EVMx: An FPGA-Based Accelerator for Smart Contract Processing

Joel Poncha Lemayian, *Member, IEEE*, Ghyslain Gagnon, *Senior Member, IEEE*, Kaiwen Zhang, *Member, IEEE*, and Pascal Giard, *Senior Member, IEEE*

***Abstract*—Ethereum leverages smart contracts (SCs) to power decentralized applications (dApps), with execution handled by the Ethereum virtual machine (EVM) within an Ethereum client. Other blockchain platforms, including Avalanche, Polkadot, Aurora, and Cardano, have also adopted the EVM. However, the performance of the EVM is often constrained by the limitations of general-purpose processors, a challenge that has been explored in the literature. This work aims to further address the limitation by proposing EVMx, a dedicated single-core SC execution engine implemented on a field programmable gate array (FPGA). EVMx follows a processor-like architecture inspired by the RISC philosophy. By exploiting the parallelism and high-speed processing capabilities of FPGA hardware, EVMx achieves a 61% to 99% reduction in execution time for commonly used operation codes compared to traditional central processing unit (CPU)-based environments. Furthermore, EVMx executes entire Ethereum blocks with a percentage reduction in execution time between 6% and 56% against comparable FPGA implementations and 98% to 99% compared to CPU-based EVMs in the literature. These results demonstrate the potential of EVMx to significantly accelerate SC execution and enhance the performance of EVM-compatible blockchains.**

***Index Terms*—Blockchain, cryptocurrency, Ethereum, Ethereum virtual machine (EVM), field programmable gate array (FPGA), hardware acceleration, smart contracts (SCs).**

I. Introduction

THE Ethereum blockchain pioneered the concept of smart contracts (SCs), transforming the blockchain ecosystem by enabling programmable, self-executing agreements. This innovation laid the foundation for the Web3 ecosystem and significantly expanded blockchain use cases beyond simple transactions [1]. This advancement is primarily enabled by Ethereum's introduction of the turing-complete Ethereum virtual machine (EVM), which acts as the execution environment for SCs. In contrast to Bitcoin, which functions mainly as a digital currency, Ethereum offers a versatile platform that allows developers to build and deploy decentralized applications (dApps), tokenize assets, and create decentralized finance (DeFi) ecosystems [2].

Subsequently, many emerging blockchain platforms adopted the Ethereum model to become EVM-compatible. This compatibility enables developers to utilize the existing Ethereum ecosystem, including programming languages, wallet software, and plugins to build and deploy applications across multiple blockchains. Notable examples of EVM-compatible blockchains include Avalanche, Polkadot, and Cardano [3].

The above blockchain paradigms operate on a peer-to-peer network composed of various nodes, each classified by its functionality and resource requirements. For example, synchronization (sync) time defines the time taken by a node to download the most up-to-date copy of the blockchain. In most EVM-compatible networks, the three primary types of nodes are light nodes, full nodes, and archival nodes [4].

Light nodes synchronize only the block headers, providing them with limited access to blockchain data. However, they can still communicate with other nodes to retrieve necessary information and validate blocks. Light nodes are the most resource-efficient option, requiring minimal hardware, energy, and EVM interaction [5].

Conversely, full nodes represent a mid-tier option. They synchronize the blockchain history up to the most recent 128 blocks. These nodes actively participate in block validation, allowing them to execute SCs and query the blockchain. Running a full node demands moderate memory (MEM), power, and EVM usage, especially during initialization, where the latest 128 blocks must be locally rebuilt [5].

Archival nodes offer the most comprehensive functionality by synchronizing the entire blockchain history from the genesis block. This allows instant access to any data, including historical information such as account balances at specific block heights. As a result, many dApps rely on archival nodes to retrieve such data efficiently, as well as propose and validate blocks. Archival nodes require significant storage (STR) and computational resources. Their initialization involves the re-execution of all historical blocks, a process that often takes days or weeks, leading to substantial EVM utilization [6]. All node types handle the same account structures, which include two main types: externally-owned accounts (EOAs) and contract accounts [7]. EOAs are controlled by users via private keys and can initiate transactions, while contract accounts are controlled by SCs' code deployed on the network. While both account types can interact with each other, only transactions involving contract accounts extensively use the

Received 14 July 2025; revised 25 September 2025 and 27 October 2025; accepted 29 October 2025. Date of publication 6 November 2025; date of current version 23 January 2026. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through an Alliance Grant in partnership with Quantum eMotion under Grant ALLRP-588315. *(Corresponding author: Joel Poncha Lemayian.)*

Joel Poncha Lemayian, Ghyslain Gagnon, and Pascal Giard are with the Department of Electrical Engineering, École de technologie supérieure (ÉTS), Montreal, QC H3C 1K3, Canada (e-mail: joel-poncha.lemayian.1@ens.etsmtl.ca; pascal.giard@etsmtl.ca).

Kaiwen Zhang is with the Department of Software Engineering and IT, ÉTS, Montreal, QC H3C 1K3, Canada (e-mail: kaiwen.zhang@etsmtl.ca).

Digital Object Identifier 10.1109/TVLSI.2025.3628118

EVM to execute SC bytecode. Notably, a recent analysis indicates that 59% of Ethereum transactions involve SC execution, reflecting the extensive utilization of the EVM. Moreover, the remaining 41% moderately use the EVM to check signatures, update account balances, and apply state changes [8].

As discussed above, sync time, block proposal, and block validation processes utilize the EVM. Within full and archival nodes, these processes are computationally intensive and can represent a performance bottleneck, lowering the transactions per second (TPS) of EVM-compatible networks [9].

A. Related Works

Several approaches have been proposed in the literature to improve the performance and efficiency of the EVM. For instance, the work in [10] proposes PaVM, a virtual machine that executes and validates SCs in parallel. PaVM enhances and extends a runtime system to perform SCs' parallel execution. Moreover, the work in [11] proposed SmartVM, a SC virtual machine that accelerates the computation of on-chain deep neural networks (DNNs). SmartVM performs parallel SC execution at the instruction level to facilitate high-performance on-chain DNN executions.

In addition, [12] and [13] introduce a blockchain processing unit (BPU) and an SC unit (SCU), respectively, both are field programmable gate array (FPGA)-based EVM implementations. These designs incorporate schedulers, decoders, and interpreters to evaluate and reorder SC operational codes (opcodes) for optimized execution, leveraging pipelining and parallelism to enhance performance. These works improve the execution time of SCs compared to central processing unit (CPU)-based EVMs.

The works in [10] and [11] remain software-based, and thus may not fully exploit the intrinsic performance advantage of hardware implementations. On the other hand, the hardware designs in [12] and [13] rely on extensive decoding and scheduling prior to executing SC code. However, SC opcodes are typically highly interdependent [14], so the cost of analyzing thousands of opcodes for parallelism often exceeds the benefit of executing a few concurrently.

B. Contributions

This work proposes EVMx, a single-core FPGA-based EVM that offloads the execution of SCs in full and archival nodes to a dedicated hardware device. EVMx adopts a processor-like architecture inspired by the RISC philosophy and implements an instruction set architecture (ISA) fully compliant with the official EVM specification. Specifically, the contributions of this work are as follows.

1) Comprehensive hardware design and implementation: a resource-efficient hardware architecture that preserves EVM semantics and stack (STK)-based model, while remaining fully compatible with bytecode generated by standard SC compilers such as Remix. The complete synthesizable VHDL implementation is made publicly available in a GitHub repository to promote transparency and reproducibility [15].
2) Optimization techniques for execution performance: strategies such as lightweight pipelined logic, simplified opcode decoding, on-the-fly special handling of corner cases, and selective exploitation of parallelism within the sequential EVM model to achieve significant speedups.
3) Extensive evaluation demonstrating performance gains: detailed experimental results showing substantial speedups for individual opcodes and entire blocks compared to CPU-based EVMs and existing FPGA implementations.
4) Insight into scalable integration with Ethereum clients: discussions on potential pathways to embed EVMx into existing Ethereum communities, addressing practical deployment challenges.

Preliminary results of this work appeared in the Proceedings of the IEEE Annual International Computer Software and Applications Conference (COMPSAC), in the Blockchain and Distributed Ledger Technologies (BlockDLT 2025) Workshop, held in July 2025 [16]. This journal version presents a comprehensive architectural redesign, an enhanced FPGA implementation, and a detailed performance analysis that significantly extends the scope and depth of the original workshop article.

C. Outline

The remainder of this work is organized as follows. Section II presents an overview of the EVM, describing its core components and functionality. Section III introduces the proposed hardware architecture of EVMx, including a detailed explanation of its input/output (I/O) system. Section IV outlines the design of EVMx's core components, while Section V focuses on the execution of SCs and opcodes, notably, `JUMPI`, `CREATE2`, and `KECCAK256`. Section VI presents implementation results and compares the execution times of various opcodes and blocks to demonstrate performance improvements. It also discusses the integration of EVMx into an Ethereum client, outlines its potential limitations, and examines its power requirements. Finally, Section VII concludes the work.

II. EVM System Overview

This section provides the necessary background on Ethereum to help the reader understand the role of the EVM in EVM-compatible blockchains. Also, we discuss the EVM itself and highlight the challenges that motivated this work.

A. Ethereum Blockain Overview

The Ethereum blockchain was introduced in Vitalik Buterin's white paper, proposing key contributions compared to preceding blockchain paradigms [1]. Among others, the main contribution was the introduction of the Turing-complete EVM, allowing Ethereum to support a wide range of computations. Moreover, in September 2022, Ethereum transitioned from the proof-of-work (PoW) to the proof-of-stake (PoS) consensus algorithm in a process called the "Merge to address scalability and efficiency challenges" or simply "the Merge" [17]. Unlike PoW, which is highly computationally intensive, PoS greatly reduces energy consumption, increases TPS, and improves scalability, albeit at a cost of added complexity and centralization [18].

After the merge, the structure of Ethereum was split into two layers: the execution layer and the consensus layer. The execution layer validates and executes all transactions. The consensus layer is responsible for achieving consensus among validators. Therefore, Ethereum nodes may run two clients, which are software running the consensus layer and execution layer protocols. Validators are responsible for proposing and validating new blocks of transactions in the Ethereum network. Therefore, they must run both execution and consensus clients. To become a validator, participants must stake Ethers (ETHs), the native currency of Ethereum, as collateral. Ethereum nodes that do not participate in staking only run the execution client. The staking mechanism incentivizes honest behavior, as validators risk losing their stake if they act maliciously [13].

As part of their responsibilities, validators participate in a structured process of block production governed by Ethereum's PoS protocol. Specifically, since the Merge, Ethereum structures time into epochs. In each epoch, a validator is randomly selected from the pool of eligible validators to propose a block. Validators who are not chosen to propose a block are assigned to committees to attest and confirm the new block [19]. Validators who serve as either proposers or attestors within the committee are required to locally execute all transactions in a block using an execution client. This execution typically involves the use of the EVM. As a result, block processing can become a significant performance bottleneck in the system.

To carry out this task, validators rely on execution clients such as Go-Ethereum (Geth) [20], Besu [21], and Nethermind [22]. Among these, Nethermind provides the fastest sync time, but at the cost of high random access MEM (RAM) usage. It utilizes techniques such as Snap sync and Fast sync, which download the leaf nodes and block headers of the Ethereum blockchain, respectively, and reconstruct the intermediate nodes locally [22]. Geth, the oldest and most widely adopted client, is valued for its stability and broad support. In contrast, Besu is primarily favored by the Hyperledger community due to its features for enterprise and permissioned networks [23]. Conversely, consensus clients like Lighthouse [24] and Teku [25] manage block finalization and validator coordination.

As previously introduced in Section I, Ethereum nodes are broadly categorized into full, light, and archival nodes. Among these, full and archival nodes heavily depend on the EVM during both initial sync and ongoing block processing. This reinforces the critical role of the EVM in node performance. Therefore, accelerating SC execution within the EVM is a promising approach to mitigating bottlenecks and improving the overall efficiency of Ethereum clients.

B. Ethereum Architecture

This subsection discusses two fundamental components of the Ethereum architecture: the Ethereum SCs and the EVM.

1) Smart Contracts: SCs are contractual agreements written in a programming language such as Solidity [26]. SCs are uploaded onto the blockchain network, as bytecode, where they automatically execute when predefined conditions are met. They eliminate the need for a trusted third party in business processes, thereby reducing costs, saving time, and minimizing risks [27]. To deploy these contracts, developers first write them in a given programming language, then the compiler compiles them into EVM bytecode. This bytecode is a sequence of opcodes that are deployed on the blockchain and are executed by the EVM. Currently, the EVM supports 140 opcodes, each associated with a specific gas (GS) cost [28]. Table I presents some opcodes, detailing their names, minimum GS requirements, input arguments, data sources, and output destinations.

TABLE I
SAMPLE EVM OPCODES. ADAPTED FROM [28]

Opcode	Name	Min Gas	Arguments	From	To
51	MLOAD	3	Offset	Memory	Stack
52	MSTORE	3	Offset, Value	Stack	Memory
20	KECCAK256	30	Offset, Size	Memory	Stack
0A	EXP	10	a, Exponent	Stack	Stack
1C	SHR	3	Shift, Value	Stack	Stack
02	MUL	5	a, b	Stack	Stack

GS is a small fee that corresponds to the resources consumed by the execution of a given opcode. It is charged by the blockchain network infrastructure to the sender of the SC. The charged fee is primarily used to compensate the validators who process and confirm the transaction [1]. Moreover, the fee is measured in Gwei (gigawei), which is the smallest unit of ETH. The amount of fees paid in ETH is calculated as

$$\text{Fee in ETH} = \frac{\text{Gas Used} \times \text{Gas Price(Gwei)}}{1\ 000{,}000\ 000} \tag{1}$$

where the GS price is the amount the sender is willing to pay per unit GS. The GS fee model serves as a safeguard against denial of service (DoS) attacks by preventing excessive computation [29]. The GS limit is the maximum amount of unit GS a user is willing to spend on a transaction. If the GS limit is exhausted, an $out-of-GS$ exception is thrown by the EVM, halting processing and reverting any changes [13].

However, the GS charged for opcodes interacting with the MEM component within the EVM is calculated differently from the GS for other opcodes. The MEM cost function is defined as [30]

$$C_{mem}(a) = G_{memory} \times a + \left\lfloor \frac{a^2}{512} \right\rfloor \tag{2}$$

where a is the highest MEM word index accessed during execution (1 word = 32 bytes) and G_{memory} is a constant GS fee charged per unit word (currently three GS units). The equation shows that for the first 736 bytes, the cost of accessing the MEM is linear; after that, the cost increases quadratically. This is designed to deter users from excessively using the MEM and prevent DoS attacks. Moreover, the EVM does not charge $C_{mem}(a)$ at every MEM access; instead, it charges the difference between the new and old access ($\Delta C_{mem}(a) = C_{mem}(a_{new}) - C_{mem}(a_{old})$), making the cost incremental [28].

2) Ethereum Virtual Machine: Fig. 1 illustrates the general structure of the EVM. The EVM follows a STK-based architecture and consists of several key components [29].

1) *Stack:* A last-in, first-out (LIFO) structure that holds up to 1024 words of 32 bytes each. It temporarily stores

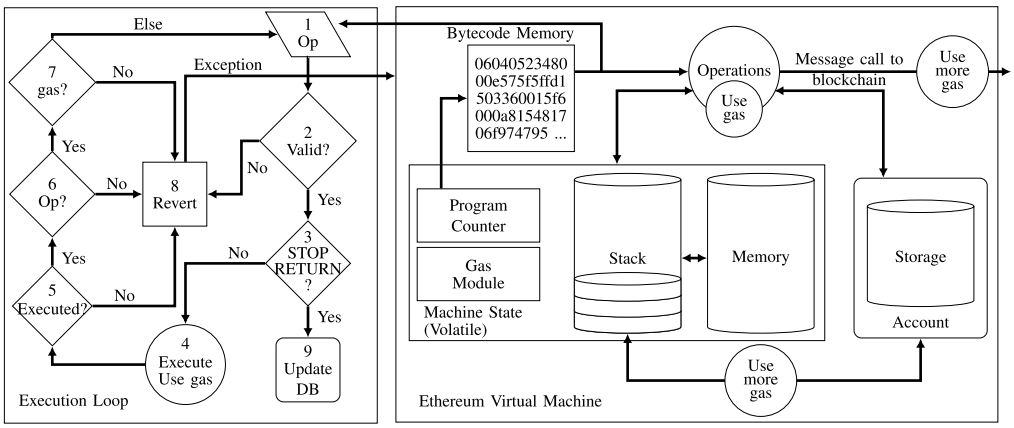

Fig. 1. General structure of the EVM. The execution loop validates and processes opcodes, updating the machine state. The EVM architecture consists of BCM, PC, GS module, STK, MEM, and STR. Adapted from [31].

intermediate results during contract execution. The cost of STK operations is relatively low.

2) *Memory:* A dynamically allocated, temporary STR area used during execution to hold SC data and erased when execution is completed. MEM expansion increases costs, making its usage moderately expensive.
3) *Storage:* A persistent key-value store, where both keys and values are 32 bytes long. Smart contracts (SCs) use STR to maintain state variables across transactions. Due to its permanence, accessing STR is costly.
4) *Bytecode MEM:* A persistent part of a SC's account state, containing the bytecode executed by the EVM.
5) *Program Counter:* A pointer that tracks the current execution position within the SC bytecode, determining which opcode to execute next.
6) *GS Module:* A metering system that calculates and tracks the total execution cost used to process an SC. It measures the cost in GS units.

Before the EVM executes the bytecode of a SC, two main steps are performed. First, when a valid transaction is received, the target address is extracted from the transaction, and the corresponding account is located in the state database. Second, the code field of the target account is checked. If the code field is empty, the transaction is treated as a simple token transfer: the balance is updated directly, and the state database is modified accordingly. However, if the code field contains the bytecode of an SC, the EVM loads this bytecode into a local MEM area and enters the execution loop [13].

The execution loop is illustrated on the left-hand side of Fig. 1. In step 1, each opcode is analyzed to determine the corresponding operation. Step 2 verifies the validity of the opcode: if the opcode is invalid, the EVM proceeds directly to step 8, where the transaction is reverted, an exception is thrown, and execution halts. If the opcode is valid, the EVM continues to step 3, where it checks for the presence of a `STOP` or `RETURN` opcode. These opcodes indicate successful contract execution. If either is encountered, the EVM exits and updates the state database accordingly. If neither is found, the EVM advances to step 4, where it executes the operation by interacting with the necessary components, e.g., using the STK for `PUSH`/`POP` operations or accessing STR for `MSTORE`. The GS cost associated with the opcode is also deducted at this stage. Next, in step 5, the EVM verifies whether the operation is executed successfully. If it did not, the EVM reverts the transaction and throws an exception. If successful, the EVM proceeds to step 6 to check whether additional opcodes remain to be executed. If no further opcodes are available and no valid termination opcode (`STOP` or `RETURN`) has been encountered, the EVM reverts the transaction, indicating that the contract execution did not complete properly. However, if there are still opcodes to be executed, the EVM proceeds to step 7, where it verifies whether sufficient GS remains. If the GS is depleted, the EVM transitions to the revert state. Otherwise, it loops back to step 1 to continue execution.

Fig. 1 also illustrates that message calls to external contracts incur higher GS costs. They facilitate ETH transfer, data exchange, and execution of other contracts, enabling a complex interaction network of SCs [31]. Similarly, STR operations are more expensive because they involve persistent writes to the blockchain's state, which require more computational effort and long-term STR [28].

Section III discusses the proposed architecture of EVMx. It presents the general overview of the proposed architecture, including its core components and its I/O system that enables external interaction.

