使用时间线类型的模块化硬件设计

RACHIT NIGAM，美国康奈尔大学PEDRO HENRIQUE AZEVEDO DE AMORIM，美国康奈尔大学ADRIAN SAMPSON，美国康奈尔大学

模块化设计是实现硬件模块大规模复用的关键挑战。然而，与软件不同，硬件设计对应于物理电路，并继承了物理电路的约束。时序约束——信号在哪个周期到达、何时读取输入——以及结构约束——乘法器多久接受一次新输入——是硬件接口的基本组成部分。现有的硬件设计语言无法对这些约束进行编码；用户必须查阅文档、构建脚本，甚至在最糟糕的情况下阅读模块的实现，才能了解如何使用该模块。我们提出了 Filament，这是一种用于模块化硬件设计的语言，支持对静态调度流水线的时序约束和结构约束进行规范与强制执行。Filament 使用时间线类型来描述给定信号可用或必需的时钟周期时间区间。Filament 支持硬件模块的安全组合，确保生成的设计正确地进行流水线化，并可预测地将其降低为高效硬件。

1 引言

与软件语言类似，硬件描述语言（HDL）中的接口通常只由参数和简单数据类型组成。然而，与软件不同，硬件受底层物理电路的约束——输入仅在特定周期内使用，输出也仅在特定周期内可用；只有当电路能够处理新输入时，才能提供这些输入。现有硬件描述语言的接口过于简陋，无法体现这些约束，因而难以进行模块化设计。为了实现类似软件库生态系统的可复用性，硬件描述语言需要一种系统化的方法，将这些硬件模块的要求编码起来。

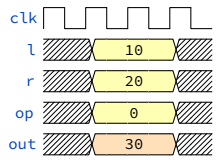
现代硬件设计语言分为三类。嵌入式硬件描述语言（eHDL）使用软件宿主语言进行元编程 [4, 16, 29, 34, 41]。加速器设计语言（ADL） [22, 25, 26, 31, 38, 51] 是更高层次的语言，能够提供新的抽象，并编译为硬件描述语言。最后，传统硬件描述语言（如 SystemVerilog 和 VHDL）是硬件设计以及与硬件工具链交互的事实标准。ADL 和 eHDL 必须编译为硬件描述语言，才能与硬件设计工具链交互，并集成经过优化的原语的专有黑盒实现。此外，ADL 通常仅限于单一领域，例如图像处理，因此可以通过定义外部函数接口，与 eHDL 和其他 ADL 交互，从中受益。

在 HDL 中，组合具有挑战性，因为接口仅公开输入输出端口的名称和值类型。然而，高效集成需要了解时序行为：分别产生输出和接收输入所需的周期数，以及模块是否可以流水线化。在当前的 HDL 中，这些时序信息是隐含的。它们可能出现在冗长的文档文件中，作为约束编码在构建脚本 [47], 中，或者完全不存在，因此用户必须阅读每个模块的实现，才能了解如何使用它。另一种方法是仅依赖延迟无关接口，这类接口消除了所有时序敏感性。相反，生产者会指示其输出何时有效，而消费者会指示其何时准备好接受新输入。尽管延迟无关接口很灵活，但效率也较低， [37], 尤其是对于始终需要相同数量时钟周期来产生输出并接受新输入的静态调度模块。

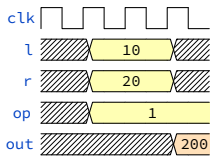
可复用硬件生态系统的关键，是一种低层机制，能够安全地且高效地组合硬件模块。安全组合要求模块指定并检查延迟、可流水线化等时序细节，而效率要求接口不会带来大量开销。效率要求还排除了对静态调度模块进行封装的做法。

```
module Add(a: 32, b: 32) -> (o: 32);
module Mul(a: 32, b: 32) -> (o: 32);
module Mux(sel: 1, a: 32, b: 32) -> (o: 32);
module ALU(op: 1, l: 32, r: 32) -> (o: 32) {
 Mul M(l, r); Add A(l, r);
 Mux Mx(op, A.out, M.out); o = Mx.out; }
```

(a) 算术逻辑单元的硬件描述语言实现



(b) 加法



(c) 乘法

Fig. 1. 执行加法和乘法时生成的算术逻辑单元实现及波形 n.

使用延迟无关接口；相反，我们希望使用时钟信号来同步模块的使用。这样就能清晰地集成硬件，无论它是用嵌入式硬件描述语言编写的、由架构描述语言生成的，还是作为专有黑盒模块实现的。

我们的解决方案是时间线类型，它能够紧凑地编码静态调度硬件流水线的延迟和吞吐量属性。静态流水线具有与数据无关的时序行为，涵盖了很大一类高效的硬件设计 [22, 25, 26, 30] ，包括大多数高级综合（HLS）工具生成的流水线 [11, 38, 42, 51]。我们的类型系统受到分离逻辑的启发 [46], ，通过确保满足所有时序约束，证明模块的流水线执行是安全的。我们的贡献如下：

- 我们刻画了静态流水线的流水线化约束，并在名为 Filament 的硬件描述语言中使用时间线类型对其进行建模。
- 我们使用硬件的基于日志的语义对这些流水线化约束进行形式化，并证明我们的类型系统相对于该模型是可靠的。
- 我们展示了 Filament 可以使用时间线类型集成来自多个硬件生成器的设计 [22,30, 49] 。
- 我们表明，与硬件生成器生成的设计相比，Filament 设计使用的资源更少，运行频率更高。

2 示例

我们将通过实现一个流水线化的算术逻辑单元（ALU），探讨组合式硬件设计所面临的挑战。

2.1 传统硬件描述语言

图 1a 展示了使用传统硬件描述语言实现的算术逻辑单元。模块的接口指定了输入和输出及其位宽。算术逻辑单元的电路由加法器和乘法器组成，它们并行执行计算；还包括一个多路复用器，它使用 op 信号。

我们将使用波形图来理解该模块的执行行为。波形图说明电路中信号随时间的变化，通常以全局时钟信号为参照。图 1b 展示了向算术逻辑单元提供输入 10 和 20 以及操作码 0 时生成的波形。请注意，输出 30 与输入在同一周期产生。然而，图 1c 展示了将 op 设为 1 时尝试执行乘法运算的情况。算术逻辑单元的时序行为发生了变化——输入提供后两个周期才产生乘积。此外，如果 op 没有在额外一个周期内置为有效，输出就会出错。问题在于，加法器是 组合逻辑的——它在与输入相同的周期产生输出；而乘法器是 时序逻辑的——它需要多个周期才能产生其

输出。由于乘法器输出比加法器晚产生，因此需要额外一个周期，并且多路复用器需要在后续周期使用 `op` 输入来选择正确的输出。`op` 是必需的。

ALU、加法器和乘法器的接口无法捕获这些细节。规避这一问题的一种方法是使用延迟不敏感接口（例如就绪–有效握手）封装每个模块。但对于细粒度组合而言，这些接口会带来可能难以接受的开销 [37]。本文旨在基于时钟周期定义高效的延迟敏感接口，并通过静态检查排除对这些接口的误用。

2.2 Filament

Filament 是一种硬件描述语言，允许用户直接指定 并检查 其模块的时序行为。每个组件都可以由多个事件参数化，这些事件用于指定其时序行为。我们的 ALU 实现表现得不可预测，因为加法器和乘法器具有不同的时序行为。Filament 允许我们使用事件 显式编码其时序行为，并以此参数化模块：

```
<escape>extern comp Add<T>(@interface[T] go: 1, @[T, T+1] left: 32,
```

这两个组件都使用事件 T 来指定其时序行为。加法器是组合逻辑——它在与输入相同的周期输出结果。这一点通过输入和输出的可用区间来表示：输入在半开区间 $[T, T+1)$内提供，该区间对应组件执行的第一个周期，输出也在同一区间内产生。相比之下，乘法器是时序逻辑——它需要两个周期才能产生输出。这通过声明输出在区间 $[T+2, T+3)$内可用来表示，该区间比输入在区间 $[T, T+1)$内提供晚两个周期。为了表明事件 T 已经发生，使用这些模块的用户必须将接口端口 `go` 设为 1，按照规定的区间提供输入，并在输出可用时读取输出。多路复用器（图中未显示）也是组合逻辑，并且在同一周期接收所有输入。

```
comp ALU<G>(
    new Mult;
    new Mux;
```

与我们的硬件描述语言实现一样，我们的 Filament 算术逻辑单元实现也会显式实例化其所需的所有硬件资源。关键区别在于，Filament 如何通过 调用 来表达对硬件实例的使用。一次调用会使用一组特定事件来安排硬件实例的执行，并提供所有输入。例如，对加法器 `a0` 的调用 `A` 使用事件 G进行调度。通过命名这些使用，Filament 可以检查模块的时序行为。加法器的 `go` 端口无需赋值；编译器会根据调度事件 G自动插入该赋值。调用是一种逻辑构造，Filament 编译时会将其消除（第 5 节）。乘法器和乘-tiplexer 也使用事件 G进行调度。多路复用器不使用实例的输出，而是使用调用上的端口，以反映特定调用所产生的输出。

2.3 检查时序行为

然而，当我们尝试编译此程序时，Filament 给出了以下错误：

```
mux := Mux<G>(op, m0.out, a0.out);
Available for [G+2, G+3) but required during [G, G+1)
```



错误指出，我们使用的多路复用器要求其所有输入都在 $[G, G+1)$ 时可用，而乘法器的输出 `m0.out`则在 $[G+2, G+3)$ 时可用。Filament 要求所有输入至少在与相应参数的要求相同的时长内可用。这正是我们原始硬件描述语言设计（第 2.1 节）中的问题——加法器的输出与乘法器的输出在不同周期可用，从而导致出乎意料的时序行为。Filament 的类型系统会在静态检查时捕获此错误。

解决方案是使用 寄存器 来存储值，并使其在后续周期中可用。寄存器的签名捕获了其时序行为——输出在输入之后一个周期可用[1]：

```
comp Reg<G>(@interface[G] en: 1, @[G, G+1] in: 32) -> (@[G+1, G+2] out: 32)
```

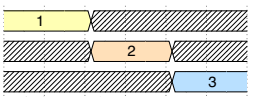
修正后的实现使用两个寄存器，使求和结果能够与乘法器的结果在同一周期内可用。第一个和第二个寄存器的输出分别在$[G+1, G+2)$和 $[G+2, G+3)$时可用。我们将多路复用器的执行安排在 $G+2$ 周期，此时两个输出均已可用。该设计仍然存在问题，因为`op`仅在 $[G, G+1)$时可用，而多路复用器却在 $[G+2, G+3)$时读取它。我们通过使 `op` 信号在 $[G, G+3)$时可用来解决这一问题。由此得到一个正确的算术逻辑单元实现。

然而，ALU 何时准备好接收新的输入并不明确：我们应该等到输出产生后再继续，还是该模块能够并行处理多个输入？

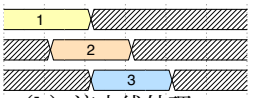
```
comp ALU<G>(@[G, G+3] op: 32, ...) {
 a0 := A<G>(l, r); R0 := new Reg; R1 := new Reg;
 r0 := R0<G>(a0.out); r1 := R1<G+1>(r0.out);
 mux := Mux<G+2>(op, r1.out, m0.out); ... }
```

2.4 流水线化

流水线化 是一种常见的优化，可使硬件并行处理多个输入。顺序模块一次处理一个输入（图 （a）），而流水线化模块可以重叠处理多个输入（图 （b））。



（a）顺序处理



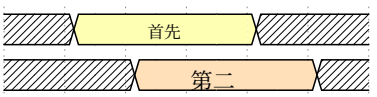
（b）流水线处理

流水线化颇具挑战，因为这需要推理同一物理资源的多个并发执行之间的交互。要正确实现流水线化，必须使用来自正确流水线级的值，并确保不存在结构冒险，即内部组件不会发生冲突使用。

Filament 提供了一个简洁的解决方案：每个事件都有一个关联的 延迟，它规定在接受新的输入之前需要等待多少个周期。`comp ALU<G:1>(...)` 我们可以更新加法器和乘法器的签名以反映这一点。

由于加法器是组合逻辑，因此每个周期都可以接收新输入。不过，乘法器每 3 个周期才能接收新输入。对于用户级组件，Filament 会确保每个事件的延迟 正确无误，也就是说，组件可以正确地进行流水线化。我们将重新设计 ALU，使其实现流水线化并且每个周期都能接收新输入，方法是指定 G 的延迟为 1。由于我们知道设计并未进行流水线化，Filament 将生成错误信息，解释该设计为何无法进行流水线化。

```
comp ALU<G:1>(
```



[1]这是寄存器的简化接口。完整接口请参见第 3.6节。

```
def init(left) -> (acc, q):
  # Initialize the computation
def nxt(a, q, div) -> (an, qn):
  # One step of the computation
def div(l, r):
  (qn, an) = init(l)
  for _ in range(0, 8):
    (qn, an) = nxt(an, qn, r)
  return qn
```

(a) 恢复除法的伪代码。

```
comp Comb<G: 1>(...) -> (
  @[G, G+1] out: 8) {
 i := new Init<G>(left);
 n0 := new Nxt<G>(i.A, i.Q, r);
 ...
 n7 := new Nxt<G>(n6.A, n6.Q, r);
 out = n7.Q;
```

(b) 完全组合逻辑除法器。

```
comp Pipe<G: 1>(...) -> (@[G+7, G+8] q: 8) {
 i := new Init<G>(left); // Instantiate and invoke
 n0 := new Nxt<G>(i.A, i.Q, r);
 ra0 := new Reg<G>(n0.A);
 rq0 := new Reg<G>(n0.Q);
 n2 := new Nxt<G+1>(ra0.out, div, rq0.out);
 ...
 out = n7.Q;
```

(c) 流水线化除法器。在连续的周期中调度的实例。

```
comp Iter<G: 8>(...) -> (@[G+7, G+8] q: 8) {
  I := new Init<G>(left);
  N := new Nxt; RA := new Reg; RQ := new Reg;
  n0 := N<G>(i.A, i.Q, r);
  ra0 := RA<G>(n0.A); rq0 := RQ<G>(n0.Q);
  n1 := N<G+1>(ra0.out, rq0.out);
  ra1 := RA<G+1>(n1.A); rq1 := RQ<G+1>(n1.Q); ...
  out = n7.Q;
```

(d) 迭代式除法器。组件在多个周期内重复使用。

图 2. 实现... 8 位恢复除法的不同实现，展示了面积-吞吐量权衡。Filament 的 t ype 系统确保 确保每种实现都经过正确的流水线化处理，且不会引入资源复用冲突。

我们的第一个问题是，签名要求输入信号 `op` 连续三个周期均可用，而流水线却可能每个周期都触发。波形图展示了这一问题——第一次迭代的 `op` 输入将与第二次迭代的输入重叠。然而，`op` 是电路中的一个物理端口，一次只能保存一个值；这是硬件设计的一项基本物理约束。Filament 要求，事件的延迟至少应与使用该事件的任意可用时间间隔的长度相同；因此，我们必须将 `op` 的可用时间间隔缩短为 1 个周期。我们选择 $[G+2, G+3)$，因为多路复用器在此区间内使用 `op`。

```
comp Mult<T: 3>(
```

接下来，Filament 提示，虽然我们的算术逻辑单元流水线每个周期都可以接收新的输入，但乘法器 `M` 每 3 个周期才能接收新的输入。这是我们所用乘法器电路的根本限制；要解决这一问题，`m0` 器。Filament 还发现了一个由组合引发的

```
Event may retrigger every cycle
Cannot safely pipeline
```

流水线中使用的子组件必须至少能够以与流水线本身相同的频率处理输入。修正这一点后，就能得到一个正确且完全流水线化的算术逻辑单元。Filament 的一个关键目标是确保改变组件的流水线化行为不会引入额外错误——流水线化算术逻辑单元与顺序算术逻辑单元一样，只会在信号在语义上有效时使用它们。

```
comp ALU<G: 1>(@interface[G] en: 1, @[G+2, G+3] op: 1, ...) {
  A := new Add; Mx := new Mux; R0 := new Reg; R1 := new Reg; FM := new FastMult; // delay = 1
  a0 := A<G>(l, r); r0 := R0<G>(a0.out); r1 := R1<G+1>(r0.out); m0 := FM<G>(l, r);
  mux := Mux<G>(op, r1.out, m0.out); o = mux.out; }
```

2.5 使用 Filament 权衡面积与吞吐量

虽然流水线化提高了组件的吞吐量，但也会增加资源使用量。对于浮点乘法器这类大型电路，通常值得在多个

时钟周期内复用同一电路。不过，电路复用会影响流水线化行为：组件启动新迭代的能力取决于子组件的共享方式。Filament 的类型系统会跟踪资源复用，并确保类型良好的组件不会因复用组件而产生结构冒险。

为了展示 Filament 如何支持对面积-吞吐量权衡进行安全探索，我们使用恢复除法算法实现了三种不同版本的除法器（图2a）。组合逻辑组件 `Init` 和 `Nxt` 分别计算商（`.Q`）和累加器值（`.A`）。对于一个8位值，我们必须应用 `Nxt` 8 次。

组合逻辑除法器。 图2b 实现了一个组合逻辑除法器，在提供输入的同一周期内计算输出。所有 `Nxt` 实例都使用事件进行调度 G，这意味着它们将在同一周期内执行。虽然该设计的延迟为1，但它的效率相当低，因为它在同一时钟周期内调度了大量复杂逻辑，并迫使设计以较低频率运行。不过，组合逻辑设计是验证算法正确性的良好起点。

流水线化除法器。 为了让设计以更高频率运行，我们可以通过调度每个 `Nxt` 实例在连续的周期中执行来实现流水线化。为了正确传递值，我们实例化寄存器，以保存每个 `Nxt` 组件的商和累加器值。图2c 展示了该实现：模块的延迟仍为1，因此它可以每个周期处理一个新值，但与组合逻辑实现不同，其延迟现在为8 个周期。流水线化还缩短了较长的组合逻辑路径，使设计能够以更高频率运行。

迭代除法器。 组合逻辑和流水线化输入都可以每个周期处理一个新输入，但需要大量硬件，因为它们分别实例化了 8 个 `Nxt` 组件，而流水线化版本还需要 16 个寄存器。我们可以通过实现一种迭代式设计，改为使用相同的 `Nxt` 组件和寄存器。

我们从组合逻辑设计开始，并将所有调用改为使用同一个实例N。Filament 告诉我们，这个设计存在错误。我们正试图将两个不同的输入发送到 `Nxt` 在同一周期内的实例。但 `Nxt` 是一个物理电路，每个周期只能处理一个输入。因此，我们必须将该实例的使用安排在不同的周期，并添加寄存器来保存值，这与流水线式实现类似。

```
comp Nxt<T:1>(...)
Delay requires uses to be 1 cycle apart
s0 := N<G>(i.A, div, i.Q); First use
s1 := N<G>(s0.AN, div, s0.QN); Second use
```

进行这些更改后，Filament 报告了新的错误信息。由于我们在 8 个周期内共享实例 `Nxt`，除法器无法每个周期都开始处理新的输入。同样，这是因为 `Nxt` 是一个物理电路，每个周期只能处理一个输入。要解决这个问题，我们可以将延迟改为 8 个周期，这样就能向 Filament 保证该实例每 8 个周期才运行一次，从而得到最终设计。（图2d）。这确保使用实例 `N` 的所有迭代都在提供新输入之前完成。Filament 以隐式方式向我们展示，重复使用该实例是一种权衡：虽然我们使用的资源更少，但吞吐量也降低了，因为与流水线式实现每个周期都能处理一个新输入相比，我们的迭代式实现每 8 个周期才能处理一个新输入。

```
comp Iter<G:1>(...)
Event may trigger every cycle
causing shared uses to conflict
s0 := N<G>(i.A, div, i.Q);
      First use
s7 := N<G+7>(s6.AN, div, s6.QN);
      Last use
```

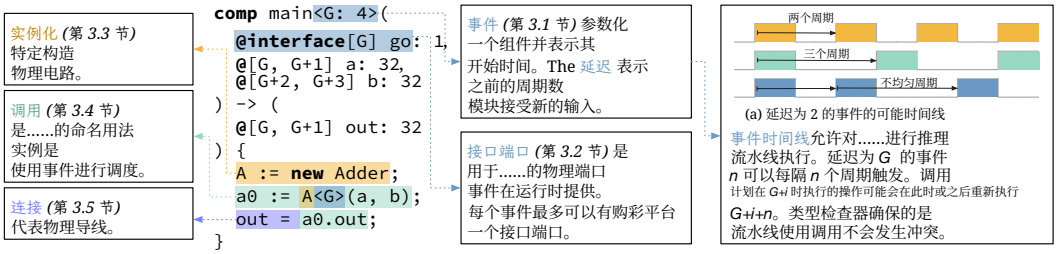


图 3. Filament 语言概览。程序由一系列 组件 定义组成，分别对应各个模块。组件的签名使用 事件 进行参数化。组件的主体由三类语句组成：实例化、连接和调用。

2.6 总结

Filament 是一种用于安全设计和组合静态流水线的硬件描述语言。具体而言，Filament 程序可以指定并检查硬件模块的时序属性，并确保：

(1) 仅当端口和导线上的值在语义上有效时，才会读取这些值。(2) 硬件实例不会以冲突的方式使用。

这些属性确保生成的流水线既安全（即不存在资源冲突），又高效（即它们可以按照接口所规定的方式重叠执行计算，且没有任何额外开销）。Filament 的实用性也扩展到了在该语言之外定义的组件。通过为外部模块指定类型签名，用户可以安全地组合模块。第 3 节概述 Filament 中的构造，第 4 节解释 Filament 的类型系统如何检查流水线安全性，第 5节展示如何将 Filament 的高级构造编译为高效的硬件。

3 THEFILAMENT 语言

图 3 概述了 Filament 语言。Filament 的抽象层次与 结构化 硬件描述语言相当，在其中计算必须显式映射到硬件上。Filament 仅有四种构造：组件、实例化、连接和调用。前三种在传统硬件描述语言中都有直接对应，而调用则是一种新颖的构造。

3.1 事件与时间线

事件是 Filament 中时间的核心抽象。设计不再使用时钟信号，而是使用事件来调度计算。Filament 编译器会生成高效的流水线化有限状态机，以具体实现事件（第 5.2 节）。

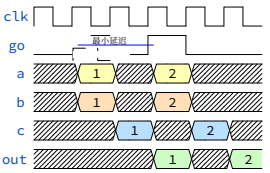
定义事件。 定义事件只有两种方式：(1) 组件签名绑定事件变量，例如 G；(2) 用户可以写入事件表达式，例如 $G+n$，其中 n 是一个常量。事件与时钟有直接关系：如果 G 发生在时钟周期 i，那么 $G+n$ 就发生在时钟周期 $i+n$。[2] 这种与时钟的关系至关重要，因为它使 Filament 能够表示基于时钟的硬件描述语言（HDL）中定义的组件的时序属性。不允许将事件变量相加（G_0+G_1），因为事件对应特定的时钟周期，将它们相加没有意义。

事件的时间线解释。 为了捕获流水线执行中潜在的资源冲突，Filament 将事件解释为一组可能的时间线。事件的时间线 G 带有延迟 n 是由以下内容组成的任意无限序列：每个脉冲的时长为 1 个周期，且相邻脉冲之间至少间隔 n 个周期。图 2a 展示了一个延迟为 2 的事件的一组有效时间线。通过赋予事件

[2]所有事件变量都在同一时钟域中运行，但未来可以取消这一限制。

```
comp AddMult<G: 2>(
  @interface[G] go: 1,
  @[G, G+1] a: 32, @[G, G+1] b: 32, @[G+1, G+2] c: 32
) -> (@[G+2, G+3] out: 32) { commands }
```

(a) Filament 中组件的签名



(b) 显示流水线使用的波形

图 4. 签名和波形图。该组件允许流水线执行，或在两个周期后重用，从而实现重叠执行。阴影区域表示未知值。

时间线解释，Filament 便可以推理 重复执行，并考虑流水线化执行可能如何相互影响。通过对这些属性进行推理，我们可以为硬件的流水线执行定义并强制实施安全属性。此外，时间线解释与硬件之间存在直接关系：事件的延迟表示用户在提供新的一组输入之前必须等待多少个周期。硬件设计师通常将这称为流水线的启动间隔（第 4.3 节）。

3.2 组件

Filament 程序以组件为单位组织，通过一组命令描述其签名中的时序行为和电路。图4 展示了 Filament 中一个组件的签名（图4a），以及一个波形图，显示两组输入并行处理的情况（图4b）。该组件通过事件进行参数化 G，延迟为2，这意味着在前一次使用后的两个周期即可开始流水线使用。

接口端口。 硬件组件通常具有 控制端口，用于指示数据端口 上的值何时有效，以及何时应执行计算。控制端口上的值始终被视为在语义上有效，而数据端口上的值仅在相应的控制端口为高电平时才有效。Filament 通过将控制端口定义为 接口端口来区分控制端口。接口端口是与特定事件相关联的 1 位端口。当接口端口设置为 1 时，它会向组件发出相应事件已发生的信号。例如，在 `go` 上设置为 1，对于一个 `AddMult` 实例（图 4a），会使模块开始处理输入。所有使用某个事件的端口的可用区间，均以相应事件的接口端口设置为 1 时刻为基准。如果某个事件没有接口端口，则模块可以假定该事件每隔 n 个周期触发一次，其中 n 是该事件的延迟。

可用区间。 组件的输入端口和输出端口通过组件绑定的事件来描述其可用性。对于 `AddMult` （图 4a），所有端口都使用事件G。可用区间是左闭右开的：例如，输入端口 `a` 在 $[G, G+1)$ 期间可用，这意味着组件被调用后的第一个周期内它可用。在组件主体内部，输入的可用时间间隔代表一项保证，而输出的可用性则要求主体满足一项要求。使用组件时，情况正好相反：输入具有必须由用户满足的要求，而输出则具有保证。

3.3 实例

硬件设计中的所有计算都必须显式映射到物理电路上。Filament 的`new` 关键字允许实例化子组件。以下程序实例化了两个名为 `A0` 和 `A1` 的`Add` 组件实例，可独立使用。这些实例化

不会为 Add的事件 T提供绑定；调用负责提供这些绑定并调度实例的执行。

```
comp Add<T: 1>(@[T, T+1] left: 32, @[T, T+1] right) -> (@[T, T+1] o: 32);
comp AddTwo<G: 1>(...) { A0 := new Add; A1 := new Add; ... }
```

3.4 调用

硬件设计中的资源复用采用时分复用的方式，即安排同一资源的不同用途在不同时间执行。具体做法是使用一个寄存器构建有限状态机（FSM），并利用寄存器的输出选择要使用的输入。示例程序计算 $(l \times r)^2$ 使用单个乘法器，利用有限状态机 `F` 在第一个周期将输入 l和 r转发到乘法器，并在第二个周期转发乘法器的输出。然而，对 `M.left` 的赋值错误地在第一个周期转发了来自 `M.out` 的值。有限状态机控制逻辑中的错误不会导致任何明显的问题；但这个错误会导致数据在无声无息中损坏，并传播到系统的其他部分。

```
F := new Reg; // FSM
F.in = F.out == 0 ? 1 : 0;
M := new Mult; A := new Add;
M.right = F.out == 0 ? r : M.out
M.left = F.out == 1 ? l : M.out
```

与之相反，在 Filament 中，每次使用实例都必须明确命名，并通过调用进行调度。乘法器 `M` 的第一次调用使用事件 G进行调度，使用输入 l和r，并命名为 `m0`。第二次调用在 $G+1$处晚一个周期进行调度，随后可以使用 `m0.out` 引用第一次执行的输出，并将其作为输入传递给乘法器。由于第二次调用晚一个周期进行调度，输入端口的要求也有所不同：输入必须在区间 $[G+1, G+2)$ 内可用

```
comp Square<T:1>(
```

而不是 $[G, G+1)$在第一次调用中使用的值。这使 Filament 能够检查 `m0.out` 在用作 `m1` 的输入时在语义上是否有效，并确保乘法器的两次使用安排在不同时间执行，从而让编译器生成正确的有限状态机来调度实例复用。每次调用只为数据端口提供输入，而省略接口端口的输入。在编译过程中，Filament 的编译器会自动推断输入端口的赋值，并生成高效的流水线有限状态机来调度调用（第 5节）。

3.5 连接

Filament 程序允许连接端口，并要求源端在语义上至少与目标端一样长。连接在物理上由持续激活的导线实现，用于连接电路中的两个端口。

```
comp Add<G:1>(@[G, G+3] source: 32) -> (@[G, G+1] dest: 32) { dest = source; }
```

3.6 与外部组件的接口

Filament 的 `extern` 关键字允许用户通过指定不含主体的类型签名，为黑盒模块提供类型安全包装器。Filament 的标准库使用 `extern` 组件定义，其中提供了乘法器和寄存器等组件的签名。

虚事件。虚事件使 Filament 能够对加法器等组件的行为进行建模；这类组件持续处于激活状态，且不需要显式的使能信号。在以下签名中，事件 G 是一个虚事件，因为该签名中没有与之对应的接口端口。第 5.4 节将介绍用户级组件如何使用虚事件。

```
extern comp Add<G: 1>(@[G, G+1] l: 32, @[G, G+1] r: 32) -> (@[G, G+1] o: 32))
```

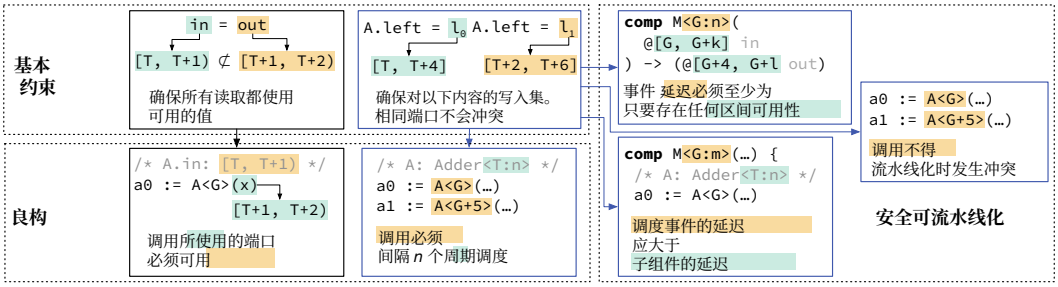


图 5. Filament 类型系统概览。硬件设计的基本约束蕴含其他约束。良构性确保组件的一次执行 是正确的。安全流水线确保组件的流水线化执行 是正确的。

排序约束。 为了充分表达外部组件的能力，Filament 允许定义事件之间的排序约束。例如，如果输入持续提供多个周期，组合逻辑组件可以在多个周期内提供有效输出。因此，组合加法器更精确的接口如下：

```
comp Add<G: L-G, L: 1>(@[G, L] l: 32, @[G, L] r: 32) -> (@[G, L] o: 32) where L > G
```

事件 G 和 L 标记输入和输出可用区间的起点和终点。为了确保区间 $[G, L]$ 是良构的，签名要求 $L > G$。组件保证只要输入持续提供，就会持续提供输出。

参数化延迟。加法器的新签名还指定了 $L - G$ 周期的参数化延迟，以表明加法器在处理一组输入时可能无法重复使用。为了生成静态

```
A := new Add;
```

流水线具有与输入无关的时序行为，因此 Filament 要求所有此类表达式都计算为常量值。与示例一样， Add 的调用必须提供某种形式的绑定： $G = T + i$ 和 $L = T + k$，使得 $k > i$，从而确保相应调用的延迟是编译时常量 $k - i$，并且满足顺序约束 $L > G$。

Filament 中寄存器的签名允许它们根据需要持续提供输出，这与加法器类似。但是，由于寄存器是状态元件，它只需要一个周期的输入。此外，延迟信号表明，寄存器可以在输出可用的最后一个周期内接受新的写入。

```
comp Register<G: L-(G+1), L: 1>(
  @interface[G] go: 1, @[G, G+1] in: 32) -> (@[G+1, L] out: 32) where L > G+1;
```

4 类型系统

Filament 的类型系统强制执行硬件设计的两项基本限制：

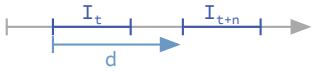
(1) 所有读取操作仅使用在语义上有效的值。端口或导线上始终会有一个值。Filament 的可用区间标记了这些值在语义上有效的时间。(2) 写入操作不会发生冲突。这是以下属性的推论：资源的使用不得发生冲突，因为资源的使用通过写入来表示。

Filament 通过两个检查阶段确保这些属性：良构性检查确保组件的单次执行正确，而安全流水线则确保组件的流水线化执行正确。

4.1 延迟良构性

事件的延迟涵盖了使用该事件进行调度的流水线各部分之间所有可能发生的冲突。Filament 要求事件的延迟至少与所有提及该事件的区间一样长，以确保实例复用不会导致其输入端口和输出端口之间发生冲突。

证明很直接：对于时刻为 t 和$t+n$ 的两次调用，满足 $n \geq d$，其中 d 是延迟。设 I_t 和 I_{t+n} 分别为输入 i的可用区间。由于我们知道区间的开始



时间至少相隔 d 个周期（$I_{t+n} - I_t \geq d$），且区间长度受限于 d（$|I_t| \leq d$），因此可以得出它们不会重叠。

4.2 良构性

有效读取. 为确保这一性质，Filament 必须确保只有在端口值在语义上有效时才读取这些值。信号用于两个地方：

(1) 连接 (第 3.5节）会将一个值从一个端口传递到另一个端口。Filament 确保输出端口的可用时长至少满足输入端口的要求。(2) 调用 (第 3.4节）使用一组事件来安排组件实例的使用。检查调用是否有效归结为两个步骤：将实例签名中的事件变量绑定到调用的事件上，即可计算出该实例输入端口的要求。接下来，每个参数本质上都表示实例输入与该参数之间的连接，并依据连接的判定标准进行检查。

无冲突。 如果某次调用使用事件d 安排一个具有延迟 G的实例，则在$[G, G+d)$期间不得重用该实例。这既确保输入端口与输出端口之间没有冲突（第 4.1节），也确保各子组件之间不存在冲突。后一项性质成立，是因为安全的流水线化约束确保有效延迟能够正确涵盖子组件之间所有可能的冲突（第 4.4节）。在例

```
comp Mult<T:3>(...);
comp main<G:10>() {
  M := new Mult;
  // busy b/w [G, G+3]
  a0 := M<G>(a, b);
  // busy b/w [G+1, G+4]
  a1 := M<G+1>(a0.out, b);
```

ample 程序中， `M` 的两次调用发生重叠，导致 Filament 拒绝该程序 .

4.3 启动间隔

流水线化是一项重要的优化，因为它使模块能够并行处理多个输入。例如，一个延迟为三个周期、但启动间隔为一个周期的乘法器，计算输出需要三个周期，但每个周期都能接收新的输入。在 Filament 中，事件的延迟对应于启动间隔。硬件设计人员通常讨论组件的启动间隔，而 Filament 将这一概念推广到允许组件具有多个事件。在这种情况下，每个事件都指定内部流水线某一部分的启动间隔。Filament 确保模块的延迟描述了一个有效的启动间隔，其定义如下：

定义 *4.1*（启动间隔）。令 $P(t)$ 表示时间 t处流水线 P 的执行。 $P(t_0) \perp P(t_1)$ 表示在 t_0 和 t_1 时刻对 P 进行的流水线执行不存在资源冲突。当且仅当满足以下条件时， I 是流水线 P 的有效启动间隔：

$$\forall n \geq 0 \; P(t) \perp P(t + I + n)$$

此定义要求流水线能够在启动间隔之后的任意 时间接受新的输入。可能还存在小于启动间隔的其他延迟，使流水线能够在再次变为无效之前的短暂时间窗口内接受新的输入。

这对应于启动间隔 I 的以下定义：

$$\forall k \neq 0\ P(t) \perp P(t + k \times I)$$

Filament 使用第一个定义，因为延迟也用于检查组件的良构性约束。如果我们使用第二个定义，良构性约束就会要求：如果某个实例被安排在时间 t执行，那么它只能在其他时间$k * t$ 再次执行；我们认为这样会降低组合性。无论如何，这并非根本限制，因为这两种定义都可以编码并强制执行。

4.4 安全流水线化

良构性确保模块的一次执行正确，即所有读取都使用有效值且不存在冲突；而安全流水线化必须确保组件的流水线化执行不会产生任何额外冲突。检查流水线化执行是否存在冲突，与检查同一实例的调用是否冲突非常相似。这是因为流水线执行本质上完全相同——实例在经过一段时间后被重新使用。Filament 必须证明：对于使用事件 G安排的一次调用，在 $G + d$ 之后的任意时间安排的另一次调用（其中 d 是延迟）都不会与第一次调用冲突。以下检查足以证明这一点。

触发子组件。 Filament 要求：当使用事件调用子组件时，该事件的延迟必须至少与子组件事件的延迟相同。

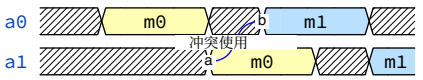
```
comp Mult<G:3>(@interface[G] go: 1, ...)
comp main<T:1>(@interface[T] go: 1, ...) { M := new Mult; m0 := M<T+2>(...) }
```

事件 $T + 2$ 用于安排对实例 M的调用，该实例的延迟为 3。然而，$T + 2$的延迟为 1，与 T相同。这会造成问题，因为 `main` 可能每个周期都触发一次，而`M` 只能每 3 个周期支持一次计算。因此，Filament 会拒绝此程序。

重用实例。先前的检查已确保：(1) 共享调用在一次流水线执行期间不会发生冲突；(2) 调用的流水线执行不会与自身发生冲突。不过，我们还需要确保共享实例的流水线化调用之间不会发生冲突。

```
comp Mult<G: 3>(...)
comp main<T: 3>(...) {
  M := new Mult;
  m0 := M<T+2>(...);
  m1 := M<T+10>(...);
```

示例程序能够通过之前的所有检查，但实际上有误：在时间 T 和T + 10 执行流水线，将导致时间 T 执行中的 `m1`



与 `m0` 发生冲突，冲突源于时间 $T+10$的执行。由于 Filament 对启动间隔的定义允许在未来的任意时间重新执行，因此我们必须要求共享实例的所有调用都在流水线执行开始之前完成。以下条件足以确保这一点：延迟必须大于共享实例最早一次调用的开始时间与最后一次调用的结束时间之间的周期数。

动态复用。 由于 Filament 组件可以由多个事件进行参数化，因此可以使用两个不同的事件调用一个实例。在示例程序中，类型检查器必须证明区间 $[G, G+3)$ 和 $[L, L+3)$ do

```
comp Dyn<G: ??, L: ??>(..) {
```

不重叠以确保无冲突。约束 $L \geq G + 3$ 足以证明这一点。然而，无法对该模块进行静态流水线处理： G 的延迟是 动态的，它取决于具体在哪个周期提供 L ，而这无法预先得知。不存在能够表示这两个事件延迟的编译时常量值。这是因为延迟描述的是时间线

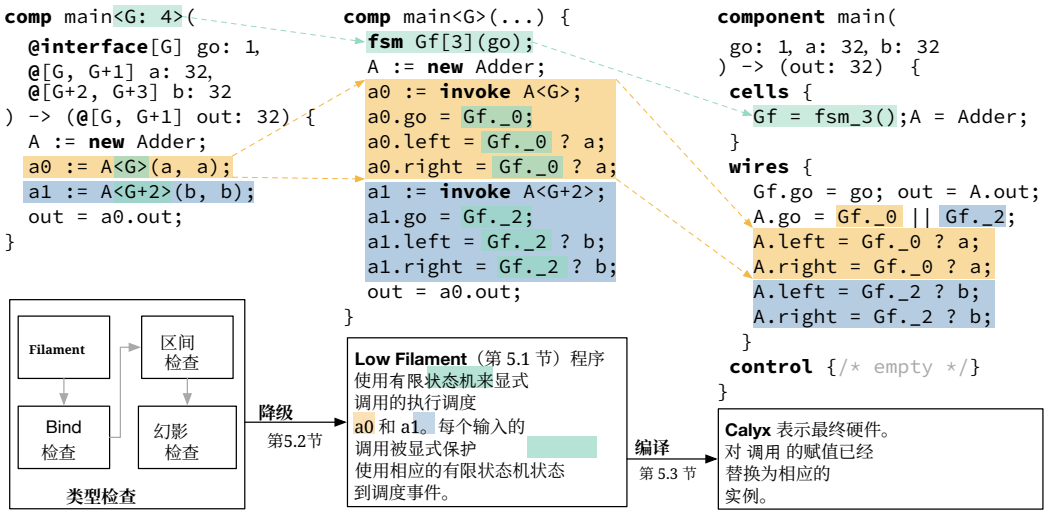


图 6. 编译流程。Filament 程序经过类型检查（第 4 节），并降级为 *Low Filament*（第 5.1 节）程序。降级（第 5.2 节）会实例化显式有限状态机以调度调用。最后，Low Filament 程序被编译为 Calyx [40]，后者会优化设计并生成硬件电路。

对于单个事件，而动态模块需要关联多个事件。Filament 的解决方案是禁止用户级组件中的事件之间存在排序约束，这也就排除了示例程序。外部组件（第 3.6节）仍可使用排序约束，但此类约束只能通过定义在 $G+n$ 事件上的自然顺序来满足。这意味着在类型正确的程序中：

（1）所有延迟均可计算为编译时常量。（2）共享组件的所有调用都使用同一事件。

这些约束使编译器能够根据类型正确的程序生成高效的静态定时流水线。将安全的动态流水线扩展到 Filament 中，是未来的一个研究方向。

5 编译

图 6 展示了编译流程的概览。Filament 编译流水线的主要目标，是将调用的抽象调度转换为明确的流水线化控制逻辑。编译器首先将程序降级为 *LowFilament*，它是 Filament 语言的一种无类型扩展，明确使用流水线有限状态机（FSM）来协调模块的执行。接下来，编译器将程序转换为 Calyx 中间语言 [40]，并在其中执行通用优化和生成电路。

5.1 Low Filament

Low Filament 是 Filament 的无类型版本，它引入了新的构造，以明确表示模块的流水线执行。

显式调用。 Low Filament 要求显式分配与调用对应的所有端口。这包括接口端口，而这些端口在高级 Filament 中由系统隐式管理。

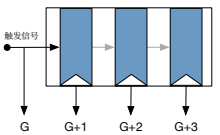
守卫赋值。 Filament 使用 守卫赋值 来表达信号的多路复用，其与 Calyx 中的守卫赋值直接对应 [40]。该赋值仅在守卫处于激活状态时，才会将 `out` 的值传递出去。其他

```
in = g1 ? out;
in = g2 ? out;
```

否则，转发到 `in` 的值未定义。Calyx 的良构性条件要求，对于任何给定的源端口，同一时间只能有一个保护条件处于激活状态。

有限状态机。Low Filament 还提供了 `fsm` 构造，用于显式实例化流水线有限状态机。这定义了具有 `F` 状态的有限状态机，并带有一个触发其执行的输入端口 `trigger`。这会生成一个大小为 n 的移位寄存器，带有端口： `F._0`、…、 `F._{n-1}`。如果在事件 G将触发信号设置为 1，端口 `F._i` 将在事件 $G+i$变为活动状态。

```
fsm F[n](trigger);
```



5.2 生成显式调度

从 Filament 到 Low Filament 的编译确保所有高级调用都已编译为显式调用。图 6 展示了使用加法器（`A`）进行两次调用（`a0` 和 `a1`）的程序的编译过程。

有限状态机生成. 编译器会为参数化模块的每个事件实例化一个有限状态机。示例程序使用 event G 来调度调用。编译器会遍历程序中的所有表达式 $G+i$，以计算流水线化有限状态机的级数。虽然原始程序没有明确提及事件 $G+3$，但它由在区间 $[G+2, G+3]$内处于活动状态的输出端口 `a1.out`隐含表示。编译器实例化了有限状态机 `Gf`，其中包含由 `go` 信号触发的 3 个状态。请注意，有限状态机的延迟不会 影响有限状态机的生成。

触发接口端口。 随后，编译器通过为加法器的接口端口 `go`生成显式赋值，来降低这些调用。第一个调用安排在 G，使用端口 `Gf._0`触发该调用；第二个调用安排在 $G+2$，使用端口 `Gf._2`。

守卫合成。 为了确保来自两个调用的赋值不会在数据端口`left` 和 `right` 上发生冲突，编译器会为这些赋值合成守卫。如果某个调用的输入端口需要在区间 $[G+s, G+e]$内接收输入，编译器就会生成守卫 `Gf._s || ... || Gf._e`。由于程序经过良好类型检查，因此可以保证各个调用的守卫表达式互不冲突（第 4 节）。

5.3 降级到 Calyx

Low Filament 的设计有意贴近 Calyx，因此编译过程很直接。对于每种有限状态机大小 n，我们都会生成一个 Calyx 组件，并将其实例化以对应 Filament 组件。有限状态机只是由寄存器依次连接而成的序列。由于 Low Filament 中所有端口的赋值都是显式的，我们只需将调用替换为相应的实例名称，即可完成编译。在示例程序中，对 `a0.left`、 `a1.left` 的赋值都会编译为对 `A.left` 的赋值。由于 Filament 保证生成的守卫彼此不相交，因此我们可以确信 Calyx 会生成正确的有限状态机。

5.4 优化连续流水线

连续流水线不会使用信号来指示其输入何时有效，而是持续处理输入。我们可以在 Filament 中使用虚事件（第 3.6 节）来表达此类流水线。虚事件没有对应的接口端口，因此不能用于触发调用。Filament 通过虚事件检查分析确保虚事件得到正确使用，该分析会确保：

$x \in vars \quad t \in events \quad p, q \in ports$

$M ::=$

<escape>def $C\langle t{:}\ n\rangle(p_1{:}\pi_1, \ldots, p_j{:}\pi_j)\{c\}$</escape>

$c ::= c_1 \bullet c_2 \mid p_d = p_s \mid x := \text{new}\ C$

<escape>| x</escape>: <escape>=</escape>调用 <escape>x</escape><escape>T</escape><escape>$p_s, \ldots, p_j$</escape>

$T ::= t \mid T + n \quad \pi ::= [T_1, T_2]$

$\tau ::= \forall\langle t{:}\ n\rangle(p_1{:}\ \pi_1, \ldots, p_j{:}\pi_j)$

(a) 抽象语法

$[\![c]\!]{:}\ \mathrm{L} \to \mathrm{L} \qquad \mathrm{L} : \mathrm{T} \to \mathrm{R} \times \mathrm{W}$

$[\![p_d = p_s]\!]\ (L) = \text{map}(\lambda(R, W).\ \text{如果}\ p_s \in W$
然后 $(R\{p_s/p_d\}W)$ else $(R, W))\ L)$

$[\![c_1 \bullet c_2]\!]\ (L) = [\![c_1]\!]\ (L) \cup [\![c_2]\!]\ (L)$

(b) 日志转换器语义

$$\frac{\Delta, \Lambda_1, \Gamma \vdash c_1 \dashv \Lambda_1', \Gamma_1 \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 \dashv \Lambda_2', \Gamma_2}{\Delta, \Lambda_1 * \Lambda_2, \Gamma \vdash c_1 \bullet c_2 \dashv \Lambda_1' * \Lambda_2', \Gamma_1 \cup \Gamma_2}$$

(c) 组合判断

图 7. Filament 的形式语义，其中命令定义为 日志转换器。类型判断会跟踪实例的活动时间线，并确保它们以互不相交的方式使用。

定义 *5.1*（幻影检查）。虚事件 G 使用正确的条件是：

(1) 它不会用于共享任何实例。(2) 它仅用于调用使用虚事件的子组件。

首先，不允许共享资源，因为任何共享实例的流水线都必须使用某个信号触发内部有限状态机，并跟踪当前激活的是该实例的哪次使用。其次，虚事件仅在类型层面可用，且无法具体化，因为它没有接口端口。因此，只有使用虚事件的组件才能通过虚事件调用。

Filament 定义了两种状态原语：寄存器 和延迟 组件。

```
comp Register<G: L-(G+1), L: 1>(
  @interface[G] en: 1, @[G, G+1] in: 32
) -> (@[G+1, L] out: 32) where L > G+1;
```

```
comp Delay<G: 1>(
  @[G, G+1] in: 32
) -> (@[G+1, G+2] out: 32);
```

从类型签名可以看出，两者的区别在于：寄存器可以在任意时长内保持一个值，而延迟只能将一个值保持一个周期。 `Delay`组件每个周期都会接收输入，因此可以提供一个周期的输出。相比之下，寄存器可以使用 `en` 信号，在任意时长内保持一个值。

编译。 编译器不会实例化有限状态机，也不会为由虚事件触发的调用合成保护条件。由于幻影检查可确保所有子组件本身都没有接口端口，编译器无需为它们生成赋值。
Filament 为连续流水线生成的代码与专家编写的代码一致。

6 形式化

图 7a 展示了 Filament 的简化语法：所有组件都可以使用且仅使用一个约束进行参数化，并且不能指定事件之间的任何排序约束。由于 Filament 不允许用户级组件中的事件进行任何形式的交互，因此多事件用户级组件在表达能力上并无本质提升。多事件外部组件的表达能力更强，但我们的形式体系不支持这类组件。Filament 程序（$\mathcal{P}$）由一系列组件构成，这些组件以命令的形式定义签名和主体，包括组合、连接、实例化和调用。

6.1 语义

图 7b 展示了 Filament 的语义，它由针对日志（$\mathcal{L}$）定义的函数构成。日志将事件（$\mathcal{T}$）映射到一个端口集合（从中读取，记为 $\mathcal{R}$），以及一个端口多重集（向其写入，记为 $\mathcal{W}$）。直观而言，日志记录了组件执行的每个周期中发生的所有读取和写入。我们跟踪写入的多重集以捕获冲突：如果同一周期内对同一端口发生多次写入，程序就会出现资源冲突。

具体日志由组件定义的语义生成，而命令只会对其进行变换。例如，端口连接会将源端口的值转发 p_s 到目标端口 p_d。我们通过在读集合 $\mathcal{R}$ 中所有满足以下条件的出现处进行替换来对此建模：当 p_s 在写集合$\mathcal{W}$ 中有定义，并将替换映射到日志中所有已定义的事件。组合体现了硬件的并行特性：它只需将两个日志合并。组合可能导致写入冲突。程序的语义是使用空日志执行指定的主组件所生成的日志。[3] 我们使用这一语义，将类型系统的良构性（第4.2节）和安全流水线（第4.4节）约束形式化。

定义 *6.1*（良构性）。当且仅当组件 M 的日志是良构的，该组件才是良构的。当且仅当对于每个事件而言，该日志 L 是良构的：

- 不存在冲突写入： $W_s = W$其中W_s是去重后的写入集。
- 读取是写入集的子集： $R \subseteq W_s$

定义 *6.2*（安全流水线）。如果组件 M具有延迟为 d 的事件 T，且 $[\![M]\!]_G$ 表示其日志，其中 T 被事件 G 替换，那么，当且仅当所有日志L_n 都格式正确时， M才是安全流水线化的：

$L_n = \forall n \geq d \ [\![M]\!]_T \cup [\![M]\!]_{T+n}$

6.2 类型系统

Filament 实现了一种受分离逻辑启发的类型系统 [46] ，以强制满足良构性和安全流水线化约束。我们的介绍重点关注一种特定的类型判断，它确保实例不会被冲突地使用。附录 A.3 提供了完整的类型系统。从整体上看，我们针对组合的类型判断（图 7c）与并发分离逻辑中使用的并行组合规则相呼应 [9]——两个命令分别在两个互不相交的资源上下文中进行检查。我们的关键见解是将分离拆分的定义改编到实例时间线上，并确保实例的重用不会产生冲突。类型判断的形式如下： $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$。 Γ 是标准类型环境， Δ 跟踪每个事件的延迟，而 Λ 是资源上下文。

资源上下文与分离拆分。 Λ 是资源上下文，以区间（π）的形式跟踪每个实例和端口的可用性。实例化后，每个实例在区间 $[0, \infty)$ 内可用。调用规则（未显示）会检查：对于具有延迟 d 的实例事件，该实例在区间 $[G, G+d)$ 内可用，其中 G 是调度事件。组合规则（图 7c）会在检查两个命令之前拆分资源上下文：

$$\Lambda = \Lambda_1 * \Lambda_2 \text{ iff } \forall (x:\pi) \in \Lambda \Rightarrow \exists \pi_1, \pi_2 . (x:\pi_1) \in \Lambda_1 \wedge (x:\pi_2) \in \Lambda_2 \wedge \pi_1 \cap \pi_2 = \emptyset \wedge \pi_1 \cup \pi_2 = \pi$$

<escape>有效的拆分是指：对于每个实例，拆分后得到的上下文具有互不相交的区间，且这些区间的并集等于原始区间。通过采用这一拆分定义，Filament 确保调用以无冲突的方式重用实例。我们随附的技术报告介绍了用于编码约束以确保良构性和安全流水线的其余类型判断，并证明了以下类型可靠性定理（附录 A.4）：</escape>

Theorem 6.3. 若 $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$ 则 $[\![c]\!]$ 符合良构要求（定义 *6.1*）。

[3]完整的语义详见附录 A。

7 评估

我们评估了 Filament 高效表达多种加速器设计的能力，以及表达由最先进的加速器生成器生成的接口的能力。我们的评估回答以下问题：

(1) Filament 能否表达由先进加速器生成器生成的接口，并与现有工具集成？(2) Filament 能否用于生成高效的加速器？

实现。 Filament 编译器采用基于编译阶段的编译器实现，使用 5426 行 Rust 代码、341 行 Verilog 代码实现标准库原语，并使用最新版本的 Calyx 编译器 [40] 生成 Verilog。所有基准测试均可在一秒内完成编译。

7.1 表达能力评估

为了展示 Filament 的表达能力，我们重点为 Aetherling [22] 生成的设计提供类型签名。

Aetherling 的时空类型。Aetherling [22] 是一种函数式数据流领域专用语言（DSL），可为图像处理任务生成静态调度的流式加速器。



Aetherling 的“时空”类型

使用户能够将数据流的形状表示为有效和无效信号组成的序列。例如，类型 `TSeq 1 1` 表示数据流中会先有一个有效元素，随后有一个无效元素。嵌套这些类型可让用户表示更复杂的形状：`TSeq 3 0(TSeq 1 1)` 表示其中有三个有效元素、没有无效值，且每个元素的形状均由 `TSeq 1 1` 描述。在我们的案例研究中，我们导入了 14 个实现两个内核的设计：二维卷积和锐化。Aetherling 的评估针对每个内核研究了 7 种设计方案，它们在资源与吞吐量之间采用不同的权衡。Filament 能够表示所有设计的接口类型，并在此过程中发现了生成接口中的若干错误。

周期精确的测试框架。 我们实现了一个通用的周期精确的 测试框架，用于测试 Filament 程序。从高层次来看：

(1) 为组件接口中精确指定的周期提供输入。 .
(2)使用事件延迟对组件的执行进行流水线处理。(3) 捕获签名中提供的区间内输出端口的值。

该测试框架通过向编译器提供一个简单的命令行标志，提取可用区间和事件延迟，并使用 cocotb Python 库执行设计 [17]。这一通用测试框架的设计依赖于类似 Filament 的系统来记录模块的时序行为；如果没有 Filament，用户就必须手动从 Verilog 代码中提取这些信息。

方法论. 我们将每个 Aetherling 设计编译为 Verilog，并使用 Aetherling 的命令行接口提取设计的延迟信息。每个基准测试包含五种充分利用资源的设计，这些设计每个周期都可以接收新的输入；此外，还包含两种资源利用不足的设计，分别每个时钟周期产生 1/3 和 1/9 个像素，并且每隔 3 个和 9 个周期接收一次新输入。我们为每个设计指定一个类型签名，并验证其输出。对于输出不匹配的设计，我们调整延迟，直到获得正确的结果。

延迟。 表 1 报告了 Aetherling 命令行接口提供的延迟，以及我们通过 Filament 的周期精确测试框架验证能够生成正确输出的延迟。在这 14 个设计中，Aetherling 报告的 5 个设计的延迟不正确。

报告的吞吐量		实际值
16	7	7
8	6	6
4	6	6
2	6	6
1	7	7
1/3	**10**	12
1/9	**16**	21

（a）报告的二维卷积延迟

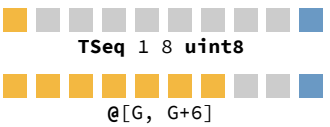
报告的吞吐量		实际值
16	7	7
8	7	7
4	7	7
2	7	7
1	8	8
1/3	**11**	13
1/9	**17**	20

(b) 锐化的报告延迟

表1。 Aetherling设计的延迟。高亮显示的延迟是Aetherling报告错误的延迟 .

利用率不足的设计。 Aetherling 探索了利用率不足的 设计的用途，这类设计每个时钟周期产生的像素少于一个。Aetherling 的编译器通过共享计算资源来优化此类设计。一个每个时钟周期产生 1/9 个像素的 Aetherling 设计，其类型为 `TSeq 1 8uint8`，表示先有 1 个有效数据项，随后有 8 个无效数据项。该类型表明，Aetherling 生成的设计只应在第一个周期使用其输入，因为后续周期提供的数据无效。然而，这个接口并不正确。

```
comp Conv2d<G: 9>(
```



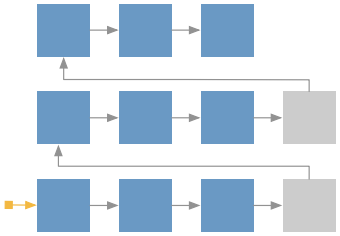
Filament类型反映了正确执行该模块所需的实际接口，它要求设计将输入信号保持六个周期，也就是说，数据元素必须连续六个周期有效，而不是仅一个周期；Aetherling的实现违反了自身的接口。Aetherling测试框架没有发现这个错误，因为它总是将所有输入信号置为有效状态，持续9个周期。相比之下，Filament测试框架只会在相应可用时间间隔指定的时长内，将输入信号置为有效状态。最后，虚事件 G 的延迟编码了这样一个事实：该设计每9个周期可以处理一个新输入。这说明，要准确描述信号可用性和流水线化的时间敏感接口，细节之处颇为微妙。

其他设计。 我们还导入了由 PipelineC 生成的设计 [30], PipelineC 是一款开源高级综合编译器，可将类 C 语言转换为 Verilog（附录 B.2）。为这些设计提供类型签名很简单，因为 PipelineC 始终会对设计进行完全流水线化，并在命令行中输出设计的延迟。

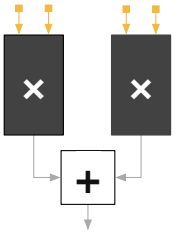
7.2 使用 Filament 进行加速器设计

我们通过在 Filament 中实现二维卷积，研究 Filament 在生成高效设计以及复用由其他语言生成的组件方面的有效性。我们构建了两种基于 Filament 的设计，并将其与由 Aetherling 生成的卷积设计进行比较。

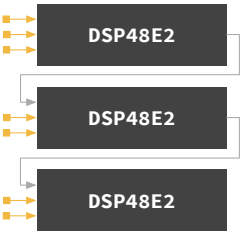
架构。 我们的实现直接受到 Aetherling 中 二维卷积 实现结构的启发，该实现每个时钟周期输出 1 个像素。该设计使用一个 3×3 滤波器，作用于一个 4×4矩阵。该 `Stencil` 模块（图 8a）实现了一个行缓冲区，用于保存最近的 11 个值，并输出与滤波器起始索引对应的 9 个值。该 `Conv2d` 卷积核将 9 个值作为输入，并生成与卷积结果对应的输出。



(a) `Stencil` 组件中的行缓冲区组件



(b) 基础设计采用流水线式三周期乘法器



(c) 基于 Reticle 的设计 uses DSP 级联

图 8. 基于 Filament 的二维卷积设计中使用的组件。模板组件提供最后三个输入，并连接到朴素乘法器或由 Reticle 生成的 DSP 级联。

Filament 中的流处理原语。 为实现行缓冲区，我们实现了一个新的 `Prev` 组件，用于输出其中存储的最后一个值。[4] 其 Verilog 实现 `Prev` 只是一个寄存器，但 Filament 为它赋予了不同的类型签名——它允许访问

```
comp Prev[SAFE]<G: 1>(
```

在提供输入的同一周期内输出，这相当于读取寄存器中的前一个值。该组件使用编译时参数 `SAFE` 来指示首次读取是否会产生未定义值。我们还定义了一个 `ContPrev` 组件，它与 `Prev` 组件相似，但使用虚事件，因此可用于连续流水线（第5.4节）。模板组件（图8a）实现为一系列 `Prev` 组件。

设计 ***1***：流水线乘法器。 基础 `Conv2D` 内核使用完全流水线化的乘法器，延迟为三个周期，并使用组合加法器。乘法器没有关联的 Verilog 实现，而是使用 Xilinx 的 LogiCORE 乘法器生成器 [2]实现。不过，Filament 通过提供类型安全的外部包装器（第3.6节），让与这些乘法器对接变得简单。

设计 ***2***：与 ***Reticle*** 集成。 我们的第二种设计使用通过 Reticle 生成的点积单元 [49]，，Reticle 是一种用于编程 FPGA 的低级语言。图 8c 展示了 Reticle 生成的架构，以利用 *DSP* 级联，从而高效利用 FPGA 上的资源。DSP 级联会显式实例化 FPGA 低级原语，并将它们连接起来以实现计算： $y = c + \Sigma_{i=0}^{3} a_i \times b_i$。与依赖综合工具从 行为 描述中推断 DSP 用法的标准编译流程不同，Reticle 会生成 结构 描述，从而实现可预测的映射

```
comp Tdot<G: 1>(clk: 1,
```

映射到数字信号处理器上。我们为 Reticle 设计提供了一个类型签名，说明输入必须以交错方式提供。请注意，这并非实现细节的泄露：每个周期都开始一次新计算的 DSP 级联，要么必须寄存所有输入，要么必须以交错方式提供输入。

评估方法。我们使用时序精确的测试框架验证所有设计的正确性，并比较各设计的面积和延迟。对于每个设计，我们逐步提高目标频率，直到最差负时序裕量小于 $0.1ns$，然后使用 Vivado v2020。2 进行综合。每个设计的吞吐量均为每个时钟周期 1 个像素。

[4] `prev` 是数据流语言和函数式响应式语言中的常见运算符。

总结。 表 2 展示了比较结果：Filament 设计可以在更高的频率下进行综合，并且比 Aetherling 设计使用更少的资源。这是因为 Filament 可以安全、直接地使用底层实现模块可直接编译为安全且高效设计的模块。相比之下，Aetherling 编译器在弥合其高级语言与底层电路之间的差距时，必须生成额外逻辑。基于 Reticle 的设计所用逻辑资源比基础 Filament 设计或 Aetherling 设计少一个数量级。这是因为 Reticle 与 Aetherling 不同，会生成底层结构化 Verilog，能够以可预测的方式映射到 DSP 资源。这体现了 Filament 作为集成平台和设计语言的实用性——Filament 设计可以安全地使用底层硬件模块，并组合由其他语言生成的复杂模块。这也揭示了 Filament 的另一种应用场景：类似 Aetherling 的语言无需直接生成 Verilog，而是可以生成 Filament 程序，使性能工程师能够进一步优化设计并消除抽象开销。

表 2. 二维卷积设计的资源使用量和频率。最佳值已突出显示。

Name	LUTs	DSPs	寄存器	频率（MHz）
Aetherling	104	10	78	769.2
Filament	128	**9**	**11**	**833.3**
Filament Reticle	**14**	**9**	20	645.1

其他设计。附录B.1 详细介绍了用 Filament 实现的其他设计：（1）浮点计算，以及（2）矩阵乘法脉动阵列 [33]。

8 相关工作

数据流语言。 响应式数据流编程语言 [8, 12, 24] 提供使用逻辑时间步调度的流操作符。软件数据流语言 [45, 48] 提供可面向多种后端（如 CPU 和 GPU）的高级声明式操作。将这些语言编译为硬件需要复杂的转换 [5, 44]。Filament 层次更低，也更易于编译，因为它直接对硬件模块进行推理，因此适合作为这些语言的硬件生成器的目标语言。

加速器设计语言。 加速器设计语言 [1, 18, 22, 25, 26, 31] 提供用于设计硬件加速器的高级抽象。Filament 是一种低级硬件描述语言，提供类型系统以直接与硬件模块交互，因此既适合作为集成语言，也适合作为 ADL 编译器的目标语言。

嵌入式硬件描述语言。 嵌入式硬件描述语言 [3, 4, 6, 16, 29, 34] 使用软件宿主语言进行元编程。大多数 eHDL 仅使用宿主语言的类型系统来确保端口宽度匹配和有符号性等简单属性。Filament 的类型系统侧重于表达硬件本身的结构和时序属性。基于规则的硬件描述语言 [7, 41] 使用受保护的原子操作 为硬件规范提供事务语义。为保留其高级语义，编译器必须生成复杂的调度逻辑，以动态中止冲突规则。相比之下，Filament 使用调用来指定程序调度，这些调用会在编译时进行检查，并可预测地映射到高效的流水线化硬件。

硬件设计的类型系统。 Dahlia [38] 是一种类 C 语言，采用次结构类型系统，确保高级程序不会违反硬件约束。Dahlia 的仿射推理可以编码到 Filament 的类型系统中。Kami [13] 是一个借助证明进行硬件设计的框架。Kami 可用于证明硬件模块的完全功能正确性，但要求用户编写证明；而 Filament 的类型系统专注于时序属性，并且能够自动工作。相比之下，Filament 专注于流水线化场景下对延迟敏感的接口。

Pi-Ware [43] 使用依赖类型来确保低层电路属性，例如确保所有端口均已连接，以及连线种类 [14] 会检查模块组合是否会产生 组合环路；这两点都与 Filament 所提供的保证无关。

会话类型。Cordial [21] 是一种受会话类型 启发的类型系统，用于推理对延迟不敏感的硬件协议；Ghica [23] 则提出了一种受博弈语义启发的类型系统来对接口进行建模；这两种工具都不对流水线化进行推理。Das 等人 [20] 为会话类型引入了线性时序逻辑中的时序模态，使其能够表达时间同步属性。尽管他们的逻辑可用于表达 Filament 的良构性属性，但由于其推理对象是所有可能发生冲突的流水线执行，因此如何编码安全流水线化约束并不明显。未来的研究方向之一是将 Filament 与会话类型领域进行形式化衔接。

时序规范。 IP-XACT [28] 是一种基于 XML 的接口定义语言，用于封装粗粒度的基于 DMA 的接口。相比之下，Filament 规定了硬件模块的底层时序行为。Synopsys 设计约束（SDC） [47] 通常用于配置 RTL 程序到物理实现的映射。虽然 SDC 可用于指定物理时序约束，例如时钟周期以及组合路径的延迟，但这些属性描述的是电路的物理实现，而非模块的时序行为。此外，这两种方案都只侧重于规范。相比之下，Filament 既能规范时序行为，也能对其进行检查。

模型检测。模型检测 [15] 是一种广泛应用于验证硬件设计的技术。SystemVerilog 断言 [50] 提供了一种基于线性时序逻辑（LTL）的规范语言，用于验证硬件设计的属性 [10, 35]。此类系统能够提供全程序保证，并且能够证明比 Filament 更通用的时序属性。Filament 专注于提供组合式保证和接口规范。Filament 签名有望被编译为线性时序逻辑规范，并通过上述工具进行检查。

9 未来工作

Filament 代表着迈向新型类型系统的一项初步探索，这类类型系统能够推理硬件设计的结构属性和时序属性。

动态流水线。虽然静态流水线涵盖了大量且颇具研究价值的硬件设计，但动态流水线对于表达复杂设计也十分重要。Filament 对流水线化约束的刻画（第 4 节）为推理动态流水线提供了形式化基础。一种可能的解决方案是允许模块除了消耗事件之外，还能生成事件；这相当于对事件类型进行存在量化，而当前系统采用的则是全称量化。其挑战在于证明这一扩展能够产生良构的动态流水线。

多态 ***Filament***。 硬件描述语言允许程序员实现具有参数化延迟的模块。例如，移位寄存器的实现可以使用一个参数来指定级数，而级数会影响其延迟。Filament 的多态扩展将允许程序员编写生成式模块，类似于嵌入式硬件描述语言，并利用类型系统保证任何有效的参数赋值都会生成安全流水线。

类型保持编译。高级综合是将 C 等命令式语言编译为硬件的过程，但其漏洞之多众所周知。以往的工作主要致力于构建完全验证的编译器 [27, 32], , 但它们未能涵盖常见的流水线优化，例如

模调度 [19]。我们认为，Filament 可用作类型化中间语言，帮助构建类型保持编译器 [36]，从而杜绝与流水线相关的高级综合编译漏洞。

10 结论

要释放可复用硬件的真正潜力，必须详细了解其实现结构和时序。Filament 通过接口呈现这些信息，使用户能够复用设计，并放心构建高性能硬件。

致谢

我们感谢 Drew Zagieboylo、Ryan Doenges、Stephen Neuendorffer、Andrew Appel、Christopher Batten 和 Zhiru Zhang 所提供的深刻见解。David Durst 的帮助使我们得以复现 Aetherling 的评估结果，并将其与 Filament 进行比较。衷心感谢匿名审稿人提供宝贵的反馈以及相关工作的线索。

本研究部分得到了架构应用驱动中心（ADA）的资助。该中心是 JUMP 计划的六个研究中心之一，而 JUMP 是由美国国防高级研究计划局（DARPA）共同赞助的半导体研究公司项目。此外，本研究还获得了美国国家科学基金会（NSF）项目资助（编号：#1845952、#2124045 和 #1909073），以及来自 Google 和 SambaNova 的捐赠。

制品

本文的制品已归档并公开[39] 为虚拟机形式。为复现我们的结果，我们建议使用 $v2020.2$。鉴于硬件综合具有非确定性，我们不期望复现结果与表2完全一致，但我们预期可以复现以下总体结论：Filament 设计占用的资源更少，运行速度更快。

参考文献

[1] AMD Inc. 2021。《Vivado 设计套件用户指南：高级综合》。UG902 (v2017.2)，2017 年 6 月 7 日。检索于 2021 年 1 月 16 日，网址：https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level- synthesis.pdf [2] AMD Inc. 2022。Xilinx LogiCORE IP 乘法器 v11.2。检索于 2022 年 10 月 27 日，网址：https://docs.xilinx.com/v/u/en- US/mult _ _ 255 gen ds C. Baaij、M. Kooijman、J. Kuper、A. Boeijink 和 M. Gerards。2010。CλaSH：使用 Haskell 对同步硬件进行结构描述。载于《Euromicro 数字系统设计会议：架构、方法与工具》。https: //doi.org/10.1109/DSD.2010.21[4] Jonathan Bachrach、Huy Vo、Brian Richards、Yunsup Lee、Andrew Waterman、Rimas Avižienis、John Wawrzynek 和 Krste Asanović。2012。Chisel：在 Scala 嵌入式语言中构造硬件。载于设计自动化会议（DAC）。https://doi.org/ 10 1145 2228360 2228584 . / . Gérard Berry。 . 纯 Esterel 的硬件实现。《Sadhana》（1992）。[5] 1992 [6] Per Bjesse、Koen Claessen、Mary Sheeran 和 Satnam Singh。1998。Lava：使用 Haskell 进行硬件设计。ACM SIGPLAN Notices (1998 10 1145 291251 289440)。https://doi.org/ . / . Thomas Bourgeat、Clément Pit-Claudel 和 Adam Chlipala。2020。Bluespec 的本质：用于基于规则的硬件设计的核心语言。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/ 10 1145 3385412 3385965 . / . Frédéric Boussinot 和 Robert De Simone。1991。ESTEREL 语言。《Proc. IEEE》（1991）。[7] [8] https://doi.org/10.1109/ 5.97299[9] Stephen Brookes。2004。并发分离逻辑的语义。载于国际并发理论会议。Springer。[10] Cadence Inc. 2022。Jasper Gold FPV 应用程序。检索于 2022 年 10 月 15 日，网址：https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html

[11] Andrew Canis、Jongsok Choi、Mark Aldham、Victor Zhang、Ahmed Kammoona、Jason H Anderson、Stephen Brown 和 Tomasz Czajkowski。2011。LegUp：面向基于 FPGA 的处理器/加速器系统的高级综合。载于国际现场可编程门阵列研讨会（FPGA）。https://doi.org/10 1145 1950413 1950423 . / . Paul Caspi、Grégoire Hamon 和 Marc Pouzet。2008。同步函数式编程：Lucid Synchrone 实验。《实时系统：描述与验证技术：理论与工具》。Hermes（2008）。[13] Joonwon Choi、Muralidaran Vijayaraghavan、Benjamin Sherman、Adam Chlipala 和 Arvind。2017。Kami：用于高级参数化硬件规范及其模块化验证的平台。载于 ACM 函数式编程国际会议（ICFP）。https://doi.org/10 1145 3110268 . / Michael Christensen、Timothy Sherwood、Jonathan Balkind 和 Ben Hardekopf。2021。连线种类：用于安全硬件组合的语言抽象。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/10 1145 3453483 3454037 . / . Edmund M Clarke。1997。模型检测。载于软件技术与理论计算机科学基础国际会议。Springer，第 54–56 页。[16] J. Clow、G. Tzimpragos、D. Dangwal、S. Guo、J. McMahan 和 T. Sherwood。2017。一种用于快速硬件原型设计和仪器化的 Python 风格方法。载于国际现场可编程逻辑与应用会议（FPL）。https://doi.org/10 23919 2017 8056860 . /FPL. . Cocotb 作者。2023。Cocotb：一种基于协程的协同仿真库，用于使用 Python 编写 VHDL 和 Verilog 测试平台。检索于 2023 年 3 月 17 日，网址：https://docs.cocotb.org/en/stable/index.html [18] Jason Cong 和 Jie Wang。2018。PolySA：基于多面体的脉动阵列自动编译。载于 IEEE/ACM 计算机辅助设计国际会议（ICCAD）。https://doi.org/10 1145 3240765 3240838 . / . J. Cong 和 Zhiru Zhang。2006。一种基于 SDC 公式的高效通用调度算法。载于设计自动化会议（DAC）。https://doi.org/10 1145 1146909 1147025 . / . Ankush Das、Jan Hoffmann 和 Frank Pfenning。2018。使用时间会话类型进行并行复杂度分析。载于 ACM 函数式编程国际会议（ICFP）。https://doi.org/10 1145 3236786 . / Jan de Muijnck-Hughes 和 Wim Vanderbauwhede。2019。硬件接口的类型纪律。载于欧洲面向对象编程会议（ECOOP）。https://doi.org/10.4230/LIPIcs.ECOOP.2019.6[22] David Durst、Matthew Feldman、Dillon Huff、David Akeley、Ross Daly、Gilbert Louis Bernstein、Marco Patrignani、Kayvon Fatahalian 和 Pat Hanrahan。2020。流式加速器的类型导向调度。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/10 1145 3385412 3385983 . / . Dan R Ghica。2009。面向硬件编译的函数接口模型：类型、签名、协议。arXiv 预印本 arXiv:0907.0749（2009）。[24] Nicholas Halbwachs、Paul Caspi、Pascal Raymond 和 Daniel Pilaud。1991。同步数据流编程语言 LUSTRE。《Proc. IEEE》（1991 10 1109 5 97300）。https://doi.org/ . / . James Hegarty、John Brunhaver、Zachary DeVito、Jonathan Ragan-Kelley、Noy Cohen、Steven Bell、Artem Vasilyev、Mark Horowitz 和 Pat Hanrahan。2014。Darkroom：将高级图像处理代码编译为硬件流水线。《ACM Transactions on Graphics》。https://doi.org/10 1145 2601097 2601174 . / . James Hegarty、Ross Daly、Zachary DeVito、Jonathan Ragan-Kelley、Mark Horowitz 和 Pat Hanrahan。2016。Rigel：灵活的多速率图像处理硬件。《ACM Transactions on Graphics》。https://doi.org/10.1145/2897824. 2925892[27] Yann Herklotz、James D. Pollard、Nadesh Ramanathan 和 John Wickerson。2021。高级综合的形式化验证。载于 ACM SIGPLAN 面向对象编程、系统、语言与应用会议（OOP- SLA）。https://doi.org/10 1145 3485494 . / IP-XACT 工作组。2009。IP-XACT。检索于 2023 年 3 月 17 日，网址：https://www.accellera.org/downloads/ standards/ip-xact [29] Jane Street。2022。HardCaml：OCaml 中的寄存器传输级硬件设计。检索于 2022 年 10 月 15 日，网址：https://github.com/janestreet/hardcaml [30] Julian Kemmerer。2022。PipelineC。检索于 2022 年 10 月 15 日，网址：https://github.com/JulianKemmerer/PipelineC[31] David Koeplinger、Matthew Feldman、Raghu Prabhakar、Yaqi Zhang、Stefan Hadjis、Ruben Fiszel、Tian Zhao、Luigi Nardi、Ardavan Pedram、Christos Kozyrakis 和 Kunle Olukotun。2018。Spatial：面向应用加速器的语言与编译器。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/10 1145 3192366 3192379 . / . Sudipta Kundu、Sorin Lerner 和 Rajesh Gupta。2008。验证高级综合。载于计算机辅助验证国际会议（CAV）。[33] Hsiang-Tsung Kung。1982。为什么要采用脉动架构？《IEEE Computer》（1982）。https://doi.org/10.1109/MC.1982. 1653825[34] Derek Lockhart、Gary Zibrat 和 Christopher Batten。2014。PyMTL：用于纵向集成计算机架构研究的统一框架。载于 IEEE/ACM 微体系结构国际研讨会（MICRO）。https://doi.

org/10.1109/MICRO.2014.50

[35] Cristian Mattarei、Makai Mann、Clark Barrett、Ross G. Daly、Dillon Huff 和 Pat Hanrahan。2018。CoSA：面向敏捷硬件设计的集成验证。载于计算机辅助设计形式化方法（FMCAD）。https://doi.org/10. 23919/FMCAD.2018.8603014[36] Greg Morrisett、David Walker、Karl Crary 和 Neal Glew。1999。从 System F 到类型化汇编语言。《ACM 编程语言与系统汇刊》（TOPLAS）（1999 10 1145 319301 319345）。

[37] https://doi.org/。/。Kevin E. Murray 2014 和 Vaughn Betz。。量化 FPGA 上延迟无关通信的成本与收益。载于现场可编程门阵列国际研讨会（FPGA）。https://doi.org/10.1145/2554688. 2554786[38] Rachit Nigam、Sachille Atapattu、Samuel Thomas、Zhijing Li、Theodore Bauer、Yuwei Ye、Apurva Koti、Adrian Samp- son 和 Zhiru Zhang。2020。使用时间敏感仿射类型实现可预测的加速器设计。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/10 1145 3385412 3385974。/。

[39] Rachit Nigam、Pedro Henrique Azevedo de Amorim 和 Adrian Sampson。2023。。《使用时间线类型的模块化硬件设计》的制品。Zenodo。https://doi.org/10 5281 7709916。/zenodo。

[40] Rachit Nigam、Samuel Thomas、Zhijing Li 和 Adrian Sampson。2021。。用于加速器生成器的编译器基础设施。载于 ACM 编程语言与操作系统架构支持国际会议（ASPLOS）。https://doi.org/10 1145 3445814 3446712。/。

[41] Rishiyur Nikhil。2004。。Bluespec System Verilog：从高级规格生成高效、正确的 RTL。载于协同设计形式化方法与模型会议（MEMOCODE）。https://doi.org/10 1109 2004 1459818。/MEMCOD。。

[42] Christian Pilato 和 Fabrizio Ferrandi。2013。。Bambu：用于综合存储密集型应用的模块化高级综合框架。载于现场可编程逻辑与应用国际会议（FPL）。https://doi。org/10 1109 2013 6645550。/FPL。。JP Pizani Flor 等。2014。。Pi-Ware：一种使用依赖类型的嵌入式硬件描述语言。硕士论文。[43] Jing Pu、Steven Bell、Xuan Yang、Jeff Setter、Stephen Richardson、Jonathan Ragan-Kelley 和 Mark Horowitz。2017。使用图像处理 DSL 为异构系统编程。《ACM 架构与代码优化汇刊》（TACO）。https://doi.org/10 1145 3107953。/ Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman P. Amarasinghe。2013。Halide：一种用于优化图像处理流水线中的并行性、局部性和重新计算的语言与编译器。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi。org/10 1145 2491956 2462176。/。John C Reynolds。。分离逻辑：一种用于共享可变数据结构的逻辑。载于 IEEE 计算机科学逻辑研讨会。IEEE。[46] 2002 [47] Synopsys Inc. 2023。Synopsys 设计约束。检索于2023年3月17日，网址：https://web.archive.org/web/20161117113913/http://www.microsemi.com/document-portal/doc_view/130022-ac152-using-synopsys-design-constraints-sdc-with-designer-app-note [48] William Thies、Michal Karczmarek 和 Saman Amarasinghe。2002。StreamIt：一种面向流式应用的语言。载于编译器构造国际会议。Springer。[49] Luis Vega、Joseph McMahan、Adrian Sampson、Dan Grossman 和 Luis Ceze。2021。Reticle：一种用于现代 FPGA 编程的虚拟机。载于 ACM SIGPLAN 编程语言设计与实现会议（PLDI）。https://doi.org/10 1145 3453483 3454075。/。Srikanth Vijayaraghavan 和 Meyyappan Ramanathan。2005。SystemVerilog 断言实用指南。Springer Science & Business Media。[51] Zhiru Zhang、Yiping Fan、Wei Jiang、Guoling Han、Changqi Yang 和 Jason Cong。2008。AutoPilot：一种基于平台的 ESL 综合系统。载于高级综合。99–112。

$$
\begin{aligned}
&[\![c]\!] : \mathrm{L}og \to \mathrm{L}og\,(\mathrm{L}og = \mathrm{T}\ \to \mathrm{R}\ \times \mathrm{W})[\![x := \text{新建}\, C]\!] && = \text{标识符}\\
&[\![p_d = p_s]\!] && = \lambda(R, W)。\text{如果}\ p_s \in W\text{则}\ (R\{p_s/p_d\}W)\ \text{否则}\ (R,\\
&W)[\![c_1 \cdot c_2]\!] && = [\![c_1]\!] \cup [\![c_2]\!][\![x_1 := \text{调用}\ x_2 T'(q_1, \cdots\ q_m)\\
&]\!] && = [\![\text{连接}(x_1, [q_1, \ldots, q_m])]\!] \circ\ [\![x_2]\!][\![M]\!] && : \mathrm{P}[\![\text{定义}\\
&Ct\colon\ n(i_0\colon\ \pi_0, \ldots, o_0\colon\ \pi_i, \ldots)\{c\}]\!] = \{\pi_0 \mapsto i_0, \ldots\} \times \{\pi_i \mapsto o_0, \ldots\}
\end{aligned}
$$

图 9. Filament 核心语言的日志转换语义。每条命令都会生成一个日志（L），将事件（T）映射为读取集（R）和写入集（W）的多重集。组件定义通过将输入的可用性映射为读取集、将输出的可用性映射为写入集，从而生成部分日志（P）。

一种完整的形式体系

A.1 语法

我们在形式化中使用了 Filament 的简化版本：所有组件都可以通过恰好一个约束进行参数化，且无法指定事件之间的任何排序约束。这两种简化都不会损失一般性，因为具有多个事件的用户级组件无法定义它们之间的任何交互——它们在功能上等同于具有互不相交事件的多个组件。

Filament 程序是由一系列组件构成的 M，每个组件都封装了流水线的结构和调度。命令 c 包括组合、端口连接（第 3.5 节）、组件实例化（第 3.3 节）以及调用（第 3.4 节）。组件使用且仅使用一个事件进行参数化，而调用则允许使用一个事件进行调度。

$$
\begin{aligned}
&x, C \in vars \quad t \in events\\
&p, q \in ports\\
&M ::=\\
&\quad \textbf{def}\ C\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)\{c\}\\
&c ::= c_1 \cdot c_2 \mid p_d = p_s \mid x := \textbf{new}\ C\\
&\quad \mid x := \textbf{invoke}\ x\langle T\rangle(p_1, \ldots, p_j)\\
&T ::= t \mid T + n \quad \pi ::= [T_1, T_2]
\end{aligned}
$$

A.2 语义

我们语义的基础是读取和写入的日志。直观来说，每个命令都会生成一个从事件到读取和写入（多重）集的函数，用以指明在特定事件中读取或写入了哪些端口。更形式化地说，每个命令都被解释为一个作用于日志的函数，如图 9 所示。该图将 Filament 程序指称为一个将事件转换为端口读取多重集（$\mathcal{L}$）、$\mathcal{R}$以及端口写入多重集（$\mathcal{W}$）的日志转换器。由于组件恰好只能使用一个事件，例如 T，日志会将诸如 T、$T+1$ 等事件映射为对子组件所定义端口的读取和写入。实例化不会影响日志，而连接会通过将 LHS 端口添加到 RHS 端口所定义事件的写入集，来改写日志。并行组合则被解释为两个命令所生成日志的并集。

最后，调用会根据其签名生成一个日志转换器：对于各自可用时间间隔内包含的每个事件，都将输入端口加入读取集（R），并将输出端口加入写入集（W）。不过请注意，签名所给出的语义使用的是

“不正确”的端口，因为它使用的是签名中给出的端口。我们通过使用 connects 元函数来解决这个问题，该元函数定义为

$$\text{connects}(x_1, [q_1, \cdots q_m]) = (x_1.p_1 = q_1); \cdots ; (x_1.p_m = q_m),$$

这里我们隐式假设端口 $x_1.p_i$ 尚未使用。直观来说，这个元编程将调用的参数转换为连接，并通过使用命令上的指称对它们进行解释来生成日志。程序的语义由一个特殊顶层组件生成的日志定义。

组件及其语义。 例如，一个组合加法器和一个延迟为两个周期的时序乘法器会生成以下日志：

<escape>⟦def 加法器G: 1{l: $[G, G+1]$</escape>，<escape>r: $[G, G+1]$</escape>，
<escape>out: $[G, G+1]$}⟧ $= G \rightarrow$ ({ l</escape>，<escape>r }</escape>，<escape>{ go</escape>，
<escape>out }⟦def mul$G$: 2{l: $[G, G+1]$</escape>，<escape>r: $[G, G+1]$</escape>，
<escape>out: $[G+2, G+3]$}⟧ $= G \rightarrow$ ({ l</escape>，<escape>r }</escape>，<escape>{ go }$G+1 \rightarrow$
($\emptyset$</escape>，<escape>{ go }$G+2 \rightarrow$ ($\emptyset$</escape>，<escape>{ out })</escape>

请注意，对实例的使用会反映在其接口端口上的写入中 go （签名中未显示）。日志表明，乘法器每 2 个周期就会接受新值：在这两个周期 G 和 $G+1$ 中，都会写入 go 端口。由于我们跟踪读取和写入的多重集，因此可以跟踪对同一端口的冲突写入。

Usin据此定义语义，我们可以定义日志上的良构性约束（第 4.2 节） :

定义 *A.1*（良构性）。当且仅当对于所有事件，日志 L 都是良构的

- 不存在冲突写入：$W_s = W$，其中 W_s 是去重后的写入集。
- 对于每个事件，读取都是写入的子集：$R \subseteq W_s$。

而在真实硬件中，端口或导线上始终都有值可用，Filament 的语义只跟踪读取时语义上有效的值。硬件资源的使用由写入表示，而在物理上不可能向一个端口写入两个值；电路会改用多路复用器，在两个值之间进行选择。多次使用同一资源会悄无声息地破坏数据。

安全流水线约束（第 4.4 节）可以通过程序语义的重复执行来定义：

定义 *A.2*（安全流水线）。如果组件 M 有一个延迟为 d 的事件 T，且 $[\![M]\!]_G$ 表示其日志，其中 T 被替换为事件 G，那么当且仅当对于每个 $n \geq d$，日志 $L_n = [\![M]\!]_T \cup [\![M]\!]_{T+n}$ 都格式良好时，M 才是安全流水线化的。

A.3 类型系统

本文重点介绍 Filament 的子结构类型系统，该系统用于跟踪资源的无冲突使用以及信号的有效性。我们省略了对端口宽度等标准特性的介绍。

类型上下文。类型判断使用以下类型上下文：

- Γ 跟踪组件和实例的类型以及端口的可用性。
- Δ 跟踪上下文中与每个事件相关的延迟。
- Λ 是时间线上下文，用于跟踪每个实例和端口的可用性。

$$\tau ::= \forall\langle t : n\rangle(p_1 : \pi_1, \ldots, p_j : \pi_j)$$
$$\Gamma ::= \cdot \mid \Gamma, C : \tau \mid \Gamma, p : \pi$$
$$\Delta ::= \cdot \mid \Delta, t : n$$
$$\Lambda ::= \cdot \mid \Lambda, I : \pi \mid \Lambda, p : \pi$$

类型上下文（Γ）和时间线上下文（Λ）分别存储端口和实例的时间线。端口的时间线是可复用的，因为在一个周期内读取端口不会消耗它。然而，实例的时间线在一个周期内被使用时会被消耗。因此，时间线上下文还提供了一种分离并集，其灵感源自分离逻辑 [46]。

拆分时间线。时间线上下文的有效分离拆分 $\Lambda = \Lambda_1 * \Lambda_2$ 当且仅当 Λ_1 和 Λ_2 绑定完全相同的实例，且对于每个实例，这些时间线彼此不相交。形式化地：

$$\Lambda = \Lambda_1 * \Lambda_2 \text{ iff } \forall (I:\pi) \in \Lambda \Rightarrow \exists \pi_1, \pi_2 . (I:\pi_1) \in \Lambda_1 \wedge (I:\pi_2) \in \Lambda_2 \wedge \pi_1 \cap \pi_2 = \emptyset \wedge \pi_1 \cup \pi_2 = \pi$$

实例化组件。实例化一个模块会将组件的签名绑定到该实例，并使资源在程序的整个时间线中可用，记作 $[0,\infty)$。

$$\frac{\Gamma(C) = \tau \qquad \Gamma' = \Gamma, I:\tau \qquad \Lambda' = \Lambda, I:[0,\infty)}{\Delta, \Lambda, \Gamma \vdash I := \textbf{new}\ C \dashv \Lambda', \Gamma'}$$

端口连接。连接端口时，会检查源端口的可用时间是否至少与目标端口所需的时间一样长：

$$\frac{\Lambda(p_d) \subseteq \Gamma(p_s)}{\Delta, \Lambda, \Gamma \vdash p_d = p_s \dashv \Lambda, \Gamma}$$

通过组合拆分时间线。组合规则使用分离拆分运算符拆分时间线上下文，并检查这两条命令。请注意，两条命令使用相同的类型上下文 Γ，这意味着先前定义的端口在两条命令中均可用：

$$\frac{\Delta, \Lambda_1, \Gamma \vdash c_1 \dashv \Lambda_1', \Gamma_1 \qquad \Delta, \Lambda_2, \Gamma \vdash c_2 \dashv \Lambda_2', \Gamma_2}{\Delta, \Lambda_1 * \Lambda_2, \Gamma \vdash c_1 \bullet c_2 \dashv \Lambda_1' * \Lambda_2', \Gamma_1 \cup \Gamma_2}$$

检查调用。调用规则强制满足良构性和安全流水线化约束，因此内容相当冗长。我们将调用的类型检查拆分为三组前提条件，在逻辑上分别对应第 4 节中介绍的属性。

$$\frac{\text{valid reads} \qquad \text{no conflicts} \qquad \text{safe pipelining}}{\Delta, \Lambda, \Gamma \vdash x := \textbf{invoke} I\langle T\rangle(q_1, .., q_j) \dashv \Lambda, \Gamma''}$$

第一组前提用于检查调用中提及的所有端口的读取是否有效，即它们的可用时间至少与实例签名所要求的一样长。最后，调用会将与该实例关联的所有端口的可用性绑定到类型上下文中。

$$\Gamma(I) = \forall \langle t:n\rangle (p_1:\pi_1, .., p_j:\pi_j) \qquad \Gamma(q_1) = \pi_1, .., \Gamma(q_j) = \pi_j$$
$$\pi_1' = \pi_1[t/T], .., \pi_j' = \pi_j[t/T] \qquad \Gamma(p_1) \subseteq \pi_1', .., \Gamma(p_j) \subseteq \pi_j'$$
$$\Gamma' = x.\{p_1:\pi_1', .., p_j:\pi_j'\} \qquad \Gamma'' = \Gamma \cup \Gamma'$$

下一组前提确保该实例在当前时间线上下文中可用。这可以确保设计的其他任何位置都不会发生对该组件的冲突使用。

$$\Gamma(I) = \forall \langle t:n\rangle (p_1:\pi_1, .., p_j:\pi_j)$$
$$\Lambda(I) = \pi \qquad [T, T+n] \subseteq \pi$$

组合规则负责选择有效的划分，以确保满足上述规则的约束。如果不存在这样的划分，则程序中对该实例存在冲突使用。

最后一组检查用于验证调用流水线化的安全性（第4.4节）：

$$\Gamma(I) = \forall \langle t:n\rangle (p_1:\pi_1, .., p_j:\pi_j)$$
$$\mathcal{E}(T) = t' \qquad \Delta(t') \geq \Delta(t)$$

A.4 类型可靠性

我们的类型系统保证定理聚焦于良构性属性（第 4.2节）。它指出，类型正确的命令会保留良构日志。我们的语义还满足第二个可靠性属性：类型正确程序生成的日志转换器只能将可用端口添加到日志的写入集中。以下定理对此进行了说明。

引理 A.3（可用性健全性）。若 $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$，则对于每个日志 $\mathcal{L}$ 和每个事件 T，令 $(\mathcal{R}, \mathcal{W}) = [\![c]\!](\mathcal{L}\ T)$，则 $p \in \mathcal{W}$ 当且仅当 $p \in \pi_2(\mathcal{L}(T))$ 或 $T \in \Lambda(p)$。

证明。该证明通过对类型推导进行归纳得出。第一种情况是显然的，因为恒等函数将任何日志映射为其自身。端口连接情况不会修改写入组件，因此与恒等情况类似。组合操作由归纳假设得出。对于调用情况，考虑转换后日志的写入中的一个端口 p，并假设 $p \notin$ L。根据构造，该 p 必须是可用的输出端口之一，这一点由类型规则保证。另一方向通过分类讨论得出。□

将上述定理专门应用于组合 $c_1 \cdot c_2$，并利用这样一个事实：根据假设，c_1 和 c_2 的 Λ 上下文互不相交，可得：

推论 A.4（写入集互不相交）。如果 $\Delta; \Lambda; \Gamma \vdash c_1 \cdot c_2 \dashv \Lambda'; \Gamma'$，那么对于日志 $(\mathcal{R}_1, \mathcal{W}_1) = [\![c_1]\!](\mathcal{L})$ 和 $(\mathcal{R}_2, \mathcal{W}_2) = [\![c_2]\!](\mathcal{L})$，$\mathcal{W}_1 - \mathcal{L}$ 与 $\mathcal{W}_2 - \mathcal{L}$ 互不相交。

定理 A.5（健全性性质）。如果 $\Delta; \Lambda; \Gamma \vdash c \dashv \Lambda'; \Gamma'$，那么若 $\mathcal{L}$ 是良构的，则 $[\![c]\!] - \mathcal{L}$ 也良构。

证明。通过对 c 的类型推导进行归纳即可得证。

- 实例化情形：根据假设，L 是良构的。
- 连接情形：根据假设，L 是良构的，因此 W 是一个集合。对于第一个条件，有两种可能：要么是 $p_d \in$ R，要么是 $p_d \notin$ R。如果第二种情况成立，那么 R$\{p_d \mapsto p_s\}$ = R $\subseteq$ W。如果 $p_d \in$ R，那么根据类型规则的假设，p_d 的可用性是 p_s 可用性的子集；并且根据输入日志的良构性，$p_s \in$ W，由此可推出 R$\{p_d \mapsto p_s\} \subseteq$ W。
- 复合情形：根据归纳假设 $\mathcal{L}_1 = [\![c_1]\!](\mathcal{L})$ 和 $\mathcal{L}_2 = [\![c_2]\!](\mathcal{L})$ 均为良构，这意味着 $\mathcal{R}_1 \subseteq \mathcal{W}_1$ 和 $\mathcal{R}_2 \subseteq \mathcal{W}_2$。根据并集运算的单调性，$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \subseteq \mathcal{W}_1 \cup \mathcal{W}_2 = \mathcal{W}$ 且第一个良构性条件成立。此外，根据推论 A.4，两个日志的写入集互不相交，因此 $[\![c_1 \cdot c_2]\!](\mathcal{L})$ 是良构的。
- 案例 InvoKe：调用规则所执行的唯一新增写入是实例的输出端口。根据构造，它们构成一个集合。为了证明读取集是写入集的子集，需要注意，实例生成的日志是故意不符合格式的，因为它使用占位符名称作为读取项的写入项。确保日志格式正确是连接操作的职责。请注意，根据定义，只有当源端位于写入日志中时，端口连接的语义才会修改日志。因此，如果要证明读取集是写入集的子集，连接语义中的每个守卫条件都必须为真，以确保日志中不存在占位符端口。这可由以下事实得出：根据构造，每个端口连接都将具有正确的类型，并且根据引理 A.3，我们可以得出结论。

□

B 评估

B.1 Filament 设计

我们在 Filament 中实现了多种设计，以评估其表达能力。每种设计都有一个对应的基准实现，该实现取自手工编写的实现或自动生成的设计。

浮点加法。我们提取了一个使用 Verilog 实现的浮点加法器。[5] 该实现提供了单周期版本和流水线化版本。随后，我们将流水线式实现转换为 Filament 设计，并使用模糊测试框架来确保该实现的输出与源实现一致。在转换过程中，我们发现了原始实现中的几个错误。每个错误都是由于流水线设计访问了前一个流水线级的信号而导致的。在 Filament 中，这类错误会立即显现，因为类型检查器禁止使用前一个级的信号来计算当前级信号的值。例如，在流水线的第二级中，加法器试图使用来自前一级的一个值：

```
// The suffix *_s2 indicates that the signal is used in the second stage.
if (exponent != 0) begin
  L1_s2 = {1'b1, Large_s2[22:1]};
end else begin
  L1_s2 = Large_s2;
end
```

在 Filament 中，类型检查器会捕获此错误：

```
M := new Mux[23];
l1_mant := M<G+1>(e2_is_zero_s2.out, large_mant_s2.out, normed_large.out);
                  Available in [G, G+1] but required in [G+1, G+2]
```

使用周期精确的测试框架，结合简单的模糊测试器，通过对组合逻辑、流水线化和 Filament 实现进行差分测试，可以轻松发现此类错误。我们从这项研究中得出的结论是：一旦 *Filament* 设计在功能上与流水线化设计一致，由于类型检查，对 *Filament* 代码进行流水线化改动不太可能引入新错误。

脉动阵列。 脉动阵列 [33] 是实现线性代数运算的一种常见设计模式。它们通过将计算安排在由处理单元组成的二维阵列中，模拟数据的二维结构，并利用计算之间的数据复用。我们采用了 Calyx [40] 论文中报告的基线脉动阵列，并用 Filament 实现了它。我们使用了 `Prev` 在第 7.2 节中实现的组件来表达脉动阵列的计算。

```
comp main<G: 1>(@interface[G] go: 1, @[G, G+1] l0: 32, @[G, G+1] l1: 32, @[G, G+1] t0: 32,
```

[5] 来源ce: https://github.com/suhasr1991/5-Stage-Pipelined-IEEE-Single-Precision-Floating-Point-Adder-Design.

```
  pe01 := new Process<G>(r00_01.prev, t1);
  pe10 := new Process<G>(l1, r00_10.prev);
  pe11 := new Process<G>(r10_11.prev, r01_11.prev);

  out00 = pe00.out; out01 = pe01.out; out10 = pe10.out; out11 = pe11.out;
}
```

处理单元 Process 执行乘累加操作：

```
comp Process<G: 1>(
  @interface[G] go: 1, @[G, G+1] left: 32, @[G, G+1] right: 32) -> (@[G, G+1] out: 32) {
  // If acc does not contain a valid value, use 0
  acc := new Prev[32, 0]<G>(add.out);
  go_prev := new Prev[1, 1]<G>(go);
  mux := new Mux[32]<G>(go_prev.prev, acc.prev, 0);

  mul := new MultComb[32]<G>(left, right);
  add := new Add[32]<G>(mux.out, mul.out);

  out = add.out;
}
```

由于使用组合乘法（MultComb）而非流水线乘法（Mult），该处理单元的运行频率会较低。不过，改用流水线乘法即可轻松解决这一问题，因为这只会改变处理模块的延迟。我们的另一种设计使用流水线乘法器（FastMult），这会改变设计的延迟。

```
comp Process<G: 1>(@interface[G] go: 1,  @[G, G+1] left: 32,
```

B.2 类型签名

我们从 PipelineC 导入设计 [30] 并在 Filament 中为它们提供类型签名。

浮点加法。 PipelineC 生成的浮点加法会由编译器根据用户提供的频率目标自动进行流水线化。我们使用附录B.1 中实现的浮点单元对该模块进行差分测试，并确保为该模块提供了正确的类型签名。

```
comp FpAdd<G: 1>(clk: 1, @[G, G+1] my_pipeline_x: 32,
```

高级加密标准（***AES***）. 我们给出一个采用 PipelineC 实现并由编译器自动进行流水线化的 AES 模块的类型签名：

```
comp AES<G: 1>(clk: 1, @[G, G+1] state_words: 128,
```