

## METHODS

# Project and Implementation of a Quantum Logic Gate Emulator on FPGA Using a Model-Based Design Approach

**AGOSTINO GIORGIO** 

Department of Electrical and Information Engineering (DEI), Polytechnic University of Bari, 70125 Bari, Italy

e-mail: agostino.giorgio@poliba.it

**ABSTRACT** Quantum computing emulators are widely used for testing quantum algorithms ideally before executing them on real quantum processors. Therefore, researchers are very active in developing emulators primarily on FPGAs. This project is complex and requires a long time and a specific expertise if carried out following the low-level design approach using a Hardware Description Language (HDL). This paper describes, step by step, the process of quick designing a quantum gate emulator using a much simpler high-level design approach. It begins with the development of a quantum gate simulator in the MATLAB® environment, and subsequently translates it into an HDL design for FPGA implementation by using the MATLAB's HDL Coder toolbox. However, while code translation may seem straightforward using MATLAB toolboxes, it becomes a non-trivial task when transitioning from a quantum computing simulator to an HDL-based quantum computing emulator. Thus, it was necessary to conduct an in-depth study to implement a quantum computing simulator in MATLAB, enabling an error-free translation of HDL code. The developed method enables the designer to leverage the highly useful and straightforward model-based design approach offered by MATLAB, rather than directly the more complex HDL approach, returning a highly optimized HDL code for configuring the FPGA as a quantum computing emulator. This topic makes the design of quantum emulators on FPGA quick, reliable, optimized and without the need for specialized hardware design skills. Additionally, two implementation examples have been described using Altera/Intel FPGA on development boards DE1\_SoC and DE5a-NET DDR4 provided by Terasic Inc.

**INDEX TERMS** Computer applications, FPGA, quantum computing, MATLAB.


## I. INTRODUCTION

Today, we are at the very beginning of a quantum revolution [1]. Companies such as IBM [2], Rigetti [3], D-Wave [4] and others, are investing in building physical quantum computers and exposing programmable interfaces to users around the world through cloud solutions.

In a recent paper [5], it is said that, thanks to the 53-qubit programmable quantum processor Sycamore, it was possible to complete a computation in 200 seconds that would take approximately 10,000 years for the world's most powerful supercomputer to solve. This is called the quantum

supremacy [6], [7]. The qubit is the quantum information unit and stands for “quantum bit”.

However, it is important to highlight that quantum computers are not meant to entirely replace classical ones. Their primary goal is to address specific and computationally demanding problems, such as the integer factorization, simulating quantum systems [8], and other applications that have been under study in recent years [9]. In most applications these two types of computation, classical and quantum, are complementary because the quantum computer collaborates with the classical computer. In fact, the classical computer prepares the computation to be conducted by the quantum computer and then retrieves the result from it for interpretation.

The associate editor coordinating the review of this manuscript and approving it for publication was Khmaies Ouahada .

Anyway, the enormous potential and power of quantum computing drives researchers around the world to develop new algorithms using to the massive parallelism in quantum computation.

At the same time, real quantum machines still have a long way to go before they become truly universal, stable, and scalable. Spontaneous decoherence, state preparation and measurement faults, or the engineering challenge of creating entangled states, are serious problems to solve in the real world.

Even if, as it was suggested by Richard Feynman and stated by the Quantum Strong Church–Turing Thesis, only quantum machines are capable of efficient emulation of quantum circuits [1] or, in other words, there is no physical way to achieve the quantum speedup on classical, sequential machines, the role of quantum emulation is crucial for debug purposes.

For this reason, quantum emulators are still being designed and built to help on different frontiers of the quantum revolution.

Despite the presence of quantum computing simulators, including the recent MATLAB add-on “Quantum computing toolbox” [10], there are significant differences between a simulator and an emulator using dedicated hardware, such as Field-Programmable Gate Arrays (FPGAs) that can be outlined as follows.

Typically, quantum simulators work at a higher level of abstraction, using classical computing resources to simulate the behaviour of a quantum system. They model quantum gates, circuits, and algorithms using classical bits and algorithms to simulate the probabilistic nature of quantum systems. Differently, an emulator implemented on an FPGA aims to replicate the quantum behaviour more closely by using the parallelism and reconfigurability of the FPGA hardware. It tries to mimic quantum operations at a lower level, often involving the direct representation of qubits and quantum gates in hardware.

In fact, the primary focus of a quantum computer emulator is to ensure that every operation is naturally parallelized, in a way that closely emulates the quantum nature of the computation, and this aim is accomplished thanks to the reconfigurable hardware of FPGAs providing inherent parallelism, enabling simultaneous execution of multiple quantum operations. Simulators are generally slower than emulators, as they rely on classical computation. They are limited by the processing power of classical processors (CPUs) and are constrained by the need to simulate quantum parallelism using classical parallelism.

Moreover, the simulation of any quantum algorithm on a standard computer will often require exponentially more time than it would on a quantum machine. While it is impossible to bypass that need for resources when emulating quantum circuits, by using FPGAs can shift the weight from time to hardware complexity.

In addition, the reconfigurable nature of FPGAs allows to perfect resource usage for more customization of quantum operations at the hardware level and supplies a balance

between flexibility and performance, making FPGA emulators more efficient in terms of both space and power consumption compared to classical simulators. This can be helpful for applications where fine-tuning of quantum gates is crucial.

The FPGA hardware emulates the behaviour of a real quantum system, enabling the efficient execution of quantum algorithms while maintaining their natural time complexity. This is essential for evaluating the performance and practicality of such algorithms in a simulated quantum computing context.

Finally, a quantum computing emulator does not have to consider the typical undesired phenomena of the real quantum world. Its sole aim is to provide users with the convenience of running a universal emulator that yields results akin to those of a quantum machine, without needing to address error correction and other undesired side effects. Therefore, a quantum emulator implements perfect qubits.

Thanks to emulators, users are enabled to design and run on quantum algorithms without the need to own an actual quantum computer.

While it is possible to implement quantum computing emulators on Graphical Processing Units (GPU), which are well-suited for intensive calculations, there are, however, some limitations encountered.

In fact, although GPUs are efficient in handling parallel operations, they may encounter constraints in optimizing for specific quantum computing algorithms. On the other hand, FPGAs offer greater flexibility in optimization for specific algorithms.

Moreover, GPUs may have higher latency compared to FPGAs in certain computational scenarios, especially when dealing with very specific data streams.

This is why the use of FPGAs is widespread in the scientific community, and it is the same hardware that has been chosen for use in this work.

In fact, in scientific literature, there are several works and papers related to the design of quantum computing emulators implemented on FPGA, but the approach followed, i.e., the hardware design approach by using the Hardware Description Languages (HDL), is always very complex and requires uncommon skills.

The purpose of this paper is to outline a dependable, high-level design methodology for a quantum computing emulator, which is then implemented on an FPGA. This process begins with designing a quantum logic gate simulator in the MATLAB environment and involves translating it into an HDL design for FPGA implementation.

The described approach offers significant advantages compared to a low-level hardware design of FPGAs configuration.

In fact, the MATLAB environment allows for a model-based design method, which is much simpler and widely used in the scientific community compared to the hardware-based approach required when designing directly in any hardware description language. This approach also

proves to be much more effective for project debugging and much faster in reaching the final implementation.

Additionally, the MATLAB code translator toolboxes enable the generation of highly optimized code tailored to the specific target hardware. Achieving this level of optimization is much more challenging when working directly in HDL and requires uncommon design skills. With the method devised and described in this paper, highly optimized HDL code can be obtained more easily.

Finally, the proposed approach is “cross-factory” because it applies to FPGAs from different manufacturers.

The procedure to translate a project developed in MATLAB into an HDL design is theoretically simple when executed with the “HDL Coder” toolbox, which is part of the MATLAB software. However, serious difficulties arise if the aim is to obtain the HDL design of quantum logic gates starting from their MATLAB models, and it becomes a real challenge if a fixed-point number representation is used. This representation allows for a dramatic reduction in the need for FPGA resources, while MATLAB functions work with floating-point double-precision arithmetic.

If a designer attempts to automatically convert the models of quantum logic gates from MATLAB code to HDL using the “HDL Coder” and the “Fixed-point designer” MATLAB toolboxes, the process fails.

This happens for several reasons.

Firstly, MATLAB functions are employed in the implementation of quantum gates, many of which are unsuitable for code translation. Not all MATLAB functions, particularly trigonometric and exponential functions widely used in simulating quantum gates, support automatic code translation. For each function, it’s possible to check the documentation web page’s “Extended capabilities” section to ascertain whether it supports automatic code translation or not.

If a function is not supported by the conversion tool it is necessary to replace it with a Look-up Table (LUT) implementation or a CORDIC (COordinate Rotation DIgital Computer) implementation.

Secondly, there are functions introduced by the Fixed-Point Designer tool that are unsupported for code translation. Therefore, if the ultimate goal is to translate the code into HDL, it is not advisable to use the Fixed-Point Designer for the automatic conversion from floating-point to fixed-point arithmetic. Instead, it is necessary to write the code of the quantum simulator directly in fixed-point arithmetic. Additionally, subscripting and casting procedures must be employed during coding to optimize resource consumption in hardware (FPGA) implementations. These procedures are unnecessary when designing only a simulator without the intention of translating it into HDL code.

Thirdly, not all MATLAB functions used to simulate quantum gates support fixed-point arithmetic, thus necessitating proper replacement.

Fourthly, for the MATLAB project to be suitable for the automatic generation of HDL code, it must be structured meticulously, distinguishing the function that will undergo

translation—the actual algorithm—and a MATLAB script in which the function will be called. This listing will constitute the testbench file.

For these reasons, simply writing a quantum computing simulator in MATLAB without adhering to proper rules and procedures is insufficient for generating optimal and error-free HDL code translation.

Therefore, the procedure to translation from a quantum computing simulator project developed in MATLAB to a quantum computing emulator developed in HDL is anything but straightforward and obvious. It requires a deep and highly detailed study, such as the one conducted and described in this paper, to make the procedure smooth and error-free.

Currently, there is no documentation on this topic.

The author specifically studied and tuned the procedure for the development of MATLAB code for quantum computing simulation that is oriented towards rapid, optimized, and error-free translation into HDL code.

Therefore, the importance and novelty of this paper are twofold. Firstly, it proposes a high-level quantum emulator design procedure based on the model-based design approach characteristic of MATLAB, a concept not previously presented in existing literature. Secondly, it defines rules for designing a quantum computing simulator in MATLAB, specifically tailored for code translation into HDL.

In this paper, the author describes the complete design flow and the step-by-step procedure for creating a quantum computing simulator in MATLAB, specifically oriented towards code translation. It culminates in the implementation of a highly optimized quantum computing emulator on an FPGA. Importantly, this process does not require in-depth knowledge of HDL or FPGA design method.

The procedure has been applied to design and implement a single qubit and a two-qubit emulator on Altera/Intel FPGAs.

Therefore, in Section II, the theoretical background regarding qubits and quantum logic gates has been described. In Section III, references found in the literature related to the design and implementation of quantum computing emulators are reviewed, highlighting that no paper – at the best of the author knowledge – deals with the proposed approach. Section IV provides a detailed explanation of the new approach. Section V presents the hardware implementation of single qubit and two-qubit emulators. In Section VI the obtained results are discussed. Finally, in Section VII, there are conclusions and final remarks.

## II. THEORETICAL BACKGROUND ABOUT QUBIT AND QUANTUM LOGIC GATES

Classical information is founded on the concept of bits - logical, dimensionless values - that can be either 0 or 1. These values correspond to two physical voltage values, such as 0 V and 5 V, and are manipulated by logic gates.

Logic gates are circuits that physically implement logical operations such as AND, OR, and NOT (or INVERTER).

Quantum computers are based on the qubits that behave according to the quantum theory and are processed by quantum logic gates.

Qubits are “prepared” in a linear combination of two distinct states that correspond to the two classical bits 0 and 1. In general, a quantum system simultaneously exists in a superposition of all its possible states, according to the principle of superposition of states. This results in having an immense amount of information (theoretically infinite) even about a single quantum information carrier compared to classical bits.

Superposition is not the only property of quantum mechanics used in quantum computation. For example, considering the measurement principle of quantum mechanics, until a measurement is performed on the system, the system itself stays in an “indefinite” state; at the time of measurement, the system’s state collapses (reduces) into one of the possible base states defined for the system, with a certain probability. Therefore, the measurement is an irreversible operation on the quantum system and the quantum computation perspective is probabilistic, in contrast to the deterministic approach of classical computation. The quantum computer can be seen as a processing unit that can simultaneously process many inputs and producing probabilistic outputs.

Due to measurement, the information reverts back to a classical nature. This occurs due to another characteristic of the quantum world: quantum interference, which increases the probability of obtaining the correct answer to the problem being solved.

A relevant feature for the implementation of quantum algorithms is the “Entanglement.” Entangled quantum elements are “intertwined”, intrinsically connected.

The elements of the quantum system are described by a single wave function, not separable into distinct functions.

The greater computational power of quantum computers arises from quantum parallelism, which allows a quantum computer to execute the same algorithm on all logical states simultaneously, using the superposition of states. This is a powerful property that no classical computer can achieve.

Consequently, calculations can be performed simultaneously on all information - i.e., on all  $2^n$  quantum states superimposed - stored in a  $n$ -qubit quantum register.

The result is that parallel computation can be achieved using a single quantum hardware module. This means that a quantum computer can, in a single computation step, perform the same mathematical operation on  $2^n$  different input numbers encoded in a coherent superposition of  $n$  qubits. To achieve the same work, any classical computer must repeat the same computation  $2^n$  times, or it requires  $2^n$  different processors working in parallel.

In order to understand the process of designing a quantum computing simulator in the MATLAB environment and subsequently translating it into a quantum computing emulator for implementation on an FPGA, it is essential to grasp the operations carried out by quantum logic gates. This entails understanding the fundamental notation used and the

algebraic operations executed on qubits, which constitute the foundation of the functionality of quantum logic gates.

### A. QUBITS, FUNDAMENTAL NOTATION AND OPERATIONS

In quantum physics, it is customary to use Dirac notation, which involves the use of symbols known as “bra” and “ket” represented as  $\langle |$  and  $| \rangle$  respectively.

The symbol  $| \rangle$  is known as a ket, it is a column vector, and it stands for a quantum state vector.

The symbol  $\langle |$  is called a bra, and it stands for a quantum state vector to which the complex conjugation operation is applied.

Subsequently, Dirac notation will be extensively used, starting from the algebraic definition of a quantum bit.

From a mathematical perspective, being a qubit a linear combination of 0 and 1 states, is expressed as in (1):

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (1)$$

where coefficients  $\alpha$  and  $\beta \in \mathbb{C}$  being  $\mathbb{C}$  the field of complex numbers.

The vectors  $|0\rangle$  and  $|1\rangle$  form an orthonormal basis, also known as the standard computational basis, as in (2) and (3), respectively:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (2)$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (3)$$

The complex coefficients  $\alpha$  and  $\beta$  are known as probability amplitudes, and their squared magnitudes are the probability of measuring the states  $|0\rangle$  and  $|1\rangle$ , respectively. This leads to the normalization condition in (4):

$$|\alpha|^2 + |\beta|^2 = 1 \quad (4)$$

This condition arises because the sum of probabilities of all possible measurement outcomes must be equal to 1. Therefore, a qubit is a unit vector, i.e., its norm is equal to 1, in a complex two-dimensional vector space  $\mathbb{C}^2$ .

A single qubit can be graphically represented using the Bloch sphere, a unit-radius sphere where points on its surface correspond uniquely to qubit states, as shown in Fig. 1.

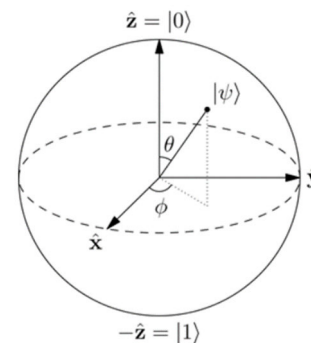


FIGURE 1. Bloch sphere.

The “north pole” corresponds to the state  $|1\rangle$ , the “south pole” to  $|0\rangle$ , and other points are superpositions of the states  $|0\rangle$  and  $|1\rangle$ .

Typically, a point on the surface of a sphere in three-dimensional space  $\mathbb{R}^3$  is represented in spherical coordinates  $(\rho, \varphi, \theta)$ , where  $\rho$  is the distance from the point on the sphere to the origin of the reference system (always  $\rho = 1$ ),  $\theta$  denotes the angle formed by the point on the sphere in  $\mathbb{R}^3$  with the positive z-axis, and  $\varphi$  is the angle formed by the projection of the vector onto the  $(x, y)$  plane with the positive x-axis.

In the case of a qubit, it is proper to represent the ket using the standard computational basis and highlighting the relationship between it and the polar coordinates. This results in (5):

$$|\psi\rangle = e^{i\gamma} \left( \cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\varphi} \sin\left(\frac{\theta}{2}\right) |1\rangle \right) \quad (5)$$

Holding the condition:

$$0 \leq \theta \leq \pi \text{ and } 0 \leq \varphi \leq 2\pi$$

The factor  $e^{i\gamma}$  is known as the global phase.

The inner product is a binary operation denoted by the symbol  $(\cdot, \cdot)$ , which takes values in the field of the vector space itself:  $(\cdot, \cdot): V \times V \rightarrow \mathbb{K}$ .

The field  $\mathbb{K}$  can be defined in several ways. For the purposes of quantum mechanics calculations, the field of complex numbers  $\mathbb{C}$  is considered. Given two vectors  $v$  and  $u$ , being respectively:

$$\begin{aligned} v &= (a_1, a_2, \dots, a_n) \\ u &= (b_1, b_2, \dots, b_n) \end{aligned}$$

the inner product is defined in (6):

$$(v, u) = \sum a_i^* b_i \text{ for } i = 1 \text{ to } n \quad (6)$$

Two vectors are said to be orthogonal if and only if their inner product is zero.

The norm of a vector is defined and denoted by the symbol  $\|\cdot\|$  as in (7):

$$\|v\| = \sqrt{(v, v)} \quad (7)$$

A unit vector is a vector with a norm equal to 1.

An orthonormal basis is a basis in which its vectors are unit vectors and pairwise orthogonal.

The inner product can be rewritten as  $\langle | \rangle$ .

The Hilbert space is a mathematical entity that plays a fundamental role in the formalization of quantum mechanics theory: the first postulate of the theory states that to every quantum system, one associates a Hilbert space, typically denoted as  $\mathcal{H}$ .

A Hilbert space is a vector space equipped with an inner product  $\langle \cdot | \cdot \rangle$  and is complete with respect to the metric induced by the norm  $\|\cdot\|$ .

A metric space is said to be complete if all Cauchy sequences of vectors in the vector space converge to an element in the space; for vector spaces of finite dimensions, this condition is always satisfied.

In quantum computation, the vector spaces dealt with are always of finite dimensions; therefore, in this context, the terms ‘‘Hilbert space’’ and ‘‘vector space’’ are equivalent.

A quantum system with multiple interacting qubits forms a quantum register and the tensor product operator supplies a rigorous mathematical description of the quantum register.

Given two vector spaces  $V$  and  $W$  of dimensions  $n$  and  $m$ , respectively, the tensor product space  $V \otimes W$  is the set of vectors in (8):

$$v \otimes w = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \otimes \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} v_1 \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \\ \vdots \\ v_n \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \end{pmatrix} \quad (8)$$

where  $v \in V, w \in W$

The dimension of the tensor product space will be the product of the dimensions of the spaces that compose it.

The properties of the tensor product between vectors, which will be useful later, are highlighted.

For every  $|v\rangle \in V$  and  $|w\rangle \in W$  with  $z$  as a scalar:

$$z(|v\rangle \otimes |w\rangle) = (z|v\rangle) \otimes |w\rangle = |v\rangle \otimes (z|w\rangle) \quad (9)$$

For every  $|v_1\rangle, |v_2\rangle \in V$  and  $|w\rangle \in W$ :

$$(|v_1\rangle + |v_2\rangle) \otimes |w\rangle = |v_1\rangle \otimes |w\rangle + |v_2\rangle \otimes |w\rangle \quad (10)$$

For every  $|v\rangle \in V$  and  $|w_1\rangle, |w_2\rangle \in W$ :

$$|v\rangle \otimes (|w_1\rangle + |w_2\rangle) = |v\rangle \otimes |w_1\rangle + |v\rangle \otimes |w_2\rangle \quad (11)$$

The tensor product is bilinear, meaning it is linear in each argument:

$$|v\rangle \otimes (a|w_1\rangle + b|w_2\rangle) = a|v\rangle \otimes |w_1\rangle + b|v\rangle \otimes |w_2\rangle \quad (12)$$

$$(c|v_1\rangle + d|v_2\rangle) \otimes |w\rangle = c|v_1\rangle \otimes |w\rangle + d|v_2\rangle \otimes |w\rangle \quad (13)$$

where  $a, b, c, d \in \mathbb{C}$  (complex numbers).

A quantum register is a space of quantum states generated by  $n$  qubits.

From a mathematical perspective, a quantum register is an element of the Hilbert space  $\mathbb{C}^{2^n}$ , with a basis that consists of the tensor product of  $2^n$  qubits in (14):

$$|i_1\rangle \otimes |i_2\rangle \otimes \dots \otimes |i_n\rangle \quad (14)$$

The terms  $|i_j\rangle$  represent the bases of the Hilbert space associated with each qubit. The equivalent notation is also used for the basis of the register  $|i_1 i_2 \dots i_n\rangle$ .

Considering, for example, a register composed of 2 qubits, the system is represented by the tensor product of two kets  $|i_1\rangle$  and  $|i_2\rangle$ ; therefore, the state of the quantum register is as in (15):

$$|\psi\rangle = \alpha_1|00\rangle + \alpha_2|01\rangle + \alpha_3|10\rangle + \alpha_4|11\rangle \quad (15)$$



Like the case of a single qubit, the normalization condition also holds for the quantum register as in (16):

$$\sum |\alpha_k|^2 = 1 \tag{16}$$

where the sum is over all possible combinations of  $\alpha$  values, ranging  $k$  from 1 to  $2^n$ .

In the case of measurement on a single qubit, the state of the register will be changed to a new state conditioned on the measurement result itself, as in (17):

$$|\psi'\rangle = \frac{\alpha_1 |00\rangle + \alpha_2 |01\rangle}{\sqrt{|\alpha_1|^2 + |\alpha_2|^2}} \tag{17}$$

Equation (17) shows the state in which the system occurs when a measurement is performed on the first qubit, resulting in an outcome of 0.

In the case where the bit 1 is measured, the state of the register results as in (18):

$$|\psi''\rangle = \frac{\alpha_3 |10\rangle + \alpha_4 |11\rangle}{\sqrt{|\alpha_3|^2 + |\alpha_4|^2}} \tag{18}$$

In both cases, the vector is normalized to have unit norm.

It is not always possible to write the state of a quantum register as the tensor product of the individual qubits taken separately. These quantum states are called entangled states. In other words, a state is said to be ‘‘Entangled’’ if and only if it is inseparable, meaning that given  $|\psi\rangle$ , the ket of a quantum system consisting of  $n$  qubits cannot be expressed as in (19):

$$|\psi\rangle = \alpha_1 v_1 \otimes \alpha_2 v_2 \otimes \dots \otimes \alpha_n v_n \tag{19}$$

With  $\alpha_i \in \mathbb{C}$  standing for the probability amplitudes of the  $i$ -th qubit and  $v_i$  being one of the two computational basis states associated with the same qubit.

In an entangled state, the measurement of one qubit affects the probability of measuring the others.

For example, consider an entangled state of a 2-qubit register in (20):

$$\frac{1}{\sqrt{2}} (|00\rangle + |11\rangle) \tag{20}$$

By measuring the state of the first qubit, which has a 50% chance of collapsing into either state 0 or state 1, is obtained also the information about the state of the second qubit: it is the same value as the first. The measurement of the system not only changes the behaviour of the first element but also influences the entire system, thereby defining the state of the second qubit as well.

### B. QUANTUM LOGIC GATES

A functionally complete set of logic gates in classical computing supplies all the necessary elements for any digital circuit design. Similarly, in quantum computing, defining elementary quantum circuits, known as quantum logic gates, and finding a subset of them that forms a functionally complete system, allows for the implementation of any quantum operation.

Quantum logic gates are transformations that act on a Hilbert space of dimension  $2^n$ , where  $n$  is the length of the input vector or, equivalently, as  $2^n \times 2^n$  matrices with values in  $\mathbb{C}$ .

Operations on qubits can be classified as operations on single qubits or multiple qubits.

For the latter, they can be reduced to a sequence of operations that act on at most two qubits within the considered vector.

Therefore, in this section are described all elementary quantum logic gates along with their mathematical formalism and operations. This description is necessary to design the quantum simulator in MATLAB environment and then, the quantum emulator in HDL.

Let  $V$  and  $W$  be two vector spaces over the field  $\mathbb{K}$ ,  $x \in \mathbb{K}$  and let  $A : V \rightarrow W$  be an operator.

$A$  is called a linear operator if and only if it satisfies the two properties in (21) and (22):

$$A(v + u) = A(v) + A(u) \tag{21}$$

$$A(xv) = xA(v) \tag{22}$$

It is possible to define a relationship that links a linear operator on vector spaces to matrices.

In fact, let  $V$  and  $W$  be two vector spaces with bases:

$$\mathbb{B}_1 = \{v_1, v_2, \dots, v_n\} \text{ and}$$

$$\mathbb{B}_2 = \{w_1, w_2, \dots, w_m\}$$

respectively and let  $f : V \rightarrow W$  be a linear operator. There exists a unique matrix  $A \in \mathbb{K}^{m,n}$  such that:

$$A(v_j) = \sum_{i=1}^m a_{ij} w_i \tag{23}$$

for  $j = 1, 2, \dots, n$

being  $a_{ij}$  the elements of the matrix  $A$ .

If the bases of the vector spaces changes, the components of the matrix will change accordingly.

The matrix defined in this way is called the associated matrix or the representation matrix of the linear operator.

Therefore, a suitable representation of linear operators is through matrices. This property allows for the use of matrix algebra in describing elementary quantum operators, as these operators are linear.

The application of a quantum operator on a system can be expressed as the product of the representation matrix of the operator  $A$  and the associated ket of the system  $|v\rangle$ :

$$|v'\rangle = A |v\rangle \tag{24}$$

The matrices associated with operations on quantum systems have specific properties that result from the requirement that their application preserves the characteristics of quantum systems.

Considering an operation on a single qubit, it is represented by a  $2 \times 2$  matrix. The normalization condition in (4) must also be preserved after the operation’s application. The associated matrix should be a unitary matrix because applying a unitary matrix to a unit vector results in another unit vector.

Operations on qubit registers also correspond to unitary operations, as in the case of single qubits.

A matrix  $A$  is said to be unitary if the (25) holds:

$$AA^\dagger = I \tag{25}$$

where  $A^\dagger$  denotes the adjoint matrix of  $A$ , and  $I$  is the identity matrix with dimensions matching those of  $A$ . The adjoint matrix of  $A$  is the conjugate transpose of  $A$ , as in (26):

$$A^\dagger = (A^T)^* \tag{26}$$

Therefore, a quantum operation on  $n$  qubits is represented by a square unitary matrix of size  $2^n$ .

The first single-qubit gate that is certainly one of the most used in the definitions of quantum algorithms is the Hadamard gate. The associated representation matrix is:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{27}$$

The Hadamard gate is often used to create a state of superposition on a single qubit. This means that after applying the Hadamard gate, the qubit can exist in a linear combination of basic states. Therefore, a measurement of the output state will have an equal probability of yielding 0 or 1, as in (28) and (29), respectively:

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ &= \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \end{aligned} \tag{28}$$

$$\begin{aligned} H|1\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \\ &= \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \end{aligned} \tag{29}$$

The probability amplitudes associated with each basis state for every obtained ket are equal to  $1/2$ . If the starting state is a generic ket  $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$ , it results as in (30):

$$\begin{aligned} H|\psi\rangle &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} |\psi\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \\ &= \frac{1}{\sqrt{2}} \begin{pmatrix} \alpha + \beta \\ \alpha - \beta \end{pmatrix} = \frac{\alpha + \beta}{\sqrt{2}}|0\rangle + \frac{\alpha - \beta}{\sqrt{2}}|1\rangle \end{aligned} \tag{30}$$

Furthermore, the Pauli matrices and the related quantum gates are to be considered. These are three matrices, generally denoted by the letters X, Y, and Z (sometimes, the notation  $\sigma_x$ ,  $\sigma_y$ , and  $\sigma_z$  is used), and they correspond to a  $180^\circ$  rotation of the qubit around the x, y, and z axes, respectively, in the Bloch sphere, i.e., in the complex space.

In quantum physics, the Pauli matrices stand for the x, z, and y components of an electron's spin, respectively.

The X gate, sometimes referred to as the quantum NOT gate, performs an inversion of the basis states, meaning it swaps the probability amplitudes associated with the two basis states. The relative matrix and applications to qubits are in (31), (32), (33) and (34)

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{31}$$

$$X|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{32}$$

$$X|1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{33}$$

$$X|\psi\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} = \beta|0\rangle + \alpha|1\rangle \tag{34}$$

The Y gate, in addition to swapping the values between them like X, changes the phases of both states by a factor of  $i = e^{i\pi/2}$ , resulting in the matrix (35) and in operations on qubits as in (36), (37) and (38):

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{35}$$

$$Y|0\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ i \end{pmatrix} \tag{36}$$

$$Y|1\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -i \\ 0 \end{pmatrix} \tag{37}$$

$$Y|\psi\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} -i\beta \\ i\alpha \end{pmatrix} = \beta e^{-\frac{i\pi}{2}}|0\rangle + \alpha e^{\frac{i\pi}{2}}|1\rangle \tag{38}$$

The Z gate only acts on the  $|1\rangle$  base state, changing its sign, as in (39), (40), (41) and (42):

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{39}$$

$$Z|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{40}$$

$$Z|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{41}$$

$$Z|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ -\beta \end{pmatrix} = \alpha|0\rangle - \beta|1\rangle \tag{42}$$

The Pauli matrices are involutory, meaning that they are equal to their own inverses. Therefore, applying a Pauli matrix twice to a quantum system does not result in a change. In mathematical terms, the square of the Pauli matrices is the identity matrix, as in (43):

$$X^2 = Y^2 = Z^2 = I \tag{43}$$

Other notable quantum gates are the phase shift gates. They are expressed by matrices having no effect on the  $|0\rangle$  state but acting on the  $|1\rangle$  state only, by introducing a phase variation to the quantum state, i.e., a phase  $\phi$ . Therefore, the probability of measuring  $|0\rangle$  or  $|1\rangle$  does not change.

In particular, the S matrix in (44) performs a 90-degree rotation around the Z-axis in the Bloch sphere. Its operations on a single qubit are in (45), (46) and (47):

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{44}$$

$$S|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{45}$$

$$S|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \end{pmatrix} \tag{46}$$

$$S|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ i\beta \end{pmatrix} = \alpha|0\rangle + i\beta|1\rangle$$

$$= \alpha|0\rangle + e^{\frac{i\pi}{2}}\beta|1\rangle \quad (47)$$

The T gate performs a 45-degree phase rotation around the Z-axis in the Bloch sphere. Its matrix representation is in (48) and its operation on a qubit are in (49), (50) and (51).

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \quad (48)$$

$$T|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (49)$$

$$T|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ e^{\frac{i\pi}{4}} \end{pmatrix} \quad (50)$$

$$T|\psi\rangle = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta e^{\frac{i\pi}{4}} \end{pmatrix} = \alpha|0\rangle + \beta e^{\frac{i\pi}{4}}|1\rangle \quad (51)$$

The Z, S, and T gates can be considered particular cases of phase shift gates, where the phase is respectively equal to  $\pi$ ,  $\frac{\pi}{2}$  and  $\frac{\pi}{4}$

A fundamental theorem for achieving universality in single-qubit operations is the single-qubit Y-Z decomposition theorem.

It is demonstrated that all operations on a single qubit can be expressed using two Z gates and one Y gate.

In fact, let  $\alpha, \beta, \gamma$  and  $\delta$  be real numbers. The unitary operator on a single qubit can be expressed according to (52):

$$U = e^{i\alpha}R_Z(\beta)R_Y(\gamma)R_Z(\delta) \quad (52)$$

being  $R_Z$  and  $R_Y$  unitary rotation matrices generated by the Pauli matrix, according to (53) and (54), respectively:

$$R_Y(\theta) = e^{-\frac{i\theta Y}{2}} = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \quad (53)$$

$$R_Z(\theta) = e^{-\frac{i\theta Z}{2}} = \begin{pmatrix} e^{-\frac{i\theta}{2}} & 0 \\ 0 & e^{\frac{i\theta}{2}} \end{pmatrix} \quad (54)$$

therefore resulting (52) in (55):

$$U = \begin{pmatrix} e^{-i(\alpha-\frac{\beta}{2}-\frac{\gamma}{2})}\cos\frac{\gamma}{2} & -e^{-i(\alpha-\frac{\beta}{2}+\frac{\gamma}{2})}\sin\frac{\gamma}{2} \\ e^{-i(\alpha+\frac{\beta}{2}-\frac{\gamma}{2})}\sin\frac{\gamma}{2} & e^{-i(\alpha+\frac{\beta}{2}+\frac{\gamma}{2})}\cos\frac{\gamma}{2} \end{pmatrix} \quad (55)$$

Logical gates that process two or more qubits are referred to as controlled gates. In these gates, the unitary operator is applied to specific qubits, known as targets, based on the state of other qubits, referred to as controls. It is denoted by the notation in (56):

$$C^{n,k}(U) \quad (56)$$

where n is the number of control qubits, k is the number of target qubits, and U is the matrix representation of the unitary operator in (55).

The first two-qubit gate is the CNOT gate (Controlled NOT), which matrix in in (57). It changes the state of the second qubit only if the first qubit is in the |1⟩ state and

leaves the other basis states unchanged, as in (58), (59), (60) and (61):

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (57)$$

$$CNOT|00\rangle = |00\rangle \quad (58)$$

$$CNOT|01\rangle = |01\rangle \quad (59)$$

$$CNOT|10\rangle = CNOT\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right)$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |11\rangle \quad (60)$$

$$CNOT|11\rangle = CNOT\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right)$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |10\rangle \quad (61)$$

Applied to a generic quantum register consisting of two qubits, as in (62):

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} \quad (62)$$

the CNOT gate perform the transformation in (63):

$$CNOT|\psi\rangle = CNOT\begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{pmatrix}$$

$$= \alpha|00\rangle + \beta|01\rangle + \delta|10\rangle + \gamma|11\rangle \quad (63)$$

The CNOT gate is the quantum analogue of the classical XOR gate applied to the second qubit, while the first qubit stays unchanged. The importance of the CNOT gate in quantum computation is because of any unitary operator can be expressed exactly using CNOT and single-qubit operations: they form a complete basis of infinite cardinality for quantum computation.

A second fundamental two-qubit logic gate is the SWAP gate, which matrix is in (64). It swaps the states between the



two qubits as in (65), (66), (67), (68) and (69):

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (64)$$

$$SWAP |00\rangle = |00\rangle \quad (65)$$

$$SWAP |11\rangle = |11\rangle \quad (66)$$

$$\begin{aligned} SWAP |10\rangle &= SWAP \left( \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right) \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |01\rangle \end{aligned} \quad (67)$$

$$\begin{aligned} SWAP |01\rangle &= SWAP \left( \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right) \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |10\rangle \end{aligned} \quad (68)$$

$$\begin{aligned} SWAP \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \begin{pmatrix} \alpha \\ \delta \\ \beta \\ \gamma \end{pmatrix} \\ &= \alpha |00\rangle + \delta |01\rangle + \beta |10\rangle + \gamma |11\rangle \end{aligned} \quad (69)$$

A third fundamental gate that works on multiple qubits is the Toffoli gate. It is a three-qubit quantum logic gate that plays a significant role in defining the relationship between classical and quantum circuits. Also known as the CCNOT gate (Controlled-Controlled-NOT), the Toffoli gate is a generalization of the CNOT gate (Controlled-NOT) to three qubits. In practice, the Toffoli gate acts as a CNOT gate controlled by two qubits. If both qubits are in the state |1>, it performs the CNOT operation on the third qubit; otherwise, nothing happens. Its matrix representation in (70) reflects this relationship:

$$CCNOT = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (70)$$

Considering a generic ket associated with a quantum register of three qubits, as in (71):

$$\begin{aligned} &\alpha_1 |000\rangle + \alpha_2 |001\rangle + \alpha_3 |010\rangle + \alpha_4 |011\rangle + \alpha_5 |100\rangle \\ &+ \alpha_6 |101\rangle + \alpha_7 |110\rangle + \alpha_8 |111\rangle \end{aligned} \quad (71)$$

applying the CCNOT gate, it becomes as in (72):

$$\begin{aligned} &\alpha_1 |000\rangle + \alpha_2 |001\rangle + \alpha_3 |010\rangle + \alpha_4 |011\rangle + \alpha_5 |100\rangle \\ &+ \alpha_6 |101\rangle + \alpha_8 |110\rangle + \alpha_7 |111\rangle \end{aligned} \quad (72)$$

The classical Toffoli gate is universal for reversible classical computations, meaning that every classical computation can be constructed in a reversible manner using the Toffoli gate. The quantum Toffoli gate, when applied to individual states of the computational basis, performs the same operations as its classical counterpart. Therefore, it can simulate classical computations on a quantum computer. This ensures that it is possible to execute any classical algorithm on a quantum computer.

Finally, another important quantum operator is measurement.

The act of measuring the state of a quantum system results in the collapse of the system from a coherent superposition state to a base state. For instance, when measuring a qubit  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$  the outcome will be a classical bit, which can be 0 or 1 with probabilities  $|\alpha|^2$  and  $|\beta|^2$ , respectively.

### III. QUANTUM COMPUTING EMULATION ON FPGA

From the theoretical background, it is possible to conclude that the quantum evolution of qubits is modelled by applying a sequence of transformations to quantum states. These transformations can be attributed to linear mathematical operators represented by matrices. The fundamental operators, from which all operations can be traced back, have been listed previously.

Any quantum algorithm can be expressed as a sequence of these operators applied to the quantum system.

Based on the earlier considerations, the implementation of quantum algorithms can be simplified to a series of vector-matrix products.

Therefore, it is possible to mimic the behaviour of a quantum computer both through software (in this case, it is called simulation) and through hardware (in which case, it is called emulation).

A comparison among the two approaches is in section I, where the advantages of the emulators have been also highlighted. Hardware-based emulation supplies significant performance improvements compared to software simulators. This explains why there are several works in the literature on this topic [8], [9], [11], [12], [13], [14], [18], [19], [20], [21], [22], [23], [24].

In general, approaches to designing quantum emulators can be categorized into three groups [8], [9], [11], [12], [13], [18], [19], [20], [21], [22], [23], [24]:

- Efficient emulation of quantum algorithms. This involves implementing a prebuilt set of operations, prioritizing time

and/or hardware resource usage over accurately reflecting the physical behaviour and universality of a quantum computer. Proposed designs include HDL libraries, often accompanied by preprocessing software, as well as CPU-like solutions.

- Emulating the behaviour of specific physical quantum circuits. This approach focuses on replicating the physics of a selected group of synthesized circuits. These circuits are typically constructed using tools provided in HDL libraries, without necessarily aiming to build a processing unit with a universal set of instructions capable of executing any quantum algorithm.

- Tools for designing and running quantum algorithms on classical architectures. Designed to allow users to write and run quantum algorithms on their classical machines, these tools do not prioritize emulating the natural massive parallelism characteristic of quantum computers.

The design method developed and described in this paper follows a universal architecture approach. Specifically tailored for FPGA implementation, the proposed method consists of modules responsible for emulating various components of quantum circuits, including quantum gates. This approach fully reflects the mathematical representation of a quantum computer's behaviour.

While many hardware-software systems for emulating quantum computing primarily focus on optimizing hardware efficiency through software, the proposed approach takes a different direction by decoupling hardware and software. In this approach, software is exclusively employed to offer programming abstraction for the hardware. Consequently, the emulator is designed to execute entire quantum algorithms, encompassing state preparation, evolution, and measurement, entirely in hardware. Furthermore, an observation across various emulator projects conducted by scientists reveals a commonality: they adopt a hardware design approach, using hardware description languages in the design phase. This choice often results in project complexity, especially in terms of optimization. The novelty of this work lies in a distinctly different approach, adopting a model-based strategy using MATLAB. This alternative allows for a faster, simpler, and more optimized design tailored for FPGA implementation, using the efficient optimization tools provided by MATLAB.

The next section supplies a detailed description of the proposed procedure.

#### IV. DESCRIPTION OF THE NEW DESIGN APPROACH OF A QUANTUM COMPUTING EMULATOR

MATLAB programming environment offers a rich variety of plugins and toolboxes designed for use across various professional domains, and it can also be used for designing digital electronic systems with FPGAs.

To ease the translation of code from the MATLAB environment to a hardware description language these tools are extensively employed.

The primary toolbox relied upon is undoubtedly HDL Coder. This is a highly significant feature of MATLAB as it enables high-level design, modelling, and coding without

requiring an in-depth knowledge of HDL languages, thanks to its automated and optimized translation capabilities.

HDL Coder helps high-level FPGA design by generating Verilog and VHDL files from MATLAB code [25].

HDL Coder does not work alone; it is complemented by a toolbox known as Fixed-Point Designer. This toolbox plays a crucial role in converting MATLAB code working in double-precision floating-point arithmetic, into fixed-point arithmetic. This translation process, which could be quite challenging if performed manually, is automated by this tool. Unfortunately, however, this tool is general-purpose and not specifically designed for code translation. As a result, it may introduce functions that are not suitable for code conversion. In specific cases, such as quantum computing simulation, it becomes essential to directly write MATLAB code that operates in fixed-point arithmetic rather than relying on automatic fixed-point conversion tools.

HDL Coder toolbox includes a Workflow Advisor, which is a graphical interface that helps programmers in translating MATLAB code into an HDL project and verifying the MATLAB project in hardware.

To translate the MATLAB code into a format usable by HDL Coder, it must be prepared for conversion to HDL. Not any MATLAB function can be translated in HDL.

Therefore, the main challenge of the procedure is to design a MATLAB model of quantum gates in fixed-point arithmetic and using MATLAB functions and custom MATLAB scripts suitable for code translation with the HDL Coder toolbox. The chosen HDL language in this work is Verilog.

In addition to the Verilog HDL code obtained from MATLAB translation, the complete emulator developed in this work includes other files written directly in Verilog HDL for data loading, algorithm application, and output pin assignment for result extraction.

#### A. PROCEDURE FOR CODE CONVERSION FROM MATLAB TO HDL

For a more efficient and faster generation of HDL code, it is advisable to design the MATLAB model of the quantum computing emulator using certain best practices:

- Employ algorithms that use adders and shifting instead of algorithms that rely on functions such as sine, division, and modulo. Addition and shifting operations consume fewer hardware resources.

- Avoid using large arrays and matrices, as they require more registers and additional RAM for storage. When their use is unavoidable, it is recommended to employ RAM mapping optimization to map these memories into RAM.

- Use the fixed-point arithmetic. Floating-point data types are inefficient for hardware implementation.

In a first step, to utilize HDL Coder with one of the supported third-party FPGA synthesis tools, it is necessary to add that tool to the system path using the `hdlsetuptool-path` function. For the Intel FPGAs used in this work the design environment is Quartus. Therefore, the syntax of the command is:

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II',
'ToolPath', ' ')
```

To create a new HDL Coder project, it is possible to start the tool either through its icon, from the “Apps” tab, or by using a manual command in the MATLAB Command Window, as follows:

```
coder -hdlcoder -new function_project
```

and it will appear a user interface of the HDL coder toolbox as in Fig. 2.

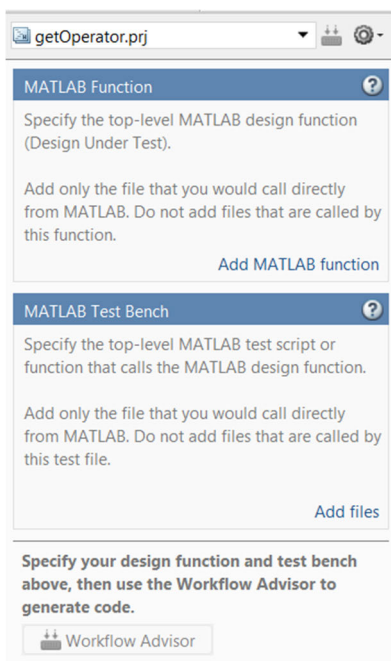


FIGURE 2. User interface of HDL coder toolbox.

In the “MATLAB Function” section is needed to select the MATLAB script to be translated and a related testbench file.

A testbench file is also a MATLAB script suitable to evaluate the algorithm in the file to be translated; it defines the required parameters, invokes the Design Under Test (DUT) function and stores the result. It is not subject to translation.

A well-written testbench file must adhere to certain requirements: it should have no inputs, use the same data types, dimensions, and complexity in every function call. A well-constructed testbench file should cover the entire range of numerical data that the function needs to manage.

The HDL Coder toolbox utilizes data collected from the testbench execution to deduce fixed-point data types when floating-point to fixed-point conversion is required. Consequently, the presence or absence of a testbench file leads to variations in the guided workflow. Another example is the determination of input variable types for the function, which is derived directly from the file.

If the MATLAB code is untranslatable, in the preliminary phase of the translation workflow, the HDL Coder toolbox returns an error message. The “Code Generation Readiness” tool [26] examines the MATLAB code to find features and functions not supported by code generation facility, ultimately providing a report. It is advisable to run the Code Generation Readiness tool before going ahead with code translation by entering the following command in the MATLAB Command Window:

```
coder.screener('function')
```

being “function” the name of the MATLAB file to be examined. A screen opens, with the analysis performed for translation compatibility of the code in the file.

To make MATLAB code compatible for translation into HDL language, it is necessary to follow proper coding rules, as later explained.

If the Code Generation Readiness tool does not detect errors, it is possible to go ahead with defining the data types of the input variables for the function. Then, it is possible to run the Workflow Advisor, as in Fig. 3.

The Workflow Advisor panel is opened, listing all the steps to be taken to reach the analysis and synthesis phase. Subsequently, it leads to the creation of the Quartus project, allowing to exit the MATLAB environment and continue with Quartus. One by one, in order, the tasks suggested by the workflow advisor must be performed.

At the first step, must be chosen whether to continue with converting the variables into fixed-point format. It should be remembered that fixed-point arithmetic is not needed but recommended to generate HDL code used to configure FPGAs. It is possible to go ahead with fixed-point conversion by setting “Convert to fixed-point at build time.” The next task involves configuring the Fixed-Point conversion tool according to design requirements.

When the tool for fixed-point conversion runs, generates a MEX function suitable for the MATLAB project at hand. If compilation is completed without errors, the tool displays compiled information (type, size, complexity) for functions and variables in the code. If compilation errors occur, the tool supplies error messages that refer to the line of code causing the compilation issues. These errors must be resolved to continue further. If the code uses functions unsupported for fixed-point conversion, the tool displays them in the “Function Replacements” tab.

The choice of data ranges for the fixed-point arithmetic conversion can be based on the ranges derived from the execution of the testbench file, user-defined ranges (referred to as static intervals), or a combination of both. When manually inputting static intervals, they take precedence over the derived ranges, and the tool incorporates them in suggesting fixed-point data types. Additionally, the proposed type can be modified and locked, preventing any further modifications by the tool.

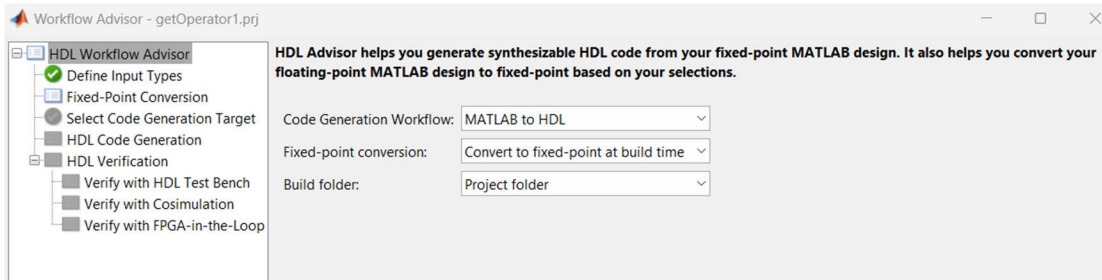


FIGURE 3. Workflow advisor screen.

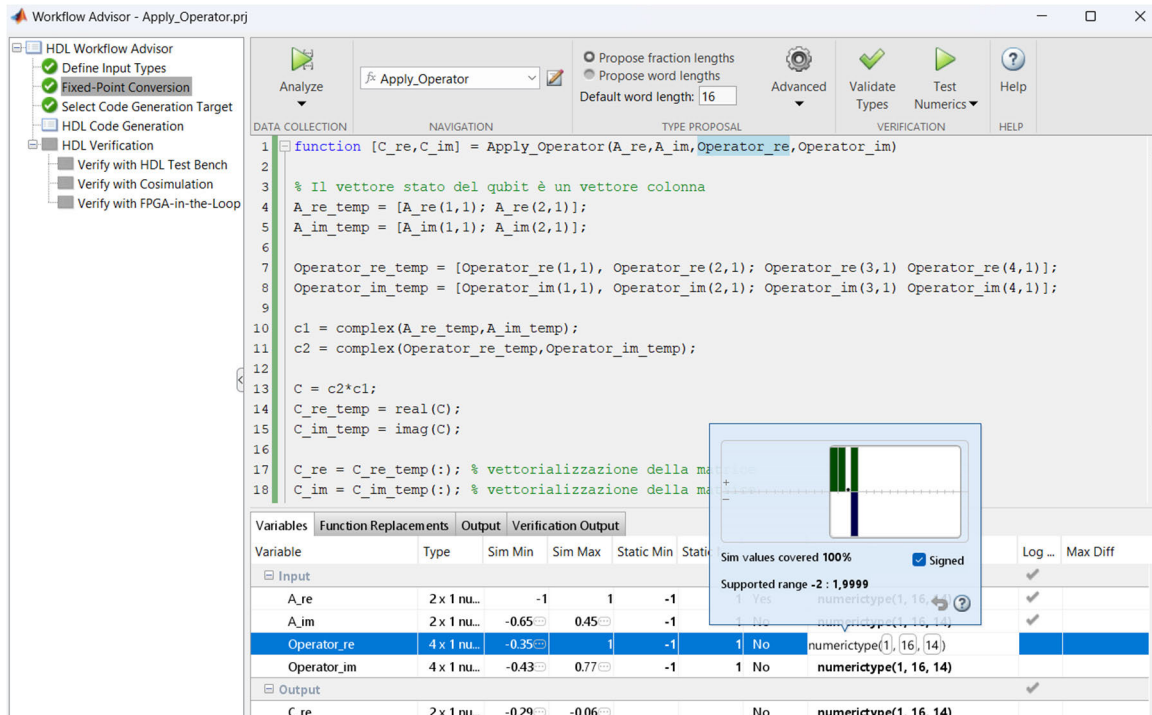


FIGURE 4. Fixed-point conversion screen.

If a testbench file is used, the box labelled “Analyse ranges using simulation” must be checked and the box labelled “Manually specify design ranges” should be checked if static intervals are used. “Analyse ranges using derived range analysis” is employed when a testbench is not used. If static intervals are manually input, two columns appear in the “Variables” section where the lower and upper bounds of the variable intervals for the function are shown.

Before continuing with code analysis, must be specified either the lengths of decimal parts based on predefined word lengths or word lengths based on predefined decimal part lengths. By default, the user specifies the word length, and the tool returns the fractional part length. Following this, the properties of variables resulting from mathematical operations invoked in the code need to be specified. The rounding mode to be executed, referred to as the “Rounding method” must be defined. Additionally, the data type and resizing of

intermediate sums and products can be decided using the properties “SumMode” and “ProductMode”. Decisions on handling potential overflow cases, denoted by “Overflow action”, also need to be made. To go ahead, a click on the “Analyze” button is needed (see Fig. 4).

The “Sim Min” and “Sim Max” tables are now populated with simulation intervals: Sim Min indicates the minimum value assigned to the variable during simulation, while Sim Max represents the maximum value assigned to the variable during simulation.

Fixed-point types are proposed based on default word length settings. The “Types” table holds information for each existing floating-point variable: Signedness, Word length, and fractional part length.

Data types are proposed using the “numerictype” notation. For example, numerictype(1,16,14) indicates a signed fixed-point type with a word length of 16 and a fractional part length of 14, while numerictype(0,14,13) indicates an



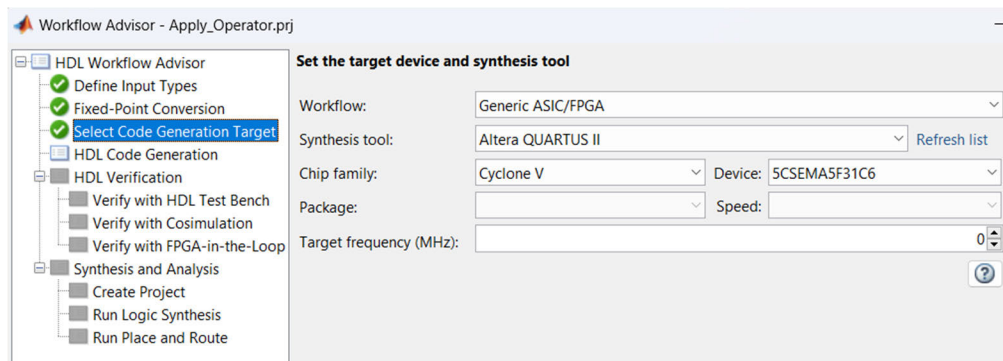


FIGURE 5. Workflow advisor device settings.

unsigned fixed-point type with a word length of 14 and a fractional part length of 13.

The “Type” proposal phase uses Sim Min/Sim Max information and combines it with user-specified word length settings to suggest a fixed-point type for each variable. It is possible also to enable the “Log histogram data” option in the “Analyze” button dropdown menu to log histogram data.

The histogram display supplies concise information about the simulation data range for a variable. The x-axis corresponds to bit weights, and the y-axis to the occurrence count. Proposed numeric type information is overlaid on this chart and can be changed. By moving the white bounding box left or right, can be changed the binary point position; the box’s width does not change because it matches the defined word length. By moving the edges left or right, is altered the fractional part length. All changes made to the proposed type are saved in the project.

If the numerical results do not meet the required precision, the word lengths must be tuned or the types must be manually changed, repeating the rest of the workflow steps until the desired results are achieved.

By selecting “Validate Types” (see Fig. 4) it is validated the code generation using the proposed fixed-point data types.

If the validation is successful, it can be tested the numerical behaviour of the fixed-point MATLAB program by clicking on “Test Numerics” (see Fig. 4). By default, if a testbench has been added to define inputs or conduct a simulation, the tool uses this file for testing numbers.

To detect overflows, the option from the “Test Numerics” dropdown menu: “Use scaled doubles to detect overflows” must be enabled. In case of detection, an “Overflow” tab will be generated with detailed information about the encountered error.

By right-clicking on “Fixed-Point Conversion” (Fig. 4) and then selecting “Run to selected Task” the workflow goes ahead with the task “Select Code Generation Target.” The type of workflow to be performed must be chosen at this stage. In this work, the MATLAB environment to generate hardware description code is used, while the

subsequent phases leading to board programming are done using the Quartus synthesis tool. Therefore, as shown in Fig. 5, “Generic ASIC/FPGA” option must be chosen.

Under “Synthesis Tool,” has been already configured “Altera QUARTUS II” or “Intel Quartus Pro” according to the `hdlsetuptoolpath` function arguments, as previously explained. Then, the Chip family and the Device used must be specified.

The final task to be performed is the generation of the HDL code. It is possible to configure the settings based on project requirements, which are organized into various tabs, the first of which is the “Target”.

In the Target tab, within the “Target Selection” section, the HDL language in which the MATLAB code is converted is defined. In this workflow, Verilog has been chosen, as in Fig. 6. In the second section, “Output settings,” the files to be generated during this task are determined: HDL code for the fixed-point MATLAB function (“Generate HDL”), script files for third-party tools (“Generate EDA Scripts”), and reports for analysing the compatibility of the MATLAB project for HDL code generation (“Check HDL conformance”).

In the “Coding Style” tab, settings related to the coding style that the tool should employ are found. In the first section, the possibility of generating code comments is decided. It is specified whether to keep MATLAB code comments, include the MATLAB source code as comments, and generate a code generation report.

In the “Coding standards” tab, it is specified whether code should be generated according to industrial standards for code writing style. In the affirmative, as in the case of this work, proper modifications can be made according to preferences.

In the “Clocks & Ports” tab, the properties of timing signals and input and output ports of the module to be generated are declared. In this case, default settings are kept.

In the “Optimizations” tab (Fig. 7), certain properties of the HDL code are defined with the intention of optimizing the definition of hardware modules. This tab is divided into four sections: “General,” “Pipelining,” “Area Optimizations,” and “Loop Optimizations.”



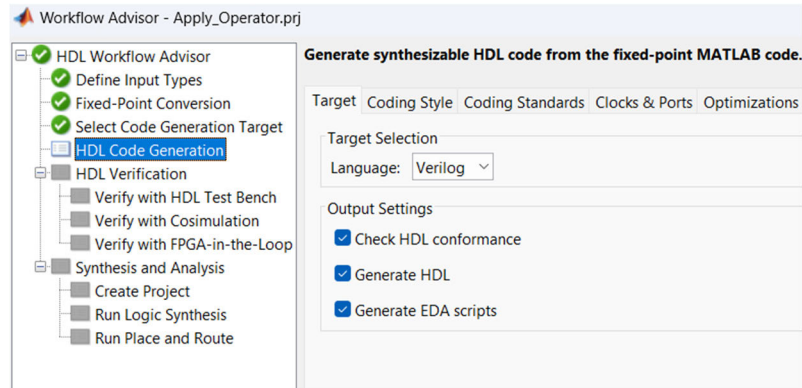


FIGURE 6. Workflow advisor HDL language settings.

The “General” section options allow to decide whether to utilize portions of the installed RAM on the devices and set the corresponding usage thresholds. The option to enable “Aggressive Dataflow Conversion” is available for consideration when employing floating-point IP core modules.

In the “Pipelining” section, various properties are defined to generate a hardware module that follows the pipelining paradigm. This involves adding intermediate registers between individual blocks of the project to reduce resource usage and optimize the maximum achievable clock frequency.

The next sections, “Area Optimizations” and “Loop Optimizations,” should only be edited if the designer has an in-depth understanding of the project. In this work, this tab was changed only in the “Pipelining” section. The other sections were changed only in the intermediate stages of trial and error to experiment with different usage scenarios, but no advantages were observed.

The “Floating Point” tab is ignored if the project does not involve the use of floating-point arithmetic functional blocks, as chosen in this work. Similarly, it should not be changed the “Frame to Sample Conversion” tab.

In the “Advanced” tab, selected options are emphasized: “Generate instantiable code for functions” to generate HDL modules for each function in the MATLAB project, and “Generate Simulink model” which creates a MATLAB functional block for use in a Simulink model. To generate a Simulink block, the tool requires the use of the Fixed-Point Designer.

In the “Diagnostics” section of the “Advanced” tab, the entry associated with the “Check for presence of reals in generated HDL code” property is changed to “Error.” This results in checking and potentially generating an error if the generated HDL code uses “real” data types. It should be noted that HDL code having “real” data types is not synthesizable. If the code is generated for simulation purposes only, this setting can be changed to “Warning” or “None”.

The “Script Options” tab concerning properties related to scripts generated during compilation, simulation, synthesis,

and linting is not taken into consideration. In the final tab, the source MATLAB code appears.

By right-clicking on “HDL Coder Generation” (Fig. 7) and then selecting “Run this Task” the code conversion is performed. In the panel below the settings tabs, it can be seen the log, which consists of the sequential and chronological recording of operations performed during the task.

The HDL code and the relevant reports generated can be examined by clicking on the hyperlinks in the code generation log window.

The procedure described above is the general process for using the MATLAB HDL Coder toolbox, with instructions for configuring settings and executing the steps to complete the conversion from MATLAB to HDL.

However, when trying to write the MATLAB code for quantum logic gates, it is found that the described procedure does not at once succeed.

All the encountered problems are described below in the paper, along with the adopted solutions and the related MATLAB code. For instance, the following are just some of the error messages that MATLAB returns when the designer attempts code translation without adhering to very precise simulator design rules:

Call to function ‘cos’ is not supported for HDL code generation, at Function ‘cos’ (#57.271.281), line 11, column 9 Function ‘applyScalarFunctionInPlace’ (#14.537.549), line 17, column 9 Function ‘cos’ (#56.290.371), line 11, column 5 Function ‘getOperator1\_fixpt’ (#1.1774.1804), line 36, column 19.

Call to function ‘sin’ is not supported for HDL code generation

Call to function ‘pow’ is not supported for HDL code generation

Call to function ‘abs’ is not supported for HDL code generation

Call to function ‘log’ is not supported for HDL code generation

Call to function ‘rtIsInf’ is not supported for HDL code generation

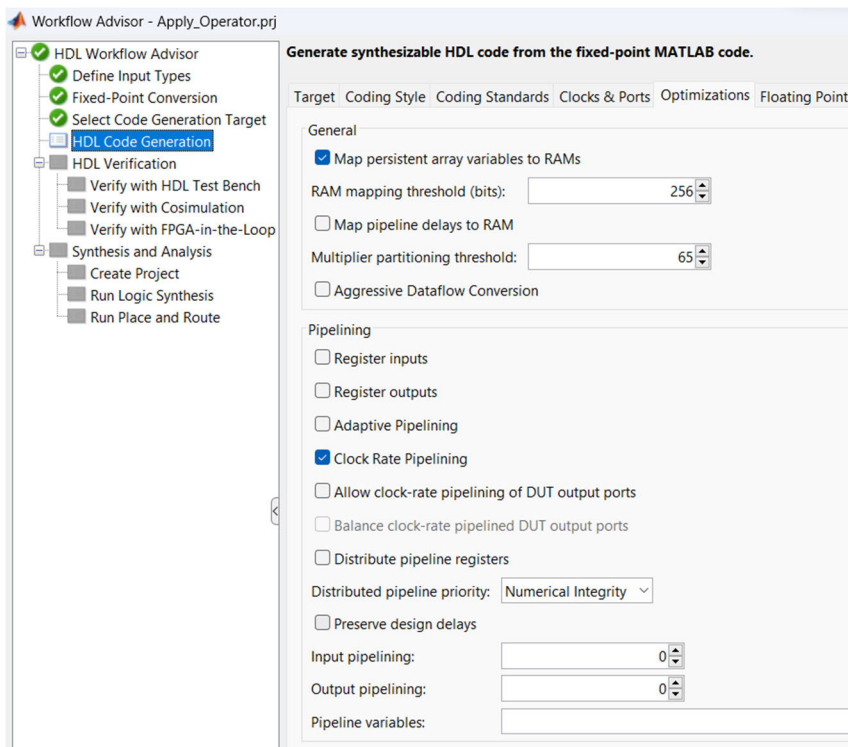


FIGURE 7. Workflow advisor Optimizations settings.

Call to function 'atan2' is not supported for HDL code generation

Call to function 'hypot' is not supported for HDL code generation

Non-scalar types are not supported for derived range analysis.

Found unsupported dimensions on matrix type at output port: 0, name 'op', in the file/function getOperator\_fixpt. Either flatten the matrix to a vector or enable 'FrameToSampleConversion' for large matrices to generate HDL code

Non-synthesizable single or double data type found in HDL code, [...] To fix this problem, cast the floating-point data type to a fixed-point data type.

Therefore, a deeper understanding of key concepts in the fixed-point conversion process and the correct writing of MATLAB code is necessary before providing it to the HDL Coder toolbox. These particularly important aspects are described in the following subsection and then applied to the quantum computing emulator project.

**B. PROCEDURE TO WRITE RIGHT MATLAB FUNCTIONS FOR HDL CODER SUCCESSFULLY PROCESSING**

To make the MATLAB code translatable, sometimes instead of an automatic floating-point to fixed-point conversion (by using the Fixed-Point Designer toolbox), a manual implementation of fixed-point arithmetic is highly recommended to successfully obtain a MATLAB project into an efficient fixed-point implementation. As previously stated, this occurs because the Fixed-Point Designer toolbox sometimes

introduces functions that are not supported for code translation. The fixed-point arithmetic is adopted by using the MATLAB “fi” data type to represent fixed-point numbers and specific functions to perform arithmetic operations on them.

The “fi” class supplies an interface to specify the number of bits used to represent the integer and fractional parts of a number, as well as the position of the fixed point.

Numeric attributes and mathematical rules to follow can be specified. The syntax for creating a “fi” object is:

```
a = fi(value, signedness, Wordlength,
Fractionlength);
```

“value” is the value of the number.

“signedness” is a flag saying whether the number is signed (1) or unsigned (0)

“Wordlength” (WL) is the total number of bits instantiated to represent the number, including those used for the integer and fractional parts.

“Fractionlength” (SL) is the number of bits used for the fractional part of the number.

The properties of the fixed-point object may be defined using two MATLAB functions, “fimath” and “numeric-type.”

The “numeric-type” function has already been described.

The “fimath” function creates a “fimath” object that describes the properties of a “fi” object related to arithmetic operations.

The MATLAB syntax for creating a “fimath” object is:

```
fm = fimath(Name, value);
```

The properties of the “fimath” object are defined using name-value pairs, where the first argument is the property and the second is the value to assign to it.

Multiple properties can be defined. Key properties are:

“RoundingMethod” specifies the rounding mode for arithmetic operations and can take values like ‘Ceiling,’ ‘Convergent,’ ‘Floor,’ ‘Nearest,’ ‘Round,’ ‘Simplest,’ ‘Zero’.

“OverflowMode” declares the overflow handling mode and can be ‘Wrap’ or ‘Saturate’.

“ProductMode” regulates the product handling mode and can be ‘FullPrecision,’ ‘SpecifyPrecision,’ ‘KeepLSB,’ ‘KeepMSB’.

“SumMode” regulates the sum handling mode and can be ‘FullPrecision,’ ‘SpecifyPrecision,’ ‘KeepLSB,’ ‘KeepMSB’.

“ProductWordLength” defines the bit length of the product.

“SumWordLength” specifies the bit length of the sum.

“MaxProductWordLength” defines the maximum acceptable bit length for the product data type, ranging from 2 to 65535.

“MaxSumWordLength” defines the maximum acceptable bit length for the sum data type, ranging from 2 to 65535.

“CastBeforeSum” specifies whether the conversion to the final data type should be performed before the sum operation and can be ‘true’ or ‘false’.

In general, by using “fimath” allows a fine control over the behaviour of fixed-point arithmetic operations, ensuring the correct implementation of the system.

For example, by the command:

```
'OverflowMode' = 'saturate'
```

the values exceeding the representable range are “saturated” to the nearest limit value, avoiding unpredictable system behaviour.

Similarly, by the command:

```
'RoundingMethod' = 'Floor'
```

the values are always rounded down.

Useful programming techniques include subscripting assignment and casting.

In MATLAB environment, subscripting assignment is a technique to assign values to specific elements or subsets of an array or matrix. It allows the designer to select and access a specific part of an array or matrix using indices and then assign new values to those selected elements.

For example, considering a  $3 \times 3$  matrix A, can be used a subscripting assignment to assign a new value to a specific element, such as  $A(2,3) = 5$ , or to select a subset of the matrix

and assign new values to those selected elements, such as  $A(2:3,1:2) = [1,2;3, 4]$ .

A special type of subscripting assignment is:

```
A(:) = b;
```

A focus on this MATLAB syntax is needed. When introducing fixed-point data types into the code, it’s possible that data types may change when overwritten, potentially causing an increase in bit width.

The particularity of this type of assignment is that the data type of the original vector or matrix (in the example, A) does not change. Once the type of the original data is defined, the value associated with A can change, but the data type is forced to be the one defined earlier.

This assignment via subscripting is used to assign a new value to fixed-point variables to preserve the existing data type, the size of the array of the overridden variable, and prevent bit growth.

These last two aspects are particularly important for programming FPGAs.

It’s worth noting that Verilog listings describe how logic elements in the FPGA are connected and which resources will be used. Therefore, it must be declared the amount of memory to use and avoiding excessive and uncontrolled use of resources. These requirements translate into the inability to define variables with variable size and the need to specify the arithmetic rules for fixed-point numbers.

Casting is the process of converting an instance of one data type into another instance of a different data type.

In MATLAB, casting can be done using type conversion functions like `int()`, `double()`, `single()`, etc. For example, it is possible to convert a double variable into an integer variable using the `int()` function, as in `int(3.14)`. Another use of the casting statement is:

```
A = cast([], 'like', '');
```

This statement is a call to the MATLAB casting function, which takes three arguments: the first is the array or variable to convert, the second is a keyword (‘like’) used to specify that the target data type should be the same as the data type of the reference variable or array provided as the third argument.

To ensure a smooth MATLAB code translation process, it is crucial to structure the project by separating the algorithm that will be translated from the remaining code. This includes functions for variable data type definitions and the testbench script, which are useful only for debugging purposes.

The portion of the MATLAB project that implements the algorithm is the only part to be converted to fixed-point. The definition of inputs or the display of results does not require code translation. Additionally, separating data type definitions from the code makes it easier to compare fixed-point implementations and potentially redirects the algorithm to a different device.

Moreover, it is advisable to add the directive `#codegen` in the MATLAB script to use the Fixed-Point Designer toolbox and to use the Code Generation Readiness tool. The `#codegen` directive indicates the intention to generate code instructing the MATLAB code analyser to assist the user in diagnosing and correcting violations that could cause errors during code generation.

An example of MATLAB script applying the functions and practices discussed is as follows:

```
function T = mytypes(dt)
    fm = get_fimath();
    nt = get_numericitype();
    switch(dt)
        case 'double'
            T.opName = double([]);
            T.op = double([]);
        case 'single'
            T.opName = single([]);
            T.op = single([]);
        case 'fixedPoint1'
            T.opName = fi([], false, 4, 0);
            T.op = fi([], true, 32, 30);
        case 'fixedPoint2'
            T.opName = fi([], false, 4, 0);
            op_real = fi([], nt, fm);
            op_imag = fi([], nt, fm);
            T.op = complex(op_real, op_imag);
    end
end
function fm = get_fimath()
    fm = fimath('RoundingMethod', 'Nearest', ...
        'OverflowAction', 'Saturate', ...
        'ProductMode', 'FullPrecision', ...
        'MaxProductWordLength', 32, ...
        'SumMode', 'FullPrecision', ...
        'MaxSumWordLength', 32);
end
function nt = get_numericitype()
    nt = numericitype('DataTypeMode', 'Scaled
double: binary point scaling', ...
        'Signedness', 'Signed', ...
        'WordLength', 32, ...
        'FractionLength', 30);
end
```

It is noticeable the use of the “ScaledDouble” data type, which is a double format that preserves data about WL (Word Length), FL (Fraction Length), and sign.

During code generation, MATLAB suggests the best FL. Also, it is noticeable a syntax different but

analogous to the one shown earlier in the `numericitype` function.

The function in the previous script takes the desired data type (“dt”) as input and returns a structure T that encompasses entities with the required arithmetic and `numericitype` characteristics. This is an example of a function defined with the intention of declaring data types.

It follows an excerpt from the function that is subject to translation, which utilizes casting and subscripting assignment for variable definition:

```
function op = operatore(opName, T) %#codegen
    op = cast(zeros(2), 'like', T.op);
    op(:) = [1 0; 0 1];
```

An alternative to the cast statement is “zeros”:

```
op = zeros(2, 'like', T.op);
```

An example of a testbench file is as follows:

```
% Test inputs
t = linspace(0, 1, 5)';
% Cast inputs
T = mytypes('fixedPoint1');
opName = cast(t, 'like', T.opName);
% Run
op = operatore(opName, T);
```

In general, utilizing the ‘fi’ data type and specific MATLAB functions enables the precise and controlled implementation of fixed-point arithmetic. On the other hand, employing the Fixed-Point Designer toolbox automates the fixed-point design process, saving time and reducing errors. Both approaches are valid, and the designer evaluates which methodology is better suited for each case.

Mathworks’ website lists all functions supported for HDL translation [27]. For each function, in the “Extended capabilities” section, is specified if it supports translation. If a MATLAB function is not supported by the translation tool, a solution may be to replace it with a Look-up Table (LUT) implementation or a CORDIC implementation.

CORDIC-based algorithms are among the most efficient for hardware implementation because they use adders and bit shift operations. They are iterative algorithms, and the designer must specify the number of iterations. The choice of the number of iterations for CORDIC algorithms depends on the required precision for HDL code generation. In general, the more iterations, the higher the precision of the CORDIC algorithm. However, increasing the number of iterations can also increase circuit complexity and power consumption. One strategy is to start with a moderate number of iterations and progressively increase it until the desired precision is achieved. It is also possible to use the convergence method to determine the optimal number of

iterations, which involves monitoring the approximation error for each iteration and stopping iterations when the error is below a certain threshold (the help for CORDIC functions provides a sample script).

It's essential to consider that the choice of the number of iterations also depends on project-specific features such as clock speed and available processing resources.

In general, finding a balance between precision and circuit complexity is crucial to achieve the best result.

Some MATLAB functions, even if supported by HDL Coder, can be replaced to obtain a more efficient fixed-point implementation. For example, to perform the division of a number by a multiple of two, it is advisable to use MATLAB's bit-shifting instruction "bitshift".

In the following, there are some essential rules to avoid errors during the fixed-point conversion phase:

- Structures and matrices are not supported for primary function inputs (or main function inputs), even though it is possible to use them inside the primary function [28].
- HDL Coder toolbox supports fixed-point numbers with a maximum word length of 128 bits [28].
- Variable-sized data is not supported, both in the testbench file and in the primary function [28].
- For each input of the primary function (or DUT - Device Under Test) whose class is fixed-point, they must be specified the "numericType" properties and the "fimath" properties [28].

HDL Coder needs to determine the properties of all variables in MATLAB files during compilation. Following this criterion, HDL Coder should be able to identify the properties of the inputs.

With the methodological insights and best practise rules just described, has been developed the quantum computing simulator which MATLAB code is oriented to the automatic translation in HDL code, in fixed-point arithmetic, that is the core of the hardware quantum computing emulator.

Therefore, in the following subsection is described in detail the MATLAB project and then the procedure for translating it into HDL code.

### C. DESIGN OF A QUANTUM COMPUTING SIMULATOR IN MATLAB

The MATLAB project originates from code previously developed by other authors [29]. However, significant modifications have been implemented in this work to transform it into a simulator suitable for code translation for hardware implementation. Furthermore, the code has been adapted to serve as a quantum computing emulator implemented on an FPGA.

The starting MATLAB code is based on David Deutsch's mathematical model of a universal quantum computer [30]. Therefore, matrix multiplication and tensor products (also known as Kronecker products, both terms will be used interchangeably) between complex numbers are the main advanced mathematical operations used in the model.

The result of the simulation performed by this MATLAB project is a probability distribution of the states of the

quantum computer, in addition to the post-computation state of the system.

A quantum algorithm powered by the quantum computer is a matrix of strings: the rows of the matrix represent the qubits, and the columns represent the algorithm's steps. Each element of the matrix represents a quantum gate.

The MATLAB project implements the previously described quantum gates and, in addition, U1, U2 and U3 as in the following.

The U3 gate is the generic single-qubit rotation gate defined by three angles,  $\theta$ ,  $\phi$ , and  $\lambda$  (Fig. 1), representing rotation angles for the X, Y, and Z axes, respectively. The corresponding matrix is in (73):

$$U3(\theta, \phi, \lambda) := \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda} \sin\left(\frac{\theta}{2}\right) \\ e^{i\phi} \sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)} \cos\left(\frac{\theta}{2}\right) \end{pmatrix} \quad (73)$$

The U1 and U2 gates are special cases of the U3 gate, introduced specifically due to hardware characteristics.

The U1 gate implements a single-qubit rotation around the Z-axis by the angle  $\theta$  given as an input argument, and corresponds to the Phase Shift gate modelled by (74):

$$U1(\lambda) = U3(0, 0, \lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} \quad (74)$$

The U2 gate implements a single-qubit rotation around the X and Z axes by angles  $\phi$  and  $\theta$ , respectively, as in (75):

$$U2(\phi, \lambda) = U3\left(\frac{\pi}{2}, \phi, \lambda\right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\theta} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix} \quad (75)$$

The introduction of U1 and U2 gates is motivated by error rates: the error rate for U1 is immeasurable, whereas the error of U3 is twice that of U2. Consequently, for enhanced fidelity and to minimize the occurrence of gate errors, a circuit exclusively utilizing U1 and U2 gates is expected to outperform one incorporating U3 gates [31], [32].

The function called "getOperator" in the developed MATLAB simulator returns the matrix corresponding to any requested single-qubit quantum gate, which serves as the operator.

The simulator allows the use of a qubit-controlled version of any of the above gates.

The function called "getControlledOperator" returns the matrix for a single-qubit operator controlled by another qubit. The Toffoli gate and Swap gate are also implemented in the MATLAB simulator and the "algorithmPreprocess" function is used to replace the Toffoli and Swap gates with their implementations using CNOT, Hadamard, and T gates.

The main function of the simulator is called "quantumComputer". This function takes three input parameters: a quantum algorithm, a list of measured qubits, and the initial state of the quantum system. The first input parameter contains a matrix representing the quantum algorithm itself, the second parameter is a row vector containing the indices of the measured qubits (indices start at 1), and the initial state contains a column vector representing the state of the



quantum computer at the beginning of the algorithm's execution. The function returns three outputs: the final quantum state, the probabilities of measuring basic states (in standard computational basis), and a matrix representing the processed quantum algorithm.

The “*initializeRegisters*” and “*initializeRegistersInt*” functions are used to generate an initial state. The first function transforms a list of individual qubits into their tensor product, resulting in the initial state of the quantum computer. It follows the relative script.

```
function q = initializeRegisters(varargin)
    %init configuration - all bits set to |0>
    normalized = zeros(nargin,2);
    normalized(:,1) = ones(nargin,1);

    for k = 1:nargin
        hlp = cell2mat(varargin(k)); %q-bits
    normalization
        normalized(k,:) = hlp/norm(hlp);
    end

    hlp = (normalized(1,:)).';
    for k = 2:nargin
        hlp = kron(hlp,(normalized(k,:)).');%Tensor
    product to get final state
    end
    q = hlp;
end
```

The second function transforms integers into the initial state of the quantum computer.

The MATLAB project includes \*.mat files containing some well-known algorithms for testing the simulator. The considered algorithms are as follows:

a) bellState.mat - Preparation of a specific Bell state as in (76):

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (76)$$

After loading the bellState.mat file, the simulation starts by running the command (in MATLAB command window):

```
quantumComputer(qAlgorithm)
```

b) bitSwap.mat - Swapping of two contained qubits.

c) Toffoli.mat - Implementation of the Toffoli gate using CNOT and single-qubit gates.

d) grover\_4bit\_1iter.mat - First iteration of Grover's algorithm searching for the solution to the logical equation

$$AB\bar{C}\bar{D} = 1$$

After loading the \*.mat file, the simulation starts by running the command:

```
quantumComputer(qAlgorithm, [1 2 4 5])
```

e) The grover\_4bit\_2iter.mat and grover\_4bit\_3iter.mat files contain two and three iterations of Grover's algorithm, respectively. The command to enter is like the previous case.

f) statePrep.mat - Preparation of a 3-qubit register with the following state distribution:

```
p(|000>) = 3 %
p(|001>) = 5 %
p(|010>) = 7 %
p(|011>) = 10 %
p(|100>) = 13 %
p(|101>) = 15 %
p(|110>) = 17 %
p(|111>) = 30 %
```

#### D. FROM MATLAB TO HDL: THE QUANTUM COMPUTING EMULATOR PROJECT FOR FPGA IMPLEMENTATION

Expanding on the foundational MATLAB design of a quantum logic gates simulator, as outlined in [29] and summarized in the preceding subsection, in this work have been developed two MATLAB simulators following best practices and code translation guidelines. This stems from the implementation of two distinct emulators: one tailored for applying quantum algorithms to a single qubit, and a second one designed for application to a two-qubit system. The need to describe two different emulator projects arises from different computational requirements and hardware resources in each case.

In fact, for emulating a quantum computing on a single qubit, an Intel Cyclone V FPGA is enough, while emulating a quantum computing on a two-qubit register requires the use of an Intel Arria 10 FPGA due to the greater need for resources.

The computation on a single-qubit system involves left multiplication between the matrix representing the quantum algorithm (of size  $2 \times 2$ ) and the column vector representing the “ket” of the system.

In cases where there are multiple operators, multiplication between the matrices of individual operators is performed.

It's important to note that matrix multiplication is not commutative, so when the algorithm involves applying two or more matrices, the order in which matrix multiplication is performed must be respected, as shown in the following example.

A representation of applying quantum operators (A, B, and C in this order) to a qubit ( $|q_0\rangle$ ) is as in (77)

$$|q_0\rangle \rightarrow -A - -B - -C - - \quad (77)$$

The equivalent mathematical expression is in (78):

$$|q_1\rangle = C * B * A * |q_0\rangle \quad (78)$$

The project of the single-qubit MATLAB simulator consists of five scripts:

*getOperator.m* returns the matrix associated with the requested quantum gate or operator.

*matrix\_complex\_mults.m* performs the multiplication between operators.

*probFinal\_oneQubit.m* calculates the probability distribution associated with qubit measurement.

*Apply\_operator.m* performs left multiplication of the matrix associated with the algorithm with the qubit ket.

*quantumComputer\_oneQubit.m* calls all the other functions and is the main function.

Two-qubit quantum computing involves qubits being interconnected so that the state of one qubit affects the state of the other, creating a quantum register. A two-qubit quantum register is a superposition of states as in (79):

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \quad (79)$$

A column vector composed of complex coefficients  $\alpha_{ij}$  represents the quantum register, obtained through the Kronecker product between the two kets of individual qubits.

The computation on a two-qubit system involves left multiplication between the matrix representing the quantum algorithm (of size  $4 \times 4$ ) and the column vector (of size  $4 \times 1$ ). To construct the algorithm, operators must be specified for each qubit (these operators are represented as  $2 \times 2$  complex matrices, like the single-qubit case). After selecting the operators for each qubit, the Kronecker product is employed to calculate the tensor product between the operators of the first qubit and those of the second qubit. The result is a complex  $4 \times 4$  matrix representing the combined effect of the operators on the two qubits, forming the operators of the two-qubit quantum register.

This process is repeated for each operator to be applied to the quantum register. Matrix multiplication is then used to combine the operators in the correct order, analogous to the single-qubit case with necessary precautions. Finally, the resulting matrix must be applied to the initial state of the quantum register using matrix multiplication.

The project of the two-qubit MATLAB simulator consists of the following six scripts:

*getOperator.m* returns the matrix associated with the requested single-qubit operator. This function is different from the single-qubit case because a new operator, the Controlled operator is needed.

*getControlledOperator.m* returns the matrix for a single-qubit operator controlled by another qubit.

*matrix\_complex\_mults.m* performs the multiplication between operators ( $4 \times 4$  matrices).

*probFinal\_twoQubit.m* calculates the probability distribution associated with measuring the state of individual qubits and the probability of measuring both qubits.

*Apply\_operator.m* performs the left multiplication of the matrix associated with the algorithm with the quantum register ket.

*quantumComputer\_twoQubit.m* calls all the other functions and is the main function of the simulator.

Upon reviewing the list of MATLAB scripts only, it becomes apparent that the original MATLAB project [29] for a quantum logic gates simulator has undergone extensive revisions to develop the two simulators necessary for single and double qubit computing.

Furthermore, substantial coding modifications have been implemented to adapt the original MATLAB code for code translation and to make it suitable for FPGA implementation. In the sections that follow, the codes for all the mentioned scripts are provided, accompanied by a more detailed description of their contents. Each script has undergone successful steps for translation into Verilog HDL, following the procedure outlined in the previous section.

### getOperator.m

This function is designed to return the matrix associated with each requested single-qubit operator. It requires four input arguments and returns one output argument.

As the first input argument, the function is passed the name of the operator to be used, denoted as “opName” that is a character string. The subsequent three variables represent

```
function op = getOperator(opName, theta, phi,
lambda)
global A;
switch opName
case 'I' % 0
op = [1 0; 0 1];
case 'X' % 1
op = [0 1; 1 0];
case 'Y' % 2
op = [0 -1i; 1i 0];
case 'Z' % 3
op = [1 0; 0 -1];
case 'H' % 4
op = (1/sqrt(2))*[1 1; 1 -1];
case 'S' % 5
op = [1 0; 0 1i];
case 'T' % 6
op = [1 0; 0 exp(1i*pi/4)];
case 'DS' % 7
op = [1 0; 0 -1i];
case 'DT' % 8
op = [1 0; 0 exp(-1i*pi/4)];
case 'U1' % 9
op = [1 0; 0 exp(1i*theta)];
case 'U2' % 10
op = (1/sqrt(2))*[1 -exp(1i*theta);
exp(1i*phi) ...
exp(1i*(theta +
phi))];
case 'U3' % 11
op = [cos(theta/2) -
exp(1i*lambda)*sin(theta/2); ...
exp(1i*phi)*sin(theta/2) exp(1i*(lambda +
phi))*cos(theta/2)];
case 'RX' % 12
op = [cos(theta/2) -1i*sin(theta/2); -
1i*sin(theta/2) cos(theta/2)];
case 'RY' % 13
op = [cos(theta/2) -sin(theta/2);
sin(theta/2) cos(theta/2)];
case 'RZ' % 14
op = [exp(-1i*theta/2) 0; 0
exp(1i*theta/2)];
case 'EXP'
op = expm(theta*A);
otherwise
op = [0 0; 0 0]; % Null matrix
end
end
```

the angles required for defining the matrices. Specifically, the first angle,  $\theta$ , is used for defining the matrices U1, U2, U3, Rx, Ry, Rz, and EXP (exponential); the angle  $\phi$  is used for matrices U2 and U3, and the angle  $\lambda$  is used for U3. The switch-case control structure is employed.

The MATLAB code of the `getOperator.m` function is:

The considered script may appear simple in its definition in a high-level language like MATLAB but proves to be very difficult to immediately convert into Verilog HDL.

Before proceeding with the code translation, it was decided to modify the data type of the variable “opName” to make it an 8-bit unsigned integer. It is noticeable that the “char” type in MATLAB reserves 16 bits per element, and “opName” must necessarily be a vector of 3 characters ( $16 \times 3 = 48$  bits). The modification results in a memory savings of 40 bits.

Moreover, it is preferable to define the “op” matrix as null by default due to in a hardware description language all possible conditions must be expressed, unlike a programming language like MATLAB, where it is optional. It was chosen to define it as a null matrix to immediately detect errors in the quantum algorithm’s definition.

Moreover, the exponential function (“exp”) is not supported for code translation. Therefore, it has been replaced with its fifth-term Taylor series expansion.

This attempt exposes additional functionalities and properties that are not supported for translation.

Also, all trigonometric functions and some MATLAB functions introduced by the Fixed-Point Designer, are not suitable for code translation. Therefore, the built-in MATLAB trigonometric functions (“sin” and “cos”) were replaced with their equivalent CORDIC functions (“cordic-sin” and “cordiccos”) [33], and (80):

$$\exp\left(1t * \frac{\pi}{4}\right) \quad (80)$$

is replaced with the associated numerical values:  $0.707429206709773 + 1i0.707429206709773$ .

For a project where variable ranges are known in advance, it is possible to force the Fixed-Point Designer to convert the code into fixed-point data types defined by the designer. Moreover, the matrix “op” has been reshaped into a single column vector by stacking the columns of the original matrix:

```
op = op(:);
```

The next step involved modifying the HDL Coder settings, specifically to recognize the presence of “real” data type variables in HDL code as an error.

The implementation of fixed-point arithmetic was continued using the “fi” constructor and MATLAB functions discussed in previous section.

The final ‘getOperator’ script suitable for code translation is:

```
function op = getOperator(opName,theta,phi,lambda)
fm = fimath('RoundingMethod','Nearest', ...
'OverflowAction','Saturate', ...
'ProductMode','SpecifyPrecision', ...
'ProductWordLength',16, ...
'ProductFractionLength',15, ...
'SumMode','SpecifyPrecision', ...
'SumWordLength',16, ...
'SumFractionLength',15, ...
'CastBeforeSum',true);
nt = numerictype(1,16,15);
op_real = fi([],nt,fm);
op_imag = fi([],nt,fm);
op_template = complex(op_real,op_imag);

[seno_phi,coseno_phi] = cordicsincos(phi,15);

half_theta = bitsra(theta,1);
[seno_half_theta,coseno_half_theta] =
cordicsincos(half_theta,15);

[seno_theta,coseno_theta] = cordicsincos(theta,15);

[seno_lambda,coseno_lambda] =
cordicsincos(lambda,15);

op = cast([0; 0; 0; 0], 'like', op_template);

if opName == fi(0,0,16,0) % Matrice Identità
op(:) = [1 0; 0 1];
elseif opName == fi(1,0,16,0) % Matrice X
op(:) = [0 1; 1 0];
elseif opName == fi(2,0,16,0) % Matrice Y
op(:) = [0 -1i; 1i 0];
elseif opName == fi(3,0,16,0) % Matrice Z
op(:) = [1 0; 0 -1];
elseif opName == fi(4,0,16,0) % Matrice H
op(:) = 0.707106781186547*[1 1; 1 -1];
elseif opName == fi(5,0,16,0) % Matrice s
op(:) = [1 0; 0 1i];
elseif opName == fi(6,0,16,0) % Matrice T
op(:) = [1 0; 0
0.707106781186547+1i*0.707106781186547];
elseif opName == fi(7,0,16,0) % Matrice DS
op(:) = [1 0; 0 -1i];
elseif opName == fi(8,0,16,0) % Matrice DT
op(:) = [1 0; 0 0.707106781186547-
1i*0.707106781186547];
elseif opName == fi(9,0,16,0) % Matrice U1
op(:) = [1 0; 0 coseno_theta+1i*seno_theta];
elseif opName == fi(10,0,16,0) % Matrice U2
op(:) = 0.707106781186547*[1 -coseno_theta-
1i*seno_theta; ...
coseno_phi+1i*seno_phi
coseno_theta*coseno_phi-
```

```

seno_theta*seno_phi+1i*(seno_theta*coseno_phi+coseno
_theta*seno_phi)];
elseif opName == fi(11,0,16,0) % Matrice U3
    op(:) = [coseno_half_theta -
(coseno_lambda*seno_half_theta +
1i*(seno_lambda*seno_half_theta)); ...

coseno_phi*seno_half_theta+1i*seno_phi*seno_half_the
ta ...
(coseno_lambda*coseno_phi-
seno_lambda*seno_phi)*coseno_half_theta+1i*(seno_lam
bda*coseno_phi+coseno_lambda*seno_phi)*coseno_half_t
heta];
elseif opName == fi(12,0,16,0) % Matrice RX
    op(:) = [coseno_half_theta -1i*seno_half_theta;
-1i*seno_half_theta coseno_half_theta];
elseif opName == fi(13,0,16,0) % Matrice RY
    op(:) = [coseno_half_theta -seno_half_theta;
seno_half_theta coseno_half_theta];
elseif opName == fi(14,0,16,0) % Matrice RZ
    op(:) = [coseno_half_theta-1i*seno_half_theta
0; 0 coseno_half_theta+1i*seno_half_theta];
else
    op(:) = [0 0; 0 0]; % Matrice Nulla
end
end

```

In the “fimath” function the approximation method used is ‘Nearest’, which ensures rounding to the nearest representable number. Additionally, in the case of equal distance between two fixed-point numbers, they are rounded to the nearest representable number in the positive infinity direction. The choice is based on two reasons: it has a moderate computational cost [34] and is the best solution in relation to the proposed numeric type. The numeric type (1,16,15) allows representing numbers ranging from -1 to a maximum of 0.999969482421875; the smallest representable value is 0.000030517578125.

Regarding how to handle Overflow, ‘Saturate’ was chosen instead of ‘Wrap.’ The latter implies that overflow should be handled by ‘wrapping’ the most significant bits around the least significant bits, in other words, if a number exceeds the maximum representable limit in a certain number of bits, the most significant bits are ‘wrapped’ around the least significant bits to produce a new number. This can lead to unexpected results and loss of precision. For example, adding two fixed-point numbers with fimath and numeric type as indicated, whose sum is 1, the calculator returns -1.

By setting ‘ProductMode’ and ‘SumMode’ to ‘FullPrecision,’ it would mean that the product and sum would be calculated with the maximum precision available on the system. This means there would be no rounding or bit

truncation during these operations. This mode of handling products and sums allows for the highest possible precision but can also lead to memory or performance issues when dealing with large amounts of data.

For these reasons, “fimath” specifies the precision and word length used for product and sum operations. Precision is specified using ‘ProductFractionLength’ and ‘SumFractionLength’ while word length is defined using ‘ProductWordLength’ and ‘SumWordLength’

In this code, it specifies that product operations will use a word length of 16 bits and a fraction length of 15 bits, and sum operations will also use a word length of 16 bits and a fraction length of 15 bits. Additionally, the ‘CastBeforeSum’ parameter specifies that data should be converted before being summed. Forcing the product and sum to have a limited number of bits, in this case with the same numeric type for factors and addends, is necessary due to the limited resources available on the FPGA Cyclone V.

The defined number of iterations is 15: for fixed-point operations, the maximum number of iterations is one less than the word length of the angle [34].

```
half_theta = bitsra(theta,1);
```

The ‘bitsra’ (BIT Shift Right Arithmetic) instruction shifts the bits to the right while maintaining the sign bit (MSB), representing the sign. It is the fastest and most efficient way to divide by a multiple of 2.

The ‘opName’ variable is also a fixed-point number with a word length of 16. This modification will be beneficial for generating words to be loaded into the ROM of the system, in the Quartus FPGA’s design environment.

The version of the same function for a two-qubit emulator has a different ‘fimath.’ In particular, ‘ProductMode’ and ‘SumMode’ are set to ‘FullPrecision’ because a high-performance FPGA board is used to its fullest potential.

Additionally, for a two-qubit emulator, a new operator is introduced, the Controlled Operator (‘opName’ == 15), to implement controlled quantum gates, quantum operators that characterize systems with two or more qubits:

```
elseif opName == fi(15,0,16,0)
    op(:) = [1 0; 0 1];
```

In the case where the required operator is a controlled operator indicated by the number 15, the corresponding qubit will act as the control qubit, and the identity operator will be applied to it.

#### quantumComputer\_oneQubit.m

It is the main function for a single qubit quantum computing execution. It follows the complete MATLAB code:

This function executes the quantum algorithm on a single qubit and requires the algorithm steps as mathematical operators to apply, following the model of David Shor’s quantum computer [23]. This function employs three quantum operators.

```
function [computerState, probFinal] =
quantumComputer_oneQubit(opName,theta,phi,lambda,ini
tState)
nt = numerictype(1, 16, 15);
template_real = fi([],nt);
template_imag = fi([],nt);
template = complex(template_real,template_imag);
computerState = cast([0; 0], 'like', template);
op_0 =
getOperator(opName(1,1),theta(1,1),phi(1,1),lambda(1
,1));
op_1 =
getOperator(opName(2,1),theta(2,1),phi(2,1),lambda(2
,1));
op_2 =
getOperator(opName(3,1),theta(3,1),phi(3,1),lambda(3
,1));
op_temp = matrix_complex_mults(op_1,op_0);
op = matrix_complex_mults(op_2,op_temp);
computerState(:) = Apply_Operator(initState,op);
probFinal = probFinal_oneQubit(computerState);
end
```

Firstly, the vector representing the quantum state after the algorithm's application is initialized as a  $2 \times 1$  array of complex numbers. Subsequently, the three quantum operators that will constitute the algorithm are obtained using the "getOperator" function, which takes "opName" "theta" "phi" and "lambda" values as inputs. These operators are then multiplied together in a specific order using the "matrix\_complex\_mults" function.

Finally, the initial qubit state is modified using the obtained operator and applied to the initial qubit state with the "Apply\_Operator" function. The final probability distribution is then calculated using the "probFinal\_oneQubit" function.

#### matrix\_complex\_mults.m

```
function C = matrix_complex_mults(A,B)
% Prodotto tra matrici complesse
% C = A*B
A_temp = reshape(A,2,2);
B_temp = reshape(B,2,2);
C_temp = A_temp*B_temp;
C = C_temp(:);
end
```

The function called "matrix\_complex\_mults", takes two complex matrices A and B as input and returns their product C. The function uses the reshape instruction to convert the column vectors A and B into  $2 \times 2$  matrices. It then employs the standard multiplication operator to calculate the product between A\_temp and B\_temp. Finally, C\_temp is converted back into a column vector, which is assigned to

the output variable C. The reshaping operation of inputs and outputs is required by the toolbox to ensure compatibility. The same procedure applies to all other auxiliary functions, whose inputs and outputs consist of matrices.

#### Apply\_Operator.m

```
function C = Apply_Operator(A,Operator)
Operator_temp = reshape(Operator,2,2);
C = Operator_temp*A;
end
```

The function "Apply\_Operator" performs the application of a matrix operator to a quantum vector. In this case as well, the 'reshape' operation is used.

#### probFinal\_oneQubit.m

```
function probFinal = probFinal_oneQubit(a)
fm = fimath('RoundingMethod', 'Nearest', ...
'OverflowAction', 'Saturate', ...
'ProductMode', 'SpecifyPrecision', ...
'ProductWordLength', 16, ...
'ProductFractionLength', 15, ...
'SumMode', 'SpecifyPrecision', ...
'SumWordLength', 16, ...
'SumFractionLength', 15, ...
'CastBeforeSum', true);
nt = numerictype(0, 16, 16);
probFinal = fi([0; 0],nt,fm);
abs_value = cordicabs(a,15);
probFinal(:) = abs_value.*abs_value;
end
```

The "probFinal\_oneQubit" function accepts an input "a" that represents the state of a qubit and aims to calculate the probability that the system collapses into a particular basis state when the qubit is measured.

MATLAB's "fimath" is used to specify the fixed-point numerical representation used in the calculations. This component uses the same settings identified in the "getOperator" function. As for the variable type, the function uses the "numerictype" component to define "probFinal" as an unsigned fixed-point element with the entire bit sequence used to describe the fractional part.

The function uses a CORDIC algorithm, "cordicabs" [33], to calculate the absolute value of the quantum state, using a precision of 15 bits. The result is saved in the variable "abs\_value."

Finally, the function returns the variable "probFinal," which is the value of "abs\_value" squared. Squaring is performed as an element-wise multiplication of the array by itself for compatibility with HDL Coder toolbox. This array represents the probability of obtaining a particular state when the qubit is measured; it should be noted that the sum of



the probabilities of obtaining each individual basis state must equal 1.

```
function [computerState, probFinal, probQubit0,
probQubit1] =
quantumComputer_twoQubit(opName_0,theta_0,phi_0,lamb
da_0,opName_1,theta_1,phi_1,lambda_1,initState_0,ini
tState_1)
    nt = numerictype(1, 16, 15);
    State_template_real = fi([],nt);
    State_template_imag = fi([],nt);
    State_template =
complex(State_template_real,State_template_imag);
    computerState =
cast(zeros(4,1),'like',State_template);
    initState =
two_qubit_register(initState_0,initState_1); %
creazione registro quantistico a 2 Qubit
    % definizione degli operatori da applicare al primo
Qubit
    op_0_0 =
getOperator(opName_0(1,1),theta_0(1,1),phi_0(1,1),la
mbda_0(1,1));
    op_1_0 =
getOperator(opName_0(2,1),theta_0(2,1),phi_0(2,1),la
mbda_0(2,1));
    op_2_0 =
getOperator(opName_0(3,1),theta_0(3,1),phi_0(3,1),la
mbda_0(3,1));
    % definizione degli operatori da applicare al
secondo Qubit
    op_0_1 =
getOperator(opName_1(1,1),theta_1(1,1),phi_1(1,1),la
mbda_1(1,1));
    op_1_1 =
getOperator(opName_1(2,1),theta_1(2,1),phi_1(2,1),la
mbda_1(2,1));
    op_2_1 =
getOperator(opName_1(3,1),theta_1(3,1),phi_1(3,1),la
mbda_1(3,1));
    % definizione degli operatori da applicare al
registro quantistico
    if (opName_0(1,1) == fi(15,0,16,0))
        ControlQubit = fi(0,0,16,0);
        op_0 =
getControlledOperator(op_0_1,ControlQubit);
    else
        if (opName_1(1,1) == fi(15,0,16,0))
            ControlQubit = fi(1,0,16,0);
            op_0 =
getControlledOperator(op_0_0,ControlQubit);
```

In the following there is the description of the MATLAB scripts for the two-qubit project, also.

### quantumComputer\_twoQubit.m

It is the main function for a two-qubit quantum computing execution, which code is:

```
else
    op_0 =
prodotto_di_kronecker(op_0_0,op_0_1);
end
end
if (opName_0(2,1) == fi(15,0,16,0))
    ControlQubit = fi(0,0,16,0);
    op_1 =
getControlledOperator(op_1_1,ControlQubit);
else
    if (opName_1(2,1) == fi(15,0,16,0))
        ControlQubit = fi(1,0,16,0);
        op_1 =
getControlledOperator(op_1_0,ControlQubit);
    else
        op_1 =
prodotto_di_kronecker(op_1_0,op_1_1);
    end
end
if (opName_0(3,1) == fi(15,0,16,0))
    ControlQubit = fi(0,0,16,0);
    op_2 =
getControlledOperator(op_2_1,ControlQubit);
else
    if (opName_1(3,1) == fi(15,0,16,0))
        ControlQubit = fi(1,0,16,0);
        op_2 =
getControlledOperator(op_2_0,ControlQubit);
    else
        op_2 =
prodotto_di_kronecker(op_2_0,op_2_1);
    end
end
op_temp = matrix_complex_mults(op_1,op_0);
algMatrix = matrix_complex_mults(op_2,op_temp);
computerState(:) =
Apply_Operator(algMatrix,initState);
[probFinal, probQubit0, probQubit1] =
probFinal_twoQubit(computerState);
end
```

The “quantumComputer\_twoQubit” function performs a quantum algorithm on a two-qubit register. The algorithm is

presented as in (81) and (82):

$$|initState_0\rangle : - - op_{00} - - op_{10} - - op_{20} \quad (81)$$

$$|initState_1\rangle : - - op_{01} - - op_{11} - - op_{21} \quad (82)$$

where “op<sub>i</sub>” are the operators to be applied to individual qubits, “i” is the operator index, and “j” is the qubit index within the quantum register.

The result of the quantum computation is as in (83):

$$|computerState\rangle = op_2 * op_1 * op_0 * |initState\rangle \quad (83)$$

The function takes as input the names of the operators, the angles for their parameterization, and the initial state of each qubit. It then creates the two-qubit register using the “two\_qubit\_register” function and defines the operators to be applied to each qubit.

Next, it defines the operators to be applied to the entire quantum register: a control structure (if-else) checks if a controlled logic gate is requested. If so, the function calculates the controlled operator using the “getControlledOperator” function; otherwise, it performs the Kronecker product between the operators to be applied to each qubit.

The final state of the computer is obtained by applying the operators in the order specified by the algorithm. The function returns the final state of the computer, the probability of the final state of the register, and the probability that the two qubits are in either the state 0 or 1.

It follows the description of the scripts and related functions called by the “quantumComputer\_twoQubit” function:

#### two\_qubit\_register.m

```
function C = two_qubit_register(c1,c2)

c_00 = c1(1,1)*c2(1,1);

c_01 = c1(1,1)*c2(2,1);

c_10 = c1(2,1)*c2(1,1);

c_11 = c1(2,1)*c2(2,1);

C = [c_00;c_01;c_10;c_11];

end
```

This script calculates the Kronecker product of two qubits, represented as  $2 \times 1$  column vectors, which is necessary for generating a two-qubit quantum register. The code uses bra-ket notation to represent the qubits, where each qubit is represented as a  $2 \times 1$  column vector.

The code uses four variables to represent the coefficients of the Kronecker product between the two qubits:

$$c_{00}, c_{01}, c_{10}, c_{11}$$

For each pair of coefficients from the two qubits,  $c1(1,1)$  and  $c2(1,1)$ ,  $c1(1,1)$  and  $c2(2,1)$ ,  $c1(2,1)$  and  $c2(1,1)$ ,  $c1(2,1)$  and  $c2(2,1)$ , they are multiplied and assigned to their respective variables. Finally, a column vector C is created with the values of the four variables, representing the result of the

Kronecker product between the two qubits, i.e., the ket of the quantum register.

#### prodotto\_di\_kronecker.m

```
function C = prodotto_di_kronecker(c1,c2)

fm = fimath('RoundingMethod', 'Nearest', ...
'OverflowAction', 'Saturate', ...
'ProductMode', 'FullPrecision', ...
'SumMode', 'FullPrecision');

nt = numerictype(1, 16, 15);

C_real = fi([],nt,fm);
C_imag = fi([],nt,fm);
C_template = complex(C_real,C_imag);
C = cast(zeros(16,1),'like', C_template);

c1_temp = reshape(c1,2,2);
c2_temp = reshape(c2,2,2);
c_00 = c1_temp(1,1)*c2_temp;
c_01 = c1_temp(1,2)*c2_temp;
c_10 = c1_temp(2,1)*c2_temp;
c_11 = c1_temp(2,2)*c2_temp;
C_temp = [c_00 c_01;c_10 c_11];
C(:) = C_temp(:);

end
```

This function calculates the Kronecker product between two complex  $2 \times 2$  matrices, which respectively represent the operator applied to the first and second qubits. The result is a complex  $4 \times 4$  matrix, which is “flattened” to support the HDL Coder tool.

The output variable is defined according to the expected numerical characteristics, like what was done in previous functions.

Then, an empty matrix “C” is created with specific precision and numeric type options. Subsequently, temporary matrices “c1\_temp” and “c2\_temp” are created to hold the input matrices. The Kronecker product is computed between each pair of elements of the temporary matrices, resulting in four matrices: “c\_00,” “c\_01,” “c\_10,” and “c\_11.” These matrices are then combined into a  $4 \times 4$  matrix “C\_temp.”

Finally, there is an assignment for subscripting, along with the flattening of the “C\_temp” matrix into a one-dimensional vector C.

#### getControlledOperator.m

The “getControlledOperator” function takes two parameters as input: “baseOp,” which represents the operator to be applied to the target qubit, and “ControlQubit,” which represents the index of the control qubit. The code returns a  $4 \times 4$  matrix representing the single-qubit operator controlled by another qubit.

Initially, properties are defined for the “op” object, including overflow action, rounding method, product mode, and sum mode. Subsequently, two variables are created for the real and imaginary elements of “op,” and

```

function op =
getControlledOperator(baseOp,ControlQubit)

fm = fimath('RoundingMethod', 'Nearest', ...
'OverflowAction', 'Saturate', ...
'ProductMode', 'FullPrecision', ...
'SumMode', 'FullPrecision');

nt = numerictype(1, 16, 15);
op_real = fi([],nt,fm);

op_imag = fi([],nt,fm);
op_template = complex(op_real,op_imag);
op = cast(zeros(16,1),'like',op_template);
baseOp_temp = reshape(baseOp,2,2);
    if ControlQubit == 0
        op(:) = [eye(2) zeros(2); zeros(2)
baseOp_temp];
    elseif ControlQubit == 1
        op(:) = [1 0 0 0; 0 baseOp_temp(1,1)
0 baseOp_temp(1,2);
                0 0 1 0; 0 baseOp_temp(2,1)
0 baseOp_temp(2,2)];
    else
        op(:) = zeros(16,1);
    end
end

```

a template for “op” is defined as a complex number composed of the two just-created variables. These are the same instructions used in the “getOperator” function.

The “baseOp\_temp” matrix is then resized into a  $2 \times 2$  matrix using the reshape function. At this point, the code performs a conditional check on “ControlQubit.” If “ControlQubit” equals 0, the control qubit is the first qubit, and the returned operator is a  $4 \times 4$  matrix.

It is worth of notice that the controlled operator described is an extension of the CNOT logic gate. Given the operator to be applied to the second qubit represented as in (84):

$$baseOp_{temp} = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix} \quad (84)$$

The corresponding controlled operator is as in (85):

$$op_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & U_{11} & U_{12} \\ 0 & 0 & U_{21} & U_{22} \end{bmatrix} \quad (85)$$

The operator is created as a  $4 \times 4$  matrix with the elements of “baseOp\_temp” in the bottom-right corner and identity elements in the top-left corner.

If “ControlQubit” equals 1, the control qubit is the second qubit, and the first qubit is the target qubit. The operator is created as a  $4 \times 4$  matrix with the elements of “baseOp\_temp”

at positions (2,2), (2,4), (4,2), and (4,4), and identity elements in the remaining positions of the main diagonal, as in (86):

$$op_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & U_{11} & 0 & U_{12} \\ 0 & 0 & 0 & 0 \\ 0 & U_{21} & 0 & U_{22} \end{bmatrix} \quad (86)$$

The “op\_1” matrix is obtained by left and right multiplying the “op\_0” matrix by the SWAP matrix [35].

The implementation of these two types of controlled operators is not provided in the original “getOperator” function, which only allows for the case where the control qubit is the first qubit.

If the “ControlQubit” flag is neither 0 nor 1, the controlled operator is set as a  $4 \times 4$  matrix of zeros to immediately detect a problem. Finally, the “op” matrix is returned as the output of the function.

The functions, **matrix\_complex\_mults.m** and **Apply\_operator.m** are as for the single-qubit case with the introduction of the only change in reshaping parameters from (2,2) to (4,4).

#### probFinal\_twoQubit.m

```

function [probFinal,probQubit0,
probQubit1]=probFinal_twoQubit(computerState)
fm = fimath('RoundingMethod', 'Nearest', ...
'OverflowAction', 'Saturate', ...
'ProductMode', 'FullPrecision', ...
'SumMode', 'FullPrecision');
nt = numerictype(0, 16, 16);
prob = fi([0; 0; 0; 0],nt,fm);
probFinal = fi([0; 0; 0; 0],nt,fm);
probQubit0 = fi([0; 0],nt,fm);
probQubit1 = fi([0; 0],nt,fm);
abs_value = cordicabs(computerState,15);
prob(:) = abs_value.*abs_value;
probQubit0(:) = [prob(1,1)+prob(2,1);
prob(3,1)+prob(4,1)];
probQubit1(:) = [prob(1,1)+prob(3,1);
prob(2,1)+prob(4,1)];
probFinal(:) = prob;
end

```

The function “probFinal\_twoQubit” calculates the probability distribution of a two-qubit quantum system. The function requires the complex vector “computerState,” which describes the quantum state following the application of the algorithm.

Next, the function calculates the probabilities associated with three possible measurement outcomes using the “cordicabs” function and performs element-wise multiplication using the notation “.\*,” similar to what was done for a single-qubit system.

Subsequently, the function calculates the probability distribution of measuring a basis state ( $|0\rangle$  or  $|1\rangle$ ) for each individual qubit by summing the probabilities associated with the basis states of the quantum register ( $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$ ,  $|11\rangle$ ) that contain that state for the individual qubit.

For example, to calculate the probability of measuring 0 for the first qubit, are summed the probabilities of obtaining the basis states  $|0\rangle \otimes |x\rangle$ , where  $|x\rangle$  is any state of the second qubit. Similarly, to calculate the probability of measuring 1 for the first qubit, are summed the probabilities of obtaining the basis states  $|1\rangle \otimes |y\rangle$ , where  $|y\rangle$  is another state of the second qubit.

These distributions are stored in the vectors “probQubit0” for the first qubit and “probQubit1” for the second qubit. Finally, the function also returns the overall probability distribution of the two-qubit quantum system which is saved in the “probFinal” vector.

### Testbench files

To test the validity of the work, it was necessary to write three testbench files, two in MATLAB and one directly in Verilog HDL, for each of the two projects:

- MATLAB testbench file to be applied to the original MATLAB project.
- MATLAB testbench file to be applied to the MATLAB project to be translated (from simulator to emulator).
- Verilog HDL testbench file to be applied to the Quartus project of the emulator.

In this section are described the two MATLAB testbench files, and in the next section will be analysed the Verilog ones.

For the single-qubit project, the MATLAB code of the testbench to be applied to the original MATLAB project is:

```
initState = [0.772790905621241+0.052860346093275i;
            0.631669339981830+0.031525306135226i];
qAlgorithm = ["RY|-pi/2" "RX|pi/4" "U3|pi/6|-pi/2|-
pi/9"];
[computerState, probFinal, algMatrix] =
quantumComputer(qAlgorithm, 1, initState);
computerState_fi = fi(computerState,1,16,15);
computerState_bin = bin(computerState_fi);
probFinal_fi = fi(probFinal,1,16,15);
probFinal_bin = bin(probFinal_fi);
```

“initState” is the ket associated with the qubit in the initial state. It is emphasized that it is important to ensure that the complex vector satisfies the normalization condition; otherwise, it would not represent a valid quantum system.

The operators specified in this script are three and require the  $R_y(-\frac{\pi}{2})$ ,  $R_x(\frac{\pi}{4})$  and  $U_3(\frac{\pi}{6}, -\frac{\pi}{2}, -\frac{\pi}{9})$ , operators, in that order.

The additional lines of code are not strictly necessary but can be useful for directly comparing the values obtained, both in fractional and binary form, with those of the emulator.

The code of the testbench file to be applied to the single-qubit MATLAB project (from simulator to emulator) is:

### QuantumComputer\_oneQubit\_tb.m

```
fm = fimath('RoundingMethod', 'Nearest', ...
           'OverflowAction', 'Saturate', ...
           'ProductMode', 'SpecifyPrecision', ...
           'ProductWordLength', 16, ...
           'ProductFractionLength', 15, ...
           'SumMode', 'SpecifyPrecision', ...
           'SumWordLength', 16, ...
           'SumFractionLength', 15, ...
           'CastBeforeSum', true);

nt1 = numerictype(1, 16, 15);
initState = complex(fi([0.772790905621241;
0.631669339981830],nt1,fm),fi([0.052860346093275;
0.031525306135226],nt1,fm));
initState_bin = bin(initState);
opName = fi([13;12;11],0,16,0);
nt = numerictype(1, 16, 13);
template = fi([],nt,fm);
theta = cast([0; 0; 0], 'like', template);
theta(:) = [-pi/2; pi/4; pi/6];
theta_bin = bin(theta);
phi = cast([0; 0; 0], 'like', template);
phi(:) = [pi/4; -pi/2; -pi/2];
phi_bin = bin(phi);
lambda = cast([0; 0; 0], 'like', template);
lambda(:) = [-pi/9; pi/3; -pi/9];
lambda_bin = bin(lambda);
[computerState_my, probFinal_my] =
quantumComputer_oneQubit(opName,theta,phi,lambda,ini
tState);
computerState_bin_my = bin(computerState_my);
probFinal_bin_my = bin(probFinal_my);
```

This testbench file performs the same functions as the previous script and is used to test the quantum computing simulator for a single qubit. The “quantumComputer\_oneQubit” function is used to execute the quantum algorithm. The file specifies the input parameters for the function, including the initial state of the qubit, the names of the quantum operators to apply, and the parameterization angles for each operator.

The “fimath” and “numerictype” objects are defined with the same settings applied in the “getOperator” function.

The initial state of the qubit and the parameterization angles for the quantum operators are converted into binary numbers using the “bin” function. The “quantumComputer\_oneQubit” function is then executed using the specified input parameters, and the result of the quantum computer’s state and the final probability are saved in “computerState\_my” and “probFinal\_my,” respectively. These

results are also converted into binary numbers for later visualization.

For the two-qubit project, the MATLAB code of the testbench to be applied to the original MATLAB project is:

```
initState=initializeRegisters([0.772790905621241+0.
052860346093275i
0.631669339981830+0.031525306135226i], ...
[0.772790905621241+0.052860346093275i, ...
0.631669339981830+0.031525306135226i]);
qAlgorithm = ["Z" "RX|pi/4" ".";"RY|-pi/2"
"RX|pi/4" "U3|pi/6|-pi/2|-pi/9"];
[computerState, probFinal, algMatrix] =
quantumComputer(qAlgorithm, [1 2], initState);
computerState_fi = fi(computerState,1,16,15);
computerState_bin = bin(computerState_fi);
```

This testbench initializes the initial state of the qubits using the “initializeRegisters” function. The value of the initial state is represented as a complex  $2 \times 2$  matrix. The variable “qAlgorithm” contains a series of quantum operations to be performed on the quantum system, with three operators planned for each qubit.

For the first qubit, the following operators are applied in this order:  $Z$ ,  $R_x\left(\frac{\pi}{4}\right)$  and “•”; the “•” operator indicates the use of a controlled gate, with the first qubit acting as the control qubit.

For the second qubit, the operators applied are  $R_y\left(-\frac{\pi}{2}\right)$ ,  $R_x\left(\frac{\pi}{4}\right)$  and  $U_3\left(\frac{\pi}{6}, -\frac{\pi}{2}, -\frac{\pi}{9}\right)$ . Schematically, it is represented as in (87) and (88):

$$|initState_0\rangle : -- Z --- R_x\left(\frac{\pi}{4}\right) --- \bullet --- \quad (87)$$

$$|initState_1\rangle : -- R_y\left(-\frac{\pi}{2}\right) --- R_x\left(\frac{\pi}{4}\right) --- \\ -- U_3\left(\frac{\pi}{6}, -\frac{\pi}{2}, -\frac{\pi}{9}\right) --- \quad (88)$$

This array is passed to the “quantumComputer” function, along with a matrix of strings representing the quantum algorithm to be executed on the quantum system. The result of the “quantumComputer” function is the final quantum state ket, the probability of measuring the basis states after measuring both qubits, and the quantum transformation matrix representing the effect of the quantum operations on the quantum system.

Finally, the final quantum state vector is converted into a “fi” data type and expressed as a binary string for easy visualization and comparison.

The code of the testbench file to be applied to the two-qubit MATLAB project (from simulator to emulator) is:

## QuantumComputer\_twoQubit\_tb.m

```
fm = fimath('RoundingMethod', 'Nearest', ...
'OverflowAction', 'Saturate', ...
'ProductMode', 'FullPrecision', ...
'SumMode', 'FullPrecision');
nt1 = numerictype(1, 16, 15); % numerictype
associato agli stati quantici
nt = numerictype(1, 16, 13); % numerictype
associato agli angoli
template = fi([],nt,fm);
theta_0 = cast([0; 0; 0],'like',template);
phi_0 = cast([0; 0; 0],'like',template);
lambda_0 = cast([0; 0; 0],'like',template);
theta_1 = cast([0; 0; 0],'like',template);
phi_1 = cast([0; 0; 0],'like',template);
lambda_1 = cast([0; 0; 0],'like',template);
initState_0 = complex(fi([0.772790905621241;
0.631669339981830],nt1,fm),
fi([0.052860346093275;
0.031525306135226],nt1,fm));
initState_0_bin = bin(initState_0);
initState_1 = complex(fi([0.772790905621241;
0.631669339981830],nt1,fm),
fi([0.052860346093275;
0.031525306135226],nt1,fm));
initState_1_bin = bin(initState_1);
opName_0 = fi([3;12;15],0,16,0);
opName_0_bin = bin(opName_0);
theta_0(:) = [0; pi/4; 0];
theta_0_bin = bin(theta_0);
phi_0(:) = [0; 0; 0];
phi_0_bin = bin(phi_0);
lambda_0(:) = [0; 0; 0];
lambda_0_bin = bin(lambda_0);
opName_1 = fi([13;12;11],0,16,0);
opName_1_bin = bin(opName_1);
theta_1(:) = [-pi/2; pi/4; pi/6];
theta_1_bin = bin(theta_1);
phi_1(:) = [0; 0; -pi/2];
phi_1_bin = bin(phi_1);
lambda_1(:) = [0; 0; -pi/9];
lambda_1_bin = bin(lambda_1);
[computerState_my,probFinal_my,probQubit_0_my,probQubit_1_my] =
quantumComputer_twoQubit(opName_0,theta_0,phi_0,lambda_0,opName_1,theta_1,phi_1,lambda_1,initState_0,initState_1);
computerState_bin_my = bin(computerState_my);
```



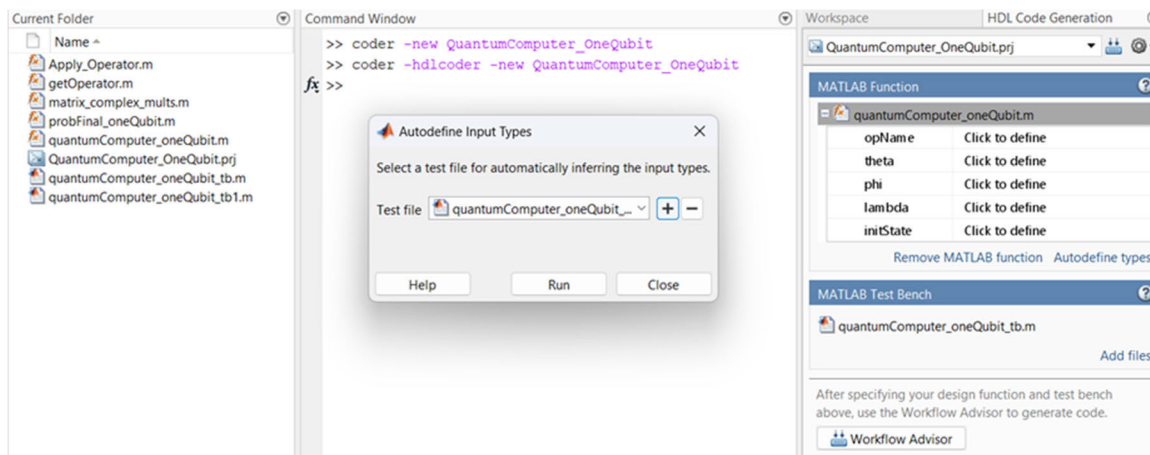


FIGURE 8. Input variables definition of the testbench file.

It calls the “quantumComputer\_twoQubit” function to simulate the evolution of a two-qubit quantum system based on the quantum operations applied to the two qubits.

The testbench defines the following variables:

fm: a fimath object defining the numeric rules.

nt1: a numerictype object with 16 bits of word length (WL), signed, and 15 bits of fraction length (FL), used to represent quantum states.

nt: a numerictype object with 16 bits of WL, signed, and 13 bits of FL, used to represent angles.

template: a template for all angles for operator parameterization.

theta\_0, phi\_0, lambda\_0: three arrays of fixed-point numbers representing the angles used for the quantum operation applied to the first qubit.

theta\_1, phi\_1, lambda\_1: three arrays of fixed-point numbers representing the angles used for the quantum operation applied to the second qubit.

initState\_0, initState\_1: two arrays of complex fixed-point numbers representing the initial quantum states of the two qubits.

opName\_0, opName\_1: two arrays of fixed-point numbers representing the quantum operations to be applied to the two qubits.

The testbench then calls the “quantumComputer\_twoQubit” function, passing the values of the variables described above as arguments. The function returns the following outputs:

computerState\_my: an array of fixed-point complex numbers representing the final quantum state of the two-qubit system.

probFinal\_my: an array of fixed-point numbers representing the total probability of the two-qubit system.

probQubit\_0\_my: an array of fixed-point numbers representing the probability of the first qubit.

probQubit\_1\_my: an array of fixed-point numbers representing the probability of the second qubit.

All variables are also expressed in binary format for comparison with values obtained from the Quartus simulation.

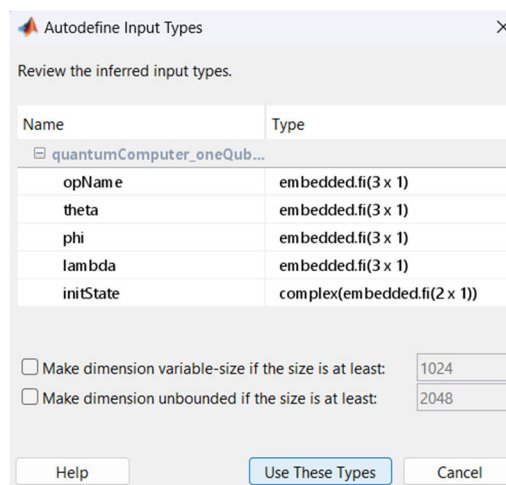


FIGURE 9. Types definition.

### E. TRANSLATION OF MATLAB PRIMARY FUNCTIONS INTO VERILOG HDL CODE

The general procedure to translate MATLAB functions into Verilog HDL listings has been outlined previously. In this section, the procedure applied to the generation of the quantum computing emulators is described, starting with the single-qubit and two-qubit MATLAB simulator projects, which were detailed earlier.

The steps for the single-qubit emulator are as follows.

Add Intel Quartus Lite Edition to the system path using the hdlsetuptoolpath function:

```
hdlsetuptoolpath('ToolName', 'Altera Quartus II',
'ToolPath',
'...\intelFPGA_lite\19.1\quartus\bin64\quartus.exe')
```

Create a new HDL Coder project:

```
coder -hdlcoder -new QuantumComputer_oneQubit
```

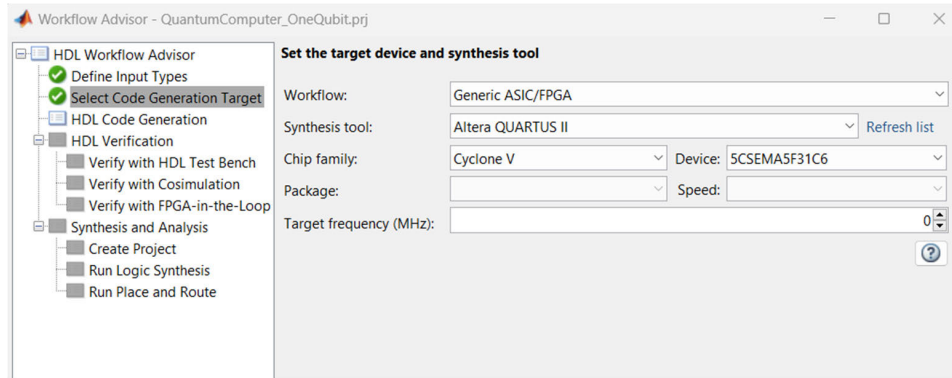


FIGURE 10. Synthesis tool and target FPGA definition in workflow advisor.

Click on “Add MATLAB Function” and select the primary function QuantumComputer\_oneQubit.m.

Click on “Autodefine types” and add the testbench file by clicking the “+” next to the “Test File” entry and selecting the file QuantumComputer\_oneQubit\_tb.m.

Then, click “Run”.

These operations allow the definition of input variables by executing the testbench file (Fig. 8).

Then, click on “Use These types”, as in Fig. 9, and then click on “Workflow Advisor”.

It is worth of notice that in workflow advisor it is declared do not to proceed with automatic fixed-point conversion by setting “Keep Original Types”.

The next task involves defining the synthesis tool and the target FPGA board, as in Fig. 10.

The final step involves specifying all the settings for HDL Code Generation as follows:

Target -> Target Selection -> Language -> Select “Verilog”;

Target -> Output Settings -> Check “Check HDL Conformance”;

Coding Style -> Generated Code comments -> Check “Include MATLAB source code as comments”;

Coding Style -> Generated Code comments -> Check “Generate report”;

Coding Standards -> Choose coding standard -> HDL Coding Standards -> Select “Industry”;

Optimizations -> Pipeling -> Check “Register Inputs”;

Optimizations -> Pipeling -> Check “Register Outputs”;

Optimizations -> Pipeling -> Check “Adaptive Pipelining”;

Advanced -> Advanced Coding Options -> Check “Generate instantiable code for functions”;

Advanced -> Diagnostics -> Check for presence of reals in the generated HDL Code -> Select “Error”.

Choosing to enforce code writing standards based on industrial recommendations is advisable to enhance the readability of Verilog code listings. This approach sets limits on the length of variable names and promotes the use of standard names for typical I/O signals such as clock and reset.

It is important to check the introduction of registers on input and output to induce the HDL Coder toolbox to insert clock enable signals on input and output, as explained in the section describing the Quartus project.

Once the task settings are defined, clicking on “Run” are displayed in the log window hyperlinks to the generated Verilog files and requested reports.

Regarding the generation of Verilog listings for the two-qubit emulator, there are no substantial differences from what was shown for the single-qubit case, except for the definition of the synthesis tool and target FPGA board (Fig. 11). The selected synthesis tool is Intel Quartus Pro Edition that supports the Arria 10 FPGA family.

The related MATLAB command is:

```
hdlsetuptoolpath('ToolName', 'Intel Quartus Pro',
'ToolPath', '
...\\intelFPGA_pro\22.3\quartus\bin64\quartus.exe')
```

The final step in the MATLAB environment project involves writing two functions that, given all the input values required by the QuantumComputer\_oneQubit and QuantumComputer\_twoQubit scripts, write Memory Initialization File (MIF) files for the FPGA implementation, used to initialize the memory of an FPGA.

These files are used to define the initial content of a memory, such as RAM or ROM, before the board’s logic is executed. MIF files are very useful when it’s necessary to specify a specific starting behaviour for the memory, such as ensuring that a memory holds a specific value at boot-up. These files are written in a text-based format and consist of lines that describe the position of each bit in memory and the initial value of each bit.

In Quartus environment, MIF files can be used to initialize memory in various ways, such as manually entering values or automatically generating them using third-party software. In this work, MATLAB was used for this purpose [36].

Before running either of the two MIF file generation functions, it is necessary to execute the corresponding testbench file (testbench for the emulator). This ensures that the required fixed-point input values are available to the function.

It follows the code for the MIF file generation:

```
function values =
generazione_file_mif_hex(opName,theta,phi,lambda,ini
tState)
    opName_hex = [opName.hex(1,:); opName.hex(2,:);
opName.hex(3,:)];
    theta_hex = [theta.hex(1,:); theta.hex(2,:);
theta.hex(3,:)];
    phi_hex = [phi.hex(1,:); phi.hex(2,:);
phi.hex(3,:)];
    lambda_hex = [lambda.hex(1,:); lambda.hex(2,:);
lambda.hex(3,:)];
    initState_real = real(initState);
    initState_imag = imag(initState);
    initState_real_hex = [initState_real.hex(1,:);
initState_real.hex(2,:)];
    initState_imag_hex = [initState_imag.hex(1,:);
initState_imag.hex(2,:)];
    % Generate Parameters
    ram_size = 16; % In words
    word_size = 16; % Can be 8, 16, 32, etc.

    values = upper([opName_hex; theta_hex; phi_hex;
lambda_hex;
initState_real_hex;
initState_imag_hex])
    % Open mif file for write
    fileID = fopen('init_mem.mif', 'w');
    fprintf( fileID, '%s%d%s\n', 'WIDTH=', word_size,
');
    fprintf( fileID, '%s%d%s\n\n', 'DEPTH=', ram_size,
');
    fprintf( fileID, '%s\n', 'ADDRESS_RADIX=UNS;');
    fprintf( fileID, '%s\n\n', 'DATA_RADIX=HEX;');
    fprintf( fileID, '%s\n', 'CONTENT BEGIN');
    % write values to file
    idx = 0;
    for ii=1:ram_size
        fprintf( fileID, ' %u : %s;\n', idx,
values(idx+1,1:4));
        idx = idx+1;
    end
    fprintf( fileID, '%s\n', 'END;');
    fclose( fileID);
    % Convert to HEX
    cmd =
'...\intelFPGA_lite\19.1\quartus\bin64\mif2hex
init_mem.mif
init_mem.hex';
    system(cmd);
end
```

The function also creates a second file, the HEX file, which is a memory initialization file alternative to the MIF file, whose values are written in hexadecimal format.

For completeness, the script for generating the MIF file for the two-qubit emulator is provided below, also. The only

significant difference is that the RAM size is 32 words, compared to the 16 words in the previous case.

```
function values =
generazione_file_mif_hex(opName_0,theta_0,phi_0,lamb
da_0,opName_1,theta_1,phi_1,lambda_1,initState_0,ini
tState_1)
    opName_0_hex = [opName_0.hex(1,:);
opName_0.hex(2,:); opName_0.hex(3,:)];
    theta_0_hex = [theta_0.hex(1,:); theta_0.hex(2,:);
theta_0.hex(3,:)];
    phi_0_hex = [phi_0.hex(1,:); phi_0.hex(2,:);
phi_0.hex(3,:)];
    lambda_0_hex = [lambda_0.hex(1,:);
lambda_0.hex(2,:); lambda_0.hex(3,:)];
    opName_1_hex = [opName_1.hex(1,:);
opName_1.hex(2,:); opName_1.hex(3,:)];
    theta_1_hex = [theta_1.hex(1,:); theta_1.hex(2,:);
theta_1.hex(3,:)];
    phi_1_hex = [phi_1.hex(1,:); phi_1.hex(2,:);
phi_1.hex(3,:)];
    lambda_1_hex = [lambda_1.hex(1,:);
lambda_1.hex(2,:); lambda_1.hex(3,:)];
    initState_0_real = real(initState_0);
    initState_0_imag = imag(initState_0);
    initState_1_real = real(initState_1);
    initState_1_imag = imag(initState_1);
    initState_0_real_hex =
[initState_0_real.hex(1,:);initState_0_real.hex(2,:)
];
    initState_0_imag_hex =
[initState_0_imag.hex(1,:);initState_0_imag.hex(2,:)]
;
    initState_1_real_hex =
[initState_1_real.hex(1,:);initState_1_real.hex(2,:)
];
    initState_1_imag_hex =
[initState_1_imag.hex(1,:);initState_1_imag.hex(2,:)]
];
    % Generate Parameters
    ram_size = 32; % In words
    word_size = 16; % Can be 8, 16, 32, etc.
    % generate vector
    values = upper([opName_0_hex; theta_0_hex;
phi_0_hex; lambda_0_hex; opName_1_hex; theta_1_hex;
phi_1_hex; lambda_1_hex; initState_0_real_hex;
initState_0_imag_hex; initState_1_real_hex;
initState_1_imag_hex])
    % Open mif file for write
    fileID = fopen('init_mem.mif', 'w');
    fprintf( fileID, '%s%d%s\n', 'WIDTH=', word_size,
');
```

```

fprintf( fileID, '%s%d%s\n\n', 'DEPTH=', ram_size,
':'):
fprintf( fileID, '%s\n', 'ADDRESS_RADIX=UNS;');
fprintf( fileID, '%s\n\n', 'DATA_RADIX=HEX;');
fprintf( fileID, '%s\n', 'CONTENT BEGIN');
% write values to file
idx = 0;
for ii=1:ram_size
    fprintf( fileID, ' %u : %s;\n', idx,
values(idx+1,1:4));
    idx = idx+1;
end
fprintf( fileID, '%s\n', 'END;');
fclose( fileID);
% Convert to HEX
cmd =
'...\intelFPGA_lite\19.1\quartus\bin64\mif2hex
init_mem.mif init_mem.hex';
system(cmd);
end

```

## V. HARDWARE IMPLEMENTATION

The hardware implementation was carried out using INTEL's FPGA, mounted on Terasic boards. The development environment used was Intel's Quartus IDE.

To implement the single-qubit emulator, the FPGA 5CSEMA5F31C6 from the Cyclone V family, mounted on the Terasic DE1\_SoC board, was utilized [37].

To implement the two-qubit emulator, the FPGA 10AX115N2F45E1SG from the Arria 10 GX family, mounted on the Terasic DE5a-NET DDR4 board [38], was used because the resources of the Cyclone V were insufficient.

In both cases, the files to be used in their respective Quartus projects (for the single-qubit emulator and for the two-qubit emulator) are the ones obtained through the code translation procedure performed starting by the respective MATLAB simulators.

To complete the projects for both emulators, the remaining task is to implement the input loading for initializing computations, as described below.

The data loading system comprises three interconnected blocks: a modulo-15 counter, a Serial-In Parallel-Out (SIPO) register, and a ROM memory block to be instantiated on the FPGA board. The configuration parameters for the ROM memory are shown in Fig. 12.

The SIPO register is designed with dimensions matching those of the ROM for seamless memory data loading. After loading, an additional clock cycle activates all data buses in parallel at the register's output. The register requires a "load" pin for initializing data loading and features a 16-bit data bus, resulting in 16 data buses, each carrying 16 bits. Furthermore, it includes a "done" output pin that signals when the register has been loaded, simultaneously halting the loading process, and making the data buses available at the output.

For each of the three modules, Block Symbol Files (BSFs) have been generated from the Verilog listings. These BSF files serve as symbolic blocks, facilitating a diagrammatic system design and freeing it from code-based programming.

## A. SINGLE QUBIT EMULATOR HARDWARE IMPLEMENTATION

The complete design of the single-qubit emulator is illustrated in Fig. 13. The schematic file serves as the Top-Level Entity of the project, representing the overall architecture of the system, its interconnections, and acting as the starting point for synthesis by the Quartus IDE.

As depicted in Fig. 13, the "done" output signal from the SIPO register functions as the clock enable signal for the block related to the actual quantum algorithm. When the data from the register becomes available at the output, it is stored in the input registers of the module responsible for quantum computation on the subsequent clock cycle. Following another clock cycle, the final data is accessible at the output of the entire system.

The significance of introducing input and output registers during the MATLAB code translation phase is now clarified: it is essential for enabling synchronization between the computation block and the data loading system.

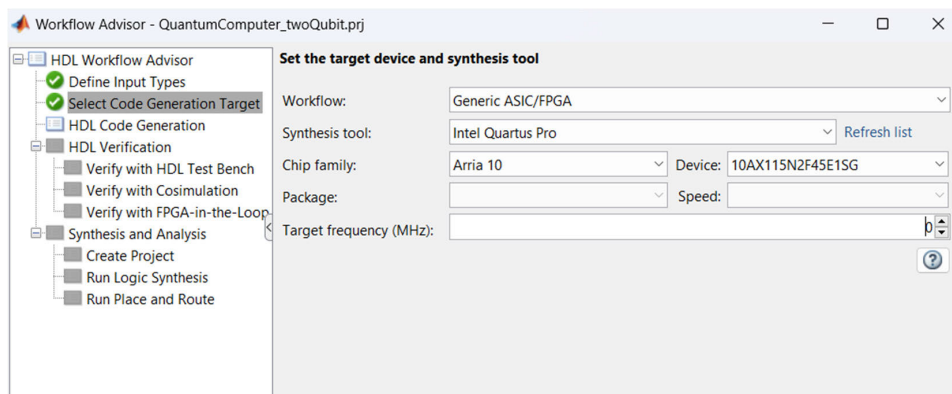


FIGURE 11. Workflow settings for the 2-qubit emulator.



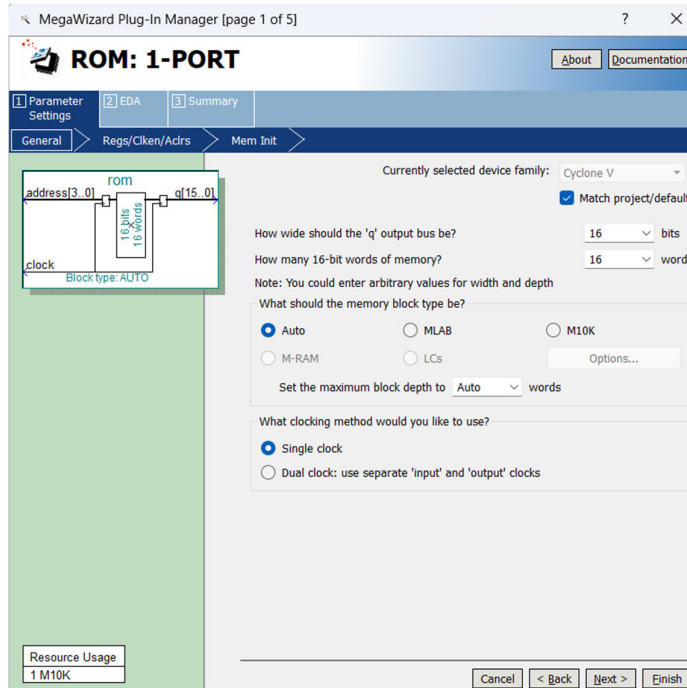


FIGURE 12. ROM configuration screen.

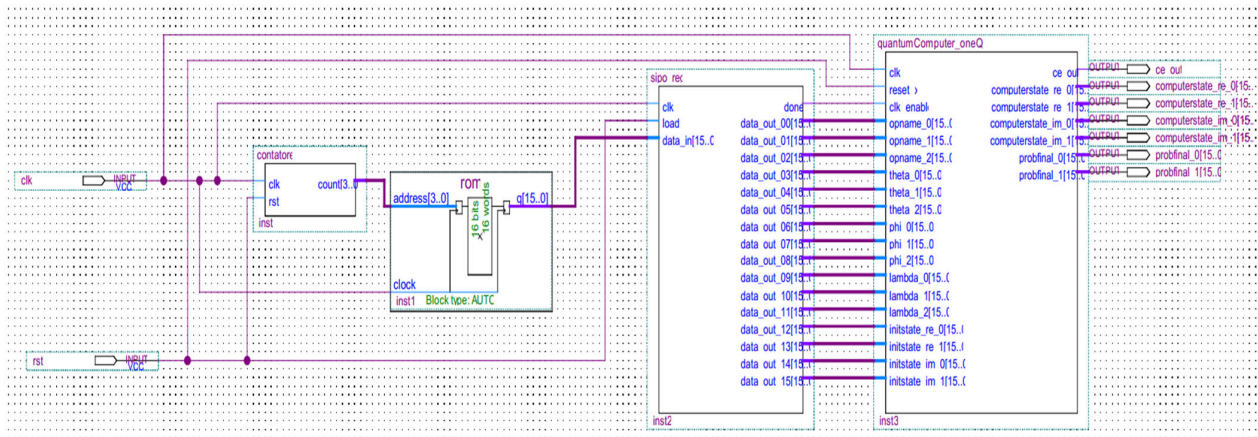


FIGURE 13. Schematic of the full design.

**B. TWO-QUBIT EMULATOR HARDWARE IMPLEMENTATION**

As mentioned earlier, the Pro version of the Quartus IDE is utilized for the two-qubit emulator. The procedures are analogous to those discussed previously: create the project, add the generated Verilog listings, and define the module that invokes all the components.

It is noteworthy that the two-qubit computational block requires 32 data buses as input. Consequently, in this system, it is necessary to adjust the data loading block: the “modulo 16” counter is replaced by a “modulo 32” counter, and similarly, the size of the ROM is increased to 32 words from the previous 16. No other differences in hardware description are observed at this level.

The testbenches for both hardware implementations were created using the waveform editor tool in the Quartus IDE, accessible through the ‘File’ > ‘New’ > ‘University Program WVF’ option in the main menu.

The testbench file serves as a graphical interface where input waveforms are easily drawn, and upon completing the calculations, the corresponding output waveforms are displayed, as shown in Figs. 14 and 15.

Users also have the option to select the radix for visualizing results, choosing between fractional (decimal) and binary. When comparing results obtained from MATLAB simulators and FPGA emulators, selecting the fractional view of the output proves to be useful.



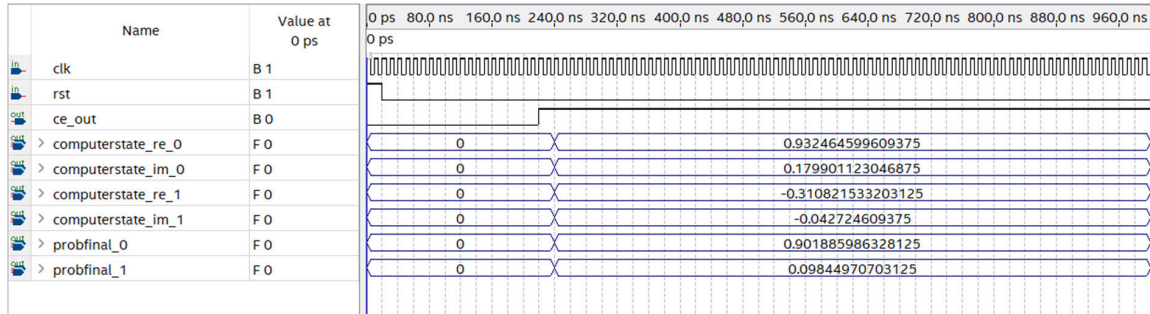


FIGURE 14. Calculation results of a 1-qubit emulator.

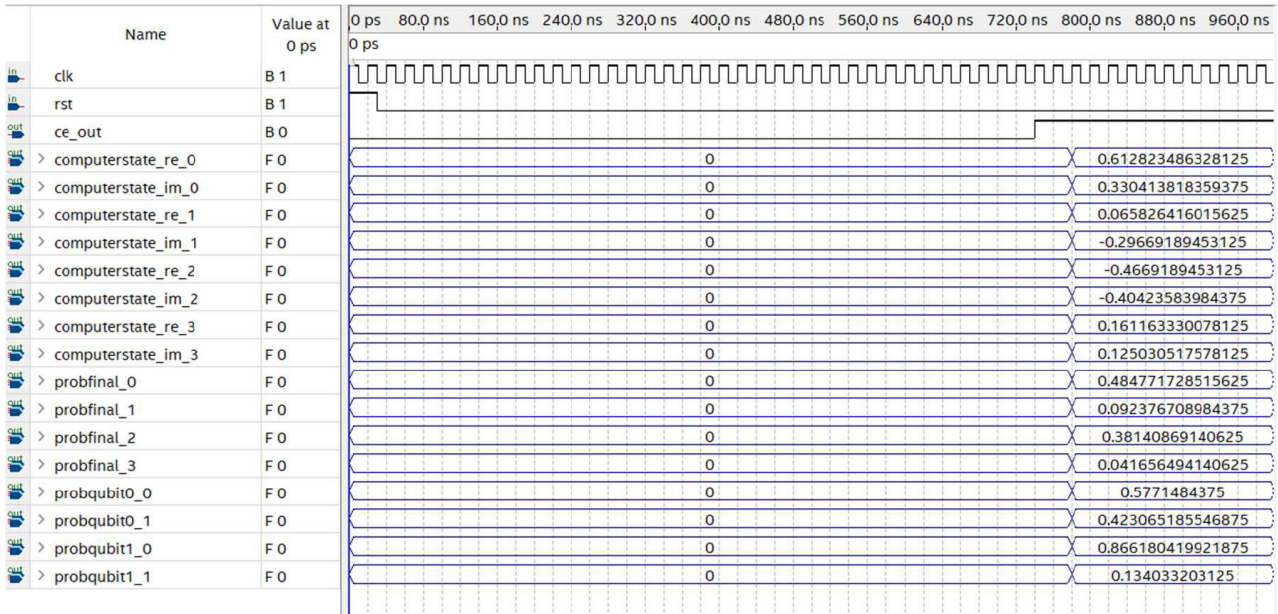


FIGURE 15. Calculation results of a 2-qubit emulator.

VI. RESULTS AND DISCUSSION

In Figs. 14 and 15 the calculation results by FPGA are shown for the single-qubit and two-qubit emulator, respectively, and they match with the MATLAB calculations, as shown in table 1 and 2. Table 3 summarizes the FPGA resource consumption for both implementations.

Anyway, a further discussion is useful to properly understand the emulators performances and the comparison with the simulator performances.

In a fixed-point system, the amount of information that can be represented is limited to a fixed number of significant digits. This means that values that cannot be exactly represented are rounded or truncated. This can lead to approximation errors in mathematical operations. Additionally, the fixed-point format can also lead to overflow or underflow, depending on the properties of the selected format. These errors can have significant consequences on the accuracy and correctness of results [39], [40], [41].

To avoid these errors, it is crucial to carefully select appropriate values for the parameters “wordlength” and “fractionlength” of the MATLAB “fi” format to ensure reliable data representation. Moreover, it is essential to verify that

intermediate and final values of calculations fall within the range supported by the format and that quantization errors are acceptable for the specific requirements of the application.

The precision of a fixed-point number is the difference between successive values representable by its data type and scale. The value of the least significant bit, and hence the precision of the number, is determined by the number of bits used to represent the fractional part. More bits used to represent the fractional part result in greater precision.

Additionally, consider the error introduced using CORDIC algorithms. Although of lesser magnitude, additional error is inevitable. The total error is comparable to the precision associated with the chosen fixed-point representation, due to the use of the maximum available iterations for fixed-point numbers with a word length of 16.

In the two designed emulators, the mathematical properties imposed differ based on the capabilities of the two FPGAs used for the respective hardware implementation.

In the case of the single-qubit emulator, results of each mathematical operation are cast, introducing additional rounding at each mathematical step. However, in the case of the two-qubit emulator, the casting is deferred until the

**TABLE 1.** Comparisons between results obtained from the single-qubit quantum computing simulator and the corresponding emulator.

	MATLAB SIMULATOR	FPGA EMULATOR	ERROR (MATLAB – FPGA)
<b>computerState(0)</b>	0.932412319455610 + 0.179816057079382i	0.932464599609375 + 0.179901123046875i	- 0.000052280153765 - 0.000085065967493i
<b>computerState(1)</b>	-0.310541942587615 - 0.042862034923031i	-0.310821533203125 - 0.042724609375000i	0.00027959061551 - 0.000137425548031i
<b>probFinal(0)</b>	0.901726547856166	0.901885986328125	-0.000159438471959
<b>probFinal(1)</b>	0.098273452143833	0.098464965820312	-0.000191513676479

**TABLE 2.** Comparisons between results obtained from the two-qubit quantum computing simulator and the corresponding emulator.

	MATLAB SIMULATOR	FPGA EMULATOR	ERROR (MATLAB – FPGA)
<b>computerState(0)</b>	0.612833800587840 + 0.330416875237365i	0.612701416015625 + 0.330474853515625i	0.000132384572215 - 0.00005797827826i
<b>computerState(1)</b>	0.065795622960353 - 0.296689479248321i	0.065917968750000 - 0.296752929687500i	- 0.000122345789647 + 0.000063450439179i
<b>computerState(2)</b>	-0.466866543252131 - 0.404204078437576i	-0.466796875000000 - 0.404327392578125i	- 0.000069668252131 + 0.000123314140549i
<b>computerState(3)</b>	0.161022007462616 + 0.125029265362173i	0.161224365234375 + 0.125061035156250i	- 0.000202357771759 - 0.000031769794077i
<b>probFinal(0)</b>	0.484740578584561	0.484603881835938	0.000136696748623
<b>probFinal(1)</b>	0.092353711097381	0.092422485351562	-0.000068774254181
<b>probFinal(2)</b>	0.381345306233764	0.381408691406250	-0.000063385172486
<b>probFinal(3)</b>	0.381345306233764	0.381408691406250	-0.000063385172486
<b>probFinal(4)</b>	0.041560404084296	0.041625976562500	-0.000065572478204
<b>probQubit_0(0)</b>	0.577094289681941	0.577026367187500	0.000067922494441
<b>probQubit_0(1)</b>	0.422905710318059	0.423034667968750	-0.000128957650691
<b>probQubit_1(0)</b>	0.866085884818324	0.866012573242188	0.000073311576136
<b>probQubit_1(1)</b>	0.133914115181676	0.134048461914062	-0.000134346732386

**TABLE 3.** Resources consumption for hardware implementation.

FPGA	SINGLE QUBIT EMULATOR	2-QUBIT EMULATOR
Device Family	Cyclone V	Arria 10
Device	5CSEMA5F31C6	10AX115N2F45E1SG
Logic Elements (LE) (in ALM)	18991/32070 (59%)	31305/427200 (7%)
Registers	812	1805
I/O Pin	99/457 (22%)	259/992 (26%)
Memory consumption (bits)	256/ 4065280 (< 1%)	512/55562240 (< 1%)
DSP	87/87 (100 %)	1080/1518 (71%)

conclusion of mathematical operations, preserving the maximum precision available in the intermediate calculation phases.

Precision is always lost during the rounding operation, leading to quantization errors and computational noise. The cost of the rounding operation and the amount of bias introduced depend on the rounding method used.

Various rounding methods exist, each with its set of inherent properties. Depending on the requirements of the design, these properties can make a rounding method desirable. Understanding the design requirements and the properties of each rounding method can help determine which one is best suited to real needs. The most important properties to consider are cost, bias, and the potential for overflow. The supported rounding methods are ceiling, convergent, floor, nearest, round, simplest, and zero. As previously mentioned, the rounding methods used for both cases are “nearest.”

Based on what has been discussed so far, by the Tables 1 and 2 comparing the results obtained from each MATLAB simulator and the corresponding FPGA emulator it can be observed the effect of error propagation introduced in each calculation stage, resulting in a maximum error on the fourth digit after the decimal point and in a minimum error on the fifth digit after the decimal point. It can be concluded that the errors introduced by the previously listed approximations do not significantly affect the validity of the results obtained, as they are on the order of  $10^{-4} - 10^{-5}$ .

Anyway, it is worth of notice that the main error is particularly evident in the calculation of the probability distribution.

Although the accuracy of the results obtained from fixed-point arithmetic processing from FPGAs is widely acceptable compared to the results obtained from MATLAB, it is believed that the accuracy can be further improved by intervening on the “cordicabs” instruction in MATLAB that is inefficient and introduces significant rounding error. Therefore, it is necessary to investigate alternative instructions that incur lower computational cost (which is high in this case) and are more accurate.

## VII. CONCLUSION

In this paper, has been described step by step the procedure for developing a quantum computing emulator implemented

on an FPGA, starting from a quantum computing simulator developed in the MATLAB environment, for the two cases of a single-qubit operation and for a two-qubit operation, respectively.

The advantages of this design approach lie in the simplicity of the project when following the “model-based” approach in the MATLAB environment, as opposed to the hardware project executed directly in an HDL language. Furthermore, it involves the creation of a highly optimized HDL code for FPGA implementation by the MATLAB HDL Coder toolbox. This level of optimization is much more challenging to achieve directly in HDL and demands uncommon design skills. With the method devised and described in this paper, optimized HDL code is obtained more easily.

Although the MATLAB HDL Coder toolbox automatically translates MATLAB code into HDL code, this translation is not applicable to all MATLAB functions and fails if applied to a quantum computing simulator if not properly designed. Therefore, this paper deeply examines and describes project rules to be applied during the simulator’s project phase, focusing on transitioning from floating-point arithmetic to fixed-point arithmetic and code translation within MATLAB. This ensures error-free and reliable code translation, resulting in a smooth and reliable design of a quantum computing emulator starting from a quantum computing simulator.

The procedure has been successfully applied and tested to create a single-qubit emulator and a two-qubit emulator, both implemented on Altera/Intel FPGAs. However, the procedure is suitable for FPGAs from any manufacturer.

The results obtained from comparing the calculations performed by the simulator and those executed by the emulator indicate that the procedure is entirely satisfactory. While the results are valid and satisfactory in terms of accuracy, further improvement could be attained by utilizing floating-point arithmetic instead of fixed-point arithmetic. However, this would necessitate significantly more FPGA resources.

## ACKNOWLEDGMENT

The author acknowledges Gaetano Losappio for his valuable support in MATLAB coding matter.

## REFERENCES

- [1] R. P. Feynman, "Simulating physics with computers," *Int. J. Theor. Phys.*, vol. 21, nos. 6–7, pp. 467–488, Jun. 1982.
- [2] *IBM Quantum Experience*. Accessed: Jan. 20, 2024. [Online]. Available: <https://quantumexperience.ng.bluemix.net/>
- [3] *Rigetti Quantum Forest*. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.rigetti.com/>
- [4] *D-Wave Systems*. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.dwavesys.com/>
- [5] F. Arute et al., "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019, doi: 10.1038/s41586-019-1666-5.
- [6] J. Preskill, "Quantum computing and the entanglement frontier," in *Proc. 5th Solvay Conf. Phys., Theory Quantum World*, Brussels, Belgium, Oct. 2011, pp. 1–19. Accessed: Mar. 16, 2024. [Online]. Available: <https://arxiv.org/pdf/1203.5813.pdf>
- [7] K. Hartnett. (Jun. 2019). *A New Law to Describe Quantum Computing's Rise?* Quanta Mag. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.quantamagazine.org/does-nevins-law-describe-quantum-computings-rise-20190618/>
- [8] B. M. Boghosian and W. Taylor, "Simulating quantum mechanics on a quantum computer," *Phys. D, Nonlinear Phenomena*, vol. 120, nos. 1–2, pp. 30–42, Sep. 1998, doi: 10.1016/S0167-2789(98)00042-6.
- [9] M. G. Jeannette. (2021). *How Quantum Computing Could Remake Chemistry*. Sci. Amer. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.scientificamerican.com/article/how-quantum-computing-could-remake-chemistry/>
- [10] *Quantum Computing MATLAB Toolbox*. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.mathworks.com/products/quantum-computing.html>
- [11] M. Aminian, M. Saeedi, M. S. Zamani, and M. Sedighi, "FPGA-based circuit model emulation of quantum algorithms," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Mar. 2008, pp. 399–404, doi: 10.1109/ISVLSI.2008.43.
- [12] A. U. Khalid, Z. Zilic, and K. Radecka, "FPGA emulation of quantum circuits," in *Proc. IEEE Int. Conf. Comput. Design, VLSI Comput. Processors*, Aug. 2004, pp. 310–315, doi: 10.1109/ICCD.2004.1347938.
- [13] A. U. Khalid, "FPGA emulation of quantum circuits," Dept. Elect. Comput. Eng., McGill Univ., Montreal, QC, Canada, Tech. Rep., 2005. Accessed: Mar. 16, 2024. [Online]. Available: <https://www.ece.mcgill.ca/~zzilic/thesis/KhalidThesisFinal.pdf>
- [14] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono, "An FPGA-based quantum circuit emulation framework using Heisenberg representation," *Int. J. Quantum Inf.*, vol. 16, no. 6, Sep. 2018, Art. no. 1850052, doi: 10.1142/S0219749918500521.
- [15] Y. Goto and M. Fujishima, "Efficient quantum computing emulation system with unitary macro-operations," *Jpn. J. Appl. Phys.*, vol. 46, no. 4S, p. 2278, Apr. 2007.
- [16] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono, "An FPGA-based quantum computing emulation framework based on serial-parallel architecture," *Int. J. Reconfigurable Comput.*, vol. 2016, pp. 1–18, 2016, doi: 10.1155/2016/5718124.
- [17] M. P. Frank, L. Oniciuc, U. H. Meyer-Baese, and I. Chiorescu, "A space-efficient quantum computer simulator suitable for high-speed FPGA implementation," *Proc. SPIE*, vol. 7342, Apr. 2009, Art. no. 734203.
- [18] G. Negovetic, M. Perkowski, M. Lukac, and A. Buller, "Evolving quantum circuits and an FPGA-based quantum computing emulator," in *Proc. Int. Workshop Boolean Problems*, 2002, pp. 1–9. Accessed: Mar. 16, 2024. [Online]. Available: <https://core.ac.uk/download/pdf/37771316.pdf>
- [19] M. Fujishima, K. Saito, M. Onouchi, and H. Hoh, "High-speed processor for quantum-computing emulation and its applications," in *Proc. Int. Symp. Circuits Syst.*, 2003, pp. IV884–IV887.
- [20] M. Fujishima, "FPGA-based high-speed emulator of quantum computing," in *Proc. IEEE Int. Conf. Field-Programm. Technol. (FPT)*, Jul. 2003, pp. 21–26.
- [21] M. Fujishima, K. Saito, and K. Hoh, "16-qubit quantum-computing emulation based on high-speed hardware architecture," *Jpn. J. Appl. Phys.*, vol. 42, pp. 2182–2184, Apr. 2003.
- [22] M. Fujishima, K. Inai, T. Kitasho, and K. Hoh, "75-qubit quantum computing emulator," in *Proc. Extended Abstr. Int. Conf. Solid State Devices Mater.*, 2003, pp. 406–407.
- [23] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134, doi: 10.1109/sfcs.1994.365700.
- [24] D. Greve. (2007). *QDD—A Quantum Computer Emulation Library*. Accessed: Jan. 20, 2024. [Online]. Available: <http://thegreves.com/david/QDD/qdd.html>
- [25] *HDL Coder MATLAB Toolbox*. Accessed: Jan. 20, 2024. [Online]. Available: <https://it.mathworks.com/products/hdl-coder.html>
- [26] Mathworks. *Code Generation Readiness Tool*. Accessed: Jan. 20, 2024. [Online]. Available: <https://it.mathworks.com/help/releases/R2022b/coder/ug/code-generation-readiness-tool.html>
- [27] Mathworks. *Function Supported By HDL Coder*. Accessed: Jan. 20, 2024. [Online]. Available: <https://it.mathworks.com/help/referencelist.html?type=function&capability=hdl>
- [28] *HDL Coder User Guide*. Accessed: Jan. 20, 2024. [Online]. Available: [https://it.mathworks.com/help/pdf\\_doc/hdlcoder/hdlcoder\\_ug.pdf](https://it.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf)
- [29] V. Martin. (2022). *Quantum Computer Simulator*. Accessed: Jan. 20, 2024. [Online]. Available: [https://it.mathworks.com/MATLAB\\_central/fileexchange/73035-quantum-computer-simulator](https://it.mathworks.com/MATLAB_central/fileexchange/73035-quantum-computer-simulator)
- [30] D. Deutsch, "Quantum theory, the church-turing principle and the universal quantum computer," *Proc. Roy. Soc. London, A. Math. Phys. Sci.*, vol. 400, pp. 97–117, Jul. 1985, doi: 10.1098/rspa.1985.0070.
- [31] J. Gambetta. (2019). *Why Use U2 and U1 Gate in IBM Quantum Computers?* Accessed: Jan. 20, 2024. [Online]. Available: <https://quantum-computing.stackexchange.com/questions/7147/why-use-u2-and-u1-gate-in-ibm-quantum-computers>
- [32] D. C. McKay, C. J. Wood, S. Sheldon, J. M. Chow, and J. M. Gambetta, "Efficient Z gates for quantum computing," *Phys. Rev. A, Gen. Phys.*, vol. 96, no. 2, pp. 1–9, Aug. 2017, Art. no. 022330. Accessed: Mar. 16, 2024. [Online]. Available: <https://link.aps.org/accepted/10.1103/PhysRevA.96.022330>, doi: 10.1103/physreva.96.022330.
- [33] Mathworks. (2023). *Cordicsin*. [Online]. Available: <https://www.mathworks.com/help/releases/R2022b/fixpoint/ref/cordicsin.html>
- [34] Mathworks. (2022). *Rounding Method*. Accessed: Jan. 20, 2024. [Online]. Available: <https://it.mathworks.com/help/releases/R2020a/fixpoint/ug/rounding.html>
- [35] C. P. Williams, *Explorations in Quantum Computing*, 2nd ed. London, U.K.: Springer, 2011.
- [36] (Jan. 2022). *MIF\_Gen—A MATLAB Utility*. Accessed: Jan. 20, 2024. [Online]. Available: [https://fpgaer.wordpress.com/2016/08/06/mif\\_gen-a-MATLAB-utility/](https://fpgaer.wordpress.com/2016/08/06/mif_gen-a-MATLAB-utility/)
- [37] terasic. (2015). *DE1-SoC User Manual V1.2.2*. Accessed: Jan. 20, 2024. [Online]. Available: [https://www.terasic.com.tw/cgi-bin/page/archive\\_download.pl?Language=English&No=836&FID=ae336c1d5103cac046279ed1568a8bc3](https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=ae336c1d5103cac046279ed1568a8bc3)
- [38] Terasic. (2021). *DE5a NET DDR4 User Manual*. Accessed: Jan. 20, 2024. [Online]. Available: [https://www.mouser.com/datasheet/2/598/DE5a\\_Net\\_User\\_Manual-1228659.pdf](https://www.mouser.com/datasheet/2/598/DE5a_Net_User_Manual-1228659.pdf)
- [39] B. Steinmann. (Apr. 2022). *Fixed Point vs Floating Point Arithmetic in FPGA*. Accessed: Jan. 20, 2024. [Online]. Available: <https://imperix.com/doc/implementation/fixpoint-vs-floating-point-in-fpga>
- [40] Intel. *Floating-Point IP Cores User Guide*. Accessed: Jan. 20, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/docs/programmable/683750/15-0/introduction.html>
- [41] Mathworks. *Fimath*. Accessed: Jan. 20, 2024. [Online]. Available: <https://it.mathworks.com/help/fixpoint/ref/embedded.fimath.html>



**AGOSTINO GIORGIO** was born in Bari, Italy, in 1969. He received the Laurea degree (Hons.) in electronic engineering from the Polytechnic University of Bari, in April 1994, and the Ph.D. degree in electronics engineering, in 1998. Since 1998, he has been affiliated with the Polytechnic University of Bari, where he is currently an Assistant Professor of electronics and the Head of the Digital Electronics Laboratory. He is the author of numerous articles and has written 15 research and teaching books. His research interests include embedded systems, design and implementation of digital systems on FPGA, artificial intelligence implementation on microcontrollers and FPGA, modeling of electronic devices for nanoelectronics and quantum computing applications, and digital electronics for computing, telemedicine, and home care applications.

...