Cairo

原文： Cairo – a Turing-complete STARK-friendly CPU architecture

参考：Specification for cairo

翻译： xb

证明系统允许一方向另外一方证明一个确定的论述是正确的。现有的实用性证明系统都要求把论述用有限域上的多项式等式来表示；这使得一个想要被证明或者验证的论述的表示过程变得相当复杂，因为它需要对每个论述引出一系列新的等式。

针对这个问题，已经有不同的处理方法，比如示例【1】。

我们推出了Cairo，一个实用的，高效的图灵完备的STARK友好的CPU架构。针对“在这个架构上执行的程序是有效的”论述，我们定义了**一套独立的多项式等式**。给定一个想要被证明的论述，Cairo支持撰写一个程序来表示这个论述，而不是设计一套多项式等式。

1. 介绍

1.1 背景

Babai, Fortnow和Lund【8】第一次把交互式证明系统应用在了扩容场景上。通常来讲，这样的系统存在两个角色，prover和verifier；他们之间执行一个协议，即prover让verifier相信一个确定的论述是正确的。论述的普遍形式是“我知道一个计算过程，给定一个确定的输入，输出一个确定的结果”，其中prover和verifier都知道计算过程和结果。一个最原始的方法是，prover把对应的输入发送给verifier，然后verifier重复执行计算过程，比对最终结果；这个办法存在两个缺点：（1）verifier知道了输入信息，暴露了**隐私**；（2）verifier需要重复执行计算过程，**不高效**。计算完整性的密码学证明系统是一个可以解决上述问题的协议：（1）针对隐私，引入了**零知识**概念；（2）支持**简洁**验证，比重复执行的性能提升指数级。

本文通过引入Cairo来解决被证明计算（论述）形式化的挑战，Cairo是一种允许以计算机程序的形式描述计算，然后生成该计算的完整性证明的体系结构。Cairo架构的设计是为了：（1）方便书写和阅读要被证明的论述；（2）高效的证明，比如基于STARK的证明系统【10】。

在许多现存的实用型证明系统中，你不得不把要被证明的计算用在有限域上的多项式等式表示；这个过程叫做“算术化”（arithmetization），它第一次被用在交互式证明的过程中【19】；比如有**算术电路**（arithmetic circuits），**QSP**（Quadratic Span Programs【15】），**AIRs**（Algebraic Intermediate Representations【10 P.14】）。

这些需求使得将这个证明系统用在实际应用中的工作变得相当复杂；另外，在算术化的过程中，其中的一些方法会在结果中出现不必要的计算（参考下面的分支和循环示例）。

通过一些示例来展示一下算术化过程应该是什么样子。从一个最简单的示例开始：判断 $x \neq y$ ；注意，多项式等式一般是 $p = 0$ 的形式，其中 p 代表着多项式；因此，上述的断言就可以转换成 $\exists a : a \cdot (x - y) = 1$ （增加一个额外的变量 a ）。稍微更复杂的示例，计算基于 2^{64} 的模加运算，它也可以转换成多项式等式的形式，增加64个二进制变量，然后计算他们的总和。

一个更复杂的示例就是在计算中需要处理**分支**的情况（比如当 $x = y$ 时，是一种计算方式；否则就是另外一种）和**循环**的情况（直到满足一个条件前，一直做重复的计算）。其中处理分支的办法就是把两个分支都转化成多项式的形式，然后通过“选择器（selectors）”，根据实际计算过程中的值，来选择执行对应的分支结果（比如等式 $z = (1 - b) \cdot x + b \cdot y$ 确保了，当 $b = 0$ 时，$z = x$ ；当 $b = 1$ 时， $z = y$ ）；循环的处理方法是，把支持迭代的次数上限用一个常量 B 表示，并精确执行循环体 B 次，如果在其中的某一步满足了条件，则下一迭代将仅仅是传递结果，直到循环结束。值得注意的是，它们都引入了一些不必要的计算，比如分支情况下，要执行所有分支；循环情况下，要执行 B 次迭代，即使循环提前结束。

解决计算的形式化过程的一个办法就是设计一个编译器（compiler）- 一个计算机程序，以代码作为输入，输出代码要执行的计算对应的多项式等式。遵循这种方法的系统包括ZKDPL【20】，Pinocchio【22】，TinyRAM for SNARKS【11】，STARKs和xJsnarks【18】，它们让算术化过程变得简单，但同时其结果也有一些缺点，例如前面提到的，执行不必要代码的低效率和限定循环中迭代次数的必要性。

另一种方法的动机来自CPUs的发明和冯·诺伊曼体系结构：人们可以设计一套多项式方程的通用系统来表示为某些**固定指令集编写**的任意计算机程序的执行。在预处理SNARKs的情境下，该方法被应用在vnTinyRAM系统中【13】。

1.2 我们的贡献

我们推出了**Cairo**，一个有效且实用的冯·诺伊曼体系结构，结合STARK证明系统，为计算的完整性生成证明；因此，它是第一个基于STARK的冯·诺伊曼体系结构，**Cairo**的主要优点有：

高效：Cairo的指令集使得相应的AIR更加高效，比如【13】的构造每个循环大约需要1000个变量，而基于Cairo的AIR模式，只需要51个变量（参考第9章节），此外，我们还提出了内置的概念(第2.8节和第7节)，这使得执行预定义操作的开销可以忽略不计（例如应用一个加密哈希函数）。

实用： Cairo支持条件分支，内存，函数调用和递归。

生产级：Cairo已经服务了多个运行在以太坊的多个密码学系统，对Cairo程序的证明生成比较频繁而且可以被部署在主网上的合约验证，更多信息可以阅读【2】。

以下论文中呈现的概念，对Cairo的性能至关重要：

代数的RISC：Cairo使用了小而简单，但又具有丰富表现力的指令集；所有的指令都可以用15个flag和3个整数来编码，参考章节**2.3.2**和**4.5**

不确定连续只读随机访问内存：不同于传统的内存读写模型，Cairo采用了一个独特的内存模型（章节2.6），其存在更多限制，比如所有存储单元的值都由prover选择，并且在执行代码时不会更改；还有一个限制使得AIR实现非常高效，即每次内存访问只有5个轨迹单元（9.7章节）。这是特别重要的，因为每条指令使用4个内存访问(一个用于获取指令，3个用于3个操作数)；事实上，大多数通常使用读写内存完成的编程任务也可以使用这种新的内存模型完成（见6和8章节）。

置换范围校验：9.9章节表述的置换范围校验，可以实现仅仅用3个轨迹元素去完成一个数的值在$[0, 2^{16})$范围内的校验（相比于原始基于二进制的方法，需要16个轨迹元素），每条指令使用3个这样的范围检查值，所以这样的效率是至关重要的。

内置：Cairo体系结构直接支持预定义操作的实现(作为一组方程)，而不是使用Cairo代码实现它们，我们把这种预定义操作称为内置（见2.8和7章节）；使用内置代码的好处是，它们大大减少了由从手写AIR转换到Cairo代码而增加的开销，这使得程序员可以从编写代码中获益，而不会遭受显著的性能开销。

有效的公共内存：Cairo的内存实现还有另一个重要的特性，每个存储单元应该与验证器共享(例如，程序的代码和输出)，只增加了4个算术运算的验证成本(不包括Fiat-Shamir哈希)。参见章节2.6.1和9.8。

冯·诺依曼不确定性的优势：例如（1）证明验证者只知道哈希(而不是代码)的程序；（2）在一个证明中证明多个不同的程序，以减少均摊验证成本。参见2.2节。

Cairo称呼来自“**C**PU **AIR**”一词 - 一种实现CPU概念的AIR（第9节）。

1.3 概览

第 2 章节介绍了 Cairo 的主要特征，并解释了在架构设计中做出的许多决定。

第 3 章节给出了 Cairo 机器的正式定义，并解释了它如何适于证明系统。

第 4 章节描述了 Cairo 机的状态转换功能。本节非常技术性，因为它解释了构成指令的 15 个flags中的每一个如何影响状态转换。 实际上，15 种可能的组合中只有极少数被使用。 它的对应部分，第 5 章节，介绍了一组有用的指令，可以使用特定的flags配置来实现。这些指令构成了 Cairo 汇编语言（尽管确切的语法超出了本文的范围）。

第 6 章节建议如何安排只读内存以允许处理函数调用（包括递归）。 换句话说，如何在 Cairo 中实现函数调用堆栈。

第 7 章节解释了builtins函数的概念，builtins函数是为选定函数优化的执行单元。

第 8 章节给出了如何处理常见编程任务（例如，整数除法和模拟读写内存）的高级概述，给出了 Cairo 的独特功能（例如，其独特的内存模型和基本算术运算的事实） 在有限域上计算，而不是更常见的 64 位整数算法）。由于 Cairo 的主要目的是生成计算完整性证明，因此必须能够使用证明系统来证明 Cairo 程序的执行成功完成。 STARK【10】是证明系统的自然候选者，因为它能够有效地处理统一计算。

第 9 章节解释了如何将 Cairo 机实现为代数中间表示 (AIR) 【10, p. 14】，这是 STARK 协议中描述的计算方式。 它包括对强制 Cairo 机行为的多项式约束的详细描述。

1.4 概念

贯穿整个文章， $\mathbb{F}$ 代表有限域，其大小为 $|\mathbb{F}|$ ，特征值为 P 。给定两个整数 $a, b \in \mathbb{Z}$ ，我们建立以下概念 $[a, b) := \{x \in \mathbb{Z} : a \leq x < b\}, [a, b] := \{x \in \mathbb{Z} : a \leq x \leq b\}$ 。

1.5 致谢

我们感谢 Eli Ben-Sasson 在 Cairo 的开发和撰写本期间提供的有帮助的评论和讨论，感谢 StarkWare 的工程师在设计过程中提供了有利的建议并帮助实施了系统，感谢 Jeremy Avigad 和 Yoav Seginer 对此文章提供了有益的评论。

2. 设计原则

Cairo的框架可以使得一个人证明任意计算的完整性，换句话说，就是使得verifier相信，给定一个输入，一个程序的执行时正确的。

Cairo是专门为利用STARK【10】协议来有效证明程序执行有效性设计的直观编程框架；即使STARK协议可以直接证明任意计算的完整性，但Cairo围绕STARK提供了一层抽象，简化了计算过程的表述。

为了直接使用STARK证明系统，计算必须被框定为AIR（Algebraic Intermediate Representation）【10, p.14】，见2.1节，这需要相当复杂的设计过程；Cairo框架引入了一种汇编语言（一个完整的编程语言，本文不做详细介绍）来描述计算过程，这比直接设计AIR更加容易。

值得注意的是，Cairo虽然被设计成基于STARK协议，但是它仍然可以用于其他基于有限域的证明系统，比如SNARKs【11】。

本章节描述了Cairo背后的设计原则，并且解释了在我们的设计过程中的一些选择和考量。

2.1 AIR 和 RAP

许多基于有限域的证明系统使用算术电路或者QSP【15】（又名R1CS）；考虑一个算术电路，支持加法门和乘法门，其中所有值都是有限域上的元素。prover获取这样的一个电路，同时给定一个输入（witness），使得最终的输出为0（0代表着数学形式的"true"或者"success"），然后，prover会生成一个证明，证明存在一些输入，使得对应的算术电路返回的结果为0。

STARK证明系统是基于AIR的，而不是算术电路或者R1CSs；AIR可以看做是一系列多项式约束（等式）列表，运行在一个由域元素（有限域 $\mathbb{F}$ ）组成的table上（二维），称之为“轨迹”（witness）。一个STARK证明是证明了存在一个轨迹满足这些约束。

通常，table的列数比较小（20左右），行数是2的幂数；每个约束被分配了一个定义域，该定义域是应用约束的一组周期性的行。 例如，约束可以应用于所有行，每四行或单行。

正如我们在9.6章节看到的那样，Cairo里的AIR不是真正的AIR（如【10, p.14】里的正式定义），它是一个具有预处理的随机性AIR（RAP，见【3】），RAP是AIR的一个广义定义，它允许在prover和verifier之间发生额外的交互步骤，这意味着：

a. 约束可能会涉及到不存在于轨迹元素中的变量 $c_0, ..., c_m \in \mathbb{F}$ ，我们称为交互变量；
b. 轨迹的列被分为两个集合：交互步骤的before和after；
c. 与要求一组满足的赋值不同，我们要求对于 $(c_0, ..., c_m)$ 中的大多数变量，存在一组满足的赋值，其中第一个列的集合与 $(c_0, ..., c_m)$ 独立；

可以修改 STARK 协议以证明描述为 RAP 的陈述（参见第 9.6 节）

因为这个概念在RAP创造之前就被用于Cairo中，因此，在后续的章节中，我们仍然用"Cairo AIR"，而不是"Cairo RAP"。

2.2 冯·诺依曼架构

Cairo框架主要处理把用计算机程序描述的计算转换成用AIR描述的计算的场景，解决这种转换的两个主要方法是：

ASIC：把一个程序编译成AIR；在这个方法中，需要写一个计算机程序（编译器），输入是由某种语言写的一个程序，输出是AIR（一套约束，但表示的是相同的计算）。这类似于编译器，输入是一个程序，输出一个ASIC或者FPGA。

CPU：设计一个AIR（和要被证明的计算无关），其计算逻辑和CPU类似；这个AIR代表了一个单一的计算 - 不断的从内存中寻找一个指令，然后执行它，然后继续寻找下一个指令。这就像在用一个普通的CPU芯片，而不是特定应用的芯片。

ASIC方法的主要优势是高效；基于计算本身构建的AIR没有解码指令和使用内存的开销；但是，就像可以利用内置来减少开销（章节2.8和章节7），这使得需要对计算手写AIR的CPU方法也可以达到和AISC方法接近的性能。如果一个计算，不能通过内置来减少开销，那么这就涉及到一个平衡：（1）接受性能的下降；（2）设计一个新的内置来提高计算的性能。

CPU的方法具有许多优点，包括：

a. 较小的约束集合：因为约束的集合和要被证明的计算没有关系，它具有固定的大小；Cairo CPU的AIR包含30-40个约束，这提高了验证开销；
b. 单一的AIR：不像ASIC方法需要输入计算程序，输出AIR约束；CPU方法只有一个单一的约束集，它可以执行任意程序。因此这种AIR的验证只需要执行一次（不是每个应用一次）。更特

别的是，它简化了证明系统的审计过程。一旦约束已经被校验，那么对于一个新的应用来讲，你只需要审计它的业务代码（这比审计多项式等式更简单）；另外一个优势是，AIR独立于应用本身，因此它也简化了建立递归STARK证明的过程（章节2.2.4）

章节2.2的剩余部分描述了CPU方法遵循冯·诺依曼架构带来的优势。在冯·诺依曼架构架构中，程序的字节码和数据位于相同的内存中，一个叫做“program counter”（PC）的寄存器指向一个内存地址。CPU（1）获取内存里的值，（2）执行该值对应的指令（有可能会影响内存单元或者通过为PC赋予不同的值来改变程序的执行过程），（3）把PC指向下一个指令，（4）重复上述过程。

2.2.1 启动加载：从hash加载程序

一个程序可能会把另外一个程序的字节码写到内存里面，然后让PC指向这个内存段，并开始执行这个程序。

一个特殊的应用就是“从hash启动加载”：一个程序，称为“the bootloader(引导程序)”，计算并且输出另外一个程序的字节码hash，然后开始执行它。通过这种方式，verifier仅仅需要知道被执行程序的hash，而不用知道所有的字节码。

这提高了隐私性和可扩展性：

隐私性：verifier可以验证程序的执行，但是不用知道其具体的计算是什么。

扩展性：假设verifier知道程序的hash，那么其验证时间和程序的大小并不是线性关系（如果验证者的输入是程序本身，而不是它的hash）。

2.2.2 在相同的证明中执行不同的程序

上一章节描述的加载器可以做一个扩展，比如执行好几个程序，输出每一个程序的字节码hash和程序的输出；注意：程序可以描述完全不同的计算。因为proof的大小和验证时间都和计算的大小呈亚线性关系，因此可以使用一个加载器执行多个程序，然后生成一个证明来确保所有程序的有效性；验证的成本在这些程序之间被摊销了。

让我们举一个例子：理论上，STARK的构造是基于STIK的，证明-验证的复杂度和轨迹的长度呈对数关系， $O(logT)$ ，见章节【10，p.21】。STARK构造（使用Merkle树承诺）增加了额外的乘法因子 $O(logT)$ ，使得验证时间的复杂度变成了 $O(log^2T)$ 。为了简单起见，让我们假设一个执行轨迹长度为T的完整性证明的验证时间就是 log^2T ，那么分别验证两个轨迹长度为1M的证明需要耗时 $2log^2T \approx 794$ ，但验证一个轨迹长度为2M的证明仅需要耗时 $log^2(2T) \approx 438$ 。因此，可以预见的是，随着更多的程序加入到批处理中，最终的摊销验证成本将越来越接近于0。

2.2.3 高级优化（即时编译JIT和字节码生成）

在程序执行过程中，一些高级优化可以通过字节码的自动生成来实现。比如，对于一个函数，不同于从内存中寻找对应的值，你可以直接拷贝程序的字节码，然后在对应的指令上直接赋予对应的值；考虑一个指令“从内存中读取x和c，然后计算x+c”，一旦c的值是公开的（此时用C表示），我们可以用高效的指令去替代它，“从内存中读取x，然后计算x+C”，C是指令的立即数。

字节码生成的形式有很多种：一个程序可以根据一些规则生成Cairo字节码然后执行。例如，我们需要计算 x_i^c ，我们可能会写一个函数获取c，然后返回利用连续的乘法来计算 x^c 函数的字节码（而不是像基本的实现那样，利用递归和条件跳转，这样效率更低）

2.2.4 增量可验证计算（递归证明）

递归证明是一个证明另外一个证明有效的证明；例如， A_0 表示一个论述，证明系统的最简单应用就是prover让verifier相信 A_0 为真。现在，让我们定义论述 A_1 “我要验证一个证明 A_0 有效的证明”，那么需要对 A_1 生成一个证明；我们可以继续定义新的 A_2 ，A_3 等。这种方式称为“增量可验证计算”，首次在【23】里定义和分析。

为了生成递归证明，你不得不把验证过程看做要被证明的论述（verifier执行的算法），对于许多证明系统，特别的，基于ASIC方法的，这形成了一个循环依赖：verifier的验证过程取决于要被证明的程序，prover证明过程取决于verifier的验证过程；但是，对于CPU方法，verifier不依赖程序本身，这简化了递归证明，也避免了循环依赖的情况。而且，利用2.2.1章节的从hash启动加载的概念，整个验证过程可以编码成一个hash，然后整个验证过程作为参数传递给自己（这也是生成递归证明的一个步骤）

2.3 指令集

指令集是Cairo CPU可以单步执行完成的操作的集合，本章节主要描述指令集的属性。

2.3.1 指标

为了设计一套最优的指令集，你应该知道哪些指标应该被优化。不像运行在物理芯片上的普通指令集，Cairo是以AIR的方式执行。普通的指令应该使得指令运行的延迟和使用到的晶体管最小，来达到最大化计算的吞吐量；一个高效的AIR的指标则不同：简单来说，设计AIR时（设计一个由AIR执行的指令集时），一个最重要的指标就是使其占用的轨迹单元最小，这或多或少的相当于多项式等式中用到的变量个数。

为了设计一个有效的指令集，你需要理解哪些属性应该优化；一个合理的标准是，一个程序（用上述指令编写的最佳程序）所期望用到的轨迹单元个数。这是一种非正式的标准，因为，为了准确起见，它需要了解程序的分布，并且需要在所有比较的指令集中，每个程序都以最有效的方式编写；然而，在整个设计过程中，仍然可以将此定义作为许多决策的指导原则。

例如，有两个指令集 A, B ，在指令集 A 中，有一个“as_bool”指令，用来判断“如果x = 0，则res = 1，否则，res = 0”，除了这个指令，指令集 B 与指令集 A 的指令完全一样；我们把基于指令集 A 的CPU执行一个指令所占用的轨迹单元数用 a 表示（为了简单，假设所有的指令消耗的单元数一样），而基于指令集 B 执行一步所需的的轨迹单元数用 b 表示（其中 $a > b$ ，由于增加了额外指令的复杂性）；另一方面，一个确定的程序如果基于指令集 A 写，将会执行 k_A 步；基于指令集 B ，将会执行 k_B 步（ $k_B > k_A$ ，有可能会使用更多的指令来执行“as_bool”）；如果 $a \cdot k_A < b \cdot k_B$ ，则指令集 A 是更适合这个程序；相反，则指令集 B 更适合。当决定是否要把一个指令引入进来的时候，你应该考虑每一步的额外消耗（ a/b ）和新增的额外步骤（ k_A/k_B ）之间的关系，同时，也理解了所谓的“typical”程序应该是什么样子。

2.3.2 代数RISC

为了配合上一章节所描述的设计基本规则，Cairo指令集试图在（1）单个指令所需的最小轨迹单元；（2）更强功能的指令以减少执行所需要的步骤数；之间寻找一个平衡。比如：

a. 有限域上的加法和乘法都需要支持，而不是64bit的整数；
b. 需要支持校验两个数是否相等，但是没有一个指令去校验两个数值之间的大小关系（这样的指令将需要更多的轨迹单元，因为有限域不支持其元素代数友好的线性化排序）；

我们把满足上述属性的的指令集称为代数RISC(Reduced Instruction Set Computer），"RISC"表示最小的指令集，"代数"表明支持域上的数学操作。使用代数RISC使得我们为Cairo构建AIR每步只需要51个轨迹单元，对于Cairo CPU的AIR描述在第9章节。

Cairo的指令集可以模拟任意图灵机，因此是图灵完备的；比如，它支持任何可行的计算。然而，实现一些基本的操作，比如元素的比较，仅仅使用Cairo的指令会耗费很多步骤，因此Cairo引入了内置的概念，其不属于指令集的一部分的执行操作代价并不与总步数相乘计算，而是和它被调用的次数有关，具体的见章节2.8和7。

2.4 寄存器

一个帮助区分指令集的重要问题是：这些指令操作什么值？通常，指令操作数，要么是通用寄存器（x64架构的rax），要么是内存单元。许多指令拥有不止一个操作数，它们对这些操作数施加了一些约束（例如一个可能的约束是：最多一个操作数是内存单元，其它的都是通用寄存器）：

几种不同方法的示例：

a. 没有通用寄存器 - 所有的操作数都直接来源于内存单元；
b. 部分通用寄存器 - 指令基于通用寄存器操作，通常，最多一个内存单元；

c. 有界堆栈机 - 可以认为它是具有一些通用寄存器的机器，其中，不同的指令使得值在不同的寄存器之间转移。在大多情况下，允许最多访问一次内存单元，而且，通常，涉及到内存的指令都是些读/写操作，不做任何的计算；

在物理系统中，内存访问是非常昂贵的，因此选项a的设计非常低效（例如，考虑一个求和循环，它必须在每次迭代中将部分和读写到内存中）；对于AIR来讲，未必如此：在Cairo的设计中，一次内存访问只需要占用5个轨迹单元，而对一个指令的解码，就需要访问16个内存单元。

因此，Cairo实现了选项a的设计 - 所有的操作数都来自于内存单元；所以，一个Cairo的指令可能会涉及到3个内存单元，其中在前两个内存单元上执行算术操作（比如加法或者乘法），然后把结果存到第三个内存单元中。

Cairo具有两个地址寄存器，叫做ap和fp，用于指定当前指令对应的操作数对应的内存单元；指令中的每3个值，你可以通过ap+off的寻址方式选择，也可以通过fp+off的方式选择（off是一个常量偏移，范围 $[2^{-15}, 2^{15})$ ，因此，一个指令可以访问 $2 \cdot 2^{16}$ 以内的任意三个元素，在很多方面，这就像拥有很多寄存器。（以一种更便宜的方式实现）

一个指令通过接受一个内存单元的值，并将其作为另一个内存单元的地址，(参见5.2节)可以访问不能用上述形式描述的内存单元。注意，地址空间可以与执行的步骤数一样大。

2.5 非确定性

考虑一个NP问题，比如SAT问题，并且考虑以下两个算法：

算法 A：给定一个SAT实例和一个赋值，如果这个赋值满足公式，则返回为真；

算法 B：给定一个SAT实例，和所有可能的赋值，如果找到一个满足的赋值，则停止且返回为真，否则，如果没有找到一个满足的赋值，则返回失败；

如果prover想让verifier相信一个确定的SAT公式是成立的，它两个算法都可以采用；这意味，无论哪一个算法返回为真，这都意味着SAT公式成立。当然，算法A更为高效，因此prover和verifier更愿意基于算法A去生成证明。即使prover没有满足的赋值，他也可以在本地运行算法B，等找到一个满足的赋值后，再去证明他有一个满足的赋值使得算法A返回为真（这比直接证明算法B返回为真高效），这是因为，证明一个计算比直接运行一个相同的计算（不生成证明）更昂贵。

我们把这种叫做非确定性编程 - prover可能会执行不属于计算部分的额外工作。

我们在前面提到，一个Cairo的指令可能会执行加法或者乘法操作；如果我们把计算x的平方根作为一个较大函数的一部分呢？确定性的方法就是使用某些平方根算法去计算 $y = \sqrt{x}$ ，这意味着我们要在执行轨迹中包含这个过程，但是非确定性的方法更加高效：prover在本地执行某些平方根算法，但是不把它纳入到证明当中，相反的，Prover只需要证明 $y^2 = x$ 即可，这只需要一个Cairo的乘法指令就可以完成。值得注意的，从Verifier的角度，这个y是猜测的，但是所有的程序都在证明这个猜测确实是正确的（特别的，在一些特别的场景下，某些不合法的猜测也会被verifier认为是有效地，比如-y的平方也是x，但它不是x的平方根）。

设计Cairo的一个重要方面就是允许编程者合理采用非确定性编程的优势。

2.5.1 Hints

为了利用非确定性编程的优势，Cairo引入了“hints”的概念。Hints是插在Cairo两个指令中间，需要prover执行的小段程序；因此仅仅是需要prover执行的，因此可以不用包含在证明里面，也可以用任意语言编写，当Cairo Runner(见第3.5节)需要模拟前面有hints的Cairo指令时，它首先运行hints，hints可能会初始化一些内存单元，然后才会继续执行Cairo指令。

2.6 内存

许多CPU架构采用随机访问读写内存，这意味着一个指令可以对任意位置的内存单元进行读，写，替换值操作；然而，这并不是唯一可能的内存模型。例如，在纯函数式编程中，一旦一个变量被设置，它的值就不能改变，因此一个写一次的内存模型可能足以有效地运行一个用纯函数式语言编写的程序。

以下是我们为Cairo结构考虑的集中内存模型：

Read-Write 内存：这是最常见的内存模型，和前面说的一样；

Write-Once 内存：在这个模型中，如果你试着对一个已经赋值的内存进行写操作，将会返回失败；类似的，如果你在写之前读，也会执行失败；

Nondeterministic Read-Only 内存：在这个内存模型中，prover选择内存中的所有值，并且内存是不可变的。Cairo程序只能读取其中的内容；

三种模型的伪代码如图1所示，它说明了，在不同的内存模型中，值信息是如何传递的；在所有的示例中，执行的操作的都是往地址20中写入7，并在后面的代码中，访问它：

```
write(
  address=20, value=7)
...
x = read(address=20)
```

(a) Read-Write Memory

```
write(
  address=20, value=7)
...
x = read(address=20)
```

(b) Write-Once Memory

```
y = read(address=20)
assert y == 7
...
x = read(address=20)
```

(c) Nondeterministic Read-Only Memory

Fig1. 使用不同的内存模型

虽然图1a，图1b有共同的代码；但是，事实上图1b和图1c具有更多的相同之处：图1c的前两个指令确保了prover把地址20的值初始化为7，所以它们的功能就像执行一个写操作的指令。但是在图1a中，我们并不能保证地址20的值就是7，因为并不知道两个指令之间的代码是什么操作；其他两个

模型就可以有这个保证。由于这种相似性，我们有时会把形如“assert read(address=20) == 7”的指令当做赋值指令。

在选择内存模型时，主要是在不同算法的有效实现和每次内存访问时，AIR的有效表示之间权衡；当我们把内存模型从读写内存变成非确定性只读内存时，算法的实现变得受限，但是以AIR的形式表示内存的访问变得更加高效。

事实证明，对于内存访问更多的程序，非确定性只读内存更加简洁（例如，第6章节展示了在只读内存中，函数的堆栈该如何实现）；并且在不适合这种内存模型的地方，你可以用它来模拟一个完全的读写内存（见章节8.5）。因此，**Cairo使用非确定性只读内存作为它的内存模型**。

事实上，有一个额外的限制可以获取更高的效率。内存地址空间是连续的，这意味着如果有个内存访问地址x，另外一个内存访问地址y，那么在地址x,y之间的地址，必定会对应一次内存访问。这个额外的限制使得prover更加高效的用AIR表示内存访问操作，每次访问仅仅占用5个轨迹单元（章节9.7）。

另外一个有意思的角度是释放和重用内存单元。在上面提到的所有内存模型中（假设AIR的实现方法和章节9.7描述的类似），你不得不为每次内存访问付出成本（从轨迹单元的角度），而不是每一个使用过的内存地址。这意味着，**在读写模型中，往一个单元重新写值和向一个新的单元写值，所花费的成本是接近的；所以，编程者不必通过来释放内存和重用内存来节省空间；需要做的仅仅是减少内存访问的次数**。

2.6.1 公共内存

Cairo将非确定性只读内存实现为AIR的一个重要优点是，它允许prover有效地说服verifier某些内存地址包含某些值。通俗来讲，给定一个地址 a_i 和值 v_i 的列表（在prover和verifier之间共享），verifier可以确信在地址 a_i 上对应的值确实是 v_i 。因为这些信息是与verifier共享的，我们把它称为“公共内存”。可以将此列表视为“边界约束”，这些约束被外部化到世界中。

这个机制非常的高效：在生成证明的过程中，生成两个随机数： z, α 。然后，verifier唯一需要做的事情就是计算下面这个表达式：

$$\prod_i (z - (a_i + \alpha \cdot v_i)) \quad \textbf{(1)}$$

并且把结果替换为AIR的一个约束。这意味着，对于每个输入来讲，verifier只需要花费一次加法，一次减法和2次乘法；还有计算列表的hash值作为Fiat-Shamir变换的一部分。

把我们的方法和和原生为AIR添加边界约束做个比较：在原生的办法中，你需要为每一个固定的轨迹单元设计一个约束，这意味着，对于一个内存单元，应该增加两个约束（地址和值）。因为在生成执行轨迹之前，你无法知道特定地址的内存单元对应的轨迹单具体位置，prover不得不把这些信息发送给verifier。

这种机制可以用于：

a. 把程序的字节码加载到内存里（见章节3.3），prover和verifier应该在被执行的程序上达成一致；
b. 传递程序的输入参数和返回值（在prover和verifier之间）；

关于用AIR实现这种机制方法的更多信息，见章节9.8。

2.6.2 在区块链应用中处理链上数据

现在，让我们针对章节2.6.1描述的高效机制考虑一个具体的应用。在一些区块链应用中，一个Cairo程序的输出之一应该是导致应用状态改变的所有日志，这些日志应该可以被verifier访问。它的目的并不是需要被verifier处理，而是仅仅放在“链上”，使得任何一个用户都可以去审查（这种通常被称为数据可用性 - data availability，通常被用在"ZK-Rollup"构造中）。通常，这个日志非常大，并且它对减少验证成本非常重要（比如，在以太坊上，这种成本叫做"gas"）；这个可以用公共内存的机制来解决：可以把这些数据放在内存单元的连续段，并且在公共内存中，包含这些单元。然后，验证开销是将数据传输到区块链的开销和每个数据元素的4个算术操作和数据hash计算开销的总和。

注意，这不是唯一可能的解决方案。处理此类数据的另一种方法，其验证成本甚至更低，是使用区块链友好的哈希函数，由验证者计算数据的哈希，并使相同的哈希计算作为被证明的语句的一部分。这种方法的问题是，通常区块链友好的哈希函数不是STARK友好的，这意味着使用这种方法会显著增加证明成本。

2.7 程序输入和程序输出

一个Cairo应该具有：

输入 - 程序的输入，这部分是不和verifier共享的，在证明系统中，这应该称为“witness”；

输出 - 在程序执行过程中生成的数据，需要和verifier共享，它应该会存储在公共内存中（章节2.6.1）

有没有一种可能，某些数据既是程序的输入又是程序的输出？例如，考虑一个程序，给定一个数 n （作为程序的输入），计算它的 n 次Fibonacci 数列 y ；在这个例子中，程序的输入是 n 。让我们为程序的输出考虑几种情况，每种情况对应的论述为：

a. 如果程序的输出包括 n 和 y ，则要证明的论述为“ n 次Fibonacci数为 y ”；
b. 如果程序的输出仅仅包括 y ，则要证明的论述变为“我知道一个数 n ，满足 n 次Fibonacci数为 y ”（注意： n 不与verifier显示共享）；

c. 如果程序的输出只包含 n ，则要证明的论述变为“我有一个 n 次Fibonacci数”（但是结果不和verifier显示共享）

2.7.1 程序输入

处理程序的输入是非常容易的 - 你可以利用Hints机制把输入解析成需要的格式（如前面所述：Hints可以用任意语言编写）并且更新对应的未初始化的内存单元。例如，在当前的Cairo Runner【4】的实现中，程序的输入是一个JSON文件，它可以被程序特定的Hints读取。

考虑前面提到的Fibonacci数列示例。程序的输入可以写成{" n "：5}形式的JSON文件。位于程序开始的Hints读取这个文件，并获取到其中 n 的值，然后把它放在一个确定的内存单元里。然后Cairo的代码会把这个单元的值传递给Fibonacci函数。

2.7.2 程序输出

程序的输出按照以下方式处理：Cairo程序把输出数据写到内存单元的连续段内。这个段的起始地址和结束地址存在内存中，其地址可以被verifier计算（例如：相对ap寄存器的初始值和最终值，见章节3.2）。verifier可以用过公共内存机制来获取这些外部的内存单元的值（包含输出数据，开始地址，结束地址的内存单元， 见章节2.6.1）。换言之，prover发送段的开始地址，结束地址和所有的输出数据， verifier结合公式（**1**）去验证他们和proof的一致性。

2.8 Builtins（内置）

正如前面章节描述的一样，向一个指令集中添加新的指令是需要成本的，即使这个指令可能不会被使用。另一方面，尝试用Cairo指令实现某种原语是很低效的，因为可能需要很多的指令 - 即使是相对简单的任务，比如整数除法。

为了解决这个问题，同时也为了支持在不新增指令的前提下支持某些预定义的任务，Cairo引入了builtins的概念。一个builtin 可能会对Cairo的内存施加一些约束（取决于builtin本身）。例如，一个builtin 可能会约束某些地址内的值在 $[0, 2^{128})$ 之间；事实上，这是一种非常实用的builtin，我们将在章节8中见到；我们把它称为范围检查 builtin，被它约束的内存单元称为范围检查单元。

Cairo没有一个特殊的指令去调用builtin。相反的，**你只需要简单的去读或者写那些被builtin影响的内存单元**。这种交互方式称为内存映射 I/O。例如，你想要验证一个值 x 在 $[0, 2^{128})$ 内，你只需要把这个值拷贝到一个范围检查单元里；如果你想要验证 x 在 $[0, B)$ 内，其中 $B < 2^{128}$ ，你可以把 x 放在一个范围检查单元里， $B - x$ 放在另外内存单元里。

从构建AIR的角度，这意味着增加builtin不会影响CPU的约束。这仅仅意味着CPU和builtin之间共享了相同的内存。下图2表示了CPU，内存和builtins之间的关系，为了调用一个builtin，Cairo程序可以和确定的内存单元交互，builtin会在这些内存单元上施加约束。

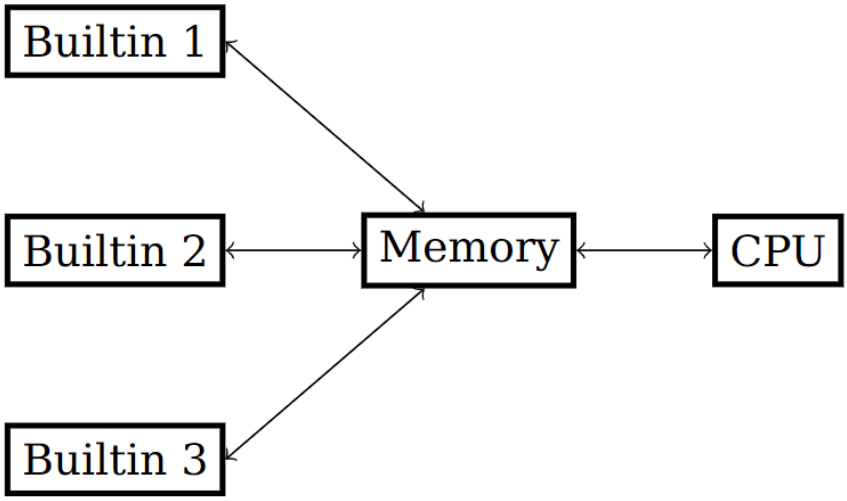


图2. Cairo组件之间的关系

值得注意的是：builtins只是Cairo架构中可选的一部分，任何人可以用具备相同功能的Cairo代码来替换它（利用非确定性编程的优势）。例如，为了实现一个范围检查 builtin，你可以用128个域元素 b_i 来表示 x 的二进制形式，然后断言 $b_i^2 = b_i \ for \ i \in [0, 128)$ 和 $x = \Sigma_i 2^i \cdot b_i$ ，这确保了 x 在期望的范围内。然而，我们比较下两种方法的成本：刚刚的计算过程至少需要 $3 \cdot 128$ 个指令；如果使用builtin（像章节9.9表述的那样实现范围检查技术），它所需要的轨迹单元数量基本上和1.5个指令差不多。

Cairo架构没有指定明确的builtins集合，任何人都可以根据自己的需要添加或者删除builtin。例如，如果一个程序如果需要多次调用Pedersen哈希，直接在架构里面嵌入一个计算Pedersen哈希的builtin是有意义的；另一方面，一个需要使用这个builtin的程序是不能在没有这个builtin的架构上运行的。注意：增加builtins意味着增加了约束，会增加验证者的时间。

3. Cairo框架

现在，我们对“ Cairo机器 ”做一个正式的定义。事实上，我们定义了两个版本：确定性的和非确定性的，其中后者是基于前者的。

确定性的Cairo机器本身不执行计算；相反的，它用来验证一个给定的计算轨迹是有效的。你可以想象一下，对于一个机器，给定一系列的状态和内存函数，然后校验两个连续的状态转换是不是有效的（根绝章节4.5呈现的规则）。如果所有的状态转换以及相对的内存都是有效的，它就返回接受；否则，拒绝。“确定性”这个描述是合理的，因为一个确定性的问题，接受或者拒绝，可以被确定性图灵机有效的解决。

非确定性的版本获取部分内存函数（可以被看做是全部内存函数上的边界约束），初始状态和最终状态（不是所有状态）。如果存在与确定性版本接受的输入一致的状态列表和完整的内存函数，则它接受。

在本章的最后部分，将会描述Cairo Runner，这些理论模型的具体实现并且展示如何使用Cairo Runner去把一个论述，比如“第 j 个Fibonacci数是 y ”转换成确定性机器的输入（prover使用）和非确定性机器的输入（verifier使用），如果机器最终的状态是接受，则说明原始的论述为真。Cairo的AIR（见章节9），允许使用STARK协议去证明非确定Cairo机器接受这些输入，因此来证明原始的论述是正确的。

3.1 确定性Cairo机器

给定一个素数域 $\mathbb{F}_P = \mathbb{Z}/P$ 和一个它的有限扩展域 $\mathbb{F}$

定义1. Cairo机器是一个接受以下**输入**的函数：

a. 步骤数 $T \in \mathbb{N}$ ；

b. 内存函数 $m : \mathbb{F} \to \mathbb{F}$ ；

c. T+1个状态序列 $S_i = \{pc_i, ap_i, fp_i\} \in \mathbb{F}^3 \ for \ i \in [0, T]$

并且**输出**接受还是拒绝。

当且仅当，对于每个 i ，从状态 i 到 状态 $i+1$ 的转换是有效的，才会输出接受；章节4.5描述了如何为机器构建一个有效的状态转换。

需要注意的是，一个单一转换是否有效只取决于涉及到的两个状态（ S_i 和 S_{i+1} ）和内存函数 m ；特别注意的是，用于状态逻辑转换的内存函数对于每个迭代都是相同的；换句话说，Cairo的内存是只读的而不是读写的。

我们把状态 S 的序列叫做Cairo的执行轨迹，我们把 S_0 叫做初始状态，把 S_T 叫做最终状态。

内存函数被正式的定义为一个函数 $m : F \to F$ ，但是在实际的应用中， F 通常特别大，并且在计算的过程中，函数 m 最多访问 $O(T)$ 个值，我们可以把 m 看做是稀疏函数，因为大部分的值都是0。

示例1. （Fibonacci数列），令 $S_0 = \{0, 5, 5\}$ ，并且令 m 满足：

$$\begin{aligned} m(0) &= 0x48307ffe7fff8000, \\ m(1) &= 0x010780017fff7fff, \\ m(2) &= -1, \\ m(3) &= 1, \\ m(4) &= 1 \end{aligned} \tag{2}$$

在4.5章节，你将会看到前两个常量代表Cairo的两个指令，具体的编码方式在本示例中不重要。需要注意：

a. 对于状态转换 $S_0 \to S_1$ ，有：

$$m(pc_0) = m(0) = 0x48307ffe7fff8000$$

仔细阅读章节4.5你会发现，这意味着 $S_0 \to S_1$ 状态转换有效当且仅当：

$$\begin{aligned} pc_1 &= pc_0 + 1, \\ ap_1 &= ap_0 + 1, \\ fp_1 &= fp_0, \\ m(ap_0) &= m(ap_0 - 1) + m(ap_0 - 2) \end{aligned}$$

因此，如果我们想让Cairo机器输出接受，我们必须设置 $S_1 = \{1, 6, 5\}$ 和 $m(5) = m(4) + m(3) = 2$ 。

b. 对于状态转换 $S_1 \to S_2$ ，有：

$$m(pc_1) = m(1) = 0x010780017fff7fff$$

根据章节4.5的描述，如果状态转换有效，则需要满足：

$$\begin{aligned} pc_2 &= pc_1 + m(pc_1 + 1), \\ ap_2 &= ap_1, \\ fp_2 &= fp_1 \end{aligned}$$

因此，如果我们想让Cairo输出接受，我们必须设置 $S_2 = \{0, 6, 5\}$

c. 对于状态转换 $S_2 \to S_3$ ，当前的 $pc = 0$ ，因此约束应该和初始的状态转换类似：

$$\begin{aligned} pc_3 &= pc_2 + 1, \\ ap_3 &= ap_2 + 1, \\ fp_3 &= fp_2, \\ m(ap_2) &= m(ap_2 - 1) + m(ap_2 - 2) \end{aligned}$$

由此推导出，我们应该设置 $S_3 = \{1, 7, 5\}$ 和 $m(6) = m(5) + m(4) = 3$ 。

心细的人可以发现，它自己形成了循环： fp 保持不变， pc 值在0 和 1之间切换，并且在它为0的时候，强制约束 $m(ap) = m(ap - 1) + m(ap - 2)$ 且 ap 自增1；因此，Cairo机器返回接受的唯一办法就是内存函数是否以Fibonacci数列的形式持续（从 $m(3)$ 开始），更准确的说 $m(3), m(4), ..., m(4 + \lceil T/2 \rceil)$ 应该满足Fibonacci数列关系。

3.2 非确定性Cairo机器

定义2. 非确定性Cairo机器是Cairo机器的非确定性版本，它是一个接受如下**输入**的函数：

a. 步骤数 $T \in \mathbb{N}$ ；

b. 局部内存函数 m^* ： $A^* \rightarrow \mathbb{F}$ $where\ A^* \in \mathbb{F}_P$ ；

c. pc 和 ap : pc_I, pc_F, ap_I 和 ap_F 的初始和最终状态（ ap_I 也用作 fp 的初始值）

并且**输出**接受还是拒绝。

当且仅当，存在一个内存函数 m ： $\mathbb{F} \rightarrow \mathbb{F}$ （函数 m^* 的扩展）以及满足 $(pc_0, ap_0, fp_0) = (pc_I, ap_I, ap_I), pc_T = pc_F, ap_T = ap_F$ 关系的一系列的状态 $S_i = \{pc_i, ap_i, fp_i\} \in \mathbb{F}^3\ for\ i \in [0, T]$ 满足**章节3.1**定义的确定性Cairo机器（即接受输入 $\{T, m, S\}$ ）时，非确定Cairo机器会接受输入 $(T, m^*, pc_I, ap_I, pc_F, ap_F)$ 。

注意，**定义1**和**定义2**的区别在于：

- **定义1**对应的确定性版本需要获取所有的状态集合，而**定义2**对应的非确定性版本只需要获取 pc 和 ap 的初始值和最终值；
- 确定性版本需要获取全部的内存函数，而非确定版本只需要获取局部内存函数；
- 对于确定性版本，当用确定性机器计算是否接受一个特定输入的时候，它可以在多项式时间内完成；对于非确定版本，为了达到多项式的时间，你需要一个非确定性机器（见2.5章节的讨论）

示例2. 令 T 是一个偶数，且满足 $T < P$ ， $2 \leq j \leq T/2$ 和 $y \in \mathbb{F}$ ；令 $A^* = \{0, 1, 2, 3, 4, j+3\}, pc_I = 0, pc_F = 1, ap_I = 5, ap_F = 5 + T/2$ ；令函数 $m^* : A^* \rightarrow \mathbb{F}$ 在（0,1,2,3,4）的取值和**公式（2）**相同，并且令 $m^*(j+3) = y$ 。我们可以宣称，当且仅当第 j 个Fibonacci值为 y ，非确定性Cairo机器才会接受这个输入。事实上，基于**示例1**这是成立的：当且仅当，存在一个函数 m^* 的扩展函数，使得确定性Cairo机器返回接受时，非确定机器才会接受。但是，正如我们看到的那样，当且仅当 $m(3), m(4), ..., m(4 + \lceil T/2 \rceil)$ 满足Fibonacci数列关系时，确定性机器才会返回接受；特别的， $m(j+3)$ 是第 j 个Fibonacci数，但它必须和 $m^*(j+3) = y$ 相等。

注意1. 从Cairo AIR的性能角度考虑（更准确的说，用AIR实现Cairo内存访问，见章节9.7），AIR将实施更严格的约束，而不仅仅是非确定性的Cairo机器所接受的事实：在代码执行期间执行的内存访问必须形成一个连续的地址范围；那就是，对于某个初始地址 a_0 和某些自然数 k ，访问的地址集合必须是这种形式 $\{a_0 + i,\ i \in [0, k)\}$ ；特别是，这意味着只能使用来自 $\mathbb{F}_P$ 的地址(而不是来自扩展域 $\mathbb{F}$ 的地址)，这是由于 A^* 是 $\mathbb{F}_P$ 子集；我们将这一额外的要求如下: Cairo代码的程序员不应该依赖于这种连续性保证(从可靠性的角度来看)，但他们应该编写这样的程序；如果输入是有效的(为了完整性)，这一要求将得到满足。

3.3 Cairo程序字节码

Cairo的字节码是一系列域元素的集合 $b = (b_0, b_1, ..., b_{|b|-1})$ ，同时包括两个索引 $prog_{start}, prog_{end} \in [0, |b|)$ ，这些元素组成了Cairo机器将执行的程序。为了执行这个程序，我

们定义了一个新的元素 $prog_{base} \in \mathbb{F}$ （也叫作程序基），使得局部内存函数 m^* 满足：
$m^*(prog_{base}+i)=b_i \ for \ i \in [0,|b|)$ ，并且令
$pc_I = prog_{base}+prog_{start}$，$pc_F = prog_{base}+prog_{end}$ 。

除了程序的字节码之外，部分内存函数还可能包含额外的输入，这些输入为代码的执行提供了额外的约束(例如，执行的输入参数就是这样处理的）。

示例3. 前面示例的Fibonacci数列程序的字节码如下所示：

$$b=(0x48307fffe7fff8000, 0x010780017fff7fff, -1), \ prog_{start}=0$$

这些bytecode被加载到 $m^*(0), m^*(1), m^*(2)$ ，因此 $prog_{base}=0$ 。

$m^*(3), m^*(4), m^*(3+j)$ 的值（更通用的 $m^*(ap_I-2), m^*(ap_I-1), m^*(ap_I-2+j)$ ）是额外的约束。

3.4 Cairo程序

下面的Fibonacci程序是用Cairo汇编写的（更多的细节见第五章）；为了简单起见，我们为每个指令增加了状态转换约束（示例中的所有指令都包含 $fp_{i+1}=fp_i$ ）。

示例4.

\# initial

\# 用（1,1）初始化Fibonacci序列

$[ap]=1; ap++$
$cs: pc_{i+1}=pc_i+2, ap_{i+1}=ap_i+1, m(ap_i)=1$
$[ap]=1; ap++$
$cs: pc_{i+1}=pc_i+2, ap_{i+1}=ap_i+1, m(ap_i)=1$

\# body

\# 迭代计数器减一

$[ap]=[ap-3]-1; ap++$
$cs: pc_{i+1}=pc_i+2, ap_{i+1}=ap_i+1, m(ap_i)=m(ap_i-3)-1$

\# 复制最后一个Fibonacci项

$[ap]=[ap-2]; ap++$
$cs: pc_{i+1}=pc_i+1, ap_{i+1}=ap_i+1, m(ap_i)=m(ap_i-2)$

\# 计算下一个Fibonacci项

$[ap]=[ap-3]+[ap-4]; ap++$
$cs: pc_{i+1}=pc_i+1, ap_{i+1}=ap_i+1, m(ap_i)=m(ap_i-3)+m(ap_i-4)$

\# 如果迭代次数不是0，跳转到body

$$jmp\ body\ if [ap-3]! = 0$$

$$cs: pc_{i+1} = \begin{cases} pc_i - 4 & \text{if } m(ap_i - 3) \neq 0 \\ pc_i + 2 & \text{if } otherwise \end{cases}, ap_{i+1} = ap_i$$

end

无限循环

$$jmp\_\_end\_\_$$

$$cs: pc_{i+1} = pc_i, ap_{i+1} = ap_i$$

这个程序的工作机制如下：假设 $m(ap_I - 1) = j$ 就是我们想要执行的迭代次数，经过前两个步骤，我们有 $(m(ap-3), m(ap-2), m(ap-1)) = (j, 1, 1)$ ，执行完后面三个步骤，变成了 $(j-1, 1, 2)$ ，然后，我们校验 $j - 1 = 0$ ，如果为假，就跳转到body；经过再一次的迭代后，值变成了 $(j-2, 2, 3)$ ，然后继续变成 $(j-3, 3, 5), (j-4, 5, 8)$ 等等，直到迭代计数器变为0；这时，结果存在 $m(ap-2)$ 里，因为这个无限循环不会改变 ap 的值，因此可以在 $m(ap_F - 2)$ 里找到结果。

一个Cairo的汇编器可以把这个程序转化成Cairo的字节码：

$$\begin{gathered} b = (0x480680017fff8000, 1, \\ 0x480680017fff8000, 1, \\ 0x482480017ffd8000, -1, \\ 0x48127ffe7fff8000, \\ 0x48307ffc7ffd8000, \\ 0x20680017fff7ffd, -4, \\ 0x10780017fff7fff, 0) \\ prog_{start} = 0, prog_{end} = 10 \end{gathered}$$

通常情况下，Cairo程序的最后一个指令是无限循环 - 这使得步数 T ，独立于程序，只要它足够大，使程序达到 $prog_{end}$ 。

3.5 Cairo Runner

Cairo Runner负责执行一个编译后的Cairo程序，执行一个Cairo程序不同于执行常规的计算程序，这是源于Cairo支持不确定性编码。例如，以下Cairo指令通过断言某个未初始化单元的平方为25来计算25的平方根：

$$[ap] = 25;\ ap++$$
$$\#[ap-1]\ is\ now\ 25,\ the\ next\ line\ enforces\ that\ [ap]\ is\ the\ square\ root\ of\ 25$$
$$[ap-1] = [ap] * [ap];\ ap++$$

事实上，即使我们意识到这个指令是在说;取 $[ap-1]$ 的平方根，有两个可能的值5和-5，并有可能只有其中一个可以满足其余的指令。以类似的方式，您可以编写一个Cairo程序来解决NP完全问题，例如SAT。

这意味着，如果没有额外的信息，某些Cairo程序可能不会高效的执行（比如：25的特定平方根和满足SAT的赋值），这些信息通过所谓的Hints来给出；Hints是给Cairo Runner的特别指示；用于解决无法轻易推断出某个值的不确定性问题。理论上，Hints可以用任何语言来写，例如在现有的Cairo Runner的实现中【4】，Hints代码块是由python语言实现的。

Cairo Runner的输出包括：

a. 一个被非确定性Cairo机器接受的输入：

$$(T, m^*, pc_I, pc_F, ap_I, ap_F)$$

其中 m^* 包含了程序的字节码（从 $prog_{base}$ 开始）和一些应该被暴露的额外信息（hints可能会执行哪些内存单元应该被添加到 m^* ）；

$pc_I = prog_{base} + prog_{start}, pc_F = prog_{base} + prog_{end}$ ；

b. 一组被确定性Cairo机器接受的输入 (T, m, S) ，构成了非确定性Cairo机器的witness；

或者，Cairo Runner可能会因为执行结果存在冲突或因为没有足够的hints导致无法计算内存单元的值而返回失败。

示例5. 接着上一个示例4，Hints应该设置 $m(ap_I - 1) = j$ 并且把 $ap_I - 1, ap_F - 2$ 的值添加到 A^* （为了暴露Fibonacci应该执行多少次以及最终的结果）

3.6 使用Cairo为计算完整性生成证明

下面描述了为一个给定计算的计算完整性生成证明的大致过程，我们用下面这个断言来表示我们想要证明的示例：

$$the\ j-th\ Fibonacci\ number\ is\ y \quad \textbf{(3)}$$

a. 为这个计算写一个Cairo程序（要么直接用汇编语言写或者用其他可以被编译成Cairo字节码的语言），并且使用hints解决非确定性的部分；在示例4中，我们用Hints实现设置 $m^*(ap_I - 1) = j$ ；

b. 把程序编译成Cairo字节码；

c. 利用Cairo Runner执行程序，获取程序的执行轨迹 S ，局部内存函数 m^* 和全部内存函数 m ，在我们的示例中 m^* 应该包含程序的字节码， $m(ap_I - 1) = j$ 和 $m(ap_F - 2) = y$ ；

d. 用为Cairo AIR服务STARK prover为下述断言生成证明：

$$The\ nondeterministic\ Cairo\ machine\ accepts \\ given\ the\ input (T, m^*, pc_I, pc_F, ap_I, ap_F)$$

我们现在必须得说明，（d）的正确性证明了（c）的正确性：非确定性Cairo机器接受输入意味着存在一组输入 (T, m, S) 使得确定性Cairo机器返回接受；因为函数 m 是 m^* 的扩展，我们知道Fibonacci程序的字节码出现在 $\{m(prog_{base} + i)\}_{i \in [0, |b|)}$ 中；而且，因为 $pc_I = prog_{base} + prog_{start}$ ，我们知道确定性Cairo机器执行的第一条指令就是Fibonacci程序的第一个指令，类似的，剩余的指令也会被执行，总共 T 步；又因为 $pc_F = prog_{base} + prog_{end}$ ，我们知道执行到了程序的无限循环部分，这意味着程序被完整且成功的执行了，并且Fibonacci的第 j 个值应为 $m(ap_F - 2) = m^*(ap_F - 2) = y$ 。

4. CPU架构

Cairo架构定义了确定性Cairo机器的核心，在3.1节中介绍过，它由一个CPU组成，CPU运行在3个寄存器pc、ap和fp上，并且可以访问(只读)内存 m 。

4.1 寄存器

Program counter(pc)：包含当前要执行的Cairo指令在内存中的地址；

Allocation pointer(ap)：根据约定，指向程序到目前为止还**没有使用过**的**第一个存储单元**。许多指令可以将其值增**加一个**，以表示该指令已经使用了另一个存储单元。注意，这仅仅是一个约定，Cairo机器**不会强迫内存单元ap没有被使用**，程序员可以决定以不同的方式使用它；

Frame pointer(fp)：指向当前函数的堆栈帧的开头。fp的值允许类似于堆栈的行为；当一个函数启动时，fp被设置为与当前ap相同，当函数返回时，fp恢复其先前的值。因此，**fp的值对于同一函数调用中的所有指令保持不变**。由于这个属性，fp可以用来处理函数的参数和局部变量。更多信息见第6节

4.2 内存

CPU可以访问一个不确定的只读随机访问内存。只读意味着在执行Cairo代码期间，内存单元的值不会改变。不确定性意味着允许prover选择所有存储单元的初始值，同样也可以选择最终值。Cairo代码仅仅由对内存值的断言组成，它们扮演着读取和写入(一次)内存值的角色，参见2.6节。第6节和第8节解释了如何在常见的编程任务中使用非确定性只读内存，包括模拟读写内存。

我们通常使用 $m(a)$ 和 $m[a]$ 表示在地址 a 出的值。

4.3 执行一个程序

正如3.1节中所描述的，确定性的Cairo机器的输入包括（1）只读内存（2）由寄存器状态序列 (pc_i, ap_i, fp_i) 表示的执行轨迹。两个连续状态之间转换的有效性是由pc寄存器指向($m[pc_i]$)的指令定义的，这是本节讨论的主要主题。每条指令都会对从一个状态到下一个状态的转换产生一些约束。在一般的CPU架构中，状态转换是确定的，CPU必须能够计算下一个给定的状态；Cairo被设计用来验证语句，因此，它可以支持不确定性状态转换。此外，可能存在没有后续状态的状态。如果这样的情况发生在一个程序的执行过程中，我们就说这个执行被拒绝了，并且prover将无法为它生成一个证明

4.4 指令结构

CPU原生的word是一个域元素，而域是一个固定的有限域，其特征 $P > 2^{63}$ ，每一个指令占用1到2个word，存在立即值的指令占用两个word（例如 "[ap] = 12345678"）；每个指令的构成包括（1）三个16bit的有符号整数偏移 $off_{dst}, off_{op0}, off_{op1} \in [2^{-15}, 2^{15})$ ，使用偏移形式编码（ $b_0, ...b_{15} \in \{0, 1\}, -2^{15} + \Sigma_{i \in [0,15]} b_i \cdot 2^i \in [-2^{15}, 2^{15})$ ）；（2）15bit的标志分成7个小组，如图3所示：

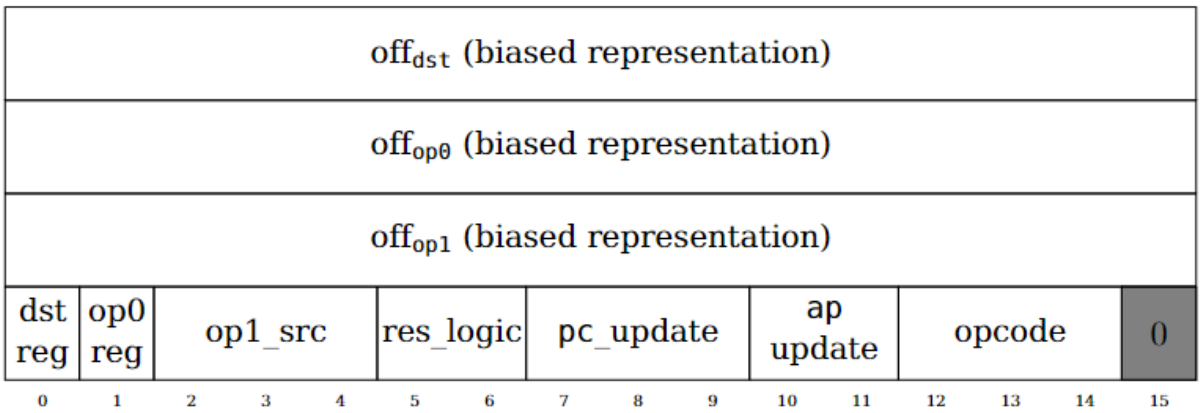


图3. 构成每条指令的第一个字的63位的结构。位按类似于小端的编码顺序排列(最低有效位在前): off_{dst} 显示为**低**16位， dst_{reg} 是第48位(从最低有效位开始)

4.5 状态转换

Cairo的状态转换函数是为了更高效的AIR实现。这里，我们给出这个事实的暗示：一个有效的3bit标志组（例如 op1_src），只会采用0,1,2,4（而不是0->7）。为了理解为什么，我们用 $b_0, b_1, b_2 \in \{0, 1\}$ 来表示op1_src，现在我们有四个函数： $b_0, b_1, b_2, 1 - b_0 - b_1 - b_2$ ，只要其中任何一个是1，其他都是0。这种函数将被用于AIR的构建，将在第9章节了解更多细节。

状态转换函数将会用到4个辅助信息：op0,op1,dst,res，这些信息可以由内存值，偏移信息，指令标志计算而来。

本章节主要介绍了状态转换函数的正式定义，有关设置指令标志以获得有意义指令的各种方法的示例，请参阅第5节。

我们使用术语Unused来描述稍后不会使用的变量。因此，不需要给它赋具体的值。

我们使用术语“Undefined Behavior”来描述导致Cairo机器未定义行为的计算。这意味着在这种情况下，有效的下一个状态的定义可能因Cairo机器的不同实现而不同；因此，程序员应该保证程序不会出现这种情况。

我们使用断言x = y表示两个值之间的另一个相等要求。如果这个需求不成立，这就不是一个有效的状态转换。在这种情况下，可能没有有效的状态转换存在，在这种情况下，Cairo机器将拒绝整个语句，并且不会生成任何证据。

注意:在数学上，状态转换函数可以返回一个state、undefined或reject。

状态转换函数可由下述的伪代码定义：

Context: m(.).

Input state: (pc, ap, and fp).

Output state: (next_pc, next_ap, and next_fp).

Compute op0.

if op0_reg == 0:

$$op0 = m(ap + off_{op0})$$

else:

$$op0 = m(fp + off_{op0})$$

Compute op1 and instruction_size.

switch op1_src:

case 0:

instruction_size = 1

$$op1 = m(op0 + off_{op1})$$

case 1:

instruction_size = 2

$op1 = m(pc + off_{op1})$

If $off_{op1} = 1$, we have op1 = immediate_value.

case 2:

instruction_size = 1

$op1 = m(fp + off_{op1})$

case 4:

instruction_size = 1

$op1 = m(ap + off_{op1})$

default:

Undefined Behavior

Compute res.

if pc_update == 4:

if res_logic == 0 && opcode == 0 && ap_update != 1:

res = Unused

else:

Undefined Behavior

else if pc_update = 0, 1 or 2:

switch res_logic:

case 0: res = op1

case 1: res = op0 + op1

case 2: res = op0 * op1

default: Undefined Behavior

else: Undefined Behavior

Compute dst.

if dst_reg == 0:

dst = m(ap + offdst)

else:

dst = m(fp + offdst)

Compute the new value of pc.

switch pc_update:

```
    case 0: # The common case:
        next_pc = pc + instruction_size
    case 1: # Absolute jump:
        next_pc = res
    case 2: # Relative jump:
        next_pc = pc + res
    case 4: # Conditional relative jump (jnz):
        next_pc =
            if dst == 0: pc + instruction_size
            else: pc + op1
    default: Undefined Behavior
# Compute new value of ap and fp based on the opcode.
if opcode == 1:
    # "Call" instruction.
    assert op0 == pc + instruction_size
    assert dst == fp
    # Update fp.
    next_fp = ap + 2
    # Update ap.
    switch ap_update:
        case 0: next_ap = ap + 2
    default: Undefined Behavior
else if opcode is one of 0, 2, 4:
    # Update ap.
    switch ap_update:
        case 0: next_ap = ap
        case 1: next_ap = ap + res
        case 2: next_ap = ap + 1
        default: Undefined Behavior
    switch opcode:
        case 0:
```

```
            next_fp = fp
        case 2:
            # "ret" instruction.
            next_fp = dst
        case 4:
            # "assert equal" instruction.
            assert res = dst
            next_fp = fp
else: Undefined Behavior
```

5. Cairo汇编

上一节描述了在给定旧状态和内存函数的情况下，每种可能的标志组合如何影响新状态。注意，可能的组合的数量是相当大的：有数千种有效的标志组合，3个偏移量有 $2^{3 \cdot 16}$ 个独立的可能值。

实际上，程序员需要一种更好的方法来描述指令，而不是列出所有的标志值。在本节中，我们将介绍为公共标志组合集提供文本名称的Cairo汇编语法。本节不是Cairo汇编的完整手册。相反，它提供了对Cairo程序集的高级描述以及一些指令示例。Cairo汇编的完整手册超出了本文的范围。参见【5】。常见算法实现的例子可以在第8节中找到。

5.1 常用语法

内存访问[x]： $[.]$ 运算符指的是在地址 x 处的内存值。x可以是ap或fp寄存器中的一个，或者它们的值加到一个常量(例如，[ap + 5])。存储在给定地址中的值也可以用作地址，例如：[[fp + 4] - 2]。Cairo机器在一条指令中最多支持两层解引用；

自增ap：ap++ 通过在指令后附加命令ap++，大多数指令的ap值都可以增加1。唯一不能使用ap++的指令是 call 指令。

5.2 断言相等

断言相等指令可以用下述语法表示：

$$< left\_handle\_op > = < right\_handle\_op >$$

它确保了公式两边是相等的，否则程序的执行将会被返回。

等式左边往往来自 $[fp + off_{dst}]$ 或者 $[ap + off_{dst}]$ ，右边有一些可能的形式(reg_0 和 reg_1 可以是 fp 或 ap ， $\circ$ 可以是加法或乘法， imm 可以是任何固定域元素):

- imm
- $[reg_1 + off_{op1}]$
- $[reg_0 + off_{op0}] \circ [reg_1 + off_{op1}]$
- $[reg_0 + off_{op0}] \circ imm$
- $[[reg_0 + off_{op0}] + off_{op1}]$

注意2：除法和减法可以分别表示为具有不同操作数顺序的乘法和加法。

正如第18页所解释的， $assert$ 指令可以被认为是一条赋值指令，其中一边是已知的，另一边是未知的。例如 $[ap] = 4$ 可以被认为是断言的 $[ap]$ 值为4，或者根据上下文将 $[ap]$ 赋值为4。

图4给出了断言相等指令的一些示例，以及每个指令对应的标志值：

	off_{dst}	off_{op0}	off_{op1}	imm	dst	op0	op1	res	pc_update	ap_update	opcode
[fp + 1] = 5	1	−1	1	5	1	1	1	0	0	0	4
[ap + 2] = 42	2	−1	1	42	0	1	1	0	0	0	4
[ap] = [fp]; ap++	0	−1	0	∅	0	1	2	0	0	2	4
[fp - 3] = [fp + 7]	−3	−1	7	∅	1	1	2	0	0	0	4
[ap - 3] = [ap]	−3	−1	0	∅	0	1	4	0	0	0	4
[fp + 1] = [ap] + [fp]	1	0	0	∅	1	0	2	1	0	0	4
[ap + 10] = [fp] + [fp - 1]	10	0	−1	∅	0	1	2	1	0	0	4
[ap + 1] = [ap - 7] * [fp + 3]	1	−7	3	∅	0	0	2	2	0	0	4
[ap + 10] = [fp] * [fp - 1]	10	0	−1	∅	0	1	2	2	0	0	4
[fp - 3] = [ap + 7] * [ap + 8]	−3	7	8	∅	1	0	4	2	0	0	4
[ap + 10] = [fp] + 42	10	0	1	42	0	1	1	1	0	0	4
[fp + 1] = [[ap + 2] + 3]; ap++	1	2	3	∅	1	0	0	0	0	2	4
[ap + 2] = [[fp]]	2	0	0	∅	0	1	0	0	0	0	4
[ap + 2] = [[ap - 4] + 7]; ap++	2	−4	7	∅	0	0	0	0	0	2	4

图4. 断言指令示例

5.3 条件和非条件跳转

jmp 指令允许更改程序计数器 pc 的值。

Cairo支持相对跳转（其操作数代表相对当前 pc 的偏移）和绝对跳转 - 分别用关键字 rel 和 abs 表示； jmp 指令或许是有条件的，比如当某个内存单元的值不为0时，触发 jmp 指令。

指令的语法如下所示：

Unconditional jumps.

jmp abs <address>

jmp rel <offset>

Conditional jumps.

jmp rel <offset> if <op> !

图5给出了 jmp 指令的一些示例，以及每个指令对应的标志值：

	off_{dst}	off_{op0}	off_{op1}	imm	dst	op0	op1	res	pc_update	ap_update	opcode
jmp rel [ap + 1] + [fp - 7]	-1	1	-7	$\emptyset$	1	0	2	1	2	0	0
jmp abs 123; ap++	-1	-1	1	123	1	1	1	0	1	2	0
jmp rel [ap + 1] + [ap - 7]	-1	1	-7	$\emptyset$	1	0	4	1	2	0	0
jmp rel [fp - 1] if [fp - 7] != 0	-7	-1	-1	$\emptyset$	1	1	2	0	4	0	0
jmp rel [ap - 1] if [fp - 7] != 0	-7	-1	-1	$\emptyset$	1	1	4	0	4	0	0
jmp rel 123 if [ap] != 0; ap++	0	-1	1	123	0	1	1	0	4	2	0

图5. 跳转指令示例

5.4 $call$ 和 ret

$call$ 和 ret 指令允许实现函数堆栈。 $call$ 指令更新程序计数器(pc)和帧指针(fp)寄存器。程序计数器的更新类似于 jmp 指令。之前 fp 的值被写入 $[ap]$ ，以允许 ret 指令将 fp 的值重置为调用之前的值；类似地，返回的 pc (调用指令后面指令的地址)被写到 $[ap+1]$ ，以允许 ret 指令跳回并继续执行调用指令后面的代码的执行。由于写入了两个存储单元， ap 向前进了2， fp 被设置为新的 ap 。

指令的语法如下：

call abs <address>

call rel <offset>

ret

图6给出了 $call$ 和 ret 指令的一些示例，以及每个指令对应的标志值：

	off_{dst}	off_{op0}	off_{op1}	imm	dst	op0	op1	res	pc_update	ap_update	opcode
call abs [fp + 4]	0	1	4	$\emptyset$	0	0	2	0	1	0	1
call rel [fp + 4]	0	1	4	$\emptyset$	0	0	2	0	2	0	1
call rel [ap + 4]	0	1	4	$\emptyset$	0	0	4	0	2	0	1

call rel 123	0	1	1	123	0	0	1	0	2	0	1
ret	−2	−1	−1	∅	1	1	2	0	1	0	2

图6. call和ret指令示例

5.5 高级 ap

指令 $ap += < op >$ 通过给定的操作数增加 ap 的值。

图7给出了高级 ap 指令的一些示，以及每个指令对应的标志：

	off_{dst}	off_{op0}	off_{op1}	imm	dst	op0	op1	res	pc_update	ap_update	opcode
ap += 123	−1	−1	1	123	1	1	1	0	0	1	0
ap += [fp + 4] + [fp]	−1	4	0	∅	1	1	2	1	0	1	0
ap += [ap + 4] + [ap]	−1	4	0	∅	1	0	4	1	0	1	0

图7. 高级ap指令示例

6 推荐的内存布局

这个部分描述了我们认为的在Cairo程序中管理内存布局的最佳实践。Cairo体系结构并**不强制**执行本节中描述的模式。相反，在这里描述它们是为了说明如何实现常见的内存管理概念，比如函数调用堆栈，由Cairo的唯一内存模型 （不确定的只读内存）实现。

6.1 函数调用栈

这部分展现了Cairo架构中的函数调用栈的实现。调用栈是一种广泛使用的模式，支持递归之类的编程流。在常见的体系结构中，调用堆栈的深度增加了，并且**每当调用一个函数时都会创建一个新帧**（frame）；每当函数返回时减少 - 并释放帧的内存。然后，内存可以被未来函数调用的帧覆盖。由于内存是只读的，所以这种方法不能在Cairo中按原样实现。不过，只需要进行小的调整。

帧指针寄存器（fp）指向调用栈的当前帧；你将看到，不将fp定义为帧的开始，而是将其定义为局部变量部分的开始是很方便的，这与普通体系结构中堆栈的行为类似。每个帧由4部分组成（当前帧）：

a. caller调用函数的参数，例如 $[fp-3], [fp-4], \ldots$

b. 指向调用函数帧的指针，位于 $[fp-2]$

c. 一旦函数返回后，要执行的指令地址（跟随在call指令后面的），位于 $[fp-1]$

d. 函数分配的局部变量，例如 $[fp], [fp+1]$

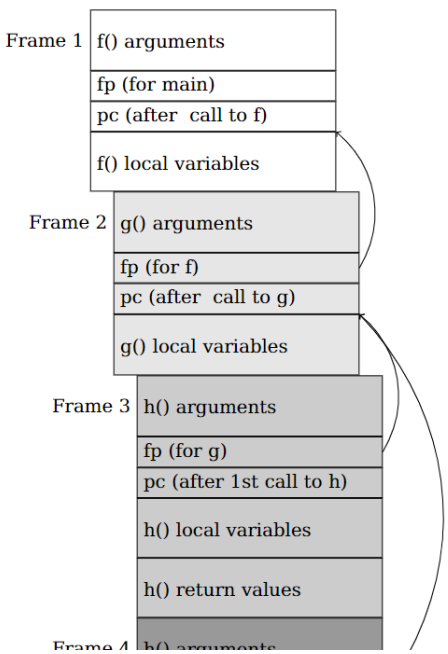
...

另外函数的返回值被放在 $[ap-1], [ap-2], \ldots$ ，其中 ap 是函数返回后， ap 寄存器的值。

```
f:
    call g
    ret
g:
    call h
    call h
    ret
h:
    ret
```

图8. 函数调用示例

图8展示了一个函数调用的代码示例，图9展示了这个函数的调用栈情况； Frame 1是函数f()的帧，在这个代码示例的开始，fp指向这个帧；当调用函数g()后，Frame 2被激活，fp指向这个帧；对函数h()的调用，激活了Frame 3；当函数h()返回时，fp又指向了Frame 2（根据 $[fp-2]$ ）；对函数h()的第二次调用，激活了Frame 4，当它再次返回时，fp又指向了Frame 2；当函数g()返回时，fp指向了Frame 1。



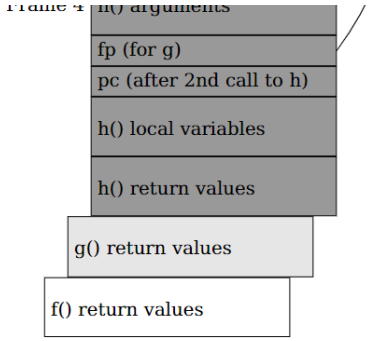


图9. 调用栈，缩进表示此时的调用栈深度

6.2 内存段

Cairo机器，如定义1中所定义的，允许随机访问内存，并可以支持非常大的地址定义域(像域一样大的尺寸)。另一方面，为了在Cairo AIR实现中获得最大的效率，Cairo强加了一个额外的要求，即被访问的内存地址集必须是连续的（见注意 1 和章节9.7）。

可以利用Cairo的非确定性来减轻连续性需求：Cairo Runner支持可重定位内存段的概念；内存段是内存里的连续段落，并且其地址范围不是固定的；并且在计算的末尾由Cairo Runner选择（见章节3.5）；我们把这个过程叫做段重定位。在段重定位过程之前，属于可重定位段的内存单元地址可以用一对 (s,t) 表示，其中 s 是标识段的整数，t是指定段内偏移量的非负整数。**在运行的结束**，Cairo Runner计算每个部分的大小(大于给定片段标识符 (s,t) 所有 t 的最小正整数。然后，Cairo Runner分配每段最后一个内存段基地址，以便一个接一个的放置；比如，如果给内存段 s 设置基地址是 $a \in \mathbb{F}$ ，则重定位地址 (s,t) 将被重定位到 $a+t$ 。

段机制为其他编程语言中的动态内存分配过程提供了一种有效的替代方案。通常，动态内存分配是一个复杂的过程：标准库和操作系统必须跟踪已分配的段，处理内存的释放，并找到分配新段的最佳位置，以减少碎片。在Cairo中，这要简单得多；你所要做的就是分配一个新的内存段(Cairo Runner选择一个未使用的 s)，程序员甚至不需要指定内存段的大小。

因为重定位是由Cairo Runner完成的，所以不用去被证明。例如，恶意验证者可能定义重叠的段。在读写内存中，这将是一个问题，人们如何知道改变一个段中的值不会改变另一个段中的值?然而，由于Cairo的内存是只读的，所以这个问题并不存在：只要程序在处理段之间的差异时是不可知的，那么重叠的段与非重叠的段是无法区分的。

下面是几个Cairo Runner定义的常用段：

a. **Program segment**：存放被执行程序的字节码；
b. **Execution segment**：存放执行栈，如章节6.1所示；
c. **User segments**：由程序定义的通用段（动态分配）；
d. **Builtin segments**：为builtins分配的段，第7章会讲到；

7 Builtins

虽然Cairo是图灵完备的架构，可以表示任何形式的计算，但是基于原生的AIRs，会产生过度的开销；与物理cpu采用的方法类似，克服这个问题的一种方法是添加许多预定义的优化的低级执行单元，称为内置单元。

这样的内置单元有：

- **Range check**：验证某个元素是否在 $[0, n)$ 内；
- **Cryptographic primitives**：例如，加密，哈希函数和签名验证；
- **General-purpose builtins**：例如执行**任何给定的算术电路**；

实际上，对于许多常见的程序，builtins的使用可以极大地减少从原始的AIRs程序迁移到Cairo程序的开销。本节描述内置组件builtins如何作为Cairo体系结构的一部分工作，特别是Cairo程序如何使用它们。

7.1 内存通信

每个builtin内存被分配一个内存段(见第6.2节)，通过这个内存段，Cairo程序可以与内置内存通信(见第2.8节的图2)。

比如，考虑一个哈希Builtin，其目的是，给定两个输入 x 和 y ，计算它们的哈希 $H(x, y)$ ；我们可以把“共享”的内存段拆成三个一组，即对于 $i \in [a, b)$, $\{m(3i), m(3i+1), m(3i+2)\}$ ，这个哈希builtin的责任就是验证 $m(3i+2) = H(m(3i), m(3i+1))$ ；然后，当Cairo程序想调用这个哈希builtin时，只需要简单把 x 和 y 分别写入 $m(3i)$ 和 $m(3i+1)$ ，对应的从

$m(3i+2)$ 读取结果；一旦我们用了地址 $\{m(3i), m(3i+1), m(3i+2)\}$ 去计算了某一个hash，那么它们就不用被用于计算其他的哈希，因为Cario的内存是不能更改的。

内存不可变性的另一个含义是，不能在内置段中维护一个指向最后使用内存地址的全局指针；相反，必须将这个指针传递给使用该内置函数的每个函数；特别是，main()入口点应该为每个使用的内置函数获取并返回指针；它获得指向内存段开头的(start)指针，并且应该在最后一次使用内置内存之后返回(stop)指针。

为了检查分配给builtins内存段的内存是否溢出，这两个指针(start和stop)通过公共内存机制导出(参见章节9.8)，并由验证器进行验证。例如，假设哈希内置有 $b-a$ 个实例，它所分配的段分布在地址 $[3a, 3b)$ 上(回想一下，每个实例有三个单元格)。然后验证器应该检查发送给main()的指针是否为3a, main()返回的指针满足 $3a \leq return\ pointer \leq 3b$ 。

Cairo Runner(见3.5节)提供了预定义的hints来计算一些内置内存段中的值。例如，在内置哈希中，一个有用的hints是通过计算前两个元素的哈希来推断 $m(3i+2)$)的值。

8 Cairo结构

在本节中，我们将描述如何使用Cairo指令构造现代编程语言的基本构件，具体来说，如何处理内存限制(不可变性)以及基本类型是域元素而不是整数的事实。

本节中的代码示例将以伪代码汇编的形式给出，其语法不会显式定义。因为Cairo是不确定的，所以代码将使用不确定的“猜测”概念，见章节2.5.1。

8.1 Loop 和 recursion

在第6章节中，我们已经知道了函数调用栈的布局和以及Cairo机器是如何支持函数调用的；下面，我们考虑一个计算 $x \cdot a^n$ 的递归函数：

```
def exp(x, a, n):
    # Compute x * a ** n.
    if n == 0:
        return x
    return exp(x * a, a, n - 1)
```

用Cairo指令实现如下：

exp:

```
# [fp - 5], [fp - 4], [fp - 3] are x, a, n, respectively
jmp body if [fp - 3] != 0
# n == 0. Return x.
[ap] = [fp - 5]; ap++
ret
body:
# Return exp(x * a, a, n - 1).
[ap] = [fp - 5] * [fp - 4]; ap++
[ap] = [fp - 4]; ap++
[ap] = [fp - 3] - 1; ap++
call exp
ret
```

特别要注意的是，call指令之后紧跟ret指令的模式。这是一个尾部递归，其中被调用函数的返回值被转发。具有循环的函数可以通过定义使用尾部递归的助手函数来实现。这种模式在函数式编程语言中很常见。

8.2 整数运算

由于Cairo的基本word是一个域元素，人们可能想知道如何根据标准体系结构实现对有界整数的基本整数算术操作或对其他数字(如 2^{64})求模；这可以通过范围检查 **builtin**实现（章节2.8）。令 M 为范围检查 **builtin**的边界，你可以把范围检查 **builtin**看做是向伪代码里面引入新的源语指令：

$$assert\ x\ in\ [0, M)$$

在本章节的其余部分，我们假设 M 远小于 P （至少小于 $P/2$ ）。

示例6.（边界 $B \leq M$ 内的加法），给定两个元素 $x, y \in [0, B)$ ，计算它们的和，并验证它们的和仍然在 $[0, B)$ 内：

```
z := x + y # Field operation.
assert z in [0, M)
assert B - z in [0, M）
```

这表明仅仅这两个range check调用就足够验证任何一个小于 B 的数。

示例7.（乘法模 $B \leq M, (B-1)^2 + B \leq P$ ），计算 $x \cdot y \ mod \ B$ 的值，其中 $x, y \in [0, B)$ ；可以“猜测”小于 B 的余数和商，然后验证 $x * y = residue + quotient \cdot B$ ，因为后者是在有限域内验证的，需要对商进行一个校验以防止溢出。

我们支持的最大的商为 $[(B-1)^2/B]$ 。

对应的算法如下：

z := x * y

residue := guess

quotient := guess

assert residue in $[0, B)$

assert quotient in $[0, (B-1)^2/B)$

assert z == residue + quotient * B

return residue

注意：使用非确定性的行 - residual:= guess 应该转换为使用未初始化的内存单元(其值将由hints设置)。关于非确定性的更多信息，请参见2.5节。

8.3 定点和浮点运算

定点运算与整数运算类似，不同的是每次乘法之后，都要除以 2^b ，其中 b 是固定精度(以位为单位)。为此，需要进行带余数的整数除法运算。

Notation：一个有 b 个定点位宽的数 x ，可以用 X 表示，满足 $x = X/2^b$

接下来，让我们看看如何计算两个定点数的相乘 $z = xy$ ，注意 $Z = z \cdot 2^b = xy \cdot 2^b = XY/2^b$ ，但无法保证 XY可以整除2^b ，所以不得不使用整数除法运算，可能会损失b位精度 - 符合定点运算的预期；整数除法的实现可以和**示例7**描述的模乘计算一样，给定 $x, y < B$ ，且假设 $(B-1)^2 + B \leq P$ ：

Compute x // y.

residue := guess

quotient := guess

assert residue in [0, y)

assert quotient in [0, B)

assert x == residue + quotient * y

return quotien

浮点数的计算会很低效但也可以实现；每个浮点数可以用2个word表示，指数和尾数。例如，(x_e, x_m) 表示浮点数 $(1 + x_m/2^b) \cdot 2^{x_e}$ 。

示例8.（浮点数乘法）给定两个浮点数 (x_e, x_m) 和 (y_e, y_m) ，计算它们乘积的伪代码如下：

z_e := bounded_addition(x_e, y_e)

z_m := fixed_point_multiplication(x_m, y_m)

if fixed_point_less_than(z_m, 2):

    return (z_e, z_m)

else:

    return (bounded_addition(z_e, 1), integer_division(z_m, 2)

加法的实现方法是，首先移动其中一个数字，使其指数相等，然后将得到的尾数相加。

8.4 获取寄存器的值

Cairo没有专门获取寄存器 ap, fp, pc 值的指令，你可以获取这些值，然后把它存在内存中；使用call指令（见章节5.4），如下：call指令把 fp, pc 的值写进内存，然后令 $fp = ap$ ；因此，经过连续的两次调用之后，内存栈里的前4个值（ $[ap - 4], [ap - 3], [ap - 2], [ap - 1]$ ）分别是 $fp, pc, ap + 2$ 和其他我们不关心的值。

下面的代码可以用来获取寄存器的值：

get_registers:

call get_ap

ret

get_ap:

ret

....

call get_registers // 连续call

We now have:

[ap - 4] == fp

[ap - 3] == pc

[ap - 2] == prev_ap + 2 == current_ap - 2

8.5 读写内存

Cairo的内存不原生支持读写内存，但是可以利用不确定性建立一个数据结构然后去模拟随机读写内存。

8.5.1 Append-only 数组

在描述如何在Cairo中实现读写字典之前，我们先描述一个重要的构建块：Append-only数组。

Append-only数组是一种数据结构，它支持在固定时间内进行两种操作:在任意索引上追加值和查询值。它不支持修改或删除值。

为了实现Append-only数组，我们在把数组的值存在了内存段里面（见章节6.2），然后维护两个指针array_start：指向数组的第一个单元；array_end：指定第一个没有被使用的内存单元。

数组的大小可以表示为array_end - array_start，如果要访问某个元素，则仅仅使用[array_start + i]。

append函数获取当前的两个指针，然后返回更新后的指针：

append_to_array(**array_start**, **array_end**, new_value):

    # Write the value at the end of the array.

    [array_end] = new_value

    # Return the updated pointers.

    return (**array_start**, **array_end + 1**)

一个Append-only数组可以作为一个操作日志：比如你有一个 读和写操作的序列。人们可以不确定地猜测 每个操作的结果，并将该操作和它的所谓结果记录在 这样一个数组。在执行结束时，人们可以翻阅日志并 检查日志是否一致，猜测的值是否正确。

8.5.2 读写字典

读写字典数据结构保存着从键到值的映射，并允许设置和检索任意键。可以在Cairo中使用Append-only数组实现这一点。

数组将保存(key, previous value, new value)的三元组；每个表示对字典的特定键的单个读和写访问。数组中这三个一组的顺序将是访问的时间顺序。每个key有可能会出现在多个三元组中（很有

可能）；当访问字典的时候，正确的previous value（事实上，是当前key对应的值）可以被（非确定性）猜测。

当用了字典之后，程序必须验证访问的三元组是连续的；即，对于一次访问 (k, p_0, n_0) ，当执行下次对同样key的访问时 (k, p_1, n_1) ，应该满足 $n_0 = p_1$ ；这个约束保证了，当给一个Key设定某个值时，下一次访问这个key返回的值就是当前写入的值。

为了校验这个约束，需要采用如下的方法：

a. prover按照key来排序访问数组（通过hints实现，同时保证相同key的访问顺序不变）；
b. 验证(猜测的)排序数组是否匹配原始访问数组的排序结果(由于不确定性，这可以在O(n)中完成，而不是通常的排序成本，O(nlog n))
c. 遍历排序后的数组，并在本地检查一致性条件

8.5.3 基于置换校验的排序

给定两个大小为 n 的数组 A, B ，校验 A 是 B 的稳定排序是可能的。可以通过猜测一个 $A \to B$ 的映射来实现：一个没有冲突的映射，从 A 的索引映射到 B 的索引，这就是置换校验。这个排序可由以下几步完成：

a. 校验 A 是排序后的，校验每个值，然后判断没有相邻递减的；
b. 校验 A 是 B 的置换：为了简单起见，假设排序后的不同的key集合为 $0, 1, ..., m-1$ ，$j_0, j_1, ..., j_{m-1}$ 是 A 中每个段的第一个索引；从完整性角度，假设 $j_m = n$ ；然后对于每一个段 $S \in [S_i, S_{i+1})$ ：
   i. 猜测 B 的索引，对应映射关系为 $I_i : S_i \to [0, n)$ ，满足：
   $$B[I_i(k)] = A(k)$$
   ii. 通过校验 I_i 的递增性来确保排序的稳定性：
   $$I_i(k+1) > I_i(k)$$

该算法还可以通过引入第三个数组并检查它是否是两个原始数组的排序来检查两个数组是否是另一个数组的置换。

9 Cairo的AIR

本章节主要描述了如何为Cairo机器构造AIRs。在章节2.1中，我们定义了AIRs是作用在域元素表（轨迹）上的一个多项式约束系统。Cairo机器的AIR是这样的一个系统，当且仅当机器输出接受的时

候，约束才是成立的；这样的AIR允许prover让verifier相信机器的输出是接受。请注意，要理解Cairo语言或体系结构，并不需要理解这一部分。

虽然这个论文不包含对AIR的完整性（如果Cairo机器接受，则AIR有一组有效的赋值）和可靠性（一组有效的赋值会使得机器返回接受）证明，大多数的约束是直观的，并且接近章节4.5的定义。由于内存（章节9.7）和置换校验约束（章节9.9），完整性和可靠性都是不完美的；定理2分析了内存约束的可靠性，置换范围校验的可靠性分析方法类似。一个即将发布的论文【7】，用LEAN定理证明了Cairo AIR的可靠性。

描述AIR的直接方法需要描述轨迹中的每个单元的含义，然后列出每个约束，解释它约束的内容；本节以另一种方式展示AIR，它在列出约束和扩展轨迹单元格之间交替进行。出于我们的目的，我们不会检查表中每个单元格的确切位置。我们将只关注单元格存在多少个实例。例如，我们将用 $N = T + 1$ 个轨迹单元去表示 $pc_i, i \in [0, T]$ ， $16N$ 个轨迹元素去表示指令的flags，占用15bit（从技术角度的原因，单元的个数需要满足2的阶）。

拥有相同功能角色的轨迹单元集合叫做virtual column，当收集了所有需要的virtual column之后，我们可以自主决定放在表格中的位置。每一个virtual column被周期的放在一个column里，例如，如果轨迹的长度（表的rows number） $L = 16N$ ，那么代表 pc_i 的虚拟列就可以每16行放置一个。最优的放置布局是容易的，因为它们之间的比率是2的阶数，在本文中，不详细说明这些。

我们还定义了virtual subcolumn，从构成virtual column的单元中周期性的获取子集，然后把这些单元看成一个virtual column。例如，我们可能会定义一个大小为 $4N$ 的虚拟列，然后我们约束所有的单元值都在 $[0, 2^{16})$ 范围内，然后定义三个大小为 N 的子列 $off_{dst}, off_{op0}, off_{op1}$ ，这允许我们对父列写一套约束（对范围验证的约束），然后根据指令的真实的使用场景来给子列 $off_{dst}, off_{op0}, off_{op1}$ 写三套约束。

因此，我们继续进行：描述几个virtual column；然后是一些约束条件；然后可能描述更多的virtual column和约束等等。

9.1 Notation

令 $\mathbb{F}$ 表示基础域 - 一个大小为 $|\mathbb{F}|$ 的有限域，其特征是 $P > 2^{63}$ ，令 T 表示执行的步数；状态的个数是 $N = T + 1$ ；轨迹的长度 $L = 16N$ ，因为轨迹的长度必须是2的幂数，所以 N 也是。我们假设 $L < P$ ，注意，有限域内 $\mathbb{F}$ 的每个整数的范围是 $[-[P/2], [P/2]]$ ，我们将交替使用两种表示方法。

9.2 论述完整性证明

按照章节3的描述，非确定性Cairo机器的是一个接受以下输入的函数：

a. 步骤数 $T \in \mathbb{N}$；

b. 局部内存函数 $m : A^* \to \mathbb{F} \ where \ A^* \in \mathbb{F}_P$ ；

c. pc 和 $ap : pc_I, pc_F, ap_I$ 和 ap_F 的初始和最终状态（ ap_I 也用作 fp 的初始值）

并且输出接受或者拒绝。

这个论述的参数叫做公开输入，准确来说就是 $(T, m^*, pc_I, ap_I, pc_F, ap_F)$ ，这代表了Prover和verifier都知道的信息。

对于参数的每一组值（公开输入），我们定义了一个可以被叫做完整性描述的论述：

定义3.（完整性论述）当非确定性Cairo机器的输入是 $(T, m^*, pc_I, ap_I, pc_F, ap_F)$ 的时候，输出接受。

本节剩余部分将专门描述上述论述的AIR。

9.3 Quadratic AIR

章节2.3.1里没有提到的一个指标是在AIR中，约束的最大阶数。只要这个阶数不大，轨迹单元的数量对AIR的性能是一个很好的估计。然而，在某些情况下，为了减少约束的最大阶数，增加一些额外的轨迹单元是有益的。正如后面将要看到的，大部分的Cairo AIR约束都是2阶的；章节9.5描述了两个特例，其原生的阶数是3；对于这些约束，我们通过增加额外的单元，使其阶数缩减到了2，这样Cairo AIR的所有约束都是2阶及以下的了。

9.4 指令flags

正如章节4描述的那样，每个指令由以下几部分组成：

a. 第一个word：15bits的flags（ f_* ）和3个偏移（ off_* ）；

b. 第二个word（可选的）：一个立即值（域元素）

其flags组定义如下：

$$
\begin{aligned}
dst_{reg} &= f_{DST\_REG} \\
op0_{reg} &= f_{OP0\_REG} \\
op1_{src} &= f_{OP1\_IMM} + 2 \cdot f_{OP1\_FP} + 4 \cdot f_{OP1\_AP} \\
res_{logic} &= f_{RES\_ADD} + 2 \cdot f_{RES\_MUL} \\
pc_{update} &= f_{PC\_JUMP\_ABS} + 2 \cdot f_{PC\_JUMP\_REL} + 4 \cdot f_{PC\_INZ} \\
ap_{update} &= f_{AP\_ADD} + 2 \cdot f_{AP\_ADD1} \\
opcode &= f_{OPCODE\_CALL} + 2 \cdot f_{OPCODE\_RET} + 4 \cdot f_{OPCODE\_ASSERT\_EQ}
\end{aligned}
$$

定义 $\widetilde{off}_* = off_* + 2^{15}$ （其中 * 代表 dst, op0, op1 中的一个），所以 $\widetilde{off}_*$ 的取值范围应该是 $[0, 2^{16})$ 。我们为 $\widetilde{off}_*$ 分配一个virtual column（三个virtual, 每个大小为N），而不是 off_* 。为了将第一个word解压，我们使用了bit-unpacking 组件：

$$inst = \widetilde{off}_{dst} + 2^{16} \cdot \widetilde{off}_{op0} + 2^{32} \cdot \widetilde{off}_{op1} + 2^{48} \cdot \sum_{i=0}^{14}(2^i \cdot f_i)$$

定义 $\tilde{f}_i = \sum_{j=i}^{14} 2^{j-i} \cdot f_j$ ，所以 $\tilde{f}_0$ 是15bit的值， $\tilde{f}_{15} = 0$ ；注意 $\tilde{f}_i$ 是 $\tilde{f}_0$ 的位前缀；我们为 $\{\tilde{f}_i\}_{i=0}^{15}$ 分配一个大小是 $16N$ 的virtual column，而不是为flags分配15个大小为 N 的virtual column。这是对AIR约束个数的优化（因此，稍微的减少了verifier的时间），为了从 $\{\tilde{f}_i\}_{i=0}^{15}$ 中获取 f_i 的值，我们使用下列等式：

$$\tilde{f}_i - 2\tilde{f}_{i+1} = \sum_{j=i}^{14}(2^{j-i} \cdot f_j) - 2 \cdot \sum_{j=i+1}^{14}(2^{j-i-1} \cdot f_j) = \sum_{j=i}^{14}(2^{j-i} \cdot f_j) - \sum_{j=i+1}^{14}(2^{j-i} \cdot f_j) = f_i$$

所以，指令的unpacking约束如下：

Instruction： $inst = \widetilde{off}_{dst} + 2^{16} \cdot \widetilde{off}_{op0} + 2^{32} \cdot \widetilde{off}_{op1} + 2^{48} \cdot \tilde{f}_0$

Bit： $(\tilde{f}_i - 2\tilde{f}_{i+1})(\tilde{f}_i - 2\tilde{f}_{i+1} - 1) = 0 \ for \ all \ i \in [0, 15)$

Last value is zero： $\tilde{f}_{15} = 0$

Offset are in range：virtual column $\widetilde{off}_*$ （ * 代表 dst, op0, op1 中的一个）实际上是章节9.9描述的置换范围校验的一个subcolumn，使得 $\widetilde{off}_* \in [0, 2^{16})$ ，从而 $off_* \in [2^{-15}, 2^{15})$

定理1.（指令解码）考虑到上述关于 $inst, \widetilde{off}_{dst}, \widetilde{off}_{op0}, \widetilde{off}_{op1}, \{\tilde{f}_i\}_{i=0}^{15}$ 的值的约束，以及 $P > 2^{63}$ ，存在一个唯一的在范围 $[0, 2^{63})$ 整数 $inst_{\mathbb{Z}} \in \mathbb{Z}$ ，满足 $inst = inst_{\mathbb{Z}}$ (回顾 $inst \in \mathbb{F}$)，而且 $f_i = \tilde{f}_i - 2\tilde{f}_{i+1}$ 的值等于 $inst_{\mathbb{Z}}$ 的第（48 + i）位，并且 $\widetilde{off}_*$ 的值和 $inst_{\mathbb{Z}}$ 的第16 * i,...,16 * i + 15对应（dst, op0, op1分别对应i = 0,1,2）

证明：

$P > 2^{63}$ 保证了任何一个在 $[0, 2^{63})$ 的整数可以唯一表示一个域元素。

"Bit" 约束保证了 $\tilde{f}_i = 2\tilde{f}_{i+1} \ or \ \tilde{f}_i = 2\tilde{f}_{i+1} + 1$ ，通过归纳法，我们知道 $\tilde{f}_{15-i} \in [0, 2^i)$ ，因此 $\tilde{f}_0 \in [0, 2^{15})$ ，置换范围校验确保了 $\widetilde{off}_* \in [0, 2^{16})$ ，因此，我们有：

$$0 \le \widetilde{off}_{dst} + 2^{16} \cdot \widetilde{off}_{op0} + 2^{32} \cdot \widetilde{off}_{op1} + 2^{48} \cdot \tilde{f}_0 < 2^{63}$$

因此，指令约束证明了整数 $inst_{\mathbb{Z}} \in [0, 2^{63})$ 的存在，并满足所有需求。

9.5 更新 pc

指令验证的大部分约束都是相当直接的，章节9.10列出了所有的约束。一个例外就是pc寄存器的更新约束，本小节主要介绍这个约束推导过程。

正如我们在章节4.5见到的一样，pc有几种更新方式 - 常规更新（pc_update = 0），绝对/相对跳转（pc_update = 1, 2），条件跳转（pc_update = 4，这个过程也叫JNZ - Jump Non-Zero），为了更好的理解本小节，建议读者再去了解一下章节4.5所述的pc更新过程。

正如章节9.4描述的一样，对于pc的更新，我们有：

$$pc\_update = f_{PC\_JMP\_ABS} + 2 \cdot f_{PC\_JMP\_REL} + 4 \cdot f_{PC\_JNZ}$$

这意味着：

$$regular\_update = 1 - f_{PC\_JMP\_ABS} - f_{PC\_JMP\_REL} - f_{PC\_JNZ}$$

注意：我们需要假设 $regular\_update, f_{PC\_JMP\_ABS}, f_{PC\_JMP\_REL}, f_{PC\_JNZ}$ 中的其中一个为1，剩余的必须为0。

考虑以下约束：

$$\begin{aligned}&(1-f_{PC\_JNZ}) \cdot next\_pc- \\ &(regular\_update \cdot (pc + instruction\_size) + \\ &f_{PC\_JUMP\_ABS} \cdot res + \\ &f_{PC\_JUMP\_REL} \cdot (pc + res)) = 0\end{aligned} \tag{5}$$

注意，上述这个约束处理了所有的pc更新情况（除了JNZ的场景），在JNZ的场景下，如果 dst = 0，则pc更新方式为regular_update；否则，更新方式为相对跳转（位置根据op1的值）。

下面这个约束对应了dst != 0的场景：

$$f_{PC\_JNZ} \cdot dst \cdot (next\_pc-(pc + op1)) = 0$$

因为，我们更想用二阶的AIR（见章节9.3），我们分配了一个新的大小为N的virtual column，t_0，然后我们把上述约束做个变换：

$$\begin{aligned}t_0 &= f_{PC\_JNZ} \cdot dst \\ t_0 \cdot (next\_pc-(pc + op1)) &= 0\end{aligned} \tag{6}$$

为了验证当dst = 0时，pc做了一个常规更新，我们增加了一个辅助变量（一个大小为N的新的virtual column），v（当dst != 0 时，用 $v = dst^{-1}$ 来填充单元）：

$$f_{PC\_JNZ} \cdot (dst \cdot v - 1) \cdot (next\_pc-(pc + instruction\_size)) = 0$$

为了实现二阶要求，我们新增了另外一个virtual column，t_1；则上述约束可以重写为：

$$\begin{aligned}t_1 &= t_0 \cdot v \\ (t_1 - f_{PC\_INZ}) \cdot (next\_pc-(pc + instruction\_size)) &= 0\end{aligned}$$

继续对其使用两个优化作为最终的版本：

a. 注意，在处理jnz指令时，res没有被用到（要么是Unused，要么是Undefined）；因此，我们可以用它来存放 v 的值（就不用为 v 重新分配一个virtual column）；

b. 可以观察到，根据 f_{PC_INZ} 的取值，等式（5）和（6）左边必须有一个为0，因此可以把它们两个结合起来：

$$
\begin{aligned}
&t_0 \cdot (next\_pc - (pc + op1)) + \\
&(1 - f_{PC\_JNZ}) \cdot next\_pc - \\
&(regular\_update \cdot (pc + instruction\_size) + \\
&f_{PC\_JUMP\_ABS} \cdot res + \\
&f_{PC\_JUMP\_REL} \cdot (pc + res)) = 0
\end{aligned}
$$

9.6 置换和交互步骤

在轨迹单元方面，使Cairo高效所需的一个关键组件是在构建轨迹期间，验证者和验证者之间的额外交互步骤，这个结构是基于文章【9】里介绍的技术。

例如，假设我们想要证明轨迹列a的单元格与另一列b的单元格相同，直到某些排列。问题是这个属性不是local的，因此没有简单的AIR约束来强制它。

核心的思想是观察两个多项式 $f(X) = \prod_{i=0}^{n-1}(X - a_i)$ 和 $g(X) = \prod_{i=0}^{n-1}(X - b_i)$ 是相等的，当且仅当 a_i 和 b_i 是相同的值的集合。第二，注意如果两个多项式不相同，我们可以通过选择一个随机场元素 $z \in \mathbb{F}$ ，并将其代入两个多项式来观察到这一点。在高概率情况下，结果将不相同(假设多项式度比域大小小得多)。

为了把这个概念转换成AIR，我们引入了两个新的轨迹列： $c_j = \prod_{i=0}^{j}(z - a_i)$ 和 $d_j = \prod_{i=0}^{j}(z - b_i)$ ，分别代表f(z)和g(z)计算过程, 对于一个域元素 $z \in \mathbb{F}$ （下面将描述 z 是如何选取的）。约束将会保证累积计算过程的正确性，而且两个列的最后一个值c和d是相等的；这就使得verifier相信 $f(z) = g(z)$ 的，这意味着多项式 $f = g$ ，继而证明 a_i 和 b_i 具有相同的值的集合。

上面的构造有一个关键的问题:一方面，重要的是，在证明者承诺了a和b之后，验证者将(随机)选择 z (否则，上述概率论证失败)。但另一方面，它必须在承诺c和d(因为它们依赖于它)之前做出选择。在常规STARK协议中，这是不可能的，因为轨迹的承诺是在一个步骤中完成的

因此，**Cairo使用了一个稍微修改过的STARK版本，在这个版本中，prover首先提交一些列(在我们的例子中是a和b)，然后verifier生成一些随机数。现在，证明程序将提交给其余的列(c和d)，它们可能依赖于随机值。我们将此方法称为在验证者和验证者之间添加另一个交互步骤(验证者发送额外的随机数)。**

Cairo对其范围检查和内存使用了上面描述的排列检查的略有改进的版本，后面的小节会有所描述。

9.7 非确定性连续只读内存

9.7.1 定义

定义4. 一次内存访问是一个数值对 $(a,v) \in \mathbb{F}^2$ ，其中 a 代表地址， v 代表这个地址上的值。一系列的内存访问形成了一个只读内存， $(a_i, v_i)\ for\ i \in [0,n), n \in [1,P)$ ；对于所有的 $i,j \in [0,n)$ ，如果 $a_i = a_j$，则$v_i = v_j$ 。如果 $\{a_i, i \in [0,n)\}$ 和 $[m_0, m_1)$ 相等(这里 $[m_0, m_1)$ 的意思是 $\{m_0 + i : i = 0,1,...,t-1\}$)，其中 $m_1 = m_0 + t, t < P$ ，则说明内存是连续的。特别地，对于给定的连续只读内存访问列表，我们可以定义一个函数 $f : [m_0, m_1) \to \mathbb{F}$ ，满足 $f(a_i) = v_i\ for\ i \in [0,n)$ 。任何扩展了 f 的函数 $m : \mathbb{F} \to \mathbb{F}$ 都被称为内存访问列表的内存函数。

9.7.2 约束

定理 2. 令 $L_1 = \{a_i, v_i\}_{i=0}^{n-1}, L_2 = \{a_i^{'}, v_i^{'}\}_{i=0}^{n-1}$ 是两个内存列表， $\{p_i\}_{i=0}^{n-1}$ 是额外的域元素序列，基于两个选择的随机数 $z, \alpha \in \mathbb{F}$ ，以较大的概率满足以下约束：

Continuity(连续性)： $(a_{i+1}^{'} - a_i^{'}) \cdot (a_{i+1}^{'} - a_i^{'} - 1) = 0\ for\ all\ i \in [0, n-1)$

Single-valued(单值性)： $(v_{i+1}^{'} - v_i^{'}) \cdot (a_{i+1}^{'} - a_i^{'} - 1) = 0\ for\ all\ i \in [0, n-1)$

Permutation（排列性）：

Initial value（初始值）： $(z - (a_0^{'} + \alpha \cdot v_0^{'})) \cdot p_0 = z - (a_0 + \alpha \cdot v_0)$

Final value（最终值）： $p_{n-1} = 1$

Cumulative product step（累积乘法）：
$(z - (a_i^{'} + \alpha \cdot v_i^{'}) \cdot p_i = (z - (a_i + \alpha \cdot v_i) \cdot p_{i-1}\ for\ i \in [1,n)$

那么 L_1 是一段连续的只读内存。

证明：

首先，给出 L_2 是一段连续的只读内存；通过连续性约束 $a_{i+1}^{'} \in \{a_i^{'}, a_{i+1}^{'}\}$ ，所以 L_2 是连续的。有一点 L_2 是一段连续的只读内存，注意，如果 $a_i^{'} = a_j^{'}$ ，对于 $i < j$ ，则根据“连续性”约束对于所有 $i < k < j$ ，都有 $a_i^{'} = a_k^{'} = a_j^{'}$ 。 “单值性” 约束保证了 $v_i^{'} = v_{i+1}^{'} = v_{i+2}^{'} = ... = v_j^{'}$ 。

下一步，给出如下公式：

$$\prod_{i=0}^{n-1}(z-(a_i^{'}+\alpha\cdot v_i^{'}))=\prod_{i=0}^{n-1}(z-(a_i+\alpha\cdot v_i)) \quad (7)$$

对所有的 $n^{'}\in[0,n)$ ，归纳公式如下：

$$p_{n'}\cdot\prod_{i=0}^{n^{'}}(z-(a_i^{'}+\alpha\cdot v_i^{'}))=\prod_{i=0}^{n^{'}}(z-(a_i+\alpha\cdot v_i)) \quad (8)$$

对于 $n^{'}=0$ ，可以通过初始化的值来满足约束；对于 $n^{'}>0$ ，遵循“累积乘法”约束。对于公式（8）在 $n^{'}-1$ 的情况，两边同时乘 $(z-(a_{n'}+\alpha\cdot v_{v'}))$ ，然后把等式左边的 $(z-(a_{n'}+\alpha\cdot v_{n'}))\cdot p_{n'-1}$ 用 $(z-(a_{n'}^{'}+\alpha\cdot v_{n'}^{'}))\cdot p_{n'}$ 代替来获取公式（8）在 $n^{'}$ 的形式。还要替换 $n^{'}=n-1$ ，然后用“最终值”约束来获取公式（7）。

前面描述了如果 L_2 是一段连续的内存，那么等式（7）成立；下面的命题1暗示了， L_1 也构成了一段连续内存。

命题1. 令 $L_1=\{(a_i,v_i)\}_{i=0}^{n-1},L_2=\{(a_i^{'},v_i^{'})\}_{i=0}^{n-1}$ 是两个内存列表，其中 L_2 是一段连续的只读内存，用 ε 来表示事件 $\prod_{i=0}^{n-1}(z-(a_i^{'}+\alpha\cdot v_i^{'}))=\prod_{i=0}^{n-1}(z-(a_i+\alpha\cdot v_i))$ ，如果：

$$Pr_{\alpha,z\in\mathbb{F}}[\varepsilon]>\frac{n^2+n}{\mathbb{F}}$$

那么， L_1 也构成了一段连续内存。

证明：

$\varepsilon^{'}$ 表示事件：对于两个列表，函数 $(a,v)\to a+\alpha\cdot v$ 没有碰撞，即对于 i,j ，如果有 $a_i+\alpha\cdot v_i=a_j+\alpha\cdot v_j$ 则 $(a_i,v_i)=(a_j',v_j')$ 。通过联合约束，我们有

$Pr_{\alpha,z\in\mathbb{F}}[\overline{\varepsilon'}]<\frac{n^2}{\mathbb{F}}$ ，所以有： $Pr_{\alpha,z\in\mathbb{F}}[\varepsilon\cap\varepsilon']>\frac{n}{\mathbb{F}}$ 。

固定 α 满足 $Pr_{z\in\mathbb{F}}[\varepsilon\cap\varepsilon^{'}]>\frac{n}{\mathbb{F}}$ ，考虑以下多项式：

$$\prod_{i=0}^{n-1}(X-(a_i+\alpha\cdot v_i))-\prod_{i=0}^{n-1}(X-(a_i^{'}+\alpha\cdot v_i^{'}))$$

这个多项式至少有 $n+1$ 个零点（根据概率公式得到），因此，这是一个零多项式（因为零点个数，大于多项式阶数）。因为 $a_i+\alpha\cdot v_i$ 是左边乘积的根，比存在一个 j 使得 $a_i+\alpha\cdot v_i=a_j^{'}+\alpha\cdot v_j^{'}$ ，又因为 $(a,v)\to a+\alpha\cdot v$ 映射没有碰撞，因此有 $(a_i,v_i)=(a_j^{'},v_j^{'})$ ；从两个乘积中去除这两项，然后继续使用上述论证，最终得到 L_1 是 L_2 的一个置换。因为属性“连续只读内存”不依赖于列表的排序，因此 L_1 也是一段连续只读内存。

9.8 公共内存

仅仅验证 $L_1 = \{(a_i, v_i)\}_{i=0}^{n-1}$ 是一段只读的连续内存是不够的；论述的一部分是验证存在一个函数 m 是 m^* （针对公共输入的函数）的扩展。为此，人为的向列表 L_1 和 L_2 添加 $\{(a, m^*(a))\}_{a \in A^*}$ 是有必要的。可以通过为这些人为的访问添加额外的轨迹单元，并且约束 L_1 里的这些单元等于 $\{a, m^*(a)\}_{a \in A^*}$ ，这要花费 $2 \cdot |A^*|$ 个约束（对于 A^* 的每一个元素，分别对其地址和值进行约束）。

虽然这个解决办法有效，但是它非常低效，因为引入了大量的额外约束；相反，我们用假的内存访问（0,0）来替代 $|A^*|$ 次 L_1 内存访问（ L_2 的内存访问保持真的 $\{(a, m^*(a))\}_{a \in A^*}$ ），并且修改9.7.2的约束，把这些访问当做真实发生一样来处理 $\{(a, m^*(a))\}_{a \in A^*}$ 。仔细观察约束条件，我们看到唯一需要改变的是乘积的计算。从使用(0,0)访问计算的乘积移动到实际的乘积是相对简单的：

$$\frac{\prod_{a \in A^*}(z - (a + \alpha \cdot m^*(a))}{z^{|A^*|}}$$

因此，我们对于最后一个值的约束变为：

$$p_{n-1} = \frac{z^{|A^*|}}{\prod_{a \in A^*}(z - (a + \alpha \cdot m^*(a)))} \quad (9)$$

如果这个约束对于（0,0）的访问数据成立，则说明对于真实的访问，最终乘积值将是1。

这提供了一种非常有效的方法来处理Cairo程序的输出：对于输出中的每个域元素，验证器只需做两次加法和两次乘法(其中一次是为了计算Eq.(9)中的分母)。

9.9 置换范围校验

正如9.4节中所解释的，一些轨迹单元格应该被限制在 $[0, 2^{16})$ 的范围内。为了检查virtual column的值 $\{a_i\}_{i=0}^{n-1}$ 是否都在该范围内，我们使用了在9.7节中看到的相同技术，并进行了以下更改：

a. 保证集合 $\{a_i : i \in [0, n)\}$ 是连续的；为此，可能需要人为的添加轨迹单元，并填充它们以填补原始值列表中存在的漏洞；
b. 设置 $\{v_i = 0 \ for \ i \in [0, n)\}$ ，并添加约束来检查 $\{(a_i, v_i)\}_{i=0}^{n-1}$ 形成一个连续的只读内存。当然，有些约束可以简化(例如， α 不再需要了)，有些约束可以删除(例如“单值性”约束)；

c. 对 a' 的最小值和最大值进行约束：

i. **Min range-check value：** $a'_0 = rc_{min}$ ；

ii. **Max range-check value：** $a'_{n-1} = rc_{max}$ ；

其中， rc_{min}, rc_{max} 是对verifier公开的，verifier只需要校验 $0 \leq rc_{min} \leq rc_{max} < 2^{16}$ 。

9.10 约束列表

本小节对章节9做了总结，并列出了所有的AIR约束；我们从一个定义列表开始，以简化下面的约束。这些定义本身并不构成约束。

- **Definition**

$$off_* = \widetilde{off}_* - 2^{15}, (* \in (dst, op0, op1))$$

$$f_i = \tilde{f}_i - 2\tilde{f}_{i+1}$$

$$\begin{aligned}
&f_{DST_REG} = f_0, && f_{PC_JUMP_REL} = f_8 \\
&f_{OP0_REG} = f_1, && f_{PC_JNZ} = f_9 \\
&f_{OP1_IMM} = f_2, && f_{AP_ADD} = f_{10} \\
&f_{OP1_FP} = f_3, && f_{AP_ADD1} = f_{11} \\
&f_{OP1_AP} = f_4, && f_{OPCODE_CALL} = f_{12} \\
&f_{RES_ADD} = f_5, && f_{OPCODE_RET} = f_{13} \\
&f_{RES_MUL} = f_6, && f_{OPCODE_ASSERT_EQ} = f_{14} \\
&f_{PC_JUMP_ABS} = f_7 && \\
&instruction_size = f_{OP1_IMM} + 1 &&
\end{aligned}$$

- **Instruction unpacking (21 trace cells)**

$$inst = \widetilde{off}_{dst} + 2^{16} \cdot \widetilde{off}_{op0} + 2^{32} \cdot \widetilde{off}_{op1} + 2^{48} \cdot \tilde{f}_0$$

$$f_i \cdot (f_i - 1) = 0, \ for\ all\ i \in [0, 15)$$

$$\tilde{f}_{15} = 0$$

- **Operand constraints (7 trace cells)**

$$dst\_addr = f_{DST\_REG} \cdot fp + (1 - f_{DST\_REG}) \cdot ap + off_{dst}$$
$$op0\_addr = f_{OP0\_REG} \cdot fp + (1 - f_{OP0\_REG}) \cdot ap + off_{op0}$$
$$op1\_addr = f_{OP1\_IMM} \cdot pc + f_{OP1\_AP} \cdot ap + f_{OP1\_FP} \cdot fp + (1 - f_{OP1\_IMM} - f_{OP1\_AP} - f_{OP1\_FP}) \cdot op0 + off_{o}$$

- **The ap and fp registers (9 trace cells)**

$$next\_ap = ap + f_{AP\_ADD} \cdot res + f_{AP\_ADD1} \cdot 1 + f_{OPCODE\_CALL} \cdot 2$$
$$next\_fp = f_{OPCODE\_RET} \cdot dst + f_{OPCODE\_CALL} \cdot (ap + 2) + (1 - f_{OPCODE\_RET} - f_{OPCODE\_CALL}) \cdot f$$
$$ap_0 = fp_0 = ap_I$$
$$ap_T = ap_F$$

- **The pc register (7 trace cells)**

$$t_0 = f_{PC\_JNZ} \cdot dst$$

$$t_1 = t_0 \cdot res$$
$$(t_1 - f_{PC\_INZ}) \cdot (next\_pc - (pc + instruction\_size)) = 0$$

$$t_0 \cdot (next\_pc - (pc + op1)) +$$
$$(1 - f_{PC\_JNZ}) \cdot next\_pc -$$
$$((1 - f_{PC\_JUMP\_ABS} - f_{PC\_JUMP\_REL} - f_{PC\_JNZ}) \cdot (pc + instruction\_size) +$$
$$f_{PC\_JUMP\_ABS} \cdot res +$$
$$f_{PC\_JUMP\_REL} \cdot (pc + res)) = 0$$

$$pc_0 = pc_I$$
$$pc_T = pc_F$$

- **Opcodes and res (2 trace cells)**

$$mul = op_0 \cdot op_1$$
$$(1 - f_{PC\_JNZ}) \cdot res = f_{RES\_ADD} \cdot (op_0 + op_1) + f_{RES\_MUL} \cdot mul + (1 - f_{RES\_ADD} - f_{RES\_MUL} - f_{PC\_JNZ}) \cdot o$$
$$f_{OPCODE\_CALL} \cdot (dst - fp) = 0$$
$$f_{OPCODE\_CALL} \cdot (op0 - (pc + instruction\_size)) = 0$$
$$f_{OPCODE\_ASSERT\_EQ} \cdot (dst - res) = 0$$

- **Memory (3 trace cells)**

这里 a^m, v^m 是两个virtual column，其具有以下subcolumns对：1.（pc, inst）2.（dst_addr, dst）3.（opp0_addr, op0）4.（opp1_addr, op1）。 如果AIR使用了builtin，每个

builtin可能会使用额外的subcolumns对；另外一个subcolumns对是针对公共内存机制（见章节9.8），对于这些单元，我们约束其所有的地址和值都是0。virtual column a^m, v^m 的大小取决于所需要的subcolumns的总和； a'^m, v'^m 是和 a^m, v^m 大小样的virtual column，里面存放的是排序后的内存访问列表。

$$(a'^m_{i+1} - a'^m_i) \cdot (a'^m_{i+1} - a'^m_i - 1) = 0 \ for\ all\ i \in [0, n-1)$$

$$(v'^m_{i+1} - v'^m_i) \cdot (a'^m_{i+1} - a'^m_i - 1) = 0 \ for\ all\ i \in [0, n-1)$$

$$(z^m - (a'^m_0 + \alpha \cdot v'^m_0)) \cdot p^m_0 = z^m - (a^m_0 + \alpha \cdot v^m_0)$$

$$p^m_{n^m-1} = \frac{z^{|A^*|}}{\prod_{a^m \in A^*}(z - (a^m + \alpha \cdot m^*(a^m)))}$$

$$(z^m - (a'^m_i + \alpha \cdot v'^m_i) \cdot p^m_i = (z^m - (a^m_i + \alpha \cdot v^m_i)) \cdot p^m_{i-1} \ for\ all\ i \in [0, n)$$

- **Permutation range-checks (3 trace cells)**

这里， a^{rc} 是一个virtual column，其subcolumn有： $\widetilde{off}_{dst}, \widetilde{off}_{op0}, \widetilde{off}_{op1}$ （至少 $3T$ 个值，伴有额外Unused 单元，用于填充）。

$$(a'^{rc}_{i+1} - a'^{rc}_i) \cdot (a'^{rc}_{i+1} - a'^{rc}_i - 1) = 0 \ for\ all\ i \in [0, n-1)$$

$$(z^{rc} - a'^{rc}_0) \cdot p^{rc}_0 = z^{rc} - a^{rc}_0$$

$$p^{rc}_{n^{rc}-1} = 1$$

$$(z^{rc} - a'^{rc}_i) \cdot p^{rc}_i = (z^{rc} - a^{rc}_i) \cdot p^{rc}_{i-1} \ for\ all\ i \in [0, n-1)$$

$$a'^{rc}_0 = rc_{min}$$

$$a'^{rc}_{n^{rc}-1} = rc_{max}$$

References

- [1] [Online]. Available: https://zkp.science/
- [2] [Online]. Available: https://medium.com/starkware/hello-cairo-3cb43b13b209
- [3] [Online]. Available: https://hackmd.io/@aztec-network/plonk-arithmetiization-air
- [4] [Online]. Available: https://github.com/starkware-libs/cairo-lang
- [5] [Online]. Available: https://cairo-lang.org/docs
- [6] [Online]. Available: https://leanprover.github.io/

- [7] J. Avigad, L. Goldberg, D. Levit, Y. Seginer, and A. Titleman, "A verified algebraic representation of cairo program execution," 2021, preprint.
- [8] L. Babai, L. Fortnow, and C. Lund, "Non-deterministic exponential time has two-prover interactive protocols," Computational Complexity, vol. 1, pp. 3–40, 1991, preliminary version appeared in FOCS ' 90.
- [9] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2012, pp.263–280.
- [10] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," IACR Cryptology ePrint Archive, vol. 2018, p. 46, 2018.
- [11] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in Proceedings of the 33rd Annual International Cryptology Conference, ser. CRYPTO ' 13, 2013, pp. 90–108.
- [12] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for R1CS," IACR Cryptology ePrint Archive, vol. 2018, p. 828, 2018.
- [13] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct noninteractive zero knowledge for a von neumann architecture," in 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, pp.781–796.
- [14] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, 2018, pp. 315–334.
- [15] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques, ser. EUROCRYPT ' 13, 2013, pp. 626–645.
- [16] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: Interactive proofs for Muggles," in Proceedings of the 40th Annual ACM Symposium on Theory of Computing, ser. STOC ' 08, 2008, pp. 113–122.
- [17] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," SIAM Journal on Computing, vol. 18, no. 1, pp. 186–208, 1989, preliminary version appeared in STOC ' 85.
- [18] A. E. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA. IEEE Computer Society, 2018, pp. 944–961. [Online]. Available: https://doi.org/10.1109/SP.2018.00018

- [19] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan, “Algebraic methods for interactive proof systems,” Journal of the ACM, vol. 39, no. 4, pp. 859–868, 1992.
- [20] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, “ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash,” in 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings. USENIX Association, 2010, pp. 193–206. [Online]. Available: http://www.usenix.org/events/sec10/tech/full_papers/Meiklejohn.pdf
- [21] N. Osaka, “A versatile memory-mapped i/o interface for microcomputers,” Behavior Research Methods & Instrumentation, vol. 13, no. 6, pp. 727–731, 1981.
- [22] B. Parno, J. Howell, C. Gentry, and M. Raykova, “Pinocchio: Nearly practical verifiable computation,” in 2013 IEEE Symposium on Security and Privacy, 2013, pp. 238–252.
- [23] P. Valiant, “Incrementally verifiable computation or proofs of knowledge imply time/space efficiency,” in Theory of Cryptography Conference. Springer, 2008, pp. 1–18.