国外计算机科学经典教材

Digital Signal Processing with Field Programmable Gate Arrays

数字信号处理的FPGA实现

Uwe Meyer-Baese 著
刘凌 胡永生 译

清华大学出版社
http://www.tup.tsinghua.edu.cn

Digital Signal Processing with Field Programmable Gate Arrays

现场可编程门阵列(FPGA)是数字信号处理的一次重大变革。Novel FPGA 系列正逐步取代前端数字信号处理算法的 ASIC 和 PDSP。本书主要讲述了这些算法的有效实现，首先介绍了当前的 FPGA 技术、元器件和用于设计最新 DSP 系统工具的概况。第 1 章中的案例研究是本书 30 多个示例的基础。后续的几章主要介绍了计算机算法的概念、理论、FIR 和 IIR 滤波器的实现、多级数字信号处理系统、DFT 和 FFT 算法以及一些在将来更具潜力的高级算法。另外，本书的每一章后面都附有相应的练习，书中每个示例的 VERILOG 源代码和词汇表都包含在附录中。

ISBN 7-302-06035-5
9 787302 060352 >
定价：48.00 元

数字信号处理的 FPGA 实现

Uwe Meyer-Baese 著

刘凌 胡永生 译

清 华 大 学 出 版 社

北京市版权局著作权合同登记号：图字：01-2002-4078

内 容 简 介

本书是一本有关最新数字信号处理(DSP)的专著。书中通过大量的程序实例，全面、精辟地介绍了利用现场可编程门阵列(FPGA)实现数字信号处理的方方面面。本书首先介绍了当前的 FPGA 技术、元器件和用于设计最新 DSP 系统工具的概况；接着主要介绍了计算机算法的概念、理论、FIR 和 IIR 滤波器的实现、多级信号处理和傅立叶变换等内容；最后讲解了一些专用算法，如数论变换和密码术算法等。

本书内容详实、讲解深入浅出、实用性极强，可作为高等院校电子、电气以及相关专业的课程教材，也可供从事数字信号处理的专业人员参考。

图书在版编目(CIP)数据

数字信号处理的 FPGA 实现/(美)贝斯著；刘凌，胡永生译.—北京：清华大学出版社，2002
书名原文：Digital Signal Processing with Field Programmable Gate Arrays
ISBN 7-302-06035-5

Ⅰ.数… Ⅱ.①贝…②刘…③胡… Ⅲ.数字信号—信号处理 Ⅳ.TN911.72
中国版本图书馆 CIP 数据核字(2002) 第 084551 号

出 版 者：清华大学出版社(北京清华大学学研大厦，邮编 100084)
http://www.tup.tsinghua.edu.cn
责任编辑：李阳
封面设计：康博
版式设计：康博
印 刷 者：国防工业出版社印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092 1/16　**印张**：23.25　**字数**：595 千字
版　　次：2003 年 1 月第 1 版　2003 年 1 月第 1 次印刷
书　　号：ISBN 7-302-06035-5/TP・3600
印　　数：0001～4000
定　　价：48.00 元

译 者 序

众所周知，信号与信号处理是信息科学中近十几年来发展最为迅速的学科之一。而FPGA(Field Programmable Gate Array)正处于革命性数字信号处理的前沿。全新的FPGA系列正在越来越多地替代ASIC和PDSP用作前端数字信号处理的运算。FPGA具有许多与ASIC相同的特点，例如：在规模、重量和功耗等方面都有所降低。而且吞吐量更高、能够更好地防止未授权复制、元器件和开发成本进一步降低，开发时间也大大缩短。还具有在线路中可重复编程的特性。从而可以产生更为经济的设计。正如我们现在已经看到的，随着FPGA在数字信号处理中的大规模应用，正在日渐深入地影响我们的生产和生活，也必将在这一领域引起深刻的变革。本书的主旨就是讲述如何用FPGA实现数字信号处理。

简洁明了是作者Uwe Meyer-baese先生的风格，也是本书的风格。与大多数强调信号处理理论的著作相比，本书摒弃了传统的说教内容，更多地从应用的角度出发，注重如何用FPGA实现这些理论，并且致力于找到最适合解决问题的途径。在本书的开头，简要地介绍了当前FPGA技术的发展和用于设计的元器件以及设计DSP系统的工具的技术发展水平，之后就给出了频率合成器的设计研究示例。以示例教学的方式讲述了设计的编译步骤、仿真、性能评估、功耗估算和平面布置图。在以后的章节中系统地介绍了数制、算法和FIR、IIR滤波器的实现以及多级信号处理和各种傅立叶变换形式及其实现。最后一章简要介绍了当前的前沿课题，对其中一些方面感兴趣的读者可以在其指导下，作进一步的研究。

在本书中，共给出33个程序示例(正文中是VHDL代码，附录中有相应的Verilog代码)，还有213张插图和57份表格，均充分体现了Uwe Meyer-baese先生这部著作的特点。译者认为这也是除了学习本书中的知识以外的更大收获。

鉴于FPGA是一项全新的技术，特别是对于国内而言，相应的中文资料少之又少，又加之本人水平有限，本书的翻译不能够说是填补空白，只是想起到抛砖引玉的效果，只要能够对读者学习、了解和掌握用FPGA实现数字信号处理起到一定的作用也就心满意足了。在翻译中定有不妥之处，恳请读者批评指正。

另外，建议读者在阅读本书之前最好认真复习一下高等代数中多项式、行列式、矩阵、线性空间、线性变换和群、环、域等方面的概念和知识，能够熟练地完成各种矩阵的运算。并且最好是预先学习过信号与系统、数字信号处理和通信系统等相关课程，这样对您理解和掌握本书的知识要点会有极大的帮助，否则您会感到非常吃力，毕竟这并不是一本简单的科普教材。

译 者

2002年6月于北京

前　言

正如可编程数字信号处理器(programmable digital signal processor，PDSP)在近 20 年前出现时的情形一样，如今，现场可编程门阵列(field-programmable gate array，FPGA)正处于革命性的数字信号处理技术的前沿。过去，前端的可编程数字信号处理(digital signal processing, DSP)算法，例如 FFT、FIR 和 IIR 滤波器，都是利用 ASIC 或者 PDSP 构建的，但现在大多为 FPGA 所代替。现代的 FPGA 系列都提供了支持以低系统开销、低成本实现高速乘-累加(Multiply-accumulate, MAC)超前进位链(Xilinx XC4000，Altera FLEX)的 DSP 算法[1]。以前的 FPGA 系列大多面向 TTL“胶合逻辑”，没有 DSP 函数需要的大量的门数量。这些前端算法的有效实现就是本书要讲解的主要内容。

在 21 世纪初，我们就看到，两个可编程逻辑器件(programmable logic device，PLD)市场的领导者(Altera 和 Xilinx)都宣称获得了超过 10 亿美元的收入。在过去 10 年中，FPGA 一直保持以 20%以上的速度稳步增长，超过 ASIC 和 PDSP 10%以上。这源于 FPGA 具有许多与 ASIC 相同的特点，比如：在规模、重量和功耗等方面都降低了，同时还具有更高的吞吐量、更好的防止非授权复制的安全性、降低了元器件和开发的成本，并且还降低了电路板测试成本。此外，还声称具有优于 ASIC 的优势，例如：开发时间的缩短(快速的原型设计)、在线路中可重复编程的性质、更低的 NRE 成本，对于需求少于 1 000 个单元的解决方案而言，还可以产生更为经济的设计。与 PDSP 相比，典型的 FPGA 设计采用的都是并行操作，例如：实现多重乘-累加调用效率、消除零乘积项以及流水线操作，也就是每个 LE 都有一个寄存器，这样流水线操作就不再需要额外的资源了。

在 DSP 硬件设计领域中的另一个趋势就是从图形化设计入口转向硬件描述语言(hardware description language，HDL)。尽管很多 DSP 算法可以用“信号流程图”来描述，但是现在已经发现采用基于 HDL 的设计入口，其“代码复用”大大高于图形化设计入口的“代码复用”。这就对 HDL 设计工程师提出了更高的要求，我们已经在本科生的课堂上开设了采用 HDL 进行逻辑设计的课程[2]。遗憾的是现在有两种流行的 HDL 语言。美国西海岸和亚洲倾向于采用 Verilog，而美国东海岸和欧洲则常使用 VHDL。对于用 FPGA 实现 DSP 而言，两种语言似乎都非常适用，尽管一些 VHDL 示例更容易阅读一些，这主要是因为在 IEEE VHDL 1076-1987 和 1076-1993 标准中支持有符号算法和乘/除运算。这一差距有望在新的 Verilog IEEE 1364-1999 标准获得批准之后消失，这一标准也包括有符号算法。其他的约束条件可能包括个人的偏爱、EDA 库和工具包的可用性、数据类型、可读性、性能和采用 PLI 进行语言扩展，以及商业、企业和市场因素等[3]。工具的提供商目前都支持这两种设计语言，而且这两种设计语言都适用于本书所采用的示例。

我们现在还是比较幸运的，因为不同来源的“基准”HDL 编译器基本上对于教学应用来说都是免费的。在本书中，我们就享受了这样的优惠。本书中所有给出的例子都是用 VHDL 和

Verilog 语言编写的，可以很容易移植到其他适当的设计入口系统中。Xilinx 的“基础系列”、ModelTech 的 ModelSim 编译器和 Synopys FC2 或者 FPGA 编译器都可以不需要在 VHDL 或者 Verilog 代码中做任何改动就能够运行。

本书结构是这样安排的。第 1 章首先简要介绍了当前的 FPGA 技术，用于设计的元器件和工具的 DSP 系统的技术发展水平。还给出一个有关频率合成器的详细的案例研究，包括编译步骤、仿真、性能评估、功耗估算和平面布置图。这一案例研究是后续章节中 30 多个设计示例的基础。第 2 章着眼于计算机算法方面，包括可行的 DSP FPGA 算法的数制表达方式，以及诸如加法器、乘法器或乘积和计算等的基本构造模块的实现。在这一章的结尾，还讨论了对 FPGA 非常有用的两个计算机算法概念：分布式算法(distributed arithmetic, DA)和 CORDIC 算法。第 3 章和第 4 章将研究 FIR 和 IIR 滤波器的理论和实现。我们将回顾如何确定滤波器的系数，并讨论针对规模或者速度的可能最佳实现。第 5 章涵盖了许多应用于多级数字信号处理系统的概念，例如：抽取、插值和滤波器组，在第 5 章结尾还讨论了采用双信道滤波器组实现小波处理器的多种可能性。第 6 章讨论了最重要的 DFT 和 FFT 算法的实现，主要包括 Rader、chirp-*z* 和 Goertzel DFT 算法，以及 Cooley-Tuckey、Good-Thomas 和 Winograd FFT 算法。在第 7 章介绍了更为专用的算法。与 PDSP 相比，这有可能对改进 FPGA 实现具有更大的潜力。这些算法包括数论变换、密码术算法和错误校正，以及通信系统的实现。附录包括 VHDL 和 Verilog 语言概述，以及 Verilog HDL 描述的示例。

致谢

本书是在我在达姆施塔特(译者注：德国西南部的一个城市，位于法兰克福东南)技术大学讲授了 4 年的 FPGA 通信系统设计课程、我以前的(德文)书[4, 5]，以及过去 10 年中我在达姆施塔特技术大学和位于盖恩斯维尔(译者注：美国佛罗里达州中北部的城市)的佛罗里达大学指导的 60 多篇硕士研究生论文设计的基础上编写的。在此，我要真挚地感谢所有在实验室和讨论会议上帮助我分析讨论的同仁们。这里，我还要特别感谢：M. Acheroy、D. Achilles、F. Bock、C. Burrus、D. Chester、D. Childers、J. Conway、R. Crochiere、K. Damm、B. Delguette、A. Dempster、C. Dick、P. Duhamel、A. Drolshagen、W. Endres、H. Eveking、S. Foo、R. Games、A. Garcia、O. Ghitza、B. Harvey、W. Hiberg、W. Jenkins、A. Laine、J. Mangen、J. Massey、 J. McClellan、F. Ohl、S. Orr、R. Perrt、J. Ramirez、H. Scheich、P. Vaidyanathan、M. Vetterli、H. Walter 和 J. Wietzke。

我还要感谢我的学生们，是他们花费无数小时来实现我的 FPGA 设计构想。特别是感谢：D. Abdolrahimi、E. Allmann、B. Annamaier、R. Bach、C. Brandt、M. Brauner、R. Bug、J. Burros、M. Burschel、H. Diehl、V. Dierkes、A. Dietrich、S. Dworak、W. Fieber、J. Guyot、T. Hattermann、T. Häuser、H. Hausmann、D. Herold、T. Heute、J. Hill、A. Hundt、R. Huthmann、T. Irmler、M. Katzenberger、S. Kenne、S. Kerkmann、V. Kleipa、M. Koch、T. Krüger、H. Leitel、J. Maier、A. Noll、T. Podzimek、W. Praefcke、R. Resch、M. Rösch、C. Scheerer、R. Schimpf、B. Schlanske、J. Schleichert、H. Schmitt、P. Schreiner、T. Schubert、D. Schulz、A. Schuppert、O. Six、O. Spiess、O. Tamm、W. Trautmann、S. Ullrich、R. Watzel、H. Wech、S. Wolf、T. Wolf 和 F. Zahn。

此外，还要特别感谢我的妻子 Anke Meyer-Baese 博士和佛罗里达大学(位于盖恩斯维尔)的 J. Harris 博士和 Fred. Taylor 博士以及 Springer 的 Paul. DeGroot。是他们帮助我完成了英文修订本。

还要感谢 DAAD、DFG、欧洲航天局和 Max Kade Foundation 大力提供了经济支持。

如果您在阅读本书的过程中发现任何错误或者是有任何改进本书的建议，请您通过 Uwe.Meyer-Baese@ieee.org 或者是通过出版商与我联系。

Uwe Meyer-baese

塔拉哈西市　2001 年 5 月

目　　录

第1章 绪 论

本章概述将要在本书中研究的算法和技术。首先简要介绍一下数字信号处理技术，然后重点讨论 FPGA 技术。最后研究 Altera EPF10K20 芯片和一个包括芯片合成、时序分析、平面布置图和功耗分析的较大规模设计示例。

1.1 数字信号处理(DSP)概述

长期以来，信号处理技术一直用于转换或产生模拟或数字信号。其中最为频繁应用的领域就是信号的滤波，我们将在第 3 章和第 4 章讨论这一问题。此外，从数字通信、语音、音频和生物医学信号处理到检测仪器仪表和机器人技术等许多领域中，都广泛地应用了数字信号处理技术。表 1-1 给出了 DSP 技术的一些应用概况[6]。

表 1-1 数字信号处理的应用

应用领域	DSP 算法
通用领域	滤波和卷积、自适应滤波、检测和校准、谱估计和傅立叶变换
语音处理	编码和解码、加密和解密、语音识别和合成、扬声器识别、回波消除、人造耳蜗植入的信号处理
音频处理	hi-fi 编码和解码、噪声消除、音频平衡、环境声学仿真、混频和编辑、声音合成
图像处理	压缩和解压缩、旋转、图像传输与分解、图像识别、图像增强、人造视网膜植入的信号处理
信息系统	语音信箱、传真、调制解调器、蜂窝移动电话、调制/解调、线路均衡器、数据加密和解密、数字通信和局域网、延拓频谱技术、无线局域网、广播和电视、生物医学信号处理
控制	伺服控制、磁盘控制、打印机控制、发动机控制、定向和导航、振动控制、动力系统监控器、自动化仪器仪表
仪表设备	波束成型、波形发生器、瞬态分析、稳态分析、科学仪器设备、雷达和声纳

数字信号处理(digital signal processing, DSP)已经发展成为一项成熟的技术，并且在许多应用领域逐步代替了传统的模拟信号处理系统。DSP 系统具有几项优势，例如：元器件对温度变化、老化以及对容许偏差的不敏感性。在过去，模拟芯片设计可以生产出越来越小的小片尺寸，可是发展到今天，随着现代亚微米设计所带来的噪声，使得数字设计在集成度方面可以比模拟

设计做得更好。这些产品就是紧凑的、低功耗并且是低成本的数字设计。

有两个事件加速了 DSP 技术的发展。其一是 Cooley 和 Tuckey(1965 年)对一种计算离散傅立叶变换(Discrete Fourier Transform, DFT)的有效算法的解密。我们将在第 6 章详细讨论这类算法。另一个里程碑就是可编程数字信号处理器(programmable digital signal processor, PDSP)在 20 世纪 70 年代后期的引入。这种 PDSP 能够在仅仅一个时钟周期内完成(定点数)“乘-累加”的计算，与同一时代“冯·诺伊曼(Von Neuman)”式微处理器为基础的系统相比较而言，有着本质上的改进。现代的 PDSP 可以包含更为复杂的功能，例如：浮点数乘法器、筒状移位器、存储体以及零架空的 A/D 和 D/A 转换器接口。EDN 每年都要出版一份有关可用的 PDSP 的详细综述[7]。图 1-1 给出了一个依靠 PDSP 来实现模拟系统的典型应用。在研究了 FPGA 的体系结构之后，我们将在 1.2.1 节和第 2 章中返回头来研究 PDSP。

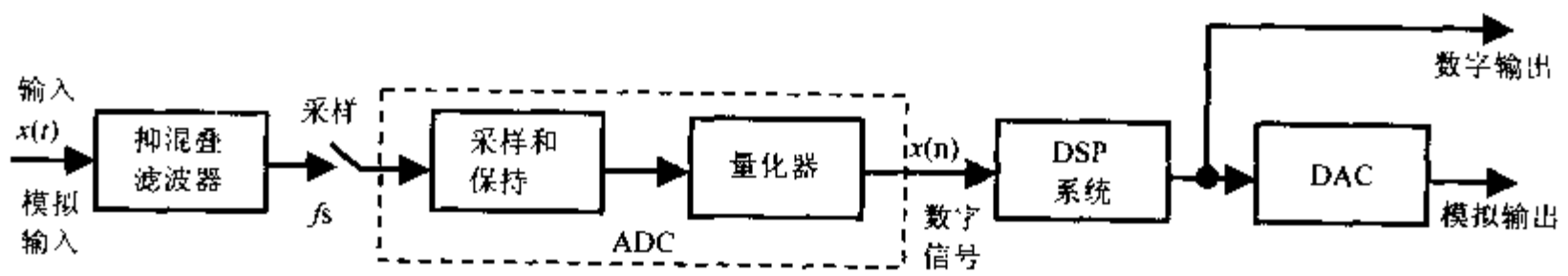


图 1-1　一个典型的 DSP 应用示例

1.2 FPGA 技术

VLSI(Very Large Scale Integration，超大规模集成)电路可以如图 1-2 所示进行分类。FPGA 是一类称为现场可编程逻辑(field-programmable logic, FPL)器件中的一员。FPL 被定义为含有现场可反复使用的小规模逻辑模块和单元的可编程器件[1]。鉴于 FPGA 是特定用途的集成电路，所以 FPGA 被认为是一种专用集成电路(application specific integrated circuit, ASIC)技术。但是，通常设计 ASIC 类电路需要额外的半导体处理步骤，而 FPL 是不需要这些步骤的。这些额外步骤能够提供更高级别的、更高性能的 ASIC，但同时也增加了不可重复的工程成本(non-reoccurring engineering, NRE)。另一方面，门阵列通常是由“与非门之海”构成，后者的功能是用户在“网络表”中提供的。在整个制造过程中都要使用这一网络表，以便获得最终金属层明显的清晰度。但是，可编程门阵列解决方案的设计者可以完全控制设计的实现过程，而不需要任何实际的集成电路制造设备或者因为后者而延缓设计进度。

注 1：Xilinx 称之为可配置的逻辑模块(configurable logic block, CLB)，而 Altera 称之为逻辑单元(logic cell, LC)或者逻辑元件(logic element, LE)。

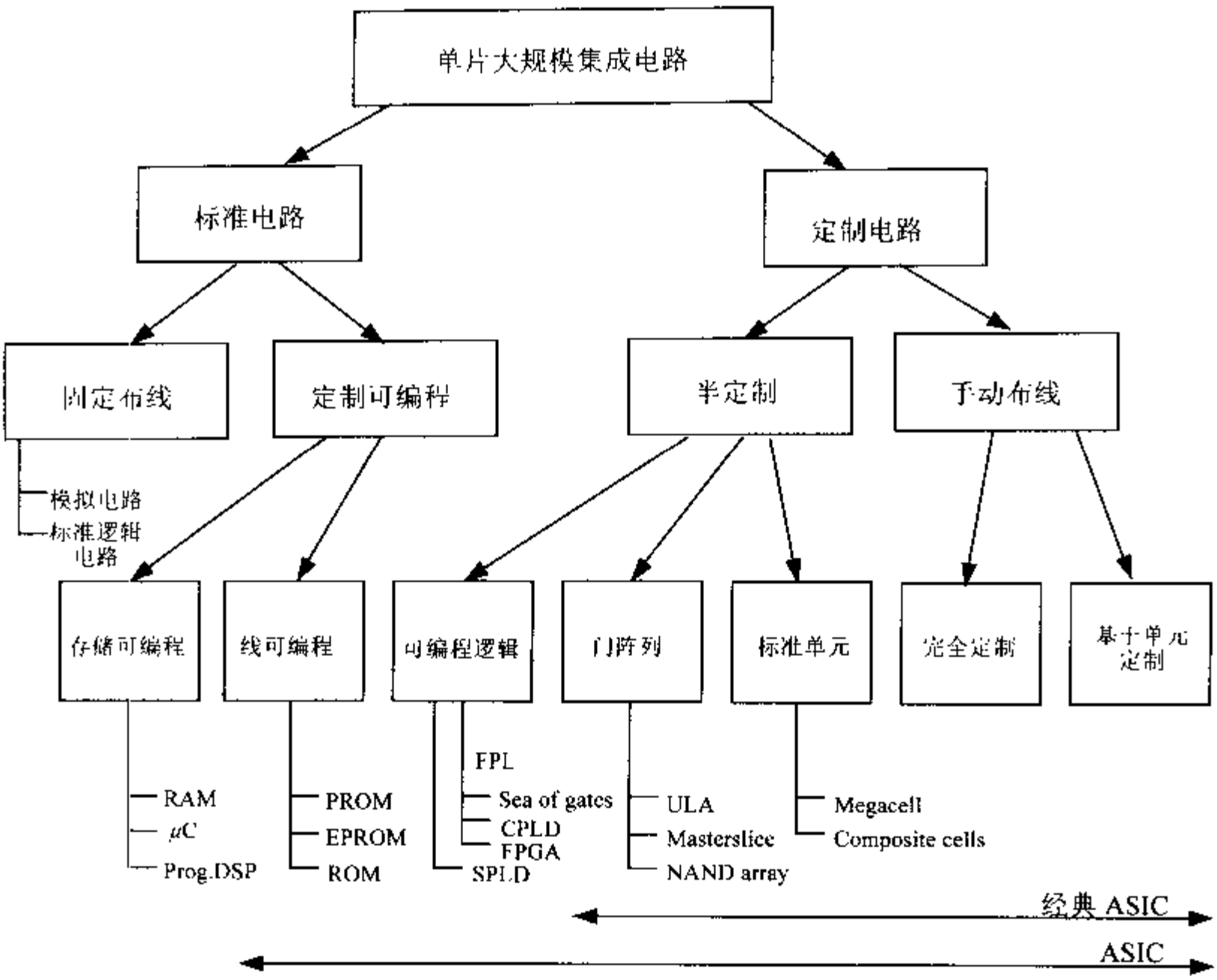


图 1-2　VLSI 电路的分类(©1995 年，VDI 出版社[4])

1.2.1　按颗粒度分类

逻辑模块规模与元器件的颗粒度相关，而元器件的颗粒度又与模块之间需要完成的布线(路由通道)工作量相关。3 种常见的不同颗粒度分类如下：

- 小颗粒度(Pilkington 或者“门海(sea of gates)”结构)
- 中等颗粒度(FPGA)
- 大颗粒度(CPLD)

1. 小颗粒度元器件

由 Pilkington 半导体公司供应的小颗粒度元器件最初得到 Plessey 公司的认可，然后是 Motorola 公司的认可。基本逻辑单元包括一个单一与非门和一个锁存器(请参阅图 1-3)。由于采用与非门可以实现任何二进制逻辑函数(请参阅练习 1.1)，所以与非门被称为通用函数。这一技术连同已经被认可的逻辑合成工具(例如 ESPRESSO)一起，还应用于门阵列的设计之中。在门阵列的与非门之间布线是采用额外的金属层来实现的。但对于可编程的结构来讲，这就成了一个瓶颈，因为与已经实现的逻辑函数相比，它对布线资源的利用率非常高。此外，构建一个简单的 DSP 对象就需要大量的与非门。例如：一个高速 4 位加法器就要用掉大约 130 个与非门。这使得小颗粒度技术在实现大多数 DSP 算法时并没有什么吸引力。

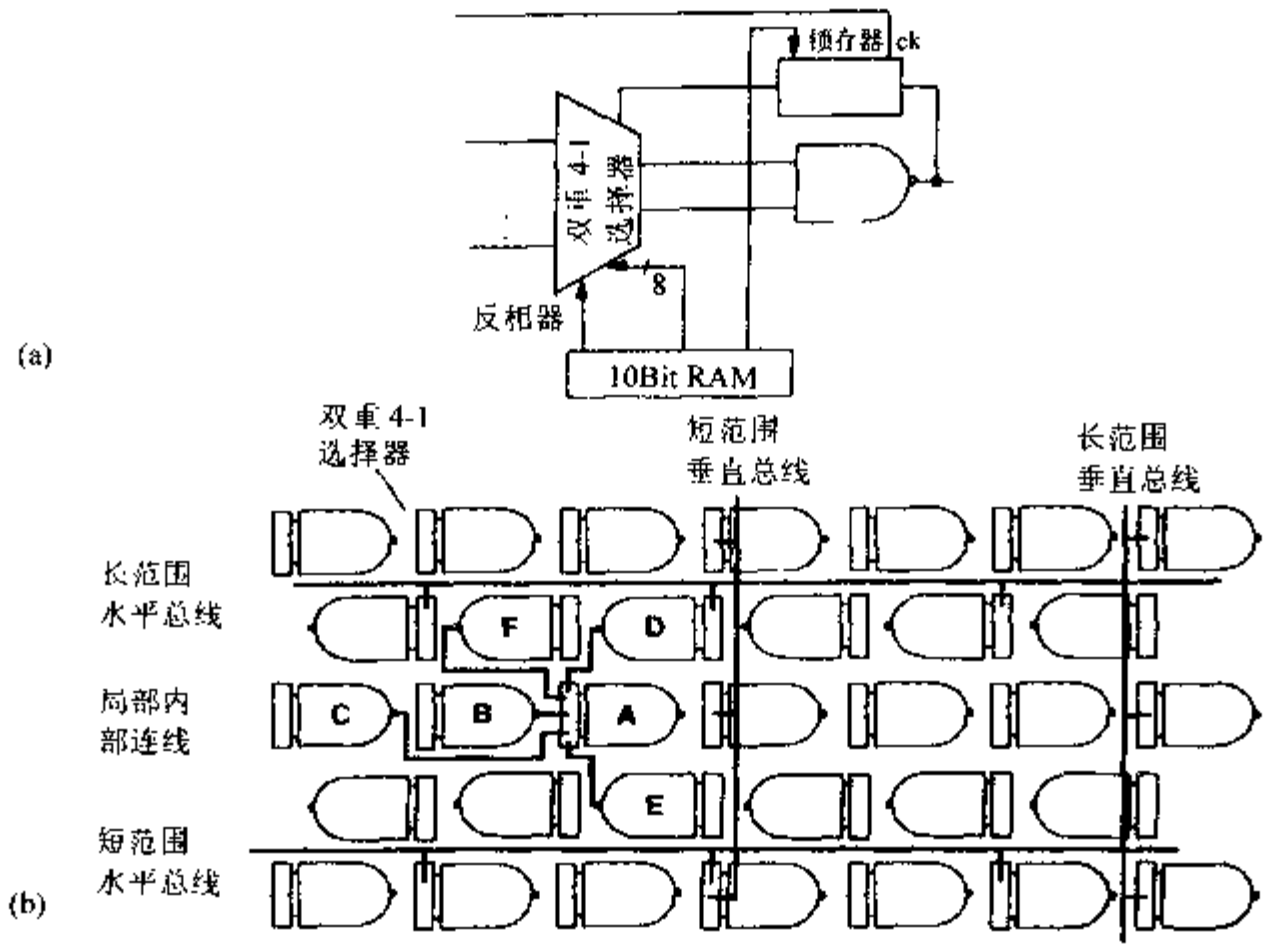


图 1-3　具有 10K 个与非逻辑模块的 Plessey ERA60100 结构[8]

(a) 基本逻辑模块　(b) 布线结构(© 1990 Plessey)

2. 中等颗粒度元器件

最为常见的 FPGA 结构如图 1-4(a)所示。图 1-5 给出了一个当前中等颗粒度 FPGA 元器件的具体示例。具有代表性的基本逻辑模块是小规模的表(例如：Xilinx 的 XC2k-4k，具有 4 位到 5 位的输入表，1 位或者 2 位的输出)或者由专用的多路复用器(multiplexer, MPX)逻辑来实现，例如：在 Actel 的 ACT-2 元器件中的所使用的 MPX[9]。布线通道的选择范围是从短到长。带有触发器的可编程 I/O 模块就附在元器件的物理边缘。

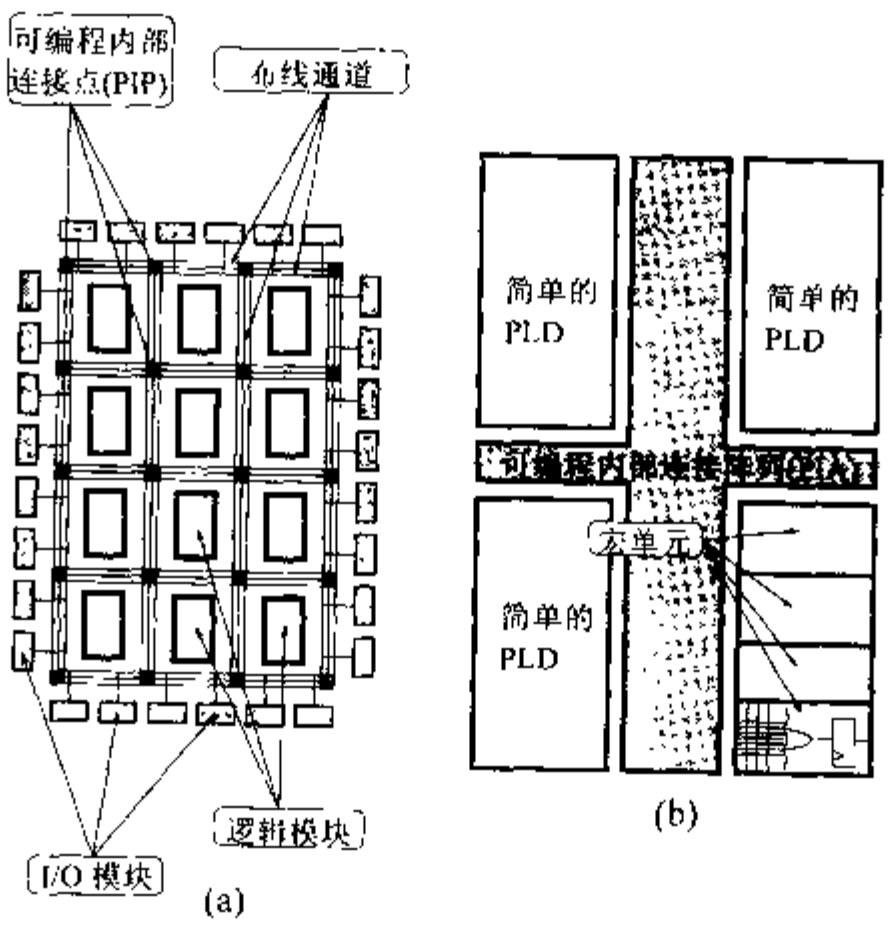


图 1-4　(a) FPGA 和(b) CPLD 的结构(©1995 年，VDI 出版社[4])

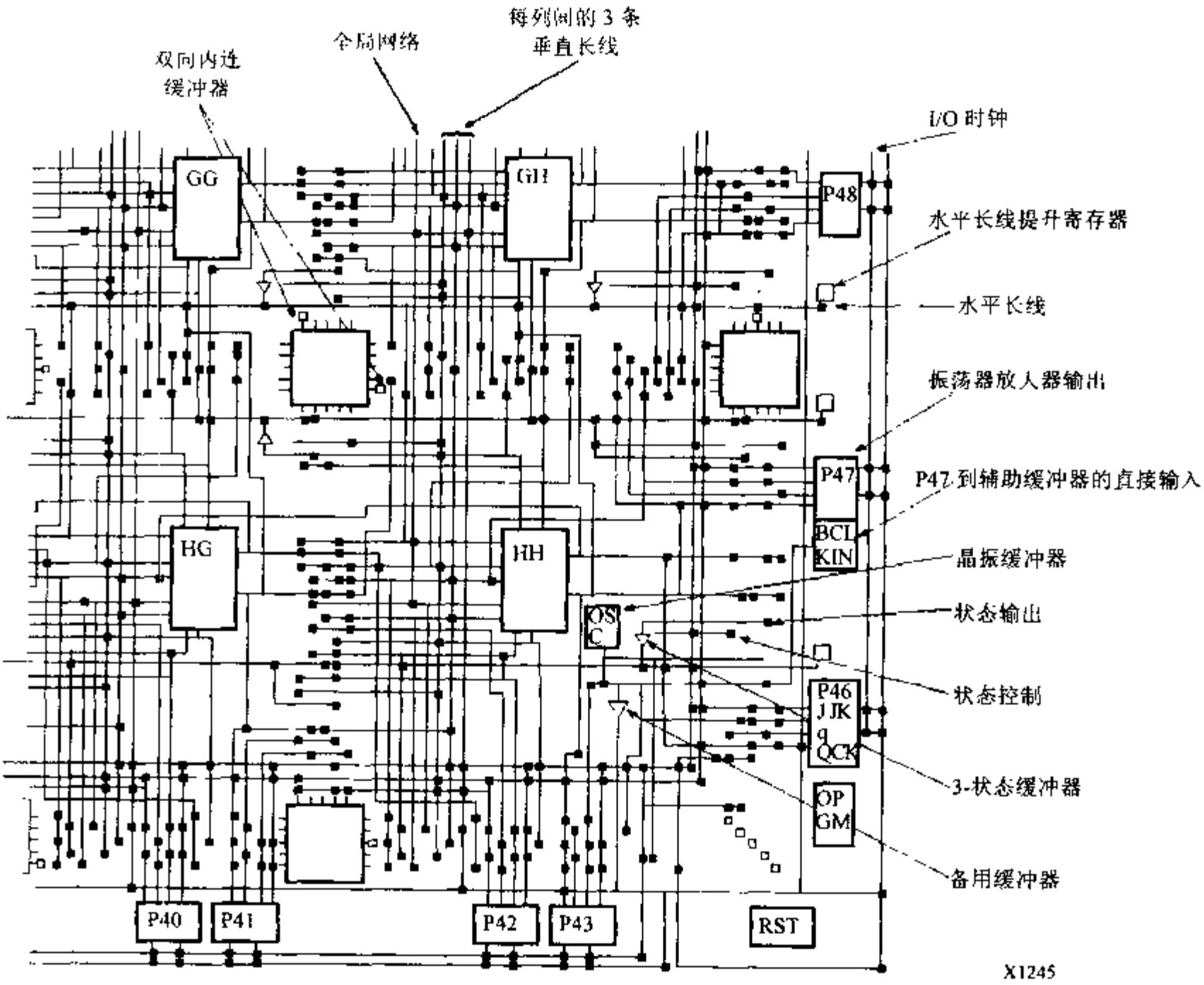


图 1-5　中等颗粒度元器件的示例(©1993 年，Xilinx)

3. 大颗粒度元器件

在图 1-4(b)中给出了大颗粒度元器件的特性，诸如复杂的可编程逻辑元器件(complex programmable logic devices, CPLD)。这些复杂的可编程逻辑元器件(CPLD)可以定义成是由简单可编程逻辑元器件(simple programmable logic devices, SPLD)组合而成的，例如：如图 1-6 所示的传统 GAL16V8 芯片。这类 SPLD 芯片由一个充当与/非阵列的可编程逻辑阵列和一个通用 I/O 逻辑模块组成。通常，CPLD 中的 SPLD 具有 8 到 10 个输入端，3 到 4 个输出端，并且支持大约 20 个乘积项。在这些 SPLD 模块之间的宽带总线(Altera 称之为可编程内连阵列，(programmable interconnect arrays, PIAs))上有可能存在短暂的延迟。通过将总线与固定的 SPLD 时限结合起来就能够提供与 CPLD 之间可预先计算的短暂的管脚到管脚之间的延迟。

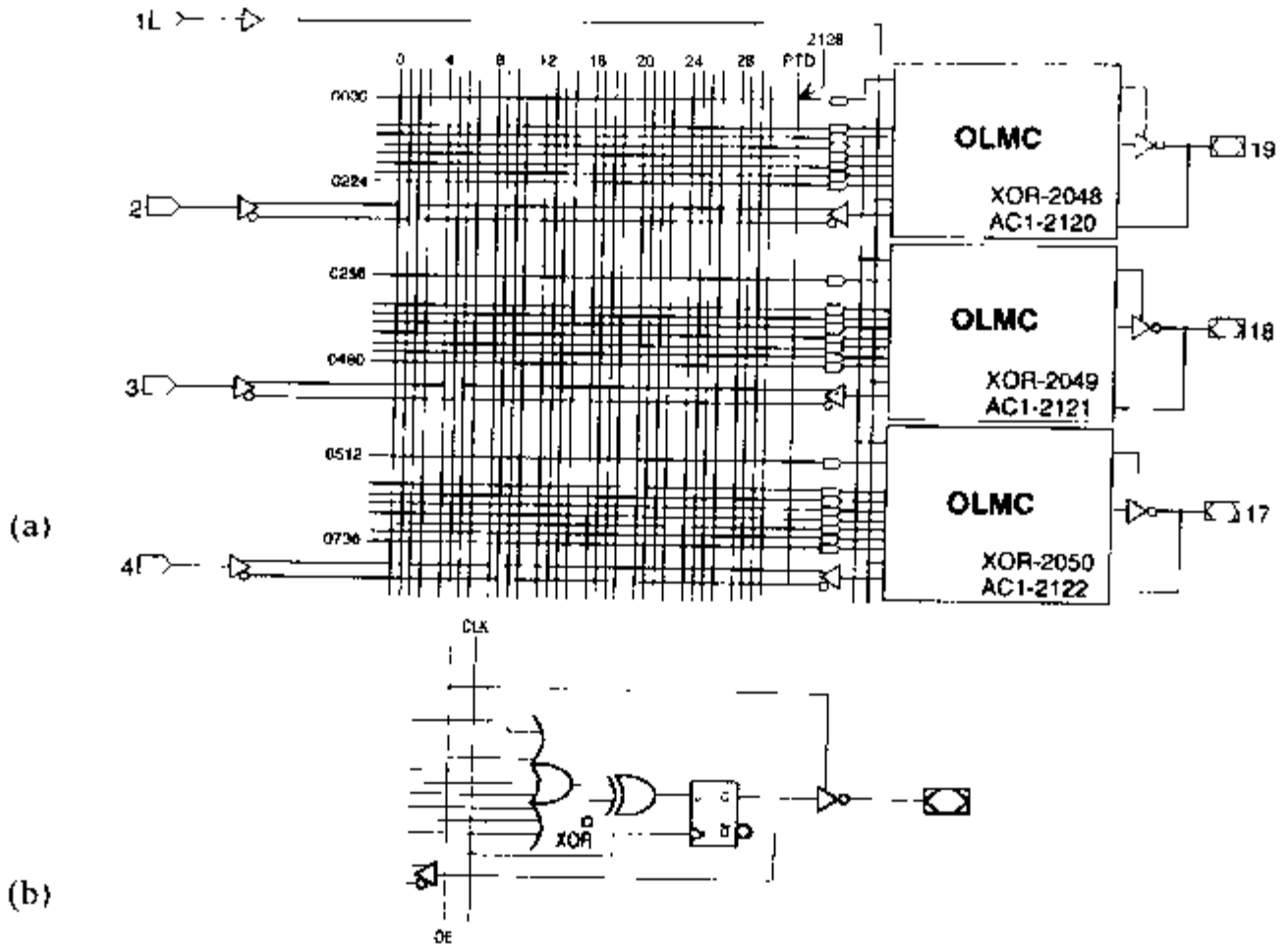


图 1-6　GAL16V8 (a) 8 个宏单元中的前 3 个　(b) 输出逻辑宏单元(OLMC)(©1997 年，Lattice)

1.2.2　按技术分类

实际上，FPL 在所有存储技术(SRAM、EPROM、E^2PROM 和抑熔断技术[10])中都是可行的。根据具体的技术可以将元器件定义为是可重复编程的还是一次性编程的。大多数 SRAM 元器件都可以通过一位流来编程，从而降低了对布线的要求，但是也相应地增加了编程的时间(通常情况下，范围是毫秒(ms)级的)。对于 FPGA 来说，具有优势地位的 SRAM 元器件是基于静态 CMOS 存储技术的，并且是系统内编程和可重复编程的。然而它们还需要一个外部“引导”器件作为组态。由于电可编程只读存储器(electrically programmable read-only memory, EPROM)需要用紫外线照射来擦除，所以经常用在一次性 CMOS 可编程模式中。CMOS 电可擦可编程只读存储器(electrically erasable programmable read-only memory, E^2PROM)可以用作可重复编程和系统内编程器件。EPROM 和 E^2PROM 都具有设置时间短的优势。因为编程信息不是下载(downloaded)到元器件上的，所以能够得到更好的保护，禁止未授权的使用。近来出现了一项基于 EPROM 的技术，称为**闪速存储器**(flash memory)的革新。这类元器件通常被视作是“页方式”的、具有更小的单元，系统内可重复的编程系统，等同于 E^2PROM 元器件。最后，在表 1-2 中简要地给出了不同元器件技术的主要优点和缺点。

表 1-2　FPL 技 术

技　术	SRAM	EPROM	E^2PROM	抑 熔 断	Flash
可重复编程	√	√	√	—	√
系统内可编程	√	—	√	—	√
易失性	√	·	—	—	—
复制保护	—	√	√	√	√

(续表)

技 术	SRAM	EPROM	E^2PROM	抑 熔 断	Flash
产品示例	Xilinx XC4K Altera Flex	Altera MAX5K Xilinx XC7K	AMD MACH Altera MAX 9K	Actel ACT	Xilinx XC9500 Cypress Ultra 37K

1.2.3 FPL 的基准

为 FPL 元器件提供客观的标准可不是一项简单任务。通常要根据设计者的经验和技巧以及设计工具的特点来预测其性能。为了确定一个有效基准，Xilinx[11]、Altera[12]和 Actel[13]共同建立了可编程电子产品性能协议(Programmable Electronic Performance Cooperative, PREP)，到目前为止已经有 10 多个成员。PREP 已经为 FPL 开发了 9 种不同的基准，我们在表 1-3 中总结出了这些标准。强调基准的中心思想就是每个厂商利用自己的元器件和软件工具在规定的元器件中尽可能多次实现简单的模块，同时尽可能地提高速度。一个元器件中相同逻辑模块的实例数量称为重复率(repetition)，这是所有基准的基础。对照 DSP 而言，表 1-3 中的基准 5 和 6 是相关联的。在图 1-7 中按照相对于频率的形式给出了 Actel(A_k)、Altera(o_k)和 Xilinx(x_k)的典型元器件的重复率。由此可以得出结论：现代的 FPGA 系列提供了最佳的 DSP 复杂度和最高速度。这要归结于现代的元器件提供了允许快速进位逻辑延迟(每比特小于 0.5 纳秒)的能力(请参阅 1.4.1 节)，它不需要昂贵的“超前进位”译码器就可以提供多位宽的快速加法电路。尽管 PREP 基准对于比较等效门数量和提高速度是有益的，但对于具体的应用而言，其他的属性也是非常重要的。这些属性包括：

- 芯片内的 RAM 或 ROM
- 管脚到管脚之间的延迟
- 内部三态总线
- 读回或边界扫描译码器
- 可编程电压变化速度或者 I/O 的电压
- 功耗

表 1-3 FPL 的 PREP 基准

编 号	基 准 名 称	说 明
1	数据信道	8 个 4 对 1 乘法器驱动一个并行负载的 8 位移位寄存器
2	定时计数器	通过 8 位数值寄存器对两个 8 位的数值进行定时和比较
3	小型状态机	具有 8 个输入和 8 个输出的 8-状态机
4	大型状态机	具有 40 个转换、8 个输入和 8 个输出的 16-状态机
5	算法	4×4 无符号乘法器；9 位累加器
6	累加器	16 位累加器
7	加法计数器	16 位可载入的二进制加法计数器
8	降值计数器	16 位可载入的二进制减法计数器
9	存储器映射	解码地址空间范围从 4KB 到 1KB 的映射

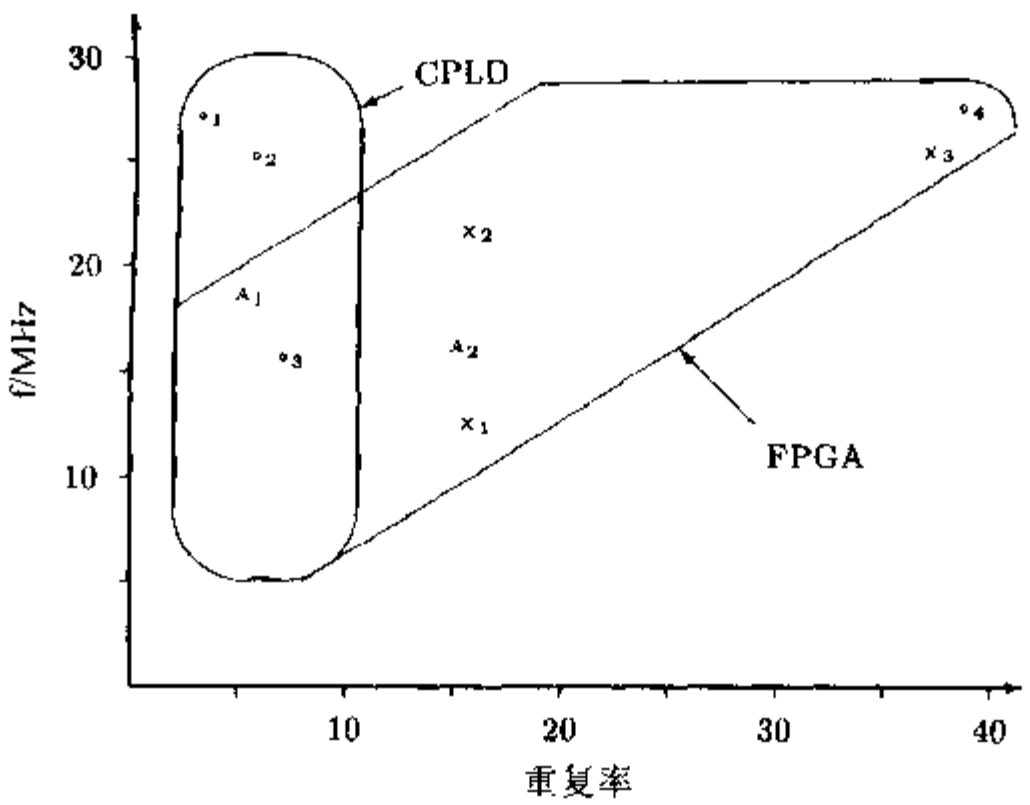


图 1-7　FPL 的基准(©1995 年，VDI 出版社[41])

图 1-8 总结了一些典型 FPL 元器件的功耗。从中可以看到 CPLD(Altera)通常具有较高的“备用”功耗，在更高频率的应用场合下，预期 FPGA(Xilinx 和 Actel)的功耗会更高一些。在 1.4.2 节可以找到更为详细的功率分析示例。

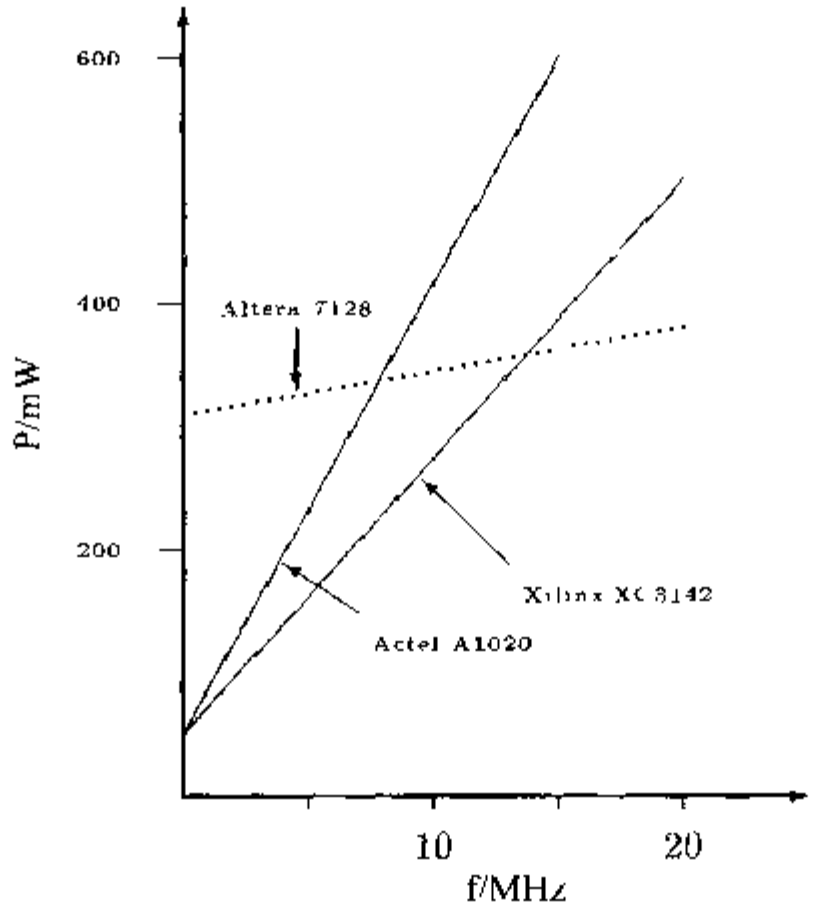


图 1-8　FPL 的功耗(©1995 年，VDI 出版社[41])

1.3　DSP 的技术要求

图 1-9 给出了厂商提供的 PLD 市场占有份额。PLD 自从 20 世纪 80 年代初期问世以来，已经获得了每年稳定增长 20%的喜人景象，超过 ASIC 增长速度 10%以上。原因似乎与 FPL 可以

提供许多和 ASIC 一样的优点这一事实相关，例如：

- 在尺寸、重量和功耗方面都有所降低
- 更高的吞吐量
- 更好的安全性能可以禁止未授权的复制
- 减少了元器件本身和开发的成本
- 降低了线路板的测试成本

而且还克服了 ASIC 的许多缺点：

- 缩短开发时间 3～4 倍(快速原形设计)
- 在线可重复编程的能力
- 在少于 1 000 个单元的解决方案中降低了 NRE 成本，从而可以得到更为经济的设计

