Quantum Multiplication Calculator Builder

Abstract

This project is a construction of multiplication calculation calculator with quantum computing processes. It uses qubits, short for quantum bits, and quantum gates, as opposed to classical bits and gates. Quantum bits can simultaneously represent the state of 0 and the state of 1. It is referred to as superposition. Quantum gates, including controlled-not gates (CX or CNOT, CCX or CCNOT, etc.), are special that even its counterparts for AND and XOR gates are also using special quantum gates like RX, RY, RZ gates and Hadamard gate.

Introduction

This project is using Qiskit.
Therefore, for more information about how Quantum Gates are implementing their functions using building blocks like Hadamard gates, etc., refer to this documentation.

Application

Most likely, multiplication by quantum computing might be used to solve complex equations like derivatives and differentials by using the Power Rule, which is as follows:

$$\frac{d}{dx}(cx^n) = c \cdot n \cdot x^{n-1}$$

Trigonometric and Logarithmic Equations

For equations like sin(x), cos(x), ln(1+x), convert them to Maclaurin Series for the Power Rule to be applied like below:

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

$$\ln(1+x) = \sum_{n=1}^{\infty}(-1)^{n-1}\frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$$

It is also compatible for calculations of trigonometric and logarithmic equations, both computations of their derivatives and the approximations of their sums.

Vector Functions

Consider a vector function $\mathbf{r}(t) = 8t^5\,\mathbf{i} + 4t^3\,\mathbf{j} + 6t^3\,\mathbf{k}$
Then, it is possible to apply other formulas to calculate tangent vector, arc length, curvature, torsion, etc. by using Power Rule.

Tangent Vector

$$\frac{ds}{dt} = |\mathbf{r}'(t)|$$

Tangent Unit Vector

$$\mathbf{T}(t) = \frac{\mathbf{r}'(t)}{|\mathbf{r}'(t)|}$$

Arc Length

$$L = \int_a^b |\mathbf{r}'(t)|\,dt$$

Normal Vectors

$$\mathbf{N}(t) = \frac{\mathbf{T}'(t)}{|\mathbf{T}'(t)|}$$

Binormal Vectors

$$\mathbf{B}(t) = \mathbf{T}(t) \times \mathbf{N}(t) = \frac{\mathbf{r}'(t)}{|\mathbf{r}'(t)|} \times \frac{\mathbf{T}'(t)}{|\mathbf{T}'(t)|}$$

Curvature

$$\kappa = \frac{|\mathbf{r}'(t) \times \mathbf{r}''(t)|}{|\mathbf{r}'(t)|^3}$$

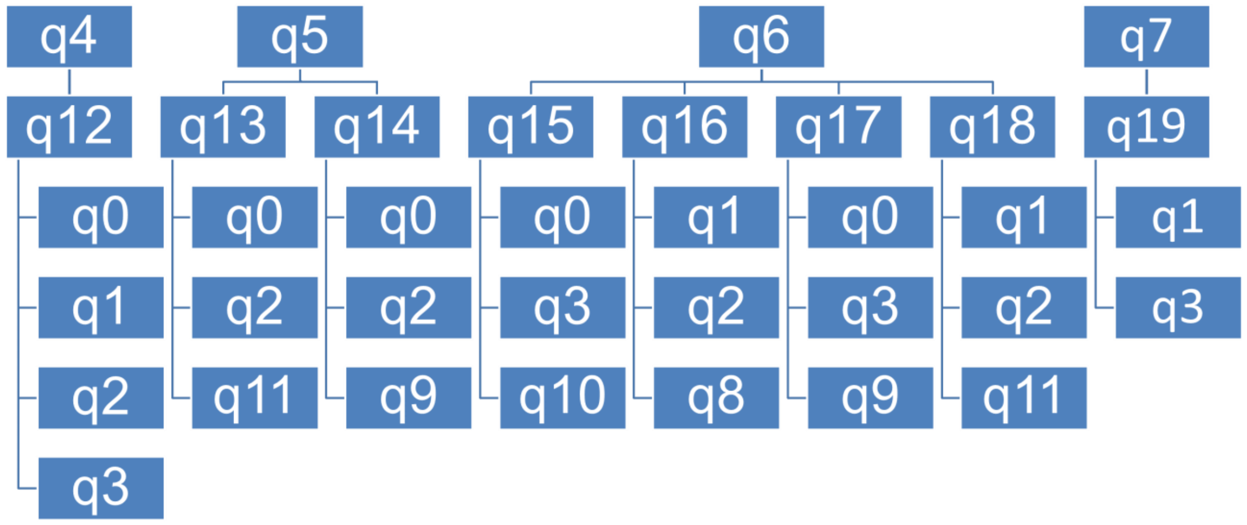
where $\mathbf{T}$: Tangent Unit Vector

Torsion

$$\tau = \frac{[\mathbf{r}'(t) \times \mathbf{r}''(t)] \cdot \mathbf{r}'''(t)}{|\mathbf{r}'(t) \times \mathbf{r}''(t)|^2}$$

