arXiv:2603.17099v1 [cs.PL] 17 Mar 2026

Verilog设计的向量化及其对验证与综合的影响

Maria Fernanda Oliveira Guimarães
UFMG 贝洛奥里藏特，巴西
maria.guimaraes@dcc.ufmg.br

Ulisses Rosa
UFMG 贝洛奥里藏特，巴西
ulissesrosa@dcc.ufmg.br

Ian Trudel
独立研究员 魁北克，加拿大
ian.trudel@researchcenter.io

João Victor Amorim Vieira
Cadence 贝洛奥里藏特，巴西
jvamorim@cadence.com

Augusto Amaral Mafra
Cadence 贝洛奥里藏特，巴西
augusto@cadence.com

Mirlaine Crepalde
Cadence 巴西贝洛奥里藏特
mirlaine@cadence.com

Fernando Magno QuintãoPereiraUFMG 巴西贝洛奥里藏特
fernando@dcc.ufmg.br

摘要

向量化是一种编译器优化，它会将针对标量值的多个操作替换为针对向量值的单个操作。虽然向量化在 rustc、clang 和 gcc 等传统编译器中十分常见，但在 Verilog 生态系统中却并不常见。这是因为，尽管 Verilog 支持向量表示法，但该语言并不保证向量化信号会以字级实体的方式运行：综合工具仍会将多个独立赋值和单个向量赋值解析为同一组并行连线。然而，向量化在其他领域具有重要优势。尤其是，即使底层硬件保持不变，它也能降低符号复杂度。Cadence® Jasper® 等形式验证工具在符号层面运行：它们推理布尔函数、状态转换和等价类，而非单个连线或门。当这些工具能够将总线视为单个符号实体时，其扩展效率会更高。本文通过介绍一种 Verilog 向量化器来支持这一观察结果。该向量化器构建于 CIRCT 编译基础设施之上，能够识别多种向量化模式，包括反相赋值、涉及复杂表达式的赋值以及模块间赋值。我们在若干电子设计自动化（EDA）工具中对其进行了试验；对于 Jasper 工具，在 ChiBench 集合中的 1,157 个设计上，它将展开时间缩短了 28.12%，并将内存消耗降低了 51.30%。

CCS 概念： • **软件与工程** → **编译器**。

关键词： Verilog、向量化、验证、优化

1 引言

向量化是一种经典的编译器优化，它将标量操作替换为针对更宽数据类型的等效操作 [2]。它被用于 `gcc` 与 `clang` 等软件编译器中，在 `-O2` 与 `-O3` 优化级别下提升运行时性能 [3]。只要适用，向量化就非常有效。用 Maleki等人 [10], 的话来说：“向量化是能够对性能产生重大影响的编译器变换之一。”

尽管向量化在经典编译器中十分重要，但处理 Verilog 设计的编译器和工具中却缺少向量化。在硬件层，向量最终会被实现为彼此独立的导线和位级逻辑，因此综合工具会在流程的早期将向量信号降为标量。这种做法使 EDA 社区普遍认为，恢复向量结构几乎没有什么益处。因此，主流 Verilog 处理框架（如 CIRCT（电路中间表示编译器与工具）和 Yosys）并未提供分析或变换，将相关标量信号重新组合成结构化向量。

硬件描述语言的向量化。本文证明，即使应用于硬件描述语言领域，向量化也能带来益处。具体而言，本文提供证据支持以下论点：

> “尽管 *Verilog* 设计的向量化不会改变其所描述的硬件，但会降低其符号复杂度，从而实现更快、更具可扩展性的分析与验证。”

为证实这一论点，章节2 介绍了三种专为 Verilog 设计量身定制的变换，而章节3 则描述了如何通过静态编译器分析和源到源重写来实现这些变换。

空间向量化与时间向量化。 本文提出的向量化形式不同于经典向量化。在这两种情形下，向量化都包括识别重复出现的计算模式，并将其合并为一种更紧凑的表示。然而，经典编译器向量化侧重于时间重复，也就是在循环的多次迭代中重复执行相同操作。这种观点源于 Allen 的开创性研究等 [2]，至今仍是现代向量化器的核心 [1]。

相比之下，本文介绍的向量化针对 RTL 设计中出现的空间重复。正如章节2所示，Verilog 描述通常通过重复的位级赋值来表达字级行为，这些赋值仅在常量索引上有所不同。我们的方法会检测在结构上等价、但作用于不同信号的赋值组，并将其改写为单个向量级表达式。因此，传统向量化以循环为中心，而我们的技术则能在没有显式迭代结构的情况下，识别结构上重复的逻辑。从这个角度看，所提出的向量化器在概念上更接近于识别并提取重复代码模式的技术 [4, 15]，而非经典的基于循环的向量化。

结果摘要。章节4介绍了在 CIRCT 基础设施中实现的向量化器。我们在 ChiBench [12] 集合中的 1,157 个设计上评估了这一实现。章节4.3表明，向量化使 Jasper 的 RTL（寄存器传输级）展开时间缩短了 28% 以上，并将其内存消耗降低了 50% 以上。章节4.4 表明，向量化还使 Cadence Genus 的展开时间平均缩短了 5 % 以上，个别基准测试的改进幅度则大得多。这些结果促使 CIRCT 社区接受了所提议的向量化器，将其作为标准优化；自 2026 年初以来，该向量化器已在其开源发行版中提供。

2 概述

Verilog 设计向量化是指检测能够共同实现字级表达式的位级操作组，并将其替换为等效向量操作的过程。本文讨论的实现支持以下三类转换的任意组合：

一般位排列： 对信号进行任意重新排序，并将其合并为单个向量赋值。

模块间向量化： 有选择地内联结构相似的模块实例，以发掘跨模块边界的向量化机会。

复杂模式： 将针对单个位定义的逻辑与算术运算合并为一个向量级表达式。

本章节剩余部分将通过一系列示例逐一说明这些模式，首先从示例2.1。

示例 2.1。 图 1展示了将一系列位赋值向量化后的效果。一般赋值的向量化面向这样一类设计：输出向量的各个位是单个输入向量各位的任意重排。尽管这类设计通常写成多个彼此独立的位级赋值，但它们共同实现了一一排列。在此示例中，若令 π：{0， 1， 2， 3} → {3， 1， 2， 0} 为一个排列，则向量化器将实现赋值 输出[i] = 输入[$\pi(i)$]。

原始设计：

```
模块 bit_mixing_向量化(输出线网 [3:0] 输出, 输入线网 [3:0] 输入);
    赋值 输出[3] = 输入[0];
    赋值 输出[1] = 输入[2];
    assign 输出[2] = 输入[3];
    assign 输出[0] = 输入[1];
模块结束
```

向量化设计：

```
模块 位_混合_向量化(输出线网 [3:0] 输出, 输入线网 [3:0] 输入);
    assign 输出 = {输入[0], 输入[3:1]};
模块结束
```

图 1。一般位排列。

正如章节 3.3 将要说明的，向量化器通过追踪每个输出位的来源，并验证所有位都源自同一个输入向量且没有重复，来检测位排列模式。这类排列会被压缩为使用拼接和切片表示的单个向量赋值。这种转换保留了原有的连线语义，同时降低了符号复杂度。正如示例 2.2 所示，所提出的向量化器支持跨模块边界识别一般排列。

示例 2.2. 图 2 展示了模块间向量化。向量化器的这一组件用于处理可向量化行为分布在多个模块实例中的情况。在这些设计中，结构相同的子模块（例如示例中的 `my_buf`）会被反复实例化，每个实例都对向量中的一位进行操作。

章节3.2 将说明向量化器如何应用选择性内联，以揭示模块边界内的冗余结构。这种转换使后续的位级或结构分析能够识别底层的向量模式。内联后，重复的标量实例会被替换为单个向量级操作，同时不改变模块接口或综合后的硬件行为。示例 2.3说明，向量化可以扩展到涉及任意数量逻辑运算或算术运算的赋值。

示例 2.3. 图 3 展示了复杂模式向量化的一个案例。这一类别涵盖重复的逐位逻辑或

原始设计：

```
模块 inter模块_向量化(输出线网 [3:0] 输出, 输入线网 [3:0] 输入);
    mybuf 缓冲区_实例1(输出[3], 输入[3]);
    mybuf 缓冲区_实例2(输出[2], 输入[2]);
    mybuf buf_inst3(输出[1], in[1]);
    mybuf buf_inst4(输出[0], in[0]);
模块结束

模块 mybuf(输出线网 o, 输入线网 i);
    assign o = i;
模块结束
```

向量化设计：

```
模块 inter模块_向量化(输出线网 [3:0] 输出, 输入线网 [3:0] 输入);
    赋值 输出 = 输入;
模块结束
```

图 2. 模块间向量化。

对一个或多个输入向量的每个位独立进行运算的算术表达式。典型示例包括条件选择、逻辑组合，或以逐位赋值形式表示的算术操作。

原始设计：

```
模块 pattern_recognition(
    输出线网 [3:0] result, 输入线网 [3:0] a、b, 输入线网 sel
);
    赋值 result[3] = (a[3] & sel) | (b[3] & ~sel);
    赋值 result[2] = (a[2] & sel) | (b[2] & ~sel);
    assign result[1] = (a[1] & sel) | (b[1] & ~sel);
    assign result[0] = (a[0] & sel) | (b[0] & ~sel);
模块结束
```

向量化设计：

```
模块 pattern_recognition(
    输出线网 [3:0] result, 输入线网 [3:0] a, b, 输入线网 sel
);
    wire [3:0] _GEN = {4{sel}};
    赋值 result = a & _GEN | b & ~_GEN;
模块结束
```

图 3. 复杂逻辑与算术模式。

章节3.4 介绍了如何识别示例 2.3 中的复杂模式。虽然这些模式无法被识别为简单置换，但向量化器会通过验证每个输出位是否由独立且结构等效的逻辑锥计算得出，来检测这些模式。如果满足这一条件，就会将重复的标量表达式合并为单个向量级表达式，从而得到更紧凑且符号效率更高的表示。

一个组合流水线。示例2.1至2.3突出展示了所提出向量化方法的一个关键方面：它能够识别结构重复代码中的可向量化行为。正如章节1所讨论的，这一视角不同于经典向量化技术；后者主要针对通过循环表达的映射与归约等模式。这种从时间递归到空间递归的视角转变是必要的，因为标准 Verilog 缺乏具有类似软件语言中明确执行语义的循环构造。相反，字级行为通常通过复制的位级结构来表达。所提出的向量化器能够识别任意组合情况下的此类重复结构，包括涉及复杂算术与逻辑表达式的排列的模块间向量化。章节3 详细说明了如何检测与这些情况中的每一种相关联的模式，并将其转换为向量化赋值。

3 实现

本文的向量化器基于 CIRCT 实现，并遵循图 4所示的流程。请注意，尽管分析和优化是在 CIRCT 的低级表示上进行的，用户仍会将其视为源到源变换。因此，向量化还具有提升代码可读性的额外好处。

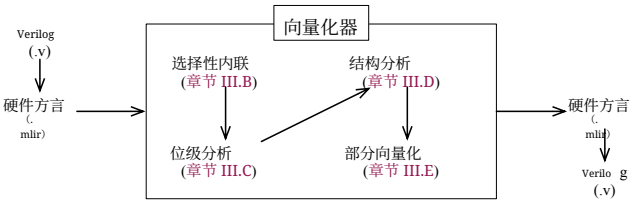

图 4。 向量化流水线。

3.1 CIRCT：底层基础设施

CIRCT 是一个用于硬件设计的开源编译器基础设施，构建于 MLIR [8]之上。它支持 Verilog 和 Chisel 等语言。CIRCT 包含多个领域专用的中间表示，每种表示都以 MLIR 方言的形式实现。本文的向量化实现是在 硬件（hw）方言中进行的，该方言引入了定宽整数、数组和结构体等数据类型，并定义了模块边界和端口。在 CIRCT 中，组合行为和时序行为通过专用方言表示，例如 comb 与 seq。例如，hw 定义模块（hw.module）、实例化（hw.instance）和连线（hw.wire），而 comb 提供诸如 comb.and 与 comb.mux之类的逻辑运算。示例 3.1展示了这些不同方言中的构造。

示例 3.1。 图 5(a) 展示了一个简单的 Verilog 赋值语句，图 5(b) 展示了其在 CIRCT 中的等效规范。当降低到 hw 与 comb 方言时，该设计在结构上表示为四条相互独立的位级连接。每个 comb.extract 操作都会从输入向量中提取一位，并将每一位独立连接到对应的输出位置。该hw 表示继承了 LLVM 生态系统中的一些约定。例如，图 5(b) 中的程序采用静态单赋值（SSA）形式 [5], ，这意味着每个变量只定义一次。此外，每个值都具有显式类型；例如，i1 （单个位）是变量%out_3的类型。

```
模块 simple_ex(
    输出线网 [3:0] 输出,
    输入线网 [3:0] in
);
    赋值 out[3] = in[3];
    赋值 out[2] = in[2];
    赋值 out[1] = in[1];
    assign 输出[0] = in[0];
模块结束
```

(**a**)

```
hw.module @simple_ex(
    %in: i4,
    %输出: !硬件输出<i4>
) {
    %in_3 = comb.extract %in 来自 3: (i4)->i1
    %in_2 = comb.extract 从 2 提取 %in: (i4)->i1
    %in_1 = comb.extract 从 1 提取 %in: (i4)->i1
    %in_0 = comb.extract 从 0 提取 %in: (i4)->i1

    %输出_3 = 硬件输出 %in_3: i1
    %输出_2 = 硬件输出 %in_2: i1
    %输出_1 = 硬件输出 %in_1: i1
    %out_0 = 硬件输出 %in_0: i1
}
```

(**b**)

图5。 (a) Verilog设计。(b) CIRCT 中的规范。

3.2 选择性内联

第 3.3、3.4 和 3.5章节中描述的分析与变换属于模块内操作；也就是说，它们在单个 Verilog 模块的边界内运行。为了实现模块间向量化，我们对子模块进行选择性内联。内联是指将一个模块的主体复制到其实例化的位置。“选择性”意味着，一个模块可以在某些实例化位置进行内联，而在其他位置不进行内联。这种方法是必要的，因为完全内联可能导致代码规模膨胀。

在不实际执行内联并重新运行第 3.3、3.4 和 3.5 章节中的分析的情况下，无法确定某项内联决策是否有助于向量化。为避免此类迭代，同时缩短编译时间，内联决策由两个用于估计模块规整性和递归规模的启发式方法来指导：

- **规整性**分析用于确定被调用模块是否由简单的组合逻辑组成；也就是说，该逻辑是否能够被向量化器处理。
- **规模**分析会计算被调用方中的操作总数，包括其中实例化的所有子模块中的操作。如果累计操作数低于 150 条硬件指令，则允许内联。

之所以强调规则性，是因为3.3 和3.4 章节中描述的分析只能对由简单组合运算构成的模块进行推理。150 条指令的阈值是通过一项实证实验选定的，该实验将在4.6 章节中介绍。这个数值是在向量化机会与设计规模之间作出的折衷。

3.3 位级数据流分析

这项静态分析会识别出一些标量赋值组，它们共同实现了位排列，如图1所示。该分析沿赋值链传播位级数据流，只有在每个输出位都能追溯到单个源向量、从而形成一一排列时，才会成功。

算法1概述了将多个位级赋值分组成单个赋值的过程。该遍维护一个位到源映射 π，其灵感来自 `BitArray`，该映射将每个输出位与其源值关联起来和源索引。如果映射不是双射，或者某一位经过了不支持的操作，则分析会失败。

算法 1 位排列向量化

阶段 1：位来源分析
对于 $i = 0$ 至 $N - 1$ **执行**
  跟踪 $out[i]$ 的数据流
  如果 $out[i]$ 并非源自单个源向量 **then**
    返回失败
  结束条件判断
  记录 $\pi[i]$，使得 $out[i] = in[\pi[i]]$
结束循环
如果 π 不是从 $\{0, \ldots, N-1\}$ 上的双射 **则**
  返回 失败
结束条件判断
阶段 2：贪心分组
$i \leftarrow 0$
当 $i < N$ **执行**
  $j \leftarrow i$
  当 $j + 1 < N$ **与** $\pi[j+1] = \pi[j] + 1$ **执行**
    $j \leftarrow j + 1$
  结束循环
  追加片段 $in[\pi[i]..\pi[j]]$
  $i \leftarrow j + 1$
结束循环
阶段 3：生成
生成 out = 拼接($Segments$)
返回成功

一旦算法1识别出一个有效置换，向量化器便构建映射 π，使得 $out[i] = in[\pi[i]]$。随后，该转换应用一种贪心分组策略，对 π 进行扫描，并将最长的连续递增子序列分组为提取片段。每个片段都作为一个`comb.extract`生成，完整结果则使用`comb.concat`组装。特殊情况将直接处理：

- 对于恒等置换，输出向量由输入向量替代。
- 对于完全逆序，输出由单个 comb.reverse 替代。
- 所有其他排列均以提取片段的拼接形式输出。

该遍本身不会尝试合并提取范围；此简化由 CIRCT 的降级和打印基础设施在后续阶段完成。算法 1 不会追踪穿过`comb.and` 或 `comb.mux`。这种追踪由第 3.4 节中所述的结构分析执行。示例 3.2 阐释了算法 1 的工作原理。

示例 3.2. 算法 1作用于 `comb` MLIR 方言，它是 CIRCT 的一部分。在我们的流程中，Verilog 首先被降级为`comb`。随后，生成的 `comb` 程序会被

向量化，之后再转换回高层 Verilog。图 6展示了这一过程。算法 1 的位来源分析在 comb 程序中检测到与赋值对应的置换输出 = {输入[0]，输入[3],输入[2]，输入[1]}。在贪心分组阶段，算法识别出这些位输入[3:1] 构成一个连续切片，可作为单个操作发射[1]。最后，向量化表示被转换回 Verilog。生成的代码符号复杂度更低，因为三个独立赋值被一个切片表达式取代。

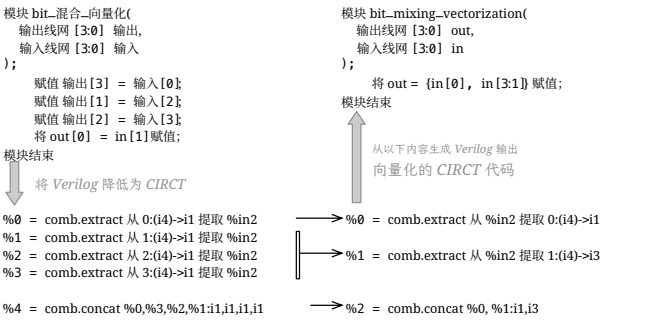

图 6. 位排列向量化示例。Verilog 中的各个位级赋值会被降级为 comb 操作，通过位来源分析进行分组，并重新生成基于片段的表达式 out = {in[0], in[3:1]}。

定理 3.3. 设 N 为目标向量的位宽，D 为数据流图的最大深度。算法 *1*（位排列向量化）在 $O(N \cdot D)$ 时间内识别并将位级赋值归组为字级拼接。

> **证明：** 该算法分为三个阶段。首先，对于输出向量中的每一位（共 N 位），递归追踪数据流以确定其来源。由于每次追踪的长度受图深度 D 限制，因此该阶段耗时为 $O(N \cdot D)$。其次，通过线性扫描（$O(N)$）识别构成有效置换的连续位序列。最后，输出生成的片段，最多需要 $O(N)$ 次操作。总复杂度由初始追踪主导，因此为 $O(N \cdot D)$。

3.4 结构分析

结构分析可识别逻辑锥。逻辑锥是影响单个标量输出位的一组上游运算。定义3.4对此概念进行了形式化。

定义 3.4 (逻辑锥)。 设电路表示为一个有向无环图，其节点为操作，边表示位级数据依赖。设 b 为一个标量输出位。 b的逻辑锥记为 $LC(b)$，递归定义如下：

基本情况： 如果 b 是主输入位或常量，则：

$$LC(b) = \{b\}.$$

递归情况： 如果 b 由某个操作生成 op其输入位为 i_1、i_2、……、i_n，则：

$$LC(b) = \{op\} \cup \bigcup_{k=1}^{n} LC(i_k).$$

定义 3.4 意味着，某一位的逻辑锥 b 包含生成 b 的操作，以及 b的后向切片；也就是说，包含所有影响 b 计算的位的逻辑锥。示例3.5 说明了这一概念。

示例 3.5。 图 7 展示了定义位 %out的逻辑锥。该示例说明了一个重要事实：逻辑锥是无环的。逻辑锥之所以无环，是因为它定义在电路的组合逻辑部分上。时序元件（例如寄存器）会打断反馈路径，因此充当逻辑锥的边界。在 comb 方言中，CIRCT 的操作保证是组合逻辑操作，因此中间表示本身已经确保不存在环。换句话说，comb 中的逻辑锥在构造上始终是有向无环图。

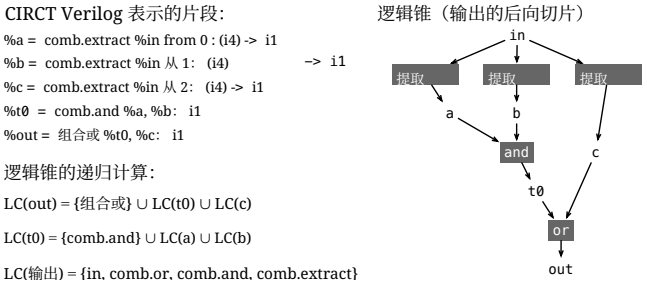

图 7。逻辑锥的计算。

3.4.1 逻辑锥的分组。 结构分析通过验证以下条件，确定一组位级操作是否可以安全地合并为单个向量操作：

1. 所有输出位的锥相互独立 （不存在跨位依赖关系）；
2. 这些锥同构 （结构相同且具有统一的位步长）。

当这两个条件成立时，向量化器会将 N重复的标量操作替换为单个向量操作。对于具有 N 位的向量类型输出值，分析会构建 N逻辑锥并执行以下检查：

- **安全性：** 各锥不得重叠；共享中间值意味着位与位之间存在依赖关系（例如进位链），这会阻碍向量化。

[1]comb.extract 指令会从给定位置开始提取一段比特；例如，comb.extract %in2 from 1 : (i4) -> i3 会从位置 1 开始提取 3 个比特，对应于切片 in[3:1]。

- **等价性：**各锥必须具有相同的运算符结构，并且在输入向量中索引推进一致（例如，$a[i]$、$b[i]$、$c[i]$）。

如果这些锥在结构上同构且在语义上相互独立，则会将它们合并，并降低为目标方言中可用的最简洁向量操作。

示例 3.6。 图3 中的每个输出位都使用相同的逻辑结构进行计算，该结构独立应用于两个输入总线中对应的位。对于位向量输出out，逻辑锥 $out[i]$ 是可能影响其值的完整上游运算集合。结构分析首先确保各个锥彼此独立，常量和不变值（如输入）除外：

$$\forall i \neq j,\ Cone(out[i]) \cap Cone(out[j]) = S \ \wedge\ S \neq \emptyset \Rightarrow \forall a \in S,\ a \text{ is invariant}$$

接下来，它会检查所有锥是否同构：

$$Cone(out[0]) \cong Cone(out[1]) \cong \cdots \cong Cone(out[N-1]),$$

这意味着它们包含相同的运算符结构和位宽变化。这项等价性检查会执行子图同构测试，以验证：(i) 每个锥中出现相同的运算符；(ii) 具有相同的结构连接和操作数顺序；(iii) 位提取遵循一致的索引步长（例如，$a[i]$、$b[i]$、控制信号）。在此情况下，所有锥都满足独立性和同构性，因此可以将 N复制的按位表达式替换为单个向量级操作（例如，条件选择）。

3.4.2 实现细节。算法2 概述了结构向量化的实现。该算法独立地处理构成 Verilog 设计的每个模块 $M\ P$，目标是向量类型的输出。对于宽度为 $v\ N$ 的向量输出，算法首先构造每个位 的反向锥 $v[i]$（此过程将在第 3.4.3 节中说明）。

算法 2 通过反向锥进行结构向量化

```
过程向量化(Design P)
对于所有 模块 M ∈ P 执行
    对于所有 输出 v 在 M 中，位宽为 N > 1 执行
        C ← [ ] {反向锥列表}
        对于 i ← 0 到 N − 1 执行
            C[i] ← 反向锥(v[i])
        结束循环
        如果独立性判定(C)与同构性判定(C) 则
            Perm ←置换提取(C)
            Expr ← 向量表达式构建(C[0], N)
            ReplaceWithVectorAssign(v, Expr, Perm)
        结束 if
    结束 for
结束 for
```

收集完所有锥后，算法2会检查两个安全条件（见第 8 行）。首先，各锥必须彼此独立，也就是说，同一操作不能出现在多个锥中。此条件可确保各个位不会共享中间值（例如进位链），否则将无法进行向量化。其次，各锥必须同构，即它们必须具有相同的运算符结构，并且其输入的位索引变化规律一致。直观而言，这保证了每个位都在输入向量的不同位置执行相同的计算。

如果两个条件都成立，算法 2 会提取索引置换，将每个输出位映射到其对应的输入位置（第 9 行）。此步骤可捕获恒等映射、反转和一般重排等情况。随后，根据一个代表性锥（最低有效位）重建向量表达式，并将标量操作提升为其向量等价操作（第 10 行）。最后，用一个保留已检测到的置换的向量赋值替换原有的标量赋值（第 11 行）。

定理 3.7. 给定一个包含 V 个操作和 E 个依赖关系的硬件模块，结构化向量化遍次（算法 *2* 和 *3*）能够在$O(N\cdot(V+E))$ 时间内正确识别同构逻辑锥。

> **证明：** 该算法构造 N 个独立的反向逻辑锥，每个输出信号位对应一个。每次反向遍历（算法 3）都会探索依赖图；在最坏情况下，它会访问每个顶点和边，耗时为 $O(V+E)$。对于 N个位，总构造时间为 $O(N\cdot(V+E))$。随后，同构检查（算法 2）会比较这N 个锥。由于只需验证已建立的锥之间运算符类型和线性索引步长是否相同，因此比较所需时间与图的大小成正比。

3.4.3 后向锥的构造。 算法3展示了 反向锥 的构造过程，该过程用于构造与标量值相关联的依赖子图。从根位开始，它通过递归访问每个值的定义操作及其操作数，对数据流图进行反向遍历。遍历会持续进行，直到到达主输入、常量或其他没有定义操作的值。最终结果是影响根位计算的操作集合。

该算法使用工作列表来确保每个值最多被访问一次，从而避免在存在共享子表达式时进行重复探索。生成的锥是模块数据流图的一个有向无环子图，如图 7 所示。该锥包含计算该位所需的所有组合依赖关系。随后，算法 2 使用此表示来比较不同位位置上的锥。通过验证以下条件来检查独立性：

算法 3反向锥构造

```
过程反向锥(root)
Worklist ← [root]
Cone ← ∅
Visited ← ∅
当Worklist ≠ ∅执行
    v ←弹出(Worklist)
    如果 v ∈ Visited 则
        继续
    结束如果
    将 v 添加到 Visited
    op ← 的定义操作为 v
    如果 op ≠ ∅，则
        将 op 添加到 Cone
        对于所有 操作数 u，对 op执行
            将 u 推入Worklist
        结束循环
    结束条件判断
结束循环
返回 Cone
```

锥之间不共享操作；结构同构则通过比较锥的形状及其输入的相对索引来确定。

定理 3.8。采用贪心窗口方法的部分向量化策略（算法 *4*）的最坏情况时间复杂度为 $\Theta(N^3)$，其中 N 是向量位宽。

> **证明：** 算法 4 在完整向量化失败时尝试寻找最大的可向量化子范围。它采用嵌套循环结构：外层循环遍历起始位$i \in [N-1, 0]$，内层循环搜索最大的后缀 j。这会产生 $O(N^2)$ 个候选子向量。对于每个候选项，它都会调用向量化谓词（位排列或结构），而这些谓词本身会随片段宽度线性增长$O(N)$。因此，总体最坏情况复杂度为 $O(N^2 \cdot N) = \Theta(N^3)$。

3.5 部分向量化

如果位级分析和结构分析无法对整个输出总线进行向量化，向量化器便会尝试尽力而为的方法。该策略不会要求整个向量都遵循某种已识别的模式，而是找出可独立向量化的连续子范围(块)。算法 4 概述了向量化流水线的这一最后部分。

该算法使用滑动窗口从最高有效位（MSB）扫描输出向量，直至最低有效位（LSB）。对于每个位位置 i，它会搜索满足位级或结构向量化标准的最大后缀 $out[i:j]$。如果找到这样的块，就立即对其进行向量化，并从位 $j-1$ 处继续扫描。

算法 4 部分向量化（贪心分段）

```
过程 PartialVectorize(OutputBus out, N)
i ← N − 1
当 i ≥ 0 时执行
    found ← 假
    对 j ← 0 到 i − 1 执行
        Chunk ← out[i : j]
        如果位检查(Chunk)或结构检查(Chunk)
        then
            应用向量化(Chunk)
            i ← j − 1
            found ← 真
            跳出
        结束 if
    结束 for
    如果不是 found 则
        发射标量(out[i])
        i ← i − 1
    结束 if
结束 while
```

如果找不到有效的块，就缩小窗口大小并继续搜索。

示例 3.9. 图 8 展示了部分向量化的一个实例。第 5-8 行中的完整四位排列与任何已知模式都不匹配。不过，部分向量化仍可能发现： $out[3:1] \leftarrow (in[3:1])$ 并且 $out[0]$适合进行结构向量化。在这种情况下，两位块分别进行向量化，最终结果则通过统一赋值重构。

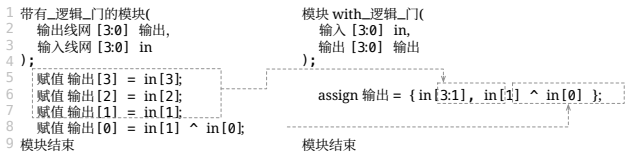

图 8.对两个位块进行部分向量化。

定理 3.10. 算法 *4* 中实现的贪心分段过程，其最坏情况时间复杂度为 $O(N^2 \cdot T_{\max}$ 检查)；当向量化谓词 T_{check} 的运行时间与向量宽度 N 成线性关系时，该复杂度可简化为 $\Theta(N^3)$。

> **证明：** 该算法采用嵌套循环结构来识别可向量化子范围。外层循环遍历起始位 $i \in [N-1, 0]$，内层循环评估潜在的结束位 $j < i$，以寻找可能的最大连续块。由此产生 $O(N^2)$ 个候选子向量。对于每个候选项，该算法都会调用向量化判定谓词（例如位排列或结构检查）。

鉴于这些谓词随所分析片段的宽度呈线性扩展（$O(N)$），最坏情况下的总复杂度为 $O(N^2 \cdot N) \Theta\ N^3$ 4 3,（）。如第 .节所述，由于发现过程具有贪心特性，实际性能通常接近 $O(N^2)$，因为一旦识别出极大块，算法便可跳过冗余检查。

4 评估

本章节评估以下研究问题：

- **RQ1：** 向量化对 ChiBench 基准测试的指令数量有何影响?
- **RQ2：** 自动向量化的绝对运行时间和渐近行为如何?
- **RQ3：** 向量化对 Jasper 的运行时间和内存消耗有何影响?
- **研究问题4：** 向量化对 Genus 的运行时间和内存消耗有何影响?
- **研究问题5：** 第 3 章节中介绍的各项技术对向量化模式数量的贡献是什么?
- **研究问题 6：** 内联阈值如何影响向量化机会?

下面介绍实验设置：

硬件： 所有实验均在搭载 AMD Ryzen 9 5900X 处理器和 32 GB 随机存取存储器的计算机上进行。该计算机配有两个 L1 缓存（各 384 KB）、一个 L2 缓存（6 MB）和一个 L3 缓存（64 MB），运行 Ubuntu Linux 22.04.5 LTS。

软件： 向量化基于 CIRCT Release 192（10 月 25 日）实现。第 4.3 章节使用 Jasper Formal Verification Platform 2023.12p002 版，第 4.4 章节使用 Genus 综合解决方案 21.19-s 版055_1。

基准测试： 实验使用 ChiBench 中的 Verilog设计 集合 [12]。ChiBench 包含从采用许可协议的开源代码仓库中挖掘出的 49,599 个设计。其中，1,157 个设计展现出了向量化机会。除非另有说明，本章节报告的数字均指这 1,157 个 Verilog 文件子集。

正确性： 我们使用 Jasper 顺序等价性检查（SEC）应用，验证了这 1,157 个向量化设计中每一个的语义等价性。

4.1 RQ1：指令数

我们以表示 Verilog设计 所需的 CIRCT 硬件指令数来衡量其规模。向量化会用一条向量化指令替换多条标量指令。因此，向量化有望缩小设计规模。本小节将探讨这一影响。

讨论： 图 9 比较了向量化前后的指令数。在呈现出向量化机会的 1,157 个设计中，904 个代码量减少，8 个指令数增加，245 个虽经过转换但保持不变。

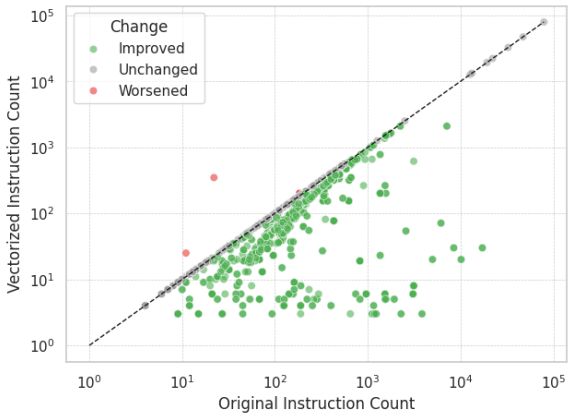

图 9。 比较向量化前后 CIRCT 指令数的散点图。对角线下方的点表示归约，上方的点表示增加，对角线上的点表示没有变化。

图 10 展示了 904 个得到改进的设计的指令数减少幅度分布。大多数减少幅度在 40% 至 50% 之间，但部分设计的指令数减少了近 100%。以下基准测试取得了最显著的四项改进：`Chi_11888`：指令数从 3,837 降至 3（99.92%）；`Chi_11336`：从 2,283 降至 3（99.87%）；`Chi_1203`：从 17,149 降至 30（99.82%）；以及 `Chi_11887`：从 3,069 降至 6（99.80%）。

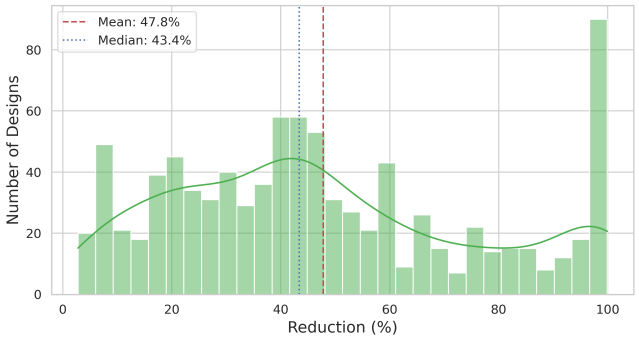

图 10. 改进设计中指令数量归约的直方图。虚线和点线分别表示平均值（47.8%）和中位数（43.4%）归约。

并非所有向量化都能缩小设计，因为维持语义等价性可能需要插入中间连线。我们分析了向量化导致代码大小增加的八种情况，并找到了一个反复出现的设计模式。所有代码扩张的实例都对应这一相同模式。这些模块实现了多路复用器，按照跨步（非连续）访问模式，从 64 位输入总线中选择 8 位切片。在原始 Verilog 中，这些访问通过直接位索引（例如 `data_i[7]`, `data_i[15]`）表示，而这

在硬件方言中会映射为成本低廉的连线引用。然而，对这些设计进行向量化时，向量化器必须通过插入多个“comb.extract”和“comb.concat”操作，显式构造输入向量，以汇集非连续的位。在这种情况下，准备向量操作数的开销超过了将多路复用器逻辑分组为向量操作所带来的收益，导致指令数增加。解决这一问题的一种方法是在向量化器中加入成本模型，以计入数据准备开销。例如，当操作数访问不连续且需要执行大量提取/拼接操作时，向量化器可以避免向量化。不过，由于这种模式很少见，我们决定不在当前实现中增加这项复杂性。

4.2 RQ2：编译时间

本章节研究向量化器的运行时间及其作为设计规模函数的渐近行为，以表明所提出的转换具有足够的实用性，可用于生产环境。

讨论： 图 11 通过在双对数坐标中绘制其运行时间（以秒为单位）与向量化 hw 指令数量之间的关系，展示了向量化遍的渐近行为。结果呈现线性趋势，其决定系数为 $R^2 = 0.982$，证实运行时间与设计规模成比例增长。与形式化验证或综合等计算量巨大的步骤相比，向量化过程所需时间微不足道。

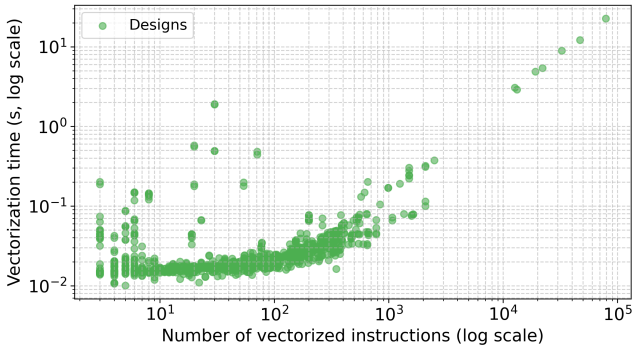

图 11。 向量化遍在双对数坐标中的渐近行为。数据点形成一条近似直线，斜率为 ≈ 1，表明其时间复杂度为线性。

4.3 RQ3：对 Jasper 的影响

Jasper 是一个形式化验证平台，提供一套用于 C/C++ 与 RTL 级分析的应用程序，使设计人员能够在开发周期的早期发现错误。在此框架中，RTL 展开组件负责将 RTL 描述转换为相应的网表表示，其中包括网络、门、以及结构连接。本章节评估向量化如何加速 Jasper 展开等流程。**讨论：** 图 12 展示了在 HDL 展开阶段，向量化对 Jasper 内存使用量的影响，涵盖这 1,157 个基准设计。原始设计的分布范围较广，且存在长尾离群值；而向量化版本则紧密聚集在低得多的内存占用中位数附近。

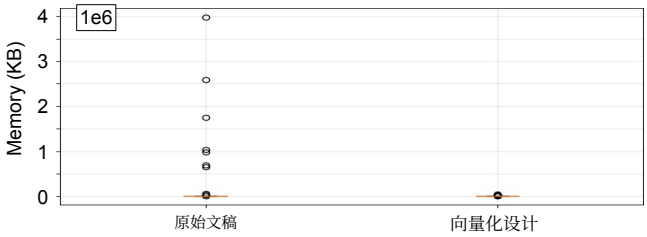

图 12。1,157 个原始设计与向量化设计的 Jasper 展开内存（KB）分布。

HDL 展开运行时间也呈现出类似的模式，如图 13 所示。这表明，向量化消除了原始设计中运行时间极长的异常值。平均而言，向量化使展开的内存消耗降低了 51.30%，运行时间缩短了 28.12%。

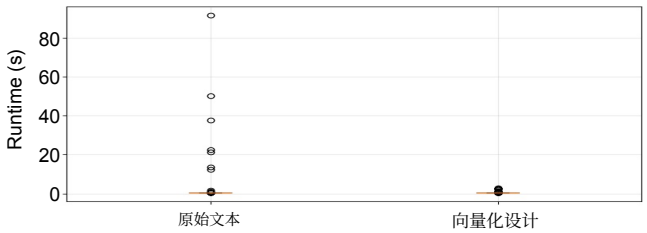

图 13。1,157 个原始设计与向量化设计的 Jasper HDL 展开运行时间（秒）分布。

4.4 RQ4：对 Genus 的影响

Cadence Genus 综合解决方案是一款具备物理感知能力的 RTL 到门级综合工具，可在确保时序收敛的同时，针对功耗、性能和面积对数字设计进行优化。它通过两阶段流水线处理 Verilog设计。第一阶段为展开，Genus 解析 RTL 代码，解析设计层次结构和各种构造，并将高级描述转换为门级网表。第二阶段为综合，执行物理感知优化、时序收敛和可测试性设计（DFT）插入，以生成针对功耗、性能和面积进行优化的门级网表。向量化可改善这两个阶段。由于向量化赋值以单个对象而非大量位级操作序列的形式表示，因此能够优化内存。减少赋值数量还可加快处理速度。例如，在优化

数据通路时，Genus 可以对整个向量执行操作，而不是处理单个位。本章节评估这些改进所带来的影响。

讨论：向量化后，我们观察到展开时间的几何平均值提升了 5.49%。虽然整体提升幅度不大，但个别基准测试显示出显著的加速效果。图 14 展示了 Genus 内存消耗、展开时间和综合时间方面提升最大的 20 项结果。在 ChiBench 的一个设计中（来自https://github.com/awai54st/LUTNet），内存消耗降低了 75%。在另一个设计中（来自 https://github.com/ehw-fit/evoapproxlib），综合时间缩短了 52%。展开时间的提升在各项基准测试中更为一致：表现最好的 20 项提升幅度为 45% 至 85%，几何平均值为 64.5%。

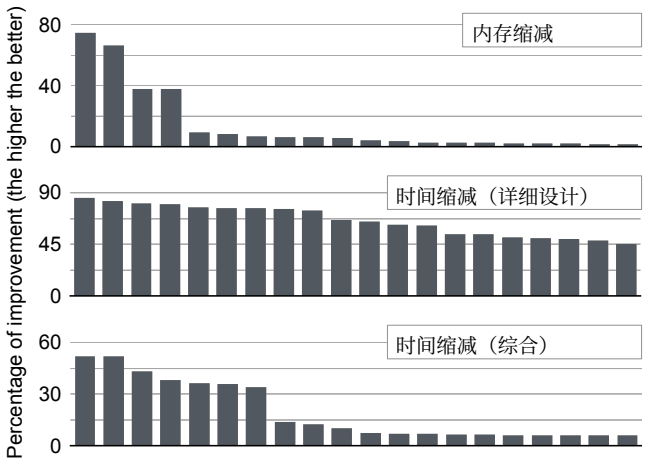

图14。Genus 流水线三个阶段中观测到的增幅最大的 20 项。每个条形代表一个不同的基准测试。不同图中的基准测试各不相同。

4.5 研究问题5：单项分析的影响

章节 3 中讨论的每项技术对向量化的整体效果贡献各异。本章节量化了它们各自的影响。

讨论。 图 15 根据促成向量化模式识别的技术，对其数量进行了细分。大多数成功案例都需要结合位级推理与结构推理，遵循章节 3.5 中的部分向量化策略。选择性内联同样有益：虽然仅适用于 289 个设计，但在这些设计中，它将向量化模式的数量从 999 个增加到 1,075 个。

纯结构向量化相对少见，在全部 1,157 个向量化设计中仅发现 55 种模式。仅通过位级数据流分析识别出的模式更为常见：我们观察到 208 个此类案例，其中 195 个对应于简单的线性赋值，输入和输出索引直接对齐。总体而言，图15 突显了

	内联	~内联	章节	示例
线性	195	195	III.C	赋值 o = i
置换	2	2		赋值 输出 = {in[0], in[3:1]}
反转	11	11		赋值 out = in[3:0]
结构	55	55	III.D	赋值 out = sel ? a : b;
部分	1,075	999	III.E	赋值 out = {in[3:1], in[1]^in[0]};

图 15。 章节 3 中各项分析对向量化模式数量的影响。内联适用于 1,157 个设计中的 289 个。

有效的向量化通常需要多种互补分析，因为混合模式策略占已识别模式的绝大多数。

4.6 研究问题 6：内联与向量化

章节 3.2 讨论了选择性内联：一种通过内联模块来创造更多向量化机会的方法。该章节提到，当前向量化器的实现采用了 150 条指令的内联阈值；因此，只有在生成的模块大小不超过 150 条 CIRCT 指令时，才允许整合代码。为了确定这一数值，我们使用不同的内联阈值（30、75、150、200、300 和 400）评估了向量化器。本章节将讨论这项实验。

讨论。 图16 展示了向量化模式的数量如何随内联阈值变化。此数量指的是单个向量化机会，而非文件数量（单个设计通常包含多个操作机会）。较大的内联阈值会带来更多向量化机会，但也会生成规模更大的设计。收益递减区间介于 150 至 200 条指令之间。将阈值从 150 条指令提高到 200 条时，我们观察到新增了八个向量化机会；而将阈值从 200 条提高到 300 条时，仅新增一个机会。

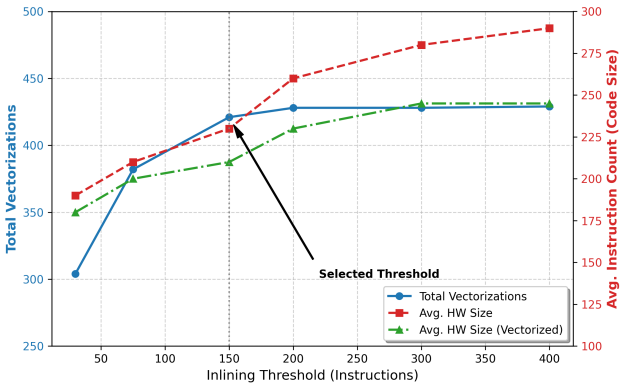

图 16. 章节中讨论的内联阈值3.2 对向量化机会数量的影响。

请注意，设计规模较大并不一定会降低应用于这些设计的 EDA 工具的性能。例如，虽然较低的阈值（如 30）会使代码绝对大小更小，但它对向量化的限制过多。因此，在同时适用于阈值 30 和 150 的 31 个可向量化设计上评估 Jasper 的性能时，我们发现使用后者的效果更好。当内联阈值为 150 条指令时，尽管内联会导致代码扩展，我们仍观察到平均内存消耗有所下降（9,224.4 KB 对比 9,260.2 KB），验证时间也有所改善（0.470 s 对比 0.474 s）。事实上，在我们的基准测试中，150 条指令的阈值为 Jasper 带来了整体最佳性能。

5 相关工作

正如我们在第1 节与2节中所讨论的，本文介绍的 Verilog（空间）向量化在概念上不同于 `clang` 和 `gcc`等编译器实现的（时间）向量化。此外，据我们所知，没有开源 RTL 工具链（包括 Yosys 和 CIRCT）会将向量化作为一种保持语义的 RTL 到 RTL 转换来执行。LegUp、Bambu 和 Xilinx Vivado 等 HLS编译器也不具备此类转换。相比之下，Jasper 在其最新版本（`2025.09FCS`）中加入了向量化功能，我们也对此进行了尝试。不过，该功能无法应用于顶层模块，目前也不支持一般位排列向量化。因此，它对本文所用的基准测试没有产生任何影响。尽管如此，我们的工作仍与多个研究领域相关，包括位向量的字级推理、从位级/网表表示中恢复字级结构，以及用于结构电路简化的代数重写技术。

字级推理与位向量求解器。在许多验证任务中，字（位向量）级推理比位展开高效得多，因此催生了一系列保留字级操作符的求解器与插值技术 [6, 13]。这些工作优化了 SMT 和等价性检查引擎所使用的内部表示，而我们的方法则会在这些引擎运行之前变换 Verilog 源代码，以展现字级结构。

从位级网络恢复字级数据通路。许多研究致力于解决逆问题：给定低层级布尔网络（例如查找表或AIG网表），重建高层级字、进位链和数据通路运算符 [9, 11]。这些“字恢复”和逆向工程技术依靠模式匹配和结构启发式方法，来检测特定的算术或多路复用器模式。虽然在概念上与我们向量化器的识别组件相关，但这些技术通常通用性较弱，在位展开之后才执行，也无法提供保持语义的RTL到RTL重写。

代数重写、*AIG/FRAIG*技术和结构压缩。已有大量研究利用基于AIG的表示进行代数重写和函数约简 [7, 14]。割点重写、FRAIG和结构哈希等技术通过优化布尔结构来缩小电路规模。这些方法与我们的工作互为补充：AIG重写在降级为位级形式后简化逻辑，而我们的向量化器则在编译流程早期恢复并编码字级行为，使验证和综合工具能够进行更具可扩展性的推理。

6 结论

本文介绍了一种用于对 Verilog 设计进行向量化的编译器变换。尽管向量化不会改变综合后的硬件，但它能降低底层规范的符号复杂度。实验结果表明，这种复杂度的降低有助于提升验证性能或综合优化等流程；使用 Jasper® 形式化验证平台和 Genus™ 综合解决方案进行测试，也观察到了这一效果。此外，由于该技术以源到源变换的方式运行，它还能提高 Verilog 设计的可读性和可维护性，从而带来超越工具性能的实际益处。鉴于这些优势，本文提出的技术正作为 CIRCT 框架的开源扩展发布。

数据可用性

本文讨论的向量化算法作为 CIRCT 标准发行版的一部分公开提供。为便于复现，该实现的独立版本可在以下地址公开获取：https://github.com/lac-dcc/manticore/，采用 GPL 3.0许可证。

致谢

本项目获得了 Cadence Design Systems 的资助。作者也感谢 FAPEMIG（Grant APQ-00440-23）、CNPq（Grant #444127/2024-0）和 CAPES（PrInt）的支持。我们感谢 CIRCT 社区将此实现纳入主代码库。我们特别感谢 Hideto Ueno 仔细审阅我们提交的所有补丁，并感谢 Fabian Schuiki 帮助我们将 comb.reverse 操作（实现位排列所必需）集成到 CIRCT的comb方言中。

参考文献

[1] Shreya Alladi、Alberto Ros 与 Alexandra Jimborean。Oml-vect：实现变换器的自动编译器驱动向量化，2026 年。[2] R. Allen 与 S. Johnson。为向量化、并行化和内联展开编译 C 语言。在 *PLDI*，PLDI '88，第 241–249 页，美国纽约州纽约市，1988 年。计算机协会。

[3] Péricles Alves、Fabian Gruber、Johannes Doerfert、Alexandros Lam- prineas、Tobias Grosser、Fabrice Rastello 与 Fernando Magno Quin- tão Pereira。运行时指针消歧。在 *OOPSLA*，OOPSLA 2015，第 589–606 页，美国纽约州纽约市，2015 年。计算机协会。[4] Breno Campos Ferreira Guimarães 与 Fernando Magno Quintão Pereira。为惰性求值而进行的惰性处理：将值调用自动转换为需求调用。在 *Compiler Construction*，CC 2023，第 239–249 页，美国纽约州纽约市，2023 年。计算机协会。[5] Ron Cytron、Jeanne Ferrante、Barry K. Rosen、Mark N. Wegman 与 F. Kenneth Zadeck。高效计算静态单赋值形式与控制依赖图。*ACM Transactions on Pro- gramming Languages and Systems*，13(4)：451–490，1991 年 10 月。[6] Alberto Griggio。用于软件验证的有效字级插值。在 *FMCAD*，第 28–36 页，得克萨斯州奥斯汀，2011 年。FMCAD Inc。[7] Daniela Kaufmann、Armin Biere 与 Manuel Kauers。结合 SAT 与计算机代数验证大型乘法器。在 *2019 For mal Methods in Computer Aided Design (FMCAD)*，第 28–36 页，2019 年。[8] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasi- lache 与 Oleksandr Zinenko。MLIR：面向领域特定计算扩展编译器基础设施。在*CGO*，第 2–14 页。IEEE Press，2021 年。

[9] Wenchao Li、Adria Gascon、Pramod Subramanyan、Wei Yang Tan、Ashish Tiwari、Sharad Malik、Natarajan Shankar 与 Sanjit A. Seshia。WordRev：在大量位级门中发现字级结构。载于 *HOST*，第 67–74 页，2013 年。[10] Saeed Maleki、Yaoqing Gao、Maria J. Garzarán、Tommy Wong 与 David A. Padua。向量化编译器的评估。载于 *PACT*，PACT '11，第 372–382 页，美国，2011 年。IEEE Computer Society。[11] Ram Venkat Narayanan、Aparajithan Nathamuni Venkatesan、Kishore Pula、Sundarakumar Muthukumaran 与 Ranga Vemuri。从查找表网表逆向工程字级模型，2023 年。[12] Rafael Sumitani、João Victor Amorim、Augusto Mafra、Mirlaine Crepalde 与 Fernando Magno Quintão Pereira。ChiBench：用于测试电子设计自动化工具的基准测试套件，2024 年。https: //arxiv.org/abs/2406.06550。[13] Wenxi Wang、Harald Søndergaard 与 Peter J. Stuckey。Wombit：一种使用字级传播的组合式位向量求解器。*J. Autom. Reason.*，63(3):723–762，2019 年 10 月。[14] Cunxi Yu、Maciej Ciesielski 与 Alan Mishchenko。基于与非图的快速代数重写。*Trans. Comp.-Aided Des.Integ. Cir. Sys.*，37(9):1907–1911，2018 年 9 月。[15] Peng Zhao 与 José Nelson Amaral。AbleGo：一个函数外提与部分内联框架：研究论文。*Softw. Pract. Exper.*，37(5):465–491，2007 年 4 月。