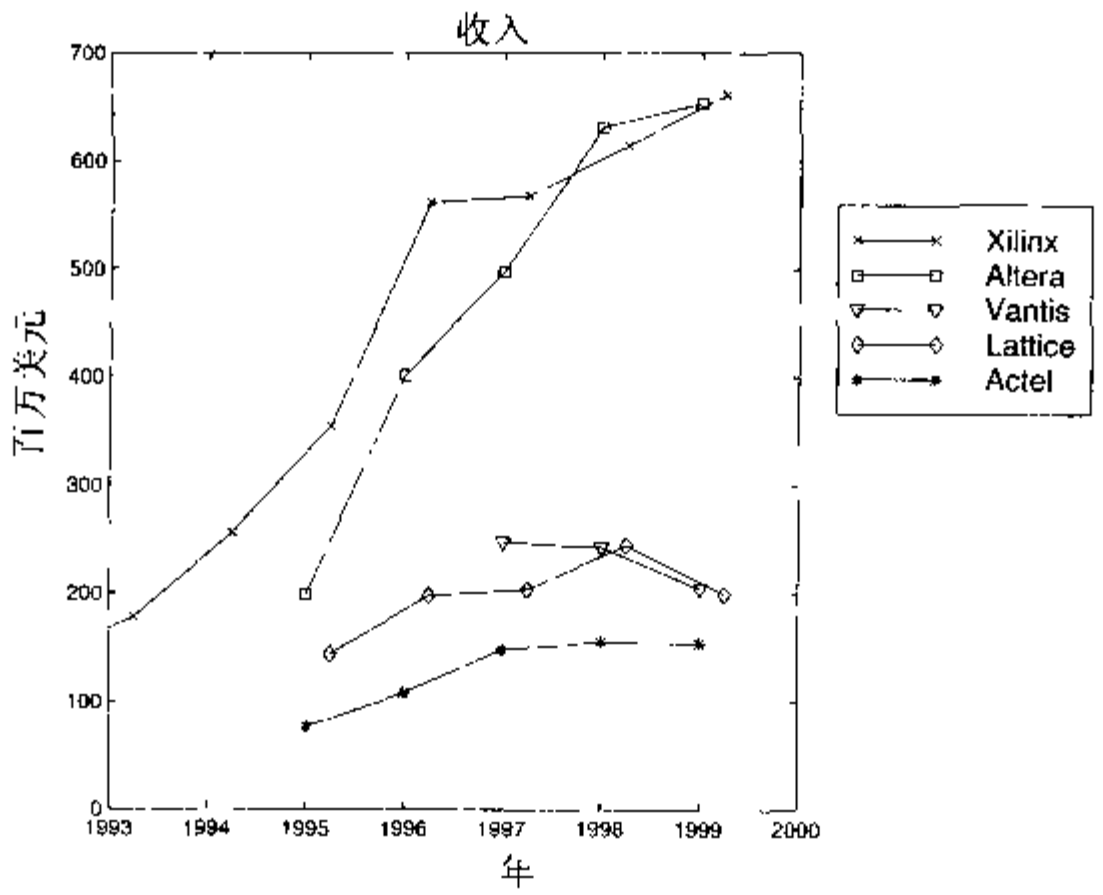


图 1-9　5 家主要厂商在 PLD/FPGA/CPLD 市场的收入

CBIC ASIC 过去应用在高端大批量(多于 1 000 个副本)的应用场合。与 FPL 相比较，典型的 CBIC ASIC 在同样大小的小片尺寸上含有 10 倍以上的门。试图解决第二个问题的方案称作**硬布线的 FPGA**，其中门阵列用来实现一个已经确定的 FPGA 设计。

FPGA 和可编程信号处理器

通用的可编程数字信号处理器(programmable digital signal processor, PDSP)[14, 15, 6] 在近 20 年里已经取得了巨大的成功。这些 PDSP 都是基于一种精简指令集计算机(reduced instruction set computer, RISC)的范例，由至少一个快速阵列乘法器(例如：16×16 位到 24×24 位定点数、或者是 32 位浮点数)和一个扩展字宽的累加器构成。PDSP 的优势源于大多数信号处理算法的乘-累加运算(multiply and accumulate, MAC)都是非常密集的，通过多级流水线结构，PDSP 可以获得仅仅受阵列乘法器的速度限制的 MAC 速度。由此可以认为 FPGA 也能够用来实现 MAC 单元[16]，但是，如果 PDSP 能够满足所需要的 MAC 速度，那么 PDSP 在成本问题上通常更具有优势。另一方面，现在我们还发现了许多高带宽的信号处理应用领域，例如：无线电、多媒体

或卫星通信，FPGA 技术可以通过一个芯片上的多级 MAC 单元来提供更多的带宽。此外，在几种后续要讨论的诸如 CORDIC、NTT 和错误校正算法等算法中，还可以进一步证明 FPL 技术要比 PDSP 更有效率。据称，在未来，PDSP 将会主宰需要复杂算法的应用领域(例如：多重 if-then-else 结构)，而 FPGA 将会统治更多前端(传感器)的应用，例如 FIR 滤波，CORDIC 算法或 FFT，而这些内容就是本书要介绍的主旨。

1.4 设计实现

通常在 VLSI 设计中，其详细程度的层次可以涉及从完全定制的 ASIC 几何布局到使用称为装置顶盒的系统设计。表 1-4 给出了相应的概述。因为 FPGA 的物理结构是可编程的，但也是固定的，所以在 FPGA 的设计过程中没有布局和布线任务。在门电平上采用寄存器转换设计语言就可以达到元器件的最佳利用。投放市场时间与 FPGA 迅速增加的复杂程度共同迫使研究方法转移到知识产权(Intellectual Property, IP)宏单元或“兆核心单元(mega core cells)”的使用上来。宏单元为设计者提供了一个预先定义的功能集合，例如：微处理器或 UART。这样，设计者就只需要指定所选择的特征或属性(例如：精确度)，“合成器”就会自动生成一份合成解决方案的硬件描述代码或者是原理图。

表 1-4 VLSI 设计层次

对 象	目 标	示 例
系统	性能规范说明	计算机、磁盘设备、雷达
芯片	算法	μp、RAM、ROM、UART、并行端口
寄存器	数据流	寄存器、ALU、COUNTER、MUX
门电路	布尔方程	AND(与)、OR(或)、XOR(异或)、FF(触发器)
线路	微分方程	晶体管、R(电阻)、L(电感)、C(电容)
布局图	无	几何形状

FPGA 技术的关键就是利用强有力的设计工具来：

- 缩短开发周期
- 提供元器件的优质利用性
- 提供合成器的选择，例如：在最佳速度和设计规模之间作出选择

在图 1-10 中给出了 CAE 工具分类法，并且应用在 FPGA 的设计流程之中。通常，决定是在图形界面还是文本界面环境下进行设计，要看设计者个人的偏好以及以前的经验。以图形方式给出 DSP 解决方案能够强调更为规则的数据流以及许多相关的 DSP 算法。而文本环境通常侧重于考虑算法控制设计，并提供更加宽泛的设计类型，正如将在后续设计示例中得到的证明一样。具体地说，对于 Altera 的 MaxPlusII 而言，据说利用文本能够设计一些可以分配到设计之中的更为特殊的属性和更为精确的性能。

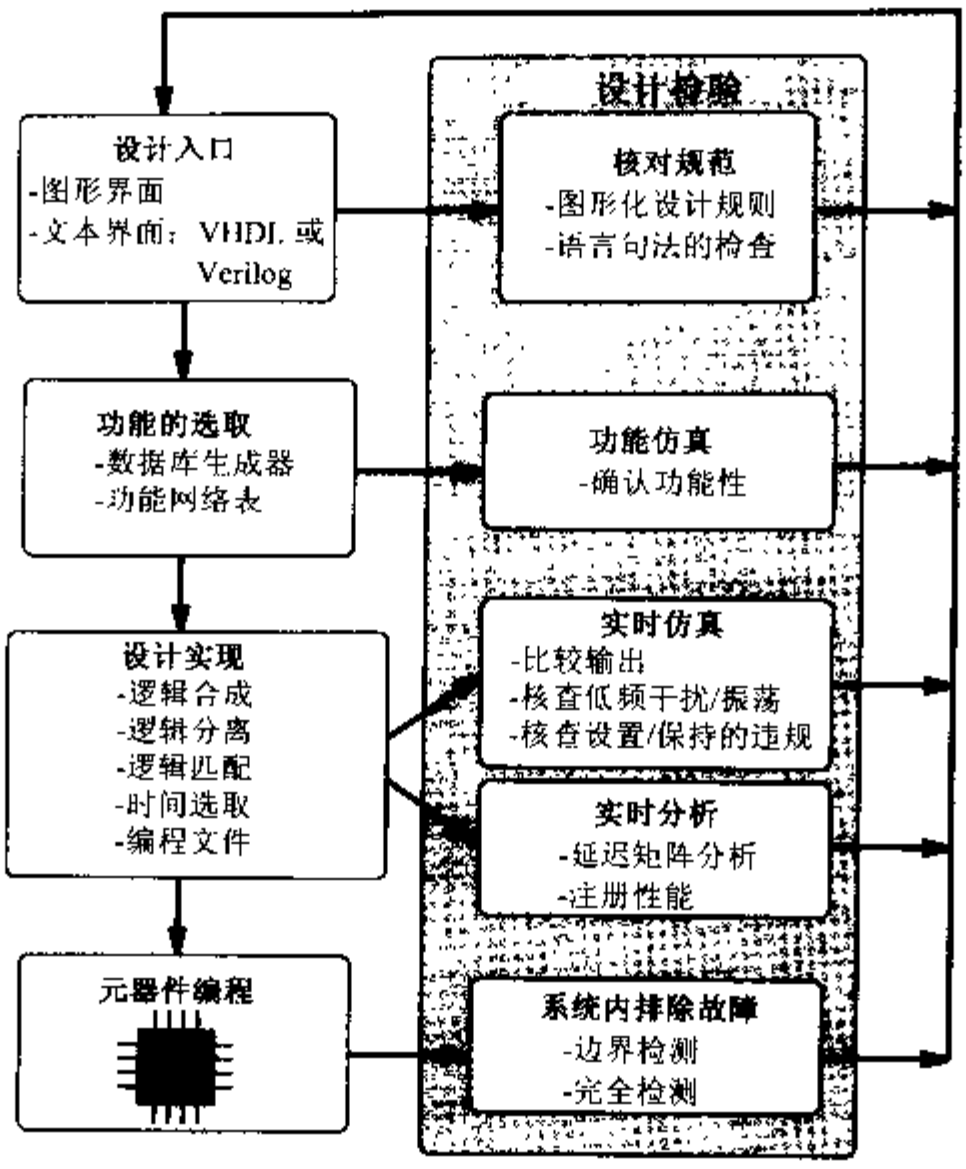


图 1-10　CAD 设计周期

例 1.1：VHDL 设计类型的比较

下面的设计示例阐述了 VHDL 环境下的 3 种设计策略。具体内容如下：

- 组件实例(结构类型，例如：图形化网络表设计)
- 数据流
- 利用 PROCESS 模板(也就是性能类型)进行顺序设计

VHDL 设计文件 example.vhd[2] 如下(--以后是注释)：

```
PACKAGE eight_bit_int IS       -- User defined type
  SUBTYPE BYTE IS INTEGER RANGE  - 128 TO 127;
END eight_bit_int;

LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY lpm;                              -- Using predefined packages
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

注 2：本例相应的 Verilog 代码文件 example.v 可以在附录 A 中找到。

```
USE ieee.std_logic_arith.ALL;

ENTITY example IS                                    ------> Interface
  GENERIC (WIDTH : INTEGER := 8);     -- Bit width
  PORT (clk   :   IN STD_LOGIC;
        a, b  :   IN BYTE;
        op1   :   IN STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        sum   :   OUT STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
        d     :   OUT BYTE);
END example;

ARCHITECTURE flex OF example IS

  SIGNAL   c, s          :   BYTE;          -- Auxiliary variables
  SIGNAL   op2, op3      :   STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);

BEGIN

  -- Conversion int -> logic vector
  op2 <= CONV_STD_LOGIC_VECTOR(b,8);

  add1: lpm_add_sub              ------> Component instantiation
    GENERIC MAP (LPM_WIDTH => WIDTH,
                 LPM_REPRESENTATION => "SIGNED",
                 LPM_DIRECTION => "ADD")
    PORT MAP (dataa => op1,
              datab => op2,
              result => op3);
  reg1: lpm_ff
    GENERIC MAP (LPM_WIDTH => WIDTH )
    PORT MAP (data => op3,
              q => sum,
              clock => clk);

  c <= a   + b ;                          ------> Data flow style

  p1: PROCESS                             ------> Behavioral style
  BEGIN
    WAIT UNTIL clk = '1';
    s <= c + s;                 ----> Signal assignment statement
  END PROCESS p1;
  d <= s;