Multiplier Hierarchy

Below is a hierarchy of a multiplier example with 2-bit (nbits = 2) showing layers of physical qubits (q0 to q11) being united into logical qubits (q12 to q19), which, in turn, being united into larger logical qubits that are, ultimately, the outputs (q4 to q7).
They are all organized in NAND Gates since they are Sum-of-Products (SOP). Since direct sums using OR gate is not possible in Qiskit quantum computing circuitry, using DeMorgan's Theorem to convert them to NAND Gates might be the alterative to do so.
(Note: Different nbit (number of bits) yield no resemblance with this example below. In other words, a very different hierachy would be produced.)



Import Modules

Import modules, including several Qiskit modules

In[1]:

```
from qiskit import QuantumCircuit, Aer, execute, transpile
from qiskit_aer import AerSimulator

from qiskit.circuit import Instruction, CircuitInstruction, Qubit, QuantumRegister
from qiskit.circuit.library.standard_gates import CXGate, CCXGate, C3XGate, C4XGate, MCXGate
from qiskit.exceptions import QiskitError

import matplotlib.pyplot as plt
%matplotlib inline
from qiskit.visualization import plot_histogram

import os
import sys
import math as m
import numpy as np
import pandas as pd
import sympy
```

```
from _functools import *
from traceback import format_exc
```

Define Functions

Functions that are general but customary should be defined. These functions are considered customary since they may or may not be standard functions and methods provided in default by Python.

Here are the functions:

1. group() function

   This function can be used for grouping a list into a list of lists with the specified length length of the sublists.

   For instance, an array [2, 8, 3, 5, 9] can be grouped into [[2, 8], [3, 5], [9]] if group(array, 2) is called.

   However, if the length of the list is not the multiple of the length of the sublists, the last sublist will have the nmber of the remaining items in the original array. In other words, the length of the last sublist will be the remainder of array_length divided by sub_array_length, which can be calculated as follows (also using % modulo operator in Python):

   Last_Sub_array_length = (array_length) mod (sub_array_length) = (array_length) % (sub_array_length)

   In the example above: array_length = 5 sub_array_length = 2

   (array_length) mod (sub_array_length) = (array_length) % (sub_array_length) = 5 % 2 = 1

   The last array [9] in the grouped array [[2, 8], [3, 5], [9]] has only a length of 1.

2. afmtsd() function

   "afmtsd" stands for "Add for making the same digits".

   This function can add some letters or numbers to the front for various purposes, including sorting purposes. For example, if one desires to sort an array ['00011', '100'], making '100' (3 digits) to become '00100' (5 digits) can make use of this function by calling afmtsd(['00011', '100'], '0', 5).

3. dstatial() function

   "dstatial" stands for "Do something to all things in a list".

   This function is similar if one calls list(map()) function. It means applying a function to all items in the list. This function comes in handy when one does not want to avoid using too many parentheses when calling the list(map()) function.

In[2]:
```
def group(list, subListLen):
    if subListLen != 0:
        grouped_list = []

        try: len_list = len(list)
        except TypeError as errordesc:
            if repr(errordesc) == 'TypeError("object of type \'int\' has no len()")':
                len_list = subListLen+1

        for a in range(0, len_list, int(subListLen)):
            if type(list) == type(0): grouped_list.append('')
            else: grouped_list.append(list[a:a+subListLen])

        if grouped_list == []: grouped_list.append('')

        return grouped_list

    else:
        pass

def afmtsd(the_list_original, chara, ndigits):
    # afmtsd stands for "Add for making the same digits"
    the_list = list(the_list_original)
    a = [chara]*(ndigits-len(the_list)) + the_list

    if type(the_list_original) == type(''): return ''.join(a)
    else: return a

def dstatial(the_function,the_list):
    # "Do something to all things in a list"
    the_new_list = []

    for tla in the_list:
        the_new_list.append(the_function(tla))

    return the_new_list
```

All Combinations of Multiplications

Generate a combination of multiplication by classical computing in order to generate the Sum-of-Products (SOP) equation.

