A zk-evm specification

Olivier Bégassat, Alexandre Belling, Théodore Chapuis-Chkaiban, Franklin Delehelle,
Blazej Kolad, Nicolas Liochon

October 2022

Contents

Introduction

0.1 Purpose

The present document is a revised and expanded version of a previous (partial) specification of a zk-evm.

0.2 Context and results

Rollups are a family of powerful scaling technologies which promise to considerably increase the capacity of the Ethereum Blockchain. An introduction to Rollups, zk-EVMs and their role in improving Ethereum capacity can be respectively found in [1, 2]. Multiple attempts at building scalable and practical rollup solutions have been positively received. zkSync [3], for instance, transpiles Yul into a zk-VM friendly bytecode. Cairo [4], on the other hand, uses a custom architecture adapted to an efficient STARK prover for smart contracts written in Cairo . Other projects, such as Hermez [5] or Scroll Tech [6] and this project aim to interpret native EVM bytecode, without transpilation or further compilation steps.

0.3 Conventions

Throughout the document we use a number of notational conventions which we explain here. These conventions apply to column names and are meant to clarify the origin and purpose of certain columns within a given trace. Others should be viewed as constructors which define new columns from existing ones.

Modules have three letter identifiers. The named modules are the following:

Module stamps. Module stamps count calls to a given module; most modules have a single stamp though the hub and ALU have several. Stamp columns are adorned with a □, thus the STO□ is the module stamp of the storage module. Module stamps are typically computed/updated in the hub module whose main purpose is to dispatch (paid for an otherwise valid) instructions to the module(s) that are equipped to carry them out. Associating a unique identifier (i.e. stamp) to such "module-calls" is crucial when the order of operations matters. This is the case for instructions pertaining to (address) warmth (i.e. the WRM module), required gas computations (i.e. GAS), RAM (i.e. MMU and RAM), the stack (i.e. HUB), storage (i.e. STO), ... to cite a few. Stateless modules such as the modules handling arithmetic (i.e. the ALU module), binary (i.e. BIN) or word comparison (i.e. WCP) opcodes don't *require* a time stamp *per se* yet are given one nonetheless. (e.g. the address warmth module corresponds to the three letter identifier WRM); the corresponding stamp column is that identifier followed by □ (e.g. HUB□).

Imported columns. Angular parentheses $\langle \cdots \rangle$ signal columns whose contents are **imported** from other modules by means of a lookup argument. By way of example: all modules[1] import their module stamp from the hub. Modules tasked with executing certain opcodes will typically import values from the stack (e.g. pairs of stack values $\langle {}_k\mathsf{VAL}^{\mathsf{hi}}\rangle, \langle {}_k\mathsf{VAL}^{\mathsf{lo}}\rangle$, for various $k \in \{1, 2, 3, 4\}$.) Many modules also imports values that aren't borrowed from the stack. E.g. the hub module imports the instruction $\langle\mathsf{INST}\rangle$ from the ROM, e.g. the GAS module imports the current, new and endowment gas values (GAS^{κ}, GAS^{ν} and $\mathsf{GAS}^{\varepsilon}$ respectively) from the hub, e.g. the OOB module imports execution context dependent data such exception flags, the size of return data $\langle\mathsf{RDS}\rangle$, the size of call data $\langle\mathsf{CDS}\rangle$ or the code size $\langle\mathsf{CODESIZE}\rangle$.

Decoded columns. A particular case of the above arises with **decoded columns**. Those are columns whose contents are extracted from a hardcoded collection of columns using a lookup argument. They are adorned with a lozenge as in $^{\lozenge}\mathsf{COL}$. By way of example: the hub contains various instruction decoded flag columns but also a $^{\lozenge}\mathsf{STACK\_PATTERN}$ column whose contents are deduced from an immutable reference table called the **instruction decoder**. Similarly the binary module imports the results of binary operations performed on pairs of bytes (and injects the relevant one into the result.)

Flag columns. Among the instruction decoded columns on finds various binary flags columns (e.g. $^{\lozenge}\mathsf{ALU}$ ⚐, $^{\lozenge}\mathsf{MMU}$ ⚐, $^{\lozenge}\mathsf{EXP}$ ⚐, $\dots$). These serve several purposes. The first is to provide an *indication* as to when modules *may* be sollicited by the hub to carry out an instruction. Thus arithmetic instructions raise the $^{\lozenge}\mathsf{ALU}$ ⚐, instructions that involve the RAM raise the $^{\lozenge}\mathsf{MMU}$ ⚐ etc $\dots$ Other flags trigger particular behaviours. For instance the PUSH ⚐ and the JUMP ⚐ trigger the expected behaviour of the program counter in the hub.

Module selector columns. When an instruction raises an instruction flag the associated module *may* get triggered. The actual trigger is usually deduced form this flag and exception flags. Such columns are tagged with a ϟ symbol

Interleaved columns. Certain arguments require us to merge columns of the same size into a single column. We use $\boxplus$ to signify the formation of such interleaved columns. E.g. starting with columns A, B and C of size n we may form the column $\mathsf{X} := \mathsf{A} \boxplus \mathsf{B} \boxplus \mathsf{C}$ defined as having length $3n$ and values

$$\begin{cases} \mathsf{X}_{3\cdot i+0} = \mathsf{A}_i \\ \mathsf{X}_{3\cdot i+1} = \mathsf{B}_i \\ \mathsf{X}_{3\cdot i+2} = \mathsf{C}_i \end{cases}$$

Row permutations. Our arithmetization requires row permutation arguments. These usually take the following form: we are given a small family of reference columns $\mathsf{REF}_1, \dots, \mathsf{REF}_p$ of equal size n (which we view as the columns of a $n \times p$ reference matrix REF). We are further given the description of an essentially unique permutation of the set $\{0, 1, \dots, n-1\}$ of rows indices, e.g. "(the essentially unique) row permutation of the matrix REF under which its rows appear lexicographically sorted". We then write $\mathsf{AUX}_j \mapsto [\mathsf{AUX}_j]^{\text{⤮}}$ for the mapping which takes an arbitrary column of the same size and applies the aforementioned row permutation to its rows.

0.4 Organization

1. ALU: ALU module; deals with opcodes performing arithmetic operations; see chapter **??**;
2. BIN: binary module; deals with opcodes performing binary operations; see chapter **??**;

[1] which are connected to the hub

3. WCP: word comparison module; deals with opcodes performing integer comparisons; see chapter 9;

4. MXP: computes memory expansion costs; may raise a flag if offsets are wildly out of bounds; see chapter 6;

5. GAS: module which performs gas checks at crucial points in time; performs the (63/64)-ths computations for `CALL`s and `CREATE`s; computes associated gas endowments; see chapter ??gas;

6. ROM: contains the bytecodes which are run and or (temporarily) deployed in a batch of transactions; see chapter 4;

7. HUB: module containing the stack and call stack; dispatches instructions to other modules; see chapter 1;

8. MMU: first stop in the life time of an opcode execution which touches RAM; performs arithmetic on offsets and various sizes to cut down execution of a single opcode into a sequence of smaller queries; see chapter 2;

9. RAM: contains the RAM of all execution context and can communicate with other data sources such as ROM and other data stores; carries out the sequence of small queries commissioned by the MMU; see chapter 3;

10. OOB: performs certain range checks required by instructions; see chapter 5;

11. STO: storage module; unique among all modules other than the hub in that it computes its own gas costs; see chapter 8;

12. ACC: address existence module; loads and udpates account data from the state; `WIP`;

13. WRM: address warmth module: loads prewarmed addresses; handles address warmth in general; built on similar principles as the storage module; see chapter ??;

The following are a few very small modules that either perform a very specific task or are used for reference for the prover

1. KEC: *two* simple modules: an INFO-module which extracts informations for whenever Keccak is executed in the zk-evm (i.e. paid for executions of `SHA3` and `CREATE2`) such as the size in bytes of the data to hash[2]; the second module serves as a data store to which to extract the message to hash;

2. LOG: same idea for logs; the information module extracts the log parameter ($\in \{09, 1, 2, 3, 4\}$), logger address and size in bytes; the second module serves as a data store for the log message;

3. EXP: computes the dynamic gas cost of the `EXP` instruction; see chapter ??;

4. SHV: shaves the leading 12 bytes off addresses; see chapter 13;

[2] The price, which depends on the *number of EVM words* rather than the *number of bytes*, is computed in the MXP

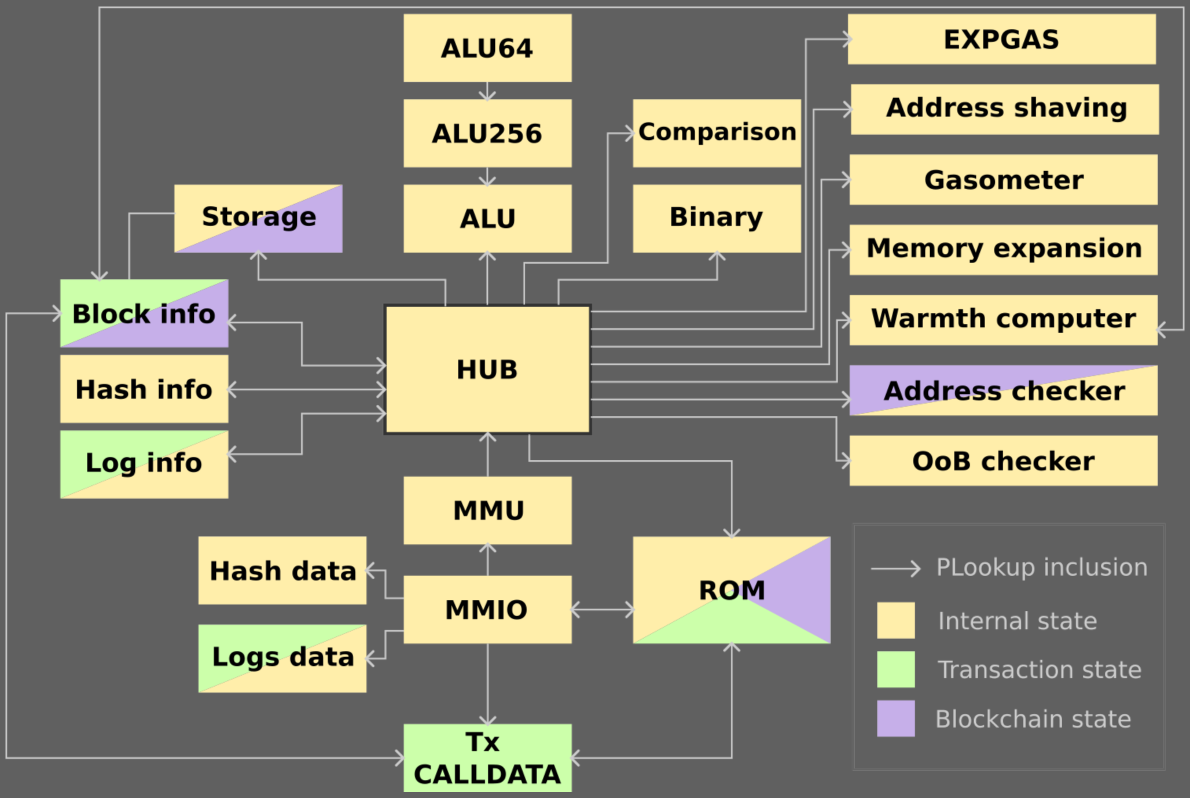

Figure 1: Modular architecture of the zk-evm. Boxes represent modules and arrows represent (plookup) inclusion proofs. If an arrow points from module ABC to module XYZ then XYZ imports a portion of its data from ABC. Arrows may be bidirectional which signals a "bilateral" inclusion proof.

0.5 Suggestions for reading this document

We suggest the reader start with the chapter on the **hub** 1. The hub is the center piece of our zk-evm design. It reads instructions from the **ROM** 4 and dispatches instructions to other modules. Various smaller modules which are directly connected to the hub (e.g. the word comparison module9 or out of bounds module 5) may prove helpful to develop some intuition for the techniques used elsewhere. After the hub, the main module of interest is certainly the RAM. In our design the RAM is split into 2 pieces: the memory management unit 2 (or offset processor) and the memory mapped input output module**??**. The **mmu** receives instructions from the hub and is tasked with breaking them down into smaller "elementary" operations. This reduction is a two phase process: the first phase ("precomputation" or "establishing" phase) extracts auxiliary data from the arguments of the opcode (offset and size parameters). The second "micro-instruction writing" phase uses these numerical parameters to build a sequence of micro-instructions (**surgeries** and **transplants**) which the **mmio** imports and carries out.

The reader should be warned: this document is a work in progress: typos — even outright mistakes — are to be expected. One module (the **address existence** module) is presently missing from the spec — it is a work in progress. Some sections have received more attention than others. The **hub** 1, the memory-mapped-input-output module**??** are among them as are various other "smaller" modules such as the binary module, the word comparison module and others.

Chapter 1

Hub

1.1 Columns

1.1.1 Conventions

Throughout the document we use a number of notational conventions which we explain here. These conventions apply to column names and are meant to clarify the origin and purpose of certain columns within a given trace. Others should be viewed as constructors which define new columns from existing ones.

Modules have three letter identifiers. The named modules are the following:

Module stamps. Module stamps count calls to a given module; most modules have a single stamp though the hub and ALU have several. Stamp columns are adorned with a $\square$, thus the $\mathsf{STO}\square$ is the module stamp of the storage module. Module stamps are typically computed/updated in the hub module whose main purpose is to dispatch (paid for an otherwise valid) instructions to the module(s) that are equipped to carry them out. Associating a unique identifier (i.e. stamp) to such "module-calls" is crucial when the order of operations matters. This is the case for instructions pertaining to (address) warmth (i.e. the WRM module), required gas computations (i.e. GAS), RAM (i.e. MMU and RAM), the stack (i.e. HUB), storage (i.e. STO), ... to cite a few. Stateless modules such as the modules handling arithmetic (i.e. the ALU module), binary (i.e. BIN) or word comparison (i.e. WCP) opcodes don't *require* a time stamp *per se* yet are given one nonetheless. (e.g. the address warmth module corresponds to the three letter identifier WRM); the corresponding stamp column is that identifier followed by $\square$ (e.g. $\mathsf{HUB}\square$).

Imported columns. Angular parentheses $\langle \cdots \rangle$ signal columns whose contents are **imported** from other modules by means of a lookup argument. By way of example: all modules[1] import their module stamp from the hub. Modules tasked with executing certain opcodes will typically import values from the stack (e.g. pairs of stack values $\langle {}_k\mathsf{VAL}^{\mathsf{hi}} \rangle, \langle {}_k\mathsf{VAL}^{\mathsf{lo}} \rangle$, for various $k \in \{1, 2, 3, 4\}$.) Many modules also imports values that aren't borrowed from the stack. E.g. the hub module imports the instruction $\langle \mathsf{INST} \rangle$ from the ROM, e.g. the GAS module imports the current, new and endowment gas values (GAS^{κ}, GAS^{ν} and $\mathsf{GAS}^{\varepsilon}$ respectively) from the hub, e.g. the OOB module imports execution context dependent data such exception flags, the size of return data $\langle \mathsf{RDS} \rangle$, the size of call data $\langle \mathsf{CDS} \rangle$ or the code size $\langle \mathsf{CODESIZE} \rangle$.

Decoded columns. A particular case of the above arises with **decoded columns**. Those are columns whose contents are extracted from a hardcoded collection of columns using a lookup argument.

[1] which are connected to the hub

They are adorned with a lozenge as in $^{\Diamond}\mathsf{COL}$. By way of example: the hub contains various instruction decoded flag columns but also a $^{\Diamond}\mathsf{STACK\_PATTERN}$ column whose contents are deduced from an immutable reference table called the **instruction decoder**. Similarly the binary module imports the results of binary operations performed on pairs of bytes (and injects the relevant one into the result.)

Flag columns. Among the instruction decoded columns on finds various binary flags columns (e.g. $^{\Diamond}\mathsf{ALU}$⚐, $^{\Diamond}\mathsf{MMU}$⚐, $^{\Diamond}\mathsf{EXP}$⚐, ...). These serve several purposes. The first is to provide an *indication* as to when modules *may* be sollicited by the hub to carry out an instruction. Thus arithmetic instructions raise the $^{\Diamond}\mathsf{ALU}$⚐, instructions that involve the RAM raise the $^{\Diamond}\mathsf{MMU}$⚐ etc ... Other flags trigger particular behaviours. For instance the PUSH⚐ and the JUMP⚐ trigger the expected behaviour of the program counter in the hub.

Module selector columns. When an instruction raises an instruction flag the associated module *may* get triggered. The actual trigger is usually deduced form this flag and exception flags. Such columns are tagged with a ϟ symbol

Interleaved columns. Certain arguments require us to merge columns of the same size into a single column. We use $\boxplus$ to signify the formation of such interleaved columns. E.g. starting with columns A, B and C of size n we may form the column $\mathsf{X} := \mathsf{A} \boxplus \mathsf{B} \boxplus \mathsf{C}$ defined as having length $3n$ and values

$$\begin{cases} \mathsf{X}_{3\cdot i+0} = \mathsf{A}_i \\ \mathsf{X}_{3\cdot i+1} = \mathsf{B}_i \\ \mathsf{X}_{3\cdot i+2} = \mathsf{C}_i \end{cases}$$

Row permutations. Our arithmetization requires row permutation arguments. These usually take the following form: we are given a small family of reference columns $\mathsf{REF}_1, \dots, \mathsf{REF}_p$ of equal size n (which we view as the columns of a $n \times p$ reference matrix REF). We are further given the description of an essentially unique permutation of the set $\{0, 1, \dots, n-1\}$ of rows indices, e.g. "(the essentially unique) row permutation of the matrix REF under which its rows appear lexicographically sorted". We then write $\mathsf{AUX}_j \mapsto [\mathsf{AUX}_j]^{\text{⤮}}$ for the mapping which takes an arbitrary column of the same size and applies the aforementioned row permutation to its rows.

1.1.2 Column descriptions

1. $\mathsf{INSTRUCTION\_STAMP}$: instruction stamp column; abbreviated to $\mathsf{INST}\square$; the first instruction takes place at $\mathsf{INST}\square = 1$; increases by 1 with every instruction;

2. $\mathsf{STACK\_STAMP}$: stack stamp column; abbreviated to abbreviated to $\mathsf{STACK}\square$; the first operation touching the batch's first transaction's root context's stack has $^{\square}\mathsf{STACK} = 1$; increases by one every time the stack is *popped*, *peeked at* or an item is *pushed* onto the stack;

How many **stack items** an instruction touches depends on the instruction itself; consecutive values of $^{\square}\mathsf{STACK}$ may jump by any value in the range $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$; the precise amount by which it jumps is decided by the **stack pattern** which the instruction follows.

3. HEIGHT: contains the current height of the current execution context's stack; the height is in the range $\{0, 1, \dots, 1024\}$ with $\mathsf{HEIGHT} = 0$ signifiying an empty stack;

4. HEIGHT^{ν}: contains the height of the current execution context's stack after dealing with the current instruction;

5. $\langle\mathsf{INST}\rangle$: instruction loaded from the ROM;

6. $\langle\mathsf{INSTRUCTION\_ARGUMENT}\rangle^{\mathsf{hi}}, \langle\mathsf{INSTRUCTION\_ARGUMENT}\rangle^{\mathsf{lo}}$: instruction argument (for `PUSH_X` instructions) loaded from the ROM; abbreviated to $\langle\mathsf{ARG}\rangle^{\mathsf{hi}}$ and $\langle\mathsf{ARG}\rangle^{\mathsf{lo}}$ respectively;

7. STATG: instruction decoded static gas cost of instruction;

8. ${}^{\diamondsuit}\mathsf{INST\_PARAMETER}$: instruction parameter obtained from instruction decoding $\langle\mathsf{INST}\rangle$; abbreviated to ${}^{\diamondsuit}\mathsf{PARAM}$;

9. ${}^{\diamondsuit}\mathsf{TWO\_LINE\_INSTRUCTION}$: instruction decoded binary flag indicating whether an instruction requires one or two rows in the execution trace; abbreviated to ${}^{\diamondsuit}\mathsf{TLI}$;

10. $\mathsf{COUNTER}$: binary counter column; abbreviated to CT;

For one line instructions (i.e. ${}^{\diamondsuit}\mathsf{TLI}_i = 0$) we have $\mathsf{CT}_i = 0$; for two line instructions (i.e. ${}^{\diamondsuit}\mathsf{TLI}_i = 1$) counter will count from 0 to 1 (i.e. $\mathsf{CT}_i = 0$ and $\mathsf{CT}_{i+1} = 0$ if we enter the instruction at row i).

11. ${}^{\diamondsuit}\mathsf{STACK\_PATTERN}$: instruction decoded "stack pattern" column; defines the pattern according to which stack values are touched or left empty; abbreviated to ${}^{\diamondsuit}\mathsf{PAT}$

12. ${}^{\diamondsuit}\mathsf{FLAG}^1, {}^{\diamondsuit}\mathsf{FLAG}^2, {}^{\diamondsuit}\mathsf{FLAG}^3$: three isntruction decoded binary flag columns;

For instance the ${}^{\diamondsuit}\mathsf{PARAM}$ associated with `DUPX`, $\mathtt{X} \in \{1, 2, \dots, 16\}$, is $\mathtt{X} - 1$ while the ${}^{\diamondsuit}\mathsf{PARAM}$ associated with `SWAPX`, $\mathtt{X} \in \{1, 2, \dots, 16\}$, is `X`. In our model, a **stack item** is fully determined by 6 parameters: the context number $\mathsf{CONTEXT\_NUMBER}$ (i.e. CN) and 5 other parameters which we describe below, though the stack items of a given row all share the same CN. We say that a stack item was **touched** by an instruction if it was either **peeked at**, **popped** or **pushed** onto stack. Every row of the present module touches up to 4 stack items. An instruction whose (instruction decoded) ${}^{\diamondsuit}\mathsf{TWO\_LINE\_INSTRUCTION}$ flag equals 0 can touch, in one way or another, up to 4 stack items; instructions whose (instruction decoded) ${}^{\diamondsuit}\mathsf{TWO\_LINE\_INSTRUCTION}$ flag equals 1 can touch, in one way or another, up to 8 stack items spread over 2 consecutive rows of the execution trace. Among the instructions raising the ${}^{\diamondsuit}\mathsf{TWO\_LINE\_INSTRUCTION}$ one finds all variations on `CALL`, the `LOG0-LOG4` instructions but also `CREATE` and `CREATE2`. The former is nonnegotiable as these instructions pop 6 or 7 items from stack and push a "success bit" onto it (which amounts to 7 or 8 touched stack items). The `LOG0`, `LOG1`, `LOG2` instuctions touch (pop) 2, 3 and 4 stack items respectively while `LOG3`, `LOG4` touch (pop) 5 and 6 stack items respectively. The `CREATE` and `CREATE2` instructions touch 4 and 5 stack items respectively. For simpler constraints we have chosen a uniform approach to all logs where the first row of the intruction touches (pops) the offset and size parameters and the next row touches (pops) the topics (if any). Similarly, the two creation instructions are dealt with uniformly.

The next 20 (!) columns contain information about the stack items an instruction touches. These 20 columns are comprised of 4 batches (parametrized by $k = 1, 2, 3, 4$) of 5 columns.

13. ${}_{k}\mathsf{HEIGHT}$: column containing the height $\in \{1, \dots, 1024\}$[2] of the k-th touched stack item;

14. ${}_{k}\mathsf{VAL}^{\mathsf{hi}}$: column containing the

15. ${}_{k}\mathsf{VAL}^{\mathsf{lo}}$: column containing the

16. ${}_{k}\mathsf{POP}$: binary column; ${}_{k}\mathsf{POP} = 1$ indicates that the item at height ${}_{k}\mathsf{HEIGHT}$ was popped; ${}_{k}\mathsf{POP} = 0$ indicates that the item at height ${}_{k}\mathsf{HEIGHT}$ was peeked at or pushed;

17. ${}^{\square}_{k}\mathsf{STACK}$: stack stamp;

The stack stamp columns will be used in the stack consistency constraints to impose a total order on the accesses to a given stack height of a given execution context. The pop flag will oscilate like so: 0 (i.e. push), 1 (i.e. pop), $0, 1, \dots$.

[2] Note the range difference between the ${}_{k}\mathsf{HEIGHT}$ columns and the HEIGHT column.

18. $\mathsf{STACK\_EXCEPTION}$: binary column; lights up precisely when an instruction raises a stack exception; depending on the instruction this is either a stack overflow or a stack underflow (or both in the case of `DUP_X` instructions); abbreviated to STX;

19. $\mathsf{STACK\_UNDERFLOW\_EXCEPTION}$: binary column; lights up precisely when an executing the current instruction would produce a stack underflow exception; abbreviated to SUX;

20. $\mathsf{STACK\_OVERFLOW\_EXCEPTION}$: binary column; lights up precisely when an executing the current instruction would produce a stack overflow exception; abbreviated to SOX;

21. $\mathsf{HEIGHT\_UNDER}$: used purely for detecting stack underflows; takes values in the range $\{0, 1, \dots, 1024\}$; abbreviated to HU;

22. $\mathsf{HEIGHT\_OVER}$: used purely for detecting stack overflows; takes values in the range $\{0, 1, \dots, 1024\}$; abbreviated to HO;

1.2 Stack

1.2.1 Heartbeat

This section describes the hearbeat of the stack module. It is imposed by two factors: the instruction decoded binary column $^{\Diamond}\mathsf{TWO\_LINE\_INSTRUCTION}$ and the $\mathsf{INSTRUCTION\_STAMP}$. The $\mathsf{COUNTER}$ column either stagnates at 0 if $^{\Diamond}\mathsf{TLI}_i = 0$ or counts from 0 to 1 if $^{\Diamond}\mathsf{TLI}_i = 1$. There are one or more padding rows at the beginning.

1. $\mathsf{INST}\square_0 = 0$;

2. $\mathsf{INST}\square$ is nondecreasing in the following sense: $\forall i,\ \mathsf{INST}\square_{i+1} \in \{\mathsf{INST}\square_i, 1 + \mathsf{INST}\square_i\}$;

3. IF $\mathsf{INST}\square_i = 0$ THEN $^{\Diamond}\mathsf{TLI}_i = 0$;

4. IF $^{\Diamond}\mathsf{TLI}_i = 0$ THEN $\Big(\mathsf{CT}_{i+1} = 0$ AND $\mathsf{CT}_i = 0\Big)$;

5. IF $\mathsf{INST}\square_i \neq 0$ THEN IF $^{\Diamond}\mathsf{TLI}_i = 1$:

   (a) IF $\mathsf{CT}_i \neq 1$ THEN
   $$\begin{cases} \mathsf{INST}\square_{i+1} = \mathsf{INST}\square_i \\ \langle \mathsf{INST} \rangle_{i+1} = \langle \mathsf{INST} \rangle_i \\ \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \end{cases}$$
   Note that in that case $^{\Diamond}\mathsf{TLI}_{i+1} = {}^{\Diamond}\mathsf{TLI}_i$ as well. Actually
   $$^{\Diamond}\mathsf{DECODED\_COLUMN}_{i+1} = {}^{\Diamond}\mathsf{DECODED\_COLUMN}_i$$
   for any instruction decoded column.

   (b) IF $\mathsf{CT}_i = 1$ THEN $\Big(\mathsf{CT}_{i+1} = 0$ AND $\mathsf{INST}\square_{i+1} = 1 + \mathsf{INST}\square_i\Big)$.

1.2.2 Counter constancy

We say that a column X is **counter-constant** if it satisfies

$$\mathsf{CT}_i \neq 0 \implies \mathsf{X}_i = \mathsf{X}_{i-1}$$

Note $\langle \mathsf{INST} \rangle$ is counter-constant by construction, see section 7.3.1. It follows that all instruction decoded flags are counter-constant. The following columns are counter-constant:

1. HEIGHT
2. HEIGHT^{ν}
3. HU
4. HO

1.2.3 Height range

We ask that the HEIGHT column satisfy the bound

$$\forall i,\ \left\{\begin{matrix}\mathsf{HEIGHT}_i\\ \mathsf{HEIGHT}_i^{\nu}\\ \mathsf{HU}_i\\ \mathsf{HO}_i\end{matrix}\right\} \in \{0,1,\dots,1024\}.$$

We test this by means of a Cairo-style small-range range-proof. Note that our arithmetization requires no further range check on the ${}_k\mathsf{HEIGHT}$, $k = 1,2,3,4$, columns. The above constraint is sufficient to enforce that:

- if the k-th stack item is nonempty then ${}_k\mathsf{HEIGHT} \in \{1,\dots,1024\}$;
- if the k-th stack item is mostly empty or empty then ${}_k\mathsf{HEIGHT} = 0$.

1.2.4 Zero padding

Beyond the heartbeat constraints and range constraints that take effect with the first row of the execution trace, all constraints detailed below apply under the assumption that

$$\boxed{\boxed{\mathsf{INST}\square_i \neq 0}}$$

In our implementation the execution trace of this module, like that of any other module, is padded with at least one row of zeros so that its length may hit a power of 2. In our implementation we include the following extra constraint for every column X of the module

$$\boxed{\textsc{if}\ \mathsf{INST}\square_i = 0\ \textsc{then}\ \mathsf{X}_i = 0}$$

1.2.5 Stack exceptions

Before the stack excavates any items it must first check whether doing so would cause an exception, i.e. a stack overflow or a stack underflow. The present section takes care of this check. It uses the HEIGHT column and instruction decoded evm parameters (δ_w, α_w) which occupy the ${}^{\diamond}\mathsf{DELTA}$ and ${}^{\diamond}\mathsf{ALPHA}$ columns respectively. and

1. We first check for stack underflows:
$$\mathsf{HU}_i = \big(2\cdot\mathsf{SUX}_i - 1\big)\cdot\big({}^{\diamond}\mathsf{DELTA}_i - \mathsf{HEIGHT}_i\big) - \mathsf{SUX}_i$$
2. $\textsc{if}$ $\mathsf{SUX}_i = 1$ $\textsc{then}$ $\mathsf{SOX}_i = 0$ (i.e. if a stack underflow occurred we set the overflow flag to 0.)
3. $\textsc{if}$ $\mathsf{SUX}_i = 0$ $\textsc{then}$ we check for overflows:
$$\mathsf{HO}_i = \big(2\cdot\mathsf{SOX}_i - 1\big)\cdot(\mathsf{HU}_i + {}^{\diamond}\mathsf{ALPHA}_i - 1024) - \mathsf{SOX}_i$$
Note that $\mathsf{SUX}_i = 0$ implies $\mathsf{HU}_i = \mathsf{HEIGHT}_i - {}^{\diamond}\mathsf{DELTA}_i$.
4. $\mathsf{STX}_i = \mathsf{SUX}_i + \mathsf{SOX}_i$.

By construction one cannot have both a stack overflow and an underflow at the same time. The preceding thus computes the binary flag $\mathsf{SUX}_i \vee \mathsf{SOX}_i = \mathsf{SUX}_i + \mathsf{SOX}_i - \mathsf{SUX}_i\cdot\mathsf{SOX}_i = \mathsf{SUX}_i + \mathsf{SOX}_i$.

1.2.6 Call stack depth exception

We provide the constraints for the CSDX flag.

1. CSDX is binary[3]

2. we impose a range constraint

$$\mathsf{CSD}_i + {}^{\diamond}\mathsf{CALL}\,\text{⚐}_i + {}^{\diamond}\mathsf{CREATE}\,\text{⚐}_i - 1025 \cdot \mathsf{CSDX}_i \ \in \ \{0, 1, \dots, 1024\}.$$

1.3 Stack patterns

1.3.1 Purpose

The present section explores stack patterns and sheds some light as to which instructions use what stack patterns. What we call a **stack pattern** is the pattern according to which the stack items touched by an individual instruction are laid out across the 4 to 8 stack items which are available to the instruction. Full details are given in section 1.4 on "one line stack patterns" and section 1.5 on "two line stack patterns".

The Ethereum Yellow Paper defines for every instruction w a pair of nonnegative integers (δ_w, α_w) where δ_w is the number of stack items w pops off the stack and α_w is the number of stack items w pushes onto the stack[4]. Similarly, every instruction has a corresponding zk-evm specific pair of nonnegative integers $(\delta_w^{\mathsf{zk}}, \alpha_w^{\mathsf{zk}})$ which, to some extent, determine the instruction's stack pattern. The pairs (δ_w, α_w) and $(\delta_w^{\mathsf{zk}}, \alpha_w^{\mathsf{zk}})$ don't necessarily coincide (though they mostly do.) For instance, the following inequalities always hold:

$$\delta_w^{\mathsf{zk}} \in \{0, 1, \dots, 7\}, \quad \alpha_w^{\mathsf{zk}} \in \{0, 1, 2\} \quad \text{and} \quad \delta_w^{\mathsf{zk}} + \alpha_w^{\mathsf{zk}} \in \{0, 1, \dots, 8\}$$

The most notable divergence between these parameter families comes from `DUP_X` and `SWAP_X` instructions, $\mathtt{X} \in \{1, \dots, 16\}$. The Ethereum Yellow Paper ascribes them, respectively, the pairs $(\delta_{\mathtt{DUP\_X}}, \alpha_{\mathtt{DUP\_X}}) = (\mathtt{X}, \mathtt{X} + 1)$ and $(\delta_{\mathtt{SWAP\_X}}, \alpha_{\mathtt{SWAP\_X}}) = (\mathtt{X} + 1, \mathtt{X} + 1)$. The stack pattern our arithmetization uses bears no dependence on `X`, as implicitly the zk-evm has:

$$(\delta_{\mathtt{DUP\_X}}^{\mathsf{zk}}, \alpha_{\mathtt{DUP\_X}}^{\mathsf{zk}}) = (1, 2) \quad \text{and} \quad (\delta_{\mathtt{SWAP\_X}}^{\mathsf{zk}}, \alpha_{\mathtt{SWAP\_X}}^{\mathsf{zk}}) = (2, 2).$$

In other words the zk-evm views `DUP_X` instructions (that don't raise a stack underflow or overflow exception) as the popping of one stack item (at height $\mathsf{HEIGHT}_i - (\mathtt{X} - 1)$) and two pushes (at height $\mathsf{HEIGHT}_i - (\mathtt{X} - 1)$ and $\mathsf{HEIGHT}_i + 1$ respectively)[5]. Similarly, the zk-evm views `SWAP_X` instructions (that don't raise a stack underflow exception) as the popping of two stack items (at height $\mathsf{HEIGHT}_i - \mathtt{X}$ and HEIGHT_i) and two pushes (at height $\mathsf{HEIGHT}_i - \mathtt{X}$ and HEIGHT_i respectively)[6]. Note that the parameter to substract from the current height[7] is read off the instruction decoded column ${}^{\diamond}\mathsf{PARAM}$.

The inequality $0 \leq \delta_w^{\mathsf{zk}} + \alpha_w^{\mathsf{zk}} \leq 8$ and our choice to excavate up to 4 stack items per row of the execution trace allow our stack to deal with every instruction in one or two rows. The instruction decoded binary column ${}^{\diamond}\mathsf{TWO\_LINE\_INSTRUCTION}$ records precisely this *hardcoded* distinction. Most of the time instructions w with $\delta_w^{\mathsf{zk}} + \alpha_w^{\mathsf{zk}} \leq 4$ have ${}^{\diamond}\mathsf{TLI} = 0$ though there are exceptions: `CREATE` and the three log instructions `LOG0`, `LOG1` and `LOG2` are counter-examples to this. We have chosen to deal with, on the one hand, `CREATE` and `CREATE2`, and on the other hand, the `LOGX` instructions, $\mathtt{X} \in \{0, \dots, 4\}$, in unified ways.

[3] and counter-constant by construction

[4] More precisely: $\delta_w \in \{0, 1, \dots, 7, 8, \dots, 17\}$ is the number of stack items w pops off of the current execution context's stack given that doing so doesn't raise a stack underflow exception, and $\alpha_w \in \{0, 1, 2, 3, \dots, 17\}$ is the number of stack items w pushes onto the current execution context's stack given that doing so doesn't raise a stack overflow exception.

[5] The value that was popped is pushed at both heights.

[6] The popped values are interchanged in the pushes.

[7] $\mathtt{X} - 1$ for `DUP_X`, `X` for `SWAP_X`

If $^{\diamondsuit}\mathsf{TLI} = 0$ and $\delta^{\mathsf{zk}} + \alpha^{\mathsf{zk}} < 4$ fewer than 4 stack items are touched. Similarly, if $^{\diamondsuit}\mathsf{TLI} = 1$ and $\delta^{\mathsf{zk}} + \alpha^{\mathsf{zk}} < 8$) fewer than 8 stack items are touched. In either case we need to also impose constraints on the "phantom stack items". The consistency checks described in section 1.3.3 ignore such rows.

A slight complication arises from the `CODECOPY` instruction. This is an instruction with $(\delta^{\mathsf{zk}}_w, \alpha^{\mathsf{zk}}_w) := (\delta_w, \alpha_w) = (3, 0)$ and $^{\diamondsuit}\mathsf{TLI} = 0$. The stack pattern of this instruction is what one would expect from an instruction following the `copyPattern`. Except that its fourth stack item is only *mostly* empty. We exploit the absence of constraints that caracterizes stack items of any execution environment at $\mathsf{HEIGHT} = 0$ (as well as any height of the 0^{th} execution environment.) This allows us to introduce the current context's $\mathsf{BC\_ADDR}$ into the $_4\mathsf{VAL}^{\mathsf{hi}}/_4\mathsf{VAL}^{\mathsf{lo}}$ fields without disturbance to stack consistency. The `RETURN` instruction, which is a $(\delta^{\mathsf{zk}}_w, \alpha^{\mathsf{zk}}_w) := (\delta_w, \alpha_w) = (2, 0)$ and $^{\diamondsuit}\mathsf{TLI} = 0$ instruction, comes with a similar complication. If the current execution context *isn't* a deployment context (i.e. $\mathsf{CTYPE} = 0$) then its fourth stack item is empty. If the current execution context *is* a deployment context (i.e. $\mathsf{CTYPE} = 1$) its fourth stack item is *mostly empty*. As before we plug the current context's $\mathsf{BC\_ADDR}$ into the $_4\mathsf{VAL}^{\mathsf{hi}}/_4\mathsf{VAL}^{\mathsf{lo}}$ fields and keep all other fields of the fourth stack item empty (i.e. $= 0$.) Again, this is without consequence for stack consistency constraints.

Here is an example: say the instruction pops $\delta^{\mathsf{zk}} = 2$ items and adds $\alpha^{\mathsf{zk}} = 1$ items and $^{\diamondsuit}\mathsf{TLI} = 0$ (i.e. it's a "one line instruction".) This stack pattern applies to most arithmetic operations, most word comparison operations and most binary operations which have two inputs and one output. Note that $^{\diamondsuit}\mathsf{TLI} = 0$ and $\delta^{\mathsf{zk}} + \alpha^{\mathsf{zk}} = 3 < 4$ so there is one "phantom stack item" (the third one). The associated stack pattern will impose values to all 4 stack items that the present line "excavates" like in figure ??

⟨INST⟩	$^{\diamondsuit}\mathsf{TLI}_i$	HEIGHT	HEIGHT^{ν}	STACK□	$\mathsf{STACK}\square^{\nu}$	$_1$ITEM	$_2$ITEM	$_3$ITEM	$_4$ITEM
`BLA`	0	h	h − **1**	st	st + **3**	⋯	⋯	∅	⋯

$_1$ITEM

$_1$HEIGHT	$_1\mathsf{VAL}^{\mathsf{hi}}$	$_1\mathsf{VAL}^{\mathsf{lo}}$	$_1$POP	$^{\square}_{1}$STACK
h − **0**	v_1^{hi}	v_1^{lo}	**1**	st + **1**

$_2$ITEM

$_2$HEIGHT	$_2\mathsf{VAL}^{\mathsf{hi}}$	$_2\mathsf{VAL}^{\mathsf{lo}}$	$_2$POP	$^{\square}_{2}$STACK
h − **1**	v_2^{hi}	v_2^{lo}	**1**	st + **2**

$_3$ITEM

$_3$HEIGHT	$_3\mathsf{VAL}^{\mathsf{hi}}$	$_3\mathsf{VAL}^{\mathsf{lo}}$	$_3$POP	$^{\square}_{3}$STACK
∅	∅	∅	∅	∅

$_4$ITEM

$_4$HEIGHT	$_4\mathsf{VAL}^{\mathsf{hi}}$	$_4\mathsf{VAL}^{\mathsf{lo}}$	$_4$POP	$^{\square}_{4}$STACK
h − **1**	v_4^{hi}	v_4^{lo}	**0**	st + **3**

Figure 1.1: The **values in this font** represent *hardcoded values associated with this particular stack pattern*. The values in **this font** are *also* hardcoded values but we reserve this font for empty stack items. Note that we consistently write ∅ to mean **0** when a field of a particular stack item is zero *because the stack item is empty*, see section 1.3.3.

1.3.2 Expected outcome

Designing stack patterns is straightforward for instructions pertaining to the binary module, the word comparison module, the arithmetic module and the storage module: the relevant instructions are relatively uniform in the number of arguments they retrieve from stack. There is more diversity for instructions touching the RAM and the call stack. Of the instructions directly touching RAM (or transaction call data) we want to achieve the following data pattern for instructions with $^{\diamondsuit}\mathsf{TLI} = 0$: While for instructions touching RAM that require two lines (i.e. $^{\diamondsuit}\mathsf{TLI} = 1$): We also list the expected stack patterns for instructions that induce changes in the call stack:

INST	Item 1	Item 2	Item 3	Item 4
CALLDATALOAD	offset	∅	∅	loaded
MLOAD	offset	∅	∅	loaded
MSTORE	offset	∅	∅	toStore
MSTORE8	offset	∅	∅	toStore
SLOAD	storage key	∅	∅	loaded
SSTORE	storage key	∅	∅	toStore
CALLDATACOPY	offset	(rel.) srcOffset	size	∅
CODECOPY	offset	srcOffset	size	(address)
EXTCODECOPY	offset	srcOffset	size	address
RETURNDATACOPY	offset	(rel.) srcOffset	size	∅
SHA3	offset	∅	size	hash
RETURN	offset	∅	size	(address)
REVERT	offset	∅	size	∅

Figure 1.2: Expected stack patterns for 1 line instructions touching the RAM module. We have already alluded to the special case of CODECOPY and its *mostly empty fourth stack item*. The property of being mostly empty (i.e. only containing address := BYTECODE_ADDRESS) is signaled by parentheses. We also signaled the same *mostly empty fourth stack item* issue with RETURN instructions ran in a deployment context. The interpretation of address := BYTECODE_ADDRESS is analoguous in this case, but now depends on the binary flag CTYPE.

INST	Item 1	Item 2	Item 3	Item 4	CT
LOG0	offset	∅	size	∅	0
	∅	∅	∅	∅	1
LOG1	offset	∅	size	∅	0
	topic1	∅	∅	∅	1
LOG2	offset	∅	size	∅	0
	topic1	topic2	∅	∅	1
LOG3	offset	∅	size	∅	0
	topic1	topic2	topic3	∅	1
LOG4	offset	∅	size	∅	0
	topic1	topic2	topic3	topic4	1
CREATE	offset	∅	size	address (or 0)	0
	∅	∅	value	∅	1
CREATE2	offset	salt	size	address (or 0)	0
	∅	∅	value	∅	1

Figure 1.3: Expected stack pattern for instructions with $^{\diamond}$TWO_LINE_INSTRUCTION = 1 touching the RAM module.

INST	Item 1	Item 2	Item 3	Item 4	CT
`CALL`	offset	R@O	size	R@C	0
	gas	address	value	success	1
`CALLCODE`	offset	R@O	size	R@C	0
	gas	address	value	success	1
`DELEGATECALL`	offset	R@O	size	R@C	0
	gas	address	$\emptyset$	success	1
`STATICCALL`	offset	R@O	size	R@C	0
	gas	address	$\emptyset$	success	1

Figure 1.4: Expected stack pattern for instructions with $^{\diamond}\mathsf{TWO\_LINE\_INSTRUCTION} = 1$ that don't touch the RAM module.

1.3.3 Empty stack item

Let $k \in \{1, 2, 3, 4\}$. We define the following "empty k-th stack item" constraint system:

$$ {}_{k}\texttt{EmptyStackItem} \iff \begin{cases} {}_{k}\mathsf{HEIGHT}_i = 0 \\ {}_{k}\mathsf{POP}_i = 0 \\ {}_{k}\mathsf{VAL}^{\mathsf{hi}}{}_i = 0 \\ {}_{k}\mathsf{VAL}^{\mathsf{lo}}{}_i = 0 \\ {}^{\square}_{k}\mathsf{STACK}_i = 0 \end{cases} $$

1.3.4 Stack exception pattern

We lay out the constraints and stack pattern associated to stack exceptions.

1. IF $\mathsf{STX}_i = 1$ THEN

   Stack Item n° 1: The first stack item is empty: ${}_{1}\texttt{EmptyStackItem}$

   Stack Item n° 2: The second item is empty: ${}_{2}\texttt{EmptyStackItem}$;

   Stack Item n° 3: The third item is empty: ${}_{3}\texttt{EmptyStackItem}$;

   Stack Item n° 4: The fourth stack item is empty: ${}_{4}\texttt{EmptyStackItem}$;

   Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i$;

   Height update: $\mathsf{HEIGHT}^{\nu}_i = 0$;

1.4 One line instruction stack patterns

1.4.1 Disclaimer

The stack patterns presented in the current section 1.4 apply **if and only if** $\mathsf{STX}_i = 0$.

1.4.2 (0,0)-pattern

Supported instructions. The `0_0_Pattern` corresponds to evm instructions w with $(\delta^{\mathsf{zk}}_w, \alpha^{\mathsf{zk}}_w) := (\delta_w, \alpha_w) = (0, 0)$, i.e.

1. STOP;
2. INVALID;
3. JUMPDEST;
4. any byte that isn't an opcode.

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following flags:

INST	$^\diamond$PAT	$^\diamond$TLI
(0,0)-instructions	0_0_Pattern	0

Constraints. We collect under the `0_0_Pattern` moniker the following collection of constraints:

Stack Item n° 1: The first stack item is empty: ${}_1$`EmptyStackItem`;

Stack Item n° 2: The second stack item is empty: ${}_2$`EmptyStackItem`;

Stack Item n° 3: The third stack item is empty: ${}_3$`EmptyStackItem`;

Stack Item n° 4: The fourth stack item is empty: ${}_4$`EmptyStackItem`; ${}_2$`EmptyStackItem`, ${}_3$`EmptyStackItem` and ${}_4$`EmptyStackItem`,

Stack stamp update: $\mathsf{STACK}□^{\nu}{}_i = \mathsf{STACK}□_i$,

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i$,

1.4.3 (0,1) and (1,0) patterns

Supported instructions. The instructions listed below are precisely the instructions with $^\diamond\mathsf{PAT} =$ `oneItemPattern`. The `oneItemPattern` corresponds to evm instructions w with $(\delta_w, \alpha_w) \in \{(1,0),(0,1)\}$, i.e. $(1,0)$-instructions. For such instructions $(\delta_w^{\mathsf{zk}}, \alpha_w^{\mathsf{zk}}) := (\delta_w, \alpha_w)$ and $^\diamond\mathsf{TLI} = 0$. The $(1,0)$-instructions are:

1. POP
2. JUMP
3. SELFDESTRUCT

and $(0,1)$-instructions:

1. ADDRESS
2. ORIGIN
3. CALLER
4. CALLVALUE
5. CALLDATASIZE
6. CODESIZE
7. GASPRICE
8. RETURNDATASIZE
9. COINBASE
10. TIMESTAMP
11. NUMBER
12. DIFFICULTY
13. GASLIMIT
14. CHAINID
15. SELFBALANCE
16. BASEFEE
17. PC
18. MSIZE
19. GAS
20. PUSH1-PUSH32

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following flags:

INST	$^\diamond$PAT	$^\diamond$TLI	$^\diamond$FLAG^1
(0,1)-instructions	oneItemPattern	0	0
(1,0)-instructions	oneItemPattern	0	1

Graphical representation. We figures below represent the `oneItemPattern` stack pattern:

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\mathsf{h}+1$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	$\emptyset$	$\emptyset$	$\emptyset$	res
${}_k\mathsf{POP}_i$	$\emptyset$	$\emptyset$	$\emptyset$	0
${}^{\square}_k\mathsf{STACK}_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\mathsf{st}+1$

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	h	$\emptyset$	$\emptyset$	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	top	$\emptyset$	$\emptyset$	$\emptyset$
${}_k\mathsf{POP}_i$	1	$\emptyset$	$\emptyset$	$\emptyset$
${}^{\square}_k\mathsf{STACK}_i$	$\mathsf{st}+1$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 1.5: The left hand side represents the stack patttern for $(0,1)$-instructions (i.e. ${}^{\diamond}\mathsf{FLAG}^1 = 0$), the right hand side represents the stack patttern for $(1,0)$-instructions (i.e. ${}^{\diamond}\mathsf{FLAG}^1 = 1$) We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i = \mathsf{st}$.

Constraints. We collect under the `oneItemPattern` moniker the following collection of constraints. They apply whenever

$$\boxed{\mathsf{STX}_i = 0}$$

Stack Item $n^\circ\,1$: The first stack item is contains a stack item *iff* ${}^{\diamond}\mathsf{FLAG}^1 = 1$:

$$\begin{cases} {}_1\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i \cdot {}^{\diamond}\mathsf{FLAG}^1, \\ {}_1\mathsf{POP}_i & = & {}^{\diamond}\mathsf{FLAG}^1, \\ {}^{\square}_1\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 1) \cdot {}^{\diamond}\mathsf{FLAG}^1. \end{cases}$$

Stack Item $n^\circ\,2$: The second item is empty: ${}_2$`EmptyStackItem`;

Stack Item $n^\circ\,3$: The third item is empty: ${}_3$`EmptyStackItem`;

Stack Item $n^\circ\,4$: The fourth stack item is contains a stack item *iff* ${}^{\diamond}\mathsf{FLAG}^1 = 0$:

$$\begin{cases} {}_4\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i + 1) \cdot (1 - {}^{\diamond}\mathsf{FLAG}^1), \\ {}_4\mathsf{POP}_i & = & 0, \\ {}^{\square}_4\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 1) \cdot (1 - {}^{\diamond}\mathsf{FLAG}^1). \end{cases}$$

Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 1$;

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i + (1 - 2 \cdot {}^{\diamond}\mathsf{FLAG}^1)$;

1.4.4 (1,1) and (2,0) patterns

Supported instructions. The stack pattern described below applies to the following instructions $(1,1)$-instructions:

- `ISZERO`
- `NOT`
- `BALANCE`
- `EXTCODESIZE`
- `EXTCODEHASH`
- `BLOCKHASH`
- `CALLDATALOAD`
- `MLOAD`
- `SLOAD`

and to the following $(2,0)$-instructions:

- `MSTORE`
- `MSTORE8`
- `SSTORE`
- `JUMPI`

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following:

INST	${}^{\Diamond}\mathsf{PAT}$	${}^{\Diamond}\mathsf{TLI}$	${}^{\Diamond}\mathsf{FLAG}^1$
`(1,1) instructions`	`twoItemPattern`	0	0
`(2,0) instructions`	`twoItemPattern`	0	1

Graphical representation. The picture is the following, for instance for `MLOAD` (${}^{\Diamond}\mathsf{FLAG}^1 = 0$) and `MSTORE` (${}^{\Diamond}\mathsf{FLAG}^1 = 1$)

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	h	$\emptyset$	$\emptyset$	h
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	$\mathsf{ARG1}$	$\emptyset$	$\emptyset$	OUT
${}_k\mathsf{POP}_i$	1	$\emptyset$	$\emptyset$	0
${}^{\Box}_k\mathsf{STACK}_i$	$\mathsf{st}+1$	$\emptyset$	$\emptyset$	$\mathsf{st}+2$

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	h	$\emptyset$	$\emptyset$	$\mathsf{h}-1$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	$\mathsf{ARG1}$	$\emptyset$	$\emptyset$	$\mathsf{ARG2}$
${}_k\mathsf{POP}_i$	1	$\emptyset$	$\emptyset$	1
${}^{\Box}_k\mathsf{STACK}_i$	$\mathsf{st}+1$	$\emptyset$	$\emptyset$	$\mathsf{st}+2$

Figure 1.6: On the left hand side is the picture for a $(1,1)$ instruction (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 0$). On the right hand side is the picture for a $(2,0)$ instruction (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 1$). We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\Box_i = \mathsf{st}$.

Constraints. We collect under the `twoItemPattern` moniker the following collection of constraints:

Stack Item n° 1: depending on the instruction contains either a **relative offset**, an **absolute offset** or a **storage key**:

$$\begin{cases} {}_1\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ {}_1\mathsf{POP}_i & = & 1, \\ {}^{\Box}_1\mathsf{STACK}_i & = & \mathsf{STACK}\Box_i + 1. \end{cases}$$

Stack Item n° 2: is empty: ${}_2$`EmptyStackItem`;

Stack Item n° 3: is empty: ${}_3$`EmptyStackItem`;

Stack Item n° 4: depending on the instruction, contains the value being loaded or being stored:

$$\begin{cases} {}_4\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - {}^{\Diamond}\mathsf{FLAG}^1 \\ {}_4\mathsf{POP}_i & = & {}^{\Diamond}\mathsf{FLAG}^1 \\ {}^{\Box}_4\mathsf{STACK}_i & = & \mathsf{STACK}\Box_i + 2. \end{cases}$$

Stack stamp update: $\mathsf{STACK}\Box^{\nu}{}_i = \mathsf{STACK}\Box_i + 2$,

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i - 2 \cdot {}^{\Diamond}\mathsf{FLAG}^1$,

For this set of instructions the interpretation of ${}^{\Diamond}\mathsf{FLAG}^1$ is that it equals 1 for storing instructions and 0 for loading instructions.

1.4.5 (2,1) and (3,1) patterns

Supported instructions. The stack pattern described below applies to the following $(\delta_w^{\mathtt{zk}}, \alpha_w^{\mathtt{zk}}) = (\delta_w, \alpha_w) = (2,1)$ instructions:

- `ADD`
- `MUL`
- `SUB`
- `DIV`
- `SDIV`
- `MOD`
- `SMOD`
- `EXP`
- `SIGNEXTEND`
- `LT`
- `GT`
- `SLT`
- `SGT`
- `EQ`
- `AND`
- `OR`
- `XOR`
- `BYTE`
- `SHL`
- `SHR`
- `SAR`
- `SHA3`

aswell as to the following $(\delta_w^{\mathtt{zk}}, \alpha_w^{\mathtt{zk}}) = (\delta_w, \alpha_w) = (3,1)$ instructions:

- `ADDMOD`
- `MULMOD`

Note that we *don't* include `CREATE` (which would have the correct signature ... maybe we should ?) The purpose of the ${}^{\diamondsuit}\mathsf{FLAG}^1$ is to differentiate between those instructions with 2 inputs (${}^{\diamondsuit}\mathsf{FLAG}^1 = 0$) and those instructions with 3 inputs (${}^{\diamondsuit}\mathsf{FLAG}^1 = 1$.)

Graphical representation. The picture is the following:

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\emptyset$	$\mathsf{h}-1$	$\mathsf{h}-1$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	ARG1	$\emptyset$	ARG2	OUT
${}_{k}\mathsf{POP}_i$	1	$\emptyset$	1	0
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st}+1$	$\emptyset$	$\mathsf{st}+2$	$\mathsf{st}+3$

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\mathsf{h}-2$	$\mathsf{h}-1$	$\mathsf{h}-2$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	ARG1	ARG3	ARG2	OUT
${}_{k}\mathsf{POP}_i$	1	1	1	0
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st}+1$	$\mathsf{st}+2$	$\mathsf{st}+3$	$\mathsf{st}+4$

Figure 1.7: Representation of the `standardPattern` for ${}^{\diamondsuit}\mathsf{FLAG}^1 = 0$ (left) and ${}^{\diamondsuit}\mathsf{FLAG}^1 = 1$ (right.) On the left hand side standard instructions with 2 arguments, on the right hand side standard instructions with 3 arguments. We chose to put the second instruction argument in the third stack item because of the `SHA3` instruction that, following expectations, expects to find its size parameter in the 3rd stack item. We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i = \mathsf{st}$.

Constraints. We collect under the `2_1_Pattern` moniker the following collection of constraints:

Stack Item n° **1:** contains the first instruction argument:

$$\begin{cases} {}_{1}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ {}_{1}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 1. \end{cases}$$

Stack Item n° **2:** contains the second instruction argument:

$$\begin{cases} {}_{2}\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^1{}_i, \\ {}_{2}\mathsf{POP}_i & = & {}^{\diamondsuit}\mathsf{FLAG}^1{}_i, \\ {}^{\square}_{2}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^1{}_i \end{cases}$$

Stack Item $n^\circ\,3$:

$$\left\{\begin{array}{lcl} {}_3\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1, \\ {}_3\mathsf{POP}_i & = & 1, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2 + {}^{\diamond}\mathsf{FLAG}^1{}_i. \end{array}\right.$$

Stack Item $n^\circ\,4$: contains the output of the instruction

$$\left\{\begin{array}{lcl} {}_4\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1 - {}^{\diamond}\mathsf{FLAG}^1{}_i, \\ {}_4\mathsf{POP}_i & = & 0, \\ {}^{\square}_{4}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 3 + {}^{\diamond}\mathsf{FLAG}^1. \end{array}\right.$$

Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 3 + {}^{\diamond}\mathsf{FLAG}^2{}_i$;

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i - 1 - {}^{\diamond}\mathsf{FLAG}^1{}_i$;

1.4.6 `DUP_X`-pattern

Supported instructions. The `dupPattern` is used by `DUP_X`, $\mathtt{X} \in \{1, 2, \ldots, 16\}$, instructions.

Relevant instruction decoded columns. Among all instruction decoded columns we only require the ${}^{\diamond}\mathsf{INST\_PARAMETER}$ column:

INST	${}^{\diamond}\mathsf{PAT}$	${}^{\diamond}\mathsf{TLI}$	${}^{\diamond}\mathsf{PARAM}$
`DUP_X`	`dupPattern`	0	$\mathtt{X} - 1$

Graphical representation. The figure below represents the `dupPattern` stack pattern:

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\mathsf{h} - {}^{\diamond}\mathsf{PARAM}$	$\emptyset$	$\mathsf{h} - {}^{\diamond}\mathsf{PARAM}$	$\mathsf{h} + 1$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	v	$\emptyset$	v	v
${}_k\mathsf{POP}_i$	1	$\emptyset$	0	0
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 1$	$\emptyset$	$\mathsf{st} + 2$	$\mathsf{st} + 3$

Figure 1.8: The stack pattern for `DUP_X` instructions. We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{st} = \mathsf{STACK}\square_i$.

Constraints. We collect under the `dupPattern` moniker the following collection of constraints:

1. First stack item:

$$\left\{\begin{array}{l} {}_1\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i - {}^{\diamond}\mathsf{PARAM}_i, \\ {}_1\mathsf{POP}_i = 1, \\ {}^{\square}_{1}\mathsf{STACK}_i = \mathsf{STACK}\square_i + 1. \end{array}\right.$$

2. Second stack item: ${}_2$`EmptyStackItem`.

3. Third stack item:

$$\left\{\begin{array}{l} {}_3\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i - {}^{\diamond}\mathsf{PARAM}_i, \\ {}_3\mathsf{POP}_i = 0, \\ {}_3\mathsf{VAL}^{\mathsf{hi}}{}_i = {}_1\mathsf{VAL}^{\mathsf{hi}}{}_i \\ {}_3\mathsf{VAL}^{\mathsf{lo}}{}_i = {}_1\mathsf{VAL}^{\mathsf{lo}}{}_i \\ {}^{\square}_{3}\mathsf{STACK}_i = \mathsf{STACK}\square_i + 2. \end{array}\right.$$

4. Fourth stack item:
$$\begin{cases} {}_4\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i + 1, \\ {}_4\mathsf{POP}_i = 0, \\ {}_4\mathsf{VAL}^{\mathsf{hi}}{}_i = {}_1\mathsf{VAL}^{\mathsf{hi}}{}_i \\ {}_4\mathsf{VAL}^{\mathsf{lo}}{}_i = {}_1\mathsf{VAL}^{\mathsf{lo}}{}_i \\ {}^{\square}_4\mathsf{STACK}_i = \mathsf{STACK}\square_i + 3. \end{cases}$$

5. $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 3$,

6. $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i + 1$,

1.4.7 SWAP_X-pattern

Supported instructions. The swapPattern is used by SWAP_X, $\mathtt{X} \in \{1, 2, \ldots, 16\}$, instructions.

Relevant instruction decoded columns. Among all instruction decoded columns we only require the $^{\diamond}\mathsf{INST\_PARAMETER}$ column:

INST	$^{\diamond}\mathsf{PAT}$	$^{\diamond}\mathsf{TLI}$	$^{\diamond}\mathsf{PARAM}$
SWAP_X	swapPattern	0	X

Graphical representation. The figure below represents the swapPattern stack pattern:

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\mathsf{h} - {}^{\diamond}\mathsf{PARAM}$	h	$\mathsf{h} - {}^{\diamond}\mathsf{PARAM}$	h
${}_k\mathsf{VAL}^{\mathsf{hi}} / {}_k\mathsf{VAL}^{\mathsf{lo}}$	v	v'	v'	v
${}_k\mathsf{POP}_i$	1	1	0	0
${}^{\square}_k\mathsf{STACK}_i$	$\mathsf{st} + 1$	$\mathsf{st} + 2$	$\mathsf{st} + 3$	$\mathsf{st} + 4$

Figure 1.9: The stack pattern for DUP_X instructions. We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{st} = \mathsf{STACK}\square_i$.

Constraints. We collect under the swapPattern moniker the following collection of constraints:

1. First stack item:
$$\begin{cases} {}_1\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i - {}^{\diamond}\mathsf{PARAM}_i, \\ {}_1\mathsf{POP}_i = 1, \\ {}^{\square}_1\mathsf{STACK}_i = \mathsf{STACK}\square_i + 1. \end{cases}$$

2. Second stack item:
$$\begin{cases} {}_2\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i, \\ {}_2\mathsf{POP}_i = 1, \\ {}^{\square}_2\mathsf{STACK}_i = \mathsf{STACK}\square_i + 2. \end{cases}$$

3. Third stack item:
$$\begin{cases} {}_3\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i - {}^{\diamond}\mathsf{PARAM}_i, \\ {}_3\mathsf{POP}_i = 0, \\ {}_3\mathsf{VAL}^{\mathsf{hi}}{}_i = {}_2\mathsf{VAL}^{\mathsf{hi}}{}_i \\ {}_3\mathsf{VAL}^{\mathsf{lo}}{}_i = {}_2\mathsf{VAL}^{\mathsf{lo}}{}_i \\ {}^{\square}_3\mathsf{STACK}_i = \mathsf{STACK}\square_i + 3. \end{cases}$$

4. Fourth stack item:

$$\begin{cases} {}_4\mathsf{HEIGHT}_i = \mathsf{HEIGHT}_i \\ {}_4\mathsf{POP}_i = 0, \\ {}_4\mathsf{VAL}^{\mathsf{hi}}{}_i = {}_1\mathsf{VAL}^{\mathsf{hi}}{}_i \\ {}_4\mathsf{VAL}^{\mathsf{lo}}{}_i = {}_1\mathsf{VAL}^{\mathsf{lo}}{}_i \\ {}^{\square}_4\mathsf{STACK}_i = \mathsf{STACK}\square_i + 4. \end{cases}$$

5. $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 4$,

6. $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i$,

1.4.8 RETURN/REVERT pattern

Supported instructions. The following stack pattern applies to

- RETURN
- REVERT

Relevant instruction decoded columns.

INST	$^{\diamond}$PAT	$^{\diamond}$TLI	$^{\diamond}$FLAG^1	CTYPE
RETURN	returnReversePattern	0	1	0
RETURN	returnReversePattern	0	1	1
REVERT	returnReversePattern	0	0	0/1

Graphical representation. The picture is the following

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	h	$\emptyset$	h − 1	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	offset	$\emptyset$	size	(BC_ADDR)
${}_k\mathsf{POP}_i$	1	$\emptyset$	1	$\emptyset$
${}^{\square}_k\mathsf{STACK}_i$	st + 1	$\emptyset$	st + 2	$\emptyset$

Figure 1.10: The first stack item contains an offset in the current execution context's RAM. The second stack item is empty. The third stack item contains the size of the return data. The fourth stack item is *mostly empty*. It contains the current BYTECODE_ADDRESS in case of a RETURN instruction happening in a deployment context. Otherwise it is $\emptyset$. We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i = \mathsf{st}$.

Constraints. We collect under the returnReversePattern moniker the following collection of constraints:

Stack Item $n°$ 1:

$$\begin{cases} \mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ \mathsf{POP}_i & = & 1, \\ {}^{\square}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 1. \end{cases}$$

Stack Item $n°$ 2: is left empty ${}_2\mathtt{EmptyStackItem}_i$;

Stack Item $n^\circ\,3$:

$$\begin{cases} \mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1, \\ \mathsf{POP}_i & = & 1, \\ {}^{\square}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2. \end{cases}$$

Stack Item $n^\circ\,4$:

$$\begin{cases} {}_{4}\mathsf{HEIGHT}_i & = & \emptyset, \\ {}_{4}\mathsf{VAL}^{\mathsf{hi}}{}_i & = & \mathsf{BC\_ADDR}^{\mathsf{hi}}{}_i \cdot {}^{\Diamond}\mathsf{FLAG}^1{}_i \cdot \mathsf{CTYPE}_i, \\ {}_{4}\mathsf{VAL}^{\mathsf{lo}}{}_i & = & \mathsf{BC\_ADDR}^{\mathsf{lo}}{}_i \cdot {}^{\Diamond}\mathsf{FLAG}^1{}_i \cdot \mathsf{CTYPE}_i, \\ {}_{4}\mathsf{POP}_i & = & \emptyset, \\ {}^{\square}_{4}\mathsf{STACK}_i & = & \emptyset \end{cases}$$

;

Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 2$;

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i - 2$;

1.4.9 Copy pattern

Supported instructions. The following stack pattern applies to

- `CODECOPY`
- `EXTCODECOPY`
- `CALLDATACOPY`
- `RETURNDATACOPY`

Relevant instruction decoded columns. The following instruction decoded flags are used to fill the stack pattern correctly for every instruction.

INST	$^{\Diamond}\mathsf{PAT}$	$^{\Diamond}\mathsf{TLI}$	$^{\Diamond}\mathsf{FLAG}^1$	$^{\Diamond}\mathsf{FLAG}^2$
`CODECOPY`	`copyPattern`	0	1	0
`EXTCODECOPY`	`copyPattern`	0	1	1
`CALLDATACOPY`	`copyPattern`	0	0	0
`RETURNDATACOPY`	`copyPattern`	0	0	1

For `CALLDATACOPY` and `RETURNDATACOPY` (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 0$) the fourth column is empty. For `CODECOPY` (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 1$, ${}^{\Diamond}\mathsf{FLAG}^2 = 0$) the fourth stack item is *mostly* empty but we stick the current code address into the value field. For `EXTCODECOPY` (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 1$, ${}^{\Diamond}\mathsf{FLAG}^2 = 1$) the fourth stack item is populated.

Graphical representation. For this set of instructions the interpretation of ${}^{\Diamond}\mathsf{FLAG}^1$ is that it equals 1 for `EXTCODECOPY` only. The picture is the following:

Constraints. We collect under the `copyPattern` moniker the following collection of constraints:

Stack Item $n^\circ\,1$: contains the **destination offset**:

$$\begin{cases} {}_{1}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - {}^{\Diamond}\mathsf{FLAG}^1{}_i \cdot {}^{\Diamond}\mathsf{FLAG}^2{}_i, \\ {}_{1}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 1. \end{cases}$$

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\mathsf{h}-1$	$\mathsf{h}-2$	$\emptyset$
${}_{k}\mathsf{VAL}^{\mathsf{hi}}/{}_{k}\mathsf{VAL}^{\mathsf{lo}}$	destOffset	(rel)offset	size	$\emptyset$
${}_{k}\mathsf{POP}_i$	1	1	1	$\emptyset$
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st}+1$	$\mathsf{st}+2$	$\mathsf{st}+3$	$\emptyset$

(`CALLDATACOPY` and `RETURNDATACOPY`)

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\mathsf{h}-1$	$\mathsf{h}-2$	$\emptyset$
${}_{k}\mathsf{VAL}^{\mathsf{hi}}/{}_{k}\mathsf{VAL}^{\mathsf{lo}}$	destOffset	(rel)offset	size	BC_ADDR
${}_{k}\mathsf{POP}_i$	1	1	1	$\emptyset$
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st}+1$	$\mathsf{st}+2$	$\mathsf{st}+3$	$\emptyset$

(`CODECOPY`)

	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	$\mathsf{h}-1$	$\mathsf{h}-2$	$\mathsf{h}-3$	h
${}_{k}\mathsf{VAL}^{\mathsf{hi}}/{}_{k}\mathsf{VAL}^{\mathsf{lo}}$	destOffset	(rel)offset	size	ADDR
${}_{k}\mathsf{POP}_i$	1	1	1	1
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st}+1$	$\mathsf{st}+2$	$\mathsf{st}+3$	$\mathsf{st}+4$

(`EXTCODECOPY`)

Figure 1.11: The first three items one pops from stack represent the offset where to start writing, the (relative) offset of where to start reading and the size (i.e. number of bytes to read.) This is all there is when ${}^{\diamond}\mathsf{FLAG}^{1}=0$. But for `EXTCODECOPY` (i.e. ${}^{\diamond}\mathsf{FLAG}^{1}={}^{\diamond}\mathsf{FLAG}^{2}=1$) there is an extra stack argument to pop: the address. For `CODECOPY` (i.e. ${}^{\diamond}\mathsf{FLAG}^{1}=1$ and ${}^{\diamond}\mathsf{FLAG}^{2}=0$) the fourth stack item is *technically* empty but we make it contain the current bytecode address. This will not perturb consistency constraints as $\mathsf{HEIGHT}=0$. We write $\mathsf{h}=\mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i=\mathsf{st}$.

Stack Item n° **2:** contains the **(naked) source offset**:

$$\begin{cases} {}_{2}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1 - {}^{\diamond}\mathsf{FLAG}^{1}{}_{i}\cdot{}^{\diamond}\mathsf{FLAG}^{2}{}_{i},\\ {}_{2}\mathsf{POP}_i & = & 1,\\ {}^{\square}_{2}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2. \end{cases}$$

Stack Item n° **3:** The third stack item contains the **size**:

$$\begin{cases} {}_{3}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 2 - {}^{\diamond}\mathsf{FLAG}^{1}{}_{i}\cdot{}^{\diamond}\mathsf{FLAG}^{2}{}_{i},\\ {}_{3}\mathsf{POP}_i & = & 1,\\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 3. \end{cases}$$

Stack Item n° **4:** The fourth stack item is empty for `CALLDATACOPY` and `RETURNDATACOPY`, mostly empty for `CODECOPY` and non empty for `EXTCODECOPY` where it contains an **address** popped off the stack:

1. $$\begin{cases} {}_{4}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i\cdot{}^{\diamond}\mathsf{FLAG}^{1}{}_{i}\cdot{}^{\diamond}\mathsf{FLAG}^{2}{}_{i}\\ {}_{4}\mathsf{POP}_i & = & {}^{\diamond}\mathsf{FLAG}^{1}{}_{i}\cdot{}^{\diamond}\mathsf{FLAG}^{2}{}_{i}\\ {}^{\square}_{4}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 4)\cdot{}^{\diamond}\mathsf{FLAG}^{1}{}_{i}\cdot{}^{\diamond}\mathsf{FLAG}^{2}{}_{i} \end{cases}$$

2. IF $\left({}^{\Diamond}\mathsf{FLAG}^{1}{}_{i} = 1 \quad \text{AND} \quad {}^{\Diamond}\mathsf{FLAG}^{2}{}_{i} = 0\right)$ THEN

$$\begin{cases} {}_{4}\mathsf{VAL}^{\mathsf{hi}}{}_{i} = \mathsf{BC\_ADDR}^{\mathsf{hi}}{}_{i} \\ {}_{4}\mathsf{VAL}^{\mathsf{lo}}{}_{i} = \mathsf{BC\_ADDR}^{\mathsf{lo}}{}_{i} \end{cases}$$

Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_{i} = \mathsf{STACK}\square_{i} + 3 + {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i} \cdot {}^{\Diamond}\mathsf{FLAG}^{2}{}_{i},$

Height update: $\mathsf{HEIGHT}^{\nu}{}_{i} = \mathsf{HEIGHT}_{i} - 3 - {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i} \cdot {}^{\Diamond}\mathsf{FLAG}^{2}{}_{i},$

1.5 Two line instruction stack patterns patterns

1.5.1 Disclaimer

The stack patterns presented in the current section 1.5 apply **if and only if** $\mathsf{STX}_{i} = 0$.

1.5.2 `LOG_X` pattern

Supported instructions. The following stack pattern applies to

- `LOG0`
- `LOG1`
- `LOG2`
- `LOG3`
- `LOG4`

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following flags:

INST	${}^{\Diamond}\mathsf{PAT}$	${}^{\Diamond}\mathsf{PARAM}$	${}^{\Diamond}\mathsf{TLI}$	${}^{\Diamond}\mathsf{FLAG}^{1}$	${}^{\Diamond}\mathsf{FLAG}^{2}$	${}^{\Diamond}\mathsf{FLAG}^{3}$
`LOG0`	`logPattern`	0	1	0	0	0
`LOG1`	`logPattern`	1	1	1	0	0
`LOG2`	`logPattern`	2	1	1	0	1
`LOG3`	`logPattern`	3	1	1	1	0
`LOG4`	`logPattern`	4	1	1	1	1

Graphical representation. The picture is the following

$(\mathsf{CT}_{i} = 0)$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_{i}$	h	$\emptyset$	$\mathsf{h} - 1$	$\emptyset$
${}_{k}\mathsf{VAL}^{\mathsf{hi}}/\,{}_{k}\mathsf{VAL}^{\mathsf{lo}}$	offset	$\emptyset$	size	$\emptyset$
${}_{k}\mathsf{POP}_{i}$	1	$\emptyset$	1	$\emptyset$
${}^{\square}_{k}\mathsf{STACK}_{i}$	$\mathsf{st} + 1$	$\emptyset$	$\mathsf{st} + 2$	$\emptyset$

Figure 1.12: This table represents the stack pattern of the first row $(\mathsf{CT}_{i} = 0)$ of a log instruction. We write $\mathsf{h} = \mathsf{HEIGHT}_{i}$ and $\mathsf{st} = \mathsf{STACK}\square_{i}$.

$(\mathsf{CT}_i = 1)$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}} / {}_k\mathsf{VAL}^{\mathsf{lo}}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
${}_k\mathsf{POP}_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
${}^{\square}_{k}\mathsf{STACK}_i$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$(\mathsf{CT}_i = 1)$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\mathsf{h} - 2$	$\mathsf{h} - 3$	$\mathsf{h} - 4$	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}} / {}_k\mathsf{VAL}^{\mathsf{lo}}$	topic1	topic2	topic3	$\emptyset$
${}_k\mathsf{POP}_i$	1	1	1	$\emptyset$
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 3$	$\mathsf{st} + 4$	$\mathsf{st} + 5$	$\emptyset$

Figure 1.13: This table represents the stack pattern of the second row ($\mathsf{CT}_i = 1$) of a `LOG0` and a `LOG3` instruction respectively. The other logs follow the same pattern. As previously we write $\mathsf{h} = \mathsf{HEIGHT}_i = \mathsf{HEIGHT}_{i-1}$ and $\mathsf{st} = \mathsf{STACK}\square_i = \mathsf{STACK}\square_{i-1}$.

Constraints. We collect under the moniker `logPattern` the following collection of constraints:

1. IF $\mathsf{CT}_i = 0$:

   Stack Item n° 1: the first stack item of the first row contains the offset:

   $$\begin{cases} {}_1\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ {}_1\mathsf{POP}_i & = & 1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 1 \end{cases}$$

   Stack Item n° 2: the fourth stack item of the first row is empty: ${}_2\texttt{EmptyStackItem}_i$;

   Stack Item n° 3: the third stack item of the first row contains the size:

   $$\begin{cases} {}_3\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1, \\ {}_3\mathsf{POP}_i & = & 1, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2 \end{cases}$$

   Stack Item n° 4: the fourth stack item of the first row is empty: ${}_4\texttt{EmptyStackItem}_i$;

   Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 2 + {}^{\diamondsuit}\mathsf{PARAM}$;

   Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i - 2 - {}^{\diamondsuit}\mathsf{PARAM}$;

2. IF $\mathsf{CT}_i = 1$:

   Stack Item "n° 5": the first stack item of the second row *may* contain a first topic:

   $$\begin{cases} {}_1\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^1 \\ {}_1\mathsf{POP}_i & = & {}^{\diamondsuit}\mathsf{FLAG}^1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 3) \cdot {}^{\diamondsuit}\mathsf{FLAG}^1 \end{cases}$$

   Stack Item "n° 6": the second stack item of the second row *may* contain a second topic:

   $$\begin{cases} {}_2\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 3) \cdot \left({}^{\diamondsuit}\mathsf{FLAG}^2 + (1 - {}^{\diamondsuit}\mathsf{FLAG}^2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^3 \right), \\ {}_2\mathsf{POP}_i & = & {}^{\diamondsuit}\mathsf{FLAG}^2 + (1 - {}^{\diamondsuit}\mathsf{FLAG}^2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^3, \\ {}^{\square}_{2}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 4) \cdot \left({}^{\diamondsuit}\mathsf{FLAG}^2 + (1 - {}^{\diamondsuit}\mathsf{FLAG}^2) \cdot {}^{\diamondsuit}\mathsf{FLAG}^3 \right) \end{cases}$$

   Stack Item "n° 7": the third stack item of the second row *may* contain a third topic:

   $$\begin{cases} {}_3\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 4) \cdot {}^{\diamondsuit}\mathsf{FLAG}^2, \\ {}_3\mathsf{POP}_i & = & {}^{\diamondsuit}\mathsf{FLAG}^2, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 5) \cdot {}^{\diamondsuit}\mathsf{FLAG}^2 \end{cases}$$

Stack Item "$n^\circ 8$": the fourth stack item of the second row *may* contain a fourth topic:

$$\begin{cases} {}_{4}\mathsf{HEIGHT}_i &= (\mathsf{HEIGHT}_i - 5) \cdot {}^{\diamond}\mathsf{FLAG}^2 \cdot {}^{\diamond}\mathsf{FLAG}^3, \\ {}_{4}\mathsf{POP}_i &= {}^{\diamond}\mathsf{FLAG}^2 \cdot {}^{\diamond}\mathsf{FLAG}^3, \\ {}^{\square}_{4}\mathsf{STACK}_i &= (\mathsf{STACK}\square_i + 6) \cdot {}^{\diamond}\mathsf{FLAG}^2 \cdot {}^{\diamond}\mathsf{FLAG}^3, \end{cases}$$

1.5.3 Call pattern

Supported instructions. The following stack pattern applies to all "call instructions" i.e. the instructions below:

- `CALL`
- `CALLCODE`
- `DELEGATECALL`
- `STATICCALL`

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following flags:

INST	${}^{\diamond}\mathsf{PAT}$	${}^{\diamond}\mathsf{TLI}$	${}^{\diamond}\mathsf{FLAG}^1$	${}^{\diamond}\mathsf{FLAG}^2$
`CALL`	`callPattern`	1	1	0
`CALLCODE`	`callPattern`	1	1	1
`DELEGATECALL`	`callPattern`	1	0	0
`STATICCALL`	`callPattern`	1	0	1

The interpretation is the following: call instructions w with ${}^{\diamond}\mathsf{FLAG}^1 = 1$ have $\delta^{\mathsf{zk}}_w = \delta_w = 7$ and those with ${}^{\diamond}\mathsf{FLAG}^1 = 0$ have $\delta^{\mathsf{zk}}_w = \delta_w = 6$ (and all call instructions have $\alpha^{\mathsf{zk}}_w = \alpha_w = 1$.) Though the stack pattern does not depend on it, we recall here the interpretation of the second flag ${}^{\diamond}\mathsf{FLAG}^2$: it differentiate between `CALL` and `CALLCODE` as well as between `DELEGATECALL` and `STATICCALL`.

Graphical representation. We represent the stack pattern when ${}^{\diamond}\mathsf{FLAG}^1 = 0$ is in figure ?? and similarly for ${}^{\diamond}\mathsf{FLAG}^1 = 0$ see figure ??.

$\mathsf{CT}_i = 0$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	$\mathsf{h} - 2$	$\mathsf{h} - 4$	$\mathsf{h} - 3$	$\mathsf{h} - 5$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	CDO	R@O	CDS	R@C
${}_{k}\mathsf{POP}_i$	1	1	1	1
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 1$	$\mathsf{st} + 2$	$\mathsf{st} + 3$	$\mathsf{st} + 4$

$\mathsf{CT}_i = 1$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\mathsf{h} - 1$	$\emptyset$	$\mathsf{h} - 5$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	gas	address	$\emptyset$	success
${}_{k}\mathsf{POP}_i$	1	1	$\emptyset$	0
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 5$	$\mathsf{st} + 6$	$\emptyset$	$\mathsf{st} + 7$

Figure 1.14: The above represents the stack patttern for ${}^{\diamond}\mathsf{FLAG}^1 = 0$ (i.e. for `DELEGATECALL` and `STATICCALLCODE` instructions). We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i = \mathsf{st}$.

Constraints. We collect under the moniker `callPattern` the following collection of constraints:

1. IF $\mathsf{CT}_i = 0$:

   Stack Item $n^\circ 1$: the first stack item of the first row of the instruction:

$$\begin{cases} {}_{1}\mathsf{HEIGHT}_i &= \mathsf{HEIGHT}_i - 2 - {}^{\diamond}\mathsf{FLAG}^1{}_i, \\ {}_{1}\mathsf{POP}_i &= 1, \\ {}^{\square}_{1}\mathsf{STACK}_i &= \mathsf{STACK}\square_i + 1. \end{cases}$$

$\mathsf{CT}_i = 0$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	$\mathsf{h} - 3$	$\mathsf{h} - 5$	$\mathsf{h} - 4$	$\mathsf{h} - 6$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	CDO	R@O	CDS	R@C
${}_{k}\mathsf{POP}_i$	1	1	1	1
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 1$	$\mathsf{st} + 2$	$\mathsf{st} + 3$	$\mathsf{st} + 4$

$\mathsf{CT}_i = 1$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_{k}\mathsf{HEIGHT}_i$	h	$\mathsf{h} - 1$	$\mathsf{h} - 2$	$\mathsf{h} - 6$
${}_{k}\mathsf{VAL}^{\mathsf{hi}} / {}_{k}\mathsf{VAL}^{\mathsf{lo}}$	gas	address	value	success
${}_{k}\mathsf{POP}_i$	1	1	1	0
${}^{\square}_{k}\mathsf{STACK}_i$	$\mathsf{st} + 5$	$\mathsf{st} + 6$	$\mathsf{st} + 7$	$\mathsf{st} + 8$

Figure 1.15: The above represents the stack patttern for ${}^{\diamond}\mathsf{FLAG}^{1} = 1$ (i.e. for `CALL` and `CALLCODE` instructions). Recall that CDO, R@O, CDS, R@C are short hand for CALLDATA_OFFSET, RETURN@OFFSET, CALLDATA_SIZE, RETURN@CAPACITY respectively. We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\square_i = \mathsf{st}$.

Stack Item $n^\circ\, 2$**:** the second stack item of the first row of the instruction:

$$\begin{cases} {}_{2}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 4 - {}^{\diamond}\mathsf{FLAG}^{1}{}_i, \\ {}_{2}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{2}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2. \end{cases}$$

Stack Item $n^\circ\, 3$**:** the third stack item of the first row of the instruction:

$$\begin{cases} {}_{3}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 3 - {}^{\diamond}\mathsf{FLAG}^{1}{}_i, \\ {}_{3}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 3. \end{cases}$$

Stack Item $n^\circ\, 4$**:** the fourth stack item of the first row of the instruction:

$$\begin{cases} {}_{4}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 5 - {}^{\diamond}\mathsf{FLAG}^{1}{}_i, \\ {}_{4}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{4}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 4. \end{cases}$$

Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_i = \mathsf{STACK}\square_i + 7 + {}^{\diamond}\mathsf{FLAG}^{1}{}_i$;

Height update: $\mathsf{HEIGHT}^{\nu}{}_i = \mathsf{HEIGHT}_i - 5 - {}^{\diamond}\mathsf{FLAG}^{1}$;

2. ☞ $\mathsf{CT}_i = 1$:

Stack Item "$n^\circ\, 5$**":** the first stack item of the second row of the instruction:

$$\begin{cases} {}_{1}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ {}_{1}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 5. \end{cases}$$

Stack Item "$n^\circ\, 6$**":** the second stack item of the second row of the instruction:

$$\begin{cases} {}_{2}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1, \\ {}_{2}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{2}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 6. \end{cases}$$

Stack Item "$n^\circ\, 7$**":** the third stack item of the second row of the instruction:

$$\begin{cases} {}_{3}\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 2) \cdot {}^{\diamond}\mathsf{FLAG}^{1}{}_i, \\ {}_{3}\mathsf{POP}_i & = & {}^{\diamond}\mathsf{FLAG}^{1}{}_i, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 7) \cdot {}^{\diamond}\mathsf{FLAG}^{1}{}_i. \end{cases}$$

Stack Item "$n^\circ\,8$": the fourth stack item of the second row of the instruction:

$$\begin{cases} {}_4\mathsf{HEIGHT}_i & = \quad \mathsf{HEIGHT}_i - 5 - {}^{\Diamond}\mathsf{FLAG}^1{}_i, \\ {}_4\mathsf{POP}_i & = \quad 0, \\ {}^{\Box}_4\mathsf{STACK}_i & = \quad \mathsf{STACK}\Box_i + 7 + {}^{\Diamond}\mathsf{FLAG}^1{}_i. \end{cases}$$

1.5.4 Create pattern

Supported instructions. The stack pattern we describe below applies to both creation instructions:

- `CREATE`
- `CREATE2`

Although the `CREATE` instruction has $(\delta_w^{\mathsf{zk}}, \alpha_w^{\mathsf{zk}}) = (\delta_w, \alpha_w) = (3,1)$ and would thus fit into a single row of the execution trace ($\delta_w^{\mathsf{zk}} + \alpha_w^{\mathsf{zk}} = 4$) we have chosen a unified approach to both create instructions.

Relevant instruction decoded columns. Among all instruction decoded columns we focus on the following flags:

INST	${}^{\Diamond}\mathsf{PAT}$	${}^{\Diamond}\mathsf{TLI}$	${}^{\Diamond}\mathsf{FLAG}^1$
`CREATE`	`createPattern`	1	0
`CREATE2`	`createPattern`	1	1

Graphical representation. We represent the stack pattern for `CREATE` instructions (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 0$) in figure ?? and that for `CREATE2` instructions (i.e. ${}^{\Diamond}\mathsf{FLAG}^1 = 1$) in figure ??.

$\mathsf{CT}_i = 0$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\mathsf{h}-1$	$\emptyset$	$\mathsf{h}-2$	$\mathsf{h}-2$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	offset	$\emptyset$	size	address (or 0)
${}_k\mathsf{POP}_i$	1	$\emptyset$	1	0
${}^{\Box}_k\mathsf{STACK}_i$	$\mathsf{st}+1$	$\emptyset$	$\mathsf{st}+2$	$\mathsf{st}+3$

$\mathsf{CT}_i = 1$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\emptyset$	$\emptyset$	h	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	$\emptyset$	$\emptyset$	value	$\emptyset$
${}_k\mathsf{POP}_i$	$\emptyset$	$\emptyset$	1	$\emptyset$
${}^{\Box}_k\mathsf{STACK}_i$	$\emptyset$	$\emptyset$	$\mathsf{st}+4$	$\emptyset$

Figure 1.16: The above represents the stack patttern for ${}^{\Diamond}\mathsf{FLAG}^1 = 0$ (i.e. for `CREATE` instructions). We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\Box_i = \mathsf{st}$.

$\mathsf{CT}_i = 0$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\mathsf{h}-1$	$\mathsf{h}-3$	$\mathsf{h}-2$	$\mathsf{h}-3$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	offset	salt	size	address (or 0)
${}_k\mathsf{POP}_i$	1	1	1	0
${}^{\Box}_k\mathsf{STACK}_i$	$\mathsf{st}+1$	$\mathsf{st}+2$	$\mathsf{st}+3$	$\mathsf{st}+4$

$\mathsf{CT}_i = 1$	Stack Item 1	Stack Item 2	Stack Item 3	Stack Item 4
${}_k\mathsf{HEIGHT}_i$	$\emptyset$	$\emptyset$	h	$\emptyset$
${}_k\mathsf{VAL}^{\mathsf{hi}}/{}_k\mathsf{VAL}^{\mathsf{lo}}$	$\emptyset$	$\emptyset$	value	$\emptyset$
${}_k\mathsf{POP}_i$	$\emptyset$	$\emptyset$	1	$\emptyset$
${}^{\Box}_k\mathsf{STACK}_i$	$\emptyset$	$\emptyset$	$\mathsf{st}+5$	$\emptyset$

Figure 1.17: The above represents the stack patttern for ${}^{\Diamond}\mathsf{FLAG}^1 = 1$ (i.e. for `CREATE2` instructions). We write $\mathsf{h} = \mathsf{HEIGHT}_i$ and $\mathsf{STACK}\Box_i = \mathsf{st}$.

Constraints. We collect under the moniker `createPattern` the following collection of constraints:

1. IF $\mathsf{CT}_i = 0$:

   Stack Item $n^\circ\,1$**:** the first stack item of the first row of the instruction:

$$\left\{\begin{array}{lcl} {}_{1}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 1, \\ {}_{1}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{1}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 1. \end{array}\right.$$

   Stack Item $n^\circ\,2$**:** the second stack item of the first row of the instruction:

$$\left\{\begin{array}{lcl} {}_{2}\mathsf{HEIGHT}_i & = & (\mathsf{HEIGHT}_i - 3) \cdot {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i} \\ {}_{2}\mathsf{POP}_i & = & {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}, \\ {}^{\square}_{2}\mathsf{STACK}_i & = & (\mathsf{STACK}\square_i + 2) \cdot {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}. \end{array}\right.$$

   Stack Item $n^\circ\,3$**:** the third stack item of the first row of the instruction:

$$\left\{\begin{array}{lcl} {}_{3}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 2, \\ {}_{3}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 2 + {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}. \end{array}\right.$$

   Stack Item $n^\circ\,4$**:** the fourth stack item of the first row of the instruction:

$$\left\{\begin{array}{lcl} {}_{4}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i - 2 - {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}, \\ {}_{4}\mathsf{POP}_i & = & 0, \\ {}^{\square}_{4}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 3 + {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}. \end{array}\right.$$

   Stack stamp update: $\mathsf{STACK}\square^{\nu}{}_{i} = \mathsf{STACK}\square_i + 4 + {}^{\Diamond}\mathsf{FLAG}^{1}$;

   Height update: $\mathsf{HEIGHT}^{\nu}{}_{i} = \mathsf{HEIGHT}_i - 2 - {}^{\Diamond}\mathsf{FLAG}^{1}$;

2. IF $\mathsf{CT}_i = 1$:

   Stack Item "$n^\circ\,5$**":** the first stack item of the second row of the instruction is always empty: ${}_{1}$`EmptyStackItem`;

   Stack Item "$n^\circ\,6$**":** the second stack item of the second row of the instruction is always empty: ${}_{2}$`EmptyStackItem`;

   Stack Item "$n^\circ\,7$**":** the third stack item of the second row of the instruction satisfies:

$$\left\{\begin{array}{lcl} {}_{3}\mathsf{HEIGHT}_i & = & \mathsf{HEIGHT}_i, \\ {}_{3}\mathsf{POP}_i & = & 1, \\ {}^{\square}_{3}\mathsf{STACK}_i & = & \mathsf{STACK}\square_i + 4 + {}^{\Diamond}\mathsf{FLAG}^{1}{}_{i}. \end{array}\right.$$

   Stack Item "$n^\circ\,8$**":** the fourth stack item of the second row of the instruction is always empty: ${}_{4}$`EmptyStackItem`;

1.6 Constraints

1.6.1 Stack consistency

This section describes the consistency constraints that ensure that any stack item excavated from the stack of a given execution context at a given height coincides with the last stack item pushed onto the same execution context's stack at the same height. We introduce some interleaved columns:

1. $\mathsf{CN}^{\boxplus 4} = \mathsf{CN} \boxplus \mathsf{CN} \boxplus \mathsf{CN} \boxplus \mathsf{CN}$
2. ${}_{1234}\mathsf{HEIGHT} = {}_{1}\mathsf{HEIGHT} \boxplus {}_{2}\mathsf{HEIGHT} \boxplus {}_{3}\mathsf{HEIGHT} \boxplus {}_{4}\mathsf{HEIGHT}$
3. ${}_{1234}^{\square}\mathsf{STACK} = {}_{1}^{\square}\mathsf{STACK} \boxplus {}_{2}^{\square}\mathsf{STACK} \boxplus {}_{3}^{\square}\mathsf{STACK} \boxplus {}_{4}^{\square}\mathsf{STACK}$
4. ${}_{1234}\mathsf{POP} = {}_{1}\mathsf{POP} \boxplus {}_{2}\mathsf{POP} \boxplus {}_{3}\mathsf{POP} \boxplus {}_{4}\mathsf{POP}$
5. ${}_{1234}\mathsf{VAL}^{\mathsf{hi}} = {}_{1}\mathsf{VAL}^{\mathsf{hi}} \boxplus {}_{2}\mathsf{VAL}^{\mathsf{hi}} \boxplus {}_{3}\mathsf{VAL}^{\mathsf{hi}} \boxplus {}_{4}\mathsf{VAL}^{\mathsf{hi}}$
6. ${}_{1234}\mathsf{VAL}^{\mathsf{lo}} = {}_{1}\mathsf{VAL}^{\mathsf{lo}} \boxplus {}_{2}\mathsf{VAL}^{\mathsf{lo}} \boxplus {}_{3}\mathsf{VAL}^{\mathsf{lo}} \boxplus {}_{4}\mathsf{VAL}^{\mathsf{lo}}$

This contains some meaningless rows: the rows i with $\mathsf{CN}_i^{\boxplus 4} = 0$ correspond to padding; the rows i with ${}_{1234}\mathsf{HEIGHT}_i = 0$ correspond to empty stack items. Consider *a* row permutation $\mathsf{X} \mapsto [X]^{\text{⤮}}$ such that

$$\left(\left[\mathsf{CN}^{\boxplus 4}\right]^{\text{⤮}}, \left[{}_{1234}\mathsf{HEIGHT}\right]^{\text{⤮}}, \left[{}_{1234}^{\square}\mathsf{STACK}\right]^{\text{⤮}} \right)$$

are in lexicographically order.

1. IF $\left(\left[\mathsf{CN}^{\boxplus 4}\right]_{i+1}^{\text{⤮}} = 0 \text{ OR } \left[{}_{1234}\mathsf{HEIGHT}\right]_{i+1}^{\text{⤮}} = 0 \right)$ we don't impose any consistency constraints;
2. IF $\left(\left[\mathsf{CN}^{\boxplus 4}\right]_{i+1}^{\text{⤮}} \neq 0 \text{ AND } \left[{}_{1234}\mathsf{HEIGHT}\right]_{i+1}^{\text{⤮}} \neq 0 \right)$ THEN
   (a) IF
   $$\begin{cases} \left[\mathsf{CN}^{\boxplus 4}\right]_{i+1}^{\text{⤮}} = \left[\mathsf{CN}^{\boxplus 4}\right]_{i}^{\text{⤮}} \\ \text{AND} \\ \left[{}_{1234}\mathsf{HEIGHT}\right]_{i+1}^{\text{⤮}} = \left[{}_{1234}\mathsf{HEIGHT}\right]_{i}^{\text{⤮}} \end{cases}$$
   THEN
   i. $\left[{}_{1234}\mathsf{POP}\right]_{i+1}^{\text{⤮}} + \left[{}_{1234}\mathsf{POP}\right]_{i}^{\text{⤮}} = 1,$
   ii. IF $\left[{}_{1234}\mathsf{POP}\right]_{i+1}^{\text{⤮}} = 1$ THEN
   $$\begin{cases} \left[{}_{1234}\mathsf{VAL}^{\mathsf{hi}}\right]_{i+1}^{\text{⤮}} = \left[{}_{1234}\mathsf{VAL}^{\mathsf{hi}}\right]_{i}^{\text{⤮}} \\ \left[{}_{1234}\mathsf{VAL}^{\mathsf{lo}}\right]_{i+1}^{\text{⤮}} = \left[{}_{1234}\mathsf{VAL}^{\mathsf{lo}}\right]_{i}^{\text{⤮}} \end{cases}$$
   In other words, the binary flag column $\left[{}_{1234}\mathsf{POP}\right]^{\text{⤮}}$ at a given height oscillates (we push, pop, push, pop, push, etc...); when popping an item (i.e. when $\left[{}_{1234}\mathsf{POP}\right]_{i+1}^{\text{⤮}} = 1$), we retrieve the value previously pushed at that height;
   (b)
   $$\text{IF} \left\{ \begin{array}{c} \left[\mathsf{CN}^{\boxplus 4}\right]_{i+1}^{\text{⤮}} \neq \left[\mathsf{CN}^{\boxplus 4}\right]_{i}^{\text{⤮}} \\ \text{OR} \\ \left[{}_{1234}\mathsf{HEIGHT}\right]_{i+1}^{\text{⤮}} \neq \left[{}_{1234}\mathsf{HEIGHT}\right]_{i}^{\text{⤮}} \end{array} \right\} \text{THEN } \left[{}_{1234}\mathsf{POP}\right]_{i+1}^{\text{⤮}} = 0$$
   i.e. the first time we encounter a given (nonzero) height of a (nonzero) context it is to push an item at that height (not to pop a nonexisting item).

1.6.2 Program counter, PUSHes and JUMPs)

The PC is updated both in the temporal execution trace and in a reordered execution trace (to resume execution where it left off when executing a `CREATE`-type instruction or a `CALL`-type instruction.) The constraints below detail the "extraordinary" or "unusual" updates to the program counter induced by `PUSH`-type instructions and `JUMP`-type instructions. To this end we introduce a column PC^{ν} that stores the *expected new program counter*. This expected new program counter isn't necessarily the next value of the program counter. Indeed exceptions[8] may impose an abrupt execution context switch.

$$\boxed{\text{The constraints below are written under the assumption that } \mathsf{CN}_i \neq 0.}$$

1. $\textsc{if}\ {}^{\diamond}\mathsf{PUSH}\,\text{⚐}_i = 1\ \textsc{then}$

   (a) Put the push argument on stack:

   $$\begin{cases} {}_4\mathsf{VAL}^{\mathsf{hi}}{}_i = \langle \mathsf{PUSH\_VALUE}^{\mathsf{hi}} \rangle_i \\ {}_4\mathsf{VAL}^{\mathsf{lo}}{}_i = \langle \mathsf{PUSH\_VALUE}^{\mathsf{lo}} \rangle_i \end{cases}$$

   (b) We set the *expected* new program counter: $\mathsf{PC}^{\nu}_i = \mathsf{PC}_i + 1 + {}^{\diamond}\mathsf{PUSH\_PARAM}_i$.

2. $\textsc{if}\ {}^{\diamond}\mathsf{JUMP}\,\text{⚐}_i = 1$ the following sets the *expected* new program counter:

   (a) $\textsc{if}\ \langle \mathsf{INST} \rangle_i = \mathtt{JUMP}$[9] $\textsc{then}$

      i. We set the *expected* new program counter:

      $$\begin{cases} \textsc{if}\ \mathsf{JOOB}_i = 1\ \textsc{then}\ \mathsf{PC}^{\nu}_i = \mathsf{CODESIZE}_i \\ \textsc{if}\ \mathsf{JOOB}_i = 0\ \textsc{then}\ \mathsf{PC}^{\nu}_i = {}_1\mathsf{VAL}^{\mathsf{lo}}{}_i \end{cases}$$

      ii. If the jump is carried out (and not thwarted by an exception) we check its validity, i.e. $\textsc{if}\ \mathsf{CN}_{i+1} = \mathsf{CN}_i\ \textsc{then}$

      $$\begin{cases} \textsc{if}\ \langle \mathsf{INST} \rangle_{i+1} = \mathtt{JUMPDEST}\ \textsc{then}\ \mathsf{JUMPX}_{i+1} = 0 \\ \textsc{if}\ \langle \mathsf{INST} \rangle_{i+1} \neq \mathtt{JUMPDEST}\ \textsc{then}\ \mathsf{JUMPX}_{i+1} = 1 \end{cases}$$

   (b) $\textsc{if}\ \langle \mathsf{INST} \rangle_i = \mathtt{JUMPI}$[10] $\textsc{then}$

      i. We set the *expected* new program counter:

         A. $\textsc{if}\ \left({}_4\mathsf{VAL}^{\mathsf{hi}}{}_i = 0\ \ \textsc{and}\ {}_4\mathsf{VAL}^{\mathsf{lo}}{}_i = 0 \right)\ \textsc{then}\ \mathsf{PC}^{\nu}_i = 1 + \mathsf{PC}_i$ (no jump is triggered);

         B. $\textsc{if}\ \left({}_4\mathsf{VAL}^{\mathsf{hi}}{}_i \neq 0\ \ \textsc{or}\ {}_4\mathsf{VAL}^{\mathsf{lo}}{}_i \neq 0 \right)\ \textsc{then}$

         $$\begin{cases} \textsc{if}\ \mathsf{JOOB}_i = 1\ \textsc{then}\ \mathsf{PC}^{\nu}_i = \mathsf{CODESIZE}_i \\ \textsc{if}\ \mathsf{JOOB}_i = 0\ \textsc{then}\ \mathsf{PC}^{\nu}_i = {}_1\mathsf{VAL}^{\mathsf{lo}}{}_i \end{cases}$$

         furthermore, if the jump is carried out (and not thwarted by an exception) we check its validity, i.e. $\textsc{if}\ \mathsf{CN}_{i+1} = \mathsf{CN}_i\ \textsc{then}$

         $$\begin{cases} \textsc{if}\ \langle \mathsf{INST} \rangle_{i+1} = \mathtt{JUMPDEST}\ \textsc{then}\ \mathsf{JUMPX}_{i+1} = 0 \\ \textsc{if}\ \langle \mathsf{INST} \rangle_{i+1} \neq \mathtt{JUMPDEST}\ \textsc{then}\ \mathsf{JUMPX}_{i+1} = 1 \end{cases}$$

   Note that the JOOB flag is justified in the rare checks module.

[8] `outOfGas` or `stackUnderflow` for `JUMP`-type instructions, `outOfGas` or `stackOverflow` for `PUSH`-type instructions

[9] In implementation we should use "$\textsc{if}\ \langle \mathsf{INST} \rangle_i \neq \mathtt{JUMPI}$" instead.

[10] Similarly, use "$\textsc{if}\ \langle \mathsf{INST} \rangle_i \neq \mathtt{JUMP}$" instead.

3. Let us (*and just this once*) write $\mathsf{UPCU} = {}^{\diamond}\mathsf{PUSH}\,\text{⚐} + {}^{\diamond}\mathsf{JUMP}\,\text{⚐}$ where UPCU is shorthand for "Unusual Program Counter Update". Then we ask that

$$\text{IF } \mathsf{UPCU}_i = 0 \text{ THEN } \mathsf{PC}_i^{\nu} = 1 + \mathsf{PC}_i$$

We give some context as to the "$\mathsf{PC}_{i+1} = \mathsf{CODESIZE}_i$" constraint. First, remark that the $\mathsf{CODESIZE}$ is indeed available (it is a column in the call stack). Secondly, recall that in our padding of the bytecode in the ROM module, we always append at least 32 zero bytes (`0x00`) after the end of the bytecode; in case of an out of bounds jump the zk-evm jumps to $\mathsf{PC} = \mathsf{CODESIZE}$ and the associated opcode will be `0x00` (not a `JUMPDEST`.)

The correct retrieval of the context's program counter the reentrance into the current context after a `CALL`-type instruction or `CREATE`-type instruction is most easily expressed after reordering of the execution trace. To that effect, consider a reordering of the columns $\mathsf{X} \mapsto [\mathsf{X}]^{\text{⤮}}$ such that

$$\Big([\mathsf{CN}]^{\text{⤮}}, [\mathsf{STACK}\square]^{\text{⤮}} \Big) \ \equiv \ \text{lex. ordered}$$

We drop the $\mathsf{CN}_{i+1} \neq 0$ assumption and replace it with:

$$\boxed{\text{The constraints below are written under the assumption that } [\mathsf{CN}]_{i+1}^{\text{⤮}} \neq 0.}$$

1. IF $[\mathsf{CN}_{i+1}]^{\text{⤮}} = [\mathsf{CN}_i]^{\text{⤮}}$ THEN
$$[\mathsf{PC}_{i+1}]^{\text{⤮}} = [\mathsf{PC}_i^{\nu}]^{\text{⤮}}$$
2. IF $[\mathsf{CN}_{i+1}]^{\text{⤮}} \neq [\mathsf{CN}_i]^{\text{⤮}}$ THEN $[\mathsf{PC}_{i+1}]^{\text{⤮}} = 0$.

Note: we could just as well express the constraints for jump and push instructions in the standard time ordered version of the execution trace. This would be more economical and their expression would be *precisely the same*, just without the ordered columns. The (context, stamp) sorted version is useful for updating the program counter in a context switch, i.e. some variation of `CALL` or `CREATE`.

1.6.3 Miscellaneous flags

The VALTF

We specify the VALTF column (short for $\mathsf{VALUE\_TRANSFER\_FLAG}$). It is a binary flag which equals 1 *iff* the instruction is a `CALL`-type instruction which transfers value. Recall that for the `callPattern` the third stack item on the second row contains the value argument (if any) of the `CALL`-type instruction. With this in mind, VALTF is defined by

1. IF ${}^{\diamond}\mathsf{CALL}\,\text{⚐}_i = 0$ THEN $\mathsf{VALTF} = 0$
2. IF ${}^{\diamond}\mathsf{CALL}\,\text{⚐}_i = 1$ THEN
   (a) IF ${}_{3}\mathsf{VAL}^{\mathsf{lo}}{}_{i+1} = 0$ THEN $\mathsf{VALTF}_i = 0$
   (b) IF ${}_{3}\mathsf{VAL}^{\mathsf{lo}}{}_{i+1} \neq 0$ THEN $\mathsf{VALTF}_i = 1$

The ACCOUNT_HAS_BALANCE_FLAG

We specify the ACCHB flag. Its specification is simple:

$$\begin{cases} \text{IF } \mathsf{BALANCE}_i = 0 \text{ THEN } \mathsf{ACCHB}_i = 0 \\ \text{IF } \mathsf{BALANCE}_i \neq 0 \text{ THEN } \mathsf{ACCHB}_i = 1 \end{cases}$$

1.6.4 Gas

This section deals with the gas in the hub. Gas is a complex topic. Instructions come with a static gas cost which is instruction decoded from $\langle\mathsf{INST}\rangle$. Instructions may incur extra costs which are computed as a combination of the following we enumerate here:

Arithmetic. The `EXP` opcode incurs a dynamic cost $G_{\text{expbyte}} \cdot \mathsf{n}$ where n is 0 is the exponent e is 0, and $\lfloor \log_{256}(\mathsf{e}) \rfloor$ otherwise. This dynamic gas cost is made available in the $\mathsf{ALU\_DYNAMIC\_GAS}$ column (which is justified in the **ALU module**.)

Storage. `SLOAD` and `SSTORE` (especially) have complex pricing; the gas cost is computed in the storage module and made available in the STOG column (it is justified in the **storage module**.)

Memory Expansion. The memory expansion cost is made available in the $\Delta\mathsf{MXC}$ column (which is justified in the memory expansion module.)

Linear cost. Certain instructions charge an extra fee that is linear in a size argument. Complexity arises from the fact that these sizes may be measured in bytes or in EVM words. For the latter case the Hub contains a $\mathsf{SIZE\_IN\_EVM\_WORDS}$ column (which is justified in the **memory expansion module**.)

`CALL` costs `CALL`-type instructions come with extra costs not found elsewhere:

Transfer cost. `CALL`s which transfer funds cost $G_{\text{callvalue}} = 9000$ more.

Address warmth. `CALL`s to warm addresses cost less; warmth of an address is justified in the

The first requirement which we impose is that gas columns ought to be counter constant

1. IF $\mathsf{CT}_{i+1} \neq 0$[11] THEN
$$\begin{cases} \mathsf{GAS}^{\omega}_{i+1} = \mathsf{GAS}^{\omega}_{i} \\ \mathsf{GAS}^{\rho}_{i+1} = \mathsf{GAS}^{\rho}_{i} \\ \mathsf{GAS}^{\kappa}_{i+1} = \mathsf{GAS}^{\kappa}_{i} \\ \mathsf{GAS}^{\nu}_{i+1} = \mathsf{GAS}^{\nu}_{i} \end{cases}$$

As a consequence we impose gas to be computed once per instruction, precisely when $\mathsf{CT}_i = 0$. We therefore impose that

$$\boxed{\text{The remainder of this section is written under the assumption } \mathsf{CT}_i = 0.}$$

The hub computes gas as follows. From the point of view of the hub, the initial gas is imported from block data. It defines the first value of GAS^{ω} within a transaction. Every instruction induces gas depletion as follows:

$$\mathsf{GAS}^{\omega} \xrightarrow{(1)} \mathsf{GAS}^{\rho} \xrightarrow{(2)} \mathsf{GAS}^{\kappa} \xrightarrow{(3)} \mathsf{GAS}^{\nu}$$

Steps (1) and (2) could easily be combined into a single step (thus rendering the GAS^{ρ} column obsolete.)

Every time a halting instruction is executed which doesn't put an end to the transaction (but only switches from the current context to its parent context) the descendant context receives a gas refund. We thus impose the following constraints:

2. IF $\mathsf{TX\#}_i \neq 0$ AND $\mathsf{TX\#}_i = \mathsf{TX\#}_{i+1}$ THEN

$$\mathsf{GAS}^{\rho}_{i+1} = \mathsf{GAS}^{\omega}_{i+1} + {}^{\Diamond}\mathsf{HALT}\,⚐_i \cdot (1 - \mathsf{GENERAL\_EXCEPTION}_i) \cdot \mathsf{GAS}^{\nu}_i$$

In other words: if the previously executed instruction was a halting operation and it didn't trigger an exception the "old gas" of the parent context receives a refund which is equal to the descendant context's remaining gas

[11] I.e. IF $\mathsf{CT}_{i+1} = 1$

The second step is about subtracting static and dynamic gas costs:

3. IF $\mathsf{TX\#}_i \neq 0$ THEN

$$\mathsf{GAS}_i^{\kappa} = \mathsf{GAS}_i^{\rho} - {}^{\Diamond}\mathsf{STATG}_i \tag{1.1}$$

$$- {}^{\Diamond}\mathsf{ALU}\,⚐_i \cdot G_{\text{expbyte}} \cdot \mathsf{EXPONENT\_SIZE\_IN\_BYTES}_i \tag{1.2}$$

$$- {}^{\Diamond}\mathsf{STO}\,⚐_i \cdot \mathsf{STOG}_i \tag{1.3}$$

$$- {}^{\Diamond}\mathsf{WRM}\,⚐_i \cdot \begin{bmatrix} \mathsf{WARM}_i \cdot (1 - {}^{\Diamond}\mathsf{SELFDESTRUCT\_FLAG}_i) \cdot G_{\text{warmaccess}} \\ +(1 - \mathsf{WARM}_i) \cdot G_{\text{coldaccountaccess}} \end{bmatrix} \tag{1.4}$$

$$- {}^{\Diamond}\mathsf{CALL}\,⚐_i \cdot \mathsf{VALTF}_i \cdot \begin{bmatrix} \mathsf{DEAD\_FLAG}_i \cdot G_{\text{newaccount}} & (a) \\ +G_{\text{callvalue}} & (b) \end{bmatrix} \tag{1.5}$$

$$- {}^{\Diamond}\mathsf{MXP}\,⚐_i \cdot \Delta\mathsf{MXC}_i \tag{1.6}$$

$$- {}^{\Diamond}\mathsf{COPY}\,⚐_i \cdot G_{\text{copy}} \cdot \mathsf{SEVMW}_i \tag{1.7}$$

$$- {}^{\Diamond}\mathsf{HASH\_FLAG}_i \cdot G_{\text{keccak256word}} \cdot \mathsf{SEVMW}_i \tag{1.8}$$

$$- {}^{\Diamond}\mathsf{LOG}\,⚐_i \cdot G_{\text{logdata}} \cdot {}_3\mathsf{VAL}^{\mathsf{lo}}{}_i \tag{1.9}$$

$$- {}^{\Diamond}\mathsf{RETURN}\,⚐_i \cdot \mathsf{CTYPE}_i \cdot G_{\text{codedeposit}} \cdot {}_3\mathsf{VAL}^{\mathsf{lo}}{}_i \tag{1.10}$$

$$- {}^{\Diamond}\mathsf{SELFDESTRUCT\_FLAG}_i \cdot \mathsf{ACCHB}_i \cdot G_{\text{newaccount}} \cdot {}_3\mathsf{VAL}^{\mathsf{lo}}{}_i \tag{1.11}$$

We provide some details. (1) accounts for static gas; static gas is justified against the **instruction decoder** (as is evident from the $\Diamond$) (2) accounts for the dynamic gas cost associated with exponentiation; the $\mathsf{EXPONENT\_SIZE\_IN\_BYTES}$ column is justified in the **RAM module**; it is zero unless the instruction is `EXP` i.e. exponentiation $\mod 2^{256}$; (3) accounts for the dynamic gas cost of storage instructions; the STOG column is justified in the **storage module**; (4) accounts for costs associated with access costs of accounts; the WARM flag is justified in the **warmth module**; (5) accounts for extraordinary costs associated with `CALL`-type instructions; there is (a) the cost associated with `CALL`ing upon a non existent account (b) the cost associated with a value transfer; (6) accounts for memory expansion costs; the $\Delta\mathsf{MXC}$ column is justified in the **memory expansion module**; (7) accounts for "copy" instructions[12] which encur a linear cost in the number of evm words copied; (8) accounts for operations that hash a slice of memory[13] and incur a linear cost in the number of evm words hashed; (9) accounts for the portion of log pricing that is linear in the number of bytes to log; (10) accounts for code deployment costs: they are paid when encountering a `RETURN` instruction (i.e ${}^{\Diamond}\mathsf{RETURN}\,⚐ = 1$) in a deployment context (i.e. $\mathsf{CTYPE}_i = 1$.) Notations for gas constants (G_{expbyte} etc...) are taken from the Ethereum Yellow Paper.

The next step in the gas computation is to compute the "new gas". The main complication arises with `CALL`-type and `CREATE`-type instructions. The Gas modules exists precisely to justify the gas endowment in these cases:

4. IF $\mathsf{TX\#}_i \neq 0$ THEN

$$\begin{aligned} \mathsf{GAS}_i^{\nu} = \mathsf{GAS}_i^{\kappa} &- {}^{\Diamond}\mathsf{CALL}\,⚐ \cdot \mathsf{GAS}_i^{\varepsilon} \\ &- {}^{\Diamond}\mathsf{CREATE}\,⚐ \cdot \mathsf{GAS}_i^{\varepsilon} \end{aligned}$$

[12] i.e. `RETURNDATACOPY`, `CALLDATACOPY`, `CODECOPY` and `EXTCODECOPY`

[13] i.e. `SHA3` and `CREATE2`

1.7 Workflow

1.7.1 Module selectors

Stamp counter-constancy constraints

Recall that a column X is counter-constant if $\mathsf{CT}_i \neq 0 \implies \mathsf{X}_i = \mathsf{X}_{i-1}$, see section 1.2.2. We impose counter-constancy constraints on "module stamp" columns:

1. $\mathsf{ACC}\square$
2. $\mathsf{ALU}\square$
3. $\mathsf{BIN}\square$
4. $\mathsf{EXP}\square$
5. $\mathsf{GAS}\square$
6. $\mathsf{KEC}\square$
7. $\mathsf{LOG}\square$
8. $\mathsf{MMU}\square$
9. $\mathsf{MXP}\square$
10. $\mathsf{OOB}\square$
11. $\mathsf{SHV}\square$
12. $\mathsf{STO}\square$
13. $\mathsf{WCP}\square$
14. $\mathsf{WRM}\square$

These constraints matter for $^{\diamond}\mathsf{TWO_LINE_INSTRUCTION}$s: a single instruction should be dispatched to the relevant modules. This is also the reason why in the following section we state all constraints under the "$\mathsf{CT}_i = 0$" hypothesis:

$$\boxed{\text{Throughout subsection 1.7.1 we systematically assume that } \mathsf{CT}_i = 0.}$$

`stackException` sensitive selectors

The **exponent module**, the **out of bounds module**, and the **storage module** are triggered *iff* (1) the stack raises no `stackException` and (2) the instruction raises the appropriate module flag. In other words:

$$\begin{cases} \mathsf{EXP}\lightning_i &= (1 - \mathsf{STX}_i) \cdot {}^{\diamond}\mathsf{EXP}\,⚐_i \\ \mathsf{OOB}\lightning_i &= (1 - \mathsf{STX}_i) \cdot {}^{\diamond}\mathsf{OOB}\,⚐_i \\ \mathsf{STO}\lightning_i &= (1 - \mathsf{STX}_i) \cdot {}^{\diamond}\mathsf{STO}\,⚐_i \end{cases}$$

and the associated module stamps are updated accordingly:

$$\begin{cases} \mathsf{EXP}\square_i &= \mathsf{EXP}\square_{i-1} + \mathsf{EXP}\lightning_i \\ \mathsf{OOB}\square_i &= \mathsf{OOB}\square_{i-1} + \mathsf{OOB}\lightning_i \\ \mathsf{STO}\square_i &= \mathsf{STO}\square_{i-1} + \mathsf{STO}\lightning_i \end{cases}$$

The inclusion of the storage module in this list may seem surprising. One would expect the storage module to only be triggered if both previously stated conditions hold *and the instruction raises no out of gas exception*. However the storage module is unique among all "instruction executing modules[14]" (other than the hub itself) in that it computes its own gas cost. `SSTORE` pricing in particular is complex and closely connected with the storage operation itself so we have chosen to do both at the same time and in the same place. It should be added that this doesn't introduce undesirable modifications to storage: the storage module is **self-reverting**. Thus any storage operation carried out by the storage module which induces an out of gas exception in the hub will be done (in storage) in such a way as to revert itself.

[14] i.e. ALU, binary, mmu and ram, word comparison

`stackException` and `callStackOverflowException` sensitive selectors

The **address shaving module**, the **memory expansion module** and the **warmth module** are triggered *iff* (1) the stack raises no `stackException`, (2) the instruction raises no `callStackOverflowException`, and (3) the instruction raises the relevant module flag. In other words:

$$\left\{\begin{array}{rcl}
\mathsf{MXP}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{CSDX}_i)\cdot {}^{\diamond}\mathsf{MXP}\,\text{⚐}_i \\
\mathsf{SHV}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{CSDX}_i)\cdot {}^{\diamond}\mathsf{SHV}\,\text{⚐}_i \\
\mathsf{WRM}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{CSDX}_i)\cdot {}^{\diamond}\mathsf{WRM}\,\text{⚐}_i
\end{array}\right.$$

and the associated module stamps are updated accordingly:

$$\left\{\begin{array}{rcrcr}
\mathsf{MXP}\,\square_i & = & \mathsf{MXP}\,\square_{i-1} & + & \mathsf{MXP}\,\lightning_i \\
\mathsf{SHV}\,\square_i & = & \mathsf{SHV}\,\square_{i-1} & + & \mathsf{SHV}\,\lightning_i \\
\mathsf{WRM}\,\square_i & = & \mathsf{WRM}\,\square_{i-1} & + & \mathsf{WRM}\,\lightning_i
\end{array}\right.$$

`stackException` and `outOfGasException` sensitive selectors

The **ALU module**, the **binary module**, the **word comparison module** and the **hash info module** are triggered *iff* (1) the stack raises no `stackException`, (2) the instruction raises no `outOfGasException`, and (3) the instruction raises the relevant module flag. In other words:

$$\left\{\begin{array}{rcl}
\mathsf{ALU}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{OOGX}_i)\cdot {}^{\diamond}\mathsf{ALU}\,\text{⚐}_i \\
\mathsf{BIN}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{OOGX}_i)\cdot {}^{\diamond}\mathsf{BIN}\,\text{⚐}_i \\
\mathsf{KEC}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{OOGX}_i)\cdot {}^{\diamond}\mathsf{KEC}\,\text{⚐}_i \\
\mathsf{WCP}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{OOGX}_i)\cdot {}^{\diamond}\mathsf{WCP}\,\text{⚐}_i
\end{array}\right.$$

and the associated module stamps are updated accordingly:

$$\left\{\begin{array}{rcrcr}
\mathsf{ALU}\,\square_i & = & \mathsf{ALU}\,\square_{i-1} & + & \mathsf{ALU}\,\lightning_i \\
\mathsf{BIN}\,\square_i & = & \mathsf{BIN}\,\square_{i-1} & + & \mathsf{BIN}\,\lightning_i \\
\mathsf{KEC}\,\square_i & = & \mathsf{KEC}\,\square_{i-1} & + & \mathsf{KEC}\,\lightning_i \\
\mathsf{WCP}\,\square_i & = & \mathsf{WCP}\,\square_{i-1} & + & \mathsf{WCP}\,\lightning_i
\end{array}\right.$$

LOG module selector

The **log-info module** is triggered *iff* (1) the stack raises no `stackException` (2) the context isn't static (3) the instruction doesn't lead to an `outOfGasException` (4) the instruction raises the "log flag." In other words:

$$\left\{\begin{array}{rcl}
\mathsf{LOG}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{CSTAT}_i)\cdot(1-\mathsf{OOGX}_i)\cdot {}^{\diamond}\mathsf{LOG}\,\text{⚐}_i \\
\mathsf{LOG}\,\square_i & = & \mathsf{LOG}\,\square_{i-1}+\mathsf{LOG}\,\lightning_i
\end{array}\right.$$

Gas module selector

The **gas module**, triggers *iff* (1) the stack raises no `stackException`, (2) the instruction raises no `callStackOverflowException` and (3) the instruction is a `CALL`-type instruction, a `CREATE`-type instruction, a halting instruction, the instruction raises an `outOfGasException` or the instruction raises a `generalException`.

In other words if we write *just this once*

$$\mathsf{GAS\_TRIGGER}_i = \big(1-\mathsf{STX}_i\big)\cdot\big(1-\mathsf{CSDX}_i\big)\cdot\left[\begin{array}{cr}
 & \mathsf{OOGX}_i \\
+ & \mathsf{GENX}_i \\
+ & {}^{\diamond}\mathsf{HALT}\,\text{⚐}_i \\
+ & {}^{\diamond}\mathsf{CALL}\,\text{⚐}_i \\
+ & {}^{\diamond}\mathsf{CREATE}\,\text{⚐}_i
\end{array}\right]$$

We then set

$$\left\{\begin{array}{lcl} \multicolumn{3}{l}{\begin{cases} \text{IF } \mathsf{GAS\_TRIGGER}_i \neq 0 \text{ THEN } \mathsf{GAS}\,\lightning_i = 1 \\ \text{IF } \mathsf{GAS\_TRIGGER}_i = 0 \text{ THEN } \mathsf{GAS}\,\lightning_i = 0 \end{cases}} \\ \mathsf{LOG}\,\lightning_i & = & (1-\mathsf{STX}_i)\cdot(1-\mathsf{CSTAT}_i)\cdot(1-\mathsf{OOGX}_i)\cdot{}^{\diamond}\mathsf{LOG}\,\text{⚐}_i \\ \mathsf{LOG}\,\square_i & = & \mathsf{LOG}\,\square_{i-1} + \mathsf{LOG}\,\lightning_i \end{array}\right.$$

Note: we really only need GENX … including OOGX is done purely for mental comfort. Note furthermore that the gas module imports OOGX.

MMU module selector

The trigger for the **MMU module** is by far the most complex trigger. The conditions that trigger a call to the MMU module are (1) the stack raises no `stackException` (2) the instruction doesn't lead to an `outOfGasException` (3) a host of instruction dependent conditions which we will describe after giving the selector expression. Thus the MMU selector is defined by the constraint

$$\mathsf{MMU}\,\lightning_i = \underbrace{(1-\mathsf{STX}_i)\cdot(1-\mathsf{OOGX}_i)}_{(0)}\cdot\left[\begin{array}{cr} & \mathsf{STD}_i \\ + & {}^{\diamond}\mathsf{REVERT}\,\text{⚐}_i\cdot\left[\mathsf{CSD}\neq 1\right]_i \\ + & {}^{\diamond}\mathsf{RETURN}\,\text{⚐}_i\cdot\left[\begin{array}{r}\mathsf{CTYPE}_i\cdot\left[\mathsf{CSD}=1\right]_i \\ +\left[\mathsf{CSD}\neq 1\right]_i\end{array}\right] \\ + & {}^{\diamond}\mathsf{LOG}\,\text{⚐}_i\cdot(1-\mathsf{CSTAT}_i) \\ + & {}^{\diamond}\mathsf{CDL}\,\text{⚐}_i\cdot(1-\mathsf{CDL\_OOB}_i) \\ + & {}^{\diamond}\mathsf{RDC}\,\text{⚐}_i\cdot(1-\mathsf{RDCX}_i) \\ + & {}^{\diamond}\mathsf{CREATE}\,\text{⚐}_i\cdot(1-\mathsf{CSDX}_i)\cdot(1-\mathsf{CSTAT}_i) \\ + & {}^{\diamond}\mathsf{CALL}\,\text{⚐}_i\cdot(1-\mathsf{CSDX}_i)\cdot(1-\mathsf{CSTAT}_i\cdot\mathsf{VALTF}_i) \end{array}\right]\quad\begin{array}{c}(1)\\(2)\\(3)\\(4)\\(5)\\(6)\\(7)\\(8)\end{array}$$

where we have used the following short hands:

$$\mathsf{STD} = {}^{\diamond}\mathsf{MMU}\,\text{⚐} - {}^{\diamond}\mathsf{RETURN}\,\text{⚐} - {}^{\diamond}\mathsf{REVERT}\,\text{⚐} - {}^{\diamond}\mathsf{CDL}\,\text{⚐} - {}^{\diamond}\mathsf{LOG}\,\text{⚐} - {}^{\diamond}\mathsf{CREATE}\,\text{⚐} - {}^{\diamond}\mathsf{CALL}\,\text{⚐}$$

and $\left[\mathsf{CSD}=1\right] = 1-\left[\mathsf{CSD}\neq 1\right]$ is the binary flag defined by $\left[\mathsf{CSD}=1\right]_i = 1 \iff \mathsf{CSD}_i = 1$.

We provide some details: (0) filters out instructions that produce a `stackException` or an `outOfGasException`; (1) STD is (by construction) a binary column; it lights up precisely for `MLOAD`, `MSTORE`, `MSTORE8`, `SHA3`, `CODEDATACOPY`, `EXTCODEDATACOPY`, `CALLDATACOPY`; thus any of these instructions which passes the “stack and gas hurdle” makes it to the MMU; (2) filters out `REVERT`s in the root context of a transaction; (3) does the same for `RETURN`s except if the root context is a deployment context (i.e. if the transaction is a “deployment transaction”); (4) filters out `LOG`-type instructions in “static” execution contexts; (5) filters out `CALLDATALOAD` instructions that raise the $\mathsf{CDL\_OOB}$ flag[15] (6) is more serious: it filters out `RETURNDATACOPY` instructions that raise the `returnDataCopyException`; (7) filters out `CREATE(2)` instructions at call stack depth = 1024 aswell as attempts to run such an instruction in a static execution context; (8) filters out `CALL`-type instructions at call stack depth = 1024 and attempts to transfer funds in a call when the execution context is static. **Note: we may not do the filtering of `CALLDATALOAD`s at the hub level: we can do it in the MMU by doing “no-op” filtering there.**

[15] recall that this flag signifies that the requested evm word is fully out of bounds of the current context's call data; it is justified in the out of bounds module;

Chapter 2

MMU

2.1 Column descriptions

It is understood that whenever we write “$\langle \mathsf{X} \rangle$ is the import of the X column” that, in reality, it is the import of $\mathsf{X} \cdot \mathsf{b}_{\mathsf{RAM}}$ where $\mathsf{b}_{\mathsf{RAM}}$ is a binary column which equals 1 *iff (a)* no exception occurs at that row and *(b)* the instruction is one that touches memory. The binary column $\mathsf{b}_{\mathsf{RAM}}$ is thus obtained as the product of an instuction decoded column which detects RAM instructions and a binary column which detects exceptions.

1. $\langle \mathsf{MMU\_STAMP} \rangle$: imported column containing the RAM stamp; abbreviated to $\langle \mathsf{MMU}\square \rangle$;
2. $\mu\mathsf{INSTRUCTION\_STAMP}$: column containging the micro instruction stamp; abbreviated to $\mu\mathsf{INST}\square$;
3. $\mathsf{IS\_MICRO\_INSTRUCTION}$: binary flag that equals zero during the precomputation phase and equals to 1 for rows containing micro instructions; abbreviated to IS_μ;
4. $\mathsf{TOTAL\_NUMBER\_OF\_MICRO\_INSTRUCTIONS}$: established during the precomputation phase; contains the total number of micro instructions the current macro instruction is converted to; abbreviate to TOT^{μ};

TOT^{μ} is constant while $\mathsf{IS}\_\mu = 0$, decreasing until it hits 0 while $\mathsf{IS}\_\mu = 1$. It hitting 0 signifies the final micro instruction in the sequence of micro instructions the macro instruction decomposes into.

5. $\langle \mathsf{OFF}^1 \rangle$: import of the ${}_1\mathsf{VAL}^{\mathsf{lo}}$ column; contains the first offset;
6. $\langle \mathsf{OFF}^2 \rangle^{\mathsf{hi}}$: import of the ${}_2\mathsf{VAL}^{\mathsf{hi}}$ column; contains a potential second offset;
7. $\langle \mathsf{OFF}^2 \rangle^{\mathsf{lo}}$: import of the ${}_2\mathsf{VAL}^{\mathsf{lo}}$ column; contains a potential second offset;
8. $\langle \mathsf{SIZE} \rangle$: import of the ${}_3\mathsf{VAL}^{\mathsf{lo}}$ column; contains a size (including code sizes);
9. $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$: import of the ${}_4\mathsf{VAL}^{\mathsf{hi}}$, ${}_4\mathsf{VAL}^{\mathsf{lo}}$ columns;

Note that we have given these imported columns suggestive names rather than, say, $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle$ etc... We do this purely for improved readability. As suggested by their names, the $\langle \mathsf{OFF}^1 \rangle$ and $(\langle \mathsf{OFF}^2 \rangle^{\mathsf{hi}}, \langle \mathsf{OFF}^2 \rangle^{\mathsf{lo}})$ will always contain offset arguments, $\langle \mathsf{SIZE} \rangle$ will always contain a size argument, while $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$ will always contain either a value loaded from RAM or call data, a value to store in RAM or an address. Note that we *don’t* import ${}_1\mathsf{VAL}^{\mathsf{hi}}$: if the instruction makes it to the RAM preprocessor its destination offset must be small (i.e. a 3 byte integer.) Note that we *do* import ${}_2\mathsf{VAL}^{\mathsf{hi}}$: the second offset (which points to either return data, call data or bytecode) can’t produce memory expansion, hence it won’t have been tested for smallness. The current module however must test for its size, hence the import.

10. ⟨CN⟩: imported column; contains the current execution context number;

11. ⟨CALLER⟩: imported column; contains the current caller execution context number;

12. ⟨RETURNER⟩: imported column; contains the current returner execution context number;

13. CONTEXT_SOURCE: column containing the execution context number of the source context; abbreviated to CN_S

14. CONTEXT_TARGET: column containing the execution context number of the target context; abbreviated to CN_T

15. COUNTER: counter column; in the precomputation phase counts either from 0 to 2 or from 0 to 15; in the rows containing micro instruction equals to 0; abbreviated to CT;

16. OFFSET_OUT_OF_BOUNDS: binary column; can only light up for code copy and call data copy instructions; signifies when an "source offset" is large; abbreviated to OFF_OOB;

17. $^\diamond$PRECOMPUTATION: instruction decoded column that indicates the precomputation type associated with a given parametrized instruction; abbreviated to $^\diamond$PRE;

"Source offsets" associated with code copy and call data instructions don't get tested for smallness (i.e. their ability to fit into 3 bytes): the Memory Expansion Module ignores them since they don't induce memory expansion. The OFF_OOB binary flag lights up as soon as the relevant offset (⟨OFF^2⟩ for `CODECOPY`, `EXTCODECOPY`, `CALLDATACOPY` instructions ⟨OFF^1⟩ for `CALLDATALOAD`) is $\geq$ the reference size ⟨REFS⟩ (which is ether the code size or the call data size.)

We now list some columns that will be passed down to the RAM data processor. These are **limb offset** and **byte offset** columns. They typically contain the quotient and remainder of the euclidean division of some *absolute* offset by 16. These values need to be justified, hence the inclusion of byte and prefix (i.e. accumulator) columns that provide the respective (short) byte decompositions.

18. SOURCE_LIMB_OFFSET: abbreviated to SLO;

19. SOURCE_BYTE_OFFSET: contains a number in the range $\{0, 1, \dots, 15\}$; abbreviated to SBO;

20. TARGET_LIMB_OFFSET: abbreviated to TLO;

21. TARGET_BYTE_OFFSET: contains a number in the range $\{0, 1, \dots, 15\}$; abbreviated to TBO;

22. NIB_1, NIB_2, NIB_3, NIB_4, NIB_5, NIB_6: nibble columns; typically contain the remainder of a euclidean division by 16 or some expression constructed from two such remainders;

23. BYTE_1, BYTE_2, BYTE_3, BYTE_4, BYTE_5, BYTE_6, BYTE_7, BYTE_8: byte columns;

24. ACC_1, ACC_2, ACC_3, ACC_4, ACC_5, ACC_6, ACC_7, ACC_8: "accumulator" columns;

The accumulator columns "accumulate" the bytes of some byte decomposition. The value whose bytes are being accumulated will typically be the quotient of some euclidean division by 16, e.g. that of some offset, some size parameter, some offset plus size parameter, or some adjusted nonnegative difference, etc… If OFF_OOB $= 0$ it targets a 3 byte integer; if OFF_OOB $= 1$ it targets 16 byte integers;

25. ⟦1⟧, ⟦2⟧, …, ⟦8⟧: ⟨MMU▢⟩-constant bit columns;

26. ALIGNED: ⟨MMU▢⟩-constant bit column; indicates whether certain offsets are aligned and hence whether certain micro-instructions *may* be done fast by the RAM data processor, i.e. without resorting to byte decompositions;

27. FAST: ⟨MMU▢⟩-constant bit column; indicates whether a micro-instructions *will* be done fast by the RAM data processor, i.e. without resorting to byte decompositions; it is completely determined by the micro-instruction;

28. MIN: column which may at times contain the "real size" of certain macro instructions; such a real size may is typically computed as a minimum between context data (e.g. CALLDATA_SIZE or CODESIZE) and a size stack argument;

29. TERNARY: $\langle \mathsf{MMU}\square \rangle$-constant ternary column i.e. it takes values in $\{0, 1, 2\}$; abbreviated to TERN;

2.2 Offset preprocessing

2.2.1 Absolute and relative offsets

In our arithmetization of memory, offsets can be **absolute** or **relative**. Thus when the RAM preprocessor imports two offset columns from the stack, $\langle \mathsf{OFF}^1 \rangle$ and $\langle \mathsf{OFF}^2 \rangle$, the interpretation of these offsets depends on the current instruction.

Absolute offsets. An execution context's RAM is a word addressable byte array. As such every byte has an **absolute position** within an execution context's memory. Offset arguments that refer to a position within the current execution context's RAM, i.e.

1. the offset argument $\langle \mathsf{OFF}^1 \rangle$ of `MLOAD`, `MSTORE`, `MSTORE8`,
2. the "destination" offset argument $\langle \mathsf{OFF}^1 \rangle$ to any `CODECOPY`-type instructions,
3. the "destination" offset argument $\langle \mathsf{OFF}^1 \rangle$ to `CALLDATACOPY` and `RETURNDATACOPY`,
4. the offset argument $\langle \mathsf{OFF}^1 \rangle$ of any `LOG`-type instructions,
5. the offset argument $\langle \mathsf{OFF}^1 \rangle$ of `SHA3`,
6. the offset argument $\langle \mathsf{OFF}^1 \rangle$ of `CREATE` and `CREATE2`,
7. the offset argument $\langle \mathsf{OFF}^1 \rangle$ of `RETURN` and `REVERT`

are *absolute*.

Relative offsets. When the current execution context $\mathscr{C}$ executes (without raising an exception) on a `CALL`-type instruction it spawns a descendant context $\mathscr{D}$. At the same time, the zk-evm fixes, once and for all, some immutable characteristics of that descendant context $\mathscr{D}$. For instance, it fixes its CALLER context number[1]. It also fixes $\mathscr{D}'s$ CALLDATA_OFFSET and CALLDATA_SIZE parameters. These are taken straight from the `CALL`-type instruction's 6 or 7 stack arguments, namely the offset and size parameters which define the call data.

Any access to call data (i.e. `CALLDATALOAD`, `CALLDATACOPY` but also `CALLDATASIZE`) performed while executing within the execution context $\mathscr{D}$ uses one or both of these execution context characteristics. In particular, the first offset parameter of `CALLDATALOAD` and the second offset parameter of `CALLDATACOPY` must be interpreted as *offsets within $\mathscr{C}$'s RAM relative to* CALLDATA_OFFSET. Accordingly, the *read* operations that these two instructions require take place in $\mathscr{C}$'s RAM using the *absolute*offsets one imagines:

- $\mathsf{CALLDATA\_OFFSET} + \langle \mathsf{OFF}^1 \rangle$ for `CALLDATALOAD`;
- $\mathsf{CALLDATA\_OFFSET} + \langle \mathsf{OFF}^2 \rangle$ for `CALLDATACOPY`;

[1] it is the context number of $\mathscr{C}$

We note at this point that a given execution context's RAM is immutable while the zk-evm is executing in a different execution context. Thus, resuming our previous discussion, $\mathscr{D}$'s call data (actually, $\mathscr{C}$'s RAM as a whole) is immutable while execution is taking place in $\mathscr{D}$ (or any of $\mathscr{C}$'s descendant contexts.) Applying changes to $\mathscr{C}$'s RAM requires first resuming execution of $\mathscr{C}$ which in turn requires exiting $\mathscr{D}$ for good.

Similarly, when $\mathscr{D}$ exits (gracefully or not) it endows $\mathscr{C}$ with (potentially emtpy) **return data**. In the zk-evm this is achieved by fixing some mutable characteristics of the parent context $\mathscr{C}$. Thus $\mathscr{C}$ is assigned a (new) $\mathsf{RETURNER}$ context number[2]. Furthermore, $\mathscr{C}$ is assigned $\mathsf{RETURNDATA\_OFFSET}$ and $\mathsf{RETURNDATA\_SIZE}$ parameters. These are zero by default unless $\mathscr{D}$ exits gracefully with a data-returning halt operation[3], in which case they are the two stack arguments to the `RETURN` or `REVERT` instruction that conclude $\mathscr{D}$'s execution.

In close analogy to the call data case, any access to return data (i.e. `RETURNDATACOPY` and `RETURNDATASIZE`) performed while executing within the execution context $\mathscr{C}$ uses one or both of the $\mathsf{RETURNDATA\_OFFSET}$ and $\mathsf{RETURNDATA\_SIZE}$ parameters. Thus the second offset parameter of `RETURNDATACOPY` is interpreted by the zk-evm as an *offset within $\mathscr{D}$'s RAM relative to* $\mathsf{RETURNDATA\_OFFSET}$. Accordingly, all *read* operations this requires take place in $\mathscr{D}$'s RAM starting at the *absolute* offset $\mathsf{RETURNDATA\_OFFSET} + \langle \mathsf{OFF}^2 \rangle$.

2.2.2 RAM constancy

We say that a column X is **stamp-constant** if it satisfies:

$$\langle \mathsf{MMU}\square \rangle_{i+1} = \langle \mathsf{MMU}\square \rangle_i \implies \mathsf{X}_{i+1} = \mathsf{X}_i.$$

All imported columns are automatically $\langle \mathsf{MMU}\square \rangle$-constant. We further ask that the following constants be $\langle \mathsf{MMU}\square \rangle$-constant

1. $\mathsf{CN\_S}$ and $\mathsf{CN\_T}$
2. $\mathsf{OFF\_OOB}$
3. *all* the nibble columns,
4. *all* the bit columns $[\![1]\!]$, ..., $[\![8]\!]$.

2.2.3 Columns established during precomputation

Some columns remain constant as long as we are in the precomputation phase. We say that a column X is **established in precomputation** if it satisfies:

$$\begin{cases} \mathsf{IS\_}\mu_{i+1} = 0 \\ \quad \text{AND} \\ \mathsf{IS\_}\mu_i = 0 \end{cases} \implies \mathsf{X}_{i+1} = \mathsf{X}_i$$

The precomputation phase (which is characterized by $\mathsf{IS\_}\mu \equiv 0$) of a macro instruction spans 3 or 16 rows depending on the binary column $\mathsf{OFF\_OOB}$. Columns that are established during precomputation are constant during precomputation. The general principle is that these columns are "vetted" during that phase and serve as micro-instruction-flow defining parameters in the micro-instruction writing phase (which is characterized by $\mathsf{IS\_}\mu \equiv 1$.) Examples include (columns containing the) quotients and remainders of euclidean divisions. These are typically euclidean divisions of offsets and sizes by 16. The following columns are established in precomputation:

[2] it is, unsurprisingly, the context number of $\mathscr{D}$.

[3] i.e. through a `REVERT` instruction that doesn't raise a memory expansion exception or a `RETURN` instruction with similar restrictions if $\mathscr{D}$ isn't a deployment context.

1. SLO and SBO
2. TLO and TBO
3. QUOT^1 and QUOT^2
4. NIB_1 and NIB_2
5. TOT^μ

Once the preprocessing exits the precomputation phase and enters the micro-instruction writing phase (which is characterized by $\mathsf{IS\_}\mu \equiv 1$) these columns may start changing. Some may increase / decrease by 1 with every successive row. This is typically the case for quotient columns which will become limb offsets in the RAM data processor. Operations that span multiple limbs will typically see their limb offsets grow by one with every successive micro-instruction (though there are exceptions). The TOT^μ column obeys this logic perfectly: it decreases by one with every micro instruction until it hits 0.

Established columns are completely reset with every new macro-instruction.

2.2.4 Binary, ternary, nibble and byte columns

The following columns are binary columns, i.e. they are columns X satisfying for for all i, $\mathsf{X}_i \cdot (1 - \mathsf{X}_i) = 0$:

1. ALIGNED
2. $\llbracket 1 \rrbracket$,
3. $\llbracket 2 \rrbracket$,
4. $\llbracket 3 \rrbracket$,
5. $\llbracket 4 \rrbracket$,
6. $\llbracket 5 \rrbracket$,
7. $\llbracket 6 \rrbracket$
8. $\mathsf{IS\_}\mu$

We ask that the following columns contain bytes (i.e. integers in the range $\{0, 1, \dots, 255\}$):

1. BYTE_1
2. BYTE_2
3. BYTE_3
4. BYTE_4
5. BYTE_5
6. BYTE_6
7. BYTE_7
8. BYTE_8

We ask that the following columns contain nibbles (i.e. integers in the range $\{0, 1, \dots, 15\}$):

1. NIB_1
2. NIB_2
3. NIB_3
4. NIB_4
5. NIB_5
6. NIB_6

2.2.5 Heartbeat

The heartbeat of the RAM preprocessor is more complex than that of most other modules. The job of the preprocessor is to decompose RAM **macro-instructions** into a series of RAM **micro-instructions**. This task is decomposed into two phases:

1. precomputation: 3 or 16 rows;
2. micro-instruction writing: arbitrary number of rows;

The precomputation does all the offsets related byte decompositions required to decide on the micro instruction flow. Most of the time offsets and sizes have already been checked for smallness by the Memory Expansion Module. For such instructions computing the requisite euclidean divisions and comparison can be done in **3 rows**.

However, offsets that point within call data or bytecode haven't been checked for smallness up to this point: we have had no reason to do so as they can't induce memory expansion. Recall that if they are too large (i.e. exceed the call data size or code size) the instruction will simply write SIZE many 0's into memory. Smallness for offsets that point to return data, while also incapable of producing memory expansion, is tested in a separate module. This module also test for max code size constraints. `RETURNDATACOPY` and `RETURN` instructions in a deployment context whose maximal offset excedes $\mathsf{RETURNDATA\_SIZE}$[4] or the $\mathsf{CODESIZE}$ parameter[5] don't make it to the RAM preprocessor in the first place. This smallness check is required. This check requires a byte decomposition of integers of that fit into $\leq 16 \cdot 8 + 1 = 129$ bits.

There is thus a nondeterministic bit $\mathsf{OFF\_OOB}$ that indicates whether offsets overshoot CDS or $\mathsf{MaxCodeSize}$. And so depending on this nondeterministic bit the precomputation phase for `CALLDATACOPY`, `CALLDATALOAD`, as well as `CODECOPY` and `EXTCODECOPY` instructions, may require **16 rows**[6].

The second phase concerns the micro-instruction writing per se. Deciding upon the order of operations is straightforward in theory but tricky when expressed in terms of contraints, we shall not dwell on it here. Suffice it to say that a given macro-instruction may decompose into an arbirary (though small) number of micro-instructions TOT^{μ}.

Both of these phases are required to process a single RAM-macro-instruction. These two phases dictate the heartbeat of the module.

1. $\langle\mathsf{MMU}\square\rangle$ is nondecreasing in the sense that $\forall i,\ \langle\mathsf{MMU}\square\rangle_{i+1} \in \{\langle\mathsf{MMU}\square\rangle_i, 1 + \langle\mathsf{MMU}\square\rangle_i\}$;

2. $\langle\mathsf{MMU}\square\rangle_0 = 0$;

3. IF $\langle\mathsf{MMU}\square\rangle_i = 0$ THEN the entire i-th row is null; in particular the first row is all zeros;

4. IF $\langle\mathsf{MMU}\square\rangle_{i+1} \neq \langle\mathsf{MMU}\square\rangle_i$ THEN

   (a) $\mathsf{IS\_}\mu_{i+1} = 0$;

   (b) $\mathsf{CT}_{i+1} = 0$;

   (c) $\mathsf{TOT}^{\mu}_{i+1} \neq 0$;

   Regarding the constraint on TOT^{μ}_{i+1}: instructions that make it to the RAM preprocessing *always* require at least one micro-instruction to process. Operations with size 0 for instance or which raise an exception are filtered out and don't make it to the preprocessor.

5. IF $\langle\mathsf{MMU}\square\rangle_i \neq 0$ THEN

   (a) IF $\mathsf{IS\_}\mu_i = 0$ THEN

      i. IF $\mathsf{OFF\_OOB}_i = 0$ THEN

         A. IF $\mathsf{CT}_i \neq 2$ THEN
         $$\begin{cases} \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \\ \mathsf{IS\_}\mu_{i+1} = 0 \end{cases}$$

         B. IF $\mathsf{CT}_i = 2$ THEN $\mathsf{IS\_}\mu_{i+1} = 1$

      ii. IF $\mathsf{OFF\_OOB}_i = 1$ THEN

         A. IF $\mathsf{CT}_i \neq 15$ THEN
         $$\begin{cases} \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \\ \mathsf{IS\_}\mu_{i+1} = 0 \end{cases}$$

         B. IF $\mathsf{CT}_i = 15$ THEN $\mathsf{IS\_}\mu_{i+1} = 1$

6. IF $\mathsf{IS\_}\mu_i = 1$ THEN $\mathsf{CT}_i = 0$

7. IF $\langle\mathsf{MMU}\square\rangle_{i+1} = \langle\mathsf{MMU}\square\rangle_i$ THEN ${\mathsf{TOT}^{\mu}}_{i+1} = {\mathsf{TOT}^{\mu}}_i - \mathsf{IS\_}\mu_{i+1}$;

[4] i.e. $\mathsf{OFF} + \mathsf{SIZE} \geq \mathsf{RDS}$

[5] i.e. $\mathsf{SIZE} > 24576$

[6] We will want provide a byte decomposition for the quotient of the euclidean division of a 129 bit integer by 16, so the result fits into 16 bytes.

In other words, during the precomputation phase TOT^{μ} remains constant and in the micro-instruction writing phase it decreases by one with every row. The first part we already imposed (when asking that TOT^{μ} be established during precomputation) but the second part is new.

8. IF $\Big(\mathsf{IS\_}\mu_i = 1 \;$ AND $\; \mathsf{TOT}^{\mu}_i \neq 0\Big)$ THEN $\mathsf{IS\_}\mu_{i+1} = 1$;

9. IF $\Big(\langle \mathsf{MMU}\square \rangle_i \neq 0 \;$ AND $\; \mathsf{TOT}^{\mu}_i = 0\Big)$ THEN $\langle \mathsf{MMU}\square \rangle_{i+1} = 1 + \langle \mathsf{MMU}\square \rangle_i$

We can also settle the behaviour of $\mu\mathsf{INSTRUCTION\_STAMP}$:

10. $\forall i,\ \mu\mathsf{INST}\square_{i+1} = \mu\mathsf{INST}\square_i + \mathsf{IS\_}\mu_{i+1}$

It is similar to TOT^{μ} in that it is (technically) established during precomputation but there is no *actual* establishing happening: $\mu\mathsf{INST}\square$ just grows monotonically with every row counting the micro-instructions. There is no resetting it in the trace.

The following illustrates the desired behaviour of these columns:

2.2.6 Byte decomposition constraints

The various byte, prefix and quotient columns satisfy byte decomposition contraints. The constraints below apply for all $\mathsf{k} \in \{1, 2, \ldots, 8\}$:

1. IF $\mathsf{IS\_}\mu_i = 0$ THEN
   (a) IF $\mathsf{CT}_i = 0$ THEN $\mathsf{ACC\_k}_i = \mathsf{BYTE\_k}_i$;
   (b) IF $\mathsf{CT}_i \neq 0$ THEN $\mathsf{ACC\_k}_i = 256 \cdot \mathsf{ACC\_k}_{i-1} + \mathsf{BYTE\_k}_i$

In other words, the $\mathsf{ACC\_k}$ accumulate bytes during the preprocessing phase (which is characterized by $\mathsf{IS\_}\mu_i = 0$). What happens outside of that phase is unspecified.

2.2.7 Data organization

⟨MMU▢⟩	OFF_OOB	CT	IS_μ	TOT$^{\mu}$	μINST□
0	0	0	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	0	0	0	0	0
1	1	0	0	33	0
	$\vdots$			$\vdots$	

Figure 2.1: The above represents the first few rows of the heartbeat columns. 0 padding is on display. There is at least one macro RAM instruction: it raises the OFF_OOB flag and hence might for instance be a code copying instruction or an instruction touching call data. This single RAM macro instruction is converted into 33 (!) micro-instructions. This rules out CALLDATALOAD.

⟨MMU▢⟩	OFF_OOB	CT	IS_μ	TOT$^{\mu}$	μINST□
	$\vdots$			$\vdots$	
$\mathtt{r}-1$	off_oob	0	1	0	μ
r	1	0	0	7	μ
r	1	1	0	7	μ
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
r	1	15	0	7	μ
r	1	0	1	6	$\mu+1$
r	1	0	1	5	$\mu+2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
r	1	0	1	1	$\mu+6$
r	1	0	1	0	$\mu+7$
$\mathtt{r}+1$	off_oob'	0	0	tot'	$\mu+7$

⟨MMU▢⟩	OFF_OOB	CT	IS_μ	TOT$^{\mu}$	μINST□
	$\vdots$			$\vdots$	
$\mathtt{s}-1$	off_oob''	0	1	0	ν
s	0	0	0	83	ν
s	0	1	0	83	ν
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s	0	2	0	83	ν
s	0	0	1	82	$\nu+1$
s	0	0	1	81	$\nu+2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
s	0	0	1	1	$\nu+82$
s	0	0	1	0	$\nu+83$
$\mathtt{s}+1$	off_oob'''	0	0	tot'''	$\nu+83$

Figure 2.2: **Left hand side.** The r-th macro-instruction decomposes into 7 micro-instructions. The corresponding rows have IS_MICRO_INSTRUCTION $= 1$ (see green cells.) It also raises the OFF_OOB flag so that the precomputation phase lasts 16 rows. When entering this macro instruction the RAM offset processor had already written μ micro-instructions.

Right hand side. The s-th macro-instruction decomposes into 83 (!) micro-instructions. The corresponding rows have IS_MICRO_INSTRUCTION $= 1$ (see green cells.) It doesn't raise the OFF_OOB flag so that the precomputation phase lasts only 3 rows. When entering this macro instuction the RAM offset processor had already produced ν individual micro-instructions.

⟨INST⟩	CN_S	CN_T	⟨⟨REFO⟩⟩	⟨⟨REFS⟩⟩	⟨OFF^1⟩	⟨OFF^2⟩	⟨SIZE⟩	⟨VAL$^{\text{hi}}$⟩⟨VAL$^{\text{lo}}$⟩	INFO	⟨#⟩	$^\diamond$PRE
MLOAD		⟨CN⟩			OFF			loaded value			1
MSTORE	⟨CN⟩				OFF			value to store			1
MSTORE8	⟨CN⟩				OFF			value to store			1
REVERT	⟨CN⟩	⟨CALLER⟩	R@O	R@C	OFF		SIZE		⟨CTYPE⟩ $= 0$		2
RETURN	⟨CN⟩	⟨CALLER⟩	R@O	R@C	OFF		SIZE		⟨CTYPE⟩ $= 0$		2
RETURN	⟨CN⟩				OFF		SIZE	BC_ADDR (†)	⟨CTYPE⟩ $= 1$	DEP#	3
CREATE	⟨CN⟩				OFF		SIZE	DEP_ADDR		DEP#	3
CREATE2	⟨CN⟩				OFF		SIZE	DEP_ADDR		DEP#	3
LOGX	⟨CN⟩				OFF		SIZE			LOG#	3
SHA3	⟨CN⟩				OFF		SIZE			SHA#	3
CODECOPY		⟨CN⟩		CODESIZE	T_OFF	S_OFF	SIZE	BC_ADDR (†)	⟨CTYPE⟩	DEP#	4CC
EXTCODECOPY		⟨CN⟩		CODESIZE (‡)	T_OFF	S_OFF	SIZE	ADDR		DEP#	4CC
CALLDATACOPY	⟨CALLER⟩	⟨CN⟩	CDO	CDS	T_OFF	S_OFF	SIZE		[CSD == 1]	TX#	4CD
RETURNDATACOPY	⟨RETURNER⟩	⟨CN⟩	RDO	RDS	T_OFF	S_OFF	SIZE				4RD
CALLDATALOAD		⟨CN⟩	CDO	CDS	OFF			loaded value	[CSD == 1]	TX#	5

Figure 2.3: Some comments: the columns ⟨OFF^1⟩, ⟨OFF^2⟩, ⟨SIZE⟩, ⟨VAL$^{\text{hi}}$⟩ and ⟨VAL$^{\text{lo}}$⟩ are imported from stack, they contain respectively ${}_1$VAL$^{\text{lo}}$, ${}_2$VAL$^{\text{lo}}$, ${}_3$VAL$^{\text{lo}}$, ${}_4$VAL$^{\text{hi}}$ and ${}_4$VAL$^{\text{lo}}$. **Recall, at this point, the discussion around CODECOPY and RETURN's *mostly empty fourth stack item*. The relevant cells are signaled with a (†).** Note that the CODESIZE argument of the EXTCODECOPY (i.e. the cell with (‡)) is in reality *unknown* to the execution context. It will be verified in the data processing module where we import from the ROM module the correct code size.

2.3 Combinatorics of overlapping intervals

2.3.1 Purpose

The purpose of the present section is to introduce the sorts of checks that the zk-evm carries out during offset processing. The question is entirely about the ways in which (integer) intervals may overlap with one another.

2.3.2 Data

The arithmetization we propose accesses data in aggregate form (i.e. as 16 byte integers) rather than on a byte by byte basis. In order to perform data operations on the byte level the zk-evm procedes with all sorts of byte slicing and recomposition operations. We provide further details about these so-called **transplants** and **surgeries** in the RAM data processor chapter. The present module is not equipped to carry these out. What it does is decompose single RAM instructions (which we dub **macro-instructions**) into a series of smaller **micro-instructions** which the data processor knows how to process. This preliminary reduction of macro-instructions into sequences of micro-instructions requires dealing with questions related to limb offsets and byte offsets. The basic definition is that the **limb offset** LO and **byte offset** BO of an offset $\mathsf{OFFSET} \in \{0, 1, 2, \dots\}$ are the quotient and remainder, respectively, of the euclidean division of OFFSET by 16:

$$\begin{cases} \mathsf{OFFSET} = 16 \cdot \mathsf{LO} + \mathsf{BO}, \\ \mathsf{BO} \in \{0, 1, \dots, 15\} \end{cases}$$

While the RAM data processor can access various “data tracks”, all of them work with limbs. The RAM preprocessor thus always works with limb offsets, byte offsets and their combinatorics — regardless of the (macro-)instruction it is tasked with processing. It may at times also set exogenous data flags to indicate to the data processor from where to pull exogenous data. We will deal with this directly in the constraints.

Data transfers from a **source** data track to a **target** data track usually follow the following pattern:

1. Relevant source limbs are only read once: all required bytes are extracted in one micro-instruction at which point the zk-evm moves on to the next source limb or to the next macro-instruction;
2. Target limbs may get written to once or twice depending on several factors such as: are offsets aligned? Will this particular target limb contain both data and (zero) padding?

One of the first questions the RAM preprocessor must answer is therefore that of

Question 1. How many limbs of data will be accessed in the source?

This number obviously contributes to $\mathsf{TOTAL\_NUMBER\_OF\_MICRO\_INSTRUCTIONS}$ column which establishes the total number of micro-instructions that the pre-processor writes for the RAM data processor to perform. The largest offset touched when reading SIZE many bytes starting at offset OFFSET is $\mathsf{OFFSET} + (\mathsf{SIZE} - 1)$. Thus, setting

$$\begin{cases} 16 \cdot \mathsf{LO\_1} + \mathsf{BO\_1} & = & \mathsf{OFFSET} \\ 16 \cdot \mathsf{LO\_2} + \mathsf{BO\_2} & = & \mathsf{OFFSET} + (\mathsf{SIZE} - 1) \\ \mathsf{BO\_1}, \mathsf{BO\_2} \in \{0, 1, \dots, 15\} \end{cases}$$

The total number of source limbs to access is $\boxed{\mathsf{LO\_2} - \mathsf{LO\_1} + 1}$.

The first micro-instruction may involve extracting a suffix from the first touched limb or extracting a chunk of (consecutive) bytes from the middle of the limb if only one limb is touched. That chunk of bytes may fit into a single target limb or straddle two consecutive ones. Deciding on which is the case is required to settle the nature of the first micro-instruction and, in case $\mathsf{TOT}^{\mu} \geq 2$, the final micro-instruction as well as any transition micro-instructions. Transitions occur when the zk-evm moves from data writing to zero padding.

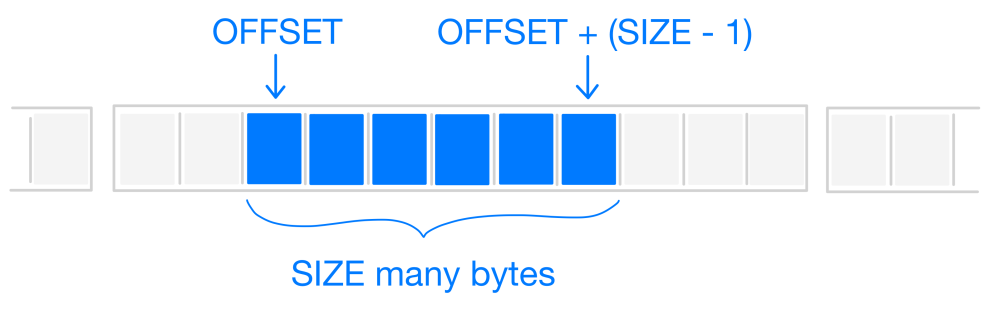

Figure 2.4: The data slice touches a single limb. The limb represented above is the LO^{th} limb and the first byte the micro-instruction will access is that at index BO *within the limb*. In the present situation $\mathsf{LO\_2} = \mathsf{LO\_1}$ and so the macro-instruction accesses a single source limb.

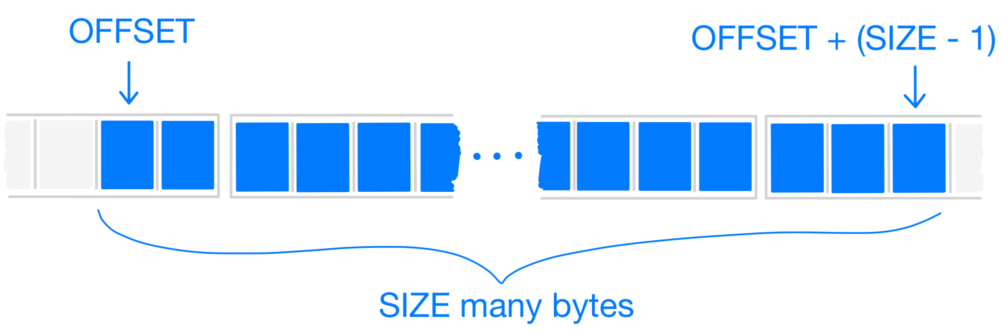

Figure 2.5: The data slice touches several limbs. In the present situation $\mathsf{LO\_2} > \mathsf{LO\_1}$ and so the macro-instruction accesses at least 2 source limbs.

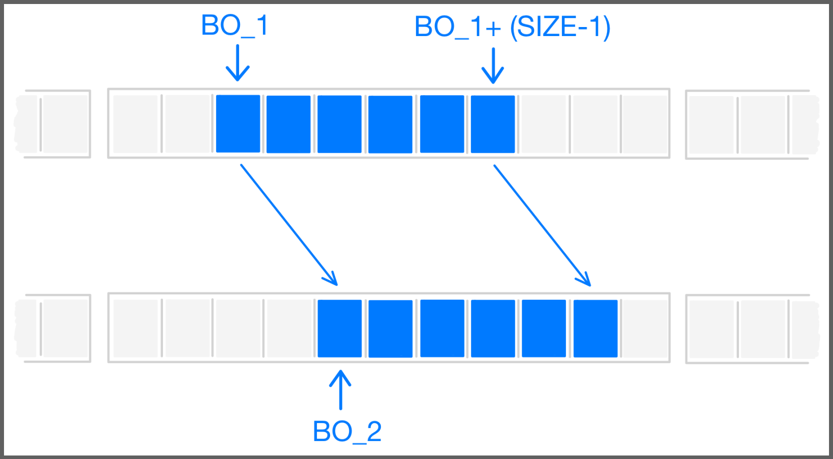

Figure 2.6: The data slice touches a single limb.

Question 2. How to determine in a constraints whether writing SIZE bytes starting at offset OFFSET requires writing one or two limbs in the target?

To answer the question we define an "answer bit" $[\![ans.]\!]$ which equals 1 if writing overlaps two

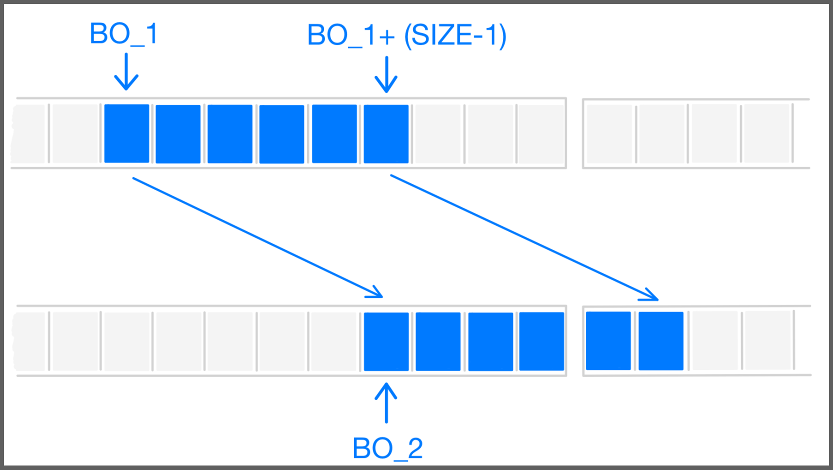

Figure 2.7: The data slice touches several limbs.

limbs and 0 if only one target limb is involved. This bit is constrained as below:

$$\begin{cases} [\![ans.]\!] = 1 & \iff \quad \mathsf{BO\_2} + (\mathsf{SIZE} - 1) > 15 \\ [\![ans.]\!] = 0 & \iff \quad \mathsf{BO\_2} + (\mathsf{SIZE} - 1) \leq 15 \end{cases}$$

which is equivalent to the constraint

$$\big(2 \cdot [\![ans.]\!] - 1\big) \cdot \big(\mathsf{BO\_2} + (\mathsf{SIZE} - 1) - 15\big) - [\![ans.]\!] = \mathsf{nibble}$$

Where nibble is a column constrained to take values in the range $\{0, 1, \ldots, 15\}$. This sort of constraint plays a prominent role in what follows. The constraint part is two fold: the equation *per se* and the range constraint on the nibble column.

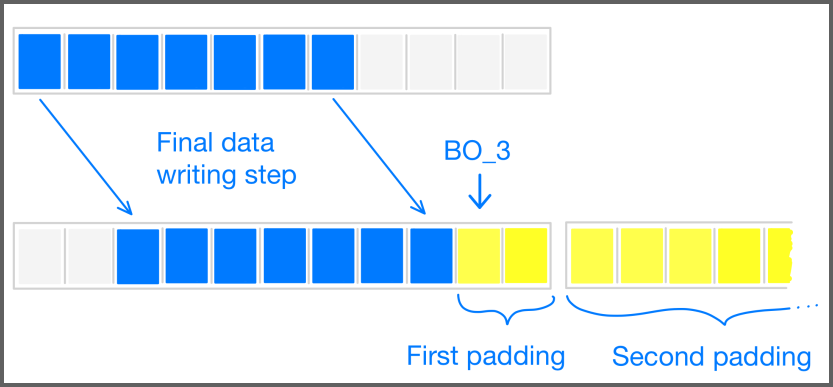

Figure 2.8: The above represents 3 consecutive micro-instructions. The first one is the final data writing step; it touches a single target limb. The second one pads *the same target limb*; thus at the step where the zk-evm transitions from data writing to zero padding the target limb offsets *isn't updated*. The nex step is just zero padding a limb.

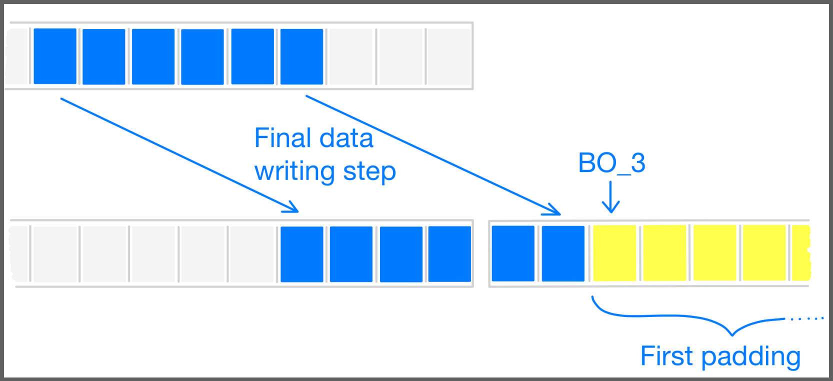

Figure 2.9: The above represents 2 consecutive micro-instructions. The first one is the final data writing step; this time it touches two target limbs. The second micro-instruction pads *the next target limb*; thus at the step where the zk-evm transitions from data writing to zero padding the target limb offsets *is updated*.

2.4 Constraints

2.4.1 Parametrized instruction decoding, preprocessing and constraints

Instruction decoding in the RAM offset processor is more involved than elsewhere. This is because:

1. it fufills different purposes in the preprocessing phase and in the micro-instruction writing phase;
2. in either phase the zk-evm does **parametrized instruction decoding**.

One of the purposes of the precomputation phase (characterized by $\mathsf{IS\_}\mu = 0$) is to produce a series of binary flags. How many of these binary flags are to be computed as well as the procdedure by which they are to be computed (among other things) can be read off the instruction decoded $^{\diamond}\mathsf{PRECOMPUTATION}$ parameter. Examples of such flags include the ALIGNED flag which tells the RAM data processor whether certain memory operations can be done without any byte decompositions ($\mathsf{ALIGNED} = 0$ means the micro-instruction will be a type of limb surgery, $\mathsf{ALIGNED} = 1$ means the micro-instruction will be a type of transplant.)

Once these binary flags are set and justified at the end of the preprocessing phase, the current opcode and these flags considered as a whole are understood as a **parametrized instruction**. In the micro-instruction writing phase (characterized by $\mathsf{IS\_}\mu = 1$) it is *parametrized instructions* that are instruction decoded in a process we dub **parametrized instruction decoding**[7] . This allows for the second phase of micro-instruction writing to produce the adequate sequence of micro-instructions.

Thus the workflow is as follows:

1. upon entering a new macro instruction the $\mathsf{IS\_}\mu$ flag is set to 0 and stays $= 0$ for either 3 or 16 rows;
2. the instruction and $\mathsf{IS\_}\mu = 0$ are instruction decoded[8]; this is mostly about retrieving the precomputation type $^{\diamond}\mathsf{PRECOMPUTATION}$;
3. the offset preprocessor executes the preprocessing associated with the precomputation type;

[7] This is technically also true of the preprocessing phase, though simpler: code copying instructions and call data instructions may have offsets that drastically go out of bounds which will alter the precomputation and the resulting sequence of micro instructions; the behaviour of `RETURN` depends on whether the current execution context is a deployment context or not. Thus instructions are decorated by two binary flags OFF_OOB and CTYPE that affect their instruction decoding in the preprocessing phase.

[8] An info bit can be part of the picture too; either $\langle\mathsf{CTYPE}\rangle$ or $[\mathsf{CSD} = 0]$

4. this produces a number of parameters and binary flags;
5. when the precomputation phase comes to an end IS_μ switches to 1;
6. the instruction and IS_$\mu = 1$ and the flags that were just produced now form a parametrized instruction;
7. this parametrized instruction is instruction decoded until TOT$^\mu$ hits zero;
8. in that time the parameters may change and lead to changes the decoded μINST;

This produces a sequence of micro-instructions. These micro-instructions are imported by the RAM data processor where each of these requests is honored in order of production.

2.4.2 Setting the FAST flag

In the following sections we detail how the offset preprocessor breaks RAM maxro-instructions down into a sequence of RAM micro-instructions. Micro-instructions are either **transplants** (i.e. fast operations i.e. operations requiring not byte decomposition to perform) or **surgeries** (i.e. slow operations i.e. operations that require the RAM data processor to carry out one or more byte decompositions.) Thus the FAST flag depends purely on the micro-instruction. It will be set without further comment on every row where IS_$\mu = 1$ according to the following:

Micro-instructions with FAST $= 1$: —

1. `RamToRam`
2. `ExoToRam`
3. `RamIsExo`
4. `KillingOne`
5. `KillingTwo`
6. `KillingThree`
7. `PushTwoRamToStack`
8. `PushOneRamToStack`
9. `PushTwoStackToRam`
10. `StoreXinAoneRequired`
11. `StoreXinAtwoRequired`
12. `StoreXinAthreeRequired`
13. `StoreXinB`
14. `StoreXinC`

Micro-instructions with FAST $= 0$: —

1. `RamLimbExcision`,
2. `RamToRamSlideChunk`,
3. `RamToRamSlideOverlappingChunk`,
4. `ExoToRamSlideChunk`,
5. `ExoToRamSlideOverlappingChunk`,
6. `PaddedExoFromOne`,
7. `PaddedExoFromTwo`,
8. `FullExoFromTwo`,
9. `FullStackToRAM`,
10. `ByteSwap`,
11. `LsbFromStackToRAM`,
12. `FirstFastSecondPadded`,
13. `FirstPaddedSecondZero`,
14. `Exceptional_RamToStack_3To2Full`,
15. `NA_RamToStack_3To2Full`,
16. `NA_RamToStack_3To2Padded`,
17. `NA_RamToStack_2To2Padded`,
18. `NA_RamToStack_2To1FullAndZero`,
19. `NA_RamToStack_2To1PaddedAndZero`,
20. `NA_RamToStack_1To1PaddedAndZero`,

(on rows where IS_$\mu = 0$ one may set FAST to 0)

2.4.3 Type 1

Instructions

The following instructions follow type 1 precomputation:

INST	IS_μ	ALIGNED	$^{\lozenge}$TO_RAM	$^{\lozenge}$PRE	μINST
MLOAD	0		0	1	
MSTORE	0		1	1	
MSTORE8	0		1	1	
MLOAD	1	0	0	1	NA_RamToStack_3To2Full
MLOAD	1	1	0	1	PushTwoRamToStack
MSTORE	1	0	1	1	FullStackToRAM
MSTORE	1	1	1	1	PushTwoStackToRam
MSTORE8	1		1	1	LsbFromStackToRAM

1. MLOAD
2. MSTORE
3. MSTORE8

Note that CALLDATALOAD, while similar (at a first glance) to MLOAD, follows a different, more complex, precomputation type. We will expand as to why in due time.

Workflow

For instructions with $^{\lozenge}\mathsf{PRE} = 1$ the precomputation consists in

1. setting and verifying the quotient and remainder of the euclidean division of $\langle\mathsf{OFF}^1\rangle$ by 16,
2. setting the ALIGNED flag to 1 if the remainder of said euclidean division is 0.

Note that the ALIGNED flag will be ignored by the MSTORE8 instruction.

The RAM data processor deals with MSTORE8 instructions in a uniform way: there are no fast MSTORE8 instructions, i.e. every MSTORE8 translates to a surgery micro instruction in the RAM data processor. Parametrized instruction decoding for MSTORE8 thus coincides with standard instruction decoding: every MSTORE8 instruction gives rise to a LsbFromStackToRAM micro instruction in the RAM data processor.

MLOAD and MSTORE instructions, on the other hand, can give rise to either fast micro instructions or slow micro instructions. The parametrized instruction decoding of MLOAD and MSTORE thus depends on a single binary flag, ALIGNED, that lights up precisely when $\langle\mathsf{OFF}^1\rangle$ is a clean multiple of 16. Thus MLOAD translates to the transplant PushTwoRamToStack when $\mathsf{ALIGNED} = 1$ and to the surgery $[3 \Rightarrow 2\,\texttt{Full}]$ when $\mathsf{ALIGNED} = 0$. Similarly MSTORE translates to the transplant PushTwoStackToRam when $\mathsf{ALIGNED} = 1$ and to the surgery $[2\,\texttt{Full} \Rightarrow 3]$ when $\mathsf{ALIGNED} = 0$.

Parametrized instruction decoder

The relevant portion of the parametrized instruction decoder looks like so:

Preprocessing

We collect under the moniker Type_1 the following collection of constraints. We jump straight to the last preprocessing step:

All constraints in this subsection assume $\mathsf{IS\_}\mu_i = 0$ AND $\mathsf{IS\_}\mu_{i+1} = 1$

1. $\mathsf{OFF\_OOB}_i = 0$;

Indeed, $\langle \mathsf{OFF}^1 \rangle_i$ already went through the Memory Expansion Module where it was tested for smallness.

2. We fix the source and target context according to the ${}^{\Diamond}\mathsf{TO\_RAM}_i$ flag:

   (a) IF ${}^{\Diamond}\mathsf{TO\_RAM}_i = 0$ THEN
$$\begin{cases} \mathsf{CN\_S}_i = \langle \mathsf{CN} \rangle_i \\ \mathsf{CN\_T}_i = 0 \end{cases}$$

   (b) IF ${}^{\Diamond}\mathsf{TO\_RAM}_i = 1$ THEN
$$\begin{cases} \mathsf{CN\_S}_i = 0 \\ \mathsf{CN\_T}_i = \langle \mathsf{CN} \rangle_i \end{cases}$$

   In other words, `MSTORE` and `MSTORE8` have target context equal to the current context ($\mathsf{CN\_T}_i = \langle \mathsf{CN} \rangle_i$) and `MLOAD` has source context equal to the current context ($\mathsf{CN\_S}_i = \langle \mathsf{CN} \rangle_i$). The other context is zero in both cases.

   We can of course subsume the above in the constraints $\mathsf{CN\_S}_i = (1 - {}^{\Diamond}\mathsf{TO\_RAM}_i) \cdot \langle \mathsf{CN} \rangle_i$ and $\mathsf{CN\_T}_i = {}^{\Diamond}\mathsf{TO\_RAM}_i \cdot \langle \mathsf{CN} \rangle_i$.

3. $\langle \mathsf{OFF}^1 \rangle_i = 16 \cdot \mathsf{ACC\_1}_i + \mathsf{NIB\_1}_i$;

4. we set the source and target limb and byte offsets:

   (a) IF ${}^{\Diamond}\mathsf{TO\_RAM}_i = 0$ THEN
$$\begin{cases} \mathsf{SLO}_{i+1} = \mathsf{SLO}_i = \mathsf{ACC\_1}_i \\ \mathsf{SBO}_{i+1} = \mathsf{SBO}_i = \mathsf{NIB\_1}_i \\ \mathsf{TLO}_{i+1} = \mathsf{TLO}_i = 0 \\ \mathsf{TBO}_{i+1} = \mathsf{TBO}_i = 0 \end{cases}$$

   (b) IF ${}^{\Diamond}\mathsf{TO\_RAM}_i = 1$ THEN
$$\begin{cases} \mathsf{SLO}_{i+1} = \mathsf{SLO}_i = 0 \\ \mathsf{SBO}_{i+1} = \mathsf{SBO}_i = 0 \\ \mathsf{TLO}_{i+1} = \mathsf{TLO}_i = \mathsf{ACC\_1}_i \\ \mathsf{TBO}_{i+1} = \mathsf{TBO}_i = \mathsf{NIB\_1}_i \end{cases}$$

   Again we can subsume the previous constraints in a linear combination as before.

5. Set the fast operation flag:
$$\begin{cases} \text{IF } \mathsf{NIB\_1}_i = 0 \text{ THEN } \mathsf{ALIGNED}_i = 1, \\ \text{IF } \mathsf{NIB\_1}_i \neq 0 \text{ THEN } \mathsf{ALIGNED}_i = 0; \end{cases}$$

6. $\mathsf{TOT}^{\mu}_i = 1$: an `MLOAD`, `MSTORE` or `MSTORE8` is dealt with by the RAM data processor in one micro instruction;

Micro-instruction writing

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1}$$

1. The source and target limb were already set.

2. IF $\mathsf{ALIGNED}_i = 1$ THEN

$$\begin{cases} \textsc{if } {}^{\diamond}\mathsf{TO\_RAM}_i = 0 \textsc{ then } \mu\mathsf{INST}_i = \texttt{PushTwoRamToStack} \\ \textsc{if } \left({}^{\diamond}\mathsf{TO\_RAM}_i = 1 \textsc{ and } \langle\mathsf{INST}\rangle_i = \texttt{MSTORE}\right) \textsc{ then } \mu\mathsf{INST}_i = \texttt{PushTwoStackToRam} \end{cases}$$

3. IF $\mathsf{ALIGNED}_i = 0$ THEN

$$\begin{cases} \textsc{if } {}^{\diamond}\mathsf{TO\_RAM}_i = 0 \textsc{ then } \mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_3To2Full} \\ \textsc{if } \left({}^{\diamond}\mathsf{TO\_RAM}_i = 1 \textsc{ and } \langle\mathsf{INST}\rangle_i = \texttt{MSTORE}\right) \textsc{ then } \mu\mathsf{INST}_i = \texttt{FullStackToRAM} \end{cases}$$

4. IF $\langle\mathsf{INST}\rangle_i = \texttt{MSTORE8}$ THEN $\mu\mathsf{INST}_i = \texttt{LsbFromStackToRAM}$

2.4.4 Type 2

Instructions

The following instructions follow type 3 precomputation:

1. `RETURN` in a non deployment context
2. `REVERT`

Workflow

The precomputation phase of type 2 is involved. It requires computing a number of euclidean divisions and doing a few comparisons. Here is the general overview of the computation:

1. The preprocessor first determines the "real size" of data to be moved, i.e. the minimum

   $$\mathsf{MIN} := \min\{\langle\mathsf{SIZE}\rangle, \langle\mathsf{R@C}\rangle\}$$

   Indeed, when returning or reverting successfully, the current execution context writes as much of its return data to its parent context as the parent context permits; the "as much as possible" part of that statement is captured by the minimun.

2. It then determines the euclidean divisions

   $$\begin{cases} \langle\mathsf{OFF}^1\rangle & = & 16\cdot\mathsf{ACC\_1} + \mathsf{NIB\_1}, \\ \langle\mathsf{OFF}^1\rangle + (\mathsf{MIN}-1) & = & 16\cdot\mathsf{ACC\_2} + \mathsf{NIB\_2}, \\ \langle\mathsf{R@O}\rangle & = & 16\cdot\mathsf{ACC\_3} + \mathsf{NIB\_3}, \\ \langle\mathsf{R@O}\rangle + (\mathsf{MIN}-1) & = & 16\cdot\mathsf{ACC\_4} + \mathsf{NIB\_4}. \end{cases}$$

   Note that all these integers have previously been checked for smallness (i.e. they fit into 3 bytes) by the Memory Expansion Module; we know that proving these euclidean divisions will require only byte decompositions of (what are *a priori* known to be) three byte integers $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$, $\mathsf{ACC\_3}$ and $\mathsf{ACC\_4}$. Note, too, that instructions with zero size will be filtered out before reaching the preprocessor.

3. The current macro-instruction is broken down into $\mathsf{TOT}^\mu = \mathsf{ACC\_2} - \mathsf{ACC\_1} + 1$ micro-instructions; there are several execution paths ahead:

   (a) $\mathsf{ACC\_2} = \mathsf{ACC\_1}$ i.e. $\mathsf{TOT}^\mu = 1$ means that the bytes to write to the caller RAM live in a single limb of the current execution context; a single surgery will suffice;

   (b) $\mathsf{ACC\_2} = \mathsf{ACC\_1} + 1$ i.e. $\mathsf{TOT}^\mu = 2$ means that the bytes to write to the caller RAM live in two contiguous RAM limbs of the current execution context;

(c) $\mathsf{ACC\_2} \geq \mathsf{ACC\_1} + 2$ i.e. i.e. $\mathsf{TOT}^{\mu} \geq 3$ means that the bytes to write to the caller RAM live in at least 3 contiguous RAM limbs; the first and last of these may only be partially copied to their destination, but $\mathsf{ACC\_2} - (\mathsf{ACC\_1} + 1) = \mathsf{TOT}^{\mu} - 2 \geq 1$ will fully carry over to the caller RAM;

The sequence of micro-instructions into which the macro-instruction decomposes reflects this structure:

(a) In the first case a single surgery will suffice; this surgery may span one or two (neighboring) limbs in the target context (i.e. the caller context); determining which surgery applies requires us to figure out which of the following holds:

$$\mathsf{NIB\_3} > \mathsf{NIB\_1} \text{ ? or } \mathsf{NIB\_3} \leq \mathsf{NIB\_1} \text{ ?}$$

In the first case a chunk of consecutive bytes from the source limb will be split and made to replace a suffix and a prefix of two neighboring limbs in the caller RAM. In the second case a chunk of consecutive bytes in the source limb will replace a chunk of bytes in a limb of the caller RAM.

(b) In the second case two surgeries are enough; again there are various possibilities for these surgeries; the previous discussion applies, but we now also have to consider the second limb, a prefix of which will replace either (a chunk of consecutive bytes of a single limb in the caller RAM) or a suffix and a prefix of two consecutive limbs in the caller RAM; determining which surgery applies requires to answer dual question:

$$\mathsf{NIB\_4} < \mathsf{NIB\_2} \text{ ? or } \mathsf{NIB\_4} \geq \mathsf{NIB\_2} \text{ ?}$$

(c) In the third case the initial surgery (which follows the logic laid out in part earlier) is followed by $\mathsf{TOT}^{\mu} - 2 \geq 1$ full writes which in turn is followed by a final surgery (which follows the logic laid out in part earlier).

4. Note that in the third case we can further distinguish between *fast* operations and *slow* ones. The $\mathsf{ACC\_2} - (\mathsf{ACC\_1} + 1)$ full writes will be fast if $\mathsf{NIB\_1} = \mathsf{NIB\_3}$, otherwise they will be slow.

Note that the arithmetization treats the second and third case on equal footing.

Parametrized instruction decoder

The relevant portion of the parametrized instruction decoder looks like so:

Figure 2.10

Context constraints

We collect under the moniker `Type_2` the following collection of constraints:

1. We fix the source and target context:

$$\begin{cases} \mathsf{CN\_S}_i = \langle \mathsf{CN} \rangle_i \\ \mathsf{CN\_T}_i = \langle \mathsf{CALLER} \rangle_i \end{cases}$$

2. $\mathsf{OFF\_OOB}_i = 0$;

Let us expand on this constraint. Before entering the RAM preprocessor the offset and size parameters of the `RETURN`/`REVERT` instruction underwent analysis in the Memory Expansion Module where they were tested for smallness. We therefore know that both of them are small (i.e. fit into 3 bytes). Their sum fits into $3 * 8 + 1$ bits and the quotient of the euclidean division by 16 of these integers fit into 3 bytes (and the remainders are nibbles.) This allows us to set, *a priori*, $\mathsf{OFF\_OOB}_i = 0$.

Preprocessing

We jump straight to the last preprocessing step:

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 0 \ \ \text{AND} \ \mathsf{IS\_}\mu_{i+1} = 1}$$

Euclidean divisions. $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$, $\mathsf{ACC\_3}$ and $\mathsf{ACC\_4}$ target quotients of certain euclidean divisions and $\mathsf{NIB\_1}$, $\mathsf{NIB\_2}$, $\mathsf{NIB\_3}$ and $\mathsf{NIB\_4}$ target the associated remainders:

$$\begin{cases} \langle \mathsf{OFF}^1 \rangle_i & = & 16 \cdot \mathsf{ACC\_1}_i + \mathsf{NIB\_1}_i \\ \langle \mathsf{OFF}^1 \rangle_i + (\mathsf{MIN}_i - 1) & = & 16 \cdot \mathsf{ACC\_2}_i + \mathsf{NIB\_2}_i \\ \langle \mathsf{R@O} \rangle_i & = & 16 \cdot \mathsf{ACC\_3}_i + \mathsf{NIB\_3}_i \\ \langle \mathsf{R@O} \rangle_i + (\mathsf{MIN}_i - 1) & = & 16 \cdot \mathsf{ACC\_4}_i + \mathsf{NIB\_4}_i \end{cases}$$

(The value of MIN_i is set below.) Note that we don't "use" $\mathsf{ACC\_2}_i$ or $\mathsf{ACC\_4}_i$ *per se*; they exist purely to justify the associated nibbles $\mathsf{NIB\_2}_i$ and $\mathsf{NIB\_4}_i$.

Comparisons. We justify the three bit columns $[\![1]\!]$, $[\![2]\!]$ and $[\![3]\!]$ and the fifth accumulator column $\mathsf{ACC\_5}$:

$$\begin{cases} \big(\langle \mathsf{R@C} \rangle_i - \langle \mathsf{SIZE} \rangle_i\big) & \cdot & \big(2 \cdot [\![1]\!]_i - 1\big) - [\![1]\!]_i & = & \mathsf{ACC\_5}_i \\ \big(\mathsf{NIB\_3}_i - \mathsf{NIB\_1}_i\big) & \cdot & \big(2 \cdot [\![2]\!]_i - 1\big) - [\![2]\!]_i & = & \mathsf{NIB\_5}_i \\ \big(\mathsf{NIB\_2}_i - \mathsf{NIB\_4}_i\big) & \cdot & \big(2 \cdot [\![3]\!]_i - 1\big) - [\![3]\!]_i & = & \mathsf{NIB\_6}_i \end{cases}$$

Thus

$$\begin{cases} [\![1]\!] = 1 & \iff & \langle \mathsf{R@C} \rangle > \langle \mathsf{SIZE} \rangle \\ [\![2]\!] = 1 & \iff & \mathsf{NIB\_3} > \mathsf{NIB\_1} \\ [\![3]\!] = 1 & \iff & \mathsf{NIB\_2} > \mathsf{NIB\_4} \end{cases}$$

Note that $\mathsf{NIB\_5}$ and $\mathsf{NIB\_6}$ don't play a functional role in type 2 instructions. Their sole purpose is in establishing $[\![2]\!]$ and $[\![3]\!]$.

Establishing minimum. We set the minimum $\mathsf{MIN} := \min\{\langle \mathsf{SIZE} \rangle, \langle \mathsf{R@C} \rangle\}$:

$$\mathsf{MIN}_i = [\![1]\!]_i \cdot \langle \mathsf{SIZE} \rangle_i + [\![1]\!]_i^\vee \cdot \langle \mathsf{R@C} \rangle_i.$$

(Recall our standing convention of writing $[\![k]\!]^\vee := (1 - [\![k]\!])$.)

Workflow parameters. We establish the $\mathsf{TOTAL\_NUMBER\_OF\_MICRO\_INSTRUCTIONS}$:

$$\mathsf{TOT}_i^\mu = \mathsf{ACC\_2}_i - \mathsf{ACC\_1}_i + 1$$

and $[\![4]\!]$ which purely measures whether $\mathsf{TOT}_i^\mu = 1$ or $\mathsf{TOT}_i^\mu > 1$:

$$\begin{cases} \text{IF } \mathsf{TOT}_i^\mu = 1 \text{ THEN } [\![4]\!]_i = 1 \\ \text{IF } \mathsf{TOT}_i^\mu \neq 1 \text{ THEN } [\![4]\!]_i = 0 \end{cases}$$

We now establish $[\![5]\!]_i$. This bit only matters when $\mathsf{TOT}_i^\mu = 1$ i.e. $[\![4]\!] = 1$. $[\![5]\!]$ decides which operation to perform when $\mathsf{NIB\_3} > \mathsf{NIB\_1}$ (i.e. $[\![2]\!]_i = 1$)

1. IF $[\![4]\!]_i = 0$ THEN $[\![5]\!]_i = 0$

2. IF $\llbracket 4 \rrbracket_i = 1$ THEN
$$\mathsf{NIB\_3} + (\mathsf{MIN}_i - 1) - 16 \cdot \llbracket 5 \rrbracket_i = \mathsf{NIB\_7}_i$$

Note that $\mathsf{NIB\_7}$ doesn't play a functional role in type 2 instructions. Its sole purpose is in establishing $\llbracket 5 \rrbracket$.

We give more details. Assume $\llbracket 4 \rrbracket_i = 1$ i.e. one micro-instruction is enough. If $\mathsf{NIB\_3} \leq \mathsf{NIB\_1}$ the single operation is necessarily a `RamToRamSlideChunk`. But if $\mathsf{NIB\_3} > \mathsf{NIB\_1}$ it could either be a `RamToRamSlideChunk` or a `RamToRamSlideOverlappingChunk`. The second case happens *iff* $\mathsf{NIB\_3} + (\mathsf{MIN}_i - 1) \geq 16$ i.e. $\llbracket 5 \rrbracket_i = 1$ We set the fast operation flag:

$$\begin{cases} \text{IF } \mathsf{NIB\_1}_i = \mathsf{NIB\_3}_i \text{ THEN } \mathsf{ALIGNED}_i = 1 \\ \text{IF } \mathsf{NIB\_1}_i \neq \mathsf{NIB\_3}_i \text{ THEN } \mathsf{ALIGNED}_i = 0 \end{cases}$$

We establish the source and target limb and byte offsets:

$$\begin{cases} \mathsf{SLO}_{i+1} & = & \mathsf{SLO}_i & = & \mathsf{ACC\_1}_i \\ \mathsf{SBO}_{i+1} & = & \mathsf{SBO}_i & = & \mathsf{NIB\_1}_i \\ \mathsf{TLO}_{i+1} & = & \mathsf{TLO}_i & = & \mathsf{ACC\_3}_i \\ \mathsf{TBO}_{i+1} & = & \mathsf{TBO}_i & = & \mathsf{NIB\_3}_i \end{cases}$$

Micro-instruction writing

We distinguish several cases. Note that

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1}$$

1. $\mathsf{SLO}_i = \mathsf{SLO}_{i-1} + \mathsf{IS\_}\mu_{i-1}$: the source limb offset grows by 1 with every instruction, regardless of anything else;

2. IF $\llbracket 4 \rrbracket_i = 1$ THEN

   (a) SLO_i and SBO_i are already set

   (b) TLO_i and TBO_i are already set

   (c) $\mathsf{SIZE}_i = \mathsf{MIN}_i$

   (d) IF $\llbracket 2 \rrbracket_i = 0$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideChunk`

   (e) IF $\llbracket 2 \rrbracket_i = 1$ THEN

   $$\begin{cases} \text{IF } \llbracket 5 \rrbracket_i = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } \llbracket 5 \rrbracket_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \end{cases}$$

   Recall that the case $\llbracket 4 \rrbracket_i = 1$ corresponds to a the single surgery, so this single constraint is sufficient.

3. IF $\llbracket 4 \rrbracket_i = 0$ there are most steps but they are less cramped. We start with TLO:

   (a) IF $\mathsf{IS\_}\mu_{i-1} = 0$ THEN
   $$\mathsf{TLO}_{i+1} = \mathsf{TLO}_i + (\mathsf{ALIGNED}_i + \llbracket 2 \rrbracket_i)$$
   Note that $\mathsf{ALIGNED}_i + \llbracket 2 \rrbracket_i = 1 \iff \mathsf{NIB\_3} \geq \mathsf{NIB\_1}$

   (b) IF $\mathsf{IS\_}\mu_{i-1} = \mathsf{IS\_}\mu_i = \mathsf{IS\_}\mu_{i+1} = 1$ THEN
   $$\mathsf{TLO}_{i+1} = \mathsf{TLO}_i + 1$$
   Note that the middle condition $\mathsf{IS\_}\mu_i = 1$ is redundant;

The previous two columns signify that if $\mathsf{NIB\_3} \geq \mathsf{NIB\_2}$ then TLO_i grows by one with every micro instruction. However when $\mathsf{NIB\_3} < \mathsf{NIB\_2}$ the first limb in the target is modified by two successive micro-instructions. This is captured by the above constraints.

- (c) IF $\mathsf{IS\_}\mu_{i-1} = 0$
  - i. $\mathsf{SIZE}_i = (15 - \mathsf{NIB\_1}_i) + 1$
  - ii. IF $[\![2]\!]_i = 0$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideChunk`
  - iii. IF $[\![2]\!]_i = 1$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideOverlappingChunk`
- (d) IF $\mathsf{IS\_}\mu_{i-1} = 1$ THEN
  - i. $\mathsf{SBO}_i = 0$
  - ii. $\mathsf{TBO}_i = \mathsf{NIB\_3}_i + 16 - \mathsf{NIB\_1}_i - 16 \cdot (\mathsf{ALIGNED}_i + [\![2]\!]_i)$

    Note that by construction, *for type 2 instructions*, $\mathsf{ALIGNED}_i$ and $[\![2]\!]_i$ measure disjiont events, so that $\mathsf{ALIGNED}_i + [\![2]\!] = \mathsf{ALIGNED}_i + [\![2]\!] - \mathsf{ALIGNED}_i \cdot [\![2]\!] = \mathsf{ALIGNED}_i \vee [\![2]\!]$ is a binary column and its interpretation is $\mathsf{ALIGNED}_i + [\![2]\!] = 1 \iff \mathsf{NIB\_1} \leq \mathsf{NIB\_3}$.
  - iii. IF $\mathsf{TOT}_i^{\mu} \neq 0$ THEN
    - A. $\mathsf{SIZE}_i = 16$
    - B. IF $\mathsf{ALIGNED}_i = 1$ THEN $\mu\mathsf{INST}_i =$ `RamToRam`
    - C. IF $\mathsf{ALIGNED}_i = 0$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideOverlappingChunk`
  - iv. IF $\mathsf{TOT}_i^{\mu} = 0$ THEN
    - A. $\mathsf{SIZE}_i = \mathsf{NIB\_2}_i + 1$
    - B. IF $[\![3]\!]_i = 0$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideChunk`
    - C. IF $[\![3]\!]_i = 1$ THEN $\mu\mathsf{INST}_i =$ `RamToRamSlideOverlappingChunk`

2.4.5 Type 3

Instructions

The following instructions follow type 3 precomputation:

1. `SHA3`
2. `LOG0`-`LOG4`
3. `CREATE` and `CREATE2`
4. `RETURN` in a deployment context

Workflow

Parametrized instruction decoder

The relevant portion of the parametrized instruction decoder looks like so:

Context constraints

We fix some context information: We collect under the moniker `Type_3` the following collection of constraints:

1. source and target contexts:
$$\begin{cases} \mathsf{CN\_S}_i = \langle \mathsf{CN} \rangle_i \\ \mathsf{CN\_T}_i = 0 \end{cases}$$
2. $\mathsf{OFF\_OOB}_i = 0$;

Let us expand on this constraint. Just as with `Type_1`, $\langle \mathsf{OFF}^1 \rangle_i$ already went through the Memory Expansion Module where it was tested for smallness. The computation also tested $\langle \mathsf{SIZE} \rangle_i$ for smallness. Thus arriving into the present module we know that both of them are small (i.e. fit into 3 bytes) and so their sum fits into $3 * 8 + 1$ bits and the quotient of the euclidean division by 16 of these integers fit into 3 bytes (and the remainders are nibbles.)

INST	IS_μ	$[\![1]\!]$	$[\![2]\!]$	ALIGNED	INFO	$^{\lozenge}$X_SHA3	$^{\lozenge}$X_LOG	$^{\lozenge}$X_ROM	$^{\lozenge}$PRE	μINST
SHA3	0					1	0	0	3	
LOGX	0					0	1	0	3	
CREATE	0					0	0	1	3	
CREATE2	0					1	0	1	3	
RETURN	0				1	0	0	1	3	
same as above	1	0	0	0	same	same	same	same	3	FullExoFromTwo
	1	0	0	1						RamIsExo
	1	1	0							PaddedExoFromOne
	1	1	1							PaddedExoFromTwo

Figure 2.11: The $^{\lozenge}$X_ROM, $^{\lozenge}$X_LOG and $^{\lozenge}$X_SHA3 columns take the same value in the IS_$\mu = 1$ case as in the IS_$\mu = 0$ case. The $[\![1]\!]$ column records whether the current micro instruction is forms a full word of exogenous data or a padded one. The $[\![2]\!]$ column records, in case where the $\langle\mathsf{SIZE}\rangle$ isn't a clean multiple of 16, the kind of final micro-instruction that will take place: either we form a padded limb of exogenous data using one limb from RAM or we form one padded limb of exogenous data using two limbs from RAM.

Preprocessing

We jump straight to the last preprocessing step, i.e. constraints below are under the assumption

$$\boxed{\mathsf{IS\_}\mu_i = 0 \quad \text{AND} \quad \mathsf{IS\_}\mu_{i+1} = 1}$$

Euclidean divisions. ACC_1, ACC_2 target the quotients of certain euclidean divisions and NIB_1, NIB_2 target the associated remainders:

$$\begin{cases} \langle\mathsf{OFF}^1\rangle_i & = & 16 \cdot \mathsf{ACC\_1}_i + \mathsf{NIB\_1}_i \\ \langle\mathsf{SIZE}\rangle_i & = & 16 \cdot \mathsf{ACC\_2}_i + \mathsf{NIB\_2}_i \end{cases}$$

Fast operation. We set

$$\begin{cases} \text{IF } \mathsf{NIB\_1}_i = 0 \text{ THEN } \mathsf{ALIGNED}_i = 1 \\ \text{IF } \mathsf{NIB\_1}_i \neq 0 \text{ THEN } \mathsf{ALIGNED}_i = 0 \end{cases}$$

Special final micro-instruction. We set

$$\begin{cases} \text{IF } \mathsf{NIB\_2}_i = 0 \text{ THEN } [\![1]\!]_i = 0 \\ \text{IF } \mathsf{NIB\_2}_i \neq 0 \text{ THEN } [\![1]\!]_i = 1 \end{cases}$$

There is a final operation with padding if the SIZE isn't a clean multiple of 16. The $[\![1]\!]$ flag detects it.

Nature of final micro-instruction. In case there is a special final instruction $[\![2]\!]$ will distinguish between the two possibilities:

1. IF $\mathsf{ALIGNED}_i = 1$ THEN $[\![2]\!]_i = 0$;

2. IF $\mathsf{ALIGNED}_i = 0$ THEN

$$\mathsf{NIB\_1}_i + (\mathsf{NIB\_2}_i - 1) - 16 \cdot [\![2]\!]_i = \mathsf{NIB\_3}_i$$

Subsuming the previous discussion:

$$\left\{\begin{array}{rcl} \mathsf{ALIGNED} = 1 & \iff & \langle \mathsf{OFF}^1 \rangle \text{ is a clean multiple of 16} \\[1ex] [\![1]\!] = 1 & \iff & \text{there's a special final operation} \\ & \iff & \langle \mathsf{SIZE} \rangle \text{ isn't a clean multiple of 16} \\[1ex] [\![2]\!] = 1 & \iff & \mathsf{NIB\_1}_i + (\mathsf{NIB\_2}_i - 1) \geq 16 \end{array}\right.$$

Workflow parameters. We establish the $\mathsf{TOTAL\_NUMBER\_OF\_MICRO\_INSTRUCTIONS}$:

$$\mathsf{TOT}_i^{\mu} = \mathsf{ACC\_2}_i + [\![1]\!]_i$$

We establish the initial source and target limb and byte offsets:

$$\left\{\begin{array}{rcl} \mathsf{SLO}_i & = & \mathsf{ACC\_1}_i \\ \mathsf{SBO}_i & = & \mathsf{NIB\_1}_i \\[1ex] \mathsf{TLO}_i & = & 0 \\ \mathsf{TBO}_i & = & 0 \end{array}\right.$$

Constraints

All constraints in this subsection assume

$$\boxed{\mathsf{IS\_}\mu_i = 1}$$

1. the source and target limb and byte offsets change very predictably:

$$\left\{\begin{array}{rcl} \mathsf{SLO}_i & = & \mathsf{SLO}_{i-1} + \mathsf{IS\_}\mu_{i-1} \\ \mathsf{SBO}_i & = & \mathsf{NIB\_1}_i \\[1ex] \mathsf{TLO}_i & = & \mathsf{TLO}_{i-1} + \mathsf{IS\_}\mu_{i-1} \\ \mathsf{TBO}_i & = & 0 \end{array}\right.$$

2. IF $\mathsf{TOT}_i^{\mu} \neq 0$ THEN
   (a) IF $\mathsf{ALIGNED}_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{RamIsExo}$
   (b) IF $\mathsf{ALIGNED}_i = 0$ THEN $\mu\mathsf{INST}_i = \mathtt{FullExoFromTwo}$
3. IF $\mathsf{TOT}_i^{\mu} = 0$ THEN
   (a) IF $\Big(\mathsf{ALIGNED}_i = 1$ AND $[\![1]\!]_i = 0\Big)$ THEN $\mu\mathsf{INST}_i = \mathtt{RamIsExo}$
   (b) IF $\Big(\mathsf{ALIGNED}_i \neq 1$ OR $[\![1]\!]_i \neq 0\Big)$
      i. $\mathsf{SIZE}_i = \mathsf{NIB\_2}_i$ THEN
      ii. IF $[\![2]\!]_i = 0$ THEN $\mu\mathsf{INST}_i = \mathtt{PaddedExoFromOne}$
      iii. IF $[\![2]\!]_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{PaddedExoFromTwo}$

2.4.6 Type 4

Instructions

The following instructions follow Type 4 precomputation:

1. CALLDATACOPY
2. RETURNDATACOPY
3. CODECOPY
4. EXTCODECOPY

Type 4 instructions have subtle differences between themselves. We thus further subdivide the type into 3 subtypes:

INST	$^{\diamond}\mathsf{PRE}$
CODECOPY	type4CC
EXTCODECOPY	type4CC
CALLDATACOPY	type4CD
RETURNDATACOPY	type4RD

Context

We collect under the moniker Type_4 the following collection of constraints (which will further depend on the ternary column TERN). It starts with setting source and target context numbers:

1. We fix the target context, it is the current execution context: $\mathsf{CN\_T}_i = \langle \mathsf{CN} \rangle_i$;
2. The source context and exodata flags depend on the instruction:
   (a) $\textsc{if}$ $^{\diamond}\mathsf{PRE} = \texttt{type4RD}$ $\textsc{then}$ $\mathsf{CN\_S}_i = \langle \mathsf{RETURNER} \rangle_i$ and
   $$\left\{\begin{array}{l} \mathsf{X\_SHA3}_i = 0 \\ \mathsf{X\_LOG}_i = 0 \\ \mathsf{X\_ROM}_i = 0 \\ \mathsf{X\_TXCD}_i = 0 \end{array}\right.$$
   (b) $\textsc{if}$ $\langle \mathsf{INST} \rangle_i = \texttt{CALLDATACOPY}$ $\textsc{then}$ $\mathsf{CN\_S}_i = \langle \mathsf{CALLER} \rangle_i$ and
   $$\left\{\begin{array}{l} \mathsf{X\_SHA3}_i = 0 \\ \mathsf{X\_LOG}_i = 0 \\ \mathsf{X\_ROM}_i = 0 \\ \mathsf{X\_TXCD} : \left\{\begin{array}{llll} \textsc{if}\ \mathsf{IS\_}\mu_i = 1 & \textsc{and} & \mathsf{TOTRD}_{i-1} \neq 0 & \textsc{then}\ \mathsf{X\_TXCD}_i = \langle \mathsf{INFO} \rangle \\ \textsc{if}\ \mathsf{IS\_}\mu_i = 0 & \textsc{or} & \mathsf{TOTRD}_{i-1} = 0 & \textsc{then}\ \mathsf{X\_TXCD}_i = 0 \end{array}\right. \end{array}\right.$$
   Recall that we distinguish between transaction call data and call data created in CALL-type instructions.
   (c) $\textsc{if}$ $\langle \mathsf{INST} \rangle_i = \texttt{CODECOPY}$ $\textsc{or}$ $\langle \mathsf{INST} \rangle_i = \texttt{EXTCODECOPY}$ $\textsc{then}$ $\mathsf{CN\_S}_i = 0$ and
   $$\left\{\begin{array}{l} \mathsf{X\_SHA3}_i = 0 \\ \mathsf{X\_LOG}_i = 0 \\ \mathsf{X\_ROM} : \left\{\begin{array}{llll} \textsc{if}\ \mathsf{IS\_}\mu_i = 1 & \textsc{and} & \mathsf{TOTRD}_{i-1} \neq 0 & \textsc{then}\ \mathsf{X\_ROM}_i = 1 \\ \textsc{if}\ \mathsf{IS\_}\mu_i = 0 & \textsc{or} & \mathsf{TOTRD}_{i-1} = 0 & \textsc{then}\ \mathsf{X\_ROM}_i = 0 \end{array}\right. \\ \mathsf{X\_TXCD}_i = 0 \end{array}\right.$$
3. $\mathsf{OFF\_OOB}_i$ is set along with $\mathsf{TERNARY}_i$

Along with CALLDATALOAD, the above are the only instructions that may set off the $\mathsf{OFF\_OOB}$ flag. As already expanded upon elsewhere, the "data source offset" $\langle \mathsf{OFF}^2 \rangle$ of these instructions points into bytecode or calldata (we deal with return data in the following paragraph). It may very well go completely out of bounds and not provoke an exception. When it does, $\mathsf{OFF\_OOB}_i$ will be set.

The case where the "data source offset" points into return data is different: we test the fact that the byte slice it points to is in bounds before the macro-instruction ever makes it to the RAM preprocessor. Recall that out of bounds RETURNDATACOPY instructions raise an exception in the evm.

Establishing TERN

We establish the TERNARY column. Recall that it is a $\langle \mathsf{MMU}\square \rangle$-constant column. Its value determines the kinds of micro-instructions the macro-instruction is translated to. There are three cases to consider:

TERN = 0: $\langle \mathsf{OFF}^2 \rangle + (\langle \mathsf{SIZE} \rangle - 1) < \langle \mathsf{REFS} \rangle$: the instruction behaves like a type 3 instruction with a caveat about the exodata source; there is no zero padding;

TERN = 1: $\langle \mathsf{OFF}^2 \rangle < \langle \mathsf{REFS} \rangle \leq \langle \mathsf{OFF}^2 \rangle + (\langle \mathsf{SIZE} \rangle - 1)$: the instruction reads at least one byte from its source and writes it to RAM; it follows it up by writing at least one 0 padding byte;

TERN = 2: $\langle \mathsf{REFS} \rangle \leq \langle \mathsf{OFF}^2 \rangle$: the instruction writes $\langle \mathsf{SIZE} \rangle$ many zeros to memory, i.e. there is only zero padding;

The trickiest case to arithmetize is $\mathsf{TERN} = 1$. We go about establishing the value of TERN. We jump straight to the last preprocessing instruction:

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 0 \ \text{ AND } \mathsf{IS\_}\mu_{i+1} = 1}$$

1. IF $\langle \mathsf{OFF}^2 \rangle_i^{\mathsf{hi}} \neq 0$ THEN
$$\begin{cases} \mathsf{TERN}_i = 2 \\ \mathsf{OFF\_OOB}_i = 1 \end{cases}$$
$\langle \mathsf{OFF}^2 \rangle_i^{\mathsf{hi}} \neq 0$ means that $\langle \mathsf{OFF}^2 \rangle$ is grossly out of bounds.

2. IF $\langle \mathsf{OFF}^2 \rangle_i^{\mathsf{hi}} = 0$ THEN

   (a) IF $\mathsf{TERN}_i = 0$ THEN
$$\left\{ \begin{array}{rcl} \mathsf{OFF\_OOB}_i & = & 0 \\ \langle \mathsf{REFS} \rangle_i - (\langle \mathsf{OFF}^2 \rangle_i^{\mathsf{lo}} + \langle \mathsf{SIZE} \rangle_i) & = & \mathsf{ACC\_1}_i \end{array} \right.$$

   (b) IF $\mathsf{TERN}_i = 1$ THEN
$$\left\{ \begin{array}{rcl} \mathsf{OFF\_OOB}_i & = & 0 \\ \langle \mathsf{OFF}^2 \rangle_i^{\mathsf{lo}} + (\langle \mathsf{SIZE} \rangle_i - 1) - \langle \mathsf{REFS} \rangle_i & = & \mathsf{ACC\_1}_i \\ \langle \mathsf{REFS} \rangle_i - (\langle \mathsf{OFF}^2 \rangle_i^{\mathsf{lo}} + 1) & = & \mathsf{ACC\_2}_i \end{array} \right.$$

   (c) IF $\mathsf{TERN}_i = 2$ THEN
$$\left\{ \begin{array}{rcl} \mathsf{OFF\_OOB}_i & = & 1 \\ \langle \mathsf{OFF}^2 \rangle_i - \langle \mathsf{REFS} \rangle_i & = & \mathsf{ACC\_1}_i \end{array} \right.$$

   Note that $\mathsf{ACC\_1}_i$ and $\mathsf{ACC\_2}_i$ don't play a "functional role", their sole purpose is in establishing TERN_i. Note furthermore that NIB_1 and NIB_2 remain unused at this point.

2.4.7 Type 4 when TERN = 0

Preprocessing

This is essentially a subcase of $\mathsf{TERN} = 1$.

2.4.8 Type 4 when $\textsf{TERN} = 1$

Preprocessing

Type 4 instructions with $\textsf{TERN} = 1$ are the most complex to arithmetize. As usual, we jump straight to the last preprocessing step, i.e.

$$\boxed{\text{All constraints in this subsection assume } \textsf{IS\_}\mu_i = 0 \ \textsc{and} \ \textsf{IS\_}\mu_{i+1} = 1}$$

Note that in the present case ($\textsf{TERN} = 1$) preprocessing takes 3 lines. The integer whose bytes are being accumulated are small (having passed smallness testing in the Memory Expansion Module or by virtue of $\textsf{TERN} = 1$) i.e. they all fit into 3 bytes. Also notice that $\textsf{ACC\_1}$ and $\textsf{ACC\_2}$ are already "used up". In what follows we start with $\textsf{ACC\_3}$, $\textsf{ACC\_4}$, ...

Euclidean divisions. $\textsf{ACC\_3}$, ..., $\textsf{ACC\_8}$ target quotients of certain euclidean divisions and $\textsf{NIB\_3}$, ..., $\textsf{NIB\_8}$ target the associated remainders:

$$\begin{cases} \langle \textsf{REFO} \rangle_i + \langle \textsf{OFF}^2 \rangle_i & = & 16 \cdot \textsf{ACC\_3}_i + \textsf{NIB\_3}_i \\ \langle \textsf{REFO} \rangle_i + \big(\langle \textsf{REFS} \rangle_i - 1 \big) & = & 16 \cdot \textsf{ACC\_4}_i + \textsf{NIB\_4}_i \\ \langle \textsf{OFF}^1 \rangle_i & = & 16 \cdot \textsf{ACC\_5}_i + \textsf{NIB\_5}_i \\ \langle \textsf{OFF}^1 \rangle_i + \big(\big(\langle \textsf{REFS} \rangle_i - \langle \textsf{OFF}^2 \rangle_i \big) - 1 \big) & = & 16 \cdot \textsf{ACC\_6}_i + \textsf{NIB\_6}_i \\ \langle \textsf{OFF}^1 \rangle_i + \big(\langle \textsf{REFS} \rangle_i - \langle \textsf{OFF}^2 \rangle_i \big) & = & 16 \cdot \textsf{ACC\_7}_i + \textsf{NIB\_7}_i \\ \langle \textsf{OFF}^1 \rangle_i + \big(\langle \textsf{SIZE} \rangle_i - 1 \big) & = & 16 \cdot \textsf{ACC\_8}_i + \textsf{NIB\_8}_i \end{cases}$$

Note that $\textsf{ACC\_7}_i$ and $\textsf{NIB\_7}_i$ could easily deduced from $\textsf{ACC\_6}_i$ and $\textsf{NIB\_6}_i$; we don't do it and resort to this generic way of establishing $\textsf{ACC\_7}_i$ and $\textsf{NIB\_7}_i$ to keep things simpler.

Comparisons. We justify the bit columns $[\![1]\!]$ and $[\![2]\!]$ and use up $\textsf{NIB\_1}$ and $\textsf{NIB\_2}$ in the process:

$$\begin{cases} \big(\textsf{NIB\_5}_i - \textsf{NIB\_3}_i \big) \cdot \big(2 \cdot [\![1]\!]_i - 1 \big) - [\![1]\!]_i & = & \textsf{NIB\_1}_i \\ \big(\textsf{NIB\_4}_i - \textsf{NIB\_6}_i \big) \cdot \big(2 \cdot [\![2]\!]_i - 1 \big) - [\![2]\!]_i & = & \textsf{NIB\_2}_i \end{cases}$$

Thus

$$\begin{cases} [\![1]\!] = 1 & \iff & \textsf{NIB\_5} > \textsf{NIB\_3} \\ [\![2]\!] = 1 & \iff & \textsf{NIB\_4} > \textsf{NIB\_6} \end{cases}$$

Workflow parameters. We set the total number of micro-instructions:

$$\begin{aligned} \textsf{TOT}^{\mu}_i \quad = \quad & (\textsf{ACC\_4}_i - \textsf{ACC\_3}_i) + 1 \\ & +(\textsf{ACC\_8}_i - \textsf{ACC\_7}_i) + 1 \end{aligned}$$

We also set the total number of instructions involving actual reads:

$$\textsf{TOTRD}_i = (\textsf{ACC\_4}_i - \textsf{ACC\_3}_i) + 1$$

Let us also write, *just this once*,

$$\textsf{TOTPD}_i = (\textsf{ACC\_8}_i - \textsf{ACC\_7}_i) + 1$$

We emphasize that we don't need a dedicated $\textsf{TOTPD}$ column , but it's convenient to understand the meaning of $[\![4]\!]$ below. The interpretation is straightforward: $\textsf{TOTPD}$ is the number of target

RAM limbs that will be affected by 0 padding. We set some binary flags:

$$\begin{cases} \text{IF } \mathsf{NIB\_3}_i = \mathsf{NIB\_5}_i \text{ THEN } \mathsf{ALIGNED}_i = 1 \\ \text{IF } \mathsf{NIB\_3}_i \neq \mathsf{NIB\_5}_i \text{ THEN } \mathsf{ALIGNED}_i = 0 \\[1ex] \text{IF } \mathsf{TOTRD}_i = 1 \text{ THEN } \llbracket 3 \rrbracket_i = 1 \\ \text{IF } \mathsf{TOTRD}_i \neq 1 \text{ THEN } \llbracket 3 \rrbracket_i = 0 \\[1ex] \text{IF } \mathsf{TOTPD}_i = 1 \text{ THEN } \llbracket 4 \rrbracket_i = 1 \\ \text{IF } \mathsf{TOTPD}_i \neq 1 \text{ THEN } \llbracket 4 \rrbracket_i = 0 \\[1ex] \text{IF } \mathsf{NIB\_6}_i = 15 \text{ THEN } \llbracket 5 \rrbracket_i = 1 \\ \text{IF } \mathsf{NIB\_6}_i \neq 15 \text{ THEN } \llbracket 5 \rrbracket_i = 0 \\[1ex] \text{IF } \mathsf{NIB\_8}_i = 15 \text{ THEN } \llbracket 6 \rrbracket_i = 1 \\ \text{IF } \mathsf{NIB\_8}_i \neq 15 \text{ THEN } \llbracket 6 \rrbracket_i = 0 \end{cases}$$

We also establish $\llbracket 7 \rrbracket$. This plays an analoguous role for `type 4` instructions as $\llbracket 5 \rrbracket$ played for `type 2` instructions: in case of a single read (i.e. $\mathsf{TOTRD}_i = 1$ i.e. $\llbracket 3 \rrbracket = 1$) it distinguishes between the two writing methods. Either a source chunk is written to a suffix and prefix of two consecutive target limbs ($\iff$ $\big(\mathsf{NIB\_5} + \big(\mathsf{NIB\_4} - \mathsf{NIB\_3} + 1\big) - 1\big) \geq 16$) or a source chunk is written to a single target limb replacing a chunk thereing ($\iff$ $\big(\mathsf{NIB\_5} + \big(\mathsf{NIB\_4} - \mathsf{NIB\_3} + 1\big) - 1\big) < 16$)

1. IF $\llbracket 3 \rrbracket = 0$ THEN $\llbracket 7 \rrbracket = 0$
2. IF $\llbracket 3 \rrbracket = 1$ THEN
$$\mathsf{NIB\_5} + \big(\mathsf{NIB\_4} - \mathsf{NIB\_3}\big) - 16 \cdot \llbracket 7 \rrbracket_i = \mathsf{NIB\_9}$$

In other words:

1. $\mathsf{ALIGNED} = 1 \iff$ the data source and RAM target offsets are aligned;
2. $\llbracket 3 \rrbracket = 1 \iff$ precisely one limb of call data, return data or bytecode is read;
3. $\llbracket 4 \rrbracket = 1 \iff$ precisely one target RAM limb has to be zero padded;
4. $\llbracket 5 \rrbracket = 1 \iff \mathsf{NIB\_6} = 15 \iff \mathsf{NIB\_7} = 0 \iff$ the first padding operation starts on a fresh limb with a byte offset of 0;
5. $\llbracket 6 \rrbracket = 1 \iff \mathsf{NIB\_8} = 15 \iff$ the final padding operation ends with a byte offset of 15;

Source and target limb and byte offsets We set source and target limb and byte offsets

$$\begin{cases} \mathsf{SLO}_{i+1} &= & \mathsf{SLO}_i &= & \mathsf{ACC\_3}_i \\ \mathsf{SBO}_{i+1} &= & \mathsf{SBO}_i &= & \mathsf{NIB\_3}_i \\[1ex] \mathsf{TLO}_{i+1} &= & \mathsf{TLO}_i &= & \mathsf{ACC\_5}_i \\ \mathsf{TBO}_{i+1} &= & \mathsf{TBO}_i &= & \mathsf{NIB\_5}_i \end{cases}$$

Note that we don't require source offsets: there is no source, we are simply writing zeros to the target context's RAM. Note furthermore that the constraint $\mathsf{TLO}_{i+1} = \mathsf{TLO}_i$ is implicit in the upcoming set of constraints. We include it purely for the reader's convenience.

Micro-instruction writing: updating TOTRD

We distinguish several cases. A complication arises from the fact that midway there is a regime change. We initially read data and write the micro-instructions that will surgically insert the relevant data into the target context's RAM. This regime holds for as long as $\mathsf{TOTRD}_{i-1} \neq 0$. The regime change takes place as we transition from row i_0 to row $i_0 + 1$ where i_0 is the row index where TOTRD where hits zero for the first time (within that $\langle \mathsf{MMU}\square \rangle$). At that point the micro-instructions the zk-evm writes switch from **data extracting micro-instructions** to **zero padding micro-instructions**. Note: we don't use the notation i_0 anywhere else. The transition condition will be couched in terms of TOTRD

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1}$$

We begin by fixing the expected behaviour of TOTRD

1. $\text{IF } \mathsf{TOTRD}_{i-1} \neq 0 \text{ THEN } \mathsf{TOTRD}_i = \mathsf{TOTRD}_{i-1} - \mathsf{IS\_}\mu_{i-1}$
2. $\text{IF } \mathsf{TOTRD}_{i-1} = 0 \text{ THEN } \mathsf{TOTRD}_i = 0$

In other words: for the first micro-instruction TOTRD duplicates the value that was established in precomputation. Beyond that point it decreases monotonically by 1 with every micro-instruction until it hits 0. We deal with the micro instructions in the first phase, i.e. reading actual data.

Micro-instruction writing: data extraction

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1 \text{ and } \mathsf{TOTRD}_{i-1} \neq 0}$$

We begin with the case where there is a single "data writing" operation, i.e. $[\![3]\!]_i = 1$:

1. $\text{IF } [\![3]\!]_i = 1 \text{ THEN }$:
   (a) SLO_i and SBO_i are already set;
   (b) TLO_i and TBO_i are already set;
   (c) $\mathsf{SIZE}_i = \mathsf{NIB\_4}_i - \mathsf{NIB\_3}_i + 1$;
   (d) $\text{IF } [\![1]\!]_i = 0 \text{ THEN}$

$$\begin{cases} \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CC} \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4RD} \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CD} \text{ THEN } \begin{cases} \text{IF } \langle \mathsf{INFO} \rangle = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } \langle \mathsf{INFO} \rangle = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \end{cases} \end{cases}$$

   (e) $\text{IF } [\![1]\!]_i = 1 \text{ THEN}$
      i. $\text{IF } [\![7]\!]_i = 0 \text{ THEN}$

$$\begin{cases} \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CC} \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4RD} \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CD} \text{ THEN } \begin{cases} \text{IF } \langle \mathsf{INFO} \rangle = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } \langle \mathsf{INFO} \rangle = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \end{cases} \end{cases}$$

      ii. $\text{IF } [\![7]\!]_i = 1 \text{ THEN}$

$$\begin{cases} \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CC} \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4RD} \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \text{IF } {}^{\diamond}\mathsf{PRE}_i = \texttt{type4CD} \text{ THEN } \begin{cases} \text{IF } \langle \mathsf{INFO} \rangle = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \text{IF } \langle \mathsf{INFO} \rangle = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \end{cases} \end{cases}$$

(f) IF $[\![7]\!]_i = 0$ THEN

$$\begin{cases} \mathsf{TLO}_{i+1} = [\![5]\!]_i + \mathsf{TLO}_i \\ \mathsf{TBO}_{i+1} = \mathsf{NIB\_7}_i \end{cases}$$

i.e. if the one operation touches a single limb in the target RAM then we move to the next limb *iff* $\mathsf{NIB\_7}_i = 0$ i.e. $[\![5]\!]_i = 1$

(g) IF $[\![7]\!]_i = 1$ THEN

$$\begin{cases} \mathsf{TLO}_{i+1} = 1 + \mathsf{TLO}_i \\ \mathsf{TBO}_{i+1} = \mathsf{NIB\_7}_i \end{cases}$$

i.e. if the one operation touches a two limbs in the target RAM then necessarily we move to the second limb that we just modified.

Recall that $[\![3]\!]_i = 1$ corresponds to a the single surgery involving actual data (i.e. at the last row of preprocessing, which is row $i-1$, $\mathsf{TOTRD}_{i-1} = 1$) so that in the current row (i.e. row i) $\mathsf{TOTRD}_i = 0$. In other words, the constraints in this block apply for a single row.

We next move to the case where there is are multiple "actual data" writing micro-instructions. We begin with the case where there is a single writing operation:

2. IF $[\![3]\!]_i = 0$ we start with the updates to TLO:

(a) IF $\mathsf{IS\_}\mu_{i-1} = 0$ THEN

$$\mathsf{TLO}_{i+1} = \mathsf{TLO}_i + (\mathsf{ALIGNED}_i + [\![1]\!]_i)$$

Note that $\mathsf{ALIGNED}_i + [\![1]\!]_i = 1 \iff \mathsf{NIB\_5} \geq \mathsf{NIB\_3}$

(b) IF $\mathsf{IS\_}\mu_{i-1} = \mathsf{IS\_}\mu_i = \mathsf{IS\_}\mu_{i+1} = 1$ THEN

$$\mathsf{TLO}_{i+1} = \mathsf{TLO}_i + 1$$

The middle condition $\mathsf{IS\_}\mu_i = 1$ is redundant (it is part of the section wide assumptions) but we include it for clarity;

The previous two constraints signify that if $\mathsf{NIB\_5} \geq \mathsf{NIB\_3}$ then TLO_i grows by one with every micro instruction. When $\mathsf{NIB\_5} < \mathsf{NIB\_3}$ the first limb in the target is modified by two successive micro-instructions. The above constraints capture this.

(c) IF $\mathsf{IS\_}\mu_{i-1} = 0$ i.e. we deal here with the first micro-instruction:

i. $\mathsf{SIZE}_i = (15 - \mathsf{NIB\_3}_i) + 1$ **We could put 16 ...**

ii. IF $[\![1]\!]_i = 0$ THEN

$$\begin{cases} \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4CC} \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \\ \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4RD} \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4CD} \text{ THEN } \begin{cases} \text{IF } \langle\mathsf{INFO}\rangle_i = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \text{IF } \langle\mathsf{INFO}\rangle_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \end{cases} \end{cases}$$

iii. IF $[\![1]\!]_i = 1$ THEN

$$\begin{cases} \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4CC} \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \\ \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4RD} \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \text{IF } {}^\Diamond\mathsf{PRE}_i = \texttt{type4CD} \text{ THEN } \begin{cases} \text{IF } \langle\mathsf{INFO}\rangle_i = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \text{IF } \langle\mathsf{INFO}\rangle_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \end{cases} \end{cases}$$

(d) IF $\mathsf{IS\_}\mu_{i-1} = 1$ AND $\mathsf{TOTRD}_{i-1} \neq 0$ THEN

i. $\mathsf{SBO}_i = 0$

ii. $\mathsf{TBO}_i = \mathsf{NIB\_5}_i + \mathsf{SIZE}_i - 16 \cdot (\mathsf{ALIGNED}_i + [\![1]\!]_i)$. Note that by construction, *and for type 4 instructions*, $\mathsf{ALIGNED}_i$ and $[\![1]\!]_i$ measure disjoint events. Thus $\mathsf{ALIGNED}_i + [\![1]\!]_i = \mathsf{ALIGNED}_i + [\![1]\!]_i - \mathsf{ALIGNED}_i \cdot [\![1]\!]_i = \mathsf{ALIGNED}_i \vee [\![1]\!]_i$ is binary. Its interpretation is $\mathsf{ALIGNED}_i + [\![1]\!]_i = 1 \iff \mathsf{NIB\_5}_i \geq \mathsf{NIB\_3}_i$.

iii. $\textsc{if}\ \mathsf{TOTRD}_i \neq 0\ \textsc{then}$

A. $\mathsf{SIZE}_i = 16$ i.e. we copy full limbs,

B. $\textsc{if}\ \mathsf{ALIGNED}_i = 1\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRam}$

$$\begin{cases} \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CC}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRam} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4RD}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRam} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CD}\ \textsc{then}\ \begin{cases} \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 0\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRam} \\ \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 1\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRam} \end{cases} \end{cases}$$

C. $\textsc{if}\ \mathsf{ALIGNED}_i = 0$

$$\begin{cases} \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CC}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4RD}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CD}\ \textsc{then}\ \begin{cases} \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 0\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 1\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \end{cases} \end{cases}$$

iv. $\textsc{if}\ \mathsf{TOTRD}_i = 0\ \textsc{then}$

A. $\mathsf{SIZE}_i = \mathsf{NIB\_4}_i + 1$

B. $\textsc{if}\ [\![2]\!]_i = 0\ \textsc{then}$

$$\begin{cases} \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CC}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4RD}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CD}\ \textsc{then}\ \begin{cases} \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 0\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideChunk} \\ \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 1\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideChunk} \end{cases} \end{cases}$$

C. $\textsc{if}\ [\![2]\!]_i = 1\ \textsc{then}$

$$\begin{cases} \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CC}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4RD}\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \textsc{if}\ {}^{\Diamond}\mathsf{PRE}_i = \texttt{type4CD}\ \textsc{then}\ \begin{cases} \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 0\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{RamToRamSlideOverlappingChunk} \\ \textsc{if}\ \langle\mathsf{INFO}\rangle_i = 1\ \textsc{then}\ \mu\mathsf{INST}_i = \texttt{ExoToRamSlideOverlappingChunk} \end{cases} \end{cases}$$

D. Below we update TLO for the padding phase of the macro-instruction decoding.

$$\textsc{if}\ [\![2]\!]_i = 1\ \textsc{then}\ \begin{cases} \mathsf{TLO}_{i+1} = 1 + \mathsf{TLO}_i \\ \mathsf{TBO}_{i+1} = \mathsf{NIB\_7}_i \end{cases}$$

in other words, if $\mathsf{NIB\_6} < \mathsf{NIB\_4}$ then the final data writing micro-instruction ($\mathsf{TOTRD}_{i-1} \neq 0$, $\mathsf{TOTRD}_i = 0$) writes on two consecutive limbs (TLO_i and $1 + \mathsf{TLO}_i$) and hence in the first padding operation we will be writing on $1 + \mathsf{TLO}_i$.

$$\textsc{if}\ [\![2]\!]_i = 0\ \textsc{then}\ \begin{cases} \mathsf{TLO}_{i+1} = [\![5]\!]_i + \mathsf{TLO}_i \\ \mathsf{TBO}_{i+1} = \mathsf{NIB\_7}_i \end{cases}$$

Otherwise the final data writing micro-instruction wrote data within the same limb (with offset TLO_i). **Actually, the two events $\{[\![2]\!]_i = 1\}$ and $\{[\![5]\!]_i = 1\}$ are mutually exclusive. So we could (and should) replace this with a single constraint $\mathsf{TLO}_{i+1} = ([\![2]\!]_i + [\![5]\!]_i) + \mathsf{TLO}_i$.**

Micro-instruction writing: zero padding

We now start with the padding phase of the micro-instruction writing.

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1 \text{ and } \mathsf{TOTRD}_{i-1} = 0}$$

We again distinguish two cases: the case where a single limb in the target context's RAM needs to be padded (i.e. $\mathsf{TOTPD}_i = 1$ i.e. $[\![4]\!]_i = 1$) and the case where at least 2 (consecutive) limbs in the target context's RAM need to be padded (i.e. $\mathsf{TOTPD}_i > 1$ i.e. $[\![4]\!]_i = 0$). Note that we use the TOTPD_i name again. We refer the reader to 2.4.8 for the definition and interpretation of this quantity.

In the first case there is only one interesting scenario: when $\mathsf{NIB\_7} = 0$ and $\mathsf{NIB\_8} = 15$. In this case we can perform a fast "limb killing" operation. Otherwise we need to excise a chunk of bytes from a RAM limb.

1. IF $[\![4]\!]_i = 1$ THEN
   (a) we have already set TLO_i and TBO_i;
   (b) out of precaution, we set $\mathsf{SLO}_i = \mathsf{SBO}_i = 0$;
   (c) IF $\Big([\![5]\!]_i = 1$ AND $[\![6]\!] = 1\Big)$ THEN $\mu\mathsf{INST}_i = \texttt{KillingOne}$
   (d) IF $\Big([\![5]\!]_i = 0$ OR $[\![6]\!] = 0\Big)$ THEN
      i. $\mathsf{TBO}_i = \mathsf{NIB\_7}_i$
      ii. $\mathsf{SIZE}_i = \mathsf{NIB\_8}_i - \mathsf{NIB\_7}_i + 1$
      iii. $\mu\mathsf{INST}_i = \texttt{RamLimbExcision}$

In the second case we write to at least two words in the target context's RAM. There is a first write that may be fast (if $\mathsf{NIB\_7} = 0$ i.e. if $[\![5]\!]_i = 1$) otherwise it's excision of a suffix, it is followed up by 0 or more full limb killings (which are fast), and the final limb is similar to the first (if $\mathsf{NIB\_8} = 15$ i.e. if $[\![6]\!] = 1$.)

1. IF $[\![4]\!]_i = 0$ THEN
   (a) IF $\mathsf{TOTRD}_{i-2} \neq 0$ THEN

$$\begin{cases} \mathsf{TBO}_i = \mathsf{NIB\_7}_i \\ \mathsf{SIZE}_i = 16 - \mathsf{NIB\_7}_i \\ \mu\mathsf{INST}_i = \begin{cases} \text{IF } [\![5]\!]_i = 1: & \texttt{KillingOne} \\ \text{IF } [\![5]\!]_i = 0: & \texttt{RamLimbExcision} \end{cases} \end{cases}$$

   Note that the constraint "$\mathsf{TBO}_i = \mathsf{NIB\_7}_i$" is redundant: we have already imposed as much at the end of the data writing phase; we repeat it here for sheer convenience. Note furthermore that we really are in the case where $\mathsf{TOTRD}_{i-2} = 1$, $\mathsf{TOTRD}_{i-1} = 0$ and $\mathsf{TOTRD}_i = 0$.

   (b) IF $\mathsf{TOTRD}_{i-2} = 0$ THEN
      i. $\mathsf{TLO}_i = \mathsf{TLO}_{i-1} + 1$
      ii. $\mathsf{TBO}_i = 0$
      iii. IF $\mathsf{TOT}^{\mu}_i \neq 0$ THEN $\mu\mathsf{INST}_i = \texttt{KillingOne}$
      iv. IF $\mathsf{TOT}^{\mu}_i = 0$ THEN

$$\begin{cases} \text{IF } [\![6]\!]_i = 0 \text{ THEN } \begin{cases} \mathsf{SIZE}_i = \mathsf{NIB\_8}_i + 1 \\ \mu\mathsf{INST}_i = \texttt{RamLimbExcision} \end{cases} \\ \text{IF } [\![6]\!]_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{KillingOne} \end{cases}$$

2.4.9 Type 4 when $\mathsf{TERN} = 2$

Preprocessing

Type 4 instructions with $\mathsf{TERN} = 2$ are the simplest Type 4 RAM macro-instructions to decompose into a sequence of micro-instructions. They correspond to grossly out of bounds offsets. The net effect on memory is just to write $\langle\mathsf{SIZE}\rangle$ many zeros starting at offset $\langle\mathsf{OFF}^1\rangle$. As per usual, we jump straight to the last preprocessing step.

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 0 \ \ \textsc{and}\ \mathsf{IS\_}\mu_{i+1} = 1}$$

Note that in the present case ($\mathsf{TERN} = 2$) preprocessing takes 16 lines. Thus the accumulators *below* have accumulated 16 bytes. Nonetheless, the integers whose bytes are being accumulated are small (having passed smallness testing in the Memory Expansion Module) i.e. they fit into 3 bytes. $\mathsf{ACC\_3}$ and $\mathsf{ACC\_4}$ are thus small (i.e. 3 byte integers.) Note furthermore that we don't use $\mathsf{ACC\_2}$ (even though it is "unused and available" in this execution branch.)

Euclidean divisions. $\mathsf{ACC\_3}$ and $\mathsf{ACC\_4}$ target quotients of certain euclidean divisions and $\mathsf{NIB\_3}$ and $\mathsf{NIB\_4}$ target the associated remainders:

$$\begin{cases} \langle\mathsf{OFF}^1\rangle_i & = & 16 \cdot \mathsf{ACC\_3}_i + \mathsf{NIB\_3}_i \\ \langle\mathsf{OFF}^1\rangle_i + (\langle\mathsf{SIZE}\rangle_i - 1) & = & 16 \cdot \mathsf{ACC\_4}_i + \mathsf{NIB\_4}_i \end{cases}$$

Workflow parameters. We set the total number of micro-instructions:

$$\mathsf{TOT}^{\mu}_i = \mathsf{ACC\_4}_i - \mathsf{ACC\_3}_i + 1$$

We set some binary flags:

$$\begin{cases} \textsc{if}\ \mathsf{TOT}^{\mu}_i = 1\ \textsc{then}\ [\![1]\!]_i = 1 \\ \textsc{if}\ \mathsf{TOT}^{\mu}_i \neq 1\ \textsc{then}\ [\![1]\!]_i = 0 \\ \\ \textsc{if}\ \mathsf{NIB\_3}_i = 0\ \textsc{then}\ [\![3]\!]_i = 1 \\ \textsc{if}\ \mathsf{NIB\_3}_i \neq 0\ \textsc{then}\ [\![3]\!]_i = 0 \\ \\ \textsc{if}\ \mathsf{NIB\_4}_i = 15\ \textsc{then}\ [\![4]\!]_i = 1 \\ \textsc{if}\ \mathsf{NIB\_4}_i \neq 15\ \textsc{then}\ [\![4]\!]_i = 0 \end{cases}$$

In other words: the $\langle\mathsf{MMU}\square\rangle$-constant binary column $[\![1]\!]$ lights up precisely when the RAM macro-instruction decomposes into a single micro-instruction; the $\langle\mathsf{MMU}\square\rangle$-constant binary columns $[\![3]\!]$ and $[\![4]\!]$ aren't all that important; their main purpose is to indicate, when $[\![1]\!] = 0$, i.e. when the RAM macro-instruction decomposes into at least 2 micro-instructions, whether the first and final instructions are fast or not.

We set source and target limb and byte offsets

$$\begin{cases} \mathsf{TLO}_{i+1} & = & \mathsf{TLO}_i & = & \mathsf{ACC\_3}_i \\ \mathsf{TBO}_{i+1} & = & \mathsf{TBO}_i & = & \mathsf{NIB\_3}_i \end{cases}$$

Note that we don't require source offsets: there is no source, we are simply writing zeros to the target context's RAM. Note furthermore that the constraint $\mathsf{TLO}_{i+1} = \mathsf{TLO}_i$ is implicit in the upcoming set of constraints. We include it purely for the reader's convenience.

Micro-instruction writing

We distinguish several cases. Note that

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1}$$

1. $\mathsf{TLO}_i = \mathsf{TLO}_{i-1} + \mathsf{IS\_}\mu_{i-1}$: the source limb offset grows by 1 with every instruction, regardless of anything else;
2. IF $[\![1]\!]_i = 1$ THEN
   (a) TLO_i and TBO_i were already set
   (b) IF $[\![3]\!]_i = 1$ AND $[\![4]\!]_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{KillingOne}$
   (c) IF $[\![3]\!]_i = 0$ OR $[\![4]\!]_i = 0$ THEN
   $$\begin{cases} \mathsf{SIZE}_i = \langle \mathsf{SIZE} \rangle_i \\ \mu\mathsf{INST}_i = \mathtt{RamLimbExcision} \end{cases}$$

   Recall that the case $[\![1]\!]_i = 1$ corresponds to establishing $\mathsf{TOT}^\mu = 1$ during the precomputation phase (i.e. a single surgery is required to carry out the macro-instruction.) This single constraint is sufficient.
3. IF $[\![1]\!]_i = 0$ the situation is more complex. By definition the macro-instruction is converted to $\mathsf{TOT}^\mu \geq 2$ micro instructions, the first and last of which may be excisions, and all intermediary ones being replacing full RAM limbs with zero. This logic is captured below:
   (d) IF $\mathsf{IS\_}\mu_{i-1} = 0$ THEN
      i. IF $[\![3]\!]_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{KillingOne}$, i.e. target byte offset of the first micro-instruction is zero and we perform at least 2 micro-instructions: the first operation thus erases an entire limb;
      ii. IF $[\![3]\!]_i = 0$ THEN
         A. $\mathsf{SIZE}_i = (15 - \mathsf{NIB\_1}_i) + 1$
         B. $\mu\mathsf{INST}_i = \mathtt{RamLimbExcision}$

      In other words, at the first micro-instruction can be either killing a whole limb (if $[\![3]\!]_i = 1$) or excising a suffix (if $[\![3]\!]_i = 0$)
   (e) IF $\mathsf{IS\_}\mu_{i-1} = 1$ THEN
      i. $\mathsf{TBO}_i = 0$ i.e. after the first micro instruction we are killing words or excising prefixes;
      ii. IF $\mathsf{TOT}^\mu_i \neq 0$ THEN $\mu\mathsf{INST}_i = \mathtt{KillingOne}$
      iii. IF $\mathsf{TOT}^\mu_i = 0$ THEN
         A. IF $[\![4]\!]_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{KillingOne}$
         B. IF $[\![4]\!]_i = 0$ THEN
         $$\begin{cases} \mathsf{SIZE}_i = \mathsf{NIB\_4}_i \\ \mu\mathsf{INST}_i = \mathtt{RamLimbExcision} \end{cases}$$
      iv. IF $\mathsf{TOT}^\mu_i = 0$ THEN
         A. $\mathsf{SIZE}_i = \mathsf{NIB\_2}_i + 1$
         B. IF $[\![3]\!]_i = 0$ THEN $\mu\mathsf{INST}_i = \mathtt{RamToRamSlideChunk}$
         C. IF $[\![3]\!]_i = 1$ THEN $\mu\mathsf{INST}_i = \mathtt{RamToRamSlideOverlappingChunk}$

2.4.10 Type 5

Instructions

This subsection deals with the preprocessing of Type 5 macro-instructions. There is only *one* such instruction, `CALLDATALOAD`. We note at this point that for a `CALLDATALOAD` to make it to preprocessing it must not have been dealt with by the Rare Checks Module. As a consequence its offset parameter will *always* satisfy

$$0 \leq \mathsf{OFFSET} < \mathsf{CDS}.$$

so that at least one byte of call data will be written to stack (with appropriate zero padding if necessary i.e. if $\mathsf{CDS} \leq \mathsf{OFFSET} + (32 - 1)$.) The number of "actual" bytes to copy is the first thing we establish below. This number *always* lives in the range $\{1, 2, \ldots, 32\}$ (i.e. 0 is excluded by what precedes.)

It might come as a surprise that there is an entire type for a single RAM instruction, especially given `CALLDATALOAD`'s apparent kinship with `MLOAD`. From the point of view of the zk-evm presented in these notes the instructions are very different. We provide further motivation for this this design choice in the chapter on RAM data processing; for now note that while is only one type 5 instruction, there are *two* different scenarios to consider. The first one is when the current call stack depth is > 1. In this case call data is located in the present context's caller's RAM. In this case the `CALLDATALOAD` macro-instruction will be converted into a single "RAM to stack" micro-instruction. The second case is that when call stack depth is $= 1$. In this case the call data (**transaction call data**, really) must be extracted from a public commitment. Retrieval is more complex in this case since, as explained in the chapter on RAM data processing, the zk-evm must load the 2 or 3 relevant limbs, temporarily store them in the 0^{th} context's RAM (overcoming its memorylessness by means of the $\langle \mathsf{EXCEPTIONAL\_RETENTION\_FLAG} \rangle$) and only then can it start writing to the imported stack value. In this case the `CALLDATALOAD` macro-instruction will be converted into a series of 3 "loading from exogenous data" micro instruction and a single "RAM to stack".

We further note that the public commitment to Transaction Calldata enforces the following padding scheme: (1) zero pad to the next multiple of 16 (2) then add two zero limbs. In other words, beyond the final byte of actual call data there are at least 32 bytes of zero padding. Given that when a `CALLDATALOAD` instruction which makes it to preprocessing is reading at least one actual byte from call data, the zk-evm will *always* be able to load 3 consecutive limbs of exogenous data from transaction call data without going out of bounds (and breaking the plookup connection.) We note at this point that we could have been fancier and only load 1, 2 or 3 limbs from transaction call data depending on whether $\mathsf{OFFSET} + 16 \geq \mathsf{CDS}$ or not. This optimization comes at further complication in the RAM preprocessor and saves 1 or 2 rows in the RAM data processor.

Preprocessing

As per usual, we jump straight to the last preprocessing step.

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 0 \ \ \textsc{and} \ \mathsf{IS\_}\mu_{i+1} = 1}$$

We begin establishing some parameters.

Setting $\mathsf{OFF\_OOB}_i$. We set $\mathsf{OFF\_OOB}_i = 0$, see previous discussion.

Setting context info. We set $\mathsf{CN\_T}_i = 0$ and

$$\begin{cases} \textsc{if} \ \mathsf{CSD}_i = 1 \ \textsc{then} \ \mathsf{CN\_S}_i = 0 \\ \textsc{if} \ \mathsf{CSD}_i \neq 1 \ \textsc{then} \ \mathsf{CN\_S}_i = \langle \mathsf{CALLER} \rangle_i \end{cases}$$

Note that, given the micro instruction we will be writing, setting $\mathsf{CN\_T}_i$ serves no purpose and can be omitted.

Establishing maximum offset. We first establish the maximum offset of a byte to be copied from call data and the number of bytes to copy, i.e. we require that:

$$\big(2 \cdot \llbracket 1 \rrbracket_i - 1\big) \cdot \big(\mathsf{CDS}_i - (\langle \mathsf{OFF}^1 \rangle_i + 32)\big) + \big(\llbracket 1 \rrbracket_i - 1\big) = \mathsf{ACC\_1}_i$$

Let write, *out of sheer convenience*, $\mathsf{NBYTES}_i = 1 + \big(2 \cdot \llbracket 1 \rrbracket_i - 1\big) \cdot \big(\mathsf{CDS}_i - (\langle \mathsf{OFF}^1 \rangle_i + 32)\big) + \big(\llbracket 1 \rrbracket_i - 1\big) = 1 + \mathsf{ACC\_1}_i$. By construction

$$\begin{cases} \llbracket 1 \rrbracket_i = 1 \iff \langle \mathsf{OFF}^1 \rangle_i + (32 - 1) \leq (\mathsf{CDS}_i - 1) \\ \llbracket 1 \rrbracket_i = 0 \iff \langle \mathsf{OFF}^1 \rangle_i + (32 - 1) > (\mathsf{CDS}_i - 1) \\ \mathsf{NBYTES} \in \{1, 2, \dots, 32\} \end{cases}$$

We will be interested in finding out whether $\mathsf{NBYTES}_i < 16$, $\mathsf{NBYTES}_i = 16$, $16 < \mathsf{NBYTES}_i < 32$ or $\mathsf{NBYTES}_i = 32$. We thus impose

$$\mathsf{ACC\_1}_i = 16 \cdot \llbracket 2 \rrbracket_i + \mathsf{NIB\_2}_i$$

which establishes the euclidean division of $\mathsf{ACC\_1}_i$ by 16 (note that in the present case $\mathsf{ACC\_1} \in \{0, 1, \dots, 31\}$ and so the quotient is either 0 or 1). Next we establish $\llbracket 3 \rrbracket$:

$$\begin{cases} \text{\textsc{if}}\ \mathsf{NIB\_2}_i \neq 15\ \text{\textsc{then}}\ \llbracket 3 \rrbracket_i = 0 \\ \text{\textsc{if}}\ \mathsf{NIB\_2}_i = 15\ \text{\textsc{then}}\ \llbracket 3 \rrbracket_i = 1 \end{cases}$$

In other words,

$\llbracket 2 \rrbracket_i$	$\llbracket 3 \rrbracket_i$		
0	0	$\iff$	$0 < \mathsf{NBYTES}_i < 16$
0	1	$\iff$	$\mathsf{NBYTES}_i = 16$
1	0	$\iff$	$16 < \mathsf{NBYTES}_i < 32$
1	1	$\iff$	$\mathsf{NBYTES}_i = 32$

Establishing TOT^μ. We impose that

$$\begin{cases} \text{\textsc{if}}\ \mathsf{CSD}_i = 1\ \text{\textsc{then}}\ \mathsf{TOT}^\mu_i = 4 \\ \text{\textsc{if}}\ \mathsf{CSD}_i \neq 1\ \text{\textsc{then}}\ \mathsf{TOT}^\mu_i = 1 \\ \mathsf{NBYTES}_i \in \{1, 2, \dots, 32\} \end{cases}$$

as already mentioned, a `CALLDATALOAD` instruction in a root context requires 3 loads from transaction call data.

Establishing alignment. We establish the euclidean division (by 16) of the **absolute offset** where reading call data begins

$$\mathsf{CDO}_i + \langle \mathsf{OFF}^1 \rangle_i = 16 \cdot \mathsf{ACC\_3}_i + \mathsf{NIB\_3}_i$$

We define associated binary flags

$$\begin{cases} \text{\textsc{if}}\ \mathsf{NIB\_3}_i = 0\ \text{\textsc{then}}\ \mathsf{ALIGNED}_i = 1 \\ \text{\textsc{if}}\ \mathsf{NIB\_3}_i \neq 0\ \text{\textsc{then}}\ \mathsf{ALIGNED}_i = 0 \end{cases}$$

Establishing $\llbracket 4 \rrbracket$. The bit column $\llbracket 4 \rrbracket$ is used to distinguish between the two ways of producing a limb containing both data and padding in the non aligned case. It only matters if $\mathsf{ALIGNED}_i = 0$. We therefore ask that $\text{\textsc{if}}\ \mathsf{ALIGNED}_i = 0\ \text{\textsc{then}}$

$$\big(2 \cdot \llbracket 4 \rrbracket_i - 1\big) \cdot \Big(\big(\mathsf{NIB\_2}_i + 1\big) - \big(15 - \mathsf{NIB\_3}_i + 1\big)\Big) - \llbracket 4 \rrbracket_i = \mathsf{NIB\_4}_i$$

In other words, given that $\mathsf{ALIGNED}_i = 0$ we have

$$\begin{cases} [\![4]\!]_i = 1 \iff \big(15 - \mathsf{NIB\_3}_i + 1\big) < \big(\mathsf{NIB\_2}_i + 1\big) \\ [\![4]\!]_i = 0 \iff \big(15 - \mathsf{NIB\_3}_i + 1\big) \geq \big(\mathsf{NIB\_2}_i + 1\big) \end{cases}$$

Establishing source and target offsets. No surprise here:

$$\begin{cases} \mathsf{SLO}_i & = & \mathsf{SLO}_{i+1} & = & \mathsf{ACC\_3}_i \\ \mathsf{SBO}_i & = & \mathsf{SBO}_{i+1} & = & \mathsf{NIB\_3}_i \\ \mathsf{TLO}_i & = & \mathsf{TLO}_{i+1} & = & 0 \\ \mathsf{TBO}_i & = & \mathsf{TBO}_{i+1} & = & 0 \end{cases}$$

Micro-instruction writing

We move on to micro-instruction writing.

$$\boxed{\text{All constraints in this subsection assume } \mathsf{IS\_}\mu_i = 1}$$

We first consider the case $\mathsf{CSD}_i \neq 1$ i.e. of call data inherited from a `CALL`-type instruction: there is nothing left to do (besides the writing the one (and only) micro-instruction). We defer it. We now consider the case $\mathsf{CSD}_i = 1$ i.e. the case of transaction call data

1. IF $\mathsf{CSD}_i = 1$ THEN
   (a) IF $\mathsf{IS\_}\mu_{i-1} = 0$ THEN

$$\begin{cases} \mu\mathsf{INST}_i & = & \texttt{StoreXinAthreeRequired} \\ \mu\mathsf{INST}_{i+1} & = & \texttt{StoreXinB} \\ \mu\mathsf{INST}_{i+2} & = & \texttt{StoreXinC} \end{cases}$$

   (b) We set limb and byte offsets, exo data flags, sizes and the $\mathsf{EXCEPTIONAL\_RETENTION\_FLAG}$:

$$\begin{cases} \text{IF } \mathsf{TOT}^{\mu}_i \neq 0 \text{ THEN } \begin{cases} \mathsf{SLO}_i = \mathsf{SLO}_{i-1} + \mathsf{IS\_}\mu_{i-1} \\ \mathsf{SBO}_i = \mathsf{SBO}_{i-1} \\ \mathsf{ERF}_i = 1 \\ \mathsf{X\_TXCD}_i = 1 \end{cases} \\ \text{IF } \mathsf{TOT}^{\mu}_i = 0 \text{ THEN } \begin{cases} \mathsf{SLO}_i = 0 \\ \mathsf{SBO}_i = \mathsf{SBO}_{i-1} \\ \mathsf{ERF}_i = 0 \\ \mathsf{X\_TXCD}_i = 0 \\ \mathsf{SIZE}_i = 1 + \mathsf{NIB\_2}_i \end{cases} \end{cases}$$

   Note that updates to the source offset are simple initially: it increase linearly. This trend ends with the final micro-instruction which resets it to 0 includes a final update to the source limb offset

Now that parameters are set we can move on to writing the final micro-instruction. At this point there is no differnence between the two cases $\mathsf{CSD}_i = 1$ and $\mathsf{CSD}_i > 1$. The only question that matters is: are offsets aligned or not?

1. IF $\mathsf{TOT}^{\mu}_i = 0$ THEN

(a) IF $\mathsf{ALIGNED}_i = 1$ THEN

   i. IF $\left(\llbracket 2 \rrbracket_i = 0 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 0\right)$ THEN

$$\begin{cases} \mu\mathsf{INST}_i &= \texttt{FirstPaddedSecondZero} \\ \mathsf{SIZE}_i &= 1 + \mathsf{NIB\_2}_i \end{cases}$$

   ii. IF $\left(\llbracket 2 \rrbracket_i = 0 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 1\right)$ THEN $\mu\mathsf{INST}_i = \texttt{PushOneRamToStack;}$

   iii. IF $\left(\llbracket 2 \rrbracket_i = 1 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 0\right)$ THEN

$$\begin{cases} \mu\mathsf{INST}_i &= \texttt{FirstFastSecondPadded} \\ \mathsf{SIZE}_i &= 1 + \mathsf{NIB\_2}_i \end{cases}$$

   iv. IF $\left(\llbracket 2 \rrbracket_i = 1 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 1\right)$ THEN $\mu\mathsf{INST}_i = \texttt{PushTwoRamToStack;}$

(b) IF $\mathsf{ALIGNED}_i = 0$ THEN

   i. IF $\left(\llbracket 2 \rrbracket_i = 1 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 1\right)$ THEN $\mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_3To2Full}$

   ii. IF $\left(\llbracket 2 \rrbracket_i = 1 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 0\right)$ THEN

$$\begin{cases} \text{IF } \llbracket 4 \rrbracket_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_3To2Padded} \\ \text{IF } \llbracket 4 \rrbracket_i = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_2To2Padded} \end{cases}$$

   In both cases, $\mathsf{SIZE}_i = 1 + \mathsf{NIB\_2}_i$.

   iii. IF $\left(\llbracket 2 \rrbracket_i = 0 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 1\right)$ THEN $\mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_2To1FullAndZero}$

   iv. IF $\left(\llbracket 2 \rrbracket_i = 0 \;\text{ AND }\; \llbracket 3 \rrbracket_i = 0\right)$ THEN

$$\begin{cases} \text{IF } \llbracket 4 \rrbracket_i = 1 \text{ THEN } \mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_2To1PaddedAndZero} \\ \text{IF } \llbracket 4 \rrbracket_i = 0 \text{ THEN } \mu\mathsf{INST}_i = \texttt{NA\_RamToStack\_1To1PaddedAndZero} \end{cases}$$

   In both cases, $\mathsf{SIZE}_i = 1 + \mathsf{NIB\_2}_i$.

Chapter 3

MMIO

3.1 Outline of the RAM arithmetization

3.1.1 RAM instructions

The **mmu module** deals with the following instructions:

1. `SHA3`
2. `MLOAD`
3. `MSTORE`
4. `MSTORE8`
5. `CALLDATALOAD`
6. `CODECOPY`
7. `EXTCODECOPY`
8. `RETURNDATACOPY`
9. `LOG0`
10. `LOG1`
11. `LOG2`
12. `LOG3`
13. `LOG4`
14. `CREATE`
15. `CALL`
16. `CALLCODE`
17. `RETURN`
18. `DELEGATECALL`
19. `CREATE2`
20. `STATICCALL`
21. `REVERT`

3.1.2 Column descriptions

Throughout this document we use the word **limb** to designate a **16**-byte integer.

The RAM data processor has, at all times, access to precisely 3 values (limbs) from RAM. These values can be chosen from distinct execution contexts, including the 0^{th} execution context which plays a special role. To specify a "value in RAM" we thus require a tuples consisting of (a) an execution context (b) a limb offset in RAM (c) the limb (i.e. value) stored at that offset. The arithmetization requires us to add to these (d) a *potentially* udpated value of that limb and (e) bytes that *potentially* spell out the byte decomposition of the limb currently in RAM (i.e. before any *potential* update). This is the purpose of the following columns. Since the RAM data processor can access three RAM slots there are three such quintuples. We give more details below.

Three *counter-constant columns* containing execution context numbers:

1. CONTEXT_A; abbreviated to CN_A;
2. CONTEXT_B; abbreviated to CN_B;
3. CONTEXT_C; abbreviated to CN_C;

Three *counter-constant columns* containing limb offsets within the corresponding execution context's RAM:

4. INDEX_A: limb offset in the RAM of context CN_A;
5. INDEX_B: limb offset in the RAM of context CN_B;
6. INDEX_C: limb offset in the RAM of context CN_C;

Three *counter-constant columns* containing the limbs currently stored at the given offsets inside the corresponding execution context's RAM:

7. VALUE_A: (limb) value currently in CN_A's RAM at INDEX_A; abbreviated to VAL_A;
8. VALUE_B: (limb) value currently in CN_B's RAM at INDEX_B; abbreviated to VAL_B;
9. VALUE_C: (limb) value currently in CN_C's RAM at INDEX_C; abbreviated to VAL_C;

Three *counter-constant columns* containing *potentially* updated values of the limbs currently stored at the given offsets inside the corresponding execution context's RAM:

10. VALUE_A_NEW; updated value in CN_A's RAM at INDEX_A; abbreviated to $\mathsf{VAL\_A}^{\nu}$;
11. VALUE_B_NEW; updated value in CN_B's RAM at INDEX_B; abbreviated to $\mathsf{VAL\_B}^{\nu}$;
12. VALUE_C_NEW; updated value in CN_C's RAM at INDEX_C; abbreviated to $\mathsf{VAL\_C}^{\nu}$;

Three *byte columns* which *may* contain the byte decompositions of VAL_A, VAL_B and/or VAL_C (depending on whether they are required for the present computation):

13. BYTE_A; byte columns;
14. BYTE_B; byte columns;
15. BYTE_C; byte columns;

We also require three *accumulator columns* which *may* witness these byte decompositions:

16. ACC_A: if $\langle \mathsf{FAST} \rangle = 0$ accumulates the bytes of the BYTE_A column;
17. ACC_B: if $\langle \mathsf{FAST} \rangle = 0$ accumulates the bytes of the BYTE_B column;
18. ACC_C: if $\langle \mathsf{FAST} \rangle = 0$ accumulates the bytes of the BYTE_C column;

The following are columns **imported** from the RAM preprocessor. Colums that are imported from the RAM preprocessor are distinguished by angular brackets as in $\langle \mathsf{X} \rangle$. *All imported colummns are counter-constant.*

19. $\langle \mathsf{MICRO\_RAM\_STAMP} \rangle$: contains the RAM micro instruction stamp; abbreviated to $\langle \mu\mathsf{RST} \rangle$;
20. $\langle {}^{\diamond}\mathsf{MICRO\_INSTRUCTION} \rangle$: contains the RAM micro instruction of the current $\langle \mu\mathsf{RST} \rangle$; abbreviated to $\langle \mu\mathsf{INST} \rangle$;
21. $\langle \mathsf{CONTEXT\_SOURCE} \rangle$: context number of the context whose RAM *may* be used as a source of limbs; abbreviated to $\langle \mathsf{CN\_S} \rangle$;
22. $\langle \mathsf{CONTEXT\_TARGET} \rangle$: context number of the context whose RAM *may* be used as a target of limbs; abbreviated to $\langle \mathsf{CN\_T} \rangle$;
23. $\langle \mathsf{IS\_INIT} \rangle$: binary flag that is smart-contract-number constant ; used to recognize the `RETURN` instructions whose return data is deployed bytecode; easily set when doing a `CREATE(2)` instruction; **for contract deployment the RAM can't detect it, it's the ROM that knows, the stack takes its instructions from the ROM and so the stack can know, too, and from the stack the RAM can know, too.**

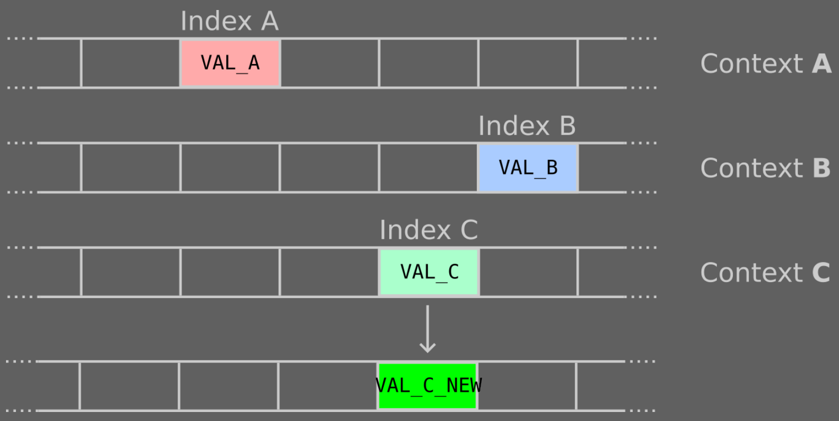

Figure 3.1: The diagram above contains all the intuition there is to convey about context numbers, indices, values and updated values. Every execution context (identified by its context number) has its own RAM, the data in RAM is addressed via an index $\in \{0, 1, \dots\}$ which for the purposes of the zk-evm always is a 4-byte integer as larger offsets are rejected before getting this far. The data itself is packaged as "limbs": 16-byte integers. Instructions may change 0, 1, 2 or even three of the available RAM limbs at any point in time. In the above only the VALUE_C is modified.

24. ⟨SOURCE_LIMB_OFFSET⟩: this imported column contains the limb offset of the first limb to read from / write to in ⟨CN_S⟩'s RAM; abbreviated to ⟨SLO⟩

25. ⟨TARGET_LIMB_OFFSET⟩: this imported column contains the limb offset of the first limb to read from / write to in ⟨CN_T⟩'s RAM; abbreviated to ⟨TLO⟩

26. ⟨SOURCE_BYTE_OFFSET⟩: this imported column contains the byte offset within the limb to read from / write to in ⟨CN_S⟩'s RAM; with values in $\{0, 1, \dots, 15\}$; abbreviated to ⟨SBO⟩;

27. ⟨TARGET_BYTE_OFFSET⟩: this imported column contains the byte offset within the limb to read from / write to in ⟨CN_T⟩'s RAM; with values in $\{0, 1, \dots, 15\}$; abbreviated to ⟨TBO⟩;

28. ⟨SIZE⟩: an imported column containing a "size" parameter used by certain limb surgeries;

29. ⟨FAST⟩: binary flag indicating whether a micro instruction is fast (i.e. occupies a single line in the RAM data processor) or slow (i.e. occupies 16 consecutive lines in the RAM data processor.)

30. ⟨EXCEPTIONAL_RETENTION_FLAG⟩: a binary flag that signals exceptional behaviour of the 0^{th} execution context's RAM; abbreviated to ⟨ERF⟩.

The 0^{th} execution context is a ficticious execution context and its RAM is subject to no internal consistency constraints. Raising the ⟨EXCEPTIONAL_RETENTION_FLAG⟩ changes this temporarily and allows the arithmetization to use the 0^{th} execution context's RAM as temporary storage.

31. ⟨STACK_VALUE_HIGH⟩: abbreviated to ⟨VAL^{hi}⟩;

32. ⟨STACK_VALUE_LOW⟩: abbreviated to ⟨VAL^{lo}⟩;

33. $\text{BYTE\_V}^{\text{hi}}$; abbreviated to ;

34. $\text{BYTE\_V}^{\text{lo}}$; abbreviated to ;

35. ACC_V^{hi}: if ⟨FAST⟩ $= 0$ accumulates the bytes of the $\text{BYTE\_V}^{\text{hi}}$ column;

36. ACC_V^{lo}: if ⟨FAST⟩ $= 0$ accumulates the bytes of the $\text{BYTE\_V}^{\text{lo}}$ column;

Given the stack pattern of instructions triggering the present module and using stack inputs / outputs (i.e. MLOAD, MSTORE, MSTORE8 and CALLDATALOAD) $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$ are imports of ${}_4\mathsf{VAL}^{\mathsf{hi}}$ and ${}_4\mathsf{VAL}^{\mathsf{lo}}$ respectively.

The RAM interacts with other data sources: the stack, logs, the ROM, transaction call data and the data to hash. The RAM module has, accordingly, access to values coming from the stack but also from **exogenous data sources** i.e. logs, ROM, transaction call data. The following two columns are counter-constant imported columns containing stack values: These columns only play a role for MSTORE, MSTORE8, MLOAD and CALLDATALOAD: for MSTORE and MSTORE8, $\mathsf{VAL\_S}^{\mathsf{hi}}$ and $\mathsf{VAL\_S}^{\mathsf{lo}}$ contain the high and low part of the argument from stack to be stored in RAM; for MLOAD and CALLDATALOAD, $\mathsf{VAL\_S}^{\mathsf{hi}}$ and $\mathsf{VAL\_S}^{\mathsf{lo}}$ contain the high and low part of the value retrieved from RAM or the call data respetively; offsets (i.e. where to store or from where to retrieve) are handled elsewhere. We come to **exogenous data columns**. These columns contain data from exogenous sources which we define as being either

- the stack,
- the rom,
- log data,
- transaction input data.

Note that transaction input data can (and does) appear both in the first batch of columns and exogenous data columns. This data comes in

37. $\langle \mathsf{EXO\_IS\_ROM} \rangle$: imported binary flag column that lights up whenever the micro instruction requires exogenous data from **ROM**; abbreviated to $\langle \mathsf{X\_ROM} \rangle$;
38. $\langle \mathsf{EXO\_IS\_LOG} \rangle$: imported binary flag column that lights up for all micro instructions unfolding a LOG0-LOG4 macro-instruction; abbreviated to $\langle \mathsf{X\_LOG} \rangle$;
39. $\langle \mathsf{EXO\_IS\_SHA3} \rangle$: imported binary flag column that lights up for all micro instructions unfolding a SHA3 instruction; abbreviated to $\langle \mathsf{X\_SHA3} \rangle$;
40. $\langle \mathsf{EXO\_IS\_TXCD} \rangle$: imported binary flag column that lights up whenever the micro instruction requires exogenous data from **transaction call data**; abbreviated to $\langle \mathsf{X\_TXCD} \rangle$;
41. INDEX_X: contains the limb offset of exogenous data;
42. $\langle \mathsf{VAL\_X} \rangle$: limb column; contains exogenous data;
43. BYTE_X: byte column; *may* contain the byte decomposition of $\langle \mathsf{VAL\_X} \rangle$;
44. ACC_X: if $\langle \mathsf{FAST} \rangle = 0$ accumulates the bytes of the BYTE_X column;

We introduce some book-keeping columns for memory operations involving call data and return data:

45. TRANSACTION_NUMBER: transaction number; imported form the main execution trace; abbreviated to TXNUM;

We now introduce some columns that are of use in producing proofs but aren't meaningful outside of that.

46. "binary plateau" columns $[\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!], [\![5]\!]$;
47. "accumulator" columns ACC_1, ACC_2, ACC_3, ACC_4, ACC_5, ACC_6;
48. "powers of 256" columns POW_256_1 and POW_256_2;
49. COUNTER: a column that either hovers around 0 or counts up from 0 to 15 and resets to 0; used for slow memory operations (i.e. when $\langle \mathsf{FAST} \rangle = 0$) when byte decompositions are needed;

The section on consistency constraints introduces further columns required for checking "execution context metatdata consistency" as well as for "memory consistency".

The RAM pre-processor converts RAM instructions into a sequence of **RAM micro instructions**. The RAM data processor only knows how to deal with micro instructions. Micro instructions can be fast ($\langle \mathsf{FAST} \rangle = 1$) or slow ($\langle \mathsf{FAST} \rangle = 0$). Fast micro instructions take up exactly one row in the RAM data processor's execution trace. Slow micro instructions take up exactly sixteen rows in the RAM data processor's execution trace. RAM micro instructions can modify 1, 2 or even 3 limbs at once through various forms of **limb surgery**. These limbs may be in RAM, imported from the stack or part of exogenous data. The modification can use as inputs 1, 2 or even 3 limbs taken from RAM, the stack or exogenous data. Limb surgeries (micro instructions that modify limbs on a byte level) are determined by their *(data) source*, *(data) target*, a *surgery pattern* and an *offset* and potentially a *size*. The micro instruction, the source and target, the offsets and the size (if any) are handed down to the RAM data processor from the pre-processor. Offsets are actually given in terms of a *limb offset* and a *byte offset* determined by euclidean division of the underlying offset by 16:

$$\mathsf{offset} = 16 \cdot \mathsf{limbOffset} + \mathsf{byteOffset}, \quad \mathsf{byteOffset} \in \{0, 1, \ldots, \mathbf{15}\}.$$

The purposed of the following columns is to transmit these data.

50. $\langle \mathsf{SOURCE\_LIMB\_OFFSET} \rangle$: limb offset $\in \mathbb{N}$ of the first (and potentially only) limb used as a byte source to modify one or more target limbs; abbreviated to $\langle \mathsf{SLO} \rangle$;

51. $\langle \mathsf{SOURCE\_BYTE\_OFFSET} \rangle$: byte offset $\in \{0, 1, \ldots, \mathbf{15}\}$ of the first byte in the first (and potentially only) source limb used to modify one or more target limbs; abbreviated to $\langle \mathsf{SBO} \rangle$;

52. $\langle \mathsf{TARGET\_LIMB\_OFFSET} \rangle$: limb offset $\in \mathbb{N}$ of the first (and potentially only) target limb to be modified; abbreviated to $\langle \mathsf{TLO} \rangle$;

53. $\langle \mathsf{TARGET\_BYTE\_OFFSET} \rangle$: byte offset $\in \{0, 1, \ldots, \mathbf{15}\}$ of the first byte in the first (and potentially only) target limb to be modified; abbreviated to $\langle \mathsf{TBO} \rangle$.

3.2 Specialized constraints

3.2.1 Binary constraints

Recall that a column X is **binary** if it satisfies: $\mathsf{X} \cdot (1 - \mathsf{X}) = 0$. The following columns are binary: $[\![1]\!]$, $[\![2]\!]$, $[\![3]\!]$, $[\![4]\!]$ and $[\![5]\!]$

3.2.2 Binary plateau constraints

Suppose X, C are columns such that

- X is binary,
- C is counter-constant.

We say that the pair (X, C) satisfies the **binary plateau constraints** if

```
1. IF $\mathsf{C}_i = 0$ THEN $\mathsf{X}_i = 1$,
2. IF $\mathsf{C}_i \neq 0$ THEN
    (a) IF $\mathsf{CT}_i = 0$ THEN $\mathsf{X}_i = 0$,
    (b) IF $\mathsf{CT}_i \neq 0$ THEN
        i. IF $\mathsf{CT}_i = \mathsf{C}_i$ THEN $\mathsf{X}_i = \mathsf{X}_{i-1} + 1$,
```

X	**1**	**1**	**1**	$\cdots$	**1**
CT	0	1	2	$\cdots$	15

X	0	0	0	$\cdots$	0	**1**	**1**	$\cdots$	**1**
CT	0	1	2	$\cdots$	$c-1$	c	$c+1$	$\cdots$	15

X	0	0	0	$\cdots$	0
CT	0	1	2	$\cdots$	15

Figure 3.2: Assuming $\texttt{Plateau}(\mathsf{X}, \mathsf{C})$, the above represents a counter-cycle's worth of X when $c = 0$, when $0 < c < 16$ and when $c \geq 16$.

ii. IF $\mathsf{CT}_i \neq \mathsf{C}_i$ THEN $\mathsf{X}_i = \mathsf{X}_{i-1}$,

and we use the shorthand

$$\texttt{Plateau}(\mathsf{X}, \mathsf{C})$$

to signify that (X, C) satisfies this constraint. In practice the columns C we will consider will be locally constant columns with values $\in \{0, 1, 2, \dots, 15\}$. Figure ?? represents portions (that is, counter-cycles) of a binary column X that satisfies a binary plateau constraint w.r.t. some counter-constant column C for different values of $c = \mathsf{C}$.

3.2.3 Power constraints

Let P and X be two columns with

- X binary column.

