Optimize zkEVM throughput: Series II

Bangkok 14th November 2024 - Devcon VII

Carlos Matallana
Protocol Team Lead
Ignasi Ramos
Protocol Team Engineer

README

https://github.com/0xPolygonHermez/zkevm-rom/blob/devcon-24/README-workshop.md

Presentation Outline

- 1. zkASM
    - Overview
    - Registers
    - Instructions
    - Examples
    - zkCounters
- 2. zkASM optimizations
- 3. Code: **opPUSH**
    - Diagram
    - Tests
- 4. Code: **mloadX/mstorex**

1

zkASM

zkASM: Overview

zkASM is the language developed by the team that is used to write the program that a compiler will build and the executor will interpret in order to build the execution trace.

```
STEP => A
0                                   :ASSERT ; Ensure it is the beginning of the execution

CTX                                 :MSTORE(forkID)
CTX - %FORK_ID                      :JMPNZ(failAssert)

B                                   :MSTORE(oldStateRoot)
```

zkASM: Overview

zkASM Compiler

We have implemented a zkASM compiler that reads a zkASM specification file and compiles it to an output file with the list steps and instructions which the executor will consume in order to compute the execution trace.

Let’s check the code!

zkASM: Registers Types

- 8 slots registers → [**V0**, **V1**,..., **V7**]
  - Each element: Goldilocks number
    - Prime field: 2^64 - 2^32 + 1
    - 63.99 bits
  - 32 bits are used for each slot
  - 8 * 32 = 256 bits

- 1 slot register → **V0**
  - One element: Goldilocks number
    - Prime field: 2^64 - 2^32 + 1
    - 63.99 bits

zkASM: Registers

- 8 slots registers → [**V0**, **V1**,..., **V7**]
  - Generic registers: **A**, **B**, **C**, **D**, **E**
  - **SR**: state root
  - **ROTL_C**: Rotate left register **C**
- 1 slot register → **V0**
  - zkCounters: **CNT_ARITH**, **CNT_BINARY**, **CNT_KECCAK_F**, **CNT_MEM_ALIGN**, **CNT_PADDING_PG**, **CNT_POSEIDON_G**, **CNT_STEPS**, **CNT_SHA256**
  - **CTX**: context number
  - **SP**: stack pointer
  - **PC**: smart contract program counter
  - **GAS**: available transaction gas
  - **zkPC**: zkrom program counter
  - **RR**: return register
  - **STEP**: zkrom step counter
  - **MAXMEM**: tracks maximum memory
  - **HASHPOS**: hash position

zkASM: Instructions (0)

- **MLOAD**: load from memory
  - op = mem[addr]

- **MSTORE**: save to memory
  - mem[addr] = op

- **HASHK**: add bytes to keccak
  - hashK[**E**].data[**HASHPOS**...**HASHPOS**+**D**-1] = op[0..**D**-1]
  - **HASHPOS** := **HASHPOS** + **D**

- **HASHKLEN**: digest keccak
  - hashK[**E**].length = op
  - hashK[**E**].digest = keccak(hashK[**E**].data)

- **HASHKDIGEST**: retrieve keccak hash digest
  - hashK[**E**].digest = op

zkASM: Instructions (1)

- **HASHP**: add bytes to linear poseidon
  - hashP[**E**].data[**HASHPOS**...**HASHPOS**+**D**-1] = op[0..**D**-1]
  - **HASHPOS** := **HASHPOS** + **D**

- **HASHPLEN**: digest linear poseidon
  - hashP[**E**].length = op
  - hashP[**E**].digest = poseidon(hashP[**E**].data)

- **HASHPDIGEST**: retrieve linear poseidon hash digest
  - hashP[**E**].digest = op

- **JMP**: jump to specific zk program counter
  - zkPC’ = addr
- **JMPN**: jump if negative
  - zkPC’ = (op < 0) ? addr : zkPC + 1;
- **JMPC**: jump if carry is positive
  - zkPC’ = (carry === 1) ? addr : zkPC + 1;

zkASM: Instructions (2)

