INTERNATIONAL HELLENIC UNIVERSITY
DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING

THESIS

Implementing a Quantum Arithmetic Logic Unit using the Qiskit SDK

Student:
Athanasios Tzimas
Student ID: 185287

Supervisor:
Ioannis K. Marmorkos

25-05-2024

Title of Dissertation Implementing a Quantum Arithmetic Logic Unit using the Qiskit SDK
Code of Dissertation 23167
Student's full name Athanasios Tzimas
Supervisor's full name Ioannis K. Marmorkos
Date of undertaking 23-04-2023
Date of completion 25-05-2024

We hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance We received in its composition. We have, furthermore, noted the various sources from which We extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, we affirm the exclusive composition of this paper by myself only, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University, does not necessarily entail the adoption of the author's views, on behalf of the Department.

Dedication

To my family and friends, who supported and listened me talk about stuff they did know or care.

I would also like to thank my supervisor Dr. Ioannis K. Marmorkos who with a lot of zeal agreed to supervise and guide me through this thesis and helped and encouraged me to learn Mathematics and Physics. Lastly, I would like to thank my mentor Dr. Konstantinos Goulianas who helped me discover teaching and helping others.

From the bottom of my heart: Thank you all.

Prolog

At the end of the 20th century as computers were evolving into bigger, faster (and more exponentially power-hungry) electrical machines, physists wondered about the physical limitations of such a systems. On his 1982 talk in MIT [1], he argued about the use of a computing system that simulated quantum physical system and he described a system abided by the laws of Quantum Mechanics that performed computations, a *Quantum Mechanical Computer*. Later, in 1984, he published a more specific article [2] describing a more complete computing system with its fundamental components. Unfortunately, quantum computers did not produce any fruitful and interesting results for many years. This status was quickly changed by Shor's work [3] on algorithms for prime factorization. This ground-breaking work was so explosive because of its ability to factorize large prime numbers in polynomial time. This meant that many cryptographic algorithms that generated keys by prime generation could be broken substantially faster with a quantum computer rather than a classical computer. In much modern times, with the work of superconducting qubits using Josephson's junctions [4] and Quantum Computers like IBM's System One [5] and Google's Quantum processors [6] quantum computers seem to be closer to the consumers than ever. In this thesis, we will implement a Quantum analogue to the classical digital circuit of the Arithmetic Logic Unit - a digital circuit that carries some of the electronic computer's arithmetic and logical operations. At the conclusion of the thesis, we will present the results of our basic Quantum Arithmetic Logic Unit while it operates four basic operations: addition, subtraction, multiplication and the two logical operations of equality and magnitude comparison.

Abstract

EN

In the last century, the theory of Quantum Mechanics helped pave the way for the creation of the transistor; a technological advancement that helped to create the electronic computer. As the computer was used more and more for different applications, physicists pondered the limits of the miniaturization of the computer. Feynman set the foundational ideas for a new kind of a computer; a Quantum computer, a computer that entales the Quantum Mechanical laws. Although the foundational work has existed for over half a century, Quantum computers are still not a commodity due to various difficulties on making error resistant qubits thus making chip manufacturing a very difficult hurdle to overcome and algorithms hard to implement and produce reliable data. In this work we are going to explore, analyze and code Quantum algorithms using the Qiskit SDK, a software development kit from IBM, to create Quantum algorithms as Quantum circuits that implement arithmetic and logic operations of a classical Arithmetic and Logic unit on a classical computer.

EL

Τον περασμένο αιώνα, η θεωρία της Κβαντομηχανικής βοήθησε να ανοίξει ο δρόμος για τη δημιουργία του τρανζίστορ- ένα τεχνολογικό απόκτημα που βοήθησε στη δημιουργία του ηλεκτρονικού υπολογιστή. Καθώς οι υπολογιστές χρησιμοποιούνταν όλο και περισσότερο για διάφορες εφαρμογές, οι φυσικοί προβληματίστηκαν για τα όρια της σμίκρυνσης των υπολογιστών. Ο Feynman έθεσε τις θεμελιώδεις ιδέες για ένα νέο είδος υπολογιστή - έναν κβαντικομηχανικό υπολογιστή, δηλαδή έναν υπολογιστή που αξιοποιεί την κβαντομηχανικούς νόμους για την λειτουργία του. Παρόλο που τα θεμέλια προυπάρχουν για πάνω από μισό αιώνα, οι κβαντικοί υπολογιστές δεν έχουν ακόμη αναπτυχθεί λόγω διαφόρων δυσκολιών στην κατασκευή qubits - κβαντικά bits - ανθεκτικά στον θόρυβο από το περιβάλλον, καθιστώντας έτσι την κατασκευή ολοκληρωμένων τσιπ και την σχεδίαση χρήσιμων αλγορίθμων που να παράγουν αξιόπιστα δεδομένα μια πολύ δύσκολη υπόθεση. Σε αυτή την εργασία θα να εξερευνήσουμε, να αναλύσουμε και θα συγγράψουμε κβαντικούς αλγορίθμους χρησιμοποιώντας το Qiskit SDK, ένα κιτ ανάπτυξης λογισμικού από την IBM, για να να δημιουργήσουμε κβαντικούς αλγορίθμους ως κβαντικά κυκλώματα που υλοποιούν αριθμητικές και λογικές πράξεις μιας Αριθμητικής και Λογικής Μονάδας όπως αυτές που υπάρχουν στους κλασικούς υπολογιστές.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In chapters 2 and 3 we are going to introduce the basis to understand some of the concepts of quantum circuits, like superposition and quantum parallelism. In chapter 4 we make a short introduction on the technologies we used to write and maintain the software used to implement the quantum circuits. In chapters 5 and 6, the main body of the thesis, we construct the basic pieces of the thesis, the Quantum Arithmetic and Logic Unit (QALU). In more detail, chapter 5, implements the basic sub-units of the QALU. In the later chapter, we piece together the QALU and test it using some of Qiskit API to run it in a real quantum computer. Lastly, in chapter 7 we draw the conclusions and make some remarks on future improvements of the design and architecture of the QALU.

Chapter 2

Elements of Classical Computing

2.1 The Binary Numeral System

The *binary numeral system* is a positional numeral system that uses two values - zero (0) and one (1) - to notate numbers. A number notated using two values is called a *binary number*.
Binary numbers are expressed as a sequence of digits $d_i d_{i-1} \dots d_1 d_0$ where i is the position of the digit d. A more concise expression of a binary number is:

$$A = \sum_{i=0}^{n} d_i \times 2^i \tag{2.1}$$

which expression is derived by the general definition

$$A = \sum_{i=0}^{n} d_i \times b^i \tag{2.2}$$

where b is the *base* or *radix* of the numeral system the number is expressed in and n is the total digits of the number. The digits of a binary number are notoriously called *binary digits* or *bits* for short.
We note that the left-most bit of a binary number is called the *most-significant bit* and the right-most *least-significant bit*, *MSB* and *LSB* resprectively.
To differentiate *decimal numbers* from binary numbers we notate that number with its base in subscript. So, the decimal number 110 will be notated as 110_{10} and the binary number 1010 as 1010_2.
We can covnert any binary number to its decimal numeral notation by following the expression 2.2. So that 1010_2 can be expressed in decimal as:

$$1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8 + 0 + 2 + 0 = 10_{10} \tag{2.3}$$

We can also do the reverse, convert a decimal number into its binary notation, by recursively dividing by the binary base and noting down the remainders until the operations gives a zero. For example, 13_{10} is converted in binary notation as follows:

1. $13 \div 2 = 6$ (remainder 1)
2. $6 \div 2 = 3$ (remainder 0)
3. $3 \div 2 = 1$ (remainder 1)
4. $1 \div 2 = 0$ (remainder 1)

If we read the remainders in reverse order we acquire the binary respresentation of the decimal number 13_{10} which is 1101_2.
The binary system is useful to all modern computers as they operate by using transistors as saturated switches. This was first realised by Shannon [7] who noted that any switching circuit can obey the rules of *binary logic*. This kind of logic was developed by Boole [8] and later concretely founded by Huntington [9]. This logic is now know as *Boolean algebra* and we will talk in more depth in section 2.4.

2.2 Binary Arithmetic

Just like in the decimal system, the binary system, defines operations that operate using binary numbers.
Addition in binary is as simple as addition in decimal. The symbol that carries the binary addition is "$+$" and the rules or *axioms* are as followed:

1. $0 + x = x$, where x is a bit. This rule defines zero as the identity element of addition, which means that any binary number that is added with zero will equal itself.
2. $1+1 = 0$, carrying 1. This is called carry and it is similar to when we add the decimal numbers together and they exceed the base (10). This produces a one that will be added to next sum of terms.

For example, we shall add $1101_2 + 1011_2$:

1	1	1	1	
	1	1	0	1
+	1	0	1	1
1	1	0	0	0

Table 2.1: An example of binary addition

Note the top row of digits, which are the carry bits from each column addition.
Binary Subtraction is also simple.

1. $0 - 0 = 0$
2. $0 - 1 = 1$, borrowing 1. This is similar to the carry except the next two digits that are going to be subtracted need to subtract the borrow too.
3. $1 - 0 = 1$
4. $1 - 1 = 0$

For example, we shall subtract $1101_2 - 1011_2$:

		1		
	1	1	0	1
−	1	0	1	1
	0	0	1	0

Table 2.2: An example of binary subtraction

Lastly, we will discuss the axioms of binary multiplication. The product of any two binary numbers can be produced by firstly calculating the *partial product* of these two numbers. The partial product is calculated by the following axioms:

1. $0 \times x = 0$
2. $1 \times x = x$. One (1) is the identity element of binary multiplication.

The product can then be calculated by summing all the partial products.
For example, we shall multiply $1101_2 \times 1011_2$:

					1	1	0	1
$\times$					1	0	1	1
					1	1	0	1
				1	1	0	1	
			0	0	0	0		
$+$		1	1	0	1			
	1	0	0	0	1	1	1	1

Table 2.3: An example of binary multiplication

2.3 Signed Binary Numbers

In general, negative numbers in decimal are notated by prefixing a minus symbol ($-$) before the number. As we previously mentioned, computers process information as sequences of bits. Thus negative numbers can not be processed "as-is" by computers, thus making the need for an encoding scheme.
One of the most simple encoding schemes to represent signed numbers is called the *signed-magnitude* representation. In this scheme, the MSB (*most-significant bit*) is considered to be the *sign*; if that bit is zero then the number is considered a positive integer otherwise it's a negative integer. Binary numbers that do not consider this scheme are called *unsigned binary numbers*.

Signed Decimal notation	Signed Binary notation
0	0000
1	0001
2	0010
$\vdots$	$\vdots$
6	0110
7	0111
-0	1000
-1	1001
$\vdots$	$\vdots$
-6	1110
-7	1111

Table 2.4: The table for 4-bit signed integers using the signed-magnitude representation

Note that in this scheme there is a negative and positive zero. This is not ideal or economical and thus, this representation scheme is not used when designing computer circuits.
By using the *signed-complement* representation scheme signed integers are represented as their complement. There are two kinds of signed-complement representations:

1. one's complement and

2. two's complement

The one's complement of a negative number is the bitwise inverse of a very bit of the corresponding positive counterpart of that number. For example, the one's complement of the signed decimal number -12_{10} is obtained by:

1. Take the unsigned bianry representation of the positive counterpart (12): 01100_2
2. Invert every bit (zero to one, one to zero): 10011_2

We should note that even with one's complement there are two representations of zero; just like the signed-magnitude scheme.
The two's complement is simply obtained by taking the one's complement of the negative number and adding one. For example, the two's complement of -12_{10} is obtained by:

1. Take the unsigned bianry representation of the positive counterpart (12): 01100_2
2. Invert every bit (zero to one, one to zero): 10011_2
3. Add one: $10011 + 1 = 10100$

We can see in the Table (2.5) the one's and two's complement of each integer decimal that can be represented using four bits of information.

Signed Decimal notation	One's complement	Two's complement
0	0000	0000
1	0001	0001
2	0010	0010
⋮	⋮	⋮
6	0110	0110
7	0111	0111
-8	-	1000
-7	1000	1001
⋮	⋮	⋮
-1	1110	1111
-0	1111	-

Table 2.5: The table for 4-bit signed integers using the signed-complement representation

Two's complement is used universaly by computers becuase the representation simplifies the circuitry that implements arithmetic operations. Subtraction, for example, of two binary numbers $A - B$ can be simplified as $A + B' + 1$ where B' is the one's complement of B.
For example, the difference of $1101_2 - 1011_2$ can be computed as follows:

1. Take the two's complement of the second term: $1011_2 \rightarrow 0100_2 + 1 = 0101_2$
2. Then compute $1101_2 + 0101_2 = 1\_0010_2$, excluding the overflow bit we get the indented difference

2.4 Boolean Algebra

Boolean algebra is a branch of algebra that operates with binary variables - variable that can be assigned either *true* or *false*. It defines three basic operations: negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$).

$$A \wedge B \tag{2.4}$$

$$A \vee B \tag{2.5}$$

$$\neg A \tag{2.6}$$

The rules of operations for each operator is described in the following *truth tables*. A truth table is a table that displays all the possible binary values of a *boolean variable* and all the possible outputs. These operators are most commonly mapped to logic functions: AND (conjunction), OR (disjunction) and NOT (negation), which can take either two input variables, for the first two, or one input variable, for the last one. These function names are also used to map the operation in the circuit model (see section 2.5).
The AND logic function states that the output will be *true* (or logical 1) if and only if both input variables are in the logical state 1.

a	b	$a \wedge b = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.6: The truth table for the conjunction operator

The OR logic function states that the output will be *true* if both or at the very least one of the input variables are in the logical state 1.

a	b	$a \vee b = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.7: The truth table for the disjunction operator

The NOT logic function inverts the state of the input variable - if the input is in the logical state 0 then it is inverted to the logical state 1 and vice versa.

a	$\neg a = a'$
0	1
1	0

Table 2.8: The truth table for the negation operator

These are also called *universal operators* because they can be used with each other to create other logic functions like: the NAND (Not-AND), NOR (Not-OR), XOR (Exclusive-OR), XNOR (Exclusive-Not-OR) are some of the most common functions other than the universal ones.

The NAND and NOR functions are self explanatory, they are just the negated AND and OR functions.

a	b	$\neg(a \wedge b) = (a \cdot b)'$
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.9: The truth table for the NAND function

a	b	$\neg(a \vee b) = (a + b)'$
0	0	1
0	1	0
1	0	0
1	1	0

Table 2.10: The truth table for the NOR function

The XOR function is the most elaborate logic function. The output is only true when only one of the inputs are true. This function is sometimes symbolised by the $\oplus$ operator.

a	b	$(\neg a \wedge b) \vee (a \wedge \neg b) = a' \cdot b + a \cdot b' = a \oplus b$
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.11: The truth table for the NAND function

2.5 The Digital Logic Gates

Computers are constructed using *digital components* - components that fundamentaly operate with binary inputs. These components manipulate voltages to produce a specific output - a logical low voltage is considered the zero or false state and a logical high voltage is considered the one or truth state. These components are called *logic gates* or *logic components*.
Just like Boolean Algebra, these components adhere to the same rules. The universal operations - the AND, OR and NOT functions - using the *circuit model* of computation are treated as circuits that accomplish the same operation.
The AND gate is a digital gate that takes two 1-bit inputs and outputs their logical conjunction (see Figure (2.1)). Thus the truth table and operator are analogous to its Boolean counterpart.

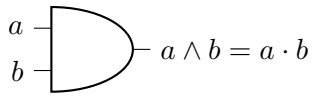


Figure 2.1: The circuit symbol for the AND function

The OR gate is a digital gate that takes two 1-bit inputs and outputs their logical disjunction (see Figure (2.2)). Again the truth table and operator are analogous to its Boolean counterpart.

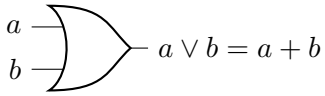


Figure 2.2: The circuit symbol for the OR function

Lastly, the NOT gate takes a 1-bit input and outputs its logical negation (see Figure (2.3)).

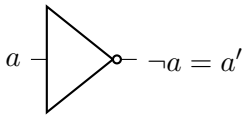


Figure 2.3: The circuit symbol for the NOT function

We would like to point out that there are also digital logic gates that symbolise the other compount logic functions - NAND, NOR, XOR and XNOR (see Figure (2.4)).