III. Proposed Hardware Architecture of EVMx

This section presents the proposed architecture of EVMx, shown in Fig. 2. The design incorporates the fundamental components of the EVM, as illustrated in Fig. 1, including the bytecode MEM (BCM), GS module, STK, MEM, STR, program counter (PC), Keccak-256 (KEC) hash function, and the arithmetic logic unit (ALU). These components work together to support the execution of SCs within EVMx.

Beyond its core components, EVMx includes several I/O interfaces designed to facilitate communication with external systems. These interfaces serve as the main link between EVMx and client software, enabling data exchange and control signals to be transmitted. Further details on this interaction are provided in Section VI-E.

The `extData` input is an array that provides the initialization EVM code executed at the beginning of contract

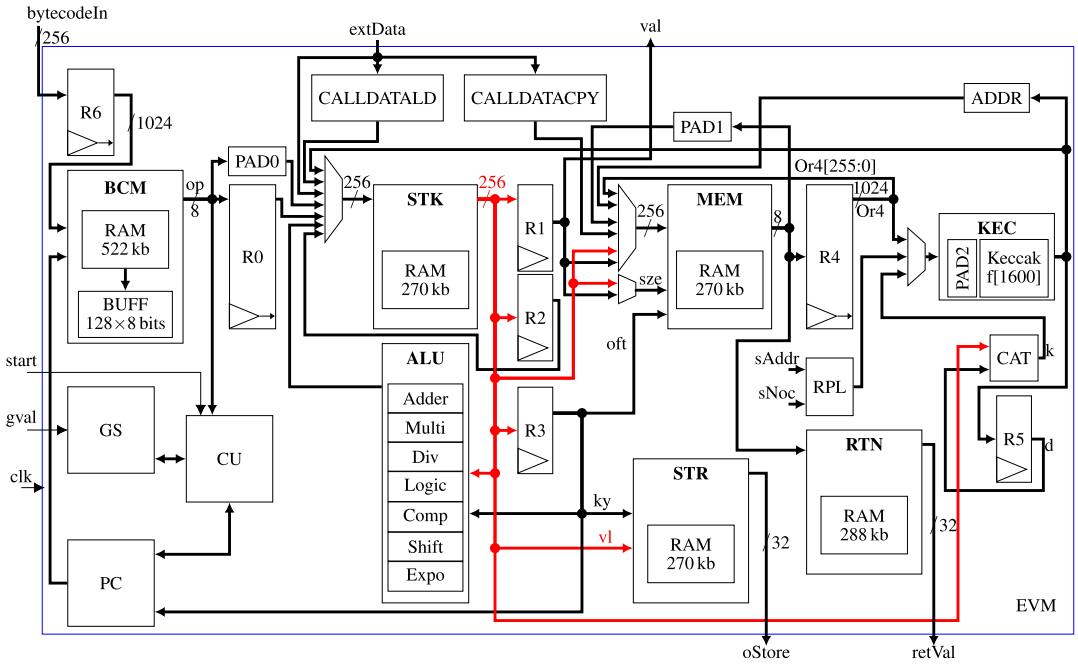

Fig. 2. Block diagram of the proposed architecture of EVMx. Here, `sAddr` denotes the sender's address, `sNonce` is the sender's nonce, and `data` represents the initializing data. The red path highlights the data flow between R1, R2, R3, ALU, MEM, STR, and CAT components.

deployment. This code corresponds to the portion of the SC bytecode generated during compilation. The `oStore` signal outputs the final STR state, while `retVal` retrieves the return (RTN) data stored in the RTN local MEM once a SC has completed execution. Furthermore, the `val` signal specifies the amount of value transferred when a new contract is created.

The proposed architecture operates with a single synchronous clock (`clk`). The `start` signal initiates the SC execution process, while `gval` loads the total available GS. Also, the `bytecodeIn` input loads the bytecode of a compiled SC to the BCM component.

Section IV discusses the core components of EVMx and their respective roles within the architecture.

IV. Core Components of EVMx

This section discusses the design of the core components of the EVMx architecture in detail. Notably, the BCM, the control unit (CU), GS module, STK, MEM, STR, PC, and the ALU. In addition, it also discusses the Keccak-256 (KEC) module as well as the CALLDATALD and CALLDATACPY components.

A. Bytecode Memory, Control Unit, and Gas

The BCM module stores the bytecode of an SC. It consists of 522 kb of dual-port synchronous RAM and a 128-byte (1024-bit) register array shown as register buffer (BUFF) in Fig. 3. Moreover, a CU governs the read and write operations within the BCM component. The RAM is designed to load one 128-byte word per clock cycle (CC), minimizing latency when fetching bytecode from the code field and enabling efficient data transfer to BUFF. BUFF is loaded with the one word from RAM and outputs one byte per CC.

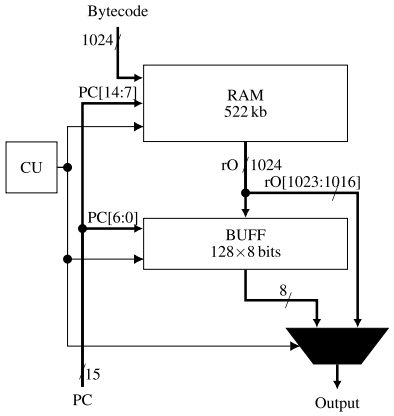

Fig. 3. Proposed architecture of BCM. The multiplexer selects the last bit of the output of RAM (rO) when BUFF is being loaded to ensure continuous SC execution.

The PC sequentially traverses both the RAM and BUFF. Specifically, the 8 most significant bits (MSBs) of the 15-bit PC address the words in RAM, while the 7 least significant bits (LSBs) index individual bytes within BUFF. Once the PC reaches the last byte in BUFF, the next 128-byte word is immediately loaded from RAM in a single CC. To avoid execution stalls during this transition, a multiplexer is used to read the 8 MSBs directly from the RAM output while the PC increments from address 0 to 1, ensuring a seamless and continuous SC execution process.

When an opcode is read from BUFF, it is sent to the CU. The CU goes through various execution stages, managing data flow by controlling read and write operations across components based on the executing opcode.

The GS component consists of a counter loaded with the GS limit and a lookup table (LUT) containing the GS values

of the opcodes. It also contains two multipliers used to compute (2). Since the values involved in this equation are relatively small, there is no need for resource-intensive multiplication algorithms. In addition, the division by 512 and the floor function are efficiently implemented as a right shift by 9 bits.

The GS module ensures that the GS cost used to execute an opcode is deducted from the total GS. Moreover, when the GS is exhausted, it sends an interrupt signal to the CU, which then halts the execution and throws an $out - of - GS$ exception.

B. Stack

The STK is made of 270 kb of RAM and processes 32-byte data during push and pop operations. To accommodate this, PAD0 pads the 1-byte data from BCM. However, certain EVM operations, such as `PUSH32`, require reading multiple bytes from BCM before pushing them to the STK. Therefore, R0 is a left-shift register that collects the required amount of data before a push to STK. This design enables a seamless interaction of BCM and STK to execute opcodes with varying operand sizes.

C. Memory

MEM is implemented using 288 kb of RAM and supports byte-addressable read and write operations. Many opcodes interact with MEM using both an offset and a size parameter as addresses to MEM. For instance, the `RETURN` opcode halts execution and returns output data based on these two parameters: the offset indicates the starting byte address in MEM from which data should be copied, while the size defines the number of bytes to copy. To facilitate such operations, MEM uses two specific parameters as shown in Fig. 2, where `oft` is the offset and `sze` is the size.

Some opcodes require accessing a contiguous chunk of data from MEM at once. For example, `KECCAK256` computes a hash over a segment of MEM starting from a given offset and spanning a specified size. Since the hash computation needs all the data to be available simultaneously, a shift register (R4) is used to accumulate this data before it is passed to the KEC module. The KEC module computes the KEC hash digest of a given input.

Conversely, other opcodes operate on data one byte at a time, enabling pipelined execution when utilizing the byte addressable MEM. For instance, the `MCOPY` operation copies a specified number of bytes from one MEM location to another. Suppose the source and destination MEM areas do not overlap. In that case, pipelining is applied: as each byte is read from MEM, it is immediately written to the target location, allowing efficient byte-by-byte transfer.

During the execution of an SC, the MEM module tracks and outputs the highest word index accessed. This value is subsequently used to calculate the GS cost associated with MEM usage, as defined by (2).

D. Storage

STR is a 270-kb RAM block with a depth and width of 1024 and 256 bits, respectively. It functions as a key-value store, where each key is a 6-byte address provided through the input `ky`, and each value is a 32-byte data word provided through the input `vl`, as illustrated in Fig. 2. The STR size was determined by analyzing the requirements of randomly selected SCs. However, since different SCs exhibit varying STR demands, the allocated size of STR can be adjusted to match the specific needs of the user or application.

As stated in Section II-B, STR is a persistent space that stores data on the blockchain. Therefore, the local STR within the proposed architecture temporarily stores data and facilitates STR operations while the EVM executes an SC. After completion, the final STR value is written to the blockchain through the `oStore` output.

E. Program Counter

In addition to sequentially incrementing from zero, the PC can also receive input from the STK to support opcodes such as `JUMP` and `JUMPI`, which require branching to specific bytecode locations. To enable this functionality, the PC is implemented using a set of registers and an adder.

The PC outputs a 15-bit address that spans the entire BCM. Specifically, the upper 8 LSBs address words within the BCM's RAM (from address 0 to 255), while the lower 7 LSBs index individual bytes within BUFF (from address 0 to 127). When a jump targets an address outside the currently-loaded BUFF segment, the PC calculates the corresponding word address in RAM, loads that word into BUFF, and resumes execution from the new location, as detailed in Section IV-A.

In addition, each time a SC bytecode is loaded into the BCM, a corresponding limit is set. This limit defines the valid range of the bytecode and ensures that the PC does not access MEM beyond the end of the contract, thus maintaining safe and bounded execution.

F. Arithmetic and Logic Unit

The ALU component performs various arithmetic and logical operations, including addition, multiplication, division, shifting, and modulo operations. Division, multiplication, and exponential operations are computationally expensive. Therefore, this subsection discusses how we efficiently implement these operations to optimize performance.

