IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 25 April 2025, accepted 21 May 2025, date of publication 26 May 2025, date of current version 3 June 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3573313

RESEARCH ARTICLE

FQsun: A Configurable Wave Function-Based Quantum Emulator for Power-Efficient Quantum Simulations

TUAN HAI VU[1], (Member, IEEE), VU TRUNG DUONG LE[1], (Member, IEEE), HOAI LUAN PHAM[1], (Member, IEEE), QUOC CHUONG NGUYEN[2], AND YASUHIKO NAKASHIMA[1], (Senior Member, IEEE)

[1]Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan
[2]Department of Mathematics, University at Buffalo, Buffalo, NY 14260, USA

Corresponding author: Vu Trung Duong Le (le.duong@naist.ac.jp)

This work was supported in part by the Japan Science and Technology Agency (JST)-Advanced Technologies for CArbon-Neutral (ALCA-Next)-Next Program, Japan, under Grant JPMJAN23F4; in part by the Japan Society for the Promotion of Science (JSPS), Grants-in-Aid for Scientific Research (KAKENHI), Japan, under Grant 22H00515; in part by the 2025 Priority Strategic Funds Promotion of open access publication of scholarly articles; and in part by the Next Generation Researchers Challenging Research Program under Grant zk24010019.

ABSTRACT Quantum computers are promising powerful computers for solving complex problems, but access to real quantum hardware remains limited due to high costs. Although software simulators like Qiskit, ProjectQ, and Pennylane offer flexibility and support for many qubits, they struggle with high power consumption and limited processing speed, particularly as qubit counts increase. Accordingly, quantum emulators implemented on dedicated hardware, such as FPGAs and analog circuits, offer a promising path for addressing energy efficiency concerns. However, existing studies on hardware-based emulators still face challenges in terms of limited flexibility and lack of fidelity evaluation. To overcome these gaps, we propose FQsun, a wave function-based quantum emulator that enhances performance by integrating four key innovations: efficient memory organization, a configurable Quantum Gate Unit (QGU), optimized scheduling, and five number precisions implemented on the Xilinx ZCU102. Experimental results demonstrate high fidelity, low mean square error, and high normalized gate speed, particularly with 32-bit versions. Benchmarking on Random Quantum Circuits, Quantum Fourier Transform, and Parameter-Shift Rule reveals that FQsun achieves a superior power-delay product and precisely, outperforms software simulators on CPUs in the processing speed range.

INDEX TERMS Quantum emulator, field-programmable-gate-arrays, quantum computing, wave-function, simulation.

I. INTRODUCTION

Quantum computing is a captivating research field that has many promising applications in factorial problem [1], searching [2], optimization [3], or quantum machine learning [4]. Numerous researchers have developed real quantum devices successfully, such as IBM Quantum [5], Google Quantum AI [6], and QuEra [7]; some even claim to have achieved "quantum supremacy". Recently, quantum computers have rapidly transformed from the Noise-Intermediate Scale Quantum (NISQ) era to the Fault-Tolerant Quantum Computer (FTQC) era, in which the logical error rate from quantum calculations is acceptable; it pursued the development of research in quantum computing. However, accessible real quantum computers such as IBM computers are limited because of the high cost.

To satisfy the growing demand for quantum computing simulations, several substantial research efforts have been conducted, as summarized in detail in Table 1. Key criteria include physical hardware implementation, benchmarking tasks, the number of simulated qubits (#Qubits), precision, and evaluation metrics. Among existing quantum simulators,

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

© 2025 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/

the most well-known include Qiskit [8], ProjectQ [9], Cirq [10], and TensorFlow Quantum (TFQ) [11], which are developed in Python. These simulators not only support a wide range of quantum simulation models with high #Qubits but also enable deployment on actual Quantum Processing Units (QPUs). Additional packages, such as QUBO [12], PennyLane [13], and cuQuantum [14], are also widely used due to their efficient performance on general-purpose processors, including CPUs and GPUs. Notably, cuQuantum is optimized specifically for NVIDIA GPUs. Another notable package is Qsun, a quantum simulation package proposed in [15], which introduces a Wave Function (WF) - based quantum simulation system to reduce the computational load. Overall, these packages exhibit flexible and increasingly fast performance on software platforms. However, as the demand for higher #Qubits grows, the general-purpose design of these software-based quantum simulators results in exponentially decreased processing speed and increased memory. Consequently, these platforms tend to consume considerable energy without meeting speed requirements, an issue that is often overlooked in current research. *In the future, if simulators running on CPUs/GPUs become widespread, which consume between 150-350 W, will have a significant environmental impact.*

To solve the limitations of software, various quantum emulator have been developed for hardware platforms, particularly analog circuits and Field-Programmable Gate Arrays (FPGA). Specifically, analog-based emulators, introduced in [16], use CMOS analog circuits to represent real numbers naturally, which are suitable for simulating quantum states. Although analog circuits achieve higher energy efficiency, scalability, and speed than software, they face limitations when they only support a single algorithm like Grover's Search Algorithm (GSA) or Quantum Fourier Transform (QFT) with lower fidelity due to noise and other unintended effects. In contrast, FPGA-based emulators, introduced in [17], [18], [19], [20], [21], [22], [23], [24], [25], and [26], offer a more practical and developable approach while still achieving high speed and energy efficiency. However, these studies lack detailed fidelity evaluations, particularly when using 8-bit and 16-bit fixed-point (FX) representations, which may not meet precision requirements. FPGA-based emulators are also limited by high storage and communication demands, restricting them to lower #Qubits. Moreover, these FPGA emulators have low flexibility, as they are optimized only for a specific group of applications such as QFT, GSA, Quantum Haar Transform (QHT). Existing works face ongoing challenges: quantum simulators consume high energy, and hardware-based emulators are often inflexible, imprecise, and difficult to implement. *Although they attempt to improve performance, most are constrained by the #Qubit they can simulate and lack metrics such as fidelity or mean-square error (MSE).*

To address the mentioned challenges, FQsun is proposed to achieve optimal speed and energy efficiency as a general quantum emulator. FQsun builds on Qsun's theoretical foundation by introducing four significant practical innovations: efficient memory organization, a configurable Quantum Gate Unit (QGU), optimal working scheduling, and multiple number precisions to maximize the flexibility, speed, and energy efficiency of the quantum emulator. FQsun is implemented on an Xilinx ZCU102 to demonstrate its effectiveness at the real-time SoC level. Through strict evaluation of execution time, accuracy, power consumption, and power-delay product (PDP), FQsun will demonstrate its superiority over software quantum simulators running on powerful Intel CPUs and NVIDIA GPUs, as well as existing hardware-based quantum emulators.

The outline of this paper is organized as follows. Section II presents the background of the study. Next, details of the FQsun hardware architecture are presented in Section III. Then, benchmarking results, evaluation, and comparisons are elaborated upon in Section IV. Finally, Section VI concludes the paper.

II. BACKGROUNDS

A. QUANTUM SIMULATOR

Due to the wide range of applications and programming languages, many quantum simulators have been proposed, from low-level languages such as C to high-level languages such as Python. The goals of quantum simulator development range from domain-specific to general purpose, then reach the boundary of quantum advantage where classical computers can no longer simulate a quantum system in acceptable runtime. The basic operation in n - qubit system can be expressed as $|\psi^{(\boldsymbol{m})}\rangle = \mathcal{U}(\boldsymbol{\theta})|\psi^{(\mathbf{0})}\rangle$ parameterized by $\boldsymbol{\theta}$, where $\mathcal{U}(\boldsymbol{\theta})$ is composed from $\{g_j\}_{j=1}^{m}$ and m is the number of quantum gates (gates). $\mathcal{U}(\boldsymbol{\theta})$ can be presented as $\bigotimes_{j=1} U^{(\boldsymbol{t})}$ which is the tensor product between operators.

The quantum state $|\psi^{(\boldsymbol{t})}\rangle$ is a column complex vector $[\alpha_0^{(\boldsymbol{t})}\ \alpha_1^{(\boldsymbol{t})}\ \ldots\ \alpha_{N-1}^{(\boldsymbol{t})}]^\intercal = \sum_{j=0}^{N-1} \alpha_j^{(\boldsymbol{t})}|j\rangle$ with $|j\rangle$ are elements of the computational basis and $\{\alpha_j^{(\boldsymbol{t})}\}$ are complex entries ($N = 2^n$). $\mathcal{U}(\boldsymbol{\theta})$ and $|\psi^{(\boldsymbol{t})}\rangle$ satisfy unitary and normalize condition, i.e, $(U^{(\boldsymbol{t})})^\dagger U^{(\boldsymbol{t})} = \mathbb{I}$ and $\langle\psi^{(\boldsymbol{t})}|\psi^{(\boldsymbol{t})}\rangle = 1$.

The target of simulating quantum computing is to get the expectation value when measuring a final quantum state under an observable $\hat{B}$: $\langle E\rangle = \langle\psi|\hat{B}|\psi\rangle$, which is known as the physical property of the system, summarized as Definition 1.

Definition 1 (Strong and weak simulation): Given a quantum circuit U and any specified object x, the strong simulation can compute any value $\langle\mathbf{0}|U|x\rangle$ and a weak simulation samples from the distribution $p(x) = |\langle\mathbf{0}|U|x\rangle|^2$ [27], [28].

The term $\mathcal{U}(\boldsymbol{\theta})|\psi^{(\mathbf{0})}\rangle$ can be computed using several approaches, primarily Matrix Multiplication (MM) [10], [14], [29], Tensor-network [30], Near-Clifford [31], [32] or WF. In principle, MM conducts vanilla matrix-vector multiplication between operators and state vector, consuming $\mathcal{O}(m \times 2^{2n})$ in time complexity and $\mathcal{O}(2^{2n} + 2^n)$ in space complexity for storing both operator and state vector.

TABLE 1. Surveyed results of related quantum simulators/emulators in the last five years. (†) Estimated qubit numbers based on projected performance given hardware constraints.

Reference	Devices	Benchmarking tasks	#Qubits	Precision	Comparison metrics
Qiskit [8]	CPUs, GPUs, QPUs	QFT, BV, Random Clifford circuits, Hamiltonian simulation, QAOA	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Gate count, Gate depth, Fidelity
ProjectQ [9]	CPUs, GPUs, QPUs	QFT, Shor's Algorithm	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Resource utilization, Circuit depth, Gate count, Fidelity
Cirq [10]	CPUs, GPUs, QPUs	Random Quantum Circuits (RQC)	Depending on hardware resources	32-bit FP	Execution time, Fidelity
TFQ [11]	CPUs, GPUs, QPUs	QAOA, QNN	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Fidelity, Gradient-based optimization
QUBO [12]	CPUs, GPUs	QFT, VQE, GSA, Adiabatic Time Evolution, Quantum Autoencoder, Quantum Classifier	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Fidelity, Overlap
PennyLane [13]	CPUs, GPUs	VQE, QAOA, Quantum Classification	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Fidelity, Variational accuracy
cuQuantum SDK [14]	GPUs	QFT, QAOA, Quantum Volume, Phase Estimation	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Speedup factor, Memory bandwidth utilization
Qsun [15]	CPUs	QLR, QNN, QDP	Depending on hardware resources	32-bit FP, 64-bit FP	Execution time, Fidelity, MSE
Analog-based emulator [16]	UMC-180 nm CMOS Analog	GSA, QFT	6 to 17 qubits	32-bit FP	Execution time, Power consumption, Circuit compatibility
ZCVU9P-based emulator [17]	FPGA(Xilinx ZCVU9P) + CPU(Xeon E5-2686 v4)	QSVM, Quantum Kernel Estimation	Up to 6 qubits	16-bit FX	Execution time, Numerical accuracy, Test accuracy
Alveo-based emulator [18]	FPGA (Xilinx Alveo U250)	QHT, 3D-QHT	10 to 32 qubits (†)	32-bit FP	Execution time, Resource utilization, Circuit depth
Arria-based emulator [19]	FPGA (Arria 10AX115N4F45E3SG)	GSA	Up to 32 qubits (†)	16-bit FX	Emulation time, Resource utilization, Frequency
XCKU-based emulator [20]	FPGA(Xilinx XCKU115)	QFT	Up to 16 qubits	16-bit FX	Execution time
Stratix-based emulator [21], [22]	FPGA(Altera Stratix EP1S80B956C6)	QFT, GSA	Up to 8 qubits	8-bit FX, 16-bit FX	Logic cell usage, Emulation time, Frequency
This work FQsun	**FPGA(Xilinx ZCU102)**	**QFT, PSR (QDP), RQC, QLR, QNN, etc.**	**3 to 17 qubits**	**16-bit FX/FP, 24-bit FX, 32-bit FX/FP**	**Execution time, PDP, Fidelity, MSE, Power, Hardware Utilization**

MM can be accelerated through some strategies, including gate fusion [33], indexing, realized-state representation [34], or sparse matrix calculation. The other approach represents quantum circuits using their corresponding WF [9], [35]. *The WF approach, in theory, requires only $\mathcal{O}(2^n)$ in space complexity for storing only the state vector, which has fewer computational operations and less temporary memory space for conducting quantum operations*. Additionally, the WF approach itself updates only non-zero amplitude elements, which can be considered an inherent form of compression.

