2025 IEEE 18th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) | 979-8-3315-6571-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/MCSOC67473.2025.00048

FlexSiMArch: An Extensible Simulator for Research and Development in Secure-by-Design Processor Technologies

Sameer Mankotia* and Daniel Conte de Leon*
*Center for Secure and Dependable Systems, University of Idaho, Moscow, Idaho, USA
Emails: mank8837@vandals.uidaho.edu, dcontedeleon@ieee.org

***Abstract*—Successful attacks on digital systems have been prevalent for decades. Several incremental solutions have been implemented including: Non-executable Stack, Control Flow, Pointer/Buffer Bounds, Trusted Execution, Stack Canaries, and Address Space Layout Randomization. However, preventing low-level attacks on digital systems is still an unsolved major challenge. Four out of the top ten KEV (Known Exploited Vulnerabilities) fall within the category of low-level weakness. We believe a top-to-bottom solution that includes co-designed and/or well-integrated: languages for requirements, design, and code, development processes, hardware, and tool-sets for the development of secure-by-design digital systems is much needed. We also believe that such a solution must be incrementally implemented and adequately integrated with current design and development processes and tools to be successfully adopted. Toward this goal, we introduce FlexSiMArch, an easily extensible Python-based simulator that supports the rapid development and evaluation of new digital processor architectures and instruction sets. Currently, FlexSiMArch supports RISC-V (RV32I, RV64I). We are using FlexSiMArch to simulate and evaluate a novel hardware-based and instruction-level security policy enforcement technology (BHPol).**

***Index Terms*—Hardware security, Simulation, Computer architecture, Security policy, Instruction set architecture.**

I. Introduction

Modern computing systems face an escalating landscape of security threats targeting low-level vulnerabilities. The MITRE Top 10 KEV Weaknesses 2024 report [1] identifies the following low-level weaknesses as currently being exploited, including, Out-of-Bounds Write (CWE-787), Type Confusion (CWE-843), Use After Free (CWE-416), and Deserialization of Untrusted Data (CWE-502). A successful exploit on a control system using these weaknesses may result in major negative impacts on physical processes and people's safety.

To help address this challenge, researchers and engineers need effective tool-sets that enable and facilitate secure-by-design innovation and the corresponding evaluation and assessment of these new technologies. FlexSiMArch was created with the objective of directly addressing the need for appropriate research and innovation tools on the secure-by-design space.

FlexSiMArch, presents a modular and inheritance-based design aimed at enabling a fast iteration cycle of design, implementation, and evaluation for novel processor and Instruction Set Architectures (ISA). The simulator currently supports RISC-V [2] 32-bit and 64-bit integer instruction sets (RV32I, RV64I) and facilitates extension to other architectures and/or instruction sets. FlexSiMArch focuses on ease of extensibility and not on simulation performance. It enables researchers to easily modify or extend its components for rapid innovation and analysis of digital novel and/or hybrid digital processors.

This paper presents the architecture, features, and applications of FlexSiMArch in novel processor and secure-by-design technologies. We detail its modular components including Node, Hart, RegisterSet, Memory, and Instruction Set Database (ISADB) and its integrated testing framework. We demonstrate how FlexSiMArch's modular and inheritance-based design enables the fast creation of new instructions, enabling the evaluation of novel processor and ISA designs.

II. Related Work

Processor simulation has made significant advances to facilitate research in architecture, performance optimization, and security analysis. Simulation tools are essential for evaluating new processor designs, validating architectural innovations, and assessing expected hardware functionality and properties prior to hardware implementation.

A. General-Purpose Processor Simulators

Well-established simulation frameworks like SimpleScalar [3], Gem5 [4], and QEMU [5] provide comprehensive features for performance modeling and architectural analysis. General-purpose simulator capabilities have been further enhanced by frameworks such as PTLsim [6], which provides cycle-accurate x86-64 simulation, and Sniper [7], which offers scalable multi-core simulation. Performance modeling has been advanced through specialized tools like McPAT [8], which provides integrated power, area, and timing analysis. More recently, Multi2Sim [9] has emerged as a unified framework for CPU-GPU computing simulation.

While these simulators excel at simulating current CPU and GPU architectures and their features, they present challenges for enabling the fast experimentation on new ISAs, hybrid and novel architectures, and security-focused research, making fast architectural experimentation and security analysis difficult and time consuming.

2771-3075/25/$31.00 ©2025 IEEE
DOI 10.1109/MCSoC67473.2025.00048

B. Performance-Focused Simulators

A distinct category of simulators prioritizes performance and scalability for large-scale architectural studies. ZSim [10] enables fast and accurate micro-architectural simulation of thousand-core systems, while MARSS [11] provides full system simulation for multi-core x86 CPUs. Dynamic binary translation approaches offer additional performance improvements.

These performance-oriented simulators have enabled research into shared resource management and cache policies for chip multiprocessors. However, their focus is on enabling the analysis of multi-core and/or multi-processor scalability rather than the ease of investigating novel processor ISAs.