We say that the pair (P, X) satisfies a **power-constraint** if it satisfies the following constraints:

1. IF $\langle \mu\mathsf{RST} \rangle_i = 0$ THEN $\mathsf{P} = 0$
2. IF $\langle \mu\mathsf{RST} \rangle_i \neq 0$ THEN
   (a) IF $\langle \mathsf{FAST} \rangle_i = 1$ THEN $\mathsf{P}_i = 0$
   (b) IF $\langle \mathsf{FAST} \rangle_i = 0$ THEN
      i. IF $\mathsf{CT}_i = 0$ THEN $\mathsf{P}_i = 1$
      ii. IF $\mathsf{CT}_i \neq 0$ THEN
         A. IF $\mathsf{X}_i = 0$ THEN $\mathsf{P}_i = \mathsf{P}_{i-1}$
         B. IF $\mathsf{X}_i = 1$ THEN $\mathsf{P}_i = 256 \cdot \mathsf{P}_{i-1}$

Power constraints will be applied in the case where X satisfies a plateau constraint so that P is initially constant $= 1$ and then grows geometrically until the end of the current counter-cycle:

CT	0	1	2	$\cdots$	$c-1$	c	$c+1$	$\cdots$	15
X	0	0	0	$\cdots$	0	**1**	**1**	$\cdots$	**1**
P	1	1	1	$\cdots$	1	256	256^2	$\cdots$	256^d

Figure 3.3: In the picture above X satisfies the plateau constraint $\texttt{Plateau}(\mathsf{X}, c)$, $0 < c < 16$, and P satisfies a power constraint $\texttt{Power}(\mathsf{P}, \mathsf{X})$. We have set $d = 16 - c$.

Its value at the end of the counter-cycle is in the set $\{256^i \mid i = 0, 1, \dots, 15\}$. We use the short hand

$$\texttt{Power}(\mathsf{P}, \mathsf{X})$$

to signify that the columns P and X satisfy a power-constraint and we may say that **P is pegged to X**.

3.2.4 Byte decomposition constraints

Suppose we have

1. a counter-constant column S,
2. and a byte column SB,
3. a third column ACC.

We say that SB computes the byte decomposition of S through ACC if

1. $\textsc{if}$ $\mathsf{CT}_i = 0$ $\textsc{then}$ $\mathsf{ACC}_i = \mathsf{SB}_i$,
2. $\textsc{if}$ $\mathsf{CT}_i \neq 0$ $\textsc{then}$ $\mathsf{ACC}_i = 256 \cdot \mathsf{ACC}_{i-1} + \mathsf{SB}_i$,
3. $\textsc{if}$ $\mathsf{CT}_i = \mathbf{15}$ $\textsc{then}$ $\mathsf{ACC}_i = \mathsf{S}_i$.

We encapsulate these constraints in the following relation

$$\texttt{ByteDec}(\mathsf{S}; \mathsf{SB}; \mathsf{ACC})$$

(Note: CT is implicit)

3.2.5 Suffix extraction

Suppose ACC, B, X are columns with

- B a byte column,
- X a binary column.

In all applications ACC will be an accumulator column, B arises as the byte decomposition of a counter-constant column S and X satisfies a plateau constraint. We abbreviate under $\texttt{IsolateSuffix}(\mathsf{ACC}, \mathsf{B}, \mathsf{X})$ the following set of constraints

1. $\textsc{if}$ $\mathsf{CT}_i = 0$ $\textsc{then}$

$$\begin{cases} \textsc{if}\ \mathsf{X}_i = 0\ \textsc{then}\ \mathsf{ACC}_i = 0 \\ \textsc{if}\ \mathsf{X}_i = 1\ \textsc{then}\ \mathsf{ACC}_i = \mathsf{B}_i \end{cases}$$

2. $\textsc{if}$ $\mathsf{CT}_i \neq 0$ $\textsc{then}$

$$\begin{cases} \textsc{if}\ \mathsf{X}_i = 0\ \textsc{then}\ \mathsf{ACC}_i = \mathsf{ACC}_{i-1} \\ \textsc{if}\ \mathsf{X}_i = 1\ \textsc{then}\ \mathsf{ACC}_i = 256 \cdot \mathsf{ACC}_{i-1} + \mathsf{B}_i \end{cases}$$

3.2.6 Prefix extraction

Suppose ACC, B, X are columns with

- B a byte column,
- X a binary column.

In all applications ACC will be an accumulator column, B arises as the byte decomposition of a counter-constant column S and X will satisfy a plateau constraint. We abbreviate under $\texttt{IsolatePrefix}(\mathsf{ACC}, \mathsf{B}, \mathsf{X})$ the following set of constraints

1. $\textsc{if}$ $\mathsf{CT}_i = 0$ $\textsc{then}$

$$\begin{cases} \textsc{if}\ \mathsf{X}_i = 0\ \textsc{then}\ \mathsf{ACC}_i = \mathsf{B}_i \\ \textsc{if}\ \mathsf{X}_i = 1\ \textsc{then}\ \mathsf{ACC}_i = 0 \end{cases}$$

2. $\textsc{if}$ $\mathsf{CT}_i \neq 0$ $\textsc{then}$

$$\begin{cases} \textsc{if}\ \mathsf{X}_i = 0\ \textsc{then}\ \mathsf{ACC}_i = 256 \cdot \mathsf{ACC}_{i-1} + \mathsf{B}_i \\ \textsc{if}\ \mathsf{X}_i = 1\ \textsc{then}\ \mathsf{ACC}_i = \mathsf{ACC}_{i-1} \end{cases}$$

3.2.7 Chunk extraction

Suppose ACC, B, X, Y are columns with

- B a byte column,
- X and Y binary columns.

In applications (and whenever this constraint is activated) X will satisfy a plateau constraint which jumps at C, Y will satisfy a plateau constraint which jumps at D for nonnegative integers[1] $0 \leq \mathsf{C} < \mathsf{D}$. Furthermore, B will contain the bytes of a (counter-constant) column S. The goal is for ACC to accumulate the bytes B_i of S for $\mathsf{C} \leq i < \mathsf{D}$. We abbreviate under $\mathtt{IsolateChunk}(\mathsf{ACC}, \mathsf{B}, \mathsf{X}, \mathsf{Y})$ the following set of constraints

1. IF $\mathsf{CT}_i = 0$ THEN
   (a) IF $\mathsf{X}_i = 0$ THEN $\mathsf{ACC}_i = 0$
   (b) IF $\mathsf{X}_i = 1$ THEN $\mathsf{ACC}_i = \mathsf{B}_i$
2. IF $\mathsf{CT}_i \neq 0$ THEN
   (a) IF $\mathsf{X}_i = 0$ THEN $\mathsf{ACC}_i = 0$
   (b) IF $\mathsf{X}_i = 1$ THEN
      i. IF $\mathsf{Y}_i = 0$ THEN $\mathsf{ACC}_i = 256 \cdot \mathsf{ACC}_{i-1} + \mathsf{B}_i$
      ii. IF $\mathsf{Y}_i = 1$ THEN $\mathsf{ACC}_i = \mathsf{ACC}_{i-1}$

3.3 Module consraints

3.3.1 Heartbeat

The columns $\langle\mu\mathsf{RST}\rangle$, $\langle\mathsf{FAST}\rangle$ and CT impose a heartbeat on the RAM module. We ask that they satisfy the following constraints:

1. $\langle\mathsf{FAST}\rangle$ is a binary column;
2. $\langle\mu\mathsf{RST}\rangle_0 = 0$;
3. $\langle\mu\mathsf{RST}\rangle_{i+1} \in \{\langle\mu\mathsf{RST}\rangle_i, 1 + \langle\mu\mathsf{RST}\rangle_i\}$[2];
4. IF $\langle\mu\mathsf{RST}\rangle_i = 0$ THEN $\langle\mathsf{FAST}\rangle_i = 0$ and $\mathsf{CT}_i = 0$;
5. IF $\langle\mu\mathsf{RST}\rangle_i \neq 0$ THEN
   (a) IF $\langle\mathsf{FAST}\rangle_i = 1$ THEN
   $$\begin{cases} \mathsf{CT}_{i+1} = \mathsf{CT}_i = 0 \\ \langle\mu\mathsf{RST}\rangle_{i+1} = 1 + \langle\mu\mathsf{RST}\rangle_i \end{cases}$$
   (b) IF $\langle\mathsf{FAST}\rangle_i = 0$ THEN
      i. IF $\mathsf{CT}_i \neq \mathbf{15}$ THEN
      $$\begin{cases} \langle\mathsf{FAST}\rangle_{i+1} = \langle\mathsf{FAST}\rangle_i \\ \langle\mu\mathsf{RST}\rangle_{i+1} = \langle\mu\mathsf{RST}\rangle_i \\ \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \end{cases}$$

[1] nibbles, actually

[2] i.e. $(\langle\mu\mathsf{RST}\rangle_{i+1} - \langle\mu\mathsf{RST}\rangle_i) \cdot (\langle\mu\mathsf{RST}\rangle_{i+1} - \langle\mu\mathsf{RST}\rangle_i - 1) = 0$

ii. IF $\mathsf{CT}_i = \mathbf{15}$ THEN

$$\begin{cases} \mathsf{CT}_{i+1} = 0 \\ \langle \mu\mathsf{RST} \rangle_{i+1} = 1 + \langle \mu\mathsf{RST} \rangle_i \end{cases}$$

6. IF $\langle \mu\mathsf{RST} \rangle_N \neq 0$ THEN
   (a) IF $\langle \mathsf{FAST} \rangle_N = 1$ no finalization contraint required;
   (b) IF $\langle \mathsf{FAST} \rangle_N = 0$ THEN $\mathsf{CT}_N = \mathbf{15}$

3.3.2 Byte decomposition constraints

We enforce the following byte decomposition constraints:

1. IF

$$\Big(\langle \mu\mathsf{RST} \rangle_i \neq 0 \quad \text{AND} \ \langle \mathsf{FAST} \rangle_i = 0 \Big)$$

THEN

$$\begin{cases} \mathtt{ByteDec}(\mathsf{VAL\_A}, \mathsf{BYTE\_A}, \mathsf{ACC\_A}), \\ \mathtt{ByteDec}(\mathsf{VAL\_B}, \mathsf{BYTE\_B}, \mathsf{ACC\_B}), \\ \mathtt{ByteDec}(\mathsf{VAL\_C}, \mathsf{BYTE\_C}, \mathsf{ACC\_C}), \\ \mathtt{ByteDec}(\langle \mathsf{VAL}^{\mathsf{hi}} \rangle, \mathsf{BYTE\_V}^{\mathsf{hi}}, \mathsf{ACC\_V}^{\mathsf{hi}}), \\ \mathtt{ByteDec}(\langle \mathsf{VAL}^{\mathsf{lo}} \rangle, \mathsf{BYTE\_V}^{\mathsf{lo}}, \mathsf{ACC\_V}^{\mathsf{lo}}) \\ \mathtt{ByteDec}(\langle \mathsf{VAL\_X} \rangle, \mathsf{BYTE\_X}, \mathsf{ACC\_X}) \end{cases}$$

Note that only some of these byte decompositions matter at any one point in time.

3.3.3 Bytehood constraints

The following columns must contain bytes:

1. $\mathsf{BYTE\_A}$,
2. $\mathsf{BYTE\_B}$,
3. $\mathsf{BYTE\_C}$,
4. $\mathsf{BYTE\_V}^{\mathsf{hi}}$,
5. $\mathsf{BYTE\_V}^{\mathsf{lo}}$,
6. $\mathsf{BYTE\_X}$,

We thus impose a bytehood constraint on

$$\mathsf{BYTE\_A} \boxplus \mathsf{BYTE\_B} \boxplus \mathsf{BYTE\_C} \boxplus \mathsf{BYTE\_V}^{\mathsf{hi}} \boxplus \mathsf{BYTE\_V}^{\mathsf{lo}} \boxplus \mathsf{BYTE\_X}$$

3.3.4 Counter constancy

We say that a column X is **counter-constant** if it satisfies

$$\begin{cases} \text{IF } \langle \mu\mathsf{RST} \rangle_i = 0 \text{ THEN } \mathsf{X}_i = 0 \\ \text{IF } \mathsf{CT}_{i+1} \neq 0 \text{ THEN } \mathsf{X}_{i+1} = \mathsf{X}_i \end{cases}$$

The table below depicts the behaviour of a typical counter-constant column X:

$\langle\mu\mathsf{RST}\rangle$	$\langle\mathsf{FAST}\rangle$	COUNTER	X
0	0	0	0
0	0	0	0
0	0	0	0
1	**1**	0	*a*
2	**1**	0	*b*
3	**0**	0	**c**
3	**0**	1	**c**
3	**0**	2	**c**
⋮	⋮	⋮	⋮
3	**0**	14	**c**
3	**0**	**15**	**c**
4	**1**	0	*d*
5	**1**	0	*e*
6	**0**	0	**f**
6	**0**	0	**f**
6	**0**	1	**f**
6	**0**	2	**f**
⋮	⋮	⋮	⋮
6	**0**	14	**f**
6	**0**	**15**	**f**
7	**0**	0	**g**
7	**0**	1	**g**
7	**0**	2	**g**
⋮	⋮	⋮	⋮

We ask that all imported columns be counter constant. Note that $\langle\mu\mathsf{RST}\rangle$ and $\langle\mathsf{FAST}\rangle$, which are imported, are counter-constant by the set of constraints from section 8.2.1. Note that we can have an arbitrary number of rows of all zero (imported) columns at the start of the execution trace.

3.4 Limb transplants

3.4.1 Purpose

Several of the micro instructions that the RAM data processor may be led to execute can be done in one fell swoop i.e. don't involve byte decompositions, cutting, grafting nor zero padding. They simply move one (or more) limb(s) from one place to another. These operations are collectively dubbed **transplants**. Transplants don't present any difficulty in terms of their arithmetization. The complexity lies in solely determining:

1. Which data source is the donor, which is the recipient?
2. Is exognenous data involved i.e. data from ROM, transaction call data or logs?
3. Are the stack values involved?
4. Does RAM undergo an update or remain identical to itself?

Resolving these points leads to greater conceptual clarity. It also leads to having many kinds of micro instructions that look very similar on the surface but differ in subtle ways and are use by different opcodes. The second point presents a conceptual challenge: some operations naturally expect 2 or 3 inputs and produce 2 or 3 outputs. However, exogenous data is only available *one limb at a time*. The third points presents a similar, though greater, challenge. We made the decision to have it so that stack values are read from and constructed *as pairs* rather than one by one[3]. This is straightforward to implement for `MLOAD`, `MSTORE`, `MSTORE8` and `CALLDATALOAD` *performed in a subcontex of the current root context*. But `CALLDATALOAD` performed in the root context of a transaction poses a proper challenge. Indeed, transaction call data, like any exogenous data, is only available one limb at a time. Yet `CALLDATALOAD`, which, in accordance with the previously stated design principle, wants to produce $\langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle$ in one go, may require up to 3 limbs from transaction call data. Note as well that this is the *only* opcode the RAM deals with that (in theory) *doesn't involve the RAM at all*: it's a direct transfer (with some potential cutting, grafting and zero padding) from transaction call data to the stack.

One may reasonably inquire at this stage how the complications arising from limited donor limb availability and `CALLDATALOAD` are related to transplants. The answer is that we introduce some transplant operations to *prepare the terrain* for proper surgeries to come later. Dealing with `CALLDATALOAD` (at the root level of a transaction) forced us to introduce an $\langle \mathsf{EXCEPTIONAL\_RETENTION\_FLAG} \rangle$ which signals exceptional behaviour of the RAM associated with the 0^{th} execution context. As can be read off the memory consistency constraints, the memory of the 0^{th} execution context is subject to *no internal consistencies*. The $\langle \mathsf{ERF} \rangle$ changes this temporarily (i.e. for up to 4 consecutive micro instructions) and allows us to use this "RAM" as a data buffer. This data buffer is then filled with limbs harvested one by one from transaction call data in up to 3 transplant operations.

A general design principle we have adopted is that operations that "write to" exogenous data (to be precise: these operations *produce* values that are then *compared* to exogenous data[4]) should happen at once (i.e. we don't produce parts of the data in steps, we produce the requisite data in one counter-cycle).

To help with readability we sometimes insert a (★) near the constraints that "do the work".

3.4.2 RAM to RAM

The following constraints pertain to aligned (i.e. the "real" or "adjusted" source and target offsets are $\equiv 0\,[16]$) limb transplants between the memories of two execution contexts. The RAM has at all times access to precisely three limbs from the RAMs of three (potentially distinct) execution contexts. Aligned transfers can thus only work one limb at a time. Only one kind of such operation is needed, which we label `RamToRam` and arithmetize like so:

$$\texttt{RamToRam} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_A}_i & (\bigstar) \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

[3] Recall that stack values are EVM words and are thus comprised of a high and a low part, $\langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle$ respectively.

[4] e.g. logs are *produced* from RAM and then *compared* to a public commitment of logs, successfully deployed bytecodes are *produced* from RAM and *compared* to existing bytecodes in ROM

3.4.3 Exodata to RAM

The following constraints may appear in aligned `(EXT)CODECOPY`s and `CALLDATACOPY`s (at the root execution context of a transaction).

$$\texttt{ExoToRam} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{VAL\_A}_i^\nu = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3.4.4 Exodata and RAM agree

The following set of constraints appears in

1. aligned `LOG0`-`LOG4` (i.e. when the offset is $\equiv 0$ [16]),
2. aligned `RETURN`s in deployment contexts ($\mathsf{CTYPE} = 1$ and deployment succedes).

We dub it `RamIsExo`:

$$\texttt{RamIsExo} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL\_X} \rangle_i = \mathsf{VAL\_A}_i & (\bigstar) \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3.4.5 Killing RAM slots

Executing some opcodes may require us to replace entire limbs with 0. This is true of

- out of bounds `CODECOPY`s,
- out of bounds `EXTCODECOPY`s,
- out of bounds `CALLDATACOPY`s.

Since we have three RAM slots at our disposal at any time we can kill up to 3 limbs in RAM per micro instruction. The following named constraints accomplish this:

1. Killing one limb:

$$\texttt{KillingOne} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = 0 & (\bigstar) \\ \mathsf{VAL\_B}_i = \mathsf{VAL\_B}_i^{\nu} = 0 \\ \mathsf{VAL\_C}_i = \mathsf{VAL\_C}_i^{\nu} = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Killing two consecutive limbs:

$$\texttt{KillingTwo} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = 0 & (\bigstar) \\ \mathsf{VAL\_B}_i^{\nu} = 0 & (\bigstar) \\ \mathsf{VAL\_C}_i = \mathsf{VAL\_C}_i^{\nu} = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3. Killing three consecutive limb:

$$\texttt{KillingThree} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = \langle \mathsf{TLO} \rangle_i + 2 \\ \mathsf{VAL\_A}_i^{\nu} = 0 & (\bigstar) \\ \mathsf{VAL\_B}_i^{\nu} = 0 & (\bigstar) \\ \mathsf{VAL\_C}_i^{\nu} = 0 & (\bigstar) \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3.4.6 RAM to stack

We use the moniker `PushTwoRamToStack` to subsume the following set of constraints:

$$\texttt{PushTwoRamToStack} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i = \mathsf{VAL\_A}_i & (\bigstar) \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = \mathsf{VAL\_B}_i & (\bigstar) \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

we also require a version where we push only one limb:

$$\texttt{PushOneRamToStack} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i = \mathsf{VAL\_A}_i & (\bigstar) \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0 & (\bigstar) \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3.4.7 Stack to RAM

We use the moniker `PushTwoStackToRam` to subsume the following set of constraints:

$$\texttt{PushTwoStackToRam} \iff \begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i & (\bigstar) \\ \mathsf{VAL\_B}_i^{\nu} = \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i & (\bigstar) \\ \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

3.4.8 Transaction call data to RAM

The following constraints allow the 0^{th} execution context's RAM (which is memoryless) to function as temporary storage where we may store up to 3 limbs taken from transaction call data. The only scenario where these constraints come into play is when executing `CALLDATALOAD` in a root execution context, i.e. `CALLDATALOAD`ing transaction call data. Note that this is the only scenario where RAM isn't involved *per se*, see table 3.13. The $\langle \mathsf{EXCEPTIONAL\_RETENTION\_FLAG} \rangle$ signals such exceptional behaviour of the 0^{th} execution context's RAM.

The RAM preprocessor will initially assess how many limbs (if any) have to be imported from transaction call data to honour a `CALLDATALOAD` instruction in a root execution context: this may be 0, 1, 2 or 3. None are needed precisely when requested 32 bytes of calldata are out of bounds, in this case no instruction is sent to the RAM data processor and the RAM preprocessor simply checks that both $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$ are both 0. Otherwise one of the following sequences of instructions is sent to the data processor:

- a `StoreXinAoneRequired` micro instruction,
- a `StoreXinAtwoRequired` micro instruction followed by a `StoreXinB` micro instruction,
- a `StoreXinAthreeRequired` micro instruction followed by `StoreXinB` and `StoreXinC` micro instructions,

invariably followed by a (fast or slow) transfer to stack values (i.e. to $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$) of the relevant portion of three limbs currently in the 0^{th} execution context's RAM.

We start by describing the `StoreXinAoneRequired`, `StoreXinAtwoRequired`, `StoreXinAthreeRequired` constraints.

1. `StoreXinAoneRequired`:

$$\begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{VAL\_A}_{i+1} = \mathsf{VAL\_A}_i = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_B}_{i+1} = \mathsf{VAL\_B}_i = 0 & (\bigstar) \\ \mathsf{VAL\_C}_{i+1} = \mathsf{VAL\_C}_i = 0 & (\bigstar) \\ \mathsf{VAL\_A}^\nu_{i+1} = \mathsf{VAL\_A}^\nu_i = 0 \\ \mathsf{VAL\_B}^\nu_{i+1} = \mathsf{VAL\_B}^\nu_i = 0 \\ \mathsf{VAL\_C}^\nu_{i+1} = \mathsf{VAL\_C}^\nu_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 1 & (\bigstar) \end{cases}$$

2. `StoreXinAtwoRequired`:

$$\texttt{StoreXinAtwoRequired} \iff \begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{VAL\_A}_{i+1} = \mathsf{VAL\_A}_i = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_B}_{i+1} = \mathsf{VAL\_B}_i & (\bigstar) \\ \mathsf{VAL\_C}_{i+1} = \mathsf{VAL\_C}_i = 0 & (\bigstar) \\ \mathsf{VAL\_A}^\nu_{i+1} = \mathsf{VAL\_A}^\nu_i = 0 \\ \mathsf{VAL\_B}^\nu_{i+1} = \mathsf{VAL\_B}^\nu_i = 0 \\ \mathsf{VAL\_C}^\nu_{i+1} = \mathsf{VAL\_C}^\nu_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 1 & (\bigstar) \end{cases}$$

3. `StoreXinAthreeRequired`:

$$\texttt{StoreXinAthreeRequired} \iff \begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{VAL\_A}_{i+1} = \mathsf{VAL\_A}_i = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_B}_{i+1} = \mathsf{VAL\_B}_i & (\bigstar) \\ \mathsf{VAL\_C}_{i+1} = \mathsf{VAL\_C}_i & (\bigstar) \\ \mathsf{VAL\_A}^\nu_{i+1} = \mathsf{VAL\_A}^\nu_i = 0 \\ \mathsf{VAL\_B}^\nu_{i+1} = \mathsf{VAL\_B}^\nu_i = 0 \\ \mathsf{VAL\_C}^\nu_{i+1} = \mathsf{VAL\_C}^\nu_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 1 & (\bigstar) \end{cases}$$

Followed by `StoreXinB` and `StoreXinC`

4. `StoreXinB`:

$$\texttt{StoreXinB} \iff \begin{cases} \mathsf{CN\_A}_i = 0 & \\ \mathsf{CN\_B}_i = 0 & \\ \mathsf{CN\_C}_i = 0 & \\ \mathsf{INDEX\_A}_i = 0 & \\ \mathsf{INDEX\_B}_i = 0 & \\ \mathsf{INDEX\_C}_i = 0 & \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i & \\ \mathsf{VAL\_A}_{i+1} = \mathsf{VAL\_A}_i & (\bigstar) \\ \mathsf{VAL\_B}_{i+1} = \mathsf{VAL\_B}_i = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_C}_{i+1} = \mathsf{VAL\_C}_i & (\bigstar) \\ \mathsf{VAL\_A}^{\nu}_{i+1} = \mathsf{VAL\_A}^{\nu}_i = 0 & \\ \mathsf{VAL\_B}^{\nu}_{i+1} = \mathsf{VAL\_B}^{\nu}_i = 0 & \\ \mathsf{VAL\_C}^{\nu}_{i+1} = \mathsf{VAL\_C}^{\nu}_i = 0 & \\ \langle \mathsf{ERF} \rangle_i = 1 & (\bigstar) \end{cases}$$

5. `StoreXinC`:

$$\texttt{StoreXinC} \iff \begin{cases} \mathsf{CN\_A}_i = 0 & \\ \mathsf{CN\_B}_i = 0 & \\ \mathsf{CN\_C}_i = 0 & \\ \mathsf{INDEX\_A}_i = 0 & \\ \mathsf{INDEX\_B}_i = 0 & \\ \mathsf{INDEX\_C}_i = 0 & \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i & \\ \mathsf{VAL\_A}_{i+1} = \mathsf{VAL\_A}_i & (\bigstar) \\ \mathsf{VAL\_B}_{i+1} = \mathsf{VAL\_B}_i & (\bigstar) \\ \mathsf{VAL\_C}_{i+1} = \mathsf{VAL\_C}_i = \langle \mathsf{VAL\_X} \rangle_i & (\bigstar) \\ \mathsf{VAL\_A}^{\nu}_{i+1} = \mathsf{VAL\_A}^{\nu}_i = 0 & \\ \mathsf{VAL\_B}^{\nu}_{i+1} = \mathsf{VAL\_B}^{\nu}_i = 0 & \\ \mathsf{VAL\_C}^{\nu}_{i+1} = \mathsf{VAL\_C}^{\nu}_i = 0 & \\ \langle \mathsf{ERF} \rangle_i = 1 & (\bigstar) \end{cases}$$

Let us explain the highlights $(\bigstar)$ we put throughout. First of all we highlight the times that $\langle \mathsf{EXCEPTIONAL\_RETENTION\_FLAG} \rangle$ is set. Notice that throughout the context whose RAM we manipulate is that of the 0^{th} context. We thus highlight the rows indicating whenever the values in A, B and C are propagated to the next row. Those are the exceptional data retention constraints. They make it so that

- if we execute a `StoreXinAoneRequired` micro instruction, the next micro instruction (which will invariably be writing to the stack) uses as input the three limbs $(\langle \mathsf{VAL\_X} \rangle_i, 0, 0)$;

- a `StoreXinAtwoRequired` micro instruction is invariably followed by a `StoreXinB` micro instruction and the next micro instruction (which will invariably be writing to the stack) uses as input the three limbs $(\langle \mathsf{VAL\_X} \rangle_i, \langle \mathsf{VAL\_X} \rangle_{i+1}, 0)$ where $\langle \mathsf{VAL\_X} \rangle_i$ and $\langle \mathsf{VAL\_X} \rangle_{i+1}$ will be consecutive values from transaction data;

- a `StoreXinAthreeRequired` micro instruction is invariably followed by `StoreXinB` and `StoreXinC` micro instructions and the next micro instruction (which will invariably be writing to the stack) uses as input the three limbs $(\langle \mathsf{VAL\_X} \rangle_i, \langle \mathsf{VAL\_X} \rangle_{i+1}, \langle \mathsf{VAL\_X} \rangle_{i+2})$ where $\langle \mathsf{VAL\_X} \rangle_i$, $\langle \mathsf{VAL\_X} \rangle_{i+1}$ and $\langle \mathsf{VAL\_X} \rangle_{i+2}$ will be consecutive values from transaction data.

The fact that $\langle \mathsf{VAL\_X} \rangle_i$, $\langle \mathsf{VAL\_X} \rangle_{i+1}$ and $\langle \mathsf{VAL\_X} \rangle_{i+2}$ will be consecutive values from transaction call data and that instruction orders are imposed in the manner described above *isn't* imposed in the data processing part of RAM: it will be imposed at the RAM preprocessing level, where the micro instructions are formed. We will thus impose using transaction call data as the exogenous data source and for the two or three consecutive instructions just described use consecutive limb offsets.invariably followed by a (fast or slow) transfer to stack values (i.e. to $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle$) of the relevant portion of three limbs currently in the 0^{th} execution context's RAM.

3.5 Surgical patterns

3.5.1 Purpose

The present section compiles all variations on cutting, grafting and padding that the RAM needs and labels them. These **surgical patterns** are couched in a neutral setting in the sense that we use place holder names such as S to SB. These will later will be replaced with actual column names such as $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle$ or $\mathsf{BYTE\_A}$. We also use **markers** for what will eventually be **byte offsets** $\in \{1, \dots, \mathbf{15}\}$.

We tend to use the same variable names over and over. Here is their general interpretation: (1) the letter S and T stand, respetively, for source and target; source and target limbs are assumed counter-constant; source limbs are generally used as a source of bytes with which to modify one or more target limbs; (2) an exponent $(-)^\nu$ is meant to signal a "new" or "updated" value i.e. a value that is computed by the constraints; "new" values are always counter-constant; (3) the letter B stands for byte; (4) the letter M stands for marker i.e. a "byte marker" or "byte offset" within a limb; (5) the letter P stands for power. Thus the reader should interpret column names such as $\mathsf{S1M}$, $\mathsf{T2B}$ and T^ν as "(byte) marker in the first source limb", "bytes of the second target limb" and "new value of the target limb." Every surgical pattern is given a detailed interpretation before any constraints are written down. A picture accompanies it to make the intent clear.

3.5.2 Single byte swap

Suppose we are given

- counter-constant columns S, T and T^ν,
- byte columns SB and TB,
- binary columns $[\![1]\!]$ and $[\![2]\!]$,
- an "accumulator" column ACC,
- a counter-constant column TM,
- a column P.

The interpretation is the following: S contains a limb from which we will extract the least significant byte; TM is a marker that marks a byte in T; T contains a limb of which we wish to modify the marked byte; $[\![1]\!]$ and $[\![2]\!]$ are binary columns with threshold at T and $\mathsf{T}+1$ respectively; P is a "powers of 256" column that will allow us modify a single byte in T; the resulting limb is recorded in T^ν.

We give the set of conditions below under a name:

1. binary plateau constraints:
   (a) $\texttt{Plateau}([\![1]\!], \mathsf{TM})$
   (b) $\texttt{Plateau}([\![2]\!], \mathsf{TM} + 1)$;
2. chunk constraint: $\texttt{IsolateChunk}(\mathsf{ACC}, \mathsf{TB}, [\![1]\!], [\![2]\!])$;

3. power constraint: $\mathtt{Power}(\mathsf{P}, [\![2]\!])$;

4. update constraint:

$$\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN } \mathsf{T}^{\nu}{}_i = \mathsf{T}_i + (\mathsf{SB}_i - \mathsf{ACC}_i) \cdot \mathsf{P}_i$$

We encapsulate these constraints in a relation

$$\mathtt{ByteSwap}\left(\begin{array}{c} \mathsf{S}, \mathsf{T}, \mathsf{T}^{\nu}; \mathsf{SB}, \mathsf{TB}; \\ \mathsf{ACC}, \mathsf{P}; \mathsf{TM}, [\![1]\!], [\![2]\!]; \end{array}\right)$$

(Note: the counter column CT is implicit in this relation.)

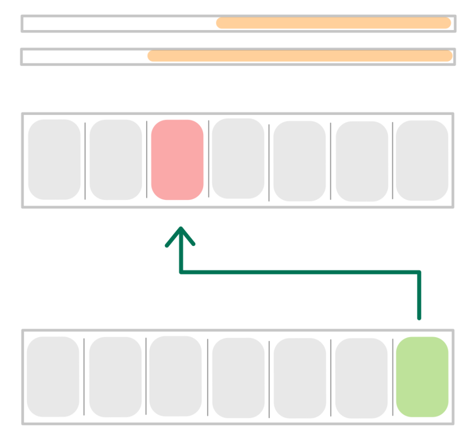

Figure 3.4: Representation of the constraints implemented by ByteSwap.

3.5.3 Excision

Suppose the following are given:

1. counter-constant columns T and T^{ν},

2. binary columns $[\![1]\!]$ and $[\![2]\!]$,

3. a byte column TB,

4. a counter-constant column TM,

5. a counter-constant column SIZE,

6. an accumulator column ACC;

7. a "powers of 256 column" column P;

The interpretation is as follows: T is a counter-constant column containing a value from which we wish to remove a chunk of consecutive bytes; TB is T's byte decomposition; T^{ν} is the counter-constant column that will contain the result of excision; TM is a byte marker in T; SIZE is the number of bytes to remove from T starting at byte offset TM; we expect $\mathsf{TM} + (\mathsf{SIZE} - 1) \leq \mathbf{15}$; $[\![1]\!]$ plateaus at TM, $[\![2]\!]$ plateaus at $\mathsf{TM} + \mathsf{SIZE}$; the bytes to be excised are accumulated in ACC; P is a "powers of 256 column" pegged to $[\![2]\!]$.

We collect the following constraints under the moniker Excision:

1. plateau constraints:

   (a) $\texttt{Plateau}([\![1]\!], \mathsf{TM})$
   (b) $\texttt{Plateau}([\![2]\!], \mathsf{TM} + \mathsf{SIZE})$

2. chunk constraint: $\texttt{IsolateChunk}(\mathsf{ACC}, \mathsf{TB}, [\![1]\!], [\![2]\!])$;
3. power constraint: $\texttt{Power}(\mathsf{P}, [\![2]\!])$
4. value enforcement:

$$\textsc{if}\ \mathsf{CT}_i = \mathbf{15}\ \textsc{then}\ \mathsf{T}^\nu_i = \mathsf{T}_i - \mathsf{ACC}_i \cdot \mathsf{P}_i$$

We subsume this collection of constraints under the moniker

$$\texttt{Excision}\left(\begin{array}{l}\mathsf{T}, \mathsf{T}^\nu; \mathsf{TB}; \mathsf{ACC}, \mathsf{P};\\ \mathsf{TM}, \mathsf{SIZE}; [\![1]\!], [\![2]\!];\end{array}\right)$$

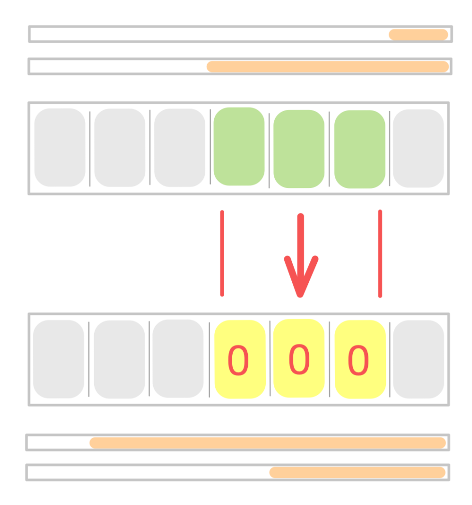

Figure 3.5: Representation of the constraints implemented by `Excision`.

3.5.4 $[1 \Rightarrow 1\ \mathsf{Padded}]$

Supppose we are given

- binary columns $[\![1]\!]$, $[\![2]\!]$ and $[\![3]\!]$,
- counter-constant columns S, T,
- a byte column SB,
- counter-constant columns SM and SIZE
- an accumulator column ACC,
- a column P.

The interpretation is as follows: S is a limb from which we will harvest a chunk of bytes; SB is the byte decomposition of S; SM is the offset within S from where we start harvesting bytes; SIZE is the number of bytes to harvest; the assumption is that $\mathsf{SM} + (\mathsf{SIZE} - 1) \leq \mathbf{15}$; T will be made to contain this chunk of bytes (left aligned); $[\![1]\!]$ plateaus at SM; $[\![2]\!]$ plateaus at $\mathsf{SM} + \mathsf{SIZE}$; $[\![3]\!]$ plateaus at SIZE; P is pegged to $[\![3]\!]$ and builds the correct power of 256 so that we may shift the harvested chunk to build the desired (left-aligned) prefix. Compare with figure ??

We the following collection of constraints ensures the desired behaviour:

1. binary plateau constraints:
    (a) $\texttt{Plateau}(\llbracket 1 \rrbracket, \mathsf{SM})$,
    (b) $\texttt{Plateau}(\llbracket 2 \rrbracket, \mathsf{SM} + \mathsf{SIZE})$,
    (c) $\texttt{Plateau}(\llbracket 3 \rrbracket, \mathsf{SIZE})$;
2. chunk constraint: $\texttt{IsolateChunk}(\mathsf{ACC}, \mathsf{SB}, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket)$;
3. power constraint: $\texttt{Power}(\mathsf{P}, \llbracket 3 \rrbracket)$;
4. value enforcement

$$\textsc{if}\ \mathsf{CT}_i = \mathbf{15}\ \textsc{then}\ \mathsf{T}_i = \mathsf{ACC}_i \cdot \mathsf{P}_i.$$

We use the short hand

$$[\mathsf{1} \Rightarrow \mathsf{1}\,\texttt{Padded}]\left(\begin{array}{c} \mathsf{S}, \mathsf{T}; \mathsf{SB}; \mathsf{ACC}, \mathsf{P}; \\ \mathsf{SM}, \mathsf{SIZE}; \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \llbracket 3 \rrbracket; \end{array}\right)$$

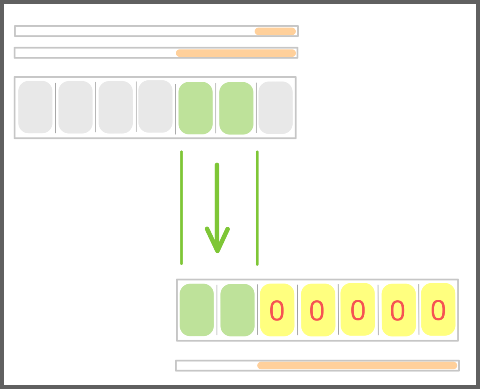

Figure 3.6: Representation of the constraints implemented by $[\mathsf{1} \Rightarrow \mathsf{1}\,\texttt{Padded}]$.

3.5.5 $[\mathsf{2} \Rightarrow \mathsf{1}\,\texttt{Padded}]$

Supppose we are given

- binary columns $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$, $\llbracket 3 \rrbracket$ and $\llbracket 4 \rrbracket$,
- counter-constant columns $\mathsf{S1}$, $\mathsf{S2}$, T,
- byte columns $\mathsf{S1B}$ and $\mathsf{S2B}$,
- counter-constant columns $\mathsf{S1M}$ and SIZE,
- accumulator columns $\mathsf{ACC\_1}$ and $\mathsf{ACC\_2}$,
- two columns $\mathsf{P1}$ and $\mathsf{P2}$.

The interpreation is as follows: $\mathsf{S1}$ contains a limb from which we extract a suffix; $\mathsf{ACC\_1}$ will accumulate the bytes of said suffix; $\mathsf{S2}$ contains a limb from which we extract a prefix; $\mathsf{ACC\_2}$ will accumulate the bytes of said prefix; $\mathsf{S1B}$ and $\mathsf{S2B}$ are the respective byte decompositions; $\mathsf{S1M}$ is the offset within $\mathsf{S1}$ from where we start harvesting bytes; SIZE is the total number of bytes to harvest; T will be made to contain the prefix extracted from $\mathsf{S1}$ followed by the prefix extracted from $\mathsf{S2}$ (left aligned); the assumption is that $\mathsf{S1M} + \mathsf{SIZE} > 16$ so that two byte sources are required to build T; $\llbracket 1 \rrbracket$ plateaus at $\mathsf{S1M}$; $\llbracket 2 \rrbracket$ plateaus at $\mathsf{S1M} + \mathsf{SIZE} - 16$; $\llbracket 3 \rrbracket$ plateaus at $16 - \mathsf{S1M}$; $\llbracket 4 \rrbracket$ plateaus at SIZE; $\mathsf{P1}$ and

$\mathsf{P2}$ are “powers of 256” columns with $\mathsf{P1}$ pegged to $\llbracket 3 \rrbracket$ and $\mathsf{P2}$ pegged to $\llbracket 4 \rrbracket$; together they build the correct powers of 256 required for shifting the extracted prefix and suffix and building T. Compare with figure ??.

The following collection of constraints ensures the desired behaviour:

1. binary plateau constraints:
   (a) $\texttt{Plateau}(\llbracket 1 \rrbracket, \mathsf{S1M})$,
   (b) $\texttt{Plateau}(\llbracket 2 \rrbracket, \mathsf{S1M} + \mathsf{SIZE} - 16)$,
   (c) $\texttt{Plateau}(\llbracket 3 \rrbracket, 16 - \mathsf{S1M})$;
   (d) $\texttt{Plateau}(\llbracket 4 \rrbracket, \mathsf{SIZE})$;
2. prefix and suffix constraints:
   (a) $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{S1B}, \llbracket 1 \rrbracket)$;
   (b) $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{S2B}, \llbracket 2 \rrbracket)$;
3. power constraints:
   (a) $\texttt{Power}(\mathsf{P1}, \llbracket 3 \rrbracket)$;
   (b) $\texttt{Power}(\mathsf{P2}, \llbracket 4 \rrbracket)$;
4. value enforcement

$$\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN } \mathsf{T}_i = \mathsf{ACC\_1}_i \cdot \mathsf{P1}_i + \mathsf{ACC\_2}_i \cdot \mathsf{P2}_i.$$

We use the short hand

$$[2 \Rightarrow 1\,\texttt{Padded}]\left(\begin{array}{c} \mathsf{S1}, \mathsf{S2}, \mathsf{T}; \mathsf{S1B}, \mathsf{S2B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{P1}, \mathsf{P2}; \\ \mathsf{S1M}, \mathsf{SIZE}; \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \llbracket 3 \rrbracket, \llbracket 4 \rrbracket; \end{array}\right)$$

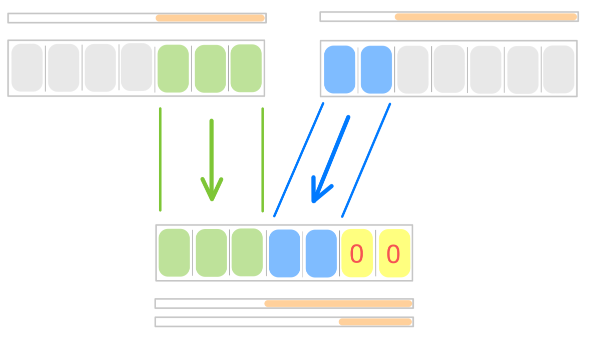

Figure 3.7: Representation of the constraints implemented by $[2 \Rightarrow 1\,\texttt{Padded}]$.

3.5.6 $[1\,\texttt{Full} \Rightarrow 2]$

Supppose we have

- binary columns $\llbracket 1 \rrbracket, \llbracket 2 \rrbracket$,
- counter-constant columns S, $\mathsf{T1}$, $\mathsf{T2}$, $\mathsf{T1}^{\nu}$, $\mathsf{T2}^{\nu}$,

- a counter-constant column $\mathsf{T1M}$,
- byte columns SB, $\mathsf{T1B}$, $\mathsf{T2B}$,
- accumulator columns $\mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}$,
- a column P.

The interpreation is as follows: S is a limb from which we will harvest *all* bytes (hence the descriptor *full*); $\mathsf{T1}$ and $\mathsf{T2}$ are limbs which we will updata using S's bytes; $\mathsf{T1}^{\nu}$ and $\mathsf{T2}^{\nu}$ are their "new" values; SB, $\mathsf{T1B}$, $\mathsf{T2B}$ are the respective byte decompositions; $\mathsf{T1M}$ is a marker for bytes in $\mathsf{T1}$; $\llbracket 1 \rrbracket$ plateaus at $\mathsf{T1M}$; $\llbracket 2 \rrbracket$ plateaus at $16 - \mathsf{T1M}$; P is pegged to $\llbracket 1 \rrbracket$ and builds the correct power of 256 so that we may change the relevant prefix of $\mathsf{T2}$.

The following collection of constraints ensures the desired behaviour.

Plateau constraints: 1. $\texttt{Plateau}(\llbracket 1 \rrbracket, \mathsf{T1M})$

2. $\texttt{Plateau}(\llbracket 2 \rrbracket, 16 - \mathsf{T1M})$

Prefix and suffix constraints: 1. $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{T1B}, \llbracket 1 \rrbracket)$,

2. $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{T2B}, \llbracket 1 \rrbracket)$,
3. $\texttt{IsolatePrefix}(\mathsf{ACC\_3}, \mathsf{SB}, \llbracket 2 \rrbracket)$,
4. $\texttt{IsolateSuffix}(\mathsf{ACC\_4}, \mathsf{SB}, \llbracket 2 \rrbracket)$,

Power constraint: $\texttt{Power}(\mathsf{P}, \llbracket 1 \rrbracket)$,

Update constraints: IF $\mathsf{CT}_i = \mathbf{15}$ THEN

1. $\mathsf{T1}^{\nu}_i = \mathsf{T1}_i + (\mathsf{ACC\_3}_i - \mathsf{ACC\_1}_i)$
2. $\mathsf{T2}^{\nu}_i = \mathsf{T2}_i + (\mathsf{ACC\_4}_i - \mathsf{ACC\_2}_i) \cdot \mathsf{P}_i$

We encapsulate all these constraints under a single relation

$$\left[1\,\texttt{Full} \Rightarrow 2\right] \left(\begin{array}{c} \mathsf{S}, \mathsf{T1}, \mathsf{T2}, \mathsf{T1}^{\nu}, \mathsf{T2}^{\nu}; \\ \mathsf{SB}, \mathsf{T1B}, \mathsf{T2B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \\ \mathsf{ACC\_3}, \mathsf{ACC\_4}, \mathsf{P}; \\ \mathsf{T1M}, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket; \end{array} \right)$$

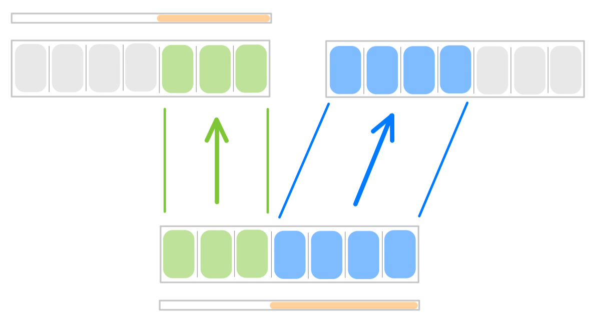

Figure 3.8: This diagram explains the $[1\,\texttt{Full} \Rightarrow 2]$ constraint and the greek letters mentioned in the constraints.

3.5.7 $[2 \Rightarrow 1\,\texttt{Full}]$

Supppose we have

- counter-constant columns $\mathsf{S1}$, $\mathsf{S2}$, T,
- a counter-constant column SM,
- binary columns $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$,
- byte columns $\mathsf{S1B}$, $\mathsf{S2B}$,
- accumulator columns $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$,
- a column P.

The interpreation is as follows: $\mathsf{S1}$, $\mathsf{S2}$ are limbs from which we will harvest a suffix and a prefix respectively; $\mathsf{S1B}$, $\mathsf{S2B}$ are the respective byte decompositions of $\mathsf{S1}$ and $\mathsf{S2}$; $\mathsf{ACC\_1}$ and $\mathsf{ACC\_2}$ accumulate the bytes of the desired suffix and prefix; T is a limb which we will construct the previously extracted suffix and prefix; SM is a marker for bytes in $\mathsf{S1}$; $\llbracket 1 \rrbracket$ plateaus at SM; $\llbracket 2 \rrbracket$ plateaus at $16 - \mathsf{SM}$; P is pegged to $\llbracket 2 \rrbracket$ and builds a power of 256: it is used to left shift the suffix extracted from $\mathsf{S1}$.

The following collection of constraints ensures the desired behaviour.

1. binary plateau constraints:
   (a) $\texttt{Plateau}(\llbracket 1 \rrbracket, \mathsf{SM})$,
   (b) $\texttt{Plateau}(\llbracket 2 \rrbracket, 16 - \mathsf{SM})$;
2. prefix and suffix constraints:
   (a) $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{S1B}, \llbracket 1 \rrbracket)$ i.e. $\mathsf{ACC\_1} \implies \alpha'$,
   (b) $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{S2B}, \llbracket 1 \rrbracket)$ i.e. $\mathsf{ACC\_2} \implies \beta$;
3. power constraint: $\texttt{Power}(\mathsf{P}, \llbracket 2 \rrbracket)$;
4. value enforcement: $\textsc{if}\ \mathsf{CT}_i = \mathbf{15}\ \textsc{then}\ \mathsf{T}_i = \mathsf{ACC\_1}_i \cdot \mathsf{P}_i + \mathsf{ACC\_2}_i$.

We encapsulate all these constraints under a single relation

$$[2 \Rightarrow 1\,\texttt{Full}] \left(\begin{array}{c} \mathsf{S1}, \mathsf{S2}, \mathsf{T}; \\ \mathsf{S1B}, \mathsf{S2B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{P}; \\ \mathsf{SM}; \llbracket 1 \rrbracket, \llbracket 2 \rrbracket; \end{array} \right)$$

3.5.8 $[1\,\texttt{Partial} \Rightarrow 1]$

Suppose we have

- binary columns $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$, $\llbracket 3 \rrbracket$, $\llbracket 4 \rrbracket$,
- counter constant columns S, T and T^ν,
- byte columns SB and TB,
- counter constant columns SM and TM,
- a counter constant column SIZE,
- a "powers" column P and "accumulator" columns $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$.

The interpretation is as follows: S and T are counter-constant columns containing limbs viewed respectively as a "source" and a "target" limb; SB and TB are their respective byte decomposition; T^ν contains the "new" value of T; SM and TM are markers $\in \{0, 1, \ldots, \mathbf{15}\}$ for for S and T respectively; we expect both $\mathsf{SM} + (\mathsf{SIZE} - 1) \leq \mathbf{15}$ and $\mathsf{TM} + (\mathsf{SIZE} - 1) \leq \mathbf{15}$; P is pegged to $[\![2]\!]$ and computes the appropriate power of 256 so that we may replace a chunk from T with a chunk from S. Compare with figure ??.

We collect the following constraints under a collective name

1. binary-plateau-constraints:
   (a) $\texttt{Plateau}([\![1]\!], \mathsf{TM})$
   (b) $\texttt{Plateau}([\![2]\!], \mathsf{TM} + \mathsf{SIZE})$
   (c) $\texttt{Plateau}([\![3]\!], \mathsf{SM})$
   (d) $\texttt{Plateau}([\![4]\!], \mathsf{SM} + \mathsf{SIZE})$
2. chunk-constraints
   (a) $\texttt{IsolateChunk}(\mathsf{ACC\_1}, \mathsf{TB}, [\![1]\!], [\![2]\!])$
   (b) $\texttt{IsolateChunk}(\mathsf{ACC\_2}, \mathsf{SB}, [\![3]\!], [\![4]\!])$
3. power-constraint: $\texttt{Power}(\mathsf{P}, [\![2]\!])$
4. update constraint:

$$\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN } \mathsf{T}^\nu{}_i = \mathsf{T}_i + (\mathsf{ACC\_2}_i - \mathsf{ACC\_1}_i) \cdot \mathsf{P}_i.$$

We encapsulate all these constraints under a single relation

$$[1\,\texttt{Partial} \Rightarrow 1]\left(\begin{array}{c} \mathsf{S}, \mathsf{T}, \mathsf{T}^\nu; \mathsf{SB}, \mathsf{TB}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{P}; \\ \mathsf{SM}, \mathsf{TM}; \mathsf{SIZE}; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!]; \end{array}\right)$$

(Note: we don't explicitly mention the CT column in this constraint, it is implicit)

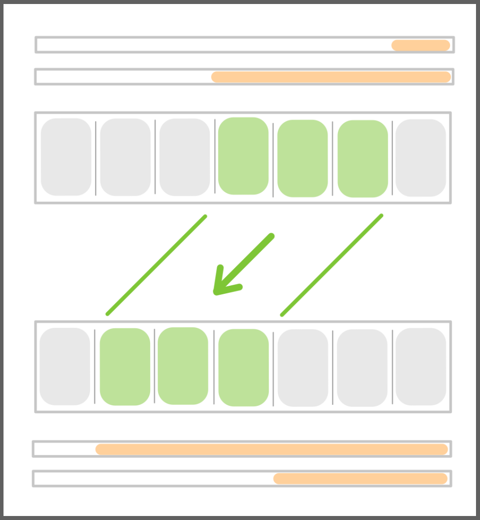

Figure 3.9: Representation of the constraints implemented by $[1\,\texttt{Partial} \Rightarrow 1]$.

3.5.9 $[1\,\texttt{Partial} \Rightarrow 2]$

Suppose we have

- binary columns $[\![1]\!]$, $[\![2]\!]$, $[\![3]\!]$, $[\![4]\!]$, $[\![5]\!]$
- counter constant columns S, $\mathsf{T1}$, $\mathsf{T2}$, $\mathsf{T1}^\nu$, $\mathsf{T2}^\nu$
- byte columns SB, $\mathsf{T1B}$ and $\mathsf{T2B}$,
- counter constant columns SM and $\mathsf{T1M}$,
- a counter constant column SIZE,
- a column P

The interpretation is as follows: S, $\mathsf{T1}$ and $\mathsf{T2}$ are counter-constant columns containing limbs viewed respectively as a "source" and two "target" limbs; SB, $\mathsf{T1B}$ and $\mathsf{T2B}$ are their respective byte decomposition; $\mathsf{T1}^\nu$ and $\mathsf{T2}^\nu$ contain the "new" value of $\mathsf{T1}$ and $\mathsf{T2}$ respectively; SM and TM are markers $\in \{0, 1, \dots, \mathbf{15}\}$ for S and $\mathsf{T1}$ respectively. We expect both $\mathsf{SM} + (\mathsf{SIZE} - 1) \leq \mathbf{15}$ and $\mathsf{TM} + (\mathsf{SIZE} - 1) \geq \mathbf{16}$. Compare with figure **??**.

We collect the following constraints under a collective name:

1. plateau constraints
   (a) $\texttt{Plateau}([\![1]\!], \mathsf{T1M})$
   (b) $\texttt{Plateau}([\![2]\!], \mathsf{T1M} + \mathsf{SIZE} - \mathbf{16})$
   (c) $\texttt{Plateau}([\![3]\!], \mathsf{SM})$
   (d) $\texttt{Plateau}([\![4]\!], \mathsf{SM} + \mathbf{16} - \mathsf{T1M})$
   (e) $\texttt{Plateau}([\![5]\!], \mathsf{SM} + \mathsf{SIZE})$
2. prefix, suffix and chunk constraints:
   (a) $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{T1B}, [\![1]\!])$,
   (b) $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{T2B}, [\![2]\!])$,
   (c) $\texttt{IsolateChunk}(\mathsf{ACC\_3}, \mathsf{SB}, [\![3]\!], [\![4]\!])$,
   (d) $\texttt{IsolateChunk}(\mathsf{ACC\_4}, \mathsf{SB}, [\![4]\!], [\![5]\!])$,
3. power-constraint: $\texttt{Power}(\mathsf{P}, [\![2]\!])$
4. update constraint:

$$\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN } \begin{cases} \mathsf{T1}^\nu{}_i = \mathsf{T1}_i + (\mathsf{ACC\_3}_i - \mathsf{ACC\_1}_i) \\ \mathsf{T2}^\nu{}_i = \mathsf{T2}_i + (\mathsf{ACC\_4}_i - \mathsf{ACC\_2}_i) \cdot \mathsf{P}_i. \end{cases}$$

We encapsulate all these constraints under a single relation

$$[1\,\texttt{Partial} \Rightarrow 2] \left(\begin{array}{c} \mathsf{S}, \mathsf{T1}, \mathsf{T2}, \mathsf{T1}^\nu, \mathsf{T2}^\nu; \mathsf{SB}, \mathsf{T1B}, \mathsf{T2B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}; \mathsf{P}; \\ \mathsf{SM}, \mathsf{T1M}, \mathsf{SIZE}; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!], [\![5]\!]; \end{array} \right)$$

(Note: we don't explicitly mention the CT column in this constraint, it is implicit)

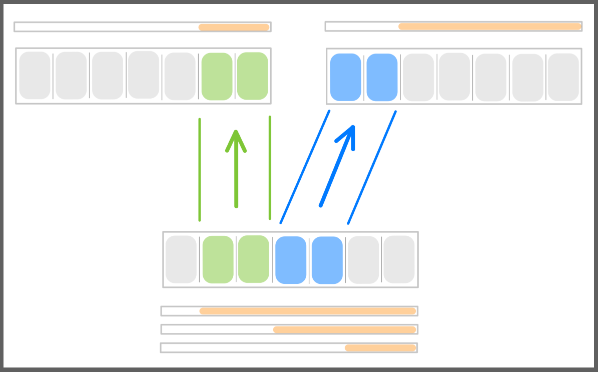

Figure 3.10: Representation of the constraints implemented by $[1\,\texttt{Partial} \Rightarrow 2]$.

3.5.10 $[2\,\texttt{Full} \Rightarrow 3]$

Suppose we are given

- counter-constant columns $\mathsf{T1}$, $\mathsf{T3}$, $\mathsf{S1}$, $\mathsf{S2}$,
- byte columns $\mathsf{T1B}$, $\mathsf{T3B}$, $\mathsf{S1B}$, $\mathsf{S2B}$,
- counter-constant columns $\mathsf{T1}^\nu$, $\mathsf{T2}^\nu$, $\mathsf{T3}^\nu$,
- counter-constant column TM,
- two binary columns $[\![1]\!]$, $[\![2]\!]$,
- a column P,
- and accumulator columns $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$, $\mathsf{ACC\_3}$, $\mathsf{ACC\_4}$, $\mathsf{ACC\_5}$, $\mathsf{ACC\_6}$.

The interpretation is as follows: $\mathsf{T1}$ and $\mathsf{T3}$ are limb columns to be modified; $\mathsf{T1B}$, $\mathsf{T3B}$ are the respective byte decompositions; $\mathsf{TM} \in \{1, \ldots, \mathbf{15}\}$ is a marker for $\mathsf{T1}$ indicating the index of the first byte to change; $\mathsf{S1}$ and $\mathsf{S2}$ are limbs from which we will extract *all* the bytes (hence the qualifier *full*); $\mathsf{S1B}$ and $\mathsf{S2B}$ are the respective byte decompositions; $\mathsf{T1}$ will have its suffix swapped out with a prefix from $\mathsf{S1}$, yielding $\mathsf{T1}^\nu$; $\mathsf{T3}$ will have its prefix swapped out with a suffix from $\mathsf{S2}$, yielding $\mathsf{T3}^\nu$; $\mathsf{T2}^\nu$ will be constructed from the remaining suffix of $\mathsf{S1}$ and the remaining prefix$\mathsf{S2}$; $[\![1]\!]$ and $[\![2]\!]$ are binary plateau columns with threshholds TM and $\mathbf{16} - \mathsf{TM}$ respetively; $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$, $\mathsf{ACC\_3}$, $\mathsf{ACC\_4}$, $\mathsf{ACC\_5}$, $\mathsf{ACC\_6}$ will hold all the relevant prefixes and suffixes; P is a "powers of 256" column pegged to $[\![1]\!]$ used to perform the adequate shifts.

1. binary plateau constraints:
   (a) $\texttt{Plateau}([\![1]\!], \mathsf{TM})$
   (b) $\texttt{Plateau}([\![2]\!], \mathbf{16} - \mathsf{TM})$
2. prefix and suffix constraints:
   (a) $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{T1B}, [\![1]\!])$,
   (b) $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{T3B}, [\![1]\!])$,
   (c) $\texttt{IsolatePrefix}(\mathsf{ACC\_3}, \mathsf{S1B}, [\![2]\!])$,
   (d) $\texttt{IsolateSuffix}(\mathsf{ACC\_4}, \mathsf{S1B}, [\![2]\!])$,
   (e) $\texttt{IsolatePrefix}(\mathsf{ACC\_5}, \mathsf{S2B}, [\![2]\!])$,

(f) $\texttt{IsolateSuffix}(\mathsf{ACC\_6}, \mathsf{S2B}, [\![2]\!])$,

3. power constraint: $\texttt{Power}(\mathsf{P}, [\![1]\!])$
4. update constraints: IF $\mathsf{CT}_i = \mathbf{15}$ THEN
   (a) $\mathsf{T1}^\nu{}_i = \mathsf{T1}_i + (\mathsf{ACC\_3}_i - \mathsf{ACC\_1}_i)$
   (b) $\mathsf{T2}^\nu{}_i = \mathsf{ACC\_4}_i \cdot \mathsf{P}_i + \mathsf{ACC\_5}_i$
   (c) $\mathsf{T3}^\nu{}_i = \mathsf{T3}_i + (\mathsf{ACC\_6}_i - \mathsf{ACC\_2}_i) \cdot \mathsf{P}_i$

We encapsulate these constraints under in a relation:

$$[\mathsf{2}\,\texttt{Full} \Rightarrow \mathsf{3}]\left(\begin{array}{c} \mathsf{T1}, \mathsf{T3}, \mathsf{S1}, \mathsf{S2}, \mathsf{T1}^\nu, \mathsf{T2}^\nu, \mathsf{T3}^\nu; \\ \mathsf{T1B}, \mathsf{T3B}, \mathsf{S1B}, \mathsf{S2B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \\ \mathsf{ACC\_4}, \mathsf{ACC\_5}, \mathsf{ACC\_6}; \\ \mathsf{P}; \mathsf{TM}; [\![1]\!], [\![2]\!]; \end{array}\right)$$

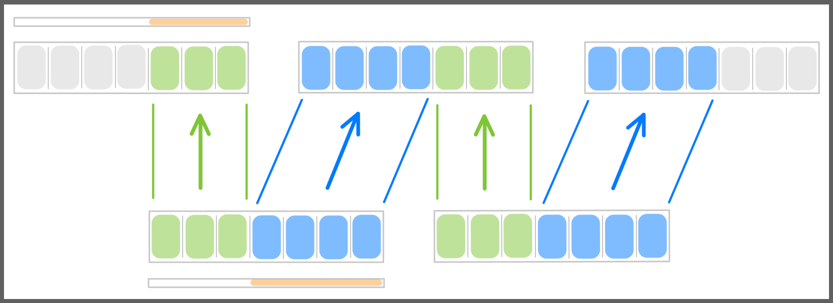

Figure 3.11: Representation of the constraints implemented by $[\mathsf{2}\,\texttt{Full} \Rightarrow \mathsf{3}]$.

3.5.11 $[\mathsf{3} \Rightarrow \mathsf{2}\,\texttt{Full}]$

Suppose we are given

- counter-constant columns $\mathsf{S1}$, $\mathsf{S2}$, $\mathsf{S3}$, $\mathsf{T1}$ and $\mathsf{T2}$,
- byte columns $\mathsf{S1B}$, $\mathsf{S2B}$, $\mathsf{S3B}$,
- SM a counter-constant column,
- binary columns $[\![1]\!]$, $[\![2]\!]$,
- accumulator columns $\mathsf{ACC\_1}$, $\mathsf{ACC\_2}$, $\mathsf{ACC\_3}$ and $\mathsf{ACC\_4}$,
- a colum P.

The intrepretation is as follows: $\mathsf{S1}$, $\mathsf{S2}$ and $\mathsf{S3}$ are viewed as “source” limbs from which we will extract prefixes and suffixes; $\mathsf{S1B}$, $\mathsf{S2B}$ and $\mathsf{S3B}$ are their byte decomposition; $\mathsf{T1}$ and $\mathsf{T2}$ are viewed as “target” limb columns; their value will be constructed from suffixes and prefixes of $\mathsf{S1}$, $\mathsf{S2}$ and $\mathsf{S3}$; SM sets a mark at a particular byte of $\mathsf{S1}$, $\mathsf{S2}$ and $\mathsf{S3}$; $[\![1]\!]$ and $[\![2]\!]$ are binary plateau columns with jump at SM and $16 - \mathsf{SM}$ respecitvely; P is a “powers of 256” column that is pegged to $[\![2]\!]$.

Figure 3.12 illustrates the effect of the $[\mathsf{3} \Rightarrow \mathsf{2}\,\texttt{Full}]$ elementary surgery. The following are the associated constraints:

Plateau constraints: 1. $\texttt{Plateau}([\![1]\!], \mathsf{SM})$

2. $\texttt{Plateau}([\![2]\!], 16 - \mathsf{SM})$

Prefix and suffix constraints: 1. $\texttt{IsolateSuffix}(\mathsf{ACC\_1}, \mathsf{S1B}, [\![1]\!])$,

2. $\texttt{IsolatePrefix}(\mathsf{ACC\_2}, \mathsf{S2B}, [\![1]\!])$,

3. $\texttt{IsolateSuffix}(\mathsf{ACC\_3}, \mathsf{S2B}, [\![1]\!])$,

4. $\texttt{IsolatePrefix}(\mathsf{ACC\_4}, \mathsf{S3B}, [\![1]\!])$,

Power constraint: $\texttt{Power}(\mathsf{P}, [\![2]\!])$

Update constraints: IF $\mathsf{CT}_i = \mathbf{15}$ THEN

1. $\mathsf{T1}_i = \mathsf{ACC\_1}_i \cdot \mathsf{P}_i + \mathsf{ACC\_2}_i$
2. $\mathsf{T2}_i = \mathsf{ACC\_3}_i \cdot \mathsf{P}_i + \mathsf{ACC\_4}_i$

We encapsulate these constraints into a single relation

$$\left[3 \Rightarrow 2\,\texttt{Full}\right]\begin{pmatrix} \mathsf{S1}, \mathsf{S2}, \mathsf{S3}, \mathsf{T1}, \mathsf{T2}; \\ \mathsf{S1B}, \mathsf{S2B}, \mathsf{S3B}; \\ [\![1]\!], [\![2]\!], \mathsf{P}, \mathsf{SM}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \\ \mathsf{ACC\_3}, \mathsf{ACC\_4}; \end{pmatrix}$$

(Note: we don't explicitly mention the CT column in this constraint, it is implicit.)

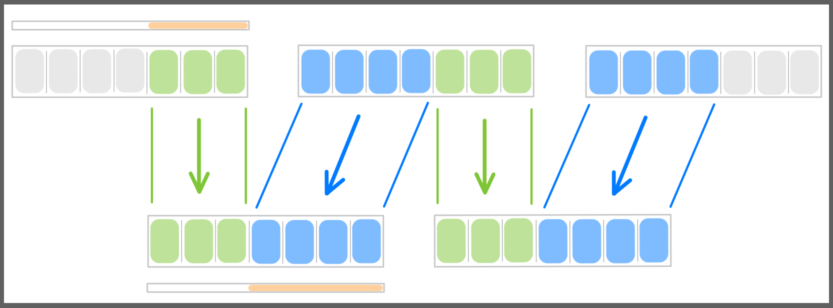

Figure 3.12: Representation of the constraints implemented by $\left[3 \Rightarrow 2\,\texttt{Full}\right]$.

3.6 Limb surgery

3.6.1 Data sources and targets

The following lists for every opcode that may trigger memory operations the possibly origin and destination.

Instructions	donor	recipient	encoding	surgeries
`LOG0-LOG4`	RAM	logs	1	6, 7, 11, 12;
`MLOAD`, `CALLDATALOAD` if $\mathsf{CALLER} \neq 0$	RAM	stack	2	1, 2;
`RETURN` if $\mathsf{CTYPE} = 1$, `CREATE(2)`	RAM	ROM	3	6, 7, 11, 12;
`CALLDATACOPY` if $\mathsf{CALLER} \neq 0$; `REVERT`, `RETURN` if $\mathsf{CTYPE} = 0$; `RETURNDATACOPY`	RAM	RAM	4 6, 8, 13;	
`MSTORE`; `MSTORE8`	stack	RAM	5	3, 4; 8;
`(EXT)CODECOPY`	ROM	RAM	6	6, 8, 9, 10;
`CALLDATACOPY` if $\mathsf{CALLER} = 0$	TXCD	RAM	7	6, 8, 9, 10;
`CALLDATALOAD` if $\mathsf{CALLER} = 0$	TXCD	stack	8	6, 8, 9, 11;

Figure 3.13: There are 8 possible data source and target configurations. The last row (i.e. `CALLDATALOAD` instructions involving transaction call data) is the only configuration not involving RAM directly. Their implementation will still involve RAM: we use the 0^{th} execution context's memoryless RAM as a pad to store 1, 2 or even 3 limbs obtained from transaction call data.

Instruction	data donor	data recipient
`SHA3`	RAM (current context)	SHA3
`MSTORE8`	stack	RAM (current context)
`MSTORE`	stack	RAM (current context)
`MLOAD`	RAM (current context)	stack
`CALLDATALOAD` if $\mathsf{CALLER} \neq 0$	RAM (caller context)	stack
`CALLDATALOAD` if $\mathsf{CALLER} = 0$	transaction call data	stack
`CALLDATACOPY` if $\mathsf{CALLER} \neq 0$	RAM (caller context)	RAM (current context)
`CALLDATACOPY` if $\mathsf{CALLER} = 0$	transaction call data	RAM (current context)
`REVERT`	RAM (current context)	RAM (caller context)
`RETURN` if $\mathsf{CTYPE} = 0$	RAM (current context)	RAM (caller context)
`RETURN` if $\mathsf{CTYPE} = 1$	RAM (current context)	ROM
`CREATE(2)`	RAM (current context)	ROM
`(EXT)CODECOPY`	ROM	RAM (current context)
`RETURNDATACOPY`	RAM (returner context)	RAM (current context)
`LOG0-LOG4`	RAM (current context)	logs

There are thus 8 possibilities in terms of data movement: To locate data within these data sources we require:

RAM: a context number and a limb offset; e.g. the current execution context number, that of the caller or that of the returner;

LOGs: a log number and a limb offset;

ROM: a code fragment number, the boolean $\mathsf{IS\_INIT}$ (indicating whether the code fragment to be read from or compared to a RAM segment) is initialization code or (currently) deployed code, and a limb offset;

Stack: nothing: just the high and low part of a value read from or written to the current execution context's stack.

3.6.2 Which opcodes require what surgeries

`MSTORE8`: we work with 1 source term (the low part of the stack value) and 1 target term (from the current RAM):

1. type 5;

MSTORE: we work with 2 source terms (the high and low part of the stack value) and 2 or 3 target terms (from the current RAM):

1. type 3 (fast operation),
2. type 4 (slow operation);

MLOAD: we work with a 2 or 3 source terms (from the current RAM) and 2 target terms (the high and low part of the stack value):

1. either type 1 (fast operation),
2. or type 2 (slow operation);

CALLDATALOAD: if $\mathsf{CALLER} \neq 0$ and $\mathsf{OFFSET} + 32 \leq \mathsf{CDS}$ we work with a 2 or 3 source terms (from the current RAM) and 2 target terms (the high and low part of the stack value):

1. either type 1 (fast operation),
2. or type 2 (slow operation);

otherwise the operation is split into two sub operations using either 1 or 2 source terms and a 1 target term (the high / low part of the stack value in that order):

1. type 6 twice (2): 66
2. type 7 (full) twice (3): 77,
3. type 6 followed by 9 (1): 69,
4. type 6 followed by 11 (2): 6b,
5. type 7 (full) followed by 11 (2): 72,
6. type 7 (full) followed by 12 (3): 7c,
7. type 11 followed by type 9 (1): b9,
8. type 12 followed by type 9 (2): c9,
9. type 9 twice: 99;

the number in parenthesis indicates the number of loads from transaction calldata required when $\mathsf{CALLER} = 0$;

LOGs and RETURN for contract deployment: —

1. a sequence of 6's potentially followed by 11: $6^*(b)$
2. a sequence of 7's potentially followed by an 11 or 12: $7^*(b/c)$

RETURN and REVERT:

1. potential 8 followed by a sequence of 6's potentially followed by an 8: $(8)6^*(8)$,
2. potential 8 followed by a sequence of 7's potentially followed by an 8: $(8)7^*(8)$

RETURNDATACOPY: we work with a single source term:

1. a sequence of 6's potentially followed by an 8: $6^*(8)$,
2. a sequence of 7's potentially followed by an 8: $7^*(8)$ (the last 7 may be incomplete if there is no 8)

CALLDATACOPY: we work with a single source term (from $\mathsf{TRANSACTION\_CALLDATA\_PADDED}$ or the CALLER's RAM) and 1 or 2 target terms in RAM:

1. potential first completion (8) followed by quick copies 6^* followed by potential loading a piece followed by potentially completing the limb with 0's (8 or $8a$) followed potentially by many full zero limbs (9^*) followed by potentially some zeros (a): $(8)6^*(8a/8)(9^*)(a)$;
2. potential first completion (8) followed by slow compies $(d)^*$ followed potentially by some zero padding (a) followed potentially by fast zeros (9^*) followed potentially by some zeros (a), i.e. $(8)(d^*)(a)(9^*)(a)$;

(EXT)CODECOPY: we work with a single source term (from ROM) 1 or 2 target terms in RAM:

1. a sequence of 6's potentially followed by 9's (padding is part of the bytecode) and/or a single 10: $6^*9^*(10)$
2. a sequence of 13's potentially followed by a 10 and potentially 9's and potentially a 10: $d^*(a9^*(10))$

3.6.3 RAM to RAM

RAM limb excision

The surgery described below is used by instructions writing to RAM where the source data may run out of bounds. In other words:

1. CALLDATACOPY,
2. RETURNDATACOPY,
3. CODECOPY,
4. EXTCODECOPY.

We label it RamLimbExcision. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i = 0 \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \end{cases}$$

2. Surgery constraint:

$$\texttt{Excision}\left(\begin{array}{c} \mathsf{VAL\_B}, \mathsf{VAL\_B}^{\nu}; \mathsf{BYTE\_B}; \mathsf{ACC\_1}; \\ \mathsf{POW\_256\_1}; \langle \mathsf{TBO} \rangle; \langle \mathsf{SIZE} \rangle; [\![1]\!], [\![2]\!]; \end{array}\right)$$

Chunk sliding no overlap

This subsection defines the RamToRamSlideChunk surgery. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \end{cases}$$

2. Surgery constraint:

$$[1\,\texttt{Partial} \Rightarrow 1]\left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}, \mathsf{VAL\_B}^{\nu}; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{POW\_256\_1}; \\ \langle\mathsf{SBO}\rangle, \langle\mathsf{TBO}\rangle; \langle\mathsf{SIZE}\rangle; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!]; \end{array}\right)$$

Chunk sliding with overlap

The surgery `RamToRamSlideOverlappingChunk` below is comprised of the following constraints:

1. Wiring constraints:

$$\left\{\begin{array}{l} \mathsf{CN\_A}_i = \langle\mathsf{CN\_S}\rangle_i \\ \mathsf{CN\_B}_i = \langle\mathsf{CN\_T}\rangle_i \\ \mathsf{CN\_C}_i = \langle\mathsf{CN\_T}\rangle_i \\ \mathsf{INDEX\_A}_i = \langle\mathsf{SLO}\rangle_i \\ \mathsf{INDEX\_B}_i = \langle\mathsf{TLO}\rangle_i \\ \mathsf{INDEX\_C}_i = \langle\mathsf{TLO}\rangle_i + 1 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \end{array}\right.$$

2. Surgery constraint:

$$[1\,\texttt{Partial} \Rightarrow 2]\left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}, \mathsf{VAL\_C}, \mathsf{VAL\_B}^{\nu}, \mathsf{VAL\_C}^{\nu}; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}, \mathsf{BYTE\_C}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}; \\ \mathsf{POW\_256\_1}; \langle\mathsf{SBO}\rangle, \langle\mathsf{TBO}\rangle, \langle\mathsf{SIZE}\rangle; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!], [\![5]\!]; \end{array}\right)$$

3.6.4 Exogenous data to RAM

Chunk sliding no overlap

The surgery `ExoToRamSlideChunk` below is used by

1. `CALLDATACOPY` in a context that is the root context of a transaction,
2. `CODECOPY` and `EXTCODECOPY`,

It is comprised of the following constraints:

1. Wiring constraints:

$$\left\{\begin{array}{l} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = \langle\mathsf{CN\_T}\rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = \langle\mathsf{TLO}\rangle_i \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle\mathsf{SLO}\rangle_i \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i = 0 \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \end{array}\right.$$

2. Surgery constraint:

$$[1\,\texttt{Partial} \Rightarrow 1]\left(\begin{array}{c} \langle\mathsf{VAL\_X}\rangle, \mathsf{VAL\_B}, \mathsf{VAL\_B}^{\nu}; \mathsf{BYTE\_X}, \mathsf{BYTE\_B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{POW\_256\_1} \\ \langle\mathsf{SBO}\rangle, \langle\mathsf{TBO}\rangle; \langle\mathsf{SIZE}\rangle; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!]; \end{array}\right)$$

Chunk sliding with overlap

The surgery `ExoToRamSlideOverlappingChunk` below is used by

1. `CALLDATACOPY` in a context that is the root context of a transaction,
2. `CODECOPY` and `EXTCODECOPY`.

It is comprised of the following constraints:

1. Wiring constraints:
$$\begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_C}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i = 0 \end{cases}$$
2. Surgery constraint:
$$[1\,\texttt{Partial} \Rightarrow 2] \left(\begin{array}{c} \langle \mathsf{VAL\_X} \rangle, \mathsf{VAL\_B}, \mathsf{VAL\_C}, \mathsf{VAL\_B}^\nu, \mathsf{VAL\_C}^\nu; \\ \mathsf{BYTE\_X}, \mathsf{BYTE\_B}, \mathsf{BYTE\_C}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}; \\ \mathsf{POW\_256\_1}; \langle \mathsf{SBO} \rangle, \langle \mathsf{TBO} \rangle, \langle \mathsf{SIZE} \rangle; \\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!], [\![5]\!]; \end{array} \right)$$

3.6.5 RAM to exogenous data

Use cases

The surgeries `FullExoFromTwo`, `PaddedExoFromTwo` and `PaddedExoFromOne` presented below are used in the following memory instructions:

1. `LOG0`-`LOG4` instructions,
2. `CREATE` and `CREATE2` instructions,
3. `RETURN` in a deployment context which is (temporarily) successful,
4. `SHA3`

Left aligned padded chunk from one RAM limb

The surgery `PaddedExoFromOne` below is comprised of the following constraints:

1. Wiring constraints:
$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{INDEX\_X}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \end{cases}$$

2. Surgery constraint:

$$[\mathsf{1} \Rightarrow \mathsf{1\,Padded}]\left(\begin{array}{c}\mathsf{VAL\_A}, \langle \mathsf{VAL\_X} \rangle; \mathsf{BYTE\_A};\\ \mathsf{ACC\_1}; \mathsf{POW\_256\_1};\\ \langle \mathsf{SBO} \rangle; \langle \mathsf{SIZE} \rangle;\\ [\![1]\!], [\![2]\!], [\![3]\!];\end{array}\right)$$

Left aligned padded chunk from two RAM limbs

The surgery `PaddedExoFromTwo` is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases}\mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i\\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i\\ \mathsf{CN\_C}_i = 0\\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i\\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1\\ \mathsf{INDEX\_C}_i = 0\\ \mathsf{INDEX\_X}_i = \langle \mathsf{TLO} \rangle_i\\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i\\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i\\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0\end{cases}$$

2. Surgery constraint:

$$[\mathsf{2} \Rightarrow \mathsf{1\,Padded}]\left(\begin{array}{c}\mathsf{VAL\_A}, \mathsf{VAL\_B}, \langle \mathsf{VAL\_X} \rangle;\\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B};\\ \mathsf{ACC\_1}, \mathsf{ACC\_2};\\ \mathsf{POW\_256\_1}, \mathsf{POW\_256\_2};\\ \langle \mathsf{SBO} \rangle, \langle \mathsf{SIZE} \rangle;\\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!];\end{array}\right)$$

Full exo limb from neighboring limbs

The surgery `FullExoFromTwo` is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases}\mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i\\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i\\ \mathsf{CN\_C}_i = 0\\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i\\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1\\ \mathsf{INDEX\_C}_i = 0\\ \mathsf{INDEX\_X}_i = \langle \mathsf{TLO} \rangle_i\\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i\\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i\\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0\end{cases}$$

2. Surgery constraint:

$$[\mathsf{2} \Rightarrow \mathsf{1\,Padded}]\left(\begin{array}{c}\mathsf{VAL\_A}, \mathsf{VAL\_B}, \langle \mathsf{VAL\_X} \rangle;\\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B};\\ \mathsf{ACC\_1}, \mathsf{ACC\_2};\\ \mathsf{POW\_256\_1}, \mathsf{POW\_256\_2};\\ \langle \mathsf{SBO} \rangle, \langle \mathsf{SIZE} \rangle;\\ [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!];\end{array}\right)$$

3.6.6 Stack to RAM

Full transfer

The following surgery, which we label `FullStackToRAM` , is used by the `MSTORE` instruction when offsets aren't aligned (i.e. $\langle \mathsf{TBO} \rangle \neq 0$).

1. cabling constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_C}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = \langle \mathsf{TLO} \rangle_i + 2 \end{cases}$$

2. surgery constraint:

$$[\mathsf{2}\,\texttt{Full} \Rightarrow \mathsf{3}] \begin{pmatrix} \mathsf{VAL\_A}, \mathsf{VAL\_C}; \langle \mathsf{VAL}^{\mathsf{hi}} \rangle, \langle \mathsf{VAL}^{\mathsf{lo}} \rangle; \\ \mathsf{VAL\_A}^{\nu}, \mathsf{VAL\_B}^{\nu}, \mathsf{VAL\_C}^{\nu}; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_C}, \mathsf{BYTE\_HI}, \mathsf{BYTE\_LO}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \\ \mathsf{ACC\_4}, \mathsf{ACC\_5}, \mathsf{ACC\_6}; \\ \mathsf{POW\_256\_1}; \langle \mathsf{TBO} \rangle; [\![1]\!], [\![2]\!]; \end{pmatrix}$$

Byte transfer

The following surgery is used by the `MSTORE8` instruction alone.

1. cabling constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_T} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{TLO} \rangle \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_B}^{\nu} = \mathsf{VAL\_B} = 0 \\ \mathsf{VAL\_C}^{\nu} = \mathsf{VAL\_C} = 0 \end{cases}$$

2. surgery constraint:

$$\texttt{ByteSwap} \begin{pmatrix} \langle \mathsf{VAL}^{\mathsf{lo}} \rangle, \mathsf{VAL\_A}, \mathsf{VAL\_A}^{\nu}; \\ \mathsf{BYTE\_LO}, \mathsf{BYTE\_A}; \\ \mathsf{ACC\_1}, \mathsf{POW\_256\_1}; \\ \langle \mathsf{TBO} \rangle, [\![1]\!], [\![2]\!]; \end{pmatrix}$$

In their entirety we dub this `LsbFromStackToRAM`

3.6.7 RAM to stack: aligned offsets

Fast high / padded low

The surgery described below is used by `CALLDATALOAD` in a context that isn't the root context when the 32 bytes to retrieve from call data go out of bounds (but more than 16 bytes are in range). We label it `FirstFastSecondPadded`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i = \mathsf{VAL\_A}_i \end{cases}$$

2. Surgery constraint:

$$[\mathsf{1} \Rightarrow \mathsf{1\,Padded}] \left(\begin{array}{c} \mathsf{VAL\_B}, \langle \mathsf{VAL}^{\mathsf{lo}} \rangle; \mathsf{BYTE\_B}; \\ \mathsf{ACC\_1}; \mathsf{POW\_256\_1}; \\ 0, \langle \mathsf{SIZE} \rangle; [\![1]\!], [\![2]\!]; \end{array} \right)$$

Note. The zero in the middle indicate the “zero column”. The column $[\![1]\!]$ will be equal to one along any counter-cycle where this constraint is active.

Padded high / zero low

The surgery described below is used by `CALLDATALOAD` in a context that isn’t the root context when the 32 bytes to retrieve from call data go out of bounds (with fewer than 16 bytes being in range). We label it `FirstPaddedSecondZero`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[\mathsf{1} \Rightarrow \mathsf{1\,Padded}] \left(\begin{array}{c} \mathsf{VAL\_A}, \langle \mathsf{VAL}^{\mathsf{hi}} \rangle; \mathsf{BYTE\_A}; \\ \mathsf{ACC\_1}; \mathsf{POW\_256\_1}; \\ 0, \langle \mathsf{SBO} \rangle; [\![1]\!], [\![2]\!]; \end{array} \right)$$

Note. The same comment as before applies.

3.6.8 RAM to stack: non-aligned offsets

Purpose

The surgeries described in this subsection:

1. `Exceptional_RamToStack_3To2Full`
2. `NA_RamToStack_3To2Full`
3. `NA_RamToStack_3To2Padded`
4. `NA_RamToStack_2To2Padded`
5. `NA_RamToStack_2To1FullAndZero`
6. `NA_RamToStack_2To1PaddedAndZero`
7. `NA_RamToStack_1To1PaddedAndZero`

All of these surgeries are used almost exclusively by `CALLDATALOAD` (except for `NA_RamToStack_3To2Full` which `MLOAD` also uses). It is a surprising fact that the arithmetization of the `CALLDATALOAD` instruction turns out feature so many subcases in our system. We go into more details about what makes this instruction particularly nasty in section 3.4.1.

Exceptional three RAM limbs $\rightsquigarrow$ two full stack elements

The surgery described below is used *exclusively* by `CALLDATALOAD` in a root context, i.e. after loading from transaction call data into the 0^{th} execution context's RAM with the $\langle \mathsf{ERF} \rangle = 1$. We label it `Exceptional_RamToStack_3To2Full`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = 0 \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = 0 \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = 0 \\ \mathsf{VAL\_B}_i^{\nu} = 0 \\ \mathsf{VAL\_C}_i^{\nu} = 0 \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[3 \Rightarrow 2\,\texttt{Full}] \left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}, \mathsf{VAL\_C}; \langle \mathsf{VAL}^{\mathsf{hi}} \rangle, \langle \mathsf{VAL}^{\mathsf{lo}} \rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}, \mathsf{BYTE\_C}; \\ \llbracket 1 \rrbracket, \llbracket 2 \rrbracket; \mathsf{POW\_256\_1}; \langle \mathsf{SBO} \rangle; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}; \end{array} \right)$$

Note. `Exceptional_RamToStack_3To2Full` will only ever be called after some preliminary loading from transaction call data to the 0^{th} execution context's RAM. Recall that these operations set the flag $\langle \mathsf{ERF} \rangle = 1$ which allows the 0^{th} execution context's RAM to retain information for a few consecutive (fast) micro instructions.

Three RAM limbs $\rightsquigarrow$ two full stack elements

The surgery described below is used by `MLOAD` under all circumstances, but also by `CALLDATALOAD`. Let us be precise about the second use case: it applies when both

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes are all within the CALLER's RAM segment that it designated as call data in the `CALL` instruction.

We label it `NA_RamToStack_3To2Full`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = \langle \mathsf{SLO} \rangle_i + 2 \\ \mathsf{VAL\_A}^{\nu}_i = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}^{\nu}_i = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}^{\nu}_i = \mathsf{VAL\_C}_i \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[3 \Rightarrow 2\,\texttt{Full}] \left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}, \mathsf{VAL\_C}; \langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle, \langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}, \mathsf{BYTE\_C}; \\ [\![1]\!], [\![2]\!]; \mathsf{POW\_256\_1}; \langle \mathsf{SBO} \rangle; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}, \mathsf{ACC\_3}, \mathsf{ACC\_4}; \end{array} \right)$$

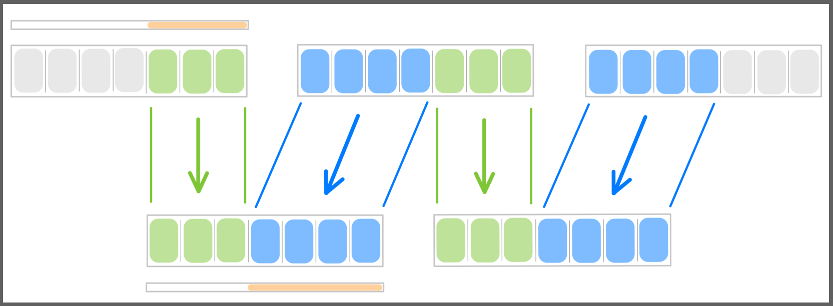

Figure 3.14: Representation of the constraints implemented by `NA_RamToStack_3To2Full`.

Three RAM limbs ⇝ a full stack element and a padded one

The surgery described below is used by `CALLDATALOAD`: it applies when

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes overflow the CALLDATA_SIZE (i.e. zero padding is required),
3. the relevant bytes span 3 limbs from the CALLER context.

(The CALLER context number is passed down in $\langle \mathsf{CN\_S} \rangle$ by the RAM preprocessor.) We label this surgery `NA_RamToStack_3To2Padded`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = \langle \mathsf{TLO} \rangle_i + 2 \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[\mathsf{2} \Rightarrow \mathsf{1}\,\texttt{Full}] \left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}; \langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}; \\ [\![1]\!], [\![2]\!]; \mathsf{POW\_256\_1}; \langle \mathsf{SBO} \rangle; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \end{array} \right)$$

and

$$[\mathsf{2} \Rightarrow \mathsf{1}\,\texttt{Padded}] \left(\begin{array}{c} \mathsf{VAL\_B}, \mathsf{VAL\_C}; \langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle; \\ \mathsf{BYTE\_B}, \mathsf{BYTE\_C}; \\ \mathsf{ACC\_3}, \mathsf{ACC\_4}; \\ \mathsf{POW\_256\_1}, \mathsf{POW\_256\_2}; \\ \langle \mathsf{SBO} \rangle, \mathsf{SIZE}; [\![1]\!], [\![3]\!], [\![2]\!], [\![4]\!]; \end{array} \right)$$

Note: $[\![1]\!]$, $[\![2]\!]$ are used twice. Also, unless I'm mistaken the order $[\![1]\!]$, $[\![3]\!]$, $[\![2]\!]$, $[\![4]\!]$ is the right one.

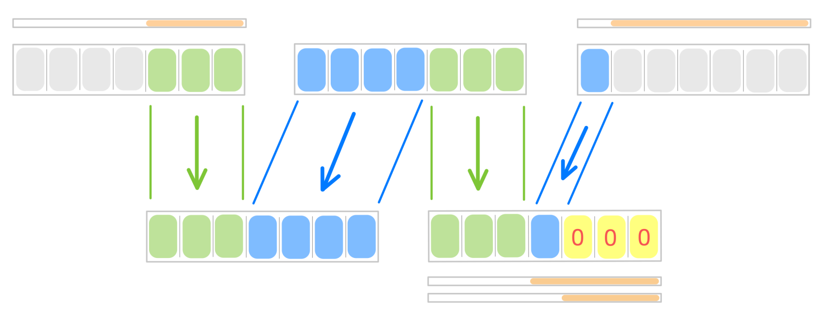

Figure 3.15: Representation of the constraints implemented by `NA_RamToStack_3To2Padded`.

Two RAM limbs ⇝ a full stack element and a padded one

The surgery described below is used by `CALLDATALOAD`: it applies when

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes overflow the $\mathsf{CALLDATA\_SIZE}$ (i.e. zero padding is required),
3. the relevant bytes span 2 limbs from the CALLER context.

(The CALLER context number is passed down in $\langle \mathsf{CN\_S} \rangle$ by the RAM preprocessor.) We label this surgery `NA_RamToStack_2To2Padded`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{TLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[2 \Rightarrow 1\,\texttt{Full}] \begin{pmatrix} \mathsf{VAL\_A}, \mathsf{VAL\_B}; \langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}; \\ [\![1]\!], [\![2]\!]; \mathsf{POW\_256\_1}; \langle \mathsf{SBO} \rangle; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \end{pmatrix}$$

and

$$[1 \Rightarrow 1\,\texttt{Padded}] \begin{pmatrix} \mathsf{VAL\_B}; \langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle; \\ \mathsf{BYTE\_B}; \\ \mathsf{ACC\_3}; \mathsf{POW\_256\_2}; \\ \langle \mathsf{SBO} \rangle, \mathsf{SIZE}; [\![1]\!], [\![3]\!], [\![4]\!]; \end{pmatrix}$$

Note: $[\![1]\!]$ is used twice.

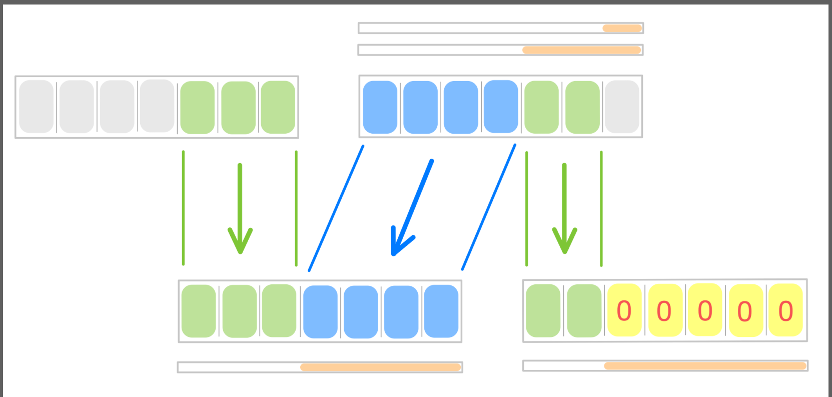

Figure 3.16: Representation of the constraints implemented by `NA_RamToStack_2To2Padded`.

Two RAM limbs ⇝ a full stack element and zero

The surgery described below is used by `CALLDATALOAD`: it applies when

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes overflow the $\mathsf{CALLDATA\_SIZE}$ (i.e. zero padding is required),
3. *precisely* 16 bytes of the requested bytes are in the call data,
4. the relevant bytes span 2 limbs from the CALLER context.

(The CALLER context number is passed down in $\langle\mathsf{CN\_S}\rangle$ by the RAM preprocessor.) We label this surgery `NA_RamToStack_2To1FullAndZero`. It is comprised of the following constraints:

1. Wiring constraints:
$$\begin{cases} \mathsf{CN\_A}_i = \langle\mathsf{CN\_S}\rangle_i \\ \mathsf{CN\_B}_i = \langle\mathsf{CN\_S}\rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle\mathsf{SLO}\rangle_i \\ \mathsf{INDEX\_B}_i = \langle\mathsf{SLO}\rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 \\ \langle\mathsf{FAST}\rangle_i = 0 \\ \langle\mathsf{ERF}\rangle_i = 0 \end{cases}$$

2. Surgery constraint:
$$[2 \Rightarrow 1\,\mathtt{Full}]\left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}; \langle\mathsf{VAL}^{\mathsf{hi}}\rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}; \\ [\![1]\!], [\![2]\!]; \mathsf{POW\_256\_1}; \langle\mathsf{SBO}\rangle; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \end{array}\right)$$

Note: we set $\langle\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0$ in the wiring constraints.

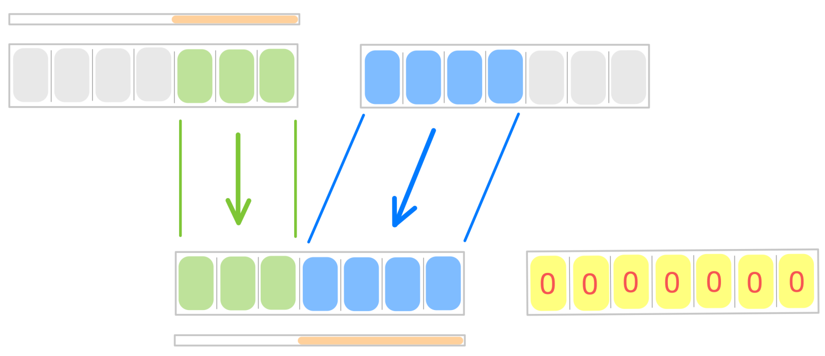

Figure 3.17: Representation of the constraints implemented by `NA_RamToStack_3To2Full`.

Two RAM limbs ⇝ a padded stack element and zero

The surgery described below is used by `CALLDATALOAD`: it applies when

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes overflow the $\mathsf{CALLDATA\_SIZE}$ (i.e. zero padding is required),
3. fewer than 16 bytes of the requested bytes are in the call data,
4. the relevant bytes span 2 limbs from the CALLER context.

(The CALLER context number is passed down in $\langle\mathsf{CN\_S}\rangle$ by the RAM preprocessor.) We label this surgery `NA_RamToStack_2To1PaddedAndZero`. It is comprised of the following constraints:

1. Wiring constraints:

$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = \langle \mathsf{SLO} \rangle_i + 1 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^{\nu} = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^{\nu} = \mathsf{VAL\_B}_i \\ \mathsf{VAL\_C}_i^{\nu} = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0 \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:

$$[2 \Rightarrow 1\,\mathsf{Padded}] \left(\begin{array}{c} \mathsf{VAL\_A}, \mathsf{VAL\_B}; \langle \mathsf{VAL}^{\mathsf{hi}} \rangle; \\ \mathsf{BYTE\_A}, \mathsf{BYTE\_B}; \\ \mathsf{ACC\_1}, \mathsf{ACC\_2}; \mathsf{POW\_256\_1}, \mathsf{POW\_256\_2}; \\ \langle \mathsf{SBO} \rangle, \mathsf{SIZE}; [\![1]\!], [\![2]\!], [\![3]\!], [\![4]\!]; \end{array} \right)$$

Note: we set $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0$ in the wiring constraints.

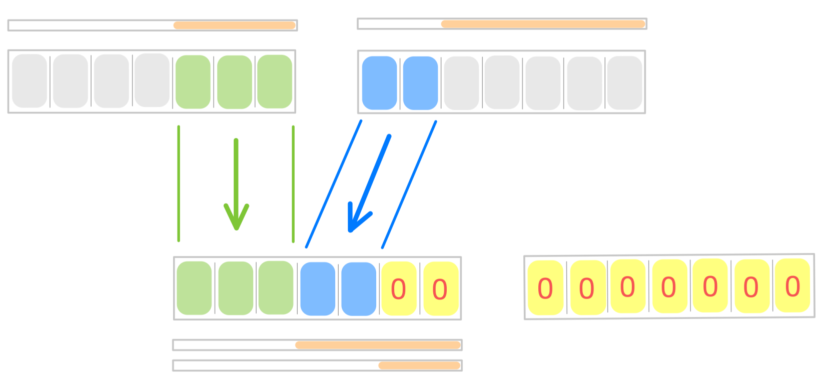

Figure 3.18: Representation of the constraints implemented by `NA_RamToStack_2To1PaddedAndZero`.

One RAM limb ⇝ a padded stack element and zero

The surgery described below is used by `CALLDATALOAD`: it applies when

1. the context executing `CALLDATALOAD` isn't the root context of a transaction,
2. the requested 32 bytes overflow the CALLDATA_SIZE (i.e. zero padding is required),
3. fewer than 16 bytes of the requested bytes are in the call data,
4. the relevant bytes span 1 limb from the CALLER context.

(The CALLER context number is passed down in $\langle \mathsf{CN\_S} \rangle$ by the RAM preprocessor.) We label this surgery `NA_RamToStack_1To1PaddedAndZero`. It is comprised of the following constraints:

1. Wiring constraints:
$$\begin{cases} \mathsf{CN\_A}_i = \langle \mathsf{CN\_S} \rangle_i \\ \mathsf{CN\_B}_i = 0 \\ \mathsf{CN\_C}_i = 0 \\ \mathsf{INDEX\_A}_i = \langle \mathsf{SLO} \rangle_i \\ \mathsf{INDEX\_B}_i = 0 \\ \mathsf{INDEX\_C}_i = 0 \\ \mathsf{VAL\_A}_i^\nu = \mathsf{VAL\_A}_i \\ \mathsf{VAL\_B}_i^\nu = \mathsf{VAL\_B}_i = 0 \\ \mathsf{VAL\_C}_i^\nu = \mathsf{VAL\_C}_i = 0 \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0 \\ \langle \mathsf{FAST} \rangle_i = 0 \\ \langle \mathsf{ERF} \rangle_i = 0 \end{cases}$$

2. Surgery constraint:
$$[1 \Rightarrow 1\,\mathsf{Padded}] \left(\begin{array}{c} \mathsf{VAL\_A}; \langle \mathsf{VAL}^{\mathsf{hi}} \rangle; \mathsf{BYTE\_A}; \\ \mathsf{ACC\_1}; \mathsf{POW\_256\_1}; \\ \langle \mathsf{SBO} \rangle, \mathsf{SIZE}; [\![1]\!], [\![2]\!], [\![3]\!]; \end{array} \right)$$

Note: we set $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i = 0$ in the wiring constraints.

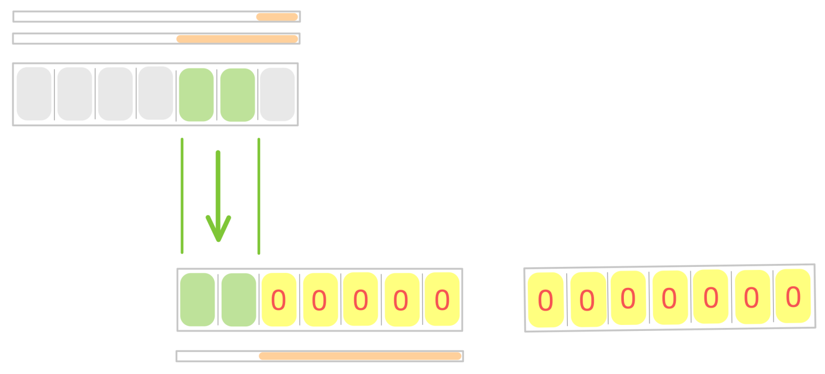

Figure 3.19: Representation of the constraints implemented by `NA_RamToStack_1To1PaddedAndZero`.

3.7 Consistency constraints

3.7.1 Call stack consistency

The execution trace carries meta information about the call stack and about offsets and sizes for call data and return data. When returning or reverting to a previous context we must recuperate said meta information. The constraints here ensure the validity of this information. We shall reorder some columns accoring to the lexicgraphic order on the pair

$$\Big(\mathsf{CONTEXT\_NUMBER}, \mathsf{RAM\_TIMESTAMP} \Big)$$

We will need the following reordered columns:

1. $[\mathsf{CN}]^{\text{🔀}}$
2. $[\square]^{\text{🔀}}$
3. $[\mathsf{CALLER}]^{\text{🔀}}$
4. $[\mathsf{CALLDATA\_OFFSET}]^{\text{🔀}}$
5. $[\mathsf{CALLDATA\_SIZE}]^{\text{🔀}}$
6. $[\mathsf{RETURNER}]^{\text{🔀}}$
7. $[\mathsf{RETURNDATA\_OFFSET}]^{\text{🔀}}$
8. $[\mathsf{RETURNDATA\_SIZE}]^{\text{🔀}}$

where by definition $\left([\mathsf{CN}]^{\text{🔀}}, [\square]^{\text{🔀}}\right)$ is lexicographically sorted. We impose the following consistency constraints:

1. call-data-meta-information consistency: IF $[\mathsf{CN}]^{\text{🔀}}_{i+1} = [\mathsf{CN}]^{\text{🔀}}_{i}$ THEN

$$\begin{cases} [\mathsf{CALLER}]^{\text{🔀}}_{i+1} = [\mathsf{CALLER}]^{\text{🔀}}_{i} \\ [\mathsf{CALLDATA\_OFFSET}]^{\text{🔀}}_{i+1} = [\mathsf{CALLDATA\_OFFSET}]^{\text{🔀}}_{i} \\ [\mathsf{CALLDATA\_SIZE}]^{\text{🔀}}_{i+1} = [\mathsf{CALLDATA\_SIZE}]^{\text{🔀}}_{i} \end{cases}$$

2. return-data-meta-information consistency: IF $\Big([\mathsf{CN}]^{\text{🔀}}_{i+1} = [\mathsf{CN}]^{\text{🔀}}_{i}$ AND $[\mathsf{RETURNER}]^{\text{🔀}}_{i+1} = [\mathsf{RETURNER}]^{\text{🔀}}_{i} \Big)$ THEN

$$\begin{cases} [\mathsf{RETURNDATA\_OFFSET}]^{\text{🔀}}_{i+1} = [\mathsf{RETURNDATA\_OFFSET}]^{\text{🔀}}_{i} \\ [\mathsf{RETURNDATA\_SIZE}]^{\text{🔀}}_{i+1} = [\mathsf{RETURNDATA\_SIZE}]^{\text{🔀}}_{i} \end{cases}$$

3.7.2 Concatenated columns and order

We introduce several interleaved columns:

1. $\mathsf{CN\_ABC} := \mathsf{CN\_A} \boxplus \mathsf{CN\_B} \boxplus \mathsf{CN\_C}$,
2. $\mathsf{INDEX\_ABC} := \mathsf{INDEX\_A} \boxplus \mathsf{INDEX\_B} \boxplus \mathsf{INDEX\_C}$
3. $\langle \mu\mathsf{RST} \rangle^{\boxplus 3} := \langle \mu\mathsf{RST} \rangle \boxplus \langle \mu\mathsf{RST} \rangle \boxplus \langle \mu\mathsf{RST} \rangle$
4. $\mathsf{VAL\_ABC} := \mathsf{VAL\_A} \boxplus \mathsf{VAL\_B} \boxplus \mathsf{VAL\_C}$
5. $\mathsf{VAL\_ABC}^{\nu} := \mathsf{VAL\_A}^{\nu} \boxplus \mathsf{VAL\_B}^{\nu} \boxplus \mathsf{VAL\_C}^{\nu}$

We introduce their reordered variants: $[\mathsf{CN\_ABC}]^{\text{🔀}}$, $[\mathsf{INDEX\_ABC}]^{\text{🔀}}$, $[\langle \mu\mathsf{RST} \rangle^{\boxplus 3}]^{\text{🔀}}$, $[\mathsf{VAL\_ABC}]^{\text{🔀}}$, $[\mathsf{VAL\_ABC}^{\nu}]^{\text{🔀}}$ where reordering is done according to the lexicographic order on the triple

$$(\mathsf{CONTEXT\_NUMBER}, \mathsf{INDEX}, \langle \mu\mathsf{RST} \rangle)$$

in other words, the reordering is such that

$$\left([\mathsf{CN\_ABC}]^{\text{🔀}},\ [\mathsf{INDEX\_ABC}]^{\text{🔀}},\ \left[\langle \mu\mathsf{RST} \rangle^{\boxplus 3}\right]^{\text{🔀}} \right)$$

are lexicographically ordered.

3.7.3 Memory consistency constraints

1. There are no constraints when $[\mathsf{CN\_ABC}]_i^{\text{🔀}} = 0$

2. IF $[\mathsf{CN\_ABC}]_i^{\text{🔀}} \neq 0$:

   (a) IF

   $$\begin{cases} [\mathsf{CN\_ABC}]_{i+1}^{\text{🔀}} = [\mathsf{CN\_ABC}]_i^{\text{🔀}} \\ \qquad \text{AND} \\ [\mathsf{INDEX\_ABC}]_{i+1}^{\text{🔀}} = [\mathsf{INDEX\_ABC}]_i^{\text{🔀}} \\ \qquad \text{AND} \\ \left[\langle \mu\mathsf{RST} \rangle^{\boxplus 3}\right]_{i+1}^{\text{🔀}} \neq \left[\langle \mu\mathsf{RST} \rangle^{\boxplus 3}\right]_i^{\text{🔀}} \end{cases}$$

   THEN $[\mathsf{VAL\_ABC}]_{i+1}^{\text{🔀}} = [\mathsf{VAL\_ABC}_i^{\nu}]^{\text{🔀}}$

   (b) IF

   $$\begin{cases} [\mathsf{CN\_ABC}]_{i+1}^{\text{🔀}} \neq [\mathsf{CN\_ABC}]_i^{\text{🔀}} \\ \qquad \text{OR} \\ [\mathsf{INDEX\_ABC}]_{i+1}^{\text{🔀}} \neq [\mathsf{INDEX\_ABC}]_i^{\text{🔀}} \end{cases}$$

   THEN $[\mathsf{VAL\_ABC}]_{i+1}^{\text{🔀}} = 0$.

In other words after reordering $\mathsf{VAL\_ABC}$ and $\mathsf{VAL\_ABC}^{\nu}$ as explained above we have, for constant $[\mathsf{CN\_ABC}]^{\text{🔀}}$ and $[\mathsf{INDEX\_ABC}]^{\text{🔀}}$

	[CN_ABC]⤮	[INDEX_ABC]⤮	$[\langle\mu\text{RST}\rangle^{\boxplus 3}]$⤮	[VAL_ABC]⤮	$[\text{VAL\_ABC}^{\nu}]$⤮
0	0	?	28	?	?
1	0	?	55	?	?
2	0	?	117	?	?
⋮	⋮	⋮	⋮	⋮	⋮
i	c	k	12	**0**	♠
$i+1$	c	k	19	♠	⋆
$i+2$	c	k	20	⋆	□
$i+3$	c	k	38	□	♡
$i+4$	c'	l	23	**0**	†
$i+5$	c'	l	27	†	◊
$i+6$	c'	$l+1$	24	**0**	⋈
$i+7$	c'	$l+1$	27	⋈	‡
$i+8$	c'	$l+1$	33	‡	‡
$i+9$	c'	$l+1$	36	‡	♣
$i+10$	c'	$l+1$	37	♣	◎
$i+11$	c'	$l+2$	25	**0**	★
$i+12$	c'	$l+2$	26	★	▲
$i+13$	c'	$l+2$	27	▲	♯
$i+14$	c'	$l+2$	33	♯	♯
$i+15$	c'	$l+2$	43	♯	⊲
⋮	⋮	⋮	⋮	⋮	⋮

Figure 3.20: ⟨MICRO_RAM_STAMP⟩ $= 27$ appears thrice (as is to be expected) and the three rows in question $(i+5, i+7, i+13)$ the values are taken from the same execution context (c') in consecutive limbs $(l,\ l+1,\ l+2)$ and the values in RAM are changed($\dagger \rightsquigarrow \lozenge$, $\bowtie \rightsquigarrow \ddagger$ and$\blacktriangle \rightsquigarrow \sharp$). This is compatible with the 27th micro RAM operation being a non aligned `MSTORE` (in theory it could also be part of a non aligned `(EXT)CODECOPY`.) In a similar vein, note that the 33rd micro RAM operation (i.e. ⟨MICRO_RAM_STAMP⟩ $= 33$) touches two consecutive RAM locations $(l+2$ and $l+2)$ in that same execution context c' without modifying their values ($\ddagger \rightsquigarrow \ddagger$ and $\sharp \rightsquigarrow \sharp$). This could be part of an aligned or non aligned logging operation, an aligned or non aligned `MLOAD`, a successful `RETURN` in a deployment context (CTYPE $= 1$) among other options. (If we wanted to more information we would have to find what other context is activated at ⟨MICRO_RAM_STAMP⟩ $= 33$, or better yet: consult the non reordered exeuction trace).

Chapter 4

ROM

4.1 The ROM module

4.1.1 Introduction

The ROM contains the bytecodes of the contracts used within a batch of transaction as well as some associated metadata such as code size and code hash. Its main role in the overall design is to provide the Main Execution Trace with the correct sequence of instructions. Most of the arithmetization below focuses on building the ROM as a seqence of padded byte codes and of extracting the correct push values from it (i.e. the X-byte long arguments of actual `PUSH_X` instructions).

There are three kinds of accesses to bytecode that the ROM deals with, with contract deployment being subdivided into 1 or 2 phases (since deployments may fail):

1. loading auxiliary data associated to an address (i.e. its code hash (CH) and code size (CS)) for `EXTCODEHASH` and `EXTCODESIZE` instructions;
2. loading the full bytecode of an already deployed smart contract to run it or to `EXTCODECOPY` from it (or both);
3. deploying a smart contract through a transaction or `CREATE(2)`:
   (a) loading the init code into ROM;
   (b) for successful deployments loading the bytecode that will be deployed at the relevant address into ROM.

The `EXTCODEHASH` and `EXTCODESIZE` instructions force a slight technical difficulty upon us. `EXTCODEHASH` was added to the EVM instruction set in EIP 1052 to avoid costly `EXTCODECOPY`'s. Loading bytecodes into ROM to hash them once more would be contrary to its purpose of these *cheap* instructions. Since the Ethereum state is aware of code hashes this isn't too bad. However, the Ethereum state is unaware of a deployed bytecode's code size. Since `EXTCODESIZE` is a cheap instruction and hashing is expensive in the zk-EVM, we cannot justify a code's size by loading it into ROM, hashing it and comparing the result to the code hash. The $\mathsf{CODEHASH}$ and $\mathsf{CODESIZE}$ are thus verified against an auxiliary mapping `map[address](hash, int)` rather than against the state. This mapping must be updated with every successful contract creation.

4.1.2 ROM specific terms

We collect in this sections pointers to definitions of ROM specific terms: **counter-constant columns** are defined in section 4.1.4, **fully-counter-constant columns** are defined in section 4.1.4, **address-constant columns** are defined in section 4.1.4, **code-fragment-constant columns** are defined in section 4.1.4.

Note that by construction

$$\text{address-cnst.} \implies \text{code-fragment-cnst.} \implies \text{fully-counter-cnst.} \implies \text{counter-cnst.}$$

In essence: address-constant columns don't change until the address changes, (slightly simplifying) code-fragment constant columns can only change once per address, (slightly simplifying) fully-counter-constant columns can only change every 32 rows and (slightly simplifying) counter-constant columns can only change every 16 rows.

4.1.3 Trace columns

The first three columns differentiate between the three cases highlighted in the introduction according to the following table:

	LOAD	INIT	DH
1. loading CS and CH only	0	0	0
2. loading already deployed bytecode	1	0	0
3.a) init code	1	1	0
3.b) bytecode being deployed	1	0	1

1. IS_LOADED: a binary, address-constant column that distinguishes between code fragments that are being loaded to ROM in their entirety and code fragments of which we only import their code hash and code size into ROM; abbreviated to LOAD;

2. IS_INITCODE: a binary, code-fragment-constant colum; INIT $= 1$ for init code and INIT $= 0$ for all deployed (or about to be successfully deployed) code; abbreviated to INIT;

3. DO_HASH: a binary, code-fragment-constant column; equals 1 only for bytecode that's been successfully deployed within the current batch, thus indicating which bytecodes must be hashed; abbreviated to DH;

Hashing the bytecode should happen only *once*: when a contract is deployed for the first time and we need to insert its code hash into the state. Tagging these initial deployments is the purpose of DO_HASH.

The following columns are used for book-keeping of different code fragments and addresses within the ROM.

4. SC_ADDRESS_HIGH and SC_ADDRESS_LOW: address-constant columns; contain the high and low parts of the address associated with the bytecode currently being loaded into ROM; abbreviated to $\mathsf{ADDR}^{\mathsf{hi}}$ and $\mathsf{ADDR}^{\mathsf{lo}}$ respectively;

5. ADDRESS_INDEX: address-constant column; a column that starts at 0 and increases by 1 with every new address encountered in the ROM; abbreviated to AI;

6. CODE_FRAGMENT_INDEX: code-fragment-constant column; a refinement of ADDRESS_INDEX: increases by 1 whenever the address changes or the IS_INITCODE changes; abbreviated to CFI;

In other words, AI counts the number of different addresses in ROM while CFI counts the number of code fragments present in ROM (regardless of whether they are fully loaded into ROM or only their metadata is loaded in.) A given address in ROM can be associated to either 1 or 2 code fragments.

The following columns are for orientation within a given code fragment. They are used to construct $\mathsf{LACS}^{\mathsf{hi}}$ and $\mathsf{LACS}^{\mathsf{lo}}$ (see below) incrementally and to figure out at what point to switch from building $\mathsf{LACS}^{\mathsf{hi}}$ to bulding $\mathsf{LACS}^{\mathsf{lo}}$ and when to reset the process.

7. COUNTER: a periodic counter; if LOAD $= 0$ we have CT $= 0$; if LOAD $= 1$ it counts up from 0 to 15 in increments of 1 and resets; such “cycles” come in pairs (see CYC); abbreviated to CT;

8. CYCLIC_BIT: a counter-constant binary column; equals to 0 if LOAD $= 0$; otherwise it flips at the onset of every new COUNTER-cycle; abbreviated to CYC;

The following columns provide “meta data” associated with a bytecode: its length, its hash but also the big endian concatenation of (bytes from the padded bytecode) into EVM words (split into high and low parts): LACS$^{\text{hi}}$ and LACS$^{\text{lo}}$ respectively. These are computed for two reasons:

- for `(EXT)CODECOPY`’s it’s simpler to be able to pull left shifted prefixes of concatenations of bytes from the (padded) bytecode rather than individual bytes; this format is compatible with the Parent/Child architecture of RAM;
- for storing the bytecode (and thus easier retrieval later) it is simpler to store EVM words of (padded) bytecode (while remembering the length of the original bytecode, of course)

9. CODESIZE: a code-fragment-constant column containing the code size of the bytecode currently being loaded into ROM; abbreviated to CS;

10. CODEHASH_HIGH and CODEHASH_LOW: code-fragment-constant columns containing the (high and low part of the) code hash of the bytecode currently being loaded into ROM; abbreviated to CH$^{\text{hi}}$ and CH$^{\text{lo}}$ respectively;

11. LEFT_ALIGNED_CODESUFFIX_HIGH and LEFT_ALIGNED_CODESUFFIX_LOW: high and low part of the EVM word obtained by left-shifting (by CT $+ 16$CYC bytes) the concatenation of the opcodes in a full COUNTER-cycle’s worth of bytecode; see figure ?? for an explanation; abbreviated to LACS$^{\text{hi}}$ and LACS$^{\text{lo}}$ respectively;

The following columns relate to the `PUSH_X` instructions that require particular constraints to work properly.

12. IS_PUSH: instruction decoded binary flag column that lights up for push instructions; abbreviated to IP;

13. IS_PUSH_DATA: binary flag that lights up for the X rows following a `PUSH_X` instruction i.e. while PPO $\neq 0$; abbreviated to IPD; this flag selects those bytes from the bytecode that contribute to a push instruction’s PUSH_VALUE_HIGH or PUSH_VALUE_LOW; it also sets the OPCODE of said lines to `INVALID`; abbreviated to IPD;

14. PUSH_PARAMETER: instruction decoded column that contains X for `PUSH_X` instructions and 0 for non push instructions; abbreviated to PP;

15. PUSH_PARAMETER_OFFSET: following a `PUSH` instruction, this counts down from PP down to 0; abbreviated to PPO;

16. PUSH_VALUE_HIGH and PUSH_VALUE_LOW: high and low part of the value that a push instruction pushes on stack; abbreviated to PV$^{\text{hi}}$ and PV$^{\text{lo}}$ respectively;

17. PUSH_VALUE_ACC_HIGH and PUSH_VALUE_ACC_LOW: “accumulator” variables used to construct PUSH_VALUE_HIGH and PUSH_VALUE_LOW byte by byte out of “data carrying bytes”; abbreviated to PVA$^{\text{hi}}$ and PVA$^{\text{lo}}$;

18. PUSH_FUNNEL_BIT: a binary flag that matters for correctly contructing PUSH_VALUE_HIGH and PUSH_VALUE_LOW; abbreviated to PFB;

Let us say something about $\mathsf{PUSH\_FUNNEL\_BIT}$: this binary flag may switch from 1 to 0 when constructing a given `PUSH` instruction's $\mathsf{PV}^{\mathsf{hi}}$ and $\mathsf{PV}^{\mathsf{lo}}$; its value determines which accumulator ($\mathsf{PVA}^{\mathsf{hi}}$ or $\mathsf{PVA}^{\mathsf{lo}}$) a data carrying raw byte from the (padded) bytecode gets funneled to. If $\mathsf{PFB} = 1$, the byte contributes to $\mathsf{PVA}^{\mathsf{hi}}$, if $\mathsf{PFB} = 0$, the byte contributes to $\mathsf{PVA}^{\mathsf{lo}}$. To make this work we set $\mathsf{PFB} = 1$ at the onset of a push instruction with $\mathsf{PP} > 16$, it remains equal to 1 for the first $\mathsf{PP} - 16$ rows constructing the push value, and then switches to 0 for the 16 remaining rows. For a push instruction with $\mathsf{PP} \leq 16$, $\mathsf{PFB} = 0$ and all raw bytes are funneled to $\mathsf{PVA}^{\mathsf{lo}}$.

The columns below are related to the bytecode itself: the bytes that make it up, how to interpret them (i.e. do they code for instructions or are they data carriers for a `PUSH_X` instruction?), how much to pad with `0x00`'s etc…:

19. $\mathsf{PADDED\_BYTECODE\_BYTE}$: raw byte from the padded bytecode; if $\mathsf{LOAD} = 1$ code is being loaded into ROM; the PBCB column lists the bytes from said bytecode one by one as well as some extraneous `0x00`'s beyond the $\mathsf{CODESIZE}$ (padding); abbreviated to PBCB;

20. OPCODE: the opcode associated to the PBCB; depends on the the context i.e. on whether the byte is shadowed by a `PUSH` instruction (i.e. $\mathsf{IPD} = 1$) and whether the $\mathsf{CODESIZE\_REACHED}$ flag is on (at which point we impose $\mathsf{PBCB} = \mathsf{OPCODE} = \mathtt{0x00}$); in all other circumstances $\mathsf{OPCODE} = \mathsf{PADDED\_BYTECODE\_BYTE}$;

21. $\mathsf{PADDING\_BIT}$: a fully-counter-constant binary column; for code that is loaded into ROM this indicates the number of full-counter-cycles of padding with `0x00`'s to append after the bytecode proper; padding is done like so: the loaded bytecode is padded with zeros beyond its $\mathsf{CODESIZE}$ until we hit the first multiple of 32 (if $\mathsf{CODESIZE}$ is a clean multiple of 32 there is no such initial padding); it is then followed up by a full counter's worth of `0x00`'s (i.e. 32 extra rows of `0x00`'s); abbreviated to PAD;

22. PC: program counter (i.e. index of the byte in the current bytecode);

23. $\mathsf{CODESIZE\_REACHED}$: a binary column that equals 0 at the onset of a given bytecode and reaches 1 at the point where $\mathsf{PC}_i = \mathsf{CODESIZE}_i$; it resets to 0 at the onset of the next full $\mathsf{COUNTER}$-cycle; abbreviated to CSR;

24. $\mathsf{IS\_BYTECODE}$: a binary column that equals 1 for bytes that are part of the bytecode of the bytecode currently loaded into the ROM and 0 for bytes that are part of the padding that may be appended to the bytecode; abbreviated to IBC.

4.1.4 Constraints

Automatic constraints when $\mathsf{LOAD}_i = 0$

The condition $\mathsf{LOAD}_i = 0$ means that the current line is a simple import of previously commited values of the `CODESIZE` and `CODEHASH` of a smartcontract that doesn't get executed or `EXTCODECOPY`'d from.

1. IF $\mathsf{LOAD}_i = 0$ THEN

   (a) $\mathsf{CT}_i = 0$ AND $\mathsf{CT}_{i+1} = 0$

   (b) $\mathsf{CYC}_i = 0$ AND $\mathsf{CYC}_{i+1} = 0$

   (c) $\mathsf{CSR}_i = 0$ AND $\mathsf{CSR}_{i+1} = 0$

   (d) $\mathsf{INIT}_i = 0$

   (e) $\mathsf{AI}_{i+1} = 1 + \mathsf{AI}_i$

   (f) $\mathsf{PC}_i = 0$ AND $\mathsf{PC}_{i+1} = 0$

   (g) $\mathsf{LACS}^{\mathsf{hi}}_i = 0$ AND $\mathsf{LACS}^{\mathsf{lo}}_i = 0$

   (h) $\mathsf{OPCODE}_i = 0$

   (i) $\mathsf{PBCB}_i = 0$

   (j) $\mathsf{PADDING\_BIT}_i = 0$

The constraint $\mathsf{INIT}_i = 0$ signifies the fact that we may only import the digest of a smart contract into ROM if that smart contract already exists in the state, i.e. is deployed.

We also require that LOAD be automatically set to 0 whenever the CS is zero, i.e.

2. $\textsc{if}\ \mathsf{CODESIZE}_i = 0\ \textsc{then}\ \mathsf{LOAD}_i = 0$,

3. furthermore, we impose the value of the Hash in case the code size is 0

$$\textsc{if}\ \mathsf{CODESIZE}_i = 0\ \textsc{then}\ \begin{cases} \mathsf{CODEHASH\_HIGH}_i = \texttt{0xc5d2460186f7233c927e7db2dcc703c0} \\ \quad \textsc{and} \\ \mathsf{CODEHASH\_LOW}_i = \texttt{0xe500b653ca82273b7bfad8045d85a470} \end{cases}$$

4. and conversely

$$\textsc{if}\ \begin{cases} \mathsf{CODEHASH\_HIGH}_i = \texttt{0xc5d2460186f7233c927e7db2dcc703c0} \\ \quad \textsc{and} \\ \mathsf{CODEHASH\_LOW}_i = \texttt{0xe500b653ca82273b7bfad8045d85a470} \end{cases} \textsc{then}\ \mathsf{CODESIZE}_i = 0$$

Where `0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470` is the SHA3 hash of the empty list.

CT constraints

The $\mathsf{COUNTER}$ column imposes a "pulse" to the ROM.

1. $\mathsf{CT}_0 = 0$,

2. $\boxed{\textsc{if}\ \mathsf{LOAD}_i = 0\ \textsc{then}\ \big(\mathsf{CT}_i = 0\ \ \textsc{and}\ \mathsf{CT}_{i+1} = 0\big)}$

3. $\textsc{if}\ \mathsf{LOAD}_i = 1\ \textsc{then}$

   (a) $\textsc{if}\ \mathsf{CT}_i \neq 15\ \textsc{then}\ \mathsf{CT}_{i+1} = \mathsf{CT}_i + 1$,

   (b) $\textsc{if}\ \mathsf{CT}_i = 15\ \textsc{then}\ \mathsf{CT}_{i+1} = 0$,

4. $\textsc{if}\ \mathsf{LOAD}_{N-1} = 1\ \textsc{then}\ \mathsf{CT}_{N-1} = 15$.

These constraints impose that the $\mathsf{COUNTER}$ column is composed of 0's and strips where CT goes from 0 to 15 one step at a time. The constraints on CYC below will impose that such $\mathsf{COUNTER}$-cycles always appear in pairs.

A column X is **counter-constant** if it satisfies

$$\textsc{if}\ \Big(\mathsf{LOAD}_i = 1\ \ \textsc{and}\ \ \mathsf{CT}_i \neq 15\Big)\ \textsc{then}\ \ X_{i+1} = X_i$$

CYC constraints

The $\mathsf{CYCLIC\_BIT}$ column is a counter-constant binary column that oscilates from 0 to 1 and back between counter cycles.

1. CYC is a binary counter-constant column;

2. $\mathsf{CYC}_0 = 0$

3. $\boxed{\textsc{if}\ \mathsf{LOAD}_i = 0\ \textsc{then}\ \big(\mathsf{CYC}_i = 0\ \ \textsc{and}\ \mathsf{CYC}_{i+1} = 0\big)}$

4. $\textsc{if}\ \mathsf{CT}_i = 15\ \textsc{then}\ \mathsf{CYC}_{i+1} = 1 - \mathsf{CYC}_i$;

5. $\textsc{if}\ \mathsf{LOAD}_{N-1} = 1\ \textsc{then}\ \mathsf{CYC}_{N-1} = 1.$

Since $\mathsf{CYC} = 0$ if $\mathsf{LOAD} = 0$ while if $\mathsf{LOAD} = 1$, $\mathsf{CYC} = 0$ initially, CT starts out at 0, increments one by one until hitting 15, at which point CYC switches to 1. The preceding implies that at the reset of CT, LOAD must remain $= 1$. Thus counter cycles appear in (consecutive) pairs.

A column X is **fully-counter-constant** if it remains constant along such consecutive pairs of counter cycles, i.e. if it satisfies

$$\textsc{if}\ \Big(\mathsf{LOAD}_i = 1\ \ \textsc{and}\ \ \big(\mathsf{CYC}_i = 0\ \textsc{or}\ \ (\mathsf{CYC}_i = 1\ \ \textsc{and}\ \ \mathsf{CT}_i \neq 15)\big)\Big)\ \textsc{then}\ \ X_{i+1} = X_i$$

$\mathsf{ADDRESS\_INDEX}$ (and ADDR) constraints

$\mathsf{ADDRESS\_INDEX}$ counts the number of smart contract addresses in the ROM. As such it starts at 0 and increases by 1 with every new smart contract address.

1. $\mathsf{AI}_0 = 0$,

2. AI is fully-counter-constant,

3. $\boxed{\textsc{if}\ \mathsf{LOAD}_i = 0\ \textsc{then}\ \mathsf{AI}_{i+1} = 1 + \mathsf{AI}_i}$

4. AI can only jump by 1, i.e. $\mathsf{AI}_{i+1} \in \{\mathsf{AI}_i, \mathsf{AI}_i + 1\}$ i.e.

$$(\mathsf{AI}_{i+1} - \mathsf{AI}_i) \cdot (\mathsf{AI}_{i+1} - \mathsf{AI}_i - 1) = 0$$

5. AI changes *iff* $\mathsf{ADDR}^{\mathsf{hi}}$ or $\mathsf{ADDR}^{\mathsf{lo}}$ changes, i.e.

$$\textsc{if}\ \begin{cases} \mathsf{ADDR}^{\mathsf{hi}}_{i+1} = \mathsf{ADDR}^{\mathsf{hi}}_i \\ \quad \textsc{and} \\ \mathsf{ADDR}^{\mathsf{lo}}_{i+1} = \mathsf{ADDR}^{\mathsf{lo}}_i \end{cases} \qquad \textsc{then}\ \ \mathsf{AI}_{i+1} = \mathsf{AI}_i$$

and similarly

$$\textsc{if}\ \mathsf{AI}_{i+1} = \mathsf{AI}_i \quad \textsc{then}\ \begin{cases} \mathsf{ADDR}^{\mathsf{hi}}_{i+1} = \mathsf{ADDR}^{\mathsf{hi}}_i \\ \quad \textsc{and} \\ \mathsf{ADDR}^{\mathsf{lo}}_{i+1} = \mathsf{ADDR}^{\mathsf{lo}}_i \end{cases}$$

Given that $\mathsf{ADDRESS\_INDEX}$ and the address change at the same times, we say that a column X is **address-constant** if it satisfies

$$\textsc{if}\ \mathsf{AI}_i = \mathsf{AI}_{i+1}\ \textsc{then}\ X_i = X_{i+1}.$$

INIT and CFI constraints

The purpose of the $\mathsf{IS\_INITCODE}$ is the differentiate between initialization code and deployed code: it equals 1 for initialization code and 0 for deployed code. When contract is deployed its initialization code is loaded into ROM. If that deployment is successful the deployed bytecode gets loaded into ROM, too.

1. $\boxed{\mathsf{INIT}\ \text{is a fully-counter-constant binary column}}$,

2. $\boxed{\textsc{if}\ \mathsf{LOAD}_i = 0\ \textsc{then}\ \mathsf{INIT}_i = 0}$,

3. $\textsc{if}\ \Big(\mathsf{AI}_i = \mathsf{AI}_{i+1}\ \ \textsc{and}\ \mathsf{INIT}_i \neq \mathsf{INIT}_{i+1}\Big)\ \textsc{then}\ \mathsf{INIT}_i = 1$ and $\mathsf{INIT}_{i+1} = 0.$

In other words: INIT can only change at the end of full counter cycles and can only change once for a given address, necessarily going from 1 to 0.

$\mathsf{CODE\_FRAGMENT\_INDEX}$ counts the code fragments present in ROM. As such it changes every time the address changes (i.e. every time AI changes) and every time $\mathsf{IS\_INITCODE}$ changes:

1. $\mathsf{CFI}_0 = 0$;
2. $\text{IF } \big(\mathsf{AI}_{i+1} = \mathsf{AI}_i \ \text{ AND } \mathsf{INIT}_{i+1} = \mathsf{INIT}_i\big) \text{ THEN } \mathsf{CFI}_{i+1} = \mathsf{CFI}_i$;
3. $\text{ELSE } \mathsf{CFI}_{i+1} = 1 + \mathsf{CFI}_i$

We say that X is **code-fragment-constant** if it satisfies

$$\text{IF } \big(\mathsf{CFI}_{i+1} = \mathsf{CFI}_i\big) \text{ THEN } X_i = X_{i+1},$$

Thus a column that is address-constant is code-fragment-constant. However, a single address in ROM corresponds to 1 or 2 code fragments. There are 2 code fragments for a single address *iff* the batch contains a succesful contract deployment to said address. In that case INIT switches (necessarily from 1 to 0). The first code fragment (with $\mathsf{INIT} \equiv 1$) corresponds to init code and the second code fragment (with $\mathsf{INIT} \equiv 0$) corresponds to the bytecode that was successfully deployed.

LOAD and INIT constraints

Recall that $\mathsf{LOAD} = 0$ *iff* we are loading the codehash and codesize *only*. In all other cases $\mathsf{LOAD} = 1$. This flag is address-constant (and thus fully-counter-constant). The INIT binary flag has $\mathsf{INIT} = 1$ *iff* the current code fragment is init code. This flag is code-fragment constant (and thus) fully-counter-constant.)

1. LOAD is an address-constant binary column;
2. INIT is a code-fragment-constant binary column;
3. **Exit constraints:**

$$\text{IF } \mathsf{LOAD}_{N-1} = 1 \text{ THEN } \begin{cases} & \mathsf{CT}_{N-1} = 15 \\ \text{AND} & \mathsf{CYC}_{N-1} = 1 \\ \text{AND} & \mathsf{PADDING\_BIT}_{N-1} = 0 \end{cases}$$

4. We include here a constraint saying that within the $\mathsf{IS\_INITCODE}$ can only jump from 1 to 0 within a given address (and not from 0 to 1.)

$$\text{IF } \begin{cases} & \mathsf{CT}_i = 15 \\ \text{AND} & \mathsf{CYC}_i = 1 \\ \text{AND} & \mathsf{AI}_{i+1} = \mathsf{AI}_i \end{cases} \quad \text{THEN} \quad \mathsf{INIT}_{i+1} \in \big\{\mathsf{INIT}_i, \mathsf{INIT}_i - 1\big\}$$

(This constraint is arithmetized by $(\mathsf{INIT}_{i+1} - \mathsf{INIT}_i) \cdot (\mathsf{INIT}_{i+1} - \mathsf{INIT}_i - 1) = 0$.) Since INIT is binary this means that under the previous circumstances INIT_i either remains constant for a given value of the address, or jumps once from 1 to 0.

Note. As will be explained below, $\mathsf{ADDRESS\_INDEX}$ changes precisely when ADDR changes. The condition $\mathsf{AI}_{i+1} = \mathsf{AI}_i$ thus simply means that the address didn't change from line i to line $i + 1$.

CT	CYC	PBCB	$\mathsf{LACS}^{\mathsf{hi}}$	$\mathsf{LACS}^{\mathsf{lo}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	0	b_0	0x $b_0\ b_1\ b_2\ \cdots b_{14} b_{15}$	0x $b_{16} b_{17} b_{18} \cdots b_{30} b_{31}$
1	0	b_1	0x $b_1\ b_2\ b_3\ \cdots b_{15} b_{16}$	0x $b_{17} b_{18} b_{19} \cdots b_{31}\ 0$
2	0	b_2	0x $b_2\ b_3\ b_4\ \cdots b_{16} b_{17}$	0x $b_{18} b_{19} b_{20} \cdots\ 0\ \ 0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
14	0	b_{14}	0x $b_{14} b_{15} b_{16} \cdots b_{28} b_{29}$	0x $b_{30}\ b_{31}\ 0 \cdots\ 0\ \ 0$
15	0	b_{15}	0x $b_{15} b_{16} b_{17} \cdots b_{29} b_{30}$	0x $b_{31}\ \ 0\ \ 0 \cdots\ 0\ \ 0$
0	1	b_{16}	0x $b_{16} b_{17} b_{18} \cdots b_{30} b_{31}$	0x $0\ \ 0\ \ 0 \cdots\ 0\ \ 0$
1	1	b_{17}	0x $b_{17} b_{18} b_{19} \cdots b_{31}\ 0$	0x $0\ \ 0\ \ 0 \cdots\ 0\ \ 0$
2	1	b_{18}	0x $b_{18} b_{19} b_{20} \cdots\ 0\ \ 0$	0x $0\ \ 0\ \ 0 \cdots\ 0\ \ 0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
14	1	b_{20}	0x $b_{30}\ b_{31}\ 0 \cdots\ 0\ \ 0$	0x $0\ \ 0\ \ 0 \cdots\ 0\ \ 0$
15	1	b_{31}	0x $b_{31}\ \ 0\ \ 0 \cdots\ 0\ \ 0$	0x $0\ \ 0\ \ 0 \cdots\ 0\ \ 0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 4.1: A full COUNTER-cycle's worth of LEFT_ALIGNED_CODESUFFIX_HIGH and LEFT_ALIGNED_CODESUFFIX_LOW.

Left shifted suffix constraints

The $\mathsf{LACS}^{\mathsf{hi}}$ and $\mathsf{LACS}^{\mathsf{lo}}$ columns are useful for dealing with `CODECOPY` and `EXTCODECOPY` instructions. Since these operations require one to load the bytecode into ROM we require

$$\boxed{\text{IF } \mathsf{LOAD}_i = 0 \text{ THEN } \left(\mathsf{LACS}^{\mathsf{hi}}_i = 0 \ \text{ AND } \ \mathsf{LACS}^{\mathsf{lo}}_i = 0\right).}$$

The expected behaviour of these columns is represented in the table below: Bytecode is listed byte by byte. The bytecode of a given smartcontract (or init code) that finds itself in the ROM is comprised of consecutive full COUNTER-cycles of bytecode. This implies that bytecode may be padded with 0's to make its length a clean multiple of 32. The padding (if any) affects neither the CODESIZE nor the CODEHASH thanks to a binary column indicating padding rows. In other words: When we load code from some contract's bytecode using either `CODECOPY` or `EXTCODECOPY`, we first import a left shifted suffix of the beginning portion of the bytecode to copy. The remainder of the words that are copied from ROM to RAM are full words constituted of the bytes of a full-counter-cycle's worth of bytes. For both the special first "partial" import and the subsequent "full" imports we use these left shifted suffix columns.

The constraints below apply when $\mathsf{LOAD}_i = 1$

$\mathsf{LACS}^{\mathsf{lo}}$ constraints. —

1. $\text{IF } \mathsf{CYC}_i = 0 \text{ THEN } \mathsf{LACS}^{\mathsf{lo}}_{i+1} = 256 \cdot \left(\mathsf{LACS}^{\mathsf{lo}}_i - 256^{15} \cdot \mathsf{PBCB}_{i+16}\right)$
2. $\text{IF } \mathsf{CYC}_i = 1 \text{ THEN } \mathsf{LACS}^{\mathsf{lo}}_i = 0$

$\mathsf{LACS}^{\mathsf{hi}}$ constraints. —

1. $\text{IF } \mathsf{CYC}_i = 0 \text{ THEN } \mathsf{LACS}^{\mathsf{hi}}_{i+1} - \mathsf{PBCB}_{i+17} = 256 \cdot \left(\mathsf{LACS}^{\mathsf{hi}}_i - 256^{15} \cdot \mathsf{PBCB}_i\right)$
2. $\text{IF } \mathsf{CYC}_i = 1 \text{ THEN } \mathsf{LACS}^{\mathsf{hi}}_i = \mathsf{LACS}^{\mathsf{lo}}_{i-16}$

CSR and PC constraints

The program counter PC is local to a code fragment; it starts at 0 and increases by 1 with every byte in the code fragment (i.e. padded bytecode):

1. $\mathsf{PC}_0 = 0$;
2. $\textsc{if}\ \mathsf{CFI}_{i+1} = \mathsf{CFI}_i\ \textsc{then}\ \mathsf{PC}_{i+1} = 1 + \mathsf{PC}_i$;
3. $\textsc{if}\ \mathsf{CFI}_{i+1} \neq \mathsf{CFI}_i\ \textsc{then}\ \mathsf{PC}_{i+1} = 0$.

$\mathsf{CODESIZE\_REACHED}$ equals 0 while the PC hasn't reached $\mathsf{CODESIZE}$, at which point it equals 1 until the end of the current CFI.

1. CSR is a binary column;
2. $\mathsf{CSR}_0 = 0$;
3. transition:
   (a) $\textsc{if}\ \left(\mathsf{CFI}_{i+1} = \mathsf{CFI}_i\right)\ \textsc{then}$

   $$(\mathsf{CSR}_{i+1} - \mathsf{CSR}_i) \cdot (\mathsf{CSR}_{i+1} - \mathsf{CSR}_i - 1) = 0$$

   i.e. $\mathsf{CSR}_{i+1} \in \left\{\mathsf{CSR}_i, \mathsf{CSR}_i + 1\right\}$ $\textsc{and}$

   $$\begin{cases} \textsc{if} & 1 + \mathsf{PC}_{i+1} = \mathsf{CS}_{i+1}\ \textsc{then}\ \ \mathsf{CSR}_{i+1} = \mathsf{CSR}_i + 1 \\ \textsc{if} & 1 + \mathsf{PC}_{i+1} \neq \mathsf{CS}_{i+1}\ \textsc{then}\ \ \mathsf{CSR}_{i+1} = \mathsf{CSR}_i \end{cases}$$

   (PC starts at 0 while CS starts at 1)

   (b) $\textsc{if}\ \left(\mathsf{CFI}_{i+1} \neq \mathsf{CFI}_i\right)\ \textsc{then}$
      i. $\textsc{if}\ \mathsf{LOAD}_{i+1} = 0\ \textsc{then}\ \mathsf{CSR}_{i+1} = 1$
      ii. $\textsc{if}\ \mathsf{LOAD}_{i+1} = 1\ \textsc{then}$
         A. $\mathsf{CS}_{i+1} \neq 0$
         B. $\textsc{if}\ \mathsf{CS}_{i+1} = 1\ \textsc{then}\ \mathsf{CSR}_{i+1} = 1$
         C. $\textsc{if}\ \mathsf{CS}_{i+1} \neq 1\ \textsc{then}\ \mathsf{CSR}_{i+1} = 0$
4. termination: $\textsc{if}\ \mathsf{LOAD}_{N-1} = 1\ \textsc{then}\ \mathsf{CSR}_{N-1} = 1$.

$\mathsf{PADDING\_BIT}$ constraints

The $\mathsf{PADDING\_BIT}$ column counts the remaining full $\mathsf{COUNTER}$-cycles of zero paddings.

1. PAD is binary and fully-$\mathsf{COUNTER}$-constant;
2. $\mathsf{PAD}_0 = \mathsf{LOAD}_0$
3. $\textsc{if}\ \mathsf{CFI}_{i+1} \neq \mathsf{CFI}_i\ \textsc{then}\ \mathsf{PAD}_{i+1} = \mathsf{LOAD}_{i+1}$
4. $\textsc{if}\ \left(\mathsf{CT}_i = 15\ \ \textsc{and}\ \mathsf{CYC}_i = 1\right)\ \textsc{then}$
   (a) $\textsc{if}\ \mathsf{CSR}_i = 0\ \textsc{then}\ \mathsf{PAD}_{i+1} = 1$;
   (b) $\textsc{if}\ \mathsf{CSR}_i = 1\ \ \textsc{and}\ \mathsf{PAD}_i = 1\ \textsc{then}\ \mathsf{PAD}_{i+1} = 0$;
   (c) $\textsc{if}\ \mathsf{PAD}_i = 0\ \textsc{then}\ \mathsf{CFI}_{i+1} = 1 + \mathsf{CFI}_i$

5. $\mathsf{PAD}_{N-1} = 0$.

At the beginning of every code fragment PAD is initialized to LOAD (i.e. to 1 if we are loading code and to 0 otherwise). If we are loading code into ROM, PAD remains constant equal to 1 for all full-COUNTER-cycles as long as CODESIZE wasn't reached. The first full-COUNTER-cycle that starts with the CODESIZE_REACHED flag set to 1 starts by decrementing PAD to 0. This is the (full) padding cycle; it ends with an imposed increment in CODE_FRAGMENT_INDEX.

This padding and these lengths of padding were chosen so that

1. there are always enough zeros following the end of the bytecode proper to construct correct push values (if needed);
2. constructing push values doesn't encroach on bytes from the next code fragment;
3. the PC column extends far enough so that $\mathsf{PC}_i + \mathsf{PP}_i + 1$ is in the PC range of the current code fragment and can (if need by) be imported in the MET directly from ROM.

It can happen that the program counter is set to a value much larger than $\mathsf{PC}_i + 1 + 1, \mathsf{PC}_i + 2 + 1, \ldots, \mathsf{PC}_i + 32 + 1$, e.g. if the program jumps to large value. These large jumps are recognized as such by the Main Execution Trace (by comparing the next PC to the code size).

IBC constraints

IS_BYTECODE equals 1 for bytes that are part of the bytecode and 0 for any bytes that are padding. IBC is essentially a shifted version of $1 - \mathsf{CSR}$.

1. IBC is a binary column;
2. initialization: $\mathsf{IBC}_0 = \mathsf{LOAD}_0$;
3. $\textsc{if}$ $\mathsf{CFI}_{i+1} \neq \mathsf{CFI}_i$ $\textsc{then}$ $\mathsf{IBC}_{i+1} = \mathsf{LOAD}_{i+1}$;
4. $\textsc{if}$ $\mathsf{CFI}_{i+1} = \mathsf{CFI}_i$ $\textsc{then}$ $\mathsf{IBC}_{i+1} = 1 - \mathsf{CSR}_i$;
5. Due to padding IBC always terminates with 0:
$$\mathsf{IBC}_{N-1} = 0.$$
6. $\textsc{if}$ $\mathsf{IBC}_i = 0$ $\textsc{then}$ $\mathsf{OPCODE}_i = \mathsf{PBCB}_i = 0$

PUSH_FUNNEL_BIT constraints

1. PFB is a binary column,
2. $\textsc{if}$ $\mathsf{IPD}_i = 0$ $\textsc{then}$ $\mathsf{PFB}_i = 0$
3. $\textsc{if}$ $\mathsf{PPO}_i = 16$ $\textsc{then}$ $\mathsf{PFB}_{i+1} = -1 + \mathsf{PFB}_i$
4. $\textsc{if}$ $\left(\mathsf{IPD}_i = 1 \ \textsc{and}\ \mathsf{PPO}_i = 0\right)$ $\textsc{then}$ $\mathsf{PFB}_i = 0$

In other words, PFB is a binary column that can only be $= 1$ for rows containing data carrying bytes. If PPO passes the threshold of 16 PFB is decremented; also when the push parameter is done being built PFB must be 0. Therefore, when building the push parameter PFB will start at 1 and go to 0 if $\mathsf{PP} > 16$, otherwise $\mathsf{PFB} = 0$ throughout.

4.1.5 Constraints related to `PUSH` instructions

The constraints below construct $\mathsf{PUSH\_VALUE}$ for `PUSH` instructions.

1. IF $\mathsf{CFI}_{i+1} = \mathsf{CFI}_i$ i.e. we remain within the same code fragment:

   (a) Increment the program counter: $\mathsf{PC}_{i+1} = \mathsf{PC}_i + 1$

   (b) IF $\mathsf{IS\_PUSH\_DATA}_i = 0$, i.e. the current byte is not one that contributes to the value pushed onto stack by a `PUSH` instruction:

      i. $\mathsf{OPCODE}_i = \mathsf{PBCB}_i$;

      ii. IF $\mathsf{IS\_PUSH\_INSTRUCTION}_i = 1$ i.e. we are reading a push instruction (say `PUSH_X`) and the next X lines are data carrying bytes that will be aggregated into (the high and low part of) the value to push on stack

         A. set $\mathsf{IS\_PUSH\_DATA}$ for the next instruction:

$$\mathsf{IS\_PUSH\_DATA}_{i+1} = 1$$

         B. $\mathsf{PUSH\_VALUE}$ remains constant:

$$\begin{cases} \mathsf{PUSH\_VALUE}^{\mathsf{hi}}_{i+1} = \mathsf{PUSH\_VALUE}^{\mathsf{hi}}_{i} \\ \mathsf{PUSH\_VALUE}^{\mathsf{lo}}_{i+1} = \mathsf{PUSH\_VALUE}^{\mathsf{lo}}_{i} \end{cases}$$

         C. initialize the `PUSH` value accumulators:

$$\begin{cases} \mathsf{PUSH\_VALUE\_ACC}^{\mathsf{hi}}_{i} = 0 \\ \mathsf{PUSH\_VALUE\_ACC}^{\mathsf{lo}}_{i} = 0 \end{cases}$$

         D. Set the $\mathsf{PUSH\_PARAMETER\_OFFSET}$ to the $\mathsf{PUSH\_PARAMETER}$:

$$\begin{cases} \mathsf{PUSH\_PARAMETER\_OFFSET}_i & = & \mathsf{PUSH\_PARAMETER}_i \\ \mathsf{PUSH\_PARAMETER\_OFFSET}_{i+1} & = & -1 + \mathsf{PUSH\_PARAMETER}_i \end{cases}$$

      iii. ELSEIF $\mathsf{IS\_PUSH\_INSTRUCTION}_i = 0$ i.e. the current instruction is not a PUSH THEN

$$\begin{cases} \mathsf{PUSH\_VALUE\_HIGH}_i = 0 \\ \mathsf{PUSH\_VALUE\_LOW}_i = 0 \\ \mathsf{PUSH\_VALUE\_ACC\_HIGH}_i = 0 \\ \mathsf{PUSH\_VALUE\_ACC\_LOW}_i = 0 \\ \mathsf{PUSH\_PARAMETER\_OFFSET}_i = 0 \\ \mathsf{IS\_PUSH\_DATA}_{i+1} = 0 \end{cases}$$

      In other words: there is nothing to push (and we don't need to construct the push value) and the following isn't a data carrying byte, i.e. a byte claimed by a PUSH instruction.

   (c) ELSEIF $\mathsf{IS\_PUSH\_DATA}_i = 1$ i.e. the current byte is a data carrying byte and contributes to a push value:

      i. $\mathsf{OPCODE}_i = \texttt{INVALID}$;

      ii. Compute the current push value:

$$\begin{cases} \text{IF } \mathsf{PFB}_i = 1 & \begin{cases} \mathsf{PVA}^{\mathsf{hi}}_i = 256 \cdot \mathsf{PVA}^{\mathsf{hi}}_{i-1} + \mathsf{PBCB}_i \\ \mathsf{PVA}^{\mathsf{lo}}_i = \mathsf{PVA}^{\mathsf{lo}}_{i-1} \end{cases} \\ \text{IF } \mathsf{PFB}_i = 0 & \begin{cases} \mathsf{PVA}^{\mathsf{hi}}_i = \mathsf{PVA}^{\mathsf{hi}}_{i-1} \\ \mathsf{PVA}^{\mathsf{lo}}_i = 256 \cdot \mathsf{PVA}^{\mathsf{lo}}_{i-1} + \mathsf{PBCB}_i \end{cases} \end{cases}$$

iii. IF $\mathsf{PUSH\_PARAMETER\_OFFSET}_i \not= \mathbf{0}$: there are still other elements to push on the stack:

A. decrement PPO: $\mathsf{PPO}_{i+1} = \mathsf{PPO}_i - 1$;

B. the next byte is still a data carrying one: $\mathsf{IPD}_{i+1} = 1$;

C. $\mathsf{PUSH\_VALUE}$ remains constant:

$$\begin{cases} \mathsf{PV}^{\mathsf{hi}}_{i+1} = \mathsf{PV}^{\mathsf{hi}}_{i}, \\ \mathsf{PV}^{\mathsf{lo}}_{i+1} = \mathsf{PV}^{\mathsf{lo}}_{i}. \end{cases}$$

iv. ELSEIF $\mathsf{PUSH\_PARAMETER\_OFFSET}_i = 0$: the current byte from the (padded) bytecode is the *final* byte contributing to the value that the push instruction may put on stack.

A. Unset the PUSH parameter flag $\mathsf{IPD}_{i+1} = 0$.

B. Confirm the $\mathsf{PUSH\_VALUE}$:

$$\begin{cases} \mathsf{PV}^{\mathsf{hi}}_{i} = \mathsf{PVA}^{\mathsf{hi}}_{i} \\ \mathsf{PV}^{\mathsf{lo}}_{i} = \mathsf{PVA}^{\mathsf{lo}}_{i} \end{cases}$$

C. The next value of the PUSH parameter is enforced by 1(b)iii and 1(b)ii.

2. IF $\mathsf{CFI}_{i+1} \neq \mathsf{CFI}_i$ THEN

(a) $\mathsf{IPD}_{i+1} = 0$, i.e. the next bytecode can't start "mid-`PUSH`",

(b) $\mathsf{OPCODE}_{i+1} = \mathsf{PBCB}_{i+1}$;

INST	PBCB	IPD	PP	PPO	PFB	$\mathsf{PVA}^{\mathsf{hi}}$	$\mathsf{PVA}^{\mathsf{lo}}$	$\mathsf{PV}^{\mathsf{hi}}$	$\mathsf{PV}^{\mathsf{lo}}$
⋮ ?	⋮ ?	⋮ ?	⋮ ?	0	0	⋮ ?	⋮ ?	⋮ ?	⋮ ?
PUSH18	0x 71	0	18	18	0	0x 0	0x 0	0x ab	0x cd ··· qr
INV.	a	1	18	17	1	0x a	0x 0	0x ab	0x cd ··· qr
INV.	b	1	18	16	1	0x ab	0x 0	0x ab	0x cd ··· qr
INV.	c	1	18	15	0	0x ab	0x c	0x ab	0x cd ··· qr
INV.	d	1	18	14	0	0x ab	0x cd	0x ab	0x cd ··· qr
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
INV.	q	1	18	1	0	0x ab	0x cd ··· q	0x ab	0x cd ··· qr
INV.	r	1	18	0	0	0x ab	0x cd ··· qr	0x ab	0x cd ··· qr
? ⋮	? ⋮	0	? ⋮	? ⋮	0	? ⋮	? ⋮	? ⋮	? ⋮

Figure 4.2: The set-up in action for a `PUSH18` instruction.

4.1.6 Contract Address comparisons

Smart contract addresses are to be listed in ascending order. A given smart contract address may be associated with both an initcode and a deployed bytecode. We ask that in this case the deployed bytecode come after the initcode. In other words we ask that the rows be ordered according to lexicographic order on

$$(\mathsf{ADDR}^{\mathsf{hi}}, \mathsf{ADDR}^{\mathsf{lo}}, \mathsf{INIT}, \mathsf{PC})$$

In other words:

INST	PBCB	IPD	PP	PPO	PFB	PVA$^{\text{hi}}$	PVA$^{\text{lo}}$	PV$^{\text{hi}}$	PV$^{\text{lo}}$
⋮ ?	⋮ ?	⋮ ?	⋮ ?	0	0	⋮ ?	⋮ ?	⋮ ?	⋮ ?
PUSH4	0x 63	0	4	4	0	0x 0	0x 0	0x 0	0x abcd
INV.	a	1	4	3	0	0x 0	0x a	0x 0	0x abcd
INV.	b	1	4	2	0	0x 0	0x ab	0x 0	0x abcd
INV.	c	1	4	1	0	0x 0	0x abc	0x 0	0x abcd
INV.	d	1	4	0	0	0x 0	0x abcd	0x 0	0x abcd
? ⋮	? ⋮	0	? ⋮	? ⋮	0	? ⋮	? ⋮	? ⋮	? ⋮

Figure 4.3: The set-up in action for a `PUSH4` instruction.

1. We first order by address,
2. within a given address we first list (if present) the init code followed (if present) by the byte code put on chain,
3. within such a code fragment we order by program counter.

The Word Comparison module imports the columns (ADDRESS_INDEX, ADDR$^{\text{hi}}$, ADDR$^{\text{lo}}$) from the ROM. We check this ordering in Word Comparion module. Within a given address we first list code fragments with INIT $= 1$ followed by code fragments with INIT $= 0$, and within code fragments list lines with PC in ascending order. Both of these constraints are enforced in the ROM.

Chapter 5

Out of bounds

5.1 Columns

5.1.1 Purpose

The present document is a revised and expanded version of a previous (partial) specification of a zk-evm.

5.1.2 Column descriptions

We start out by listing the imported columns

1. $\langle\mathsf{OOB}\,\square\rangle$: imported column containing the module time stamp; counts the rare checks;
2. $\langle\mathsf{REFS}\rangle$: imported column containing the relevant (i.e. *instruction dependent*) "reference size";
3. $\langle\mathsf{OFF}^{\mathsf{hi}}\rangle$ and $\langle\mathsf{OFF}^{\mathsf{lo}}\rangle$: imported columns containing the high and low part respectively of an "offset" column;
4. $\langle\mathsf{SIZE}^{\mathsf{hi}}\rangle$ and $\langle\mathsf{SIZE}^{\mathsf{lo}}\rangle$: imported columns containing the high and low part respectively of a "size" column;
5. $\langle\mathsf{JOOB}\rangle$: imported binary column containing the jump out of bounds flag;
6. $\langle\mathsf{CDL\_OOB}\rangle$: imported binary column containing the call data load out of bounds flag;
7. $\langle\mathsf{RETDCX}\rangle$: imported binary column containing the return data copy exception flag;
8. $\langle\mathsf{MAXCSX}\rangle$: imported binary column containing the max code size exception flag;

Note that the $\langle\mathsf{JOOB}\rangle$, $\langle\mathsf{RETDCX}\rangle$, $\langle\mathsf{MAXCSX}\rangle$ flags are importe from the MET where they seemingly appear "out of thin air". They will be justified in the present module. We also import some decoded columns (out of sheer convenience)

9. $\langle {}^{\diamond}\mathsf{JUMP}\,\text{⚐}\rangle$, $\langle {}^{\diamond}\mathsf{RDC}\,\text{⚐}\rangle$, $\langle {}^{\diamond}\mathsf{CDL}\,\text{⚐}\rangle$, $\langle {}^{\diamond}\mathsf{RETURN}\,\text{⚐}\rangle$: imported binary flags that record which instruction the rare checks module is dealing with;

We introduce the module specific columns

10. $\mathsf{RIDICULOUSLY\_OOB}$: binary, $\langle\mathsf{OOB}\,\square\rangle$-constant column; indicates whether relevant offsets are ridiculously out of bounds, by which we mean that relevant "high parts" are nonzero; abbreviated to ROOB;

11. $\textsf{OOB}$: binary, $\langle\textsf{OOB}\square\rangle$-constant column; indicates whether relevant offsets are out of bounds (but not necessarily ridiculously so);
12. $\textsf{COUNTER}$: counter column; counts up starting at 0 to either 0, **3** or **16**;
13. $\textsf{BYTE\_1}$: byte column;
14. $\textsf{ACC\_1}$: accumulator column; accumulates the bytes from the previous column;

5.2 Heartbeat

The heartbeat is simple and resembles that of other modules. We define a column $\textsf{X}$ to be $\langle\textsf{OOB}\square\rangle$-constant if it satisfies

$$\forall i,\ \langle\textsf{OOB}\square\rangle_{i+1} = \langle\textsf{OOB}\square\rangle_i \implies \textsf{X}_{i+1} = \textsf{X}_i$$

The construction of the $\textsf{OOB}\square$ stamp in the MET and the selection process for the rows (with potentially nonzero values) which the present module imports enforce that *all imported columns of the present module are automatically* $\langle\textsf{OOB}\square\rangle$*-constant.* We further ask that the following columns be $\langle\textsf{OOB}\square\rangle$-constant:

1. $\textsf{OOB}$
2. $\textsf{ROOB}$

What follows are the heartbeat constraints.

1. $\langle\textsf{OOB}\square\rangle_0 = 0$
2. IF $\langle\textsf{OOB}\square\rangle_i = 0$ THEN the entire i^{th} row vanishes;
3. $\forall i,\ \langle\textsf{OOB}\square\rangle_{i+1} \in \{\langle\textsf{OOB}\square\rangle_i, 1 + \langle\textsf{OOB}\square\rangle_i\}$;

In other words, $\langle\textsf{OOB}\square\rangle$ either remains constant or increases one by one.

4. IF $\langle\textsf{OOB}\square\rangle_i \neq 0$ THEN
   (a) IF $\langle\textsf{OOB}\square\rangle_i \neq \langle\textsf{OOB}\square\rangle_{i-1}$ THEN $\textsf{CT}_i = 0$
   (b) IF $\textsf{ROOB}_i = 1$ THEN $\langle\textsf{OOB}\square\rangle_{i+1} = 1 + \langle\textsf{OOB}\square\rangle_i$.
   (c) IF $\textsf{ROOB}_i = 0$ AND $\textsf{OOB}_i = 1$ THEN
      i. IF $\textsf{CT}_i \neq \mathbf{16}$ THEN
      $$\begin{cases} \langle\textsf{OOB}\square\rangle_{i+1} = \langle\textsf{OOB}\square\rangle_i \\ \textsf{CT}_{i+1} = 1 + \textsf{CT}_i \end{cases}$$
      ii. IF $\textsf{CT}_i = \mathbf{16}$ THEN
      $$\begin{cases} \langle\textsf{OOB}\square\rangle_{i+1} = 1 + \langle\textsf{OOB}\square\rangle_i \\ \textsf{CT}_{i+1} = 0 \end{cases}$$
   (d) IF $\textsf{ROOB}_i = 0$ AND $\textsf{OOB}_i = 0$ THEN
      i. IF $\textsf{CT}_i \neq \mathbf{3}$ THEN
      $$\begin{cases} \langle\textsf{OOB}\square\rangle_{i+1} = \langle\textsf{OOB}\square\rangle_i \\ \textsf{CT}_{i+1} = 1 + \textsf{CT}_i \end{cases}$$
      ii. IF $\textsf{CT}_i = \mathbf{3}$ THEN
      $$\begin{cases} \langle\textsf{OOB}\square\rangle_{i+1} = 1 + \langle\textsf{OOB}\square\rangle_i \\ \textsf{CT}_{i+1} = 0 \end{cases}$$

In other words there are three scenarios from the point of view of the heartbeat:

1. relevant offsets may be ridiculously out of bounds, i.e. $\mathsf{ROOB} = 1$: in this case the present module knows how to produce the correct result in a single line of execution trace;
2. relevant offsets aren't ridiculously out of bounds, i.e. $\mathsf{ROOB} = 0$:
   (a) offsets may still be out of bounds ($\mathsf{OOB}_i = 1$) but establishing this may require up to **17** many rows;
   (b) offsets are within bounds ($\mathsf{OOB}_i = 0$) and establishing this requires up to **4** many rows;

Note that we chose **4** (and thus $\mathbf{3} = \mathbf{4} - 1$ in the heartbeat) as a cut-off point due to the fact that "realistic" call data sizes and return data sizes are **4**-byte integers. We give more details as to this decision in the memory expansion module. Note that in the "$\mathsf{ROOB} = 0$, $\mathsf{OOB}_i = 1$" case we find ourselves (like in the Memory Expansion Module) working on *sums* of limbs (minus another "small" limb), i.e. integers that (may) require $1 + 8 \cdot 16$ bits to represent whence the $\mathbf{16} = \mathbf{17} - 1$ in the heartbeat.

5.3 Constraints

5.3.1 Bytehood, byte decompositions, binary and ternary checks

We ask that $\mathsf{BYTE\_1}$ be a byte column. We ask that $\mathsf{ACC\_1}$ accumulate its bytes, i.e.

1. $\textsf{IF}\ \mathsf{OOB}\square_i \neq \mathsf{OOB}\square_{i-1}\ \textsf{THEN}\ \mathsf{ACC\_1}_i = \mathsf{BYTE\_1}_i$
2. $\textsf{IF}\ \mathsf{OOB}\square_i = \mathsf{OOB}\square_{i-1}\ \textsf{THEN}\ \mathsf{ACC\_1}_i = 256 \cdot \mathsf{ACC\_1}_{i-1} + \mathsf{BYTE\_1}_i$.

The accumulator constructs a relatively small (1-byte, **4**-byte or **17**-byte) integer. This target integer will later be set to be the currently relevant "adjusted nonnegative difference". We ask that ROOB and OOB be binary columns.

5.3.2 `CALLDATALOAD` specific instructions

With `CALLDATALOAD` instructions we simply check whether the "(relative) offset" where the the evm starts reading the call data being loaded into the stack is $\geq \mathsf{CALLDATA\_SIZE}$. Indeed, the data carrying bytes of call data occupy, within that call data, the (relative) indices $\{0, 1, \ldots, \mathsf{CDS} - 1\}$.

ROOB	OOB		
1		$\Longleftrightarrow$	the high part of the offset is nonzero
0	1	$\Longleftrightarrow$	"offset $\geq \mathsf{CALLDATA\_SIZE}$"
0	0	$\Longleftrightarrow$	"offset $< \mathsf{CALLDATA\_SIZE}$"

We list the constraints *per se*.

$$\boxed{\text{All constraints in this subsection assume } \langle {}^{\diamond}\mathsf{CDL}\,⚐\rangle_i = 1}$$

1. We compute ROOB_i:
$$\begin{cases} \textsf{IF}\ \langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i \neq 0\ \textsf{THEN}\ \mathsf{ROOB}_i = 1 \\ \textsf{IF}\ \langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i = 0\ \textsf{THEN}\ \mathsf{ROOB}_i = 0 \end{cases}$$
2. $\textsf{IF}\ \mathsf{ROOB}_i = 1\ \textsf{THEN}\ \langle \mathsf{CDL\_OOB} \rangle_i = 1$;
3. $\textsf{IF}\ \mathsf{ROOB}_i = 0$:

Target constraint.

$\textsc{if}\ \mathsf{OOB}_i = 1\ \textsc{then}$

$$\textsc{if}\ \mathsf{CT}_i = \mathbf{16}\ \textsc{then}\ \langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i - \mathsf{REFS}_i = \mathsf{ACC\_1}_i$$

$\textsc{if}\ \mathsf{OOB}_i = 0\ \textsc{then}$

$$\textsc{if}\ \mathsf{CT}_i = \mathbf{3}\ \textsc{then}\ \mathsf{REFS}_i - \langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i - 1 = \mathsf{ACC\_1}_i$$

Flag constraint. $\langle \mathsf{CDL\_OOB} \rangle_i = \mathsf{OOB}_i$;

5.3.3 RETURNDATACOPY specific instructions

With `RETURNDATACOPY` we must check whether “offset + size” excedes the return data size. For `RETURNDATACOPY` to fail (other than for gas related reasons) the instruction must access a byte beyond the provided return data. Valid indices of return data for the set $\{\mathsf{i} \mid 0 \leq \mathsf{i} < \langle \mathsf{RDS} \rangle\}$. The slice of contiguous bytes of with initial index “offset” and size “size” covers the indices $\{\mathsf{i} \mid \mathsf{offset} \leq \mathsf{i} < \mathsf{offset}+\mathsf{size}\}$. This interval is included in the previous one *iff* $\mathsf{size} \leq \langle \mathsf{RDS} \rangle$. The following table subsumes the situation. It applies to `RETURNDATACOPY` only:

ROOB	OOB		
1		$\iff$	the offset or the size high part is nonzero
0	1	$\iff$	“offset + size > RDS”
0	0	$\iff$	“offset + size ≤ RDS”

We list the constraints *per se*.

$$\boxed{\text{All constraints in this subsection assume } \langle {}^{\diamond}\mathsf{RDC}\,⚐ \rangle_i = 1}$$

1. We compute ROOB_i:

$$\begin{cases} \textsc{if}\ \langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i \neq 0\ \textsc{then}\ \mathsf{ROOB}_i = 1 \\ \textsc{if}\ \langle \mathsf{SIZE}^{\mathsf{hi}} \rangle_i \neq 0\ \textsc{then}\ \mathsf{ROOB}_i = 1 \\ \textsc{if}\ \big(\langle \mathsf{OFF}^{\mathsf{hi}} \rangle_i = 0\ \ \textsc{and}\ \langle \mathsf{SIZE}^{\mathsf{hi}} \rangle_i = 0\big)\ \textsc{then}\ \mathsf{ROOB}_i = 0 \end{cases}$$

2. $\textsc{if}\ \mathsf{ROOB}_i = 1\ \textsc{then}\ \langle \mathsf{RETDCX} \rangle_i = 1$;

3. $\textsc{if}\ \mathsf{ROOB}_i = 0$:

   Target constraint. We test whether “offset + size” excedes the return data size

   (a) $\textsc{if}\ \mathsf{OOB}_i = 1\ \textsc{then}$

   $$\textsc{if}\ \mathsf{CT}_i = \mathbf{16}\ \textsc{then}\ \big(\langle \mathsf{OFF}^{\mathsf{lo}} \rangle_i + \langle \mathsf{SIZE}^{\mathsf{lo}} \rangle_i\big) - \mathsf{REFS}_i - 1 = \mathsf{ACC\_1}_i$$

   (b) $\textsc{if}\ \mathsf{OOB}_i = 0\ \textsc{then}$

   $$\textsc{if}\ \mathsf{CT}_i = \mathbf{3}\ \textsc{then}\ \mathsf{REFS}_i - \big(\langle \mathsf{OFF}^{\mathsf{lo}} \rangle_i + \langle \mathsf{SIZE}^{\mathsf{lo}} \rangle_i\big) = \mathsf{ACC\_1}_i$$

   Flag constraint. $\langle \mathsf{RETDCX} \rangle_i = \mathsf{OOB}_i$;

5.3.4 JUMP / JUMPI specific instructions

With `JUMP` / `JUMPI` instructions we must check whether “counter” excedes the code size. Recall that the “counter” (which aims to be the next value of the $\textsf{PC}$ column) should point within the currently executing bytecode. Bytes within that bytecode are indexed starting from 0. (Potentially) valid indices are thus in the range $\{0, \dots, \langle \textsf{CODESIZE} \rangle - 1\}$. The following table applies to `JUMP` / `JUMPI` instructions only:

ROOB	OOB		
1		$\iff$	the high part of the new counter is nonzero
0	1	$\iff$	“counter $\geq$ $\textsf{CODESIZE}$”
0	0	$\iff$	“counter $<$ $\textsf{CODESIZE}$”

We list the constraints *per se*.

$$\boxed{\text{All constraints in this subsection assume } \langle {}^{\diamond}\textsf{RDC}\,\text{⚐}\rangle_i = 1}$$

1. We compute $\textsf{ROOB}_i$:
$$\begin{cases} \textsc{if}\ \langle \textsf{OFF}^{\textsf{hi}} \rangle_i \neq 0\ \textsc{then}\ \textsf{ROOB}_i = 1 \\ \textsc{if}\ \langle \textsf{OFF}^{\textsf{hi}} \rangle_i = 0\ \textsc{then}\ \textsf{ROOB}_i = 0 \end{cases}$$
2. $\textsc{if}$ $\textsf{ROOB}_i = 1$ $\textsc{then}$ $\langle \textsf{JOOB} \rangle_i = 1$;
3. $\textsc{if}$ $\textsf{ROOB}_i = 0$:

   Target constraint.
   $\textsc{if}$ $\textsf{OOB}_i = 1$ $\textsc{then}$
   $$\textsc{if}\ \textsf{CT}_i = \mathbf{16}\ \textsc{then}\ \langle \textsf{OFF}^{\textsf{hi}} \rangle_i - \textsf{REFS}_i = \textsf{ACC\_1}_i$$
   $\textsc{if}$ $\textsf{OOB}_i = 0$ $\textsc{then}$
   $$\textsc{if}\ \textsf{CT}_i = \mathbf{3}\ \textsc{then}\ \textsf{REFS}_i - \langle \textsf{OFF}^{\textsf{hi}} \rangle_i - 1 = \textsf{ACC\_1}_i$$

   Flag constraint. $\langle \textsf{JOOB} \rangle_i = \textsf{OOB}_i$;

5.3.5 RETURN specific instructions

With `RETURN` in a deployment context (i.e. $\textsf{CTYPE} = 1$) we must check whether the size of the `RETURN` instruction excedes the maximum code size of $\texttt{0x6000} = 24576$. For `RETURN` to fail (other than for gas related reasons) we must have $\textsf{size} > 24576$.

ROOB	OOB		
1		$\iff$	the offset (or size) high part is nonzero
0	1	$\iff$	“size $>$ 24576”
0	0	$\iff$	“size $\leq$ 24576”

We list the constraints *per se*.

$$\boxed{\text{All constraints in this subsection assume } \langle {}^{\diamond}\textsf{RETURN}\,\text{⚐}\rangle_i = 1}$$

1. We compute $\textsf{ROOB}_i$:
$$\begin{cases} \textsc{if}\ \langle \textsf{SIZE}^{\textsf{hi}} \rangle_i \neq 0\ \textsc{then}\ \textsf{ROOB}_i = 1 \\ \textsc{if}\ \langle \textsf{SIZE}^{\textsf{hi}} \rangle_i = 0\ \textsc{then}\ \textsf{ROOB}_i = 0 \end{cases}$$

2. $\textsc{if}$ $\mathsf{ROOB}_i = 1$ $\textsc{then}$ $\langle \mathsf{MAXCSX} \rangle_i = 1$;

3. $\textsc{if}$ $\mathsf{ROOB}_i = 0$:

   Target constraint. We test whether "size" excedes the return data size

   (a) $\textsc{if}$ $\mathsf{OOB}_i = 1$ $\textsc{then}$

   $$\textsc{if}\ \mathsf{CT}_i = \mathbf{16}\ \textsc{then}\ \left(\langle \mathsf{SIZE}^{\mathsf{lo}} \rangle_i \right) - \mathsf{REFS}_i - 1 = \mathsf{ACC\_1}_i$$

   (b) $\textsc{if}$ $\mathsf{OOB}_i = 0$ $\textsc{then}$

   $$\textsc{if}\ \mathsf{CT}_i = \mathbf{3}\ \textsc{then}\ \mathsf{REFS}_i - \langle \mathsf{SIZE}^{\mathsf{lo}} \rangle_i = \mathsf{ACC\_1}_i$$

   Flag constraint. $\langle \mathsf{MAXCSX} \rangle_i = \mathsf{OOB}_i$;

Chapter 6

Memory expansion

6.1 Memory expansion module

6.1.1 Introduction

The purpose of the **memory expansion module** is

- to update, when appropriate, the current context's $\mathsf{MEMORY\_SIZE\_IN\_BYTES}$ (i.e. MSIZE);
- to verify $\langle \Delta\_\mathsf{MEMORY\_EXPANSION\_COST} \rangle$ (i.e. $\langle \Delta\mathsf{MXC} \rangle$) associated with memory expanding operations;
- to recognize grossly out of bounds memory operations and to verify $\langle \mathsf{MEMORY\_EXPANSION\_EXCEPTION} \rangle$ (i.e. $\langle \mathsf{MXX} \rangle$).

If we're being precise, this module *verifies* the claimed $\Delta\mathsf{MXC}$ gas which it imports from the central trace as $\langle \Delta\mathsf{MXC} \rangle$.

The memory expansion module is triggered by those instructions that raise the (instruction decoded) memory expansion flag $^\diamond\mathsf{MEMORY\_EXPANSION\_FLAG}$ (i.e. $^\diamond\mathsf{MXF}$). The Hub keeps a tally of the number of (potentially) memory expanding operations: the $\mathsf{MX}\square$ column. This stamp grows by 1 with every (potentially) memory expanding operation. In essence it satisfies $\mathsf{MX}\square_{i+1} = \mathsf{MX}\square_{i+1} + {}^\diamond\mathsf{MXF}_{i+1}$[1]. The present module imports it under $\langle \mathsf{MX}\square \rangle$. Its import is required as the order of operations is important in assessing an instruction's memory expansion cost.

(Potentially) memory expanding operations are split into three broad **memory expansion types**[2]. An instruction's $^\diamond\mathsf{MEMORY\_EXPANSION\_TYPE}$ (i.e. $^\diamond\mathsf{MXT}$) is instruction decoded in the main execution trace. The present module imports the memory expansion type $\langle {}^\diamond\mathsf{MXT} \rangle$ along with relevant values from the stack (i.e. four stack values). The zk-evm considers `MSIZE` a **type 0 instruction** (the only of its kind.) **Type 1a and 1b instructions** are those instructions whose memory expansion cost depends purely on an *offset* i.e. instructions with an implicit *size* parameter (32 for **type 1a** and 1 for **type 1b** depending on the instruction):

1. `MLOAD` (**type 1a**);
2. `MSTORE` (**type 1a**);
3. `MSTORE8` (**type 1b**).

Type 2 instructions compute memory expansion in terms on a a single *offset* and *size*:

[1] This is slightly simplified as the Hub needs to distinguish between instructions with $^\diamond\mathsf{TWO\_LINE\_INSTRUCTION} = 1$ and those with $^\diamond\mathsf{TWO\_LINE\_INSTRUCTION} = 0$.

[2] Five types in practice.

1. `CREATE`;
2. `CREATE2`;
3. `RETURN`;
4. `REVERT`;
5. `LOG0`-`LOG4`;
6. `SHA3`;
7. `CODECOPY`;
8. `EXTCODECOPY`;
9. `CALLDATACOPY`;
10. `RETURNDATACOPY`.

Type 3 instructions are `CALL`-type instruction. These compute memory expansion in terms on *two* sets of *offset* and *size* parameters: (a) the offset and size defining call data (b) the offset and size defining the memory segment where the callee might write (part of) its return data. The zk-evm needs to determine the maximum value memory expansion cost between the two. The relevant instructions are

1. `CALL`;
2. `CALLCODE`;
3. `STATICCALL`;
4. `DELEGATECALL`.

The present module deals with memory expansion uniformly. To that effect it first recognizes trivial cases: either when offsets or sizes are far too large or when no memory expansion *can* happen since relevant sizes are zero ($\textsf{NOOP} = 1$). In case offsets and sizes satisfy both requirements it produces the maximal offset(s) that may be written to or read from. If there are two sets of offsets

The reason memory expansion is its own module is that its internal clock (its "heartbeat") is distinct from that of the stack, that of the RAM pre-processor and that of the RAM data processor. Indeed, gas computations are done ahead of instruction execution. As such some instructions may trigger the memory expansion module only to later be filtered out (for causing an OOG exception) and thus never making it to the RAM offset processor (let alone the data processor.)

6.1.2 Columns

The following columns determine the heartbeat of the module:

1. ⟨MEMORY_EXPANSION_STAMP⟩: imported column; abbreviated to ⟨MX□⟩;
2. ⟨$^\diamond$MEMORY_EXPANSION_TYPE⟩: imported column: contains the memory expansion type of the instruction which triggered the memory expansion module; abbreviated to ⟨$^\diamond$MXT⟩;
3. ⟨MEMORY_EXPANSION_EXCEPTION⟩: imported binary column; indicates whether *both* maximal offsets fit into 4 bytes or not; abbreviated to ⟨MXX⟩;
4. RIDICULOUSLY_OUT_OF_BOUNDS: counter-constant binary column indicating whether an offset or a size is ridiculously out of bounds; abbreviated to ROOB;
5. NOOP: counter-constant binary column; lights up if relevant size(s) are zero (i.e. the underlying instruction is a no-op *from the point of view of memory expansion and that alone*);
6. COUNTER: counter column; grows monotonically and resets to 0 with every new instruction; abbreviated to CT;

The COUNTER column counts either from 0 to 3 or from 0 to 16. Which of the two is the cut-off point depends on whether the maximal offset fits into 4 bytes ($\langle\textsf{MXX}\rangle = 0$) or not ($\langle\textsf{MXX}\rangle = 1$). Maximal offsets are defined as sums of two 16 byte integers and as such may overflow 16 bytes. Computing their byte decomposition may thus require 17 bytes. This explains CT's threshold at 16 rather than the $15 = 16 - 1$ found in other modules. Imported columns are surrounded by $\langle \cdots \rangle$.

7. $\langle \mathsf{CN} \rangle$: imported column; contains the execution context number currently executing an instruction raising the memory expansion flag;

Its only purpose is in

8. $\langle {}_1\mathsf{VAL}^{\mathsf{hi}} \rangle$, $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle$: import of (the high and low part of) the first stack value;
9. $\langle {}_2\mathsf{VAL}^{\mathsf{hi}} \rangle$, $\langle {}_2\mathsf{VAL}^{\mathsf{lo}} \rangle$: same, *mutatis mutandis*;
10. $\langle {}_3\mathsf{VAL}^{\mathsf{hi}} \rangle$, $\langle {}_3\mathsf{VAL}^{\mathsf{lo}} \rangle$: same, *mutatis mutandis*;
11. $\langle {}_4\mathsf{VAL}^{\mathsf{hi}} \rangle$, $\langle {}_4\mathsf{VAL}^{\mathsf{lo}} \rangle$: same, *mutatis mutandis*;
12. BYTE_0, . . . , BYTE_8: byte columns;
13. ACC_0, . . . , ACC_6: accumulator columns;

The first byte columns BYTE_0, . . . , BYTE_6 provide the bytes accumulated in ACC_0, . . . , ACC_6. The byte columns BYTE_7, BYTE_8 provide auxiliary bytes that are used, for instance, to *complete* byte decompositions of *medium-sized* numbers (i.e. 6-byte integers) that may appear in calculations.

14. MSIZE: counter-constant column; contains the size of active memory in bytes (counting continuously from position 0) *before* excution of the current instruction; though imported, its value is only justified here;
15. MSIZE^{ν}: counter-constant column; *may* contain the size of active memory in bytes (counting continuously from position 0) *after* excution of the current instruction;
16. MEMORY_EXPANSION_COST: counter-constant column; cotains the currently valid value of "$C_{\text{mem}}(a)$"[3]; abbreviated to MXC;
17. $\mathsf{MEMORY\_EXPANSION\_COST}^{\nu}$: counter-constant column; either retains the value MXC if no memory expansion took place or, if memory expansion *did* occur, stores the updated value of MXC; abbreviated to MXC^{ν};
18. $\langle \Delta\mathsf{\_MEMORY\_EXPANSION\_COST} \rangle$: imported column containing the claimed gas expansion cost; abbreviated to $\langle \Delta\mathsf{MXC} \rangle$;
19. $\langle \mathsf{SIZE\_IN\_EVM\_WORDS} \rangle$: imported column; for instructions of type 2 that are in bounds contains $\lceil \mathsf{SIZE}/32 \rceil$; abbreviated to $\langle \mathsf{SEVMW} \rangle$;

The claimed gas expansion cost is verified in the present module as $\langle \Delta\mathsf{MXC} \rangle = \mathsf{MXC}^{\nu} - \mathsf{MXC}$.

20. COMP: binary counter-constant column; equals 1 *iff* $\mathsf{LAST\_OFFSET}^1 \geq \mathsf{LAST\_OFFSET}^2$; comes into play only for memory expanding instructions involving two offsets;
21. MAX_OFFSET: equals $\max \left\{ \langle \mathsf{LAST\_OFFSET}^1 \rangle, \langle \mathsf{LAST\_OFFSET}^2 \rangle \right\}$ *when both are in bounds*;
22. MEMORY_EXPANSION_EVENT: given $\mathsf{ROOB} = \mathsf{NOOP} = 0$, indicates whether $\mathsf{MAX\_OFFSET} > \mathsf{MSIZE}$ and thus whether the current instruction *may* incur memory expansion cost; abbreviated to MXE;

[3] Notation taken from the Ethereum Yellow Paper [7]:

$$C_{\text{mem}}(a) = G_{\text{mem}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

where a is the current number of evm words in memory.

6.1.3 Offset bounds

Touching (i.e. reading or writing) the byte at offset OFFSET in memory may incur a gas cost on top of the intrinsic gas cost of the instruction. This extra **memory expansion cost** applies whenever the offset is greater than any other offset previously touched, or more precisely: when the byte belongs to slice of 32 consecutive bytes with word offset $a = \lceil(\mathsf{OFFSET} + 1)/32\rceil$ greater than that of any previously accessed byte.

OFFSET	0	1	···	31	32	33	···	63	64	65	···	95	···
word offset	$a = 1$				$a = 2$				$a = 3$				

The memory expansion module has a notion of **smallness**: small integers are 4-byte integers. Whenever an $\mathsf{offset} + \mathsf{size}$ excedes this threshold (given that $\mathsf{size} \neq 0$) the memory expansion module "gives up" on the instruction and raises the $\langle\mathsf{MXX}\rangle$-flag. We further refer the reader to the following paragraph from the Ethereum Yellow Paper [7]

> [The] total fee for memory-usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices [...] are included in the range [...] Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality.

The memory expansion cost for a byte offset $\mathsf{OFFSET} \geq 256^4$ is, setting and $G_{\text{memory}} = 3$

$$\geq G_{\text{memory}} \cdot a + \lfloor a^2/512\rfloor \approx 35 \text{ TGas}$$

When a (potentially) memory expanding instruction has its largest offset(s) "within bounds" and isn't a noop, the memory expansion module sets the *two* last touched offsets as $\mathsf{LAST\_OFFSET}^1 = \mathsf{OFFSET}_1 + (\mathsf{SIZE}_1 - 1)$ and $\mathsf{LAST\_OFFSET}^2 = \mathsf{OFFSET}_2 + (\mathsf{SIZE}_2 - 1)$. If the memory expansion type $\langle {}^{\Diamond}\mathsf{MXT}\rangle$ requires only one offset the second one is instead set to 0; if a size parameter is zero then the associated $\mathsf{LAST\_OFFSET}$ is also set to 0. With all these parameters in place the first task of the present module is to determine the maximum of the two:

$$\mathsf{MAX\_OFFSET} = \max\left\{\mathsf{LAST\_OFFSET}^1, \mathsf{LAST\_OFFSET}^2\right\}$$

It then compares $\mathsf{MAX\_OFFSET}$ to MSIZE. If $\mathsf{MAX\_OFFSET} \leq \mathsf{MSIZE}$ its work is essentially done as no memory expansion is triggered. Otherwise, if $\mathsf{MAX\_OFFSET} > \mathsf{MSIZE}$ it computes $\lceil \mathsf{MAX\_OFFSET}/32\rceil$ which it does by establishing the following variation on the euclidean division

$$\mathsf{MAX\_OFFSET} + 1 = 32 \cdot \mathsf{q} - \mathtt{r} \qquad (\clubsuit)$$

with $\mathtt{r} \in \{0, 1, \ldots 31\}$. In the arithmetization, $\mathsf{q} = \mathsf{ACC\_5}$. The next step is to compute the quadratic part of the memory expansion cost which requires determining the euclidean division of q by 512:

$$\begin{cases} \mathsf{q}^2 &= 512 \cdot \mathsf{q}' + \mathtt{r}', \quad 0 \leq \mathtt{r}' < 512, \\ \mathtt{r}' &= 256 \cdot \epsilon + \mathtt{b}, \qquad \epsilon \in \{0, 1\}, \ 0 \leq \mathtt{b} < 256, \end{cases} \qquad (\spadesuit)$$

In the arithmetization $\mathsf{q}' = \mathsf{ACC\_6}$.

Given the smallness bounds on $\mathsf{MAX\_OFFSET} < 2 \cdot 256^4 = 2^{32+1}$, we see that

1. q can be of the order of magnitude of $\approx 2^{27}$ and defines a 4-byte integer
2. q' can be of the order of magnitude of $\approx 2^{45}$ and defines a $(4 + 2)$-byte integer

The memory expansion cost is therefore, with $G_{\text{mem}} = 3$:

$$C_{\text{mem}} = G_{\text{memory}} \cdot \mathsf{q} + \mathsf{q}'$$

6.2 General constraints

6.2.1 Heartbeat

The columns $\langle \mathsf{MX\square} \rangle$, $\langle {}^{\diamond}\mathsf{MXT} \rangle$, $\langle \mathsf{MXX} \rangle$, ROOB, NOOP and CT define the heartbeat of the memory expansion module. Here are the constraints they satisfy:

1. $\langle \mathsf{MX\square} \rangle_0 = 0$;
2. $\langle \mathsf{MX\square} \rangle$ is nondecreasing, i.e. $\forall i,\ \langle \mathsf{MX\square} \rangle_{i+1} \in \{\langle \mathsf{MX\square} \rangle_i, 1 + \langle \mathsf{MX\square} \rangle_i\}$;
3. IF $\langle \mathsf{MX\square} \rangle_i = 0$ THEN the whole row is null;
4. IF $\langle \mathsf{MX\square} \rangle_{i+1} \neq \langle \mathsf{MX\square} \rangle_i$ THEN $\mathsf{CT}_{i+1} = 0$;
5. IF $\big(\langle \mathsf{MX\square} \rangle_i \neq 0$ AND $\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType0}\big)$ THEN

$$\begin{cases} \mathsf{ROOB}_i & = & 0 \\ \mathsf{NOOP}_i & = & 1 \\ \langle \mathsf{MXX} \rangle_i & = & 0 \\ \langle \mathsf{MX\square} \rangle_{i+1} & = & 1 + \langle \mathsf{MX\square} \rangle_i \end{cases}$$

   In other words, `MSIZE` instructions occupy a single line in the memory expansion module.

Nothing interesting happens if offsets are ridiculously out of bounds. The $\mathsf{MEMORY\_EXPANSION\_EXCEPTION}$ is automatically triggered.

6. IF $\langle \mathsf{MX\square} \rangle_i \neq 0$ AND $\mathsf{ROOB}_i = 1$ THEN

$$\begin{cases} \langle \mathsf{MXX} \rangle_i & = & 1 \\ \langle \mathsf{MX\square} \rangle_{i+1} & = & 1 + \langle \mathsf{MX\square} \rangle_i \end{cases}$$

Nothing interesting happens if none of the sizes are nonzero, i.e. if the instruction is a noop from the point of view of memory expansion.

7. IF $\Big(\langle \mathsf{MX\square} \rangle_i \neq 0$ AND $\mathsf{ROOB}_i = 0$ AND $\mathsf{NOOP}_i = 1\Big)$ THEN

$$\begin{cases} \mathsf{CT}_i = 0 \\ \langle \mathsf{MXX} \rangle_i = 0 \\ \langle \mathsf{MX\square} \rangle_{i+1} = 1 + \langle \mathsf{MX\square} \rangle_i \end{cases}$$

The following case is the main case of interest: the zk-evm deals with a real instruction, i.e. $\langle \mathsf{MX\square} \rangle \neq 0$, offsets aren't ridiculously out of bounds, i.e. $\mathsf{ROOB} = 0$, and some size is nonzero, i.e. $\mathsf{NOOP} = 0$.

8. IF $\Big(\langle \mathsf{MX\square} \rangle_i \neq 0$ AND $\mathsf{ROOB}_i = 0$ AND $\mathsf{NOOP}_i = 0\Big)$ THEN
   (a) IF $\langle {}^{\diamond}\mathsf{MXT} \rangle \neq \texttt{memExpType0}$ THEN
      i. IF $\langle \mathsf{MXX} \rangle_i = 0$ THEN
         A. IF $\mathsf{CT}_i \neq \mathbf{3}$ THEN

$$\begin{cases} \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \\ \langle \mathsf{MX\square} \rangle_{i+1} = \langle \mathsf{MX\square} \rangle_i \\ \langle \mathsf{MXX} \rangle_{i+1} = \langle \mathsf{MXX} \rangle_i \end{cases}$$

         B. IF $\mathsf{CT}_i = \mathbf{3}$ THEN $\langle \mathsf{MX\square} \rangle_{i+1} = 1 + \langle \mathsf{MX\square} \rangle_i$
      ii. IF $\langle \mathsf{MXX} \rangle_i = 1$ THEN

A. IF $\mathsf{CT}_i \neq \mathbf{16}$ THEN

$$\begin{cases} \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i \\ \langle \mathsf{MX}\square \rangle_{i+1} = \langle \mathsf{MX}\square \rangle_i \\ \langle \mathsf{MXX} \rangle_{i+1} = \langle \mathsf{MXX} \rangle_i \end{cases}$$

B. IF $\mathsf{CT}_i = \mathbf{16}$ THEN $\langle \mathsf{MX}\square \rangle_{i+1} = 1 + \langle \mathsf{MX}\square \rangle_i$

9. IF $\Big(\langle \mathsf{MX}\square \rangle_N \neq 0$ AND $\mathsf{ROOB}_N = 0 \Big)$ THEN
   (a) IF $\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType0}$ there are no “end of the execution trace” constraints
   (b) IF $\langle {}^{\diamond}\mathsf{MXT} \rangle_i \neq \texttt{memExpType0}$ THEN
      i. IF $\langle \mathsf{MXX} \rangle_N = 0$ THEN $\mathsf{CT}_N = \mathbf{3}$;
      ii. IF $\langle \mathsf{MXX} \rangle_N = 1$ THEN $\mathsf{CT}_N = \mathbf{16}$.

In other words the module doesn’t terminate mid instruction.

6.2.2 Counter constancy

We say that a column X is **counter-constant** if it satisfies

$$\forall i,\ \mathsf{CT}_i \neq 0 \implies \mathsf{X}_i = \mathsf{X}_{i-1}.$$

Note that $\langle \mathsf{MX}\square \rangle$ and $\langle \mathsf{MXX} \rangle$ are counter-constant by construction. Note that counter constancy of $\langle \mathsf{MX}\square \rangle$ implies counter constancy for *all* imported columns. The following columns are counter-constant:

1. COMP
2. MXE
3. MSIZE
4. MSIZE^{ν}
5. MXC
6. MXC^{ν}

6.2.3 ROOB flag

The present section specifies the behaviour of the $\mathsf{RIDICULOUSLY\_OUT\_OF\_BOUNDS}$ column (i.e. ROOB). Its behaviour depends on the memory expansion type of the instruction at hand.

1. IF $\langle \mathsf{MX}\square \rangle_i = 0$ THEN $\mathsf{ROOB}_i = 0$;

2. IF $\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType0}$ THEN $\mathsf{ROOB}_i = 0$;

3. IF $\Big(\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType1a}$ OR $\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType1b} \Big)$ THEN

$$\begin{cases} \text{IF } \langle {}_{1}\mathsf{VAL}^{\mathsf{hi}} \rangle_i \neq 0 \text{ THEN } \mathsf{ROOB}_i = 1 \\ \text{IF } \langle {}_{1}\mathsf{VAL}^{\mathsf{hi}} \rangle_i = 0 \text{ THEN } \mathsf{ROOB}_i = 0 \end{cases}$$

4. IF $\langle {}^{\diamond}\mathsf{MXT} \rangle_i = \texttt{memExpType2}$ THEN

$$\mathsf{ROOB}_i = 1 \iff \begin{cases} \langle {}_{3}\mathsf{VAL}^{\mathsf{hi}} \rangle_i \neq 0 \\ \qquad\text{OR} \\ \Big(\langle {}_{1}\mathsf{VAL}^{\mathsf{hi}} \rangle_i \neq 0 \ \text{ AND } \ \langle {}_{3}\mathsf{VAL}^{\mathsf{lo}} \rangle_i \neq 0 \Big) \end{cases}$$

This translates to

$$\begin{cases} \textsc{If}\ \mathsf{ROOB}_i = 1\ \textsc{then}\ \Big(\langle {}_3\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0\ \ \textsc{or}\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0\Big) \\ \textsc{If}\ \mathsf{ROOB}_i = 0\ \textsc{then}\ \begin{cases} \langle {}_3\mathsf{VAL}^{\mathsf{hi}}\rangle_i = 0 & \textsc{and} \\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 \end{cases} \end{cases}$$

5. $\textsc{If}\ \langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType3}\ \textsc{then}$

$$\mathsf{ROOB}_i = 1 \iff \begin{cases} \langle {}_3\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0 & \textsc{or} \\ \Big(\langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0\ \ \textsc{and}\ \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0\Big) & \textsc{or} \\ \langle {}_4\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0 & \textsc{or} \\ \Big(\langle {}_2\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0\ \ \textsc{and}\ \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0\Big) \end{cases}$$

In constraints this becomes

$$\begin{cases} \textsc{If}\ \mathsf{ROOB}_i = 1\ \textsc{then}\ \begin{cases} \langle {}_3\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0 & \textsc{or} \\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0 & \textsc{or} \\ \langle {}_4\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0 & \textsc{or} \\ \langle {}_2\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0 \end{cases} \\ \textsc{If}\ \mathsf{ROOB}_i = 0\ \textsc{then}\ \begin{cases} \langle {}_3\mathsf{VAL}^{\mathsf{hi}}\rangle_i = 0 & \textsc{and} \\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 & \textsc{and} \\ \langle {}_4\mathsf{VAL}^{\mathsf{hi}}\rangle_i = 0 & \textsc{and} \\ \langle {}_2\mathsf{VAL}^{\mathsf{hi}}\rangle_i \cdot \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 \end{cases} \end{cases}$$

We provide some context. If $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType0}$ then the instruction is `MSIZE` which takes no size or offset arguments and can't provoke an out of bounds error. Thus $\mathsf{ROOB}_i = 0$ automatically. If $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1a}$ or $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1b}$ then the instruction is one of `MLOAD`, `MSTORE`, `MSTORE8`. It takes an offset (${}_1\mathsf{VAL}$) as its sole memory-expansion-relevant-argument. Offsets that are greater than 4 bytes may never occur because of gas related expenses, let alone offsets that occupy > 16 bytes (as witnessed by $\langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle \neq 0$.) If $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2}$ the instruction takes offset (${}_1\mathsf{VAL}$) and size (${}_3\mathsf{VAL}$) arguments. For such an instruction to be ridiculously out of bounds either its size parameter has to be huge (as witnessed by ${}_3\mathsf{VAL}^{\mathsf{hi}} \neq 0$) or its offset parameter has to be huge and its size parameter nonzero (as witnessed by ${}_1\mathsf{VAL}^{\mathsf{hi}} \neq 0$ and ${}_3\mathsf{VAL} \neq 0$.) The case $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType3}$ is entirely analoguous to `memExpType2` except that there are two $(\mathsf{offset}, \mathsf{size})$ pairs to consider.

6.2.4 NOOP flag

The present section computes the NOOP flag. Its definition is simple: a noop *from the point of view of memory expansion* happens precisely when all relevant size parameters are zero. No-op dependent constraints are subordinate to $\mathsf{ROOB} = 0$, in other words: the value of NOOP matters only when $\mathsf{ROOB} = 0$. As such the zk-evm focuses on the low part of the size parameter(s). The "no-operation" check could be handled in the main execution trace, but for simplicity it is handled in the present module.

1. $\textsc{If}\ \langle \mathsf{MX}\square\rangle_i = 0\ \textsc{then}\ \mathsf{NOOP}_i = 0$[4]

2. $\textsc{If}\ \langle \mathsf{MX}\square\rangle_i \neq 0\ \textsc{then}$

[4] Note that this condition is redundant: it was already imposed in the heartbeat section 13.2.1

(a) IF $\mathsf{ROOB}_i = 1$ THEN $\mathsf{NOOP}_i = 1$

(b) IF $\mathsf{ROOB}_i = 0$ THEN

i. IF $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType0}$ THEN $\mathsf{NOOP}_i = 1$

ii. IF $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1a}$ THEN $\mathsf{NOOP}_i = 0$

iii. IF $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1b}$ THEN $\mathsf{NOOP}_i = 0$

iv. IF $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2}$ THEN

$$\begin{cases} \text{IF } \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 \text{ THEN } \mathsf{NOOP}_i = 1 \\ \text{IF } \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0 \text{ THEN } \mathsf{NOOP}_i = 0 \end{cases}$$

v. IF $\langle {}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType3}$ THEN $\mathsf{NOOP}_i = 0$

$$\begin{cases} \text{IF } \Big(\langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0 \;\text{ AND }\; \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i = 0\Big) \text{ THEN } \mathsf{NOOP}_i = 1 \\ \text{IF } \langle {}_3\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0 \text{ THEN } \mathsf{NOOP}_i = 0 \\ \text{IF } \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i \neq 0 \text{ THEN } \mathsf{NOOP}_i = 0 \end{cases}$$

We further settle the expected behaviour in case of a (from the point of view of the memory expansion module) noop. No memory expansion happens, memory size remains the same. The associated constraints are thus:

1. IF $\mathsf{NOOP}_i = 1$ THEN

$$\begin{cases} \langle \Delta\mathsf{MXC}\rangle_i & = & 0 \\ \mathsf{MSIZE}_i^{\nu} & = & \mathsf{MSIZE}_i \\ \mathsf{MXC}_i^{\nu} & = & \mathsf{MXC}_i \\ \text{IF } \langle {}^{\diamond}\mathsf{MXT}\rangle_i & = & \texttt{memExpType0} \text{ THEN } \begin{cases} \langle {}_4\mathsf{VAL}^{\mathsf{hi}}\rangle_i = 0 \\ \langle {}_4\mathsf{VAL}^{\mathsf{lo}}\rangle_i = \mathsf{MSIZE}_i \end{cases} \end{cases}$$

Recall that `MSIZE` is the only **type 0** instruction. The constraints in that case push the memory size onto the stack.

6.2.5 Byte decompositions

We impose the following constraints, for $k = 0, 2, \ldots, 6$

1. IF $\mathsf{CT}_i = 0$ THEN $\mathsf{ACC\_k}_i = \mathsf{BYTE\_k}_i$

2. IF $\mathsf{CT}_i \neq 0$ THEN $\mathsf{ACC\_k}_i = 256 \cdot \mathsf{ACC\_k}_{i-1} + \mathsf{BYTE\_k}_i$

The byte columns $\mathsf{BYTE\_7}$ and $\mathsf{BYTE\_8}$ serve a different purpose: rather than partake in a classical byte decomposition, they are used to store auxiliary bytes. They only play a role in case $\mathsf{ROOB} = \mathsf{NOOP} = 0$. We further impose **bytehood** constraints in all 8 byte columns $\mathsf{BYTE\_0}, \ldots, \mathsf{BYTE\_8}$.

6.3 Specialized constraints

6.3.1 Standing hypothesis

All constraints in section 6.3 and all its subsections assume $\langle \mathsf{MX}\square\rangle_i \neq 0, \mathsf{NOOP}_i = 0$ and $\mathsf{ROOB}_i = 0$

6.3.2 Max offsets

This section defines maximal offsets. We define a pair of maximal offsets (even in case the instruction requires only one offset, size pair). Definitions depend on the memory expansion type.

1. $\text{IF } \langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1a}$

$$\begin{cases} \mathsf{LAST\_OFFSET}^1_i = \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i + 31 \\ \mathsf{LAST\_OFFSET}^2_i = 0 \end{cases}$$

2. $\text{IF } \langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType1b}$

$$\begin{cases} \mathsf{LAST\_OFFSET}^1_i = \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i \\ \mathsf{LAST\_OFFSET}^2_i = 0 \end{cases}$$

3. $\text{IF } \langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2}$

$$\begin{cases} \mathsf{LAST\_OFFSET}^1_i = \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i + \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i - 1 \\ \mathsf{LAST\_OFFSET}^2_i = 0 \end{cases}$$

4. $\text{IF } \langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType3}$

$$\begin{cases} \begin{cases} \text{IF } \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i \neq 0 \text{ THEN } \mathsf{LAST\_OFFSET}^1_i = \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i + \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i - 1 \\ \text{IF } \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = 0 \text{ THEN } \mathsf{LAST\_OFFSET}^1_i = 0 \end{cases} \\ \begin{cases} \text{IF } \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i \neq 0 \text{ THEN } \mathsf{LAST\_OFFSET}^2_i = \langle {}_2\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i + \langle {}_4\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i - 1 \\ \text{IF } \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = 0 \text{ THEN } \mathsf{LAST\_OFFSET}^2_i = 0 \end{cases} \end{cases}$$

Let us clarify the preceding constraints. In the case where $\langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2}$ the standing assumption $\mathsf{NOOP}_i = 0$ implies that $\langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i \geq 1$. In the case where $\langle {}^{\Diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType3}$ it can happen that $\langle {}_1\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = \langle {}_3\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = 0$ or $\langle {}_2\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = \langle {}_4\mathsf{VAL}^{\,\mathsf{lo}}\rangle_i = 0$ (but, given our standing assumption that $\mathsf{NOOP}_i = 0$, not both.) To prevent having one $\mathsf{LAST\_OFFSET}$ be equal to -1 we test for nullity of sizes.

The $\mathsf{LAST\_OFFSET}^2$ and $\mathsf{LAST\_OFFSET}^2$ thus produced are nonnegative and the sum of two 16 byte integers. They are thus **17** byte integers.

6.3.3 Offsets are out of bounds

This subsection is about justifying raising the $\langle\mathsf{MXX}\rangle$ flag if one (at least) of the max offsets isn't a **4**-byte integer.

1. $\text{IF } \Big(\langle\mathsf{MXX}\rangle_i = 1 \ \text{ AND } \mathsf{CT}_i = \mathbf{16}\Big) \text{ THEN}$

$$\Big(\mathsf{LAST\_OFFSET}^1_i - 256^{\mathbf{4}} - \mathsf{ACC\_1}_i\Big) \cdot \Big(\mathsf{LAST\_OFFSET}^2_i - 256^{\mathbf{4}} - \mathsf{ACC\_2}_i\Big) = 0.$$

in other words one of $\mathsf{LAST\_OFFSET}^1$ or $\mathsf{LAST\_OFFSET}^2$ is $\geq 256^{\mathbf{4}}$

6.3.4 Offsets are in bounds

Preliminary computations

This section computes memory expansion in case both maximal offsets are in bounds. The first point is to establish this bound assertion. We then compare the two maximal offsets and store the greatest of the two in $\mathsf{MAX\_OFFSET}$.

All constraints in this subsection further assume that $\langle\mathsf{MXX}\rangle_i = 0$ and $\mathsf{CT}_i = \mathbf{3}$.

1. The zk-evm computes, for instructions of $\langle{}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2}$ the number of evm-words $\lceil\mathsf{size}/32\rceil$ the data to hash or copy occupies:

   $$\text{IF } \langle{}^{\diamond}\mathsf{MXT}\rangle_i = \texttt{memExpType2} \text{ THEN } \begin{cases} \langle{}_3\mathsf{VAL}^{\mathsf{lo}}\rangle & = & 32\cdot\mathsf{ACC\_0}_i - \mathsf{BYTE\_7}_{i-2} \\ \mathsf{BYTE\_7}_{i-3} & = & (256-32)+\mathsf{BYTE\_7}_{i-2} \end{cases}$$

   Note that the bytehood constraint on $\mathsf{BYTE\_7}$ enforces $\mathsf{BYTE\_7}_{i-2} \in \{0,1,\dots,31\}$.

2. The zk-evm confirms smallness of both $\mathsf{LAST\_OFFSET}^1$ and $\mathsf{LAST\_OFFSET}^2$:

   $$\begin{cases} \mathsf{ACC\_1}_i & = & \mathsf{LAST\_OFFSET}^1_i \\ \mathsf{ACC\_2}_i & = & \mathsf{LAST\_OFFSET}^2_i \end{cases}$$

   In other words, $\mathsf{LAST\_OFFSET}^1$ and $\mathsf{LAST\_OFFSET}^2$ are both $\mathbf{4}$-byte integers;

3. The zk-evm compares $\mathsf{LAST\_OFFSET}^1$ and $\mathsf{LAST\_OFFSET}^2$:

   $$\mathsf{ACC\_3}_i = \left(\mathsf{LAST\_OFFSET}^1_i - \mathsf{LAST\_OFFSET}^2_i\right)\cdot(2\cdot\mathsf{COMP}_i - 1) + (\mathsf{COMP}_i - 1)$$

   In other words:

   $$\begin{cases} \mathsf{COMP} = 1 & \iff & \mathsf{LAST\_OFFSET}^1 \geq \mathsf{LAST\_OFFSET}^2 \\ \mathsf{COMP} = 0 & \iff & \mathsf{LAST\_OFFSET}^1 < \mathsf{LAST\_OFFSET}^2 \end{cases}$$

4. The zk-evm sets $\mathsf{MAX\_OFFSET}_i$ to be the maximum of the two max offsets:

   $$\mathsf{MAX\_OFFSET}_i = \mathsf{COMP}_i\cdot\mathsf{LAST\_OFFSET}^1_i + (1-\mathsf{COMP}_i)\cdot\mathsf{LAST\_OFFSET}^2_i$$

   In other words,

   $$\begin{cases} \text{IF } \mathsf{LAST\_OFFSET}^1 \geq \mathsf{LAST\_OFFSET}^2 \text{ THEN } \mathsf{MAX\_OFFSET} = \mathsf{LAST\_OFFSET}^1 \\ \text{IF } \mathsf{LAST\_OFFSET}^1 < \mathsf{LAST\_OFFSET}^2 \text{ THEN } \mathsf{MAX\_OFFSET} = \mathsf{LAST\_OFFSET}^2 \end{cases}$$

5. The zk-evm decides whether memory expansion took place

   $$\mathsf{ACC\_4}_i = \left(\mathsf{MAX\_OFFSET}_i + 1 - \mathsf{MSIZE}_i\right)\cdot(2\cdot\mathsf{MXE}_i - 1) - \mathsf{MXE}_i.$$

   In other words,

   $$\begin{cases} \mathsf{MXE} = 1 & \iff & \mathsf{MAX\_OFFSET} + 1 > \mathsf{MSIZE} \\ \mathsf{MXE} = 0 & \iff & \mathsf{MAX\_OFFSET} + 1 \leq \mathsf{MSIZE} \end{cases}$$

6. Some parameters are updated

   (a) IF $\mathsf{MXE}_i = 0$ THEN

   $$\begin{cases} \mathsf{MSIZE}^{\nu}_i & = & \mathsf{MSIZE}_i, \\ \mathsf{MXC}^{\nu}_i & = & \mathsf{MXC}_i, \end{cases}$$

   In other words: if no memory expansion took place then MSIZE and MXC don't change .

   (b) IF $\mathsf{MXE}_i = 1$ THEN

   $$\begin{cases} \mathsf{MSIZE}^{\nu}_i & = & \mathsf{MAX\_OFFSET}_i + 1 \\ \mathsf{MXC}^{\nu}_i & = & \text{... next section ...} \end{cases}$$

   in other words, if memory expansion took place then we update the memory size; the updated expansion cost will be computed in the following section.

Memory expansion cost update

$$\boxed{\text{All constraints in this subsection further assume that } \langle\mathsf{MXX}\rangle_i = 0, \mathsf{CT}_i = 3 \text{ and } \mathsf{MXE}_i = 1.}$$

We compute the updated expansion cost. The following constraints apply *iff* $\mathsf{MXE} = 1$ in the current counter-cycle.

1. $\mathsf{ACC\_5}$ accumulates the bytes of $\lceil \mathsf{MSIZE}^\nu_i / 32 \rceil$:

$$\begin{cases} \mathsf{MSIZE}^\nu_i = 32 \cdot \mathsf{ACC\_5}_i - \mathsf{BYTE\_7}_i \\ \mathsf{BYTE\_7}_{i-1} = \mathsf{BYTE\_7}_i + (256 - 32) \quad (1) \end{cases}$$

   The bytehood constraint on $\mathsf{BYTE\_7}$ and (1) imply that $\mathtt{r} := \mathsf{BYTE\_7}_i \in \{0, 1, \ldots, 31\}$. This verifies eq. (♣).

2. $\mathsf{ACC\_6}$ accumulates the first 4 bytes of the euclidean division of $\mathsf{ACC\_5}^2_i$ by 512:

$$\begin{cases} \mathsf{ACC\_5}^2_i & = & 512 \cdot \Big(\mathsf{ACC\_6}_i + 256^{4} \cdot \mathsf{BYTE\_8}_{i-2} + 256^{4+1} \cdot \mathsf{BYTE\_8}_{i-3} \Big) \\ & & \quad + 256 \cdot \mathsf{BYTE\_8}_{i-1} + \mathtt{b} \\ \mathsf{BYTE\_8}^2_{i-1} & = & \mathsf{BYTE\_8}_{i-1} \end{cases} \qquad (\bigstar)$$

3. We settle MXC^ν:

$$\begin{aligned} \mathsf{MXC}^\nu_i \quad = \quad & G_{\text{mem}} \cdot \mathsf{ACC\_5}_i \\ & + \mathsf{ACC\_6}_i \\ & \quad + 256^{4} \cdot \mathsf{BYTE\_8}_{i-2} \\ & \qquad + 256^{4+1} \cdot \mathsf{BYTE\_8}_{i-3} \end{aligned}$$

4. verify the gas expansion cost:

$$\langle \Delta\mathsf{MXC} \rangle_i = \mathsf{MXC}^\nu_i - \mathsf{MXC}_i.$$

We provide some explanatory details regarding equations (★). What is being verified is the following

$$\begin{cases} \mathsf{ACC\_5}^2_i & = & 512 \cdot \mathsf{q}' + \mathtt{r}' & (4) \\ \mathsf{q}' & := & \mathsf{ACC\_6}_i + 256^{4} \cdot \mathtt{b}_4 + 256^{4+1} \cdot \mathtt{b}_5 & (3) \\ \mathtt{r}' & := & 256 \cdot \epsilon + \mathtt{b} & (2) \\ \epsilon \text{ is a bit i.e. } \epsilon^2 = \epsilon & & & (1) \\ \mathtt{b}_4 & := & \mathsf{BYTE\_8}_{i-2} & \\ \mathtt{b}_5 & := & \mathsf{BYTE\_8}_{i-3} & \\ \epsilon & := & \mathsf{BYTE\_8}_{i-1} & \\ \mathtt{b} & := & \mathsf{BYTE\_8}_i & \end{cases}$$

(1) verifies that $\epsilon := \mathsf{BYTE\_8}_{i-1}$ is a bit; (2) verifies that $\mathtt{r}' \in \{0, 1, \ldots, 511\}$; (3) verifies that q' is a 6-byte integer; (4) verifies the euclidean division of $\mathsf{ACC\_5}^2_i$ by 512; together they verify Eq. (♠).

6.4 Consistency constraints

We impose consistency constraints. Consider a row permutation $\mathsf{X} \rightsquigarrow [\mathsf{X}]^{\text{⤮}}$ such that the rows of the following columns are listed in lexicographic order:

$$\Big([\langle \mathsf{CN} \rangle]^{\text{⤮}}, [\langle \mathsf{MX}\square \rangle]^{\text{⤮}} \Big)$$

We write "*a* row permutation" since there may be some (inconsequential) ambiguity: the module can start with an arbitrary number of null rows. Otherwise the ordering is unique. This row permutation groups together, in chronological order, all instructions raising the memory expansion flag executed within the same execution context. The constraints below thus impose coherence between values of MSIZE and MXC that may be separated (in the temporal execution trace) by memory expanding instructions in a descendant context.

1. IF $[\langle \mathsf{CN} \rangle]_i^{\text{🔀}} \neq 0$:

   (a) IF $[\langle \mathsf{CN} \rangle]_{i+1}^{\text{🔀}} = [\langle \mathsf{CN} \rangle]_i^{\text{🔀}}$ THEN

   $$\begin{cases} [\mathsf{MSIZE}]_{i+1}^{\text{🔀}} & = & [\mathsf{MSIZE}_i^{\nu}]^{\text{🔀}} \\ [\mathsf{MXC}]_{i+1}^{\text{🔀}} & = & [\mathsf{MXC}_i^{\nu}]^{\text{🔀}} \end{cases}$$

   (b) IF $[\langle \mathsf{CN} \rangle]_{i+1}^{\text{🔀}} \neq [\langle \mathsf{CN} \rangle]_i^{\text{🔀}}$ THEN

   $$\begin{cases} [\mathsf{MSIZE}]_{i+1}^{\text{🔀}} & = & 0 \\ [\mathsf{MXC}]_{i+1}^{\text{🔀}} & = & 0 \end{cases}$$

Chapter 7

Gas

7.1 Purpose

7.1.1 Purpose

The present document is a revised and expanded version of a previous (partial) specification of a zk-evm.

7.1.2 Triggers

The gas module is triggered by both exceptions and (context switching) instructions. The instructions that always trigger a call to this module are:

1. `STOP`
2. `RETURN`
3. `REVERT`
4. `SELFDESTRUCT`
5. `CREATE`
6. `CREATE2`
7. `CALL`
8. `CALLCODE`
9. `STATICCALL`
10. `DELEGATECALL`

Furthermore, any exception triggers a call to the present module. Out of gas exception behave differently from other exceptions in this respect.

We define `CALL`-type instructions to be any instruction among `CALL`, `CALLCODE`, `STATICCALL`, `DELEGATECALL`. We define `CREATE`-type instructions to be `CREATE` and `CREATE2` instructions.

7.2 Columns

7.2.1 Column descriptions

1. $\langle \mathsf{GAS}^{\omega} \rangle$: imported column containing the «old gas»; its value is computed in the Hub;
2. $\langle \mathsf{GAS}^{\kappa} \rangle$: imported column containing the «current gas»; its value is computed in the Hub;
3. $\langle \mathsf{GAS}^{\nu} \rangle$: imported column containing the «new gas»; its value is computed in the present module;
4. $\langle \mathsf{GAS}^{\varepsilon} \rangle$: imported column containing the «gas endowment»; its value is computed in the present module;

The **old gas** $\langle \mathsf{GAS}^{\omega} \rangle$ is the gas available before the instruction starts processing. The **current gas** $\langle \mathsf{GAS}^{\kappa} \rangle$ is the gas available after adding refunded gas (from successfully exiting a context or exiting via a `REVERT` instruction) and subtracting static gas costs and dynamic gas cost. Note that we define dynamic gas cost as the sum of (a) memory expansion cost (b) linear costs for $(b).(i)$ hashing (i.e. `SHA3` and `CREATE2`) $(b).(ii)$ copying (for `RETURNDATACOPY`, `CALLDATACOPY`, `CODECOPY`, `EXTCODECOPY`), $(b).(iii)$ code deployment (for `RETURN` in a deployment context), $(b).(iv)$ logging (for `LOG0`-`LOG4` (c) dynamic costs for `SLOAD` and `SSTORE` (d) address access costs (e) account existence cost (f) value transfer cost (for `CALL`s and `CREATE`s.) Note that this excludes costs child context gas endowments. The **gas endowment** $\langle \mathsf{GAS}^{\varepsilon} \rangle$ is the part of the gas endowment gifted by a parent context to its descendant context spawned through a `CREATE`-type instruction or `CALL`-type instruction. The **new gas** $\langle \mathsf{GAS}^{\nu} \rangle$ is the gas available after processing the instruction. It is obtained as either the or the Note that $\langle \mathsf{GAS}^{\nu} \rangle$ **doesn't** include gas refunds which may take place when exiting an execution context.

5. $\langle {}^{\diamond}\mathsf{L\_FLAG} \rangle$: imported binary column;
6. ⟨◇CALL⚐⟩: imported binary column; lights up precily for `CALL`-type instructions;

The instruction decoded $^{\diamond}\mathsf{L\_FLAG}$ lights up precisely for instructions which require the evaluation of the L function on some inputs. Recall that the L function si defined as

$$L(x) = x - \lfloor x/64 \rfloor.$$

The instruction decoded ◇CALL⚐ lights up precisely for `CALL`-type instructions, i.e. `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`.

7. $\langle {}_1\mathsf{VAL}^{\mathsf{hi}} \rangle$, $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle$: imported columns containing the high and low part of the first stack item;
8. $\langle \mathsf{GENERAL\_EXCEPTION} \rangle$: imported binary flag signaling whether an exception occurs at the current instruction;
9. $\langle \mathsf{OOGX} \rangle$: imported binary flag signaling an out of gas exception;
10. $\langle \mathsf{MXX} \rangle$: imported binary flag; signals whether memory expansion produced a gas cost so large it single handedly excedes 256^{4};
11. $\mathsf{LARGE\_BYTE\_DECOMPOSITION\_FLAG}$: binary flag that indicates whether a byte decomposition is required; abbreviated to LBDF;

The $\langle \mathsf{MXX} \rangle$ was justified in the memory expansion module. For `CALL`-type instructions $\langle {}_1\mathsf{VAL} \rangle$ contains the gas parameter.

12. CT: counter column; counts either from 0 to **3** or from 0 to **5** depending on $\langle \mathsf{OOGX} \rangle$;

7.3 Constraints

7.3.1 Heartbeat

The heartbeat of the gas module depends on the instruction at hand and on whether an out of gas exception occurred or not. We give more context. The zk-evm works under the assumption that the initial gas provided in the transaction is a 4-byte integer (i.e. ≤ 4.3 BGas.) An execution context's gas is continuously depleted as instructions pour in. The zk-evm raises the `outOfGasExceptionFlag` at row i *iff*

$$\mathsf{GAS}_i^{\omega} \geq 0 \quad \text{and} \quad \mathsf{GAS}_i^{\kappa} < 0.$$

The largest amount of gas that can be subtracted from the old gas to obtain the current gas comes from memory expansion. If the memory expansion module raised the $\langle \mathsf{MXX} \rangle$ flag the memory expansion gas is so large as to require no further verification for the assertion $\mathsf{GAS}_i^{\kappa} < 0$; thus only $\mathsf{GAS}_i^{\omega} \geq 0$ needs to be proven. We impose the following constraints:

1. ⟨GAS▢⟩$_0 = 0$; furthermore IF ⟨GAS▢⟩$_i = 0$ THEN the whole row is null;

2. $\langle \mathsf{GAS}\square \rangle$ is nondecreasing, i.e. $\langle \mathsf{GAS}\square \rangle_{i+1} \in \{\langle \mathsf{GAS}\square \rangle_i, 1 + \langle \mathsf{GAS}\square \rangle_i\}$;
3. IF $\langle \mathsf{GAS}\square \rangle_{i+1} \neq \langle \mathsf{GAS}\square \rangle_i$ THEN $\mathsf{CT}_{i+1} = 0$;
4. IF $\langle \mathsf{MXX} \rangle_i = 1$ THEN $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_i = 1$, furthermore
   (a) IF $\mathsf{CT}_i \neq \mathbf{3}$ THEN
   $$\begin{cases} \mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i, \\ \langle \mathsf{GAS}\square \rangle_{i+1} = \langle \mathsf{GAS}\square \rangle_i, \end{cases}$$
   (b) IF $\mathsf{CT}_i = \mathbf{3}$ THEN
   $$\begin{cases} \mathsf{CT}_{i+1} = 0, \\ \langle \mathsf{GAS}\square \rangle_{i+1} = 1 + \langle \mathsf{GAS}\square \rangle_i, \end{cases}$$
5. IF $\langle \mathsf{MXX} \rangle_i = 0$ THEN
   (a) IF $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_i = 1$:
      i. IF $\mathsf{CT}_i \neq \mathbf{5}$ THEN $\mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i$,
      ii. IF $\mathsf{CT}_i = \mathbf{5}$ THEN $\mathsf{CT}_{i+1} = 0$,
   (b) IF $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_i = 0$:
      i. IF $\mathsf{LBDF}_i = 1$ THEN
         A. IF $\mathsf{CT}_i \neq \mathbf{15}$ THEN $\mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i$,
         B. IF $\mathsf{CT}_i = \mathbf{15}$ THEN $\mathsf{CT}_{i+1} = 0$,
      ii. IF $\mathsf{LBDF}_i = 0$ THEN
         A. IF $\mathsf{CT}_i \neq \mathbf{3}$ THEN $\mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i$,
         B. IF $\mathsf{CT}_i = \mathbf{3}$ THEN $\mathsf{CT}_{i+1} = 0$.

We provide some details below:

- $\langle \mathsf{MXX} \rangle = 1$ *if and only if* the memory expansion module noticed that a size parameter of the present instuction is so large as to drive the memory expansion gas cost completely out of bounds. In this case the $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle$ exception is necessarily set and the gas module is only required to prove that the old gas $\langle \mathsf{GAS}^\omega \rangle$ is nonnegative. Given that the original transaction gas is a **4**-byte integer and gas can only decrement, the old gas amount $\langle \mathsf{GAS}^\omega \rangle$ must be a (nonnegative) **4**-byte integer.

- $\langle \mathsf{MXX} \rangle = 0$ can mean several things.
   - If $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle = 1$ then the zk-evm checks for $\langle \mathsf{GAS}^\omega \rangle \geq 0$ being a **4**-byte integer and $\langle \mathsf{GAS}^\kappa \rangle < 0$: $\langle \mathsf{GAS}^\kappa \rangle$ is obtained in the main execution trace by subtracting static gas and dynamic gas from $\langle \mathsf{GAS}^\omega \rangle$ (and potentilally adding refunded gas from a successful `CALL`-type instruction or `CREATE`-type instruction.) $\langle \mathsf{MXX} \rangle = 0$ implies that the dynamic gas cost is a **6**-byte integer and thus $-\langle \mathsf{GAS}^\kappa \rangle > 0$ is one, too.
   - If $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle = 0$ there is no out of gas exception i.e. $\langle \mathsf{GAS}^\kappa \rangle$ must be a nonnegative 4 byte integer. The only case that requires inquiry is that of a `CALL`-type instruction whose gas parameter is large so that $\mathsf{LARGE\_BYTE\_DECOMPOSITION\_FLAG} = 1$ i.e. $\langle {}_1\mathsf{VAL}^{\mathsf{hi}} \rangle = 0$ and $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle > L(\langle \mathsf{GAS}^\kappa \rangle)$. Establishing this inequality requires a **16**-byte-decomposition.

Largeness of the gas parameter is defined as $\langle {}_1\mathsf{VAL}^{\mathsf{hi}} \rangle \neq 0$ or $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle > L(\langle \mathsf{GAS}^\kappa \rangle)$. Note that whenever $\langle {}_1\mathsf{VAL}^{\mathsf{hi}} \rangle \neq 0$ the gas parameter is *very* large but establishing this requires no byte decomposition.

The constraints that follow are the usual constraints that impose that the last instruction is carried out to completion.

6. IF $\langle \mathsf{MXX} \rangle_N = 1$ THEN $\mathsf{CT}_N = \mathbf{3}$

7. IF $\langle \mathsf{MXX} \rangle_N = 0$ THEN

   (a) IF $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_N = 1$ THEN $\mathsf{CT}_N = \mathbf{5}$

   (b) IF $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_N = 0$:

      i. IF $\Big(\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 1$ AND $\mathsf{LBDF}_N = 1 \Big)$ THEN $\mathsf{CT}_N = \mathbf{15}$

      ii. IF $\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 0$ OR $\Big(\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 1$ AND $\mathsf{LBDF}_N = 0 \Big)$ THEN $\mathsf{CT}_N = \mathbf{3}$

Note: the original formulation of the finalization constraints concluded with

1. ...

2. if $\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION} \rangle_N = 0$:

   (a) if $\Big(\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 1$ and $\mathsf{LBDF}_N = 1 \Big)$ then $\mathsf{CT}_N = 15$

   (b) if $\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 0$ or $\Big(\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_N = 1$ and $\mathsf{LBDF}_N = 0 \Big)$ then $\mathsf{CT}_N = 3$

The present formulation is quicker and equivalent.

7.3.2 Constancy constraints

We say that a column X is $\langle \mathsf{GAS}\,\square \rangle$-constant if it satisfie

$$\langle \mathsf{GAS}\,\square \rangle_{i+1} = \langle \mathsf{GAS}\,\square \rangle_i \implies \mathsf{X}_{i+1} = \mathsf{X}_i.$$

Imported columns are automatically $\langle \mathsf{GAS}\,\square \rangle$-constant. We further ask that LBDF be $\langle \mathsf{GAS}\,\square \rangle$-constant.

7.3.3 Byte decompositions

We impose the following byte decomposition constraints.

1. IF $\langle \mathsf{GAS}\,\square \rangle_{i+1} \neq \langle \mathsf{GAS}\,\square \rangle_i$ THEN $\mathsf{ACC\_k}_{i+1} = \mathsf{BYTE\_k}_{i+1}$

2. IF $\langle \mathsf{GAS}\,\square \rangle_{i+1} = \langle \mathsf{GAS}\,\square \rangle_i$ THEN $\mathsf{ACC\_k}_{i+1} = 256 \cdot \mathsf{ACC\_k}_i + \mathsf{BYTE\_k}_{i+1}$

These apply for $k = 1, 2, 3, 4$. We further impose bytehood constraints on $\mathsf{BYTE\_k}$, $k = 1, 2, 3, 4, 5$.

7.3.4 The LARGE_BYTE_DECOMPOSITION_FLAG

The LARGE_BYTE_DECOMPOSITION_FLAG flag is set whenever a `CALL`-type instruction has a large gas parameter. Note that it intervenes only for `CALL`-type instructions with $\langle \mathsf{MXX} \rangle = 0$. These are the associated constraints:

1. IF $\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_i = 0$ THEN $\mathsf{LBDF}_i = 0$;

2. IF $\Big(\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_i = 1$ AND $\langle {}_1\mathsf{VAL}^{\,\mathsf{hi}} \rangle_i \neq 0 \Big)$ THEN $\mathsf{LBDF}_i = 0$.

Thus $\mathsf{LBDF}_i = 1$ may only hapen if both $\langle {}^{\diamond}\mathsf{CALL}\,\text{⚐} \rangle_i = 1$ AND $\langle {}_1\mathsf{VAL}^{\,\mathsf{hi}} \rangle_i = 0$ i.e. the underlying instruction is a `CALL`-type instruction and its gas parameter is a 16-byte integer. In this case it will hold that

$$\begin{cases} \mathsf{LBDF}_i = 1 \iff \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}} \rangle_i \leq (\text{maxGasAllowance}) \\ \mathsf{LBDF}_i = 0 \iff \langle {}_1\mathsf{VAL}^{\,\mathsf{lo}} \rangle_i > (\text{maxGasAllowance}) \end{cases}$$

With $\text{maxGasAllowance} = L(\langle \mathsf{GAS}^{\kappa} \rangle_i)$.

7.3.5 Target constraints

The target constraints reproduce the logic of the heartbeat.

1. $\textsc{if}\ \langle \mathsf{MXX}\rangle_i = 1\ \ \textsc{and}\ \mathsf{CT}_i = \mathbf{3}\ \textsc{then}$

$$\langle \mathsf{GAS}^{\omega}\rangle_i = \mathsf{ACC\_1}_i$$

   This establishes that before executing the instruction the remaining gas was nonnegative.

2. $\textsc{if}\ \langle \mathsf{MXX}\rangle_i = 0\ \textsc{then}$

   (a) $\textsc{if}\ \Big(\langle \mathsf{OUT\_OF\_GAS\_EXCEPTION}\rangle_i = 1\ \ \textsc{and}\ \mathsf{CT}_i = \mathbf{5}\Big)\ \textsc{then}$

$$\begin{cases} \langle \mathsf{GAS}^{\omega}\rangle_i & = & \mathsf{ACC\_1}_i \\ -\langle \mathsf{GAS}^{\kappa}\rangle_i + 1 & = & \mathsf{ACC\_2}_i \end{cases}$$

   This establishes that $\langle \mathsf{GAS}^{\omega}\rangle_i \geq 0$ and $\langle \mathsf{GAS}^{\kappa}\rangle_i < 0$

   (b) $\textsc{if}\ \langle \mathsf{OUT\_OF\_GAS\_EXCEPTION}\rangle_i = 0\ \textsc{then}$

   i. $\mathsf{LBDF}_i = 0\ \ \textsc{and}\ \mathsf{CT}_i = \mathbf{3}\ \textsc{then}$

   A.

$$\begin{cases} \langle \mathsf{GAS}^{\omega}\rangle_i & = & \mathsf{ACC\_1}_i \\ \langle \mathsf{GAS}^{\kappa}\rangle_i & = & \mathsf{ACC\_2}_i \end{cases}$$

   Note: the first target constraint is redundant.

   B. $\textsc{if}\ \langle {}^{\Diamond}\mathsf{L\_FLAG}\rangle_i = 1\ \textsc{then}$

   - the zk-evm computes $\lfloor \langle \mathsf{GAS}^{\kappa}\rangle_i / 64\rfloor$:

$$\begin{cases} \langle \mathsf{GAS}^{\kappa}\rangle_i & = & 64\cdot \mathsf{ACC\_3}_i + \mathsf{BYTE\_5}_i \\ \mathsf{BYTE\_5}_{i-1} & = & \mathsf{BYTE\_5}_i + (256-64) \end{cases}$$

     In other words, $\mathsf{ACC\_3}_i = \lfloor \langle \mathsf{GAS}^{\kappa}\rangle / 64\rfloor$. The second constraint witnesses the fact that $\mathsf{BYTE\_5}_i \in \{0, 1, \dots, 63\}$ is the remainder.

   - $\textsc{if}\ \langle {}^{\Diamond}\mathsf{CALL}\,⚐\rangle_i = 0\ \textsc{then}$

$$\begin{cases} \langle \mathsf{GAS}^{\varepsilon}\rangle_i & = & \langle \mathsf{GAS}^{\kappa}\rangle_i - \mathsf{ACC\_3}_i \\ \langle \mathsf{GAS}^{\nu}\rangle_i & = & \mathsf{ACC\_3}_i \end{cases}$$

     $\langle {}^{\Diamond}\mathsf{L\_FLAG}\rangle_i = 1$ and $\langle {}^{\Diamond}\mathsf{CALL}\,⚐\rangle_i = 0$ corresponds to a `CREATE`/`CREATE2` instruction. The above computes the $(63/64)$ths of the currently available gas $\langle \mathsf{GAS}^{\kappa}\rangle_i$ which are provided to the descendant context and the new gas balance of the current context (pre gas refund from exiting the descendant context).

   - $\textsc{if}\ \langle {}^{\Diamond}\mathsf{CALL}\,⚐\rangle_i = 1\ \textsc{then}$

$$\begin{cases} \textsc{if}\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i = 0\ \textsc{then} & \begin{cases} \langle \mathsf{GAS}^{\kappa}\rangle_i - \mathsf{ACC\_3}_i - \langle {}_1\mathsf{VAL}^{\mathsf{lo}}\rangle_i = \mathsf{ACC\_4}_i \quad (\star) \\ \langle \mathsf{GAS}^{\varepsilon}\rangle_i = \langle {}_1\mathsf{VAL}^{\mathsf{lo}}\rangle_i \\ \langle \mathsf{GAS}^{\nu}\rangle_i = \langle \mathsf{GAS}^{\kappa}\rangle_i - \langle {}_1\mathsf{VAL}^{\mathsf{lo}}\rangle_i \end{cases} \\ \textsc{if}\ \langle {}_1\mathsf{VAL}^{\mathsf{hi}}\rangle_i \neq 0\ \textsc{then} & \begin{cases} \langle \mathsf{GAS}^{\varepsilon}\rangle_i = \langle \mathsf{GAS}^{\kappa}\rangle_i - \mathsf{ACC\_3}_i \\ \langle \mathsf{GAS}^{\nu}\rangle_i = \mathsf{ACC\_3}_i \end{cases} \end{cases}$$

     $\langle {}^{\Diamond}\mathsf{L\_FLAG}\rangle_i = 1$ and $\langle {}^{\Diamond}\mathsf{CALL}\,⚐\rangle_i = 1$ corresponds to a `CALL`-type instructions. Constraint $(\star)$ means that the maximum gas allowance that the instruction tries to pass down to the descendant context is $\leq$ the maximum gas allowance that the present context may pass down to a descendant context.

ii. IF $\mathsf{LBDF}_i = 1$ AND $\mathsf{CT}_i = \mathbf{15}$ THEN

A.

$$\begin{cases} \langle \mathsf{GAS}^{\omega} \rangle_i & = & \mathsf{ACC\_1}_i \\ \langle \mathsf{GAS}^{\kappa} \rangle_i & = & \mathsf{ACC\_2}_i \end{cases}$$

Note: the first target constraint is (again) redundant.

B.

$$\begin{cases} \langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle_i - \big(\langle \mathsf{GAS}^{\kappa} \rangle_i - \mathsf{ACC\_3}_i + 1 \big) = \mathsf{ACC\_4}_i \quad (\star\star) \\ \langle \mathsf{GAS}^{\varepsilon} \rangle_i = \langle \mathsf{GAS}^{\kappa} \rangle_i - \mathsf{ACC\_3}_i \\ \langle \mathsf{GAS}^{\nu} \rangle_i = \mathsf{ACC\_3}_i \end{cases}$$

Constraint $(\star\star)$ means that $\langle {}_1\mathsf{VAL}^{\mathsf{lo}} \rangle_i > \langle \mathsf{GAS}^{\kappa} \rangle_i - \mathsf{ACC\_3}_i = L(\langle \mathsf{GAS}^{\kappa} \rangle_i)$, the maximum gas allowance of a descendant context. As such the descendant context is endowed with $L(\langle \mathsf{GAS}^{\kappa} \rangle_i)$ (which may be augmented by a call stipend $G_{\text{callstipend}} = 2300$ if the `CALL`-type instruction includes a value transfer.)

Note that $\mathsf{LBDF}_i = 1$ may only happen when the present instruction is a `CALL`-type instruction.

Chapter 8

Storage

8.1 Storage module

8.1.1 Storage instructions

The **storage module** deals with `SSTORE` and `SLOAD`. It is sensitive to exceptions (including `REVERT` instructions.) **Storage instructions** (i.e. `SSTORE` and `SLOAD`) are precisely the instruction which raise the $\mathsf{STORAGE\_FLAG}$ in the Hub.

8.1.2 Column descriptions

1. $\langle\mathsf{STORAGE\_STAMP}\rangle$: imported column containing the Hub's storage stamp; abbreviated to $\langle\mathsf{STO}\square\rangle$;

The $\mathsf{STORAGE\_STAMP}$ column in the Hub grows by one with every storage instruction.

The following columns contain imported columns that represent "execution context variables." These play a role for reordering arguments.

2. $\langle\mathsf{TX\#}\rangle$: imported column containing the transaction number;
3. $\langle\mathsf{CN}\rangle$: imported column containing the current execution context;
4. $\langle\mathsf{STO}\square\mathsf{REV}\rangle$: import of a context-number constant column; contains the storage time stamp at which a revert is to occur; the current context reverts *iff* $\langle\mathsf{STO}\square\mathsf{REV}\rangle \neq 0$.

Every execution context $\mathscr{C}$ has a **reverter context** $\mathscr{R}$ (if $\mathscr{C}$ does not revert its reverter is the special 0^{th} execution context.) Otherwise it is the nearest ancestor context of $\mathscr{C}$ which is *directly* responsible for a rollback. A context $\mathscr{C}$ may be its own reverter. We say that an execution context is **directly responsible for reverting** if its execution leads to an exception or a (successful) `REVERT` instruction.

The imported $\langle\mathsf{STO}\square\mathsf{REV}\rangle$ column contains 0 if the current context does not revert. Otherwise it contains the storage stamp at which the (present context's) reverter context $\mathscr{R}$ reverts. This time stamp is common to all contexts that have $\mathscr{R}$ as their reverter context. While every child context of $\mathscr{R}$ reverts they may do so for different reasons and at different $\langle\mathsf{STO}\square\mathsf{REV}\rangle$. Indeed, it can happen that a context which inherits a revert flag also produces its own exception. In this case it is its own reverter and defines its own revert time stamp (which it passes down to all its descendant contexts.)

5. $\langle\mathsf{INST}\rangle$: imported column containing the current instruction;
6. $\mathsf{VAL}^{\mathsf{hi}}$ and $\mathsf{VAL}^{\mathsf{lo}}$: contain the high and low part of a value in storage as the instruction starts execution;
7. $\langle\mathsf{VAL}^{\mathsf{hi}}\rangle^{\nu}$ and $\langle\mathsf{VAL}^{\mathsf{lo}}\rangle^{\nu}$: imported columns containing the value in storage after execution of the instruction;

Given the stack pattern for storage instructions, $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle^{\nu}$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle^{\nu}$ are imports of the 4th stack item ${}_{4}\mathsf{VAL}^{\mathsf{hi}}$ and ${}_{4}\mathsf{VAL}^{\mathsf{lo}}$. Note that $\mathsf{VAL}^{\mathsf{hi}}$ and $\mathsf{VAL}^{\mathsf{lo}}$ **aren't** imported columns. This is because $\langle \mathsf{VAL}^{\mathsf{hi}} \rangle^{\nu}$ and $\langle \mathsf{VAL}^{\mathsf{lo}} \rangle^{\nu}$ are the values that will find themselves on the stack after the instruction is done while $\mathsf{VAL}^{\mathsf{hi}}$ and $\mathsf{VAL}^{\mathsf{lo}}$ are the values in storage before anything happens. In case of an $\langle \mathsf{INST} \rangle = \mathtt{SLOAD}$ these values are the same. In case of an $\langle \mathsf{INST} \rangle = \mathtt{SSTORE}$ the imported values are meant to replace the pre-existing values.

8. $\langle \mathsf{STORAGE\_ADDRESS} \rangle^{\mathsf{hi}}$ and $\langle \mathsf{STORAGE\_ADDRESS} \rangle^{\mathsf{lo}}$: imported columns; contain the address of the contract whose storage may be altered by the current execution context; abbreviated to $\langle \mathsf{SADDR} \rangle^{\mathsf{hi}}$ and $\langle \mathsf{SADDR} \rangle^{\mathsf{lo}}$ respectively;

9. $\langle \mathsf{STORAGE\_KEY} \rangle^{\mathsf{hi}}$ and $\langle \mathsf{STORAGE\_KEY} \rangle^{\mathsf{lo}}$: imported columns; contain the storage key accessed by the current instruction; abbreviated to $\langle \mathsf{KEY} \rangle^{\mathsf{hi}}$ and $\langle \mathsf{KEY} \rangle^{\mathsf{lo}}$ respectively;

As in the case of values, the stack pattern of storage instructions imposes that $\langle \mathsf{KEY} \rangle^{\mathsf{hi}}$ and $\langle \mathsf{KEY} \rangle^{\mathsf{lo}}$ are imports of the first stack item ${}_{1}\mathsf{VAL}^{\mathsf{hi}}$ and ${}_{1}\mathsf{VAL}^{\mathsf{lo}}$. To simplify notations we may at times write $\langle \mathsf{SADDR} \rangle$ and $\langle \mathsf{KEY} \rangle$ to signify the pairs $(\langle \mathsf{SADDR} \rangle^{\mathsf{hi}}, \langle \mathsf{SADDR} \rangle^{\mathsf{lo}})$ and $(\langle \mathsf{KEY} \rangle^{\mathsf{hi}}, \langle \mathsf{KEY} \rangle^{\mathsf{lo}})$ respectively. We do this even when defining (variations of) lexicographic orders. Thus when we write $a_1 < a_2$ for addresses a_1 and a_2 it is to be understood as $a_1^{\mathsf{hi}} < a_2^{\mathsf{hi}}$ or $(a_1^{\mathsf{hi}} = a_2^{\mathsf{hi}}$ and $a_1^{\mathsf{lo}} < a_2^{\mathsf{lo}})$ and similarly for keys.

The values *initially* in storage are subject to constraints with the permanent state, as are the values that are last set for a given address and key pair. Identifying the relevant rows is the job of FACCF and LACCF detailed below.

10. $\mathsf{FIRST\_ACCESS\_FLAG}$: binary flag; lights up precisely once per batch of transactions and per (touched) storage key; lights up the first time that key is touched; abbreviated to FACCF;

11. $\mathsf{LAST\_ACCESS\_FLAG}$: binary flag; lights up precisely once per batch of transactions and per (touched) storage key; lights up the last time that key is touched; abbreviated to LACCF;

Pre-warmed storage keys that are never called upon by a storage instruction count as untouched. The columns below are required for gas metering and computing gas refunds.

12. $\mathsf{ORIGINAL\_VALUE}^{\mathsf{hi}}$ and $\mathsf{ORIGINAL\_VALUE}^{\mathsf{lo}}$: contain the value in storage at the beginning of a transaction (and thus may change from one transaction to another); abbreviated to $\mathsf{ORIG}^{\mathsf{hi}}$ and $\mathsf{ORIG}^{\mathsf{lo}}$ respectively;

13. $\langle \mathsf{STORAGE\_GAS} \rangle$: storage gas cost; abbreviated to $\langle \mathsf{STOG} \rangle$;

14. $\langle \mathsf{REFUND\_GAS} \rangle$: computes gas refunds which may be associated with an `SSTORE` instruction; abbreviated to $\langle \mathsf{REFG} \rangle$;

15. $\mathsf{REFUND\_DIRTY\_CLEAR}$: computes the $r_{\text{dirtyclear}}$ refund function from the Ethereum Yellow Paper; abbreviated to REFDC;

16. $\mathsf{REFUND\_DIRTY\_RESET}$: computes the $r_{\text{dirtyreset}}$ refund function from the Ethereum Yellow Paper; abbreviated to REFDR;

17. $\mathsf{PREWARM}$: binary flag indicating whether a storage key was prewarmed for a transaction;

18. WARM: binary flag indicating whether a storage key is warm within the execution of a transaction;

The $\underrightarrow{\mathsf{DOM}\square}$ and $\underleftarrow{\mathsf{SUB}\square}$ columns below play a technical role in reverting contexts. They are used to unwind the successive changes made to storage. The subordinate stamp colum $\underleftarrow{\mathsf{SUB}\square}$ will be endowed with the opposite order from the natural one whence the arrow pointing to the left adorning it.

19. $\underrightarrow{\mathsf{DOM}\square}$, $\underleftarrow{\mathsf{SUB}\square}$: “dominant” and “subordinate” stamp columns;

20. $\mathsf{COUNTER}$: binary column; always equal to 0 except for storage instructions in a reverting execution context; in this case it counts from 0 to 1; abbreviated to CT;

8.2 Constraints

8.2.1 Heartbeat

The heartbeat of the storage module is simple: the storage stamp grows by one with every row and the counter column is $\equiv 0$. The only exception to that rule happens when executing storage instructions in a reverting context. In this case every storage instruction occupies *two* rows (say i and $i+1$) with $\mathsf{CT}_i = 0$, $\mathsf{CT}_{i+1} = 1$ and $\mathsf{STO}\square_i = \mathsf{STO}\square_{i+1}$). Whether a context reverts or not can be read off the $\langle\mathsf{STO}\square\mathsf{REV}\rangle$: if it is nonzero then the context reverts, otherwise it doesn't.

1. $\langle\mathsf{STO}\square\rangle_0 = 0$
2. IF $\langle\mathsf{STO}\square\rangle_i = 0$ then
$$\begin{cases} \mathsf{CT}_i & = & 0 \\ \mathsf{CT}_{i+1} & = & 0 \\ \underrightarrow{\mathsf{DOM}\square}_i & = & 0 \\ \underleftarrow{\mathsf{SUB}\square}_i & = & 0 \end{cases}$$
3. $\forall i,\ \langle\mathsf{STO}\square\rangle_{i+1} \in \{\langle\mathsf{STO}\square\rangle_i, 1 + \langle\mathsf{STO}\square\rangle_i\}$ i.e. $\langle\mathsf{STO}\square\rangle$ is nondecreasing with jumps $= 1$;
4. IF $\langle\mathsf{STO}\square\rangle_{i+1} \neq \langle\mathsf{STO}\square\rangle_i$ THEN $\mathsf{CT}_{i+1} = 0$;
5. IF $\langle\mathsf{STO}\square\rangle_i \neq 0$
   (a) $\langle\mathsf{STO}\square\mathsf{REV}\rangle_i = 0$ THEN $\langle\mathsf{STO}\square\rangle_{i+1} = 1 + \langle\mathsf{STO}\square\rangle_i$
   (b) $\langle\mathsf{STO}\square\mathsf{REV}\rangle_i \neq 0$ THEN
$$\begin{cases} \text{IF } \mathsf{CT}_i = 0 \text{ THEN } \begin{cases} \mathsf{CT}_{i+1} = 1 \\ \langle\mathsf{STO}\square\rangle_{i+1} = \langle\mathsf{STO}\square\rangle_i \\ \langle\mathsf{INST}\rangle_{i+1} = \langle\mathsf{INST}\rangle_i \end{cases} \\ \text{IF } \mathsf{CT}_i = 1 \text{ THEN } \langle\mathsf{STO}\square\rangle_{i+1} = 1 + \langle\mathsf{STO}\square\rangle_i \end{cases}$$
   (Note that the constraint $\langle\mathsf{INST}\rangle_{i+1} = \langle\mathsf{INST}\rangle_i$, when $\langle\mathsf{STO}\square\mathsf{REV}\rangle_i \neq 0$ and $\mathsf{CT}_i = 0$, is redundant given the storage stamp remains the same)
6. IF $\Big(\langle\mathsf{INST}\rangle_N = \texttt{SSTORE}$ AND $\langle\mathsf{STO}\square\mathsf{REV}\rangle_N \neq 0\Big)$ THEN $\mathsf{CT}_N = 1$.

Note that the only instructions being loaded in are `SLOAD` and `SSTORE`. We may thus replace equality constraints such as “IF $\langle\mathsf{INST}\rangle_i = \texttt{SLOAD}$ THEN $\cdots$” with “IF $\langle\mathsf{INST}\rangle_i \neq \texttt{SSTORE}$ THEN $\cdots$”.

8.2.2 Prewarmed storage keys

Like in the warmth module (but unlike most other modules) rows with $\langle\mathsf{STO}\square\rangle_i = 0$ serve a double purpose: they are used both for **padding** and for **loading pre-warmed storage keys**. In constraints we will enforce that

$$\mathsf{PREWARM}_i = 1 \iff \Big(\langle\mathsf{STO}\square\rangle_i = 0 \ \text{ AND } \ \langle\mathsf{TX}\#\rangle_i \neq 0\Big)$$

and that a key is only pre-warmed once per transaction. In accordance with the first condition we impose that:

1. IF $\mathsf{PREWARM}_i = 1$ THEN $\Big(\langle\mathsf{STO□}\rangle_i = 0 \;$ AND $\langle\mathsf{TX\#}\rangle_i \neq 0\Big)$;
2. IF $\Big(\langle\mathsf{STO□}\rangle_i = 0 \;$ AND $\langle\mathsf{TX\#}\rangle_i \neq 0\Big)$ THEN $\mathsf{PREWARM}_i = 1$;

The first task of the storage module is to load all prewarmed storage keys. Justifying the prewarmed addresses is done by means of a bilateral plookup inclusion proof where on one side we have

$$\Big[\langle\mathsf{TX\#}\rangle, \langle\mathsf{SADDR}\rangle, \langle\mathsf{KEY}\rangle\Big] \odot \mathsf{PREWARM}$$

and on the other side of that bilateral plookup we have a commitment to the prewarmed storage keys per transaction. Prewarmed keys will be loaded at $\langle\mathsf{STO□}\rangle = 0$; this enforces that the corresponding rows will appear first in the relevant row reordering.

Note. We have again, for simplicity's sake, suppressed $(\,\cdot\,)^{\mathsf{hi}}$ and $(\,\cdot\,)^{\mathsf{lo}}$ in $\langle\mathsf{SADDR}\rangle$ and $\langle\mathsf{KEY}\rangle$.

Note. In the above we define $\mathsf{Z} \overset{\text{def.}}{=} \mathsf{X} \odot \mathsf{Y}$ to be the coordinate-wise product of the columns X and Y, i.e. the column vector with, for all i, $\mathsf{Z}_i = \mathsf{X}_i \cdot \mathsf{Y}_i$. We extend the notation to families of column vectors $\mathsf{X}^1, \mathsf{X}^2, \dots, \mathsf{X}^r$ thus writing $\big[\mathsf{X}^1, \dots, \mathsf{X}^r\big] \odot \mathsf{Y}$ rather than $\big[\mathsf{X}^1 \odot \mathsf{Y}, \dots, \mathsf{X}^r \odot \mathsf{Y}\big]$.

8.2.3 Instruction related constraints

The instruction related constraints depend on whether the current instruction will be reverted or not. For instance, and as mentioned in the heartbeat section, storage instructions in a reverting context occupy two rows. The associated constraints are as follows.

We first deal with constraints in a non-reverting context:

1. IF $\langle\mathsf{STO□REV}\rangle_i = 0$ THEN
   (a) We set $\underrightarrow{\mathsf{DOM□}}$ and $\underleftarrow{\mathsf{SUB□}}$:
   $$\begin{cases} \underrightarrow{\mathsf{DOM□}}_i = 2 \cdot \langle\mathsf{STO□}\rangle_i \\ \underleftarrow{\mathsf{SUB□}}_i = 2 \cdot \langle\mathsf{STO□}\rangle_i \end{cases}$$
   (b) IF $\langle\mathsf{INST}\rangle_i = \texttt{SLOAD}$ THEN
   $$\begin{cases} \mathsf{VAL}_i^{\mathsf{hi}} = \langle\mathsf{VAL}^{\mathsf{hi}}\rangle_i^{\nu} \\ \mathsf{VAL}_i^{\mathsf{lo}} = \langle\mathsf{VAL}^{\mathsf{lo}}\rangle_i^{\nu} \end{cases}$$

We next deal with contraints in a reverting context. We settle the expected behaviour at the first of two rows.

2. IF $\Big(\langle\mathsf{STO□REV}\rangle_i \neq 0 \;$ AND $\mathsf{CT}_i = 0\Big)$ THEN
   (a) The following always hold:
   $$\begin{cases} \underleftarrow{\mathsf{SUB□}}_i & = & 2 \cdot \langle\mathsf{STO□}\rangle_i \\ \underleftarrow{\mathsf{SUB□}}_{i+1} & = & 2 \cdot \langle\mathsf{STO□}\rangle_i \\ \underrightarrow{\mathsf{DOM□}}_i & = & 2 \cdot \langle\mathsf{STO□}\rangle_i \\ \underrightarrow{\mathsf{DOM□}}_{i+1} & = & 2 \cdot \langle\mathsf{STO□REV}\rangle_i + 1 \\ \mathsf{WARM}_{i+1} & = & \mathsf{WARM}_i \end{cases}$$
   (b) IF $\langle\mathsf{INST}\rangle_i = \texttt{SSTORE}$ THEN
   $$\begin{cases} \mathsf{VAL}_{i+1}^{\mathsf{hi}} & = & \langle\mathsf{VAL}^{\mathsf{hi}}\rangle_i^{\nu} \\ \mathsf{VAL}_{i+1}^{\mathsf{lo}} & = & \langle\mathsf{VAL}^{\mathsf{lo}}\rangle_i^{\nu} \\ \langle\mathsf{VAL}^{\mathsf{hi}}\rangle_{i+1}^{\nu} & = & \mathsf{VAL}_i^{\mathsf{hi}} \\ \langle\mathsf{VAL}^{\mathsf{lo}}\rangle_{i+1}^{\nu} & = & \mathsf{VAL}_i^{\mathsf{lo}} \end{cases}$$
   i.e. the “old” and “new” values are “swapped” from one line to the next.

(c) IF $\langle\mathsf{INST}\rangle_i = \mathtt{SLOAD}$ THEN

$$\left\{\begin{array}{l} \mathsf{VAL}^{\mathsf{hi}}_{i+1} = \mathsf{VAL}^{\mathsf{hi}}_{i} = \langle \mathsf{VAL}^{\mathsf{hi}}\rangle^{\nu}_{i} = \langle \mathsf{VAL}^{\mathsf{hi}}\rangle^{\nu}_{i+1} \\ \mathsf{VAL}^{\mathsf{lo}}_{i+1} = \mathsf{VAL}^{\mathsf{lo}}_{i} = \langle \mathsf{VAL}^{\mathsf{lo}}\rangle^{\nu}_{i} = \langle \mathsf{VAL}^{\mathsf{lo}}\rangle^{\nu}_{i+1} \end{array}\right.$$

The figure below captures the expected behaviour of the storage module execution trace in a reverting context. In it we focus solely on a give storage address and storage key.

⟨CN⟩	⟨SA⟩	⟨KEY⟩	⟨S□R⟩	⟨STO□⟩	⟨INST⟩	CT	$\underrightarrow{\text{DOM□}}$	$\underleftarrow{\text{SUB□}}$	VAL	$\langle\mathsf{VAL}\rangle^{\nu}$	WARM
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
c	addr	key	$\mathsf{rev_c}$	s	SLOAD	0	$2 \cdot \mathsf{s}$	$2 \cdot \mathsf{s}$	v_0	v_0	w
c	addr	key	$\mathsf{rev_c}$	s	SLOAD	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot \mathsf{s}$	v_0	v_0	w
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + 1$	SSTORE	0	$2 \cdot (\mathsf{s} + 1)$	$2 \cdot (\mathsf{s} + 1)$	v_0	v	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + 1$	SSTORE	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s} + 1)$	v	v_0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + k$	SSTORE	0	$2 \cdot (\mathsf{s} + k)$	$2 \cdot (\mathsf{s} + k)$	v	v'	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + k$	SSTORE	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s} + k)$	v'	v	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + l$	SLOAD	0	$2 \cdot (\mathsf{s} + l)$	$2 \cdot (\mathsf{s} + l)$	v'	v'	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + l$	SLOAD	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s} + l)$	v'	v'	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + m$	SLOAD	0	$2 \cdot (\mathsf{s} + m)$	$2 \cdot (\mathsf{s} + m)$	v'	v'	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + m$	SLOAD	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s} + m)$	v'	v'	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + m + 1$	SSTORE	0	$2 \cdot (\mathsf{s} + m + 1)$	$2 \cdot (\mathsf{s} + m + 1)$	v'	v''	1
c	addr	key	$\mathsf{rev_c}$	$\mathsf{s} + m + 1$	SSTORE	1	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s} + m + 1)$	v''	v'	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 8.1: The above represents a series of consecutive storage instructions executed in a reverting ($\mathsf{rev_c} \neq 0$) execution context (c) touching a given storage key (key) of a particular contract account (with address addr.) Given that the present context reverts, (storage) instructions occupy 2 lines each. There are three of them and three SLOAD instructions. The initial value of the WARM flag is $\mathsf{w} \in \{0, 1\}$. In the above $1 < k < l < m$ and $m + 1 \leq \mathsf{rev_c}$.
We have used abbreviations to allow for more columns on a single page. Thus ⟨SA⟩ is short hand for ⟨SADDR⟩ and ⟨S□R⟩ is short hand for ⟨STO□REV⟩.

One may wonder why SLOAD instructions in reverting contexts also occupy two rows when all the information they contain is duplicated. The answer lies in reverting the WARM flag: in terms of unwinding modifications made to the value stored at a particular storage key if SLOAD instructions (in reverting contexts) there would be no harm in making SLOAD instructions occupy a single line. The issue arises with the WARM flag. The advantage of always using two rows to represent storage instructions in reverting contexts is that the (original value of the) warmth flag finds itself both at the beginning and at the end of the reordered sequence of modifications done to a particular address independently of what storage instruction is executed first.

The figure below captures the expected behaviour of the reordered execution trace in a reverting context:

$[\langle\mathsf{KEY}\rangle]$🔀	$[\langle\mathsf{SADDR}\rangle]$🔀	$\left[\underrightarrow{\mathsf{DOM}\square}\right]$🔀	$\left[\underleftarrow{\mathsf{SUB}\square}\right]$🔀	$[\mathsf{VAL}]$🔀	$[\langle\mathsf{VAL}\rangle^{\nu}]$🔀	$[\mathsf{WARM}]$🔀
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
key	addr	$2 \cdot \mathsf{s}$	$2 \cdot \mathsf{s}$	v_0	v_0	w
key	addr	$2 \cdot (\mathsf{s}+1)$	$2 \cdot (\mathsf{s}+1)$	v_0	v	1
key	addr	$2 \cdot (\mathsf{s}+k)$	$2 \cdot (\mathsf{s}+k)$	v	v'	1
key	addr	$2 \cdot (\mathsf{s}+l)$	$2 \cdot (\mathsf{s}+l)$	v'	v'	1
key	addr	$2 \cdot (\mathsf{s}+m)$	$2 \cdot (\mathsf{s}+m)$	v'	v'	1
key	addr	$2 \cdot (\mathsf{s}+m+1)$	$2 \cdot (\mathsf{s}+m+1)$	v'	v''	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s}+m+1)$	v''	v'	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s}+m)$	v'	v'	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s}+l)$	v'	v'	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s}+k)$	v'	v	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot (\mathsf{s}+1)$	v	v_0	1
key	addr	$2 \cdot \mathsf{rev_c} + 1$	$2 \cdot \mathsf{s}$	v_0	v_0	w
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 8.2: The above represents the same sequence of storage instructions but *reordered*. The "first rows" of `SLOAD` and `SSTORE` instructions executed in a reverting execution context appear in the same order as they do in the time ordered execution trace (but in immediate succession, i.e. without gaps). The "second rows" appear at the tail end of the rows containing all accesses to the storage key key of the account with address addr (and any access to the same key happening in a descendant context with the same reverter.) They all have the same $[\langle\mathsf{STO}\square\rangle]$🔀 (which is equal to $\langle\mathsf{STO}\square\mathsf{REV}\rangle$). The updates to the value in storage are being "unwound" in reverse chronological order thanks to $\left[\underleftarrow{\mathsf{SUB}\square}\right]$🔀 having the opposite order to the natural one.

8.3 Consistency

8.3.1 Batch level consistency

This section deals with constraints that capture batch-wide behaviours and properties of storage. These include:

1. updating the value stored at a particular storage key across all transactions in the batch that touch that storage key;

2. reverting said updates if the context that induced them reverts;

3. recognizing the first and last time a storage key is *properly* touched by a storage instruction;

The adverb *properly* is meant to distinguish "proper" or "real, instruction induced" accesses to storage keys from pre-warming-related "non-accesses". To make the distinction clear: the execution trace of the storage module includes rows (with nonzero $\langle \mathsf{TX\#} \rangle$ but zero $\langle \mathsf{STO}\square \rangle$) that aren't induced by a storage instruction yet contain a nonzero transaction number, a storage address and a storage key. Rows that are induced by an instructions have $\langle \mathsf{STO}\square \rangle \neq 0$.

We introduce a row permutation which groups together all "accesses", proper or not, across all transactions in the batch, to a given storage key k of a given storage address a. We show in figure ?? the desired effect this ordering has on storage in reverting contexts. This row permutation should be such that, for fixed values of $\langle \mathsf{SADDR} \rangle = \mathsf{a}$ and $\langle \mathsf{KEY} \rangle = \mathsf{k}$:

1. the rows with $\langle \mathsf{SADDR} \rangle = \mathsf{a}$ and $\langle \mathsf{KEY} \rangle = \mathsf{k}$ form a contiguous block;

2. the pre-warming rows (if any) are listed at the beginning of this block;

3. these may be followed by a succession of:

   (a) first rows, in chronological order, of storage instructions, with $\langle \mathsf{STO}\square \rangle < \langle \mathsf{STO}\square\mathsf{REV} \rangle$;

   (b) second rows, in reverse chronological order, of a subset of the previous storage instructions, all with $\langle \mathsf{STO}\square \rangle = \langle \mathsf{STO}\square\mathsf{REV} \rangle$ for the same value or $\langle \mathsf{STO}\square\mathsf{REV} \rangle$.

The "first row" of a storage instruction is characterized by $\mathsf{CT} = 0$; the "second row" of a storage instruction, if present, is characterized by $\mathsf{CT} = 1$; the second rows unwind storage instruction performed in a reverting context. Listing these second rows in reverse chronological order is what allows to zk-evm to "unwind" storage instructions.

Consider a row permutation $\mathsf{X} \mapsto [\mathsf{X}]^{\text{⤮}}$ such that the rows of the tuple of columns

$$\left([\langle \mathsf{SADDR} \rangle]^{\text{⤮}}, [\langle \mathsf{KEY} \rangle]^{\text{⤮}}, \left[\underrightarrow{\mathsf{DOM}\square}\right]^{\text{⤮}}, \left[\underleftarrow{\mathsf{SUB}\square}\right]^{\text{⤮}}, \right)$$

follow the following variation $\prec$ on the lexicographic order:

$$(a, k, s, r) \prec (a', k', s', r') \iff \begin{cases} a < a' & & & & \text{OR} \\ a = a' & \text{AND } k < k' & & & \text{OR} \\ a = a' & \text{AND } k = k' & \text{AND } s < s' & & \text{OR} \\ a = a' & \text{AND } k = k' & \text{AND } s = s' & \text{AND } r > r' & \end{cases}$$

The fact that the final comparison is reversed explains our notation for $\underleftarrow{\mathsf{SUB}\square}$. We impose the following consistency constraints:

1. $\text{IF } \langle \mathsf{STO}\square \rangle_i = 0 \text{ THEN } \mathsf{FACCF}_i = \mathsf{LACCF}_i = 0$

2. $\text{IF } \left([\langle \mathsf{SADDR} \rangle]^{\text{⤮}}_{i+1} \neq [\langle \mathsf{SADDR} \rangle]^{\text{⤮}}_{i} \text{ OR } [\langle \mathsf{KEY} \rangle]^{\text{⤮}}_{i+1} \neq [\langle \mathsf{KEY} \rangle]^{\text{⤮}}_{i} \right) \text{ THEN}$

$$\begin{cases} \text{IF } [\langle \mathsf{STO}\square \rangle]^{\text{⤮}}_{i+1} \neq 0 \text{ THEN } [\mathsf{FACCF}]^{\text{⤮}}_{i+1} = 1 \\ \text{IF } [\langle \mathsf{STO}\square \rangle]^{\text{⤮}}_{i} \neq 0 \text{ THEN } [\mathsf{LACCF}]^{\text{⤮}}_{i} = 1 \end{cases}$$

3. IF $\left([\langle\mathsf{SADDR}\rangle]^{\text{⤨}}_{i+1} = [\langle\mathsf{SADDR}\rangle]^{\text{⤨}}_{i} \;\text{ AND }\; [\langle\mathsf{KEY}\rangle]^{\text{⤨}}_{i+1} = [\langle\mathsf{KEY}\rangle]^{\text{⤨}}_{i}\right)$ THEN

$$\begin{cases} \text{IF } \left([\langle\mathsf{STO}\square\rangle_i]^{\text{⤨}} = 0 \;\text{ AND }\; [\langle\mathsf{STO}\square\rangle_{i+1}]^{\text{⤨}} \neq 0\right) \text{ THEN } [\mathsf{FACCF}_{i+1}]^{\text{⤨}} = 1 \\ \text{IF } [\langle\mathsf{STO}\square\rangle_i]^{\text{⤨}} \neq 0 \text{ THEN } \begin{cases} \left[\mathsf{VAL}^{\mathsf{hi}}\right]^{\text{⤨}}_{i+1} = \left[\langle\mathsf{VAL}^{\mathsf{hi}}\rangle^{\nu}\right]^{\text{⤨}}_{i} \\ \left[\mathsf{VAL}^{\mathsf{lo}}\right]^{\text{⤨}}_{i+1} = \left[\langle\mathsf{VAL}^{\mathsf{lo}}\rangle^{\nu}\right]^{\text{⤨}}_{i} \\ [\mathsf{FACCF}]^{\text{⤨}}_{i+1} = 0 \\ [\mathsf{LACCF}]^{\text{⤨}}_{i} = 0 \end{cases} \end{cases}$$

The condition means that at row i and $i+1$ we are accessing the same storage key. Therefore neither $\mathsf{FIRST\_ACCESS\_FLAG}_{i+1}$ nor $\mathsf{LAST\_ACCESS\_FLAG}_{i}$ should be set.

4. IF $[\langle\mathsf{STO}\square\rangle]^{\text{⤨}}_{N} \neq 0$ THEN $[\mathsf{LACCF}]^{\text{⤨}}_{N} = 1$.

8.3.2 Transaction level consistency

We introduce a second row permutation to deal with consistency constraints at the transaction level. We require a row permutation that will group all rows that touch a particular storage key within a given transaction into a block of contiguous rows. To that end, consider a row reordering $\mathsf{X} \mapsto [\mathsf{X}]^{\text{⤨}}$ such that the rows of the tuple of columns

$$\left([\langle\mathsf{TX\#}\rangle]^{\text{⤨}}, [\langle\mathsf{SADDR}\rangle]^{\text{⤨}}, [\langle\mathsf{KEY}\rangle]^{\text{⤨}}, \left[\underrightarrow{\mathsf{DOM}\square}\right]^{\text{⤨}}, \left[\underleftarrow{\mathsf{SUB}\square}\right]^{\text{⤨}},\right)$$

follow the following variation on the standard lexicographic order:

$$(t,a,k,s,r) \prec (t',a',k',s',r') \iff \begin{cases} t < t' & \text{OR} \\ t = t' \;\text{ AND }\; a < a' & \text{OR} \\ t = t' \;\text{ AND }\; a = a' \;\text{ AND }\; k < k' & \text{OR} \\ t = t' \;\text{ AND }\; a = a' \;\text{ AND }\; k = k' \;\text{ AND }\; s < s' & \text{OR} \\ t = t' \;\text{ AND }\; a = a' \;\text{ AND }\; k = k' \;\text{ AND }\; s = s' \;\text{ AND }\; r > r' \end{cases}$$

We use this order to set the context entry flag for each storage slot. This flag is used in the temporal trace to set the value at context entry of any storage key that is sollicited within the execution of a particular context. We first enforce that

1. IF

$$\left\{\begin{array}{lcll} [\langle\mathsf{TX\#}\rangle]^{\text{⤨}}_{i+1} & \neq & [\langle\mathsf{TX\#}\rangle]^{\text{⤨}}_{i} & \text{OR} \\ [\langle\mathsf{SADDR}\rangle]^{\text{⤨}}_{i+1} & \neq & [\langle\mathsf{SADDR}\rangle]^{\text{⤨}}_{i} & \text{OR} \\ [\langle\mathsf{KEY}\rangle]^{\text{⤨}}_{i+1} & \neq & [\langle\mathsf{KEY}\rangle]^{\text{⤨}}_{i} & \end{array}\right\}$$

THEN

$$\begin{cases} [\mathsf{WARM}]^{\text{⤨}}_{i+1} = [\mathsf{PREWARM}]^{\text{⤨}}_{i+1} \\ \text{IF } [\langle\mathsf{STO}\square\rangle_{i+1}]^{\text{⤨}} \neq 0 \text{ THEN } \begin{cases} \left[\mathsf{ORIG}^{\mathsf{hi}}_{i+1}\right]^{\text{⤨}} = \left[\mathsf{VAL}^{\mathsf{hi}}_{i+1}\right]^{\text{⤨}} \\ \left[\mathsf{ORIG}^{\mathsf{lo}}_{i+1}\right]^{\text{⤨}} = \left[\mathsf{VAL}^{\mathsf{lo}}_{i+1}\right]^{\text{⤨}} \end{cases} \end{cases}$$

In other words: when entering a domain of rows pertaining either to the next transaction, a different storage address or a different storage key, the initial warmth of the storage key is determined by the PREWARM

2. IF

$$\left\{\begin{array}{llll} [\langle\mathsf{TX\#}\rangle]_i^{\text{⤨}} & \neq & 0 & \text{AND} \\ [\langle\mathsf{TX\#}\rangle]_{i+1}^{\text{⤨}} & = & [\langle\mathsf{TX\#}\rangle]_i^{\text{⤨}} & \text{AND} \\ [\langle\mathsf{SADDR}\rangle]_{i+1}^{\text{⤨}} & = & [\langle\mathsf{SADDR}\rangle]_i^{\text{⤨}} & \text{AND} \\ [\langle\mathsf{KEY}\rangle]_{i+1}^{\text{⤨}} & = & [\langle\mathsf{KEY}\rangle]_i^{\text{⤨}} & \end{array}\right\}$$

THEN

$$\begin{cases} \text{IF } [\langle\mathsf{STO□}\rangle_i]^{\text{⤨}} = 0 \text{ THEN } \begin{cases} [\langle\mathsf{STO□}\rangle_{i+1}]^{\text{⤨}} \neq 0 & (1) \\ [\mathsf{WARM}]_{i+1}^{\text{⤨}} = 1 & (3) \\ \left[\mathsf{ORIG}_{i+1}^{\mathsf{hi}}\right]^{\text{⤨}} = \left[\mathsf{VAL}_{i+1}^{\mathsf{hi}}\right]^{\text{⤨}} & (4) \\ \left[\mathsf{ORIG}_{i+1}^{\mathsf{lo}}\right]^{\text{⤨}} = \left[\mathsf{VAL}_{i+1}^{\mathsf{lo}}\right]^{\text{⤨}} & (4) \end{cases} \\ \text{IF } [\langle\mathsf{STO□}\rangle_i]^{\text{⤨}} \neq 0 \text{ THEN } \begin{cases} \text{IF } \left([\mathsf{CT}]_i^{\text{⤨}} = 1 \text{ AND } [\mathsf{CT}]_{i+1}^{\text{⤨}} = 0\right) \text{ THEN } [\mathsf{WARM}]_{i+1}^{\text{⤨}} = [\mathsf{WARM}]_i^{\text{⤨}} & (6) \\ \text{IF } \left([\mathsf{CT}]_i^{\text{⤨}} = 0 \text{ AND } [\mathsf{CT}]_{i+1}^{\text{⤨}} = 0\right) \text{ THEN } [\mathsf{WARM}]_{i+1}^{\text{⤨}} = 1 & (7) \\ \left[\mathsf{ORIG}_{i+1}^{\mathsf{hi}}\right]^{\text{⤨}} = \left[\mathsf{ORIG}_{i}^{\mathsf{hi}}\right]^{\text{⤨}} & (8) \\ \left[\mathsf{ORIG}_{i+1}^{\mathsf{lo}}\right]^{\text{⤨}} = \left[\mathsf{ORIG}_{i}^{\mathsf{lo}}\right]^{\text{⤨}} & (8) \end{cases} \end{cases}$$

In the above (1) signifies that, when present, there is a single pre-warming line per transaction, storage address and storage key; (3) signifies that a pre-warmed storage key starts out warm; (4) sets the original value in storage for that transaction (which is required for SSTORE gas metering); (8) propagates the original value; (6) recognizes the fact that rows i with $[\mathsf{CT}]_i^{\text{⤨}} = 1$ are the second row of a storage instruction in a reverting context and that when a storage instruction is reverted the warmth of the touched storage key is reverted to its value prior to the instruction; (7) recognizes the fact that if both $[\mathsf{CT}]_{i+1}^{\text{⤨}} = [\mathsf{CT}]_i^{\text{⤨}} = 0$ then the access at (reordered row index) $i+1$ follows an (as of yet) unreverted access so that the storage key is warm. The precondition $[\langle\mathsf{STO□}\rangle_i]^{\text{⤨}} \neq 0$ implies that the present line isn't the first access to that storage slot, and thus the zk-evm must set WARM_{i+1} to true.

Note that we could have alternatively used the constraints

$$\left([\mathsf{CT}]_i^{\text{⤨}} = 1 \text{ AND } [\mathsf{CT}]_{i+1}^{\text{⤨}} = 0\right) \iff \left\{\begin{array}{c} \left[\underrightarrow{\mathsf{DOM□}}\right]_i^{\text{⤨}} \neq \left[\underleftarrow{\mathsf{SUB□}}\right]_i^{\text{⤨}} \\ \text{AND} \\ \left[\underrightarrow{\mathsf{DOM□}}\right]_{i+1}^{\text{⤨}} = \left[\underleftarrow{\mathsf{SUB□}}\right]_{i+1}^{\text{⤨}} \end{array}\right\}$$

and

$$\left([\mathsf{CT}]_i^{\text{⤨}} = 0 \text{ AND } [\mathsf{CT}]_{i+1}^{\text{⤨}} = 0\right) \iff \left\{\begin{array}{c} \left[\underrightarrow{\mathsf{DOM□}}\right]_i^{\text{⤨}} = \left[\underleftarrow{\mathsf{SUB□}}\right]_i^{\text{⤨}} \\ \text{AND} \\ \left[\underrightarrow{\mathsf{DOM□}}\right]_{i+1}^{\text{⤨}} = \left[\underleftarrow{\mathsf{SUB□}}\right]_{i+1}^{\text{⤨}} \end{array}\right\}$$

8.3.3 Gas constraints

We now move on to computing ⟨STORAGE_GAS⟩ and ⟨REFUND_GAS⟩. We use the following notations lifted from the Ethereum Yellow Paper:

1. $G_{\text{warmaccess}} = 100$
2. $G_{\text{coldsload}} = 2100$
3. $G_{\text{sset}} = 20000$
4. $G_{\text{sreset}} = 2900$
5. $R_{\text{sclear}} = 15000$

$$\boxed{\text{All constraints below assume that } \langle \mathsf{STO\square} \rangle_i \neq 0 \text{ and } \mathsf{CT}_i = 0.}$$

The STOG column contains the gas cost of an individual operation. It depends on the instruction and other criteria.

1. IF $\langle \mathsf{INST} \rangle_i = \texttt{SLOAD}$ THEN

$$\begin{cases} \text{IF } \mathsf{WARM}_i = 1 \text{ THEN } \mathsf{STOG}_i = G_{\text{warmaccess}} \\ \text{IF } \mathsf{WARM}_i = 0 \text{ THEN } \mathsf{STOG}_i = G_{\text{coldsload}} \end{cases}$$

   i.e. $\mathsf{STOG}_i = G_{\text{warmaccess}} \cdot \mathsf{WARM}_i + G_{\text{coldsload}} \cdot (1 - \mathsf{WARM}_i)$.

2. IF $\langle \mathsf{INST} \rangle_i = \texttt{SSTORE}$ THEN

   (a) IF

$$\begin{cases} \mathsf{VAL}^{\mathsf{hi}}_i = \langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle^{\nu}_i \\ \qquad \text{AND} \\ \mathsf{VAL}^{\mathsf{lo}}_i = \langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle^{\nu}_i \end{cases}$$

   THEN

$$\mathsf{STOG}_i = G_{\text{warmaccess}} + G_{\text{coldsload}} \cdot (1 - \mathsf{WARM}_i)$$

   (b) IF

$$\begin{cases} \mathsf{VAL}^{\mathsf{hi}}_i \neq \mathsf{ORIG}^{\mathsf{hi}}_i \\ \qquad \text{OR} \\ \mathsf{VAL}^{\mathsf{lo}}_i \neq \mathsf{ORIG}^{\mathsf{lo}}_i \end{cases}$$

   THEN

$$\mathsf{STOG}_i = G_{\text{warmaccess}} + G_{\text{coldsload}} \cdot (1 - \mathsf{WARM}_i)$$

   (c) IF

$$\begin{cases} \mathsf{VAL}^{\mathsf{hi}}_i = \mathsf{ORIG}^{\mathsf{hi}}_i & \text{AND} \\ \mathsf{VAL}^{\mathsf{lo}}_i = \mathsf{ORIG}^{\mathsf{lo}}_i & \text{AND} \\ \left\{ \begin{array}{l} \mathsf{VAL}^{\mathsf{hi}}_i \neq \langle \mathsf{VAL}^{\,\mathsf{hi}} \rangle^{\nu}_i \\ \qquad \text{OR} \\ \mathsf{VAL}^{\mathsf{lo}}_i \neq \langle \mathsf{VAL}^{\,\mathsf{lo}} \rangle^{\nu}_i \end{array} \right\} \end{cases}$$

   THEN

   i. IF $\left(\mathsf{ORIG}^{\mathsf{hi}}_i = 0 \text{ AND } \mathsf{ORIG}^{\mathsf{lo}}_i = 0 \right)$ THEN $\mathsf{STOG}_i = G_{\text{sset}} + G_{\text{coldsload}} \cdot (1 - \mathsf{WARM}_i)$

   ii. IF $\left(\mathsf{ORIG}^{\mathsf{hi}}_i \neq 0 \text{ OR } \mathsf{ORIG}^{\mathsf{lo}}_i \neq 0 \right)$ THEN $\mathsf{STOG}_i = G_{\text{sreset}} + G_{\text{coldsload}} \cdot (1 - \mathsf{WARM}_i)$

We now tackle $\langle \mathsf{REFUND\_GAS} \rangle$: $\langle \mathsf{REFG} \rangle$ computes the gas refund associated with `SSTORE` instructions.

1. IF $\langle \mathsf{STO\square REV} \rangle_i \neq 0$ THEN $\langle \mathsf{REFG} \rangle_i = 0$
2. IF $\langle \mathsf{STO\square REV} \rangle_i = 0$ THEN

(a) IF

$$\begin{cases} \mathsf{VAL}_i^{\mathsf{hi}} = \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \mathsf{VAL}_i^{\mathsf{lo}} = \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{cases}$$

THEN $\langle \mathsf{REFG} \rangle_i = 0$

(b) IF

$$\begin{cases} \left\{ \begin{array}{c} \mathsf{VAL}_i^{\mathsf{hi}} \neq \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \text{OR} \\ \mathsf{VAL}_i^{\mathsf{lo}} \neq \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{array} \right\} & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{hi}} = \mathsf{ORIG}_i^{\mathsf{hi}} & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{lo}} = \mathsf{ORIG}_i^{\mathsf{lo}} & \text{AND} \\ \left\{ \begin{array}{c} \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \neq 0 \\ \text{OR} \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \neq 0 \end{array} \right\} & \text{AND} \end{cases}$$

THEN $\langle \mathsf{REFG} \rangle_i = 0$

(c) IF

$$\begin{cases} \left\{ \begin{array}{c} \mathsf{VAL}_i^{\mathsf{hi}} \neq \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \text{OR} \\ \mathsf{VAL}_i^{\mathsf{lo}} \neq \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{array} \right\} & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{hi}} = \mathsf{ORIG}_i^{\mathsf{hi}} & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{lo}} = \mathsf{ORIG}_i^{\mathsf{lo}} & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} = 0 & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} = 0 & \end{cases}$$

THEN $\langle \mathsf{REFG} \rangle_i = R_{\text{sclear}}$

(d) IF

$$\begin{cases} \left\{ \begin{array}{c} \mathsf{VAL}_i^{\mathsf{hi}} \neq \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \text{OR} \\ \mathsf{VAL}_i^{\mathsf{lo}} \neq \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{array} \right\} & \text{AND} \\ \left\{ \begin{array}{c} \mathsf{VAL}_i^{\mathsf{hi}} \neq \mathsf{ORIG}_i^{\mathsf{hi}} \\ \text{OR} \\ \mathsf{VAL}_i^{\mathsf{lo}} \neq \mathsf{ORIG}_i^{\mathsf{lo}} \end{array} \right\} & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} = 0 & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} = 0 & \end{cases}$$

THEN $\langle \mathsf{REFG} \rangle_i = \mathsf{REFDC}_i + \mathsf{REFDR}_i$

At last, we constrain the REFDC and REFDR columns. We start with REFDC:

1. IF $\mathsf{ORIG}_i^{\mathsf{hi}} = \mathsf{ORIG}_i^{\mathsf{lo}} = 0$ THEN $\mathsf{REFDC}_i = 0$

2. IF

$$\left\{\begin{array}{ll} \left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} \neq 0 \\ \text{OR} \\ \mathsf{ORIG}_i^{\mathsf{lo}} \neq 0 \end{array}\right\} & \text{AND} \\ \left\{\begin{array}{c} \mathsf{VAL}_i^{\mathsf{hi}} \neq 0 \\ \text{OR} \\ \mathsf{VAL}_i^{\mathsf{lo}} \neq 0 \end{array}\right\} & \text{AND} \\ \left\{\begin{array}{c} \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i \neq 0 \\ \text{OR} \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i \neq 0 \end{array}\right\} & \end{array}\right.$$

   THEN $\mathsf{REFDC}_i = 0$

3. IF

$$\left\{\begin{array}{ll} \left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} \neq 0 \\ \text{OR} \\ \mathsf{ORIG}_i^{\mathsf{lo}} \neq 0 \end{array}\right\} & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{hi}} = 0 & \text{AND} \\ \mathsf{VAL}_i^{\mathsf{lo}} = 0 & \text{AND} \end{array}\right.$$

   THEN $\mathsf{REFDC}_i = -R_{\text{sclear}}$

4. IF

$$\left\{\begin{array}{ll} \left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} \neq 0 \\ \text{OR} \\ \mathsf{ORIG}_i^{\mathsf{lo}} \neq 0 \end{array}\right\} & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} = 0 & \text{AND} \\ \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} = 0 & \text{AND} \end{array}\right.$$

   THEN $\mathsf{REFDC}_i = R_{\text{sclear}}$

We deal with REFDR:

1. IF

$$\left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} \neq \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \text{OR} \\ \mathsf{ORIG}_i^{\mathsf{lo}} \neq \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{array}\right\}$$

   THEN $\mathsf{REFDR}_i = 0$

2. IF

$$\left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} = \langle \mathsf{VAL}^{\mathsf{hi}} \rangle_i^{\nu} \\ \mathsf{ORIG}_i^{\mathsf{lo}} = \langle \mathsf{VAL}^{\mathsf{lo}} \rangle_i^{\nu} \end{array}\right\}$$

   THEN $\mathsf{REFDR}_i = 0$

   (a) IF

$$\left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} = 0 \\ \mathsf{ORIG}_i^{\mathsf{lo}} = 0 \end{array}\right\}$$

   THEN $\mathsf{REFDR}_i = G_{\text{sset}} - G_{\text{warmaccess}}$

   (b) IF

$$\left\{\begin{array}{c} \mathsf{ORIG}_i^{\mathsf{hi}} \neq 0 \\ \text{OR} \\ \mathsf{ORIG}_i^{\mathsf{lo}} \neq 0 \end{array}\right\}$$

   THEN $\mathsf{REFDR}_i = G_{\text{sreset}} - G_{\text{warmaccess}}$

Chapter 9

Word comparison

9.1 Word comparison module

9.1.1 Introduction

The **word comparison module** deals with the word comparison instructions of the evm, that is:

- `LT`
- `GT`
- `SLT`
- `SGT`
- `EQ`
- `ISZERO`

9.1.2 Columns

We list the named columns of the word comparison module. The first two dictate its (simple) heartbeat.

1. $\langle \mathsf{WCP}\square \rangle$: imported column containing the word comparison stamp;
2. $\mathsf{ONE\_LINE\_INSTRUCTION}$: binary column; equals 1 *if and only if* $\langle \mathsf{INST} \rangle \in \{\mathtt{EQ}, \mathtt{ISZERO}\}$; abbreviated to OLI;
3. $\mathsf{COUNTER}$: either hovers around 0 or counts continuously up from 0 to **15** and then resets;
4. $\langle \mathsf{INST} \rangle$: imported column; contains the instruction;
5. $\langle \mathsf{ARGUMENT\_1\_HIGH} \rangle$, $\langle \mathsf{ARGUMENT\_1\_LOW} \rangle$: imported columns containing the high and low part of the first instruction argument respectively; abbreviated to $\langle \mathsf{ARG1}^{\mathsf{hi}} \rangle$, $\langle \mathsf{ARG1}^{\mathsf{lo}} \rangle$;
6. $\langle \mathsf{ARGUMENT\_2\_HIGH} \rangle$, $\langle \mathsf{ARGUMENT\_2\_LOW} \rangle$: imported columns containing the high and low part of the second instruction argument respectively; abbreviated to $\langle \mathsf{ARG2}^{\mathsf{hi}} \rangle$, $\langle \mathsf{ARG2}^{\mathsf{lo}} \rangle$;
7. $\langle \mathsf{RESULT\_HIGH} \rangle$, $\langle \mathsf{RESULT\_LOW} \rangle$: imported columns containing the high and low part of the instruction result respectively; abbreviated to $\langle \mathsf{RES}^{\mathsf{hi}} \rangle$, $\langle \mathsf{RES}^{\mathsf{lo}} \rangle$;

Given the stack pattern of the word comparison instructions $\langle \mathsf{ARG1}^{\mathsf{hi}} \rangle$, $\langle \mathsf{ARG1}^{\mathsf{lo}} \rangle$ are imports of ${}_1\mathsf{VAL}^{\mathsf{hi}}$ and ${}_1\mathsf{VAL}^{\mathsf{lo}}$, $\langle \mathsf{ARG2}^{\mathsf{hi}} \rangle$, $\langle \mathsf{ARG2}^{\mathsf{lo}} \rangle$ of ${}_3\mathsf{VAL}^{\mathsf{hi}}$ and ${}_3\mathsf{VAL}^{\mathsf{lo}}$, and $\langle \mathsf{RES}^{\mathsf{hi}} \rangle$, $\langle \mathsf{RES}^{\mathsf{lo}} \rangle$ of ${}_4\mathsf{VAL}^{\mathsf{hi}}$ and ${}_4\mathsf{VAL}^{\mathsf{lo}}$ respectively. This is compatible with the stack pattern of `ISZERO` whose third stack item is vacuous. Note furthermore that $\langle \mathsf{RES}^{\mathsf{hi}} \rangle$ is expected to be 0 and $\langle \mathsf{RES}^{\mathsf{lo}} \rangle$ should be binary.

8. $\mathsf{BYTE\_1}, \dots, \mathsf{BYTE\_6}$: byte columns;
9. $\mathsf{ACC\_1}, \dots, \mathsf{ACC\_6}$: "(byte) accumulator" columns;
10. $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$, $\llbracket 3 \rrbracket$, $\llbracket 4 \rrbracket$: four counter-constant binary columns;

9.2 Constraints

9.2.1 Heartbeat

The heartbeat of the word comparison module is simple: if $\langle \mathsf{WCP}\square \rangle_i \neq 0$ and $\mathsf{OLI} = 0$ then CT counts continuously from 0 to **15**, otherwise it is 0.

1. $\langle \mathsf{WCP}\square \rangle_0 = 0$;
2. $\langle \mathsf{WCP}\square \rangle$ is nondecreasing in the sense that $\langle \mathsf{WCP}\square \rangle_{i+1} \in \{\langle \mathsf{WCP}\square \rangle_i, 1 + \langle \mathsf{WCP}\square \rangle_i\}$;
3. IF $\langle \mathsf{WCP}\square \rangle_i = 0$ THEN the whole row is null;
4. IF $\langle \mathsf{WCP}\square \rangle_{i+1} \neq \langle \mathsf{WCP}\square \rangle_i$ THEN $\mathsf{CT}_{i+1} = 0$;
5. IF $\langle \mathsf{WCP}\square \rangle_i \neq 0$ THEN
   (a) IF $\mathsf{OLI}_i = 1$ THEN $\langle \mathsf{WCP}\square \rangle_{i+1} = 1 + \langle \mathsf{WCP}\square \rangle_i$;
   (b) IF $\mathsf{OLI}_i = 0$ THEN
      i. IF $\mathsf{CT}_i \neq \mathbf{15}$ THEN $\mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i$
      ii. IF $\mathsf{CT}_i = \mathbf{15}$ THEN $\langle \mathsf{WCP}\square \rangle_{i+1} = 1 + \langle \mathsf{WCP}\square \rangle_i$
6. IF $\mathsf{OLI} = 0$ THEN $\mathsf{CT}_N = \mathbf{15}$.

9.2.2 Counter constancy constraints

We declare a column X to be **counter-constant** if it satisfies

$$\mathsf{CT}_i \neq 0 \implies \mathsf{X}_i = \mathsf{X}_{i-1}.$$

We impose that the following bit columns $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$, $\llbracket 3 \rrbracket$, $\llbracket 4 \rrbracket$ be counter-constant. Note that imported columns are automatically counter-constant. Note furthermore that counter-constancy for $\llbracket 1 \rrbracket$ and $\llbracket 2 \rrbracket$ follows from section 9.2.5.

9.2.3 Byte decompositions, bytehood and binaryness

We enforce "byte accumulation constraints" for $k = 1, \dots, 6$:

1. IF $\mathsf{CT}_i = 0$ THEN $\mathsf{ACC\_k}_i = \mathsf{BYTE\_k}_i$;
2. IF $\mathsf{CT}_i \neq 0$ THEN $\mathsf{ACC\_k}_i = 256 \cdot \mathsf{ACC\_k}_{i-1} + \mathsf{BYTE\_k}_i$;

We further ask that $k = 1, \dots, 6$ the $\mathsf{BYTE\_k}$ columns contain bytes. We also ask that B, B, B, B be binary columns, i.e. $\mathsf{B}k \cdot (1 - \mathsf{B}k) \equiv 0$.

9.2.4 OLI constraints

We constrain the OLI column:

1. OLI is binary;
2. IF $\langle \mathsf{INST} \rangle_i = \mathtt{EQ}$ THEN $\mathsf{OLI}_i = 1$
3. IF $\langle \mathsf{INST} \rangle_i = \mathtt{ISZERO}$ THEN $\mathsf{OLI}_i = 1$
4. IF $\big(\langle \mathsf{INST} \rangle_i \neq \mathtt{EQ}$ AND $\langle \mathsf{INST} \rangle_i \neq \mathtt{ISZERO}\big)$ THEN $\mathsf{OLI}_i = 0$

9.2.5 Target constraints

We first settle the behaviour of the first two bit columns:

1. $\textsc{if}$ $\mathsf{WCP}\square_i \neq 0$ $\textsc{then}$

$$\begin{cases} \textsc{if}\ \langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i = \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i\ \textsc{then}\ [\![1]\!]_i = 1 \\ \textsc{if}\ \langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i \neq \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i\ \textsc{then}\ [\![1]\!]_i = 0 \\ \textsc{if}\ \langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i = \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i\ \textsc{then}\ [\![2]\!]_i = 1 \\ \textsc{if}\ \langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i \neq \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i\ \textsc{then}\ [\![2]\!]_i = 0 \end{cases}$$

We fix the targets of the accumulator columns:

2. $\textsc{if}$ $\mathsf{CT}_i = \mathbf{15}$ $\textsc{then}$

   (a) the first four accumulator columns provide the byte decompositions of the arguments, i.e.

$$\begin{cases} \mathsf{ACC\_1}_i &= \langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i \\ \mathsf{ACC\_2}_i &= \langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i \\ \mathsf{ACC\_3}_i &= \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i \\ \mathsf{ACC\_4}_i &= \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i \end{cases}$$

   (b) the remaining two accumulator columns compute certain nonnegative adjusted differences, i.e.

$$\begin{cases} \mathsf{ACC\_5}_i &= (2 \cdot [\![3]\!] - 1) \cdot \big(\langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i - \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i\big) - [\![3]\!]_i \\ \mathsf{ACC\_6}_i &= (2 \cdot [\![4]\!] - 1) \cdot \big(\langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i - \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i\big) - [\![4]\!]_i \end{cases}$$

   In other words:

$$\begin{cases} [\![3]\!]_i = 1 \iff \langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i > \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i \\ [\![3]\!]_i = 0 \iff \langle \mathsf{ARG1}^{\mathsf{hi}}\rangle_i \leq \langle \mathsf{ARG2}^{\mathsf{hi}}\rangle_i \\ [\![4]\!]_i = 1 \iff \langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i > \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i \\ [\![4]\!]_i = 0 \iff \langle \mathsf{ARG1}^{\mathsf{lo}}\rangle_i \leq \langle \mathsf{ARG2}^{\mathsf{lo}}\rangle_i \end{cases}$$

9.2.6 Result constraints

We fix the behaviour of the result columns. The first one is always zero:

1. $\langle \mathsf{RES}^{\mathsf{hi}}\rangle_i = 0$: this constraints verifies the nullity of the high part of the result that finds itself on the stack;

The behaviour of $\langle \mathsf{RES}^{\mathsf{lo}}\rangle$ is instruction dependent:

2. $\textsc{if}$ $\langle \mathsf{WCP}\square\rangle_i \neq 0$ $\textsc{then}$

   (a) $\textsc{if}$ $\mathsf{OLI}_i = 1$ $\textsc{then}$ $\langle \mathsf{RES}^{\mathsf{lo}}\rangle_i = \mathsf{eq}_i$

   (b) $\textsc{if}$ $\mathsf{OLI}_i = 0$ $\textsc{then}$

      i. $\textsc{if}$ $\langle \mathsf{INST}\rangle_i = \mathtt{SLT}$ $\textsc{then}$ $\langle \mathsf{RES}^{\mathsf{lo}}\rangle_i = \mathsf{slt}_i$

      ii. $\textsc{if}$ $\langle \mathsf{INST}\rangle_i = \mathtt{LT}$ $\textsc{then}$ $\langle \mathsf{RES}^{\mathsf{lo}}\rangle_i = \mathsf{lt}_i$

      iii. $\textsc{if}$ $\langle \mathsf{INST}\rangle_i = \mathtt{SGT}$ $\textsc{then}$ $\langle \mathsf{RES}^{\mathsf{lo}}\rangle_i = 1 - \mathsf{lt}_i$

      iv. $\textsc{if}$ $\langle \mathsf{INST}\rangle_i = \mathtt{GT}$ $\textsc{then}$ $\langle \mathsf{RES}^{\mathsf{lo}}\rangle_i = 1 - \mathsf{slt}_i$

   where we use the short-hands $\mathsf{eq}_i := [\![1]\!]_i \cdot [\![2]\!]_i$, $\mathsf{slt}_i := [\![3]\!]_i + [\![1]\!]_i \cdot [\![4]\!]_i$ and $\mathsf{lt}_i := \mathsf{eq}_i + \mathsf{slt}_i$.

Chapter 10

Binary

10.1 Constraint set for the Binary module.

We list Binary module specific terms and where to find their definitions: **pivot-instructions** 10.1.1 and **shift-instructions** 10.1.1, **COUNTER-cycle** 10.1.2, **locally-constant column** 10.1.4 and **stamp-constant column** 10.1.4, **micro-shift COUNTER-cycle** and **macro-shift COUNTER-cycles** of shift-instructions.

Some constraints are repeats. We have highlighted them like so.

10.1.1 Binary Instructions

In this module we deal with the following instructions:

- AND
- OR
- XOR
- NOT
- SHL
- SHR
- SAR
- SGNX
- BYTE

(**Note.** We write SGNX to mean SIGNEXTEND) Most complexity comes from the **pivot-instructions** (i.e. SIGNEXTEND and BYTE) and the **shift-instructions** (i.e. SHL, SHR, and SAR.) The term "pivot-instruction" was chosen because the execution of both SIGNEXTEND and BYTE requires extracting a particular byte (the PIVOT_BYTE) from the second argument of the instruction which then plays a key role.

10.1.2 Columns

Our arithmetization uses the following columns:

1. COUNTER: goes from 31 to 0, decreasing by 1 with every row, and reseting to 31 after hitting 0; must start with 31 and end with 0; abbreviated to CT;

We call **COUNTER-cycle** any set of 32 consecutive rows where the first value of COUNTER is 31 (and its final value is 0). Instructions operate byte by byte, and thus take a multiples of 32 rows to execute: AND, OR, XOR, NOT, SGNX and BYTE take 1 COUNTER-cycle to execute, SHL, SHR and SAR take 6. Columns that remain constant along COUNTER-cycles are called **locally-constant**.

The following are three locally-constant bit columns. They are used during the macro-shift COUNTER-cycles of shift-instructions.

2. BIT_0: locally-constant binary column; abbreviated to $\llbracket 0 \rrbracket$; we set $\llbracket 0 \rrbracket^\vee = 1 - \llbracket 0 \rrbracket$;

3. BIT_1: locally-constant binary column; abbreviated to $\llbracket 1 \rrbracket$; we set $\llbracket 1 \rrbracket^\vee = 1 - \llbracket 1 \rrbracket$;

4. BIT_2: locally-constant binary column; abbreviated to $[\![2]\!]$; we set $[\![2]\!]^\vee = 1 - [\![2]\!]$;

A row index i with $\mathsf{COUNTER}_i = [\![0]\!]_i = [\![1]\!]_i = [\![2]\!]_i = 0$ marks the end of an instruction, and a new instruction starts at row $i + 1$. This coincides with jumps in BINARY_STAMP (see below). A non shift-instruction initializes these bits to $[\![2]\!][\![1]\!][\![0]\!] = 000$, a shift-instruction initializes them to $[\![2]\!][\![1]\!][\![0]\!] = 101$ (the binary digits of 5).

5. BINARY_STAMP: locally-constant column; increases by 1 with every new binary instruction; must start with 0; abbreviated to BS;

Columns that remain constant while BS remains constant are called **stamp-constant**.

The next batch of columns pertains to the (one or two) inputs and outputs (results) of an instruction and their byte decompositions.

6. INPUT_1: locally-constant column; contains the EVM word on top of the stack when the instruction begins; sometimes abbreviated to I1;

7. INPUT_2: locally-constant column; when $\mathtt{INST} \neq \mathtt{NOT}$ it is the second item on the stack from the top; when $\mathtt{INST} = \mathtt{NOT}$ it is zero; abbreviated to I2;

8. RES: locally-constant column; for non shift-instruction will contain the result of the instruction; for shift-instructions it contains the result of the instruction but only during the instruction's final COUNTER-cycle;

9. PREFIX_1: initialized with the leading byte of INPUT_1; grows by one byte with every row until COUNTER resets; abbreviated to P1;

10. PREFIX_2: initialized with the leading byte of INPUT_2; grows by one byte with every row until COUNTER resets; abbreviated to P2;

11. PREFIX_RES: initialized with the leading byte of RES; grows by one byte with every row until COUNTER resets; abbreviated to PR;

12. BYTE_1: bytes of INPUT_1 listed from most significative to least significative; abbreviated to B1;

13. BYTE_2: same but for INPUT_2; abbreviated to B2;

14. BYTE_RES: same but for RES; abbreviated to BR.

The following are technically useful columns.

15. ZERO_BACP: BACP stands for **B**eyond **A** **C**ertain **P**oint; any COUNTER-cycle of this column contain nonzero values (possibly none) followed by zeros (at least one); the amount of nonzero values in a COUNTER-cycle depends on ZERO_BACP_PARAM;

16. ZERO_BACP_PARAM: locally-constant column; value $\in \{0, \ldots, 31\}$ that marks the first time ZERO_BACP vanishes within the COUNTER-cycle; i.e. ZERO_BACP_PARAM $+ 1$ is the number of zeros in a COUNTER-cycle's worth of ZERO_BACP; sometimes abbreviated to ZP;

17. BYTE_BITS: binary column which plays a role in shift-instructions and pivot-instructions; abbreviated to BB;

18. PIVOT_BYTE: locally-constant column that contains a byte; used in pivot-instructions (if INPUT_1 is in range): for SGNX instructions it contains the byte containing the sign bit, for BYTE instructions it contains the byte that will be output in the end; abbreviated to PB;

Interpretation of BB. During the micro-shift COUNTER-cycle of a shift-instruction BB is 24 zeros followed by the 8 bits of INPUT_1 least significant byte. This COUNTER-cycle's worth of BB is repeated for the 5 following macro-shift COUNTER-cycles. For pivot-instructions the 16 final bits are the bit decompositions of the pivot PIVOT_BYTE followed by the bit decompositions of INPUT_1's least significant byte. For non shift-instructions and non pivot-instructions BB is 0.

Let us write $b_7, \ldots, b_1, b_0$ for the final 8 bits of BB in a given COUNTER-cycle. For shift-instructions and pivot-instructions these represent the 8 bits of the least significant byte of the first argument of the shift-instruction. The last three are combined together to form

$$\mathsf{LOW\_3} = 4 \cdot b_2 + 2 \cdot b_1 + b_0 \in \{0, 1, \ldots, 7\}.$$

NOTE. There is no LOW_3 column.

The **micro-shift** parameter μSHP (which controls bit shifting within bytes) is deduced from it (and the shift direction boolean SHD) as explained below. The remaining 5 bits in BYTE_BITS control branching behaviour of the 5 COUNTER-cycles that follow, i.e. the **macro-shift** COUNTER-cycles i.e. shifting by whole bytes. They are inserted in the DB column as needed.

For the pivot-instruction, the penultimate 8 bits of BB in the COUNTER-cycle are the bits of the pivot byte PB of the instruction, i.e. the byte of the second argument which contains the sign bit. The first one of these is thus the sign bit, and we store it in NEG (introduced below).

19. DECISION_BIT: locally-constant binary column; used in shift-instructions where it is made to contain, in succession, the 5 leading bits of the least significant byte of INPUT_1; abbreviated to DB;

20. NEG: stamp-constant binary column; used for SIGNEXTEND instructions where it contains the sign bit of the pivot byte;

21. IN_RANGE_FLAG: stamp-constant binary column; for shift-instructions it equals 1 if and only if INPUT_1 $\in [\![0, 256[\![$, i.e. if INPUT_1 equals its least significant byte; for pivot-instructions instructions it equals 1 if and only if INPUT_1 $\in [\![0, 32[\![$; abbreviated to IRF;

If SHIFT_FLAG $= 1$ and IN_RANGE_FLAG $= 0$ (i.e. if INPUT_1 ≥ 256) then we are shifting INPUT_2, a 256 bit integer, by at least 256 bits; for both SHL and SHR instructions this means that and the result is 0; for SAR the result is 0 if the sign bit (i.e. the leading bit of INPUT_2) is 0 while it is 0xfff$\cdots$ff (a string of 64 f's) if the sign bit is 1.

If PIVOT_FLAG $= 1$ and IN_RANGE_FLAG $= 0$ (i.e. if INPUT_1 ≥ 32) then the BYTE instruction returns 0 while theSGNX instruction returns INPUT_2 as is.

What follows are columns related to the micro-shift COUNTER-cycle of a shift-instruction.

22. μSHIFT_FLAG: locally-constant binary column; is zero except during the micro-shift COUNTER-cycle of a shift-instruction when it equals 1; abbreviated to μSHF;

NOTE. μSHF is redundant: it coincides with $[\![2]\!] \cdot [\![1]\!]^\vee \cdot [\![0]\!]$. We keep it around for sheer convenience.

23. μSHIFT_PARAM: stamp-constant column with values in $\in \{0, 1, \ldots, 7, 8\}$; holds the micro-shift parameter that determines the bit shift to apply to individual bytes during the micro-shift COUNTER-cycle of a shift-instruction; equals LOW_3 $\in \{0, 1, \ldots, 7\}$ if SHD $= 1$; equals $8 -$ LOW_3 $\in \{1, \ldots, 7, 8\}$ if SHD $= 0$; abbreviated to μSHP;

24. $^\Diamond$SPLIT_AND_SHIFTED_PREFIX: deduced from INPUT_2 and μSHP using a look up table; abbreviated to $^\Diamond$SNS_PREFIX;

25. $^\Diamond$SPLIT_AND_SHIFTED_SUFFIX: deduced from INPUT_2 and μSHP using a look up table; abbreviated to $^\Diamond$SNS_SUFFIX;

26. $^{\diamond}$ONES: deduced from INPUT_2 and μSHP using a look up table; it is used for SAR instructions; contains an integer from the set

$$\{00000000, 10000000, 11000000, 11100000, 11110000, 11111000, 11111100, 11111110\}$$

used to pad the first right shifted byte in case its leading bit is 1.

What follows are the instruction column and instruction decoded flag columns.

27. INST: stamp-constant column; contains instruction opcodes.
28. AND_FLAG: binary column deduced from INST by lookup table; lights up for AND instructions; abbreviated to ANDF;
29. OR_FLAG: binary column deduced from INST by lookup table; lights up for OR instructions; abbreviated to ORF;
30. XOR_FLAG: binary column deduced from INST by lookup table; lights up for XOR instructions; abbreviated to XORF;
31. NOT_FLAG: binary column deduced from INST by lookup table; lights up for NOT instructions; abbreviated to NOTF;
32. SHIFT_FLAG: binary column deduced from INST by lookup table; lights up shift-instructions, i.e. for SHL, SHR and SAR; abbreviated to SHF;
33. SHIFT_DIRECTION: binary column deduced from INST by lookup table; equals 1 for all instructions except for SHL when it equals 0; abbreviated to SHD;
34. SAR_FLAG: binary column deduced from INST by lookup table; lights up for SAR instructions; abbreviated to SARF;
35. PIVOT_FLAG: binary column deduced from INST by lookup table; lights up pivot-instructions, i.e. BYTE and SGNX; abbreviated to PF;
36. SIGNEXTEND_FLAG: binary column deduced from INST by lookup table; lights up for the SGNX instruction; abbreviated to SGNXF.

What follows are plookup obtained columns that contain the results of bit operations on pairs of bytes.

37. AND: contains the bit-wise AND of BYTE_1 and BYTE_2;
38. OR: contains the bit-wise OR of BYTE_1 and BYTE_2;
39. XOR: contains the bit-wise XOR of BYTE_1 and BYTE_2;
40. NOT: contains the bit-wise NOT of BYTE_1.

10.1.3 Lookup tables and Plookup constraints

Binary instruction decoder

The following columns are obtained by instruction decoding the INST column using the binary instruction decoder 10.1.

1. AND_FLAG
2. OR_FLAG
3. XOR_FLAG
4. NOT_FLAG,
5. SHIFT_FLAG
6. SHIFT_DIRECTION

Inst	ANDF	ORF	XORF	NOTF	SHF	SHD	SARF	PF	SGNXF
AND	**1**	0	0	0	0	**1**	0	0	0
OR	0	**1**	0	0	0	**1**	0	0	0
XOR	0	0	**1**	0	0	**1**	0	0	0
NOT	0	0	0	**1**	0	**1**	0	0	0
SHR	0	0	0	0	**1**	**1**	0	0	0
SAR	0	0	0	0	**1**	**1**	**1**	0	0
SHL	0	0	0	0	**1**	0	0	0	0
SGNX	0	0	0	0	0	**1**	0	**1**	**1**
BYTE	0	0	0	0	0	**1**	0	**1**	0

Figure 10.1: Binary instruction decoder.

7. SAR_FLAG,
8. PIVOT_FLAG
9. SIGNEXTEND_FLAG.

NOTE. These are (technically) stamp-constant binary columns, but since they are obtained by instruction decoding there is no need to enforce either of these constraints.

AND, OR, XOR and NOT lookup table

The values in the AND, OR, XOR and NOT columns are deduced from the bytes in the BYTE_1 and BYTE_2 columns by means of a lookup table of the form

BYTE_1	BYTE_2	AND	OR	XOR	NOT
b	b'	$b \wedge b'$	$b \vee b'$	$b \oplus b'$	$\neg b$

where the first two arguments run through all pairs of bytes (b, b').

Split and shifted prefix and suffix and ones

The $^\diamond$SPLIT_AND_SHIFTED_PREFIX$_i$ and $^\diamond$SPLIT_AND_SHIFTED_SUFFIX$_i$ columns contain the result of splitting the binary representation of the byte BYTE_2$_i$ in two and shifting the resulting bits "to the opposite side". The $^\diamond$ONES column contains a byte that is a (left aligned) sequence of one's (possibly none) followed by zero's (at least one). The associated lookup table depends on two parameters: a byte $B = \mathtt{a\,b\,c\,d\,e\,f\,g\,h}$ (taken from the BYTE_2 column) in big endian binary representation, and a micro-shift parameter μSHP $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

The $^\diamond$SNS_PREFIX, $^\diamond$SNS_SUFFIX and $^\diamond$ONES columns are obtained by lookup using BYTE_2 and μSHP according to the lookup table 10.2.

10.1.4 Technical constraints

Trivial constraints

The following constraints apply when SHF$_i = 0$ AND PF$_i = 0$.

1. ZP$_i = 31$,
2. BB$_i = 0$,
3. NEG$_i = 0$,
4. IRF$_i = 0$,

LEFT SHIFT		
LOW_3	$^{\Diamond}$SNS_SUFFIX	$^{\Diamond}$SNS_PREFIX
0	00000000	abcdefgh
1	0000000a	bcdefgh0
2	000000ab	cdefgh00
3	00000abc	defgh000
4	0000abcd	efgh0000
5	000abcde	fgh00000
6	00abcdef	gh000000
7	0abcdefg	h0000000

RIGHT SHIFT		
LOW_3	$^{\Diamond}$SNS_SUFFIX	$^{\Diamond}$SNS_PREFIX
0	abcdefgh	00000000
1	0abcdefg	h0000000
2	00abcdef	gh000000
3	000abcde	fgh00000
4	0000abcd	efgh0000
5	00000abc	defgh000
6	000000ab	cdefgh00
7	0000000a	bcdefgh0

μSHP	$^{\Diamond}$SNS_SUFFIX	$^{\Diamond}$SNS_PREFIX	$^{\Diamond}$ONES	LB
0	abcdefgh	00000000	00000000	a
1	0abcdefg	h0000000	10000000	a
2	00abcdef	gh000000	11000000	a
3	000abcde	fgh00000	11100000	a
4	0000abcd	efgh0000	11110000	a
5	00000abc	defgh000	11111000	a
6	000000ab	cdefgh00	11111100	a
7	0000000a	bcdefgh0	11111110	a
8	00000000	abcdefgh	unspecified	a

Figure 10.2: The first two tables represent the expected $^{\Diamond}$SNS_SUFFIX and $^{\Diamond}$SNS_PREFIX values according to LOW_3 and shift direction. The bottom table combines the two and outputs the expected split and shifted prefixes and suffixes in terms of μSHP. It also contains the associated $^{\Diamond}$ONES column.

5. also constrain the BYTE_RES column:

$$\begin{aligned}\mathsf{BYTE\_RES}_i \;=\; & \;\;\mathsf{AND}_i \cdot \mathsf{AND\_FLAG}_i \\ & + \;\;\mathsf{OR}_i \cdot \;\;\mathsf{OR\_FLAG}_i \\ & + \mathsf{NOT}_i \cdot \mathsf{NOT\_FLAG}_i \\ & + \mathsf{XOR}_i \cdot \mathsf{XOR\_FLAG}_i\end{aligned}$$

NOTE. The first condition and the constraints for ZB (section 10.1.4) ensures that $\mathsf{ZB}_i = 0$, too.

Specifying BYTE_RES in the case $\mathsf{SHF}_i = 1$ or $\mathsf{PF}_i = 1$ is more complex; we deal with in the section 10.1.5 and section 10.1.6 respectively.

1. IF $\mathsf{SHF}_i = 0$ THEN
   (a) $\mathsf{DB}_i = 0$,
   (b) $\mu\mathsf{SHF}_i = 0$,
   (c) $\mu\mathsf{SHP}_i = 0$.
2. IF $\mathsf{PF}_i = 0$ THEN $\mathsf{PB}_i = 0$.

Binaryness constraints

Recall that a column X is **binary** if it satisfies the constraint

$$X_i \cdot (1 - X_i) = 0$$

The following columns are binary: $\llbracket 2 \rrbracket$, $\llbracket 1 \rrbracket$, $\llbracket 0 \rrbracket$, NEG, BB, DB, $\mu\mathsf{SHF}$, IRF. We thus have the following constraints:

1. $\llbracket 2 \rrbracket_i \cdot (1 - \llbracket 2 \rrbracket_i) = 0$
2. $\llbracket 1 \rrbracket_i \cdot (1 - \llbracket 1 \rrbracket_i) = 0$
3. $\llbracket 0 \rrbracket_i \cdot (1 - \llbracket 0 \rrbracket_i) = 0$
4. $\mathsf{NEG}_i \cdot (1 - \mathsf{NEG}_i) = 0$
5. $\mathsf{BB}_i \cdot (1 - \mathsf{BB}_i) = 0$
6. $\mathsf{DB}_i \cdot (1 - \mathsf{DB}_i) = 0$
7. $\mu\mathsf{SHF}_i \cdot (1 - \mu\mathsf{SHF}_i) = 0$
8. $\mathsf{IRF}_i \cdot (1 - \mathsf{IRF}_i) = 0$

$\mathsf{COUNTER}$ constraints

$\mathsf{COUNTER}$ is supposed to loop continuously from 31 down to 0.

1. $\mathsf{COUNTER}_0 = 31$, i.e. we initialize $\mathsf{COUNTER}$ at 31,
2. $\textsc{if}$ $\mathsf{COUNTER}_{i-1} \neq 0$ $\textsc{then}$ $\mathsf{COUNTER}_i = \mathsf{COUNTER}_{i-1} - 1$
3. $\textsc{if}$ $\mathsf{COUNTER}_{i-1} = 0$ $\textsc{then}$ $\mathsf{COUNTER}_i = 31$
4. $\mathsf{COUNTER}_{N-1} = 0$, i.e. $\mathsf{COUNTER}$ must end with 0,

Locally-constant columns

We say that a column X is **locally-constant** if it satisfies the following constraint:

$$\textsc{if}\ \mathsf{COUNTER}_{i-1} \neq 0\ \textsc{then}\ \mathsf{X}_i = \mathsf{X}_{i-1}$$

The following columns are locally-constant: $\mu\mathsf{SHIFT\_FLAG}$, $\llbracket 0 \rrbracket$, $\llbracket 1 \rrbracket$, $\llbracket 2 \rrbracket$, $\mathsf{BINARY\_STAMP}$, $\mathsf{ZERO\_BACP\_PARAM}$, $\mu\mathsf{SHIFT\_PARAM}$, $\mathsf{PIVOT\_BYTE}$, $\mathsf{INPUT\_1}$, $\mathsf{INPUT\_2}$, RES. Hence we have the following constraints:

1. $\textsc{if}$ $\mathsf{COUNTER}_{i-1} \neq 0$ $\textsc{then}$
   (a) $\llbracket 0 \rrbracket_i = \llbracket 0 \rrbracket_{i-1}$
   (b) $\llbracket 1 \rrbracket_i = \llbracket 1 \rrbracket_{i-1}$
   (c) $\llbracket 2 \rrbracket_i = \llbracket 2 \rrbracket_{i-1}$
   (d) $\mathsf{BS}_i = \mathsf{BS}_{i-1}$
   (e) $\mathsf{INPUT\_1}_i = \mathsf{INPUT\_1}_{i-1}$
   (f) $\mathsf{INPUT\_2}_i = \mathsf{INPUT\_2}_{i-1}$
   (g) $\mathsf{RES}_i = \mathsf{RES}_{i-1}$
   (h) $\mathsf{PB}_i = \mathsf{PB}_{i-1}$
   (i) $\mathsf{DB}_i = \mathsf{DB}_{i-1}$
   (j) $\mu\mathsf{SHF}_i = \mu\mathsf{SHF}_{i-1}$
   (k) $\mathsf{ZP}_i = \mathsf{ZP}_{i-1}$

we could have replace the condition $\mathsf{COUNTER}_{i-1} \neq 0$ with $\mathsf{COUNTER}_i \neq 31$.

Range proofs

We require a range proof that the columns $\mathsf{BYTE\_1}$, $\mathsf{BYTE\_2}$, $\mathsf{BYTE\_RES}$, $\mathsf{PIVOT\_BYTE}$ only contains bytes, i.e. values in the range $\llbracket 0, 256\llbracket$. This constraint is applied to the interleaved column $\mathsf{BYTE\_1} \odot \mathsf{BYTE\_2} \odot \mathsf{BYTE\_RES} \odot \mathsf{PIVOT\_BYTE}$.
NOTE. We threw $\mathsf{PIVOT\_BYTE}$ into the mix because we don't constrain it universally (i.e. we only care about it for pivot-instructions) and so that the resulting vector has length 128 for non shift-instructions 6*128 for shift-instructions. In any case, with 2 non shift-instructions or 1 shift instructions this column has length ≥ 256 and so can be used in a Cairo-style range proof.

BYTE / PREFIX / INPUT constraints

For $(\mathsf{B}, \mathsf{P}, \mathsf{I})$ any of the following columns triples

1. $(\mathsf{BYTE\_1}, \mathsf{PREFIX\_1}, \mathsf{INPUT\_1})$,
2. $(\mathsf{BYTE\_2}, \mathsf{PREFIX\_2}, \mathsf{INPUT\_2})$,
3. $(\mathsf{BYTE\_RES}, \mathsf{PREFIX\_RES}, \mathsf{RES})$

We implemnent the following constraints:

1. I is locally-constant,
2. IF $\mathsf{COUNTER}_i = 31$, THEN $\mathsf{P}_i = \mathsf{B}_i$
3. IF $\mathsf{COUNTER}_i \neq 31$, THEN $\mathsf{P}_i = 256 \cdot \mathsf{P}_{i-1} + \mathsf{B}_i$
4. IF $\mathsf{COUNTER}_i = 0$, THEN $\mathsf{P}_i = \mathsf{I}_i$

$\llbracket 0 \rrbracket$, $\llbracket 1 \rrbracket$, and $\llbracket 2 \rrbracket$ constraints

Recall the abbreviations $\llbracket 0 \rrbracket = \mathsf{BIT\_0}$, $\llbracket 1 \rrbracket = \mathsf{BIT\_1}$, $\llbracket 2 \rrbracket = \mathsf{BIT\_2}$. We think of $\llbracket 2 \rrbracket_i \llbracket 1 \rrbracket_i \llbracket 0 \rrbracket_i$ as being the (big endian) base 2 digits of a locally-constant counter that is initialized at 0 for non shift-instructions and at 5 for shift-instructions. This counter, while > 0, decreases by one at the end of every $\mathsf{COUNTER}$-cycle. The $\mathsf{COUNTER}$-cycle where it is 0 marks the final $\mathsf{COUNTER}$-cycle of the current instruction. In other words, non shift-instructions span 1 $\mathsf{COUNTER}$-cycle while shift-instructions span 6 $\mathsf{COUNTER}$-cycles. In case of a shift-instruction the interpretation is as follows: the first $\mathsf{COUNTER}$-cycle performs micro-shifting (i.e. bit shift within bytes), the next 5 $\mathsf{COUNTER}$-cycle perform for macro-shifts (i.e. potentially moving the bytes by 1, 2, 4, 8 or 16 indices).

1. $\llbracket 0 \rrbracket$, $\llbracket 1 \rrbracket$ and $\llbracket 2 \rrbracket$ are locally-constant;
2. $\llbracket 0 \rrbracket$, $\llbracket 1 \rrbracket$ and $\llbracket 2 \rrbracket$ are binary;
   NOTE. it seems reasonable that we may omit the "binaryness" conditions given what follows;
3. initialization of the bits:
   (a) $\llbracket 0 \rrbracket_0 = \mathsf{SHIFT\_FLAG}_0$
   (b) $\llbracket 1 \rrbracket_0 = 0$
   (c) $\llbracket 2 \rrbracket_0 = \mathsf{SHIFT\_FLAG}_0$
4. IF $\mathsf{COUNTER}_{i-1} = 0$ AND ($\llbracket 0 \rrbracket_{i-1} = 0$ AND $\llbracket 1 \rrbracket_{i-1} = 0$ AND $\llbracket 2 \rrbracket_{i-1} = 0$) THEN
   (a) $\llbracket 0 \rrbracket_i = \mathsf{SHIFT\_FLAG}_i$
   (b) $\llbracket 1 \rrbracket_i = 0$

(c) $\llbracket 2 \rrbracket_i = \mathsf{SHIFT\_FLAG}_i$

In other words, at the onset of a non shift-instructions $\llbracket 2 \rrbracket_i \llbracket 1 \rrbracket_i \llbracket 0 \rrbracket_i = 000$ (i.e. 0 in binary) while at the onset of a shift-instruction, $\llbracket 2 \rrbracket_i \llbracket 1 \rrbracket_i \llbracket 0 \rrbracket_i = 101$ (i.e. 5 in binary).

5. $\textsc{if}$ $\mathsf{COUNTER}_{i-1} = 0$ $\textsc{and}$ ($\llbracket 0 \rrbracket_{i-1} = 1$ $\textsc{or}$ $\llbracket 1 \rrbracket_{i-1} = 1$ $\textsc{or}$ $\llbracket 2 \rrbracket_{i-1} = 1$) $\textsc{then}$

   (a) $\llbracket 0 \rrbracket_i = 1 - \llbracket 0 \rrbracket_{i-1}$, i.e. the zero-th bit flips after every $\mathsf{COUNTER}$-cycle of a shift-instruction,

   (b) $\llbracket 1 \rrbracket_i = \llbracket 1 \rrbracket_{i-1} \cdot \llbracket 0 \rrbracket_{i-1} + (1 - \llbracket 1 \rrbracket_{i-1}) \cdot (1 - \llbracket 0 \rrbracket_{i-1})$, i.e. the first bit flips whenever the zero-th bit is zero,

$$\begin{cases} \textsc{if} \ \llbracket 0 \rrbracket_{i-1} = 0 & \textsc{then} \ \llbracket 1 \rrbracket_i = 1 - \llbracket 1 \rrbracket_{i-1} \\ \textsc{if} \ \llbracket 0 \rrbracket_{i-1} = 1 & \textsc{then} \ \llbracket 1 \rrbracket_i = \llbracket 1 \rrbracket_{i-1} \end{cases}$$

   (c) $\llbracket 2 \rrbracket_i = \llbracket 2 \rrbracket_{i-1} \cdot (\llbracket 1 \rrbracket_{i-1} + \llbracket 0 \rrbracket_{i-1} - \llbracket 1 \rrbracket_{i-1} \cdot \llbracket 0 \rrbracket_{i-1}) + (1 - \llbracket 2 \rrbracket_{i-1}) \cdot (1 - \llbracket 1 \rrbracket_{i-1}) \cdot (1 - \llbracket 0 \rrbracket_{i-1})$, i.e. the second bit flips whenever both the first and zero-th bit are both zero:

$$\begin{cases} \textsc{if} \ \left(\llbracket 0 \rrbracket_{i-1} = 0 \ \textsc{and} \ \llbracket 1 \rrbracket_{i-1} = 0 \right) & \textsc{then} \ \llbracket 2 \rrbracket_i = 1 - \llbracket 2 \rrbracket_{i-1} \\ \textsc{if} \ \left(\llbracket 0 \rrbracket_{i-1} = 1 \ \textsc{or} \ \llbracket 1 \rrbracket_{i-1} = 1 \right) & \textsc{then} \ \llbracket 2 \rrbracket_i = \llbracket 2 \rrbracket_{i-1} \end{cases}$$

   In other words, after every $\mathsf{COUNTER}$-cycle within a given shift-instruction the base 2 integer $\llbracket 2 \rrbracket \llbracket 1 \rrbracket \llbracket 0 \rrbracket$ decreases by 1;

6. finalization of the bits:

   (a) $\llbracket 0 \rrbracket_{N-1} = 0$

   (b) $\llbracket 1 \rrbracket_{N-1} = 0$

   (c) $\llbracket 2 \rrbracket_{N-1} = 0$

$\mathsf{BINARY\_STAMP}$ constraints

1. BS is locally-constant;

2. $\mathsf{BS}_0 = 0$, i.e. BS is initialized at 0;

3. $\textsc{if}$ $\mathsf{COUNTER}_{i-1} = 0$ $\textsc{then}$

$$\mathsf{BS}_i = \mathsf{BS}_{i-1} + (1 - \llbracket 0 \rrbracket_{i-1}) \cdot (1 - \llbracket 1 \rrbracket_{i-1}) \cdot (1 - \llbracket 2 \rrbracket_{i-1})$$

In other words, $\textsc{if}$ $\mathsf{COUNTER}_{i-1} = 0$ $\textsc{and}$ $\llbracket 2 \rrbracket_{i-1} \llbracket 1 \rrbracket_{i-1} \llbracket 0 \rrbracket_{i-1} = 000$ $\textsc{then}$ $\mathsf{BS}_i = \mathsf{BS}_{i-1} + 1$, while $\textsc{if}$ $\mathsf{COUNTER}_{i-1} = 0$ $\textsc{and}$ $\llbracket 2 \rrbracket_{i-1} \llbracket 1 \rrbracket_{i-1} \llbracket 0 \rrbracket_{i-1} \neq 000$ $\textsc{then}$ $\mathsf{BS}_i = \mathsf{BS}_{i-1}$

Stamp-constant columns

Let X be a column. We say that X is **stamp-constant** if it satisfies the following constraint:

$$\textsc{if} \ \mathsf{BINARY\_STAMP}_i = \mathsf{BINARY\_STAMP}_{i-1} \ \textsc{then} \ X_i = X_{i-1}$$

The following columns are stamp-constant: INST, NEG, $\mu\mathsf{SHP}$, $\mathsf{IN\_RANGE\_FLAG}$. Hence we have the following constraints:

1. $\textsc{if}$ $\mathsf{BINARY\_STAMP}_i = \mathsf{BINARY\_STAMP}_{i-1}$ $\textsc{then}$

   (a) $\mathsf{INST}_i = \mathsf{INST}_{i-1}$

   (b) $\mathsf{NEG}_i = \mathsf{NEG}_{i-1}$

(c) $\mu\mathsf{SHP}_i = \mu\mathsf{SHP}_{i-1}$

(d) $\mathsf{IRF}_i = \mathsf{IRF}_{i-1}$

Since BINARY_STAMP is locally-constant it follows that stamp-constant columns are locally-constant, too. Note that all columns that are deduced from INST by means of a lookup table (that is AND_FLAG, OR_FLAG, XOR_FLAG, NOT_FLAG, SHIFT_FLAG, SHIFT_DIRECTION, SAR_FLAG, PIVOT_FLAG, SIGNEXTEND_FLAG) are thus automatically stamp-constant (but we don't need to include the associated constraint).

μSHIFT_FLAG constraints

We have

1. $\boxed{\mu\mathsf{SHF} \text{ is locally-constant;}}$
2. $\boxed{\mu\mathsf{SHF} \text{ is a binary column;}}$
3. we initialize it $\mu\mathsf{SHF}_0 = \mathsf{SHIFT\_FLAG}_0$;
4. IF $\mathsf{COUNTER}_{i-1} = 0$:
   (a) IF $\mathsf{BS}_i \neq \mathsf{BS}_{i-1}$ THEN $\mu\mathsf{SHF}_i = \mathsf{SHIFT\_FLAG}_i$,
   (b) IF $\mathsf{BS}_i = \mathsf{BS}_{i-1}$ THEN $\mu\mathsf{SHF}_i = 0$.

In other words, $\mu\mathsf{SHF}_i = 0$ for non shift-instructions, while for shift-instructions $\mu\mathsf{SHF}_i = 1$ during the micro-shift COUNTER-cycle and $\mu\mathsf{SHF}_i = 0$ during the macro-shift COUNTER-cycles.

ZERO_BACP_PARAM constraints

We begin with the trivial case of ZP, i.e. that of a non shift-instruction and non pivot-instruction.

1. $\boxed{\mathsf{ZP} \text{ is locally-constant;}}$
2. $\boxed{\text{IF } \mathsf{SHF}_i = 0 \ \text{ AND } \mathsf{PF}_i = 0 \text{ THEN } \mathsf{ZP}_i = 31}$;

We now deal with constraining ZP for shift-instructions and then pivot-instructions.

ZP constraints for shift-instructions. During the micro-shift COUNTER-cycle of a shift-instruction $\mathsf{ZP}_i = 7$. In the first macro-shift COUNTER-cycle we set ZP either to 0 or 30 depending on the SHIFT_DIRECTION. From then on out $(\mathsf{ZP} - \texttt{cst})$ follows a geometric progression with ratio 2 for the remaining COUNTER-cycles of the shift-instruction (for some constant `cst` which depends on SHIFT_DIRECTION).

1. IF $\mathsf{SHF}_i = 1$ THEN
   (a) IF $\mu\mathsf{SHF}_i = 1$ THEN $\mathsf{ZP}_i = 7$;
   (b) IF $\mathsf{BS}_i = \mathsf{BS}_{i-1}$ AND $\mathsf{COUNTER}_i = 31$ THEN
      i. IF $\mu\mathsf{SHF}_{i-1} = 1$ THEN
$$\mathsf{ZP}_i = 30 \cdot \mathsf{SHD}_i$$
      In other words during the first macro-shift COUNTER-cycle of a shift-instruction
$$\begin{cases} \text{IF } \mathsf{SHD}_i = 0: & \mathsf{ZP}_i = 0 \\ \text{IF } \mathsf{SHD}_i = 1: & \mathsf{ZP}_i = 30 \end{cases}$$

BS	SHF	SHD	μSHF	$[\![2]\!]$	$[\![1]\!]$	$[\![0]\!]$	ZP
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x-1$	?	?	0	0	0	0	?
$\boldsymbol{x}$	1	0	1	1	0	1	7
$\boldsymbol{x}$	1	0	0	1	0	0	0
$\boldsymbol{x}$	1	0	0	0	1	1	1
$\boldsymbol{x}$	1	0	0	0	1	0	3
$\boldsymbol{x}$	1	0	0	0	0	1	7
$\boldsymbol{x}$	1	0	0	0	0	0	15
$x+1$	?	?	μ	μ	0	μ	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

BS	SHF	SHD	μSHF	$[\![2]\!]$	$[\![1]\!]$	$[\![0]\!]$	ZP
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x-1$	?	?	0	0	0	0	?
$\boldsymbol{x}$	1	1	1	1	0	1	7
$\boldsymbol{x}$	1	1	0	1	0	0	30
$\boldsymbol{x}$	1	1	0	0	1	1	29
$\boldsymbol{x}$	1	1	0	0	1	0	27
$\boldsymbol{x}$	1	1	0	0	0	1	23
$\boldsymbol{x}$	1	1	0	0	0	0	15
$x+1$	?	?	μ	μ	0	μ	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 10.3: The left hand side represents the 6 COUNTER-cycles of a shift-instruction with SHIFT_DIRECTION $= 0$. The right hand side represents the 6 COUNTER-cycles of a shift-instruction with SHIFT_DIRECTION $= 1$. Every row represents a full COUNTER-cycle, i.e. 32 rows of the actual execution trace.

BS	SHF	μSHF	SGNXF	$[\![2]\!]$	$[\![1]\!]$	$[\![0]\!]$	ZP
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x-1$	?	0	?	0	0	0	?
$\boldsymbol{x}$	**0**	**0**	**0**	**0**	**0**	**0**	**30**
$x+1$	?	μ	?	μ	0	μ	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

BS	SHF	μSHF	SGNXF	$[\![2]\!]$	$[\![1]\!]$	$[\![0]\!]$	ZP
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x-1$	?	0	?	0	0	0	?
$\boldsymbol{x}$	**0**	**0**	**1**	**0**	**0**	**0**	**?**
$x+1$	?	μ	?	μ	0	μ	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 10.4: The table on the left represents the 1 COUNTER-cycle of an instruction that is neither a shift-instruction nor a `SIGNEXTEND` instruction. The table on the right represents the 1 COUNTER-cycle of a `SIGNEXTEND` instruction.

ii. IF $\mu\mathsf{SHF}_{i-1} = 0$ THEN

$$\mathsf{ZP}_i - 1 + 32 \cdot \mathsf{SHD}_i = 2 \cdot (\mathsf{ZP}_{i-1} - 1 + 32 \cdot \mathsf{SHD}_{i-1})$$

(**NOTE.** $\mathsf{SHD}_i = \mathsf{SHD}_{i-1}$ since SHD is stamp-constant.) In other words

$$\begin{cases} \text{IF } \mathsf{SHIFT\_DIRECTION}_i = 0 : & \mathsf{ZP}_i + 1 = 2 \cdot (\mathsf{ZP}_{i-1} + 1) \\ \text{IF } \mathsf{SHIFT\_DIRECTION}_i = 1 : & 31 - \mathsf{ZP}_i = 2 \cdot (31 - \mathsf{ZP}_{i-1}) \end{cases}$$

In other words along shift-instructions the values of ZP evolve as follows While for non shift-instructions it evolves as follows:

ZP constraints for pivot-instructions. We deal with this case in section 10.1.6.

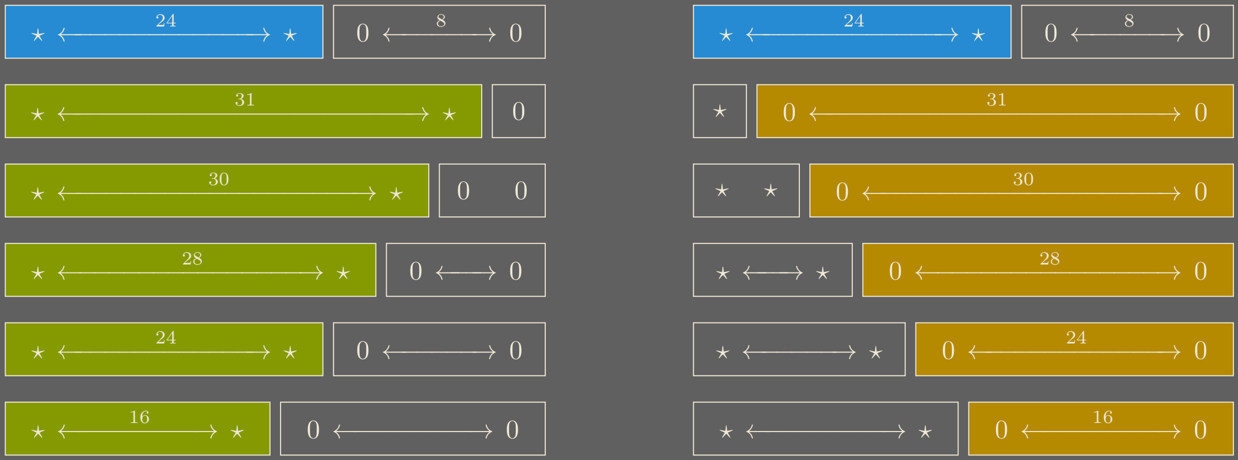

Figure 10.5: Pattern of nonzero values vs zeros in the ZERO_BACP column. Both the left hand side and right hand side represent the ZERO_BACP column (as rows rather than columns) during the 6 COUNTER-cycles of a shift-instruction. The first row is the micro-shift COUNTER-cycle: in both cases there are 24 nonzero values followed by 8 zeros. The following rows (columns) on the left hand side represent the pattern for `SHL` full byte shifting. The green values are nonzero and are those indices that will be made to contain bytes. The columns on the right hand side represent the pattern for `SHR` full byte shifting. The yellow values are zero and are at those indices that will be made to contain bytes.

ZERO_BACP constraints

A COUNTER-cycle's worth of ZERO_BACP values contains a string of nonzero values (possibly none) followed by zeros (at least one), i.e. it looks like so (with $\star$ indicating nonzero values)

COUNTER	$\cdots$	31	30	$\cdots$	$\mathsf{p}+1$	p	$\mathsf{p}-1$	$\cdots$	1	0	31
ZERO_BACP_PARAM	$\cdots$	p	p	$\cdots$	p	p	p	$\cdots$	p	p	$\cdots$
ZERO_BACP	$\cdots$	$\star$	$\star$	$\cdots$	$\star$	0	0	$\cdots$	0	0	$\cdots$

1. IF $\mathsf{COUNTER}_i = 31$, THEN $\mathsf{ZB}_i = \mathsf{CT}_i - \mathsf{ZP}_i$
2. IF $\mathsf{COUNTER}_i \neq 31$, THEN $\mathsf{ZB}_i = \mathsf{ZB}_{i-1} \cdot (\mathsf{CT}_i - \mathsf{ZP}_i)$

BYTE_BITS constraints

We ask that BB satisfy the following constraints:

1. BB is a binary column;
2. IF $\mathsf{SHF}_i = 0$ AND $\mathsf{PF}_i = 0$ THEN $\mathsf{BB}_i = 0$;
3. IF $\mathsf{SHF}_i = 1$:
   (a) IF $\mu\mathsf{SHF}_i = 1$ AND $\mathsf{ZB}_i \neq 0$ THEN $\mathsf{BB}_i = 0$;
   (b) IF $\mu\mathsf{SHF}_i = 1$ AND $\mathsf{COUNTER}_i = 0$ THEN

$$\mathsf{BYTE\_1}_i = \sum_{k=0}^{7} 2^k \cdot \mathsf{BB}_{i-k}$$

Recall that during the micro-shift phase of a shift-instruction $\mathsf{ZP} = 7$. The above two constraints thus mean the following: the values in BB during the micro-shift COUNTER-cycle of a shift-instruction are comprised of the 24 zeros followed by 8 bits that are the bit decomposition of the least significant byt of INPUT_1.

(c) IF $\mu\mathsf{SHF}_i = 0$ THEN $\mathsf{BB}_i = \mathsf{BB}_{i-32}$;

i.e. one transfers bits from the previous COUNTER-cycle of BB to the next COUNTER-cycle.

4. IF $\mathsf{PF}_i = 1$:

   (a) IF $\mathsf{COUNTER}_i = 0$ THEN

$$\begin{cases} \mathsf{BYTE\_1}_i = \displaystyle\sum_{k=0}^{7} 2^k \cdot \mathsf{BB}_{i-k} \\ \mathsf{PB}_i = \displaystyle\sum_{k=0}^{7} 2^k \cdot \mathsf{BB}_{i-8-k} \in [\![0, 256[\![\end{cases}$$

The above two constraints thus mean the following: the final 8 bits in BB are the bit decomposition of the least significant byt of INPUT_1, and the 8 bits preceding them are the bit decomposition of the pivot byte. We reproduce these constraints later in context 10.1.6.

DECISION_BIT constraints

DB is a locally-constant binary column. For non shift-instruction, DB is zero. During the micro-shift COUNTER-cycle of a shift-instruction DB is zero. During the macro-shift COUNTER-cycles of a shift-instruction DB will contain in sequence the 5 most significant bits of the byte that is recorded in the last 8 bits of BB.

1. DB is locally-constant;

2. DB is a binary column;

3. IF $\mathsf{SHF}_i = 0$ THEN $\mathsf{DB}_i = 0$;

4. IF $\mathsf{SHF}_i = 1$ THEN :

   (a) IF $\mu\mathsf{SHF}_i = 1$ THEN $\mathsf{DB}_i = 0$;

   (b) IF $\mu\mathsf{SHF}_i = 0$ AND $\mathsf{COUNTER}_i = 0$ THEN

$$\begin{aligned} \mathsf{DB}_i \quad = \quad & [\![2]\!]_i \cdot [\![1]\!]_i^\vee \cdot [\![0]\!]_i^\vee \cdot \mathsf{BB}_{i-3} \\ + \; & [\![2]\!]_i^\vee \cdot [\![1]\!]_i \cdot [\![0]\!]_i \cdot \mathsf{BB}_{i-4} \\ + \; & [\![2]\!]_i^\vee \cdot [\![1]\!]_i \cdot [\![1]\!]_i^\vee \cdot \mathsf{BB}_{i-5} \\ + \; & [\![2]\!]_i^\vee \cdot [\![1]\!]_i^\vee \cdot [\![0]\!]_i \cdot \mathsf{BB}_{i-6} \\ + \; & [\![2]\!]_i^\vee \cdot [\![1]\!]_i^\vee \cdot [\![0]\!]_i^\vee \cdot \mathsf{BB}_{i-7} \end{aligned}$$

   (Recall the convention $[\![k]\!]^\vee = 1 - [\![k]\!]$, for $k = 0, 1, 2$.)

I.e. in the first macro-shift CT-cycle, DB contains $\mathtt{LSB}_3$, in the second one, $\mathtt{LSB}_4$, in the third one, $\mathtt{LSB}_5$, in the fourth one, $\mathtt{LSB}_6$ and in the fifth one, $\mathtt{LSB}_7$. Here the $\mathtt{LSB}_i$, $i = 0, \ldots, 7$, are the big endian base 2 digits of the least significant byte LSB of the first argument of the shift-instruction (INPUT_1.)

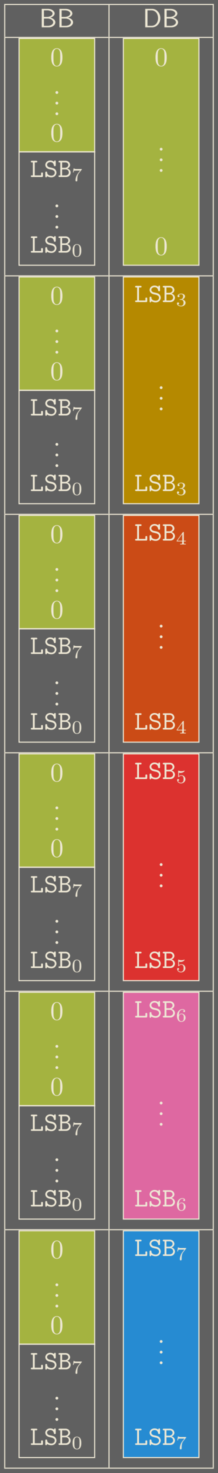

Figure 10.6: The above represents the 6 COUNTER-cycles of a shift-instruction. The first COUNTER-cycle is the micro-shift cycle. The values in the BB column during the first COUNTER-cycle are reproduced in the BB column of the 5 macro-shift COUNTER-cycles of the instruction. Green boxes represent zeros. Monochrome boxes represent locally-constant columns.

IN_RANGE_FLAG constraints

The IN_RANGE_FLAG column tests whether INPUT_1 is in range. This is only relevant for pivot-instructions andshift-instructions. Depending on the instruction this means different things.

1. IRF is a binary flag;
2. IRF is stamp-constant;
3. IF $\mathsf{SHF}_i = 0$ AND $\mathsf{PF}_i = 0$ THEN $\mathsf{IRF}_i = 0$;

For shift-instructions "INPUT_1 in range" means $\mathsf{INPUT\_1} \in [\![0, 256[\![$:

4. IF $\mathsf{SHF}_i = 1$ AND $\mu\mathsf{SHF}_i = 1$ AND $\mathsf{COUNTER}_i = 0$ THEN

$$\begin{cases} \text{IF } \mathsf{INPUT\_1}_i = \mathsf{BYTE\_1}_i : & \mathsf{IN\_RANGE\_FLAG}_i = 1 \\ \text{IF } \mathsf{INPUT\_1}_i \neq \mathsf{BYTE\_1}_i : & \mathsf{IN\_RANGE\_FLAG}_i = 0 \end{cases}$$

i.e. during the final step of the micro-shift COUNTER-cycle of a shift-instruction we compare INPUT_1 with its least significant byte and if they agree (i.e. if INPUT_1 is a byte) then IN_RANGE_FLAG is set to 1, otherwise to 0;

NOTE. we can drop the condition "$\mathsf{SHF}_i = 1$" since "$\mu\mathsf{SHF}_i = 1$" can only occur during (the micro shift phase of) a shift-instruction;

For pivot-instructions "INPUT_1 in range" means $\mathsf{INPUT\_1} \in [\![0, 32[\![$:

5. IF $\mathsf{PF}_i = 1$ AND $\mathsf{COUNTER}_i = 0$ THEN

$$\begin{cases} \text{IF } \mathsf{INPUT\_1}_i = \sum_{k=0}^{4} 2^k \cdot \mathsf{BB}_{i-k} : & \mathsf{IRF}_i = 1 \\ \text{IF } \mathsf{INPUT\_1}_i \neq \sum_{k=0}^{4} 2^k \cdot \mathsf{BB}_{i-k} : & \mathsf{IRF}_i = 0[\![\end{cases}$$

Indeed, for pivot-instructions the final 8 bits of the BB column contain the bits of the least significant byte of INPUT_1.

μSHIFT_PARAM constraints.

μSHP contains the micro-shift parameter which is used for byte slicing. As such, its value only matters for shift-instructions. Furthermore its value depends on whether it's a right shift (i.e. `SHR` and `SAR`) or a left shift (i.e. `SHL`). By construction it is a number in the range $\{0, 1, \ldots, 8\}$.

1. μSHP is stamp-constant;
2. IF $\mathsf{SHF}_i = 0$ THEN $\mu\mathsf{SHP}_i = 0$;
3. IF $\mathsf{SHF}_i = 1$ AND $\mu\mathsf{SHF}_i = 1$ AND $\mathsf{COUNTER}_i = 0$ THEN

$$\begin{cases} \text{IF } \mathsf{SHD}_i = 1 : & \mu\mathsf{SHP}_i = \sum_{k=0}^{2} 2^k \cdot \mathsf{BB}_{i-k} \\ \text{IF } \mathsf{SHD}_i = 0 : & \mu\mathsf{SHP}_i = 8 - \sum_{k=0}^{2} 2^k \cdot \mathsf{BB}_{i-k} \end{cases}$$

i.e. we set SHD at the last row of the micro-shift COUNTER-cycle of a shift-instruction.

10.1.5 Shift-instruction constraints

The micro-shift COUNTER-cycle (i.e. first COUNTER-cycle of a shift-instruction i.e. the COUNTER-cycle where $\mu\mathsf{SHF} = 1$) uses the plookup justified shifted prefixes and suffixes (as well as the $^\diamond\mathsf{ONES}$ column in case of a `SAR` instruction) to compute the bytes of the micro-shifted word INPUT_2. The results are stored in the BYTE_RES column.

The following 5 COUNTER-cycles of a shift-instruction (i.e. its macro-shift COUNTER-cycles) do three things:

1. they copy the previous COUNTER-cycle's BYTE_RES column into the current COUNTER-cycle's BYTE_1 column
2. they insert the right/left shifted version of the previous COUNTER-cycle's BYTE_RES column into the current COUNTER-cycle's BYTE_2 column
3. depending on the current COUNTER-cycle's DECISION_BIT column they copy (the current COUNTER-cycle's) BYTE_1 or BYTE_2 into (the current COUNTER-cycle's) BYTE_RES.

Micro-shift constraints

The following constraints apply when $\mathsf{SHF}_i = 1$ AND $\mu\mathsf{SHF}_i = 1$.

1. IF $\mathsf{SHD}_i = 1$ (i.e. micro shift phase of a `SHR` or `SAR` instruction) THEN
   (a) IF $\mathsf{COUNTER}_i = 31$:
      i. Pad the leading BYTE_RES with the appropriate number of ones during the micro-shift phase of a `SAR` instruction:

$$\begin{aligned}\mathsf{BYTE\_RES}_i \quad &= \quad {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_SUFFIX}_i \\ &\quad\; + \mathsf{SAR\_FLAG}_i \cdot {}^\diamond\mathsf{ONES}_i \cdot \cdot\, \mathsf{LB}_i\end{aligned}$$

      ii. Set NEG_i: $\mathsf{NEG}_i {=} \mathsf{SAR\_FLAG}_i \cdot \mathsf{LB}_i$ i.e. $\mathsf{NEG}_i = 0$ unless we are doing a `SAR` instruction and the leading bit of INPUT_2 is 1, in which case $\mathsf{NEG}_i = 1$
   (b) ELSEIF $\mathsf{COUNTER}_i \neq 31$:

$$\mathsf{BYTE\_RES}_i = {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_SUFFIX}_i + {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_PREFIX}_{i-1}$$

2. IF $\mathsf{SHD}_i = 0$ (i.e. micro shift phase of a `SHL` instruction) then
   (a) IF $\mathsf{COUNTER}_i \neq 0$:

$$\mathsf{BYTE\_RES}_i = {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_PREFIX}_i + {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_SUFFIX}_{i+1}$$

   (b) ELSEIF $\mathsf{COUNTER}_i = 0$:

$$\mathsf{BYTE\_RES}_i = {}^\diamond\mathsf{SPLIT\_AND\_SHIFTED\_PREFIX}_i$$

Macro-shift constraints

The following constraints apply when $\mathsf{SHF}_i = 1$ AND $\mu\mathsf{SHF}_i = 0$.

1. IF $\mathsf{IN\_RANGE\_FLAG}_i = 0$:

$$\begin{cases}\mathsf{BYTE\_1}_i = 0\\ \mathsf{BYTE\_2}_i = 0\\ \mathsf{BYTE\_RES}_i = 255 \cdot \mathsf{NEG}_i\end{cases}$$

NOTE. Recall that for shift-instructions NEG can only be nonzero for a `SAR` instruction.

In other words, if the instruction requires us to shift by ≥ 256 bits, the result of the shift-instruction will be zero in all cases except when executing a `SAR` instruction on a negative second argument, in which case the expected result is -1, i.e. `0xf` `f` $\cdots$ `f` a string of 64 `f`'s. The second argument is negative *iff* the leading bit of the second input of the instruction is 1 (i.e. INPUT_2 from the micro-shift COUNTER-cycle represents a negative integer), i.e. if $\mathsf{NEG}_i = 1$.

2. IF $\mathsf{IN\_RANGE\_FLAG}_i = 1$

   (a) $\mathsf{BYTE\_1}_i$ is deduced simply:

   $$\mathsf{BYTE\_1}_i = \mathsf{BYTE\_RES}_{i-32}$$

   In other words, after the first COUNTER-cycle of a shift (i.e. the micro-shift) the following COUNTER-cycles of that shift-instruction copy BYTE_RES from the previous COUNTER-cycle into the current COUNTER-cycle's BYTE_1 column.

   (b) $\mathsf{BYTE\_2}_i$ is more involved:

      i. IF $\mathsf{SHIFT\_DIRECTION} = 0$

         A. IF $\mathsf{ZERO\_BACP}_i \neq 0$:

         $$\begin{aligned}\mathsf{BYTE\_2}_i \quad = \quad & \llbracket 2\rrbracket_i \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32+1} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i \cdot \llbracket 0\rrbracket_i \cdot \mathsf{BYTE\_RES}_{i-32+2} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i \cdot \llbracket 1\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32+4} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i \cdot \mathsf{BYTE\_RES}_{i-32+8} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32+16}\end{aligned}$$

         B. IF $\mathsf{ZERO\_BACP}_i = 0$:

         $$\mathsf{BYTE\_2}_i = 0$$

      ii. IF $\mathsf{SHIFT\_DIRECTION}_i = 1$

         A. IF $\mathsf{ZERO\_BACP}_i \neq 0$:

         $$\mathsf{BYTE\_2}_i = 255 \cdot \mathsf{NEG}_i$$

         Note that $\mathsf{NEG}_i \neq 0$ during a shift-instruction can only happen for `SAR` instructions.

         B. IF IF $\mathsf{ZERO\_BACP}_i = 0$:

         $$\begin{aligned}\mathsf{BYTE\_2}_i \quad = \quad & \llbracket 2\rrbracket_i \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32-1} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i \cdot \llbracket 0\rrbracket_i \cdot \mathsf{BYTE\_RES}_{i-32-2} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i \cdot \llbracket 1\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32-4} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i \cdot \mathsf{BYTE\_RES}_{i-32-8} \\ + \; & \llbracket 2\rrbracket_i^\vee \cdot \llbracket 1\rrbracket_i^\vee \cdot \llbracket 0\rrbracket_i^\vee \cdot \mathsf{BYTE\_RES}_{i-32-16}\end{aligned}$$

   (c) $\mathsf{BYTE\_RES}_i$ is deduced simply from $\mathsf{BYTE\_2}_i$, $\mathsf{BYTE\_1}_i$ and $\mathsf{DECISION\_BIT}_i$:

   $$\mathsf{BYTE\_RES}_i = \mathsf{DB}_i \cdot \mathsf{BYTE\_2}_i + (1 - \mathsf{DB}_i) \cdot \mathsf{BYTE\_1}_i$$

   in other words,

   $$\begin{cases} \text{IF } \mathsf{DB}_i = 1 : & \mathsf{BYTE\_RES}_i = \mathsf{BYTE\_2}_i \\ \text{IF } \mathsf{DB}_i = 0 : & \mathsf{BYTE\_RES}_i = \mathsf{BYTE\_1}_i \end{cases}$$

10.1.6 Pivot-instruction constraints

$$\boxed{\text{The following constraints apply when } \mathsf{PIVOT\_FLAG}_i = 1.}$$

We start by verifying bits from the BYTE_BITS column.

1. IF $\mathsf{COUNTER}_i = 0$ THEN :
   (a) $\mathsf{B1}_i = \sum_{k=0}^{7} 2^k \cdot \mathsf{BB}_{i-k} \in [\![0, 256[\![$
   (b) $\mathsf{PB}_i = \sum_{k=0}^{7} 2^k \cdot \mathsf{BB}_{i-8-k} \in [\![0, 256[\![$
   (c) we specify NEG_i:
   $$\begin{cases} \text{IF } \mathsf{SGNXF} = 1: & \mathsf{NEG}_i = \mathsf{BB}_{i-15} \in \{0, 1\} \\ \text{IF } \mathsf{SGNXF} = 0: & \mathsf{NEG}_i = 0 \end{cases}$$

   The first condition verifies the final 8 bits of BB in the COUNTER-cycle as being those of the least significant byte of INPUT_1 — this matters both for pivot-instructions. The second condition verifies the preceding 8 bits as being those of PIVOT_BYTE and the third condition verifies the sign bit NEG — these two conditions only matter for `SGNX`.

   NOTE. The first two constraints were already mentioned in section 10.1.4.

   (d) we specify ZP_i:
   $$\begin{cases} \text{IF } \mathsf{SGNXF}_i = 1: & \mathsf{ZP}_i = \sum_{k=0}^{4} 2^k \cdot \mathsf{BB}_{i-k} \in [\![0, 32[\![\\ \text{IF } \mathsf{SGNXF}_i = 0: & \mathsf{ZP}_i = 31 - \sum_{k=0}^{4} 2^k \cdot \mathsf{BB}_{i-k} \in [\![0, 32[\![\end{cases}$$

   The value of ZP depends on whether we the instruction is `SGNX` or `BYTE`. The reason for this discrepancy is that the position of the pivot byte for a `SGNX` instruction (i.e. the byte containing the sign bit) is defined by its offset from INPUT_2's *least significant byte*, while the position of the pivot byte for a `BYTE` instruction (i.e. the byte that the instruction selects and returns) is defined by its offset from INPUT_2's *most significant byte*.

NOTE. Recall that NEG, PB, ZP are locally-constant columns so the preceding constraints completely fix these columns for the full COUNTER-cycle of `BYTE` and `SGNX` instructions.

We now move on to obtaining the PB from the byte decomposition of I2.

2. IF $\mathsf{ZERO\_BACP}_i = 0$ AND $\mathsf{ZERO\_BACP}_{i-1} \neq 0$ AND $\mathsf{COUNTER}_i \neq 31$ THEN $\mathsf{PIVOT\_BYTE}_i = \mathsf{BYTE\_2}_i$
3. IF $\mathsf{ZERO\_BACP}_i = 0$ AND $\mathsf{COUNTER}_i = 31$ THEN $\mathsf{PIVOT\_BYTE}_i = \mathsf{BYTE\_2}_i$

In most cases we recognize the pivot byte as the byte from the BYTE_2 column in the first row where ZB switches from a nonzero value to 0. If $\mathsf{ZP} = 31$ there is no such switch, and so the above constraint corrects for that.

We now constrain BYTE_RES

4. IF $\mathsf{SGNXF}_i = 1$, i.e. for a `SIGNEXTEND` instruction, there are two cases to distinguish. If INPUT_1 isn't in range then the result is just INPUT_2 itself. If, on the other hand, INPUT_1 is in range the result may need to be padded with 0x00's or `0xff`'s according to the sign bit NEG:
   (a) IF $\mathsf{IRF}_i = 0$
   $$\mathsf{BYTE\_RES}_i = \mathsf{BYTE\_2}_i$$
   (b) IF $\mathsf{IRF}_i = 1$
   $$\begin{cases} \text{IF } \mathsf{ZB}_i \neq 0 & \text{THEN } \mathsf{BYTE\_RES}_i = 255 \cdot \mathsf{NEG}_i \\ \text{IF } \mathsf{ZB}_i = 0 & \text{THEN } \mathsf{BYTE\_RES}_i = \mathsf{BYTE\_2}_i \end{cases}$$

in other words, we discard all bytes from $\mathsf{ZERO\_BACP}$ preceding the pivot byte and replace them with zeros if the sign bit of the pivot byte is 0 or with ones if the sign bit of the pivot byte is 1, and keep all bytes from $\mathsf{ZERO\_BACP}$ following (and including) the pivot byte.

5. $\textsc{if}$ $\mathsf{SGNXF}_i = 0$, i.e. for a `BYTE` instruction:

$$\begin{cases} \textsc{if} \ \mathsf{ZB}_i \neq 0 & \textsc{then} \ \mathsf{BYTE\_RES}_i = 0 \\ \textsc{if} \ \mathsf{ZB}_i = 0 & \textsc{then} \ \mathsf{BYTE\_RES}_i = \mathsf{PB}_i \cdot \mathsf{IRF}_i \end{cases}$$

in other words, the first 31 bytes of the result are always zero, and the final byte is the pivot byte if $\mathsf{INPUT\_1}$ is in range, otherwise it's 0.

COUNTER	SGNXF	ZB	BYTE_2	PB	BB	NEG	ZP	BYTE_1	BYTE_RES
31	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
30	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
29	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
28	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
27	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
26	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
25	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
24	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
23	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
22	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
21	1	⋆	· · ·	PB	0 / 1	PB_7	ZP	· · ·	pad
20	1	0	PB	PB	0 / 1	PB_7	ZP	· · ·	PB
19	1	0	⋆ ⋆ ⋆	PB	0 / 1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
18	1	0	⋆ ⋆ ⋆	PB	0 / 1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
17	1	0	⋆ ⋆ ⋆	PB	0 / 1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
16	1	0	⋆ ⋆ ⋆	PB	0 / 1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
15	1	0	⋆ ⋆ ⋆	PB	PB_7	PB_7	ZP	· · ·	⋆ ⋆ ⋆
14	1	0	⋆ ⋆ ⋆	PB	PB_6	PB_7	ZP	· · ·	⋆ ⋆ ⋆
13	1	0	⋆ ⋆ ⋆	PB	PB_5	PB_7	ZP	· · ·	⋆ ⋆ ⋆
12	1	0	⋆ ⋆ ⋆	PB	PB_4	PB_7	ZP	· · ·	⋆ ⋆ ⋆
11	1	0	⋆ ⋆ ⋆	PB	PB_3	PB_7	ZP	· · ·	⋆ ⋆ ⋆
10	1	0	⋆ ⋆ ⋆	PB	PB_2	PB_7	ZP	· · ·	⋆ ⋆ ⋆
9	1	0	⋆ ⋆ ⋆	PB	PB_1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
8	1	0	⋆ ⋆ ⋆	PB	PB_0	PB_7	ZP	· · ·	⋆ ⋆ ⋆
7	1	0	⋆ ⋆ ⋆	PB	LSB_7	PB_7	ZP	· · ·	⋆ ⋆ ⋆
6	1	0	⋆ ⋆ ⋆	PB	LSB_6	PB_7	ZP	· · ·	⋆ ⋆ ⋆
5	1	0	⋆ ⋆ ⋆	PB	LSB_5	PB_7	ZP	· · ·	⋆ ⋆ ⋆
4	1	0	⋆ ⋆ ⋆	PB	LSB_4	PB_7	ZP	· · ·	⋆ ⋆ ⋆
3	1	0	⋆ ⋆ ⋆	PB	LSB_3	PB_7	ZP	· · ·	⋆ ⋆ ⋆
2	1	0	⋆ ⋆ ⋆	PB	LSB_2	PB_7	ZP	· · ·	⋆ ⋆ ⋆
1	1	0	⋆ ⋆ ⋆	PB	LSB_1	PB_7	ZP	· · ·	⋆ ⋆ ⋆
0	1	0	⋆ ⋆ ⋆	PB	LSB_0	PB_7	ZP	LSB	⋆ ⋆ ⋆
	Loc. Cst. *Binary*			*Loc. Cst.*	*Binary*	*Loc. Cst.* *Binary*	*Loc. Cst.*		

Figure 10.7: A full COUNTER-cycle's worth of columns of a `SGNX` instruction in the case where $\text{IRF} = 1$. Time flow is from top to bottom. The padding `pad` in the BYTE_RES column is either `0x00` or `0xff` according to whether $\text{NEG} = 0$ or $\text{NEG} = 1$. The $\texttt{PB}_i$, $i = 0, \ldots, 7$, are the bits of the pivot byte `PB`, the $\texttt{LSB}_i$, $i = 0, \ldots, 7$, are the bits of the least significant byte `LSB` of INPUT_1.

Chapter 11

ALU

11.1 ALU Dispatcher

11.1.1 ALU DISPATCHER

The ALU DISPATCHER is the intermediary between the hub and the ALU256. It's role is to decompose complex arithmetic opcodes (`ADD`, `MUL`, `SUB`, `DIV`, ...) into a sequence of `ADD` / `MUL` operations to be transmitted to the ALU256.

Instructions treated

- `ADD`
- `MUL`
- `SUB`
- `DIV`
- `MOD`
- `EXP`
- `SMOD`
- `SDIV`
- `ADDMOD`
- `MULMOD`

Trace columns

Main Execution columns

- INST
- $\mathsf{ARG}^{i,\{high,low\},\mathsf{DISP}}, i \in [0,1]$: Contains the $i^{th}, i \in [1,2]$ input of the operation.
- $\mathsf{OUT}^{\{high,low\},\mathsf{DISP}}$: Contains the result of the operation to be transmitted back to the ALU DISPATCHER.
- $\langle \mathsf{ALU}\square \rangle$

ALU256 link columns: $i \in [0,1]$: Contains the i^{th} input of the operation.
$k \in [0,3]$: Contains input for the k^{th} register.

- $\mathsf{ALU}\square^{k,256}$
- $^{\diamond}\mathsf{ADD\_FLAG}^{k,256}$
- $^{\diamond}\mathsf{MUL\_FLAG}^{k,256}$
- $\mathsf{ARG}^{i,k,\{high,low\},256}, i \in [0,1]$: Contains the $i^{th}, i \in [1,2]$ input of the operation.

- $\mathsf{OUT}^{k,\{high,low\},256}$: Contains the result of the operation to be transmitted back to the ALU DISPATCHER.
- $\mathsf{OVERFLOW\_FLAG}^k$: is set if the ALU256 result has overflown

Instruction decoder columns These columns, combined with the INST column, should be included in the instruction decoder.

- $^{\Diamond}\mathsf{ADD\_FLAG}$
- $^{\Diamond}\mathsf{SUB\_FLAG}$
- $^{\Diamond}\mathsf{MUL\_FLAG}$
- $^{\Diamond}\mathsf{DIV\_FLAG}$
- $^{\Diamond}\mathsf{MOD\_FLAG}$
- $^{\Diamond}\mathsf{EXP\_FLAG}$

Auxiliary columns for DIV/MOD/SMOD/SDIV/EXP operations

- $\mathsf{QUOTIENT}^i, i \in [0,3]$: auxiliary variables that contains 128 bit decomposition of the quotient. (The quotient for the ADDMOD/`MULMOD` operation is a 512 bit number)
- For the `SMOD`/`SDIV`
  $\mathsf{BIT\_0}, \mathsf{ACC\_CARRY\_0}$ are used to calulate bit decompoistion of the dividend.
  $\mathsf{BIT\_1}, \mathsf{ACC\_CARRY\_1}$ are used to calulate bit decompoistion of the quotient.
- For the EXP
  $\mathsf{BIT\_0}, \mathsf{ACC\_CARRY\_0}$ are used to calulate bit decompoistion of the exponent.
- $\mathsf{REM}^{high,low}$ auxiliary variables that contains high and low bits of the remainder for `MOD`, `DIV`, `SMOD`, `SDIV`, `ADDMOD`, `MULMOD`.
- $\mathsf{DIVIDEND}^{high,low}$ auxiliary variables that contains high and low bits of the dividend for `MOD`, `DIV`, `SMOD`, `SDIV`, `ADDMOD`, `MULMOD`.
- $\mathsf{STEP\_FLAG}^j, j \in [0,3]$: step flag for the ALU DISPATCHER (mod operation)

Constraint set

1. $\langle \mathsf{ALU}\square \rangle$:

$$\begin{cases} \langle \mathsf{ALU}\square \rangle_0 = 0 \\ \langle \mathsf{ALU}\square \rangle_{i+1} \in \{\langle \mathsf{ALU}\square \rangle_i, 1 + \langle \mathsf{ALU}\square \rangle_i\} \end{cases}$$

2. IF $\langle \mathsf{ALU}\square \rangle_i = 0$: then the entire i-th row is null; in particular the first row is all zeros;
3. IF $^{\Diamond}\mathsf{ADD\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$: perform an addition

   (a) Set the inputs and results for the ALU256:

$$\begin{cases} \mathsf{ARG}_i^{j,\{high,low\},256} = \mathsf{ARG}_i^{j,\{high,low\},\mathsf{DISP}}, j \in [0,1] \\ \mathsf{OUT}_i^{\{high,low\},256} = \mathsf{OUT}_i^{\{high,low\},\mathsf{DISP}} \\ \\ ^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ ^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \end{cases}$$

(b) Update the $\mathsf{ALU}\square^{i,256}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(c) Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

4. ELSEIF $^{\Diamond}\mathsf{SUB\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$: perform a subtraction

(a) Set the inputs and results for the ALU256:

$$\begin{cases} \mathsf{ARG}_i^{0,\{high,low\},256} = \mathsf{OUT}_i^{\{high,low\},\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,\{high,low\},256} = \mathsf{ARG}_i^{1,\{high,low\},\mathsf{DISP}} \\ \mathsf{OUT}_i^{\{high,low\},256} = \mathsf{ARG}_i^{0,\{high,low\},\mathsf{DISP}} \\ \\ ^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ ^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \end{cases}$$

(b) Update the $\mathsf{ALU}\square^{i,256}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(c) Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

5. ELSEIF $^{\Diamond}\mathsf{MUL\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$: perform a multiplication

(a) Set the inputs and results for the ALU256:

$$\begin{cases} \mathsf{ARG}_i^{j,\{high,low\},256} = \mathsf{ARG}_i^{j,\{high,low\},\mathsf{DISP}}, j \in [0,1] \\ \mathsf{OUT}_i^{\{high,low\},256} = \mathsf{OUT}_i^{\{high,low\},\mathsf{DISP}} \\ \\ ^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 0 \\ ^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 1 \end{cases}$$

(b) Update the $\mathsf{ALU}\square^{i,256}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(c) Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

6. ELSEIF $^\diamond\mathsf{MOD\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$: modulo operation
The modulo operation asserts the following equality:

$$\mathsf{DIVIDEND} = MOD * \mathsf{QUOTIENT} + \mathsf{REM};\ \mathsf{REM} < MOD$$

(a) Initialize DIVIDEND, MOD and REM columns:

$$\left\{ \begin{array}{l} \mathsf{DIVIDEND}_i^{low} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{DIVIDEND}_i^{high} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \\ MOD_i^{low} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{REM}_i^{low} = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{REM}^{high} = \mathsf{OUT}_i^{high,\mathsf{DISP}} \end{array} \right.$$

(b) Set the inputs for the $register^0$ and assert that $\mathsf{REM} < MOD$

$$\left\{ \begin{array}{l} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{REM}_i^{low} \\ \mathsf{ARG}_i^{0,0,high,256} = \mathsf{REM}_i^{high} \\ \\ \mathsf{OUT}_i^{0,low,256} = MOD_i^{low} \\ \mathsf{OUT}_i^{0,high,256} = MOD_i^{high} \\ \\ ^\diamond\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ ^\diamond\mathsf{MUL\_FLAG}_i^{0,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^0 = 0 \end{array} \right.$$

(c) Set the inputs for the $register^1$ in order to constrain result on $MOD * \mathsf{QUOTIENT}$

$$\left\{ \begin{array}{l} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{QUOTIENT}_i^0 \\ \mathsf{ARG}_i^{0,1,high,256} = \mathsf{QUOTIENT}_i^1 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{1,1,high,256} = MOD_i^{high} \\ \\ ^\diamond\mathsf{ADD\_FLAG}_i^{1,256} = 0 \\ ^\diamond\mathsf{MUL\_FLAG}_i^{1,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^1 = 0 \end{array} \right.$$

(d) Set the inputs for the $register^2$ in order to constrain result on $\mathsf{OUT}^2 == \mathsf{DIVIDEND}$

i. Set inputs

$$\left\{ \begin{array}{l} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{OUT}_i^{1,low,256} \\ \mathsf{ARG}_i^{0,2,high,256} = \mathsf{OUT}_i^{1,high,256} \\ \\ \mathsf{ARG}_i^{1,2,low,256} = \mathsf{REM}_i^{low} \\ \mathsf{ARG}_i^{1,2,high,256} = \mathsf{REM}_i^{high} \\ \\ ^\diamond\mathsf{ADD\_FLAG}_i^{02,256} = 1 \\ ^\diamond\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{array} \right.$$

ii. IF $\boldsymbol{MOD_i^{low}} \boldsymbol{\neq 0}$ or $\boldsymbol{MOD_i^{high}} \boldsymbol{\neq 0}$

$$\begin{cases} \mathsf{OUT}_i^{2,low,256} = \mathsf{DIVIDEND}_i^{low} \\ \mathsf{OUT}_i^{2,high,256} = \mathsf{DIVIDEND}_i^{high} \end{cases}$$

iii. ELSEIF $\boldsymbol{MOD_i^{low}{=}0}$ and $\boldsymbol{MOD_i^{high}{=}0}$ special case for $MOD = 0$

$$\begin{cases} \mathsf{OUT}_i^{2,low,256} = 0 \\ \mathsf{OUT}_i^{2,high,256} = 0 \end{cases}$$

(e) Update the $\mathsf{ALU}\square^{i,256}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(f) Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

7. ELSEIF $^{\diamond}\mathsf{DIV\_FLAG}_{\boldsymbol{i}}\boldsymbol{=1}$: perform a division

(a) Initialize $\mathsf{DIVIDEND}$, MOD and $\mathsf{QUOTIENT}$columns:

$$\begin{cases} \mathsf{DIVIDEND}_i^{low} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{DIVIDEND}_i^{high} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \\ MOD_i^{low} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{QUOTIENT}_i^0 = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{QUOTIENT}_i^1 = \mathsf{OUT}_i^{high,\mathsf{DISP}} \end{cases}$$

(b) Same constraints as for MOD: 6b, 6c, 6d, 6e, 6f

8. ELSEIF $^{\diamond}\mathsf{SMOD\_FLAG}_{\boldsymbol{i}}\boldsymbol{=1}$:

(a) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\boldsymbol{0}}\boldsymbol{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\boldsymbol{1}}\boldsymbol{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\boldsymbol{2}}\boldsymbol{=0}$: first step:

i. Initialize $\mathsf{ACC\_CARRY}$

$$\begin{cases} \mathsf{ACC\_CARRY\_0}_i = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ACC\_CARRY\_1}_i = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \end{cases}$$

ii. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \end{cases}$$

iii. $\mathsf{ALU\square}^{k,256}, k \in [0,3]$ remains constant

$$\begin{cases} \mathsf{ALU\square}_i^{0,256} = \mathsf{ALU\square}_{i-1}^{3,256} \\ \mathsf{ALU\square}_i^{1,256} = \mathsf{ALU\square}_i^{0,256} \\ \mathsf{ALU\square}_i^{2,256} = \mathsf{ALU\square}_i^{1,256} \\ \mathsf{ALU\square}_i^{3,256} = \mathsf{ALU\square}_i^{2,256} \end{cases}$$

iv. Set $\mathsf{STEP\_COUNTER}$ for new operation:

$$\mathsf{STEP\_COUNTER}_i = 0$$

v. Set $\langle\mathsf{ALU\square}\rangle$ for new operation:

$$\langle\mathsf{ALU\square}\rangle_i = \langle\mathsf{ALU\square}\rangle_{i-1} + 1$$

vi. The $\langle\mathsf{ALU\square}\rangle$ stamp remains constant:

$$\langle\mathsf{ALU\square}\rangle_{i+1} = \langle\mathsf{ALU\square}\rangle_i$$

vii. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

(b) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$:

i. IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}\mathbf{=127}$ The operation should be over

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

ii. ELSEIF $\mathsf{STEP\_COUNTER}_{\boldsymbol{ii}}^{\mathbf{0}}\ \mathbf{\not=127}$

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \\ \mathsf{STEP\_COUNTER}_{i+1} = \mathsf{STEP\_COUNTER}_i + 1 \end{cases}$$

iii. $\mathsf{ACC\_CARRY}$ bit decomposition

$$\begin{cases} \mathsf{BIT\_0}_i = \mathsf{BIT\_0}_i * \mathsf{BIT\_0}_i \\ \mathsf{ACC\_CARRY\_0}_i = 2 * \mathsf{ACC\_CARRY\_0}_{i+1} + \mathsf{BIT\_0}_i \\ \\ \mathsf{BIT\_1}_i = \mathsf{BIT\_1}_i * \mathsf{BIT\_1}_i \\ \mathsf{ACC\_CARRY\_1}_i = 2 * \mathsf{ACC\_CARRY\_1}_{i+1} + \mathsf{BIT\_1}_i \end{cases}$$

iv. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \end{cases}$$

v. $\mathsf{ALU}\square^{k,256}, k \in [0,3]$ remains constant

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

vi. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

(c) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$:

i. Same constraints as for MOD: 6b, 6c, 6d, 6e,

ii. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^{0} = \mathsf{QUOTIENT}_i^{0} \\ \mathsf{QUOTIENT}_{i+1}^{1} = \mathsf{QUOTIENT}_i^{1} \end{cases}$$

iii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

iv. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 0 \end{cases}$$

(d) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$:

i. IF $\mathsf{BIT\_1}_{\boldsymbol{i}}\mathbf{=1}$ Constraint for $MOD = -\mathsf{ARG}^{1,0,\mathsf{DISP}}$

A. Inputs and results:

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{0,0,high,256} = MOD_i^{high} \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{ARG}_i^{1,0,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{ARG}_i^{1,0,high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{0,low,256} = 0 \\ \mathsf{OUT}^{0,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 1 \end{cases}$$

B. IF $\boldsymbol{MOD_i^{low}}\mathbf{=0}$ and $\boldsymbol{MOD_i^{high}}\mathbf{=0}$ special case for 0=-0

$$\mathsf{OVERFLOW\_FLAG}^{0} = 0$$

C. ELSEIF $\boldsymbol{MOD_i^{low}} \not= \mathbf{0}$ or $\boldsymbol{MOD_i^{high}} \not= \mathbf{0}$

$$\mathsf{OVERFLOW\_FLAG}^0 = 1$$

ii. ELSEIF $\mathsf{BIT\_1}_{\boldsymbol{i}}\mathbf{=0}$ $MOD = \mathsf{ARG}^{0,0,256}$

$$\begin{cases} MOD_i^{low} = \mathsf{ARG}_i^{1,0,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{1,0,high,\mathsf{DISP}} \end{cases}$$

iii. IF $\mathsf{BIT\_0}_{\boldsymbol{i}}\mathbf{=1}$ $\mathsf{REM} = -\mathsf{OUT}^{\mathsf{DISP}}$

A. Inputs and results:

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{REM}_i^{low} \\ \mathsf{ARG}_i^{0,0,high,256} = \mathsf{REM}_i^{high} \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{OUT}_i^{high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{0,low,256} = 0 \\ \mathsf{OUT}^{0,high,256} = 0 \\ \\ {}^{\diamondsuit}\mathsf{ADD\_FLAG}_i^{0,256} = 0 \\ {}^{\diamondsuit}\mathsf{MUL\_FLAG}_i^{0,256} = 1 \end{cases}$$

B. IF $\mathsf{REM}_{\boldsymbol{i}}^{\boldsymbol{low}}\mathbf{=0}$ and $\mathsf{REM}_{\boldsymbol{i}}^{\boldsymbol{high}}\mathbf{=0}$

$$\mathsf{OVERFLOW\_FLAG}^0 = 0$$

C. ELSEIF $\mathsf{REM}_{\boldsymbol{i}}^{\boldsymbol{low}} \not= \mathbf{0}$ or $\mathsf{REM}_{\boldsymbol{i}}^{\boldsymbol{high}} \not= \mathbf{0}$

$$\mathsf{OVERFLOW\_FLAG}^0 = 1$$

iv. ELSEIF $\mathsf{BIT\_0}_{\boldsymbol{i}}\mathbf{=0}$ $\mathsf{REM} = \mathsf{OUT}^{\mathsf{DISP}}$

$$\begin{cases} \mathsf{REM}_i^{low} = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{REM}_i^{high} = \mathsf{OUT}_i^{high,\mathsf{DISP}} \end{cases}$$

v. Update the $\mathsf{ALU}\square^{k,256}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + \mathsf{BIT\_0} \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + \mathsf{BIT\_1} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

vi. Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

vii. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

9. ELSEIF ${}^{\diamondsuit}\mathsf{SDIV\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$:

(a) The same constraints as for SMOD: 8a, 8b, 8c

(b) IF $\mathsf{STEP\_FLAG}_i^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_i^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_i^{\mathbf{2}}\mathbf{=0}$:

- i. IF $\mathsf{BIT\_0}_i\mathbf{=1}$ and $\mathsf{BIT\_1}_i\mathbf{=1}$;
  $MOD = -\mathsf{ARG}^{1,\mathsf{DISP}}$ and $\mathsf{QUOTIENT} = \mathsf{OUT}^{1,\mathsf{DISP}}$

  - A. Set inputs

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{0,0,high,256} = MOD_i^{high} \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{0,low,256} = 0 \\ \mathsf{OUT}^{0,high,256} = 0 \\ \\ \mathsf{QUOTIENT}_i^{0} = \mathsf{OUT}^{low,\mathsf{DISP}} \\ \mathsf{QUOTIENT}_i^{1} = \mathsf{OUT}^{high,\mathsf{DISP}} \\ \\ {}^{\diamondsuit}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ {}^{\diamondsuit}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \end{cases}$$

  - B. IF $\boldsymbol{MOD_i^{low}}\mathbf{=0}$ and $\boldsymbol{MOD_i^{high}}\mathbf{=0}$ special case for 0=-0

$$\mathsf{OVERFLOW\_FLAG}^{0} = 0$$

  - C. ELSEIF $\boldsymbol{MOD_i^{low}}\ \mathbf{\neq 0}$ or $\boldsymbol{MOD_i^{high}}\ \mathbf{\neq 0}$

$$\mathsf{OVERFLOW\_FLAG}^{0} = 1$$

- ii. ELSEIF $\mathsf{BIT\_0}_i\mathbf{=1}$ and $\mathsf{BIT\_1}_i\mathbf{=0}$
  $MOD = \mathsf{ARG}^{1,\mathsf{DISP}}$ and $\mathsf{QUOTIENT} = -\mathsf{OUT}^{1,\mathsf{DISP}}$

  - A. Inputs and results:

$$\begin{cases} MOD_i^{low} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{ARG}_i^{0,1,low,256} = \mathsf{QUOTIENT}_i^{0} \\ \mathsf{ARG}_i^{0,1,high,256} = \mathsf{QUOTIENT}_i^{1} \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{OUT}^{low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,1,high,256} = \mathsf{OUT}^{high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{0,low,256} = 0 \\ \mathsf{OUT}^{0,high,256} = 0 \\ \\ {}^{\diamondsuit}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ {}^{\diamondsuit}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \end{cases}$$

  - B. IF $\mathsf{QUOTIENT}_i^{\mathbf{0}}\mathbf{=0}$ and $\mathsf{QUOTIENT}_i^{\mathbf{1}}\mathbf{=0}$ special case for 0=-0

$$\mathsf{OVERFLOW\_FLAG}^{1} = 0$$

C. ELSEIF $\mathsf{QUOTIENT}_i^{\mathbf{0}} \not= \mathbf{0}$ or $\mathsf{QUOTIENT}_i^{\mathbf{1}} \not= \mathbf{0}$

$$\mathsf{OVERFLOW\_FLAG}^1 = 1$$

iii. ELSEIF $\mathsf{BIT\_0}_{\boldsymbol{i}}\mathbf{=0}$ and $\mathsf{BIT\_1}_{\boldsymbol{i}}\mathbf{=1}$
$MOD = -\mathsf{ARG}^{1,\mathsf{DISP}}$ and $\mathsf{QUOTIENT} = -\mathsf{OUT}^{1,\mathsf{DISP}}$

A. Inputs and results for MOD:

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{0,0,high,256} = MOD_i^{high} \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{0,low,256} = 0 \\ \mathsf{OUT}^{0,high,256} = 0 \\ \\ {}^\diamond\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ {}^\diamond\mathsf{MUL\_FLAG}_i^{0,256} = 0 \end{cases}$$

B. IF $\boldsymbol{MOD_i^{low}=0}$ and $\boldsymbol{MOD_i^{high}=0}$ special case for 0=-0

$$\mathsf{OVERFLOW\_FLAG}^0 = 0$$

C. ELSEIF $\boldsymbol{MOD_i^{low}} \not= \mathbf{0}$ or $\boldsymbol{MOD_i^{high}} \not= \mathbf{0}$

$$\mathsf{OVERFLOW\_FLAG}^0 = 1$$

D. Inputs and results for QUOTIENT:

$$\begin{cases} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{QUOTIENT}_i^0 \\ \mathsf{ARG}_i^{0,1,high,256} = \mathsf{QUOTIENT}_i^1 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,1,high,256} = \mathsf{OUT}_i^{high,\mathsf{DISP}} \\ \\ \mathsf{OUT}^{1,low,256} = 0 \\ \mathsf{OUT}^{1,high,256} = 0 \\ \\ {}^\diamond\mathsf{ADD\_FLAG}_i^{1,256} = 1 \\ {}^\diamond\mathsf{MUL\_FLAG}_i^{1,256} = 0 \end{cases}$$

E. IF $\mathsf{QUOTIENT}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ and $\mathsf{QUOTIENT}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ special case for 0=-0

$$\mathsf{OVERFLOW\_FLAG}^1 = 0$$

F. ELSEIF $\mathsf{QUOTIENT}_i^{\mathbf{0}} \not= \mathbf{0}$ or $\mathsf{QUOTIENT}_i^{\mathbf{1}} \not= \mathbf{0}$

$$\mathsf{OVERFLOW\_FLAG}^1 = 1$$

iv. ELSEIF $\mathsf{BIT\_0}_i$**=0** and $\mathsf{BIT\_1}_i$**=0**
$MOD = \mathsf{ARG}^{1,\mathsf{DISP}}$ and $\mathsf{QUOTIENT} = \mathsf{OUT}^{1,\mathsf{DISP}}$

A. Set MOD and $\mathsf{QUOTIENT}$

$$\begin{cases} MOD_i^{low} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \\ \mathsf{QUOTIENT}_i^0 = \mathsf{OUT}^{low,\mathsf{DISP}} \\ \mathsf{QUOTIENT}_i^1 = \mathsf{OUT}^{high,\mathsf{DISP}} \end{cases}$$

(c) Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + \mathsf{BIT\_1} \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + xor(\mathsf{BIT\_0}, \mathsf{BIT\_1}) \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(d) Increase the ALU DISPATCHER stamp:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

(e) Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

10. ELSEIF `MULMOD`**=1**:
`MULMOD` has to satisfy the following expression:

$$\mathsf{ARG}^{0,\mathsf{DISP}} * \mathsf{ARG}^{1,\mathsf{DISP}} = \mathsf{QUOTIENT} * MOD + \mathsf{REM};\ \mathsf{REM} < MOD$$

where

- $\mathsf{ARG}^{0,\mathsf{DISP}}$: 32 byte number
- $\mathsf{ARG}^{1,\mathsf{DISP}}$: 32 byte number
- MOD: 32 byte number
- $\mathsf{QUOTIENT}$: 64 byte number
- REM: 32 byte number

Since we can't perform 64 byte arithmetic directly, we need to decompose the above expression in terms of 128 bytes:

$$\mathsf{ARG}^{0,\mathsf{DISP}} * \mathsf{ARG}^{1,\mathsf{DISP}} = x_0 + x_1 * 2^{128} + x_2 * 2^{256} + x_3 * 2^{384}$$

$$\mathsf{QUOTIENT} * MOD + \mathsf{REM} = y_0 + y_1 * 2^{128} + y_2 * 2^{256} + y_3 * 2^{384}$$

and compare only the relevant limbs $x_0 = y_0$, $x_1 = y_1$...
In steps 0-4: decompose:

$$\mathsf{QUOTIENT} * MOD + \mathsf{REM}$$

(a) IF $\mathsf{STEP\_FLAG}_i^{\mathbf{0}}$**=0** AND $\mathsf{STEP\_FLAG}_i^{\mathbf{1}}$**=0** AND $\mathsf{STEP\_FLAG}_i^{\mathbf{2}}$**=0**:

i. Set MOD and REM:

$$\begin{cases} \mathsf{REM}_i^{low} = \mathsf{OUT}^{low,\mathsf{DISP}} \\ \mathsf{REM}_i^{high} = \mathsf{OUT}^{low,HIGH} \\ \\ MOD_i^{low} = \mathsf{ARG}_i^{2,low,\mathsf{DISP}} \\ MOD_i^{high} = \mathsf{ARG}_i^{2,high,\mathsf{DISP}} \end{cases}$$

ii. $Register^0$

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{QUOTIENT}_i^0 \\ \mathsf{ARG}_i^{0,0,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,0,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{1,0,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{0,256} = 0 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{0,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^0 = 0 \end{cases}$$

iii. $Register^1$

$$\begin{cases} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{OUT}_i^{0,low} \\ \mathsf{ARG}_i^{0,1,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{REM}_i^{low} \\ \mathsf{ARG}_i^{1,1,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{1,256} = 1 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{1,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^1 = 0 \end{cases}$$

iv. $Register^2$

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{QUOTIENT}_i^0 \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = MOD_i^{high} \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{2,256} = 0 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{2,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

v. $Register^3$

$$\begin{cases} \mathsf{ARG}_i^{0,3,low,256} = \mathsf{QUOTIENT}_i^1 \\ \mathsf{ARG}_i^{0,3,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,3,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{1,3,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{3,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^3 = 0 \end{cases}$$

vi. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_{i}^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_{i}^{high} \\ \\ MOD_{i+1}^{low} = MOD_{i}^{low} \\ MOD_{i+1}^{high} = MOD_{i}^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^{0} = \mathsf{QUOTIENT}_{i}^{0} \\ \mathsf{QUOTIENT}_{i+1}^{1} = \mathsf{QUOTIENT}_{i}^{1} \\ \mathsf{QUOTIENT}_{i+1}^{2} = \mathsf{QUOTIENT}_{i}^{2} \\ \mathsf{QUOTIENT}_{i+1}^{3} = \mathsf{QUOTIENT}_{i}^{3} \end{cases}$$

vii. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_{i}^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_{i}^{1,256} = \mathsf{ALU}\square_{i}^{0,256} + 1 \\ \mathsf{ALU}\square_{i}^{2,256} = \mathsf{ALU}\square_{i}^{1,256} + 1 \\ \mathsf{ALU}\square_{i}^{3,256} = \mathsf{ALU}\square_{i}^{2,256} + 1 \end{cases}$$

viii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_{i}$$

ix. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 0 \end{cases}$$

(b) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$:

i. $Register^{0}$

$$\begin{cases} \mathsf{ARG}_{i}^{0,0,low,256} = \mathsf{QUOTIENT}_{i}^{1} \\ \mathsf{ARG}_{i}^{0,0,high,256} = 0 \\ \\ \mathsf{ARG}_{i}^{1,0,low,256} = MOD_{i}^{high} \\ \mathsf{ARG}_{i}^{1,0,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_{i}^{0,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_{i}^{0,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^{0} = 0 \end{cases}$$

ii. $Register^{1}$

$$\begin{cases} \mathsf{ARG}_{i}^{0,1,low,256} = \mathsf{QUOTIENT}_{i}^{2} \\ \mathsf{ARG}_{i}^{0,1,high,256} = 0 \\ \\ \mathsf{ARG}_{i}^{1,1,low,256} = MOD_{i}^{low} \\ \mathsf{ARG}_{i}^{1,1,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_{i}^{1,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_{i}^{1,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^{1} = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{QUOTIENT}_i^3 \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = MOD_i^{low} \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{2,256} = 0 \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{2,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. *Register*^{3}

$$\begin{cases} \mathsf{ARG}_i^{0,3,low,256} = \mathsf{OUT}_{i-1}^{0,high,256} \\ \mathsf{ARG}_i^{0,3,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,3,low,256} = \mathsf{OUT}_{i-1}^{2,low,256} \\ \mathsf{ARG}_i^{1,3,high,256} = 0 \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^3 = 0 \end{cases}$$

v. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \\ \mathsf{QUOTIENT}_{i+1}^2 = \mathsf{QUOTIENT}_i^2 \\ \mathsf{QUOTIENT}_{i+1}^3 = \mathsf{QUOTIENT}_i^3 \end{cases}$$

vi. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} + 1 \end{cases}$$

vii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

viii. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

(c) IF $\mathsf{STEP\_FLAG}_i^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_i^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_i^{\mathbf{2}}\mathbf{=0}$:

i. $Register^0$

$$\left\{\begin{array}{l}
\mathsf{ARG}_i^{0,0,low,256} = \mathsf{OUT}_{i-2}^{3,low,256} \\
\mathsf{ARG}_i^{0,0,high,256} = 0 \\[1em]
\mathsf{ARG}_i^{1,0,low,256} = \mathsf{OUT}_{i-1}^{3,low,256} \\
\mathsf{ARG}_i^{1,0,high,256} = 0 \\[1em]
{}^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^0 = 0
\end{array}\right.$$

ii. $Register^1$

$$\left\{\begin{array}{l}
\mathsf{ARG}_i^{0,1,low,256} = \mathsf{OUT}_{i}^{0,low,256} \\
\mathsf{ARG}_i^{0,1,high,256} = 0 \\[1em]
\mathsf{ARG}_i^{1,1,low,256} = \mathsf{OUT}_{i-2}^{1,high,256} \\
\mathsf{ARG}_i^{1,1,high,256} = 0 \\[1em]
{}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^1 = 0
\end{array}\right.$$

iii. $Register^2$

$$\left\{\begin{array}{l}
\mathsf{ARG}_i^{0,2,low,256} = \mathsf{OUT}_{i}^{1,low,256} \\
\mathsf{ARG}_i^{0,2,high,256} = 0 \\[1em]
\mathsf{ARG}_i^{1,2,low,256} = \mathsf{REM}_{i}^{high} \\
\mathsf{ARG}_i^{1,2,high,256} = 0 \\[1em]
{}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^2 = 0
\end{array}\right.$$

iv. $Register^3$

$$\left\{\begin{array}{l}
\mathsf{ARG}_i^{0,3,low,256} = \mathsf{OUT}_{i-2}^{2,high,256} \\
\mathsf{ARG}_i^{0,3,high,256} = 0 \\[1em]
\mathsf{ARG}_i^{1,3,low,256} = \mathsf{OUT}_{i-2}^{3,high,256} \\
\mathsf{ARG}_i^{1,3,high,256} = 0 \\[1em]
{}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^3 = 0
\end{array}\right.$$

v. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\left\{\begin{array}{l}
\mathsf{REM}_{i+1}^{low} = \mathsf{REM}_{i}^{low} \\
\mathsf{REM}_{i+1}^{high} = \mathsf{REM}_{i}^{high} \\
\\
MOD_{i+1}^{low} = MOD_{i}^{low} \\
MOD_{i+1}^{high} = MOD_{i}^{high} \\
\\
\mathsf{QUOTIENT}_{i+1}^{0} = \mathsf{QUOTIENT}_{i}^{0} \\
\mathsf{QUOTIENT}_{i+1}^{1} = \mathsf{QUOTIENT}_{i}^{1} \\
\mathsf{QUOTIENT}_{i+1}^{2} = \mathsf{QUOTIENT}_{i}^{2} \\
\mathsf{QUOTIENT}_{i+1}^{3} = \mathsf{QUOTIENT}_{i}^{3}
\end{array}\right.$$

vi. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\left\{\begin{array}{l}
\mathsf{ALU}\square_{i}^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\
\mathsf{ALU}\square_{i}^{1,256} = \mathsf{ALU}\square_{i}^{0,256} + 1 \\
\mathsf{ALU}\square_{i}^{2,256} = \mathsf{ALU}\square_{i}^{1,256} + 1 \\
\mathsf{ALU}\square_{i}^{3,256} = \mathsf{ALU}\square_{i}^{2,256} + 1
\end{array}\right.$$

vii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_{i}$$

viii. Set next step flags:

$$\left\{\begin{array}{l}
\mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\
\mathsf{STEP\_FLAG}_{i+1}^{1} = 1 \\
\mathsf{STEP\_FLAG}_{i+1}^{2} = 0
\end{array}\right.$$

(d) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$:

i. $Register^{0}$

$$\left\{\begin{array}{l}
\mathsf{ARG}_{i}^{0,0,low,256} = \mathsf{OUT}_{i-2}^{0,low,256} \\
\mathsf{ARG}_{i}^{0,0,high,256} = 0 \\
\\
\mathsf{ARG}_{i}^{1,0,low,256} = \mathsf{OUT}_{i-2}^{1,low,256} \\
\mathsf{ARG}_{i}^{1,0,high,256} = 0 \\
\\
{}^{\Diamond}\mathsf{ADD\_FLAG}_{i}^{0,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_{i}^{0,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^{0} = 0
\end{array}\right.$$

ii. $Register^{1}$

$$\left\{\begin{array}{l}
\mathsf{ARG}_{i}^{0,1,low,256} = \mathsf{OUT}_{i}^{0,low,256} \\
\mathsf{ARG}_{i}^{0,1,high,256} = 0 \\
\\
\mathsf{ARG}_{i}^{1,1,low,256} = \mathsf{OUT}_{i-2}^{1,high,256} \\
\mathsf{ARG}_{i}^{1,1,high,256} = 0 \\
\\
{}^{\Diamond}\mathsf{ADD\_FLAG}_{i}^{1,256} = 1 \\
{}^{\Diamond}\mathsf{MUL\_FLAG}_{i}^{1,256} = 0 \\
\mathsf{OVERFLOW\_FLAG}^{1} = 0
\end{array}\right.$$

iii. $Register^2$

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{OUT}_i^{1,low,256} \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = \mathsf{OUT}_{i-2}^{3,high,256} + \mathsf{OUT}_{i-1}^{0,high,256} + \mathsf{OUT}_{i-1}^{1,high,256} + \mathsf{OUT}_{i-1}^{2,high,256} \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. $Register^3$

$$\begin{cases} \mathsf{ARG}_i^{0,3,low,256} = \mathsf{OUT}_{i-2}^{0,high,256} \\ \mathsf{ARG}_i^{0,3,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,3,low,256} = \mathsf{OUT}_{i-2}^{1,high,256} + \mathsf{OUT}_{i-2}^{2,low,256} + \mathsf{OUT}_{i-1}^{3,high,256} + \\ \quad \mathsf{OUT}_i^{0,high,256} + \mathsf{OUT}_i^{1,high,256} + \mathsf{OUT}_i^{2,high,256} \\ \mathsf{ARG}_i^{1,3,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^3 = 0 \end{cases}$$

v. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \\ \mathsf{QUOTIENT}_{i+1}^2 = \mathsf{QUOTIENT}_i^2 \\ \mathsf{QUOTIENT}_{i+1}^3 = \mathsf{QUOTIENT}_i^3 \end{cases}$$

vi. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} + 1 \end{cases}$$

vii. REM and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}^{low}_{i+1} = \mathsf{REM}^{low}_{i} \\ \mathsf{REM}^{high}_{i+1} = \mathsf{REM}^{high}_{i} \\ \\ MOD^{low}_{i+1} = MOD^{low}_{i} \\ MOD^{high}_{i+1} = MOD^{high}_{i} \\ \\ \mathsf{QUOTIENT}^{0}_{i+1} = \mathsf{QUOTIENT}^{0}_{i} \\ \mathsf{QUOTIENT}^{1}_{i+1} = \mathsf{QUOTIENT}^{1}_{i} \\ \mathsf{QUOTIENT}^{2}_{i+1} = \mathsf{QUOTIENT}^{2}_{i} \\ \mathsf{QUOTIENT}^{3}_{i+1} = \mathsf{QUOTIENT}^{3}_{i} \end{cases}$$

viii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_{i}$$

ix. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}^{0}_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^{1}_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^{2}_{i+1} = 1 \end{cases}$$

(e) if $\mathsf{STEP\_FLAG}^{\mathbf{0}}_{\boldsymbol{i}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}^{\mathbf{1}}_{\boldsymbol{i}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}^{\mathbf{2}}_{\boldsymbol{i}}\mathbf{=1}$:

i. $Register^{0}$

$$\begin{cases} \mathsf{ARG}^{0,0,low,256}_{i} = \mathsf{QUOTIENT}^{2}_{i} \\ \mathsf{ARG}^{0,0,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,0,low,256}_{i} = MOD^{high}_{i} \\ \mathsf{ARG}^{1,0,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{0,256}_{i} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{0,256}_{i} = 1 \\ \mathsf{OVERFLOW\_FLAG}^{0} = 0 \end{cases}$$

ii. $Register^{1}$ check: $\mathsf{REM} < MOD$

$$\begin{cases} \mathsf{ARG}^{0,1,low,256}_{i} = \mathsf{OUT}^{3,low,256}_{i-1} \\ \mathsf{ARG}^{0,1,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,1,low,256}_{i} = \mathsf{OUT}^{0,low,256}_{i} \\ \mathsf{ARG}^{1,1,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{1,256}_{i} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{1,256}_{i} = 0 \\ \mathsf{OVERFLOW\_FLAG}^{1} = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{REM}_i^{low} \\ \mathsf{ARG}_i^{0,2,high,256} = \mathsf{REM}_i^{high} \\ \\ Ras_i^{2,low,256} = MOD^{low} \\ Ras_i^{2,high,256} = MOD^{high} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} + 0 \end{cases}$$

v. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

vi. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 1 \end{cases}$$

In steps 5-7: decompose:

$$\mathsf{ARG}^{0,low,\mathsf{DISP}} * \mathsf{ARG}_i^{1,low,\mathsf{DISP}}$$

(f) IF $\mathsf{STEP\_FLAG}_i^0\boldsymbol{=1}$ AND $\mathsf{STEP\_FLAG}_i^1\boldsymbol{=0}$ AND $\mathsf{STEP\_FLAG}_i^2\boldsymbol{=1}$:

i. *Register*^{0}

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,0,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^0 = 0 \end{cases}$$

ii. *Register*^{1}

$$\begin{cases} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,1,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,1,high,256} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{1,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{1,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^1 = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ {}^\diamond\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\ {}^\diamond\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. *Register*^{3}

$$\begin{cases} \mathsf{ARG}_i^{0,3,low,256} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,3,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,3,low,256} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,3,high,256} = 0 \\ \\ {}^\diamond\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ {}^\diamond\mathsf{MUL\_FLAG}_i^{3,256} = 1 \\ \mathsf{OVERFLOW\_FLAG}^3 = 0 \end{cases}$$

v. IF $\boldsymbol{MOD^{low}=0}$ **and** $\boldsymbol{MOD^{high}=0}$

$$\mathsf{OUT}_{i-5}^{1,low,256} = 0$$

vi. ELSEIF $\boldsymbol{MOD^{low}\neq 0}$ **or** $\boldsymbol{MOD^{high}\neq 0}$

$$\mathsf{OUT}^{0,low,256} = \mathsf{OUT}_{i-5}^{1,low,256}$$

vii. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} + 1 \end{cases}$$

viii. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \\ \mathsf{QUOTIENT}_{i+1}^2 = \mathsf{QUOTIENT}_i^2 \\ \mathsf{QUOTIENT}_{i+1}^3 = \mathsf{QUOTIENT}_i^3 \end{cases}$$

ix. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

x. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}^{0}_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^{1}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{2}_{i+1} = 1 \end{cases}$$

(g) If $\mathsf{STEP\_FLAG}^{\mathbf{0}}_{\boldsymbol{i}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}^{\mathbf{1}}_{\boldsymbol{i}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}^{\mathbf{2}}_{\boldsymbol{i}}\mathbf{=1}$:

i. *Register*^{0}

$$\begin{cases} \mathsf{ARG}^{0,0,low,256}_{i} = \mathsf{OUT}^{0,high,256}_{i-1} \\ \mathsf{ARG}^{0,0,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,0,low,256}_{i} = \mathsf{OUT}^{1,low,246}_{i-1} \\ \mathsf{ARG}^{1,0,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{0,256}_{i} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{0,256}_{i} = 0 \\ \mathsf{OVERFLOW\_FLAG}^{0} = 0 \end{cases}$$

ii. *Register*^{1}

$$\begin{cases} \mathsf{ARG}^{0,1,low,256}_{i} = \mathsf{OUT}^{2,low,256}_{i-1} \\ \mathsf{ARG}^{0,1,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,1,low,256}_{i} = \mathsf{OUT}^{0,low,256}_{i-1} \\ \mathsf{ARG}^{1,1,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{1,256}_{i} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{1,256}_{i} = 0 \\ \mathsf{OVERFLOW\_FLAG}^{1} = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}^{0,2,low,256}_{i} = \mathsf{OUT}^{1,high,256}_{i-1} \\ \mathsf{ARG}^{0,2,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,2,low,256}_{i} = \mathsf{OUT}^{2,high,256}_{i-1} \\ \mathsf{ARG}^{1,2,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{2,256}_{i} = 1 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{2,256}_{i} = 0 \\ \mathsf{OVERFLOW\_FLAG}^{2} = 0 \end{cases}$$

iv. *Register*^{3}

$$\begin{cases} \mathsf{ARG}^{0,3,low,256}_{i} = \mathsf{OUT}^{3,low,256}_{i-1} \\ \mathsf{ARG}^{0,3,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,3,low,256}_{i} = \mathsf{OUT}^{2,low,256}_{i} \\ \mathsf{ARG}^{1,3,high,256}_{i} = 0 \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}^{3,256}_{i} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}^{3,256}_{i} = 1 \\ \mathsf{OVERFLOW\_FLAG}^{3} = 0 \end{cases}$$

v. IF $\boldsymbol{MOD^{low}=0}$ and $\boldsymbol{MOD^{high}=0}$

$$\mathsf{OUT}^{2,low,256}_{i-4} = 0$$

vi. ELSEIF $\boldsymbol{MOD^{low}\ \not=0}$ or $\boldsymbol{MOD^{high}\ \not=0}$

$$\mathsf{OUT}^{1,low,256} = \mathsf{OUT}^{2,low,256}_{i-4}$$

vii. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square^{0,256}_{i} = \mathsf{ALU}\square^{3,256}_{i-1} + 1 \\ \mathsf{ALU}\square^{1,256}_{i} = \mathsf{ALU}\square^{0,256}_{i} + 1 \\ \mathsf{ALU}\square^{2,256}_{i} = \mathsf{ALU}\square^{1,256}_{i} + 1 \\ \mathsf{ALU}\square^{3,256}_{i} = \mathsf{ALU}\square^{2,256}_{i} + 1 \end{cases}$$

viii. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}^{low}_{i+1} = \mathsf{REM}^{low}_{i} \\ \mathsf{REM}^{high}_{i+1} = \mathsf{REM}^{high}_{i} \\ \\ MOD^{low}_{i+1} = MOD^{low}_{i} \\ MOD^{high}_{i+1} = MOD^{high}_{i} \\ \\ \mathsf{QUOTIENT}^{0}_{i+1} = \mathsf{QUOTIENT}^{0}_{i} \\ \mathsf{QUOTIENT}^{1}_{i+1} = \mathsf{QUOTIENT}^{1}_{i} \\ \mathsf{QUOTIENT}^{2}_{i+1} = \mathsf{QUOTIENT}^{2}_{i} \\ \mathsf{QUOTIENT}^{3}_{i+1} = \mathsf{QUOTIENT}^{3}_{i} \end{cases}$$

ix. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_{i}$$

x. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}^{0}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{1}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{2}_{i+1} = 1 \end{cases}$$

(h) IF $\mathsf{STEP\_FLAG}^{\boldsymbol{0}}_{\boldsymbol{i}}\boldsymbol{=1}$ AND $\mathsf{STEP\_FLAG}^{\boldsymbol{1}}_{\boldsymbol{i}}\boldsymbol{=1}$ AND $\mathsf{STEP\_FLAG}^{\boldsymbol{2}}_{\boldsymbol{i}}\boldsymbol{=1}$:

i. $Register^{0}$

$$\begin{cases} \mathsf{ARG}^{0,0,low,256}_{i} = \mathsf{OUT}^{0,high,256}_{i-1} \\ \mathsf{ARG}^{0,0,high,256}_{i} = 0 \\ \\ \mathsf{ARG}^{1,0,low,256}_{i} = \mathsf{OUT}^{1,high,256}_{i-1} \\ \mathsf{ARG}^{1,0,high,256}_{i} = 0 \\ \\ {}^{\diamondsuit}\mathsf{ADD\_FLAG}^{0,256}_{i} = 1 \\ {}^{\diamondsuit}\mathsf{MUL\_FLAG}^{0,256}_{i} = 0 \\ \mathsf{OVERFLOW\_FLAG}^{0} = 0 \end{cases}$$

ii. *Register*^{1}

$$\begin{cases} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{OUT}_{i-1}^{3,low,256} \\ \mathsf{ARG}_i^{0,1,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{OUT}_i^{0,low,256} \\ \mathsf{ARG}_i^{1,1,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{1,256} = 1 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{1,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^1 = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{OUT}_{i-1}^{2,high,256} + \mathsf{OUT}_{i-1}^{3,high,256} + \mathsf{OUT}_i^{1,high,256} \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = \mathsf{OUT}_{i-2}^{3,high,256} \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ {}^\Diamond\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\ {}^\Diamond\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. IF $\boldsymbol{MOD^{low}{=}0}$ and $\boldsymbol{MOD^{high}{=}0}$

$$\begin{cases} \mathsf{OUT}_{i-4}^{2,low,256} = 0 \\ \mathsf{OUT}_{i-3}^{1,low,256} = 0 \end{cases}$$

v. ELSEIF $\boldsymbol{MOD^{low}\not=0}$ or $\boldsymbol{MOD^{high}\not=0}$

$$\begin{cases} \mathsf{OUT}^{1,low,256} = \mathsf{OUT}_{i-4}^{2,low,256} \\ \mathsf{OUT}^{2,low,256} = \mathsf{OUT}_{i-3}^{1,low,256} \end{cases}$$

vi. Update $\mathsf{ALU}\square^{k,64}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

vii. REM, MOD and $\mathsf{QUOTIENT}$ remains constant

$$\begin{cases} \mathsf{REM}_{i+1}^{low} = \mathsf{REM}_i^{low} \\ \mathsf{REM}_{i+1}^{high} = \mathsf{REM}_i^{high} \\ \\ MOD_{i+1}^{low} = MOD_i^{low} \\ MOD_{i+1}^{high} = MOD_i^{high} \\ \\ \mathsf{QUOTIENT}_{i+1}^0 = \mathsf{QUOTIENT}_i^0 \\ \mathsf{QUOTIENT}_{i+1}^1 = \mathsf{QUOTIENT}_i^1 \\ \mathsf{QUOTIENT}_{i+1}^2 = \mathsf{QUOTIENT}_i^2 \\ \mathsf{QUOTIENT}_{i+1}^3 = \mathsf{QUOTIENT}_i^3 \end{cases}$$

viii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$$

ix. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}^0_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^1_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^2_{i+1} = 0 \end{cases}$$

11. ELSEIF $^{\diamond}\mathsf{ADDMOD\_FLAG}_{\boldsymbol{i}}\mathbf{=1}$:

(a) Same constraints as for `MULMOD`: 10a, 10b, 10c, 10d, 10e

In step 5: decompose:

$$\mathsf{ARG}^{0,low,\mathsf{DISP}} + \mathsf{ARG}_i^{1,low,\mathsf{DISP}}$$

(b) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=1}$:

i. *Register*^{0}

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,0,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{ARG}_i^{1,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = 0 \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{0,256} = 1 \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{0,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^0 = 0 \end{cases}$$

ii. *Register*^{1}

$$\begin{cases} \mathsf{ARG}_i^{0,1,low,256} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,1,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,1,low,256} = \mathsf{ARG}_i^{1,high,\mathsf{DISP}} \mathsf{ARG}_i^{1,1,high,256} = 0 \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{1,256} = 1 \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{1,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^1 = 0 \end{cases}$$

iii. *Register*^{2}

$$\begin{cases} \mathsf{ARG}_i^{0,2,low,256} = \mathsf{OUT}_i^{0,high,256} \\ \mathsf{ARG}_i^{0,2,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,2,low,256} = \mathsf{OUT}_i^{1,low,256} \\ \mathsf{ARG}_i^{1,2,high,256} = 0 \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{2,256} = 1 \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{2,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^2 = 0 \end{cases}$$

iv. *Register*^{3}

$$\begin{cases} \mathsf{ARG}_i^{0,3,low,256} = \mathsf{OUT}_i^{1,high,256} \\ \mathsf{ARG}_i^{0,3,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,3,low,256} = \mathsf{OUT}_i^{2,high,256} \\ \mathsf{ARG}_i^{1,3,high,256} = 0 \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 1 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 0 \\ \mathsf{OVERFLOW\_FLAG}^3 = 0 \end{cases}$$

v. IF $\boldsymbol{MOD^{low}{=}0}$ and $\boldsymbol{MOD^{high}{=}0}$

$$\begin{cases} \mathsf{OUT}_{i-5}^{1,low,256} = 0 \\ \mathsf{OUT}_{i-3}^{2,low,256} = 0 \\ \mathsf{OUT}_{i-2}^{2,low,256} = 0 \end{cases}$$

vi. ELSEIF $\boldsymbol{MOD^{low} \not= 0}$ or $\boldsymbol{MOD^{high} \not= 0}$

$$\begin{cases} \mathsf{OUT}_i^{0,low,256} = \mathsf{OUT}_{i-5}^{1,low,256} \\ \mathsf{OUT}_i^{2,low,256} = \mathsf{OUT}_{i-3}^{2,low,256} \\ \mathsf{OUT}_i^{3,low,256} = \mathsf{OUT}_{i-2}^{2,low,256} \end{cases}$$

vii. Update $\mathsf{ALU}\square^{k,256}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} + 1 \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} + 1 \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} + 1 \end{cases}$$

viii. The ALU DISPATCHER stamp remains constant:

$$\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1$$

ix. Set next step flags:

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

12. ELSEIF ${}^{\Diamond}\mathsf{EXP\_FLAG}_{\boldsymbol{i}}\boldsymbol{=1}$:

(a) $\mathsf{BIT\_0}$ is binary:

$$\mathsf{BIT\_0}_i * \mathsf{BIT\_0}_i = \mathsf{BIT\_0}_i$$

(b) IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}\boldsymbol{=0}$: if we start a new arithmetic operation, we have to set auxiliary variables:

i. Initialize $\mathsf{ACC\_CARRY}$

$$\begin{cases} \mathsf{ACC\_CARRY\_0}_i = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ACC\_CARRY\_1}_i = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \mathsf{STEP\_COUNTER} = 0 \end{cases}$$

ii. Compute $\mathsf{OUT}_i^{0,low,256}$ and $\mathsf{OUT}_i^{0,high,256}$

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = 1 \\ \mathsf{ARG}_i^{0,0,high,256} = 0 \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{OUT}_i^{low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{OUT}_i^{high,\mathsf{DISP}} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 1 \end{cases}$$

iii. Update $\mathsf{ALU}\square^{k,256}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

iv. $\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$

(c) ELSEIF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}} \not= \mathbf{0}$: if we continue the previous arithmetic operation

i. IF $\mathsf{ACC\_CARRY\_0}_{\boldsymbol{i}} = \mathbf{0}$ and IF $\mathsf{ACC\_CARRY\_1}_{\boldsymbol{i}} = \mathbf{0}$ computation is done:

$$\begin{cases} \mathsf{OUT}_{i-1}^{0.low,256} = 1 \\ \mathsf{OUT}_{i-1}^{0.high,256} = 0 \\ \langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i + 1 \end{cases}$$

ii. ELSEIF $\mathsf{ACC\_CARRY\_0}_{\boldsymbol{i}} \not= \mathbf{0}$ or IF $\mathsf{ACC\_CARRY\_1}_{\boldsymbol{i}} \not= \mathbf{0}$

A. $\langle \mathsf{ALU}\square \rangle_{i+1} = \langle \mathsf{ALU}\square \rangle_i$

B. IF $\mathsf{BIT\_0}_{\boldsymbol{i}} = \mathbf{0}$

$$square$$

C. ELSEIF $\mathsf{BIT\_0}_{\boldsymbol{i}} \not= \mathbf{0}$

$$squareAndMultiply$$

13. $\boldsymbol{square} =$

(a) Square the inputs

$$\begin{cases} \mathsf{ARG}_i^{0,0,low,256} = \mathsf{OUT}_{i+1}^{0,low,256} \\ \mathsf{ARG}_i^{0,0,high,256} = \mathsf{OUT}_{i+1}^{0,high,256} \\ \\ \mathsf{ARG}_i^{1,0,low,256} = \mathsf{OUT}_{i+1}^{0,low,256} \\ \mathsf{ARG}_i^{1,0,high,256} = \mathsf{OUT}_{i+1}^{0,low,256} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 1 \end{cases}$$

(b) Update $\mathsf{ALU}\square^{k,256}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

(c) $\mathsf{STEP\_COUNTER}_{i+1} = \mathsf{STEP\_COUNTER}_i + 1$

(d) IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}\mathbf{=129}$

$$\begin{cases} \mathsf{ACC\_CARRY\_0}_{i+1} = \mathsf{ACC\_CARRY\_1}_i \\ \mathsf{ACC\_CARRY\_0}_i = 0 \\ \mathsf{ACC\_CARRY\_1}_{i+1} = 0 \end{cases}$$

(e) ELSEIF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}} \mathbf{\neq 129}$

$$\mathsf{ACC\_CARRY\_0}_i = 2 * \mathsf{ACC\_CARRY\_0}_{i+1}$$

14. $\boldsymbol{squareAndMultiply} =$

(a) IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i+1}}\mathbf{=}\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}$

i. Multiply the inputs

$$\begin{cases} \mathsf{ARG}_i^{0,0,low} = \mathsf{ARG}_i^{0,low,\mathsf{DISP}} \\ \mathsf{ARG}_i^{0,0,high} = \mathsf{ARG}_i^{0,high,\mathsf{DISP}} \\ \\ \mathsf{ARG}_i^{1,0,low} = \mathsf{OUT}_i^{0,low,256} \\ \mathsf{ARG}_i^{1,0,high} = \mathsf{OUT}_i^{0,high,256} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{3,256} = 0 \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{3,256} = 1 \end{cases}$$

ii. Update $\mathsf{ALU}\square^{k,256}, k \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,256} = \mathsf{ALU}\square_{i-1}^{3,256} + 1 \\ \mathsf{ALU}\square_i^{1,256} = \mathsf{ALU}\square_i^{0,256} \\ \mathsf{ALU}\square_i^{2,256} = \mathsf{ALU}\square_i^{1,256} \\ \mathsf{ALU}\square_i^{3,256} = \mathsf{ALU}\square_i^{2,256} \end{cases}$$

iii.

$$\mathsf{ACC\_CARRY\_0}_{i+1} = \mathsf{ACC\_CARRY\_0}_i$$

(b) IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i+1}} \mathbf{\neq}\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}$

i. The same as for $\boldsymbol{square}$ 13a, 13b, 13c

ii. IF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}}\mathbf{=129}$

$$\begin{cases} \mathsf{ACC\_CARRY\_0}_{i+1} = \mathsf{ACC\_CARRY\_1}_i \\ \mathsf{ACC\_CARRY\_0}_i = 0 \\ \mathsf{ACC\_CARRY\_1}_{i+1} = 0 \end{cases}$$

iii. ELSEIF $\mathsf{STEP\_COUNTER}_{\boldsymbol{i}} \mathbf{\neq 129}$

$$\mathsf{ACC\_CARRY\_0}_i = 2 * \mathsf{ACC\_CARRY\_0}_{i+1} + 1$$

11.2 ALU 264

11.2.1 ALU256

This submodule is linked to the ALU DISPATCHER - requests for arithmetic operations are submitted by the ALU DISPATCHER to the ALU256 which, in turn, communicates with its child submodule - the ALU64, sending requests to compute 64 bit arithmetic operations. Since that the arithmetic operations of this module may involve numbers whose size exceed the maximal field size (roughly 254 bits for the bn254 curve), the ALU256 will decompose the result of the arithmetic operation in two parts: the high and the low 128 bits (in big endian decomposition, ie the high 128 bits are the most significant bits), which will be denoted by superscripts: $\cdot^{high}$ and $\cdot^{low}$. The ALU256 communicates with the ALU64 using 4 input wires: that way, every row of the ALU256 can send at most 4 different requests to the ALU64, making the computation more efficient.

To clarify a little the constraint set (in particular the way we define the requests to the ALU64), we provide here a table that details the communications between the ALU256 and the ALU64. A pair of inputs together with operation flag is called a *Register*

Step	$I_i^{0,0,64}$	$Op^{0,64}$	$I_i^{1,0,64}$	$I_i^{0,1,64}$	$Op^{1,64}$	$I_i^{1,1,64}$	$I_i^{0,2,64}$	$Op^{2,64}$	$I_i^{1,2,64}$	$I_i^{0,3,64}$	$Op^{3,64}$	$I_i^{1,3,64}$
1	A^0	$\times$	B^0	A^0	$\times$	B^1	A^1	$\times$	B^0	A^0	$\times$	B^2
2	A^1	$\times$	B^1	A^2	$\times$	B^0	A^0	$\times$	B^3	A^1	$\times$	B^2
3	A^2	$\times$	B^1	A^3	$\times$	B^0	A^1	$\times$	B^3	A^2	$\times$	B^2
4	A^3	$\times$	B^1	A^2	$\times$	B^3	A^3	$\times$	B^2	A^3	$\times$	B^3
5	R_{i-4}^1	$+$	C_{i-4}^0	R_{i-4}^2	$+$	R_i^0	C_{i-4}^1	$+$	C_{i-4}^2	R_{i-4}^3	$+$	R_{i-3}^0
6	R_{i-4}^1	$+$	R_{i-1}^2	C_{i-1}^0	$+$	C_{i-1}^1	R_{i-1}^3	$+$	R_i^0	R_i^1	$+$	R_i^2
7	C_{i-6}^3	$+$	C_{i-5}^0	R_{i-5}^2	$+$	C_{i-5}^1	R_{i-5}^3	$+$	R_{i-4}^0	R_{i-4}^1	$+$	R_i^0
8	R_{i-1}^1	$+$	R_{i-1}^2	C_{i-3}^2	$+$	C_{i-3}^3	C_{i-2}^0	$+$	$C(ADD)^0$	R_{i-1}^3	$+$	R_i^0
9	R_{i-1}^1	$+$	R_{i-1}^2	R_{i-1}^3	$+$	R_i^0						

Figure 11.1: Communication details between the ALU64 and the ALU256 for a multiplication, we assume here that $\mathsf{ARG}^{0,256} = A = 2^{192} \cdot A^3 + 2^{128} \cdot A^2 + 2^{64} A^1 + A^0$ and $\mathsf{ARG}^{1,256} = B = 2^{192} \cdot B^3 + 2^{128} \cdot B^2 + 2^{64} B^1 + B^0$. We also note $R := \mathsf{OUT}$, $C := \mathsf{CARRY\_RES}$, and $C(ADD)^0 = C_{i-2}^2 + C_{i-2}^3$. Here the final result, at the last step, is given by $\mathsf{OUT}^{high} = 2^{64} \cdot R_i^1 + R_{i-3}^3$ and $\mathsf{OUT}^{low} = 2^{64} \cdot R_{i-4}^1 + R_{i-8}^0$.

Instructions treated

- `ADD`
- `MUL`

Trace columns

ALU DISPATCHER inclusion columns:

- *Instruction*
- $\mathsf{ARG}^{i,\{high,low\},256}, i \in [0,1]$: Contains the $i^{th}, i \in [1,2]$ input of the operation.
- $\mathsf{OUT}^{\{high,low\},256}$: Contains the result of the operation to be transmitted back to the ALU DISPATCHER.
- $\mathsf{OVERFLOW\_FLAG}$: is set iff the result of the operation has overflown
- $\mathsf{ALU}\square^{256}$

ALU64 link columns:

- $\mathsf{ALU}\square^{k,64}, k \in [0,3]$
- $^{\Diamond}\mathsf{ADD\_FLAG}^{k,64}, k \in [0,3]$

- $^{\diamondsuit}\mathsf{MUL\_FLAG}^{k,64}, k \in [0,3]$
- $\mathsf{ARG}i, k, 64, i \in [0,1], k \in [0,3]$: Contains the inputs to be transmitted to the ALU64.
- $\mathsf{OUT}^{k,64}, k \in [0,3]$: Contains the 64 bits of the result of the operation.
- $\mathsf{CARRY\_RES}^{k,64}, k \in [0,3]$: Contains the bits of the carry of the result.
- $\mathsf{STEP\_FLAG}^0$, $\mathsf{STEP\_FLAG}^1$, $\mathsf{STEP\_FLAG}^2$, $\mathsf{STEP\_FLAG}^3$: Encodes the step number of operation.

Arithmetic instruction flags

- $^{\diamondsuit}\mathsf{ADD\_FLAG}^{256}$
- $^{\diamondsuit}\mathsf{MUL\_FLAG}^{256}$

Constraint set

1. $\mathsf{ALU}\square^{256}$:

$$\begin{cases} \mathsf{ALU}\square_0^{256} = 0 \\ \mathsf{ALU}\square_{i+1}^{256} \in \{\mathsf{ALU}\square_i^{256}, 1 + \mathsf{ALU}\square_i^{256}\} \end{cases}$$

2. IF $\mathsf{ALU}\square_i^{256} = 0$: then the entire i-th row is null; in particular the first row is all zeros;
3. IF $^{\diamondsuit}\mathsf{ADD\_FLAG}^{\mathbf{256}}\mathbf{=1}$: addition
   (a) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$: first step
      i. Initialize the inputs for the ALU64:

$$\begin{cases} \mathsf{ARG}_i^{j,low,256} = 2^{64} * \mathsf{ARG}_i^{j,1,64} + \mathsf{ARG}_i^{j,0,64}, j \in [0,1] \\ \mathsf{ARG}_i^{j,high,256} = 2^{64} * \mathsf{ARG}_i^{j,3,64} + \mathsf{ARG}_i^{j,2,64}, j \in [0,1] \\ \\ ^{\diamondsuit}\mathsf{ADD\_FLAG}^{k,64} = 0, \forall k \in [0,3] \\ ^{\diamondsuit}\mathsf{MUL\_FLAG}^{k,64} = 1, \forall k \in [0,3] \end{cases}$$

      ii. Set the step flag for the next operation

$$\mathsf{STEP\_FLAG}_{i+1} = 1$$

      iii. Keep the $\mathsf{ALU}\square^{256}$ constant

$$\mathsf{ALU}\square_{i+1}^{256} = \mathsf{ALU}\square_i^{256}$$

      iv. Update the $\mathsf{ALU}\square^{i,64}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,64} = \mathsf{ALU}\square_{i-1}^{3,64} + 1 \\ \mathsf{ALU}\square_i^{1,64} = \mathsf{ALU}\square_i^{0,64} + 1 \\ \mathsf{ALU}\square_i^{2,64} = \mathsf{ALU}\square_i^{1,64} + 1 \\ \mathsf{ALU}\square_i^{3,64} = \mathsf{ALU}\square_i^{2,64} + 1 \end{cases}$$

   (b) ELSEIF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$: second step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{OUT}_{i-1}^{1,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{CARRY\_RES}_{i-1}^{0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0 \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{OUT}_{i-1}^{2,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{CARRY\_RES}_{i-1}^{1,64} + \mathsf{CARRY\_RES}_i^{0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0 \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{OUT}_{i-1}^{3,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{CARRY\_RES}_{i-1}^{2,64} + \mathsf{CARRY\_RES}_i^{1,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 1 \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 0 \end{cases}$$

iv. Set the result values

$$\begin{cases} \mathsf{OUT}_i^{high,256} = 2^{64} * \mathsf{OUT}_i^{2,64} + Res_i^{1,64} \\ \mathsf{OUT}_i^{low,256} = 2^{64} * \mathsf{OUT}_i^{0,64} + Res_{i-1}^{0,64} \end{cases}$$

v. IF $\mathsf{CARRY\_RES}_{\boldsymbol{i}}^{\mathbf{2,64}}+\mathsf{CARRY\_RES}_{\boldsymbol{i-1}}^{\mathbf{3,64}}\mathbf{=0}$: Do not set the overflow flag

$$\mathsf{OVERFLOW\_FLAG}_i = 0$$

vi. ELSEIF $\mathsf{CARRY\_RES}_{\boldsymbol{i}}^{\mathbf{2,64}}+\mathsf{CARRY\_RES}_{\boldsymbol{i-1}}^{\mathbf{3,64}} \not= \mathbf{0}$: Set the overflow flag

$$\mathsf{OVERFLOW\_FLAG}_i = 1$$

vii. Unset the step flags for the next operation

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^0 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^1 = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^2 = 0 \end{cases}$$

viii. Increase the $\mathsf{ALU}\square^{256}$

$$\mathsf{ALU}\square_{i+1}^{256} = \mathsf{ALU}\square_i^{256} + 1$$

ix. Update the $\mathsf{ALU}\square^{i,64}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,64} = \mathsf{ALU}\square_{i-1}^{3,64} + 1 \\ \mathsf{ALU}\square_i^{1,64} = \mathsf{ALU}\square_i^{0,64} + 1 \\ \mathsf{ALU}\square_i^{2,64} = \mathsf{ALU}\square_i^{1,64} + 1 \\ \mathsf{ALU}\square_i^{3,64} = \mathsf{ALU}\square_i^{2,64} \end{cases}$$

4. IF ${}^{\Diamond}\mathsf{MUL\_FLAG}_{\boldsymbol{i}}^{\mathbf{256}}\mathbf{=1}$: multiplication

   (a) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: first step - compute the extreme terms

i. Input values for the ALU64

$$\begin{cases} \mathsf{ARG}_i^{j,low,256} = 2^{64} * \mathsf{ARG}_i^{j,1,64} + \mathsf{ARG}_i^{j,0,64}, j \in [0,1] \\ \mathsf{ARG}_i^{j,high,256} = 2^{64} * \mathsf{ARG}_i^{j,3,64} + \mathsf{ARG}_i^{j,2,64}, j \in [0,1] \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{k,64} = 0, \forall k \in [0,3] \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{k,64} = 1, \forall k \in [0,3] \end{cases}$$

ii. Update the Update the $\mathsf{ALU}\square^{i,64}, i \in [0,3]$

$$\begin{cases} \mathsf{ALU}\square_i^{0,64} = \mathsf{ALU}\square_{i-1}^{3,64} + 1 \\ \mathsf{ALU}\square_i^{1,64} = \mathsf{ALU}\square_i^{0,64} + 1 \\ \mathsf{ALU}\square_i^{2,64} = \mathsf{ALU}\square_i^{1,64} + 1 \\ \mathsf{ALU}\square_i^{3,64} = \mathsf{ALU}\square_i^{2,64} + 1 \end{cases}$$

iii. Keep the $\mathsf{ALU}\square^{256}$ constant

$$\mathsf{ALU}\square_{i+1}^{256} = \mathsf{ALU}\square_i^{256}$$

iv. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 0 \end{cases}$$

(b) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$ - second step

i. $Register0$:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{ARG}_{i-1}^{0,0,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{ARG}_{i-1}^{1,1,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 1, \end{cases}$$

ii. $Register1$:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{ARG}_{i-1}^{0,0,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{ARG}_{i-1}^{1,1,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 1, \end{cases}$$

iii. $Register2$:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{ARG}_{i-1}^{0,1,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{ARG}_{i-1}^{1,0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 1, \end{cases}$$

iv. $Register0$:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{ARG}_{i-1}^{0,0,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{ARG}_{i-1}^{1,2,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 1, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}^{0}_{i+1} = 0 \\ \mathsf{STEP\_FLAG}^{1}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{2}_{i+1} = 0 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(c) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$- third step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}^{0,0,64}_{i} = \mathsf{ARG}^{0,1,64}_{i-2} \\ \mathsf{ARG}^{1,0,64}_{i} = \mathsf{ARG}^{1,1,64}_{i-2} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{0,64}_{i} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{0,64}_{i} = 1, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}^{0,1,64}_{i} = \mathsf{ARG}^{0,2,64}_{i-2} \\ \mathsf{ARG}^{1,1,64}_{i} = \mathsf{ARG}^{1,0,64}_{i-2} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{1,64}_{i} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{1,64}_{i} = 1, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}^{0,2,64}_{i} = \mathsf{ARG}^{0,0,64}_{i-2} \\ \mathsf{ARG}^{1,2,64}_{i} = \mathsf{ARG}^{1,3,64}_{i-2} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{2,64}_{i} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{2,64}_{i} = 1, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}^{0,3,64}_{i} = \mathsf{ARG}^{0,1,64}_{i-2} \\ \mathsf{ARG}^{1,3,64}_{i} = \mathsf{ARG}^{1,2,64}_{i-2} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{3,64}_{i} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{3,64}_{i} = 1, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}^{0}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{1}_{i+1} = 1 \\ \mathsf{STEP\_FLAG}^{2}_{i+1} = 0 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(d) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: fourth step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}^{0,0,64}_{i} = \mathsf{ARG}^{0,2,64}_{i-3} \\ \mathsf{ARG}^{1,0,64}_{i} = \mathsf{ARG}^{1,1,64}_{i-3} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}^{0,64}_{i} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}^{0,64}_{i} = 1, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{ARG}_{i-3}^{0,3,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{ARG}_{i-3}^{1,0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 1, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{ARG}_{i-3}^{0,1,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{ARG}_{i-3}^{1,3,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 1, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{ARG}_{i-3}^{0,2,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{ARG}_{i-3}^{1,2,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 1, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 1 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(e) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: fifth step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{ARG}_{i-4}^{0,3,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{ARG}_{i-4}^{1,1,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 1, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{ARG}_{i-4}^{0,2,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{ARG}_{i-4}^{1,3,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 1, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{ARG}_{i-4}^{0,3,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{ARG}_{i-4}^{1,2,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 1, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{ARG}_{i-4}^{0,3,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{ARG}_{i-4}^{1,3,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 0, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 1, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 1 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(f) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: sixth step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}0,0,64_i = \mathsf{OUT}_{i-4}^{1,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{OUT}_{i-4}^{0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{OUT}_{i-4}^{2,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{OUT}_{i}^{0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{CARRY\_RES}_{i-4}^{1,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{CARRY\_RES}_{i-4}^{2,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 1, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 0, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{OUT}_{i-4}^{3,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{OUT}_{i3}^{0,64} \\ \\ {}^{\Diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 1, \\ {}^{\Diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 0, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 1 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(g) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: seventh step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{OUT}_{i-4}^{1,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{OUT}_{i-1}^{2,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{CARRY\_RES}_{i-1}^{0,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{CARRY\_RES}_{i-1}^{1,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{OUT}_{i-1}^{3,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{OUT}_i^{0,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 0, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{OUT}_i^{1,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{OUT}_i^{2,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 0, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 1 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(h) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=0}$: eight step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{CARRY\_RES}_{i-6}^{3,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{CARRY\_RES}_{i-5}^{0,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{OUT}_{i-5}^{2,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{CARRY\_RES}_{i-5}^{1,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{OUT}_{i-5}^{3,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{OUT}_{i-4}^{0,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 0, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{OUT}_{i-4}^{1,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{OUT}_{i}^{0,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 0, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 0 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(i) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=1}$: ninth step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{OUT}_{i-1}^{1,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{OUT}_{i-1}^{2,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{CARRY\_RES}_{i-3}^{2,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{CARRY\_RES}_{i-3}^{3,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0, \end{cases}$$

iii. *Register*^{2}:

$$\begin{cases} \mathsf{ARG}_i^{0,2,64} = \mathsf{CARRY\_RES}_{i-2}^{0,64} \\ \mathsf{ARG}_i^{1,2,64} = \mathsf{CARRY\_RES}_{i-2}^{2,64} + \mathsf{CARRY\_RES}_{i-2}^{3,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{2,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{2,64} = 0, \end{cases}$$

iv. *Register*^{3}:

$$\begin{cases} \mathsf{ARG}_i^{0,3,64} = \mathsf{OUT}_{i-1}^{3,64} \\ \mathsf{ARG}_i^{1,3,64} = \mathsf{OUT}_{i}^{0,64} \\ \\ {}^{\diamond}\mathsf{ADD\_FLAG}_i^{3,64} = 1, \\ {}^{\diamond}\mathsf{MUL\_FLAG}_i^{3,64} = 0, \end{cases}$$

v. Set the next step flags

$$\begin{cases} \mathsf{STEP\_FLAG}_{i+1}^{0} = 1 \\ \mathsf{STEP\_FLAG}_{i+1}^{1} = 0 \\ \mathsf{STEP\_FLAG}_{i+1}^{2} = 0 \end{cases}$$

vi. Same constraints as 3(a)iv and 3(a)iii

(j) IF $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{0}}\mathbf{=1}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{1}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{2}}\mathbf{=0}$ AND $\mathsf{STEP\_FLAG}_{\boldsymbol{i}}^{\mathbf{3}}\mathbf{=1}$: tenth step

i. *Register*^{0}:

$$\begin{cases} \mathsf{ARG}_i^{0,0,64} = \mathsf{OUT}_{i-1}^{1,64} \\ \mathsf{ARG}_i^{1,0,64} = \mathsf{OUT}_{i-1}^{2,64} \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{0,64} = 1, \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{0,64} = 0, \end{cases}$$

ii. *Register*^{1}:

$$\begin{cases} \mathsf{ARG}_i^{0,1,64} = \mathsf{OUT}_{i-1}^{3,64} \\ \mathsf{ARG}_i^{1,1,64} = \mathsf{OUT}_{i}^{0,64} \\ \\ ^{\diamond}\mathsf{ADD\_FLAG}_i^{1,64} = 1, \\ ^{\diamond}\mathsf{MUL\_FLAG}_i^{1,64} = 0, \end{cases}$$

iii. Set the new stamps for the ALU64:

$$\begin{cases} \mathsf{ALU}\square_i^{64,0} = \mathsf{ALU}\square_{i-1}^{64,3} + 1 \\ \mathsf{ALU}\square_i^{64,1} = \mathsf{ALU}\square_{i}^{64,0} + 1 \\ \mathsf{ALU}\square_i^{64,2} = \mathsf{ALU}\square_{i}^{64,0} \\ \mathsf{ALU}\square_i^{64,3} = \mathsf{ALU}\square_{i}^{64,0} \end{cases}$$

iv. Set the result values:

$$\begin{cases} \mathsf{OUT}_i^{low,256} = 2^{64} * \mathsf{OUT}_{i-4}^{1,64} + \mathsf{OUT}_{i-8}^{0,64} \\ \mathsf{OUT}_i^{high,256} = 2^{64} * \mathsf{OUT}_{i}^{1,64} + \mathsf{OUT}_{i-3}^{3,64} \end{cases}$$

v. Check if the result has overflown.

$$
\begin{array}{l}
\mathsf{OVERFLOW\_SUM} = \\
\mathsf{CARRY\_RES}^{2,64}_{i-7}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{3,64}_{i-7}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{2,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{3,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{0,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{1,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{2,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{3,64}_{i-6}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{0,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{1,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{2,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{OUT}^{3,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{0,64}_{i-5}.IsNonzeroBinary()+ \ . \\
\mathsf{CARRY\_RES}^{1,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{2,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{3,64}_{i-5}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{0,64}_{i-2}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{1,64}_{i-2}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{2,64}_{i-2}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{3,64}_{i-2}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{0,64}_{i-1}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{1,64}_{i-1}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{2,64}_{i-1}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{3,64}_{i-1}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{0,64}_{i}.IsNonzeroBinary()+ \\
\mathsf{CARRY\_RES}^{1,64}_{i}.IsNonzeroBinary()+
\end{array}
$$

vi. IF OVERFLOW_SUM=**0**

$$\mathsf{OVERFLOW\_FLAG}_i = 0$$

vii. ELSEIF OVERFLOW_SUM $\neq$**0**

$$\mathsf{OVERFLOW\_FLAG}_i = 1$$

viii. Set the next step flags

$$
\begin{cases}
\mathsf{STEP\_FLAG}^0_{i+1} = 0 \\
\mathsf{STEP\_FLAG}^1_{i+1} = 0 \\
\mathsf{STEP\_FLAG}^2_{i+1} = 0
\end{cases}
$$

11.3 ALU 64

11.3.1 ALU64

This submodule treats the operations transmitted by the ALU256 (`MUL` and `ADD` operations), computes the associated result and carry, and performs range checks to ensure safe 64-bit arithmetic.

Trace columns

- $^\diamond$ADD_FLAG
- $^\diamond$MUL_FLAG

- $\mathsf{ALU}\square^{64}$
- $Input^i, i \in \{0,1\}$
- OUT
- $\mathsf{CARRY\_RES}$

Constraint set

- $\mathsf{ALU}\square^{64}$:

$$\begin{cases} \mathsf{ALU}\square^{64}_0 = 0 \\ \mathsf{ALU}\square^{64}_{i+1} \in \{\mathsf{ALU}\square^{64}_i, 1 + \mathsf{ALU}\square^{64}_i\} \end{cases}$$

- IF $\mathsf{ALU}\square^{64}_i = 0$: then the entire i-th row is null; in particular the first row is all zeros;
- IF $^{\diamondsuit}\mathsf{ADD\_FLAG}_i = 1$: addition
  - Compute the result and the carry:

$$\mathsf{ARG}^0_i + \mathsf{ARG}^1_i = \mathsf{OUT}_i + 2^{64} \cdot \mathsf{CARRY\_RES}_i$$

- ELSEIF $^{\diamondsuit}\mathsf{MUL\_FLAG}_i = 1$: multiplication
  - Compute the result and the carry:

$$\mathsf{ARG}^0_i \cdot \mathsf{ARG}^1_i = \mathsf{OUT}_i + 2^{64} \cdot \mathsf{CARRY\_RES}_i$$

Range constraints

$\mathsf{ARG}^i, i \in \{0,1\}$, OUT and $\mathsf{CARRY\_RES}$ should all belong to the range $[0, 2^{64}[$, ie they should be composed of at most 8 bytes or 4 double-bytes.

Chapter 12

EXP dynamic gas

12.1 Exponent module

12.1.1 Introduction

The **exponent byte size module** is a tiny module which carries out a computation required to compute the dynamic gas cost of `EXP` instructions. It doesn't carry out `EXP` instructions, that is the perview of the ALU module, rather it computes the size in bytes of the exponent which is required for establishing the dynamic gas cost of `EXP`.

12.1.2 Columns

1. $\langle \mathsf{EXP}\square \rangle$: imported column containing the exponentiation time stamp;

The hub contains an $\mathsf{EXP}\square$ column: it's a simple stamp colum that whose value increases by 1 every time the hub encounters an `EXP` instruction.

2. $\langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{EXPNT}^{\mathsf{lo}} \rangle$: imported columns containing the high and low parts of the exponent;
3. $\langle \mathsf{SIZE} \rangle$: imported column containing the size in bytes of the exponent;

Given the stack pattern of the `EXP` instruction, $\langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle$ and $\langle \mathsf{EXPNT}^{\mathsf{lo}} \rangle$ columns are imports of the (high and low part of the) third stack item's value ${}_3\mathsf{VAL}^{\mathsf{hi}}$ and ${}_3\mathsf{VAL}^{\mathsf{lo}}$. The imported $\langle \mathsf{SIZE} \rangle$ column is justified in the present module. It will be made to contain the size in bytes size of the exponent. Recall the convention for the size in bytes for an (EVM word) exponent $\mathsf{e} \equiv (\mathsf{e}^{\mathsf{hi}}, \mathsf{e}^{\mathsf{lo}})$:

$$\begin{cases} \text{IF } \mathsf{e}^{\mathsf{hi}} \neq 0 & \text{THEN } \mathsf{size} = 1 + \lfloor \log_{256}(\mathsf{e}^{\mathsf{hi}}) \rfloor + 16 \\ \text{IF } \left(\mathsf{e}^{\mathsf{hi}} = 0 \ \text{ AND } \ \mathsf{e}^{\mathsf{lo}} \neq 0\right) & \text{THEN } \mathsf{size} = 1 + \lfloor \log_{256}(\mathsf{e}^{\mathsf{lo}}) \rfloor \\ \text{IF } \left(\mathsf{e}^{\mathsf{hi}} = 0 \ \text{ AND } \ \mathsf{e}^{\mathsf{lo}} = 0\right) & \text{THEN } \mathsf{size} = 0 \end{cases}$$

4. DO_BYTE_DECOMPOSITION: binary column; equals 0 if and only if the exponent is zero; abbreviate to DOBD;
5. BYTE_1: byte column;
6. ACC_1: "accumulator" column; accumulates the bytes from BYTE_1;
7. PLATEAU_BIT: binary "pivot bit" column; abbreviated to PBIT;
8. COUNTER: counter colum; either hovers around zero or counts from 0 to **15**; abbreviated to CT;

12.2 General constraints

12.2.1 The DOBD flag

We set the DOBD flag: $\mathsf{DOBD} = 1 \iff$ the exponent is nonzero, i.e.:

$$\begin{cases} \text{IF } \big(\langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle_i = 0 \;\; \text{AND } \langle \mathsf{EXPNT}^{\mathsf{lo}} \rangle_i = 0\big) \text{ THEN } \mathsf{DOBD}_i = 0 \\ \text{IF } \langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle_i \neq 0 \text{ THEN } \mathsf{DOBD}_i = 1 \\ \text{IF } \langle \mathsf{EXPNT}^{\mathsf{lo}} \rangle_i \neq 0 \text{ THEN } \mathsf{DOBD}_i = 1 \end{cases}$$

Note that the hearbeat imposes in particular that if $\langle \mathsf{EXP}\square \rangle_i = 0$ the whole row is zero, in particular the exponent is zero.

12.2.2 Heartbeat

The hearbeat of the present module is simple: every call to it occupies **16** lines except if the exponent is 0 i.e. if $\mathsf{DOBD}_i = 0$.

1. $\langle \mathsf{EXP}\square \rangle_0 = 0$;
2. $\langle \mathsf{EXP}\square \rangle$ is nondecreasing in the sense that $\langle \mathsf{EXP}\square \rangle_{i+1} \in \{\langle \mathsf{EXP}\square \rangle_i, 1 + \langle \mathsf{EXP}\square \rangle_i\}$;
3. $\text{IF } \langle \mathsf{EXP}\square \rangle_i = 0 \text{ THEN } (\mathsf{DOBD}_i = 0 \;\; \text{AND } \mathsf{CT}_i = 0)$;
4. $\text{IF } \langle \mathsf{EXP}\square \rangle_{i+1} \neq \langle \mathsf{EXP}\square \rangle_i \text{ THEN } \mathsf{CT}_{i+1} = 0$;
5. $\text{IF } \langle \mathsf{EXP}\square \rangle_i \neq 0 \text{ THEN}$
   (a) $\text{IF } \mathsf{DOBD}_i = 0 \text{ THEN } \langle \mathsf{EXP}\square \rangle_{i+1} = 1 + \langle \mathsf{EXP}\square \rangle_i$
   (b) $\text{IF } \mathsf{DOBD}_i = 1 \text{ THEN}$
      i. $\text{IF } \mathsf{CT}_i \neq \mathbf{15} \text{ THEN}$
      $$\begin{cases} \mathsf{CT}_{i+1} & = & 1 + \mathsf{CT}_i \\ \mathsf{DOBD}_{i+1} & = & 1 \end{cases}$$
      ii. $\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN } \langle \mathsf{EXP}\square \rangle_{i+1} = 1 + \langle \mathsf{EXP}\square \rangle_i$
6. $\text{IF } \mathsf{DOBD}_N = 1 \text{ THEN } \mathsf{CT}_N = \mathbf{15}$.

12.2.3 Byte decomposition

We impose byte decompositions:

1. $\text{IF } \mathsf{DOBD}_i = 0 \text{ THEN } \mathsf{BYTE\_1}_i = 0$;
2. $\text{IF } \mathsf{CT}_i = 0 \text{ THEN } \mathsf{ACC\_1}_i = \mathsf{BYTE\_1}_i$;
3. $\text{IF } \mathsf{CT}_i \neq 0 \text{ THEN } \mathsf{ACC\_1}_i = 256 \cdot \mathsf{ACC\_1}_{i-1} + \mathsf{BYTE\_1}_i$;

We further impose that $\mathsf{BYTE\_1}$ contain bytes.

12.2.4 Target constraints

We fix the target of the accumulator column:

1. $\text{IF } \mathsf{CT}_i = \mathbf{15} \text{ THEN}$

$$\begin{cases} \text{IF } \langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle_i \neq 0 \text{ THEN } \mathsf{ACC\_1}_i = \langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle_i \\ \text{IF } \langle \mathsf{EXPNT}^{\mathsf{hi}} \rangle_i = 0 \text{ THEN } \mathsf{ACC\_1}_i = \langle \mathsf{EXPNT}^{\mathsf{lo}} \rangle_i \end{cases}$$

Note that $\mathsf{CT}_i = \mathbf{15}$ can only happen if $\mathsf{DOBD}_i = 1$ (and thus $\langle \mathsf{EXP}\square \rangle_i \neq 0$.)

12.2.5 PLATEAU_BIT constraints

The plateau bit PBIT is a binary colum. It only plays a role if the exponent is nonzero. Its purpose is to switch from 0 to 1 at the precise moment the trace encounters the leading byte of the exponent. Here are its constraints:

1. PBIT is a binary column i.e. $\mathsf{PBIT}_i \cdot (1 - \mathsf{PBIT}_i) = 0$;
2. IF $\mathsf{DOBD}_i = 0$ THEN $\mathsf{PBIT}_i = 0$;
3. IF $\mathsf{DOBD}_i = 1$ THEN
   (a) IF $\mathsf{CT}_i \neq \mathbf{15}$ THEN $\mathsf{PBIT}_{i+1} \in \{\mathsf{PBIT}_i, 1 + \mathsf{PBIT}_i\}$ i.e. PBIT_i is nondecreasing within a counter cycle;
   (b) IF $\mathsf{CT}_i = 0$ THEN
   $$\begin{cases} \text{IF } \mathsf{BYTE\_1}_i = 0 \text{ THEN } \mathsf{PBIT}_i = 0 \\ \text{IF } \mathsf{BYTE\_1}_i \neq 0 \text{ THEN } \mathsf{PBIT}_i = 1 \end{cases}$$
   (c) IF $\mathsf{CT}_i \neq \mathbf{15}$ AND $\mathsf{PBIT}_i = 0$ THEN
   $$\begin{cases} \text{IF } \mathsf{BYTE\_1}_{i+1} = 0 \text{ THEN } \mathsf{PBIT}_{i+1} = 0 \\ \text{IF } \mathsf{BYTE\_1}_{i+1} \neq 0 \text{ THEN } \mathsf{PBIT}_{i+1} = 1 \end{cases}$$

12.2.6 ⟨SIZE⟩ constraints

We constrain the ⟨SIZE⟩ column.

1. IF $\mathsf{DOBD}_i = 0$ THEN $\langle\mathsf{SIZE}\rangle_i = 0$
2. IF $\mathsf{DOBD}_i = 1$ THEN
   (a) IF $\mathsf{CT}_i = 0$ AND $\mathsf{PBIT}_i = 1$ THEN
   $$\begin{cases} \text{IF } \langle\mathsf{EXPNT}^{\mathsf{hi}}\rangle_i \neq 0 \text{ THEN } \langle\mathsf{SIZE}\rangle_i = 32 - \mathsf{CT}_i \\ \text{IF } \langle\mathsf{EXPNT}^{\mathsf{hi}}\rangle_i = 0 \text{ THEN } \langle\mathsf{SIZE}\rangle_i = 16 - \mathsf{CT}_i \end{cases}$$
   Note: by hypothesis $\mathsf{CT}_i = 0$ so we may just as well write "$\langle\mathsf{SIZE}\rangle_i = 32$" or "$\langle\mathsf{SIZE}\rangle_i = 16$" depending on whether $\langle\mathsf{EXPNT}^{\mathsf{hi}}\rangle_i \neq 0$ or not;
   (b) IF $\left(\mathsf{CT}_i \neq \mathbf{15} \text{ AND } \mathsf{PBIT}_i = 0 \text{ AND } \mathsf{PBIT}_{i+1} = 1\right)$ THEN
   $$\begin{cases} \text{IF } \langle\mathsf{EXPNT}^{\mathsf{hi}}\rangle_i \neq 0 \text{ THEN } \langle\mathsf{SIZE}\rangle_i = 32 - \mathsf{CT}_{i+1} \\ \text{IF } \langle\mathsf{EXPNT}^{\mathsf{hi}}\rangle_i = 0 \text{ THEN } \langle\mathsf{SIZE}\rangle_i = 16 - \mathsf{CT}_{i+1} \end{cases}$$

Chapter 13

Address Shaving

13.1 Address shaving module

13.1.1 Introduction

The **address shaving module** is a tiny and very simple module whose sole purpose is to reduce mod 2^{160} stack arguments that ought to be interpreted as addresses. It is triggered by the following instructions:

1. `BALANCE`
2. `EXTCODESIZE`
3. `EXTCODECOPY`
4. `EXTCODEHASH`
5. `CALL`
6. `CALLCODE`
7. `STATICCALL`
8. `DELEGATECALL`
9. `SELFDESTRUCT`

These are precisely the instructions that raise the $\mathsf{ADDRESS\_SHAVING\_FLAG}$ in the hub.

13.1.2 Columns

1. $\langle\mathsf{SHAVE}\square\rangle$: imported stamp column;

The address shaving module is activated every time an instruction which requires address shaving is loaded into the hub. Every such call increases the $\mathsf{ADDRESS\_SHAVING\_STAMP}$ (i.e. $\mathsf{SHAVE}\square$) in the hub by 1.

2. $\langle\mathsf{ADDR}^{\mathsf{hi}}\rangle$: imported column containing the high part of the appropriate stack argument containing the address argument of the instruction at hand;
3. $\langle\mathsf{LOW4}\rangle$: imported column containing the shaved version of the high part of the address arguement;
4. CT: counter column: counts continuously from 0 to 15 and resets;
5. $\mathsf{BYTE\_1}$: byte column;
6. $\mathsf{ACC\_1}$ and $\mathsf{ACC\_2}$: accumulator column; accumulates the bytes from $\mathsf{BYTE\_1}$;
7. PBIT: binary colum that switches from 0 to 1 when $\mathsf{CT}_i = 12$

13.2 Constraints

13.2.1 Heartbeat

The heartbeat of the address shaving module is very simple: the CT column counts from 0 to $\mathbf{15}$ unless the $\langle \mathsf{SHAVE}\square \rangle$ is zero, in which case it hovers at 0.

1. $\langle \mathsf{SHAVE}\square \rangle_0 = 0$
2. $\langle \mathsf{SHAVE}\square \rangle$ is nondecreasing in the sense that $\langle \mathsf{SHAVE}\square \rangle_{i+1} \in \{\langle \mathsf{SHAVE}\square \rangle_i, 1 + \langle \mathsf{SHAVE}\square \rangle_i\}$
3. IF $\langle \mathsf{SHAVE}\square \rangle_i = 0$ THEN $\mathsf{CT}_i = 0$
4. IF $\langle \mathsf{SHAVE}\square \rangle_{i+1} \neq \langle \mathsf{SHAVE}\square \rangle_i$ THEN $\mathsf{CT}_{i+1} = 0$
5. IF $\langle \mathsf{SHAVE}\square \rangle_i \neq 0$ THEN
   (a) IF $\mathsf{CT}_i \neq \mathbf{15}$ THEN $\mathsf{CT}_{i+1} = 1 + \mathsf{CT}_i$
   (b) IF $\mathsf{CT}_i = \mathbf{15}$ THEN $\langle \mathsf{SHAVE}\square \rangle_{i+1} = 1 + \langle \mathsf{SHAVE}\square \rangle_i$
6. IF $\langle \mathsf{SHAVE}\square \rangle_N \neq 0$ THEN $\mathsf{CT}_N = \mathbf{15}$

13.2.2 PBIT contraints

The PBIT column is a binary colum that hovers around zero until CT reaches the value 12 at which point it switches to 1. The associated constraints are as follows:

1. PBIT is binary;
2. IF $\mathsf{CT}_i = 0$ THEN $\mathsf{PBIT}_i = 0$;
3. IF $\mathsf{CT}_i \neq 0$ THEN $\mathsf{PBIT}_i \in \{\mathsf{PBIT}_{i-1}, 1 + \mathsf{PBIT}_{i-1}\}$;
4. IF $\mathsf{CT}_i = 12$ THEN $\big(\mathsf{PBIT}_{i-1} = 0$ AND $\mathsf{PBIT}_i = 1\big)$;

13.2.3 Byte decomposition

We impose the following byte decomposition:

1. IF $\langle \mathsf{SHAVE}\square \rangle_i = 0$ THEN $\mathsf{BYTE\_1}_i = 0$;
2. IF $\mathsf{CT}_i = 0$ THEN
$$\begin{cases} \mathsf{ACC\_1}_i = \mathsf{BYTE\_1}_i \\ \mathsf{ACC\_2}_i = 0 \end{cases}$$
;
3. IF $\mathsf{CT}_i \neq 0$ THEN $\mathsf{ACC\_1}_i = 256 \cdot \mathsf{ACC\_1}_{i-1} + \mathsf{BYTE\_1}_i$
$$\begin{cases} \mathsf{ACC\_1}_i = 256 \cdot \mathsf{ACC\_1}_{i-1} + \mathsf{BYTE\_1}_i \\ \mathsf{ACC\_2}_i = 256 \cdot \mathsf{ACC\_1}_{i-1} + \mathsf{PBIT}_i \cdot \mathsf{BYTE\_1}_i \end{cases}$$
;

We further impose that $\mathsf{BYTE\_1}$ contain bytes.

13.2.4 Target constraints

We fix the target of the accumulator column:

1. IF $\mathsf{CT}_i = \mathbf{15}$ THEN
$$\begin{cases} \langle \mathsf{ADDR}^{\mathsf{hi}} \rangle_i & = & \mathsf{ACC\_1}_i \\ \langle \mathsf{LOW4} \rangle_i & = & \mathsf{ACC\_2}_i \end{cases}$$

Bibliography

[1] Vitalik Buterin. *An Incomplete Guide to Rollups*. 2021. URL: https://vitalik.ca/general/2021/01/05/rollup.html.

[2] DeGate Team. *An article to understand zkEVM, the key to Ethereum scaling*. 2021. URL: https://medium.com/degate/an-article-to-understand-zkevm-the-key-to-ethereum-scaling-ff0d83c417cc.

[3] *ZK-sync official website*. URL: https://zksync.io/.

[4] Lior Goldberg, Shahar Papini, and Michael Riabzev. *Cairo – a Turing-complete STARK-friendly CPU architecture*. Cryptology ePrint Archive, Report 2021/1063. https://ia.cr/2021/1063. 2021.

[5] *Hermez official website*. URL: https://hermez.io/.

[6] *Scroll tech github repository*. URL: https://github.com/scroll-tech/.

[7] DR. Gavin Wood. "Ethereum : A secure decentralised generalised transaction ledger". In: (2022).