B. SIMULATING QUANTUM CIRCUIT BY WAVE-FUNCTION APPROACH

The MM approach operates in two steps. Firstly, $U^{(t)}$ is constructed by tensor-product between g_j and $\mathbb{I}$; later, we conduct MM operation between $U^{(t)}$ and $|\psi^{(t-1)}\rangle$, where $U^{(t)}|\psi^{(t-1)}\rangle = |\psi^{(t)}\rangle$. With the unfavorable, $U^{(t)} \in \mathbb{C}^{N\times N}$, there is an exponential resource scaling of both state vector and operators with increasing #Qubits.

Instead of constructing a large matrix such as $U^{(t)}$, The WF approach uses m gates $\{g_j\}$ directly, took only $2^{n+\hat{n}_j+1}$

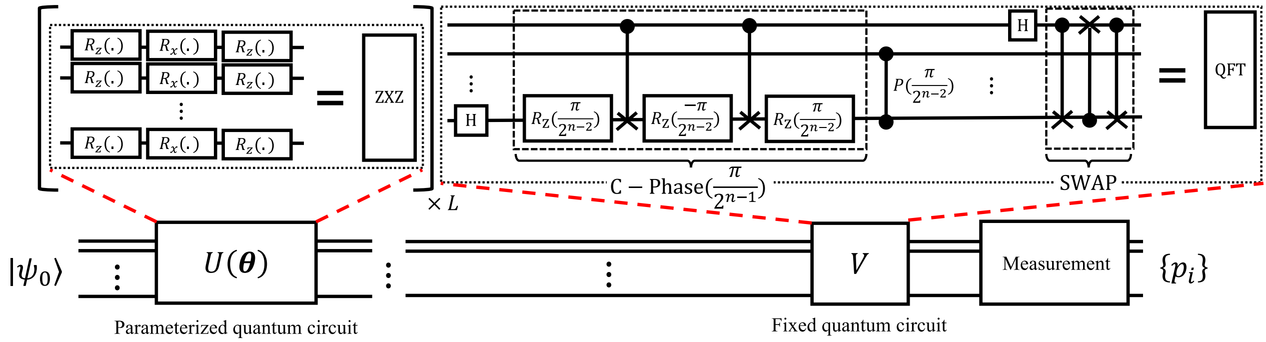


FIGURE 1. Quantum circuit simulation model where it can be divided into parameterized and fixed parts, denoted as $U(\theta)$ and V, respectively (both can be notated as $U^{(t)}$). (Inset left) An example of a parameterized part is the n-qubit ZXZ layer, related to the PSR problem in Section IV-A3. (Inset right) An example of a fixed part is the QFT circuit which includes a series of H, Control-Phase (C-Phase) and SWAP gates, note that a C-Phase can be decomposed into $R_z(.)$ and CX gates.

multiplication and $2^{n+\hat{n}_j+1}$ addition per gate, with $\hat{n}_j \in [1, 2]$ is the number of operand of the gate j; instead of 2^{2n} multiplication and 2^{2n-2} addition following by MM approach. The idea behind this technique is to update only non-zero amplitudes in $|\psi^{(t)}\rangle$ sequentially:

$$|\psi^{(m)}\rangle = \mathcal{W}(g_m) \ldots \mathcal{W}(g_2)\mathcal{W}(g_1)|\psi^{(0)}\rangle. \tag{1}$$

The Equation (1) models the evolution of the quantum state under a sequence of gates, where $g \in \mathcal{G} = \{H, S, CX, R_i(.)\}, i \in \{x, y, z\}$, $\{H, S\}$ and CX are constant 2×2 and 4×4 matrices, respectively. $R_i(.)$ is the 2×2 parameterized matrix with single parameter $\theta \in [0, 2\pi]$. Clifford + R_i is chosen for two purposes. First, the universal gate set known as Clifford + T belongs to the above set as Definition 2, where the T gate is equivalent to the $R_z(\pi/4)$ gate, and the second is that simulating the variational quantum model requires the use of parameterized gates.

Definition 2 (Gate Set for Strong Simulation): Clifford group include $\{H, S, CX\}$ can be used to simulate by the classical algorithm in polynomial time and space, follow Gottesman-Knill theorem [36], which can be extended to Clifford + R_i.

The transition function $\mathcal{W}(g)$ updates $\{\alpha_j\}$ directly depending on the type of gate [37], with the acting on the $(w_0)^{\text{th}}$ qubit as Equation (2):

$$\mathcal{W}(g) : \begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^{w_0}} \end{bmatrix} \rightarrow g \begin{bmatrix} \alpha_{s_i} \\ \alpha_{s_i+2^{w_0}} \end{bmatrix}, \tag{2}$$

where $s_i = \lfloor i/2^{w_0} \rfloor 2^{w_0+1} + (i \bmod 2^{w_0})$, for all $i \in \left[0, 2^{n-1} - 1\right]$. This function requires a much lower number of operations than a matrix-vector multiplication from the MM approach. We unify the implementation of single- and multiple-qubit gates into a common framework; the operation of single-qubit and CX gates are outlined in Algorithm. 1 and Algorithm. 2, respectively.

Algorithm 1 Operation of a Single-Qubit Gate

Require: $\{\alpha_j^{(t)}\}$, target qubit $w_0 \in [0, n)$, $g = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \in \mathcal{G}$
Ensure: $\sum_j |\alpha_j^{(t)}|^2 = 1$
 $n \leftarrow \log_2 |\{\alpha_j\}|$, cut $\leftarrow 2^{n-w_0-1}$
 $|\psi^{(t+1)}\rangle \leftarrow [\alpha_0^{(t+1)}\ \alpha_1^{(t+1)}\ \ldots\ \alpha_{N-1}^{(t+1)}]^\intercal = [0\ 0\ \ldots\ 0]^\intercal$,
 state $\leftarrow [\text{bin}(0, n), \text{bin}(1, n), \ldots, \text{bin}(N-1, n)]$ ▷ bin(i, k) is the function that represent i by k binary bit,
 for $j \leftarrow [0, 1, \ldots, N-1]$ **do**
  if state$[j][w_0] == 0$ **then**
   $\alpha_j^{(t+1)} \leftarrow \alpha_j^{(t+1)} + a \times \alpha_j^{(t)}$
   $\alpha_{j+\text{cut}}^{(t+1)} \leftarrow \alpha_{j+\text{cut}}^{(t+1)} + b \times \alpha_j^{(t)}$
  else
   $\alpha_j^{(t+1)} \leftarrow \alpha_j^{(t+1)} + d \times \alpha_j^{(t)}$
   $\alpha_{j-\text{cut}}^{(t+1)} \leftarrow \alpha_{j-\text{cut}}^{(t+1)} + c \times \alpha_j^{(t)}$
  end if
 end for
 return $\{\alpha_j^{(t+1)}\}$

C. PRELIMINARY CHALLENGES FOR DEVELOPING A HIGH-EFFICIENCY QUANTUM EMULATOR

To develop a quantum emulator that meets the demands of speed, flexibility, and energy efficiency, several core challenges must be addressed. Each of these challenges highlights essential requirements that drive the need for innovative design improvements in quantum emulation systems:

Challenge 1: Efficient Memory Management. Quantum emulation requires managing and processing large datasets, especially as the number of qubits increases, which dramatically expands memory demands. Efficient data handling and minimized memory access times are crucial to achieving high processing speeds without excessive energy consumption.

Challenge 2: Flexible Quantum Gate Configuration. Different quantum algorithms rely on a variety of quantum

Algorithm 2 Operation of CX Gate

Require: $\{\alpha_j^{(t)}\}$, control-target qubits $w_0, w_1 \in [0, n)$, $g = \begin{bmatrix} \mathbb{I} & 0 \\ 0 & \mathbb{X} \end{bmatrix} \in \mathcal{G}$

Ensure: $\sum_j |\alpha_j^{(t)}|^2 = 1$

$n \leftarrow \log_2 |\{\alpha_j\}|$, cut $\leftarrow 2^{n-w_1-1}$

$|\psi^{(t+1)}\rangle \leftarrow [\alpha_0^{(t+1)}\ \alpha_1^{(t+1)}\ \ldots\ \alpha_{N-1}^{(t+1)}]^\intercal = [0\ 0\ \ldots\ 0]^\intercal$,

state $\leftarrow$ [bin(0, n), bin(1, n), ..., bin($N-1$, n)],

for $j \leftarrow [0, 1, \ldots, N-1]$ **do**

  if state[j][w_0] == 1 **then**

    if state[j][w_1] == 1 **then**

      $\alpha_{j-\text{cut}}^{(t+1)} \leftarrow \alpha_{j-\text{cut}}^{(t+1)} + \alpha_j^{(t)}$

    else

      $\alpha_{j+\text{cut}}^{(t+1)} \leftarrow \alpha_{j+\text{cut}}^{(t+1)} + \alpha_j^{(t)}$

    end if

  else

    $\alpha_j^{(t+1)} \leftarrow \alpha_j^{(t)}$

  end if

end for

return $\{\alpha_j^{(t+1)}\}$

gates, but many emulators are constrained by limited gate support. To effectively execute diverse algorithms, an adaptable gate configuration system is needed to facilitate a wide range of computations with minimal hardware reconfiguration.

Challenge 3: Optimized Task Scheduling. Quantum emulators must execute sequential calculations, where delays in one step can impact the entire system's performance. To avoid bottlenecks, the emulator requires an optimized scheduling mechanism to ensure continuous processing without latency, thus maximizing resource utilization and computation speed.

Challenge 4: Multi-Level Precision Support. Quantum algorithms often demand different levels of numerical precision, from FX to FP, depending on application requirements. Supporting various precision levels allows the emulator to balance accuracy with resource usage, adapting performance to meet the specific needs of each quantum task while minimizing computational overhead.

Each of these challenges underscores the necessity of a robust, scalable emulator that can dynamically adjust to complex quantum workloads while maintaining efficiency and precision across diverse applications.

III. PROPOSED ARCHITECTURE

A. SYSTEM OVERVIEW ARCHITECTURE

Figure 2 details the proposed FQsun architecture on an FPGA at the system-on-chip (SoC) level. The system comprises four components: Qsun Framework, FQsun Software, Processing System (PS), and FQsun in Programmable Logic (PL).

The Qsun Framework, implemented in Python, includes libraries and programs for quantum simulation. These programs can instruct the hardware to execute gates through the FQsun C library in FQsun Software. Users proficient in C programming can directly write C programs by calling quantum circuits defined in the FQsun library to achieve higher performance. Within the PS, the embedded CPU runs software such as the Qsun Framework and FQsun Software to control FQsun in PL for executing quantum simulation tasks. In the PL, the proposed FQsun consists of four modules: AXI Mapper, memory system, FQsun controller, and QGU. The AXI Mapper manages and routes data exchanged between the PS and the FQsun memory system. The FQsun memory system comprises Control Buffers, Context Memory, and a pair of Ping/Pong Amplitude Memory modules. Control Buffers store control signals for the host FQsun Controller. Context Memory holds configuration information such as gate type, gate location (cut, $\sin(\theta/2)$, $\cos(\theta/2)$, w_0, w_1), and gate parameters. Notably, the Ping/Pong Amplitude 2×2048 (KB) Memory is the largest memory unit designed to store $\{\alpha_j^{(t)}\}$ for QGU. Finally, QGU contains six fundamental gate units: $\{H, S, CX, R_x, R_y, R_z\}$. These basic gates can be combined to implement most other gates, ensuring high flexibility. The system overview architecture is designed to ensure compatibility between the PS and FQsun in hardware for stable and optimal operation. By supporting the FQsun library in Python and C, users can easily utilize FQsun to perform quantum tasks.

TABLE 2. Supported gates on FQsun, including Clifford + R_j set with additional gates $\{T, X, Y, Z\}$.

Gate	Matrix Representation	Param	Equivalent to
H	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	w_0	Basic gate
S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	w_0	Basic gate
CX	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	w_0, w_1	Basic gate
$R_x(\theta)$	$\begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$	w_0, θ	Basic gate
$R_y(\theta)$	$\begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$	w_0, θ	Basic gate
$R_z(\theta)$	$\begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$	w_0, θ	Basic gate
T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix}$	w_0	$R_z(\pi/4)$
X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	w_0	$R_x(\pi)$
Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	w_0	$R_y(\pi)$
Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	w_0	$R_z(\pi)$

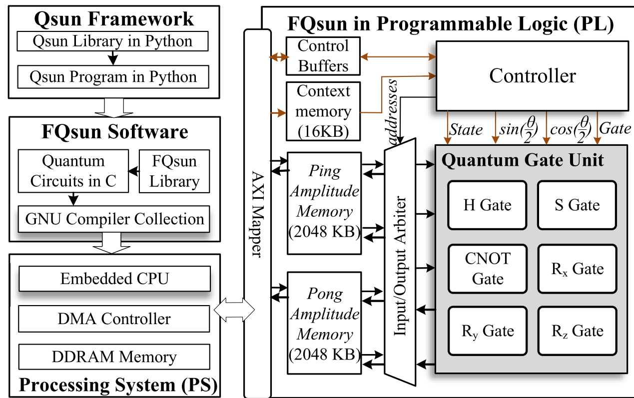


FIGURE 2. Overview architecture of our FQsun at the system-on-chip (Soc) level on FPGA. The arrow stands for dataflow. FQsun in PL communicates with PS through AXI Mapper and DMA Controller.

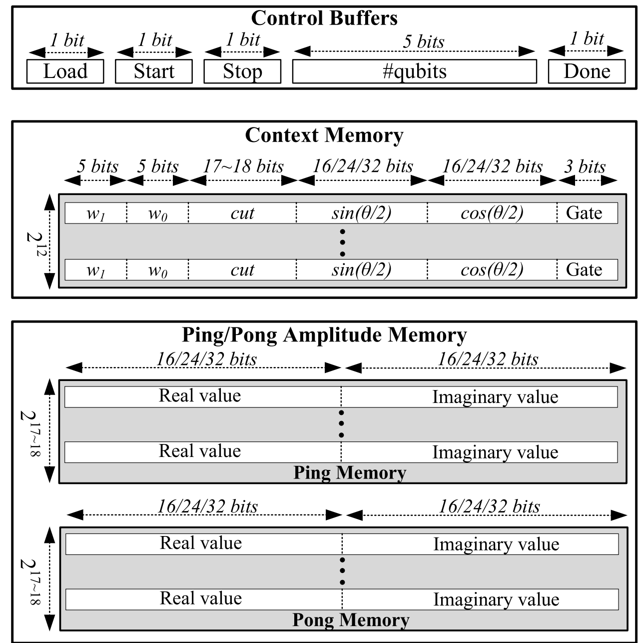


FIGURE 3. Memory organization of the proposed FQsun including (Top) Control buffers (Middle) Context memory and (Bottom) Ping/Pong amplitude memory.

B. EFFICIENT MEMORY ORGANIZATION

As mentioned in Section II-C, matrix size N grows exponentially with #Qubits that the emulator supports. Meanwhile, hardware platforms such as FPGAs are limited by storage resources. To support large #Qubits, the memory organization of FQsun must be optimized. Figure 3 illustrates the memory organization of the proposed FQsun for efficient operation on FPGA. There are three types of memory: Control Buffers, Context Memory, and Ping/Pong Amplitude Memory.

Control Buffers: These buffers store 1-bit control signals such as load, start, done, and stop. The load signal is used by the host to indicate the state where it writes context and amplitude data. Additionally, the host uses a 5-bit signal to indicate #Qubits, which FQsun is supporting. After completing the load state, the host triggers the start signal to begin the session. The done signal is continuously read by the host to track the completion time of each gate operation. Finally, the stop signal is used to end the session by resetting the address register of the Context Memory.

Context Memory (16 KB): This memory stores configuration data for FQsun and has $d = 2048$. The configuration data includes a 3-bit gate opcode indicating the type of gate to be executed. The values $\sin(\theta/2)$ and $\cos(\theta/2)$, with a bit width of 16/24/32 depending on the precision type, are parameters for R_i, pre-computed by FQsun software to reduce hardware complexity. Additionally, parameters such as the 5-bit cut and gate positions $\{w_0, w_1\}$ must also be stored. With this Context Memory design, FQsun can support the execution of up to 2048 gates in a computational sequence.

Ping/Pong Amplitude Memory (2 MB): This memory must have a $d = 2^{17}$ or $d = 2^{18}$ to support up to 17 or 18 qubits for 16-bit precision or 32-bit precision, calibrated to fit the maximum BRAM capacity of the Xilinx ZCU102. In case Ping/Pong Amplitude Memory is applied to FPGAs with more abundant BRAM resources, such as the Alveo or Versal series, FQsun could support a larger #Qubits. To enable amplitude updates as described in Algorithms 1 and 2, a dual-port Ping/Pong memory design is applied. Specifically, if the ping memory holds the current amplitude $\{\alpha_j^{(t)}\}$, the pong memory will store the new amplitude $\{\alpha_j^{(t+1)}\}$, and vice versa. This design is essential on the grounds that, during the calculation of $\{\alpha_{j+cut}^{(t+1)}\}$ and $\{\alpha_{j-cut}^{(t+1)}\}$, both the current and new amplitudes are needed. The dual-port memory architecture was chosen to maximize data throughput and eliminate read/write conflicts during amplitude updates ($\{\alpha_j\}$ and $\{\alpha_{j+cut}\}$) for H, R_x and R_y gates. Using a single-port memory would improve BRAM efficiency but double the latency, which is unacceptable for real-time emulation. Besides, higher-order port memories are not necessarily in this scenario.

C. CONFIGURABLE QUANTUM GATE UNIT (QGU)

Existing quantum emulators face difficulties in supporting a wide variety of quantum simulation applications. This challenge arises due to the limited flexibility of their computational hardware architectures in executing complex gates critical for many applications, thereby restricting their capabilities. Consequently, the QGU hardware architecture is proposed to enable FQsun to achieve the highest level of flexibility in supporting any quantum simulation applications. Supported gates on FQsun, including Clifford + R_i set with additional gates $\{T, X, Y, Z\}$.

The QGU micro-architecture, detailed in Figure 4, comprises five main modules that implement six fundamental gates consistent with the software design. The computational data for the QGU includes $\{\alpha_j^{(t)}\}$, $\{\alpha_j^{(t+1)}\}$, $\{\alpha_{j\pm cut}^{(t+1)}\}$, and parameters ($\sin(\theta/2)$ and $\cos(\theta/2)$) used by R_i.

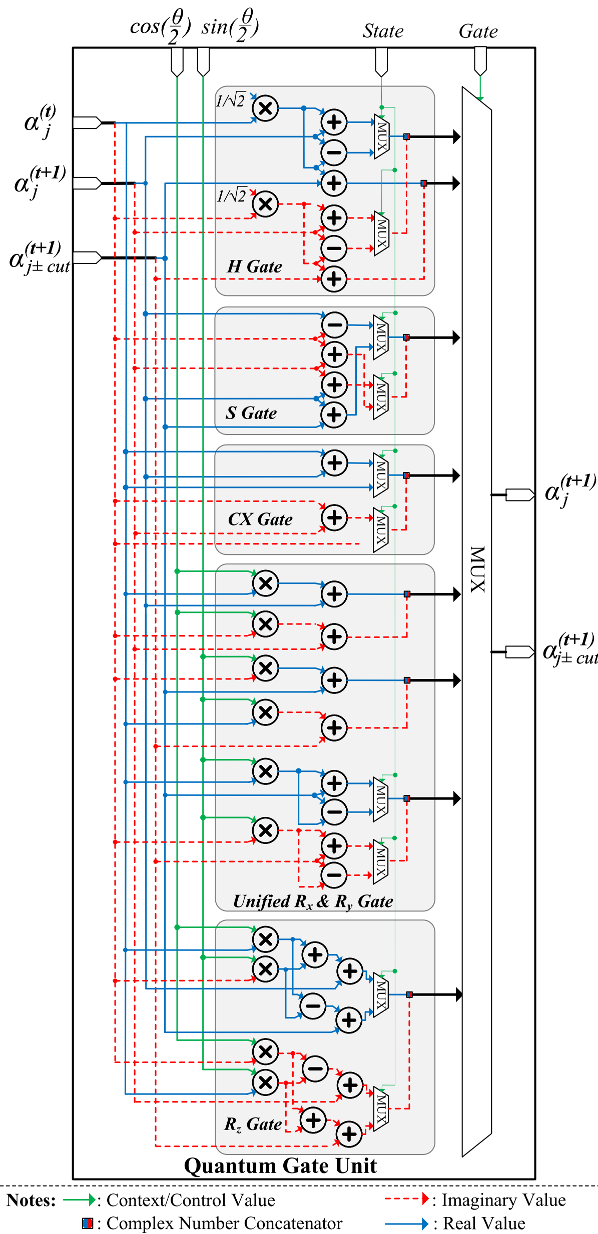


FIGURE 4. **Micro-architecture of QGU, showing modular hardware units for gates. Each module supports amplitude updates using pipelined multiply-accumulate units. Multiplexers route control and data based on gate type and state.**

Essentially, the operations of R_x and R_y are relatively similar; hence, they are implemented using *a Unified R_x & R_y Gate unit* to conserve hardware resources. To ensure proper execution of the `if-else` logic as outlined in Algorithms. 1 & 2, a 2-to-1 multiplexer system controlled by the state value is used to select the output value. Finally, a multiplexer controlled by the gate operation input is employed. By combining the fundamental gates to form other gates, the QGU can be configured to execute all gates utilized by Qsun [15], enabling diverse quantum applications. Moreover, to ensure the highest operating frequency, the operators within the quantum gate modules, such as multiplication and addition, are pipelined to shorten the critical path. Specifically, for FP numbers, multiplication and addition each require two stages, while for FX numbers, multiplication requires two stages, and addition does not require any stage.

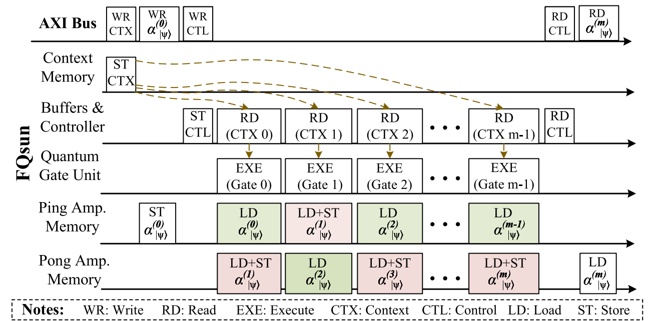


FIGURE 5. **Timing chart of FQsun operation.**

D. OPTIMAL WORKING SCHEDULING

When operating at the SoC level, without a well-defined timing schedule for read and write operations in the memory system, FQsun cannot operate efficiently due to various issues, such as incorrect timing for control signal transmission when the computational data is not available or incorrect data read/write in the Ping/Pong Amplitude Memory.

To address these issues, the timing diagram of FQsun for executing a working session is detailed in Figure 5. At the beginning of a session, context data is stored in the Context Memory, which contains configuration commands for FQsun to execute a quantum circuit consisting of m gates for a specific application. Next, the system writes initial amplitudes $\{\alpha_j^{(\mathbf{0})}\}$ into the Ping Amplitude Memory. The read/write process for amplitude data incurs the highest memory access time because the amplitude vector is large ($N = 2^n$). Once the ping memory is fully loaded with data, the system sends a control signal to initiate the FQsun working session. An internal control register called the Program Counter (PC) is used to track the instruction index to determine the current gate for execution. After reading the instruction from the Context Memory at address PC $= 0$, QGU loads the initial amplitude $\{\alpha_j^{(\mathbf{0})}\}$ from Ping Amplitude Memory and the initial new amplitude $\{\alpha_j^{(\mathbf{1})}\}$ from Pong Amplitude Memory. This data is processed by Gate 0 in N iterations to produce the updated new amplitude $\{\alpha_j^{(\mathbf{0})}\}$, increment the PC for the next instruction and clear the data in Ping Memory (new initial amplitude is zero). At PC $= 1$, QGU proceeds to read the context of Gate 1, read amplitude $\{\alpha_j^{(\mathbf{1})}\}$ from Pong Amplitude Memory, and new amplitude $\{\alpha_j^{(\mathbf{2})}\}$ from Ping Amplitude Memory. After executing and storing the updated amplitude $\{\alpha_j^{(\mathbf{2})}\}$ in Ping Amplitude Memory, the PC is incremented by 1, and Pong Amplitude Memory is cleared. This alternating process continues,

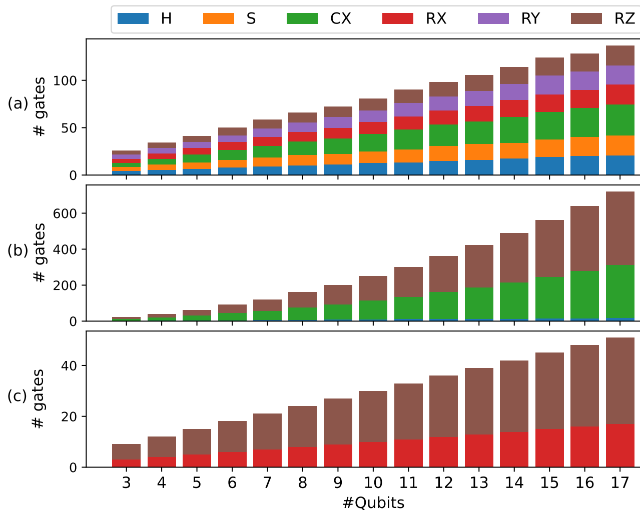


FIGURE 6. Gate distribution for (a) RQC (b) QFT and (c) PSR.

updating the new amplitude data and storing it in Ping/Pong Amplitude Memory with each gate execution. Throughout FQsun's computation, the host PS continuously reads and checks the Done flag to determine when the hardware has completed its task. Accordingly, the host PS reads the final amplitude $\{\alpha_j^{(m)}\}$ from Pong or Ping Amplitude Memory if m is odd or even, respectively. Typically, FQsun's working schedule ensures that the host PS runs the FQsun session accurately and successfully.

E. MULTIPLE NUMBER PRECISIONS

The use of various numerical precisions significantly impacts accuracy, hardware resources, and execution speed. Unfortunately, almost all related work on quantum emulators has primarily focused on performance evaluation while lacking detailed criteria for accuracy metrics. In this study, FQsun is designed with five configurations, each corresponding to a different precision: 16-bit floating point (**FP16**), 32-bit floating point (**FP32**), 16-bit fixed point (**FX16**), 24-bit fixed point (**FX24**), and 32-bit fixed point (**FX32**). A comparison of the architectures, number of pipeline stages, range, and complexity is shown in Table 3. The following discussion elaborates on the key considerations:

Pipeline and Complexity: FP numbers utilize two pipeline stages for both the multiplier and the adder/subtractor units, while FX numbers require only two stages for the multiplier and one stage for the adder/subtractor. FP operation modules are more complex due to handling exponent, mantissa, normalization, and rounding operations. In contrast, FX operations are simpler and involve integer-based arithmetic with fixed multipliers, adders, or subtractors. As a result, the latency of FX design is lower compared to FP design.

Range and Precision: following normalize condition, $|\alpha_j^{(t)}| \in [-1, 1]$; then, FX numbers used in FQsun are configured with 1 sign bit, 1 integer bit, and the remaining

TABLE 3. Preliminary analysis of five suitable number precisions for FQsun.

Format	Pipeline stages	Range (Precision)	Complexity
FP16 (1-bit sign, 5-bit exponent, 10-bit mantissa)	Two for multiplier, two for adder/subtr.	$[6.1 \times 10^{-5}, 6.6 \times 10^{-4}]$ with 10-bit precision	High due to 5-bit exponent, 10-bit mantissa handling
FP32 (1-bit sign, 8-bit exponent, 23-bit mantissa)	Two for multiplier, two for adder/subtr.	$[1.2 \times 10^{-38}, 3.4 \times 10^{38}]$ with 23-bit precision	Higher due to 8-bit exponent, 23-bit mantissa handling, normalization, and rounding
FX16 (1-bit sign, 1-bit integer, 14-bit fractional)	Two for multiplier, one for adder/subtr.	$[-1, 1]$ with 14-bit precision (6.10×10^{-5})	Lower complexity, only integer-based arithmetic with fixed operators
FX24 (1-bit sign, 1-bit integer, 22-bit fractional)	Two for multiplier, one for adder/subtr.	$[-1, 1]$ with 22-bit precision (2.38×10^{-7})	Lower complexity, similar to FX16 but with finer precision due to additional fractional bits
FX32 (1-bit sign, 1-bit integer, 30-bit fractional)	Two for multiplier, one for adder/subtr.	$[-1, 1]$ with 30-bit precision (9.31×10^{-10})	Lower complexity, similar to FX24 but with finer precision due to additional fractional bits

bits allocated to the fractional part to optimize precision. Although FP numbers offer a higher maximum precision compared to FX numbers, they come at the cost of increased hardware resources and execution time. It is essential to consider the required accuracy for applications to determine the most appropriate numerical precision. The required quantization bitwidth needed to achieve target fidelity increases based on the number of qubits, depending on the quantum algorithm. Since $\sum_j |\alpha_j^2| = 1$, $\mathbb{E}[\alpha_j] \propto \frac{1}{2^n}$, leading to linear increase in bitwidth. For example, FX32 is suitable for below 32-qubit algorithms.

Overall, FP numbers provide higher accuracy but incur higher latency and resources, while FX numbers offer lower accuracy with reduced hardware resource usage and latency. To identify the most suitable configuration, detailed measurements will be conducted in the following section.

IV. VERIFICATION AND RESULTS

A. TASKS USED FOR BENCHMARKING

We chose three tasks to benchmark our proposed emulator with #Qubits from 3 to 17. They are motivated by the quantum machine learning applications [4], [38] including the Random Quantum Circuits (RQC) sampling [6], quantum differentiable with Parameter-Shift Rule (PSR) technique [39], and the Quantum Fourier Transform (QFT) as a core algorithmic component to many applications [40]. Since QFT is one of the simple, standard algorithms for benchmarking the performance of quantum simulators and emulators, we test RQC to understand how the proposed universal hardware performs. Finally, PSR is the most popular technique used

TABLE 4. Properties of benchmarking tasks with different gate types and number of gates.

Task	RQC	QFT	PSR
Gate	Clifford+R_i	H, CX, R_z	R_x, R_z
Output	$\{\lvert\alpha\rvert^2\}$	$\{\lvert\alpha\rvert^2\}$	$\{\boldsymbol{\theta}^{(t+1)}, \lvert\alpha\rvert^2\}$
#Gates	$n \times d$	$n\ (H)$ $(n+3)(n-1)\ (CX)$ $\frac{3}{2}n(n-1)\ (R_z)$	$2n\ (R_z)$ $n\ (R_x)$

in the quantum machine learning applications. In Figure 6, we illustrate the gate distribution after transpiling the original circuit by the Clifford+R_i set. The scaling of #gates is polynomial complexity based on #Qubits. The quantum circuit presentation of the PSR and QFT are shown in Figure 1 (Inset left) and (Inset right), respectively. Where the circuit sampling from RQC is the combination of used gates with the smallest depth.

1) RANDOM QUANTUM CIRCUITS (RQC)

The pseudo-RQC is a popular benchmarking problem for quantum systems, the problem was first implemented on the Sycamore chip to prove quantum supremacy [6]. The quantum circuit is constructed gate-by-gate by randomly choosing from a fixed pool and is designed to minimize the circuit depth. The sequences of gates $\{g_j\}$ create a "supremacy circuit" with high classical computational complexity. The RQC dataset from [6] provided the quantum circuit up to 53 qubits, $1,113$ single-qubit gates, and 430 two-qubit gates, then proved that the quantum computer could perform this task in seconds instead of a thousand years on classical computers. However, this dataset limits users when they want to benchmark different cases. Based on our previous work [41], we generated a bigger RQC dataset, which used an extendable gate pool and different circuit construction strategies. The size of the generated dataset is 14 GB of $15,000$ random circuits with a depth range from 1 to 10 and #Qubits from 3 to 17.

2) QUANTUM FOURIER TRANSFORM

Quantum Fourier Transform (QFT) [42] is an important component in many quantum algorithms, such as phase estimation [43], order-finding, and factoring [44]. QFT circuit is utilized from Hadamard (H), controlled phase ($CP(\theta)$), and SWAP gates where $CP(\theta)$ and SWAP can be decomposed into $R_z(\theta)$ and CX gates. These gates transform on an orthonormal basis $\{|0\rangle, \ldots, |N-1\rangle\}$ with the below action on the basis state $|j\rangle \rightarrow \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} e^{2\pi ijk/N}$. If the state $|j\rangle$ is written by using the n-bit binary representation $j = j_1 \ldots j_n$:

$$\mathrm{QFT}(|j_1, \ldots, j_n\rangle) \rightarrow \frac{(|0\rangle + e^{2\pi i 0.j_n}|1\rangle) \otimes \ldots \otimes (|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle)}{\sqrt{N}} \tag{3}$$

where $0.j_l \ldots j_n = j_l/2 + \ldots + j_n/2^{n-l+1}$. We apply the QFT on the zero state $|\mathbf{0}\rangle$, which should return the exact equal superposition of all possible n-qubit states, means $\alpha_j = \alpha_k\ \forall j, k \in [0, N)$.

3) QUANTUM DIFFERENTIABLE PROGRAMMING (QDP) WITH PARAMETER-SHIFT RULE (PSR)

Starting with $ZXZ(\boldsymbol{\theta})$, we try to optimize associated cost value $C(\boldsymbol{\theta})$, which is known as the target of the optimization process by first-order optimizers such as Gradient Descent (GD), for example:

$$C(\boldsymbol{\theta}) = \sum_{j=0}^{N-1} j|\alpha_j|^2;\ \boldsymbol{\theta}_j' \leftarrow \boldsymbol{\theta}_j - \gamma \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \tag{4}$$

where $\boldsymbol{\theta}$ is updated by the general PSR technique [39]. Because the parameterized gate set is limited as a one-qubit rotation gate, only 2-term PSR is used. The gradient is notated as $\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \{\partial_{\theta_j} C(\boldsymbol{\theta})\}_{j=0}^{m-1}$ and partial derivative $\partial_{\theta_j} C(\boldsymbol{\theta})$ is presented in Equation (5) with $\boldsymbol{e}_j$ is the jth-unit vector:

$$\partial_{\theta_j} C(\boldsymbol{\theta}) = \frac{1}{\sqrt{2}}[C(\boldsymbol{\theta} + \frac{\pi}{2}\boldsymbol{e}_j) - C(\boldsymbol{\theta} - \frac{\pi}{2}\boldsymbol{e}_j)] \tag{5}$$

The optimal parameter $\boldsymbol{\theta}^*$ can be obtained through $2m \times n_{\text{iter}}$ quantum evaluations after n_{iter} iterations.

B. COMPARISON METRICS

Essentially, using the proposed simulators without knowledge about their metrics, such as error rate and execution time, can lead to bias during the experiment. Therefore, we propose two types of metrics: accuracy for reliability and performance for comparison between FQsun and other emulators/simulators.

Accuracy metrics: As mentioned in Section II, the target of the quantum emulator is to simulate the transition function $\mathcal{W}(g_j)$, receives $|\psi^{(t-1)}\rangle$ and returns $|\psi^{(t)}\rangle$. To measure the similarity between computational states ρ and theoretical state σ, trace fidelity $\mathcal{F}$ is defined as:

$$\mathcal{F}(\rho, \sigma) = \left(\mathrm{Tr}(\sqrt{\sqrt{\rho}\sigma\sqrt{\rho}})\right)^2 \tag{6}$$

where the theoretical state σ is presented in double precision. Since computing the square root of a positive semi-definite matrix consumes a lot of resources, the fidelity can be reduced to $|\langle\psi_\rho \mid \psi_\sigma\rangle|^2$ for $\rho = |\psi_\rho\rangle\langle\psi_\rho|$ and $\sigma = |\psi_\sigma\rangle\langle\psi_\sigma|$. $\mathcal{F}(\rho, \sigma) = 1$ means $\rho \equiv \sigma$.

Since there is a limited number of decimals on the quantum emulator, it makes the systematic error on amplitudes $\{\alpha_j\}$ increase as system size. Because $|\{\alpha_j\}| = 2^n$ and $\sum_j |\alpha_j|^2 = 1$, then $\mathbb{E}[\alpha_j] = 1/2^n$ decrease exponentially based on #Qubits. MSE is calculated as:

$$\mathrm{MSE}(|\psi_\rho\rangle, |\psi_\sigma\rangle) = \frac{1}{2^n}\left(\sum_j |\alpha_j^\rho - \alpha_j^\sigma|^2\right) \tag{7}$$

to get the accumulated error of all amplitudes.

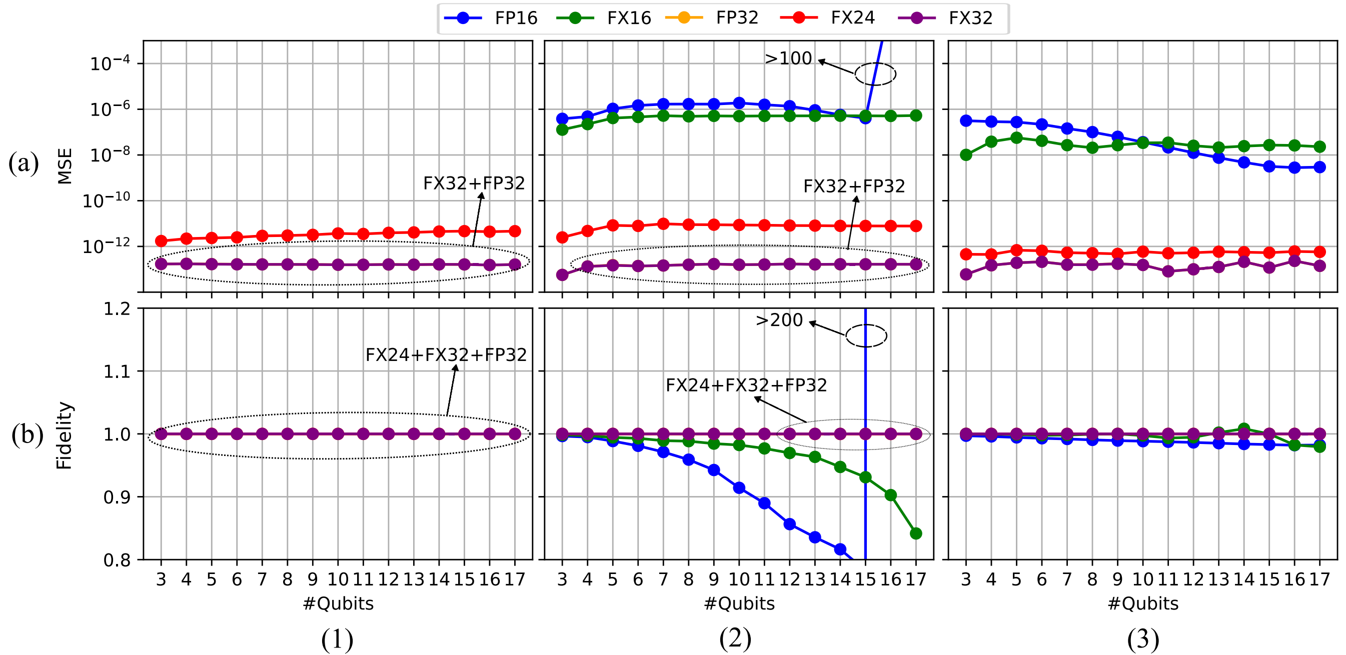


FIGURE 7. (a) $\text{MSE}_{\text{FQsun}}$ and (b) $\mathcal{F}_{\text{FQsun}^*}$. All three tasks include (1) RQC (at d=10), (2) QFT, and (c) PSR. In all cases, FP32 and FX32 are overlapped. For QFT, the FP16 version performs the wrong results since it can not present any $\alpha_j \leq 1/2^{16} \approx 10^{-5}$.

Performance metrics: The novelty of FQsun is the high performance, measured via execution time, power-delay product (PDP), and memory efficiency. The execution time (t) is counted by creating a quantum circuit to receive amplitudes. PDP is simply equal to Power (Joules) $\times$ t. When the designed quantum emulator consumes extra-low power compared with CPU/GPU, this metric is important for verifying the performance of FQsun.

C. IMPLEMENTATION AND RESOURCE UTILIZATION ON FPGA

The FQsun designs were implemented using Verilog HDL and realized on a 16nm Xilinx ZCU102 FPGA, utilizing Vivado 2021.2 to obtain synthesis and implementation results. The system underwent testing through five designs with different precision: **FP16**, **FX16**, **FX24**, **FP32**, **FX32**. For convenience in presentation, we denote the following sets: **Software** = {PennyLane, Qiskit, ProjectQ, Qsun}, **FQsun** = {**FP16**, **FX16**, **FX24**, **FP32**, **FX32**} and **FQsun*** = **FX24**, **FP32**, **FX32**}.

Subsequently, three primary applications as described in Section IV-A were implemented to demonstrate computational efficiency and high flexibility. System accuracy across each design will be recognized by considering fidelity $\mathcal{F}$ and MSE. Consequently, the efficiency and practicality of each FQsun quantum emulator design will be clearly illustrated, alongside detailed presentations of quantum simulator packages on powerful CPUs. Table 5 provides a detailed report on the utilization of **FQsun** when implemented on the Xilinx ZCU102, based on the number of lookup tables (LUTs), flip-flops (FFs), Block RAMs (BRAMs), and digital signature processors (DSPs). Accordingly, the designs utilize between 9,226 and 18,093 LUTs, 1,440 to 7,031 FFs, 344 to 464 BRAMs, and 14 to 88 DSPs. It should be noted that the **FP16** and **FX16** versions exhibit the highest BRAM utilization due to their capacity to accommodate up to 18 qubits. Conversely, the remaining configurations (**FQsun***) support maximally 17 qubits, resulting in a reduced BRAM requirement. The significantly lower DSP utilization compared to FP numbers demonstrates that applying FX numbers to the FQsun emulator results in greater power efficiency due to each DSP being equivalent to multiple LUTs in terms of implementation.

Overall, **FX32** utilizes the most LUTs, while **FP32** utilizes the most FFs and DSPs. **FX16** and **FP16** utilize the fewest resources. Additionally, all FQsun versions are designed to avoid utilizing DSPs to conserve hardware resources.

D. GATE SPEED

The properties of gate speed are presented in Table 6. The execution time per gate is given by $\#\text{Period}_{\text{version}} \times \#\text{Cycle} \times 2^n$ for looping through amplitudes, which depends on the version, gate type, and number of qubits, respectively. These results highlight the trade-off between precision and speed across different versions. For example, while the **FP16** version has the shortest period of 6.66 ns, the **FX32** version requires a longer period of 9.35 ns to execute a gate, reflecting the additional time needed to achieve higher precision. Furthermore, the FX versions generally require fewer cycles per gate compared to FP versions, with reductions ranging from 1.5 to 2 times depending on the gate type. For example, the CX gate requires 4 cycles in **FX16**

TABLE 5. Utilization of FQSun on ZCU102 FPGA.

Design	Name	LUTs	FFs	BRAM	DSP
FP16	Buffers & AXI Mapper	268	290	0	0
	Context Memory	4,428	0	0	0
	Ping & Pong Memories	3,071	36	464	0
	QGU	5,530	2,752	0	44
	FQsun Controller	937	973	0	0
	Total	**14,234**	**4,051**	**464**	**44**
FP32	Buffers & AXI Mapper	295	327	0	0
	Context Memory	6,556	0	0	0
	Ping & Pong Memories	3,796	36	456	0
	QGU	6,258	5,296	0	88
	FQsun Controller	1,188	1,372	0	0
	Total	**18,093**	**7,031**	**456**	**88**
FX16	Buffers & AXI Mapper	264	289	0	0
	Context Memory	4,387	0	0	0
	Ping & Pong Memories	3,168	36	464	0
	QGU	457	142	0	14
	FQsun Controller	990	973	0	0
	Total	**9,266**	**1,440**	**464**	**14**
FX24	Buffers & AXI Mapper	272	307	0	0
	Context Memory	5,722	0	0	0
	Ping & Pong Memories	3,086	24	344	0
	QGU	684	206	0	28
	FQsun Controller	939	828	0	0
	Total	**10,703**	**1,635**	**344**	**28**
FX32	Buffers & AXI Mapper	297	327	0	0
	Context Memory	6,888	0	0	0
	Ping & Pong Memories	3,934	36	456	0
	QGU	914	326	0	56
	FQsun Controller	1,136	1,372	0	0
	Total	**13,169**	**2,061**	**456**	**56**

TABLE 6. Basic gate's period and number of cycles per gate on FQsun.

Version	Period (ns)	Number of Cycles per Gate					
		H	S	CX	R_x	R_y	R_z
FP16	6.66	6	4	4	6	6	8
FP32	7.35	6	4	4	6	6	8
FX16	7.35	4	2	2	4	4	4
FX24	8.00	4	2	2	4	4	4
FX32	9.35	4	2	2	4	4	4

but 6 cycles in **FP16**, a difference that becomes critical in high-speed applications. This efficiency in FX versions could be advantageous in time-sensitive quantum computations, where lower #Cycles allow for faster gate operations while maintaining adequate precision. Therefore, depending on the application, a balance between speed (fewer #Cycles) and accuracy (higher precision) can be strategically chosen to optimize performance.

E. MEAN-SQUARE ERROR EVALUATION

To demonstrate the accuracy, the measured MSE of **FQsun** on the real-time ZCU102 FPGA system is shown in Figure 7. Based on the analyzed data, the designs **FQsun*** exhibit the lowest MSE values, making them the most suitable candidates for use in quantum emulation. Specifically, **FP32** and **FX32** demonstrate extremely low MSE values ranging from 5.686×10^{-14} to 1.656×10^{-13}, indicating greatly high accuracy. **FX24** also shows acceptable MSE values, ranging from 2.481×10^{-12} to 7.798×10^{-12}, which are sufficient for large quantum applications. In contrast, **FP16** and **FX16** do not meet the required accuracy standards. **FP16** displays significantly high MSE values, with MSE reaching 0.055 at $n = 16$ and spiking to 129.824 at $n = 17$, rendering it unsuitable for precise quantum applications. Similarly, **FX16** shows higher MSE values, ranging from 1.278×10^{-7} to 5.327×10^{-7}, which are inadequate for applications demanding high precision. Thus, the three designs in **FQsun*** with the lowest MSE: **FX24**, **FP32**, and **FX32**, will be considered for comparison in the next sections.

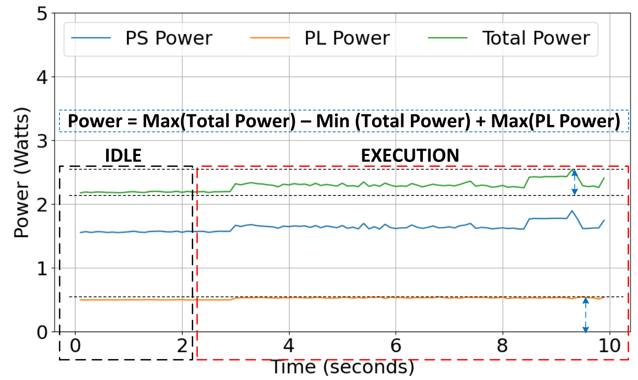


FIGURE 8. Power measurement of FQsun (FX32) on RQC.

F. FIDELITY EVALUATION

MSE offers a basic accuracy estimate, but fidelity is a more precise metric for assessing quantum simulation performance. The results for RQC, as shown in Figure 7 (1), emphasize the accuracy gap between the three designs. Specifically, $\mathcal{F}_{\mathbf{FP32/FX32}(\mathrm{RQC})} \in [1 - 7.6 \times 10^{-7}, 1 + 1.2 \times 10^{-7}]$, remaining measurably close to 1. Both designs demonstrate superior stability and precision, making them the best performers among the three. By contrast, $\mathcal{F}_{\mathbf{FX24}(\mathrm{RQC})} \in [1 - 7.6 \times 10^{-7}, 1 - 2 \times 10^{-6}]$ shows a broader range of fidelity. For QFT and PSR in Figure 7 (2)-(3), demonstrate again that $\mathcal{F}_{\mathbf{FP32/FX32}(\mathrm{QFT/PSR})} \in [1 - 7 \times 10^{-6}, 1 + 2.4 \times 10^{-5}]$, have prominently small variations. Both designs exhibit stable performance, with values consistently close to ideal accuracy. This positions **FP32** and **FX32** as the optimal choices in terms of fidelity. On the other hand, $\mathcal{F}_{\mathbf{FX24}(\mathrm{QFT/PSR})} \in [1 - 6 \times 10^{-5}, 1 + 3 \times 10^{-5}]$, still shows slightly larger variations. While **FX24** maintains a high level of accuracy, these more noticeable deviations suggest a lower stability compared to **FP32** and **FX32**.

To maintain the accuracy of FPGA, we list all the feasible data structures. We also evaluate the impact of data structure on the above tasks. Consistent with the commentary from [17], there is a trade-off between high fidelity and

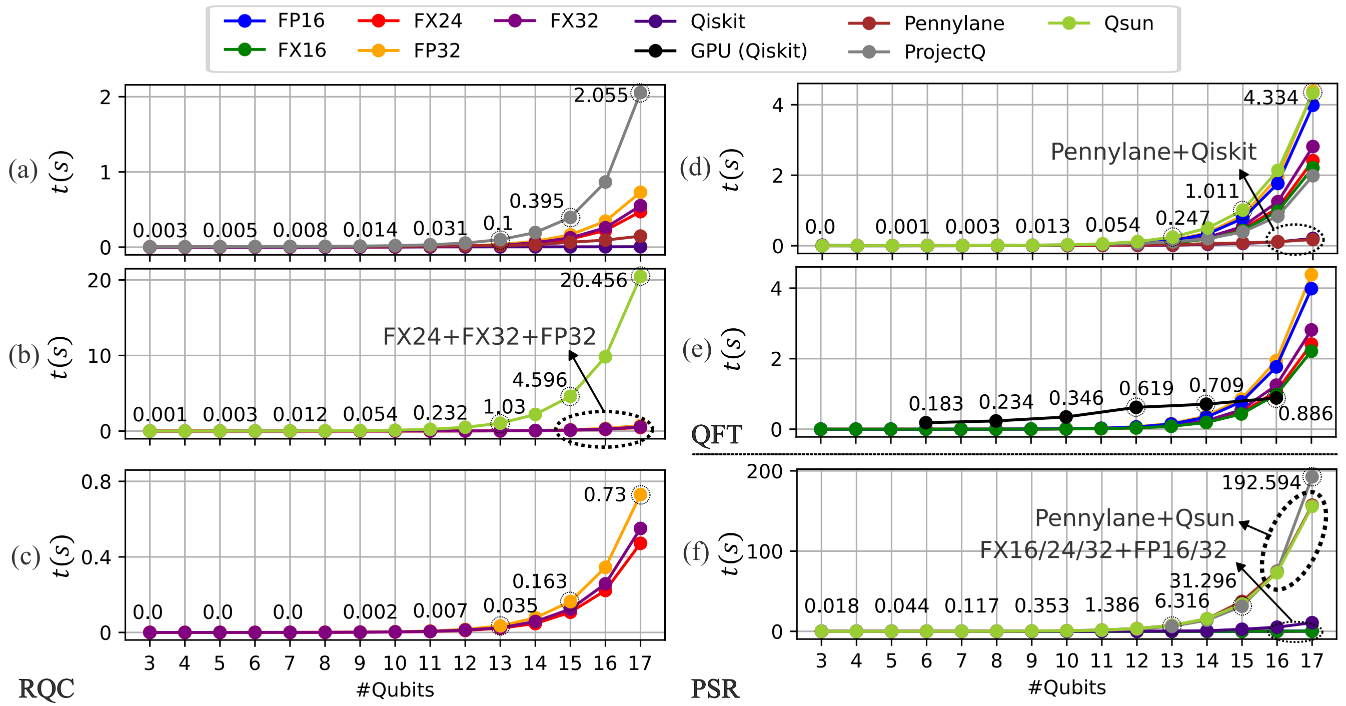


FIGURE 9. The execution time on (a-c) RQC, compare between (a) **Software**/{Qsun} + **FQsun***, (b) Qsun + **FQsun*** and (c) **FQsun***. *d*=10. (d) QFT execution time comparison between **Software** + **FQsun**. (e) We compare GPU-accelerated Qiskit's state-vector simulator [45] and FQsun, as Qiskit is a leading high-performance quantum simulation package. Since the A100 data is only available on even #Qubits, we only plot until *n*=16. FQsun demonstrates a clear speed advantage for $n < 15$. (f) PSR execution time comparison between **Software** + **FQsun**. The figures on the plot from (a) to (f) are from ProjectQ, Qsun, FP32, Qsun, GPU (Qiskit) and ProjectQ, respectively.

quantization bit-width. As the small scale, the required bit-width for $\mathcal{F} \approx 1$ is low but increases fast based on #Qubits [46].

G. DETAILED POWER CONSUMPTION ANALYSIS

To accurately measure power consumption in real time, the INA226 sensor on the ZCU102 FPGA was utilized. Detailed results are shown in Figure 8, where the power consumption values for the PS, PL, and total power are presented when **FP32** executes RQC. Specifically, the PS consumes a maximum of 1.84 W (with 0.34 W dynamic power). The PL consumes a maximum of 0.61 W. The total power reaches a maximum of 2.41 W (with 0.23 W dynamic power). Accordingly, the power consumption for FQsun is determined by adding the total dynamic power to the maximum PL power, reaching 0.81 W. These power values are kept for all #Qubits.

H. COMPARISON WITH RELATED SOFTWARE ON POWERFUL CPU/GPU

To demonstrate the speed advantage of FQsun, the simulation time of **FQsun*** is presented and compared with **Software**. All software and hardware trials are run on an Intel i9-10940X CPU @ 3.30GHz at least 100 times then take the average. During testcases, no data corruption, simulation error, or system resets occurred. We validated metrics with known Qiskit outputs, ensuring correctness over runs.

The comparisons in Figure 9 (a-c) are conducted with PSR on different sets of versions. In Figure 9 (a), **FQsun** shows comparable execution speed with most software simulators at $n < 13$. When compared with Qsun, **FQsun*** shows faster computational speed, especially for $n > 11$, demonstrating a significant improvement over the corresponding software version. Finally, a comparison between **FQsun*** versions reveals that the FX version provides faster computational speed, with the difference increasing gradually based on #Qubits. Overall, FQsun offers better processing speed than ProjectQ and Qsun emulators and is slower than Qiskit and PennyLane. At the same time, FX versions are considered to have the best processing speed. Depending on the accuracy requirements, an appropriate version can be selected.

Figure 9 (d-e) shows the mean of the execution time when simulating QFT and PSR. In the QFT benchmark, **FQsun** is slower than Qiskit and PennyLane for $n \in [11, 17]$. However, it demonstrates superior speed compared to other software simulations and is slightly faster than Qiskit. For larger #Qubits, the GPU's performance is slightly better due to its parallel processing property. The highlight of FQsun is its significantly higher energy efficiency, consuming less than 1 W of power, while GPUs typically consume hundreds. In Figure 9 (f), Qiskit still achieves top performance but is lower than all **FQsun** versions, while Pennylane and Qsun are overlapped; ProjectQ got the worst performance with 192.598 (s) at $n = 17$.

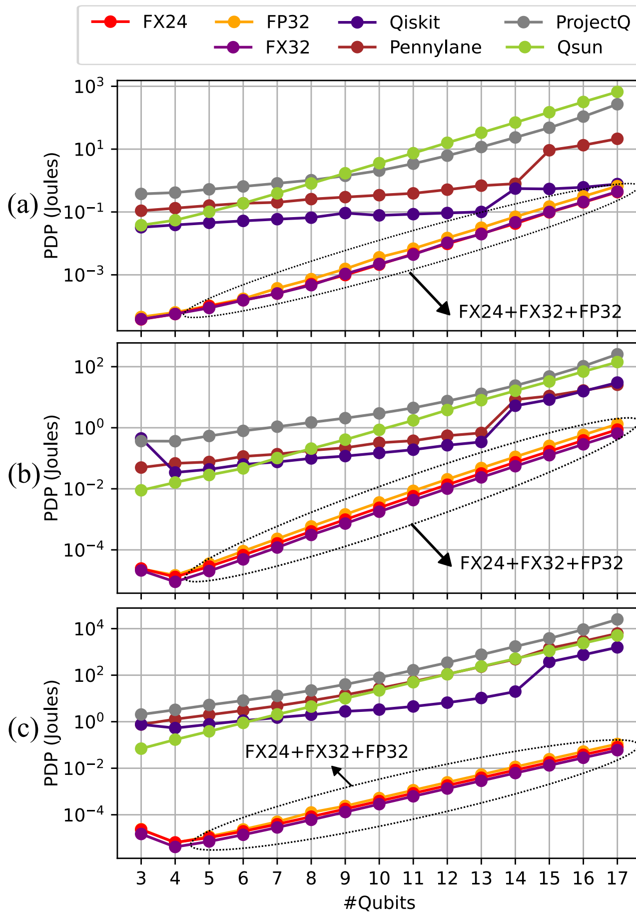


FIGURE 10. **PDP on Software + FQsun* through (a) RQC (b) QFT and (c) PSR. The PDP comparison results show that FQsun has optimal energy efficiency, making it suitable for the long term to save costs.**

To further demonstrate energy efficiency, Figure 10 presents a comparison of PDP between **FQsun*** and four software simulations. Accordingly, FQsun achieves significantly better PDP than software simulations in all three tasks. In QFT, FQsun provides a PDP from 1.30×10^{-5} to 1.31, better than **Software** from $\mathbf{3.62 \times 10^{2}}$ to $\mathbf{4.0 \times 10^{4}}$ times; and from 4.14×10^{-6} to 8.04×10^{-2}, better than **Software** from $\mathbf{3.18 \times 10^{3}}$ to $\mathbf{7.84 \times 10^{5}}$ times when simulating PSR. Finally, for RQC, FQsun achieves from 2.56×10^{-5} to 3.72×10^{-1}, better than **Software** from $\mathbf{1.66 \times 10^{0}}$ to $\mathbf{9.87 \times 10^{3}}$ times.