To perform division and modulo operations, we use a nonrestoring division algorithm [32]. Moreover, we employ a Booth multiplier algorithm [33] to perform multiplication and a binary exponential algorithm [34] to perform exponentiation. These algorithms are implemented such that they avoid dedicated resources like digital signal processors (DSPs) blocks, which would otherwise increase the FPGA resource usage of EVMx. Also, the algorithms are optimized to handle edge cases, enabling faster execution of opcodes.

1) Division: To perform division, we implement the nonrestoring division algorithm, which leverages an adder and shift registers. Unlike the restoring algorithm, it avoids the step of reinstating the previous value when a subtraction yields a negative result, leading to better efficiency by avoiding extra addition operations during restoration [32]. Fig. 4(a) illustrates the proposed architecture, while Fig. 4(b) presents the corresponding flow diagram. In the figure, $x \Leftarrow y$ indicates

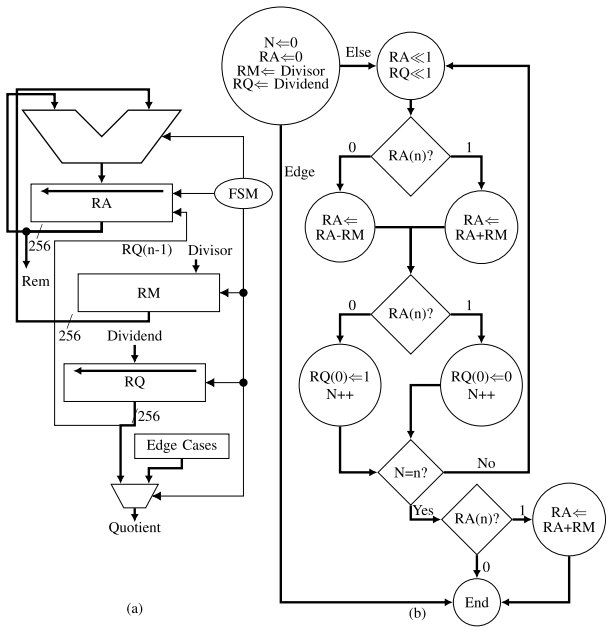

Fig. 4. Architecture (a) and flow diagram (b) of the nonrestoring division algorithm. Rem is the remainder after division.

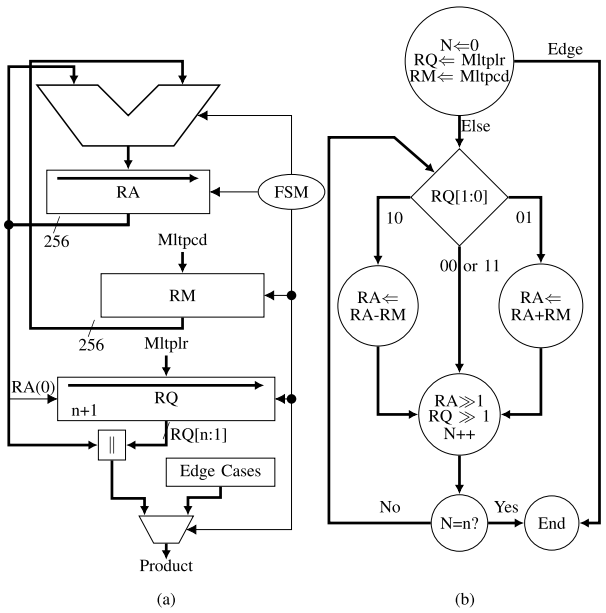

Fig. 5. Architecture of booth multiplication algorithm (a) and its flow diagram (b), where Mltplr is the multiplier and Mltpcd is the multiplicand.

loading register x with value y, while $x \ll y$ means left shifting register x by y bits.

The design uses two left-shift registers: RA and RQ. Register RQ holds the dividend and shifts left on each iteration, while register RA stores the intermediate results of addition or subtraction and simultaneously shifts in the MSB of RQ. As depicted in Fig. 4(b), the addition or subtraction operation is selected based on the MSB of RA. Furthermore, subtraction is implemented using 2's complement arithmetic, allowing the use of a single adder, further optimizing resource utilization.

We also handle critical edge cases, including scenarios where the divisor is 0, 1, greater than the dividend, a power of two, or when the dividend is 0. Each of these cases is resolved within a single CC. Our analysis shows that the majority of division operations performed by the EVM fall into these edge-case categories. For example, many SCs perform division by 2^{224}. In this case, we detect division by a power of two and perform a right-shift operation instead.

2) Multiplication: We use the Booth algorithm to perform multiplication (i.e., the `MUL` opcode) [33]. This algorithm relies on shift registers and a single adder to perform optimized multiplication by reducing the number of addition and subtraction operations, especially when processing consecutive 1s or 0s. The proposed Booth multiplier architecture is shown in Fig. 5(a), and its operational flow is shown in Fig. 5(b). In the figure, $x \gg y$ means right-shifting register x by y bits.

Our architecture employs two right-shift registers: RA and RQ. The register RQ holds the multiplier and shifts in the LSB of RA, while RA is used to store and shift the intermediate results of addition or subtraction. Subtraction is performed using two's complement arithmetic, allowing the design to utilize a single adder for both addition and subtraction operations.

In addition, the architecture accounts for edge cases, such as when the multiplicand or multiplier is 0, 1, or a power of two.

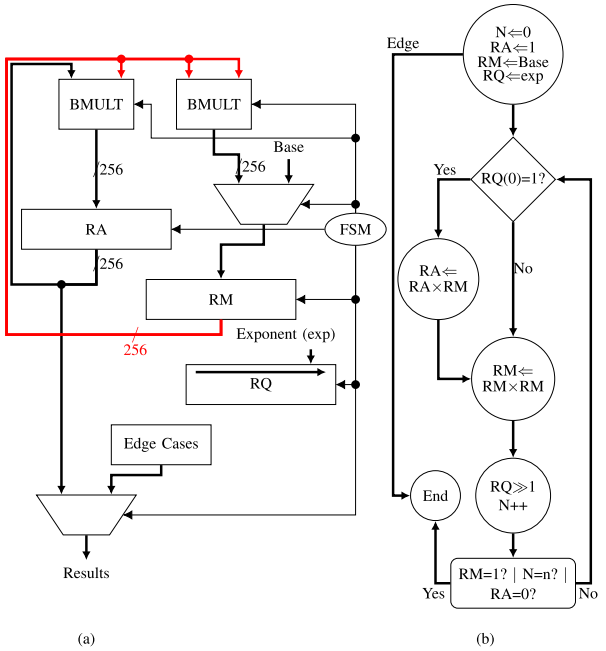

Fig. 6. Architecture of the binary exponentiation algorithm (a) and its corresponding flow diagram (b). The red path highlights the data transfer from the RM module to the BMULT modules.

These specific cases are handled in just one CC, significantly reducing the number of cycles required to complete the multiplication.

3) Exponential: We implement the binary exponentiation method to execute the exponential operation (`EXP` opcode) [34]. This algorithm performs a sequence of squaring and multiplication operations based on the binary representation of the exponent. The proposed hardware architecture is shown in Fig. 6(a), and the corresponding operational flow is illustrated

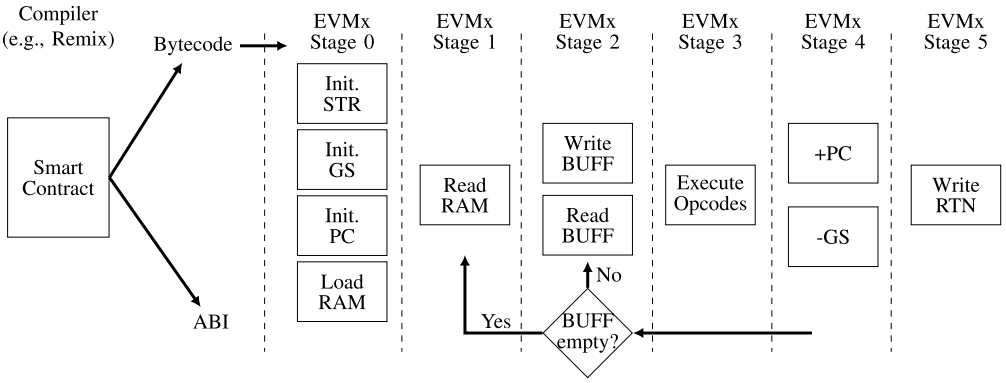

Fig. 7. Pipeline stages of the proposed EVMx architecture. EVMx executes the bytecode, generated from a compiled SC, through Stages 0–5. A feedback loop connects Stages 1, 2, and 4, where the buffer is repeatedly read until its last element is reached and the next word is fetched from RAM.

in Fig. 6(b). In the figure, | symbolizes the bit-wise OR operation.

We utilize two Booth multipliers (`BMULT`), described above, running in parallel to perform multiplication and squaring operations, as guided by the flow diagram. This helps reduce computation time when RQ(0) = 1. Moreover, a shift register, RQ, holds the exponent and shifts it to the right, with each LSB determining whether the current operation is a multiplication or a squaring.

In the worst-case scenario, n iterations are needed to complete the exponentiation, where n is the number of bits of the operands. However, we enhance efficiency by terminating the algorithm early when registers RM or RA reach 1 and 0, respectively. In addition, we handle the edge case of computing powers of two, i.e., 2^x, where x is an integer, using a left-shift operation, further reducing computational overhead.

G. Other Components

The KEC component computes the KEC hash digest, while the address (ADR) component extracts the 20-byte Ethereum address from the digest. In addition, the CAT component calculates the output `k` as

$$\texttt{k} = \texttt{0xFF} \| \texttt{sAddr} \| \texttt{salt} \| \texttt{d} \tag{3}$$

where ‖ is the concatenation symbol, `0xFF` is the required byte prefix for the `CREATE2` opcode, `sAddr` is the address of the account deploying the new contract, `salt` is a user-defined input, and `d` is the KEC digest. Also, recursive length prefix (RPL) computes the RPL encoding for executing the `CREATE` opcode, where `sNoc` is the sender's nonce.

The `CALLDATALD` component is used to execute the `CALLDATALOAD` opcode. It extracts 32 bytes from the transaction's input data, starting at a specified byte offset within the current execution environment.

Similarly, the `CALLDATACPY` component is used to implement the `CALLDATACOPY` opcode. It reads a specified number of bytes from the transaction input data, beginning at a given offset, and writes them to MEM. Moreover, it transfers one byte per CC as MEM is byte addressable.

V. SC Execution on EVMx

This section discusses the execution process of an SC bytecode within the EVMx architecture. Moreover, it presents example executions of the `CREATE2`, `JUMPI`, and `KECCAK256` opcodes to demonstrate the system's functionality.

A. SC Execution Process on EVMx

The proposed architecture follows the EVM execution loop illustrated in Fig. 1. In addition, Fig. 7 depicts the different stages of SC code execution on EVMx (stages 0 to 5). Unlike [12] and [13], EVMx bypasses the decoding and reordering stage entirely (which would be between the compiler stage and stage 0 in Fig. 7), opting instead to process operations in their original sequence. Before stage 0, the compiler stage executes on a host computer, where the SC is compiled into application binary interface (ABI) and bytecode. The resulting bytecode is then offloaded to EVMx for execution.

The bytecode execution process on EVMx begins by deploying the initialization EVM code, which sets up contract-specific parameters such as STR variables, PC, and all other block and environment variables [35]. This code is part of the bytecode and is deployed through the `initData` input shown in Fig. 2. In parallel, the PC is initialized to zero and the SC's bytecode is loaded into BCM using the `bytecodeIn` input. This bytecode originates from the deployed SC code stored in the account's code field. To make reasonable use of the available pins on the FPGA, we constrain the interface to 256 bits. Data are therefore loaded in 256-bit chunks, with R6 accumulating four chunks (1024 bits total) before they are simultaneously transferred to the BCM. As discussed in Section IV-A, the BCM is a 1024 × 256-bit MEM that can store up to a maximum size of 522 kb of SC bytecode.

At the same time, the GS module is also initialized with the provided GS limit. Once these components are set, the `start` input triggers the CU, which drives the EVMx architecture through the execution loop.

The PC generates addresses for both the RAM and the 128 × 8 bits buffer within the BCM. It begins by reading the first 1024 bits from the RAM and writing them into the buffer. The PC then sequentially traverses this buffer, which outputs one byte at a time through `op`, as illustrated in Fig. 2. Once the entire buffer has been traversed, the PC loads the next 1024 bits from the RAM. Each byte output through `op` may represent either an opcode or a value. Moreover, the CU

interprets op to generate the appropriate control signals to execute the corresponding operation within EVMx.

For example, EVMx may execute an SC whose bytecode begins as shown in Fig. 8. As the PC increments, the op value becomes 0×61, which corresponds to the PUSH2 operation. This operation pushes the next two bytes ($0 \times$ 00EE) onto the STK. As illustrated by this example, certain opcodes are followed by immediate values. In this case, the PC advances to read the next two bytes, which are accumulated in RO. These two bytes are then padded to form a 32-byte word before being pushed onto the STK.

EVMx continues to execute instructions by incrementing PC and processing each opcode, while deducting the corresponding GS fees. If an opcode is invalid, its execution fails or the GS is exhausted, the execution is halted immediately. Otherwise, if the transaction completes successfully, the execution loop exits via a terminating opcode such as STOP or RETURN, then STR is updated and any RTN value uploaded to RTN.

B. Executing CREATE2 on EVMx

The CREATE2 opcode generates a new Ethereum account and associates it with a specific code at a predictable address using (3) [28]. The newly created address is then pushed onto the STK. During execution, the CREATE2 opcode uses four values by popping them from STK: value, offset, size, and salt. In particular, value specifies the amount of cryptocurrency (in wei) to send to the new account. The offset variable indicates the byte offset in MEM where the initialization code for the new account is stored, while size defines the length (in bytes) of the code to be copied. Finally, salt is a 32-byte user-defined value that ensures the creation of a unique EVM-compatible blockchain address.

In Fig. 2, when the CU receives the CREATE2 opcode, i.e., 0xF5, it first pops value from the STK. Next, it pops offset and stores the popped value in R1. It then pops size and stores the popped offset in R3 in parallel. The system then reads data from MEM based on offset and size, accumulating it in the left-shift register R4. Once the data is fully copied, its digest is computed using KEC and stored in R5, which has output d.

When the KEC digest is loaded into R5, the salt is simultaneously popped from the STK. Then, CAT computes k as shown in (3), and the result is passed through the KEC component once more. Finally, the first 20 bytes of the resulting KEC digest are pushed onto STK as the address of the newly created account.

C. Executing JUMPI on EVMx

The JUMPI opcode executes a conditional PC alteration, hence breaking the sequential execution of the deployed bytecode [28]. It is used to implement loops and conditional statements. This process involves popping two values to the STK: counter and b. The PC will only jump to a new destination if b is not zero; otherwise, it is incremented by one. Moreover, the value counter is the byte offset where the PC should jump to. Also, the destination of the jump must be a specific opcode called JUMPDEST ($0 \times$ 5B). For example,

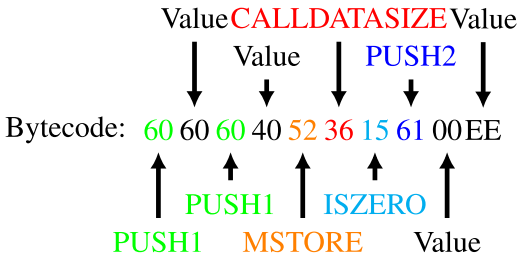

Fig. 8. Example of an EVM bytecode fragment, illustrating data handling and STR operations.

if counter = 1C and b = 1, then PC becomes 1C after executing JUMPI. This new PC address has to point to the JUMPDEST opcode; otherwise, the jump is invalid, resulting in an error being thrown and the execution reverted.

Executing the JUMPI opcode within the EVMx architecture is more intricate due to the structure of BCM, which comprises a RAM and a 128-byte buffer. The complete SC bytecode is stored in RAM, while only a 128-byte segment is buffered in BUFF for immediate access. The PC traverses this buffer, and once it reaches index 126, the next bytecode segment is loaded from RAM.

During the execution of JUMPI, counter is first popped from STK. Then b is popped in parallel with loading counter into register R3. The CU then evaluates b to determine whether a jump should occur. If b is nonzero, the CU verifies two conditions: whether the destination address is within BUFF and whether it points to the JUMPDEST opcode. If the destination lies outside BUFF, the corresponding bytecode segment is fetched from RAM before continuing with the execution process.

D. Executing KECCAK256 on EVMx

The KECCAK256 opcode computes the KEC hash digest of a given input. The hash function generally takes an input of any given size and outputs a 256-bit number. Subsequently, in the EVM, the opcode is used to compute the digest of a segment of MEM. It takes two inputs: offset and size. The offset input is the byte offset in MEM where the data will start to be copied from, while size is the size in bytes of the data to copy. The data to be hashed must all be available at the same time at the input of the KEC hash function. Moreover, the function applies the pad10*1 rule and pads the input message to a multiple of 1088 bits to match the size of the internal state of its sponge function as [36]

$$\texttt{Padding} = (\texttt{message}) \,\|0 \times 01\,\|(00\ldots00)\|\,0 \times 80. \quad (4)$$

In EVMx (see Fig. 2), the offset value is first popped from the STK and loaded into register R3, while size is popped in parallel. EVMx then uses the shift register R4 to sequentially accumulate up to 1024 bits of message data from MEM, starting from the given offset. Based on the size value, the padding module within the KEC unit applies the pad10*1 rule as shown in (4), to extend the message to the nearest multiple of 1088 bits. Once padding is complete, KEC performs the KEC hash computation, and the resulting 256-bit digest is pushed back onto STK.

TABLE II
RESOURCES UTILIZED BY CORE COMPONENTS IN EVMX ON A ZYNQ ULTRASCALE FPGA

Component	LUTs	Registers	RAM (kbits)
Stack (STK)	10 991	10	270
Memory (MEM)	2 230	548	288
Storage (STR)	0	0	270
Bytecode memory (BCM)	1 097	22	522
Keccak (KEC)	2 523	1 620	0
Return memory (RTN)	0	0	288
Arithmetic Logic unit (ALU)	5 564	5 295	0

TABLE III
RESOURCE UTILIZATION COMPARISON ON AMD ULTRASCALE FPGAS

Metrics	BPU [12]	SCU [13]	**EVMx**
Platform	ZC706	XCU250	**ZCU104**
Area			
kLUTs	82.25	142.82	**29.04(13%)**
Registers	67 646	98 850	**13 489(3%)**
DSP	402	225	**0**
RAM (kbits)	450	3 090	**1 638(13%)**
Frequency (MHz)	100	300	**142**

VI. IMPLEMENTATION RESULTS AND DISCUSSIONS

This section presents the implementation results of the proposed EVMx architecture. It begins by describing the implementation process, synthesis setup, and functional verification. Next, it reports the resources required by some core components of EVMx, followed by the resource requirements of the complete design. EVMx is then compared against works from the literature. Subsequently, the section then analyzes some of the most popular opcodes within verified SCs and their execution time on EVMx is compared against that in works from the literature. Also, it evaluates EVMx's performance on complete Ethereum blocks of varying sizes and compares execution times against similar works. Finally, this section provides insights into integrating EVMx with an Ethereum client and discusses its potential operational limitations as well as suggestions for improvements.