C. Architecture-Specific Simulators

Simulators designed for specific architectures, such as Spike [12] and ARM's ecosystem tools, provide detailed representations of their respective architectures. Hardware developers commonly use these tools to validate implementations and provide accurate validation of processor designs. The RISC-V ecosystem has seen significant growth in simulation tools, including the Rocket Chip generator [13], which enables rapid generation of RISC-V implementations. Specialized RISC-V simulators target IoT security education [14], CREATOR [15] offers an integrated educational development environment for RISC-V programming, and McGrew et al. [16] provide frameworks for undergraduate education. RISC-V extensions and customizations, as documented by Waterman et al. [17] and Lee et al. [18], have enabled agile processor development. The security analysis of the RISC-V architecture has been advanced through works such as Feng et al. [19] and Nguyen et al. [20], which examine security enforcement and lightweight cryptography implementations.

D. Security-Focused Simulation Tools

Security-focused simulation tools such as SecVerilog [21] and RIFLE [22] are designed to examine specific security issues. RIFLE supports information flow tracking, while SecVerilog facilitates hardware security verification. The landscape of processor security vulnerabilities has been extensively documented through works examining speculative execution exploits like Spectre [23] and Meltdown [24], as well as transient execution attacks such as Foreshadow [25] and ZombieLoad [26]. Security simulation frameworks have evolved to address these threats, with tools like GRIFFIN [27] leveraging Intel processor trace for control flow protection.

Although these tools provide valuable security insights, they typically focus on particular vulnerabilities rather than offering a comprehensive framework for secure-by-design analysis and evaluation.

E. Gap Analysis and Contribution

Current processor simulation tools present significant barriers to rapid architectural experimentation. Table I compares FlexSiMArch with representative simulators across key dimensions relevant to secure-by-design research.

TABLE I: Quantitative Comparison of Simulator Codebases [3], [4], [5], [12]

Simulator	Language	Lines of Code	Core Files
Gem5	C++	∼500,000	∼2,500
Spike	C++	∼15,000	∼150
QEMU	C	∼1,400,000	∼8,000
SimpleScalar	C	∼30,000	∼50
FlexSiMArch	**Python**	**∼3,500**	**∼25**

1) Practical Extension Barriers in Existing Simulators: *Gem5* requires modifying multiple tightly-coupled C++ components to add instructions. Adding a single instruction requires changes to decoder tables, execution units, and potentially timing models, involving hundreds of lines across 5-10 files. We estimate that adding a new instruction to Gem5's would take several weeks of work.

Spike, while more modular than Gem5, embeds instruction semantics in C++ execution units. Adding custom instructions requires recompiling the simulator and understanding its internal dispatch mechanisms. Instruction set extensions typically require modifying 3-4 core files with 200-500 lines of code. We estimate that adding a new instruction to Spike would take several weeks of work.

QEMU prioritizes execution speed through dynamic binary translation, making architectural modifications complex. Adding instructions requires understanding QEMU's intermediate representation (TCG) and modifying its code generation infrastructure, estimated to involve several hundreds of lines of code changes.

2) How FlexSiMArch Reduces Extension Barriers: FlexSiMArch's declarative instruction definition approach separates specification from implementation. New instructions are added through database entries rather than code modifications, reducing the typical extension from hundreds of lines to a few lines of code per instruction. The inheritance-based component design isolates architectural variants, preventing modifications to one architecture from affecting others. Python's dynamic typing and introspection capabilities enable runtime instruction dispatch without recompilation.

These design choices directly address the practical barriers that make architectural experimentation time-prohibitive in existing simulators. Our vector extension case study (Section V) demonstrates this advantage: adding 45 instructions required 184 lines of code and approximately 4 hours of work, compared to an estimated several weeks for similar extensions in C/C++-based simulators. While FlexSiMArch sacrifices execution speed ($20\times$ slower than Spike), we believe this trade-off is justified for research scenarios where development and evaluation agility outweighs runtime performance.

III. System Architecture

This section describes the design philosophy, components, and interaction mechanisms of FlexSiMArch.

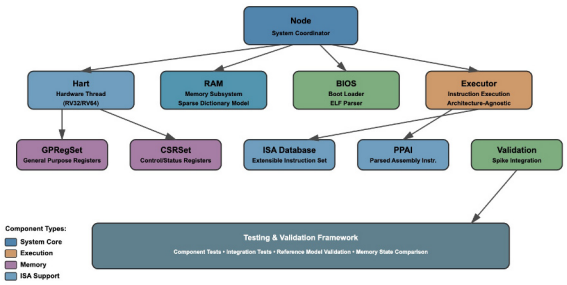


Fig. 1: FlexSiMArch 's high-level architecture depicting core components and their interactions. The inheritance-based design enables architectural flexibility through abstraction layers that separate interface from implementation.

A. Modular and Distributed Architecture

FlexSiMArch employs an inheritance model and modular, component-based design, with clear separation of concerns to facilitate architectural extensibility. The simulator consists of several interconnected components working together to create a complete simulation environment as described below and shown in Fig. 1.

A `Node` acts as the container of processing units and Resource and I/O manager. A `Hart` provides an abstract hardware thread implementation with architecture-specific derived implementations. Within a Hart, one or more `CRSet` (Control Registers) implement desired Control and Status Registers and a `GPRegSet` implements general-purpose registers with configurable width and naming support. A component `RAM` simulates (one or more) Random Access Memory devices. A `BIOS` manages system checks and bootloading including ELF file loading. The ISA Database (`ISADB`) maintains extensible instruction set definitions and properties. The `Executor` uses the data and definitions on the ISADB to implement instruction execution across architectural variants or newly defined instructions. A `Console` module enables human-machine communication with one or more Nodes. A set of integrated testing and validation tools enable evaluation against reference models or specific-purpose tests.

B. Core System Components

1) Node Component: The Node's responsibilities include hardware thread coordination, memory subsystem coordination, peripheral component coordination, boot-loading, and node-wide state maintenance. It was designed to handle node-level operations such as resets, interrupts, and exception propagation within a simulated local computation unit while maintaining the architecture's modularity.

FlexSiMArch was designed to enable a Node to contain as many Harts, RAMs, and I/O, as desired and of varied types. FlexSiMArch was also designed to enable distributed simulation through the deployment of Nodes, RAM, and I/O across a compute cluster using MQTT as the communication backbone. Though, this functionality is currently incomplete.

```python
class Hart:
    """Base Hart class defining interface for
       Hardware Thread abstractions"""
    def getHartAddrBusWidthInBytes(self) -> int:
        """Returns address bus width in bytes"""
        return None

    def setProgramCounter(self, newPC: int) -> bool:
        """Sets program counter with validation"""
        return True

    def execute(self) -> bool:
        """Executes a processor cycle"""
        pass

class Hart_RV64(Hart):
    """RISC-V 64-bit Hardware Thread implementation
       """
    def __init__(self):
        self.__HART_DATA_BUS_WIDTH_IN_BITS = 64
        self.__HART_ADDR_BUS_WIDTH_IN_BITS = 64
        self.__registerSetGP =
            GPRegSet_RV_32_64Bit_32Regs(
            register_width=64)
        self.__csr = CSR_RV64I()
        self.__isaDB = ISA_DB_RV64I_LP64()
```

Listing 1: Hart base class interface definition

2) Hart Component: The Hart component implements an abstract hardware thread with architecture-specific derivative implementations. In RISC-V nomenclature [2], a "hart" represents a hardware thread, and this component provides the execution context for a single CPU thread.

The base Hart class defines the generic interface that all implementations must support, including methods for program counter manipulation, register set(s) access, and generic instruction execution. FlexSiMArch's inheritance-based design allows specific architectural variants, such as Hart_RV64 and Hart_RV32, to derive from this base class and implement architecture-specific behaviors, register widths, and instruction handling.

Listing 1 shows simulator code excerpts for the base `Hart` class and a `Hart_RV64` specific implementation.

3) RAM Component: The RAM component handles memory operations, storage, and retrieval. It provides a flexible memory model for supporting research in diverse memory implementations. For example, virtual memory translation, cache implementations, transactive memory, type-aware memory, and advanced memory controllers. The RAM component supports any desired data bus, address bus, and word sizes, and storage/retrieval approaches.

A current implementation of the RAM component uses a dictionary-based and word-level approach to implement a simple, virtualized, but not translated memory controller. Plus, to minimize space overhead it stores only non-zero memory values. Listing 2, shows the code for one method in this implementation. Other memory approaches may be created and evaluated by extending the RAM base class.

```
def setWordValueFromBitArray(self, wordAddressAsUInt
    : int,
                             dataAsBA: bitarray) ->
                                bool:
    """Sparse memory model using dictionary storage
        """
    inputDataIsNonZeroValue = dataAsBA.any()
    refToCurrentValue = self.__ramWordMap.get(
        wordAddressAsUInt)
    addressKeyExists = (refToCurrentValue is not
        None)

    if (inputDataIsNonZeroValue and addressKeyExists
        ):
        # Update existing memory location
        self.__updateDataWordValueAtAddrKey(
            wordAddressAsUInt, dataAsBA)
        return True
    elif (inputDataIsNonZeroValue and not
        addressKeyExists):
        # Add new entry to sparse memory map
        self.__addAddrKeyAndDataWordValue(
            wordAddressAsUInt, dataAsBA)
        return True
    elif (not inputDataIsNonZeroValue and
        addressKeyExists):
        # Remove entry to maintain sparse
            representation
        self.__deleteAddrKeyAndWordValue(
            wordAddressAsUInt)
        return True
    elif (not inputDataIsNonZeroValue and not
        addressKeyExists):
        # Zero data, address not in dictionary -
            already zero
        return True
    else:
        # Unexpected state - should not occur
        return False
```

Listing 2: RAM sparse memory implementation: Sample method

4) Instruction Set Architecture Database: The ISA database (ISADB) specifies instruction sets. It was designed to provide a flexible mechanism for defining new instructions by separating the instruction specification and formatting from the implementation. The ISADB uses a declarative approach to define ISA instructions, with attributes specifying corresponding instruction properties. Each instruction definition includes mnemonic, base architecture and/or extension classes, category, instruction length, pseudo-instruction status, operand count, instruction description, and for each operand: type, length, and encoding. We believe that this approach enables researchers to quickly introduce and evaluate new instructions while minimizing needed simulator changes.

