Bridging Worlds: A Performance and Feasibility Analysis of RISC-V Integration with the Ethereum Virtual Machine

Full Draft Report

Nwele Uchenna (developeruche)

x.com/developeruche · github.com/developeruche

18 April 2025

Abstract

This is a design-and-feasibility report: we describe an EVM-compatible execution environment built on the RISC-V instruction set, and we report no performance measurements. We implement an RV32IM interpreter and assembler in Rust, map EVM opcodes onto RV32IM by hand, and route every access to blockchain state through an environment-call surface of 44 ecalls, so that contract logic executes as ordinary RISC-V code while storage, account and block context remain outside the sandbox. A counter contract written in RV32IM assembly deploys and executes against this VM, storing RISC-V bytecode in the account code slot that normally holds EVM bytecode. Our central finding concerns the register budget rather than speed. The calling convention leaves 30 of the 32 architectural registers free for ecall arguments, and because a 256-bit EVM word occupies eight of them, the opcodes that carry several words saturate the file: CALL consumes 26 registers and SSTORE 16, while LOG2, LOG3, LOG4, DELEGATECALL, STATICCALL and CALLCODE have no calling convention in the implementation at all. LOG4 alone requires 34 argument registers and cannot be expressed in a register-only mapping; supporting it needs a memory-backed calling convention we have not built. We make no claim about execution speed relative to the EVM: gas metering is absent and the REVM integration is incomplete, so no comparison against an optimised EVM implementation was possible.

Contents

1 Introduction

Ethereum's Virtual Machine (EVM) remains the cornerstone of its smart contract functionality, providing a sandboxed execution environment that has enabled a diverse ecosystem of decentralized applications. However, as blockchain technology matures, questions emerge about the long-term viability and limitations of the current EVM architecture. This research investigates an alternative approach by exploring RISC-V, an open instruction set architecture, as a potential foundation for executing smart contracts.

The convergence of blockchain technology and RISC-V presents intriguing possibilities. RISC-V's open nature, simplicity, and growing ecosystem make it an attractive candidate for reimagining blockchain execution environments. By enabling smart contracts to be written in languages that target RISC-V, this integration would broaden the set of toolchains that can produce contract code.

This research addresses several fundamental questions:

1. Can RISC-V effectively implement the functionality required for EVM-compatible smart contract execution?
2. How might this integration affect the developer experience and ecosystem?
3. What fundamental design challenges must be overcome for practical implementation?

Scope. This is a design-and-feasibility report. We built the interpreter, the assembler

and the ecall surface, and we ran a contract on them; we did not measure execution speed, and we make no claim about it. Gas metering is not implemented, so no cost comparison is possible, and the REVM integration described in Section 3.2 is incomplete, so the RISC-V interpreter has not been timed against a production EVM. Every architectural finding we report below is a static property of the mapping (register counts, calling conventions, which opcodes have one) rather than a measurement.

2 Background and Related Work

2.1 The Ethereum Virtual Machine (EVM)

The EVM is a quasi-Turing complete, 256-bit stack machine, specified normatively in the Yellow Paper [1] and, in executable form, in the execution-layer specification [2]. Its key components are a stack, volatile memory, and persistent storage. Execution proceeds by interpreting bytecode instructions (opcodes) that manipulate these components, and gas mechanics regulate computational cost. The 256-bit word is the source of most of the friction: it is four times the width of the registers on the 64-bit hardware that actually runs the interpreter, so every arithmetic operation is emulated in software. That emulation cost is what motivated the EVM optimisation work of 2016 [3] and the just-in-time compiler shipped and later withdrawn from Go Ethereum [11].

2.2 The RISC-V ISA

RISC-V is an open ISA based on reduced instruction set computing (RISC) principles [4, 5]. Its base integer ISA (e.g., RV32I, RV64I) is small and can be extended with standard extensions (e.g., 'M' for integer multiplication/division, 'A' for atomics, 'C' for compressed instructions). This modularity allows tailoring processors for specific needs. A significant advantage is the mature and growing compiler support (GCC, LLVM), enabling compilation from various languages.

2.3 Alternative Blockchain Execution Environments

WASM-based. WebAssembly was designed as a portable, low-level compilation target with a formal semantics and near-native performance [6], which makes it the obvious candidate for a chain that wants an existing toolchain rather than a bespoke one. Polkadot places a WASM runtime at the centre of its design, so that a chain's state-transition function is itself a WASM blob that can be upgraded without a hard fork [7]. EOSIO built its own WASM backend, eos-vm, for lower dispatch latency than a general-purpose engine [8]. Both keep the sandbox and replace only the instruction set. We take the same position, substituting RISC-V for WASM, and inherit the same question they face: how contract code reaches chain state once the instruction set no longer has opcodes for it.

Register-based VMs. Solana compiles contracts written in Rust and C through LLVM to a register-based bytecode and schedules non-overlapping transactions in parallel [9]. That design shows a register machine is workable as a contract VM, but it was free to choose its account model and its calling convention together. We are not: our target is EVM semantics on a register machine, and Section 6 shows the register file is exactly where that constraint binds.

RISC-V for blockchain execution. RISC-V has been proposed before as a state-transition target, most directly for L2 execution [12]. That line of work treats RISC-V as a proving or verification target rather than as the L1 contract VM. We instead map EVM

opcode semantics onto RV32IM directly and keep the account and storage model unchanged.

2.4 Bridging EVM and Other Architectures

EVM JIT and transpilation. Rather than replace the instruction set, this line of work keeps EVM bytecode and compiles it at run time. The evmjit library translated EVM bytecode to LLVM IR [10], and Go Ethereum shipped a JIT-EVM on the same premise [11]; neither survived in production, and the reasons given at the time were compilation latency and the difficulty of keeping a second execution path bug-compatible with the interpreter. Our work differs in target rather than in technique: we do not compile EVM bytecode faster, we replace it, executing contract logic as native RV32IM and leaving the state model alone.

3 Methodology: Designing and Implementing a RISC-V EVM

To achieve these goals, the applied methodology would be broken into a two-phase approach:

Phase One: Custom RISC-V EVM Architecture

1. Design of the core components of this new RISC-V EVM VM; this would also involve blockchain related operations fashioned as `opcodes` (Rust implementation).
2. Implementation of a RISC-V IM32 assembler compatible with blockchain requirements. This would be tasked to assemble RISC-V assembly written smart contracts to RISC-V machine code (code).
3. Optimization of the implementation for performance and security spec constraints.
4. Benchmarking and preliminary analysis. *(Planned; not carried out in this draft. See Scope in Section 1.)*

Phase Two: Integration with Existing Ethereum Runtime

1. Adaptation of custom RISC-V EVM to the REVM (Rust Ethereum Virtual Machine) API.
2. Implementation of blockchain-specific operations (execution of blocks and transactions) within RISC-V constraints.
3. Comparative performance testing between custom implementation and standard EVM.
4. Analysis of register utilization and optimization opportunities.

3.1 Phase One: Custom RISC-V EVM Implementation

1. **Architecture Design.** We chose the RV32IM subset of the RISC-V ISA as the target, providing basic integer operations and multiplication/division, deemed sufficient for initial EVM opcode mapping. The design focused on creating a RISC-V interpreter capable of managing state analogous to the EVM's (stack, memory, simulated storage access). During this phase blockchain related operations fashioned as ecalls were ignored. Here is the code implementing this phase. *It is important to disclose this code was implemented before this research idea was conceived, making this implementation not connected to the blockchain in any way, which is exactly the point.*
2. **EVM Opcode Mapping.** Each relevant EVM opcode was mapped to a sequence of RV32IM instructions. Simple arithmetic opcodes (`ADD`, `SUB`) translated relatively directly, while stack manipulations (`PUSH`, `POP`, `DUP`, `SWAP`) required careful management of RISC-V registers or memory to simulate the EVM stack. More complex opcodes presented significant challenges:

- `SSTORE`: Requires interaction with the storage mechanism and involves multiple parameters, straining register availability.
- `LOG*`: Opcodes like `LOG2`, `LOG3`, `LOG4` consume numerous inputs (memory offset/length, topics), leading to high register pressure, sometimes exceeding available registers in a simple mapping.
- `CALL` variants: Involve managing call frames, arguments, and return values, demanding complex sequences of RISC-V instructions.

3. **RISC-V Interpreter/Assembler.** A basic interpreter for the chosen RV32IM instruction subset was implemented in Rust. This interpreter managed the RISC-V register file and simulated memory access. A rudimentary assembler was developed to convert mapped EVM logic (represented as RISC-V instruction sequences) into executable format for the interpreter (code).
4. **Handling Blockchain Context.** Direct interaction with blockchain state (storage, account balances, block information) is essential. We designed an abstraction layer using simulated **environment calls** (`ecall`s in RISC-V terminology). This approach mirrors the concept outlined in the `counter_riscvim32_smart_contract_asm` example, where specific `ecall` instructions trigger host functions to retrieve context information (e.g., `env::block_number()`, `storage::get()`). This isolates blockchain-specific logic from the core RISC-V execution of contract logic but necessitates developers using these specific library calls (see).

3.2 Phase Two: Adaptation to REVM API

To facilitate comparison and leverage a mature EVM framework, we adapted our custom RISC-V execution core to fit within the REVM architecture. REVM provides well-defined traits and structures for EVM components like the `Interpreter`, `Host`, and `Database`.

1. **Integration.** I implemented REVM's `Interpreter` trait, replacing its core instruction loop with calls to our RISC-V interpreter for executing the mapped EVM bytecode (now represented as RISC-V code).
2. **State Management.** REVM's `Database` and `Host` interfaces were used to handle state lookups (storage, balances, code) required by the execution, replacing the simpler simulation used in Phase One. This allowed interaction with more realistic EVM state representations.
3. **Goal.** The primary goal of this phase was to enable more direct (though still preliminary) comparisons by running the same contract logic through both a standard REVM interpreter and our RISC-V-based interpreter operating within the same REVM framework.

This phase is to be completed on the final research result.

4 Experimentation and Test on Draft Implementation

For the purpose of this research a draft implementation of this VM has been implemented, and a `riscv_assembler` as well. The code can be found here:

1. Draft RISCVIM32-EVM: `https://github.com/developeruche/riscv-evm-experiment/tree/main/crates/research-draft/riscv_evm/src`
2. RISCVIM32 assembler: `https://github.com/developeruche/riscv-assembler`

To test the compatibility of the VM, a simple counter smart contract written as RISC-V assembly was deployed and executed on this VM. Here is what this smart contract looks like:

```
# SPDX-License-Identifier: MIT
# Simple Counter Contract in RISC-V RV32IM Assembly

# --- Ecall Definitions (Using the same as provided) ---
.equ ECALL_KECCAK256, 0x20    # [offset, size] -> hash
.equ ECALL_ADDRESS, 0x30      # Address of the current executing contract |-> address
.equ ECALL_BALANCE, 0x31      # Native balance of the current caller [address] -> balance
.equ ECALL_ORIGIN, 0x32       # Address of the transaction origin |-> address
.equ ECALL_CALLER, 0x33       # Address of the current calling address |-> address
.equ ECALL_CALLVALUE, 0x34    # Deposit value for this Tx |-> value
.equ ECALL_CALLDATALOAD, 0x35 # Load a Word(256bits) from the calldata [i] -> data[i]
.equ ECALL_CALLDATASIZE, 0x36 # Returns the size of the calldata |-> usize
.equ ECALL_CALLDATACOPY, 0x37 # Copy calldata from input to memory [destOffset, offset, size]
.equ ECALL_CODESIZE, 0x38     # Returns the size of the code |-> usize
.equ ECALL_CODECOPY, 0x39     # Copy code from input to memory [destOffset, offset, size]
.equ ECALL_GASPRICE, 0x3A     # Gas price now
.equ ECALL_EXTCODESIZE, 0x3B  # Get the size of an External account's code [address] -> usize
.equ ECALL_EXTCODECOPY, 0x3C  # Get the code of an External account [address, destOffset,
   offset, size]
.equ ECALL_RETURNDATASIZE, 0x3D # Get size of output data from the previous call
.equ ECALL_RETURNDATACOPY, 0x3E # Copy output data from the previous call [destOffset, offset, size]
.equ ECALL_EXTCODEHASH, 0x3F  # Get hash of an account's code [address] -> hash
.equ ECALL_BLOCKHASH, 0x40    # Get hash of recent block [blockNumber] -> hash
.equ ECALL_COINBASE, 0x41     # Get the block's beneficiary address |-> address
.equ ECALL_TIMESTAMP, 0x42    # Get the block's timestamp |-> timestamp
.equ ECALL_NUMBER, 0x43       # Get the block's number |-> blockNumber
.equ ECALL_PREVRANDAO, 0x44   # Get the block's difficulty |-> difficulty
.equ ECALL_GASLIMIT, 0x45     # Get the block's gas limit |-> gasLimit
.equ ECALL_CHAINID, 0x46      # Get the chain ID |-> chainId
.equ ECALL_SELFBALANCE, 0x47  # Get balance of currently executing account |-> balance
.equ ECALL_BASEFEE, 0x48      # Get the base fee |-> baseFee
.equ ECALL_BLOBHASH, 0x49     # Get versioned hashes [index] -> blobVersionedHashesAtIndex
.equ ECALL_BLOBBASEFEE, 0x4A  # Returns the blob base-fee of the current block |-> blobBaseFee
.equ ECALL_GAS, 0x5A          # Amount of available gas
.equ ECALL_LOG0, 0xA0         # Append log record with no topics [offset, size]
.equ ECALL_LOG1, 0xA1         # Append log record with one topic [offset, size, topic]
.equ ECALL_LOG2, 0xA2         # Append log record with two topics [offset, size, topic1, topic2]
.equ ECALL_LOG3, 0xA3         # Append log record with three topics [offset, size, topic1,
   topic2, topic3]
.equ ECALL_LOG4, 0xA4         # Append log record with four topics [offset, size, topic1-4]
.equ ECALL_CREATE, 0xF0       # Create a new account with code [value, offset, size] -> address
.equ ECALL_CALL, 0xF1         # Call into an account [gas, address, value, argsOffset, argsSize,
   retOffset, retSize]
.equ ECALL_CALLCODE, 0xF2     # Call with alternative code [gas, address, value, argsOffset,
   argsSize, retOffset, retSize]
.equ ECALL_RETURN, 0xF3       # Halt execution returning output data [offset, size]
.equ ECALL_DELEGATECALL, 0xF4 # Call with alternative code, persisting sender and value [gas,
   address, argsOffset, argsSize, retOffset, retSize]
.equ ECALL_CREATE2, 0xF5      # Create account with predictable address [value, offset, size,
   salt] -> address
.equ ECALL_STATICCALL, 0xFA   # Static call into an account [gas, address, argsOffset, argsSize,
   retOffset, retSize]
.equ ECALL_REVERT, 0xFD       # Halt execution reverting state changes [offset, size]
.equ ECALL_SLOAD, 0x54        # Loads a word (32-bytes) from storage
.equ ECALL_SSTORE, 0x55       # Stores a word (32-bytes) to storage


# --- Storage Slot Definitions ---
.equ SLOT_COUNTER_1, 0
.equ SLOT_COUNTER_2, 0
.equ SLOT_COUNTER_3, 0
.equ SLOT_COUNTER_4, 0
.equ SLOT_COUNTER_5, 0
```

```
.equ SLOT_COUNTER_6, 0
.equ SLOT_COUNTER_7, 0
.equ SLOT_COUNTER_8, 0


.text           # Entry point for deployment (initcode)

# ============================================================
# INITCODE SECTION
# Runs only once during deployment.
# Sets up initial state and returns the runtime code.
# ============================================================
_start:
    # --- Initialize Counter Value to 0 ---
    addi x1, zero, SLOT_COUNTER_1
    addi x2, zero, SLOT_COUNTER_2
    addi x3, zero, SLOT_COUNTER_3
    addi x4, zero, SLOT_COUNTER_4
    addi x5, zero, SLOT_COUNTER_5
    addi x6, zero, SLOT_COUNTER_6
    addi x7, zero, SLOT_COUNTER_7