- **CALL**: jump to specific routine and set return register
  - zkPC' = addr
  - **RR** = op0 (**CONST**)

- **RETURN**: go back to return register
  - zkPC' = addr (loaded from RR)

- **ASSERT**: assertion
  - **A** === **op**

- **SLOAD**: storage state-tree load
  - key0 = [**C0**, **C1**, **C2**, **C3**, **C4**, **C5**, **C6**, **C7**]
  - key1 = [**A0**, **A1**, **A2**, **A3**, **A4**, **A5**, **B0**, **B1**]
  - key = HP(key1, HP(key0))
  - op = storage.get(**SR**, key)

zkASM: Instructions (3)

- **SSTORE**: storage state-tree store
  - key0 = [C0, C1, C2, C3, C4, C5, C6, C7]
  - key1 = [A0, A1, A2, A3, A4, A5, B0, B1]
  - value = [D0, D1, D2, D3, D4, D5, D6, D7]
  - SR’ = storage.get(SR, key, value)

- **ARITH**: arithmetic state machine
  - A*B + C = D*(2^256) + op

- **ADD**: addition
  - A + B = op (set carry as overflow)

- **SUB**: subtraction
  - A - B = op (set carry as underflow)

- **LT**: less than
  - (A < B) = op (set carry as a result)

zkASM: Instructions (4)

- **SLT**: signed less than
  - signed(**A** < **B**) = op (set carry as a result)
- **EQ**: equal
  - (**A** == **B**)= **op** (set carry as a result)
- **AND**: bitwise logic gate **AND**
  - **A** & **B** = **op**
- **OR**: bitwise logic gate **OR**
  - **A** | **B** = **op**
- **XOR**: bitwise logic gate **XOR**
  - **A** ^ **B** = **op**

zkASM: Instructions (5)

- **MEM_ALIGN_RD**: memory align read
  - MemorySlot0=**A**, MemorySlot1=**B**, Value=op, Offset=**C**
  - Value == (MemorySlot0 << offset*8) | (MemorySlot1 >> 256 - offset*8)

- **MEM_ALIGN_WR**: memory align write 32 bytes
  - MemorySlot0=**A**, MemorySlot1=**B**, Value=op, Offset=**C**, WriteSlot0=**D**, WriteSlot1=**E**
  - _W0 = (MemorySlot0 & (2^{256} - 2^(256 - offset*8))) | (Value >> offset*8)
  - _W1 = (MemorySlot1 & (2^{256} - 1 >> offset*8) | (Value << 256 - offset*8)
  - _W0 === WriteSlot0, _W1 === WriteSlot1

- **MEM_ALIGN_WR8**: memory align write 8 bit
  - MemorySlot0=**A**, Value=op, Offset=**C**, WriteSlot0=**D**
  - _W0 = (MemorySlot0 & (MaskByte > offset*8)) | ((Value & 0xFF) << 8* (31 - offset))
  - _W0 === WriteSlot0

zkASM: Examples

Generic registers

A, B, C, D, E -> 8 elements of goldilock prime filed number ->32 bits -> 256 bits
A = A7, A6, A5, A4, A3, A2, A1, A0

Registry assignments:

- 5 => A // A7-A1 = [0], A0 = [5]

- A => C // A == C

- 4294967295 (2**32 - 1) => B // B7-B1 = [0], B0 = [0xffffffff]
- A + B => B // B7-B1 = [0], B0 = [0x100000004] ❗ OVERFLOW

ROTL_C

C = 0x 9146dd22 7e54eb92 21ee4527 2735a8ae 52de694a f5749e7a b473b3f1 1328c778
; [7], [6], [5], [4], [3], [2], [1], [0]