5) Execution Engine: The Executor component manages instruction execution implementing instruction semantics for all instructions defined in the ISADB database. During instruction execution, the Executor collaborates with the Hart component for execution context, the RegisterSet(s) for register storage and computation, the ISA Database for instruction definition and formatting, and the PPAI. The PPAI is the Portable Processor Assembly Instruction data-structure used within FlexSiMArch for extensible and portable instruction representation.

The Executor operates at the assembly level (text and not binary) facilitating human understanding and analysis. The Executor takes a binary executable (currently in *.elf* format, but extensible to any other binary executable format) and decompiles it with the help of the Python PWNTools library. Then it passes each instruction address and text to the PPAI constructor. Then each instruction is encoded as a PPAI object that is used by the Executor within the given execution context (Hart, RegisterSet(s), RAM(s), I/O(s)). This approach was chosen to facilitate ease of use and extensibility including the creation and evaluation of complex new instructions and/or pseudo-instructions. In the future we plan on separating some of this functionality into a `FetchAndDecode` component.

C. Console and HTIF Support

The human-machine interface (Console I/O) uses MQTT-based messaging for Node-to-I/O communication. The MQTT protocol enables reliable and asynchronous, and optionally authenticated and encrypted, message exchange between simulated processor Node(s) and I/O Console(s) (with Terminal and Keyboard) connected to that set of Nodes.

This approach facilitates the future addition of new features, for example, I/O storage and replay, I/O priority, and concurrent N-N node-console communication. These functions would be essential for complex distributed simulations and appear not supported in most currently available hardware simulators.

At the simulation level, FlexSiMArch currently supports the Host-Target Interface (HTIF) protocol. Figures 2a and 2b show sample console and debug window outputs, respectively.

IV. Key Features and Capabilities

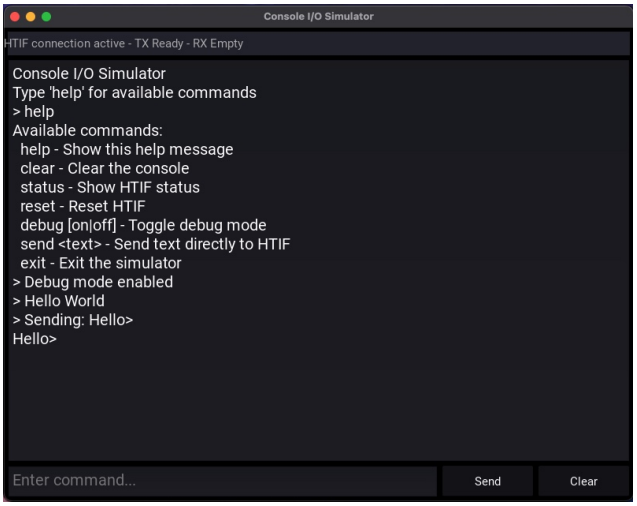
A. Ease of Extensibility

FlexSiMArch's inheritance-based design facilitates the fast creation and evaluation of new or enhanced ISAs and may represent a fundamental design innovation. Rather than creating separate simulators for each architecture, or generic simulators aimed at supporting all current ISAs, FlexSiMArch adopts a modular and object-oriented approach to enable the implementation of specialized (simulated) hardware components from abstract base classes.

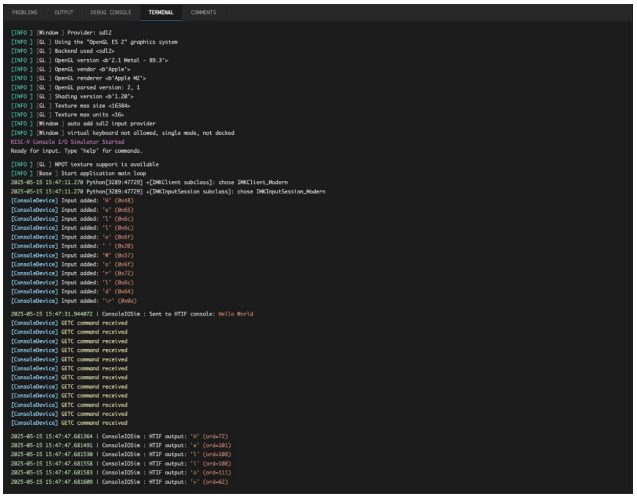
For example, the Hart base class defines the hardware thread interface, while derived classes like Hart_RV64 and Hart_RV32 provide architecture-specific implementations. Similarly, the GPRegSet class enables the creation disparate register set designs.

The ISA Database employs inheritance to extend basic instruction sets with new instructions or architectural variants. Each new instruction definition requires a single definition on the ISADB plus specification of the instruction semantics in the Executor class. This declarative approach abstracts implementation complexity while providing necessary information for instruction parsing and validation.

This design pattern provides the advantages of using high-level agnostic interfaces while still enabling specific or special



(a) FlexSiMArch's Console I/O interface.



(b) Terminal depicting I/O support with HTIF.

Fig. 2: Console I/O implementation in FlexSiMArch. The interface provides both character-level HTIF communication and higher-level command processing, enabling researchers to interact with simulated architectures through a unified interface.

purpose implementations. This aims to enable fast comparative evaluations of processor component and ISA designs.

V. Case Study: Vector Extension Implementation

To demonstrate FlexSiMArch's extensibility in practice, we present a detailed case study implementing a subset of the RISC-V Vector Extension (RVV) [28]. This case study illustrates the three-step process researchers follow when extending FlexSiMArch with new instructions: (1) defining instruction formats in the ISA Database, (2) implementing execution semantics in the Executor, and (3) adding specialized hardware components when needed. Table II details the 45 vector instructions implemented across six functional categories,

```python
def execute(self, ppai, regs, csr, pc):
    """Execute instruction across any supported
        architecture"""
    mnemonic = ppai.ppaiDict['ppaiInsMnemonicAsStr']

    # Determine architecture based on register width
    reg_width = regs.getRegWidthInBits()
    if reg_width == 32:
        instr_len = 4  # 32-bit architecture
    elif reg_width == 64:
        instr_len = 4  # 64-bit architecture
    else:
        instr_len = self._determine_instr_length(
            mnemonic, reg_width)

    # Dispatch to appropriate handler
    if mnemonic in self.common_instructions:
        return self._execute_common_instruction(
            mnemonic, ppai, regs, csr, pc, instr_len
            )
    elif mnemonic in self.arch_specific_instructions
        .get(reg_width, {}):
        return self.
            _execute_arch_specific_instruction(
            mnemonic, ppai, regs, csr, pc, instr_len
            , reg_width)
    elif mnemonic in self.custom_instructions:
        return self._execute_custom_instruction(
            mnemonic, ppai, regs, csr, pc, instr_len
            )
    else:
        raise NotImplementedError(f"Instruction {
            mnemonic} not implemented")
```

Listing 3: Flexible instruction execution framework: Execute method code

demonstrating coverage of configuration, arithmetic, reduction, comparison, and memory operations.

Table II details the vector instructions implemented for this case study.

A. Implementation Approach

The vector extension implementation required modifications to three simulator components, totaling 184 lines of code. This three-component approach demonstrates FlexSiMArch's separation of concerns: instruction syntax (ISA Database), execution behavior (Executor), and hardware state (Register Sets) are independently specified and modified.

Component breakdown: (1) ISA Database received 30 lines defining instruction formats and operand types (Listing 4), (2) Executor received 50 lines implementing execution semantics for vector operations (Listing 5), and (3) a new VectorRegSet component required 100 lines to manage vector register state and configuration (Listing 6). The total implementation time was approximately 1 day, including testing and validation.

B. Step 1: ISA Database Extension

The first step involved adding vector instruction definitions to the ISA database (ISADB), requiring minimal code changes focused on declarative instruction specifications. Listing 4 shows FlexSiMArch ISADB code for 3 of the 45 new instructions.

TABLE II: RISC-V Vector Extension Instructions Implemented in FlexSiMArch for this Case Study

Category	Instr.	Description	RV64V	RV32V
Vector Configuration				
	`vsetvli`	Set vector length (immediate)	•	•
	`vsetvl`	Set vector length (register)	•	•
Vector Arithmetic Operations				
	`vadd.vv`	Vector-vector addition	•	•
	`vadd.vs`	Vector-scalar addition	•	•
	`vadd.vi`	Vector-immediate addition	•	•
	`vsub.vv`	Vector-vector subtraction	•	•
	`vsub.vs`	Vector-scalar subtraction	•	•
	`vmul.vv`	Vector-vector multiplication	•	•
	`vmul.vs`	Vector-scalar multiplication	•	•
Vector Reduction Operations				
	`vredsum.vs`	Reduction sum	•	•
	`vredmax.vs`	Reduction maximum	•	•
	`vredmin.vs`	Reduction minimum	•	•
Vector Mask Operations				
	`vmseq.vv`	Mask: elements equal	•	•
	`vmsne.vv`	Mask: elements not equal	•	•
	`vmsltu.vv`	Mask: $v_1 < v_2$ (unsigned)	•	•
	`vmslt.vv`	Mask: $v_1 < v_2$ (signed)	•	•
Vector Memory Load Operations				
	`vle8.v`	Load 8-bit elements	•	•
	`vle16.v`	Load 16-bit elements	•	•
	`vle32.v`	Load 32-bit elements	•	•
	`vle64.v`	Load 64-bit elements	•	—
Vector Memory Store Operations				
	`vse8.v`	Store 8-bit elements	•	•
	`vse16.v`	Store 16-bit elements	•	•
	`vse32.v`	Store 32-bit elements	•	•
	`vse64.v`	Store 64-bit elements	•	—
Total Instructions			**24**	**22**

Each instruction definition in Listing 4 specifies seven key attributes: (1) mnemonic (e.g., 'vsetvli'), (2) supported architectures (['RV64V']), (3) functional category ('Vector-Config'), (4) instruction length in bits (32), (5) pseudo-instruction flag (False), (6) operand count (3), and (7) operand descriptors specifying type, encoding, width, and sign-ness. This declarative approach allows the simulator to automatically parse and validate new instructions without the need to develop new fetch and decode logic for each new instruction.

C. Step 2: Execution Logic Implementation

The second step involved implementing execution semantics for vector instructions in the Executor component:

The execution logic in Listing 5 demonstrates how FlexSiMArch dispatches instructions to appropriate handlers. For vector configuration (`vsetvli`), the executor extracts operands from the PPAI structure, configures the vector unit through the VectorRegSet, and updates the destination register with the actual vector length. For arithmetic operations (`vadd.vv`), the executor delegates element-wise computation to specialized methods in VectorRegSet, maintaining separation between control flow and computational logic.

```
def __populateVectorInstructions(self) -> None:
    """Populates ISA database with RISC-V vector
        extension instructions"""

    # Vector configuration instructions
    self.__addInstrToDBMap(
        'vsetvli',
        ['RV64V'],
        'Vector-Config',
        32, False, 3,
        'Dst|Reg|64|U',
        'Src|Reg|64|U', 'Src|Imm|12|U',
        'Set vector length and configuration')

    # Vector arithmetic operations
    self.__addInstrToDBMap(
        'vadd.vv',
        ['RV64V'],
        'Vector-Arithmetic',
        32, False, 3,
        'Dst|VReg|VLEN|S',
        'Src|VReg|VLEN|S',
        'Src|VReg|VLEN|S',
        'Vector-vector addition')

    # Vector memory operations
    self.__addInstrToDBMap(
        'vle32.v',
        ['RV64V'],
        'Vector-Load',
        32, False, 2,
        'Dst|VReg|VLEN|S',
        'Src|Reg+Off|64,12|A',
        None,
        'Vector load 32-bit elements')
```

Listing 4: Vector instruction database definitions.This listing has code for 3 of the 45 new in- structions.

```
def execute_vector_instruction(self, mnemonic, ppai,
     regs, vregs, csr, pc):
    """Execute vector instructions for RISC-V Vector
         Extension"""

    instr_len = regs.getRegWidthInBits() // 16

    if mnemonic == 'vsetvli':
        rd = ppai.ppaiDict['ppaiOp1RegIdxAsInt']
        rs1 = ppai.ppaiDict['ppaiOp2RegIdxAsInt']
        vtypei = ppai.ppaiDict['ppaiOp3ValuAsInt']
        avl = regs.getRegValueAsUIntByRegNr(rs1)
        vl = vregs.configure_vector_unit(avl, vtypei
            )
        if rd != 0:
            regs.setRegValueByRegNrFromUIntZero
            -ExtendToRegWidth(rd, vl)
        return pc + instr_len

    elif mnemonic == 'vadd.vv':
        vd = ppai.ppaiDict['ppaiOp1RegIdxAsInt']
        vs1 = ppai.ppaiDict['ppaiOp2RegIdxAsInt']
        vs2 = ppai.ppaiDict['ppaiOp3RegIdxAsInt']
        vregs.vector_add(vd, vs1, vs2)
        return pc + instr_len
```

Listing 5: Excerpt of Vector Instruction Execution Logic

```python
class VectorRegSet(GPRegSet):
    """Vector register file for RISC-V Vector
        Extension"""

    def __init__(self, num_registers=32,
        max_vector_length=1024):
        super().__init__()
        self.__num_vector_registers = num_registers
        self.__max_vector_length = max_vector_length
        self.__current_vector_length = 0
        self.__vector_element_width = 32
        self.__vector_registers = [[] for _ in range
            (num_registers)]

    def configure_vector_unit(self, avl, vtypei):
        """Configure vector unit and return actual
            vector length"""
        vsew = (vtypei >> 3) & 0x7  # Extract
            element width
        vlmul = vtypei & 0x7        # Extract length
             multiplier
        self.__vector_element_width = 8 << vsew
        vl = min(avl, self.__max_vector_length)
        self.__current_vector_length = vl
        return vl

    def vector_add(self, vd, vs1, vs2):
        """Element-wise vector addition"""
        for i in range(self.__current_vector_length)
            :
            if i < len(self.__vector_registers[vs1])
                 and i < len(self.__vector_registers
                [vs2]):
                result = self.__vector_registers[vs1
                    ][i] + self.__vector_registers[
                    vs2][i]
                mask = (1 << self.
                    __vector_element_width) - 1
                result &= mask
                if i < len(self.__vector_registers[
                    vd]):
                    self.__vector_registers[vd][i] =
                         result
                else:
                    self.__vector_registers[vd].
                        append(result)
```

Listing 6: Vector Register Set Case Study Implementation Code Exerpt

D. Step 3: Vector Register Implementation

The final step involved implementing a specialized register set to support vector operations, following FlexSiMArch's inheritance pattern. In FlexSiMArch, RegisterSet component(s) are in charge of managing the storage and retrieval of register values and also of implementing basic specialized operations such as arithmetic and binary operations within a given register.

The VectorRegSet implementation in Listing 6 extends FlexSiMArch's base register class to add vector-specific functionality. The `configure_vector_unit` method interprets the RISC-V `vtypei` encoding to set element width and vector length, following the RVV specification. Vector-specific operations, for example `vector_add`, iterate over the configured vector length, performing element-wise operations with automatic width masking. This design allows the same register set implementation to support multiple element widths (8, 16, 32, 64 bits) without separate code paths.

