1 Processor

处理器模块是 OlaVM 的核心模块之一。它一步一步的执行编译器提供的指令，在执行过程中生成 trace。零知识证明者会使用 trace 生成证明。处理器模块包含了下边这些模块：

- `instructions set architecture(ISA)`
- `prophet executor`
- `builtin`
- `memory`
- `trace generator`

1.1 Instruction Set Architecture

OlaVM 设计指令地址和内存地址在同一空间访问。因为目前虚拟机的空间足够使用，而使用统一空间对于存储进行约束时的列可以更小，减少开销。

1.1.1 register

目前在设计 ZKVM memory hierarchy 的 L0 层时，业内主要有两种模型，一种是寄存器模型，一种是 stack 模型。对于 stack 模型，状态的更新是按照 stack 的先进后出模型实现。对于约束的设计相对简洁。但是缺点是状态的访问不友好，不能随机访问 stack 中的某个地址的数据，对于寄存器模型只需要一条指令完成的读取操作，stack 模型需要多次进行 pop 和 push 的操作。或者为了提高效率，如 miden 的设计中增加 swap 类指令辅助完成 stack 内数据访问。对于 register 模型，可以随机访问任意寄存器。但是缺点是约束模型需要对上下指令进行 copy 约束，证明的开销比 stack 大。

像白皮书设计的那样，OlaVM 选择的是寄存器模型。there are 9 general-purpose registers. 符号表示为：$\mathtt{r}_0 - \mathtt{r}_8$。其中的 $\mathtt{r}_8$ 作为 `fp`(frame pointer)。`fp` 为 $\mathtt{r}_8$ 的别名。

pc 指针加载程序后初始指向地址 0 的指令。之后 pc 指向的地址值随着指令的执行改变。在没执行 jump 指令和 call 指令时，每执行一条指令，如果指令没有使用立即数，pc 寄存器里的地址加 1，否则加 2。

1.1.2 Olavm instructions

为了尽量减少约束的 degree。OlaVM 选用了精简的指令集。

Olavm 采用 Word(位宽) = 64bits 对指令编码。一条指令通过一个 opcode 和最多三个操作数来编码，操作数可以是寄存器名称，也可以是立即数，一个指令编码由下面的 8 个 fields 组成。

- Field1: 字段名称 NULL。占 1 bit，不使用。
- Field2: 字段名称 OP1 immediate flag(OP1_IMM, imm)。包含 1 bit, 如果是 0，表示 A 是一个寄存器名称，如果是 1 表示这是一个立即数。
- Field3: 字段名称 operation0 register(OP0_REG, reg_src1)。指示使用源寄存器 0 的序号，包含了 9 bits，分别指示 0-8 共 9 个寄存器，bit 值为 1 代表本指令使用该序号 OP0 寄存器，为 0 则不使用。
- Field4: 字段名称 operation1 register(OP1_REG, reg_src2)。指示使用源寄存器 1 的序号，包含了 9 bits，分别指示 0-8 共 9 个寄存器，bit 值为 1 代表本指令使用该序号 OP1 寄存器，为 0 则不使用。
- Field5: 字段名称 destination register(DST_REG, reg_dst)。指示目标寄存器的序号, 包含了 9 bits，分别指示 0-8 共 9 个寄存器，bit 值置为 1 代表本指令使用该序号 DST 寄存器，为 0 则不使用。

- Field6: 字段名称 opcode selector(opcode_sel)。指示指令的 opcode，包含了 19 bits，分别指示 ISA 支持的 19 种指令，bit 值置为 1 代表本指令使用该 opcode，为 0 则不使用。
- Field7: 字段名称 padding bits(paddings)。这个 filed 一共有 16 bits。所有 bit 使用 0 进行填充，后续可以扩充定义别的字段。
- Field8: 字段名称字段名称 immediate data(immediate)。这个 field 会根据 field2 字段决定是否存在，如果 filed2 值为 1，这个字段将存在一个立即数值，这个字段是 2 Word。

OlaVM 的指令集定义如下表, A 表示可能是立即数或者寄存器：

Type	*Instruction*	*Operands*	*Description*	*flag*
Field Operation	ADD	ri rj A	Compute rj + A and store the result in ri	
	MUL	ri rj A	Compute rj * A and store the the result in ri	
Cmp	EQ	ri A	Equality comparison	ri = A
	NEQ	rj A	check whether rj is not equal A and store the the result in flag	rj != A
	ASSERT	ri A	Assert ri == A and vm will hang up if assertion fail	
Move	MOV	ri A	Copy the data of A to ri	
Flow	JMP	A	Set pc equal A	
	END		only appears at the end of the program	
	CJMP	A	If flag = 1, set pc equal A, else increment pc as usual	
	CALL	A	The Call instruction consists of the following steps 1. store return pc to the frame address [fp-1]. 2. jump to the address A	
	RET		The Ret instruction consists of the following two steps 1. use the memory address stored in the fp register to find the returned pc and jump to the location of the returned pc. 2. update the fp register to the fp before the call	
RAM	MLOAD	ri [A]	Read the value from memory [A] and store it into ri. A can be a intermediate data or register value	
	MSTORE	[A] rj	Read the value from register rj and store it into memory [A]. A can be a intermediate data or register value	
BUILTIN	RANGE CHECK	rj	range check the value in rj	
	AND	ri rj A	Compute rj and A and store the the result in ri	
	OR	ri rj A	Compute rj or A and store the the result in ri	
	XOR	ri rj A	Compute rj xor A and store the the result in ri	
	NOT	ri A	Compute not A and store the the result in ri	
	GTE	rj A	check whether rj is great than or equal to A and store the the result in flag	rj >= A

表 1: Instruction set

由上述定义可得：指令编码在不存在立即数的情况下占用 2W 空间，存在立即数的情况下占用 4W 空间。operand 的分配规则：当且仅当 reg 变化时才能放到 DST_REG，如果只有一个源 operand 的情况存放在 OP1_REG 而不是放在 OP0_REG。编码格式见下表：

NULL	OP1_IMM	OP0_R8	OP0_R7	OP0_R6	OP0_R5	OP0_R4	OP0_R3	OP0_R2	OP0_R1	OP0_R0	OP1_R8	OP1_R7	OP1_R6	OP1_R5	OP1_R4
OP1_R3	OP1_R2	OP1_R1	OP1_R0	DST_R8	DST_R7	DST_R6	DST_R5	DST_R4	DST_R3	DST_R2	DST_R1	DST_R0	ADD	MUL	EQ
ASSERT	MOV	JMP	CJMP	CALL	RET	MLOAD	MSTORE	END	RANGE_CHECK	AND	OR	XOR	NOT	NEQ	GTE
padding															
immediate data 48-63 bits															
immediate data 32-47 bits															
immediate data 16-31 bits															
immediate data 0-15 bits															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

表 2: OlaVM 指令编码格式

执行指令的状态转换伪代码如下：

```
new_op:
opcode = opcode_list[instuction[pc].opcode_sel]
imm = instuction[pc].imm
```

```
ri = reg_dst
rj = reg_src1

if imm == 1:
    next = pc + 2
    A = immediate
else
    next = pc + 1
    A = reg_src2
endif

switch opcode:
    ADD:
        ri = rj + A
        break
    MUL:
        ri = rj * A
        break
    EQ:
        if ri == A:
            flag = 1
        else
            flag = 0
        break
    MOV:
        ri = A
        break
    JMP:
        pc = A
        goto new_op
    CJMP:
        if flag == 1:
            pc = A
        goto new_op
    CALL:
        [fp-1] = next
        pc = A
        goto new_op
    RET:
        fp = [fp-2]
        pc = [fp-1]
        goto new_op
    MLOAD:
        ri = [A]
        break
    MSTORE:
        [A] = ri
        break
    ASSERT:
        if ri != A:
```

```
            panic("ASSERT failed")
    RANGE_CHECK:
        insert ri value to rangetable trace table
        break
    AND:
        ri = rj & A
        break
    OR:
        ri = rj | A
        break
    XOR:
        ri = rj xor A
        break
    NOT:
        ri = not
        break
    NEQ:
        flag = ri != A
        break
    GTE:
        flag = ri >= A
        break
pc = next
goto new_op
```