    addi x8, zero, SLOT_COUNTER_8

    addi x9, zero, 0
    addi x10, zero, 0
    addi x11, zero, 0
    addi x12, zero, 0
    addi x13, zero, 0
    addi x14, zero, 0
    addi x15, zero, 0
    addi x16, zero, 1

    # Store initial counter value to storage
    addi x31, zero, ECALL_SSTORE
    ecall

    # --- Return Runtime Code ---
    # Calculate the size and offset of the runtime code section
    addi x5, zero, runtime_code_start # Get address of runtime code start
    addi x6, zero, runtime_code_end

    sub x3, x6, x5            # x3 = length of runtime code

    # Return the copied code
    add x1, zero, x5                # mem_offset = x5
    add x2, zero, x3                # length of runtime code
    addi x31, zero, ECALL_RETURN
    ecall

    # End of initcode. Should not be reached after ECALL_RETURN.


# ============================================================
# RUNTIME CODE SECTION
# This code is stored on the blockchain after deployment.
# It handles subsequent calls to the contract.
# ============================================================
runtime_code_start:
    # --- Runtime Entry Point & Function Dispatcher ---
    # Read function selector (first 4 bytes of calldata)
    addi x1, zero, 0                # offset = 0
    addi x31, zero, ECALL_CALLDATALOAD  # setup for the ECALL
    ecall                     # Returns first 32 bytes: Note in this case x2 would hold the first  ↩
   4 bytes


    # Function dispatcher - compare with known selectors
    addi x3, zero, 0x00000037   # Selector for increment()
```

```
    beq x2, x3, _increment

    addi x3, zero, 0x00000020   # Selector for getValue()
    beq x2, x3, _getValue

    addi x3, zero, 0x00000055   # Selector for setValue(uint256)
    beq x2, x3, _setValue

    # Fallback: If no function matches, revert
    jal x0, _revert_default

# --- Function Implementations ---

_increment:
    # Increment the counter value by 1
    # 1. Load current counter value
    # 2. Add 1
    # 3. Store new counter value
    # 4. Return success (true)

    # Load current counter value
    addi x1, zero, SLOT_COUNTER_1
    addi x2, zero, SLOT_COUNTER_2
    addi x3, zero, SLOT_COUNTER_3
    addi x4, zero, SLOT_COUNTER_4
    addi x5, zero, SLOT_COUNTER_5
    addi x6, zero, SLOT_COUNTER_6
    addi x7, zero, SLOT_COUNTER_7
    addi x8, zero, SLOT_COUNTER_8

    addi x31, zero, ECALL_SLOAD  # setup for the ECALL
    ecall                     # Returns counter value in [x9 - x16]


    # Increment counter (need to handle potential overflow from any of the limbs)
    addi x16, x16, 1            # counter_last_limb += 1, next is to check for overflow and propagate if need be
    beq x16, zero, _inc_overflow      # If counter_last_limb wrapped to 0, increment counter_last_limb - 1
    jal x0, _inc_store

_inc_overflow:
    addi x15, x15, 1            # counter_last_limb-1 += 1
    bne x15, zero, _inc_store      # If counter_last_limb-1 does not wrap to 0, store because there was no overflow

    # Increment counter_last_limb - 2
    addi x14, x14, 1
    bne x14, zero, _inc_store

    # Increment counter_last_limb - 3
    addi x13, x13, 1
    bne x13, zero, _inc_store

    # Increment counter_last_limb - 4
    addi x12, x12, 1
    bne x12, zero, _inc_store

    # Increment counter_last_limb - 5
    addi x11, x11, 1
    bne x11, zero, _inc_store

    # Increment counter_last_limb - 6
    addi x10, x10, 1
    bne x10, zero, _inc_store

    # Increment counter_last_limb - 7
    addi x9, x9, 1
```

```
    bne x9, zero, _inc_store


    # some missing code see the link below;


    # Store initial counter value to storage
    addi x31, zero, ECALL_SSTORE
    ecall

    # Return success (true = 1)
    jal x0, _return_true

# --- Helper Routines ---

_return_true:
    # Prepare return value 'true' (uint256(1))
    addi x2, zero, -48           # Allocate stack space
    addi x31, zero, ECALL_SSTORE
    addi x3, zero, 0
    addi x4, zero, 1

    sw x3, 0(x2)             # Store counter_limb_1 at sp
    sw x3, 4(x2)             # Store counter_limb_2 at sp+4
    sw x3, 8(x2)             # Store counter_limb_3 at sp+8
    sw x3, 12(x2)            # Store counter_limb_4 at sp+12
    sw x3, 16(x2)            # Store counter_limb_5 at sp+16
    sw x3, 20(x2)            # Store counter_limb_6 at sp+20
    sw x3, 24(x2)            # Store counter_limb_7 at sp+24
    sw x4, 28(x2)            # Store counter_limb_8 at sp+28

    # Return true u256(1)
    add x1, zero, x2               # mem_offset = 0
    addi x2, zero, 32               # length = 8 bytes (uint256)
    addi x31, zero, ECALL_RETURN
    ecall

    # Clean up stack (should not be reached after ECALL_RETURN)
    addi x2, x2, 8
    jal x0, _return_true

_revert_default:
    addi x1, zero, 0
    addi x2, zero, 0
    addi x31, zero, ECALL_REVERT
    ecall

runtime_code_end:              # Mark the end of the runtime code section
```

Complete counter RISC-V smart contract: code.

In order to deploy this contract and make a call, here is the API design for this draft implementation:

```
use revm::{
    Context as RevmEthContext, DatabaseCommit, MainContext,
    context::{ContextTr, JournalTr},
    database::CacheDB,
    primitives::{Address, U256},
};
use riscv_assembler::assembler::assemble;
use riscv_evm::{
    context::Context,
    ecall_manager::process_ecall,
    riscv_evm_core::{MemoryChuckSize, e_constants::*, interfaces::MemoryInterface},
    utils::{bytes_to_u32, u32_vec_to_address, u32_vec_to_bytes},
```

```
    vm::Vm,
};
mod contract;

fn main() {
    let from_addr = Address::from([
        0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99,
        0x00, 0xAA, 0xBB, 0xCC, 0xDD,
    ]);

    let mut vm = Vm::new();
    let eth_context = RevmEthContext::mainnet().with_db(CacheDB::default());
    let mut context = Context::new(eth_context);

    context.current_caller = from_addr;

    // ==================================
    // deploying the counter contract
    // ==================================
    let init_code = assemble(contract::ASSEMBLE_CODE).unwrap().code;
    let init_code = u32_vec_to_bytes(&init_code, init_code.len() * 4);
    let init_offset = 900;

    // Write init code to memory
    for (i, &byte) in init_code.iter().enumerate() {
        vm.memory
            .write_mem(init_offset + i as u32, MemoryChuckSize::BYTE, byte as u32);
    }

    // Set up Create ECALL
    vm.registers.write_reg(ECALL_CODE_REG, 0xF0); // Create
    vm.registers.write_reg(CREATE_INPUT_REGISTER_1, init_offset);
    vm.registers
        .write_reg(CREATE_INPUT_REGISTER_2, init_code.len() as u32);

    // Set value (0.0..01ETH)
    let value = 10_000_000_000_000_000_000u64;
    let value_bytes: [u8; 32] = U256::from(value).to_be_bytes();
    for i in 0..8 {
        vm.registers.write_reg(
            CREATE_INPUT_REGISTER_3 + i as u32,
            bytes_to_u32(&value_bytes[i * 4..(i + 1) * 4]),
        );
    }

    // Load account of the creator so it can transfer value
    context
        .eth_context
        .journal()
        .load_account(context.address)
        .unwrap();

    // Going ahead to execute this ecall
    let result = process_ecall(&mut vm, &mut context).unwrap();

    // Commit all state changes to database
    for i in result {
        context.eth_context.db().commit(i.state);
    }

    // Check that output registers were written
    let addr1 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_1);
    let addr2 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_2);
    let addr3 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_3);
    let addr4 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_4);
    let addr5 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_5);

    // Reconstruct address to check it's valid
```

```
    let address_bytes = u32_vec_to_address(&[addr1, addr2, addr3, addr4, addr5]);
    println!("Address: {:?}", Address::from(address_bytes));

    context
        .eth_context
        .journal()
        .load_account(Address::from(address_bytes))
        .unwrap();
    let new_contract = context
        .eth_context
        .journal()
        .load_account_code(Address::from(address_bytes))
        .unwrap()
        .clone()
        .info
        .code
        .unwrap();
    println!("This is the runtime code: {:?}", new_contract); // NOTE: this runtime code is RISC-V bytecode, not native EVM bytecode
}
```

The aim of the draft implementation is to show that there is little to no change to the main Ethereum protocol in achieving this complete change of the execution sandbox (EVM to RISC-V EVM): storing the RISC-V bytecode in the same location (account code section) where the EVM bytecode is stored, replacing `stack` related operations with `register` operations, and so on.

5 Implementation Status

This section records what the draft implementation covers and what it does not. None of it is a performance measurement; the figures below are static properties of the mapping, read off the implementation's calling conventions.

Arithmetic, logic and stack opcodes map onto RV32IM without difficulty: each becomes a short instruction sequence, and the EVM stack is simulated in registers and memory. The host interface was the part we expected to fight and did not. REVM's `Context` [13], which carries storage, balances and block data, plugs into the draft unchanged, so the RISC-V core reuses an existing state layer rather than a bespoke one.

The environment-call surface is complete at 44 calls, covering `KECCAK256`, the account and block context accessors, the four `LOG` variants, `SLOAD`/`SSTORE`, and the `CREATE`/`CALL` family (all ecalls).

Complete as a surface, that is, but not as an implementation. Six of the 44 are declared but carry no calling convention: `LOG2`, `LOG3`, `LOG4`, `DELEGATECALL`, `STATICCALL` and `CALLCODE` have no argument registers assigned to them. The reason is register pressure, and it is the one architectural result this draft produces, and the register budget is set out below.

6 Discussion

6.1 Interpreting the Results

Executing EVM semantics on RV32IM is feasible: the counter contract deploys and runs, and the host interface carries the state access. What the draft cannot tell us is what it costs. We took no timings, so we cannot say whether this mapping is faster or slower than an optimised EVM, and two things stand in the way of finding out. Gas metering is not implemented, which rules out any comparison in the units Ethereum actually charges in.

The REVM integration of Section 3.2 is unfinished, which rules out running both interpreters over the same contract in the same harness, the only comparison that would mean anything. Until both exist, the honest position is that this work establishes a mapping, not its cost.

6.2 Developer Experience Revisited

One motivation for exploring alternative VMs is to attract developers familiar with mainstream languages. RISC-V, with its LLVM backend, theoretically enables this. However, this experiment underscores a critical point: writing the *logic* of a smart contract in a language like Rust is only part of the challenge. Developers must still understand and interact with the blockchain's unique context: storage layout, transaction semantics, account models, gas economics, and security considerations. Abstraction libraries (like the conceptual `riscv-evm-utils`) are necessary but represent a new layer of specific APIs developers must learn. Therefore, while the language barrier might be lowered, the *blockchain domain knowledge* barrier remains substantial. It is not "all roses"; significant learning is still required.

6.3 Architectural Implications and Future Directions

This experiment highlights a fundamental architectural crossroads:

- **Option A: High-Fidelity EVM Emulation on RISC-V.** Sticking closely to EVM semantics ensures compatibility but, as observed, can lead to architectural friction and potential performance overhead on RISC-V. Overcoming this might require complex JIT compilers or custom RISC-V extensions specifically designed to accelerate EVM operations, potentially compromising the generality of RISC-V.
- **Option B: Native RISC-V Blockchain Design.** Abandoning strict EVM compatibility allows for a design that leverages RISC-V's strengths natively. This could involve rethinking blockchain storage, addressing schemes, account models, and signature verification to align better with a register-based architecture. While potentially much more performant and cleaner architecturally, this represents a radical departure, sacrificing compatibility and requiring a massive ecosystem transition.

This leads to profound questions about Ethereum's evolution: Are we tethered to the EVM's design philosophy indefinitely? What will Ethereum's execution layer look like in 10–20 years? Is the burden of such fundamental innovation feasible for core developers, or will it inevitably be pushed to Layer 2 solutions, potentially increasing fragmentation? These are difficult, politically charged questions, but essential for the long-term health and scalability of the ecosystem. The desire for alternative VMs stems from the real limitations of the EVM, but the path forward is far from clear.

6.4 Limitations Revisited

The conclusions drawn here are heavily influenced by the preliminary nature of the experiment. The lack of gas metering, optimizations, and the experimental quality of the code mean that performance observations are indicative rather than definitive. A fully optimized, production-ready RISC-V EVM might exhibit different characteristics.

7 Conclusion and Future Work

This paper presented an initial experimental exploration of implementing an EVM-compatible execution environment on the RISC-V ISA. Through a two-phase approach involving a custom interpreter and integration with REVM, I investigated the feasibility and challenges.

My main conclusion is multifaceted: while RISC-V offers a pathway to potentially leverage broader language toolchains for smart contract development, it is not a straightforward solution for replacing or enhancing the EVM. I observed significant architectural friction when mapping complex EVM opcodes to RISC-V, notably register pressure, suggesting potential performance overheads in a direct emulation approach. Furthermore, the complexity of blockchain state interaction remains a significant hurdle for developers, irrespective of the contract language used. This work suggests that the path towards leveraging alternative ISAs like RISC-V for Ethereum-like blockchains forces a difficult choice between maintaining EVM compatibility with potential inherent inefficiencies, or pursuing a radical, incompatible redesign for a potentially more performant "native" architecture.

Future work. Significant work is required to mature this exploration:

1. **Implement gas metering.** Accurately implement EVM gas calculation for RISC-V instruction sequences.
2. **Implement EVM optimizations.** Incorporate features like EIP-2929 (storage warming) [14].
3. **Comprehensive benchmarking.** Conduct rigorous benchmarks against optimized EVM implementations (e.g. geth, REVM) using standard test suites.
4. **Optimization.** Explore JIT compilation techniques or optimized interpretation strategies for the RISC-V execution core.
5. **Bug fixing and stabilization.** Develop a robust, well-tested implementation.
6. **Explore custom extensions.** Investigate the potential benefits and drawbacks of designing custom RISC-V ISA extensions tailored for accelerating EVM operations.
7. **Theoretical analysis.** Further analyze the trade-offs between EVM emulation and a native RISC-V blockchain design.

This research contributes early insights into the complex interplay between ISA design and blockchain execution environments, highlighting both the opportunities and the profound challenges in evolving platforms like Ethereum.

Code availability. The draft implementation discussed throughout this report is at `https://github.com/developeruche/riscv-evm-experiment/tree/main/crates/research-draft`.

8 References

1. G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger.* Ethereum Project Yellow Paper. `https://ethereum.github.io/yellowpaper/paper.pdf`
2. Ethereum Foundation. *Ethereum Execution Layer Specification: specification for the execution layer.* `https://github.com/ethereum/execution-specs`
3. J. Wilcke. *Optimising the Ethereum Virtual Machine.* May 2016. `https://medium.com/@jeff.ethereum/optimising-the-ethereum-virtual-machine-58457e61ca15`

4. A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1.* Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.
5. A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.9.* Technical Report UCB/EECS-2016-129, EECS Department, University of California, Berkeley, July 2016.
6. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. *Bringing the Web up to Speed with WebAssembly.* In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2017. doi:10.1145/3062341.3062363
7. G. Wood. *Polkadot: Vision for a Heterogeneous Multi-Chain Framework.* Draft 1. `https://github.com/polkadot-io/polkadotpaper`
8. EOSIO. *eos-vm: a low-latency, high performance and extensible WebAssembly backend library.* `https://github.com/EOSIO/eos-vm`
9. A. Yakovenko. *Solana: A New Architecture for a High Performance Blockchain, v0.8.13.* `https://solana.com/solana-whitepaper.pdf`
10. Ethereum. *evmjit: the Ethereum EVM JIT.* `https://github.com/ethereum/evmjit`
11. J. Wilcke. *Go Ethereum's JIT-EVM.* Ethereum Foundation Blog, June 2016. `https://blog.ethereum.org/2016/06/02/go-ethereums-jit-evm`
12. 0xDanRobins. *RISC-V for L2 State Transition Computation by Goshen.* Ethereum Research, March 2023. `https://ethresear.ch/t/15054`
13. bluealloy. *REVM: Rust implementation of the Ethereum Virtual Machine.* `https://github.com/bluealloy/revm`
14. V. Buterin and M. Swende. *EIP-2929: Gas cost increases for state access opcodes.* Ethereum Improvement Proposals, September 2020. `https://eips.ethereum.org/EIPS/eip-2929`