polygon zkEVM

Technical Document

Validating an EVM Opcode Example: MulMod

v.0.1

February 21, 2023

Contents

1 MulMod Opcode: Introduction

The `MULMOD` opcode performs the modular multiplication operation $r = a \cdot b \mod n$, where a, b, n, and r are 256-bit integers (that is, $a, b, c \in \{0, \dots, 2^{256} - 1\}$). The result r is the remainder obtained when the product of a and b is divided by n. This operation can also be expressed as $a \cdot b = k \cdot n + r$, where k is the quotient obtained when $a \cdot b$ is divided by n, and r is the remainder. In this context, r is in the range $0 \leq r < n$. For example, if $a = 11$, $b = 2$ and $n = 6$, then the result is $r = 4 = 11 \cdot 2 \mod 6$. The condition $r < n$ guarantees that, for a given (a, b, n), the tuple (k, r) is unique. In our example, $(a, b, n) = (11, 2, 6) \to (3, 4)$: $11 \cdot 2 = 3 \cdot 6 + 4$.

In our zkEVM architecture, we have a state machine called **Arith** that can verify 256-bit arithmetic operations. In particular, a combination of a sum and a multiplication. In more detail, providing the tuple $(x_1, y_1, x_2, y_2, y_3)$ **Arith** can verify that the tuple fulfills the following expression:

$$\begin{aligned} &x_1 \cdot y_1 + x_2 = y_2 \cdot 2^{256} + y_3 \\ &\text{with } x_1, y_1, x_2, y_2, y_3 \in \{0, \dots, 2^{256} - 1\} \end{aligned}$$

Thus, to implement the `MULMOD` opcode using **Arith**, we have to express the modular multiplication as a set of equations with the previous form.

2 MulMod Opcode via zkEVM

2.1 Reducing MULMOD to 256-bit arithmetic

Reducing its implementation to 256-bit arithmetic means that the multiplication and modulus operations will be performed using 256-bit integers instead of larger ones. This can be achieved by splitting the larger integers into 256-bit chunks and performing the operations on these smaller chunks. However, care must be taken to ensure that the final result is correct, as using smaller integers can lead to overflow or rounding errors. Additionally, this implementation will take more machine clocks due to the increased number of operations needed to perform the computation.

Recall that we want to check that a given tuple (a, b, k, n, r) fulfills the following constraints:

$$\begin{aligned} &a \cdot b = k \cdot n + r \\ &\text{where } a, b, n, r < 2^{256}. \end{aligned}$$

Notice that k can less than, equal to or bigger than 2^{256}. But, in any case, $k < 2^{512}$ because $a, b < 2^{256}$. In order to be able to do the check using 256-bits arithmetic, we split k in its unique limbs $k_l, k_h \in \{0, \dots, 2^{256} - 1\}$ of 256 bits. That is, k_l and k_h are the unique integers satisfying:

$$\begin{aligned} &k = k_h \cdot 2^{256} + k_l, \\ &\text{where } a, b \in \{0, \dots, 2^{256} - 1\}. \end{aligned}$$

Therefore, after replacing the new expression for k in the former constraints, we obtain the subsequent ones:

$$\begin{aligned} &a \cdot b = (k_h \cdot 2^{256} + k_l) \cdot n + r \iff \\ &a \cdot b = (k_h \cdot n) \cdot 2^{256} + (k_l \cdot n + r) \\ &\text{where } k_h \cdot n < 2^{256}. \end{aligned}$$

However, note that the expression $k_l \cdot n + r$ can be less than, equal to or bigger than 2^{256}. Henceforth, we express $k_l \cdot n + r$ with its unique 256-bit limbs $d_1, e \in \{0, \dots, 2^{256} - 1\}$:

$$k_l \cdot n + r = d_1 \cdot 2^{256} + e.$$

Replacing in our previous equation:

$$\begin{gathered} a \cdot b = (k_h \cdot n + d_1) \cdot 2^{256} + e \\ \text{where } k_h \cdot n + d_1, e < 2^{256}. \end{gathered}$$

On the other hand:

$$\begin{gathered} a \cdot b = d \cdot 2^{256} + e \\ \text{where } d, e < 2^{256}. \end{gathered}$$

And we had:

$$\begin{gathered} a \cdot b = (k_h \cdot n + d_1) \cdot 2^{256} + e \\ \text{where } k_h \cdot n + d_1, e < 2^{256}. \end{gathered}$$

Therefore:

$$d = k_h \cdot n + d_1.$$

2.2 MULMOD Implementation Summary

If we provide a tuple $(a, b, n, d, e, k_h, k_l, d_1, r)$ that fulfills the following equations:

$$\begin{gathered} a \cdot b + 0 = d \cdot 2^{256} + e, \\ k_l \cdot n + r = d_1 \cdot 2^{256} + e, \\ k_h \cdot n + d_1 = 0 \cdot 2^{256} + d, \\ r < n, \\ \text{where } a, b, n, d, e, k_h, k_l, d_1, r \in \{0, \dots, 2^{256}\}. \end{gathered}$$

Then,

$$r = a \cdot b \mod n.$$

The previous equations will be checked with the 256-bit arithmetic state machine. Finally, for optimization purposes, we need to take into account the special cases:

$$\text{If } n \in \{0, 1\} \text{ then } r = 0.$$

Similarly, we will distinguish the special case in which k_h is 0. Note that, in this case, $d_1 = d$ so we can reduce the checks to provide a tuple (a, b, n, d, e, k_l, r) that fulfills the following equations:

$$\begin{gathered} a \cdot b + 0 = d \cdot 2^{256} + e \\ k_l \cdot n + r = d \cdot 2^{256} + e \end{gathered}$$

3 zkEVM Specificity

3.1 Memory SM

In first place, the Polygon zkEVM architecture incorporates a memory SM that can check memory operations (reads and writes). The memory SM plays a critical role in this process because it ensures that memory operations, such as reads and writes, are performed correctly and consistently. By checking the memory operations in a zero-knowledge context, the memory SM can guarantee that the computation was performed correctly without revealing any of the private data.

The memory operates with addresses of 32 bits and words of 32 Bytes. This provides a maximum total size of $2^{32} \cdot 32$ Bytes = 128 Gigabytes. Transactions can have different memory contexts, each limited to 64 MB (which is a memory expansion of 8.5M gas).

The two instructions for using the memory SM are `MLOAD(address)` and `MSTORE(address)`. In the assembly, we can use memory labels to store the value of a registry in the address pointed by the label. The concrete addresses for memory labels are defined in a configuration file. For example,

```
A   :MSTORE(arithA)
```

stores the value of registry `A` at the address pointed by the memory label `arithA`.

3.2 `STACK`

Since the EVM is a stack-based virtual machine, we reserve an address space to create a stack within the memory of the zkEVM. The classical pointer called `STACK POINTER` (`SP`) contains the address of the **next free position on the** `STACK`. A `POP` from the `STACK` can be implemented as:

```
SP -1 => SP
$ => A    :MLOAD(SP)
```

where we decrement `SP` to reposition it on the last element of the stack and then we load this element into registry `A`. Similarly, a `PUSH` into the `STACK` can be implemented as:

```
0   :MSTORE (SP++)
```

which saves a `0` at the top of the stack and increments `SP`. An important note about both the stack and the memory is that the stack pointer and the memory are per context.

3.3 Binary SM

The Polygon zkEVM architecture incorporates also a **Binary** SM that can check binary operations. More specifically, the binary state machine implements **additions**, **subtractions**, comparators (**less than for signed and unsigned integers** and **equality checks**) and bitwise operations (**AND**, **OR** and **XOR**). The binary uses a `carry` column for the carries of additions and subtractions, and for returning the binary results of comparisons. For example:

```
; C=1 if A < B, else C=0
$   => C     :LT
```

The previous instruction stores a `1` in registry `C` if the value in registry `A` is less than the value in registry `B` or stores a `0` in registry `C` otherwise.