E. Case Study Cost Evaluation

The inclusion for support for the selected set of 45 vector instructions required less than 200 lines of code across three files (30 lines for instruction definitions, 50 lines for execution logic, 100 lines for vector register file). FlexSiMArch achieved this efficiency through a modular extension-focused architecture. This, by separating instruction definitions from implementation, isolating execution logic from register management, and aiming for a simulator that minimizes modifications during extensions.

VI. Validation Framework

FlexSiMArch includes flexible testing infrastructure to help ensure correctness and facilitating evaluation during development and extension.

For unit tests, each simulator component (i.e.: Hart, GPRegSet, RAM, ...) has an associated `_Test` file containing several tests for each method on the respective component. This file is an executable Python file that automatically runs all defined unit tests (all functions ending in `_test`) for a given component and aggregates and reports unit test results to a `TestManager`.

For integration tests, we implemented two approaches (1) Instruction (ISA) coverage and (2) Expected memory state. In the near future, we plan to add I/O testing.

A. Target Testsuite and ISA Coverage

We ran integration tests against the RISC-V Toolset ISA Tests [29]. We removed all source files not related to RV32I and RV64I. This resulted in a test suite containing 1,025 source files within 84 folders. In this case, the binary *.elf* files passed into Spike and FlexSiMArch were compiled and linked from their respective C source files using the standard RISC-V compiler tool-chain. For this compilation, we removed the compressed option in the RISC-V Toolset ISA Tests `makefile`. RISC-V defines a compressed code option where some instructions maybe encoded using 16-bits instead of the standard fixed 32-bit instruction length.

Instruction coverage is understood as the target binary (*.elf*) successfully completed execution without failures within both simulators (Spike and FlexSiMArch). FlexSiMArch successfully executed all binaries on the resulting test suite albeit without corresponding I/O.

B. Expected Memory State Validation

We created a JSON-based specification for defining memory regions and their expected values. This approach enables the comparison of the state of (simulated) memory, after the execution of a target binary, between FlexSiMArch and Spike. This approach also enables the incremental addition of test-cases, and future automated test-case generation. Listing 7 shows an example of this JSON test specification for a `32_bit.elf` binary.

```
{
  "test_suite": "Auto_Generated_32_bit.elf",
  "elf_file": "./boot/32_bit.elf",
  "timestamp": "20250404_000823",
  "test_cases": [
    {
      "name": "auto_test_0",
      "size": 256,
      "start_address": "1000"
    },
    {
      "name": "auto_test_1",
      "size": 512,
      "start_address": "2000"
    }
  ]
}
```

Listing 7: JSON test configuration structure

For this test suite, we implemented an iterative testing loop. In each iteration, both Spike and FlexSiMArch execute the target `.elf` file, and their resulting memory states are compared using `diff`. When discrepancies are detected in the diff output, JSON configuration files are automatically generated to specify the exact memory regions where differences occurred, enabling detailed investigation of the mismatch. Conversely, when no differences are found (indicating correct execution), the system automatically generates JSON files with randomly selected memory regions to expand test coverage. These automatically generated JSON files can be manually refined later to focus on specific memory regions of interest or to add targeted test cases. FlexSiMArch achieved 100% pass rate on all automatically generated tests in this manner.

VII. Limitations and Considerations

While FlexSiMArch offers significant advantages, several limitations should be acknowledged. The Python implementation, while enhancing code clarity and extensibility, may result in slower execution compared to C/C++-based simulators. However, this trade-off is justified by the substantial reduction in development complexity and the enabling of rapid architectural experimentation.

The current implementation focuses primarily on RISC-V architectures, though the inheritance-based design facilitates extension to other architectural families and hybrid architectures. The validation framework, while comprehensive for supported architectures, would require extension for broader architectural coverage.

VIII. Conclusion and Potential Target Applications

FlexSiMArch is already helping accelerate secure and hybrid processor architecture research by providing a modular, flexible, and architecture-independent simulation platform.

Areas of investigation that FlexSiMArch aims to support include: (1) Processor- and/or device-integrated security features and their properties; (2) Vulnerability exposures and potential mitigations; (3) Memory implementation approaches, for example, transactive memory, object-based security enabled memory, and novel virtual memory management approaches; (4) Hybrid computation platforms with multiple processor types; for example, combined binary and other digital, artificial neural, optical, vector- and/or matrix-optimized, and/or quantum; (5) Novel hardware-level security features, such as BHPol and/or typed assembly.

IX. Future Development Directions

In this section, we detail several needed or potential areas for future improvement.

Binary Hierarchical Policy Management: Full integration of the BHPol (Binary Hierarchical Policy) technology [30]. BHPol is the binary implementation version of HPol (Hierarchical Policy Framework). In HPol, a policy is a tuple (or path) of security-relevant entities or states which belong to a graph. BHPol leverages binary notation and lexicographical ordering to encode security-related entities/states hierarchically (in a partial order). BHPol enables the efficient and hardware-level encoding and querying of hierarchical (including multi-level) security policies. For additional details on the BHPol and HPol technologies we point readers to these related articles [30]–[32].