ROTL_C = 0x 7e54eb92 21ee4527 2735a8ae 52de694a f5749e7a b473b3f1 1328c778 9146dd22
; [6], [5], [4], [3], [2], [1], [0], [7]

JMPN - JMPZ - JMPNZ

JMPN: jump if op[0] is negative

```
A           :JMPN(jumpIfNeg) // jump to jumpIfNeg if A[0] is neg (more than 32 bits)
```

JMPZ: jump if op[0] was zero

```
A           :JMPZ(jumpIfZero) // jump to jumpIfZero if A[0] is zero
```

JMPNZ: jump if op[0] was different of zero

```
A           :JMPNZ(jumpIfNoZero) // jump to jumpIfNoZero if A[0] is not zero
```

JMPC: jump if carry bit, only use with binary operations

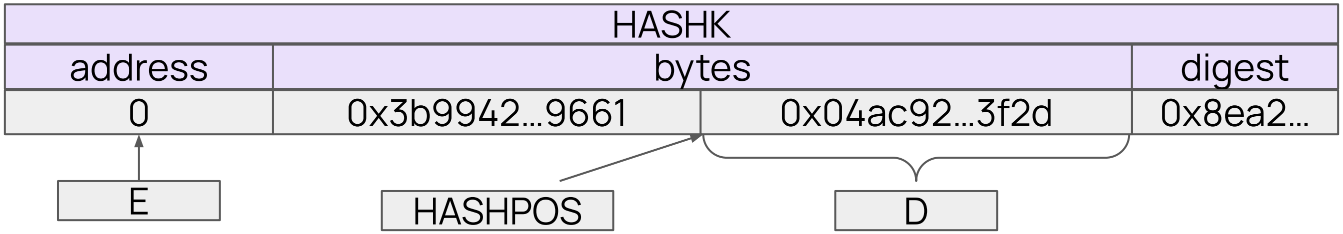
JMPNC: jump if no carry bit, only use with binary operations

```
$           :LT, JMPC(carry, noCarry) // jump to carry if A is LT B, else jump no noCarry
```

zkASM: Examples

HASHK - HASHKLEN - HASHKDIGEST

- **HASHK**: add bytes to linear poseidon. Dependant on **HASHPOS**, **D** registers
  - **HASHPOS** is the offset, **D** is the size
  - hashK[**E**].data[**HASHPOS**...**HASHPOS**+**D**-1] = op[0...**D**-1]
  - **HASHPOS** := **HASHPOS** + **D**
- **HASHKLEN**: close hash with length **HASHPOS**
- **HASHKDIGEST**: get the digest




```
        32 => HASHPOS
        32 => D
        A                           :HASHK(E)
```

zkASM: Examples

ASSERT

- Breaks verification if
  - **A** !== **op**
  - Code example:

```
1 => A
2                    :ASSERT
```

Free Input

- Loads any value into a register
  - Value is not verified. It is freely chosen by the prover
  - Support basic syntax operations: **+**, **-**, ***** , **&**, **||**, ...

```
${B & 0x00000001}                    :JMPZ(hashLeft0)
```

zkASM: zkCounters

```
; COUNTERS
CONST %MIN_STEPS_FINISH_BATCH = 200 ; min steps to finish tx
CONST %TOTAL_STEPS_LIMIT = 2**23

CONST %MAX_CNT_STEPS_LIMIT = %TOTAL_STEPS_LIMIT - %MIN_STEPS_FINISH_BATCH
CONST %MAX_CNT_ARITH_LIMIT = %TOTAL_STEPS_LIMIT / 32
CONST %MAX_CNT_BINARY_LIMIT = %TOTAL_STEPS_LIMIT / 16
CONST %MAX_CNT_MEM_ALIGN_LIMIT = %TOTAL_STEPS_LIMIT / 32
CONST %MAX_CNT_KECCAK_F_LIMIT = (%TOTAL_STEPS_LIMIT / 155286) * 44
CONST %MAX_CNT_PADDING_PG_LIMIT = (%TOTAL_STEPS_LIMIT / 56)
CONST %MAX_CNT_POSEIDON_G_LIMIT = (%TOTAL_STEPS_LIMIT / 30)
CONST %MAX_CNT_SHA256_F_LIMIT = ((%TOTAL_STEPS_LIMIT - 1) / 31488) * 7

CONST %SAFE_RANGE = 20 ; safe guard counters to not take into account (%RANGE = 1 / SAFE_RANGE)

CONST %MAX_CNT_STEPS = %MAX_CNT_STEPS_LIMIT - (%MAX_CNT_STEPS_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_ARITH = %MAX_CNT_ARITH_LIMIT - (%MAX_CNT_ARITH_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_BINARY = %MAX_CNT_BINARY_LIMIT - (%MAX_CNT_BINARY_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_MEM_ALIGN = %MAX_CNT_MEM_ALIGN_LIMIT - (%MAX_CNT_MEM_ALIGN_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_KECCAK_F = %MAX_CNT_KECCAK_F_LIMIT - (%MAX_CNT_KECCAK_F_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_PADDING_PG = %MAX_CNT_PADDING_PG_LIMIT - (%MAX_CNT_PADDING_PG_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_POSEIDON_G = %MAX_CNT_POSEIDON_G_LIMIT - (%MAX_CNT_POSEIDON_G_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_SHA256_F = %MAX_CNT_SHA256_F_LIMIT - (%MAX_CNT_SHA256_F_LIMIT / %SAFE_RANGE)
CONST %MAX_CNT_POSEIDON_SLOAD_SSTORE = 518
```

zkASM: zkCounters

Batch

Steps	Arith	Binary	Memory Align	Keccak	Poseidon Padding	Poseidon	Sha256
7968988	249037	498074	249037	2258	142307	265639	1769

2

zkASM optimizations

zkASM Optimizations: SAVE, RESTORE

- Feature to save and restore registers. The registers involved are B, C, D, E, RR, RCX, and RID. Also, the **op** was saved
- Useful when a routine uses instructions that must use some registers, so it is needed to save/recover those registers. Save steps

```
VAR GLOBAL tmpB
VAR GLOBAL tmpC
VAR GLOBAL tmpD
VAR GLOBAL tmpE
function:
B                     :MSTORE(tmpB)
C                     :MSTORE(tmpC)
D                     :MSTORE(tmpD)
E                     :MSTORE(tmpE)
;; do some function
$ => B                :MLOAD(tmpB)
$ => C                :MLOAD(tmpC)
$ => D                :MLOAD(tmpD)
$ => E                :MLOAD(tmpE)
                      :RETURN
```

```
function:
                :SAVE(B, C, D, E, RR, RCX)
;; do some function
                :RESTORE
                :RETURN
```

zkASM Optimizations: Assume Free Input

With the internal **assumeFree** flag, we indicate to the processor that it has to verify the FREE INPUT, not the **op**. This behavior could be used with memory-style instructions like **MLOAD** and **HASHx**. In assembly, to indicate it, we use the prefix **F_** on instructions, like **F_MLOAD**, **F_HASHP**, etc. For example:

```
; increment value of B with contents of var1

; using MLOAD (worse case) - 4 steps
C               :MSTORE(tmpC)
$ => C          :MLOAD(var1)
B + C => B
$ => C          :MLOAD(tmpC)

; using F_MLOAD - 1 step
$ + B => B    :F_MLOAD(var1)
```

zkASM Optimizations: MEM_ALIGN with variable length

- One previous limitation of MEM_ALIGN was that it only worked with 32-byte values, and it wasn't useful to read bytes over 256-bit memory.

- These extra features are right/left alignment and little/big (by default right-alignment and big-endian).