I. COMPARISON WITH QISKIT BASED ON STABILIZER ORDER

The results from the previous section show that Pennylane and Qiskit can achieve better execution times than **FQsun** in RQC and QFT tasks from $n > 10$. Pennylane and Qiskit are the multi-paradigm simulation packages, hence, it will use the best paradigm for each type of circuit rather than the full state-vector as ProjectQ, Qsun, and **FQsun**. Therefore, Pennylane and Qiskit will provide different gate speeds for different circuits. The paradigm is decided based on the stabilizer order property, which ranges from 1 to 4^{n}. The stabilizer order relates to the complexity of the circuit; the more non-Clifford on qubits, the higher the stabilizer order. Using stabilizer formalism, we can simulate low-order circuits like QFT in only 0.11 (s) at $n = 64$ [28], much better than the state-vector approach because QFT uses non-Clifford gates on only the last qubits. In the case of PSR, the high-order circuits use many non-Clifford gates, which require the state-vector simulation, and then Pennylane and Qiskit are slower than FQsun.

For a fair comparison, we evaluate Qiskit and **FX32** on 10-qubit $W_{\text{chain}} + XYZ$ ansatz as shown in Figure 11 (a) to make sure that all simulators follow the state-vector paradigm. Figure 11 (b) demonstrates how the gap between Qiskit and our emulator changed based on the stabilizer order. Execution times are measured again for each additional gate. The alternating action of CNOT and non-Clifford gates increases the order based on #Gates, until 4^{10} for all stabilizers. Notice that the number of stabilizers is equal to #Qubits. The results show that **FX32** achieve the slope better than Qiskit **10.71** times, take only 1.96×10^{-3} (s) compared to 2.1×10^{-2} (s) at #Gate = 400. It can be explained that in the case of $W_{\text{chain}} + ZXZ$, the stabilizer is quickly increased to maximum between #Layers = 2 (#Gates = 80) and #Layers = 3 (#Gates = 120).

J. COMPARISON WITH HARDWARE-BASED QUANTUM EMULATORS

Table 7 presents a comparison between FQsun and existing FPGA-based quantum emulators [17], [19], [20], [24], [25] in terms of frequency, precision, execution time,#Gates, and normalized gate speed. #Qubits stand for the maximum of supported #Qubits. Note that Table 7 is limited by differences in platform; we normalize the evaluation where possible using normalized gate speed and provide fair contextual benchmarks. All figures are taken on their own FPGA.

Compared to other quantum emulators [16], [17], [21], [22], [23], our emulator supports computation with a higher #Qubits (17 compared to under 5), demonstrating enhanced applicability for various applications. The work [21], [22], [23] achieve the best normalized gate speed from 10^{-10} to 10^{-9} (s). However, the hardware design is fixed for #Qubits such as 3; then, it only takes one cycle per gate application. Furthermore, the reported execution time is measured on simulation and does not implement SoC. The work [17] and [25] also show a lower execution time than our work. Again, these designs support only the QFT circuit and then have no use case because they provide a fixed output.

For a fair comparison, FQsun is comparable to other flexible emulators that can potentially support a wide range of quantum algorithms, including Arria 10AX115N4F45E3SG [19] and AMD Xilinx Zynq-7000 FPGA [24]. Regarding execution time, **FX32** achieves a computation speed-up of **17.9** times and **6.1** times compared to [19] and [24], respectively. Furthermore, FQsun supports computations across five precision types and offers different performance and accuracy.

TABLE 7. Comparative analysis of FQsun and existing FPGA-based emulators on QFT's performance.

Works	Device	Frequency (MHz)	Flexibility	Precision	#Qubits	Execution time (s)	#Gates †	NGS ††
[17]	Xilinx XCVU9P	233	✗	18-bit FX	16	1.20×10^{-3}	-	-
[19]	Arria 10AX115N4F45E3SG	233	✓	32-bit FP	16	1.84×10^{1}	528	5.33×10^{-7}
[20]	Xilinx XCKU115	160	✗	16-bit FX	16	2.70×10^{-1}	136	3.03×10^{-8}
[24]	AMD Xilinx Zynq-7000	100	✓	32-bit FX	6	1.15×10^{-4}	10	1.80×10^{-7}
[25]	2 × Intel Stratix 10 MX2100	299	✗	32-bit FP	30	4.47×10^{0}	465	8.95×10^{-12}
This work	Xilinx ZCU102	150	✓	16-bit FP	18	8.90×10^{0}	810	4.19×10^{-8}
		136	✓	32-bit FP	17	4.30×10^{0}	721	4.55×10^{-8}
		136	✓	16-bit FX	18	4.90×10^{0}	810	2.31×10^{-8}
		125	✓	24-bit FX	17	2.41×10^{0}	721	2.52×10^{-8}
		125	✓	32-bit FX	17	2.81×10^{0}	721	2.97×10^{-8}

† The #Gate in this work is higher than other work due to the no-use of control-rotation gates.
†† The **N**ormalized **G**ate **S**peed (NGS) (s / (gate × amplitude)) = Execution time / (#Gates × $2^{\#\text{Qubits}}$), smaller is better.

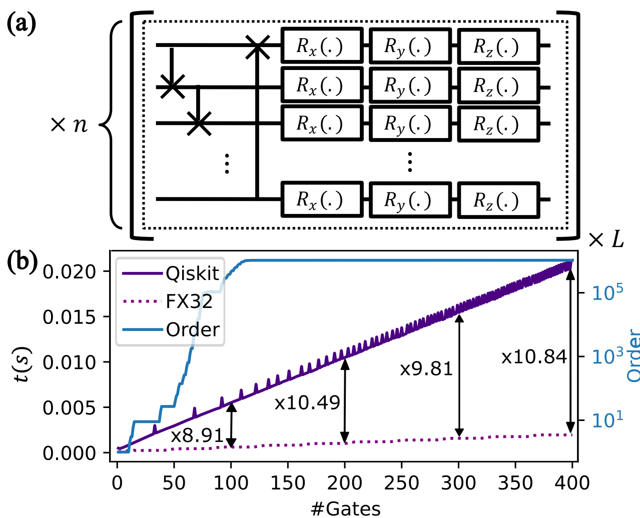


FIGURE 11. (a) 10-layer $|W_{\text{chain}} + XYZ(\theta)\rangle$ is used for benchmarking in Section IV-I (b) The execution time between Qiskit and FX32 based on the stabilizer order (in logarithm scale), their corresponding slopes are 5.1×10^{-5} and 4.8×10^{-6}.

V. CONFIGURATIONS BEYOND 17/18 QUBITS

In case the memory for $\{\alpha_j\}$ is larger than the amount of BRAM, the state vector must be partitioned into smaller fragments (2^{17} elements per chunk), stored temporarily in BRAM, and swapped in/out from external DDRAM. In this model, BRAM acts as a cache for partial state vectors, and DDRAM provides bulk storage, enabling emulation of #Qubits > 17. The throughput model changes from:

$$\text{TP}_{\#\text{Qubits}\leq 17} = \text{Total transferred bit} / (t_w + t_r + t_e \times m) \quad (8)$$

to:

$$\text{TP}_{\#\text{Qubits}>17} = \text{Total transferred bit} / ((t_w + t_r + t_e) \times m) \quad (9)$$

due to increased memory swapping; where t_w, t_r and t_e represent the times for writing to, reading from, and executing a single gate on FQsun, respectively. The total transferred bits depend on #Qubits, precision, and algorithm. For example, simulating a 17-qubit QFT circuit (**FX32**) with 721 gates transfers $2^{17} \times 64$ (bits) $\times 721$ (gates) $\approx 6.02 \times 10^{10}$ (bits) $\approx$ 7.52 (GB) in 2.81 (s) (see Table 7). The FQsun's maximum throughput following Equation (8) is 2.15 (Gb/s). On the ZCU102 FPGA, the throughput between DDRAM and FQsun is 40 (Gb/s). Hence, FQsun remains scalable to higher #Qubits, limited primarily by available DDRAM capacity (4 GB) and external memory bandwidth. However, the trade-off is that gate execution time increases linearly with the number of memory swaps per gate. This follows Equation (9) instead of Equation (8), reducing throughput compared to fully in-BRAM operation.

Alternatively, scalability can be achieved by migrating to larger FPGAs with significantly larger BRAM capacity, such as Xilinx Alveo U280 with 72 (MB) of BRAM. In this approach, the state vector is fully resident on-chip, preserving current throughput. Resource usage still scales exponentially with #Qubits due to the $\mathcal{O}(2^n)$ state space, but the pipeline latency and QGU performance remain unchanged. Supporting 19 qubits in **FX32** would require $2 \times 2^{19} \times 64$ (bits) $\approx$ 64 (MB) of BRAM. Such capacity is within reach of high-end FPGA platforms, and the core design of FQsun remains portable to these platforms with minor adjustments in memory mapping.

VI. CONCLUSION

In conclusion, the FQsun demonstrates significant advancements in quantum emulators, delivering improved speed, accuracy, and energy efficiency compared to traditional software simulators and existing hardware emulators. Leveraging optimized memory architecture, a configurable QGU, and support for multiple number precisions, FQsun efficiently bridges the gap between simulation flexibility and hardware constraints. Experimental results reveal that FQsun achieves high fidelity and minimal MSE across various quantum tasks,

outperforming comparable emulators on execution time and PDP. The emulator's flexibility in supporting diverse quantum gates enables versatile applications, positioning FQsun as a reliable platform for advanced quantum simulation research. Moreover, the energy efficiency demonstrated by FQsun is particularly noteworthy in an era where sustainable computing is a priority.

While FQsun, like all wave-function-based quantum simulators, remains fundamentally bounded by the exponential scaling of state-vector size with increasing #Qubit; our hardware design proves that a quantum emulator can be more efficient than the same type that runs on CPU/GPU. While our current implementation is on the Xilinx ZCU102, the underlying architecture is designed to scale. Future work will focus on porting FQsun to more advanced FPGA platforms, such as Xilinx Alveo or Versal series, to support larger quantum systems. Additionally, we are exploring the integration of software acceleration techniques, such as gate-fusion [33], accelerated PSR [47], and realized-state representation [34], to further enhance performance.

DATA AVAILABILITY

The codes and data used for this study are available at https://github.com/NAIST-Archlab/FQsun.

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Rev.*, vol. 41, no. 2, pp. 303–332, Jan. 1999.

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput.*, 1996, pp. 212–219.

[3] Z. Zhou, Y. Du, X. Tian, and D. Tao, "QAOA-in-QAOA: Solving large-scale MaxCut problems on small quantum machines," *Phys. Rev. Appl.*, vol. 19, no. 2, Feb. 2023, Art. no. 024027.

[4] W. Guan, G. Perdue, A. Pesah, M. Schuld, K. Terashi, S. Vallecorsa, and J.-R. Vlimant, "Quantum machine learning in high energy physics," *Mach. Learn., Sci. Technol.*, vol. 2, no. 1, Mar. 2021, Art. no. 011003.

[5] S. Bravyi, A. W. Cross, J. M. Gambetta, D. Maslov, P. Rall, and T. J. Yoder, "High-threshold and low-overhead fault-tolerant quantum memory," *Nature*, vol. 627, no. 8005, pp. 778–782, Mar. 2024.

[6] F. Arute et al., "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019.

[7] D. Bluvstein et al., "Logical quantum processor based on reconfigurable atom arrays," *Nature*, vol. 626, no. 7997, pp. 58–65, Feb. 2024.

[8] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, "Quantum computing with qiskit," 2024, *arXiv:2405.08810*.

[9] D. S. Steiger, T. Häner, and M. Troyer, "ProjectQ: An open source software framework for quantum computing," *Quantum*, vol. 2, p. 49, Jan. 2018.

[10] S. V. Isakov, D. Kafri, O. Martin, C. V. Heidweiller, W. Mruczkiewicz, M. P. Harrigan, N. C. Rubin, R. Thomson, M. Broughton, K. Kissell, E. Peters, E. Gustafson, A. C. Y. Li, H. Lamm, G. Perdue, A. K. Ho, D. Strain, and S. Boixo, "Simulations of quantum circuits with approximate noise using qsim and cirq," 2021, *arXiv:2111.02396*.

[11] M. Broughton et al., "TensorFlow quantum: A software framework for quantum machine learning," 2021, *arXiv:2003.02989*.

[12] P. Date, D. Arthur, and L. Pusey-Nazzaro, "QUBO formulations for training machine learning models," *Sci. Rep.*, vol. 11, no. 1, p. 10029, May 2021.

[13] V. Bergholm et al., "PennyLane: Automatic differentiation of hybrid quantum-classical computations," 2018, *arXiv:1811.04968*.

[14] H. Bayraktar et al., "CuQuantum SDK: A high-performance library for accelerating quantum science," in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, vol. 1, Sep. 2023, pp. 1050–1061.

[15] Q. C. Nguyen, L. B. Ho, L. Nguyen Tran, and H. Q. Nguyen, "Qsun: An open-source platform towards practical quantum machine learning applications," *Mach. Learning: Sci. Technol.*, vol. 3, no. 1, Mar. 2022, Art. no. 015034.

[16] S. Mourya, B. R. La Cour, and B. Datta Sahoo, "Emulation of quantum algorithms using CMOS analog circuits," *IEEE Trans. Quantum Eng.*, vol. 4, pp. 1–16, 2023.

[17] S. Liang, Y. Lu, C. Guo, W. Luk, and P. H. J. Kelly, "PCQ: Parallel compact quantum circuit simulation," in *Proc. IEEE 32nd Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, May 2024, pp. 24–31.

[18] E. El-Araby, N. Mahmud, M. J. Jeng, A. MacGillivray, M. Chaudhary, Md. A. I. Nobel, S. I. U. Islam, D. Levy, D. Kneidel, M. R. Watson, J. G. Bauer, and A. E. Riachi, "Towards complete and scalable emulation of quantum algorithms on high-performance reconfigurable computers," *IEEE Trans. Comput.*, vol. 72, no. 8, pp. 2350–2364, Aug. 2023.

[19] 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, no. 9, pp. 1017–1037, Sep. 2020.

[20] Y. Hong, S. Jeon, S. Park, and B.-S. Kim, "Quantum circuit simulator based on FPGA," in *Proc. 13th Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2022, pp. 1909–1911.

[21] M. Aminian, M. Saeedi, M. Saheb Zamani, and M. Sedighi, "FPGA-based circuit model emulation of quantum algorithms," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Apr. 2008, pp. 399–404.

[22] A. U. Khalid, Z. Zilic, and K. Radecka, "FPGA emulation of quantum circuits," in *Proc. IEEE Int. Conf. Comput. Design, VLSI Comput. Processors*, Oct. 2004, pp. 310–315.

[23] 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, no. 1, 2016, Art. no. 5718124.

[24] A. Silva and O. G. Zabaleta, "FPGA quantum computing emulator using high level design tools," in *Proc. Eight Argentine Symp. Conf. Embedded Syst. (CASE)*, Aug. 2017, pp. 1–6.

[25] H. M. Waidyasooriya, H. Oshiyama, Y. Kurebayashi, M. Hariyama, and M. Ohzeki, "A scalable emulator for quantum Fourier transform using multiple-FPGAs with high-bandwidth-memory," *IEEE Access*, vol. 10, pp. 65103–65117, 2022.

[26] T. Suzuki, T. Miyazaki, T. Inaritai, and T. Otsuka, "Quantum AI simulator using a hybrid CPU–FPGA approach," *Sci. Rep.*, vol. 13, no. 1, p. 7735, May 2023.

[27] C. Huang, M. Newman, and M. Szegedy, "Explicit lower bounds on strong quantum simulation," *IEEE Trans. Inf. Theory*, vol. 66, no. 9, pp. 5585–5600, Sep. 2020.

[28] J. Mei, M. Bonsangue, and A. Laarman, "Simulating quantum circuits by model counting," in *Computer Aided Verification*, A. Gurfinkel and V. Ganesh, Eds., Cham, Switzerland: Springer, 2024, pp. 555–578.

[29] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable quantum computing and development with a high-level DSL," in *Proc. Real World Domain Specific Lang. Workshop*, New York, NY, USA, Feb. 2018, pp. 1–10.

[30] D. Strano, B. Bollay, A. Blaauw, N. Shammah, W. J. Zeng, and A. Mari, "Exact and approximate simulation of large quantum circuits on a single GPU," in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, vol. 1, Sep. 2023, pp. 949–958.

[31] É. Descamps and B. Dakic, "On the stabilizer formalism and its generalization," *J. Phys. A, Math. Theor.*, vol. 57, no. 45, Oct. 2024, Art. no. 455301.

[32] C. Gidney, "Stim: A fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, Jul. 2021.

[33] T. Häner and D. S. Steiger, "0.5 petabyte simulation of a 45-qubit quantum circuit," in *Proc. SC17: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, Nov. 2017, pp. 1–10.

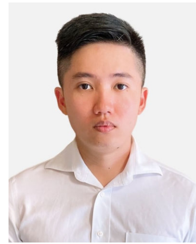
[34] K.-S. Jin and G.-I. Cha, "QPlayer: Lightweight, scalable, and fast quantum simulator," *ETRI J.*, vol. 45, no. 2, pp. 304–317, Apr. 2023.

[35] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya, T. Yamamoto, T. Yan, T. Kawakubo, Y. O. Nakagawa, Y. Ibe, Y. Zhang, H. Yamashita, H. Yoshimura, A. Hayashi, and K. Fujii, "Qulacs: A fast and versatile quantum circuit simulator for research purpose," *Quantum*, vol. 5, p. 559, Oct. 2021.

[36] S. Aaronson and D. Gottesman, "Improved simulation of stabilizer circuits," *Phys. Rev. A, Gen. Phys.*, vol. 70, no. 5, Nov. 2004, Art. no. 052328.
[37] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "QuEST and high performance simulation of quantum computers," *Sci. Rep.*, vol. 9, no. 1, p. 10736, Jul. 2019.
[38] M. Schuld and F. Petruccione, *Machine Learning With Quantum Computers*, vol. 676. Cham, Switzerland: Springer, 2021.
[39] D. Wierichs, J. Izaac, C. Wang, and C. Y.-Y. Lin, "General parameter-shift rules for quantum gradients," *Quantum*, vol. 6, p. 677, Mar. 2022.
[40] D. Wakeham and M. Schuld, "Inference, interference and invariance: How the quantum Fourier transform can help to learn from data," 2024, *arXiv:2409.00172*.
[41] T. H. Vu, V. T. D. Le, H. L. Pham, and N. Yasuhiko, "Efficient random quantum circuit generator: A benchmarking approach for quantum simulators," in *Proc. Int. Conf. Comput. Commun. Technol. (RIVF)*, 2024, pp. 1–18.
[42] D. Coppersmith, "An approximate Fourier transform useful in quantum factoring," 2002, *arXiv: quant-ph/0201067*.
[43] A. Y. Kitaev, "Quantum measurements and the Abelian stabilizer problem," 1995, *arXiv: quant-ph/9511026*.
[44] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, Nov. 1994, pp. 124–134.
[45] A. J. Gangapuram, A. Läuchli, and C. Hempel, "Benchmarking quantum computer simulation software packages: State vector simulators," *SciPost Phys. Core*, vol. 7, no. 4, p. 075, Nov. 2024.
[46] A. Laing, A. Peruzzo, A. Politi, M. R. Verde, M. Halder, T. C. Ralph, M. G. Thompson, and J. L. O'Brien, "High-fidelity operation of quantum photonic circuits," *Appl. Phys. Lett.*, vol. 97, no. 21, Nov. 2010, Art. no. 211109.
[47] V. Tuan Hai, L. Vu Trung Duong, P. Hoai Luan, and Y. Nakashima, "Efficient parameter-shift rule implementation for computing gradient on quantum simulators," in *Proc. Int. Conf. Adv. Technol. Commun. (ATC)*, Oct. 2024, pp. 449–454.

TUAN HAI VU (Member, IEEE) received the B.S. degree in software engineering and the M.S. degree in computer science from the University of Information Technology, Vietnam National University, in 2021 and 2023, respectively. He is currently pursuing the Ph.D. degree with the Architecture Laboratory, Nara Institute of Science and Technology, Japan. His research interests include quantum simulation acceleration and quantum machine learning.

VU TRUNG DUONG LE (Member, IEEE) received the Bachelor of Engineering degree in IC and hardware design from Vietnam National University Ho Chi Minh City (VNU-HCM)—University of Information Technology (UIT), in 2020, and the master's degree in information science and the Ph.D. degree from Nara Institute of Science and Technology (NAIST), Japan, in 2022 and 2024, respectively. He is currently an Assistant Professor with the Computing Architecture Laboratory, NAIST. His research interests include computing architecture, reconfigurable processors, and accelerator design for quantum emulators and cryptography.



HOAI LUAN PHAM (Member, IEEE) received the bachelor's degree in computer engineering from Vietnam National University Ho Chi Minh City—University of Information Technology (UIT), Vietnam, in 2018, and the master's and Ph.D. degrees in information science from Nara Institute of Science and Technology (NAIST), Japan, in 2020 and 2022, respectively. Since October 2022, he has been with NAIST, as an Assistant Professor; and also with UIT, as a Visiting Lecture. His research interests include blockchain technology, cryptography, computer architecture, circuit design, and accelerators.

QUOC CHUONG NGUYEN received the B.S. degree in theoretical physics from the University of Natural Sciences, Vietnam National University, in 2018. He is currently pursuing the Ph.D. degree with the State University of New York at Buffalo, Buffalo, NY, USA. His research interests include quantum machine learning and algorithms for discrete optimization problems.

YASUHIKO NAKASHIMA (Senior Member, IEEE) received the B.E., M.E., and Ph.D. degrees in computer engineering from Kyoto University, in 1986, 1988, and 1998, respectively. He was a Computer Architect with the Computer and System Architecture Department, Fujitsu Ltd., from 1988 to 1999. From 1999 to 2005, he was an Associate Professor with the Graduate School of Economics, Kyoto University. Since 2006, he has been a Professor with the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include computer architecture, emulation, circuit design, and accelerators. He is a fellow of IEICE, a Senior Member of IPSJ, and a member of IEEE CS and ACM.