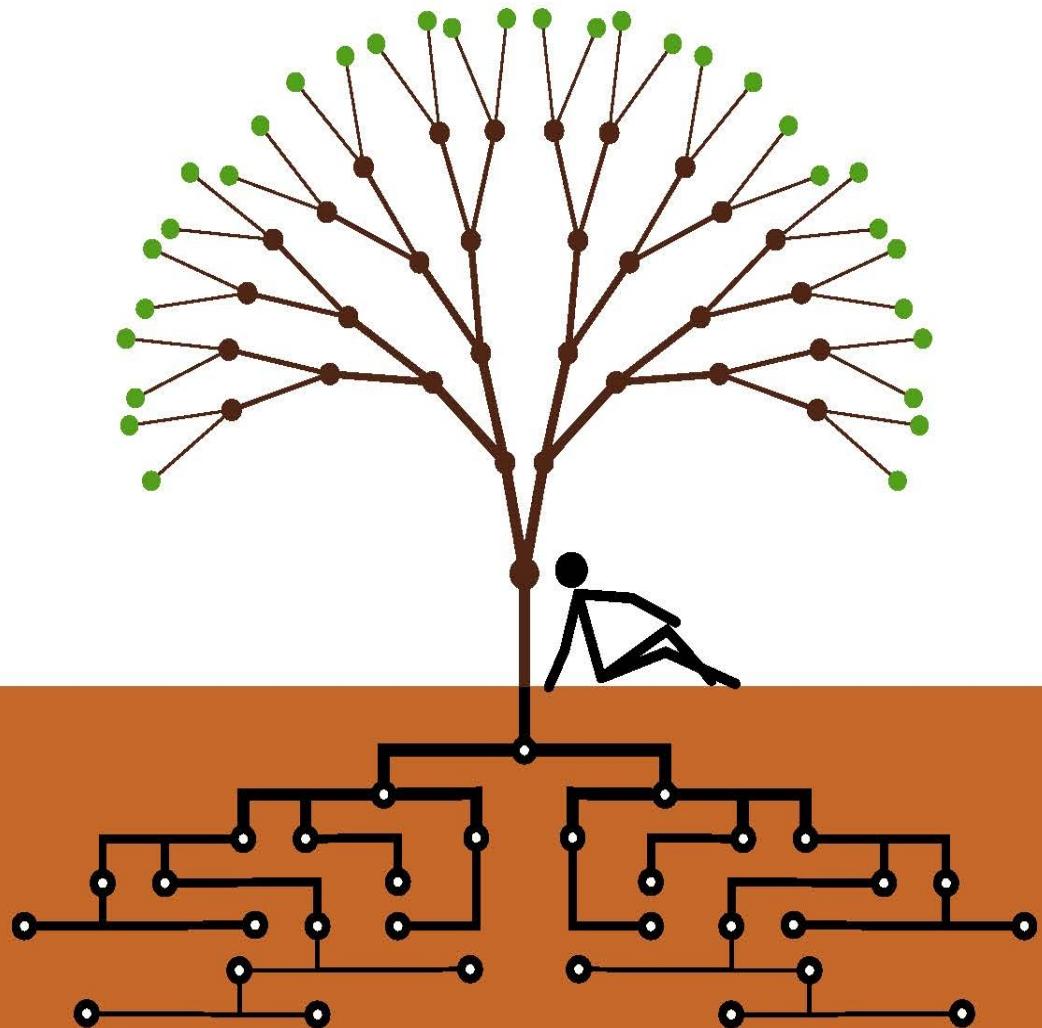


Parallel Programming for FPGAs

The HLS Book

FPGA并行编程
- 以HLS实现数字信号处理为例



Ryan Kastner
Janarbek Matai
Stephen Neuendorffer

译者：胡成龙、李彦晔、曹越
陈雯、吴彦北、胡博
王芝斌、杨勇、陆佳华
审阅：高亚军、孙广宇

目录

序

Introduction	1.1
译序	1.2

正文

第零章 前言	2.1
第一章 简介	2.2
第二章 FIR滤波器	2.3
第三章 CORDIC	2.4
第四章 离散傅立叶变换	2.5
第五章 快速傅里叶变换	2.6
第六章 稀疏矩阵向量乘法	2.7
第七章 矩阵乘法	2.8
第八章 前缀和与直方图	2.9
第九章 视频系统	2.10
第十章 排序算法	2.11
第十一章 霍夫曼排序	2.12

附录

词汇表汇总	3.1
参考文献	3.2

FPGA并行编程

-- 以HLS实现信号处理为例

国内鲜有介绍HLS的书，我们希望通过翻译 `Parallel Programming for FPGAs` 这本书，让更多的人来了解HLS和FPGA开发。

本书中文翻译和更新可在以下网址浏览

- [推荐] Gitbook Page: <https://xupsh.gitbook.io/pp4fpgas-cn/>
- [备用] Github Page: <https://xupsh.github.io/pp4fpgas-cn/>

本文电子版可以在以下网址下载

- pdf: <https://github.com/xupsh/pp4fpgas-cn/releases>

本文案例可以在以下网址下载

- HLS工程源代码: <https://github.com/xupsh/pp4fpgas-cn-hls>

问题反馈

- 如有问题请在这里指正
 - 提出ISSUE: <https://github.com/xupsh/pp4fpgas-cn/issues/new>
 - 在Gitbook Page:<https://xupsh.gitbook.io/pp4fpgas-cn>电子书阅读页面下方留言
- 此书翻译稿目前为初稿，欢迎各位提宝贵意见、反馈意见，授课支持，实验平台试用请联系 joshual@xilinx.com或xup_china@xilinx.com

LICENSE

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.

The newest version of this book can be found at <http://hlsbook.ucsd.edu>. The authors welcome your feedback and suggestions.

CONTRIBUTERS

统筹 胡成龙@sonnyhcl

章节	译者
README	胡成龙@sonnyhcl
00 Preface	李彦晔@Zaoldyeckk
01 Introduction	李彦晔@Zaoldyeckk
02 FIR Filters	曹越@onlypinocchio
03 CORDIC	曹越@onlypinocchio
04 Discrete Fourier Transform	陈雯@vickyiii
05 Fast Fourier Transform	陈雯@vickyiii
06 Sparse Matrix Vector Multiplication	吴彦北@wuyanbei24
07 Matrix Multiplication	吴彦北@wuyanbei24
08 Prefix Sum and Histogram	胡博@jianxin251
09 Video System	胡博@jianxin251
10 Sorting Algorithms	王芝斌@WangZhibin
11 Huffman Encoding	杨勇@rowen800
词汇表	李彦晔@Zaoldyeckk
参考文献	李彦晔@Zaoldyeckk

译序

利用FPGA进行算法加速和实现已经被广泛认知，但对于很多没有FPGA和HDL设计经验的开发者而言，往往又觉得开发门槛较高，因此全球相关的科研和工程人员都在致力于如何将FPGA技术介绍给更多的开发者，使更多人从FPGA的并行性，高性能，低功耗，灵活配置中获益。其中，Vivado HLS（高层次综合）就是一个成功的代表。通过Vivado HLS工具中，开发者可利用C/C++语言对FPGA进行编程，这项技术已经趋于成熟，在Xilinx客户的工程实践中也已广泛采用。

为此Xilinx研究院一直在考虑如何将HLS这项技术更好的介绍给学术圈，帮助原来熟悉FPGA开发的提高开发效率，帮助原来不了解FPGA开发的快速上手FPGA算法实现。此次Xilinx研究院的Steve首席工程师与UCSD的Ryan Kastner教授团队合作，推出了这本电子书，全书通过10个算法实现案例完整的介绍了通过HLS工具利用C/C++语言快速实现高性能FPGA实现的过程。看到这本电子书，我顿时觉得这就是我想要找的内容。这本书通过多年的工程、科研、教学经验的积累，深入浅出的将HLS实现方法，硬件设计的考虑以及系统优化都一一介绍。

因此，在此书刚刚推出后，我们在FPGA技术群里就发起了翻译志愿者的活动，当天就得到了本书的翻译者曹越，吴彦北，胡博，王芝斌，杨勇的支持，同时由Xilinx实习生胡成龙作为翻译项目经理，通过Github平台带领志愿者们开始协同翻译的工作，并将其电子版以博客的形式部署到GitHub Page和Gitbook上。Xilinx实习生李彦晔和陈雯也完成了部分翻译和审校工作，以及全部工程实现。大家都抱着一个心态，希望尽快将此书介绍给国内的读者。利用工作之余的时间，在短短两个月的时间内完成了本书的翻译，初次校对，以及全书工程实现的目标。本书的工程实现也有一定的特点，相关工程不仅仅实现了算法的并行编程，同时利用Pynq框架，在Pynq-Z2上通过Python实现了算法实现过程的可视化，这是相比于原始英文版加强之处。

相信通过阅读此书，可以让更多人了解、尝试并喜欢上FPGA开发，本书只是一个开始，我们也欢迎更多人的爱好者和我们一起来拓展HLS的应用场景，并将知识记录和传递下去。有任何疑问和建议，请与我联系 joshua.lu@xilinx.com。

陆佳华

2018年8月

译者风采

本次翻译我们用了技术社区内最geek的范儿，向技术社区内征求志愿者，通过在GitHub上协作写作，共同完成了本稿书籍的翻译工作。下面我们将以负责翻译的章节顺序介绍各位译者的风采。



胡成龙，复旦大学计算机科学技术学院在读研究生，Xilinx实习生



李彦晔，伊利诺伊大学香槟分校本科在读，Xilinx实习生



曹越，现就职于中国科学院电子学研究所，研究方向为实时信号处理。工作至今已完成多个合成孔径雷达
实时信号处理项目，在雷达成像、动目标检测和识别领域有丰富的工程经验。



陈雯，东南大学电气学院在读研究生，Xilinx实习生



吴彦北，武汉邮电科学研究院硕士毕业，目前主要工作方式向是医疗图像处理FPGA平台相关的开发。



胡博，本硕毕业于哈尔滨工程大学。上学期间多次参加国家挑战杯（国家三等奖）、"博创杯"全国嵌入式设计大赛（国家一等奖）、Robosub全球水下机器人大赛（全球第五）、微软imageinecup等比赛。目前主要从事视频编解码、图像预处理、目标识别与跟踪等相关工作。近几年技术方向积累多围绕SOC相关开发来构建自己的知识体系，主要是逻辑设计（Verilog和HLS）、驱动开发（Linux及裸核）、Linux系统上应用程序设计等。目前正在研究深度学习在前端智能中的应用。



王芝斌，江南大学计算机科学与技术硕士毕业，原*OpenHEC FPGA*云初创公司联合创始人、技术合伙人，现任科大讯飞研究院高级软件工程师，从事异构计算、云计算方面的工作。基于*HLS*的光流算法加速工作发表在*FPGA*、*FPT*、*FSP*和*ACM TACO*等国际会议或期刊上。



杨勇（网名：rowen800），2013年于华中师范大学获得无线电物理专业硕士学位，具有5年的*FPGA*设计和板级硬件开发经历，主要从事数字信号处理、无线通信以及*FPGA*应用技术的研究。活跃于各大电子技术论坛，分享了多篇博文，个人的技术博客深受广大网友的喜爱。

前言

"当有人说‘我想要一个编程语言，这个语言我只需要直接写我要干什么’的时候，你还是给他一个棒棒糖吧。" -Alan Perlis

本书将着重介绍高层次综合（HLS）算法的使用并以此完成一些比较具体、细分的FPGA应用。我们的目的是让读者认识到用HLS创造并优化硬件设计的好处。当然，FPGA的并行编程肯定是有别于在多核处理器、GPU上实行的并行编程，但是一些最关键的概念是相似的，例如，设计者必须充分理解内存层级和带宽、空间局部性与时间局部性、并行结构和计算与存储之间的取舍与平衡。

本书将更多的作为一个实际应用的向导，为那些对于研发FPGA系统有兴趣的读者提供帮助。对于大学教育来说，这本书将更适用于高阶的本科课程或研究生课程，同时也对应用系统设计师和嵌入式程序员有所帮助。我们不会对C/C++方面的知识做过多的阐述，而会以提供很多的代码的方式作为示范。另外，读者需要对基本的计算机架构有所熟悉，例如流水线（pipeline），加速，阿姆达尔定律（Amdahl's Law）。以寄存器传输级（RTL）为基础FPGA设计知识并不是必需的，但会对理解本书有所帮助。

本书囊括了很多对教学很有帮助的内容。每个章节均有一些小问题留给读者，这些问题将有助于加深对于材料的理解。在加州大学圣地亚哥分校（UCSD）的CSE 237C这门课里也有很多用HLS开发的项目，如果有出于教育目的的需要，我们可以对提出申请的读者分享这些课程项目的文件。这些HLS项目主要是与数字信号分析相关，重点于无线交流系统的开发。每个单独的项目都或多或少与书中的某一章节有所关联。这些项目以赛灵思大学计划（Xilinx University Program）使用的FPGA开发板为基础而开发，设计基础参考 <http://www.xilinx.com/support/university.html>。赛灵思也同时提供这些开发板的商业订单。同时我们鼓励读者在 <http://xilinx.com> 申请Vivado HLS的试用许可。

本书并不着重于HLS算法本身。HLS处理方面的具体内容已经有很多的资源供读者参考，包括计划，资源分配，捆绑[51, 29, 18, 26]的算法。本书更多的将会是引导学生掌握各类算法怎样分明的协同工作，提供具体的HLS语言开发程序的使用案例，因此，其他的一些更注重于算法与概念本身的材料会对理解本书很有帮助。本书也不着重于FPGA的细分结构和RTL设计方法，但是同样这方面的材料可以作为很好的辅助材料。

本书将主要使用赛灵思的Vivado HLS来完成类C代码到RTL的转换，C语言下的示例是针对Vivado HLS语法而完成的。本书不仅将介绍Vivado HLS的具体使用，而且会介绍那些最基本的HLS概念，这些概念应当适用于所有开发工具。我们同时鼓励读者尝试其他工具以真正理解这些概念，而不仅仅是在我们使用的工具里“学会”如何一步步操作。

希望你能享受这本书，并祝一切好运。

本书英文原版以下网址浏览

- hlsbook by UCSD: <http://kastner.ucsd.edu/hlsbook/>

本文电子版可以在以下网址下载

- pdf: <https://github.com/xupsh/pp4fpgas-cn/releases>

本文案例可以在以下网址下载

- HLS工程源代码: <https://github.com/xupsh/pp4fpgas-cn-hls>

问题反馈

- 如有问题请在这里指正
 - 提出ISSUE: <https://github.com/xupsh/pp4fpgas-cn/issues/new>
 - 在Gitbook Page:<https://xupsh.gitbook.io/pp4fpgas-cn/>电子书阅读页面下方留言
- 此书翻译稿目前为初稿，欢迎各位提宝贵意见、反馈意见，授课支持，实验平台试用请联系 joshual@xilinx.com或xup_china@xilinx.com

欢迎关注 **Xilinx**学术合作 以及 **PYNQ**中文社区



第一章 介绍

1.1 高层次综合(HLS)

硬件设计与处理近几年来发展迅速。过去我们的电路相对简单，硬件设计师们可以很方便的画出每一个晶体管，规划他们的连接方式，甚至他们的板上位置。可以说所有工作都是人工完成的。但随着越来越多晶体管的设计需要，硬件工程师也越来越需要依赖自动化设计工具来帮助他们完成设计，而这些设计工具也相对应的变得越来越精密。工程师在这些设计工具的协助下也更具效率。他们不再具体操作每一个晶体管，而只需要设计数字电路，电子设计自动化工具（EDA）把这些抽象而概括的电路自动转换成实际的部件构造版图。

米德和康威（Mead&Conway）的方法[50]，也就是使用一种硬件语言描述语言（Verilog, VHDL），并把其编译成片上设计的方法在上世纪80年代开始广为使用。但这之后硬件的复杂度还在以指数函数的增长速度发展，硬件工程师们只好寻求更加概括而高层的编程语言，RTL应运而生。在RTL里，设计师不需要考虑怎么构造一个寄存器或怎样安置这些寄存器，而只需要考虑这些寄存器在设计中起到怎样的作用。EDA工具可以先把RTL转化成数电模型，再由模型转换成一个设备上的具体电路实施方案。所谓“方案”其实就是在编译出的文件，这些文件可以用于规定某个自定义设备，也可以用于编程一些现有的设备，比如FPGA。如我们现在所见，新的设计方法确实帮助工程师们的设计思路变得更加清晰。更多关于这方面的探讨参考注释[42]。

HLS则是在这基础上更高层的一种方法，设计师们在HLS下需要更多的考虑大的架构而非某个单独部件或逐周期运行。设计师在HLS下需要注重的是系统的运行模式，HLS工具会负责产生具体的RTL微结构。最早大多数HLS工具是基于Verilog的，用户需要使用Verilog语言进行描述，工具也通过Verilog产生RTL。现如今很多HLS工具开始使用C/C++作为设计师端的语言。当然，选择HLS工具最重要的还是看它能否综合我们需要的程序，而不是它使用什么语言。

总体来说，HLS可以自动完成以下曾经需要手动完成的工作：

- HLS自动分析并利用一个算法中潜在的并发性
- HLS自动在需要的路径上插入寄存器，并自动选择最理想的时钟
- HLS自动产生控制数据在一个路径上出入方向的逻辑
- HLS自动完成设计的部分与系统中其他部分的接口
- HLS自动映射数据到储存单位以平衡资源使用与带宽
- HLS自动将程序中计算的部分对应到逻辑单位，在实现等效计算的前提下自动选取最有效的实施方式

HLS的目标是根据用户提供的输入和限制自动替用户做出很多决定。每个HLS工具在实际施行的效率上相差甚远，这其中我们有一些非常不错的选择，例如赛灵思Vivado HLS, LegUp, Mentor Catapult HLS。他们出众的特性在于可以支持更多更广泛的程序转换。我们在本书中将使用Vivado HLS作为演示软件，但是设计的思路与技巧在各个软件中应当是通用的，读者只需要在各自的软件中对语法进行稍微的调整。

大多数HLS工具需要用户提供功能的规范，交互的描述，一个对接的计算设备，和目标优化方向。而对于Vivado HLS来说，用户需要：

- 一个用C/C++/System C编写的函数
- 一个测试平台用于验证结果
- 一个FPGA开发版
- 期望的时钟周期
- 一个简单的实施指导

HLS工具没有强大到可以处理任何代码。很多我们平时在软件编程中常用的概念在硬件实施中很难实现，所以硬件描述语言对于具体实施会更加灵活。通常这些HLS工具需要用户提供一些附加信息（通过suggestion或#pragma）来帮助完善程序，因此我们说HLS工具会同时“限制”又“加强”了一门语言。举例而言，HLS工具一般无法处理动态内存分配，大部分工具对标准库的支持也非常有限。用户也应当避免系统调用和递归以尽量降低复杂程度。除去这些设计限制之外，HLS工具的处理范围非常的广（包括直接内存访问，流，片上内存），优化效率也很高。

根据Vivado HLS的使用指南，我们将对我们的输入程序作出以下规范：

- 不使用动态内存分配（不使用malloc(),free(),new和delete()）
- 减少使用指针对指针的操作
- 不使用系统调用（例如abort(),exit(),printf()），我们可以在其他代码例如测试平台上使用这些指令，但是综合的时候这些指令会被无视（或直接删掉）
- 减少使用其他标准库里的内容（支持math.h里常用的内容，但还是有一些不兼容）
- 减少使用C++中的函数指针和虚拟函数
- 不使用递归方程
- 精准的表达我们的交互接口

当RTL级的设计可用时，大多数HLS工具会进行标准RTL设计流。而在赛灵思Xilinx Vivado设计套装里进行的是逻辑综合，将RTL级设计转换成一个FPGA逻辑部件的连线表，这份连线表不仅包含需要的逻辑部件还包含他们的连接方式。Vivado之后将连线表和目标设备中的可用资源相关联，这个过程被称作布局及布线（PAR）。产出的FPGA配置被附在比特流（bitstream）上，用户可以将比特流上传到FPGA以实现想要的功能。比特流实质上是用二进制表示FPGA上每一个可用资源的配置，包括逻辑部件的使用，连线的方式，和片上的内存。大型FPGA例如赛灵思UltraScale FPGA拥有超过十亿个可配置比特，较小的FPGA上也至少有几亿个可配置比特。

1.2 FPGA构造

了解HLS的第一步是熟悉FPGA的构造，因为很多HLS的优化都是和这些构造特点息息相关的。过去几十年来，FPGA变得越发大而复杂，也加入了片上内存、自定义数据路径，高速I/O，和多核处理器等等精密结构。在这一节，我们只讨论FPGA中与HLS相关的结构特点，其他无关内容不会被详细描述。了解FPGA的现代结构后再学习HLS会有助于读者对于其理解。

FPGA由一个可编程逻辑模块的矩阵和与之相连的内存组成，通常这些模块是以查找表（LUT）的形式存在，也就是说把地址信号输入进去，对应内存位置的内容会直接被输出出来。一个N位查找表可以以一个N位输入真值表的方式来表示。

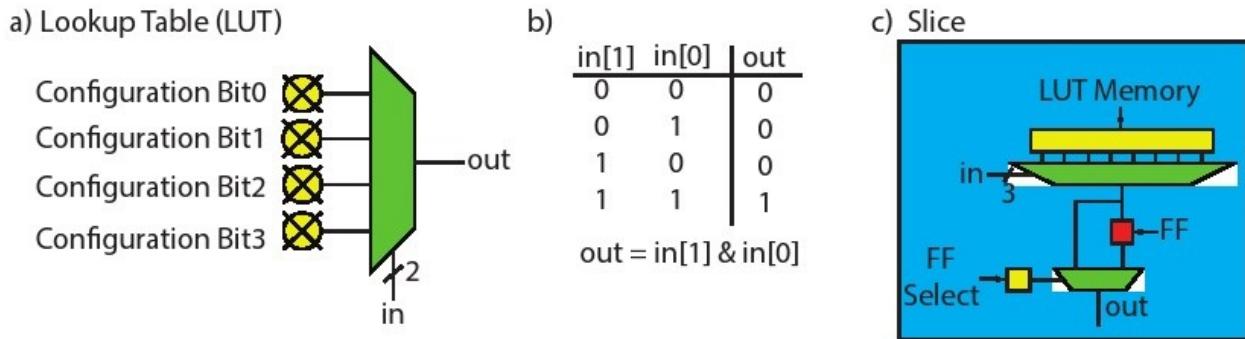


图1.1：a部分是一个两位输入查找表，写作2-LUT。每一个*configuration bit*，即备置比特，可以根据查找表的功能变化而变化，这样的特性让它具有了可编写的特性。b部分是AND门的编写方式，out一列的数值从上到下依次对应了配置比特0-3的数值。c部分是一个由查找表和触发器组成的简单的*slice*。这个查找表拥有九个配置比特，其中八个决定查找表的功能，剩下一个决定直接使用查找表的输出或使用触发器中储存的输出。*slice*的性质见下文。

上图中的a部分是一个2位输入查找表，共有 2^2 个配置比特。使用者通过编写程序来控制这些比特以实现某种功能。b部分是一个2位输入AND门的真值表，通过对应4个可能的结果产出（out一列），我们可以把a中的2位查找表编写成b中的AND门，即四个查找表输入依次对应b中的00, 01, 10, 11。按照这个模式编写查找表，我们可以轻松的改变它的功能，让它充当我们需要的部件。对于小的布尔逻辑（Boolean），这样的编写方式更加的灵活高效。实际中的FPGA大多使用4-6位输入的查找表作为运算基础，一些大型FPGA内甚至有几百万个这一级别的查找表。

怎样将图1.1中的查找表编写成一个XOR门呢？一个OR门？我们需要一个几位输入的查找表？

一个2位输入的查找表最多可以被编写出多少种形态？一个n位输入的查找表呢？

触发器（FF）是FPGA最基本的内存单位，通常触发器是配有查找表的，这样是为方便查找表之间的复制与组合。在这基础上再加入一个规定它们的函数（例如全加器），就可以创建一个更为复杂的逻辑单位，称为可配置逻辑块（CLB）或逻辑矩阵块（LAB）。有些设计工具中还会把它称作片（Slice）。为避免歧义，我们将在下文中用slice作描述，这样读者可以对在Vivado设计工具中出现的Slice更加熟悉。一个slice是由一个三位输入查找表和一个触发器组成的slice。slice可以变得更加复杂一点，比如常见的全加器。FPGA内部通常有一些定义好的全加器slice，这看起来有点违背FPGA的“可编写性”。但实际上使用全加器在硬件设计中太过于常见，把所有的全加器每次重新编写成一个slice会降低效率。灵活性和高效综合考虑，一些被配置好的slice是一个对整个系统有益的设计。

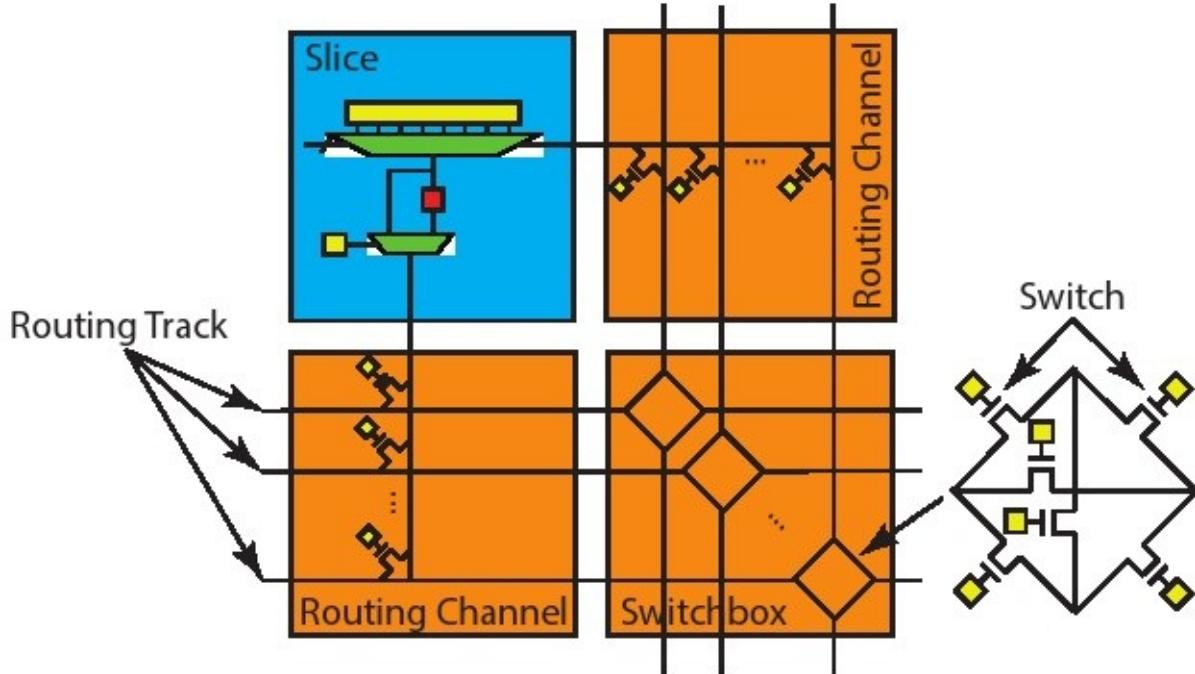


图1.2:由查找表和触发器组成的slice，通常slice比图上结构更加复杂一点，slice之间通过连线通道（routing channel）和开关盒（switchbox）相连，这两个用于连接的设备提供了一个同样可编写的互联和自定义的数据传输方向。开关盒是一个包含很多开关（传输晶体管制成）的部分，提供了编写传输路径的能力

可编写的互联是FPGA最关键的特性之一，它能提供一个slice之间更灵活的连线网络。slice的输入与输出全都与连线通道相连，连线通道也是通过配置比特来决定每个slice的输入输出通向哪里，而通道本身则与开关盒相连。开关盒由很多传输晶体管充当的开关所组成，它的工作便是连接通道与通道。

图1.2展示了一个slice，连线通道和开关盒之间的连接方式。slice的每个输入输出都应与通道中的一条路线相连。所谓路线，我们可以简单的把它想成一跟比特层级的跳线，在物理层级上这条线路是由传输晶体管构成的，同样具有可编写性。

开关盒像是一个连接矩阵，沟通不同连接通道中的各个路线。FPGA一般有一个2D的形式，能给使用者一个大概的2D计算模型，我们称之为岛状结构。在岛状结构里，每个slice都是一个逻辑岛，岛与岛之间通过连线通道和开关盒相连。在这里每个开关盒在上下左右四个方向连接了四个连线通道。

连线通道和开关盒中的所有开关都通过使用者的编写控制着逻辑部件之间的联系。现如今业界对于电路层级的FPGA架构已经了解的很深了，连线通道的数量，开关盒的连接方式，slice的结构等等都有很详尽的资料。我们在注释中附上了一些书籍[12, 10, 30]，有兴趣的读者可以参考一下。当然使用HLS工具不需要了解那么多细节的资料，这方面更多的知识是作为理解HLS优化工作的辅助。

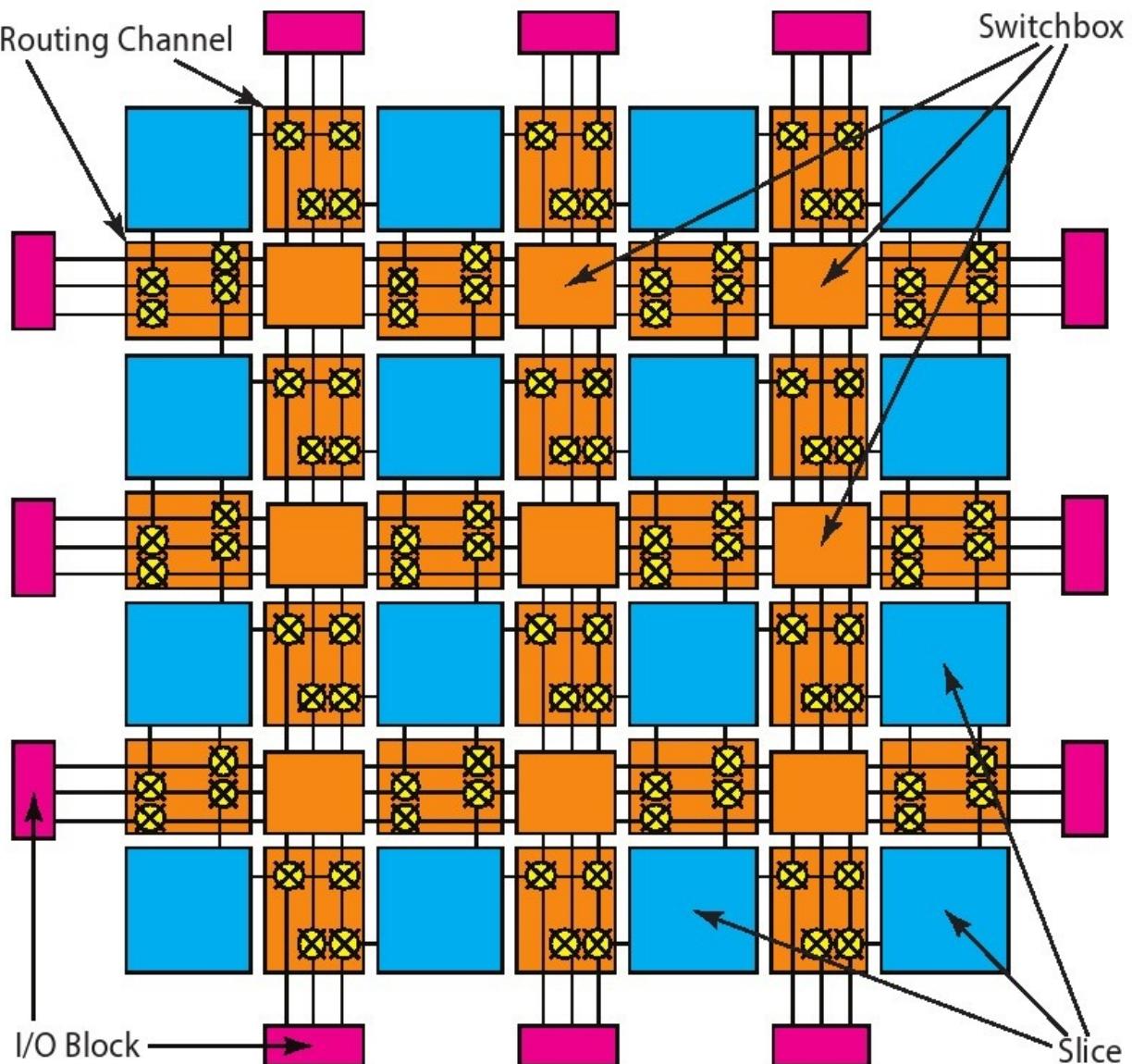


图1.3:2D的FPGA岛状结构。每个slice内的逻辑与内存通过连接通道和开关盒相连。IO模块有一个对外接口，可以通往内存，处理器，传感器等等。一些FPGA上IO直接与芯片引脚相连，一些FPGA则用起连接逻辑架构和片上的一些资源。

图1.3提供的是一个更概括性的结构互联，可以比较清楚的看到各部分之间的物理连接方式。FPGA的逻辑部分通过一些IO模块与外部设备相联系，像微控制器（通过AXI接口连接片上ARM处理器），传感器（通过A/D接口连接天线），作动器（通过D/A接口连接电机）都是可以实现的。近来发展的FPGA又集成了自定义片上I/O处理器，像内存控制，无线收发，模拟与数字转换器这类的装置。

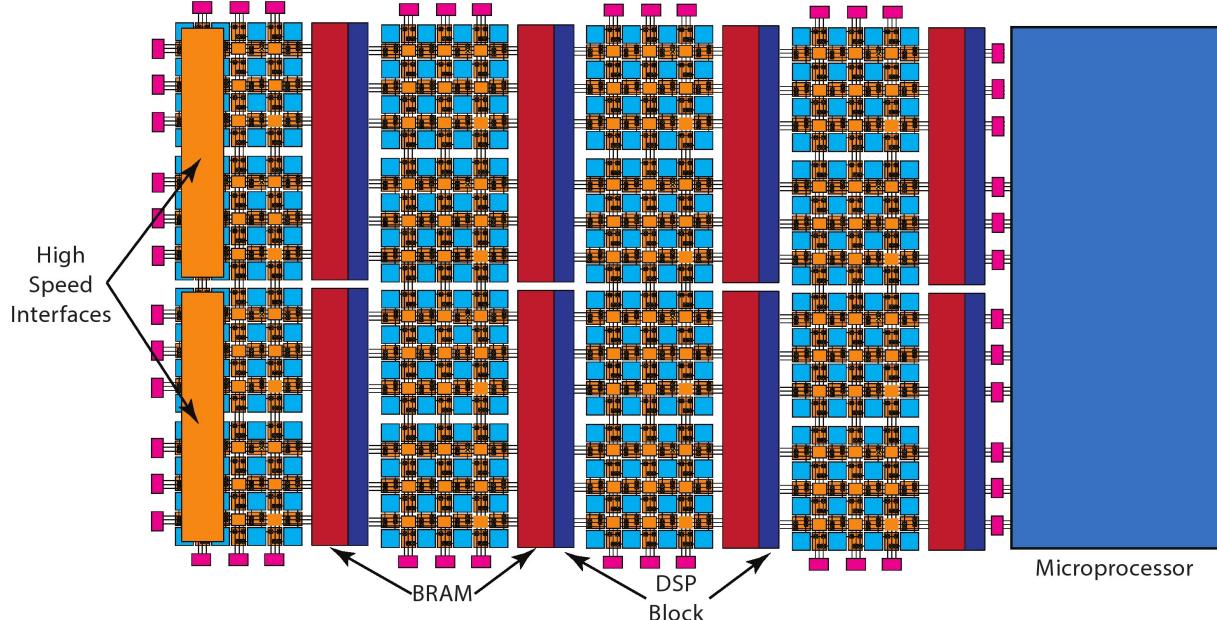


图1.4:现代FPGA变得更加异构化，一些FPGA除了可编程部分之外，加入了很多预配好的结构比如寄存器堆，自定义数据路径，高速互联等等。现如今FPGA通常配有一个或多个处理器，比如ARM或x86核，两者协同控制系统。

我们说到FPGA上要承载的晶体管变得越来越多，这也是FPGA上多了很多预配好的资源的原因。这部分硬件用于完成特定工作。很多设计都需要大量的加法和乘法，因此FPGA厂商把这部分的内容预配好以直接使用。像DSP48数据路径已经被用一种高效的方法预配好，添加了乘法、加法、乘积、逻辑操作等一系列算数。对于DSP48这样的模块来说，它们依旧保留了一定的可编写性，但不像其他可编程逻辑那样完全灵活。这样综合而言，用户在DSP48这样的模块上进行乘法这样的操作会比重新编写高效的多。所以我们说灵活性和效率有时候是此消彼长的。现代FPGA会含有成百上千个DSP48模块，如图1.4所示。

比较自搭乘积和DSP48基础上乘积的性能，两种情况下可获得的最高频率分别是多少？FPGA资源利用上有什么变化吗？

块RAM (BRAM) 是另一个预配好的模块。BRAM是一个支持多种内存形式和接口的可配置随机储存器，可以储存字节，对字，全字，双字等等。BRAM还可以把这些数据传给本地片上总线（与可编程逻辑交流）或处理器总线（与片上处理器交流）等等接口。总体来说它有两个功能，一是芯片上各部分的数据转移，二是储存大一些的数据集。slice经过编写也可以储存数据（通过触发器），但这样做会增加额外消耗。

属性	外部内存	BRAM	触发器
数量	1-4	几千	几百万
单个大小	GB级	KB级	bit级
总量	GB级	MB级	100KB级
宽度	8-64	1-16	1
总带宽	GB每秒	TB每秒	100TB每秒

表1.5:三种形式内存存储比较。外部内存存储密度最高但带宽有限，触发器拥有最好的带宽但储存容量太小，BRAM则像是两者之间的中间值。

一块BRAM通常有大约32000比特的储存容量，可以以 32000×1 比特， 16000×2 比特， 8000×4 比特等等形式存在。串联在一起可以拥有更大的容量，Vivado工具可以完成这方面的配置，而Vivado HLS的优势也在于这里，设计者不再需要考虑这一层级的细节。通常BRAM和DSP48放置在一起，对于HLS设计来说，我们可以直接把BRAM想成一个寄存器堆，它可以直接输出到一个自定义的数据路径（DSP48），可以与处理器交流，也可以像可编程逻辑上的数据路径传输数据。

思考怎样把一个很大的数组存在BRAM和可编程逻辑里。它的性能如何变化？资源使用呢？

表1.5是一个不同内存形式比较的表格。如表格所示，所有触发器最后可以形成一个几百KB的储存，它们每个周期都可以被读写所以总带宽非常的大，但很显然他们的储存容量不尽如人意。BRAM在不牺牲很大带宽的前提下，提供了更大的储存密度。带宽的牺牲主要在于每个周期BRAM只有1-2个入口可以被接通。外部内存对于带宽的牺牲更大，但提供了最大的容量。把应用数据放在哪里是非常关键的一个设计决定，我们会在整本书里经常提到。Vivado HLS工具也允许设计者清楚指明到底要将这段数据放在哪里。

片上晶管的繁多也丰富了我们的预配资源，片上的处理器其实就是一个很好的代表。现如今的高端FPGA会含有4个甚至更多的微处理器（比如ARM核心），小型的FPGA上附有一个处理器也变得很常见。处理器使芯片有了运行操作系统（比如Linux）的能力，它可以通过驱动和外部设备交流，可以运行更大的软件包比如OpenCV，可以运行更高级的语言（比如python）并以更快的速度运行。处理器经常成为了整个系统的控制者，协调了各方之间的数据转移，也协调了各个IP核心（包括用HLS自定义的IP核和第三方IP核）和板上资源的关系。

1.3 FPGA设计与处理

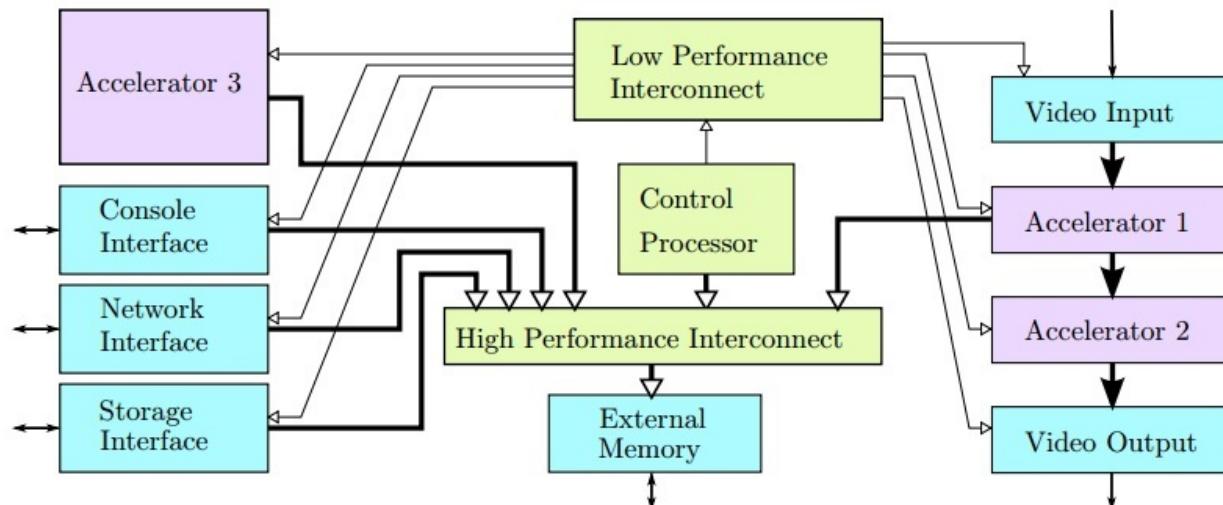


图1.6:设想中的嵌入式FPGA设计结构图，包括接口核心（蓝色框），标准核（绿色框），和应用加速核（紫色框）。注意应用加速核也可能自带流接口，内存对应接口。

由于FPGA大小和复杂度的不断提升，设计师更倾向于从高层建造自己的设计。这样一来，FPGA设计更多是由一个个大组件，或IP核组建而成，如图1.6。在整个设计的外围临近I/O引脚的地方通常是一些少量的逻辑，它们一般用来完成关键时序和协议，比如内存控制模块，视频接口核心或模拟数字转换器。这部分逻辑我们称之为I/O接口核，通常以RTL的形式构架并需要加上其他的时序限制。时序限制的目的是阐明信号本身与信号变化规则之间的时序关系。设置限制的时候必须要考虑信号蔓延到电路板和连接装置的影

响。使用I/O引脚附近逻辑的考虑是为了高速接口的实施需要，这些逻辑更适合在"高速"前提下实现数据的序列化和反序列化，时钟的恢复与分布，和精准延迟某些信号以不断从寄存器获取数据。I/O接口核心在不同的FPGA架构上差别比较大，FPGA供应商一般会有设计参考或成品部件，因此我们不会展开太多细节。

除了I/O引脚，FPGA一般会有标准核，处理器核心，片上内存核连接开关都属于标准核。标准核另外还包括原生的函数处理部件比如滤波器，FFT，编解码器等等。这些核心的参数和接入方式在不同的设计中相差很大，但它们并不是在设计中真正造成差异的部件，相反他们是相对"水平的"技术部分，可以被插入到各类不同的应用领域。FPGA厂商同样也提供这些模块，但设计师其实很少情况下接触到它们。不像IO接口核心，标准核心主要是同步电路，它除了时钟时序限制之外不大有限制。这些特点让标准核更容易在不同FPGA中兼容，当然，被转移到另一种FPGA结构中时还是需要一定优化的。

最后一种核心是针对应用的加速器核，同标准核一样，加速核通常是由时钟限制而规定的同步电路，但这些核却是系统设计师们在具体应用中不可避免要接触的部分。如果把一个设计的系统比做一道菜，那加速器核就像是秘制配方，它是让每个人的菜肴各有风味的关键。最理想的情况是设计师又快又轻松地设计出了这样的高性能核然后把它们以很快的速度集成到整个系统里，这也是我们这本书的主要目标，用HLS设计出快而高效的核。

图1.6中的系统通过两种方法可以实现。第一种方法是把HLS产生的加速器核当作一个普通的核。用HLS创造出这种核之后把他们与IO接口核和标准核组合到一起（可以通过Vivado IP Integrator这样的软件），这样我们就得到了完整的设计。这个方法叫做以核为基础的设计方法，与使用HLS之前的FPGA设计方法十分相似。第二种方法则着重于设计样板或平台，称为平台为基础的设计方法，这种方法下设计师先用IO接口核和标准核组合出一个样板，然后再用HLS通过壳（shell）的接口将各式算法或对象组合进去。只要壳支持双边的接口，加速器核在平台与平台之间的移动也非常容易。

1.4 设计优化

1.4.1 性能特点

在开始讨论怎么去优化之前，我们先要讨论一下判断一个设计特点的标准。计算时间就是一个衡量设计好坏的重要标准。很多人把时钟周期数作为一个同步电路性能的指标，但实际上对于两个使用不同时钟的电路这是不得当的，而时钟不同又是HLS下的绝大多数情况。比如说，我们现在已经规定好了Vivado HLS的输入时钟限制，那么工具根据时钟的不同会从同一段代码中产生不同的结构，所以这不是一个很恰当的比较方式。秒数是一个更好的对应比较指标。Vivado HLS工具会提供一个周期数和周期频率的报告，用户可以用此得出某段代码的操作时间。

改变时钟频率有时候可以优化设计。Vivado HLS工具把时钟频率作为一个输入，所以改变一个输入可以导致产出的结构完全不同。我们会在后文继续讨论。书中章节2.4描述了根据时钟周期决定限制。书中章节2.5讨论了改变时钟周期如何通过操作链提升生产力。

我们用任务（task）这个术语来表示一个行为的基本单位，用户可以在Vivado HLS中发现与之对应的是调用函数。任务延迟就是任务开始到任务完成中间的这段时间。任务间隔则是任务开始到下一个任务开始之间的这段时间。所有的任务输入，输出和计算的时间都被算在任务延迟里，但是任务的开始并不等同于读取输入，同样任务的结束也不等同于写出输出。在很多设计中，数据率是一个很重要的东西，它同时取决于任务间隔和函数参数的多少。

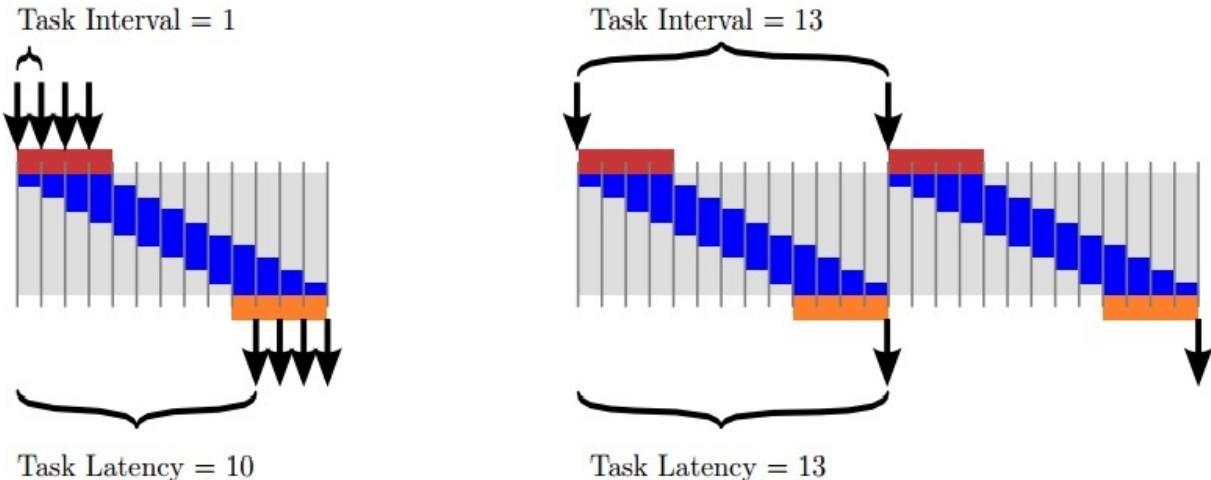


图1.7:两种不同设计的任务间隔和任务延迟，上方的弧线指示的是任务间隔，下方的弧线指示的是任务延迟。左右两个设计的区别在于，左边是流水线（pipeline），右边使用了更顺序化的设计。

图1.7表示的是两种设计的实施设想，横向轴是时间轴（从左到右增大），纵向是设计中不同的函数单位。红色表示的是输入有关的操作，橙色表示的是输出有关的操作，正在活跃的运算符用深蓝表示，不活跃的则用浅蓝表示。每一个进入的箭头表示的是一个任务的开始，而出去的箭头表示任务的完成。左侧的图表示的是一个每个周期都执行新任务的结构设计。与之对应的是完全流水（fully-pipelined）结构。右侧表示的则是一个完全不一样的结构，系统每次读取四段输入，处理数据，然后再合成一个4段数据的输出。这种结构的任务延迟和任务间隔是一样的（13个周期），并且每一周期内只有一个任务在执行。这个结构和左边的流水形成了鲜明对比，左边的结构在同一周期内显然有多个任务在执行。HLS中的流水和处理器中的流水概念相似，但是不再使用处理器中操作分5个阶段并把结果写入寄存器堆的方法，Vivado HLS工具构造的是一个只适用于特定板子，可以完成特定程序的电路，所以它能更好的调整流水的阶段数量，初始间隔（连续两组数据提供给流水之间的间隔），函数单位的数量和种类，还有所有部件之间的互联。

Vivado HLS工具通过计算一个任务输出到输入之间这个过程需要的寄存器数来决定周期。因此，0周期的任务延迟是可以实现的，也就是组合逻辑下路径上没有任何寄存器。另一个常用的工作是计算输入输出并把结果存到寄存器里，通过这些数据找到路径上的寄存器数。这样的计算有花费很多的周期。

很多工具把任务间隔称为生产力（throughput）。这个词语听起来和间隔没什么关系。一个任务间隔的变长不可避免的会减少一段固定时间内能完成的任务数，也就是“生产的力度”。还有一些工具用延迟来描述读输入和写输出的关系。非常不幸的是，在一些复杂的设计中，任务的特点很难仅仅用输入输出来分析，比如有时候一个任务需要读很多次数据。

1.4.2 面积和产力的取舍

为了更深入的讨论使用HLS工具过程中的问题，我们需要分析一个简单但很常见的硬件函数----有限脉冲响应（FIR）滤波器。FIR会对输入做固定系数下的卷积，它可以被用作充当各式滤波器（高通，低通，带通），最简单的FIR可能就是一个移动平均滤波器。有关FIR的具体内容会在第二章展开，在这里我们从高层简要的谈一下。

```

#include "stdio.h"

#define NUM_TAPS 4
void fir(int input, int *output, int taps[NUM_TAPS]);

const int SIZE = 256;

int main() {
    int taps[] = {1, 2, 0, -3, 0, 4, -5, 0, 1, -2, 0, -3, 0, 4, -5, 0};
    int out = 0;
    for (int i = 0; i < SIZE; i++) {
        fir(i, &out, taps);
    }
    printf("result = %d\n", out);
    if (out == -1452) {
        return 0;
    } else {
        return 1;
    }
}

```

(代码样例) 图1.8:四抽头FIR滤波器的代码

图1.8中的C代码可以作为一个HLS的任务描述。这段代码可以直接作为Vivado HLS工具的输入，工具会自动分析并产生一个等效的RTL电路。这个过程具体细节比较复杂，我们暂时不做深究，只需要把它当作一个编译器去理解，像是gcc，只不过这个编译器输出的是RTL硬件描述。编译器的复杂性是它非常关键的原因之一，因为它不需要用户理解每一个细节。但理解编译器如何工作其实有助于设计师写出更高效的代码，这点对于HLS尤其重要，因为综合电路的构建方式有很多种，只理解它软件流是不够的。比如HLS设计师需要考虑流水，内存排布，I/O接口这些软件设计师不需要考虑的内容。

回到编译器，理解它的关键问题在于：这段代码中产生的是什么电路？这个问题的答案分多钟，还和你所用的HLS工具有关。那么通常工具有以下几种合成方式：

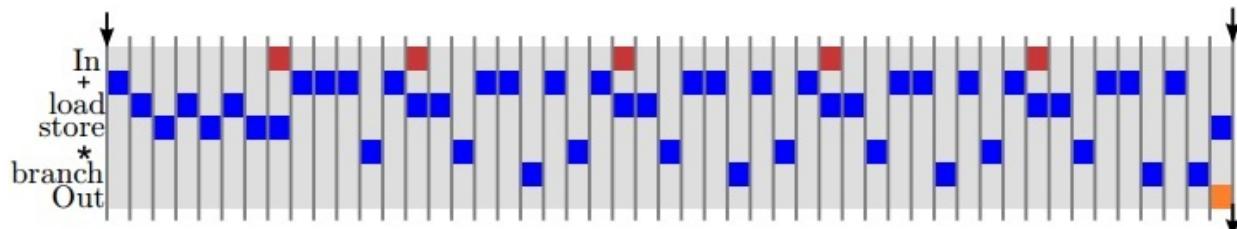
第一种可能的产出电路是按照顺序执行每行代码产出的电路，这时候工具就像一个简单的RISC处理器。下面的图1.9中的代码是图1.8中的代码在赛灵思Microblaze处理器下的汇编代码版本。虽然已经经过了优化，但还是有很多指令用来执行计算数组索引（array index）和控制循环。这样的指令我们假设它每个循环都要执行一次，那么我们在49个循环之后才能得到滤波器得出的结果。我们可以很明了地得到一个结果，那就是一个周期内执行的指令数是影响性能的一个重要的壁垒。有时候对于一个架构的提升就是让它处理的指令变得更复杂，让同一个指令能做的事情变得更多。HLS的一个特点就是在决定结构上的一些此消彼长的设计时，不再需要考虑让它适用于指令集的结构限制。在HLS设计中，设计出一个在同周期内执行成百上千个RISC级指令外加几百个周期程度流水的系统是非常常见的。

```

fir:
    .frame r1,0,r15          # vars= 0, regs= 0, args= 0
    .mask 0x00000000
    addik r3,r0,delay_line.1450
    lwi   r4,r3,8           # Unrolled loop to shift the delay line
    swi   r4,r3,12
    lwi   r4,r3,4
    swi   r4,r3,8
    lwi   r4,r3,0
    swi   r4,r3,4
    swi   r5,r3,0           # Store the new input sample into the delay line
    addik r5,r0,4           # Initialize the loop counter
    addk  r8,r0,r0           # Initialize accumulator to zero
    addk  r4,r8,r0           # Initialize index expression to zero
$L2:
    muli  r3,r4,4           # Compute a byte offset into the delay_line array
    addik r9,r3,delay_line.1450
    lw    r3,r3,r7           # Load filter tap
    lwi   r9,r9,0           # Load value from delay line
    mul   r3,r3,r9           # Filter Multiply
    addk  r8,r8,r3           # Filter Accumulate
    addik r5,r5,-1           # update the loop counter
    bneid r5,$L2
    addik r4,r4,1           # branch delay slot, update index expression

    rtsd  r15, 8
    swi   r8,r6,0           # branch delay slot, store the output
.end   fir

```



(代码样例) 图1.9:RISC风格下图1.8中的代码的汇编版。在赛灵思Microblaze处理器下施行。这段代码由 `microblazeel-xilinx-linux-gnu-gcc -O1 -mno -xl -soft -mul S fir.c` 指令产生

这是Vivado HLS默认下产出的是非常顺序化的结构。所谓顺序化的结构，是指循环和分支都被写作控制逻辑以控制寄存器、功能单元等部件。这其实和RISC处理器的概念相同，除了我们提到过产出的结果是RTL结构下的状态机。这种结构更倾向于限制那些使用资源去并行的功能单元。顺序化结构可以从大多数程序中生成，无需对原代码做太多的修改和优化，所以对HLS初学者非常的简单。但它同样存在一些缺陷。顺序化的结构很难解析码流，主要出于控制逻辑的复杂度。另外，控制逻辑负责规定任务延迟和任务间隔。顺序化结构的性能有时取决于处理的数据。

Vivado HLS可以产出更加流水，平行，性能上也更好的结构。其中之一叫做函数流水。函数流水结构是把函数内所有的代码都当作计算数据路径的一部分，再加上少量的控制逻辑。循环和分支被转换成无限制的结构。这种结构特点分明，容易分析，一般用于处理连续而简单的高码率数据。函数流水结构可以在更大的设计中充当组件，因为它的行为比较简单，方便共享资源，但这种结构的缺点在于适用范围相对较小，不是所有代码都可以被设计成平行结构。

用户可以通过在代码中添加`#pragma HLS pipeline`来指导Vivado HLS工具产生函数流水结构。这段指令需要一个参数来规划流水的起始间隔，也就是一个函数流水的任务间隔。图1.10展示了一个可行的设计----每周期一抽头的架构。任务用到了一个乘法器和一个加法器完成滤波器。这种设计的任务间隔和任务延迟都是4个周期。图1.11展示的是一个每周期一样本的结构，它使用了4个乘法器和3个加法器。这种设计的任务延迟和任务间隔都是1个周期，所以它每个周期都接受一个新的输入。当然这两种之外还有很多可行的设计，比如每周期两抽头设计，或每周期两样本设计，在一些特定应用中各自有各自的优势，我们将在第二章中讨论更多优化。

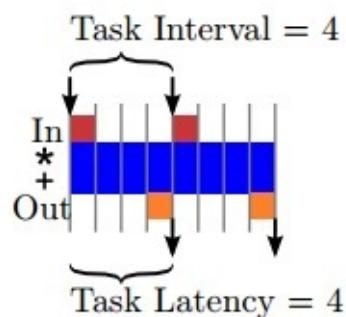
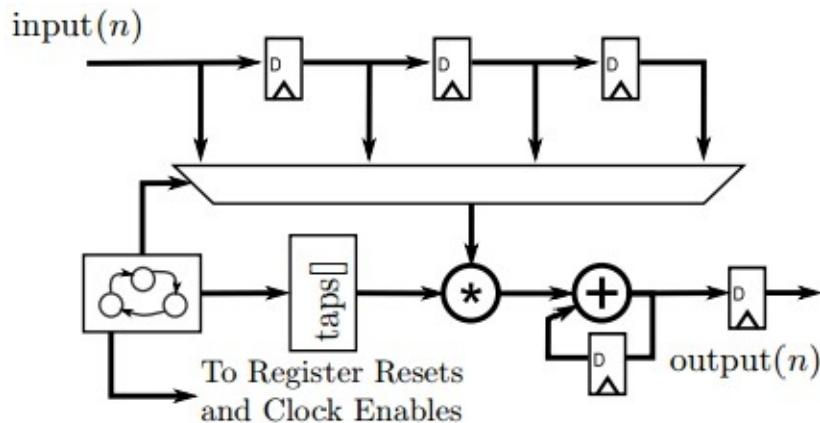
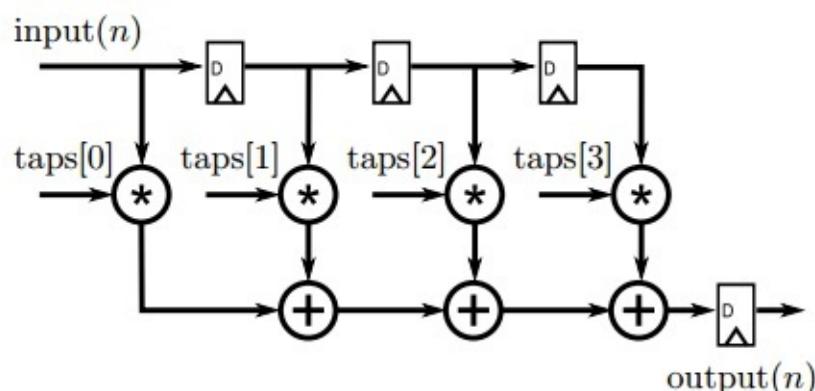


图1.10:每周期一抽头设计。可以由图1.8中的代码产出



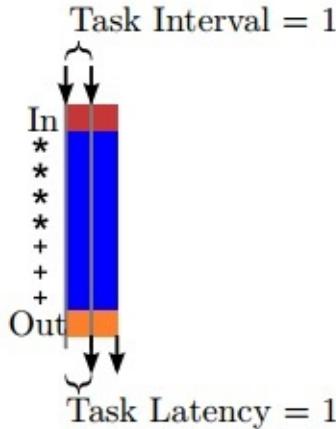


图1.11:每周期一样本设计。可以由图1.8中的代码加入函数流水指令产出

实际应用中，复杂的设计在顺序化和并行化的结构之间会有很多取舍的考虑。这些取舍在Vivado HLS中很大程度上取决于设计者的决定和代码内容。

1.4.3 处理速率的限制

我们看到了很多改变架构会改变任务间隔的例子，这样做通常来讲可以提升处理速率。但是读者需要意识到任何结构的任务间隔都是有一定的限度的。最关键的限制来自于递归和反馈循环，还有一些其他的例如资源限制也很重要。

递归（recurrence），这里是指某个部件的计算需要这个部件之前一轮计算的结果，递归是限制产力的重要因素，即使在流水结构中也是如此[56, 43]。分析算法中的递归并产出正确的硬件设计是非常关键的一步，同样，选择一个尽量避免很多递归的算法也是设计中非常关键的一步。

递归在很多代码结构中都会出现，比如静态变量（图1.8），顺序的循环（图1.10）。它存在于很多顺序化结构中，也有很多会随着改编成流水结构而消失。对于顺序化结构递归有时候不影响处理速率，但是在流水结构中是一个很不理想的状况。

另一个影响速率的关键因素就是资源限制，其中一种形式是设计边缘的跳线，因为一个同步电路中的每根跳线在每周期只能传送抓取1个比特的数据。因此，如果 `int32_t f(int32_t x)` 这样形式的函数作为一个单独模块在100MHz的频率和1的任务间隔下运行，它最大的数据处理量就是3.2G比特。另一种资源限制来自于内存，因为大多数内存每周期只支持一定次数的访问。还有一种资源限制来自于用户所给的限制，如果用户规定了在综合中可用的操作数，这其实是给处理率添加了限制条件。

```

#define NUM_TAPS 4

void block_fir(int input[256], int output[256], int taps[NUM_TAPS],
               int delay_line[NUM_TAPS]) {
    int i, j;
    for (j = 0; j < 256; j++) {
        int result = 0;
        for (i = NUM_TAPS - 1; i > 0; i--) {
#pragma HLS unroll
            delay_line[i] = delay_line[i - 1];
        }
        delay_line[0] = input;

        for (i = 0; i < NUM_TAPS; i++) {
#pragma HLS pipeline
            result += delay_line[i] * taps[i];
        }
        output[j] = result;
    }
}

```

(代码样例) 图1.12：另一种FIR滤波器代码

1.4.4 代码风格

每个工程师在设计时都该问自己：我写的这段代码有最好的利用算法吗？在很多情况下，我们追求的不是结果质量达到极致，而是代码更易于更改更灵活。虽然这其实是个因人而异的风格问题，但有些代码风格确实会限制HLS工具产出的结构的质量。

举例而言，在不同的工具中输入图1.8的代码，图1.10和图1.11都是可能的产出结果。但是加入了图1.12中的那些指令之后就会一定产出特定的一种结果。这个情况下延迟线被展开，乘积的for循环都被用流水的方式实施，产出的结构会于图1.11中的结构相似。

本章介绍了很多不同处理率的方法，其中最快的甚至到了每周期一样本的结构。但是，还有很多的应用需要更高的处理率，比如每周期多个样本。这样的设计需要怎样的代码呢？以设计一个每周期四样本的FIR滤波器为例，这样的设计需要多少资源（加法器和乘法器的数量）？与每周期一样本相比哪个资源使用更多？

我们将会在第二章具体讨论每种优化怎样影响性能和资源使用。

1.5 重建代码

写出一个非常优化的HLS代码不是一两步就可以完成的工作，设计者必须对程序的应用有很深的理解，才能让HLS工具利用指令产生最高效的结构。

在这本书接下来内容里，我们会以应用为主题，讨论几个常见应用的综合的过程，包括数字信号处理，排序，矩阵操作，视频处理。理解算法是非常重要的一步，因为对代码的调整经常不止于加几句指令，有时候还需要重写整段。

重建代码，对于工具链来说经常变成很难读懂的行为，需要与硬件对应好关系，所以它不仅要求对算法的理解还要求对硬件微结构有比较深的理解。一般来说现成的算法原代码产出的结构比普通的CPU程序还低效，即使使用流水，展开等方法也没起到太大的作用。所以最好的方法还是自己写出一个等效但适合高层次综合的算法。

重建代码与它原来的软件版本通常区别很大。一些研究指明重建是提升效率的非常重要的一步 [46,47,15,14,39]。用户在写重建代码时一定要时刻分析潜在的硬件设计。

在本书接下来的内容里，我们会展示之前提到的几个应用程序用于产出硬件结构的代码，具体包括FIR，离散傅里叶变换（DFT），快速傅里叶变换（FFT），稀疏矩阵乘矢量（SpMV），矩阵相乘，排序，哈夫曼编码。我们会讨论重建代码对最终硬件结构的影响，具体来说，针对每一章我们计划：

1. 强调重建代码对于高质量设计的重要性，比如在高性能和低使用面积上。
2. 对常见的内容提供重建的代码
3. 讨论重建对于硬件的影响
4. 使用必要的HLS指令以实现最好的设计

整本书来说，我们的示例会引导读者从最基础的设计到更有效的设计，因为我们相信理解来自于对示例的研究。每一章会采用不一样的优化策略，包括流水，数据流，循环优化，数组分离，带宽优化等等。另外，我们也会提供对于重建代码必要的洞察训练和知识。

1.6 本书结构

就像我们之前所说的，这本书的宗旨是以示例教学。每章将展示一个应用，逐步构建HLS，并一层层的优化。每章都只会用到一小部分优化策略，每章内容的难度也是逐步增加的。第二章我们会分析相对简单的FIR滤波器，而到了第九章我们会分析复杂的视频处理系统。

我们这样的教学方法当然也是会有弊有利，我们认为好处主要体现在：1) 读者可以清晰的看到优化是如何具体实施的 2) 每一章都会展示怎么具体的写HLS代码 3) 有些应用解释起来比较简单，但实际实施却是另外一回事，简单不完整的示例经常不够读者学习。

相对的，缺点主要在于：1) 大多数应用还是要求读者对计算和背景有一定的理解，而真正理解计算部分又有时需要比较深的数学背景。例如，FFT的最好结构需要读者深入理解DFT和FFT的数学背景。出于这个原因，有一些章节（比如第四章DFT第五章FFT）以一些数学介绍为开头。有些读者认为这些数学知识对于具体实施HLS没什么帮助，但我们认为这部分内容对于代码重建是非常必需的。2) 有时候一件没那么具体的示例其实能更好的概括代码，具体示例中细枝末节反而会让读者很疑惑。

每章的结构大致相同，一般会以一些必要的背景介绍开始。对于大多数章节程序的背景介绍没有程序本身听上去那么复杂，比如第七章矩阵相乘，但是还是有例如第四章DFT这样的章节我们会介绍大量的数学知识。介绍之后我们会提供一个基准方法——一个不经过任何优化但是结果正确的HLS构建方法。然后我们就会开始介绍不同的优化。每章内容包含的优化内容也有多有少，像第三章只比较强调带宽，第二章就描述了很多优化方法。一些非常关键的优化策略会贯穿全书被多次提到。

我们建议读者按顺序阅读本书。像我们在第二章会介绍后面出现的大多数优化策略，然后在后续的章节才会对其中的一些策略深入讲解。还有应用的难度也是逐渐增长的。但其实从各应用本身的内容上来说，各章交叉不大，所以如果读者已经是一个比较有经验的HLS设计师，那么完全可以根据需要只读某几章某一部分。比如说第十章排序，读者如果已经有了一定的HLS基础就不需要从头开始读这本书。

下面的表1.1提供了一个各优化的总览表，读者可以看到每章使用了哪些优化，其中第二章除了各种对于FIR滤波器的优化之外，还简单的介绍了一下HLS的设计过程。总体来说后面的章节会更注重某几个优化并详细介绍。

章节	FIR	CORDIC	DFT	FFT	SpMV	矩阵	直方图	视频	排序	哈夫曼
展开循环	x		x	x	x		x		x	
循环流水	x		x	x	x		x	x	x	x
带宽优化	x	x								x
函数内嵌	x									x
分层	x			x			x	x	x	x
数组优化			x	x	x	x	x	x	x	x
任务流水				x			x	x	x	x
测试平台					x	x			x	x
一同仿真					x					
实时计算						x		x	x	
接口交互								x		

表1.1:一个优化策略和章节的对照表

第三章到第五章可以算作一个系列，这个系列着重于建造数字信号处理模块（CORDIC，DFT，FFT）。这些章节都侧重于某一个优化策略，比如第三章的带宽优化，第四章的数组优化，第五章的数组优化和任务流水。以第四章DFT为例，第四章介绍了数组优化，特别介绍了怎样利用数组分离来提升片上内存带宽。这一章也提到了展开循环和循环流水，并且讲述了让这些优化共存的方法。

第五章描述了快速傅立叶变换的优化，它其实本身就是DFT的一个重构代码。FFT本身就是一个阶段化很明显的算法，所以非常适合任务流水。最终版优化代码需要一些其他优化包括循环流水，展开，数组优化等。每一章其实都在附录中的项目有所关联，他们最终的集成到一起可以组成一个无线交流系统。

第六章到第十一章对更多的应用做出了讲解。第六章讲述了如何使用测试平台和RTL同仿真，还讨论了一下数组和循环的优化。这些基本的优化策略很常见，在大多数程序中都有使用。第七章介绍了数据流的实时计算这个策略。第八章展示了两种应用（前缀和，直方图），这两个应用本身相对简单，但重建他们的代码需要很小心的实行优化。第九章会用很大篇幅讲述不同接口与交互的使用，比如视频直播需要某种特定的总线与内存接口。除此之外还需要一些数组和循环的优化。第十章介绍了几种排序算法，所以自然需要很大量的优化。最后一章则是建立了一个复杂的数据压缩结构，会包含大量复杂的模块。

第二章 FIR滤波器

2.1 概述

有限脉冲响应(FIR)滤波器在数字信号处理(DSP)领域中很常用——它们可能是这个领域应用最广泛的运算。因为它们可以采用高度优化的体系结构，所以它们非常适合于硬件实现。它有一个关键特性即对连续信号元素进行线性变换。这个特性可以很好的映射成一种数据结构（例如，FIFOs或者抽头延时线），这些数据结构可以在硬件中高效实现。一般来说，流处理很适合在FPGA中映射实现，例如，在本书中介绍的大多数例子都包含某种形式的流处理。

滤波器的两个基本应用是信号重建和信号分离。信号分离更常用到：将输入信号分离到不同部分。通常，我们认为它们是频率范围不同的信号，例如，我们可能设计低通滤波器，用来去除不感兴趣的高频信号。或者，我们为了解调某特定信号，设计一个带通滤波器来筛选该特定信号频率，例如，在频移键控解调过程中隔离频率。信号重建是指滤除可能混入有用信号的噪声和其他失真，例如，通过无线信道传输数据，信号重建包括平滑信号和移除直流分量。

数字FIR滤波器经常处理由采样连续信号产生的离散信号。最熟悉的是在时间上抽样，即，在离散信号下进行处理。这些处理都是在固定采样间隔的条件下进行的。例如，我们用模数转换器以固定时间间隔采样天线的电压。或者，也可以通过采样光电二极管电流来测量光的强度。或者，在空间中采样。例如，我们可以采样一个由光电二极管组成的图像传感器不同位置的值，来创建一个数字图像。在文档[41]中可以找到对信号和采样更深入的描述。

采样数据格式取决于应用场景。数字通信领域通常使用复数(in-phase和quadrature或I/Q值)来表示一个采样数据。在本章后续，我们将设计一个复数FIR滤波器来处理这样的数据。在图像处理领域，我们经常把每个像素看作一个样本。一个像素可以有多个字段，例如红色、绿色和蓝色(RGB)颜色通道。我们希望根据应用以不同的方式对这些通道进行滤波。

本章目的是提供一个从选用算法到应用高级语言综合进行算法硬件实现的基本流程。这个流程的第一步是对算法本身有深刻的理解。这使我们做代码重构和设计优化更加容易。下一节介绍FIR滤波器的理论和计算。本章的其余部分介绍对FIR滤波器进行各种基于HLS的优化。这是为了让读者对这些优化方式的有全面的认识。在后续章节中，每种优化方式都将进行详细介绍。

2.2 背景

对滤波器输入脉冲信号得到的输出信号为该滤波器的脉冲响应。线性时不变滤波器的脉冲响应包含关于滤波器的完整信息。顾名思义，FIR滤波器（严格的线性时不变系统）的脉冲响应是有限的。在已知滤波器脉冲响应的前提下，我们可以通过卷积的方法算出该滤波器任意输入的输出响应。这个运算过程结合滤波器脉冲响应（又可称为系数或阶）和输入信号用来计算输出信号。滤波器的输出也可以通过其他方式进行计算（例如，通过频域计算），在本章中，我们关注时域计算。

N-阶FIR滤波器的系数 $h[]$ 与输入信号 $x[]$ 的卷积可由差分方程表示：

$$y[i] = \sum_{j=0}^{N-1} h[j] \cdot x[i-j] \quad (2.1)$$

注意，要计算一个N-阶滤波器的输出值，需要N个乘法和N-1个加法。

滑动均值滤波器是低通FIR滤波器的一种简单形式，其所有系数都是相同的且和为1。例如3点滑动滤波器，其系数 $h = [1/3, 1/3, 1/3]$ 。同时由于它卷积核的形状特点，它也被称为矩形滤波器。或者你可以想象一个滑动均值滤波器，它将输入信号的几个相邻样本相加并求平均值。在上面卷积方程中，用 $1/N$ 代替 $h[j]$ 并对公式重新排布整理，达到对N个元素求平均值相似的效果：

$$y[i] = \frac{1}{N} \sum_{j=0}^{N-1} x[i-j] \quad (2.2)$$

每个输出信号可以按照上面公式计算，上面公式总共使用 $N-1$ 次加法和一次乘数为 $1/N$ 的乘法。很多时候最后的乘法也可以被其它运算操作重新组合并合并。因此滑动均值滤波器比常规FIR滤波器更简单。例如，当 $N = 3$ 时，我们这样运算来计算 $y[12]$ ：

$$y[12] = \frac{1}{3} \cdot (x[12] + x[11] + x[10]) \quad (2.3)$$

这个过滤器是因果系统，这意味着输出数据与当前输入值及以前的数据有关。系统的因果特性是可以改变的，例如以当前样本为数据中心时刻的处理，即 $y[12] = 1/3 \cdot (x[11] + x[12] + x[13])$ 。虽然从根本上来说因果特性是系统分析的一个重要属性，但是对于硬件实现来说它没有那么重要，一个有限非因果滤波器可以通过数据缓冲或者重排列来实现转因果系统的转换。

滑动均值滤波器可以用来平滑信号，例如去除随机(大部分是高频)噪声。随着 N 的阶数越高，我们会平均更多的样本，相应的我们必须进行更多的计算。对于滑动均值滤波器， N 值的增大等效于减小输出信号带宽。根本上讲，它就像一个低通滤波器(虽然不是非常理想)。直觉上，这样解释是有意义的。当我们对越来越多的样本进行平均时，我们消除了输入信号中更高的频率分量。也就是说，“平滑”等同于降低高频分量。滑动滤波器是最优的减少白噪声同时保持最陡峭阶跃响应的滤波器，即高给定边缘锐度的情况下把噪声压到最低。

注意通常来说，滤波器系数可以用来精确地创建许多不同类型的滤波器：低通滤波器，高通滤波器，带通滤波器等等。一般来说，设计滤波器时，阶数越大提供的自由度越多，设计滤波器的性能越好。有大量的文献致力于为特定应用需求设计滤波器系数。在实现滤波器时，这些系数数值基本上是无关的，我们可以忽略系数本身是如何求得。但是，正如我们看到的滑动均值滤波器，滤波器的结构和特定系数对实现该滤波器需要执行的操作数产生很大影响。例如，对称滤波器有多个相同系数，我们可以将系数分组来减少乘法次数。在其他情况下，可以将固定滤波系数转换为移位和加法运算[34]。在这种情况下，滤波器系数可以极大改变滤波器实现性能和所消耗资源[52]。但是我们要暂时忽略这一点，重点关注设计具有固定系数的滤波器结构特点，而不是单纯利用固定系数滤波器运算。

2.3 FIR结构基础

编程实现11阶FIR滤波器代码如图2.1。这个函数有两个输入端口，一个端口是输入数据 x 和另一个端口是输出结果 y 。由于每次执行该函数会提供一个函数输入数据并接收一个函数输出数据，多次调用这个函数后，完成整个输出信号的计算。因为我们在获取更多信号时这段代码可以根据需求调用很多次，所以这段代码可以方便地用流模式架构进行建模。

滤波器系数存储在函数内部声明的数组 $c[]$ 中，定义为静态常数。注意系数是对称的。即它们以中心值 $c[5] = 500$ 为镜像对称。许多FIR滤波器具有这种对称性。我们可以利用这个特点来减少数组 $c[]$ 所需的存储容量。

代码对不同变量类型使用 **typedef**。虽然这不是必需的，但是可以方便地更改数据类型。正如我们后续要讨论的位宽优化——特别是针对每个变量设定整数和分数位数——其在性能和资源方面优化显著。

重写代码利用滤波器系数的对称性，也就是说优化C[]，使它含有6个因素(C[0]到C[5])，代码的其余部分还有那些改变？这些改变对资源有什么影响？它如何改变性能？

```
#define N 11
#include "ap_int.h"

typedef int coef_t;
typedef int data_t;
typedef int acc_t;

void fir(data_t *y, data_t x){
    coef_t[N] = {
        53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53
    };
    static
    data_t shift_reg[N];
    acc_t acc;
    int i;
    acc = 0;
    Shift_Accum_Loop:
    for(i = N - 1; i >= 0; i--) {
        if(i == 0) {
            acc += x * C[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * C[i];
        }
    }
    *y = acc;
}
```

图2.1：11阶FIR滤波器代码，该代码功能正确但完全没有优化。

该代码设计为流函数。它一次只接收一个数据，因此它必须存储以前的数据。由于这是一个11阶滤波器，我们必须存储之前的10个数据。这是shift_reg[]矩阵的作用。因为矩阵数据在函数中被多次调用，所以该矩阵声明为静态类型。

每次 **for** 循环有两个基本运算。首先，它对输入数据执行乘累加操作(当前输入数据x和存储在shift_reg[]中之前输入的数据)。每次循环执行一个常量与一个输入数据的乘法，并将求和结果存储在变量acc中。同时该循环通过shift_array来实现数值移动，它的操作行为像FIFO。它存储输入数据x到shift_reg[0]中，并通过shift_reg将之前的元素“向前”移动：

```
shift_array[10] = shift_array[9]
shift_array[9] = shift_array[8]
shift_array[8] = shift_array[7]
...
shift_array[2] = shift_array[1]
shift_array[1] = shift_array[0]
shift_array[0] = x
```

Shift_Accum_Loop的标签不是必需的，但是它对调试很有帮助。Vivado HLS工具将这些标签添加到代码视图中。

在for循环完成后，acc变量是所有输入数据和FIR系数卷积运算的结果。最后的结果赋值到FIR滤波器函数输出端口y。这就完成了按照流处理流程计算得到的FIR滤波器的一个输出结果。

这个函数没有高效地实现FIR滤波器。它是串行执行的，并且使用了大量不必要的控制逻辑。后续部分提供一些优化手段来提高它的性能。

2.4 计算性能

在我们进行优化之前，有必要定义精确的度量。当评价设计性能时，必须仔细地说明度量。例如，有许多不同的方法来说明设计运行的“快速”。例如，你可以说运算速度为X位/秒。或者它可以执行Y操作/秒。针对FIR滤波器运行速度的度量方法为滤波操作数/秒。然而，另一种度量方法是乘累加操作:MACs/秒。在某种程度上说每种度量方法都是相互关联的，但是当比较不同的实现方法时同类比较就变得很重要。例如，将一个采用位/秒度量方法的设计与另一个使用滤波操作数/秒的设计直接进行比较，很可能会产生误导;充分了解不同设计的相对优势，需要我们使用相同的度量方法来比较它们，而这可能需要额外的信息，例如从滤波操作数/秒等效成位/秒，需要了解关于输入和输出数据位宽的信息。

前面提到的所有指标单位都是秒。高级语言综合工具以时钟周期数和时钟频率为评价单位。频率与时钟周期成反比。应用它们统计1秒内运算数量。时钟周期数和频率都很重要:一个工作频率很低只需要一个时钟周期的设计，不一定比工作频率很高但需要10个时钟周期的设计更好。

优化时钟周期的同时还要对时钟频率进行优化，这是Vivado HLS工具一个复杂的功能。注意可以使用TCL指令为Vivado HLS工具指定目标运行频率。例如,creat_clock -period 5 指令设置代码运行时钟周期5ns即时钟频率200 MHz。请注意这只是一个目标时钟频率，它主要影响工具优化代码为多少个运算操作。在生成RTL之后，Vivado HLS工具在这个时钟频率进行目标初始时序估计。然而电路性能还有一些不确定性存在，只有当设计在全部布局布线之后电路性能才能确定。

虽然更高频率在通常情况下是能达到更高性能的关键，但对于整个系统来说提高时钟频率并不一定是在整个系统中最优的优化方式。较低的时钟频率为工具在单个周期中组合多个相关操作提供了更多的时间余量，这个过程叫做操作链接。在这个过程中有些情况下可以通过提高逻辑综合优化效果和增加适配器件的代码规模来提高性能。改进的操作链接可以提高(或者降低)流水处理数据输入间隔。一般来说，提供一个频率约束但该频率值不超过实际运行时钟频率。将时钟周期约束在5–10ns的范围内通常是一个好的开始。一旦你开始优化设计，你可以改变时钟周期并观察优化结果。我们将在下一节中详细介绍操作链接。

因为Vivado HLS处理的是时钟频率估计值，在估计中它确实包含了一些可能使估算出错的余量。这个余量是保证在生成RTL电路时有足够的时序余量可以成功完成布局布线。这个余量可以使用set_clock_uncertainty 的TCL脚本进行直接设置。注意此指令只影响HLS生成RTL，与RTL级时序约束中时钟不确定性不同。由Vivado HLS生成的对RTL实现流程的时序约束完全是基于目标时钟周期的约束。

把你的任务需求和计算性能度量结合起来是很有必要的。在我们示例中，fir滤波器每次执行都会有一个数据输出。但是对于每一次fir运算我们执行的是 $N = 11$ 次乘累加操作。因此如果你的度量指标单位是乘累加/秒，你应该计算fir运算延迟秒数，然后除以11，得到一次乘累加操作需要的时间。

当我们进行流水线和其他优化时，性能统计变得更加复杂。在这种情况下，理解处理间隔和处理延迟之间的区别是很重要的。现在是更新你对这两个性能指标理解的好时机。关于这两个指标我们已经在1.4章中讨论过。后续我们将继续讨论不同优化策略如何对不同性能指标产生影响。

2.5 操作链接

操作链接 是Vivado HLS为了优化最终设计而进行地一项重要优化。尽管设计人员没有太多控制权，但设计人员要明白它的工作原理尤其是在性能方面的影响，这是尤为重要的。考虑FIR滤波器在各阶运算中做的是乘法累加操作。假设加法运算需要2个ns，乘法运算需要3个ns。如果我们把时钟周期设为1ns（或者等效时钟频率1Ghz），那么它要花费5个时钟周期来完成乘累加操作。如图2.2 a所示。乘法操作需要3个周期，加法操作需要两个周期。乘累加操作总时间为5个时钟周期 \times 1ns = 5ns。因此我们处理性能为1/5ns = 2亿次乘累加/秒。

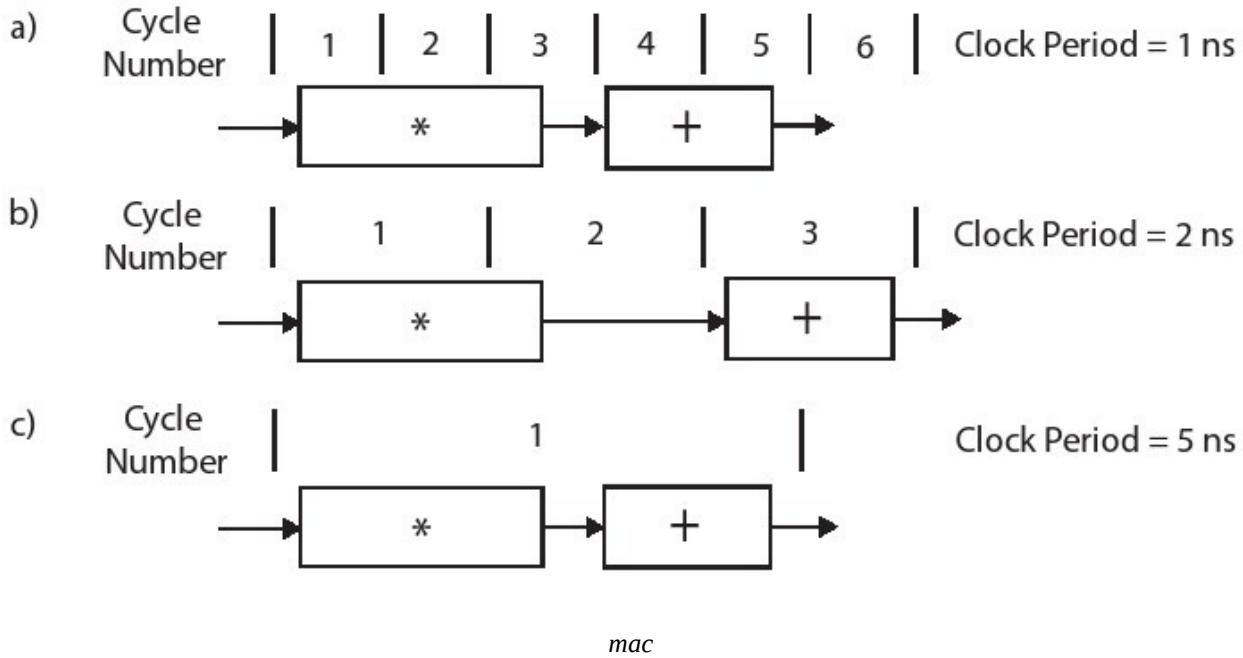


图2.2:随着时钟周期的延长，乘累加操作的性能也会发生变化。假设乘法运算需要3个ns，加法运算需要2个ns。图a)时钟周期为1ns时，一个乘累加操作需要5个周期，因此处理性能是2亿次乘累加/秒。图b)时钟周期为2ns，乘累加运算需要3个时钟周期，因此处理性能约1.67亿次乘累加/秒。图c)时钟周期个5ns。通过使用操作链接，乘累加操作需要一个时钟周期，因此处理性能是2亿次乘累加/秒。

如果我们将时钟周期增加到2ns，乘法运算用时将超过两个周期，那么加法操作必须等到第3周期才开始，它可以在一个周期内完成。因此乘累加操作需要3个周期，所以总共需要6ns。这时处理性能大约为1.67次乘累加操作 /秒。这个结果比之前1ns时钟周期的要低。这个可以用第2个时钟周期中没有执行任何操作的“死时间”来解释。

然而增加时钟周期并不是在所有情况下都会导致更糟糕的结果。例如，如果我们将时钟周期设为5ns，我们可以使用操作链接将乘法和加法在同一个时钟周期内执行。如图2.2 c所示，因此乘累加操作在5ns的时钟周期下只需要1个时钟周期，因此我们可以执行2亿次乘累加 /秒。这与图2.2 a时钟周期更快（1ns）的处理性能相同。

到目前为止，我们可以在一个时钟周期内只执行两个操作。可以在一个时钟周期内链接多个操作。例如如果时钟周期是10ns，我们可以按顺序执行5个加法操作，或者我们可以实现两个连续乘累加操作。

显而易见，时钟周期在Vivado HLS优化设计的过程中起着重要作用。时钟周期与Vivado HLS其他优化方式一起使用，使时钟周期对设计优化的影响变得更加复杂。完全理解Vivado HLS整个优化过程并不重要，这个观点是很明确的，因为每个新版本都在不断改进提升这个工具的性能。然而，重要的是要了解这个工具是如何工作的。这更有利你更好地理解优化结果，甚至使你编写的代码更加优化。

```

Shift_Accum_Loop:
for(i = N-1;i > 0;i--){
    shift_reg[i] = shift_reg[i-1];
    acc += shift_reg[i] * c[i];
}

acc += x * c[0];
shift_reg[0] = x;

```

图2.3:将for循环中条件语句删除，可以实现更有效的硬件结构。

由于Vivado HLS对不同的时钟频率可以生成不同的硬件结构。因此整体性能的优化和最佳目标时钟周期的确认仍然需要用户的创造力。在大多数情况下，我们建议坚持在小范围时钟周期内进行优化。例如在本章项目中，我们建议将时钟周期设为10ns，并将重点放在理解不同优化手段(例如流水处理)如何用于创建不同的处理架构。100Mhz时钟频率是相对容易实现的，而且它提供了良好的初始结果。当然可以创建更快时钟频率的设计。200Mhz以及更快的频率是有可能设计的，但这时经常需要在时钟频率和其他优化策略之间进行更好的权衡。你可以通过更改时钟周期来观察处理性能上的差异。不幸的是，这里没有好的准则来选择最佳频率。

将基本FIR架构的时钟周期(图2.1)从10ns改为1ns。哪个时钟会提供最好的性能?哪个时钟会是面积优化最优?你为什么会这样认为?你理解趋势了吗?

2.6 代码提升

for循环内部的if/else语句效率很低。在代码中每个控制结构，Vivado HLS会生成硬件逻辑电路来检查条件是否满足，这个检查在每个循环中都执行。此外这种条件结构限制了if或else分支中语句的执行;这些语句只有在解决了if条件语句之后才能执行。

当 $x==0$ 时if语句进行检查，这个只发生在最后一次迭代中。因此if分支中的语句可以从循环中“升起”。也就是说，我们可以在循环结束后执行这些语句，然后在循环中删除if/else控制流。最后我们必须改变执行“第0次”迭代的循环边界。这个转换如图2.3所示。这显示了for循环所需要的更改。

最终结果是实现一个更加紧凑的结构，适合进一步的循环优化，例如，展开和流水线。我们稍后讨论这些优化。

对比删除if/else之前和之后的实现结果。它们有哪些性能不同？它们的资源变化了多少？

```

TDL:
for(i = N - 1;i > 0;i--){
    shift_reg[i] = shift_reg[i - 1];
}
shift_reg[0] = x;

acc = 0;
MAC:
for(i = N-1;i >= 0;i--)
{
    acc += shift_reg[i] * c[i];
}

```

图2.4:将for循环分解为两个独立循环的代码片段。

有变化？

2.7 循环拆分

我们在for循环中执行两个基本操作。第一个是通过shift_reg数组进行数据移位。第二个是进行乘累加运算来计算输出样本。循环分裂是分别在两个循环中实现各自操作。虽然这样做看起来不像是一个好主意，但是这样做允许我们在每个循环上分别进行优化。这可能是有利的，尤其是在对不同循环进行不同优化策略的情况下。

图2.4中代码显示了手动循环拆分优化结果。代码片段将图2.3的循环分割成两个循环。注意两个循环的标签名称，第一个是TDL 第二个是MAC。延时线(TDL)是数字信号处理中FIFO操作的术语;MAC是“乘累加”的缩写。

比较循环拆分前后，实现性能上有什么不同?资源使用情况如何变化?

每个循环单独拆分往往不能提高硬件实现效率。但是它可以实现每个循环独立地进行优化，这可能比优化原始的整体for循环更可能得到好结果。反之亦然;将两个(或多个)循环合并到一个循环中可能会产生最好的结果。这高度依赖于大多数优化都是正确的应用场景。一般来说对于如何优化代码没有“最优法则”。优化思路不一样导致优化过程也会有差异。因此重要的是要有很多可以使用的技巧，更好的是对优化工作原理有深入的了解。只有这样，你才能是得到最好的硬件算法实现。让我们继续学习一些额外的技巧.....

```

TDL:
for(i = N - 1;i > 1;i = i - 2){
    shift_reg[i] = shift_reg[i - 1];
    shift_reg[i - 1] = shift_reg[i - 2];
}
if(i == 1){
    shift_reg[1] = shift_reg[0];
}
shift_reg[0] = x;

```

图2.5:手动展开fir11函数中TDL循环。

2.8 循环展开

默认情况下Vivado HLS 将循环综合成顺序执行方式。该工具创建了一个数据路径，该路径实现了循环体中语句的执行。数据路径顺序执行每次循环迭代运算。这就创建了一个有效的区域架构;但是它限制了在循环迭代中可能出现的并行运算。

循环展会通过循环次数(称为因子)来复制循环的主体。每次循环迭代循环次数减少相同因子。在最好情况下，当循环中没有任何语句依赖于前一次迭代生成的任何数据时，这时循环主体大大增加并行性，从而使系统运行速度更快。

图2.4中的第一个for循环(带有标签TDL)通过shift_reg数组进行数据移位。循环迭代从最大值($N-1$)到最小值($i = 1$)。通过展开这个循环,我们可以创建一个并行执行的数据路径进行移位操作。

图2.5显示了以因子2将循环展开后的结果。此代码重复循环主体两次。每次循环迭代都执行两个移位操作。因此我们迭代次数变为原来的一半。

请注意，在for循环之后还有一个if条件判断。当循环次数为奇数时，这个判断是必需的。在这种情况下，我们必须自己执行最后一次“半”迭代。if语句中的代码执行最后的“半”迭代，即将数据从shift_reg[0]移动到shift_reg[1]。

还要注意for循环展开开头的结果。减量操作从 $i--$ 变为 $i = i-2$ 。这是因为我们在每次迭代中都做了两次“工作”，因此我们应该减少2而不是1。

最后，终止for循环的条件从 $i > 0$ 变为 $i > 1$ 。我们应该确保“最后一次”迭代能够完成而不引发错误。如果最后的迭代循环执行 $i= 1$,那么第二个语句会试图读取shift_reg[-1]。我们为了不执行这个非法操作，所以在for循环之后的if语句中进行最后的移位操作。

编写相应的代码按照因子3对这个TDL循环进行手动展开。这如何改变循环体呢?循环头需要哪些更改?for循环之后的if语句附加代码仍然是必需的吗?如果是，那现在又有什么不同呢?

循环展开可以提高总体性能，前提是我们可以并行执行一些(或全部)语句。在展开代码中，每次迭代要求我们从shift_reg数组中读取两个值;而且我们还要在同一个数组上写两个值。因此如果我们希望并行地执行这两个语句，我们必须能够在相同的周期内对shift_reg数组执行两个读操作和两个写操作。

假设我们将shift_reg数组存储在一个BRAM中，BRAM有两个读端口和一个写端口。因此，我们可以在一个循环中可以执行两个读操作，但我们只能在两个周期内顺序进行写操作。

有一些方法可以在一个循环中执行这两个语句。例如，我们可以将所有shift_reg数组的值存储在独立寄存器中。每个周期都可以对每个寄存器进行读写。在这种情况下，我们可以在一个循环中执行这两条的语句。你可以使用指令#pragma HLS array_partition variable=shift_reg complete设置Vivado HLS ,使其将所有放在shift_reg数组中的值，放到寄存器中。这是一个重要的优化，因此稍后我们将详细讨论array_partition 指令。

用户可以使用unroll指令告诉Vivado HLS自动循环展开。为了自动实现图2.5中手动完成的循环展开，我们应该将指令#pragma HLS unroll factor=2放入代码的主体中，具体位置为for循环后面。虽然我们一直可以手动执行循环展开，但是允许工具为我们执行循环展开实现起来更容易，它使代码更易阅读;而且可以减少编码错误。

使用Unroll指令将TDL循环自动展开。当你增加unroll因子时，资源(FFs、LUTs、BRAMs、dsp48等)的数量如何改变?它如何影响吞吐量?当使用数组分区指令与unroll指令一起使用时会产生什么结果?如果你不使用unroll指令会产生什么结果?

现在考虑图2.4中第二个for循环(带有MAC标签)。该循环将数组c[]中的值与shift_reg[]数组的值相乘。在每次迭代中，它要访问两个数组中第i个值。然后把乘法结果加到acc变量中。

这个循环的每次迭代执行一次乘法和一次加法操作。每次迭代执行一次从数组shift_reg[]和数组c[]的读取操作。将这两个值相乘的结果累加到变量acc中。

加载和乘法操作在for循环中是独立的。加法操作取决于它是如何实现的，可能取决于前一次迭代运算的结果。但是，可以展开这个循环并删除此迭代运算的依赖关系。

图2.6展示的代码，该代码进行展开因子为4的MAC循环展开。第一个for是展开的循环。for循环起始控制语句的修改与我们展开TDL的修改方式类似。运行范围更改为*i*>=3，并且在展开循环的每次迭代中，*i*的值每次减少4。

虽然在原始for循环内有依赖关系，但在循环展开的情况下，依赖关系优化没了。循环依赖关系是由于acc变量的存在；由于每次迭代运算中乘累加的结果都要写入到这个变量中，而且每次迭代也读取这个寄存器值(以执行累加和)，它就形成了在迭代运算中一个读写的依赖项。注意由于编写实现方式的改变，在展开后循环中不再有依赖acc变量项。因此在展开的for循环中我们可以自由地并行运算四个独立MAC操作。

在for循环展开之后还有一个加法for循环。这是执行任意剩余部分迭代运算的必要条件。就像我们要求TDL中有if语句一样，它在可能的最后一次迭代操作中进行运算。当展开的for循环初始的迭代次数不为4的倍数时就会出现这种情况。

```
acc = 0;
MAC:
for(i = N - 1;i >= 3;i -= 4){
    acc += shift_reg[i] * c[i] +
    shift_reg[i - 1] * c[i - 1] +
    shift_reg[i - 2] * c[i - 2] +
    shift_reg[i - 3] * c[i - 3];
}

for(;i >= 0; i--){
    acc += shift_reg[i] * c[i];
}
```

图2.6:在fir11函数中以展开因子4手动将MAC循环展开。

再一次说明，我们可以通过将指令#pragma HLS unroll factor=4插入到MAC循环体中，控制Vivado HLS工具自动以因子4将循环展开。

通过在指令中指定优化参数 skip_exit_check，Vivado HLS将不会增加对循环最后部分迭代运算的检查。当你知道循环永远不需要这些最后部分的迭代时，跳过运算检查是很有用的。或者执行最后几次迭代运算不会对结果产生(主要的)影响，因此这些运算可以被跳过。通过使用这个指令，Vivado HLS工具不会创建额外的for循环迭代。因此产生的硬件更简单，而且资源利用率更高。

当没有指定因子参数时for循环将完全展开。这相当于以最大迭代次数的展开；在这种情况下，完全展开和以因子11展开两者是等效的。在这两种情况下，循环主体都被复制了11次。而且循环控制没有什么作用；没有必要保留一个计数器来检查循环退出条件是否满足。为了进行完整的展开，循环的边界必须是静态确

定的，即在编译时Vivado HLS必须能够知道for循环的迭代次数。

完整的循环展开实现程序最大程度的并行性，这样的代价是需要很多资源。因此，可以在“较小”的循环上执行完整的循环。但是大迭代次数的循环展开(例如迭代一百万次)通常是不可行的。通常情况下，Vivado HLS将运行很长一段时间(并且往往在经过几个小时综合之后都会失败)，如果这样的循环进行展开，它展开结果会是生成非常大的代码。

如果你的设计在15分钟内不能综合完成，你应该仔细考虑优化效果。当然大型设计可能会花费Vivado HLS大量时间进行综合。但是作为一个初始的用户，你的设计应该相对快速地综合完成。如果花了很长时间，那很可能意味着你使用了一些不打算使用的指令明显扩展了综合代码。

针对MAC循环设置不同的展开因子综合一系列设计。性能如何变化?展开因子数是如何影响资源数量?将这些结果与通过展开TDL发现的趋势进行比较。

2.9 循环流水

在默认情况下，Vivado HLS以顺序方式综合循环操作。例如图2.1中的For循环将依次执行循环中的每次迭代。也就是说，第二次迭代中的所有语句是在第一次迭代的所有语句完成时才会发生;对于后续的迭代也是如此。即使在并行执行的迭代语句中，也可能发生顺序执行方式。其他情况下，在前一次迭代所有语句执行完成之前，可以启动后续迭代中的一些语句。这种执行行为除非设计者明确指出它应该这样做，否则不会自动发生。这产生了 循环流水 的思想，它允许同时执行多个循环迭代运算。

考虑图2.4中MAC循环。每次迭代执行一个乘累加(MAC)操作。在for循环体内部这个MAC运算四个操作:

- 读取c[]:从c数组加载指定数据。
- 读取 shift_reg[]:从shift_reg数组加载指定数据。
- *:数组c[]和shifit_reg[]相乘。
- +:将这些相乘的结果累积到acc变量中。

MAC for循环中一次迭代运算的流程图如图2.7 A所示。读取操作需要两个时钟周期，这是因为第一个时钟周期提供内存地址，第二个时钟周期完成数据传递。由于这两个操作之间没有依赖关系，所以可以并行执行。* 操作可以从第2个周期开始;假设它花费三个周期才能完成，即完成时是第4个时钟周期。在第4个周期，+操作开始并且能够完成。因此整个MAC循环需要4个时钟周期才能完成。

有许多与for循环相关的性能指标。迭代延迟 是执行一次循环运算所需的时钟周期数。MAC 循环迭代延迟为4个时钟周期。for 循环延迟 是完成整个循环所需的时钟周期数。这个时钟周期数包括初始化周期数(例如,i = 0),条件判断周期数(例如,i>= 0),和增量计算周期数(例如,i--)。假设这三个for循环控制语句与循环并行执行，Vivado HLS报告此MAC延迟为44个时钟周期。这个时钟周期数是由迭代的次数(11)乘以迭代延迟(4)再加上一个判断循环停止的额外周期。然后减去1。也许这里唯一奇怪的是“减1”。我们后面就会讲到。首先在下一次迭代开始时需要一个额外时钟周期检查条件语句是否满足(不满足)，然后退出循环。现在来解释“减1”:Vivado HLS通过数据输出就绪的时钟周期来计算延迟。在这种情况下，最后一个数据在第43周期准备好。也就等效于在第43周期结束第44个周期开始时写入一个寄存器。另一种计算延迟的方式为延时时钟周期数量等于输入数据和输出数据之间的最大寄存器数。

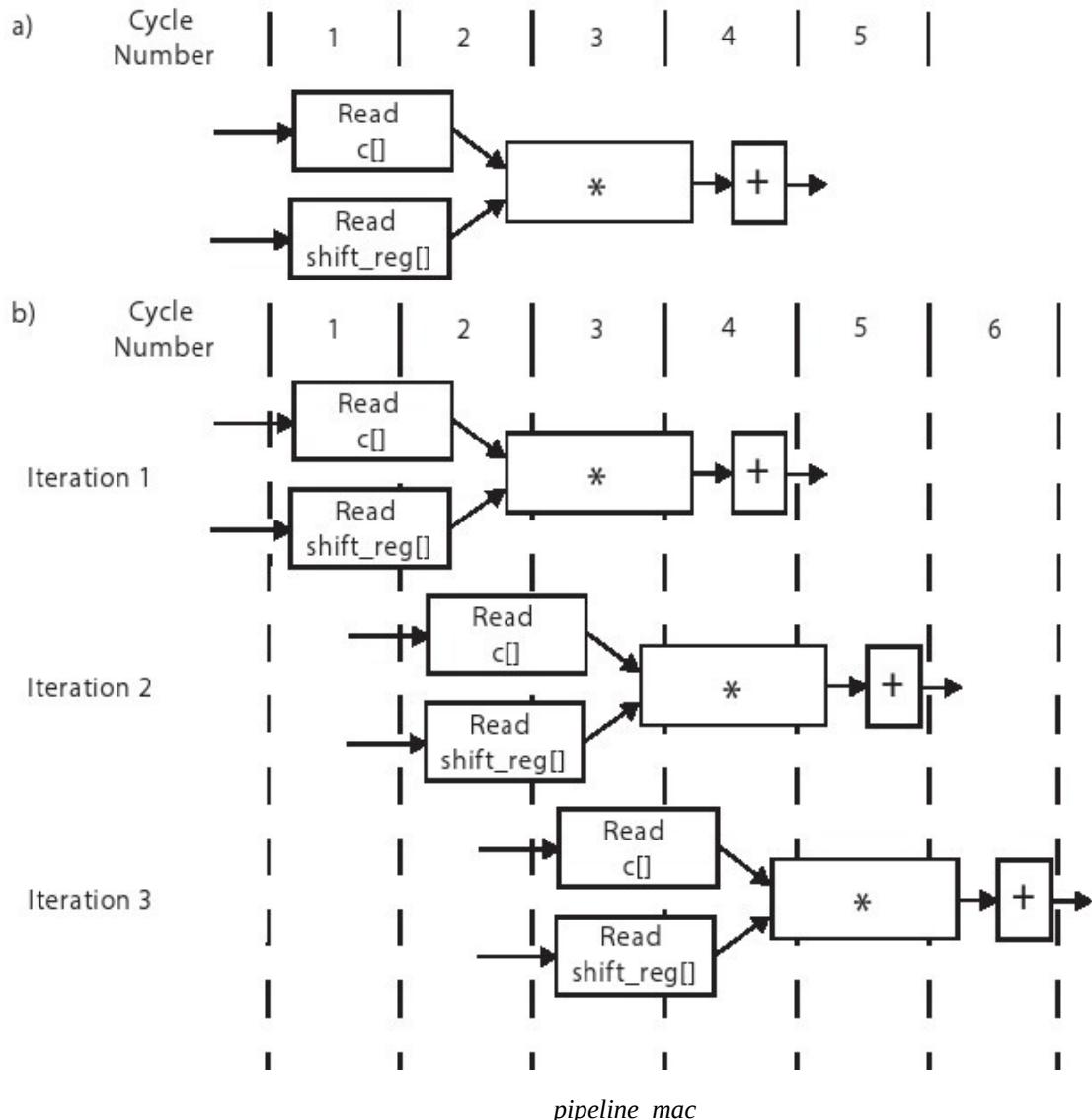


图2.7: a)表示MAC for循环时序图。b)表示三个迭代运算用流水形式优化的MAC for循环。

循环流水线是将for循环多个迭代运算进行重叠的优化。图2.7 b)是MAC for 循环流水示例。图中显示同时执行三个迭代。第一个迭代等价于图2.7 a中所示的非流水线版本，其区别是后续迭代运算的开始时间。非流水线版本，第二次迭代在第一次迭代完成后开始，即第5个时钟周期。然而流水线版本第二次迭代可以在第一次迭代完成之前启动。在图中，第二次迭代从第二个周期开始，第三次迭代从第三个周期开始。其余的迭代在每个时钟开始运算。因此，最后一次迭代即第11次迭代将从第11周期开始，并且在第14周期中完成，因此循环延迟是14。

循环起始间隔(II)是另一个重要的性能度量。它定义为本次循环到下一次循环开始的时钟周期数。在本例中，循环II值为1，这意味着我们可以在每个周期中启动新的迭代循环。图2.7 b中图形化地描述了这一点。II值可以使用指令进行设置。例如指令#pragma HLS pipeline II=2 设置 Vivado HLS 工具II=2。请注意由于代码中的资源约束或逻辑依赖关系，II的设定值不一定总能实现。输出报告将告诉你Vivado HLS工具能够实现什么目标。

设置II值为1，并以1的增量进行递增。增加II的数值对循环延迟产生什么样的影响？趋势是什么？在某种程度上，将II设为更大的值是没有意义的。这个例子中的值是多少？对于普通循环你如何描述II值？

任何for循环都可以进行流水操作，现在我们考虑将TDL循环进行流水操作。TDL for循环有一个与MAC for循环相似的头。循环主体是按照2.3节中介绍的方式进行数组元素移位。每次循环有两个操作：一个读，一个写到shift_reg数组。循环迭代延迟为2个时钟周期。读操作需要两个周期，写操作在第二个周期末尾开始执行。for循环按照非流水操作需要20个时钟周期。

通过在循环头部后面插入指令#pragma HLS pipeline II=1将这个循环进行流水优化。综合的结果是循环间隔为1个时钟周期。也就是说每个时钟周期都可以开始循环迭代运算。

通过对示例进行微小改动，我们可以演示当由于资源限制Vivado HLS工具不能实现II=1的情况。为此，我们要指明shift_reg数组的内存类型。我们也可以不指定资源，将它留给Vivado HLS工具来决定。但是我们使用指令来指定内存类型，例如，指令#pragma HLS resource variable=shift_reg core=RAM_1P 强制Vivado HLS工具使用单端口RAM。当使用该指令与循环流水优化相结合时，Vivado HLS工具将无法使用II=1来连接此循环。这是因为这段代码流水操作需要在同一周期中同时进行读写操作。使用单端口RAM是不可能实现的。这在图2.8 b中很明显。注意第2个时钟周期，我们需要在迭代1中完成对数据shift_reg的写操作，在迭代2中对相同数组进行读操作。我们可以删除II=1的确定优化（例如，#pragma HLS pipeline），从而允许HLS有更多调整自由度。在这种情况下，HLS将自动增加初始间隔，直到找到可行的流程表。

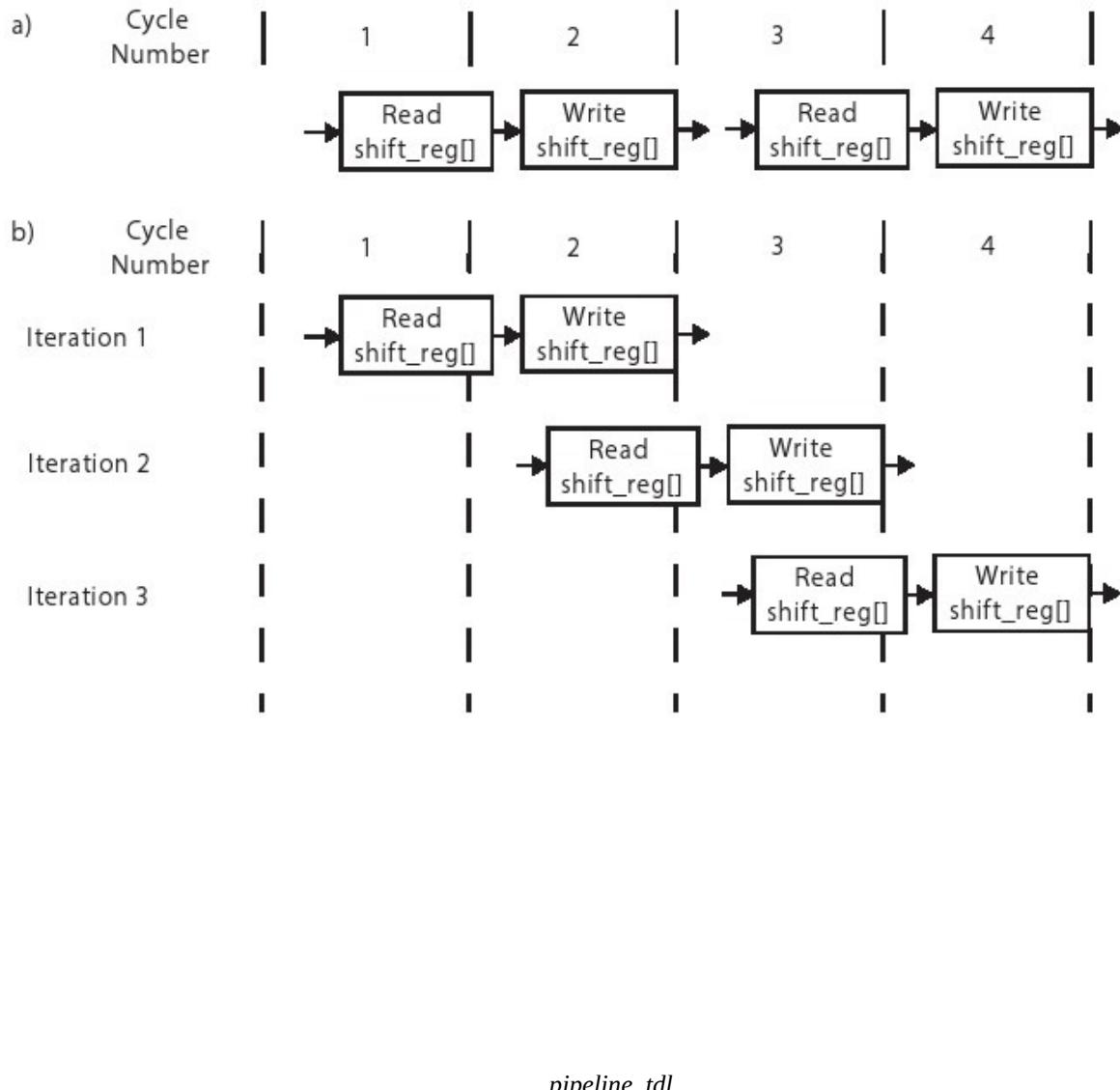


图2.8:a)TDL for循环两次迭代流程表。b)II=1 , TDL for循环三次迭代流程表。

RESOURCE指令允许用户强制设置Vivado HLS工具进行运算操作到硬件资源的映射方式。这种映射可以在矩阵（如上所示）同时也可以在变量中。思考代码 `a=b+c`；我们可以用RESOURCE指令 `#pragma HLS RESOURCE variable=a core=AddSub_DSP` 来告诉Vivado HLS工具，加法操作应DSP48资源来实现。在Vivado HLS文档[63]中有各种硬件核资源描述。一般来说，建议让Vivado HLS自己选择实现资源。如果不能满足要求，设计人员再进行约束。

2.10 位宽优化

C语言提供了许多数据类型来描述各种操作。在这点上，我们已经应用的 `int` 类型，Vivado HLS将其视为一个32位带符号整数。C语言还提供了浮点数据类型，例如 `float` 和 `double`，以及整数数据类型，例如 `char`、`short`、`long` 和 `long long`。整数数据类型可能都是正数。所有这些数据类型位宽都是2的幂次。

这些C语言数据类型实际位宽可能由于处理器架构不同而不同。例如，**int** 类型在微处理上可以是16位，通用处理器上可能是32位。C标准规定最小位宽(例如， **int** 类型至少为16位)和各类型之间关系(例如，**long** 类型不小于 **int** 类型，**int** 类型不小于 **short** 类型)。C99语言标准消除了这种含糊不清规定，并定义多种类型，如**int8_t**、**int16_t**、**int32_t**和**int64_t**。

使用这些不同数据类型首要好处是解决软件中大量数据存储容量问题。对于大型数组，使用8bit而不是16bit可以将内存容量要求减少一半。缺点是可以8bit表示值的范围减少了。有符号8bit数据表示范围为[-128,127]，而有符号16bit数据表示范围是[-32768,32767]。位宽较小数据类型的相关操作可能需要更少的时钟周期，或者可以实现更多指令并行操作。

在FPGA实现中也可以看到相同好处，但这种好处更加明显。由于Vivado HLS工具生成自定义数据路径，它将生成与指定数据类型相匹配的实现结果。例如，语句 `a = b*c` 根据运算数据类型它会有不同的延迟和资源使用情况。如果所有变量都是32位宽，那么需要执行的原始布尔操作要比仅为8位宽的变量多。因此，必须使用更多的FPGA资源(或更复杂的资源)来实现。此外，更复杂的逻辑通常需要更多流水线来实现相同频率。一个32位乘法可能需要5个内部寄存器来达到与使用一个内部寄存器的8位乘法相同的运算频率。因此，操作延迟将更大(5个周期而不是1个周期)，而应用HLS必须考虑到这一点。

创建一个简单设计，实现代码 `a = b*c`。变量类型分别更改为 **char**, **short**, **int**, **long**, **long long**。在每种情况下，相乘运算需要多少个周期？不同的运算类型分别需要多少资源？

8bit数值计算需要什么类型原始布尔操作？当进行32bit数值计算这些需求有什么变化？提示：需要多少原始十进制操作才能实现两个8位十进制数值乘法？

在许多情况下为了优化硬件资源，需要处理位宽不是2的幂次的数据。例如，模数转换器通常输出结果为10bits，12bits，或14bits。我们可以将这些值映射到16bits，但这可能会降低处理性能并增加资源消耗。为了更准确表示这些值，Vivado HLS提供了任意精度数据类型，这些数据类型可以表示为有符号或者无符号任意位宽数据。

有符号和无符号的数据类型有两个单独的类：

- 无符号：`ap_uint`
- 有符号：`ap_int`

`width` 是1到 $1024^{\wedge}1^{\wedge}$ 之间的整数。例如，`ap_int<8>`是一个8位有符号数(和**char**一样)，`ap_uint<32>`是一个32位无符号数(与**unsigned int**相同)。它有一个更强大的数据类型，这种数据类型可以任意设置位宽，例如，`ap_uint<4>`或`ap_int<53>`。要使用这些数据类型，你必须使用c++并且包含头文件 `ap_int.h`，即在你的项目中添加`#include "ap_int.h"`代码，并在添加“.cpp^2^”文件名后缀。

考虑图2.1fir 滤波器代码中滤波器系数数组 `c[]`。在这里复现是为了阅读方便:`coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};`。数据类型 `coef_t` 被定义为 `int`，意思是我们在有32位精度。32位精度对于滤波器系数来说是多余的，因为它们的范围从-91到500。因此，我们可以使用较小的数据类型。由于数值有正有负，所以我们需要一个有符号的数据类型。这11个数据的最大绝对值是500，而 $\log_2(500) = 9$ bits。又因为我们需要负数，我们再加1位。因此，`coef_t` 可以声明为 `ap_int<10>`。

在fir函数中，变量*i*适合的数据类型是什么(见图2.1)？

我们还可以更准确地定义fir函数中其他变量的数据类型，例如acc和shift_reg。首先考虑shift_reg数组。它是存储输入变量x最新更新的11个值，因此我们可知shift_reg值可以安全的拥有与x相同的数据类型，用“安全”的意思是，在精度上没有损失，即，如果shift_reg设成较小位宽的数据类型，那么输入数据X就需要删除一些重要数据位，来满足shift_reg的存储要求。例如，如果x被定义为16位(`ap_uint<16>`)，而shift_reg被定义为12位(`ap_uint<12>`)，那么当我们将x变量数值存储在shift_reg变量中时，我们就会去掉x变量中4个最重要的数据位。

为变量acc定义适当的数据类型是一项更加困难的任务。acc变量存储了shift_reg和滤波器系数数组c[]的乘积加总和。即，滤波器输出值。如果我们希望安全，那么我们将估算可能存储在acc中的最大值，并将数据位宽设为和该值一样。

为实现这一点，我们必须了解在执行算术运算时，位宽如何增加。考虑运算 $a = b + c$ ，其中b为`ap_uint<10>`、c为`ap_uint<10>`，那么变量a的数据类型是什么？在这里我们可以做一个最坏情况的分析，假设a和b都是最大值 $2^{10} - 1 = 1023$ 。二者相加结果为 $a = 2046$ ，可以表示为11位无符号数，即，`ap_uint<11>`。一般来说，在做加法时，运算结果要比两个加数中最大数值的位宽还要多一位。也就是说，当`ap_uint` b和`ap_uint` c相加时，a的数据类型为`ap_uint`，其中 $z = \max(x, y) + 1$ 。这个结论同样适用于有符号数加法。

通过以上方法解决了acc数据类型设计的部分问题，但是我们还必须处理乘法运算。使用相同的术语，我们希望通过x和y的数据位宽来确定数值z的数据位宽(即，`ad_int a, ap_int b, ap_int c`)。对于运算 $a = b * c$ ，我们不再详细介绍，位宽的运算公式为 $z = x + y$ 。

提供以上两个公式来确定acc安全位宽。

最终，我们将acc存储到变量y中，y是函数的输出端口。因此，如果acc的位宽大于变量c的位宽，acc的值将被截断并存储到y。因此，更重要的是确保acc位宽足够大，以便它可以处理完整精度的乘积加操作吗？

问题答案在于应用场景对于精度的要求。在许多数字信号处理应用中，数据本身是有噪声的，这意味着较低的数据位数可能没有意义。此外，在信号处理应用中，我们经常在处理数据时进行数值近似，而导致额外误差的引入。因此，我们确保acc变量精度使其返回一个完全精确的结果可能并不是最重要的。另一方面，为了减少计算中整体的舍入误差，最好在积累过程中保留更多比特，然后再针对最终的结果进行截位。其他应用如科学计算，通常需要更大动态范围，在这种应用下更多使用浮点数而不是整数或定点运算。那么正确的答案是什么？最终取决于设计者对于精度的要求。

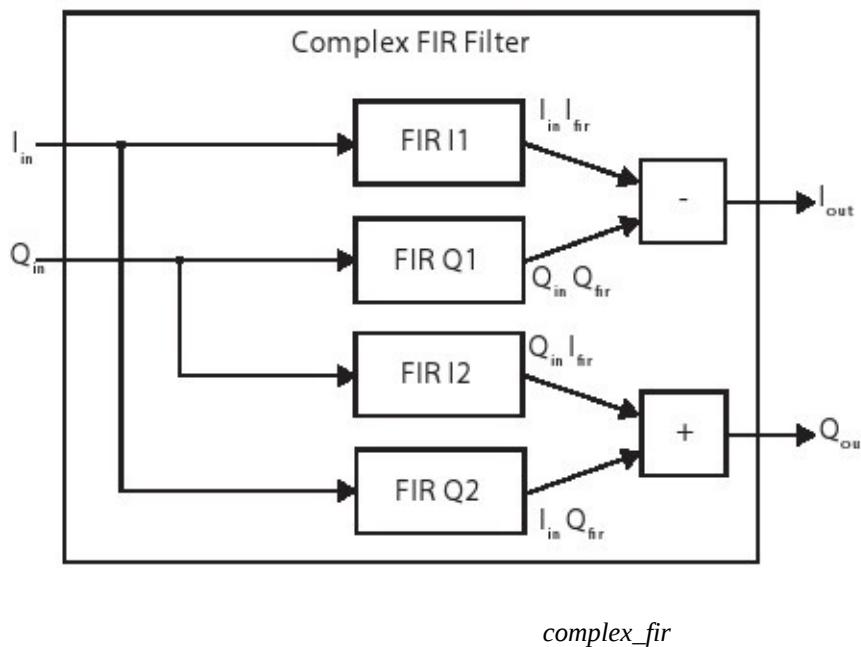
2.11 复数FIR滤波器

到目前为止，我们只研究了实数数字滤波器。许多数字无线通信系统都由实部(I)和虚部(Q)组成的复数数字系统(更多的细节见第三章)。幸运的是，我们可以使用实数FIR滤波器来创建复数FIR滤波器，如下所述。

考虑方程2.4来理解如何利用实数FIR滤波器建立复数FIR滤波器。假设(I_{in}, Q_{in})是我们希望输入滤波器的一个数据。其中复数FIR滤波器系数表示为(I_{fir}, Q_{fir})。处理系统中将会有不止一个输入样本和复数滤波器系数，但我们现在不用担心这点。

$$(I_{in} + jQ_{in})(I_{fir} + jQ_{fir}) = (I_{in}I_{fir}Q_{in}Q_{fir}) + j(Q_{in}I_{fir} + I_{in}Q_{fir}) \quad (2.4)$$

方程2.4显示了复数FIR滤波器的一个系数与输入复数数据的乘法。方程右侧显示复数FIR滤波器输出实数部分是 $I_{in}I_{fir} - Q_{in}Q_{fir}$ ，和虚数部分为 $Q_{in}I_{fir} + I_{in}Q_{fir}$ 。这意味着我们可以将复数FIR过滤器运算拆分为四个实数滤波器，如图2.9所示。



complex_fir

图2.9:一个复数FIR滤波器由4个实数FIR滤波器组成。输入的I和Q样本被分配到4个不同的实数FIR滤波器中。FIR滤波器存储实部(FIR I)和虚部(FIR Q)复数系数。

```

typedef int data_t;
void firI1(data_t *y, data_t x);
void fitQ1(data_t *y, data_t x);
void firI2(data_t *y, data_t x);
void fitQ2(data_t *y, data_t x);

void complexFIR(data_t Iin, data_t Qin, data_t *Iout, data_t *Qout){
    data_t IinIfir, QinQfir, QinIfir, IinQfir;

    firI1(&IinIfir, Iin);
    firQ1(&QinQfir, Qin);
    firI2(&QinIfir, Qin);
    firQ2(&IinQfir, Iin);
    * Iout = IinIfir + QinQfir;
    * Qout = QinIfir - IinQfir;
}

```

图2.10:Vivado HLS使用四个实数FIR滤波器分层实现一个复数FIR滤波器。

复数FIR滤波器输入数值是复数(I_{in}, Q_{in})并输出复数的过滤结果(I_{out}, Q_{out})。图2.9描述了复数滤波器的原理框图，复数滤波器使用了四个实数FIR滤波器(FIR I1, FIR Q1, FIR I2, FIR Q2)。滤波器FIR I1和FIR I2是等效的，即它们的系数相同。FIR Q1和FIR Q2也是相同的。每个滤波器的输出对应于一个等式2.4的因子。然后对这些输出进行加法或者减法，来得到最终滤波的复数输出结果(I_{out}, Q_{out})。

我们使用层次化结构来定义复数FIR滤波器。Vivado HLS使用函数来实现层次化结构。使用前面的实数FIR函数 **void fir(data_t * y, data_t x)**，我们可以创建封装了实数fir函数的复数FIR滤波器。此代码如图2.10所示。

该代码定义了四个函数firI1、firQ1、firI2和firQ2。每个函数都是完全相同的代码，即图2.1中的fir函数。通常，我们不需要复制函数；而是简单地调用同一个函数四次。然而，这在本例中是不可以的，因为fir函数中shift_reg变量使用了 **static** 关键字。

函数调用充当接口。Vivado HLS工具不能跨越函数边界进行优化。也就是说，每个fir函数都被独立综合，并且在complexFIR函数中或多或少的被当作一个黑盒来处理。如果你希望Vivado HLS工具针对某个特定函数在其父函数中共同与其它代码共同优化，你可以使用**inline**指令。这个指令将该函数代码添加到父函数中，并消除层次结构。虽然这有提高性能和资源面积优化，但是它也产生了需要工具综合的大量代码。代码综合可能需要很长时间，甚至会综合失败，或者导致不可优化设计。因此，要小心使用内联指令。还要注意，Vivado HLS工具可以自动进行函数内联。这些函数通常具有少量代码。

```
float mul(int x,int y){
    return x * y;
}
float top_function(float a,float b,float c,){
    return mul(a,b) + mul(c,d) + mul(b,c) + mul(a,d);
}
float inlined_top_function(float a,float b,float c,float d){
    return a * b + c * d + b * c + a * d;
}
```

图2.11:一个简单的例子说明**inline**指令。`top_function`有四个函数调用`mul`函数。如果我们在`mul`函数中添加**inline**指令，其结果与`inlined_top_function`函数结果类似。

`inline`指令去除了函数边界，这使Vivado HLS对代码可以进行额外优化，但这种额外优化方式可能带来复杂的综合问题，例如，它将使编译时间变长。同时它在执行函数调用时将排除顶层关联。它有多种方式维持代码结构，同时使层次结构更加易读。图2.11所示代码提供了一个**inline**指令如何工作的示例，`inlined_top_function`函数是应用`mul`函数应用**inline**指令的结果。

Vivado HLS在某些情况下自动应用**inline**指令。例如，图2.11由于`mul`函数很小，工具很有可能对该函数进行**inline**操作。你可以通过**inline**指令的**off**参数来强制工具保持函数层次结构。`inline`指令也有递归参数即在**inline**函数中，其调用的子函数也**inline**处理。也就是说所有子函数都会把代码展开到母函数中。这个可能会导致代码膨胀，因此这个功能要谨慎使用。一个**inline**函数不会有单独报告，因为所有的逻辑已经融合到了母函数中。

2.12 总结

本章描述了使用Vivado HLS工具进行FIR滤波器的设计和优化。文章目标是提供HLS处理过程概况。处理的第一步是理解FIR滤波器计算背后的基本概念。这并不需要深刻的数学理解，但肯定要有足够的知识可以编写Vivado HLS工具能够综合的代码。这可能需要进行不同语言(例如，MATLAB、Java、C++、Python等)的转换。很多时候为了应用更简单的数据结构程序代码需要重构。例如，明确指明数据资源类型。同时这经常需要删除系统调用和HLS工具不支持代码。

创建一个优化的体系结构需要对HLS工具如何进行综合以及对RTL代码优化过程有基本理解。当然，精确地理解时序表、绑定、资源分配等等HLS算法是没有必要的(而且很多时候这些都是有专利的)。但是，对这个过程有一个大致了解，可以帮助设计师编写出与硬件匹配更好的代码。纵观整章，我们讨论了HLS综合过程的一些关键特性，这些特性是进行各种优化必须要理解的。尤其重要的是要理解HLS性能报告，关于性能报告我们在第2.4章中已经对此进行了描述。

此外，我们还介绍了一些基本HLS优化方法(包括循环和位宽优化)。我们以FIR滤波器设计为例强调了它们的优点和潜在缺点。这些都是常见的优化，可以广泛应用于各种场景。在接下来的章节，我们通过讨论其他更复杂的应用场景，来介绍更多关于这些优化的细节。

第三章 CORDIC

3.1 概述

CORDIC(坐标旋转数字算法)是一种计算三角、双曲和其他数学函数的有效方法。它是一种数字算法，每次运算均产生一次结果输出。这使我们能够根据应用需求调整算法精度；增加运算迭代次数可以得到更精确的结果。运算精度与运算性能和占用资源并列，是一种通用的设计评估指标。CORDIC是只使用加法、减法、移位和查找表实现的简单算法，这种算法在FPGA中实现效率高，在硬件算法实现中经常用到。

CORDIC算法是1950年由Jack Volder发明，它最开始是作为数字解决方案替代模方案应用于B-58轰炸机实时导航上，它的功能是计算旋转角度。在那个时代用硬件实现乘法的成本是相当高的，同时CPUs的计算能力也非常有限。因此这个算法需要有低的运算复杂度和使用简单的运算操作。多年之后，它被应用于数学协处理器[24]、线性系统[3]、雷达信号处理[4]、傅立叶变换[21]和其它数字信号处理算法中。现在，它广泛应用于FPGA设计中。Vivado HLS用CORDIC进行三角函数计算，同时CORDIC也是现代FPGA IP CORE库中的标准运算模块。

本章目标是演示如何使用高级语言创建优化CORDIC算法。随着本书的深入，我们研究设计的硬核复杂性也在逐渐增加。CORDIC算法是一种迭代算法；因此，大多数计算都在一个for循环中执行。代码本身并不复杂，但是用来创建最优硬件实现结构是需要设计人员对代码有深入理解的。一个优秀的HLS设计人员如果希望创建最优设计，就必须理解算法。因此，我们在本章前部分给出了CORDIC算法的数学和计算背景。

我们在本章强调的HLS主要优化方式是为变量选择正确的数据表示。正如我们在本章后面所讨论的，设计人员必须仔细权衡运算结果精度、性能和设计的资源利用率。数据表示是这种权衡的一个重要因素——“较大”数字（那些大位宽数据），通常是以增加使用资源(更多的寄存器和逻辑块)和降低处理性能为代价来提供更精确数据。我们将在第3.5.5节中提供了关于数据表示和任意精度数据类型的背景。

本章与工程应用相结合，针对运算精度(计算的准确性)、资源占用和处理性能的权衡进行了更深入的实验。本章目的是提供足够的知识经验，以便人们可以在这个项目中进行练习，例如，这一章和那个工程是相互补充的。那个工程的目标是建立一个相位探测器，它使用了一个CORDIC和一个复数匹配滤波器，这个滤波器我们在前一章中已经介绍过它了。

3.2 背景

CORDIC核心思想是在一个二维平面上高效地执行一组矢量旋转。在这些旋转运算的基础上增加一些简单控制，我们就可以实现各种基础操作，例如，三角函数，双曲函数，对数函数，实乘和复乘，以及矩阵分解和因式分解。CORDIC已经广泛应用于信号处理、机器人技术、通信和许多科学计算领域。由于CORDIC占用资源少，所以常用在FPGA设计中。

在下文中，我们将介绍CORDIC如何执行给定输入角 θ 的正弦和余弦的过程。这是使用一系列矢量旋转来完成的，这些简单的操作在硬件中使用非常有效。在高层次，算法使用一系列旋转来工作，目标是达到目标输入角 θ 。实现这种效率的关键创新是旋转可以以需要最少计算的方式完成。尤其我们使用乘以2的常数幂来执行旋转。这意味着简单地在硬件中移动位是非常有效的，因为它不需要任何逻辑。图3.1提供了用于计算 $\cos\phi$ 和 $\sin\phi$ 的CORDIC程序的高级概述。在这种情况下，我们在x轴上开始初始旋转矢量，即 0° 角。然后，我们执行一系列迭代旋转；在这个例子中，我们只执行四次旋转，但通常这是40次旋转。每个后续旋

转使用越来越小的角度，这意味着每次迭代都会为输出值增加更多精度。在每次迭代中，我们决定以较小的角度进行正向或负向旋转。我们旋转的角度值是先验固定的；因此，我们可以轻松地将它们的值存储在一个小内存中，并保持我们到目前为止已经旋转的累积角度的运行总和。如果该累积角度大于我们的目标角度 φ ，则我们执行负旋转。如果它更小，那么旋转就是正的。一旦我们完成了足够数量的旋转，我们就可以通过直接读取最终旋转矢量的x和y值来确定 $\cos\varphi$ 和 $\sin\varphi$ 。如果我们的最终向量的幅度为1，则 $x = \cos\varphi$ 且 $y = \sin\varphi$ 。

我们从一些术语开始，目的是重新定义你的一些基本的三角函数和矢量概念。如果熟悉的话就不必看了。但请记住，创建高效硬件设计最重要的一个方面是真正理解应用程序；只有这样，设计师才能有效地利用优化指令并执行代码重构，这是获得最有效设计所必需的。

CORDIC算法的基本目标是以有效的方式执行一系列旋转。让我们首先考虑如何进行旋转。在二维中，旋转矩阵是：

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (3.1)$$

CORDIC使用迭代算法将矢量 v 旋转到某个角度目标，这取决于CORDIC正在执行的功能。一次旋转是 $v_i = R_i * v_{i-1}$ 形式的矩阵向量乘法。因此，在CORDIC的每次迭代中，我们执行以下操作来执行一次旋转，即矩阵向量乘法：

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad (3.2)$$

写出线性方程，新旋转矢量的坐标是：

$$x_i = x_{i-1} \cos \theta - y_{i-1} \sin \theta \quad (3.3)$$

和

$$y_i = x_{i-1} \sin \theta + y_{i-1} \cos \theta \quad (3.4)$$

这正是我们需要简化的操作。我们想要执行这些旋转而不执行任何乘法。首先考虑 90° 旋转。在这种情况下，旋转矩阵是：

$$R(90^\circ) = \begin{bmatrix} \cos 90^\circ & -\sin 90^\circ \\ \sin 90^\circ & \cos 90^\circ \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad (3.5)$$

因此我们只需要执行操作：

$$\begin{aligned} x_i &= x_{i-1} \cos 90^\circ - y_{i-1} \sin 90^\circ \\ &= x_{i-1} \cdot 0 - y_{i-1} \cdot 1 \\ &= -y_{i-1} \end{aligned} \quad (3.6)$$

和

$$\begin{aligned} y_i &= x_{i-1} \sin 90^\circ + y_{i-1} \cos 90^\circ \\ &= x_{i-1} \cdot 1 + y_{i-1} \cdot 0 \\ &= x_{i-1} \end{aligned} \quad (3.7)$$

把这放在一起，我们可以得到

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -y \\ x \end{bmatrix} \quad (3.8)$$

可以看到这需要的计算非常少;旋转矢量简单地使y值无效，然后交换x和y值。二进制补码无效需要等效于加法器的硬件。因此，我们实现了有效执行90°旋转的目标。

如果你想在-90°之间旋转怎么办？什么是旋转矩阵R (-90°)？此u旋转需要什么类型的计算？如何设计能够通过-90°执行正负旋转的最有效电路，例如旋转方向是电路的输入？

虽然我们可以旋转±90°，但如果我们希望在旋转到目标角度时都有好的分辨率，我们需要旋转更小的角度。也许我们可能希望旋转的下一个自然角度是±45°。使用公式3.1中的旋转矩阵得到

$$R(45^\circ) = \begin{bmatrix} \cos 45^\circ & -\sin 45^\circ \\ \sin 45^\circ & \cos 45^\circ \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \quad (3.9)$$

计算旋转操作数值，我们可以得到

$$x_i = x_{i-1} \cos 45^\circ - y_{i-1} \sin 45^\circ = x_{i-1} \cdot \sqrt{2}/2 - y_{i-1} \cdot \sqrt{2}/2 \quad (3.10)$$

和

$$y_i = x_{i-1} \sin 45^\circ + y_{i-1} \cos 45^\circ = x_{i-1} \cdot \sqrt{2}/2 + y_{i-1} \cdot \sqrt{2}/2 \quad (3.11)$$

把计算结果代入矩阵向量运算

$$\begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2x - \sqrt{2}/2y \\ \sqrt{2}/2x + \sqrt{2}/2y \end{bmatrix} \quad (3.12)$$

与旋转±90°相比，这肯定不如计算效率高。±90°旋转是理想的，因为乘法运算的倍数是非常简单的常数（在这种情况下为0,1和-1）。CORDIC的关键是以有效的方式进行这些旋转，例如用容易的乘法计算的方式定义旋转矩阵。也就是说，我们希望更像前一个±90°，而不像我们刚刚描述的±45°旋转那样更难计算。如果我们“强制”旋转矩阵成为易于乘法的常数怎么办？例如，乘以任意两次幂变为换档操作。如果我们将旋转矩阵中的常量设置为2的幂，我们可以非常容易地执行旋转而不需要乘法。这是CORDIC背后的理念 - 找到非常有效的旋转计算，同时最大限度地减少任何副作用。我们将更详细地讨论这些“副作用”，但这里有一个工程决策。为了获得有效的计算，我们必须放弃一些东西；在这种情况下，我们必须处理这种情况，即旋转也会执行缩放，即它会改变旋转矢量的大小 - 稍后会更多。为了进一步探索“简单”旋转矩阵，请考虑矩阵

$$R() = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \quad (3.13)$$

转换成相应的计算形式

$$x_i = x_{i-1} - y_{i-1} \quad (3.14)$$

和

$$y_i = x_{i-1} + y_{i-1} \quad (3.15)$$

用矩阵向量的形式表示

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x - y \\ x + y \end{bmatrix} \quad (3.16)$$

这个很容易计算，而且不需要任何“困难”的乘法。但这次运算结果是什么呢？结果证明这次运算实现了完美的45度旋转；现在，我们得到了一个高效方式来实现一次45度旋转。但是，这个变换也把矢量以 $\sqrt{2}$ 进行了量化。这个矩阵行列式的平方根表明变换量化矢量的大小，即，矢量长度是如何变化的。这里矩阵行列式为 $1 - (-1) = 2$ 。因此，这个操作实现角度旋转45度和尺度变化 $\sqrt{2}$ 倍。这是CORDIC运算进行的折中；我们可以使旋转运算变得容易，但它的副作用是缩放矢量的长度。根据应用场景不同，这不一定是个问题。但是现在，我们暂时不考虑缩放问题，集中讨论如何推广高效旋转操作。

现在我们介绍高效矩阵旋转概念，即，只进行加/减和2的幂次乘法运算(即移位操作)。再考虑旋转矩阵

$$R_i(\theta) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \quad (3.17)$$

通过使用下面的三角恒等式

$$\cos(\theta_i) = \frac{1}{\sqrt{1 + \tan^2(\theta_i)}} \quad (3.18)$$

$$\sin(\theta_i) = \frac{\tan(\theta_i)}{\sqrt{1 + \tan^2(\theta_i)}} \quad (3.19)$$

我们可以将旋转矩阵变为

$$R_i = \frac{1}{\sqrt{1 + \tan^2(\theta_i)}} \begin{bmatrix} 1 & \tan(\theta_i) \\ \tan(\theta_i) & 1 \end{bmatrix} \quad (3.20)$$

如果我们限制 $\tan(\theta_i)$ 的值是2的幂次，那么旋转运算可以简化为数据移位（乘法）和加法。具体为，我们设 $\tan(\theta_i) = 2^{-i}$ 。旋转矩阵就变成了

$$v_i = K_i \begin{bmatrix} 1 & -2^{-i} \\ 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \quad (3.21)$$

其中

$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (3.22)$$

这里有几点需要注意。 2^{-i} 相当于数据向右移动*i*位，即，等效于2的幂次除法。这基本上可以等效为一个简单的不需要任何资源的结构，即，在硬件实现上，它基本上是“无消耗”的。这是一个巨大的优点，但它也存在一些缺点。首先，我们受限只能旋转角度 θ ，其中 $\tan(\theta_i) = 2^{-i}$ 。后续我们将证明这不是什么严重问题。第二，我们只展示了一个方向的旋转；而CORDIC要求能够旋转 $\pm\theta$ 。这个可以通过添加 σ 值（1或-1）来表示正向或者逆向旋转来修正这个错误。我们可能在每次迭代/旋转中有不同的 σ_i 。因此旋转操作可概括为

$$v_i = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} \quad (3.23)$$

最后，旋转矩阵需要乘以 k_i ，在迭代过程中 k_i 通常被省略，然后在一系列旋转完成后进行补偿。比例因子累积为

$$K(n) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (3.24)$$

和

$$K = \lim_{n \rightarrow \infty} K(n) \approx 0.6072529350088812561694 \quad (3.25)$$

不同迭代的比例因子可以预先计算并存储。如果我们总是做固定次数的旋转，这个比例因子就是一个常数。这种修正也可以在旋转之前适当地缩放 v_o 来提前进行。有些情况下可以忽略这个比例因子，但这会导致处理增益。

$$A = \frac{1}{K} = \lim_{n \rightarrow \infty} \prod_{i=0}^{n-1} \sqrt{\frac{1}{1+2^{-2i}}} \approx 1.64676025812107 \quad (3.26)$$

在每次迭代中，我们需要知道刚刚执行的旋转角 θ_i 。其中 $\theta_i = \arctan 2^{-i}$ 。我们可以提前计算每一个 i 对应的 θ_i 值，然后把它们存储在片上内存中，之后我们可以像用查找表一样用这些值。此外，我们还有一个决定是顺时针还是逆时针旋转的机制，即，我们必须确定 σ 的值为1还是-1。这个决定取决于所需CORDIC的模式。例如，计算 $\cos\theta$ 和 $\sin\theta$ ，我们保存所有旋转角度的累加和。我们比较这个值和目标角 θ 的大小，如果当前累加和的角度小于 θ 则执行正向旋转，如果当前累加和大于 θ 则我们进行逆向旋转。

表3.1提供了CORDIC前7次迭代的统计信息。第一行是“零”旋转(即 $i = 0$)，这是一次45度旋转。它的比例因子为1.41421。第二行旋转因子为 $2^{-1} = 0.5$ 。这个结果的旋转角度为 $\theta = \arctan(2^{-1}) = 26.565^\circ$ ，这个旋转的比例因子为1.11803。CORDIC增益是所有比例因子的积。在这个例子中，它的比例因子是前两个比例因子之积，即， $1.58114 = 1.41421 * 1.11803$ 。这个过程的数值在增加，而旋转角度和比例因子越来越小。值得注意的是CORDIC比例因子最终趋于稳定 ≈ 1.64676025812107 ，数值正如公式3.26。另外，请注意，当旋转角度变小时，它们对数据精度的影响也逐步减弱。

描述第*i*次迭代对结果精度的影响？也就是说，它改变了哪几位？更多的迭代运算如何改变最终结果的精度，例如。当CORDIC迭代次数增加时 $\sin\theta$ 和 $\cos\theta$ 的值发生怎样的变化？

表3.1:CORDIC前7次迭代的旋转角度、比例因子和CORDIC增益。注意，角度每次迭代大约减小一半。比例因子表示在旋转过程中旋转矢量增加的长度。CORDIC增益是所有旋转矢量比例因子的累积，例如，某一次迭代的CORDIC增益是本次迭代和与该次迭代之前所有比例因子累积。

i	2^{-i}	Rotating Angle	Scaling Factor	CORDIC Gain
0	1.0	45.000°	1.41421	1.41421
1	0.5	26.565°	1.11803	1.58114
2	0.25	14.036°	1.03078	1.62980
3	0.125	7.125°	1.00778	1.64248
4	0.0625	3.576°	1.00195	1.64569
5	0.03125	1.790°	1.00049	1.64649
6	0.015625	0.895°	1.00012	1.64669

3.3 计算正弦和余弦

现在我们可以更精确地使用CORDIC计算一些给定角 θ 的正弦和余弦值。为了计算正弦和余弦值，我们从x轴正方向上的一个矢量开始(例如，初始角度45度)，然后执行一系列旋转直到我们逼近给定角 θ 。之后，我们可以直接读取旋转矢量的x和y值，这两个值即为对应 $\cos\theta$ 和 $\sin\theta$ 。这里假设最终矢量幅度等于1，你会看到计算正余弦并不难实现。

让我们用一个示例来具体说明：计算 $\sin 60^\circ$ 和 $\cos 60^\circ$ ，即， $\theta = 60^\circ$ 。这个过程如图3.2所示。在这个例子中我们执行了五次旋转得到一个角度近似为 60° 。我们初始矢量为0度即它从x轴正半轴开始。第一次旋转对应序号*i* = 0旋转了45度角(见表3.1)。由于我们想要得到 60° ,所以我们沿正方向旋转。矢量旋转后得到 45° 角;同时还要注意，它的幅度约为1.414。现在，我们继续序号*i* = 1的旋转。因为我们希望得到 60° 角,所以我们沿正方向旋转。矢量旋转之后的结果角度为 $45^\circ + 26.565^\circ = 71.565^\circ$,比例因子为1.118;

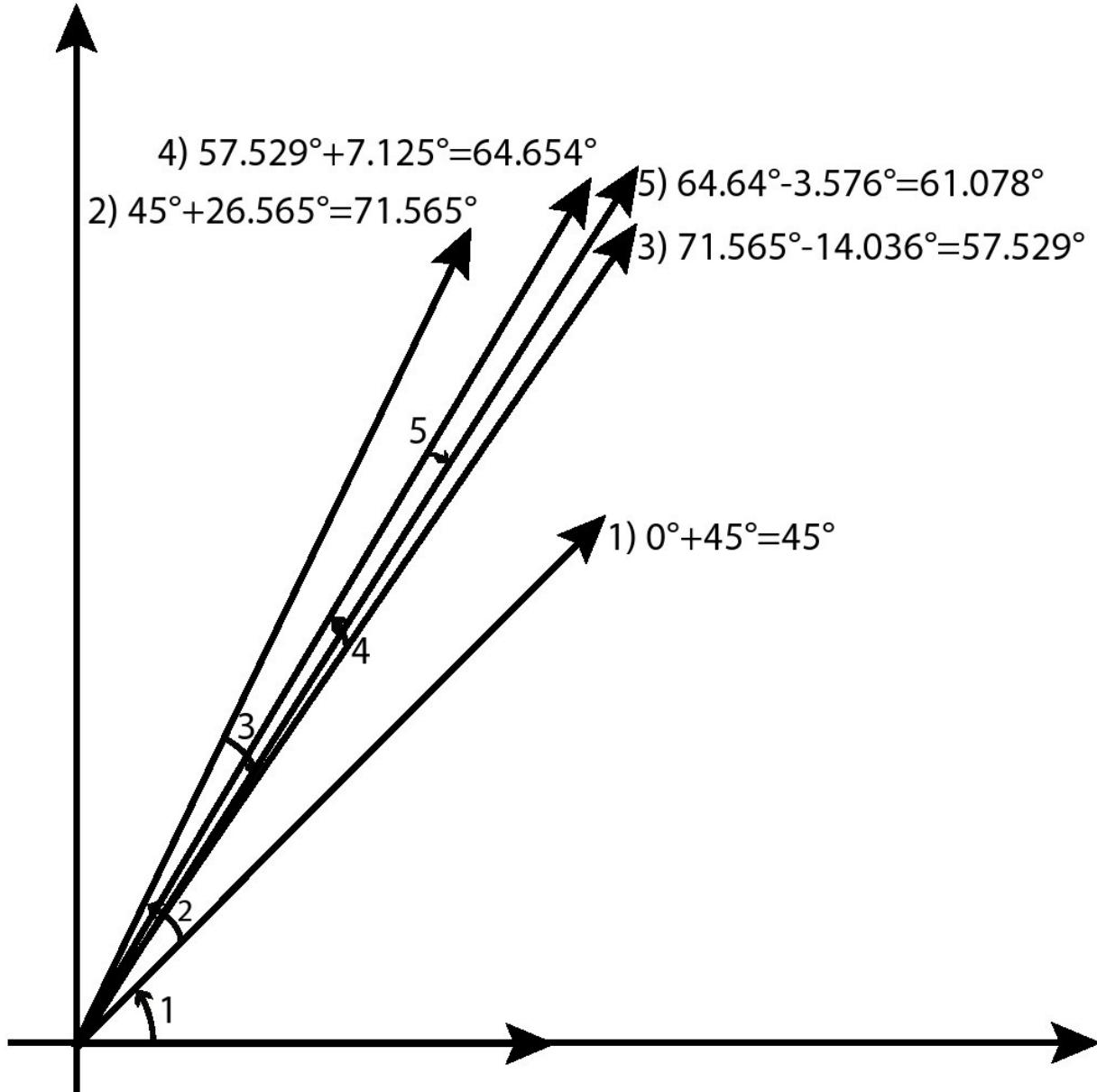


图3.2用CORDIC算法计算 $\cos 60^\circ$ 和 $\sin 60^\circ$ 。使用递增数*i*(0, 1, 2, 3, 4)来表示执行五次旋转，最终旋转结果为61.078度。这个矢量对应的x和y值可以近似为指定角度的余弦和正弦值。

两次旋转得到的比例因子为 $1.414 \times 1.118 = 1.581$ ，这也是CORDIC增益。继续讨论 $i=2$,现在我们得到的角大于 60° 的目标角,所以我们要沿负方向旋转，旋转后矢量角度为 57.592° 比例因子为1.630。整个过程伴随*i*值不断增大，同时旋转角度越来越小，最终会近似达到期望角度。同样，请注意随着旋转次数的增加，CORDIC增益逐渐趋于稳定。

当我们做了足够多的旋转后，数据精度将会满足我们的要求，最后我们得到一个与期望输入角近似的矢量。这个矢量的x和y值，对应 $A_R \cos 60^\circ$ 和 $A_R \sin 60^\circ$,如果 $A_R = 1$, 那么这个x与y的值正是我们想要得到的。由于我们通常知道将要执行的旋转次数，所以我们可以通过将预设初始矢量的大小为CORDIC增益的倒数来确保 $A_R = 1$, 在我们的例子中,假设执行了五次旋转,如图3.2,可知需要设置初始矢量的值为 $1.64649^{-1} = 0.60735$ (当*i*=5时CORDIC增益的倒数;见表3.1)。由此，我们可以直接设置一个初始矢量为(0.60735, 0)。

如果我们再多做一次旋转，结果会变成怎样？再多做两次旋转（三次，四次，等等）会怎么样？当我们执行更多的旋转，精度会变成多少（例如，与MATLAB运算相比）？在一般情况下，你认为几次旋转是适合的？

做更多的旋转有没有可能会使精度变得更差？提供一个发生这种情况的例子。

图3.3提供了使用CORDIC算法实现正弦和余弦值计算的代码。它将输入角作为目标角，输出这个角对应的正弦和余弦值。代码使用数组cordic_phase作为查找表，这个查找表存储每次迭代的旋转角度。这个角度对应于表3.1中的“旋转角度”列中的值。我们假设cordic.h文件定义不同的数据类型（例如，COS_SIN_TYPE和THETA_TYPE）并设置NUM_ITERATIONS为某个常数。数据类型可以更改为不同的定点或浮点类型，设置NUM_ITERATIONS值要同时考虑我们期望的精度、资源和吞吐量。

注意变量sigma被设置为一个二位宽整数。因为我们知道它的可能取值只有 ± 1 ，我们可以优化它的数据类型使它比使用int类型有更小的资源占用和更佳的性能。我们稍后将讨论如何在Vivado HLS中指定它们的数据类型。

此代码接近于“软件”版本。它有多种方式来提高其性能并减小资源面积。我们将在本章后面讨论如何优化这段代码。

3.4 笛卡尔向极坐标转换

通过一些修改，CORDIC可以实现其它功能。例如，它可以实现笛卡尔和极坐标系转换；我们将在本节详细地描述这一点。CORDIC还可以做其他很多功能，我们把它留给读者作为练习。

一个二维矢量v可以使用笛卡儿坐标系统(x, y)或极坐标系统(r, θ)来表示，对于极坐标系 r 是半径坐标(矢量的长度)和 θ 是角度坐标。这两种坐标系都有优缺点。例如，如果我们想做一个旋转，那么极坐标形式更容易实现，而笛卡尔坐标系更适合描述线性变换。

两种坐标系之间的转换关系如下式所示：

$$x = r \cos \theta \quad (3.27)$$

$$y = r \sin \theta \quad (3.28)$$

$$r = \sqrt{x^2 + y^2} \quad (3.29)$$

$$\theta = \text{atan2}(y, x) \quad (3.30)$$

atan2在arctan函数中定义为

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases}$$

```

//The file cordic.h holds definitions for the data types and constant values

#include "cordic.h"

//The cordic_phase array holds the angle for the current rotation
THETA_TYPE cordic_phase[NUM_ITERATIONS] = {
    45, 25.56, 14.036, 7.125
    3.576, 1.790, 0.895, ...
};

void
cordic(THETA_TYPE theta, COS_SIN_TYPE &s, COS_SIN_TYPE &c)
{
    //Set the initial vector that we will rotate
    //current_cos = I;current = Q
    COS_SIN_TYPE current_cos = 0.60735;
    COS_SIN_TYPE current_sin = 0.0;

    //Factor is the 2^(-L) value
    COS_SIN_TYPE factor = 1.0;

    //This loop iteratively rotates the initial vector to find the
    //sine and cosine value corresponding to the input theta angle
    for(int j = 0; j < NUM_ITERATIONS; j++){
        //Determine if we are rotating by a positive or negative angle
        int sigma = (theta < 0) ? -1 : 1;

        //Save the current_cos, so that it can be used in the sine calculation
        COS_SIN_TYPE temp_cos = current_cos;

        //Perform the rotation
        current_cos = current_cos - current_sin * sigma * factor;
        current_sin = temp_cos * sigma * factor + current_sin;

        //Determine the new theta
        theta = theta - sigma * cordic_phase[j];

        //Calculate next 2^(-L) value
        factor = factor >> 1;
    }

    //Set the final sine and cosine values
    s = current_sin;c = current_cos;
}

```

图3.3:CORDIC代码实现计算给定角度的正弦和余弦值。

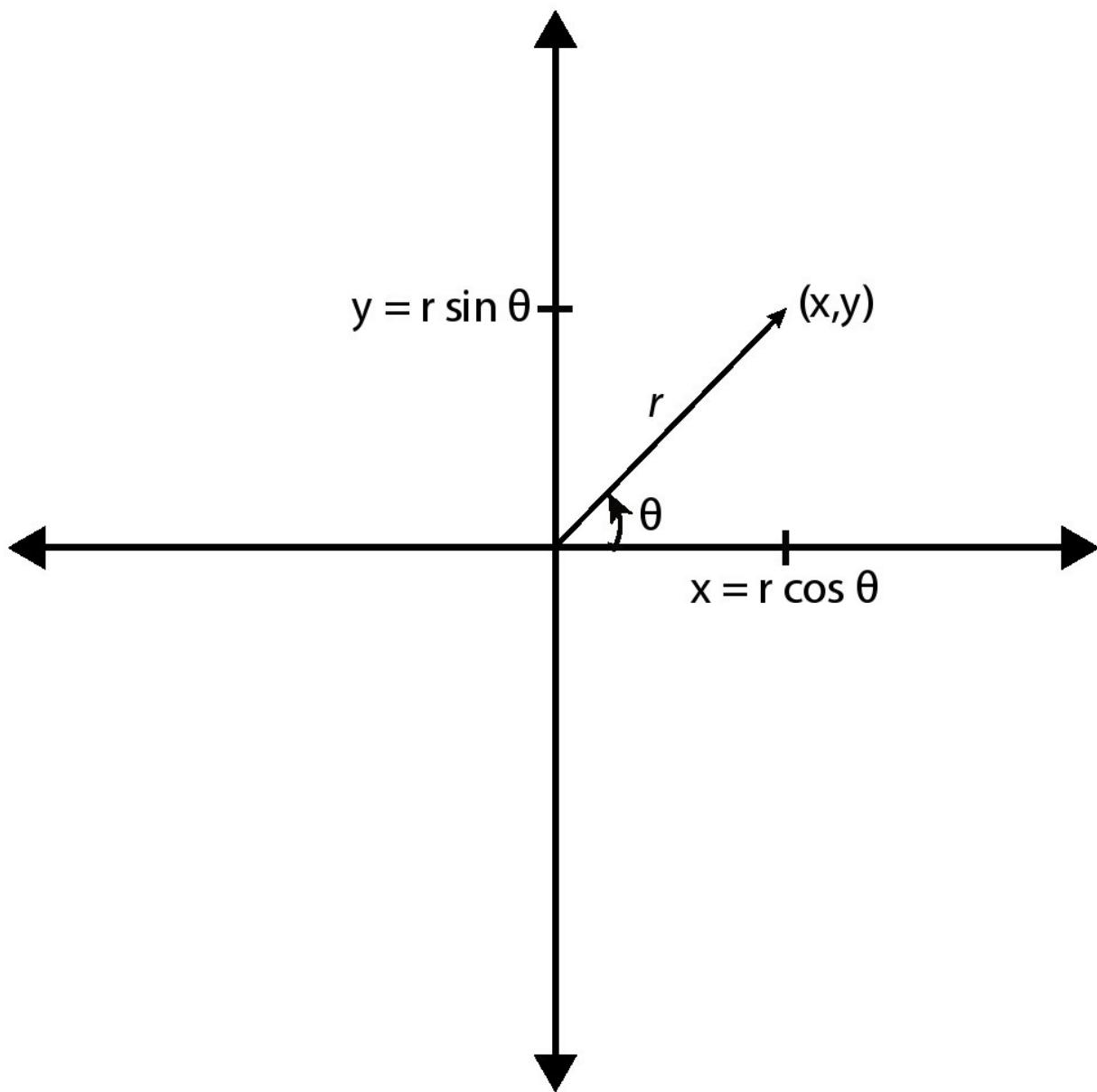


图3.4:图中显示了一个二维平面和一个同时用笛卡尔形式 (x,y) 和极坐标形式 (r,θ) 表示的矢量,通过该矢量说明这两个坐标系之间的关系。

这里提供了一种在两个坐标系之间转换的方法。然而，这些操作在硬件中并不容易实现。例如， \sin ， \cos ，平方根和 \arctan 都不是简单操作，它们需要大量的运算资源。但是我们可以使用CORDIC算法通过一系列简单的迭代旋转来实现这些操作。

给定一个用笛卡尔坐标系表示的数 (x, y) ，我们使用CORDIC算法可以计算它的角度和幅度坐标(即，将它转换成极坐标形式)。为实现这一点，我们将给定的笛卡尔坐标系下的数据旋转为0度。一旦这个旋转完成，幅度就是最终旋转矢量的x值。要确定角度坐标，我们只需关注CORDIC执行旋转的累计角度。矢量旋转($i=0$ 第一次旋转、 $i=1$ 第二次旋转、 $i=3$ 第三次旋转……)的角度可以存储在查找表中，用来计算 \sin / \cos 。因此，我们只需要通过对这些角进行加减就可以得到总的旋转角度，而这里的加减操作取决于旋转方向。

该算法与计算给定角的正弦和余弦相似。我们执行一系列带有递增值*i*的旋转，使最终的矢量位于(靠近)x的正半轴上(即0度)。这可以用正旋转或负旋转来实现，正负旋转取决于矢量的y值，而这个矢量的幅度和角度值都是我们希望确定的。

算法执行第一步旋转得到的初始矢量在象限I或者IV。旋转 90° 的正负取决于初始矢量的y值。如果y值是正数，我们可以推算出我们在第I或者第II象限。反向旋转 -90° 将使我们进入IV或者I象限。一旦我们进入这些象限中，我们可以保证最终能够渐近的接近目标角度0度。如果在象限III或象限IV，初始矢量y值将是负的。我们要做一个 90° 正向旋转，让我们进入IV和I象限。回想一下， $\pm 90^\circ$ 的旋转是通过将矢量的x和y值求反，然后将两个值互换实现。关于旋转 $\pm 90^\circ$ 的情况在图3.5中说明。

这里存在旋转矢量最终径向数值的问题；它的大小与旋转前初始大小不同；它受CORDIC增益的影响。当然，也可以通过乘以对应CORDIC增益的倒数(例如， $1/1.647 = 0.607$)来精确计算矢量径向值。然而，这违背了设计CORDIC算法避免乘法运算的目的。不幸的是，这种乘法不能简单地使用移位和加法来实现。幸运的是，这个因素通常并不重要。例如，在无线通信中用于调制的调幅键控中，你只需要知道相对幅度。或者在其他应用中，这个振幅增益可以由系统的其他部分来补偿掉。

3.5 数字表示

cordic函数使用当前常用的变量类型。例如，变量sigma被定义为int类型，其他变量使用自定义数据类型(例如，THETA_TYPE和COS_SIN_TYPE)。在许多情况下，HLS工具能够进一步优化这些值，来实现硬件资源优化。举个例子，在图3.3中，变量sigma的值为1或-1，即使将变量声明为至少包含32位位宽的int类型，在不改变程序功能的情况下，只有很少的位数用来实现变量。在其它情况下，特别是变量经常出现在函数输入、存储和递归中，HLS工具不能针对变量进行自动优化。在这些情况下，修改代码使用较小的数据类型是避免资源浪费的重要优化手段。

虽然减少变量位宽通常是一个好的优化方法，但是这种优化方法可能改变程序的行为。一个小位宽数据类型不能像大位宽数据类型那样表征大量信息，而且无限的二进制数可以表示所有无限精度的实数。幸运的是，作为设计人员，我们可以选择符合特定需求的精度，并且可以在精度、资源占用和性能之间进行权衡。

在进一步讨论cordic函数数值表示方法之前，我们首先介绍数值表示的背景知识。我们提供基础知识，因为这在理解Vivado HLS所提供的数据类型进行数值表示时非常重要。下一节将从数值表示的基本背景开始讲解，然后讨论Vivado HLS中可用的任意精度变量。

3.5.1 二进制和十六进制数

计算机和fpga通常使用二进制来表示数据，这使数据可以由开关信号组成的二进制数字或简单的位来有效地表示。二进制数在大多数情况下都像普通的十进制数一样工作，但是如果你不熟悉它们的工作原理，它们常常会导致混淆错误，在许多嵌入式系统和fpga中尤其如此，在这些系统中，最小化数据位宽可以极大地提高系统总体性能和效率。在本节中，我们将总结二进制算法以及计算机表示数字的基本方法。

许多读者可能熟悉这些知识，如果是这样的话，你可以略读这些部分或完全跳过它们。我们建议阅读第3.5.5节，因为那里针对Vivado HLS说明了怎样定义任意的数据类型，这是一个用来进行CORDIC数值优化的重要方法。

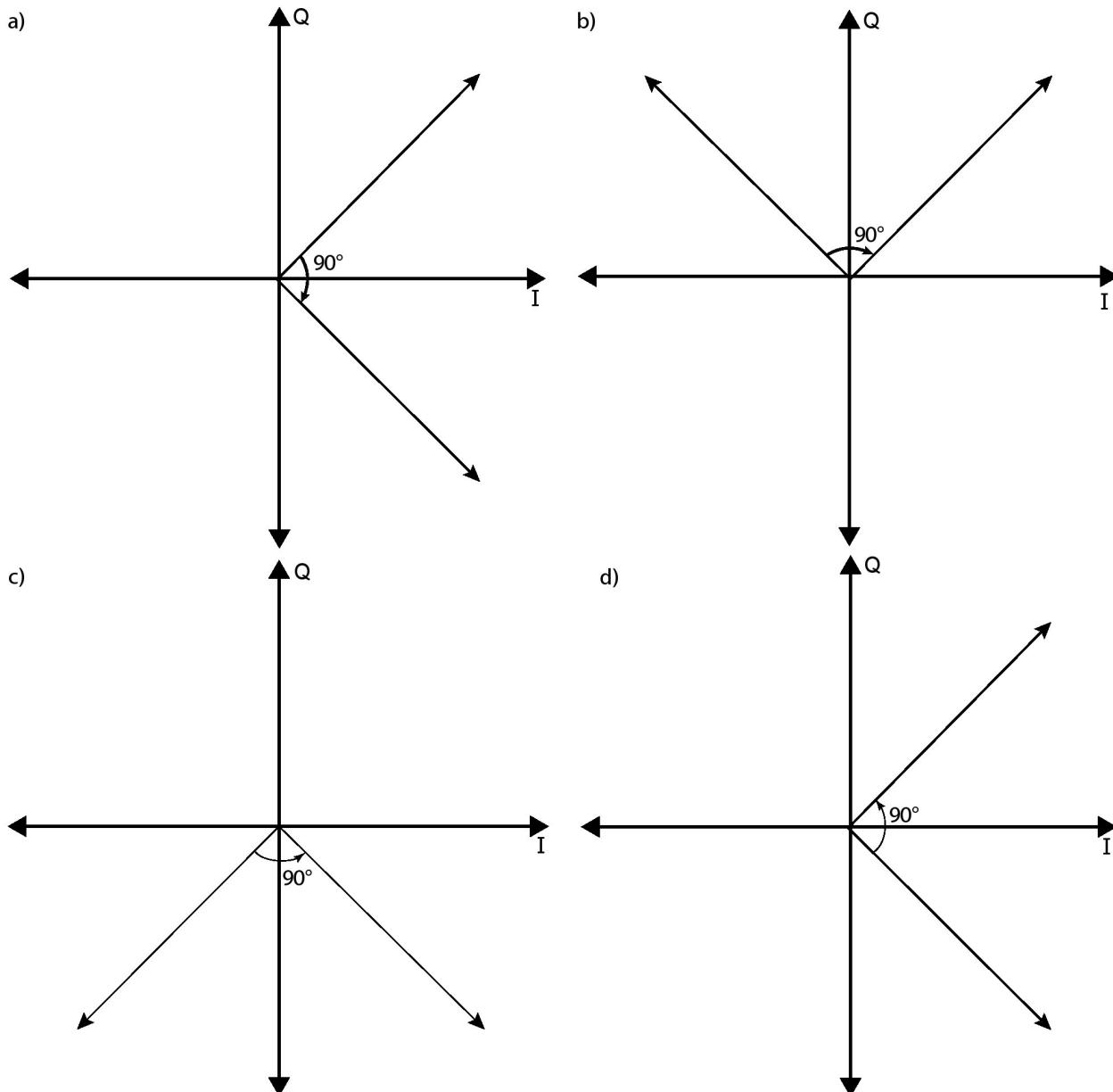


图3.5：笛卡尔坐标系转换为极坐标系的第一步是执行一个正或负90度的旋转，这次旋转的目的是为了使初始矢量到象限I或IV。一旦矢量到了这两个象限，后续的旋转将使矢量到达最终角度 0° 。此时，初始矢量的径向值是最终旋转矢量的x值，初始矢量的角度是CORDIC所做旋转角度累加和。a)和b)展示了一个初始y值为正的例子，这意味着矢量位于象限I或II中。旋转-90度将它们旋转到相应象限中。c)和d)给出了类似的情况，当初始矢量y值为负值时，我们希望旋转+90度使矢量转到第I或第IV象限。

当我们表示一个普通整数，例如4062，它可以表示成 $(4 \ 1000)+(0 \ 100)+(6 \ 10)+(2 \ 1) = 4062$ ，或者我们可以按如下方式表示

10^3	10^2	10^1	10^0	unsigned
4	0	6	2	=4062

二进制数表达方式与这个相似，不过没有用到数字0到9和10的幂。二进制数用0和1以及2的幂来表示数值

2^3	2^2	2^1	2^0	unsigned
1	0	1	1	=11

因为 $(1 \cdot 8) + (0 \cdot 4) + (1 \cdot 2) + (1 \cdot 1) = 11$ 。为了避免歧义，二进制数经常在前边加“0b”，这个使0b1011很显然表示的是十进制11而不是数值1011。二进制数最高位是最关键的位置，同时二进制数最低位是最不重要的位置。

十六进制数用0到15和16的幂次来表示数值

16^3	16^2	16^1	16^0	unsigned
8	0	3	15	=32831

为了避免歧义，数字10到15表示为字母“A”到“F”，同时十六进制数据前缀“0x”，因此在c语言中上面的数值可表示为0x803F。

注意，二进制同样可以表示小数，这种类型数据通常叫做定点数，通过简单的拓展这种形式也可以表示负数。因此0b1011.01等效于：

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	unsigned
1	0	1	1	0	1	=11.25

因为 $8 + 2 + 1 + 1/4 = 11.25$ 。不幸的是，C标准没有提供二进制表示常数的形式，尽管gcc和一些其他的编译器允许整型常数（没有小数点），这些常量前边会加“0b”。C99标准提供了一种描述浮点常数的方法，这种方法通过十六进制和指数来表示，注意指数是无法省略的，甚至值为0的情况下也不能省略。

```
float p1 = 0xB.4p0; //Initialize p1 to "11.25"
float p2 = 0xB4p-4; //Initialize p2 to "11.25"
```

一般来说，对于一个无符号数，只需要填写数值中非零的数字，没有显示的数字都可以假设为零。因此，用更多的数字表示相同的值很容易：只要添加尽可能多的零就可以了。这个过程通常称为零扩展。注意，每个附加的数字都增加了可以表示数值的范围。在二进制数中每添加一位，可以使表示的数字范围增加一倍，而在十六进制数字中每增加一位会使可表示的数字范围增加16倍。

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	unsigned
0	0	0	0	1	0	1	1	0	1	0	0	=11.25

注意，二进制数可以定义为任意位宽，而不仅仅是8、16和32位。例如System C[2]通过定义一系列模板类，可以实现任意精度整数和定点数（包括sc_int<>, sc_uint<>, sc_bigint<>, sc_ubigint<>, sc_fixed<>, 和sc_ufixed<>）尽管它们最初是为系统模型定义的，而不是为了综合，但这些类在HLS工具中可以广泛使用。Vivado HLS包含相似的模板类（ap_int<>, ap_uint<>, ap_fixed<>, 和ap_ufixed<>）这些类在通常情况下在仿真和综合上都会比SystemC类性能好。

通过使用0值，任意精度数被很好的定义(虽然用处不大)。列出所有能用0位数字表示的数字。

3.5.2 负数

负数的表示比正数稍微复杂一些，一定程度上是因为它有不同的表示方法。一种简单的方法是用符号位表示负数，通常称为有符号幅值表示。这种表示只是在数字前面添加了一个额外的位，以表示它是否有符号。有符号幅值表示法的奇怪之处是，有不止一种方法可以表示0。这使得有些简单操作例如 **operator == 0** 实现起来更加复杂。

$+/-$	2^1	2^0	signed magnitude
0	1	1	=3
0	1	0	=2
0	0	1	=1
0	0	0	=0
1	0	0	=-0
1	0	1	=-1
1	1	0	=-2
1	1	1	=-3

另一种表示负数的方法是用偏移码表示。这种表示方法是增加了一个常量偏移(通常等于最高位数值)，否则视为正数：

2^2	2^1	2^0	biased
1	1	1	=3
1	1	0	=2
1	0	1	=1
1	0	0	=0
0	1	1	=-1
0	1	0	=-2
0	0	1	=-3
0	0	0	=-4

然而，到目前为止，最常用的用于实现负数的技术被称为“二进制补码”。二进制补码中最重要的位表示数值的符号(即数值表示为符号+大小)，二进制补码也表示偏移量是否可以用。理解这种表示方法的思路是将最高位表示成对应位宽的负数。

-2^2	2^1	2^0	two's complement
0	1	1	=3
0	1	0	=2
0	0	1	=1
0	0	0	=0
1	1	1	=-1
1	1	0	=-2
1	0	1	=-3
1	0	0	=-4

-2^4	2^3	2^2	2^1	2^0	two's complement
0	0	0	1	1	=3
0	0	0	1	0	=2
0	0	0	0	1	=1
0	0	0	0	0	=0
1	1	1	1	1	=-1
1	1	1	1	0	=-2
1	1	1	0	1	=-3
1	1	1	0	0	=-4

无符号数与二进制补码之间的一个重要区别是，我们需要确切地知道数值有多少位来表示，因为最高位与其它位代表不同的意义。此外，当扩展带符号的二进制补码时要复制符号位到所有新增的比特位。这个过程通常称为符号位扩展。在本书的其余部分中，除非另有说明，我们通常假设所有带符号的数字都用二进制补码表示。

N位二进制补码能表示的正数最大是多少?最大的负数是多少?

给定一个正数x,如何用二进制补码来表示 $-x$?-0的二进制补码是什么?如果x是N位二进制补码所能表示的最大的负数,那么 $-x$ 是什么?

3.5.3 溢出、下溢和舍入

当一个数值大于给定位宽所能表示的最大值时，就发生溢出(Overflow)。类似地，当一个数值小于可以表示的最小值时，就发生下溢(Underflow)。处理溢出或下溢的常见方法是简单地删除原始数字中最重要的位，这个操作通常称为包装(Wrapping)。

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
0	0	1	0	1	1	0	1	0	0	=11.25
	0	1	0	1	1	0	1	0	0	=11.25
		1	0	1	1	0	1	0	0	=11.25
			0	1	1	0	1	0	0	=3.25

通过包装二进制补码的方式解决溢出和下溢的问题可能会导致正数变为负数，负数变为正数。

-2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two' complement	
1	0	1	1	0	1	0	0	=-4.75	

-2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	two' complement	
0	1	1	0	1	0	0	=3.25	

同样，当一个数值不能通过给定小数位数来精确地表示时，需要采用舍入来解决这个问题。同样，这里有几种常见的方法来对数值进行舍入。最简单的方法就是去掉额外不能表示的小数位数，但这样就会使数值变的更负。这种舍入的方法通常被称为向下舍入或向负无穷舍入。当向下舍入到最近的整数时，它就能与float()函数对应起来，尽管它也可能舍入到其他小数位。

0b0100.00	= 4.0		0b0100.0	= 4.0
0b0011.11	= 3.75		0b0011.1	= 3.5
0b0011.10	= 3.5		0b0011.1	= 3.5
0b0011.01	= 3.25		0b0011.0	= 3.0
0b0011.00	= 3.0	Round to Negative Infinity	0b0011.0	= 3.0
0b1100.00	= -4.0	→	0b1100.0	= -4.0
0b1011.11	= -4.25		0b1011.1	= -4.5
0b1011.10	= -4.5		0b1011.1	= -4.5
0b1011.01	= -4.75		0b1011.0	= -5.0
0b1011.00	= -5.0		0b1011.0	= -5.0

同样，也可以使用类似的方法使舍入后数值变得更大(这种称为向上舍入或向正无穷舍入与ceil()函数相对应),向绝对值较小的方向舍入(称为向零舍入与trunc()函数相对应),或向绝对值更大的值舍入(称为远离零舍入或向无穷舍入和round()函数相对应)。然而，这些操作中没有一个总可以最小化舍入误差。一种更好的方法叫做向最接近偶数舍入、收敛舍入或四舍六入五成双，它在lrint()函数中实现。正如你所期望的，这种舍入方式总是选择最近可表示数字。另外，如果有两个误差相等的数，则取偶数。如果最后一个数值是零，

那么任意精度数是一个偶数。这种方法是处理IEEE浮点数舍入的默认方法，因为它不仅可以最小化舍入误差，而且还可以确保计算随机数舍入误差之和趋于零。

0b0100.00	= 4.0		0b0100.0	= 4.0
0b0011.11	= 3.75		0b0100.0	= 4.0
0b0011.10	= 3.5		0b0011.1	= 3.5
0b0011.01	= 3.25		0b0011.0	= 3.0
0b0011.00	= 3.0	→ Round to Nearest	0b0011.0	= 3.0
0b1100.00	= -4.0	Even	0b1100.0	= -4.0
0b1011.11	= -4.25		0b1100.0	= -4.0
0b1011.10	= -4.5		0b1011.1	= -4.5
0b1011.01	= -4.75		0b1011.0	= -5.0
0b1011.00	= -5.0		0b1011.0	= -5.0

3.5.4 二进制运算

二进制加法与十进制加法相似，可以简单地对齐每位二进制数并进行加法运算，注意正确处理从一列到下一列的进位操作。注意，两个N位数值加或减的结果通常需要N+1位的数值来正确表示才不能溢出。对于小数，总是在符号位进行增加位宽

	2^3	2^2	2^1	2^0	unsigned
		0	1	1	=3
+		0	1	1	=3
=	0	1	1	0	=15

注意由于减法结果可能是负数，那个“额外的位数”就成了有符号二进制补码数据的符号位。

	2^3	2^2	2^1	2^0	2^{-1}	unsigned
		1	1	1	1	=7.5
+		1	1	1	1	=7.5
=	1	1	1	1	0	=15

二进制数乘法与十进制数乘法相似。通常说，两个N位数据做乘法，最后结果位宽是 $2 \times N$

	2^3	2^2	2^1	2^0	unsigned
	0	0	1	1	=3
+	0	0	1	1	=3
=	0	0	0	0	=0

	2^{-4}	2^3	2^2	2^1	2^0	unsigned
		0	0	1	1	=3
+		1	1	1	1	=15
=	1	1	1	1	0	=-12(two's complement)

由于对符号位要进行单独处理，所以有符号数的操作更加复杂。然而，对于结果数据位宽的计算方法仍然适用：两个N位数值进行加减法，结果位宽为N+1位，两个N位有符号数据做乘法，结果将有2*N位。

关于除法怎么分析呢？这个数值位宽是否可以精确的表示两个N位数值的除法呢？

3.5.5 表示C/C++中任意精度整数

根据C99语言标准，已经定义了许多标准数据类型精度，例如**int**和**long**。虽然有些程序可以改变这些数据类型的精度来构建代码，但大多数程序是不能这样用的。在C99标准inttypes.h头文件中有一个小的改进，这个头文件中定义了固定位宽的有符号数据类型int8_t、int16_t、int32_t和int64_t，以及相应的无符号类型uint8_t、uint16_t、uint32_t和uint64_t。尽管这些数据类型有明确的位宽定义，但是它们使用起来仍然有些不方便。例如，即便像下面这样简单的代码也会有意外的行为发生。

```
#include "inttypes.h"
uint16_t a = 0x4000;
uint16_t b = 0x4000;
//Danger! p depends on sizeof(int)
uint32_t p = a * b;
```

尽管a和b的值可以用16位二进制数表示，而且他们的积(0x10000000)可以精确的用32位二进制数表示，但是依据C99的转换协议在计算开始时会把a和b的值强制转换为**int**类型，然后计算出一个整型的结果，之后再将处理结果拓展到32位。尽管不常见，对于C99编译器来说整数的精度只有16位。更进一步，C99标准对于整数只定义了4位宽度，然而FPGA系统经常应用自定义的位宽来进行算法运算。同时，用printf()函数不支持这类类型的数据打印，要求应用额外的宏定义来编写代码。如果你考虑用定点进行运算，情况会变得更糟。以下代码中，我们考虑a和b为定点数，在相同的结构下运算产生正确结果。

```
#include "inttypes.h"
//4.0 represented with 12 fractional bits
uint16_t a = 0x4000;
//4.0 represented with 12 fractional bits
uint16_t b = 0x4000;
//Danger! p depends on sizeof(int)
uint32_t p = (a*b) >> 12;
```

在以下两种情况，正确的代码都要求在输入变量相乘之前将变量位宽转换为与结果位宽一致。

```
#include "inttypes.h"
uint16_t a = 0x4000;
uint16_t b = 0x4000;
//p is assigned to 0x10000000
uint32_t p = (uint32_t) a*(uint32_t) b;
```

```
#include "inttypes.h"
//4.0 represented with 12 fractional bits
uint16_t a = 0x4000;
//4.0 represented with 12 fractional bits
uint16_t b = 0x4000;
//Danger! p depends on sizeof(int)
uint32_t p = ((uint32_t) a*(uint32_t) b) >> 12;
```

当用整数代表定点数值时，定点数的格式是非常重要的，只有这样在做乘法后的归一化操作才能正确。通常定点数用“Q”的形式来给出小数位数。例如“Q15”格式表示有15位小数，而且它经常用来表示16位有符号变量。这样的变量数值区间为[-1,1]。相似的，“Q31”表示有31位小数。

由于这些原因，最好使用c++和Vivado HLS模板类`apint<>`, `ap_uint<>`, `ap_fixed<>`, `ap_ufixed<>`来表示任意精度数据。`ap_int<>`和`ap_uint<>`模板类需要整数参数来定义他们的位宽。计算函数通常产生结果的位宽足够宽，可以表示正确结果，参照3.5.4节中的规则。只有当结果分配给较小的位宽时，才会发生溢出或下溢。

```
#include "ap_int.h"
ap_uint<15> a = 0x4000;
ap_uint<15> b = 0x4000;
//p is assigned to 0x10000000
ap_uint<30> p = a*b;
```

`ap_fixed<>`和`ap_ufixed<>`模板类相似，它们都需要两个整数参数来定义位宽（数据所有位数）和数据整数的位宽。

```
#include "ap_fixed.h"
//4.0 represented with 12 fractional bits
ap_ufixed<15,12> a = 4.0;
//4.0 represented with 12 fractional bits
ap_ufixed<15,12> b = 4.0;
//p is assigned to 16.0 represented with 12 fractional bits
ap_ufixed<18,12> p = a*b;
```

注意`ap_fixed<>`和`ap_ufixed<>`模板类位宽数据要为正数，但是数值可以是任意的。特殊情况下，整数的位宽可能是零（暗示这个数是纯小数）或者这个数和数值位宽总数相同（暗示这个数没有小数）。然而，整数的位宽可以是负数或者比总位宽大的数！这些数据格式描述的数值是什么样子的？`ap_fixed<8,-3>`能表示的最大数和最小数是什么？`ap_fixed<8,12>`呢？

3.5.6 浮点

Vivado HLS还可以综合浮点运算。浮点数据提供了很高精度，但这是有代价的；它需要大量的计算资源，这意味着很大的硬件资源开销和多个时钟周期延迟。因此，应该避免应用浮点数，除非应用场景根据精度要求必须用浮点数。事实上，本章的首要目标是让读者了解在工程中如何有效地从浮点数过渡到定点数。不幸的是，这通常是一项非常重要的任务，并且没有完美的标准方法可以自动执行转换。这个原因一部分是由于转换到定点将降低处理运算精度，所以浮点转换定点的性能资源折衷最好留给设计人员决定。

用高级语言综合开发应用程序，初始数据类型都是采用浮点类型。这使设计人员专注于一个功能的正确实现。一旦实现了这一点，她就可以优化数据类型，来减少资源使用和/或提高性能。

将CORDIC中所有变量从float更改为int，这对资源使用有什么影响？它如何改变延迟？吞吐量如何变化？精度会有变化吗？

3.6 进一步优化

在本节中，我们将对优化CORDIC函数的方法提出一些简要的想法和建议。在权衡吞吐量、精度和资源的同时，我们关注不同优化方式如何对处理结果精度产生影响。

最后，CORDIC运算得到一个近似值。随着迭代次数的增加，近似值的误差通常会减小。这对应于我们在cordic函数中执行**for**循环的次数，该循环次数由NUM_ITERATION进行设置。即使我们执行了大量迭代，我们仍然可能得到一个近似值。这样做的一个原因是我们可以接近，但从来没有得到完全匹配的目标角度。然而，我们可以通过选择执行更大或更小的迭代角度来调整精度。上述算法中需要更改的地方都可以通过修改NUM_ITERATIONS的值来实现。NUM_ITERATIONS数值依赖于使用这个CORDIC算法应用程序所需精度。

随着数据类型的变化，资源、吞吐量、和正余弦结果精度会发生怎样的变化？

常量NUM_ITERATIONS如何影响资源、吞吐量和精度？它如何影响current_cos和current_sin的初始值？需要修改cordic_phase矩阵吗？可以根据NUM_ITERATIONS的数值优化数据类型吗？

for循环中的计算占用了大部分时间。如何最好地进行代码转换和/或使用编译指令来优化它？

在硬件中两输入复用器可以高效实现 σ 变量。你可以调整代码以便高级语言综合（HLS）工具以这种方式实现它吗？

当前代码假设给定的角是在正负90度之间。你能添加代码使其可以处理任何 $\pm 180^\circ$ 之间的角度吗？

3.7 结论

在本章中，我们研究了基于矢量旋转计算三角函数和双曲函数的坐标旋转数字算法(CORDIC)。我们从一个由CORDIC方法实现的计算背景开始。我们特别关注了如何使用CORDIC方法计算给定角度的正弦和余弦值。此外，我们还讨论了如何使用相同的CORDIC方法来计算给定复数的振幅和相位。

在此之后，我们将重点放在可以对CORDIC算法进行的优化上。由于它是一种迭代算法，所以在执行迭代次数上和计算结果的精度及准确性之间存在着基本的权衡。我们讨论了如何降低精度/准确性，来节省FPGA资源和提高处理性能。

在CORDIC函数中，我们引入了自定义数据类型的概念。这就提供了一种减小中间和最终结果精度来减少延迟、增加吞吐量和减少资源使用的思路。同时，Vivado HLS提供了一种专门生成大值数据类型的方法。我们提供了数值表示的背景，并介绍了这些自定义数据类型。

通常，精度、资源利用率和性能之间存在复杂的关系。我们讨论了一些权衡方法，并提供了关于如何更好地优化cordic函数的一些建议。我们把多种优化方式和权衡分析留给读者作为练习。

第四章 离散傅里叶变换

本章介绍了DFT，并将重点放在了介绍了DFT在FPGA实现中的算法优化。DFT运算的核心是以一组固定系数执行矩阵向量乘法。在4.6章节中，我们首先将DFT运算初始优化集中在将其简化为矩阵 - 向量乘法，随后介绍了DFT使用Vivado HLS代码的完整实现方式。另外，我们也描述了如何最佳地优化DFT计算以增加吞吐量。第4.5章中，我们将的优化工作集中在阵列分区优化上。

本章的前两小节有大量的数据计算和推导，这可能看起来有些多余，但是它对于我们充分理解代码重构优化以下一章快速傅里叶变换的对称性计算有着很大作用。但是如果你对HLS 优化内容更感兴趣，可以直接跳至第4.6章开始阅读。

4.1 傅里叶级数

为了解释离散傅里叶变换，我们首先要了解傅里叶级数。傅立叶级数提供了一种可选方法来观察信号从 $-\pi$ 到 π 的一个周期内的连续实值周期信号。Jean Baptiste Joseph Fourier的开创性成果表明，在 2π 周期内任何连续的周期性信号都可以用周期为 2π 的余弦和正弦和表示。最终，傅里叶级数的表现形式如下：

$$\begin{aligned} f(t) &\sim \frac{a_0}{2} + a_1 \cos(t) + a_2 \cos(2t) + a_3 \cos(3t) + \dots \\ &\quad b_1 \sin(t) + b_2 \sin(2t) + b_3 \sin(3t) + \dots \\ &\sim \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(nt) + b_n \sin(nt)) \end{aligned} \quad (4.1)$$

其中参数 a_0, a_1, \dots 和 b_0, b_1, \dots 的计算公式如下：

$$\begin{aligned} a_0 &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) dt \\ a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \cos(nt) dt \quad (4.2) \\ b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(t) \sin(nt) dt \end{aligned}$$

有几个需要注意的点是：首先式4.2中的参数 $a_0, a_1, \dots, b_0, b_1, \dots$ 被称作傅里叶参数。其中参数 a_0 被称作直流分量（来自于对早期电流分析的参考），其中 $n=1$ 频率分量称为基波，而其他频率（ $n \geq 2$ ）分量统称为高次谐波。基波和高次谐波的概念来自声学和音乐。其次，函数f以及cos()和sin()函数都有 2π 个周期；我们很快就会展现如何将这个周期改变为其他值。直流分量 a_0 等同于 $\cos(0 \cdot t) = 1$ 时的系数，因此使用符号a。因为 $\sin(0 \cdot t) = 0$ ，所以不需要 b_0 的值。最后，在某些情况下，函数f和它的傅里叶级数之间是近似相等的关系，这种不连续的现象我们称之为吉布斯现象。而这是只是一个仅与傅里叶级数有关的小问题，与其他傅立叶变换无关。因此，今后我们将忽略式[4.1]中的“近似”（ \sim ），直接视为“相等”（ $=$ ）。

表示除 π 以外的周期性函数需要对变量进行简单的更改。假设一个函数的周期范围在 $[-L, L]$ 而不是 $[-\pi, \pi]$ ，则设：

$$t \equiv \frac{\pi t'}{L} \quad (4.3)$$

以及

$$dt \equiv \frac{\pi dt'}{L} \quad (4.4)$$

这是一个简单地将周期区间从 $[-\pi, \pi]$ 变换到期望的 $[-L, L]$ 的一个线性方程，将 $t' = \frac{Lt}{\pi}$ 代入到式4.1得：

$$f(t') = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(\frac{n\pi t'}{L}) + b_n \sin(\frac{n\pi t'}{L})) \quad (4.5)$$

用同样的方法解得a和b的各项参数可解得：

$$\begin{aligned} a_0 &= \frac{1}{L} \int_{-L}^L f(t') dt' \\ a_n &= \frac{1}{L} \int_{-L}^L f(t') \cos\left(\frac{n\pi t'}{L}\right) dt' \quad (4.6) \\ b_n &= \frac{1}{L} \int_{-L}^L f(t') \sin\left(\frac{n\pi t'}{L}\right) dt' \end{aligned}$$

我们也可以利用欧拉公式 $e^{jnt} = \cos(nt) + j \sin(nt)$ 来得出一个更简洁的公式。

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{jnt} \quad (4.7)$$

其中，傅里叶参数 c_n 是一个较为复杂的指数表达式：

$$c_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-jnt} dt \quad (4.8)$$

假设 $f(t)$ 是一个具有 2π 个周期的周期函数，将这个公式与式4.1等效，傅里叶参数 $a_n, b_n, and c_n$ 之间的数值关系为：

$$a_n = c_n + c_{-n} \text{ for } n = 0, 1, 2, \dots$$

$$b_n = j(c_n - c_{-n}) \text{ for } n = 0, 1, 2, \dots$$

$$C_n = \begin{cases} 1/2 (a_n - jb_n) & n > 0 \\ 1/2a_0 & \\ 1/2 (a_{-n} + jb_{-n}) & n < 0 \end{cases} \quad (4.9)$$

我们需要注意的是参数 a_n, b_n, c_n 的公式中引入了“负”频率的概念。虽然从物理的角度上看它没有实际意义，但在数学上我们可以将其视为复平面上的“负”旋转。正频率表示复数在复平面上以逆时针方向旋转，负频率表示我们在复平面上以顺时针方向旋转。

余弦，正弦和复指数之间的关系更加证明了上面这个理论，余弦既可以看作复指数的实数部分，也可以推导为一个正频率和一个负频率两个复指数的和，如式4.10所示。

$$\cos(x) = \operatorname{Re}(e^{jx}) = \frac{e^{ix} + e^{-ix}}{2} \quad (4.10)$$

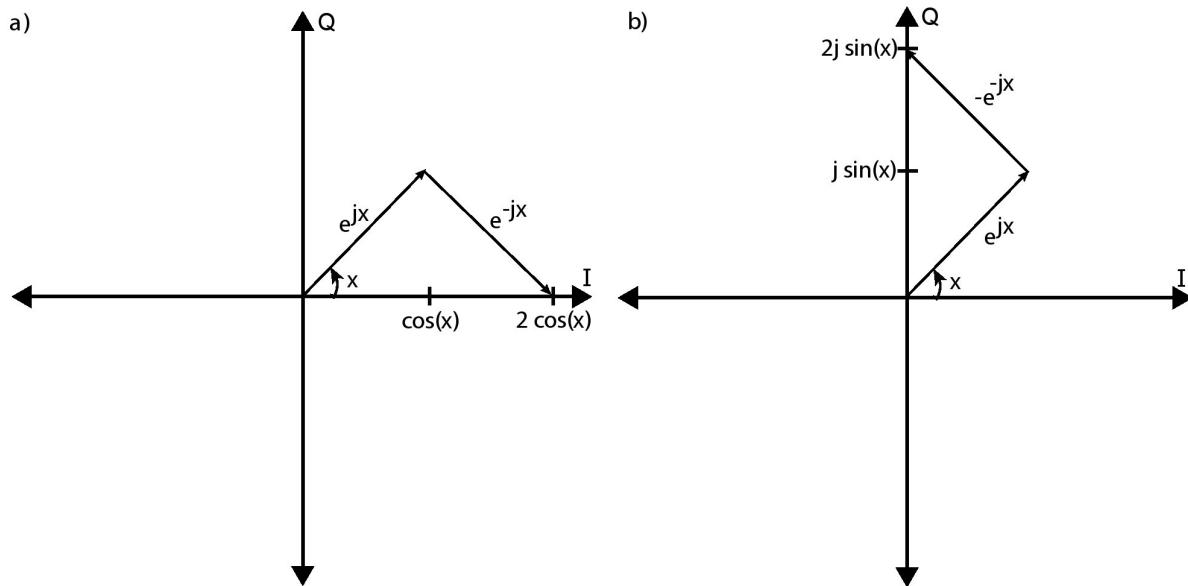
正弦和复指数之间的关系如式4.11所示，与余弦的不同点在于这里我们减去负频率并除以 $2j$ 。

$$\sin(x) = \operatorname{Im}(e^{jx}) = \frac{e^{ix} - e^{-ix}}{2j} \quad (4.11)$$

这两种正余弦和复指数之间的关系都可以用复平面矢量图的形式来理解，如下图4.1所示。图4.1中a)部分显示了余弦的推导过程，这里我们添加两个复平面向量 e^{jx} and e^{-jx} ，图中可以看出这两个向量和是一个在实轴上的向量，大小为 $2\cos(x)$ 。所以，当我们把这两个向量的和除以2就到了式4.10中的 $\cos(x)$ 的值。图4.1中b)部分显示了类似的正弦的推导过程，这里我们添加两个复平面向量 e^{jx} and $-e^{-jx}$ ，图中可以看出这两个向量差是一个在虚轴上的向量，大小为 $2j\sin(x)$ 。所以，当我们把这两个向量的差除以 $2j$ 就到了式4.11中的 $\sin(x)$ 的值。

4.2 DFT背景介绍

上一节我们探究了傅里叶级数的分析数学基础，证明了它对于周期连续性信号的作用，而离散傅里叶变换是针对于离散的周期信号的。DFT可以将有限数量的等间隔样本转换为有限数量的复数正弦曲线。换句话说，它将一个采样函数从一个域（通常是时域）转换到频域。复数正弦曲线的频率取为与输入函数的采样周期相关的频率的基频的整数倍。离散信号和周期信号最重要的关联在于它可以用一组有限的数字表示。因此，可以使用数字系统来实现DFT。



$e^{jx} + e^{-jx}$ 的和。这个求和的结果恰好落在实轴上，其值为 $2\cos(x)$ 。b) 部分显示了一个类似的总和，只是这次对矢量 e^{jx} and $-e^{-jx}$ 求和。这个总和落在虚轴上，其值为 $2\sin(x)$ 。>

DFT适用于同时包含实数和复数的输入函数。直观上，为了轻松入门，我们暂时忽略复数部分，从实数信号开始了解实际DFT的工作原理。

关于术语的简要说明：我们使用小写函数变量来表示时域中的信号，大写函数变量来表示频域中的信号。我们使用()表示连续函数，用[]表示离散函数。例如， $f()$ 是连续的时域函数， $F()$ 是其连续的频域表示。类似地， $g[]$ 是时域中的离散函数， $G[]$ 是将该函数转换到频域。

让我们从图4.2开始分析，左图是一个具有N个样本或从0到N-1运行的点的实值时域信号 $g[]$ 。当我们用DFT分析时域信号时，会得出对应于各个频率的余弦和正弦幅度的频域信号。这些可以看作是余弦幅值对应复数的实数值，而正弦幅值对应复数的虚数的复数，其中包含有 $N/2+1$ 个余弦（实数）和 $N/2+1$ 个正弦（虚数）值。我们称之为复频域函数 $G[]$ 。注意，频域中样本的数量为 $(N/2+1)$ 是由于我们正在分析一个只包含实数的时域信号，复数时域信号经过DFT后将变为具有N个样本的频域信号。

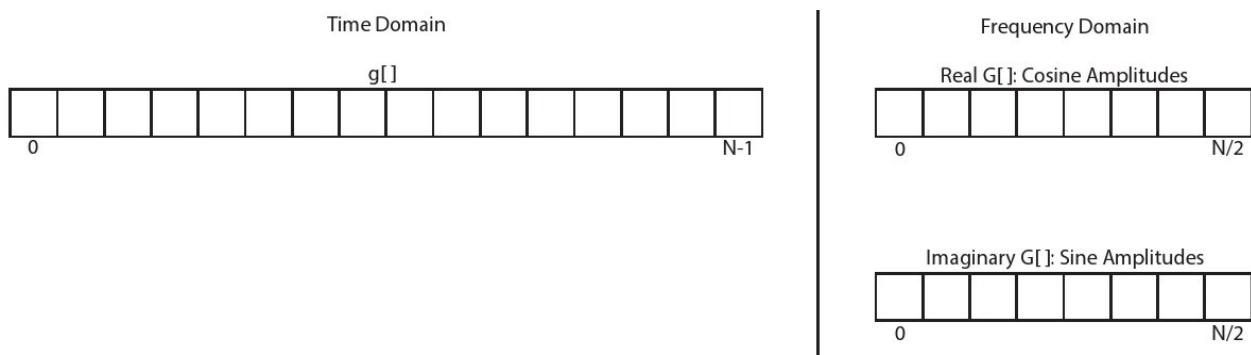


图4.2：一个有 N 个样本点的离散实值时域函数 g 可以用一个具有 $N/2+1$ 个样本点的频域函数表示。每个频域采样都有一个余弦值和一个正弦幅值。这两个幅度值可以共同表示为一个复数，余弦幅度表示实部，正弦幅度表示虚部。

一个具有 N 个样本点的DFT可以通过一个 $N \times N$ 矩阵乘以一个大小为 N 的矢量来确定。 $G = S \cdot g$ 其中

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & s & s^2 & \dots & s^{N-1} \\ 1 & s^2 & s^4 & \dots & s^{2(N-1)} \\ 1 & s^3 & s^6 & \dots & s^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & s^{N-1} & s^{2N-1} & \dots & s^{(N-1)(N-1)} \end{bmatrix} \quad (4.12)$$

其中 $s = e^{-\frac{i2\pi}{N}}$ 。因此，频域中的样本被推导为

$$G[k] = \sum_{n=0}^{N-1} g[n]s^{kn} \quad \text{for } k = 0, \dots, N-1 \quad (4.13)$$

图4.3提供了8个样本点的DFT操作的系数的可视化分析图。八点频域采样是通过将8个时域采样与S矩阵的对应行相乘而得到的。S矩阵的行0对应于与时域采样的平均值成比例的DC分量。将S矩阵的第1行与 g 相乘，得出围绕单位圆圈旋转一圈时的余弦和正弦振幅值。由于这是一个8点DFT，这意味着每个相位偏移45°。执行8个45°旋转将围绕单位圆完整旋转一圈。第2行是相似的，唯一不同点是围绕单位圆执行两次旋转，即每次旋转90°。这是一个更高的频率。第3排做三次旋转；第4排四轮旋转等等。每一个这样的行时间列乘法中都给出了适当的频域样本。

我们可以注意到S矩阵是对角对称矩阵，即 $S[i][j] = S[j][i]$ 。另外， $S[i][j] = s^i s^j = s^{i+j}$ 。在第四行周围也会出现有趣的对称性现象。行3和行5中的相量是彼此的共轭复数，即 $S[3][j] = S[5][j]^*$ 。类似地，行2和6($S[2][j] = S[6][j]^*$)以及行1和7($S[1][j] = S[7][j]^*$)都是彼此的共轭复数。正是由于这个原因，具有 N 个采样点的实值输入信号的DFT在频域中仅具有 $N/2+1$ 个余弦和正弦值。剩余的 $N/2$ 个频域值提供了冗余信息，因此不需要它们。然而，当输入信号复杂时，情况并非如此。在这种情况下，频域将有 $N+1$ 个余弦和正弦值。

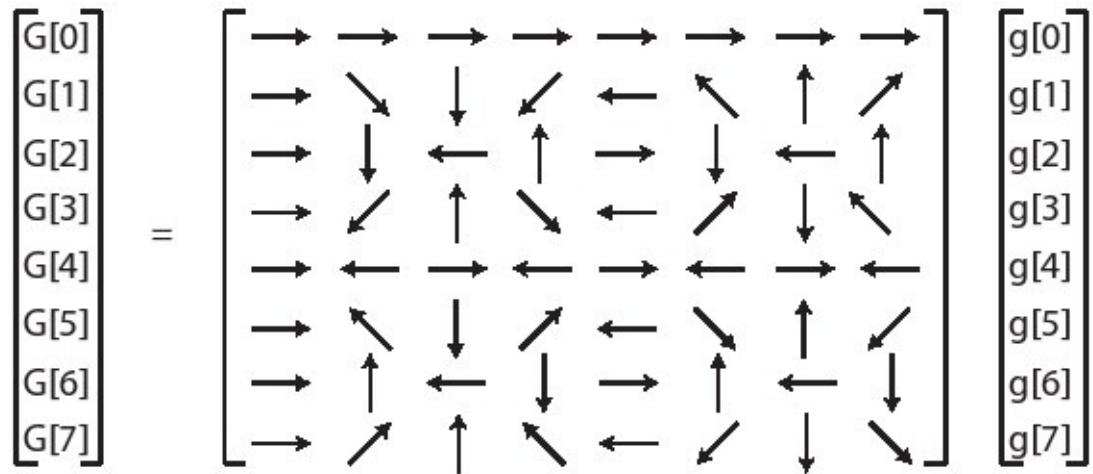


图4.3 矩阵S中元素的复数向量图.

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    BaseType i, j;
data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
dot_product_loop:
        for (j = 0; j < SIZE; j++) {
            sum += V_In[j] * M[i][j];
        }
        V_Out[i] = sum;
    }
}
```

图4.4 实现矩阵向量乘法的简单代码

4.3 矩阵向量乘法的优化

矩阵向量乘法是DFT计算的核心，输入的时域向量将乘以一个固定特殊值的矩阵，输出的结果是与输入时域信号表示相对应的频域矢量。在本节中，我们讨论如何在硬件中实现矩阵向量乘法。我们把这个问题分解成最基本的形式（见图4.4）。这让我们能够更好地将讨论集中在算法优化上，而不是集中在使用功能正确的DFT代码的所有难点上。在下一节中我们将构建一个DFT内核。

图4.4中的代码提供了将该算法实现到硬件中的原始形式，代码中使用当前被映射为浮点型的**BaseType**的自定义数据类型。虽然这在刚开始看起来可能是多余的，但是可以方便我们在将来轻松地将变量（例如，具有不同精度的有符号或无符号的定点）进行不同的数字表示。**matrix_vector** 功能共有三个参数，我们对前两个参数进行乘法计算，输入矩阵和向量分别是 **BaseType M[SIZE][SIZE]** 和 **BaseType V_In[SIZE]**。第三个参数 **BaseType V_Out[SIZE]** 是合成向量。我们将M=S和V_in设为采样时域信号，则Vout将包含DFT。SIZE是决定输入信号中样本数量的常数，相应地也决定了DFT的大小。

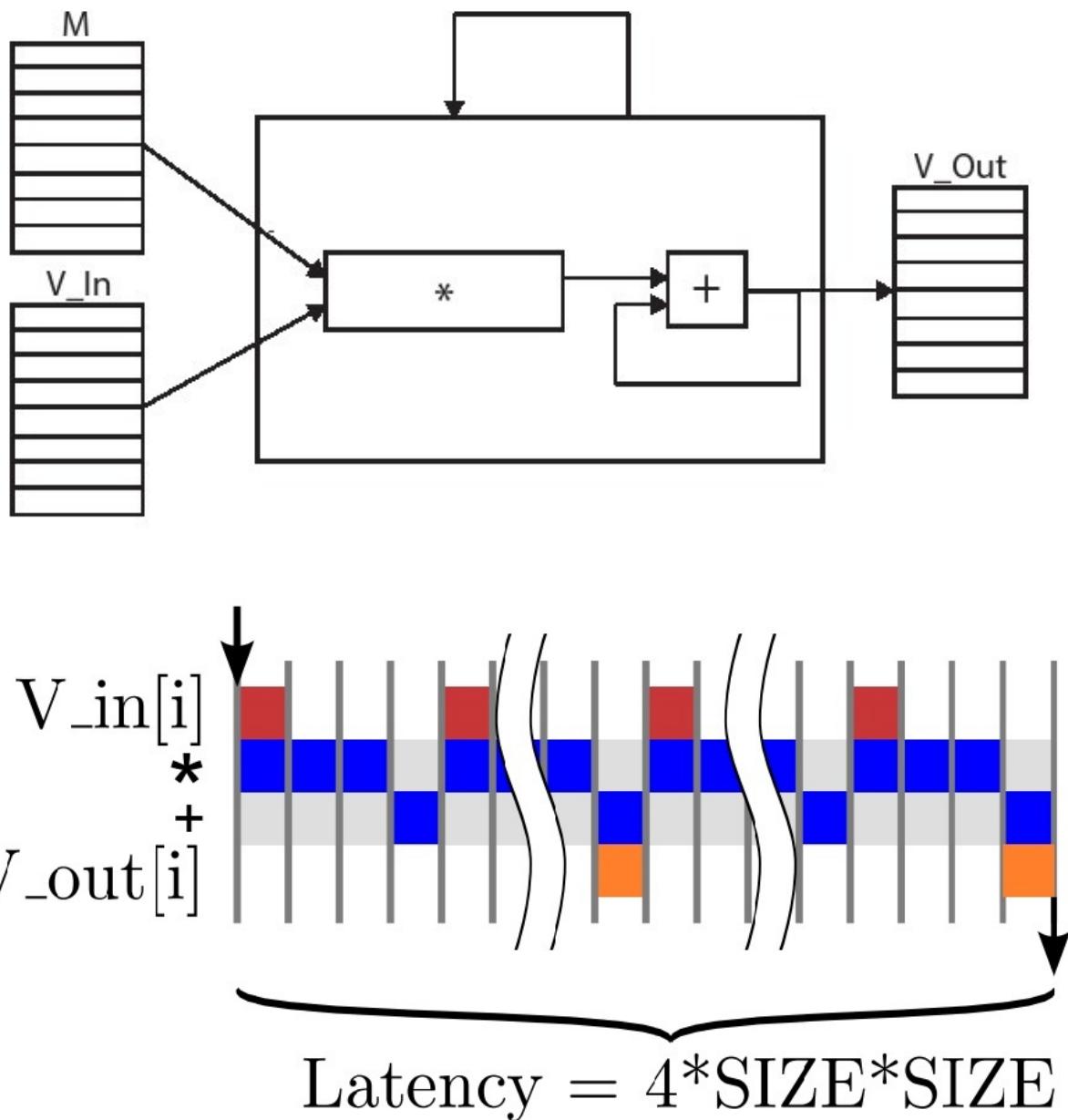


图4.5 一种图4.4矩阵向量乘法代码可能的实现方式。

这个算法本身只是一个嵌套的for循环。内部循环（`dot_product_loop`）从0至SIZE-1计算DFT的系数。但是，这个相对简单的代码在映射到硬件执行时，就有许多种设计方案可选择。

无论何时执行HLS，你都应该考虑希望合成怎样的结构体系。内存结构的组织在这个过程中显得尤为重要。这个问题可以归结为你将代码中的数据存储到哪里？因为将变量映射到硬件时有许多选项。该变量可能只是一组信号（如果它的值永远不需要在一个周期内保存）、寄存器、RAM或FIFO。但所有这些选项都需要你在速度和面积之间作出折衷的选择。

另一个我们需要考虑的重要因素是代码并行度的可用性。纯粹的顺序代码到硬件上实现相当困难。换句话说，一个具有足够并行可行性的代码，从纯粹顺序执行到完全并行实现有一个可以选择的实现自由度。这样的选择显然会带来不同的面积和速度。我们将研究内存配置和并行性如何影响DFT矩阵向量的硬件实现。

图4.5显示了包含一个乘法和一个加法运算符的矩阵向量乘法的顺序结构。我们创建逻辑以访问存储在BRAM中的 V_{In} 和矩阵 M 。计算 V_{Out} 的每个元素并存储到BRAM中。这种体系结构本质上是将图4.4中的代码合成为无指令的结果。它不占用大量面积，但任务延迟和任务间隔相对较大。

4.4 流水线和并行运行

在矩阵乘法的例子中，我们可以很大程度地利用并行思想来解决问题。首先关注每次迭代循环执行的内部循环表达式 $sum += V_i n[j] * M[i][j]$ 。乘法运行时，计数变量 SUM 在每次迭代中都被重复利用并赋予新的值。如图4.6所示，这个内部循环可以重新表述，此时变量 SUM 已被完全消除，并在较大表达式中替换为多个中间值。

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    BaseType i, j;
    data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        V_Out[i] = V_In[0] * M[i][0] + V_In[1] * M[i][1] + V_In[2] * M[i][2] +
                    V_In[3] * M[i][3] + V_In[4] * M[i][4] + V_In[5] * M[i][5] +
                    V_In[6] * M[i][6] + V_In[7] * M[i][7];
    }
}
```

图4.6: 手动展开矩阵向量乘法内部循环实例

循环的展开可以由Vivado HLS在流水线中自动执行，也可以通过使用`#pragma HLS unroll`或者流水线外的等价指令来实现。

我们应该已经发现替换内部循环的新表达式应具有大量的并行性。如此而来每个乘法可以同时执行，并且可以使用加法器树来执行求和。这个计算的数据流图如图4.7所示。

如果我们希望展开内循环的表达式的任务延迟最小，那么所有的八个乘法运算都应该并行执行。假设乘法有3个周期的延迟且加法有1个周期的延迟，则所有 $V_{In}[j] * M[i][j]$ 操作在第三周期结束时完成。使用加法器树对这八个中间结果进行求和需要 $\log_2 8 = 3$ 个周期。因此，对于每次迭代，数据循环主体将共有6个周期的延迟，并且需要8个乘法器和7个加法器，如图4.8左侧所示。需要注意的是，如果在循环4-6中重复使用加法器，这会将加法器的数量减少到4个。但是，在FPGA上加法器通常是无法共享的，因为加法器和多路复用器需要相同数量的FPGA资源（对于2输入运算，大约1个LUT每比特）。

如果我们不愿意使用8个乘法器，则可以增加执行该功能的周期数量来减少资源使用量。例如，使用4个乘法器会使得8个 $V_{In}[j] * M[i][j]$ 乘法操作带来6个周期的延迟，那么完成整个数据循环体将会有9个周期的总延迟，如图4.8的右侧所示。为了使用更少的乘法器，我们需要牺牲更多的时间周期来完成内部循环。

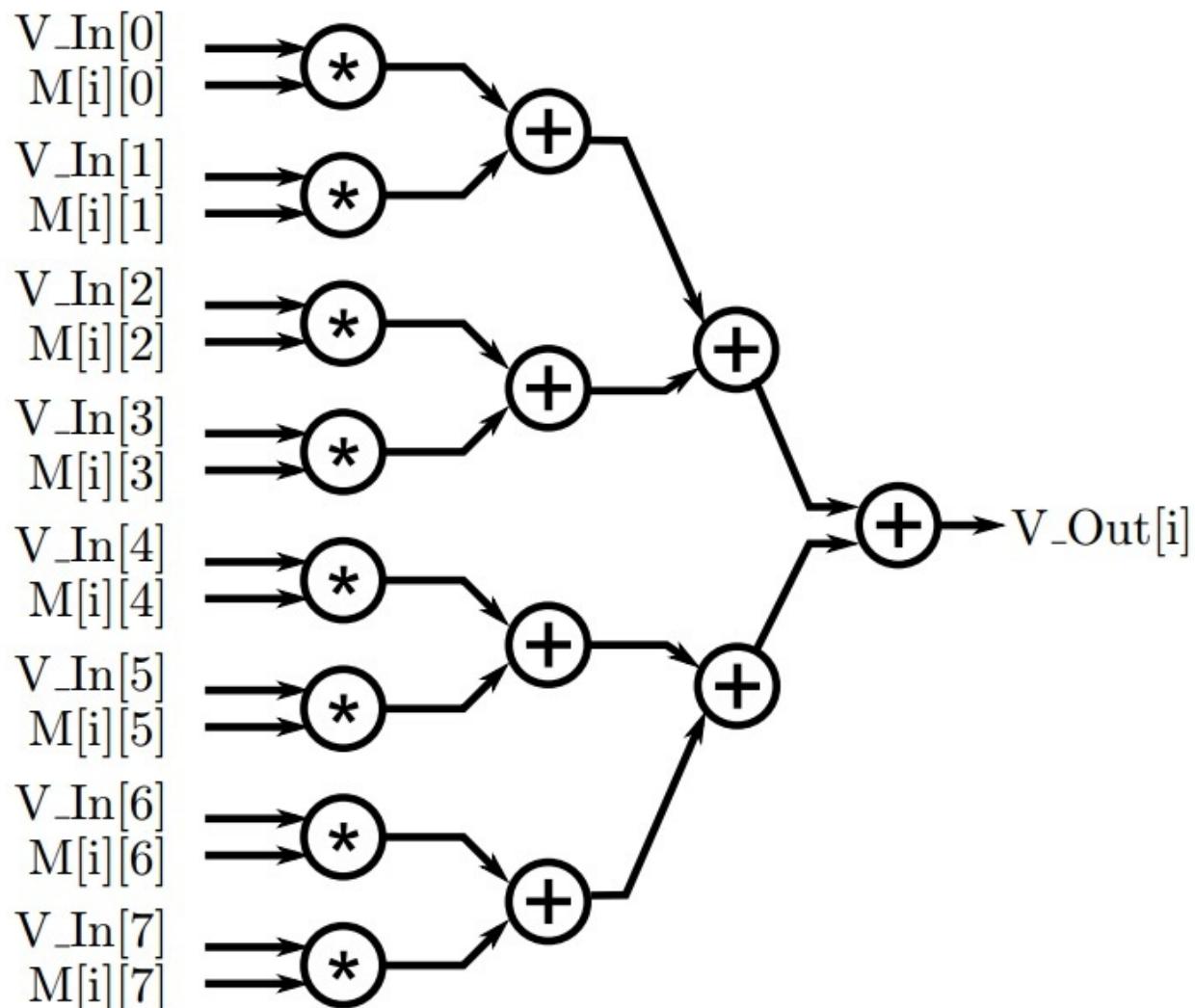


图4.7：图4.6的内循环代码对应的数据流图。

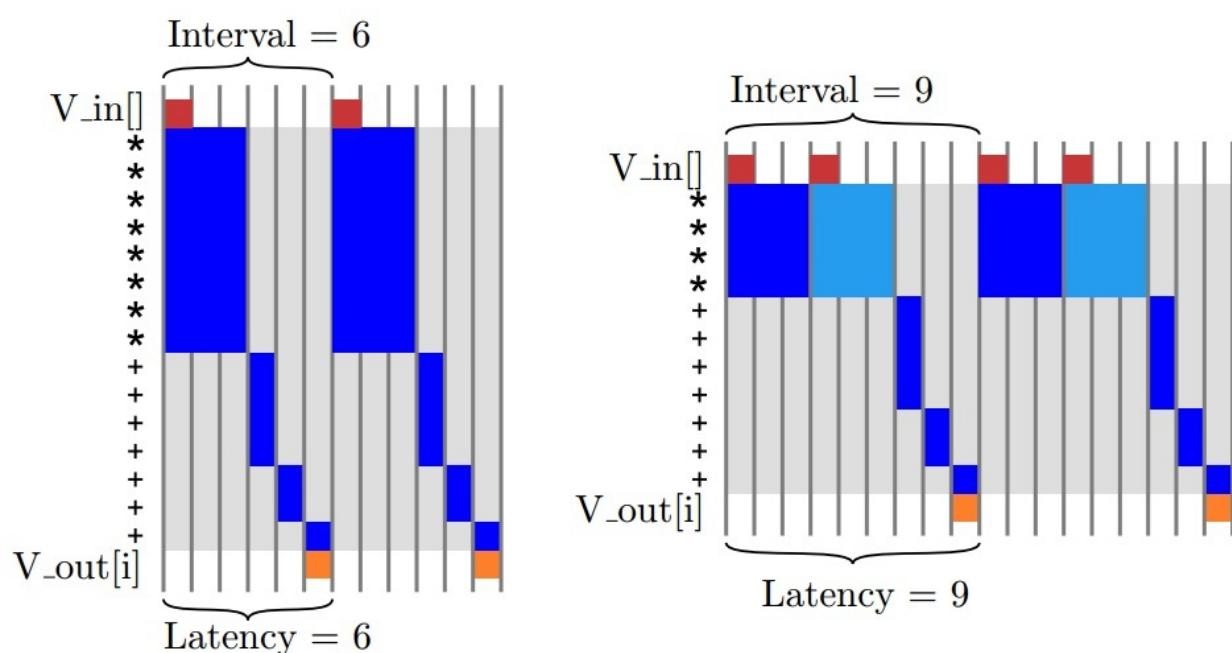


图4.8：由图4.6的内部循环实现的两种不同的顺序模式操作图。

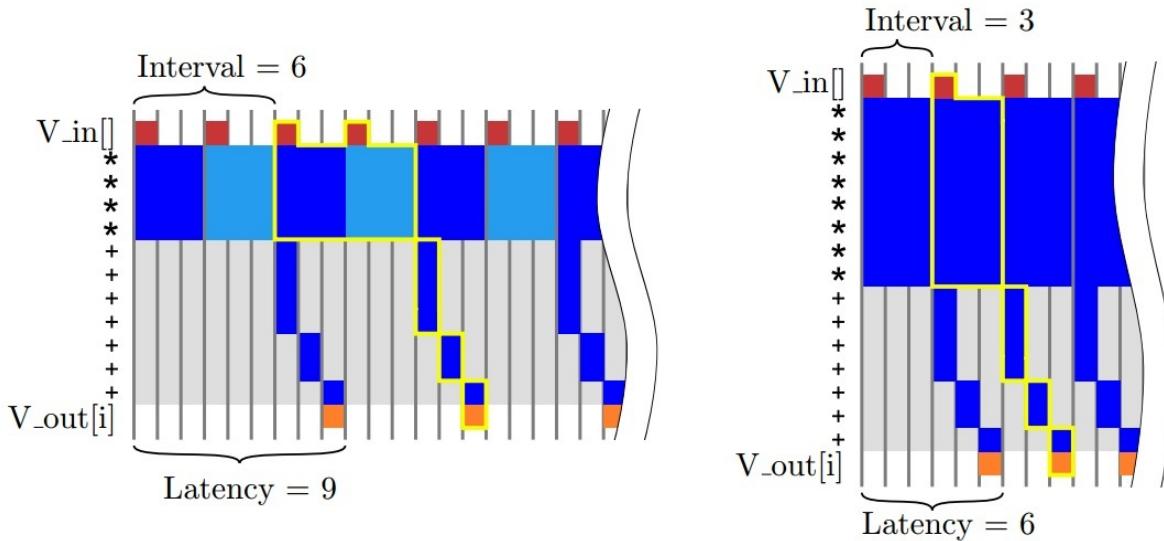


图4.9：由图4.6的内部循环实现的两种不同的流水线模式操作图。

从图4.8中我们可以明显看出有很多重要的时间段并没有执行有效的工作，因而降低了设计的总体效率。我们应该尽量缩短这些时间来提高效率。在这种情况下，可以发现`data_loop`的每次迭代实际上是完全独立的，这意味着它们可以同时执行。正如我们展开`dot_product_loop`一样，也可以展开数据循环并同时执行所有的乘法运算。但是，这需要大量的FPGA资源。我们还有更好的选择是尽快地启动循环的每次迭代，意味着前一次循环仍在执行。这个过程被称为循环流水线化，我们通过`#pragma HLS pipeline`在Vivado HLS中实现。在大多数情况下，循环流水线会减少循环的间隔时间，但不会影响延迟时间。循环流水线的实现如图4.9所示。

截至目前，我们的关注点集中在操作运行的延迟上。功能单元通常也是流水线式的，Vivado HLS中的大多数功能单元都是间隔为1的流水线式的。尽管单次乘法操作可能需要3个周期才能完成，但新的乘法操作可以从流水线乘法器的每个时钟周期开始。通过这种方式，单个功能单元可以同时执行多个乘法操作。例如，有3个周期延迟且间隔为1的乘数可以同时执行三次乘法运算。

充分利用流水线乘法器的优势就在于我们就可以在不添加额外运算符的前提下减少内部循环的延迟。图4.10中左边展示了使用三个流水线乘法器一种可能的实现方式。在这种情况下，乘法操作可以并发执行（因为它们没有数据依赖性），而加法操作只有在第一次乘法完成之后才能开始。在右图中，显示了该设计间隔为3的流水线版本，如果将`#pragma HLS pipeline II=3`应用于`data_loop`，则与Vivado HLS的结果类似。这样不仅个别操作在同一个操作符上并发执行，而且这些操作可能来自于不同`data_loop`的迭代。

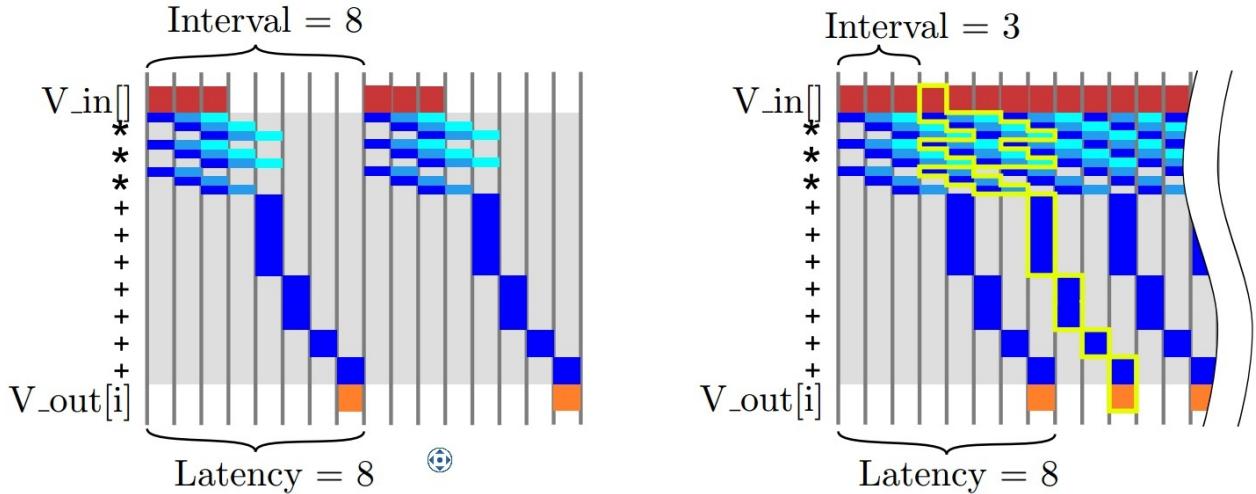


图4.10：由图4.6的内部循环实现的两种不同的流水线乘法器操作。

现在你可能已经观察到，我们可以在不同的层次级别上进行流水线操作，包括算法级别，循环级别和功能级别。此外，不同级别的流水线在很大程度上也是独立的！我们可以在顺序循环中使用流水线操作符，或者我们可以使用顺序操作符来构建流水线循环，也可以构建大型功能的流水线实现。这些功能可以像原始运算单元一样在Vivado HLS 中共享。我们实例化了多少运算单元，它们的个体成本以及使用频率如何才是最重要的。

4.5 存储权衡和数据分区

到了本小节，我们已经假定数组中的数据 $V_{In}[], M[][],$ 和 $V_{Out}[]$ 可以随时访问，但是实际上，数据的放置的位置对整个处理器的性能和资源使用情况有重要影响。在大多数处理器系统中，内存架构是固定的，我们只能调整程序以尝试最大程度地利用可用的内存层次结构，例如注意尽可能减少寄存器溢出和缓存丢失。在HLS设计中，我们还可以利用不同的存储器结构，并尝试找到最适合特定算法的存储器结构。通常，大量数据存储在片外存储器如DRAM、闪存或网络连接的存储器中，但是数据访问时间通常很长，大约为几十到几百（或更多）个周期。由于大量的电流必须流过长电线已访问片外存储器，所以使用片外存储消耗的能量也比较大。相反，片上存储器可以快速访问并且功耗要低得多，只是它可以存储的数据量有限。有一种常见的操作模式类似于通用CPU的内存层次结构中的缓存效果，它是将数据重复地加载到块中的片上存储器上。

当我们选择片上存储器的时候，需要在嵌入式存储器（例如Block RAM）或触发器（FF）之间权衡。基于触发器的存储器允许在一个时钟内对不同地址的数据进行多次读取，也可以在一个时钟周期内读取、修改和写入基于触发器的存储器。然而，即使在资源配置最好的设备中，FF的数量通常也限制在大约10万字节左右。实际上，以便有效地使用其他FPGA资源，大多数基于FF的存储器应该小得多。Block RAM (BRAM) 提供更高的容量，拥有Mbytes的存储量，其代价是有限的可访问性。例如，单个BRAM可以存储大于1到4千字节的数据，但是在每个时钟周期只可以对该数据的两个不同的地址进行访问。此外，BRAM需要尽可能减少流水线操作（比如，读操作必须具有至少一个周期的延迟）。因此，我们的基本的权衡点在于工程所需的带宽与容量。

如果说数据的吞吐量我们需要考虑的头号问题，则所有数据都将存储在FF中。这将允许任何元素在每个时钟周期内被访问尽可能多的次数。但是，随着矩阵阵列变大，这种方案也将变得不可行。在矩阵向量乘法的情况下，存储1024位乘以1024位矩阵的32位整数将需要大约4兆字节的存储器。即使使用BRAM来存储，由于每个BRAM存储大约4KBytes，也需要大约1024个BRAM块。另一方面，使用单个大型基于BRAM的内存意味着我们一次只能访问两个元素。这明显降低了性能，如图4.7所示，它需要在每个时钟周期访问

多个数组元素 ($V_{In}[]$ 的所有 8 个元素以及 $M[][]$ 的 8 个元素)。在实际工程中，大多数设计需要更大的阵列分布存放在更小的 BRAM 存储器中，这种方法称为阵列分区。较小的数组（通常用于索引较大的数组）可以完全划分为单独的标准变量并映射到 FF。匹配流水线选择和数组分区以最大限度地提高运算符使用率和内存使用率是 HLS 设计探索的一个重要方面。

Vivado HLS 将自动执行一些阵列分区，但由于阵列分区倾向于某些特定设计，因此通常需要我们利用好工具以获得最佳结果。阵列分区的全局配置可在 config_array_partition 选项中找到。单个数组可以使用 array_partition 指令来显示分区，并将指令数组分区完成将数组的每个元素分解到它自己的寄存器中，最终形成基于 FF 内存。与许多其他基于指令的优化一样，通过手动重写代码也可以实现相同的效果。一般情况下，最好使用工具指令，因为它避免了引入错误并易于代码维护。

回到图 4.4 中的矩阵向量乘法代码，我们可以通过添加几个指令来实现高度并行，如图 4.11 所示，最终的体系结构如图 4.12 所示。请注意，内部 j 循环由 Vivado HLS 自动展开，因此 j 在每次使用的时候都被替换为常量。此设计演示了阵列分区的最常见用法，其中分区的数组维度（在本例中为 $V_{In}[]$ 和第二维的 $M[][]$ ）都被索引为常量（在本例中为循环索引 j 来展开循环）。这使得多路复用器可以无需访问分区阵列架构。

我们还可以用更少的乘法器降低性能以实现其他设计。例如，在图 4.10 中，这些设计只使用三个乘法器，因此我们只需要在每个时钟周期读取三个矩阵 $M[][]$ 和矢量 $V_{in}[]$ 的元素。完全分割这些数组会导致额外的多路复用，如图 4.13 所示。实际上，阵列只需要分成三个物理存储器。同样，这种分区可以通过重写代码手动实现，也可以使用 array_partition 循环指令在 Vivado HLS 中实现。

让我们从包含数据的矩阵 X 来分析：

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
    #pragma HLS array_partition variable=M dim=2 complete
    #pragma HLS array_partition variable=V_In complete
    BaseType i, j;
    data_loop:
        for (i = 0; i < SIZE; i++) {
            #pragma HLS pipeline II=1
            BaseType sum = 0;
            dot_product_loop:
                for (j = 0; j < SIZE; j++) {
                    sum += V_In[j] * M[i][j];
                }
                V_Out[i] = sum;
        }
}
```

图 4.11 选用阵列分割和流水线操作的矩阵向量乘法

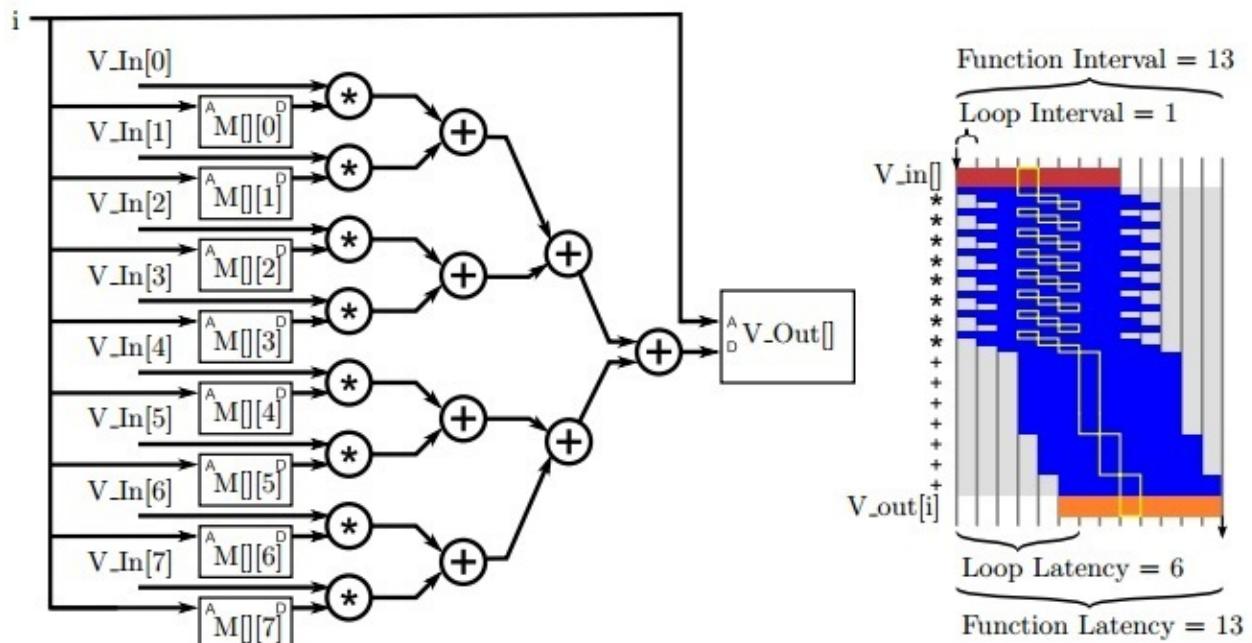
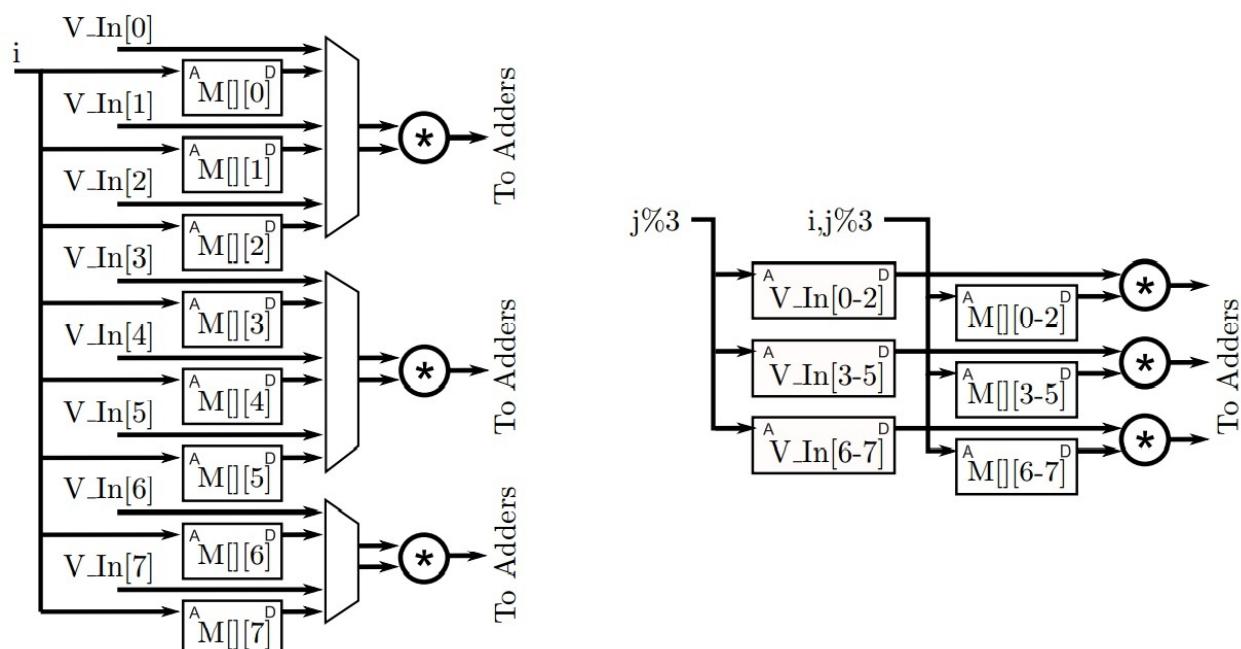


图4.12 具有特定的阵列分区和流水线选择的矩阵-矢量乘法体系结构。右图的流水线寄存器已经被删除。

图4.13 $I = 3$ 时具有特定的阵列分区选择的矩阵向量乘法体系结构。如左图所示，阵列已被分割得超过需要以至于使用了多路复用器。如右图所示，数组用 $factor=3$ 分区，这样减少了多路复用器的使用，但 j 循环索引成为地址计算的一部分。

相似地，如果我们使用 `array_partition variable=x factor=2 block` 指令可以将它分为两个矩阵向量：
[1 2 3 4 5] and [6 7 8 9]

让我们来研究一下变化的流水线II和阵列分区对性能和面积的影响，比较根据每秒的矩阵向量乘法运算（吞吐量）和根据展开阵列分区因子的数量的性能高低，并绘制相同的区域趋势图（如显示LUT，FF，DSP模块，BRAM）。再思考这两种情况的总趋势是什么？你会选择哪种设计？为什么？

通过流水线操作并将部分循环展开应用于 **dot_product_loop**，我们也可以得到类似的结果。图4.14显示了将矩阵向量乘法代码的内部循环展开2倍的结果。你可以发现，循环的边界现在增加到2；每个循环迭代需要2个矩阵M[][]和向量V_in[]每次迭代并执行两次乘法而不是一次。使用这种方式循环展开后，对应于原始循环的两次迭代，Vivado HLS可以并行地在两个表达式中实现这些操作。请注意，如果没有适当的数组分区，展开内部循环可能不会提高性能，因为并发读取操作的数量受到内存端口数量的限制。在这种情况下，我们可以将来自偶数列的数据存储在一个BRAM中，将来自奇数列的数据存储在另一个BRAM中。这是因为展开的循环总是执行一次偶数迭代和一次奇数迭代。

```
#define SIZE 8
typedef int BaseType;

void matrix_vector(BaseType M[SIZE][SIZE], BaseType V_In[SIZE], BaseType V_Out[SIZE]) {
#pragma HLS array_partition variable=M dim=2 cyclic factor=2
#pragma HLS array_partition variable=V_In cyclic factor=2
    BaseType i, j;
data_loop:
    for (i = 0; i < SIZE; i++) {
        BaseType sum = 0;
        dot_product_loop:
            for (j = 0; j < SIZE; j+=2) {
#pragma HLS pipeline II=1
                sum += V_In[j] * M[i][j];
                sum += V_In[j+1] * M[i][j+1];
            }
        V_Out[i] = sum;
    }
}
```

图4.14 内部循环展开2倍的矩阵向量乘法代码。

HLS工具可以使用 **unroll** 指令自动展开循环。该指令采用一个因子作为参数，它是一个正整数，表示循环体应该展开的次数。

使用 **array_partition cyclic factor = 2** 指令和将 M[][] 和向量 V_in[] 手动划分为单独的数组有着相同的效果。考虑一下应该如何修改代码才能更改访问模式呢？现在我们手动将循环展开为两倍。原始代码（没有数组分区和不展开），只执行数组分区的代码，同时执行数组分区和循环展开代码之间的性能结果有哪些不同呢？最后，使用指令执行数组分割和循环展开的结果与手动执行的结果相比有哪些不同呢？

在这段代码中，我们看到数组分区通常与流水线操作并行执行。通过2倍的数组分割可以使性能提高2倍，我们可以用将内环部分展开2倍或将外环的II减少2倍来实现。提升性能需要相应数量的阵列分区。在矩阵向量乘法中，这种关系相对简单，因为对内部循环中的每个变量只有一次访问权限。在其他代码中，关

系可能更复杂。无论如何，设计者的目标应该是确保例化的FPGA资源得到有效利用。一般情况下，将性能提高2倍将使用大约两倍的资源，相反将性能降低2倍可以节约一半的资源。

让我们来研究一下循环展开和阵列分区对性能和面积的影响，比较根据每秒的矩阵向量乘法运算（吞吐量）和根据展开阵列分区因子的数量的性能高低，并绘制相同的区域趋势图（如显示LUT，FF，DSP模块，BRAM）。再思考这两种情况的总趋势是什么？你会选择哪种设计？为什么？

4.6 Baseline实现

上一节中我们讨论了执行DFT的核心计算——矩阵向量乘法的一些优化方法。然而将矩阵向量乘法转移到功能完备的DFT进行硬件实现，还会出现其他问题。在本节中，我们将重点转移到DFT，并讨论如何优化以使其执行起来最有效。

处理大量复杂数字的能力是这一节我们需要着重考虑的点之一。如4.2节所述，因为S矩阵的元素是复数，所以实值信号的DFT几乎总是一个复数值信号。执行复数值信号的DFT以产生复数值结果也很常见。此外，我们需要处理分数或可能的浮点数据，而不是整数。这会增加实现的成本，尤其是需要执行浮点操作的时候。另外，浮点运算符（特别是加法运算符）比整数加法具有更大的延迟。这可以使得 $I = 1$ 的循环更难以实现。第二个变化是我们希望能够将我们的设计容量扩展到一个大输入矢量的大小，比如 $N = 1024$ 个输入样本。然而如果我们直接使用矩阵向量乘法，那么我们必须存储整个S矩阵。由于这个矩阵是输入大小的平方，因此这么大的一个输入矢量的存储实现起来比较困难。本章我们将讨论解决这两个复杂问题的技巧。

正如使用HLS创建硬件实现时的典型实例一样，让我们从简单代码的实现开始。这里我们有一个可以保证它具有正确的功能baseline代码。通常，这些代码都以非常连续化的方式运行；它们没有高度优化，因此可能无法达到所需的性能指标。但是，这是确保设计人员理解算法功能的必要步骤，并且可以作为未来优化的起点。图4.15显示了DFT的baseline实现。这使用双重嵌套for循环。内部循环将S矩阵的一行与输入信号相乘。该代码不是将S矩阵作为输入读取，而是基于当前循环索引，在内部循环的每次迭代中计算S中的元素。我们使用cos()和sin()函数将该相量转换为具有实部和虚部的笛卡尔坐标。该代码可将相量与适当采样的输入信号相乘并累加结果。经过该内环的N次迭代之后，每个S矩阵的列元素都被计算得出一个频域样本。外循环也迭代了N次，S矩阵的每一行都被迭代一次。最终，代码计算了N次矩阵W的表达式，但是cos()和sin()函数以及复数乘加计算进行了 n^2 次。

此代码使用函数调用来计算cos()和sin()值。Vivado HLS能够使用其内置的数学库来实现这样的数学计算。第3章介绍了用于实现三角函数（包括CORDIC）的几种可能的算法[22]。但是，要使这些函数生成精确的结果可能代价比较高。因为输入量不是任意的，取消这些函数调用也有几种方法。我们将在稍后更详细地讨论这些权衡方法。这个代码的顺序执行代码如图4.16所示。

```

#include <math.h>                                // Required for cos and sin functions
typedef double IN_TYPE;                         // Data type for the input signal
typedef double TEMP_TYPE; // Data type for the temporary variables
#define N 256                                     // DFT Size

void dft(IN_TYPE sample_real[N], IN_TYPE sample_imag[N]) {
    int i, j;
    TEMP_TYPE w;
    TEMP_TYPE c, s;

    // Temporary arrays to hold the intermediate frequency domain results
    TEMP_TYPE temp_real[N];
    TEMP_TYPE temp_imag[N];

    // Calculate each frequency domain sample iteratively
    for (i = 0; i < N; i += 1) {
        temp_real[i] = 0;
        temp_imag[i] = 0;

        //  $(2 * \pi * i)/N$ 
        w = (2.0 * 3.141592653589 / N) * (TEMP_TYPE)i;

        // Calculate the jth frequency sample sequentially
        for (j = 0; j < N; j += 1) {
            // Utilize HLS tool to calculate sine and cosine values
            c = cos(j * w);
            s = sin(j * w);

            // Multiply the current phasor with the appropriate input sample and keep
            // running sum
            temp_real[i] += (sample_real[j] * c - sample_imag[j] * s);
            temp_imag[i] += (sample_real[j] * s + sample_imag[j] * c);
        }
    }

    // Perform an inplace DFT, i.e., copy result into the input arrays
    for (i = 0; i < N; i += 1) {
        sample_real[i] = temp_real[i];
        sample_imag[i] = temp_imag[i];
    }
}

```

图4.15：DFT的baseline code

如果你要使用你设计的CORDIC（如从第3章开始），那么此代码需要做什么修改？改变CORDIC核心的准确性会使DFT硬件资源使用情况发生变化吗？它会如何影响性能？

请使用HLS实现DFT的基线代码，实现后查看报告，与乘法和加法相比，实现三元函数的相对成本是多少？对哪些操作尝试优化更有意义？通过流水线操作内循环可以实现什么性能？

4.7 DFT优化

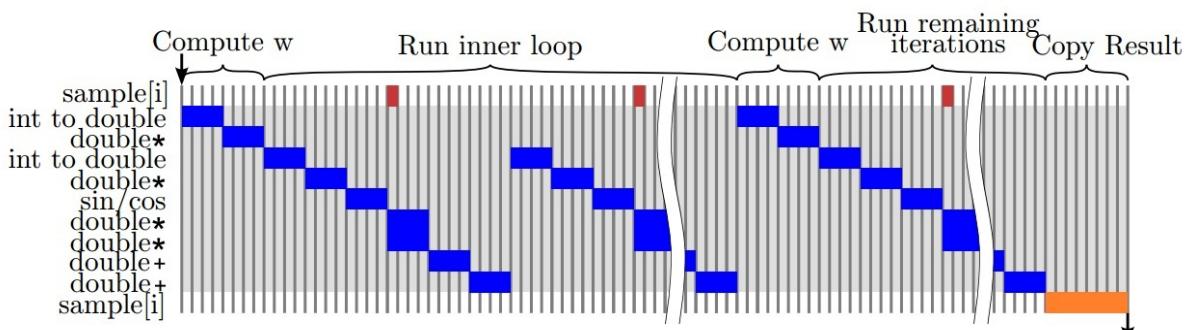
上一节的基线的DFT实现使用了相对较高的 **double** 数据类型。实现浮点运算尤其是双精度浮点运算通常代价很高并且需要很多流水线操作。我们可以从图4.16中看到这显着影响了循环的性能。通过流水线操作，这些高延迟操作的影响不那么重要，因为可以同时执行多个循环执行。此代码中的例外是用于累加结果的变量temp real []和temp imag []。这个累加是一种循环，并在流水线化内循环时限制了可实现的II。该运算符的依赖性如图4.17所示。

一种可能的解决方案是降低计算的精度。这种方法在实际应用时是有价值的，因为它减少了每个操作所需的资源，减少了存储值所需的内存，并且通常也减少了操作的延迟。例如，我们可以使用32位浮点型或16位半型来替代双精度型。许多信号处理系统完全避免了浮点数据类型，并使用定点数据类型3.5。对于常用的整数和定点精度，每个加法可以在一个循环中完成，从而使循环在II = 1处流水线化。

如果将所有数据类型从双精度型更改为浮点型，那么图4.15中的代码综合结果会发生什么变化？还是从一倍到一半？或到一个固定的点值？这是如何改变性能（间隔和延迟）和资源使用情况的？它是否会更改输出频率域采样值？

用浮点累加实现II = 1的普适解决方案是用不同的顺序处理数据。看图4.17，我们看到由于j循环是内部循环，所以重复是存在的（用箭头表示）。如果内循环是i循环，那么在下一次迭代开始之前我们就不需要累加的结果。我们可以通过交换两个循环的顺序来实现这一点。这种优化方式通常被称为循环交换或流水线交织处理[40]。由于外循环i内部的额外代码，我们可能不是很容易地发现可以重新排列循环。由于S矩阵是对角对称的，因此i和j可以在w的计算中交换。

结果是我们现在可以在内部循环中实现I的II。但是我们需要为临时实值和临时图像数组设置额外的存储器，存储计算的中间值一直到再次需要这些数据。



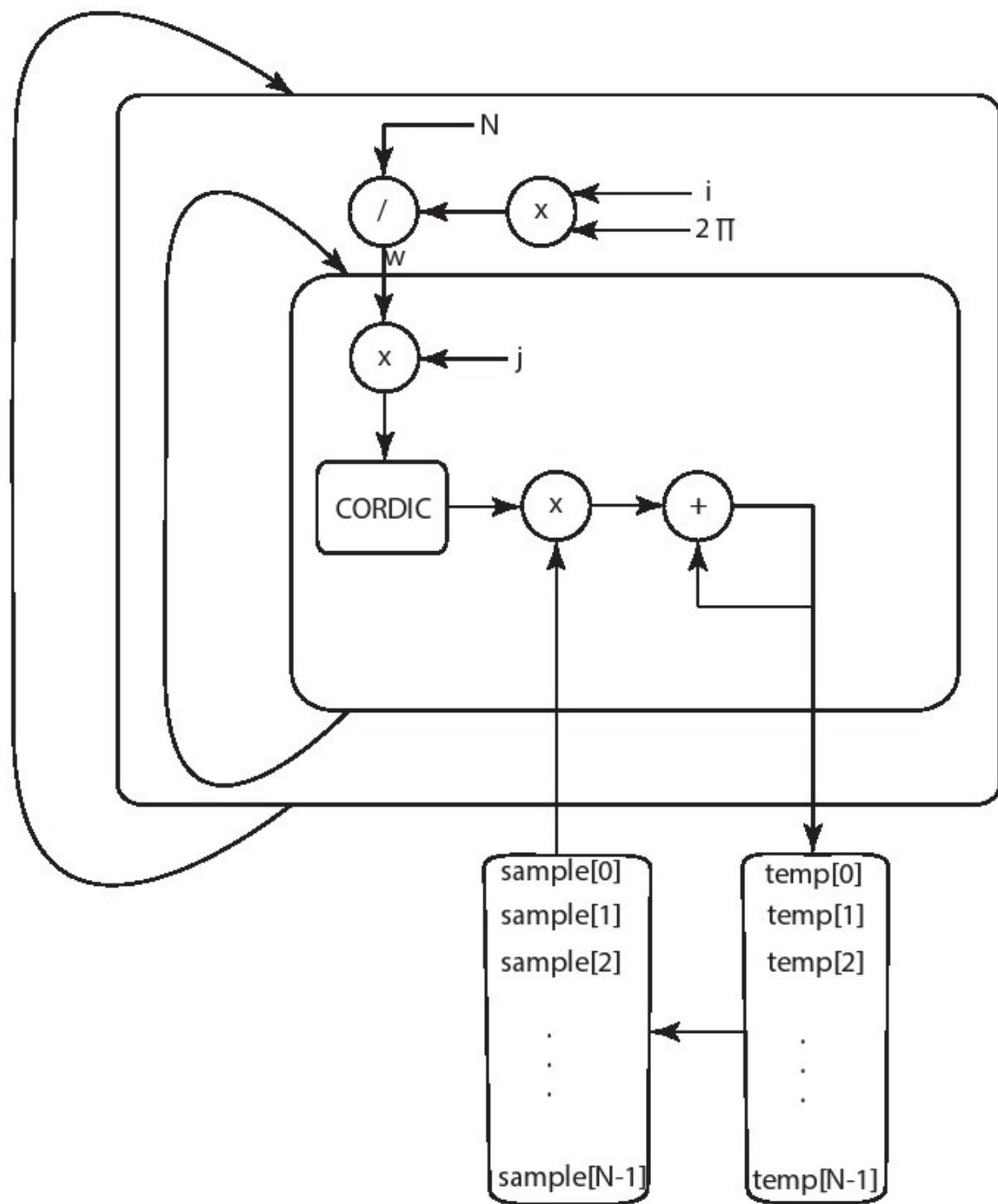


图4.16：DFT的高层体系结构图，如图4.15所示。这不是该体系结构的综合视图，例如，它缺少与更新循环计数器*i*和*j*的相关内容。该图想向读者提供关于如何合成这种体系结构的近似概念。这里我们假定浮点运算符需要4个时钟周期。.

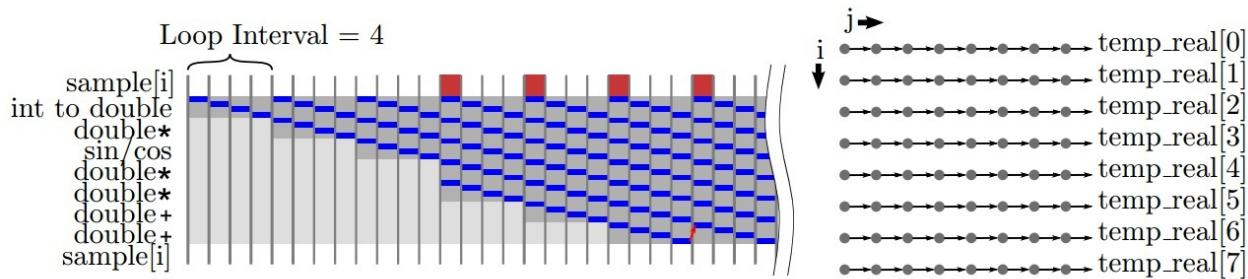


图4.17：图4.16中的行为的流水线版本。在这个设计案例下，由于每个浮点的添加需要4个时钟周期才能完成，并且在下一个循环开始之前需要上一个循环的结果加入计算（以红色显示相关性），所以设置循环的启动间隔为4个间隔。所有迭代的依赖关系汇总在右图中。.

重新排列图4.15中代码的循环，并显示您可以使用`i`的II来管理内部循环。

根据DFT中S矩阵的结构，我们可以应用其他优化方式来完全消除三角运算。回想一下，S矩阵的每个元素的复矢量是通过单位圆的固定旋转角度来计算的。S矩阵的行 $S[0][]$ 对应于单位圆周的零旋转，行 $S[1][]$ 对应于单次旋转，并且随后的行对应于单位圆周更大角度的旋转。我们可以发现，第二行 $S[1][]$ 相对应的向量覆盖了来自其他行的所有向量，因为8个列向量每个绕单位圆旋转 45° ，一共则围绕单位圆旋转了一圈。我们可以通过研究图9.11来直观地确认这个现象。这样我们可以只存储第二行这一次旋转中的正弦和余弦值，然后索引到这个存储器中的值以计算相应行的必要值。这只需要 $2 \times N = O(N)$ 个存储单元可以有效减少存储器的 $O(N)$ 的大小。对于1024个点的DFT，存储器的存储需求将减少到 1024×2 个条目。假设有一个32位的固定值或浮点值，则只需要8KB大小的片上存储器。与明确存储整个S矩阵相比，明显减少了存储容量。我们将矩阵S的这个一维存储表示为 s'

$$S' = S[1][.] = [1 \quad s \quad s^2 \quad \dots \quad s^{n-1}]$$

导出一维数组

为了进一步提高性能，我们可以应用一种与矩阵向量乘法非常相似的技术。之前我们发现了提高矩阵向量乘法的性能需要对 $M[][]$ 数组进行分区。但是如果用 s' 表示S矩阵则意味着不再有一种有效的方法来划分 s' 以增加我们在每个时钟上读取的数据量。S的每一个奇数行和列都包括 s' 的每个元素。因此，我们无法像对S一样对 s' 的值进行分区。这样增加我们从存储 s' 的内存中读取数据端口的数量的唯一方法是复制存储。幸运的是，不像与必须读取和写入的内存，复制只读的数组的存储是相对容易的。事实上，Vivado HLS将只对在初始化且从未例化的只读存储器 (ROM) 自动执行此优化。这种功能的一个优点是我们可以简单地将`sin()` 和`cos()` 调用移动到数组初始化中。在大多数情况下，如果此代码位于函数的开头并仅初始化阵列，则Vivado HLS能够完全优化三角函数计算并自动计算ROM的内容。

设计一个利用

为了有效地优化设计，我们必须考虑代码的每个部分。往往是最薄弱的环节决定了设计的整体性能，这意味着，如果设计有一个瓶颈，将显著影响设计的性能。当前版本的DFT可以对输入和输出数据进行就地操作，即它存储结果相同的数组作为输入数据，输入数组 `sample_real` 和 `sample_imag` 都是有效的存储器端口，也就是说，你可以把这些参数的数组存储在相同的存储位置。这样，在任意给定的周期，我们只能获取其中一个阵列的一个数据，这可能会在函数中并行的乘法和加法运算方面产生瓶颈。这就是我们为什么必须将所有的输出结果存储在一个临时数组的原因，然后将所有这些结果复制到函数结尾处的“`sample`”数组中。如果我们没有执行就地操作，则不需要这样做。

修改DFT函数接口，使输入和输出存储在单独的数组中。这会如何影响你的可以执行优化？它如何改变性能？区域结果如何？

4.8 结语

在本章中，我们探究了离散傅里叶变换（DFT）的硬件实现和优化方法。DFT是数字信号处理的基本操作，需要采样时域的信号并将其转换到频域。在本章的开头，我们描述了DFT的数学背景。这对于理解下一章（FFT）中所做的优化很重要。本章的其余部分集中介绍了指定和优化DFT以在FPGA上进行高效的实现。

由于DFT的核心是执行矩阵向量乘法，所以我们最初花费了一些时间来描述在执行矩阵向量乘法的简化代码上的指令级优化。这些指令级优化由HLS工具完成。我们用这个机会来阐明HLS工具执行指令优化的过程，希望如上的优化过程能让你直观地了解到工具优化的结果。

本章后节，我们为DFT提供了正确的功能实现方案，讨论了一些可以改善性能的优化，具体来说就是将系数阵列划分为不同存储器以提高吞吐量。阵列的分区优化通常是构建最高性能体系结构的关键方法。

第五章 快速傅里叶变换

当取样样本数量为N时，直接使用矩阵向量乘法来执行离散傅里叶变换需要 $\mathcal{O}(n^2)$ 次乘法和加法操作。我们可以通过利用矩阵中的常数系数的结构来降低运算的复杂度。在这里S矩阵负责对DFT的系数进行编码码。矩阵的每一行对应于绕复数单位圆周旋转的固定圈数（详情请参阅第4.2章）。这些运算值有大量的冗余，可以利用这些值来降低算法的复杂性。

这里使用的'Big O'符号描述了基于输入数据大小的算法复杂度的一般顺序。有关'Big O'符号及其在分析算法中的使用的完整描述，请参见[17](#)

快速傅立叶变换（FFT）使用基于S矩阵对称性的分块处理方法。FFT因Cooley-Tukey算法[16]而广为流行，它需要 $\mathcal{O}(n \log n)$ 次操作来计算与DFT相同的函数。这可以在大规模信号执行傅立叶变换时提供显着的加速。

计算DFT的分块处理的方法最初由Friedrich Gauss在19世纪初提出。然而，由于高斯的这部分研究并没有在他生前出版，而是出现在他去世后的一些收藏的文献中，所以这件作品一点都不含糊。海德曼等人[32]为FFT的研究史提供了一个很好的背景。

本章的重点是让读者更好地理解FFT算法，因为这是硬件设计优化的重要部分。首先，我们从数学的角度理解一下FFT；其次将讨论的重点放在如何缩小FFT的大小；最后，讨论一些不同的硬件实施方法。

5.1 背景

FFT算法利用DFT计算中的对称性来降低算法的复杂度。为了更好地理解这个问题，让我们首先关注一下DFT算法中的研究点，以2点DFT为例。DFT背景介绍执行一个矩阵向量乘法计算 $G[] = S[] \cdot g[]$ ，其中 $g[]$ 是输入向量， $G[]$ 是频域的输出数据， $S[]$ 是DFT参数。我们遵循第4.2章中所述相同的系数矩阵记法以及的输入和输出向量。

对于一个两个样本点的DFT，矩阵S的值为：

$$S = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \quad (5.1)$$

这里我们使用 $W = e^{-j2\pi}$ 的概念，其中W的上标表示添加到分子中的值，W的下标表示在复指数的分母值。举个例子， $W_4^{23} = e^{-j2\pi \cdot 2 \cdot 3 / 4}$ 。这和4.2节中讨论的DFT中s的值一样，其中 $s = e^{-j2\pi / N}$ ，s和W之间的关系为 $s = W_N$ 。

$e^{-j2\pi}$ 或者W经常被称作旋转因子，这个术语起源于1966年Gentleman和Sande的论文[27]。

$$\begin{bmatrix} G[0] \\ G[1] \end{bmatrix} = \begin{bmatrix} W_2^{00} & W_2^{01} \\ W_2^{10} & W_2^{11} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \end{bmatrix} \quad (5.2)$$

把这两个等式拓展到2点DFT则有：

$$\begin{aligned} G[0] &= g[0] \cdot e^{-j2\pi \cdot 0 \cdot 0 / 2} + g[1] \cdot e^{-j2\pi \cdot 0 \cdot 1 / 2} \\ &= g[0] + g[1] \end{aligned} \quad (5.3)$$

由于 $e^0 = 1$ ，所以其中的第二个频率项为：

$$\begin{aligned} G[1] &= g[0] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 0}{2}} + g[1] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 1}{2}} \\ &= g[0] - g[1] \end{aligned} \quad (5.4)$$

其中 $e^{\frac{-j2\pi \cdot 1 \cdot 1}{2}} = e^{-j\pi} = -1$ 。

图5.1为这个计算提供了两种不同的表示方法。a) 部分是2点DFT的数据流图。这是我们熟悉的本书的一种习惯的表达方法。b) 部分展示了用于相同计算的蝶形结构。这是数字信号处理中使用的典型结构，特别是用于表示FFT中的计算。

蝴蝶结构是一种更紧凑的表示形式，可用于表示大型数据流图。当两条线合在一起时，这表示一个加法操作。箭头线上的任何标签都表示该标签乘以该线上的值。该图中有两个标签：底部水平线上的'-'标志表示该值应该被否定。此后加上由相交的两条线表示的加法与减法相同。第二个标签是 W_2^0 。虽然这是乘法是不必要的（因为 $W_2^0 = 1$ ，这意味着它乘以值'1'），但是把它表示出来是因为它在多样本点的FFT计算中很常见。

现在让我们考虑一个稍微大一点的DFT——4点DFT，有4个输入、4个输出以及 4×4 的S矩阵如式5.5所示：

$$S = \begin{bmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{bmatrix} \quad (5.5)$$

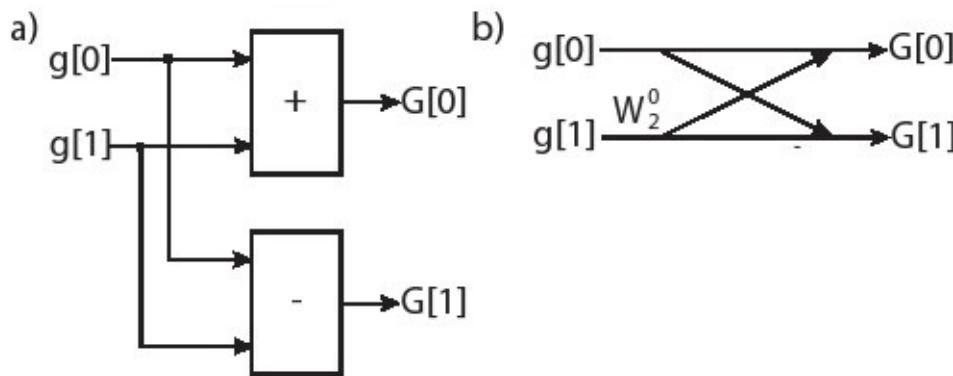


图5.1 a) 部分是2点DFT/FFT的数据流图。b) 部分显示了该运算的蝴蝶结构。它是数字信号处理领域中计算FFT的常见表示方法。

用于计算频率输出项的DFT方程为：

$$\begin{bmatrix} G[0] \\ G[1] \\ G[2] \\ G[3] \end{bmatrix} = \begin{bmatrix} W_4^{00} & W_4^{01} & W_4^{02} & W_4^{03} \\ W_4^{10} & W_4^{11} & W_4^{12} & W_4^{13} \\ W_4^{20} & W_4^{21} & W_4^{22} & W_4^{23} \\ W_4^{30} & W_4^{31} & W_4^{32} & W_4^{33} \end{bmatrix} \cdot \begin{bmatrix} g[0] \\ g[1] \\ g[2] \\ g[3] \end{bmatrix} \quad (5.6)$$

现在我们逐个写出 $G[]$ 中每个频域值的方程。 $G[0]$ 的方程为：

$$g[0] \cdot e^{-j2\pi \cdot 0 \cdot 0} + g[1] \cdot e^{-j2\pi \cdot 0 \cdot 1} + g[2] \cdot e^{-j2\pi \cdot 0 \cdot 2} + g[3] \cdot e^{-j2\pi \cdot 0 \cdot 3}$$

$$\begin{aligned} G[0] &= g[0] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 0 \cdot 3}{4}} \\ &= g[0] + g[1] + g[2] + g[3] \end{aligned} \quad (5.7)$$

G[1]的方程为：

$$\begin{aligned} G[1] &= g[0] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 1 \cdot 3}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} + g[2] \cdot e^{\frac{-j4\pi}{4}} + g[3] \cdot e^{\frac{-j6\pi}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} + g[2] \cdot e^{-j\pi} + g[3] \cdot e^{\frac{-j2\pi}{4}} e^{-j\pi} \\ &= g[0] + g[1] \cdot e^{\frac{-j2\pi}{4}} - g[2] - g[3] \cdot e^{\frac{-j2\pi}{4}} \end{aligned} \quad (5.8)$$

其中方程简化的依据是 $e^{-j\pi} = -1$ 。

G[2]的方程为：

$$\begin{aligned} G[2] &= g[0] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 2 \cdot 3}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j4\pi}{4}} + g[2] \cdot e^{\frac{-j8\pi}{4}} + g[3] \cdot e^{\frac{-j12\pi}{4}} \\ &= g[0] - g[1] + g[2] - g[3] \end{aligned} \quad (5.9)$$

其中方程的运算是通过基于旋转的简化来完成的，例如， $e^{\frac{-j8\pi}{4}} = 1$ 以及 $e^{\frac{-12j\pi}{4}} = -1$ 。这两个例子都运用到了 $e^{-j2\pi}$ 等于 1。换句话说，任何具有 2π 旋转的复指数都是相等的。

最终 G[3] 的方程为：

$$\begin{aligned} G[3] &= g[0] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 0}{4}} + g[1] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 1}{4}} + g[2] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 2}{4}} + g[3] \cdot e^{\frac{-j2\pi \cdot 3 \cdot 3}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} + g[2] \cdot e^{\frac{-j12\pi}{4}} + g[3] \cdot e^{\frac{-j18\pi}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} - g[2] + g[3] \cdot e^{\frac{-j10\pi}{4}} \\ &= g[0] + g[1] \cdot e^{\frac{-j6\pi}{4}} - g[2] - g[3] \cdot e^{\frac{-j6\pi}{4}} \end{aligned} \quad (5.10)$$

我们尚未发现我们的简化和上一个周期有何联系，它由 $e^{\frac{-j18\pi}{4}}$ 开始，因为他们相差了 2π 个旋转周期所以可以简化或者说是等于 $e^{\frac{-j10\pi}{4}}$ 。旋转 2π 等于 1 即 $e^{\frac{-j8\pi}{4}} = 1$ 。最后，再旋转 π 个角度就相当于 -1，就变成了 $e^{\frac{-j6\pi}{4}}$ 。从另一个角度来看这个 $e^{\frac{-j6\pi}{4}} \cdot e^{\frac{-j18\pi}{4}}$ ，其中 $e^{\frac{-j4\pi}{4}} = 1$ 。我们没有将式 5.10 彻底简化是为了在以下等式中证明对称性。通过重新排序，我们可以将这四个方程视为：

$$\begin{aligned} G[0] &= (g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}} (g[1] + g[3]) \\ G[1] &= (g[0] - g[2]) + e^{\frac{-j2\pi 1}{4}} (g[1] - g[3]) \\ G[2] &= (g[0] + g[2]) + e^{\frac{-j2\pi 2}{4}} (g[1] + g[3]) \\ G[3] &= (g[0] - g[2]) + e^{\frac{-j2\pi 3}{4}} (g[1] - g[3]) \end{aligned} \quad (5.11)$$

从式 5.11 可以看出，几种不同的对称性开始出现。首先，我们可以将输入数据划分为偶数和奇数元素，即，对元素 g[0] 和 g[2] 进行类似的操作，对于奇数元素 g[1] 和 g[3] 也是如此。此外，我们可以看到在这些偶数和奇数元素上存在加法和减法对称性。在计算输出频率 G[0] 和 G[2] 期间，将偶数和奇数元素相加在一起。在计算频率 G[1] 和 G[3] 时，将偶数和奇数元素减。最后，每个频率项中的奇数元素乘以常数复指数 W_4^i ，其中 i 表示频率输出，即 G[i]。

我们来看一下括号里面的项，可以发现有 2 点 FFT。比如，我们先考虑一下括号里面的偶数输入项 g[0] 和 g[2]。如果我们将这些偶数项用 2 点 FFT 展开，那么低频分量（直流分量）就是 g[0] + g[2]（参见式 5.3），高频分量计算结果为 g[0] - g[2]（参见式 5.4）。2 点 FFT 同样适用于奇数项输入 g[1] 和 g[3]。

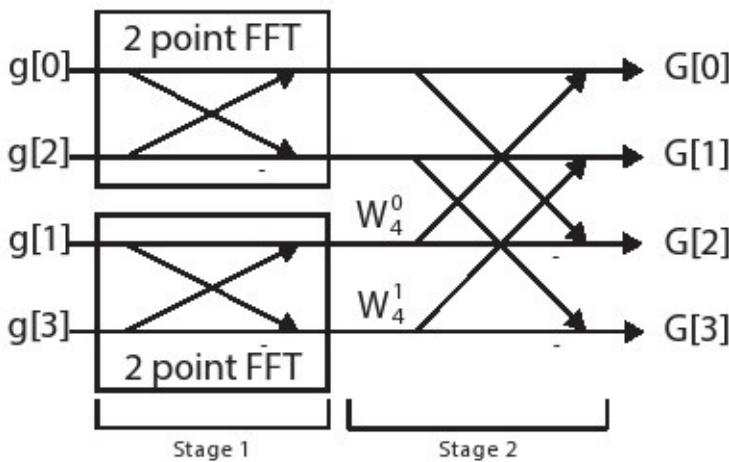
我们在这些等式上做更多的变换。

$$(g[0] + g[2]) + e^{\frac{-j2\pi 0}{4}} (g[1] + g[3])$$

$$\begin{aligned}
 G[0] &= (g[0] + g[2]) + e^{-j\frac{2\pi}{4}}(g[1] + g[3]) \\
 G[1] &= (g[0] - g[2]) + e^{-j\frac{2\pi}{4}}(g[1] - g[3]) \\
 G[2] &= (g[0] + g[2]) - e^{-j\frac{2\pi}{4}}(g[1] + g[3]) \\
 G[3] &= (g[0] - g[2]) - e^{-j\frac{2\pi}{4}}(g[1] - g[3])
 \end{aligned} \tag{5.12}$$

最后两个等式里的旋转变化是由 $e^{-j\frac{2\pi}{4}} = -e^{-j\frac{\pi}{2}}$ 和 $e^{-j\frac{2\pi}{4}} = -e^{-j\frac{3\pi}{4}}$ 两个等式得来的。我们可以跨两个等式，共享乘法项系数，进一步降低了矩阵乘法的复杂度。

图5.2显示了四点FFT的蝶形图。我们可以看到第一阶段是对偶数（顶部蝶形）和奇数（底部蝶形）输入值执行的两个2点FFT运算。通过使用公式5.12中所示的简化方式，将奇数的2点FFT输出乘以适当的旋转因子，可以用于所有四个输出项。



四点FFT分为两个阶段。阶段1使用两个2点FFT {一个2点FFT用于偶数输入值，另一个2点FFT用于奇数输入值。阶段2执行剩余的操作以完成FFT计算，如公式5.12所述。

我们看到这样的趋势使得从DFT的 $\mathcal{O}(n^2)$ 操作到FFT的 $\mathcal{O}(n \log n)$ 操作的复杂性降低。关键的想法是通过递归来构建计算。4点FFT使用两个2点FFT。扩展到更大的FFT尺寸就是8点FFT使用两个4点FFT，每个FFT使用两个2点FFT（总共四个2点FFT）。16点FFT使用两个8点FFT，依此类推。

在32点FFT中使用了多少个2点FFT？64点FFT有多少？64点FFT需要多少4点FFT？128点FFT怎么样？在N点FFT（其中N>8）中，2点，4点和8点FFT的通用公式是什么？

现在让我们来正式推导出这种关系，它提供了一种描述FFT递归结构的通用方法。假设我们正在计算N点FFT。给定输入值 $g[]$ 的频域值 $G[]$ 的计算公式为：

$$G[k] = \sum_{n=0}^{N-1} g[n] \cdot e^{-j\frac{2\pi kn}{N}} \text{ for } k = 0, \dots, N-1 \tag{5.13}$$

我们可以把这个公式分为两部分，一部分是偶数部分，一部分是奇数部分。

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{-j\frac{2\pi k(2n)}{N}} + \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{-j\frac{2\pi k(2n+1)}{N}} \tag{5.14}$$

该等式的第一部分处理偶数输入，是 $g[]$ 和 e 的指数中的 $2n$ 项。第二部分对应于两个部分的 $2n+1$ 的奇数输入。还要注意的是在两种情况下，由于我们将它分为两个部分所以每个公式的求和总和现在变为 $N/2-1$ 。

我们将公式5.14转换为如下：

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi kn}{N/2}} + \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi k(2n)}{N}} \cdot e^{\frac{-j2\pi k}{N}} \quad (5.15)$$

在第一个求和（偶数输入）中，我们只需将2移动到分母中，使其现变为N/2。第二个求和（奇数输入）使用幂规则来分离+1，留下两个复数指数。我们可以进一步表达这个等式为：

$$G[k] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi kn}{N/2}} + e^{\frac{-j2\pi k}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi kn}{N/2}} \quad (5.16)$$

这里我们只修改第二个求和。首先，我们在求和之外拉出一个不依赖于n的复数指数。我们也将第二个移动到分母中，就像我们在第一次求和中所做的那样。请注意，这两个求和现在具有相同的复指数 $e^{\frac{-j2\pi kn}{N/2}}$ 。最后，我们将其简化为

$$G[k] = A_k + W_N^k B_k \quad (5.17)$$

其中 A_k 和 B_k 分别是第一部分和第二部分的总和。我们回想一下 $W = e^{-j2\pi}$ 。它正是通过将偶数和奇数项分成两个求和项来完整地表述N点FFT。

让我们假设只用公式5.17来计算前N/2项，即G[0]到G[N/2-1]，再使用另一个等式导出剩余的N/2项，即从G[N/2]到G[N-1]的项。虽然这可能看起来很愚蠢（为什么要做更多的数学计算而且是不必要的？），但你会发现这将使我们能够利用更加对称的方式，得到一种我们在4点FFT研究中看到的那种计算模式。

为了计算更高频率G[N/2]到G[N-1]，让我们推导出相同的方程，但这次使用 $k=N/2, N/2+1, \dots, N-1$ 。因此，我们希望计算：

$$G[k + N/2] = \sum_{n=0}^{N-1} g[n] \cdot e^{\frac{-j2\pi(k+N/2)n}{N}} \text{ for } k = 0, \dots, N/2 - 1 \quad (5.18)$$

这类似于具有不同指数的公式5.13，即我们用公式5.13代替k，其中 $k+N/2$ 。用我们之前执行的相同变换集，我们可以将它直接移动到等式5.16，但用 $k+N/2$ 替换k，可得：

$$G[k + N/2] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{\frac{-j2\pi(k+N/2)n}{N/2}} + e^{\frac{-j2\pi(k+N/2)}{N}} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{\frac{-j2\pi(k+N/2)n}{N/2}} \quad (5.19)$$

我们可以减少求和中的复指数，如下所示：

$$e^{\frac{-j2\pi(k+N/2)n}{N/2}} = e^{\frac{-j2\pi kn}{N/2}} \cdot e^{\frac{-j2\pi(N/2)n}{N/2}} = e^{\frac{-j2\pi kn}{N/2}} \cdot e^{-j2\pi n} = e^{\frac{-j2\pi kn}{N/2}} \cdot 1 \quad (5.20)$$

第一次简化使用了幂规则来分割指数。第二次简化取消了在第二指数中的N/2项。最后的简化使用n是非负整数的事实，因此 $e^{-j2\pi n}$ 将始终是 2π 的倍数的旋转。这意味着该项始终等于1。

现在让我们来处理第二个复指数：

$$e^{\frac{-j2\pi(k+N/2)}{N}} = e^{\frac{-j2\pi k}{N}} \cdot e^{\frac{-j2\pi N/2}{N}} = e^{\frac{-j2\pi k}{N}} \cdot e^{-j\pi} = -e^{\frac{-j2\pi k}{N}} \quad (5.21)$$

第一次简化使用幂规则分割指数。第二次对第二个指数进行了一些简化。我们通过等式 $e^{-j\pi n} = -1$ 得到了最终形式。

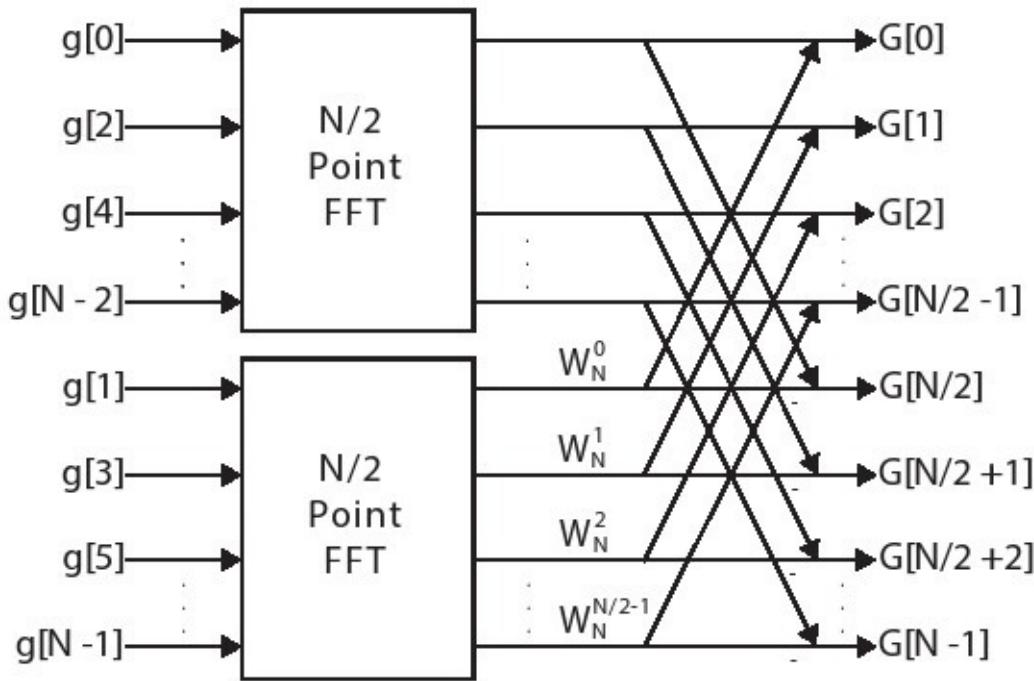


图5.3：从两个 $N/2$ 点FFT构建 N 点FFT。在偶数输入上执行上 $N/2$ 点FFT; 较低的 $N/2$ 点FFT使用奇数输入。

通过将式5.20和式5.21代入式5.19.我们得到

$$G[k + N/2] = \sum_{n=0}^{N/2-1} g[2n] \cdot e^{-j2\pi kn/N^2} - e^{-j2\pi k} \cdot \sum_{n=0}^{N/2-1} g[2n+1] \cdot e^{-j2\pi kn/N^2} \quad (5.22)$$

注意到它和式5.16很相似，我们把它带入式5.16可以得到：

$$G[k + N/2] = A_k - W_N^k B_k \quad (5.23)$$

我们可以使用公式5.17和5.23从两个 $N/2$ 点FFT来创建 N 点FFT。请记住， A_k 对应于偶数输入值函数， B_k 对应奇数输入值的函数。公式5.17涵盖了前 $N/2$ 项，公式5.23对应于较高的 $N/2$ 频率。

图5.3显示了从两个 $N/2$ 点FFT导出的 N 点FFT。 A_k 对应于顶部的 $N/2$ FFT， B_k 对应于底部的 $N/2$ FFT。输出项 $G[0]$ 到 $G[N/2-1]$ 乘以，而输出项 $G[N/2]$ 到 $G[N-1]$ 乘以 W_N^0 。注意，输入 $g[]$ 被分成偶数和奇数元素，分别馈入顶部和底部的 $n/2$ 点FFT。

我们可以使用通用公式来创建刚刚导出的FFT，以递归创建 $N/2$ 点FFT。也就是说， $N/2$ 个点FFT中的每一个可以使用两个 $N/4$ 点FFT来实现。并且每个 $N/4$ 点FFT可以使用两个 $N/8$ 点FFT，依此类推，直到我们达到最基本的2点FFT。

图5.4显示了一个8点FFT并突出显示了这种递归结构。带有虚线的方框表示不同的FFT大小。最外面的框表示8点FFT。这由两个4点FFT组成。这4个点FFT中的每一个都具有两个2点FFT，总共四个2点FFT。

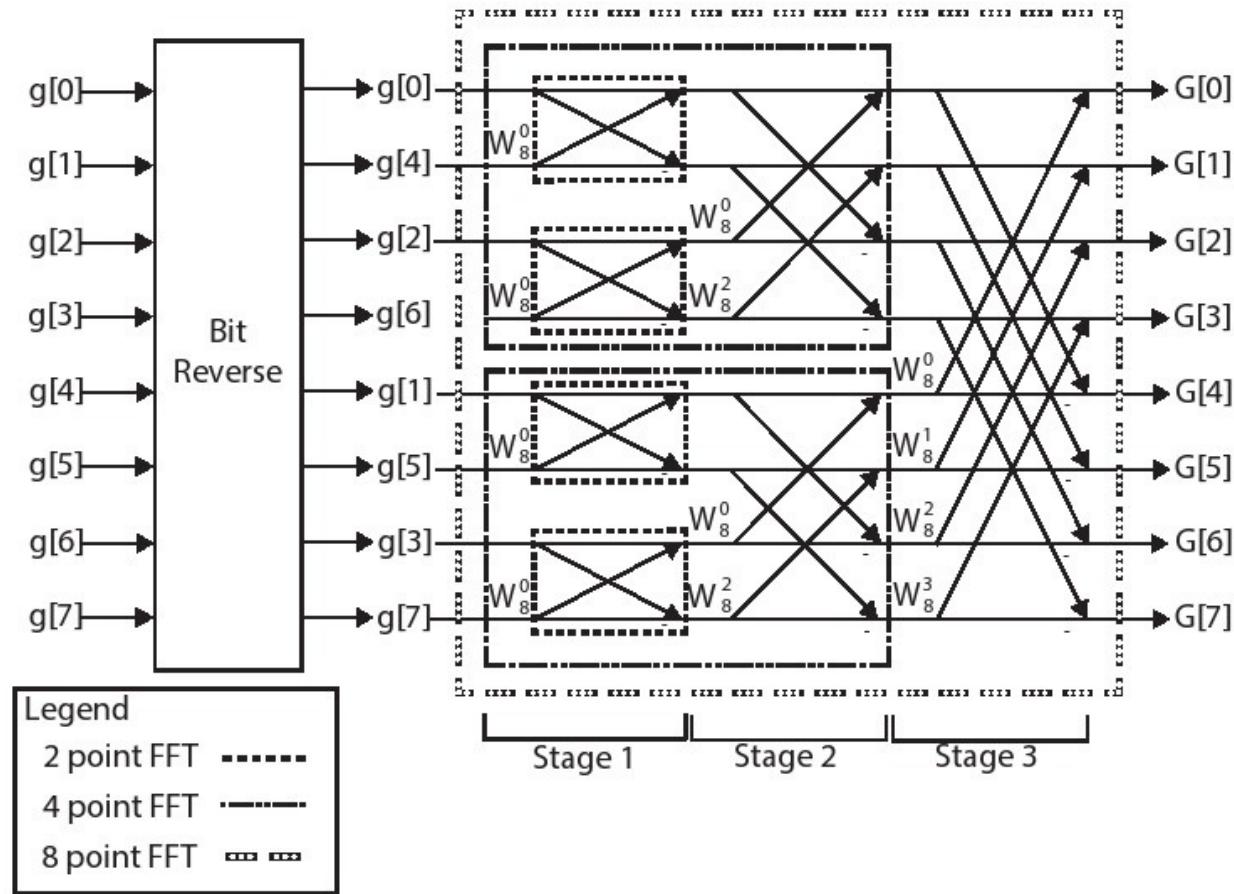


图5.4：递归构建的8点FFT。有两个4点FFT，每个FFT使用两个2点FFT。必须将输入重新排序为偶数和奇数元素两次。这导致基于索引的比特反转进行重新排序。

另外还要注意的是，输入量必须在输入8点FFT之前重新排序。这是因为不同的N/2点FFT采用偶数和奇数输入。上面的四个输入对应于偶数输入，下面的四个输入对应奇数输入。但是，它们被重新排序了两次。如果我们将偶数集分开，则将偶数和奇数输入分开 $\{g[0], g[2], g[4], g[6]\}$ 和奇数集 $\{g[1], g[3], g[5], g[7]\}$ 。现在让我们再次对偶数集重新排序。在偶数集 $g[0]$ 和 $g[4]$ 是偶数元素， $g[2]$ 和 $g[6]$ 是奇数元素。因此重新排序它为 $\{g[0], g[4], g[2], g[6]\}$ 。对于初始奇数集可以同样产生重新排序的集合 $\{g[1], g[5], g[3], g[7]\}$ 。

最终的重新排序是通过交换索引，将顺序排列的值按位取反来完成的。表5.1显示了索引及其三位二进制值。该表显示了8点FFT的8个索引，以及第2列中每个索引的相应二进制值。第三列是第二列的位反转二进制值。最后一列是对应反转二进制数的十进制数值。

表5.1：索引，该索引的三位二进制值，位反转二进制值，以及结果位反转索引。

索引	二进制	位反转二进制	位反转索引
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

查看第一行，初始索引0的二进制值为000，当反转时保持为000。因此，不需要交换该索引。由图5.4可知g[0]保留在同一位置。在第二行中，索引1具有二进制值001，反转时是100或4。因此，最初在索引1处开始的数据，即g[1]应该在第四位置结束。并且查看索引4，我们看到位反转值为1，因此交换g[1]和g[4]。

假设FFT是2的幂，则无论FFT的输入大小如何，该位反转过程都有效。FFT通常是2的幂，我们就可以递归地实现它们。

对于一个32点FFT，索引1的值应该和哪个位置的值交换呢？索引2的值呢？

这样我们就完成了对FFT的数学理论学习，还有很多有关如何优化FFT的细节。你可能认为我们花了太多时间讨论FFT的精细的知识点，这是一本关于FPGA的并行编程的书而不是数字信号处理的书。但这突出了创建最佳硬件实现的重要部分，设计人员必须很好地理解正在开发的算法。因为没有这个基础很难创建一个好的硬件实现算法。下一节将介绍如何创建良好的FFT硬件实现方法。

5.2 Baseline 实现

在本章我们将讨论使用Vivado HLS工具来实现 Cooley-Tukey FFT[16]算法的不同方法。这与我们在上一节中描述的算法相同。我们从代码的通用版本开始，然后描述如何对其进行重组以实现更好的硬件设计。

当顺序执行时，FFT中的 $\mathcal{O}(n \log n)$ 运算需要 $\mathcal{O}(n \log n)$ 个时间步长。通常，并行实现将并行执行FFT的某些部分。并行化FFT的一种常用方法是将计算组织为 $\log n$ 个阶段，如图5.8所示。每个阶段的操作都取决于前一阶段的操作，自然导致跨任务的流水线操作。这种架构允许以 $\log n$ 个FFT与由每个阶段的体系结构确定的任务间隔，来进行同时计算。我们将使用第5.4节中的dataflow指令讨论任务流水线。

FFT中的每个阶段也包含显著的并行性，因为每个蝶形计算独立于同一阶段中的其他蝶形计算。每个时钟周期执行 $n/2$ 蝶形计算，任务间隔为1，可以允许以1的任务间隔计算整个阶段。当与数据流架构结合使用时，FFT算法中的所有并行性都可以被利用。尽管可以构造这样的架构，但是除了非常小的信号之外几乎从不使用它，因为必须在每个时钟周期提供整个新的SIZE样本块以保持流水的充分利用。例如，以250 MHz运行的复杂32位浮点值的1024点FFT将需要1024点（8字节/点） $250 * 10^9 \text{ Hz} = 1\text{太字节}/\text{秒}$ 的数据到FPGA中。实际上，设计人员必须保证计算架构与系统中所需的数据速率相匹配。

假设时钟频率为250MHz，每个时钟周期接收一个采样，大约需要执行多少蝶形计算才能使用1024点FFT处理每个采样？16384点FFT怎么样？

在本节的其余部分中，我们提供了使用函数原型void fft (DTYPE X_R [SIZE], DTYPE X_I [SIZE]) 优化FFT的代码，其中DTYPE是用于表示输入数据的用户可定制数据类型。这可以是int，float或固定点类型。例如，# define DTTYPE int将DTYPE定义为int。请注意，我们选择在两个单独的数组中实现复数的实部和虚部。X_R数组保存实数输入值，X_I数组保存虚数值。X_R[i]和X_I [i]将第i个复数保存在单独的实部和虚部中。

我们在本节中描述了FFT实现中的一个变化，就是FFT对象由实数成了复数。虽然这似乎是一个重大变化，但核心思想保持不变。唯一的区别是数据有两个值（对应于复数的实部和虚部），运算操作（加，乘等）更加复杂一些。

此函数原型使用的是就地实现方法（in-place implementation）。也就是说，输出数据存储在与输入数据相同的阵列中。这样做消除了对输出数据的额外阵列的需要，减少了实现所需的内存量。但是，这可能会限制性能，因为我们必须读取输入数据并将输出数据写入相同的数组。如果可以提高性能，输出数据就可以使用单独的数组。资源使用和性能之间总是存在权衡，这里也是如此。最佳实现方案取决于应用要求（例如，高吞吐量，低功耗，FPGA大小，FFT大小等）

我们从FFT的代码开始研究，因为这是一个典型的软件实现代码。图5.5显示了一个嵌套的三个**for**循环结构。外部**for**循环，标记为**stage_loop**在每次迭代期间实现FFT的一个阶段。有 $\log_2(n)$ 个阶段，其中N是输入样本的数量。这些阶段在图5.4中清楚标明，该8点FFT具有 $\log_2(8) = 3$ 级。你可以看到每个阶段执行相同数量的计算或相同数量的蝶形运算。在8点FFT中，这个阶段有四个蝶形运算。

对于N点FFT，每个阶段有多少个蝶形运算？整个FFT一共有多少个蝶形运算？

第二个**for**循环，标记为**butterfly_loop**，执行当前阶段的所有蝶形运算。**butterfly_loop**有另一个嵌套**for**循环，标记为**dft_loop**。**dft_loop**的每次迭代执行一次蝶形运算。请记住，我们正在处理复数，并且必须执行复杂的加法和乘法。

dft_loop中的第一行确定了蝴蝶运算的偏移量。蝴蝶操作的“宽度”会根据阶段而变化。如图5.4所示，阶段1对相邻元素执行蝶式运算，阶段2对索引相差2的元素执行蝶式运算，阶段3对索引相差4的元素执行蝶式运算。计算这个差值并存储在*i_lower*变量中。我们可以发现存储在变量**numBF**中的这个差值在每个阶段都是不同的。

dft_loop中的剩余操作通过旋转因子和加法或减法操作来执行乘法。变量temp_R和temp_I在乘以旋转因子W之后保持数据的实部和虚部。变量c和s是W的实部和虚部，使用sin()和cos()计算内置功能。我们也可以使用CORDIC，例如第3章中开发的CORDIC，来更好地控制实现。最后，使用蝶式计算的结果更新X_R[]和X_I[]数组的元素。

dft_loop和**butterfly_loop**各自根据阶段执行不同的次数。然而，在一个阶段中执行**dft_loop**的总次数是恒定的。**butterfly_loop**中的**for**循环的迭代次数取决于该阶段中唯一W旋转因子的数量。再次参考图5.4，我们可以看到阶段1仅使用一个旋转因子，在这种情况下为 W_8^0 。阶段2使用两个独特的旋转因子，阶段3使用四个不同的W值。因此，**butterfly_loop**在阶段1中仅具有一次迭代，阶段2中具有2次迭代，并且在阶段3中具有四次迭代。类似地，**dft_loop**的迭代次数改变。它在阶段1中对于8点FFT迭代四次，在阶段2中迭代两次，在阶段3中仅迭代一次。然而在每个阶段中，**dft_loop**体总共执行的次数相同，对于一个8点DFT每个阶段执行总计四个蝶形运算。

```

void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE]) {
    DTYPE temp_R; // temporary storage complex variable
    DTYPE temp_I; // temporary storage complex variable
    int i, j, k; // loop indexes
    int i_lower; // Index of lower point in butterfly
    int step, stage, DFTpts;
    int numBF; // Butterfly Width
    int N2 = SIZE2; // N2=N>>1

    bit_reverse(X_R, X_I);

    step = N2;
    DTYPE a, e, c, s;

    stage_loop:
    for (stage = 1; stage <= M; stage++) { // Do M stages of butterflies
        DFTpts = 1 << stage; // DFT = 2^stage = points in sub D
    FT
        numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
        k = 0;
        e = -6.283185307178 / DFTpts;
        a = 0.0;
        // Perform butterflies for j-th stage
        butterfly_loop:
        for (j = 0; j < numBF; j++) {
            c = cos(a);
            s = sin(a);
            a = a + e;
            // Compute butterflies that use same W**k
            dft_loop:
            for (i = j; i < SIZE; i += DFTpts) {
                i_lower = i + numBF; // index of lower point in butterfly
                temp_R = X_R[i_lower] * c - X_I[i_lower] * s;
                temp_I = X_I[i_lower] * c + X_R[i_lower] * s;
                X_R[i_lower] = X_R[i] - temp_R;
                X_I[i_lower] = X_I[i] - temp_I;
                X_R[i] = X_R[i] + temp_R;
                X_I[i] = X_I[i] + temp_I;
            }
            k += step;
        }
        step = step / 2;
    }
}

```

图5.5：使用三个嵌套for循环的FFT的常见实现方法。虽然它在处理器上作为软件运行良好，但它对于硬件实现来说远非最佳。

Vivado HLS对每个合成函数执行重要的静态分析，包括计算每个循环可以执行的次数的界限。这些信息有许多来源，包括代码中的可变位宽，范围和**assert()**函数。当与循环II结合使用时，Vivado HLS可以计算FFT功能的延迟或间隔的界限。在某些情况下（通常当循环边界是可变的或包含条件结构时），该工具无法计算代码的延迟或间隔并返回“？”。
在合成图5.5中的代码时，Vivado HLS可能无法确定butterfly_loop和dft_loop迭代的次数，因为这些循环具有可变边界。**tripcount**指令使用户能够向Vivado HLS工具提供有关循环执行次数的更多信息，该工具可用于分析设计的性能。它需要三个可选参数min，max和average。在这段代码中，我们可以为dft_loop添加一个指令。通过应用此指令，

Vivado HLS工具可以计算循环和整体设计的延迟和间隔值的界限。由于Vivado HLS工具使用你提供的数字，如果你为该工具提供了错误的tripcount，则报告的任务延迟和任务间隔将不正确，出现垃圾输入和垃圾输出。

在图5.5中使用tripcount指令进行FFT的适当方法是什么？你应该设置min，max和average参数吗？如果FFT的大小发生变化，你是否需要修改tripcount参数？

5.3 位反转

我们还没有谈到位反转功能，它可以反转输入数据值，以便我们执行就地FFT。当输入值是混合的时候，使用该功能就使得输出数据的顺序是正确的。我们现在详细讨论这个功能。

图5.6显示了位反转功能的一种可能实现方法。它将代码分为两个函数部分。第一个是bit_reverse（位反转）功能，对给定数组中的数据进行重新排序，以便每个数据位于数组中的不同索引处。该函数调用另一个函数reverse_bit，它接受一个输入整数并返回该输入的位反转值。

让我们从reverse_bit函数的简要概述开始。该函数逐位通过输入变量并将其转换为rev变量。for循环体由几个按位操作组成，它们对输入的位进行重新排序。虽然这些操作不是非常复杂，但是这个代码的意图是将for循环完全展开，Vivado HLS可以识别输入的位，并简单地连接到输出。这样的话，reverse_bits函数功能的实现应该根本不需要逻辑资源，而只需要连线。这是通过展开循环而大大简化了必须执行的操作数量的情况。在不展开循环的情况下，必须按顺序执行各个“或”操作。虽然这个循环可以流水线化，但‘或’操作仍将在FPGA中使用纯逻辑实现，并且执行循环将具有由被反转的位数确定的延迟时间（在这种情况下为`\gls{fft}_BITS`）。

```

#define FFT_BITS 10           // Number of bits of FFT, i.e., log(1024)
#define SIZE 1024            // SIZE OF FFT
#define SIZE2 SIZE >> 1 // SIZE/2
#define DTTYPE int

unsigned int reverse_bits(unsigned int input) {
    int i, rev = 0;
    for (i = 0; i < FFT_BITS; i++) {
        rev = (rev << 1) | (input & 1);
        input = input >> 1;
    }
    return rev;
}

void bit_reverse(DTTYPE X_R[SIZE], DTTYPE X_I[SIZE]) {
    unsigned int reversed;
    unsigned int i;
    DTTYPE temp;

    for (i = 0; i < SIZE; i++) {
        reversed = reverse_bits(i); // Find the bit reversed index
        if (i < reversed) {
            // Swap the real values
            temp = X_R[i];
            X_R[i] = X_R[reversed];
            X_R[reversed] = temp;

            // Swap the imaginary values
            temp = X_I[i];
            X_I[i] = X_I[reversed];
            X_I[reversed] = temp;
        }
    }
}

```

图5.6：FFT实现的第一阶段重新排序输入数据。这是通过将输入数组中索引*i*处的值与对应于*i*的位反转索引处的值进行交换来完成的。函数reverse_bits给出与输入参数对应的位反转值，并且交换对应反转位中的数组中的值。

```

void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                 DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stage_one(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                    DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stages_two(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                     DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);
void fft_stage_three(DTYPE X_R[SIZE], DTYPE X_I[SIZE],
                      DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]);

void fft(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
    #pragma HLS dataflow
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];

    bit_reverse(X_R, X_I, Stage1_R, Stage1_I);
    fft_stage_one(Stage1_R, Stage1_I, Stage2_R, Stage2_I);
    fft_stages_two(Stage2_R, Stage2_I, Stage3_R, Stage3_I);
    fft_stage_three(Stage3_R, Stage3_I, OUT_R, OUT_I);
}

```

图5.7：代码将8点FFT分为4个阶段，每个阶段都是一个单独的功能。bit_reverse是第一阶段，8点FFT还有三个阶段。

当没有应用指令时，反向位功能的延迟是多少？循环流水线时的延迟是多少？整个函数流水线时的延迟是多少？

为了实现更好的设计，盲目地使用优化指令很诱人。然而，这可能适得其反。最好的设计师对应用程序和可用的优化都有深刻的理解，并仔细考虑这些以达到最佳效果。

现在让我们优化一下bit_reverse函数。此函数有一个for循环，它遍历输入数组的每个索引。值得注意的是有两个输入数组X_R[]和X_I[]。由于我们处理的是复数，我们必须存储实部（在数组X_R[]中）和虚部（在数组X_I[]中）。X_R[i]和X_I[i]保持第i个输入的实数和虚数值。在for循环的每次迭代中，我们通过调用reverse_bit函数找到索引反转值，并交换存储在索引i中的实数和虚数值。当我们浏览所有SIZE索引时，我们最终会浏览到每一个被反转的索引下的值。因此，代码仅根据if (i<reversed) 条件来执行索引下对应的数值的交换。

5.4 任务流水化

将FFT算法划分为多个阶段使Vivado HLS能够生成一种让算法的不同阶段在不同的数据集上运行的实现方法。这种优化方法使用dataflow指令启用此优化，称为任务流水。这是一种常见的适用于各种硬件优化的方法。

我们可以自然地将FFT算法划分为 $\log_2(N + 1)$ 级，其中N是FFT的点数。第一级交换输入数组中的每个元素，其中元素位于数组中的位反转地址。在该位反转阶段之后，我们执行蝶形运算的 $\log_2 N$ 阶段。这些蝶形级中的每一个具有相同的计算复杂度。图5.7描述了如何将8点FFT分成四个独立的任务。针对每个阶段

的任务代码具有单独的功能分别是：位反转、第一阶段、第二阶段和第三阶段。每个阶段有两个输入数组和两个输出数组：一个用于实部，另一个用于复数的虚部。假设DTYPE在别处已经定义好，例如作为int，float或定点数据类型。

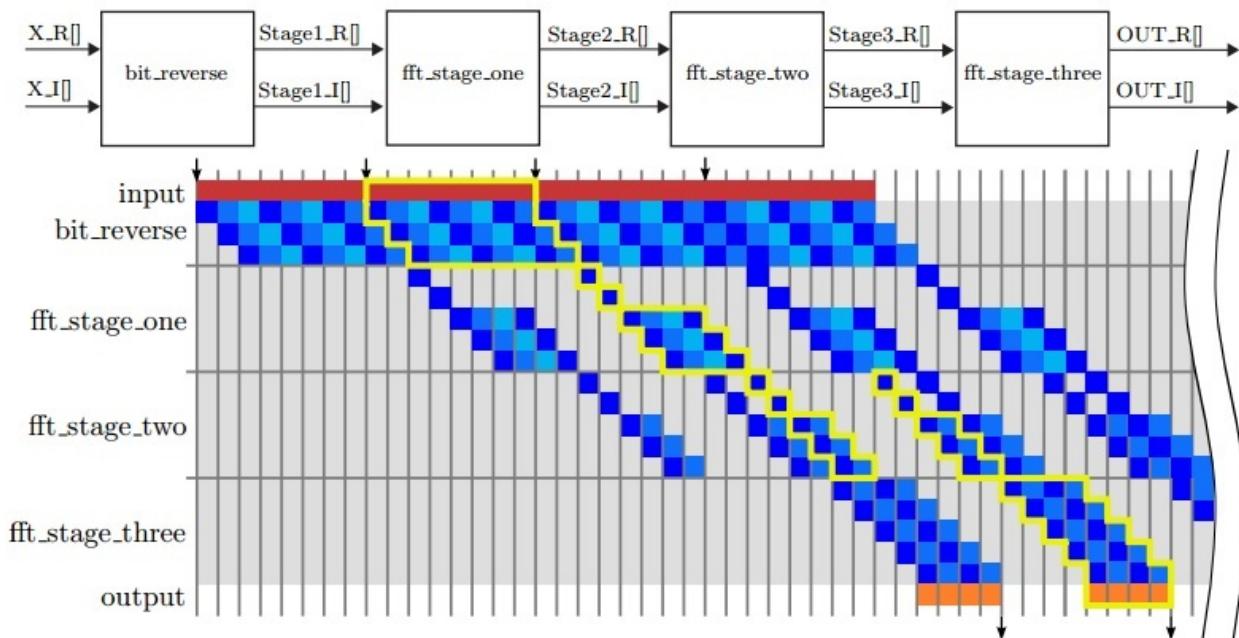


图5.8：该将FFT分成了不同的阶段允许跨越每个阶段的任务流水线。该图演示了同时执行4个具有三个FFT级（即，8点FFT）的示例。

重构FFT代码允许我们执行任务流水线操作。图5.8给出了一个例子。在这个操作中，我们允许第二个任务在第一个任务仅完成第一个函数bit_reverse之后就启动，而不是等待第一个任务完成代码中的所有四个函数再开始第二个任务。第一个任务继续按顺序执行流水线中的每个阶段，然后按顺序执行剩余的任务。一旦流水线已满，所有四个子功能同时执行，但每个子功能都在不同的输入数据上运行。类似地，同时计算四个8点FFT，每个点在硬件的不同组件上执行。图5.8的中间部分显示垂直四级中的每一级代表一个8点FFT。水平方向待变时间的增加。因此，一旦我们开始了第四个8点FFT，我们就有四个FFT同时运行。

dataflow 指令可以从函数和循环中构造出单独的流水线阶段（通常称为进程）。虽然图5.7中的代码仅使用函数，但我们可以使用四个循环而不是四个函数来实现类似的结果。实际上，我们可以通过直接使用`#pragma HLS unroll`指令在原始代码中展开外部stage_loop来得到这个结果。

dataflow 指令和**pipeline**指令都生成能够流水线执行的电路。关键的区别在于任务流水的粒度不一样。**pipeline** 指令构造了一个在循环级别上有效的流水线化的体系结构，由指令中的II所决定。运算符是静态调度的，如果II大于1，则可以在同一运算符上共享运算。**dataflow** 指令构造了一种体系结构，该体系结构可以有效地进行流水线操作，以获取（可能未知的）时钟周期数的操作，例如在数据块上运行循环的行为。这些粗粒度操作不是静态调度的，是通过流水线中的数据握手来动态地控制的。在FFT的情况下，每个阶段是对数据块（整个阵列）的操作，其需要大量的周期。在每个阶段中，循环都对块中的数据执行单独的操作。因此，在这种情况下，通常在顶层使用**dataflow**指令来形成粗粒度管道，并与每个循环中的**dataflow** 指令相结合，以形成对每个单独数据元素上进行细粒度流水操作。

dataflow 指令必须要有存储器设置以保证在不同进程之间传递数据。在Vivado HLS可以保证进程按顺序访问数据的情况下，它使用FIFO实现存储。这要求我们用与从数组中读取数组的顺序相同的顺序来写入数组。如果不是这种情况，或者如果Vivado HLS无法确定是否满足此流式传输条件，则可以使用乒乓操作来实现存储。乒乓操作由两个（或更多）概念数据块组成，每个数据块都是原始数组的大小。其中一个数据块可以由源进程写入，而另一个数据块由目标进程读取。术语“乒乓”就来自每次执行任务时对每个数据

块的读写都是交替的。也就是说，源进程将数据写入一个数据块，然后在开始之前切换到另一个数据块。下一个任务中，目标进程从生成器未写入的块中读取数据。因此，源进程和目标进程永远不能同时从同一个数据块写入和读取。

乒乓操作需要足够的内存来存储每个通信数组至少两次。FIFO通常可以占用更少的资源，尽管确定每个fifo的最小尺寸通常是一个困难的设计问题。但是，与FIFO不同，乒乓操作中的数据可以按任何顺序写入和读取。因此，当按顺序生成和使用数据时，FIFO通常是最佳选择，而当没有这种常规数据访问模式时，用乒乓操作来缓存数据是更好的选择。

有效地使用**dataflow**函数仍然需要优化每个单独进程的行为。流水线中的每个单独流程仍然可以使用我们之前看到的技术进行优化，例如代码重组，流水线操作和展开。例如，我们已经在5.3节讨论了bit_reverse函数的一些优化。通常，在考虑整体性能目标的同时优化各个任务非常重要。很多时候，最好从小功能开始，并了解如何隔离优化它们。作为设计人员，通常可以更容易理解一小段代码中发生的事情，并希望快速确定最佳优化方案。在优化每个单独的功能之后，我们就可以指定小功能的特定实现方式以优化的更大功能模块，你可以向上移动层次结构，最终到达顶部。

但是，必我们也须在总体目标范围内考虑局部优化。特别是对于数据流的一些设计，整个流水线的实现间隔永远不会小于每个单独过程的间隔。再看图5.7，假设位反转的间隔为8个周期，第一个阶段需要12个周期，第二个阶段需要12个周期，而第三个阶段需要14个周期。使用**dataflow**时，总的任务间隔为14，由所有任务或是函数的最大值决定。这意味着你应该小心平衡不同流程之间的优化，目标是创建一个平衡的流水线通道，其中每个流程的间隔大致相同。在该示例中，改善bit_reverse函数的间隔不能改善fft功能的整体间隔。实际上，如果可以用更少的资源实现整个FFT功能，增加bit_reverse函数的延迟可能是有益的。

5.5 结论

我们总体目标是创建一个满足应用目标的最优化设计。这个目标可能是创建一个资源使用最少的实现方式。或者，无论FPGA的大小、功率、能量限制如何，目标是创建一个能够实现最高吞吐量的方案。再或者，如果对应用程序有实时约束要求，传递结果的延迟可能也很重要。所有优化方法都会以不同的方式改变这些因素。

一般来说，没有一种固定的算法可以完成你设计的优化。它是应用功能，时序约束和设计者自身能力的一个综合体现。然而，设计人员必须深入了解应用程序本身，时序约束以及综合工具的各方面问题。

我们试图在本章中说明以上这些。虽然FFT是一种经过充分研究的算法，但由于它综合了大量已知的硬件实现技巧，所以仍然可以作为高级综合的良好示例。当然，我们也没有给出优化的所有技巧。无论如何，我们试图提供一些关于FFT关键优化的见解，我们希望这些优化可以作为你如何使用Vivado HLS工具来优化FFT的指南。

首先，我们要了解算法，所以花了很多时间来解释FFT的基础知识，以及它与DFT的关系。我们希望读者理解这些基础知识，因为这是构建最佳硬件实现方案的最重要部分。当然，设计人员可以将C/MATLAB/Java/Python代码转换为Vivado HLS并探寻有效的实现方式。同样，设计师可能会盲目地应用指令来获得更好的优化结果。但是，如果不深入理解算法，设计师就无法接近最佳结果本身。

其次，我们使用**dataflow**指令介绍任务级流水线。这是一项强大的优化方法，是通过代码重组无法实现的。也就是说，设计者必须使用此优化来获得这样的设计。因此，设计师必须了解它的功能，缺点和用法。

另外，我们基于前面章节中的一些优化方式构建了循环，例如，循环展开和流水线操作。所有这些对于获得优化的FFT硬件设计都很重要。虽然我们没有在这些优化上花费太多时间，但它们非常重要。

最后，我们试图向读者强调这些优化不能孤立地完成。有时优化是独立的，它们可以孤立地完成。例如，我们可以专注于其中一项任务（例如，我们在第5.3节中所优化的bit_reverse函数）。但很多时候，不同的优化会相互影响。例如，inline指令将影响函数的流水线操作。还有一点很重要，当我们优化任务或是功能时，可以通过功能层次从下向上传播。同时，设计人员也必须了解优化对本地和全局算法的影响。

第六章 稀疏矩阵向量乘法

稀疏矩阵向量乘（SpMV）把一个稀疏矩阵与一个向量相乘。稀疏矩阵是指矩阵中大部分元素为0的矩阵。这里的向量本身也可是稀疏的，但通常情况下是密集的。作为一种通用的运算，在科学应用、经济模型、数据挖掘、信息检索中广泛应用。例如，在利用迭代法求解稀疏线性方程组和特征值的问题。同时，也被应用于网页搜索排名和计算机视觉（图像重构等）。

本章会引入几个与HLS相关的新概念，并进一步深入之前讨论过的优化。本章的目标之一是引入一种更复杂的数据结构。我们用压缩行存储（CRS）来保存稀疏矩阵。另一个目标是演示如何进行性能测试。我们编写了简单的激励用来检验设计是否正确。这在硬件设计中十分重要，Vivado®HLS 工具采用HLS C 编写激励，并能轻松的对工具生成的RTL代码进行多方面的验证。这是基于HLS设计比基于RTL设计的巨大优势之一。章节中也会讲解如何采用Vivado®HLS工具进行C/RTL联合仿真。不同SpMV设计会带来性能上差异，因为执行时间和稀疏矩阵是密切相关的，所以我们必须通过输入数据来确定任务执行之间的间隔以及任务延迟。

6.1 背景

图6.1显示了一个 4×4 的矩阵M表示的2种方式。其中图6.1-a采用通用的二维方式16个元素来表示矩阵，每个元素存储在自己对应的位置上。图6.1-b采用CRS的方式表示相同的矩阵。**CRS**作为一种数据结构，由3个数组组成。值(values)数组保存矩阵中非零元素的值。列索引(columnIndex)数组和行指针 (rowPtr) 数组对非零元素的位置信息进行编码。列索引存储每一列的元素，行指针包含每一行第一个元素的值。**CRS** 结构避免存储矩阵中的0值，确实在数值数组中确实没有存储0。但是在这个例子中，虽然数值数组不保存0，但是列索引数组和行指针数组作为标记信息，表示了矩阵的形态。**CRS** 广泛用于大型的矩阵但是仅有有少量的非零元素（少于10%或者更低），这样可以简化这类矩阵的存储以及相关的运算。

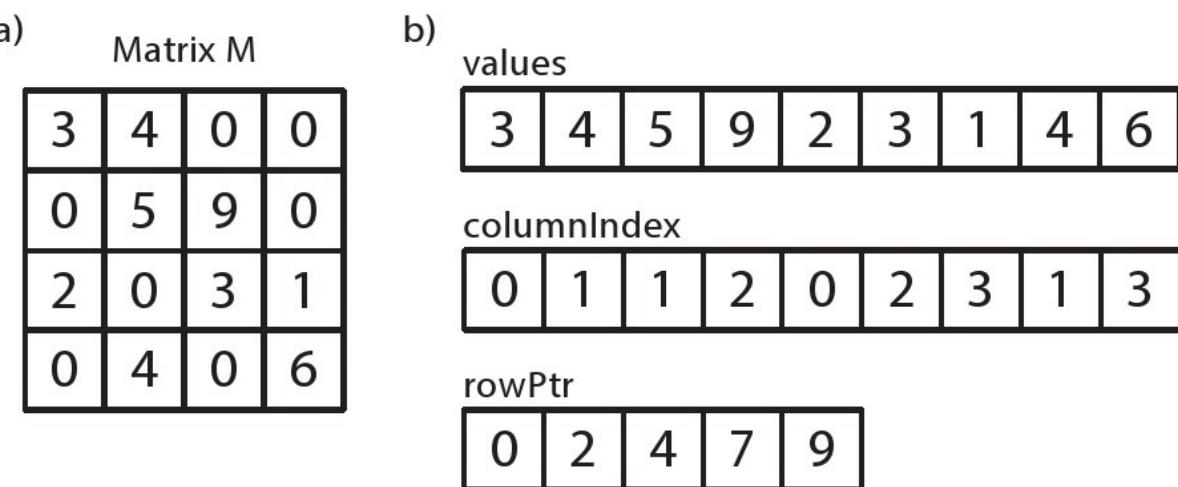


图 6.1: M 是一个 4×4 矩阵，用两种方式表示：同“密集”矩阵一样存在二维数组之中；作为稀疏矩阵，以行压缩存储的形式保存，行压缩存储是一种由3个数组组成的数据结构。

但是，CRS对矩阵的稀疏性没有要求，可以适用于任何矩阵。作为一种针对矩阵的通用方法，但不见得是最高效的方式。CRS结构也不见得是表示稀疏矩阵最高效的方式，其他稀疏矩阵表示方法也在被使用。

更准确的讲，CRS作为一种数据结构由3个数组构成：值(values)、列索引(columnIndex)、行索引(rowPtr)。值数组和列索引表示稀疏矩阵M中的每一个非零元素，这些数组表示矩阵M采用行的方式，从左到右，从上到下。矩阵中的数据保存在值数组中，列索引数组保存数据在数组中水平方向的位置，如果 values[k] 表示 M_{ij} 其中 $columnIndex[k] = j$ 。数组rowPtr用 $n + 1$ 的长度来表示n行矩阵。rowPtr[k] 表示在行 k 之前，矩阵中所有元素的数目，其中 $rowPtr[0] = 0$ 且最后一个元素rowPtr[k] 总是表示当前矩阵k行之前所有非零元素的个数 M_{ij} ，其中 $rowPtr[i] \leq k \leq rowPtr[i + 1]$ 。如果行k包含任何非0元素，那么rowPtr[k] 将包含当前行的第一个元素。注意，如果当前行没有非0元素，那么 rowPtr 数组中的值将会重复出现。

从图6.1 a) 中，我们可以行优先的方式遍历矩阵，从而确定值 (values) 数组在CRS中的形式。只要发现一个非0元素，它的值会被保存在下一个索引 i 中，同时，它的列号columnIndex[i] 会被保存在列数组中。另外，在我们访问一个新行的时候，我们保存下一个值的索引 i 在rowPtr数组中。所以，rowPtr 数组的第一个元素总是0。从图 6.1 b) 中，我们可以把矩阵转换为二维数组表示的方式。第一步是根据rowPtr数组，确定每一行中非0 元素的个数。对行 i 而言，该行中元素的数目为 $rowPtr[i] - rowPtr[i + 1]$ 的差值。所以当前行的值可以从values数组values[rowPtr[i]] 开始，通过递归得到。在我们的示例矩阵中，因为前rowPtr 数组前2个元素是0和2，所以我们知道第一行有2个非0元素，即value[0] 和value[1]。第一个非0元素在values数组中,value[0] 是3。该值所对应的列号为1，因为columnIndex[0] = 0。以此类推，矩阵中第二行元素的个数为 $k \in [2, 4)$,第三行的元素个数为 $k \in [4, 7)$ 。最后，共有9个非0元素在矩阵中，所以rowPtr最后一个值是9。

```
#include "spmv.h"

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE]){
L1: for (int i = 0; i < NUM_ROWS; i++) {
    DTYPE y0 = 0;
    L2: for (int k = rowPtr[i]; k < rowPtr[i+1]; k++) {
        #pragma HLS unroll factor=8
        #pragma HLS pipeline
        y0 += values[k] * x[columnIndex[k]];
    }
    y[i] = y0;
}
}
```

图6.2: 主体代码演示了系数矩阵向量乘 (SpMV) $y=M.x$ 的计算。采用CRS的方式，通过rowPt*、columnIndex 和 value 保存矩阵M。第一个for循环通过迭代访问每一行，第二个for循环访问每一列，实现矩阵M中非0元素和向量中对应的元素相乘并保存值在向量y中。

给定一个二维数组表示一个矩阵，通过C代码实现矩阵CRS格式。编写对应的C代码实现将矩阵从CRS格式转化为二维数组的形式。

结果表明，通过采用CRS的方式，我们能高效的实现稀疏矩阵乘法，不需要将矩阵转化为二维形式。实际上，对于大型的矩阵仅仅只有一小部分非0元素，稀疏矩阵向量乘法会比第四章中讨论的密集矩阵向量乘高效很多。因为我们直接找到非0元素，并执行非0元素对应的运算。

6.2 基本实现

图6.2 提供了基本代码对系数矩阵乘法的实现。函数`spmv`函数有5个参数，分别是`rowPtr`、`columnIndex`，以及`values` 对应矩阵 **M** 的 CRS 格式中包含的3个参数，这和图6.1中描述的数据结构等价。参数 **y** 用于保存输出的结果，参数**x**表示输入的被乘向量 **x**。变量`NUM_ROWS`表示矩阵**M**中行号。变量`NNZ`表示矩阵中非0元素的个数。最后，变量`SIZE`表示数组**x**和数组**y**中元素的个数。

外层`for`循环标签为**L1**，对矩阵的行进行遍历。将矩阵当前的行与向量**x**相乘，得到输出的结果**y**。内层循环标签为**L2**，实现对矩阵**M**中每列元素的遍历。L2循环迭代计算`rowPtr[i + 1] - rowPtr[i]`计算每一行非0元素的个数。每次循环计算，能从`value`数组中读取矩阵M的非0元素然后对应的从**x**数组中取得被乘向量**x**的值，对应相乘。`columnIndex[k]` 中的值保存了对应的列号**k**。

```
#ifndef __SPMV_H__
#define __SPMV_H__

const static int SIZE = 4; // SIZE of square matrix
const static int NNZ = 9; //Number of non-zero elements
const static int NUM_ROWS = 4; // SIZE;
typedef float DTTYPE;
void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTTYPE values[NNZ], DTTYPE y[SIZE], DTTYPE x[SIZE]);

#endif // __MATRIXMUL_H__ not defined
```

图6.3： `spmv`函数和激励的头文件

6.3 测试平台

图6.4 展示了一个针对`spmv`函数测试平台。测试平台通过定义`matrixvector`函数，直接实现矩阵向量乘法，它不考虑矩阵是否为稀疏矩阵以及矩阵是否采用CRS方式表示。我们比较`matrixvector`函数输出和`spmv`函数的输出。

在通常的测试平台中，需要实现的函数都会有个“黄金”参考，作为用户期望综合的结果。测试平台会比较黄金用例的输出和通过Vivado®HLS综合的代码执行结果。最好的实践方式是，测试平台既可以用于黄金用例，也可用于被综合的代码。这样就保证了两者实现的正确性。

测试平台在主函数`main`中执行。这里我们通过设置`fail`变量初始化为0，当`spmv`函数的输出成结果与`matrixvector`函数输出结果不相同时，变量置1。定义与矩阵**M**相关的变量、被乘向量**x** 和输出结果**y**。对于矩阵**M**，即有普通模式，也有CSR模式（保存为`values`、`columnIndex`、`rowPtr`）。矩阵**M**的`value`如图6.1中所示，输出向量**y**有两种，其中`y_sw`数组保存`matrixvector`函数输出的结果，`y`数组保存`spmv`函数输出的结果。

在定义好所有的输入变量和输出变量之后，分别调用`spmv`函数和`matrixvector`函数并输入合适的数 据。接下来的`for`循环用于比较`y_sw`和`y`中的每一个对应的结果。如果其中一个不相同，则将`fail`标志置1。最后，程序会打印测试的结果并返回`fail`变量。

```

#include "spmv.h"
#include <stdio.h>

void matrixvector(DTYPE A[SIZE][SIZE], DTYPE *y, DTYPE *x)
{
    for (int i = 0; i < SIZE; i++) {
        DTYPE y0 = 0;
        for (int j = 0; j < SIZE; j++)
            y0 += A[i][j] * x[j];
        y[i] = y0;
    }
}

int main(){
    int fail = 0;
    DTYPE M[SIZE][SIZE] = {{3,4,0,0},{0,5,9,0},{2,0,3,1},{0,4,0,6}};
    DTYPE x[SIZE] = {1,2,3,4};
    DTYPE y_sw[SIZE];
    DTYPE values[] = {3,4,5,9,2,3,1,4,6};
    int columnIndex[] = {0,1,1,2,0,2,3,1,3};
    int rowPtr[] = {0,2,4,7,9};
    DTYPE y[SIZE];

    spmv(rowPtr, columnIndex, values, y, x);
    matrixvector(M, y_sw, x);

    for(int i = 0; i < SIZE; i++)
        if(y_sw[i] != y[i])
            fail = 1;

    if(fail == 1)
        printf("FAILED\n");
    else
        printf("PASS\n");

    return fail;
}

```

图6.4：一个简单`spmv`函数的简单测试平台。测试平台生成了一个用例，并且计算矩阵的向量乘法通过稀疏矩阵乘法(`spmv`)和非系数矩阵乘法(`matrixvector`)。

这个测试平台相对简单并且可能无法充分验证所有的输入都能正常输出。最主要的原因是，它仅仅只用了一个矩阵作为例子，相反，一个好的激励会测试许多矩阵。通常，会通过随机的方式产生输入的测试用例，并且重点测试边界用例。在这个例子中，我们不仅要保证值正确计算，同时保证通过加速器正确的被执行了，而且编译时间相关的parameter改变会在实现不同加速单元值折中。最关键的是，在相同的parameter上，我们能通过随机产生很多输入数据来进行测试。编译时间相关的参数每次发生变化，都需要我们重新编译代码。

创建一个复杂的激励来，通过随机数方式生成许多组测试数据。稀疏矩阵编译时间参数应该是可以修改的（例如，`SIZE`，`NNZ`等）。创建一个HLS综合脚本，在编译时间参数合理范围改变时，能执行代码很多次。

6.4 指定循环的属性

如果直接将上述代码进行综合，我们可以得到函数运行的时钟周期及资源占用率。但是，我们不能得到模块执行所需的时钟周期、任务执行的延迟和任务执行之间的间隔。因为这些都取依赖于输入数据，由**spmv**函数外部因素决定。最主要的因素是，内层循环执行的次数是由矩阵**M**中非0元素个数决定的。非0元素的个数在代码中是由常量**NNZ**决定的，虽然可以调用函数计算不同大小的矩阵，但是实际迭代次数是和输入数据相关的。另外，性能也会因为非0元素的分布、综合优化的约束产生不同。更复杂的是，迭代的次数由输入决定，许多可能的输入并没有被遍历。所以，对于工具而言，不通过复杂的分析和额外的信息，工具是不能知道**spmv**函数执行需要多少时钟周期。**Vivado®HLS** 工具也不能进行上述的分析。

spmv函数能正常工作的前提条件是什么？证明给定的前提条件，矩阵中每个非0元素实是不是在对应一次内层循的执行？

有几种方式能帮助工具进行性能的分析，其中一种方式就是想**Vivado®HLS**提供循环边界的额外信息。这可以通过使用**loop_tripcount** directive实现，它能让设计者指定最小、最大和平均迭代次数针对特定的循环。通过提供这些值，**Vivado®HLS** 能提供时钟周期级别的评估。

使用**loop_tripcount** directive 用变量指定循环的最小，最大和平均迭代次数，这样**Vivado®HLS** 工具能对当前设计时钟周期数目进行估计。这些不影响最后综合的结果，只会影响综合报告。

对**spmv**函数使用**loop_tripcount** directive，语法格式 `# pragma HLS loop_tripcount min=X, max=Y, avg=Z` 其中X，Y，Z正的常量。哪个循环需要使用directive?当改变参数（min、max和avg）以后，综合报告有什么不同？这对时钟周期有影响吗？这对资源占用有影响吗？

loop_tripcount 引导能帮助设计者对函数的性能有个原始的估计。这样能比较相同的函数通过使用不同的directives或者对代码本身重构。但是，这不能确定**min**、**max**和**avg** 参数。这也很难确定边界条件**min**和**max**的值。如果有测试平台，就有一种更准确的方式用于计算**spmv**函数执行的时钟周期数，那就是C/RTL协同仿真。

6.5 C/RTL 协同仿真

C/RTL 协同仿真能自动化测试**Vivado®HLS**工具生成的RTL代码，只需要在综合的时候提供测试平台。每次执行综合以后的代码和提供的测试平台，记录输入和输出结果。输入的值按照时钟转换成输入向量。这里的输入向量用于针对生成的RTL代码进行仿真，同时记录输出向量。更新综合后的代码，再次运行测试平台并保存输入和输出数据。测试平台如果返回值是0，则表示成功；若激励返回非0值，则表示失败。

C/RTL 协同仿真流程将**VIVADO®HLS**生成的**RTL**代码，通过C 测试平台，实现时钟周期级别的仿真。这样，就能准确对生成的**RTL**代码进行性能评估，即使性能与输入数据有关。被综合的函数运行周期最小值，最大值，平均值以及间隔在仿真完成以后都能准确的得到。

注意这些和时钟周期相关的参数是通过激励中测试数据得到的。所以，结果的质量和测试平台的质量息息相关。如果测试平台没有很好的对函数执行测试，那么结果将不准确。另外，输入测试向量都是基于理想的时序，不能反映模型实际工作时，外部接口对函数的影响。实际的性能可能会比仿真的要低，如果

执行过程中阻塞在输入数据或对外部存储的访问上。不过，对于循环边界调试时变量的情况，设计者可以通过协同仿真的方式确定时钟周期个数。

C/RTL协同仿真能提供循环边界是变量的函数的延迟。它反馈函数运行时延迟的最小值、最大值和平均值以及函数运行间隔。这些延迟和测试平台输入的数据是强相关的。

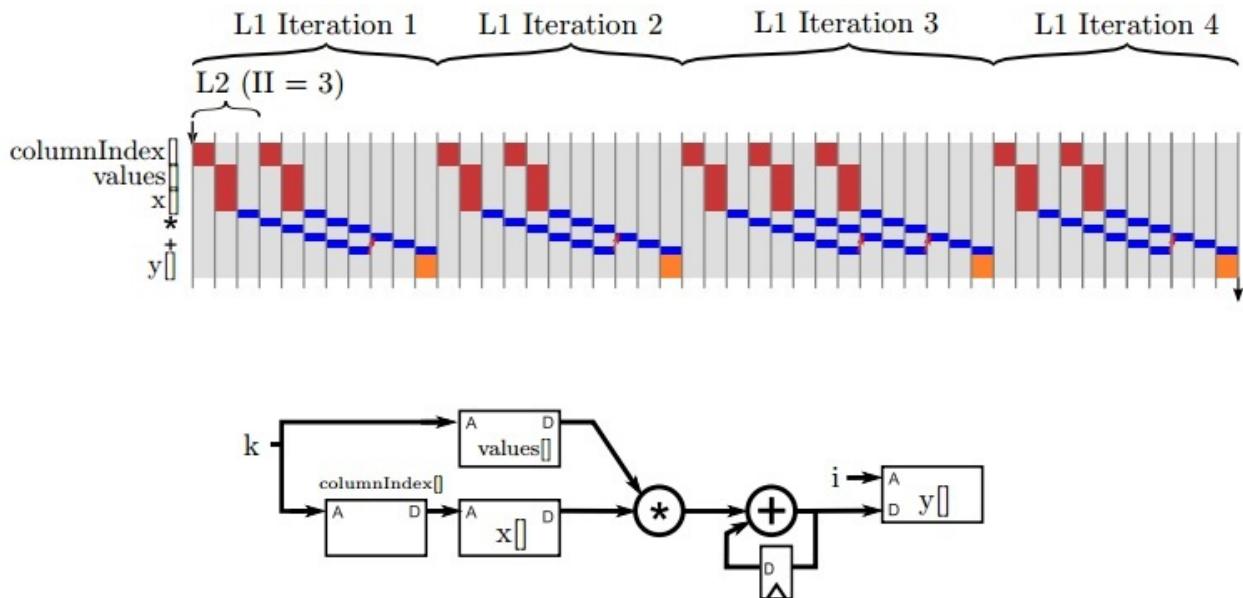


图6.5 spmv函数内部循环流水执行过程和结构

当采用图6.4提供的测试平台时，函数运行的最小值、最大值和平均值以及函数间隔是多少个时钟周期？

6.6 循环的优化与数组的分块

我们可以通过Vivado®HLS 工具得到当前函数的性能和面积的评估结果，然后可以考虑如何对函数进行优化。流水线、循环展开、数组分块是第一类最常用的优化方法。最典型的方式是从最内层的循环，然后根据需要向外层循环进行。

在这个例子中，对最内层的L2循环进行流水化也许是我们最先和最容易想到的优化方式。这个连续迭代的循环在执行上流水以后，总体运行会加快。如果不采用流水，L2 循环将按照串行执行。注意，L1 循环此时还是按照串行的方式执行。

图6.5演示了spmv函数在L2循环采用流水方式时运行的步骤。每次L2的循环都被 $II = 3$ 流水化。流水线允许在外层循环执行一次迭代时，内层循环执行多次循环迭代。此时，内层循环II受限于递归（recurrence）操作。 $II = 3$ 是因为我们认为加法器有3个时钟周期的延迟。外部循环没有采用流水的方式，所以内层的循环必须在下外层L2循环开始执行前，计算完成并输出结果。

对最内层的L2 for 循环进行流水化，通过在spmv函数中增加流水directive如图6.2所示。II(initiation interval)最后是多少？在你指定II的值以后，最终目标的II值是增大了还是减少了？

观察执行步骤，我们可以发现有几个因素限制了循环执行性能。第一个因素，递归（recurrence）操作限制了循环的**II**。第二个因素，外层的循环没有采用流水的方式。一种高效计算稀疏矩阵向量乘法的方式，每个时钟周期把乘法器和加法器使用起来。当前的设计离这个目标还很远。

在章节4.3中，我们探究了几种设计优化技术，其中包括对不同的循环进行流水，循环展开，数组分割。掌握在这些技术之间进行权衡是一项挑战，因为它们之间经常相互依赖。我们通常联合使用这些技术，为了得到好的性能谨慎的选择其中一种而不选择另一种也许结果会更糟糕。例如，在我们使用循环展开是，设计者需要明白它对数据访问的影响。增加了对数据访问的操作但是设计性能又受限于数据访问时，优化毫无益处。同样，如果提供了冗余的存储端口，实际中使用率不高，这样对提高性能毫无帮助反而增加了资源的消耗。

仔细思考一下上述优化技术组合后复杂多变的样式，我们建议你尝试下面的练习：

对spmv设计进行综合，采用表6.1提供的10种directives，每种都有不同的流水，展开和分割针对不同的循环和数组。这些分割在不同的数组（values、columnIndex、x）上使用。你看到结果的趋势是如何的？增加了展开和分割，是有利还是不利？面积？性能如何？为什么？

表6.1 稀疏矩阵向量乘法可优化的方式

	L1	L2
case1	-	-
case2	-	pipeline
case3	pipeline	-
case4	unroll=2	-
case5	-	pipeline,unroll=2
case6	-	pipeline,unroll=2,cyclic=2
case7	-	pipeline,unroll=4
case8	-	pipeline,unroll=4,cyclic=4
case9	-	pipeline,unroll=8
case10	-	pipeline,unroll=8,cyclic=8
case11	-	pipeline,unroll=8,block=8

如果你完成了上述练习，你会发现盲目的使用优化directives，可能不会得到你期望的结果。通常在设计时，在思考下考虑应用的特性，选择针对设计的特定优化方式。当然，这也需要一些直觉能力和一些专用工具投入使用。虽然，搞清楚像Vivado®HLS这样复杂工具中每一个细节是困难乃至不可能的，但是我们能基于关键的方面建立思考模型。

上面我们在用例3和4中考虑对外层循环L1进行流水化操作而不是对内层循环。这种变化针对一个任务，可以提高潜在的并行程度。为了完成优化，Vivado®HLS 工具必须展开代码中所有的内层循环L2。如果循环能全部展开，这样能减少计算循环边界的时间，同时也能消除递归（recurrences）。但是代码中的内层循环Vivado HLS是无法完全展开的，因为循环边界不是常量。

例如在实现上面提到的例子3，在最外层的循环**L1**使用流水化directive。在不设定目标**II**时，**II**值是多少？资源占用率发生了什么变化？增加了**II**后资源占用率结果如何？这与之前采对**L2**循环进行流水化，结果有什么不同？这和最基本的设计（无 directives）相比有什么不同？当你对外层循环进行展开时，结果到底如何？（提示：检查综合后的日志信息）

另外一种增加并行化的方式是对内层循环进行局部循环展开，就像之前例子5到10。这种变化实现更多的并行化，通过在相同的循环迭代中，执行更多的操作。有些情况，Vivado HLS 工具在对内层循环进行流水化时，通过实现更多操作来提高性能。但是，这还是很难提高内层循环的**II**，由于内层循环的递归操作。但是，在**II**大于1的情况下，许多操作可以共享同一个计算单元。

图6.6展示了一个局部展开的代码。在这段代码中，**L2**循环被分成2个循环，分别为**L2_1**和**L2_2**。最内层的循环**L2_2**执行的次数由参数**S**确定。内部循环包含了最原始的**L2**循环，其中循环边界是由最原始的**L2**循环确定的。代码中，**L2_1**循环包含了不确定次数的乘法和加法操作，运算次数由参数**S**确定，和一次递归完成累加 $y_0 + yt = yt$ 。

注意图6.6中的代码和自动循环展开的代码是由一点点区别的。自动循环展开复制计算，但是保留每次计算先后顺序（除了当前的例子）。这就导致了计算顺序由内层循环决定，如图6.7左所示。对计算顺序进行调整后，操作上的依赖关系如图6.7 左边所示。在当前的代码中，最后累加求和是一个递归（recurrence）。当使用浮点数据类型时，这种调整计算顺序的操作可能对程序产生改变，所以Vivado HLS对这种类型的代码不进行操作顺序自动调整。

这个设计可能会被综合、实现如图6.8所示的结果。在这个例子中， $S = 3$ 与**II**最匹配，乘法器的延迟正好是3。所有的运算过程都是在一个乘法器和加法器上执行。比较这个例子与图6.5中的例子，我们可以发现一些缺点。最明显的是，内层循环的流水线长度很长，实现的时候需要多个更多的周期刷新流水线的输出，才能执行下一次外层**L1**循环。处理一行中非零元素和执行块**S**相同。一行有个3个元素和一行有一个元素计算的时间是相同的。剩下的运算也需要在循环流水线中执行，即使他们的结果没有用。为了严格的比较两个设计的特性，我们需要了解设计对矩阵每行非零元素个数的预期。

```
#include "spmv.h"

const static int S = 7;

void spmv(int rowPtr[NUM_ROWS+1], int columnIndex[NNZ],
          DTYPE values[NNZ], DTYPE y[SIZE], DTYPE x[SIZE])
{
    L1: for (int i = 0; i < NUM_ROWS; i++) {
        DTYPE y0 = 0;
        L2_1: for (int k = rowPtr[i]; k < rowPtr[i+1]; k += S) {
#pragma HLS pipeline II=S
            DTYPE yt = values[k] * x[columnIndex[k]];
            L2_2: for (int j = 1; j < S; j++) {
                if(k+j < rowPtr[i+1]) {
                    yt += values[k+j] * x[columnIndex[k+j]];
                }
            }
            y0 += yt;
        }
        y[i] = y0;
    }
}
```

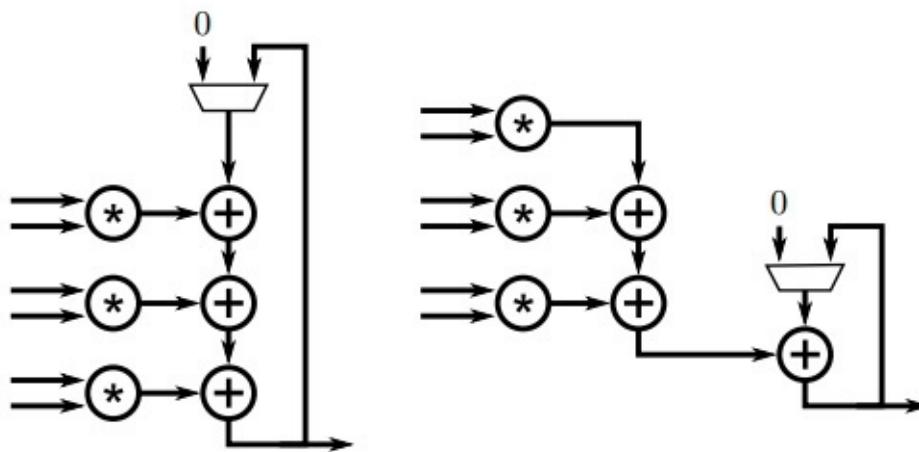
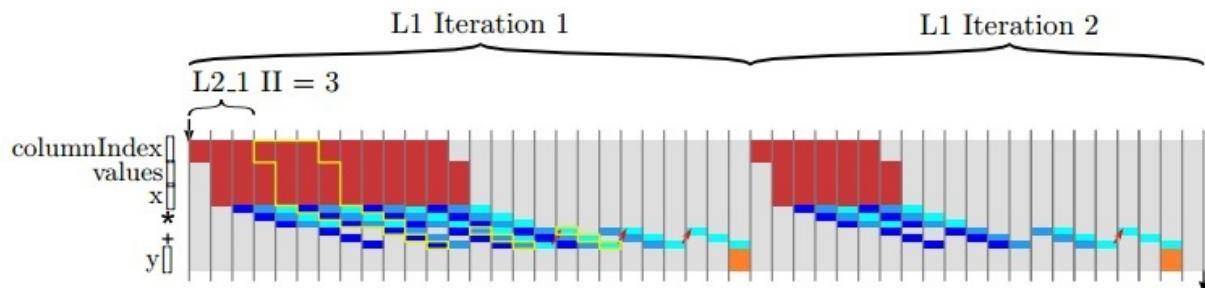
图6.6 局部展开图6.2中`smpv`函数

图6.7 针对累加的两种不同方式的局部展开。左边的版本有3个加法器进行递归操作，相反右边的版本只有1个加法器进行递归累加

图6.8 图6.6中`spmv`函数基于部分展开和内部流水线处理后执行过程

如果矩阵每行非零元素很少，则采用第一种实现方式较优；如果矩阵中每行非零元素较多，则第二种实现方式更好。

需要注意，这里存在一个关于先有鸡还是先有蛋的问题。我们需要知道目标器件和时钟周期，这样才能确定流水线中加法器能不能满足时序要求。只有在我们知道流水线的级数之后（也许S=1时，Vivado HLS才能识别到加法递归），我们才能选择合适版本的参数S，来满足 $II = 1$ 。一旦我们确定了S，我们能通过C/RTL协同仿真来，通过一组测试数据，确定是不是达到了性能上的要求。因为循环边界是可变的，所以得到的性能参数是依赖于数据的，所以我们需要设定不同的S，来找到性能的最大值。改变器件的类型和工作频率会影响之前所有的设计！进款看来去高层次综合（HLS）对解决问题提供的帮助不多，相比于RTL开发新版本然后进行验证，它开发起来快（代码编写方便）。

图6.8可以实现时，S与加法器流水线等级相同。如果S设定较大，结果会怎样？如果S设定较小，结果会怎样？如果目标II小于S会怎样？如果目标II大于S会怎样？

6.7 小结

在本章节中，我们介绍了系数矩阵向量乘法（SpMV），这延续了之前对矩阵运算的研究。SpMV 显得很有趣，因为它采用了一种特别的数据结构。为了减少大量的存储，矩阵采用行压缩的方式存储，这样就要求我们以一种非直接的方式对矩阵进行访问。

这一章节首先我们学习了Vivado®HLS工具测试和仿真的能力。我们采用一个基于SpMV简单的激励文件，讲解HLS工作流程。另外，我们对Vivado®HLS工具中C/RTL 协同仿真进行了讲解。这对我们得到设计准确性能结果是十分重要。矩阵越不稀疏，则更多的计算需要执行。在测试平台确定以后，协同仿真可以提供程序运行的精确仿真。这样就可以达到执行周期和性能结果。最后，我们讨论了采用循环优化和数组分块对代码进行优化。

第七章 矩阵乘法

本章节将讨论稍复杂的设计—矩阵乘法。我们考虑两种不同的版本。开始，我们用直接的方式实现，将两个矩阵作为输入，输出结果是它们的乘积。我们称这整体矩阵乘法。接下来，我们实现块矩阵乘法。这里，输入到函数的矩阵被分块，函数计算分块的结果。

7.1 背景

矩阵乘法是一个二元运算，两个矩阵计算结果为一个新的矩阵。矩阵乘法本身是采用构成矩阵的向量，进行的线性运算。最常见矩阵乘法称为矩阵积。当矩阵**A**维度是 $n \times m$ ，矩阵**B**维度是 $m \times p$ ，那么矩阵积**AB**则是一个 $n \times p$ 维的矩阵。

更准确的描述，我们这样定义：

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{bmatrix} \quad (7.1)$$

$$\mathbf{AB} = \begin{bmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{bmatrix} \quad (7.2)$$

其中**AB**_{ij}运算定义为：**AB**_{ij} = $\sum_{k=1}^m A_{ik} * B_{kj}$

现在，我们看一个简单的例子：

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} \quad (7.3)$$

这个矩阵乘法积结果是

$$\mathbf{AB} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} & A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} \\ A_{21}B_{21} + A_{22}B_{21} + A_{23}B_{31} & A_{21}B_{22} + A_{22}B_{22} + A_{23}B_{32} \end{bmatrix} \quad (7.4)$$

```

void matrixmul(int A[N][M], int B[M][P], int AB[N][P]) {
    #pragma HLS ARRAY_RESHAPE variable=A complete dim=2
    #pragma HLS ARRAY_RESHAPE variable=B complete dim=1
    /* for each row and column of AB */
    row: for(int i = 0; i < N; ++i) {
        col: for(int j = 0; j < P; ++j) {
            #pragma HLS PIPELINE II=1
            /* compute (AB)i,j */
            int ABij = 0;
            product: for(int k = 0; k < M; ++k) {
                ABij += A[i][k] * B[k][j];
            }
            AB[i][j] = ABij;
        }
    }
}

```

图7.1一个通用的3层for循环实现了矩阵乘法。

最外层的**for**，标签**rows** 和**cols**，通过迭代实现输出矩阵**AB**的行和列的遍历。最内层的循环，标签**product**完成**A**矩阵对应行与矩阵**B**对应列的乘和累加，最后的结果是矩阵**AB**中对应位置的一个元素。

矩阵乘法是数值运算的基础。计算大型矩阵之间的乘积会消耗大量的时间。因此，它在数值计算中一个非常重要的问题。根本上讲，矩阵运算是向量空间上的线性运算；矩阵乘法提供了一种线性变换的方式。这些应用有线性坐标变换（例如，变换，旋转在图形领域），统计物理中的高维矩阵（例如，矩阵论）和拓扑操作（例如，确定从一个定点到另一个定点，路径是否存在）。所以，它是一个值得被学习的问题，同时，有许多关于他的算法。这些算法目的是为了提高性能，减少算法对存储的需求。

7.2 计算矩阵乘法

我们从最常用的方法用于计算矩阵乘法—三层嵌套**for**循环开始，开始计算过程的优化。图7.1所示的代码实现了这种运算。最外层的**for**循环，标签为**rows**和**cols**，通过迭遍历了数据矩阵**AB**的行和列。最内层的**for**循环计算了向量**A**一行与向量**B**一列的点积运算。每个点积运算通过一些列的计算得到的结果只是矩阵**AB**中的一个元素。从概念上说，我们执行了**P**矩阵向量乘法，每个对应矩阵**B**的一列。

在这个例子中，我们对**col**循环采用了**pipeline** directive，并且设定期望任务间隔是1。这样，最内层的**for**循环的完全展开的，我们期望的电路大概要执行M次乘累加操作并且执行的间隔是N × P个时钟周期。正如第四章中讨论到的，这是最合理的方式。我们选择在函数不同的地方使用**pipeline** directive来实现在面积和速度之间的平衡。例如，当我们把相同的directive放在函数的最顶层（所有的**for**循环之外），那么所有的循环都会被完全展开，那么会有N × M × P次乘累加运算单元且运行间隔会是1。如果把directive放在内层**row**循环，那么会有M × P次乘累加单元并且执行间隔是N时钟周期。通过对相应的矩阵进行分块，上述设计要求和可以很容易满足。当然也可对最内层循环实行流水线，那么设计中只有个乘累加单元但是如果要达到II =1且运行时钟频率很高是不可能的，因为变量**AB_{ij}**累加是一个递归。我们也可以对不同的循环进行部分展开，来实现其他的设计要求。最根本的平衡还是在设计结构的复杂度例如乘累加单元的数目和性能例如硬件执行的效率。最理想的方式，当资源翻倍以后，对应所用的时钟周期减半，尽管这在实际中是很难实现的。

改变pipeline directive的位置。位置对资源的使用有什么影响？对性能的改变有什么作用？从函数间隔的角度考虑，那种选择能实现最好的性能？那种能提供最小的资源使用率？你认为在哪里放置directive最佳？如果矩阵的维度发生了变化，是不是你之前的选择也要变化？

每个时钟周期执行大量的运算，就需要为它们提供运算的对象，保存每个运算出处的结果。之前，我们使用了array_partition directive来提高在内存上可访问的次数。当数组的分块越来越小，每次存储的访问顺序在编译时就可以确定。所以数组分块是最简单和高效的方法用于提升在单位时间内，对内存的访问。这个directive不仅仅是将存储空间的地址分为不同的区间，也可以将多个存储空间合并以一个。这种变化存储空间数据位宽变大用于保存数组，但是不能改变总存储的比特数目。这其中的区别如图7.2所示。

array_reshape和**array_partition**都提高了一个时钟周期内可以读取的数组元素个数。它们支持相同的操作，可以循环和阻塞分块或者根据多维度的数据进行不同维度的分块。在使用**array_reshape**的时候，所有的元素在变换后的数组中共用同一个地址，但是**array_partition**变换后数组中地址是不相关的。看起来人们更喜欢使用**array_partition**因为它更灵活，每个独立的小空间更小，可能会导致存储资源使用不充分。**array_reshape** directive会形成大的存储块，这样可能更容易高效地映射到FPGA的资源上。特别是Xilinx Virtex Ultrascale+器件最小的block RAM (**BRAM**)是18 Kbits，可以支持不同深度和宽度的组合。当数组分块小于18kbits后，那么**BRAM**就没有被充分使用。如果我们使用最原始的矩阵，4-bit数组，维度是[1024][4]，这个数组就可放到一个单独的**BRAM**中，它配置为4Kbit x 4。对这个数组的第二个维度进行分块，那么将使用4个 1Kbit x 4的存储单元，每个都要比**BRAM**小很多。通过使用**array_reshape**将数组变为1Kbit x 16，可以被一个**BRAM**存储。

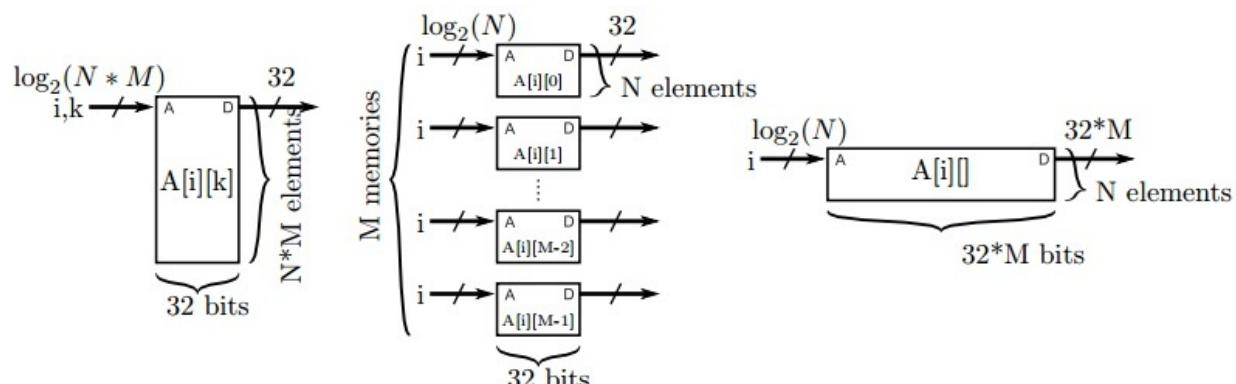


图 7.2:三种实现二维矩阵的方式。左边采用原始的方式，保存 $N \times M$ 个元素。中间数组使用了array_partition directive进行了变形，结果是使用了 M 个存储单元，每个保存 N 个元素。右边的，使用array_shape directive进行了变形，结果保存在一个存储单元，带有 N 个入口，每个入口保存原始矩阵的 M 个元素

注意这里的分块是有效的因为分块的维度（A 的列和 B 的行）是通过常量访问的。另外，不可分块的维度是通过相同变量访问的。当对多维矩阵进行分块时，这是一条很实用的经验用于确定哪个维度可以进行分块。

去掉array_reshape directive。这对性能有什么影响？这对资源使用率有什么影响？通过改变array_reshape的参数，对这些数组有没有影响？这种情况下，和只使用array_reshape有什么区别？

矩阵的形状可能对你进行的优化有潜在的影响。有些应用可能使用很小的矩阵，例如 2×2 或 4×4 。在这种情况下，可能期望设计实现最高的性能，通过在函数的入口直接使用**pipeline** directive。当矩阵的维度变大，例如 32×32 时，这种方法很显然不可实现因为单个器件上的资源是有限的。也可能没有足够的**DSP**资源在每个时钟周期内完成大量的乘法运算，或者没有足够的外部存储带宽用芯片和外部数据的交互。在系统中许多**FPGA** 设计需要和外部数据速率相匹配，例如模数转换 (A/D) 或 通信系统中的波特率。在这些设计中，通常使用**pipeline** directive对嘴内层循环，目的是为了实现计算间隔和系统数据速率匹配。在这些例子中，我们需要探索不同资源和速率之间的平衡，通过将**pipeline**放到内层循环或对循环部分展开。当处理非常大的矩阵包含成千上万的元素时，我们需要考虑更复杂结构。下一章节我们将讨论在大型设计中对矩阵进行缩放，称之为**blocking** 和**tiling**

去掉**array_reshape** directive。这对性能有什么影响？这对资源使用率有什么影响？通过改变 **array_reshape** 的参数，对这些数组有没有影响？这种情况下，和只使用**array_reshape** 有什么区别？

7.3 块矩阵乘法

块矩阵 是被分割出来子矩阵。直观的理解是，可以通过在水平方向和垂直方向画线对矩阵元素进行分割。得到的块作为原始矩阵的子矩阵。反过来，我们可以认为原始矩阵是块矩阵的集合。这就自然的派生出许多算法在线性代数运算中，例如矩阵的乘法，通过把大型的矩阵切分成许多小的矩阵，然后基于块进行运算。

例如，当我们讨论等式7.3和7.4 描述的矩阵**A**和矩阵**B**之间的乘法，可以认为矩阵中每一个元素 A_{11} 或 B_{23} 作为一个数或者一个复数。或者，我们也可以把它们当作是原始矩阵的一个块。在这种情况下，如果每个独立的块维度是匹配的，我们可以对块进行计算而不是对原始矩阵进行运算。例如，我们计算 \mathbf{AB}_{11} ，通过计算两个矩阵积和两个矩阵加得到 $\mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} + \mathbf{A}_{13}\mathbf{B}_{31}$

出于许多原因，矩阵分块是一种非常实用的计算方式。其中一个主要的原因是根据块，我们能发现更多的结果，探索更多的算法。实际上，之前我们已经采用的优化对循环进行变形，例如循环展开，可以看做是一种形式简单的分块。另外我们采用矩阵分块是基于矩阵天然的形式。如果矩阵有许多的0，那么许多小的乘积就是0。如果在流水线运算中统计这些0会很困难，但是如果跳过一个块0会比较简单。许多矩阵是**block-diagonal**，对角线上的块是非零，块周围的都是0。另外一个原因，分块将结果分解成为许多小问题和少量数据集合的求解。这样就增加了在本地完成数据的计算。在处理器系统中，很常见的看到选择合适的块和处理器存储结构匹配，或者是处理器支持的向量数据类型匹配。同理适用于FPGA，我们选怎合适的块大小与片上的存储资源相匹配，或者与片上乘累加单元数目匹配。

直到现在，我们都假设加速器总是有可用的数据在执行任务之前。但是，在设计处理大型数据表，例如大型的矩阵，这看起来不是必要的条件。因为让加速器能直接处理所有的数据是不可能的，我们能设计加速器，在他需要数据时才接收输入数据。这样就可以让加速器更高效的使用片上存储资源。我们称这位流水结构 (streaming architecture)，因为每次传输输入数据 (也可能是输出数据) 按次进行而不是一次处理所有的。

流水结构在应用中非常普遍。在有些设计选择时，我们将大型的计算分解为许多小的计算。例如，我们设计矩阵乘法系统，每次从外部读取和处理一个数据块。在其他的情况下，我们可能处理数据流因为数据是从物理环境中实时采样而来，例如，从A/D转换器。还有其他的例子，我们能处理的数据是来顺序的来自于之前的计算或加速单元。实际上，我们在5.4节已经见过这样的例子了。

流水的方式潜在的好处之一是减少了输入数据和输出数据对存储的需求。这里我们假设的是数据分块，得到分块的结果，完成了数据的处理，并且不需要存储。当下一个数据到达，我们在小的存储空间上对之前旧的数据进行改写。

接下来，我们设计一个流水结果用于计算矩阵乘法。我们将输入的矩阵A 和B 进行了分块，每个块包含一定的行和列。使用块，我们计算得到矩阵乘积AB 部分结果。接下来，我们采用流水的方式得到下一个块，计算AB 另外一个部分的结果，知道整个矩阵乘法计算完成。

图7.3提供描述了我们设计的流水结构。我们的结构中有变量**BLOCK_SIZE**用于指示每次执行从矩阵A 取到的行，从矩阵B取到的列，**BLOCK_SIZE*BLOCK_SIZE** 的结果矩阵与AB矩阵对应。

图7.3中的例子，设定**BLOCK_SIZE = 2**。所以每次我们执行时从矩阵A 取2行，从矩阵B取两列，按照我们设计的流水结构。通过调用**blockmatmul**函数得到的结果针对AB的一个2x2的矩阵。

当我们定义4x4的矩阵是，我们需要执行4次完成计算。每次我们得到一组2x2的结果是AB 矩阵的一部分。图展示了我们如何输入行和列数据。在图7.3 a)中，我们先输入矩阵A的头2行和矩阵B 的头2列，。函数计算一个2x2的矩阵对应结果矩阵AB的开始2行和2列的数据。

图7.3 b)中，我们仍然使用矩阵A的头2行，但是这次我们输入矩阵B最后的2列。我们不需要从矩阵A读取新的行，因为它和之前执行的数据是相同的。我们这次得到的2x2矩阵额结果对应的是矩阵AB “左上角”的数据。

图7.3 c)中，矩阵A 和矩阵B都需要输入新的数据。这次我们输入矩阵A最后2行的数据和矩阵B开始2列的数据。这次计算的结果对应矩阵AB“左下角”的值。

最后执行流水块矩阵乘法，如图7.3 d)所示，使用上一次迭代矩阵A最后2行的数据。输入矩阵B最后两列的数据。得到结果对用的是矩阵AB“右下角”的元素。

在展示矩阵分块成分的代码之前，我们定义我们可能用到的数据类型。图7.4展示了工程调用的头文件。我们自定义用户数据类型**DTYPE**，用于指示在矩阵A 和B相乘过程总用到的数据类型，与AB相对应。

使用用户自定义的数据类型是很好的编程实践。在未来的设计迭代中，设计中只用修改一处源文件，这样你能轻松的修改数据类型。根据设计的类型修改数据类型是非常常见的。例如，刚开始设计我们采用的是**float**或者**double**类型得到了正确结果。后期你可能使用定点化数据，这样在出错时能提供基本的框架。定点化数据类型减少了资源的使用，提高了性能，这些是以损失部分数据精度为条件的。你可能会尝试很多种定点类型的数据直到你找到在精度和准确率/错误率、性能和资源占用率之间的平衡。

a)

$$\begin{array}{|c|c|c|c|} \hline A_{11} & A_{12} & A_{13} & A_{14} \\ \hline A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ \hline A_{41} & A_{42} & A_{43} & A_{44} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline B_{11} & B_{12} & B_{13} & B_{14} \\ \hline B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ \hline B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline AB_{11} & AB_{12} & AB_{13} & AB_{14} \\ \hline AB_{21} & AB_{22} & AB_{23} & AB_{24} \\ \hline AB_{31} & AB_{32} & AB_{33} & AB_{34} \\ \hline AB_{41} & AB_{42} & AB_{43} & AB_{44} \\ \hline \end{array}$$

b)

$$\begin{array}{|c|c|c|c|} \hline A_{11} & A_{12} & A_{13} & A_{14} \\ \hline A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ \hline A_{41} & A_{42} & A_{43} & A_{44} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline B_{11} & B_{12} & B_{13} & B_{14} \\ \hline B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ \hline B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline AB_{11} & AB_{12} & AB_{13} & AB_{14} \\ \hline AB_{21} & AB_{22} & AB_{23} & AB_{24} \\ \hline AB_{31} & AB_{32} & AB_{33} & AB_{34} \\ \hline AB_{41} & AB_{42} & AB_{43} & AB_{44} \\ \hline \end{array}$$

c)

$$\begin{array}{|c|c|c|c|} \hline A_{11} & A_{12} & A_{13} & A_{14} \\ \hline A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ \hline A_{41} & A_{42} & A_{43} & A_{44} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline B_{11} & B_{12} & B_{13} & B_{14} \\ \hline B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ \hline B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline AB_{11} & AB_{12} & AB_{13} & AB_{14} \\ \hline AB_{21} & AB_{22} & AB_{23} & AB_{24} \\ \hline AB_{31} & AB_{32} & AB_{33} & AB_{34} \\ \hline AB_{41} & AB_{42} & AB_{43} & AB_{44} \\ \hline \end{array}$$

d)

$$\begin{array}{|c|c|c|c|} \hline A_{11} & A_{12} & A_{13} & A_{14} \\ \hline A_{21} & A_{22} & A_{23} & A_{24} \\ \hline A_{31} & A_{32} & A_{33} & A_{34} \\ \hline A_{41} & A_{42} & A_{43} & A_{44} \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline B_{11} & B_{12} & B_{13} & B_{14} \\ \hline B_{21} & B_{22} & B_{23} & B_{24} \\ \hline B_{31} & B_{32} & B_{33} & B_{34} \\ \hline B_{41} & B_{42} & B_{43} & B_{44} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline AB_{11} & AB_{12} & AB_{13} & AB_{14} \\ \hline AB_{21} & AB_{22} & AB_{23} & AB_{24} \\ \hline AB_{31} & AB_{32} & AB_{33} & AB_{34} \\ \hline AB_{41} & AB_{42} & AB_{43} & AB_{44} \\ \hline \end{array}$$

图 7.3: 一种可能的分块方式是讲矩阵分解为2个4x4的矩阵。整个AB 乘被分为4个矩阵乘法，分别是对A进行2x4个块，对B进行4x2个块

```

#ifndef _BLOCK_MM_H_
#define _BLOCK_MM_H_
#include "hls_stream.h"
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

typedef int DTTYPE;
const int SIZE = 8;
const int BLOCK_SIZE = 4;

typedef struct { DTTYPE a[BLOCK_SIZE]; } blockvec;

typedef struct { DTTYPE out[BLOCK_SIZE][BLOCK_SIZE]; } blockmat;

void blockmatmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols,
                  blockmat & ABpartial, DTTYPE iteration);
#endif

```

图7.4：分块矩阵乘法头文件的定义。文件定义了函数内部使用的数据结构，最关键的是，函数`blockmatmul`的接口。

SIZE 确定了每次矩阵相乘的行数和列数。这样就限制了矩阵类型为平方矩阵，但是我们可以通过修改代码中不同的地方来处理不同类型的矩阵。我们把这作为一个练习留给读者。 { % hint style='tip' %} 修改代码，实现对各种维度矩阵的支持。 { % endhint %}

变量**BLOCK_SIZE**定义了每次运算从矩阵**A**取的行数和矩阵**B**取的列数。这也定义了一次流水，函数能处理的数量。我们每次从函数得到的输出数据是矩阵**AB**的一个**BLOCK_SIZE x BLOCK_SIZE** 的。

数据类型**blockvec**是用于传递矩阵**A**的**BLOCK_SIZE**行和矩阵**B**的**BLOCK_SIZE**供函数每次执行。数据类型**blockmat**用于保存部分矩阵**AB**的结果。最后，数据类型**blockmat**是一个**BLOCK_SIZExBLOCK_SIZE** 的数组。它用于保存执行函数**matmatmul**计算的结果。

函数原型本身的两个输入类型是**hls::stream&**。它是一串**blockvec**类型的数据。记住，一个**blockvec**是由**BLOCK_SIZE**个元素构成的数组。

在Vivado®HLS 中，通过调用类**hls::stream<>** 创建FIFO类型的数据结构，可以很好的进行仿真和综合。数据通过**write()** 函数顺序的写入，通过**read()** 函数读出。这个库之所以被创建，是因为流水在硬件设计中，是一种常见的数据传递形式，并且这种操作可以用C语言进行多种方式的描述，例如，通过数组。实际上，Vivado®HLS 在处理负责的访问或多维数组时，很难推测出流水行为的。内建的流库方便程序员准确的指定流访问的顺序，避免在接口有任何限制。 { % hint style='info' %} **hls::stream** 只能通过引用的方式在函数之间进行传递，例如我们在图7.5中在函数**blockmatmul**中用法。 { % endhint %}

图7.5中的代码展示了通过流水的方式分块矩阵乘法。代码由3部分构成，分别标记为**loadA**、**partialsum** 和**writeoutput**。

代码的第一部分，标签为**loadA**，只要当 $it \% (SIZE/BLOCK_SIZE) == 0$ 程序才会执行。这样可以帮我节约时间，通过复用之前函数运行用到矩阵 **A** 的数据。

记住，每次函数**blockmatmul**执行的时候，我们从矩阵A取BLOCK_SIZE行，从矩阵B取BLOCK_SIZE列。矩阵A每个BLOCK_SIZE行对应多个BLOCK_SIZE列。变量it记录函数**blockmatmul**被调用的次数。所以，在每次执行函数时，我们可以判断是不是需要从矩阵A取新行。当不需要输入新行，这样就节省了我们的时间。当需要取新的数据，通过从**Arows**取数据存到一个静态的二维矩阵A[BLOCK_SIZE] [SIZE]中。

```
#include "block_mm.h"
void blockmatmul(hls::stream<blockvec> &Arows, hls::stream<blockvec> &Bcols,
    blockmat &ABpartial, int it) {
#pragma HLS DATAFLOW
int counter = it % (SIZE/BLOCK_SIZE);
static DTTYPE A[BLOCK_SIZE][SIZE];
if(counter == 0){ //only load the A rows when necessary
loadA: for(int i = 0; i < SIZE; i++) {
    blockvec tempA = Arows.read();
    for(int j = 0; j < BLOCK_SIZE; j++) {
#pragma HLS PIPELINE II=1
        A[j][i] = tempA.a[j];
    }
}
DTYPE AB[BLOCK_SIZE][BLOCK_SIZE] = { 0 };
partialsum: for(int k=0; k < SIZE; k++) {
    blockvec tempB = Bcols.read();
    for(int i = 0; i < BLOCK_SIZE; i++) {
        for(int j = 0; j < BLOCK_SIZE; j++) {
            AB[i][j] = AB[i][j] + A[i][k] * tempB.a[j];
        }
    }
}
writeoutput: for(int i = 0; i < BLOCK_SIZE; i++) {
    for(int j = 0; j < BLOCK_SIZE; j++) {
        ABpartial.out[i][j] = AB[i][j];
    }
}
}
```

图7.5：函数**blockmatmul**从矩阵A输入一系列大小为BLOCK_SIZE的行，从矩阵B输入一系列大小为BLOCK_SIZE的列，创建一个BLOCK_SIZExBLOCK_SIZE分布结果，作为矩阵AB的一部分。代码的第一部分（标记为`loadA`）保存A的行到局部存储，第二部分是嵌套的`partialsum for`循环执行部分结果的计算，最后的一部分（标记为`writeoutput`）将之前计算返回的分布结果放到合适格式中。

需要对**stream**类进行一些解释说明才能完全掌握这段代码，并且使用它。**stream**类型变量**Arows**由许多**blockvec**类型的元素构成。**blockvec**是一个BLOCK_SIZExBLOCK_SIZE的矩阵。每个**Arows**中的元素都是一个数组，用于保存矩阵A中BLOCK_SIZE行的数据。所以在每次调用**blockmatmul**函数时，**Arow**流会有SIZE个元素，每个用于保存BLOCK_SIZE行。语句`tempA = Arows.read()`从**Arows**流中取一个元素。然后，将这些对应的元素赋值给局部矩阵A中对应的位置。

stream类对操作符`<<`进行了重载，它等价于函数**read()**。所以，语句`tempA = Arows.read()`和`tempA << Arows`执行时等价的。

剩余部分的计算式是部分求和。大部分都是在函数**blockmatmul**中进行。

流类型变流类型量**Bcols**用法和变量 **Arrows** 相似。但是，它不是存储矩阵**A**的行，而是存储**B**矩阵参与当前运算的列数据。每次调用**blockmatmul**函数都会从矩阵**B**取到新的列数据。所以我们不需要像矩阵 **A** 那样，每次取数据时进行条件判断。函数本身执行步骤和图7.1中的**matmul**很相似，但是我们只计算 **BLOCK_SIZE x BLOCK_SIZE** 的结果与矩阵**AB**对应。因此我们每次通过迭代矩阵**A**中 **BLOCK_SIZE** 行和矩阵**B**中 **BLOCK_SIZE** 列。但是每行和每列都有 **SIZE** 个元素，因此循环的边界在外层**for**循环。

函数最后的部分实现数据到本地矩阵**AB**的赋值，之前矩阵维度是 **BLOCK_SIZE x BLOCK_SIZE**，对应的是输出矩阵**AB**的部分结果。

程序的三个部分中，中间部分用于计算部分求和，是对运算要求最高的部分。通过观察代码，可以发现这部分嵌套了 3 层**for**循环，迭代的次数为 **SIZE x BLOCK_SIZE x BLOCK_SIZE**。第一部分迭代的次数为 **SIZE x BLOCK_SIZE**，最后一部分迭代的次数为 **BLOCK_SIZE x BLOCK_SIZE**。所以，我们集中对程序中间部分进行优化，例如分部求和（**partialsum**）嵌套的**for**循环。

嵌套循环优化最开始的出发点都是对最内层的**for**循环进行流水化。如果这样操作需要耗费大量的资源，那么设计人员可以把 **pipelinedirective** 放到稍外层的**for**循环。是否导致设计耗费资源是由 **BLOCK_SIZE** 决定的，如果资源占用率很小，那么把 **pipelinedirective** 放在现在的位置是可行的。甚至把 **pipelinedirective** 放在最外层的**for**循环也是可行的。这样，内层的 2 层 **for** 循环都会展开，可能会导致资源使用量大幅上升。但是，这样性能也是有提升的。

改变 **BLOCK_SIZE** 是如何影响性能和资源占用率的？改变常量 **SIZE** 影响是什么？移动 **pipelinedirective** 在 **partialsum** 函数的嵌套的三层 **for** 循环之间，性能和资源占用率会发生什么变化？

函数开始部分的 **dataflowdirective** 实现函数部分之间的流水线，例如 **loadA** **for** 循环，**partialsum** 嵌套的 **for** 循环、**writeoutput** **for** 循环。使用这个 **directive** 可以减少函数 **blockmatmul** 运行的间隔。但是，代码三个部分中最大运行间隔是最终的限制条件。也就是说，函数 **blockmatmul** 运行间隔的最大值，我们称之为 **interval(blockmatmul)** 是要大约等于 **interval(loadA)** , **interval(partials)** , **interval(writeoutput)** 中的最大值的。

$$\max(\text{interval}(\text{blockmatmul}), \text{interval}(\text{loadA}), \text{interval}(\text{partials}), \text{interval}(\text{writeoutput})) \geq \text{interval}(\text{blockmatmul}) \quad (7.5)$$

当我们对函数 **blockmatmul** 进行优化的时候，我们需要牢牢记住等式 7.5。例如，假设 **interval(partials)** 比函数部分中其他两个都大。针对 **interval(loadA)** 和 **interval(writeoutput)** 的优化都是无用的，因为 **interval(blockmatmul)** 不会变小。所以，设计者需要将所有的注意力放在对 **interval(partials)** 的优化上，目标性能的优化就在嵌套的三层 **for** 循环上。

在进行性能优化时意识到这一点是非常重要的。开发人员可以对函数中其它两个部分进行资源优化。通常情况下这种减少资源占用率的优化都会提高执行间隔和延迟。在这个例子中，这种增加间隔不会对函数 **blockmatmul** 整个性能有影响。实际上最理想的方式，对函数的三个部分同时进行优化，它们有相同的运行间隔，这样我们就可以轻松的在运行间隔和资源占用上实现平衡（实际可能不总是这样）。

函数 **blockmatmul** 的测试平台如图 7.6 和 7.7。我们将它分成为两个图片，这样可读性更强，因为这段代码实在太长了。到目前为止，我们都是没有看到测试平台的。我们展示这个测试平台有一下几点原因。首先，它让我们知道函数 **blockmatmul** 的工作流程。特别是，它将输入的矩阵进行分块，然后一个块接一个块的输入给函数 **blockmatmul**。其次，它演示了如何在复杂的设中使用 **stream** 模块进行仿真。最后，他给读者一个如何建立合适测试平台的概念。

函数`matmatmul_sw`通过简单的三层**for**循环实现了矩阵乘法。它将2个二维的矩阵作为输入，输出是一个二维矩阵。这和我们之前在图7.1中看到的函数`matrixmul`很像。我们用这个函数的结果和硬件版本分块矩阵乘法进行做比较。

让我们先从图7.6开始，观察测试平台的前半部分。开始的代码部分完成对函数其余部分变量的初始化。变量`fail`用于记录矩阵乘法是否计算正确。在函数的后面会检查这个变量。变量`strm_matrix1`和`strm_matrix2`是`hls::stream<>`类型，分别用于保存矩阵A的行和矩阵B的列。这些流类型的变量中每个元素是一个`< blockvec >`。参考图7.4所示的头文件`block_MM.h`。我们反复调用定义的`blockvec`作为数组数据，我们用`blockvec`保存一行和一列数据。

变量`stream`在HLS的命名空间中。所以，我们使用对应的命名空间，并且加载`hls::stream`，或者直接简化用`stream`。但是，实际使用更倾向`zaistream`前保留`hls::`，可以提醒读者这里的`stream`是和Vivado®HLS相关的而不是来自于其他C库函数。同样，这样可以避免潜在与其他新命名空间产生冲突。

接下来在代码开始部分定义的变量是`strm_matrix1_element`和`strm_matrix2_element`。这两个变量用来作为每个`blockvec`变量分别写入流变量`strm_matrix1`和`strm_matrix2`的中间过度变量。变量`blockmatmul`用于保存函数`blockmatmul`的输出结果。注意这个变量使用`blockmat`类型，它是一个大小为`BLOCK_SIZE`×`BLOCK_SIZE`的二维矩阵，在头文件`block_mm.h`定义（见图7.4）。最后定义的是矩阵A、B、`matrix_swout`和`matrix_hwout`。它们都是大小为`SIZE`×`SIZE`的二维矩阵，其中数据类型都是`DTYPE`。

你可以对流使用初始化进行命名。这是一个很好的方式，因为它输出的错误信息更准确。如果没有名字，错误信息只会提供一个通用的参考执行流和对应的数据类型。如果你有多个流，在声明的时候都使用相同的数据类型，那么你很难指出错误信息到底指向那个流。给流变量命名是通过给变量一个名字参数，例如`hls::stream<matrix1("strm_matrix1")>`

接下来嵌套的**for**循环`initmatrices`设定四个二维矩阵A、B、`matrix_sout`和`matrix_hwout`。变量A和B是输入矩阵。它们是由在[0,512)之间的随机数构成。我们选择512没有其他的原因，因为9比特可以保存对应所有的值。需要注意的是，`DTYPE`是作为`int`类型，所以它的范围是明显大于[0,512)，在后面的优化过程中我们通常用范围更小的定点数。矩阵`matrix_swout`和`matrix_hwout`都被初始化为0。他们后续通过调用函数`matmatmul`和`blockmatmul`进行赋值。第二部分的测试平台如图7.7所示。这是`main`函数中最后的一部分了。

图中的第一部分是一系列复杂的嵌套的**for**循环。所有这些**for**循环的目的是为了实现对输入矩阵A、B的处理，使它们能流入到函数`blockmatmul`中。函数`blockmatmul`保存结果在`matrix_hwout`中。

外部的两层**for**循环通过每步执行，实现矩阵以块的形式输入。你可以发现每次迭代都是以`BLOCK_SIZE`作为步进。接下来的两个**for**循环把矩阵A的行写到`strm_matrix1_element`，把矩阵B的列写到`strm_matrix2_element`中。在执行上述步骤时都是一个元素一个元素进行的，通过使用变量`k`访问单个的值从行（列）然后将它们写入到一维数组中。注意，`strm_matrix1_element`和`strm_matrix2_element`都是`blockvec`数据类型，即长度为`BLOCK_SIZE`的一维数组。它们用于保存行或列的`BLOCK_SIZE`个元素。内层循环迭代`BLOCK_SIZE`次。流类型变量`strm_matrix1`和`strm_matrix2`被写入`SIZE`次。换句话说，就是对整行（列）进行缓存，缓存中每个元素保存`BLOCK_SIZE`个值。

类stream通过重载>>操作符，它等价于函数 write (data)。这和对 read() 函数重载，使其等价于操作符<<。因此，表达式 strm_matrix1.write(strm_matrix1_element) 和 strm_matrix1_element >> strm_matrix1 执行是相同的。

```
#include "block_mm.h"
#include <stdlib.h>
using namespace std;

void matmatmul_sw(DTYPE A[SIZE][SIZE], DTYPE B[SIZE][SIZE],
                   DTYPE out[SIZE][SIZE]){
    DTYPE sum = 0;
    for(int i = 0; i < SIZE; i++){
        for(int j = 0; j < SIZE; j++){
            sum = 0;
            for(int k = 0; k < SIZE; k++){
                sum = sum + A[i][k] * B[k][j];
            }
            out[i][j] = sum;
        }
    }
}

int main() {
    int fail = 0;
    hls::stream<blockvec> strm_matrix1("strm_matrix1");
    hls::stream<blockvec> strm_matrix2("strm_matrix2");
    blockvec strm_matrix1_element, strm_matrix2_element;
    blockmat block_out;
    DTYPE A[SIZE][SIZE], B[SIZE][SIZE];
    DTYPE matrix_swout[SIZE][SIZE], matrix_hwout[SIZE][SIZE];

    initmatrices: for(int i = 0; i < SIZE; i++){
        for(int j = 0; j < SIZE; j++){
            A[i][j] = rand() % 512;
            B[i][j] = rand() % 512;
            matrix_swout[i][j] = 0;
            matrix_hwout[i][j] = 0;
        }
    }
}
//the remainder of this testbench is displayed in the next figure
```

图7.6：分块矩阵乘法测试平台的第一部分。函数被分为两个部分，因为太长不能在单页上一次显示。测试平台剩下的部分在图7.7中。其中有一个“软件”版本的矩阵乘法、变量的声明和初始化。

```

// The beginning of the testbench is shown in the previous figure
int main() {
    int row, col, it = 0;
    for(int it1 = 0; it1 < SIZE; it1 = it1 + BLOCK_SIZE) {
        for(int it2 = 0; it2 < SIZE; it2 = it2 + BLOCK_SIZE) {
            row = it1; //row + BLOCK_SIZE * factor_row;
            col = it2; //col + BLOCK_SIZE * factor_col;

            for(int k = 0; k < SIZE; k++) {
                for(int i = 0; i < BLOCK_SIZE; i++) {
                    if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1_element.a[i] = A[row+i][k];
                    strm_matrix2_element.a[i] = B[k][col+i];
                }
                if(it % (SIZE/BLOCK_SIZE) == 0) strm_matrix1.write(strm_matrix1_element);
                strm_matrix2.write(strm_matrix2_element);
            }
            blockmatmul(strm_matrix1, strm_matrix2, block_out, it);

            for(int i = 0; i < BLOCK_SIZE; i++)
                for(int j = 0; j < BLOCK_SIZE; j++)
                    matrix_hwout[row+i][col+j] = block_out.out[i][j];
            it = it + 1;
        }
    }

    matmatmul_sw(A, B, matrix_swout);

    for(int i = 0; i<SIZE; i++)
        for(int j = 0; j<SIZE; j++)
            if(matrix_swout[i][j] != matrix_hwout[i][j]) { fail=1; }

    if(fail==1) cout << "failed" << endl;
    else cout << "passed" << endl;
}

return 0;
}

```

图7.7：分块矩阵乘法的第二部分。第一部分在图7.6中。这部分演示如何将流类型数据送入函数 `blockmatmul`，代码比较函数和用简化三重`for`循环实现之间的结果。

这段代码最后一部分从高亮的`if`语句开始。这和矩阵A的值相关。本质上，这里是为了我们不用经常向`strm_matrix1`中写入相同的值。矩阵A的值可以对应多次`blockmatmul`函数的调用。图7.3是关于这的讨论。这里的`if`语句高亮是为了强调，不能在这里一次次写入相同值。因为函数`blockmatmul`只在有必要的时候读这些数据。所以当我们连续写数据，代码执行不正常，因为流写入的数据是多于读出的数据。

在测试平台中调用`blockmatmul`函数输入数据。在函数调用以后，它收到部分计算结果`block_out`。接下来的两层`for`循环把这些结果填入到数组`matrix_hwout`正确的位置。

在执行一些列复杂的`for`循环之后，分块矩阵乘法实现了。测试平台用于保证代码写入是正确的。它通过比较调用函数`blockmatmul`计算的输出结果和调用`matmatmul_sw`计算的输出结果。在函数调用以后，测试平台迭代遍历二维矩阵`matrix_hwout`和`matrix_swout`，保证每个元素都是相等的。如果其中一个或多个元素不相同，它会置`fail`为1。测试平台结束时会打印输出`failed`或`passed`。

需要注意的是，你不能直接比较函数**blockmatmul**用于实现矩阵乘法的性能，与图7.1中的代码。因为它调用了函数**blockmatmul**函数很多次用于实现整个矩阵乘法。比较需要在相同类型之间进行

为了实现完整的矩阵乘法，一个函数需要用来确定**blockmatmul**被调用多少次。这个函数应该是通用的，它应该不需要制定特殊的值如**BLOCK_SIZE**或矩阵的大小（例如，**SIZE**）。

比较分块矩阵乘法和矩阵乘法之间的资源占用率。当增大矩阵的大小，占用率是如何变化的？块大小是不是在资源占用率中起到了重要作用？通常的趋势是怎样的？

比较分块矩阵乘法和矩阵乘法之间的性能。当增大矩阵的大小，性能如何变化？块大小是否在性能中起到了重要作用？选择两种有相同资源占用率的架构，这两种架构之间性能差异有多大？

7.4 总结

矩阵乘法提供了一种不同的方式来实现矩阵乘法。函数通过流的方式输入矩阵的部分，然后计算矩阵部分的结果。函数通过多次计算，完成对整个矩阵乘法的计算。

第八章 前缀和与直方图

8.1 前缀和

前缀和是许多应用中经常使用的运算算子，例如在递推关系、压缩问题、字符串比较、多项式评估、直方图、基数排序和快速排序应用中[11]。为了创建高效的FPGA设计，下面我们将对前缀和运算进行重新设计。

前缀和本来是一系列数字的累加和。若给定一序列输入中 in_n ，前缀和中第n项的值是输入前n项的累加和 out_n ，即 $out_n = in_0 + in_1 + in_2 + \dots + in_{n-1} + in_n$ 。以下展示的是前四个输出元素的计算过程。

$$\begin{aligned}out_0 &= in_0 \\out_1 &= in_0 + in_1 \\out_2 &= in_0 + in_1 + in_2 \\out_3 &= in_0 + in_1 + in_2 + in_3 \\&\dots\end{aligned}$$

当然，在实际应用中，我们不希望存储和重新计算以前所有输入的累加和，因此我们使用递推方程式表示：

$$out_n = out_{n-1} + in_n$$

递推方程的劣势是在计算 out_n 之前必须先计算出 out_{n-1} ，这从根本上限制了计算时的并行性的扩展和吞吐量的提高。相反，计算前缀和的原始方程由于计算每个输出都可以独立计算，所以很明显可以并行执行，但是代价是进行一系列的冗余计算。使用C语言实现的递推方程如图8.1所示。理想情况下，我们期望代码中循环的II (Initiation interval)=1，但即使对于这样简单的代码，也是具有挑战性的。在Vivado@HLS中进行综合这段代码运行结果如图8.1所示。

这段代码的编写方式是将每个输出数值都写入输出寄存器out[]中，然后在下一次迭代中再次从寄存器中读出上一次输出的数值。由于读取寄存器的延迟是1个时钟周期，因此从寄存器中读取的数据只有在下一个时钟周期才能被处理。结果是，这样的代码设计只能实现II (Initiation interval)=2的循环设计。在这种情况下，有一种简单的改良此代码的方法：我们可以使用一个单独的局部变量来进行累加操作，而不是像以前一样从数组中读回前一次累加的数值。在CPU处理器代码设计中，避免使用额外的外部存储器来替代寄存器作为数据访问的方式更有优势，同样在HLS设计中这样的数据访问方式更重要，因为其他的处理操作很少能成为系统的性能瓶颈。该操作方式代码如图8.2所示。

```
#define SIZE 128
void prefixsum(int in[SIZE], int out[SIZE]) {
    out[0]=in[0];
    for(int i = 1; i < SIZE; i++) {
        #pragma HLS PIPELINE
        out[i] = out[i-1] + in[i];
    }
}
```

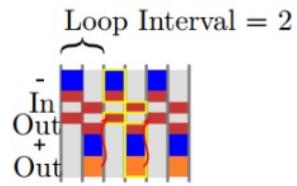


图8.1 前缀和实现代码及行为描述

```
#define SIZE 128
void prefixsum(int in[SIZE], int out[SIZE]) {
    int A = in[0];
    for(int i=0; i < SIZE; i++) {
        #pragma HLS PIPELINE
        A = A + in[i];
        out[i] = A;
    }
}
```

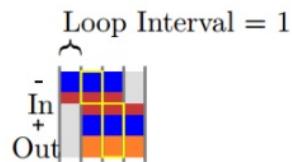


图8.2 优化后的前缀和代码及行为描述

你可能会问，为什么编译器无法自动优化内存负载和存储以改进II的设计。事实证明，Vivado@hls能够优化加载和数组存储，但仅适用于单个基本块范围内的读取和写入。如果我们展开循环，你可能看到如图8.3所示。注意，我们也通过添加约束**array_partition**的方式来达到在接口处可以同时读取和写入多个数值。在这种情况下，Vivado@HLS能够消除循环体内大部分out[]数组的读操作，但我们仍然只能实现循环间隔II=2的设计。在这种情况下，循环中第一次加载读数仍然存在。但是我们可以通过修改代码使用局部变量来替代从out[]数组中读取数据。

理想情况下，当我们展开内部循环时，则在每个时钟中就可以执行更多的操作和减少函数运算的时间间隔。如果我们将展开因子设置2，则性能提升一倍。如果展开因子系数设置为4，则性能提升4倍，即循环在展开时，系统性能以线性方式变化。虽然大部分情况是这样，但是当我们展开内部嵌套循环时，设计中的某些方面是不会变化的。在大多数情况下，例如当循环的迭代间隔是执行很长时间时，展开内部循环对于整个函数性能的提升没有显著影响。但是，对着循环迭代次数的减少，展开循环会产生更大的影响。循环流水线设计中最大成分是流水线自身深度。由Vivado@HLS为流水线循环生成的组合逻辑要求在循环执行后刷新流水线。

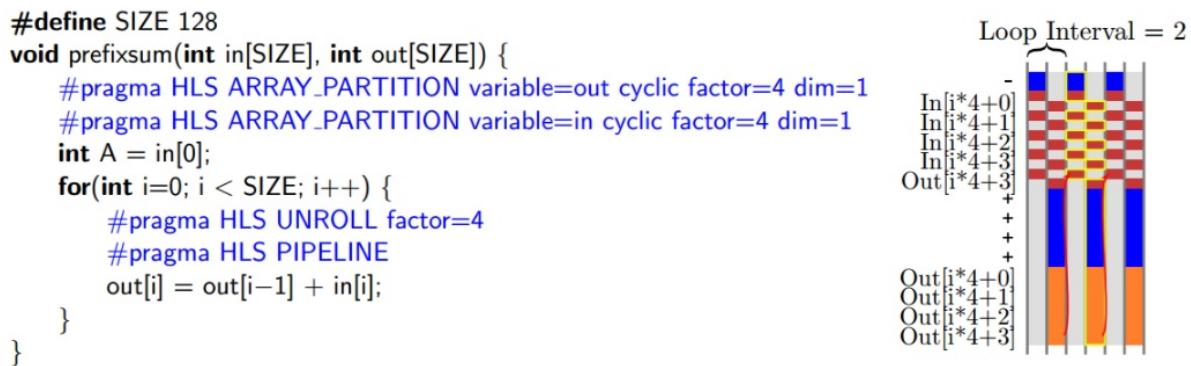
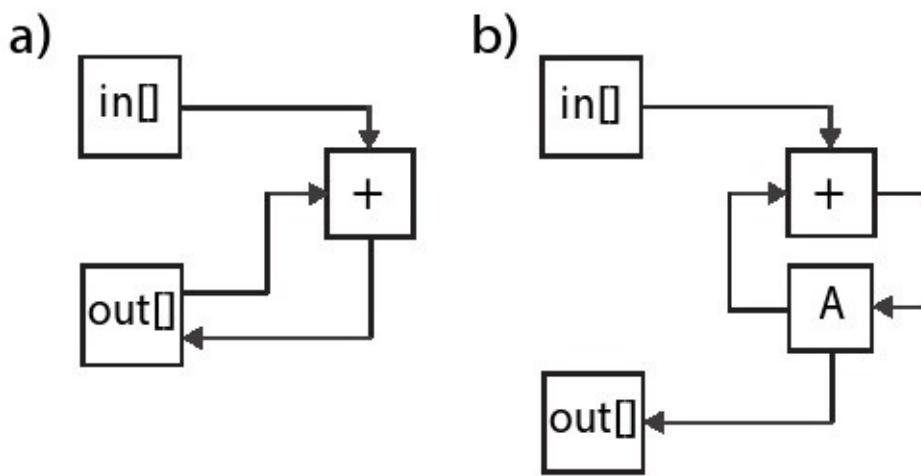


图8.3 使用unroll,pipeline和array_partition指令优化的前缀和代码

图8.4 a)部分展示了与图8.1中代码相对应的体系结构。对 $out[]$ 数组的依赖限制了循环 $II = 1$ 设计的实现。如图8.2中代码所示，使用局部变量计算循环能够减少循环中的延迟并实现 $II = 1$ 的设计

通过设置不同的展开因子来将对应的图8.2中前缀和代码中**for**循环展开，并结合数组分割来达到使循环 $II = 1$ 。前缀和函数延迟如何变化？资源利用率如何变化？你为什么认为是这样变化的？当循环完全展开时会发生什么？

图8.4中展示了图8.1和图8.2中的代码综合产生的硬件架构。在a)部分中，我们可以看到流程中的'loop'包含存储 $out[]$ 数组的输出存储部分，而在b)部分，流程中的循环部分仅包含一个存储累加值的寄存器和只写的输出存储器。简化重复并消除不必要的内存访问是优化HLS代码的常见做法。

本节的目标是表明即使代码中的小改动有时也会对硬件设计产生重大的影响。有些变化可能不一定直观，但可以通过开发工具的反馈来识别。

Data

4	2	4	4	3	3	3	4
---	---	---	---	---	---	---	---

Histogram Values

Bins	2	3	4
Counts	1	4	3

Histogram

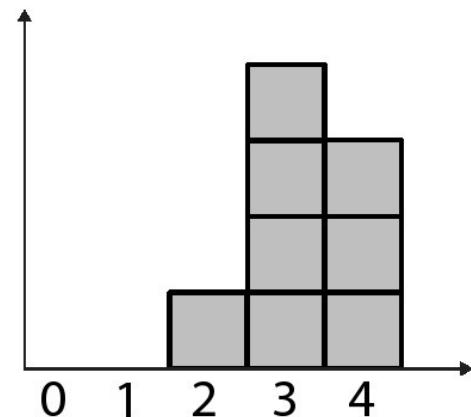


图8.5 一个直方图实例

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
    int val;
    for(int i = 0; i < INPUT SIZE; i++) {
        #pragma HLS PIPELINE
        val = in[i];
        hist[val] = hist[val] + 1;
    }
}
```

图8.6：计算直方图的原始代码。for循环遍历输入数组并递增hist数组的相应元素。

8.2 直方图

直方图表示离散信号的概率分布。当给定一系列离散输入值时，直方图计算每个值在序列中出现的次数。当通过除以输入总数进行标准化处理时，直方图就变成序列的概率分布函数。直方图常应用于图像处理、信号处理、数据库处理以及许多其他领域。在许多情况下，将高精度的输入数据量化处理成更小的数据间隔是直方图中一种常见处理。在本节的内容中，我们将略过数据实际处理过程，而重点关注数据分块操作。

图8.5中是一个直方图应用的简单例子。该数据集由一系列[0,4]中整数表示。下面显示了相应的直方图，在直方图中显示了每个数据块区间的计数，其中每个数据块区间的高度对应每个单独值在数据集中的个数。图8.6中显示了直方图函数的计算代码。

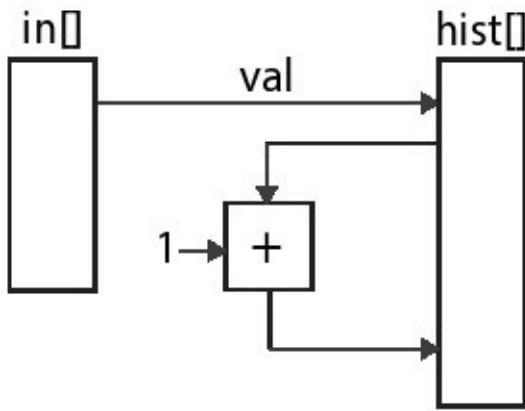


图8.7：图中对应的是图8.6中的代码产生的体系结构。数组中的`val`数据用于索引`hist`数组。`val`会增加并存储回相同的位置。

代码最后看起来与前一小节中的前缀和计算代码非常相似。不同之处在于，前缀和基本上只执行一次累加，而在直方图函数中我们为每个分块均计算一次累加。另一个区别是，在前缀和运算中，我们每次都加上一个新的输入值，而在直方图运算中，我们只加1。当使用**pipeline**指令对函数内部循环进行约束时，会遇到和图8.1中代码同样问题，由于内存的重复读写，系统只能实现 $II = 2$ 的循环。这是因为在每次迭代循环中我们均需要从`hist[]`数组中读取数据和写入数据。图8.7中显示了图8.6中代码的硬件体系结构。从图中可以看到针对`hist[]`数组进行了读取和写入操作。其中，`val`变量用作`hist[]`数组的索引，并且该索引变量读取,递增并写回到同一个位置。

8.3 直方图优化和错误依赖

让我们更深入的观察上面的处理过程。在循环的第一次迭代中，我们读取位置 x_0 处的`hist`数组值并将其写回到相同的位置 x_0 处`hist`数组中。由于读操作需要一个时钟延迟，所以写入数组操作必须在下一个时钟周期发生。然后在下一次迭代循环中，我们读取另一个`hist`数组中另一个位置 x_1 中数组。 x_0 和 x_1 都取决于输入值，并可以取任何值。因此我们考虑到综合成电路时最坏的情况，如果 $x_0 = x_1$,则在前一个写入完成前，位置 x_1 处的读取无法开始。因此，我们必须在读写之间进行切换。

事实证明，只要 x_0 和 x_1 是独立的，我们就必须在读写之间进行切换。如果它们实际上不是独立的呢？例如，我们可能知道数据源实际不会连续产生两个完全相同的数据。那我们现在该怎么做呢？如果我们可以将 x_0 和 x_1 是不同的地址额外信息提供给HLS工具，那么它就能够同时在位置 x_1 处读取，而在位置 x_0 处写入数据了。在Vivado@HLS中，可以通过设置**dependence**指令来完成。

修改后的代码如图8.8所示。上面我们明确表示该函数使用需要一些前提条件。在代码中，我们使用**assert**断言来完成对第二个前提条件的检查。在Vivado@HLS中，这个断言在仿真过程中被启用，以确保仿真测试过程中向量满足所需的前提条件。软件提供的**dependence**指令可以避免前提条件对综合电路的影响。也就是说，它向Vivado@HLS软件工具指示使用特定的方式读取和写入`hist`数组。在这种情况下，迭代循环之间读操作和写操作间距离是2。在这种情况下，距离为n将表明在迭代次数 $i+n$ 中的读操作仅依赖于迭代次数 i 中的写操作。因此，断言`in[i+1] != in[i]`，但可能出现`in[i+2] == in[i]`，所以正确距离是2。

```

#include <assert.h>
#include "histogram.h"
// Precondition: hist[] is initialized with zeros.
// Precondition: for all x, in[x] != in[x+1]
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
    #pragma HLS DEPENDENCE variable=hist inter RAW distance=2
    int val;
    int old = -1;
    for(int i = 0; i < INPUT SIZE; i++) {
        #pragma HLS PIPELINE
        val = in[i];
        assert(old != val);
        hist[val] = hist[val] + 1;
        old = val;
    }
}

```

图8.8：计算直方图的替代函数。通过向Vivado @HLS添加指令约束输入，就可以在不显着更改代码的情况下实现 $II = 1$ 的设计。

在图8.8中，我们向代码添加了一个前提条件，使用一个断言对其进行了检查，并且使用**dependence**指令指示工具对于前提条件的约束。如果你的测试文件不满足这个先决条件会发生什么？如果删除**assert()**会发生什么？Vivado@HLS是否仍检查前提条件？如果前提条件与**dependence**指令不一致，会发生什么？

不幸的是，如果我们不愿意接受附加的先决条件，那么**dependence**指令就不能真正帮助我们。同样清楚的是，因为我们可能需要使用存储在hist数组中的所有值，所以我们不能直接应用于前缀函数相同的优化。另一种方法是使用不同的技术实现hist数组，例如我们可以完全划分hist数组，从而使用触发器（FF）资源实现数组。由于在一个时钟周期内写入FF的数据在下一个时钟周期可立即使用，这解决了重复问题并且在涉及少量分块时这也是一个不错的解决方案。图8.9显示了这种设计的结构。然而，当需要大量分块时，这往往是一个糟糕的解决方案。通常的直方图是用成百上千个分块进行构建并且对于大数据集可能需要许多位精度来计算所有输入。这导致大量FF资源和大型多路复用器，其中多路复用器也需要逻辑资源。通常在块RAM（BRAM）中存储较大的直方图是一个更好的解决方案。

回到图8.6中的代码，我们看到，架构必须能够处理两种不同的情况。一种情况是输入包含连续的同一个分块中的值。在这种情况下，我们希望使用一个简单的寄存器以最小的延迟执行累加。第二种情况是当输入中不包含连续的同一个分块中的值，在这种情况下，我们需要读取，修改并将结果写回到内存，并且可以保证hist数组的读操作不会受到前一个写操作的影响。我们已经看到，这两种情况都可以单独实施，也许我们可以将他们组合成单一设计。图8.10中所示了完成此操作的代码。该代码使用一个本地变量old来存储上一次迭代分块的结果，同时使用另一个本地变量accu存储此次迭代分块的计数。每次迭代循环时，我们都会检查当前分块是否与上次迭代分块值相同。如果是这样，那么我们可以简单地增加accu。如果不是，那么我们需要将数值存储在hist数组中，然后在hist数组中使用当前值更替。无论哪种情况，我们都会使用当前的值更新old和accu值。该代码对应的体系结构如图8.11所示。

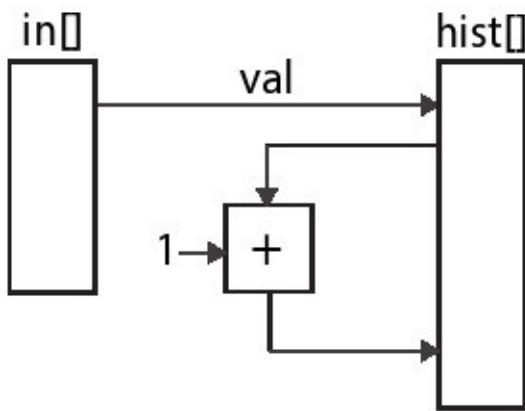


图8.9：当`hist`数组完全分区时，由图8.6中的代码产生的体系结构。

在这段代码中，我们仍然需要一个**dependence**指令，就像图8.8中那样，但是形式略有不同。在这种情况下，在同一个迭代循环中，读取和写入访问是在不同地址中进行的。这两个地址都依赖于输入数据，因此可以指向`hist`数组中的任何单独像素。因此，Vivado@HLS假定这两种访问都可以访问相同的位置，并以交替的周期顺序对数组进行读取和写入操作，则循环的II = 2。但是，通过查看代码，我们可以很容易地看到`hist[old]`和`hist[val]`永远不会访问相同的位置，因为他们位于条件`if(old == val)`的`else`分支中。在一次迭代（内部依赖）内，写入操作（RAW）之后的读取操作永远不会发生，因此这是错误的依赖关系。在这种情况下，我们不使用**dependence**指令来通知工具关于函数的先决条件，而是关于代码本身的属性。

综合图8.6和图8.10中的代码。这两种情况下启动间隔（II）是多少？当你从图8.10中的代码中删除**dependence**指令时会发生什么？在这两种情况下，循环间隔怎么变化？资源使用情况如何？

对于图8.10中的代码，你可能会问为什么像Vivado@HLS这样的工具无法确定性能。事实上，虽然在这样的一些简单情况下，更好的代码分析可以将if条件属性传播到每个分支，但我们必须接受存在一些代码段，其中内存访问的性能是不可判定的。在这种情况下最高的性能只能通过添加用户信息的静态进程来实现。最近的一些研究工作通过引入一些研究来寻求改进设计中的动态控制逻辑[60, 40, 19]。

```

#include "histogram.h"
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
    int acc = 0;
    int i, val;
    int old = in[0];
#pragma HLS DEPENDENCE variable=hist intra RAW false
    for(i = 0; i < INPUT SIZE; i++) {
        #pragma HLS PIPELINE II=1
        val = in[i];
        if(old == val) {
            acc = acc + 1;
        } else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}

```

图8.10：从`for`循环中删除写入依赖后的读取。这需要一个`if/else`结构，看起来似乎会增加设计的不必要的复杂性。尽管数据路径更复杂，但它允许更有效的流水线操作。

图8.11展示的是图8.10中代码重构后的图形化描述。图中并非所有的操作都在这里显示，但实现功能的主要思想都显示出来。你可以看到两个单独的`if`和`else`区域（用虚线表示）。图中将`Acc`变量复制了两次目的是让图形更容易理解；在实际设计中只有一个寄存器用于该变量。该图显示了对应于计算的`if`子句和对应于数据读写的`else`子句的两个分离的数据路径。

8.4 提高直方图性能

通过一些努力，我们已经实现了 $II = 1$ 的设计。以前我们已经看到通过部分展开内部循环方式可以进一步减少设计的执行时间。但是对于直方图函数是有点困难的，有几个原因如下。第一个原因是顺序循环执行，即下次循环执行必须在这次循环计算完成的前提下才能开始，除非我们能够以某种方式分解输入数据。第二个原因是在回路 $II = 1$ 的情况下，电路需要在在个时钟周期内同时执行读取和写入`hist`数组，这样就需要占用FPGA中BRAM资源的两个端口。之前我们已经考虑过数组分区方法来增加存储数组内存端口的数量，但是由于访问顺序输入数据，所以没有特别好的方式来分割`hist`数组。

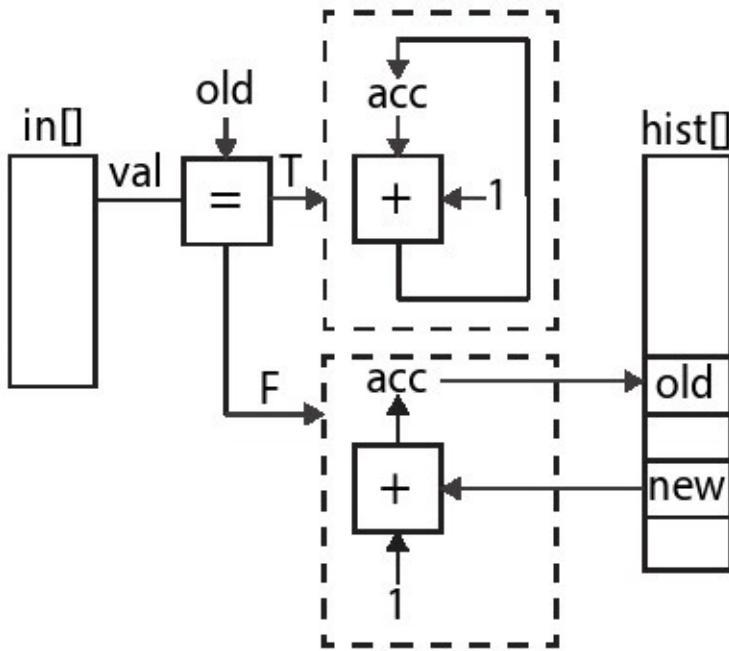


图8.11：图中所示是与图8.10中的代码对应的数据路径的描述。这里包含对应于`if`和`else`子句两个独立的部分。该图显示了计算的重要部分，并省略了一些细节。

不过，并不是没有指望了。因为我们可以将直方图计算分解为两个阶段来达到更多的并行性。在第一阶段，我们将输入数据分成若干独立的分块。每个分块的直方图可以使用我们之前的直方图解决方案独立计算。第二阶段，将各个直方图组合起来生成完整数据集的直方图。这种分块（或映射）和合并（或还原）过程与MapReduce框架[20]采用的过程非常相似，并且是并行计算的常见模式。Map-reduce模式适用于包含交换和关联操作的循环，例如这种情况下的加法。完整方案如图8.12所示。

图8.13中式实现这种架构的代码。直方图函数实现了Map-reduce模式的'map'部分，并且会多次实例化。该代码与图8.10中的代码非常相似。主要的区别在于我们添加了额外的代码来初始化hist数组。`Histogram_map` 函数输入数组是hist数组中一个分区数据。`Histogram_reduce` 函数实现了模式中的“还原”部分。它将分区数据的直方图作为输入，并通过将每个直方图的计数相加，将它们组合成完整的直方图。在我们的图8.13的代码示例中，我们只有两个处理对象，因此将两个输入数组hist1和hist2合并。这可以很容易的扩展以处理更多的元素。

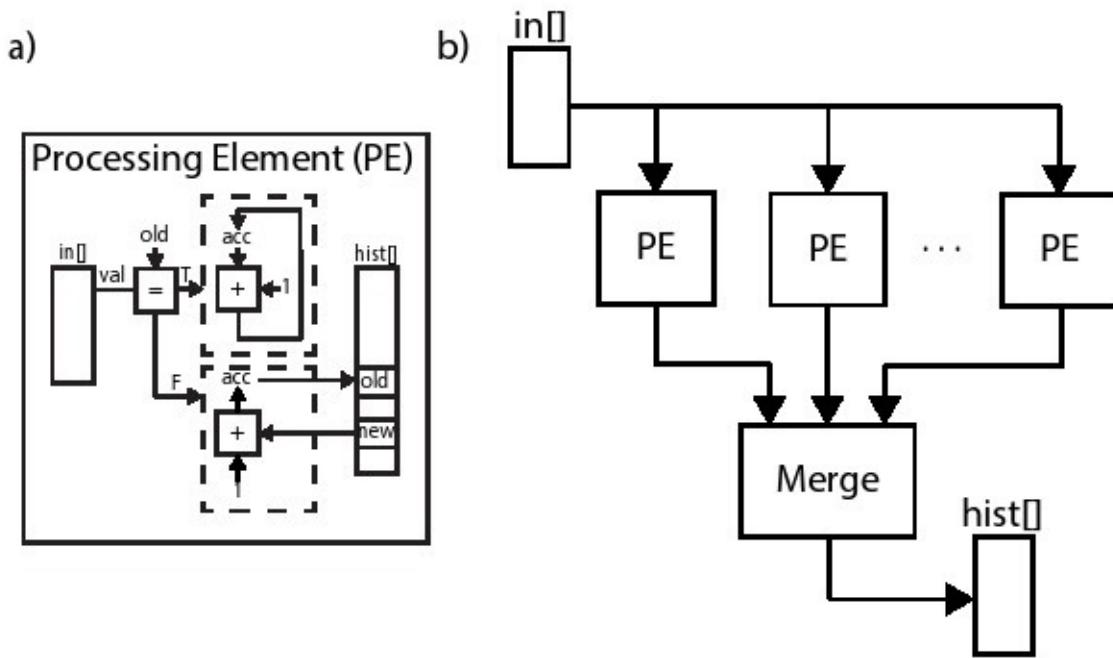


图8.12：使用map-reduce模式实现的直方图计算。a) 部分中的处理单元（PE）与图8.11所示的结构相同。`in`数组是分块后的，每个分块由一个单独的PE处理。合并块组合各个直方图以创建最终直方图。

新的直方图函数将输入数据分成两个分区，分别存储在`inputA`和`inputB`数组中。它使用`histogram_map`函数计算每个分区的直方图，然后将其存储在`hist1`和`hist2`数组中。这两个数组被输入到`histogram_reduce`函数中。该函数将它们合并后并将结果存储在`hist`数组中，其中合并的`hist`数组是顶层直方图函数的最终输出。

修改图8.13中的代码包含支持可参数化改变PE个数的变量`NUM_PE`？提示：你需要根据数组分块数量`NUM_PE`以及循环将数组合并成一个数组。当你改变PE的数量时，吞吐量和任务间隔会发生什么变化？

我们在`histogram`函数中使用`dataflow`指令来达到任务级流水线设计。在这种情况下有三个处理过程：两个`partial_histogram`函数处理实例和两个`histogram_reduce`函数处理实例。在一个任务中，因为两个`partial_histogram`处理的数据相互独立，所以可以同时执行。`Histogram_reduce`函数处理过程必须在`partial_histogram`处理完成后才开始。因此，`dataflow`指令本质上是创建了一个两阶段的任务管道。第一阶段执行`partial_histogram`函数，而第二阶段执行`histogram_reduce`函数。与任何数据流设计一样，整个`histogram`函数的间隔取决于两个阶段的最大启动间隔。第一个阶段的两个`partial_histogram`函数时相同的，并且具有相同的间隔 ($II_{histogram_map}$)。`Histogram_reduce`函数将由另一个间隔 ($II_{histogram_reduce}$)。顶层`histogram`函数的启动间隔 $II_{histogram}$ 是 $\max(II_{histogram_map}, II_{histogram_reduce})$ 。

```

#include "histogram_parallel.h"
void histogram_map(int in[INPUT SIZE/2], int hist[VALUE SIZE]) {
    #pragma HLS DEPENDENCE variable=hist intra RAW false
    for(int i = 0; i < VALUE SIZE; i++) {
        #pragma HLS PIPELINE II=1
        hist[i] = 0;
    }
    int old = in[0];
    int acc = 0;
    for(int i = 0; i < INPUT SIZE/2; i++) {
        #pragma HLS PIPELINE II=1
        int val = in[i];
        if(old == val) {
            acc = acc + 1;
        } else {
            hist[old] = acc;
            acc = hist[val] + 1;
        }
        old = val;
    }
    hist[old] = acc;
}
void histogram_reduce(int hist1[VALUE SIZE], int hist2[VALUE SIZE], int output[VALUE SIZE]) {
    for(int i = 0; i < VALUE SIZE; i++) {
        #pragma HLS PIPELINE II=1
        output[i] = hist1[i] + hist2[i];
    }
}
//Top level function
void histogram(int inputA[INPUT SIZE/2], int inputB[INPUT SIZE/2], int hist[VALUE SIZE]){
    #pragma HLS DATAFLOW
    int hist1[VALUE SIZE];
    int hist2[VALUE SIZE];

    histogram_map(inputA, hist1);
    histogram_map(inputB, hist2);
    histogram_reduce(hist1, hist2, hist);
}

```

图8.13：图示是使用任务级并行和流水线操作的直方图的另一种实现。直方图操作分为两个子任务，这两个子任务在两个`histogram_map`函数中执行。使用`histogram_reduce`函数将这些结果合并到最终的直方图结果中。顶层的`histogram`函数是将这三个函数连接在一起。

当你添加或更改 `pipeline` 指令时，会发生什么？比如，在`histogram_reduce`函数中为 `for` 循环添加 `pipeline` 指令是否有益？将`pipeline`指令移动到直方图映射函数中，也即将它拉到当前所在 `for` 循环的外部，那么结果是什么？

本节目标是学习直方图计算的优化算法，而这也是许多应用程序中另外一个小但重要的核心。关键是因为对于我们的程序，工具可以理解的东西通常是有限制的。在某些情况下，我们必须注意如何写代码，而在其他情况下，实际上我们必须提供给工具更多关于代码或代码执行环境的信息。特别的，内存访问形

式的性能通常会严重影响HLS生成正确且高效的硬件。在Vivado@HLS中，可以使用 **dependence** 指令表示这些性能。有些时候这些优化或许与直觉相反，比如在8.10中添加的 **if/else** 结构。一些情况下优化或许要求一些创造性，就像分布式计算在图8.12和8.13的应用。

8.5 结论

本节我们学习了前缀和与直方图内核。尽管这些方法看起来不同，但他们都包含重复的内存访问。如果内存访问不是流水线的，这些重复访问就会限制吞吐量。这两种情况，我们可以通过重构代码来去除重复。前缀和会非常容易，因为它的访问模式就是确定的。直方图情况下，我们必须重新写代码以解决重复访问的问题，或者确保实践中不会发生重复。无论是这两种的哪种情况，我们都需要一种方法来向Vivado®HLS描述一些信息，关于环境或者工具自身无法确定的代码部分。这些信息会在 **dependece** 指令中被捕获。最后，我们研究了两种算法的并行化方法，以便他们可以在每个时钟周期处理大量数据样本。

第九章 视频系统

9.1 背景

视频处理是目前FPGA上常见的应用。其中一个选择FPGA处理视频应用的原因是目前的FPGA时钟处理频率可以很好地满足常见视频数据对处理速率的要求。例如，一般的高清视频格式，全高清视频格式(1080p@60HZ)需要的处理速率是 $1920 \text{ (像素/行)} \times 1080 \text{ (行/帧)} \times 60 \text{ (帧/秒)} = 124,416,000 \text{ (像素/秒)}$ 。

当对数字视频流(1080p@60HZ)进行编码时(1080p视频帧实际传输时像素大小为2200x1125，其中行2200像素 = 行同步+行消隐前肩+有效像素(1920像素)+行消隐后肩，场1125行=场同步+场消隐前肩+有效行(1080行)+场消隐后肩)，其中消隐像素也会伴随有效像素以148.5MHz(1080P@60HZ)的频率在FPGA流水线电路中进行处理。为了达到更高的处理速率可以通过在每个时钟周期内处理多个采样数据的方式实现。数字视频是如何传输的详细内容介绍将在9.1.2小节中展开。另一个选择FPGA处理视频应用需求的原因主要是因为视频按扫描线顺序从左上角像素到右下角像素的顺序进行处理，如图9.1所示。这种可预测的视频数据处理顺序允许FPGA在无需消耗大量存储器件的条件下构建高效的数据存储架构，从而有效地处理视频数据。这些体系结构的详细信息将在第9.2.1小节中介绍。

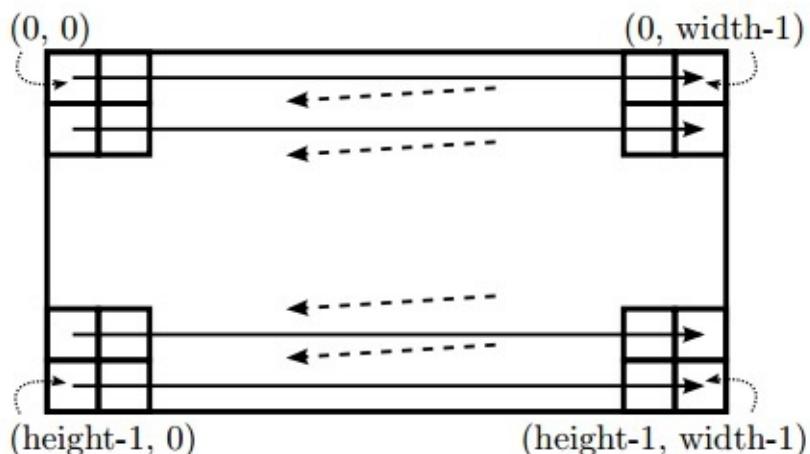


图 9.1: 视频帧的扫描处理顺序

视频处理应用是HLS一个非常好的目标应用领域。首先，视频处理通常是可以容忍处理延迟的。尽管一些应用可能会将整体延迟限制到一帧之内，但是许多视频处理应用中是可以容忍若干帧的处理延迟。因此，在HLS中使用接口和时钟约束方式实现的高效的流水线处理，可以不用考虑处理时延问题。其次，视频算法往往非常不统一，算法一般是基于算法专家的个人喜好或擅长的方式被开发的。一般情况下它们是被使用可以快速实现和仿真的高级语言开发的。虽然全高清视频处理算法在FPGA处理系统上可以以每秒60帧处理速度运行或者是在笔记本电脑上通过使用综合的C/C++代码以每秒一帧的处理速度运行是很常见的，但是RTL级仿真时仅能一个小时一帧的速度运行。最后，视频处理算法通常以适合HLS容易实现的嵌套循环编程风格编写。这意味着来自算法开发人员编写的许多视频算法的C / C ++原型代码均可以被综合成相应的FPGA电路。

9.1.1 视频像素表示

许多视频输入和输出系统都是围绕人类视觉系统感知光线的方式进行优化的。第一个原因就是眼球中的视锥细胞对红、绿和蓝光敏感。其他颜色都可以视为红、绿和蓝色的融合。因此，摄像机和显示器模仿人类视觉系统的功能，对红、绿、蓝光采集或者显示红、绿和蓝像素，并用红色、绿色和蓝色分量的组合来表示颜色空间。最常见的情况是每个像素使用24位数据位表示，其中红绿蓝每个分量各占8位数据位，其中颜色分量的数据位的位数也存在其他情况，例如高端系统中的每个像素颜色分量可以占10位位深或甚至12位位深。

第二原因是人类视觉系统对亮度比对颜色更敏感。因此，在一个视频处理系统内，通常会将RGB颜色空间转换到YUV颜色空间，它将像素描述为亮度(Y)和色度(U和V)的分量组合，其中色度分量U和V是独立于亮度分量Y的。例如常见视频格式YUV422，就是对于4个像素点，采样4个Y的值，两个U的值，两个V的值。这种格式应用于视频压缩时的采样叫色度子采样。另一种常见的视频格式YUV420表示采样4个像素值由4个Y值，一个U值和一个V值组成，进一步减少了表示图像所需像素数据。视频压缩通常在YUV色彩空间中进行。

第三个方面是由于眼睛中的眼杆和感光部分对绿光比对红光更加敏感，并且大脑主要从绿光中获取亮度信息。因此，图像传感器和显示器采用马赛克模式，比如采样2个绿色像素和1个红像素与1个蓝像素比例的Bayer模式。于是在相同数量焦平面采集单元或显示器显示单元条件下系统可以采集或显示更高分辨率的图像，从而降低图像采集传感器或显示器的制造成本。

视频系统多年来一直围绕人类视觉系统进行设计。最早的黑白相机对蓝绿光敏感，以匹配眼睛对该颜色范围内亮度的敏感度。然而，不幸的是它们对红光不太敏感，因此红色(如红妆)在相机上看起来不正常。当时针对这个问题的解决方案绝对是低技术含量的：拍照时演员穿着华丽的绿色和蓝色化的妆。

9.1.2 数字视频格式

除了表示个别的像素之外，数字视频格式必须以视频帧的形式对像素进行组织和编码。在很多情况下，通过同步或同步信号的方式来表示连续像素序列的视频帧开始和结束。在某些视频接口标准(如数字视频接口或DVI)中，同步信号使用单独的物理线路引出。在其他标准(如数字电视标准BTIR 601/656)中，同步信号的开始和结束由不会出现在视频信号中的特殊像素值表示。

相邻行像素之间(从左到右顺序扫描)由水平同步信号分隔开，且相邻行像素之间的水平同步信号由若干时钟周期的脉冲信号组成。另外，在水平同步信号和有效像素信号中间还有若干时钟周期的无效信号，这些信号称为行前肩和行后肩。同样，相邻帧信号之间(从上到下顺序扫描)由垂直同步脉冲分隔。每个视频帧之间有若干行有效的帧同步信号。注意，相邻帧之间的垂直同步信号仅在视频帧的开始出现。同样，帧视频信号中除了同步信号和视频有效信号外，还有无效视频行组成的场前肩和场后肩信号。另外，大多数数字视频格式包含表示有效视频像素起始的数据使能信号。因此，所有无效的视频像素一起组成水平消隐间隔和垂直消隐间隔。这些信号如图9.2所示。

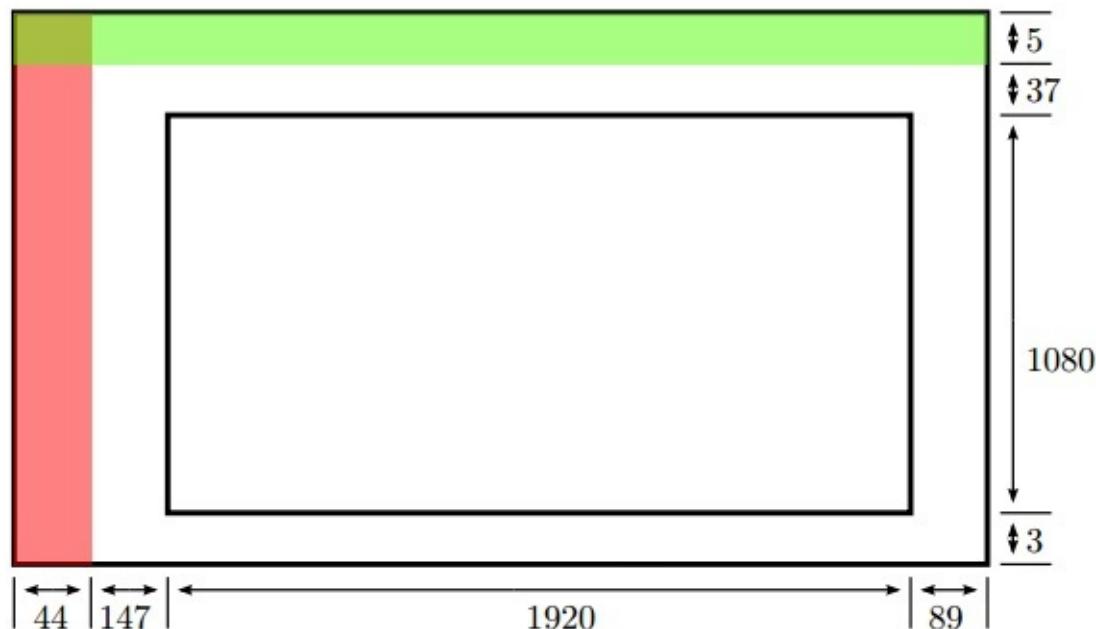


图9.2：1080P@60Hz高清视频信号中的典型的同步信号

数字视频信号均是对原来电视标准的模拟信号（美国的NTSC和许多欧洲国家的PAL）通过抽样、量化和编码后形成。由于用于模拟阴极射线管扫描的硬件具有有限的电平转换速率，所以水平和垂直同步间隔的时间要满足从硬件恢复扫描到下一行的开始。同步信号由若干个时钟周期的低电平组成。另外，由于电视无法有效地显示靠近同步信号的像素，因此引入前肩信号和后肩信号，这样就可以显示更多的图片像素。即使如此，由于许多电视机都是基于电子束扫描原理设计的，所以其中的图像数据有20%的边框处的像素是不可见。

如图9.2所示，典型1080P (60Hz) 视频帧共包含2200 1125个数据采样像素点。在每秒60帧的情况下，这相当于每秒总共148.5百万的采样像素点。这比1080P (60Hz) 帧中的有效视频像素的平均速率 ($1920 \times 1080 \times 60 =$ 每秒124.4百万像素) 高很多。现在的大多数FPGA都可以满足以这个时钟速率进行高效的处理，并且是在一个时钟周期进行一次采样的方式下运行的。对于更高分辨率的处理系统需求，如4K或2K的数字影院，或者每秒120帧甚至240帧的处理需求，就需要一个时钟周期采样更多像素点，通过增加数据处理的吞吐量方式来运行。注意，我们通常可以通过在HLS中展开循环的方式来生成此类结构（请参见1.4.2小节）。同样，当处理低分辨率和低帧率需求时，优选的方案是采用操作共享的方式，多个时钟周期去处理一次采样。这样的处理结构是通过指定循环的处理间隔方式实现。

```

#include "video_common.h"
unsigned char rescale(unsigned char val, unsigned char offset, unsigned char scale) {
    return ((val - offset) * scale) >> 4;
}

rgb_pixel rescale_pixel(rgb_pixel p, unsigned char offset, unsigned char scale) {
    #pragma HLS pipeline
    p.R = rescale(p.R, offset, scale);
    p.G = rescale(p.G, offset, scale);
    p.B = rescale(p.B, offset, scale);
    return p;
}

void video_filter_rescale(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
    rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH],
    unsigned char min, unsigned char max) {
    #pragma HLS interface ap_hs port = pixel_out
    #pragma HLS interface ap_hs port = pixel_in
    row_loop:
    for (int row = 0; row < MAX_WIDTH; row++) {
        col_loop:
        for (int col = 0; col < MAX_HEIGHT; col++) {
            #pragma HLS pipeline
            rgb_pixel p = pixel_in[row][col];
            p = rescale_pixel(p,min,max);
            pixel_out[row][col] = p;
        }
    }
}

```

图9.3 简单视频滤波器的实现代码

例如，在图9.3中代码展示了一个简单的视频处理应用程序，程序中执行每次循环的II = 1(Initiation interval)，也就是一个时钟周期处理一次采样数据。代码中采用嵌套循环的方式是按照图9.1中所示的扫描线顺序处理图像中的像素。II = 3的设计方式可以通过共享计算组件的方式来减少资源使用量。通过设置内层循环展开因子为2和输入输出数组拆分因子为2就可以满足一个时钟处理两个像素。这种例子相对简单，因为每个组件和每个像素的处理是相对独立的。更复杂的功能可能不会因为资源共享而处理更方便，也可能无法同时处理多个像素。

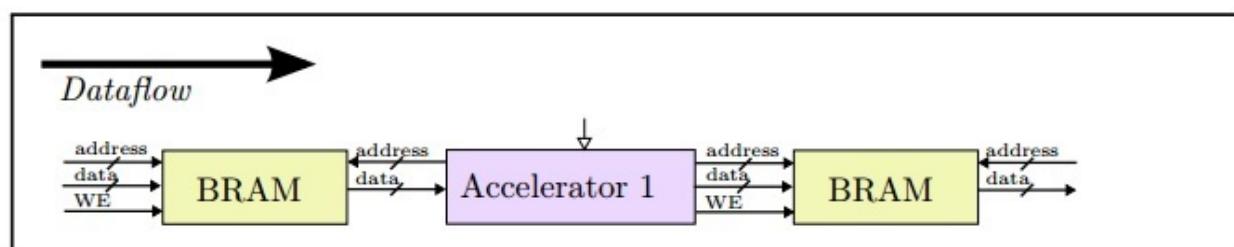


图9.4 将视频处理设计与BRAM接口集成

```

void video_filter(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
    rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface ap_memory port = pixel_out // The default
    #pragma HLS interface ap_memory port = pixel_in // The default
}

```

高速计算机视觉应用是需要满足每秒处理10000帧的200*180像素的视频帧的速度要求。这样的应用通常是使用一个高速图像传感器直接和FPGA直接连接，并且中间不需要有同步信号。在这种情况下，你会每个时钟周期内进行多少次采样处理呢？FPGA可以很好的完成处理么？答案是你可以使用HLS编写嵌套循环的结构来完成这样的设计。

9.1.3 视频处理系统架构

到目前为止，我们专注于构建视频处理应用部分的程序而不关心如何将其整合到一个大的系统程序中去。在很多情况下，如图9.1.2中示例代码，大部分像素处理发生在循环内，并且当循环运行时每个时钟周期仅处理一个像素。在本节中，我们将讨论将视频处理部分程序集成到大的系统程序中的情况。

默认情况下，Vivado@HLS软件会将代码中的函数数组接口综合成简单的存储器接口。其中，存储器写数据接口由地址总线、数据总线和写使能信号线组成。存储器每次读取和写入数据均有相应地址变化，并且数据读取和写入时间固定。如图9.4所示，将这种接口与片上存储资源Block RAM资源集成在一起使用很简单方便。但是即使在大容量的芯片，片上Block RAM资源也是很稀缺的，因此如果存储大容量的视频资源会很快消耗完片上Block RAM资源。

针对每像素24位的1920x1080视频帧，芯片需要消耗多少BlockRAM资源才能完成存储一帧视频帧？片上Block RAM资源最多又能存储多少帧呢？

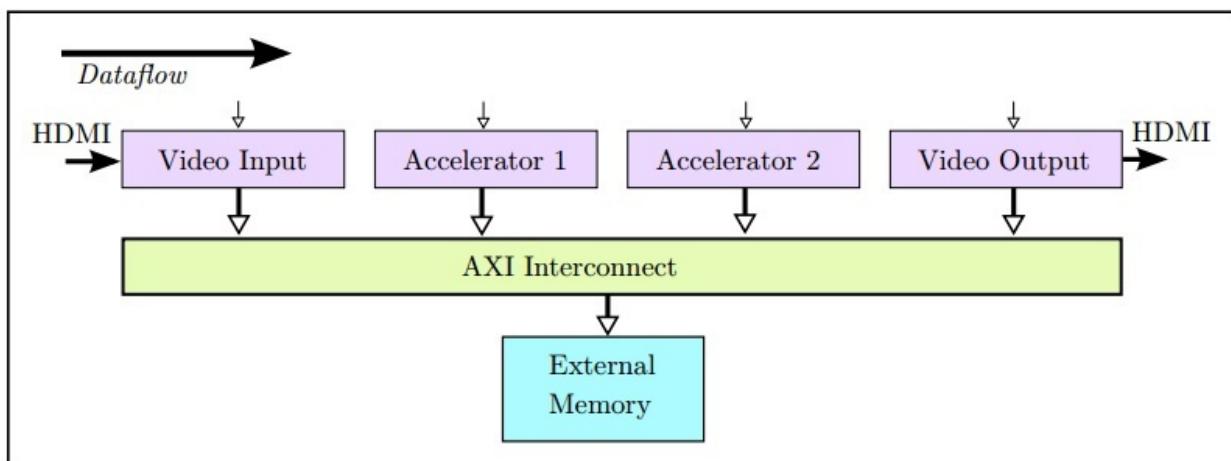


图9.5 将视频处理设计与外部存储接口集成

```

void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface s_axi port = pixel_out
    #pragma HLS interface s_axi port = pixel_in
}

```

通常，大部分视频系统更好的选择是将视频帧存储在外部存储器中，比方说DDR。在图9.5中展示了系统与外部存储器典型系统集成模式。在FPGA内部，外部存储器的控制器（MIG）集成了ARM AXI4标准化从接口与FPGA内的其他模块进行通信。FPGA内的其它模块使用AXI4主接口或者是通过专门的AXI4内部互联模块与外部存储器的AXI4从接口相连。AXI内部互联模块允许多个主设备模块访问多个从设备模块。该架构抽象了外部存储器的操作细节，允许在无需修改FPGA设计中其它模块的情况下，允许不同外部存储器模块和标准互换使用。

尽管大多数处理器都使用高速缓存，从而高性能处理数据。但通常情况下，如图9.5所示，基于FPGA实现的视频处理系统，无需片上高速缓存。在处理器系统中，高速缓存可以提供对先前访问的数据更低时延的访问，并通过始终读取或写入完整的高速缓存行来提高对外部存储器访问的带宽。一些处理器甚至还使用更复杂的数据处理机制，如预取和推测读取，以减少对外部存储器的访问时延和增加对外部存储器的访问带宽。对于大多数基于FPGA的视频处理系统，因为大多数视频算法访问外部存储器是可预测的，所以使用线性缓冲区和窗口缓冲区可以一次从外部缓冲区中读取多个数据。另外，当以猝发模式对外部存储器访问时，Vivado@HLS能够提前调度地址变换，避免读取数据时拖延计算。

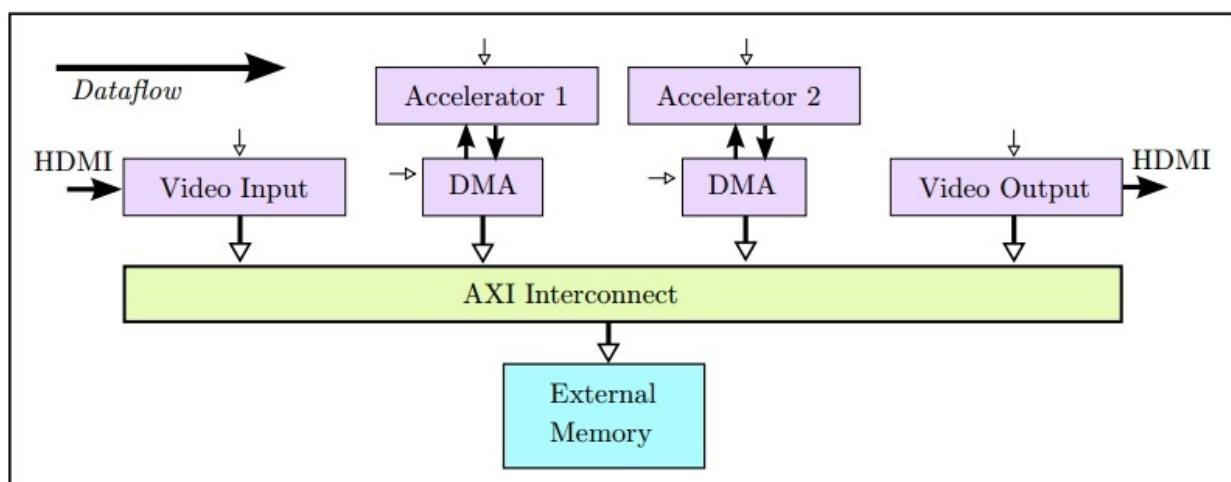


图9.6:通过DMA将视频处理设计模块与外部存储器接口集成

```

void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface s_axi port = pixel_out
    #pragma HLS interface s_axi port = pixel_in
}

```

```

void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH], pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface ap_hs port = pixel_out
    #pragma HLS interface ap_hs port = pixel_in

    video_filter(hls::stream<pixel_t> &pixel_in, hls::stream<pixel_t> &pixel_out)
}

```

图9.7: HLS中流接口的代码实现方式

如图9.6所示为另一种存储器体系架构。在这种架构中，算法处理模块通过外部直接存储器（DMA）与外部存储器连接，DMA完成与外部存储控制器（MIG）之间地址转换的细节操作。算法处理模块可以以AXI总线数据流（AXIS）的形式将数据传输给DMA控制器。DMA假设数据是算法处理模块生成的，并将数据写入外部存储器。在Vivado@HLS中，有多种编码方式可以将代码中的数组接口综合成AXIS流接口，如图9.7所示是其中一种。在这种情况下，C代码与前面看到的代码相同，但接口约束指令不同。另外一种代码实现方法是使用 `hls::stream<>` 方式显式建立总线流接口。无论那种情况都必须注意，DMA中生成数据的顺序要与C代码中访问数据的顺序相同。

AXIS数据流接口的一个优点是它允许在设计系统中，采用多个算法处理模块级联的方式，并且不需要在外部存储器中存储算法处理计算中间结果。在一些应用例子中，如图9.8所示，FPGA系统在没有外部存储器的情况下，将从输入接口（如HDMI）上接收到的像素数据处理后直接将它们发送到输出接口。这样的设计通常要求算法处理模块的吞吐量必须达到需求，以满足外部接口严格实时约束的要求。当一些复杂的算法构建时，系统难以保证其数据吞吐量需求，因此系统如果至少能提供一帧缓冲区，就可以为算法构建提供更大的灵活性。当输入和输出像素速率不同或者可能不相关时（例如接收任意输入视频格式并输出不同任意视频格式的系统），帧缓冲可以起到简化构建系统的作用。

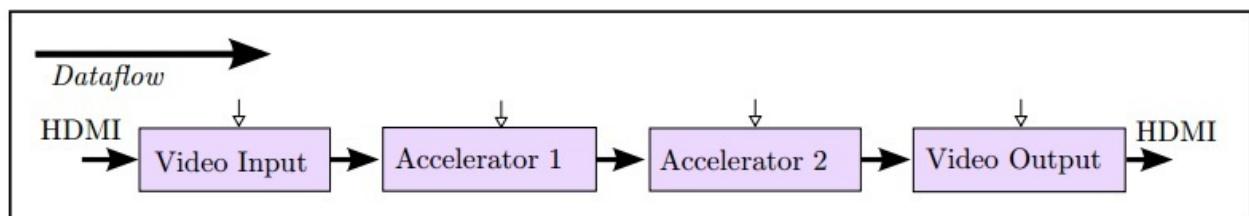


图9.8：将视频处理设计与流接口集成

```

void video_filter(pixel_t pixel_in[MAX_HEIGHT][MAX_WIDTH],
    pixel_t pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface ap_hs port = pixel_out
    #pragma HLS interface ap_hs port = pixel_in
}
    
```

9.2 实现

当一个系统实际处理视频数据时，一般从视频处理算法实现角度来进行系统分解集成。在本章的剩余部分，我们将假设输入数据是按照流的方式送入系统，并且按照扫描线的顺序进行处理的。采用HLS开发，我们不关注代码RTL级具体实现方式，只关注HLS设计能满足所需要的性能指标要求即可。

9.2.1 行缓冲和帧缓冲

在视频处理算法中，通常计算一个输出像素的值时需要输入像素及其周围的像素的值做参考，我们把储存计算所需要的输入像素值的区域叫窗口。从概念上讲，按照窗口大小去Z型扫描图片，然后根据扫描到的像素值计算出输出结果就可以。如图9.9所示示例代码中，示例代码是针对视频帧进行二维滤波。示例代码中在计算每一个输出像素结果前需要从输入视频帧中读取相应的窗口像素数据（存储在数组中）。

在图9.9中，代码中包含的 `int wi =row + i - 1;int wj = col + j -1;`，解释这些表达式为什么包含”-1”这一项。提示：如果滤波器核换成 7×7 ，而不是 3×3 ，”-1”这个数字项会改变么？

注意，在此代码中，每计算一个输出像素值，必须多次读入像素并填充到窗口缓冲区中。如果每个时钟周期只能执行一次读操作，示例代码的执行性能会受限于读入像素速率大小。因此示例代码基本上是图1.8中所示一维滤波器的二维版本。另外，因为输入像素不是以正常扫描线顺序读取的，所以接口约束也只有有限的几个可选。（本主题将在9.1.3小节中更详细讨论）。

仔细观察相邻的窗口缓冲区中缓冲的数据，你会发现缓冲的数据高度重叠，这意味着相邻窗口缓冲区之间数据更高的依存性。这也意味来自输入图像的像素可以被本地缓存或者高速缓存存储，以备数据被多次访问使用。通过重构代码来每次只读取输入像素一次并存储在本地缓冲区中，这样可以使系统性能得到更好的表现。在视频系统中，由于本地缓冲区存储窗口周围多行视频像素，所以本地缓冲区也称为线性缓冲区。线性缓冲区通常使用Block RAM(BRAM)资源实现，而窗口缓冲区则使用触发器(FF)资源实现。图9.10所示是使用线性缓冲区重构的代码。注意，对于 $N \times N$ 图像滤波器，只需要 $N-1$ 行存储在线性缓冲区中即可。

```

rgb_pixel filter(rgb_pixel window[3][3]) {
    const char h[3][3] = {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};
    int r = 0, b = 0, g = 0;
    i_loop: for (int i = 0; i < 3; i++) {
        j_loop: for (int j = 0; j < 3; j++) {
            r += window[i][j].R * h[i][j];
            g += window[i][j].G * h[i][j];
            b += window[i][j].B * h[i][j];
        }
    }
    rgb_pixel output;
    output.R = r / 16;
    output.G = g / 16;
    output.B = b / 16;
    return output;
}

void video_2dfilter(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
                     rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    rgb_pixel window[3][3];
    row_loop: for (int row = 0; row < MAX_HEIGHT; row++) {
        col_loop: for (int col = 0; col < MAX_WIDTH; col++) {
            #pragma HLS pipeline
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    int wi = row + i - 1;
                    int wj = col + j - 1;
                    if (wi < 0 || wi >= MAX_HEIGHT || wj < 0 || wj >= MAX_WIDTH) {
                        window[i][j].R = 0;
                        window[i][j].G = 0;
                        window[i][j].B = 0;
                    } else
                        window[i][j] = pixel_in[wi][wj];
                }
            }
            if (row == 0 || col == 0 || row == (MAX_HEIGHT - 1) || col == (MAX_WIDTH - 1)) {
                pixel_out[row][col].R = 0;
                pixel_out[row][col].G = 0;
                pixel_out[row][col].B = 0;
            } else
                pixel_out[row][col] = filter(window);
        }
    }
}

```

图9.9: 没有使用线性缓冲区的2D滤波代码

如图9.10所示代码，是使用线性缓冲区和窗口缓冲区方式实现的，其中代码实现过程如图9.11所示。代码每次执行一次循环时，窗口会移动一次，使用来自于1个输入像素和两个来自于行缓冲区缓存的像素来填充窗口缓冲区。另外，输入的新像素被移入线性缓冲区，准备被下一行的窗口运算过程所使用。请注意，由于为了每个时钟周期处理一个像素并输出结果，所以系统必须在每个时钟周期内完成对窗口缓冲区数据的读取和写入。另外，当展开“i”循环后，对于窗口缓冲区数组而言，每个数组索引都是一个常量。在这种情况下，Vivado@HLS将转换数组中的每个元素成一个标量变量（一个成为标量化的过程）。窗口数组中的元素随后使用触发器(FF)编译实现。同样，线性缓冲区的每一行数据都会被访问两次（被读取一次和写入一

次）。示例代码中明确约束行缓冲区中每一行数组元素被分割成到一块单独存储区中。根据MAX WIDTH的可能取值情况，最后决定使用一个或者多个Block RAM实现。注意，每个Block RAM可在每个时钟周期支持两次独立访问。

线性缓冲区是应用于模块式计算的重用缓冲区的一种特殊例子。如图9.9中所示，重用缓冲区和线性缓冲区的高级综合是目前一个热点研究领域。参考[8, 31]。

Vivado@HLS包含 `hls::linebuffer<>` 和 `hls::window buffer<>` 类，它们可以简化窗口缓冲区和行缓冲区的管理。

对于 3×3 的图像滤波器核，则存储每个像素占用4字节的 1920×1080 图像的一行数据需要使用多少个FPGA片上Block RAM？

9.2.2 因果滤波器

图9.10中实现的滤波器是每个时钟周期读取一个输入像素，计算出一个输出像素。该滤波器实现原理与图9.9中所示不完全相同。输出结果是从先前读取像素的窗口缓冲区中数据计算出来的。窗口缓冲区中取数顺序是“向上和向左”。因此，输出图像相对于输入图像是“向下和向右”移动的。这种情况类似于信号处理中的因果滤波器和非因果滤波器的原理。大多数信号处理理论的重点是因果滤波器。因为只有因果滤波器对于时间采样信号（例如，其中 $x[n] = x(n * T)$ 和 $y[n] = y(n * T)$ ）才有实用价值。

在因果滤波器

```

void video_2dfilter_linebuffer(rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
                                rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface ap_hs port=pixel_out
    #pragma HLS interface ap_hs port=pixel_in
    rgb_pixel window[3][3];
    rgb_pixel line_buffer[2][MAX_WIDTH];
    #pragma HLS array_partition variable=line_buffer complete dim=1
    row_loop: for (int row = 0; row < MAX_HEIGHT; row++) {
        col_loop: for (int col = 0; col < MAX_WIDTH; col++) {
            #pragma HLS pipeline
            for(int i = 0; i < 3; i++) {
                window[i][0] = window[i][1];
                window[i][1] = window[i][2];
            }
            window[0][2] = (line_buffer[0][col]);
            window[1][2] = (line_buffer[0][col] = line_buffer[1][col]);
            window[2][2] = (line_buffer[1][col] = pixel_in[row][col]);
            if (row == 0 || col == 0 ||
                row == (MAX_HEIGHT - 1) ||
                col == (MAX_WIDTH - 1)) {
                pixel_out[row][col].R = 0;
                pixel_out[row][col].G = 0;
                pixel_out[row][col].B = 0;
            } else {
                pixel_out[row][col] = filter(window);
            }
        }
    }
}

```

图 9.10: 使用线性缓冲区实现2D滤波器的代码

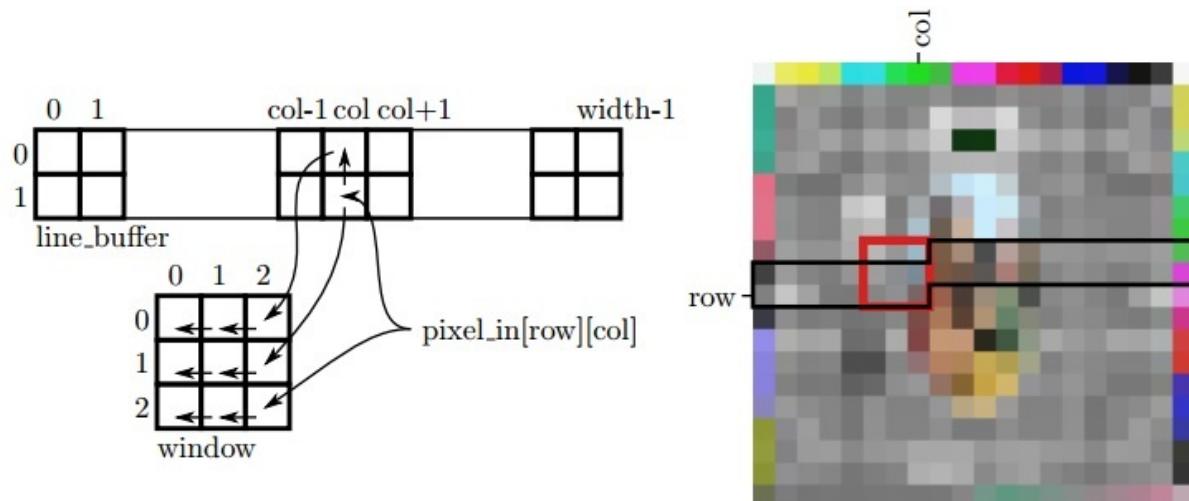


图9.11:图中所示为图9.10中的代码实现的存储器。 存储器在特定的循环周期时将从线性缓冲区中读取出数据存储在窗口缓冲区的最右端区域。 黑色的像素存储在行缓冲区中，红色的像素存储在窗口缓冲区中。

通过卷积的定义证明前面原理：

就本书而言，大多数变量不是时间采样信号和单输入单输出系统。对于设计时间采样信号的系统，在使用HLS开发过程中可以进行时序约束处理。只要系统能达到所需的任务延迟，那么就认为系统设计是正确的。

在大多数视频处理算法中，在上文代码中所引入的空间位移不是预期的，是需要被消除的。虽然有很多种修改代码的方法来解决这个问题，但是一种常见的方式是扩展迭代域。这种技术是通过增加少量的循环边界，以便第一次循环迭代就读取第一个输入像素，但是第一个输出像素直到等到后面的迭代空间才写出。修改后的版本滤波器代码如图9.12所示。代码原理如图9.14所示，和图9.10中原始缓冲区中代码相同。使用HLS编译后，可以看出数据依赖性以完全相同的方式得到满足，并且待综合的代码是硬件可实现的。

9.2.3 边界条件

在大多数情况下，计算缓冲区窗口只是输入图像的一个区域。然而，在输入图像的边界区域，滤波器进行计算的范围将超过输入图像的边界。根据不同应用的要求，会有很多不同的方法来解决图像边界处的计算问题。也许最简便的方法来解决边界问题就是忽略边界，这样输出的图像就会比输入的图像长宽各少滤波器核大小。但是在输出图像大小固定的应用中，如数字电视，这种方法就行不通了。

```

void video_2dfilter_linebuffer_extended(
    rgb_pixel pixel_in[MAX_HEIGHT][MAX_WIDTH],
    rgb_pixel pixel_out[MAX_HEIGHT][MAX_WIDTH]) {
    #pragma HLS interface ap_hs port=pixel_out
    #pragma HLS interface ap_hs port=pixel_in
    rgb_pixel window[3][3];
    rgb_pixel line_buffer[2][MAX_WIDTH];
    #pragma HLS array_partition variable=line_buffer complete dim=1
    row_loop: for(int row = 0; row < MAX_HEIGHT+1; row++) {
        col_loop: for(int col = 0; col < MAX_WIDTH+1; col++) {
            #pragma HLS pipeline II=1
            rgb_pixel pixel;
            if(row < MAX_HEIGHT && col < MAX_WIDTH) {
                pixel = pixel_in[row][col];
            }
            for(int i = 0; i < 3; i++) {
                window[i][0] = window[i][1];
                window[i][1] = window[i][2];
            }
            if(col < MAX_WIDTH) {
                window[0][2] = (line_buffer[0][col]);
                window[1][2] = (line_buffer[0][col] = line_buffer[1][col]);
                window[2][2] = (line_buffer[1][col] = pixel);
            }
            if(row >= 1 && col >= 1) {
                int outrow = row-1;
                int outcol = col-1;
                if(outrow == 0 || outcol == 0 ||
                   outrow == (MAX_HEIGHT-1) || outcol == (MAX_WIDTH-1)) {
                    pixel_out[outrow][outcol].R = 0;
                    pixel_out[outrow][outcol].G = 0;
                    pixel_out[outrow][outcol].B = 0;
                } else {
                    pixel_out[outrow][outcol] = filter(window);
                }
            }
        }
    }
}

```



图9.13:不同滤波器滤波后的效果图。图b时图9.9中代码生成效果图，图c是由图9.10中代码生成效果图，图d是由图9.12中代码生成效果图与图像b相同

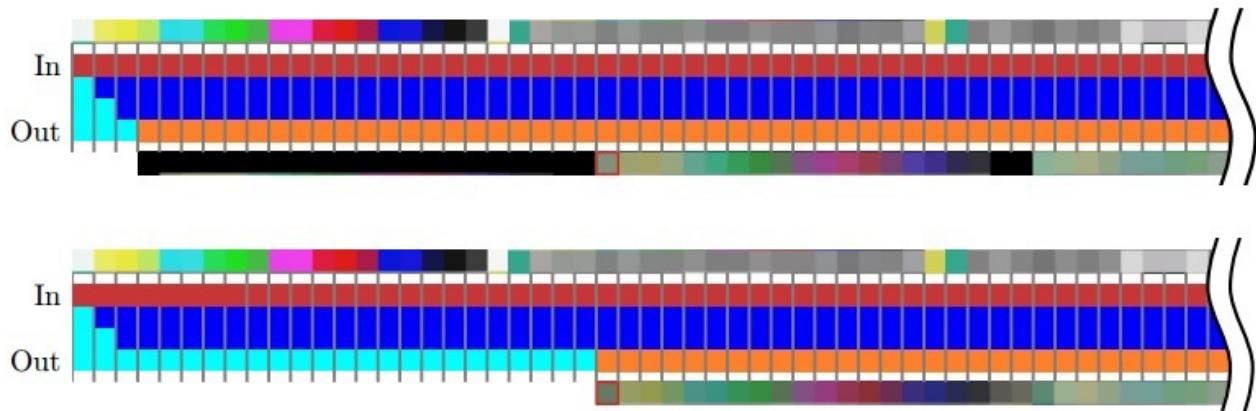


图9.14：使用线性缓冲区方法代码执行时序。上方执行顺序为图9.10中代码执行时序。下方执行时序为图9.12中代码执行时序。红色框标记的像素在两种实现中以相同的周期输出，在上方执行时序中表示第二行第二个像素，在下方时序中表示第一行第一个像素



图9.15：不同边界条件处理方法效果图

另外，当处理大量大小略微不同的图片时，需要一系列大小的滤波器核来进行处理，这种情况就复杂了。图9.10中的代码展示，输出图像通过在边界处填充已知值的像素（一般填充黑色）来达到与输入图像大小相同。或者通过其他方式来合成缺失的值。

- 缺少的输入值可以用常量来填充；

- 可以从输入图像的边界像素处填充缺少的输入值；
- 缺失的输入值可以在输入图片内部像素中进行重构。

当然，也存在更复杂和更计算密集的方案。图9.16中展示了一种处理边界条件的方案。此方案是通过计算每个像素在窗口缓冲区中的偏移地址方法。这种方案的一个明显的缺点就是每次从窗口缓冲区中读取的地址都是变化的地址。因此，在计算滤波之前，这个变量会导致多路复用。对于 $N \times N$ 的抽头滤波器，针对N个输入将会消耗大约 $N \times N$ 个多路复用器。对于滤波器而言，多路复用器占滤波器资源消耗的大部分。另一种处理数据写入窗口缓冲区时的边界条件是以规则模式移动窗口缓冲区。在这种情况下，只消耗N个多路复用器，而不是 $N \times N$ 个，从而是资源消耗量降低很多。

修改图9.16中的代码，从窗口缓冲区中读取数据使用常量地址。你节省了多少硬件资源？

9.3 结论

视频处理是一种常见的非常适合使用HLS实现的FPGA应用。大多数视频处理算法有一个共同特点是数据局部性，即可以在使用少量的外部存储器访问的情况下，能够实现流式传输和本地缓存的应用。

```

void video 2dfilter linebuffer extended constant(
    rgb pixel pixel in[MAX HEIGHT][MAX WIDTH], rgb pixel pixel out[MAX HEIGHT][MAX WIDTH]) {
    #pragma HLS interface ap hs port=pixel out
    #pragma HLS interface ap hs port=pixel in
    rgb pixel window[3][3];
    rgb pixel line buffer[2][MAX WIDTH];
    #pragma HLS array partition variable=line buffer complete dim=1
    row loop: for(int row = 0; row < MAX HEIGHT+1; row++) {
        col loop: for(int col = 0; col < MAX WIDTH+1; col++) {
            #pragma HLS pipeline II=1
            rgb pixel pixel;
            if(row < MAX HEIGHT && col < MAX WIDTH) {
                pixel = pixel in[row][col];
            }
            for(int i = 0; i < 3; i++) {
                window[i][0] = window[i][1];
                window[i][1] = window[i][2];
            }
            if(col < MAX WIDTH) {
                window[0][2] = (line buffer[0][col]);
                window[1][2] = (line buffer[0][col] = line buffer[1][col]);
                window[2][2] = (line buffer[1][col] = pixel);
            }
            if(row >= 1 && col >= 1) {
                int outrow = row-1;
                int outcol = col-1;
                rgb pixel window2[3][3];
                for (int i = 0; i < 3; i++) {
                    for (int j = 0; j < 3; j++) {
                        int wi, wj;
                        if (i < 1 - outrow) wi = 1 - outrow;
                        else if (i >= MAX HEIGHT - outrow + 1) wi = MAX HEIGHT - outrow;
                        else wi = i;
                        if (j < 1 - outcol) wj = 1 - outcol;
                        else if (j >= MAX WIDTH - outcol + 1) wj = MAX WIDTH - outcol;
                        else wj = j;
                        window2[i][j] = window[wi][wj];
                    }
                }
                pixel out[outrow][outcol] = filter(window2);
            }
        }
    }
}

```

图9.16: 使用线性缓冲区核常量边界来实现处理边界条件的代码，这种方法处理消耗硬件资源成本很大

第十章 排序算法

10.1 简介

排序是许多系统中最常见的一个通用算法。它的核心算法是在处理大量排序数据时，用二叉树检索法进行高效处理，其时间复杂度仅为 $O(\log N)$ 。比如，数序序列

$$A = 1, 4, 17, 23, 31, 45, 74, 76$$

从这个序列中判断是否存在一个数，不需要将它与所有8个元素进行比较，仅仅需要从排序序列中间元素选取并检查需要判断的数字是否大于或小于元素。如果检索元素45，第一步，可以从比较 $A(4)=31$ 来开始，从45到31我们可以消除 $A(0 \dots 4)$ 只考虑 $A(5 \dots 7)$ ，即

$$A = 1, 4, 17, 23, 31, 45, 74, 76$$

下一步，如果与 $A(6)=74$ 比较，可以得到 $45 < 74$ ，可以消除除 $A(5)$ 以外的所有情况，此时

$$A = 1, 4, 17, 23, 31, 45, 74, 76$$

最终确认A(5)确实包含在序列中。

在上面的例子中，序列A可以代表各种不同的概念。A可以代表一个数学集合，可以搜索元素是否在集合内。在集合中，A也可以表示数据中的一部分，通常称为关键字，用于索引其余信息。关键字可以是一个人的名字，基于关键字的搜索，可以获取该人的其余数据信息，比如他们的出生日期等等。在有些情况下，关键字还可能是更抽象的对象，比如一些数据或密钥的加密散列。在这种情况下，数据存储顺序可能是随机的，但只要获得正确的密码散列，仍然可以搜索它。不管何种情况，排序和搜索所需的基本操作类似，都要需要比较两个不同的值，在这一章中将忽略这些问题。

在通用处理器系统中，有各种各样的排序算法应用，参考[36]。它们在空间复杂度和时间复杂度方面有所不同，但绝大多数都需要 $O(N \log N)$ 次比较才能对N个元素进行排序。对一个排序序列使用二叉树插入1个新元素时，搜索 $O(\log N)$ 次即能找到新值的正确位置；当需要插入N个元素时，这个过程需要重复N次。

在实际应用场景下，插入元素的成本可能非常大，这取决于排序的数据结构。在通用处理器系统中，有各种各样的因素影响综合性能，比如处理大数据集或多核之间并行化计算时的局部存储问题。HLS中也有类似的考虑，通常以增加资源开销为代价，来降低计算处理的时间。在很多实际案例中，要取得最佳设计往往要从算法和实现技术方式两个方面来综合考虑。算法和实现技术的权衡目前是一个新兴的研究领域，参考[45, 54, 49]。

性能外的特性也影响排序算法的选择。例如：

- 稳定：如果输入数据有两个字段具有相同的键值，它们以相同的顺序输出，那么排序是稳定的。例如，对一组记录进行排序，使用年龄作为排序主键，包含人名和年龄。在输入数据中，两人都是25岁，John排在Jane前面。一个稳定的排序需要确保John和Jane的位置跟输入数据保持一致。
- 在线：算法允许数据在接收时进行排序。在数据排序时不可以访问，或者必须从外部存储器顺序读取等情况，在线是非常必要的措施。
- 原位：一个包含N个元素的数组可以使用N个元素存储空间进行排序，也有一些算法在排序过程中需要额外的存储。
- 适应：对于已经排序的数据来说是有效的。如果数据已经排序，一些算法可能会运行得更快，即只需要 $O(N)$ 的时间复杂度。

10.2 插入排序

插入排序是一种基本的排序算法，其核心思想是将一个新的元素插入到一个有序数组中，并继续保持有序。每步将一个待排序的记录，按其关键码值的大小插入前面已经排序的文件中适当位置上，直到全部插入完为止。

例如有一个长度为 N 的无序数组，进行 $N - 1$ 次的插入即能完成排序；第一次，数组第1个数认为是有序的数组，将数组第二个元素插入仅有1个有序的数组中；第二次，数组前两个元素组成有序的数组，将数组第三个元素插入由两个元素构成的有序数组中……第 $N - 1$ 次，数组前 $N-1$ 个元素组成有序的数组，将数组的第 N 个元素插入由 $N-1$ 个元素构成的有序数组中，则完成了整个插入排序。

例如对输入数组A进行排序，先考虑数组第1个数 $A[0]$ 视为元素个数为1的有序序列；下一步考虑数组的第二个元素 $A[1]$ 插入这个有序序列中；然后依次把数组A中的每个元素 $A[i]$ 插入到这个有序序列中，因此，需要一个外部循环，扫描数组A的每一个元素。在每一次插入过程中，假设这是要将 $A[i]$ 插入到前面的有序序列中，需要将 $A[i]$ 和 $A[0 \dots i - 1]$ 进行比较，确定要插入的合适位置，这就需要一个内部循环。图10.1展示一个数组插入排序的单步视图。第一趟第1个元素值3是有序的，因为任何只带有一个元素的数组都是按顺序排列的。第二趟插入数组元素值2，与第一个元素值3比较后，被放置到有序数组中的第一个元素，将之前有序数组中的元素值3移到右边。第三趟插入数组的第三个元素 $A[2] = 5$ ，由于其已经位于正确的位置，因此无需做任何移动操作。第四趟把数组元素4插入到有序数组中元素5前面即可。最后一趟插入数组元素值1时，元素1需要插入到有序数组中的第一个元素位置，前面的有序数组元素全部右移一个位置。

$\{ \underline{3}, 2, 5, 4, 1 \}$
 $\{ 3, \underline{2}, 5, 4, 1 \}$
 $\{ 2, \underline{3}, 5, 4, 1 \}$
 $\{ 2, 3, \underline{5}, 4, 1 \}$
 $\{ 2, 3, 4, \underline{5}, 1 \}$
 $\{ 2, 3, 4, 5, \underline{1} \}$
 $\{ 1, \underline{2}, 3, 4, 5 \}$

图10.1 插入排序算法在数组上操作。初始数组显示在上面，在算法的每个步骤中，都要比较下划线元素并将其按按顺序排序在左边；在每个阶段，阴影部分按顺序排列。

插入排序是一种稳定、在线、原位、自适应的排序算法。插入排序通常用在处理小批量数组。例如，复杂算法产生的大量数据可以分解为若干个组小批量的数组，然后利用插入排序处理这些小批量的数组，最终结果通过组合排序数组形成。

10.2.1 插入排序的基本实现

```

#include "insertion_sort.h"
void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(int i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        int j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && A[j-1] > item) {
            #pragma HLS pipeline II=1
            A[j] = A[j-1];
            j--;
        }
        A[j] = item;
    }
}

```

图10.2：插入排序的参考代码。外部循环`for`用来遍历数组元素，内部循环`while`用来遍历已经排序的序列，将当前元素移动到对应位置。

图10.2是插入排序的C参考代码。外部循环标记为L1，由于单个元素 $A[0]$ 已经排序，这里仅需要从元素 $A[1]$ 迭代到元素 $A[SIZE - 1]$ ，其中 $SIZE$ 表示数组元素的长度。L1的每次循环首先拷贝当前要插入有序序列中的元素 $A[i]$ ，然后再执行内部L2循环寻找该值索引的合适位置。内部循环的判断条件是还没有循环到数组末尾(即条件为 $j > 0$)，并且数组元素是大于即将要插入的元素(即条件 $A[j - 1] > item$)。只要该判断条件满足，排序子数组的元素右移(操作 $A[j] = A[j - 1]$)；当判断条件不满足时，即已经查询到要插入元素的坐标位置；此时退出循环，找到索引的正确位置直接插入元素。当第*i*次迭代完成后，从 $A[0]$ 到 $A[i]$ 的元素按排序顺序排列。

图10.2中的插入排序代码是一个简单的实现，没有做任何优化。这里可以使用Vivado HLS不同的优化指令，如**pipeline**, **unroll** 和 **array_paration**等进行优化。最简单的优化策略即是使用**pipeline**指令，使得内部循环L2支持流水功能。对于插入排序算法，内部循环访问不同的数组元素时，没有数据相关性，因此设置期望的流水线启动间隔为1(即II=1)。生成的加速器平均需要 $N^2 / 4$ 次数据比较，由于每个时钟周期都需要比较一次，因此流水线的处理延迟大概需要 $N^2 / 4$ 个时钟周期，参考[58]。实际上，外部循环的顺序执行的计算延迟稍高。为了获取更好的性能，可以尝试将**pipeline**指定应用到外部循环L或者函数本身，或者还可以使用部分循环展开组合。插入排序的一些优化选项可以参考表10.1。

表10.1：图10.2中插入排序的C代码实现可以尝试的优化选项

	指令 (Directives)	启动间隔 (II)	周期 (period)	逻辑资源Slices
1	L2: pipeline II=1	?	?	?
2	L2: pipeline II=1	?	?	?
	L2: unroll factor=2			
	array partition variable=A cyclic factor=2			
3	L1: pipeline II=1	?	?	?
4	L1: pipeline II=1	?	?	?
	L1: unroll factor=2			
	array partition variable=A complete			
5	function pipeline II=1	?	?	?
	array partition variable=A complete			

探索一下表10.1中的优化选项，分别综合这些优化设计并确定启动间隔(II)、时钟周期和所需要的逻辑资源，分析综合报告，获取哪些优化选项在改进延迟和吞吐量方面效果最明显。如果合并这些优化选型到插入排序算法设计里面会发生什么？

图10.2所示的插入排序代码与前面几章的其他嵌套循环程序比较类似，但是有一些方面确实很难优化。优化选项1尝试的流水线启动间隔也无法达到1，虽然数据通路上并没有重要的相关性，但是在控制通路上循环是否能够执行的判断影响了其流水线性能。在这种情况下，必须读取 $A[i - 1]$ 来确定循环是否能够执行，而该循环检查并不能在读取数据A的第一级流水线完成。这是典型的递归例子，其包含了HLS生成的循环控制逻辑。如果碰到类似的递归逻辑，Vivado HLS报告中会指出循环退出条件不能在第一个启动间隔的时钟周期中去调度；当然也有可能会发生退出或者保持当前控制逻辑的状态。一种解决方案是增加一条读取 $A[i - 1]$ 的操作，才能保证循环退出检查在1个启动间隔周期内调度，其代码如图10.3所示。

```
#include "insertion_sort.h"
void insertion_sort(DTYPE A[SIZE]) {
    L1:
    for(i = 1; i < SIZE; i++) {
        DTYPE item = A[i];
        j = i;
        DTYPE t = A[j-1];
        L2:
        while(j > 0 && t > item) {
            #pragma HLS pipeline II=1
            A[j] = t;
            t = A[j-1];
            j--;
        }
        A[j] = item;
    }
}
```

图10.3：根据表10.1中的优化选项1重构插入排序

表中的优化选项2是以factor因数为2展开内部循环，即在每一个时钟周期内完成两个移位操作，可能会降低插入排序的延迟。由于每次存储访问不能分配到不同的存储区域，因此即使添加数组分区指令优化，Vivado HLS也不能达到启动间隔为1的循环。

在Vivado HLS中，数组分区array_partition优化指令是用来实现多组独立的存储块区域。例如优化指令（array_partition variable=A cyclic factor=4）表示划分数组A中为4个独立的子存储块，通常这个指令使得每个时钟周期内存储访问速度是未优化前的4倍，但是其限制是一个时钟周期内的每次存储访问必须对应其中的一块子存储块。数组元素A[i]可以存放在任意一个分区，而数组元素A[4*i+2]仅可以访问第3个子模块；需要额外的交叉逻辑来解决同一个周期内同时访问A[i]、A[i+1]、A[i+6]和A[i+7]。在编译时只能保证静态偏移量下的访问可以抵达不同的存储块，但是实际的存储块直到运行时才能确定。当然在访问不同存储块区域时，还可以使用缓冲逻辑保证那些不能在一个周期内抵达的访问。要是访问同一存储块区域，缓冲逻辑可以延迟几个周期直到所有访问完成。多端口的架构设计通过一个或两个物理端口获取数据副本，保证在每个时钟周期中有一定数量的访问可以完成，参考参考[\[62,1,37\]](#)。

表中的优化选项3也未能实现显著的改进，主要原因是内部循环L2没有静态可计算的环路，Vivado HLS无法重构外部循环L1的流水线。这里主要关注插入算法在指令优化上的设计空间，好的替代方案还需要进队代码重构。要设计出最佳的代码重构不仅需要理解算法，还需要理解HLS的综合的体系结构，参考参考[\[28,48\]](#)。接下来的小节会主要对插入排序的代码进行重构，其代码结构与图10.2中的有很大差异。

设计一个最佳基于HLS的算法加速需要考虑下面几个方面。首先，编写高效的高层综合语言需要工程师是必须理解硬件设计，比如循环展开和存储划分等一些基本思路。其次，工程师需要对应用程序和硬件实现都有足够的理解，才能正确评估任何吞吐量问题。第三，也是最重要的，为了达到最佳的设计结果，即高性能和资源的低开销，通常需要重构代码才能创建一个高效的硬件结构。这些与软件设计有很大的差异。

10.2.2 并行化插入排序

为了提高插入排序的性能，优化目标是每个时钟周期插入一个新元素。当最后一个元素插入到排序列表中时，可能会需要改变数组中的所有元素。对于图10.2中的代码，意味着内部循环L2实际上顺序扫描比较了数组中的所有元素。要想在每个时钟周期中插入一个新元素，首先需要足够的硬件操作资源，才可以对数组中的每个元素进行比较。为了保证外部循环的流水线，优化内部循环的变量边界为常量边界，才可以使得内部循环体展开并集成到外部循环体中，参考代码如图10.4所示。

```

#include "insertion_sort_parallel.h"
#include "assert.h"
void insertion_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
    #pragma HLS array_partition variable=B complete
    L1:
    for(int i = 0; i < SIZE; i++) {
        #pragma HLS pipeline II=1
        DTYPE item = A[i];
        L2:
        for(int j = SIZE-1; j >= 0; j--) {
            DTYPE t;
            if(j > i) {
                t = B[j];
            } else if(j > 0 && B[j-1] > item) {
                t = B[j-1];
            } else {
                t = item;
                if (j > 0)
                    item = B[j-1];
            }
            B[j] = t;
        }
    }
}

```

图10.4：重构表10.1中优化选型3的插入排序代码

重构图10.2中的内部循环体的实现代码，把退出条件优化为if条件，放在新的内部循环体L2中；增加其他分支判断条件，扩大内部循环的次数；增加的分支判断条件在原始循环时不执行任何操作，也不会被执行；数组的赋值在循环体L2内部执行，而不是在L2循环体外部执行。内部循环展开时， j 都成为常量，数组B的每次读写都将在常量索引下执行。另一方面， $item$ 变量是分配给内部循环的一份元素拷贝；在编译期间Vivado HLS会创建一个独立的寄存器用来实现电路上的多路复用。

将单个变量转换为多个副本是编译器里常用的内部转换优化机制，这种中间表示形式即是静态单一赋值（SSA）。要合并来自代码中不同点的值，SSA中间表示形式包括 ϕ 函数都是由希腊字母 ϕ 表示。这些 ϕ 函数在Vivado HLS产生的电路中生成多路复用器，如果您仔细查看，你可能会发现工具中相关的资源报告。

图10.4中的并行插入排序本质上生成了多组内部循环体，这个内部循环体的结构主要是由几个多路复用器、一个决定谁最小的比较器和1个存储数组元素的寄存器等组成。当然，每个阶段还可能包括缓冲寄存器，以确保生成的电路逻辑在一个有效的时钟频率。若把内部循环体L2看为排序单元，那么插入排序函数即是由一个一维数组的排序单元和一些在输入输出接口层的额外逻辑组成，在这种情况下，外部迭代体仅需要 $SIZE$ 个时钟周期就可以处理完。这个排序单元的主要特性是每个排序单元只与相邻的排序单元通信，而不是所有的单元。像这样的设计被称为脉动阵列，是一种常见的并行算法优化的技术。在很多情况下，只要在不同的循环之间的通信是有限的，当展开内部循环的优化，包括排序和脉动阵列的实现，都可以称为隐式脉动阵列。

10.2.3 显式脉动阵列插入排序

脉动阵列已经有了很好的研究，许多并行算法都采用了脉动阵列的技术，特别是使用线性排序单元阵列来实现插入排序，参考参考[59,9,45,5]。脉动阵列本身的核心概念就是让数据在运算单元的阵列中进行流水，减少访存的次数，并且使得结构更加规整，布线更加统一，提高频率。这里描述一种显式的数据流编码风格，通过**dataflow**优化指令就可以实现脉动阵列的直观方法。

图10.5展示了插入排序的脉动阵列的实现方式。每个单元都是相同的，每个单元的输入端口 in 用来接收当前寄存器中的值，较小的值发送到输出端口 out ，而较大的值存放在本地寄存器 $local$ 中，其实每个单元的功能也就是 $out = \min(in, local)$ 。第 i 号单元的输出结果传递给线性阵列的下一个（即第 $i + 1$ 号）单元的输入，当新的输入到来时，会与存储在阵列中的本数组进行比较，直到找到正确的位置。如果新的输入大于阵列中的所有值，排序后的值将向右移动一个单元阵列；如果新的输入小于阵列中的所有值，此值会在阵列中传输，最终会被存放在最右边的阵列单元中。当所有的数据都处理完时，最小的元素存放在第 $N - 1$ 个阵列单元，直接从输出读出即可。

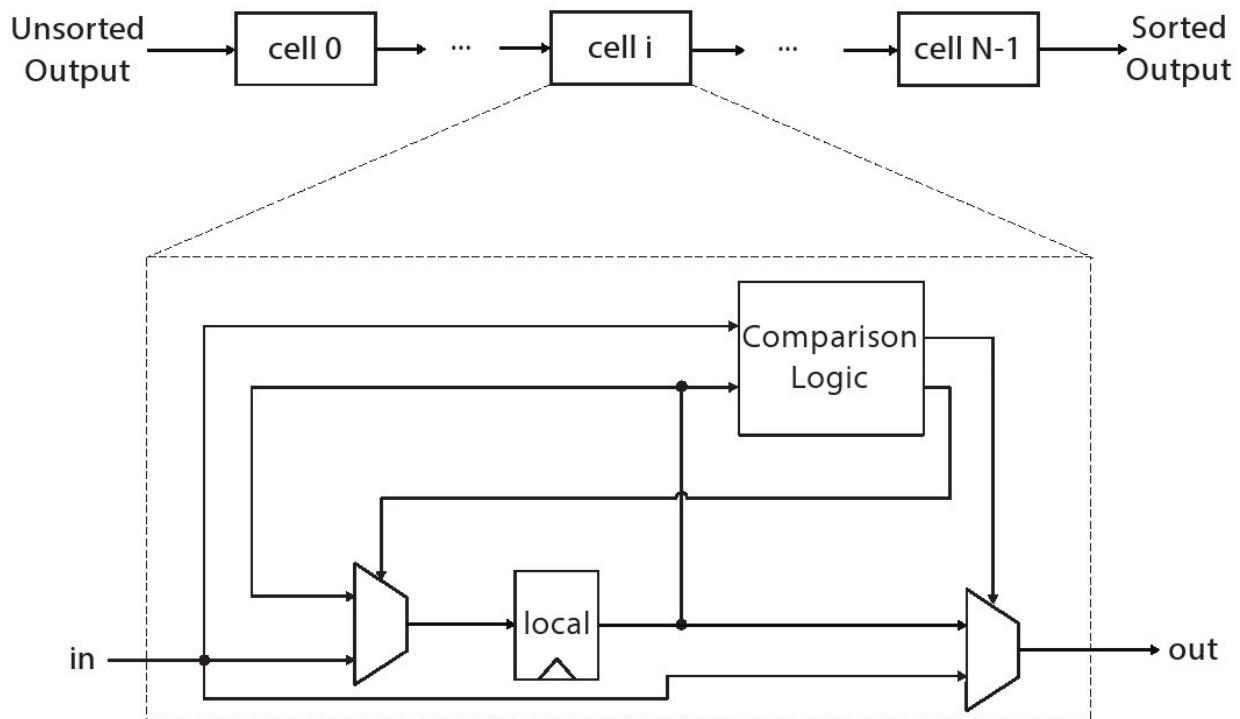


图10.5：插入单元的体系结构。每个单元只存放一个本地值，在每次执行中，获取输入值，将其与本地值进行比较，并把较小的值写入输出端口。排序 N 个元素需要 N 个单元。

一个插入单元的核心代码如图10.6所示。输入和输出变量都声明为HLS的数据流类型。DTYPE是一个类型参数，允许对不同类型进行操作。本地变量存放阵列中的某一个元素，添加了**static**关键字是为了在多个函数调用中保存它的值。这里需要注意的是使用相同的函数，复制单元功能 N 次；每个单元必须有一个单独的单元静态变量，一个静态变量不能跨 N 个函数共享。

```

void cell0(hls::stream<DTYPE> & in, hls::stream<DTYPE> & out)
{
    static DTTYPE local = 0;
    DTTYPE in_copy = in.read();
    if(in_copy > local) {
        out.write(local);
        local = in_copy;
    }
    else
    {
        out.write(in_copy);
    }
}

```

图10.6：一个插入单元cell0对应的Vivado HLS C代码，其他单元格除了需要具体不同的函数名（比如cell1、cell2等）之外，其他的功能代码都时相同的。代码与图10.5中架构图所展示的功能相同，输入和输出变量采用了HLS的数据流接口，此接口提供了一种抽象层次更高跟方便的方法去生成和仿真FIFO。

图10.7展示了8个元素进行排序的代码，实现的主要功能即是将8个插入单元连接在一起。若需要扩展更多的元素，只需要复制更多的单元函数即可。HLS定义并实例化了单元之间的数据流变量类型，并以合适的方式连接每个单元的输入和输出接口。

```

void insertion_cell_sort(hls::stream<DTYPE> &in, hls::stream<DTYPE> &out)
{
    #pragma HLS DATAFLOW
    hls::stream<DTYPE> out0("out0 stream");
    hls::stream<DTYPE> out1("out1 stream");
    hls::stream<DTYPE> out2("out2 stream");
    hls::stream<DTYPE> out3("out3 stream");
    hls::stream<DTYPE> out4("out4 stream");
    hls::stream<DTYPE> out5("out5 stream");
    hls::stream<DTYPE> out6("out6 stream");
    // Function calls;
    cell0(in, out0);
    cell1(out0, out1);
    cell2(out1, out2);
    cell3(out2, out3);
    cell4(out3, out4);
    cell5(out4, out5);
    cell6(out5, out6);
    cell7(out6, out);
}

```

图10.7：插入单元对8个元素进行排序，函数的输入是HLS的数据流接口，输出是通过变量out按顺序的输出元素，输出顺序从最小的元素开始，以最大的元素结束。

上面insertion_cell_sort函数是从最小值开始输出数据，后面的输出越来越大的元素。如果要首先输出最大的元素，后面的输出越来越小，需要做哪些改进，才可能反转这个输出顺序？

为了对整个元素进行排序，需要多次调用insertion_cell_sort子函数。每一个插入单元排序的调用都提供了一个数据元素进行排序；第一次调用时，这些数据元素将存放在哪里呢？这个问题的解决方法是把输入数据在初始化成局部变量0，这个在所有的单元函数中完成的。

将静态变量local初始化为0是对数据类型的假设。这种假设的成立条件是什么？换句话说，范围是多少才能被正确处理输入数据的值？如果输入数据在insertion_cell_sort函数这个范围之外？有没有更好的方法去初始化局部变量？

在调用了插入单元排序函数8次后，所有的数据都存放在每个单元格函数的8个局部变量中。

要得到第一个排序元素，需要调用insertion_cell_sort函数多少次？对于所有元素数据的排序，需要调用多少次？如果把数组拆成N个单元，又需要多少次调用insertion_cell_sort函数才能完成整个排序？

指令dataflow的作用是实现了8个插入单元函数组成任务流水线结构。在顶层函数的每个执行过程中都流水地处理一个新的输入示例，流水线的每个阶段都去调用单元函数。当调用单元函数8次时，即只能对序列排序8个值；在单元函数之间不包含递归，因此在资源足够的情况下可以任意扩展和调用单元函数。

整个数组的插入排序需要多少个周期？所有排序的数据都是从插入单元函数中的参数输出的吗？如果删除dataflow优化指令，循环计数将如何变化？如何改进资源利用率？

图10.8所示的是testbench的参考代码，testbench生成要排序的随机数据存放在input数组中，多次调用insertion_cell_sort函数进行排序，处理结果存放在cell_output数据中。接下来，使用同样的输入数据，插入排序处理过程采用图10.2所示的insertion_sort函数。最后，testbench比较这两种不同实现方式的结果，如果两种不同实现方式的排序结构相同，则testbench通过。

```

#include "insertion_cell_sort.h"
#include <iostream>
#include <stdlib.h>

const static int DEBUG=1;
const static int MAX_NUMBER=1000;
int main () {
    int fail = 0;
    DTYPE input[SIZE];
    DTYPE cell_output[SIZE] = {0};
    hls::stream<DTYPE> in, out;

    //generate random data to sort
    if(DEBUG) std::cout << "Random Input Data\n";
    srand(20); //change me if you want different numbers
    for(int i = 0; i < SIZE; i++) {
        input[i] = rand() % MAX_NUMBER + 1;
        if(DEBUG) std::cout << input[i] << "\t";
    }

    //process the data through the insertion_cell_sort function
    for(int i = 0; i < SIZE*2; i++) {
        if(i < SIZE) {
            //feed in the SIZE elements to be sorted
            in.write(input[i]);
            insertion_cell_sort(in, out);
            cell_output[i] = out.read();
        } else {
            //then send in dummy data to flush pipeline
            in.write(MAX_NUMBER);
            insertion_cell_sort(in, out);
            cell_output[i-SIZE] = out.read();
        }
    }

    //sort the data using the insertion_sort function
    insertion_sort(input);

    //compare the results of insertion_sort to insertion_cell_sort; fail if they differ
    if(DEBUG) std::cout << "\nSorted Output\n";
    for(int i = 0; i < SIZE; i++) {
        if(DEBUG) std::cout << cell_output[i] << "\t";
    }
    for(int i = 0; i < SIZE; i++) {
        if(input[i] != cell_output[i]) {
            fail = 1;
            std::cout << "golden=" << input[i] << "hw=" << cell_output[i] << "\n";
        }
    }

    if(fail == 0) std::cout << "PASS\n";
    else std::cout << "FAIL\n";
    return fail;
}

```

图10.8 : *insertion_cell_sort*函数的测试平台

在testbench中，常量SIZE的值表示要排序元素的数量，在本节的运行示例中即为8。常量DEBUG是用来提供testbench中的一些输出详细说明；设置成非0表示打开debug信息，设置成0表示关闭debug信息。输入数组Input中的数据是由rand函数随机产生的；如果需要更新产生的数据，可以调用带参数信息的srand函数。

值得注意的是，testbench中一共调用了 $SIZE * 2$ 次insertion_cell_sort函数。前面SIZE次调用都会向insertion_cell_sort函数中输入一个元素，但不会产生有效的结果输出；后面SIZE次的调用提供虚拟无效的输入数据，并产生一个已经排序的元素，从最小的元素开始输出。

在排序操作开始进行时，图10.6中的代码是在不同的状态下的本地局部变量，这意味着只能对一个数组进行排序。在大多数实际场景中，不仅需要对多个数组进行排序，还希望在不存在气泡的情况下，对每个数组进行反向处理。推荐大家改进代码和testbench，实现对多个数组的排序。

10.3 归并排序

归并操作是 John von Neumann在1945年发明的一种稳定的分治算法，参考[36](#)。归并排序算法的基本思想是将两个有序序列合并成一个较大的有序序列，可以在 $O(N)$ 时间内完成。从概念上讲，将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序，组成最终有序序列。

分治操作是归并排序中最核心的算法，不需要算术或者数据移动，只需要考虑每个元素的输入数组是否一个排序的子数组；所有的操作都是需要将两个已经有序的子数组合并成一个大的有序数组，其排序的过程如图10.9所示。

```
width = 1, A[] = {3,7,6,4,5,8,2,1}
width = 2, A[] = {3,7,4,6,5,8,1,2}
width = 4, A[] = {3,4,7,6,1,2,5,8}
width = 8, A[] = {1,2,3,4,5,6,7,8}
```

图10.9：归并排序的操作示例。在顶部，每一个元素的初始状态是长度为1的排序子数组；操作示例中每一步操作，都是围绕将两个已经有序的子数组合并成一个大的有序数组，组成最终的有序序列。

将两个排序数组组合成一个较大的排序数组的过程通常称为two-finger算法。图10.10展示了两个已经排序的输入数组in1和in2合并到一个大的输出数组out中。two-finger操作初始化时指向每个数组第一个元素（数值in1中的元素3和数组in2中的元素1），在算法执行的过程中，两个指针会指向数组中的不同元素。

$\text{in1}[] = \{\underline{3}, 4, 6, 7\}$	$\text{in2}[] = \{\underline{1}, 2, 5, 8\}$	$\text{out}[] = \{ \}$
$\text{in1}[] = \{\underline{3}, 4, 6, 7\}$	$\text{in2}[] = \{\underline{\blacksquare}, \underline{2}, 5, 8\}$	$\text{out}[] = \{1\}$
$\text{in1}[] = \{\underline{3}, 4, 6, 7\}$	$\text{in2}[] = \{\blacksquare, \underline{\blacksquare}, \underline{5}, 8\}$	$\text{out}[] = \{1, 2\}$
$\text{in1}[] = \{\blacksquare, \underline{4}, 6, 7\}$	$\text{in2}[] = \{\blacksquare, \underline{\blacksquare}, \underline{5}, 8\}$	$\text{out}[] = \{1, 2, 3\}$
$\text{in1}[] = \{\blacksquare, \blacksquare, \underline{6}, 7\}$	$\text{in2}[] = \{\blacksquare, \blacksquare, \underline{5}, 8\}$	$\text{out}[] = \{1, 2, 3, 4\}$
$\text{in1}[] = \{\blacksquare, \blacksquare, \underline{6}, 7\}$	$\text{in2}[] = \{\blacksquare, \blacksquare, \blacksquare, \underline{8}\}$	$\text{out}[] = \{1, 2, 3, 4, 5\}$
$\text{in1}[] = \{\blacksquare, \blacksquare, \blacksquare, \underline{7}\}$	$\text{in2}[] = \{\blacksquare, \blacksquare, \blacksquare, \underline{8}\}$	$\text{out}[] = \{1, 2, 3, 4, 5, 6\}$
$\text{in1}[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$\text{in2}[] = \{\blacksquare, \blacksquare, \blacksquare, \underline{8}\}$	$\text{out}[] = \{1, 2, 3, 4, 5, 6, 7\}$
$\text{in1}[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$\text{in2}[] = \{\blacksquare, \blacksquare, \blacksquare, \blacksquare\}$	$\text{out}[] = \{1, 2, 3, 4, 5, 6, 7, 8\}$

图10.10：合并两个有序数组的操作示例。顶部是初始状态，算法的每一步都需要考虑下划线的元素，并将其中的一个元素排序在输出有序数组中。

图10.10的第一步是初始化状态，输入的两个数组各有四个元素，输出数组是空的。第二步，比较两个数组中的第一个元素，并且把最小的元素写入输出数组中（即比较3和1中小元素1并排序到数组out中）；此时数组in2的指针指向下一个元素。第三步，继续比较两个数组中当前指针所指向元素的大小，并把小的元素排序到输出数组中（即比较3和2中的小元素2并排序到数组out中）。继续类似的操作，直到其中一个数组元素移空；最后只需要拷贝未处理完的数组中剩下的元素到输出数组out中。

归并排序算法是递归函数调用的典型示例。大多数的高层综合语言不支持递归，或者以有边界的方式支持递归。这里我们关注归并排序算法的非递归实现方式，代码看起来与通常CPU架构下的完全不同，但是算法的核心思想是完全相同的。

10.3.1 归并排序的基本操作

图10.11展示了非递归方式实现的归并排序的基本代码，处理数组排序大概需要 $N \log N$ 次比较，并且需要额外的临时存储空间。代码里首先考虑数组中的每个元素作为一个排序子数组，外部循环的每次迭代都将排序子数组合并成较大的排序子数组。第一次迭代，对最大SIZE为2的子数组进行排序；第二次迭代，对最大SIZE为4的子数组进行排序；然后是8，以此类推。需要注意的是，这里的输入数组的长度并不是2的幂次方，也可能有一些子数组小于最大尺寸。

```

#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2-1], subarray2 is in[i2..i3-1], result is in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
        // Select the smallest available element.
        if((f1 < i2 && in[f1] <= in[f2]) || f2 == i3) {
            out[index] = in[f1];
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = in[f2];
            f2++;
        }
    }
}

void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    // Each time through the loop, we try to merge sorted subarrays of width elements
    // into a sorted subarray of 2*width elements.
    stage:
    for (int width = 1; width < SIZE; width = 2 * width) {
        merge_arrays:
        for (int i1 = 0; i1 < SIZE; i1 = i1 + 2 * width) {
            // Try to merge two sorted subarrays:
            // A[i1..i1+width-1] and A[i1+width..i1+2*width-1] to temp[i1..2*width-1]
            int i2 = i1 + width;
            int i3 = i1 + 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge(A, i1, i2, i3, temp);
        }

        // Copy temp[] back to A[] for next iteration
        copy:
        for(int i = 0; i < SIZE; i++) {
            A[i] = temp[i];
        }
    }
}

```

图10.11：非递归方式的归并排序实现。*merge_sort*函数每一步都将两个已经有序的子数组合并成一个大的有序数组，直到整个数组有序

归并排序算法的主入口函数为*merge_sort*，函数的数组为A，需要内部存储存放数组temp；参数SIZE决定两个数组的大小，参数DTYPE决定数据的排序类型。*merge_sort*函数主要有两个嵌套的for循环构成，外部循环体主要跟踪每个排序子数组中的元素数量；初始化时将每个元素作为单独的子数组，因此width宽度初始化为1。每次循环都会生成较大的排序子数组，这些新的子数组绝大多数情况下都是两倍的元素。但宽度大于或者等于参数SIZE的值时，循环终止，此时所有的元素都在一个有序数组中。

内部循环体merge_arrays的主要功能时归并两个有序数组，这些有序数组最多有width个元素，从索引i1和i2开始。然后调用merge函数将有序子数组合并并复制到内部存储temp数组中。这里需要处理循环末尾的边界情况，若参数SIZE不是2的幂次方，子数组可能包含小于宽度的元素。在完成子数组的归并到temp数组后，需要把当前temp数组中的元素按顺序存放在数组A中，以便开展下一轮的循环迭代。

通过不断调整SIZE的值，获取那些值比较合适？当调用merge函数时，i1、i2和i3之间的可能关系有哪些？如果限定参数SIZE，HLS产生的电路又有什么影响？

merge函数对数组in中的两个子数组采用了two-finger算法。数组in作为函数的输入数组，数组out作为函数的输出数组；而且还需要输入变量i1、i2和i3，来描述需要合并的两个子数组的边界范围。一个子数组从第i1坐标索引开始，到第i2个坐标索引之前截止；还有一个子数组从第i2个坐标索引开始，到第i3个坐标索引截止；合并后输出子数组将存储在数组out的坐标索引i1到i3之间。

merge函数包含一个遍历数组排序输出的循环；每一个循环迭代都将一个元素放在输出数组out中的正确排序的位置。函数中的变量f1和f2对应每个子数组的坐标索引。if条件里面的主要功能即是选择in[f1]和in[f2]中相对小的元素，并拷贝到输出数组out下一个排序的位置中。而if条件里面还需要处理几种特殊的情况；一种情况是f1等于i2，此时in[f1]已经超过子数组的边界，需要在in[f2]里面选择最小的元素；同样地。如果f2等于i3，此时in[f2]已经超过子数组的边界，需要在in[f1]里面选择最小的元素。

在处理过程中数组in会发生什么变化？在外部循环的每次迭代之后数组in的元素排序会是什么？当merge_sort函数返回时，数组in的元素排序会是什么？

综合后的性能报告可能无法确定归并排序的延迟和间隔，为什么会出现这种情况？如何评估loop tripcount指令值，哪一个才是合适的最小值、最大值和平均值？

图10.11中的代码并没有使用Vivado HLS的一些特定优化指令。对多层嵌套for循环的优化，通常先从内部循环优化开始，然后再进行外部循环的优化。常用的循环优化指令有流水线指令pipeline和展开指令unroll。

使用pipeline指令设置不同的优化参数，还可以使用unroll指令展开for循环，哪一种方式提供了最好性能？性能和资源利用率之间的最佳权衡又是什么？代码方面的设计缺陷阻碍了更高的性能？这些方面都是算法设计考虑的因素，不能仅关注代码实现层面。

pipeline和unroll往往会受到资源上面的限制，因此需要考虑数组输入输出的端口数量。图10.11中的数组都是一维的，此时设计者还需要仔细考虑数组的访问模式，以确保性能优化与资源限制匹配。

建议使用array_partition、pipeline和unroll指令来优化循环，探索出最佳的策略。只考虑性能方面，最佳的设计是什么？在考虑资源利用率和性能之间的权衡后，最佳的设计是什么？

最佳设计往往只有通过重构代码才能实现。虽然Vivado HLS提供了许多指令来启动一个常规的公共代码优化，但是为每一个优化提供一个指令的做法是不切实际的。在下一节中，将描述一种方法重构归并排序的代码，以增加排序的吞吐量。

10.3.2 重构归并排序

图10.11实现的内部循环merge函数要取得1个时钟周期的循环启动间隔是非常有挑战的。一方面数组in有四次读操作，且有两个不同的读操作地址。HLS综合工具需要分析出其中有些读取操作是冗余的，但是FPGA的片上BRAM资源只能支持每个时钟周期的两个访问，再加上这些读操作在不同的基本块中，编译器更难消除冗余负载。通过重构代码来消除冗余读取，编译器只需要更少的优化来实现1个时钟周期的循环启动间隔，重构后的代码如图10.12所示。另外一方面，在变量f1和f2有递归关系，这些变量在if条件内部递增，并且确定了下一次循环时数组in中的需要比较元素的位置信息。这种动态浮动的比较方式也会影响时钟频率和循环的启动间隔。

```
#include "merge_sort.h"
#include "assert.h"

// subarray1 is in[i1..i2-1]; subarray2 is in[i2..i3-1]
// sorted merge is stored in out[i1..i3-1]
void merge(DTYPE in[SIZE], int i1, int i2, int i3, DTYPE out[SIZE]) {
    int f1 = i1, f2 = i2;
    // Foreach element that needs to be sorted...
    for(int index = i1; index < i3; index++) {
#pragma HLS pipeline II=1
        DTYPE t1 = in[f1];
        DTYPE t2 = in[f2];
        // Select the smallest available element.
        if((f1 < i2 && t1 <= t2) || f2 == i3) {
            out[index] = t1;
            f1++;
        } else {
            assert(f2 < i3);
            out[index] = t2;
            f2++;
        }
    }
}
```

图10.12：重构merge函数的代码实现：在Vivado HLS中循环的启动间隔可以达到1

图10.13描述了重构代码的行为结构，虽然内部循环实现了1个时钟周期的启动间隔，但是在内部循环结束时，在下一个阶段的流水线执行之前，中间的缓冲数据必须先清空。一旦循环的次数增加，中间的气泡问题还是影响流水线性能的关键因素。由于循环静态分析的局限性，影响性能的代码部分很难可视化，而且内部循环的次数是有变量参数决定的。

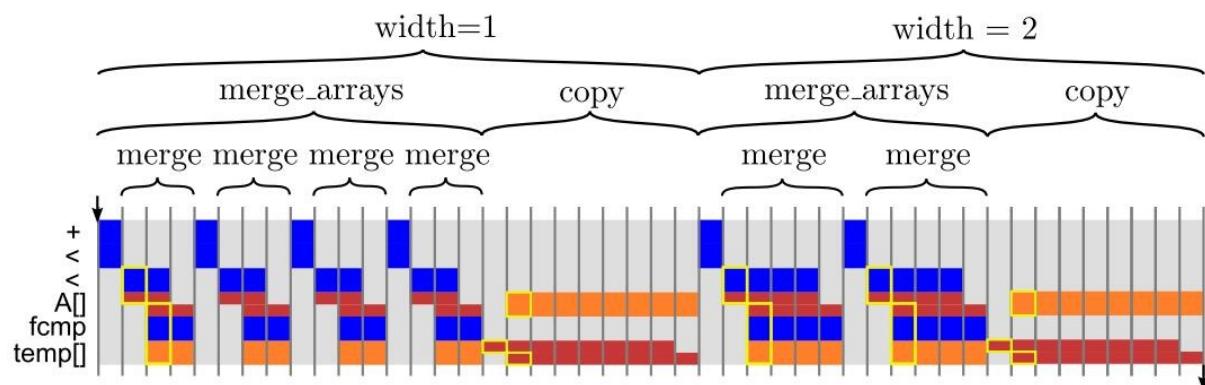


图10.13：图10.12中重构代码的行为结构

常见的方法是将多层循环嵌套优化成一个单一循环，减少流水线中循环退出时的刷新次数，Vivado HLS工具也支持了多层次循环嵌套的自动优化。然而，图10.11中的代码实现并不是一个完美的循环嵌套，需要重构merge函数以支持循环嵌套的完美合并，重构后的代码如图10.14所示。其最大的变化就是merge_arrays函数循环次数也是确定的，使得编译器更好地理解实现。

```

#include "merge_sort.h"
#include "assert.h"

void merge_sort(DTYPE A[SIZE]) {
    DTYPE temp[SIZE];
    stage:
        for (int width = 1; width < SIZE; width = 2 * width) {
            int f1 = 0;
            int f2 = width;
            int i2 = width;
            int i3 = 2*width;
            if(i2 >= SIZE) i2 = SIZE;
            if(i3 >= SIZE) i3 = SIZE;
            merge_arrays:
                for (int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
                    DTYPE t1 = A[f1];
                    DTYPE t2 = A[f2];
                    if((f1 < i2 && t1 <= t2) || f2 == i3) {
                        temp[i] = t1;
                        f1++;
                    } else {
                        assert(f2 < i3);
                        temp[i] = t2;
                        f2++;
                    }
                    if(f1 == i2 && f2 == i3) {
                        f1 = i3;
                        i2 += 2*width;
                        i3 += 2*width;
                        if(i2 >= SIZE) i2 = SIZE;
                        if(i3 >= SIZE) i3 = SIZE;
                        f2 = i2;
                    }
                }
            }

            copy:
                for(int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
                    A[i] = temp[i];
                }
            }
}

```

图10.14：重构merge函数的代码实现：在Vivado HLS中循环的启动间隔可以达到1，而且流水线的气泡很少

评估图10.14中代码的性能，即使内部循环已经实现了1个时钟周期的启动间隔，该设计是否有效合理地利用了硬件加速？有没有进一步改进merge_sort函数延迟的方法，使得整个延迟在大约在 $N \log N$ 个时钟周期？

前面重点介绍了如何优化merge_sort函数，在不大幅增加资源利用率的情况下，降低计算处理的延迟，提高了加速器的计算效率。提高加速器并行计算能力的方法主要是进一步降低延迟或者提高吞吐量。前面介绍的内部循环的展开优化和数组分区优化可以在每一个时钟周期内同时执行更多的任务。还有一种提高并行计算能力的方法是寻找粗粒度的任务级流水线。这里可以把归并排序的内部迭代体merge_arrays创建多份硬件资源，以达到粗粒度的任务级流水。如图10.15所示，通过展开连续的内部循环体merge_arrays，可以同时处理不同的数据集，以提高整个加速器的计算性能。

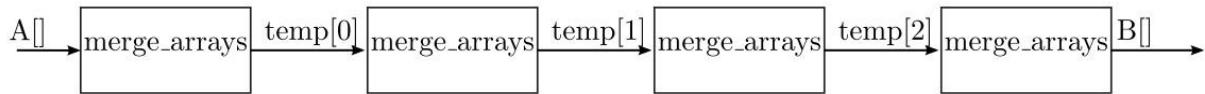


图10.15：归并排序中任务级流水线的体系结构，可以容纳16个元素的排序

图10.16是加入任务级流水和数据流优化后的代码实现。虽然有很多方面都与原始代码类似，但是有几个重要的区别。第一个区别是merge_arrays函数的循环都提取到一个函数中，方便了顶层函数的代码重构；第二个区别是merge_sort_parallel函数的输入和输出分别在一个独立的数组中，通过Vivado HLS的数据流指令就能够构建一个数据流水线的体系结构；第三，temp数组用在merge_arrays函数之间的乒乓操作数据流中，减少中间缓存的不必要的拷贝信息；同时重构为一个二维数组，使得通道数的表示参数化。

```

#include "merge_sort_parallel.h"
#include "assert.h"

void merge_arrays(DTYPE in[SIZE], int width, DTYPE out[SIZE]) {
    int f1 = 0;
    int f2 = width;
    int i2 = width;
    int i3 = 2*width;
    if(i2 >= SIZE) i2 = SIZE;
    if(i3 >= SIZE) i3 = SIZE;
    merge_arrays:
        for (int i = 0; i < SIZE; i++) {
#pragma HLS pipeline II=1
            DTYPE t1 = in[f1];
            DTYPE t2 = in[f2];
            if((f1 < i2 && t1 <= t2) || f2 == i3) {
                out[i] = t1;
                f1++;
            } else {
                assert(f2 < i3);
                out[i] = t2;
                f2++;
            }
            if(f1 == i2 && f2 == i3) {
                f1 = i3;
                i2 += 2*width;
                i3 += 2*width;
                if(i2 >= SIZE) i2 = SIZE;
                if(i3 >= SIZE) i3 = SIZE;
                f2 = i2;
            }
        }
    }

void merge_sort_parallel(DTYPE A[SIZE], DTYPE B[SIZE]) {
#pragma HLS dataflow

    DTYPE temp[STAGES-1][SIZE];
#pragma HLS array_partition variable=temp complete dim=1
    int width = 1;

    merge_arrays(A, width, temp[0]);
    width *= 2;

    for (int stage = 1; stage < STAGES-1; stage++) {
#pragma HLS unroll
        merge_arrays(temp[stage-1], width, temp[stage]);
        width *= 2;
    }

    merge_arrays(temp[STAGES-2], width, B);
}

```

图10.16：重构归并排序的代码实现：利用了Vivado HLS的dataflow和pipeline的指令优化

merge_sort_parallel函数里面包含多个阶段的merge_arrays函数调用；第一次调用时从输入端读取数据，并把处理完的结果写入temp数组中；在循环执行过程中，多次调用merge_arrays，并写入temp数组的其他分区；最后一个调用时写入排序后的结果到数组B中。在资源足够的情况下，参数SIZE和STAGES可以支持更大的吞吐。

评估图10.16中重构后的代码性能，分析其实现的延迟和启动间隔，大概需要多大的片上存储？

10.4 总结

本章介绍了一些基本的排序算法。插入排序操作平均需要 $N^2/4$ 次数据比较，极端下需要 N^2 次比较，但是仅需要少量的存储资源；展示了插入排序中的几种提升性能的不同方式；由于流水线的气泡问题， N 个比较器大约需要 N 个时钟周期的间隔才能完成插入排序。归并排序的比较次数相对插入排序较少，大约仅需要 $N \log N$ 次比较，但是需要额外的片上存储开销存储中间缓存。一种高效的设计方案是通过重构优化代码，取得在 1 个比较器的情况下，归并排序大概仅需 $N \log N$ 个时钟周期即可完成。另外一种高效的设计方案是采用任务级流水线，每一个时钟周期有 $\log N$ 个比较， N 个时钟周期的间隔才能完成归并排序。与插入排序相比，同样的计算处理周期，归并排序需要较少的比较器，但是带来的代价是额外的存储资源开销。归并排序如果需要更多的片上存储，其整体的延迟也会变大。

在实际系统中，基于FPGA的排序设计方案都必须解决资源和时间性能之间的权衡问题。还有一种使用不同折中方案的排序算法叫基数排序，与本章的排序算法不太一样，其更关注数据的有效范围作为元素之间的比较项。基数排序的一部分实现在下一章的Huffman编码会有更详细的介绍。

排序算法的并行结构通常被描述为排序网络，排序网络有时可以是脉动阵列或者流水线。通过研究这些现有的并行结构来获得HLS的设计灵感，然后通过C代码来实现它。在Vivado HLS中，可以使用任意一种方法来描述这些循环流水线、数据流水线或者两者的组合。

第十一章 霍夫曼编码

11.1 背景

无损数据压缩是高效数据存储中的一个关键要素，而霍夫曼编码则是其中最流行的可变长度编码算法[33]。给定一组数据符号以及它们出现的频率，霍夫曼编码将以更短的代码分配给更频繁（出现的）符号这种方式生成码字来最小化平均代码长度。由于它保证了最优性，霍夫曼编码已被各种应用广泛采用[25]。在现代多级压缩设计中，它经常用作系统的后端，来提高特定领域前端的压缩性能，如GZIP[23]，JPEG[57]和MP3[59]。尽管算术编码[61]（霍夫曼编码的一个广义版本，它将整个消息转换为单个数字）可以对大多数场景实现更好的压缩，但是由于算术编码专利问题[38]，霍夫曼编码通常成为许多系统的首选算法。

Canonical霍夫曼编码与传统的霍夫曼编码相比有两大优势。在基本霍夫曼编码中，编码器将完整霍夫曼树结构传递给解码器。因此，解码器必须遍历树来解码每个编码的符号。另一方面，Canonical霍夫曼编码仅将每个符号的位数传送给解码器，然后解码器重构每个符号的码字。这使得解码器在内存使用和计算需求中更加高效。因此，我们专注于Canonical霍夫曼编码。

在基本的霍夫曼编码中，解码器通过遍历霍夫曼树，从根开始直到它到达叶节点来解压缩数据。这种方式有两个主要缺点：第一、它需要存储整个霍夫曼树，从而增加了内存使用量。此外，为每个符号遍历树的计算成本是很高的。Canonical霍夫曼编码通过使用标准的规范格式创建代码来解决这两个问题。使用Canonical编码的好处在于：我们只需要传输每个霍夫曼码字的长度。一个Canonical霍夫曼代码具有两个额外的属性：首先，较长代码比相同长度前缀的较短代码具有更高的数值。其次，具有相同长度的代码随着符号值的增加而增加。这意味着如果我们知道每个代码长度的起始符号，我们可以很容易重建Canonical霍夫曼编码。霍夫曼树本质上等同于“排序”版本的原始霍夫曼树，以便更长的码字在树的最右边的分支上，树的同一级别上的所有节点按照符号的顺序排序。

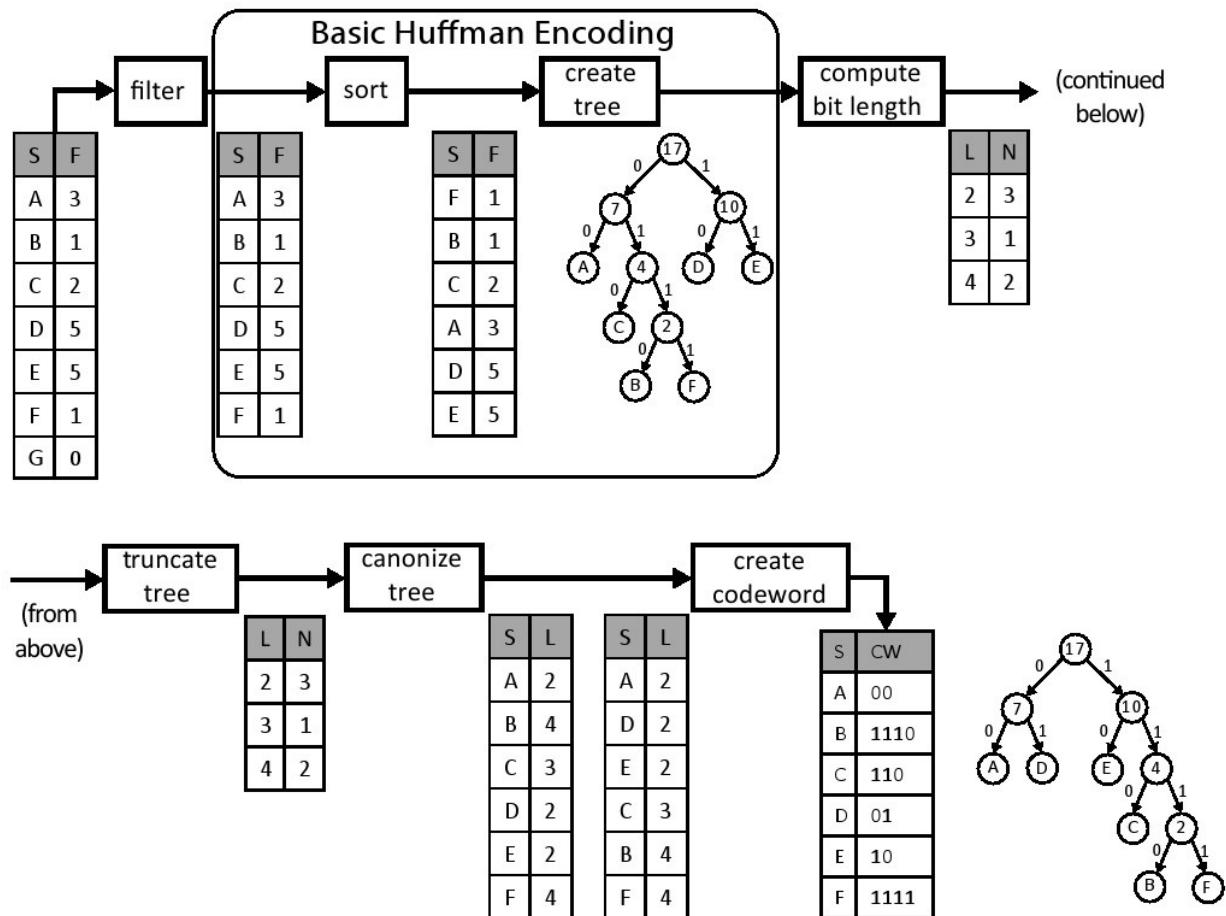


图11.1：Canonical霍夫曼编码过程。符号被过滤、排序，并且被用来建造霍夫曼树，而不是将整个树传递给解码器（就像“基本的”霍夫曼编码所完成的那样），这样编码使得只有树的符号长度是解码器所需要的。请注意，最终Canonical树与初始树在流程开始创建时是不同的。

图11.1展示了创建Canonical霍夫曼编码的过程。filter模块仅传递非零频率的符号。sort模块根据它们的频率按升序重新排列符号。接下来，create tree模块使用以下三个步骤构建霍夫曼树：

1. 使用两个最小频率节点作为初始子树并通过求它们频率的和来生成新的父节点；
2. 它将新的中间节点添加到列表中，并再次分类；
3. 它从列表中选择两个最小元素并重复这些步骤直到剩下最后一个元素为止。

结果是霍夫曼树中的每个叶节点代表可以被编码的符号，并且每个内部节点被标记为孩子树节点的频率。通过将树中的左边和右边与位0和1相关联，我们可以根据从根节点到达它的路径来确定每个符号唯一的码字。例如，A的码字是00，而B的码字是1110。这样就完成了基本的霍夫曼编码过程，但是对于创建Canonical霍夫曼树这并不是必须的。

为了创建霍夫曼树，我们做了几个额外的转换。首先，compute bit len模块计算每个码字的位长度，然后对每个长度的频率进行计数，结果是一个码字长度的直方图（见第8.2节）。在这个例子中，我们有三个符号（A, D, E），代码长度为2。因此，计算的直方图表中包含了位置2中的数值3。接下来，truncate tree模块重新平衡霍夫曼树以避免出现过长的码字。这可以在编码时间稍微增加的代价下提高解码器的速度。这在图11.1的示例中不是必需的。我们设定了树的最大高度为27。最后，canonize tree模块创建两个排序表，第一个表格包含了符号和按符号排序的符号长度。第二个表格包含了符号和按长度排序的符号长度。这些表格简化了为每个符号创建Canonical霍夫曼码字的过程。

create codeword模块通过遍历已排序的表创建Canonical霍夫曼码字表。从排序表中的第一个码字开始，它被分配了适当长度全零码字。每个具有相同位长度的后续符号被赋值为随后的码字，它是在前一个码字上简单地加1而形成的。在我们的例子中，符号A，D和E的位长度均是 $l = 2$ ，而分配的码字分别是A = 00，D = 01，E = 10。请注意符号是按字母顺序考虑的，这是使树规范化所必需的。这个过程一直持续直到我们得到一个需要一个更大长度的码字，在这种情况下，我们不仅增加了先前的码字，而且还向左移位以生成正确长度的码字。在这个例子中，下一个符号是C，长度为3，其接收的码字C = $(10 + 1) \ll 1 = 11 \ll 1 = 110$ 。接下来，下一个符号是B，长度为4。再一次，我们增加和移位一次。于是B的代码字为B = $(110 + 1) \ll 1 = 1110$ 。符号F的最终码字为F = 1110 + 1 = 1111。详见第11.2.7章。

Canonical霍夫曼代码的创建包括许多复杂的和内在的顺序计算。例如，create_tree模块需要跟踪被创建的子树的正确顺序、需要仔细的内存管理，可以被利用的并行性非常有限。下面我们来讨论使用Vivado HLS进行Canonical霍夫曼编码设计的硬件架构和实现。

```
#include "huffman.h"
#include "assert.h"
void huffman_encoding(
    /* input */ Symbol symbol_histogram[INPUT_SYMBOL_SIZE],
    /* output */ PackedCodewordAndLength encoding[INPUT_SYMBOL_SIZE],
    /* output */ int *num_nonzero_symbols) {
#pragma HLS DATAFLOW

    Symbol filtered[INPUT_SYMBOL_SIZE];
    Symbol sorted[INPUT_SYMBOL_SIZE];
    Symbol sorted_copy1[INPUT_SYMBOL_SIZE];
    Symbol sorted_copy2[INPUT_SYMBOL_SIZE];
    ap_uint<SYMBOL_BITS> parent[INPUT_SYMBOL_SIZE-1];
    ap_uint<SYMBOL_BITS> left[INPUT_SYMBOL_SIZE-1];
    ap_uint<SYMBOL_BITS> right[INPUT_SYMBOL_SIZE-1];
    int n;

    filter(symbol_histogram, filtered, &n);
    sort(filtered, n, sorted);

    ap_uint<SYMBOL_BITS> length_histogram[TREE_DEPTH];
    ap_uint<SYMBOL_BITS> truncated_length_histogram1[TREE_DEPTH];
    ap_uint<SYMBOL_BITS> truncated_length_histogram2[TREE_DEPTH];
    CodewordLength symbol_bits[INPUT_SYMBOL_SIZE];

    int previous_frequency = -1;
    copy_sorted:
    for(int i = 0; i < n; i++) {
        sorted_copy1[i].value = sorted[i].value;
        sorted_copy1[i].frequency = sorted[i].frequency;
        sorted_copy2[i].value = sorted[i].value;
        sorted_copy2[i].frequency = sorted[i].frequency;
        // std::cout << sorted[i].value << " " << sorted[i].frequency << "\n";
        assert(previous_frequency <= (int)sorted[i].frequency);
        previous_frequency = sorted[i].frequency;
    }

    create_tree(sorted_copy1, n, parent, left, right);
    compute_bit_length(parent, left, right, n, length_histogram);

#ifndef __SYNTHESIS__
    // Check the result of computing the tree histogram
    int codewords_in_tree = 0;
    merge_bit_length:
    for(int i = 0; i < TREE_DEPTH; i++) {
```

```

#pragma HLS PIPELINE II=1
if(length_histogram[i] > 0)
    std::cout << length_histogram[i] << " codewords with length " << i << "\n";
codewords_in_tree += length_histogram[i];
}
assert(codewords_in_tree == n);
#endif

truncate_tree(length_histogram, truncated_length_histogram1, truncated_length_histogram
2);
canonize_tree(sorted_copy2, n, truncated_length_histogram1, symbol_bits);
create_codeword(symbol_bits, truncated_length_histogram2, encoding);

* num_nonzero_symbols = n;
}

```

图11.2: 显示了整个“top” *Huffman_encoding*函数，它设置了数组和其他在各个子函数之间传递的变量，并实例化了这些函数。

有些额外的数据复制可能是不必要的，这是因为我们使用了dataflow指令，这给子函数间变量的传递起到了限制作用。特别是，部分函数之间，生产者和消费者的数据关系，有一些严格的规则，这就要求我们复制一些数据。例如，我们创建了数组parent，Left和Right的两个副本。我们也与数组截短位长度相同。前者是在top霍夫曼编码函数的for循环中完成的；后者是在canonize tree函数内完成的。

dataflow指令限制函数中的信息流动。许多限制强化了子函数之间严格的生产者和消费者关系。一个这样的限制是，一个数组应该只由一个函数写入，并且它只能由一个函数读取。即它只能作为从一个函数的输出到另一个函数的输入。如果多个函数从同一数组中读取，Vivado R.HLS将综合代码，但会发出警告并且不会使用数据流水线式架构。因此，使用数据流模式通常需要将数据复制到多个数组中。如果一个函数试图从一个数组读写，而这个数组同时被另一个函数访问，则会出现类似的问题。在这种情况下，必须保留函数内部的数据一个额外的内部副本。我们将讨论这两个要求以及如何遵守它们，正如我们在本章其余部分所述的那样。

11.2 实现

Canonical霍夫曼编码过程本质上是被分成子函数。因此，我们可以逐一处理这些子函数。在我们这样做之前，我们应该考虑这些函数的每个接口。图11.4显示了函数及其输入和输出的数据。为了简单起见，它只显示与数组的接口，由于它们很大，我们可以假设它们存储在RAM块中（BRAMs）。在我们描述这些函数以及他们的输入和输出之前，我们需要讨论huffman.h中定义的常量，自定义数据类型和函数接口。图11.3显示了这个文件的内容。

INPUT_SYMBOL_SIZE参数定义了作为编码输入的符号的最大数量。在这种情况下，我们将其设置为256，从而启用8位ASCII数据的编码。TREE_DEPTH参数指定初始霍夫曼树生成期间单个代码字长度的上限。当霍夫曼树在truncate_tree函数中重新平衡时，CODEWORD_LENGTH参数指定了目标树的高度。CODEWORD_LENGTH_BITS常量决定编码码字长度所需的位数，等于 $\log_2[\text{CODEWORD_LENGTH}]$ ，在这种情况下也即等于5。

```

#include "ap_int.h"

// input number of symbols
const static int INPUT_SYMBOL_SIZE = 256;

// upper bound on codeword length during tree construction
const static int TREE_DEPTH = 64;

// maximum codeword tree length after rebalancing
const static int MAX_CODEWORD_LENGTH = 27;

// Should be log2(INPUT_SYMBOL_SIZE)
const static int SYMBOL_BITS = 10;

// Should be log2(TREE_DEPTH)
const static int TREE_DEPTH_BITS = 6;

// number of bits needed to record MAX_CODEWORD_LENGTH value
// Should be log2(MAX_CODEWORD_LENGTH)
const static int CODEWORD_LENGTH_BITS = 5;

// A marker for internal nodes
const static ap_uint<SYMBOL_BITS> INTERNAL_NODE = -1;

typedef ap_uint<MAX_CODEWORD_LENGTH> Codeword;
typedef ap_uint<MAX_CODEWORD_LENGTH + CODEWORD_LENGTH_BITS> PackedCodewordAndLength;
typedef ap_uint<CODEWORD_LENGTH_BITS> CodewordLength;
typedef ap_uint<32> Frequency;

struct Symbol {
    ap_uint<SYMBOL_BITS> value;
    ap_uint<32> frequency;
};

void huffman_encoding (
    Symbol in[INPUT_SYMBOL_SIZE],
    PackedCodewordAndLength encoding[INPUT_SYMBOL_SIZE],
    int *num_nonzero_symbols
);

```

如图11.3：顶层函数huffman_encoding的参数、自定义数据类型和函数接口。

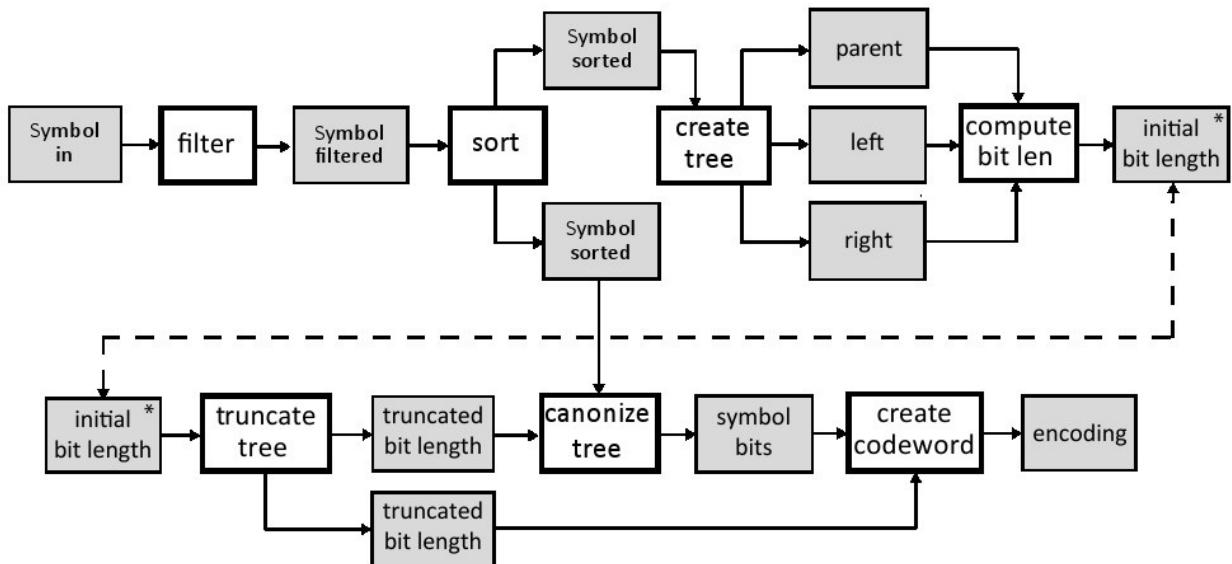


图11.4：Canonical霍夫曼编码的硬件实现框图。灰色块表示由不同子函数生成和消耗的重要输入和输出数据。白色块对应于函数（计算核心）。注意，数组初始位长度出现两次，以使图形更加清晰。

我们创建一个自定义数据类型Symbol来保存对应于输入值和它们的频率的数据。该数据类型在需要访问这些信息的编码过程中被用于filter, sort , 以及其它一些函数。数据类型有两个字段：数值和频率。在这里我们假定被编码的数据块包含不超过 2^{32} 个符号。最后，huffman.h文件具有huffman_encoding函数接口。这是Vivado HLS工具指定的顶层函数。它有三个参数, 第一个参数是大小为INPUT SYMBOL SIZE的符号数组。该数组表示被编码块中数据频率的直方图。接下来的两个参数是输出。encoding参数输出每个可能符号的码字。Num_nonzero_symbols参数是来自输入数据的非零符号的数量，这与filter操作后剩余的符号数量相同。系统的输入是一个Symbol数组，这保存了数组IN中的符号值和频率。每个符号保存10-bit value和32-bit frequency。该数组的大小设置为常数INPUT_SYMBOL_SIZE，在本例中为256。filter模块从in数组读取，并将其输出写入到filtered数组。这是一个Symbols数组，它保存了sort模块输入的非零元素个数。sort模块将按频率排序的符号写入两个不同的数组中 - 一个用于create tree模块，另一个用于canonize tree模块。create tree模块从排序后的数组中创建一个霍夫曼树并将其存储到三个数组中 ((parent, left, and right)) ;这些数组拥有霍夫曼树每个节点的所有信息。使用霍夫曼树信息，compute bit len模块计算每个符号的位长并将该信息存储到initial bit len数组。我们将最大条目数设置为64，覆盖最高64位的频率数，这对于大多数应用来说是足够的，因为我们的霍夫曼树能够重新平衡它的高度。truncate tree模块重新平衡树高，并将每个码字的位长度信息复制到两个单独的truncated bit length数组中。它们每个都有完全相同的信息，但它们必须被复制以确保Vivado HLS工具可以执行流水线功能;稍后我们将会更详细地讨论它。canonize tree模块遍历sort模块中的每个符号，并使用truncated bit length数组分配适当的位长度。canonize模块的输出是一个数组，其中包含每个符号码字的位长度。最后，create codeword模块为每个符号生成canonical码字。

```

#include "huffman.h"
// Postcondition: out[x].frequency > 0
void filter(
    /* input */ Symbol in[INPUT_SYMBOL_SIZE],
    /* output */ Symbol out[INPUT_SYMBOL_SIZE],
    /* output */ int *n) {
#pragma HLS INLINE off
    ap_uint<SYMBOL_BITS> j = 0;
    for(int i = 0; i < INPUT_SYMBOL_SIZE; i++) {
#pragma HLS pipeline II=1
        if(in[i].frequency != 0) {
            out[j].frequency = in[i].frequency;
            out[j].value = in[i].value;
            j++;
        }
    }
    *n = j;
}

```

图11.5：filter函数在输入数组in中迭代，并将具有非零频率字段的任何符号条目添加到输出数组out。此外，它记录非零频率元素的数量并将其传递到输出参数n。

11.2.1 过滤

霍夫曼编码过程的第一个函数是filter，如图11.5所示。这个函数的输入是Symbol数组。输出是另一个Symbol数组，它是输入数组in的一个子集。Filter函数删除任何频率等于0的条目。函数本身只是简单的迭代in数组，如果其frequency域不为零，则将每个元素存储到out数组中。另外，该函数计算输出的非零条目数，作为输出参数n传递，使其他函数只能处理“有用的”数据。

Vivado HLS可以自动内联函数以便生成更高效的架构。大多数情况下，这发生在小函数上。inline指令允许用户明确指定Vivado HLS是否应该内联特定的功能。在这种情况下，INLINE off 确保该功能不会被内联，并且会在生成寄存器传输级（RTL）设计中作为模块出现。在这种情况下，禁用内联可以使我们能够获得该函数的性能和资源使用情况，并确保它将作为顶层数据流设计中的一个流程来实施。

11.2.2 分类

如图11.6所示，sort函数根据对输入符号的frequency值进行排序。该函数由两个for循环组成，标记为copyin_to_sorting和radix_sort。copy_in_to_sorting循环把输入的数据从in数组搬到sorting数组中，这确保了in数组是只读的，以满足在顶层使用的dataflow指令的要求。sorting函数在整个执行过程中读取和写入sorting数组。即使对于这样的简单循环，使用pipeline指令生成最有效的结果和最准确的性能估计是很重要的。radix sort循环实现核心基数排序算法。通常，基数排序算法通过一次处理一个数字或一组比特来对数据进行排序。每个数字的大小决定了排序的基数。我们的算法在32比特的Symbol.frequency变量时处理4比特。因此我们使用基数 $r = 2^4 = 16$ 排序。对于32位数字中的每个4比特数字，我们进行计数排序。radix_sort循环执行这8个计数排序操作，以4为单位迭代到32。基数排序算法也可以从左到右（首先是最低有效位）或从右到左（首先是最高有效位）操作。算法从最低有效位运行到最高有效位。在代码中，基数可以通过设置RADIX和BITS_PER_LOOP参数来配置。

如果我们增加或减少基数会发生什么？这将如何影响执行的计数排序操作的次数？这将如何改变资源使用，例如数组的大小呢？

代码在sorting[]和previous_sorting[]中存储当前排序的状态。每次迭代radix_sort_loop循环，sorting[]的当前值被复制到previous_sorting[]，然后这些值在被复制回sorting[]时被排序.digit_histogram[]和digit_location[]数组用于radix_sort_loop来实现对特定数字的计数排序。两个array_partition数组声明这两个数组应该完全映射到寄存器中。这些数组很小而且使用频繁，因此不占用很多资源就可以提供良好的性能。最后，current_digit[]存储了基数排序的当前迭代中为每个项目排序的数字。该代码还包含两个assert () 调用，用于检查有关num_symbols符号输入。由于该变量决定了in数组中有效元素的数量，因此它必须以数组的大小为界。这样的断言通常是良好的防御性编程实践，以确保满足该函数的假设。在Vivado HLS中他们也有一个额外的目的。由于num_symbols符号决定了许多内部循环执行的次数，Vivado HLS可以基于这些断言推断循环的计数。另外，Vivado HLS还使用这些断言在电路实现时最小化变量的位宽。

```

#include "huffman.h"
#include "assert.h"
const unsigned int RADIX = 16;
const unsigned int BITS_PER_LOOP = 4; // should be log2(RADIX)
typedef ap_uint<BITS_PER_LOOP> Digit;

void sort(
    /* input */ Symbol in[INPUT_SYMBOL_SIZE],
    /* input */ int num_symbols,
    /* output */ Symbol out[INPUT_SYMBOL_SIZE]) {
    Symbol previous_sorting[INPUT_SYMBOL_SIZE], sorting[INPUT_SYMBOL_SIZE];
    ap_uint<SYMBOL_BITS> digit_histogram[RADIX], digit_location[RADIX];
#pragma HLS ARRAY_PARTITION variable=digit_location complete dim=1
#pragma HLS ARRAY_PARTITION variable=digit_histogram complete dim=1
    Digit current_digit[INPUT_SYMBOL_SIZE];

    assert(num_symbols >= 0);
    assert(num_symbols <= INPUT_SYMBOL_SIZE);
copy_in_to_sorting:
    for(int j = 0; j < num_symbols; j++) {
#pragma HLS PIPELINE II=1
        sorting[j] = in[j];
    }

radix_sort:
    for(int shift = 0; shift < 32; shift += BITS_PER_LOOP) {
init_histogram:
    for(int i = 0; i < RADIX; i++) {
#pragma HLS pipeline II=1
        digit_histogram[i] = 0;
    }

compute_histogram:
    for(int j = 0; j < num_symbols; j++) {
#pragma HLS PIPELINE II=1
        Digit digit = (sorting[j].frequency >> shift) & (RADIX - 1); // Extract a digit
        current_digit[j] = digit; // Store the current digit for each symbol
        digit_histogram[digit]++;
        previous_sorting[j] = sorting[j]; // Save the current sorted order of symbols
    }

    digit_location[0] = 0;
find_digit_location:
    for(int i = 1; i < RADIX; i++)
#pragma HLS PIPELINE II=1
        digit_location[i] = digit_location[i-1] + digit_histogram[i-1];

re_sort:
    for(int j = 0; j < num_symbols; j++) {
#pragma HLS PIPELINE II=1
        Digit digit = current_digit[j];
        sorting[digit_location[digit]] = previous_sorting[j]; // Move symbol to new sorted
location
        out[digit_location[digit]] = previous_sorting[j]; // Also copy to output
        digit_location[digit]++;
    }
}
}
}

```

图11.6：sort函数根据输入符号的频率值对其进行基数排序。

之前我们已经看到loop_tripcount指令向Vivado HLS提供循环次数信息。使用assert（）语句有许多相同的用途，有一些优点和缺点。使用assert（）语句的一个优点是它们检查仿真过程，并且这些信息可以用来进一步优化电路。然而，loop_tripcount指令只影响性能分析，不是用于优化。另一方面，assert（）语句只能用于对变量值进行限制，但不能用于设置期望值或平均值，只能通过loop_tripcount指令完成。在大多数情况下，建议首先通过assert（）语句提供最差情况边界，然后在必要时添加loop_tripcount指令。

radixsort循环的主体被分成四个子循环，分别标记为*init_histogram*，*compute_histogram*，*find_digit_location*，和*re_sort*。*init_histogram*和*compute_histogram*结合基于当前考虑的数字来计算输入的直方图。这会产生每次每个数字在*digit histogram[]*中出现的数量。*compute_histogram*循环还存储当前正在为*current digit[]*中每个符号排序的数字。接下来，*finddigit_location*循环计算所得直方图值的前缀之和，把结果存在*digit location[]*。在计数排序的情况下，*digit_location[]*包含新排序数组中每个数字的第一个符号的位置。最后，*re_sort*循环根据这些结果对符号重新排序，将每个元素放置在新排序数组中的正确位置。它使用*current digit[]*中存储的密钥从*digitlocation[]*中选择正确的位置。每次通过*re_sort*循环时，该位置都会递增，以将下一个具有相同数字的元素放入排序数组中的下一个位置。总体而言，每次都是通过*radix_sort*循环迭代来实现对一个数字的计数排序。计数排序是一个稳定的排序，具有相同数字的元素保持相同的顺序。在基于每个数字的稳定排序之后，数组以正确的最终顺序返回。前面我们在8.2和8.1章中讨论过直方图和前缀求和算法。在这种情况下，我们使用简单的代码和*digit_histogram[]*和*digit_location[]*完整的分解，就可以实现I的循环II来计算直方图和前缀和，因为箱子的数量非常小。*re_sort*循环的优化与此类似。由于唯一的循环是通过相对较小的*digit location[]*数组，实现I的循环II也很简单。注意这种方法的工作原理主要是因为我们配置的RADIX相对较小。较大的RADIX值时，将*digit_histogram[]*和*digit_location[]*作为存储是更好的，这可能需要额外的优化来实现I的循环II。在此代码的上下文中可能有意义的另一种替代方法是将*digixPrase[]*和*digitalPosith[]*完全的分解与*init_histogram*和*find_digit_location*循环完全的展开结合起来。这些循环访问这些小数组中的每个位置并使用最少量的逻辑进行操作。在这种情况下，尽管展开循环可能会导致为每个循环体复制电路，但实现此电路所需的资源较少，因为阵列访问将处于固定索引。但是，改变更大的值的BITS_PER_LOOP参数是禁止的，因为每个额外的位都会使RADIX参数翻倍，展开循环的成本也翻了一番。这对于参数化代码是一种常见的情况，不同的优化对于不同的参数值是有意义的。

在第8.2和第8.1章中提到的前缀求和和直方图循环中，执行优化时，性能和利用率结果会发生什么变化？在这种情况下，优化是必要的吗？

*re_sort*循环是否能够实现一个周期的特定启动间隔？为什么或为什么不？

对于大数据集（n>256），图11.6中代码的近似延时（以n为单位）是多少？代码的哪些部分支配周期数量？这将如何随着RADIX参数的变化而改变？

请注意，`re_sort`循环不仅将排序的数组存储在`sorting[]`中，而且还存储排序的数组在`out[]`。虽然这可能看起来多余，但我们需要确保只写入`out[]`以遵从顶层dataflow指令的要求。在这种情况下，`out[]`将会用部分排序的结果重写多次，但只有最终结果传递给后面的函数。

dataflow指令对于执行任务级别流水线优化有几个要求。其中之一就是需要单一的生产者和消费者之间的数据任务。由于我们想在霍夫曼编码过程执行如图11.4所示的任务级别的流水线操作，我们必须确保每项任务都符合这一要求。在`sort`函数中，这其中一个任务，它只能消耗（读取但不写入）输入参数数据，并且只产生（写入但不读取）输出参数数据。为了满足这个要求，我们创建了内部`sorting`数组，它读取和写入整个函数。我们从函数开头的参数`in`中复制输入数据，并将最终结果写入函数结尾处的输出参数。这确保我们遵循生产者/消费者dataflow指令的要求。

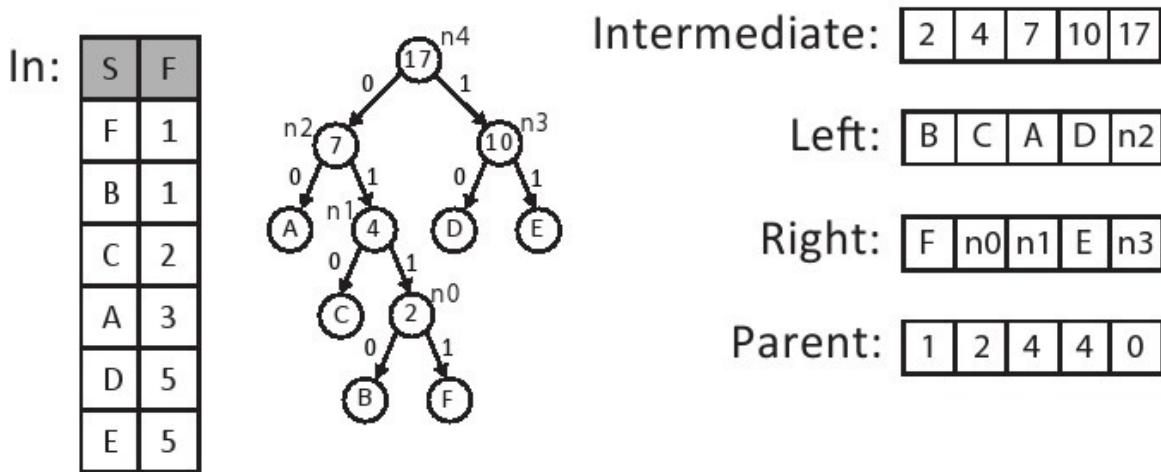
11.2.3 创建树

霍夫曼编码过程中的下一个功能形成了代表霍夫曼编码的二叉树。这在图11.8所示的`create_tree`函数中实现。`in[]`包含`num_symbols`符号元素，按最低频率到最高频率排序。该函数创建一个这些符号的二叉树，存储在三个名为`parent`，`left`和`right`的输出数组中。`left`和`right`数组表示树中每个中间节点的左侧和右侧子结点。如果子结点是一个叶结点，那么左边或右边数组的相应元素将包含孩子的符号值，否则包含特殊符号`INTERNAL_NODE`。同样，父数组保存每个中间结点的父结点的索引。树的根结点的父结点被定义为索引零点。树也是有序的，就像父母一样总是比孩子有更高的指数。因此，我们可以很好地实现树的自下而上和自上而下遍历。图11.7显示了这些数据结构的一个例子。六个符号按其频率排序并存储在数组中。霍夫曼树的结果存储在`parent`，`left`和`right`三个数组中。另外，每个中间节点的频率都存储在`frequency`数组中。为了便于说明，我们直接表示左右数组的节点号（例如，`n0`，`n1`等）。这些将在现实中保存一个特殊的内部结点值。

虽然在这样的树中存储复杂的数据结构可能是奇怪的，但在不允许数据分配的嵌入式编程中，这是非常普遍的[53]。事实上，`malloc()`和`free()`的c库实现通常以这种方式实现低级内存管理，以便操作系统返回的更大的内存分配（通常称为页面）中创建小的分配，这使得操作系统能够有效地管理内存的大量分配，并使用通常处理大型数据块的处理器页表和磁盘存储器来协调虚拟内存。4千字节是这些页面的典型大小。有关使用数组实现数据结构的更多想法，请参见[58]。

在霍夫曼树中，每个符号都与树中的叶结点相关联。树中的中间节点通过将两个符号以最小的频率分组并使用它们作为新中间节点的左右结点来创建。该中间结点的频率是每个子结点的频率之和。此过程通过迭代地创建具有最小频率的两个节点的中间节点，它可以包括其他的中间节点或叶节点。当所有中间结点都被并入二叉树中时，树的创建过程就完成了。在代码中有很多方法可以表示这个过程。例如，我们可以显式地创建一个表示树中每个节点按频率排序的数组。在这种情况下，选择结点添加到树中很简单，因为它们将始终位于已排序数组的相同位置。另一方面，将新创建的结点插入到列表中相对比较复杂，因为数组必须再次排序，还需要移动周围的元素。或者，我们可以向数组中的数据结构添加指针式数组索引，以便在不实际移动数据的情况下逻辑地排序数据。这会减少数据复制，但是会增加访问每个元素的成本并

需要额外的存储空间。在数据结构设计中，许多常规算法权衡适用于HLS的上下文中，也适用于处理器中。



然而，在这种情况下，我们可以做一些额外的简化观察。最重要的观察是，新的中间结点总是按频率顺序创建的。我们可以创建一个中间节点，其频率小于某个叶节点的频率，但是我们将永远不会创建一个比已经创建的中间节点更小的中间节点。这表明我们可以通过将节点存储在两个单独的数据结构中来维护排序的数据结构：一个排序的符号数组和一个排序的中间节点数组。当我们使用每个列表中的最低频率元素时，我们只需要追加到中间节点列表的末尾。还有一点额外的复杂性：因为我们可能需要从两个数组中删除零个，一个或两个元素，但事实证明这比复制节点数组复杂得多。

从概念上讲，这个算法与10.3节中讨论的合共排序算法非常相似。关键的区别在于元素从排序后的数组中移除时的操作。在合共排序中，最少的元素只是简单地插入到合适的数组位置。在这种情况下，两个最小元素被识别，然后合并到一个新的树结点中。

create_tree函数的实现代码如图11.8所示。第一个代码块定义我们在函数中使用的局部变量。frequency[]存储每个被创建的中间节点的频率，in_count跟踪哪些符号已被赋予树中的父节点，tree_count跟踪哪些新创建的中间结点已经被赋予父结点。通过主循环的每次迭代都会创建一个没有父级的新中间节点，所以tree_count和i之间的所有中间节点都尚未被分配到树的父节点中。

```

#include "huffman.h"
#include "assert.h"
void create_tree (
    /* input */ Symbol in[INPUT_SYMBOL_SIZE],
    /* input */ int num_symbols,
    /* output */ ap_uint<SYMBOL_BITS> parent[INPUT_SYMBOL_SIZE-1],
    /* output */ ap_uint<SYMBOL_BITS> left[INPUT_SYMBOL_SIZE-1],
    /* output */ ap_uint<SYMBOL_BITS> right[INPUT_SYMBOL_SIZE-1]) {
    Frequency frequency[INPUT_SYMBOL_SIZE-1];
    ap_uint<SYMBOL_BITS> tree_count = 0; // Number of intermediate nodes assigned a parent.
    ap_uint<SYMBOL_BITS> in_count = 0; // Number of inputs consumed.

    assert(num_symbols > 0);
    assert(num_symbols <= INPUT_SYMBOL_SIZE);
    for(int i = 0; i < (num_symbols-1); i++) {
#pragma HLS PIPELINE II=5
        Frequency node_freq = 0;

        // There are two cases.
        // Case 1: remove a Symbol from in[]
        // Case 2: remove an element from intermediate[]
        // We do this twice, once for the left and once for the right of the new intermediate node.

        assert(in_count < num_symbols || tree_count < i);
        Frequency intermediate_freq = frequency[tree_count];
        Symbol s = in[in_count];
        if((in_count < num_symbols && s.frequency <= intermediate_freq) || tree_count == i) {
            // Pick symbol from in[].
            left[i] = s.value; // Set input symbol as left node
            node_freq = s.frequency; // Add symbol frequency to total node frequency
            in_count++; // Move to the next input symbol
        } else {
            // Pick internal node without a parent.
            left[i] = INTERNAL_NODE; // Set symbol to indicate an internal node
            node_freq = frequency[tree_count]; // Add child node frequency
            parent[tree_count] = i; // Set this node as child's parent
            tree_count++; // Go to next parentless internal node
        }

        assert(in_count < num_symbols || tree_count < i);
        intermediate_freq = frequency[tree_count];
        s = in[in_count];
        if((in_count < num_symbols && s.frequency <= intermediate_freq) || tree_count == i) {
            // Pick symbol from in[].
            right[i] = s.value;
            frequency[i] = node_freq + s.frequency;
            in_count++;
        } else {
            // Pick internal node without a parent.
            right[i] = INTERNAL_NODE;
            frequency[i] = node_freq + intermediate_freq;
            parent[tree_count] = i;
            tree_count++;
        }
        // Verify that nodes in the tree are sorted by frequency
        assert(i == 0 || frequency[i] >= frequency[i-1]);
    }

    parent[tree_count] = 0; //Set parent of last node (root) to 0
}

```

图11.8：霍夫曼树的创建完整的代码。该代码将已排序的符号数组`in`作为输入，该数组中的元素数量为`n`，并输出`left`, `right`和`parent`三个数组的霍夫曼树。

主循环包含两个相似的代码块。每个代码块将`[in_count].frequency`中下一个可用符号的频率与下一个`[tree_count]`可用的中间结点的频率相比较，然后选择两个最低频率合并为一个新的中间结点的叶子。第一个代码块对新节点的左侧子节点执行这个过程，并将结果存储在`left[i]`中。第二个代码块选择新节点的右侧子节点执行这个过程，并将结果存储在`right[i]`中。这两种情况我们都需要小心确保这个比较是有意义的。在第一次循环迭代中，`tree_count==0`和`i==0`，所以没有有效的中间节点需要考虑，我们必须始终选择一个输入符号。在循环的最后迭代期间，所有的输入符号可能都会被使用，因此`in_count==num symbols`，我们必须始终使用一个中间节点。

循环的迭代次数以一种有趣的方式依赖于`num_symbols`的输入。由于每个输入符号都变成了二叉树中的一个叶节点，我们知道会有`num_symbols-1`个中间节点会被创建，因为这是二叉树的基本属性，在循环结束时，我们将创建`num symbols-1`个新节点，每个节点有两个子节点。这些子结点的`num_symbols`将是输入符号，`num_symbols-2`将是中间节点。在没有父节点的情况下，将有一个中间节点作为树的根。这最后的节点在最后一行代码中被赋予一个父索引0，这就完成了霍夫曼树的创建。

树中的中间节点的子节点可以是一个符号节点，也可以是中间节点。在创建霍夫曼树时，这些信息并不重要，尽管稍后我们在遍历树时会很重要。如果相应的子节点是内部节点的话，为了存储这个差别，一个特殊的值`INTERNAL_NODE`被存储在`left[]`和`right[]`数组中。请注意，这个存储本质上需要在数组中再加一位来表示。结果，`left[]`和`right[]`数组会比你想象的大一点。

在图11.8中，对于大数据集 ($n > 256$)，代码的近似延时 (以 n 为单位) 是多少？代码的哪些部分支配周期数量？

11.2.4 计算比特长度

`Compute_bit_length`函数为每个符号确定了树的深度。深度很重要，因为它决定了用于编码的每个符号的位数。计算树中每个节点的深度是使用下面的循环完成的：

$$\begin{aligned} \text{depth}(\text{root}) &= 0 \\ \forall n! = \text{root}, \text{depth}(n) &= \text{depth}(\text{parent}(n) + 1) \quad (11.1) \\ \forall n, \text{child_depth}(n) &= \text{depth}(n) + 1 \end{aligned}$$

可以通过遍历从根节点开始的树并按顺序探索每个内部节点来计算这种循环。当我们遍历每个内部节点时，我们可以计算节点的深度和任何子节点的相应的深度（增加1）。事实证明，我们实际上并不关心内部节点的深度，而只关心子节点的深度。下面的代码对这种循环进行了计算：

$$\begin{aligned} \text{child_depth}(\text{root}) &= 1 \\ \forall n! = \text{root}, \text{child_depth}(n) &= \text{child_depth}(\text{parent}(n) + 1) \quad (11.2) \end{aligned}$$

这个函数的代码如图11.9所示。函数的输入参数表示霍夫曼树的`parent[]`, `left[]`和`right[]`。`num_symbols`包含输入符号的数量，它比树的中间节点的数量多一个。`length_histogram[]`数组的每个元素都存储具有给定深度的符号数量。因此，如果有深度为3的5个符号，则`length_histogram[3] = 5`。`child_depth[]`在遍历树时存储每个内部节点的深度。每个内部结点的深度在`traverse_tree`循环中被确定后，`length_histogram[]`被更新。`Internal_length_histogram[]`用于确保我们的功能符合`dataflow`指令的要求，其中输出数组`length_histogram[]`永远不会被读取，`init_histogram`初始化这两个数组。

init_histogram循环具有II = 1的pipeline指令。是否有可能满足这个II？如果我们将II增加到更大的值，会发生什么？如果我们不应用这个指令，会发生什么？

树中的内部节点从根节点遍历，根节点具有最大的索引，直到索引为零。由于节点的数组是以自下而上的顺序创建的，因此该反向顺序导致树的自顶向下遍历，使得能够在通过节点的单个路径中计算每个节点的递归。对于每个节点，我们确定它的孩子的深度。然后，如果该节点存在有任何符号的孩子，那么我们确定有多少孩子，并相应地更新直方图。内部节点的子节点由特殊值INTERNAL_NODE来表示。

```
#include "huffman.h"
#include "assert.h"

void compute_bit_length (
    /* input */ ap_uint<SYMBOL_BITS> parent[INPUT_SYMBOL_SIZE-1],
    /* input */ ap_uint<SYMBOL_BITS> left[INPUT_SYMBOL_SIZE-1],
    /* input */ ap_uint<SYMBOL_BITS> right[INPUT_SYMBOL_SIZE-1],
    /* input */ int num_symbols,
    /* output */ ap_uint<SYMBOL_BITS> length_histogram[TREE_DEPTH]) {
    assert(num_symbols > 0);
    assert(num_symbols <= INPUT_SYMBOL_SIZE);
    ap_uint<TREE_DEPTH_BITS> child_depth[INPUT_SYMBOL_SIZE-1];
    ap_uint<SYMBOL_BITS> internal_length_histogram[TREE_DEPTH];

    init_histogram:
        for(int i = 0; i < TREE_DEPTH; i++) {
            #pragma HLS pipeline II=1
            internal_length_histogram[i] = 0;
        }

        child_depth[num_symbols-2] = 1; // Depth of the root node is 1.

    traverse_tree:
        for(int i = num_symbols-3; i >= 0; i--) {
            #pragma HLS pipeline II=3
            ap_uint<TREE_DEPTH_BITS> length = child_depth[parent[i]] + 1;
            child_depth[i] = length;
            if(left[i] != INTERNAL_NODE || right[i] != INTERNAL_NODE){
                int children;
                if(left[i] != INTERNAL_NODE && right[i] != INTERNAL_NODE) {
                    // Both the children of the original node were symbols
                    children = 2;
                } else {
                    // One child of the original node was a symbol
                    children = 1;
                }
                ap_uint<SYMBOL_BITS> count = internal_length_histogram[length];
                count += children;
                internal_length_histogram[length] = count;
                length_histogram[length] = count;
            }
        }
    }
}
```

图11.9：用于确定每个比特长度上符号数量的完整代码。

对于大数据集 ($n > 256$)，图11.9中代码的近似延时（以 n 为单位）是多少？代码的哪些部分支配周期数量？

此代码有几个循环。例如，一次循环发生是因为直方图计算。在这种情况下，该循环用II的3合成。如果你在pipeline指令中，目标是较低的II，会发生什么？你可以重写代码来消除循环并达到较低的II吗？

11.2.5 截断树

霍夫曼编码过程的下一部分是重组具有大于MAX_CODEWORD_LENGTH所指定的深度的节点。这是通过寻找任何具有更大深度的符号，并将它们移动到比所指定的目标更小的级别来完成的。有趣的是，这完全可以通过操作符号深度直方图来完成，只要直方图以与原始树上相同的模式一致的方式被修改。输入直方图包含在input_length_histogram中，它是前面章节中compute_bit_length()函数导出的。有两个相同的输出数组truncated_length_histogram1和truncated_length histogram2。它们在稍后的过程中传递给两个单独的函数(canonize_tree and create_codewords)，因此我们必须使两个数组遵守单个生产者，单个消费者的dataow指令的约束。

代码如图11.10所示。copy_in循环从输入数组input_length histogram复制数据。move_nodes循环包含修改直方图的大部分处理。最后，input_length_histogram函数将内部结果复制到函数结尾的其他输出。

copy_in for循环并未被优化。如果我们在这个循环中使用pipeline或unroll指令，延迟和启动间隔会发
生什么？设计的总延时和启动间隔会发生什么变化？

该函数继续在第二个节点移动 for循环中执行大部分循环计算。for循环是通过从最大的索引(TREE_DEPTH-树的最大深度)迭代遍历截断长度直方图数组开始的。这样继续下去直到找到一个非零元素或i达到MAX_CODEWORD_LENGTH的值。如果我们找不到一个非零元素，那意味着霍夫曼树的初始输入没有深度大于目标深度的节点。换句话说，我们可以退出这个函数不用执行任何截断。如果有一个大于目标深度的值，那么该函数继续重新组织树，使所有节点的深度都小于目标深度，这是通过重新排序 while 循环中的操作完成的。当有节点移动时，节点移动 for循环从深度最大的节点开始，然后继续，直到所有节点重新排列的深度都小于目标深度。每个节点移动for循环的迭代都在一个深度的移动结点上进行。

```

#include "huffman.h"
#include "assert.h"
void truncate_tree(
    /* input */ ap_uint<SYMBOL_BITS> input_length_histogram[TREE_DEPTH],
    /* output */ ap_uint<SYMBOL_BITS> output_length_histogram1[TREE_DEPTH],
    /* output */ ap_uint<SYMBOL_BITS> output_length_histogram2[TREE_DEPTH]
) {
    // Copy into temporary storage to maintain dataflow properties
copy_input:
    for(int i = 0; i < TREE_DEPTH; i++) {
        output_length_histogram1[i] = input_length_histogram[i];
    }

    ap_uint<SYMBOL_BITS> j = MAX_CODEWORD_LENGTH;
move_nodes:
    for(int i = TREE_DEPTH - 1; i > MAX_CODEWORD_LENGTH; i--) {
        // Look to see if there is any nodes at lengths greater than target depth
reorder:
        while(output_length_histogram1[i] != 0) {
#pragma HLS LOOP_TRIPCOUNT min=3 max=3 avg=3
            if (j == MAX_CODEWORD_LENGTH) {
                // Find deepest leaf with codeword length < target depth
                do {
#pragma HLS LOOP_TRIPCOUNT min=1 max=1 avg=1
                    j--;
                } while(output_length_histogram1[j] == 0);
            }

            // Move leaf with depth i to depth j+1.
            output_length_histogram1[j] -= 1; // The node at level j is no longer a leaf.
            output_length_histogram1[j+1] += 2; // Two new leaf nodes are attached at level j+1.

            output_length_histogram1[i-1] += 1; // The leaf node at level i+1 gets attached here.
            output_length_histogram1[i] -= 2; // Two leaf nodes have been lost from level i.

            // now deepest leaf with codeword length < target length
            // is at level (j+1) unless j+1 == target length
            j++;
        }
    }

    // Copy the output to meet dataflow requirements and check the validity
    unsigned int limit = 1;
copy_output:
    for(int i = 0; i < TREE_DEPTH; i++) {
        output_length_histogram2[i] = output_length_histogram1[i];
        assert(output_length_histogram1[i] >= 0);
        assert(output_length_histogram1[i] <= limit);
        limit *= 2;
    }
}

```

图11.10：重新组织霍夫曼树的完整代码，以便任何节点的深度低于参数MAX_CODEWORD_LENGTH指定的目标值。

重新排序 while循环在每次迭代中移动一个节点。第一个 if语句用于找到最大深度的叶节点。然后，我们将这个结点设为中间结点，并将其添加到比目标的深度大的叶节点作为子节点。这个if从句有一个 do/while循环，从目标向下迭代，寻找truncated_length_histogram数组中非零的记录。它的工作方式与开头的节点移动for循环相似。当它发现最深的叶节点小于目标深度时就停止，该节点的深度存储在j中。

现在我们有一个深度为i大于目标的节点，以及一个深度小于存储在j中的目标的节点。我们将节点从深度i和节点从j移到到深度为j+1的子节点。因此，我们添加两个符号到truncated_length_histogram[j+1]。我们正在生成一个深度为j的新的中间节点，因此我们从该级别去掉一个符号。我们移动另一个叶结点从深度i到深度i-1。我们从truncated_length_histogram[i]中减去2，因为其中1个节点进入级别j+1，另一级到达级别i-1。这些操作是在truncated_length_histogram数组上的四个语句中完成的。因为我们添加了一个符号到级别j+1，我们更新j，它拥有目标级别以下的最高级别，然后我们重复这个程序直到没有深度大于目标的附加符号。

该函数通过创建新的位长度的另一副本完成的。这是通过存储数组truncated_length_histogram1的更新位长度到truncated_length_histogram2中完成的。我们将这两个数组传递给huffman_encoding顶层函数的最后的两个函数，我们需要这两个数组来确保满足dataflow指令的约束。

11.2.6 Canonize树

编码过程的下一步是确定每个符号的位数，我们在图11.11所示的canoniz_tree函数中执行此操作。该函数按排序顺序的符号数组、符号总数 (num_symbols) 和霍夫曼树直方图的长度作为输入。symbol_bits[]包含每个符号使用的编码位数。因此，如果具有值0x0A的符号以4位编码，则symbol_bits[10] = 4。

canonization过程由两个循环组成，分别是init_bits 和 process_symbols。 init_bits循环首先执行，初始化symbol_bits[]数组为0。然后process_symbols循环按照从最小频率到最大频率的排序顺序处理符号。通常，频率最低的符号被分配最长的码，而频率最高的符号被分配最短的码。每次通过process_symbols循环时，我们都分配到一个符号的长度。符号的长度由内部do / while循环决定，它遍历长度直方图。该循环查找尚未分配码字的最大位长，并将该长度中的码字数存储在count中。每次通过外循环，count递减，直到我们用完码字。当count变为零时，内部do/while循环再次执行去寻找要分配的代码字的长度。

```

#include "huffman.h"
#include "assert.h"
void canonize_tree(
    /* input */ Symbol sorted[INPUT_SYMBOL_SIZE],
    /* input */ ap_uint<SYMBOL_BITS> num_symbols,
    /* input */ ap_uint<SYMBOL_BITS> codeword_length_histogram[TREE_DEPTH],
    /* output */ CodewordLength symbol_bits[INPUT_SYMBOL_SIZE] ) {
    assert(num_symbols <= INPUT_SYMBOL_SIZE);

    init_bits:
        for(int i = 0; i < INPUT_SYMBOL_SIZE; i++) {
            symbol_bits[i] = 0;
        }

        ap_uint<SYMBOL_BITS> length = TREE_DEPTH;
        ap_uint<SYMBOL_BITS> count = 0;

        // Iterate across the symbols from lowest frequency to highest
        // Assign them largest bit length to smallest
    process_symbols:
        for(int k = 0; k < num_symbols; k++) {
            if (count == 0) {
                //find the next non-zero bit length
                do {
#pragma HLS LOOP_TRIPCOUNT min=1 avg=1 max=2
                    length--;
                    // n is the number of symbols with encoded length i
                    count = codeword_length_histogram[length];
                }
                while (count == 0);
            }
            symbol_bits[sorted[k].value] = length; //assign symbol k to have length bits
            count--; //keep assigning i bits until we have counted off n symbols
        }
    }
}

```

图11.11：完整的canonizing霍夫曼树代码，它定义了每个符号的位数

```

int k = 0;
process_symbols:
for(length = TREE_DEPTH; length >= 0; length--) {
    count = codeword_length_histogram[length];
    for(i = 0; i < count; i++) {
#pragma HLS pipeline II=1
        symbol_bits[sorted[k++].value] = length;
    }
}

```

图11.12：图11.11中process_symbols循环的替代循环结构。

`process_symbols`循环无法进行流水线操作，因为内部do / while循环无法展开。这有点尴尬，因为内部循环通常只执行一次，步进到直方图中的下一个长度。只有在极少数情况下，例如我们碰巧没有分配任何码字，内部循环需要执行多次。在这种情况下，没有太多的损失，因为循环中的所有操作都是简单的操作，除了内存操作之外，这些操作不可能是流水线操作。还有其他的方式来构建这个循环，但是，它可以是流水线的。一种可能性是使用外部for循环迭代`codeword_length_histogram[]`和内部循环来计算每个符号，如图11.12所示。

实现图11.11中的代码和图11.12中的替代代码结构。哪一个会实现更高的性能？哪种编码风格对你来说更自然？

11.2.7 创建码字

编码过程中的最后一步是为每个符号创建码字。该流程根据Canonical霍夫曼代码的特性，按顺序简单地分配每个符号。第一个特性是具有相同长度前缀的较长的代码比短码有更高的数值。第二个特性是相同长度的代码随着符号值的增加而增加。为了在保持代码简单的同时实现这些属性，确定每个长度的第一个码字是有用的。如果我们知道由码字长度直方图给出的每个长度的码字的数量，则可以使用以下循环来得出结果：

$$\begin{aligned} \text{first_codeword}(1) &= 0 \\ \forall i > 1, \text{first_codeword}(i) &= \\ (\text{first_codeword}(i - 1) + \text{codeword_length_histogram}(i - 1)) &\ll 1 \end{aligned} \quad (11.3)$$

实际上，不是一个接一个地分配码字，而是这个循环首先分配了所有的码字。这使我们能够按照符号值的顺序来分配码字，而不用担心按长度或频率对它们进行排序。除了将码字分配给符号外，我们还需要对码字进行格式化，使得它们可以很容易地用于编码和解码。使用霍夫曼编码的系统通常以位反转的顺序存储码字。这使得解码过程更容易，因为从根节点到叶结点，比特流被按照与解码期间树遍历相同的顺序存储。

实现`createcodewords`函数的代码如图11.13所示。`Symbol_bits[]`包含每个符号的码字和长度，`codeword_length_histogram[]`包含每个长度的码字数量，`encoding[]`的输出表示每个符号的编码。每个元素由实际的码字和每个码字的长度打包在一起组成。码字的最大长度由`MAX_CODEWORD_LENGTH`参数给定。反过来，这又决定了保持码字所需的位数，它由`CODEWORD_LENGTH_BITS`给定。`CODEWORD_LENGTH_BITS`编码数组中每个元素的最低有效位包含从输入数组`symbol_bits`接收到的相同值。每个编码元素的高位`MAX_CODEWORD_LENGTH`包含实际码字。使用27位`MAX_CODEWORD_LENGTH`导致`CODEWORD_LENGTH_BITS`为5是一个特别有用的组合，因为`encoding[]`中的每个元素满足一个32位字长度。

该代码主要由两个循环组成，分别是`first_codewords`和`assign_codewords`。`first_codewords`循环寻找每个长度的第一个码字，并执行公式11.3的循环。`assign_codewords`循环最终将每个符号与每个码字相关联。

码字是通过每个码字的长度并索引到`first_codeword[]`的正确元素中找到的。该代码的主要复杂性在于位反转过程，它是基于`bit_reverse32`函数。我们之前在FFT章节中已经讨论了这个函数（参见第5.3章），所以我们不会在这里再讨论它。在反转码字中的位之后，下一个语句删除最不重要的'0'位，只留下位反转的码字，然后将位反转的码字与低位中的符号的长度一起打包在高位中，并存储在`encoding[]`中。最后，`first_codeword[]`中的值是递增的。

在图11.13的代码中，输入实际上包含一些冗余信息。特别是，我们可以使用直方图来计算每个码字symbol_bits[]的长度，计算存储在codeword_length_histogram[]中每个比特长度的符号数。相反，在此代码中，我们选择复用最初计算的直方图的truncate_tree（）函数。这样，我们可以通过重新计算直方图来节省存储空间。这是一个很好的考量吗？在这个函数中，需要多少资源来计算直方图？通过流水线传递直方图需要多少资源？

估计图11.13中代码的延迟。

```
#include "huffman.h"
#include "assert.h"
#include <iostream>
void create_codeword(
    /* input */ CodewordLength symbol_bits[INPUT_SYMBOL_SIZE],
    /* input */ ap_uint<SYMBOL_BITS> codeword_length_histogram[TREE_DEPTH],
    /* output */ PackedCodewordAndLength encoding[INPUT_SYMBOL_SIZE]
) {
    Codeword first_codeword[MAX_CODEWORD_LENGTH];

    // Computes the initial codeword value for a symbol with bit length i
    first_codeword[0] = 0;
    first_codewords:
    for(int i = 1; i < MAX_CODEWORD_LENGTH; i++) {
#pragma HLS PIPELINE II=1
        first_codeword[i] = (first_codeword[i-1] + codeword_length_histogram[i-1]) << 1;
        Codeword c = first_codeword[i];
        //           std::cout << c.to_string(2) << " with length " << i << "\n";
    }

    assign_codewords:
    for (int i = 0; i < INPUT_SYMBOL_SIZE; ++i) {
#pragma HLS PIPELINE II=5
        CodewordLength length = symbol_bits[i];
        //if symbol has 0 bits, it doesn't need to be encoded
        make_codeword:
        if(length != 0) {
            //           std::cout << first_codeword[length].to_string(2) << "\n";
            Codeword out_reversed = first_codeword[length];
            out_reversed.reverse();
            out_reversed = out_reversed >> (MAX_CODEWORD_LENGTH - length);
            // std::cout << out_reversed.to_string(2) << "\n";
            encoding[i] = (out_reversed << CODEWORD_LENGTH_BITS) + length;
            first_codeword[length]++;
        } else {
            encoding[i] = 0;
        }
    }
}
```

图11.13:为每个符号生成canonical霍夫曼码字的完整代码。可以在知道每个符号的比特数的情况下计算码字（存储在输入数组symbol_bits[]中）。另外，我们有另一个输入数组codeword_length_histogram[]，其在每个条目中存储具有该位长度的码字的符号数，输出是存储在encoding[]数组中的每个符号的代码字。

现在让我们来看看我们的运行示例，并说明如何使用它来导出最初的码字。在这个例子中，符号A，D和E对于它们的编码有两比特；符号C有三比特；而符号B和F有四比特。因此，我们有：

$$\begin{aligned} \text{bit_length}(1) &= 0 \\ \text{bit_length}(2) &= 3 \\ \text{bit_length}(3) &= 1 \\ \text{bit_length}(4) &= 2 \end{aligned} \quad (11.4)$$

使用公式11.3来计算第一个码字的值，我们定义：

$$\begin{aligned} \text{first_codeword}(1) &= 0 & = 0b0 \\ \text{first_codeword}(2) &= (0 + 0) \ll 1 & = 0b00 \\ \text{first_codeword}(3) &= (0 + 3) \ll 1 & = 0b110 \\ \text{first_codeword}(4) &= (6 + 1) \ll 1 & = 0b1110 \end{aligned} \quad (11.5)$$

一旦我们确定了这些值，然后从小到大依次考虑每个符号。对于每个符号，确定其码字的长度并分配下一个码字适当的长度。在运行示例中，考虑符号按字母顺序A，B，C，D，E和F排列，符号A有两比特用于编码，执行查找first_codeword[2]=0。因此，分配给A的码字是0b00，我们增加first_codeword[2]的值到1。符号B有四个比特。由于first_codeword[4]=14=0b1110，所以码字是0b1110。符号C有三个比特。first_codeword[3]=6=0b110，码字是110。符号D具有两比特，因此first_codeword[2]=1=0b01；请记住，我们在将码字分配给符号A后这个值将递增。符号E有两位，所以它的码字是0b01+1=0b10。F有4比特，所以它的码字是0b1110+1=0b1111。所有符号的最终码字是：

$$\begin{aligned} A &\rightarrow 00 \\ B &\rightarrow 1110 \\ C &\rightarrow 110 \\ D &\rightarrow 01 \\ E &\rightarrow 10 \\ F &\rightarrow 1111 \end{aligned} \quad (11.6)$$

11.2.8 测试平台

代码的最后部分是测试平台，如图11.14所示。总体结构是从文件中读取输入的频率值，使用huffman_encoding函数处理它们，并将结果码字与存储在文件中的现有黄金参考进行比较。main()函数首先创建从文件中读取频率所需的变量（在这里，文件是huffman.random256.txt）并将它们放入in[]中，这是在file_to_array函数中完成的，该函数将输入数据的文件名和数据长度（数组长度）作为输入，并将该文件中的输入存储到array[]变量中。这个文件包含了每个符号的频率，频率值按照符号顺序存储，因此文件第一个值表示符号'0'的频率，依此类推。

main()函数通过使用文件中的频率在in[]中进行初始化，然后它会调用顶层的霍夫曼函数。该函数返回encoding[]中的编码符号值。由于处理结果应该是一个前缀码，因此我们检查前缀码的属性是否确实得到满足，然后将结果与存储在文件huffman.random256.gold中的黄金参考中的代码字进行比较。然后将结果写入名为random256.out的文件中，并使用diff工具执行文件比较操作。如果文件相同，diff工具返回'0'，如果文件不同则返回非零。因此，如果条件不同时，则执行if条件，而当文件相同时，执行else条件。在这两种情况下，我们都会打印一条消息并将return_val设置为适当的值。Vivado RHLS工具在协同仿真过程中使用该返回值来检查结果的正确性。如果检查通过，则返回值为0；如果不通过，则返回值不为零。

```
#include "huffman.h"
```

```

#include <stdio.h>
#include <stdlib.h>

void file_to_array(const char *filename, ap_uint<16> *&array, int array_length) {
    printf("Start reading file [%s]\n", filename);
    FILE * file = fopen(filename, "r");
    if(file == NULL) {
        printf("Cannot find the input file\n");
        exit(1);
    }

    int     file_value = 0;
    int     count = 0;
    array = (ap_uint<16> * ) malloc(array_length*sizeof(ap_uint<16>));

    while(1) {
        int eof_check = fscanf(file, "%x", &file_value);
        if(eof_check == EOF) break;
        else {
            array[count++] = (ap_uint<16>) file_value ;
        }
    }
    fclose(file);

    if(count != array_length) exit(1);
}

int main() {
    printf("Starting canonical Huffman encoding testbench\n");
    FILE * output_file;
    int return_val = 0;
    ap_uint<16> * frequencies = NULL;
    file_to_array("huffman.random256.txt", frequencies, INPUT_SYMBOL_SIZE);

    Symbol in[INPUT_SYMBOL_SIZE];
    for (int i = 0 ; i < INPUT_SYMBOL_SIZE; i++) {
        in[i].frequency = frequencies[i];
        in[i].value = i;
    }

    int num_nonzero_symbols;
    PackedCodewordAndLength encoding[INPUT_SYMBOL_SIZE];
    huffman_encoding(in, encoding, &num_nonzero_symbols);

    output_file = fopen("huffman.random256.out", "w");
    for(int i = 0; i < INPUT_SYMBOL_SIZE; i++)
        fprintf(output_file, "%d, %x\n", i, (unsigned int) encoding[i]);
    fclose(output_file);

    printf ("\n*****Comparing against output data***** \n\n");
    if (system("diff huffman.random256.out huffman.random256.golden")) {
        fprintf(stdout, "*****\n");
        fprintf(stdout, "FAIL: Output DOES NOT match the golden output\n");
        fprintf(stdout, "*****\n");
        return_val = 1;
    } else {
        fprintf(stdout, "*****\n");
        fprintf(stdout, " PASS: The output matches the golden output\n");
        fprintf(stdout, "*****\n");
        return_val = 0;
    }

    printf("Ending canonical Huffman encoding testbench\n");
}

```

```
    return return_val;  
}
```

图11.14：完整的*canonical*霍夫曼编码测试平台代码。该代码使用输入文件中的数据初始化*in*数组，并将它传递给顶层的*huffman_encoding*函数，然后它将结果码字存储到一个文件中，并将其与另一个黄金参考文件进行比较，最后打印出比较结果，并返回适当的值

11.3 结论

霍夫曼编码是许多应用中常用的数据压缩类型。虽然使用霍夫曼码进行编码和解码是相对简单的操作，但生成霍夫曼码本身可能是计算上具有挑战性的问题。在许多系统中，拥有相对较小的数据块是有利的，这意味着必须经常创建新的霍夫曼码并加速它是值得。与我们在本书中研究过的其他算法相比，创建霍夫曼码包含许多具有完全不同代码结构的步骤。有些相对容易实现并行化，而另一些实现起来则更具有挑战性。算法的某些部分自然具有较高的O(n)复杂度，这意味着它们必须更加并行化以实现平衡的流水线。但是，使用Vivado HLS中的数据流指令，这些不同的代码结构可以相对容易地链接在一起。

词汇表

数组分离

把一个逻辑数组分离到不同的内存中储存

比特流

用于编程FPGA功能的配置数据

BRAM

一个可编程的RAM模块，嵌入在FPGA上以完成数据储存和交流

C/RTL同步模拟

用C测试平台的测试矢量来完成HLS产生的RTL设计这个过程

压缩行储存

也就是CRS，是一个表示稀疏矩阵的方法。它能够有效的表示一个很大但有很多无效数字的矩阵。

数据率

任务处理输入数据的频率。通常以比特每秒的单位表示，取决于输入数据的大小。

离散傅立叶变换

DFT，以一个离散信号为输入，把它转换成频率域的信号的过程。

EDA

电子设计自动化工具，是一些帮助完成硬件设计的软件工具。

快速傅里叶变换

FFT，优化版本的离散傅立叶变换，需要更少量的操作。

触发器

触发器（FF）是可以储存信息的电路。我们通常默认它可以储存1比特的信息，是组成数字电路的最基本的模块之一。

有限脉冲响应

FIR，一个常见的数字信号处理任务，将一个输入信号和既定系数决定的信号做卷积。FIR通常用硬件实现比较有效。

FPGA

现场可编程门阵列，是一种出厂后仍可以被用户自定义集成电路。

HLS

高层次综合，是一种将算法描述转换成RTL的硬件设计语言，它可以详细规定电路每个周期的行为。

I/O模块

一个I/O模块可以提供FPGA与系统中其他部分连接的接口。I/O模块可以与内存（包括片上内存和不在片上的DRAM），微处理器（通过AXI或其他协议），传感器，校准器等部分交流。

IP核

一个RTL层级的部件，具有完善的接口来接入整个设计。通常出于知识产权（Intellectual Property）目的向其他公司隐藏，由此得名。

逻辑综合

把一个glsl设计转换成设备层的连线表的过程。

循环交叉

一种改变循环里操作的顺序的代码转变。这种转变通常是解决递归的有效方式。

循环流水

允许一个循环的多个步骤同时运行，享有同样的功能单位。

查找表

查找表（LUT）是一种地址信号作为输入，相对应位置的存储数值作为输出的内存形式。它是现代FPGA的一个重要组成部分。

连线表

设计的中间连接部分，由设计层的主要部件和他们之间的连接方式组成。在FPGA设计中“主要部件”包括查找表，触发器，和BRAM。

部分循环展开

循环的主体被复制成多次执行的过程。通常被用于处理系统（PS）以减少循环的占用（overhead），也会有利于向量化。

放置与路由

把设备层主要部件的连线表转换成一个具体装置配置的过程。

处理

数据流结构中的一个独立部分。

处理部件

设计中的一个可以简单的处理同步执行的部件，在HLS中通常出现在数据流结构的描述中。

递归（recurrence）

一种代码结构，产出的电路中带有反馈循环，这种结构会限制电路的产力。

ROM

只读内存（Read-Only Memory）是一种被初始化到一定数值的内存，它只能被读取无法被写入数据。在很多情况下用ROM储存是一种非常好的优化因为他们存储的数据不会变。

路由通道

路由通道提供FPGA内各部件灵活的连接方式。

RTL

寄存器转移级（Register Transfer Level），是一种硬件描述，它用各个寄存器之间的逻辑操作来搭建一个同步数字电路。它也是现代数字设计的一个常见设计入口。

slice

一些查找表，触发器和mux的集合，具体配置在FPGA资源配置报告中有详述。

分类单元

一个简单的组件，是分类算法的系统的一部分。通常分类单元可以执行两个对象的比较和交换。

稳定分类

一种分类算法，如果它能保证排序前2个相等数的前后位置顺序和排序后它们两个的前后位置顺序相同，那么它就是稳定分类。

静态单赋值

静态单赋值是编译器中的一种中介码，每个变量在这里只被赋值一次。这种形式让优化变得更容易。

交换盒

交换盒连接各个路由通道，为可编程逻辑和I/O模块之间数据的流通提供了一个灵活的路由结构。

脉动阵列

一个协同完成复杂算法的处理部件阵列。脉动阵列的设计方向通常是让每个处理部件封装一些本地信息，并且只和他们的本地邻居进行交流。这样简单的增大阵列的尺寸就可以增加处理的项目的尺寸。

任务

行为，或者说高层次综合计算的基本的原子级单位。高层次综合中与之对应的是函数调用。

任务流水

用流水的形式在一个加速器上同时执行多个任务。

任务间隔

一个任务开始和下一个任务开始之间的间隔。

任务延迟

一个任务开始和它结束之间的间隔。

参考文献

1

Ameer M.S. Abdelhadi and Guy G.F. Lemieux. Modular multi-ported SRAM-based memories. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 35{44. ACM, 2014. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554773. URL <http://doi.acm.org/10.1145/2554688.2554773>.

2

SystemC. Accellera, 2.3.2 edition, October 2017. URL <http://www.accellera.org/downloads/standards/systemc>.

3

Hassan M. Ahmed, Jean-Marc Delosme, and Martin Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Computer*, 15(1):65{82, 1982.s

4

Raymond J. Andraka. Building a high performance bit-serial processor in an FPGA. In Proceedings of Design SuperCon, volume 96, pages 1{5, 1996.

5

Oriol Arcas-Abella et al. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In Proceedings of the International Field Programmable Logic and Applications Conference (FPL), pages 1{8. IEEE, 2014.

6

AMBA AXI and ACE Protocol Specification. ARM Limited, v1.0 edition, 2013. URL <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022e/index.html>. ARM IHI 0022E.

7

Bryce E. Bayer. Color imaging array, July 1976. US 3971065.

8

Samuel Bayliss and George A. Constantinides. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 195{204. ACM, February 2012.

9

Marcus Bednara et al. Tradeo analysis and architecture design of a hybrid hardware/software sorter. In Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP), pages 299{308. IEEE, 2000.

10

Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In Proceedings of the International Field Programmable Logic and Applications Conference (FPL), pages 213{222. Springer, 1997.

11

Guy E. Blelloch. Prex sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

12

Stephen Brown and Jonathan Rose. FPGA and CPLD architectures: A tutorial. IEEE Design and Test of Computers, 13(2):42{57, 1996.

13

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 33{36. ACM, 2011.

14

Jianwen Chen, Jason Cong, Ming Yan, and Yi Zou. FPGA-accelerated 3D reconstruction using compressive sensing. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 163{166. ACM, 2012.

15

Jason Cong, Bin Liu, Stephen Neuendorfer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD), 30(4):473{491, 2011.

16

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. Mathematics of Computation, 19(90):297{301, 1965. ISSN 00255718. URL <http://www.jstor.org/stable/2003354>.

17

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliord Stein. Introduction to Algorithms, Third Edition. MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

18

Philippe Coussy and Adam Morawiec. High-level synthesis, volume 1. Springer, 2010.

19

Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 189{194, 2017. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021754. URL <http://doi.acm.org/10.1145/3020078.3021754>.

20

Jerey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1):107{113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL <http://doi.acm.org/10.1145/1327452.1327492>.

21

Alvin M. Despain. Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers*, 100(10):993{1001, 1974.

22

J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In Proceedings of the International Field Programmable Logic and Applications Conference (FPL), pages 29{34, August 2007. doi: 10.1109/FPL.2007.4380621.

23

L Peter Deutsch. DEFLATE compressed data format specication version 1.3. 1996.

24

Jean Duprat and Jean-Michel Muller. The CORDIC algorithm: new results for fast VLSI implementation. *IEEE Transactions on Computers*, 42(2):168{178, 1993.

25

Brian P. Flannery, William H. Press, Saul A. Teukolsky, and William Vetterling. Numerical recipes in C. Press Syndicate of the University of Cambridge, New York, 1992.

26

Daniel D. Gajski, Nikil D. Dutt, Allen C.H. Wu, and Steve Y.L. Lin. High-Level Synthesis: Introduction to Chip and System Design. Springer Science & Business Media, 2012.

27

W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and prot. In Proceedings of the November 7-10, 1966, fall joint computer conference, pages 563{578. ACM, 1966.

28

Nivia George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Su-jeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In Proceedings of the International Field Programmable Logic and Applications Conference (FPL), pages 1{8. IEEE, 2014.

29

Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alexandru Nicolau. SPARK: A Parallelizing Approach to the High-level Synthesis of Digital Circuits. Kluwer, 2004. ISBN 1-4020-7837-4.

30

Scott Hauck and Andre DeHon. Recongurable computing: the theory and practice of FPGA-based computation, volume 1. Morgan Kaufmann, 2010.

31

James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. ACM Trans. Graph., 35(4):85:1{85:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925892. URL <http://doi.acm.org/10.1145/2897824.2925892>.

32

M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. ASSP Magazine, IEEE, 1(4):14{21, October 1984. ISSN 0740-7467. doi: 10.1109/MASSP.1984.1162257.

33

David A. Human. A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9):1098{1101, 1952.

34

Ryan Kastner, Anup Hosangadi, and Farzan Fallah. Arithmetic optimization techniques for hardware and software design. Cambridge University Press, 2010.

35

David Knapp. Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler. Prentice-Hall, 1996. ISBN 0-13-569252-0.

36

Donald Ervin Knuth. The art of computer programming: sorting and searching, volume 3. Pearson Education, 1998.

37

Charles Eric Laforest, Zimo Li, Tristan O'rourke, Ming G. Liu, and J. Gregory Stean. Composing multi-ported memories on FPGAs. ACM Transactions on Recongurable Technology and Systems (TRETS), 7(3):16:1{16:23, September 2014. ISSN 1936-7406. doi: 10.1145/2629629. URL <http://doi.acm.org/10.1145/2629629>.

38

Glen G. Langdon Jr, Joan L. Mitchell, William B. Pennebaker, and Jorma J. Rissanen. Arithmetic coding encoder and decoder system, February 27 1990. US Patent 4,905,297.

39

Dajung Lee, Janarbek Matai, Brad Weals, and Ryan Kastner. High throughput channel tracking for JTRS wireless channel emulation. In Proceedings of the International Field Programmable Logic and Applications Conference (FPL). IEEE, 2014.

40

Edward A. Lee and David G. Messerschmitt. Pipeline interleaved programmable DSPs: Architecture. IEEE Transactions on Acoustics, Speech, and Signal Processing (TASSP), 35(9): 1320{1333, September 1987.

41

Edward A. Lee and Pravin Varaiya. Structure and Interpretation of Signals and Systems, Second Edition. 2011. ISBN 0578077191. URL LeeVaraiya.org.

42

Edward Ashford Lee. *Plato and the Nerd: The Creative Partnership of Humans and Technology*. MIT Press, 2017. ISBN 978-0262036481.

43

C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*, 1993.

44

G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. Architecture and synthesis for area-efficient pipelining of irregular loop nests. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)*, 36(11):1817{1830, November 2017. ISSN 0278-0070. doi: 10.1109/TCAD.2017.2664067.

45

Rui Marcelino et al. Sorting units for FPGA-based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, pages 11{22. Springer, 2008.

46

Janarbek Matai, Pingfan Meng, Lingjuan Wu, Brad T Weals, and Ryan Kastner. Designing a hardware in the loop wireless digital channel emulator for software defined radio. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. IEEE, 2012.

47

Janarbek Matai, Joo-Young Kim, and Ryan Kastner. Energy efficient canonical human encoding. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2014.

48

Janarbek Matai, Dustin Richmond, Dajung Lee, and Ryan Kastner. Enabling FPGAs for the masses. *arXiv preprint arXiv:1408.5870*, 2014.

49

Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 195{204. ACM, 2016. ISBN 978-1-4503-3856-1. doi: 10.1145/2847263.2847268. URL <http://doi.acm.org/10.1145/2847263.2847268>.

50

Carver Mead and Lynn Conway. Introduction to VLSI systems, volume 1080. Addison-Wesley Reading, MA, 1980.

51

Giovanni De Micheli. Synthesis and optimization of digital circuits. McGraw-Hill Higher Education, 1994.

52

Shahnam Mirzaei, Anup Hosangadi, and Ryan Kastner. FPGA implementation of high speed multipliers using add and shift method. In Computer Design, 2006. ICCD 2006. International Conference on, pages 308{313. IEEE, 2007.

53

MISRA. Guidelines for the Use of the C Language in Critical Systems. March 2013. ISBN 978-1-906400-10-1. URL <https://www.misra.org.uk>.

54

Rene Mueller et al. Sorting networks on FPGAs. The VLDB Journal|The International Journal on Very Large Data Bases, 21(1):1{23, 2012.

55

Jorge Ortiz et al. A streaming high-throughput linear sorter system with contention buering. International Journal of Reconigurable Computing, 2011.

56

Marios C. Papaeftymiou. Understanding retiming through maximum average-weight cycles. In SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures, pages 338{348, 1991.

57

William B. Pennebaker. JPEG: Still image data compression standard. Springer, 1992.

58

Robert Sedgewick. Algorithms in C. Addison-Wesley, 2001. ISBN 978-0201756081.

59

Bhaskar Sherigar and Valmiki K. Ramanujan. Human decoder used for decoding both advanced audio coding (AAC) and MP3 audio, June 29 2004. US Patent App. 10/880,695.

60

F. Winterstein, S. Bayliss, and G. A. Constantinides. High-level synthesis of dynamic data structures: A case study using Vivado HLS. In Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), pages 362{365, December 2013. doi: 10.1109/FPT.2013.6718388.

61

Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. Communications of the ACM, 30(6):520{540, 1987.

62

UltraScale Architecture Congurable Logic Block (UG574). Xilinx, v1.5 edition, February 2017. URL https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

63

Vivado Design Suite User Guide: High-Level Synthesis (UG902). Xilinx, v2017.1 edition, April 2017. URL https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf.

64

UltraScale Architecture Conguration (UG570). Xilinx, v1.7 edition, March 2017. URL https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf.