Fault Proof VM: MIPS.sol

The MIPS.sol smart contract is an on-chain implementation of a virtual machine (VM) that encompasses the 32-bit, Big-Endian, MIPS III Instruction Set Architecture (ISA). This smart contract is the counterpart to the off-chain MIPSEVM golang implementation of the same ISA. Together, the on-chain and off-chain VM implementations make upCannon , Optimism's Fault Proof Virtual Machine (FPVM). Cannon is a singular instance of a FPVM that can be used as part of the Dispute Game for Optimism's (and Base's) optimistic rollup L2 blockchain. The Dispute Game itself is modular, allowing for any FPVM to be used in a dispute; however, Cannon is currently the only FPVM implemented and thus will be used in all disputes.

Control Flow

In the above video, we can see that the MIPS.sol contract interacts with two other contracts: FaultDisputeGame.sol and PreimageOracle.sol. FaultDisputeGame.sol is the deployed instance of a Fault Dispute Game for an active dispute, and PreimageOracle.sol stores Pre-images.

- Pre-images contain data from both L1 and L2, which includes information such as block headers, transactions, receipts, world state nodes, and more. Pre-images are used as the inputs to the derivation process used to calculate the true L2 state, and subsequently the true L2 state is used to resolve a dispute game.
- A Fault Dispute Game, at a high-level, will effectively determine what L2 state is currently agreed-upon, and move through L2 state until the first disagreed-upon state is found. How the Pre-images are determined and populated into the PreimageOracle.sol contract is out-of-scope for this reference document on the MIPS.sol contract, as that contract only consumes Pre-images that have already been populated by the off-chain Cannon implementation.

The MIPS.sol contract is called by a running instance of a dispute game i.e. by a FaultDisputeGame.sol contract, and is only called once a dispute game reaches a leaf node in the state transition tree that is currently being disputed. A leaf node represents a single MIPS instruction (in the case that we're using Cannon as the FPVM) that can then be run on-chain. Given a Pre-image, which is the previously agreed-upon L2 state up until this instruction, and the instruction state to run in the MIPS.sol contract, the fault dispute game can determine the true post state (or Post-image). This true post state will then be used to determine the outcome of the fault dispute game by comparing the disputed post-state at the leaf node with the post-state proposed by the disputer.

Contract State

The MIPS.sol contract only contains a single immutable variable, which corresponds to the address of the PreimageOracle.sol contract. Otherwise, the contract is stateless, meaning that all state related to playing a MIPS instruction on-chain comes from either the FaultDisputeGame.sol instance, or the PreimageOracle.sol. Having a stateless MIPS.sol contract means that it can be used by any fault dispute game that is using the Cannon FPVM; the MIPS.sol contract does not need to be re-deployed per fault dispute game instance. Subsequently, any fault dispute game that is using the same MIPS.sol contract will also share the same PreimageOracle.sol contract. Note that the PreimageOracle.sol contract is stateful, but how state is stored in the contract and differentiated between different fault dispute game instances is out-of-scope for this document.

While the MIPS.sol contract is stateless, meaning it does not store state in the contract directly, the contract does require up to 3 different types of witness data in order to perform a single MIPS instruction on-chain:

- Packed VM execution state
- Memory proofs
- Pre-images

The Pre-images have already been discussed above, so we will discuss the packed VM execution state and the memory proofs. As a final note on Pre-images during on-chain execution, it is entirely possible that the MIPS.sol contract never runs a MIPS instruction on-chain that requires the contract to read from PreimageOracle.sol. There is no requirement to read from the PreimageOracle.sol during instruction execution, but that doesn't mean the PreimageOracle.sol contract is not being used. The Pre-image information that has been read in previous off-chain instructions leading up to the execution of a single instruction on-chain may still reside in the constructed VM memory. Thus, even when the instruction run on-chain does not explicitly read from PreimageOracle.sol, Pre-image data may still influence the merkle root that represents the VM's memory.

Packed VM Execution State

In order to execute a MIPS instruction on-chain, the MIPS.sol contract needs to know important state information such as the instruction to run, the values in each of the general purpose registers, etc. More specifically, a tightly-packedState struct contains all the relevant information that the MIPS.sol contract needs to know. This struct is passed to the contract from the FaultDisputeGame.sol contract when it invokes thestep function in MIPS.sol (which in-turn executes a single MIPS instruction on-chain). The following information is stored in theState struct. Seedoc reference(opens in a new tab) andcode reference(opens in a new tab) for details:

1. memRoot
2.
    - The merkle root hash of the binary merkle tree that represents the MIPS VM's monolithic 32-bit memory space.
3. preimageKey
4.
    - Key that uniquely identifies the Pre-image data to be read (if applicable) from the PreimageOracle.sol contract.
5. preimageOffset
6.
    - Each Pre-image is 32 bytes long, however the word size for 32-bit MIPS is 4 bytes. Therefore, only 4 bytes from a Pre-image will be read during a read syscall. ThepreimageOffset
7. serves as the offset into the Pre-image being read, so that an instruction can continue reading a Pre-image 4 bytes at a time.
8. pc
9.
    - The program counter, which points to the memory location that contains the MIPS instruction to execute on-chain.
10. nextPC
11.
    - The next program counter, which functions similarly to the program counter except that it points to the next instruction to be executed on-chain. Note that the next instruction may or may not bePC + 8
12. in the event of a branch or jump instruction. Additionally, thenextPC
13. effectively functions as thebranch delay slot(opens in a new tab)
14. for the MIPS ISA.
15. lo
16.
    - Special purpose 32-bit register that stores the low-order 32-bits from certain multiplication instructions (or special move instructions that store into this register).
17. hi
18.
    - Special purpose 32-bit register that stores the high-order 32-bits from certain multiplication instructions (or special move instructions that store into this register)
19. heap
20.
    - Pointer to the most recent memory allocation returned via themmap
21. syscall.
22. exitCode
23. -uint8
24. value that represents UNIX exit status code of the VM.
25. exited
26.
    - Boolean that indicates whether the VM has exited or not.
27. step
28.
    - Counts the total number of instructions that have been executed.
29. registers
30.
    - Array of 32, 32-bit values that represent the general purpose registers for the MIPS ISA.

State Hash

The state hash is thebytes32 value returned to the active Fault Dispute Game upon the completion of a single MIPS instruction in the MIPS.sol contract. The hash is derived by taking thekeccak256 of the above packed VM executionState struct, and then replacing the first byte of the hash with a value that represents the status of the VM. This value is derived from theexitCode andexited values, and can be:

- Valid (0)
- Invalid (1)
- Panic (2) or
- Unfinished (3)

The reason for adding the VM status to the state hash is to communicate to the dispute game whether the VM determined the proposed output root was valid or not. This in turn prevents a user from disputing an output root during a dispute game, but provides the state hash from a cannon trace that actually proves the output root is valid.

Memory Proofs

Using a 32-bit ISA means that the total size of the address space (assuming no virtual address space) is2^32 = 4GiB . Additionally, the MIPS.sol contract is stateless, so it does not store the MIPS memory in contract storage. The primary reason for this is because having to load the entire memory into the MIPS.sol contract in order to execute a single instruction

on-chain is prohibitively expensive. Additionally, the entire memory would need to be loaded per fault proof game, requiring multiple instances of the MIPS.sol contract. Therefore, in order to optimize the amount of data that needs to be provided per on-chain instruction execution while still maintaining integrity over the entire 32-bit address space, Optimism has converted the memory into a binary merkle tree.

The binary merkle tree (or hash tree) used to store the memory of the MIPS VM has leaf values that are 32 bytes and has a fixed depth of 27 levels. This in turn allows the binary merkle tree to span the full 32-bit address space:2^27 * 32 = 2^32 (Seememory proofs(opens in a new tab) for more details). In order to ensure the integrity of the entire address space each time memory is read or written to, one or more memory proofs are provided by the FaultDisputeGame.sol contract each time a MIPS instruction is executed on-chain in MIPS.sol. A memory proof consists of the current leaf value and 27 sibling nodes (28, 32-byte values in total), where the sibling nodes are thekeccak256 hash of its own child nodes. Using the leaf value, its 27 sibling nodes, and the memory address converted to its binary representation as a guide (0 or 1 tells the order to concatenate left and right values), we can calculate a merkle root. This merkle root should be exactly the same as the merkle root stored in the VM execution State struct.

Reading to memory and writing to memory work similarly, both involve calculating the merkle root. In the case of a memory write, MIPS.sol must take care to verify the provided proof for the memory location to write to is correct. Additionally, writing to memory will change the merkle root stored in the VM executionState struct.

State Calculation

While the MIPS.sol contract may only execute a single instruction on-chain, the off-chain Cannon implementation executes all prerequisite MIPS instructions for all state transitions in the disputed L2 state required to reach the disputed instruction that will be executed on-chain. This ensures that the MIPS instruction executed on-chain has the correct VM state and necessary Pre-images stored in the PreimageOracle.sol contract to generate the true post-state that can be used to resolve the dispute game.

Functions

oracle

The external vieworacle (opens in a new tab) function is a getter function that returns the PreimageOracle address cast as a IPreimageOracle interface.

SE

The internal pureSE (opens in a new tab) function performs a sign-extension (SE) on the provideduint32 value given the number of bits that currently represents the value. While the function operates over an unsigned integer, it follows the typical procedure forsign-extension of signed values(opens in a new tab) represented using two's complement.

outputState

The internaloutputState (opens in a new tab) function computes thekeccak256 hash of all values in the VM executionState struct, and then masks the first two bits of the hash with the current status of the VM as derived from theexitCode andexited values. Despite the complexity of the function (due to the use of assembly), the implementation is effectively tightly packing all the variables in theState struct together, then taking thekeccak256 of the packed bytes. A Solidity equivalent implementation can be viewed in theoutputState (opens in a new tab) function located in the foundry test suite.

handleSyscall

The internalhandleSyscall (opens in a new tab) function handles the syscall MIPS instruction. Only a subset of all syscall numbers are supported by the MIPS.sol contract; however, any syscall that is not explicitly supported will return 0s instead of reverting. Most syscalls that are supported partially mimic the behavior specified by the linux manual pages. The two most important syscall numbers that are supported are read and write.

syscall read

When given file descriptor 5 as the target of the read syscall,handleSyscall will call the PreimageOracle.sol contract to read a 32-byte Pre-image at the current location determined by thepreimageKey stored in the VM executionState struct. Once the 32-byte Pre-image has been read, the function will then determine the number of bytes to read (up to 4 bytes) and the location in memory to store the bytes. The number of bytes to read may be any value between 1 and 4, depending on the alignment of the offset in the returned Pre-image and the alignment of the memory position to write the data to.

syscall write

When given the file descriptor 6 as the target of the write syscall,handleSyscall will not call the PreimageOracle.sol contract to write a new Pre-image. Only the off-chain Cannon implementation writes to the PreimageOracle.sol contract. Instead, the

function computes the new value of thepreimageKey given the 4-byte value that would be written to the PreimageOracle.sol contract. Additionally, the function resets thepreimageOffset to 0. It is expected that the off-chain Cannon implementation has already written the data to the PreimageOracle.sol contract at the location of the newly-derivedpreimageKey .

handleBranch

The internalhandleBranch (opens in a new tab) function handles the multiple branch opcodes and conforms to the MIPS specification for the instructions.

handleHiLo

The internalhandleHiLo (opens in a new tab) function handles the multiplication, division, and move opcodes that interact with thehi andlo registers and conforms to the MIPS specification for the instructions.

handleJump

The internalhandleJump (opens in a new tab) function handles J-type opcodes and conforms to the MIPS specification for the instructions.

handleRd

The internalhandleRd (opens in a new tab) function handles storing a value into a specified register. Certain instructions may include a conditional value, which determines whether the value is stored in the register or not.

proofOffset

The internal pureproofOffset (opens in a new tab) function handles calculating the offset in calldata to the start of a memory proof given an index. The calldata is provided to this function via the top-levelstep function call. Looking at the calldata, there are two bytes values passed:stateData andproof . Thebytes stateData value is the packed VM executionState struct, and thebytes proof value represents one or more memory proofs that may be necessary for the on-chain execution of the MIPS instruction. ThestateData and memory proof(s) are encoded in calldata using theSolidity ABI encoding specification(opens in a new tab) . Using this information, we can derive the hardcoded values that are being used in theproofOffset function:

Offset Description Num. Bytes Notes Start of the calldata 4 Function selector 32 Contains the offset to the first dynamic bytes argument akastateData 32 Contains the offset to the second dynamic bytes argument akaproof 32 Contains the length of the first dynamic bytes argumentstateData Start + 100 bytes = Offset to the start of the packedState struct 256 226 bytes for the packed VM executionState struct + 32-byte word alignment for calldata = 256 bytes 32 Contains the length of thebytes calldata proof Start + 388 bytes = Offset to the start of the memory proof(s) 28 * 32 = 896 The first memory proof (required), which is 28, 32-byte values where the first value is the leaf node and the 27 proceeding values are the sibling nodes used to calculate the merkle root 28 * 32 = 896 Within thebytes calldata proof parameter, there can be either 1 or 2 memory proofs

readMem

The internal purereadMem (opens in a new tab) function is a helper function that, given a 32-bit address and an index to a memory proof in calldata, validates the leaf node using the memory proof and then returns the specific 4-byte value to read. Note that a leaf node is 32-bytes so at mostreadMem will only read 4-bytes due to the 32-bit MIPS architecture. The validation given a leaf node and its 27 sibling nodes follows the standard logic for verifying inclusion in a merkle tree. The address is used as the path in order to determine the order for hashing two nodes (or the leaf and a node) together. Once the top of the tree has been reached, and the merkle root calculated, the calculated root is checked against the merkle root stored in the VM executionState struct. Once the leaf node has been verified, the logic will shift to the correct location within the 32-bytes value to read from. Also note thatreadMem will always start at an aligned 4-byte location; therefore it is up to subsequent logic to determine the correct position within a 4-byte word in unaligned memory read or write situations.

writeMem

The internal purewriteMem (opens in a new tab) function is a helper function that, given a 32-bit address, an index to a memory proof in calldata, and a 32-bit value to write, calculates the new merkle root of the VM executionState struct. Calculating the new merkle root follows the same logic as in thereadMem function, except that the new value will be stored in the State struct. Note thatwriteMem does not verify the proof used to calculate the new merkle root.

⚠ It iscritically important that the memory proof being used is verified beforehand by calling thereadMem function. Also note thatwriteMem , similar toreadMem , only works over 4-byte aligned words. Therefore, it is up to the logic that callswriteMem to generate a value to write such that if an unaligned memory write occurs, the value already reflects that.

step

The publicstep (opens in a new tab) function is the top-level call that executes a single MIPS instruction. This function will be

called by an active dispute game in order to determine the true post state given a pre state and a MIPS instruction to run. At a high-level, the function performs the following steps:

1. Verifies and unpacks the stateData variable into the VM executionState
2. struct (in memory).
3. Reads the instruction located at the program counter (pc
4.).
5. Interprets and executes the MIPS instruction according to the MIPS specification.
6. Writes results to registers or memory (if applicable) and updates the VM executionState
7. struct accordingly.

execute

The internal pureexecute (opens in a new tab) function handles the execution of MIPS instructions that are not handled by other functions. Theexecute function primarily handles R-type instructions according to the MIPS specification, however other instructions will pass through this function. Instructions handled by other functions will simply return. Invalid or unsupported instructions will cause theexecute function to revert.

Common Bitwise Operation Use Cases

1. Isolating certain bits from a number can be done using the & operator (and(x,y) in Yul), this is also known as generating a bitmask.
2. Combining bits from two numbers together can be done using the | operator (or(x, y) in Yul).
3. Modulo arithmetic can be expressed using the following bitwise operation:x % y = x & (y - 1), ex. x % 4 = x & 3.
4. Note this is only equivalent for unsigned integers.
5. Multiplication using a value with a base of 2 can be expressed using the following bitwise operation:x * y = x << z, where y = 2^z
6. Ex.x * 8 = x << 3, where 8 = 2^3

Table Of Supported MIPS Instructions

Instruction Name Opcode Num. Funct Num. Other Num. SYSCALL (System Call) 0x00 0x0C - J (Jump) 0x02 - - JR (Jump Register) 0x00 0x08 - JAL (Jump and Link) 0x03 - - JALR (Jump and Link Register) 0x00 0x09 - BEQ (Branch on Equal) 0x04 - - BNE (Branch on Not Equal) 0x05 - - BLEZ (Branch on Less Than or Equal to Zero) 0x06 - - BGTZ (Branch on Greater Than Zero) 0x07 - - BLTZ (Branch on Less Than Zero) 0x01 - 0x00 BGEZ (Branch on Greater Than or Equal to Zero) 0x01 - 0x01 MOVZ (Move Conditional on Zero) 0x00 0x0A - MOVN (Move Conditional on Not Zero) 0x00 0x0B - MFHI (Move from HI) 0x00 0x10 - MTHI (Move to HI) 0x00 0x11 - MFLO (Move from LO) 0x00 0x12 - MTLO (Move to LO) 0x00 0x13 - MULT (Multiply Word) 0x00 0x18 - MULTU (Multiply Unsigned Word) 0x00 0x19 - DIV (Divide Word) 0x00 0x1A - DIVU (Divide Unsigned Word) 0x00 0x1B - ADD (Add Word) 0x00 0x20 - ADDU (Add Unsigned Word) 0x00 0x21 - ADDI (Add Immediate Word) 0x08 - - ADDIU (Add Immediate Unsigned Word) 0x09 - - SUB (Subtract Word) 0x00 0x22 - SUBU (Subtract Unsigned Word) 0x00 0x23 - SLT (Set on Less Than) 0x00 0x2A - SLTU (Set on Less Than Unsigned) 0x00 0x2B - SLTI (Set on Less Than Immediate) 0x0A - - SLTIU (Set on Less Than Immediate Unsigned) 0x0B - - AND (And) 0x00 0x24 - ANDI (And Immediate) 0x0C - - OR (Or) 0x00 0x25 - ORI (Or Immediate) 0x0D - - XOR (Exclusive Or) 0x00 0x26 - XORI (Exclusive Or Immediate) 0x0E - - NOR (Nor) 0x00 0x27 - SLL (Shift Word Left Logical) 0x00 0x00 - SRL (Shift Word Right Logical) 0x00 0x02 - SRA (Shift Word Right Arithmetic) 0x00 0x03 - SLLV (Shift Word Left Logical Variable) 0x00 0x04 - SRLV (Shift Word Right Logical Variable) 0x00 0x06 - SRAV (Shift Word Right Arithmetic Variable) 0x00 0x07 - MUL (Multiply Word to Register) 0x1C 0x02 - CLZ (Count Leading Zeros in Word) 0x1C 0x20 - CLO (Count Leading Ones in Word) 0x1C 0x21 - LUI (Load Upper Immediate) 0x0F - - LL (Load Linked Word) 0x30 - - LB (Load Byte) 0x20 - - LBU (Load Byte Unsigned) 0x24 - - LH (Load Halfword) 0x21 - - LHU (Load Halfword Unsigned) 0x25 - - LW (Load Word) 0x23 - - LWL (Load Word Left) 0x22 - - LWR (Load Word Right) 0x26 - - SB (Store Byte) 0x28 - - SH (Store Halfword) 0x29 - - SW (Store Word) 0x2B - - SWL (Store Word Left) 0x2A - - SWR (Store Word Right) 0x2E - - SC (Store Conditional Word) 0x38 - - SYNC (Synchronize Shared Memory) 0x00 0x0F -

Further Reading

- Cannon Overview(opens in a new tab)
- Cannon FPVM Specification(opens in a new tab)
- MIPS Reference Sheet(opens in a new tab)
- MIPS IV ISA Specification(opens in a new tab)
- MIPS32 Architecture For Programmers Volume II (for SPECIAL2 opcodes)(opens in a new tab)
- MIPS Assembly Wiki Book(opens in a new tab)
- MIPS syscall numbers(opens in a new tab)
- Yul Instructions(opens in a new tab)
- Solidity ABI Encoding Specification(opens in a new tab)