zirgen总结

1. 入门指南

我们目前没有以任何打包形式发布 Zirgen，因此它只能通过此存储库获得。

```
https://github.com/risc0/zirgen.git
```

假设您之前已经从此存储库克隆并构建了内容，使用 Bazel 构建 Zirgen 非常简单，只需使用以下命令即可。请注意，这并不是绝对必要的，Bazel 也会在您使用它来调用测试时自动（重新）构建 Zirgen。

先安装bazelisk

```
brew install bazelisk
```

```
bazel build //zirgen/dsl:zirgen
bazelisk build //zirgen/dsl:zirgen
```

Hello world!

追随我们的脚步，让我们来看一个经典的 "Hello world" 程序。这个程序已经作为测试和示例提供，因此我们可以使用以下命令从此存储库的根目录运行它：

```
$ bazel run //zirgen/dsl:zirgen -- $(pwd)/zirgen/dsl/test/hello_world.zir --
test
...
Running 0
[0] Log: Hello world!
Lookups resolved
```

此命令向 Zirgen 可执行文件传递了两个参数。第一个 $(pwd)/zirgen/dsl/test/hello_world.zir 指定了我们要运行的 Zirgen 文件的路径。第二个 --test 指定我们要运行我们正在运行的文件中定义的所有测试。通常，Zirgen 的输出是一个生成的 Rust 或 C++ 库，然后需要与 RISC Zero 证明系统集成。为了简

化这里的操作并作为开发过程中的一种有用实践，最简单的方法是通过在电路代码旁边编写测试来试验 Zirgen，这些测试可以使用 --test 选项在内置解释器中运行，而无需进行此集成工作。现在，文件的重要部分如下：

```
test {
  Log("Hello world!");
}
```

关键字 test 声明后面（用大括号括起来的内容）是一个测试。测试可以有一个可选名称，但如果没有命名，则会用顺序编号标记。这会导致文本 "Running 0" 被写入标准输出，标记该测试的执行开始。语句 Log("Hello world!"); 是导致文本 "[0] Log: Hello world!" 被写入标准输出的原因。

2. 概念概述

现在我们已经看到了并运行了 "hello world"，我们可以开始深入研究 Zirgen 的实质。虽然该语言确实借鉴了命令式、函数式和逻辑编程的思想，但构建算术电路的挑战足够奇特，以至于需要一种专门的抽象模型。RISC Zero 证明系统是基于 STARK 的，因此证明系统的电路基本上需要推理有限域元素网格（witness）之间的多项式约束，以及基于程序输入和witness中的其他值填充它们的过程（witness生成）。 虽然原则上 STARK 的witness生成可以按任何顺序填充值，但 Zirgen 施加了所有列在移动到下一行之前填充的限制。这样，可以将 Zirgen 程序视为生成执行轨迹下一行的过程。这为执行轨迹创建了一个时间隐喻(metaphor)，我们将 STARK witness的行称为 "周期cycles"，类似于微处理器的时钟周期。计算的 "持续时间"（周期数）在运行时可配置，并由证明者在证明开始前确定。 因此，一个特定的电路负责定义以下几点：

- witness中有多少列？How many columns are there in the witness?
- 列之间存在哪些约束？What constraints exist between the columns?
- witness是如何填充的？How is the witness populated?

2.1 组件Components

Zirgen 中的核心抽象单位是组件。基本上，组件是这三个问题的答案。整个电路是一个称为 Top 的特殊组件（类似于 C/Rust 中的 main 函数），它必须是自包含的，但构成它的较小组件不必如此。这意味着组件可以向它没有分配的列添加约束，而这些列是由另一个组件分配的。组件还可以存储不属于witness的中间计算值。

2.2 Val

Val 组件类型表示有限域(finite field)元素(element)。值得注意的是，Val 不需要写入witness，而是作为电路执行过程中所有计算的 "中间值"。因此，Val 不会在witness中分配任何列，不会生成任何约束，对witness没有影响。它纯粹是一个存在于电路执行 "旁边on the side" 的值。

2.3 寄存器

现在，为了实际将值放入witness中，我们引入了寄存器的概念：寄存器是一个分配并填充witness中单列的组件。 Zirgen 中最基本的寄存器类型是 NondetReg 组件，它分配单列，用给定值写入，并不施加任何约束。"nondet" 是 "nondeterministic"（不确定性）的缩写，在此上下文中意味着该值是不受约束的，因此证明者原则上可以为其分配任何值。 另一种非常常见的寄存器类型是 Reg 组件，它也是内置于语言中的。它被定义为一个 NondetReg，另外还约束该列的值等于传递给构造函数的值

```
component Reg(v: Val) {
   reg := NondetReg(v);
   v = reg;
   reg
}
```

有关不确定性的更详细解释，请参见 Cairo paper 的第 2.5 节。

3. 约束

约束描述了列值之间必须满足的条件，如果证明者创建了有效的证明。这些约束必须是常量和witness中记录的值上的多项式方程。约束方程必须是多项式，因此必须在没有除法的情况下表达，并且由于 RISC Zero 证明系统的当前参数，不能包含次数大于 5 的项。例如，可以使用以下组件将值约束为 0 或 1：

```
component IsBit(v: Val) {
  v * (v - 1) = 0;
}

test zero_is_bit {
  r := NondetReg(0);
  IsBit(r);
}

test one_is_bit {
  r := NondetReg(1);
  IsBit(r);
}

test_fails two_is_not_bit {
```

```
    r := NondetReg(2);
    IsBit(r);
}
```

Supercomponents超组件

通常，能够以相同方式使用多个相关类型是有用的。例如，乘法定义为任意两个 Val，但在编写电路时，通常希望乘以寄存器中的值。现在，如果有一个 r: Reg，可以写 3 * r.reg，但这过于冗长。相反，最好允许寄存器像 Val 一样使用，将其视为从witness中读取的值，因此可以改写为 3 * r。Zirgen 通过超组件的概念实现了这种子类型关系 --- 每个组件都有一个超组件（有一个例外），任何组件都可以隐式转换为其超组件的类型，或其超组件的超组件，递归到称为 Component 的简单组件，这是唯一没有超组件的组件。因此，可以将组件的 "超组件链" 定义为特定组件可以转换为的类型序列。例如，Reg 的超组件链是 Reg -> NondetReg -> Val -> Component。

4. 构建一个斐波那契*Fibonacci*电路

让我们通过从头到尾构建一个完整的电路来直接开始吧！我们将构建一个基于我们网站上的 STARK by Hand 文档中的斐波那契电路的电路，这将带我们了解许多基本的语言特性以及如何将它们与 RISC Zero 证明系统集成。

4.1 Setting up the project

首先，我们需要为我们的电路创建一个新的 Zirgen 文件。Zirgen 文件通常以 .zir 为扩展名，本示例的最终源代码位于 /zirgen/dsl/examples/fibonacci.zir。现在，我们可以在该文件中创建一个不做任何事情但满足编译器要求的简单 Top 组件：

```
component Top() {}
```

接下来，我们需要告诉 Bazel 关于我们的新电路，并告诉它如何构建我们的项目。在与我们的新电路源文件相同的目录中创建一个 Build.bazel 文件，内容如下

```
package(
    default_visibility = ["//visibility:public"],
)

load("//bazel/rules/zirgen:dsl-defs.bzl", "zirgen_genfiles")

zirgen_genfiles(
    name = "FibonacciIncs",
    zir_file = ":fibonacci.zir",
    zirgen_outs = [
```

```
        (
            ["--emit=rust"],
            "fibonacci.rs.inc",
        ),
    ],
)
```

这会添加一个名为 GenerateFibonacciIncs 的新的 Bazel 构建命令，该命令生成与电路相关的 Rust 库代码。如果您现在运行它，它将在 fibonacci.zir 旁边创建一个名为 fibonacci.rs.inc 的文件。对于我们在 examples 目录中的电路代码，我们可以这样生成它：

```
bazel run //zirgen/dsl/examples:GenerateFibonacciIncs
```

现在该库代码实际上并没有做任何事情，因为电路还没有做任何事情，但在教程结束时我们会讨论生成代码中的内容以及如何将其与证明系统集成。我们将在没有它的情况下进行大部分电路开发周期，使用 Zirgen 测试和解释器。以下命令（将 fibonacci.zir 的路径替换为项目中的正确路径）应quietly succeed：

```
bazel run //zirgen/dsl:zirgen -- $(pwd)/zirgen/dsl/examples/fibonacci.zir --
test
```

4.2 Building the circuit

4.2.1 Inputs and outputs

现在一切都设置好了，我们可以开始构建我们的电路了！让我们从设置输入和输出开始。我们的电路将有 3 个输入：斐波那契序列的两个初始项 f0 和 f1，以及要计算的步数 steps。它还将有一个输出：steps 步后的斐波那契序列的第 steps 项 f_last。这些输入都不需要是秘密的，所以我们将它们全部设为在电路外部定义的全局常量。它们是全局的，因为任何带有 global 关键字的声明或定义都将引用相同的组件，并且它们是常量的，因为它们在电路的整个执行过程中只有一个值，所以它们的值不会从一个周期到另一个周期发生变化。实际上，它们存在于witness之外，并最终向验证者公开，所以它们是公开的。

```
component Top() {
  global f0: Reg;
  global f1: Reg;
  global steps: Reg;
}
```

4.2.2 Handling our base case

现在让电路只做一个计算周期。在第一个周期，我们将输入值移动到witness的列中并计算下一个项。将以下代码添加到 Top 组件的末尾：

```
// Copy global inputs into the witness
d1 := Reg(f0);
d2 := Reg(f1);

// Compute the next Fibonacci term
d3 := Reg(d1 + d2);

// Write the next term as the output
global f_last := Reg(d3);
```

我们还可以编写一个测试，为这些全局变量提供值并检查输出：

```
test FirstCycle {
  // Supply inputs
  global f0 := Reg(1);
  global f1 := Reg(2);
  global steps := Reg(1);

  top := Top();

  // Check the output
  global f_last : Reg;
  f_last = 3;
}
```

使用上述命令运行此测试，我们应该看到它通过了！这意味着 f_last 被设置为预期的 3。

4.2.3 Two cycles and beyond!两个周期及以后

现在，为了进行多个周期，我们需要做更多的工作。在第一个周期，我们需要将初始项复制到witness中。但对于所有后续周期，我们希望从前一个周期计算后续的斐波那契项，因此我们需要一种机制来区分第一个周期和其他周期。方便的是，我们可以使用证明者提供的外部组件（extern）查询当前的周期号，只需在使用前声明它：

```
extern GetCycle() : Val;
```

这表示 GetExtern 是一个构造函数不带参数的组件，并且其超组件类型为 Val。extern 关键字表示实现来自证明者，这不幸意味着恶意的证明者可以使用不同的实现来欺骗验证者，因此我们需要做更多的工作来正确约束来自它的值。我们将把它封装在一个新组件中：

```
component CycleCounter() {
  global total_cycles := NondetReg(1);

  cycle := NondetReg(GetCycle());
  is_first_cycle := IsZero(cycle);

  [is_first_cycle, 1-is_first_cycle] -> ({
    // First cycle; previous cycle should be the last cycle.
    cycle@1 = total_cycles - 1;
  }, {
    // Not first cycle; cycle number should advance by one for every row.
    cycle = cycle@1 + 1;
  });
  cycle
}
```

这里有很多内容，让我们分解一下。首先，我们引入一个新的全局变量，用于存储witness中的总周期数。看起来这可能是证明者通过不匹配总周期数来破坏事情的地方，但请记住，验证者知道所有全局变量，并将负责稍后检查这一点。其次，我们使用我们的 GetCycle extern 查询证明者的周期号，并将其记录到一个寄存器中。现在剩下的就是引入约束：我们希望周期号从 0 开始，每个周期增加 1。因此，如果周期号为 0，我们将前一个周期号（循环回witness的末尾）约束为总周期数减 1（因为我们从 0 开始），对于所有其他周期，我们将当前周期号约束为前一个周期号加 1。确定周期号是否为 0 是使用 IsZero 组件完成的，我们将在稍后定义，然后我们使用一个多路复用器根据当前周期是否为 0 引入我们的两个约束之一。最后，在最后一行中，我们将 cycle 声明为我们的 CycleCounter 组件的超组件，以便它可以被强制转换为 NondetReg 和 Val。 继续 IsZero！Zirgen 定义了一些我们将在这里使用的内置组件：Isz，如果其参数为 0，则为 1，如果其参数为非零，则为 0，以及 Inv，计算其参数的乘法逆元，如果其参数为 0，则为 0。这些都不会创建任何约束，因此与 GetCycle 一样，我们也需要正确约束这些值。事不宜迟：

```
component IsZero(val: Val) {
  // Nondeterministically 'guess' the result
  isZero := NondetReg(Isz(val));

  // Compute the inverse (for non-zero values), for zero values, Inv returns 0
  inv := NondetReg(Inv(val));

  // isZero should be either 0 or 1
```

```
    isZero * (1 - isZero) = 0;
    // If isZero is 0 (i.e. nonzero) then val must have an inverse
    val * inv = 1 - isZero;
    // If isZero is 1, then val must be zero
    isZero * val = 0;
    // If isZero is 1, then inv must be zero
    isZero * inv = 0;
    isZero
  }
```

最好为新组件添加一个测试。重新运行测试，我们应该看到 IsZeroTest 和 FirstCycle 都通过了。

```
  test IsZeroTest {
    IsZero(0) = 1;
    IsZero(1) = 0;
    IsZero(2) = 0;
  }
```

现在我们可以完成 Top 的实现。首先，我们需要构建我们的周期计数器。它的构造函数不带参数，所以我们只需在全局声明之后添加这一行：

```
  cycle := CycleCounter();
```

接下来，我们需要根据是否在第一个周期调整 d1 和 d2 的值。方便的是，我们的周期计数器已经在一个正确约束的寄存器中有了这个值，所以我们可以用它来定义一个多路复用器选择器：

```
  d2 : Reg;
  d3 : Reg;
  d1 := Reg([cycle.is_first_cycle, 1 - cycle.is_first_cycle] -> (f0, d2@1));
  d2 := Reg([cycle.is_first_cycle, 1 - cycle.is_first_cycle] -> (f1, d3@1));
```

这里发生了什么？首先，我们前向声明 d2 和 d3 以便我们可以通过“backs”访问它们——这是我们如何引用它们在前一个周期的值。然后，我们使用一个多路复用器在 cycle.is_first_cycle 字段上计算 d1 和 d2 的值。回想一下，这个寄存器包含 1 如果周期是 0，否则包含 1。因此在第一个周期，选择器数组变为 [1, 0]，所以我们的多路复用器评估为它们的第一个分支（分别是 f0 和 f1），在所有其他周期中，选择器数组变为 [0, 1]，所以它们评估为它们的第二个分支（分别是 d2@1 和 d3@1）。 这段代码有点冗长，因为 cycle.is_first_cycle 被重复了四次。可能更清楚的是给它一个更短的名字。请注意，

尽管这个表达式的类型是 IsZero，因此为它分配了两个寄存器，但这样做只是“别名”组件而不引入任何新的列到witness中。我们可以这样重写：

```
first := cycle.is_first_cycle;
d2 : Reg;
d3 : Reg;
d1 := Reg([first, 1-first] -> (f0, d2@1));
d2 := Reg([first, 1-first] -> (f1, d3@1));
```

最后一项任务：对于我们计算的最后一步，我们需要将 d3 的值复制到我们的全局输出寄存器 f_last 中。请注意，我们只能写入全局寄存器一次，因为全局变量不是“每个周期”的，所以我们必须用另一个多路复用器来保护这个赋值。作为一个额外的调试功能，我们可以使用内置的 Log extern 记录这个赋值。当我们运行测试时，这会显示写入该全局变量的值以及执行的周期。

```
// If cycle = steps, write the next term to the output
terminate := IsZero(cycle - steps);
[terminate, 1 - terminate] -> ({
  global f_last := Reg(d3);
  Log("f_last = %u", f_last);
}, {});
```

所以 Top 的最终实现应该如下所示

```
component Top() {
  global f0: Reg;
  global f1: Reg;
  global steps: Reg;

  cycle := CycleCounter();
  first := cycle.is_first_cycle;

  // Copy previous two terms forward
  d2 : Reg;
  d3 : Reg;
  d1 := Reg([first, 1-first] -> (f0, d2@1));
  d2 := Reg([first, 1-first] -> (f1, d3@1));

  // Compute the next Fibonacci term
  d3 := Reg(d1 + d2);

  // If cycle = steps - 1, write the next term to the output
```

```
    terminate := IsZero(cycle - steps + 1);
    [terminate, 1 - terminate] -> ({
      global f_last := Reg(d3);
    }, {});
  }
```

Adding some multi-cycle tests添加一些多周期测试

TODO：周期计数器目前不起作用，因为解释器在第一个周期未能检查 cycle@1 = total_cycles - 1 约束，因为最后一个周期尚未填充！我暂时注释掉了这个约束，这会影响周期计数器的完整性。 现在我们可以编写一些跨多个周期执行的测试。为了做到这一点，我们只需要更改 steps 值，并约束写入 f_last 的值与预期值匹配。请注意，约束是每个周期检查的，所以我们需要保护这个约束只在终止周期或之后检查。这里有很多共享的结构，所以我们可以将测试用例抽象到一个新组件中！

```
  component FibTest(f0: Val, f1: Val, steps: Val, out: Val) {
    // Supply inputs
    global f0 := Reg(f0);
    global f1 := Reg(f1);
    global steps := Reg(steps);

    top := Top();

    // Check the output
    [top.terminate, 1 - top.terminate] -> ({
      global f_last : Reg;
      f_last = out;
    }, {});
  }

  test FirstCycle {
    FibTest(1, 2, 1, 3);
  }

  test SecondCycle {
    FibTest(1, 2, 2, 5);
  }

  test SixthCycle {
    FibTest(1, 2, 6, 34);
  }
```

使用 --test 调用 Zirgen 运行我们的电路只运行一个周期，但现在我们需要至少运行 6 个周期，以便我们的所有新测试都有意义！为此，我们只需添加 --test-cycles=6（或任何其他正整数）来设置运行测

试的周期数

```
bazel run //zirgen/dsl:zirgen -- $(pwd)/zirgen/dsl/examples/fibonacci.zir --test --test-cycles=6
```

5. 组件

因为本节中的思想对语言来说非常重要，所以我们已经在概念概述中触及了很多内容。本节将更深入地探讨具体细节，并更深入地介绍语法和语义。 从根本上说，组件由以下几部分定义：

1. 描述组件逻辑结构的“值”
2. 描述witness中列结构的“布局”
3. 有效witness必须满足的一组约束
4. 填充其列的算法

组件组合的能力是 Zirgen 的一个核心设计目标，因此它提供了一些内置的组件作为核心构建块。这些在Builtin Components中描述，本节中的示例假设对这些有基本的了解。就组件的组合方式而言，有两种基本机制：组件可以打包在一起形成具有命名和未命名字段的结构，这是本节讨论的主题；或者可以根据“选择器值”在多路复用器中从一组备选项中选择一个。这些与函数式编程和类型理论中常见的乘积类型和和类型密切相关。

5.1 类似结构的组件

为了构建我们自己的组件，我们需要为我们的新组件类型定义构造函数。这样的构造函数是用 component 关键字创建的，后跟一个标识符，该标识符命名新的组件类型，通常应以大写字母开头。然后它接受一个可选的由一个或多个类型注释的类型参数列表，用尖括号分隔，以及一个必需的由零个或多个类型注释的参数组成的括号列表。接下来是构造函数体，用大括号括起来。 让我们分解一下。这是最简单的可能组件，它不接受任何参数并且有一个空体

```
component MyTrivialComponent() {}
```

5.1.1 构造函数体Constructor Bodies

组件构造函数的主体由以分号终止的语句组成，这些语句之一是：

- 命名成员定义
- 匿名成员定义
- 约束

我们可以通过提供一个标识符，后跟一个定义等号 :=，然后是一个表达式，在我们的构造函数中定义一个命名成员：

```
component MyRegPair() {
  x := Reg(2);
  y := Reg(5);
}
```

5.1.2 超组件

TODO

5.1.3 构造函数参数

我们还可以像这样参数化组件构造函数，其中参数表示为标识符，后跟冒号，后跟类型名称，参数用逗号分隔

```
component Pair(x: Val, y: Val) {
  x := Reg(x);
  y := Reg(y);
}
```

5.1.4 Type Parameters类型参数

类型本身也可以是参数化的。类型参数的表示方式与构造函数参数相同，但在类型名称和参数列表之间用尖括号分隔。这些目前受到很大限制，因为它们的类型必须是 Val 或 Type（可以用类型名称实例化）。例如：

```
component ConstPair<X: Val, Y: Val>() {
  x := X;
  y := Y;
}
```

5.1.5 关于递归的说明A note about recursion

我们选择不允许组件递归包含具有相同构造函数的子组件——即 MyComponent<X: Val> 不能包含任何其他实例的 MyComponent，即使具有不同的类型参数。虽然这样的构造可能非常优雅，但任意递归在电路中存在某些理论上的困难，并且实现受限形式的递归会使编译器实现复杂化。在实践中，这对

于通常在算术电路中描述的计算类型来说是完全可以的，尽管将来可能会重新考虑这一设计决策。然而，目前只需注意以下内容不起作用，并且会导致编译时错误：

```
component Fib(n: Val) {
  isZeroOrOne := InRange(0, n, 2);
  [isZeroOrOne, 1 - isZeroOrOne] -> (
    1,
    Fib(n - 1) + Fib(n - 2)
  )
}
```

6. 复用器Muxes

除了上一节讨论的类似结构体的组件外，组合组件的另一种主要模式是复用器。类似于结构体是 A 和 B 和 C，复用器是 恰好一个 A 或 B 或 C，类似于其他语言中的和类型/联合/变体。我们称这些可能性为复用器的 臂，在执行轨迹的特定行上恰好有一个复用器的臂是 活动的。活动的复用器臂由一个称为 选择器 的字段元素数组确定，该数组必须可转换为 Array<Val, N>，其中 N 是臂的数量，并且必须是对应于活动臂的元素为1，所有其他臂为0。 复用器组件本身也是一个组件，因此我们需要给它一个合适的定义。

- 复用器的值表示是其臂的值表示的交错并集
- 复用器的布局是其臂的布局的交错并集，并具有本节稍后讨论的一些附加结构
- 复用器的约束集是其臂的约束乘以选择器。也就是说，对于具有选择器 s 的复用器，如果第 i 个臂有约束 $l_{i,j} = r_{i,j}$ ，则复用器有约束 $s[i] * (l_j - r_j) = 0$
- 复用器的见证生成算法检查选择器，并运行活动臂(active arm)的见证生成算法

6.1 语法Syntax

与在顶层声明的类似结构体的组件不同，复用器通过在其使用位置的表达式定义，具有特殊的语法，并且可以在允许表达式的任何地方使用。语法是选择器的表达式后跟 ->，然后是括号内的逗号分隔的臂的表达式列表。 复用器的一个简单应用是做类似于控制流的事情。例如，以下电路显示了一种模式，在名为 Init 的组件中在轨迹的“第一”行执行一些初始化，并在所有后续行中在名为 Step 的组件中执行一些计算步骤。

```
component CycleCounter() { ... }

component Top() {
  cycle := CycleCounter();

  [cycle.is_first, 1 - cycle.is_first] -> (
```

```
      Init(),
      Step()
    );
  }
```

该电路实例化了一个 CycleCounter 组件，该组件提供了一种可靠的方法来计数周期并区分轨迹的“第一”行（TODO：编写一个单独的文档来实现 CycleCounter）。然后，如果是第一个周期，则复用器表现为 Init 臂，否则表现为 Step 臂。实际上，有 if/else 子句的语法糖可能会使这看起来更熟悉。此代码与前面的示例完全相同：

```
component Top() {
  first := NondetReg(IsFirstCycle());

  if (first) {
    Init()
  } else {
    Step()
  }
}
```

6.2 超组件Super Components

我们刚才提到复用器是表达式，这引出了一个有趣的问题：复用器表达式的“类型”是什么？显然，如果所有臂具有相同的类型，则复用器应该能够用作该类型。禁止这样做会非常限制，实际上，以下代码根据 cond 的值记录“a 是 5”或“a 是 7”。

```
component Top() {
  ...
  a := if (cond) { 5 } else { 7 };
  Log("a is %u", a);
}
```

考虑一个更复杂的情况，例如以下情况，其中复用器臂具有略微不同的类型 NondetReg 和 Val。好吧，NondetReg 已经非常类似于 Val，因为你可以对其应用字段操作，将其传递给期望 Val 的构造函数，并以其他所有方式使用它。

```
component Top() {
  ...
  a := if (cond) { NondetReg(GetCycle()) } else { 0 };
  Log("a is %u", a);
```

```
5   }
```

这之所以有效，是因为 NondetReg 可以通过其超链转换为 Val。由于复用器必须恰好有一个活动臂，但通常任何一个都可能在轨迹的行上活动，因此复用器应该可以转换为其所有臂都可以的任何类型——并且由于类型可以转换为其超链中的任何类型，复用器的超类型是其臂的 最小公共超类型。 一组组件的最小公共超类型是该组所有成员的超链中出现的第一个类型。举几个例子，考虑这个类型层次结构：

```
1   graph TD;
2       Component --- A
3       A --- B
4       B --- D
5       B --- E
6       A --- C
7       C --- F
8       C --- G
```

以下都是正确的陈述：

- {B} 的最小公共超类型(The least common super)是 B
- {B, C} 的最小公共超类型是 A
- {D, E} 的最小公共超类型是 B
- {B, D} 的最小公共超类型是 B
- {B, F} 的最小公共超类型是 A

关于最小公共超类型的最后一点说明：“最小”一词是因为它是出现在类型层次结构图中“最低”的公共超类型。这与代数上组件类型的集合形成一个半格，最小公共超类型作为连接操作有关。

6.3 布局Layout

复用器的布局实际上是其活动臂的布局，但给它一些附加结构是有利的。由于复用器表达式的类型是其所有臂的最小公共超类型，因此通常需要引用其在前几个周期的值。复用器的最小公共超类型保证在所有臂中具有相同的布局，而所有其他由臂使用的寄存器可能在复用器臂之间重用。对这些寄存器的回溯没有保证；在指示的周期上读取这些寄存器会从未定义的寄存器中读取。 考虑以下示例。复用器的公共超类型是 Reg，因此每个复用器臂的最终 Reg 构造函数调用落在同一列中。然而，第一个复用器臂使用两个“临时”寄存器，x 和 y，而第二个只使用一个临时寄存器，z。编译器可以将 z 放在与 x、y 相同的列中，或者可能都不放。

```
1   component Top() {
2     ...
```

```
    [a, b] -> ({
      x := Reg(...);
      y := Reg(...);
      Reg(x * (x + 1) * y * (y + 1))
    }, {
      z : Reg;
      z := Reg(z@1 + 2); // Undefined behavior!
      Reg(z * z * z)
    })
    ...
  }
```

6.4 度数考虑Degree considerations

由于选择器被乘以其臂的所有约束，因此这些约束的度数会增加相应选择器的度数。因为电路中的约束度数最多为5，有时这是一个重要的编程考虑。考虑以下情况：

```
component Top() {
  x := Reg(...);
  y := Reg(...);
  z := Reg(...);

  // Ensure mux arms are mutually exclusive
  x * y * z = 0;
  x * y + y * z + x * z = 1;

  [x * y, y * z, x * z] -> ({
    (x - 1) * (x - 2) * (x - 3) = 0;
  }, {
    (y - 4) * (y - 5) * (y - 6) = 0;
  }, {
    (z - 7) * (z - 8) * (z - 9) = 0;
  })
}
```

尽管在复用器中使用的约束看起来是度数3，但由于它们出现在具有度数2选择器的复用器臂中，因此它们的度数增加到5。对于这个特定的电路，完整的约束集是：

$xyz = 0$ (degree 3)

$xy + yz + xz = 0$ (degree 2)

$xy(x-1)(x-2)(x-3) = 0$ (degree 5)

$yz(y-4)(y-5)(y-6) = 0$ (degree 5)

$xz(z-7)(z-8)(z-9) = 0$ (degree 5)

7. 内置组件Builtin Components

7.1 Val

Intuitively, a `Val` is a field element. Formally, its *value* is a representation of that field element, its *layout* is trivial since it need not be stored in a register, its *constraint set* is empty, and its witness generation algorithm is a no-op.

`Val`s can be represented syntactically with numeric literals like `42` (decimal), `0xFF` (hexadecimal), or `0b10101` (binary) notation, and since they are finite field elements they can be added, subtracted, multiplied, and divided according to the rules of modular arithmetic. These operations are themselves components that can be coerced to `Val`, and have "identifier" names (`Add`, `Sub`, `Mul`, `Div`) as well as the more familiar infix operators (`+`, `-`, `*`, `/`). For example, `3 * 4` is just syntactic sugar for `Mul(3, 4)`.

直观地说，Val 是一个域元素。形式上，它的 值 是该域元素的表示，它的 布局 是微不足道的，因为它不需要存储在寄存器中，它的 约束集 是空的，并且它的见证生成算法是无操作的。 Val 可以用数字字面量表示，如 42（十进制）、0xFF（十六进制）或 0b10101（二进制）表示，并且由于它们是有限域元素，它们可以根据模运算规则进行加、减、乘和除。这些操作本身就是可以强制转换为 Val 的组件，并且有“标识符”名称（Add、Sub、Mul、Div）以及更熟悉的中缀运算符（+、-、*、/）。例如，3 * 4 只是 Mul(3, 4) 的语法糖。

7.2 Add

```
1   component Add(a: Val, b: Val) : Val;
```

Add 的超类型是 Val，它是 a 和 b 的有限域和。利用语法糖，这也可以写成 a + b。

7.3 Sub

```
1   component Sub(a: Val, b: Val) : Val;
```

Sub 的超类型是 Val，它是 a 和 b 在有限域中的和的加法逆。利用语法糖，这也可以写成 a - b.

7.4 Mul

```
component Mul(a: Val, b: Val) : Val;
```

Mul 的超类型是 Val，它是 a 和 b 的有限域乘积。利用语法糖，这也可以写成 a * b。

7.5 Neg

```
component Neg(v: Val) : Val;
```

The super of `Neg` is a `Val` which is the additive inverse of `v` in the finite field. Taking advantage of syntactic sugar, this can also be written as `-v`.

Neg 的超类型是 Val，它是 v 在有限域中的加法逆。利用语法糖，这也可以写成 -v

7.6 Inv

```
component Inv(v: Val) : Val;
```

The super of `Inv` is a `Val` which is the multiplicative inverse of `v` in the finite field. This computation is nondeterministic, so to constrain it you might do the following:

Inv 的超类型是 Val，它是 v 在有限域中的乘法逆。这种计算是非确定性的，因此为了约束它，你可以这样做：

```
vInv := NondetReg(Inv(v));
v * vInv = 1;
```

7.7 BitAnd

```
component BitAnd(a: Val, b: Val) : Val;
```

The super of `BitAnd` is a `Val` which is the bitwise and of `a` and `b`, treating them as integers. This computation is nondeterministic.

BitAnd 的超类型是 Val，它是 a 和 b 的按位与，将它们视为整数。这种计算是非确定性的。

7.8 InRange

```
component InRange(a: Val, b: Val, c: Val) : Val;
```

The super of `InRange` is a `Val` which is 1 if, treating the arguments as integers, `a <= b < c`, and 0 otherwise. However, if `a > c` then it causes a runtime error. This computation is nondeterministic.

InRange 的超类型是 Val，如果将参数视为整数，a <= b < c 则为 1，否则为 0。但是，如果 a > c 则会导致运行时错误。这种计算是非确定性的。

7.9 NondetReg

```
component NondetReg(v: Val);
```

Intuitively, a `NondetReg` is a `Val` that is recorded in the witness. Formally, its *value* is a structure containing `v`, its *layout* is a single column, its *constraint set* is empty, and its witness generation algorithm writes the given value into its column.

直观地说，NondetReg 是一个记录在见证中的 Val。形式上，它的 值 是包含 v 的结构，它的 布局 是单列的，它的 约束集 是空的，并且它的见证生成算法将给定值写入其列中。

7.10 Reg

A `Reg` is a wrapper around a `NondetReg`, which additionally adds a constraint to that column to be equal to the given value. It's definition is exactly:

Reg 是 NondetReg 的包装器，另外在该列中添加了一个约束，使其等于给定值。它的定义如下：

```
component Reg(v: Val) {
    reg := NondetReg(v);
    v = reg;
    reg
}
```

As a rule of thumb, if `v` is computed as a polynomial of constants and other registers, it's probably better to use `Reg` than `NondetReg`. `Val`s computed nondeterministically or returned by externs can't be used in constraints without registerizing, so use `NondetReg` for such values.

作为经验法则，如果 v 是作为常量和其他寄存器的多项式计算出来的，使用 Reg 可能比使用 NondetReg 更好。非确定性计算的 Val 或由外部返回的 Val 不能在没有寄存器化的情况下用于约束，

因此对于此类值使用 NondetReg。