QOSF Screening Task 3

We will be using qiskit.QuantumCircuit as our circuit object throughout the notebook.

0. Task Description

Please write a simple compiler – program, which translates one quantum circuit into another, using a restricted set of gates.

You need to consider just the basic gates for the input circuit, such as (I, H, X, Y, Z, RX, RY, RZ, CNOT, CZ).

The output circuit should consist only from the following gates: RX, RZ, CZ. In other words, each gate in the original circuit must be replaced by an equivalent combination of gates coming from the restricted set (RX, RZ, CZ) only.

For example, a Hadamard gate after compilation looks like this:

RZ(pi/2) RX(pi/2) RZ(pi/2)

Analyze what's the overhead of the compiled program compared to the original one and propose how to improve it. What we mean by overhead is the following: by replacing all the initial gates with the restricted set of gates given in the problem, you will see that the resulting circuit is much more involved than the original one. This is what we called the overhead, and you may think about how to treat this problem, i.e. you could try to simplify as much as possible the resulting circuit.

1. Gate Identities

Rotational operators are defined as:

$$R_x(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -isin(\frac{\theta}{2}) \\ -isin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$
$$R_y(\theta) = \begin{pmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix}$$
$$R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

It's easy to observe the following identities:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \propto R_x(\pi) \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \propto R_y(\pi) \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \propto R_z(\pi)$$

where $\propto$ denotes "proportional to", or "**equivalent up to global phase**".

With the famous commutation relations of X, Y and Z, we can also observe that:

$$\sigma_i\sigma_j \propto \sigma_j\sigma_i$$

where $\sigma_i, \sigma_j \in \{I, X, Y, Z\}$

We can also easily express S in terms of $R_z(\theta)$:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \propto R_z(\frac{\pi}{2})$$

Therefore, we have a way to express $R_y(\theta)$ in terms of $R_x(\theta)$ and $R_z(\theta)$ by using the Clifford property of S gate:

$$\begin{aligned} R_y(\theta) &= SR_x(\theta)S^\dagger \\ &\propto R_z(\frac{\pi}{2})R_x(\theta)R_z(-\frac{\pi}{2}) \end{aligned}$$

and also a way to express Y:

$$\begin{aligned} Y &\propto R_y(\pi) \\ &\propto R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2}) \end{aligned}$$

We can also express H in terms of $R_x(\theta)$ and $R_z(\theta)$:

$$\begin{aligned} H &= \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ &\propto R_y(\frac{\pi}{2})R_z(\pi) \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \end{aligned}$$

Therefore, transforming CX to CZ, $R_x(\theta)$ and $R_z(\theta)$ is also realized using the Clifford property of H gate:

$$\begin{aligned} CX_{0,1} &= H_1CZ_{0,1}H_1 \\ &\propto [R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1CZ_{0,1}[R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1 \\ &\propto [R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})]_1CZ_{0,1}[R_z(\pi)R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1 \end{aligned}$$

We can further observe X gate is equvalent to a 180° rotation around X-axis (or simple X gate) sandwiched by two 90° around Z-axis:

$$\begin{aligned} X &= HZH \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})ZR_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})Rx(\pi)R_z(\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})XR_z(\frac{\pi}{2}) \end{aligned}$$

By symmetry, we have the other 5 equations:

$$\begin{aligned} X &\propto R_y(\frac{\pi}{2})XR_y(\frac{\pi}{2}) \\ Y &\propto R_x(\frac{\pi}{2})YR_x(\frac{\pi}{2}) \\ Y &\propto R_z(\frac{\pi}{2})YR_z(\frac{\pi}{2}) \\ Z &\propto R_x(\frac{\pi}{2})ZR_x(\frac{\pi}{2}) \\ Z &\propto R_y(\frac{\pi}{2})ZR_y(\frac{\pi}{2}) \end{aligned}$$

Therefore, a new way with one less gate to express Y in terms of $R_x(\theta)$ and $R_z(\theta)$ is:

$$\begin{aligned} Y &\propto R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2}) \\ &\propto R_z(\frac{\pi}{2})R_z(\frac{\pi}{2})R_x(\pi)R_z(-\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &\propto R_z(\pi)R_x(\pi) \end{aligned}$$

Therefore, we have all the identities used for circuit rewriting. Next we will mention several identities that will be useful for circuit optimization.

The two very obvious ones are gate merging:

$$R_x(\theta_1)R_x(\theta_2) = R_x(\theta_1+\theta_2) \qquad R_z(\theta_1)R_z(\theta_2) = R_z(\theta_1+\theta_2)$$

Two of the equations mentioned before can also be used during circuit optimization. One is:

$$R_x(\pi)R_z(\pi) \propto R_z(\pi)R_x(\pi)$$

which can be used to change order and seek for gate merging. The other is:

$$\begin{aligned} R_z(\frac{\pi}{2})R_x(\pi)R_z(\frac{\pi}{2}) &\propto R_x(\pi) \\ R_x(\frac{\pi}{2})R_z(\pi)R_x(\frac{\pi}{2}) &\propto R_z(\pi) \end{aligned}$$

which can be used to directly reduce the number of gates.

Another identity can be found by sandwiching $R_z(\theta)$ with H gate and expand it into a series of $R_z(\theta)$ and $R_x(\theta)$:

$$\begin{aligned} R_x(\theta) &= HR_z(\theta)H \\ &\propto R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})R_z(\theta)R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ &= R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\theta+\pi)R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \end{aligned}$$

for wich we can do the same for $R_z(\theta)$ and reverse the order to get two very useful identities for circuit optimization:

$$\begin{aligned} R_z(\alpha)R_x(\frac{\pi}{2})R_z(\theta)R_x(\frac{\pi}{2})R_z(\beta) &\propto R_z(\alpha-\frac{\pi}{2})R_x(\theta-\pi)R_z(\beta-\frac{\pi}{2}) \\ R_x(\alpha)R_z(\frac{\pi}{2})R_x(\theta)R_z(\frac{\pi}{2})R_x(\beta) &\propto R_x(\alpha-\frac{\pi}{2})R_z(\theta-\pi)R_x(\beta-\frac{\pi}{2}) \end{aligned}$$

It's also worth mentioning that $R_z(\theta)$ can freely move across CZ on both control and target qubit:

$$\begin{aligned} CZ_{0,1}R_z(\theta)_0 &\propto R_z(\theta)_0CZ_{0,1} \\ CZ_{0,1}R_z(\theta)_1 &\propto R_z(\theta)_1CZ_{0,1} \end{aligned}$$

and that concludes all the gate identities we'll be using throught the notebook.

2. Initialization

This part of the notebook intializes packages and defines global variables.

In [1]:
```
#initialization
import math
import numpy as np
import qiskit
from qiskit import execute, Aer
from qiskit.circuit.library import IGate, XGate, YGate, ZGate, HGate, SGate, RXGate, RYGate, RZGate, CX
Gate, CZGate
from qiskit.quantum_info import Statevector
import random
from tabulate import tabulate

pi = np.pi

GATE_SET = {'i', 'h', 'x', 'y', 'z', 'rx', 'ry', 'rz', 'cx', 'cz'}
BASIS_GATE_SET = {'rx', 'rz', 'cz'}

I = IGate().to_matrix()
X = XGate().to_matrix()
Y = YGate().to_matrix()
Z = ZGate().to_matrix()
H = ZGate().to_matrix()
S = ZGate().to_matrix()
RX = lambda t:RXGate(t).to_matrix()
RY = lambda t:RYGate(t).to_matrix()
RZ = lambda t:RZGate(t).to_matrix()
CX = CXGate().to_matrix()
CZ = CZGate().to_matrix()
```

```
c:\python37\lib\site-packages\requests\__init__.py:91: RequestsDependencyWarning: urllib3 (1.25.10) o
r chardet (3.0.4) doesn't match a supported version!
  RequestsDependencyWarning)
```

3. Helper Functions

This part of the notebook defines several helper functions.

In [2]:
```
# Helper functions

def presice_matrix(gates):
    # return the matrix of a list of gates in the precision of machine epsilon
    if len(gates) == 0:
        return 1
    return gates[-1].to_matrix().dot(presice_matrix(gates[:-1]))

def remove_global_phase(mat):
    # return the matrix with its global phase removed
    # global phase = the phase of the first non-zero term in first row
    for i in range(len(mat[0])):
        if np.round(mat[0][i], 3) != 0:
            global_phase_angle = np.angle(mat[0][i])
            break
    global_phase = np.e**(global_phase_angle*1j)
    return mat/global_phase

def matrix(gates, precision=3):
    # return the matrix of a list of gates in certain precision
    if precision == None:
        return precise_matrix(gates)
    else:
        return np.round(presice_matrix(gates), precision)

def matrix_ng(gates, precision=3):
    # return the matrix of a list of gates without global phase removed in certain precision
    if len(gates) == 0:
        return 1
    mat = presice_matrix(gates)
    mat = remove_global_phase(mat)
    if precision == None:
        return mat
    else:
        return np.round(mat, precision)

def equal(gates1, gates2):
    # return if two lists of gates are equal
    return (matrix(gates1) == matrix(gates2)).all()

def equal_ng(gates1, gates2):
    # return if two lists of gates are equal up to global phase
    return (matrix_ng(gates1) == matrix_ng(gates2)).all()

def equal_mat(mat1, mat2):
    # return if two matrices are equal
    return (mat1 == mat2).all()

def equal_mat_ng(mat1, mat2):
    # return if two lists of gates are equal up to global phase
    return (remove_global_phase(mat1) == remove_global_phase(mat2)).all()

def commute(gates1, gates2):
    # return if two lists of gates commute
    return equal(gates1+gates2, gates2+gates1)

def commute_ng(gates1, gates2):
    # return if two matrices commute up to global phase, or AB = exp(i*theta) BA
    return equal_ng(gates1+gates2, gates2+gates1)

def commute_mat(mat1, mat2):
    # return if two matrices commute
    return equal_mat(mat1.dot(mat2), mat2.dot(mat1))

def commute_mat_ng(mat1, mat2):
    # return if two matrices commute up to global phase, or AB = exp(i*theta) BA
    return equal_mat_ng(mat1.dot(mat2), mat2.dot(mat1))

def gate_to_str(gate):
    # return info about a gate in str format
    return gate.name if len(gate.params)==0 else f'{gate.name}({np.round(gate.params[0]/pi, 3)}pi)'

def equal_circuit(qc1, qc2):
    # return if two circutis are equal
    backend_sim = Aer.get_backend('unitary_simulator')
    job_sim = execute([qc1, qc2], backend_sim)
    result_sim = job_sim.result()
    unitary1 = result_sim.get_unitary(qc1)
    unitary2 = result_sim.get_unitary(qc2)
    return np.allclose(unitary1, unitary2)

def equal_circuit_ng(qc1, qc2):
    # return if two circuits are equal up to global phase
    s1 = Statevector.from_instruction(qc1)
    s2 = Statevector.from_instruction(qc2)
    return s1.equiv(s2)

def equal_test(gates1, gates2):
    # print if two set of gates are equal or euqal up to global phase
    print(f'{[gate_to_str(g) for g in gates1]} vs {[gate_to_str(g) for g in gates2]}')
    print(f'equal: {equal(gates1, gates2)}')
    print(f'equal up to global phase: {equal_ng(gates1, gates2)}')
    print()

def commute_test(gates1, gates2):
    # print if two set of gates commute or commute up to global phase
    print(f'{[gate_to_str(g) for g in gates1]} vs {[gate_to_str(g) for g in gates2]}')
    print(f'commute: {commute(gates1, gates2)}')
    print(f'commute up to global phase: {commute_ng(gates1, gates2)}')
    print()
```

4. Identity Verification

We then use the helper functions defined above to verify some of the gate identies derived in the first section.

In [3]:
```
gates1 = [XGate()]
gates2 = [ZGate()]
commute_test(gates1, gates2)

gates1 = [YGate()]
gates2 = [XGate(), ZGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(-pi/2), RXGate(pi), RZGate(pi/2)]
gates2 = [XGate(), ZGate()]
equal_test(gates1, gates2)

gates1 = [XGate()]
gates2 = [RXGate(pi)]
equal_test(gates1, gates2)

gates1 = [HGate()]
gates2 = [RZGate(pi), RYGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [HGate()]
gates2 = [RZGate(pi/2), RXGate(pi/2), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RYGate(1.2345)]
gates2 = [RZGate(-pi/2), RXGate(1.2345), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RXGate(5.4338)]
gates2 = [RXGate(2.2922), RXGate(pi)]
equal_test(gates1, gates2)

gates1 = [RXGate(1.2345)]
gates2 = [RZGate(pi/2), RXGate(pi/2), RZGate(1.2345+pi), RXGate(pi/2), RZGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RZGate(1.2345)]
gates2 = [RXGate(pi/2), RZGate(pi/2), RXGate(1.2345+pi), RZGate(pi/2), RXGate(pi/2)]
equal_test(gates1, gates2)

gates1 = [RYGate(pi/2), XGate(), RYGate(pi/2)]
gates2 = [XGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(pi/2), XGate(), RZGate(pi/2)]
gates2 = [XGate()]
equal_test(gates1, gates2)

gates1 = [RXGate(pi/2), YGate(), RXGate(pi/2)]
gates2 = [YGate()]
equal_test(gates1, gates2)

gates1 = [RZGate(pi/2), YGate(), RZGate(pi/2)]
gates2 = [YGate()]
equal_test(gates1, gates2)

gates1 = [RXGate(pi/2), ZGate(), RXGate(pi/2)]
gates2 = [ZGate()]
equal_test(gates1, gates2)

gates1 = [RYGate(pi/2), RZGate(pi), RYGate(pi/2)]
gates2 = [RZGate(pi)]
equal_test(gates1, gates2)

print(commute_mat_ng(np.kron(I, RZ(5.4321)), CZ))
print(commute_mat_ng(np.kron(RZ(5.4321), I), CZ))
```

```
['x'] vs ['z']
commute: False
commute up to global phase: True

['y'] vs ['x', 'z']
equal: False
equal up to global phase: True

['rz(-0.5pi)', 'rx(1.0pi)', 'rz(0.5pi)'] vs ['x', 'z']
equal: False
equal up to global phase: True

['x'] vs ['rx(1.0pi)']
equal: False
equal up to global phase: True

['h'] vs ['rz(1.0pi)', 'ry(0.5pi)']
equal: False
equal up to global phase: True

['h'] vs ['rz(0.5pi)', 'rx(0.5pi)', 'rz(0.5pi)']
equal: False
equal up to global phase: True

['ry(0.393pi)'] vs ['rz(-0.5pi)', 'rx(0.393pi)', 'rz(0.5pi)']
equal: True
equal up to global phase: True

['rx(1.73pi)'] vs ['rx(0.73pi)', 'rx(1.0pi)']
equal: True
equal up to global phase: True

['rx(0.393pi)'] vs ['rz(0.5pi)', 'rx(0.5pi)', 'rz(1.393pi)', 'rx(0.5pi)', 'rz(0.5pi)']
equal: False
equal up to global phase: True

['rz(0.393pi)'] vs ['rx(0.5pi)', 'rz(0.5pi)', 'rx(1.393pi)', 'rz(0.5pi)', 'rx(0.5pi)']
equal: False
equal up to global phase: True

['ry(0.5pi)', 'x', 'ry(0.5pi)'] vs ['x']
equal: True
equal up to global phase: True

['rz(0.5pi)', 'x', 'rz(0.5pi)'] vs ['x']
equal: True
equal up to global phase: True

['rx(0.5pi)', 'y', 'rx(0.5pi)'] vs ['y']
equal: True
equal up to global phase: True

['rz(0.5pi)', 'y', 'rz(0.5pi)'] vs ['y']
equal: True
equal up to global phase: True

['rx(0.5pi)', 'z', 'rx(0.5pi)'] vs ['z']
equal: True
equal up to global phase: True

['ry(0.5pi)', 'rz(1.0pi)', 'ry(0.5pi)'] vs ['rz(1.0pi)']
equal: True
equal up to global phase: True

True
True
```

5. Random Circuit Generator

This function can generate a random circuit given number of qubits, number of gates and a restricted set of gates.

In [4]:
```
def random_qc(num_q, num_g, gate_set):
    # create a random circuit with num_q qubits, num_g gates, and a restricted set of gates gate_set
    qc = qiskit.QuantumCircuit(num_q)
    for _ in range(num_g):
        g_name = random.choice(list(gate_set))
        if g_name[0] == 'c':
            c_index, t_index = random.sample(range(num_q), 2)
            getattr(qc, g_name)(c_index, t_index)
        elif g_name[0] == 'r':
            index = random.randint(0, num_q-1)
            param = random.uniform(-2*pi, 2*pi)
            getattr(qc, g_name)(param, index)
        else:
            index = random.randint(0, num_q-1)
            getattr(qc, g_name)(index)
    return qc
```

6. Circuit Rewriting

We rewrite the circuit by replacing each of the gate with their expressions in $\{R_x, R_z, CZ\}$. A complete set of rewriting rules:

$$\begin{aligned} I &\rightarrow Wire \\ H &\rightarrow R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})R_z(\frac{\pi}{2}) \\ X &\rightarrow R_x(\pi) \\ Y &\rightarrow R_x(\pi)R_z(\pi) \\ Z &\rightarrow R_z(\pi) \\ R_x(\theta) &\rightarrow R_x(\theta) \\ R_y(\theta) &\rightarrow R_z(\frac{\pi}{2})R_x(\theta)R_z(-\frac{\pi}{2}) \\ R_z(\theta) &\rightarrow R_z(\theta) \\ CX_{0,1} &\rightarrow [R_z(\frac{\pi}{2})R_x(\frac{\pi}{2})]_1CZ_{0,1}[R_z(\pi)R_x(\frac{\pi}{2})R_z(\frac{\pi}{2})]_1 \\ CZ_{0,1} &\rightarrow CZ_{0,1} \end{aligned}$$

In [5]:
```
def rewrite(qc):
    # rewrite the circuit with {Rx, Rz, CZ} only
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    for gate_info in qc.data:
        gate = gate_info[0]
        g_name = gate.name
        index_list = gate_info[1]
        if len(index_list) == 1:
            index = index_list[0]
            if g_name == 'i':
                pass
            elif g_name == 'h':
                output_qc.rz(pi/2, index)
                output_qc.rx(pi/2, index)
                output_qc.rz(pi/2, index)
            elif g_name == 'x':
                output_qc.rx(pi, index)
            elif g_name == 'y':
                output_qc.rx(pi, index)
                output_qc.rz(pi, index)
            elif g_name == 'z':
                output_qc.rz(pi, index)
            elif g_name == 'rx':
                output_qc.rx(gate.params[0], index)
            elif g_name == 'ry':
                output_qc.rz(-pi/2, index)
                output_qc.rx(gate.params[0], index)
                output_qc.rz(pi/2, index)
            elif g_name == 'rz':
                output_qc.rz(gate.params[0], index)
        else:
            c_index, t_index = index_list
            if g_name == 'cx':
                output_qc.rz(pi/2, t_index)
                output_qc.rx(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
                output_qc.cz(c_index, t_index)
                output_qc.rx(pi/2, t_index)
                output_qc.rz(pi/2, t_index)
            elif g_name == 'cz':
                output_qc.cz(c_index, t_index)
    return output_qc
```

7. Circuit Optimization (Solving the Overhead Problem)

In this section, we will focus on circuit optimization. **Our main goal is to reduce both the number of gates and circuit depth, and we will focus on reducing the number of one-qubit gates**.

I. Gate Cleaning

The first way of optimization is to remove gates with parameter values as mutiples of 2π; also ensures angles are in the range $[-\pi, \pi]$. Examples:

$$R_x(4\pi) \rightarrow I \qquad R_z(3\pi) \rightarrow R_z(\pi)$$

The cleaning process will not take place individually, but during and after other processes of optimization.

In [6]:
```
def clean_gates(qc):
    # remove all Rx and Rz gates with parameter as mutiples of 2pi
    # ensure parameters in the range (-pi, pi)
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    num_gates = len(qc.data)
    for i in range(num_gates):
        gate_info_i = qc.data[i]
        index_list_i = gate_info_i[1]
        gate_i = gate_info_i[0]
        g_name_i = gate_i.name

        if g_name_i in {'rz', 'rx'}:
            param = gate_i.params[0]
            if math.remainder(param, 2*pi) == 0:
                pass
            elif param > pi or param < -pi:
                getattr(output_qc, g_name_i)(math.remainder(param, 2*pi), index_list_i[0])
            else:
                getattr(output_qc, g_name_i)(param, index_list_i[0])
        elif g_name_i == 'cz':
            getattr(output_qc, g_name_i)(*index_list_i)

    return output_qc
```

II. Gate Merging

The merging process will be the main way to reduce the number of gates

$$R_x(\theta_1)R_x(\theta_2) = R_x(\theta_1+\theta_2) \qquad R_z(\theta_1)R_z(\theta_2) = R_z(\theta_1+\theta_2)$$

The code also takes into account the commutation property of $R_z(\theta)$ and CZ:

$$\begin{aligned} CZ_{0,1}R_z(\theta)_0 &\propto R_z(\theta)_0CZ_{0,1} \\ CZ_{0,1}R_z(\theta)_1 &\propto R_z(\theta)_1CZ_{0,1} \end{aligned}$$

which is very useful to combine two $R_z(\theta)$ gates across CZ.

We will also remove consecutive pairs of CZ gates. Under computational basis CZ gate has no distinction between control and target qubits, and its conjugate transpose is itself, so adjacent pairs can be directly removed:

$$CZ_{0,1} = CZ_{1,0} \qquad CZ_{0,1}CZ_{1,0} = I$$

The code breaks down into two parts: **merge_once** and **merge_all**. **merge_once** searches through the circuit and merge every pair it finds, while **merge_all** continues merging gates untill reaching the optimal circuit.

In [7]:
```
def merge_once(qc):
    # Merge adjacent gates with same type once.
    output_qc = qiskit.QuantumCircuit(qc.num_qubits)
    num_gates = len(qc.data)
    check_list = []
    for i in range(num_gates):
        if i in check_list:
            continue

        gate_info_i = qc.data[i]
        index_list_i = gate_info_i[1]
        gate_i = gate_info_i[0]
        g_name_i = gate_i.name

        if g_name_i in {'rz', 'rx'}:
            index_i = index_list_i[0]

            for j in range(i+1, num_gates):

                gate_info_j = qc.data[j]
                index_list_j = gate_info_j[1]
                gate_j = gate_info_j[0]
                g_name_j = gate_j.name

                if index_i not in index_list_j:
                    continue

                if g_name_j == 'cz':
                    if g_name_i == 'rz':
                        pass
                    elif g_name_i == 'rx':
                        getattr(output_qc, g_name_i)(gate_i.params[0], index_i)
                        break
                elif g_name_j == g_name_i:
                    param_sum = gate_i.params[0]+gate_j.params[0]
                    getattr(output_qc, g_name_i)(param_sum, index_i)
                    check_list.append(j)
                    break
                else:
                    getattr(output_qc, g_name_i)(gate_i.params[0], index_i)
                    break
            else:
                getattr(output_qc, g_name_i)(gate_i.params[0], index_i)

        elif g_name_i == 'cz':
            for j in range(i+1, num_gates):

                gate_info_j = qc.data[j]
                index_list_j = gate_info_j[1]
                gate_j = gate_info_j[0]
                g_name_j = gate_j.name

                set_i = set(index_list_i)
                set_j = set(index_list_j)

                if set_j.intersection(set_i) == set():
                    continue
                if set_i == set_j:
                    check_list.append(j)
                    break
                elif g_name_j == 'rz':
                    pass
                else:
                    getattr(output_qc, g_name_i)(*index_list_i)
                    break
            else:
                getattr(output_qc, g_name_i)(*index_list_i)
    return output_qc

def merge_all(qc):
    # repeart the merging process untill the circuit is optimal
    old_qc = qc
    new_qc = clean_gates(merge_once(old_qc))
    while len(new_qc.data) < len(old_qc.data):
        old_qc = new_qc
        new_qc = clean_gates(merge_once(old_qc))
    return new_qc
```

III. Swapping $R_x(\pi)$ and $R_z(\pi)$

This part uses the property:

$$R_x(\pi)R_z(\pi) \propto R_z(\pi)R_x(\pi)$$

and tries to switch such pairs found in circuit. Each time it swaps it will merge the gates again, and see if swapping allow more gates to be merged together.

The code splits into two parts as well. **swap_once** swaps one pair each time, and **swap_and_merge_all** is a combination of swapping and merging.

In [8]:
```
def swap_once(qc, check_list=[]):
    # swap the position of adjacent Rx(pi) and Rz(pi) once
    output_qc = qc.copy()
    num_gates = len(qc.data)
    checked = None
    for i in range(num_gates):
        if i in check_list:
            continue
        gate_info_i = output_qc.data[i]
        index_list_i = gate_info_i[1]
        gate_i = gate_info_i[0]
        g_name_i = gate_i.name

        if g_name_i not in {'rz', 'rx'}:
            continue

        if gate_i.params[0] == pi:
            index_i = index_list_i[0]
            for j in range(i+1, num_gates):

                gate_info_j = output_qc.data[j]
                index_list_j = gate_info_j[1]
                gate_j = gate_info_j[0]
                g_name_j = gate_j.name

                if index_i in index_list_j:
                    if len(index_list_j)==1 and g_name_i != g_name_j:
                        if gate_j.params[0] == pi:
                            output_qc.data[i], output_qc.data[j] = output_qc.data[j], output_qc.data[i]
                            checked = i
                    break
            if checked != None:
                break

    return output_qc, checked

def swap_and_merge_all(qc):
    # try if the swap makes further optimization possible. Merge all if possible.
    current_qc = merge_all(qc)
    re_qc, checked = swap_once(current_qc)
    check_list = []
    while checked != None:
        mer_re_qc = merge_all(re_qc)
        if len(mer_re_qc.data) < len(current_qc.data):
            current_qc = mer_re_qc
            re_qc, checked = swap_once(current_qc)
        else:
            check_list.append(checked)
            re_qc, checked = swap_once(current_qc, check_list)

    return current_qc
```

IV. Group Replacement

This part finds particular groups of gates appeared in circuit and use two sets of identities to further optimize the circuit. The first is:

$$\begin{aligned} R_z(\frac{\pi}{2})R_x(\pi)R_z(\frac{\pi}{2}) &\propto R_x(\pi) \\ R_x(\frac{\pi}{2})R_z(\pi)R_x(\frac{\pi}{2}) &\propto R_z(\pi) \end{aligned}$$

and the second is:

$$\begin{aligned} R_z(\alpha)R_x(\frac{\pi}{2})R_z(\theta)R_x(\frac{\pi}{2})R_z(\beta) &\propto R_z(\alpha-\frac{\pi}{2})R_x(\theta-\pi)R_z(\beta-\frac{\pi}{2}) \\ R_x(\alpha)R_z(\frac{\pi}{2})R_x(\theta)R_z(\frac{\pi}{2})R_x(\beta) &\propto R_x(\alpha-\frac{\pi}{2})R_z(\theta-\pi)R_x(\beta-\frac{\pi}{2}) \end{aligned}$$

The funcion **find_next_n_gates** is a helper function in this section, which finds the next n gates after a given position in a circuit. The main function **group_and_merge_all** find the 4 particular groups from the above two equations and replace them by their simplified form. The extra 3 cases come from the fact that $R_z(\alpha)$ commutes with CZ. For example, consider the group:

$$R_z(\alpha)_1CZ_{0,1}[R_x(\frac{\pi}{2})R_z(\theta)R_x(\frac{\pi}{2})R_z(\beta)]_1$$

which can be optimized to:

$$R_z(\alpha-\frac{\pi}{2})_1CZ_{0,1}[R_x(\theta-\pi)R_z(\beta-\frac{\pi}{2})]_1$$

so we need to take into account how CZ can be inserted in between the group of gates.

In [9]:
```python
def find_next_n_gates(qc, i, n):
    # find the next n gates after a given gate in a circuit
    gate_infos = qc.data
    gate_list = []
    for j in range(i+1, len(gate_infos)):
        set_i = set(gate_infos[i][1])
        set_j = set(gate_infos[j][1])
        if set_j.intersection(set_i) != set():
            gate_list.append(j)
        if len(gate_list) == n:
            return gate_list

    while len(gate_list) != n:
        gate_list.append(None)
    return gate_list

# 4 lists of replacing rules as tuples
# name_tuple_list: search for matched gate names
# param_tuple_list: search for matched parameters
# r_name_tuple_list: replace macthed gates
# r_param_tuple_list: index or parameter replacing rules
#                     number: inherent index_list: index of param_tuple
#                     tuple: inherent parameter: (index of param_tuple, shift)

name_tuple_list = [('rx', 'rz', 'rx'),
                   ('rz', 'rx', 'rz'),
                   ('rx', 'rz', 'rx', 'rz', 'rx'),
                   ('rz', 'rx', 'rz', 'rx', 'rz'),
                   ('rz', 'cz', 'rx', 'rz', 'rx', 'rz'),
                   ('rz', 'rx', 'rz', 'rx', 'cz', 'rz'),
                   ('rz', 'cz', 'rx', 'rz', 'rx', 'cz', 'rz')]

param_tuple_list = [(pi/2, pi, pi/2),
                    (pi/2, pi, pi/2),
                    (None, pi/2, None, pi/2, None),
                    (None, pi/2, None, pi/2, None),
                    (None, None, pi/2, None, pi/2, None),
                    (None, pi/2, None, pi/2, None, None),
                    (None, None, pi/2, None, pi/2, None, None)]

r_name_tuple_list = [('rz',),
                     ('rx',),
                     ('rx', 'rz', 'rx'),
                     ('rz', 'rx', 'rz'),
                     ('cz', 'rz', 'rx', 'rz'),
                     ('rz', 'rx', 'rz', 'cz'),
                     ('cz', 'rz', 'rx', 'rz', 'cz')]



r_param_tuple_list = [((1, 0),),
                      ((1, 0),),
                      ((0, -pi/2), (2, -pi), (4, -pi/2)),
                      ((0, -pi/2), (2, -pi), (4, -pi/2)),
                      (1, (0, -pi/2), (3, -pi), (5, -pi/2)),
                      ((0, -pi/2), (2, -pi), (5, -pi/2), 4),
                      (1, (0, -pi/2), (3, -pi), (6, -pi/2), 5)]

def group_and_merge_all(qc):
    # find macthed group of gates and replace according to the lists defined above

    for r_index in range(len(name_tuple_list)):

        check_set = set()
        output_qc = qiskit.QuantumCircuit(qc.num_qubits)
        num_gates = len(qc.data)

        name_tuple = name_tuple_list[r_index]
        param_tuple = param_tuple_list[r_index]

        r_name_tuple = r_name_tuple_list[r_index]
        r_param_tuple = r_param_tuple_list[r_index]

        for i in range(num_gates):

            if i in check_set:
                continue

            pos_list = [i]+find_next_n_gates(qc, i, len(name_tuple)-1)

            if None in pos_list:
                pass
            else:

                i_name_tuple = tuple(qc.data[p][0].name for p in pos_list)
                i_param_tuple = tuple(None if (param_tuple[j] == None or len(qc.data[pos_list[j]][0].pa
rams) == 0) \
                                      else qc.data[pos_list[j]][0].params[0] for j in range(len(param_t
uple)))

                if i_name_tuple == name_tuple and i_param_tuple == param_tuple:
                    for j in range(len(r_name_tuple)):
                        n = r_name_tuple[j]
                        p = r_param_tuple[j]
                        if type(p) == int and n == 'cz':
                            pos = pos_list[p]
                            for k in range(pos_list[p-1]+1, pos):
                                getattr(output_qc, qc.data[k][0].name)(*qc.data[k][0].params, *qc.data[
k][1])
                                check_set.add(k)
                            getattr(output_qc, 'cz')(*qc.data[pos][1])
                        elif type(p) == tuple:
                            getattr(output_qc, n)(qc.data[pos_list[p[0]]][0].params[0]+p[1], qc.data[i]
[1][0])
                    check_set = check_set.union(set(pos_list))
                    continue

            gate_info = qc.data[i]
            gate = gate_info[0]
            gate_name = gate.name
            index_list = gate_info[1]

            if gate_name in {'rx', 'rz'}:
                getattr(output_qc, gate_name)(gate.params[0], *index_list)
            elif gate_name == 'cz':
                getattr(output_qc, gate_name)(*index_list)

        qc = merge_all(output_qc)

    return output_qc
```

8. Transpilation and Optimization Tests

In this section we will test our transpilation and optimization processes.

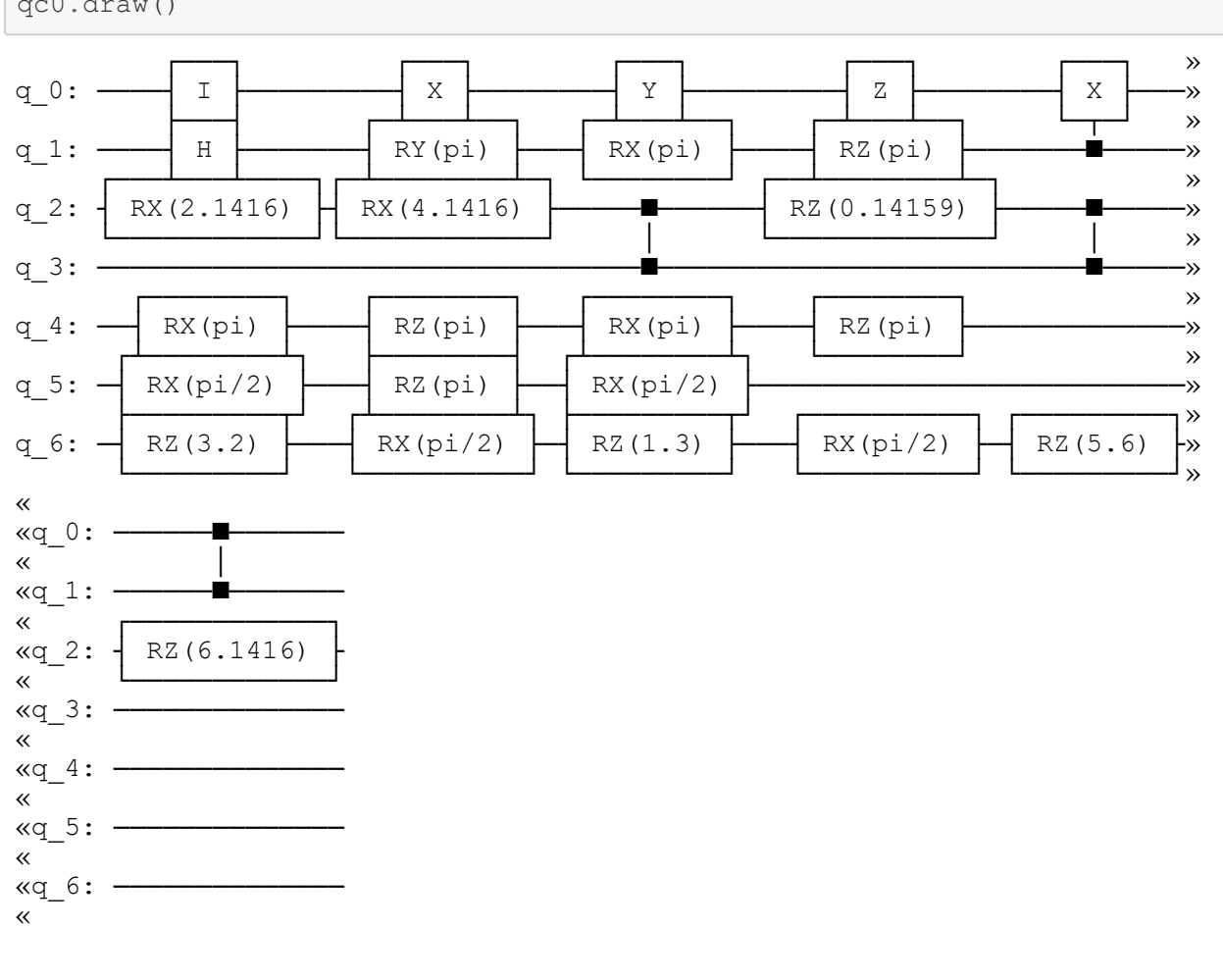
We want to first define two tests: one tests if transpiled circuits are equal to intial circuit up to global phase and have the correct set of gate, one tests how well our optimization processes do.

In [10]:
```python
def transpilation_results(qc_list, process_list):
    # compare the equality and gate set of a list of circuits

    gate_set_list = [set(key for key in qc.count_ops()) for qc in qc_list]
    equality_list = [equal_circuit_ng(qc_list[0], qc) for qc in qc_list]

    print(tabulate(zip(process_list, equality_list, gate_set_list), \
                   headers=['Process', 'Equal Up to GP', 'Gate Set'],
                   tablefmt='github'))

def optimization_results(qc_list, process_list):
    # compare the depth and number of gates of a list of circuits

    depth_list = []
    num_gates_list = []

    for qc in qc_list:
        depth_list.append(qc.depth())
        num_gates_list.append(len(qc.data))

    print(tabulate(zip(process_list, depth_list, num_gates_list), \
                   headers=['Process', 'Circuit Depth', 'Number of Gates'],
                   tablefmt='github',
                   numalign='left'))
```

We then define a circuit with each qubit testing for a specific feature. Specifically, **qubit_0** and **qubit_1** are for rewriting test, **qubit_2** and **qubit_3** are for merging test, **qubit_4** is for swapping test, **qubit_5** and **qubit_6** are for group replacement test.

In [11]:
```python
qc0 = qiskit.QuantumCircuit(7)

# for rewriting test
qc0.i(0)
qc0.x(0)
qc0.y(0)
qc0.z(0)
qc0.h(1)
qc0.ry(pi, 1)
qc0.rx(pi, 1)
qc0.rz(pi, 1)
qc0.cx(1, 0)
qc0.cz(0, 1)

# for merging test
qc0.rx(pi-1, 2)
qc0.rx(pi+1, 2)
qc0.cz(3, 2)
qc0.rz(pi-3, 2)
qc0.cz(2, 3)
qc0.rz(pi+3, 2)

# for swapping test
qc0.rx(pi, 4)
qc0.rz(pi, 4)
qc0.rx(pi, 4)
qc0.rz(pi, 4)

# for group replacement test
qc0.rx(pi/2, 5)
qc0.rz(pi, 5)
qc0.rx(pi/2, 5)

qc0.rz(3.2, 6)
qc0.rx(pi/2, 6)
qc0.rz(1.3, 6)
qc0.rx(pi/2, 6)
qc0.rz(5.6, 6)

qc0.draw()
```

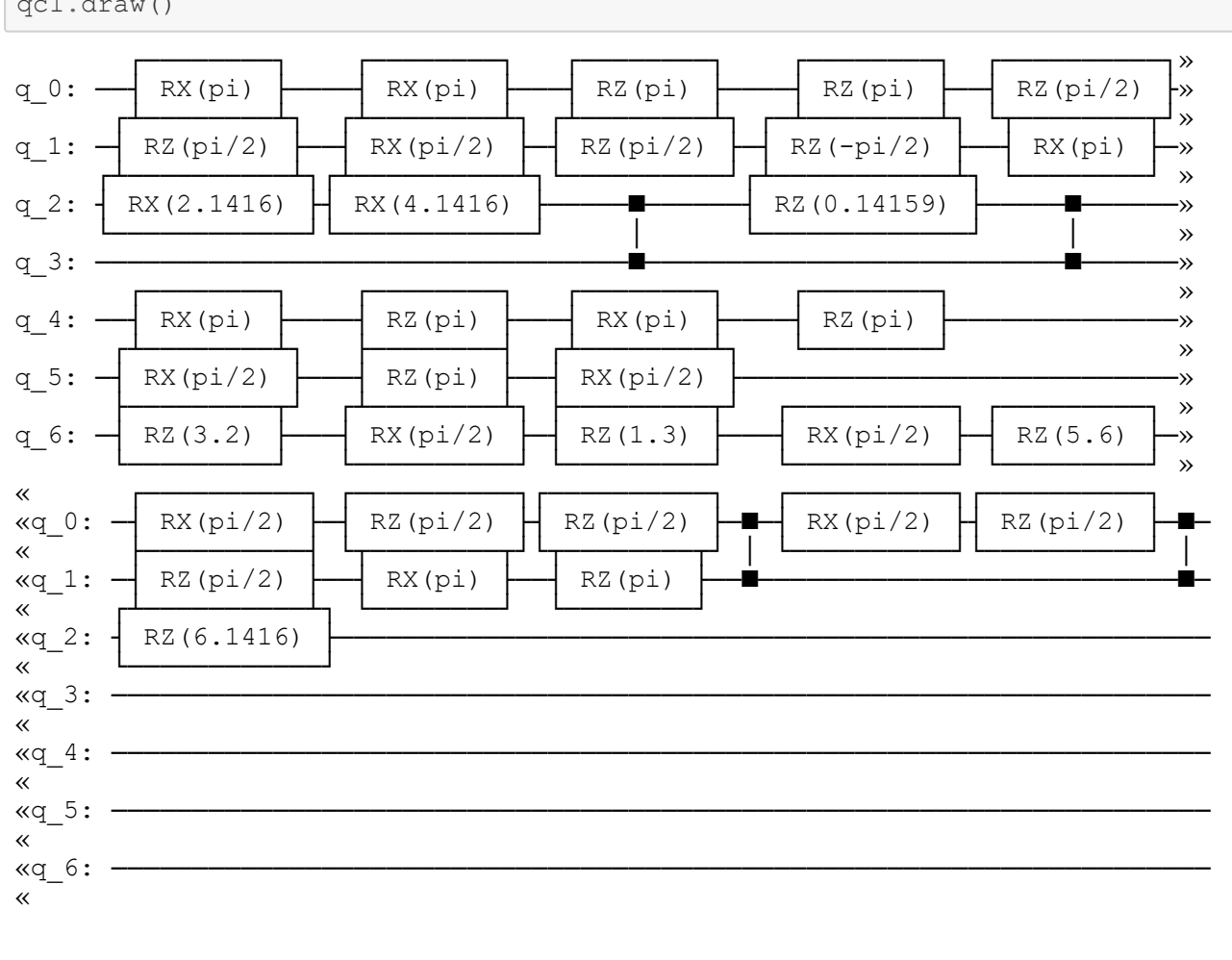
Out[11]:



After rewriting, gates on **qubit_0** and **qubit_1** should be fully replaced by $\{R_x, R_z, CZ\}$:

In [12]:
```python
qc1 = rewrite(qc0)
qc1.draw()
```

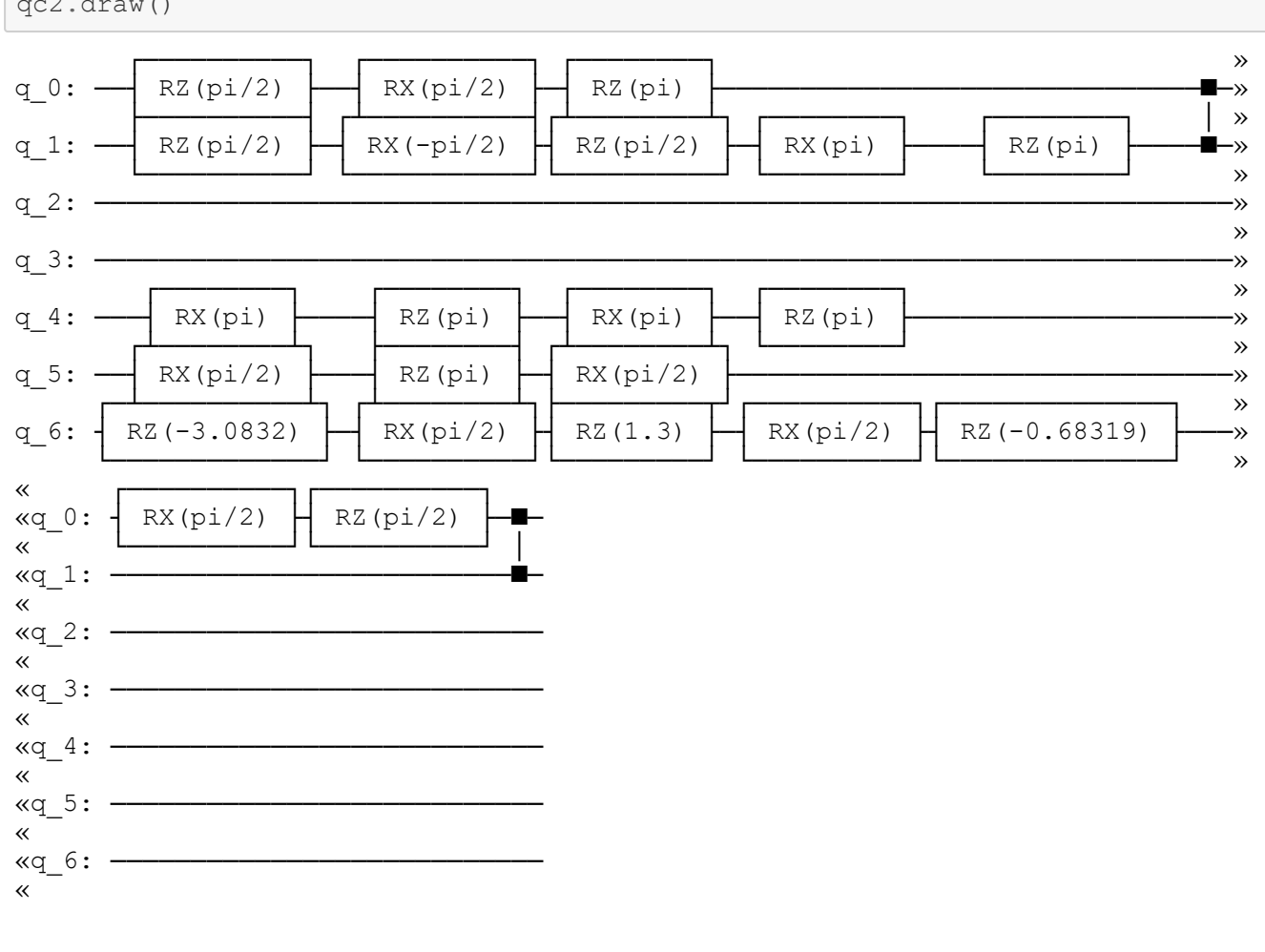
Out[12]:



After merging, gates on **qubit_2** and **qubit_3** should disapper:

In [13]:
```python
qc2 = merge_all(qc1)
qc2.draw()
```

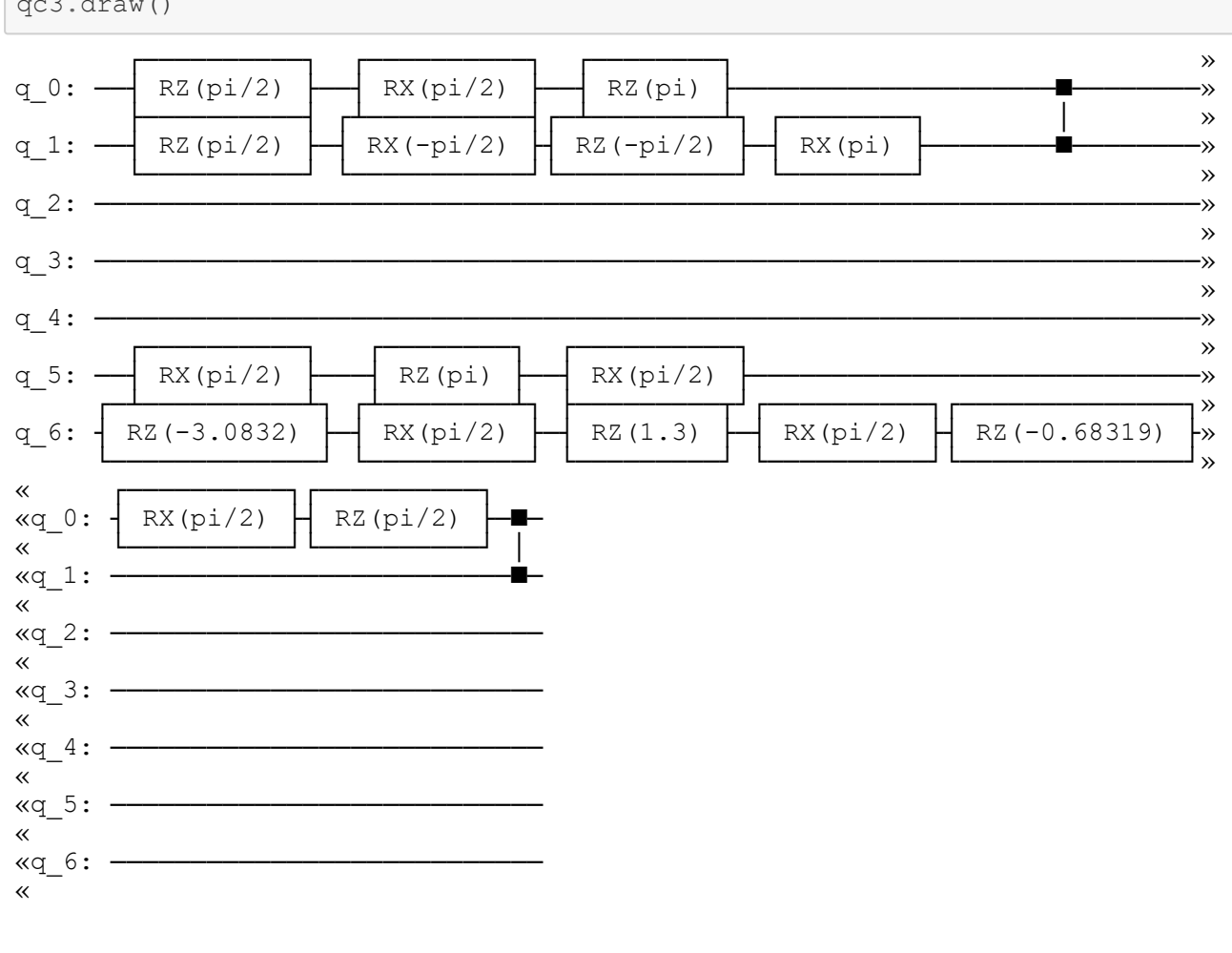
Out[13]:



After swapping, gates on **qubit_4** should disappear:

In [14]:
```python
qc3 = swap_and_merge_all(qc2)
qc3.draw()
```

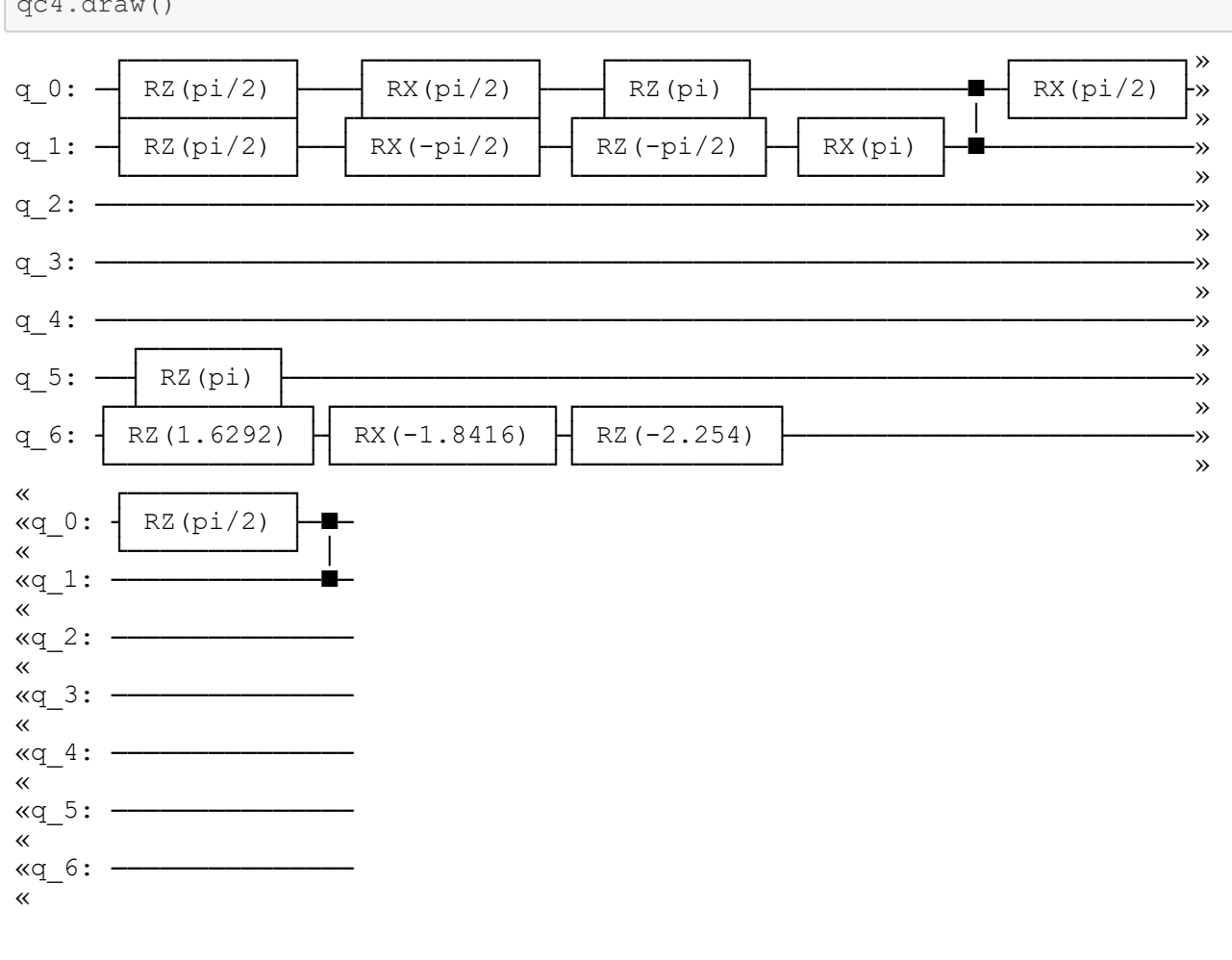
Out[14]:



After group replacement, gates on **qubit_5** and **qubit_6** shoud have number reduced:

In [15]:
```python
qc4 = group_and_merge_all(qc3)
qc4.draw()
```

Out[15]:



The circuits above should be equivalent up to global phase, and the gate set used in the last 4 circuits should be $\{R_x, R_z, CZ\}$:

In [16]:
```python
transpilation_results([qc0, qc1, qc2, qc3, qc4], \
                      ['Initial Circuit', 'Circuit Rewriting', 'Gate Merging', 'Swapping', 'Group Repla
cement'])
```

```
| Process           | Equal Up to GP   | Gate Set                                                 |
|-------------------|------------------|----------------------------------------------------------|
| Initial Circuit   | True             | {'id', 'cz', 'h', 'cx', 'rz', 'ry', 'rx', 'z', 'y', 'x'} |
| Circuit Rewriting | True             | {'cz', 'rx', 'rz'}                                       |
| Gate Merging      | True             | {'cz', 'rx', 'rz'}                                       |
| Swapping          | True             | {'cz', 'rx', 'rz'}                                       |
| Group Replacement | True             | {'cz', 'rx', 'rz'}                                       |
```

The circuit depth and number of gates should also be reduced compared to the circuit after rewrting:

In [17]:
```python
optimization_results([qc0, qc1, qc2, qc3, qc4], \
                     ['Initial Circuit', 'Circuit Rewriting', 'Gate Merging', 'Swapping', 'Group Replac
ement'])
```

```
| Process           | Circuit Depth   | Number of Gates   |
|-------------------|-----------------|-------------------|
| Initial Circuit   | 6               | 28                |
| Circuit Rewriting | 12              | 38                |
| Gate Merging      | 9               | 24                |
| Swapping          | 8               | 19                |
| Group Replacement | 8               | 15                |
```

9. Stress Test

In this last section we will be stress testing our program. We will feed a huge random circuit with 1000 gates, and see how our program performs on such huge circuit. We'll verify that the final transpiled and optimized circuit is equivalent to the inital circuit.

In [18]:
```python
qc0 = random_qc(3, 1000, GATE_SET)
#qc0.draw()
```

In [19]:
```python
qc1 = rewrite(qc0)
#qc1.draw()
```

In [20]:
```python
qc2 = merge_all(qc1)
#qc2.draw()
```

In [21]:
```python
qc3 = swap_and_merge_all(qc2)
#qc3.draw()
```

In [22]:
```python
qc4 = group_and_merge_all(qc3)
#qc4.draw()
```

In [23]:
```python
transpilation_results([qc0, qc1, qc2, qc3, qc4], \
                      ['Initial Circuit', 'Circuit Rewriting', 'Gate Merging', 'Swapping', 'Group Repla
cement'])
```

```
| Process           | Equal Up to GP   | Gate Set                                                 |
|-------------------|------------------|----------------------------------------------------------|
| Initial Circuit   | True             | {'id', 'cz', 'h', 'cx', 'rz', 'ry', 'rx', 'z', 'y', 'x'} |
| Circuit Rewriting | True             | {'cz', 'rx', 'rz'}                                       |
| Gate Merging      | True             | {'cz', 'rx', 'rz'}                                       |
| Swapping          | True             | {'cz', 'rx', 'rz'}                                       |
| Group Replacement | True             | {'cz', 'rx', 'rz'}                                       |
```

In [24]:
```python
optimization_results([qc0, qc1, qc2, qc3, qc4], \
                 ['Initial Circuit', 'Circuit Rewriting', 'Gate Merging', 'Swapping', 'Group Replacemen
t'])
```

```
| Process           | Circuit Depth   | Number of Gates   |
|-------------------|-----------------|-------------------|
| Initial Circuit   | 551             | 1000              |
| Circuit Rewriting | 1160            | 2029              |
| Gate Merging      | 788             | 1325              |
| Swapping          | 756             | 1231              |
| Group Replacement | 591             | 981               |
```

The number of gates is reduced from 2029 to 981 -- more than half of the gates are removed!