Simulated Performance Measurement: The addition of measurement counters and variables across all components is functionality we intend to add in the near future. Using FlexSiMArch, we would like to be able to, for example, (1) Compare the number of expected processor cycles used in the execution of a given set of binaries across different ISAs (within or across the same, extended, or disparate architectures); (2) Compare the expected latency and throughput of different RAM (or other types) module implementations. Though, enabling performance gains in current processors is not one of FlexSiMArch objectives, evaluating the performance effects of new security features is an important area of need.

Advanced Security Analysis: Development of specialized security analysis tools within FlexSiMArch to accelerate vulnerability and mitigation research, including information flow tracking, boundary analysis, and other novel hardware-level exploit detection and/or prevention mechanisms.

Additional Architecture Support: The addition of support for other widely used architectures, for example x86 (AMD/Intel) and Arm, would enhance FlexSiMArch's utility in hardware and firmware security research.

Performance Optimization: Future work will focus on improving simulation speed while preserving extensibility. Planned enhancements include: (1) JIT compilation using PyPy or Numba to accelerate frequently-executed instructions, (2) Optional C/C++ modules for performance-critical operations such as memory access, (3) Parallel execution support for multi-hart simulations, and (4) Instruction caching to eliminate redundant parsing. These improvements aim to make FlexSiMArch suitable for larger and longer simulation runs.

ACKNOWLEDGMENT

This research was supported by Schweitzer Engineering Laboratories and the State of Idaho.

REFERENCES

[1] MITRE Corporation, "2024 cwe top 10 kev weaknesses," 2024, common Weakness Enumeration. Accessed: 2025-09-20. [Online]. Available: https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html

[2] RISC-V International, "The risc-v instruction set manual: Volume one: Unprivileged isa," May 2025, accessed: Oct. 2025. [Online]. Available: https://docs.riscv.org/reference/isa/_attachments/riscv-unprivileged.pdf

[3] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41. USENIX, 2005, p. 46.

[6] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 2007, pp. 23–34.

[7] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, pp. 1–12.

[8] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2009, pp. 469–480.

[9] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2012, pp. 335–344.

[10] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 475–486.

[11] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: A full system simulator for multicore x86 cpus," in *48th ACM/EDAC/IEEE Design Automation Conference*. IEEE, 2011, pp. 1050–1055.

[12] RISC-V International, "Spike risc-v isa simulator," 2020. [Online]. Available: https://github.com/riscv/riscv-isa-sim

[13] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio *et al.*, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016.

[14] L. Auer, C. Skubich, and M. Hiller, "A security architecture for risc-v based iot devices," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2019, pp. 1077–1082.

[15] D. Camarmas-Alonso, F. Garcia-Carballeira, A. Calderon-Mateos, and E. Del-Pozo-Puñal, "Creator: An educational integrated development environment for risc-v programming," *IEEE Access*, vol. 12, pp. 119 850–119 861, 2024.

[16] T. McGrew, E. Schonauer, and P. Jamieson, "Framework and tools for undergraduates designing risc-v processors on an fpga in computer architecture education," in *International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 2019, pp. 846–851.

[17] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa," EECS Department, UC Berkeley, Tech. Rep., 2016.

[18] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, K. Asanović, and D. Patterson, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, 2016.

[19] L. Feng, J. Huang, L. Li, H. Zhang, and Z. Wang, "Rvdfi: A risc-v architecture with security enforcement by high performance complete data-flow integrity," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2499–2512, 2022.

[20] N. H. Nguyen, D. H. A. Le, V. T. D. Le, V. T. Nguyen, T. H. Vu, and H. L. Pham, "Li-rv: A fast and efficient risc-v based coprocessor for lightweight cryptography," in *21st International SoC Design Conference (ISOCC)*. IEEE, 2024, pp. 1–6.

[21] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security: Secverilog," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 1–14.

[22] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *37th International Symposium on Microarchitecture (MICRO-37)*. IEEE, 2004, pp. 243–254.

[23] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1–19.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*. USENIX, 2018, pp. 973–990.

[25] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens *et al.*, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *USENIX Security Symposium*. USENIX, 2018, pp. 991–1008.

[26] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 1285–1300.

[27] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1. ACM, 2017, pp. 585–598.

[28] RISC-V International, "Risc-v vector extensions v1.0," September 2021. [Online]. Available: https://github.com/riscvarchive/riscv-v-spec/releases/tag/v1.0

[29] ——, "Risc-v tools: Tests," May 2025. [Online]. Available: https://deepwiki.com/riscv-software-src/riscv-tools/2.2-riscv-tests

[30] S. Mankotia, D. Conte de Leon, and J. Johnson-Leung, "Hierarchical firmware-level security policy for industrial control systems," in *2025 IEEE 5th Cyber Awareness and Research Symposium (CARS)*. IEEE, October 2025, to appear in IEEE Xplore.

[31] D. Conte de Leon, M. Brown, A. Jillepalli, A. Stalick, and J. Alves-Foss, "High-level and formal router policy verification," *Journal of Computing Sciences in Colleges*, vol. 33, no. 1, pp. 1–12, October 2017.

[32] A. H. Alkhorem, D. Conte de Leon, A. A. Jillepalli, and J. Song, "Formalizing permission to delegate and delegation with policy interaction," *Sensors*, vol. 25, no. 16, p. 4915, August 2025.