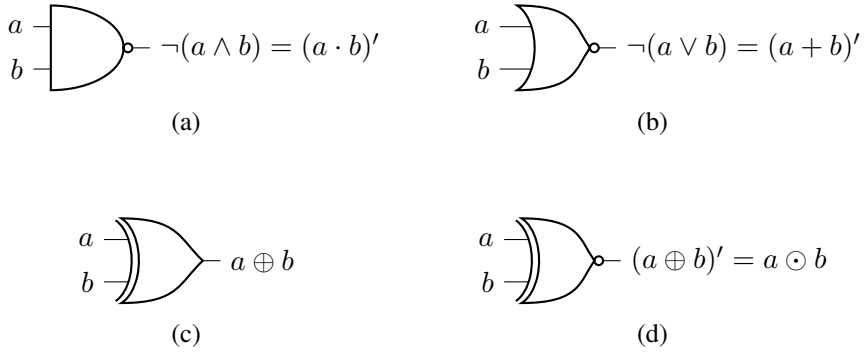


Figure 2.4: The digital logic gates for (a) NAND, (b) NOR, (c) XOR and (d) XNOR functions

2.6 Logic Circuits

We can use the logic gates introduced in section 2.5 to create *logic circuits* - circuits that compute some Boolean complex function. For example, we can produce a logic circuit that computes the sum of two bits.
As we have noted in section 2.2 the rules of addition are:

1. $0 + x = x$
2. $1 + 1 = 0$ carrying 1

Note that the sum S can be produced by the XOR gate and the carry C with the AND gate. Putting those two components together we get the following circuit:

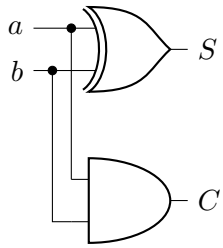


Figure 2.5: The logic circuit of the half adder

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 2.12: The truth table of the half adder

This digtal logic circuit is called *half adder*. Why is it called the *half* adder? This is because this circuit is not aware of any previous carry, thus it is not a *complete* circuit. As we have seen from section 2.2 carrying is very important to the calculation of the sum.
The circuit that does take into account for a previous carry bit is called the *full adder*. The previous carry is called the *carry-in* bit and it's the third input for this circuit.

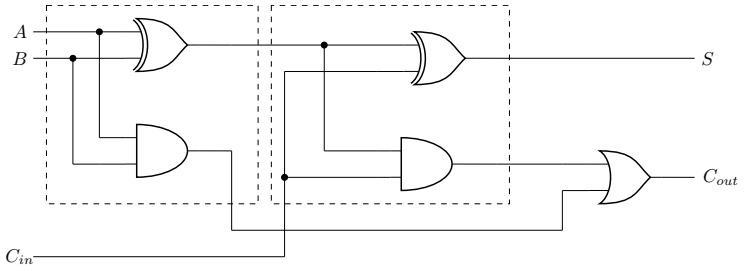


Figure 2.6: The logic circuit of the full adder using two half adders

a	b	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 2.13: The truth table of the full adder

We can see by the diagrams that the full adder circuit can be made out of two half adder circuits. These kind of circuits, that are made from other simple circuits, are called *combinational circuits*.
By placing full adders in sequence we can make another combinational circuit called the *ripple carry adder*. This circuit can compute the sum of n-bit inputs. It's called a *ripple* carry adder because in order to compute the n-th sum bit the circuit must first compute the $n-1$-th sum bit thus some latency is present on the circuit. We must note at this point that there are other better implementations of n-bit adders but it is out of the scope of this work. In figure (2.7) we can see an example of a 4-bit ripple carry adder which computes, for each input bit A_i, B_i and C_{in_i}, the sum bit S_i and the carry-out bit and passes it to the next full-adder block as the carry-in bit $C_{in_{i+1}}$. The logic continues until the circuit computes the sum bits for all the input bits.

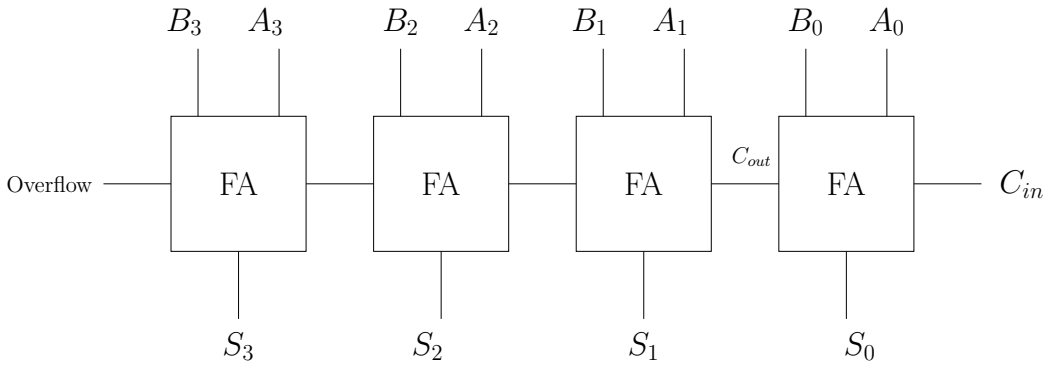


Figure 2.7: The logic circuit of a 4-bit ripple carry adder

2.7 A Basic Arithmetic Logic Unit

In Classical Computing and Digital Logic Design, the *Arithmetic Logic Unit* (ALU) is a fundamental component of the *Central Processing Unit* (CPU) of the computer that is responsible of carrying all of its the arithmetic and logical operations [10]. Depending on the design of the processor, an ALU can implement a number of arithmetic or logical operations. An example set of arithmetic operations of an ALU may be: addition, subtraction, multiplication and division. We must note that these are not the only ones. Many ALUs also implement boolean *bitwise* operations, operations done between bits, like: AND, OR, NOT, XOR, Shift left and right.
At the other hand, an example of logical operation set of an ALU may be: Testing Equality/Inequality and Magnitude Comparison (Greater or Less than).
An ALU generally has two operand inputs, inputs that the ALU will operate upon, a Function or Operation code, a binary sequence that instructs the ALU to do the correct operation with the given operand inputs and the Status input which is a bit sequence where each bit stands for a specific Status code or Status flag and its job is to note the status of the last operation that was executed by the ALU. The two only outputs of the ALU are the Status output, essentially this output will wrap-around and update the Status register, and the Ouput value where the product of the operation of the operands is stored. The diagram of this digital component is given in figure (2.8).

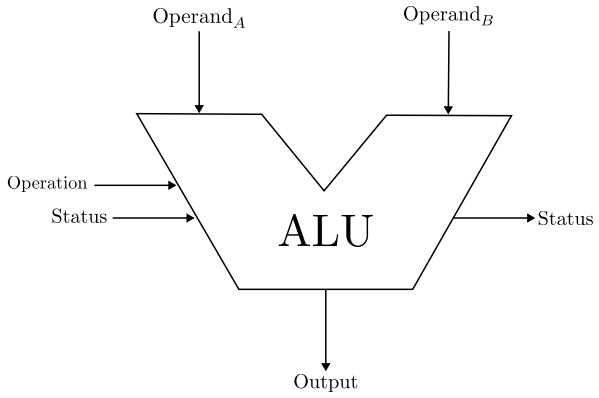


Figure 2.8: The Arithmetic Logic Unit diagram

We shall turn our attention now to the function/operation codes of the ALU. For a given number of implemented operations n, the ALU may need a bit-sequence of length:

$$N_n = \lceil \log_2(n) \rceil \tag{2.7}$$

So for $n = 8$, eight operations, the ALU needs $N_8 = \lceil \log_2(8) \rceil = 3$ or a bit sequence of three bits wide. So we assign each operation to a binary number and we acquire the following table:

Name	Function/Operation Code	Operation	Mnemonic Name
Addition	000	$A \leftarrow A + B$	`ADD`
Subtraction	001	$A \leftarrow A - B$	`SUB`
Multiplication	010	$A \leftarrow A \times B$	`MUL`
Division	011	$A \leftarrow A \div B$	`DIV`
Left Bit-Shift B-times	100	$A \leftarrow A << B$	`LBS`
Right Bit-Shift B-times	101	$A \leftarrow A >> B$	`RBS`
Test Equality	110	$A = B$?	`EQ`
Test Magnitude	111	$A \geq B$?	`GTE`

Table 2.14: An example function/operation code table for a basic Arithmetic Logic Unit

For this kind of operation set we can also set which operation will interact with the Status register and update its flags. For example, the most logical pick may be the `EQ` and `GTE` operations.
Following the above table we can schematically construct a very basic ALU:
The circuit in figure (2.9) computes all of the operations simultaneously but only outputs and/or updates the Status register when the appropriate Operationc code is set, which is controlled by the *multiplexer* circuit placed at the very end of the ALU's circuit.
We want to note that historically ALUs implement integer-based arithmetic operations. The unit that operates with real numbers encoded with a floating-point scheme, or *floating-point numbers*, is called the *Floating-Point Unit*. This kind of units are fundamentally different and we will not analyze them further, since they are out of the scope of this thesis.

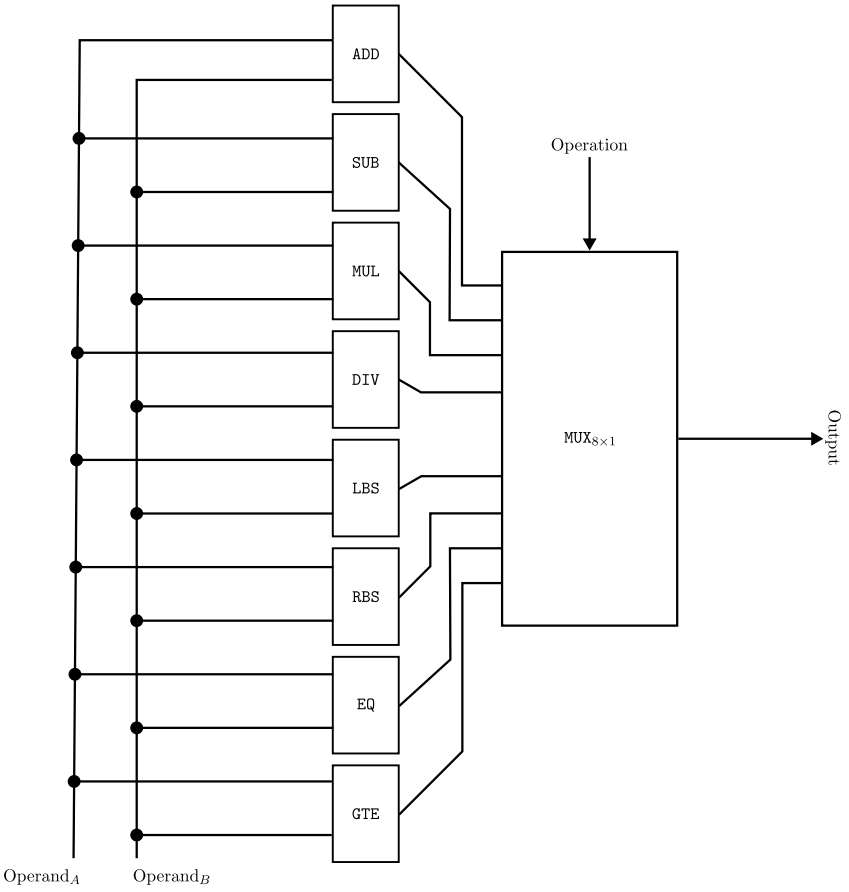


Figure 2.9: A simple logic design of an example Arithmetic Logic Unit

Chapter 3

Elements of Quantum Computing

We mentioned in the previous chapter that computer are electronic systems that process binary information is the form of voltages, thus these systems are governed by the macroscopic laws of Classical Mechanics. But as they shurnk down to the microscopic levels of the atoms, new problems arose. By manipulating and storing information in atoms, electrons, spins and *quantum systems* in general, now the computer itself must obey the laws of Quantum Mechanic, a notion that Richard Feynman introduced in the early '80s.

3.1 The Dirac Notation

We already are familiar with the standard vector notation. Let $\vec{v}$ be a n-dimensional vector whom's elements are real numbers. This is notated as:

$$\vec{v} = \sum_{i=1}^{n} v_i \hat{d}_i \in \mathbb{R}^n \tag{3.1}$$

where $\hat{d}_i$ is the unit vector of the i-th dimension. For a three-dimensional real space the vector $\vec{v}$ would be notated as:

$$\vec{v} = v_1 \hat{d}_1 + v_2 \hat{d}_2 + v_3 \hat{d}_3 \in \mathbb{R}^3 \tag{3.2}$$

The notation is similar for elements in the complex space $\mathbb{C}^n$. The only real difference is the nomenclature where vectors that are in that space are called *complex vectors*.
We can also notate vectors as a $n \times 1$ column matrix, where each element of that matrix is the vector's corresponding element:

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n \tag{3.3}$$

In Quantum Mechanics, vectors are represented using a different kind of notation, the *Dirac notation* or *bra-ket notation*. This notation was introduced by the American physicist and electrical engineer Paul Dirac in 1939 [11]. This introduced two new symbols: the *bra* (symbolised with a $\langle|$) which represents a vector quantity whose elements are a vertical $1 \times n$ matrix, and the *ket* (symbolised with a $|\rangle$) which represents a vector quantity whose elements are a horizontal $n \times 1$ matrix. We also note that in Quantum Mechanics all vectors (and their elements) are in a complex space. For example, let $\vec{v}$ be a vector in the complex space $\mathbb{V}^n$, we would notate it in Dirac notation as:

$$|v\rangle = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \text{ where } v_i \in \mathbb{C}, |v\rangle \in \mathbb{V}^n \tag{3.4}$$

This is also read as "ket v". Note that the arrow symbol ($\rightarrow$) on top of label-name of the vector $\vec{v}$ is absent with this notation.
Subsequently, the "bra v" is the *conjugate transpose* or the *Hermitian conjugate* (symbolised with † and pronounced as "dagger") of the $|v\rangle$

$$\langle v| = \begin{bmatrix} v_1^* & v_2^* & \dots & v_n^* \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots v_n \end{bmatrix}^\dagger \tag{3.5}$$

3.1.1 Inner Product

The inner product of two complex vectors $|a\rangle, |b\rangle \in \mathbb{V}^n$ is defined as

$$\langle a|b\rangle = \begin{bmatrix} a_1^* & a_2^* \dots a_n^* \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i^* b_i = c, \text{ where } a_i, b_i, c \in \mathbb{C} \tag{3.6}$$

There are also three properties defined for the inner product:

$$\langle a|b\rangle = \langle b|a\rangle^* \tag{3.7}$$

$$\langle a|\alpha b + \beta c\rangle = \alpha \langle a|b\rangle + \beta \langle a|c\rangle, \text{ where } \alpha, \beta \in \mathbb{C} \tag{3.8}$$

$$\langle \alpha a + \beta c|b\rangle = \alpha^* \langle a|b\rangle + \beta^* \langle c|a\rangle, \text{ where } \alpha, \beta \in \mathbb{C} \tag{3.9}$$

The inner product of two vectors $|a\rangle, |b\rangle \in \mathbb{V}^n$ is zero when the vectors are orthogonal to each other.

3.1.2 Hilbert Space

When in an abstract complex vector space $\mathbb{V}^n$ the operation of the inner product is defined then that space is also called a *Hilbert space*. In Quantum Computing, every vector that represents the state of the quantum system is a vector in Hilbert space.

3.2 Operators

In Quantum Mechanics, operators are mathematical objects that act on vectors and transform them. The action of an operator, let $\hat{A}$ be an operator and $|\phi\rangle$ a qubit, is a expressed as:

$$\hat{A}|\phi\rangle = |\phi'\rangle \tag{3.10}$$

There are also defined properties for operators. Two operators $\hat{A}, \hat{B}$ are equal when:

$$\hat{A}\left|\phi\right\rangle = \hat{B}\left|\phi\right\rangle \tag{3.11}$$

Addition between two operators $\hat{A}, \hat{B}$ is defined as:

$$(\hat{A}+\hat{B})\left|\phi\right\rangle = \hat{A}\left|\phi\right\rangle + \hat{B}\left|\phi\right\rangle, \forall \left|\phi\right\rangle \in \mathbb{H}^n \tag{3.12}$$

The product of two operators $\hat{A}, \hat{B}$ defines the order of operations that act on a certain vector $|\phi\rangle$ as:

$$\hat{A}\hat{B}\left|\phi\right\rangle = \hat{A}(\hat{B}\left|\phi\right\rangle) = \hat{A}\left|\phi'\right\rangle \tag{3.13}$$

We should note that the addition of two operators is commutative $\hat{A}+\hat{B} = \hat{B}+\hat{A}$ but that does not apply in general to the product of two operators $\hat{A}\hat{B} \neq \hat{B}\hat{A}$.

3.3 Matrices of Operators

It is often simpler to represent the transformations of an operator using a matrix representation. Each element of that matrix represents the transformation on a specific basis. For example, let $\hat{A}$ be an operator and $\{|i\rangle\}, i = 1, 2, \ldots n$ be a basis in a Hilbert space $\mathbb{H}^n$, the operator can be represented as $n \times n$ matrix:

$$\hat{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{bmatrix} \tag{3.14}$$

There are many operators that are useful in Quantum Mechanics (and in extension also in Quantum Computing). For example, the Identity operator, notated with an $\hat{I}$, is an operator that when applied on any vector $|\phi\rangle$ lefts it un-transformed:

$$\hat{I}\left|\phi\right\rangle = \left|\phi\right\rangle \tag{3.15}$$

and is represented as:

$$\hat{I} = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \tag{3.16}$$

3.4 Hermitian Conjugate

Let $|u\rangle$ be a state vector of a quantum system and $\hat{A}$ be an operator that operates on $|u\rangle$:

$$\hat{A}\left|u\right\rangle = \left|u'\right\rangle \tag{3.17}$$

The same logic can be applied for its bra counterpart:

$$\left\langle u\right|\hat{A}^\dagger = \left\langle u'\right| \tag{3.18}$$

where $\hat{A}^\dagger$ is the *Hermitian conjugate* of the operator $\hat{A}$ - its an operator that when operated on $\langle u|$ results in $\langle u'|$ just like when its non-Hermitian conjugate $\hat{A}$ operates on $|u\rangle$ and results in $|u'\rangle$.
Thus, when we take the conjugate of $\hat{A}^\dagger$ then:

$$(\hat{A}^\dagger)^\dagger = \hat{A} \tag{3.19}$$

The matrix of $\hat{A}^\dagger$ is defined as the *conjugate transpose* matrix of the operator $\hat{A}$:

$$\hat{A}^\dagger = (\hat{A}^T)^* \tag{3.20}$$

where $\hat{A}^T$ is the *transpose* matrix of $\hat{A}$ and it is defined as:

$$\hat{A}^T_{ij} = \hat{A}_{ji} \tag{3.21}$$

3.5 Quantum Mechanics for Quantum Computing

Before we begin to operate a quantum computer or make quantum circuits we have to discuss what are the underline laws that govern those systems.
Quantum Mechanics is a mathematical framework that is used to create physical theories or to describe properties of physical systems. It is also a framework that helps physicists to describe nature at the microscopic level, something that Classical Mechanics failed to do.
This mathematical framework has four *axioms* that are used to interconnect the physical with the mathematical world.

3.5.1 The Axioms of Quantum Mechanics

Axiom One: The State of a Quantum System

The state of a quantum system can be represented as a complex vector, which vector is in the complex Hilber space. The state of a quantum system is also called the *state vector* of that quantum system. In quantum computing usually systems are *two level systems*, that are systems that have two states (binary systems).
The *qubit* is the quantum equivalent of the classical bit - instead of storing classical information (zeros or ones) is stores quantum information expressed using some *basis states*. The most frequent state base is the *computational basis*. The computational basis describes two linearly seperable and orthogonal unitary vectors $\{|0\rangle, |1\rangle\} \in \mathbb{H}^2$. With those two states we can encode information and process it using quantum gates. Physicaly these two state have to represent a physical state of the system. Classical computers use two voltage levels (+5V and +0V, for example), in quantum computers we can use many physical properties of a quantum system like: the spin of an electron (up or down), the polarization of a photon (horizontal or vertical) or even the energy levels of an atom. A general state $|\phi\rangle$ of a system can be written as a superposition of the basis states $|0\rangle$ and $|1\rangle$:

$$|\phi\rangle = a\,|0\rangle + b\,|1\rangle = \begin{bmatrix} a \\ b \end{bmatrix} \tag{3.22}$$

where $a, b \in \mathbb{C}$ are the complex coefficients called the *probability amplitudes* of the state vector $|\phi\rangle$. We can also notate them using the name of the state vector

$$|\phi\rangle = \phi_0\,|0\rangle + \phi_1\,|1\rangle = \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} \tag{3.23}$$

Axiom Two: Time Evolution of a Quantum System

The time evolution of a quantum system $|\phi(t)\rangle$ can be expresed by unitary operators of the form $U(t, t_0)$ as follows:

$$|\phi(t)\rangle = U(t, t_0)\,|\phi(t_0)\rangle \tag{3.24}$$

Also if an operator is hermitian ($A = A^\dagger$) then the operator $U = e^{iA}$ is unitary because:

$$UU^\dagger = e^{iA}(e^{iA})^\dagger = e^{iA}e^{-iA^\dagger} = e^{iA}e^{-iA} = I \tag{3.25}$$

Axiom Three: Measurement

By now we can encode information and we can define its evolution in time, but it would be gratuitous if we could not measure the outcome of any computation. Thus the third axiom states that to make a measurement on a quantum system $|\phi\rangle \in \mathbb{H}$, defined using the basis state $|i\rangle \in \mathbb{H}$, then a measurement can occur on $|\phi\rangle = \sum_i^n a_i\,|i\rangle$. The outcome is one of the states $|i\rangle$ of the system with a probability to be in that state $|a_i|^2$. After the outcome, the system *collapses* to the that state $|i\rangle$.

Axiom Four: Composite Systems

We can combine n independent quantum systems that are expressed as state vectors $|\phi_i\rangle \in \mathbb{H}_i$ where $i = 0, 1, \ldots, n$ as a *composite quantum system* $|\phi\rangle$. This can be done by applying the tensor product to each of the independent systems

$$|\phi\rangle = |\phi_0\rangle \otimes |\phi_1\rangle \otimes \ldots \otimes |\phi_n\rangle = |\phi_0\phi_1 \ldots \phi_n\rangle \tag{3.26}$$

That state vector $|\phi\rangle$ is a complex vector in the complex Hilber space

$$\mathbb{H} = \mathbb{H}_0 \otimes \mathbb{H}_1 \otimes \ldots \mathbb{H}_n \tag{3.27}$$

As we mentioned previously, in Quantum Computing, we study two-level systems - systems with two basis states. This means that the composite system $|\phi\rangle$ of n independent two-level quantum systems is in a complex Hilbert space of 2^n dimensions.

3.6 Reversible Computation

"Classical" computers use three fundamental logic gates to process binary information: the AND, OR and NOT gates as we have introduced them in the previous chapter. A very simple question can rise from analyzing these logic gate "Can we know the state of the inputs by just looking at the outputs"? In other words, "Can we reverse the computation done by those logic gates"?
The NOT gate is *reversible* because its output is the negation of the input and we can infer the input by just negating again. This is also a postulate of the Boolean Algebra:

$$\neg(\neg x) = x \tag{3.28}$$

The AND and OR gates are not reversible. For the AND gate, we have three outputs that result in a zero (0) and only one possibility for an output of one (1), thus we cannot infer for three outputs what the inputs are with absolute certainty. For the OR gate is the same but instead of three outputs of zero this gate has three possibilities of outputs of one (1). It seems like we lost some information of what the inputs were. In fact we do lose information in form of heat [12]. We can analyze for the other compound gates: NAND, NOR, XOR and XNOR. We can find out easily that all of those gate are not reversible for the same reasons.
The most important thing to consider is the energy loss in the form of heat. It has been shown [13] that by making logic components reversible we can save a lot of power generated by the computer. These kind of

components are called *reversible logic gates*. This idea of reversible computation is the basis that Feynman used to impose the idea of the Quantum Computer, a computer that uses reversible components that obey the laws of Quantum Mechanics [1].

3.7 Landauer's Principle

When one bit of information is erased, the least amount of energy is expelled to the enviroment given by the following equation:

$$E = k_B T \ln 2 \tag{3.29}$$

where k_B is the *Boltzmann constant* and T is the temperature of the enviroment. This is know as the *Landauer's Principle*[12]. We can use this equation to calculate the total work produced by the computer by defining a Boolean function $f : \{0,1\}^m \to \{0,1\}^n$ that takes m-input bits and produces n-output bits - this funtion will represent all of the logical operations of the computer. If we consider that $m >> n$, this means that some bits are erased each time the computer carries a computation[14], then the total energy loss using equation (3.8) is:

$$E_{total} = (m-n) k_B T \ln 2 \tag{3.30}$$

Because Equation (3.9) gives the minimum amount of energy produced, the real amount of energy produced by modern computers is much higher that E_{total}.

3.8 The Qubit

As we have already described in a previous section, qubits are the fundamental units of measuring the information contents of a quantum system. In classical terms, computers use two voltage levels to represent the two states of a bit: choosing arbitrarily, +0V for the zero state and +5V for the one state. Quantum systems map those logical states of qubits to some physical property of the quantum system, which can be its spin, for electrons, or its polarity, for photons. Mathematically we represent those states (0 and 1) with the *unit vectors* $|0\rangle$ and $|1\rangle$ which constitute the computational basis.
These basis are complex vectors $|0\rangle, |1\rangle \in \mathbb{H}^2$ and are written as:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ and } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.31}$$

The big difference from the classical bit is that qubits are a superposition of some two basis states. Although this is a very interesting phenomenon, computation constitutes that at some point we would need to output/read the result. We have to remind at this point that this would constitute a measurement and the qubit would have to "collapse" to one of the two basis states with a probability equal to the square of the probability amplitude of that basis state.

3.8.1 The Bloch Sphere

The Bloch sphere is an attempt to depict the state of qubit as a vector that originates from the center of a sphere with a radius $r = 1$.
Let $|q\rangle = q_0 |0\rangle + q_1 |1\rangle \in \mathbb{H}^2$ be a qubit with $q_0, q_1 \in \mathbb{C}$ be its probability amplitudes. We can parameterize those coefficients as:

$$q_0 = e^{i\gamma} \cos \frac{\theta}{2} \qquad\qquad q_1 = e^{i\delta} \sin \frac{\theta}{2} \tag{3.32}$$

We can now re-write the qubit $|q\rangle$ as:

$$|q\rangle = e^{i\gamma}(\cos \frac{\theta}{2} |0\rangle + e^{i\phi} \sin \frac{\theta}{2} |1\rangle) \tag{3.33}$$

where $\phi = \gamma - \delta$. The *global phase* $e^{i\gamma}$ is neglected most of the time thus qubit $|q\rangle$ can be written as a matrix:

$$|q\rangle = \begin{bmatrix} \cos \frac{\theta}{2} \\ e^{i\phi} \sin \frac{\theta}{2} \end{bmatrix} \tag{3.34}$$

To map the *spherical coordinates* (r, ϕ, θ) to cartesian coordinates we use the equalities:

$$q_x = r \sin\theta \cos\phi \tag{3.35}$$
$$q_y = r \sin\theta \sin\phi \tag{3.36}$$
$$q_z = r \cos\theta \tag{3.37}$$

So the qubit $|q\rangle$ can be plotted with coordinates (q_x, q_y, q_z).

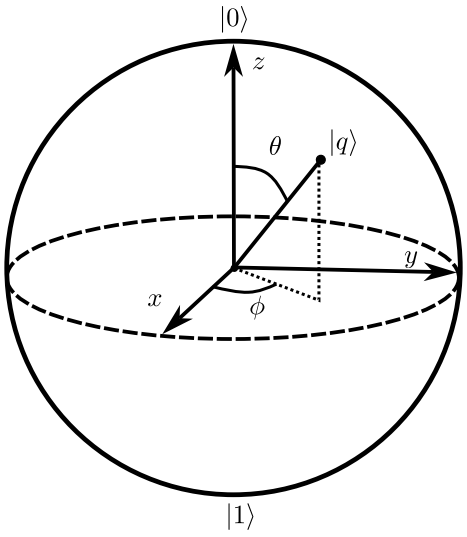


Figure 3.1: The Bloch sphere

We can note also the rest of the basis state that a qubit can have:

1. along the z-axis we can distinguish the computational basis (see 3.1):
   (a) $|0\rangle$ where the $+z$-axis crosses Bloch sphere's upper boundary
   (b) $|1\rangle$ where the $-z$-axis crosses Bloch shpere's lower boundary
2. along the x-axis:
   (a) at the cross section $+x$-axis where angles $\theta = \pi/2$ and $\phi = 0$ we have the basis state $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$

(b) at the cross section $-x$-axis where angles $\theta = \pi/2$ and $\phi = \pi$ we have the basis state $|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$

3. along the y-axis:

(a) at the cross section $+y$-axis where angles $\theta = \pi/2$ and $\phi = \pi/2$ we have the basis state $|i+\rangle$

(b) at the cross section $-y$-axis where angles $\theta = \pi/2$ and $\phi = 3\pi/2$ we have the basis state $|i-\rangle$

In this work we are going to be consered mostly with the computational basis states.

3.9 Quantum Gates

Computers use logic gates that map the functions of boolean logic functions. We have talked about the AND, OR and NOT logic gates. These gates are used to construct the operational units of a "classical" computer. In contrast, quantum computers need logic gates that adhere to the laws of Quantum Mechanics - the foremen-tioned *quantum gates*.
There are two kinds of quantum gates:

1. single-qubit gates, quantum gates that operate on only one qubit, and
2. multi-qubit gates, quantum gates that operate on multiple qubits or *quantum registers*.

3.9.1 Quantum Registers

On that note, a *quantum register* is the tensor product of multiple qubits. For example, if we wanted to store the binary number $101_2 = 5_{10}$ on a quantum computer, we would need three qubits $|x\rangle = |1\rangle, |y\rangle = |0\rangle, |z\rangle = |1\rangle$. By combining them using the tensor operator we create a quantum register

$$|\phi\rangle = |x\rangle \otimes |y\rangle \otimes |z\rangle \text{ or } |\phi\rangle = |1\rangle \otimes |0\rangle \otimes |1\rangle \tag{3.38}$$

We can also write a shorthand notation for the quantum register $|\phi\rangle$:

$$|\phi\rangle = |xyz\rangle \text{ or just } |\phi\rangle = |101\rangle \tag{3.39}$$

where $|\phi\rangle \in \mathbb{H}^3$.

3.9.2 Single Qubit Gates

The Identity gate is a quantum gate that operates on a single qubit and leaves it untransformed. In matrix notation is represented as:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.40}$$

Let $|\phi\rangle = \phi_0 |0\rangle + \phi_1 |1\rangle$, the Identity gate will tranform the single qubit as

$$I |\phi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} = \begin{bmatrix} 1 \cdot \phi_0 + 0 \cdot \phi_1 \\ 0 \cdot \phi_0 + 1 \cdot \phi_1 \end{bmatrix} = \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} = |\phi\rangle \tag{3.41}$$

The schematic representation of the Identity gate is
The NOT gate or X gate is a quantum gate that operates on a single qubit. It tranforms the qubit by swapping its amplitude coefficients. Let $|\phi\rangle = \phi_0 |0\rangle + \phi_1 |1\rangle$, the the X gate will tranform it as

$$X |\phi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix} = \begin{bmatrix} 0 \cdot \phi_0 + 1 \cdot \phi_1 \\ 1 \cdot \phi_0 + 0 \cdot \phi_1 \end{bmatrix} = \begin{bmatrix} \phi_1 \\ \phi_0 \end{bmatrix} \tag{3.42}$$

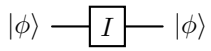


Figure 3.2: The schematic representation of the Identity gate

If the qubit $|\phi\rangle$ was either on state $|0\rangle$ or $|1\rangle$ this gate would act as a logical negation - flipping the state of the qubit. For example, let $|\phi\rangle = |0\rangle = 1 \cdot |0\rangle + 0 \cdot |1\rangle$ then

$$X|\phi\rangle = 0 \cdot |1\rangle + 1 \cdot |1\rangle = |1\rangle = |\phi'\rangle \tag{3.43}$$

The schematic representation of the X gate is

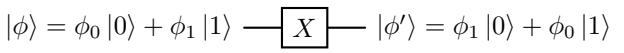


Figure 3.3: The schematic representation of the X gate

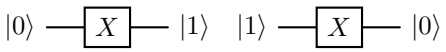


Figure 3.4: The X gate actions when the input qubit is at the $|0\rangle$ and $|1\rangle$ state respectively

The Hadamard gate is a quantum gate that operates on single qubits. It tranforms the qubit by putting it on a superposition of its basis states. In matrix notation is represented as

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{3.44}$$

Let a single qubit $|\phi\rangle$ have the computation basis $|0\rangle, |1\rangle$, then $|\phi\rangle = \phi_0|0\rangle + \phi_1|1\rangle$. The Hadamard gate will tranform it as

$$H|\phi\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}\phi_0\\ \phi_1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\cdot\phi_0 + 1\cdot\phi_1\\ 1\cdot\phi_0 - 1\cdot\phi_1\end{bmatrix} = \frac{1}{\sqrt{2}}[(\phi_0+\phi_1)|0\rangle + (\phi_0-\phi_1)|1\rangle] \tag{3.45}$$

The schematic representation of the Hadamard gate is
We can analyze further by assigning $|\phi\rangle$ to be on the two computation basis states:

1. for $|\phi\rangle = |0\rangle$:

$$H|\phi\rangle = H|0\rangle = H(1|0\rangle + 0|1\rangle) = \frac{1}{\sqrt{2}}[(1+0)|0\rangle + (1-0)|1\rangle] = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle \tag{3.46}$$

2. for $|\phi\rangle = |1\rangle$:

$$H|\phi\rangle = H|1\rangle = H(0|0\rangle + 1|1\rangle) = \frac{1}{\sqrt{2}}[(0+1)|0\rangle + (0-1)|1\rangle] = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle \tag{3.47}$$

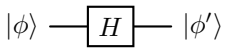


Figure 3.5: The schematic representation of the Hadamard gate

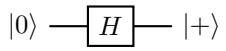


Figure 3.6: The tranformation of the $|0\rangle$ state to the $|+\rangle$ state

3.9.3 Multi-Qubit Gates

The Controlled-NOT gate or CX gate is a quantum gate that operates on two qubits. This gate operates by having a *control* and *target* qubit. If the control qubit is on the $|1\rangle$ state then the target qubit is tranformed by an X gate tranformation - practically flipping its basis state. In matrix notation is represented as

$$CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.48}$$

Let $|\phi\rangle$ be a quantum register of two qubits $|x\rangle, |y\rangle$. Then $|\phi\rangle = |x \otimes y\rangle = |xy\rangle$. We shall note the matrix and linear notation of the quantum register is

$$|\phi\rangle = \begin{bmatrix} x_0 y_0 \\ x_0 y_1 \\ x_1 y_0 \\ x_1 y_1 \end{bmatrix} = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix} = \phi_0 |00\rangle + \phi_1 |01\rangle + \phi_2 |10\rangle + \phi_3 |11\rangle \tag{3.49}$$

The CX gate will act upon $|\phi\rangle$ as follows

$$CX |\phi\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \end{bmatrix} = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_3 \\ \phi_2 \end{bmatrix} = \phi_0 |00\rangle + \phi_1 |01\rangle + \phi_3 |10\rangle + \phi_2 |11\rangle \tag{3.50}$$

The schematic representation of the CX gate is
Lastly we will discuss the Controlled-CX gate or CCX gate. The CCX gate is quantum gate that operates on three qubits. This gate is similar to the CX gate but instead of having only one control qubit it has two. In matrix notation is represented as

$$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.51}$$

Figure 3.7: The tranformation of the $|1\rangle$ state to the $|-\rangle$ state

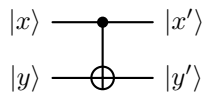


Figure 3.8: The schematic representation of the CX gate

Let $|\phi\rangle$ be a quantum register of three qubits $|x\rangle, |y\rangle, |z\rangle$. Then $|\phi\rangle = |x \otimes y \otimes z\rangle = |xyz\rangle$. We shall note the matrix and linear notation of the quantum register is

$$|\phi\rangle = \begin{bmatrix} x_0y_0z_0 \\ x_0y_0z_1 \\ x_0y_1z_0 \\ x_0y_1z_1 \\ x_1y_0z_0 \\ x_1y_0z_1 \\ x_1y_1z_0 \\ x_1y_1z_1 \end{bmatrix} = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \phi_5 \\ \phi_6 \\ \phi_7 \end{bmatrix} = \phi_0 |000\rangle + \phi_1 |001\rangle + \phi_2 |010\rangle + \phi_3 |011\rangle + \phi_4 |100\rangle + \phi_5 |101\rangle + \phi_6 |110\rangle + \phi_7 |111\rangle \tag{3.52}$$

The CCX gate will act upon $|\phi\rangle$ as follows

$$CCX |\phi\rangle = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix} \begin{bmatrix} \phi_0 \\ \phi_1 \\ \vdots \\ \phi_6 \\ \phi_7 \end{bmatrix} = \begin{bmatrix} \phi_0 \\ \phi_1 \\ \vdots \\ \phi_7 \\ \phi_6 \end{bmatrix} = \phi_0 |000\rangle + \phi_1 |001\rangle + \ldots + \phi_7 |110\rangle + \phi_6 |111\rangle \tag{3.53}$$

The schematic representation of the CCX gate is

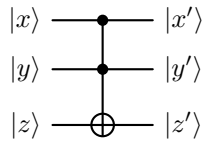


Figure 3.9: The schematic representation of the CCX gate

Chapter 4

The Qiskit SDK

The Qiskit SDK is an open-source framework used to build, simulate, compile and verify Quantum circuits using IBM's Quantum Platform. This software was launched by IBM in 2017 and it is maintained and updated by its open-source community on GitHub [15]. As of writing this thesis Qiskit just announced version `1.0`. In this section we will briefly describe the basics of using the Qiskit SDK.

4.1 Building Quantum Circuits

Building Quantum circuit using Qiskit is very easy. Qiskit is written in the Python Programming Language and thus it can be installed by the user from its official Python Package Index using `pip` [16]. After installing the dependencies we can `import` the Quantum Circuit class from the Qiskit API. Then we can instanciate an `QuantumCircuit` object with a number of qubits and bits. We can then apply various Quantum gates onto the circuit. For example we can build a Quantum circuit with two qubits, q_0 and q_1, and then apply a Hadamard gate on q_0 and an X gate on q_1.

```
from qiskit import QuantumCircuit

circuit = QuantumCircuit(2)

circuit.h(0)
circuit.x(1)

print(circuit.draw("latex_source"))
```

q_0 : — H —
q_1 : — X —

Listing 1: Building a simple Quantum circuit and its diagram compiled from its LaTeXsource code

Qiskit also provides us with functions that can visualize a circuit. Every QuantumCircuit object has a member function for printing the circuit using many different output methods like: ASCII text (great if you are in a CLI enviroment), as a Matplotlib plot if you are in an interactive Python enviroment or output the LaTeXsource code of the circuit diagram for the highest resolution.

4.2 Measuruments in Qiskit

Measuring the outcome of the qubits in a Quantum circuit is very important in Quantum Computing because it allows us to verify our algorithm quantitativily. Qiskit allows us to measure in two ways: measure a qubit into a specific bit or measure every qubit at once into a classical register.
If we want to measure each qubit or group of qubit into specific bits or classical registers we must ensure to instanciate the circuit object with the appropriate amount of bits. This can be done by passing an integer as the second parameter on instanciation. Then we can specify which qubit line we want to measure into which bit line by invoking the `measure()` member method of the circuit object.

```
from qiskit import QuantumCircuit

circuit = QuantumCircuit(2, 2) # 2 qubits + 2 bits

circuit.h(0)
circuit.x(1)

circuit.measure(0, 0) # measure q_0 and store the outcome on c_0
circuit.measure(1, 1) # measure q_1 and store the outcome on c_1

print(circuit.draw("latex_source"))
```

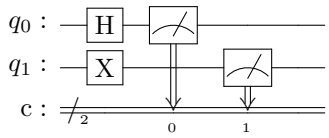


Figure 4.1: Measuring qubit states into the appropriate bits

Finally, if we want to measure all qubits at once we can invoke the `measure_all()` member method of the circuit object. Note that we do not need to specify how many bits the cirucit needs, Qiskit auto-calculates the appropriate amount and supplies the circuit with a pre-allocated classical register named `meas`.

4.3 Custom Quantum Gates

Qiskit let us create our own gates. This can be done by build the desired gate from the basis gate set supplied by the Qiskit SDK and then invoke the `to_gate()` member method of the circuit object. For example, we can create a new gate G from the circuit we created from the previous section. After we create the custom gate we can use the `append` member method to append it to our base circuit.

4.4 Local Simulations, Transpilation and Sending a Job on a Real Quantum Computer

After measuring the Quantum circuit we can use Qiskit to execute a Quantum circuit in two ways:

1. execute it using a simulator on your local machine or
2. send your circuit to be run as a *job* on real Quantum hardware

```
from qiskit import QuantumCircuit

circuit = QuantumCircuit(2, 2)

circuit.h(0)
circuit.x(1)

circuit.measure_all() # measure everything in one batch

print(circuit.draw("latex_source"))
```

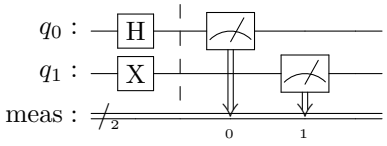


Figure 4.2: Measuring all the qubit outcomes of the circuit at once

We are going to mention how to execute Quantum circuits using both ways.
IBM provides a high performance local simulator called *Aer* (pronounced *air*). There are even two versions of the Aer simulator: a GPU-based simulator that can use a *Graphics Processing Unit* to accelarate the simulation execution or a CPU-based simulator. We are going to use the later and more specifically on a Intel i5 four core, eight thread CPU. We did this as to prove that these kinds of circuits can run even on moderately modern hardware.

4.4.1 The Aer Simulator

For this example we will execute the simple Quantum circuit that we created on the previous sections.
Before importing anything we should install the `Aer` simulation library. Again this is extremely simple as we can use `pip` to install those dependencies automatically. After the installation, we can import the `AerSimulator` class from the `qiskit_aer` library we just installed and instanciate it as an object named `sim`.
We should note that to execute circuits with custom gate components we should *transpile* the circuit before executing it. Transpilation in Qiskit is the process of converting a Quantum circuit using a simpler set of *basis gates*. Transpilation is very important in Qiskit and it will come handy when we want to execute circuits on real Quantum computers. Transpilation is taken care of by the `transpile()` method from the `qiskit` library. It takes two positional arguments: the circuit(s) to be transpiled and the simulator that the circuit(s) will be transpiled for. Strictly speaking this simple example circuit does not need any further simplification but for the purposes of demonstrations we will transpile the circuit.
The final step is to invoke the `run` member method of the instanciated simulator. We can also invoke the `result()` method right after the invokation of the `run()` method to get back the result counts of the execution.
The *result counts* are the counts of the probabilities for each qubit in the Quantum circuit. The default max count limit is 1024 and it can be changed accordingly. We will leave the default values. To plot those result we will use the `qiskit.visualization` library that comes with the base Qiskit installation. The `plot_histogram()` method plots the result as histogram where the x-axis is the output state of the Quantum circuit and the y-axis is how many times this was the output. Note that to get the counts of the result invoke the `get_counts()` method.

```python
from qiskit import QuantumCircuit

example_circuit = QuantumCircuit(2, name="G")

example_circuit.h(0)
example_circuit.x(1)

example_gate = example_circuit.to_gate()

circuit = QuantumCircuit(2)

circuit.append(example_gate, [0, 1])
print(circuit.draw("latex_source"))
```

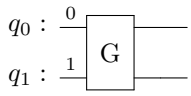


Figure 4.3: Creating a custom gate G and appending it to another circuit

```python
from qiskit_aer import AerSimulator
from qiskit import transpile

sim = AerSimulator()
circuit = transpile(circuit, sim)
result = sim.run(circuit).result()
```

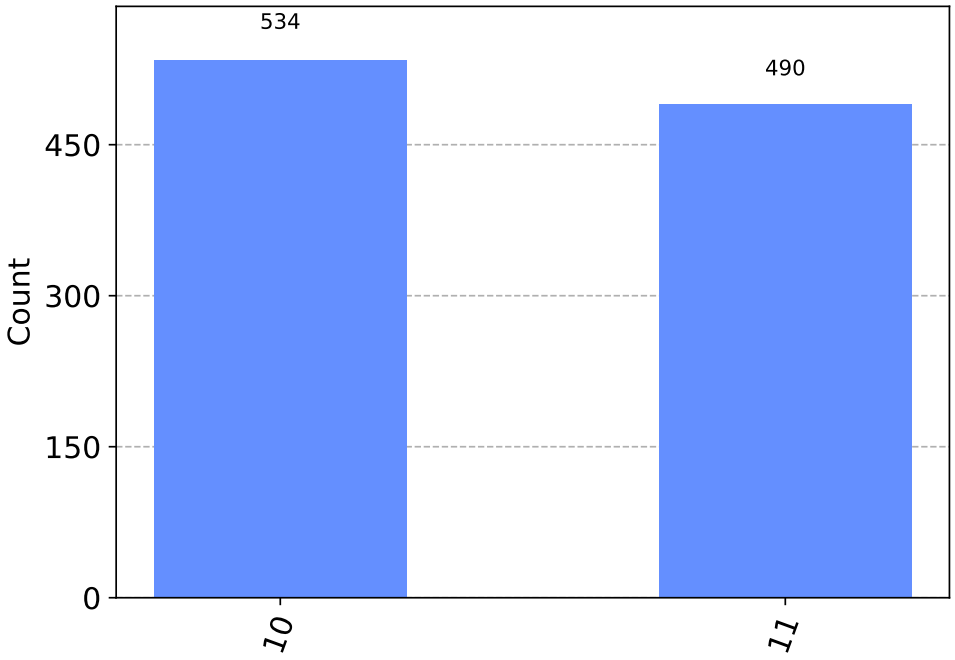
Listing 2: Executing the example Quantum circuit on a local CPU-based Aer simulator

We can see that on the x-axis we have two outputs: `10` with a count of 534 and `11` with a count of 490 which count up to 1024. This means that qubit q_0 is 50% of the time in the $|0\rangle$ state and the other 50% of the time in the $|1\rangle$ state. This makes sense because we applied earlier a Hadamard gate on the qubit putting it on superposition of those two basis states. As for q_1, for every qubit is initialized to the $|0\rangle$ state, and by applying the X gate we invert it from $|0\rangle$ to $|1\rangle$.
We would like to note that the bitstring label for each output in the histogram is represented in little-endian, meaning that the first measurement of a qubit is going to be displayed as the least-significant bit of that label.

4.4.2 Executing on a Real Quantum Computer

In general, executing a Quantum circuit on a real Quantum computer is not much different than executing on a local simulator, although, some additional steps are needed.
First of all, the IBM Quantum Platform provides users with nine different *compute resources*, essential Quantum computers that are avalaible for upto 10 minutes of compute time per month for each user. Each of these Quantum computers are based on a different set of basis gates. For example, the `ibm_osaka` computer is based on IBM's Eagle r3 Quantum Processor, scoring 127 qubits in total and with basis gates: ECR, ID, RZ, SX and X gates.
This means that for each Quantum computer, our circuit needs to be transpiled specificly in mind of the basis

```
from qiskit.visualization import plot_histogram
plot_histogram(result.get_counts())
```



Listing 3: The plotted result of the simualtion for the example circuit

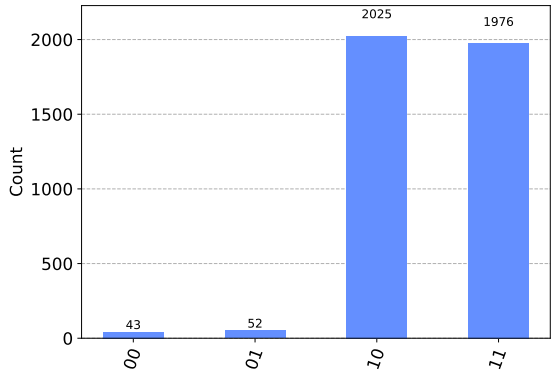
gates of the *target compute resource*.
On that note, an extra library must be installed on our system to configure the circuit to be sent over the Internet to IBM's Quantum Platform, the `QiskitRuntimeService`. This is a trivial task, as it can be installed by `pip`. After the installation, we can instanciate a `QiskitRuntimeService` object named `service`. IBM recommends to configure the service by saving your credentials (API Token) locally. This can be done by invoking the `save_account()` member method of the `service` object. We can then specify which *channel*, or service, and *token* (your acount's API token). It is additionaly recommented to set the positional argument `set_as_default` to **`True`**. We should note that this is an extremely unsafe operational because anyone with access to our operating system user account has direct access to our IBM Quantum Platform account.
After that everytime we instanciate a `QuantumRuntimeService` object the configuration is automatically set to use our credentials.
The next step is to find a Quantum computer to execute our circuit. Qiskit provides the very handy `least_busy()` member method, that finds the least busy backend (Quantum computer) at that specific time. This method also can specify what kind of backend we may request, because IBM provides also simulator in addition to real Quantum computers, with setting the positional argument `simulator` to **`False`**.
The next step is the transpilation. IBM recommends to build a target-specific transpiler which using the `generate_preset_passmanager()` method from the `qiskit.transpile` library.
This method can also be configured to optimize the circuit at a specific level by the positional argument `optimization_level` and set a target backend which will be used to transpile the circuit to use the set of basis gates of that target.

After all the initializations and configurations we can run the transpiler with our circuit by invoking the `run` member method from the generated transpiler.
Finally, we have to instanciate a *Qiskit primitive*. A Qiskit primitive is an abstraction layer that lets users specify on a high-level settings to be passed onto the runtime without implementing it by hand. We will use a pre-configured Qiskit primitive, the *SampleV2*. This primitive will configure the runtime to return the probabilities of the output qubits of our circuit, something like the counts results we got from the previous subsection.

```
from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit.transpiler.preset_passmanagers import
    generate_preset_pass_manager
from qiskit_ibm_runtime import SamplerV2 as Sampler

service = QiskitRuntimeService()
backend = service.least_busy(operational=True, simulator=False)
pm = generate_preset_pass_manager(optimization_level=1, backend=backend)
circuit = pm.run([circuit])
job = Sampler(backend).run([circuit])
result = job.result()

plot_histogram(result[0].data.meas.get_counts())
```



Listing 4: Executing the example circuit on the IBM Osaka Quantum Computer

As we can see two new bars appear seemingly out of nowhere. We can see that 0.01% of the time the output is in the $|00\rangle$ state and 0.012% of the time in the $|01\rangle$ state. This is expectable because modern state-of-the-art Quantum computers are very susceptible to noise. This is a very good example of this phenomenon, where noise introduces unexpected and unwanted behavior to our system, where previously, when executed on a simulator, there was none of this ouput. This is a very serious problem for scientists and engineers that construct Quantum computers. Unfortunately, we are not going to cover noise mitigitation strategies but this work could definitely benefit from these kinds of strategies because as you introduce more qubits and more gates onto your circuit design noise will affect the behavior of your system over-all.

Chapter 5

QALU's Implementation with the Qiskit SDK

In this chapter we will present five individual quantum circuits that perform fundamental arithmetic and logic operations: addition, subtraction, multiplication and magnitude comparisons (less, greater than or equal). We are also going to present the Python code that implement those circuits as well as their diagrammatic representations.

5.1 The Quantum Half Adder

In this section we are going to construct, using the Qiskit SDK, a quantum circuit that implements the classic Half-Adder circuit. First we shall consider the classic circuit diagram of the Half-Adder circuit from Chapter 2.

5.1.1 Analysing the diagram and logic of the Quantum Half-Adder

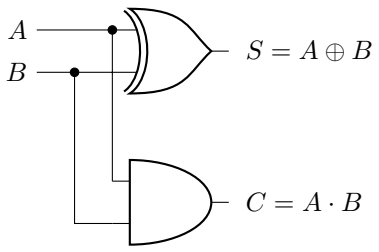


Figure 5.1: The classical Half-Adder circuit diagram

The circuit takes two 1-bit inputs, A and B and produces two 1-bit outputs S (the sum, $A + B$) and C (the possible carry of S). As we can see from the diagram, to compute the sum of the inputs we have to apply an XOR gate on the inputs. As for the carry, we apply the AND gate. From this simple analysis we can produce the truth table of this circuit to better construct its quantum-counterpart.
Now the construction of the quantum Half-Adder is matter of matching the boolean functions, the XOR and AND function to be exact, to the appropriate quantum gates.
Let A and B be two 1-qubit inputs. The XOR boolean function maps directly with the Feynman gate (or controlled-NOT/CNOT gate). We can deduce that from comparing the two gates truth table.
We omitted the third column from table (a) to better emphasize the similarities of the two gates.

A	B	$S = A \oplus B$	$C = A \cdot B$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 5.1: The truth table of the classical Half-Adder circuit

$\|A\rangle$	$\|B\rangle$	$\|B\rangle' = \|A \oplus B\rangle$
0	0	0
0	1	1
1	0	1
1	1	0

(a) CNOT's truth table

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

(b) XOR's truth table

Table 5.2: The truth tables of the CNOT (a) and XOR (b) gates side-by-side

The AND boolean function cannot be mapped directly to any quantum gate/operation because it is not reversible. This does not mean that it is impossible to construct the same functionality using quantum gates. We shall borrow a "garbage" qubit $|O\rangle = 0$ to simulate the boolean AND function. By applying the Toffoli gate with control qubits $|A\rangle$, $|B\rangle$ and the borrowed qubit $|O\rangle$, as the target, we simulate the AND functionality directly.

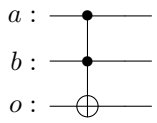


Figure 5.2: The Toffoli gate diagram

At the end of the computation, $|O\rangle$ will store the conjunction of $|A\rangle$ and $|B\rangle$. This can be seen more clearly from the side-by-side view of the truth tables of the two gates.

$\|A\rangle$	$\|B\rangle$	$\|O\rangle$	$\|O\rangle' = \|A \cdot B\rangle$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1

(a) CCNOT's truth table

A	B	$A \cdot B$
0	0	0
0	1	1
1	0	1
1	1	0

(b) AND's truth table

Table 5.3: The truth tables of the Toffoli/CCNOT (a) and AND (b) gates side-by-side

The complete circuit has to be constructed with caution because if we apply the CNOT gate first, we inevitably change the state of $|B\rangle$. With this knowledge at hand we have to apply the CCNOT gate as the first computational step and later the CNOT gate.
We are going to enumerate over each computational step:

1. $|O'\rangle = CCNOT\,|A\rangle \otimes |B\rangle \otimes |O\rangle = CCNOT\,|A, B, O\rangle = |A \oplus B\rangle$

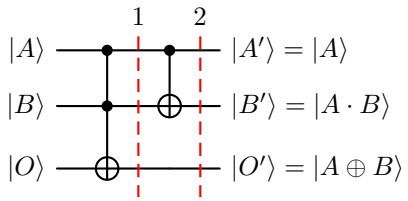


Figure 5.3: The quantum Half-Adder circuit with slices that indicate each computational step

2. $|B'\rangle = CNOT\,|A\rangle \otimes |B\rangle = |A \cdot B\rangle$

$\|A\rangle$	$\|B\rangle$	$\|O\rangle$	$\|A'\rangle = \|A\rangle$	$\|B'\rangle = \|C\rangle$	$\|O'\rangle = \|S\rangle$
0	0	0	0	0	0
0	1	0	0	0	1
1	0	0	1	0	1
1	1	0	1	1	0

Table 5.4: The truth table of the quantum Half-Adder circuit

As we can see $|B'\rangle$ stores the carry of A and B and $|O'\rangle$ stores the sum of A and B.

5.1.2 Implementing the Quantum Half Adder in Python

First we need to import some useful classes from the Qiskit library. `QuantumCircuit` and `QuantumRegister` will be used to create the Half-Adder circuit.

```
from qiskit import QuantumCircuit, QuantumRegister
```

Listing 5: The initial imports for the quantum Half-Adder circuit

After importing the initial classes we instantiate three object A, B and O, which are all `QuantumRegister`'s and one object which is a `QuantumCircuit`.

```
a = QuantumRegister(1, name="A")
b = QuantumRegister(1, name="B")
o = QuantumRegister(1, name="O")
circuit = QuantumCircuit(a, b, o)
```

Listing 6: Instantiating the circuit object of the quantum Half-Adder circuit

We can now apply all the necessary operations to compute the sum and the carry of A and B. First we apply the CCNOT gate to the circuit, with control qubits A, B and as the target qubit O. This can be done by calling the `circuit.ccx()` method of the `QuantumCircuit` class.
Lastly we apply the CNOT gate, with control qubit A and target qubit B. This can be done by calling the `circuit.cx()` method of the `QuantumCircuit` class.
Putting it all together, we can implement the computational steps from Figure 5.3.

```
circuit.ccx(a, b, o)
circuit.cx(a, b)
```

Listing 7: The computational steps of the Half-Adder in Python3

5.1.3 Executing the Quantum Half Adder circuit

We shall execute the Quantum Half Adder circuit that we implemented above first using the Aer Simulator and then using a real Quantum Computer from the IBM Quantum Platform.
We initialize the qubits A and B with the state $|1\rangle$ and O with the state $|0\rangle$, thus we perform the half addition of $1 + 1$, and run the simulator. The results of the execution can be seen in Figure (5.4).

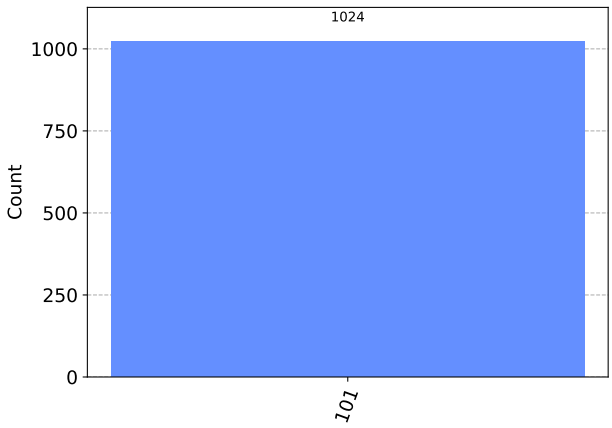


Figure 5.4: The result of executing the Quantum Half Adder circuit using the Aer Simulator

We can note that the result of the output $101 = |101\rangle = |AB'O'\rangle = |ASC_{out}\rangle$. To read the output with the correct endianness we must invert it and we shall read it as $|C_{out}SA\rangle$. If we exclude qubit A the result is 10_2 which is 2 in decimal.
Running the circuit with the same initial states for A, B and O the IBM Osaka Quantum Computer gives the following result show in Figure (5.5).
We can clearly see that the Quantum Computer resulted in more outputs that the simulator. This is due to errors while executing the circuit for 4096 times. Although some times the Quantum Computer gave the wrong answers, the most probable output was the correct one (101), with a probability of $\sim 89.5\%$.

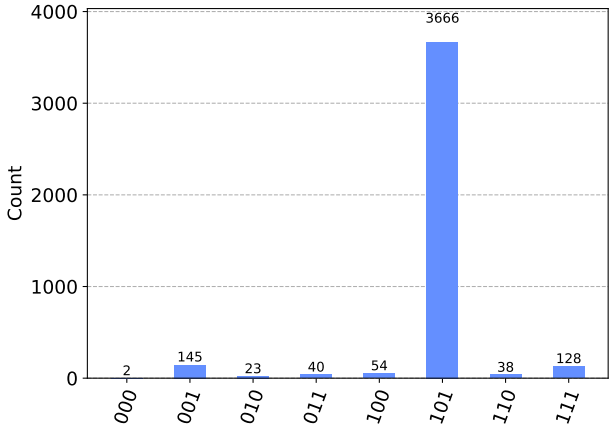


Figure 5.5: The result of executing the Quantum Half Adder circuit on the IBM Osaka Quantum Computer

5.2 The Quantum Full Adder

In this section we are going to construct, using the Qiskit SDK, a quantum circuit that implements the classic Full-Adder circuit. We are also going to extend this circuit to compute the full addition of any n-qubit inputs.

5.2.1 Analysing the diagram and logic of the Quantum Full-Adder

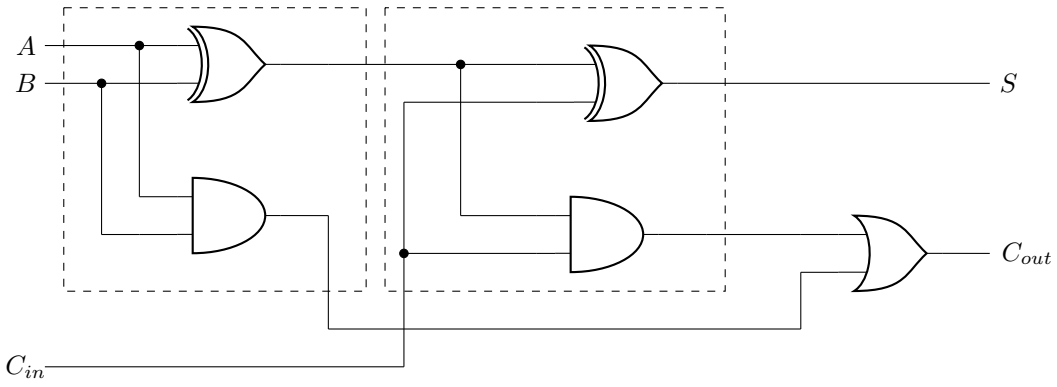


Figure 5.6: The classical Full-Adder circuit diagram

The Full-Adder circuit is a circuit that computes the sum S and carry C_{out} of two 1-bit inputs A and B whilist regarding a possible 1-bit carry from some previous addition C_{in}. This is a much more complex circuit than the previously mentioned Half-Adder. It requires two XOR gates, two AND gate and on OR gate. Although it seems that this circuit is bigger it can be made more simple by just considering two groups of gates. We have indicated with two dash-lined boxes the gate-groups that are equivalent to Half-Adder circuits.
The first Half-Adder takes inputs A and B and produces two outputs S_1 and C_1. The second Half-Adder takes inputs S_1 and C_{in} and produces S (which is the sum of $A + B + C_{in}$) and C_2. At last an OR gate is applied to C_1 and C_2 producing C_{out}, the carry out.
We are going to implement the quantum-equivalent with the same techinque that we used to make the quantum Half-Adder circuit. Just like the quantum Half-Adder, the AND gates need additional "garbage" qubits to be implemented in a quantum fashion. Let $|O\rangle$ be the "garbage" qubit that will help to simulate the AND function.

A	B	C_{in}	$S = A \oplus B \oplus C_{in}$	$C_{out} = (A \oplus B) \cdot C_{in} + A \cdot B$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 5.5: The truth table of the classical Full-Adder circuit

Also, let $|A\rangle$, $|B\rangle$ and $|C_{in}\rangle$ be the inputs.

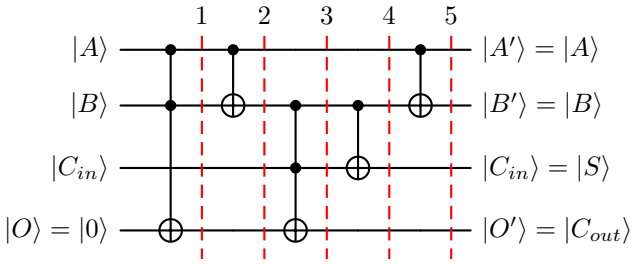


Figure 5.7: The diagram of the quantum Full-Adder with slices that indicate each computational step

We are going to enumerate through all of the computational steps:

1. $|O\rangle = CCNOT\,|A, B, O\rangle = |C_1\rangle$
2. $|B\rangle = CNOT\,|A, B\rangle = |S_1\rangle$
3. $|C_1\rangle = CCNOT\,|S_1, C_{in}, C_1\rangle = |C_{out}\rangle$
4. $|S_1\rangle = CNOT\,|S_1, C_{in}\rangle = |S\rangle$
5. $|B'\rangle = CNOT\,|A, S_1\rangle = |B\rangle$

We can see that steps 1-2 and 3-4 are gate sequences matching the behaviour of the quantum Half-Adder circuit. This means that we can replace those gates with a gate that implements the quantum Half-Adder. Let QHA be a unitary operator:

$$QHA = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{5.1}$$

Now we can edit Figure 5.5 by using the QHA.

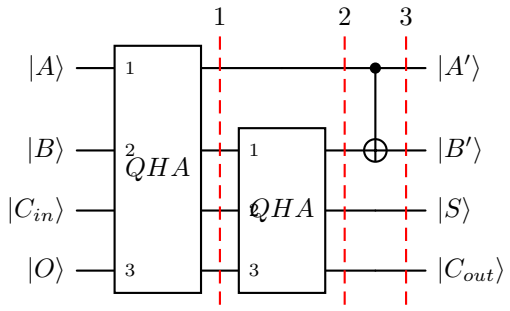


Figure 5.8: The combinational quantum Full-Adder made with two Quantum Half-Adders

5.2.2 Implementing the Quantum Full Adder in Python

Just like the Half-Adder implementation, we need to import the two basic classes: `QuantumCircuit` and `QuantumRegister`. We initialize the `circuit` object with the appropriate 1-qubit registers A, B, C_{in} and O.

```
from qiskit import QuantumCircuit, QuantumRegister

a = QuantumRegister(1, name="A")
b = QuantumRegister(1, name="B")
cin = QuantumRegister(1, name="Cin")
o = QuantumRegister(1, name="O")
circuit = QuantumCircuit(a, b, cin, o)
```

Listing 8: Initializing the quantum Full-Adder circuit

According to Figure 5.5 we have to apply five computational steps.

```
circuit.ccx(a, b, o)
circuit.cx(a, b)
circuit.ccx(b, cin, o)
circuit.cx(b, cin)
circuit.cx(a, b)
```

Listing 9: The computations that implement the quantum Full-Adder

We can also create a custom gate to implement the Full-Adder using Half-Adders. This can be achieved my re-implementing the quantum Half-Adder as shown in 5.1.3. and creating a new gate out of that circuit by using the `circuit.to_gate()` method.
Then we can append the gate to the Full-Adder circuit by the `circuit.append()` method. This method takes a quantum gate as the first parameter and a list of quantum registers for the second parameter.
It is evident that Listings 5 and 7 are equivalent, they produce the exact same output. Listing 7 has the advantage of being of a much more modular design which helps with code maintanabilty. Although it is out of the scope of this thesis we wanted to point this design choice for the sake of transparency.
In the same spirit we can create a custom gate for the Full-Adder. Let QFA be a unitary operator:

```
from qiskit import QuantumCircuit, QuantumRegister
# re-implement the half-adder circuit
half_adder = QuantumCircuit(3)
half_adder.ccx(0, 1, 2)
half_adder.ccx(1, 2)
half_adder_gate = half_adder.to_gate() # create the new gate
```

Listing 10: Creating the Half-Adder gate

```
circuit.append(half_adder_gate, (a[:] + b[:] + cin[:]))
circuit.append(half_adder_gate, (b[:] + cin[:] + o[:]))
circuit.cx(a, b) # don't forget to restore b
```

Listing 11: Implementating the Full-Adder with the Half-Adder gate

$$QFA = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix} \tag{5.2}$$

```
from qiskit import QuantumCircuit, QuantumRegister

a = QuantumRegister(1, name="a")
b = QuantumRegister(1, name="b")
cin = QuantumRegister(1, name="cin")
o = QuantumRegister(1, name="o")
circuit = QuantumCircuit(a, b, cin, o)

circuit.ccx(a, b, o)
circuit.cx(a, b)
circuit.ccx(b, cin, o)
circuit.cx(b, cin)
circuit.cx(a, b)
full_adder_gate = circuit.to_gate()
```

Listing 12: Creating the Full-Adder gate

We can now use the Full-Adder gate to implement much bigger quantum circuits without too much trouble. We want to point out that this design is not particulary efficient as it uses one "garbage" qubit [2]. For the sake of simplicity we are going to use this circuit as it is very straight-forward to understand.

5.2.3 Extending the circuit to n-qubits

The Full-Adder circuit in subsection 5.2.2 is not very useful as it can only compute the sum of 1-qubit registers. In this section we are going to use all of the previous knowledge to create a Full-Adder circuit that can compute the sum of n-qubit register inputs.
By using the gate that we devised in Listing 8, we can create the Quantum Ripple Carry Adder.
For each n-qubits iterate over the quantum registers and apply the QFA gate. We have to note that at then end of each iteration the circuit must apply a SWAP gate on qubit Cin_{i+1} and C_{out}. This must be applied at the end of each iteration because C_{out} stores the carry-out from the QFA and it must be carried down to the next carry-in qubit for the next iteration. This SWAP gate must not be applied at the end of the circuit because it will swap the last sum S_{n-1} qubit and the $Overflow$ qubit.

```
from qiskit import QuantumCircuit, QuantumRegister

n = -1 # init n to a negative integer
while n < 0:
    n = input("Input an integer (>0): ")

a = QuantumRegister(n, name="A")
b = QuantumRegister(n, name="B")
cin = QuantumRegister(n, name="Cin")
o = QuantumRegister(1, name="Cout")
circuit = QuantumCircuit(a, b, cin, o)

# create the full adder gate
full_adder = QuantumCircuit(4)
full_adder.ccx(0, 1, 3)
full_adder.cx(0, 1)
full_adder.ccx(1, 2, 3)
full_adder.cx(1, 2)
full_adder.cx(0, 1)
full_adder_gate = full_adder.to_gate()

# append the gate to complete the circuit logic
for i in range(n):
    circuit.append(full_adder_gate,  (a[i], b[i], cin[i], o[:]))
    if i+1 < n:
        circuit.swap(cin[i+1], o)
```

Listing 13: Creating the Quantum Ripple Carry Adder

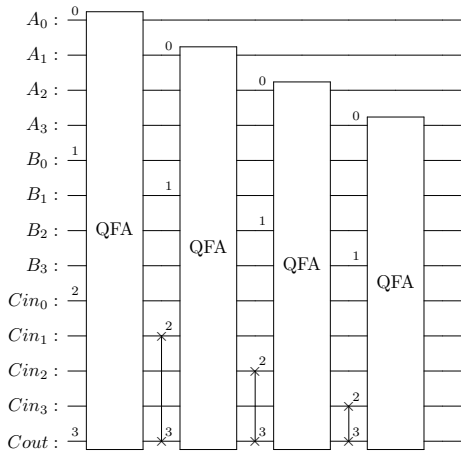


Figure 5.9: The circuit diagram of a Quantum Ripple Carry Adder with $n = 4$

5.3 The Quantum Adder-Subtractor

In this section we are going to construct using the Qiskit SDK the quantum cirucit of the Adder-Subtractor. This will be the final version of the integer addition-subtraction unit of the Quantum Arithmetic Logic Unit.

5.3.1 Analysing the diagram and logic of the Adder-Subtractor circuit

The Adder-Subtractor circuit, is a digital circuit that computes the sum or the difference of two n-bit inputs and produces an n-bit sum or difference, denoted by O, and an carry-out overflow bit $C_{overflow}$. This digital circuit is constructed using the Ripple Carry Adder circuit. As mentioned in Chapter 2, binary subtraction can be implemented by addition if we take the 2s-complement of the subtrahend and sum it with the minuend. This saves us from using two different circuits - it takes less logic gates to implement the two operations.

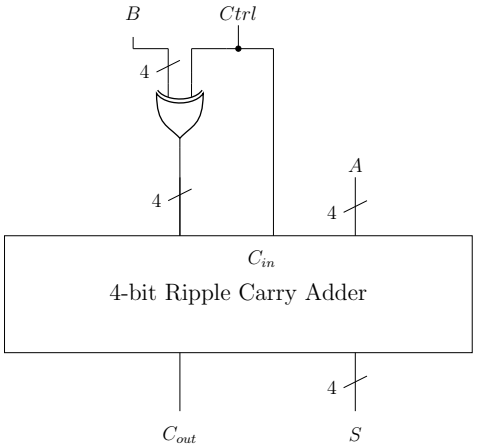


Figure 5.10: The diagram of the classical Adder-Subtractor circuit

The Adder-Subtractor works by having one extra wire, denoted as $CTRL$ in the diagram, that controls the mode of the circuit: zero for addition and one for subtraction. We will skip the first mode because we have covered that in the previous subsection. The more interesting part is the subtraction mode.

While in subtraction mode, input B and $CTRL$ bit are XOR'ed together to produce the 1s-complement of $B \rightarrow B'$. To get the 2s-complement, we need to add one to $B' + 1$. This can be done by supplying the C_{in} input of the 4-bit Ripple Carry Adder with the $Ctrl$ wire.
To implement this quantum circuit we are going to need the $QRCA$ gate that we created on the previous subsection and $n + 1$ CNOT gates, so that we can take the 1s-complement of input B and C_{in_0}. First we apply the CNOT gate, with the control qubit set to $Ctrl$ and target each qubit of register B. We also do the same for the C_{in_0} qubit. Finally we apply the $QRCA$ - the quantum Ripple Carry Adder. This will transform register C_{in} into the register Out which essential stores either the summation or difference of input quantum registers A and B.

5.3.2 Implementing the Quantum Adder-Subtractor in Python

For this implementation we are going to use the Quantum Ripple Carry Adder, discussed in the previous section. First, we are going to initialize all the apropriate registers, A and B which store the terms, $CTRL$ which stores information about the mode of the unit, C_{in} which is either initialized to all-zeros ($|0^{\otimes n}\rangle$) or all-zeros except the LSB (least significant (qu)bit) which is set by $CTRL$.

```
from qiskit import QuantumCircuit, QuantumRegister

ctrl = QuantumRegister(1, name="Ctrl")
a = QuantumRegister(n, name="A")
b = QuantumRegister(n, name="B")
cin = QuantumRegister(n, name="Cin")
cout = QuantumRegister(1, name="Cout")
circuit = QuantumCircuit(ctrl, a, b, cin, o)
```

Listing 14: Initialization of the Adder-Subtractor circuit

Next we are going to invert all qubits of register B by applying the CNOT gate to each qubit controlled by 1-qubit register $CTRL$. We also invert the LSB of register C_{in} to acquire the 2s-complement of register B. After that, we apply the QRCA gate and finally un-compute reigster B.

```
for i in range(n):
    circuit.cx(ctrl, b[i])
circuit.cx(ctrl, cin[0])
circuit.append(rca, (a[:] + b[:] + cin[:] + o[:]))
for i in reversed(range(n)):
    circuit.cx(ctrl, b[i])
```

Listing 15: Step-by-step instructions of the quantum Adder-Subtractor circuit

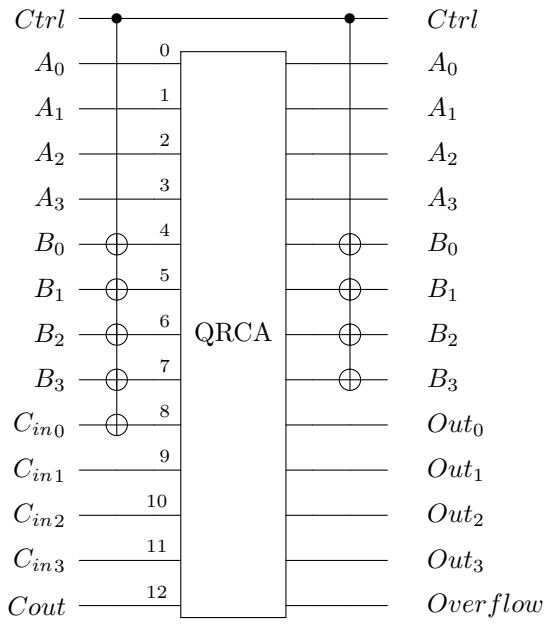


Figure 5.11: The diagram of the quantum Adder-Subtractor circuit

5.4 The Quantum Integer Multiplier

In this section we are going to implement a quantum circuit that computes the product of two 2-qubit numbers. We are going to first discuss the design of the classic circuit and then focus on the implementation of its quantum analogue using the Qiskit SDK.

5.4.1 Analysing the diagram and logic of the classical Integer Multiplier circuit

The Integer Multiplier circuit, or just the Multiplier circuit, is a digital circuit that takes two n-bit inputs and computes their arithmetic product. The circuit we are going to analyse implements the binary multiplication algorithm, by first computing the partial products and then summing them up to produce the final product [17]. Let A and B be two n-bit classic registers. For the purposes of simplicity let $n = 2$. To compute their multiplication product we first produce the *partial products* of $A \times B$. This is done digitally by applying an AND gate on every A_i and B_j pair. Then we apply a Half-Adder for the partial products. We note that the LSB of PP does not need to be added to the Half-Adders.

			B_1	B_0
		$+$	A_1	A_0
			PP_1	PP_0
	$\times$	PP_3	PP_2	
	P_3	P_2	P_1	P_0

Table 5.6: Multiplication for two 2-bit integers

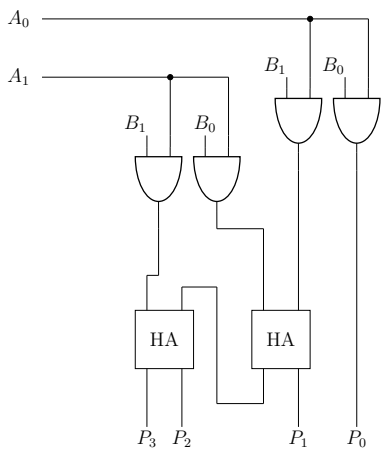


Figure 5.12: The diagram of the classical Multiplier circuit for two 2-bit inputs

5.4.2 Implementing the Quantum Multiplier in Python

The quantum implementation is an expensive circuit, qubit-wise, as it needs to compute the partial products. As we mentioned previously, the AND function can be simulated by the Toffoli gate by using one "garbage" qubit to store the conjunction of the two conrtol qubits, thus we need n^2 qubits to store the partial products of two n-qubit inputs.
As always we initialize the circuit with the two 2-qubit input registers A and B. We also include in our circuit the $2n^2$-qubit output quantum register.

```
from qiskit import QuantumCircuit, QuantumRegister

n = 2
a = QuantumRegister(n, name="A")
b = QuantumRegister(n, name="B")
out = QuantumRegister(2*n**2, name="out")

circuit = QuantumCircuit(a, b, out)
```

Listing 16: Initialization of the quantum Multiplier circuit

We proceed to compute the partial-products of $A \times B$ by applying the CNOT gate for each qubit-pair $A_i B_j$.

```
k = 0
for i in range(n):
    for j in range(n):
        circuit.cx(a[i], b[j], out[k])
        k += 1
```

Listing 17: Computing the partial-products for the quantum Multiplier circuit

Despite the expensive nature of the circuit, the overall compexity is pretty low due to the modular nature of the circuit. We can use the QFA from Section 5.2 to implement the additions of the partial products but we have to be careful as to which pair of inputs we supply the QFAs.

```
circuit.append(qfa, (out[1], out[2], out[4], out[5]))
circuit.append(qfa, (out[3], out[5], out[6], out[7]))
```

Listing 18: Summing the partial-products to compute the product of $A \times B$

To produce the products we need to compute the sum of $P_1 + P_2$. This in-turn will produce a sum and a carry-out, P_1 and C_{out_0}. The C_{out_0} will be used for the next addition with P_3. Finally, after the last addition we have computed the product of $A \times B$. The output is stored in qubits Out_0, Out_4, Out_6 and Out_7, with Out_7 be the most-significant qubit of the product.

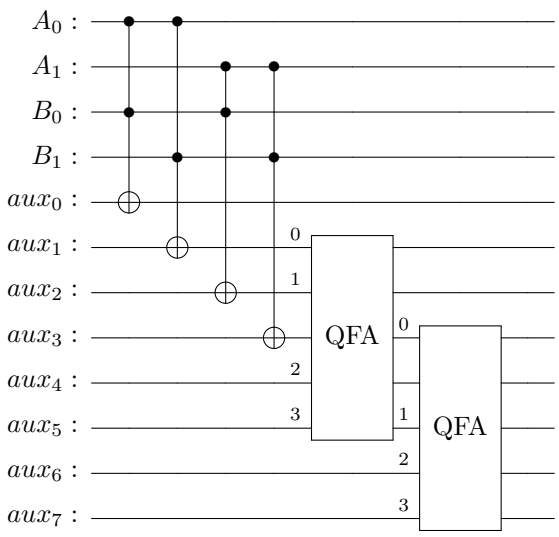


Figure 5.13: The complete quantum Multiplier unit for 2-qubit inputs

5.5 The Quantum Integer Comparator

In this section we are going to analyse a classical circuit that compares two integer numbers A and B and produces three different signals:

1. signal when A is equal with B
2. signal when A is greater than B and
3. signal when A is less than B

5.5.1 Analysing the diagram and logic of the classical Comparator circuit

In the digital logic, a comparator circuit is used to compare two binary encoded numbers. The encoding of (signed or unsigned) is particularly useful because in the case of the signed-magnitude encoding, we just need to compare the most-significant bits of the two numbers. If the numbers have different signs then we need only to check their signs, that means their most-significant bits, and in the later case, when they have the same sign, we need to check their absolute magnitude [10].

For the purposes of simplicity, we are going to analyze a *serial binary comparator* circuit.
Some classical computers have the capabilities to compute multiple comparison logics like: $>$, $<$, $\geq$, $\leq$, $=$ and $\neq$. Some other computers can only produce a sset of the those comparisons. We are going to analyze only a sset of those comparisons, specificaly we are going to analyze and construct a digital logic circuit that computes only $=$, $\neq$, $<$ and $>$. We can produce the truth table of a circuit that computes the equality of two 1-bit numbers easily.

A	B	Equals
0	0	$= (1)$
0	1	$\neq (0)$
1	0	$\neq (0)$
1	1	$= (1)$

Table 5.7: The truth table of a circuit that implements equality check between two bits

Note that the truth table is equivelant to the XNOR gate's truth table. In fact, if we had n-bit inputs, we would just need n XNOR gates and a n-input AND gate to implement a n-input equality comparator.
Next, we can implement a circuit that compares two bits and produces 0 when $<$ and 1 when $\geq$

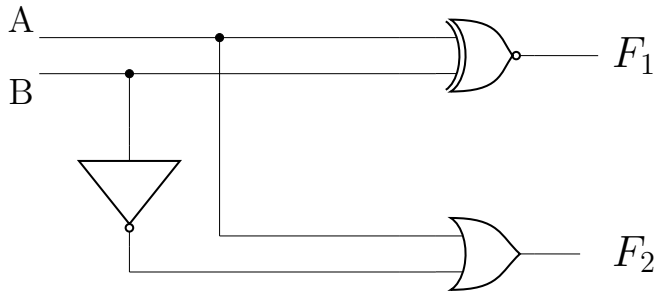
A	B	Greater Equals or Less than
0	0	$1(\geq)$
0	1	$0(<)$
1	0	$1(\geq)$
1	1	$1(\geq)$

Table 5.8: The truth table of a circuit that implements the "greater or equals than" between two bits

The boolean expression that implements this circuit is the following:

$$F = A + B' \tag{5.3}$$

Putting it all together we construct the following logic circuit.



$$\text{Where } F_1 = \begin{cases} A = B, \text{ when } 1 \\ A \neq B, \text{ when } 0 \end{cases} \quad \text{and where } F_2 = \begin{cases} A \geq B, \text{ when } 1 \\ A < B, \text{ when } 0 \end{cases} \tag{5.4}$$

A	B	F_1	F_2
0	0	1	1
0	1	0	0
1	0	0	1
1	1	1	1

Table 5.9: The truth table of the classical comparator circuit for two bits

We are going to present the diagram for a quantum comparator of integers. The design is based on the work of Nascimento, Kowanda and de Oliveira [18]. The design specifies two important constructs:

1. the unitary gate S - a quantum gate that computes the difference of two-qubits
2. the inverse of the unitary gate S, $S^\dagger$ - a quantum gate that un-computes the computed difference

We use the $S^\dagger$ gate to restore B to its initial state, because S gate computes the difference of A and B and stores it to B. Just like the quantum Half-Adder from section 5.1, we un-compute the changes to a particular qubit so it can be used for some later computation.

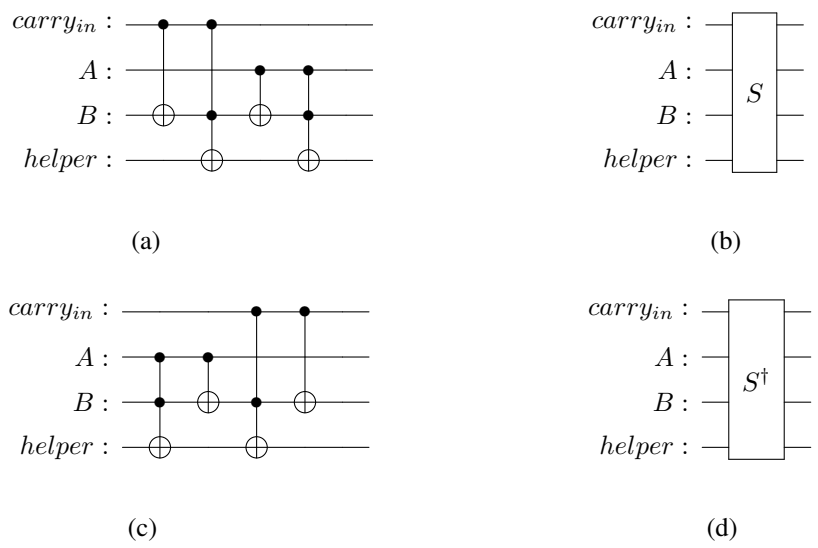


Figure 5.14: The unitary S and $S^\dagger$ gates' circuit (a, c) and gate (b, d) diagrams

To compare two variable-sized quantum registers A, B, we need to apply the S gate to each pair A_i, B_i and $ancilla_i, ancilla_{i+1}$. Note that we need $n+1$ ancilla qubits to implement this design. We iteratively apply the above.
Apply a multi-inverse-control CNOT gate to circuit. Control qubits are register B's qubits, which stores the difference of $A - B$ in 2s-complement, and target the O_0 qubit which is the least-significant qubit of a special purpose register we shall call the *status register*.
Lastly, we copy, by applying a regular CNOT gate, the most-significant qubit of register B, which stores the sign of the difference, to qubit O_1 of the status register.
Finally, we uncompute iteratively using the $S^\dagger$ gate.

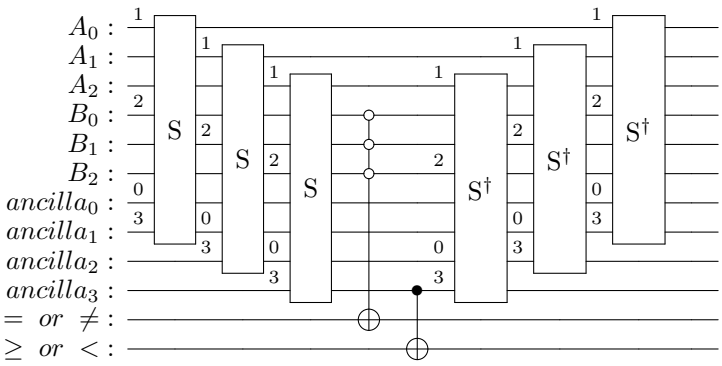


Figure 5.15: The diagram of the quantum Integer Comparator circuit

5.5.2 Implementing the NKO Comparator in Python

To implement the comparator circuit we need to construct the S and $S^\dagger$ gates.

```
s = QuantumCircuit(4, name="S")
s.cx(0, 2)
s.ccx(0, 2, 3)
s.cx(1, 2)
s.ccx(1, 2, 3)
s = s.to_gate()

sdag = QuantumCircuit(4, name="S_dag")
sdag.ccx(1, 2, 3)
sdag.cx(1, 2)
sdag.ccx(0, 2, 3)
sdag.cx(0, 2)
sdag = sdag.to_gate()
```

Listing 19: Constructing the S and $S^\dagger$ gates using Qiskit

After that we init the main circuit. The size of the registers are fixed to $n = 3$, although it can be set to any positive integer. We also include some ancilla qubits.

```
n = 3
a = QuantumRegister(n, name="A")
b = QuantumRegister(n, name="B")
anc = QuantumRegister(n+1, name="ancilla")
eq = QuantumRegister(1, name="Eq")
geq = QuantumRegister(1, name="Geq")

circuit = QuantumCircuit(a, b, anc, eq, gl, name="NKOComparator")
```

Listing 20: Instantiating the NKO Comparator Quantum circuit

Following the initialization is the main circuit logic. First apply the S gate to compute the difference of $A - B$ and store it to B.

```
# compute diff
for i in range(n):
    circuit.append(s, (anc[i], a[i], b[i], anc[i+1]))
```

Listing 21: Computing the difference of A and B on the NKO Comparator Quantum circuit

After we computed the difference, check if all the qubits of register B are zero. This can be done by using a Multi-Controlled CNOT Gate (MCX) from the `qiskit.circuit.library`. We set the MCX gate to use inverted logic for the control qubits, so if every control qubit is in the zero state the target qubit will be inverted. If every qubit of register B are in the zero state, the two registers stored the same number thus A and B are equal.

```
from qiskit.circuit.library import MCXGate
circuit.append(MCXGate(n, ctrl_state=0), (b[:] + eq[:]))
```

Listing 22: Appending a Multi-Controlled X Gate with inverted logic control qubits onto the NKO Comparator Quantum circuit to logically test $A = B$

To check if A is greater or less than B, we can check the sign qubit of the difference. This is easily done by applying a CNOT gate to most-significant qubit of the difference.

```
circuit.cx(anc[-1], geq)
```

Listing 23: Checking the MSB of the difference of A and B to logically test $A@ > B$

Finally, we un-compute by applying the $S^{\dagger}$ gate to the circuit in reverse.

```
for i in reversed(range(n)):
    circuit.append(sdag, (anc[i], a[i], b[i], anc[i+1]))
```

Listing 24: Iteratively un-computing A and B to restore the Quantum registers to their initial state

```
from qiskit import QuantumCircuit, QuantumRegister

n = 3

a = QuantumRegister(n, name="A")
b = QuantumRegister(n, name="B")
anc = QuantumRegister(n+1, name="ancilla")
eq = QuantumRegister(1, name="eq")
gl = QuantumRegister(1, name="geq")

circuit = QuantumCircuit(a, b, anc, eq, geq, name="NKOComparator")

s = QuantumCircuit(4, name="S")
s.cx(0, 2)
s.ccx(0, 2, 3)
s.cx(1, 2)
s.ccx(1, 2, 3)
s = s.to_gate()

sdag = QuantumCircuit(4, name="S_dag")
sdag.ccx(1, 2, 3)
sdag.cx(1, 2)
sdag.ccx(0, 2, 3)
sdag.cx(0, 2)
sdag = sdag.to_gate()

# compute diff
for i in range(n):
    circuit.append(s, (anc[i], a[i], b[i], anc[i+1]))

from qiskit.circuit.library import MCXGate
circuit.append(MCXGate(n, ctrl_state=0), (b[:] + eq[:])) # compute equals
circuit.cx(anc[-1], geq) # if msb < 0 then a < b else b >= a

# un-compute diff
for i in reversed(range(n)):
    circuit.append(sdag, (anc[i], a[i], b[i], anc[i+1]))
```

Listing 25: The complete code of the NKO Comparator Quantum circuit

Chapter 6

The Complete Quantum ALU

By gathering the previous Quantum circuits we can create the Quantum Arithmetic Logic Unit very easily. Before we do just that we would like to take a step back and analyse how this Unit may operate.
Just like a classical ALU, the Quantum ALU operates on some general purpose registers and on a register that stores the status of logic operations. On top of those, the Quantum ALU, may need an *operation code* or *opcode* to command it to do a specific operation. We shall analyze those opcodes further.

6.1 The Quantum ALU's Opcodes

We had introduced four different Quantum circuits in the previous chapters:

1. a Quantum Adder-Subtractor,
2. a Quantum Multiplier and
3. a Quantum Comparator

with each Quantum circuit giving us the operations of: addition, subtraction, multiplication and magnitued comparison. We can encode these four operations using a bitstring of length $n = 2$ because $log_2 4 = 2$. The mapping of each opcode with its corresponding operation and Quantum circuit can be seen at the table below.

Opcode	Operation	Quantum circuit
00	Addition	QAS (Addition mode)
01	Subtraction	QAS (Subtraction mode)
10	Multiplication	$QMUL$
11	Magnitude Comparison	$QCMP_{NKO}$

Table 6.1: The Opcode table of the Quantum ALU

6.2 The Quantum ALU's circuit

We will implement the Quantum ALU as any other Quantum circuit we have already presented. This Quantum circuit will have in total sixteen qubits. The first two-qubit Quantum register is called the *opcode* register and it stores the bitstring of which operation the Quantum ALU must complete, the two two-qubit general purpose Quantum registers A and B is where we store the binary encoded numbers we want to be operating, the eig!ht-qubit output Quantum register Out where it is used to store the output of each operation and lastly, the two-qubit status Quantum register $Status$ where each qubit corresponds to a logic flag.

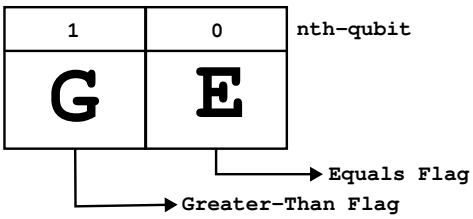


Figure 6.1: The diagram of the Status Quantum registers

```
from qiskit import QuantumCircuit, QuantumRegister

op = QuantumRegister(2, name="Opcode")
a = QuantumRegister(2, name="A")
b = QuantumRegister(2, name="B")
out = QuantumRegister(8, name="Out")
stat = QuantumRegister(2, name="Status")

qalu = QuantumCircuit(op, a, b, out, stat)
```

Listing 26: The initialization Python code for the Quantum circuit of the Quantum ALU

After the initialization of the circuit we have to append each of the Quantum circuits that implement each of the operations as custom Quantum gates.
To control when each of the operation will be selected accordingly to the opcode bitstring we are going to use the member method `control()` of the `Gate` class. This function takes numerous parameters but we are going to use only two of those: the `num_control_qubits` and the `ctrl_state` parameters. The `num_control_qubits` stores how many qubits will be used control qubits to signal the activation of the gate and the `ctrl_state` parameter annotates what is the control state of each of the control qubits. For instance, the bitstring `"101"` annotates that the least-significant and most-significant qubits will be true when in the $|1\rangle$ state and the qubit in position 1 is going to be true in the $|0\rangle$ state (inverse logic).
Using this method it is very easy to map each gate/operation to the appropriate opcode bitstring:
`ctrl_state="10"` for the multiplication gate and `ctrl_state="11"` for the comparison gate. We just have to supply the opcode register when appending.
The addition and subtraction operations where left last because they are not that straig!ht-forward to append. These operations are implemented by one gate that can change its mode by a control signal as an independent input. The other two operations needed to set the `num_ctrl_qubits=2` because they did not use a control signal as an input. This means that we can use one qubit of the opcode register as an input for the Quantum Adder-Subtractor and thus we have to set `num_ctrl_qubits=2` and `ctrl_state="0"` because according to the opcode table the most-significant qubit of the those operations is always in the $|0\rangle$ state.

```
qalu.append(addsub.control(1, ctrl_state="0"),\
    ([op[1]] + [op[0]] + a[:] + b[:] + out[:n+1]))
qalu.append(mul.control(2, ctrl_state="10"),\
    (op[:] + a[:] + b[:] + out[:]))
qalu.append(cmp.control(2, ctrl_state="11"),\
    (op[:] + a[:] + b[:] + out[:n+1] + stat[:]))
```

Listing 27: Appending the custom Quantum gates of the operations to the Quantum ALU

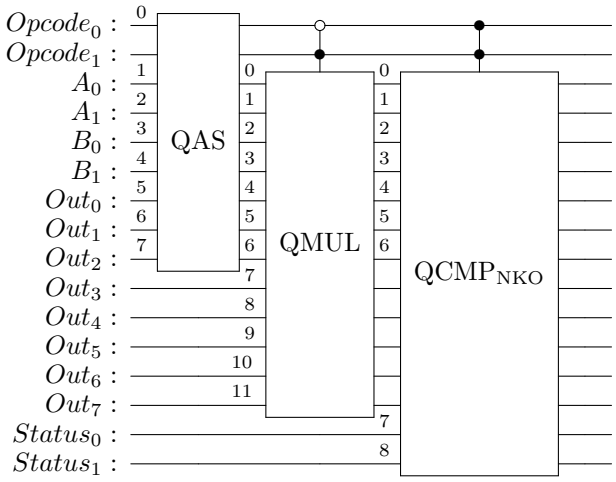


Figure 6.2: The Quantum circuit diagram of the Quantum ALU

6.3 Experimental Results on Simulators and Quantum Computers

6.3.1 Testing the Addition and Subtraction operations on the Aer Simulator and on a real Quantum Computer

We will now demonstrate all of the operations of the Quantum ALU starting with addition and subtraction. Before executing any arithmetic and logic operation we have to initialize the Opcode register with the appropriate opcode (see Table 6.1).
Therefore, to perform an addition, the Opcode register must be initialized with the $|00\rangle$ state. For this demonstration registers A and B are initialized with the states $|11\rangle = |3\rangle$ and $|10\rangle = |2\rangle$ respectively Listing (28):

```
qalu.x(a[0])
qalu.x(a[1]) # a = 11
qalu.x(b[1]) # b = 10
```

Listing 28: Initializing the Quantum registers A and B with the appropriate values

Executing the Quantum ALU circuit on an Aer simulator, the Output register arrives at the state $|00000101\rangle = |5\rangle$ which is the expected value. In Figure (6.3) we can see this even more clearly (we omitted the five leading zeros for a clearer picture of the result):
We shall now execute a subtraction operation with the same initial values for the A and B registers and submit a job to an IBM Quantum Computer so that the circuit can be executed. We want to note that the expected

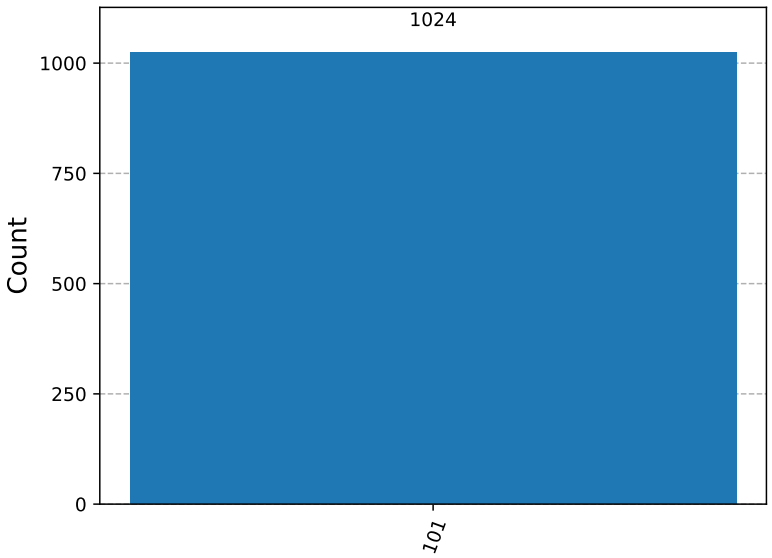


Figure 6.3: The histogram of the possible values of the Output register when the Quantum ALU performed an addition with A=3 (11) and B=2 (10) on the Aer Simulator

output is 101_2 which can be interpreted in many ways but because the Adder-Subtractor implements a design where it performs signed and unsigned-mangitude addition and subtraction and because $A \geq B$ the third qubit of the output register can be omitted thus the result becomes 01_2 or just 1 which is the expected output.
Before showing any experimental results, we would like to note that the Aer simulator and the IBM Quantum Computers execute the quantum circuits multiple times. The Aer simulator executes them 1024 times while the IBM Quantum Computers execute them four times more, or 4096 times. Thus the probability of the state of a specific quantum register is either obtained by the following equation:

$$P(o) = \lceil \frac{c}{n} 100 \rceil \tag{6.1}$$

where o is the state of the quantum register, c is the *count*, the number of times the specific state was the actual output, and n is the total number of executions which can be either 1024 (for the Aer simulations) or 4096 (for any IBM Quantum Computer).
In Figure (6.4) we can see multiple expected output values. Although we did not use any gate that puts any qubit in superposition, Quantum Computers are vulnerable to noise. Noise, just like in classical computers, can cause problems like flipping a single bit's state from example; a memory cell of a Random Access Memory module. In the case of the Quantum computer, a qubit can be flipped and give erroneous results.
We can see that the expected value (101_2) has a probability of 17.64% to be output'ed by the circuit.

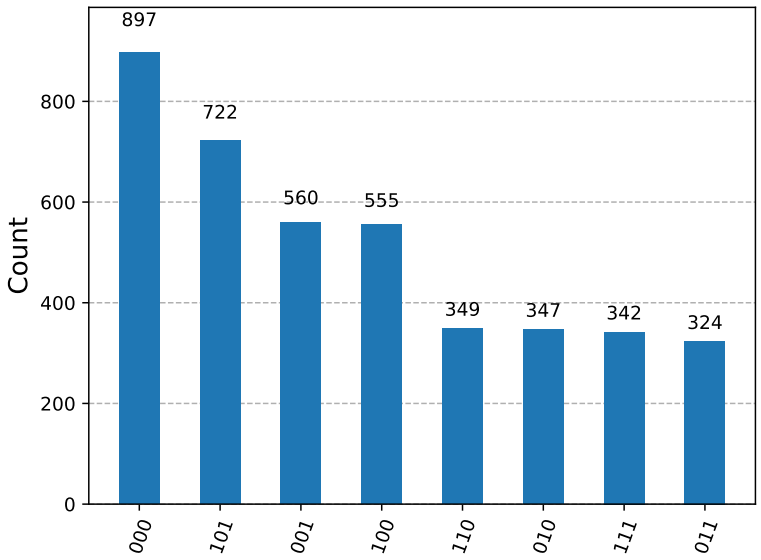


Figure 6.4: The histogram of the possible values of the Output register (with omitted the five leading zero'ed qubits) when the Quantum ALU performed an addition with A=3 (11) and B=2 (10) on the IBM Kyoto Quantum Computer

6.3.2 Testing the Multiplication operation on the Aer Simulator and on a real Quantum Computer

We shall demonstrate the multiplication operation with another pair of inputs. This time we shall initialize both A and B with the same value. We arbitrarily chose 10_2 or 2_{10}.
Before anything else, we initialize the Opcode register with the appropriate opcode ($|10\rangle$) and then initialize the A and B registers.

```
qalu.x(op[1]) # op = 10
qalu.x(a[1])
qalu.x(b[1])
```

Listing 29: The initialization of the Opcode, A and B Quantum registers to perform the multiplication operation

The result from the Aer simulator (see Figure (6.5))is what we expected. The result output only shows the qubits 0, 4, 6 and 7.
Executing the same operation on a Quantum computer yields again the same behavior. We get counts of possible outputs instead of a definite answer (see Figure (6.6)).
We note that the expected output ($0100_2 = 4_{10}$) has a 9.42% probability.

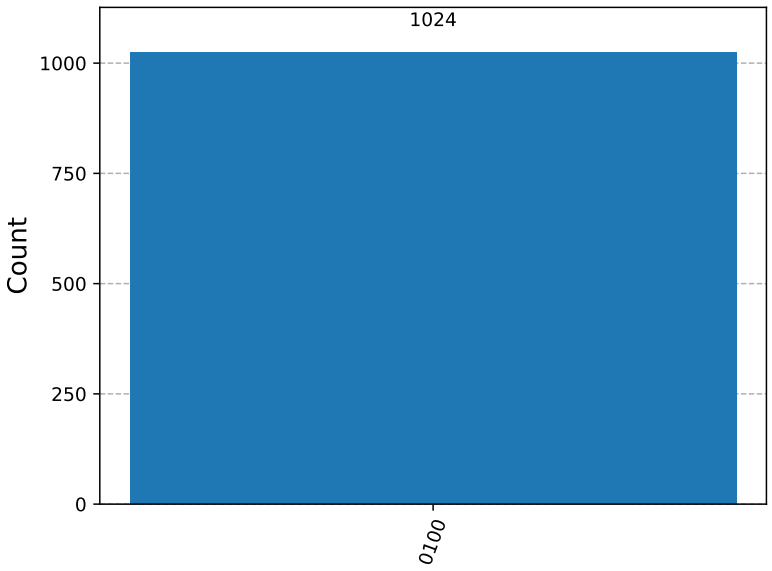


Figure 6.5: The histogram of the possible values of the Output register when the Quantum ALU performed a multiplication with A=B=2 (10) on the Aer Simulator

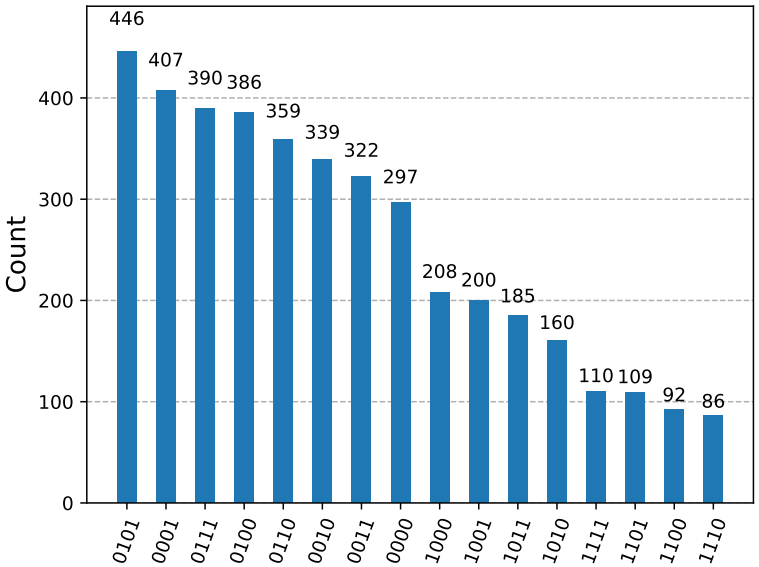
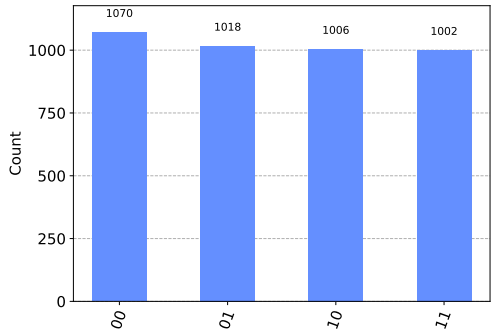


Figure 6.6: The histogram of the possible values of the Output register when the Quantum ALU performed a multiplication with A=B=2 (10) on the IBM Osaka Quantum Computer

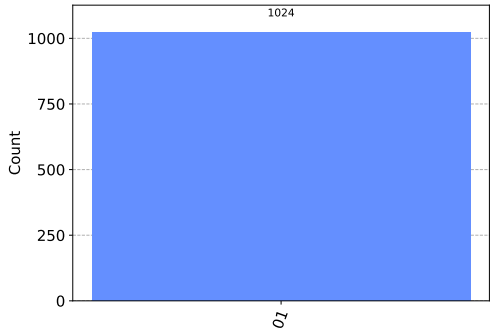
6.3.3 Testing the Comparison operation on the Aer Simulator and on a real Quantum Computer

Lastly, we executed the NKO Comparator with the two two-qubit registers inputs A and B to be again 3 and 2 respectively and we only measured the two qubits of the status Quantum register. The execution of the Quantum

circuit happened on the IBM Osaka Quantum Computer too. We will again contrast the real-hardware result with the simulation results.



(a)



(b)

Figure 6.7: The results of the comparison operation executed on the IBM Osaka (a) and on the Aer Simulator (b)

The same story applies here, where the expected value, output'ed by the Aer Simulator, 10 which means "$A \neq B$ and $A \geq B$", is not the most probable output when executed 4096 times on the Quantum computer. We again compiled a table containing the probability of each output based on how many times the output occured while executing.

Output Bit Sequence	Count	Probabilty ($\frac{\text{Count}}{4096}$)
00	1070	26.123%
01	1018	24.854%
10	1006	24.561%
11	1002	24.463%

Table 6.2: The result of probabilities of each output value of the execution of the comparison operation on the IBM Osaka

We can see that each output has relatively close probability of output meaning that the Quantum computer does not give us a reliable output.

Chapter 7

Conclusions

We have shown that it is possible to construct a very basic Quantum Arithmetic Logic Unit but due to the considerable compexity of the proposed designs of the Quantum algorithms for each operation, the complete Unit is very susceptible to noise. We shall go over the details of each design.

The Adder-Subtractor Quantum circuit was based on the proposed circuit by Richard P. Feynman [2] but the design itself needs a n-qubit output register to hold the computed output. There are other designs of Quantum adder circuits that use the *Quantum Fourier Transform* or just *QFT* to encode one of the inputs to the *Fourier basis* transforming it from the initial state p to ϕp and then by using the qubits from the other input perform conditional R_z rotations on $|p\rangle$. This will compute $|p+q\rangle$ assuming the other input is q. Such a Quantum algorithm and circuit was designed by Draper [19] and it is used by Qiskit. Lastly, the Draper adder uses only $2n$ qubit instead of $3n+2$ qubits of this work's Adder-Subtractor.

The Multiplier Quantum circuit is probably the number one contender for bringing unnecessary complexity to the complete system. This design uses $2n^2+2n$ qubits which drives the complexity of the complete system by n^2. For a future work, a complete overhaul and change of the circuit is completely needed to drive down the complexity of the circuit, which in turn can drive down the compute time needed to run on a Quantum compute, something that did happen and pushed us to only execute parts of the complete design on real hardware instead of as a whole. A proposed alternative algorithm may be the Qiskit's own Multiplier Quantum circuit like the Ruiz-Perez et al. [20] design (which is also called the RGQFT Multiplier). In contrast, the RGQFT Multiplier uses only $3n$ qubits in total.

Moreover, the NKO Comparator was a design implemented from previous work [18]. The design is robust, concise and future work on the design may include adding more calculations of other statuses like: detecting and overflow from addition/subtraction.

Lastly, for future work, the Quantum Arithmetic Logic Unit can be extended to include the operation of integer division. This was not implemented in this thesis due to the inherent complexity of the integer division algorithm[21], which requires a circuit that loops and compares values until a terminal state is reached.

Chapter 8

Bibliography

[1] R. P. Feynman, "Simulating physics with computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, Jun 1982.

[2] R. P. Feynman, "Quantum mechanical computers," *Optics News*, vol. 10, no. 6, pp. 60–61, 1984.

[3] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, p. 1484–1509, Oct. 1997.

[4] B. Josephson, "Possible new effects in superconductive tunnelling," *Physics Letters*, vol. 1, no. 7, pp. 251–253, 1962.

[5] IBM, "IBM's quantum platform homepage." `https://quantum.ibm.com` [Accessed: (2024-05-03)].

[6] T. Lanting, A. J. Przybysz, A. Y. Smirnov, F. M. Spedalieri, M. H. Amin, A. J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J. P. Hilton, E. Hoskinson, M. W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, S. Uchaikin, A. B. Wilson, and G. Rose, "Entanglement in a quantum annealing processor," *Phys. Rev. X*, vol. 4, p. 021041, May 2014.

[7] C. E. Shannon, "A symbolic analysis of relay and switching circuits," Master's thesis, Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, 1937.

[8] B. George, "The mathematical analysis of logic being an essay towards a calculus of deductive reasoning," 1847.

[9] E. V. Huntington, "Sets of independent postulates for the algebra of logic," *Transactions of the American Mathematical Society*, vol. 5, no. 3, pp. 288–309, 1904.

[10] P. I. Giannakopoulos, *Logic Circuits*. Kallipos, 2015.

[11] P. A. M. Dirac, "A new notation for quantum mechanics," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 35, no. 3, p. 416–418, 1939.

[12] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 183–191, 1961.

[13] C. H. Bennett, "The thermodynamics of computation—a review," *International Journal of Theoretical Physics*, vol. 21, pp. 905–940, Dec 1982.

[14] I. K. Marmorkos, *Principles of Quantum Computing*. Kallipos, 2024.

[15] IBM, "Qiskit's github repository." `https://github.com/Qiskit/qiskit` [Accessed: 2024-05-03].

[16] IBM, "Qiskit's pip search index." `https://pypi.org/project/qiskit` [Accessed: 2024-05-03].

[17] M. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog*. Pearson, 2017.

[18] A. L. Nascimento, L. A. B. Kowanda, and W. R. Oliveira, "Uma unidade lógica e aritmética reversível - a reversible arithmetic and logic unit," *WECIQ*, 2006.

[19] T. G. Draper, "Addition on a quantum computer," 2000.

[20] L. Ruiz-Perez and J. C. Garcia-Escartin, "Quantum arithmetic with the quantum fourier transform," *Quantum Information Processing*, vol. 16, Apr. 2017.

[21] H. Thapliyal, E. MuÑoz-Coreas, T. S. S. Varun, and T. S. Humble, "Quantum circuit designs of integer division optimizing t-count and t-depth," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 2, pp. 1045–1056, 2021.