POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Arithmetic circuits for quantum computing: a software library

Relatori:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Ph.D. Giovanna Turvani

Candidato:
Lorenzo Raggi

Ottobre 2020

Summary

Nowadays, quantum computers are capturing the attention of researchers because of their potential capability to solve problems otherwise not easily solvable with classical computers. In parallel to the constant research aimed at solving problems related to the physical implementation of such computers, there is also another branch of research trying to develop quantum algorithms, many of which need to perform arithmetic operations.
In this thesis, the development of a software library of quantum arithmetic circuits - capable of interfacing with Quantum Computing frameworks such as Qiskit, Cirq and T-ket - is presented. With the aim of implementing this software library, the basics of quantum computation were first studied (chapter 1) and subsequently an extensive bibliographic research was carried out, aimed at identifying the best performing arithmetic circuits currently available (chapter 2). After determining the circuits to be implemented, the software development has begun using the Python language (chapter 3). During the development of the library, with the aim of making simpler future development, code's modularity - by organizing the general package in many subpackages - and readability - by respecting the guidelines proposed in the PEP8 style guide - were looked for. In addition to what has already been said, a detailed documentation has been produced using Sphinx. Once the development of the library was completed, the functionality of each circuit was tested with simulations on classical computers; moreover, some components were tested on real free-accessible quantum computers, in order to characterize their behavior (chapter 4).
Even though the results obtained on real hardware proved that many of the circuits implemented in this library will be reliably executed by more sophisticated and performing quantum computers, this library can be currently employed in an effective way for simulating quantum circuits on classical computers.

Table of contents

List of tables

List of figures

Chapter 1

Fundamental concepts

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."
Alan Turing

A quantum computer is a system that exploits quantum mechanics principles in order to run algorithms computationally cheaper than their corresponding to be run on a classical computer. What makes the race for quantum supremacy beautiful and compelling are the applications that are supposed to be achievable with the advent of quantum computers. The fields of application of Quantum Computing may involve sectors such as medicine, finance, logistics, telecommunications, cryptography and many more. The innovative charge linked to the world of Quantum Computing pushes giants such as Google, IBM and many others to invest a lot of money and brilliant minds in exploring this partially unknown field.

The aim of this chapter is to give an overview about the basic concepts necessary to understand what will be shown in the next chapters. Starting from the essential mathematical tools, the focus will be the fundamentals of quantum computation.
It was decided to avoid a strict formalism in favor of a simpler and clearer presentation of the various concepts, so that a reader who knows nothing about Quantum Computing will be able to understand the core of the thesis.

1.1 Linear Algebra Recalls

In this first section all the mathematical tools needed to understand the content of the thesis will be listed briefly.

Matrices: transpose, conjugate, adjoint

- **Transpose**: the transposed matrix of a matrix is the one obtained by exchanging its rows with columns.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \longrightarrow A^T = \begin{bmatrix} a_{11} & \dots & a_{m1} \\ a_{12} & \ddots & \vdots \\ \vdots & \dots & a_{mn} \end{bmatrix}$$

- **Conjugate**: the conjugate matrix of a matrix is obtained by performing the complex conjugate of each element.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \longrightarrow \overline{A} = \begin{bmatrix} \overline{a_{11}} & \overline{a_{12}} & \dots \\ \vdots & \ddots & \vdots \\ \overline{a_{m1}} & \dots & \overline{a_{mn}} \end{bmatrix}$$

- **Adjoint (or Dagger)**: the adjoint of a matrix is obtained by conjugating and transposing the matrix.

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \longrightarrow A^\dagger = (\overline{A})^T = \begin{bmatrix} \overline{a_{11}} & \dots & \overline{a_{m1}} \\ \overline{a_{12}} & \ddots & \vdots \\ \vdots & \dots & \overline{a_{mn}} \end{bmatrix}$$

It is noted that, in general, these three operations are idempotent and respect both addition and scalar multiplication. The only exception is the scalar multiplication for the case of Adjoint:

$$(c \cdot A)^\dagger = \overline{c} \cdot A^\dagger$$

Dirac notation In Quantum Mechanics the Dirac notation is widely used to indicate vectors:

- **ket**: $|\Psi\rangle = [c_0,c_1,...,c_{n-1}]^T$
- **bra**: $\langle\Psi| = |\Psi\rangle^\dagger = [\overline{c_0},\overline{c_1},...,\overline{c_{n-1}}]$

Inner product The inner product is a product between two vectors, *e.g* $|\Psi\rangle$ and $|\Psi'\rangle$, and it is defined as follows:

$$\langle\Psi'|\Psi\rangle = \begin{bmatrix} \overline{c'_0} & \overline{c'_1} & ... & \overline{c'_{n-1}} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \overline{c'_0}\cdot c_0 + \overline{c'_1}\cdot c_1 + ...$$

Tensor product Assume to have two matrices as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix}$$

The tensor product between A and B can be defined as:

$$A \otimes B = \begin{bmatrix} a_{11}\begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} & a_{12}\begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} & \dots & \dots & a_{1n}\begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ a_{m1}\begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} & \dots & \dots & \dots & a_{mn}\begin{bmatrix} b_{11} & b_{12} & \dots \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{bmatrix} \end{bmatrix}$$

Equivalently:

$$A\otimes B = \begin{bmatrix} a_{11}\cdot b_{11} & a_{11}\cdot b_{12} & \dots & \dots & a_{12}\cdot b_{11} & a_{12}\cdot b_{12} & \dots & \dots & \dots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots & \ddots \\ a_{11}\cdot b_{m1} & \dots & \dots & a_{11}\cdot b_{mn} & a_{12}\cdot b_{m1} & \dots & \dots & a_{12}\cdot b_{mn} & \dots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots & \ddots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots & \ddots \\ a_{m1}\cdot b_{11} & a_{m1}\cdot b_{12} & \dots & \dots & a_{m2}\cdot b_{11} & a_{m2}\cdot b_{12} & \dots & \dots & \dots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots & \ddots \\ a_{m1}\cdot b_{m1} & \dots & \dots & a_{m1}\cdot b_{mn} & a_{m2}\cdot b_{m1} & \dots & \dots & a_{m2}\cdot b_{mn} & \dots \end{bmatrix}$$

Unitary matrix A complex square matrix U is called *unitary* if its adjoint and its inverse coincide, *i.e.*:

$$U^\dagger U = UU^\dagger = I \tag{1.1}$$

where I is the identity matrix, *i.e.* a diagonal matrix whose non-zero elements are equal to 1:

$$I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{bmatrix}$$

1.2 Qubit

Qubit stays for "Quantum Bit" and it is the fundamental element when speaking about quantum information. Particularly, even if in reality a qubit can be encoded onto different physical quantities (*e.g.* spins, quantized electrical quantities, *etc.*), there is a mathematical representation that allows to consider qubits regardless their physical implementation. A single qubit can be used to represent a two-state quantum-mechanical system, and it can be expressed as shown in Equation 1.2.

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \qquad \textit{where} \quad \alpha,\beta \in \mathbb{C} \tag{1.2}$$

If classical bits are expressed by using vectors, it becomes simpler to explain the key difference among bits and qubits: while a standard bit can assume only two values - either 0 or 1 - a qubit can ideally assume infinite values, which are linear combinations of the two basis states. More precisely, the state of a qubit is fully described by the two complex coefficients α and β, which can assume any complex value with magnitude square between 0 and 1 (it will be clarified later). It results that the state of a single qubit can be the **superposition** of the two basis states $|0\rangle$ and $|1\rangle$.

CLASSICAL COMPUTING

A bit can assume a value which is either 0 or 1 (suppose to call them $|0\rangle$ and $|1\rangle$ even if they do not represent quantum states).

$$|0\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |1\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

It is noted that in this notation if one value is equal to 1, the other is 0. The row labels represent the binary values. The column vector may represent either the binary value 0 or 1, depending on which entry is set to 1.

QUANTUM COMPUTING

A qubit, in general, can be expressed as a superposition of the orthonormal basis states $|0\rangle$ and $|1\rangle$:

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

1.2.1 Measurement

By looking at the Equation 1.2, it is possible to imagine that by measuring a qubit an infinite number of outcomes can be obtained. This is false.
When measuring a qubit, it is stated by Quantum Mechanics postulates (detailed later) that the result of a measurement can be:

- $|0\rangle$ with probability $|\alpha|^2$
- $|1\rangle$ with probability $|\beta|^2$

These are the only two kinds of outputs one can expect from a measurement. Thus, it is not possible - with a single measurement - to measure all the superposition states a qubit can assume, but only the basis states $|0\rangle$ and $|1\rangle$ with a given probability.
Since the measuring operation has to do with probability, it is as shown in Equation 1.3.

$$|\alpha|^2 + |\beta|^2 = 1 \tag{1.3}$$

The Equaton 1.3 also clarifies the contraints on the magnitude square of α and β: they are equal to probabilities whose sum must be necessarily equal to 1. For these reasons, α and β are called **probability amplitudes**. There is something more about the measuring operation: if a qubit which is in a certain superposition of states is measured - in addition to the fact that the outcome will be one of the two basis states - it also results that the qubit state will collapse into the measured state. In this way, if further measurements on the same qubit are performed, the same result with probability 1 will always be obtained. Thus, the measurement operation is *irreversible*. The following example can help clarifying this concept.
Assume to have a qubit which is in a generic state $|\Psi\rangle$:

$$|\Psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Assuming that the outcome of the first measurement will be $|1\rangle$, then the state $|\Psi'\rangle$ of the qubit after the measurement will be:

$$|\Psi'\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Performing further measurements on this qubit, the result will be always $|1\rangle$.

1.2.2 Superposition

From Equation 1.2 it is known that the state of a qubit can be a superposition of the two basis states, while Equation 1.3 tells that the sum of the square of the magnitudes of the two coefficients α and β must be equal to 1.
This means that - if either α or β is equal to 0 - the qubit state is something similar to the standard bits.

$$\beta = 0 \longrightarrow |0\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \alpha = 0 \longrightarrow |1\rangle = \begin{array}{c} \mathbf{0} \\ \mathbf{1} \end{array} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Instead, if both coefficients are different from zero, the qubit becomes something more powerful than a standard bit, since it becomes an entity able to handle "more information".
The concept of superposition can be applied to an arbitrary number of qubits, but before doing that, it is necessary to see how to assemble quantum systems in order to understand how to represent a system composed by an arbitrary number of qubits.

Assembling Quantum Systems

To assemble quantum systems, it is necessary to perform the *tensor product* between them.
For example, having two different two-level quantum systems,

$$|\Psi\rangle = \begin{bmatrix} a \\ b \end{bmatrix} \qquad |\phi\rangle = \begin{bmatrix} c \\ d \end{bmatrix},$$

it is necessary to perform their tensor product in order to “mingle” them. The state of the overall system can therefore be expressed as:

$$|\Psi\rangle \otimes |\phi\rangle = \begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a \cdot c \\ a \cdot d \\ b \cdot c \\ b \cdot d \end{bmatrix}$$

Returning on the topic of superposition, having more than one quibit determines that:

$$2\ qubits \quad \longrightarrow \quad |\Psi\rangle = \begin{array}{c} \scriptstyle 00 \\ \scriptstyle 01 \\ \scriptstyle 10 \\ \scriptstyle 11 \end{array} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \qquad where \quad |a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$$

In this case, it is possible to observe a superposition of four different basis states, *i.e.* four complex numbers are needed in order to fully describe the superposition of states.
See now the general case of N qubits:

$$N\ qubits \quad \longrightarrow \quad |\Psi\rangle = \begin{array}{c} \scriptstyle 00..00 \\ \scriptstyle 00..01 \\ \scriptstyle ...... \\ \scriptstyle ...... \\ \scriptstyle ...... \\ \scriptstyle 11..10 \\ \scriptstyle 11..11 \end{array} \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_i \\ \dots \\ a_{N-1} \\ a_N \end{bmatrix} \qquad where \quad \sum_{i=1}^{N} |a_i|^2 = 1$$

In this case, it has been obtained a superposition of 2^N basis states, so 2^N complex numbers are needed to describe the system.
It is noted that there are several equivalent ways to indicate a couple of qubits (or

in general N qubits):

$$|Q1\rangle \otimes |Q2\rangle\,, \quad |Q1 \otimes Q2\rangle\,, \quad |Q1\rangle\,|Q2\rangle\,, \quad |Q1{,}Q2\rangle\,, \quad |Q1Q2\rangle$$

Now put together what is known about **measurement** and **superposition**.

As made clear right above, 2^N probability amplitudes are needed to fully describe an N-qubit quantum state. Thus, in order to exploit this kind of quantum information there is the need to develop algorithms which might be able of taking advantage of superposition of different basis states but that, in the end, leave the system in a state such that it is possible to measure one of the basis states - corresponding to the problem's solution - with a very high probability.

Intuitive example

In order to better understand the advantage that superposition can provide, an intuitive example (not formally correct) is reported in the following.
Assume to have four items, and to associate to each of them a binary code as shown in Figure 1.1.

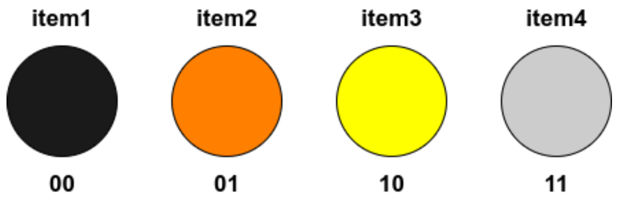


Figure 1.1: Example: items.

The aim is to detect the yellow item.
Assume to have a function f which is able to find the yellow item, *i.e.* the one labelled with 10. To do that, the function receives as input a four-element vector column (one entry for each item) and gives as output the same vector column but with a minus sign on the third element, *i.e.* the one corresponding to the yellow item.

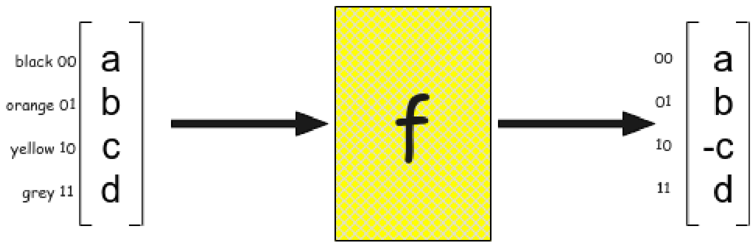


Figure 1.2: Example: function f able to “find” the yellow item

By using this setup, the same problem is solved with two different approaches: classical approach and quantum approach.

Classical Approach When using a traditional approach, it is necessary to analyze one item at the time in order to find the right one. In the worst case scenario - *i.e.* the case in which the correct item is the last to be analyzed - four iterations are required.

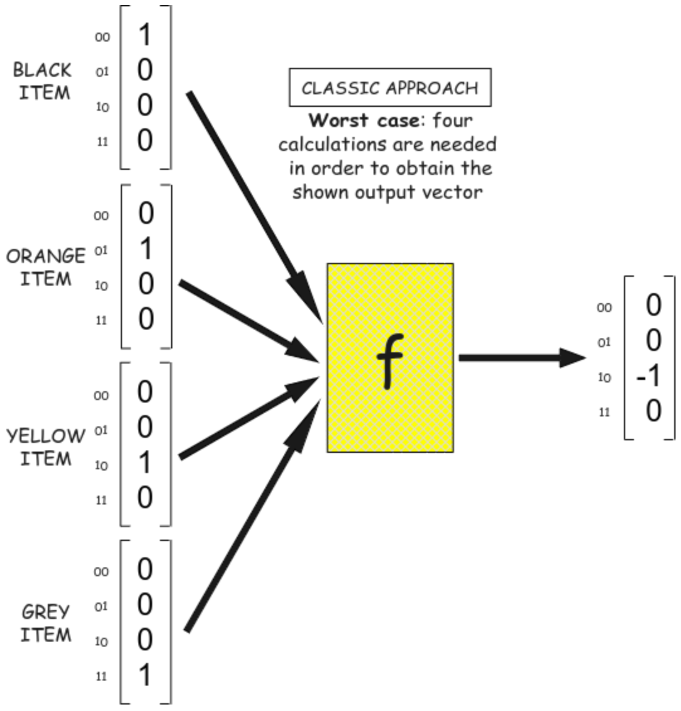

BLACK ITEM
ORANGE ITEM
YELLOW ITEM
GREY ITEM
CLASSIC APPROACH
Worst case: four calculations are needed in order to obtain the shown output vector
f
00 01 10 11
0 0 -1 0

Figure 1.3: Example: classical approach

Quantum Approach When following the quantum approach, it is possible to create a *superposition* of the inputs and proceed as shown in Figure 1.4.

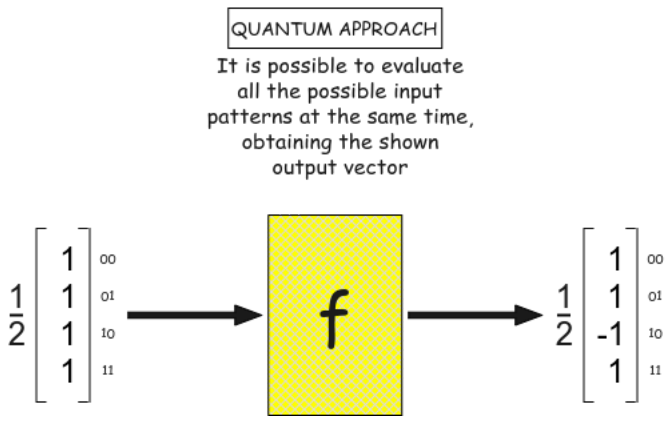

QUANTUM APPROACH
It is possible to evaluate all the possible input patterns at the same time, obtaining the shown output vector
f
1/2 [1 1 1 1] 00 01 10 11
1/2 [1 1 -1 1] 00 01 10 11

Figure 1.4: Example: quantum approach - first step

In this way it is possible to evaluate all possible input patterns at the same time, obtaining the output shown in Figure 1.4, in which the wanted item - the yellow one - is marked with a minus sign.
At this point it should be clear what people mean when - referring to Quantum Computing - talk about "concurrent evaluation". Now it should be explained why the quantum approach proves to be favourable even if the measurement process is probabilistic. Two problems must be still overcome in order to have some quantum advantage:

1. *at least* 2^2 measurements should be performed in order to find the correct result;

2. the measurement operation is not reversible.

It is clear that something more is necessary. What actually quantum algorithms do is to perform a sort of "boost" which significantly increases the probability of measuring the solution of the problem, as shown in Figure 1.5, where the "boost" tries to force a final state close to $|10\rangle$.[1]

[1]This example is actually a simplified (and not mathematically correct) version of what is called "Grover's algorithm".

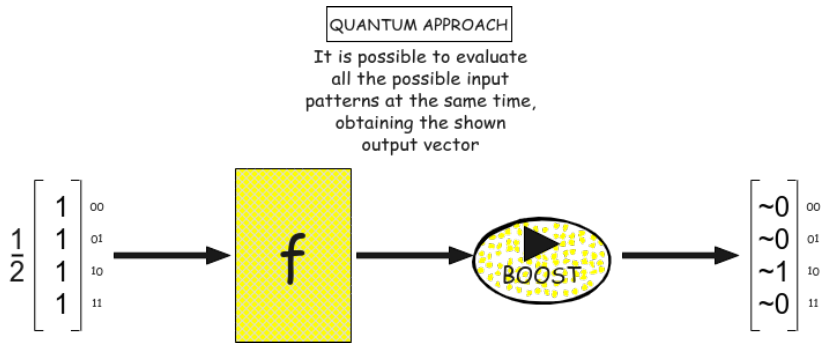

QUANTUM APPROACH
It is possible to evaluate all the possible input patterns at the same time, obtaining the shown output vector
1/2
1 1 1 1
00 01 10 11
f
BOOST
~0 ~0 ~1 ~0
00 01 10 11

Figure 1.5: Example: quantum approach - complete solution

1.2.3 Entanglement

Entanglement is a form of quantum mechanical correlation which tells that - in some cases - the state of a single quantum system could depend “instantly” on the state of other quantum systems. To say it in other words, entanglement tells that not always an assembled complex system can be understood in terms of its constituents. According to what has just been stated, two important definitions can be introduced([3]):

- States separable into the tensor product of states from the constituent subsystems are referred to as **separable** states.
- States that are not separable are called **entangled** states.

A very famous example of entangled states are the EPR pairs (or Bell states):

$$|\Psi_1\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad |\Psi_2\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}$$

$$|\Psi_3\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \qquad |\Psi_4\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

Take as example $|\Psi_1\rangle$. If the first qubit is measured, and the result is $|0\rangle$, then also the measurement of the second qubit will give $|0\rangle$ as a result. In general, for these states, it holds that measured one of the two qubits, also the state of the other qubit is well determined.

1.2.4 Quantum Mechanics Postulates

All things that have been said are nothing more than an intuitive and partial explanation of Quantum Mechanics postulates. To have a broader and more formal view of the topic, Quantum Mechanics postulates have to get involved.
It is noted that there are several equivalent versions of these postulates; here it is decided to follow the one proposed by [6].

Postulate 1 *Any isolated physical system has its own state space, which is a Hilbert space*[a]*. The system can be fully described by a unit vector in the system's state space, i.e. the state vector of the system.*

[a]A complex vector space equipped with an inner product.

Comment This postulate justifies Equation 1.2, according to which the state vector of a qubit can be expressed as a linear combination of the elements $|0\rangle$ and $|1\rangle$, which are an orthonormal basis for the state space.
The fact that the state vector $|\Psi\rangle$ is unitary warrants Equation 1.3. In fact, it can be demonstrated that the expression

$$\langle\Psi|\Psi\rangle = 1 \tag{1.4}$$

is equivalent to Equation 1.3.
Equation 1.4 is also known as *normalization condition* for state vectors.

Postulate 2 *Unitary transformations describe the evolution of a closed quantum system. Thus, a closed quantum system observed in different instants of time - t_1 and t_2 - will be characterized by two different states - $|\Psi\rangle$ and $|\Psi'\rangle$ - in relation with each other through the unitary operator U*[a]*:*

$$|\Psi'\rangle = U\,|\Psi\rangle$$

[a]It depends on the considered time instants.

Comment This postulate refers to something not explained so far. Particularly, it states that if a quantum system is closed, its evolution can be described through unitary operators (see Equation 1.1). In reality it is impossible to have a completely isolated system, since there will always be a kind of interaction between systems close to each others. However, there are some cases in which it is possible to consider a system closed with good approximation.
By looking at unitary operators in this way:

$$|\Psi(t+1)\rangle = U\,|\Psi(t)\rangle$$

and by keeping in mind Equation 1.1, it should be clear that this kind of transformations are reversible, *i.e.* if a unitary operator U is applied from t to t+1, it is always possible to restore the state at the instant of time t by applying a unitary operator $U^\dagger$ from t+1 to t+2.[2]

Postulate 2' *The evolution in time of the state of a closed quantum system is defined by the Schrödinger equation,*

$$i\hbar\frac{\partial\,|\Psi\rangle}{\partial t} = \hat{H}(t)\,|\Psi\rangle\,, \tag{1.5}$$

where $\hbar$ is the reduced Planck's constant and $\hat{H}$ is a fixed Hermitian operator known as the Hamiltonian of the closed system.[a]

[a]Sometimes $\hbar$ is absorbed into $\hat{H}$.

Comment This postulate introduces the concept of the Hamiltonian, which is a mathematical operator that allows to know the total energy[3] of a system. The Hamiltonian is a tool for deriving two important features of a quantum system. First, the fact that it is contained in the Schrödinger equation - whose solution allows to know the evolution of a system in the time domain - is a concept that underlies

[2]In the following a time step i (from $t+i$ to $t+i+\delta t$) will be called "time click" (as in [3]).
[3]Given by the sum of kinetic and potential energies.

the implementation of quantum gates[4]. In fact, if an additional contribution to the internal Hamiltonian of the system is added in a finite time window, it is possible to obtain a temporal unitary evolution of the system corresponding to the quantum gate to be implemented. This can be proved analytically as follows:

$$\begin{aligned}
& i\hbar\frac{\partial\left|\Psi\right\rangle}{\partial t} = \hat{H}(t)\left|\Psi\right\rangle \\
& \frac{\partial\left|\Psi\right\rangle}{\left|\Psi\right\rangle} = -\frac{i}{\hbar}\hat{H}(t)\partial t \\
& \ln\left(\frac{\left|\Psi\right\rangle}{\Psi_0}\right) = -\frac{i}{\hbar}\int_0^\tau \hat{H}(t)\partial t \\
& \left|\Psi\right\rangle = e^{-\frac{i}{\hbar}\int_0^\tau \hat{H}(t)\partial t}\cdot\left|\Psi_0\right\rangle \\
& \text{if } \hat{H}(t) = H \text{ then } \left|\Psi\right\rangle = e^{-\frac{i}{\hbar}\cdot H\tau}\cdot\left|\Psi_0\right\rangle \\
& \left|\Psi\right\rangle = U\left|\Psi_0\right\rangle \ \textit{where} \text{ U is a unitary operator}
\end{aligned} \tag{1.6}$$

From Equation 1.6 it can be proved that:

$$\begin{cases} \text{Without external excitations} \longrightarrow U = I \\ \text{With external excitations} \longrightarrow U \neq I \end{cases}$$

When U is different from the identity matrix, it can represent a given quantum gate (R_x, R_y, R_z, CX, *etc.*).

Another important aspect of the Hamiltonian is that it allows to solve the Equation 1.7, which is called stationary[5] Schrödinger equation. The latter is an eigenvalue equation which allows to know what are the energy levels in which a given system can be.

$$H\left|\Psi\right\rangle = \lambda\left|\Psi\right\rangle \tag{1.7}$$

In summary:

- Solving the stationary Schrödinger equation (see Equation 1.7) allows to understand what are the energy levels of a system and to understand what are the states in which the system can be measured.

[4]Quantum gates are explained in Section 1.3.

[5]Stationary means that it does not allow evaluations in the time domain.

- Solving the Schrödinger equation (see Equation 1.5) allows to obtain information about the temporal evolution of the system of interest.

Postulate 3 *Quantum measurements are defined by a multitude of measurement operators M_m. These operators act on the state space of the system being measured. Measurement outcomes that may occur in the observation are referred with the index m. Let $|\Psi\rangle$ be the state of the quantum system right before the measurement then the probability that result m occurs is given by*

$$p(m) = \langle\Psi| M_m^\dagger M_m |\Psi\rangle ,$$

and the state of the system after the measurement is

$$\frac{M_m |\Psi\rangle}{\sqrt{\langle\Psi| M_m^\dagger M_m |\Psi\rangle}}$$

The measurement operators satisfy the completeness equation

$$\sum_m M_m^\dagger M_m = I,$$

expressing the fact that probabilities sum to one:

$$1 = \sum_m p(m) = \sum_m \langle\Psi| M_m^\dagger M_m |\Psi\rangle .$$

Comment This is a formal version of what has been said about measurement. Particularly, by considering Equation 1.2, it has been said that a qubit has $|\alpha|^2$ probability of measuring $|0\rangle$ and $|\beta|^2$ of measuring $|1\rangle$. Moreover, it was explained that - after the measurement - its state collapses in the measured basis state. The goal, now, is to try to demonstrate the rightness of what has been said by applying this postulate.
Measuring a single qubit in the computational basis $\{|0\rangle, |1\rangle\}$ can give two possible outcomes (*i.e.* the element of the orthonormal basis $\{|0\rangle, |1\rangle\}$) defined by the two

(Hermitian) **measurement operators**:

$$\begin{cases} M_0 = |0\rangle\langle 0| & \textit{where} \quad M_0^2 = M_0 \\ M_1 = |1\rangle\langle 1| & \textit{where} \quad M_1^2 = M_1 \end{cases} \tag{1.8}$$

From Equation 1.8 it is possible to see that the completeness equation is satisfied:

$$\sum_{i=0}^{1} M_i^\dagger M_i = M_0^\dagger M_0 + M_1^\dagger M_1 = M_0 + M_1 = I$$

If the generic state

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

is measured, then the probabilities of measuring $|0\rangle$ and $|1\rangle$ are:

$$\begin{cases} p(0) = \langle\Psi| M_0^\dagger M_0 |\Psi\rangle = \langle\Psi| M_0 |\Psi\rangle = |\alpha|^2 \\ p(1) = \langle\Psi| M_1^\dagger M_1 |\Psi\rangle = \langle\Psi| M_1 |\Psi\rangle = |\beta|^2 \end{cases}$$

Therefore the state after measurement in the two cases will be:

$$\begin{cases} \frac{M_0|\Psi\rangle}{|\alpha|} = \frac{\alpha}{|\alpha|}|0\rangle \sim |0\rangle \\ \frac{M_1|\Psi\rangle}{|\beta|} = \frac{\beta}{|\beta|}|1\rangle \sim |1\rangle \end{cases}$$

Postulate 4 *Given a composite physical system, its state space can be described by the tensor product of the state spaces of its components. In addition to that, having n subsystems with the generic system number i prepared in the state $|\Psi_i\rangle$, then the state of the total system is $|\Psi_1\rangle \otimes |\Psi_2\rangle \otimes ... \otimes |\Psi_n\rangle$.*

Comment This is the last postulate, and it is the formal version of what has been said about assembling quantum systems.
In addition to that, this postulate allows to talk about entanglement. Consider for instance the Bell state

$$|I^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

For this state it is not possible to find two single qubit states, $|\psi\rangle, |\phi\rangle$ such that $|I^+\rangle = |\psi\rangle|\phi\rangle$. This is due to the fact that this is an entangled state.

1.2.5 Bloch Sphere

The Bloch sphere has a unitary radius and it is centered in the origin. A vector $\hat{n}$ on the Bloch sphere with a tail at the origin and a head on the surface of the sphere is called "Bloch vector". A Bloch vector allows to graphically represent the state of a qubit.

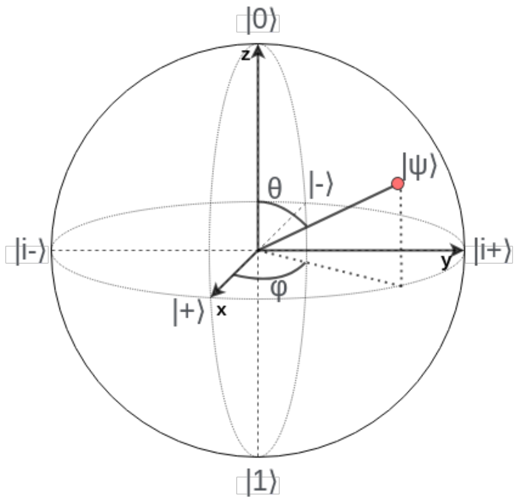


Figure 1.6: Bloch sphere

By looking at the Bloch sphere, it is possible to obtain an equivalent expression for a generic qubit state:

$$|\Psi\rangle \in \mathbb{C}^2 \quad \Longleftrightarrow \quad \hat{n} \in \mathbb{R}^3$$

$$|\Psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \quad \Longleftrightarrow \quad |\Psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{+i\varphi} sin\frac{\theta}{2}|1\rangle \quad where \quad \begin{cases} \theta \in [0;\pi] \\ \varphi \in [0;2\pi] \end{cases} \tag{1.9}$$

It is a common convention to use a real value as coefficient of $|0\rangle$. This can be done since only the relative phase among $|0\rangle$ and $|1\rangle$ assumes a physical meaning. Global phase has no effects on the system's state. Apart from canonical states $|0\rangle$ and $|1\rangle$ - which permit to describe the qubit state with a linear combinations of two vectors lying on the z-axis - other four remarkable states are shown in Figure 1.6 and Table 1.1. These vectors lie along the x and y axes.

Table 1.1: Remarkable states

State	**Equation**
$\|+\rangle$	$\frac{\|0\rangle+\|1\rangle}{\sqrt{2}}$
$\|-\rangle$	$\frac{\|0\rangle-\|1\rangle}{\sqrt{2}}$
$\|i+\rangle$	$\frac{\|0\rangle+i\|1\rangle}{\sqrt{2}}$
$\|i-\rangle$	$\frac{\|0\rangle-i\|1\rangle}{\sqrt{2}}$

1.3 Quantum Gates

Quantum gates are entities represented by unitary matrices that can be used to manipulate the state of one or more qubits by changing the state vector $|\Psi\rangle$, with the normalization condition (see Equation 1.4) continuing to be valid. Another key point about quantum gates is that they must be reversible, *i.e.* when an operator is applied to a given state, it must be always possible to reconstruct the input state starting from the output one.

A N qubit operator can be represented with a $2^N - by - 2^N$ unitary matrix.

Before proceeding with the discussion of the various gates, it has to be noted that in the case of quantum circuits there are two different conventions regarding the order in which the qubits in a circuit have to be read: the "traditional notation" and the "IBM notation". In this chapter, the *traditional notation* - where the top qubit is the most significant one - will be used.

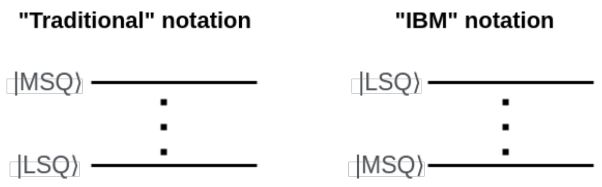


Figure 1.7: Traditional notation and IBM notation

1.3.1 One-Qubit Gates

Pauli X-gate This gate is the "quantum equivalent" of the classical "not" gate. Generally speaking, it is able to flip the $|0\rangle$ state in $|1\rangle$ state (and *vice versa*). By looking at the Bloch sphere, it is possible to interpret the action of this gate in terms of a rotation around the x-axis of π-radians. Because of the effect this gate has on the state of a qubit, it is also called *bit-flip* gate.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

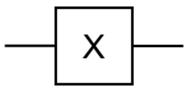


Figure 1.8: X gate symbol

In general:

$$|\Psi'\rangle = X\,|\Psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

It can be observed that the X gate swaps the probability amplitudes of $|0\rangle$ and $|1\rangle$.

Example: This example shows what happens if an X gate is applied on a qubit whose state is $|0\rangle$.

$$|\Psi\rangle = X\,|0\rangle = |1\rangle$$

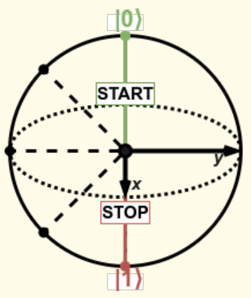


Figure 1.9: Example: X gate

Pauli Z-gate This gate is able to flip the $|+\rangle$ state in $|-\rangle$ state (and *vice versa*). The effect of this gate consists in a rotation around the z-axis of π-radians of a generic state on the Bloch sphere.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

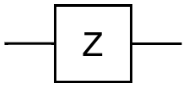


Figure 1.10: Z gate symbol

In general:

$$|\Psi'\rangle = Z\,|\Psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix}$$

As it can be seen from the previous calculation, this gate is able to change the relative phase of a qubit (minus sign on β). For this reason it is also referred as *phase-flip* gate.

