量子計算期末專案報告

108755001 應物碩二 郭庭愷

動機

當我在量子計算課程中，學習到加法器(adder)等基本的運算元件時，就立刻想像到:在不遠的未來，當量子計算的硬體技術困難被解決後，例如退相干(decoherence)時間、多量子位元錯誤更正等等……通用的量子電腦就會出現巨大的需求，科技公司就會開始著手設計規格化的，量子版的邏輯電路。我認為從近通用運算(near-term computing)到真正的量子電腦的過渡時間並不會太短，因此此相關的問題兼具了前瞻性與實用性。

何謂量子基本算術運算

我們對於傳統的代數運算都十分熟悉。如果用布林代數(Boolean algebra)，也就是二進制位元(bit)來表示代數，可以大致分為兩種:

(1) 二元算數(binary arithmetic): 加(addition)、減(substraction)、乘(multiplication)、除(division)、取餘數(或模，modulus)

(2) 位元間的邏輯運算(bitwise operation): 且(AND),或(OR),異或(XOR),非(NOT)。

在傳統的邏輯電路，也就是以電晶體為主的電路上，是利用電壓控制與表示邏輯與數值上的 0 與 1。而現在我們要在量子邏輯電路上重新設計與實現這些基本算術運算。但此量子邏輯電路是可以獨立於其物理層的實作方式，無論是基於固態元件亦或是基於超導體、量子陷阱、核磁共振等等。

基於這些最最基本的運算，我們可以構造更為複雜的運算作為更高層次運算的基本元件。

主要貢獻

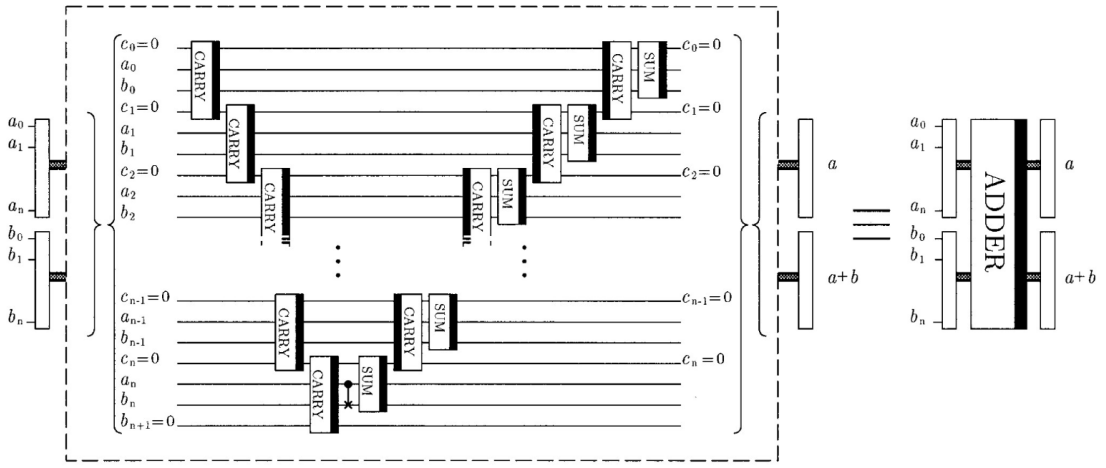
我參考了[1]，去解決量子電腦的基本代數運算的問題。問題的目標是設計出$\mathrm{a}^{\mathrm{b}}\ mod\ N$ 運算的量子版本電路。但礙於時間限制，無法完全實現此電路。因此我退而求其次，實現其部分重要元件$\mathrm{a}+\mathrm{b}\ mod\ N$。

方法論

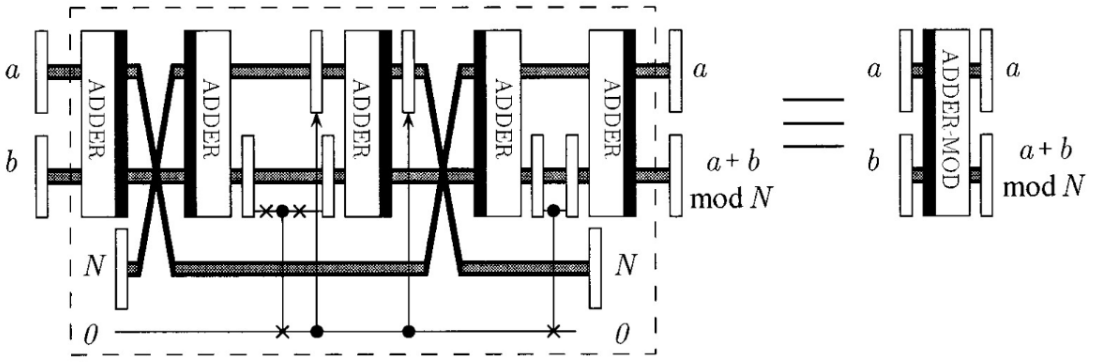
我認為原本論文的思路是採取階層化(hierarchical)的邏輯電路設計。這是因為當要實作的目標運算複雜到一定程度後上述，純粹使用基本運算規則的電路是很難直接設計出來的。我們需要透過階層化的設計，將幾個較基本的運算單元連結組合後，進行封裝(encapsulation)，將這個封裝後的模組(module)視為一個運算的黑盒子(blackbox)，這樣反而能夠讓電路的整體架構顯得更加清晰。

具體步驟如下:

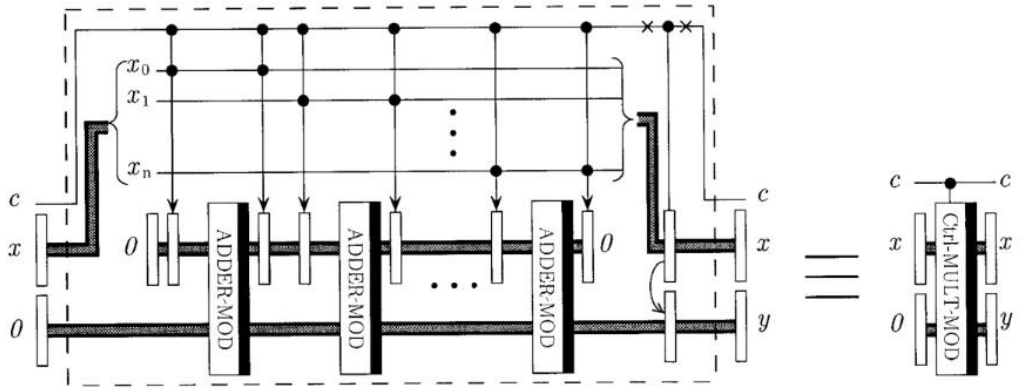
1. 建立一個量子加法器，如[1]中 Fig.2 所示:



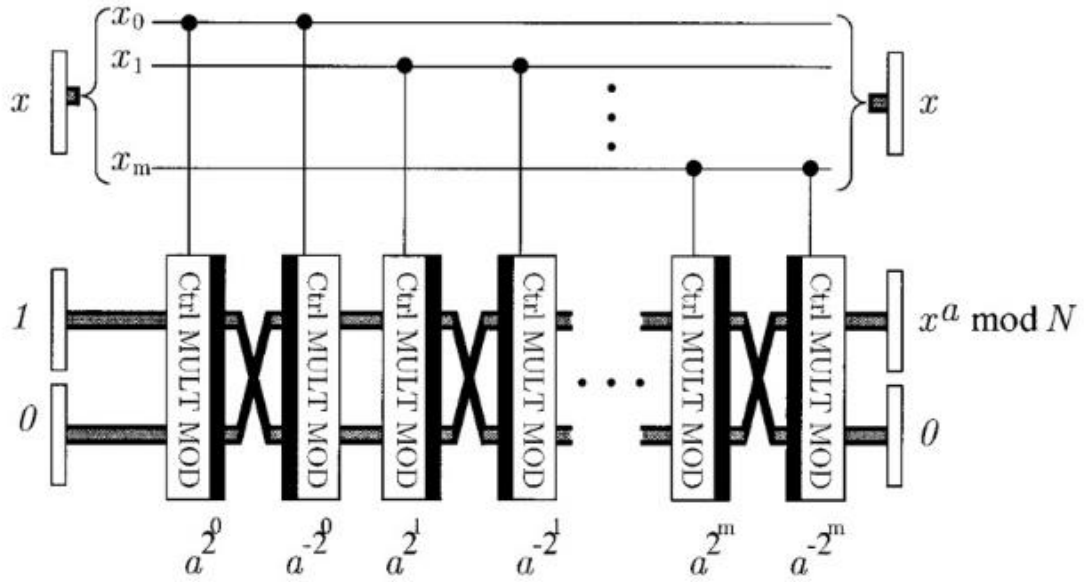
2. 根據(1)，建立一個量子餘數器。這也是目前專案實作的進度。如[1]中 Fig.4 所示:



3. 根據(2)，建立一個 quantum controlled multiple moduler，如[1]中 Fig.5 所示:

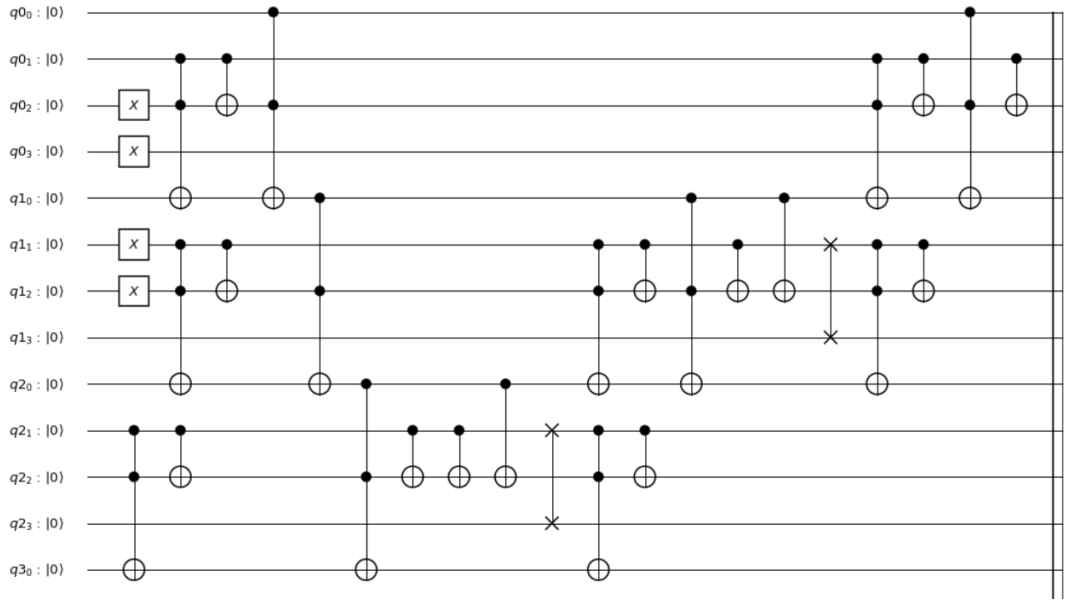


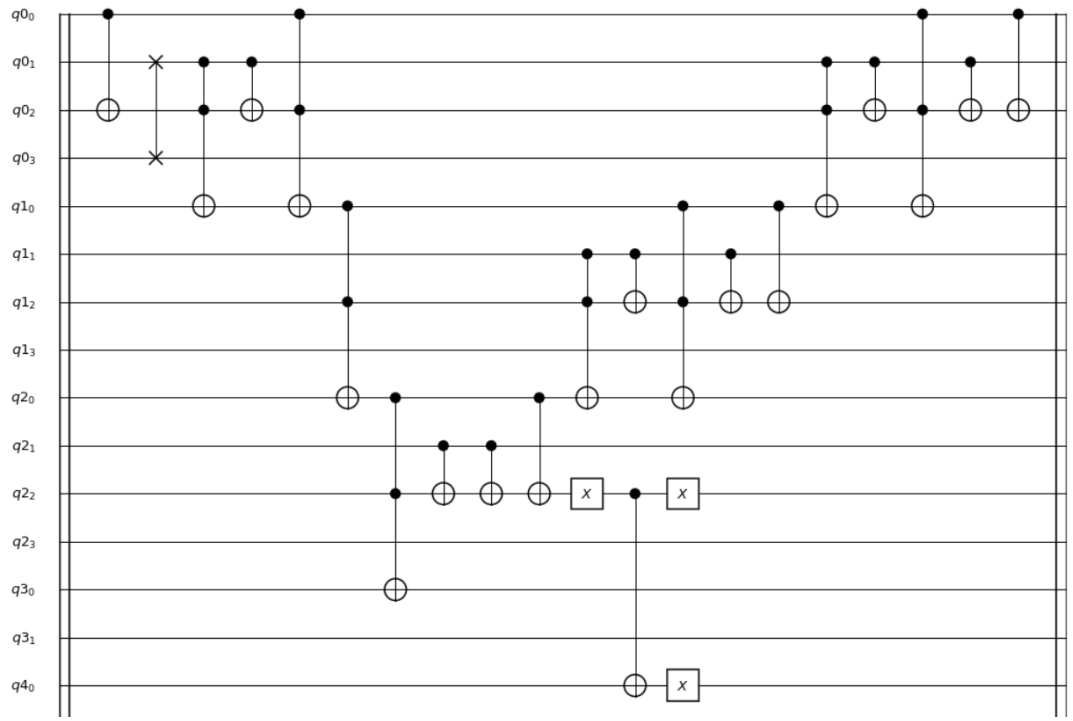
4. 根據(3)，建立一個 quantum modular exponential 電路，如[1]中 Fig.6 所示:

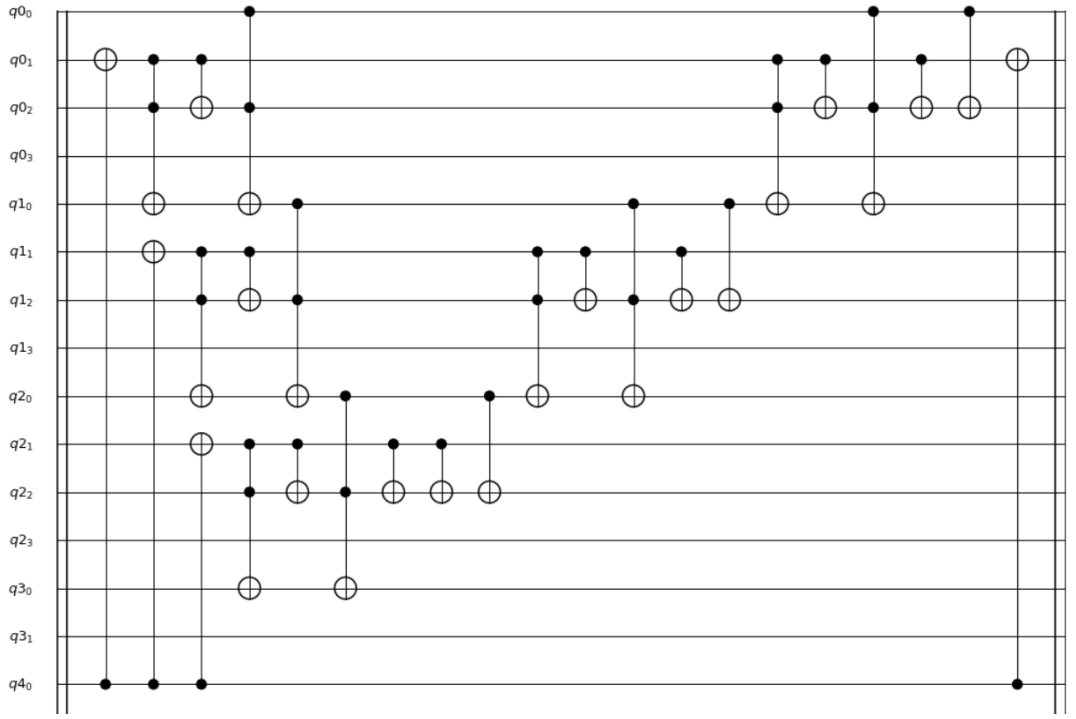


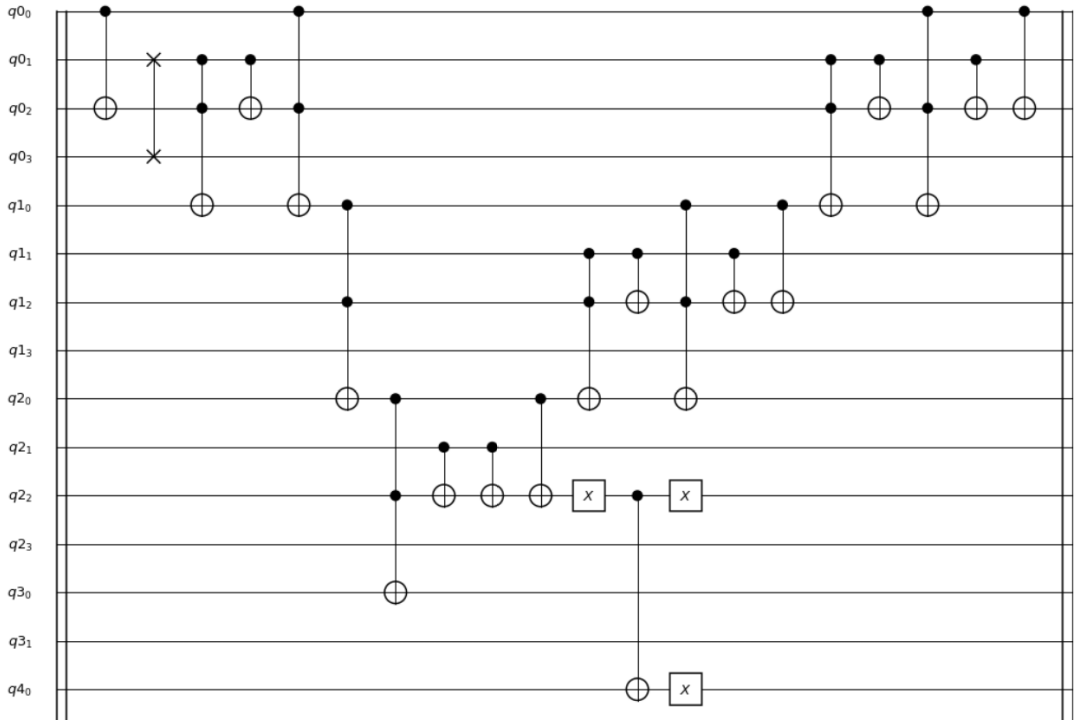
結果與討論

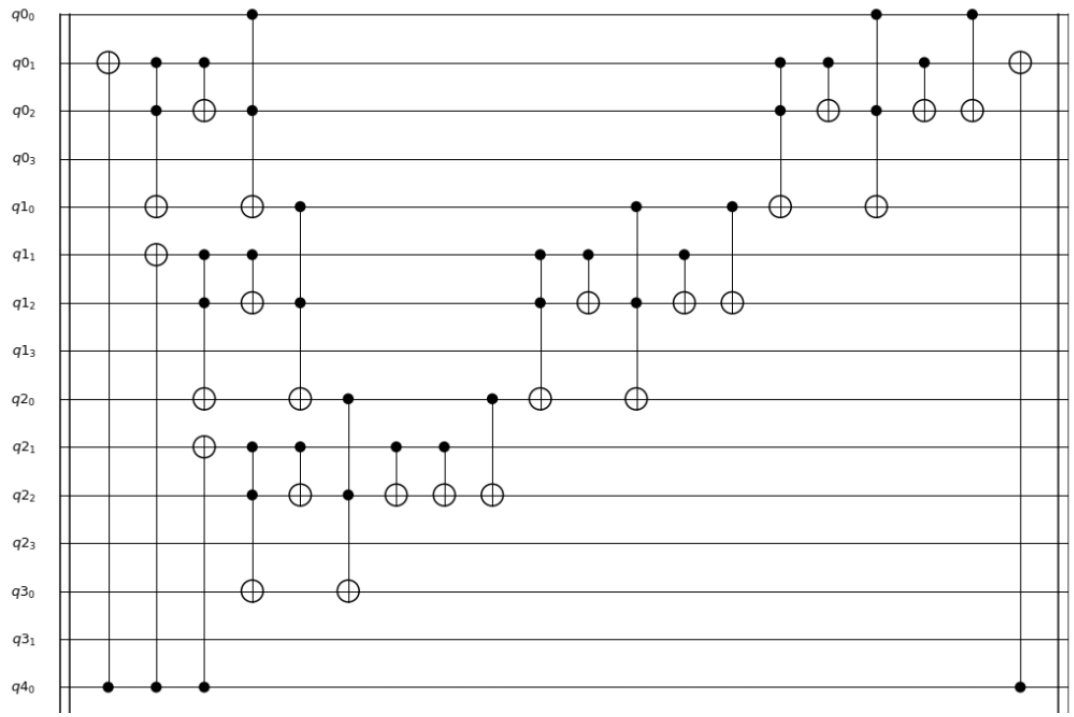
實作出的電路結果如下。由於原本的電路由左至右展開過長，因此將其拆分成由上而下的閱讀順序，如圖所示:

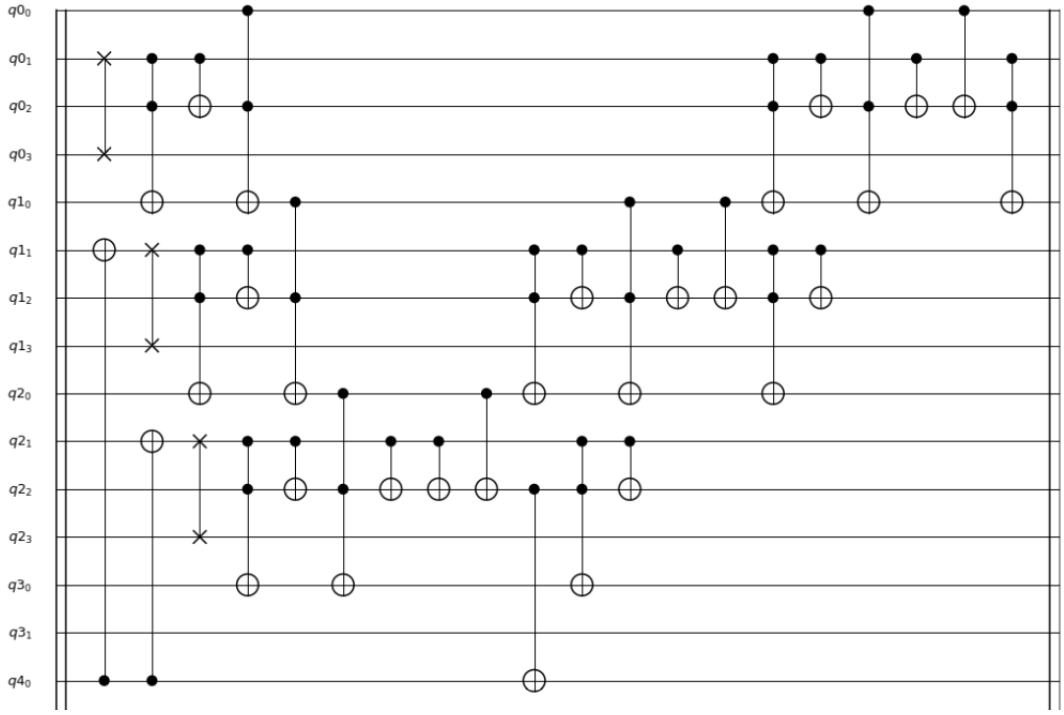

$q0_0$: |0⟩
$q0_1$: |0⟩
$q0_2$: |0⟩
$q0_3$: |0⟩
$q1_0$: |0⟩
$q1_1$: |0⟩
$q1_2$: |0⟩
$q1_3$: |0⟩
$q2_0$: |0⟩
$q2_1$: |0⟩
$q2_2$: |0⟩
$q2_3$: |0⟩
$q3_0$: |0⟩
X
X
X
X

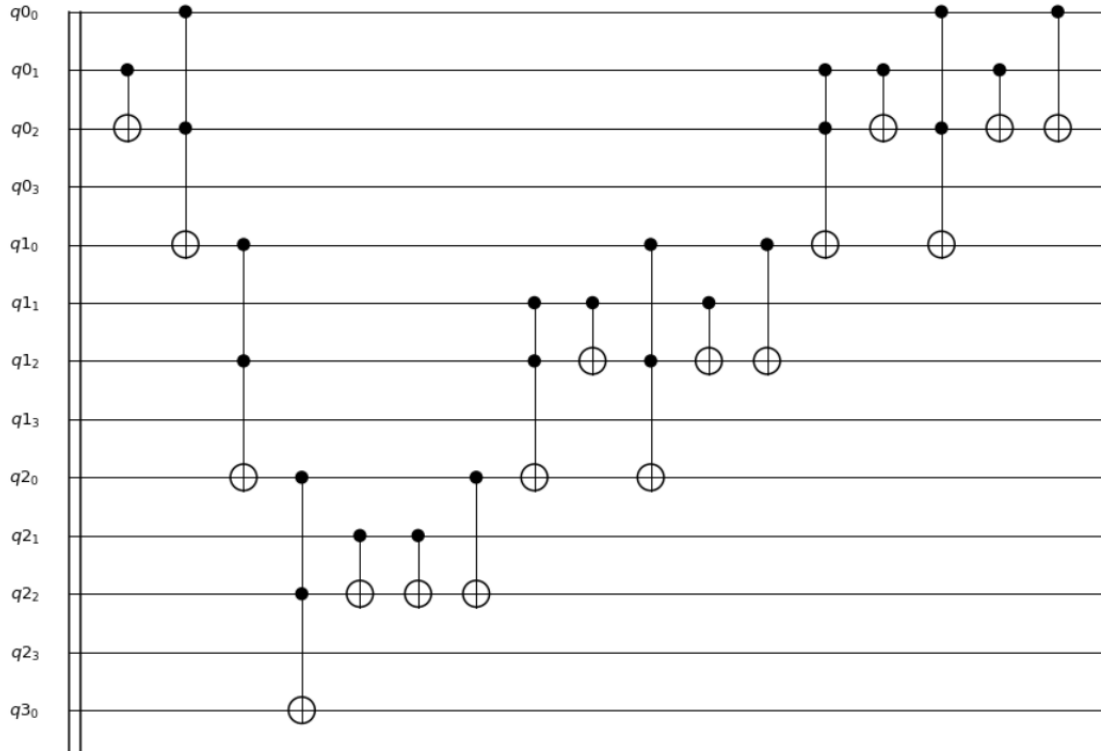

$q0_0$
$q0_1$
$q0_2$
$q0_3$
$q1_0$
$q1_1$
$q1_2$
$q1_3$
$q2_0$
$q2_1$
$q2_2$
$q2_3$
$q3_0$
$q3_1$
$q4_0$
X
X
X


$q0_0$
$q0_1$
$q0_2$
$q0_3$
$q1_0$
$q1_1$
$q1_2$
$q1_3$
$q2_0$
$q2_1$
$q2_2$
$q2_3$
$q3_0$
$q3_1$
$q4_0$


$q0_0$
$q0_1$
$q0_2$
$q0_3$
$q1_0$
$q1_1$
$q1_2$
$q1_3$
$q2_0$
$q2_1$
$q2_2$
$q2_3$
$q3_0$
$q3_1$
$q4_0$
X
X
X


$q0_0$
$q0_1$
$q0_2$
$q0_3$
$q1_0$
$q1_1$
$q1_2$
$q1_3$
$q2_0$
$q2_1$
$q2_2$
$q2_3$
$q3_0$
$q3_1$
$q4_0$


$q0_0$
$q0_1$
$q0_2$
$q0_3$
$q1_0$
$q1_1$
$q1_2$
$q1_3$
$q2_0$
$q2_1$
$q2_2$
$q2_3$
$q3_0$
$q3_1$
$q4_0$



可以看出，如果將階層化的功能模組完全展開成電路，則做為一個人類，是很難判讀出此電路的功能性的。

除了前述的模組，我還寫了兩個輔助性質的模組，一個是用於電路輸入的初始化(initializer)，可以直接將十進制的數值轉為二進制表示的狀態；另外一個是實現兩個位元之間狀態的交換(swaper)，因為交換操作在此電路中頻繁出現。

我測試了我的 4 量子位元輸入版本電路，經過量測後，證實了其正確性。由於此電路考慮過程中使用的量子位元，總共需要 24 個量子位元。但目前 IBMQ experience 只提供了最多 16 位元的伺服器，因此目前只能在 qasm 上模擬結果，運行時間總計 0.00899 秒。

未來展望

除了未來有時間可以實作出整個電路之外，還希望能夠驗證其計算複雜度。另外，論文中提到這個電路可以做為更高效的 Shor 演算法的基石。未來也可以往此方向研究。

參考文獻

[1] Vedral, Barenco, Ekert, “Quantum Networks for Elementary Arithmetic Operations”, PhysRevA54.147,1996

addmod

January 15, 2020

[1]:
```
import qiskit
from qiskit import ClassicalRegister, QuantumRegister,QuantumCircuit,Aer,execute
from qiskit.tools.monitor import job_monitor,backend_monitor,backend_overview
from qiskit.tools.visualization import plot_histogram
from qiskit import IBMQ
import numpy as np

from matplotlib import pyplot as plt
```

[2]:
```
device = Aer.get_backend('qasm_simulator')
# the maximum number qubit of a simulator = 32
```

[4]:
```
def adder(num1,num2):

    qc.ccx(zero[num1],zero[num2],one[0])
    qc.cx(zero[num1],zero[num2])
    qc.ccx(zero[0],zero[num2],one[0])
#     qc.barrier()

    qc.ccx(one[num1],one[num2],two[0])
    qc.cx(one[num1],one[num2])
    qc.ccx(one[0],one[num2],two[0])
#     qc.barrier()

    qc.ccx(two[num1],two[num2],three[0])
    qc.cx(two[num1],two[num2])
    qc.ccx(two[0],two[num2],three[0])
#     qc.barrier()

    qc.cx(two[num1],two[num2])
#     qc.barrier()

    qc.cx(two[num1],two[num2])
    qc.cx(two[0],two[num2])
#     qc.barrier()

    qc.ccx(one[num1],one[num2],two[0])
    qc.cx(one[num1],one[num2])
```

```
    qc.ccx(one[0],one[num2],two[0])
#     qc.barrier()

    qc.cx(one[num1],one[num2])
    qc.cx(one[0],one[num2])
#     qc.barrier()

    qc.ccx(zero[num1],zero[num2],one[0])
    qc.cx(zero[num1],zero[num2])
    qc.ccx(zero[0],zero[num2],one[0])
#     qc.barrier()

    qc.cx(zero[num1],zero[num2])
    qc.cx(zero[0],zero[num2])
#     qc.barrier()



def init_num(num1,num2,num3):
    num1 = bin(num1)[2:]
    num2 = bin(num2)[2:]
    num3 = bin(num3)[2:]
    for i,qb in enumerate(num1[::-1]):
        if qb=='1':
            if i==0:
                qc.x(zero[1])
            if i==1:
                qc.x(one[1])
            if i==2:
                qc.x(two[1])
    for i,qb in enumerate(num2[::-1]):
        if qb=='1':
            if i==0:
                qc.x(zero[2])
            if i==1:
                qc.x(one[2])
            if i==2:
                qc.x(two[2])
    for i,qb in enumerate(num3[::-1]):
        if qb=='1':
            if i==0:
                qc.x(zero[3])
            if i==1:
                qc.x(one[3])
            if i==2:
                qc.x(two[3])
```

```
def swap(num1,num2):
    qc.swap(zero[num1],zero[num2])
    qc.swap(one[num1],one[num2])
    qc.swap(two[num1],two[num2])


def addmodN(a,b,N,posa,posb,posN):
    init_num(a,b,N)
    adder(posa,posb)
#     qc.barrier()
    # need swap N and a
    swap(posa,posN)
#     qc.barrier()
    adder(posa,posb)
#     qc.barrier()
    qc.x(two[posb])
    qc.cx(two[posb],aux[0])
    qc.x(two[posb])
#     qc.barrier()
    qc.x(aux[0])
    qc.cx(aux[0],zero[posa])
    qc.cx(aux[0],one[posa])
    qc.cx(aux[0],two[posa])
#     qc.barrier()
    adder(posa,posb)
#     qc.barrier()
    qc.cx(aux[0],zero[posa])
    qc.cx(aux[0],one[posa])
    qc.cx(aux[0],two[posa])
#     qc.barrier()
    swap(posa,posN)
#     qc.barrier()
    adder(posa,posb)
#     qc.barrier()
    qc.cx(two[posb],aux[0])
#     qc.barrier()
    adder(posa,posb)
```

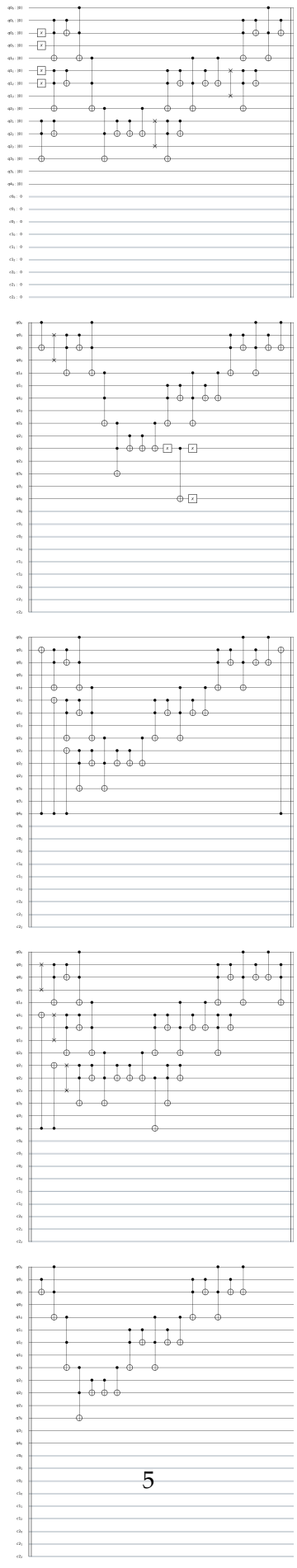
[5]:
```
# transpile()
# addmodN(a=3,b=5,N=4,posa=1,posb=2,posN=3)
# qc.draw(output='mpl')
```

[6]:
```
# An three qubit a add b mod N quantum circuit

zero = QuantumRegister(4)
one = QuantumRegister(4)
two = QuantumRegister(4)
three = QuantumRegister(2)
```

```
aux = QuantumRegister(1)
m1 = ClassicalRegister(3)
m2 = ClassicalRegister(3)
m3 = ClassicalRegister(3)

qc = QuantumCircuit(zero,one,two,three,aux,m1,m2,m3)
import time

ti = time.time()
init_num(2,3,1)
adder(1,2)
#qc.barrier()
# need swap N and a
swap(1,3)
#qc.barrier()
adder(1,2)
#qc.barrier()
qc.x(two[2])
qc.cx(two[2],aux[0])
qc.x(two[2])
#qc.barrier()
qc.x(aux[0])
qc.cx(aux[0],zero[1])
qc.cx(aux[0],one[1])
qc.cx(aux[0],two[1])
#qc.barrier()
adder(1,2)
#qc.barrier()
qc.cx(aux[0],zero[1])
qc.cx(aux[0],one[1])
qc.cx(aux[0],two[1])
#qc.barrier()
swap(1,3)
#qc.barrier()
adder(1,2)
#qc.barrier()
qc.cx(two[2],aux[0])
#qc.barrier()
adder(1,2)

tf = time.time()

print(tf-ti)
qc.draw(output='mpl')
```

```
0.008996009826660156
```

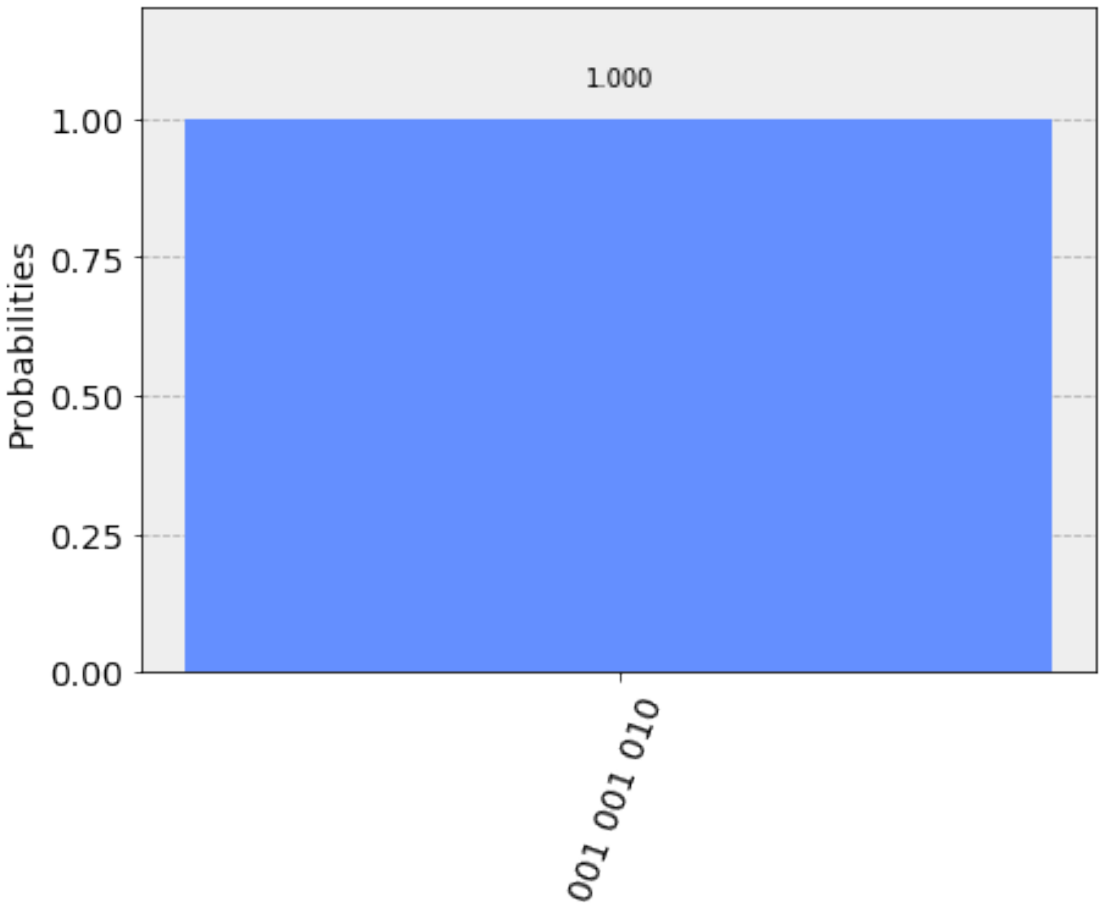
[6]:



[7]:
```
ti1 = time.time()
qc.measure(zero[1],m1[0])
qc.measure(one[1],m1[1])
qc.measure(two[1],m1[2])

qc.measure(zero[2],m2[0])
qc.measure(one[2],m2[1])
qc.measure(two[2],m2[2])

qc.measure(zero[3],m3[0])
qc.measure(one[3],m3[1])
qc.measure(two[3],m3[2])

tf1 = time.time()
print(tf1-ti1)

prob = execute(qc,device,shots=1024)
result = prob.result()

# Nf = list(result.get_counts().keys())[0][0:3]
# apbf = list(result.get_counts().keys())[0][4:7]
# af = list(result.get_counts().keys())[0][8:11]

# print('a final:',af)
# print('b final:',apbf)
# print('n final:',Nf)
plot_histogram(result.get_counts())
# plt.show()
# qc.draw(output='mpl')
```

```
0.0019981861114501953
```

[7]:



[ ]:
```
# a=2:010
# a+b mod 0: 010+011 = 101
# N =0

# a=2:010 b=011
# a+b mod 1: 010+011mod1 = 101mod1 = 1
# N =1
```

[8]:
```
qc.qubits
```

[8]:
```
[(QuantumRegister(4, 'q0'), 0),
 (QuantumRegister(4, 'q0'), 1),
 (QuantumRegister(4, 'q0'), 2),
 (QuantumRegister(4, 'q0'), 3),
 (QuantumRegister(4, 'q1'), 0),
 (QuantumRegister(4, 'q1'), 1),
 (QuantumRegister(4, 'q1'), 2),
 (QuantumRegister(4, 'q1'), 3),
 (QuantumRegister(4, 'q2'), 0),
 (QuantumRegister(4, 'q2'), 1),
```

```
(QuantumRegister(4, 'q2'), 2),
(QuantumRegister(4, 'q2'), 3),
(QuantumRegister(2, 'q3'), 0),
(QuantumRegister(2, 'q3'), 1),
(QuantumRegister(1, 'q4'), 0)]
```

[ ]:

[ ]: