POLITECNICO DI TORINO
Repository ISTITUZIONALE

AMARETTO, Accelerating Quantum Algorithm Development With FPGA Emulation

Original
AMARETTO, Accelerating Quantum Algorithm Development With FPGA Emulation / Conti, C., Volpe, D., Graziano, M., Turvani, G.. - In: IEEE TRANSACTIONS ON COMPUTERS. - ISSN 0018-9340. - 75:10(2026), pp. 3544-3555. [10.1109/tc.2026.3710326]

Availability:
This version is available at: 11583/3017755 since: 2026-11-11T12:56:35Z

Publisher:
IEEE

Published
DOI:10.1109/tc.2026.3710326

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

19 November 2026

AMARETTO, Accelerating Quantum Algorithm Development With FPGA Emulation

Christian Conti, *Member, IEEE*, Deborah Volpe, *Member, IEEE*, Mariagrazia Graziano, and Giovanna Turvani

***Abstract*—In recent decades, Quantum Computing (QC) has gained widespread attention among researchers and industries, driven by its potential to outperform classical systems in several key areas. However, validating novel quantum algorithms poses substantial challenges due to limited qubit availability and noise in current quantum devices. Although software simulators are frequently employed to validate quantum circuits, they are time-consuming and require high memory availability since as the number of qubits increases, the memory occupation grows exponentially. Hardware emulators have recently emerged as a faster, more efficient, cost-effective alternative. This article introduces AMARETTO (quAntuM ARchitecture EmulaTion TechnOlogy), a new architecture designed to improve quantum computing emulation on resource-constrained Field-Programmable Gate Arrays (FPGAs). To provide a flexible and complete tool for quantum algorithm development, the emulator supports two universal gate sets, Clifford+T and rotational. AMARETTO mimics quantum computing's inherent parallelism through multiple concurrent processing elements, achieving high throughput by harnessing the sparse nature of the gate matrices, coupled with pipelining in a Reduced Instruction Set Computer (RISC)-like architecture. A dedicated compiler translates OpenQASM 2.0 into AMARETTO's instructions, enabling support for custom gates leveraging the equivalences known in the literature. Experimental results on an AMD Zynq UltraScale+ MPSoC ZCU106 demonstrate successful emulation of quantum circuits up to eighteen qubits with a throughput of $12.207\frac{\text{kgates}}{\text{s}}$, validated against Qiskit simulators. Our evaluation confirms competitive performance with existing FPGA-based emulators in terms of throughput and scalability, while targeting an affordable FPGA platform.**

***Index Terms*—Quantum computing emulation, field programmable gate array, quantum algorithm verification, quantum computing simulation.**

I. Introduction

QUANTUM Computing (QC) has emerged as a leading computational paradigm, promising to overcome the limitations of classical computing in big data processing, attracting widespread interest among researchers and leading technological industries. The potential advantages of QC solutions are evident in applications such as optimization [1], and machine learning [2]. However, validation of new QC algorithms is challenging due to limitations in current quantum hardware. Today's, state-of-the-art devices suffer limited connectivity and noise, compromising circuit fidelity. Moreover, high costs, make such technology an exclusive domain of major or specialized companies, which grant access to their hardware through cloud-based platforms, often with associated fees.

The most established approach for verifying novel quantum algorithms involves simulating the evolution of ideal circuits on classical computing platforms. This method enables cost-effective debugging by providing access to information about the quantum state, including phase-related details that are otherwise inaccessible in real quantum devices without additional and expensive state tomography operations [3].

Software simulation suffers from long execution times and high memory demands, limiting large circuit simulations. Moreover, the hardware requirements for simulating quantum circuits vary considerably with problem size. For example, as shown in Fig. 1, guidelines from Google Quantum AI recommend moving from local execution to Graphic-Processing-Unit (GPU) or distributed GPU solutions as the qubit count increases, in agreement with findings reported in the literature [4], [5]. Hardware emulators have the potential to outperform software simulators in describing the qubits state evolution thanks to their flexibility and capability to replicate the parallel nature of QC. For intermediate scale, investigating the Field-Programmable Gate Array (FPGA) alternative holds meaningful promise. Furthermore, FPGAs offer wider optimization potential compared to software when handling application-specific architectures [6].

This work introduces AMARETTO[1] (quAntuM ARchitecture EmulaTion TechnOlogy), a Single Instruction Multiple Data (SIMD) Reduced-Instruction-Set-Computer (RISC)-like

Received 15 October 2025; revised 17 June 2026; accepted 30 June 2026. Date of publication 7 July 2026; date of current version 10 September 2026. This work was supported by AMD under the AMD University program. Recommended for acceptance by R. Canal. *(Corresponding author: Giovanna Turvani.)*

Christian Conti, Mariagrazia Graziano, and Giovanna Turvani are with the Politecnico di Torino, 10129 Torino, Italy (e-mail: christian.conti@polito.it; mariagrazia.graziano@polito.it; giovanna.turvani@polito.it).

Deborah Volpe is with the INGV, 00143 Rome, Italy (e-mail: deborah.volpe@ingv.it).

Digital Object Identifier 10.1109/TC.2026.3710326

[1]The acronym AMARETTO was chosen as a tribute to the traditional Piedmontese almond biscuit, which is made from sweet and bitter almonds, sugar, and egg whites, and baked until crisp.

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

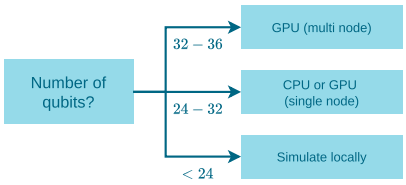


Fig. 1. Guidelines from Google Quantum AI for choosing appropriate hardware to run `qsim` simulations [4].

architecture for quantum computing emulation on resource-constrained FPGAs. It supports Clifford+T and rotational gate sets and exploits the sparse nature of quantum gate matrices by computing only the essential operations. A dedicated transpiler converts OpenQASM 2.0 [7] into RISC-like instructions compatible with the architecture, ensuring compatibility with all OpenQASM 2.0 gates by utilizing state-of-the-art equivalences. The reduced precision impact and the the validity of the results are evaluated by exploiting the Qiskit state vector simulation as a reference, obtaining a fidelity above 0.95 for all the circuits of the MQT Bench tool [8]. We are considering as the target FPGA that on board the AMD Zynq UltraScale+ MPSoC ZCU106 and a twenty-bit fixed-point numeric representation of qubits state, reaching an emulation capability of eighteen qubits with a throughput of $12,207\frac{\text{kgates}}{\text{s}}$. The outcomes are encouraging and prove the approach's validity, allowing the simulation of the same number of emulable qubits of other FPGA-based works in literature and a wider set of gates on a smaller and more accessible FPGA.

The article is organized as follows: Section II introduces the basics required for classical-quantum computer simulation and the related works in the state-of-the-art. Section III presents the proposed SIMD architecture in details. Section IV shows the obtained results and explains the validation methodology. Finally, in Section V, conclusions are drawn, and future perspectives are illustrated.

II. Background and Related Works

This section provides a brief introduction to quantum computing (see [9] for a comprehensive overview) and the specific challenges of emulating quantum circuits on classical hardware (further details in [10]).

A. Quantum Computing Emulation

Quantum computing (QC) is an emerging computational paradigm that exploits the unique principles of quantum mechanics, such as superposition and entanglement, to process information in an intrinsically parallel manner. The fundamental unit of information in QC is the qubit, which differs significantly from a classical bit. While a bit deterministically assumes one of two well-defined states, 0 or 1, a qubit exists in a superposition of states. A qubit can be represented as a linear combination of its basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}, \quad (1)$$

where $\alpha, \beta \in \mathbb{C}$ are probability amplitudes, satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. Using Dirac notation, the state of a qubit is expressed through its state vector, which provides a state-space representation of the quantum system, joining amplitude and phase information. This formalism is fundamental for describing the behavior and dynamics of quantum systems. A n-qubit system can be represented as a single-state vector. If qubits are not interacting, as they are at initialization at the beginning of a quantum program, the global state vector is obtained by computing the tensor product of the involved single-qubit states:

$$\begin{aligned}|\psi\rangle = |\psi_{n-1}\rangle \otimes |\psi_{n-2}\rangle \otimes \cdots \otimes |\psi_1\rangle \otimes |\psi_0\rangle = \\ = c_{0\cdots0}|0\cdots0\rangle + \cdots + c_{1\cdots1}|1\cdots1\rangle\,, \end{aligned} \quad (2)$$

The state of the qubits can be transformed by applying quantum gates, each of which is described as a unitary matrix of dimension $2^m \times 2^m$, where m represents the number of qubits involved. These gates act as the classical counterparts in a circuit, modifying the circuit's state in a controlled and reversible manner. When applied to multiple qubits, quantum gates can induce entanglement. When qubits become entangled, they are strongly correlated, making the state of one qubit intrinsically dependent on the state of the other. Once entanglement occurs, the qubits cannot be evaluated as separate entities; instead, they must be considered as a unique, inseparable quantum system.

To classically simulate a quantum circuit — i.e., apply a series of transformations to the quantum system to implement the desired algorithm — it is necessary to compute, for each layer of the circuit, the product of a $2^n \times 2^n$ matrix and the 2^n state vector associated with the qubit system. Each layer can be recognized as a set of gates applied to different qubits, processed one-by-one from the most significant qubit to the least significant one, before advancing to the next layer.

Within a layer, all gates can ideally be executed concurrently, since they involve different qubits, hence, not presenting data dependencies. The matrices representing these layers are constructed using the tensor product of individual gate matrices applied to each qubit in the circuit. For qubits not targeted by a specific gate, the identity matrix (I) can be adopted as identity element in the tensor product.

Example: Consider the quantum circuit depicted in Fig. 2. The initial state vector is $|\psi_0\rangle = |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$

In the first layer, an X gate is applied to qubit 0, and an H gate is applied to qubit 1. The equivalent transformation matrix for this layer is computed as:

$$\mathrm{G}_1 = \mathbb{I} \otimes \mathrm{H} \otimes \mathrm{X} = \begin{pmatrix}1&0\\0&1\end{pmatrix} \otimes \frac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix} \otimes \begin{pmatrix}0&1\\1&0\end{pmatrix}.$$

The state vector after applying the first layer becames:

$$|\psi_1\rangle = \mathrm{G}_1|\psi_0\rangle.$$

Similarly, to compute the transformation for the second layer, the following operation is performed:

$$\mathrm{G}_2 = \mathbb{I} \otimes \mathrm{CZ}_{0,1},$$

and the output state is obtained as:

$$|\psi_2\rangle = \mathrm{G}_2|\psi_1\rangle.$$

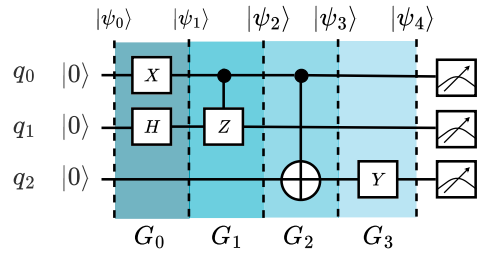


Fig. 2. Three-qubit quantum circuit. Vertical dashed lines depict individual layers, described with the matrices G_x, each formed as the tensor product of that layer's gates.

TABLE I
MAXIMUM NUMBER OF QUBITS THAT CAN BE SIMULATED FOR A QUANTUM CIRCUITS ON VARIOUS SUPERCOMPUTING PLATFORMS. THE TABLE ILLUSTRATES THE EXPONENTIAL GROWTH IN REQUIREMENTS, AS THE MEMORY NEEDED DOUBLES WITH EACH ADDITIONAL QUBIT

Device	Memory (PB)	Qubits
Summit	2.80	47
Sierra	1.38	46
Sunway TaihuLight	1.31	46
Phi	0.80	45

This process is repeated for each layer until the final state of the system is obtained.

Alternatively, the equivalent matrix of the entire quantum circuit can be computed and applied directly to the initial state to obtain the final result. The overall circuit transformation is given by the product of the equivalent matrices of all layers:

$$\mathrm{G} = \mathrm{G}_L \mathrm{G}_{L-1} \cdots \mathrm{G}_2 \mathrm{G}_1,$$

where L represents the number of layers, and G_i denotes the equivalent transformation matrix of the i^{th} layer.

The previous examples put in evidence that manipulating and simulating qubit states fundamentally involves performing complex linear algebra operations on exponentially large vectors and matrices.

B. Previous Work

Different FPGA architectures have been proposed, aiming to overcome the limitations of software simulation of qubits state evolution. Indeed, as reported in [11] and shown in Table I, in addition to the execution time, one of the most critical limitations to scalability lies in the memory requirements for storing a complete state vector. Therefore, even so-called supercomputers are constrained to simulating quantum circuits with, at most, 47 qubits. Similarly, hardware-based emulators also encounter scalability challenges due to limited memory availability. However, as proved in the following works, it is possible to achieve performance comparable to that of supercomputers by exploiting custom-designed hardware architectures. These architectures enable more efficient resource utilization, reduced power consumption, and lastly provide a more affordable and scalable solution.

The majority of the proposed FPGA's based emulators exploit a matrix-vector product approach, exploiting the concurrent execution of the several required multiplications on the reconfigurable hardware, by computing each layer of the circuit to obtain the final state vector.

This section outlines existing quantum circuit emulators, highlighting their key characteristics and limitations relative to our approach. Table II summarizes their performance.

An illustrative example, one of the first quantum circuit emulator, is described in [12] where the quantum circuit is emulated by computing interactions among of basis states on an N-dimensional hypercube. This methodology allows the authors to emulate 16 qubits, with an APEX 20KE1500 FPGA.

The architecture proposed in [15] adopts 32-bit floating-point arithmetic, in contrast to the common trend of using fixed-point representations to reduce storage requirements. Despite this, it successfully emulates a four-qubit system on an Intel Arria 10 FPGA.

Another possible solution is presented in [16], where the authors propose an architecture capable of supporting arbitrary user-defined gates. The emulation system interfaces with an external computer and accelerates the computation of the product between the equivalent layer matrices and the state vector. However, scalability remains a significant challenge, as resource requirements grow rapidly with the number of qubits. In fact, the authors were only able to synthesize a two-qubit emulator on an Intel Altera Cyclone V FPGA.

In [17], the authors optimize the architecture previously introduced in [15] by defining a Complex Multiply-and-Accumulate (CMAC) unit for the probabilty amplitudes update. This CMAC unit served as the basis for the processing element used in AMARETTO, further described in Section III-C. To address scalability challenges, the design leverages external Static Random Access Memory (SRAM) for data storage, enabling the emulation of specific algorithms —such as the Quantum Fourier Transform (QFT) — for quantum systems of up to 32 qubits.

Noteworthy is the solution proposed in [18], which leverages Intel's OpenCL FPGA SDK and a 32-bit floating-point representation to implement the FPGA kernels. This design successfully emulates 29 qubits on a single Nallatech PCIe-385N A7 FPGA, directly interacting with the host computer via PCIe. In contrast, AMARETTO in the proposed work relies on the quad-core Arm Cortex-A53 MPCore™, the Processing System (PS), to manage communication and scheduling of external clients.

In [19] an architecture with an efficient routing of the stored coefficients of the state vector through a permutation network is proposed and implemented on Intel Agilex FPGA, such architecture, is able to elaborate up to 7 qubits.

Beyond the idea of extending memory through off-chip solutions, [20] employs a custom SATA storage controller to interface the emulator with external Hard Disk Drives (HDDs). This approach enables the storage of the state vector for a quantum system with up to 43 qubits. In [21], the authors propose a complete architecture for accelerating quantum circuits on an AMD Alveo U280 FPGA, capable of handling 17 qubits using 32-bit fixed-point precision. The work highlights the

TABLE II
COMPARISON BETWEEN AMARETTO IMPLEMENTATION RESULTS AND THE CURRENT LITERATURE. AMARETTO IS IMPLEMENTED WITH 20-BIT FIXED-POINT REPRESENTATION, AND 16 PE INSIDE THE ARITHMETIC UNIT

Architecture	AMARETTO	AEQUAM [13] *	[12]	[14]	[15]	[16]	[17]	[18]	[19]	[20]	[21]
Qubit	18	5	16	8	4	2	32	Up to 29*	7	Up to 46‡	17
Device	AMD Zynq UltraScale+ MPSoC ZCU106	Intel Cyclone 10LP	Altera APEX 20KE150	Altera Stratix EP1S80B956C6	Altera Arria 10	Altera Cyclone V	Altera Arria 10	Altera Startix V	Altera Agilex AGFB01	AMD Virtex UltraScale+ XCVU13P AMD Kintex UltraScale+ XCKU15P‡	AMD Alveo U280
Memory	1.3 MB	160 B	8.2 MB	-	128 B	10 B	34.3 GB	4.3 GB	-	140 TB	1 MB
Logic Utilization (LU)	58 886	20 993 LE	-	740/gate†	374 021	8 000	56 220	60 029	168 931+	523 000	14 773
LU/Qubit	3 271	4 198	-	-	93 505	4 000	1 756	2 069	24 133	12 162	869
DSP	160	30	-	-	1 364	-	49	16	-	1 968	256
Representation	20-FX	20-FX	-	10/18-FX	32-FP	10-FX	32-FP	32-FP	18-FX	64-FP	32-FX
Frequency	100 MHz	109 MHz	60 MHz	82.1 MHz	233 MHz	-	233 MHz	308 MHz	-	150 MHz – 156.25 MHz	250 MHz

*Parallel configuration of AEQUAM has been considered.
†Refers to the average utilization of elements among the supported gates.
*The achieved qubits count depends on the circuit, the best case is reported.
+Data are referred to a 7 qubit implementation.
‡This work is a multi-device system depending on the SATA generation, and external storage size, 46 qubit are theorized, the data refer to a 43 qubit system.

speed-up achieved in emulation, thanks to the high clock frequency enabled by their architectural design.

Building upon the butterfly-based coefficient selection methodology to further exploit the inherent sparsity of the matrices that characterize quantum gate operations, from an idea originally theorized in [22], the authors previously proposed AEQUAM [13], a SIMD emulation toolchain. Although AEQUAM validated the feasibility of combining butterfly-driven state-vector updates with a custom quantum instruction set, it remained a proof-of-concept prototype with limitations in scalability, execution speed, and memory-access efficiency. AMARETTO transforms this preliminary prototype into a practical and scalable emulation framework through three primary architectural enhancements. First, it introduces dedicated trigonometric computation units, eliminating the need to precompute and preload angle values for rotational gates. Second, it features a completely redesigned, pipelined processing-element datapath that ensures a uniform-latency execution model across all supported gates. By avoiding gate-dependent execution times, AMARETTO achieves higher throughput, improved clock frequencies, and reduced hardware resource utilization compared to AEQUAM's heterogeneous datapath. Finally, AMARETTO implements an advanced memory subsystem leveraging paging, memory interleaving, virtualization, and memory multi-pumping. These techniques enable multiple concurrent accesses to the state vector, substantially improving memory efficiency and allowing the emulation of larger quantum systems on resource-constrained FPGAs. While AEQUAM provided valuable insights, it primarily served as a proof of feasibility. The complete architectural renovation introduced in AMARETTO transforms the original prototype into a practical and scalable emulation framework. AMARETTO can be deployed as a loosely coupled accelerator alongside a processing system, utilizing high-performance, extensible buses for rapid communication. This architecture effectively offloads the compute-intensive task of quantum circuit simulation from the main processor. As a result, AMARETTO is capable of supporting larger quantum systems while achieving substantially higher throughput and improved resource efficiency, making it highly suitable for more realistic quantum emulation workloads.

All these recent works demonstrate the effectiveness of FPGA-emulation approach, which is emerging as a relevant trend in the field of quantum computing. Emulation helps overcome the challenges of simulation, which requires significantly higher computational resources to achieve only marginal improvements in circuit dimensionality.

III. AMARETTO

AMARETTO (quAntuM ARchitecture EmulaTion TechnOlogy) is an efficient architecture for quantum circuit emulation on FPGA platforms, supporting both Clifford+T and rotational gate sets. It is designed for implementation on a wide range of modern FPGAs by exploiting built-in hardware resources such as Block Random Access Memories (BRAMs), Digital Signal Processing (DSP) blocks, multipliers, and carry chains. A high level of portability across different devices is guaranteed by adopting a dedicated interface, that bridges communication between, it this work, the Processing System (PS), or any other controller, and AMARETTO itself, without requiring changes to the whole architecture

Two variants of the architecture have been defined: a preliminary and simplified *Light* version [23], [24], which prioritizes resource and power minimization to ensure compatibility with low-tier FPGAs, and a Full version, designed for high-performances through extensive parallelization of the required operations. Although they shared common components, discussed in the following, their specific features of each are detailed in Sections III-B and III-C.

AMARETTO strategically reduces the computational complexity by exploiting the butterfly-like mechanism, presented in Section II and shown in Fig. 3. This mechanism isolates the interacting probability amplitudes required to compute the output state vector, exploiting the sparse structure of the equivalent gate matrices. By doing so, it reduces redundant operations and avoids the need to compute the full equivalent matrix for each layer, consequently reducing significantly the number of operations needed for each step of the circuit evaluation. The butterfly approach reduces the number of operations per layer to $2^{N_{qubit}-1}$, where N_{qubit} is the number of qubits in the circuit, when the gate targets an uncontrolled qubit. In addition, in

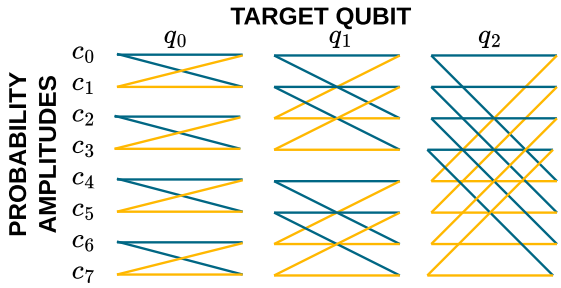


Fig. 3. The butterfly-like mechanism for selecting interacting couples of probability amplitude for each potential target qubit in a three-qubit system. For controlled gate, only a specific subset of these couples is considered.

the case of controlled gates involving two qubits, it is possible to selectively filter the interacting pairs associated with basis states where the control qubit is set to one, further reducing the number of operations to $2^{N_{qubit}-2}$.

Furthermore, a 20-bit fixed-point two's complement representation is adopted, comprising 2 integer bits and 18 fractional bits, with a nearest-even rounding mechanism, able to represent a minimum value equal to $\sim 3.82e^{-6}$, a value small enough to be considered equal to zero in this field [13]. This choice significantly reduces both the area and complexity of arithmetic operators compared to floating-point implementations. More important, it also decreases the memory footprint per probability amplitude, enabling the emulation of a greater number of qubits on the same hardware platform.

The selected numerical precision is justified by the analysis in [13], which proves that the accuracy of the simulation remains largely unaffected by the use of fixed-point arithmetic within the butterfly-based computation scheme.

The AMARETTO environment is designed with a strong focus on user-friendliness, enabling users to describe quantum circuits using popular quantum programming frameworks such as *Qiskit*, *PennyLane*, or *TKET*. These frameworks produce code in the OpenQASM 2.0 format, which is then parsed by the AMARETTO compiler. The compiler translates the circuit into a sequence of supported instructions, which are transmitted to the emulator for execution. Once AMARETTO completes the processing of the instructions describing the quantum circuit, it returns the probability amplitudes of the final quantum state vector. These outputs are made directly available for user analysis, enabling seamless integration into existing quantum algorithm workflows.

A. Instruction Set Architecture and Compiler

The Instruction Set Architecture (ISA), reported in Table III is organized into three categories:

- **S-type** (Set-up type instructions): used during initialization phase to set the number of qubits in the circuit;
- **G-type** (Gate type instructions): responsible for executing supported quantum gates;
- **R-type** (Read type instructions): used to retrieve the resulting probability amplitudes of the state vector associated with the quantum system, either during execution or upon completion.

TABLE III
SUMMARY OF THE ISA INSTRUCTIONS CURRENTLY SUPPORTED BY AMARETTO

Mnemonic	Type	Description
Nop	G-type	No operation
X	G-type	Pauli-X (NOT) gate
Y	G-type	Pauli-Y gate
Z	G-type	Pauli-Z gate
S	G-type	Phase or Z90 gate
$S^\dagger$	G-type	Conjugate transpose of the S gate
H	G-type	Hadamard gate
T	G-type	T gate
$T^\dagger$	G-type	Conjugate transpose of the T gate
R_x	G-type	Rotation around X-axis
R_y	G-type	Rotation around Y-axis
R_z	G-type	Rotation around Z-axis
U_1	G-type	Diagonal gate, equivalent to R_z up to a phase factor
set qubit	S-type	Configure the emulator with the number of qubits in the circuit
barrier	R-type	Retrieve the probability amplitudes of the state vector
measure	R-type	Retrieve the probability amplitudes of the state vector and terminate the execution

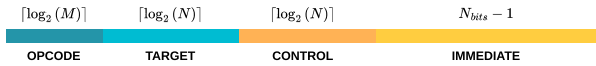


Fig. 4. AMARETTO G-type instruction format, consisting of $\lceil \log_2(M) \rceil$ bits for the opcode, where M is the number of supported operations, and two groups for the identification of control and target qubits, each with $\lceil \log_2(N) \rceil$ bits, where N is the number of supported qubit. Lastly, an immediate field, representing the π-normalized angle.

As shown in Fig. 4, G-type instructions consist of an `opcode` field identifying the specific gate operation, fields for the `target` and `control` qubits (which coincide in the case of single-qubit gates), and an `immediate` field containing the normalized rotation angles. In contrast, both S-type and R-type instructions use only the `opcode` for identification. The S-type uses the `immediate` field to specify the number of qubits, while for the R-type, all other fields are neglected, as only the read operation is encoded in the opcode. For the target FPGA, a 64-bit instruction format is adopted for both architectures to ensure compliance with the Advanced eXtensible Interface (AXI) on-chip communication bus protocol.

Within this format, 5 bits are reserved for the `opcode`, supporting up to 32 distinct instruction codes and allowing room for future extensions. Both the `target` and `control` qubits are identified by encoding their position within the circuit into their respective fields. The size of these fields depends on hardware capabilities; in the current configuration, which supports emulation of circuits with up to 18 qubits, these fields are 5 bits wide. Finally, the instruction includes a 19-bit `immediate` field, which encodes the required π-normalized rotation angles in the range $[-1, 1)$ using an 18-bit fixed-point two's complement representation.

The described instructions are derived from the OpenQASM 2.0 representation of the quantum circuit, using a dedicated compiler that translates the user description of the quantum circuit into machine-level instructions.

Furthermore, the compiler indirectly supports additional gates, all the gates supported by the OpenQASM 2.0, and user defined gates, by exploiting known equivalences in the literature to bring them to the allowable gate set.

In the following, the architecture of AMARETTO will be extensively discussed. In particular, Section III-B presents the basic implementation, detailing each component and the functionality of the architecture. Subsequently, Section III-C discusses the improvements made to the design modules and how they have been implemented.

B. AMARETTO Light: An Efficient Quantum Emulator

AMARETTO Light is a preliminary and simplified version of the overall architecture. Initially developed to reduce the area footprint to its minimum, it evolved into a specific instance of the Full system in which the number of PEs is reduced to one. Additionally, the memory system is simplified, operating without interleaving or sorting networks. As a result, it processes data sequentially, without parallelizing the state vector. AMARETTO is based on a RISC-like architecture, as shown in Fig. 5. It includes a register file (QSRF), where each entry stores two words corresponding to the real and imaginary parts of the quantum state vector. The arithmetic unit (QAU) is composed of a control unit and a datapath, responsible for evaluating the effects of quantum gates on the probability amplitudes of interacting qubit pairs. These interactions are identified by the butterfly module (QSS), which implements a modified version of the butterfly algorithm originally proposed in [22]. To support the evaluation of rotational gates, a trigonometric unit (TU) capable of computing sine and cosine functions is integrated alongside the arithmetic unit. A control unit (QECU) manages the coordination of various operations by generating appropriate control signals, decoding instructions received from an input First-In-First-Out (FIFO) memory, and handling the transmission of the final state vector through an output FIFO. AMARETTO interfaces with the programmable logic using two asynchronous FIFOs to handle Clock Domain Crossing (CDC). This interface approach enhances portability across devices that may not support a common interconnect standard such as AXI. Instead, it relies on conventional FIFO signaling for both data reception and transmission, improving design flexibility and compatibility.

a) Quantum State Register File (QSRF): The QSRF, sized at 2^N entries, where N corresponds to the number of supported qubits, is an internal register file (RF) dedicated to storing the quantum state vector during the execution of a circuit. To simplify address generation, the state vector is stored in a linear, ordered fashion: the probability amplitude of the state $|00\ldots00\rangle$ corresponds to address 0, while the amplitude of the state $|2^N-1\rangle$ is stored at address 2^N-1. To optimize resource utilization, the QSRF is implemented using the Block RAM (BRAM) available within the Programmable Logic (PL) of the target device. The register file is logically designed to support two read and two write ports. However, due to BRAM limitations, it is physically implemented with two read ports and a single write port. To emulate multi-port write functionality,

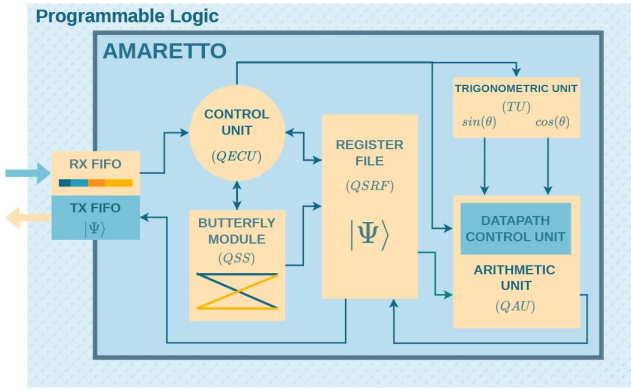


Fig. 5. Overview of the AMARETTO Light architecture, highlighting the various modules and their interconnections. The diagram distinguishes between the accelerator domain and the surrounding programmable logic domain.

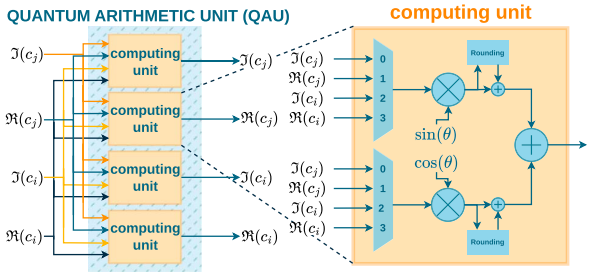
the QSRF operates at twice the nominal clock frequency of the accelerator. This technique, known as pumping [25], enables the reading and writing of two probability amplitudes per clock cycle. As a result, memory bandwidth is effectively increased, significantly reducing the overall system latency.

b) Butterfly Module (QSS): The Butterfly Module, or Quantum State Selector (QSS), is one of the key components that distinguishes AMARETTO from other state-of-the-art quantum circuit emulators. It is responsible for computing, pair by pair, the addresses of the interacting probability amplitudes. The QSS implements a modified version of the butterfly algorithm traditionally used in Fast Fourier Transform (FFT) computations as in [14]. Depending on the target qubit of a given quantum gate, it dynamically generates all the address pairs corresponding to the quantum state amplitudes that must be processed. These addresses are then sent to the QSRF for data retrieval. In the case of controlled gates, not all amplitude pairs require updates. In such situations, the QSS additionally decodes the control qubit and avoids the generation of address pairs where the control qubit is evaluated to zero. This selective computation further optimizes execution efficiency and resource utilization.

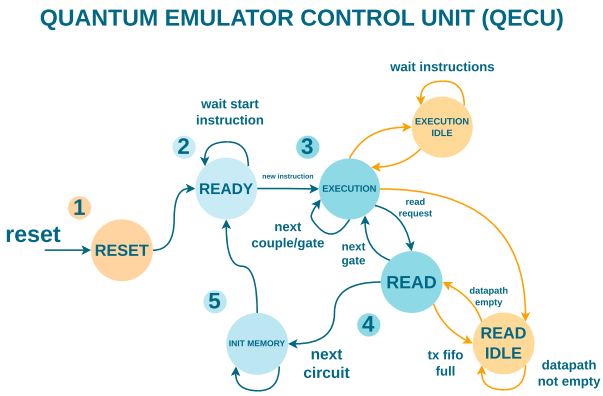
c) Arithmetic Unit (QAU): The Quantum Arithmetic Unit (QAU), shown in Fig. 6(a), is composed of two main components: a standardized datapath (DP) and a control unit. The datapath supports the execution of all Clifford+T and rotational gates, based on the observation that all supported gates can be implemented as follows:

$$\begin{aligned} c_{i_{\text{out}}} &= \alpha \sin(\theta) + \beta \cos(\theta) + i\,(\gamma \sin(\theta) + \delta \cos(\theta)) \\ c_{j_{\text{out}}} &= \epsilon \sin(\theta) + \zeta \cos(\theta) + i\,(\eta \sin(\theta) + \iota \cos(\theta)), \end{aligned} \tag{3}$$

where α, β, γ, δ, ϵ, ζ, η, and ι are real coefficients determined by the specific gate. These coefficients select either the real or imaginary parts of the input probability amplitudes. The variables $c_{i_{\text{out}}}$ and $c_{j_{\text{out}}}$ denote the output probability amplitudes corresponding to the i^{th} and j^{th} basis states, respectively. The parameter θ represents either an immediate angle for parametric gates or a fixed gate-dependent angle for Clifford+T gates. The sine and cosine terms are exploited to control the sign or cancel



(a) AMARETTO Quantum Arithmetic Unit, containing four computing units, evaluating real and imaginary parts of two probability amplitudes according to Equation 3.



(b) AMARETTO QECU Finite State Machine (FSM). The numbers indicate the working flow.

Fig. 6. AMARETTO Quantum Emulator Control Unit and Quantum Arithmetic Unit.

the components by exploiting trigonometric properties. The datapath is replicated four times within the QAU, enabling concurrent processing of the real and imaginary parts of both probability amplitudes in each interacting pair. Furthermore, to enhance precision in the results, for each multiplication the guard, rounding and sticky bit are computed and isolated to perform a rounding to the nearest even approximation mechanism. Additionally, due to the regular structure of Equation (3), instruction-level parallelism is effectively exploited by introducing several pipeline stages into the architecture. This pipelined design significantly reduces execution latency, particularly because couples processed during a gate operation are independent of one another and thus free of data hazards. The pipeline achieves peak efficiency when the number of probabilty amplitude pairs to update meets or exceeds the number of pipeline stages, i.e., when:

$$2^{N_q-2} \geq N_{\text{pipe}} \Rightarrow N_q \geq \lceil \log_2 (N_{\text{pipe}}) + 2 \rceil = N_{q_{\min}} \quad (4)$$

where N_q is the number of qubits in the circuit, and N_{pipe} is the number of pipeline stages. When $N_q < N_{q_{\min}}$, data dependencies between consecutive gates necessitate the insertion of stalls to maintain correctness. To save area in the case of small circuits ($N_q < N_{q_{\min}}$), the architecture still elaborates N_{pipe} elements, but simply discards any results beyond 2^{N_q}. This design choice avoids the need for a dedicated stall-management unit, keeping the same time penalty without incurring additional hardware cost. The second component of the QAU is a compact control unit, implemented as a Look-Up Table (LUT). This module selects the appropriate real and imaginary coefficients based on the gate type, enabling efficient and flexible datapath control across different quantum gate operations.

d) Trigonometric Unit (TU): The Trigonometric Unit (TU) exploits the approach reported in [26], which proposes an efficient solution based on the exploitation of LUTs and the Taylor series to compute both sine and cosine of an input angle. This solution achieve a better balance between area and accuracy, compared to more common COordinate Rotation DIgital Computer (CORDIC) algorithm, or LUT-based solutions. Although CORDIC can be fully unrolled, the considered approach involves fewer processing elements, whose performances can be further improved through the addition of pipeline to reduce the critical path in the TU.

e) Control Unit (QECU): The Quantum Emulator Control Unit (QECU), implemented as a Finite State Machine (FSM) and shown in Fig. 6(b), is responsible for managing and synchronizing all modules within the AMARETTO architecture. The QECU operates through five main states: *RESET*, *READY*, *EXECUTION*, *READ*, and *INIT MEMORY*. Moreover, it includes two pause states that allow the system to wait for data reception from, or data transmission to, the host.

f) Exposed Interfaces: The interaction with the external host is implemented through asynchronous FIFO buffers, platform-independent components, and a platform-dependent interface. Two FIFO buffers are used to interface with AMARETTO: one for receiving instructions to execute, and the other for transmitting the state vector. Both buffers are designed to handle clock domain crossing, hence, mitigating the possibility of metastability issues. In the current implementation, the FIFOs are directly connected to the AMBA 4 AXI4-Stream bus, enabling seamless communication with the ARM-based host system, of the development board. This setup supports the use of Direct Memory Access (DMA), allowing high-throughput data transfer with minimal CPU intervention and efficient utilization of memory and bandwidth resources.

C. AMARETTO Full: A Vector Accelerator

AMARETTO Full represents the natural extension of the light version presented in Section III-B. It prioritizes performance over area efficiency by increasing the number of processing elements available per operation and enabling multiple concurrent elaborations on distinct elements of the state vector(vector-style organization). Indeed, the *Light* version of the accelerator can be traced back to a particular case of the *Full* system, in which the number of processing elements (PE) is minimized to maximize area savings. AMARETTO Full is engineered to saturate the available area at a fixed qubit budget:

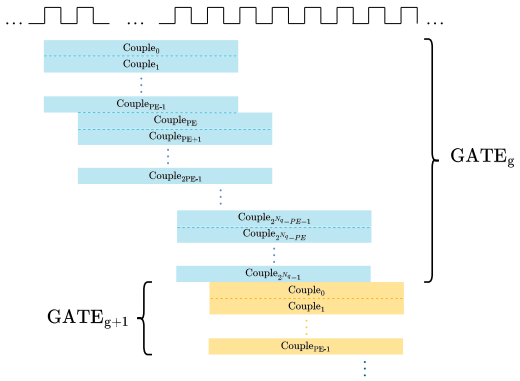


Fig. 7. Batch elaboration of the state vector's coefficient with pipeline segmentation.

its flexible degree of parallelism lets us trade otherwise-unused logic for additional PEs, reducing time-to-solution. This vector-style organization peaks when circuits have many qubits:

$$\frac{2^{N_q-2}}{PE} \geq N_{\text{pipe}} \Rightarrow N_q \geq \lceil \log_2 (N_{\text{pipe}} PE) + 2 \rceil = N_{q_{\min}} \quad (5)$$

where PE is the number of processing elements. Specifically, with nine pipeline stages and sixteen PE, for $N_q > 10$ data dependencies vanish, so all PEs produce a new useful coefficient each cycle to be written into the register file, overall producing a batch of results of size equal to the number of PE each cycle (Fig. 7). On the other hand, the Light variant is simply the minimal-PE instantiation of the same architecture when area saving is crucial.

Furthermore, AMARETTO is fully customizable to fit into different FPGAs and user requirements. The number of PEs, the structure of the register file, the precision of the fixed-point representation, and the depth of the FIFO interfaces can all be configured at pre-synthesis time, improving portability across platforms, also avoiding the use of any Intellectual Property (IP) of the device vendor. As can be seen in Fig. 8 the overall structure of the accelerator is analogous to the one presented in Fig. 5, with additional logic to handle the concurrent execution of operations on different sections of the state vector stored inside the register file.

Indeed, the full expansion of AMARETTO relays on the addition of multiple computation units inside the QAU, capable of performing the same operation multiple times on different sections of the array, thus making the arithmetic section of AMARETTO a complete vector accelerator. To guarantee that each PE always has new data available, the register file has also been modified and reorganized into pages, each with separate read and write ports, with the insertion of a sorting layer to address the data to the PEs.

The following highlights the improvements of the accelerator.

a) Quantum State Register File (QSRF): Thanks to the peculiar nature of the operations to be executed, where elements of the same array are selected through the butterfly algorithm and each pair of interacting coefficients is separated by an offset that is a power of two, it is possible to efficiently reorganize the storage of the array within memory. The vector is divided into chunks that are stored across different memory pages, which are small, independent memory blocks with separate access ports, instead of being stored sequentially in a unified memory structure.

Storing the elements sequentially across multiple pages instead of vertically stacking consecutive memory locations on a single page ensures at most two concurrent accesses to a single page for both read and write operations. Therefore, by again exploiting pumping with a factor of two, it is possible to double the number of ports for each page.

This kind of memory organization guarantees stall-free circuit execution by ensuring that fresh data is provided to all processing elements, as long as the number of register file pages equals the number of elements. However, this approach requires additional logic to operate correctly. Specifically, the QSS generates the indices of the element pairs involved in the computation, but these indices must be translated into corresponding memory addresses for each page. As such, the indices can be considered virtual addresses that must be mapped to physical addresses by the Register File Management Unit (RFMU). The RFMU is responsible for extracting the page offset and page index from each virtual address and forwarding them to the QSRF.

Besides, the elements read from the QSRF are ordered according to the memory scheme, which does not match the coupling required for the computation. Therefore, an additional sorting step is performed by exploiting the physical addresses, using a small sorting network responsible for pairing the correct elements of the array to the respective PE. Similarly, after the computation, the coefficients must be reordered before being correctly stored back into the QSRF. To this end, a second sorting network, which performs the reverse operation of the first, is included to ensure that the elements of the state vector are properly written into the QSRF.

b) Butterfly Module (QSS): As in the *light* version, the QSS is responsible for computing the interacting pairs of the state vector based on the involved qubits. However, the module has been divided into two sections. The first, QSS, generates concurrently the virtual address of different interacting pairs of coefficients involved in the computation, while the second module, in charge of generating the write enable signals for the register file, the Register File Write Enabler (QSE), generates a mask containing the enable signals for each write port of the register file, masking the unused state coefficients when an operation involves a controlled gate and consequently avoiding the updated of values not involved in the current instruction even if they have been read from the register file. Both the mask and the virtual addresses are then fed into the QSRF, where they are used for the read and write operations of the current instruction.

c) Arithmetic Unit (QAU): The arithmetic unit consist of replicated PE to match the number of pages in the register file, allowing each PE to process a different couple of complex coefficients avoiding stall matching the throughput of the internal memory as described in paragraph III-C0a. The structure of the PEs is identical to that of the lighter version of AMARETTO:

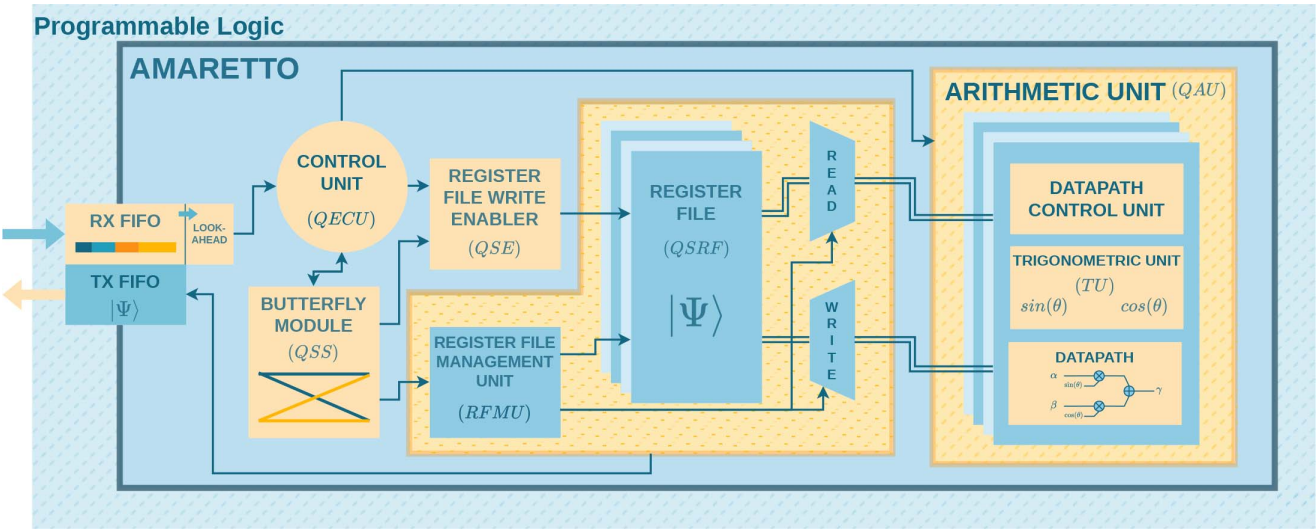


Fig. 8. Overview of the AMARETTO Full architecture, highlighting the various modules and their interconnections.

each PE includes a control unit that decodes instructions and configures the datapath, a datapath that executes the operation in Equation 3, and a trigonometric unit. Although the control and trigonometric units could be shared across all PEs, replication is preferred due to their small size, which simplifies placement, reduces FPGA congestion, thus also saving resources.

d) Look-ahead: To speed up the overall execution of the emulation, a so-called look-ahead module is chained to the head of the input FIFO. This module consists of a small two-register shift buffer, with empty and full signals, in which two instructions are always loaded. At each instruction request, the instruction at the head is popped out, the tail is shifted forward, and a new instruction is pushed in. The peculiarity of this buffer is that it allows a pre-fetch of the next instruction to be executed. The ability to know the next instruction in advance is useful for accelerating circuit execution by enabling the concurrent execution of the current instruction and the preparation of the next operation, particularly when it diverges from the current instruction flow.

IV. Results

This section presents implementation results, compares them to the state of the art, and outlines our validation methodology and scalability considerations.

a) Functional verification: The functional verification of the architecture was performed by considering quantum circuits expressed in OpenQASM 2.0 from available in MQT-Bench [8] and comparing the resulting state vectors with those produced by the Qiskit state-vector simulator. The results are evaluated with different metrics to ensure the correctness of the results from different point of view:

1) The Fidelity (F), that describes how similar two states are, defined as:

$$F(\rho,\sigma) = \left|\langle\psi_\rho \mid \psi_\sigma\rangle\right|^2$$

2) The Hellinger Distance (HD), that measures the similarity between two probability distributions from a classical point of view, defined as:

$$H(p,q) = \frac{1}{\sqrt{2}}\sqrt{\sum_i \left(\sqrt{p_i} - \sqrt{q_i}\right)^2}$$

3) The Orthodromic Distance (OD), the shortest distance between two points on the surface of a sphere. Since state vectors can be seen as points on a sphere, their spherical distance quantifies the divergence between the two results.

OD more reliably reflects discrepancies in complex amplitudes. In all tests, OD and HD were < 0.05 while F exceeded 0.95, satisfying the acceptance criterion.

The design was synthesized on the AMD Zynq UltraScale+ MPSoC ZCU106 using Vivado 2023.1 with default directives. The qubit count was progressively increased, up to a maximum of 18 qubits. The main bottleneck identified during synthesis was the availability of RAM. This is coherent with the expectations since the RAM occupation, used only for storing the state (QSRF block), scales as $\mathcal{O}(N_{\text{bit}}2^{N_q})$, where N_{bit} is the number of bits exploited for number representation. Moreover, the memory footprint also depends on the selected degree of parallelization, since virtually increasing the number of read and write ports requires organizing the memory into additional pages, which implies supplementary logic to handle the memory, finally leading to additional area occupation. On the other hand, the utilization of Configurable Logic Blocks (CLBs) and Digital Signal Processing (DSP) units can be optimized according to the resource availability of the target board by selecting an appropriate degree of parallelization, i.e., the number of PE. Indeed, QAU occupation scales as $\mathcal{O}(N_{\text{bit}}PE)$, where PE is the number of processing elements(independently from the number of qubits) and the QSS scales as $\mathcal{O}(N_qPE)$. The communication interface varies depending on the target platform, whereas

TABLE IV
RESOURCE UTILIZATION OF THE AMARETTO ARCHITECTURE DEPENDING ON THE NUMBER OF INSTANTIATED PROCESSING ELEMENTS (PES), MAINTAINING A CONSTANT MAXIMUM EMULATION CAPACITY OF 18 QUBITS

# of PE	LUT	Flip-Flop	BRAM	DSP
1	8290	2 388	288	10
2	10 174	3356	288	20
4	15 893	5868	288	40
8	28 200	10 489	288	80
16	58 886	19 754	288	160

the other blocks remain constant. Consequently, the memory contribution becomes the dominant factor in this context.

b) Implementation results: Table IV details the resource consumption of various AMARETTO configurations based on the number of Processing Elements (PEs) within the arithmetic unit. Throughout these configurations, the memory capacity is fixed to support 18 qubits, the upper limit of the target FPGA. As expected, BRAM utilization remains constant, whereas DSP block usage scales linearly with the PE count, highlighting the DSP blocks needed to insatiate a PE. In contrast, LUT and Flip-Flop utilization exhibits a non-linear trend. This overhead arises from the escalating complexity of the QSRF wrapping logic, where the routing requirements for the fully-connected read and write networks increase quadratically with the number of endpoints.

The best AMARETTO configuration is compared against the current literature in Table II. However, direct comparisons are not trivial due to architectural differences, variations in target boards from different vendors, and the slightly heterogeneous nature of their fundamental building blocks. In addition, the information reported in related works is often incomplete, and in many cases, the synthesis results depend on the specific quantum circuit under evaluation, differently from AMARETTO. In fact, AMARETTO has the distinctive advantage of not requiring re-synthesis when executing a new circuit. Moreover, thanks to its wide set of supported gates, it can execute any application whose qubit requirements fit within the maximum synthesizable capacity of the selected platform. Despite these challenges, some observations can still be drawn. Although the board adopted in this work is not particularly large, the architecture in [17] is the only one achieving a higher number of emulable qubits (even though limited to the QFT algorithm), by using external memory. Nevertheless, the design in [17] requires significantly more memory per qubit than AMARETTO, since it employs a larger number of bits for numerical representation and stores both the state vector and the gate matrices. Furthermore, considering only architectures relying only on on-chip memories, the relative logic utilization metric (LU/Qubit) reported in Table II shows that AMARETTO achieves the lowest value among the compared designs.

c) Time analysis: Despite operating at a lower clock frequency than [15], [17], AMARETTO is expected to achieve higher performance, since the execution of a single gate, i.e. a single update of the entire state vector, in full-pipeline

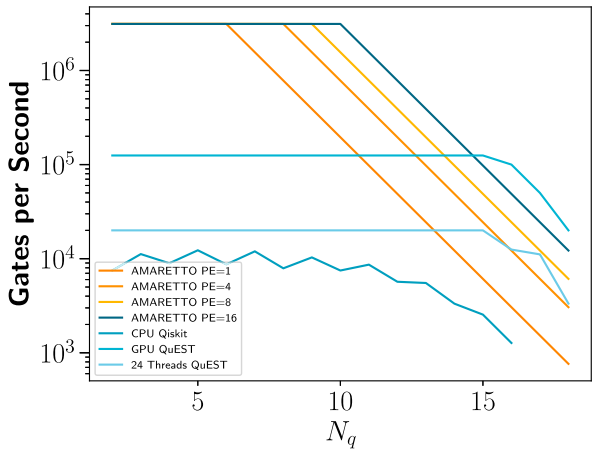


Fig. 9. Throughput (gates/s) of AMARETTO, varying the number of processing elements, Qiskit state vector simulator executed on a CPU and QuEST (an open-source GPU-Multi-thread accelerator for universal quantum circuit simulation) executed in GPU and multiThreads mode.

condition requires only $\mathcal{O}\left(\frac{N}{2PE}\right)$ clock cycles, compared to $\mathcal{O}(N^2)$ in those architectures, where N denotes the state vector length, i.e., 2^{N_q}. Specifically, our emulator is able to emulate $\frac{2PE}{T_{ck}2^{N_q}}\frac{\text{gate}}{\text{s}}$, which corresponds to 12.207 $\frac{\text{kgate}}{\text{s}}$ considering eighteen qubits (N_q), nine pipeline stages (N_{pipe}), a clock period of 10 ns (T_{ck}) and sixteen processing elements (PE). Fig. 9 reports throughput (gates/s) as a function of the qubit count. We compare AMARETTO with CPU execution using Qiskit State Vector and with QuEST [27], an open-source simulator that exploits GPU acceleration and multi-threaded CPU execution. QuEST data points are reconstructed from Fig. 6 in [27] as the inverse of the time per gate. The figure also shows how increasing the number of processing elements boosts AMARETTO's throughput, if the number of qubits is sufficiently high to avoid stalls.

Moreover, the AMARETTO execution time running the benchmark circuits of MQT bench has been measured and compared against Qiskit state vector simulator, the Qiskit QASM simulator. The software test was performed on a single-process Intel(R) Xeon(R) Gold 6134 CPU 0x00040 3.20 GHz optacore, Model 85, with a memory of about 103 GB [28]. Specifically, Fig. 10 shows the execution time as a function of the quantum circuit size, i.e. $N_g 2^{N_q}$, where N_g is the number of gates to emulate and N_q the number of qubits. The obtained results are coherent with the expectations since the execution time required is expected to scale as $\left(2^{\max(N_q, N_{q_{\min}})-1}\frac{N_g(2-\alpha)}{2PE} + (N_{\text{pipe}}-1)\right)T_{\text{clk}}$, where α is the percentage of controlled gates, implying a linear scalability with circuit complexity $2^{N_q}N_g$. AMARETTO shows a run time orders of magnitude lower than software emulators, proving the advantage of the hardware emulation. The benefits increase with the quantum circuit size.

d) Power Analysis: We analyzed the SoC's power consumption during quantum circuit emulation. Fig. 11 shows the board power, divided into the Processing System (PS), running PetaLinux with PYNQ, the Programmable Logic (PL), and their

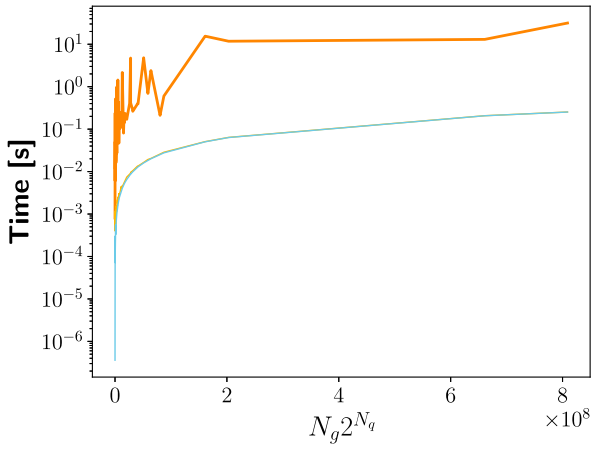


Fig. 10. Execution time of Qiskit state vector simulator, and AMARETTO (with and without considering the time required for transmitting the instructions and retrieving the state vector), as a function of $N_g 2^{N_q}$, where N_g is the number of gates to emulate and N_q the number of qubits. The independent variable is expected to represent the increase of complexity with the butterfly computation mechanism.

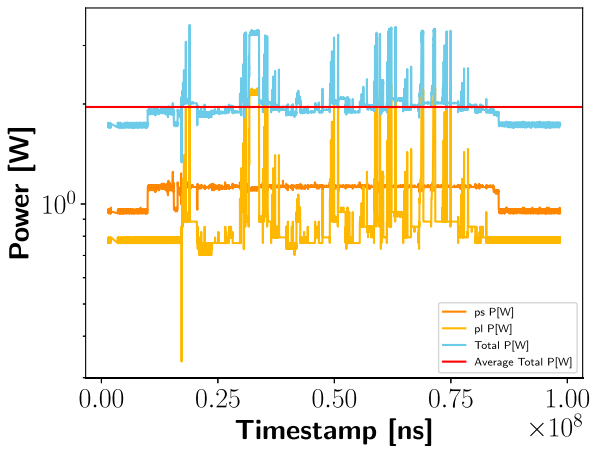


Fig. 11. Power consumption of the SoC implementing AMARETTO, showing the Processing System (PS) power, the Programmable Logic (PL) power, and their sum (Total).

sum (Total). Power measurements were collected using Texas Instruments INA226 sensors installed on the board, which monitor the system's power rails and are accessed through an external I^2C device.

From Fig. 11, it is possible to observe an increase in power consumption in the PS when the accelerator is active, as well as power spikes in the PL during quantum circuit elaboration. During idle and setup operations, such as register file cleaning, the PL power consumption drops to an almost constant level. On average, the power consumption of the entire system during benchmarking with [3] is approximately 1.95 W.

V. Conclusion

The growing interest in quantum computing has accelerated the exploration of efficient methodologies to validate novel quantum algorithms. Given the limited qubit availability and the presence of noise in current quantum devices, alternative approaches are essential. Moreover, execution on real-devices does not allows an effective debugging through direct access to information about the quantum state. In this work, we introduced AMARETTO, a specialized architecture for quantum computing emulation on low-tier FPGAs. AMARETTO supports both Clifford+T and rotational gate sets, enabling the execution of arbitrary quantum circuits.

Built on a RISC-like structure and optimized for the sparse nature of quantum gate matrices, AMARETTO makes less complex the functional verification of quantum algorithms. Its integration into user-friendly software further improves accessibility and usability, making the system valuable for both academic researchers and industrial practitioners.

A comparative analysis between AMARETTO's execution time and the Qiskit state-vector simulator highlights the clear advantage of hardware emulation over software. AMARETTO consistently achieves execution times at least two orders of magnitude lower, with advantages that increases as the circuit size increases. Furthermore, the successful emulation of eighteen qubits on a AMD MPSoC ZCU106 proves the potential of this approach, achieving results competitive with other state-of-the-art works. This represents a promising solution for addressing the challenges of validating new quantum algorithms and opens the way to practical quantum computing applications.

Although the obtained results are promising, several opportunities for improvement exist to further increase the efficiency and scalability of AMARETTO. At the architectural level, optimizing the address generation mechanism, currently based on the butterfly-like selection of interacting amplitudes, could significantly lower latency and resource consumption, especially in the *Full* architecture variant. Exploring custom numerical representations offers another possible refinement. Specifically, reduced-precision floating-point formats represent an important direction. Such formats may offer a better trade-off between numerical accuracy and hardware efficiency compared to fixed-point arithmetic, potentially enabling the emulation of larger quantum circuits on the same device. To maximize parallel execution, support for Packed-Single-Instruction-Multiple-Data (PSIMD) instructions can reduce the area of the PEs. This allows different PEs to share resources, minimizing the replication of internal components. Furthermore, accessing external memory resources could significantly extend the scalability of AMARETTO by overcoming the storage limitations imposed by on-chip BRAM, enabling the simulation of more complex quantum systems at the expense of increased memory-access latency, which can be mitigated with the addition of multiple master controllers in the data plane to parallelize data transfers and saturate the external memory bandwidth. Looking ahead, future development will focus on extending portability across heterogeneous hardware platforms and integrating advanced compilation strategies to fully exploit hardware capabilities and minimize communication overhead.

References

[1] E. Zahedinejad and A. Zaribafiyan, "Combinatorial optimization on gate model quantum computers: A survey," 2017, *arXiv:1708.05294*.

[2] N. Mishra et al., "Quantum machine learning: A review and current status," in *Proc. Data Manage., Analytics Innov.*, Singapore: Springer, 2021, pp. 101–145.
[3] R. Wille et al., "The MQT handbook: A summary of design automation tools and software for quantum computing," in *Proc. IEEE Int. Conf. Quantum Softw. (QSW)*, Shenzhen, China, 2024, pp. 1–8, doi: 10.1109/QSW62656.2024.00013.
[4] Google Quantum AI, "Choosing hardware for your qsim simulation," 2024. Accessed: Aug. 27, 2025. [Online]. Available: https://quantumai.google/qsim/choose_hw
[5] S. V. Isakov et al., "Simulations of quantum circuits with approximate noise using qsim and Cirq," 2021, *arXiv:2111.02396*.
[6] H. Li and Y. Pang, "FPGA-accelerated quantum computing emulation and quantum key distillation," *IEEE Micro*, vol. 41, no. 4, pp. 49–57, Jul./Aug. 2021.
[7] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, "Open quantum assembly language," 2017, *arXiv:1707.03429*.
[8] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, 2023.
[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (10th anniversary ed.). Cambridge; New York: Cambridge University Press, 2010.
[10] H. Li and Y. Pang, "FPGA-accelerated quantum computing emulation and quantum key distillation," *IEEE Micro*, vol. 41, no. 4, pp. 49–57, 2021.
[11] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Full-state quantum circuit simulation by using data compression," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, ACM, 2019, pp. 1–24.
[12] M. Fujishima, K. Saito, and K. Hoh, "16-Qubit quantum-computing emulation based on high-speed hardware architecture," *Jpn. J. Appl. Phys.*, vol. 42, no. 4S, 2003, Art. no. 2182.
[13] L. Lagostina, D. Volpe, M. Zamboni, and G. Turvani, "AEQUAM: Accelerating quantum algorithm validation through FPGA-based emulation," *IEEE Access*, 2025.
[14] C. Conceição and R. Reis, "Efficient emulation of quantum circuits on classical hardware," in *Proc. IEEE 6th Latin Amer. Symp. Circuits Syst. (LASCAS)*, IEEE, 2015, pp. 1–4.
[15] N. Mahmud and E. El-Araby, "A scalable high-precision and high-throughput architecture for emulation of quantum algorithms," in *Proc. 31st IEEE Int. Syst.-on-Chip Conf. (SOCC)*, IEEE, 2018, pp. 206–212.
[16] J. Pilch and J. Długopolski, "An FPGA-based real quantum computer emulator," *J. Comput. Electron.*, vol. 18, pp. 329–342, Dec. 2019.
[17] N. Mahmud, B. Haase-Divine, A. Kuhnke, A. Rai, A. MacGillivray, and E. El-Araby, "Efficient computation techniques and hardware architectures for unitary transformations in support of quantum algorithm emulation," *J. Signal Process. Syst.*, vol. 92, pp. 1017–1037, Jul. 2020.
[18] Y. Moawad, A. Brown, R. Steijl, and W. Vanderbauwhede, "Optimising iteration scheduling for full-state vector simulation of quantum circuits on FPGAs," 2024, *arXiv:2411.18354.*
[19] L. A. Belfore II, "A scalable FPGA architecture for quantum computing simulation," 2024, *arXiv:2407.06415*.
[20] K. Wei, H. Amano, R. Niwase, Y. Yamaguchi, and T. Miyoshi, "QuTrefoil: Large-scale quantum circuit simulator working on FPGA with SATA storages," *IEEE Trans. Comput.*, vol. 74, no. 4, pp. 1306–1321, Apr. 2025.
[21] V. D. Tran, T. H. Vu, V. T. D. Le, H. L. Pham, and Y. Nakashima, "QEA: An accelerator for quantum circuit simulation with resources efficiency and flexibility," 2025, *arXiv:2503.14951.*
[22] G. Negovetic, M. Perkowski, M. Lukac, and A. Buller, "Evolving quantum circuits and an FPGA-based quantum computing emulator," 5th Int. Workshop Boolean Problems, Freiberg, Germany, Sep. 2002.
[23] C. Conti, D. Volpe, M. Graziano, M. Zamboni, and G. Turvani, "AMARETTO: Enabling efficient quantum algorithm emulation on low-tier FPGAs," in *Proc. IEEE Int. Conf. Electron. Circuits Syst.*, 2024, pp. 1–4.
[24] C. Conti, D. Volpe, G. A. Cirillo, M. Graziano, M. Zamboni, and G. Turvani, "Towards quantum circuit emulation on low-tier FPGAs," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2024, pp. 512–513.
[25] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported SRAM-based memories," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays, (FPGA)*, New York, NY, USA: ACM, 2014, pp. 35–44.
[26] F. De Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 83–88, 2014.
[27] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Sci. Reports*, vol. 9, no. 1, 2019, Art. no. 10736.
[28] "Intel Xeon Gold 6134 processor - product specification." Accessed: Nov. 17, 2022. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/120493/intel-xeon-gold-6134-processor-24-75m-cache-3-20-ghz.html



Christian Conti (Member IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering in 2020 and 2024, respectively, from the Politecnico di Torino, where he is currently working toward the Ph.D. degree in electrical, electronics and communications engineering. His research interests include quantum computing emulation on classical hardware (FPGA, CPU, and GPU), digital design, qubit control, and quantum circuit compilation.



Deborah Volpe (Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering in 2019 and 2021, respectively, and the Ph.D. degree in electrical, electronics and communications engineering in 2025, from the Politecnico di Torino. She is currently a Postdoctoral Researcher with INGV. Her research interests include the quantum computers emulation on classical hardware, quantum approaches for solving optimization problems, probabilistic computing, and Ising machines.



Mariagrazia Graziano received the M.Sc. and Ph.D. degrees in electronic engineering from the Politecnico di Torino in 1997 and 2001, respectively. She is currently an Associate Professor with Politecnico di Torino. She has been served as Adjunct Faculty with the University of Illinois, Chicago since 2008 and as Lecturer with EPFL since 2015. Her research interests include on nanotechnology, emerging devices, nanocomputing architectures, field-coupled nanocomputing, and quantum computing.



Giovanna Turvani received the M.Sc. and Ph.D. degrees in electronic engineering from the Politecnico di Torino in 2012. She is currently an Associate Professor with Politecnico di Torino. In 2016, she was a Postdoctoral Research Associate with the Technical University of Munich. Her research interests include CAD tools for non-CMOS/field-coupled nanocomputing, architectural design, quantum-computing, microwave-imaging hardware, and low-power IoT systems.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement