zkMIPS: a high-level specification

ZKM Research Team

Version 0.95

Abstract

In September of 2023, ZKM Research started developing a zkVM for the MIPS instruction set and processor architecture. This paper presents a high-level description of the system and software architecture, including the rationale for the most important design decisions. zkMIPS code is available in https://github.com/zkMIPS/zkm. Readers can give us feedback on this paper in https://discord.com/channels/1125877344972849232/1246097911239016509 and ask questions in https://discord.com/channels/1125877344972849232/1246864756250251346.

1 Introduction

A zkVM is a primitive which permits one party $\mathcal{A}$ to outsource a computation to another (computationally more powerful) party $\mathcal{B}$ in a verifiable manner. The output computed by $\mathcal{B}$ comes with a proof (or seal) that the result was computed correctly. In the case of zkMIPS, the program to be executed is specified using the MIPS instruction set, possibly after compilation from some other language.

A valid computation can be interpreted as a table whose columns represent the list of CPU variables defining the processor's overall state, and whose rows represent each step of the computation process. This table is known as the **trace record**, or simply as the trace. Moving from one trace row to the next represents a state transition, which must correspond to a valid computation step. In this setup, verifying the correctness of a computation is equivalent to verifying the transition between each pair of subsequent trace rows. One way to make this verification more efficient for $\mathcal{A}$ is to encode the entire trace as a mathematical object, usually a polynomial, in a process called **arithmetization**.

Another important building block for zkVMs is the routine run between $\mathcal{A}$ and $\mathcal{B}$ to help the verification of the desired polynomial properties. The polynomial resulting from the arithmetization process is usually of the size (degree) of the program being proved, meaning verifying this polynomial naively is roughly the same complexity of verifying the program itself. To optimize this process, the parties involved engage in an **Interactive Oracle Proof** (IOP) where $\mathcal{B}$ sends some commitment data to $\mathcal{A}$ and $\mathcal{A}$ queries values that facilitate the verification. Using the data provided by $\mathcal{B}$ beforehand, these queried values can be verified to convince $\mathcal{A}$ that $\mathcal{B}$ did not cheat along the way.

Combining arithmetization and IOP can reduce the verification time from linear to poly-logarithmic on the size of the program, in which case we say the resulting proof is **succinct**. We stress that the zk in zkVM means Zero-Knowledge but often refers to succinctness, not to privacy. In other words, a zkVM might not be private and still be considered a zkVM because it produces succinct proofs.

The range of IOPs used for succinct verification is somewhat small. On the other hand, the range of arithmetization techniques is large and choosing among them is not easy. In the first place, the underlying mathematical theory is highly abstract, not easy to comprehend, and often there is no consensus terminology among works in the area. This is aggravated by the fact that what is described in a paper is not necessarily what has been implemented, while these digressions often are not well documented. One either has access to high-level theoretical papers or low-level access to existing open-source libraries that contain very little documentation on an intermediate level. Of course, these low-level libraries are highly relevant in order to reduce development time while obtaining good performance.

The main purpose of this paper is to describe this confusing landscape in detail, and discuss the choices made by ZKM. This document covers the theoretical aspects of the arithmetization models and IOPs used in zkMIPS, as well as the design choices made during the development process. Where theory and practice do not align, we opted to privilege the way the protocol is implemented and often ignore details that are better explained in the original papers. In other words, this document aims to enlighten developers by plugging the gap between papers and code, though it may not be satisfactory for hardcore mathematicians. We provide this kind of information as guidance and we believe that it will be intrinsically useful for the community, independent of zkMIPS.

Comparison to other works

The past 2-3 years have seen the rise of numerous projects with similar objectives. zkEVMs are trying to implement verifiable computing for Ethereum's EVM bytecode, while Risc0, Jolt, and SP1 target the development of a zkVM for the RISC-V instruction set. We chose MIPS for a variety of reasons. MIPS has been around for almost 4 decades, and has a strong presence in industry with many legacy applications. As a result, anything compiles to MIPS and MIPS compiles to anything. MIPS is also extensively used in IoT devices. In addition, the MIPS-R3000 instruction set has not changed over time so is very stable, unlike EVM bytecode (frequent minor opcode changes) or RISC-V (allows custom instructions). And finally, patent concerns do not apply in the setting in which we use MIPS.

Structure of the paper

Section 2 describes the MIPS architecture, which includes certain assumptions for the sake of simplification. Section 3 gives a general description of arguments of knowledge, discussing STARK, PLONK and Plonky2 in particular. In Section 4, the content of the previous two sections come together, resulting in an explanation of the zkMIPS protocol.

2 MIPS architecture

The Microprocessor without Interlocked Pipelined Stages (MIPS) is a well-known and widely adopted class of 32/64-bit computer architectures developed by MIPS Computer Systems. The 32-bit specification is a big-endian, register-based architecture with 32 General Purpose Registers (GPRs) of 32-bit, allocated according to Table 1, and a 4GB memory addressed by 2^{32} words of 4 bytes each.

Variable	Name	Description	Size
R0	Zero	Always contains 0	u32
R1	AT	Assembler temporary	u32
R2..R3	V0..V1	Return values	u32
R4..R7	A0..A3	Parameters values	u32
R8..R15	T0..T7	Temporary values	u32
R16..R23	S0..S7	Saved values	u32
R24..R25	T8..T9	Temporary values	u32
R26..R27	K0..K1	Reserved for kernel	u32
R28	GP	Global pointer	u32
R29	SP	Stack pointer	u32
R30	S8	Saved values	u32
R31	RA	Return address	u32

Table 1: General Purpose Registers

In general, MIPS programs operate on a 32-bit cycle counter, a 32-bit program counter representing the memory address of the current instruction (optionally, one can include a next value for this counter), 32-bit high and low results for 32-bit multiplication and division operations, an exit boolean, and an 8-bit exit flag.

In the specific case of zkMIPS, it also contains a 32-bit succinct representation of the memory state defined as the Keccak-based Merkle root whose leaves correspond to 4KB memory pages. An extensive description of the variables contained in a zkMIPS CPU state is given in Table 2. This memory state representation is necessary because the memory corresponds to the program file itself: instead of loading the program into memory, zkMIPS loads memory into a special region of the program file. Furthermore, all variables included in intermediary states of the zkMIPS VM (i.e. counters, high/low results, and exit variables) are recorded in memory after each cycle. This approach allows steps of the execution of this VM to be described more easily since the input (program) and output (valid final CPU state) for the proof generation are seen as states of the same object.

Variable	Name	Description	Size
Cycle	Cycle counter	Counts how many instructions have been processed	u32
PC	Program counter	Points to the address of the current instruction	u32
NextPC	Next Program counter	Points 8 bytes past the address of the current instruction address	u32
HI	High	Multiplication/division results	u32
LO	Low		u32
Exited	Exit flag	Indicates program finalization	bool
ExitCode	Exit code	Indicates program finalization status	u8
R0	Zero	Always contains 0	u32
...	...	...	...
R31	RA	Return address	u32
MemRoot	Memory state root	Merkle tree of current memory state	u8[32]

Table 2: CPU state

In the remainder of this section we describe how a file containing a MIPS program is organized (Section 2.1) and which instructions are supported by the MIPS CPU (Section 2.2). As just explained, the concepts of program and memory are blurred in the context of zkMIPS proof generation since the program file is used to represent the memory. Therefore, to explain how zkMIPS memory is organized and how it is used as the memory, we focus on the zkMIPS specification of a MIPS program file.

2.1 Program

Name	Description	Points to
ELF header	Program specification	Program header
		Section header
Program header	List the set of segments used at runtime	
.text		
.data		
Section header	Lists the set of program sections	.text
		.data

Table 3: Program/memory files

2.2 Instructions

zkMIPS supports a subset of 61 MIPS instructions of 32 bits each. The first or the last 6 bits of an instruction are used for identification. The first 6 bits of an instruction are called the *opcode* and, when an instruction has opcode `000000`, its last 6 bits are called *funct*. The values contained in the opcode and funct fields define the syntax and the semantics of each instructions. MIPS instructions have four possible syntax formats, namely the R, I, J and Special formats. Instructions that have functs are usually of the R or the Special formats, and instructions that do not are of the I or the J formats.

The 20 bits between opcode and funct of R format instructions are divided intoto four 5-bit fields, with the first two encoding input registers, the third encoding an output register, and the last encoding an extra input for some instructions. The last 26 bits of I format instructions are divided intoto two 5-bit fields encoding two inputs or one input and one output registers, and one 16-bit field encoding a half-word input. The last 26 bits of J format instructions encode one single input. There is also one special format for instructions that invoke system events, this one with the last 6 bits encoding a funct field and the middle 20 bits encoding one single input. This scheme is illustrated in Table 4.

Type	6 bits [31..26]	5 bits [25..21]	5 bits [20..16]	5 bits [15..11]	5 bits [10..6]	6 bits [5..0]
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	imm(ediate)		
J	opcode	addr(ess)				
Special	opcode	code				funct

Table 4: Syntax of MIPS instruction

The 61 MIPS instructions supported by zkMIPS belong to 7 original MIPS instruction categories, namely *arithmetic*, *branch and jump*, *load/store and memory*, *logic*, *move*, *shift* and *trap*, as described in Tables 5 to 11 and listed in Table 12. These categories are important to map how instructions will be modeled in our zkVM, as explained in Section 4.3.

Instructions from the arithmetic category perform arithmetic operations over 32-bit values stored in input registers (R format instructions) or signed 16-bit inputs (I format instructions). These 16-bit inputs are extended to 32-bit inputs according to their sign; thus the `sext` function from Table 5 performs a *signed extension*. The same happens for signed 16-bit inputs from branch and jump instructions and for some from load/store and memory instructions (see Tables 6 and 7, respectively). Unsigned 16-bit inputs from remaining load/store and memory instructions are handled differently; they can be simply extended with zeros, so the `zext` function from Table 7 performs a *zero extension*.

Instructions from the branch and jump category modify the PC according to signed 16-bit inputs (I format branch instructions), unsigned 26-bit inputs (J format jump instructions) or 32-bit values stored in registers (R format jump instructions). The signed 16-bit inputs correspond to instruction indexes relative to the current PC, while the unsigned 26-bit inputs and 32-bit values stored in registers correspond to absolute instruction indexes. Since memory is byte-aligned and the PC corresponds to the address of the current instruction, these instruction indexes must be converted to the instruction addresses. To do so, signed 16-bit instruction indexes are first shifted 2 bits to the left, then extended and added to the current PC to address memory positions within the 2^{17} bytes (128kB) memory range from the current PC. Analogously, unsigned 26-bit instruction indexes are first padded with zeros to encode byte-aligned addresses instead of word-aligned, and then appended to the first 6 bytes from the current PC to address memory positions within the 2^{18} bytes (256MB) memory region the current PC points to.

Instructions from the load/store category perform memory operations based on input or output registers, base memory positions stored in registers and 16-bit memory offsets (I format instructions). Values loaded from memory can be signed or zero extended, depending on the opcode. The memory position from which bytes, half-words or words are loaded is given as an unrestricted byte index, meaning they can be loaded starting from byte indexes inside memory words. In addition to that, left or right halves of registers can be loaded or stored while maintaining the other half unchanged, and there are instructions to lock memory position atomically. For a better comprehension on how load/store and memory instructions are implemented, we refer to [10].

Instructions from the logic category perform usual logic operations over 32-bit values stored in

#	Name	Syntax						Semantics
1	ADD	000000	rs	rt	rd	00000	100000	$\texttt{rd} = \texttt{rs} + \texttt{rt}$
2	ADDI	001000	rs	rt	imm			$\texttt{rt} = \texttt{rs} + \texttt{sext}(\texttt{imm})$
3	ADDIU	001001	rs	rt	imm			$\texttt{rt} = \texttt{rs} + \texttt{sext}(\texttt{imm})$
4	ADDU	000000	rs	rt	rd	00000	100001	$\texttt{rd} = \texttt{rs} + \texttt{rt}$
5	CLO	011100	rs	rt	rd	00000	100001	$\texttt{rd} = \texttt{count\_leading\_ones}(\texttt{rs})$
6	CLZ	011100	rs	rt	rd	00000	100000	$\texttt{rd} = \texttt{count\_leading\_zeros}(\texttt{rs})$
7	DIV	000000	rs	rt	00000	00000	011010	$(\texttt{hi}, \texttt{lo}) = \texttt{rs}/\texttt{rt}$
8	DIVU	000000	rs	rt	00000	00000	011011	$(\texttt{hi}, \texttt{lo}) = \texttt{rs}/\texttt{rt}$
9	MUL	011100	rs	rt	rd	00000	000010	$\texttt{rd} = \texttt{rs} \times \texttt{rt}$
10	MULT	000000	rs	rt	00000	00000	011000	$(\texttt{hi}, \texttt{lo}) = \texttt{rs} \times \texttt{rt}$
11	MULTU	000000	rs	rt	00000	00000	011001	$(\texttt{hi}, \texttt{lo}) = \texttt{rs} \times \texttt{rt}$
12	SLT	000000	rs	rt	rd	00000	101010	$\texttt{rd} = \texttt{rs} < \texttt{rt}$
13	SLTI	001010	rs	rt	imm			$\texttt{rt} = \texttt{rs} < \texttt{sext}(\texttt{imm})$
14	SLTIU	001011	rs	rt	imm			$\texttt{rt} = \texttt{rs} < \texttt{sext}(\texttt{imm})$
15	SLTU	000000	rs	rt	rd	00000	101011	$\texttt{rd} = \texttt{rs} < \texttt{rt}$
16	SUB	000000	rs	rt	rd	00000	100010	$\texttt{rd} = \texttt{rs} - \texttt{rt}$
17	SUBU	000000	rs	rt	rd	00000	100011	$\texttt{rd} = \texttt{rs} - \texttt{rt}$

Table 5: Syntax and semantics of arithmetic instructions

#	Name	Syntax						Semantics
18	BEQ	000100	rs	rt	imm			$\texttt{rs} = \texttt{rt} \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
19	BGEZ	000001	rs	00001	imm			$\texttt{rs} \geq 0 \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
20	BGTZ	000111	rs	00000	imm			$\texttt{rs} > 0 \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
21	BLEZ	000110	rs	00000	imm			$\texttt{rs} \leq 0 \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
22	BLTZ	000001	rs	00000	imm			$\texttt{rs} < 0 \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
23	BNE	000101	rs	rt	imm			$\texttt{rs} \neq \texttt{rt} \Rightarrow \texttt{PC} = \texttt{PC} + \texttt{sext}(\texttt{imm} << 2)$
24	J	000010	addr					$\texttt{PC} = \texttt{PC}[31..28] \| \texttt{addr} \| 00$
25	JAL	000011	addr					$\texttt{r31} = \texttt{PC} + 8$ $\texttt{PC} = \texttt{PC}[31..28] \| \texttt{addr} \| 00$
26	JALR	000000	rs	00000	rd	shamt	001001	$\texttt{rd} = \texttt{PC} + 8$ $\texttt{PC} = \texttt{rs}$
27	JR	000000	rs	00000	00000	shamt	001000	$\texttt{PC} = \texttt{rs}$

Table 6: Syntax and semantics of branch and jump instructions

input registers (R format instructions) or 16-bit inputs (I format instructions). In addition to these operations, the logic category also features an instruction to load 16-bit inputs (I format instructions) to the upper-most half of a GPR. Instructions from the move category copy 32-bit values between GPRs (R format instructions), or between general-purpose and high/low registers. Instructions from the shift category perform arithmetic or logical shifts over GPRs, according to 5-bit values stored in input registers or shamt fields (R format instructions). Finally, the syscall instruction from the trap category invokes special functions according to 20-bit inputs (special format instructions). For details we refer to [10].

#	Name	Syntax				Semantics
28	LB	100000	rs	rt	imm	$rt = sext(mem[rs + imm])$
29	LBU	100100	rs	rt	imm	$rt = zext(mem[rs + imm])$
30	LH	100001	rs	rt	imm	$rt = sext(mem[rs + imm : rs + imm + 1])$
31	LHU	100101	rs	rt	imm	$rt = zext(mem[rs + imm : rs + imm + 1])$
32	LL	110000	rs	rt	imm	$rt = mem[rs + imm : rs + imm + 3]$
33	LW	100011	rs	rt	imm	$rt = mem[rs + imm : rs + imm + 3]$
34	LWL	100010	rs	rt	imm	$rt[31 : 16] = mem[rs + imm : rs + imm + 1]$
35	LWR	100110	rs	rt	imm	$rt[15 : 0] = mem[rs + imm - 1 : rs + imm]$
36	SB	101000	rs	rt	imm	$mem[rs + imm] = rt[7 : 0]$
37	SC	111000	rs	rt	imm	$rt = 1 \Rightarrow mem[rs + imm : rs + imm + 3] = rt$ $rt \neq 1 \Rightarrow rt = 0$
38	SH	101001	rs	rt	imm	$mem[rs + imm : rs + imm + 1] = rt[15 : 0]$
39	SW	101011	rs	rt	imm	$mem[rs + imm : rs + imm + 3] = rt$
40	SWL	101010	rs	rt	imm	$mem[rs + imm : rs + imm + 1] = rt[31 : 16]$
41	SWR	101110	rs	rt	imm	$mem[rs + imm - 1 : rs + imm] = rt[15 : 0]$

Table 7: Syntax and semantics of load/store and memory instructions

#	Name	Syntax						Semantics
42	AND	000000	rs	rt	rd	00000	100100	$rd = rs \wedge rt$
43	LUI	001111	00000	rt	imm			$rt = imm << 16$
44	NOR	000000	rs	rt	rd	00000	100111	$rd = !(rs \vee rt)$
45	OR	000000	rs	rt	rd	00000	100101	$rd = rs \vee rt$
46	ORI	001101	rs	rt	imm			$rt = rs \vee zext(imm)$
47	XOR	000000	rs	rt	rd	00000	100110	$rd = rs \oplus rt$
48	XORI	001110	rs	rt	imm			$rt = rs \oplus zext(imm)$

Table 8: Syntax and semantics of logic instructions

#	Name	Syntax						Semantics
49	MFHI	000000	00000	00000	rd	00000	010000	$rd = hi$
50	MFLO	000000	00000	00000	rd	00000	010010	$rd = lo$
51	MOVN	000000	rs	rt	rd	00000	001011	$rt \neq 0 \Rightarrow rd = rs$
52	MOVZ	000000	rs	rt	rd	00000	001010	$rt = 0 \Rightarrow rd = rs$
53	MTHI	000000	rs	00000	00000	00000	010001	$hi = rs$
54	MTLO	000000	rs	00000	00000	00000	010011	$lo = rs$

Table 9: Syntax and semantics of move instructions

#	Name	Syntax						Semantics
55	SLL	000000	00000	rt	rd	shamt	000000	$rd = rt << shamt$
56	SLLV	000000	rs	rt	rd	00000	000100	$rd = rt << rs[4 : 0]$
57	SRA	000000	00000	rt	rd	shamt	000011	$rd = rt >> shamt$
58	SRAV	000000	rs	rt	rd	00000	000111	$rd = rt >> rs[4 : 0]$
59	SRL	000000	00000	rt	rd	shamt	000010	$rd = rt >> shamt$
60	SRLV	000000	rs	rt	rd	00000	000110	$rd = rt >> rs[4 : 0]$

Table 10: Syntax and semantics of shift instructions

#	Name	Syntax			Semantics
61	SYSCALL	000000	code	001100	syscall

Table 11: Syntax and semantics of trap instructions

#	Name	Description
1	ADD	Add
2	ADDI	Add Immediate Word
3	ADDIU	Add Immediate Unsigned Word
4	ADDU	Add Unsigned Word
5	CLO	Count Leading Ones in Word
6	CLZ	Count Leading Zeros in Word
7	DIV	Divide Word
8	DIVU	Divide Unsigned Word
9	MUL	Multiply Word to GPR
10	MULT	Multiply Word
11	MULTU	Multiply Unsigned Word
12	SLT	Set on Less Than
13	SLTU	Set on Less Than Unsigned
14	SLTI	Set on Less Than Immediate
15	SLTIU	Set on Less Than Immediate Unsigned
16	SUB	Subtract Word
17	SUBU	Subtract Unsigned Word

(a) Arithmetic instructions

#	Name	Description
18	BGEZ	Branch on Greater Than or Equal to Zero
19	BLTZ	Branch on Less Than Zero
20	BEQ	Branch on Equal
21	BNE	Branch on Not Equal
22	BLEZ	Branch on Less Than or Equal to Zero
23	BGTZ	Branch on Greater Than Zero
24	JR	Jump Register
25	JALR	Jump and Link Register
26	J	Jump
27	JAL	Jump and Link

(b) Branch and jump instructions

#	Name	Description
28	LB	Load Byte
29	LBU	Load Byte Unsigned
30	LH	Load Halfword
31	LHU	Load Halfword Unsigned
32	LL	Load Linked Word
33	LW	Load Word Left
34	LWL	Load Word Left
35	LWR	Load Word Right
36	SB	Store Byte
37	SC	Store Conditional Word
38	SH	Store Halfword
39	SW	Store Word
40	SWL	Store Word Left
41	SWR	Store Word Right

(c) Load/store and memory instructions

#	Name	Description
42	AND	And
43	LUI	Load Upper Immediate
44	NOR	Not Or
45	OR	Or
46	ORI	Or Immediate
47	XOR	Exclusive Or
48	XORI	Exclusive Or Immediate

(d) Logic instructions

#	Name	Description
49	MFHI	Move From HI Register
50	MFLO	Move From LO Register
51	MOVN	Move Conditional on Not Zero
52	MOVZ	Move Conditional on Zero
53	MTHI	Move To HI Register
54	MTLO	Move To LO Register

(e) Move instructions

#	Name	Description
55	SLL	Shift Word Left Logical
56	SLLV	Shift Word Left Logical Variable
57	SRA	Shift Word Right Arithmetic
58	SRAV	Shift Word Right Arithmetic Variable
59	SRL	Shift Word Right Logical
60	SRLV	Shift Word Right Logical Variable

(f) Shift instructions

#	Name	Description
61	SYSCALL	System Call

(g) Trap instructions

Table 12: Instructions categories

3 Arguments of Knowledge

Cryptographic proof systems can model algorithms as 2-party protocols in contexts where one of the parties can run the underlying algorithm faster than the other. Usually, the party that can run this algorithm more easily either has access to a more efficient computer or it possesses some secret information; in the latter case we say the protocol is a **proof-of-knowledge**. In both cases, cryptographic proofs allow this privileged party to convince the other one that the underlying algorithm was executed correctly (when run on its efficient computer or when fed with its secret information). For this reason, we denominate these parties the **Prover** and the **Verifier**, respectively.

These proofs should be designed to optimize the interests of both the Prover and the Verifier. On the Prover side, this means the proof system execution succeeds with high probability when the Prover is faithfully following the protocol, i.e. it can execute the underlying algorithm successfully and it follows the protocol description correctly. This property is called **completeness**. On the Verifier side, this means the proof system execution fails with high probability when the Prover is not faithfully engaged in the protocol, i.e. it cannot execute the underlying algorithm successfully or it does not follow the protocol description. This property is called **soundness**.

Sometimes these proofs are required to run in time polynomial in the logarithm of the input to that algorithm. In other words, as the input size N increases, Prover and Verifier times increase polynomially on $\log(N)$. When the communication complexity of a proof is also polynomial on $\log(N)$, we say the proof is**succinct**. One additional property modern proof systems may present to enable particular proof designs is soundness holding exclusively against polynomial-time Provers. When a proof system presents this property, we call it an **argument**. When the Prover possesses some secret information, cryptographic proofs may also keep this information secure by revealing no useful information about the secret object besides its usefulness to compute the underlying algorithm. This property is called **Zero-Knowledge** (ZK).

zkMIPS is divided into a hierarchy of proofs distributed in three mandatory layers and one optional layer. Each of these layers utilize different succinct and ZK arguments-of-knowledge, namely:

1. **Scalable and Transparent Argument-of-Knowledge**[2] (STARK) is used in the layer that generates the lowest proofs in this hierarchy;

2. **Multivariate Look-ups based on Logarithmic Derivatives**[9] (LogUp), implemented using STARK, is used in the layer that generates the middle proofs in this hierarchy;

3. **Permutations over Lagrange-bases for Oecumenical Non-interactive Arguments-of-Knowledge**[6] (PLONK) is used in the layer that generates the highest proofs in this hierarchy;

4. **Groth16**[8] is used in the optional proving layer when a zkMIPS proof must be verified on-chain;

Figure 1 illustrates how these different proof systems are used in zkMIPS proof generation.

To model algorithm executions, STARK, PLONK and Groth16 proof systems use finite automatons. The state of such automatons is composed of a list of internal variables. In this setup, a computation is defined by a sequence of states starting from some well-defined initial state and finishing in some valid final state, such that each pair of subsequent states represents a valid state transition. Those proof systems operate over a table with rows representing states and columns representing internal variables, which is precisely the **trace record** mentioned in Section 1. In the context of zkMIPS, rows represent states from the program execution and columns indirectly represent CPU variables described in Table 2 (the exact representation of these variables inside the trace record will be explained in Section 4.2).

Valid state transitions are represented as polynomials over a finite field $\mathbb{F}$ of prime order p. These polynomials are called **constraint polynomials**. Valid states are defined in terms of polynomials over the same field, but for soundness reasons their evaluation is restricted to an appropriate subset of this field. These polynomials are called **witness polynomials**. During the proving procedure, entries for constraint polynomials are picked from witness polynomial evaluations.

The exact representation of constraint and witness polynomials depends on each proof system. Sections 3.2, 3.3 and 3.5 elaborate on how these polynomials are defined and how their correctness is evaluated in the STARK, PLONK and LogUp proof systems, respectively. For didactic purposes, the notation for constraint and witness polynomials in these sections is the same: the i-th witness polynomial is denoted by W_i and the j-th constraint polynomial is denoted by C_j. In general, for

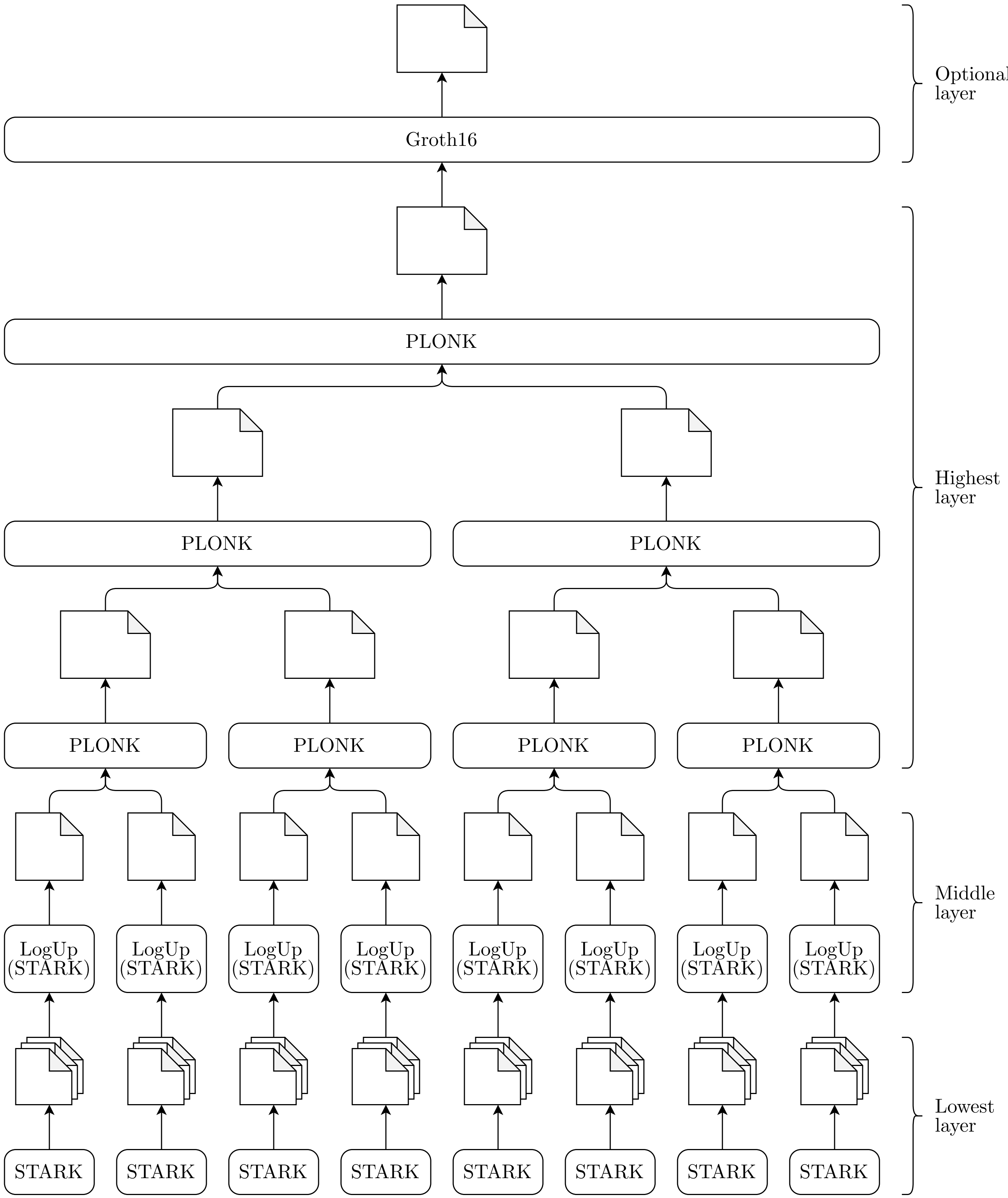

Figure 1: Proof systems hierarchy

a trace record with U rows and V columns, the numbers of constraint and witness polynomials in a proof that verifies that trace record are $O(U)$ and $O(V)$, respectively.

The protocols from Sections 3.2 and 3.3 model general computations, while the LogUp proof system, explained in Section 3.5, serves a completely different purpose, but is still a major piece in zkMIPS architecture. Section 3.4 describes the proof system implemented by the library we chose to use in our codebase, Plonky2[12]. This library implements a variation of STARK that borrows a few pieces from PLONK and a variation of PLONK that borrows a few pieces from STARK. For this reason, some simplifications are made during the description of these protocols in Sections 3.2 and 3.3 in order to keep the text as succinct as possible, yet helpful for the understanding of the protocol Plonky2.

The following section elaborates on the algebraic definitions and properties that enable the proof systems explained in these sections. Readers not interested in algebraic definitions can skip these paragraphs, as they are not essential for a high-level understanding of those proof systems.

3.1 Preliminaries

Constraint and witness polynomial representations Let $\mathbb{F}^*$ be the multiplicative subgroup of $\mathbb{F}$ of order $p-1$, and let k be the largest positive integer such that $2^k \mid p-1$ (the security properties from the proof systems explained in the following sections require p to be chosen such that $k \geq 20$). Also, let G be a proper subgroup of $\mathbb{F}^*$ of order $N = 2^n$ such that $n < k$, and let g be a generator of G chosen at random. Constraint polynomials are defined as polynomial functions $\mathbb{F}^* \to \mathbb{F}^*$, while witness polynomials are defined as polynomial functions $G \to \mathbb{F}^*$. Now let $\vec{w}_i = [w_{i,1}, \ldots, w_{i,U}]$ represent the i-th trace column, meaning each $w_{i,j}$ represents the value of the i-th variable on the j-th state. Then, the pre-encoded witness polynomials are defined as described in Equations 1 and 2.

$$w_i(g^{j-1}) = w_{i,j} \ \forall\ i \in [1, V],\ j \in [1, U] \tag{1}$$

$$w_i(g^{j-1}) = 0 \ \forall\ i \in [1, V],\ j \in (U, N] \tag{2}$$

Low Degree Extension Let G' be a proper subgroup of $\mathbb{F}^*$ of order $M = 2^m$ such that $n < m < k$ (implying $G \subsetneq G' \subsetneq \mathbb{F}^*$), let γ be a generator of $\mathbb{F}^*$ chosen at random, and let H be the coset of G' such that $H = \{\gamma \cdot g^{j-1} \mid j \in [1, M]\}$. A polynomial function $H \to \mathbb{F}^*$ that has the same coefficient representation as a polynomial function $G \to \mathbb{F}^*$ is called its **Low Degree Extension** (LDE). The (fully-encoded) witness polynomial W_i is the LDE of the pre-encoded witness polynomial w_i, as described in Equation 3. The coset H is called *evaluation domain* of this LDE, and the ratio $\beta = M/N = 2^{m-n}$ is called its *blow-up factor* (this is the ratio used to pick some H for a given G).

$$W_i(X) = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1} \ \forall\ X \in H \text{ such that}$$
$$w_i(Y) = a_0 + a_1 \cdot Y + \cdots + a_{N-1} \cdot Y^{N-1} \ \forall\ Y \in G \tag{3}$$

Fast Fourier Transform The compilation of vector representation of the pre-encoded witness polynomials to their coefficient representation is an instance of the **Inverse Discrete Fourier Transform** (IDFT). The compilation of coefficient representation of the pre-encoded witness polynomials to the vector representation of fully-encoded witness polynomials is an instance of the **Discrete Fourier Transform** (DFT). These steps are described in Equations 4 and 5. To compute the DFT and the IDFT, the proof systems explained in the following sections use the well-known **Fast Fourier Transform** (FFT) algorithm, which runs in time $O(N \cdot \mathsf{log}\,(N))$ and dominates their proof generation time complexity. For details about the Fast Fourier Transform, see for instance the 9th chapter of [5].

$$(g^0, w_{i,1}), \ldots, (g^{U-1}, w_{i,U}), (g^U, 0), \ldots, (g^{N-1}, 0) \xrightarrow{\mathsf{IDFT}} a_0, a_1, \ldots, a_{N-1} \tag{4}$$

$$a_0, a_1, \ldots, a_{N-1} \xrightarrow{\mathsf{DFT}} (\gamma \cdot g^0, z_0), \ldots, (\gamma \cdot g^{M-1}, z_{M-1}) \tag{5}$$

The IDFT guarantees the coefficient representation of pre-encoded or fully-encoded witness polynomials is uniquely defined by N evaluations over G or H. This means the evaluation of LDEs adds redundancy to the representation of polynomials, resulting in a soundness factors of $\frac{1}{\beta} = \frac{N}{M} = \frac{|G|}{|H|}$. Using terminology from coding theory, each w_i can be seen as a word that is encoded as (blown-up to) a Reed Solomon code-word W_i. Since H is larger than and disjoint from G, LDEs can be used to detect cheating: if w_i differs from some polynomial in one evaluation, then W_i differs from the LDE of that polynomial in at least $1 - \frac{1}{\beta}$ the evaluations, resulting in $\mathsf{log}\,(\beta)$ bits of soundness per round.

Polynomial commitments To allow the Verifier to check polynomial properties over a given domain, some proof systems use Merkle-trees to commit to all possible evaluations of certain polynomials, as illustrated in Figure 2 for the domain $[0, 7]$. Given the Merkle-root, the Verifier can query any polynomial evaluation from the Prover, which in turn provides that particular evaluation and the Merkle-path to its respective leaf. This type of commitment must feature two properties: the Verifier cannot compute polynomial evaluations by itself, and the Prover cannot compute different polynomial evaluations for the same element. These properties are called **hiding** and **binding**, respectively, and they can be easily achieved in practice by choosing an adequate hash function. To improve commitment and evaluation time, a sequence of polynomials $(P_0, \ldots, P_k)$ can be committed together using their linear combinations. Formally, the coefficients from this combination should be chosen uniformly at random, but in practice they can be defined as powers of a single random α, as described in Equation 6. This way, the Prover can provide Merkle-paths to a particular evaluation of the combined polynomial, along with separate evaluations of all polynomials from the underlying linear combination, so the Verifier can check this set of polynomial evaluations match their combined polynomial evaluation.

$$P(X) \coloneqq \sum_{i=1}^{k} \alpha^{i-1} \cdot P_i(X) \tag{6}$$

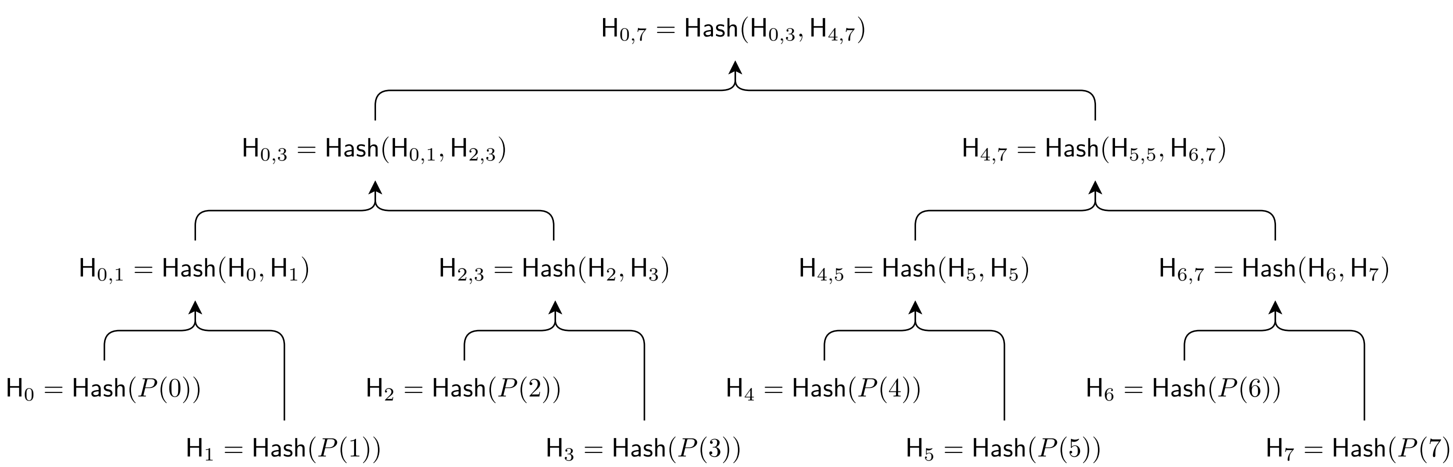

Figure 2: Polynomial commitments via Merkle-tree

3.2 STARK

The STARK transition function can be seen as an abstraction of CPU transition functions, meaning its function operates over the entire state of the algorithm execution. As a side-effect, the entire current and next state are given as input to the transition function, and witness selection is embedded into constraint polynomials. For instance, if a transition models a multiplication operation between witness values w_8 and w_7 and writes the result to the witness value w_6, then the entire current and next state of the set of witnesses are received by the constraint, but only the variables involved in the multiplication appear in the actual polynomial (see Figure 3). As a result, constraints represent the entire logic involved in a specific operation.

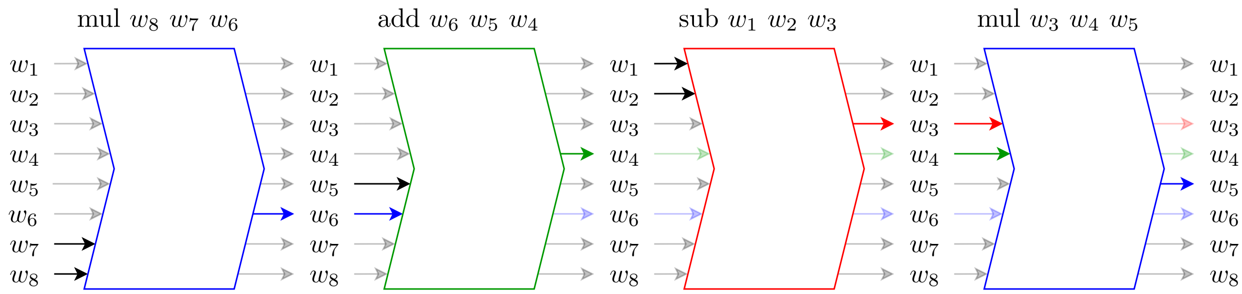

Figure 3: State transition on STARK-based zkVM

In their basic formulation, these constraint polynomials are $2V$-variate polynomial $C_1, ..., C_U$. The first V variables of each C_j represent the j-th state of the algorithm and the last V variables represent the subsequent state. Then, the vectors $W_j = (w_{1,j}, \dots, w_{V,j})$ and $\vec{w}_{j+1} = (w_{1,j+1}, \dots, w_{V,j+1})$ represent a valid state transition from the j-th to $(j+1)$-th state if and only if Equation 7 holds. Using notation for witness polynomials, the trace record is valid if and only if Equation 8 holds.

$$C_j\left(w_{1,j}, \dots, w_{V,j}, w_{1,j+1}, \dots, w_{V,j+1}\right) = 0 \;\forall\; j \in [1, U] \tag{7}$$

$$C_j(W_1(g^{j-1}), \dots, W_V(g^{j-1}), W_1(g^j), \dots, W_V(g^j)) = 0 \;\forall\; j \in [1, U] \tag{8}$$

The polynomial representation from Equation 7 applied to the example illustrated in Figure 3 can be seen in Equations 9 to 12. These transitions will be valid if and only if Equation 13 holds.

$$C_{j'}\left(w_{1,j'}, \dots, w_{8,j'}, w_{1,j'+1}, \dots, w_{V,j'+1}\right) = \left(w_{8,j'} \cdot w_{7,j'}\right) - w_{6,j'+1} \tag{9}$$

$$C_{j'+1}\left(w_{1,j'+1}, \dots, w_{8,j'+1}, w_{1,j'+2}, \dots, w_{V,j'+2}\right) = \left(w_{6,j'+1} + w_{5,j'+1}\right) - w_{4,j'+2} \tag{10}$$

$$C_{j'+2}\left(w_{1,j'+2}, \dots, w_{8,j'+2}, w_{1,j'+3}, \dots, w_{V,j'+3}\right) = \left(w_{1,j'+2} - w_{2,j'+2}\right) - w_{3,j'+3} \tag{11}$$

$$C_{j'+3}\left(w_{1,j'+3}, \dots, w_{8,j'+3}, w_{1,j'+4}, \dots, w_{V,j'+4}\right) = \left(w_{3,j'+3} \cdot w_{4,j'+3}\right) - w_{5,j'+4} \tag{12}$$

$$C_{j'}(W_1(g^{j'-1}), \dots, W_8(g^{j'-1})) = C_{j'+1}(W_1(g^{j'}), \dots, W_8(g^{j'})) = C_{j'+2}(W_1(g^{j'+1}), \dots, W_8(g^{j'+1})) = C_{j'+3}(W_1(g^{j'+2}), \dots, W_8(g^{j'+2})) = 0 \tag{13}$$

These particular polynomial compositions can be expressed as univariate polynomials by implicitly encoding witness polynomials inside constraint polynomials, as described in Equation 14. The key to this transformation is the fact that each witness polynomial appears twice in the polynomial composition, evaluated on g^{j-1} and on g^j, suggesting the replacement of g^{j-1} by X. Since these univariate polynomials represent the polynomial composition from Equation 8, they are called **composed polynomials**. Using composed polynomials, the trace record is valid if and only if Equation 15 holds.

$$C_j(X) \coloneqq C_j(W_1(X), \dots, W_V(X), W_1(g \cdot X), \dots, W_V(g \cdot X)) \;\forall\; j \in [1, U] \tag{14}$$

$$C_j(g^{j-1}) = 0 \;\forall\; j \in [1, U] \tag{15}$$

To check whether the j-th composed polynomial evaluates to 0 on g^{j-1}, the polynomial is divided by the polynomial of degree 1 that evaluates to 0 on this value, as described in Equation 16. The result is a rational function which is equivalent to a polynomial of degree $\mathsf{deg}\left(C_j\right) - 1$ if and only if Equation 15 holds. Since all of these rational functions are expected to be polynomials, they are called **quotient polynomials**. Using quotient polynomials, the trace record is valid if and only if Equation 17 holds.

$$Q_j(X) \coloneqq \frac{C_j(X)}{X - g^{j-1}} \ \forall\ j \in [1, U] \tag{16}$$

$$\mathsf{deg}\left(Q_j\right) = \mathsf{deg}\left(C_j\right) - 1 \ \forall\ j \in [1, U] \tag{17}$$

Witness and quotient polynomials are committed to using their linear combinations, as described in Equations 18 and 19, which are conveniently called the **combined witness polynomial** and the **combined quotient polynomial**.

$$W(X) \coloneqq \textstyle\sum_{i=1}^{V} \alpha^{i-1} \cdot W_i(X) \tag{18}$$

$$Q(X) \coloneqq \textstyle\sum_{j=1}^{U} \alpha^{j-1} \cdot Q_j(X) \tag{19}$$

To check the polynomial nature of the combined quotient polynomial, STARK uses the FRI protocol (see Section 3.2.1). This protocol convinces the Verifier with probability β^{-1} (for β the blow-up factor from the LDE) that a rational function is close to some polynomial of low-degree 2^d. This implies the correctness of the trace record when the rational function is defined as a valid quotient polynomial and d is defined as m (the logarithm of the witness polynomial degree).

The correctness of the trace record holds because the combined quotient function is close to some polynomial if and only if each individual quotient function is close to some polynomial of the same degree. For all quotient functions to be close to polynomials of the same degree, the respective underlying composed polynomials must also be of the same degree. Now take $d_{\mathsf{max}} \coloneqq \mathsf{max}_{j \in [1,U]}\left(\mathsf{log}\left(\mathsf{deg}\left(C_j\right)\right)\right)$, for $\{C_j\}_{j \in [1,U]}$ the composed polynomials from Equation 14, and take $D_{\mathsf{max}} = 2^{d_{\mathsf{max}}}$. Then, all quotient polynomials can be made of the same degree by *padding* each C_j with $X^{D_{\mathsf{max}}+1-\mathsf{deg}(C_j)}$ in Equation 16, where the $+1$ in the exponent of X ensures that the degree of resulting quotient polynomials is a power of 2.

$$Q_j(X) \coloneqq \frac{C_j(X)}{X - g^{j-1}} \cdot X^{D_{\mathsf{max}} - \mathsf{deg}(C_j)} \tag{20}$$

The entire generation and verification of these polynomials is called **Algebraic Linking IOP** (ALI) and is described in Algorithm 1.

Algorithm 1 Algebraic Linking IOP (ALI)

Prover computes the trace record from Page 8

Prover and Verifier computes $C_1, \dots, C_U$ from Pages 8 and 12

Verifier chooses $g \xleftarrow{\$} G$, $\gamma \xleftarrow{\$} \mathbb{F}^*$ such that $\mathsf{deg}\,(\gamma) = p - 1$, and $\alpha \xleftarrow{\$} \mathbb{F}^*$, and sends them to Prover

Prover computes $\vec{w}_1, \dots, \vec{w}_V$ from Pages 8 and 10

Prover computes $w_1, \dots, w_V$ from Equations 1 and 2

Prover computes $W_1, \dots, W_V$ from Equation 3

Prover computes $C_1, \dots, C_U$ from Equation 14

Prover computes $Q_1, \dots, Q_U$ from Equation 16

Prover computes W and Q from Equations 18 and 19

Prover computes commitments to W and Q, and sends them to Verifier

for all $k \in [0, \lceil \mathsf{log}_\beta\,(\epsilon) \rceil]$ **do**

  Verifier chooses $t_k \xleftarrow{\$} H$, and queries $W(t)$

  Provers sends Merkle path to $W(t_k)$, and evaluations $W(t_k), W_1(t_k), \dots, W_V(t_k)$

  if Equation 18 does not hold for $W(t_k), W_1(t_k), \dots, W_V(t_k)$ **then** Verifier rejects

  Verifier queries $W(g \cdot t_k)$

  Provers sends Merkle path to $W(g \cdot t_k)$, and evaluations $W(g \cdot t_k), W_1(g \cdot t_k), \dots, W_V(g \cdot t_k)$

  if Equation 18 does not hold for $W(g \cdot t_k), W_1(g \cdot t_k), \dots, W_V(g \cdot t_k)$ **then** Verifier rejects

  Verifier queries $Q(t_k)$

  Provers sends Merkle path to $Q(t_k)$, and evaluations $Q(t_k), Q_1(t_k), \dots, Q_U(t_k)$

  if Equation 19 does not hold for $Q(t_k), Q_1(t_k), \dots, Q_U(t_k)$ **then** Verifier rejects

Verifier accepts

3.2.1 Fast Reed-Solomon IOP of Proximity (FRI)

As explained in the end of the previous section, Q is a rational function that is equivalent to a polynomial if and only if the Prover uses suitable witness polynomials. This section describes a protocol called **Fast Reed-Solomon IOP of Proximity** (FRI) [1] that verifies this property. A single instance of the FRI protocol shows that a low-degree extended function is δ-close to a polynomial of degree lower than or equal to some fixed power of 2, say $D = 2^d$. This means that a fraction δ of all possible evaluations of this function equals the evaluations of a polynomial of degree D. The application of an LDE (see Page 10) adds a redundancy β required to the δ-closeness evaluation, implying $\delta = \beta^{-1} = N/M = 2^{n-m}$.

The word *fast* from the protocol refers to the FFT, because the recursion step used in FRI is inspired by the one used in the FFT. This recursion divides a function f_i in even and odd functions, namely f_i^{even} and f_i^{odd}, according to the degree of each of its factors. Then, it combines these functions to generate a closely-related function f_{i+1} that will be used in the next recursion step. Later, the relation between subsequent pairs of recursive functions is tested by appropriate commitment queries. The amount of times the relation between these recursive functions are tested proves the closeness to some polynomial of degree D. Namely, i tests imply (β^{-i})-closeness to such a polynomial.

FRI recursion is defined in a way to represent polynomial $f_i(X)$ as a combination of $f_i^{\mathsf{even}}(X^2)$ and $f_i^{\mathsf{odd}}(X^2)$, where f_i^{odd} is multiplied by X to correct the degree of its factors, as described in Equation 21 for $D_i \coloneqq \mathsf{deg}\,(f_i)$ such that D_i is even (which should hold for all iterations but the last). Since these even and odd functions are defined from their evaluation on X^2, their domain size and degree are also half the domain size and degree of their predecessor, as described in Equation 22.

$$f_i(X) = a_0 + a_1 \cdot X + \cdots + a_{D_i - 1} \cdot X^{D_i - 1} + a_{D_i} \cdot X^{D_i} =: f_i^{\mathsf{even}}(X^2) + X \cdot f_i^{\mathsf{odd}}(X^2) \;\Rightarrow$$
$$f_i^{\mathsf{even}}(X) = a_0 + \cdots + a_{D_i/2} \cdot X^{D_i/2} \text{ and } f_i^{\mathsf{odd}}(X) = a_1 + \cdots + a_{D_i/2-1} \cdot X^{D_i/2-1} \quad (21)$$

$$f_i : H_i \to \mathbb{F}^* \;\Rightarrow\; \begin{cases} f_i^{\mathsf{even}}, f_i^{\mathsf{odd}} : H_{i+1} \to \mathbb{F}^*, \text{ where } \left|H_{i+1} \coloneqq \{h^2 \mid h \in H_i\}\right| = \frac{1}{2} \cdot |H_i| \\ \mathsf{deg}\left(f_i^{\mathsf{even}}\right) \le \frac{1}{2} \cdot D_i \text{ and } \mathsf{deg}\left(f_i^{\mathsf{odd}}\right) \le \frac{1}{2} \cdot D_i - 1 \end{cases} \quad th \tag{22}$$

Then the Prover replaces the factor X multiplying f_i^{odd} by some $r_i \in \mathbb{F}^*$ chosen at random by Verifier, denotes the resulting function by f_{i+1}, as described in Equation 23. Once f_i^{even}, f_i^{odd} and f_{i+1} are defined, the Prover commits to these polynomials and sends their commitments to the Verifier. This is done for all $i \in [0, d]$ (assuming f_0 has been committed to) and it is called the **commit phase**.

$$f_{i+1}(X) \coloneqq f_i^{\mathsf{even}}(X) + r_i \cdot f_i^{\mathsf{odd}}(X) \tag{23}$$

After the commit phase, the Verifier checks the relation between each i-th and $(i+1)$-th recursive functions by querying $f_i(s_i)$, $f_i(-s_i)$ and $f_{i+1}(s_i^2)$, for some $s_i \in \mathbb{F}^*$ chosen at random. It uses $f_i(s_i)$ and $f_i(-s_i)$ to compute $f_i^{\mathsf{even}}(s_i^2)$ and $f_i^{\mathsf{odd}}(s_i^2)$, as described in Equations 24 and 25, and uses these values to check $f_{i+1}(s_i^2)$, as described in Equation 26. This process is called the **query phase**.

$$f_i^{\mathsf{even}}(s_i^2) = \frac{f_i(s_i) + f_i(-s_i)}{2} = \frac{f_i^{\mathsf{even}}(s_i^2) + \cancel{s_i \cdot f_i^{\mathsf{odd}}(s_i^2)} + f_i^{\mathsf{even}}(s_i^2) - \cancel{s_i \cdot f_i^{\mathsf{odd}}(s_i^2)}}{2} \tag{24}$$

$$f_i^{\mathsf{odd}}(s_i^2) = \frac{f_i(s_i) - f_i(-s_i)}{2 \cdot s_i} = \frac{\cancel{f_i^{\mathsf{even}}(s_i^2)} + \cancel{s_i} \cdot f_i^{\mathsf{odd}}(s_i^2) - \cancel{f_i^{\mathsf{even}}(s_i^2)} + \cancel{s_i} \cdot f_i^{\mathsf{odd}}(s_i^2)}{2 \cdot \cancel{s_i}} \tag{25}$$

$$f_{i+1}(s_i^2) \stackrel{?}{=} f_i^{\mathsf{even}}(s_i^2) + r_i \cdot f_i^{\mathsf{odd}}(s_i^2) \tag{26}$$

At the end of the query phase, $\mathsf{deg}(f_0) \le d$ if and only if $\mathsf{deg}(f_d) = 1$, i.e. f_d is a linear function. The Verifier is convinced of this fact with soundness β if f_{d+1} is a constant function, as described in Equations 27 and 28. This can be easily checked if the Verifier queries more than one evaluation of this function. For this reason and to increase the soundness guarantee, the query phase is run several times and the f_{d+1} evaluations queried in each of them are compared to each other.

$$f_d(X) = a_0 + a_1 \cdot X \eqqcolon f_d^{\mathsf{even}}(X^2) + X \cdot f_d^{\mathsf{odd}}(X^2) \;\Rightarrow f_d^{\mathsf{even}}(X) = a_0 \text{ and } f_d^{\mathsf{odd}}(X) = a_1 \tag{27}$$

$$f_{d+1}(X) \coloneqq f_d^{\mathsf{even}}(X) + r_d \cdot f_d^{\mathsf{odd}}(X) = a_0 + r_d \cdot a_1 \tag{28}$$

This entire process is illustrated in Figure 4 and described in Algorithm 2, for the target soundness from Section 3.2. For a rigorous analysis of FRI soundness, see [4].

Algorithm 2 Fast Reed-Solomon IOP of Proximity (FRI)

1: Prover sets $f_0 \coloneqq f$
2: Prover computes a commitment to f_0 and sends it to Verifier
3: **for all** $i \in [0, d]$ **do**
4: Prover computes f_i^{even} and f_i^{odd} from Equation 21
5: Verifier chooses $r_i \xleftarrow{\$} \mathbb{F}^*$ and sends it to Prover
6: Prover computes f_{i+1} from Equation 23
7: Prover computes a commitment to f_{i+1} and sends the commitment to Verifier
8: **for all** $k \in [0, \lceil \mathsf{log}_\beta(\epsilon) \rceil]$ **do**
9: **for all** $i \in [0, d]$ **do**
10: Verifier chooses $s_{k,i} \xleftarrow{\$} H$, and queries $f_i(s_{k,i})$, $f_i(-s_{k,i})$ and $f_{i+1}(s_{k,i}^2)$
11: Verifier computes $f_i^{\mathsf{even}}(s_{k,i}^2)$ and $f_i^{\mathsf{odd}}(s_{k,i}^2)$ from Equations 24 and 25
12: **if** Equation 26 does not hold for $f_{i+1}(s_{k,i}^2)$, $f_i^{\mathsf{even}}(s_{k,i}^2)$ and $f_i^{\mathsf{odd}}(s_{k,i}^2)$ **then** Verifier rejects
13: **if** $k > 0$ **and** $f_{d+1}(s_{k,d}^2) \ne f_{d+1}(s_{k-1,d}^2)$ **then** Verifier rejects
14: Verifier accepts

Finally, we stress that ZK properties follow partially from the hiding properties of the hash function used to generate Merkle trees. Since Verifier queries $\mathsf{log}_\beta(\epsilon)$ sets of polynomials, a negligible amount of information is leaked if $\mathsf{log}_\beta(\epsilon) = \mathsf{poly}(n)$. Intuitively, for $f : G \to \mathbb{F}^*$, consider that any $\mathsf{deg}(f) - 1$

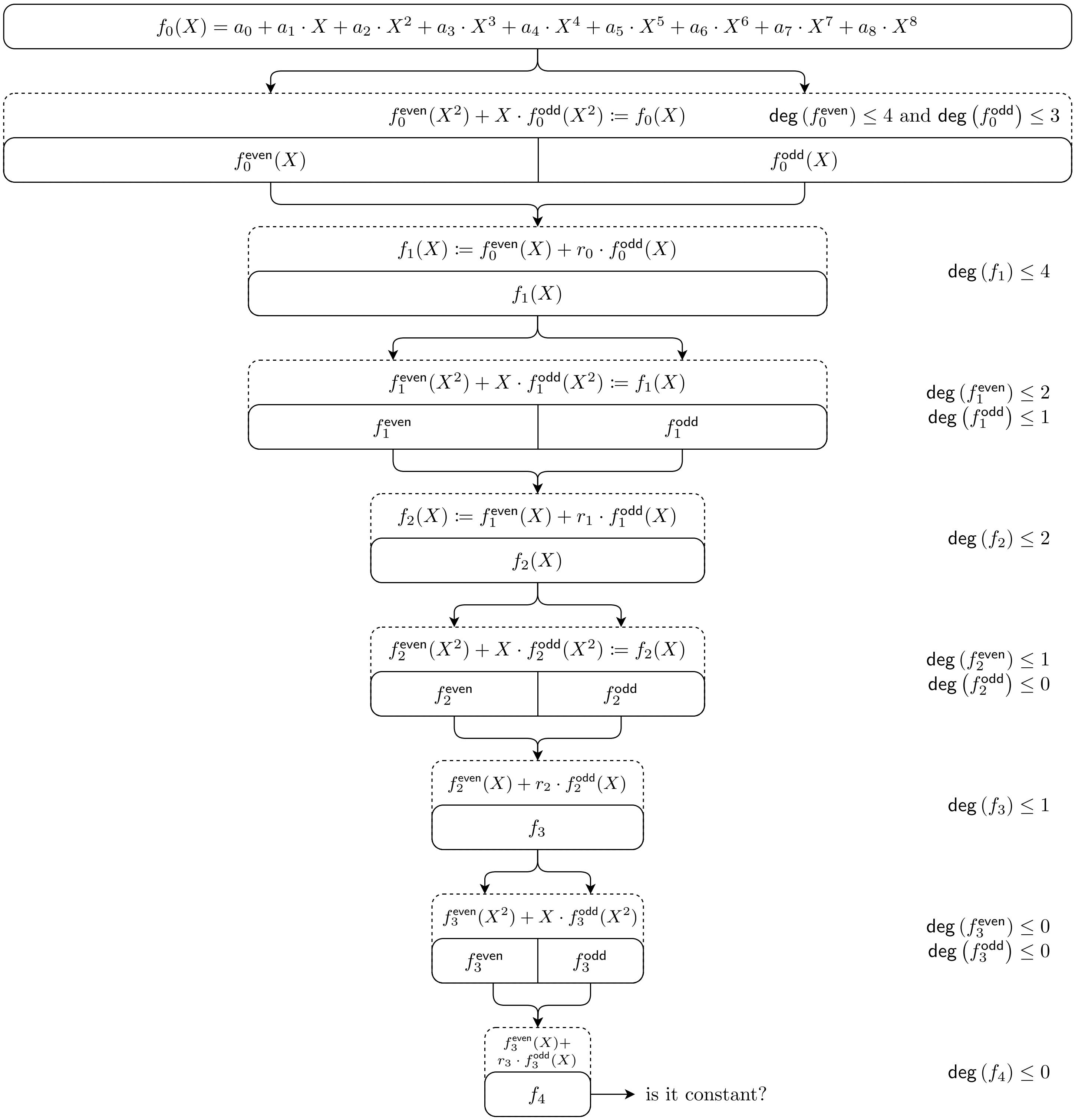

Figure 4: Fast Reed-Solomon IOP of Proximity (FRI)

evaluations of f define $|G|$ possibilities for it. Hence, our protocol inherits hiding properties from the Merkle tree's hash function as long as $\mathsf{log}_\beta(\epsilon) = \mathsf{poly}(U)$ and $|G| = O(2^U)$, which holds by construction. For a formal proof of this protocol's ZK properties, we refer to [4].

3.2.2 Domain Extending for Eliminating Pretenders (DEEP)

A single FRI execution provides β bits of soundness to the proof and therefore, for target ϵ bits of soundness, it must be repeated $\mathsf{log}_\epsilon(\beta)$ times. To achieve better soundness without repetition, polynomial degrees can be verified indirectly using an auxiliary polynomial evaluation on a value chosen by the Verifier after they receive polynomial commitments. This value must be sampled from *outside the box* that defines witness polynomials and their LDEs, which can be seen as a domain extension. This extension helps the Verifier to catch Provers cheating during the commitment opening and, for this reason, this method is called **Domain Extending for Eliminating Pretenders** (DEEP)[3].

The *boxes* that define witness polynomials and their LDEs are G and H, respectively, meaning the Verifier must choose a value $z \xleftarrow{\$} \mathbb{F}^* \setminus (G \cup H)$ and send it to the Prover. As a response, the Prover sends back evaluations of polynomials that will be DEEP-verified, namely $f_1(z), \dots, f_{d+1}(z)$, $f_1^{\mathsf{even}}(z), \dots, f_d^{\mathsf{even}}(z)$ and $f_1^{\mathsf{odd}}(z), \dots, f_d^{\mathsf{odd}}(z)$. Using these evaluations, Prover and Verifier define polynomials $f_1^z, \dots, f_{d+1}^z$ as in **??**, respectively, and use them instead of their non-DEEP counterparts whenever a relationship between these polynomials needs to be verified inside FRI.

$$f_i^{\mathsf{even},z}(X) \coloneqq f_i^{\mathsf{even}}(X) - f_i^{\mathsf{even}}(z) \quad f_i^{\mathsf{odd},z}(X) \coloneqq f_i^{\mathsf{odd}}(X) - f_i^{\mathsf{odd}}(z) \quad f_{i+1}^z(X) \coloneqq f_{i+1}(X) - f_{i+1}(z) \tag{29}$$

The choice of $z \notin (G \cup H)$ after the polynomials have been committed to implies that the soundness for FRI over their DEEP counterparts is greater than the soundness for FRI over the original polynomials. The reason for this gain in soundness is the existence of a polynomial F of degree D close to a function f if and only if there exists a polynomial F^z of degree $D-1$ close to f^z. Since z is chosen after the commitment to f, and the commitment scheme used is sufficiently binding (by assumption), there is no way for the Prover to cheat during future evaluations, specially in polynomial evaluations from FRI.

DEEP can be employed directly to ALI and FRI, in which case we call them DEEP-ALI and DEEP-FRI, and it guarantees roughly the same soundness to both protocols. However, the soundness gain of combining DEEP-ALI and DEEP-FRI is small and does not compensate for the increased complexity, hence we describe only DEEP-FRI in Algorithm 3 and the STARK formulation using DEEP-ALI and FRI in Algorithm 4.

Algorithm 3 Domain Extending for Eliminating Pretenders for Fast Reed-Solomon IOP of Proximity (DEEP-FRI)

Prover sets $f_0 \coloneqq f$
Prover computes a commitment to f_0 and sends it to Verifier
for all $i \in [0, d]$ **do**
    Prover computes f_i^{even} and f_i^{odd} from Equation 21
    Verifier chooses $r_i \xleftarrow{\$} \mathbb{F}^*$ and sends it to Prover
    Prover computes f_{i+1} from Equation 23
    Prover computes a commitment to f_{i+1} and sends the commitment to Verifier
Verifier $z \xleftarrow{\$} \mathbb{F}^* \setminus (G \cup H)$ and sends it to Prover
Prover computes $f_0(z)$ and sends it to Verifier
for all $i \in [0, d]$ **do**
    Prover computes $f_i^{\mathsf{even}}(z)$, $f_i^{\mathsf{odd}}(z)$ and $f_{i+1}(z)$, and sends them to Verifier
for all $i \in [0, d]$ **do**
    Verifier chooses $s_i \xleftarrow{\$} H$, and queries $f_i(s_i)$, $f_i(-s_i)$ and $f_{i+1}(s_i^2)$
    Verifier computes $f_i^{\mathsf{even}}(s_i^2)$ and $f_i^{\mathsf{odd}}(s_i^2)$ from Equations 24 and 25
    Verifier computes $f_i^z(s_i)$, $f_i^z(-s_i)$, $f_i^{\mathsf{even},z}(s_i^2)$, $f_i^{\mathsf{odd},z}(s_i^2)$ and $f_{i+1}^z(s_i^2)$ from Equation 29
    if Equation 24 does not hold for $f_i^z(s_i)$, $f_i^z(-s_i)$ and $f_i^{\mathsf{even},z}(s_i^2)$ **then** Verifier rejects
    if Equation 25 does not hold for $f_i^z(s_i)$, $f_i^z(-s_i)$ and $f_i^{\mathsf{odd},z}(s_i^2)$ **then** Verifier rejects
    if Equation 26 does not hold for $f_{i+1}^z(s_i^2)$, $f_i^{\mathsf{even},z}(s_i^2)$ and $f_i^{\mathsf{odd},z}(s_{k,i}^2)$ **then** Verifier rejects
if $f_{d+1}^z(s_d^2) \neq f_{d+1}^z(s_d^2)$ **then** Verifier rejects
Verifier accepts

Algorithm 4 Scalable and Transparent Argument of Knowledge (STARK)

Prover and Verifier engage in Algorithm 1
Prover and Verifier engage in Algorithm 3 for $f \coloneqq W$ and $f \coloneqq Q$

3.3 PLONK

PLONK transition function can be seen as an abstraction of circuit transition functions, meaning gates are abstracted as constraints and wires are abstracted as witness polynomials. In this setup, only the necessary subsets of the current and next states are given as input to transition functions. For instance, consider the example from the previous section, i.e. a transition modeling $w_6 = w_8 \cdot w_7$. In this example, the constraints receive as input the previous state of the subset of witnesses $\{w_8, w_7\}$ and yield as output the next state of the subset of witnesses $\{w_6\}$ (see Figure 5). As a result, these constraints represent the logic intrinsic to that specific operation, as described by the constraints from Equations 30 to 32.

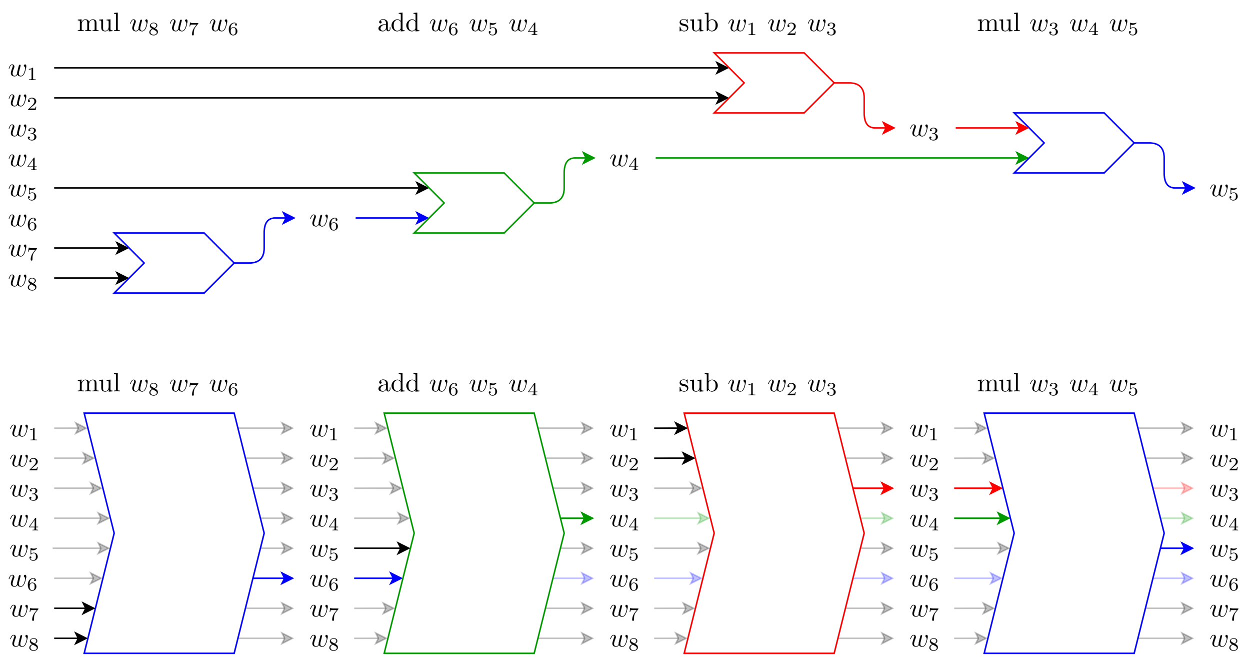

Figure 5: State transition on PLONK-based zkVM

$$C_{\mathsf{mul}}\left(X_{\mathsf{in1}}, X_{\mathsf{in2}}, X_{\mathsf{out}}\right) = \left(X_{\mathsf{in1}} \cdot X_{\mathsf{in2}}\right) - X_{\mathsf{out}} \tag{30}$$
$$C_{\mathsf{add}}\left(X_{\mathsf{in1}}, X_{\mathsf{in2}}, X_{\mathsf{out}}\right) = \left(X_{\mathsf{in1}} + X_{\mathsf{in2}}\right) - X_{\mathsf{out}} \tag{31}$$
$$C_{\mathsf{sub}}\left(X_{\mathsf{in1}}, X_{\mathsf{in2}}, X_{\mathsf{out}}\right) = \left(X_{\mathsf{in1}} - X_{\mathsf{in2}}\right) - X_{\mathsf{out}} \tag{32}$$

In their basic formulation, these constraint polynomials can model gates that connect to an arbitrary number of wires (the number of polynomial variables is arbitrary) and that are executed in no particular order. A structural side-effect from this weak polynomial definition is the need to ensure that the values used as input to each constraint equal the values output by the last constraints that modified that wire. To implement this *wiring* function, the proof system uses special constraints to ensure wire integrity and input selection, and special variables to ensure correct ordering of operations.

Constraints used for wire integrity are called **copy constraints**, and they can ensure the integrity of the same or different wires. When guaranteeing integrity of the same wire, copy constraints can embed the witness polynomial corresponding to that wire into a univariate polynomial as described in Equation 33. When guaranteeing integrity of different wires, copy constraints can embed the witness polynomials corresponding to those wires into a univariate polynomial, as described in Equation 34.

$$CC_i(X) = W_i(X) - W_i(g \cdot X) \tag{33}$$
$$CC_{i,i'}(X) = W_i(X) - W_{i'}(X) \tag{34}$$

Constraints used for input selection are called **selector constraints**, and they are defined as a combination of witness polynomials and special binary variables called **selector variables**. These variables depend only on the algorithm being proven, so they can be computed by the Verifier through combinations of Lagrange polynomials and certain polynomial identities. Inputs to the gates from Figure 5 can be selected by constraints S_{in1}, S_{in2} and S_{out} described in Equations 35 to 37 using sets of selector variables $\{S_{i,\mathsf{in1}}\}_{i\in[1,8]}$, $\{S_{i,\mathsf{in2}}\}_{i\in[1,8]}$ and $\{S_{i,\mathsf{out}}\}_{i\in[1,8]}$.

$$S_{\mathsf{in1}}(X) = \textstyle\sum_{i=1}^{8} S_{i,\mathsf{in1}}(X)\cdot W_i(X) \tag{35}$$
$$S_{\mathsf{in2}}(X) = \textstyle\sum_{i=1}^{8} S_{i,\mathsf{in2}}(X)\cdot W_i(X) \tag{36}$$
$$S_{\mathsf{out}}(X) = \textstyle\sum_{i=1}^{8} S_{i,\mathsf{out}}(X)\cdot W_i(g\cdot X) \tag{37}$$

The polynomials from Equations 30 to 32 applied to the example illustrated in Figure 5 can be seen in Equations 38 to 41. Combined with copy and selector constraints from Equations 33 to 37, their evaluation can be expressed as in Equation 42 (**composed polynomial**), where variables $\{S_i\}_{i\in[1,8]}$ select copy constraints $\{CC_i\}_{i\in[1,8]}$. These transitions will be valid if and only if the polynomial evaluations in Equations 43 to 46 hold (polynomial evaluations not listed in these equations should equal 0).

$$C_{\mathsf{mul}}(w_{8,j'}, w_{7,j'}, w_{6,j'+1}) \coloneqq (w_{8,j'}\cdot w_{7,j'}) - w_{6,j'+1} \tag{38}$$
$$C_{\mathsf{add}}(w_{6,j'+1}, w_{5,j'+1}, w_{4,j'+2}) \coloneqq (w_{6,j'+1} + w_{5,j'+1}) - w_{4,j'+2} \tag{39}$$
$$C_{\mathsf{sub}}(w_{1,j'+2}, w_{2,j'+2}, w_{3,j'+3}) \coloneqq (w_{1,j'+2} - w_{2,j'+2}) - w_{3,j'+3} \tag{40}$$
$$C_{\mathsf{mul}}(w_{3,j'+3}, w_{4,j'+3}, w_{5,j'+4}) \coloneqq (w_{3,j'+3}\cdot w_{4,j'+3}) - w_{5,j'+4} \tag{41}$$

$$\begin{aligned} C(X) \coloneqq\ & S_{\mathsf{mul}}(X)\cdot C_{\mathsf{mul}}(S_{\mathsf{in1}}(X), S_{\mathsf{in2}}(X), S_{\mathsf{out}}(X)) + S_{\mathsf{add}}(X)\cdot C_{\mathsf{add}}(S_{\mathsf{in1}}(X), S_{\mathsf{in2}}(X), S_{\mathsf{out}}(X)) + \\ & S_{\mathsf{sub}}(X)\cdot C_{\mathsf{sub}}(S_{\mathsf{in1}}(X), S_{\mathsf{in2}}(X), S_{\mathsf{out}}(X)) + \textstyle\sum_{i=1}^{8} S_i(X)\cdot (W_i(X) - W_i(g\cdot X)) = 0 \end{aligned} \tag{42}$$

$$S_{\mathsf{mul}}(g^{j'-1}) = 1 \quad S_{8,\mathsf{in1}}(g^{j'-1}) = 1 \quad S_{7,\mathsf{in2}}(g^{j'-1}) = 1 \quad S_{6,\mathsf{out}}(g^{j'-1}) = 1 \quad S_{i\neq 6}(g^{j'-1}) = 1 \tag{43}$$
$$S_{\mathsf{add}}(g^{j'}) = 1 \quad S_{6,\mathsf{in1}}(g^{j'}) = 1 \quad S_{5,\mathsf{in2}}(g^{j'}) = 1 \quad S_{4,\mathsf{out}}(g^{j'}) = 1 \quad S_{i\neq 4}(g^{j'}) = 1 \tag{44}$$
$$S_{\mathsf{sub}}(g^{j'+1}) = 1 \quad S_{1,\mathsf{in1}}(g^{j'+1}) = 1 \quad S_{2,\mathsf{in2}}(g^{j'+1}) = 1 \quad S_{3,\mathsf{out}}(g^{j'+1}) = 1 \quad S_{i\neq 3}(g^{j'+1}) = 1 \tag{45}$$
$$S_{\mathsf{mul}}(g^{j'+2}) = 1 \quad S_{3,\mathsf{in1}}(g^{j'+2}) = 1 \quad S_{4,\mathsf{in2}}(g^{j'+2}) = 1 \quad S_{5,\mathsf{out}}(g^{j'+2}) = 1 \quad S_{i\neq 5}(g^{j'+2}) = 1 \tag{46}$$

At this point, it should be clear that PLONK builds algorithmic structures by employing specialized constraints and variables, whereas STARK builds them using highly-specific constraints to model all the logic involved in a particular transition. An advantage of the PLONK approach is the fact that a single simple polynomial can encode the logic from the entire program, reducing proving steps and compiling proving logic to selectors that can easily be computed by the Verifier. STARK, on the other hand, requires several polynomial compositions that cannot be checked efficiently by the Verifier.

In this context, the **quotient polynomial** is defined as the division of the final constraint polynomial by a polynomial that evaluates 0 over all elements of G. This polynomial is known as the **vanishing polynomial** and, because G is a multiplicative subgroup of F^*, it can be computed as in Equation 47, thus the quotient polynomial can be computed as in Equation 48. The algebraic identity behind Equation 47 is the fact that the multiplicative subgroup of F^* with size N is unique and, because it is multiplicative, the n-th power of each element contained in it equals 1.

$$Z(X) \coloneqq (X - g^0)\cdot(X - g^1)\cdots(X - g^{N-1}) = X^N - 1 \tag{47}$$
$$Q(X) \coloneqq \frac{C(X)}{Z(X)} = \frac{C(X)}{X^N - 1} \tag{48}$$

Finally, with a single query over the quotient and the final constraint polynomial, the Verifier can check whether Equation 42 evaluates to 0 in every element of the trace domain G. This holds by design because selector polynomials ensure only the right constraint polynomials should hold in each step of the algorithm. In addition to the polynomial queries, Prover and Verifier may also engage in a FRI instance to check the low-degreeness of witness polynomials. The entire generation and verification of these polynomials is described in Algorithm 5.

Algorithm 5 Permutations over Lagrange-bases for Oecumenical Non-interactive Arguments-of-Knowledge (PLONK)

Prover computes the trace record from Page 8
Prover and Verifier compute $C_1, \ldots, C_U$ from Pages 8 and 19
Verifier chooses $g \xleftarrow{\$} G$, $\gamma \xleftarrow{\$} \mathbb{F}^*$ such that $\mathsf{deg}\left(\gamma\right) = p - 1$, and $\alpha \xleftarrow{\$} \mathbb{F}^*$, and sends them to Prover
Prover computes $\vec{w}_1, \ldots, \vec{w}_V$ from Pages 8 and 10
Prover computes $w_1, \ldots, w_V$ from Equations 1 and 2
Prover computes $W_1, \ldots, W_V$ from Equation 3
Prover computes C from Equation 42
Prover and Verifier compute Z from Equation 47
Prover computes Q from Equation 48
Prover computes W from Equation 18
Prover computes commitments to W and Q, and sends them to Verifier
for all $k \in [0, \lceil \mathsf{log}_\beta\left(\epsilon\right) \rceil]$ **do**
    Verifier chooses $t_k \xleftarrow{\$} H$, and queries $W(t)$
    Provers sends Merkle path to $W(t_k)$, and evaluations $W(t_k), W_1(t_k), \ldots, W_V(t_k)$
    if Equation 18 does not hold for $W(t_k), W_1(t_k), \ldots, W_V(t_k)$ **then** Verifier rejects
    Verifier queries $W(g \cdot t_k)$
    Provers sends Merkle path to $W(g \cdot t_k)$, and evaluations $W(g \cdot t_k), W_1(g \cdot t_k), \ldots, W_V(g \cdot t_k)$
    if Equation 18 does not hold for $W(g \cdot t_k), W_1(g \cdot t_k), \ldots, W_V(g \cdot t_k)$ **then** Verifier rejects
    Verifier queries $Q(t_k)$
    Provers sends Merkle path to $Q(t_k)$
    Verifier computes $C(t_k)$ from Equation 42
    if Equation 42 does not hold for each $C(t_k)$ and $W_1(t_k), \ldots, W_V(t_k), W_1(g \cdot t_k), \ldots, W_V(g \cdot t_k)$ **then** Verifier rejects
    if Equation 16 does not hold for each $Q_j(t_k)$ and $W_1(t_k), \ldots, W_V(t_k), W_1(g \cdot t_k), \ldots, W_V(g \cdot t_k)$ **then** Verifier rejects
Verifier accepts

3.4 Plonky2

Plonky2 implements variations of vanilla STARK and vanilla PLONK. In their implementation of STARK, called **Starky**, quotient polynomials are defined in a way to evaluate to zero in all elements from G, instead of only where the underlying composed polynomial should hold. This is done using Lagrange polynomials as in vanilla PLONK. As a side-effect, Starky constraints mix CPU and Circuit-based representations, meaning there are several constraints to choose from (though in practice it is better to choose as few as possible) and each of them receives the entire current and next states as input. In the end, Starky constraint and quotient polynomials are of the form described in Equations 49 and 50.

$$\begin{aligned} C(X) \coloneqq S_1(X) \cdot C_1(W_1(X), \ldots, W_V(X), W_1(g \cdot X), \ldots, W_V(g \cdot X)) + \\ S_2(X) \cdot C_2(W_1(X), \ldots, W_V(X), W_1(g \cdot X), \ldots, W_V(g \cdot X)) + \cdots + \\ S_{U'}(X) \cdot C_{U'}(W_1(X), \ldots, W_V(X), W_1(g \cdot X), \ldots, W_V(g \cdot X)) \end{aligned} \tag{49}$$

$$Q(X) \coloneqq \frac{C(X)}{X^n - 1} \tag{50}$$

In their implementation of PLONK, called **Plonky**, quotient polynomials are committed using DEEP-FRI instead of KZG, the original commitment scheme used in vanilla PLONK. As a side-effect, the Plonky polynomials (composed, quotient and vanishing polynomials) must now be low-degree extended. Because the arithmetization process is the same, the zero-test which verifies that quotient polynomials evaluate to 0 over G is still required, as well as the permutation proof which verifies that copy constraints hold over witness polynomials. Table 13 compares vanilla STARK, Starky, vanilla PLONKY and Plonky.

<table>
<tr><th rowspan="2">Feature</th><th rowspan="2">STARK</th><th colspan="2">Plonky2</th><th rowspan="2">PLONK</th></tr>
<tr><th>Starky</th><th>Plonky</th></tr>
<tr><td>Constraint representation</td><td>CPU-like</td><td>Mixed</td><td colspan="2">Circuit-like</td></tr>
<tr><td>Quotient representation</td><td>Local zeros</td><td colspan="3">Global zeros (Lagrange-based)</td></tr>
<tr><td>Polynomial commitments</td><td colspan="3">FRI</td><td>KZG</td></tr>
<tr><td>Commitment domain</td><td colspan="3">LDE of G</td><td>G</td></tr>
<tr><td>Quotient testing</td><td rowspan="3">FRI</td><td rowspan="3">DEEP-FRI</td><td colspan="2">Zero-test</td></tr>
<tr><td>Witness testing</td><td colspan="2">Permutation</td></tr>
<tr><td>Low-Degree testing</td><td>DEEP-FRI</td><td>-</td></tr>
</table>

Table 13: A comparison of STARK, Starky, PLONKY and Plonky.

Another important detail from Plonky2 is the algebraic field chosen to implement their proofs. The library uses the **Goldilocks field** $\mathbb{G} = \mathbb{F}_p$, for $p = 2^{64} - 2^{32} + 1$, which is fully compatible with 32-bit architectures but requires a few algebraic tricks to represent 64-bit values. For instance, 2^{96} cannot be natively represented as a Goldilocks element, but it is equivalent to -1 over $\mathbb{G}$ because 2^{64} is equivalent to $2^{32} - 1$ (see Equations 51 and 52). This fact allows the Goldilocks element n' equivalent to a 128-bit value n to be efficiently computed over this field using its 32-bit representation (n_0, n_1, n_2, n_3), as described in Equation 53.

$$2^{64} \equiv 2^{32} - 1 \pmod p \tag{51}$$

$$2^{96} \equiv 2^{64} \cdot 2^{32} \equiv (2^{32} - 1) \cdot 2^{32} \equiv 2^{64} - 2^{32} \equiv 2^{32} - 1 - 2^{32} \equiv -1 \pmod p \tag{52}$$

$$\begin{aligned} n = n_0 + 2^{32} \cdot n_1 + 2^{64} \cdot n_2 + 2^{96} \cdot n_3 \Rightarrow \\ n' \equiv n_0 + 2^{32} \cdot n_1 + (2^{32} - 1) \cdot n_2 + (-1) \cdot n_3 \equiv n_0 - n_2 - n_3 + (n_1 + n_2) \cdot 2^{32} \pmod p \end{aligned} \tag{53}$$

There are no native 64-bit operations in the MIPS version implemented by zkMIPS, but a similar trick is employed to represent 32-bit GPRs as pairs of 16-bit columns. Even though the choice for 16-bit columns increases the space necessary to represent GPRs, it results in smaller proofs because a lot of internal variables cannot surpass 2^{16}. This topic will be elaborated in Section 4.

3.5 LogUp

Modern arguments of knowledge often use special proof systems to show the correspondence between different vectors. These proof systems are called **lookup schemes**, and they prove that a given vector $\vec{v} = (v_0, v_1, \dots)$ is a multi-set of a **target vector** $\vec{t} = (t_0, t_1, \dots)$, i.e. they prove that for each $i \in [0, |v|)$, there exists some $j \in [0, |\vec{t}|)$ such that $t_j = v_i$. Given a **multiplicity vector** $\vec{m} = [m_0, \dots, m_{|\vec{t}|}]$, some lookup schemes can additionally show that there exist m_j possible values i such that $v_i = t_j$.

Using polynomials, $t_j = v_i$ can be expressed as in Equation 54. Similarly, assuming t is a simple set (each element appears once), $t_j = v_{i_1} = \cdots = v_{i_{m_j}}$ can be expressed as in Equation 55. In this setup, $\vec{v}$ is a multi-set of $\vec{t}$ if and only if Equation 56 holds. This property can be shown by letting the Prover commit to the left-hand and right-hand sides of this equation, and running some IOP with the Verifier to prove these polynomials are equal.

$$X - t_j = X - v_i \tag{54}$$

$$(X - t_j)^{m_j} = \prod_{k=1}^{m_j} (X - v_{i_k}) \tag{55}$$

$$\prod_{j=1}^{|t|} (X - t_j)^{m_j} = \prod_{i=1}^{|v|} (X - v_i) \tag{56}$$

zkMIPS uses a state-of-the-art lookup scheme called LogUp[9] to prove program instructions were correctly verified in their own special modules. LogUp proves the multi-set relationship between two vectors using the properties stated in Equations 57 and 58 instead of the ones from Equations 55 and 56. A full specification of this protocol will be given in the final version of this paper.

$$\frac{m_j}{X + t_j} = \sum_{k=1}^{m_j} \frac{1}{X + v_{i_k}} \tag{57}$$

$$\sum_{j=1}^{|t|} \frac{m_j}{X + t_j} = \sum_{i=1}^{|v|} \frac{1}{X + v_i} \tag{58}$$

4 High-level design of the zkMIPS protocol

The first step to prove the correct execution of a MIPS program inside zkMIPS is to collect every internal CPU state during the program execution. See Table 2. This can be done on the Prover side by running the program and storing into a table the value assumed by each CPU variable after each instruction. This table is a preliminary version of the **trace record**; it contains the columns described in Table 2. This preliminary trace allows the direct verification of state transitions during program execution by checking whether each pair of subsequent rows matches the MIPS CPU transition function.

The exact transition function implied by each instruction is defined according to the MIPS specifications (see Tables 5 to 11) and the way it is proved in a zkVM depends on the chosen proof model. In order to reduce complexity and increase efficiency, we decided to divide the actual zkMIPS proving procedure in three dependent layers illustrated in Figure 6 and described below.

First layer: proving segments are consistent Program execution is divided into small sequential executions called **segments**. Each segment is proved independently in the second proving layer. To distinguish the trace proved in the first layer from those proved in the second layer, we call the trace proved in the first layer the **program trace**. It is important to stress that the program trace is not explicitly logged in practice; instead, the MIPS VM running on the first layer only logs the first and the last CPU states from each segment. When all proofs from the second layer have been produced, they are recursively combined into one single proof for the correct execution of the entire program trace. This process is called continuation and, for this reason, the proof produced at the end of this layer is called a **continuation proof**.

Second layer: division in modules Each segment execution is divided into smaller, non-sequential, executions called **modules**, named in a reference to CPU modules responsible for special instructions processing. Each module combines all segment instructions from an independent subset of MIPS instructions and is proved independently in the third proving layer. Namely, the main proving modules are arithmetic, logic, memory and control, and the instructions proved by them are described in Table 14. Additionally, a special Keccak hashing CPU module is simulated through modules optimized for this operation, namely the Keccak and Keccak-Sponge modules. To distinguish traces proved in the second layer from those proved in the third layer, we call traces proved in the second layer **segment traces**. Unlike the program trace, segment traces must be logged. When all proofs from the third layer have been produced, they are combined into one single proof (using a lookup scheme) for the correct execution of the entire segment trace. This proof is called a **segment proof**.

Third layer: specialized STARK proofs each module execution is proved independently using specialized STARK proofs. This layer is where transition functions are finally proved. Traces proved in the third layer are called **module traces** and they do not contain repeated instructions, as might be the case for segment traces. We consider two instructions repeated if they have the same MIPS instruction and same input values, with possibly different input registers (the values in these registers when the instructions are executed must be the same). This property can slightly reduce proving redundancy and improve performance.

It should be clear that continuation proofs cannot be produced in the first layer before segment proofs have been produced in the second layer, because continuation proofs depend on segment proofs. However, segment proofs can be produced in the second layer before module proofs have been produced in the third layer, because segment proofs are simply lookup proofs from segment to module traces. Chronologically, this means second and third layers can run in parallel.

There is a clear correspondence between the proof systems described in Figure 1 and the proving layers described in Figure 6, except for the additional layer from the latter. The main proving layers and how their proofs are composed will be explained in Sections 4.1 to 4.3. The additional layer and the choice of the proof system it uses will be described in Section 4.4.

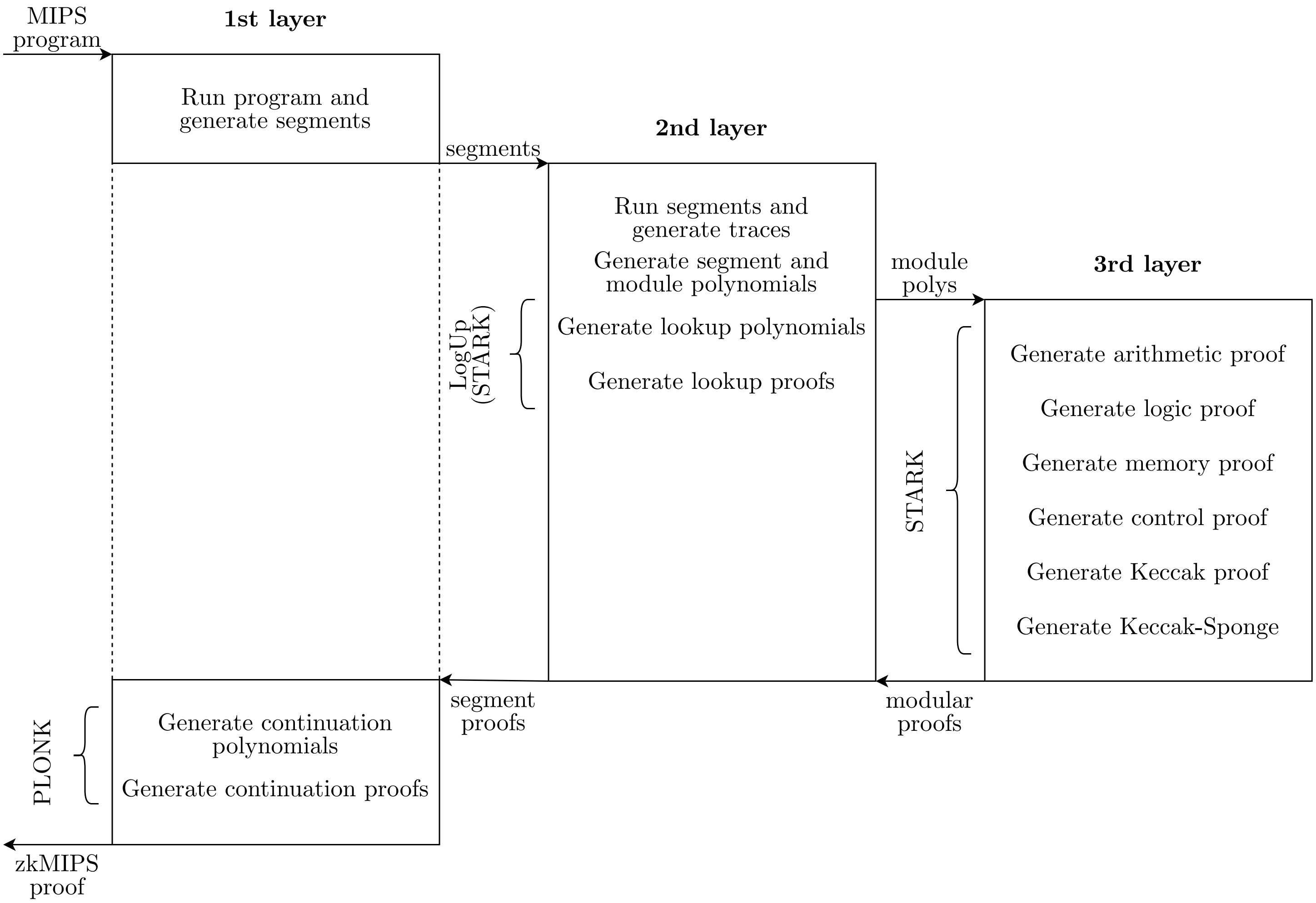

Figure 6: Proving layers

#	Name	Type
1	ADD	arithmetic
2	ADDI	arithmetic
3	ADDIU	arithmetic
4	ADDU	arithmetic
7	DIV	arithmetic
8	DIVU	arithmetic
9	MUL	arithmetic
10	MULT	arithmetic
11	MULTU	arithmetic
12	SLT	arithmetic
13	SLTI	arithmetic
14	SLTIU	arithmetic
15	SLTU	arithmetic
16	SUB	arithmetic
17	SUBU	arithmetic
43	LUI	logic
49	MFHI	move
50	MFLO	move
53	MTHI	move
54	MTLO	move
55	SLL	shift
56	SLLV	shift
57	SRA	shift
58	SRAV	shift
59	SRL	shift
60	SRLV	shift

(a) Arithmetic module

#	Name	Type
5	CLO (?)	arithmetic
6	CLZ (?)	arithmetic
42	AND	logic
44	NOR	logic
45	OR	logic
46	ORI	logic
47	XOR	logic
48	XORI	logic

(b) Logic module

#	Name	Type
28	LB	load/store and memory
29	LBU	load/store and memory
30	LH	load/store and memory
31	LHU	load/store and memory
32	LL	load/store and memory
33	LW	load/store and memory
34	LWL	load/store and memory
35	LWR	load/store and memory
36	SB	load/store and memory
37	SC	load/store and memory
38	SH	load/store and memory
39	SW	load/store and memory
40	SWL	load/store and memory
41	SWR	load/store and memory
51	MOVN (?)	move
52	MOVZ (?)	move

(c) Memory module

#	Name	Type
18	BEQ	branch and jump
19	BGEZ	branch and jump
20	BGTZ	branch and jump
21	BLEZ	branch and jump
22	BLTZ	branch and jump
23	BNE	branch and jump
24	J	branch and jump
25	JAL	branch and jump
26	JALR	branch and jump
27	JR	branch and jump
61	SYSCALL	trap

(d) Control module

Table 14: Opcodes implemented by each proving module

4.1 Continuation proofs

The first proving layer in zkMIPS proves that segments are consistent with each other. This layer runs in two steps that are invoked separately by the entity proving the program. First, zkMIPS is invoked to run the input program and divide its execution into segments, which is done independently of the main proving procedure. At the end, when all segment proofs have been generated, this layer is invoked again to combine them into a single proof that shows the correctness of the entire program execution.

The program division procedure periodically pauses the VM running the MIPS program after a constant number of instructions (the size of segments, which is received as input by zkMIPS) and stores the memory state, the PC and the cycle counter at that point. This set of variables is called the **image id** and it is given as private input to the segment prover as it tells exactly where and how to start the proving procedure. A compressed version of the image id is given as public input to the segment prover. This compression replaces the memory state with a Merkle tree of the memory state, which is enough to allow the verification of every memory access made by the Prover, using Merkle paths.

When the continuation proving is invoked, image ids are used to show subsequent segments match, by comparing the initial image id of each segment with the last image id of the previous segment. Once the proof for the correspondence of a pair of subsequent segments is ready, their underlying polynomials and FRI proofs are batched together (segment proofs are Starky) and are attached to their correspondence proof, thus creating a proof for the execution of the larger segment trace.

This procedure is repeated recursively until a single proof is obtained, proving the correctness of the entire program trace. Because continuation proofs are mostly FRI proofs, whose verification steps are also recursive, a verifier circuit for these proofs is relatively simple. Thus, continuation proofs can be written using Plonky instead of Starky, which further reduces proof size, proving time and verification complexity. The fact that Plonky and Starky use the same field and have their polynomials low-degree tested by the same protocol makes them fully compatible and composable with each other.

4.2 Segment proofs

The second proving layer is where the most expressive parts of zkMIPS proofs are generated. Given the memory state, PC and cycle counter from the beginning of a segment, this prover executes the program from that point and collects memory hashes from each instruction of that segment until the segment ends. The image id from the last instruction is the output of that proof and should be equal to the input of the next proof.

Segment and module traces are divided into columns containing 16-bit values. Because proofs are written using Plonky2, 16-bit values are represented by Goldilocks elements, which are roughly 64 bits. A lookup verifies that values stored are inside the 16-bit range. This lookup uses a special **range counter** column containing all values allowed. The reason for this 64-bit representation of 16-bit values comes from a trade-off: 16-bit values imply 2^{16} elements in the range counter column and a segment size at least this large; 32-bit values imply the segment size is greater than 2^{32}. Empirically, segment sizes between 2^{18} and 2^{23} result in the best performance.

Since MIPS is a 32-bit architecture, the values stored in GPRs take up two columns each. However, segment and module trace columns do not contain values from all GPRs in every cycle. Instead of logging GPRs directly into the trace record, the segment prover logs them into a register file in the format of Table 2, and then converts these values to a trace in the format of Table 15.

Including all registers in the trace record requires copy constraints to ensure values do not change when they do not have to (when a register is not touched during the execution of an instruction). Logging 32 register values of 32-bit each into 16-bit trace columns would require 64 columns for these values, and 32 columns for variables selecting their copy constraints. Currently, zkMIPS trace record has 57 columns, meaning GPR inclusion would increase trace size by more than 150%.

Each state of the register file where GPRs are logged to must be written to zkMIPS memory. This means memory needs to be accessed in every instruction and register consistency is guaranteed by memory and Keccak modules (memory accesses require Keccak-based Merkle paths) instead of the segment proof. This approach might seem costly at first, but it completely removes the need for copy constraints because the register file can be succinctly modified by constraint polynomials that change specific memory positions.

<table>
<tr><th></th><th>Group</th><th colspan="2">Arithmetic</th><th></th><th></th><th></th><th></th><th>MULT/DIV</th></tr>
<tr><td>1</td><td rowspan="26">OPCODE COLS</td><td colspan="7">IS_ADD</td></tr>
<tr><td>2</td><td colspan="7">IS_ADDU</td></tr>
<tr><td>3</td><td colspan="7">IS_ADDI</td></tr>
<tr><td>4</td><td colspan="7">IS_ADDIU</td></tr>
<tr><td>5</td><td colspan="7">IS_SUB</td></tr>
<tr><td>6</td><td colspan="7">IS_SUBU</td></tr>
<tr><td>7</td><td colspan="7">IS_MULT</td></tr>
<tr><td>8</td><td colspan="7">IS_MULTU</td></tr>
<tr><td>9</td><td colspan="7">IS_MUL</td></tr>
<tr><td>10</td><td colspan="7">IS_DIV</td></tr>
<tr><td>11</td><td colspan="7">IS_DIVU</td></tr>
<tr><td>12</td><td colspan="7">IS_SLLV</td></tr>
<tr><td>13</td><td colspan="7">IS_SRLV</td></tr>
<tr><td>14</td><td colspan="7">IS_SRAV</td></tr>
<tr><td>15</td><td colspan="7">IS_SLL</td></tr>
<tr><td>16</td><td colspan="7">IS_SRL</td></tr>
<tr><td>17</td><td colspan="7">IS_SRA</td></tr>
<tr><td>18</td><td colspan="7">IS_SLT</td></tr>
<tr><td>19</td><td colspan="7">IS_SLTU</td></tr>
<tr><td>20</td><td colspan="7">IS_SLTI</td></tr>
<tr><td>21</td><td colspan="7">IS_SLTIU</td></tr>
<tr><td>22</td><td colspan="7">IS_LUI</td></tr>
<tr><td>23</td><td colspan="7">IS_MFHI</td></tr>
<tr><td>24</td><td colspan="7">IS_MTHI</td></tr>
<tr><td>25</td><td colspan="7">IS_MFLO</td></tr>
<tr><td>26</td><td colspan="7">IS_MTLO</td></tr>
<tr><td>27</td><td rowspan="20">SHARED COLS</td><td colspan="2" rowspan="2">INPUT_REG_0</td><td rowspan="2">AUX_REG_0</td><td rowspan="2">MOD_OUT_ AUX_RED</td><td></td><td rowspan="2">MOD_ INPUT_0</td><td></td></tr>
<tr><td>28</td><td></td><td></td></tr>
<tr><td>29</td><td colspan="2" rowspan="2">INPUT_REG_1</td><td rowspan="4">AUX_REG_1</td><td colspan="2">MOD_MOD_IS_ZERO</td><td rowspan="2">MOD_ INPUT_1</td><td></td></tr>
<tr><td>30</td><td rowspan="3">MOD_AUX_ INPUT_LO</td><td></td><td></td></tr>
<tr><td>31</td><td colspan="2" rowspan="2">INPUT_REG_2</td><td></td><td rowspan="2">MOD_ MODULUS</td><td></td></tr>
<tr><td>32</td><td></td><td></td></tr>
<tr><td>33</td><td colspan="2" rowspan="2">OUTPUT_REG</td><td rowspan="3">AUX_REG_2</td><td rowspan="3">MOD_AUX_ INPUT_HI</td><td></td><td rowspan="2">MOD_ OUTPUT</td><td rowspan="2">OUTPUT_ REG_LO</td></tr>
<tr><td>34</td><td></td></tr>
<tr><td>35</td><td rowspan="2">AUX_INPUT_ REG_0</td><td rowspan="4">AUX_INPUT_ REG_DBL</td><td rowspan="2">MUL_AUX_ INPUT_LO</td><td rowspan="4">MOD_QUO_ INPUT</td><td rowspan="2">OUTPUT_ REG_HI</td></tr>
<tr><td>36</td><td colspan="2">MOD_DIV_DENOM_IS_ZERO</td></tr>
<tr><td>37</td><td rowspan="2">AUX_INPUT_ REG_1</td><td></td><td></td><td rowspan="2">MUL_AUX_ INPUT_HI</td><td rowspan="4">MULT_ AUX_LO</td></tr>
<tr><td>38</td><td></td><td></td></tr>
<tr><td>39</td><td colspan="2" rowspan="2">AUX_INPUT_REG_2</td><td></td><td></td><td></td><td></td></tr>
<tr><td>40</td><td></td><td></td><td></td><td></td></tr>
<tr><td>41</td><td></td><td></td><td></td><td></td><td></td><td></td><td rowspan="4">MULT_ AUX_HI</td></tr>
<tr><td>42</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>43</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>44</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>45</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>46</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>47</td><td rowspan="11">EXTRA COLS</td><td colspan="7">RANGE_COUNTER</td></tr>
<tr><td>48</td><td colspan="7">RC_FREQUENCIES</td></tr>
<tr><td>49</td><td colspan="7" rowspan="8">AUX_EXTRA</td></tr>
<tr><td>50</td></tr>
<tr><td>51</td></tr>
<tr><td>52</td></tr>
<tr><td>53</td></tr>
<tr><td>54</td></tr>
<tr><td>55</td></tr>
<tr><td>56</td></tr>
<tr><td>57</td><td colspan="7">NUM_ARITH_COLUMNS</td></tr>
</table>

Table 15: Trace columns

These 57 trace columns are divided into three main groups described below:

- Opcode columns (1 to 26) define which arithmetic operation should be proven in a given row.
- Shared columns (27 to 46) contain columns used by module proofs. The most important columns from this group are input and output register columns (27 to 34) which, as the name suggests, receive the values input to and output by instructions. The role of each shared column changes depending on the instruction, and sometimes zkMIPS uses macros to refer to these roles easily. Table 15 shows these macros and to which shared columns they refer to.
- Extra columns (47 to 57) include columns used to verify other columns are well-formatted. The most important columns from this group are the range counter column (47) and the frequency counter columns (48), which count how many times the range counter value from the same row appears in other columns, i.e. the multiplicity vector from the range counter lookup.

Once the segment and module traces have been generated, which happens in parallel as they encode the same rows, they are compiled into segment and module trace polynomials. In parallel to module proving, the segment provers can compile segment and module columns to LogUp polynomials.

4.3 Module proofs

The third proving layer is where the most meaningful parts of zkMIPS proofs are processed. This layer ensures the correctness of polynomials defined in segment proofs, but in practice there is no distinction between these two proving layers. The distinction made in this document is conceptual and tries to abstract what is proved by pure Starky (third layer) from what is proved by lookup proofs (second layer).

Constraint and witness polynomials for segment and module proofs can be generated and processed in parallel because they are the same. Constraint polynomials are in theory the same; in practice, there are no explicit constraints for segment proofs, since they are simply lookups. Witness polynomials, on the other hand, evaluate to the same values in the same order, with segment columns being defined sequentially and module columns non-sequentially; in other words, the set of values in segment columns equals the union of set of values in module columns.

Since these module witness polynomials have their correctness evaluated by Starky proofs, they are eventually low-degree extended to the same domains and proved in parallel. This LDE and the subsequent FRI commit and query phases are executed at the end of each segment, along with the lookup proof. The resulting proofs are combined into the segment proof.

The final version of this paper will elaborate on how constraint polynomials from module proofs are generated and how these proofs are combined at the end of a segment proving.

4.4 On-chain proofs

The optional proving layer compiles the final hash-based Plonky proof, output by the continuation process, into an elliptic-curve-based Groth16 proof. The verification of this proof requires a pairing function that is natively supported by the EVM. This improves on-chain verification performance because the hash functions necessary for FRI verification do not have to be simulated on-chain. Instead, only a succinct verification of this hash function (batched to the proof of modules) is performed by means of a Groth16 proof for the hash function verification algorithm.

Given a hash of the initial memory state of a program, the final continuation proof guarantees that, starting from the first instruction of this program, there exists a sequence of valid CPU states that halts a MIPS VM with the correct result. This property ensures, by design, logic, memory and register integrity.

5 Future work

The design described here is still a work in progress and will be continuously improved to ensure zkMIPS remains relevant in the field. Whenever a new feature is added to the codebase, it will be incorporated into this document. We invite everyone reading this document to contribute to our GitHub repository. Readers can give us feedback on this paper in our feedback channel on Discord, and ask questions in our questions channel.

Currently, the zkMIPS development team is preparing two modifications to the codebase. The first is the implementation of a modification to the LogUp proof system. Our code currently uses the same IOP from the original LogUp paper[9]. Recently, inspired by the Lasso proof system[13], the paper was updated[11] with a protocol called GKR[7]. This update improved the IOP used to globally verify polynomial properties of LogUp polynomials, allowing the optimization introduced by Lasso to be employed in our proof system as well. This white paper will be updated in the coming weeks with a detailed description of the revised LogUp protocol.

The second optimization is the replacement of the hash function used to compute Merkle trees. The new hash function, called Poseidon2, is optimized for Zero-Knowledge proofs. The current hash function, Keccak, was chosen for its native compatibility with EVM bytecodes, making on-chain verification of the Groth16 proof cheaper. Poseidon2, on the other hand, is natively compatible with ZKPs, making Merkle roots proving cheaper in the off-chain proving layers, thus indirectly reducing the final proof size and verification time. This white paper will be updated in the coming weeks with a detailed discussion on the hash function choice.

Once these updates to our codebase are ready, an additional performance section will be added with a benchmark of the new proof system and a comparison to competitors, along with the description of these modifications. Whenever new features are being considered for zkMIPS, the future work section will be updated with a brief description of the planned updates.

References

[1] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity, 2018.

[2] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018. https://eprint.iacr.org/2018/046.

[3] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. Deep-fri: Sampling outside the box improves soundness. Cryptology ePrint Archive, Paper 2019/336, 2019. https://eprint.iacr.org/2019/336.

[4] Alexander R. Block, Albert Garreta, Jonathan Katz, Justin Thaler, Pratyush Ranjan Tiwari, and Michal Zajac. Fiat-shamir security of fri and related snarks. Cryptology ePrint Archive, Paper 2023/1071, 2023. https://eprint.iacr.org/2023/1071.

[5] Gilles Brassard and Paul Bratley. *Algorithmics - theory and practice*. Prentice Hall, 1988.

[6] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019. https://eprint.iacr.org/2019/953.

[7] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4), sep 2015.

[8] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. https://eprint.iacr.org/2016/260.

[9] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022. https://eprint.iacr.org/2022/1530.

[10] Mips architecture for programmers volume ii-a: The mips32 instruction set manual. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf, 2016.

[11] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using gkr. Cryptology ePrint Archive, Paper 2023/1284, 2023. https://eprint.iacr.org/2023/1284.

[12] PolygonZero. Plonky2: Fast recursive arguments with plonk and fri. https://github.com/0xPolygonZero/plonky.

[13] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023. https://eprint.iacr.org/2023/1216.