Example: This example shows what happens if a Z gate is applied on a qubit whose state is $|+\rangle$.

$$|\Psi\rangle = Z\,|+\rangle = |-\rangle$$

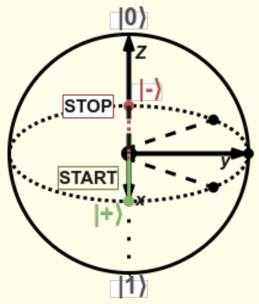


Figure 1.11: Example: Z gate

Pauli Y-gate In general, this gate is able to flip the $|0\rangle$ state in $|1\rangle$ state (and *vice versa*). In this case, the final state has both a different relative phase and a different amplitude probability. Since its action on the qubit state corresponds to the one that can be achieved by combining a Pauli X gate and a Pauli Z gate, it is usually called *bit-phase-flip* gate.
The effect of this gate on a generic state can be observed on the Bloch sphere as a rotation around the y-axis of π-radians.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

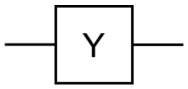


Figure 1.12: Y gate symbol

In general:

$$|\Psi'\rangle = Y|\Psi\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} -i \cdot \beta \\ i \cdot \alpha \end{bmatrix} = -i \begin{bmatrix} \beta \\ -\alpha \end{bmatrix}$$

Example: This example shows what happens if a Y gate is applied on a qubit whose state is $|0\rangle$.

$$|\Psi\rangle = Y|0\rangle = i|1\rangle \sim |1\rangle$$

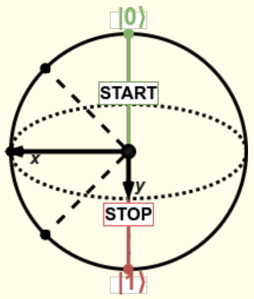


Figure 1.13: Example: Y gate

Arbitrary rotations There are three gates - R_x, R_y and R_z - that allow to do an arbitrary rotation around the x, y and z axis, respectively. These operators are:

$$R_x(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -i \cdot sin\frac{\theta}{2} \\ -i \cdot sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix} \quad R_z(\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}$$

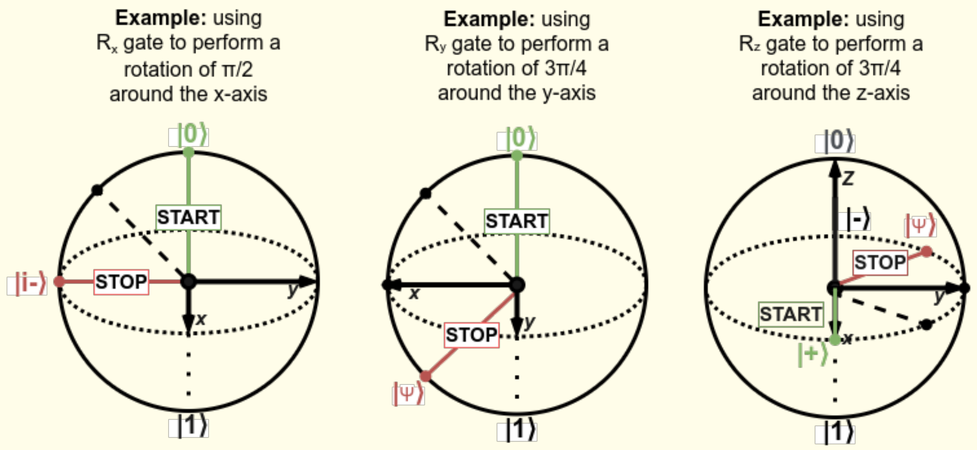


Figure 1.14: Examples: R_x, R_y, R_z

It must be pointed out that while R_x and R_y change the probabilities of the system states, R_z does not (*i.e.* the probability of measuring $|0\rangle$ rather than $|1\rangle$ remains the same). What R_z changes is the relative phase of the qubit.

R_z: SPECIAL CASES

The two following gates are particular cases of the R_z gate.

T-gate This gate lets the qubit perform a rotation around the z-axis equal to $\frac{\pi}{4}$-radians.

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$

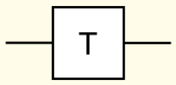


Figure 1.15: T gate symbol

S-gate By looking at the Bloch sphere, it corresponds to a rotation around the z-axis equal to $\frac{\pi}{2}$-radians.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix}$$

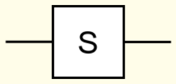


Figure 1.16: S gate symbol

Hadamard gate Generally speaking, it is possible to describe this gate by saying that it is able to move the state $|0\rangle$ to the state $|+\rangle$, and the state $|1\rangle$ to the state $|-\rangle$.

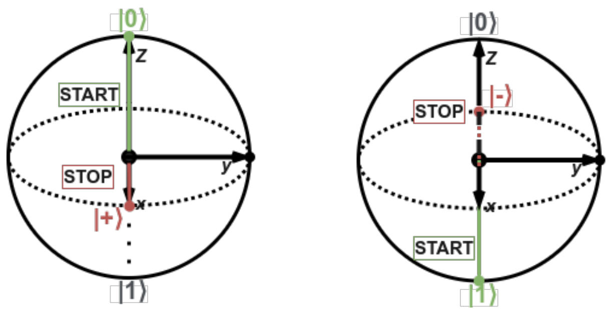


Figure 1.17: Example: H gate

The unitary matrix and the circuit symbol characterizing this gate are:

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

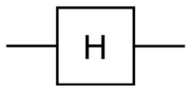


Figure 1.18: H gate symbol

The Hadamard gate is one of the most important one-qubit gates. This is due to the fact that - when applied to a basis state - it allows to obtain a superposition of basis states, each of them being characterized by the same probability to be measured. For instance:

$$|\Psi'\rangle = H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

By applying what has been said about measurement operation, it is clear that there are 50% of chances of measuring $|1\rangle$ and 50% of chances of measuring $|0\rangle$. This fact is so important since it is exploited by almost all quantum algorithms. In fact, it is common in quantum algorithms starting from a set of N qubits in a given state (*e.g.* one of the basis states), and applying a set of Hadamard gates in order to obtain a uniform superposition of 2^N states. This allows to perform a concurrent evaluation of all possible patterns.

The Hadamard gate can be seen by another point of view: it allows to move the frame of reference from the z-axis to the x-axis. In other words, it can be used to transform one basis to another:

$$(|0\rangle, |1\rangle) \iff (\frac{|0\rangle + |1\rangle}{\sqrt{2}}, \frac{|0\rangle - |1\rangle}{\sqrt{2}}) = (|+\rangle, |-\rangle)$$

1.3.2 Two-Qubit Gates

Generic controlled Gate Among the multiple-qubit gates, there is a wide range of gates which is based on the same principle: a given number of control qubits decide if a given operation on another set of qubits must be performed or not. To explain this concept, the two-qubit case - one control qubit and one target qubit - will be used.

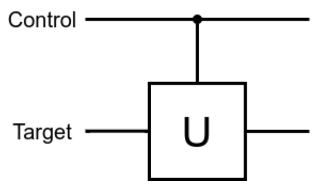


Figure 1.19: Generic two-qubit controlled gate

It is possible to define the generic operation performed by the single-qubit gate U as follows:

$$U = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}$$

Assuming that the action of U on the target qubit must be taken only if the first qubit is equal to $|1\rangle$, for the controlled-U gate it holds that:

$$cU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \tag{1.10}$$

Referring to Equation 1.10, it is possible to see what happens for all the possible input patterns:

$$|\Psi'\rangle = cU \cdot |00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = |00\rangle$$

$$|\Psi'\rangle = cU \cdot |01\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = |01\rangle$$

$$|\Psi'\rangle = cU \cdot |10\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ u_{00} \\ u_{10} \end{bmatrix} = |1\rangle \otimes U\,|0\rangle$$

$$|\Psi'\rangle = cU \cdot |11\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ u_{01} \\ u_{11} \end{bmatrix} = |1\rangle \otimes U\,|1\rangle$$

All the single qubit gates previously presented can be theoretically implemented in the "controlled version".

Controlled-NOT (CNOT) The controlled-NOT - also known as CNOT or CX - is the controlled variant of the Pauli X gate, *i.e.* the X gate is applied to the target qubit only if the control qubit assumes the value $|1\rangle$.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

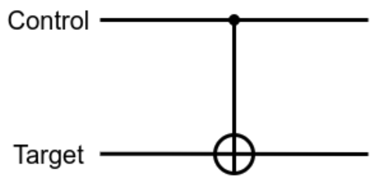


Figure 1.20: CNOT gate symbol

Swap gate This gate allows to swap two qubits. It is defined as follows:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

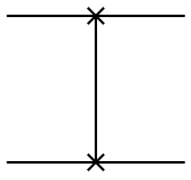


Figure 1.21: SWAP gate symbol

In general, it acts in this way:

$$|\Psi'\rangle = SWAP \cdot |\Psi\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

It can be seen that:

$$InputState : |00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \longrightarrow OutputState : |00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$InputState : |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \longrightarrow OutputState : |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Similar reasoning can be done for input states $|10\rangle$ and $|11\rangle$.
A final observation that can be made about the SWAP gate is that it can be implemented, for example, using three CNOT gates as shown in Figure 1.22.

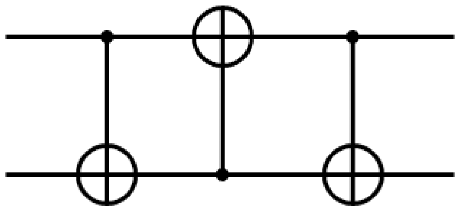


Figure 1.22: Implement the SWAP gate using CNOT gates

1.3.3 Three-Qubit Gates

Toffoli (CCNOT) The Toffoli gate has two control qubits and one target qubit. The X operation is applied to the target qubit if and only if both control qubits are set to $|1\rangle$.

$$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

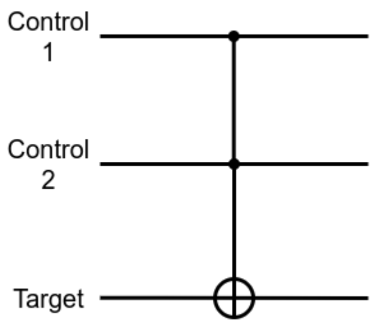


Figure 1.23: Toffoli gate symbol

Fredkin(CSWAP) This gate has one control qubit and two target qubits: the SWAP operation is performed on the target qubits if and only if the control qubit is set to $|1\rangle$.

$$CSWAP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

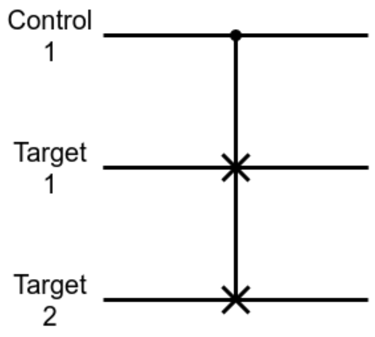
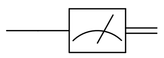


Figure 1.24: Fredkin gate symbol

1.4 Typical Notation for Other Circuit Elements

In Table 1.2 other symbols needed to understand quantum circuits are summarized.

Table 1.2: Some circuit symbols

Name	**Meaning**	**Symbol**
Bit	This symbol is used to represent a classical bit	═══
Qubit	This symbol is used to represent a qubit	───
N Qubits	This symbol is used to represent N qubits	$-\!\!/^{n}\!-$
Measurement	This symbol is used to represent the measurement operator	

1.5 How To Analyze a Quantum Circuit

In Section 1.3 it was explained that quantum operators are described by means of unitary matrices. Then, a quantum circuit can be seen as set of gates - each one represented by a proper unitary matrix - connected to each other. There can be two kinds of connections between gates belonging to the same circuit: *series* and *parallel* connections.
In order to be able to understand the behaviour of a given circuit, it is necessary to understand how to compute the overall unitary matrix describing the action of gates placed in parallel or in series. Before moving on by explaining how to compute what just said, it is important to clarify one concept inherent to quantum circuits, that will be more clear in the next paragraphs.

The "time-flow" is to be understood from left to right, *i.e.* the evolution of the state of a qubit has a physical meaning if considered from left to right. However, when the "matrix transfer function" of the whole circuit (or of a part of the circuit) has to be computed, unitary matrices must be written from right to left, *i.e.* the first (leftmost) gate in the circuit is described by the rightmost unitary matrix.

1.5.1 Gates Connected in Series

The overall transfer function of two gates connected in series can be computed as shown in Figure 1.25.

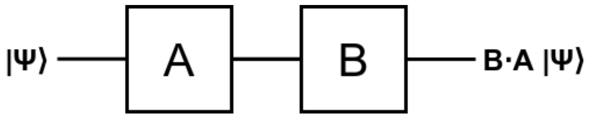


Figure 1.25: Series of two generic one-qubit quantum gates

It is clear that this concept can be extended to an arbitrary number of gates N.

1.5.2 Gates Connected in Parallel

When two gates are placed in parallel, and there is the need to compute the overall unitary matrix acting on the two qubits, the procedure shown in Figure 1.26 can be applied.

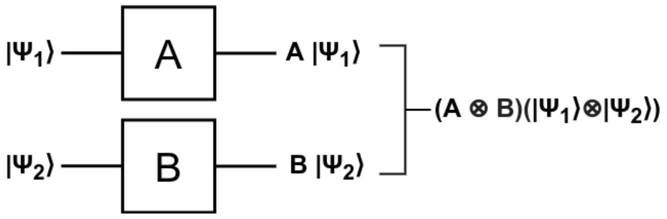


Figure 1.26: Parallel of two generic one-qubit quantum gates

Also in this case the computation can be extended to the fully general case of N qubits. It is noted that - if a quantum gate is applied only to a subset of qubits - qubits where no gates are acting can be treated as affected by an identity (see Figure 1.27).

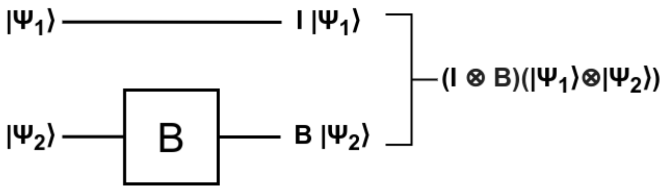


Figure 1.27: Parallel connection - particular case

1.5.3 Circuit Composition

In the following an example showing how to analyze a quantum circuit is proposed.

Example: how to read a quantum circuit

Compute the output state of the circuit shown in Figure 1.28, where A, B, C and D represent generic gates.

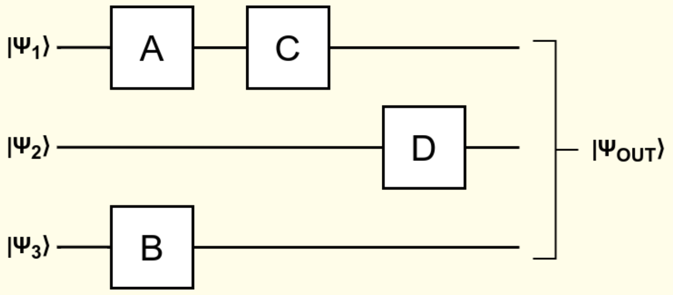


Figure 1.28: Example: generic quantum circuit

Solution

To analyze this circuit two steps have to be followed:

1. Write a unique expression for the three input qubits by performing the tensor product among them:

$$|\Psi_1\Psi_2\Psi_3\rangle = |\Psi_1\rangle \otimes |\Psi_2\rangle \otimes |\Psi_3\rangle$$

2. Compute the overall matrix transfer function by looking at the circuit from right to left (according to what stated before):

$$|\Psi_{out}\rangle = (I_2 \otimes D \otimes I_2) \cdot (C \otimes I_4) \cdot (A \otimes I_2 \otimes B)\, |\Psi_1\Psi_2\Psi_3\rangle$$

   where I_k is an identity matrix of order k.

A step-by-step analysis is presented in Figure 1.29.

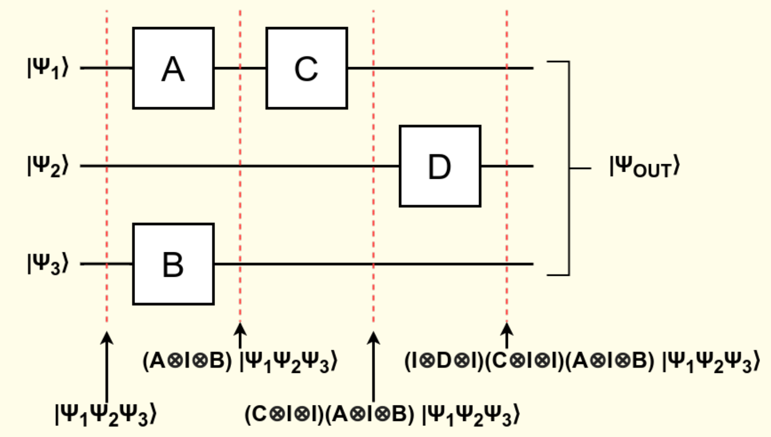

$|\Psi_1\rangle$
$|\Psi_2\rangle$
$|\Psi_3\rangle$
A
C
D
B
$|\Psi_{OUT}\rangle$
$(A\otimes I\otimes B)\,|\Psi_1\Psi_2\Psi_3\rangle$
$(I\otimes D\otimes I)(C\otimes I\otimes I)(A\otimes I\otimes B)\,|\Psi_1\Psi_2\Psi_3\rangle$
$|\Psi_1\Psi_2\Psi_3\rangle$
$(C\otimes I\otimes I)(A\otimes I\otimes B)\,|\Psi_1\Psi_2\Psi_3\rangle$

Figure 1.29: Example: generic quantum circuit - step by step solution

Knowing equivalences between quantum gates can sometimes speed-up the circuit analysis, for this reason some useful equivalences are listed in Table 1.3.

Table 1.3: Some equivalences between gates

$X^2 = Y^2 = Z^2 = I$	$H = \frac{1}{\sqrt{2}}(X + Z)$
$S = T^2$	$Z = HXH$
$-1Y = HYH$	$-1Y = XYX$

1.5.4 Example of Quantum Circuits

This section shows an example in which a real quantum circuit is analyzed by following two different strategies:

- Exploiting the matrix calculation, as done before.
- Adopting a faster method based on truth tables of different gates.

Given the quantum circuit in Figure 1.30, define the states $|\Psi_1\rangle$, $|\Psi_2\rangle$ and $|\Psi_3\rangle$.

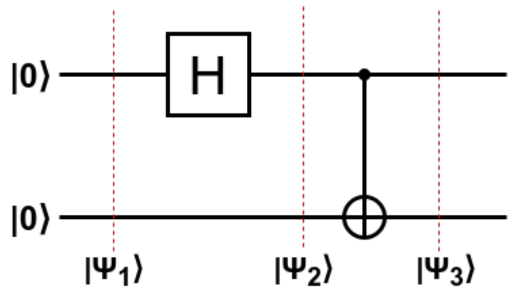


Figure 1.30: Example: analyzing a quantum circuit

Method 1: matrix calculation

In this simple quantum circuit two operators are involved: the Hadamard gate and the CNOT gate. They are represented by the two following unitary matrices:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The steps to follow to compute the states $|\Psi_1\rangle$, $|\Psi_2\rangle$ and $|\Psi_3\rangle$ are set out below.

Step 1: Compute $|\Psi_1\rangle$

$$|\Psi_1\rangle = |0\rangle \otimes |0\rangle = |00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Step 2: Compute $|\Psi_2\rangle$

$$|\Psi_2\rangle = (H \otimes I) \cdot |00\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

Step 3: Compute $|\Psi_3\rangle$

$$|\Psi_3\rangle = CNOT \cdot (\frac{|00\rangle + |10\rangle}{\sqrt{2}}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Method 2: operators truth tables

This approach - typically much quicker to apply than the previous one - exploits the truth tables shown in Table 1.4 for the involved operators.

Table 1.4: Truth tables for H and CNOT

Hadamard	**CNOT**
$H\,|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$	$CNOT\,|0x\rangle = |0x\rangle$
$H\,|1\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$	$CNOT\,|1x\rangle = |1\overline{x}\rangle$

Step 1: Compute $|\Psi_1\rangle$

$$|\Psi_1\rangle = |0\rangle \otimes |0\rangle = |00\rangle$$

Step 2: Compute $|\Psi_2\rangle$

$$|\Psi_2\rangle = H\,|0\rangle \otimes |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Step 3: Compute $|\Psi_3\rangle$

$$|\Psi_3\rangle = \frac{CNOT\,|00\rangle + CNOT\,|10\rangle}{\sqrt{2}} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

1.6 Universal Sets of Quantum Gates

For the classical computation, the NAND gate can be considered a universal gate, that is to say all the functions can be created using only NAND gates. It is now interesting to understand which sets of quantum gates can be considered universal in the world of Quantum Computing.
The first point to clarify is that the classical world is a subspace of the quantum one; this means that - by using the quantum approach - one is able to implement all classical functions with quantum gates, while the contrary is not possible.
There are therefore two aspects to highlight: which set of quantum gates allows to implement any kind of "classical function" and which set of quantum gates is capable to implement any kind of "quantum function".

Quantum gates and classical functions It can be demonstrated that the CC-NOT (Toffoli) and X gates can be used to implement any kind of "classical function". Figure 1.31 shows that an equivalent of the NAND gate can be implemented by using a Toffoli gate, assuming that an ancilla qubit can be initialized to $|1\rangle$. If a qubit is initialized to $|0\rangle$, a X gate is clearly sufficient for having $|1\rangle$.

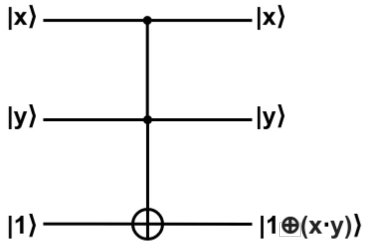


Figure 1.31: NAND gate using Toffoli

Quantum gates and quantum functions Before showing some examples of universal quantum sets, it is good to note that these sets can be said "universal" by implicitly accepting a given tolerance. This is to say that even in presence of a set of gates allow performing an arbitrary rotation around the three axes of the Bloch sphere, an "infinitely precise" hardware is required to be able to implement any kind of quantum circuit without errors, and this is clearly not feasible.
Some universal quantum sets are listed below:

- H, S, CNOT, T (Clifford + T)
- H, CCNOT
- R_x,R_y,R_z,$CNOT$

1.7 Physical Implementation of Qubits

The aim of this section is to show how qubits are actually implemented on real systems. Quantum computers have not yet reached such stable and performing solutions as to be reproduced on a large scale and many different technologies are being explored today. Among the existing technologies to implement qubits, it is possible to mention superconducting qubits, trapped-ion qubits, electron-spin qubits *etc.*

In this section, whose purpose is only to provide an introduction to the subject, without going too much into detail, only superconducting technology will be taken into consideration. The choice of explaining only the superconducting technology arises from the fact that some of the library's circuits have been tested on IBM's free access quantum computers, which are based on this technology. For this reason, having an overview of the hardware implementation of such quantum computers can help the reader to interpret the graphs proposed in chapter 4 more effectively.

1.7.1 Superconducting quantum computers

Before proceeding with the explanation of the key aspects related to superconducting qubits, it is noted that this explanation takes inspiration from [13], which the reader is invited to read to have a more rigorous vision of the topic.

Qubits Implementation

Transmon qubit The simplest way to represent a *transmon qubit* is by using a LC resonator. Studying this system from a quantum point of view, the circuit can be seen as a quantum mechanical system whose behavior can be described by the Quantum Harmonic Oscillator equations. Generally speaking, the latter is the quantum version of the well-known harmonic oscillator. The evolution over time of this system - assumed to be "closed" - is described by the Schrödinger equation. By obtaining the Hamiltonian and solving the equation, it can be demonstrated that the energy diagram describing this kind of system is like the one shown in Figure 1.32.[6]

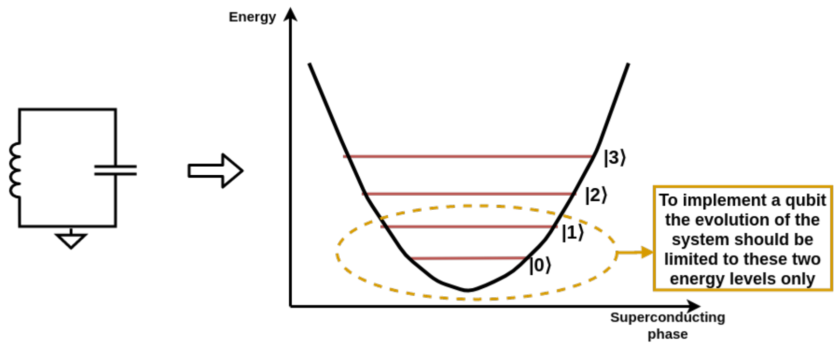

Energy
|3⟩
|2⟩
|1⟩
|0⟩
To implement a qubit the evolution of the system should be limited to these two energy levels only
Superconducting phase

Figure 1.32: Quantum Harmonic Oscillator: qualitative behaviour

In the energy diagram shown in Figure 1.32, it can be observed that the states in which the system can evolve are many. In fact, each horizontal line represents a possible state in which the system may be. Since the aim is to implement a qubit - *i.e.* a quantum mechanical system characterized by only two possible basis states - it is necessary to ensure that the system is able to "move" only between two energy levels, which typically are those characterized by the lowest energy. To do this, a *Josephson junction*[7] is employed in place of the inductor.

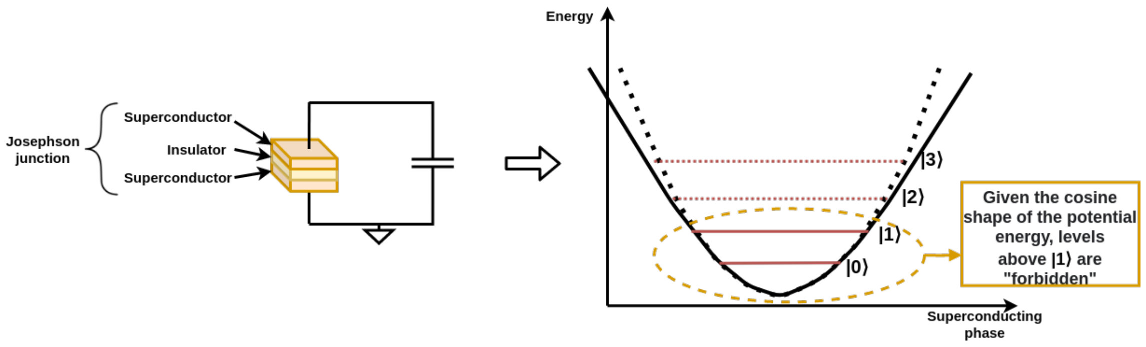


Figure 1.33: Transmon qubit: qualitative behaviour

[6]It is emphasized that the speech is deliberately qualitative, as a rigorous mathematical treatment of this topic is beyond the scope of this thesis.

[7]It makes that the potential energy shapes cosinusoidal instead of parabolic.

The fact of having inserted a Josephson junction instead of the inductor modifies the energy diagram as shown in Figure 1.33, ensuring that any excess energy supplied to the system does not result in a qubit's transition to a state that is different from the two computational states.[8] To be more precise, the oscillator becomes *anharmonic*. At this point, it can be demonstrated ([13]) that the Hamiltonian of the system can be written as the sum of two terms depending respectively on the energy required to charge the capacitor (E_C) and on the Josephson energy (E_J). To minimize what is called *charge noise*, a regime where the Josephson energy is much greater than the one related to the capacitor charges is going to be reached. This is the case of the transmon qubit of IBM quantum computers. Now that the reader has a very basic knowledge about superconducting qubits[9], it is good to see how multiple qubits can interact to each other.

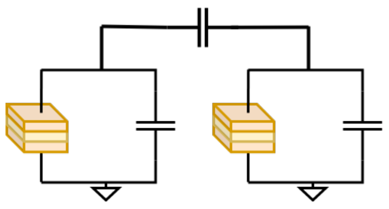
Qubits interaction When dealing with quantum circuits, the need to let different qubits "communicate" to each other - in order to implement multi-qubit gates - often arises. There are two possibilities to couple different qubits:

- Direct coupling (see Figure 1.34);
- Coupling using coupler (see Figure 1.35).

A capacitive coupling allows the interaction of the two systems through the electric field; *vice versa*, the magnetic coupling lets the two systems "communicate" through the magnetic field. Once two qubits are coupled, a unique Hamiltonian can be used to describe the evolution of the two. This Hamiltonian can be obtained by adding three terms: the Hamiltonians of the two systems and what is called the *interaction Hamiltonian*. The latter depends mainly on the kind of coupling implemented.

[8]It should be noted that the states described by higher energies are still part of the system, although the system state cannot easily "access them" with a proper device engineering.

[9]It should be noted that the structure presented is only a basic scheme. Current quantum computers adopt more sophisticated implementations of superconducting qubits, such as flux qubits or fluxonium qubits. Their analysis is beyond the scope of this thesis.


(a) Direct capacitive coupling

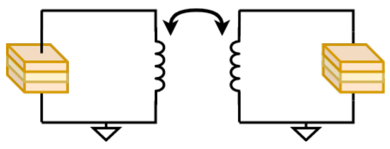
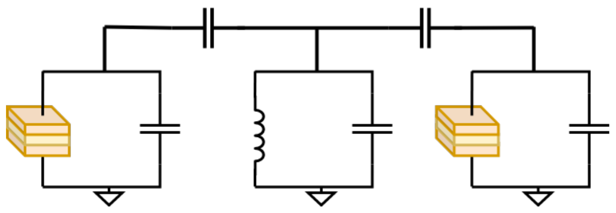

(b) Direct inductive coupling

Figure 1.34: Transmon qubits: direct coupling


(a) Capacitive coupling using a coupler

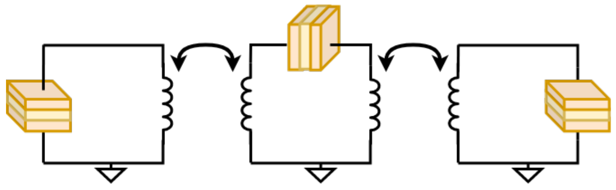

(b) Inductive coupling using a coupler

Figure 1.35: Transmon qubits: coupling via coupler

Qubit Noise

In this section, the purpose is to mention which are the noise sources that make the behavior of the superconducting qubit not ideal, and which parameters allow describing how good a qubit is.

Effects of noise In the previous paragraphs each qubit was considered to be a "closed" quantum system, such that its time evolution was well determined by its Hamiltonian. If noise is added to this ideal system, what happens is that the evolution of the system does not respect anymore what stated by the Schrödinger equation. To say it better, the system is no more deterministic and its evolution can be "predictable" - with the Schrödinger equation - only for a "short period".

The effects of noise on a two-level quantum system can be generally described by referring to two quantities: the relaxation time and the decoherence time.

Relaxation time The phenomenon of relaxation is a mechanism that describes how a system that is located at a higher energy level tends, over time, to reduce its energy. It is also equivalent to say that, if the system is initialized to $|1\rangle$, the probability of measuring $|1\rangle$ gradually reduces. It is experimentally demonstrated that, in most cases, the probability associated with the high energy state is characterized by an exponential decay. Relaxation time constant is typically called T1.

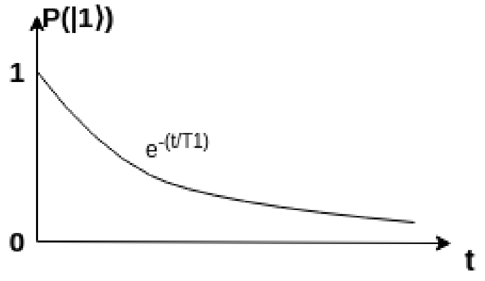


Figure 1.36: Relaxation

Relaxation phenomena are usually due to interactions with the external environment, as it is not possible in practice to obtain a completely isolated system.

Decoherence time Decoherence is a phenomenon for which, after a given amount of time, it is no longer possible to describe the state of a qubit as a single vector on the Bloch sphere. For example, starting from the state $|+\rangle$, which is described by the state vector $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, after a certain time interval, it is no more possible to represent that state with a single vector on the Bloch sphere. In other words, while the initial state was a coherent superposition of states, that is, described by a single vector, the state that is generated after a given amount of time is a decoherent mixture of different superposition states.

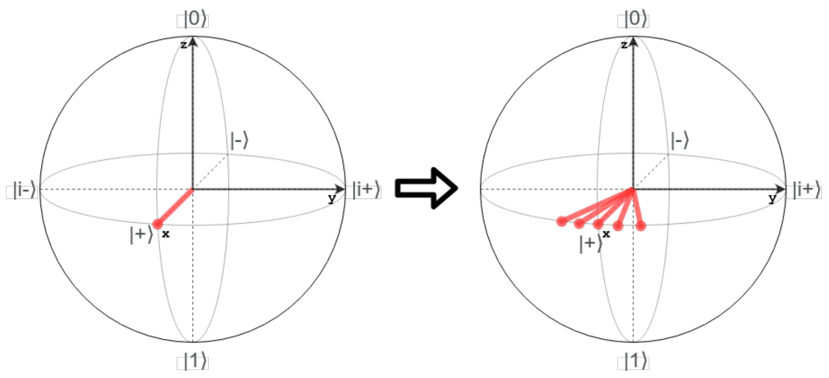


Figure 1.37: Decoherence

Again, decoherence is due to phenomena related to the external environment. In this context, the qubit is described by the so-called *density matrix*, the explanation of which is beyond the scope of this thesis. Decoherence - similarly to relaxation - experimentally shows an exponential decay with the time constant T2 called *decoherence time*.

Noise sources Now that the problem, at least in broad terms, should be clear, it is interesting to investigate the main sources of noise that can be found in a superconducting qubit. Noise sources can be divided in two main categories: systematic and stochastic noise sources.

Systematic noise This category includes all sources of noise such that it is possible to know the value of the error introduced and compensate for it. In other words, they are those sources of noise that do not change, or change predictably, every time they occur. For this reason they are easy to compensate. An example would be a pulse that rotates the phase of a qubit a little more or a little less than expected.

Stochastic noise Noises belonging to this category are the most dangerous. Such noises have a random behavior and are more difficult to compensate. Examples of stochastic noises can be unwanted electric or magnetic fields, thermal noise of components in control circuits *etc.* Decoherence and other critical parameters of qubits typically depend largely on noises belonging to this category.

Gates Implementation

The purpose of this section is to show how quantum gates are physically implemented.

Single qubit gates Suppose to have two levels - $|0\rangle$ and $|1\rangle$ - separated by an energy equal to $\hbar\omega_0$, where ω_0 is the qubit resonant frequency. In order to implement the desired gate, the idea is to apply for a finite time a sine wave electromagnetic oscillation[10] with frequency ω_0, amplitude A and phase φ.

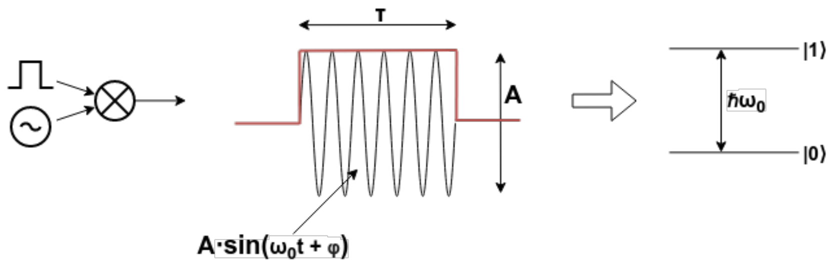


Figure 1.38: Single-qubit gates: physical implementation

By applying this signal, the amount of rotation θ is proportional to A and θ as shown in Equation 1.11,

$$\theta \propto A \cdot \tau \tag{1.11}$$

where θ is a rotation around the x or y axis depending on the value assumed by phase φ. In particular:

$$\begin{cases} R_x : & \varphi = 0 \\ R_y : & \varphi = \frac{\pi}{2} \end{cases}$$

This model - called *Magnetic Resonance* - allows to implement in a "direct" way only the R_x and R_y gates. There are several ways to implement the R_Z gate. For example, it can be implemented using gates R_x and R_y as shown in Equation 1.12.

$$\begin{cases} R_z(\theta) = & HR_x(\theta)H \\ H = R_x(\pi)R_y(\frac{\pi}{2}) \end{cases} \longrightarrow R_z(\theta) = R_x(\pi)R_y(\frac{\pi}{2})R_x(\theta)R_x(\pi)R_y(\frac{\pi}{2}) \tag{1.12}$$

[10]It is noted that, in reality, the modulating signal is not a rectangular pulse, but a Gaussian one.

However, there are some problems with this solution. The first, quite obvious, is the high number of gates required. The second is that, typically, the various technologies tend to maintain a fixed amplitude and to vary the duration[11]. This fact makes gates slower. Another approach that allows to implement the R_z gate more efficiently is that it is possible to demonstrate that a rotation of the state vector around the z-axis equal to θ, can be achieved by rotating the x and y axes of $-\theta$, *i.e.* by changing the reference system. In other words, it is possible to software implement the R_z gate by manipulating the x and y phases as shown in Equation 1.13, where θ represents the desired rotation around the z axis.

$$\begin{cases} \varphi_{xnew} = \varphi_{xold} - \theta \\ \varphi_{ynew} = \varphi_{yold} - \theta \end{cases} \quad where \quad \varphi_{yold} = \frac{\pi}{2} + \varphi_{xold} \tag{1.13}$$

In this way, R_z gates are virtually implemented without any electromagnetic wave.

Two qubits gates The aim of this paragraph is to show how two-qubit gates are implemented. To do this, the CNOT gate will be used as example.
Suppose to have two transmon qubits as shown in Figure 1.34a. Also assume that each qubit has its own resonance frequency - ω_1 and ω_2 (typically ω_1 and ω_2 are very close to each other) - and that an electromagnetic wave oscillating at the frequency of the target is applied to the control qubit.

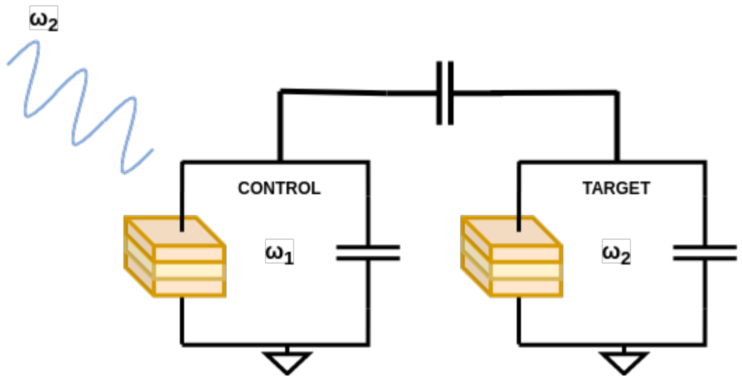


Figure 1.39: Cross-resonance

[11]For instance, if the amplitude is kept constant, the duration of an $R_x(\pi/2)$ is half of $R_x(\pi)$.

In this situation it can be shown that, when a qubit is not excited at its resonance, but at a neighbor to it, *i.e.* if the detuning δ[12] is small enough ($\delta \ll \omega_1$), the overall effect on the system is a small rotation around the z-axis of the control qubit and a rotation around the x-axis of plus or minus θ of the state of the target qubit, where the sign for the rotation depends on the value of the control qubit, as shown in Table 1.5. This mechanism is called *cross-resonance* and it is at the basis of IBM quantum hardware.

Table 1.5: Effect of cross resonance on target qubit

Control	**Target**
$\|0\rangle$	$R_x(\theta)$
$\|1\rangle$	$R_x(-\theta)$

The last step for implementing a CNOT gate is to compensate the cross resonance effect on the target qubit when the control qubit state is $|0\rangle$, and to make the overall rotation equal to π when the state of the control qubit is equal to $|1\rangle$. This can be achieved by placing appropriate single qubit gates before and after cross resonance.

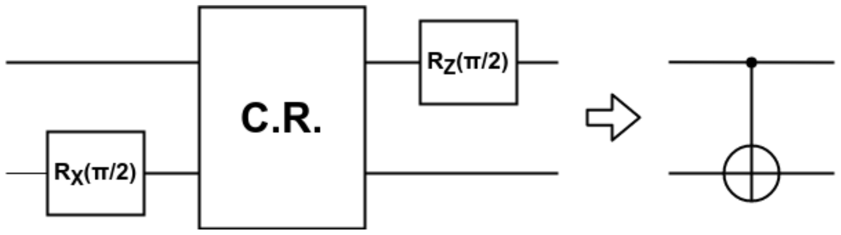


Figure 1.40: CNOT physical implementation

1.8 Quantum Computers: Instruction Sets

The purpose of this section is to emphasize that different quantum computers can be characterized by different instruction sets, *i.e.* different sets of native gates. This translates into the fact that all the quantum gates presented up to this point are

[12] $\delta = \omega_1 - \omega_2$

“translated” by each quantum computer in accordance with its set of characteristic gates.
In this section IBM superconducting qubits will be considered.

1.8.1 IBM Superconducting Native Set

As stated in [11], IBM quantum computers, based on superconductors, are characterized by four basis gates: U1, U2, U3 and CNOT. The first two gates are actually a subcase of the U3 one.

U1 It is called *phase gate*, and its action is equivalent to the one of the R_Z, previously presented. By using this gate, it is possible to implement the Z, T, S gates and their inverse.

$$U1(\lambda) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{bmatrix}$$

U2 Its duration is one unit of gate time. The unitary matrix representing this gate is:

$$U2(\varphi,\lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\varphi} & e^{i\lambda+i\varphi} \end{bmatrix}$$

It is important since it allows to implement the Hadamard gate.

$$H = U2(0,\pi)$$

U3 It allows to construct any kind of single-qubit gate and its duration is twice that of U2. The matrix representation is:

$$U3(\theta,\varphi,\lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} cos(\frac{\theta}{2}) & -e^{i\lambda}sin(\frac{\theta}{2}) \\ e^{i\varphi}sin(\frac{\theta}{2}) & e^{i\lambda+i\varphi}cos(\frac{\theta}{2}) \end{bmatrix}$$

CX It is the CNOT gate previously presented.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

To get an idea of how the gates presented up to this point are actually implemented in IBM quantum computers, it is recommended to take a look at the appendix A.

Chapter 2

Quantum arithmetic circuits: state of the art

"I keep six honest serving-men
(They taught me all I knew);
Their names are What and Why and When
And How and Where and Who."

Rudyard Kipling

Many quantum algorithms require arithmetic circuits in order to be implemented.

This chapter gives an overview about the quantum arithmetic circuits implemented in the proposed software library. The first section provides an introduction to the OpenQASM language, widely used hereinafter. The next one aims to show how to construct equivalents of the elementary classical logic gates using quantum gates. The remaining sections are dedicated to the four main operations: addition, subtraction, multiplication and division. In general, in each of the sections dedicated to the fundamental operations, there is an emphasis on the parallelism that exists in the way in which the various operations are carried out in the two worlds: classical and quantum ones. Going into detail, it was decided to highlight two aspects of each quantum circuit: how it is implemented - *i.e.* necessary steps expressed in a pseudo OpenQASM language - and what are its main characteristics. For further details on the circuits in question, it is recommended to consult the references in the bibliography.

2.1 OpenQASM: a Basic Guide

Although the developed library can be used with different frameworks for Quantum Computing, it is worth pointing out that it was born with the idea of generating quantum circuits described using the OpenQASM language. It is therefore good to take an overview of this language. It is noted that this language is presented in this chapter as here it will be used a pseudo OpenQASM language to describe the various circuits.

Programming languages Generally speaking, programming languages for Quantum Computing are born with the idea of simplifying the process of running a quantum algorithm on real hardware, where "simplifying" means adding a layer of abstraction between the operations that the algorithm needs to execute and how they have to be applied on real systems, in order to "manipulate" qubits appropriately. In other words, programming languages allow to translate algorithms operations into abstract entities - gates - which in turn will be translated into the sequence of operations necessary to change in the desired way the state of the target qubits. Using this additional layer of abstraction, the programmer does not have to care about the specific characteristics of the target where the algorithm is going to be run, as they will be taken into account when compiling the code. To make this concept clearer, it is good to highlight the main steps that are performed whenever an algorithm runs on a real quantum computer:[1]

Compilation This first step is performed on classical computers and its aim is to generate a quantum circuit on which offline optimizations have been performed;

Circuit generation This second step is also typically performed on a classical computer. It aims to generate an optimized circuit in such a way that it can be compatible with the limits of the target architecture;

Execution In this phase the circuit is executed on the target quantum computer;

[1]See [8] for a more detailed explanation of the topic.

Post-processing This last phase is about the analysis of the output results. It is done on classical computers.

Now that the reader should have an idea of what Quantum Computing programming frameworks are and what they are for, it is possible to proceed with a brief introduction to the OpenQASM language.

General rules OpenQASM stays for Open Quantum Assembly Language. It is a programming language for Quantum Computing defined at IBM which follows these general rules ([8]):

- The first line of code (excluding comments) must be OPENQASM M.m, with the two "m", upper and lower case, to indicate the version. This line can occur only once in the code;
- Semicolons are used to separate statements;
- Whitespace are not considered;
- The language is case sensitive;
- Comments begin with // and end with a newline.

Main statements The OpenQASM language allows to define quantum and classical registers as follows:

$$\text{Quantum Register} \rightarrow qreg\ name[size]; \quad \text{Classical Register} \rightarrow creg\ name[size];$$

IBM quantum gates - corresponding to those seen in chapter 1 - can be used importing the "qelib1.inc" file. To import files, the following syntax has to be used:

include "filename";

In general, a gate can be inserted into the circuit description using the following syntax:

gate_name qreg_name[i],... ;

For example, assume to have a two-qubit quantum register q (*qreg q[2];*). In order to apply a CNOT gate on the two qubits - assuming that the control qubit is *q[0]* and the target qubit is *q[1]* - it is possible to proceed as follows:

```
cx q[0], q[1];
```

Within an OpenQASM file, it is also possible to define custom gates that can be reused multiple times within code. To do this, proceed as follows:

```
gate gate_name q1, q2, ...
{
...
gate description
...
}
```

The last of the “main” statements is the one related to the measurement operator, which can be inserted into the circuit as follows:

```
measure qbit -> cbit;
```

The reader should note that the description provided of the language is not complete, and that only the basic notions necessary to correctly interpret the codes proposed later in this and subsequent chapters have been covered. For further details [8] is suggested.

2.2 Quantum gates as classical

In order to understand the behaviour of the quantum arithmetic circuits presented in the following, it could be useful to introduce an approach that allows to perform a kind of parallelism between classical and quantum gates ([12]). Please note that although some gate implementations may seem redundant, they find their usefulness in boolean oracle designs, *i.e.* a kind of black box needed in different quantum algorithms.

NOT The classical NOT gate can be implemented in the quantum world by using one X gate or two X gates combined with a CNOT one.

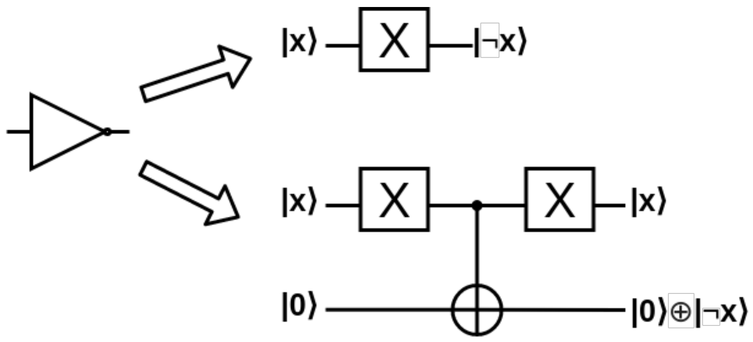


Figure 2.1: NOT gate using quantum gates

AND The Toffoli gate can be used to obtain an equivalent of the classical AND gate:

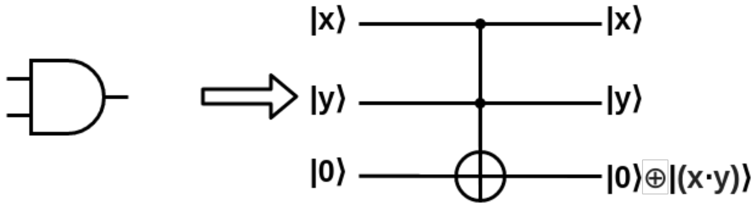


Figure 2.2: AND gate using quantum gates

OR The quantum version of the classical OR gate makes use of five X gates and one Toffoli:

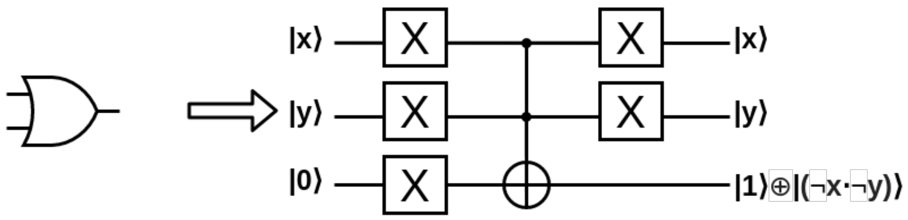


Figure 2.3: OR gates using quantum gates

XOR There are different ways to implement an equivalent of the classical XOR gate using quantum gates:

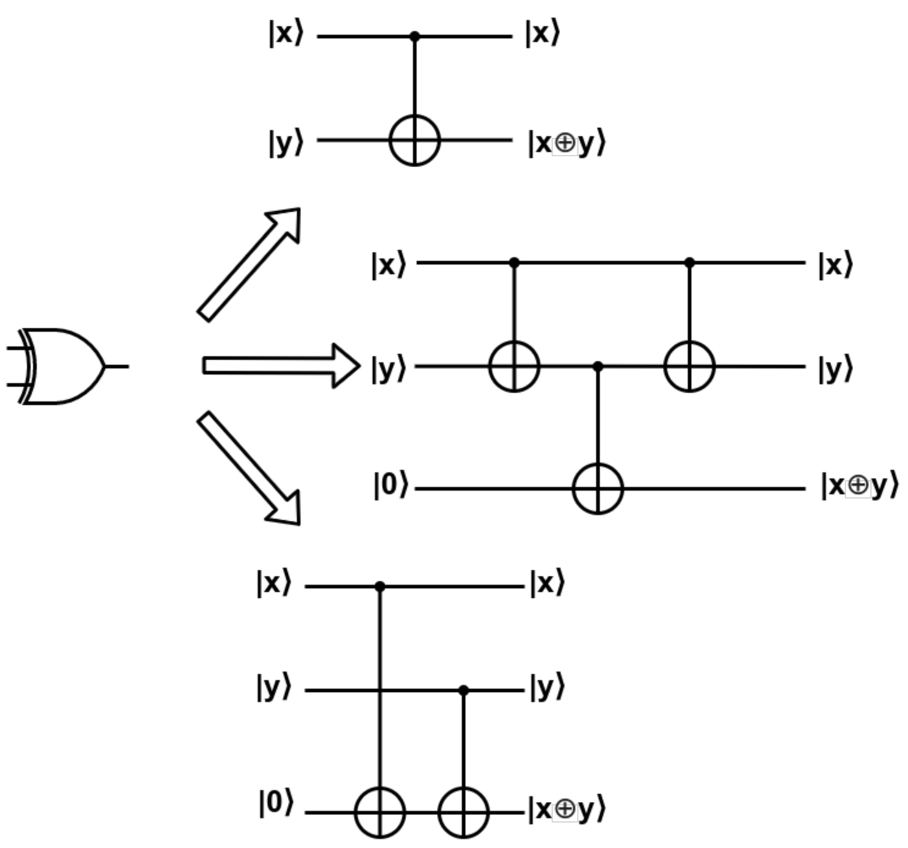


Figure 2.4: XOR gates using quantum gates

2.3 Addition

2.3.1 Classical computation: Full Adder, Half Adder

When dealing with digital computation, the most straightforward technique to implement addition of integer numbers is the ripple carry adder, based on a chain of full adders.

Full Adder (FA) A full adder is a component having three input bits and two output bits.

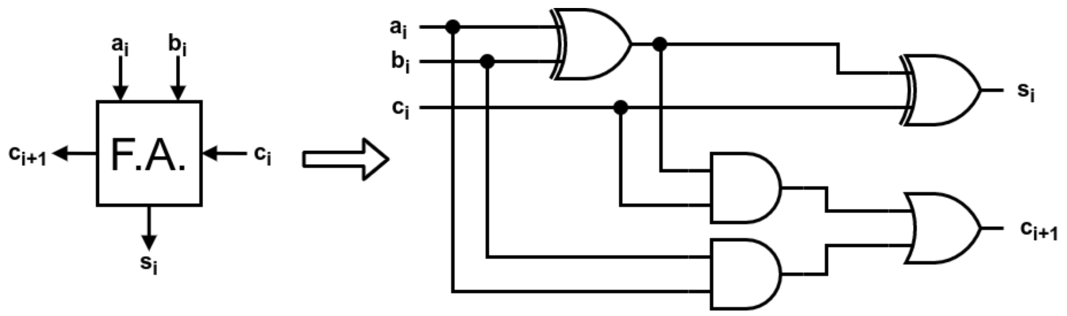


Figure 2.5: Classical FA

Particularly, the three inputs a_i, b_i and c_i have the same weight, while the two outputs have different weights: the sum bit's (s_i) weight is the same of the three inputs, while the one of the carry out (c_{i+1}) is one more than the weight of the incoming bits.
The full adder behaviour is detailed in Table 2.1.

Table 2.1: FA truth table

a_i	b_i	c_i	s_i	c_{i+1}
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

From Table 2.1 a couple of equations describing the FA behaviour can be obtained:

$$\begin{cases} s_i = a_i \oplus b_i \oplus c_i \\ c_{i+1} = c_i(a_i \oplus b_i) + a_i b_i \end{cases} \tag{2.1}$$

Half Adder (HA) If c_i is always equal to zero, the sum between two bits can be calculated using a half adder (HA), where the sum bit (s_i) and the carry out bit (c_{i+1}) are defined as follows:

$$\begin{cases} s_i = a_i \oplus b_i \\ c_{i+1} = a_i b_i \end{cases} \tag{2.2}$$

2.3.2 Classical computation: Ripple Carry Adder

By allocating in parallel n full adders, it is possible to obtain an n bits integer adder.

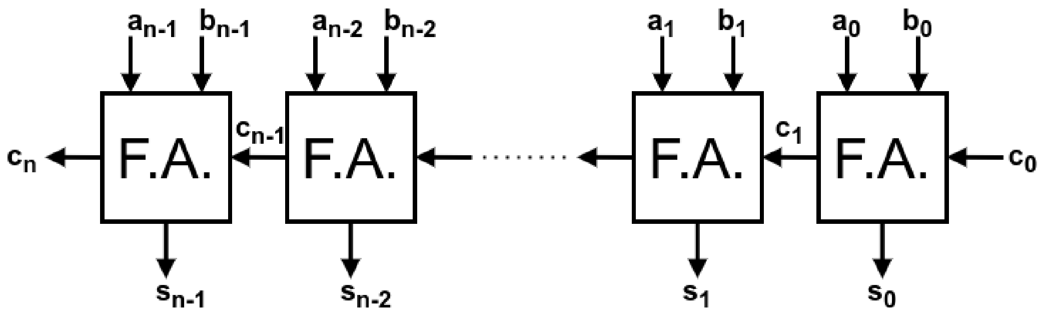


Figure 2.6: Classical RCA

In digital electronics literature there exist several more efficient ways, in terms of cost and delay, to implement integer adders; but the implementation of this kind of adders goes beyond the aims of this thesis.
The classical FA, HA and RCA have been introduced since the purpose now is to introduce their quantum counterparts.

2.3.3 Quantum computation: Full Adder and Half Adder

Quantum Half Adder (QHA) By keeping in mind what stated in Section 2.2, it is quite easy to implement classical circuits by using quantum gates. In particular, it has been said that an equivalent of the XOR gate can be implemented by using a CNOT quantum gate, while an AND gate making use of a Toffoli gate. The "quantum version" of the Half Adder - whose behaviour is described by Equation 2.2 - is shown in Figure 2.7.

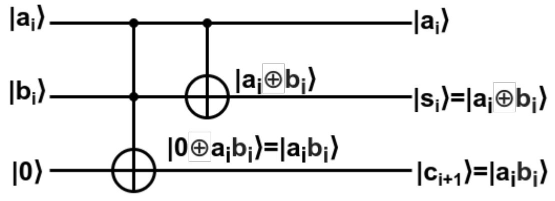

$|a_i\rangle$
$|b_i\rangle$
$|0\rangle$
$|a_i\oplus b_i\rangle$
$|0\oplus a_ib_i\rangle=|a_ib_i\rangle$
$|a_i\rangle$
$|s_i\rangle=|a_i\oplus b_i\rangle$
$|c_{i+1}\rangle=|a_ib_i\rangle$

Figure 2.7: Quantum Half Adder

Quantum Full Adder (QFA) Similarly to what has been done in the HA case, a "quantum version" of the FA - whose behaviour is described by Equation 2.1[2] - can be obtained as shown in Figure 2.8.

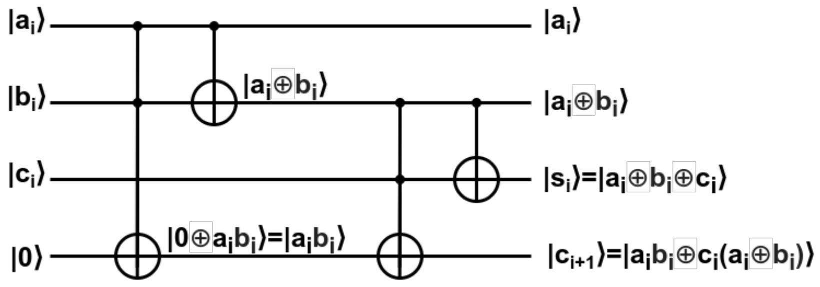

$|a_i\rangle$
$|b_i\rangle$
$|c_i\rangle$
$|0\rangle$
$|a_i\oplus b_i\rangle$
$|0\oplus a_ib_i\rangle=|a_ib_i\rangle$
$|a_i\rangle$
$|a_i\oplus b_i\rangle$
$|s_i\rangle=|a_i\oplus b_i\oplus c_i\rangle$
$|c_{i+1}\rangle=|a_ib_i\oplus c_i(a_i\oplus b_i)\rangle$

Figure 2.8: Quantum Full Adder

2.3.4 Quantum computation: addition - advanced solutions

There are several solutions that allow performing addition in an efficient way. Three different kinds of adders - found in the literature - have been implemented in this library. The following sections will give an overview of these three different solutions. The reader is suggested to examine [2], [4] and [7] for further details.

[2]It can be observed that - in classical electronics - an OR gate is typically used for the carry out calculation (see Equation 2.1 and Figure 2.5). Although in the quantum case a XOR gate is used rather than an OR, the behaviour of the FA remains unchanged.

Before proceeding, please note that from here on, IBM notation (see Figure 1.7) will be used in the representation of the circuits.

Integer quantum adder - Cuccaro, Draper, Moulton, Kutin

In [2] the same approach - except for small modifications - is used to implement two versions of the same adder: with and without carry in.

Adder without carry in

This version of the adder makes use of an ancillary qubit. As detailed in [2], the design of this adder is achieved by following two steps:

1. Development of the basic structure;
2. Optimization by visual inspection of the basic structure.

Basic structure This adder makes use of two building blocks: MAJ and UMA. These are three qubits components that allow to implement a reversible version of the "classical" ripple carry adder. The behaviour of these gates is detailed in Figure 2.9.

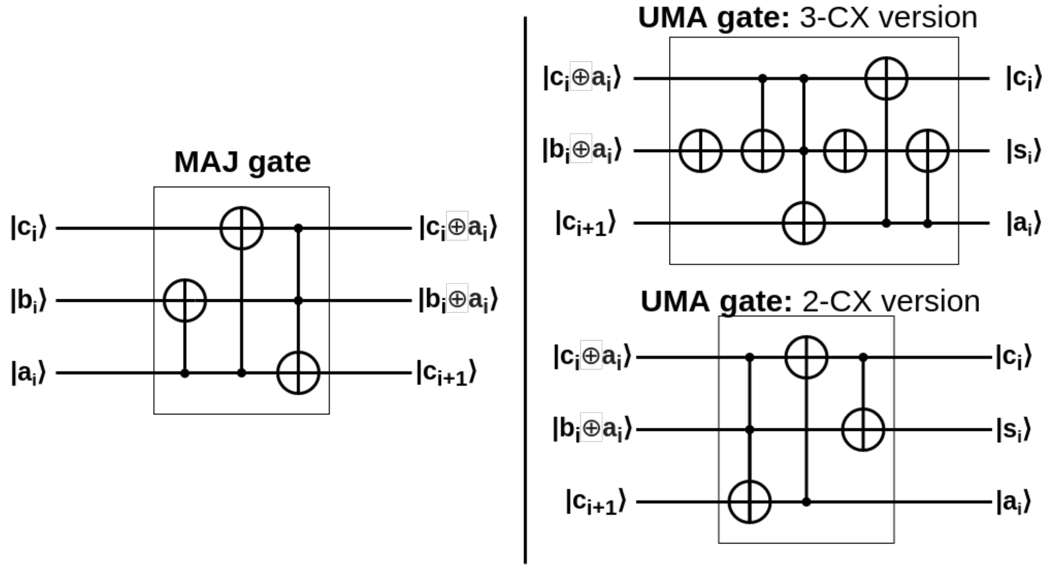


Figure 2.9: MAJ and UMA gates

As already stated, the two blocks above allow - if correctly ordered - to implement an integer adder. An example of a three-qubit integer adder is shown in Figure 2.10.

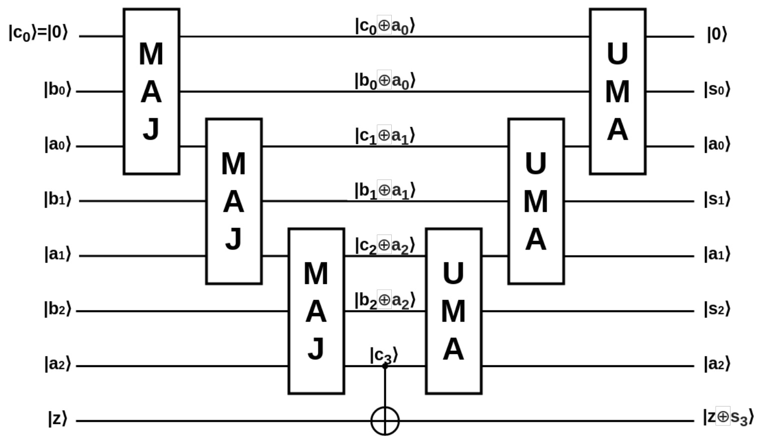


Figure 2.10: Integer quantum adder with three-qubit operands (without simplifications) - Cuccaro, Draper, Moulton, Kutin

From Figure 2.10 it can be seen how the MAJ gates cascade is used to generate the carry outs related to the various stages, while the UMA gate sequence is used both to write the output result and to restore the initial value of the addend a.
Before proceeding with optimization step, it is good to illustrate how to implement a version of this adder having n qubits addends. To this end, assume that all qubits are stored within the same quantum register q. If the number of qubits of each addend is n, considering the two additional qubits - the carry in[3] and the carry out qubits - the adder is characterized by a total number of qubits equal to $2n + 2$. The quantum register can therefore be considered as follows[4]:

$$qreg\ q[2n+2];$$

where:

- Addend A: q[i] with i(even) $\in$ [2,2n]
- Addend B: q[i] with i(odd) $\in$ [1,2n-1]

[3]In the "without carry in" version of the adder, this bit becomes an ancilla qubit.
[4]From here on, unless otherwise indicated, a pseudo OpenQASM notation will be used to explain the various algorithms.

- Carry in[5]: q[0]

- Carry out: q[2n+1]

The steps necessary to implement an adder with n-qubit operands are detailed below.

Step 1 Apply $n-1$ MAJ gates such as:[6]

$$MAJ\ q[i-1]\ q[i]\ q[i+1]; \quad with\ \text{ i(odd) from 1 to } 2n-1$$

Step 2 Apply a CNOT gate:

$$cx\ q[2n]\ q[2n+1];$$

Step 3 Apply $n-1$ UMA gates as follows:

$$UMA\ q[i-1]\ q[i]\ q[i+1]; \quad with\ \text{ i(odd) from } 2n-1 \text{ to 1}$$

Circuit optimization It was previously shown that there are two versions of the UMA gate: 2-CX and 3-CX. By choosing to use the more complex version of the two - *i.e.* the 3-CX - it is possible to apply a greater number of simplifications and thus obtain a more performing circuit. Therefore, looking at the structure shown in Figure 2.10, the various blocks can be replaced with their implementations (using the 3-CX version for the UMA gates) thus obtaining the structure shown in Figure 2.11.

[5]See footnote 3.
[6]The index i can range from 0 to 2n+1.

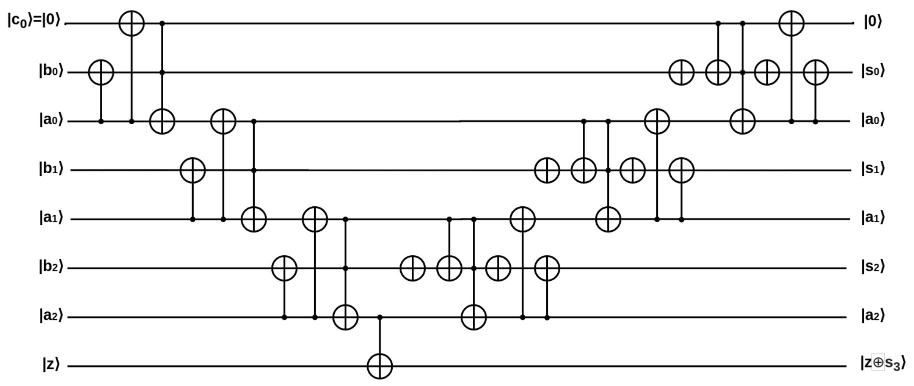


Figure 2.11: Integer quantum adder with three-qubit operands using UMA gates of type 3-CX(without simplifications) - Cuccaro, Draper, Moulton, Kutin

The previous implementation can be simplified in order to reduce both the circuit depth and the number of quantum gates. Below all the simplifications are detailed.

Optimization 1 The first optimization is to move the first CNOT of the MAJ gates to the beginning of the circuit and the last CNOT of the UMA gates to the end of the circuit, as shown in Figure 2.12.

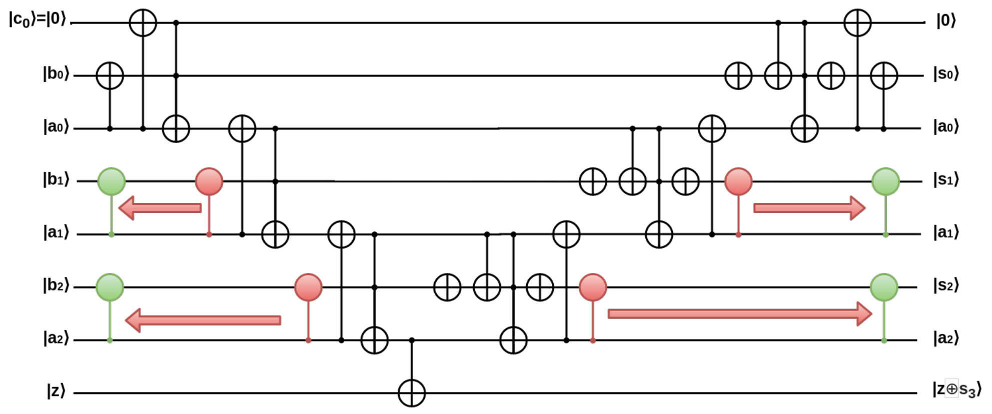


Figure 2.12: First optimization (Cuccaro, Draper, Moulton, Kutin)

In Figure 2.12 it is now possible to identify two groups of CNOT gates aligned with each other, one at the beginning of the circuit and one at the end. It is therefore possible to perform each group of CNOT gates in a single time-slice[7],

[7]It is supposed to work on a quantum computer that can run an arbitrary number of gates in parallel.

without changing the overall behavior of the adder.

Optimization 2 The second possible optimization consists in properly ordering the gates of the MAJ components. Specifically, the order of the Toffoli gate and of the CNOT gate of two consecutive MAJ components can be inverted as shown in Figure 2.13.

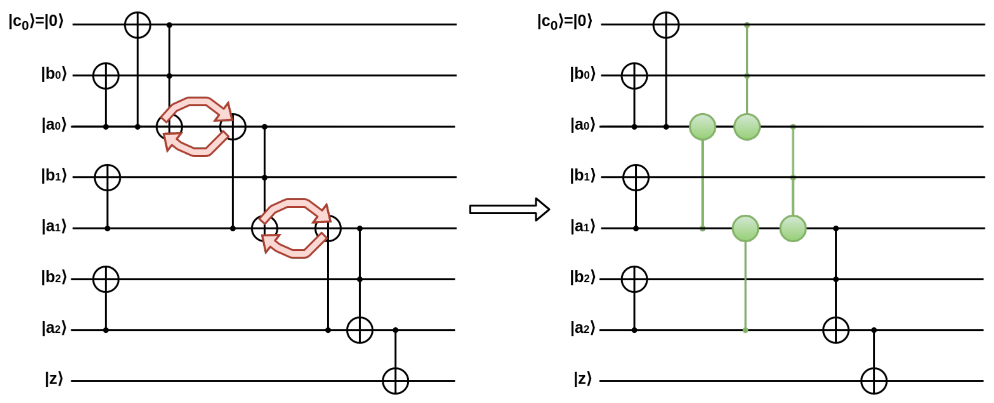


Figure 2.13: Second optimization (Cuccaro, Draper, Moulton, Kutin)

The improvement consists in the fact that now two gates - Toffoli and CNOT - are aligned with each other, and it is therefore possible to execute them simultaneously.

Optimization 3 This optimization is similar to the previous one. The difference is that, in this case, gates of UMA components are swapped.

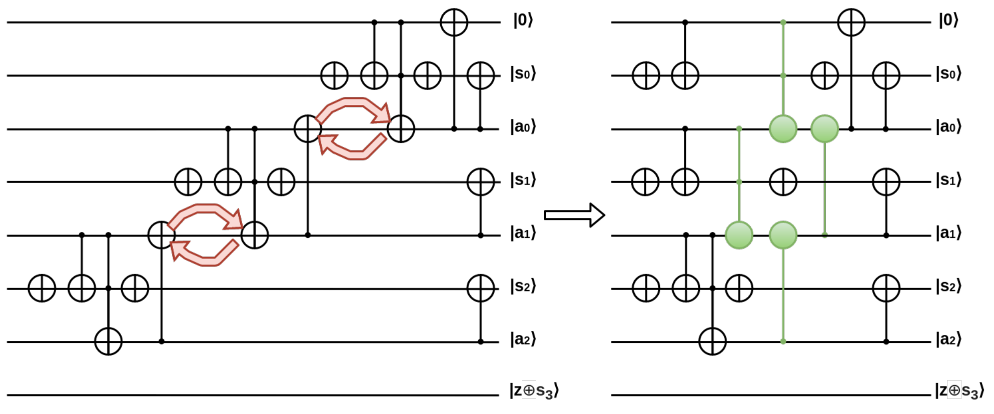


Figure 2.14: Third optimization (Cuccaro, Draper, Moulton, Kutin)

Similarly to what observed in the previous optimization, also in this case it is possible to simultaneously execute a Toffoli gate and a CNOT gate.

Optimization 4 Taking into account the fact that the aim is to implement an adder where the carry in is always zero, it is possible to simplify the structures of the first MAJ component and of the last UMA component as shown in Figure 2.15.

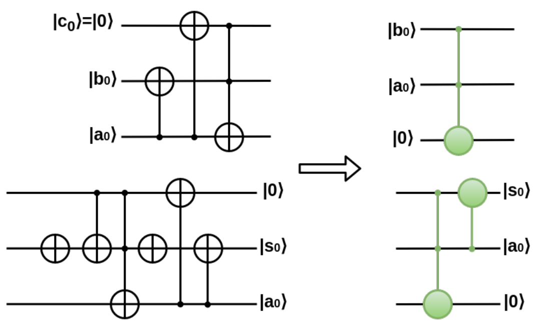


Figure 2.15: Fourth optimization (Cuccaro, Draper, Moulton, Kutin)

In this case, in addition to improving the depth of the circuit, the cost in terms of number of gates is also improved, as it is possible to remove from the circuit two X gates and four CNOT gates.

Optimization 5 The last optimization that can be carried out concerns the central part of the circuit, which can be simplified as shown in Figure 2.16.

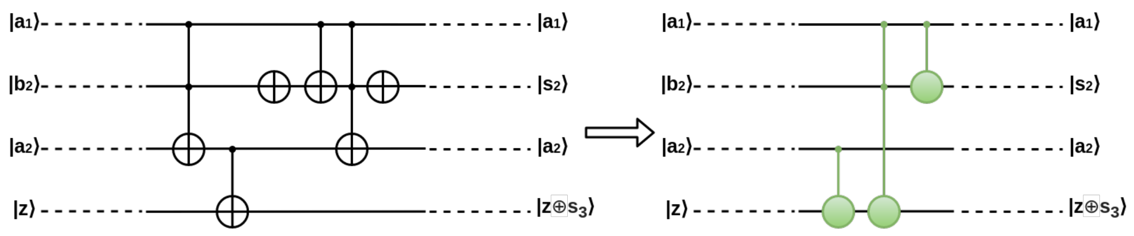


Figure 2.16: Fifth optimization (Cuccaro, Draper, Moulton, Kutin)

This optimization saves two X gates and one Toffoli gate.

Optimized circuit Having applied all the previous optimizations, the circuit becomes as in Figure 2.17.

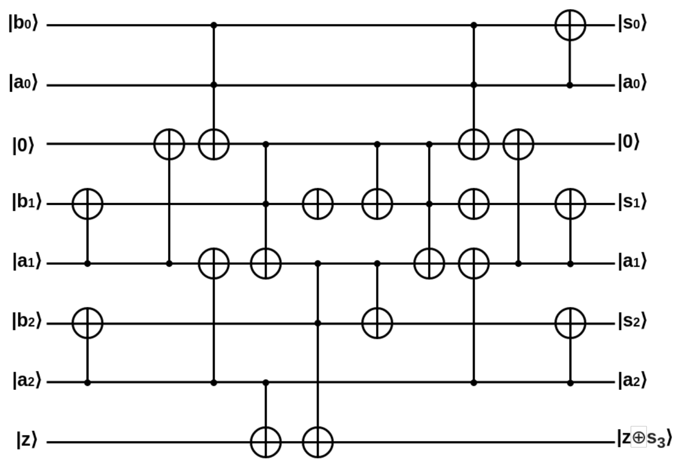


Figure 2.17: Integer quantum adder optimized (Cuccaro, Draper, Moulton, Kutin)

It is noted that, for sake of simplicity, all the simplifications have been applied to a three-qubit operands version of the adder, but the procedure can be easily generalized to the case of an adder with parallelism N.

Adder with carry in

To manage a possible carry in, it is possible to reuse the structure seen above, as long as the optimization number four is not applied.

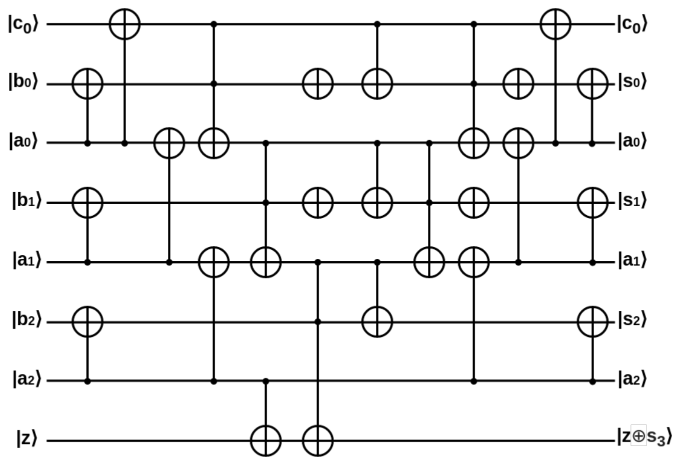


Figure 2.18: Integer quantum adder with carry in (Cuccaro, Draper, Moulton, Kutin)

Key features

The tables shown below summarize the main characteristics of the two adders proposed in [2].

Table 2.2: No. of qubits (Cuccaro, Draper, Moulton, Kutin adder)

Carry in version	Operands A and B	Carry in	Carry out	Ancilla	Tot.
No	2n	0	1	1	2n+2
Yes	2n	1	1	0	2n+2

Table 2.3: Implementation cost (Cuccaro, Draper, Moulton, Kutin adder)

Carry in version	No. X gates	No. CX gates	No. Toffoli gates
No	2n-4	5n-3	2n-1
Yes	2n-2	5n+1	2n-1

Table 2.4: Circuit depth (Cuccaro, Draper, Moulton, Kutin adder)

Carry in version	CX depth	Toffoli depth	Total depth
No	5	2n-1	5*CX$_{weight}$+(2n-1)*CCX$_{weight}$
Yes	7	2n-1	7*CX$_{weight}$+(2n-1)*CCX$_{weight}$

Integer quantum adder - Takahashi, Tani, Kunihiro

In [4] an adder without carry in is proposed. This adder can be implemented following a series of six steps, as highlighted in Figure 2.19, where a three-qubit adder is implemented.

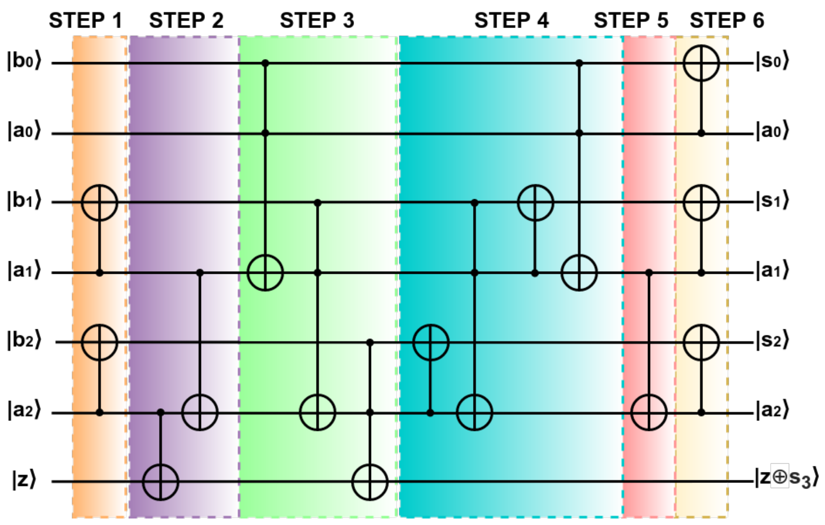


Figure 2.19: Integer quantum adder: Takahashi, Tani, Kunihiro

In order to show the steps to follow to build an adder of a generic parallelism n, assume that all qubits are stored within the same quantum register q. If the number of qubits of each operand is n, considering the additional qubit necessary for the carry out, the adder is characterized by a total number of qubits equal to $2n + 1$. The quantum register can therefore be considered as follows[8]:

$$qreg\ q[2n + 1];$$

where:

- Addend A: q[i] with i(odd) ∈ [1,2n-1]
- Addend B: q[i] with i(even) ∈ [0,2n-2]
- Carry out: q[2n]

The steps necessary to implement an adder with operands of n qubits are detailed below.

[8]From here on, unless otherwise indicated, a pseudo OpenQASM notation will be used to explain the various algorithms.

Step 1 Apply $n-1$ CNOT gates such as:[9]

$$cx\ q[i]\ q[i-1]; \quad with\ \ \text{i(odd) from 3 to } 2n-1$$

Step 2 Apply $n-1$ CNOT gates such as:

$$cx\ q[2n-1]\ q[2n];$$
$$cx\ q[i]\ q[i+2]; \quad with\ \ \text{i(odd) from } 2n-3 \text{ to } 3$$

Step 3 Apply n Toffoli gate as follows:

$$ccx\ q[i]\ q[i+1]\ q[i+3]; \quad with\ \ \text{i(even) from 0 to } 2n-4$$
$$ccx\ q[2n-2]\ q[2n-1]\ q[2n];$$

Step 4 Apply $n-1$ CNOT gates and $n-1$ Toffoli gates as follows:

```
for $i$(odd) from $2n-1$ to 3:
    $cx\ q[i]\ q[i-1];$
    $ccx\ q[i-3]\ q[i-2]\ q[i];$
```

Step 5 Apply $n-2$ CNOT gates such as:

$$cx\ q[i]\ q[i+2]; \quad with\ \ \text{i(odd) from 3 to } 2n-3$$

Step 6 Apply n CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with\ \ \text{i(odd) from 1 to } 2n-1$$

Key features

The key features of this adder are presented in the following.

[9] The index i can range from 0 to 2n.

Table 2.5: No. of qubits (Takahashi, Tani, Kunihiro adder)

Operands A and B	**Carry in**	**Carry out**	**Ancilla**	**Tot.**
2n	0	1	0	2n+1

Table 2.6: Implementation cost (Takahashi, Tani, Kunihiro adder)

No. X gates	**No. CX gates**	**No. Toffoli gates**
0	5n-5	2n-1

Table 2.7: Circuit depth (Takahashi, Tani, Kunihiro adder)

CX depth	**Toffoli depth**	**Total depth**
3n-2	2n-1	(3n-2)*CX$_{weight}$+(2n-1)*CCX$_{weight}$

Integer quantum adder - Thapliyal, Ranganathan

In [7], two adders - with and without carry in - are proposed.
Before showing the key features of these two kinds of adders, it is necessary to explain some new quantum gates which work like building blocks for the two circuits.

Controlled square root of X gate The square root of X ($\sqrt{X}$) is a quantum gate which owes its name to the following properties:

$$\sqrt{X} \cdot \sqrt{X} = X$$
$$\sqrt{X^\dagger} \cdot \sqrt{X^\dagger} = X$$
$$\sqrt{X} \cdot \sqrt{X^\dagger} = \sqrt{X^\dagger} \cdot \sqrt{X} = I$$

Possible implementations for the controlled square root of X and for its inverse are shown in Figures 2.20 and 2.21 respectively.

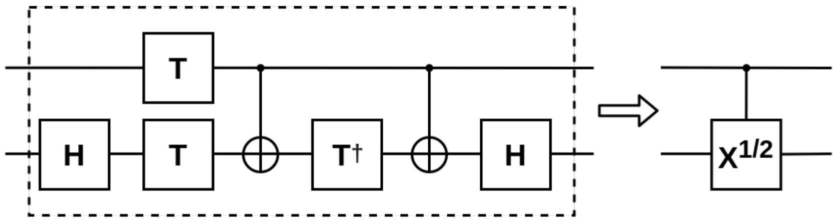


Figure 2.20: Controlled square root of X gate: a possible implementation

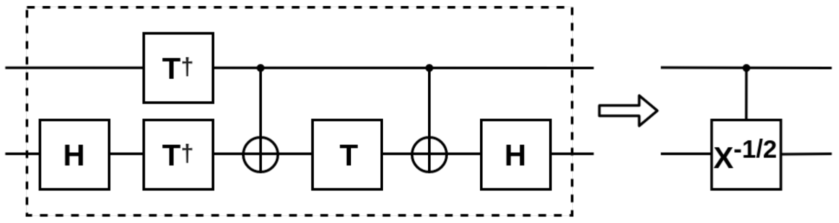


Figure 2.21: Controlled square root of X gate (dag): a possible implementation

Peres gate The Peres gate is a three inputs/three outputs quantum gate that can be considered universal in the *classical domain*, since it allows to implement NOT and AND, thus any logic function. A possible implementation is shown in Figure 2.22.

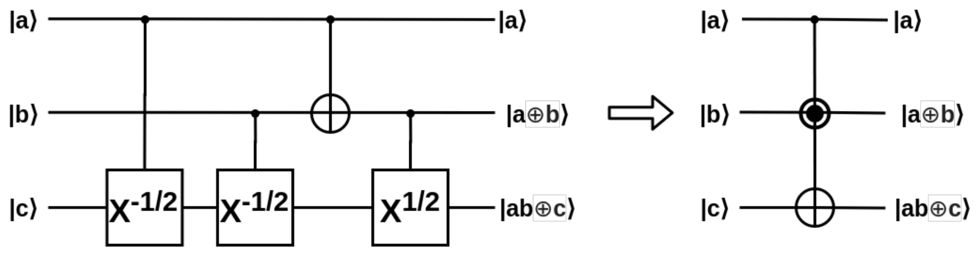


Figure 2.22: Peres gate

TR gate The TR gate, proposed in [5] by Thapliyal and Ranganathan, is a three inputs/three outputs gate. The implementation of this gate is shown in Figure 2.23.

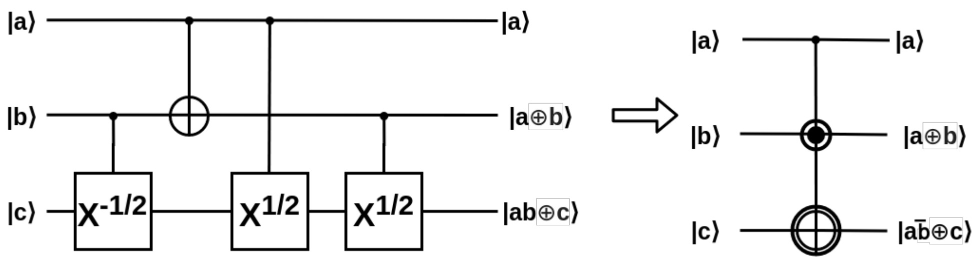


Figure 2.23: TR gate

This gate can be very useful when designing quantum arithmetic circuits since it allows to perform the following three different functions:

$$a \oplus b \qquad a \cdot \overline{b} \oplus c \qquad a \cdot \overline{b} \text{ if } c \text{ is set to } 0$$

Adder without carry in

The aim is now to illustrate the generic algorithm which allows to build the adder in question. To do this with greater clarity, a version with three-qubit operands of the adder itself is proposed in Figure 2.24.

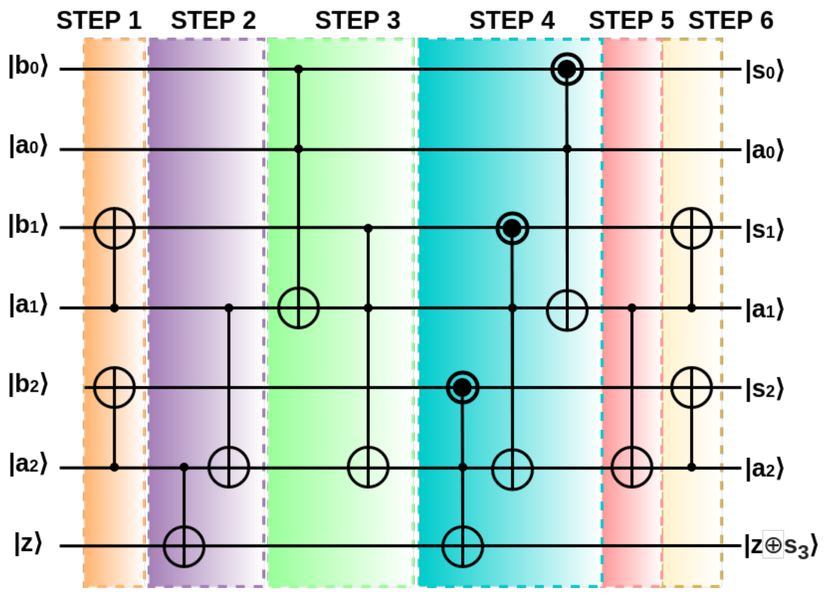


Figure 2.24: Integer quantum adder without carry in (Thapliyal, Ranganathan)

To show the steps to follow to build an adder of a generic parallelism n, assume that all qubits are stored within the same quantum register q. If the number of qubits of each operand is n, considering the additional qubit necessary for the carry out, the adder is characterized by a total number of qubits equal to $2n + 1$. The quantum register can therefore be considered as follows[10]:

$$qreg\ q[2n+1];$$

where:

- Addend A: q[i] with i(odd) $\in$ [1,2n-1]
- Addend B: q[i] with i(even) $\in$ [0,2n-2]
- Carry out: q[2n]

The necessary steps are the following.

Step 1 Apply $n-1$ CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with\ \text{ i(odd) from 3 to } 2n-1$$

Step 2 Apply $n-1$ CNOT gates such as:

$$cx\ q[2n-1]\ q[2n];$$
$$cx\ q[i]\ q[i+2]; \quad with\ \text{ i(odd) from } 2n-3 \text{ to } 3$$

Step 3 Apply $n-1$ Toffoli gate as follows:

$$ccx\ q[i]\ q[i+1]\ q[i+3]; \quad with\ \text{ i(even) from 0 to } 2n-4$$

Step 4 Apply n Peres[11] gates as follows:

$$prs\ q[2n-1]\ q[2n-2]\ q[2n];$$
$$prs\ q[i+1]\ q[i]\ q[i+3]; \quad with\ \text{ i(even) from } 2n-4 \text{ to } 0$$

[10]The index i can range from 0 to 2n.

[11]The Peres gate is not a native OpenQASM gate. Here it is called 'prs' since this is the name used within the developed quantum arithmetic library.

Step 5 : Apply $n-2$ CNOT gates such as:

$$cx\ q[i]\ q[i+2]; \quad with \ \text{i(odd) from 3 to } 2n-3$$

Step 6 Apply $n-1$ CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with \ \text{i(odd) from 3 to } 2n-1$$

Adder with carry in

In Figure 2.25 is presented a three-qubit version of the adder which is able to handle a carry in.

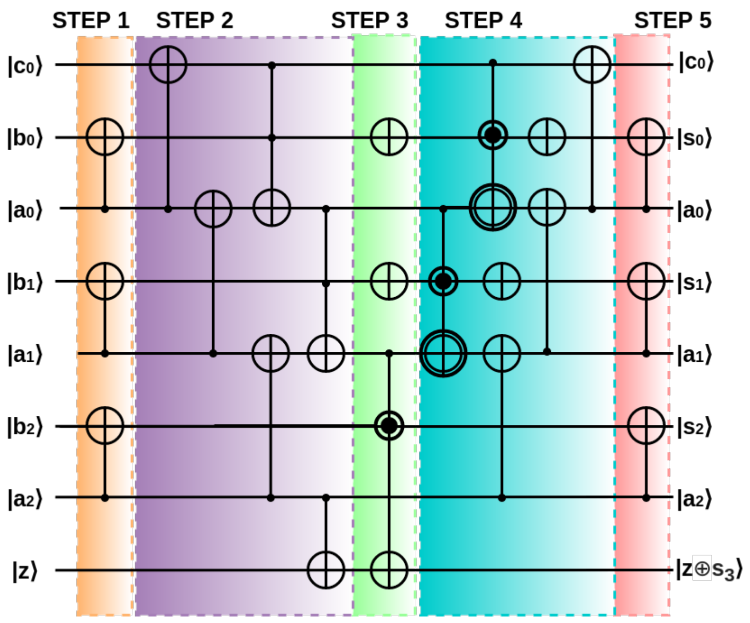


Figure 2.25: Integer quantum adder with carry in (Thapliyal, Ranganathan)

Similarly to what has been done for the version without carry in of the adder, it is possible to proceed showing the steps necessary to build this circuit. Before proceeding, it should be noted that, in this case, the number of qubits is $2n+2$, as an additional qubit to handle the carry in is needed this time. So, the generic quantum register can be written as[12]:

$$qreg\ q[2n+2];$$

where:

- Carry in: q[0]
- Addend A: q[i] with i(even) $\in$ [2,2n]
- Addend B: q[i] with i(odd) $\in$ [1,2n-1]
- Carry out: q[2n+1]

The steps to follow to build a generic adder are as follows.

Step 1 Apply n CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with\ \text{ i(even) from 2 to } 2n$$

Step 2 Apply $n+1$ CNOT gates and $n-1$ Toffoli gates such as:

$$cx\ q[i]\ q[i-2]; \quad with\ \text{ i(even) from 2 to } 2n$$
$$cx\ q[2n]\ q[2n+1];$$
$$ccx\ q[i]\ q[i+1]q[i+2]; \quad with\ \text{ i(even) from 0 to } 2n-4$$

Step 3 Apply $n-2$ X gates and a Peres gate as follows:

$$x\ q[i]; \quad with\ \text{ i(odd) from 1 to } 2n-3$$
$$prs\ q[n]\ q[2n-1]\ q[2n+1];$$

[12]This time the index i can range from 0 to 2n+1.

Step 4 Apply $n-1$ TR[13] gates, $n-1$ X gates and n CNOT gates as follows:

$tr\ q[i]\ q[i+1]\ q[i+2];$ *with* i(even) from $2n-4$ to 0

$x\ q[i];$ *with* i(odd) from $2n-3$ to 1

$cx\ q[i]\ q[i-2];$ *with* i(even) from $2n$ to 2

Step 5 Apply n CNOT gates such as:

$cx\ q[i]\ q[i-1];$ *with* i(even) from 2 to $2n$

Key features

Now that the reader has an idea of the "new" gates used to build these adders and of the adders themselves, it is possible to show the key features.

Table 2.8: No. of qubits (Thapliyal, Ranganathan adder)

Carry in version	**Operands A and B**	**Carry in**	**Carry out**	**Ancilla**	**Tot.**
No	2n	0	1	0	2n+1
Yes	2n	1	1	0	2n+2

Table 2.9: Implementation cost (Thapliyal, Ranganathan adder)

Carry in version	**No. X gates**	**No. CX gates**	**No. Toffoli gates**
No	0	4n-5	n-1
Yes	2n-2	4n+1	n-1

[13]The TR gate is not a native OpenQASM gate. Here it is called 'tr' since this is the name used within the developed quantum arithmetic library.

Table 2.10: Implementation cost (cont'd) (Thapliyal, Ranganathan adder)

Carry in version	**Peres gates**	**TR gates**
No	n	0
Yes	1	n-1

Table 2.11: Circuit depth (Thapliyal, Ranganathan adder)

Carry in version	**CX depth**	**Toffoli depth**	**Peres depth**	**TR depth**
No	2n-1	n-1	n	0
Yes	5	n-1	1	n-1

Table 2.12: Circuit depth (cont'd) (Thapliyal, Ranganathan adder)

Carry in version	**Total depth**
No	(2n-1)*CX$_{weight}$+(n-1)*CCX$_{weight}$+(n-1)*Peres$_{weight}$+0*TR$_{weight}$
Yes	5*CX$_{weight}$+(n-1)*CCX$_{weight}$+1*Peres$_{weight}$+(n-1)*TR$_{weight}$

2.4 Subtraction

The subtractors implemented in the proposed quantum arithmetic library are based on the adder structures explained in the previous section. The mathematical relationships that allow to exploit an adder as if it were a subtractor are shown in Equation 2.3.[14]

$$\begin{cases} a - b = \overline{(\overline{a} + b)} \\ a - b = a + \overline{b} + 1 \end{cases} \tag{2.3}$$

Against this background, it is clear that it is sufficient to add properly X gates in order to obtain the subtraction between operands a and b. Therefore, without explaining again the same structures seen previously, the latter are reported directly with the modifications applied in order to obtain a subtractor.

Integer quantum subtractor - Cuccaro, Draper, Moulton and Kutin

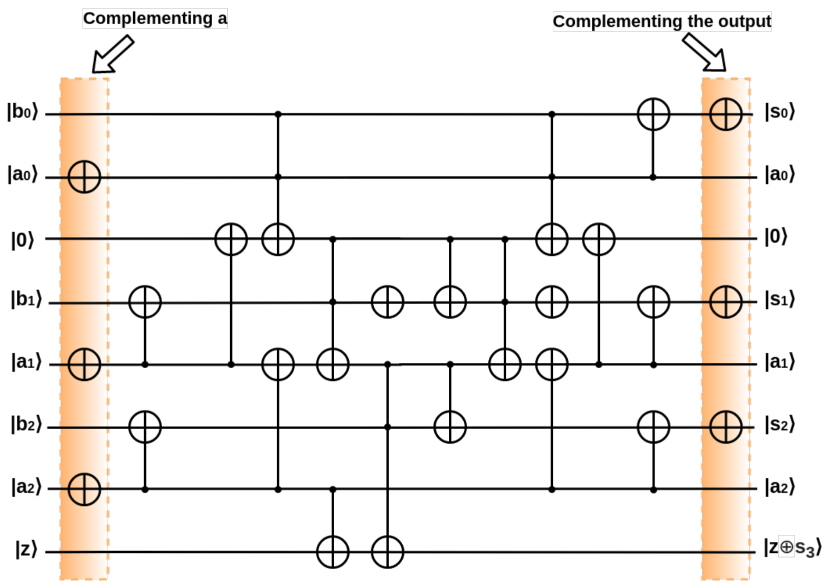


Figure 2.26: Integer quantum subtractor: $a-b = \overline{(\overline{a} + b)}$ (Cuccaro, Draper, Moulton and Kutin)

[14]The relationship $a-b = a+\overline{b}+1$ requires an adder able to handle a carry in, so the Takahashi, Tani and Kunihiro adder cannot be used in this case.

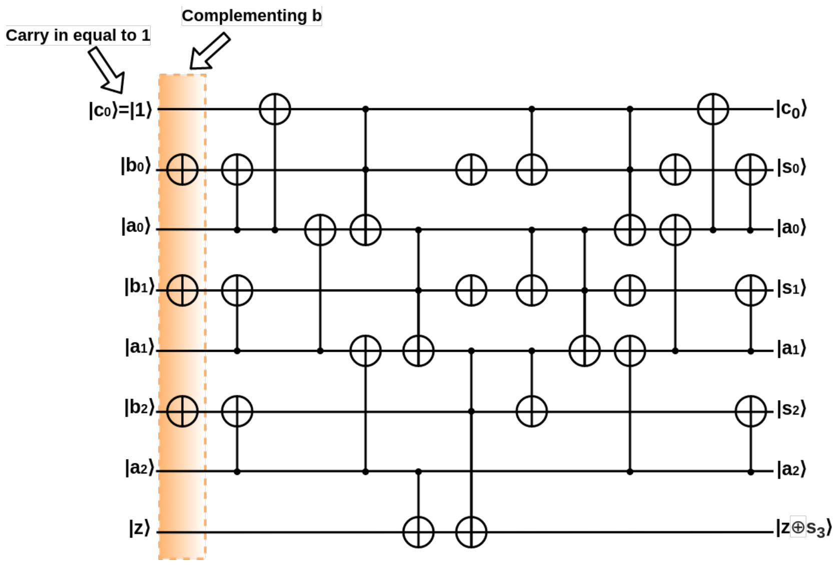


Figure 2.27: Integer quantum subtractor: $a-b = a+\overline{b}+1$(Cuccaro, Draper, Moulton and Kutin)

Integer quantum subtractor - Takahashi, Tani and Kunihiro

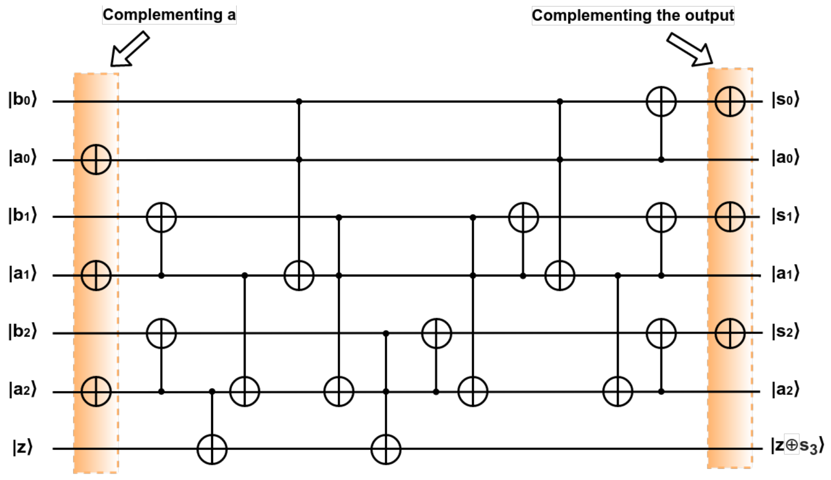


Figure 2.28: Integer quantum subtractor: $a - b = \overline{(\overline{a} + b)}$ (Takahashi, Tani and Kunihiro)

Integer quantum subtractor - Thapliyal and Ranganathan

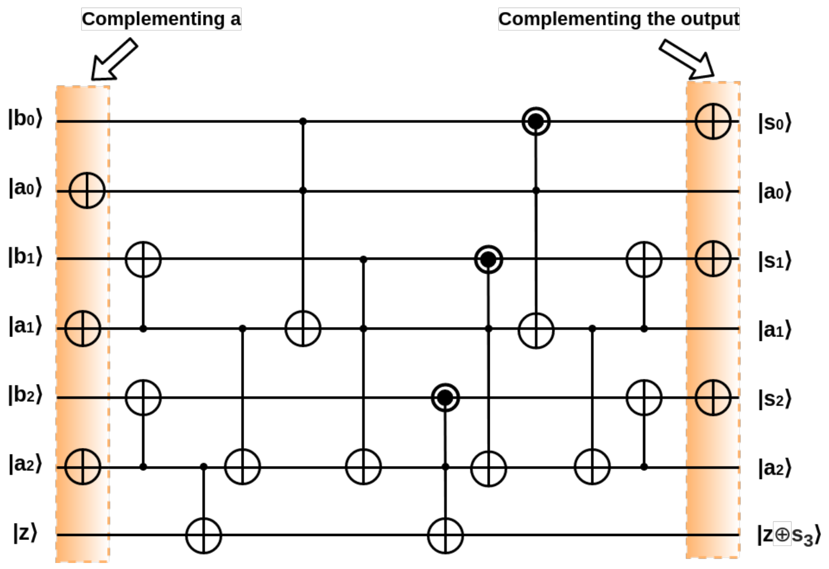


Figure 2.29: Integer quantum subtractor: $a - b = \overline{(\overline{a} + b)}$ (Thapliyal and Ranganathan)

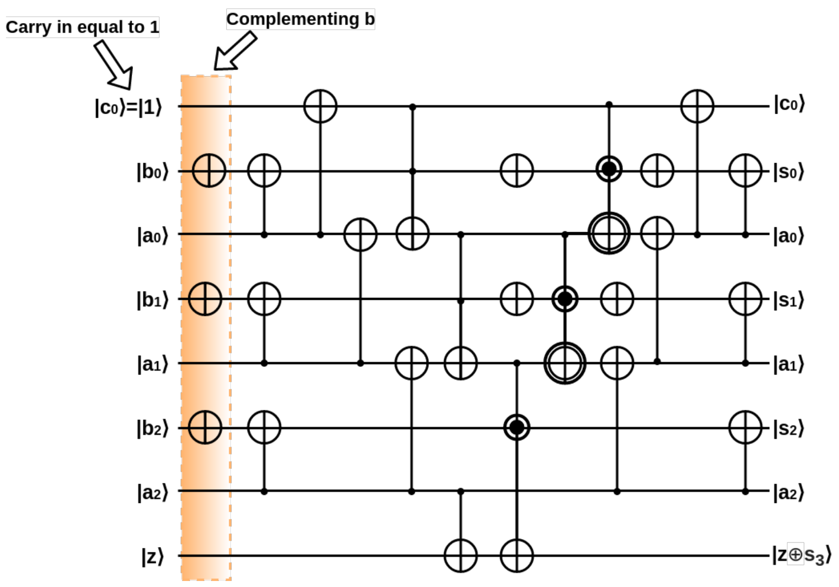


Figure 2.30: Integer quantum subtractor: $a - b = a + \overline{b} + 1$(Thapliyal and Ranganathan)

2.5 Multiplication

2.5.1 Classical computation: multiplication

When talking about binary multiplication, two operands - the multiplicand (A) and the multiplier (X) - of n bits each have to be considered. Using this notation, it is possible to express the product between A and X as:

$$A \cdot X = P \longrightarrow [a_{n-1}a_{n-2}...a_1a_0] \cdot [x_{n-1}x_{n-2}...x_1x_0] = [p_{2n-1}p_{2n-2}...p_1p_0],$$

so the final product will be a binary number on $2n$ bits. A very useful notation to represent this type of operation is the *dot notation*. An example in which two numbers of four bits each are multiplied together is shown in Figure 2.31.

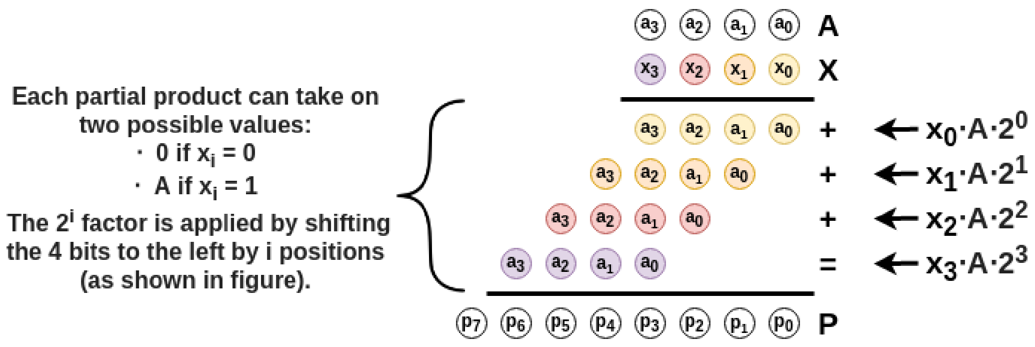

Each partial product can take on two possible values:
· 0 if $x_i = 0$
· A if $x_i = 1$
The 2^i factor is applied by shifting the 4 bits to the left by i positions (as shown in figure).
a_3 a_2 a_1 a_0 A
x_3 x_2 x_1 x_0 X
+ ← $x_0 \cdot A \cdot 2^0$
+ ← $x_1 \cdot A \cdot 2^1$
+ ← $x_2 \cdot A \cdot 2^2$
= ← $x_3 \cdot A \cdot 2^3$
p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 P

Figure 2.31: Dot notation example: multiplication

As shown in Figure 2.31, the final product between the multiplicand A and the multiplier X can be calculated as the sum of the various partial products, *i.e.*:

$$P = \sum_{i=0}^{n-1} x_i \cdot A \cdot 2^i$$

In the world of classical computation, two algorithms that allow an efficient hardware implementation of the previous formula - *i.e.* of the binary multiplication - are called R-shift and L-shift methods; however, their description is beyond the scope of this thesis.

2.5.2 Quantum computation: multiplication - advanced solutions

Integer quantum multiplier - Muñoz-Coreas, Thapliyal

A possible implementation of a quantum multiplier is proposed in [9]. This implementation mirrors what is shown in Figure 2.31: multiplier qubits $(x_{n1}..x_1x_0)$ are taken one at a time to decide whether a certain partial product $(x_i \cdot A \cdot 2^i)$ should be added or not. For this purpose, a controlled component which is able to perform the sum or not - depending on the value of the control qubit - is employed. Hence, before showing the complete structure of the multipliers and its main characteristics, it is good to illustrate how the controlled Add-Nop is implemented.

Controlled Add-Nop This component, as already mentioned, is able to perform a sum or do nothing according to the value assumed by the control qubit. Its implementation is proposed in [9]. Figure 2.32 shows an example of this components with operands of three qubits each.

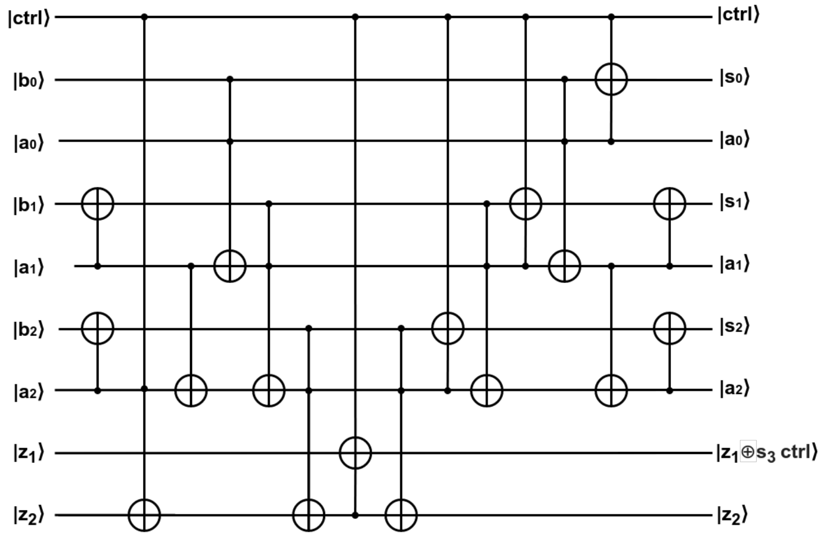


Figure 2.32: Controlled Add-Nop

Generally speaking, the number of qubits necessary to implement this circuit is $2n + 3$, where n is the operands parallelism. So, the generic quantum register can be written as[15]:

$$qreg\ q[2n+3];$$

where:

- Control: q[0]
- Addend A: q[i] with i(even) ∈ [2,2n]
- Addend B: q[i] with i(odd) ∈ [1,2n-1]
- Carry out: q[2n+1]
- Extra output qubit: q[2n+2]

The steps to follow to build a generic controlled Add-Nop are the following.

Step 1 Apply $n-1$ CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with\ \text{ i(even) from 4 to } 2n$$

Step 2 Apply a Toffoli gate and $n-2$ CNOT gates such as:

$$ccx\ q[0]\ q[2n]\ q[2n+2];$$
$$cx\ q[i]\ q[i+2]; \quad with\ \text{ i(even) from } 2n-2 \text{ to } 4$$

Step 3 Apply $n-1$ Toffoli gates as follows:

$$ccx\ q[i]\ q[i+1]\ q[i+3]; \quad with\ \text{ i(odd) from 1 to } 2n-3$$

Step 4 Apply three Toffoli gates as follows:

$$ccx\ q[2n-1]\ q[2n]\ q[2n+2];$$
$$ccx\ q[0]\ q[2n+2]\ q[2n+1];$$
$$ccx\ q[2n-1]\ q[2n]\ q[2n+2];$$

[15]This time the index i can range from 0 to 2n+2

Step 5 Apply $2n-1$ Toffoli gates as follows:

```
for i(even) from $2n$ to 4:
    $ccx\ q[0]\ q[i]\ q[i-1];$
    $ccx\ q[i-3]\ q[i-2]\ q[i];$

$ccx\ q[0]\ q[2]\ q[1];$
```

Step 6 Apply $n-2$ CNOT gates such as:

$$cx\ q[i]\ q[i+2]; \quad with\ \text{ i(even) from 4 to } 2n-2$$

Step 7 Apply $n-1$ CNOT gates such as:

$$cx\ q[i]\ q[i-1]; \quad with\ \text{ i(even) from 4 to } 2n$$

Key features

The key features of this component are illustrated below.

Table 2.13: No. of qubits (Muñoz-Coreas, Thapliyal controlled Add-Nop)

Operands A and B	**Control**	**Carry out**	**Extra qubit**	**Tot.**
2n	1	1	1	2n+3

Table 2.14: Implementation cost (Muñoz-Coreas, Thapliyal controlled Add-Nop)

No. CX gates	**No. Toffoli gates**
4n-6	3n+2

Table 2.15: Circuit depth (Muñoz-Coreas, Thapliyal controlled Add-Nop)

CX depth	Toffoli depth	Total depth
2n-3	3n+2	(2n-3)*CX$_{weight}$+(3n+2)*CCX$_{weight}$

Integer multiplier A three-qubit version of the multiplier is presented in Figure 2.33.

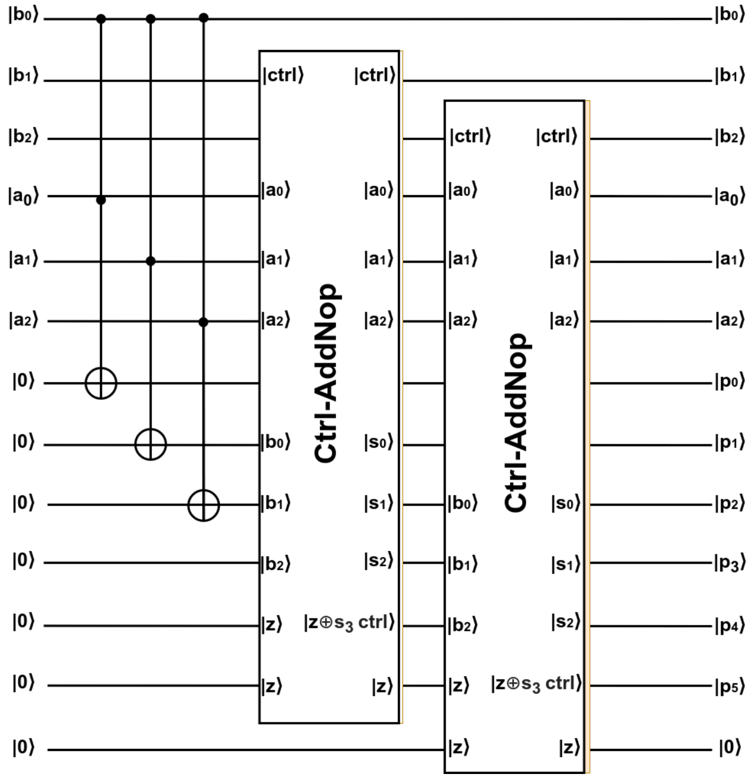


Figure 2.33: Integer quantum multiplier (Muñoz-Coreas, Thapliyal)

To show the steps to follow to build a multiplier of a generic parallelism n, assume that all qubits are stored within the same quantum register q. The required number of qubits is $4n + 1$:

- $2n$ qubits to store the two operands
- $2n$ ancilla qubits to store the final product

- 1 ancilla qubits necessary to implement the multiplier

So, the generic quantum register can be written as:

$$qreg\ q[4n+1];$$

where:

- Multiplicand A: q[i] with i ∈ [n,2n-1]
- Multiplier B: q[i] with i ∈ [0,n-1]
- Product: q[i] with i ∈ [2n,4n-1]
- Extra qubit: q[4n]

Below the steps to follow:

Step 1 Apply n Toffoli gates such as:

$$ccx\ q[0]\ q[i]\ q[i+3]; \quad with \ \text{ i from } n \text{ to } 2n-1$$

Step 2 Apply $n-1$ controlled Add-Nop components of parallelism n as follows:

$$ctrlAddNop\ q[i]\ \underbrace{q[n]...q[2n-1]}_{\text{multiplicand}}\ \underbrace{q[2n+i]...q[3n+i+1]}_{\text{partial product qubits}}; \quad with\ \text{ i(odd) from 1 to } n-1$$

Key features

The key features of this multiplier are presented below.

Table 2.16: No. of qubits (Muñoz-Coreas, Thapliyal multiplier)

Operands A and B	**Ancilla**	**Tot.**
2n	2n+1	4n+1

Table 2.17: Implementation cost (Muñoz-Coreas, Thapliyal multiplier)

No. Toffoli gates	**No. ctrlAdd-Nop**
n	n-1

Table 2.18: Circuit depth (Muñoz-Coreas, Thapliyal multiplier)

Toffoli depth	**CtrlAdd-Nop depth**	**Total depth**
n	n-1	n*CCX$_{weight}$+(n-1)*ctrlAddNop$_{weight}$

2.6 Division

2.6.1 Classical computation: division

When dealing with division in classical computation approach, the following notation can be used:[16]

- dividend Z ($2n$ bits)
- divisor D (n bits)
- quotient Q (n bits)
- remainder S (n bits)

Also in this case it is possible to use the *dot notation* in order to imagine how a possible division algorithm works. For instance, if n is equal to four, it is possible to represent the computation as shown in Figure 2.34.

[16]In quantum integer dividers presented in the following section, the operands parallelism can be different from the one shown here. However the concept still remain the same. So, do not care about these different parallelisms.

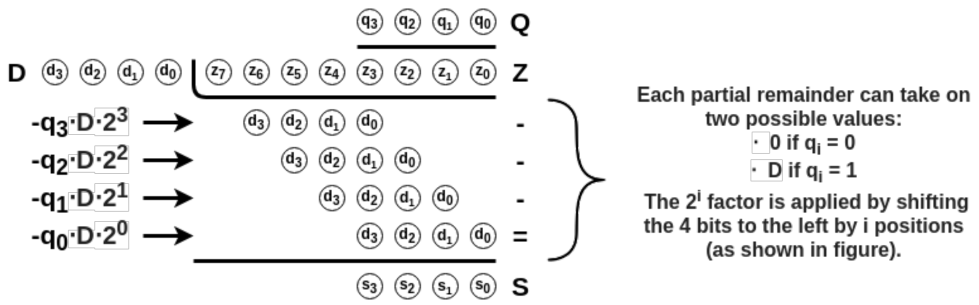

Q
D
Z
-q3·D·2^3
-q2·D·2^2
-q1·D·2^1
-q0·D·2^0
S
Each partial remainder can take on two possible values:
· 0 if qi = 0
· D if qi = 1
The 2^i factor is applied by shifting the 4 bits to the left by i positions (as shown in figure).

Figure 2.34: Dot notation example: division

Four subtractions from Z have to be performed in order to obtain the final remainder (S), that will be consisting of four digits. There is one thing to clarify. When dealing with division, two outputs have to be computed: the quotient Q and the remainder S. As the reader will have already noticed from Figure 2.34, digits of quotient Q are used as if they were known in order to calculate the remainder S. Therefore it is necessary to explain how it is possible to use the digits of Q if they are not known at the beginning of the computation. Here things are rather simple because of the fact that a binary format is involved, so every q_i, by itself, is equal either to 0 or to 1 (there are not other options). If q_i is equal to 1, a subtraction $-q_i \cdot d \cdot 2^i$ have to be performed, while if the same digit of q is equal to 0, the subtraction has to be skipped. Therefore the solution consists in making all the subtractions, and seeing from time to time which is the sign of the partial result obtained: if the sign is negative, this means that the subtraction should not have been carried out (*i.e.* $q_i = 0$); *vice versa*, if the sign of the result is positive, the subtraction had to be carried out (*i.e.* $q_i = 1$). In order to implement this mechanism, there are two ways to go:

- Restoring algorithm
- Non restoring algorithm

Restoring algorithm A restoring divider simply follows what has been explained above. It looks at the sign of the partial remainder after every calculation in order to decide if the current value of the partial remainder has to be kept or not. In other words, given a certain iteration, if the sign of the partial remainder is negative, the reverse operation is carried out in the same iteration, *i.e.* add to the partial remainder an amount equal to the subtracted amount. If the partial remainder sign is positive, everything is left as it is, without performing further operations.

Non restoring algorithm The key idea for the non-restoring algorithm is to accept a negative partial remainder, an try to compensate at a later step this kind of error by doing a slightly different operation. So, rather than avoiding the generation of a negative remainder, the idea is to temporarily accept a negative partial remainder and to compensate it in the following iteration.

2.6.2 Quantum computation: division - advanced solutions

Integer quantum dividers - Thapliyal, Muñoz-Coreas, Varun and Humble

In [10] two different methods are proposed for implementing a quantum divider: one based on the restoring algorithm, and the other based on the non restoring algorithm.

Background

Before going on with the description of the two quantum dividers, it is good to explore the building blocks used to develop the two proposed quantum dividers.

Restoring divider building blocks This divider makes use of two type of components that have already been discussed in detail in the previous sections:

- Subtractor (see Section 2.4)
- Controlled Add-Nop (see Section 2.5.2)

There is just one note to make: the controlled Add-Nop used in this divider is slightly different from the one shown in Section 2.5.2. In particular, since there is no need to manage the carry out, it is possible to reduce the depth and the implementation cost of the component. In Figure 2.35 is shown how the previously presented controlled Add-Nop is transformed when implementing the restoring divider.

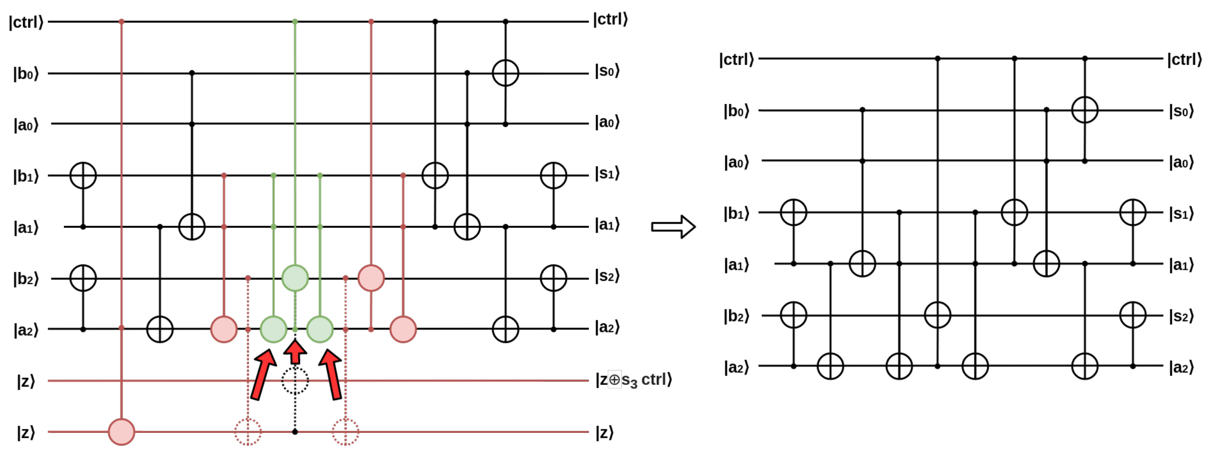


Figure 2.35: Controlled Add-Nop: restoring divider

Generally speaking, cost and performances are very similar to the controlled Add-Nop detailed in Section 2.5.2. In particular, both cost and total delay are reduced by four Toffoli gates compared to the original case.

Non restoring divider building blocks The main components used in this kind of divder are:

- Subtractor (see Section 2.4)
- Controlled Add-Nop (see Section 2.5.2)
- Controlled Add-Sub

As far as the controlled Add-Nop is concerned, the observation to be made is the same as that of the restoring case. The controlled Add-Sub, on the other hand, can be easily obtained by considering that the various subtractors presented in this library have been implemented by properly adding X gates to the adders structures, in order to satisfy one of the two Equations 2.3. Making a similar reasoning, it can be seen that, in order to obtain a controlled Add-Sub, it is possible to take an adder and to perform two steps:

- Add a control qubit to the adder structure;
- Implement one of the Equations 2.3 by applying CNOT gates (controlled by the control qubit) instead of X gates.

The divider implemented in this library makes use of the adder described in Section 2.3.4 to implement the controlled Add-Sub. An example of three-qubit controlled Add-Sub is proposed in Figure 2.36.

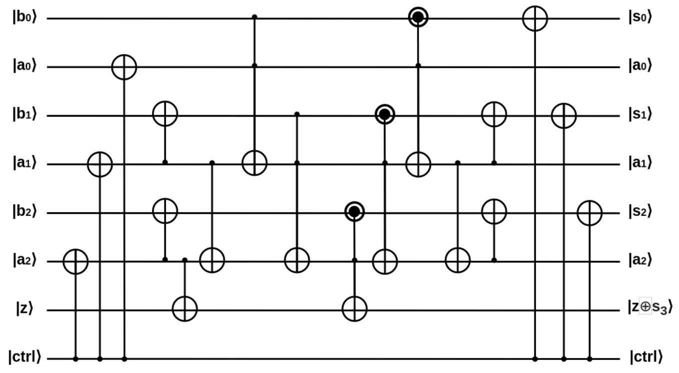


Figure 2.36: Controlled Add-Sub: non restoring divider

Without repeating all the analysis made in the case of the adder, it can be observed that - in this case - it is necessary to add to the implementation cost a number equal to 2n CNOT gates and to the overall delay a factor equal to $2n \cdot CNOT_{weight}$.

Divider: restoring algorithm

In Figure 2.37 is presented a three-qubit version of the restoring divider.

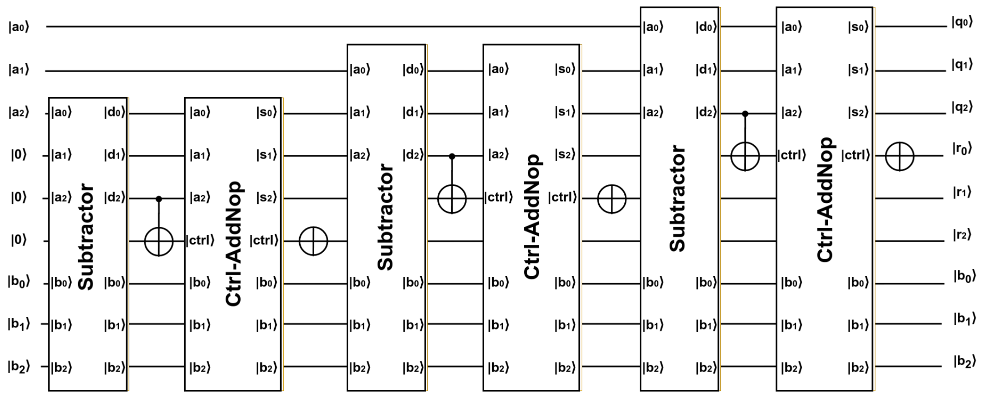


Figure 2.37: Integer quantum restoring divider (Thapliyal, Muñoz-Coreas, Varun and Humble)

To show the steps to follow to build a restoring divider of a generic parallelism n, assume that all qubits are stored within the same quantum register q. The required number of qubits is $3n$:

- $2n$ qubits to store the two operands
- n ancilla qubits to store the final quotient

So, the generic quantum register can be written as:

$$qreg\ q[3n];$$

where:

- Dividend A: q[i] with i ∈ [0,n-1]
- Ancilla: q[i] with i ∈ [n,2n-1]
- Divisor B: q[i] with i ∈ [2n,3n-1]

In this case, the implementation can be described as a single step.

Step 1 Apply n subtractors, n controlled Add-Nop, n CNOT and n X gates as follows:

```
for i from $n-1$ to 0:
    $sub\ \underbrace{q[i]...q[n-1+i]}_{\text{partial remainder}}\ \underbrace{q[2n]...q[3n-1]}_{\text{divisor}};$
    $cx\ q[i+n-1]\ q[i+n];$

    $ctrlAddNop\ \underbrace{q[i+n]}_{\text{control}}\ \underbrace{q[i]...q[n-1+i]}_{\text{partial remainder}}\ \underbrace{q[2n]...q[3n-1]}_{\text{divisor}};$
    $x\ q[i+n];$
```

Key features

The key features of this divider are presented below.

Table 2.19: No. of qubits (Thapliyal, Muñoz-Coreas, Varun and Humble restoring divider)

Operands A and B	**Ancilla**	**Tot.**
2n	n	3n

Table 2.20: Implementation cost (Thapliyal, Muñoz-Coreas, Varun and Humble restoring divider)

No. X gates	**No. CX gates**	**No. subtractors**	**No. ctrlAdd-Nop**
n	n	n	n

Table 2.21: Circuit depth (Thapliyal, Muñoz-Coreas, Varun and Humble restoring divider)

CX depth	**Subtractor depth**	**CtrlAdd-Nop depth**
n	n	n

Table 2.22: Circuit depth (Thapliyal, Muñoz-Coreas, Varun and Humble restoring divider) (cont'd)

Total depth
n*CX$_{weight}$+n*sub$_{weight}$+n*ctrlAddNop$_{weight}$

Divider: non restoring algorithm

In Figure 2.38 is presented a three-qubit version of the non restoring divider.

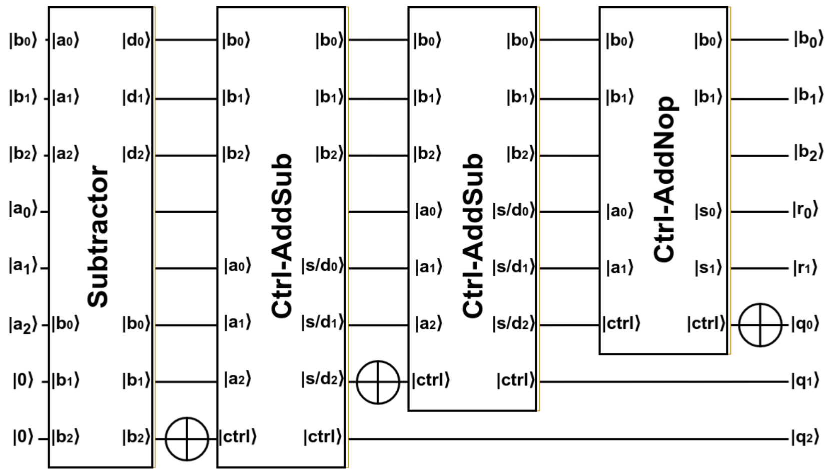


Figure 2.38: Integer quantum non restoring divider (Thapliyal, Muñoz-Coreas, Varun and Humble)

To show the steps to follow to build a non restoring divider of a generic parallelism n, assume that all qubits are stored within the same quantum register q. The required number of qubits is $3n - 1$:

- $2n$ qubits to store the two operands
- $n - 1$ ancilla qubits to store the final remainder

So, the generic quantum register can be written as:

$$qreg\ q[3n-1];$$

where:

- Dividend A: q[i] with i ∈ [n,2n-1]
- Divisor B: q[i] with i ∈ [0,n-1]
- Ancilla: q[i] with i ∈ [2n,3n-2]

Below the steps to follow:

Step 1 Apply a subtractor such as:

$$sub\ \underbrace{q[0]...q[n-1]}_{\text{divisor}}\,\underbrace{q[2n-1]...q[3n-2]}_{\text{partial remainder}};$$

Step 2 Apply $n-1$ controlled Add-Sub components of parallelism n and $n-1$ X gates as follows:

```
for i from $3n-2$ to $2n$ :
    $x\ q[i]$;
    $ctrlAddSub\ \underbrace{q[i]...q[n-1+i]}_{\text{partial remainder}}\ \underbrace{q[2n]...q[3n-1]}_{\text{divisor}}\ \underbrace{q[i]}_{\text{control}}$
```

Step 3 Apply a controlled Add-Nop of parallelism $n-1$ and a X gate such as:

```
$ctrlAddNop\ \underbrace{q[2n-1]}_{\text{control}}\ \underbrace{q[0]...q[n-1]}_{\text{divisor}}\ \underbrace{q[n]...q[2n-1]}_{\text{partial remainder}}$
$x\ q[2n-1]$;
```

Key features

The key features of this divider are presented below.

Table 2.23: No. of qubits (Thapliyal, Muñoz-Coreas, Varun and Humble non restoring divider)

Operands A and B	**Ancilla**	**Tot.**
2n	n-1	3n-1

Table 2.24: Implementation cost (Thapliyal, Muñoz-Coreas, Varun and Humble non restoring divider)

No. subtractors	**No. ctrlAdd-Nop**	**ctrlAdd-Sub**
1	1	n-1

Table 2.25: Circuit depth (Thapliyal, Muñoz-Coreas, Varun and Humble non restoring divider)

Subtractor depth	**CtrlAdd-Nop depth**	**CtrlAdd-Sub depth**
1	1	n-1

Table 2.26: Circuit depth (Thapliyal, Muñoz-Coreas, Varun and Humble non restoring divider) (cont'd)

Total depth
1*sub$_{weight}$+1*ctrlAddNop$_{weight}$+(n-1)*ctrlAddSub$_{weight}$

Chapter 3

Quantum arithmetic circuits: software library

"La mer
Qu'on voit danser le long des golfes clairs
À des reflets d'argent"
Charles Trenet

Translating algorithms for Quantum Computing into quantum circuits is a necessary operation in order to run them on real hardware. This operation requires that each algorithmic step - such as a generic arithmetic operation between two operands - is translated into a circuit description through the use of quantum gates. Therefore, in order to speed up the algorithm to circuit translation, it was decided to develop a library - the Quantum Arithmetic Library (QAL) - capable of generating OpenQASM descriptions of arithmetic operators of arbitrary parallelism.

The aim of this chapter is to give an overview about the developed software library. The first section aims to provide a general overview of the contents and organization of the library, providing a brief description of all the implemented functions. The next section, on the other hand, is intended to show the reader what this library is capable of. For this purpose, different circuits are used as an example in order to show all the steps, from the creation of the circuit to the final testing on the quantum simulator for the validation of the circuit itself. Finally, the third and last section explains how to use the blocks generated by this library in other well known Quantum Computing frameworks as Qiskit, Cirq and T-Ket, also emphasizing the fact that the circuits produced by this library can be used as building-blocks in larger circuits with a very high degree of freedom regarding the qubits connection.

3.1 An Overview of the Library

The Quantum Arithmetic Library (QAL) is made-up of several packages which allow generating OpenQASM files of arithmetic components such as adders, subtractors, multipliers and dividers. With the aim of making simpler future development, code's modularity - by organizing the general package in many subpackages - and readability - by respecting the guidelines proposed in the PEP8 style guide ([1]) - were looked for. A detailed documentation has been also produced using Sphinx. The library is organized as shown in Figure 3.1.

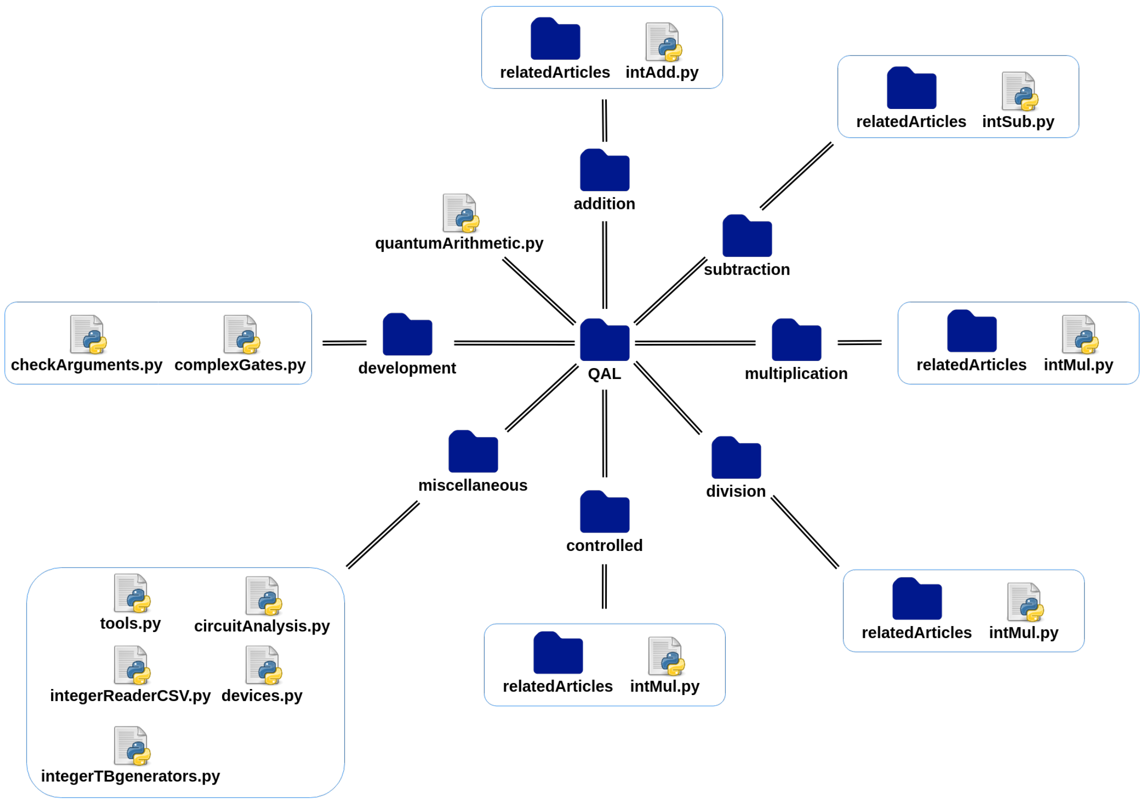


Figure 3.1: QAL: organization

It is noted that the directory "relatedArticles", if any, contains all the articles on which implemented circuits are based.
At this point it is possible to proceed with a brief description of each subpackage and its modules. It should be noted that, in this case, the description of the various modules will be very concise. For further details the reader is invited to consult the

extensive documentation produced using Sphinx. An overview of the documentation is given in the appendix B.

3.1.1 QAL.addition subpackage

This subpackage is conceived with the idea of inserting all the modules related to addition. In the current version of the library, only one module - intAdd.py - belongs to this subpackage.

intAdd.py module

The "intAdd" module contains functions capable of generating OpenQASM descriptions of integer adders. In particular, circuits presented in Section 2.3.4 have been implemented. The implemented functions are set out in Table 3.1.

Table 3.1: QAL library - "intAdd.py" module

Function	**Description**
cuccaro2004add()	It generates an OpenQASM description of the Cuccaro, Draper, Moulton, Kutin adder ([2]).
takahashi2009add()	It generates an OpenQASM description of the Takahashi, Tani, Kunihiro adder ([4]).
thapliyal2013add()	It generates an OpenQASM description of the Thapliyal, Ranganathan adder ([7]).

3.1.2 QAL.subtraction subpackage

Modules whose functions are designed to generate OpenQASM descripton of subtractors have to be placed inside this subpackage. In the current version of the library, only one module - intSub.py - belongs to this subpackage.

intSub.py module

The “intSub” module contains functions capable of generating OpenQASM descriptions of integer subtractors. In particular, circuits presented in Section 2.4 have been implemented. The available functions are set out in Table 3.2.

Table 3.2: QAL library - “intSub.py” module

Function	Description
cuccaro2004sub()	It generates an OpenQASM description of the Cuccaro, Draper, Moulton, Kutin subtractor ([2]).
takahashi2009sub()	It generates an OpenQASM description of the Takahashi, Tani, Kunihiro subtractor ([4]).
thapliyal2013sub()	It generates an OpenQASM description of the Thapliyal, Ranganathan subtractor ([7]).

3.1.3 QAL.multiplication subpackage

This subpackage is designed to collect modules related to multiplication. In the current version of the library, only one module - intMul.py - is present in this subpackage.

intMul.py module

The “intMul” module contains functions capable of generating OpenQASM descriptions of integer multipliers. In particular, circuit presented in Section 2.5.2 has been implemented. The implemented functions are set out in Table 3.3.

Table 3.3: QAL library - "intMul.py" module

Function	**Description**
coreas2017mul()	It generates an OpenQASM description of the Muñoz-Coreas, Thapliyal multiplier ([9]).

3.1.4 QAL.division subpackage

This subpackage is conceived with the idea of inserting all the modules related to the division operation inside it. In the current version of the library, only one module - intDiv.py - is present in this subpackage.

intDiv.py module

It contains functions capable of generating OpenQASM descriptions of integer dividers. In particular, circuits presented in Section 2.6.2 have been implemented. The implemented functions are set out in Table 3.4.

Table 3.4: QAL library - "intDiv.py" module

Function	**Description**
thapliyal2018divRestoring()	It generates an OpenQASM description of the Thapliyal, Muñoz-Coreas, Varun and Humble restoring divider ([10]).
thapliyal2018divNonRestoring()	It generates an OpenQASM description of the Thapliyal, Muñoz-Coreas, Varun and Humble non restoring divider ([10]).

3.1.5 QAL.controlled subpackage

This subpackage is designed to collect modules related to quantum arithmetic circuits whose behaviour depends on the value of a control qubit. In the current version of the library, only one module - intCtrl.py - belongs to this subpackage.

intCtrl.py module

The "intCtrl" module contains functions capable of generating OpenQASM descriptions of quantum arithmetic circuits able to perform operations on integer number depending on the value of a given control qubit. In particular, controlled circuits employed in multiplication and division quantum circuits have been implemented. The implemented functions are set out in Table 3.5.

Table 3.5: QAL library - "intCtrl.py" module

Function	**Description**
thapliyal2017ctrlAddNop()	It generates an OpenQASM description of the Muñoz-Coreas, Thapliyal controlled Add-Nop ([9]).
thapliyal2017ctrlAddSub()	It generates an OpenQASM description of the Thapliyal, Muñoz-Coreas, Varun and Humble controlled Add-Sub ([10]).

3.1.6 QAL.miscellaneous subpackage

This package contains modules that can be used to handle files generated by using this library and to perform circuit analysis. A fair number of modules are present in this subpackage. This is due to the fact that, while some of them have been designed to be used directly by the user, others serve as support to other modules with the aim of making the code more modular, reusable and readable. All the modules present in this subpackage are set out below.

tools.py module

The functions defined in this module can be used by the user to manipulate files generated with this library and to perform other operations such as generating a testbench version of an arithmetic circuit, remove "gate" statements from files produced by this library *etc*. All the available functions are listed in Table 3.6.

Table 3.6: QAL library - "tools.py" module

Function	**Description**
incFileMerger()	It allows to merge different .inc files
tbFormatQasmSimulator()	It allows to generate a "testbench version" of OpenQASM files produced by this library. See Section 3.2.3 for further details.
readCSVfromTB()	It allows to analyze the .csv file produced by IBM quantum experience. It is designed with the aim of analyzing a .csv file. See Section 3.2.3 for further details.
noGateVersion()	It removes "gate" statements from an OpenQASM description of a circuit generated by using this library. This may be required for compatibility reasons with other languages.

circuitAnalysis. py module

This module contains functions that allow the user to perform analysis on a quantum circuit described using the OpenQASM language. Generally speaking, functions defined in this module can also be used on circuits not generated by this library. In Table 3.7 all the available functions are listed.

Table 3.7: QAL library - "circuitAnalysis.py" module

Function	**Description**
extractAvailableQubits()	Given an OpenQASM decription where one or more quantum register are declared, it returns a list containing all the available qubits. For instance, if a three-qubit quantum register q is defined in the OpenQASM file (qreg q[3];), this function will return the following list: [q[0], q[1], q[2]]
getQubitsDependencies()	Given an OpenQASM description, it allows to find qubit dependencies, *i.e.* it is able to detect if a qubit is connected to another one and the relation control-target which exists between the two.
gateCountPerQubit()	Given an OpenQASM description, it allows to find the number and the type of gates acting on each qubit of the circuit.
longestPath()	It performs an analysis of qubits' paths and detects the most critical one.
pathsOrderedByCriticality()	It returns qubits ordered by decreasing delay.

devices.py module

Typically quantum computers parameters (*e.g.* decoherence time, relaxation time, *etc.*) are updated daily. In order to allow the user to save these parameters on file and to retrieve them at a later date, the library provides the "device" module, containing functions capable of generating *device files* and reading information from these files. The available functions are set out in Table 3.8. It is noted that - as with the other modules - the reader is invited to consult the code documentation for further details.

Table 3.8: QAL library - “devices.py” module

Function	**Description**
creatingDeviceFile()	It can be used to generate device file.
extractDeviceParam()	It can be used to generate read a device file.

integerReadersCSV.py module

This module contains functions which support the readCSVfromTB() function analyzing an input .csv file (see Section 3.2.3). Available functions are presented in Table 3.9.

Table 3.9: QAL library - “integerReadersCSV.py” module

Function	**Description**
twoComp()	Convert a 2’s complement value to int.
integerAddersCSV()	Analyze csv files of integer adders.
integerSubtractorsCSV()	Analyze csv files of integer subtractors.
integerMultipliersCSV()	Analyze csv files of integer multipliers.
integerDividersCSV()	Analyze csv files of integer dividers.
integerAddNopCSV()	Analyze csv files of integer controlled Add-Nop.
integerAddSubCSV()	Analyze csv files of integer controlled Add-Sub.

integerTBgenerators.py module

This module contains functions which support the tbFormatQasmSimulator() function building the “testbench version” (see Section 3.2.3) of a given OpenQASM quantum circuit generated by this library. The available functions are set out in Table 3.10.

Table 3.10: QAL library - ’integerTBgenerators.py’ module

Function	**Description**
extractInfoForTBgen()	Extract information regarding quantum registers and circuit type.
integerAddersTBgen()	Testbench generator for integer adders.
integerSubtractorsTBgen()	Testbench generator for integer subtractors.
integerMultipliersTBgen()	Testbench generator for integer multipliers.
integerDividersTBgen()	Testbench generator for integer dividers.
integerControlledTBgen()	Testbench generator for integer controlled components.

3.1.7 QAL.development subpackage

This subpackage contains modules that are used by other modules in this library. It is thought for ensuring that new functionalities are properly developed. The content of this subpackage consists of two modules, detailed in the following.

checkArguments.py module

Functions defined in this module can be used to check arguments format and to extract informations from them. All the available functions are set out in Table 3.11.

Table 3.11: QAL library - "checkArguments.py" module

Function	Description
check_arguments_format()	It performs the parsing of the input arguments.
extract_qreg_names()	It returns names of passed quantum registers.
extract_qreg_contents()	It returns the contentent of passed quantum registers.
extract_qreg_lengths()	It returns the lengths of passed quantum registers.
create_qreg_and_usage()	It returns the qreg declaration and comments on qubits meaning.

complexGates.py module

Functions defined in this module can be used to generate a "gate" statement to be employed in the OpenQASM files. In particular, all the functions contained in this module return a string describing the gate to be used. The implemented functions are set out in Table 3.12.

Table 3.12: QAL library - "complexGates.py" module

Function	Description
controlledSqrtX()	It can be used to generate a gate in OpenQASM language corresponding to the controlled square root of not. By doing this, it is possible to use the "controlledSqrtX q1 q2;" statement in the qasm file.
controlledSqrtXdg()	It can be used to generate a gate in OpenQASM language corresponding to the adjoint of the controlled square root of not. By doing this, it is possible to use the "controlledSqrtXdg q1 q2;" statement in the qasm file.

Continued on next page

Table 3.12 – continued from previous page

Function	Description
prs()	It can be used to generate a gate in OpenQASM language corresponding to the Peres gate. By doing this, it is possible to use the "prs q1 q2 q3;" statement in the qasm file.
tr()	It can be used to generate a gate in OpenQASM language corresponding to the TR gate. By doing this, it is possible to use the "tr q1 q2 q3;" statement in the qasm file.
thapliyal2013sub_gate()	It can be used to generate a gate in OpenQASM language corresponding to the thapliyal2013sub gate. By doing this, it is possible to use the "thapliyal2013sub q1 q2 q3...;" statement in the qasm file.
thapliyal2017ctrlAddNop_gate()	It can be used to generate a gate in OpenQASM language corresponding to the thapliyal2017ctrlAddNop gate. By doing this, it is possible to use the "thapliyal2017ctrlAddNop q1 q2 q3...;" statement in the qasm file.
thapliyal2017ctrlAddSub_gate()	It can be used to generate a gate in OpenQASM language corresponding to the thapliyal2017ctrlAddSub gate. By doing this, it is possible to use the "thapliyal2017ctrlAddSub q1 q2 q3...;" statement in the qasm file.

3.2 User Guide

The purpose of this section is to illustrate what the QAL library allows to do. In this regard, different arithmetic circuits will be used. Note that in the following explanation, in addition to show how to generate OpenQASM descriptions of arithmetic circuits, some of the support functions that this library provides will be also discussed.