- To avoid using more registers, all these parameters are "coded"
  - mode = offset_bytes (0-64)
  - + 128 * length (0-32, 0 for compatibility is equivalent to 32)
  - + 8192 * left_alignment (0-1)
  - + 16384 * little_endian (0-1)

zkASM Optimizations: MEM_ALIGN with variable length

To read from memory, we need to read two consecutive addresses (M0, M1) and store them on registers A and B, respectively. Use C to store mode and OP to obtain bytes read.

```
; MEM_ALIGN_RD
;
;         <--------------- 32 bytes ---------------> <--------------- 32 bytes --------------->
;        +-------------------------------------------+-------------------------------------------+
; M0 (A) | M0(A)                  ¦                  | M1 (B)¦                                   |
;        +-------------------------------------------+-------------------------------------------+
;                                 |  |  |  |  |      |   |
;                                 ▼  ▼  ▼  ▼  ▼      ▼   ▼
;        <-- offset(bytes) -->   +-------------------------+
;                                 |V(OP)                    |
;                                 +-------------------------+
;                                  <-- length (bytes) -->
```

zkASM Optimizations: MEM_ALIGN with variable length

Another parameter of mem_align is its alignment, which defines how bytes read are stored inside the 32-byte register. If the length is 32, the left and right alignments are the same.

```
;
;                          <------------- 32 bytes ------------>
;                          -------------------------------------
; RIGHT ALIGNMENT          | 000...000 | V(OP)                 |
; (default)                -------------------------------------
;                                        <--- length (bytes) --->
;
;                          <------------- 32 bytes ------------>
;                          -------------------------------------
; LEFT ALIGNMENT           | V(OP)                 | 000...000 |
;                          -------------------------------------
;                          <--- length (bytes) --->
;
```

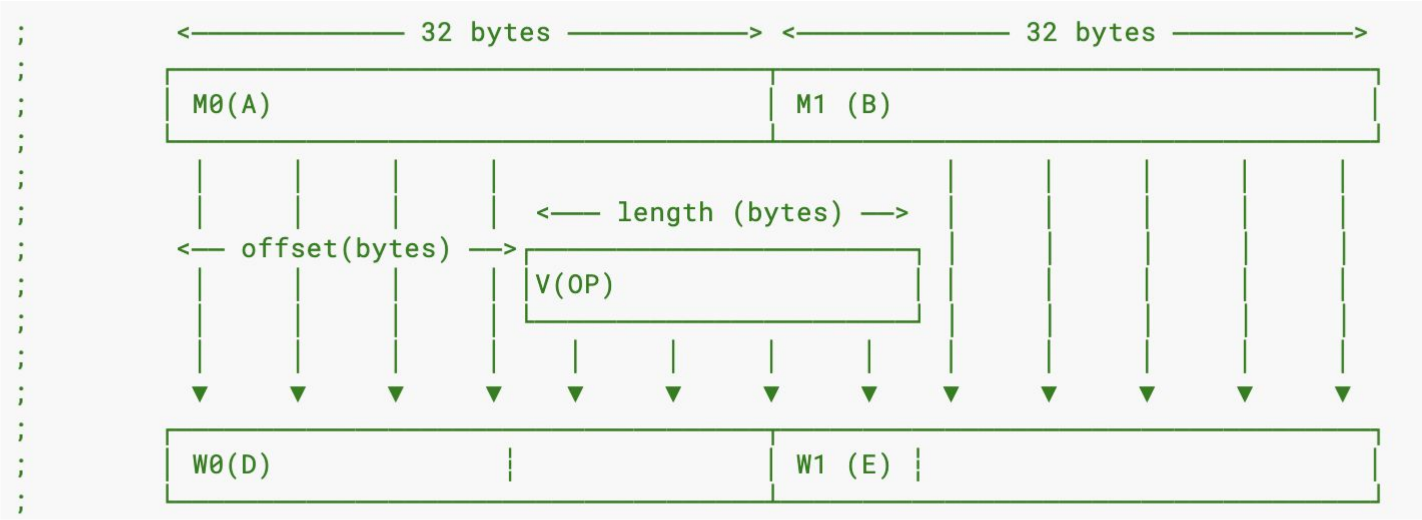
zkASM Optimizations: MEM_ALIGN with variable length

The last parameter of mem_align is its endian type, which defines how the ordered bytes are read. Big endian takes bytes as they are in memory, while little endian reverses them.

```
;
;          <------------- 32 bytes -------------> <------------- 32 bytes ------------->
;         ┌────────────────────────────────────────┬────────────────────────────────────────┐
; M0 (A)  │ M0(A)                 A2   A1   A0   │ B31  B30                       M1(B) │
;         └────────────────────────────────────────┴────────────────────────────────────────┘
;                                 |    |    |      |    |
;                                 |    |    |      |    |
;                                 ▼    ▼    ▼      ▼    ▼
;                                ┌──────────────────────────┐
;  BIG ENDIAN              V(OP) │ A2   A1   A0   B31  B30 │
;  (default)                     └──────────────────────────┘
;
;                                ┌──────────────────────────┐
;  LITTLE ENDIAN           V(OP) │ B30  B31  A0   A1   A2   A3 │
;                                └──────────────────────────┘
```

zkASM Optimizations: MEM_ALIGN with variable length

Finally, operation MEM_ALIGN_WR, in this case, verifies D as W0 and E as W1, indicating how memory M0 and M1 were modified after "writing" n bytes of OP with offset. The extra parameters alignment and endian define how to take this byte from V (OP).



zkASM Optimizations: MEM_ALIGN with variable length

```
; #3168 w=[101376-101407]
; ·---------·---------·---########################################
0x101112131415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2Fn => A
; ########################--·---------·---------·---------·-------
0x303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E4Fn => B
12n => C
$ => A :MEM_ALIGN_RD
0x1C1D1E1F202122232425262728292A2B2C2D2E2F303132333435363738393A3Bn :ASSERT

; #3172 w=[101504-101535]
0x101112131415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2Fn => A
; ·---------·---------·---########################################
0x101112131415161718191A1B0E1E2E3E4E5E6E7E8E9EAEBECEDEEEEF0F1F2F3Fn => D
12n => C
0x303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E4Fn => B
; ########################--·---------·---------·---------·-------
0x4F5F6F7F8F9FAFBFCFDFEFFF3C3D3E3F404142434445464748494A4B4C4D4E4Fn => E
0x0E1E2E3E4E5E6E7E8E9EAEBECEDEEEEF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFFFn :MEM_ALIGN_WR
```

zkASM Optimizations: ARITH_MOD (modular arithmetic)

- Given registers **A,B,C,D,op** it checks whether the linear operation **A*B + C** is equal to **op** modulo **D**
- Useful when doing modulo operations (ex: ecrecover, mulmod)

$$A \cdot B + C = \mathrm{op} \pmod{D}$$

```
; Compute A%D and return it on register C
checkRem:
        ; A·1 + 0 = op (mod D)
        1 => B
        0 => C
        ${A % D} => E       :ARITH_MOD
        E => C              :RETURN
```

3

Code: opPUSH

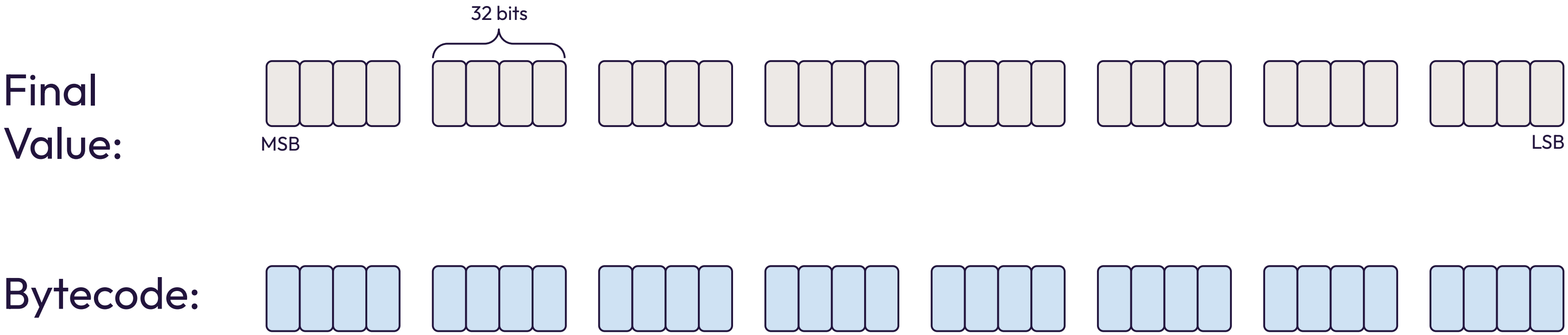
opPUSH: Diagram

- Bytecode is loaded at the very beginning of the transaction and it must match the **hashBytecode** in the state-tree

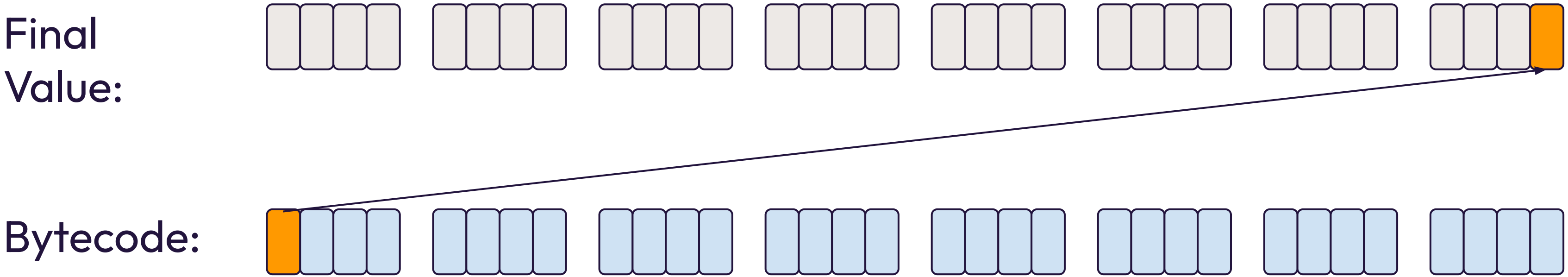
```
        ; get hash contract
        $ => A                          :MLOAD(txDestAddr)
        %SMT_KEY_SC_CODE => B
        $ => A                          :SLOAD

        ; get a new hashPId
        $ => E                          :MLOAD(nextHashPId)
        E                               :MSTORE(contractHashId)
        E+1                             :MSTORE(nextHashPId)

        ; load contract bytecode
        A                               :HASHPDIGEST(E)
```

- Limitations:
    - Bytes can be read as many times as it needs but it must be always in the same way (offset & size)
    - Always reads bytes 1 by 1

opPUSH: Diagram

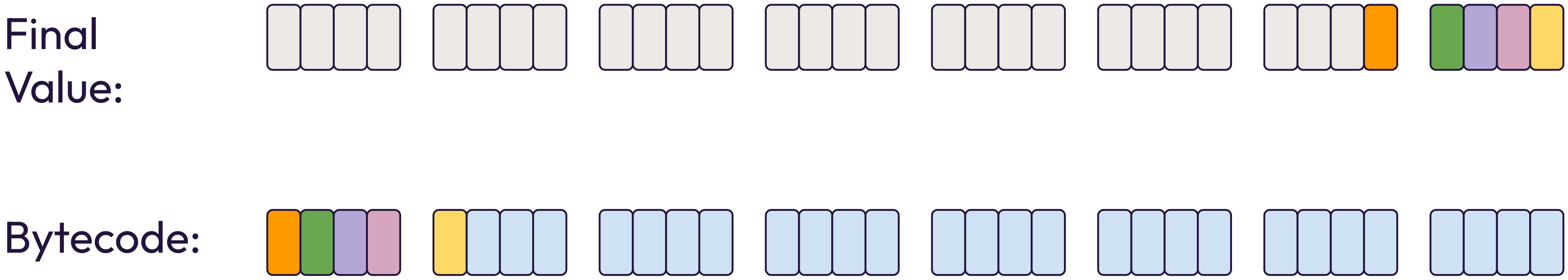


1 byte



opPUSH: Diagram

5 byte



opPUSH: Tests

- Load an array of bytes as if it is a bytecode:

```
; add bytes one by one
0x0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F20n => A     :CALL(initLoop)
0x2122232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F40n => A     :CALL(initLoop)
0x4142434445464748494A4B4C4D4E4F505152535455565758595A5B5C5D5E5F60n => A     :CALL(initLoop)
```

0x010203...	...	...	...	...	...	...	...5D5E5F60

- Perform multiple PUSH opcodes with variable length:

```
;; Start test PUSH1
; PUSH1 reading 1 byte at position 0 in the bytecode
0 => PC
1 => D,E                    :CALL(readPush)
0x01 => A
E                         :ASSERT
```

opPUSH: **Tests**

0x010203... ... ...5D5E5F60

```
;; Start test PUSH3
; PUSH3 reading 3 byte at position 0 in the bytecode
0 => PC
3 => D,E                        :CALL(readPush)
0x010203 => A
E                               :ASSERT
```

```
;; Start test PUSH32
; PUSH32 reading 32 byte at position 42 in the bytecode
42 => PC
32 => D,E                        :CALL(readPush)
0x2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748494An => A
E                               :ASSERT

0 => A,B,C,D,E,CTX, SP, PC, GAS,  SR, HASHPOS, RR ; Set all registers to 0
                                                        :JMP(finalizeExecution)
```

Performance comparison

OLD

```
{
  cntArith: 0n,
  cntBinary: 278n,
  cntKeccakF: 0n,
  cntSha256F: 0n,
  cntMemAlign: 0n,
  cntPoseidonG: 2n,
  cntPaddingPG: 2n,
  cntSteps: 6417
}
```

→

NEW

```
{
  cntArith: 0n,
  cntBinary: 1n,
  cntKeccakF: 0n,
  cntSha256F: 0n,
  cntMemAlign: 0n,
  cntPoseidonG: 2n,
  cntPaddingPG: 2n,
  cntSteps: 1068
}
```

3

Code: mloadX/mstoreX

Example before optimization:

bytesToStore:
0x00000000**FF**00000000000000000000000000000000000000000000000000000
Offset = 4 bytes
Current memory:
0xAAAAAAAA**AA**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

1- Shift left (32 - offset - length) + shift right
0x000000000**00**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2- Shift right (32 - offset - length) + shift left
0xAAAAAAAA**00**0000000000000000000000000000000000000000000000000000000
3- SUM
0xAAAAAAAA**FF**AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
4- MemAlign

Before optimization		After optimization
cntArith: 7		**cntArith: 0**
cntBinary: 12		**cntBinary: 0**
cntKeccakF: 0		cntKeccakF: 0
cntSha256F: 0		cntSha256F: 0
cntMemAlign: 0	→	cntMemAlign: 1
cntPoseidonG: 0		cntPoseidonG: 0
cntPaddingPG: 0		cntPaddingPG: 0
cntSteps: 240		**cntSteps: 25**

Thank you!

Carlos Matallana
Protocol and Integration Team, Polygon zkEVM
carlos@polygon.technology

@KrlsMata

Ignasi Ramos
Protocol and Integration Team, Polygon zkEVM
ignasi@polygon.technology

@0xIgnasi

polygon zkEVM