In[3]:

```
data = [0, 1]

nbits = 2
lendata, digits = len(data), nbits*2

numsList = []
ProductCBinFinals = []

# Create Raw Table List of Lists to be converted to Pandas
# so that it can be printed as tables

# (Note: RawTable is treated as a temporary variable designed only for printing
#       so that it can be replaced later for printing other data)
RawTable = []

for i in range(lendata**digits):
    nums = [data[(i//lendata**d)%lendata] for d in range(digits)[::-1]]

    numsGrouped = group(nums, nbits)

    nums0 = numsGrouped[0]
    nums1 = numsGrouped[1]

    # Product using Classical Computation
    # (Convert nums0 and nums1, binary number representation, to decimal numbers A and B)
    A, B = np.sum(2**np.arange(digits//2)[::-1]*nums0), np.sum(2**np.arange(digits//2)[::-1]*nums1)
    ProductClassical = A * B
    # Convert the Classical Product to its Binary (Base 2) version
    ProductCBin = list(afmtsd(bin(ProductClassical)[2:], '0', digits))

    # Finalize the ProductCBin
    ProductCBinFinal = dstatial(int, ProductCBin)

    numsList += [nums]
    ProductCBinFinals += [ProductCBinFinal]

    RawTable.append([i, A, B, nums0, nums1, ProductClassical, nums, ProductCBinFinal])

# Maximize Pandas Column Width
pd.set_option('display.max_colwidth', None)

# Create Pandas DataFrame based on RawTable
# (Note: Table is treated as a temporary variable designed only for printing
#       so that it can be replaced later for printing other data)
Table = pd.DataFrame(RawTable, columns=['i', 'A', 'B', 'nums0', 'nums1', 'ProductClassical = A * B', 'Input
Qubits', 'Output Qubits (ProductCBinFinal)'])

# Align Text Center
Table = Table.style.set_properties(**{'text-align': 'center'})

Table
```

Out[3]:

	i	A	B	nums0	nums1	ProductClassical = A * B	Input Qubits	Output Qubits (ProductCBinFinal)
0	0	0	0	[0, 0]	[0, 0]	0	[0, 0, 0, 0]	[0, 0, 0, 0]
1	1	0	1	[0, 0]	[0, 1]	0	[0, 0, 0, 1]	[0, 0, 0, 0]
2	2	0	2	[0, 0]	[1, 0]	0	[0, 0, 1, 0]	[0, 0, 0, 0]
3	3	0	3	[0, 0]	[1, 1]	0	[0, 0, 1, 1]	[0, 0, 0, 0]
4	4	1	0	[0, 1]	[0, 0]	0	[0, 1, 0, 0]	[0, 0, 0, 0]

	i	A	B	nums0	nums1	ProductClassical = A * B	Input Qubits	Output Qubits (ProductCBinFinal)
5	5	1	1	[0, 1]	[0, 1]	1	[0, 1, 0, 1]	[0, 0, 0, 1]
6	6	1	2	[0, 1]	[1, 0]	2	[0, 1, 1, 0]	[0, 0, 1, 0]
7	7	1	3	[0, 1]	[1, 1]	3	[0, 1, 1, 1]	[0, 0, 1, 1]
8	8	2	0	[1, 0]	[0, 0]	0	[1, 0, 0, 0]	[0, 0, 0, 0]
9	9	2	1	[1, 0]	[0, 1]	2	[1, 0, 0, 1]	[0, 0, 1, 0]
10	10	2	2	[1, 0]	[1, 0]	4	[1, 0, 1, 0]	[0, 1, 0, 0]
11	11	2	3	[1, 0]	[1, 1]	6	[1, 0, 1, 1]	[0, 1, 1, 0]
12	12	3	0	[1, 1]	[0, 0]	0	[1, 1, 0, 0]	[0, 0, 0, 0]
13	13	3	1	[1, 1]	[0, 1]	3	[1, 1, 0, 1]	[0, 0, 1, 1]
14	14	3	2	[1, 1]	[1, 0]	6	[1, 1, 1, 0]	[0, 1, 1, 0]
15	15	3	3	[1, 1]	[1, 1]	9	[1, 1, 1, 1]	[1, 0, 0, 1]

Then, use Sympy Module to organize the SOP from a Dictionary.
Here is an example that is similar to the one mentioned in a Sympy documentation page about SOP.

In[4]:
```
w,x,y,z = sympy.symbols('w,x,y,z')
minterms = [{w: 0, x: 1}, {y: 1, z: 1, x: 0}]
sympy.SOPform([w, x, y, z], minterms)
```

Out[4]:

$$(x \wedge \neg w) \vee (y \wedge z \wedge \neg x)$$

Convert SOP to Quantum Circuit

Here is an important function used to convert SOP to Quantum Circuit:

Method	**Purpose**
Sympy atoms() method	Method of Sympy SOPform object to split by a Conjunction to split AND project

In[5]:

```
ProductCBinFinalsArr = np.array(ProductCBinFinals)
ProductCBinColNum = ProductCBinFinalsArr.shape[1]

termsCompr1 = []
prodinSOPLen = []

qubitNums = digits*2 + ProductCBinColNum

# Create Raw Table List of Lists to be converted to Pandas
# so that it can be printed as tables
RawTable = []

for i in range(ProductCBinColNum):
    ProductArr = ProductCBinFinalsArr[:, i]
    ProductArr1Indices = np.arange(len(ProductArr))[ProductArr == 1]

    # numsArr contains every possible combinations of input states, whether YES or NO
    # ProductArr1Inputs contain the inputs only if they make Product output = 1
    numsArr = np.array(numsList)
    ProductArr1Inputs = numsArr[ProductArr1Indices]

    # Determine how many Input (num of cols in from the table)
    inputNums = ProductArr1Inputs.shape[1]

    ProductRawEq1 = []
    symbols = [sympy.Symbol('q%s' % QubitIndex) for QubitIndex in range(inputNums)]

    for ProductArr1InputsCont in ProductArr1Inputs:
        ProductRawEq1 += [[{symbols[inputIndex]:ProductArr1InputsCont[inputIndex] for inputIndex in
range(len(ProductArr1InputsCont))}]]

    ProductRawEq2 = np.array(ProductRawEq1)
    ProductRawEq3 = ProductRawEq2.flatten().tolist()
    SOP = sympy.SOPform(symbols, ProductRawEq3)
    SOPProducts = SOP.atoms(sympy.And)

    SOP_NAND = bool(1)

    for SOPProduct in SOPProducts:
        SOP_NAND &= ~SOPProduct

    SOP_NAND = ~SOP_NAND

    termsList = [tuple(str(SOPProduct).split(' & ')) for SOPProduct in SOPProducts]
    termsCompr1.extend(termsList)
    prodinSOPLen.append(len(termsList))

    RawTable.append([f'q{i+digits}', SOP_NAND])

# Maximize Pandas Column Width
pd.set_option('display.max_colwidth', None)

# Create Pandas DataFrame based on RawTable
Table = pd.DataFrame(RawTable, columns=['Output Qubits', 'Boolean/Logical Equations'])

# Align Text leftmost
Table = Table.style.set_properties(**{'text-align': 'left'})

# Align Table Header leftmost
Table = Table.set_table_styles([dict(selector = 'th', props=[('text-align', 'center')])])

Table
```

Out[5]:

	Output Qubits	Boolean/Logical Equations
0	q4	q0 & q1 & q2 & q3
1	q5	~(~(q0 & q2 & ~q1) & ~(q0 & q2 & ~q3))

	Output Qubits	Boolean/Logical Equations
2	q6	~(~(q0 & q3 & ~q1) & ~(q0 & q3 & ~q2) & ~(q1 & q2 & ~q0) & ~(q1 & q2 & ~q3))
3	q7	q1 & q3

Physical Qubits Inner Logical Qubits Outer Logical Qubits or Outputs

In[6]:

```
termsCompr2 = []

# Create Raw Table List of Lists to be converted to Pandas
# so that it can be printed as tables
RawTable = []

for termsCompr1Cont in termsCompr1:
    if termsCompr1Cont not in termsCompr2:
        termsCompr2.append(termsCompr1Cont)

        RawTable.append([termsCompr1Cont])

# Maximize Pandas Column Width
pd.set_option('display.max_colwidth', None)

# Create Pandas DataFrame based on RawTable
Table = pd.DataFrame(RawTable, columns=['Inner Logical Qubits'])

# Align Text leftmost
TableStyle1 = Table.style.set_properties(**{'text-align': 'left'})

# Count the number "Logical" Qubits (groups of Physical Qubits: q12 to q19)
# (It is shown in the hierachy on the top of this Notebook with nbits = 2 as an example)
Table.loc['Count'] = Table[['Inner Logical Qubits']].count()

qubitNumsInitial = qubitNums

qubitNums += len(termsCompr2)
```

```
print('Number of Qubits:', qubitNums)

Table

#termsCompr2, len(termsCompr2)
```

```
Number of Qubits: 20
```

Out[6]:

	Inner Logical Qubits
0	(q0, q1, q2, q3)
1	(q0, q2, ~q3)
2	(q0, q2, ~q1)
3	(q1, q2, ~q3)
4	(q1, q2, ~q0)
5	(q0, q3, ~q1)
6	(q0, q3, ~q2)
7	(q1, q3)
Count	8

Organizing by Preparing for the First Layer

Here is another important function used to convert SOP to Quantum Circuit:

Method	Purpose
str.split()	Method of str object to split by letter "q" It is the easiest to do so since it is the only separation between "~" symbol (if any) and the qubit number.

In [7]:

```
termsCompr3 = []

for termsCompr2Index in range(len(termsCompr2)):
    targetQubit = (termsCompr2Index+qubitNumsInitial,)

    # MCXG stands for MCXGate, which is a multiple-qubit-controlled CNOT or CX Gate
    QubitMCXGCommand = ()

    for controlQubit in termsCompr2[termsCompr2Index]:
        controlQubitProfile = controlQubit.split('q')

        # If controlQubitProfile[0] == '':
            # then controlQubit is not inverted
            # Note: '' converted to boolean is False

        if controlQubitProfile[0]:
            controlQubitProfile = int(controlQubitProfile[1])+digits*2

        else:
            controlQubitProfile = int(controlQubitProfile[1])

        QubitMCXGCommand += (controlQubitProfile,) # tuple can only be concatenated, not appended

    termsCompr3.append(QubitMCXGCommand+targetQubit)

print(termsCompr3)
```

```
[(0, 1, 2, 3, 12), (0, 2, 11, 13), (0, 2, 9, 14), (1, 2, 11, 15), (1, 2, 8, 16), (0, 3, 9, 17), (0, 3, 10, 18), (1, 3,
19)]
```

Define print_exec() as a combination between exec() and print()

print() function	For printing the command
exec() function	For executing the command from strings

In [8]:

```
def print_exec(command, sep=' ', end='\n'):
    print(command, sep=sep, end=end)
    exec(command)
```

Create a Quantum Circuit

mCirc stands for "Multiplier Circuit"

In[9]:

```
mCirc = QuantumCircuit(qubitNums, digits)
```

Create Inverse Qubits

In[10]:

```
for InputNum in range(digits):
    inverseInputNum = InputNum+digits*2
    print_exec(f'mCirc.cx({InputNum}, {inverseInputNum})')
    print_exec(f'mCirc.x({inverseInputNum})')
```

```
mCirc.cx(0, 8)
mCirc.x(8)
mCirc.cx(1, 9)
mCirc.x(9)
mCirc.cx(2, 10)
mCirc.x(10)
mCirc.cx(3, 11)
mCirc.x(11)
```

Simplify by organizing several qubits onto one

Creation of Inner Layer of Logical Qubits

In[11]:

```
operationSOP1 = []
operationSOP1Checkpoint = 0

for operationSOP1Index in prodinSOPLen:
    slicedTerms =
termsCompr3[operationSOP1Checkpoint:operationSOP1Checkpoint+operationSOP1Index]

    operationSOP1.append(slicedTerms)
    operationSOP1Checkpoint += operationSOP1Index

print(prodinSOPLen, end='\n\n\n')
print(operationSOP1, end='\n\n\n')
```

```
[1, 2, 4, 1]
```

```
[[(0, 1, 2, 3, 12)], [(0, 2, 11, 13), (0, 2, 9, 14)], [(1, 2, 11, 15), (1, 2, 8, 16), (0, 3, 9, 17), (0, 3, 10, 18)], [(1,
3, 19)]]
```

Qiskit Implementation

In[12]:

```
for operationSOP1Cont in operationSOP1:
    for SOPProduct in operationSOP1Cont:
        print_exec(f'mCirc.append(MCXGate(num_ctrl_qubits={len(SOPProduct)-1}), {SOPProduct})')
        print_exec(f'mCirc.x({SOPProduct[-1]})', end='\n\n')

    print()
```

```
mCirc.append(MCXGate(num_ctrl_qubits=4), (0, 1, 2, 3, 12))
mCirc.x(12)



mCirc.append(MCXGate(num_ctrl_qubits=3), (0, 2, 11, 13))
mCirc.x(13)

mCirc.append(MCXGate(num_ctrl_qubits=3), (0, 2, 9, 14))
mCirc.x(14)



mCirc.append(MCXGate(num_ctrl_qubits=3), (1, 2, 11, 15))
mCirc.x(15)

mCirc.append(MCXGate(num_ctrl_qubits=3), (1, 2, 8, 16))
mCirc.x(16)

mCirc.append(MCXGate(num_ctrl_qubits=3), (0, 3, 9, 17))
mCirc.x(17)

mCirc.append(MCXGate(num_ctrl_qubits=3), (0, 3, 10, 18))
mCirc.x(18)



mCirc.append(MCXGate(num_ctrl_qubits=2), (1, 3, 19))
mCirc.x(19)
```

Creation of Outer Layer of Logical Qubits (Output Qubits)

In[13]:

```
operationSOP2 = []

for operationSOP1Index in range(len(operationSOP1)):
    operationSOP1Cont = operationSOP1[operationSOP1Index]

    opSOP1ControlQubits = tuple(operationSOP1aCont[-1] for operationSOP1aCont in
operationSOP1Cont)
    opSOP1TargetQubits = (operationSOP1Index+digits,)

    operationSOP2Cont = opSOP1ControlQubits + opSOP1TargetQubits
    operationSOP2.append(operationSOP2Cont)

print(operationSOP2, end='\n\n\n')
```

```
[(12, 4), (13, 14, 5), (15, 16, 17, 18, 6), (19, 7)]
```

Qiskit Implementation

In[14]:

```
for operationSOP2Cont in operationSOP2:
    print_exec(f'mCirc.append(MCXGate(num_ctrl_qubits={len(operationSOP2Cont)-1}),
{operationSOP2Cont})')
    print_exec(f'mCirc.x({operationSOP2Cont[-1]})', end='\n\n')
```

```
mCirc.append(MCXGate(num_ctrl_qubits=1), (12, 4))
mCirc.x(4)

mCirc.append(MCXGate(num_ctrl_qubits=2), (13, 14, 5))
mCirc.x(5)

mCirc.append(MCXGate(num_ctrl_qubits=4), (15, 16, 17, 18, 6))
mCirc.x(6)
mCirc.append(MCXGate(num_ctrl_qubits=1), (19, 7))
mCirc.x(7)
```

Qubit Measurement

"Measure a quantum bit (qubit) in the Z basis into a classical bit (cbit)."
(Source: https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html#qiskit.circuit.QuantumCircuit.measure)

In[15]:
```
mCirc.measure(np.arange(digits)+digits, range(digits)[::-1])
```

Out[15]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1fb527155d0>
```

Determine Input Value (Either 0 or 1)

Initialization Gates

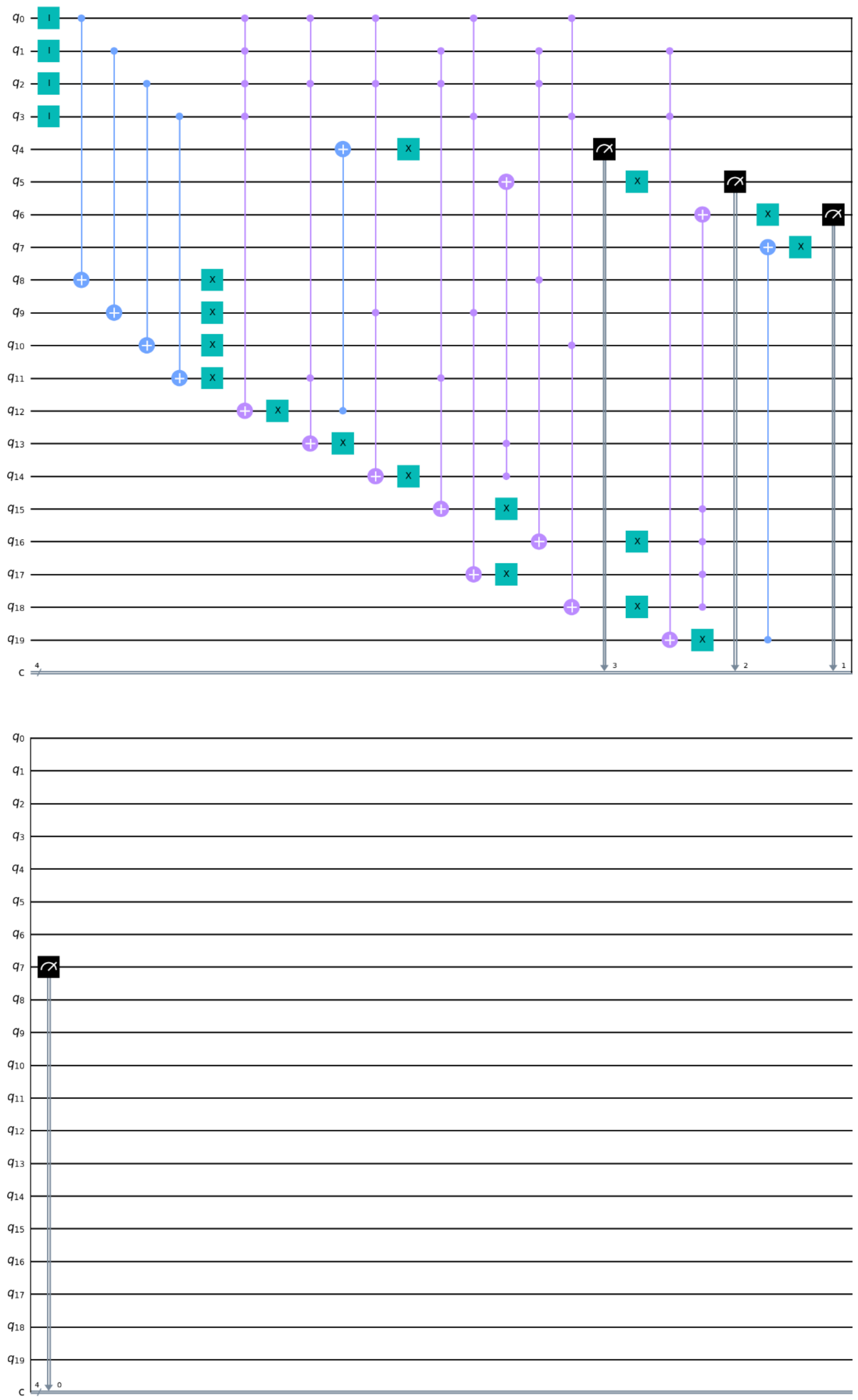
First, zero it out at the beginning.
All qubits start from ground state |0>. Create manipulable initialization gates as many as the input qubits.
The identity gate means that it remains the same state as previous, which, in this case, the ground state |0>.

Later, the Identity gate can be converted to an X Gate or NOT gate.

In[16]:
```
# Determine whether new Initialization Gates are needed to be inserted or not
inputDet = sum([circInst.operation.name == 'x' or circInst.operation.name == 'id' for circInst in
mCirc.data[:digits]]) == digits

# If the Initialization Gates do not exists or are incomplete, then insert them.
if inputDet < digits:
    initGates = []
    qreg = mCirc.data[0].qubits[0].register

    for inputIndex in range(digits):
        initGates.append(CircuitInstruction(operation=Instruction(name='id', num_qubits=1, num_clbits=0,
params=[]),
                                            qubits=(Qubit(qreg, inputIndex),),
                                            clbits=()))

    mCirc.data = initGates + mCirc.data

mCirc.draw(output='mpl')
```

```
C:\Users\mulia\AppData\Local\Temp\ipykernel_20064\1908320097.py:7: DeprecationWarning: The
property ``qiskit.circuit.bit.Bit.register`` is deprecated as of qiskit-terra 0.17. It will be removed no earlier
than 3 months after the release date. Instead, use
:meth:`~qiskit.circuit.quantumcircuit.QuantumCircuit.find_bit` to find all the containing registers within a
circuit and the index of the bit within the circuit.
  qreg = mCirc.data[0].qubits[0].register
```

Out[16]:



q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{11}
q_{12}
q_{13}
q_{14}
q_{15}
q_{16}
q_{17}
q_{18}
q_{19}
c
I
X
4
3
2
1
0

If 0: Apply IGate (since Qubits start at Ground State or 0)

If 1: Apply XGate

Then, put the XGates and/or IGates in front of all other instructions so that they would be implemented first.

Backends used

Furthermore, there are 2 types of backends, which are quantum computing simulators and real quantum computers:

Backend Name	Capability
qasm_simulator	Implementations only with lower memory usage
matrix_product_state Simulator	Implementations with higher memory usage

Finally, check whether the classical computing multiplications are **exactly** the same with their quantum counterpart.

In[17]:

```
backends = [Aer.get_backend('qasm_simulator'),
            AerSimulator(method='matrix_product_state')]
```

Implement the Program with Combinations of Input States

Here, matrix_product_state Simulator is used since it can implement with higher memory usage

In[18]:

```
# Create Raw Table List of Lists to be converted to Pandas
# so that it can be printed as tables
RawTable = []

try:
   backend = backends[1]

   for i in range(lendata**digits):
      nums = []
      inputDet = []

      for d, inputIndex in zip(range(digits)[::-1], range(digits)):
          # Binary Input
          BinInput = data[(i//lendata**d)%lendata]

          nums.append(BinInput)

          # BinInput can be 0 or 1, translating to "i" or "x", respectively
          name = ['id', 'x'][BinInput]

          mCirc.data[inputIndex].operation.name = name

      # Product using Classical Computation
      numsGrouped = group(nums, nbits)
      nums0 = numsGrouped[0]
      nums1 = numsGrouped[1]

      A, B = np.sum(2**np.arange(digits//2)[::-1]*nums0), np.sum(2**np.arange(digits//2)[::-1]*nums1)
      ProductClassical = A * B

      ProductCBin = list(afmtsd(bin(ProductClassical)[2:], '0', digits))
      ProductCBinFinal = dstatial(int, ProductCBin)

      result = execute(mCirc, backend=backend).result()
      counts = result.get_counts()

      countsFinal = dstatial(int, list(list(counts)[0]))
      countsFinalNp = np.array(countsFinal)

      Product_mCirc = np.sum(2**np.arange(digits)[::-1]*countsFinalNp)

      RawTable.append([i, A, B, numsGrouped, ProductCBinFinal, countsFinal, Product_mCirc, np.array(
ProductCBinFinal) == countsFinalNp])

      # Maximize Pandas Column Width
      pd.set_option('display.max_colwidth', None)

      # Create Pandas DataFrame based on RawTable
      Table = pd.DataFrame(RawTable, columns=['i', 'A', 'B', 'numsGrouped', 'ProductCBinFinal', 'countsFi
nal', 'Product_mCirc', 'np.array(ProductCBinFinal) == countsFinalNp'])

      # Align Text leftmost
      Table = Table.style.set_properties(**{'text-align': 'left'})

      # Align Table Header leftmost
      Table = Table.set_table_styles([dict(selector = 'th', props=[('text-align', 'center')])])
```

```
except QiskitError:
    # Get error message by using traceback.format_exc()
    errmsg = format_exc()

    # If the error is really Insufficient memory problem, then output it using file=sys.stderr in the print() funct
ion
    if "ERROR: Insufficient memory to run circuit circuit-120 using the statevector simulator." in errmsg:
        print('\n'+errmsg, file=sys.stderr)
```

Table

Out[18]:

	i	A	B	Nums Grouped	ProductCBinFinal	Counts Final	Product_ mCirc	np.array(ProductCBinFinal) == countsFinalNp
0	0	0	0	[[0, 0], [0, 0]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
1	1	0	1	[[0, 0], [0, 1]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
2	2	0	2	[[0, 0], [1, 0]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
3	3	0	3	[[0, 0], [1, 1]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
4	4	1	0	[[0, 1], [0, 0]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
5	5	1	1	[[0, 1], [0, 1]]	[0, 0, 0, 1]	[0, 0, 0, 1]	1	[True True True True]
6	6	1	2	[[0, 1], [1, 0]]	[0, 0, 1, 0]	[0, 0, 1, 0]	2	[True True True True]
7	7	1	3	[[0, 1], [1, 1]]	[0, 0, 1, 1]	[0, 0, 1, 1]	3	[True True True True]
8	8	2	0	[[1, 0], [0, 0]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
9	9	2	1	[[1, 0], [0, 1]]	[0, 0, 1, 0]	[0, 0, 1, 0]	2	[True True True True]
10	10	2	2	[[1, 0], [1, 0]]	[0, 1, 0, 0]	[0, 1, 0, 0]	4	[True True True True]
11	11	2	3	[[1, 0], [1, 1]]	[0, 1, 1, 0]	[0, 1, 1, 0]	6	[True True True True]

	i	A	B	Nums Grouped	ProductCBinFinal	Counts Final	Product_mCirc	np.array(ProductCBinFinal) == countsFinalNp
12	12	3	0	[[1, 1], [0, 0]]	[0, 0, 0, 0]	[0, 0, 0, 0]	0	[True True True True]
13	13	3	1	[[1, 1], [0, 1]]	[0, 0, 1, 1]	[0, 0, 1, 1]	3	[True True True True]
14	14	3	2	[[1, 1], [1, 0]]	[0, 1, 1, 0]	[0, 1, 1, 0]	6	[True True True True]
15	15	3	3	[[1, 1], [1, 1]]	[1, 0, 0, 1]	[1, 0, 0, 1]	9	[True True True True]