3.2.1 Arithmetic Functions: a Standard Interface

All functions capable of generating arithmetic circuits have been designed in order to have a common interface[1]. The prototype of a generic arithmetic function is shown below, so that the parameters common to all arithmetic functions implemented in the QAL library can be illustrated.

```
def arithmeticFunction(file_name, flag, reg_a=False,
↪  reg_b=False, reg_c=False, qasm=True):
```

Although all arithmetic functions have the purpose of generating an OpenQASM circuit description, through the parameters shown in the previous function definition it is possible to define different aspects such as: whether an output file should be generated or not, the number of quantum registers and the order in which qubits need to be connected. The meaning of each parameter is detailed in the following.

file_name Each arithmetic function, by default, returns a string containing the OpenQASM description of its defined circuit. Through this parameter, however, the user can decide whether to generate an output file or not. If this parameter is set to "None", no files will be generated; otherwise the name to be given to the file to be generated has to be passed.

flag, reg_a, reg_b, reg_c Generally speaking, an arithmetic circuit for Quantum Computing is characterized by two operands - A and B in the following - and by a set of extra qubits - C in the following - that can include either qubits necessary for the particular operator (*e.g.* carry in, carry out etc.) or ancilla qubits. This library allows five different possibilities to map into real quantum registers these

[1]Depending on the function, there may be some additional parameters needed to define specific characteristics of the circuit in question. For more details, see the code documentation.

three groups of qubits. All the possibilities are illustrated in the following.[2]

Case 1 A, B and C belong to the same quantum register. So, when calling the generic *arithmeticFunction*, *flag* has to be equal to 1 and *reg_a* has to be defined as follows:[3]:

```
my_circ = arithmeticFunction(file_name, 1, ['regOneName',[[[A
↪  indexes],[B indexes],[C indexes]]])
```

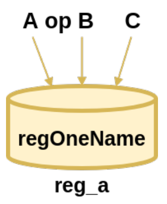


Figure 3.2: QAL: case 1

Case 2 A and B belong to the same quantum register while C belongs to another one. So, when *arithmeticFunction* is called, *flag* has to be equal to 2 and *reg_a and reg_b* have to be defined as follows[4]:

```
my_circ = arithmeticFunction(file_name, 2, ['regOneName',[[[A
↪  indexes],[B indexes]]], ['regTwoName',[C indexes]])
```

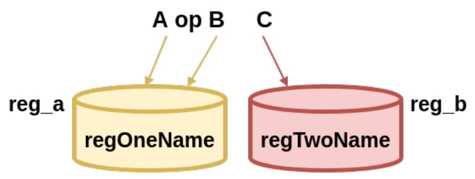


Figure 3.3: QAL: case 2

[2]How qubits indexes have to be passed will be clarified in the next few paragraphs. In the following examples a generic notation -A indexes, B indexes, C indexes - will be used.

[3]If a register is not used, it has to be omitted when calling the function.

[4]See footnote 3.

Case 3 A and C belong to the same quantum register while B belongs to another one. In this case, the call to *arithmeticFunction* has to occur by passing *flag* equal to 3 and *reg_a and reg_b* as follows[5]:

```
my_circ = arithmeticFunction(file_name, 3, ['regOneName',[[A
↪  indexes],[C indexes]]], ['regTwoName',[B indexes]])
```

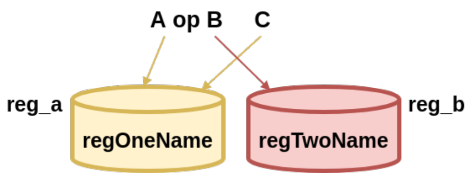


Figure 3.4: QAL: case 3

Case 4 B and C belong to the same quantum register while A belongs to another one. So, when calling *arithmeticFunction*, *flag* has to be equal to 4 and *reg_a and reg_b* have to be defined as follows[6]:

```
my_circ = arithmeticFunction(file_name, 4, ['regOneName',[A
↪  indexes]], ['regTwoName',[[B indexes],[C indexes]]])
```

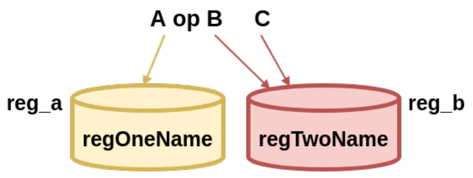


Figure 3.5: QAL: case 4

Case 5 A, B and C belong to different quantum registers. So, when *arithmeticFunction* is called, *flag* has to be equal to 5 and *reg_a, reg_b and reg_c* have to be defined as follows.

[5]See footnote 3.
[6]See footnote 3.

```
my_circ = arithmeticFunction(file_name, 5, ['regOneName',[A
↪  indexes]], ['regTwoName',[B indexes]], ['regThreeName',[C
↪  indexes]])
```

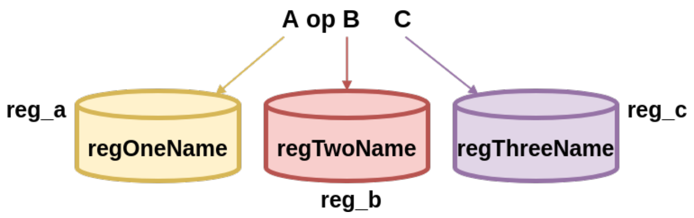


Figure 3.6: QAL: case 5

qasm As already mentioned, this library allows to both generate complete Open-QASM files, *i.e.* circuit descriptions in their own right, and to generate include files, *i.e.* files containing "gate" statements that can be imported into other projects. In this regard, the *qasm* parameter determines what it is intended to generate: if it is equal to True (the default value), a "standard" OpenQASM file will be generated, conversely, if it is set to False, an include file will be generated.

3.2.2 Basic Functionalities: Practical Examples

Now that the main parameters necessary to use the arithmetic functions have been defined, it is possible to proceed with some examples to show in detail the various functionalities of this library.

Import the library

The first thing to do in order to use the functions defined in the library is importing them. It is possible to import functions in three different ways:

```
#Import all functions
from QAL.quantumArithmetic import *

#Import a specific module
```

```python
from QAL.subpackage import module

#Import a specific function
from QAL.subpackage.module import my_function
```

Dynamic directories allocation

Since the next examples will show how to generate files and what types of files can be generated, it is good to explain where these files will be saved. The QAL library dynamically allocates only the directories needed to save the generated files. The way this happens is explained by the graph shown in Figure 3.7.

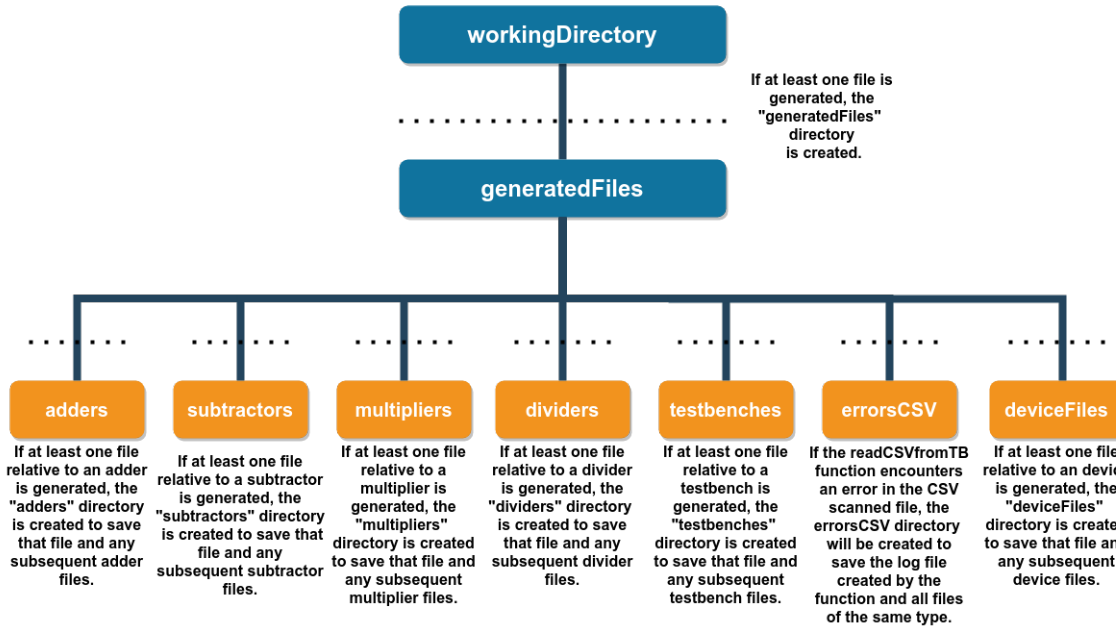

workingDirectory
If at least one file is generated, the "generatedFiles" directory is created.
generatedFiles
adders
If at least one file relative to an adder is generated, the "adders" directory is created to save that file and any subsequent adder files.
subtractors
If at least one file relative to a subtractor is generated, the "subtractors" directory is created to save that file and any subsequent subtractor files.
multipliers
If at least one file relative to a multiplier is generated, the "multipliers" directory is created to save that file and any subsequent multiplier files.
dividers
If at least one file relative to a divider is generated, the "dividers" directory is created to save that file and any subsequent divider files.
testbenches
If at least one file relative to a testbench is generated, the "testbenches" directory is created to save that file and any subsequent testbench files.
errorsCSV
If the readCSVfromTB function encounters an error in the CSV scanned file, the errorsCSV directory will be created to save the log file created by the function and all files of the same type.
deviceFiles
If at least one file relative to an device is generated, the "deviceFiles" directory is created to save that file and any subsequent device files.

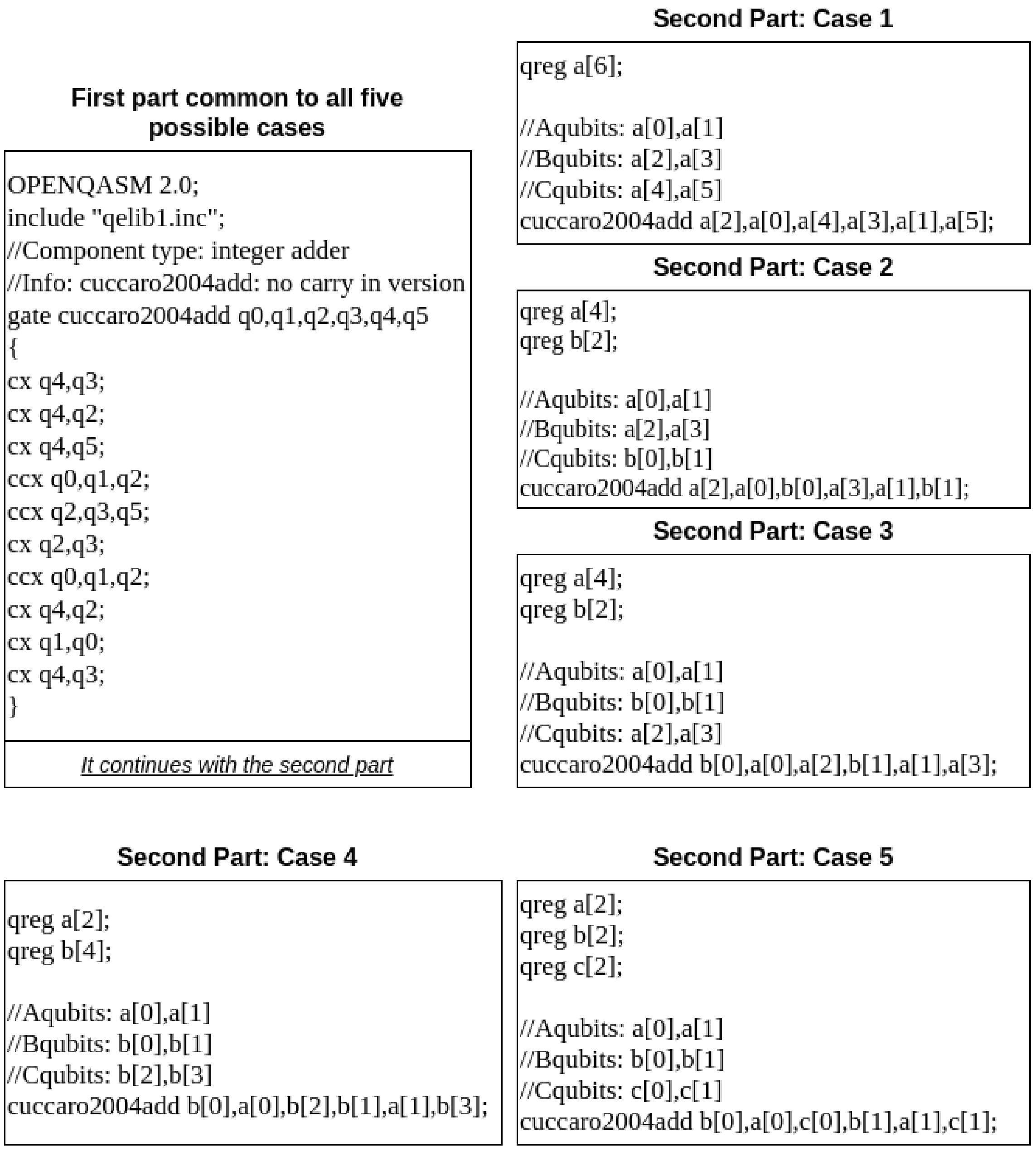
Figure 3.7: Dynamic directories allocation

Mapping operators in real quantum registers

The purpose now is to show in practice what is said in Section 3.2.1. In this regard, a two-qubit version of the adder proposed by Cuccaro, Draper, Moulton and Kutin will be used as an example. The total number of qubits involved will therefore be six: two for operand A, two for operand B and two as extra qubits (C).

```
#Case 1:
#Mapping:
# ■A[0,1]→a[0,1]   ■B[0,1]→a[2,3]   ■C[0,1]→a[4,5]
circ = cuccaro2004add("cuccaro_adder_case1",1,['a',[[0,1],[2,3],
↪  [4,5]]])

#Case 2
#Mapping:
# ■A[0,1]→a[0,1]   ■B[0,1]→a[2,3]   ■C[0,1]→b[0,1]
circ = cuccaro2004add("cuccaro_adder_case2",2,['a',[[0,1],
↪  [2,3]],['b',[0,1]])
#Case 3
#Mapping:
# ■A[0,1]→a[0,1]   ■B[0,1]→b[0,1]   ■C[0,1]→a[2,3]
circ = cuccaro2004add("cuccaro_adder_case3",3,['a',[[0,1],
↪  [2,3]],['b',[0,1]])

#Case 4
#Mapping:
# ■A[0,1]→a[0,1]   ■B[0,1]→b[0,1]   ■C[0,1]→b[2,3]
circ = cuccaro2004add("cuccaro_adder_case4",4,['a',[0,1]],['b',
↪  [[0,1],[2,3]]])

#Case 5
#Mapping:
# ■A[0,1]→a[0,1]   ■B[0,1]→b[0,1]   ■C[0,1]→c[0,1]
circ = cuccaro2004add("cuccaro_adder_case5",5,['a',[0,1]],
↪  ['b',[0,1]],['c',[0,1]])
```

The generated output file can be seen as two “independent” sections as highlighted in Figure 3.8. The first section is common in all five cases, and essentially contains the OpenQASM language header, several comments, and the general gate definition. How the second part is generated, on the other hand, depends on the case in question.



First part common to all five possible cases

```
OPENQASM 2.0;
include "qelib1.inc";
//Component type: integer adder
//Info: cuccaro2004add: no carry in version
gate cuccaro2004add q0,q1,q2,q3,q4,q5
{
cx q4,q3;
cx q4,q2;
cx q4,q5;
ccx q0,q1,q2;
ccx q2,q3,q5;
cx q2,q3;
ccx q0,q1,q2;
cx q4,q2;
cx q1,q0;
cx q4,q3;
}
```

It continues with the second part

Second Part: Case 1

```
qreg a[6];

//Aqubits: a[0],a[1]
//Bqubits: a[2],a[3]
//Cqubits: a[4],a[5]
cuccaro2004add a[2],a[0],a[4],a[3],a[1],a[5];
```

Second Part: Case 2

```
qreg a[4];
qreg b[2];

//Aqubits: a[0],a[1]
//Bqubits: a[2],a[3]
//Cqubits: b[0],b[1]
cuccaro2004add a[2],a[0],b[0],a[3],a[1],b[1];
```

Second Part: Case 3

```
qreg a[4];
qreg b[2];

//Aqubits: a[0],a[1]
//Bqubits: b[0],b[1]
//Cqubits: a[2],a[3]
cuccaro2004add b[0],a[0],a[2],b[1],a[1],a[3];
```

Second Part: Case 4

```
qreg a[2];
qreg b[4];

//Aqubits: a[0],a[1]
//Bqubits: b[0],b[1]
//Cqubits: b[2],b[3]
cuccaro2004add b[0],a[0],b[2],b[1],a[1],b[3];
```

Second Part: Case 5

```
qreg a[2];
qreg b[2];
qreg c[2];

//Aqubits: a[0],a[1]
//Bqubits: b[0],b[1]
//Cqubits: c[0],c[1]
cuccaro2004add b[0],a[0],c[0],b[1],a[1],c[1];
```

Figure 3.8: Example: cuccaro2004add(), output file

Return value

As already mentioned, all the arithmetic functions that this library provides return a string corresponding to the OpenQASM description of the circuit. The aim of this section is to emphasize that the returned value, for compatibility reasons with the employed Quantum Computing frameworks, differs from what is written into the file. Particularly, since some frameworks do not support the OpenQASM "gate" statement, a circuit description functionally identical to the one written on the file, but not using "gate" statement, is returned. In the following snippet of code, it is possible to observe what has been said. The reader is invited to make a comparison with Figure 3.8.

```
#Case 1:
circ = cuccaro2004add(None,1,['a',[[0,1],[2,3], [4,5]]])
print(circ)
```

Output:

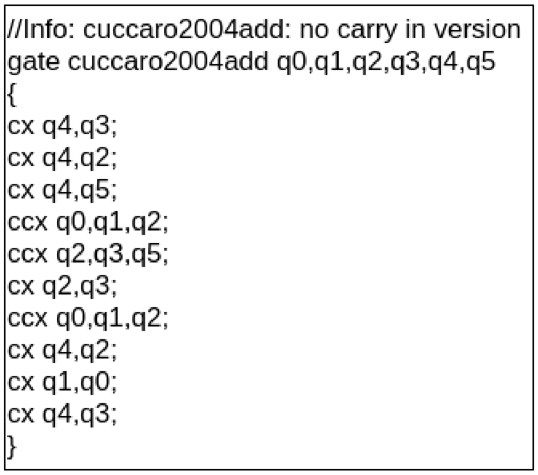
```
OPENQASM 2.0;
include "qelib1.inc";
//Component type: integer adder
//Info: cuccaro2004add: no carry in version



qreg a[6];

//Aqubits: a[0],a[1]
//Bqubits: a[2],a[3]
//Cqubits: a[4],a[5]
cx a[1],a[3];
cx a[1],a[4];
cx a[1],a[5];
ccx a[2],a[0],a[4];
ccx a[4],a[3],a[5];
```

```
cx a[4],a[3];
ccx a[2],a[0],a[4];
cx a[1],a[4];
cx a[0],a[2];
cx a[1],a[3];
```

Include file

The OpenQASM language allows to import custom libraries into project files. In this case, libraries are files containing components defined using the “gate” statement. The QAL library, as already mentioned, allows to choose whether the file or return value to be generated should be a stand-alone OpenQASM file or a library file[7]. How to generate include files is shown in the following example.

```
#Case 1:
circ = cuccaro2004add("cuccaro_adder_include",1,['a',[[0,1],
↪  [2,3],[4,5]]], qasm = False)
```



```
//Info: cuccaro2004add: no carry in version
gate cuccaro2004add q0,q1,q2,q3,q4,q5
{
cx q4,q3;
cx q4,q2;
cx q4,q5;
ccx q0,q1,q2;
ccx q2,q3,q5;
cx q2,q3;
ccx q0,q1,q2;
cx q4,q2;
cx q1,q0;
cx q4,q3;
}
```

Figure 3.9: Example: cuccaro2004add(), include file

[7]In the case of include files there is no difference between the generated file and the return value.

It is noted that when a library file is generated, the called function - in addition to generate the desired file - prints to standard output how that component must be invoked to have the qubits mapped as specified in the parameters passed to the function. Referring to the previous example, the statements printed on the standard output will be the following.

Output:

```
#==========Usage==========#
include "cuccaro_adder.inc";
...
...
cuccaro2004add a[2],a[0],a[4],a[3],a[1],a[5];

#=======================#
```

3.2.3 Test on simulator

During library development, it was necessary to test the proper functioning of the implemented circuits. In this regard, instead of proceeding with a traditional approach, *i.e.* testing all possible combinations one at a time, it was decided to take advantage of the concept of "superposition of states" offered by Quantum Mechanics in order to be able to evaluate all possible inputs and their outputs in a "single shot". The strategy in question is based on three main steps:

Step 1 Modifying the Circuit Under Test to create the conditions for which all possible combinations can be evaluated at the same time, taking advantage of the principle of superposition of states;

Step 2 Running the circuit on a "quantum simulator" using the IBM quantum experience platform;

Step 3 Analyze the results.

In the following all the various steps are explained in detail.

Step 1: Testbench Generation At this stage the aim is to modify the circuit so that all possible combinations could be evaluated simultaneously. Assuming that the C.U.T is a generic arithmetic component with two-qubit operands[8], the modification made to the circuit is the one shown in Figure 3.10.

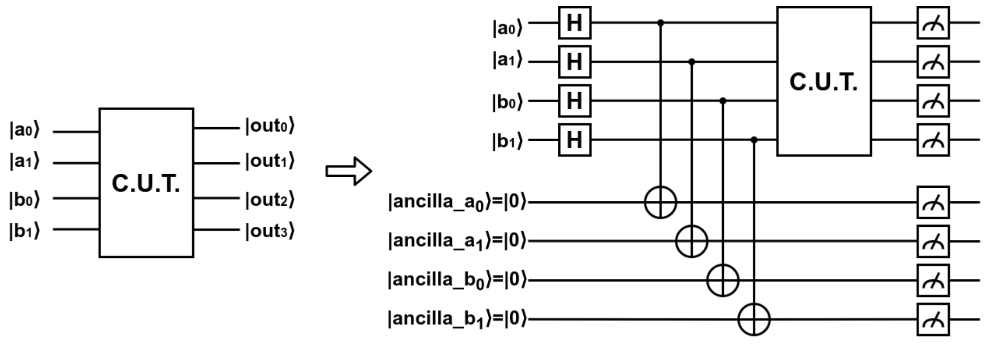


Figure 3.10: Testbench version of a C.U.T.

As shown in the previous figure, Hadamard gates, CNOT gates and measuring operators have been added to the C.U.T. Therefore, before proceeding with an analytical demonstration of the method in question, it is good to explain the role of each added element.

Hadamard gates These gates - applied to the two operands of the analyzed arithmetic circuit - aim to achieve a coherent superposition of all possible incoming values, thus giving the possibility to simultaneously evaluate all possible combinations of the inputs and their outcomes.

CNOT gates Having the need to measure the superposition of states generated by the Hadamard gates, and not being able to do so by applying measurement operators directly on the qubits related to the two operands[9], CNOT

[8]The example shows the reasoning on a circuit where operands have a parallelism of two qubits. However, this approach can be generalized to circuits of parallelism N.

[9]As explained in Section 1.2.1, the measurement operation is irreversible, *i.e.* it is not possible to measure a superposition of states without "destroying" the superposition itself. Since the aim is to use the state generated by the Hadamard gates as input for the C.U.T., there is the need to find a way to know the input state without "destroying" it.

gates were inserted with the aim of "copying" the superposition generated by Hadamard gates on ancilla qubits. It should be noted that, although the term "copying" was used to make the concept more immediate, it is not formally correct. In Quantum Mechanics, in fact, the *non-cloning theorem* states that it is not possible to "clone" an unknown quantum state a priori. However, without violating the non-cloning theorem, it is possible to use Hadamard and CNOT gates in combination to generate an entanglement between a pair of qubits, one associated to an operand register, the other related to an ancilla register. In this way the measurement value provided by the ancilla register permits to immediately know which was the corresponding entangled operand.

Measurements As visible in the Figure 3.10, measurement operators have been added so that it is possible to evaluate both the two input states and the output state.

Before proceeding with the explanation of the simulation phase, the analytical demonstration that justifies this approach is proposed in the following.

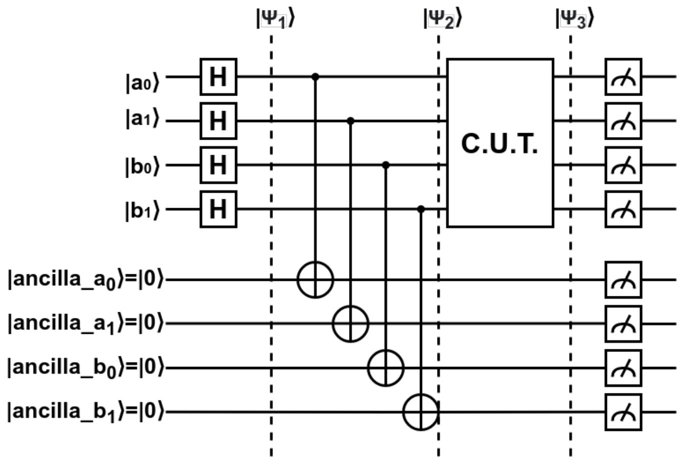


Figure 3.11: Demonstration of test methodology

Proof. Let A and B operands of N qubits each and aA and aB two ancilla registers. Each of them can therefore represent an integer number in the range $[0,2^N - 1]$.

$$\begin{cases} A = k \ \ with \ \ k \in [0,2^N - 1] \\ B = \ \ l \ \ with \ \ l \in [0,2^N - 1] \end{cases}$$

Hence the state $|\Psi_1\rangle$ (see Figure 3.11) can be expressed as[10]:

$$|\Psi_1\rangle = \frac{1}{\sqrt{2^N}}\sum_k |k\rangle_A \otimes \frac{1}{\sqrt{2^N}}\sum_l |l\rangle_B \otimes |0\rangle_{aA}^{\otimes N} \otimes |0\rangle_{aB}^{\otimes N}$$

It is possible now to calculate the $|\Psi_2\rangle$ state.

$$\begin{aligned}
|\Psi_2\rangle &= \frac{1}{\sqrt{2^N}}\sum_k |k\rangle_A |k\rangle_{aA} \otimes \frac{1}{\sqrt{2^N}}\sum_l |l\rangle_B |l\rangle_{aB} = \\
&= \frac{1}{\sqrt{2^N}}\sum_k\sum_l |k\rangle_A |l\rangle_B |k\rangle_{aA} |l\rangle_{aB} = \\
&= \frac{1}{\sqrt{2^N}}(|0\rangle_A |0\rangle_B |0\rangle_{aA} |0\rangle_{aB} + |0\rangle_A |1\rangle_B |0\rangle_{aA} |1\rangle_{aB} + \\
&\quad + |0\rangle_A |2\rangle_B |0\rangle_{aA} |2\rangle_{aB} + ... + |0\rangle_A \left|2^N-1\right\rangle_B |0\rangle_{aA} \left|2^N-1\right\rangle_{aB} + \\
&\quad + |1\rangle_A |0\rangle_B |1\rangle_{aA} |0\rangle_{aB} + |1\rangle_A |1\rangle_B |1\rangle_{aA} |1\rangle_{aB} + \\
&\quad + |1\rangle_A |2\rangle_B |1\rangle_{aA} |2\rangle_{aB} + ...)
\end{aligned}$$

Assuming that the C.U.T. acts on two generic x and y inputs with an unitary evolution U as follows:

$$U_{C.U.T.} |x\rangle_A |y\rangle_B = |f(x,y)\rangle\,,$$

state $|\Psi_3\rangle$ can be written as:

$$\begin{aligned}
|\Psi_3\rangle &= \frac{1}{2^N}\sum_k\sum_l |f(k,l)\rangle_{AB} |k\rangle_{aA} |l\rangle_{aB} = \\
&= \frac{1}{2^N}(|f(0,0)\rangle_{AB} |0\rangle_{aA} |0\rangle_{aB} + ... + \left|f(0,2^N-1)\right\rangle_{AB} |0\rangle_{aA} \left|2^N-1\right\rangle_{aB} + \\
&\quad + |f(1,0)\rangle_{AB} |1\rangle_{aA} |0\rangle_{aB} + ... + \left|f(1,2^N-1)\right\rangle_{AB} |1\rangle_{aA} \left|2^N-1\right\rangle_{aB} + ... + \\
&\quad + \left|f(2^N-1,0)\right\rangle_{AB} \left|2^N-1\right\rangle_{aA} |0\rangle_{aB} + ... + \\
&\quad + \left|f(2^N-1,2^N-1)\right\rangle_{AB} \left|2^N-1\right\rangle_{aA} \left|2^N-1\right\rangle_{aB}
\end{aligned}$$

$\square$

[10] It is noted that quantum registers are ordered from top to bottom, while their parallelism is oriented according to IBM notation, as shown in Figure 3.11.

It can be clearly observed a uniform distribution of state where ancilla registers contain the operands and the qubits related to registers A and B contain the result. It is important to precise that this parallel evaluation requires a quantum circuit with a number of qubits which is the double of that of a purely sequential test.

Step 2: Simulation This step is to simulate the "testbench version" of the circuit on a quantum simulator using the IBM quantum experience platform. The choice to use this platform arises from the fact that it allows to download a csv file containing the results of the simulation.

Step 3: Data Analysis In this last step, the results are analyzed. In particular, the aim is to verify that the output produced by each input pair is consistent with what is expected by the operator under analysis.

Test on Simulator: a Practical Example

Now that the theory behind this testing methodology should be clear, it is possible to proceed with a practical example. In particular, suppose to test the proper functioning of the multiplier proposed by Muñoz-Coreas and Thapliyal by choosing for the operands a parallelism of two qubits. This multiplier can be implemented by using the *coreas2017mul()* function.

```
coreas = coreas2017mul(None,1,['a',[[0,1],[2,3],[4,5,6,7,8]]])
print(coreas)
```

Output:

```
OPENQASM 2.0;
include "qelib1.inc";
//Component type: integer multiplier
//Info: coreas2017mul: integer multiplier
```

```
qreg a[9];

//Aqubits: a[0],a[1]
//Bqubits: a[2],a[3]
//Cqubits: a[4],a[5],a[6],a[7],a[8]
ccx a[2],a[0],a[4];
ccx a[2],a[1],a[5];
cx a[1],a[6];
ccx a[3],a[1],a[7];
ccx a[5],a[0],a[1];
ccx a[6],a[1],a[8];
ccx a[3],a[8],a[7];
ccx a[6],a[1],a[8];
ccx a[3],a[1],a[6];
ccx a[5],a[0],a[1];
ccx a[3],a[0],a[5];
cx a[1],a[6];
```

It is noted that in the previous snippet of code no file has been generated. This choice is due to the fact that the *tbFormatQasmSimulator()* function, used in the following, can accept an OpenQASM description in input or via file, or through a string. In this case, it is decided to pass the circuit description as a string. The reader is recommended to consult the documentation for more detailed information about how the functions work.
Once the multiplier file has been generated, the first step is to generate the "test-bench version" of this circuit. This is possible thanks to the *tbFormatQasmSimulator()* function in the library. The file for the multiplier in its testbench version will then appear as follows.

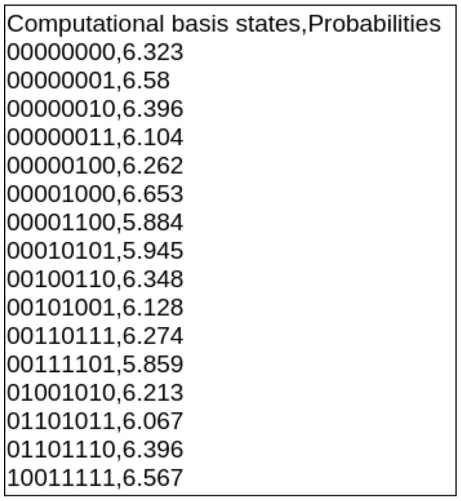
```python
coreas_tb = tbFormatQasmSimulator(False,coreas,"my_mul_tb")
print(coreas_tb)
```

Output:

```
OPENQASM 2.0;
include ”qelib1.inc”;
//Component type: integer multiplier
//Info: coreas2017mul: integer multiplier



qreg a[9];

qreg aancilla[2];
qreg bancilla[2];
creg ameas[2];
creg bmeas[2];
creg res[4];
h a[0];
h a[1];
h a[2];
h a[3];
cx a[0],aancilla[0];
cx a[1],aancilla[1];
cx a[2],bancilla[0];
cx a[3],bancilla[1];

measure aancilla -> ameas;
measure bancilla -> bmeas;

//Aqubits: a[0],a[1]
//Bqubits: a[2],a[3]
//Cqubits: a[4],a[5],a[6],a[7],a[8]
ccx a[2],a[0],a[4];
ccx a[2],a[1],a[5];
cx a[1],a[6];
```

```
ccx a[3],a[1],a[7];
ccx a[5],a[0],a[1];
ccx a[6],a[1],a[8];
ccx a[3],a[8],a[7];
ccx a[6],a[1],a[8];
ccx a[3],a[1],a[6];
ccx a[5],a[0],a[1];
ccx a[3],a[0],a[5];
cx a[1],a[6];
measure a[4]->res[0];
measure a[5]->res[1];
measure a[6]->res[2];
measure a[7]->res[3];
```

At this point, the "my_mul_tb.qasm" file, which contains the testbench version of the circuit, has been generated, so it is possible to proceed uploading the file to the IBM quantum experience platform and launching the simulation on the "ibmq_qasm_simulator". When the simulation is finished, the downloadable csv file will be as shown in Figure 3.12.



```
Computational basis states,Probabilities
00000000,6.323
00000001,6.58
00000010,6.396
00000011,6.104
00000100,6.262
00001000,6.653
00001100,5.884
00010101,5.945
00100110,6.348
00101001,6.128
00110111,6.274
00111101,5.859
01001010,6.213
01101011,6.067
01101110,6.396
10011111,6.567
```

Figure 3.12: Example: coreas2017mul(), two-qubit, simulation, csv file

Each line in the csv file, excluding the header, corresponds to one of the possible input-output combinations. In particular, reading the binary numbers in column "Computational basis states" from left to right, it can be seen that the first four digits represent the final product, while the last two pairs of digits represent the two operands.

Instead of proceeding with a visual inspection of the generated file, it is possible to use the proposed *readCSVfromTB()* function in the library to analyze the file. The function in question can assume two different behaviors depending on the analysis results: if all the results are correct, it prints a message on standard output that reports the success of the scan; conversely, if errors have been found in the csv file, an error message is printed on the standard output and a log file is also generated. In the following both cases are illustrated.

Correct behaviour The following snippet of code shows the output of the *readCSVfromTB()* function if the parsed csv file is correct.

```
err = readCSVfromTB("intmul","my_mul_results",2)
```

Output:

```
Correct behaviour.
```

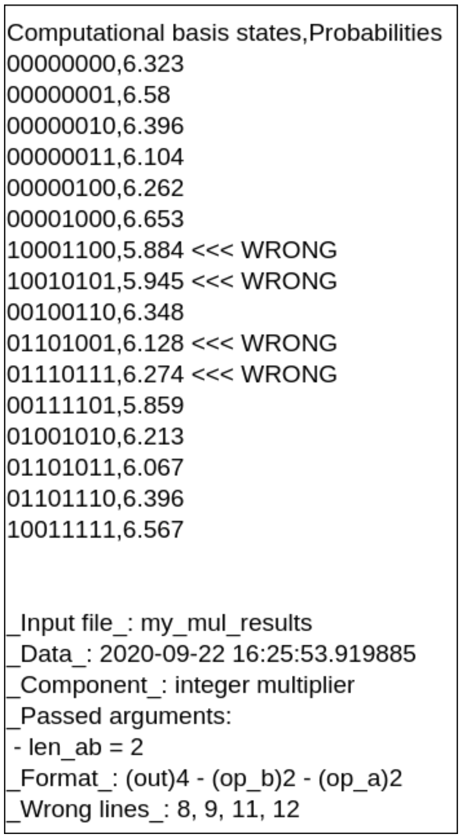
Wrong behaviour At this point, it is good to see the case in which there is an error in the csv file. To do this, suppose to change some values in the generated csv file in order to make some of the results wrong. In this case, the behaviour of the *readCSVfromTB()* function will be as follows.

```
err = readCSVfromTB("intmul","my_mul_results",2)
```

Output:

```
Errors in csv file. A .log file with details has been generated.
```

The generated log file is shown in Figure 3.13.



```
Computational basis states,Probabilities
00000000,6.323
00000001,6.58
00000010,6.396
00000011,6.104
00000100,6.262
00001000,6.653
10001100,5.884 <<< WRONG
10010101,5.945 <<< WRONG
00100110,6.348
01101001,6.128 <<< WRONG
01110111,6.274 <<< WRONG
00111101,5.859
01001010,6.213
01101011,6.067
01101110,6.396
10011111,6.567


_Input file_: my_mul_results
_Data_: 2020-09-22 16:25:53.919885
_Component_: integer multiplier
_Passed arguments:
 - len_ab = 2
_Format_: (out)4 - (op_b)2 - (op_a)2
_Wrong lines_: 8, 9, 11, 12
```

Figure 3.13: Example: coreas2017mul(), two-qubit, simulation, log file

3.2.4 Use with Other Languages

This library has been designed to be compatible with well-known frameworks related to the world of Quantum Computing, such as Qiskit, Cirq and T-ket. Below is shown how to use it in various cases. It is noted that, in each paragraph, before illustrating the example of using the QAL library with the various languages, a brief introduction is made to the languages themselves, in order to make the reader able to understand the code proposed as example.

Qiskit

Introduction Qiskit[11] is an open-source framework developed by IBM that allows to describe quantum circuits and to run them either on a simulator or on real

[11]First release in 2017.

quantum computers. It is composed by four different components (Terra, Aqua, Aer, Ignis), each of them allowing a given set of functionalities. Since the purpose of this brief introduction is only to allow the reader to correctly interpret the example code proposed in the next paragraph, in the following, a rundown of basic commands will be proposed. In general, Qiskit allows the user to generate a QuantumCircuit object and hang all the necessary gates to it. The syntax to use to create an object of type QuantumCircuit is as follows:

qc = QuantumCircuit(No.qubits, No.bits)

To define gates, instead, it is possible to proceed as follows[12]:

qc.gate(qubits)

To use the OpenQASM circuit descriptions produced by the Quantum Arithmetic Library, the statement sequence to use is like this:

#Create an OpenQASM description of the desired component

$my\_arithmetic = arithmeticFunction(...)$

#Define a QuantumCircuit using the OpenQASM description

$qc\_arithmetic = QuantumCircuit.from\_qasm\_str(my\_arithmetic)$

#Give a name to the QuantumCircuit object

$qc\_arithmetic.name = “arithmeticComponentName''$

#Translate the QuantumCircuit into an instruction

$arithmetic\_gate = qc\_arithmetic.to\_instruction()$

At this point it is possible to use the generated instruction an arbitrary number of times. To do that, proceed in this way:

qc.append(arithmetic_gate, qubits)

One final aspect concerns the measurement operators, which can be inserted into the circuit as follows:

[12]The reader shall be aware about the fact that the one shown is the most basic method for hanging gates. Actually the syntax can also be different.

qc.measure(qubits, bits)

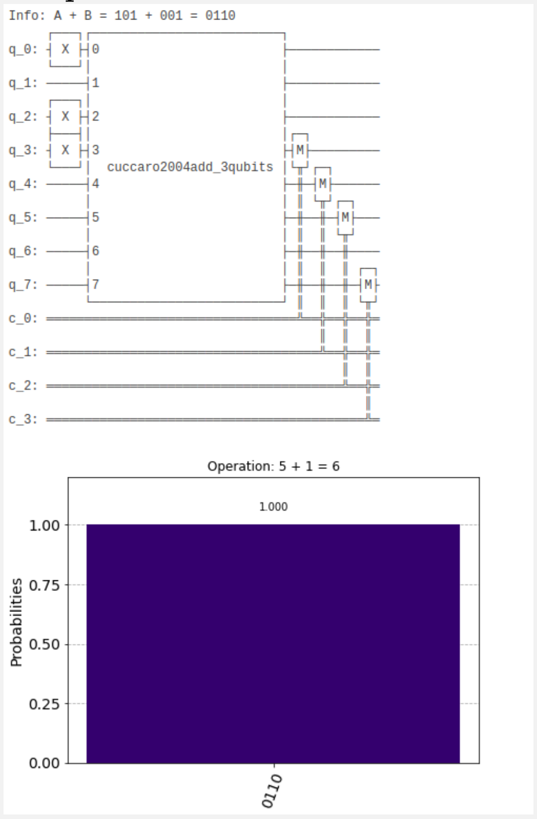
Without going into detail, it is possible to observe that the commands *get_backend, execute, result, get_counts* etc. can be used to run the circuit on "qasm_simulator" or real hardware. For more information about this framework it is suggested to see [12].

Example

```
from qiskit import (
  QuantumCircuit,
  execute,
  Aer)
from qiskit.visualization import plot_histogram
from QAL.quantumArithmetic import *

#Generating an OpenQASM description of the 3-qubits
↪  cuccaro2004adder
cuccaro_qasm =
↪  cuccaro2004add(None,1,['q',[[0,1,2],[3,4,5],[6,7]]])
qc_add = QuantumCircuit.from_qasm_str(cuccaro_qasm)
qc_add.name = "cuccaro2004add_3qubits"
cuccaro_adder = qc_add.to_instruction()

#Now it is possible to use the 3-qubits adder in our Qiskit
↪  circuit
qc = QuantumCircuit(8,4)

#Goal: Compute 101 + 001 --> Expected result: 0110
#operand A
qc.x(0)
qc.x(2)
#operand B
```

```
qc.x(3)

qc.append(cuccaro_adder,[0,1,2,3,4,5,6,7])
qc.measure([3,4,5,7],range(0,4))

# Use 'qasm_simulator'
simulator = Aer.get_backend('qasm_simulator')

# Execute the circuit on the qasm simulator and store the
↪   result
counts = execute(qc, simulator,
↪   shots=1000).result().get_counts(qc)

print("Info: A + B = 101 + 001 = 0110")
# Draw the circuit
print(qc)
# Plot the result
plot_histogram(counts, color='midnightblue', title="Operation: 5
↪   + 1 = 6")
```

Output:



```
Info: A + B = 101 + 001 = 0110
```

Figure 3.14: Example: how to use the QAL with Qiskit

Cirq

Introduction In 2018 Google announced an open source framework called Cirq, which is able to decribe quantum circuits and to run them either on simulator or on real hardware. As with Qiskit, a rundown of the basic commands needed to understand the example code proposed in the next paragraph will be put in place below. Cirq gives the user the possibility to create Circuit objects, to which qubits can be connected. To define an object of type Circuit it is possible to proceed as follows:

$$circuit = cirq.Circuit()$$

As for qubits, Cirq allows to define three different types of qubits: GridQubit, LineQubit and NamedQubit. Although all three types can in fact be used to describe qubits, each of them is more or less useful depending on the context in which it is used. Since the purpose now is not to provide detailed guidance to Cirq, it will only be shown how to declare qubits of type Named, which are what will then be used in the sample circuit. To define a NamedQubit, it is possible to proceed in this manner:

qub = cirq.NamedQubit(qubit_name)

Having defined both an object of type Circuit and objects of type qubits, it is therefore possible to insert gates inside the circuit. Although there are many possibilities to do this, the method also used in the proposed example is shown below.

circuit.append(cirq.gate_name(qubit))

It is now possible to show how an OpenQASM description produced by the QAL library can be used within a Cirq circuit description; to do this the following instructions can be used:

#Create an OpenQASM description of the desired component

my_arithmetic = arithmeticFunction(...)

#Create a custom circuit from OpenQASM description

arithmetic_gate = my_arithmetic.circuit_from_qasm(my_arithmetic)

To use the new component it is possible to proceed in this way:

circuit.append(arithmetic_gate)

Finally, to insert a measurement operator within a circuit, one of the possibilities is as follows:

circuit.append(cirq.measure(qubit))

In the code proposed as an example, the Cirq simulator will be used in its basic configuration. To do this, the following syntax will be use:

simulator = cirq.Simulator()

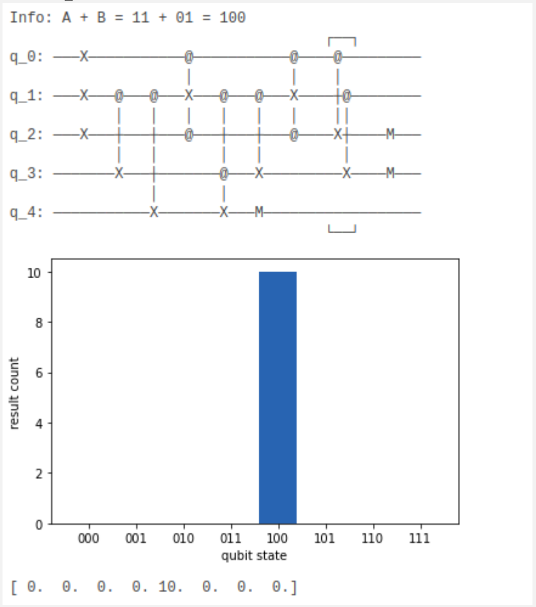
result = simulator.run(circuit,options...)

For further details the reader is invited to consult [16].

Example

```
import cirq
from cirq.contrib.qasm_import import circuit_from_qasm
from QAL.quantumArithmetic import *

#Generating an OpenQASM description of the 2-qubits
↪  takahashi2009add
takahashi_qasm =
↪  takahashi2009add(None,1,['q',[[0,1],[2,3],[4]]])
takahashi_adder = circuit_from_qasm(takahashi_qasm)

#Now it is possible to use the 2-qubits adder in our Cirq
↪  circuit
circuit = cirq.Circuit()

#The name of the circuit qubits must be compliant with the
↪  name
#of the generated arithmetic component
qub = []
for i in range(0,6):#6 is the number of qubits
        qub.append(cirq.NamedQubit(f'q_{i}'))

#Goal: Compute 11 + 01 --> Expected result: 100
#Operand A (11)
circuit.append([cirq.X(qub[0])])
circuit.append([cirq.X(qub[1])])
#Operand B (01)
circuit.append([cirq.X(qub[2])])
circuit.append(takahashi_adder)

for i in range(2,4):
        circuit.append(cirq.measure(qub[i]))
circuit.append(cirq.measure(qub[4]))

# Execute the circuit on the simulator and store the result
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=10)

print("Info: A + B = 11 + 01 = 100")
# Draw the circuit
print(circuit)
# Plot the result
print(cirq.plot_state_histogram(result))
```

Output:



```
Info: A + B = 11 + 01 = 100

                                    ┌──┐
q_0: ───X───────────@───────────@───@────────
                    │           │   │
q_1: ───X───@───@───X───@───@───X───┼@───────
            │   │   │   │   │   │   ││
q_2: ───X───┼───┼───@───┼───┼───@───X┼───M───
            │   │       │   │        │
q_3: ───────X───┼───────@───X────────X───M───
                │       │
q_4: ───────────X───────X───M────────────────
                                    └──┘
```

```
[ 0.  0.  0.  0. 10.  0.  0.  0.]
```

Figure 3.15: Example: how to use the QAL with Cirq

T-ket

Introduction T-ket is a software platform developed by Cambridge Quantum Computing (CQC). As the other languages, it basically allows to describe quantum circuit and to execute them on simulator or on real quantum computers. In the following the basic commands will be shown.
The principle of operation of this language is very similar to that of the other two mentioned above. In particular, it is possible to define a quantum circuit as follows:

qc = Circuit(qubits, bits)

Once a quantum circuit has been instantiated, it is possible to hang the various gates to it like this:

qc.gate_name(qubit)

To use OpenQASM description, the following instructions can be used:

#Create an OpenQASM description of the desired component
$my\_arithmetic = arithmeticFunction(...)$
#Create a custom circuit from OpenQASM description
$arithmetic\_gate = circuit\_from\_qasm\_str(my\_arithmetic)$

It is possible to append the generated circuit in this way:

qc.add_circuit(arithmetic_gate, qubits)

Finally, the measurement operators can be inserted as follows:

qc.Measure(qubit, bit)

Again, for the purposes of this very brief introduction, it is not particularly interesting to pay attention on the syntax used to run the circuit on a simulator. See [14] for more details.

Example

```
from pytket import Circuit
from pytket.qasm import circuit_from_qasm_str
from pytket.backends.ibm import AerBackend
import numpy
from matplotlib import pyplot
from QAL.quantumArithmetic import *

#Generating an OpenQASM description of the 3-qubits
↪  cuccaro2004add
cuccaro_qasm =
↪  cuccaro2004add(None,1,['q',[[0,1,2],[3,4,5],[6,7]]])
qc_add = circuit_from_qasm_str(cuccaro_qasm)

#Now it is possible to use the 3-qubits adder in our Cirq
↪  circuit
qc = Circuit(8,4)

#Goal: Compute 110 + 010 --> Expected result: 1000
#Operand A (110)
qc.X(1)
qc.X(2)
#Operand B (010)
qc.X(4)

qc.add_circuit(qc_add, [0,1,2,3,4,5,6,7])

qc.Measure(3,0)
qc.Measure(4,1)
qc.Measure(5,2)
qc.Measure(7,3)
```

```
# Execute the circuit on the simulator and store the result
b = AerBackend()
b.compile_circuit(qc)
handle = b.process_circuit(qc, 100)
counts = b.get_counts(handle)

#Plot the result
res = {}
for key in counts.keys():
        new_key = ''.join(map(str,key[::-1]))
        res[new_key] = counts[key]

pyplot.title("Operation: 6 + 2 = 8")
_ = pyplot.bar(res.keys(),res.values(), width = .5, color = 'b')
```

Output:

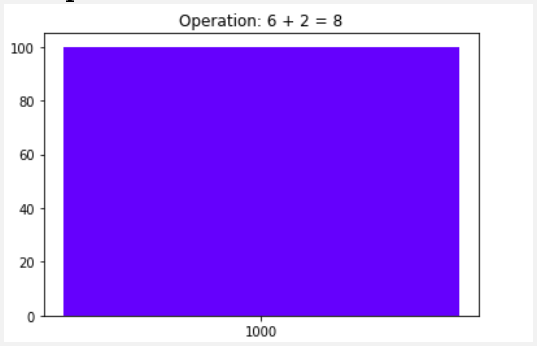


Figure 3.16: Example: how to use the QAL with T-Ket

3.2.5 Flexibility in Connecting Qubits

The purpose of this section is to highlight how it is possible to insert the arithmetic circuits generated within larger circuits and how the function interface allows to connect the various qubits with extreme flexibility. To do this, an example where the aim is to perform the following calculation is proposed:

$$OUT = (A + B) \cdot (C - D)$$

This example is only intended to highlight the potential of the library.

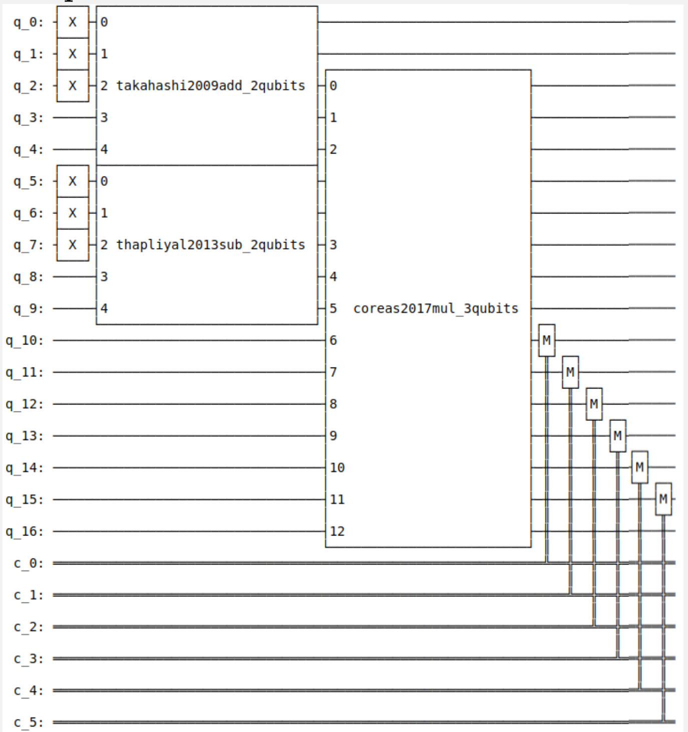
Example

```
from qiskit import (
  QuantumCircuit,
  execute,
  Aer)
from qiskit.visualization import plot_histogram
from QAL.quantumArithmetic import *

#Generating an OpenQASM description of the 2-qubits
↪  takahashi2009adder
takahashi_qasm =
↪  takahashi2009add(None,1,['q',[[0,1],[2,3],[4]]])
qc_add = QuantumCircuit.from_qasm_str(takahashi_qasm)
qc_add.name = "takahashi2009add_2qubits"
takahashi_adder = qc_add.to_instruction()


#Generating an OpenQASM description of the 2-qubits
↪  takahashi2013subtractor
thapliyal_qasm =
↪  thapliyal2013sub(None,1,['q',[[0,1],[2,3],[4]]])
qc_sub = QuantumCircuit.from_qasm_str(thapliyal_qasm)
qc_sub.name = "thapliyal2013sub_2qubits"
thapliyal_subtractor = qc_sub.to_instruction()
```

```
#Generating an OpenQASM description of the 3-qubits
↪  coreas2017multiplier
coreas_qasm = coreas2017mul(None,1,['q',[[0,1,2],[3,4,5],
↪  [6,7,8,9,10,11,12]]])
qc_mul = QuantumCircuit.from_qasm_str(coreas_qasm)
qc_mul.name = "coreas2017mul_3qubits"
coreas_multiplier = qc_mul.to_instruction()


#Now it is possible to use the 3-qubits adder in our Qiskit
↪  circuit
qc = QuantumCircuit(17,6)

#operand A=3
qc.x(0)
qc.x(1)
#operand B=1
qc.x(2)

#operand C=3
qc.x(5)
qc.x(6)
#operand D=1
qc.x(7)

#Perform A+B
qc.append(takahashi_adder,[0,1,2,3,4])#result stored in 3,4,5
#Perform C-D
qc.append(thapliyal_subtractor,[5,6,7,8,9])#result stored in
↪  7,8,9
#Perform (A+B)(C-D)
```

```
qc.append(coreas_multiplier,[2,3,4,7,8,9,10,11,12,13,14,15,16])

qc.measure(range(10,16),range(0,6))

# Use 'qasm_simulator'
simulator = Aer.get_backend('qasm_simulator')

# Execute the circuit on the qasm simulator and store the
↪  result
counts = execute(qc, simulator,
↪  shots=1000).result().get_counts(qc)

# Draw the circuit
print(qc)
# Plot the result
plot_histogram(counts, color='midnightblue', title="Operation:
↪  (3+1)x(3-1)=8")
```

Output:



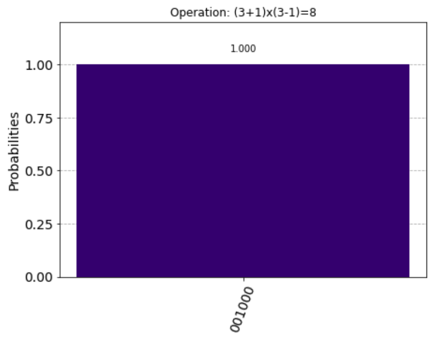


Figure 3.17: Example: how to use arithmetic blocks ad sub-blocks

Chapter 4

Quantum processors: benchmark using arithmetic circuits

"Glory is like a circle in the water,
Which never ceaseth to enlarge itself,
Till by broad spreading, it disperses to naught."
William Shakespeare

Although the world of quantum computers is making great strides every day, their current performances do not allow to execute too complex circuits. These devices, in fact, still have a limited parallelism and a too high noise sensitivity.

The aim of this chapter is to show the results obtained by running some of the circuits implemented in the QAL library on some free-accessible quantum computers. The first section aims to illustrate general information to the reader about how the various tests were carried out (which circuits were analysed, which target architectures, *etc.*). The second section, on the other hand, is intended to show the results obtained from the various tests. Within this section, in addition to showing why circuits implemented in this library cannot be reliably executed on current quantum computers, some "anomalous" behaviors related to the results obtained will be also commented.

4.1 Setup Definition

Once the library development was complete, some of the implemented circuits have been tested on free-accessible quantum computers in order to verify if the implemented circuits can be reliably executed on real hardware. The aim of this section is to explain the methodology used to carry out the various tests.

4.1.1 Analyzed Circuits

Remembering what is said in Section 1.7, today's quantum computers find a critical parameter in the depth of the circuit. Particularly, the closer the execution time of a circuit is to the qubits' decoherence times[1], the more the reliability of the final result will be negatively affected. For this reason, among the circuits implemented, it was decided to analyze those characterized by the shortest circuit depth: the two adders implemented by the functions takahashi2009add and thapliyal2013add[2]. It is noted that this choice was due to the fact that, at least theoretically, the execution of the other circuits would have led to results more influenced by the problems plaguing the current available quantum computers, due to their longer depth and a significantly higher number of involved qubits. It is also noted that - in order to obtain reliable results - each of the possible incoming combinations[3] has been tested ten times.

4.1.2 Quantum Computers

Since the total number of qubits required to simulate the two adders mentioned in the previous paragraph is five (see Section 2.3.4), it was decided to use four different free-accessible quantum computers offered by IBM: Ourense, Valencia, Vigo and Yorktown, all of which have five qubits. A characteristic problem of these devices is that qubits are not "all connectable to each other". In other words, it is not possible to arbitrarily choose the qubits on which to run a two-qubit gate. Each quantum computer has a *coupling map*, that is a graph representing on which pairs of qubits

[1]In most cases the decoherence time is more critical than the relaxation time.

[2]In the version without carry in.

[3]Since the parallelism of the two operands is equal to two, the total number of possible inbound combinations is equal to $2^2 \cdot 2^2 = 16$.

a two-qubit gate can be applied. The coupling maps of the quantum computers examined are shown in Figure 4.1.

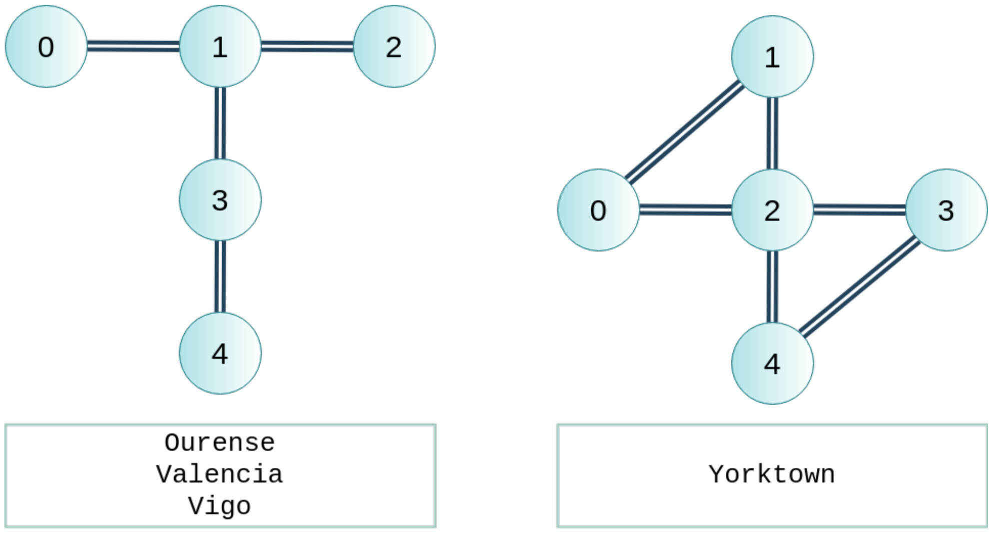


Figure 4.1: Yorktown, Ourense, Valencia, Vigo: coupling map

The reader may have already guessed that, often, there is the need to modify a circuit in order to make it compatible with the target architecture. In fact, during the "layout" operation, that is, during the mapping of "virtual"[4] in "real"[5] qubits, a series of SWAP gates are added to compensate for the limited connectivity between real qubits. It can be observed that, although these additional gates allow to execute circuits otherwise not executable, they increases the depth of the circuit, reducing performances. Further comments can be made about the devices in question. All the devices examined belong to the IBM quantum computer family called "Canary". In particular, Yorktown is part of the "Revision 2" category, while Ourense, Valencia and Vigo are part of the "Revision 3" category ([15]). A further observation is about the Quantum Volume, which is a figure of merit used by IBM to characterize the overall performance of its quantum computers. Since the performance of quantum computers depends not only on the number of qubits, but also on many other factors,

[4] "Virtual" qubits are the qubits defined in the circuit description. A virtual qubit, therefore, is not associated with any physical properties (relaxation, decoherence etc). It is only taken into account from a functional point of view.

[5] "Real" qubits are the qubits physically present in the quantum computer. Each real qubit, therefore, is characterized by its own physical properties (*e.g.* relaxation time, decoherence time etc.) that are experimentally measurable.

IBM has decided to use a single number to characterize its devices: the Quantum Volume. The bigger the Quantum Volume, the better the performances of the device. Among the quantum computers examined, there are two, Ourense and Yorktown, with a Quantum Volume of 8, while two others, Valencia and Vigo, have a Quantum Volume of 16.

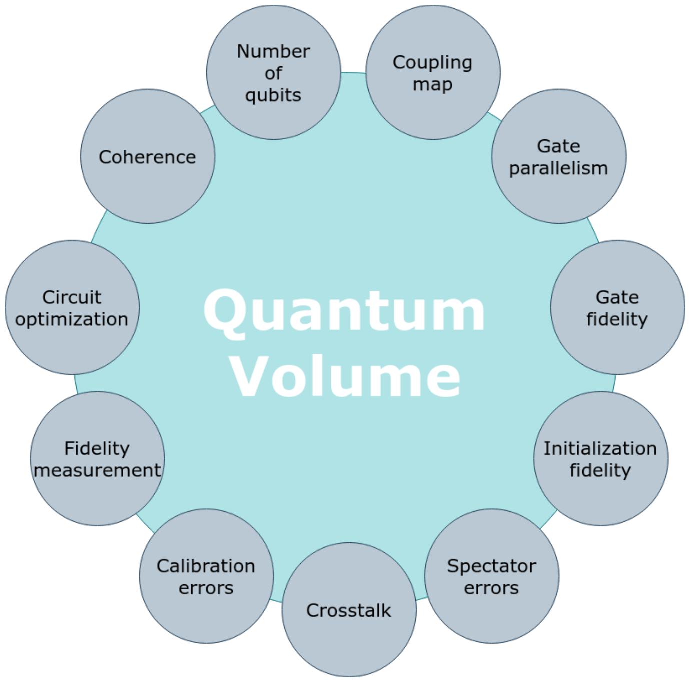


Figure 4.2: Quantum Volume

A final observation that can be made about the examined quantum computers concerns the relaxation and decoherence times of the various qubits and the error distributions of the various gates. Before proceeding showing the values of these parameters, it is good to make an important note: due to the high number of simulations, the two circuits examined were tested on different days, resulting in different parameters of the devices[6]. Therefore, if the reader should notice in the following - for the same device - different parameters (*e.g.* decoherence times, relaxation times etc.), know that this is the reason.

Relaxation and decoherence times are described in Tables 4.1 and 4.2.

[6]IBM updates quantum computers parameters daily.

Table 4.1: Relaxation and decoherence times: Takahashi adder

Qubit	Vigo		Valencia		Ourense		Yorktown	
	T1 [us]	**T2** [us]	**T1** [us]	**T2** [us]	**T1** [us]	**T2** [us]	**T1** [us]	**T2** [us]
Q0	121.27	18.27	84.28	60.25	86.74	34.26	70.86	18.7
Q1	96.57	105.87	120.8	69.37	106.68	31.36	46.74	21.79
Q2	119.96	130.05	75.71	44.22	82.92	84.67	41.68	41.27
Q3	120.89	101.25	101.65	44.23	124.41	103.06	68.22	39.86
Q4	83.22	51.05	88.75	141.26	121.32	27.15	63.01	45.8

Table 4.2: Relaxation and decoherence times: Thapliyal adder

Qubit	Vigo		Valencia		Ourense		Yorktown	
	T1 [us]	**T2** [us]	**T1** [us]	**T2** [us]	**T1** [us]	**T2** [us]	**T1** [us]	**T2** [us]
Q0	143.56	21.75	115.58	65.62	99.54	76.41	71.02	26.62
Q1	87.19	95.03	85.34	56.03	75.9	32.41	50.9	23.28
Q2	109.41	143.47	64.86	55.08	97.57	113.37	41.51	63.24
Q3	76.6	83.96	72.31	51.78	105.08	103.72	38.12	25.68
Q4	107.43	23.72	35.95	48.11	72.36	27.54	40.64	42.7

As for the distribution of errors related to the different gates, the typical error values for the various devices are shown in Figures 4.3 and 4.4[7]. Although only errors relating to the U2 and CNOT gates appear in the figures just mentioned, a null error can be assumed for U1 - as it is implemented via software - and twice the error of U2 can be supposed for U3, as its duration is twice that of U2.

[7]Although these values are also updated daily, for clarity it was preferred to show average values.

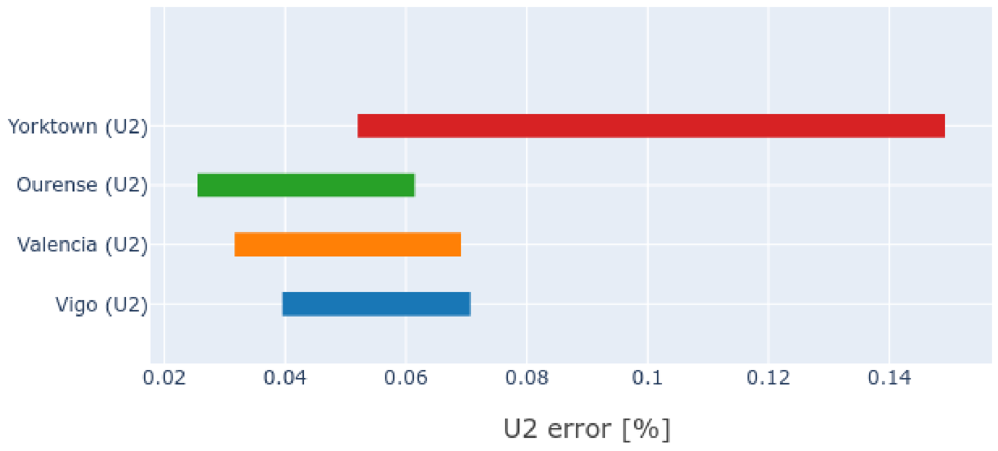


Figure 4.3: Gate U2: typical errors

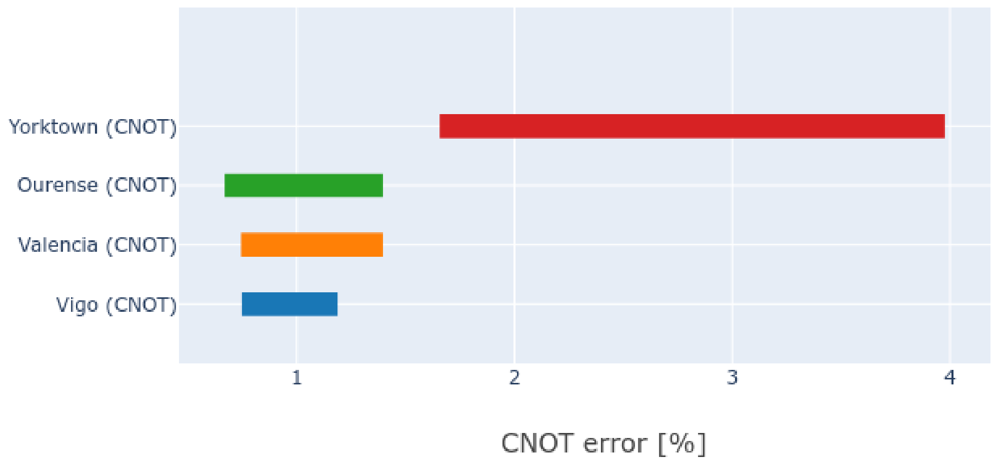


Figure 4.4: Gate CNOT: typical errors

4.2 Experimental Results

Correct Output Probability The first figure of merit that has been evaluated is the probability to measure a correct outcome. This probability - that is shown for each of the sixteen combinations in charts of Figures 4.5 and 4.6 - is always less than 0.45. This shows how today's technology does not allow to reliably perform the arithmetic circuits implemented in this library.

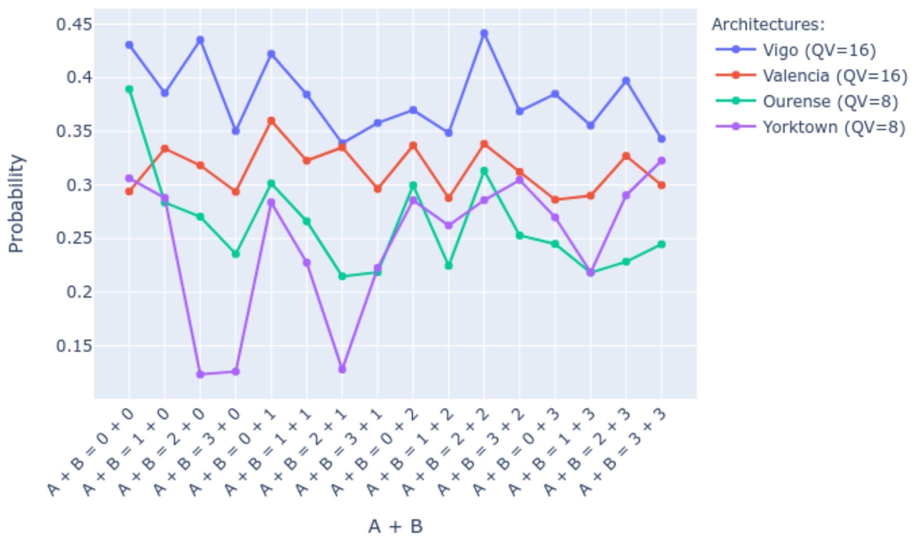


Figure 4.5: Correct outcome probabilities [Takahashi adder, 2-qubits]

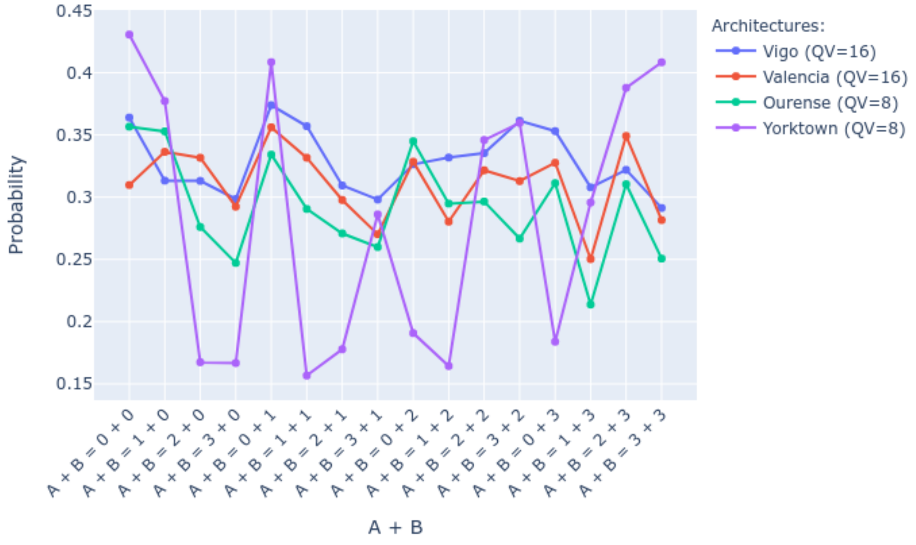


Figure 4.6: Correct outcome probabilities [Thapliyal adder, 2-qubits]

The first aspect that catches attention when looking at charts in Figures 4.5 and 4.6 is the “anomalous” behavior of the quantum computer Yorktown. In fact, even though it is characterized by higher connectivity than the other devices - thus involving fewer gates[8] (see Figures 4.7 and 4.8) - it shows in many cases a very low probability of measuring the correct result.

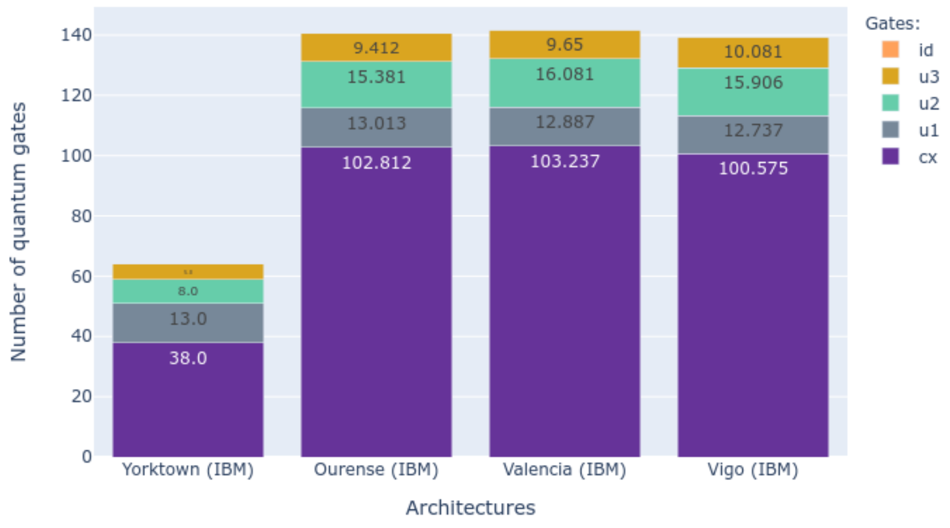


Figure 4.7: No. quantum gates: Takahashi adder

[8]The more qubits of the device are connected to each other, the smaller the number of gates that must be added in the layout phase to make the circuit compatible with the target architecture.

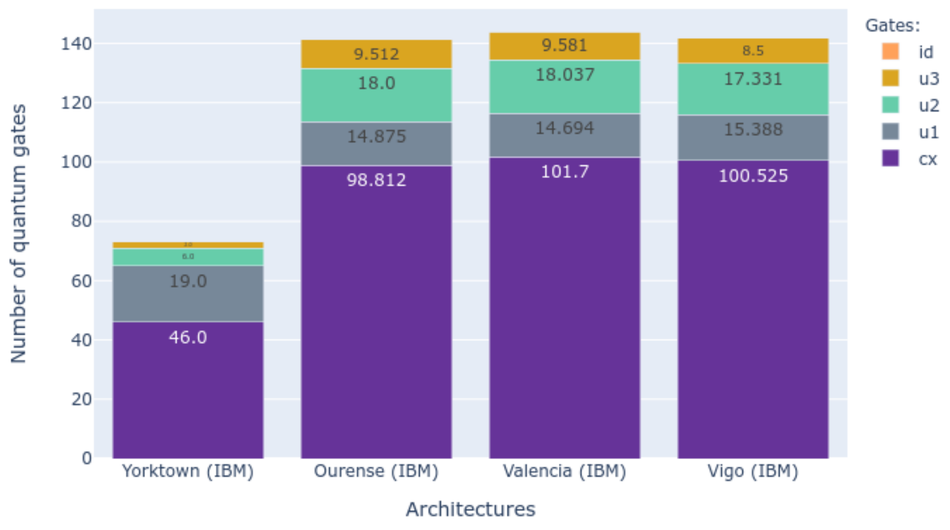


Figure 4.8: No. quantum gates: Thapliyal adder

A possible answer could lie in three facts:

1. Although the number of gates used is about half that of other quantum computers, Yorktown's gates have - on average - significantly higher error rates (see Figures 4.3 and 4.4);

2. Decoherence and relaxation times (see Figures 4.9 and 4.10) are - on average - worse than those of other devices;

3. Remembering how the various superconducting qubits are coupled with each other (see Section 1.7), it is possible that - due to the high connectivity that characterizes the Yorktown device - the crosstalk phenomenon is much more important than in other devices, introducing further errors.

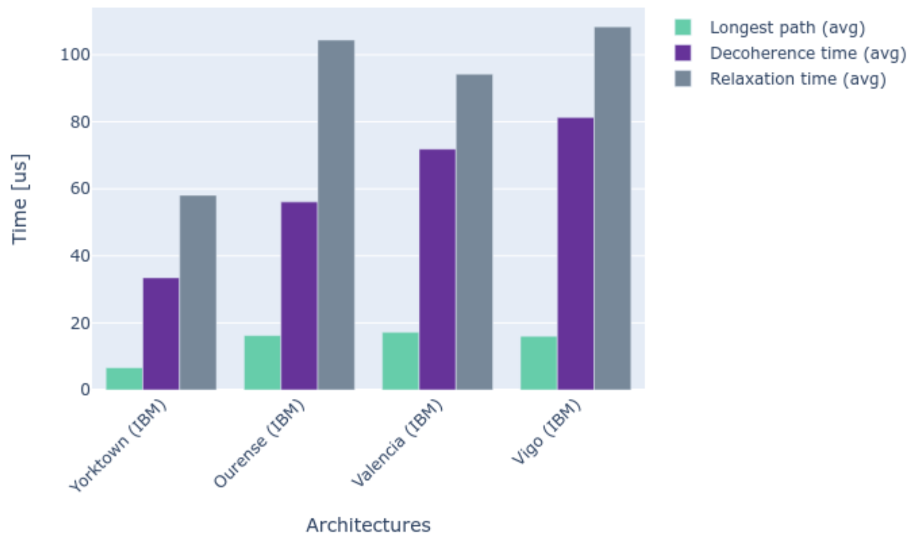


Figure 4.9: Longest path, decoherence time and relaxation time: Takahashi adder

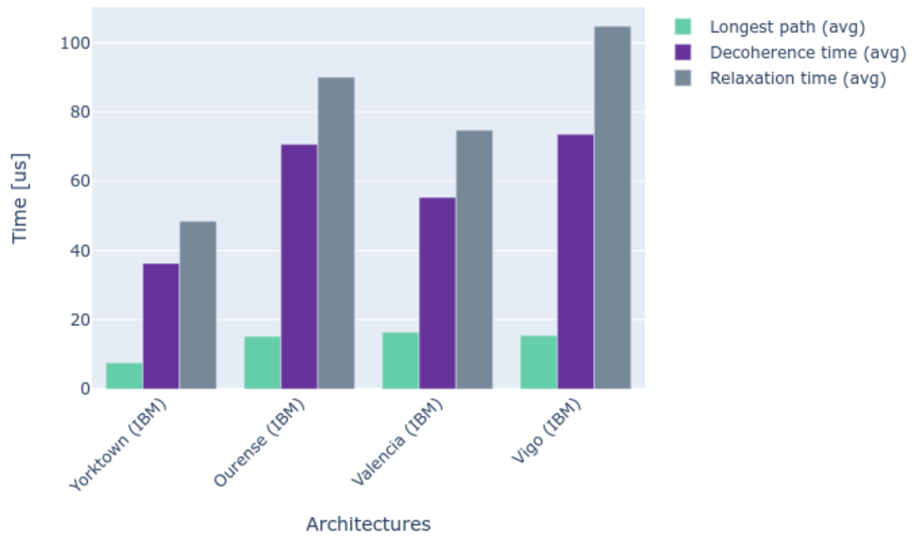


Figure 4.10: Longest path, decoherence time and relaxation time: Thapliyal adder

Another observation that can be made about charts in Figures 4.5 and 4.6 is that although quantum computers with a higher Quantum Volume do not always have the greatest chance of getting a correct result, their behavior is more stable and the average probability of getting a correct result is greater, as shown in Figure 4.11.

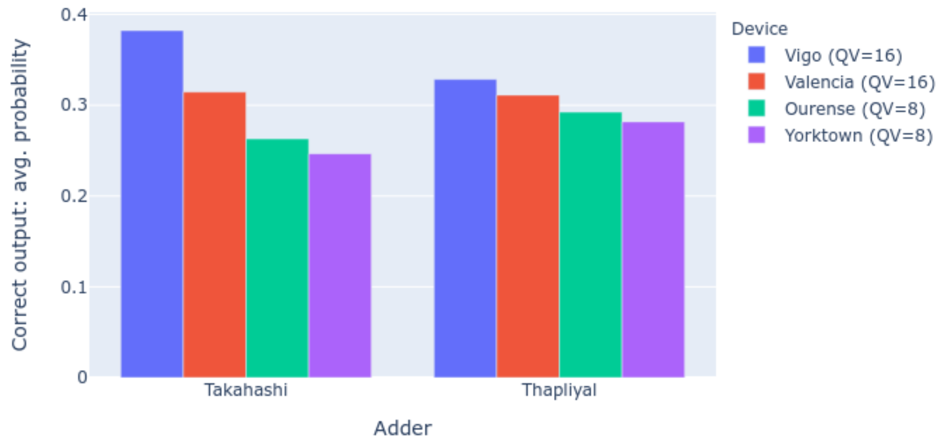


Figure 4.11: Correct outcomes average probabilities

At the beginning of this chapter it was noted that performances are heavily influenced by the qubits' decoherence times. In particular, it was observed that the greater the critical path delay with respect to the decoherence time, the worse the performances. This statement would seem to be at odds with charts in Figures 4.9 and 4.10 and the chances of getting a correct output shown in Figures 4.5 and 4.6. In fact, it is possible to see from these charts that - although the decoherence time is significantly larger than the average calculated delay for the critical path - the chances of getting a correct output never exceed 0.45. In order to justify this, two observations can be made:

- First, the decoherence time shown in Figures 4.9 and 4.10 is obtained by averaging between the decoherence times of all qubits. By analyzing the specifications of the various target devices, it can be seen that, in reality, qubits belonging to the same device have significantly different decoherence times (see

Tables 4.1 and 4.2 where the relaxation and decoherence times of the various qubits - rounded to two decimal places - are shown). This means that some qubits, those characterized by shorter decoherence times, can affect more the goodness of the final result[9]. Taking this into account would have required to check the length of the path associated with each qubit. This approach, however, would have been too complicated since the mapping between virtual and real qubits can change from execution to execution.

- Second, quantum computers under consideration do not actually allow multiple gates to run in parallel. To explain this concept, refer to Figure 4.12.

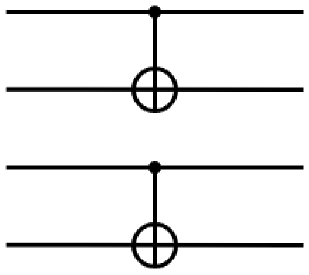


Figure 4.12: Two CNOT gates aligned

Although it would appear that the two CNOT gates can run in parallel, since they act on different qubits, it is possible that they cannot be run in parallel due to technological limitations (*e.g.* crosstalk phenomena could be very significant). This means that, in fact, the critical path calculated earlier is a very optimistic estimate of what happens on real hardware, where gates that theoretically could run in parallel, are actually run in series. Not knowing exactly how this fact is handled in the various devices, it was not possible to make a precise estimate of how it actually affects the performances.

4.2.1 Layout effort

One last parameter that is worth showing, although it has already been mentioned in the previous paragraphs, is the *layout effort*, which gives an idea of how many

[9]Clearly it all depends on how the mapping between virtual qubits and real qubits is made.

gates need to be added to a particular circuit to make it executable on a certain target architecture.

$$layout\_effort = \frac{No.gates\_after\_layout - No.gates\_before\_layout}{No.gates\_before\_layout} \cdot 100$$

Figures 4.13 and 4.14 show the number of gates before[10] and after the layout operation for each target architecture, and the corresponding layout effort.

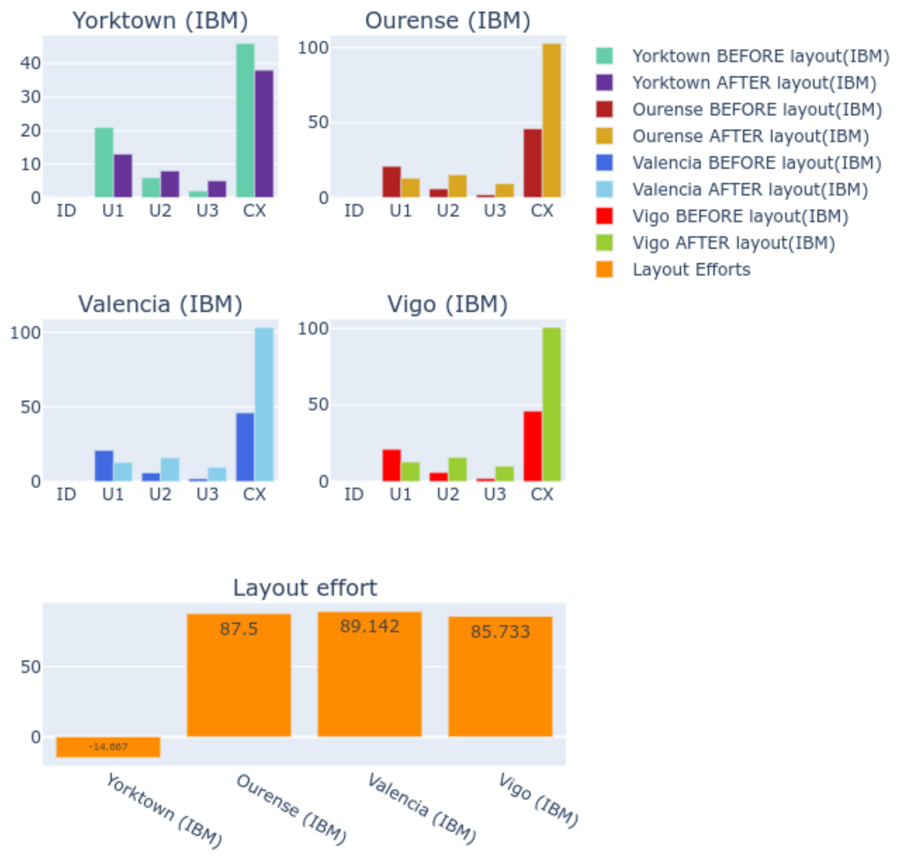


Figure 4.13: Additional gates due to layout operation: Takahashi adder

[10]It is noted that, before calculating the number of gates prior to the layout operation, a translation of the gates in the OpenQASM description of the circuit (X, CX, CCX, T, etc.) was made in the basis set of the targets (ID, U1, U2, U3 and CX); this translation has been carried out without introducing any optimization.

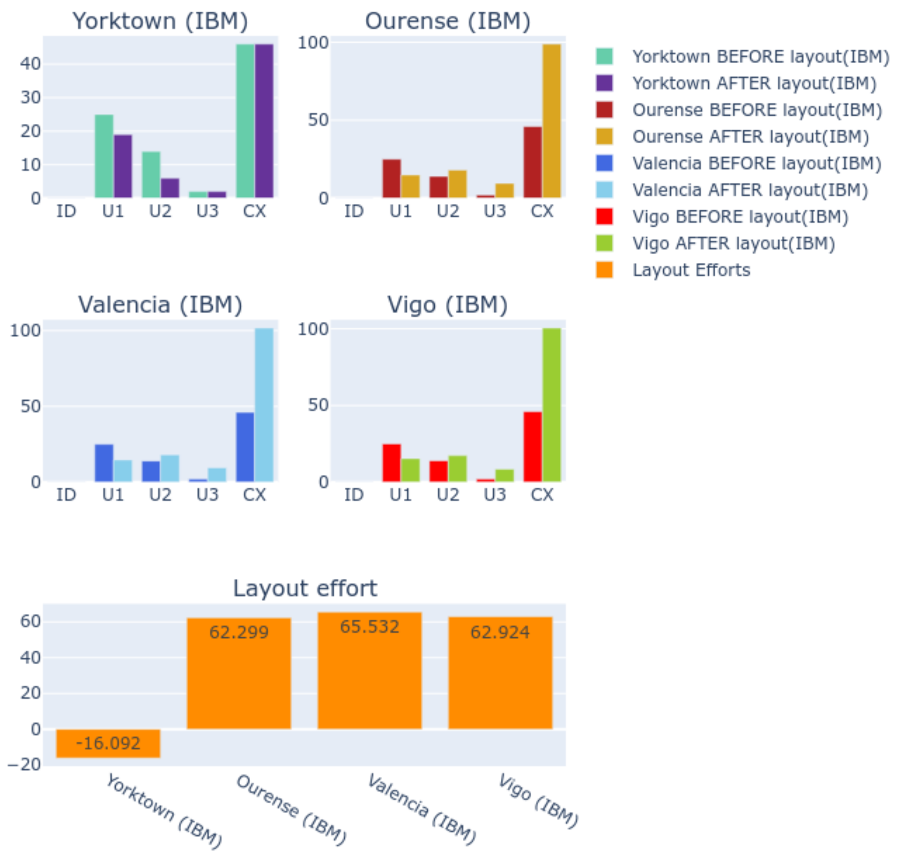


Figure 4.14: Additional gates due to layout operation: Thapliyal adder

From Figures 4.13 and 4.14 it is possible to see that the lower the connectivity of the target device, the greater the number of gates that must be added to the circuit in order to make it compatible with the target architecture. In particular, in the case of the Yorktown quantum computer, thanks to the high connectivity of the various qubits, it is possible to optimize the circuit in such a way that the number of gates after the layout is less than the number of gates before the layout.

Chapter 5

Conclusion

Automating the generation of arithmetic circuits for Quantum Computing is the idea behind this thesis. In this regard, after a careful bibliographic research about the state of the art in the field of arithmetic circuits for Quantum Computing, the Python programming language was used to create a library capable of generating circuit descriptions of arbitrary parallelism.

The library, developed according to the PEP8 standard and widely documented using the Sphinx tool, is able to generate OpenQASM descriptions of arithmetic circuits that, for instance, can be used as sub-blocks in the implementation of oracle-based algorithms, such as the Grover's algorithm. The developed library is also able to interface with other programming languages for Quantum Computing, such as Qiskit, Cirq and T-ket. It is noted that the library also offers numerous support functions that allow both to manipulate and analyze the generated circuit descriptions and to analyze the results of the executions performed on simulator or real hardware.

Once the development was finished, some of the implemented circuits have been tested on free-accessible quantum computers. The result of the analysis shows that the implemented circuits cannot be reliably executed on real hardware currently available. However, it should be noted that the results obtained show a significant improvement as the Quantum Volume increases. It is therefore possible to hypothesize that, by testing the library circuits on higher performing quantum computers - *e.g.* the Honeywell trapped-ion quantum computer (QV=64) - the performance of the circuits improves significantly.

Arithmetic circuits implemented in the library are all based on integer/fixed-point numbers. A possible extension of the library could therefore concern the implementation of circuits for floating point arithmetic, which could be used, for example, to accelerate simulations of quantum physical systems.

A further observation is inherent to the coding of the information. Since the developed library is able to generate circuits that encode information on binary strings, a possible extension, besides the introduction of new circuits based on the encoding of information on binary strings, could consist in inserting other types of quantum circuits able to encode the information in ways different from the binary one. In fact, in Quantum Computing there are other types of information encoding, such as amplitude encoding and phase encoding, that do not use binary strings to encode information.

It is interesting to bring to the reader's attention the fact that the methodology used to implement the various arithmetic circuits can be applied to any type of quantum circuit, not necessarily arithmetic. From this perspective, the proposed library can be seen as a first step towards a software library for Quantum Computing able to use the same interface to implement a wide range of functions.

“ Qui nasce, qui muore il Mio canto:
e parrà forse vano
accordo solitario;
ma tu che ascolti, recalo
al tuo bene e al tuo male;
e non ti sarà oscuro.”

Clemente Rebora

Appendices

Appendix A

Basic quantum gates translated in native IBM set

A.1 Single-Qubit Gates

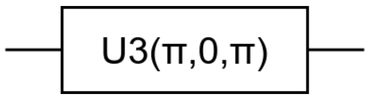


Figure A.1: Pauli X gate using IBM superconducting basis gates

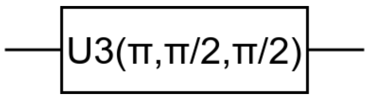


Figure A.2: Pauli Y gate using IBM superconducting basis gates

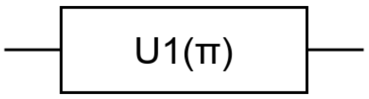


Figure A.3: Pauli Z gate using IBM superconducting basis gates

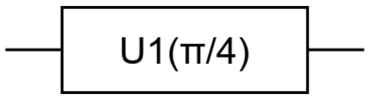


Figure A.4: T gate using IBM superconducting basis gates

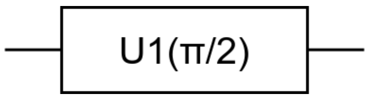


Figure A.5: S gate using IBM superconducting basis gates

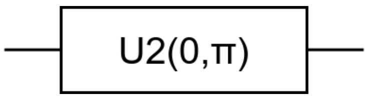


Figure A.6: H gate using IBM superconducting basis gates

A.2 Multiple-Qubits Gates

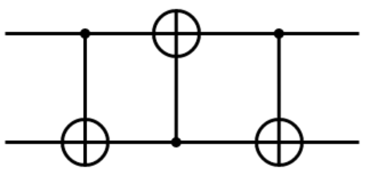


Figure A.7: SWAP gate using IBM superconducting basis gates

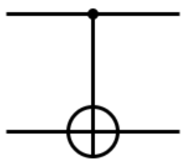


Figure A.8: CNOT gate using IBM superconducting basis gates

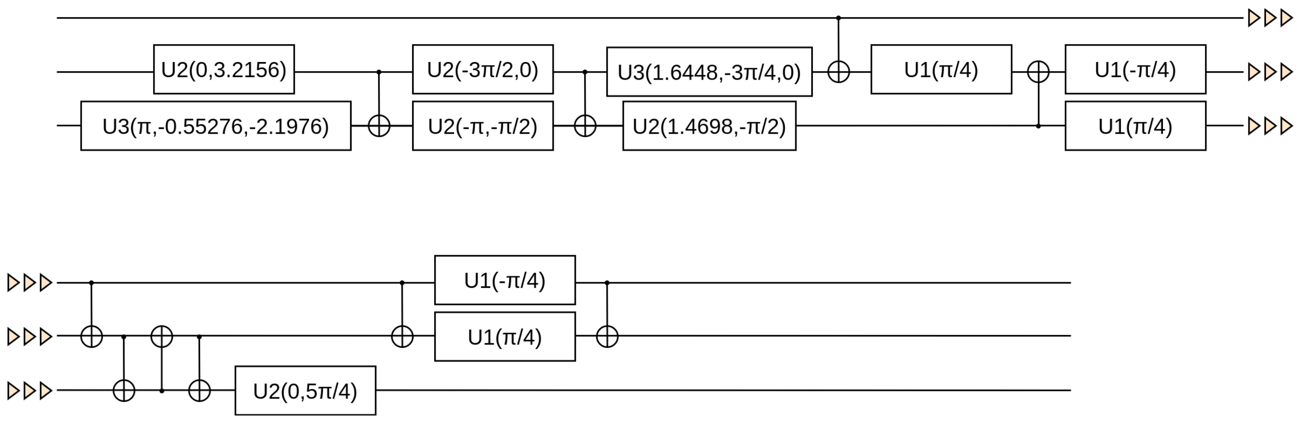


Figure A.9: Toffoli (CCNOT) gate using IBM superconducting basis gates

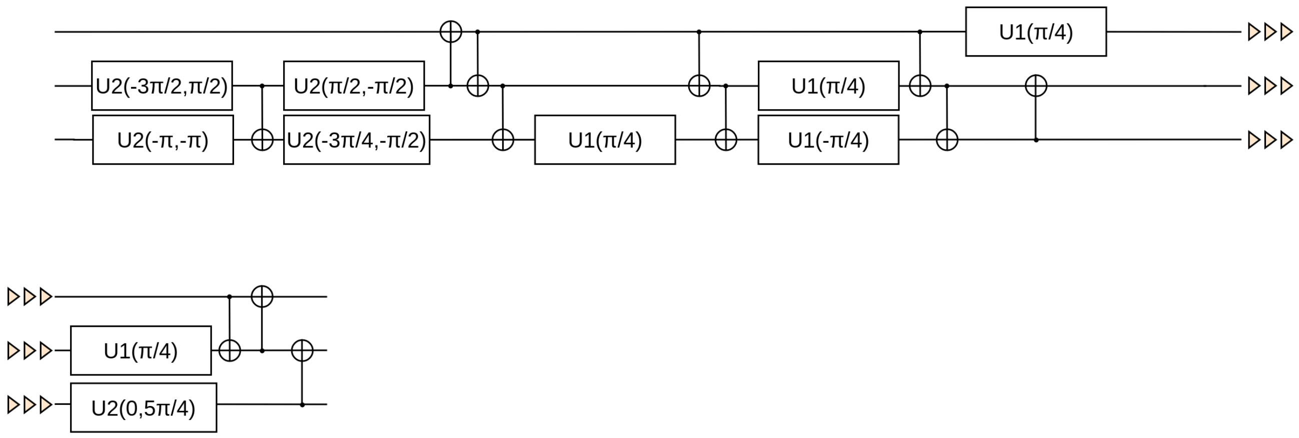
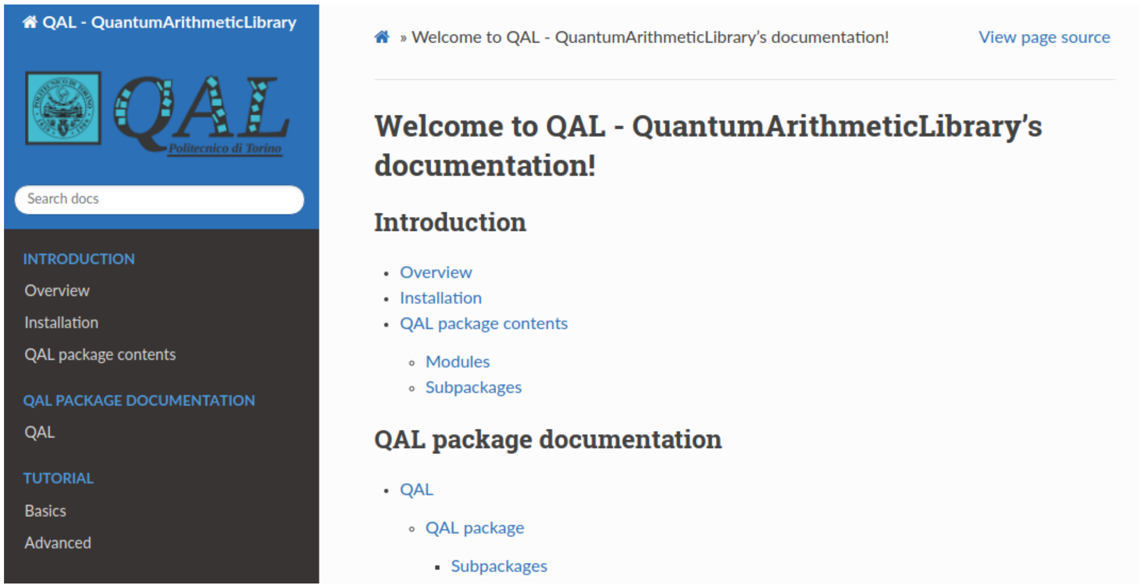


Figure A.10: Fredkin(CSWAP) gate using IBM superconducting basis gates

Appendix B

QAL library documentation: an overview

The library documentation is structured in three sections: an introductory first part, where general information about the library is given; a second part containing the documentation of the code, where all the various functions are described in detail; and a final part, which shows how to use the library.
Three snapshots of the documentation are proposed below just to give the reader an idea of how the interface looks like.



Welcome to QAL - QuantumArithmeticLibrary's documentation!

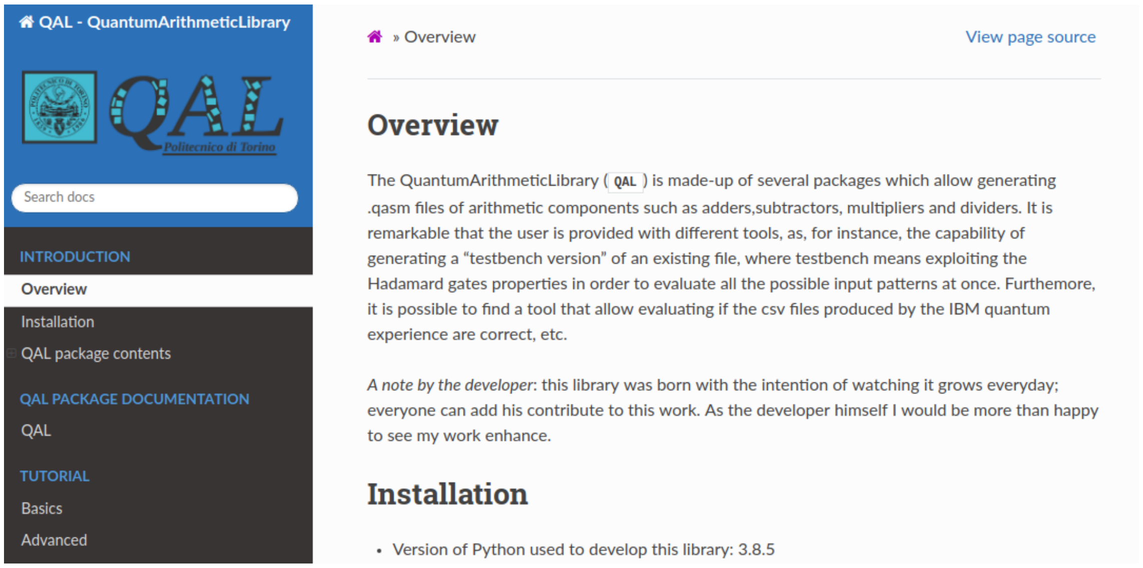
Introduction

- Overview
- Installation
- QAL package contents
  - Modules
  - Subpackages

QAL package documentation

- QAL
  - QAL package
    - Subpackages

Figure B.1: QAL documentation: index (snapshot)



Overview

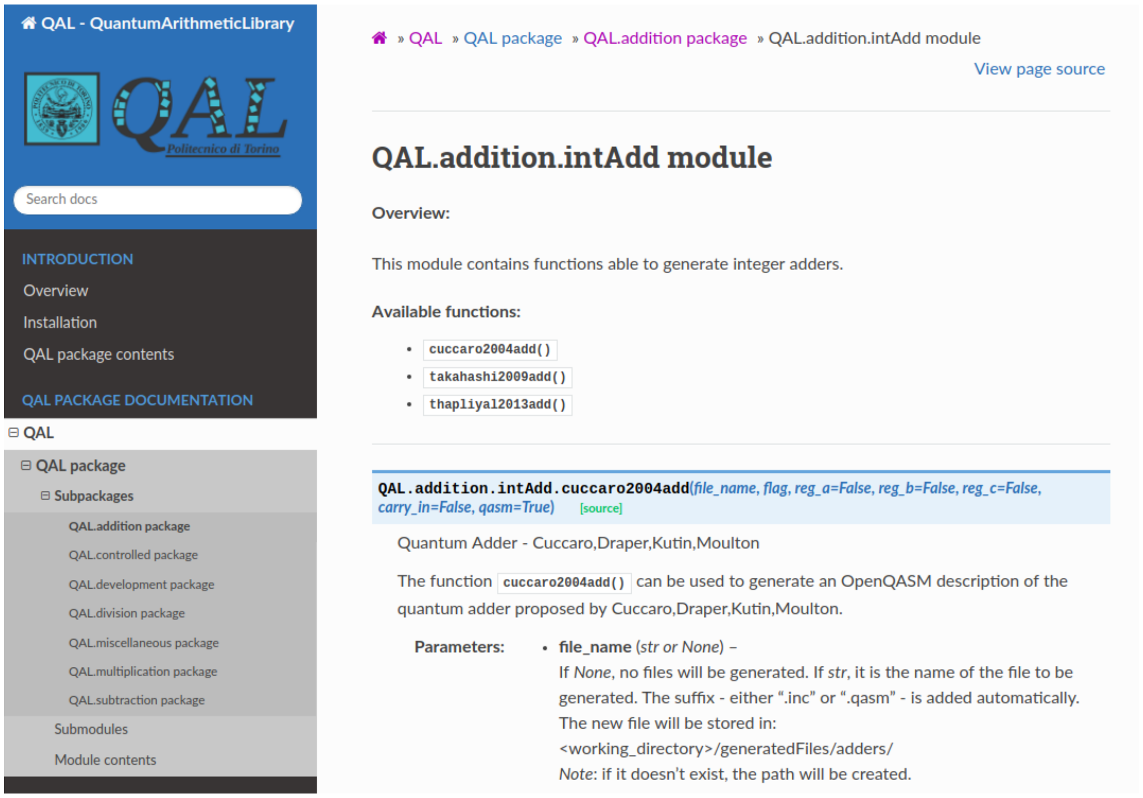
The QuantumArithmeticLibrary (`QAL`) is made-up of several packages which allow generating .qasm files of arithmetic components such as adders,subtractors, multipliers and dividers. It is remarkable that the user is provided with different tools, as, for instance, the capability of generating a "testbench version" of an existing file, where testbench means exploiting the Hadamard gates properties in order to evaluate all the possible input patterns at once. Furthemore, it is possible to find a tool that allow evaluating if the csv files produced by the IBM quantum experience are correct, etc.

A note by the developer: this library was born with the intention of watching it grows everyday; everyone can add his contribute to this work. As the developer himself I would be more than happy to see my work enhance.

Installation

- Version of Python used to develop this library: 3.8.5

Figure B.2: QAL documentation: introduction (snapshot)



QAL.addition.intAdd module

Overview:

This module contains functions able to generate integer adders.

Available functions:

- `cuccaro2004add()`
- `takahashi2009add()`
- `thapliyal2013add()`

`QAL.addition.intAdd.cuccaro2004add`(*file_name, flag, reg_a=False, reg_b=False, reg_c=False, carry_in=False, qasm=True*) [source]

Quantum Adder - Cuccaro,Draper,Kutin,Moulton

The function `cuccaro2004add()` can be used to generate an OpenQASM description of the quantum adder proposed by Cuccaro,Draper,Kutin,Moulton.

Parameters:
- **file_name** (*str or None*) –
  If *None*, no files will be generated. If *str*, it is the name of the file to be generated. The suffix - either ".inc" or ".qasm" - is added automatically. The new file will be stored in: <working_directory>/generatedFiles/adders/
  Note: if it doesn't exist, the path will be created.

Figure B.3: QAL documentation: sample function (snapshot)

Bibliography

[1] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *Style Guide for Python Code*. PEP 8. 2001. URL: https://www.python.org/dev/peps/pep-0008/.

[2] Steven A. Cuccaro et al. *A new quantum ripple-carry addition circuit*. 2004. arXiv: quant-ph/0410184 [quant-ph].

[3] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511813887.

[4] Yasuhiro Takahashi, Seiichiro Tani, and Noboru Kunihiro. *Quantum Addition Circuits and Unbounded Fan-Out*. 2009. arXiv: 0910.2530 [quant-ph].

[5] H. Thapliyal and N. Ranganathan. "Design of Efficient Reversible Binary Subtractors Based on a New Reversible Gate". In: *2009 IEEE Computer Society Annual Symposium on VLSI*. 2009, pp. 229–234.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511976667.

[7] Himanshu Thapliyal and Nagarajan Ranganathan. "Design of efficient reversible logic-based binary and BCD adder circuits". In: *ACM Journal on Emerging Technologies in Computing Systems* 9.3 (Sept. 2013), 1â31. ISSN: 1550-4840. DOI: 10.1145/2491682. URL: http://dx.doi.org/10.1145/2491682.

[8] Andrew W. Cross et al. *Open Quantum Assembly Language*. 2017. arXiv: 1707.03429 [quant-ph].

[9] Edgard Muñoz-Coreas and Himanshu Thapliyal. *T-count Optimized Design of Quantum Integer Multiplication*. 2017. arXiv: 1706.05113 [quant-ph].

[10] Himanshu Thapliyal et al. *Quantum Circuit Designs of Integer Division Optimizing T-count and T-depth*. 2018. arXiv: 1809.09732 [quant-ph].

[11] Héctor Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. DOI: 10.5281/zenodo.2562110.

[12] Abraham Asfaw et al. *Learn Quantum Computation Using Qiskit.* 2019. URL: http://community.qiskit.org/textbook.

[13] P. Krantz et al. “A quantum engineer’s guide to superconducting qubits”. In: *Applied Physics Reviews* 6.2 (June 2019), p. 021318. ISSN: 1931-9401. DOI: 10.1063/1.5089550. URL: http://dx.doi.org/10.1063/1.5089550.

[14] Seyon Sivarajah et al. “t$|$ket$\rangle$: A retargetable compiler for NISQ devices”. In: *Quantum Science and Technology* (2020). URL: https://iopscience.iop.org/article/10.1088/2058-9565/ab8e92.

[15] Jerry Chow and Jay Gambetta. *Quantum Takes Flight: Moving from Laboratory Demonstrations to Building Systems.* URL: https://www.ibm.com/blogs/research/2020/01/quantum-volume-32/.

[16] The Cirq Contributors. *Cirq, a python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits.* URL: https://github.com/quantumlib/Cirq.