3.4 Jumps in the Main SM: `JMP`, `JMPC`, `JMPN`

The main SM has three assembly instructions for jumps: `JMP`, `JMPC` and `JMPN`.

```
; Unconditionally jumps to label myLabel
       :JMP(myLabel)

; Jumps to label myLabel if A < B
$       :LT, JMPC(myLabel)

; Jumps to label myLabel if A is negative
A       :JMPN(myLabel)
```

- `JMP`: Jumps unconditionally to a label in the program.
- `JMPC`: Jumps to a label in the program if `carry` $\neq 1$. In the example, carry is 1 if registry `A` is **Less Than (LT)** registry `B`.
- `JMPN`: Jumps to a label in the program if `op0` is a negative value. In the example, the value of registry `A`. Negative values go from -1 to -2^{32}.

3.5 Arith SM

3.5.1 Arith SM: Basic Usage

We use the **Arith** SM with the `:ARITH` instruction. The **Arith** uses the `x1, y1, x2, y2` and `y3` 256-bit registries to check the following expression:

$$x_1 \cdot y_1 + x_2 = y_2 \cdot 2^{256} + y_3$$

In the **Main** SM, we use registry `A` as x_1, `B` as y_1, `C` as x_2, `D` as y_2 and `op` as y_3 to verify the arithmetic operation.

```
1 $${var _valArith = A * B + C}
2 ${_valArith >> 256} => D
3 ${_valArith} => E :ARITH
```

In the example, at line 1, we instruct the executor into computing the big number `A*B+C` from the current values of the registries `A`, `B` and `C` at the **Main** SM and store this value in an executor variable arbitrarily called `_valArith`. At line 2, the executor computes the value of `y2` as a free input by shifting 256 bits `_valArith` and stores the result in registry `D`. Finally, at line 3, the executor computes `op` (value of `y3`) as a free input, stores `op` in registry `E` and also the **Arith** SM verifies the arithmetic expression.

3.5.2 Arith SM: `CALL` with an Assembly Subroutine

Another way of using the **Arith** SM is to use an assembly subroutine. Subroutines are invoked with `CALL(label)`. When a `CALL` is performed, the execution jumps into the specified label in the program. When the label's associated code finishes, the program execution continues at the line after the initial `CALL` invocation.

```
; calling the Arith
   A          :MSTORE(arithA)
   B          :MSTORE(arithB)
              :CALL(mulARITH)

; continues here after the assembly subroutine
   A          :MLOAD(arithOverflow)
   B          :MLOAD(arithRes1)
```

In particular, the mulARITH subroutine computes pure multiplications using the **Arith** SM to verify that: $\mathtt{A} \cdot \mathtt{B} + 0 = \mathtt{D} \cdot 2^{256} + \mathtt{op}$. The mulARITH subroutine uses memory to read the operands and leave the results. In this regard, observe that before jumping into mulARITH, we provide the operands to the **Arith** SM under the memory labels arithA and arithB. When the invocation of mulArith finishes, the values D and E are stored in memory under the memory labels arithOverflow and arithRes1, respectively.

3.5.3 Arith SM: mulARITH Subroutine

```
 1  ; zkevm-rom:main/utils.zkasm
 2
 3  mulARITH: ; assembly subroutine for multiplication
 4      RR                  :MSTORE(tmpZkPC)
 5      zkPC+1 => RR        :JMP(storeTmp)
 6      $ => A              :MLOAD(arithA)
 7      $ => B              :MLOAD(arithB)
 8      0 => C
 9      $${var _mulArith = A * B}
10      ${_mulArith >> 256} => D
11      ${_mulArith} => E  :ARITH
12      E                   :MSTORE(arithRes1)
13      D                   :MSTORE(arithOverflow)
14      zkPC+1 => RR        :JMP(loadTmp)
15      $ => RR             :MLOAD(tmpZkPC)
16                          :JMP(RR)
```

The registry RR contains the code return address. The value of RR is set when invoking the subroutine. In the subroutine, at line 4, we store the current value of the registry RR at the memory label tmpZkPC. This is because in line 6 we will jump to another subroutine. In line 5, we set the value of RR to the next line for correctly coming from the tmpZkPC subroutine.

```
storeTmp:
    A                   :MSTORE(tmpVarA)
    B                   :MSTORE(tmpVarB)
    C                   :MSTORE(tmpVarC)
    D                   :MSTORE(tmpVarD)
    E                   :MSTORE(tmpVarE)
                        :JMP(RR)

loadTmp:
    $ => A                  :MLOAD(tmpVarA)
    $ => B                  :MLOAD(tmpVarB)
    $ => C                  :MLOAD(tmpVarC)
    $ => D                  :MLOAD(tmpVarD)
    $ => E                  :MLOAD(tmpVarE)
                            :JMP(RR)
```

The `storeTmp` subroutine stores the values of the registries `A, B, C, D` and `E` in temporary memory labels and then jumps back to the return address at `RR`. The `loadTmp` subroutine does the opposite.

```
1  ; zkevm-rom:main/utils.zkasm
2
3  mulARITH: ; assembly subroutine for multiplication
4      RR               :MSTORE(tmpZkPC)
5      zkPC+1 => RR     :JMP(storeTmp)
6      $ => A           :MLOAD(arithA)
7      $ => B           :MLOAD(arithB)
8      0 => C
9      $${var _mulArith = A * B}
10     ${_mulArith >> 256} => D
11     ${_mulArith} => E :ARITH
12     E                :MSTORE(arithRes1)
13     D                :MSTORE(arithOverflow)
14     zkPC+1 => RR     :JMP(loadTmp)
15     $ => RR          :MLOAD(tmpZkPC)
16                      :JMP(RR)
```

Let us return to the `multARITH` subroutine. In lines 6 and 7, the multiplicands are loaded from the associated memory labels. This subroutine is for pure multiplications, so in line 8, registry `C` is set to 0. Then, the executor is instructed to compute the proper `y2` and `y3` values. For computing these values, at line 9, the executor is instructed to compute the big number `A*B` and to store the result in an executor variable called `_mulArith`. Notice that we can specify pure executor functions inline using the syntax `$${...}`. These pure executor functions do not directly contribute to the execution trace. At line 10, `D` is computed by taking the bits 256 to 511 of `_mulArith` and stored at registry `D`. A line 11, `E` is computed by taking the bits 0 to 255 of `_mulArith`. At this line, the `E` value is at `op`, `:ARITH` checks the operation and the `E` value is stored at the registry `E`. At lines 12 and 13 the result of the operation is stored in memory: `E` in `arithRes1` and `D` in `arithOverflow`. Finally, in line 14 we restore the previous values of registries, in line 15 we read the return code address and finally jump to this address in line 16.

3.6 Manage Errors: `stackUnderflow` and `outOfGas`

When we have errors, we call a function at the executor to log the error and then, we jump to the `handleError` code label.

```
 1 ; Stack and gas checks
 2     SP - 3          :JMPN(stackUnderflow)
 3     SP - 1 => SP
 4     GAS-8 => GAS    :JMPN(outOfGas)
 5
 6 stackOverflow:
 7     ${eventLog(onError, overflow)}
 8     :JMP(handleError)
 9
10 outOfGas:
11     ${eventLog(onError, OOG)}
12     :JMP(handleError)
13
14 handleError:
15     ; revert all state changes
16     $ => SR          :MLOAD(initSR)
```

Line 2 checks that we have 3 elements at the top of the stack to do the MULMOD operation. Line 3 positions the `SP` at the address of the top element. Line 4 checks that there is enough gas remaining for doing the operation. When there's this type of error (`outOfGas`, `stackOverFlow`) the behavior is very similar to process an invalid OPcode:

- All the gas is consumed of the current Context (current call).
- We return to the last context pushing a 0 to the stack.
- Also we recover the `initSR` of the current context, meaning that all the changes will be reverted (in storage, nonce, bytecode, etc.) except for the gas consumption.
- Also the `gasRefund` will not be modified.

4 MULMOD Opcode Assembly

We are under the situation where the top of the stack looks like `STACK` $= [a, b, n, ...]$. The assembly can be found at `zkevm-rom:main/opcodes.zkasm`. It is important that a free input is just used once. If you use a free input multiple times, you should check that they are the consistent in the assembly. In our opcode, if we introduce "a" several types, assembly must check that always the same "a" is introduced (binary equal). Typically, free inputs that are used several times are those of the SM, e.g. in memory we can put the same free input multiple times but the assembly checks consistency.

```
; Stack and gas checks
SP - 3          :JMPN(stackUnderflow)
SP - 1 => SP
GAS-8 => GAS    :JMPN(outOfGas)

; Load multiplicands
$ => A          :MLOAD(SP--)    ; (A<=a,B,C,D,E)
$ => B          :MLOAD(SP--)    ; (A=a,B<=b,C,D,E)
A               :MSTORE(arithA) ; [arithA<=a]
B               :MSTORE(arithB) ; [arithA=a,arithB<=b]
${var _mulab = A * B}           ; FIX with $$

; Check: if(n is 0 or 1) {r=0}
$ => A          :MLOAD(SP)      ; (A<=n,B=b,C,D,E)
2 => B                          ; (A=n,B<=2,C,D,E)
$               :LT, JMPC(zeroOneMod)

; Check: a*b = d*2^256 + e
      :CALL(mulARITH) ; [arithA=a, arithB=b, arithOverflow<=d, arithRes1<=e
          ]

$${var _k = _mulab/A}
; Check kh == 0
${_k >> 256} => B ; (A=n,B<=kh,C,D,E) kh=k_b256-511
${cond(B == 0)} :JMPN(mulModNoKH)
```

```
; Check: kl*n+r = d1*2^256+e
${_k % (1 << 256)} => B                   ; (A=n,B<=kl,C,D,E)      op=kl=
    k_b0-255
${_mulab % A} => C                        ; (A=n,B=kl,C<=r,D,E)    op=r=A*B
    %n
${(B * A + C) >> 256} => D                ; (A=n,B=kl,C=r,D<=d1,E) op=d1=(B
    *A+C)_b256-511
$               :MLOAD(arithRes1), ARITH  ; (A=n,B=kl,C=r,D=d1,E)  op=[
    arithRes1]=e

; Check: assert r<n
A => B               ; (A=n,B<=n,C=r,D=d1,E)
C => A               ; (A<=r,B=n,C=r,D=d1,E)
$ => A          :LT
1               :ASSERT

; Check: kh*n+d1 = 0*2**256+d
${_k >> 256} => A                           ; (A<=kh,B=n,C=r,D=d1,E)  op=kh
    =k_b256-511
D => C                                      ; (A=kh,B=n,C<=d1,D=d1,E)
0 => D                                      ; (A=kh,B=n,C=d1,D<=0,E)
$               :MLOAD(arithOverflow), ARITH  ; (A=kh,B=n,C=d1,D=0,E)   op=[
    arithOverflow]=d

; PUSH r
C               :MSTORE(SP++)
                :JMP(readCode)
```

```
mulModNoKH:
  ; Check: kl*n+r = d1*2^256+e where d1=d
  ${_k} => B                                ; (A=n,B<=kl,C,D,E)     op=kl=k since kh=0
  ${_mulAB % A} => C                        ; (A=n,B=kl,C<=r,D,E)   op=r=a*b%n
  $ => D          :MLOAD(arithOverflow)     ; (A=n,B=kl,C=r,D<=d,E) op=[arithOverflow]=d
  $               :MLOAD(arithRes1), ARITH  ; (A=n,B=kl,C=r,D=d,E)  op=[arithRes1]=e

  ; Check: assert r<n
  A => B                ; (A=n,B<=n,C=r,D=d,E)
  C => A                ; (A<=r,B=n,C=r,D=d,E)
  $ => A           :LT
  1                :ASSERT

  ; PUSH r
  C                :MSTORE(SP++)
                   :JMP(readCode)

zeroOneMod:
    0              :MSTORE(SP++)
                   :JMP(readCode)
```

4.1 Load Multiplicands

The first step of the MULMOD implementation is to POP the A and B values and load them into the proper memory labels:

```
1 ; Load multiplicands
2 $ => A          :MLOAD(SP--)    ; (A<=a,B,C,D,E)
3 $ => B          :MLOAD(SP--)    ; (A=a,B<=b,C,D,E)
4 A               :MSTORE(arithA) ; [arithA<=a]
5 B               :MSTORE(arithB) ; [arithA=a,arithB<=b]
6 ${var _mulab = A * B}           ; FIX with $$
```

Line 2 loads multiplicand a and stores it in the registry A. This is denoted in comments as (Registry=Value, ...). Line 3 does the same for multiplicand b. Line 4 stores the value in registry A into memory at label arithA. This is denoted in comments as [mem_label=Value, ...]. Line 5 stores the value in registry B into memory at label arithB. Finally, at line 6, we instruct the executor to compute the big number $A \cdot B$ and store the result in the executor variable called _mulab.

Note. Ideally $$ should be used at line 6 because it is a pure executor function but currently, $ has to be used which uses a row of the execution trace (this might be fixed in the future).

4.2 Special Cases Check

Next, we check if $n \in \{0, 1\}$.

```
  ; Check: if(n is 0 or 1) {r=0}
  $ => A          :MLOAD(SP)      ; (A<=n,B=b,C,D,E)
  2 => B                          ; (A=n,B<=2,C,D,E)
  $               :LT, JMPC(zeroOneMod)

zeroOneMod:
    0                :MSTORE(SP++)
                     :JMP(readCode)
```

The LT is an operation verified by the **Binary** state machine. LT must be 1 if the value of the registry A is lower than the value of the registry B and, 0 otherwise. The result of LT is introduced as a free input in the execution trace of the **Main** SM and verified by the **Binary** SM. JMPC jumps if the carry of the binary operation is 1. In this code, if A < B the executor introduces carry=1 as the free input and, JMPC jumps to zeroOneMod. At zeroOneMod, we store a 0 at the top of the stack as the result of the MULMOD operation and, finally, we jump to read the next opcode.

4.3 $a \cdot b$ Decomposition in 256-bit Limbs

```
  ; Check: a*b = d*2^256 + e
        :CALL(mulARITH) ; [arithA=a, arithB=b, arithOverflow<=d, arithRes1<=e
            ]
```

If n is not a special case, we do not jump but we compute the decomposition of $a \cdot b$:

$$a \cdot b = d \cdot 2^{256} + e$$

Recall that the values a and b are stored at memory, at the memory labels arithA and arithB. The decomposition values d and e, are also stored at memory after calling the subroutine, in particular, at the memory labels arithOverflow and arithRes1, respectively.

4.4 Check if $k_h = 0$

Next, we compute k_h and store it in the register B:

```
  $${var _k = _mulab/A}
  ; Check kh == 0
  ${_k >> 256} => B ; (A=n,B<=kh,C,D,E) kh=k_b256-511
  ${cond(B == 0)} :JMPN(mulModNoKH)
```

Observe that we treat the case where $k_h = 0$ separately because this case needs one check less. If $k_h = 0$ we jump into the code label mulModNoKH. Notice that we jump using a condition checked by the executor: cond(B==0) and introduced as a free input. This is a correct way of implementing the verification because the **Arith** SM will check all the expressions that enforce that r is the correct result of the MULMOD operation.

4.5 Case where $K_h = 0$

```
1  mulModNoKH:
2    ; Check: kl*n+r = d1*2^256+e where d1=d
3    ${_k} => B                              ; (A=n,B<=kl,C,D,E)     op=kl=k since kh=0
4    ${_mulAB % A} => C                      ; (A=n,B=kl,C<=r,D,E)   op=r=a*b%n
5    $ => D         :MLOAD(arithOverflow)    ; (A=n,B=kl,C=r,D<=d,E) op=[arithOverflow]=d
6    $              :MLOAD(arithRes1), ARITH ; (A=n,B=kl,C=r,D=d,E)  op=[arithRes1]=e
7
8    ; Check: assert r<n
9    A => B                ; (A=n,B<=n,C=r,D=d,E)
10   C => A                ; (A<=r,B=n,C=r,D=d,E)
11   $ => A          :LT
12   1               :ASSERT
13
14   ; PUSH r
15   C               :MSTORE(SP++)
16                   :JMP(readCode)
```

Recall that we already checked the decomposition of $a \cdot b$ into 256-bits limbs. When $k_h = 0$, we just need to check the following constraints:

$$k_l \cdot n + r = d \cdot 2^{256} + e$$
$$r < n$$

Line 4 loads r in `C`. Line 5 loads the previously computed d. Line 6 loads the previously computed e into `op` and triggers the **Arith** check using the `:ARITH` instruction. Lines 9, 10, 11 and 12 check $r < n$. If r is not less than n, we abort the execution using the `ASSERT` instruction. The ASSERT instruction makes sure that the value at the `A` register is the same as the value of `op`. If this is not fulfilled, the proof cannot be generated. The following constraint, then, its introduced into the PIL of the **Main** state machine:

```
(A-op) * assert = 0
```

Note: The only way that an assert fails is because the executor has tried to manipulate the free inputs and they do not fulfill the assert condition. If everything is OK, we push the result to the top of the stack and read the next opcode (lines 15 and 16).

4.6 Case where $K_h \neq 0$

```
1  ; Check: kl*n+r = d1*2^256+e
2  ${_k % (2 << 256)} => B                ; (A=n,B<=kl,C,D,E)      op=kl=
       k_b0-255
3  ${_mulab % A} => C                     ; (A=n,B=kl,C<=r,D,E)    op=r=A*B
       %n
4  ${(B * A + C) >> 256} => D             ; (A=n,B=kl,C=r,D<=d1,E) op=d1=(B
       *A+C)_b256-511
5  $              :MLOAD(arithRes1), ARITH ; (A=n,B=kl,C=r,D=d1,E)  op=[
       arithRes1]=e
6
7  ; Check: assert r<n
8  A => B              ; (A=n,B<=n,C=r,D=d1,E)
9  C => A              ; (A<=r,B=n,C=r,D=d1,E)
10 $ => A         :LT
11 1              :ASSERT
12
13 ; Check: kh*n+d1 = 0*2**256+d
14 ${_k >> 256} => A                        ; (A<=kh,B=n,C=r,D=d1,E)  op=kh
       =k_b256-511
15 D => C                                   ; (A=kh,B=n,C<=d1,D=d1,E)
16 0 => D                                   ; (A=kh,B=n,C=d1,D<=0,E)
17 $             :MLOAD(arithOverflow), ARITH ; (A=kh,B=n,C=d1,D=0,E)   op=[
       arithOverflow]=d
18
19 ; PUSH r
20 C              :MSTORE(SP++)
21                :JMP(readCode)
```

In this case, all the expressions below have to be verified:

$$k_l \cdot n + r = d_1 \cdot 2^{256} + e$$
$$k_h \cdot n + d_1 = 0 \cdot 2^{256} + d$$
$$r < n$$

Lines 13 to 21 do the same as for the case $K_h = 0$. Lines 1 to 5 check that $k_l \cdot n + r = d_1 \cdot 2^{256} + e$. Lines 7 to 11 check that $r < n$. Previously computed values in memory are used with free inputs. Line 2 instructs the executor to compute k_l and introduces this value as a free input that is copied to registry `B`. k_l is computed as the 256 least significant bits of the big number $a \cdot b/n$. Line 3 instructs the executor to compute n and introduce this value as a free input that is copied to `C`. Line 4 does something similar for the d_1 value, which is copied to `D`. Since $k_h \neq 0$, line 7 starts the verification that $r < n$. For the check, we use d previously stored at the memory label `arithOverflow`. Value k_h is computed as the bits from 256 to 511 of k. Finally, observe that k_h is introduced as a free input and copied to `B`.

5 Counters and Batch Errors

Counters are a way to control that the total number of steps do not exceed the maximum polynomial size. Here, it is state that the maximum number of steps of the main we can take for the opcode is 150:

```
%MAX_CNT_STEPS - STEP - 150 :JMPN(outOfCounters)
```

We can invoke a maximum of 2 times the **Binary** SM (two `:LT`) and 3 times the **Arith** SM (three `:ARITH` instructions). Hence, we will add the following code at the beginning of the opcode

```
%MAX_CNT_ARITH - CNT_ARITH - 3 :JMPN(outOfCounters)
%MAX_CNT_BINARY - CNT_BINARY - 2 :JMPN(outOfCounters)
```

Here `%MAX_CNT_ARITH` and `%MAX_CNT_BINARY` are constants that define the maximum steps can take each of these state machines. If we go out of counters, then, there is an error in the processing of the batch (it is not an error of the user). We log this and restore the SR to the state root previous to the batch:

```
outOfCounters:
    ${eventLog(onError, OOC)}
                    :JMP(handleBatchError)

handleBatchError:
    $ => SR         :MLOAD(batchSR)
    ${eventLog(onFinishTx)}
    ${eventLog(onFinishBatch)}
                    :JMP(processTxsEnd)
```

6 Running Integration Tests

6.1 Running Integration Tests: Repositories

We are going perform integration tests with the prover for a contract that uses the `MULMOD` Opcode. First, download the needed repositories:

```
$ git clone git@github.com:0xPolygonHermez/zkevm-rom.git
$ git clone git@github.com:0xPolygonHermez/zkevm-proverjs.git
$ git clone git@github.com:0xPolygonHermez/zkevm-testvectors.git
```

Then, in each repository, install the dependencies with `npm install`. Next, we are going to create a smart contract that uses the desired Opcode(s), in this case, the `MULMOD` Opcode.

6.2 Running Integration Tests: Smart Contract

In the `zkevm-testvectors` repository, go to `tools-calldata/evm/contracts`. There, create a new contract with whatever you want to test. In this case, we create a new file called `OpMulMod` with the following content:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.7;

contract OpMuldMod {
  function Mulmod( uint256 a, uint256 b, uint256 n ) public view returns (uint256 result) {
    assembly {
      result := mulmod(a, b, n)
    }
  }
}
```

6.3 Running Integration Tests: Test Input Vectors

In the `zkevm-testvectors` repository, go to the directory `tools-calldata/evm/generate-test-vectors`. The easiest way to proceed is just copy one of the files and use it as template:

- Copy one of the files to a new file called `gen-test-mulmod.json`.
- Then, just edit the `"txs"` and `"contracts"` properties of the JSON file. Any time you want to do a modification, e.g. changing the inputs, you just need to modify these two properties (the rest is updated automatically).

```
    [
    { "contracts": [{"contractName": "OpMuldMod", "paramsDeploy": {} }]},
    "txs": [{
        "from": "0x4d5Cf5032B2a844602278b01199ED191A86c93ff",
        "to": "contract",
        "nonce": "0",
        "value": "0",
        "contractName": "OpMuldMod",
        "function": "MulMod",
        "params": [ 256, 100000, 256 ],
        "gasLimit": 100000,
        "gasPrice": "1000000000",
        "chainId": 1000
    }],
}
]
```

- Then, execute the following command for generating the inputs for the executor:

```
zkevm-testvectors/tools-calldata/evm$ ./gen-input.sh gen-test-mulmod.json
```

  Note: the `gen-input.sh` script will search for the file in `tools-calldata/evm/generate-test-vectors/`. The previous command creates two files:

  `zkevm-testvectors/state-transition/calldata/test-mulmod.json`

  This file contains data necessary for creating the inputs for the executor. You don't have to do anything with this file. It is useful only if you want to do executor debugging.

  `zkevm-testvectors/inputs-executor/calldata/test-mulmod_0.json`

  This is the actual executor input. The 0 in the name of the file is because we only add one test case in our example. If we add several tests then, multiple inputs (with more sequential numbers) would be created.

6.4 Running Integration Tests: Execution Trace

Now we can create the execution trace with the input. To do so, go to the `zkevm-proverjs` repository and in project root, execute the following command:

```
$ node --max-old-space-size=4096 src/main_executor.js /path/to/test-mulmod_0.json -r /path/to/zkevm-rom/build/rom.json -d -s
```

When the command finishes, you can take a look at the traces at:
`zkevm-proverjs/src/sm/sm_main/logs-full-trace/test-mulmod_0__full_trace_0.json`
Search at this file for `"opcode": "MULMOD"`. Remark that the test will check that the trace fulfills the associated PIL.