A. Methodology

EVMx was implemented in VHDL, and we made the complete source code publicly available in a GitHub repository to promote transparency and reproducibility [15]. Logic synthesis, technology mapping, and place and route were performed using Vivado 2024.2, targeting a Xilinx ZCU104 FPGA board. The board features a Zynq UltraScale+ ZU7EV MPSoC. The ZU7EV includes a programmable logic (PL) section with 230 400 LUTs, 28 800 configurable logic blocks (CLBs), 460 800 registers, and 44.2 Mb of RAM. In addition, it features a processing system (PS) with a quad-core ARM Cortex-A53 processor. However, EVMx's implementation entirely resides within the PL. We use worst-case timing estimates to derive the maximum achievable clock frequency. The functionality of EVMx is verified by comparing its states with opcode executions on Playground [28] and Remix [37]. Both tools allow step-by-step execution of SCs. This makes it easy to track the behavior of each opcode during execution. Moreover, the intermediate and final states of key components, including the STK, MEM, and RTN data, were closely monitored and used as a reference for validating EVMx.

B. Resources Utilized by EVMx on an FPGA

Table II summarizes the resources utilized by some of the core components in EVMx. Specifically, it shows the LUTs, registers, and RAM blocks required by each component. DSP blocks are not reported in the table, as none of the components use them. The table shows that STK requires the highest number of LUTs, highlighting the use of combinational logic to implement the LIFO mechanism of the STK and using RAM for MEM. Also, KEC uses the most registers without any RAM, while MEM, STR, and RTN heavily rely on RAM.

Table III compares the resource utilization of EVMx with related works. The platform column lists the types of FPGAs used. In the table, BPU [12] employs the ZC706, SCU [13] uses the XCU250, and EVMx uses the ZCU104. The ZC706 belongs to an earlier generation than the XCU250 and ZCU104. However, the LUTs in UltraScale+ devices (XCU250 and ZCU104) are broadly comparable to those in 7-series FPGAs (ZC706): they have six inputs. Registers are also comparable. While DSP blocks were enhanced between the 7-series and UltraScale+ families, moving from DSP48E1 to DSP48E2 with wider pre-adders, improved cascading, and optional single instruction, multiple data (SIMD) modes, EVMx does not utilize any DSP blocks. Finally, whereas 7-series FPGAs rely solely on block RAMs (BRAMs), UltraScale+ devices offer both BRAMs and UltraRAMs (URAMs), the latter being large on-chip MEM blocks optimized for high-capacity STR. BPU and SCU use the BRAM18, while EVMx utilizes BRAM36. The 18 and 36 indicate the size in Kilobits of each BRAM block.

According to Table III, EVMx requires the lowest amount of resources, highlighting its suitability for resource-constrained applications. Specifically, EVMx requires 65% and 80% fewer LUTs than BPU and SCU, respectively, and 80% and 86% fewer registers. Also, contrary to both BPU and SCU, it does not require any DSP blocks. Regarding MEM, EVMx uses 45% less RAM than SCU, but 73% more than BPU.

The reduced resource requirements of EVMx stem from its streamlined design, which omits additional scheduler and decoder blocks as well as dedicated configurable units present in BPU and SCU for executing opcodes of frequently-used SCs in parallel. This efficiency, however, comes with a trade-off: EVMx does not analyze contracts for parallel-execution opportunities and executes bytecode exactly as loaded.

Despite this limitation, EVMx requires under 13% of the available resources of the ZU7EV FPGA, leaving ample room for additional logic. This available capacity could, for example, be used to instantiate multiple EVMx cores on the same FPGA, thereby supporting concurrent execution of multiple SCs.

C. Execution Time Analysis of Ethereum codes (opcodes) on EVMx

Fig. 9 shows the most frequently used opcodes in verified Ethereum SCs, based on a sample collected from

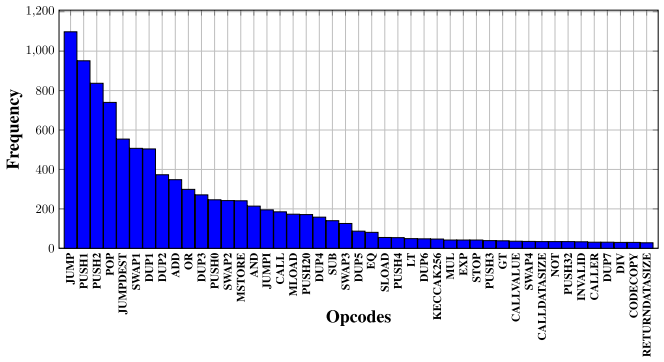

Fig. 9. Frequency distribution of the most commonly used opcodes in 60 verified Ethereum SCs randomly selected from Etherscan.

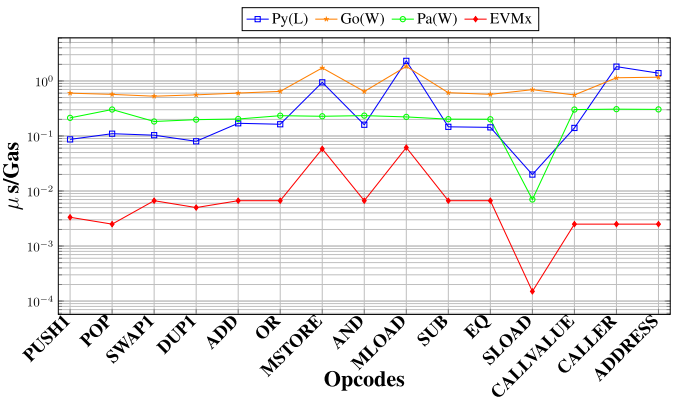

Fig. 10. Execution time of selected opcodes per unit GS on EVMx and CPU-based EVMs running within Ethereum clients.

TABLE IV

PROCESSING TIME OF SELECTED EVM OPCODES ON CPUS (WINDOWS AND LINUX, INTEL I7 3.50 GHZ) [39] AND ON THE PROPOSED EVMX IMPLEMENTED ON A ZCU104 FPGA

Category	Opcode	Name	Gas	LPy (ns)	WGo (ns)	WPa (ns)	**EVMx** (ns)	Δ^a %
Arithmetic	x01	ADD	3	510	602	610	**28**	95
	x03	SUB	3	440	611	606	**28**	94
Logic	x14	EQ	3	430	571	604	**28**	93
	x16	AND	3	480	643	703	**28**	94
	x17	OR	3	490	646	701	**28**	94
Environmental	x30	ADDRESS	2	2770	1170	608	**7**	99
	x33	CALLER	2	3640	1142	614	**7**	99
	x34	CALLVALUE	2	80	556	604	**7**	91
Memory/Stack	x50	POP	2	220	570	605	**7**	97
	x51	MLOAD	3	6950	1838	666	**259**	61
	x52	MSTORE	3	2830	1726	684	**245**	64
	x54	SLOAD	100	1990	694	701	**21**	97
	x60	PUSH1	3	260	600	640	**14**	95
	x90	SWAP1	3	310	528	550	**28**	91
	x80	DUP1	3	240	559	594	**21**	91

[a] $\Delta = \frac{\text{Min(LPy, WGo, WPa)} - \text{EVMx}}{\text{Min(LPy, WGo, WPa)}} \times 100$, where Min is the minimum function.

Etherscan [38]. The dataset was obtained by randomly selecting 20 Ethereum blocks and extracting 2–3 verified SCs from each. These findings are consistent with those reported in [29], which analyzed approximately 40 k verified SCs; notably, the top ten opcodes in [29] also appear among the top 15 in Fig. 9.

The figure shows that certain opcodes, such as `JUMP`, `PUSH1`, and `PUSH2`, appear frequently, while others are rarely used. We analyse the execution time of some of the most commonly used opcodes. Furthermore, we compare their execution times on EVMx against those of software-based EVMs from the literature.

Table IV presents a comparison of execution time for the selected opcodes in various software EVMs running on CPUs against EVMx. The comparison focuses on software CPU-based EVMs. This is because the hardware-based implementations in the literature do not make their designs publicly available nor report execution time for individual opcodes [12], [13]. To evaluate CPU performance, we refer to the benchmarks in [39], where three different Ethereum execution clients, PyEthApp, Geth, and Parity, were deployed on CPU platforms. These clients ensure compliance with Ethereum protocol rules and run the EVM when processing SCs. The evaluations were performed on Windows and Linux systems using an Intel i7 3.50-GHz processor.

TABLE V

COMPARISON OF EXECUTION TIME OF ETHEREUM BLOCKS AGAINST WORKS IN THE LITERATURE

Block	# of Txs	CPU (μs) [13]	BPU (μs) [12]	SCU (μs) [13]	**EVMx** (μs)	Δ^a %
		Intel i7-7700k	ZC706	XCU250	ZCU104	
6653186	190	66 040	1 708.40	318.51	**292.65**	8
6653197	102	56 669	1 410.90	301.68	**230.57**	24
6653232	115	57 129	1 129.90	235.76	**101.89**	56
6653205	16	3 696	218.40	64.65	**53.38**	17
6653208	78	37 037	1 410.90	243.73	**228.07**	6
6653209	159	46 092	1 595.80	372.01	**312.65**	16
6653220	9	4 184	165.70	52.21	**45.50**	13

[a] $\Delta = \frac{\text{SCU} - \text{EVMx}}{\text{SCU}} \times 100$.

Table IV also shows the best-performing software EVM from each client in [39] and compares them against EVMx. Specifically, EVMx is evaluated against the PyEthApp client on Linux (LPy), the Geth client on Windows (WGo), and the Parity client on Windows (WPa). The Δ column shows that across all the opcodes, EVMx outperforms its software counterparts. The execution-time reduction ranges from 61% to 99%, where most reductions are above 90%. EVMx can execute SCs much faster than CPU-based EVMs.

Similar results are observed in Fig. 10, which analyzes the execution time to GS ratio. The EVMx curve consistently lies below all other curves across all opcodes, aligning with the Δ column in Table IV. This demonstrates that EVMx achieves the lowest execution time per unit of GS, indicating that executing opcodes on it is computationally less expensive than executing on CPU-based EVMs. As a result, users consume fewer resources to perform the same tasks while benefiting from higher TPS.

D. Execution Time Analysis of Ethereum Blocks on EVMx

Table V compares the execution time of Ethereum blocks against other hardware implementations from the literature. Specifically, we analyze the processing of sample blocks `6653186`, `6653197`, `6653232`, `6653205`, `6653208`, `6653209`, and `6653220`. These blocks have different sizes, indicated by the number of transactions in each block. Moreover, the SCs

TABLE VI
COMPARISON OF BLOCK TIME AND BYTECODE LOADING TIME ON EVMX

Block	Block Time (μs)	Bytecode Loading Time (CC)	Bytecode Loading Time (μs)	Δ[a] %
6653186	292.65	9 593	67.15	23
6653197	230.57	9 011	63.08	27
6653232	101.89	3 800	26.60	26
6653205	53.38	2 189	15.32	29
6653208	228.07	7 368	51.57	22
6653209	312.65	10 703	74.92	23
6653220	45.50	1 521	10.65	23

[a] $\Delta = \frac{\text{Bytecode Loading Time } (\mu s)}{\text{Block time}} \times 100$.

within these blocks contain diverse operations, including MEM-heavy operations and extensive STR interactions, offering a good benchmark to evaluate the generality and scalability of EVMx. For example, the contract at address `0xA62142888ABa8370742bE823c1782D17A0389Da1` inside block `6653220` is a dApp running a game called Fomo3D [40]. The game combines elements of gambling, game theory, and social engineering, offering players the opportunity to earn cryptocurrency. The Fomo3D contract features a substantial codebase that is both MEM-heavy and computationally demanding.

The above blocks were also executed on an Intel i7-7700K quad-core CPU running at 4.2 GHz, as well as on FPGA-based EVMs. Specifically, BPU [12] was implemented on a ZC706 FPGA, while SCU [13] was deployed on an XCU250 FPGA. In [13], several single- and multicore SCU implementations were evaluated. For comparison with the proposed EVMx, we select the best-performing SCU design.

The Δ column in Table V shows the percentage difference in execution time between SCU and EVMx. It can be observed that EVMx outperforms SCU when executing all the blocks, with the difference ranging from 6% to 56%. SCU analyzes the bytecode to detect dependencies and reorders instructions accordingly before execution. In addition, it uses a reordering buffer to write the results of out-of-order executions to STR elements correctly. These mechanisms may contribute to the time differences observed in the Δ column.

When analyzing individual blocks, we observe that block `6653232` has the highest execution time reduction while `6653208` has the lowest. The table shows that EVMx executes block `6653232` 561× faster than the software implementation running on a CPU, 11× faster than BPU, and 2× faster than SCU. Moreover, EVMx executes block `6653208` 162× faster than the implementation on a CPU, 6× faster than BPU, and 1.1× faster than SCU. In addition, the average block execution time improvements of EVMx compared to CPU implementation, BPU, and SCU are 99%, 81%, and 20%, respectively. These results demonstrate EVMx's potential to improve the performance of the EVM in executing SCs.

In Table VI, we provide a more detailed analysis of the block execution times previously presented in Table V. Given that efficient loading of SCs bytecode is a known challenge in EVMx, we examine the total time spent loading the bytecode of all SCs (corresponding to EVMx Stage 0 in Fig. 7) within each block, using the method described in Section V-A. The bytecode loading time is reported in both CCs and microseconds (μs). It represents the cumulative loading time per block. The Δ column indicates the proportion of this loading time relative to the total block execution time.

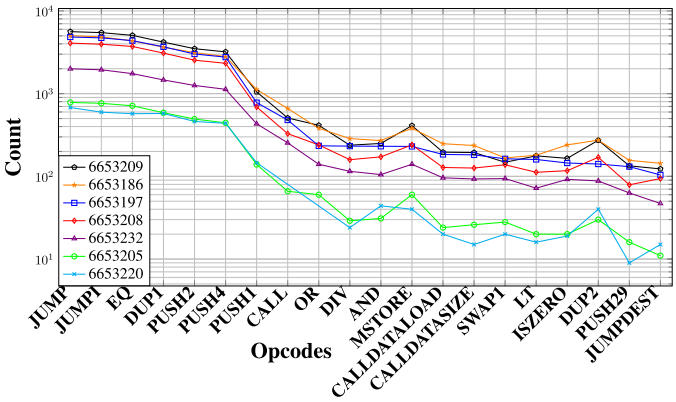

Fig. 11. Top 20 opcodes executed by SCs within Ethereum blocks. Blocks with high opcodes count have more SC.

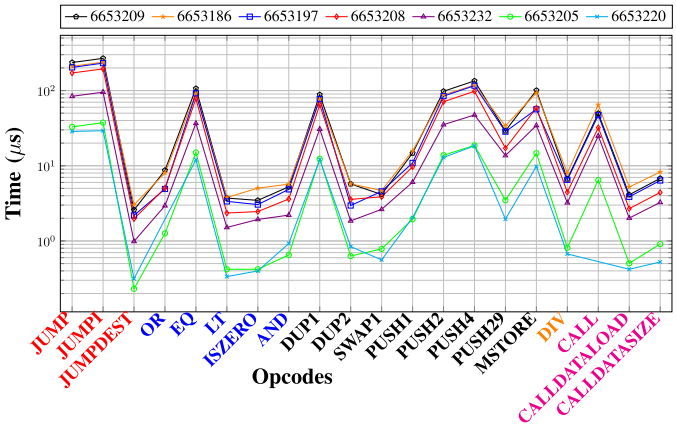

Fig. 12. Top 20 opcodes executed by SCs within Ethereum blocks. Opcodes are grouped and color-coded: red (flow control), blue (logic), black (MEM/STK), orange (arithmetic), and magenta (environmental calls).

Across all blocks, bytecode loading accounts for approximately 22%–29% of total execution time, meaning that EVMx dedicates the remaining 70%–80% to executing the actual SC logic (corresponding to stages 1 to 5 in Fig. 7). This highlights EVMx's ability to accelerate SC execution and provides an opportunity to further improve performance by optimizing the bytecode loading phase.

Furthermore, we analyze the opcodes within the sampled blocks to understand their execution characteristics and performance implications. For clarity, we focused on the most frequently utilized opcodes in each block and selected the top 20 opcodes common across all blocks. Fig. 11 presents these opcodes. In general, blocks with a higher number of transactions, as shown in Table V, also contain a larger number of opcodes. This trend is expected, as a greater volume of transactions typically leads to increased contract activity and, consequently, more opcode executions.

Fig. 12 presents the cumulative execution time on EVMx of the top 20 opcodes from the sampled SCs. The opcodes are grouped and color-coded according to the functional

categories, as shown in Table IV: flow control (red), logic (blue), MEM/STK operations (black), arithmetic (orange), and environmental calls (magenta). The results show significant variation in execution time across opcode groups. Flow control instructions such as `JUMP`, `JUMPI`, and `JUMPDEST` exhibit notably higher cumulative execution times than many logic and arithmetic instructions, reflecting their frequent utilization within SCs.

MEM- and STK-related operations (`MSTORE`, `PUSH29`) and environmental calls (`CALL`) also exhibit higher execution times, although they occur less frequently than most opcodes. This suggests that STR access and the processing of large data structures impose greater overhead. In contrast, lightweight operations such as `OR` and `SWAP` demonstrate relatively lower latency, consistent with their simpler functional roles.

It is also noteworthy that while `DIV` has a relatively high utilization rate, as shown in Fig. 11, its execution time in Fig. 12 remains lower than most opcodes, e.g., compared to `PUSH29`. This is largely because SCs were found to often contain edge-case divisions, e.g., division by powers of two. As discussed in Section IV-F.1, EVMx heavily optimizes these cases. Such optimizations likely contribute to the improved execution time compared to BPU and SCU, suggesting EVMx's superior acceleration in arithmetic operations.

E. Integrating EVMx With an Ethereum Client

EVM-based blockchain networks rely on clients to connect with peers and maintain network operations. For instance, Ethereum users may run clients such as Geth [20], Teku [25], Lighthouse [24], or Nethermind [22]. These clients are independent implementations of Ethereum that validate data according to its protocols, ensuring the integrity and security of the network. In Ethereum, each node typically runs two clients: a consensus client, e.g., Teku, responsible for implementing the consensus algorithm, and an execution client, e.g., Geth, which listens to new transactions broadcast within the network and executes them using the EVM. Even though the integration of EVMx with EVM-compatible clients is beyond the scope of this work, this section provides insights into how such an integration with an Ethereum consensus client may be implemented.

EVMx is a hardware accelerator deployed on the PL of the ZCU104 FPGA. While several integration strategies are possible, one viable approach is to host the software Ethereum execution client (e.g., Geth) on a Linux operating system running on the PS side of the ZCU104 FPGA. Through a dedicated kernel module, the Ethereum client in the PS can interact with EVMx in the PL. Specifically, the client transfers the SC bytecode from the PS to the PL over an advanced extensible interface (AXI), where control signals and status registers are accessed via AXI4-Lite. Also, larger data transfers can be handled through AXI4 or AXI4-Stream with direct MEM access (DMA) support. The client within the PS initiates execution by writing control commands, and upon completion, the results are retrieved from the PL using the same interface.

Alternatively, the Ethereum client can run on a CPU external to the FPGA, or even on a separate computer. The bytecode may be transferred to the BCM module via a high-speed interface such as Ethernet, PCIe, or USB. The same interface can also handle initialization and handshaking between EVMx and the client. After execution, the resulting data can be transmitted back through the chosen interface for submission to the blockchain.

F. Power Analysis

We employ the Vivado power analysis tool to estimate the power requirements of EVMx, using the estimator's default parameters. While these estimates may not be fully accurate, since the tool notably assumes worst-case switching activity and operating conditions, they provide a reasonable indication of the maximum expected power consumption.

Post-implementation power estimation indicates that EVMx consumes a total of 1.472 W on the ZCU104 FPGA. Of this, dynamic power accounts for 59% (0.88 W), while static power contributes for 41% (0.60 W). The primary sources of dynamic power are signal activity (45%) and logic operations (34%). The estimated junction temperature was 26.4 °C. Although the current analysis relies on vectorless estimation, future work will incorporate switching activity from post-implementation simulations, along with direct power measurements on the FPGA board, to enhance accuracy.

G. Potential Operational Limitations and Ways Forward

One of the main challenges of EVMx is the limited on-chip MEM available when executing large SCs. The BCM storing the bytecode currently has a maximum capacity of 522 kb. For larger SCs, this capacity may need to be extended. While not a concern on FPGAs such as the ZU7EV, which provides ample RAM, it may constrain scalability on smaller FPGAs with limited on-chip MEM. To overcome this limitation, the full bytecode could instead be stored in external MEM and streamed in segments to on-chip MEM within the FPGA. For example, the ZCU104 board could hold 4 GB of bytecode in its DDR4 MEM [41]. This would provide efficient, temporary access during execution.

In addition, transferring a large volume of data through the `bytecodeIn` input can significantly increase SC execution latency. This latency stems from the intentionally constrained input width, limited to maintain reasonable FPGA pin utilization. Consequently, the bytecode is currently loaded at a rate of 256 bits per CC. This limitation could be mitigated in several ways, notably by widening the interface to reduce transfer cycles or by duplicating the BCM and preloading the next SC during execution.

VII. Conclusion

This work presents EVMx, an FPGA-based EVM accelerator that offloads SC execution to dedicated hardware. The design follows a processor-like architecture inspired by the RISC philosophy, maintaining simplicity while prioritizing resource-usage minimization. To achieve this, the proposed architecture integrates optimized components. Experimental results show a 61% to 99% execution-time reduction for commonly used opcodes compared to CPU-based EVMs. Moreover, EVMx executes entire Ethereum blocks between 147× to 560× faster than CPU-based implementations, and 1.1× to 2× faster than the state-of-the-art FPGA implementations. These results demonstrate the potential of EVMx

to improve the execution time of SCs compared to existing solutions, consequently, improving the performance of EVM-compatible blockchains. Furthermore, the design requires only 13% of the available LUTs on a ZU7EV FPGA while operating at a clock frequency of over 140 MHz, leaving ample resources for additional functionality or allowing full and archival nodes to use multiple EVMx instances on the same FPGA.

REFERENCES

[1] V. Buterin et al., "A next-generation smart contract and decentralized application platform," *White Paper*, vol. 3, no. 37, pp. 1–2, 2014.

[2] B. Sriman and S. G. Kumar, "Decentralized finance (DeFi): The future of finance and defi application for Ethereum blockchain based finance market," in *Proc. Int. Conf. Adv. Comput., Commun. Appl. Informat. (ACCAI)*, Jan. 2022, pp. 1–9.

[3] R. Jia and S. Yin, "To EVM or not to EVM: Blockchain compatibility and network effects," in *Proc. ACM CCS Workshop Decentralized Finance Secur.*, Nov. 2022, pp. 23–29.

[4] Alchemy.(2022). *Archive Nodes—Everything You Need to Know*. Accessed: 2025. [Online]. Available: https://www.alchemy.com/overviews/archive-nodes

[5] Y. Gao, J. Shi, X. Wang, Q. Tan, C. Zhao, and Z. Yin, "Topology measurement and analysis on Ethereum P2P network," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2019, pp. 1–7.

[6] A. Battah, Y. Iraqi, and E. Damiani, "Blockchain-based reputation systems: Implementation challenges and mitigation," *Electronics*, vol. 10, no. 3, p. 289, Jan. 2021.

[7] Ethereum Foundation.(2025). *Ethereum Accounts*. Accessed: Sep. 17, 2025. [Online]. Available: https://Ethereum.org/en/developers/docs/accounts/

[8] M. Jin, R. Liu, and M. Monperrus, "On-chain analysis of smart contract dependency risks on Ethereum," 2025, *arXiv:2503.19548*.

[9] M. Cortes-Goicoechea, T. Mohandas-Daryanani, J. L. Muñoz-Tapia, and L. Bautista-Gomez, "Can we run our Ethereum nodes at home?," *IEEE Access*, vol. 12, pp. 44401–44423, 2024.

[10] Y. Fang, Z. Zhou, S. Dai, J. Yang, H. Zhang, and Y. Lu, "PaVM: A parallel virtual machine for smart contract execution and validation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 1, pp. 186–202, Jan. 2024.

[11] T. Li et al., "SmartVM: A smart contract virtual machine for fast on-chain DNN computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4100–4116, Dec. 2022.

[12] T. Lu and L. Peng, "BPU: A blockchain processing unit for accelerated smart contract execution," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[13] T. Lu and L. Peng, "SCU: A hardware accelerator for smart contract execution," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Dec. 2023, pp. 356–364.

[14] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "EtherSolve: Computing an accurate control-flow graph from Ethereum bytecode," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC)*, May 2021, pp. 127–137.

[15] J. P. Lemayian, G. Gagnon, K. Zhang, and P. Giard. (2025). *EVMx: An FPGA-Based Accelerator for Smart Contract Processing*. Accessed: Oct. 24, 2025. [Online]. Available: https://github.com/JPL24hub/EVMx

[16] J. P. Lemayian, H. Bensalem, G. Gagnon, K. Zhang, and P. Giard, "EVMx: An FPGA-based smart contract processing unit," in *Proc. IEEE 49th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jul. 2025, pp. 1708–1713.

[17] Ethereum Foundation.(2025). *The Merge*. Accessed: Sep. 17, 2025. [Online]. Available: https://Ethereum.org/en/roadmap/merge/

[18] D. Vujičić, D. Jagodić, and S. Randić, "Blockchain technology, bitcoin, and Ethereum: A brief overview," in *Proc. 17th Int. Symp. INFOTEH-JAHORINA (INFOTEH)*, Mar. 2018, pp. 1–6.

[19] D. Grandjean, L. Heimbach, and R. Wattenhofer, "Ethereum proof-of-stake consensus layer: Participation and decentralization," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, Jul. 2024, pp. 253–280.

[20] *Go-Ethereum (Geth): Official Go Implementation of the Ethereum Protocol*. Accessed: Sep. 17, 2025. [Online]. Available: https://geth.Ethereum.org/

[21] Hyperledger Besu.(2025). *Besu Ethereum Client—Hyperledger Besu Documentation*. Accessed: Sep. 17, 2025. [Online]. Available: https://besu.hyperledger.org/

[22] Nethermind.(2025). *Nethermind—Building Blocks for the Decentralized Future*. Accessed: Sep. 17, 2025. [Online]. Available: https://www.nethermind.io/

[23] S. Aggarwal and N. Kumar, "Hyperledger," *Adv. Comput.*, vol. 121, pp. 323–343, Apr. 2021.

[24] Sigma Prime Lighthouse.(2025). *Lighthouse—Ethereum Consensus Client by Sigma Prime*. Accessed: Sep. 17, 2025. [Online]. Available: https://lighthouse.sigmaprime.io/

[25] ConsenSys.(2025). *Teku—Ethereum 2.0 Client for Institutional Staking*. Accessed: Jul. 2025. [Online]. Available: https://consensys.io/teku

[26] Ethereum. (Jan. 30, 2025). *Solidity*. [Online]. Available: https://docs.soliditylang.org/en/latest/

[27] Z. Zheng et al., "An overview on smart contracts: Challenges, advances and platforms," *Future Gener. Comput. Syst.*, vol. 105, pp. 475–491, Apr. 2020.

[28] EVM Codes.(2025). *An Ethereum Virtual Machine Opcodes Interactive Reference*. Dune. Accessed: Sep. 17, 2025. [Online]. Available: https://www.evm.codes/

[29] S. Bistarelli, G. Mazzante, M. Micheletti, L. Mostarda, D. Sestili, and F. Tiezzi, "Ethereum smart contracts: Analysis and statistics of their source code and opcodes," *Internet Things*, vol. 11, Sep. 2020, Art. no. 100198.

[30] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Jan. 2013.

[31] Ethereum Foundation.(2025). *Ethereum Virtual Machine (EVM)*. Accessed: Sep. 17, 2025. [Online]. Available: https://Ethereum.org/en/developers/docs/evm/

[32] U. S. Patankar, M. E. Flores, and A. Koel, "Division algorithms—From past to present chance to improve area time and complexity for digital applications," in *Proc. IEEE Latin Amer. Electron Devices Conf. (LAEDC)*, Feb. 2020, pp. 1–4.

[33] B. V. Dharani, S. M. Joseph, S. Kumar, and D. Nandan, "Booth multiplier: The systematic study," in *Proc. Int. Conf. Commun. Cyber Phys. Eng.*, 2020, pp. 943–956.

[34] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Samir, "Comparative power analysis of modular exponentiation algorithms," *IEEE Trans. Comput.*, vol. 59, no. 6, pp. 795–807, Jun. 2010.

[35] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. Sebastopol, CA, USA: O'Reilly Media, 2018.

[36] M. J. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," Federal Inf. Process. Standards, Gaithersburg, MD, USA, Tech.~Rep. FIPS 202, 2015.

[37] Remix Project.(2025). *Remix—Ethereum IDE. Ethereum Foundation*. Accessed: Sep. 17, 2025. [Online]. Available: https://remix.Ethereum.org/

[38] Etherscan.(2025). *Etherscan—Ethereum Blockchain Explorer*. Accessed: Sep. 17, 2025. [Online]. Available: https://etherscan.io/

[39] A. Aldweesh, M. Alharby, M. Mehrnezhad, and A. van Moorsel, "The OpBench Ethereum opcode benchmark framework: Design, implementation, validation and experiments," *Perform. Eval.*, vol. 146, Mar. 2021, Art. no. 102168.

[40] Team Just.(2025). *Fomo3D—Ethereum-Based Decentralized Lottery Game*. Accessed: Sep. 17, 2025. [Online]. Available: https://fomo3d.net/

[41] Xilinx.(2018). *ZCU104 Evaluation Board User Guide (UG1267)*. Accessed: Sep. 17, 2025. [Online]. Available: https://tinyurl.com/zcu104