ConsenSys zk-EVM 说明书(译)

原文: **ConsenSys, Applied R&D** 团队发布的zk-EVM说明书

翻译: by Ola

注：本译文稍微丰富了小部分概念的描述以便读者容易理解，如virtual column、permutation argument等

本书描述的zk-EVM算术化阶段支持三个设计目标：(1) 支持包括internal smart contract call、error management和gas management在内的所有的EVM opcodes；(2) 执行bytecodes；(3)最小的prover time。 我们尽力把所有opcodes的算术化过程和Ethereum黄皮书[1]上的说明一致。我们也针对现有的zkp方案所面临的技术性实现问题提供了一个原始、全面的解决办法。

1. 介绍

1.1 行业现状

在目前的区块链生态里，增大区块容量以减少每笔交易的平均消耗是一个巨大的挑战。

Rollups是一种很有前途的技术，它将大大提高以太坊区块链的容量。在[2,3]中分别介绍了Rollups、zk-EVMs以及它们在提高以太坊容量方面的作用。最近，诸多知名机构参与构建可扩展的和实用型的Rollups方案；例如，zkSync[4]将Yul转换成zk-VM友好的字节码。另一方面，Cairo[5]使用一种custom结构，该体系结构适应于使用Cairo编写的智能合约的高效STARK证明程序。

其他项目，如Hermez[6]或Scroll Tech[7]的目标是直接解释EVM字节码，而不需要任何中间的编译器或额外的编译步骤。这也是我们所采取的方法。

1.2 与本项目的关系

受Cairo[5]的现有设计的启发，我们借用了虚拟列（virtual column）、执行轨迹（execute trace）、范围证明（range proof）和内存完整性轨迹排列技巧（memory integrity trace permutation trick） 以及可验证只读内存（provable read-only-memory）的概念。我们将这些概念扩展和调整到EVM体系结构的不同部分。最值得注意的是，我们能够将EVM堆栈调整为只读架构，从而大大减少了执行轨迹的大小：我们只需要6个指针来表示stack，而EVM stack的简单实现至少需要1024个指针。

我们的zk-EVM体系结构包含不同的模块，每个模块的任务都是证明智能合约（smart contract）某特定部分的执行。这里我们借鉴了Hermez[8]的方法。

1.3 论文结构

论文的剩余部分由以下几点组成：

a. zk-EVM架构：描述我们zk-EVM架构相关的主要组件；

b. 工具和概念：介绍我们zk-EVM架构用到的基础工具和方法；

c. 约束集：给出我们zk-EVM部分组件的底层约束系统；

为了提高本文的可读性，我们选择仅为几个(有代表性的)模块提供完整的约束系统，我们努力尽可能全面地描述这些模块。其他模块，如主执行轨迹和存储模块，已经完全设计好了——但是，考虑到这些组件的复杂性，我们选择略微推迟它们的发布。如果有兴趣，您可以直接联系我们以获得更多关于这些模块的信息。

考虑到我们计算的复杂性，错误是在所难免的。

2. zk-EVM架构

我们的zk-EVM模块(module)由很多特定功能的子模块(submodule)组成：

- ROM(Read-Only-Memory)：包含要执行的合约字节码；
- 主执行轨迹(Main execution trace)：指令的执行轨迹，并生成一个stack 内存；
- 一些特定功能的小模块，针对RAM、storage、binary和arithmetic opcodes；
- 指令解码器(Ins-decoder)：把EVM opcodes解析成一系列的指令标志（译者注：就是不会被改变的public data）；

ROM的主要目的是把合约的字节码转换成可以被zk-EVM执行的连续指令（可以被设想成一个区块内被调用的合约的所有字节码）。

每个模块只对应EVM所有指令集的一个特定子集。 模块使用指令解码器提供的flag分解的特定flag来选择与它们有关的指令。这些flag和相关参数的正确性由指令解码器的包含证明来保证——一*段不可改的public data*(译者注：即校验子模块上使能的flag和指令解析器上存储的flag的包含关系)。每个模块都有自己的内部执行轨迹和对应的约束系统(constraint system)。每个模块和和整个模块间的关系也由总线机制（一般是plookup包含证明）来保证。

该组织如图 1 所示：

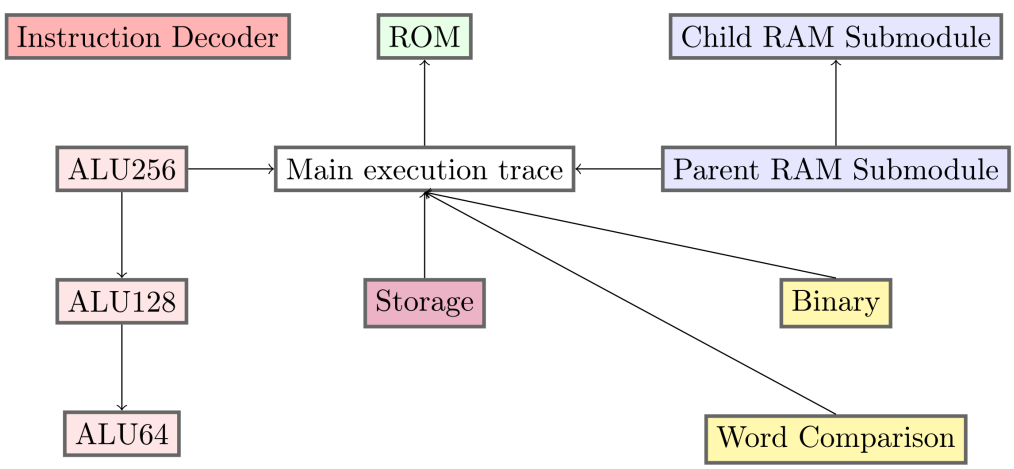


图1：zk-EVM的模块架构

-> 表示plookup包含证明。**请注意**：主执行轨迹，ALU，存储，RAM，Binary以及字比较模块同样指向指令解码器。

2.1 zk-EVM概览

我们将提供更多的zk-EVM模块细节以及他们在实际架构中的角色。2.3节对这些组件的相互作用和共同构成zk-EVM的方式进行了高度描述。

关于子模块：我们把zk-EVM模块划分成了多个子模块，这些模块之间通过Plookup证明技术连接起来。这些子模块和整体模块之间满足一个层级关系：父-子模块把（模块相关的一些）EVM指令拆分成原子操作，这些操作在子-子模块中被执行。通过平衡不同子模块之间的复杂逻辑计算，简化了复杂的EVM操作的数学形式化。

上述的父/子子模块结构已经在RAM模块和ALU模块上实现。

2.1.1 主执行轨迹

主执行轨迹，或栈轨迹（stack trace）是程序执行的核心部分。每条指令都要经过主执行轨迹，要么直接执行(比如PUSH,POP,DUP,SWAP等)，要么通过总线系统发送到其他模块——后面会展开描述。

栈内存（Stack memory）：主执行轨迹会重建EVM的栈架构。我们选择了一个ROM模型去重建这个栈：栈内存是一个ROM地址到元组(Vla, Ptr)的映射，其中Val是栈值，Ptr是一个指向包含前一个栈元素的内存单元的指针。

Mem	0	1	2	$\cdots$	$N-1$	N
Val	v_0	v_1	v_2	$\cdots$	v_{N-1}	v_N
Ptr	$\emptyset$	p_1	p_2	$\cdots$	p_{N-1}	p_N

图2：栈内存的表示

大多数的EVM opcodes需要0，1或2个输入（译者注：即栈空间大小在2以内），也有一些 EVM 指令需要的更大，比如CALL需要7。SWAP和DUP指令最大需要16(这些指令会拿到stack的第16个元素并且会访问到1-16之间的所有元素；然而，事实证明如果只追踪stack的最上面两个栈顶元素（而不是所有的16个栈顶元素）将会更加有效，因此我们的主执行轨迹也只追踪最上面两个栈顶元素。对于参数超过3个及以上（显式或隐式）的opcodes，可能需要分为多步处理（例如，执行轨迹中的多行），连续弹出stack上的2个元素（译者注：直至目标个数为止）。

相对来说，采用我们的stack跟踪方式以处理智能合约间的调用和智能合约批次处理是很容易的：stack内存可以通过添加指向自身的stack元素来重新初始化，这可以防止合约调用与它不应该访问的stack单元进行交互。处理智能合约间的调用是现有zk-EVMs模型中经常出现的一个问题

译者注：这儿说的栈结构，感觉更多像链式结构，Ptr指向前一个栈元素的内存单元指针，这样所有的栈元素都链起来了。另外，根据EVM的yellow paper[1]和源码，DUP16只需要直接访问栈顶第16个元素，并不会逐一访问栈中的1-16个元素。在ethereum 的geth客户端中 DUP1-16实现代码如下：

参数size为1-16

```
// make dup instruction function
func makeDup(size int64) executionFunc {
    return func(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext) ([]byte, error) {
        scope.Stack.dup(int(size))
        return nil, nil
    }
}
```

```
func (st *Stack) dup(n int) {
    st.push(&st.data[st.len()-n])
}
```

2.1.2 RAM模块

有效地模拟EVM的RAM本身就是一件有挑战的事情。实际中，有一些RAM相关的指令可能会从两个不同合约的内存中拷贝任意长度的字节(可能大于EVM的字长度32bytes)，如：RETURNDATACOPY, CALLDATACOPY, RETURN, BYTECODECOPY等。而另外一些指令最多操作32字节，如MSTORE, MLOAD, MSTORE8(1个字节)。

由于既能够对字节操作也能够对字操作，所以EVM内存有着复杂的I/O关系。用MSTORE可以以字节为粒度去操作内存，而MLOAD只能以字(32字节)为操作单元。EVM 内存具有字节粒度和 32 字节字地址，即寻址空间为 2^{256} ，每个元素是一个字节。

我们将内存的表示方法做了些调整，将EVM的字储存到字长整数倍的地址，而不是任意地址。保存或读取非字长整数倍的字，就涉及到内存中的最多两个相邻字。图3展示了一种这样的情况(选取4字节为一个字长)。

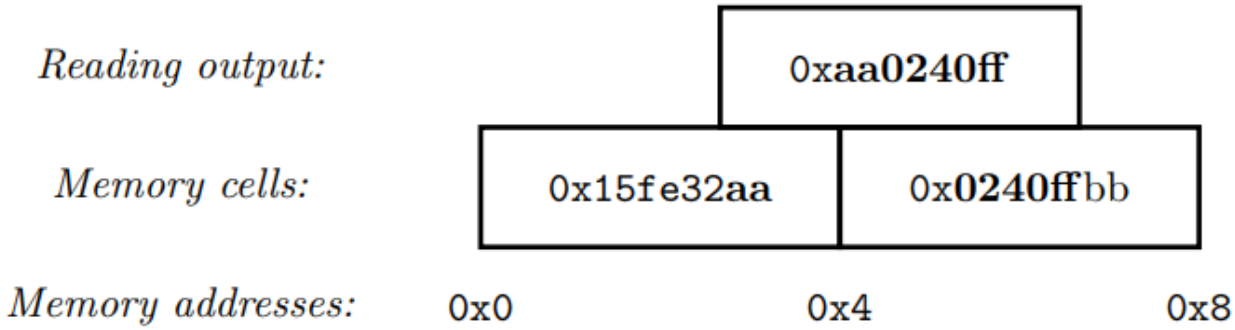


图3. 非整数偏移内存读取示例

为了处理上述挑战，我们决定将RAM模块拆分成两个字模块: Child RAM子模块和Parent RAM子模块。两个字模块中的数值一致性由Parent RAM到Child RAM的Plookup包含证明来保证。

Parent RAM子模块: Parent RAM子模块处理来自main execution trace的内存指令：先将指令拆分成若干简单的内存READ/WRITE指令，这些READ/WRITE指令操作范围最多涉及两个连续的内存字；然后将这些简单指令发给Child RAM处理。

Child RAM子模块: Child RAM子模块执行Parent RAM发送过来的简单指令，返回结果并校验RAM内存的连续性：检查RAM地址的连续；读取操作的内容与上一步储存的内容一致。

实例: 图4是一个4字节字长的Parent RAM与Child RAM的交互过程。这个过程是对一个RETURN操作符的描述，从当前RAM地址[0x13a2, 0x13aa]读取数据保存到调用方RAM地址[0xaabe, 0xaac6]中。

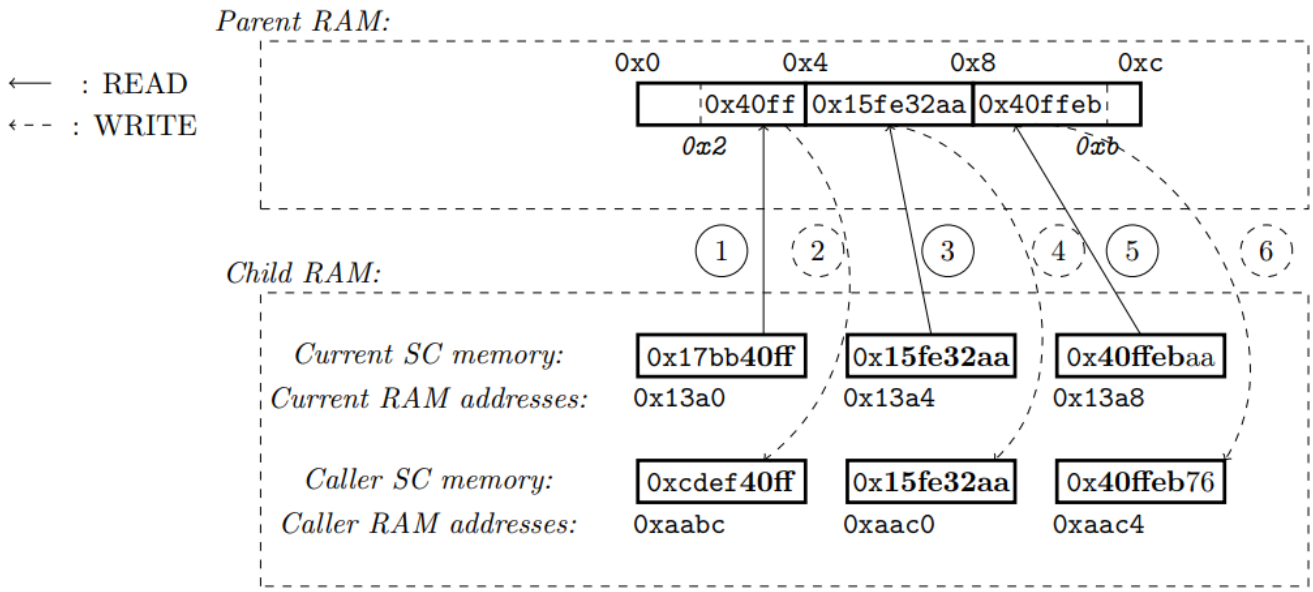


图4. Parent模块和Child模块之间的交互示意图

如图4所示，这个操作可以分解为6个步骤:

1. 读取当前RAM地址**0x13a0**的最后两个字节。
2. 将调用方RAM地址**0xaabc**的最后两个字节替换为上一步读取的内容。
3. 读取当前RAM地址**0x13a4**的全部内容。

4. 将上一步读取到的字保存到调用方RAM地址**0xaac0**中。
5. 读取当前RAM地址**0x13a8**的前三个字节。
6. 将调用房RAM地址**0xaac4**的前三个字节替换为上一步读取的内容。

2.1.3 算术操作模块

ALU：Arithmetic and Logic Unit，算术逻辑单元

本模块模拟了EVM的256-bit算术运算。它由3个子模块构成：256ALU，128ALU，64ALU；分别对应256bit，128bit，64bit算术运算。通过plookup argument技术来链接三个模块间的关系。图中给出了一个256bit数相加的简单示例。

256ALU：256-bit ALU把256-bit的输入拆分成2个128-bit数据元素（高128-bit和低128-bit），并把拆分后的数据发送到128ALU模块；

128ALU：和256ALU的模式一样，128-bit的输入会被拆分成2个64-bit的数据元素，然后发送到64ALU模块；

64ALU：执行真实的计算过程：

a. 检查输入的合法性，确实为64-bit，使用range proof（可继续拆分成4个16bit数据元素）；
b. 执行计算操作；
c. 结果逐级往上发送

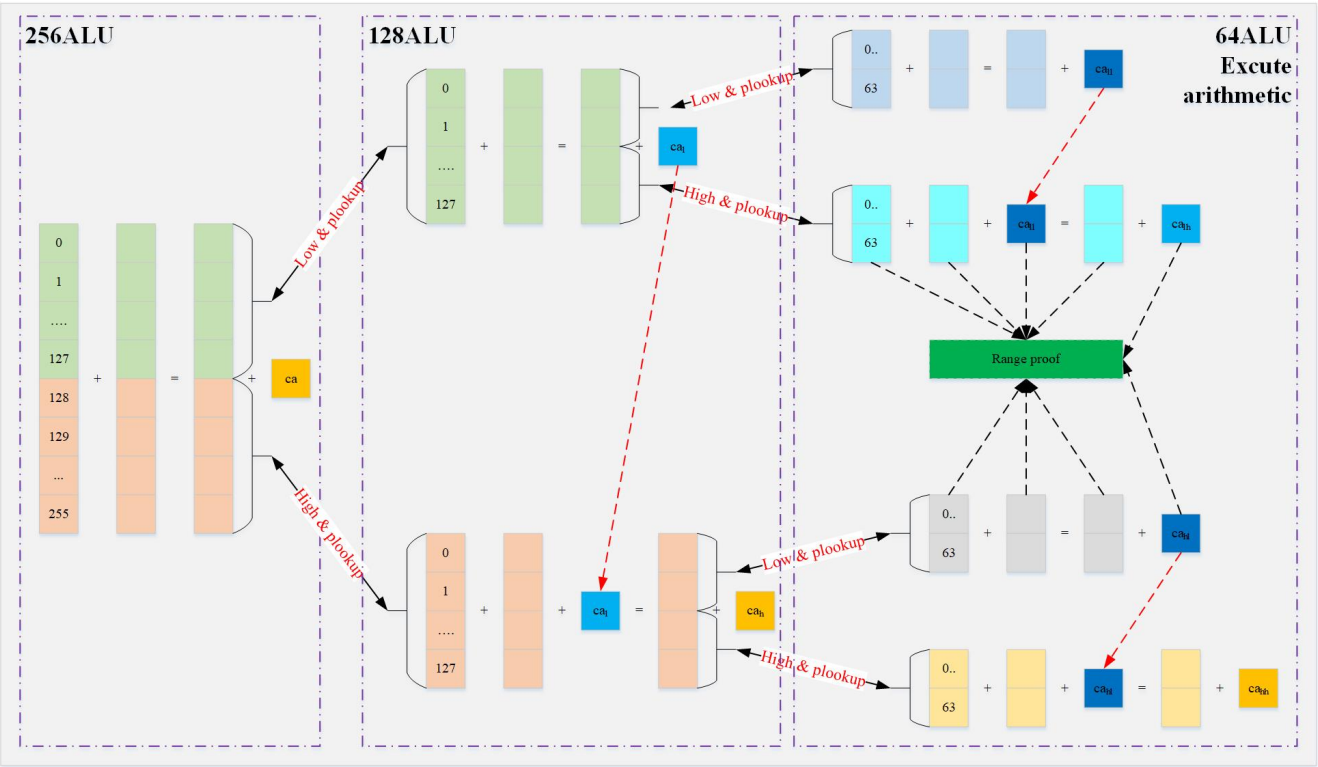


图5. 算术加法过程示意图

2.1.4 Binary, word比较模块

Binary 和 word 比较模块是两个不同的模块，它们会把field元素拆分成16-bit的word，然后进行binary相关操作(AND, XOR..)。

Binary module：主要依赖plookup argument技术，证明：

a. 一个四元组(Inst, input1,input2,res)存在于一个主执行轨迹（main excute trace）里，证明输入数据的有效性。
b. 三元组（byte1, byte2, byteres）存在于一个public table（8-bit AND table）里，证明AND操作的正确性

以下Table简单说明了上述处理过程，其中timestamp列表示被执行的指令轨迹，在处理新的指令前值不变（指令会分多步执行，把input拆分成8-bit，然后进行binary操作）

Inst	TimeStamp	Input1	Byte1	Carry1	Input2	Byte2	Carry2	res	Byteres
AND	1	0x1ea1ff	0xff	0x1ea1	0xff00ff00	0x00	0xff00ff	0x00a100	0x00
AND	1	0x1ea1ff	0xa1	0x1e	0xff00ff00	0xff	0xff00	0x00a100	0xa1
AND	1	0x1ea1ff	0x1e	0x00	0xff00ff00	0x00	0xff	0x00a100	0x00
AND	1	0x1ea1ff	0x00	0x00	0xff00ff00	0xff	0x00	0x00a100	0x00
XOR	2	0x1001	0x01	0x10	0x1010	0x10	0x10	0x1100	0x11
XOR	2	0x1001	0x10	0x00	0x1010	0x10	0x00	0x1100	0x00
...	...	...	...		...	...			...

表1. Binary模块执行轨迹示例

译者注：还需要约束：

a. 对于新的Inst的第一行，需满足：

$$Byte1_{row1} + 2^8 * Carry1_{row1} = input1_{row1}$$

b. 其他行需满足：

$$Byte1_{rowi} + 2^8 * Carry1_{rowi} = Carry1_{rowi-1}$$

c. 对于input2同理

Word comparison module：word 比较模块主要是比较两个words的大小关系(LT,GT,SLT,SGT,EQ)，并返回一个布尔类型的结果。例如，当满足a < b时，word 比较模块返回1，否则返回0；下表中，使用了**Binary** 模块相同的方式完成word比较的过程（为了方便期间，仍然以8-bit为处理单位）

TS	Signed	Equal	Inst	Input1	Input2	res	Byte1	Byte2	Bcomp	Prefix1	Prefix2	Comp
0	1	0	SLT	0xa3ff22	0xa3ffb7	1	0xa3	0xa3	0	0xa3	0x00a3	0
0	1	0	SLT	0xa3ff22	0xa3ffb7	1	0xff	0xff	0	0xa3ff	0xa3ff	0
0	1	0	SLT	0xa3ff22	0xa3ffb7	1	0x22	0xb7	1	0xa3ff22	0xa3ffb7	1
1	0	1	LT	0x00a12c	0x00a12c	1	0x00	0x00	1	0x00	0x00	1
1	0	1	LT	0x00a12c	0x00a12c	1	0xa1	0xa1	1	0x00a1	0x00a1	1
1	0	1	LT	0x00a12c	0x00a12c	1	0x2c	0x2c	1	0x00a12c	0x00a12c	1
...	...	...		...	...	...	...	...	...	...	...	...

表2. 字比较模块执行轨迹示例

依然需要使用plookup argument技术证明：

a. {Signed, Equal, Inst, input1, input2,res} 与主执行轨迹数据一致；
b. {Byte1, Byte2, Bcomp}存在于一个公开的public table里（8-bit comparison Table, 256*256 rows）

译者注：还需要约束：

a. 对于新的Inst， 第一行需满足：

$$Bcomp_1 = Comp_1$$

b. 其他行需满足：

$$Comp_i = Bcomp_i \ \& \ Comp_{i-1}$$

c. 最后一行还需满足：

$$Comp_{last} = Res$$

2.1.5 存储模块

storage模块用来表示以`字(32bytes)`为粒度存储和方便寻址的存储模块。此模块将会使用智能合约之前的storage 状态进行初始化(ZKEVM还需要保证初始化的存储一致性)。storage 模块输出执行过智能合约后的最新存储状态。

这个模块主执行2个opcode, 分别为`SSTORE`和`SLOAD`。因为存储以及读取过程的一些检查措施，使得这个模块的设计变得复杂，主要的检查项包含如下：

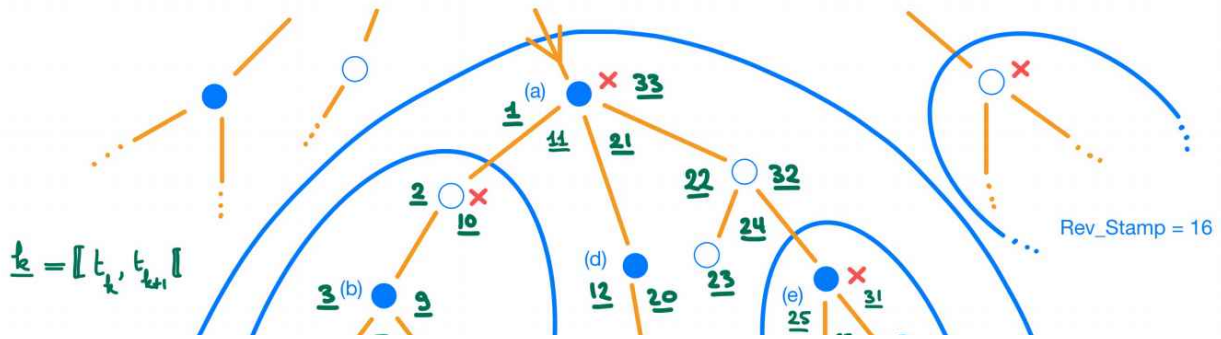
- 将从ROM中加载出来的智能合约存储初始化
- 检查在执行过程中存储使用的值是否正确
- 如果交易失败，则storage 修改的状态需要回滚
- 执行结束需要计算storage hash
- 计算与storage 相关的gas消耗情况

在一个给定的交易中，智能合约的调用形成了一棵有根的树，它的边（合约调用）是有指向性的。其顶点（智能合约）带有标签：一个智能合约地址和一组智能合约编号（1 + dv，其中dv是顶点v的出度）。对于一个给定的智能合约地址和storage地址，我们定义了一些额外的标签：storage timestamp（代表存储值的有效时间）、 revert stamp、previous revert stamp（代表当前有效存储的恢复标签、parent revert stamp (表示如果需要revert，我们应该将storage还原到哪个标签节点)。这些标签不仅取决于智能合约的地址，而且还取决于存储地址。这允许我们将标签只作用被智能合约触及到的存储地址。而不是整个智能合约存储地址。有关用来处理被revert transaction的工具的更详细解释（如revert stamps），请参考3.11节。

这些标签的主要属性如下：

- 具有给定`revert stamps`的`begin`和`end` stamps 标签 通过`empty interval`（empty interval 是指如果该存储地址的值没有在subtree 中触及）或具有`entry`和`exit` storage time stamps 的 interval 形成一个区间
- `Previous_Revert_Stamp` 表示上次对存储地址处的数据进行相关修改的Revert Stamp
- `Parent_Revert_Stamp` 标识退出当前的 revert stamp的时候需要回滚到的一个Revert stamp

接下里通过一些图来了解storage module 的设计, 下面图中蓝色实心点表示对一个智能合约 A 的调用过程，我们对于每个节点编号 (a-g) 进行区分。



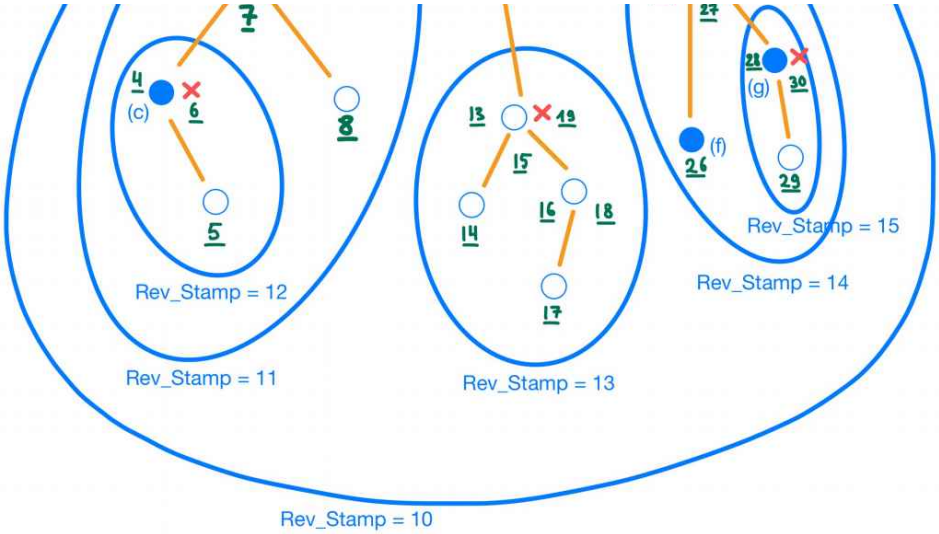
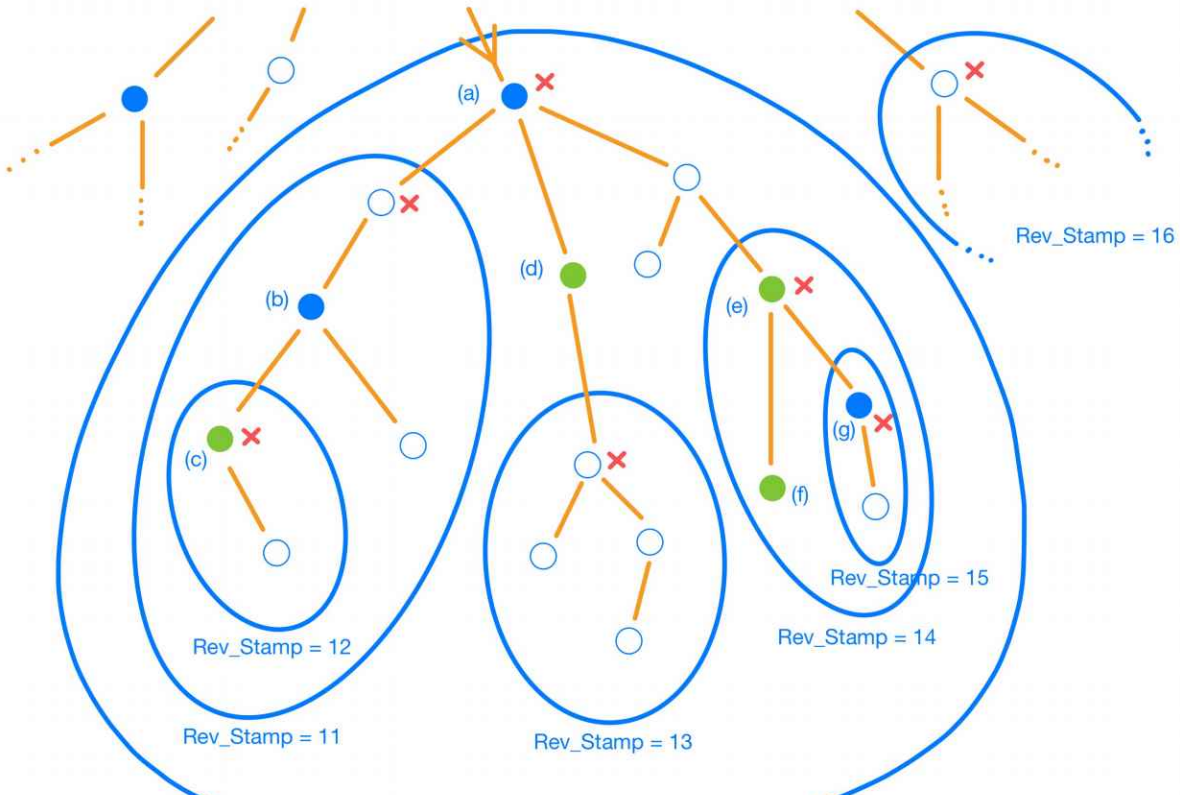


图6. 代表了交易的嵌套智能合约调用树的一部分。蓝色实心点代表对给定智能合约 A 的调用，空心点代表对其他智能合约的调用。 红色十字代表出现 `revert` 情况。 带下划线的整数 k = [[ta, tb+1 [[表示给定智能合约执行存储操作的 STORAGE_TIMESTAMP 间隔。



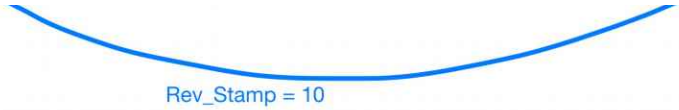


图7. 表示在合约调用树中，将对于A合约调用产生的存储地址读取或者写入节点标记为绿色

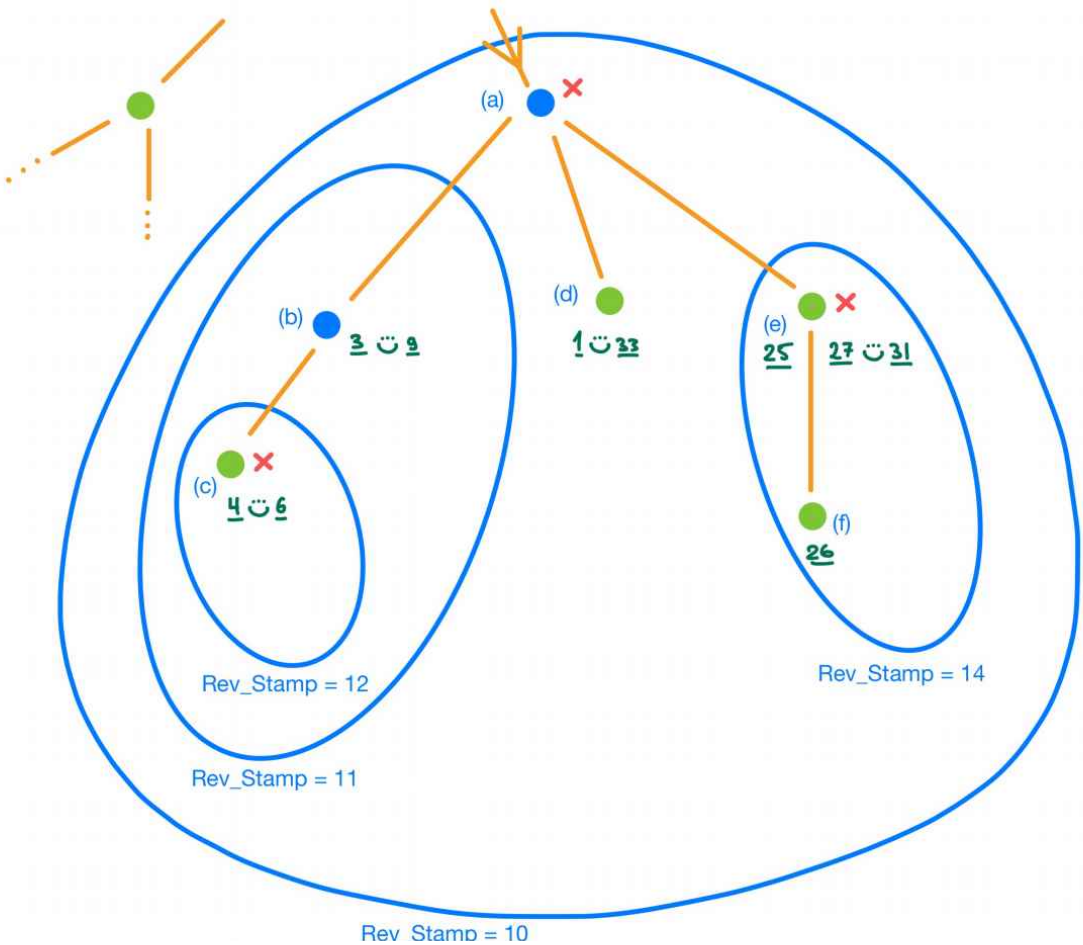


图8. 简化了图像信息。这其中丢弃了除了对A智能合约嵌套调用以外的其他智能合约调用，也丢弃了对A智能合约中不访问 `storage` 的调用以及 它的后代也不访问 `storage` 的调用。我们使用简写 $a \breve{\cup} b$ 表示 $\bigcup_{k=a,\ldots,b}$ 。我们使用 k = [[ta, tb+1 [[表示 storage 的STORAGE_STAMP 有效间隔时期。由于调用(a) revert 了，因此必须为 REVERT_STAMP = 10 这个区间初始化合约 A 的存储地址 `addr` ，这个初始值有效性持续到在REVERT_STAMP = 10 区间内的下次对于A 合约的 存储地址 `addr` 的修改。由于只有对 A 的调用（d）在 REVERT_STAMP = 10 时访问 addr存储地址,因此它的间隔是 $10 \breve{\cup} 33$ 。对 A 的调用 (b) 不访问 addr 但它的一个后代访问。 因此，我们必须在存储地址 addr 处加载当前有效值,值的有效期为 $3 \breve{\cup} 9$ 。call(b) 的后代，call(c) 访问 `addr` 存储地址 并且 revert了，它是 合约A 在 REVERT_STAMP = 12 这个区间的唯一一次调用，因此它的 STORAGE_STAMP 有效期是整个 call(c) 周期。即 $4 \breve{\cup} 6$ 。在 REVERT_STAMP = 14 时有两次对 A 的调用，一次revert了，一次没有revert（但是仍然会revert,因为它在一个REVERT_STAMP内），因此有 3 个 STORAGE_STAMP 间隔划分 STORAGE_STAMP 区间，其中 REVERT_STAMP = 14的时候被 25, 26 和 $27 \breve{\cup} 31$ 分割。

SC	$ADDR$	REV	SS	$Inst$	Val	Beg	End	$PREV$	$PARENT$
A	$addr$	10	τ_0	$INIT$	v_0	t_1	t_{10}	6	6
A	$addr$	10	τ_1	$SLOAD$	v_0	t_{10}	t_{34}	10	6
A	$addr$	10	τ_2	$SLOAD$	v_0	t_{10}	t_{34}	10	6
A	$addr$	10	τ_3	$SSTORE$	v_1	t_{10}	t_{34}	10	6

A	$addr$	10	τ_4	$SSTORE$	v_2	t_{10}	t_{34}	10	6
A	$addr$	10	τ_5	$SLOAD$	v_2	t_{10}	t_{34}	10	6
A	$addr$	11	τ_6	$INIT$	v_1	t_3	t_{10}	10	10
A	$addr$	12	τ_7	$INIT$	v_1	t_4	t_7	11	11
A	$addr$	12	τ_8	$SSTORE$	v_3	t_4	t_7	12	11
A	$addr$	12	τ_9	$SSTORE$	v_4	t_4	t_7	12	11
A	$addr$	12	τ_{10}	$SLOAD$	v_4	t_4	t_7	12	11
A	$addr$	14	τ_{11}	$INIT$	v_2	t_{25}	t_{26}	10	10
A	$addr$	14	τ_{12}	$SLOAD$	v_2	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{13}	$SSTORE$	v_6	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{14}	$SSTORE$	v_6	t_{25}	t_{26}	14	10
A	$addr$	14	τ_{15}	$SLOAD$	v_6	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{16}	$SLOAD$	v_6	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{17}	$SSTORE$	v_7	t_{26}	t_{27}	14	10
A	$addr$	14	τ_{18}	$SLOAD$	v_7	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{19}	$SSTORE$	v_8	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{20}	$SLOAD$	v_8	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{21}	$SSTORE$	v_9	t_{27}	t_{32}	14	10
A	$addr$	14	τ_{22}	$SSTORE$	v_{10}	t_{27}	t_{32}	14	10

表3. Storage模块部分执行轨迹示例

上面的表格 表示 storage execution trace，去除了其余的操作，只留下了存储执行相关的部分操作 trace。表格的表头排序方式为 合约地址在最前面，这里合约地址为A,然后是存储地址，这里表示为 `addr`, 接下来是REVERT_STAMP， 然后是STORAGE_STAMP。在表中都使用简短描述， Inst 表示 INSTRUCTION, SS 表示 STORAGE_STAMP, Beg 表示 Begin_STORAGE_STAMP, End 表示 End_STORAGE_STAMP, REV 表示 REVERT_STAMP,PREV 表示 PREVIOUS_REVERT_STAMP, PARENT 表示 PARENT_REVERT_STAMP, Val 表示VALUE。

经过分析，表格描述以下几点内容：

1. 在一个REVERT_STAMP 区间内刚开始需要执行的指令都是INIT，及时该区间不涉及storage的读写（参考REVERT_STAMP = 11），并且在INIT的时候 PREVIOUS_REVERT_STAMP 与 PARENT_REVERT_STAMP 值相同。
2. 在一个REVERT_STAMP 区间内 INIT 只会执行一次。
3. 在一个REVERT_STAMP 区间内，一旦执行过INIT后，PREV 的值将会是当前的REVERT_STAMP, PARENT_REVERT_STAMP 值保持不变。
4. 每个REVERT_STAMP区间不是包含的关系，而是一种父子关系，但是每个REVERT_STAMP 只会处理跟自己存储操作相关的内容，而不会处理自己的儿子REVERT_STAMP内容。

2.2 算术化主要想法

我们将介绍zk-EVM算术化阶段涉及到的主要概念，对模块进行更为详细的描述，包括他们的结构以及彼此间的交互方式。

2.2.1 执行轨迹

每个模块都有对应的执行轨迹。执行轨迹可以表现成一个2维数组（更高级别的表现形式），在实际的应用中，他们会被编码成多项式，并计算出对应的多项式承诺（低级别的表现形式）。执行轨迹的行数（rows）和模块中被执行的指令个数有关；执行轨迹的列数(columns)和被执行的模块相关（module-dependent），和模块中被执行的指令个数无关。执行轨迹的列数可以是在10-100之间的任意数值，而执行轨迹的行数一般是2的阶数（不够的话，则需要填充适当缺省信息）。

在实际应用中，描述每个执行轨迹的列是相当乏味的；因此，我们从Starkware开发的cairo那里借助了虚拟列/虚拟子列（virtual column/subcolumn）的概念，其可以方便且有效地描述zk-EVM的算术化过程。

虚拟列/虚拟子列: 使用我们对执行轨迹的高级表示法，一个虚拟列是执行轨迹中某一列的行的二倍周期子集。举例来说，给定一个大小为 2^i 的执行轨迹， C 代表这个执行轨迹的一个列， VC 定义为：

$$VC_k = C_{3+2^4\times k}, \forall k \in [[0, 2^{i-4}[[$$

这就是包含了 C 的所有行序号模16等于3的行的虚拟列。

一个虚拟子列是由另一列或虚拟列获得的一组虚拟列。因此，虚拟子列的概念假定了属于同一列的不同子列之间的关系依赖。

虚拟列和虚拟子列概念的主要优点是，不受特定列的确切大小的约束：执行轨迹的一列可以由几个虚拟列堆叠在一起组成。

译者注：译者从Cairo社区中得到虚拟列相关的内容，放在此处，便于读者理解。

> Cairo白皮书中关于虚拟列/虚拟子列的描述：
>
> 一组具有相同角色的跟踪单元被称为虚拟列。在收集了所有需要的虚拟列后，我们可以决定如何将它们放置在二维表中。每一个虚拟列都被周期性地放置在一个单列里面。例如，如果跟踪长度（行数）为L=16N，那么代表 pc_i 的虚拟列将在表中每16行有一个单元。由于大小之间的比率是二的幂，所以优化放置的过程很容易，然而，我们在本文中不涉及这个问题。
>
> 我们还定义了虚拟子列的概念，即从构成虚拟列的单元格中抽取一个周期性的子集，并将这些单元格作为一个虚拟列。例如，我们可以定义一个大小为4N的虚拟列，在这个虚拟列上我们将强制要求所有的值都在[0, 2^16]范围内，然后为 off_{dst} 、 off_{op0} 、 off_{op1} 定义三个大小为N的子列。这样我们就可以为父列写一套约束条件（例如，验证值是否在范围内的约束条件），并根据指令中的使用方式为 off_{dst} 、 off_{op0} 、 off_{op1} 写三套约束条件。
>
> 因此，我们的程序如下：描述几个虚拟列；然后是涉及它们的一些约束；然后可能描述更多的虚拟列和更多的约束；等等。

> Cairo社区关于虚拟列的讨论：
>
> 虚拟列只是一组代表某种作用的单元格。例如，如果我有一个重复约束，说x_i是非零的，我们可以引入一个新的y_i的 “虚拟列”，附加约束x_iy_i=1。
>
> 计算轨迹的更简单的例子是一台运行在时空(T,S)的图灵机。我们可以把它的计算表示为一个TS表，其中行是步骤，列是磁带单元。在这个例子中，列的作用很明显。在我们的例子中，我们有许多具有不同作用的单元（有寄存器pc/ap/fp），存储单元，以及用于更容易表达约束条件的辅助变量（例如，上面例子中的y_i）。每一个计算步骤都有一个完整的行（如在TM跟踪的例子中）是太昂贵了，造成了巨大的浪费。这就是为什么单元格被划分为语义角色，并以一种试图最小化表格大小和避免未使用的单元格的方式分布在表格中。

2.2.2 模块架构

模块的执行轨迹可以按列分成多个不同的集合，每一个集合在证明生成的过程中都有特定的角色，下图中给出了这样一个模块架构图。

- 指令列集合（绿色）：和给定的EVM opcode相对应的一系列的flag的集合，比如op_ADD，会对应256ALU, 128ALU,64ALU指令的flags，通过plookup argument可以确保模块中使用的flags和指令解码器中的flags的一致性（公开的固定输入）；
- 主执行列集合：和模块正在执行的指令相对应的的列的集合，通过plookup argument技术可以保证模块中正在执行的指令和主执行轨迹列集合的一致性；
- 模块特定时间执行列集合（蓝色）：是每个模块的核心部分，当模块执行到此列时，需要满足既定的各种约束；
- 模块特定空间执行列集合（黑色）：对应每个模块的内存模型；伴随着模块特定时间列集合被执行，这次列用来检查内存模型的一致性；例如，当一个read/write操作发生时，需要检查RAM在此操作发生前后的一致性；
- 范围证明列集合（橘色）：需要用于范围证明的列

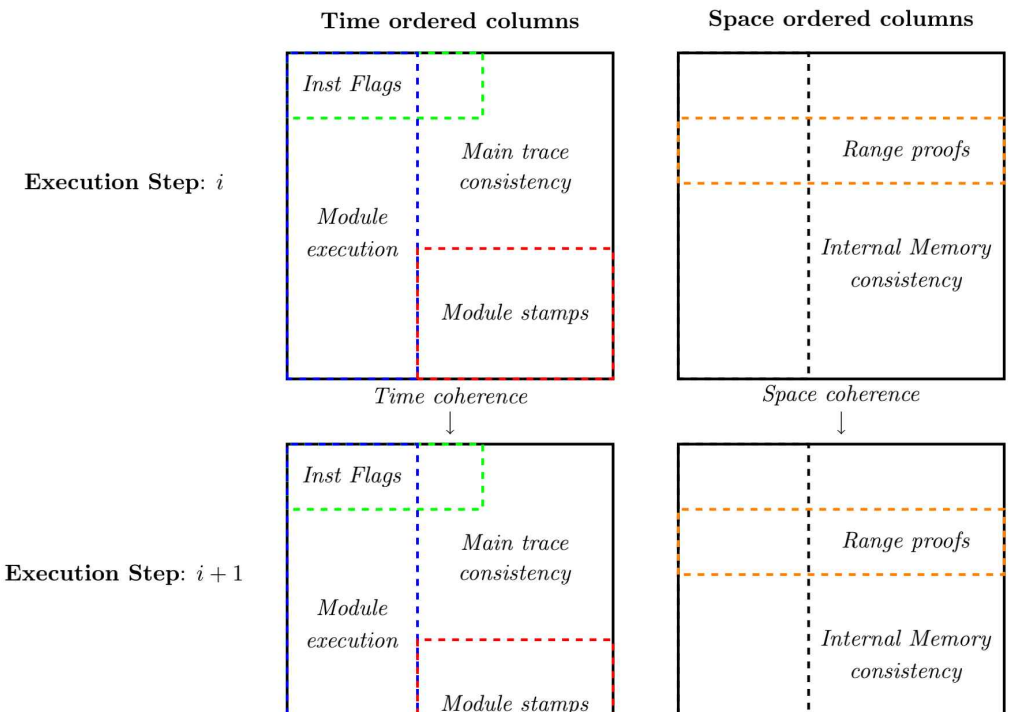


Figure 10: A module architecture illustration

图9. 模块架构说明

一些列可以属于多个特定集合，一些模块也可以包含多个特定集合；比如，在主执行轨迹里，存在两个特定空间列集合，一个确保stack内存的一致性，一个确保call stack内存的一致性。另外。一些模块可能用不到上述列的集中集合，比如**Binary**模块，不需要用到range proof列集合。

为了简化形式，我们增加了一个抽象层，把上述的5个列集合分为2个大的集合：按**时间排序的集合**和按**空间排序的集合**：

时间排序集合：展示了内部时间的一致性；两个连续的指令应该在一个模块内被连续的执行，这需要维护一个全局的执行顺序来保证时间的一致性；

空间排序集合：描述底层的内存模型；这些列被用来确保空间的一致性。一个虚拟列的连续行，应当保证其访问的地址是连续的或者是相同的。这些列确保了zk-EVM内存的一致性，比如从一个特定地址read的数据应该和上一个store在这个地址上的数据一致。

2.2.3 解决内部合约调用和批量调用

合约的内部调用意思是两个独立的执行环境（caller 和called 合约）之间的复杂交互过程。在我们的设计模式中，如果出现一个新的内部合约调用，就会创建一个新的执行环境。为了能正确的模拟内部合约调用的场景，需要注意以下几点：

- 在调用CALL指令的情况下，连个执行环境是完全独立的，双方不相互依赖（例如：被调用的合约不能访问调用者的合约内存）；

- caller和called合约之间的部分交互式允许的（例如：在caller的合约里存储从called合约里返回的值）；
- 其他的不常见的opcodes（例如：DELEGATECALL）允许called合约访问caller合约的执行环境；
- 一个合约的两个call操作会创建两个完全不同的执行环境；

我们打算用CALL STACK结构去跟踪合约的执行环境，来应对内部合约调用的场景；这个结构对于解决链式的合约调用场景非常有用，因为它需要成功的恢复前一个执行环境

CALL STACK内存和主执行轨迹保持一致性由plookup argument来确保，STACK内存同样。CALL STACK内存是一个连续的ROM，对应一个个唯一的智能合约以及和合约对应的其他的相关信息{SC_ADDRESS, Prev_TOP, Prev_RSP, Prev_PC...}。当有一系列的合约被依次执行时，合约的索引从0开始，当执行到一个新的合约时，索引加1（有可能是内部调用，也有可能是执行了一笔新的交易）；不同的智能合约根据其地址可以区分，一个给定的智能合约有可能会被调用很多次（多笔交易都调用同一个合约），我们用Next_SC_NUM来跟踪没有被分配的CALL_STACK地址。

下图为CALL STACK的一个图形化示例:

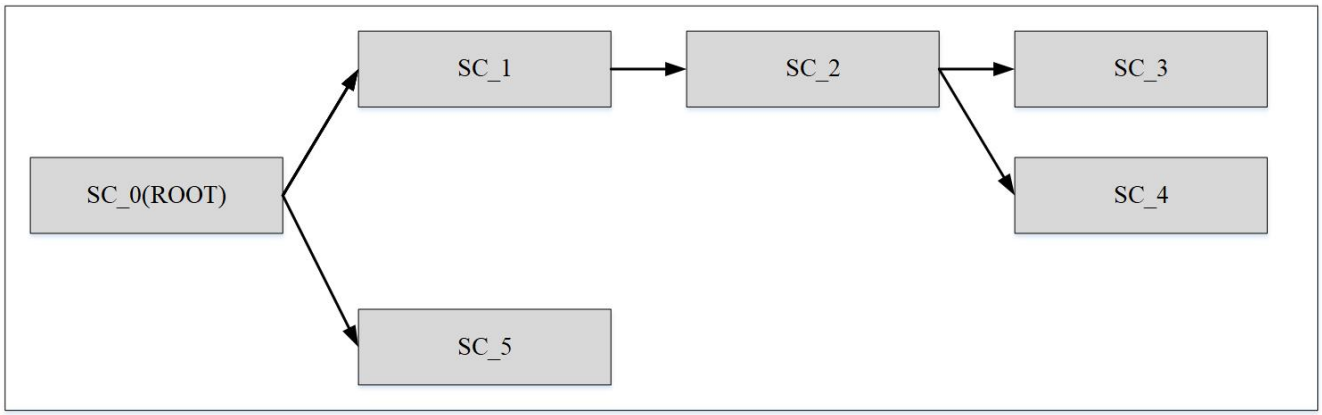


图10. 合约调用示意图

下表中给出了内部合约调用以及批量合约调用的部分执行轨迹：

Step	Inst	SC_Num	PC	Caller_Num	Prev_PC	Depth
0	GLOB_INIT	0	0	0	0	0
1	PUSH	1	0	0	0	0
2	PUSH	1	0	0	0	0
...	...	...	...	...	...	...
103	CALL	1	92	0	0	0
104	PUSH	2	0	1	92	1
...	...	...	...	...	...	...

208	CALL	2	750	1	92	1
209	PUSH	3	0	2	750	2
...	...	...	...	...	...	...
287	RETURN	3	75	2	750	2
288	PUSH	2	751	1	92	1
...	...	...	...	...	...	...
354	CALL	2	754	1	92	1
355	PUSH	4	0	2	754	2
...	...	...	...	...	...	...
360	RETURN	4	43	2	754	2
361	SWAP2	2	755	1	92	1
...	...	...	...	...	...	...
365	RETURN	2	780	1	92	1
366	MLOAD	1	93	0	0	0
	...	...	...	...	...	...
387	RETURN	1	143	0	0	0
388	PUSH	5	0	0	0	0
...	...	...	...	...	...	...
453	RETURN	5	230	0	0	0
0	GLOB_END	0	1	0	0	0

表4. 合约调用执行轨迹示例

2.3 整合到一起

我们的zk-EVM架构已经是相当的复杂 -- 模块，指令解码，flags，总线系统等等。本章节将注重总结“这些模块以及他们各自的约束系统如何与主执行轨迹相关联”，并且把这些融进zk-EVM架构里。设想一下，我们想证明一个智能合约的执行过程（其中合约的bytecode和storage已经存在了我们的zk-rollup系统里），应该主要分为以下几个步骤：

- 首先，我们把合约的所有bytecode存在ROM里，然后建立了对应的bytecode column和PC column；为了证明ROM里的bytecode是正确的，我们计算它的root hash，并且和储存在zk-rollup上的codehash相比较，用来确保，**ROM里存的是正确的合约程序**；
- 接下来，我们需要**证明程序被正确的执行**；在智能合约执行的时候，我们把顺序执行的指令写到**主执行轨迹**里，为了保证执行opcode确实是ROM里的bytecode，我们利用plookup argument技术去保证执行的Op和ROM里的Bytecode一致（比较执行轨迹和ROM里的PC和bycode是否一致）

我们的zk-EVM架构不能直接的解析这些EVM指令，因此，我们不得不把 $bytecode_{exectrace}$ 列翻译成一系列各模块对应的flags。这就是指令解码器的作用（一个公共的，不变的表格，其对一系列数值的承诺可以被用来一致性校验），指令解码器允许把EVM opcodes拆分成一系列可以被主执行轨迹和其他相关模块识别的flags（ $IDFlag_1, ..., IDFlag_n$)。那么，问题就转化成了检查 "执行轨迹里的指令flags（暂且标记为（ $ETFlag_1, ..., ETFlag_n$))和硬编码在指令解码器里的flags的一致性" 问题，可以用plookup argument来解决。

- 在具备n+1个元素的两个元组之间进行plookup argument协议，
  $(Bytecode_{exectrace}, ETFlag_1, ..., ETFlag_n)$ 和
  $(EVM_{opcode}, IDFlag_1, ..., IDFlag_n)$

需要注意的是，在主执行轨迹和各模块执行轨迹里，都需要进行指令拆分。一般性的思想是：

- 主执行轨迹里的flags可以通过plookup argument协议来证明模块执行轨迹和主执行轨迹之间的一致性关系；
- 模块里的flags被用来进行模块执行轨迹的约束满足性的校验；

现在，EVM的指令可以被zk-EVM方案解析了，接下来，我们需要执行这些指令并且证明这些指令被正确的执行了。通过给这些指令flags的特定赋值，整个zk-EVM电路里的不同功能模块将会被激活：

a. 例如，为了确保合约里的Memory相关的指令发送到了RAM模块，主执行轨迹里的RAM_Instruction flag应该被赋值。当这个指令被赋值时，那么相关的约束也要被激活，比如当一个memory指令执行结束后，RAM的timestamp应该自增1；
b. 在RAM模块中，通过给相关的flags赋值，来激活模块内部的一些行为；比如，用于区分MSTORE和MSTORE8指令的flags等等；
c. RAM执行轨迹会填充一常量列（值为1）
d. JUMP指令通常会引起PC的不寻常改变，当JUMP_FLAG被赋值的时候，表明执行PC普通更新的circuit将不会激活，执行PC特殊更新的circuit将会被激活；
e. 需要注意的是：模块内约束的数量和模块被要执行的指令个数是完全独立的，模块间的一致性约束数量取决于模块本身，是固定的；通过给指令特定的flags赋值，其对应的约束和行为将被激活；

每个模块和主执行轨迹之间通过plookup argument来约束关系，当一个指令被发送到对应的模块是，在主执行轨迹中，对应的MODULE_INST flag将会被赋值，MODULE_STAMP用来跟踪发送到

对应模块的指令数，而stack上的top元素也会作为模块的输入；因此，plookup argument应该证明模块的（MODULE_INST, INST,MODULE_STAMP, INPUT1,INPUT2）列集合存在于主执行轨迹的某些列中。

3. 工具和概念

我们将在本章主要介绍zk-EVM架构中涉及到的相关工具和技术。读完本章后，您将理解如何（1）在一个给定的执行轨迹内，证明一些子列之间的关系；（2）在不同的执行轨迹内，证明某些子列之间的关系，也就是zk-EVM子模块之间的关系；（3）以上两点完整了描述了zk-EVM的逻辑框架。

3.1 交叉操作符 $\odot$

给定两个长度为N的列向量 $A=(a_0,a_1,...,a_{N-1})^T$和$B=(b_0,b_1,...,B_{n-1})^T$ ，A和B交叉的结果就是一个长度为2N的向量，其元素由向量A和B的元素交叉连接组成：

$$A\odot B=(a_0,b_0,a_1,b_1,...a_{N-1},b_{N-1})^T$$

如果A和B被看做是多项式， $A\odot B$ 被看做是多项式R，则有：

$$k=even,R(k)=A(k/2);k=odd,R(k)=B((k-1)/2)$$

3.2 基于列的置换论证[5, 9](Permutation argument)

置换论证的目的是为了证明：对于向量A和B，一个是另外一个的置换排列（相同的元素集合，不同的顺序）。对于向量A和B，如果满足上述条件，则存在一个置换多项式 $\sigma(x)$ 满足，对于任意一个元素i，有： $B_i=A_{\sigma(i)}$;相同的值得集合，其元素的乘积是相等的；但这只是个充分不必要条件（例如{1,4}和{2,2}）；因此，需要引入一个随机数（不是prover生成的），来证明以下等式成立：

$$[\prod_{A}^{B}]_i=\prod_{0\le l<i}(z-A_l)/(z-B_l)$$

需满足： $[\prod_{A}^{B}]_N=1$

注意：这里的置换论证并没有指定具体的索引，和PLONK的描述的稍有不同；和Halo2的思想一致。

3.3 基于矩阵的置换论证

我们把3.2描述的置换论证协议进行扩充。假设存在两个矩阵A和B，如果存在一个置换多项式 $\sigma(x)$ 满足： $B_{i,j} = A_{\sigma(i),j}$ ，则定义一个新的多项式，如下：

$$[\prod_{A}^{B}]_i = \prod_{0 \le l < i} (z - \sum_{k=1}^{r} \theta_k A_{l,k}) / (z - \sum_{k=1}^{r} \theta_k B_{l,k})$$

上述公式证明了两个矩阵之间的批量置换关系，事实上，如果两个矩阵之间的每一列满足章节3.2所描述的列置换关系，那么其列之间线性累加得到的新的列也满足置换关系（同样是相同的值得集合）。

在我们的实际应用中，矩阵A和B分别对应的r列数据， $A_1, A_2, ...A_r$ 和 $B_1, B_2, ...B_r$ ，他们都取自不同模块的执行轨迹；我们需要这样的置换论证的原因是：

空间-时间重新排序：stack，memory和storage模块必须满足两个独立的约束，即时间的一致性和空间的一致性。比如，stack的状态必须和EVM的执行过程相一致，因此，当执行一个pop操作时，相应的pointer值应该做对应的修改，这是**时间一致性**；另一方面，当执行连续的几个pop操作时，其弹出的值有可能是已经写入很久了，因此，我们需要保证，当前弹出的值和之前写入的值是一致的，这是**空间一致性**。

基于Cairo的range proof：将在后面介绍(3.5章节)。

我们用 $B = \tilde{A}$ 来表示矩阵B是矩阵A的批量置换关系，当我们用到这个表达式的时候，就表明我们会在执行轨迹上默认填充一列 $[\prod_{A}^{B}]$ 来witness置换论证协议。

译者注：这里的批量置换论证，也是基于**相同列**索引下的置换论证，和Halo2里的略有不同，其支持任意行之间的置换关系。

3.4 Plookup[10]

我们的zk-EVM架构是模块化的 —— 不同的功能对应不同的单元，它们对应不同的约束集和不同的执行轨迹（但执行轨迹之间也是存在某种关系的）；因此，就会出现，有的模块负责256-bit的运算，有的模块负责二进制运算，有的模块负责memory模块的操作。所以，对于一个模块来说，可以把相应的过程代理到对应的模块上去去执行，但是它们之间的关系，需要Plookup argument来保证：

a. 不同子列之间的Plookup argument：我们可以通过Plookup argument来约束执行轨迹的某些列之间的数值关系，这主要有两个目的：
   i. 通过约束两个模块之间的关系，来保证数据的有效性；比如，第二个模块执行的是特定的计算过程，通过Plookup argument来约束第一个模块和第二个模块执行轨迹的某一列的数据，来确保数据确实是满足特定的计算关逻辑的；
   ii. 去校验某些模块本身不能验证的约束；
b. 基于Tables的Plookup argument：在有限域上执行某些算术运算是非常低效的，因此可以通过一个预先生成的Tables，证明执行轨迹里的某些列和Table之间的Plookup关系，来简化电路规模。可以通过设置flags来使能相应的列，来证明其和table之间的关系。

3.5 范围证明(Range proof)

基于Cairo的范围证明用于证明“一个值在给定的连续区间范围[a,b]内”。首先，我们需要对这一列数据进行排序；然后，填充缺失的值；最后，我们使用置换论证来保证重新排序后的列，其取值范围是[a,b]，并且相邻行之间的值要么相同，要么+1。

译者注: 和Halo2的Lookup论证思想类似。

3.6 if-elseif-else逻辑

EVM的执行逻辑需要处理当某种条件是否被触发时对应的分支逻辑。特别的，我们需要处理以下类别的情况：

a. 给定一个布尔变量b，根据b =0 或者b = 1来执行轨迹P0和P1；更普遍的是，给定一系列的约束变量 $c \in (c_1, c_2, ...c_k)$ ，通过给定c不同的赋值，来执行不同路径 $P_1, P_2, ...P_k$ ；

b. 给定一个任意的变量 $a, a \in F$ ，根据a的取值是否为0来执行 P_0 或 $P_{\neq 0}$

通常，变量的值可以从执行轨迹上获取；如果变量是布尔类型，则约束可以定义为 $b^2 - b = 0$ 且对应的分支约束的执行可以约束为 $((1-b)P_0 + bP_1)$ ，当b = 0时，执行 P_0 ；当b=1时，执行 P_1 。对于一些列的变量c，则需要分别约束 $\prod(c - c_i) = 0$ 和 $\sum_i L_i(c)P_i$ ；对于b的场景，我们需要区别a是否为0，因此，我们需要新增一列，这一列的取值的1/a，然后校验a * (1/a)是否是一个布尔值。

3.7 GKR哈希[11]

我们的zk-EVM方案使用了和GKR算法有效结合的代数哈希函数，比如MIMC哈希。因此，当我们对不同的子模块的执行轨迹进行Plookup argument论证时，我们还需要一个额外的GKR验证模块。

参考链接：https://ethresear.ch/t/using-gkr-inside-a-snark-to-reduce-the-cost-of-hash-verification-down-to-3-constraints/7550

3.8 模块时间戳

在我们的设计中，时间戳是用来表示不同模块之间的链接关系的。对于每个模块来讲，存在两种时间戳，指令时间戳和原子时间戳：当一个新的指令被开始执行时，指令时间戳+1；在指令的执行过程中，每向模块bussing发送一次请求，原子时间戳+1；比如，一个Call操作需要访问7个元素，那就需要向memory模块发送4次请求，执行结束后，指令时间戳+1，原子时间戳+4。这些时间戳被用来证明：

a. 子模块执行的指令同样存在于主执行轨迹中；

b. 从父模块发送到子模块的请求已经被处理；

GLOBAL_INIT是一个特殊的opcode，它最早被执行，用于跟踪执行的指令总数，模块调用和初始化模块的初始时间戳，它将会和最后一个执行的GLOBAL_END opcode进行比较。

3.9 约束传播

后续的一个重点是：当我们把ROM模型和Plookup argument结合时，对于执行轨迹，会隐式的引出一些约束；比如，当执行一个指令后，memory的Read Stack Pointer（RSP）和指定栈顶的指针被重新指定，那么其对应的值也要一并修改(Val(RSP)和Val(Top))。

3.10 标志和指令解码器

我们zk-evm的另一个主要组件是指令解码器，它基本上是一个多路复用器，它将一个操作码与一系列标志序列相关联，这些标志序列完全决定了我们zk-evm的行为。指令解码器是一个公开的数据，它清晰表明了不同标志集之间的约束，比如，当CALL标志被赋值时，RETURN标志必须是一直关闭的，因为这是CALL指令执行过程中，永远不会用的指令。这极大的减少了执行轨迹的规模，因为指令解码器本身的约束导致一些执行分支根本不用去执行。

3.11 处理 revert 交易

EVM 中的 `REVERT` 操作码和异常情况终止会恢复当前交易中已执行的更改 - 当前执行的合约的存储必须重新初始化到它之前的值，还有那些来自当前合约的嵌套调用的存储也需要重新初始化到它之前的值。在本节及后续章节中，我们将介绍以下两个非常重要的定义。对于 `REVERT` 操作码的形式化很有用。

1. 一个 `reverted set` 是 revert 交易的子代的集合。 任何revert 后的 的交易都属于一个 `reverted set`。我们接下来介绍几种工具和概念来处理 REVERT 操作码和 error flag：
    - 一个 `Curr_REV` 虚拟列，当当前交易或者子交易被reverted 的时候激活。
    - `REV_FLAG` 虚拟列，在执行 `REVERT` 操作码时激活。 设置 `REVERT_FLAG` 时，也必须设置 `interrupt flag`
    - 一个 INTERRUPT_FLAG，它是一个虚拟列，只要设置了 error flag 或执行了 REVERT 操作码，就会激活。一个 `REV_STAMP` 对每个还原集都是唯一的。 请注意，如果给定的还原集 $\mathcal{R}_C$ 包含在另一个还原集 $\mathcal{R}_P$ 中，则与属于 $\mathcal{R}_C$ 的交易关联的revert stamp 不同于属于 $\mathcal{R}_P - \mathcal{R}_C$ 的交易
    - `Parent_REV_STAMP`：跟踪包含当前 `reverted set` 的 `reverted set` 的 revert stamp。 按照惯例，如果当前交易不包含在任何 `reverted set` 中，则 `Parent_REV_STAMP = 0`
    - `Next_REVERT_STAMP`：跟踪下一个revert stamp的编号

为了连贯地构建这些标志，必须检测何时进入或离开 `reverted set`，以及在内部合约调用的情况下，reverted 的交易如何自动传输 `Curr_REVERTED` flag。 基本上，这三种情况完全确定了 `Curr_REVERTED` 和 `REVERT_STAMP` 更新:

1. 如果当前指令是 RETURN 或 STOP，没有设置 `Curr_REVERTED` 并且设置了 `Curr_REVERTED` flag，则父合约也必须reverted（即设置其 `Curr_REVERTED` 标志)。否则

存在合约的正确执行（它已经返回了一个值）和它已经被还原的事实之间的矛盾。

2. 如果当前指令为CALL，并且设置了 `Curr_REVERTED` flag，则设置被调用的合约 `Curr_REVERTED` flag
3. 如果当前指令既不是CALL，也不是RETURN，也不是STOP，则验证下一条指令的Curr_REVERTED标志与当前指令相同。
4. 如果设置了 `INTERRUPT_FLAG` ，请检查是否设置了 `Curr_REVERTED` flag。将当前的 `REVERT_STAMP` 设置为 `Next_REVERT_STAMP` ，并增加 `Next_REVERT_STAMP` 。定义了用于 `REVERTED` 更新的工具后，现在可以开始使用它们按照 `REVERTED` 语句更新 `STORAGE` 。

3.12 错误标识

根据以太坊黄皮书定义，以下是触发EVM异常停止的情况：

- gas 不足
- 非法指令
- JUMP/JUMPI到一个无效的目标地址
- 栈空间不足
- 栈溢出（栈大小超过1024）
- 从不存在的位置复制 RETURNDATA
- static call 期间 修改状态

大多数这些情况都可以简单地使用上面描述的工具和技术来处理。例如:

- 针对gas不足错误标志的处理，可以对每一步的gas值与其需要减少的量进行一个word comparison，当gas值不足以进行继续执行的时候会引发错误标志。
- 使用 `stack size` 跟踪器可以轻松处理 `stack` 相关的错误，并验证 `stack size` 是否始终包含在 [0,1023] 中，如果不再是这种情况，则设置错误标志。
- `RETURNDATA` 相关标志通过跟踪 `RETURNDATASIZE` 并在溢出的情况下设置相关的错误标志来处理。
- `static call` 调用期间的状态修改通过在 `static call` 调用期间，调用了其他的禁止指令时引发标志来处理
- 为了检查JUMP/JUMPI的目的地的有效性，可以对当前程序中PC的最大值进行范围检查，并在跳转到PUSH参数的情况下在ROM的合约的bytecode中添加一个 `PUSH_ARG` 标志。 本方案是在以太坊黄皮书识别非法跳转目的地情况的方案。
- 指令有效性检查有点复杂。 实际上，这相当于验证指令操作码不属于 80 个 EVM 指令的集合（这不是一个连续的集合值）。 通常，此检查是使用 Plookup 执行的，它很难适应 try/catch 逻辑。执行指令有效性检查的一种解决方案是用 INVALID 指令，然后施加一个操作码最大值。 然后，对于每条指令，必须执行范围证明以验证操作码是否属于授权的一组值。 在该值范围内，Plookup 将区分指令解码器内的合法和无效操作码。

3.13 Fees/Gas Costs

在处理完error flags 之后，处理fees（或gas costs）是zk-EVM的另一个棘手的部分，尽管对于在智能合同执行中允许向后兼容是必要的：

根据以太坊黄皮书，智能合约执行中可能会产生三种类型的 gas 成本：

1. 当前执行的opcode固定的gas费用。 EVM 的大多数常见指令（ADD、SWAP、DUP、...）都是这种情况
2. memory 扩展的 费用 取决于当前meory的大小。 此扩展费用取决于到目前为止存储在 RAM 中并更新的 `EVM words`，在程序执行过程中动态更新。memory 扩展的费用等价于storage的模式，当第一次访问某一个给定地址的存储时候将会收取更多的gas费用，其他时候则相对第一次便宜一些。还有一种情况是将storage 存储的值设置为0将会返还一定的gas。
3. 最后一类gas费用与智能合约交互有关。 此费用用于执行*CALL* 或 *CREATE* 等指令。 这些费用称为动态费用，是适应 zk-EVM 框架的最复杂的费用类别。

当前的费用价值将使用 *gas* 虚拟列进行跟踪。 当前内存大小在 *MEM_SIZE* 变量中跟踪，该变量在 child 内存模块中作为内存段更新被访问。 这个 *MEM_SIZE* 变量允许我们在程序执行中计算动态的 gas fee。

4. 字比较模块

字比较模块处理以下4种字比较指令：

- *LT*
- *GT*
- *SLT*
- *SGT*

分别返回当a，b作为输入时，布尔值 a <= b，a >= b，a < b，以及a > b。

4.1 Trace列

本模块由以下列组成：

- *WCTimeStamp*：每条指令都会加1，并且在执行字比较指令的过程中保持不变。
- *Inst*：指令
- *SwitchFlag*
- *EqFlag*
- *Input_1*：要比较的第一个字，在指令执行过程中保持不变。
- *Input_2*：要比较的第二个字，在指令执行过程中保持不变。
- *Comp*：返回*Input_1* < *Input_2*比较结果，在给定的时间戳中保持不变。如*Comp*计算*SLT*的结果。

- *Prefix_1*：一个byte一个byte构造*Input_1*的前缀序列
- *Prefix_2*：和*Prefix_1*一样，用于*Input_2*
- *B_1*：组成*Input_1*的字节序列，从最好字节开始到最低字节
- *B_2*：组成*Input_2*的字节序列，从最好字节开始到最低字节
- *BComp*：此列包含*B_1* < *B_2*比较的结果
- *Decided*：只要知道*Comp*的结果，这一列就可以从0切换到1
- *Res*

*SwitchFlag*和*EqFlag*编码了四个比较操作：

	SwitchFlag = 0	*SwitchFlag* = 1
EqFlag = 0	*SLT*	*SGT*
EqFlag = 1	*LT*	*GT*

4.2 Trace约束

结果列使用等价计算指令的结果

$$\begin{cases} LT : a <= b \Longleftrightarrow (a < b)\ OR\ (a = b) \\ SGT : a > b \Longleftrightarrow NOT((a < b)\ OR\ (a = b)) \\ GT : a >= b \Longleftrightarrow NOT(a < b) \end{cases}$$

因此，我们只需计算比较a<b，这就是*Comp*列的目的，它总是计算布尔值 $Input\_1 < Input\_2$ 。

1. 列*Inst*以及其标识列*SwitchFlag*、*EqFlag*，被Plookup验证为指令解码器。
2. 列*B_1*、*B_2*和*BComp*是根据一个存储所有256×256字节比较的查找表进行Plookup验证的。
3. *WC_TimeStamp*被初始化为对字数统计（Word Count）模块的总调用次数。这个条件是必须的，以确保包含证明（inclusion proof）不会“错过”对字数统计模块的请求。
4. i = 1， $WC\_TimeStamp_1$ = 0
5. i >= 1， $WC\_TimeStamp_{i+1} = WC\_TimeStamp_i$ 或者 $WC\_TimeStamp_{i+1} = WC\_TimeStamp_i + 1$ 。比如：
   $(WC\_TimeStamp_{i+1} - WC\_TimeStamp_i)(WC\_TimeStamp_{i+1} - WC\_TimeStamp_i - 1) = 0$
6. i <= 1， $WC\_TimeStamp_{i+32} = WC\_TimeStamp_i$ 或者 $WC\_TimeStamp_{i+32} = WC\_TimeStamp_i + 1$ 。因为字的长度固定是32个字节，所以32行足以用来字比较。

7. *Res*、*Comp*、以及*Decided*是二进制的，所以对于所有的i，有：
   $Res_i(Res_i-1)=0, Comp_i(Comp_i-1)=0, Decided_i(Decided_i-1)=0$ 。

其他的变量根据 $WC\_TimeStamp_i = WC\_TimeStamp_{i-1}$ 是否成立来更新：

1. **如果**- $WC\_TimeStamp_i = WC\_TimeStamp_{i-1}+1$ ：
   a. $(B\_1)_i = (Prefix\_1)_i$ 且 $(B\_2)_i = (Prefix\_2)_i$
   b. $(Prefix\_1)_{i-1} = (Input\_1)_{i-1}$ 且 $(Prefix\_2)_{i-1} = (Input\_2)_{i-1}$ ，比如，我们希望前缀等于字尾的输入比较。
   c. **如果** - $(B\_1)_i = (B\_2)_i$ 那么 $Decided_{i-1}=0$
   d. **否则如果** - $(B\_1)_i \neq (B\_2)_i$ 那么 $Decided_i = 1$ 且 $Comp_i = BComp_i$
   e. **如果** - $(Input\_1)_i = (Input\_2)_i$ 那么 $Comp_i = 0$
2. **否则如果**- $WC\_TimeStamp_i = WC\_TimeStamp_{i-1}$ 那么
   a. $(Prefix\_1)_i = 256\cdot(Prefix\_1)_{i-1} + (B\_1)_i$ 且
   $(Prefix\_2)_i = 256\cdot(Prefix\_2)_{i-1} + (B\_2)_i$
   b. 输入、指令、以及其标识，还有 $Input\_1 < Input\_2$ 比较结果，再执行单个字比较过程中保持不变：

$$\begin{cases}(Input\_1)_{i-1} = (Input\_1)_i \\ (Input\_2)_{i-1} = (Input\_2)_i \\ Inst_{i-1} = Inst_i \\ SwitchFlag_{i-1} = SwitchFlag_i \\ EqFlag_{i-1} = EqFlag_i \\ Comp_{i-1} = Comp_i \\ Res_{i-1} = Res_i \end{cases}$$

   c.
      i. **如果**- $Decided_{i-1}=0$ ：
         1. **如果**- $(B\_1)_i = (B\_2)_i$ 那么 $Decided_i = 0$ 。
         2. **否则如果**- $(B\_1)_i \neq (B\_2)_i$ 那么 $Decided_i = 1$ 且 $Comp_i = BComp_i$
      ii. **否则如果**- $Decided_{i-1}=1$ 那么 $Decided_i = Decided_{i-1}$ 。

我们已经规定在指令执行过程中，*Res*保持不变。

1. **如果**- $SwitchFlag_i = 0$ 且 $EqFlag_i = 0$ ：那么 $Res_i = Comp_i$
2. **如果**- $SwitchFlag_i = 0$ 且 $EqFlag_i = 1$ ：那么 $Res_i = 1$ 当且仅当 $Comp_i = 1$ 或 $(Input\_1)_i = (Input\_2)_i$

3. **如果**- $SwitchFlag_i = 1$ 且 $EqFlag_i = 0$ ：那么 $Res_i = 0$ 当且仅当 $Comp_i = 1$ 或 $(Input\_1)_i = (Input\_2)_i$
4. **如果**- $SwitchFlag_i = 1$ 且 $EqFlag_i = 1$ ：那么 $Res_i = 1 - Comp_i$

5. Parent RAM子模块的约束

5.1 相关指令

- CALL
- RETURN
- MSTORE
- MSTORE8
- MLOAD
- CALLDATALOAD

暂未实现的指令

- CALLDATACOPY
- RETURNDATACOPY
- RETURNDATASIZE
- CODECOPY
- EXTCODECOPY
- DELEGATECALL
- STATICALL
- CALLCODE

5.2 Trace列

5.2.1 Stack Trace列：

- Stack_Inst: 将被执行的指令。
- RAM_STAMP
- $Input_1$: 第一个输入
- $Input_2$: 第二个输入
- Result: 指令结果(如果有，将被返回到Stack中)
- END_FLAG^0
- INVALID_INSTRUCTION

5.2.2 指令解析列：

- IOPC_FLAG: 如果被设置，选取交互性的内存opcode如：CALL，RETURN。否则选取非交互的内存opcode如：MLOAD，MSTORE，MSTORE8，CALLDATALOAD。
- MOPC_FLAG: 非交互式内存操作码的位选择标志。如果设置，选取MSTORE，MSTORE8，否则选取MLOAD，CALLDATALOAD。
- Size_FLAG: 内存中将被读/写字数
- RET_FLAG: RETURN指令
- CALL_FLAG: CALL指令
- CALLDATA_FLAG: (CALL指令过程中)指明内存被用作写calldata。
- INIT_FLAG: CALL指令后初始化RAM

5.2.3 Parent RAM子模块指令解析列：

- Inst_BInt_Offset: 指令起始点字的内部偏移量
- Inst_BWd_Offset: 指令起始字的偏移量
- Inst_EInt_Offset: 指令终止字的内部偏移量
- Inst_EWd_Offset: 指令终止字的偏移量
- Curr_BInt_Offset: 当前起始字的内部偏移量
- Curr_EInt_Offset: 当前终止的内部偏移量
- Curr_BWd_Offset: 当前起始字的偏移量
- Curr_EWd_Offset: 当前终止字的偏移量(被约束为Curr_BWd_Offset或Curr_BWd_Offset+1)
- Curr_Wd_Val: 当前读/写字的值
- Aux_BWd_Offset, Aux_BInt_Offset, Aux_EWd_Offset, Aux_EInt_Offset, Curr_Aux_BWd_Offset, Curr_Aux_EWd_Offset, Curr_Aux_BInt_Offset, Curr_Aux_EInt_Offset: 合约调用的辅助参数

5.2.4 Child RAM模块列(整字内存)

- CRAM_Inst: 将被Child RAM子模块执行的指令(可以是读(CRAM_Inst=1)或写(CRAM_Inst=1))
- CRAM_STAMP: Child RAM子模块时间戳
- CRAM_BWd_Offset: 读/写的起始字地址
- CRAM_EWd_Offset: 读/写终止字地址
- CRAM_BInt_Offset: 读/写起始字的内部起始地址
- CRAM_EInt_Offset: 读/写终止字的内部终止地址
- Val(CRAM_Inst): RAM指令的值
- CRAM_SC_Num: 读/写的合约内存
- CRAM_CALLDATA_FLAG: 标志着指令是作用于RAM还是CALLDATA

5.2.5 调用栈/合约的批量处理

- SC_Num: 当前合约编号
- Caller_Num: 调用方合约编号
- Ret_Offset(SC_Num)，Ret_Len(SC_Num): 包含在执行栈中

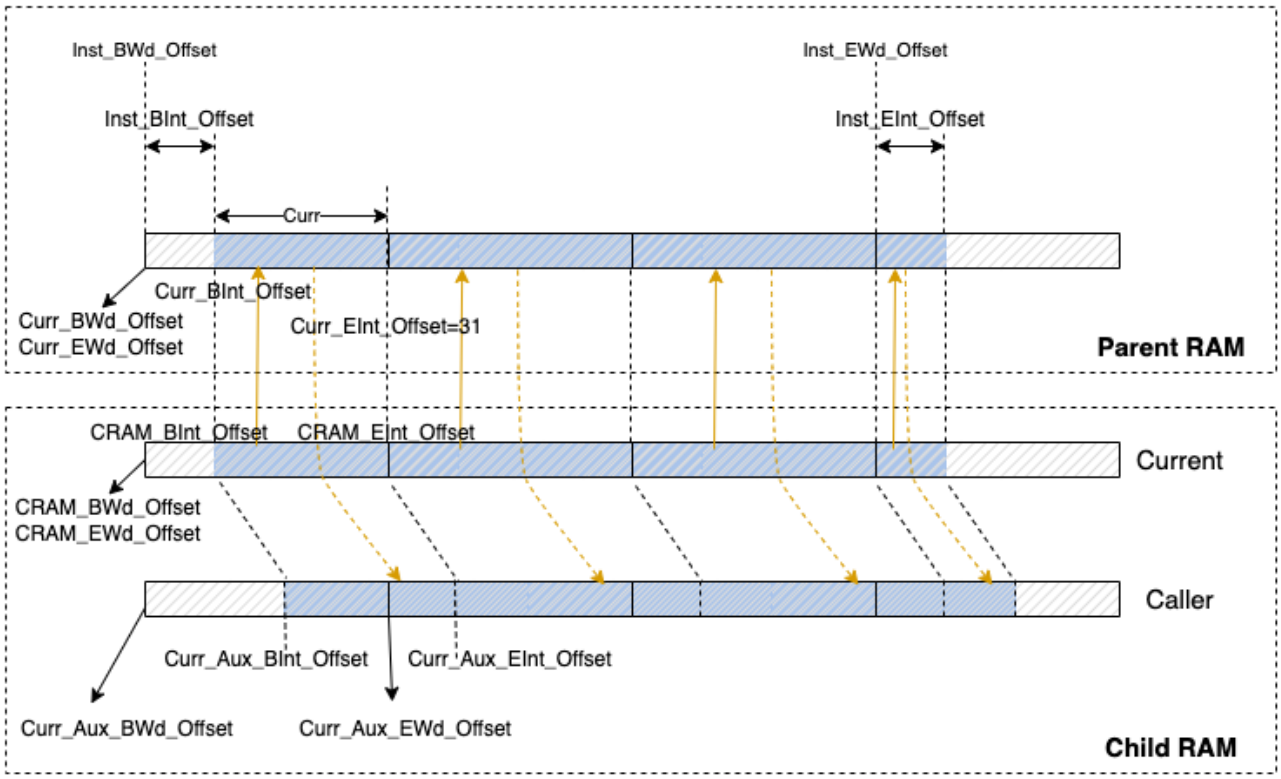
5.2.6 内存大小，Gas消耗

- Curr_Mem_Size: 当前内存EVM字大小
- Max_Mem_Size: 最大内存EVM字大小
- Curr_CALLDATA_Size: 当前CALLDATA的EVM字大小
- Mem_INCREASE_FLAG: 标志内存是否增加
- ΔCurr_Mem_Size: 最大内存与当前内存的差值
- Gas_RAM: RAM指令消耗的gas
- Cost_RAM: 当前步骤消耗的gas
- Carry_Gas_RAM: 用作Gas_RAM计算的进位标志
- $Curr\_Mem^2/512$ ：当前 $Curr\_Mem^2$ 处以512取整
- *COPY_Cost: 外部数据复制操作的消耗

5.2.7 辅助项

- AUX_TRANS

译者注：下图为2.1.2节实例的拓展，添加了些相关列供参考，方便读者理解相关列的含义和约束过程。



5.3 opcode约束

5.3.1 MSTORE/MLOAD具体初始化

If - $(IOPC\_FLAG)_i$ = 0 (MSTORE, MSTORE8, MLOAD, CALLDATALOAD 指令)

1. 取第一个输入(内存地址偏移量)，并且设置指令的起始字地址和字内部偏移量:

$$(Input_1)_1 = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i$$

2. 用Size(MSTORE/MLOAD为31，MSTORE8为1)设置指令终止字地址和字内部偏移量

$$(Input_1)_i + Size_i = 32 * Inst\_EWord_Offset_i + Inst\_EInt\_Offset_i$$

3. If - $(MOPC\_FLAG)_i = 1$ (MSTORE, MSTORE8指令):

   (a) 设置第二个输入(写入内存的字)并且设置指令终止的字偏移量和字内部偏移量：

$$(Input_2)_i = Val(CRAM\_Inst)_i$$

   (b) 设置Child RAM指令来储存第二个输入:

$$CRAM\_Inst_i = 1$$

4. Else If - $(MOPC\_FLAG)_i = 0$ (MLOAD, CALLDATALOAD语句)

   (a) 设置Result的输入(将要从内存中读取的字)并且设置指令终止的字偏移量和字内部偏移量:

$$(Result)_i = Val(RAM\_Inst)_i$$

   (b) 设置Child RAM的语句为读取result的值

$$CRAM\_Inst_i = 0$$

5.3.2 INIT指令初始化

- If - $INIT\_FLAG_i = 1$:

1. 初始化起始字地址和字内部偏移量:

$$32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i = 0$$

2. 初始化终止字地址和字内部偏移量:

$$32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i = 32 * Max\_Mem\_Size_i$$

3. 写入内存:

$$CRAM\_Inst_i = 1$$

4. 设置初始值为0:

$$Val(CRAM\_Inst)_i = 0$$

5.3.3 RETURN具体初始化

If - RET_FLAG = 1:

1. 设置第一个输入(指令起始偏移量)

$$(Input_1)_i = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i$$

2. 设置第二个输入(RETUEN的字长)

$$(Input_1)_i + (Input_2)_i = 32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i$$

3. 初始化辅助项的起始字地址和字内偏移量:

$$Ret\_Offset(SC\_Num)_{i+1} = 32 * Aux\_BWd\_Offset_{i+1} + Aux\_BInt\_Offset_{i+1}$$

4. 用返回值地址和长度初始化辅助项的终止字地址及其字内偏移量:

$$Ret\_Offset(SC\_Num)_{i+1} + Ret\_Len(SC\_Num)_{i+1} = 32 * Aux\_EWd\_Offset_{i+1} + Aux\_EInt\_Offset_{i+1}$$

5. If - $RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1$ ，第一步是读:

$$CRAM\_Inst = 0$$

5.3.4 CALL指令具体初始化

If - $CALL\_FLAG_i = 1$

1. If - 是CALL的第一步: $RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1$: 初始化CALLDATA参数:

   (a) 根据内存地址偏移和长度解析指令:

$$\begin{cases} (Input_2)_{i+2} = 32 * Inst\_BWord\_Offset_i + Inst\_BInt\_Offset_i \\ (Input_2)_{i+2} + (Input_i)_{i+3} = 32 * Inst\_EWord\_Offset_i + Inst\_EInt\_Offset_i \end{cases}$$

   (b) 将RETURN数据的偏移量设置为辅助内存偏移量:

$$\begin{cases} Aux\_BWd\_Offset_i = 0 \\ Aux\_BInt\_Offset_i = 0 \\ (Input_1)_{i+3} = 32 * Aux\_EWord\_Offset_i + Aux\_EInt\_Offset_i \end{cases}$$

2. If - 是CALL的最后一步: $RAM\_STAMP_{i+1} = RAM\_STAMP_i + 1$: 初始化新的CALLDATA大小(在main execution trace约束中实现)

5.3.5 一般值约束

1. If - $RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1$:

   (a) 初始化当前字偏移和字内偏移量

$$\begin{cases} Curr\_BWd\_Offset_i = Inst\_BWd\_Offset_i \\ Curr\_BInt\_Offset_i = Inst\_BInt\_Offset_i \end{cases}$$

5.3.6 更新内存大小

内存大小在指令开始执行的时候更新。在指令执行过程中，内存大小报纸不变。

1. If - $RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1$ AND $END\_FLAG^0 = 0$:

   (a) If - $Mem\_Increase_i = 0$:

   i. 验证内存大小大于当前的最大偏移量:

   $$\Delta Curr\_Mem\_Size_i = Curr\_Mem\_Size_i - Inst\_EWd\_Offset_i$$

   ii. 不增加内存大小:

   $$Curr\_Mem\_Size_i = Curr\_Mem\_Size_{i-1}$$

   (b) Else If - $Mem\_Increase_i = 1$:

   i. 验证内存大小小于当前最大内存偏移量:

   $$\Delta Curr\_Mem\_Size_i = Inst\_EWd\_Offset_i - Curr\_Mem\_Size_i$$

   ii. 增加内存大小:

   $$Curr\_Mem\_Size_i = Inst\_EWd\_Offset_i$$

2. Else If - $RAM\_STAMP_i = RAM\_STAMP_{i-1}$ AND $END\_FLAG^0 = 0$ 保持内存大小不变:

$$\begin{cases} \Delta Curr\_Mem\_Size_i = \Delta Curr\_Mem\_Size_{i-1} \\ Curr\_Mem\_Size_i = Curr\_Mem\_Size_{i-1} \end{cases}$$

3. Else If - $END\_FLAG^0 = 1$: 在main execution trace中处理。

5.3.7 RAM gas消耗:

- 一般消耗计算:

$$\begin{cases} Cost\_RAM_i = *COPY\_Cost_i + Curr\_Mem^2/512_i + 3*Curr\_Mem\_Size_i \\ Gas\_RAM_i = Cost\_RAM_i - Cost\_RAM_{i-1} \\ Curr\_Mem\_Sizei_1^2 = Curr\_Mem\_Size^2/512_i*512 + Carry\_Gas\_RAM \end{cases}$$

- If - $CALLDATA\_COPY\_FLAG = 1$:

$$*COPY\_Cost_i = 3*CALLDATA\_Size_i$$

- Else If - $CALLDATA\_COPY\_FLAG = 0$:

$$*COPY\_Cost_i = 0$$

5.3.8 Child RAM内部偏移量:

这些条件允许根据当前字偏移量调整子 RAM 内部偏移量。

1. 始终增加Child RAM的时间戳

$$CRAM\_STAMP_{i+1} = CRAM\_STAMP + 1$$

2. 约束 $Curr\_[\Phi, B, E]\_[Wd, Int]\_Offset$

(a) 始终约束 $Curr\_BWd\_Offset_i = Curr\_EWd\_Offset_i$

(b) If - $Curr\_EWd_Offset_i = Inst\_EWd\_Offset_i$

$$Curr\_EInt\_Offset_i = Inst\_EInt\_Offset_i$$

(c) Else if - $Curr\_EWd\_Offset_i \neq Inst\_EWd\_Offset_i$

$$Curr\_EInt\_Offset_i = 31$$

(d) If - $Curr\_BWd\_Offset_i = Inst\_BWd\_Offset_i$

$$Curr\_BInt\_Offset_i = Inst\_BInt\_Offset_i$$

(e) Else If - $Curr\_BWd\_Offseti \neq Inst\_BWd\_Offset_i$

$$Curr\_BInt\_Offset_i = 0$$

3. 约束 $Curr\_Aux\_[\Phi, B, E]\_[Wd, Int]\_Offset$ 。 Aux_Offset 的值依照 $Curr\_Offset$ 来约束: 我们不能想约束 $Curr\_Inst$ 一样约束 $Curr\_Aux$ ，因为各自的内部偏移量可能不同。

(a) If - $RAM\_STAMP_i = RAM\_STAMP_{i-1} + 1$ (一般初始化值约束)

i. 初始化当前辅助偏移量

$$\begin{cases} Curr\_Aux\_BWd\_Offset_i = Aux\_BWd\_Offset_i \\ Curr\_Aux\_BInt\_Offset_i = Aux\_BInt\_Offset_i \end{cases}$$

ii. 初始化当前辅助终止偏移量

$$\begin{aligned} 32*Curr\_Aux\_EWd\_Offset_i + Curr\_Aux\_EInt\_Offset_i = \\ 32*Curr\_Aux\_BWd\_Offset_i + Curr\_Aux\_BInt\_Offset_i + \\ (Curr\_EInt\_Offset_i - Curr\_BInt\_Offset_i) \end{aligned}$$

(b) If - $Curr\_BWd\_Offset_i \neq Curr\_BWd\_Offset_{i-1}$ AND $RAM\_STAMP_{i-1} = RAM\_STAMP_i$ (当前指令偏移不变):

i. 保持每个偏移量不变:

$$\begin{cases} Curr\_Aux\_BWd\_Offset_i = Curr\_Aux\_BWd\_Offset_{i-1} \\ Curr\_Aux\_EWd\_Offset_i = Curr\_Aux\_EWd\_Offset_{i-1} \\ Curr\_Aux\_BInt\_Offset_i = Curr\_Aux\_BInt\_Offset_{i-1} \\ Curr\_Aux\_EInt\_Offset_i = Curr\_Aux\_EInt\_Offset_{i-1} \end{cases}$$

5.3.9 值一致性/约束:

1. If - $CALLDATA\_FLAG = 0$:不设置CALLDATA标志:

$$CRAM\_CALLDATA_i = 0$$

2. Else If - $CALLDATA\_FLAG = 1$

   (a) If - $IOPC\_FLAG = 1$:

      i. If - $CRAM\_Inst = 0$: $CRAM\_CALLDATA\_FLAG_i = 0$

      ii. If - $CRAM\_Inst = 1$: $CRAM\_CALLDATA\_FLAG_i = 1$

   (b) Else If - $IOPC\_FLAG = 1$:

$$CRAM\_CALLDATA\_FLAG_i = 1$$

3. If - $MOPC\_FLAG_0 = 1$ (MSTORE/MLOAD):

   (a) 设置当前Child RAM的合约编号为执行的合约编号

$$CRAM\_SC\_Num_i = SC\_Num_i$$

   (b) 设置Child RAM的起止内部偏移量为当前起止内部偏移量

$$CRAM\_BInt\_Offset_i = Curr\_BInt\_Offset_i$$

$$CRAM\_EInt\_Offset_i = Curr\_EInt\_Offset_i$$

   (c) 同样地字的偏移量。

4. Else If - $IOPC\_FLAG_i = 1$

   (a) If - $CRAM\_Inst = 0$ (从当前合约读)

      i. If - $RETURN_FLAG = 1$ OR $CALL\_FLAG = 1$

         A. 设置Child RAM的合约编号为当前的合约编号

$$CRAM\_SC\_Num_i = SC\_Num_i$$

      ii. 设置Child RAM内部偏移量为当前内部偏移量

$$CRAM\_BInt\_Offset_i = Curr\_BInt\_Offset_i$$

$$CRAM\_EInt\_Offset_i = Curr\_EInt\_Offset_i$$

      iii. 同样设置字偏移量

   (b) If - $CRAM\_Inst = 1$ (想caller合约写)

      i. If - $RETURN\_FLAG = 1$

         A. 设置Child RAM的合约编号为caller的合约编号(将数据返回caller合约的内存)

$$CRAM\_SC\_Num_i = Caller\_Num_i$$

      ii. If - $CALL\_FLAG = 1$

A. 设置Child RAM的合约编号为下一个合约编号(写下一个合约的calldata)

$$CRAM\_SC\_Num_i = Next\_SC\_Num_i$$

iii. 设置Child RAM的字内偏移量为返回的字内偏量

$$CRAM\_BInt\_Offset_i = Curr\_Aux\_BInt\_Offset_i$$

$$CRAM\_EInt\_Offset_i = Curr\_Aux\_EInt\_Offset_i$$

iv. 同样设置字的偏移量

v. 约束上一步读的值为当前存储的值:

$$Val(CRAM\_Inst)_i = Val(CRAM\_Inst)_{i-1}$$

5.3.10 过渡约束:

1. If - $RET\_FLAG_i = 1$ AND $RAM\_STAMP_{i+2} = RAM\_STAMP_i$ AND $CRAM\_Inst_i = 0$ (读并返回):

   (a) 下一条指令为向caller内存写，再下一条指令又是读:

   $$CRAM\_Inst_{i+1} = 1$$

   $$CRAM\_Inst_{i+1} = 0$$

   (b) 读写一轮后，字偏移量增长:

   $$Curr\_Wd\_Offset_{i+2} = Curr\_Wd\_Offset_i + 1$$

2. If - $MOPC\_FLAG_i = 1$,AND $RAM\_STAMP_{i+1} = RAM\_STAMP_i$ 增加字偏移量

   $$Curr\_Word\_Offset_{i+1} = Curr\_Word\_Offset_i + 1$$

5.3.11 结束指令约束:

If - $End\_Wd\_Offset_i = Curr\_Wd\_Offset_i$ ，增加RAM时间戳

$$RAM\_STAMP_{i+1} = RAM\_STAMP_i + 1$$

Else If - $End\_Wd\_Offset_i \neq Curr\_Wd\_Offset_i$ ，保持RAM时间戳不变

$$RAM\_STAMP_{i+1} = RAM\_STAMP_i$$

6. Child RAM子模块约束

6.1 架构中的角色

Child RAM模块的引入是为了减轻Parent RAM模块的复杂度以简化系统的约束架构。Child RAM模块被设计用来执行以下操作:

- 初始化所有将被读写的内存单元。

- 对于ROM中加载RAM/calldata，读/写至多两个连续字地址中的至多32个字节。
- 验证ROM中加载的智能合约的RAM/calldata的完整性。

我们将本章节分为两个部分：一方面我们将描述Child RAM模块的读/写相关操作约束，另一方面我们将描述如何验证RAM模块的内存完整性。

6.2 Child RAM读/写操作的约束列

6.2.1 Parent模块包含的列：

这些列用来在Child RAM和Parent RAM之间通信。

- $CRAM\_Inst$ ：将被Child模块执行的指令(可以是读($CRAM\_Inst = 0$)或写($CRAM\_Inst = 1$))
- $INIT\_OPERATION$ ：对给定合约初始化RAM的特殊操作
- $CRAM\_STAMP$ ：Child模块的时间戳
- $CRAM\_BWd\_Offset$ ：读/写字的起始地址
- $CRAM\_EWd\_Offset$ ：读/写字的终止地址
- $CRAM\_BInt\_Offset$ ：读/写起始的字中偏移量
- $CRAM\_EInt\_Offset$ ：读/写终止的字中偏移量
- $Val(CRAM\_Inst)$ ：RAM指令的值
- $CRAM\_SC\_Num$ ：读/写的合约
- $CRAM\_CALLDATA\_FLAG$ ：标识用于决定指令是作用在RAM上还是CALLDATA上
- $INVALID\_INSTRUCTION$ ：无效指令标识，档RAM INIT操作在无效位置被执行是打开

在下面章节中，为了简化Child RAM模块约束的描述，我们将省略CRAM前缀。

6.2.2 CRAM执行相关列

这些列在Child RAM中被使用，用来执行CRAM的读/写操作：

- $Curr\_Wd\_Offset$ ：当前读/写字偏移
- $Curr\_Int\_Offset$ ：当前读/写字内偏移
- $Curr\_Val$ ：当前读/写的值
- $Curr\_Byte$ ：当前读/写的字节
- $Curr\_Carry$ ：当前已经读/写的字节
- $Curr\_Byte\_Num$ ：用来辅助记录当前的字节数
- $Inst\_Range\_Flag$ ：以下情况会被打开

$$2^8 Curr\_Wd\_Offset + Curr\_Int\_Offset \in [2^8 * BWd\_Offset + BInt\_Offset, 2^8 * EWd\_Offset + EInt\_Offset]$$

- $Inst\_Byte$ ：当前指令读/写的字节
- $Inst\_Carry$ ：当前指令已读/写的字节
- $Prev\_Val$ ：前一个被储存的值($Curr\_Wd\_Offset, Curr\_Int\_Offset$ 中)
- $Prev\_Carry$ ：前一个已经储存的值($Curr\_Wd\_Offset, Curr\_Int\_Offset$ 中)
- $Prev\_Byte$ ：前一个被储存的字节($Curr\_Wd\_Offset, Curr\_Int\_Offset$ 中)

6.3 执行约束集

首先要区分两种情况：当前执行是INIT操作或当前指令不是INIT操作。

INIT指令很巧妙，它将当前交易的相关的所有内存单元都初始化。一种方法是将零地址到 MAX_Mem_Size 之间所有字都初始化为零。

- If - $CRAM\_STAMP_i = CRAM\_STAMP_{i-1} + 1$ ：
  a. 初始化 $Curr\_Wd\_Offset$ ：

$$Curr\_Wd\_Offset_i = BWd\_Offset_i$$

  b. 初始化 $Curr\_Int\_Offset$ ：

$$Curr\_Int\_Offset_i = BInt\_Offset_i$$

  c. 初始化 $Curr\_Byte\_Num$ ：

$$Curr\_Byte\_Num_i = 31$$

  d. 初始化 $Curr\_Carry$ ：

$$Curr\_Carry_i = 0$$

  e. 初始化 $Prev\_Carry$ ：

$$Prev\_Carry_i = 0$$

  f. 初始化 $Inst\_Carry$ ：

$$Inst\_Carry_i = 0$$

- If - $INIT\_OPERATION_i = 1$ ：
  a. 写入值:

$$CRAM\_Inst_i = 1$$

  b. 指定存零:

$$Val(CRAM\_Inst)_i = 0$$

  c. If - $Curr\_Wd\_Offset_i = CRAM\_EWd\_Offset$
    i. 增加CRAM时间戳:

$$CRAM\_STAMP_{i+1} = CRAM\_STAMP_{i+1}$$

d. Else If - $Curr\_Wd\_Offset_i \neq CRAM\_EWd\_Offset$:

i. CRAM时间戳不变

$$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i$$

ii. 增加 $Curr\_Wd\_Offset$

$$Curr\_Wd\_Offset_{i+1} = Curr\_Wd\_Offset_i + 1$$

- Else If - $INIT\_OPERATION_i = 0$:

a. If - $BInt\_Offset_i = 0$ AND $EInt\_Offset_i = 31$ AND $CRAM\_BWd\_Offset_i = CRAM\_EWd\_Offset_i$: 快速读/写操作:

i. 设置当前RAM的值为要读/写的值:

$$Curr\_Val_i = Val(CRAM\_Inst)$$

ii. 增加RAM时间戳:

$$CRAM\_STAMP_{i+1} = CRAM\_STAMP_i + 1$$

b. Else If - $BInt\_Offset_i \neq 0$ OR $EInt\_Offset_i \neq 31$ OR $CRAM\_BWd\_Offset_i \neq CRAM\_EWd\_Offset_i$: 慢速读/写操作

i. If - $Curr\_Byte\_Num \neq 0$:

1. 传递已读/写的值:

$$\begin{cases} Curr\_Carry_{i+1} = 2^8 * Curr\_Carry_i + Curr\_Byte_i \\ Prev\_Carry_{i+1} = 2^8 * Prev\_Carry_i + Prev\_Byte_i \end{cases}$$

2. 传递当前值和前一个值:

$$\begin{cases} Curr\_Val_{i+1} = Curr\_Val_i \\ Prev\_Val_{i+1} = Prev\_Val_i \end{cases}$$

3. 减小 $Curr\_Byte\_Num$:

$$Curr\_Byte\_Num_{i+1} = Curr\_Byte\_Num_i - 1$$

ii. If - $Curr\_Byte\_Num = 0$:

1. 检查 $Curr\_Carry$ 和 $Prev\_Carry$ 分别与 $Curr\_Val$ 和 $Prev\_Val$ 一致:

$$\begin{cases} Curr\_Carry_i = Curr\_Val_i \\ Prev\_Carry_i = Prev\_Val_i \end{cases}$$

2. If - $Curr\_Wd_Offset_i = EWd\_Offset_i$:

前进到下一条指令: $CRAM_STAMP_{i+1} = CRAM_STAMP_i + 1$

3. Else If - $Curr\_Wd\_Offset_i = EWd\_Offset_i - 1$:

   a. 移动到下一个字:

   $$Curr\_Wd\_Offset_{i+1} = Curr\_Wd\_Offset_i + 1$$

   b. 重置 $Curr\_Byte\_Num$:

   $$Curr\_Byte\_Num_{i+1} = 31$$

   c. 重置 $Curr\_Carry$ 和 $Prev\_Carry$:

   $$\begin{cases} Curr\_Carry_i = 0 \\ Prev\_Carry_i = 0 \end{cases}$$

iii. If - $32*Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i = 32*BWd\_Offset_i + BInt\_Offset_i$: 设置 $Inst\_Range\_Flag$:

$$Inst\_Range\_Flag_i = 1$$

iv. Else If - $32*Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i = 32*EWd\_Offset_i + EInt\_Offset_i$:

1. 关闭 $Inst\_Range\_Flag$:

$$Inst\_Range\_Flag_{i+1} = 0$$

2. 检查 $Inst\_Carry$ 与 $Val(CRAM\_Inst)$ 一致:

$$Inst\_Carry = Val(CRAM\_Inst)$$

v. Else If - $32*Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i \neq 32*BWd\_Offset_i + BInt\_Offset_i$ AND $32*Curr\_Wd\_Offset_i + Curr\_Int\_Offset_i \neq 32*EWd\_Offset_i + EInt\_Offset_i$ 保持 $Inst\_Range\_Flag$ 不变:

$$Inst\_Range\_Flag_{i+1} = Inst\_Range\_Flag_i$$

vi. If - $Inst\_Range\_Flag_i = 1$:

1. 当前的字节就是指令的字节:

$$Curr\_Byte_i = Inst\_Byte_i$$

2. 移动语句的值:

$$Inst\_Carry_{i+1} = 2^8 * Inst\_Carry_i + Inst\_Byte_i$$

vii. Else If - $Inst\_Range\_Flag_i = 0$:

1. 当前字节为上一个字节:

$$Curr\_Byte_i = Prev\_Byte_i$$

2. 不移动语句值:

$$Inst\_Carry_{i+1} = Inst\_Carry_i$$

6.4 内存一致性列

这些列用来校验从ROM加载进来的每个智能合约的RAM/Calldata的一致性。这些列是Child RAM约束列(6.2节)的空间重排列。准确地说，是6.2节(SC_Num ， $CALLDATA\_FLAG$ ， $Curr\_Wd\_Offset$ ， $CRAM\_STAMP$)按照字典顺序排序。

6.4.1 组合列:

- $\widetilde{SC\_Num}$
- $\widetilde{CALLDATA\_FLAG}$
- $\widetilde{Curr\_Wd\_Offset}$
- $\widetilde{CRAM\_STAMP}$

6.4.2 强制排序的列:

这些列被强制排序，按照上面定义的字典顺序排序:

- $\widetilde{CRAM\_Inst}$: Child RAM的指令
- $\widetilde{INIT\_OPERATION}$: 初始化标志
- $\widetilde{INVALID\_INSTRUCTION}$: 当 $INIT\_OPERATION$ 在错误的位置执行时被打开
- $\widetilde{Curr\_Val}$
- $\widetilde{Prev\_Val}$

6.4.3 范围检查:

为了检查虚拟列 $\widetilde{CRAM\_STAMP}$ 正确增加，必须检查 $\widetilde{CRAM\_STAMP_i}$ 和 $\widetilde{CRAM\_STAMP_{i+1}}$ 的差值。这需要我们引入下面的虚拟列

- $\Delta\widetilde{CRAM\_STAMP}$
- $\Delta\widetilde{CRAM\_STAMP}^i$ ： $\Delta\widetilde{CRAM\_STAMP}$ 的16-bit分解，用来检查 $\Delta\widetilde{CRAM\_STAMP} \in [0, 2^{128}]$ (差额没有溢出)

6.5 内存一致性的约束集

1. If - $\widetilde{SC\_Num_i} = \widetilde{SC\_Num_{i+1}}$

(a) If - $\widetilde{CALLDATA\_FLAG_i} = \widetilde{CALLDATA\_FLAG_{i+1}}$

i. If - $\widetilde{Curr\_Wd\_Offset_i} = \widetilde{Curr\_Wd\_Offset_{i+1}}$

A. If - $\widetilde{INIT\_FLAG_{i+1}} = 1$: 打开 $INVALID\_INSTRUCTION$ 标志，用户在无效的时间戳尝试初始化RAM

$$\widetilde{INVALID\_INSTRUCTION_{i+1}} = 1$$

B. 计算 $\widetilde{\Delta CRAM\_STAMP}$:

$$\widetilde{\Delta CRAM\_STAMP_i} = \widetilde{CRAM\_STAMP_{i+1}} - \widetilde{CRAM\_STAMP_i}$$

C. 约束当前值与下一步的前一步值一致:

$$\widetilde{Prev\_Val_{i+1}} = \widetilde{Curr\_Val_i}$$

D. If - $\widetilde{\Delta CRAM\_STAMP_i} = \widetilde{\Delta CRAM\_STAMP_{i+1}}$: 约束 $Curr\_Val$ 和 $Prev\_Val$ 不变:

$$\begin{cases} \widetilde{Curr\_Val_i} = \widetilde{Curr\_Val_{i+1}} \\ \widetilde{Prev\_Val_i} = \widetilde{Prev\_Val_{i+1}} \end{cases}$$

E. Else If - $\widetilde{\Delta CRAM\_STAMP_i} \neq \widetilde{\Delta CRAM\_STAMP_{i+1}}$ AND $\widetilde{CRAM\_Inst} = 0$ (在两个连续的时间戳读)；约束值一致:

$$\widetilde{Curr\_Val_i} = \widetilde{Curr\_Val_{i+1}}$$

ii. Else If - $\widetilde{Curr\_Wd\_Offset_i} \neq \widetilde{Curr\_Wd\_Offset_{i+1}}$

A. 约束下一个操作是初始化内存单元:

$$\widetilde{INIT\_OPERATION_{i+1}} = 1$$

(b) Else If - $\widetilde{CALLDATA\_FLAG_i} \neq \widetilde{CALLDATA\_FLAG_{i+1}}$

i. 约束下一个操作是初始化内存单元:

$$\widetilde{INIT\_OPERATION_{i+1}} = 1$$

2. Else If - $\widetilde{SC\_Num_i} \neq \widetilde{SC\_Num_{i+1}}$: 约束下一个操作是初始化内存单元:

$$\widetilde{INIT\_OPERATION_{i+1}} = 1$$

参考文献

[1] DR. Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: (2021). https://ethereum.github.io/yellowpaper/paper.pdf

[2] Vitalik Buterin. *An Incomplete Guide to Rollups.* Jan. 2021. URL: https://vitalik.ca/general/2021/01/05/rollup.html

[3] DeGate Team. *An article to understand zkEVM, the key to Ethereum scaling*. Sept. 2021. URL: https://medium.com/degate/an-article-to-understand-zkevm-the-key-to-ethereum-scaling-ff0d83c417cc

[4] *ZK-sync official website.* URL: https://zksync.io/

[5] 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.

[6] *Hermez official website.* URL: https://hermez.io/

[7] *Scroll tech github repository.* URL: https://github.com/scroll-tech/

[8] *Hermez presentation at the EthCC4*. URL: https://youtu.be/17d5DG6L2nw

[9] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: *Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. https://ia.cr/2019/953 2019.

[10] Ariel Gabizon and Zachary J. Williamson. plookup: *A simplified polynomial protocol for lookup tables*. Cryptology ePrint Archive, Report 2020/315. https://ia.cr/2020/315 2020.

[11] Olivier Begassat Alexandre Belling. *Using GKR inside a SNARK to reduce the cost of hash verification down to 3 constraints*. June 2020. url: https://ethresear.ch/t/using-gkr-inside-a-snark-to-reduce-the-cost-of-hash-verification-down-to-3-constraints/7550