UNIVERSITÀ DEGLI STUDI DI VERONA

DIPARTIMENTO DI INFORMATICA

Final project for the "Quantum Computing" course:

Quantum desk calculator in Qiskit

A.A 2020/2021

Studenti:

Luca Marzari VR457336
Deborah Pintani VR464800

Professoressa:

ALESSANDRA DI PIERRO

Contents

Chapter 1

Problem definition

The task is to build a quantum circuit that performs the difference between the binary representation of two positive integers. The circuit can be defined as a variation of the quantum adder explained in class and using the Quantum Fourier Transform. The circuit must be implemented in `Qiskit` and demonstrated on a one or two instances. The project can be extended with the implementation of a complete desk calculator.

Chapter 2

Building the calculator

First of all we implement the addition operation in the same way as it is implemented on a classical computer (construction of a Full Adder) and through the use of the Toffoli gate. Next we will outline the disadvantages of these approaches and analyze the advantages of using the Quantum Fourier Transform (QFT) instead.

We will then continue the report by implementing, based on the considerations written about QFT, all other operations such as: subtraction, multiplication and division. Each operation will be tested both on the Qiskit offline simulator and on the online simulator made available by IBM (`https://quantum-computing.ibm.com/services?services=simulators`). The source code of this project is available at `https://github.com/LM095/QuantumDeskCalculator`

2.1 Addition

We begin by translating the sum of two numbers in a quantum circuit. There are actually many ways to solve the problem, and we show two of them.

The first one is called the "Toffoli sum", since it uses Toffoli gates and CNOT gates. It is the simpler one since it mimics the well-known classical sum circuit (half adder and ripple-carry adder).
The second approach is using the Quantum Fourier Transform (QFT), which transforms between two bases, the computational (Z) basis, and the Fourier basis. Using this method, we count the different rotations around the Z axis.

2.1.1 Toffoli Addition

Understanding the fundamentals

Before diving into the actual code, it is important to understand the fundamentals.

The Toffoli Addition is based on the classical binary sum circuits: the Half Adder and the Ripple-Carry Adder (RCA).

Let us first take a look at the Half Adder Circuit, that is shown in Figure 2.1.

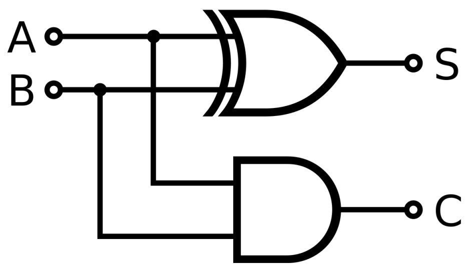


Figure 2.1: Classic Half Adder circuit

In short, the Half Adder Circuit executes the sum of the two input bits `A` and `B` (where they can be 0 or 1), outputting the addition in the `S` output. Furthermore, it calculates the carry bit in the `C` output. The carry bit is one when both `A` and `B` are one.
However, the circuit does not consider a possible carry-in bit. This problem is overcame in the Full Adder circuit.

The Half Adder Circuit is composed of an `AND` gate and a `XOR` gate. The two inputs are passed both to the `XOR` logic gate, which will be responsible for computing the partial sum, and to the `AND` logic gate, which will be responsible for computing the carry.
The truth table of the Half Adder is shown in Figure 2.2.

From the table we can deduce that the add function can be described by the following equations:

$$S = A_i \oplus B_i$$

$$C_{out} = A_i \cdot B_i$$

As mentioned before, the Half Adder does not consider a possible carry-in bit, making the circuit only suitable to add two single bits. To consider the carry-in bit, we have to build a circuit that accept 3 inputs: `A`, `B` and

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2.2: Truth table of the Half Adder Circuit

`Cin`, where the first two are the bits to add, and the last one is the carry-in bit that the previous operation computed. This circuit is called Full Adder. The circuit is shown in Figure 2.3, while the truth table can be found in Figure 2.4.

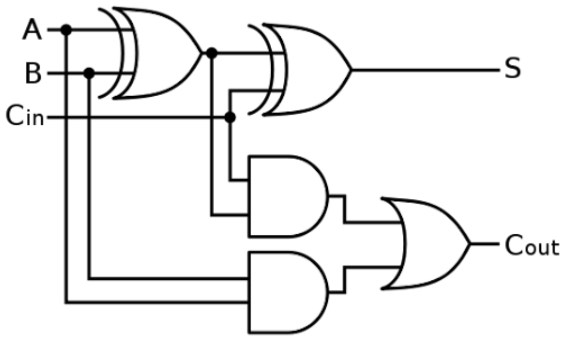


Figure 2.3: Full Adder Circuit

Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.4: Truth table of the Full Adder Circuit

The `S` output is again the `XOR` of the three inputs, while the `Cout` output is one if at least two of the three inputs are one.

The boolean expressions for the Full Adder are the following ones:

$$S = (A_i \oplus B_i) \oplus Cin_i$$
$$C_{out} = (A_i \cdot B_i) + ((A_i \oplus B_i) \cdot Cin_i)$$

When performing a binary additions however, we want to add more than a pair of bits. We can use many Full Adder circuits in cascade to create a more complex circuit. This approach creates a Ripple-Carry Adder (RCA). In particular, if the size of the largest number to add is `n`, the RCA uses `n` Full Adders. The carry-out bit of the `i`-th RCA is the carry-in bit of the following `i+1`-th RCA.

An example of a RCA circuit is shown in Figure 2.5.

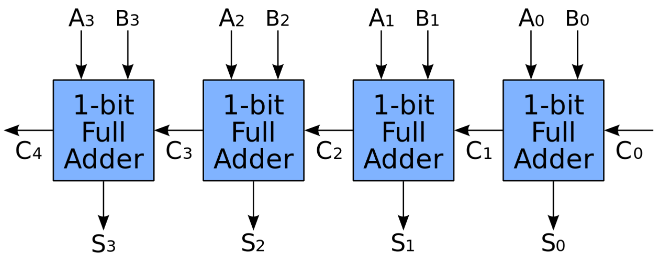


Figure 2.5: An example of a Ripple-Carry Adder

Implementation in Qiskit

In Qiskit we implement a direct translation of the Ripple-Carry Adder (RCA).

First, we ask two numbers to add. Note that we decided to keep both number under 64, since the max number that a user can input is $63_{10} = 111111_2$, which is 6 bits. Listing 2.1 shows this first part.

```python
from qiskit import *

input1 = int(input("Enter a positive integer between 0 and 15: "))
input2 = int(input("Enter a positive integer between 0 and 15: "))

while (input1 < 0 or input1 > 15) or (input2 < 0 or input2 > 15):
    input1 = int(input("Enter a positive integer between 0 and
    15: "))
    input2 = int(input("Enter a positive integer between 0 and
    15: "))

first = '{0:{fill}3b}'.format(input1, fill='0')
second = '{0:{fill}3b}'.format(input2, fill='0')
```

Listing 2.1: Asking the user two numbers to sum

Then, we need to initialize the quantum registers that will hold: the two input numbers, the carry bits, the result and three support bits. Last register to declare is a `ClassicalRegister`, which is used to read the final sum.

To know how many bits there are in each register, we first need to find the largest of the two input numbers, take its length and use it to correctly initialize the registers. At the end, we draw the circuit.
Notice that the result `QuantumRegister ris` and the `ClassicalRegister cl` contains `n+1` bits. That is because the sum of two large numbers could overflow and potentially add an additional bit to represent the result. The corresponding code is shown in Listing 2.2.

```
# We check which number is the longest and we take that length
    'n' to declare two quantumRegisters that contain 'n'
   qubits.
l = len(first)
l2 = len(second)
if l > l2:
    n = l
else:
    n = l2

# Initializing the registers; three quantum registers with n
   bits each
# 1 more with n+1 bits, which will hold the sum of the two #
   numbers
# The last q-register has three bits to temporally store
   information
# The classical register has n+1 bits, which is used to make
   the sum readable
a = QuantumRegister(n) #First number
b = QuantumRegister(n) #Second number
c = QuantumRegister(n) #Carry bits
ris = QuantumRegister(n+1) #Result bits
supp = QuantumRegister(3) #support bit
cl = ClassicalRegister(n+1) #Classical output

# Combining all of them into one quantum circuit
```

```
qc = QuantumCircuit(a, b, c, supp, ris, cl)
qc.draw('mpl')
```

Listing 2.2: Initializing the registers

To execute the sum, we need to translate the input numbers, and to do so, we iterate over them. When a `1` is found, we apply an X-gate to negate the corresponding q-bit, which is always initialized to `0`, effectively flipping the q-bit to `1`.
Notice that in addition to applying the X-gate, we write the corresponding q-bit in reverse order, encoding the number in reverse. That is because first of all Qiskit's least significant bit has the lowest index (0) and moreover when we execute the sum, we usually start from the end of the number, the least significant digits, and move on to the most significant digit. The corresponding code is shown in Listing 2.3.

```
# Setting up the registers using the values inputted
# since the qubits initially always start from 0, we use not
   gate to set 1 our qubits where in the binary number
# inserted there is a 1
for i in range(l):
    if first[i] == "1":
        qc.x(a[l - (i+1)]) #Flip the qubit from 0 to 1
for i in range(l2):
    if second[i] == "1":
        qc.x(b[l2 - (i+1)]) #Flip the qubit from 0 to 1
qc.draw('mpl')
```

Listing 2.3: Encoding the classical bits to quantum bits

In this part, we are in the heart of computation. We translate and apply the Ripple-Carry Adder boolean expressions that we saw in Subsubsection 2.1.1.
We recall them here:

$$S = (A_i \oplus B_i) \oplus Cin_i$$
$$C_{out} = (A_i \cdot B_i) + ((A_i \oplus B_i) \cdot Cin_i)$$

Notice that we iterate over `n-1`: that is because the last (more significant) digit does not have to write its C_{out} in the `c` q-register, but instead it has to write it in the `result` q-register (this is the case where we have to use the `n+1` digit in the result).

The sum `S` is pretty straight-forward, while the C_{out} part is more complex.

In the Carry part, we replaced the OR gate with the XOR. In fact the two AND, which are the inputs for the OR, can never both be equal to one, so

the last case in the truth table (A=1, B=1) of XOR and OR does not apply, making XOR and OR equal.
We did this replacement because there is no easy way to effectively encode an OR gate in a quantum circuit, while the XOR one is pretty much already included.

Moreover, we had to declare and use three support q-bits (`supp`), to temporarily store intermediate gates results. At the end of each iteration we reset them.

The code snippet is shown in Listing 2.4.

```python
# Ripple Carry Adder

# sum = (a xor b) xor cin
# cout = (a and b) OR ((a xor b) and cin)
# Note: instead of using an OR gate we can use a XOR one. In
   fact the two AND, which are the inputs for the OR, can
   never both be equal to one, so the last case in the truth
   table (A=1, B=1) of XOR and OR does not apply, making XOR
   and OR equal

for i in range(n-1):
    #SUM
    #sum = (a xor b) xor cin
    # use cnots to write the XOR of the inputs on qubit 2
    qc.cx(a[i], ris[i])
    qc.cx(b[i], ris[i])

    qc.cx(c[i], ris[i])

    #CARRY
    #cout = (a and b) XOR ((a xor b) and cin)
    #cout = i+1
    #cin = i
    # use ccx to write the AND of the inputs on qubit 3
    qc.ccx(a[i],b[i],supp[2])

    qc.cx(a[i], supp[0])
    qc.cx(b[i], supp[0])

    qc.ccx(supp[0], c[i], supp[1])

    qc.cx(supp[1], c[i+1])
    qc.cx(supp[2], c[i+1])

    #reset supp to 0
    qc.reset([9]*5)
    qc.reset([10]*5)
    qc.reset([11]*5)
```

Listing 2.4: Realizing the sum on `n-1` q-bits

Outside the loop, we execute the sum of the last (most significant) digit, where we store the C_{out} output in the last result bit instead of passing it at the next C_{in}.
The code is shown in Listing 2.5.

```
#SUM
#sum = (a xor b) xor cin
qc.cx(a[n-1], ris[n-1])
qc.cx(b[n-1], ris[n-1])

qc.cx(c[n-1], ris[n-1])

#CARRY
#cout = (a and b) XOR ((a xor b) and cin)
#cout = i+1
#cin = i
# use ccx to write the AND of the inputs on qubit 3
qc.ccx(a[n-1],b[n-1],supp[2])

qc.cx(a[n-1], supp[0])
qc.cx(b[n-1], supp[0])

qc.ccx(supp[0], c[n-1], supp[1])

#note: instead of saving the last cout in the c qubit, we
   store it in the last digit of the result (hence the longer
   ris + 1)
qc.cx(supp[1], ris[n])
qc.cx(supp[2], ris[n])

#reset to 0 supp
qc.reset([9]*5)
qc.reset([10]*5)
qc.reset([11]*5)
```

Listing 2.5: Realizing the sum on the last q-bit

At the end, we measure the `result` using the `ClassicalRegister cl` as the output, and we draw the circuit. Then, we execute the job on the simulator and print the final result.

The code is shown in Listing 2.6

```
for i in range(n+1):
    qc.measure(ris[i], cl[i])
qc.draw('mpl')

#Set chosen backend and execute job
num_shots = 5 #Setting the number of times to repeat
   measurement
job = execute(qc, backend=Aer.get_backend('qasm_simulator'),
   shots=num_shots)

#Get results of program
job_stats = job.result().get_counts()
print(job_stats)
```

Listing 2.6: Measuring and printing the result

We decided to not depict an example circuit in this report because it is unnecessary: adding two numbers using this 'classical method' generates a huge and not easily understandable circuit. To see a more reasonable circuit for the sum, we refer to the Subsubsection 2.1.2.

2.1.2 Quantum Fourier Transform Addition

Before we start discussing the implementation of addition using Quantum Fourier Transform (QFT), we present the mathematical formula and how it is possible to implement the circuit that performs this operation. As mentioned before, the QFT is the quantum implementation of the discrete Fourier transform over the amplitudes of a wave function and transforms the computational basis into the Fourier basis.

Let us see an example with one qubit case: we know that with one qubit we have the states zero ($|0\rangle$) and the state one ($|1\rangle$), so there are two basis states. For two qubits we have $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, so 2^2 possible basis state. Generalizing this process we will have with n qubits 2^n possible states. So for simplicity let us consider $N = 2^n$, and $|\tilde{x}\rangle = QFT\,|x\rangle$ where $\tilde{x}$ is the Fourier basis mentioned before and x is our computational basis. So we have the QFT formula:

$$QFT\,|x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi i x y}{N}} \,|y\rangle$$

in the case of one qubit we have:

$$\begin{aligned}
QFT\,|x\rangle &= \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1} e^{\frac{2\pi ixy}{N}}\,|y\rangle \\
&= \frac{1}{\sqrt{2}}\sum_{y=0}^{2-1} e^{\frac{2\pi xy}{2}}\,|y\rangle \\
&= \frac{1}{\sqrt{2}}\left[e^{\frac{2\pi ix0}{2}}\,|0\rangle + e^{\frac{2\pi ix1}{2}}\,|1\rangle\right] \\
&= \frac{1}{\sqrt{2}}\left[\,|0\rangle + e^{i\pi x}\,|1\rangle\right] \\
&= \frac{1}{\sqrt{2}}\left[\,|0\rangle + |1\rangle\,\right] = |+\rangle \quad \text{in the case of x=0} \\
&= \frac{1}{\sqrt{2}}\left[\,|0\rangle - |1\rangle\,\right] = |-\rangle \quad \text{in the case of x=1}
\end{aligned}$$

Now if we consider this single qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and we calculate the QFT as done before, this operation is exactly the result of applying the Hadamard operator*(H)* on the qubit:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

If we apply the H operator to the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ we obtain the new state:

$$\frac{1}{\sqrt{2}}(\alpha+\beta)|0\rangle + \frac{1}{\sqrt{2}}(\alpha-\beta)|1\rangle \equiv \tilde{\alpha}|0\rangle + \tilde{\beta}|1\rangle$$

In general we can write the QFT[1] as:

$$
\begin{aligned}
QFT_N|x\rangle &= \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}\omega_N^{xy}|y\rangle \\
&= \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1} e^{2\pi i xy/2^n}|y\rangle \text{ since } \omega_N^{xy} = e^{2\pi i\frac{xy}{N}} \text{ and } N = 2^n \\
&\text{rewriting in fractional binary notation } y = y_1\dots y_n, y/2^n = \sum_{k=1}^{n} y_k/2^k \\
&= \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1} e^{2\pi i\left(\sum_{k=1}^{n} y_k/2^k\right)x}|y_1\dots y_n\rangle \\
&\text{rewriting in fractional binary notation } y = y_1\dots y_n, y/2^n = \sum_{k=1}^{n} y_k/2^k \\
&= \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}\prod_{k=1}^{n} e^{2\pi i x y_k/2^k}|y_1\dots y_n\rangle \\
&\text{after rearranging the sum and products, and expanding} \\
&= \frac{1}{\sqrt{N}}\bigotimes_{k=1}^{n}\left(|0\rangle + e^{2\pi i x/2^k}|1\rangle\right)\sum_{y=0}^{N-1} = \sum_{y_1=0}^{1}\sum_{y_2=0}^{1}\cdots\sum_{y_n=0}^{1} \\
&= \frac{1}{\sqrt{N}}\left(|0\rangle + e^{\frac{2\pi i}{2}x}|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{2^2}x}|1\rangle\right)\otimes\dots\otimes\left(|0\rangle + e^{\frac{2\pi i}{2^{n-1}}x}|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{2^n}x}|1\rangle\right)
\end{aligned}
$$

Example: Let us consider $n = 3$ qubits, so $N = 2^3 = 8$, if we take $|x\rangle = |5\rangle = |101\rangle$ we write:

$$
QFT\,|x\rangle = |\tilde{x}\rangle = |\tilde{5}\rangle = \frac{1}{\sqrt{8}}\left(|0\rangle + e^{\frac{2\pi i}{2}5}\,|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{4}5}|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{8}5}|1\rangle\right)
$$

Given these equations that describe mathematically what QFT does, let us see what is the circuit that implements this operation and what it actually does on the qubits.
We have seen that we can write the QFT of any arbitrary number of qubits as:

$$
|\tilde{x}\rangle = \frac{1}{\sqrt{N}}\left(|0\rangle + e^{\frac{2\pi i}{2^1}x}\,|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{2^2}x}|1\rangle\right)\otimes\left(|0\rangle + e^{\frac{2\pi i}{2^3}x}|1\rangle\right)\otimes\cdots\otimes\left(|0\rangle + e^{\frac{2\pi i}{N}x}|1\rangle\right)
$$

with $N = 2^n$.
So what we write is $|x\rangle = |x_1\rangle\otimes|x_2\rangle\otimes\cdots\otimes|x_n\rangle$ where we map to $|x_1\rangle \to \left(|0\rangle + e^{\frac{2\pi i}{2^1}x}\,|1\rangle\right)$, to $|x_2\rangle \to \left(|0\rangle + e^{\frac{2\pi i}{2^2}x}\,|1\rangle\right)$ and so on. So each qubit goes from $|x_k\rangle$ to $|0\rangle + e^{\frac{2\pi i x}{2^k}}\,|1\rangle$.

Ingredients to build the circuit

- Hadamard gate: Hadamard gate applied to a particular qubit x_k, so x_k is either zero or one has two possible results that could be expressed as
$$H|x_k\rangle = |0\rangle + e^{\frac{2\pi i x_k}{2}}|1\rangle/\sqrt{2}$$
in other words Hadamard gate applied with that generic state x_k allows us to apply a phase of $e^{\frac{2\pi i x_k}{2}}$ to the qubit
- a two-qubit controlled rotation $CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix}$ where $UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{2\pi i}{2^k}\right) \end{bmatrix}$ The action of $CROT_k$ on a two-qubit state $|x_l x_j\rangle$ is simple: if the first qubit controlled $x_l = 0$, so $CROT_k|0x_j\rangle$ the operation does nothing, i.e. leaves the state of x_j unchanged: $CROT_k|0x_j\rangle = |0x_j\rangle$. Otherwise, is the qubit controlled is 1 a rotation is applied: $CROT_k|1x_j\rangle = \left(e^{\frac{2\pi i x_j}{2^k}}\right)|1x_j\rangle$.

Given these two gates, a circuit that implements an n-qubit QFT is shown in Figure 2.6.

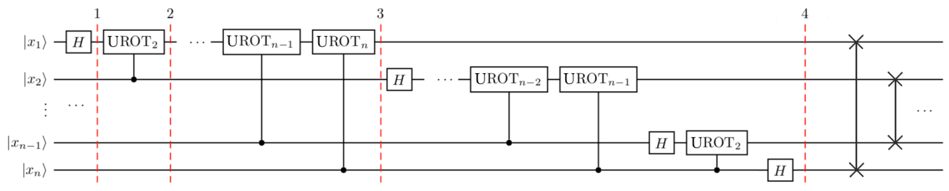


Figure 2.6: Circuit that implements an n-qubit QFT

We now present the algorithm via implementation in Qiskit of the above image in Listing 2.7.

```
from qiskit import *
import numpy as np
pi = np.pi

def ourQFT(nqubits):
    ourQFT_circuit = QuantumCircuit(nqubits)

    for qubit in range(nqubits):
        ourQFT_circuit.h(qubit)
        for otherqubit in range(qubit+1,nqubits):
            ourQFT_circuit.cu1( pi/ (2**(otherqubit-qubit)) ,
 otherqubit, qubit)
```

```

    return ourQFT_circuit
```

Listing 2.7: Our implementation of QFT

So in order to implement the circuit that performs QFT on nqubits ($|x_1 x_2 \cdots x_n\rangle$):

1. we use Hadamard on the first qubit and transform the input state to:

$$H_1|x_1 x_2 \ldots x_n\rangle = \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2 x_3 \ldots x_n\rangle$$

2. we apply $UROT_2$ on the first qubit controlled by the second qubit and we obtain:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2 x_3 \ldots x_n\rangle$$

This is done inside the for loop with command `ourQFT_circuit.cu1` which is a diagonal and symmetric gate that induces a phase on the state of the target qubit, depending on the control state. After the application of the last $UROT_n$ gate on the first qubit controlled by the n-th qubit we have:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x_n + \frac{2\pi i}{2^{n-1}}x_{n-1} + \ldots + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2 x_3 \ldots x_n\rangle$$

which can be written as:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes |x_2 x_3 \ldots x_n\rangle$$

3. we repeat this process until the last qubit and at the end we have:

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^{n-1}}x\right)|1\rangle\right] \otimes$$

$$\ldots \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^1}x\right)|1\rangle\right]$$

which is exactly the QFT of the input state as derived above.

Kindly note that here the order of qubits is reversed in the output state, so we must reverse the order of the qubits.

Why using QFT for sum? What does it mean?

Even in scientific quantum literature, researchers first suggested reversible circuit version of known classical sum implementation, like we did in Subsection 2.1.1.

However, there is not a convenient way to mirror the classical Ripple-Carry adder in the quantum world. First, because we need to extend the circuit to allow reversible computation. Then, we obviously need to add some supplementary memory registers to hold information (carry, support bits, result etc...). Lastly, computing a sum using a classical implementation in a quantum circuit takes a very long time.

The ideal addition algorithm for a quantum computer may not be similar to its classical counterpart. An elegant quantum alternative is using the Quantum Fourier Transform (QFT)[2]. In particular, the use of the QFT saves memory and time.

The QFT can be interpreted as a change of basis: the computational (Z) basis, and the Fourier basis.

In the computational basis, we store numbers in binary using the states $|0\rangle$ and $|1\rangle$. In the Fourier basis however, we store numbers using different rotations around the Z-axis. The number we want to store regulates the angle at which each qubit is rotated around the Z-axis.

Intuitively, the sum of two qbits using the QFT can be seen as the sum of the rotations around the Fourier axis.
Begin by encoding the first number using the QFT. Then, starting from the number just encoded, sum for each qbit the rotations of the second number. Finally, apply the inverse of the QFT, which brings the result in the computational basis. It is like we are counting the number of rotations for each qbit.

Implementation in Qiskit

We divided the code using functions, so let us explain them one by one.
We are not going to fully explain the math behind them, because that part is covered at the beginning of Subsection 2.1.2.

The function `executeQFT` computes the QFT of the first input number, effectively changing the basis from the computational one to the Fourier one.
As seen in Subsubsection 2.1.2, the first ingredient to build a QFT circuit

is the Hadamard gate, one for each qbit register.
Then, we need to apply as many controlled rotation (`cr`) as the qbit binary position.

To better understand the last concept, let us examine Figure 2.7, which depicts the quantum translation and QFT of the number 1_{10} using 4 bits: 0001_2.

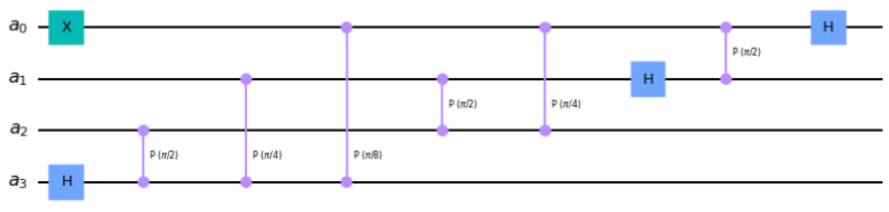


Figure 2.7: Hadamard and controlled rotation example on number 0001_2

You can also see that the number is encoded backwards: `a3` is 0, `a2` is 0, `a1` is 0 and `a0` is 1.
This notation actually makes sense if you think about the binary positional system: the last digit in a binary number - the least significant - has index `0`, while the first digit - the most significant - has index `n-1` where `n` is the number's length.

In this particular example, excluding the Hadamard gates, when applying controlled rotations, we apply:

- 3 rotations to `a3`, which is the most significant qbit in position 3;
- 2 rotations to `a2` that has position 2;
- 1 rotation to `a1`, in position 1;
- no rotation to `a0`, in position 0, which is the least significant qbit.

In this way, when applying controlled rotations, we follow the number's positional indexes.

Listing 2.8 shows the code to compute the QFT.

```python
import math
import operator
from qiskit import *

'''
qc: input quantum circuit
reg: input register to execute QFT
```

```python
n: n-th qbit to apply hadamard and phase rotation
pie: pie number
'''
def executeQFT(qc, reg, n, pie):
    # Executes the QTF of reg, one qubit a time
    # Apply one Hadamard gate to the n-th qubit of the
    # quantum register reg, and then apply repeated phase
    # rotations with parameters being pi divided by
    #increasing powers of two

    qc.h(reg[n])
    for i in range(0, n):
        #cp(theta, control_qubit, target_qubit[,    ])
        qc.cp(pie/float(2**(i+1)), reg[n-(i+1)], reg[n])
```

Listing 2.8: QFT: `executeQFT` function

The function `evolveQFTStateSum` evolves the previous encoded state $|F(\psi(reg_a))\rangle$ of the first number to $|F(\psi(reg_a + reg_b))\rangle$, now also considering the second number.
Intuitively, starting from the number just encoded using the QFT, sum for each qbit the corresponding rotations of the second number.

The operation is very similar to the previous function just described above.

The code is shown in Listing 2.9.

```python
'''
qc: input quantum circuit
reg_a: first input register to execute QFT
reg_b: second input register to execute QFT
n: n-th qbit to apply hadamard and phase rotation
pie: pie number
'''
def evolveQFTStateSum(qc, reg_a, reg_b, n, pie):
    # Evolves the state |F(psi(reg_a))> to
    # |F(psi(reg_a+reg_b))> using the QFT conditioned on the
    # qubits of the reg_b.
    # Apply repeated phase rotations with parameters being pi
    # divided by increasing powers of two.

    l = len(reg_b)
    for i in range(n+1):
        if (n - i) > l - 1:
            pass
        else:
            qc.cp(pie/float(2**(i)), reg_b[n-i], reg_a[n])
```

Listing 2.9: QFT: `evolveQFTStateSum` function

After executing the previous two function, it is time to go back to the computational basis. To compute so, the function `inverseQFT` applies repeated phase rotations and then apply a Hadamard gate.

Unlike the function `executeQFT`, the inverse is applied starting from the last qbit - the least significant - and going to the first qbit - the most significant -.
However, the procedure always respects the rule described above, where for each qbit we apply as many controlled rotation as the digit's position in the binary system.

An example of inverse QFT using the number 0001_2, as in the previous example, is depicted in Figure 2.8.

The code in Listing 2.10 shows the inverse QFT.

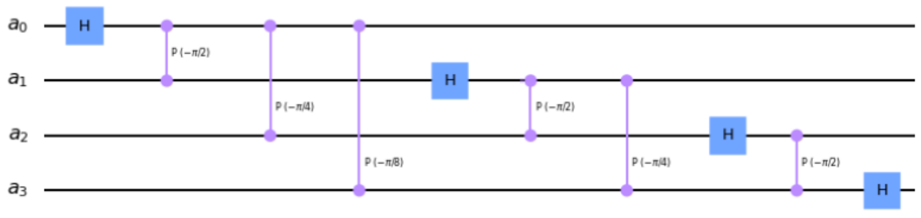


Figure 2.8: Inverse QFT example on number 0001_2

```python
'''
qc: input quantum circuit
reg: input register to execute QFT
n: n-th qbit to apply hadamard and phase rotation
pie: pie number
'''
def inverseQFT(qc, reg, n, pie):
    # Executes the inverse QFT on a register reg.
    # Apply repeated phase rotations with parameters being pi
    # divided by decreasing powers of two, and then apply a
    # Hadamard gate to the nth qubit of the register reg.

    for i in range(n):
        #cp(theta, control_qubit, target_qubit[, ...])
        qc.cp(-1*pie/float(2**(n-i)), reg[i], reg[n])
    qc.h(reg[n])
```

Listing 2.10: QFT: `inverseQFT` function

We also created the `initQubits` function to encode a classical binary number in qbits, by flipping in the inverse order the corresponding qbit. That is because first of all Qiskit's least significant bit has the lowest index (0) and moreover, when we execute the sum, we usually start from the end of the number,the least significant digits, and move on to the most significant digit.

The code is shown in Listing 2.11.

```python
def initQubits(str, qc, reg, n):
    # Flip the corresponding qubit in register if a bit in
    # the string is a 1
    for i in range(n):
        if str[i] == "1":
            qc.x(reg[n-(i+1)])
```

Listing 2.11: QFT: `initQubits` function

We decided to use colors to print the results and the UI, so we coded the function `printResult`. It measures the result, storing it in the `Classi-calRegister cl`. Then, the function executes the actual operation in the quantum simulator, printing the result.

The code is shown in Listing 2.12.

```python
def printResult(first, second, qc,result, cl, n, operator):
    # Measure qubits
    for i in range(n+1):
        qc.measure(result[i], cl[i])

    # Execute using the local simulator
    print(bcolors.BOLD + bcolors.OKCYAN + 'Connecting to
  local simulator...' + bcolors.ENDC)

    # Set chosen backend and execute job
    num_shots = 100 #Setting the number of times to repeat
  measurement
    print(bcolors.BOLD + bcolors.OKCYAN + 'Connect!' +
  bcolors.ENDC)
    print(bcolors.BOLD + bcolors.OKCYAN + f'Running the
  experiment on {num_shots} shots...' + bcolors.ENDC)
    job = execute(qc, backend=Aer.get_backend('qasm_simulator
  '), shots=num_shots)

    # Get results of program
    job_stats = job.result().get_counts()
    for key, value in job_stats.items():
        res = key
        prob = value
    print(bcolors.BOLD + bcolors.OKGREEN + f'\n{first} {
  operator} {second} = {res} with a probability of {prob}%' +
    bcolors.ENDC)
```

Listing 2.12: QFT: `printResult` function

The `sum` function gathers all the previous functions and calls them to perform the QFT addition.

The code is shown in Listing 2.13.

```
def sum(a, b, qc):
    n = len(a)-1
    # Compute the Fourier transform of register a
    for i in range(n+1):
        executeQFT(qc, a, n-i, pie)

    # Add the two numbers by evolving the Fourier transform
    # F(psi(reg_a))> to |F(psi(reg_a+reg_b))>
    for i in range(n+1):
        evolveQFTStateSum(qc, a, b, n-i, pie)

    # Compute the inverse Fourier transform of register a
    for i in range(n+1):
        inverseQFT(qc, a, i, pie)
```

Listing 2.13: QFT: `sum` function

The actual `main` of the program asks for two number to add, converts them in binary, creates the needed quantum registers and then calls the `sum` function.

The code is shown in Listing 2.14.

```
input1 = int(input("Enter a positive integer between 0 and
  2047: "))
input2 = int(input("Enter a positive integer between 0 and
  2047: "))

while (input1 < 0 or input1 > 2047) or (input2 < 0 or input2
  > 2047):
    input1 = int(input("Enter a positive integer between 0
  and 2047: "))
    input2 = int(input("Enter a positive integer between 0
  and 2047: "))

first = '{0:{fill}3b}'.format(input1, fill='0')
second = '{0:{fill}3b}'.format(input2, fill='0')

l1 = len(first)
l2 = len(second)

# Making sure that 'first' and 'second' are of the same
```

```python
# length by padding the smaller string with zeros
if l2>l1:
    first,second = second, first
    l2, l1 = l1, l2
second = ("0")*(l1-l2) + second
n = l1

# Add a qbit to 'a' and 'b' in case of overflowing results
# (the result is only read on 'a', but since 'a' and 'b'
# should have the same lenght, we also add a qbit to 'b')
a = QuantumRegister(n+1, "a")
b = QuantumRegister(n+1, "b")
cl = ClassicalRegister(n+1, "cl")

qc = QuantumCircuit(a, b, cl, name="qc")

# Flip the corresponding qubit in register a if a bit in the
# string first is a 1
initQubits(first, qc, a, n)

# Flip the corresponding qubit in register b if b bit in the
# string second is a 1
initQubits(second, qc, b, n)

addition.sum(a,b,qc)
printResult(first, second, qc,a, cl, n, operator)
```

Listing 2.14: QFT: `main` program

An example circuit of the QFT addition is depicted in Figure 2.10. It sums $1_{10} + 2_{10} = 0001_2 + 0010_2 = 3_{10} = 0011_2$.

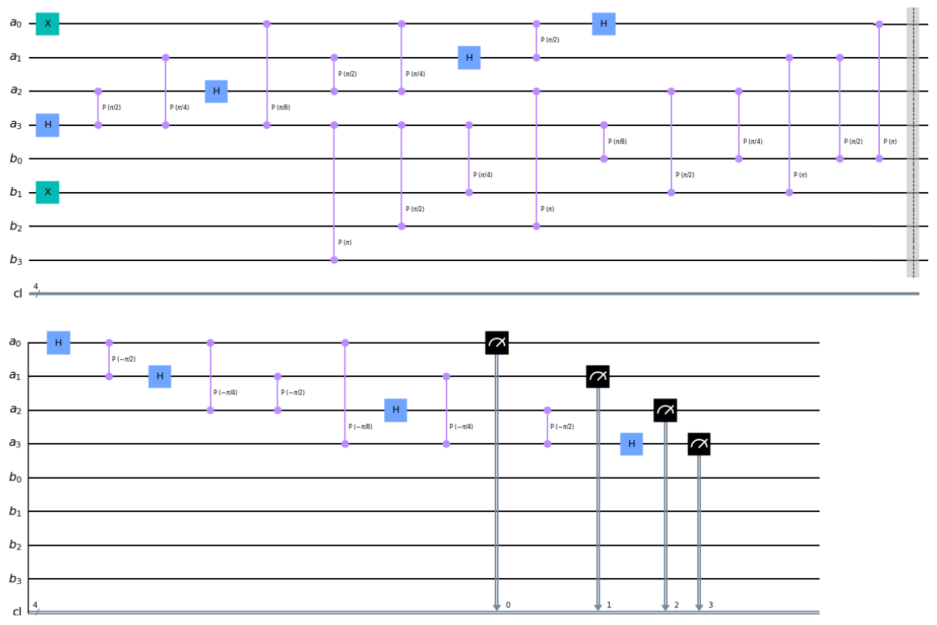
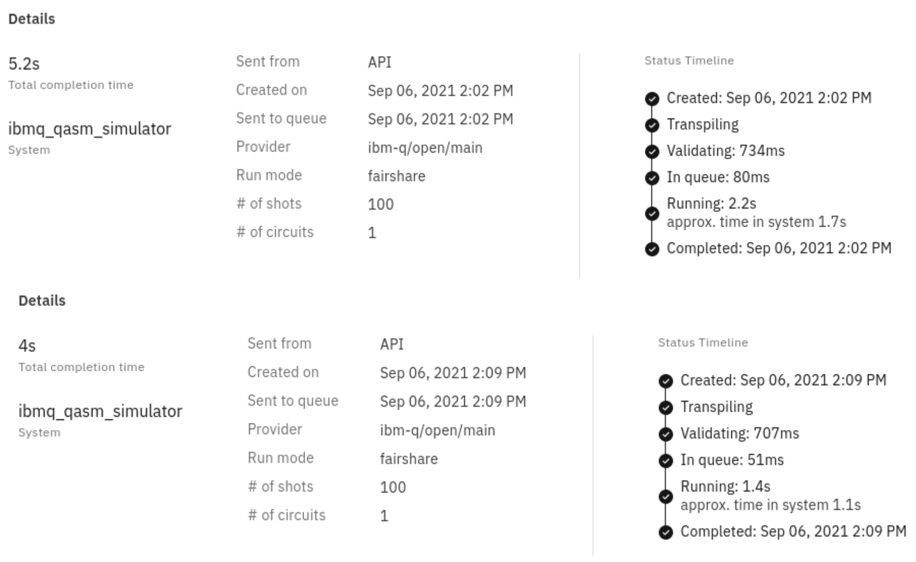


Figure 2.9: QFT: sum circuit of numbers $1_{10} + 2_{10} = 0001_2 + 0010_2 = 3_{10} = 0011_2$

We now provide a comparison in terms of execution time on IBM's offline simulator[3] of the execution of the $(1_{10} + 2_{10})$ addition operation via implementation with Toffoli and QFT.

As we can see in Figure 2.10, the implementation using QFT is faster in computing the result, moreover it allows us to use many less qubits to reach the final result. In fact looking at the circuit needed to reach the result by implementation using Toffoli, shown in figure 2.11, we note that several qubits are used as support, which does not happen using the QFT.

We have performed several tests on different addends and the implementation via QFT has always been faster than the implementation via Toffoli. For these reasons, the next operations will be implemented exclusively via QFT.



Details

5.2s
Total completion time

ibmq_qasm_simulator
System

Sent from	API
Created on	Sep 06, 2021 2:02 PM
Sent to queue	Sep 06, 2021 2:02 PM
Provider	ibm-q/open/main
Run mode	fairshare
# of shots	100
# of circuits	1

Status Timeline

- Created: Sep 06, 2021 2:02 PM
- Transpiling
- Validating: 734ms
- In queue: 80ms
- Running: 2.2s
  approx. time in system 1.7s
- Completed: Sep 06, 2021 2:02 PM

Details

4s
Total completion time

ibmq_qasm_simulator
System

Sent from	API
Created on	Sep 06, 2021 2:09 PM
Sent to queue	Sep 06, 2021 2:09 PM
Provider	ibm-q/open/main
Run mode	fairshare
# of shots	100
# of circuits	1

Status Timeline

- Created: Sep 06, 2021 2:09 PM
- Transpiling
- Validating: 707ms
- In queue: 51ms
- Running: 1.4s
  approx. time in system 1.1s
- Completed: Sep 06, 2021 2:09 PM

Figure 2.10: Comparison between run time execution of addition operation using Toffoli and QFT implementation.
The first figure depicts the run time execution for Toffoli implementation, the second one for QFT.

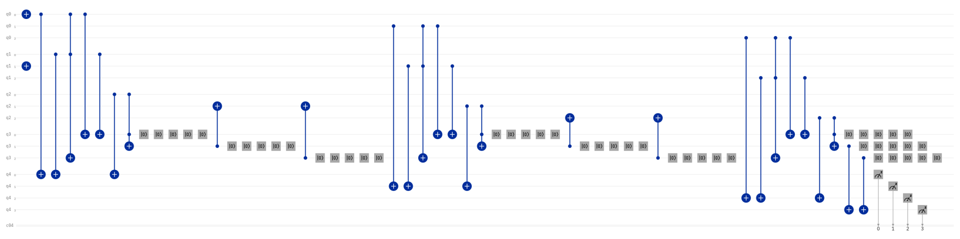


Figure 2.11: Toffoli: sum circuit of numbers $1_{10} + 2_{10} = 0001_2 + 0010_2 = 3_{10} = 0011_2$.

2.2 Quantum Fourier Transform Subtraction

If we think of how subtraction between binary numbers occurs, then in classical computation, we know that we can get the result using the "one complement" of the addend, add one, then make the sum between the result obtained and the first original number and finally remove the first digit of the sum. If we do not have an extra digit, we are trying to subtract a larger number from a smaller one.
That is, if we make a simple example we will have:

$$\begin{aligned} 101_2 - 011_2 &= 101_2 + 100_2 \\ &= 101_2 + (100_2 + 001_2) \\ &= 101_2 + 101_2 \\ &= 1010_2 \rightarrow \cancel{1}010_2 \\ &= 010_2 \end{aligned}$$

In fact $5_{10} - 3_{10} = 2_{10}$ which is exactly 010_2.

Once again using the QFT allows us to skip numerous steps and save qubits. The algorithm we follow for subtraction is essentially identical to that followed for addition, but instead of performing positive rotations in the function `evolveQFTStateSum` which, as mentioned before, evolves the previous encoded state $|F(\psi(reg_a))\rangle$ of the first number to $|F(\psi(reg_a + reg_b))\rangle$, we perform negative rotations.

The new function `evolveQFTStateSub` is shown in Listing 2.15.

```python
'''
qc: input quantum circuit
reg_a: first input register to execute QFT
reg_b: second input register to execute QFT
n: n-th qbit to apply hadamard and phase rotation
pie: pie number
'''
def evolveQFTStateSub(qc, reg_a, reg_b, n, pie):
    # Evolves the state |F(  (reg_a))> to |F(  (reg_a-reg_b))>
    # using the quantum Fourier transform conditioned on the
    # qubits of the reg_b.
    # Apply repeated phase rotations with parameters being pi
    # divided by increasing powers of two.

    l = len(reg_b)
    for i in range(n+1):
        if (n - i) > l - 1:
            pass
        else:
            qc.cp(-1*pie/float(2**(i)), reg_b[n-i], reg_a[n])
```

Listing 2.15: QFT: `evolveQFTStateSub` function

Intuitively since we want to perform the subtraction, first we convert the two numbers into their respective Fourier bases and then we perform as many negative rotations on the Z axis as the number encoded in the B register.

The `sub` function is shown in Listing 2.16.

```
def sub(a, b, qc):
    n = len(a)

    # Compute the Fourier transform of register a
    for i in range(0, n):
        executeQFT(qc, a, n-(i+1), pie)
    # Add the two numbers by evolving the Fourier transform
    # F(psi(reg_a))> to |F(psi(reg_a-reg_b))>
    for i in range(0, n):
        evolveQFTStateSub(qc, a, b, n-(i+1), pie)
    # Compute the inverse Fourier transform of register a
    for i in range(0, n):
        inverseQFT(qc, a, i, pie)
```

Listing 2.16: QFT: `sub` function

We decided to skip the `main` function since it is very similar to the `main` used in the sum. If you are interested in running it refer to the `deskCalculator.ipynb` file or the examples in the readme of the above cited repo. It executes the same operations but calling the `sub` method.

An example circuit of the QFT subtraction is depicted in Figure 2.12. It subtracts $5_{10} - 3_{10} = 0101_2 - 0011_2 = 2_{10} = 0010_2$.

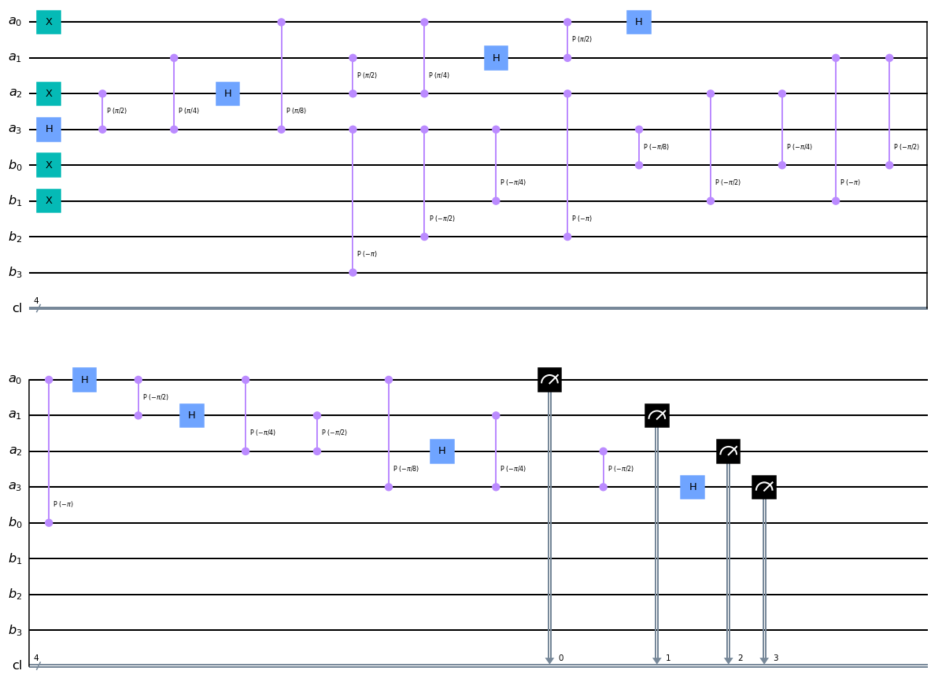


Figure 2.12: QFT: subtract circuit of numbers $5_{10} - 3_{10} = 0101_2 - 0011_2 = 2_{10} = 0010_2$

2.3 Quantum Fourier Transform Multiplication

Even in the decimal numeral system, the product between `a` and `b` can be done by adding the first factor, `a`, as many times as `b` indicates.
Since this property is independent from the numeral system, we use it to multiply numbers in binary.

Let us make an example, by multiplying $5_{10} \cdot 4_{10}$:

$$\begin{aligned} 5_{10} \cdot 4_{10} &= 0_{10} + 5_{10} = 5_{10} && \text{first loop} \\ &= 5_{10} + 5_{10} = 10_{10} && \text{second loop} \\ &= 10_{10} + 5_{10} = 15_{10} && \text{third loop} \\ &= 15_{10} + 5_{10} = 20_{10} && \text{fourth loop} \end{aligned}$$

where 20_{10} is the result of the multiplication.

To prove that this works even when using binary numbers, see the following example:

$$\begin{aligned} 5_{10} \cdot 4_{10} &= 101_2 \cdot 100_2 \\ &= 0_2 + 101_2 = 101_2 && \text{first loop} \\ &= 101_2 + 101_2 = 1010_2 && \text{second loop} \\ &= 1010_2 + 101_2 = 1111_2 && \text{third loop} \\ &= 1111_2 + 101_2 = 10100_2 && \text{fourth loop} \end{aligned}$$

where 10100_2 is 20_{10} in decimal.

Our multiplication function `multiply` is shown in Listing 2.17.

```python
def multiply(a, secondDec, result, qc):

    n = len(a) -1
    # Compute the Fourier transform of register 'result'
    for i in range(n+1):
        executeQFT(qc, result, n-i, pie)

    # Add the two numbers by evolving the Fourier transform
    # F(psi(reg_a))> to |F(psi((second * reg_a))>, where we
    # loop on the sum as many times as 'second' says doing
    # incremental sums
    for j in range(secondDec):
        for i in range(n+1):
            evolveQFTStateSum(qc, result, a, n-i, pie)

    # Compute the inverse Fourier transform of register a
    for i in range(n+1):
        inverseQFT(qc, result, i, pie)

```

Listing 2.17: QFT: `multiply` function

The multiplication is done by performing `b` times the addition of `a`.

We skip the actual `main`, since it is similar to the ones for the sum and subtraction.

However, at the beginning of the multiplication, we only encode `a` in qubits and since the value of `b` is used only to know how many times we need to repeat the addition of `a`, we decided not to transform it in a quantum register but to keep it as a normal integer.

2.4 Quantum Fourier Transform Division

To implement the last elementary operation we need two quantum registers to encode dividend and divisor and an additional quantum register to use as an accumulator. Let us see the process to perform integer division in the following example. Suppose we want to divide the number $8_{10} = 1000_2$ by $2_{10} = 010_2$, what we are going to do is continue to subtract from the dividend the divisor until the subtraction, implemented via QFT, brings the qubit into a state that would equate to a number (in binary notation) larger than the dividend itself. For example:

$$
\begin{aligned}
&accumulator = 0\\
&\qquad 8-2 = 01000-00010 = 6\ (00110) \qquad \text{accumulator} += 1\\
&\qquad 6-2 = 00110-00010 = 4\ (00100) \qquad \text{accumulator} += 1\\
&\qquad 4-2 = 00100-00010 = 2\ (00010) \qquad \text{accumulator} += 1\\
&\qquad 2-2 = 00010-00010 = 0\ (00000) \qquad \text{accumulator} += 1\\
&\qquad 0-2 = 11110 \qquad \text{accumulator} += 1
\end{aligned}
$$

At this point we exit from the loop and if we check the value of the accumulator it will be 5, but we have subtracted too much, so we subtract 1 to obtain 4 which is the correct result. We notice that intuitively what we are doing is subtracting iteratively until the rotations on the z axis in the Fourier base do not lead to a wrong result, in fact as we can see if we try to subtract 2 to 0 we obtain an underflow, that is with a rotation from 00000_2

we go to 11111_2 and with another rotation we arrive at 11110_2.
We present below the code 2.18 to perform the division with the procedure just described.

```
1  def div(dividend, divisor, accumulator,c_dividend, circ,
      cl_index):
2      d = QuantumRegister(1)
3      circ.add_register(d)
4      circ.x(d[0])
5  
6      c_dividend_str = '0'
7  
8      while c_dividend_str[0] == '0':
9          subtraction.sub(dividend, divisor, circ)
10         addition.sum(accumulator, d, circ)
11 
12         for i in range(len(dividend)):
13             circ.measure(dividend[i], c_dividend[i])
14 
15         result = execute(circ, backend=Aer.get_backend('
      qasm_simulator'),
16                               shots=10).result()
17 
18         counts = result.get_counts("qc")
19         #print(counts)
20         c_dividend_str = list(counts.keys())[0] #.split()[0]
21 
22 
23     subtraction.sub(accumulator, d, circ)
24 
```

Listing 2.18: QFT: `division` function

Before starting with the while loop we create a qubit `d` and set it to 1 which will be used as a constant to add to the accumulator at each iteration. Moreover, to know when to exit the cycle we check the first most significant digit of the result of the subtraction between dividend and divisor, because if at the beginning the first digit was not 1 surely it will never become 1 since we are performing subtractions. To be sure to enter the cycle at the beginning we force the first digit to be 0 with the command on line 6. We note that given Heisenberg's uncertainty principle, we cannot know the value of the state of a qubit until we measure it. For this reason, in order to know the value of the subtraction between the dividend and divisor, it is necessary to perform measurements at each iteration. The results of the measurement are then used to retrieve the first most significant digit to know whether or not to continue with the cycle.

Bibliography

[1] Qiskit, "Qiskit documentation." [Online]. Available: https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html

[2] A. Cherkas and S. Chivilikhin, "Quantum adder of classical numbers," in *Journal of Physics: Conference Series*, vol. 735, no. 1. IOP Publishing, 2016, p. 012083.

[3] IBM, "ibmq_qasm_simulator." [Online]. Available: https://quantum-computing.ibm.com/services?services=simulators&system=ibmq_qasm_simulator