END flex;
```

只有在设计的功能仿真(在 MaxPlusII 编译器模式下，选择选项 Processing | Functional SNF Extractor)成功之后，我们才可以开始图 1-10 中所示的设计实现。在 MaxPlusII 编译器模式下，选择选项 Processing | Timing SNF Extractor，接下来就可以看到编译窗口中出现了 3 个额外的词条，分别是 Logic Synthesizer, Fitter 和 Timing SNF Extractor。在启动编译器后就可以对设计进行同步仿真、核查误操作或测定其 Registered Performance(在此仅仅给出了几个选项)。在上述步骤均成功后，如果有硬件线路板(像 Altera 教学线路板)的话，就可以如图 1-10 所示对元器件进行编程，并采用“读回”的方法完成剩余的硬件测试。

1.4.1　FPGA 的结构

在 21 世纪初，有两个系列的 FPGA 元器件似乎拥有最具吸引力的实现 DSP 算法的特性，这是因为这些 FPGA 具有快速进位逻辑的能力，从而能够以超过 50MHz 的速度实现 32 位(非流水线)的加法[1, 18, 19]。

这两个系列就是 Xilinx XC4000 系列和 Altera FLEX 10K 系列元器件。后者是 Altera 的 8K 元器件再加上额外的称作嵌入式阵列模块(embedded array block, EAB)的 2KB RAM 模块。Xilinx 元器件具有 FPGA 中典型的宽泛的路由选择级，而 Altera 元器件则是基于 Altera 的 CPLD 中使用的宽带总线结构，但是 FLEX 10K 的基础模块已经不再是 CPLD 中大规模的 PLA。现在取而代之的是 FPGA 典型的中等颗粒度器件，例如：小规模的查询表(small look-up tables, LUT)。

Xilinx XC4000 系列的基本逻辑单元称作可配置逻辑模块(configurable logic block, CLB)，具有两个独立的 4 输入 1 输出的 LUT 和快速进位，另外一个 3 输入 1 输出的 LUT 将两个独立的 LUT 和两个触发器如图 1-11 所示连接起来。Xilinx 元器件具有 5 层路由，从 CLB 到 CLB，再到跨过整个芯片的长线。每一个 CLB 都可以用作 16×2 或 32×1 位的 RAM 或 ROM。表 1-5 列出了 Xilinx XC4000 系列的部分元器件。

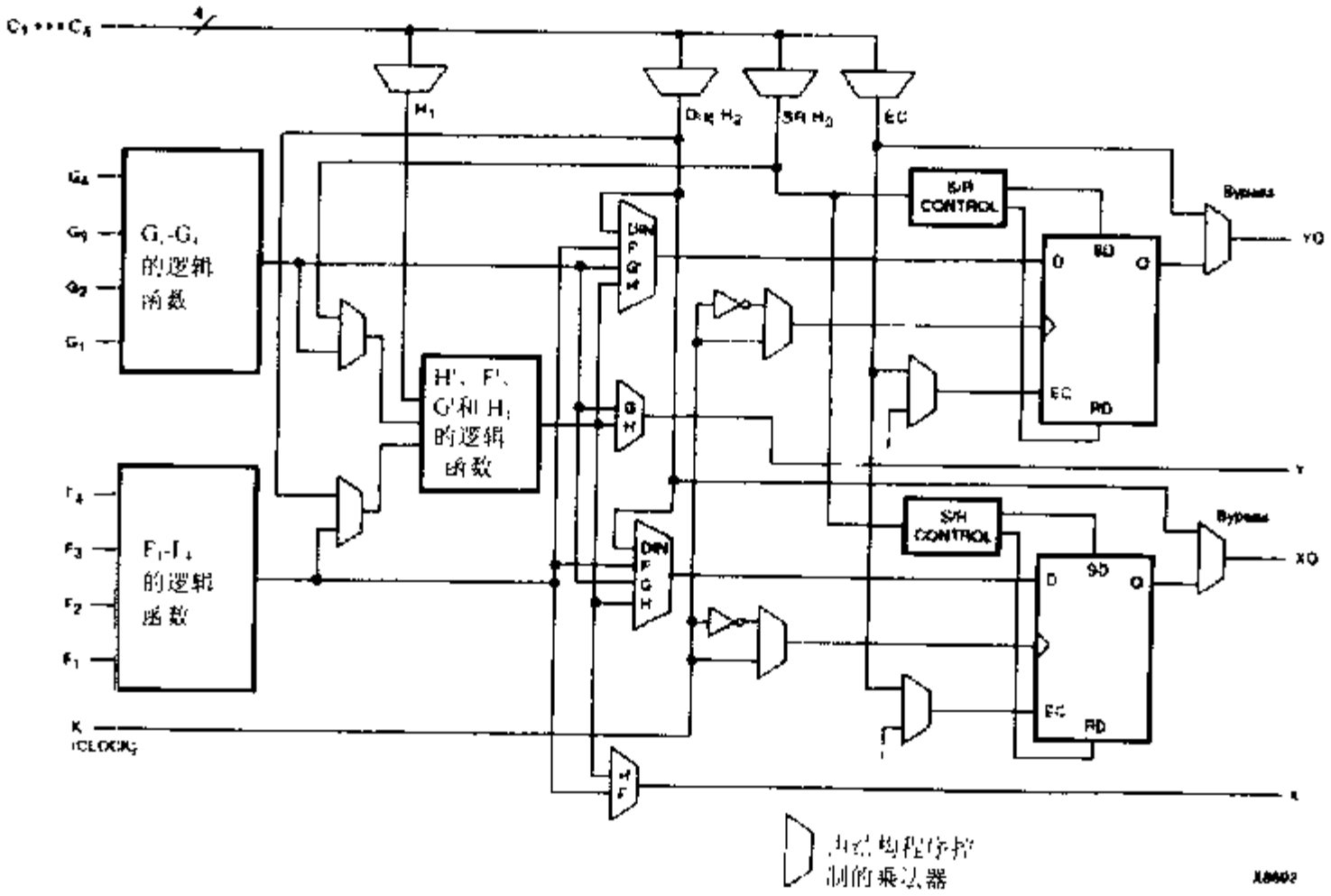


图 1-11　XC4000 逻辑单元(©1993 年，Xilinx)

表 1-5　Xilinx XC4000 系列

元　器　件	CLB 总　量	触发器位数	最大 RAM KB	最大 I/O
XC4003	100	360	3.2	80
XC4005	196	616	6.3	112
XC4010	400	1120	12.8	160
XC4025	1024	2560	32	256
XC4085	3136	7168	100	448
XC40150	5184	11520	165	448
XC40250	8464	18400	270	448

Altera FLEX 10K 元器件的基本逻辑模块使用小规模 LUT 实现了中等颗粒度。10K 元器件是在 Altera 8K 元器件的基础上再加上被称为嵌入式阵列模块(embedded array block, EAB)的 2KB RAM 模块。Altera FLEX 10K 元器件中的基本逻辑模块称作逻辑元件(logic element, LE)[3]，如图 1-12 所示，包括一个触发器、一个 4 输入 1 输出的 LUT，或一个 3 输入 1 输出的 LUT 和一个快速进位或者与/非乘积项扩展电路。8 个 LC 组成一个逻辑阵列模块(logic array block, LAB)。每一排包括一个嵌入式阵列模块(embedded array block, EAB，例如：2KB 的 RAM 或 ROM)，可以配置成 256×8、512×4、1024×2 或 2048×1 的存储器元器件。这些 EAB 和 LAB 通过每列 100 到 300 根线的高速宽带总线连接起来，如图 1-13 所示。表 1-6 给出了 Altera FLEX 10K 系列的部分元器件。

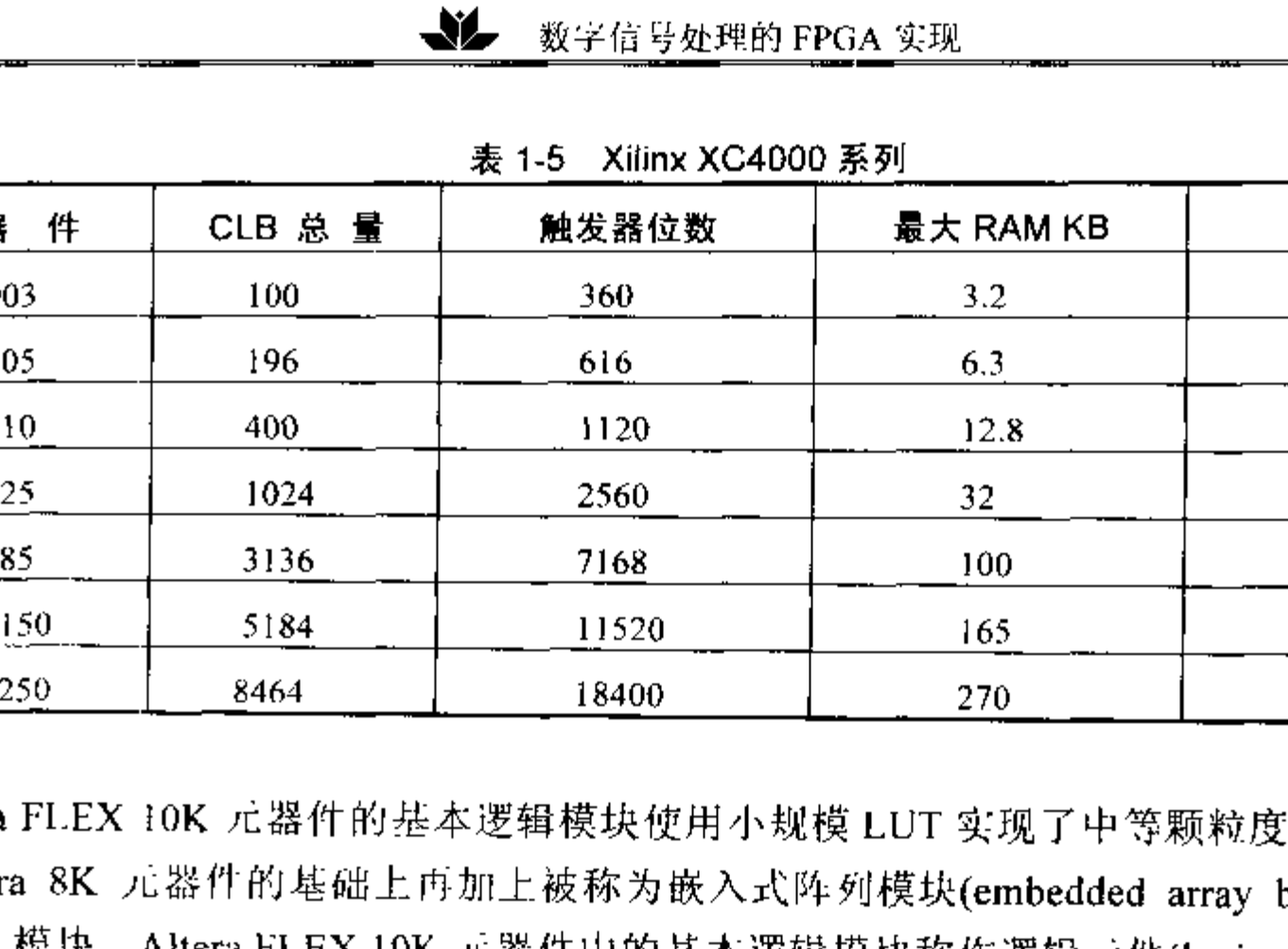


图 1-12　FLEX 逻辑单元(©1996 年，Altera)

注 3：有时候在“设计报告文件”中也称作逻辑单元(logic cell, LC)。

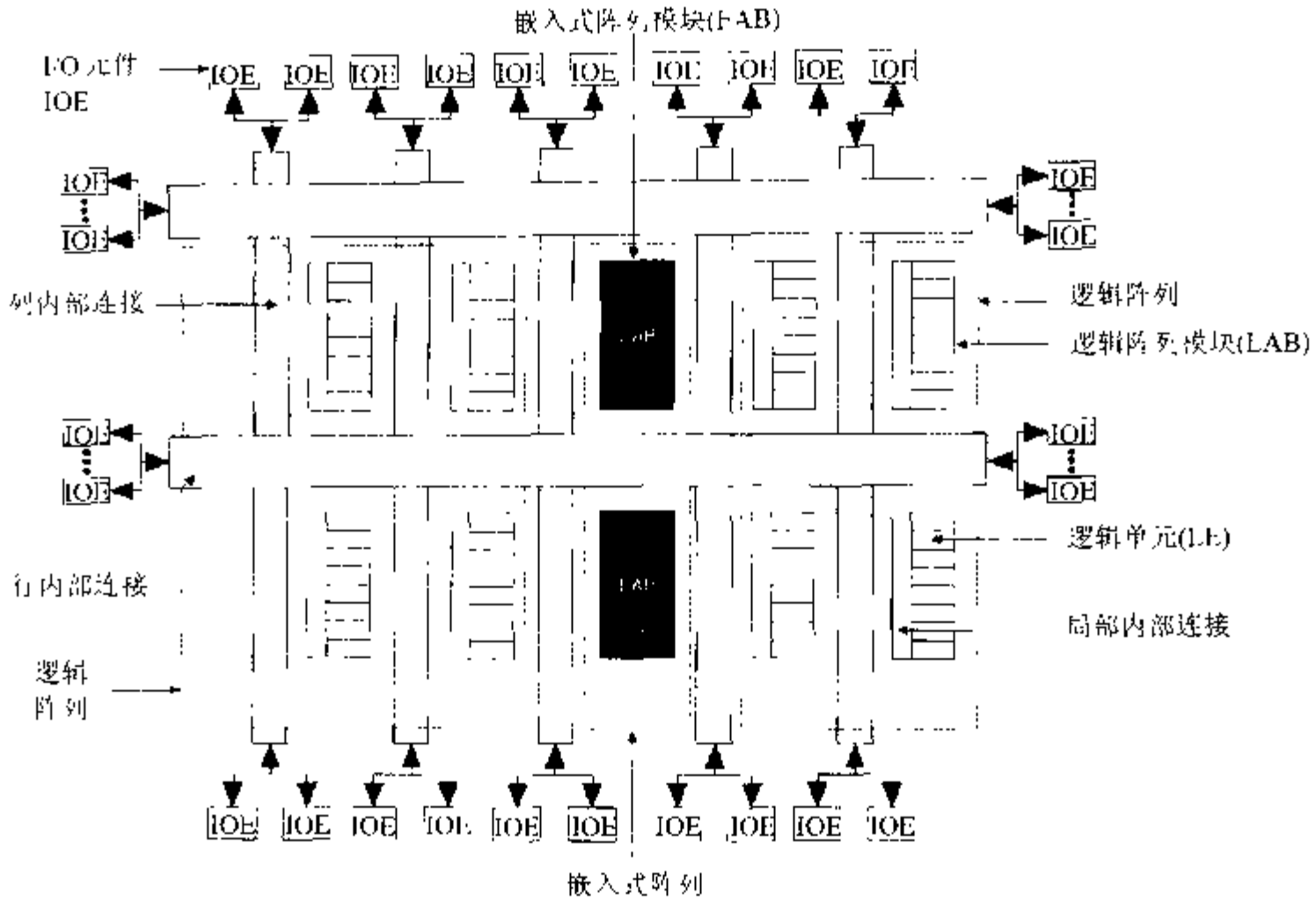


图 1-13　FLEX 10K 元器件内的总体总线结构(©1996 年，Altera)

表 1-6　FLEX 10K 系列

元　器　件	逻辑元件总数	触发器位数	EAB 模 块	最大 RAM KB	最 大 I/O
EPF10K10	576	720	3	6	134
EPF10K20	1152	1344	6	12	189
EPF10K30	1728	1968	6	12	246
EPF10K40	2304	2576	8	16	189
EPF10K50	2880	3184	10	20	310
EPF10K70	3744	4096	9	18	358
EPF10K100	4992	5392	12	24	406
EPF10K130	6656	7120	16	32	470
EPF10K250	12160	12624	20	40	470

如果将这两种分别来自 Altera 和 Xilinx 的路由策略加以比较，就会发现这两种方法都很有价值：Xilinx 的方法拥有更多的局部路由资源而全局资源则较少，这对 DSP 的使用是有促进作用的，因为绝大部分数字信号处理算法都是处理局部数据的。Altera 具有宽带总线的方法也有其价值，因为典型的操作不是在“位片(bit slice)”操作中一位一位地处理，更为常见的是必须把 16 位到 32 位的宽带数据矢量转移到下一个 DSP 模块中。

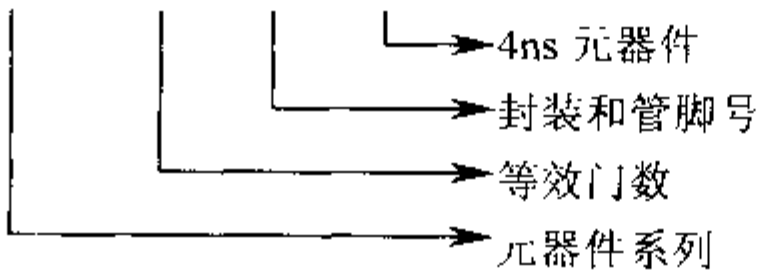
1.4.2　Altera EPF10K20RC240-4

在本书中，从头到尾都将使用的元器件是由 Altera 的大学课程提供的、作为演示线路板一

部分的 Altera EPF10K20RC240-4 元器件。

该元器件命名法的解释如下：

EPF10K20RC240- 4



在可能的情况下，在具体的设计示例中用到 Altera 的元器件时，都使用 Altera 支持的软件。MaxPlusII 软件是一个完全集成的系统，包括 VHDL 和 Verilog 编辑器、合成器、仿真器和位流发生器。鉴于所有的示例在 VHDL 和 Verilog 下都是可行的，可能还会需要其他的仿真器。例如：与元器件无关的 Synopsys FC2 或 ModelTech 编译器就可以成功地应用到示例中。

1. 逻辑资源

EPF10K20 是 Altera 10K 系列中的一员，其门复杂程度相当于大约 20 000 个 2 输入与非门。然而，可以实现的完整加法器的最大数量应该是一个更为有用的 DSP 应用尺度。从表 1-6 中可以看到，EPF10K20 元器件具有 1 152 个基础逻辑单元(logic element, LE)。这也是可以实现完整加法器的最大数目。每个 LE 都可以用作一个 4 输入 LUT，或在“算法”模式中用作带有一个额外快速进位的 3 输入 LUT，如图 1-12 所示。8 个 LE 组成一个逻辑阵列模块(logic array block, LAB)。LAB 的数目就是 1152/8=144。这 144 个 LAB 安排在 6 行 24 列中。元器件还包括一个位于每行中心的 2KB 的存储器模块(称为嵌入式阵列模块(embedded array block, EAB))。因此 EPF10K20 具有 6 个 EAB，也就是 12KB 的存储器。图 1-13 给出了元器件的部分平面布置图。

2. 布线资源

每个 LAB 都有 22 个来自各行的输入信号和 8 个来自逻辑元件的信号，还有 4 个额外的 LAB 控制信号(例如：寄存器的预置)和两个局部进位和串级内连。为了将 LAB 连接起来，EPF10K20 采用了快速宽带行总线和列总线，称作“捷径内连”。每个行总线为 144 条线宽，而每列是 24 个信道。为了改良可布线性，Altera 将行互联分成标准长度(全部为 96 个信道)和半标准长度(2×48 =96 个信道)。半标准长度信道在信道中间 EAB 所在的位置截止。EAB 可以访问两个半标准长度信道。可以注意到很有趣的是，长进位链跳越了备用的行，所以每两个 EAB 才会遇到相同的进位链(请参阅图 1-17)。

3. 定时估算

Altera 的 MaxPlusII 软件计算了多种时间数据，例如：Delay Matrix、Registered Performance 和 Hold Matrix。要获取完整的所有时间参数的描述，请浏览 Altera 的网页[19]。为了获得最佳性能，有必要了解软件在物理上是如何实现设计的。因此对解决方案进行一下大致的评估还是有益的，然后再决定如何改进设计。

例 1.2　16 位加法器的速度

假定设计者需要实现一个 16 位加法器并估算其设计的最大速度。加法器可以在两个 LAB

上实现，每个都需要使用快速进位链。通过“相同行”的延迟必须加以考虑。总延迟计算如下：首先，两个输入必须是稳定的 t_{co}。其次，必然产生第一次进位 t_{cgen}，这个时间按照 7 倍或更多倍数于第一个 LAB 内部进位的时间计算。接下来，信号通过行串级内连 $t_{samerow}$。在第二个 LAB 内部，有 7 个附加的进位必须计算。最高有效位(Most Significant Bit, MSB)要通过一个 LUT 完成求和。结果存储在 LE 寄存器中。下面总结了这些时间数据：

LE 寄存器时钟到输出的延迟	t_{co} =	0.2 ns
数据输入到进位输出的延迟	t_{cgen} =	1.5 ns
进位输入到进位输出的延迟	$7 \cdot t_{cico}$ =	$7 \cdot 0.3 = 2.1$ ns
行路由延迟	$t_{samerow}$ =	2.9 ns
进位输入到进位输出的延迟	t_{cico} =	$7 \cdot 0.3 = 2.1$ ns
LE 查阅表延迟	t_{LUT} =	1.9 ns
LE 寄存器设置时间	t_{su} =	2.7 ns
总计	=	13.4 ns

大致的延迟是 13.4ns，或者说速度是 74.6MHz。这一设计预计需要使用大约 16 个 LE(请参阅练习 1.7)。

如果所使用的两个 LAB 没有安排在同一行的话，还要有同列延迟 $t_{samecolumn}=4.4$ns(代替 $t_{samerow}$)。最差的情况出现在如果所用的两个 LAB 在不同的行上。此时，延迟变成 $t_{diffrow}=10.1$ns。因此，正如在 Altera 的入门手册 231-241 页[20]中所描述的那样，核查平面布置图以及检查可能的“手动”改进平面布置图是非常重要的。

4. 功耗

FPGA 的功耗可以说是一个关键的设计约束条件，特别是对于移动应用。因此推荐使用 3.3V 或 2.5V 级的元器件。为了估算 Altera 元器件 EPF10K20RC240-4 的功耗，必须考虑 3 个方面的因素，也就是：

(1) 维持功耗：$I_{standby}\approx 0.5$mA

(2) I/O 功耗 $I_{I/O}$

(3) 有效功耗 I_{active}

前两项是与设计无关的，CMOS 技术中产生的维持功耗非常少。有效电流主要与时钟频率和使用的 LE 数目有关。Altera 提供了下述估算有效电流的经验公式：

$$I_{active}=98 \cdot f_{max} \cdot N \cdot T_{LE} \cdot \mu A/(\text{MHz} \cdot \text{LE}) \tag{1.1}$$

其中 f_{max} 是按 MHz 计算的最大工作频率，N 是元器件中使用的所有逻辑单元的总数，T_{LE} 是逻辑单元在每个时钟周期内触发的平均百分比(典型值是 12%)。例如：假定设计者使用了 EPF10K20RC240-4 的所有 LE，且最大工作频率是 24MHz，那么电流值就约为 338mA。

下面的案例研究将用作以后章节的例子和自学问题的一个详细的图解。

1.4.3 案例研究：频率合成器

下面案例研究的设计目标就是基于 Philips PM5190(大约 1979 年，请参阅图 1-14)模式实现

一个经典的频率合成器。频率合成器包括一个 32 位的累加器，有 8 个最高有效位(most significant bits, MSB)连在一个 SIN-ROM 的查阅表(lookup table) (LUT)上，从而产生所需要的输出波形。利用 Altera 的 MaxPlusII 软件的图形化解决方案如图 1-15 所示。下面的 VHDL 文本文件实现采用了“组件示例(component instantiation)”的设计，其中包括：

(1) 设计的编译

(2) 设计结果和平面布置图

(3) 设计的仿真

(4) 性能评估

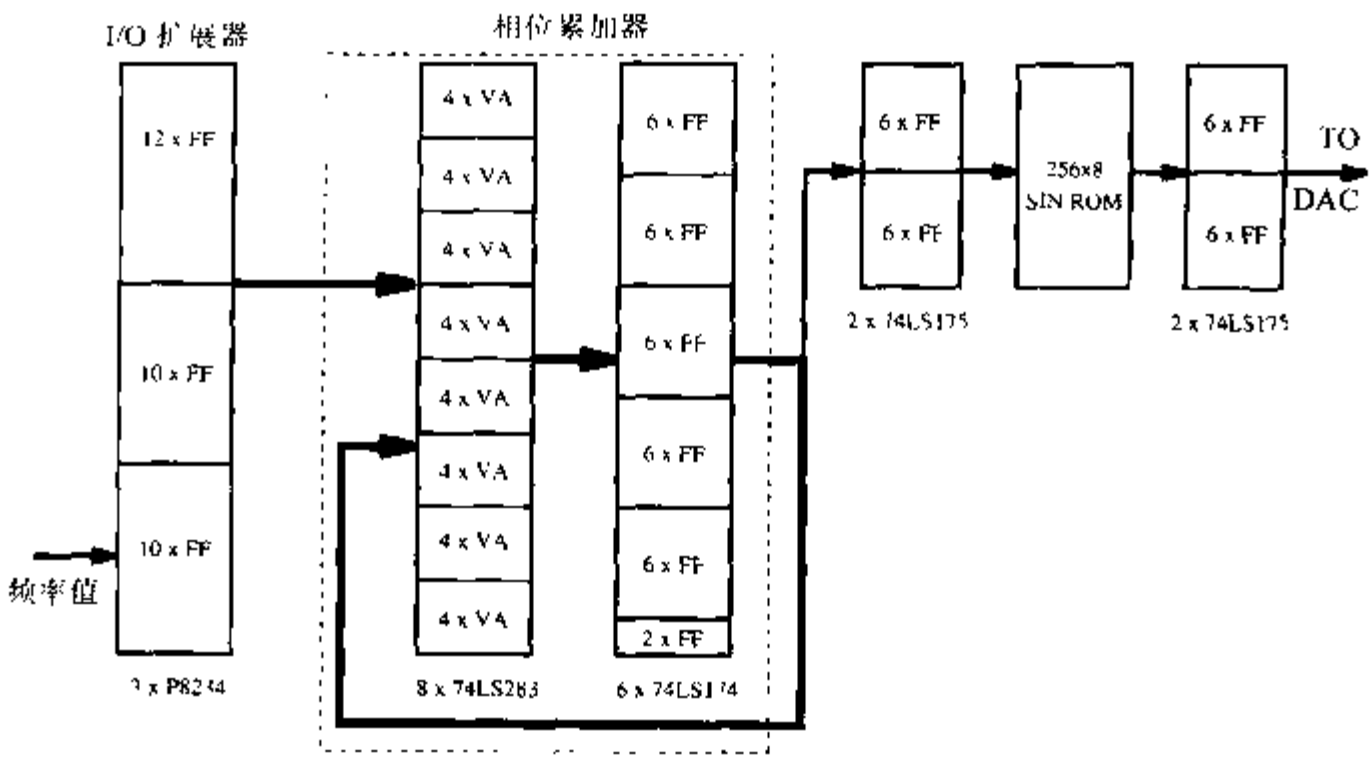


图 1-14　PM5190 频率合成器

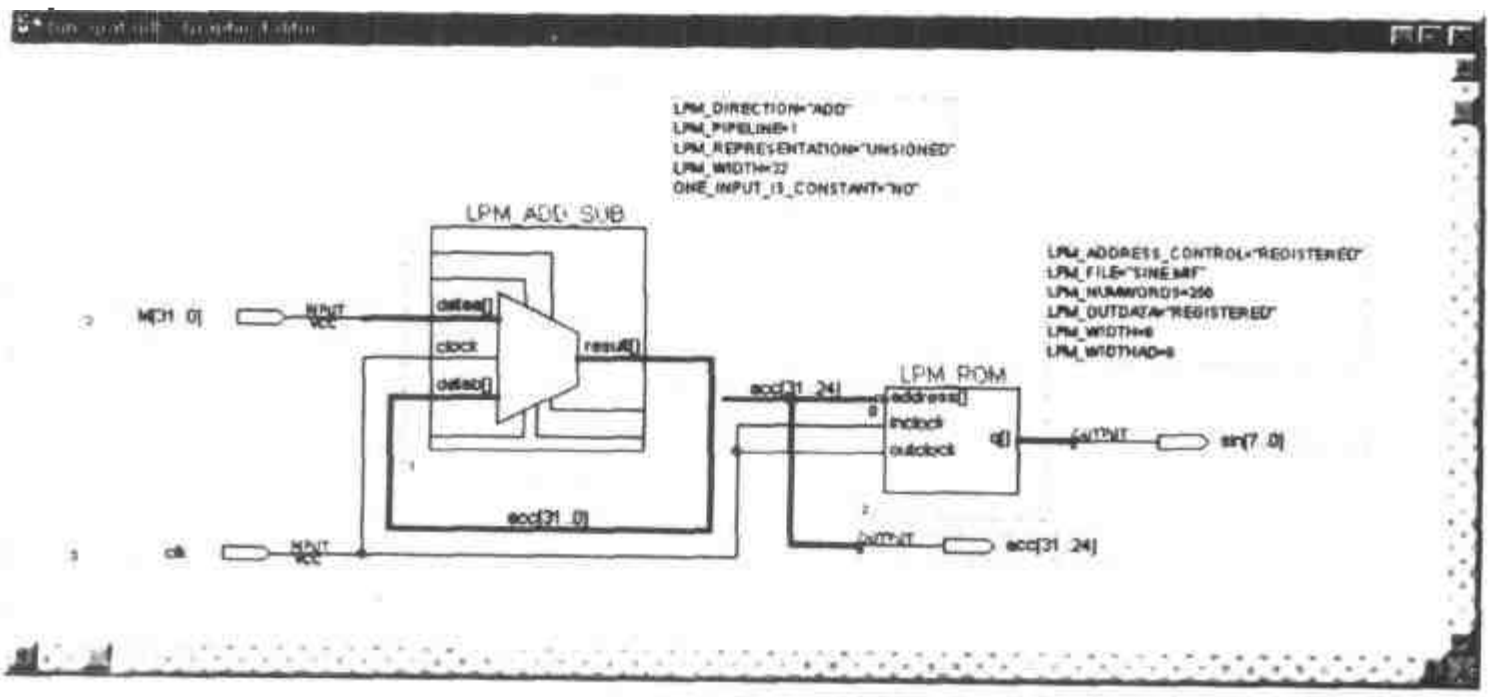


图 1-15　频率合成器的图形化设计

1. 设计的编译

在 MaxPlusII 软件环境下，要检查和编译文件，首先要启动软件，选择 File|Open 加载

fun text.vhd 文件。注意：上边和左边的菜单有所变化。VHDL 设计[4]如下。

```
--  A 32 bit function generator using accumulator and ROM

LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY fun_text IS
  GENERIC ( WIDTH     : INTEGER := 32);       -- Bit width
  PORT ( M            : IN  STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
         sin, acc     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
         clk          : IN  STD_LOGIC);
END fun_text;

ARCHITECTURE fun_gen OF fun_text IS

  SIGNAL s, acc32 : STD_LOGIC_VECTOR(WIDTH-1 DOWNTO 0);
  SIGNAL msbs     : STD_LOGIC_VECTOR(7 DOWNTO 0);
                                                -- Auxiliary vectors
BEGIN

  add1: lpm_add_sub                  -- Add M to acc32
    GENERIC MAP ( LPM_WIDTH => WIDTH,
                  LPM_REPRESENTATION => "SIGNED",
                  LPM_DIRECTION => "ADD",
                  LPM_PIPELINE => 0)
    PORT MAP ( dataa => M,
               datab => acc32,
               result => s );

  reg1: lpm_ff                       -- Save accu
    GENERIC MAP ( LPM_WIDTH => WIDTH)
    PORT MAP ( data => s,
               q => acc32,
               clock => clk);

  select1: PROCESS (acc32)
             VARIABLE i : INTEGER;
           BEGIN
```

注 4：这一案例相应的 Verilog 代码文件 fun_text.v 可以参阅附录 A。

```
                FOR i IN 7 DOWNTO 0 LOOP
                  msbs(i) <= acc32(31 - 7+i);
                END LOOP;
              END PROCESS select1;

  acc <= msbs;

  rom1: lpm_rom
    GENERIC MAP ( LPM_WIDTH => 8,
                  LPM_WIDTHAD => 8,
                  LPM_FILE => "sine.mif")
    PORT MAP ( address => msbs,
               inclock => clk,
               outclock => clk,
               q => sin);

END fun_gen;
```

在代码开头部分的对象 LIBRARY 中包括预定义模块和定义。Entity 模块确定了元件的 I/O 接口和类属变量。附带元件说明的 3 个模块(请参阅标识符 add1、reg1、rom1)称为类子程序。“select1” PROCESS 结构是用来选择 8 个最高有效位并在 ROM 中寻址的。为将目标设置为当前文件，需要选择 File | Select | Set Project to Current File。为了优化速度的设计，可以选择菜单 Assign|Global Project Logic Synthesis 的选项 Optimize10(速度)，并设定 Global Project Logic Synthesis Style 为 FAST。通过菜单中的 Assign|Device for Device Family 的选项 FLEX10K 来设定元器件的类型为 FLEX10K20。我们选择的元器件是 FLEX10K20RC240-4。下一步利用快捷键<Ctrl+K>或者选择 File|Project|Save&Check 来启动语法检查程序。编译器检查基本的语法错误并生成网络表文件 fun_text.cnf。在语法检查成功后，就可以选择编译器窗口内的 START 按钮或者选择 File|Project|Save&Compile 开始进行编译。如果所有的编译步骤都顺利完成，设计也就完整地实现了。图 1-16 总结了编译过程中的所有处理步骤，这些步骤如 MaxPlusII 编译器窗口中所示。

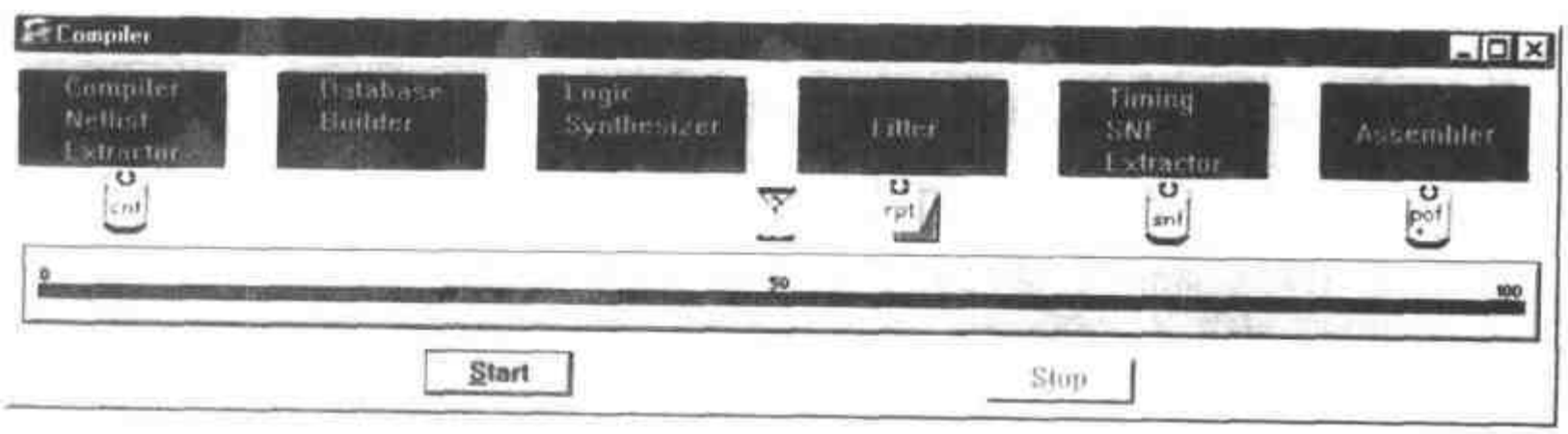


图 1-16　MaxPlusII 中的编译步骤

2. 平面布置图

通过打开 File|Open|fun_text.rpt，或者双击“rpt”按钮，就能够在编译器窗口(请参阅图 1-16)中看到设计结果。在 Utilities|Find Text|LCs 下的“device summary”中可以看到所使用的 LC 和存储器模块的数量。在报告文件中可以看到元器件的输出管脚和逻辑合成的结果(例如：逻辑方程)。核对包括偏移二进制码格式的正弦表的存储器初始化文件 sine.mif，选择 MaxPlusII|Ploorplan Editor 就可以看到其物理实现。单击“reduce scale”按钮就可以生成图 1-17 所示的屏幕。注意：累加器使用的是快速进位链，因而只有偶数列用作改进路由，具体解释请参阅 1.4.2 节。

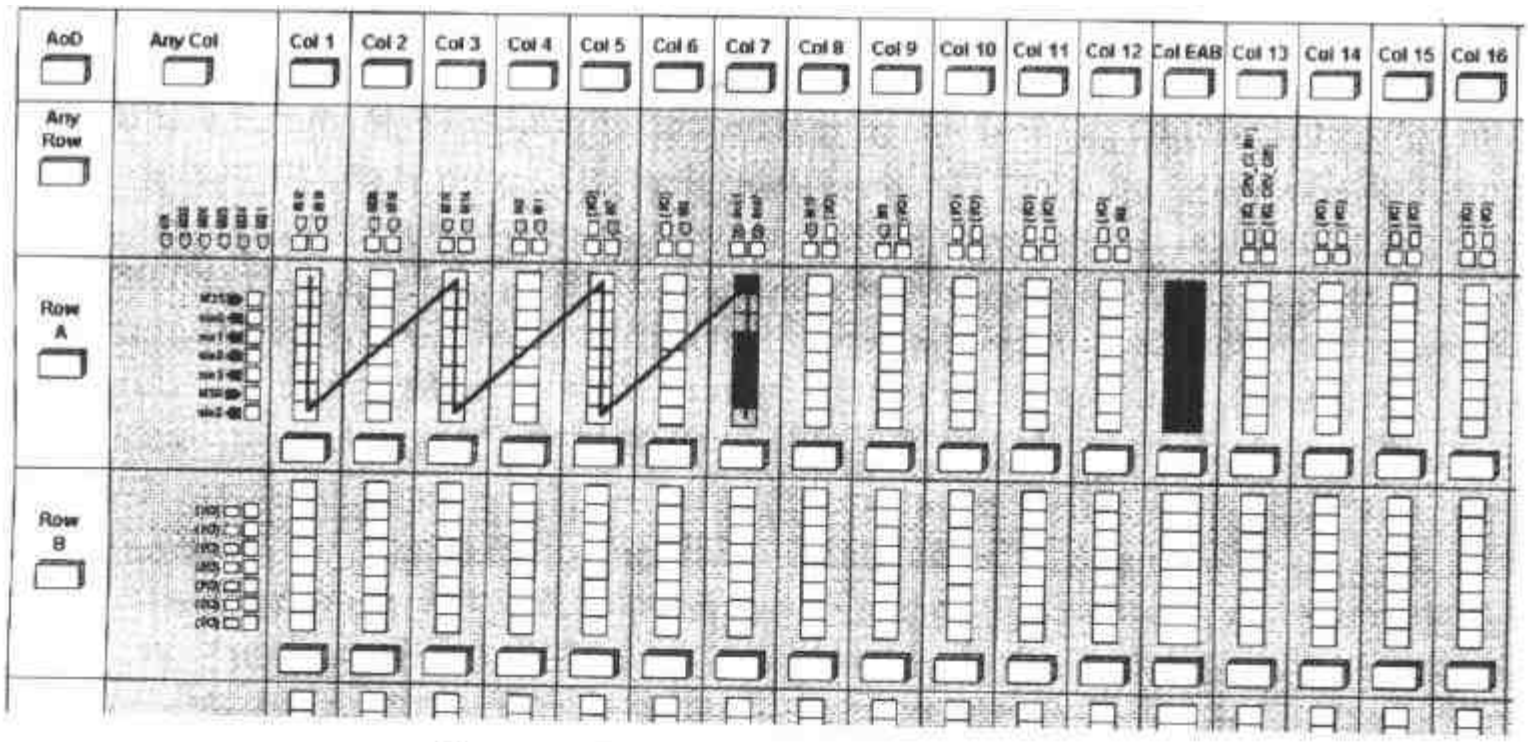


图 1-17　频率合成器设计的平面布置图

3. 仿真

开始仿真时，打开准备好的波形，选择 File|Open|fun_text.scf 命令。注意：上边和左边的菜单已经变化了。从菜单 File|End Time 中设定时间为 1μs。在 fun_text.scf 窗口中单击符号并在重写时钟窗口设定(左侧菜单按钮)时钟周期为 25ns。设定 M=715827883($M=2^{32}/6$)，这样合成器的周期就是 6 个时钟周期长。选择 MaxPlusII|Simulator 并单击“开始”按钮就开始进行仿真了。应该给出一个与图 1-18 相近的输出。注意：ROM 是按着二进制偏移(例如：zero=128)编码的。当完成以后，改变频率，就出现一个 8 个循环的周期，也就是，($M=2^{32}/8$)，重复仿真的上述过程。

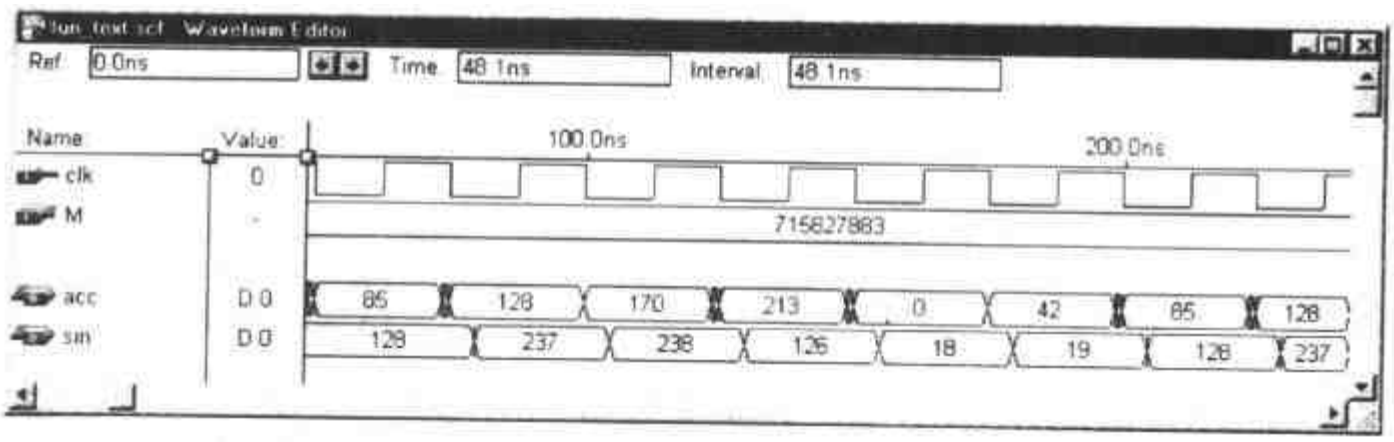


图 1-18　频率合成器设计的 VHDL 仿真

4. 性能分析

要进行性能分析，先要进入 MaxPlusII | Timing Analyzer。注意：菜单又有所变化。选择 Analysis | Registered Performance，就会出现相应的 Registered Performance 屏幕。单击“开始”按钮来测定寄存器的性能。结果应该与图 1-19 类似。

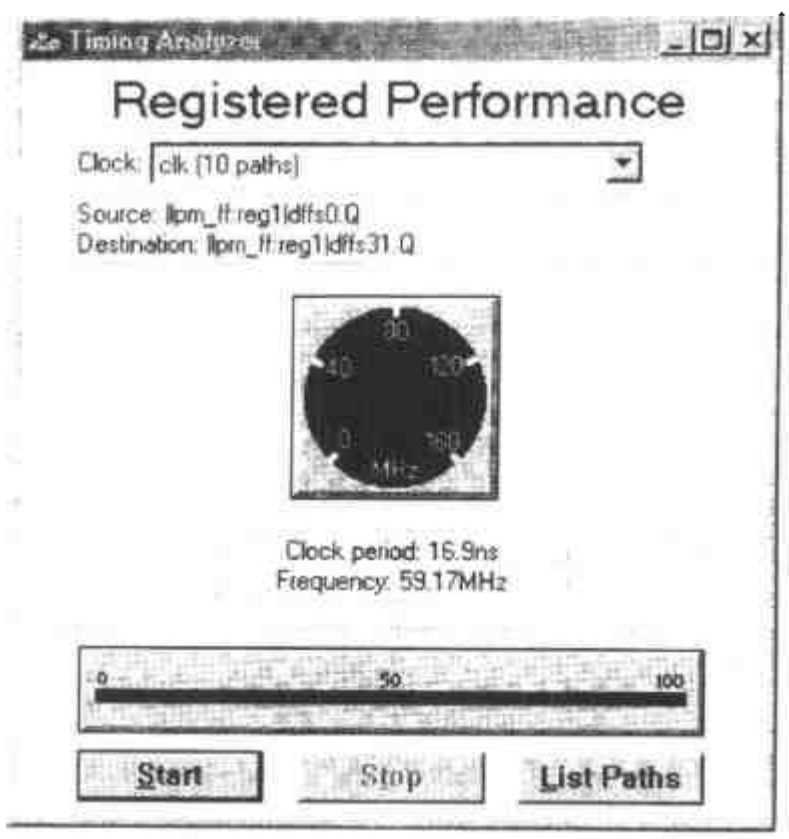


图 1-19　频率合成器设计的寄存器性能

在此就结束了频率合成器的案例研究。

1.5　练习

1.1：只使用两输入与非门来实现一个全加器：

(a) $s=a \oplus b \oplus c_{in}$(注：⊕＝XOR，或非门)

(b) $c_{out}=a \cdot b+c_{in} \cdot (a+b)$ (注：+＝OR，或门；·＝AND，与门)

(c) 利用与非门实现非门、与门和或门，从而说明 2 输入与非门是通用的。

练习使用 MaxPlusII

1.2：(a) 在函数模式下，利用 MaxPlusII 编译器编译文件 example.vhd。选择编译器选项 Processing | Functional SNF Extractor。

(b) 利用文件 example.scf 仿真设计。

注意：

如果您没有使用 MaxPlusII 软件的经验，可以参阅 1.4.3 节的案例研究。

(c) 利用带有定时抽取的 MaxPlusII 编译器编译文件 example.vhd，选择编译器选项 Processing | Timing SNF Extractor。

(d) 利用文件 example.scf 仿真设计。

(e) 打开仿真器窗口中的选项 Check Outputs，并比较函数和实现的 SNF。

1.3：(a) 为大致如图 1-20 所示的 clk、a、b、op1 生成一个波形文件。

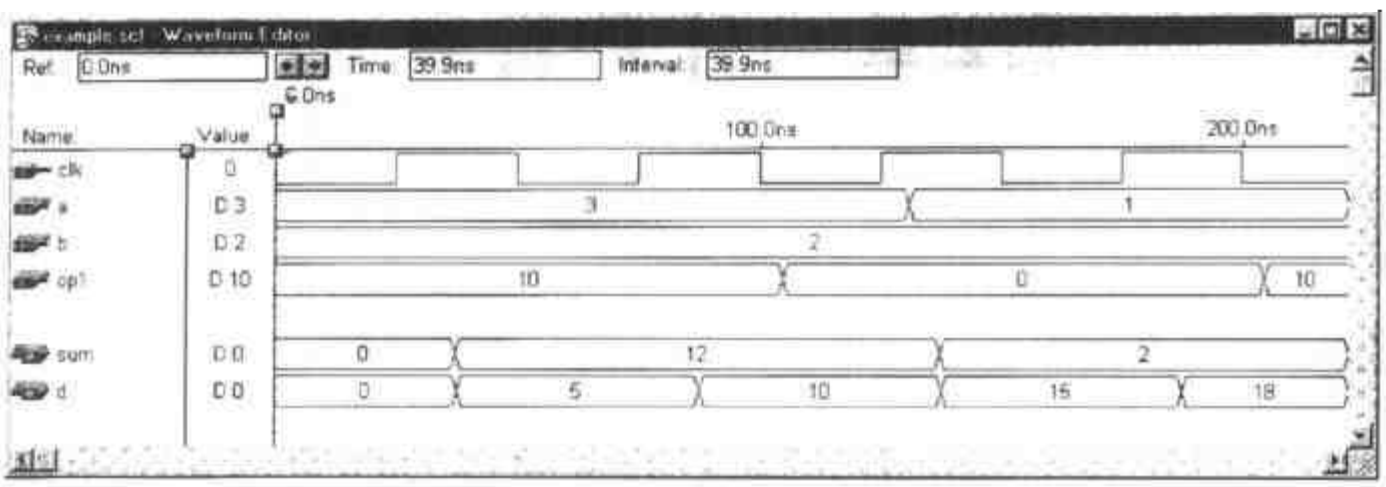


图 1-20　例 1.1 的波形文件

(b) 利用 VHDL 代码 example.vhd 进行仿真。

(c) 解释 a、b、op1 和 sum、d 之间的代数关系。

1.4：(a) 用合成类型 Fast 和 Normal 编译文件 fun_text.vhd(Assign | Global Project Logic Synthesis)。

(b) 评估(a)中两个设计的 Registered Performance 和 LC 的使用情况并解释其结果。

1.5：(a) 用合成类型 Fast 和编译器选项 Processing | Timing SNF Extractor，编译文件 fun_text.vhd(Assign | Global Project Logic Synthesis)。

利用波形文件 fun_text.snf 并且

(b1) 设定时钟信号周期为 20ns，应用仿真器检查 Setup/Hold，Check Outputs，Oscillation 以及 Glitch。

(b2) 设定时钟信号周期为 15ns，应用仿真器检查 Setup/Hold，Check Outputs，Oscillation 以及 Glitch。

1.6：(a) 打开文件 fun_text.scf 并进行仿真。

(b) 选中上边菜单栏标有 Initialize 的仿真窗口，选择 Initialize Memory 并输出 Intel 十六进制形式的 ROM 表文件 sine.hex。

(c) 更换成 fun_text.vhd 文件，并令其使用 Intel 十六进制形式的 ROM 表文件 sine.hex，进而通过仿真确定正确的结果。

1.7：(a) 使用 MaxPlusII 软件的 LPM_ADDSUB 宏设计一个 16 位加法器。

(b) 测定其 Registered Performance 并将结果与例 1.2 中的数据进行比较。

第2章　计算机算法

2.1　概述

在计算机算法中，有两个基本设计准则是非常重要的：分别是数字表示法和代数运算的实现[21, 22, 23, 24, 25]。我们首先来讨论可行的数字表示法(例如：定点数或浮点数)，然后是基本的运算，像加法器和乘法器，最后是更为繁琐的运算，诸如求平方根和应用 CORDIC 算法计算角函数的有效实现。

由于其物理位级编程结构的特点，FPGA 提供了大量实现数字信号处理算法所需要的计算机算法。这恰好与带有定点多级累加器内核的可编程数字信号处理器(programmable digital signal processors, PDSP)相反。在 FPGA 设计中仔细地选择位宽就能够从本质上做到节约。

2.2　数字表示法

必须仔细考虑确定是定点数还是浮点数更适合于问题的解决，特别是在工程的早期阶段。一般可以认为：定点数的实现具有更高的速度和更低廉的成本，而浮点数则具有更高的动态范围且不需要换算，这对较为复杂的算法可能更有吸引力。图 2-1 给出了传统和非传统定点数和浮点数的数字表示法的一个概观。两套系统都由许多各自的标准所覆盖，当然，如果需要的话，也可以以一种专有形式实现。

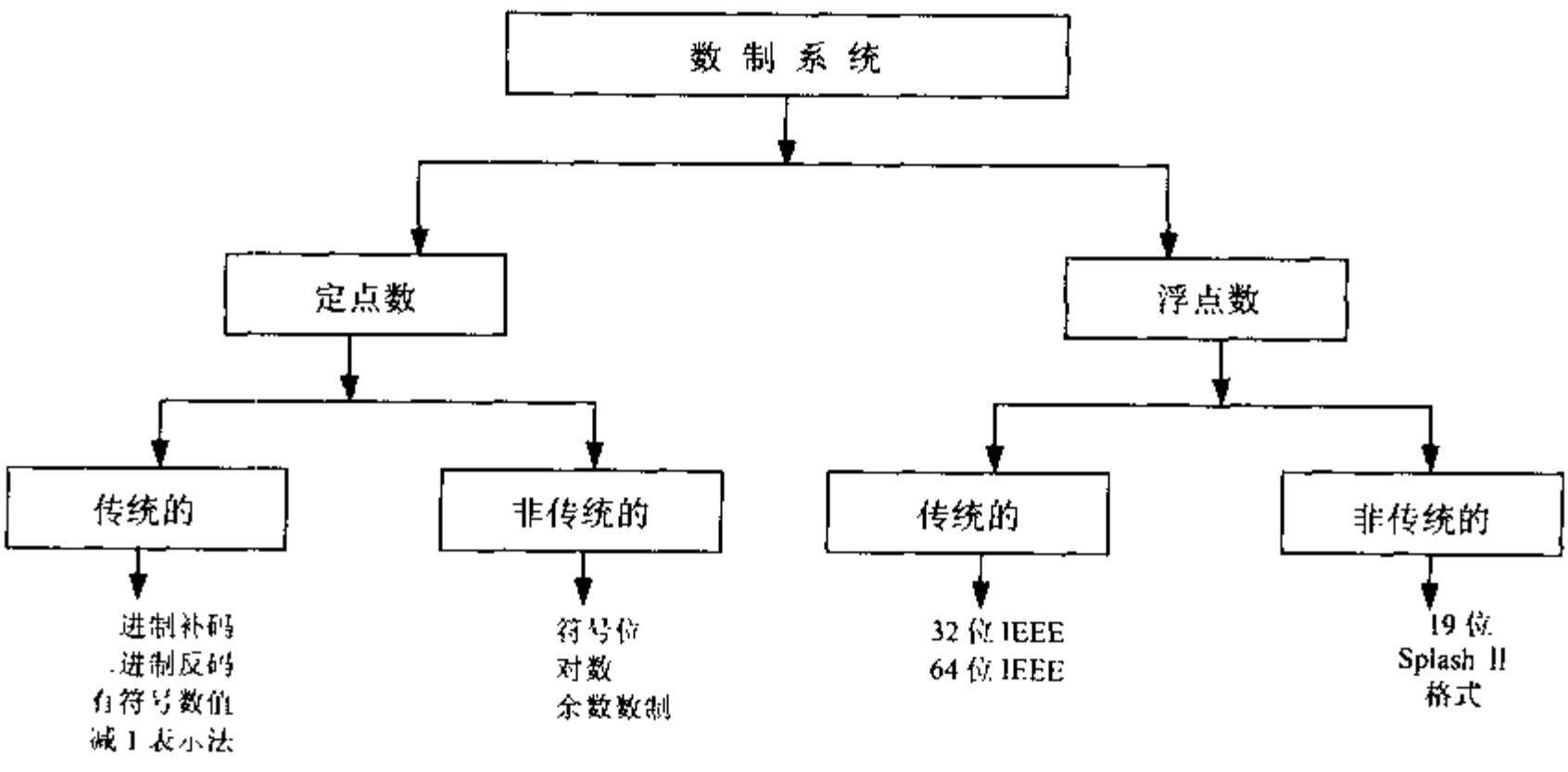


图 2-1　数字表示法概观

2.2.1　定点数

首先回顾一下图 2-1 和表 2-1 中的定点数系统。

表 2-1　有符号二进制数的常规编码

二 进 制 数	2C	1C	D1	SM
011	3	3	4	3
010	2	2	3	2
001	1	1	2	1
000	0	0	1	0
111	-1	-0	-1	-3
110	-2	-1	-2	-2
101	-3	-2	-3	-1
100	-4	-3	-4	-0

1. 无符号整数

设 X 是一个 N 位无符号二进制数，则其范围是[0，2^N-1]，表达式如下：

$$X=\sum_{n=0}^{N-1}x_n2^n \tag{2.1}$$

其中 x_n 是 X 的第 n 位二进制数字(也就是 $x_n\in[0,1]$)。数字 x_0 称作**最低有效位**(Least Significant Bit，LSB)，具有相当于个位的权重。数字 x_{N-1} 就是**最高有效位**(Most Significant Bit, MSB)，具有相当于 2^{N-1} 的权重。

2. 有符号数值

在有符号数字表示法中，数字和符号是单独表示的。第一位代表符号，余下的 $N-1$ 位代表数字，表达式如下：

$$X=\begin{cases}\sum_{n=0}^{N-1}x_n2^n & X\geq 0\\ -\sum_{n=0}^{N-1}x_n2^n & X<0\end{cases} \tag{2.2}$$

这一表达式的范围是[-2^{N-1}，2^{N-1}]，有符号数字表示法的优点就是简化了溢出的禁止，但缺点就是加法不得不根据哪一个操作数更大而分开来运算。

3. 二进制补码(Two's Complement, 2C)

有符号整数的 N 位二进制补码表达式如下：

$$X=\begin{cases}\sum_{n=0}^{N-1}x_n2^n & X\geq 0\\ 2^N-\sum_{n=0}^{N-1}x_n2^n & X<0\end{cases} \tag{2.3}$$

其表达式的范围是[-2^{N-1}，$2^{N-1}-1$]。二进制补码表示法是目前 DSP 领域内最为流行的有符号数字表示法。这是因为它使得累加多个有符号数值成为可能，而且最终结果是在 N 位范围

内，我们可以忽略任何算术上的溢出。例如我们计算两个3位数，其过程如下：

$$\begin{aligned} 3_{10} &\leftrightarrow 011_{2C} \\ -2_{10} &\leftrightarrow 110_{2C} \\ 1_{10} &\leftrightarrow 1.001_{2C} \end{aligned}$$

溢出可以忽略。所有的计算都是取模2^N。这样就有可能出现不能够正确表示中间值的情形，但只要最终值有效，结果就是正确的。例如计算3位的数字2+2-3，会得到一个中间值$010+010=100_{2C}$，也就是-4_{10}，但是结果$100-011=100+101=001_{2C}$，是正确的。

二进制补码还可以用来实现模2^N算法，而且不需要在算法中作任何改动，我们将在第5章设计CIC滤波器时用到这些知识。

4. 二进制反码(也称作1的补码，One's Complement，1C)

N位二进制反码数字表示法可以表示的整数范围是$[-2^{N-1}-1, 2^{N-1}-1]$。在二进制反码中，正整数和负整数除了符号位之外具有相同的表示方法。也就是说，事实上0需要额外的表达式(请参阅表2-1)。二进制反码中有符号数的标准表达式如下：

$$X=\begin{cases}\sum_{n=0}^{N-1}x_n 2^n & X\geq 0\\ 2^N-1-\sum_{n=0}^{N-1}x_n 2^n & X<0\end{cases} \tag{2.4}$$

例如：在表2-1第三列中给出了3和-3的3位二进制反码表达式。

请看下面的简单示例：

$$\begin{aligned} 3_{10} &\leftrightarrow 0\ 1\ 1_{1C} \\ -2_{10} &\leftrightarrow 1\ 0\ 1_{1C} \\ 1_{10} &\leftrightarrow 1.\ 0\ 0\ 0_{1C} \\ \text{进位} &\quad \rightarrow\ \rightarrow\ \rightarrow\ 1_{1C} \\ 1_{10} &\leftrightarrow 0\ 0\ 1_{1C} \end{aligned}$$

在二进制反码中需要“进位回绕(carry wrap-around)”加法。在最高有效位与最低有效位相加得到正确结果时，就会出现进位。

尽管如此，这种数字表示法还是能够有效地实现模2^N-1运算，而且不需要校正。因此二进制反码在实现特定的DSP算法(例如：整数计算环2^N-1的Mersenne变换，请参阅第7章)时，还是有其特殊价值的。

5. 减1表示法(Diminished one System, D1)

减1表示法是一种有偏移的数字表示法。正整数与二进制补码相比减少了1。N位D1数值范围是$[-2^{N-1}, 2^{N-1}]$(不含0)。D1数字表示法的编码规则定义如下：

$$X=\begin{cases}\sum_{n=0}^{N-1}x_n 2^n-1 & X\geq 0\\ 2^N-\sum_{n=0}^{N-1}x_n 2^n & X<0\\ 2^N & X=0\end{cases} \tag{2.5}$$

从下面两个D1数相加可以看到，对于D1而言还必须计算补码和颠倒进位的加法。

$$
\begin{array}{rcccccc}
3_{10} & \leftrightarrow & & 0 & 1 & 0_{D1} \\
-2_{10} & \leftrightarrow & & 1 & \bar{1} & 0_{D1} \\
1_{10} & \leftrightarrow & 1. & 0 & 0 & 0_{D1} \\
\text{进位} & & \rightarrow & \boxed{\cdot -1} & \rightarrow & 0_{D1} \\
1_{10} & \leftrightarrow & & 0 & 0 & 0_{D1}
\end{array}
$$

D1 数不需要在算法上作任何改动就能够有效地实现模 2^N+1 运算。在第 7 章中，将利用这一结论在 2^N+1 计算环中实现费尔马 NTT(Fermat Network Transfer Table，Fermat 网络传输表)。

2.2.2　非传统定点数

接下来，我们根据图 2-1 继续回顾数字表示法。下面将要讨论的非传统定点数的数字表示法不像 2C 表示法那样经常使用，但是在特定的应用场合或者解决特殊问题时，还是能够显著地提高效率。

1. 有符号数字量(Signed Digit Numbers, SD)

有符号数字量表示法与前面小节中谈到的传统二进制有所不同，实际上它具有三重值(也就是说数字的值域是{0，1，－1}，其中－1 经常写成$\bar{1}$)。

SD 表示法应用在超前进位加法器或乘法器中已经被证明能够降低复杂性，这是因为：通常可以通过非零元素的数量来估计乘法的效率，而应用 SD 表示法可以降低非零元素的数量。统计表明，数字的二进制补码编码中有一半数位是零。对于 SD 编码而言，零元素的密度增加到三分之二，如下面的例题所示：

例 2.1：SD 编码

考虑利用 5 位二进制数和 SD 代码对 10 进制数 $15=1111_2$ 进行编码。具体表达式如下：

(1) $15_{10}=16_{10}-1_{10}=10\bar{1}00_{SD}$

(2) $15_{10}=16_{10}-2_{10}+1_{10}=100\bar{1}1_{SD}$

(3) $15_{10}=16_{10}-4_{10}+3_{10}=10\bar{1}10_{SD}$

(4) 等等

与 2C 代码不同，SD 表达式不是惟一的。我们称具有最少非零元素的表示法为正则有符号数字量表示法(canonic signed digit 表示法)，也称作 CSD。可以利用下面的算法生成一个“经典的”CSD 代码。

> **算法 2.2：经典的 CSD 代码**
> 从最低有效位开始，用 $10\ldots0\bar{1}$ 取代所有大于或等于 2 的 1 序列。

这种经典 CSD 代码是独一无二的，而且另一个特性就是最终表达式在两个数位(可取值为 1，$\bar{1}$ 或 0)之间至少有一个 0。

例 2.3：经典的 CSD 代码

再研究一下应用 5 位二进制数和 CSD 代码对 10 进制数 15 进行编码的问题。其表达式是 $1111_2=1000\bar{1}_{CSD}$。与例 2.1 的 SD 编码相比较，就可以看到：只有第一个表达式是 CSD 代码。

再来研究一个编码的例题：

$$27_{10}=11011_2=1110\bar{1}_{SD}=100\bar{1}0\bar{1}_{CSD} \tag{2.6}$$

可以看到：尽管第一次 $011\rightarrow10\bar{1}$ 的代换没有降低复杂程度，但是生成了一个长度为 3 的选择，复杂程度从 3 个加法降低到了 2 个减法。

另一方面，鉴于硬件复杂程度的约束，经典 CSD 编码也不总是能够生成“最佳”CSD 编码，因为在算法 2.2 中，加法也是由减法替代的，当没有这样的减法时，就会出现如前所述的情形。例如 011_2 就编码成 $10\bar{1}_{CSD}$，如果利用这一编码生成一个常数乘法器，减法就需要为最低有效位准备一个全加器来取代半加法器。下面给出的 CSD 编码将给出一种非零元素最少的 CSD 码，但也是减法最少的。

算法 2.4：最佳 CSD 编码

(1) 从最低有效位开始，用 $10\ldots0\bar{1}$ 取代所有大于 2 的 1 序列。此外还需用 $110\bar{1}$ 取代 1011。

(2) 从最高有效位开始，用 011 代替 $10\bar{1}$。

2. 自由进位加法器

SD 数字表达式可以实现自由进位加法器。Tagaki 等人[26]引入了如表 2-2 所示的方案。其中 u_k 是中间和，c_k 是第 k 位的进位(也就是要加到 u_{k+1} 上)。

表 2-2　利用 SD 表达式作自由进位二进制数加法

x_ky_k	00	01	01	$0\bar{1}$	$0\bar{1}$	11	$\bar{1}\bar{1}$
$x_{k-1}y_{k-1}$	-	均非 $\bar{1}$	至少 1 个 $\bar{1}$	均非 $\bar{1}$	至少 1 个 $\bar{1}$	-	-
c_k	0	1	0	0	$\bar{1}$	1	$\bar{1}$
u_k	0	$\bar{1}$	1	$\bar{1}$	1	0	0

例 2.5：自由进位加法

SD 表示法中 29 与 -9 的加法执行过程如下：

$$\begin{array}{cccccccc} & 1 & 0 & 0 & \bar{1} & 0 & 1 & x_k \\ + & 1 & \bar{1} & 1 & \bar{1} & 1 & 1 & y_k \\ \hline 0 & 0 & 0 & 1 & 1 & 1 & & c_k \\ & 1 & \bar{1} & 1 & 0 & \bar{1} & 0 & u_k \\ \hline & 1 & 1 & 0 & 1 & 0 & 0 & s_k \end{array}$$

但是由于二进制逻辑的负荷，利用 FPGA 实现表 2-2 时，需要为 c_k 和 u_k 提供 4 输入操作数，也就是说实现表 2-2 需要一个 $2^8\times4$ 位的 LUT。

3. 乘法器-加法器图(Multiplier Adder Graph, MAG)

可以看到乘法器的成本与 A 中非零元素 a_k 的数量直接相关。而 CSD 表示法将这一成本降低到最小。CSD 也是将要在习题 2.2 中讨论的布思乘法器[21]的基础。

但在最佳 CSD 意向中，经常是首先将系数拆分成几个因子，再实现具体因子效率会更高些[27, 28, 29, 30]。图 2-2 以系数 93 为例说明了这一拆分选择。直接二进制和 CSD 码如下：93_{10}

$=1011101_2=1100\bar{1}01_{CSD}$，用 2C 码需要四个加法器，CSD 码需要三个加法器。系数 93 可以表示成 $93=3\cdot31$，每个因子需要一个加法器(请参阅图 2-2)。因子数量的复杂性就降到 2。可以通过几条路径来组合这些不同的因子。所需加法器的数量通常是指常数系数乘法器的成本。图 2-3 根据 Dempster 等人的提议给出了从需要 1 个到 4 个加法器的所有可能配置。利用这张图，就可以合成所有成本在 1 个到 4 个加法器之中的系数($k_i\in N_0$)。依据：

成本 1：(1)　$A=2^{k_0}(2^{k_1}\pm2^{k_2})$

成本 2：(1)　$A=2^{k_0}(2^{k_1}\pm2^{k_2}\pm2^{k_3})$

(2)　$A=2^{k_0}(2^{k_1}\pm2^{k_2})(2^{k_3}\pm2^{k_4})$

成本 3：(1)　$A=2^{k_0}(2^{k_1}\pm2^{k_2}\pm2^{k_3}\pm2^{k_4})$

·
·
·

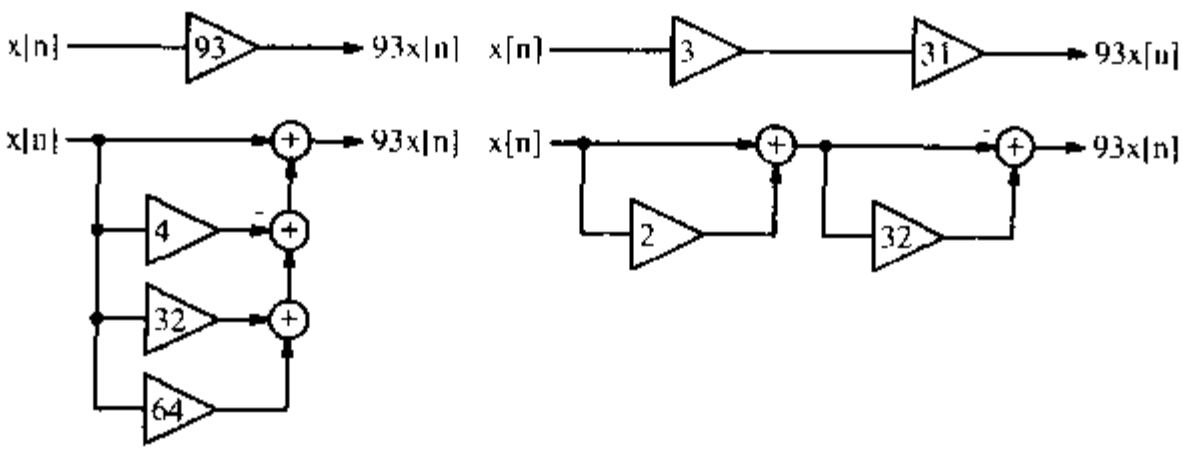


图 2-2　常数因子 93 的两类实现

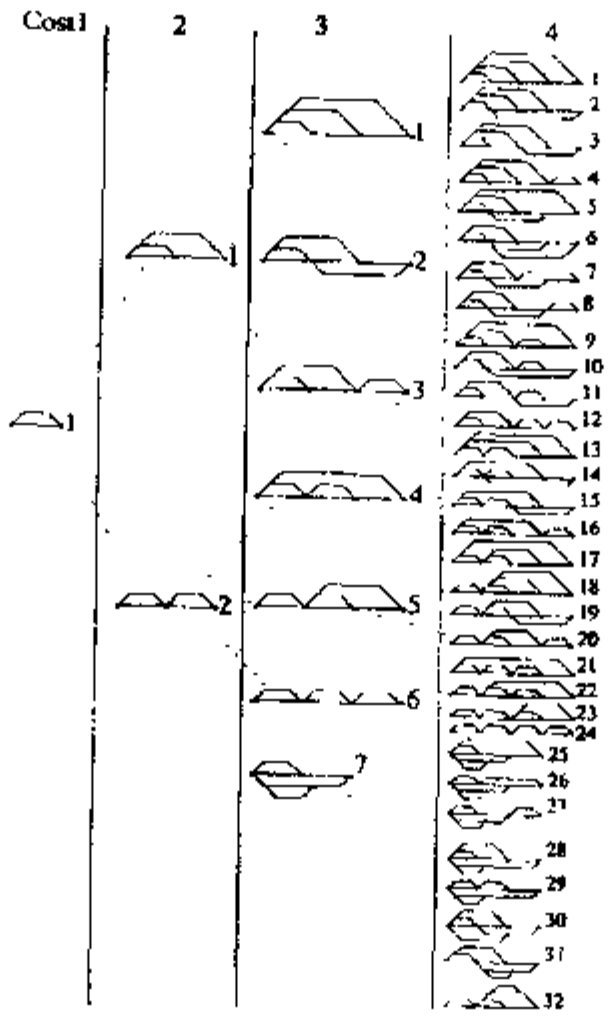


图 2-3　1-4 图的可能成本。每个节点是一个加法器或减法器，每条边与 2 的幂次形式的因子结合起来(©1995 IEEE [29])

利用这一技术，表 2-3 给出所有成本在 0 与 3 之间的 8 位整数的最佳的编码[5]。

表 2-3 运用乘法器-加法器图技术实现所有 8 位整数的成本 C(也就是加法器的数量)

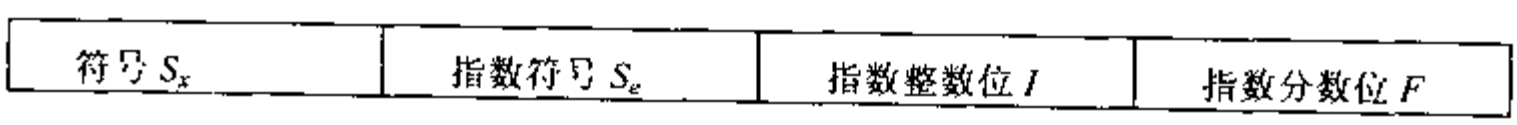
C	系 数
0	4，8，16，32，64，128，256
1	3，5，6，7，9，10，12，14，15，17，18，20，24，28，30，31，33，34，36，40，48，56，60，62，63，65，66，68，72，80，96，112，120，124，126，127，129，130，132，136，144，160，192，224，240，248，252，254，255
2	11，13，19，21，22，23，25，26，27，29，35，37，38，39，41，42，44，46，47，49，50，52，54，55，57，58，59，61，67，69，70，71，73，74，76，78，79，81，82，84，88，92，94，95，97，98，100，104，110，111，113，114，116，118，119，121，122，123，125，131，133，134，135，137，138，140，142，143，145，146，148，152，156，158，159，161，162，164，168，176，184，188，190，191，193，194，196，200，208，216，220，222，223，225，226，228，232，236，238，239，241，242，244，246，247，249，250，251，253
3	43，45，51，53，75，77，83，85，86，87，89，90，91，93，99，101，102，103，105，106，107，109，115，117，139，141，147，149，150，151，153，154，155，157，163，165，166，167，169，170，172，174，175，177，178，180，182，183，185，186，187，189，195，197，198，199，201，202，204，206，207，209，210，212，214，215，217，218，219，221，227，229，230，231，233，234，235，237，243，245
4	171，173，179，181，203，205，211，213
	通过因子分解得到的最小成本
2	45=5·9，51=3·17，75=5·15，85=5·17，90=2·9·5，93=3·31，99=3·33，102=2·3·17，105=7·15，150=2·5·15，153=9·17，155=5·31，165=5·33，170=2·5·17，180=4·5·9，186=2·3·31，189=3·7·9，195=3·65，198=2·3·33，204=4·3·17，210=2·7·15，217=7·31，231=7·33·
3	171=3·57，173=8+165，179=51+128，181=1+180，211=1+210，213=3·71，205=5·41，203=7·29

4. 对数数制(Logarithmic Number System, LNS)

对数数制(Logarithmic Number System, LNS)[31, 32]与由固定尾数和分数指数构成的浮点数制类似。在对数数制中，x 的表达式是：

$$x = \pm r^{\pm e_x} \tag{2.7}$$

其中 r 是数制的基数，e_x 是对数数制的指数。对数数制格式包括一位整数的符号位与一位指数的符号位以及表示整数位 I 和分数位 F 精确度的指数位。以图形的形式说明其格式如下：



LNS 与浮点数一样，精确度不一致。x 的值较小时精确度就比较高，而 x 的值比较大时精

确度就比较低了，请看下面的例题。

例 2.6：LNS 编码

假设基数为 2 的 9 位 LNS 数，具有两个符号位、3 位整数精度和 4 位分数精度。举例说，LNS 编码 00011.0010 是如何转换成实数数制的呢？两个符号位表明整数和指数均为正。整数部分是 3，分数部分是 $2^{-3}=1/8$。因而实数表达式就是 $2^{3+1/8}=2^{3.125}=8.724$。还会发现 $-2^{3.125}=$ 10 011.0010，$2^{-3.125}=$01100.1110。如图 2-4 所示，9 位 LNS 格式所能够表达的最大数是 $2^{8-1/16}\approx256$，最小数是 $2^{-8}=0.0039$。相比之下，8 位正有符号定点数的最大值是 $2^8-1=255$，最小非零正整数是 1。两套 9 位数制比较的结果如图 2-4 所示。

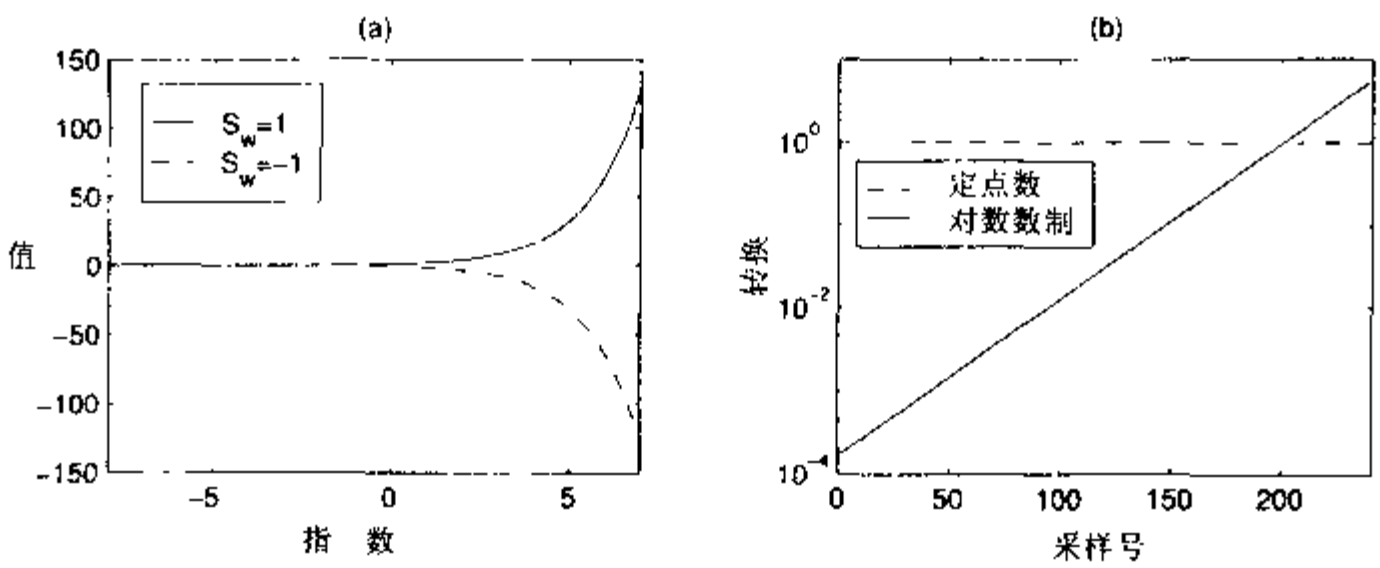


图 2-4　LNS 处理 (a) 值 (b) 转换

历史上，LNS 的优势在于能够有效地实现乘法、除法、求平方根或者平方。例如：乘积 $C=A\times B$，其中 A、B 和 C 是 LNS 数：

$$C=r^{e_a}\times r^{e_b}=r^{e_a+e_b}r^{e_c} \tag{2.8}$$

也就是说 LNS 数乘积的指数只是两个指数简单相加的和。可惜的是，相比较而言，加法和减法就复杂多了。加法和减法运算按照如下过程进行，其中假定 $A>B$：

$$C=A+B=2^{e_a}+2^{e_b}=2^{e_a}\underbrace{(1+2^{e_b-e_a})}_{\Phi+(\Delta)}=2^{e_c} \tag{2.9}$$

其中指数 $e_c=e_a+\phi^+(\Delta)$，而 $\Delta=e_b-e_a$，$\phi^+(u)=\log_2(\Phi(\Delta))$。减法也有与之相似的表，可以用 $\phi(u)=\log_2(\Phi^-(\Delta))$，$\Phi^-(\Delta)=(1-2^{e_b-e_a})$ 计算。历史上，Jurij Vega(1754-1802) 在其著作“Logarithmorm Completus”中论述到，这样的表达式是用来表示有理数的，其中还包括 Zech 计算的表。$\log_2(1-2^n)$项通常特指 Zech 对数。

LNS 算法按照如下方式执行[31]。令 $A=2^{e_a}$、$B=2^{e_b}$、$C=r^{e_c}$，S_A、S_B、S_C 代表每个数的符号位：

运　算	操　作
乘法　$C=AB$	$e_c=e_a+e_b$；　$S_c=S_a$ XOR S_b
除法　$C=A/B$	$e_c=e_a-e_b$；　$S_c=S_a$ XOR S_b
加法　$C=A+B$	$e_c=\begin{cases}e_a+\phi^+(e_b-e_a)A\geq B\\ e_b+\phi^+(e_a-e_b)B>A\end{cases}$

(续表)

运　算	操　作
减法　$C=A-B$	$e_c=\begin{cases} e_a+\phi^{-}(e_b-e_a) & A\geq B \\ e_b+\phi^{-}(e_a-e_b) & B>A \end{cases}$
开平方　$C=\sqrt{A}$	$e_c=e_a/2$
平方　$C=A^2$	$e_c=2e_a$

通过使用部分表[31]或者线性插值[33]的方法，可以减小 Zech 对数所必需的表规模。不过这些方法已经超出我们当前所要讨论的范畴了。

5. 余数数制(Residue Number System, RNS)

RNS 实际上是一种古代代数系统，其历史可以回溯到 2000 年以前。RNS 是一种整数运算系统，其中定义了基本的加、减和乘运算。这些基本运算可以在不连通的短整型字长信道中同时进行[34, 35]。RNS 是在考虑正整数基集$\{m_1, m_2, \ldots m_L\}$的情况下定义的，其中 m_l' 是相对(对偶)最为重要的。结果数字表达式的动态范围是 M，其中 $M=\prod_{l=1}^{L} m_l$。对于有符号数而言，X 的整数值是约束在 $X\in[-M/2, M/2]$。RNS 算法是在一个同构计算环内定义的：

$$Z_M \cong Z_{m_1}\times Z_{m_1}\times\cdots\times Z_{m_l} \tag{2.10}$$

其中 $Z_M=Z/(M)$，与整数模 M 的计算环相关，后者称为余数类模 mod$M$。整数 X 到 RNS L-数组 $X\leftrightarrow(x_1,x_2,\ldots,x_L)$的映射是通过 $x_l=X \bmod m_l$ 定义的，其中 $l=1,2,\ldots,L$。定义□为代数运算的符号+、－和*，如果 Z、X、$Y\in Z_M$ 则有：

$$Z=X\square Y \bmod M \tag{2.11}$$

这与 $Z\leftrightarrow(z_1,z_2,\ldots,z_L)$是同构的，特别是：

$$\begin{array}{ll} X \overset{(m_1,m_2,\ldots,m_l)}{\longleftrightarrow} & (\langle X\rangle_{m_1},\langle X\rangle_{m_2},\ldots,\langle X\rangle_{m_l}) \\ Y \overset{(m_1,m_2,\ldots,m_l)}{\longleftrightarrow} & (\langle Y\rangle_{m_1},\langle Y\rangle_{m_2},\ldots,\langle Y\rangle_{m_l}) \\ \hline Z=X\square Y \overset{(m_1,m_2,\ldots,m_l)}{\longleftrightarrow} & (\langle X\square Y\rangle_{m_1},\langle X\square Y\rangle_{m_2},\ldots,\langle X\square Y\rangle_{m_l},) \end{array}$$

由此可以看出，RNS 算法是“对偶”定义的。$Z=X\square Y \bmod M$ 的 L 个元素是在 L 个短整型字长 mod(m_l)信道中同时计算的，信道的字宽是受 $w_l=[\log_2(m_l)]$位限制的(通常是 4 到 8 位)。在实际应用中，大多数 RNS 运算系统中使用的是小型 RAM 或者 ROM 表来实现模映射 $z_l=x_l\square y_l \bmod m_l$。

例 2.7：　RNS 算法

下面来研究一下基于相关质数模集{2,3,5}的 RNS 数制，它具有 $M=2\cdot3\cdot5=30$ 的动态范围。在 Z_{30} 中有两个整数，分别是 7_{10} 和 4_{10}。7 和 4 的 RNS 表达式分别为 $7=(1,1,2)_{RNS}$ 和 $4=$

$(0,1,4)_{RNS}$。还有，两者的和、差、积分别是 11，3 和 28，它们也在 Z_{30} 中，其计算如下：

$$\begin{array}{lll} 7 \overset{(2,3,5)}{\longleftrightarrow} (1,1,2) & 7 \overset{(2,3,5)}{\longleftrightarrow} (1,1,2) & 7 \overset{(2,3,5)}{\longleftrightarrow} (1,1,2) \\ +4 \overset{(2,3,5)}{\longleftrightarrow} +(0,1,4) & -4 \overset{(2,3,5)}{\longleftrightarrow} -(0,1,4) & \times 4 \overset{(2,3,5)}{\longleftrightarrow} \times (0,1,4) \\ \hline 11 \overset{(2,3,5)}{\longleftrightarrow} (1,2,1) & 3 \overset{(2,3,5)}{\longleftrightarrow} (1,0,3) & 28 \overset{(2,3,5)}{\longleftrightarrow} (0,1,3) \end{array}$$

RNS 数制已经在定制的 VLSI 元器件[36]、GaAs 和 LSI[35]中得到应用。目前已经知道的是，在 Xilinx XC4000 系列 FPGA 中，RNS 针对短整型专门提供了 $2^4\times2$ 位的表，对提高速度有显著作用[37]。对于较宽字长的模运算，Altera FLEX 的 $2^8\times8$ 位表可以令 RNS 算法和 RNS 到整数转换的设计受益匪浅。有了支持较宽字长模运算的能力，设计高精度高速的 FPGA 系统就可以成为现实了。

过去实现具有实际价值 RNS 系统的最大障碍到目前为止已经成功地被破解了[38]。实现 RNS 到整数的解码、除法或者是绝对值缩放，首先都需要将数据从 RNS 格式转换成整数。通常所使用的转换理论被称为中国余数定理(Chinese Remainder Theorem, CRT)和混和基数转换(Mixed-radix conversion, MRC)算法[34]。实际上，MRC 是生成整数加权数制表达式的数位，而 CRT 直接给出了 RNS 到 L-数组的一个映射。CRT 定义如下：

$$X \bmod M \equiv \sum_{l=0}^{L-1} \hat{m}_l < \hat{m}_l^{-1} x_l >_{m_l} \quad \bmod M \tag{2.12}$$

其中 $\hat{m}_l = M/m_l$ 是整数，而 $\hat{m}_l^{-1}$ 是 $\hat{m}_l \bmod m_l$ 的相反数，也就是说 $\hat{m}_l\ \hat{m}_l^{-1} \equiv 1 \bmod m_l$。通常设计所需要的 RNS 计算的输出要远远小于最大动态范围 M。在这些场合有一种称为 ε-CRT[39]的非常有效算法，可以用来实现即时即地的 RNS 到整数的转换。

6. 索引乘法器

实际上，RNS 有好几种变化。经常使用的一种是基于“索引”算法的[34]。它在某些方面与对数算法类似。索引域中的计算是根据是否所有的模都是质数这一事实来进行的，根据数论可知：存在一个本原元素，一个生成元 g，也就是：

$$a \equiv g^{\alpha} \bmod p \tag{2.13}$$

这一公式可以生成 Z_p 域内除零之外的所有元素(表示为 $Z_p/\{0\}$)。也就是说，实际上在 $Z_p/\{0\}$ 中的整数 a 和 Z_{p-1} 域中的指数之间存在一一对应的映射。而索引 α 与生成元 g 和整数 a 之间的关系可以表述成 $\alpha = \mathrm{ind}_g(a)$。

例 2.8：　索引编码

考虑一个质数模 p=17，生成元 g=3 将要生成 $Z_p/\{0\}$ 域中的元素。译码表如下所示。为了计数的需要，特例 a=0 表示成 $g^{-\infty}$=0。

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathrm{ind}_3(a)$	-∞	0	14	1	12	5	15	11	10	2	3	7	13	4	9	6	8

RNS 数的乘法运算可以按如下规则进行：

(1) 映射 a 和 b 到索引域，也就是：$a=g^{\alpha}$ 和 $b=g^{\beta}$

(2) 将索引数值模 $p-1$ 相加，也就是：$v=(\alpha+\beta)\mathrm{mod}(p-1)$

(3) 将所得的和映射回到初始索引域内，也就是：$n=g^{v}$

如果数据是以索引形式处理的，则只需要将 $\mathrm{mod}(p-1)$指数相加即可。接下来利用下面这个例子加以说明。

例 2.9：索引乘法

考虑一个质数模 p=17，生成元 g=3，例 2.8 中已经给出了结果。a=2 与 b=4 相乘的运算过程如下：

$$(\mathrm{ind}_g(2)+\mathrm{ind}_g(4))\mathrm{mod}16=(14+12)\mathrm{mod}16=10$$

由例 2.8 中的表可以看到 $\mathrm{ind}_g(8)=10$，正好对应整数 8，也就是所期望的结果。

7. 索引域中的加法

在大多数情况下，DSP 算法既需要乘法也需要加法。索引算法能够很好地解决乘法运算，但是对加法就没有什么价值了。在技术上，可以将索引 RNS 数据转换回进行加法运算比较简单的 RNS 的方法来进行加法运算。一旦结果计算出来后再映射回到索引域中。另一种方法就是根据 Zech 对数。索引编码数 a 和 b 的和表达如下：

$$d=a+b=g^{\delta}=g^{\alpha}+g^{\beta}=g^{\alpha}(1+g^{\beta-\alpha})=g^{\beta}(1+g^{\alpha-\beta}) \tag{2.14}$$

现在我们将 Zech 对数定义成：

定义 2.10：Zech 对数

$$Z(n)=\mathrm{ind}_g(1+g^n) \quad \longleftrightarrow \quad g^{Z(n)}=1+g^n \tag{2.15}$$

接下来按下面的形式重写(2.14)：

$$g^{\delta}=g^{\beta}\cdot g^{Z(\alpha-\beta)} \quad \longleftrightarrow \quad \delta=\beta+Z(\alpha-\beta) \tag{2.16}$$

相加是在索引域中进行的，所以需要一次加法、一次减法和一次 Zech LUT。接下来的一个小例题就说明了在索引域中 2+5 相加的法则。

例 2.11：Zech 对数

质数模 p=17 和 g=3 的 Zech 对数表如下：

N	-∞	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Z(n)	0	14	12	3	7	9	15	8	13	-∞	6	2	10	5	4	1	11

表中 2 和 5 的指数值的定义可以参照例 2.8，计算过程如下：

$$2+5=3^{14}+3^{5}=3^{5}(1+3^{9})=3^{5+Z(9)}=3^{11}\equiv 7\mathrm{mod}17$$

需要特别注意的是特例 $a+b\equiv 0$，与之对应的是[40]：

$$-X\equiv Y\mathrm{mod}\,p \quad \longleftrightarrow \quad g^{\alpha+(p-1)/2}\equiv g^{\beta}\mathrm{mod}\,p$$

也就是说，在索引域中当和等于零时，$\beta=\alpha+(p-1)/2 \bmod (p-1)$。下面是一个例题。

例 2.12：　5 和 12 在原始域中的加法运算过程如下：

$$5+12=3^5+3^{13}=3^5(1+3^8)=3^{5+Z(8)} \equiv 3^{-\infty}\equiv 0 \bmod 17$$

8. 利用 QRNS 计算复数乘法

如果我们处理复数数据，就会看到 RNS 会产生另一种有趣的性质。这种称为 QRNS 的表达式能够非常有效地计算乘法，接下来我们就将讨论这个问题。

当用 RNS 数字对复数的实部和虚部进行编码时，最终的数制就称为复数 RNS，或者是 CRNS。CRNS 中的复数加法需要执行两次实数相加。复数 RNS(CRNS)乘法是按四个实数乘积、一个加法和一个减法的形式定义的。当使用一种称作二次 RNS(也称为 QRNS)的 RNS 变量时，情况就从根本上发生了变化。QRNS 是建立在 $p=4k+1$ 形式的高斯质数已知特性的基础上的，其中 k 是正整数。模的这一选择的重要性来源于 Zp 中多项式 x^2+1 的因式分解。这一多项式具有两个根：j 和 $-j$，其中 j 和 $-j$ 是属于余数类 Zp 的实整数。这与复数域内的 x^2+1 的因式分解形式形成了鲜明的对比。在复数域中，根是复数形式的：$x_{1,2}=\alpha \pm j\beta$，其中 $j=(-1)^{1/2}$ 是虚部操作符。将一个 CRNS 数转换成 QRNS 数是通过变换 f：$Zp^2 \rightarrow Zp^2$ 来实现的，过程如下：

$$f(a+jb)=((a+jb) \bmod p,(a-jb) \bmod p)=(A,B) \tag{2.17}$$

在 QRNS 中，加法和乘法是按组件形式实现的，其定义如下：

$$(a+jb)+(c+jd) \leftrightarrow (A+C,B+D) \bmod p \tag{2.18}$$

$$(a+jb)(c+jd) \leftrightarrow (AC,BD) \bmod p \tag{2.19}$$

绝对值的平方可以按如下方式计算：

$$|a+jb|^2 \leftrightarrow (A\cdot B) \bmod p \tag{2.20}$$

从 QRNS 到 CRNS 的反演映射定义如下：

$$f^{-1}(A,B)=2^{-1}(A+B)+j(2j)^{-1}(A-B) \bmod p \tag{2.21}$$

取高斯质数 $p=13$，$(a+jb)=(2+j1)$和$(c+jd)=(3+j2)$的复数乘积是$(2+j1)(3+j2)=(4+j7) \bmod 13$。在该情况下，需要四次实数乘法、一次实数加法和一次实数减法来完成这一乘积。

例 2.13：QRNS 乘法

二次方程 $x^2\equiv(-1) \bmod 13$ 有两个根 $j=5$ 和 $-j=-5\equiv 8 \bmod 13$。QRNS 数据就变成：

$$(a+jb)=2+j \leftrightarrow (2+5\cdot 1,2+8\cdot 1)=(A,B)=(7,10) \bmod 13$$

$$(c+jd)=3+j2 \leftrightarrow (3+5\cdot 2,3+8\cdot 2)=(C,D)=(0,6) \bmod 13$$

组件形式的乘法结果$(A,B)(C,D)=(7,10)(0,6)\equiv(0,8) \bmod 13$ 只需要两次实数乘法。按(2.21)定义的到 CRNS 的反演映射，其中 $2^{-1}\equiv 7$，$(2j)^{-1}=10^{-1}\equiv 4$。解方程 $2\cdot x\equiv 1 \bmod 13$ 和 $10\cdot x\equiv 1 \bmod 13$，分别得到 7 和 4，接下来是：

$$f^{-1}(0,8)=7(0+8)+j4(0-8) \bmod 13\equiv 4+j7 \bmod 13 \quad \surd$$

图 2-5 给出了 CRNS 和 QRNS 之间映射关系的图解。

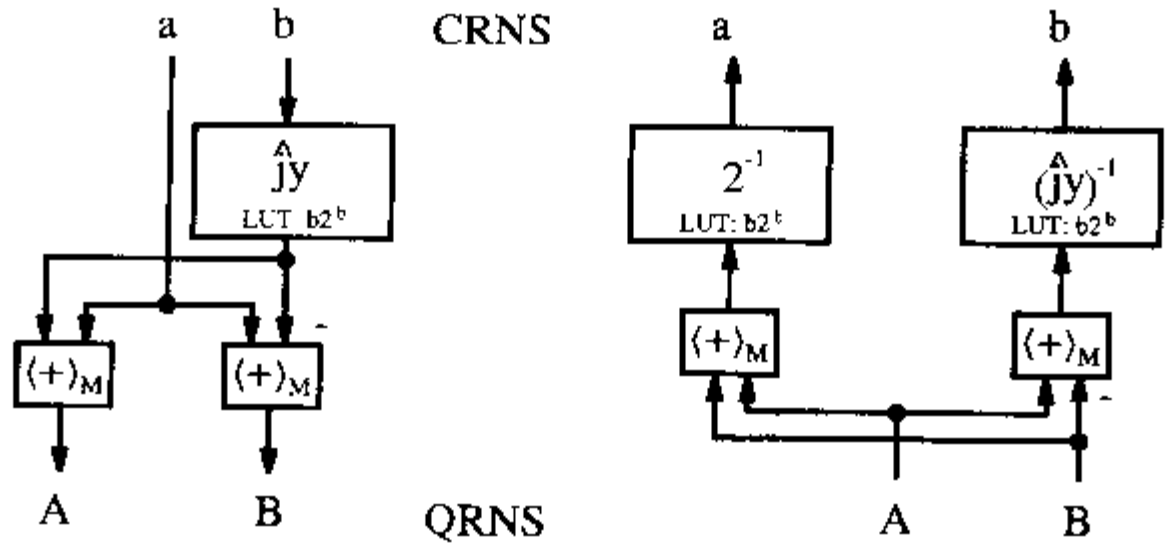


图 2-5　CRNS↔QRNS 的转换

2.2.3　浮点数

浮点数制可以在更大的动态范围内提供更高的分辨率，通常当定点数由于受其精度和动态范围所限不能胜任时，浮点数就能够为之提供解决方案。当然，同时也在速度和复杂程度方面带来了损失，大多数浮点数制都遵循单精度或者双精度的 IEEE 浮点数标准[41]。标准浮点数字长由一个符号位 S、指数 e 和无符号(小数)的规格化尾数 m 构成。其格式如下：

S	指数 e	无符号尾数 m

浮点数字长的代数表达式形式如下：

$$X=(-1)^S\cdot 1.m\cdot 2^{e\text{-bias}} \tag{2.22}$$

指数 $e=1\ldots1_2$ 是为∞预备的，而 $e=0$ 则是为 0 预备的。IEEE 单精度和双精度格式的其他参数请参阅表 2-4。

表 2-4　IEEE 浮点数标准

	单　精　度	双　精　度
字长	32	64
尾数	23	52
指数	8	11
偏移 Bias	127	1023
范围	$2^{\pm 38}\approx 3.8\times 10^{38}$	$2^{\pm 1024}\approx 9\times 10^{307}$

在浮点数乘法中，尾数部分可以像定点数一样相乘，而把指数部分相加。浮点数减法通常会更加复杂一些，这是因为首先要将尾数归一化，就是要将两个数都调整到较大的指数。然后再将两个尾数相加。对于加法和乘法混和运算来说，最终的归一化，就是将结果尾数再统一成小数 1.m 形式的表达式，这是非常必要的。

单精度 IEEE 浮点数格式之所以使用 FPGA 的资源，主要是由于 23×23 位快速整数乘法器

的缘故。为此，Shirazi 等人[42]已经开发了一种修正格式，可以在其定制的计算设备上实现多种算法，这一计算设备名为 SPLASH-2，是一种基于 Xilinx XC4010 元器件的多功能 FPGA 板(请参阅表 2-5)。他们使用了一种 18 位格式，这样就可以在多功能 FPGA 板的 36 位宽的系统总线上传输两位操作数。这一 18 位格式具有 10 位尾数、7 位指数和一个符号位，表示范围是 3.7×10^{19}。可以在 3 个流水线级履行一个加法器和一个乘法器的功能。有关单精度浮点数乘法器-加法器的参数请参阅表 2-5。

表 2-5　采用 Xilinx XC4010 定制 18 位浮点数设计

	加　法　器	乘　法　器
函数生成器(也就是 LE)	224	352
触发器	112	112
速度	8.6 MHz	4.9 MHz

2.3　二进制加法器

一个基本 N 位二进制加法器/减法器由 N 个全加器(full adder, FA)组成。每个全加器都执行如下的布尔方程：

$$s_k = x_k \quad \text{XOR} \quad y_k \quad \text{XOR} \quad c_k \tag{2.23}$$

$$= x_k \oplus y_k \oplus c_k \tag{2.24}$$

这就定义了和的位。进位位按如下方法计算：

$$c_{k+1} = (x_k \quad \text{AND} \quad y_k)\text{OR}(x_k \quad \text{AND} \quad c_k)\text{OR}(y_k \quad \text{AND} \quad c_k) \tag{2.25}$$

$$= (x_k \cdot y_k) + (x_k \cdot c_k) + (y_k \cdot c_k) \tag{2.26}$$

仅就 2C 加法器来讲，最低有效位可以减少到一个半加器，因为进位位输入是 0。

最简单的加法器结构称为并行加法器，如图 2-6(a)所示，是位串行格式的。如果在 FPGA 中有较大的表，几个位就可以安置在一个 LUT 之中，如图 2-6(b)所示。对于这种“一次两位”加法器来讲，最长的延迟来自进位的脉冲通过所有阶段的时候。目前已经采取了许多技术来缩短这一进位延迟，比如：跳跃进位、先行进位、条件和进位选择加法器。这些技术都能够提高加法的速度，可以用在老一代 FPGA 系列之中(例如：Xilinx 的 XC3000)，因为这些元器件本身没有提供内部快速进位逻辑。现代的系列，例如 Xilinx 的 XC4K 或者 Altera 的 Flex，都具有特别快的“脉冲进位逻辑”，比通过常规逻辑 LUT 的延迟要快得多[1]。Altera 采用的是快速表(请参阅图 1-12)，而 Xilinx 的 XC4K 则是采用硬连线译码器，根据图 2-7 所示的多级信道结构来实现进位逻辑。快速进位逻辑在现代 FPGA 系列中的出现，消除了在开发硬件中集成先行进位电路的必要。

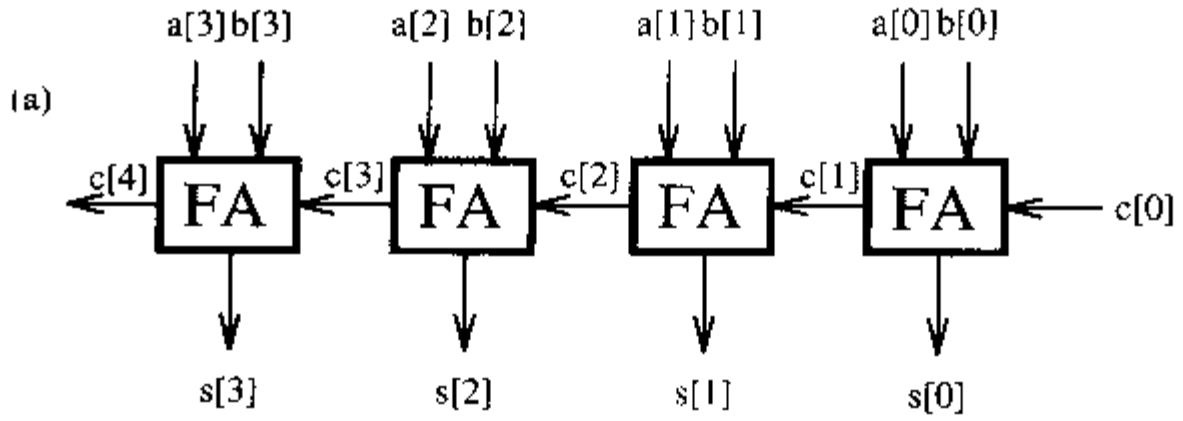


图 2-6　二进制补码加法器

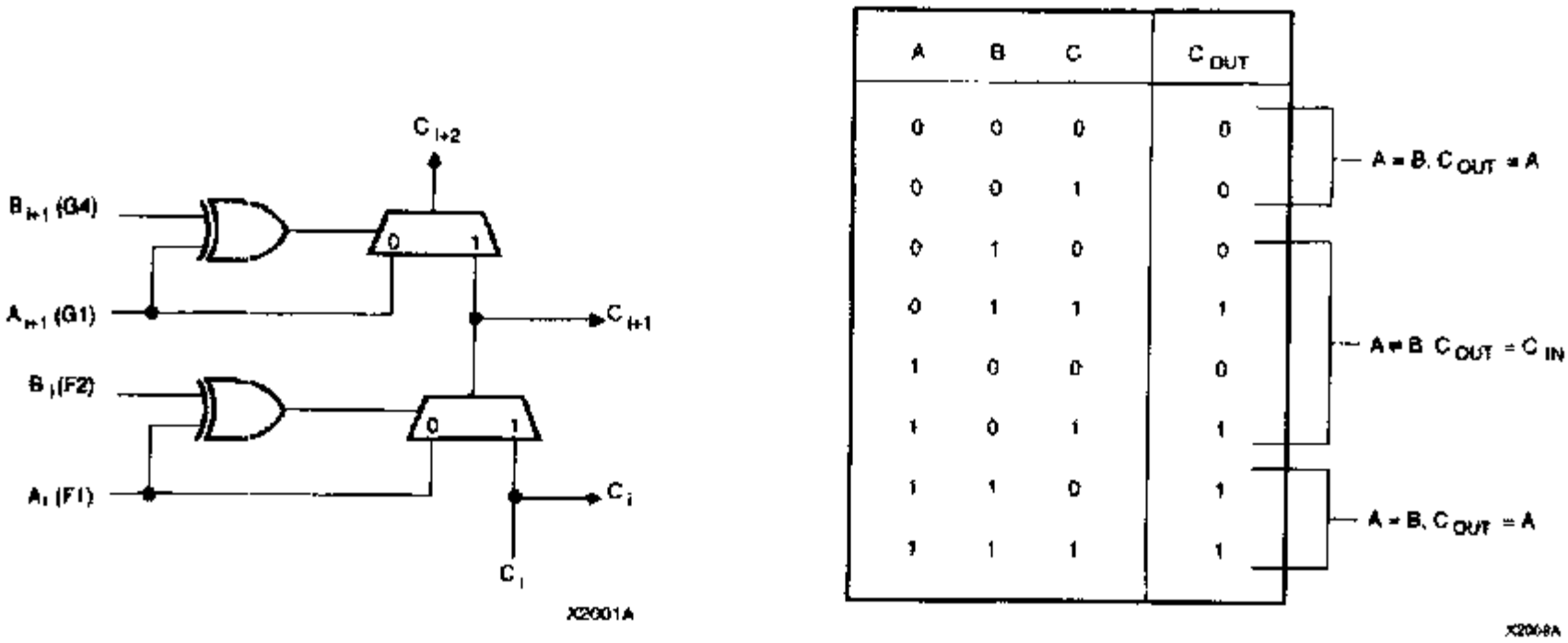


图 2-7　XC4000 快速进位逻辑(©1993 Xilinx)

图 2-8 总结了用 lpm_add_sub 兆函数组件实现的 *N* 位二进制加法器的规模和 Registered Performance。如果是通过 I/O 单元接收操作数，则通过 FLEX 元器件总线的延迟占主要部分，如果采用 I/O 单元寄存器就会显著降低性能(选择项：Assign|Global Project Logic Synthesis|Automatic Fast I/O)。如果数据是从局部寄存器传送过来的，性能就会提高。当由于三个因素之一需要增加 LC 的使用时，这类附加的 LC 寄存器分配就会在项目报告文件中出现。同步的已注册设计是不需要任何额外资源的。典型的设计将会达到这两个实例之间的速度。

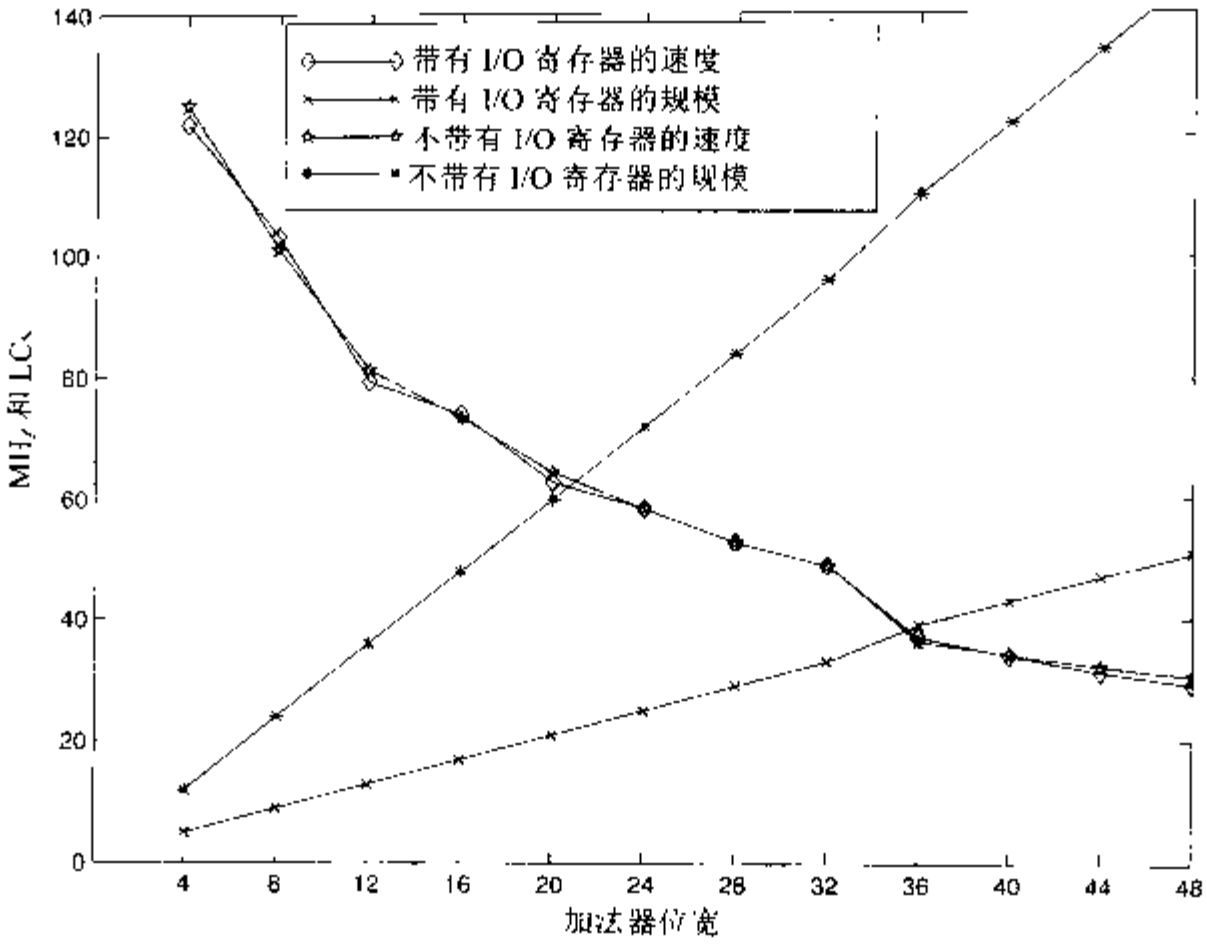


图 2-8　Flex 10K 的加法器速度和规模

2.3.1　流水线加法器

由于 DSP 算法的内部数据流规则，就决定了在 DSP 解决方案中流水线技术会得到广泛应用。典型的可编程数字信号处理器 MAC[14, 15, 6] 至少带有 4 个流水线级。处理器同时进行：

(1) 对指令译码

(2) 将操作数下载到寄存器中

(3) 执行乘法并存储乘积

(4) 累加乘积

流水线规则也可以应用在 FPGA 的设计之中。只需要极少或者根本不需要额外的成本，因为每一个逻辑元件都包括一个触发器，这个触发器或者没有用到，或者是用来存储布线资源。采用流水线有可能将一个算术操作分解成一些小规模的基本操作，将进位和中间值存储在寄存器中，并在下一个时钟周期内继续运算。在文献中这样的加法器通常称为“**进位存储加法器**”(carry save adder，CSA)[1]。这样，问题就出现了，我们应该将加法器分成多少部分呢？我们应不应该使用位级呢？对于 Altera 的 Flex10K 元器件来讲，一个合理的选择就是采用一个带有 8 个 LC 和 8 个 FF 的 LAB 组成一个流水线组件。举例说明一下就可以看出：比如构成一个 5 位流水线加法器，由于 5 位流水线加法器不是配置在一个 LAB 之中的，所以具体性能参数请参阅表 2-6。

表 2-6　采用带有流水线选择的预定义 LPM 模块合成的 5 位流水线加法器的性能

流 水 线 级	使用 I/O 寄存器的		没有使用 I/O 寄存器的	
	MHz	LC	MHz	LC
0	123.45	6	120.48	15
1	121.95	10	123.45	18

注 1：进位存储加法器这一名称也用在 Wallance 乘法器中，请参阅练习题 2.1。

(续表)

流 水 线 级	使用 I/O 寄存器的		没有使用 I/O 寄存器的	
	MHz	LC	MHz	LC
2	112.35	14	123.45	24
3	112.35	20	120.45	32
4	112.35	28	123.45	41
5	103.91	31	121.95	45

由于在一个 LAB 中触发器的数量是 8，并且我们需要一个额外的触发器作为进位输出，为了获得最大的 Registered Performance，我们应该采用最大的 7 位模块规模。

只有具有最高有效位的模块才可以是 8 位字宽的，因为我们不再需要为进位提供额外的触发器。由此可以导出以下结论：

(1) 采用一个额外的流水线级，我们就能够构造一个达到 7+8=15 位字长的加法器。

(2) 采用两个流水线级，我们就能够构造一个达到 7+7+8=22 位字长的加法器。

(3) 采用三个流水线级，我们就能够构造一个达到 7+7+7+8=29 位字长的加法器。

表 2-7 给出了这种流水线加法器的 Registered Performance 和 LC 的利用情况。从表 2-7 中可以看出：如果我们统计一下流水线级的大概数量就会发现，尽管位宽提高了，但是 Registered Performance 还是几乎保持不变。

表 2-7　流水线加法器的性能。最大位宽为 15 位、22 位和 29 位加法器的规模和速度

位　　宽	使用 I/O 寄存器的		没有使用 I/O 寄存器的		流 水 线 级	设计文件名称
	MHz	LC	MHz	LC		
9-15	97.08	26	94.33	61	1	add_1p.vhd
16-22	94.33	58	91.74	113	2	add_2p.vhd
23-29	91.74	105	90.90	180	3	add_3p.vhd

下面的例子给出了一个 15 位流水线加法器的代码，其图形化解释请参阅图 2-9。

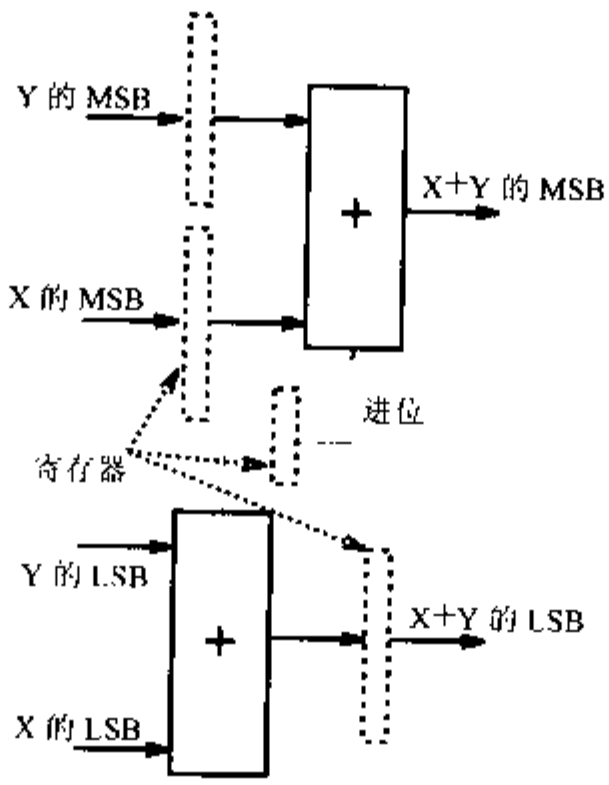


图 2-9　流水线加法器

例 2.14：15 位流水线加法器的 VHDL 设计

下面来研究一下图 2-9 中给出图形化解释的 15 位流水线加法器的 VHDL 代码[2]。根据合成类型，这一设计运行的速度在 90 到 104MHz 之间。

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY add_1p IS
  GENERIC (WIDTH  : INTEGER := 15;  -- Total bit width
           WIDTH1 : INTEGER := 7;   -- Bit width of LSBs
           WIDTH2 : INTEGER := 8;   -- Bit width of MSBs
           ONE    : INTEGER := 1);  -- 1 bit for carry reg.
  PORT (x,y  : IN   STD_LOGIC_VECTOR(WIDTH - 1 DOWNTO 0);
                                                  -- Inputs
        sum : OUT STD_LOGIC_VECTOR(WIDTH - 1 DOWNTO 0);
                                                  -- Result
        clk  : IN   STD_LOGIC);
END add_1p;

ARCHITECTURE flex OF add_1p IS
  SIGNAL l1, l2, r1, q1                 -- LSBs of inputs
                        : STD_LOGIC_VECTOR(WIDTH1 - 1 DOWNTO 0);
  SIGNAL l3, l4, r2, q2, u2, h2         -- MSBs of inputs
                        : STD_LOGIC_VECTOR(WIDTH2 - 1 DOWNTO 0);
  SIGNAL s              : STD_LOGIC_VECTOR(WIDTH - 1 DOWNTO 0);
                                        -- Output register
  SIGNAL cr1, cq1       : STD_LOGIC_VECTOR(ONE - 1 DOWNTO 0);
                                        -- LSBs carry signal


BEGIN
  PROCESS   -- Split in MSBs and LSBs and store in registers
  BEGIN
    WAIT UNTIL clk = '1';
    -- Split LSBs from input x,y
    FOR k IN WIDTH1 - 1 DOWNTO 0 LOOP
      l1(k) <= x(k);
      l2(k) <= y(k);
    END LOOP;
    -- Split MSBs from input x,y
```

注 2：这一例子相应的 Verilog 代码文件 add_1p.v 可以在附录 A 中找到。

```
  FOR k IN WIDTH2 - 1 DOWNTO 0 LOOP
    l3(k) <= x(k+WIDTH1);
    l4(k) <= y(k+WIDTH1);
  END LOOP;
END PROCESS;
  ---------- First stage of the adder ----------
add_1: lpm_add_sub                    -- Add LSBs of x and y
        GENERIC MAP ( LPM_WIDTH => WIDTH1,
                      LPM_REPRESENTATION => "UNSIGNED",
                      LPM_DIRECTION => "ADD")
        PORT MAP ( dataa => l1, datab => l2,
                   result => r1,   cout => cr1(0));
reg_1: lpm_ff                 -- Save LSBs of x+y and carry
        GENERIC MAP ( LPM_WIDTH => WIDTH1 )
        PORT MAP ( data => r1, q => q1,clock => clk );
reg_2: lpm_ff
        GENERIC MAP ( LPM_WIDTH => ONE )
        PORT MAP ( data => cr1, q => cq1, clock => clk );

add_2: lpm_add_sub                    -- Add MSBs of x and y
        GENERIC MAP ( LPM_WIDTH => WIDTH2,
                      LPM_REPRESENTATION => "UNSIGNED",
                      LPM_DIRECTION => "ADD")
        PORT MAP (dataa => l3, datab => l4, result => r2);
reg_3: lpm_ff                         -- Save MSBs of x+y
        GENERIC MAP ( LPM_WIDTH => WIDTH2 )
        PORT MAP ( data => r2, q => q2, clock => clk );
---------- Second stage of the adder ----------
-- One operand is zero
h2 <= (OTHERS => '0');

-- Add result from MSBs (x+y) and carry from LSBs
add_3: lpm_add_sub
        GENERIC MAP ( LPM_WIDTH => WIDTH2,
                      LPM_REPRESENTATION => "UNSIGNED",
                      LPM_DIRECTION => "ADD")
        PORT MAP ( cin => cq1(0), dataa => q2,
                   datab => h2, result => u2 );

PROCESS                       -- Build a single registered output
BEGIN                         -- word of WIDTH=WIDTH1+WIDHT2
  WAIT UNTIL clk = '1';
  FOR k IN WIDTH1 - 1 DOWNTO 0 LOOP
    s(k) <  q1(k);
  END LOOP;
  FOR k IN WIDTH2 - 1 DOWNTO 0 LOOP
    s(k+WIDTH1) <= u2(k);
```

```
    END LOOP;
  END PROCESS;

  sum <= s ;      -- Connect s to output pins
END flex;
```