下边是一段算数运算的指令编码的例子，l1 和 l2 实现了指令 *mov r0 0x8*，l3 和 l4 实现了指令 *mov r1 0x2*，l5 和 l6 实现了指令 *mov r2 0x3*，l7 实现了指令 *add r3 r0 r1*，l8 实现了指令 *mul r4 r3 r2*，l9 实现了 builtin*range_check r4*，计算结果 $r0 = 8, r1 = 2, r2 = 3, r3 = 10, r4 = 30$。

```
  l1:  0x4000000840000000
  l2:  0x8
  l3:  0x4000001040000000
  l4:  0x2
  l5:  0x4000002040000000
  l6:  0x3
  l7:  0x0020204400000000
  l8:  0x0100408200000000
  l9:  0x0001000000400000
```

1.1.3 procedure call standard

fp 寄存器存加载程序后初始化指向 frame 堆栈的首地址。之后在执行 call 指令时地址会增加。在执行 ret 指令时 fp 寄存器指向地址会回退。使用指令 call 调用函数，fp 指向新的 frame 以后，函数返回的 pc 地址放在：`[fp-1]`，函数调用前的 fp 指向的地址放在：`[fp-2]`，前 4 个入参依次放在 $\mathtt{r}_0 - \mathtt{r}_3$ 四个寄存器里。从第 5 个入参开始，依次递减放在：`[fp-3]`，`[fp-4]` $\cdots$。函数内部局部变量从 `[fp]` 开始，fp 地址递增存放。返回值存放在 $\mathtt{r}_0$ 中，如果返回值不是一个域元素，则需要通过返回数据的内存指针来实现。

比如：调用函数 `foo(a: felt, b: felt, c: felt, d: felt, e: felt)`，入参：`a=0x1, b=0x2, c=0x3, d=0x4, e=0x5`。

```
func foo(a: felt, b: felt, c: felt, d: felt, e: felt)
    let sum = 0;
    sum = a+b;
    sum = sum * c;
    sum = sum + d + e
    return ;
```

则函数调用前后的 fp，pc 和内存状态如下图，其中黄色表示内存地址，红色表示指令地址，蓝色表示寄存器：

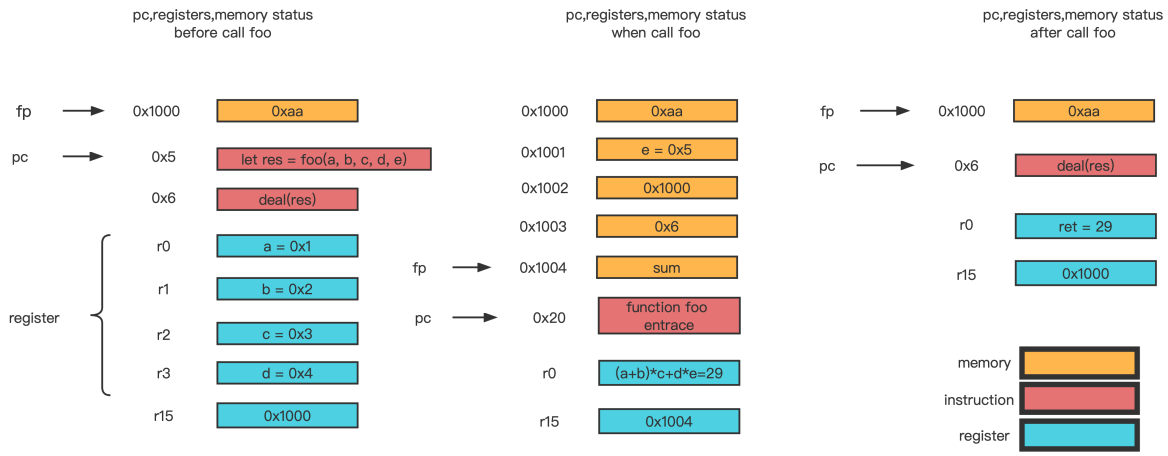

图 1: OlaVM 函数调用模型

1.2 Prophet Executor

prophet 用 %{ ... %} 来表示。prophet 被执行不影响 pc、fp 寄存器存储的地址。但是 h= 会更改 vm 保存在内存中的变量值。下边是一个 prophet 的例子：

```
fn main() {
    let a = 3
    // Use custom prophet to print the value of a
    // local 用来获取函数中的变量，用来进行和vm通信
    %{print(local.a)%}
}
```

虚拟机如何处理 prophets

为了能够正确处理 prophets，虚拟机会实现两个方法：‘binding_prophet‘ 和 ‘execute_prophet‘ 。

‘binding_prophet‘ 主要作用是将用户合约中写的 prophets 和虚拟机中采用 rust 实现的对应方法进行绑定，并且获取对应的 prophet 参数进行处理。

‘execute_prophet‘ 用于执行用户合约中编写的写的 prophets，prophets 的执行会优先于虚拟机指令的执行。

虚拟机执行编译后的程序的过程如下：下图是虚拟机执行编译后的程序的示意图2：

binding_prophet

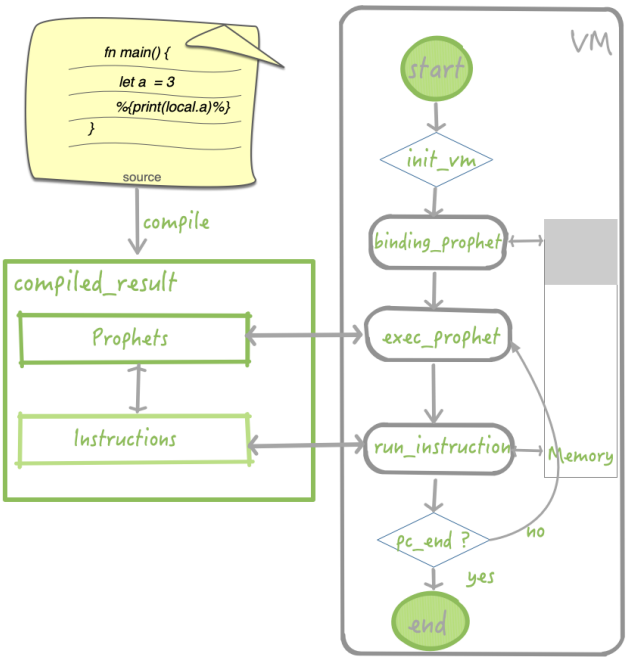

图 2: OlaVM prophets 执行流程图

该方法在执行前调用，该方法有以下参数：

- String 形式的 prophet code
- prophet 中定义的变量名称与对应合约编译后 json 中的引用 id 号的 HashMap
- 应用 id 号对应的 ProphetReference 的 HashMap

该方法返回一个 dynamic structure，后续会被 'execute_prophet' 方法使用

execute_prophet

当有提示要执行时，这个方法会在每个虚拟机步骤的开始被调用。它接收由 binding_prophet 创建的动态结构以及程序常量和一组包含对虚拟机内部的有限访问的代理。

- binding_prophet 返回的动态结构, 包含变量以及 code 以及内存索引内容
- vm 的不可变引用，vm 这个结构包含了内存段管理器和运行上下文的可变引用等内容

变量值如何在 prophets 和 vm 中传递

每个程序中的变量的地址和值都可以通过 ProphetReference 结构获取数据，prophet processor 提供了访问变量值 (value) 和地址 (ptr) 的辅助函数

1.3 builtin

OlaVM 使用 builtin 来实现 range check, hash 等功能。VM 实现 builtin 功能需要编译器提供需要支持的 builtin 类型，编码定义在表 1的 builtin 模块。

1.3.1 range check

Range check 需要验证指定寄存器内存的值是否为 u32 类型。VM 读到 builtin 的 range check 指令，会将要 check 的寄存器的值放入到 builtin 的 range check 表。

1.3.2 bitwise

VM 支持对于两个寄存器内的 u32 类型的数进行与，或，异或和取反操作。VM 读到 builtin 的 bitwise 指令, 将 bitwise 操作的寄存器的值放入到 builtin 的 bitwise 表。将进行 bitwise 操作的两个输入数放入 builtin 的 range check 表。

1.3.3 comparison

VM 支持对于两个寄存器内的 u32 类型的数进行大于等于和不等于操作。VM 读到 builtin 的 comparison 指令, 将 comparison 操作的寄存器的值放入到 builtin 的 comparison 表。将进行 comparison 操作的两个输入数放入 builtin 的 range check 表。将 gte 的 diff 字段的数也要放入 builtin 的 range check 表。该 diff 为两个输入的差的绝对值。

1.4 Memory

OlaVM 的内存设计为可随机访问为的读写模式。对其约束也是按照该模式进行设计。读写内存地址的值通过将地址存入寄存器，之后通过 $[r_n]$ 来进行读写操作。Memory 的结构采用 B-Tree 模式。可以通过地址随机寻址，读取和写入数据到 Memory。B-Tree 的优点插入数据时按照地址进行了排序。因此在下边两个场景下提高了效率。

1. 随机读内存地址数据数据, 根据内存地址通过 B-tree 数据结构快速找到地址下存的数据。
2. 根据内存地址和 clk 排序生成 memory trace table 时，顺序在插入时已排好，可以直接顺序处理，不用再次排序。

下图是内存数据结构的示意图3：

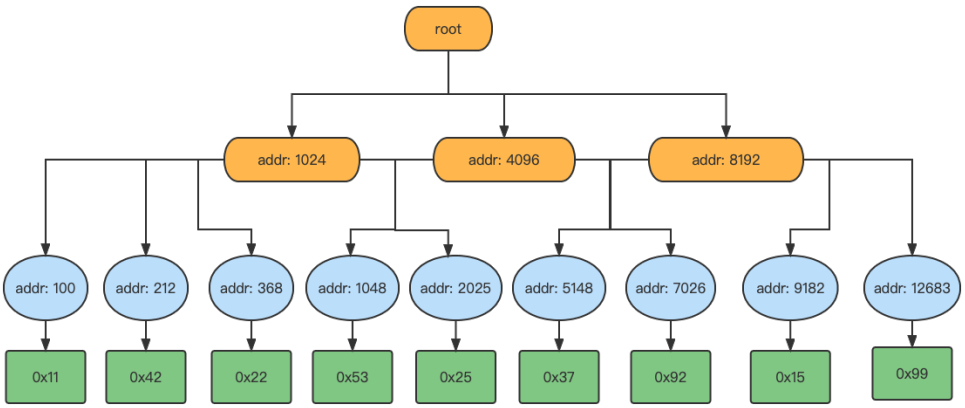

图 3: OlaVM memory 存储数据结构

1.5 Trace Generator

当 OlaVM 在执行编译好的指令时，它会在每执行一条指令时，将对应的内存状态和寄存器状态保存在一张 trace 表中。

1.5.1 Main Trace

trac 主表的列分为 5 部分 context, instruction, register selectors, opcode selectors, builtin selectors。

context columns

主 trace 的 context 相关列一共 12 列。其中 VM 的时钟 clk 和取指令地址 pc 各占一列。flag 状态标志占一列。寄存器状态从 $\mathtt{r}_0 - \mathtt{r}_8$ 一共 9 列。注意 registers 的 9 列存的是执行前的状态。

- `clk`：VM 的 clock 的缩写。操作内存时的 VM 时钟。在 addr 相同的情况下，clk 是单调增的。
- `pc`：VM 操作内存时的指令地址。
- `flag`：VM 操作内存时的指令地址。
- `registers`：寄存器状态从 $\mathtt{r}_0 - \mathtt{r}_8$ 一共 9 列。

context 相关列的定义见下表。

clk	pc	flag	r0	· · ·	r8

表 3: OlaVM 的 main trace 表 context 相关列结构

instruction columns

主 trace 的 instruction 相关列一共 4 列。定义如下：

- `instruction`：指令的编码，具体定义见表 2。
- `op1_imm`：指令是否带立即数。1 为带，0 为不带。
- `opcode`：指令的 opcode_sel 指示。具体见下边列表 4。
- `immediate_value`：如果 op1_imm 为 1，则存放指令带的立即数，否则为 0。

ADD	MUL	EQ	ASSERT	MOV	JMP	CJMP	CALL	RET	MLOAD
2^{34}	2^{33}	2^{32}	2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}	2^{25}
MSTORE	END	RANGE_CHECK	AND	OR	XOR	NOT	NEQ	GTE	
2^{24}	2^{23}	2^{22}	2^{21}	2^{20}	2^{19}	2^{18}	2^{17}	2^{16}	

表 4: OlaVM 的 main trace 表 opcode value 列表

register selector columns

主 trace 的 register selector 相关列一共 31 列。定义如下：

- `op0`：操作数 0 的值。
- `op1`：操作数 1 的值。
- `dst`：目的寄存器的值。
- `aux0`：辅助列，个别指令会用到。
- `op0 regster selectors`：指示 op0 使用通用寄存器 op0_reg(参与 op0 列与指示的 context 里 op0_reg 的 copy 约束)。
- `op1 regster selectors`：指示 op1 使用通用寄存器 op1_reg(参与 op1 列与指示的 context 里 op1_reg 的 copy 约束)。
- `destination regster selectors`：指示 dst 使用通用寄存器 dst_reg(参与 dst 列与指示的 context 里 dst_reg 的 copy 约束)。

op0	op1	dst	aux0	sel_op0_r0	···	sel_op0_r8	sel_op1_r0	···	sel_op1_r8	sel_dst_r0	···	sel_dst_r8

表 5: OlaVM 的 main trace 表 register selector 相关列结构

register selector 相关列的定义见下表。

opcode selector columns

主 trace 的 opcode selector 相关列一共 12 列。指示本行指令使用的 opcode。相关列的定义见下表。

sel_add	sel_mul	sel_eq	sel_assert	sel_mov	sel_jmp	sel_cjmp	sel_call	sel_ret	sel_mload	sel_mstore	sel_end

表 6: OlaVM 的 main trace 表 opcode selector 相关列结构

builtin selector columns

主 trace 的 builtin selector 相关列一共 7 列。指示本行指令使用的 builtin。相关列的定义见下表。

sel_range_check	sel_and	sel_or	sel_xor	sel_not	sel_neq	sel_gte

表 7: OlaVM 的 main trace 表 builtin selector 相关列结构

1.5.2 Memory Trace

OlaVM 的内存 trace 由 17 列组成

- `mem_is_rw`：内存地址是读写还是 write_once。
- `addr`：内存地址 address 的缩写。在表中是单调增的。
- `clk`：VM 的 clock 的缩写。操作内存时的 VM 时钟。在 addr 相同的情况下，clk 是单调增的。
- `op`：本次操作内存的指令编号。
- `rw`：VM 操作内存时的操作状态，目前状态有两种：0 代表读和 1 代表写。
- `value`：本次被操作内存的存储值。
- `addr_diff`：本行和上一行的地址差。
- `addr_diff_inv`：本行和上一行的地址差的逆。
- `clk_diff`：本行和上一行的时钟差。
- `addr_diff_cond`：读写内存段为 0，其他类型的内存段，为该内存段最大地址与当前地址的差。
- `filter_for_main`：标记是否可被 main trace 访问。
- `rw_addr_inchanged`：读写地址段本行和上一行的地址是否变化，不变为 1，改变为 0。
- `region_prophet`：标记 prophet 段。
- `region_poseidon`：标记 poseidon 段。
- `region_ecdsa`：标记 ecdsa 段。
- `rc_value`：本次内存操作需要 range check 的 diff 值，addr_diff 或者 clk_diff。
- `filter_for_rc`：标记是否可被 range check trace 访问。

1.5.3 Range Check trace

OlaVM 的 range check trace 由 6 列组成

- `cpu_filter`：cpu table 过滤列。

- `memory_filter`：内存 table 过滤列。
- `cmp_filter`：比较 builtin 过滤列。
- `value`：要 range check 的值。
- `limb_lo`：check value 的第 16 位。
- `limb_hi`：check value 的第 16 位。

格式见下表 8。

cpu_filter	memory_filter	cmp_filter	value	limb_lo	limb_hi

表 8: OlaVM 的 range check 的 trace 表结构

1.5.4 Bitwise trace

OlaVM 的 bitwise trace 由 16 列组成。同时 bitwise 操作的两个输入数会放入 builtin 的 range check 表。

- `tag`：选择 bitwise 的操作类型，0, 1, 2 分别对应 AND, OR, XOR。
- `op1_value`：bitwise 操作的第一个输入值。
- `op2_value`：bitwise 操作的第二个输入值。
- `res_value`：bitwise 操作的结果值。
- `limb op1_0`：op1 value 的第 0-7 位。
- `limb op1_1`：op1 value 的第 8-15 位。
- `limb op1_2`：op1 value 的第 16-23 位。
- `limb op1_3`：op1 value 的第 24-31 位。
- `limb op2_0`：op2 value 的第 0-7 位。
- `limb op2_1`：op2 value 的第 8-15 位。
- `limb op2_2`：op2 value 的第 16-23 位。
- `limb op2_3`：op2 value 的第 24-31 位。
- `limb res_0`：res value 的第 0-7 位。
- `limb res_1`：res value 的第 8-15 位。
- `limb res_2`：res value 的第 16-23 位。
- `limb res_3`：res value 的第 24-31 位。

格式见下表 9。

tag	op1_value	op2_value	res_value	op1_0	op1_1	op1_2	op1_3	op2_0	op2_1
op2_2	op2_3	res_0	res_1	res_2	res_3				

表 9: OlaVM 的 bitwise 的 trace 表结构

1.5.5 Comparison trace

OlaVM 的 comparison trace 由 6 列组成。同时 comparison 操作的两个输入数会放入 builtin 的 range check 表。

- `op1 value`：比较操作的第一个输入值。

- `op2 value`：比较操作的第二个输入值。
- `diff`：比较操作的两个输入的差，sel=1 时为差的绝对值。
- `diff_limb_lo`：比较操作的两个输入的差的低 32 位。
- `diff_limb_lo`：比较操作的两个输入的差的高 32 位。
- `filter_looked_for_range_check`：比较操作的 filter。

格式见下表 10。

op1 value	op2 value	diff	diff_lo	diff_hi	filter_looked_for_range_check

表 10: OlaVM 的 comparison 的 trace 表结构

1.6 Stdlib

VM 支持 stdlib 库的执行。比如无符号数的算术计算，常用 hash 函数 (SHA256) 等。

1.6.1 u32 arithmatic

OlaVM 支持 u32 数据类型的算法计算。包括下边操作：

- `u32_add`
- `u32_sub`
- `u32_mul`
- `u32_div`

`u32 ADD`

u32 的加法算法伪代码如下：

```
func u32_add(data_src_1: Felt, data_src_2: Felt) {
    MOV r_src1 data_src_1
    MOV r_src2 data_src_2

    range_check(r_src1, MAX_u32)
    range_check(r_src2, MAX_u32)

    ADD r_dst r_src1 r_src2
    %{
        (r_dst_hi, r_dst_lo) = split64(r_dst)
    %}


    range_check(r_dst_lo, MAX_u32)
    range_check(r_dst_hi, MAX_u32)

    MUL r_assert r_dst_hi 2^32
    ADD r_assert r_assert r_dst_low
    EQ r_dst r_assert

    return (r_dst_hi, r_dst_lo)
```

```
}
```

u32 SUB

u32 的减法算法伪代码如下：

```
func u32_sub(data_src_1: Felt, data_src_2: Felt) {
    MOV r_src1 data_src_1
    MOV r_src2 data_src_2

    range_check(r_src1, MAX_u32)
    range_check(r_src2, MAX_u32)

    local borrow_flag
    local difference
    %{
        if r_src1 > r_src2 {
            borrow_flag = 0
        } else {
            borrow_flag = 2^32
        }
        difference = borrow_flag + r_src1 - r_src2
    %}

    ADD r_assert1 r_src2 difference
    ADD r_assert2 r_src1 borrow_flag
    EQ r_assert1 r_assert2

    return (difference, borrow_flag)
}
```

u32 MUL

u32 的乘法算法伪代码如下：

```
func u32_mul(data_src_1: Felt, data_src_2: Felt) {
    MOV r_src1 data_src_1
    MOV r_src2 data_src_2

    range_check(r_src1, MAX_u32)
    range_check(r_src2, MAX_u32)

    MUL r_dst r_src1 r_src2
    %{
        (r_dst_hi, r_dst_lo) = split64(r_dst)
    %}

    range_check(r_dst_lo, MAX_u32)
    range_check(r_dst_hi, MAX_u32)

    MUL r_assert r_dst_hi 2^32
    ADD r_assert r_assert r_dst_low
    EQ r_dst r_assert
```

```
    return (r_dst_hi, r_dst_lo)
}
```

u32 DIV

u32 的除法算法伪代码如下:

```
func u32_div(data_src_1: Felt, data_src_2: Felt) {
    MOV r_src1 data_src_1
    MOV r_src2 data_src_2

    range_check(r_src1, MAX_u32)
    range_check(r_src2, MAX_u32)

    local quotient
    local remainder
    %{
        quotient = r_src1 / r_src2
        remainder = r_src1 % r_src2
    %}

    MUL r_assert r_src2 quotient
    ADD r_assert r_assert remainder
    EQ r_src1 r_assert

    return (r_dst_hi, r_dst_lo)
}
```

1.6.2 u64 arithmatic

OlaVM 支持 u64 数据类型的算法计算。包括下边操作:

- u64_add
- u64_sub
- u64_mul
- u64_div

u64 ADD

u64 的加法算法伪代码如下:

```
func u64_add(data_src_1_lo: Felt, data_src_1_hi: Felt, data_src_2_lo: Felt, data_src_2_hi
    : Felt) {
    range_check(r0, MAX_u32)
    range_check(r1, MAX_u32)

    ADD r4 r0 r1
    %{
        (mem[addr_hi0], mem[addr_lo0]) = split64(r4)
    %}

    mload r5, mem[addr_lo0]
```

```
    mload r6, mem[addr_hi0]

    range_check(r5, MAX_u32)
    range_check(r6, MAX_u32)


    MUL r7 r6 2^32
    ADD r7 r7 r0
    ASSERT r4 r7

    range_check(r2, MAX_u32)
    range_check(r3, MAX_u32)

    ADD r4 r2 r3
    ADD r4 r4 r6

    %{
        (mem[addr_hi1], mem[addr_lo1]) = split64(r4)
    %}

    mload r0, mem[addr_lo1]
    mload r1, mem[addr_hi1]

    range_check(r0, MAX_u32)
    range_check(r1, MAX_u32)

    MUL r1 r1 2^32
    ADD r1 r1 r0
    ASSERT r4 r1

    return (mem[addr_lo0], mem[addr_lo1], mem[addr_hi1])
}
```

u64 MUL

u64 的乘法算法伪代码如下：

```
func u64_mul(data_src_1_lo: Felt, data_src_1_hi: Felt, data_src_2_lo: Felt, data_src_2_hi
    : Felt) {
    mload r0 [fp-3]
    mload r1 [fp-5]

    range_check(r0, MAX_u32)
    range_check(r1, MAX_u32)

    MUL r2 r0 r1
    %{
        (mem[addr_hi0], mem[addr_lo0]) = split64(r2)
    %}

    mload r0, mem[addr_lo0]
    mload r1, mem[addr_hi0]
```

```
    range_check(r0, MAX_u32)
    range_check(r1, MAX_u32)


    MUL r3 r1 2^32
    ADD r3 r3 r0
    ASSERT r2 r3

    mload r4 [fp-4]
    mload r5 [fp-6]

    range_check(r4, MAX_u32)
    range_check(r5, MAX_u32)

    MUL r2 r4 r5
    ADD r2 r2 r1

    %{
        (mem[addr_hi1], mem[addr_lo1]) = split64(r2)
    %}

    mload r6, mem[addr_lo1]
    mload r7, mem[addr_hi1]

    range_check(r6, MAX_u32)
    range_check(r7, MAX_u32)

    MUL r7 r7 2^32
    ADD r7 r7 r6
    ASSERT r2 r7

    return (mem[addr_lo0], mem[addr_lo1], mem[addr_hi1])
}
```