In [1]:

```python
# Import the qiskit library
import math
import numpy as np
import scipy as sp
import matplotlib.pylab as plt
import warnings

warnings.filterwarnings("ignore")

from qiskit import QuantumCircuit
from qiskit.circuit.library import PauliEvolutionGate
from qiskit.primitives import StatevectorEstimator
from qiskit.quantum_info import Statevector, SparsePauliOp
from qiskit.synthesis import (
    SuzukiTrotter,
    MatrixExponential,
    QDrift,
    ProductFormula,
    LieTrotter
)
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

from qiskit_ibm_runtime import QiskitRuntimeService, SamplerV2
```

In [2]:
```python
def get_hamiltonian(nqubits, J, h, alpha):

    # List of Hamiltonian terms as 3-tuples containing
    # (1) the Pauli string,
    # (2) the qubit indices corresponding to the Pauli string,
    # (3) the coefficient.
    ZZ_tuples = [("ZZ", [i, i + 1], -J) for i in range(0, nqubits - 1)]
    Z_tuples = [("Z", [i], -h * np.sin(alpha)) for i in range(0, nqubits)]
    X_tuples = [("X", [i], -h * np.cos(alpha)) for i in range(0, nqubits)]

    # We create the Hamiltonian as a SparsePauliOp, via the method
    # `from_sparse_list`, and multiply by the interaction term.
    hamiltonian = SparsePauliOp.from_sparse_list([*ZZ_tuples, *Z_tuples, *X_tuples], num_qubits=nqubits
    return hamiltonian.simplify()
```

In [5]:
```python
n_qubits = 6

hamiltonian = get_hamiltonian(nqubits=n_qubits, J=0.2, h=1.2, alpha=np.pi/8.0)
hamiltonian
```

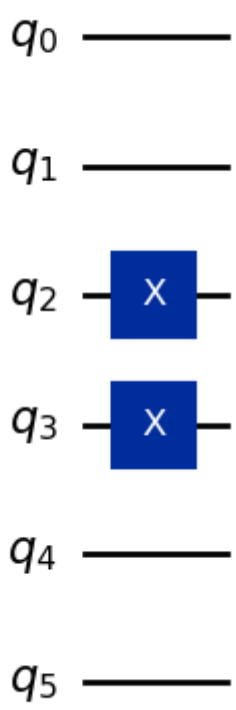
Out[5]:
```
SparsePauliOp(['IIIIZZ', 'IIIZZI', 'IIZZII', 'IZZIII', 'ZZIIII', 'IIIIIZ', 'IIIIZI', 'IIIZII', 'IIZIII', '
IZIIII', 'ZIIIII', 'IIIIIX', 'IIIIXI', 'IIIXII', 'IIXIII', 'IXIIII', 'XIIIII'],
              coeffs=[-0.2       +0.j, -0.2       +0.j, -0.2       +0.j, -0.2       +0.j,
 -0.2       +0.j, -0.45922012+0.j, -0.45922012+0.j, -0.45922012+0.j,
 -0.45922012+0.j, -0.45922012+0.j, -0.45922012+0.j, -1.10865544+0.j,
 -1.10865544+0.j, -1.10865544+0.j, -1.10865544+0.j, -1.10865544+0.j,
 -1.10865544+0.j])
```

In [6]:
```python
num_timesteps = 60
evolution_time = 30.0
dt = evolution_time / num_timesteps
product_formula_lt = LieTrotter()
product_formula_st2 = SuzukiTrotter(order=2)
product_formula_st4 = SuzukiTrotter(order=4)
```

In [7]:

```
initial_circuit = QuantumCircuit(n_qubits)
initial_circuit.prepare_state('001100')
# Change reps and see the difference when you decompose the circuit
initial_circuit.decompose(reps=1).draw("mpl")
```

Out[7]:



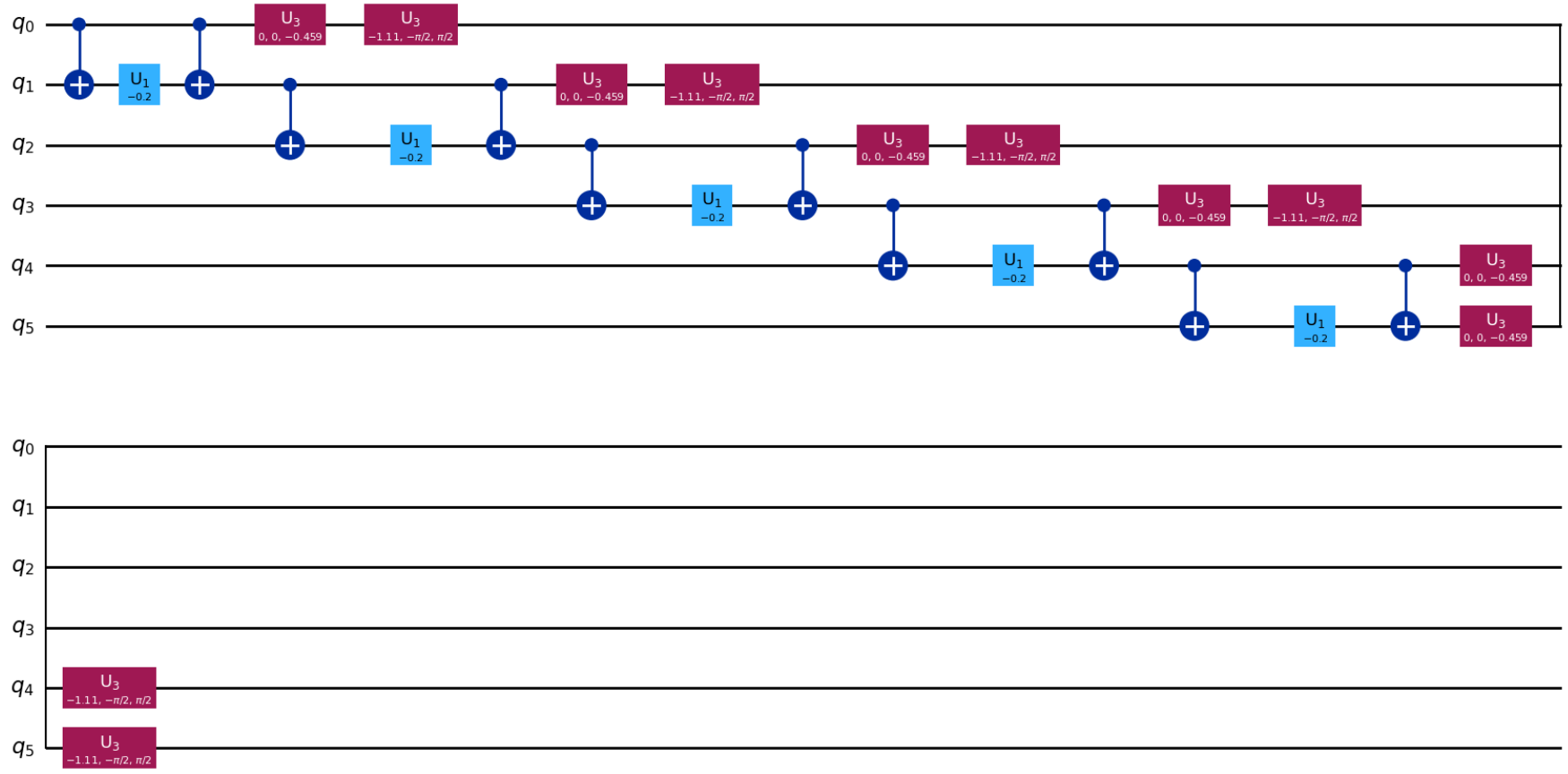
In [8]:

```
single_step_evolution_gates_lt = PauliEvolutionGate(
    hamiltonian, dt, synthesis=product_formula_lt
)
single_step_evolution_lt = QuantumCircuit(n_qubits)
single_step_evolution_lt.append(single_step_evolution_gates_lt, single_step_evolution_lt.qubits)

print(
    f"""
Trotter step with Lie-Trotter
-----------------------------
Depth: {single_step_evolution_lt.decompose(reps=3).depth()}
Gate count: {len(single_step_evolution_lt.decompose(reps=3))}
Nonlocal gate count: {single_step_evolution_lt.decompose(reps=3).num_nonlocal_gates()}
Gate breakdown: {", ".join([f"{k.upper()}: {v}" for k, v in single_step_evolution_lt.decompose(reps=3).c
"""
)
single_step_evolution_lt.decompose(reps=3).draw("mpl")
```

```
Trotter step with Lie-Trotter
-----------------------------
Depth: 17
Gate count: 27
Nonlocal gate count: 10
Gate breakdown: U3: 12, CX: 10, U1: 5
```

Out[8]: Global Phase: 1.8776603565143235



In [9]:

```
magnetization= SparsePauliOp.from_sparse_list(
    [("Z", [i], 1.0) for i in range(0, n_qubits)], num_qubits=n_qubits
) / n_qubits
correlation = SparsePauliOp.from_sparse_list(
    [("ZZ", [i, i + 1], 1.0) for i in range(0, n_qubits - 1)], num_qubits=n_qubits
) / (n_qubits - 1)
print('magnetization : ', magnetization)
print('correlation : ', correlation)
```

```
magnetization :  SparsePauliOp(['IIIIIZ', 'IIIIZI', 'IIIZII', 'IIZIII', 'IZIIII', 'ZIIIII'],
              coeffs=[0.16666667+0.j, 0.16666667+0.j, 0.16666667+0.j, 0.16666667+0.j,
 0.16666667+0.j, 0.16666667+0.j])
correlation :  SparsePauliOp(['IIIIZZ', 'IIIZZI', 'IIZZII', 'IZZIII', 'ZZIIII'],
              coeffs=[0.2+0.j, 0.2+0.j, 0.2+0.j, 0.2+0.j, 0.2+0.j])
```

In [10]:

```
# Initiate the circuit
evolved_state = QuantumCircuit(initial_circuit.num_qubits)
# Start from the initial spin configuration
evolved_state.append(initial_circuit, evolved_state.qubits)
# Initiate Estimator (V2)
estimator= StatevectorEstimator()
# Set number of shots
shots = 10000
# Translate the precision required from the number of shots
precision = np.sqrt(1 / shots)
energy_list = []
mag_list = []
corr_list = []
# Estimate expectation values for t=0.0
job = estimator.run([(evolved_state, [hamiltonian, magnetization, correlation])], precision=precision)
# Get estimated expectation values
evs = job.result()[0].data.evs
energy_list.append(evs[0])
mag_list.append(evs[1])
corr_list.append(evs[2])
# Start time evolution
for n in range(num_timesteps):
    # Expand the circuit to describe delta-t
    evolved_state.append(single_step_evolution_gates_lt, evolved_state.qubits)
    # Estimate expectation values at delta-t
    job = estimator.run([(evolved_state, [hamiltonian, magnetization, correlation])], precision=precisi
    # Retrieve results (expectation values)
    evs = job.result()[0].data.evs
    energy_list.append(evs[0])
    mag_list.append(evs[1])
    corr_list.append(evs[2])
# Transform the list of expectation values (at each time step) to arrays
energy_array = np.array(energy_list)
mag_array = np.array(mag_list)
corr_array = np.array(corr_list)
```

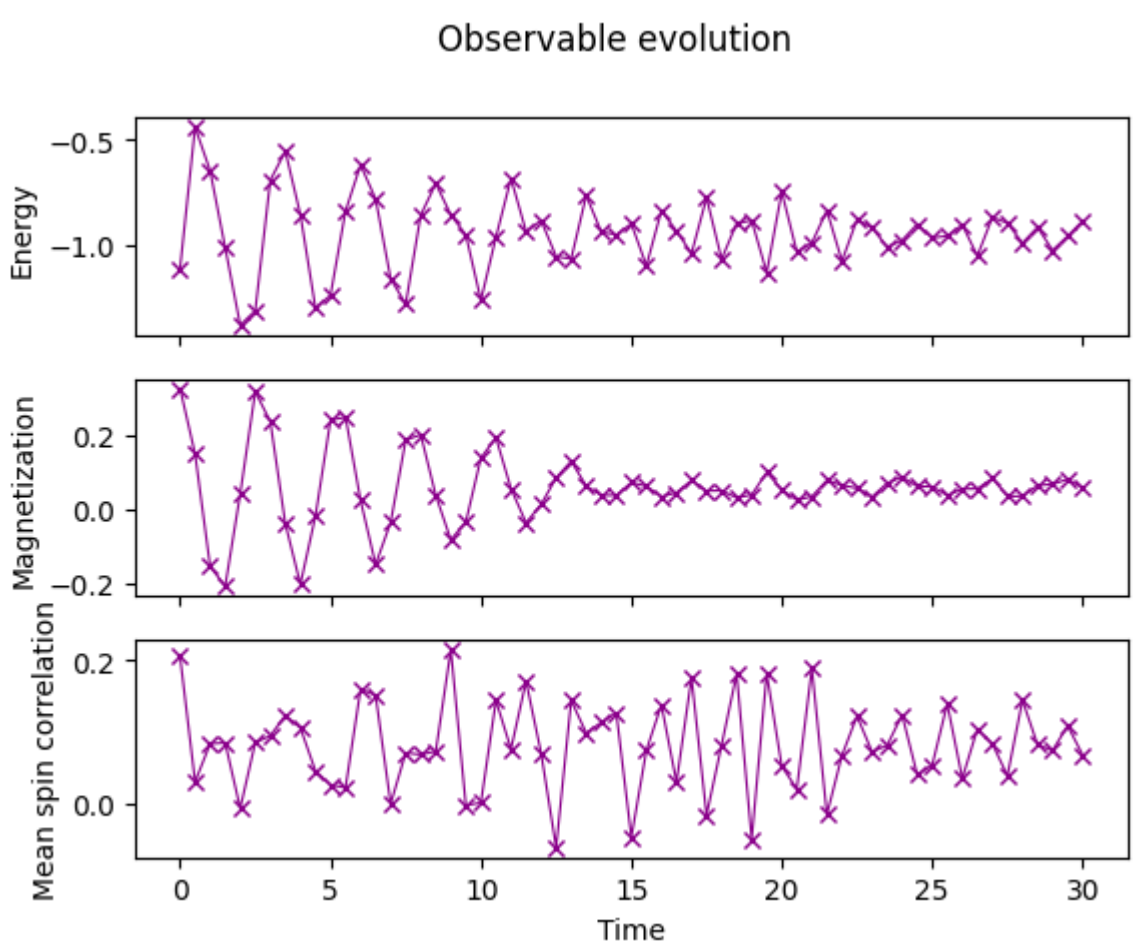
In [12]:

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(3, sharex=True)
times = np.linspace(0, evolution_time, num_timesteps + 1)  # includes initial state
axes[0].plot(
    times, energy_array, label="First order", marker="x", c="darkmagenta", ls="-", lw=0.8
)
axes[1].plot(
    times, mag_array , label="First order", marker="x", c="darkmagenta", ls="-", lw=0.8
)
axes[2].plot(
    times, corr_array, label="First order", marker="x", c="darkmagenta", ls="-", lw=0.8
)
axes[0].set_ylabel("Energy")
axes[1].set_ylabel("Magnetization")
axes[2].set_ylabel("Mean spin correlation")
axes[2].set_xlabel("Time")
fig.suptitle("Observable evolution")
```

Out[12]:

```
Text(0.5, 0.98, 'Observable evolution')
```

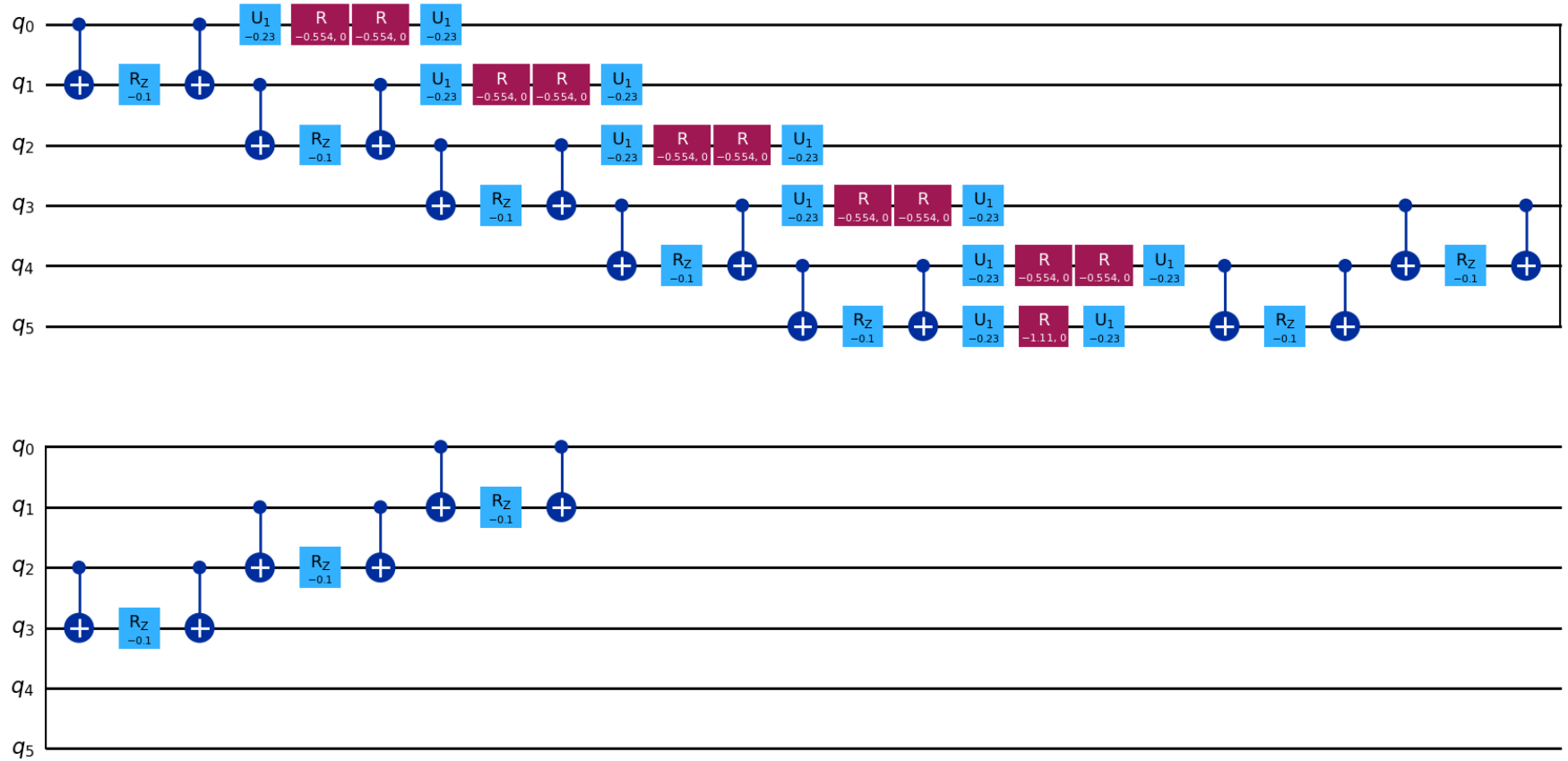

Observable evolution
Energy
−0.5
−1.0
Magnetization
0.2
0.0
−0.2
Mean spin correlation
0.2
0.0
0
5
10
15
20
25
30
Time

In [13]:

```
# Modify the line below (Use PauliEvolutionGate)
single_step_evolution_gates_st2 = PauliEvolutionGate(
    hamiltonian, dt, synthesis=product_formula_st2
)
single_step_evolution_st2 = QuantumCircuit(n_qubits)
single_step_evolution_st2.append(single_step_evolution_gates_st2, single_step_evolution_st2.qubits)
# Let us print some stats
print(
    f"""
Trotter step with second-order Suzuki-Trotter
------------------------------
Depth: {single_step_evolution_st2.decompose(reps=3).depth()}
Gate count: {len(single_step_evolution_st2.decompose(reps=3))}
Nonlocal gate count: {single_step_evolution_st2.decompose(reps=3).num_nonlocal_gates()}
Gate breakdown: {", ".join([f"{k.upper()}: {v}" for k, v in single_step_evolution_st2.decompose(reps=3)
"""
)
single_step_evolution_st2.decompose(reps=2).draw("mpl")
```

```
Trotter step with second-order Suzuki-Trotter
------------------------------
Depth: 34
Gate count: 53
Nonlocal gate count: 20
Gate breakdown: U3: 23, CX: 20, U1: 10
```

Out[13]: Global Phase: 1.3776603565143228



In [14]:

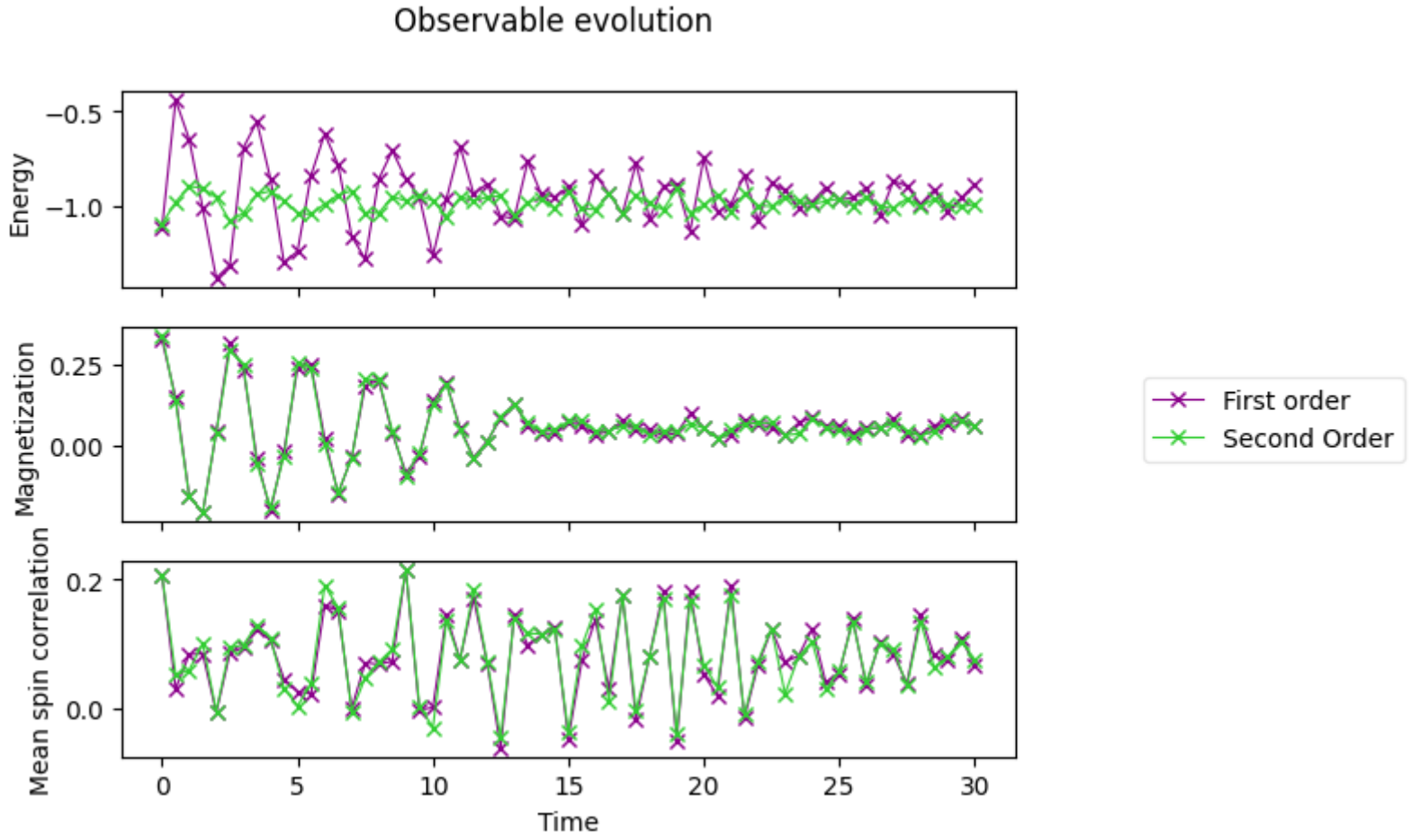
```
# Initiate the circuit
evolved_state = QuantumCircuit(initial_circuit.num_qubits)
# Start from the initial spin configuration
evolved_state.append(initial_circuit, evolved_state.qubits)
# Initiate Estimator (V2)
estimator= StatevectorEstimator()
# Set number of shots
shots = 10000
# Translate the precision required from the number of shots
precision = np.sqrt(1 / shots)
energy_list_st2 = []
mag_list_st2 = []
corr_list_st2 = []
# Estimate expectation values for t=0.0
job = estimator.run([(evolved_state, [hamiltonian, magnetization, correlation])], precision=precision)
# Get estimated expectation values
evs = job.result()[0].data.evs
energy_list_st2.append(evs[0])
mag_list_st2.append(evs[1])
corr_list_st2.append(evs[2])
# Start time evolution
for n in range(num_timesteps):
    # Expand the circuit to describe delta-t
    evolved_state.append(single_step_evolution_gates_st2, evolved_state.qubits)
    # Estimate expectation values at delta-t
    job = estimator.run([(evolved_state, [hamiltonian, magnetization, correlation])], precision=precisi
    # Retrieve results (expectation values)
    evs = job.result()[0].data.evs
    energy_list_st2.append(evs[0])
    mag_list_st2.append(evs[1])
    corr_list_st2.append(evs[2])
# Transform the list of expectation values (at each time step) to arrays
energy_array_st2 = np.array(energy_list_st2)
mag_array_st2 = np.array(mag_list_st2)
corr_array_st2 = np.array(corr_list_st2)
```

In [15]:

```
axes[0].plot(
    times, energy_array_st2, label="Second Order", marker="x", c="limegreen", ls="-", lw=0.8
)
axes[1].plot(
    times, mag_array_st2, label="Second Order", marker="x", c="limegreen", ls="-", lw=0.8
)
axes[2].plot(
    times, corr_array_st2, label="Second Order", marker="x", c="limegreen", ls="-", lw=0.8
)

# Replace the legend
# legend.remove()
legend = fig.legend(
    *axes[0].get_legend_handles_labels(),
    bbox_to_anchor=(1.0, 0.5),
    loc="center left",
    framealpha=0.5,
)
fig
```

Out[15]:



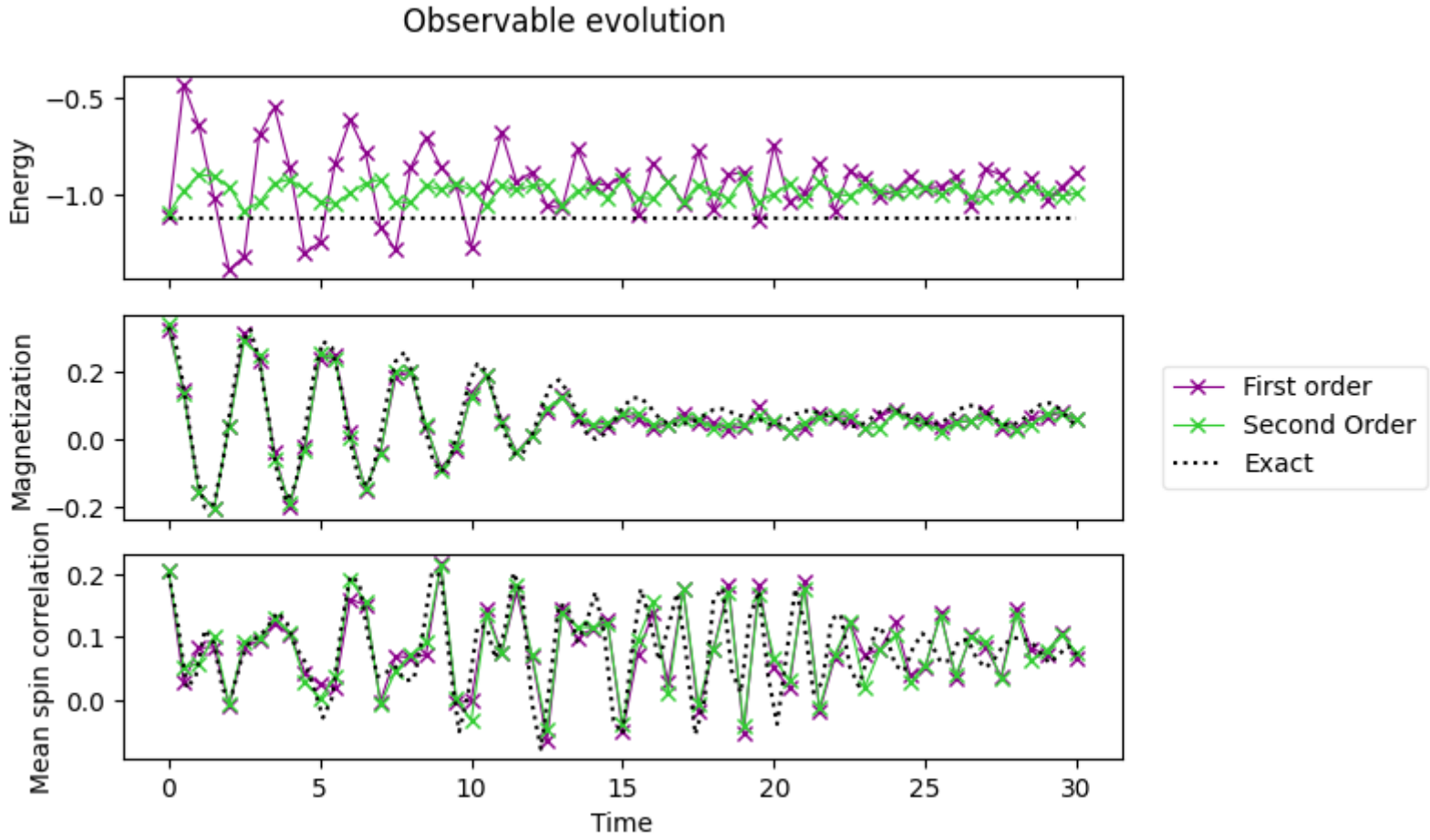
In [16]:

```
exact_times =  np.array([0.0, 0.3, 0.6, 0.8999999999999999, 1.2, 1.5, 1.7999999999999998, 2.1, 2.4, 2.69
exact_energy =  np.array([-1.1184402376762155, -1.1184402376762157, -1.1184402376762157, -1.11844023767(
exact_magnetization =  np.array([0.3333333333333333, 0.2631676963341 5005, 0.0912947227110664, -0.093177
exact_correlation =  np.array([0.2, 0.1247704225763532, 0.01943938494098705, 0.03854917181332821, 0.111
```

In [17]:

```
axes[0].plot(exact_times, exact_energy, c="k", ls=":", label="Exact")
axes[1].plot(exact_times, exact_magnetization, c="k", ls=":", label="Exact")
axes[2].plot(exact_times, exact_correlation, c="k", ls=":", label="Exact")
# Replace the legend
legend.remove()
# Select the labels of only the first axis
legend = fig.legend(
    *axes[0].get_legend_handles_labels(),
    bbox_to_anchor=(1.0, 0.5),
    loc="center left",
    framealpha=0.5,
)
fig.tight_layout()
fig
```

Out[17]:



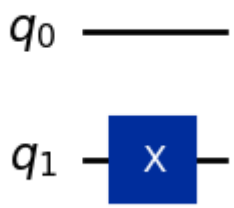
In [18]:

```
1 n_qubits_2 = 2
2 dt_2 = 1.6
3 product_formula = LieTrotter(reps=1)
```

In [19]:

```
# We prepare an initial state ↓↑ (10).
# Note that Statevector and SparsePauliOp interpret the qubits from right to left
initial_circuit_2 = QuantumCircuit(n_qubits_2)
initial_circuit_2.prepare_state('10')
# Change reps and see the difference when you decompose the circuit
initial_circuit_2.decompose(reps=1).draw("mpl")
```

Out[19]:



In [20]:

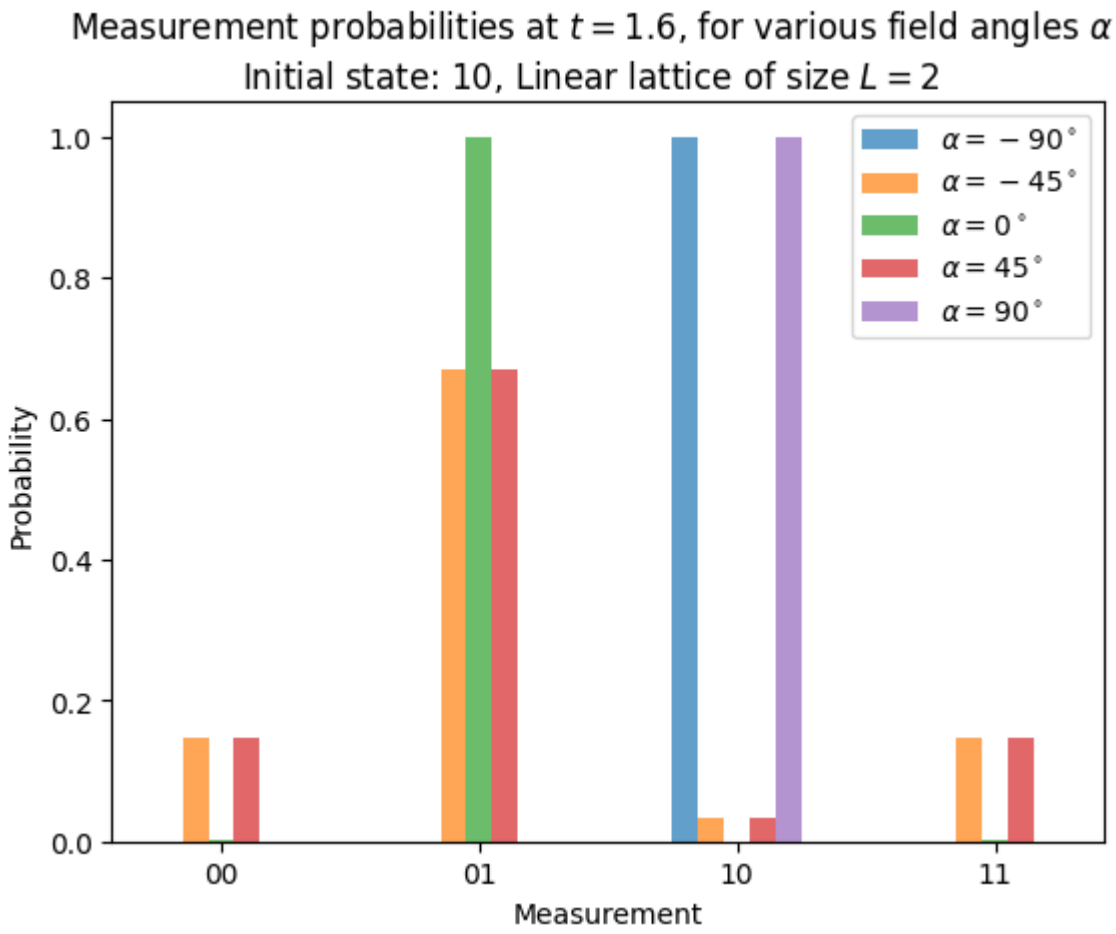
```
bar_width = 0.1
# initial_state = Statevector.from_label("10")
final_time = 1.6
eps = 1e-5

# We create the list of angles in radians, with a small epsilon
# the exactly longitudinal field, which would present no dynamics at all
alphas = np.linspace(-np.pi / 2 + eps, np.pi / 2 - eps, 5)

for i, alpha in enumerate(alphas):
    evolved_state_2 = QuantumCircuit(initial_circuit_2.num_qubits)
    evolved_state_2.append(initial_circuit_2, evolved_state_2.qubits)
    hamiltonian_2 = get_hamiltonian(nqubits=2, J=0.2, h=1.0, alpha=alpha)
    single_step_evolution_gates_2 = PauliEvolutionGate(
        hamiltonian_2, dt_2, synthesis=product_formula
    )
    evolved_state_2.append(single_step_evolution_gates_2, evolved_state_2.qubits)
    evolved_state_2 = Statevector(evolved_state_2)
    # Dictionary of probabilities
    amplitudes_dict = evolved_state_2.probabilities_dict()
    labels = list(amplitudes_dict.keys())
    values = list(amplitudes_dict.values())
    # Convert angle to degrees
    alpha_str = f"$\\alpha={int(np.round(alpha * 180 / np.pi))}^\\circ$"
    plt.bar(np.arange(4) + i * bar_width, values, bar_width, label=alpha_str, alpha=0.7)

plt.xticks(np.arange(4) + 2 * bar_width, labels)
plt.xlabel("Measurement")
plt.ylabel("Probability")
plt.suptitle(
    f"Measurement probabilities at $t={final_time}$, for various field angles $\\alpha$\n"
    f"Initial state: 10, Linear lattice of size $L=2$"
)
plt.legend()
```

Out[20]: <matplotlib.legend.Legend at 0x773954d632f0>


Measurement probabilities at $t = 1.6$, for various field angles α
Initial state: 10, Linear lattice of size $L = 2$
Probability
1.0
0.8
0.6
0.4
0.2
0.0
00
01
10
11
Measurement
$\alpha = -90^\circ$
$\alpha = -45^\circ$
$\alpha = 0^\circ$
$\alpha = 45^\circ$
$\alpha = 90^\circ$

In [21]:
```
circuit_list = []
for i, alpha in enumerate(alphas):
    evolved_state_2 = QuantumCircuit(initial_circuit_2.num_qubits)
    evolved_state_2.append(initial_circuit_2, evolved_state_2.qubits)
    hamiltonian_2 = get_hamiltonian(nqubits=2, J=0.2, h=1.0, alpha=alpha)
    single_step_evolution_gates_2 = PauliEvolutionGate(
        hamiltonian_2, dt_2, synthesis=product_formula
    )
    evolved_state_2.append(single_step_evolution_gates_2, evolved_state_2.qubits)
    evolved_state_2.measure_all()
    circuit_list.append(evolved_state_2)
```

In [23]:
```
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService(channel="ibm_cloud",
                               token="yqgL4G6pkrcOagKTg8rFD0ubWZtbuz0uCPNH52Qysdf0",
                               instance="firsthwinstance")
```

In [24]:
```
#from qiskit_ibm_runtime import QiskitRuntimeService
# Save account to disk and save it as the default.
QiskitRuntimeService.save_account(channel="ibm_cloud",
                                  token="yqgL4G6pkrcOagKTg8rFD0ubWZtbuz0uCPNH52Qysdf0",
                                  instance="firsthwinstance",
                                  name="Renzo Diomedi", overwrite=True)

# Load the saved credentials
service = QiskitRuntimeService(name="Renzo Diomedi")
```

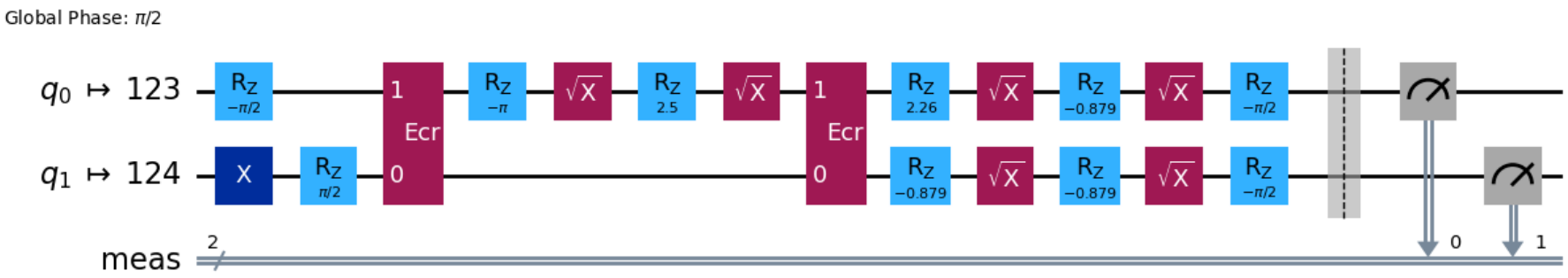
In [26]:
```
service = QiskitRuntimeService(channel="ibm_cloud")
backend = service.least_busy(operational=True, simulator=False)
backend.name
```

Out[26]: 'ibm_sherbrooke'

In [27]:
```
pm = generate_preset_pass_manager(backend=backend, optimization_level=3)
circuit_isa = pm.run(circuit_list)
```

In [28]:
```
circuit_isa[1].draw("mpl", idle_wires=False)
```

Out[28]:



In [29]:
```
sampler = SamplerV2(mode=backend)
job = sampler.run(circuit_isa)
print("job id:", job.job_id())
```

```
job id: d0j015csflic739neuo0
```

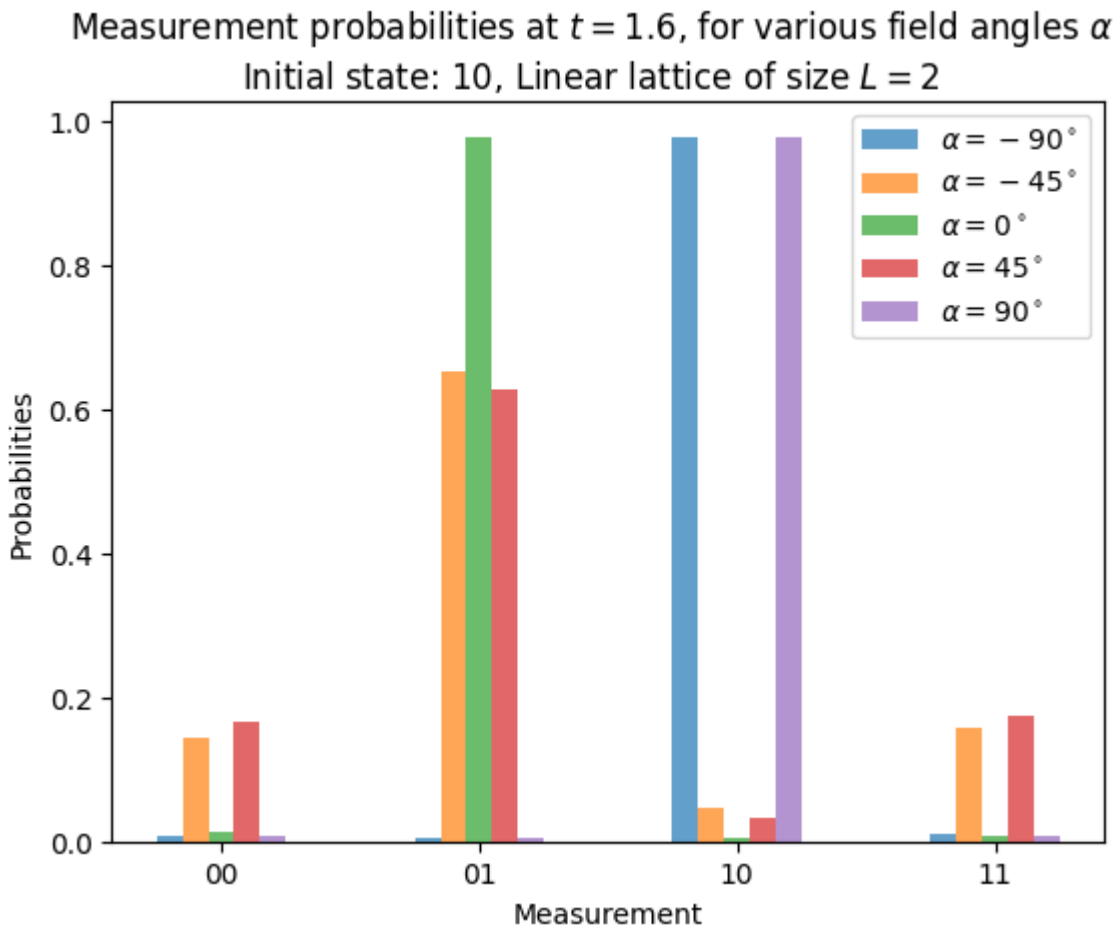
In [30]:
```
results = job.result()
```

In [31]:
```
list_temp = ['00', '01', '10', '11']

for i, alpha in enumerate(alphas):
    # Dictionary of probabilities
    amplitudes_dict = results[i].data.meas.get_counts()
    values = []
    for str_temp in list_temp:
        values.append(amplitudes_dict[str_temp]/4096.0) # devided by default number of shots
    # Convert angle to degrees
    alpha_str = f"$\\alpha={int(np.round(alpha * 180 / np.pi))}^\\circ$"
    plt.bar(np.arange(4) + i * bar_width, values, bar_width, label=alpha_str, alpha=0.7)

plt.xticks(np.arange(4) + 2 * bar_width, labels)
plt.xlabel("Measurement")
plt.ylabel("Probabilities")
plt.suptitle(
    f"Measurement probabilities at $t={final_time}$, for various field angles $\\alpha$\n"
    f"Initial state: 10, Linear lattice of size $L=2$"
)
plt.legend()
```

Out[31]:
```
<matplotlib.legend.Legend at 0x773954961d60>
```

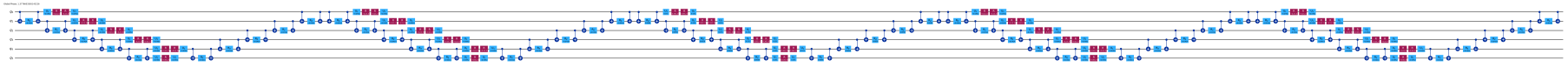

Measurement probabilities at $t = 1.6$, for various field angles α
Initial state: 10, Linear lattice of size $L = 2$
Probabilities
1.0
0.8
0.6
0.4
0.2
0.0
00
01
10
11
Measurement
$\alpha = -90^\circ$
$\alpha = -45^\circ$
$\alpha = 0^\circ$
$\alpha = 45^\circ$
$\alpha = 90^\circ$

In [32]:

```
# Modify the line below (Use PauliEvolutionGate)
single_step_evolution_gates_st4 = PauliEvolutionGate(
    hamiltonian, dt, synthesis=product_formula_st4
)
single_step_evolution_st4 = QuantumCircuit(n_qubits)
single_step_evolution_st4.append(single_step_evolution_gates_st4, single_step_evolution_st4.qubits)
# Let us print some stats
print(
    f"""
Trotter step with second-order Suzuki-Trotter
------------------------------
Depth: {single_step_evolution_st4.decompose(reps=3).depth()}
Gate count: {len(single_step_evolution_st4.decompose(reps=3))}
Nonlocal gate count: {single_step_evolution_st4.decompose(reps=3).num_nonlocal_gates()}
Gate breakdown: {", ".join([f"{k.upper()}: {v}" for k, v in single_step_evolution_st4.decompose(reps=3)
"""
)
single_step_evolution_st4.decompose(reps=2).draw("mpl", fold=-1)
```

```
Trotter step with second-order Suzuki-Trotter
------------------------------
Depth: 170
Gate count: 265
Nonlocal gate count: 100
Gate breakdown: U3: 115, CX: 100, U1: 50
```

Out[32]:



In [ ]:

```
1
```