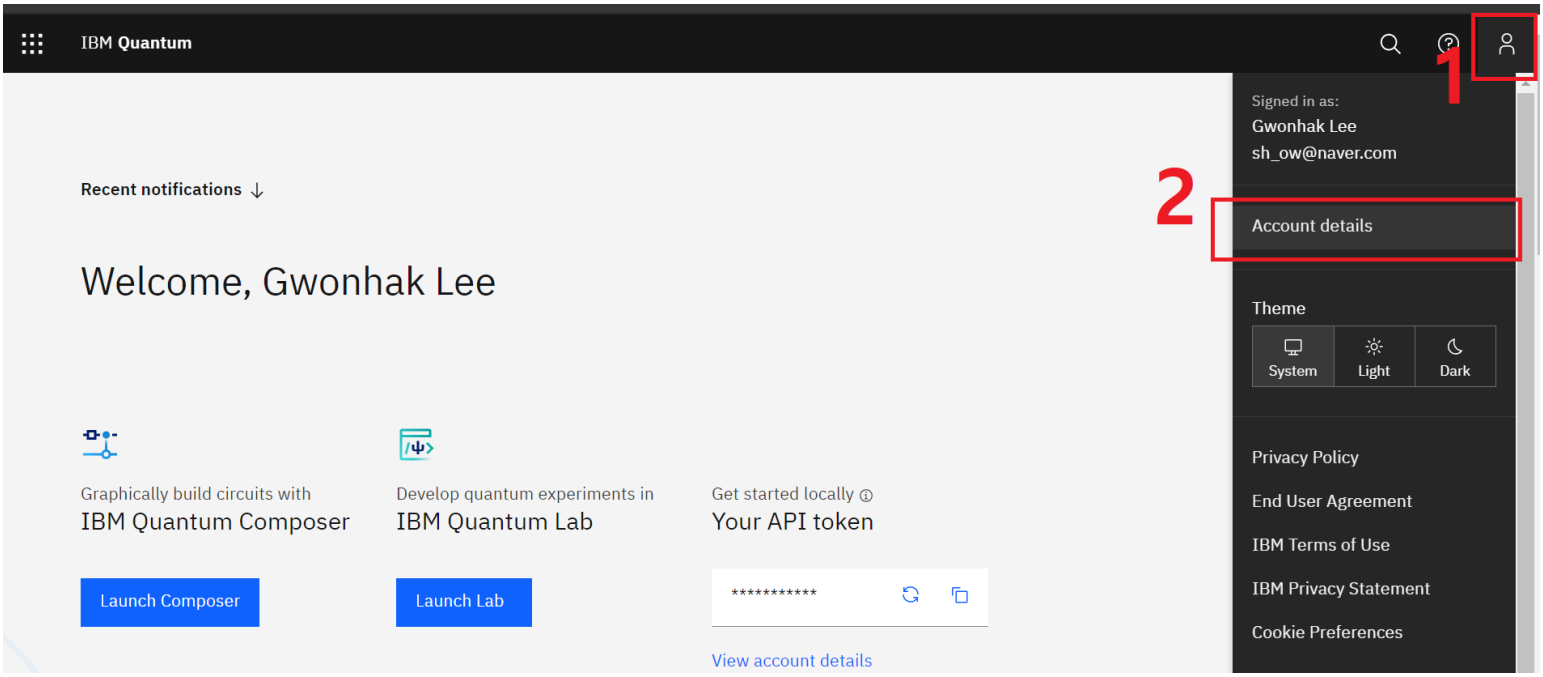
1-6. Install Dependencies





2. IBMQ Account

(https://quantum-computing.ibm.com/)



Gwonhak Lee

Account details

sh_ow@naver.com

SKKU

Edit

Delete account

Privacy & security

IBM Quantum End User Agreement

Notification Settings

	Email	In Tool
Product updates and announcements	Off	Off
IBM Quantum newsletter	Off	
Tips about using our tools	Off	
Requests for feedback to help improve our tools	Off	
Account and privacy notifications IBM is required to send emails related to servicing your account	On	

Use IBM Quantum services with Qiskit
on a local environment

1. Install Qiskit
2. Follow the instructions to access the IBM Quantum services from Qiskit

API Token

****

Regenerate

Copy token

1. Hello Qiskit (A Quick Introduction)

Author : Gwonhak Lee (gwonhak@gmail.com)

Qiskit 활용의 간단한 예시로써, Bell 상태 측정에 대한 회로를 구현하고, CPU를 이용한 양자회로 시뮬레이션을 수행합니다.

$$|\text{Bell}_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

기본적인 qiskit 개발의 단계는 다음과 같이 생각할 수 있습니다.

1. 양자회로 구성
    A. 객체의 선언 (qubits, classical bits, quantum circuit)
    B. 게이트를 적용하여 회로 구성
2. 양자회로 실행
    A. 양자실험 (시뮬레이션 혹은 실제 양자프로세서)
    B. 결과 획득 및 시각화

각 단계에 대한 예시를 구체적으로 살펴보도록 하겠습니다.

0. 필요한 요소 불러오기

먼저 필요한 Package를 불러옵니다.

In [7]:
```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.providers.aer import AerProvider
from qiskit import execute
```

1-1. 양자회로 구현을 위한 객체 선언

다음으로, 회로 구현에 필요한 양자 및 고전 bit 레지스터(저장공간)와 양자회로를 선언합니다.

Bell state를 구현하기 위해 2개의 qubit과, 이를 측정하는 2개의 Classical bit, 그리고 이들을 포함하는 양자회로를 선언합니다.

In [8]:
```
qr = QuantumRegister(2) # 2 qubits
cr = ClassicalRegister(2) # 2 classical bits
qc = QuantumCircuit(qr, cr) # a quantum circuit with 2 qubits and 2 bits
```

다음으로, 선언한 양자회로를 출력하여 확인합니다.

(아직 어떠한 게이트도 놓지 않았기 때문에 빈 회로가 출력됩니다.)

In [9]:
```
qc.draw()
```

Out[9]:
```
q12_0:

q12_1:
```

```
c1: 2/
```

1-2. Bell 회로 구성

다음 식과 같은 순서로 양자연산을 수행하여 Bell상태를 준비하는 회로를 구현합니다.

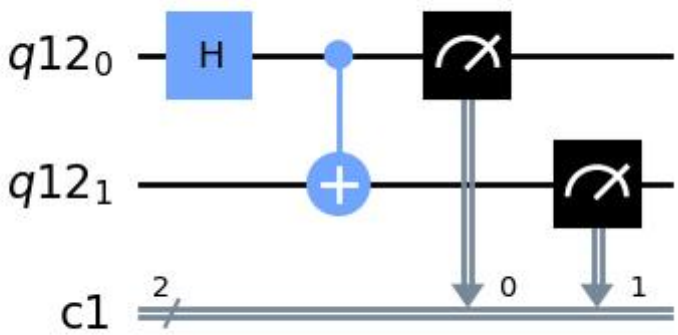
$$|00\rangle \xrightarrow{H_0} \frac{1}{\sqrt{2}}(|00\rangle+|10\rangle) \xrightarrow{CX_{0\to1}} \frac{1}{\sqrt{2}}(|00\rangle+|11\rangle)$$

0번째 qubit에 hadamard 연산과 0번째, 1번째 qubit에 cnot 연산을 취한 뒤, 측정하여 `cr` 에 저장합니다.

- 초기에 모든 qubit은 $|0\rangle$ 상태로 준비되어있습니다.
- 회로를 출력할 때, `'mpl'` 옵션을 넣어 텍스트가 아닌 그림 형태로 출력할 수 있습니다.
- 활용할 수 있는 gate의 종류는 qiskit document 에서 확인할 수 있습니다.
  https://qiskit.org/documentation/tutorials/circuits/3_summary_of_quantum_operations.html

In [10]:
```
qc.h(qr[0]) # 0번째 qubit에 hadamard 연산을 취합니다.
qc.cx(qr[0], qr[1]) # 0번째 qubit을 control, 1번째 qubit를 target으로 하는 CNOT 연산을
qc.measure(qr, cr) # qr을 측정하여 cr에 저장하는 측정 연산을 취합니다.

qc.draw('mpl') # 완성된 회로를 출력합니다.
```

Out[10]:



2-1. QASM 시뮬레이션

backend는 양자회로를 실행하기 위한 시뮬레이터 또는 실제 양자 프로세서입니다. 앞서 구성한 회로를 시뮬레이션 하기위해 먼저 qasm simulator backend를 불러옵니다.

- `AerProvider()` 는 Local PC를 이용하는 시뮬레이션 backend를 제공합니다. (2번째 튜토리얼)
- `IBMQProvider()` 는 IBM의 양자 프로세서 또는 고성능 클라우드를 이용한 시뮬레이션 backend를 제공합니다. (3번째 튜토리얼)
- `FakeProvider()` 는 IBM의 양자 프로세서의 특성(qubit connectivity, noise model 등)을 반영한 Local 시뮬레이션 backend를 제공합니다.

In [11]:
```
```

```
qasm_simulator = AerProvider().get_backend('qasm_simulator')
```

다음으로, 실험을 진행하고 결과값을 얻습니다.

QASM 시뮬레이터에서는 실제 양자프로세서와 같이 `shots = 2048` 에 해당하는 횟수의 동일한 양자실험을 반복하여 결과의 횟수 를 확인할 수 있습니다.

`get_counts()` 를 통해 얻은 실험결과는 python의 기본 자료구조인 dictionary(`Dict[str, int]`)의 형태로, Pauli-Z(Computational) basis에서 측정된 bitstring을 key로 하고 outcome의 횟수를 value로 합니다.

앞선 Bell State를 나타내는 식에서, $|00\rangle$상태와 $|11\rangle$ 이 동일한 확률로 관측될 것으로 예상되고 이를 확인할 수 있습니다.

- 양자회로를 실행하기 위한 방법으로는 `execute()` , `backend.run()` 그리고 `QuantumInstance` 를 이용하는 방법 등이 있습니다.

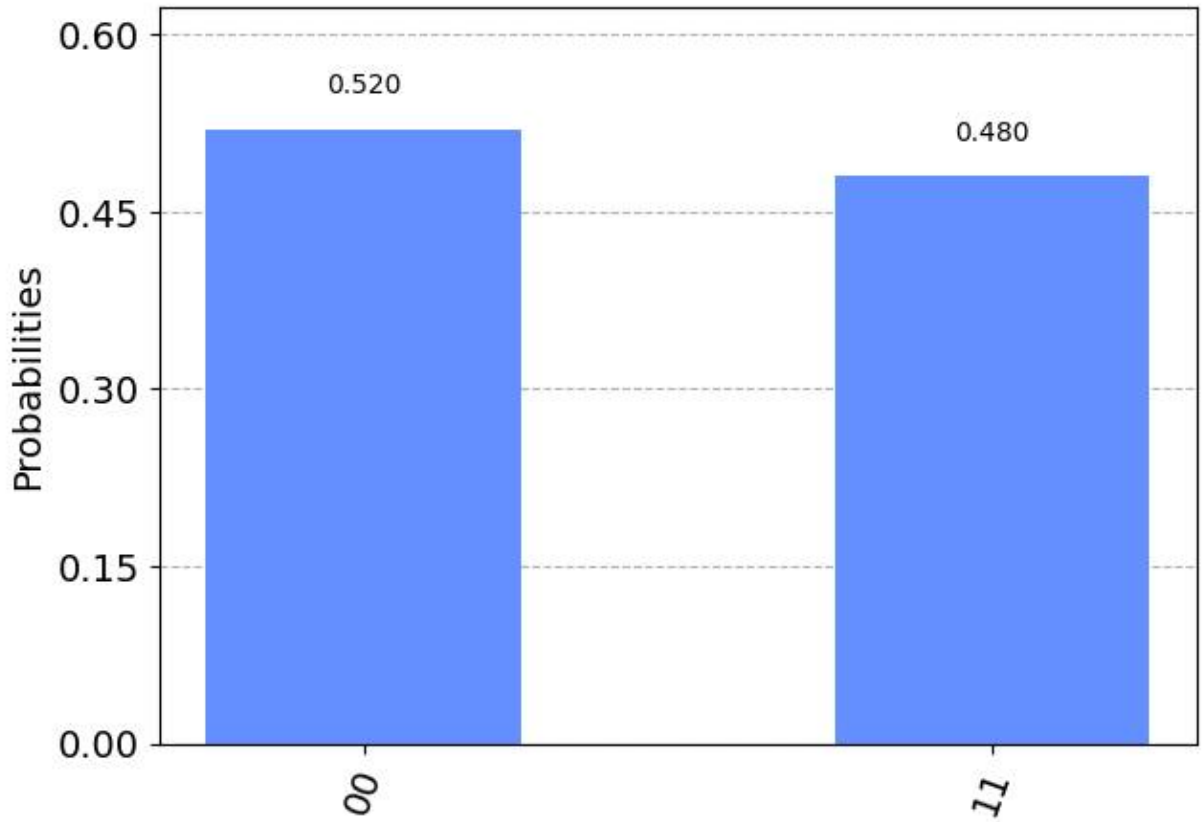
In [12]:
```
job_qasm = execute(qc, backend=qasm_simulator, shots=2048)

# 다른 방법
# job_qasm = qasm_simulator.run(qc, shots=2048)
#
# 또는
#
# from qiskit.utils import QuantumInstance
# qi = QuantumInstance(qasm_simulator, shots=2048)
# result = qi.execute(qc)
# counts = result.get_counts()

counts = job_qasm.result().get_counts()

print("Bell measurement result")
for k, v in counts.items():
    print(k, v)
```

```
Bell measurement result
11 984
00 1064
```

얻은 결과를 다음과 같이 시각화 할 수도 있습니다.

In [13]:
```
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

Out[13]:


0.60
0.45
0.30
0.15
0.00
Probabilities
0.520
0.480
00
11

2. Simulators

Author : Gwonhak Lee (gwonhak@gmail.com)

Qiskit Aer에서는 양자회로의 CPU (또는 GPU) 기반 시뮬레이션 backend를 제공합니다. 대표적으로 Aer에서 제공하는 Simulation Backend는 다음과 같습니다.

이름	설명	결과
qasm_simulator	이상적이거나 노이즈가 있는 양자 프로세서를 emulation하여 **measurement count** 를 반환합니다.	Counts
statevector_simulator	이상적인 시뮬레이션을 수행하여 **최종 양자상태**를 반환합니다.	Final state (Vector)
unitary_simulator	이상적인 양자회로의 최종 **Unitary Matrix** 를 반환합니다.	Unitary Matrix

(GPU 기반 Aer는 Linux 운영체제에서만 가능하므로 본 튜토리얼에서는 포함하지 않습니다.)

0. 필요한 요소 불러오기

In [13]:
```
import numpy as np

from qiskit.providers.aer import AerProvider
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit.circuit import Parameter
from qiskit.tools.visualization import plot_histogram
import qiskit.providers.aer.noise as noise
from qiskit.quantum_info import hellinger_fidelity
```

1-1. Ideal QASM Simulation

qasm simulator는 양자 프로세서를 emulation하여 measurement count를 반환합니다. 주어진 실험 횟수(shots)에 대해 결과의 빈도수를 확인할 수 있습니다.

이번 실험에서는 3개의 qubit에 대해 다음과 같은 연산을 수행하는 회로를 구현하였습니다.

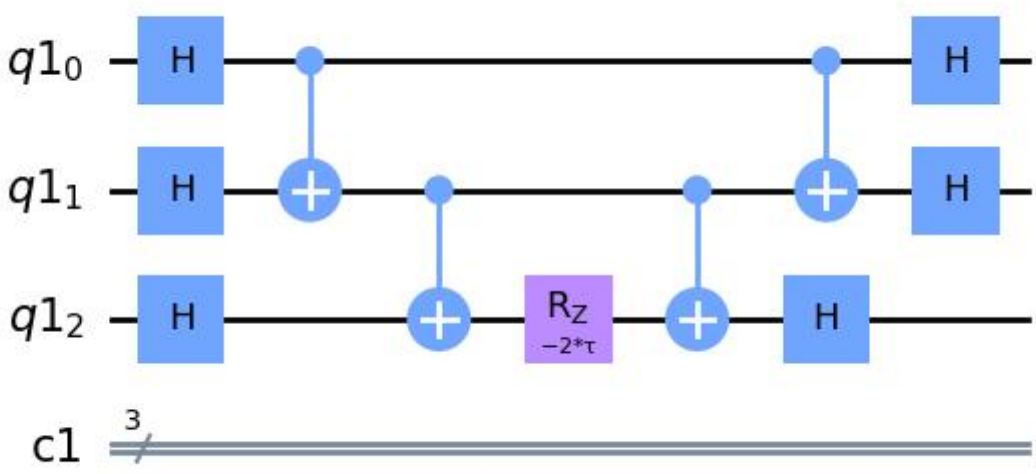
$$U(\tau) = \exp(-i\tau X_0X_1X_2)$$
$$= \cos(\tau)\, I - i\sin(\tau)\, X_0X_1X_2$$

- **Binding Parameters** : 다음과 같이 파라미터 Tau를 선언하고, 이를 회로에 적용시킨 뒤 시뮬레이션 단계에서 tau에 적당한 값을 지정할 수 있습니다.

In [14]:
```
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
qc = QuantumCircuit(qr, cr)

tau = Parameter('τ')

qc.h(qr)
qc.cx(qr[0], qr[1])
qc.cx(qr[1], qr[2])
```

```
qc.rz(-2 * tau, qr[2])

qc.cx(qr[1], qr[2])
qc.cx(qr[0], qr[1])
qc.h(qr)
qc.draw('mpl')
```

Out[14]:

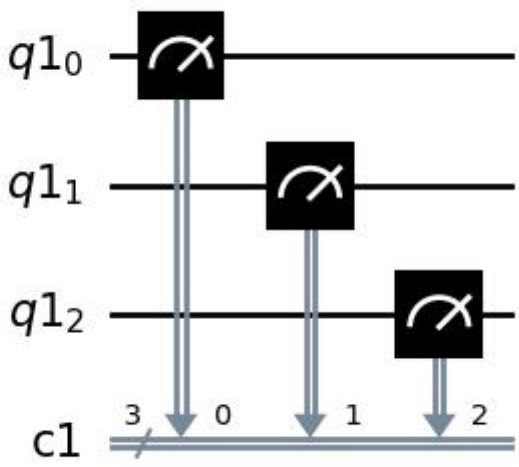


composing circuit : 다음과 같이 compose method를 통해 측정 회로 meas 를 따로 정의하고, 앞서 정의한 회로 qc 와 병합한 회로 qasm_qc 를 생성 할 수 있습니다.

In [15]:

```
meas = QuantumCircuit(qr, cr)
meas.measure(qr, cr)
meas.draw('mpl')
```

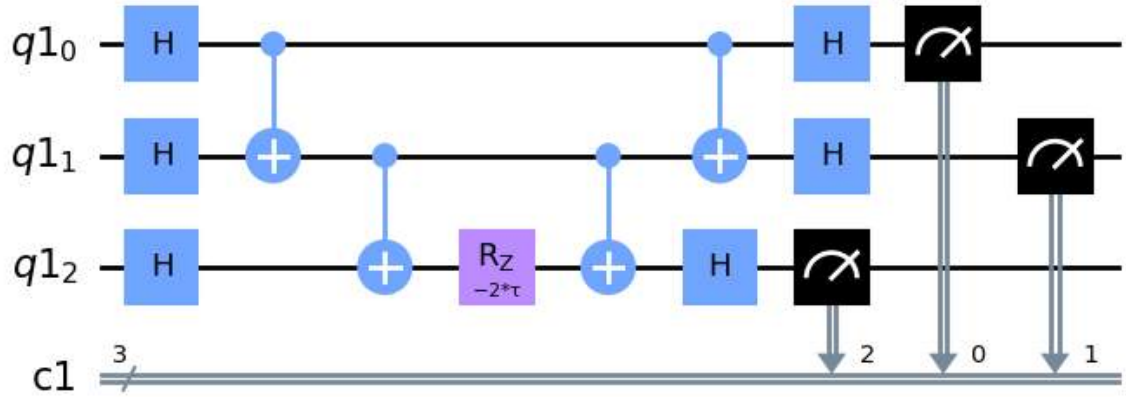
Out[15]:



In [16]:

```
qasm_qc = qc.compose(meas)
qasm_qc.draw('mpl')
```

Out[16]:

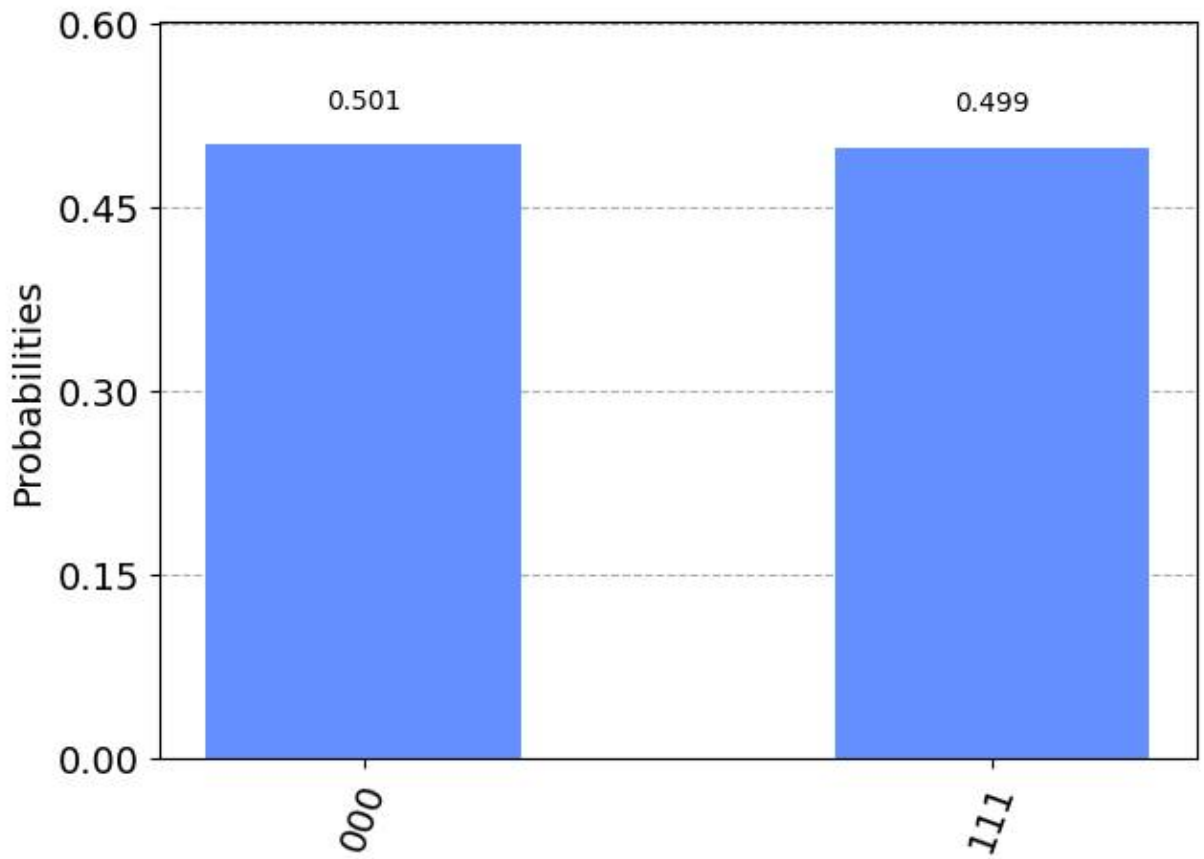


이제 tau에 `pi/4` 를 대입한 뒤 qasm simulation을 수행합니다.

In [17]:
```
bind_qasm_qc = qasm_qc.bind_parameters({tau: np.pi/4})

qasm_backend = AerProvider().get_backend('qasm_simulator')
job_qasm = qasm_backend.run(bind_qasm_qc, shots=4096)
counts_qasm = job_qasm.result().get_counts()

print(counts_qasm)
plot_histogram(counts_qasm)
```

```
{'000': 2054, '111': 2042}
```

Out[17]:



1-2. Noisy QASM Simulation

다음으로, QASM Simulation에 depolarization error를 추가하는 간단한 예시를 살펴보겠습니다.

Depolarization Error

$$E(\rho) = (1-\lambda)\rho + \lambda \mathrm{Tr}[\rho]\frac{I}{2^n}$$

- If λ = 0 this is the identity channel. E(ρ) = ρ
- If λ = 1 this is a completely depolarizing channel. E(ρ) = I / 2 ^{n}
- If λ = 4^n / (4^n - 1) this is a uniform Pauli error channel:

$$E(\rho) = \sum_j P_j \rho P_j / (4^n - 1) \text{ for all } P_j \neq I.$$

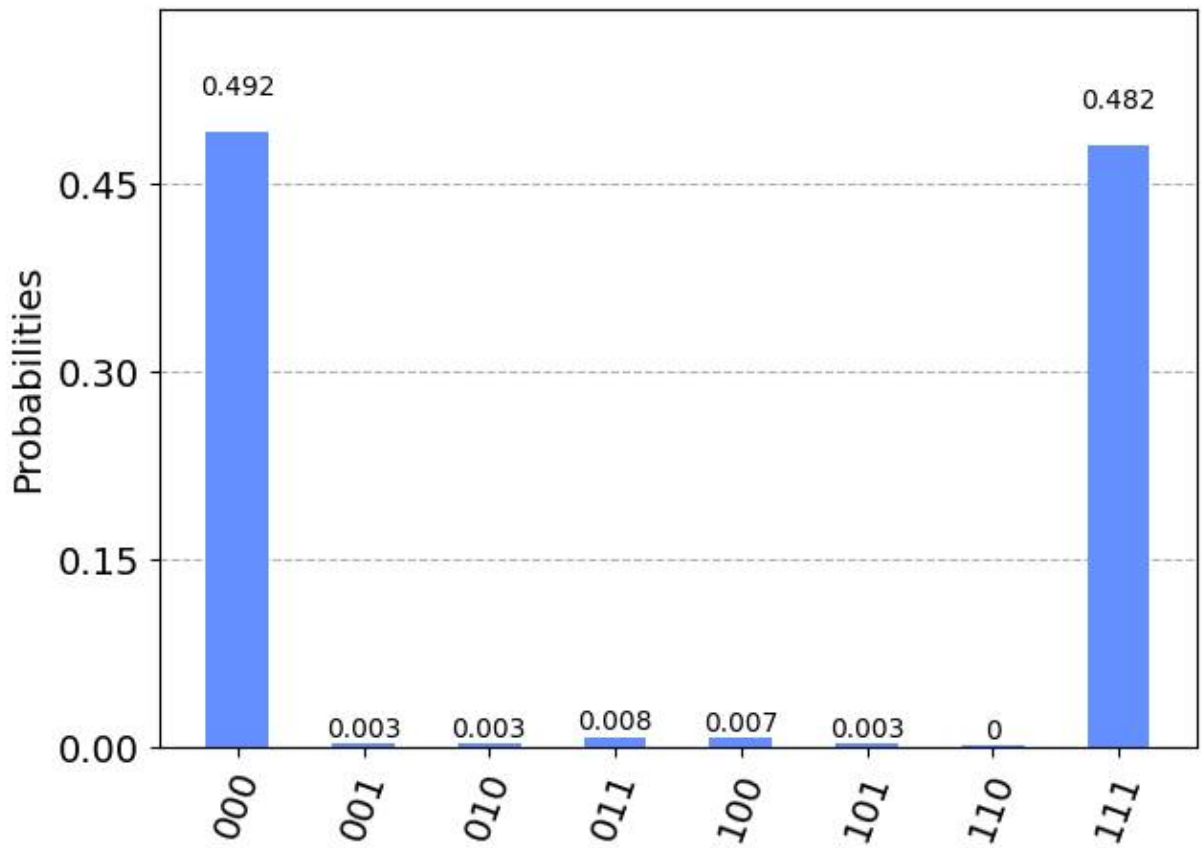
In [18]:
```
# Error probabilities
prob_1 = 0.001  # 1-qubit gate
prob_2 = 0.01   # 2-qubit gate

# Depolarizing quantum errors
error_1 = noise.depolarizing_error(prob_1, num_qubits=1)
error_2 = noise.depolarizing_error(prob_2, num_qubits=2)

# Add errors to noise model
noise_model = noise.NoiseModel()
noise_model.add_all_qubit_quantum_error(error_1, ['h', 'rz'])
noise_model.add_all_qubit_quantum_error(error_2, ['cx'])
```

생성한 노이즈모델과 함께 qasm simulation을 수행합니다.

In [19]:
```
job_qasm_noisy = qasm_backend.run(bind_qasm_qc, shots=4096, noise_model=noise_model)
counts_qasm_noisy = job_qasm_noisy.result().get_counts()
print(counts_qasm_noisy)
plot_histogram(counts_qasm_noisy)
```

```
{'000': 2015, '111': 1974, '100': 29, '010': 14, '011': 33, '001': 14, '101': 13, '110': 4}
```

Out[19]:



Depolarization noise model을 적용하였을때 Ideal한 case에서 관측되지 않는 결과들과 함께 오차가 발생했음을 확인할 수 있습니다.

Ideal case와 비교하기 위해 fidelity를 계산합니다.

$$H(P,Q) = \left(\sum_i \sqrt{p_i q_i}\right)^2$$

In [20]:
```
print(hellinger_fidelity(counts_qasm, counts_qasm_noisy))
```

0.9738638048244478

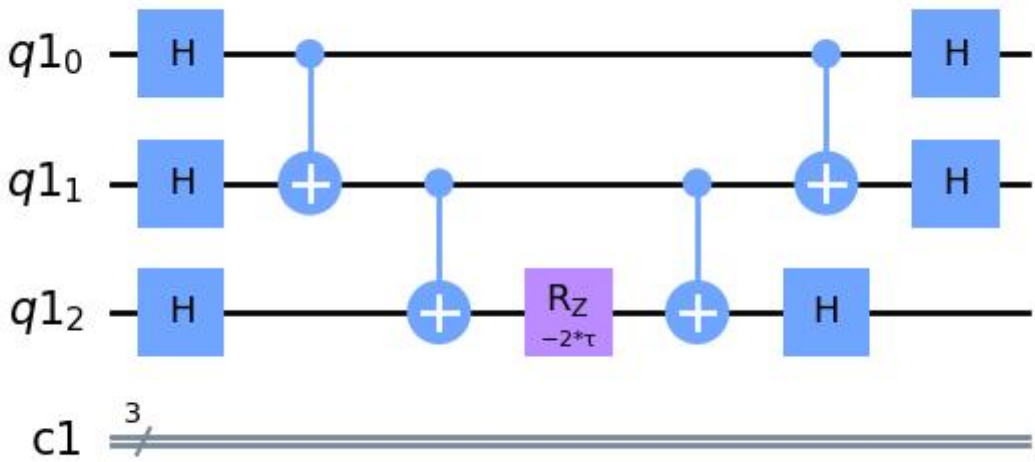
2. Statevector Simulation

다음으로, 최종 양자상태를 확인할 수 있는 statevector simulation을 수행합니다.

- statevector simulation을 수행하는 회로에는 측정 gate를 배제한 다음 회로를 사용합니다.

In [21]:
```
qc.draw('mpl')
```

Out[21]:



In [22]:
```
bind_qc = qc.bind_parameters({tau: np.pi/4})
sv_backend = AerProvider().get_backend('statevector_simulator')
job_sv = sv_backend.run(bind_qc)
final_sv = job_sv.result().get_statevector()

np.set_printoptions(suppress=True)
print(final_sv)
```

```
[ 0.70710678-0.j          -0.          +0.j          -0.          +0.j
 -0.          +0.j          -0.          +0.j          -0.          -0.j
 -0.          -0.j           0.          +0.70710678j]
```

양자상태가 numpy vector 형태로 주어진 것을 확인할 수 있습니다.

2. Statevector Simulation

다음으로, 회로의 unitary 행렬을 확인할 수 있는 unitary simulation을 수행합니다.

- unitary simulation을 수행하는 회로에는 측정 gate를 배제한 회로를 사용합니다.

In [23]:
```
unitary_backend = AerProvider().get_backend('unitary_simulator')
job_unitary = unitary_backend.run(bind_qc)
unitary = job_unitary.result().get_unitary()
print(unitary)
```

```
[[ 0.70710678-0.j          0.        +0.j          -0.        +0.j
  -0.        +0.j         -0.        +0.j          -0.        -0.j
  -0.        +0.j          0.        +0.70710678j]
 [-0.        +0.j          0.70710678-0.j          -0.        +0.j
  -0.        +0.j         -0.        -0.j          -0.        +0.j
   0.        +0.70710678j -0.        -0.j         ]
 [-0.        +0.j         -0.        +0.j           0.70710678-0.j
   0.        +0.j         -0.        +0.j           0.        +0.70710678j
  -0.        +0.j         -0.        -0.j         ]
 [-0.        +0.j         -0.        +0.j           0.        +0.j
   0.70710678-0.j          0.        +0.70710678j -0.        +0.j
  -0.        -0.j          0.        +0.j         ]
 [-0.        +0.j         -0.        -0.j          -0.        +0.j
   0.        +0.70710678j  0.70710678-0.j          0.        +0.j
  -0.        +0.j         -0.        +0.j         ]
 [-0.        -0.j         -0.        +0.j           0.        +0.70710678j
  -0.        +0.j          0.        +0.j           0.70710678-0.j
  -0.        +0.j         -0.        +0.j         ]
 [-0.        -0.j          0.        +0.70710678j -0.        +0.j
  -0.        -0.j         -0.        +0.j          -0.        +0.j
   0.70710678-0.j         -0.        +0.j         ]
 [ 0.        +0.70710678j -0.        +0.j          -0.        -0.j
  -0.        +0.j         -0.        +0.j          -0.        +0.j
   0.        +0.j          0.70710678-0.j         ]]
```

In [24]:
```
for idx, x in np.ndenumerate(unitary):
    if abs(x) > 1e-7:
        print(f"{idx}, {x.real if abs(x.real) > 1e-7 else 0.0} {' + j'+ str(x.imag)
```

```
(0, 0), 0.7071067811865477 0.0
(0, 7), 0.0  + j0.7071067811865479
(1, 1), 0.7071067811865477 0.0
(1, 6), 0.0  + j0.7071067811865479
(2, 2), 0.7071067811865477 0.0
(2, 5), 0.0  + j0.7071067811865479
(3, 3), 0.7071067811865477 0.0
(3, 4), 0.0  + j0.7071067811865479
(4, 3), 0.0  + j0.7071067811865479
(4, 4), 0.7071067811865477 0.0
(5, 2), 0.0  + j0.7071067811865479
(5, 5), 0.7071067811865477 0.0
(6, 1), 0.0  + j0.7071067811865479
(6, 6), 0.7071067811865477 0.0
(7, 0), 0.0  + j0.7071067811865479
(7, 7), 0.7071067811865479 0.0
```

3. Running on IBM Q

Author : Gwonhak Lee (gwonhak@gmail.com)

다음으로 IBM의 양자컴퓨터를 활용하는 방법에 대해 살펴보겠습니다.

0. 필요한 요소 불러오기

In [14]:
```
from qiskit import IBMQ, QuantumRegister, QuantumCircuit, ClassicalRegister, transpi
from qiskit.tools import backend_overview, backend_monitor
from qiskit.providers.ibmq.job import job_monitor
from qiskit.quantum_info import hellinger_fidelity
from qiskit.providers.ibmq import IBMQAccountCredentialsNotFound
from qiskit.providers.aer import AerProvider
from qiskit.tools.visualization import plot_gate_map, plot_histogram
```

1. QPU Backend 확인

IBMQ의 계정을 활성화합니다.

1. IBMQ 홈페이지(https://quantum-computing.ibm.com/)에 로그인합니다.
2. 좌측 상단의 계정 아이콘 -> Account Details 를 클릭합니다.
3. 좌측 중앙에 Copy Token 버튼을 클릭하여 아래 Block의 token 변수에 string 형태로 입력합니다.

In [15]:
```
token = 'f5876d2c6ea00ce4cefe8fcf6324956e042cd2a7cd57f6efc1e7ba38727964f10cef560830337
try:
    IBMQ.disable_account()
except IBMQAccountCredentialsNotFound:
    pass
IBMQ.enable_account(token)
```

Out[15]:
```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

활용할 수 있는 backend의 목록을 확인합니다.

In [16]:
```
backend_overview()
```

```
ibmq_manila
-----------
Num. Qubits:  5
Pending Jobs: 4
Least busy:   False
Operational:  True
Avg. T1:      163.6
Avg. T2:      58.3

ibmq_quito
----------
Num. Qubits:  5
Pending Jobs: 95
Least busy:   False
Operational:  True
Avg. T1:      89.8
Avg. T2:      109.8

ibmq_belem
----------
Num. Qubits:  5
Pending Jobs: 76
Least busy:   False
Operational:  True
Avg. T1:      88.7
Avg. T2:      108.8

ibmq_lima
---------
Num. Qubits:  5
Pending Jobs: 0
Least busy:   True

ibmq_bogota
-----------
Num. Qubits:  5
Pending Jobs: 1
Least busy:   False

ibmq_santiago
-------------
Num. Qubits:  5
Pending Jobs: 101
Least busy:   False
```

```
Operational:  True          Operational:  True          Operational:  True
Avg. T1:      95.6          Avg. T1:      88.2          Avg. T2:      94.8
Avg. T2:      104.4         Avg. T2:      113.4         Avg. T2:      113.0
```

```
ibmq_armonk
-----------
Num. Qubits:  1
Pending Jobs: 385
Least busy:   False
Operational:  True
Avg. T1:      244.9
Avg. T2:      255.7
```

특정한 qpu backend의 자세한 사항을 확인할 수 있습니다.

아래 예시에서는 `ibmq_manila` 의 특성을 확인합니다.

In [17]:
```
IBMQ_provider = IBMQ.get_provider()
ibmq_manila_backend = IBMQ_provider.get_backend('ibmq_manila')
backend_monitor(ibmq_manila_backend)
```

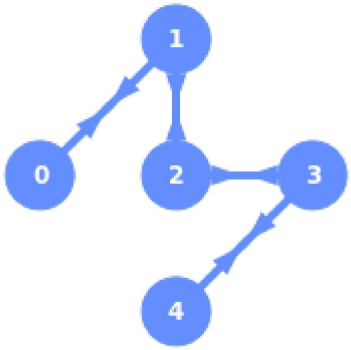
```
ibmq_manila
===========
Configuration
-------------
    n_qubits: 5
    operational: True
    status_msg: active
    pending_jobs: 4
    backend_version: 1.0.5
    basis_gates: ['id', 'rz', 'sx', 'x', 'cx', 'reset']
    local: False
    simulator: False
    quantum_volume: 32
    credits_required: True
    sample_name: family: Falcon, revision: 5.11, segment: L
    input_allowed: ['job']
    pulse_num_channels: 9
    memory: True
    meas_levels: [1, 2]
    qubit_lo_range: [[4.46277998307546, 5.46277998307546], [4.338419511638877, 5.33841
9511638877], [4.536938824727899, 5.536938824727899], [4.45130087049468, 5.451300870494
68], [4.566354311434293, 5.566354311434293]]
    u_channel_lo: [[{'q': 1, 'scale': (1+0j)}], [{'q': 0, 'scale': (1+0j)}], [{'q': 2,
'scale': (1+0j)}], [{'q': 1, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}], [{'q': 2,
'scale': (1+0j)}], [{'q': 4, 'scale': (1+0j)}], [{'q': 3, 'scale': (1+0j)}]]
    description: 5 qubit device
    conditional: False
    url: None
    default_rep_delay: 250.0
    processor_type: {'family': 'Falcon', 'revision': '5.11', 'segment': 'L'}
    max_shots: 8192
    dt: 0.2222222222222222
    acquisition_latency: []
    rep_delay_range: [0.0, 500.0]
    supported_instructions: ['u3', 'cx', 'x', 'acquire', 'setf', 'sx', 'shiftf', 'rese
t', 'u1', 'id', 'delay', 'rz', 'measure', 'u2', 'play']
    meas_lo_range: [[6.663214088, 7.663214088], [6.7833221550000005, 7.783322155000000
5], [6.7189281020000005, 7.7189281020000005], [6.6101423420000005, 7.610142342000000
```

5], [6.846997692, 7.846997692]]
coupling_map: [[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4, 3]]
hamiltonian: {'description': 'Qubits are modeled as Duffing oscillators. In this case, the system includes higher energy states, i.e. not just |0> and |1>. The Pauli operators are generalized via the following set of transformations:₩n₩n$(₩₩mathbb{I}-₩₩sigma_{i}^z)/2 ₩₩rightarrow 0_i ₩₩equiv b^₩₩dagger_{i} b_{i}$,₩n₩n$₩₩sigma_{+} ₩₩rightarrow b^₩₩dagger$,₩n₩n$₩₩sigma_{-} ₩₩rightarrow b$,₩n₩n$₩₩sigma_{i}^X ₩₩rightarrow b^₩₩dagger_{i} + b_{i}$.₩n₩nQubits are coupled through resonator buses. The provided Hamiltonian has been projected into the zero excitation subspace of the resonator buses leading to an effective qubit-qubit flip-flop interaction. The qubit resonance frequencies in the Hamiltonian are the cavity dressed frequencies and not exactly what is returned by the backend defaults, which also includes the dressing due to the qubit-qubit interactions.₩n₩nQuantities are returned in angular frequencies, with units 2*pi*GHz.₩n₩nWARNING: Currently not all system Hamiltonian information is available to the public, missing values have been replaced with 0.₩n', 'h_latex': '₩₩begin{align} ₩₩mathcal{H}/₩₩hbar = & ₩₩sum_{i=0}^{4}₩₩left(₩₩frac{₩₩omega_{q,i}}{2}(₩₩mathbb{I}-₩₩sigma_i^{z})+₩₩frac{₩₩Delta_{i}}{2}(0_i^2-0_i)+₩₩Omega_{d,i}D_i(t)₩₩sigma_i^{X}₩₩right) ₩₩₩₩₩ & + J_{0,1}(₩₩sigma_{0}^{+}₩₩sigma_{1}^{-}+₩₩sigma_{0}^{-}₩₩sigma_{1}^{+}) + J_{1,2}(₩₩sigma_{1}^{+}₩₩sigma_{2}^{-}+₩₩sigma_{1}^{-}₩₩sigma_{2}^{+}) + J_{2,3}(₩₩sigma_{2}^{+}₩₩sigma_{3}^{-}+₩₩sigma_{2}^{-}₩₩sigma_{3}^{+}) + J_{3,4}(₩₩sigma_{3}^{+}₩₩sigma_{4}^{-}+₩₩sigma_{3}^{-}₩₩sigma_{4}^{+}) ₩₩₩₩₩ & + ₩₩Omega_{d,0}(U_{0}^{(0,1)}(t))₩₩sigma_{0}^{X} + ₩₩Omega_{d,1}(U_{1}^{(1,0)}(t)+U_{2}^{(1,2)}(t))₩₩sigma_{1}^{X} ₩₩₩₩₩ & + ₩₩Omega_{d,2}(U_{3}^{(2,1)}(t)+U_{4}^{(2,3)}(t))₩₩sigma_{2}^{X} + ₩₩Omega_{d,3}(U_{6}^{(3,4)}(t)+U_{5}^{(3,2)}(t))₩₩sigma_{3}^{X} ₩₩₩₩₩ & + ₩₩Omega_{d,4}(U_{7}^{(4,3)}(t))₩₩sigma_{4}^{X} ₩₩₩₩₩ ₩₩end{align}', 'h_str': ['_SUM[i,0,4,wq{i}/2*(I{i}-Z{i})]', '_SUM[i,0,4,delta{i}/2*0{i}*0{i}]', '_SUM[i,0,4,-delta{i}/2*0{i}]', '_SUM[i,0,4,omegad{i}*X{i}||D{i}]', 'jq0q1*Sp0*Sm1', 'jq0q1*Sm0*Sp1', 'jq1q2*Sp1*Sm2', 'jq1q2*Sm1*Sp2', 'jq2q3*Sp2*Sm3', 'jq2q3*Sm2*Sp3', 'jq3q4*Sp3*Sm4', 'jq3q4*Sm3*Sp4', 'omegad1*X0||U0', 'omegad0*X1||U1', 'omegad2*X1||U2', 'omegad1*X2||U3', 'omegad3*X2||U4', 'omegad4*X3||U6', 'omegad2*X3||U5', 'omegad3*X4||U7'], 'osc': {}, 'qub': {'0': 3, '1': 3, '2': 3, '3': 3, '4': 3}, 'vars': {'delta0': -2.1573187977651487, 'delta1': -2.1753119475601674, 'delta2': -2.159281266514359, 'delta3': -2.158603148482815, 'delta4': -2.1495256907311115, 'jq0q1': 0.011845444218797994, 'jq1q2': 0.01196783968906386, 'jq2q3': 0.012402113956012375, 'jq3q4': 0.01218691037040823, 'omegad0': 0.958459680297847, 'omegad1': 0.9867433019584951, 'omegad2': 0.9541014919519683, 'omegad3': 0.9720864126008828, 'omegad4': 0.9836397693823058, 'wq0': 31.18206627242469, 'wq1': 30.40068638550042, 'wq2': 31.648020016692 75, 'wq3': 31.10994088091767, 'wq4': 31.832842970569903}}
online_date: 2021-04-28 04:00:00+00:00
allow_object_storage: True
open_pulse: False
rep_times: [1000.0]
meas_map: [[0, 1, 2, 3, 4]]
backend_name: ibmq_manila
multi_meas_enabled: True
pulse_num_qubits: 3
dynamic_reprate_enabled: True
channels: {'acquire0': {'operates': {'qubits': [0]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire1': {'operates': {'qubits': [1]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire2': {'operates': {'qubits': [2]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire3': {'operates': {'qubits': [3]}, 'purpose': 'acquire', 'type': 'acquire'}, 'acquire4': {'operates': {'qubits': [4]}, 'purpose': 'acquire', 'type': 'acquire'}, 'd0': {'operates': {'qubits': [0]}, 'purpose': 'drive', 'type': 'drive'}, 'd1': {'operates': {'qubits': [1]}, 'purpose': 'drive', 'type': 'drive'}, 'd2': {'operates': {'qubits': [2]}, 'purpose': 'drive', 'type': 'drive'}, 'd3': {'operates': {'qubits': [3]}, 'purpose': 'drive', 'type': 'drive'}, 'd4': {'operates': {'qubits': [4]}, 'purpose': 'drive', 'type': 'drive'}, 'm0': {'operates': {'qubits': [0]}, 'purpose': 'measure', 'type': 'measure'}, 'm1': {'operates': {'qubits': [1]}, 'purpose': 'measure', 'type': 'measure'}, 'm2': {'operates': {'qubits': [2]}, 'purpose': 'measure', 'type': 'measure'}, 'm3': {'operates': {'qubits': [3]}, 'purpose': 'measure', 'type': 'measure'}, 'm4': {'operates': {'qubits': [4]}, 'purpose': 'measure', 'type': 'measure'}, 'u0': {'operates': {'qubits': [0, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u1': {'operates': {'qubits': [1, 0]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u2': {'operates': {'qubits': [1, 2]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u3': {'operates': {'qubits': [2, 1]}, 'purpose': 'cross-resonance', 'type': 'control'}, 'u4': {'operates': {'qubits': [2, 3]}, 'purpose': 'cross-resonance', 'type':

```
'control'}, 'u5': {'operates': {'qubits': [3, 2]}, 'purpose': 'cross-resonance', 'typ
e': 'control'}, 'u6': {'operates': {'qubits': [3, 4]}, 'purpose': 'cross-resonance',
'type': 'control'}, 'u7': {'operates': {'qubits': [4, 3]}, 'purpose': 'cross-resonanc
e', 'type': 'control'}}
    allow_q_object: True
    meas_kernels: ['hw_qmfk']
    dtm: 0.2222222222222222
    discriminators: ['hw_qmfk', 'quadratic_discriminator', 'linear_discriminator']
    qubit_channel_mapping: [['u0', 'd0', 'u1', 'm0'], ['u3', 'u0', 'u1', 'm1', 'd1',
'u2'], ['d2', 'm2', 'u2', 'u4', 'u5', 'u3'], ['m3', 'u7', 'u6', 'u4', 'u5', 'd3'], ['u
6', 'd4', 'u7', 'm4']]
    n_registers: 1
    measure_esp_enabled: False
    n_uchannels: 8
    conditional_latency: []
    parametric_pulses: ['gaussian', 'gaussian_square', 'drag', 'constant']
    uchannels_enabled: True
    max_experiments: 75

Qubits [Name / Freq / T1 / T2 / RZ err / SX err / X err / Readout err]
----------------------------------------------------------------------
    Q0 / 4.96278 GHz / 129.01354 us / 101.74870 us / 0.00000 / 0.00022 / 0.00022 / 0.0
2510
    Q1 / 4.83842 GHz / 212.80826 us / 48.67210 us / 0.00000 / 0.00025 / 0.00025 / 0.03
290
    Q2 / 5.03694 GHz / 164.49714 us / 24.07609 us / 0.00000 / 0.00022 / 0.00022 / 0.02
140
    Q3 / 4.95130 GHz / 157.12573 us / 73.03125 us / 0.00000 / 0.00017 / 0.00017 / 0.01
800
    Q4 / 5.06635 GHz / 154.44051 us / 44.02548 us / 0.00000 / 0.00070 / 0.00070 / 0.04
280

Multi-Qubit Gates [Name / Type / Gate Error]
--------------------------------------------
    cx4_3 / cx / 0.00679
    cx3_4 / cx / 0.00679
    cx2_3 / cx / 0.00619
    cx3_2 / cx / 0.00619
    cx1_2 / cx / 0.00918
    cx2_1 / cx / 0.00918
    cx0_1 / cx / 0.00739
    cx1_0 / cx / 0.00739
```

IBM의 Superconducting QPU는 CNOT 게이트를 적용할 수 있는 qubit pair (Connectivity)가 제한되어 있습니다. `ibmq_manila` 는 다음과 같은 linear connectivity를 가지고 있습니다.

In [18]:
```
plot_gate_map(ibmq_manila_backend, plot_directed=True)
```



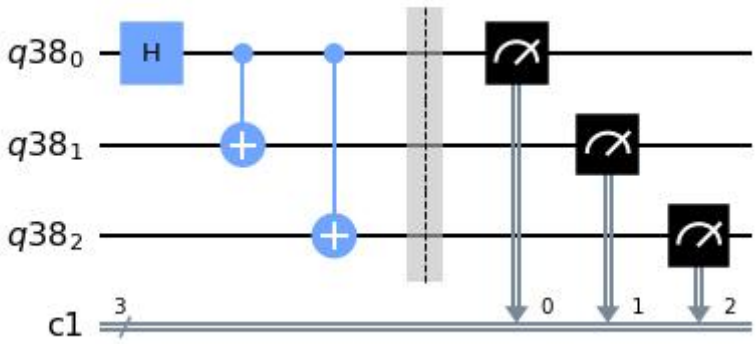
2. QPU Backend 사용하기 및 회로 최적화

3 qubit ghz상태를 준비하여 측정하는 회로를 구성하여 qpu backend를 사용해보겠습니다.

$$|GHZ_3\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

먼저 회로를 구현합니다.

In [19]:
```
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
ghz3 = QuantumCircuit(qr, cr)
ghz3.h(qr[0])
ghz3.cx(qr[0], qr[1])
ghz3.cx(qr[0], qr[2])
ghz3.barrier()
ghz3.measure(qr, cr)

ghz3.draw('mpl')
```



주어진 회로를 ibmq_manila_backend 를 통해 실행합니다.

In [20]:
```
job_exp = ibmq_manila_backend.run(ghz3, shots=2048)
job_monitor(job_exp) # An error must be raised here.
```

```
Job Status: ERROR - The Qobj uses gates (['h']) that are not among the basis gates
(['id', 'rz', 'sx', 'x', 'cx', 'reset']). Error code: 1106.
```

여기서, 에러 메세지가 출력될 것입니다. 그 이유는 manila qpu에서 Hadamard gate를 구현할 수

없기 때문입니다. qpu에서 물리적으로 구현 가능한 basis gate는 다음과 같이 확인할 수 있습니다.

In [21]:
```
print(ibmq_manila_backend.configuration().basis_gates)
```

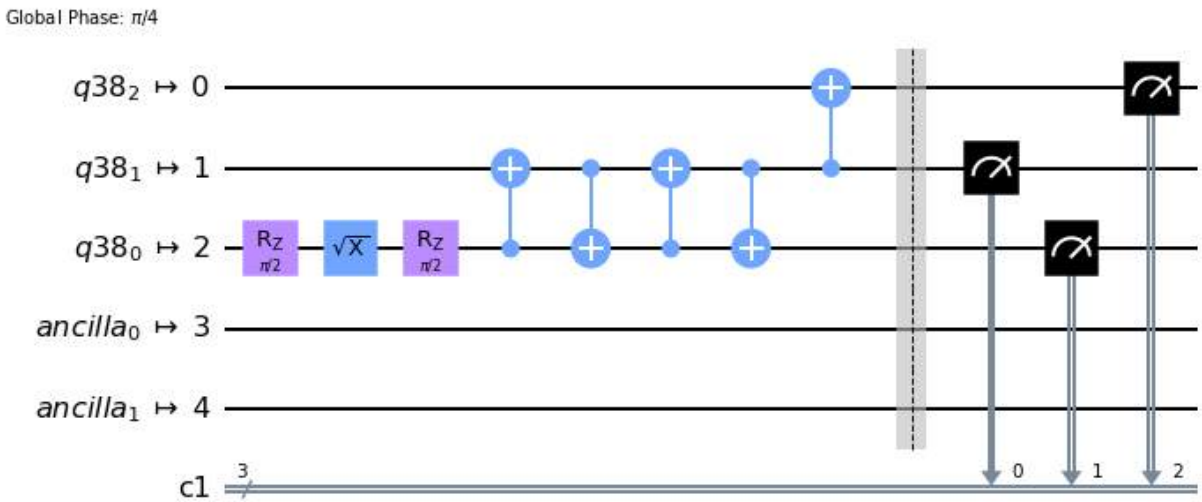
```
['id', 'rz', 'sx', 'x', 'cx', 'reset']
```

주어진 회로를 manila backend의 특성에 맞도록 변환하기 위해 transpile 과정을 거칩니다. hadamard gate가 manila backend에서 제공하는 basis gate들로 변환되었음을 확인할 수 있습니다. 또한 connectivity에 맞게 SWAP operation이 적용되었음을 확인할 수 있습니다.

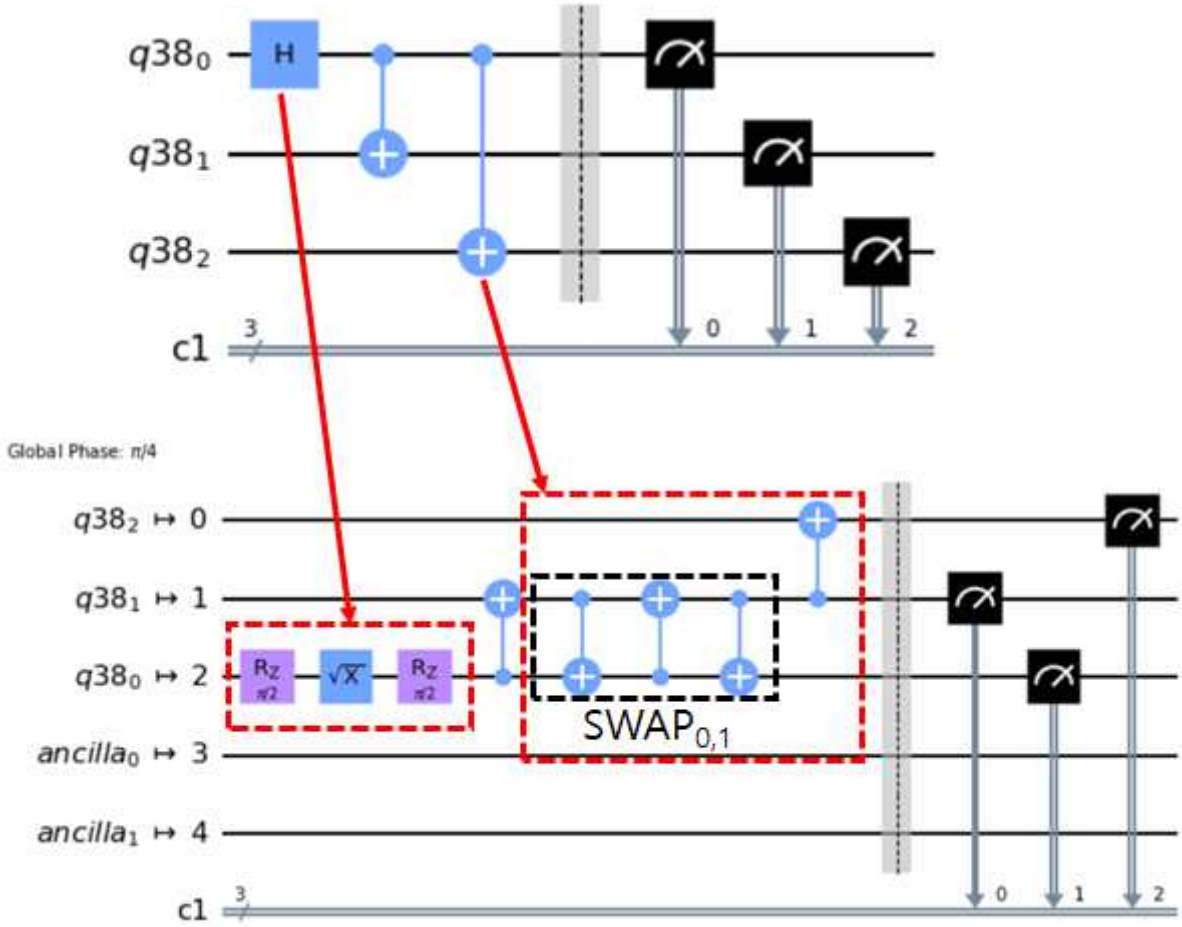
transpile 과정에서 기본적으로 제공되는 과정은 다음과 같습니다.

- Physical Qubit Layout (Connectivity)에 맞게 virtual to physical Qubit mapping
- 회로의 모든 gate들을 Basis gate set으로 구현
- 측정 gate 전에 barrier 삽입
- Qubit mapping에 맞도록 SWAP gate 생성
- SWAP gate를 3개의 CNOT gate로 분해
- CNOT-CNOT 상쇄
- Single qubit gate 최적화

In [22]:
```
transpiled_circuit = transpile(ghz3, backend=ibmq_manila_backend)
transpiled_circuit.draw('mpl')
```

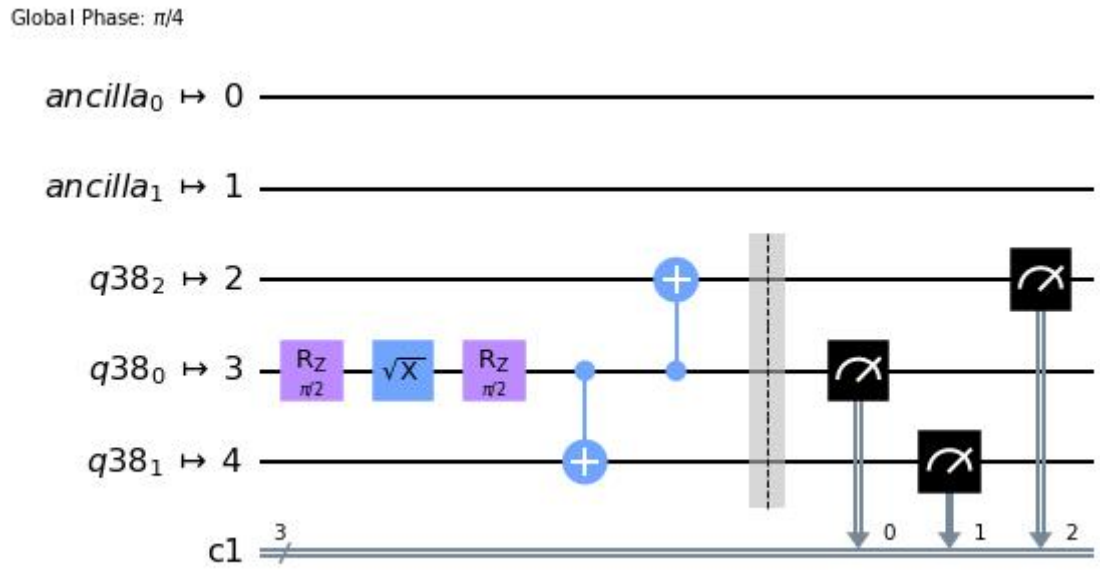


기본 transpile 과정을 거친 회로를 비교하면 다음과 같이 H gate가 basis gate로 decompose 되었음을 확인할 수 있고, qubit connectivity에 맞게 qubit에 SWAP 연산이 적용됨을 확인할 수 있습니다.



변환된 회로에서 여기서 0번째 qubit을 중앙으로 옮긴다면 불필요한 SWAP gate를 줄일 수 있음을 확인할 수 있습니다. 이는 `transpile` 에 `optimization_level` 옵션을 추가하여 자동으로 수행할 수 있습니다.

In [23]:
```
transpiled_circuit_opt = transpile(ghz3, backend=ibmq_manila_backend, optimization_le
transpiled_circuit_opt.draw('mpl')
```



`optimization_level` 은 0 ~ 3의 정수를 입력할 수 있으며, 각 단계마다 다음의 회로 최적화 기능이 추가됩니다.

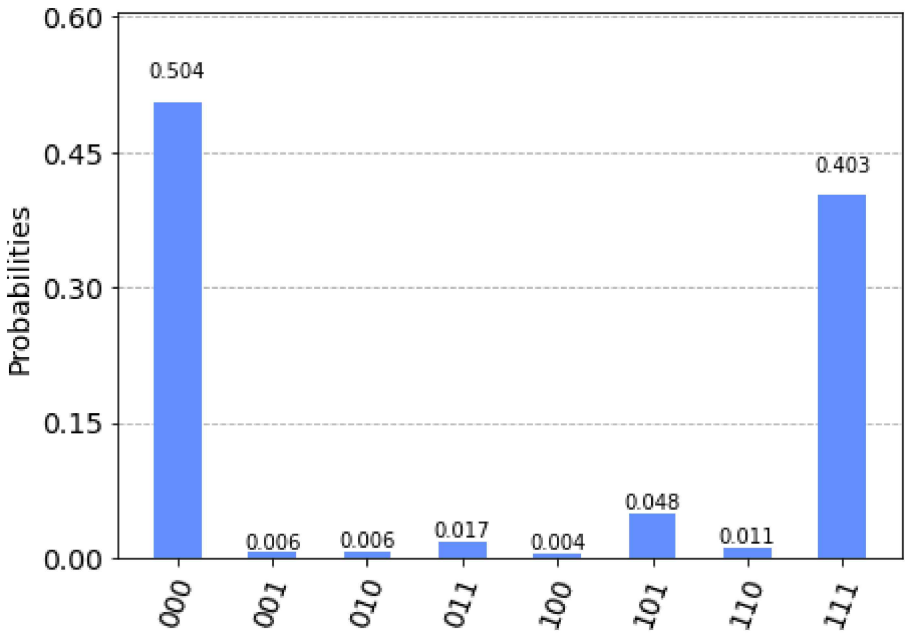
- 0단계 : 최적화 수행하지 않음
- 1단계(기본) : 이웃한 게이트만 소거
- 2단계 : Noise-adaptive qubit mapping, Gate cancellation using commutativity
- 3단계 : Heavier noise-adaptive qubit mapping, Gate cancellation using commutativity and unitary synthesis.

이제 주어진 회로를 QPU에서 실행하고 결과를 확인합니다.

In [24]:
```
job_exp = ibmq_manila_backend.run(transpiled_circuit_opt, shots=2048)
job_monitor(job_exp)
```

```
Job Status: job has successfully run
```

In [25]:
```
noisy_counts = job_exp.result().get_counts()
plot_histogram(noisy_counts)
```



Ideal simulation을 수행하고, fidelity를 확인합니다.

In [26]:
```
qasm_backend = AerProvider().get_backend('qasm_simulator')
job_ideal = qasm_backend.run(ghz3, shots=2048)
ideal_counts = job_ideal.result().get_counts()

print(hellinger_fidelity(noisy_counts, ideal_counts))
```

```
0.9023008480548862
```

3. Fake Backend 사용하기

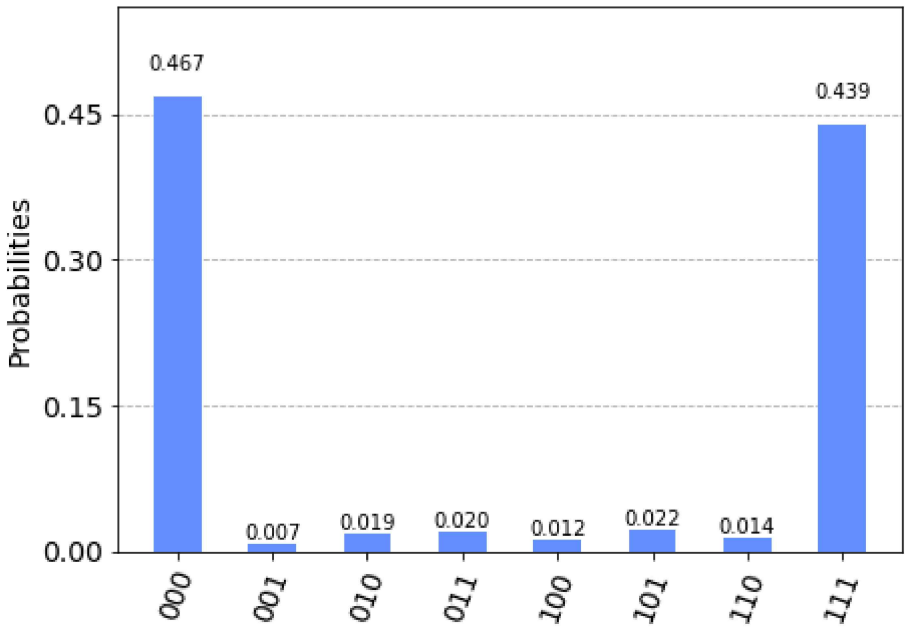
Fake backend를 활용하면 IBM QPU의 특성 및 노이즈 모델 등에 대한 정보를 불러와 시뮬레이션을 수행 할 수 있습니다. ibmq_rome QPU에 해당하는 fake backend를 불러옵니다.

In [30]:
```
from qiskit.test.mock import FakeRome

fake_rome = FakeRome()
```

ghz3 회로를 fake_rome에서 실행합니다. 이상적인 결과에 비해 노이즈가 있음을 확인할 수 있습

니다.

In [28]:
```
rome_transpiled = transpile(ghz3, backend=fake_rome, optimization_level=3)
job_fake_rome = fake_rome.run(rome_transpiled, shots=2048)
counts_fake_rome = job_fake_rome.result().get_counts()
plot_histogram(counts_fake_rome)
```



4. Readout Error Mitigation

Author : Gwonhak Lee (gwonhak@gmail.com)

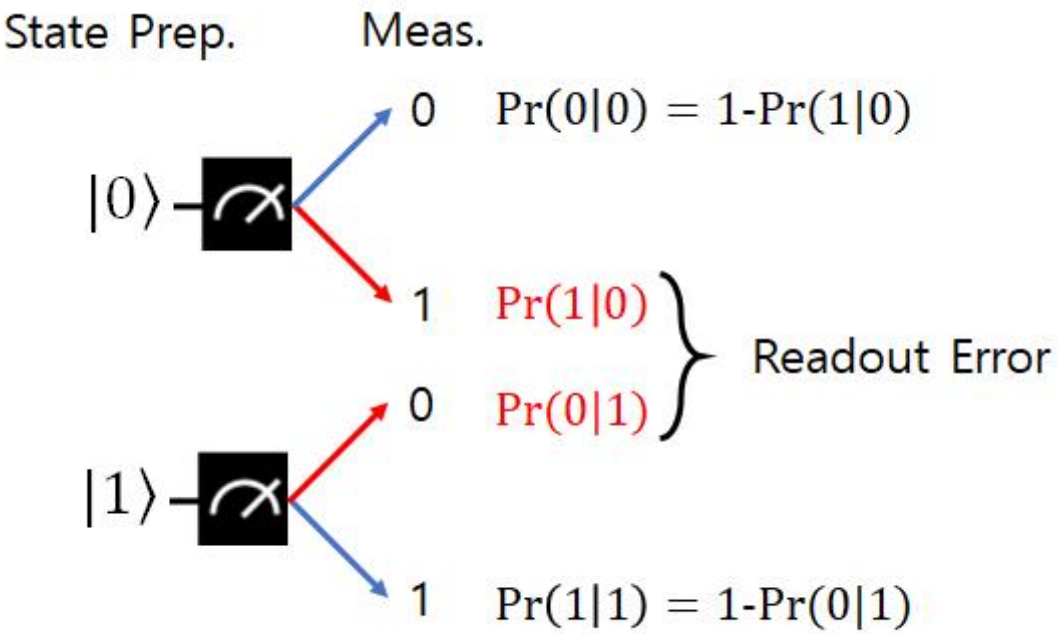
다음으로, Noisy backend에서 Read-out error를 보정할 수 있는 방법에 대해 살펴보겠습니다. Read-out error는 가장 간단하게 보정할 수 있는 noise 요소입니다.

(다른 advanced error mitigation 방법을 사용하기 위해서는 `mitiq` (github, arXiv:2009.04417) 라이브러리를 활용 할 수 있습니다.)

In [39]:
```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.test.mock import FakeRome
from qiskit.providers.aer import AerProvider
from qiskit.quantum_info import hellinger_fidelity
```

1-1. Read-out Error

Read-out error는 measurement 단계에서 나타나는 에러로써, 다음과 같이 준비된 $|0\rangle$ 또는 $|1\rangle$ 상태에 대해 반대로 측정되는 경우에 해당합니다.



Readout Error가 있을 때 실험을 통해 측정된 분포는 다음 식과 같이 주어집니다.

$$\begin{aligned}\Pr(\text{meas}=0) &= \Pr(\text{meas}=0|\text{prep}=0)\times\Pr(\text{prep}=0) \\ &\quad + \Pr(\text{meas}=0|\text{prep}=1)\times\Pr(\text{prep}=1) \\ \Pr(\text{meas}=1) &= \Pr(\text{meas}=1|\text{prep}=0)\times\Pr(\text{prep}=0) \\ &\quad + \Pr(\text{meas}=1|\text{prep}=1)\times\Pr(\text{prep}=1)\end{aligned}$$

$$\begin{pmatrix}\Pr(\text{m}=0)\\ \Pr(\text{m}=1)\end{pmatrix} = \begin{pmatrix}\Pr(0|0) & \Pr(0|1)\\ \Pr(1|0) & \Pr(1|1)\end{pmatrix}\begin{pmatrix}\Pr(\text{p}=0)\\ \Pr(\text{p}=1)\end{pmatrix}$$

여기서 측정 gate의 readout error 확률, $\Pr(\text{m}|\text{p})$을 알고 있다면, 측정된 결과의 분포$\Pr(\text{p})$로부터 readout error가 제거된 분포 $\Pr(\text{p})$를 알 수 있습니다.

$$\overrightarrow{\Pr}(\text{Meas}) = \mathbf{M}\overrightarrow{\Pr}(\text{Prep})$$

$$\overrightarrow{\Pr}(\text{Prep}) = \mathbf{M}^{-1}\overrightarrow{\Pr}(\text{Meas})$$

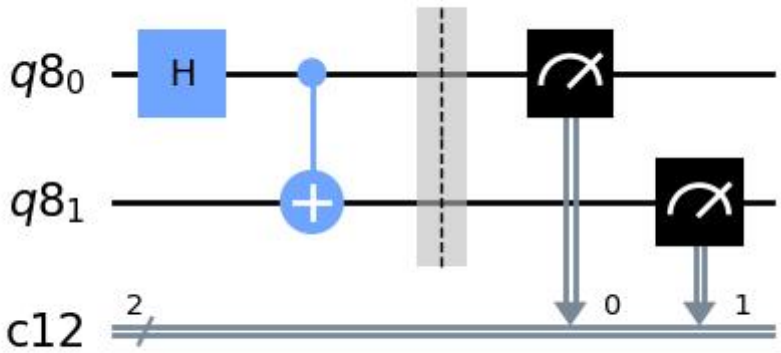
이 과정을 구현해보도록 하겠습니다.

2. Readout Error Mitigation (Without tools)

먼저 실험 대상으로, 다음 Bell state 회로를 준비하여 사용하겠습니다.

In [40]:
```
qr = QuantumRegister(2)
cr = ClassicalRegister(2)
qc = QuantumCircuit(qr, cr)

qc.h(0)
qc.cx(0,1)
qc.barrier()
qc.measure(qr, cr)

qc.draw('mpl')
```

Out[40]:



위 회로에 대하여 fake backend를 이용한 시뮬레이션을 실행하여 분포$\Pr(m)$를 구해보겠습니다.

In [41]:
```
num_shots = 4096

fake_rome = FakeRome()
job_fake_rome = fake_rome.run(qc, shots=4096)
result_fake_rome = job_fake_rome.result()
counts_fake_rome = result_fake_rome.get_counts()

print(counts_fake_rome)
pr_m = np.zeros(4, dtype=float)
for b, v in counts_fake_rome.items():
    pr_m[int(b, base=2)] = v/num_shots
print(pr_m)
```

```
{'00': 1889, '11': 1940, '01': 119, '10': 148}
[0.46118164 0.02905273 0.03613281 0.47363281]
```

read out error $\Pr(m|p)$를 측정하기 위해 다음과 같이 모든 computational basis에 대해 상태를 준비하여 실험을 수행합니다.

In [42]:
```
pr_m_p = np.zeros((4,4), dtype=float)
```

```
qr_mit = QuantumRegister(2)
cr_mit = ClassicalRegister(2)

# state prep = 00
qc_00 = QuantumCircuit(qr_mit, cr_mit)
qc_00.barrier()
qc_00.measure(qr_mit, cr_mit)
counts_00 = fake_rome.run(qc_00, shots=num_shots)W
    .result().get_counts()

pr_m_p[0][0] = counts_00['00']/num_shots if '00' in counts_00 else 0.0
pr_m_p[1][0] = counts_00['01']/num_shots if '01' in counts_00 else 0.0
pr_m_p[2][0] = counts_00['10']/num_shots if '10' in counts_00 else 0.0
pr_m_p[3][0] = counts_00['11']/num_shots if '11' in counts_00 else 0.0


# state prep = 01
qc_01 = QuantumCircuit(qr_mit, cr_mit)

qc_01.x(0)

qc_01.barrier()
qc_01.measure(qr_mit, cr_mit)
counts_01 = fake_rome.run(qc_01, shots=num_shots)W
    .result().get_counts()

pr_m_p[0][1] = counts_01['00']/num_shots if '00' in counts_01 else 0.0
pr_m_p[1][1] = counts_01['01']/num_shots if '01' in counts_01 else 0.0
pr_m_p[2][1] = counts_01['10']/num_shots if '10' in counts_01 else 0.0
pr_m_p[3][1] = counts_01['11']/num_shots if '11' in counts_01 else 0.0


# state prep = 10
qc_10 = QuantumCircuit(qr_mit, cr_mit)

qc_10.x(1)

qc_10.barrier()
qc_10.measure(qr_mit, cr_mit)
counts_10 = fake_rome.run(qc_10, shots=num_shots)W
    .result().get_counts()

pr_m_p[0][2] = counts_10['00']/num_shots if '00' in counts_10 else 0.0
pr_m_p[1][2] = counts_10['01']/num_shots if '01' in counts_10 else 0.0
pr_m_p[2][2] = counts_10['10']/num_shots if '10' in counts_10 else 0.0
pr_m_p[3][2] = counts_10['11']/num_shots if '11' in counts_10 else 0.0


# state prep = 11
qc_11 = QuantumCircuit(qr_mit, cr_mit)

qc_11.x(0)
qc_11.x(1)

qc_11.barrier()
qc_11.measure(qr_mit, cr_mit)
counts_11 = fake_rome.run(qc_11, shots=num_shots)W
    .result().get_counts()

pr_m_p[0][3] = counts_11['00']/num_shots if '00' in counts_11 else 0.0
pr_m_p[1][3] = counts_11['01']/num_shots if '01' in counts_11 else 0.0
pr_m_p[2][3] = counts_11['10']/num_shots if '10' in counts_11 else 0.0
pr_m_p[3][3] = counts_11['11']/num_shots if '11' in counts_11 else 0.0
```

```
np.set_printoptions(suppress=True)
print(pr_m_p)
```

```
[[0.95996094 0.03808594 0.03833008 0.0012207 ]
 [0.01977539 0.93725586 0.00024414 0.03100586]
 [0.02001953 0.0012207  0.93554688 0.03198242]
 [0.00024414 0.0234375  0.02587891 0.93579102]]
```

앞서 구한 실험 분포에 조건부확률 행렬의 역행렬을 곱하여 readout error가 보정된 결과를 얻을 수 있습니다.

In [43]:

```
pr_p = np.matmul(np.linalg.inv(pr_m_p), pr_m)
print("Before mitigation: ", pr_m)
print("After  mitigation: ", pr_p)
```

```
Before mitigation:  [0.46118164 0.02905273 0.03613281 0.47363281]
After  mitigation:  [0.4791668  0.00415884 0.01107895 0.50559541]
```

결과를 비교하기 위해 noiseless backend에서 시뮬레이션을 실행합니다.

In [44]:

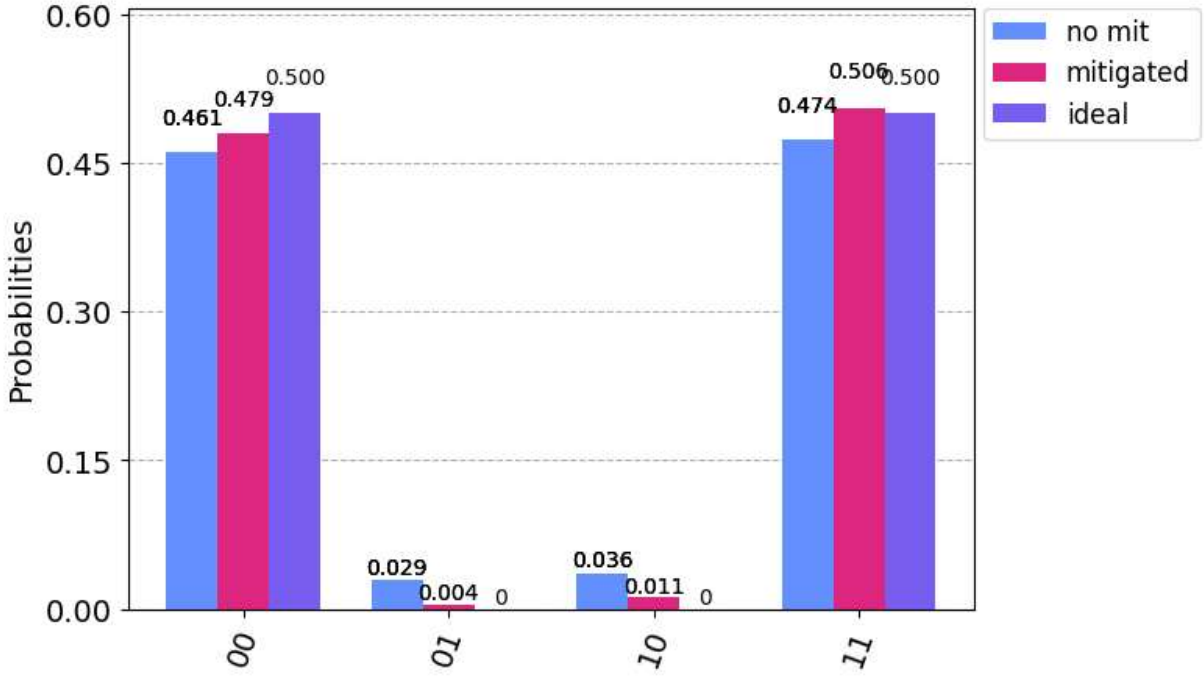
```
qasm_backend = AerProvider().get_backend("qasm_simulator")
counts_qasm = qasm_backend.run(qc, shots=num_shots).result().get_counts()

pr_ideal = np.zeros(4, dtype=float)
for b, v in counts_qasm.items():
    pr_ideal[int(b, base=2)] = v/num_shots
print("Ideal probs: ", pr_ideal)

# convert pr_p to dict form
counts_mit = dict()
for i, v in enumerate(pr_p):
    counts_mit[bin(i)[2:].rjust(2, '0')] = v * num_shots

print("Fidelity without mitigation: ", hellinger_fidelity(counts_fake_rome, counts_qas
print("Fidelity with mitigation: ", hellinger_fidelity(counts_mit, counts_qasm))
```

```
Ideal probs:  [0.49975586 0.         0.         0.50024414]
Fidelity without mitigation:  0.9347759748597199
Fidelity with mitigation:  0.9845912520029243
```

In [45]:

```
from qiskit.visualization import plot_histogram
plot_histogram([counts_fake_rome, counts_mit, counts_qasm],
               legend=['no mit', 'mitigated', 'ideal'])
```

Out[45]:



3. Readout Error Mitigation (With tools)

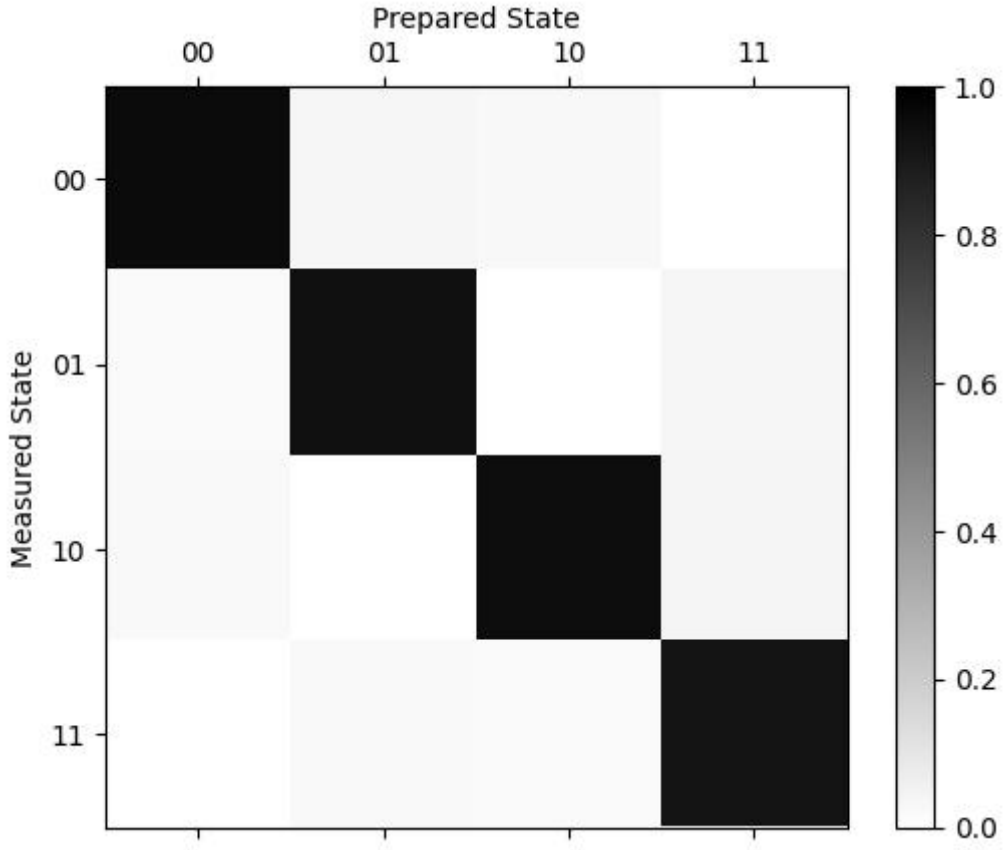
qiskit에서 제공하는 readout mitigation tool을 이용하여 위의 과정을 다음과 같이 쉽게 구현할 수도 있습니다.

In [46]:
```
from qiskit.ignis.mitigation import CompleteMeasFitter, complete_meas_cal

meas_calibs, state_labels = complete_meas_cal(qr=qr_mit, circlabel='mcal')
for ckt in meas_calibs:
    print(ckt)
```

```
q9_0: ─░─┤M├─────
       ░ └╥┘┌───┐
q9_1: ─░──╫─┤M├
       ░  ║ └╥┘
c14: 2/═══╩══╩═
          0  1
          ┌───┐ ░ ┌───┐
q9_0: ─┤ X ├─░─┤M├─────
       └───┘ ░ └╥┘┌───┐
q9_1: ──────────╫─┤M├
          ░  ║ └╥┘
c14: 2/═══╩══╩═
          0  1
               ┌───┐
q9_0: ─────────┤M├─────
          ┌───┐└╥┘┌───┐
q9_1: ─┤ X ├─╫─┤M├
       └───┘  ║ └╥┘
c14: 2/═══════╩══╩═
              0  1
          ┌───┐ ┌───┐
q9_0: ─┤ X ├─┤M├─────
       ├───┤ └╥┘┌───┐
q9_1: ─┤ X ├──╫─┤M├
       └───┘  ║ └╥┘
c14: 2/═══════╩══╩═
              0  1
```

In [47]:
```
job_calibs = fake_rome.run(meas_calibs, shots=num_shots)
meas_fitter = CompleteMeasFitter(job_calibs.result(), state_labels, circlabel='mcal')

print(meas_fitter.cal_matrix)
meas_fitter.plot_calibration()
```

```
[[0.95336914 0.03833008 0.03051758 0.00195312]
 [0.02148438 0.93652344 0.00048828 0.0378418 ]
 [0.02490234 0.00097656 0.94677734 0.0402832 ]
 [0.00024414 0.02416992 0.0222168  0.91992188]]
```



In [48]:
```
# Get the filter object
meas_filter = meas_fitter.filter

# Results with mitigation
mitigated_results = meas_filter.apply(result_fake_rome)
mitigated_counts = mitigated_results.get_counts(0)

print("Fidelity with mitigation: ", hellinger_fidelity(mitigated_counts, counts_qasm))
```

```
Fidelity with mitigation:  0.9964055019466439
```

5. Application - Quantum Phase Estimation

Author : Gwonhak Lee (gwonhak@gmail.com)

Qiskit에서 제공하는 양자알고리즘 솔루션 중 하나인 Quantum Phase Estimation을 활용해봅니다.

0. 필요한 요소 불러오기

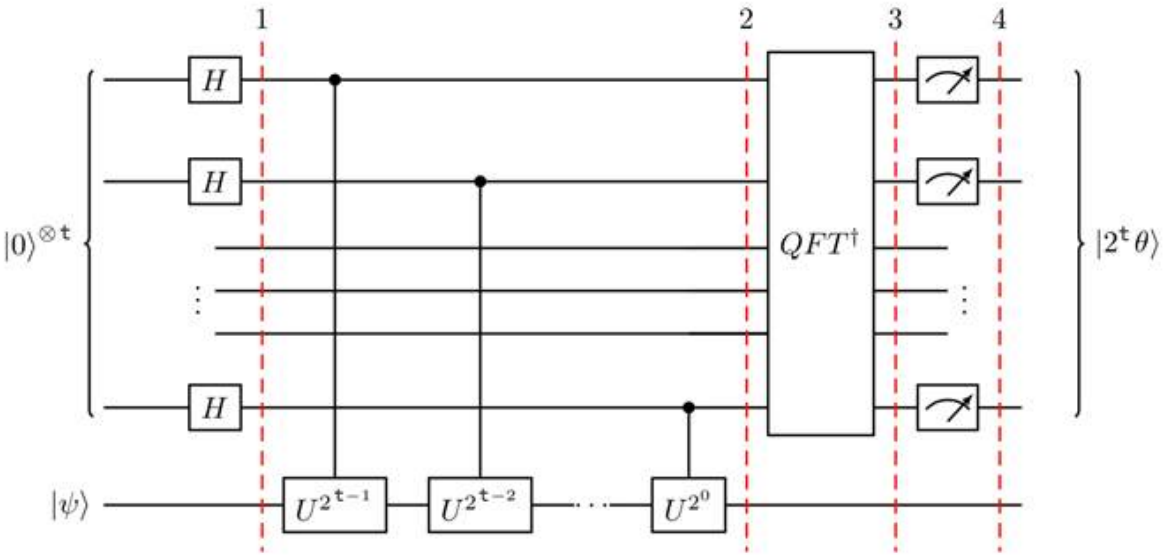
In [1]:
```
import numpy as np

from qiskit import transpile, QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.algorithms.phase_estimators import PhaseEstimation
from qiskit.providers.aer import AerProvider
from qiskit.tools.visualization import plot_histogram
from qiskit.utils import QuantumInstance
```

1. Quantum Phase Estimation

Quantum phase estimation은 Quantum Fourier Transform을 활용하여 Exponential speed-up을 달성하는 양자 알고리즘으로, 주어진 unitary matrix의 eigen phase를 계산합니다. (참고자료)

Phase estimation의 회로는 다음과 같습니다. 여기서 $|\psi\rangle$는 U의 eigenstate로써, 해당하는 eigenvalue는 $e^{i2\pi\theta}, \theta \in [0,1)$입니다.

(모든 Unitary 행렬의 eigenvalue는 모두 크기가 1인 복소수입니다.)



1. Hadamard 연산 후의 상태는 다음과 같습니다.

$$|0\rangle^{\otimes t} \otimes |\psi\rangle \rightarrow \frac{1}{\sqrt{2^t}} \sum_{k=0}^{2^t-1} |k\rangle \otimes |\psi\rangle$$

1. Controlled-U 를 위와 같이 적용하면, k의 binary 표현에서 most significant bit부터 차례대로 $U^{2^{t-1}}, U^{2^{t-2}}, \cdots, U^{2^0}$을 적용하기 때문에 다음과 같은 상태가 됩니다.

(note : $U^k|\psi\rangle = e^{i2k\pi\theta}$)

$$\frac{1}{\sqrt{2^t}}\sum_{k=0}^{2^t-1} e^{i2k\pi\theta}|k\rangle \otimes |\psi\rangle$$

1. 다음으로, Inverse Quantum Fourier Transform을 취하면,

$$\frac{1}{2^t}\sum_{x=0}^{2^t-1}\sum_{k=0}^{2^t-1} e^{-\frac{i2\pi k}{2^t}(x-2^t\theta)}|x\rangle \otimes |\psi\rangle$$

여기서 $|x\rangle$ 을 측정하면 $x \sim 2^t\theta$에서 가장 빈번하게 발생함을 알 수 있습니다.

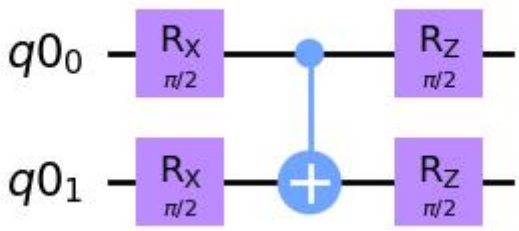
Qiskit에서는 quantum phase estimation을 쉽게 구현할 수 있는 방법이 포함되어 있습니다. 이를 이용하는 실습을 진행합니다.

먼저, Eigen phase를 계산할 unitary 회로를 준비합니다.

In [2]:
```
num_unitary_qubits = 2

unitary_qubit = QuantumRegister(num_unitary_qubits)
unitary_circuit = QuantumCircuit(unitary_qubit)
for i in range(num_unitary_qubits):
    unitary_circuit.rx(0.5*np.pi, i)
for i in range(num_unitary_qubits - 1):
    unitary_circuit.cx(i, i+1)
for i in range(num_unitary_qubits):
    unitary_circuit.rz(0.5*np.pi, i)

print("depth = ", unitary_circuit.depth())
unitary_circuit.draw('mpl')
```

```
depth =  3
```

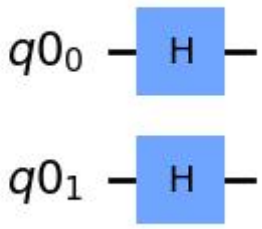
Out[2]:



다음으로, 초기 상태를 준비하기 위한 회로를 준비합니다. 이 때, 초기 상태는 관심있는 eigen state와 overlap이 많을 것으로 가정합니다.

In [3]:
```
state_prep = QuantumCircuit(unitary_qubit)
state_prep.h(unitary_qubit)
state_prep.draw('mpl')
```

Out[3]:



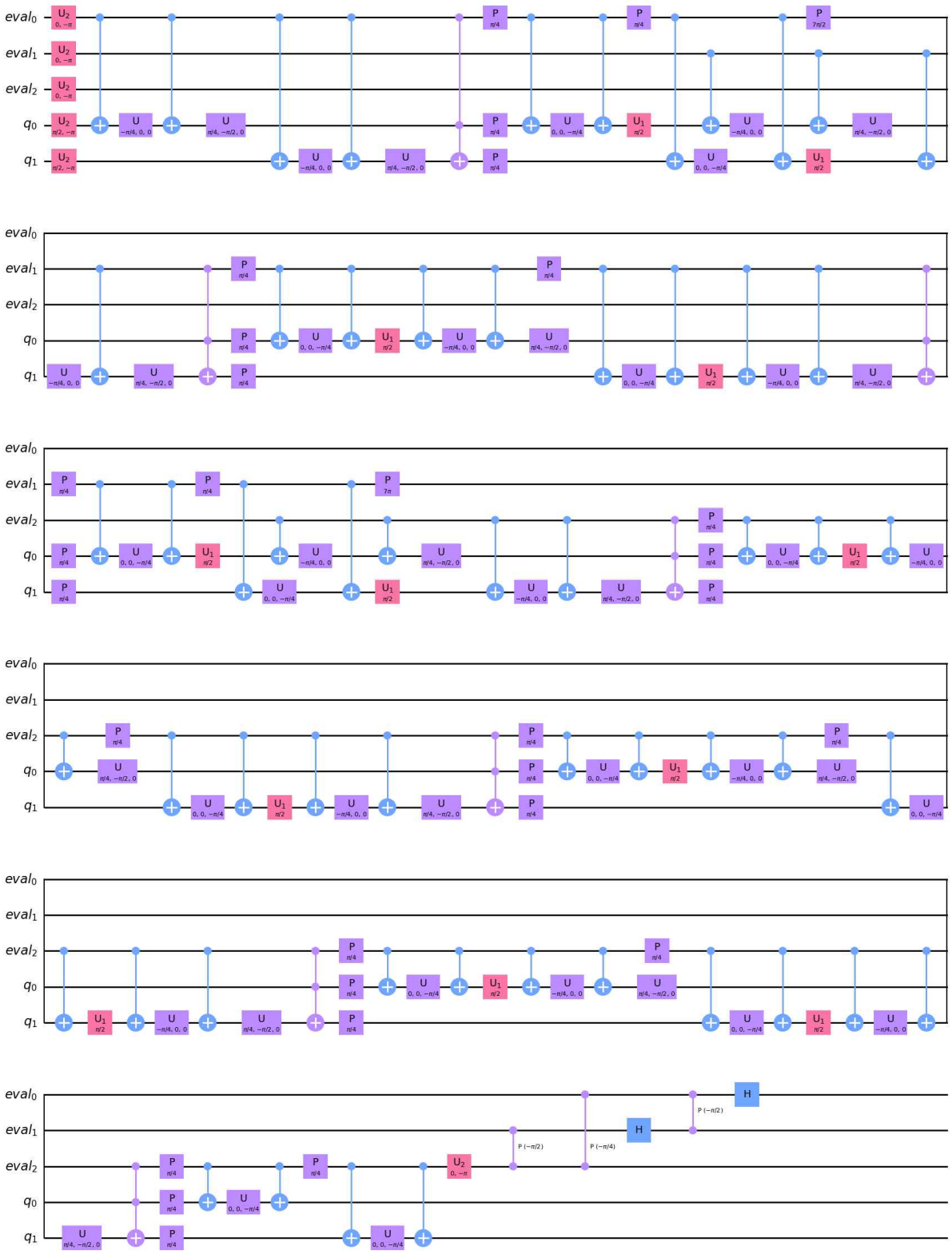
evaluation qubit의 갯수를 지정하고, Phase Estimation 회로를 구현합니다.

In [4]:

```
num_evaluation_qubits = 3

qasm_backend = AerProvider().get_backend("qasm_simulator")
qpe = PhaseEstimation(num_evaluation_qubits=num_evaluation_qubits,
                      quantum_instance=QuantumInstance(backend=qasm_backend,
                                                       shots=4096,
                                                       optimization_level=3))

ckt = qpe.construct_circuit(unitary_circuit, state_preparation=state_prep)
ckt = transpile(ckt, backend=qasm_backend)
print(f"depth = {ckt.depth()}")
ckt.draw('mpl')
```

```
depth = 122
```

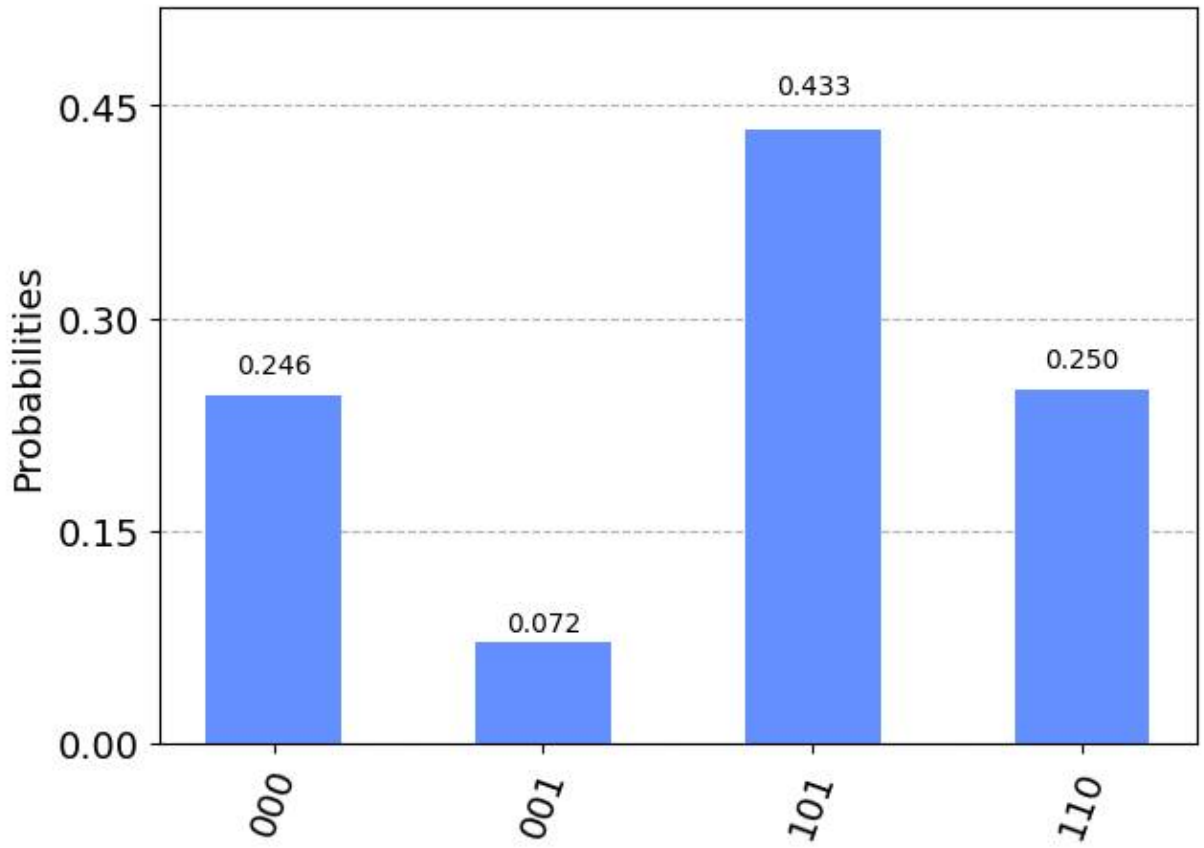
Out[4]:



QPE 알고리즘을 실행하고 결과를 출력합니다.

In [5]:
```
res = qpe.estimate(unitary_circuit, state_prep)
plot_histogram(res.phases)
```

Out[5]:



In [6]:
```
phase_list = sorted(res.phases.keys(), key=lambda x: res.phases[x], reverse=True)
for p in sorted(phase_list):
    bin_p = int(p, 2) / 2**num_evaluation_qubits
    print(f"phase[rad] : {2*bin_p*np.pi}, occurence : {res.phases[p]}")
```

```
phase[rad] : 0.0, occurence : 0.24560546875
phase[rad] : 0.7853981633974483, occurence : 0.072021484375
phase[rad] : 3.9269908169872414, occurence : 0.4326171875
phase[rad] : 4.71238898038469, occurence : 0.249755859375
```

실험으로 얻은 값을 비교하기 위해 unitary simulator를 이용하여 unitary 회로의 행렬 형태를 얻은 뒤, eigen problem의 해를 구한다.

In [7]:
```
unitary_simulator = AerProvider().get_backend('unitary_simulator')
job_unitary = unitary_simulator.run(unitary_circuit)
mat = job_unitary.result().get_unitary()
print(mat)
```

```
[[ 0. -0.5j -0.5+0.j  -0.5+0.j   0. +0.5j]
 [-0.5+0.j   0. -0.5j  0. -0.5j  0.5+0.j ]
 [ 0. -0.5j -0.5+0.j   0.5+0.j   0. -0.5j]
 [ 0.5+0.j   0. +0.5j  0. -0.5j  0.5+0.j ]]
```

In [8]:
```
eigval, eigvec = np.linalg.eig(mat)
assert all(np.isclose(np.abs(eigval), 1.0))
eigph = np.angle(eigval)
for i, p in enumerate(eigph):
    if p < 0:
        eigph[i] = 2*np.pi + p
print("eig phase = ", sorted(eigph))
for i, v in enumerate(eigvec):
    print(f"eigvec {i} = {v/np.linalg.norm(v, ord=2)}")
```

```
eig phase =  [0.7853981633974485, 3.9269908169872414, 4.71238898038469, 6.283185307179
```

```
586]
eigvec 0 = [ 6.53281482e-01+1.50455499e-17j -5.00000000e-01+6.49835883e-16j
  2.44494543e-16+5.00000000e-01j -2.70598050e-01+5.69363576e-17j]
eigvec 1 = [ 6.53281482e-01+0.00000000e+00j  5.00000000e-01+0.00000000e+00j
 -1.94159624e-16-5.00000000e-01j -2.70598050e-01-5.05517184e-17j]
eigvec 2 = [ 2.70598050e-01-1.94796269e-16j -1.37484272e-16-5.00000000e-01j
  5.00000000e-01+0.00000000e+00j  6.53281482e-01+0.00000000e+00j]
eigvec 3 = [-2.70598050e-01-1.57807601e-16j -1.46825868e-16-5.00000000e-01j
  5.00000000e-01-6.09556052e-16j -6.53281482e-01+6.41820141e-17j]
```

(추가) Transpile optimization

fake rome backend를 활용하여 transpile optimization 과정의 차이를 살펴보겠습니다.

In [9]:
```
from qiskit.test.mock import FakeRome
fake_rome = FakeRome()

# 측정게이트 추가
eval_qubits = ckt.qregs[0]
creg = ClassicalRegister(len(eval_qubits))
ckt.add_register(creg)
ckt.measure(eval_qubits, creg)
```

Out[9]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x2871e03f8e0>
```

In [10]:
```
ckt_lv0 = transpile(ckt, backend=fake_rome, optimization_level=0)
ckt_lv1 = transpile(ckt, backend=fake_rome, optimization_level=1)
ckt_lv2 = transpile(ckt, backend=fake_rome, optimization_level=2)
ckt_lv3 = transpile(ckt, backend=fake_rome, optimization_level=3)
```

In [11]:
```
for i, ckt_opt in enumerate([ckt_lv0, ckt_lv1, ckt_lv2, ckt_lv3]):
    cnot_count = ckt_opt.count_ops()['cx'] if 'cx' in ckt_opt.count_ops() else 0
    print(f"opt_level={i}, depth={ckt_opt.depth()}, cnot_count={cnot_count}")
```

```
opt_level=0, depth=525, cnot_count=221
opt_level=1, depth=332, cnot_count=202
opt_level=2, depth=321, cnot_count=194
opt_level=3, depth=314, cnot_count=138
```

6. Exercises

Author : Gwonhak Lee (gwonhak@gmail.com)

앞서 배운 내용들을 활용하여 간단한 예제 문제들을 해결해보겠습니다.

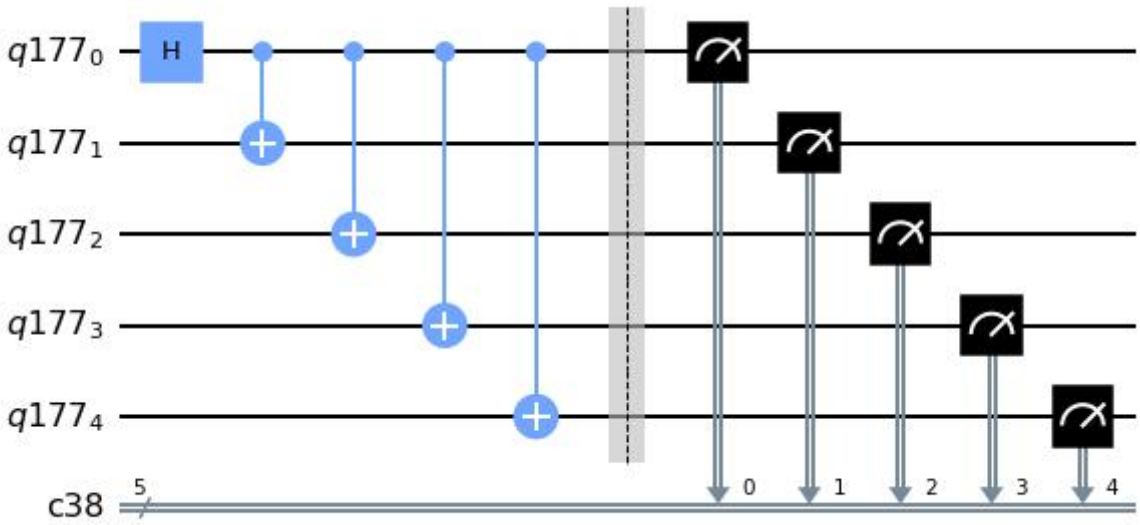
In [66]:
```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, execute, tran
from qiskit.providers.aer import AerProvider
from qiskit.tools.visualization import plot_histogram
import random
import numpy as np
```

Exercise 1. n-GHZ 상태 측정

n-Qubit에 대한 GHZ 상태를 준비하고, 측정하는 회로를 구성하고, histogram을 출력합니다.

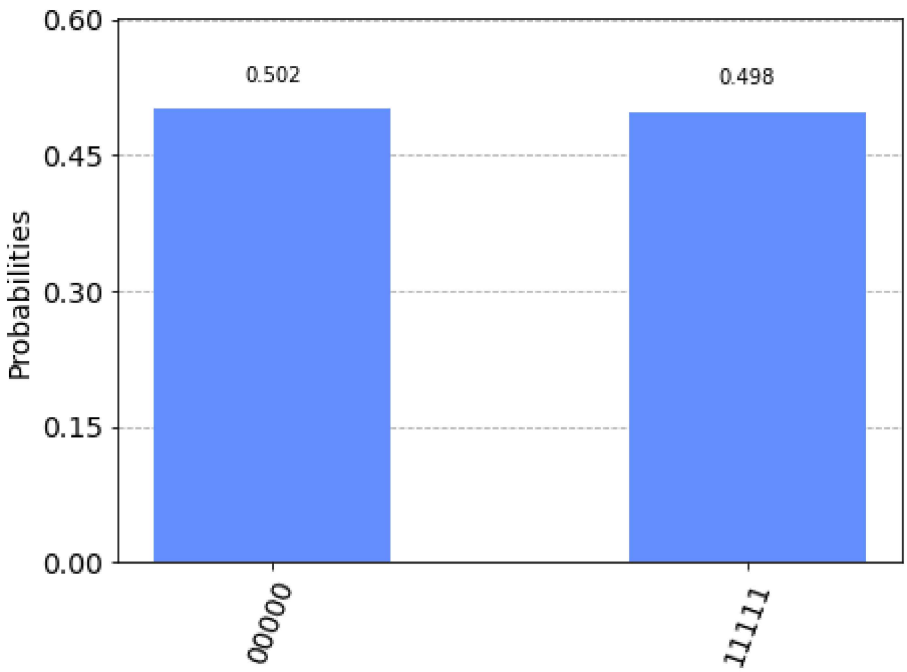
$$|GHZ_n\rangle = \frac{1}{\sqrt{2}}(|0\cdots00\rangle + |1\cdots11\rangle)$$

In [67]:
```
def ghz(n):
    # 객체의 선언
    qr_ghz = QuantumRegister(n)
    cr_ghz = ClassicalRegister(n)
    qc_ghz = QuantumCircuit(qr_ghz, cr_ghz)

    # 회로 구성
    qc_ghz.h(qr_ghz[0])
    for i in range(1, n):
        qc_ghz.cx(0, i)

    qc_ghz.barrier()
    qc_ghz.measure(qr_ghz, cr_ghz)

    return qc_ghz

# 5-ghz 회로 출력
qc = ghz(5)
qc.draw('mpl');
```

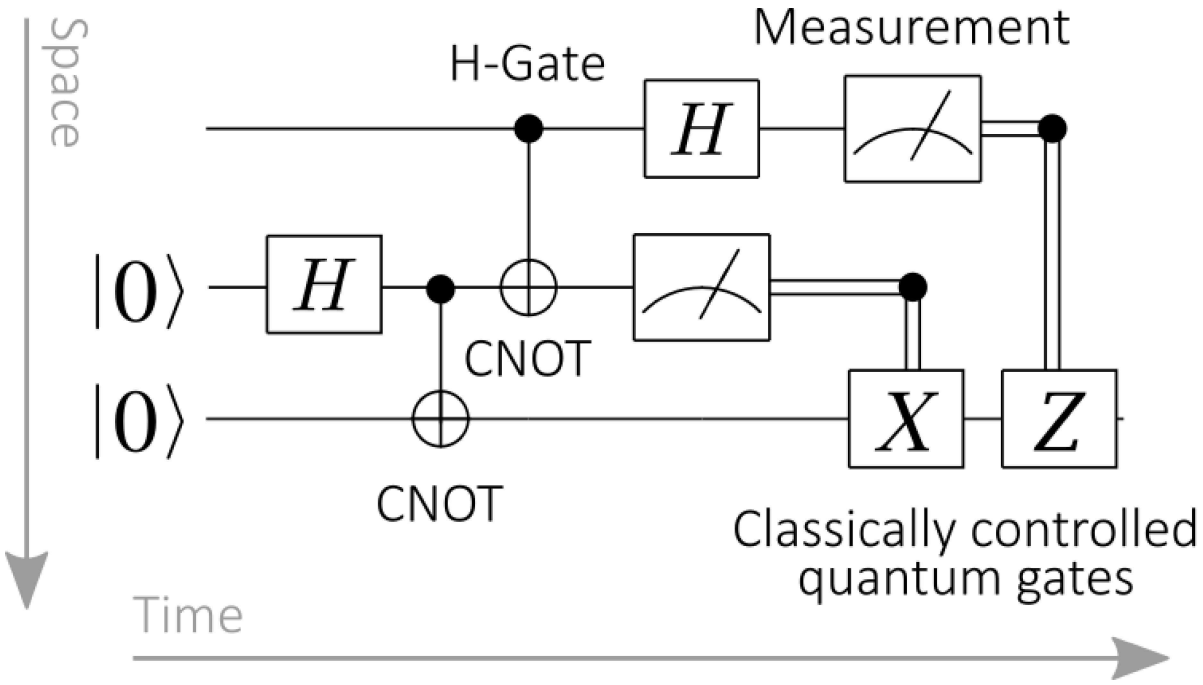


In [68]:

```
# qasm 시뮬레이션
qasm_simulator = AerProvider().get_backend("qasm_simulator")
job_qasm = execute(qc, backend=qasm_simulator, shots=1024)
counts = job_qasm.result().get_counts()

# 시각화
plot_histogram(counts);
```



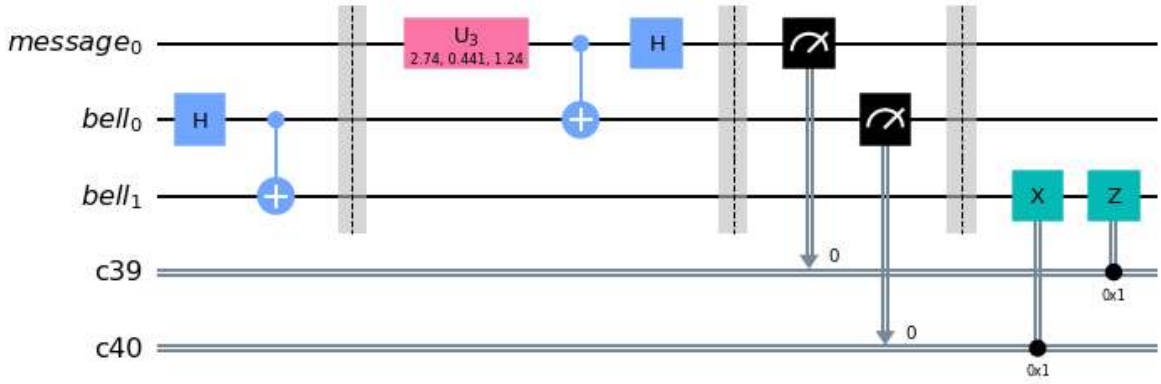
Exercise 2. Quantum Teleportation



hint : use .c_if()

In [69]:

```
bell_qr = QuantumRegister(2, 'bell')
message_qr = QuantumRegister(1, 'message')
cr1 = ClassicalRegister(1)
cr2 = ClassicalRegister(1)
qtel_qc = QuantumCircuit(message_qr, bell_qr, cr1, cr2)

a1, a2, a3 = ( random.random() * np.pi for _ in range(3) )

# complete the circuit
qtel_qc.h(bell_qr[0])
qtel_qc.cx(bell_qr[0], bell_qr[1])
qtel_qc.barrier()

qtel_qc.u3(a1, a2, a3, message_qr)

qtel_qc.cx(message_qr, bell_qr[0])
qtel_qc.h(message_qr)
qtel_qc.barrier()

qtel_qc.measure(message_qr, cr1)
qtel_qc.measure(bell_qr[0], cr2)
qtel_qc.barrier()

qtel_qc.x(bell_qr[1]).c_if(cr2, 1)
qtel_qc.z(bell_qr[1]).c_if(cr1, 1)

qtel_qc.draw('mpl');
```



In [70]:
```
sv_simulator = AerProvider().get_backend('statevector_simulator')
job_sv = sv_simulator.run(qtel_qc)
sv = job_sv.result().get_statevector()
print(sv)
print([sum(sv[:4]), sum(sv[4:])])
```

```
[ 0.         +0.j          0.         -0.j          0.         +0.j
  0.20042987+0.j         -0.         +0.j         -0.         +0.j
 -0.         +0.j          0.88608591+0.41794691j]
[(0.2004298720534752+0j), (0.8860859135671616+0.4179469107033526j)]
```

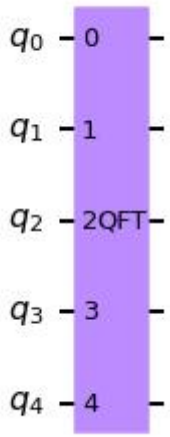
In [71]:
```
verif_ckt = QuantumCircuit(1)
verif_ckt.u3(a1, a2, a3, 0)
sv_verif = sv_simulator.run(verif_ckt).result().get_statevector()
print(sv_verif)
```

```
[0.20042987+0.j         0.88608591+0.41794691j]
```

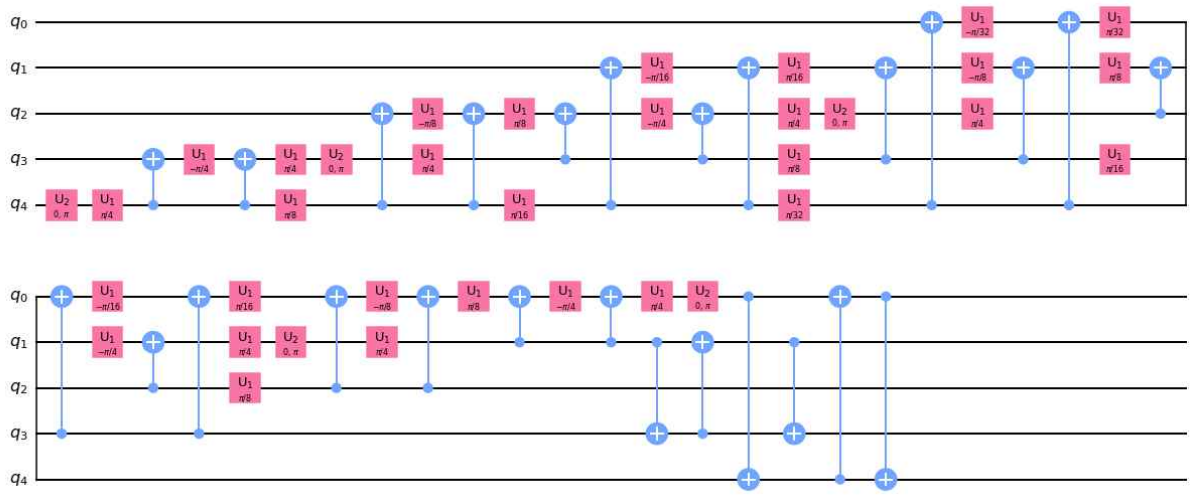
Exercise 3. Quantum Fourier Transform

qft회로를 transpile하여 u1, u2, 그리고 cx 게이트로 표현합니다.

In [72]:
```
from qiskit.circuit.library.basis_change.qft import QFT

qft_ckt = QFT(5)
qft_ckt.draw("mpl");
```



In [73]:
```
qft_ckt_tr = transpile(qft_ckt, basis_gates=['u1', 'u2', 'cx'])
qft_ckt_tr.draw('mpl');
```



qft회로를 Linear connectivity Layout에 transpile합니다.

In [76]:
```
coupling_map = list()
for i in range(5 - 1):
    coupling_map.append([i, i+1])
    coupling_map.append([i+1, i])
qft_ckt_tr_lin = transpile(qft_ckt, basis_gates=['u1', 'u2', 'cx'],
                           coupling_map=coupling_map)
print(coupling_map)
qft_ckt_tr_lin.draw('mpl');
```

```
[[0, 1], [1, 0], [1, 2], [2, 1], [2, 3], [3, 2], [3, 4], [4, 3]]
```