这一 15 位流水线加法器的仿真结果如图 2-10 所示。注意：140 和 130 的加法生成了一个来自低 7 位加法器的进位，但是 120+5=125<127 没有产生进位。

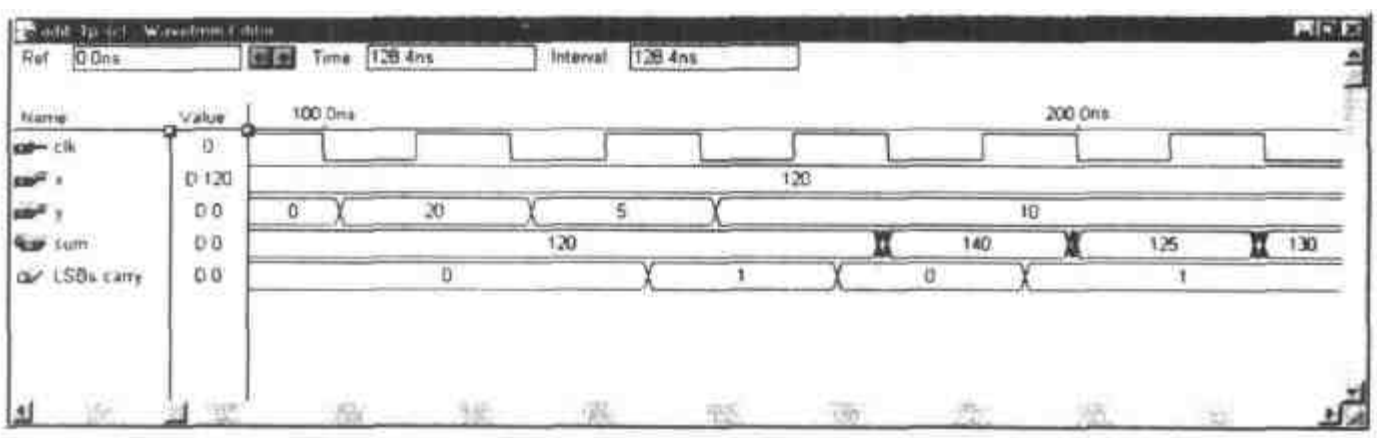


图 2-10　流水线加法器的仿真结果

2.3.2　模加法器

模加法器是 RNS-DSP 设计中最重要的组成部件。它既可以用于加法，也可以通过索引算法用于乘法。在下面的讨论中我们将要为 FPGA 描述一些设计方案。

现在有很多种模加法的设计[43]。图 2-11 所示的设计仅仅采用了 LE，对于 FPGA 是可行的。Altera 的 FLEX 元器件包含少量 2KB 的 ROM 或者是 RAM(EAB)，可以构造成 $2^8\times8$，$2^9\times4$，$2^{10}\times2$ 或者是 $2^{11}\times1$ 的表，用于模 m_i 校正。下面的表给出了 6、7 和 8 位模加法器的规模和 Registered Performance[44]。

流水线级数		位　数		
		6	7	8
MPX	0	41.3MSPS 27LE	46.5MSPS 31LE	33.7MSPS 35LE
MPX	2	76.3MSPS 16LE	62.5MSPS 18LE	60.9MSPS 20LE
MPX	3	151.5MSPS 27LE	138.9MSPS 31LE	123.5MSPS 35LE
ROM	3	86.2MSPS 7LE 1EAB	86.2MSPS 8LE 1EAB	86.2MSPS 9LE 2EAB

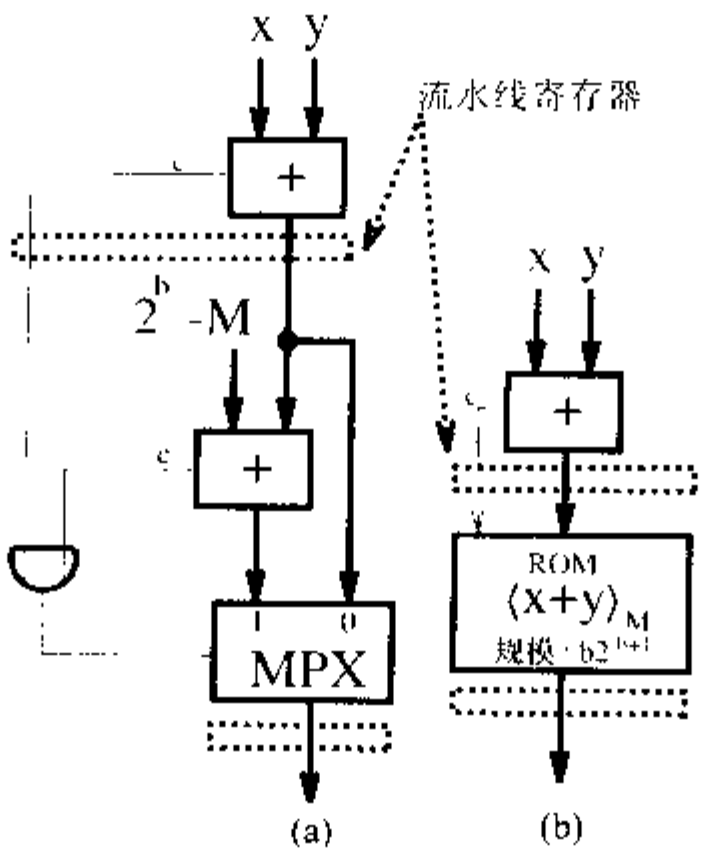


图 2-11　模加法 (a) MPX-ADD 和 MPX-ADD 流水线 (b) ROM 流水线

尽管图 2-11 所示的 ROM 提供了很高的速度，但是 ROM 本身产生了一个 4 周期的流水线延迟，而且 ROM 的数量也是有限的。此外，ROM 对于前面所讨论过的缩放表格比例是强制性的。多级加法器(multiplexed-adder, MPX-Add)的速度相对较慢，即使在每列都加上一个进位链也是如此。流水线方式通常需要相等数量的 LE 作为非流水线方式，而且速度要求大约是二倍。最大吞吐能力是在 6 位流水线信道中的两个模块上实现加法器的时候。

2.4　二进制乘法器

两个 N 位二进制数的乘积用 X 和 $A=\sum_{k=0}^{N-1}a_k 2^k$ 表示，按“手工计算”的方法给出就是：

$$P=A\cdot X=\sum_{k=0}^{N-1}a_k 2^k X \tag{2.27}$$

从中可以看出，只要 $a_k\neq 0$，输入 X 就随着 k 的位置连续地变化，然后累加 $X2^k$。如果 $a_k=0$，相应的转换相加就可以忽略了(也就是空操作 nop)下面的 VHDL 例子就是采用了这种“手工计算”方案来进行两个 8 位整数相乘。

例 2.15　8 位乘法器

8 位乘法器的 VHDL 代码[2]如下。乘法的执行分 3 个阶段完成。首先是下载 8 位操作数并且重置乘积寄存器。在第二阶段 s1 中，进行实际的串行-并行乘法运算。在第三步 s2 中，乘积被传输到输出寄存器 y。

注 2：这一例子相应的 Verilog 代码文件 mul_ser.v 可以在附录 A 中查到。

```
PACKAGE eight_bit_int IS                    -- User defined types
  SUBTYPE BYTE IS integer RANGE - 128 TO 127;
  SUBTYPE TWOBYTES IS integer RANGE   32768 TO 32767;
END eight_bit_int;

LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY ieee;                          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY mul_ser IS                                  ------> Interface
  PORT ( clk   : IN   STD_LOGIC;
         x     : IN   BYTE;
         a     : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
         y     : OUT TWOBYTES);
END mul_ser;

ARCHITECTURE flex OF mul_ser IS

  TYPE STATE_TYPE IS (s0, s1, s2);
  SIGNAL state      : STATE_TYPE;

BEGIN

  States: PROCESS       ------> Multiplier in behavioral style
    VARIABLE   p, t   : TWOBYTES;              -- Double bit width
    VARIABLE count    : integer RANGE 0 TO 7;
  BEGIN
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN s0 =>              -- Initialization step
        state <= s1;
        count := 0;
        p : = 0;              -- Product register reset
        t  : = x;             -- Set temporary shift register to x
      WHEN s1 =>              -- Processing step
        IF count = 7 THEN -- Multiplication ready
          state <= s2;
          ELSE
          IF a(count) = '1' THEN
            p := p + t;       -- Add 2^k
          END IF;
```

```
            t := t * 2;
            count := count + 1;
            state <= s1;
          END IF;
        WHEN s2 =>          -- Output of result to y and
          y <= p;           -- start next multiplication
          state <= s0;
      END CASE;
    END PROCESS States;

END flex;
```

图 2-12 给出了 13 和 5 相乘的仿真结果。寄存器 t 给出了 5，10，20...序列的部分乘积。因为 $13_{10}=00001101_{2C}$，在乘积的最终结果 65 中，乘积寄存器 p 只被更新了 3 次。在状态 s2 中，将结果 65 传输到乘法器的输出 y 中。

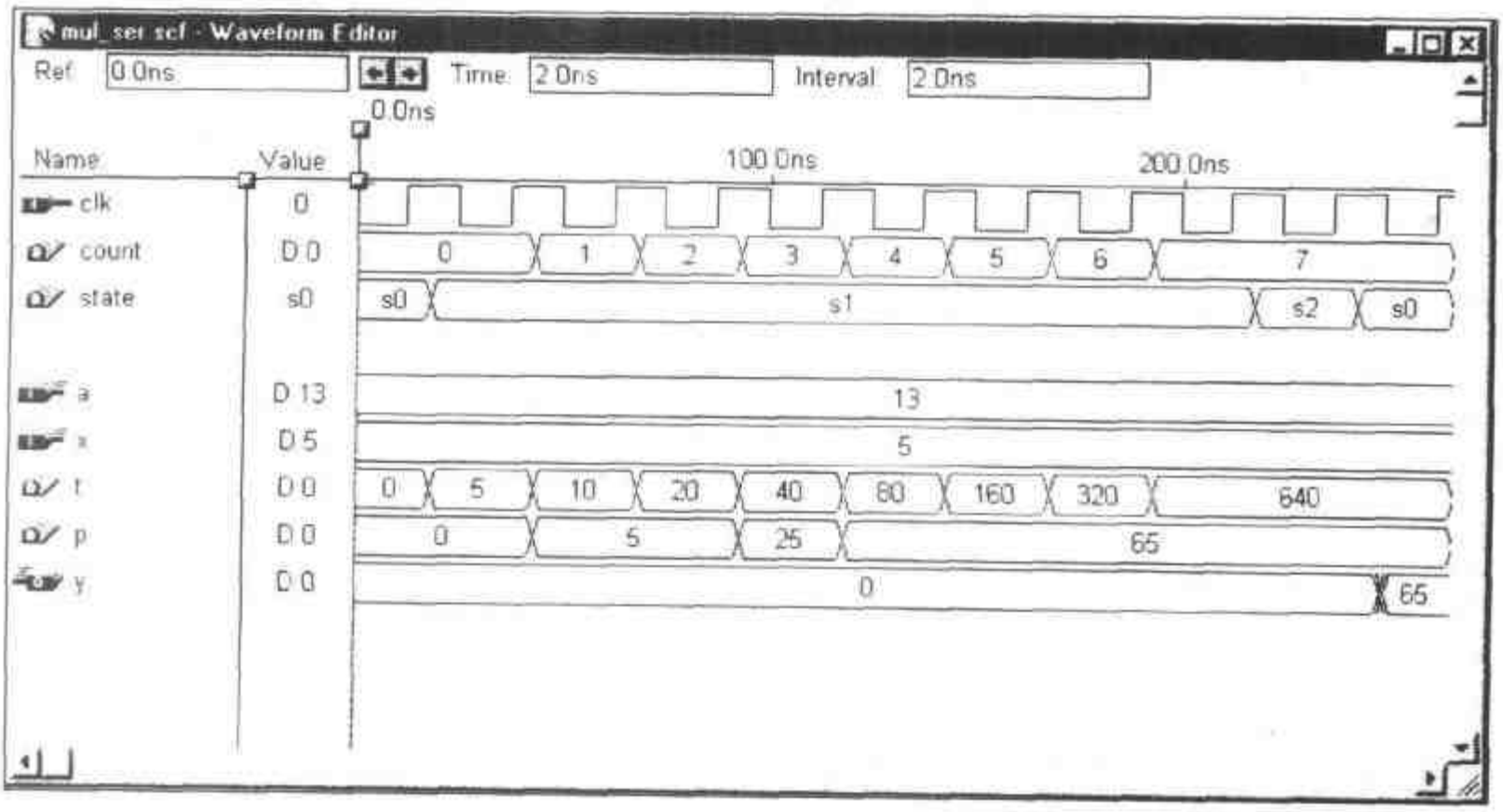


图 2-12　移位加法乘法器的仿真结果

由于第一个操作数是并行形式的(也就是 X)，而第二个操作数 A 是逐位形式的，所以我们刚才描述的乘法器就称为**串行/并行乘法器**。如果两个操作数都是串行的，那么这一结构称为**串行/串行乘法器**[45]。这样的乘法器只需要一个全加器，但是串行/串行乘法器的等待时间为大 $O(N^2)$，因为状态机需要大约 N^2 个周期。

另一种方法就是通过增加复杂性来换取速度，称之为“**阵列**”，或者是**并行/并行乘法器**。图 2-13 给出了一个 4×4 位阵列乘法器。注意两个操作数都是并行提交给 N^2 个加法器单元的加法器阵列的。

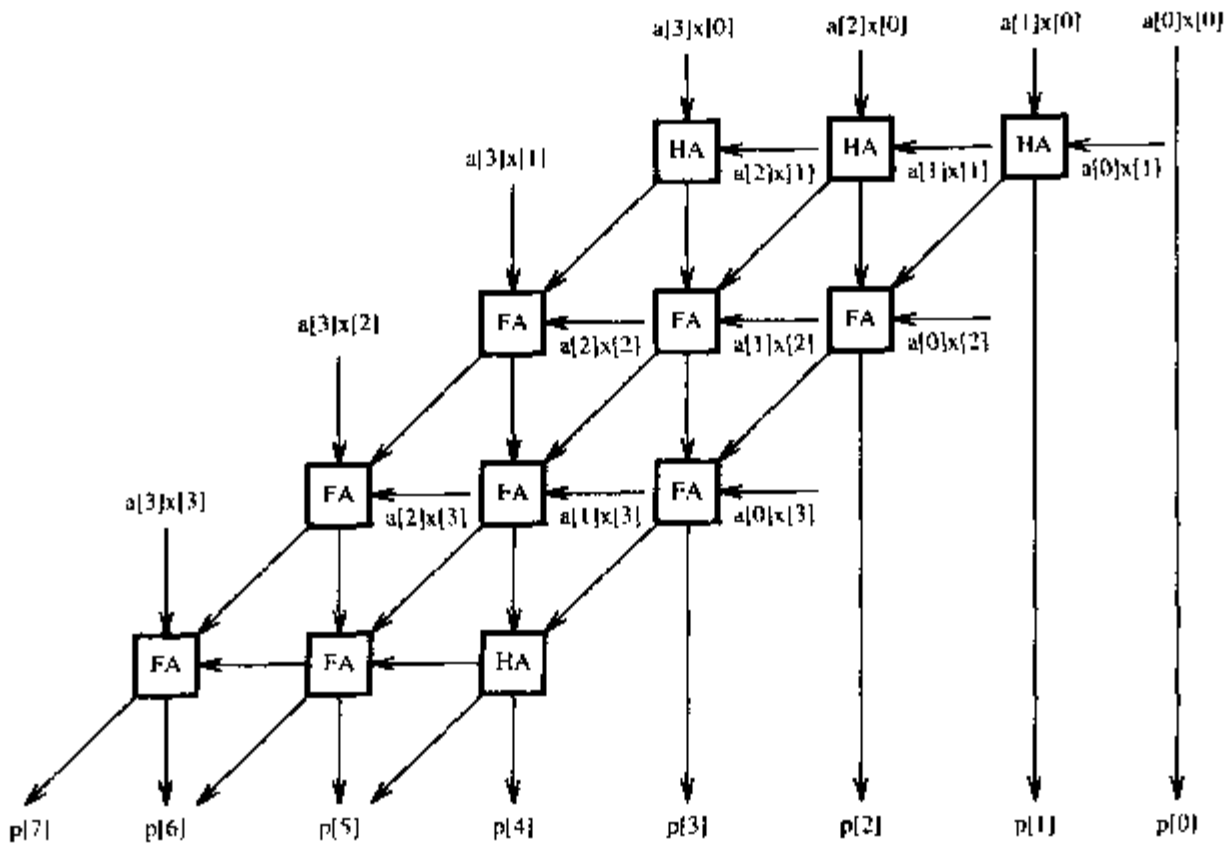


图 2-13　4 位阵列乘法器

如果完成进位与和累加所需要的时间相同的话，这一方法就是可行的。但是对于现在的 FPGA 来讲，进位计算执行的速度要比和累加的速度快，而且另一种结构对于 FPGA 来讲效率更高。这一阵列乘法器的思想如图 2-14 所示，是一个 8×8 位乘法器。这一结构在第一步就将两个相邻的部分乘积 $a_n X2^n$ 和 $a_{n+1}X2^{n+1}$ 结合起来的结果再加到最终输出乘积上。这是一个“手工计算”思想的直接阵列形式，所以必然能够生成一个正确的结果。

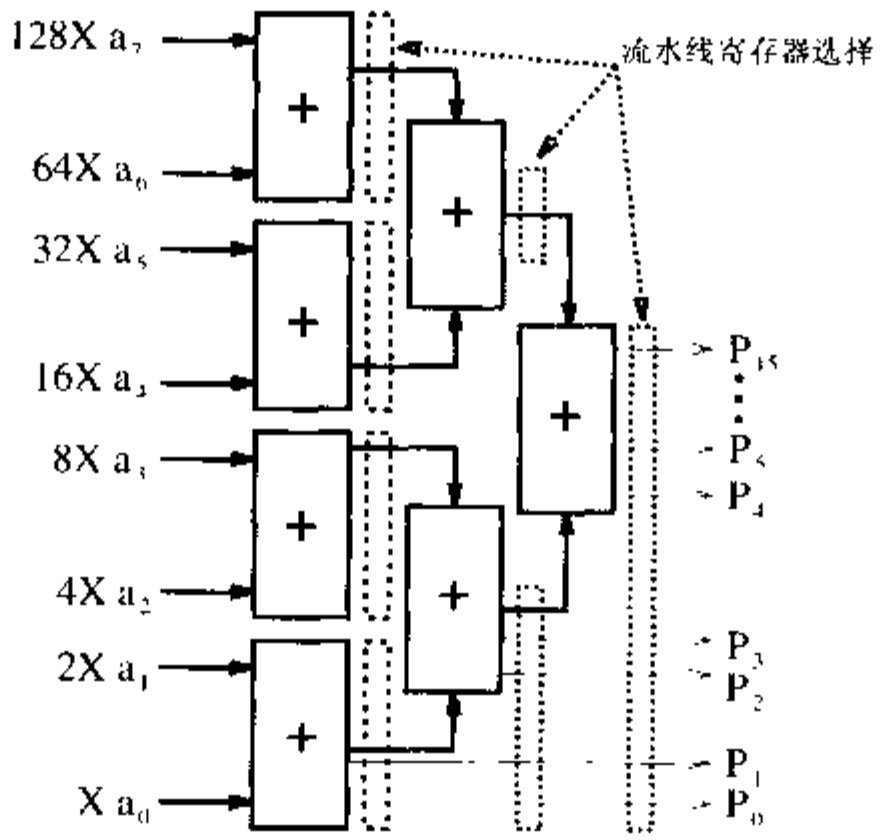


图 2-14　FPGA 的快速阵列乘法器

从图 2-14 中，可以看出，这种阵列乘法器为实现(并行)二叉树乘法器提供了机会。而：

$$\text{二叉树乘法器的步骤总数}=\log_2(N) \tag{2.28}$$

这种可选结构还可以使得在树的层中引入流水线级更为容易。根据(2.28)，达到最大吞吐量所需要的流水线级的数量是：

位宽	2	3～4	5～8	9～16	17～32
最佳流水线级数量	1	2	3	4	5

图 2-15 给出了 3 种流水线式 $N\times N$ 位乘法器的 Registered Performance，其中使用了 MaxPlusII 的 lpm_mult 函数，范围是从 4×4 到 16×16 位，没有(虚线)和有(实线)I/O 单元寄存器的情形。图 2-16 给出了乘法器采用(实线)和没有采用(虚线)I/O 单元寄存器的效果。将输入寄存器置于乘法器附近(关闭选项：Assign |Global Project Logic Synthesis|Automatic Fast I/O)，在性能上稍有提高。在 FLEX10K20 中，乘法器的最大规模是一个 20×20 位的单元，使用了 872 个 LC，具有 4 个流水线级，其 Registered Performance 是以 37.03MHz 的速度运行的。

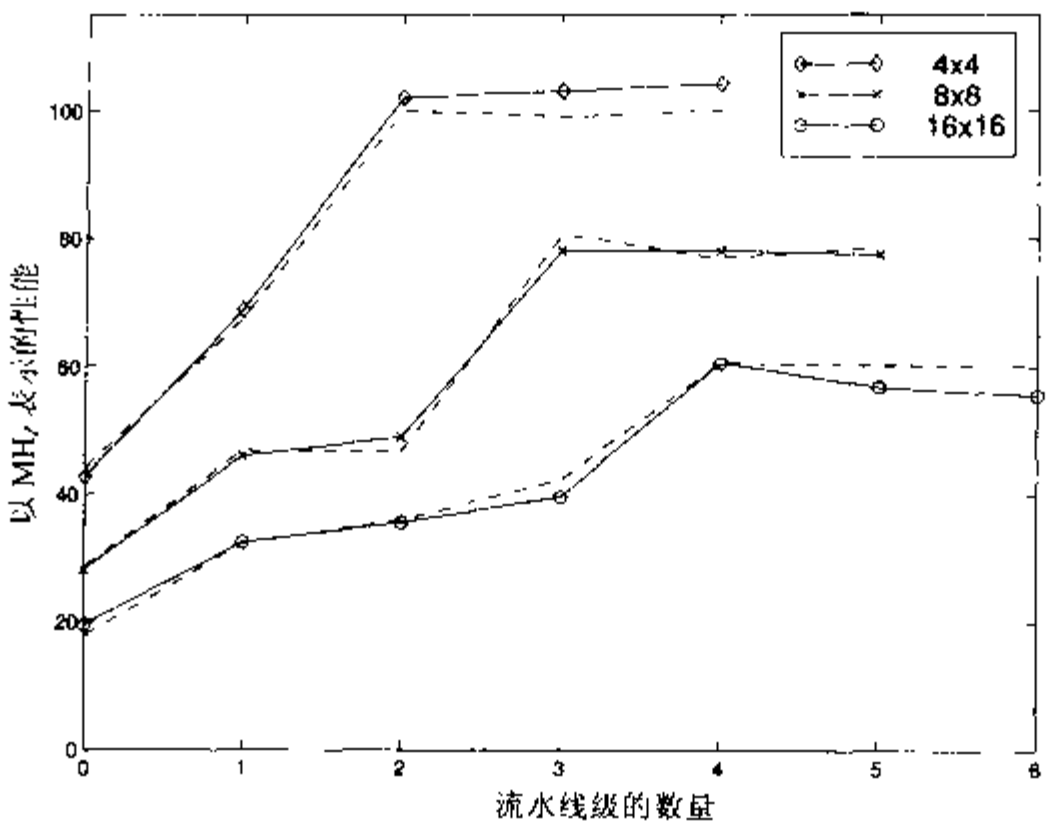


图 2-15　FPGA 阵列乘法器的性能

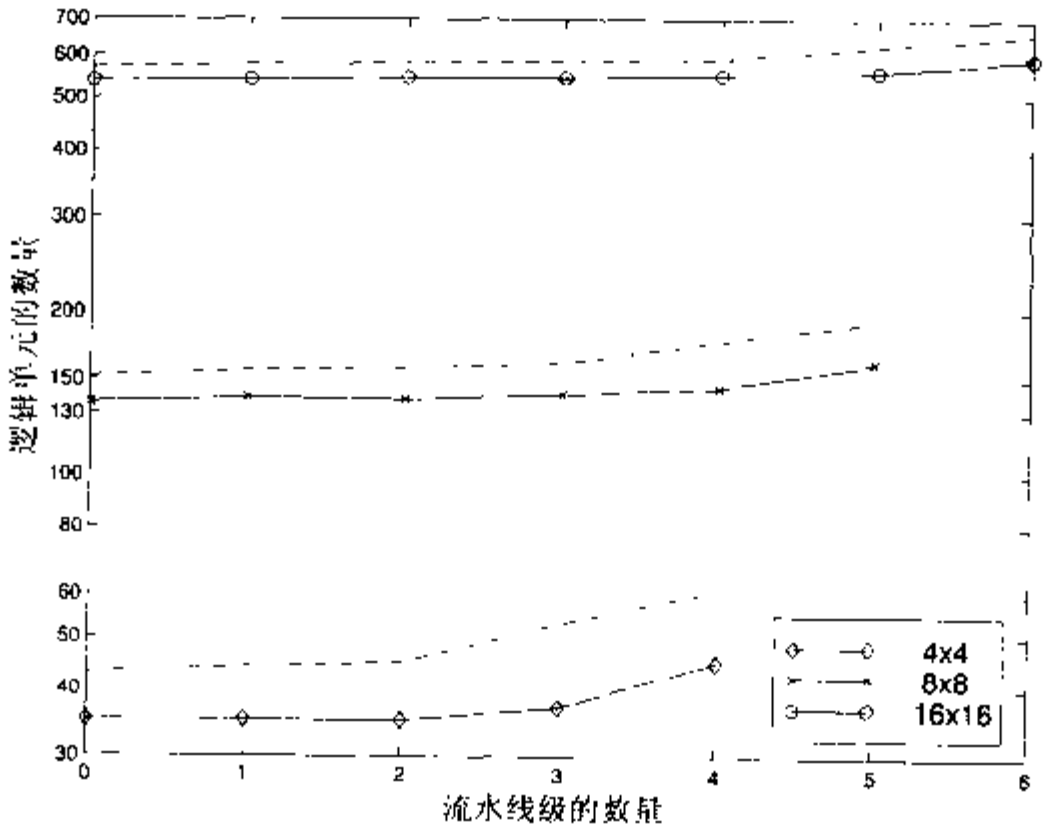


图 2-16　阵列乘法器在 LC 方面的工作量

通常应用在 ASIC 领域中的另一种乘法器结构包括布思乘法器和 Wallace 树乘法器。我们将在练习 2.1 和 2.2 中加以讨论，但是它们现在已经很少用在与 FPGA 有关的领域中了。

乘法器模块

$2N\times 2N$ 乘法器可以按着 $N\times N$ 乘法器模块的方式来定义。最终的乘法器定义如下：

$$\begin{aligned} P &= Y\cdot X=(Y_2 2^N+Y_1)(X_2 2^N+Y_1) \\ &= Y_2X_2 2^{2N}+(Y_2X_1+Y_1X_2)2^N+Y_1X_1 \end{aligned} \tag{2.29}$$

其中下标 2 和 1 分别代表 N 位的两部分中最有效和最无效的一半。这种划分结构可以用在由于 FPGA 的容量有限而不能够实现一个所需要规模的乘法器时的情形，也可以用在 2KB EAB 模块实现乘法器的情形。FLEX10K20 中的 EAB 模块数量只有 6 个，所以最大的对称乘法器就是 8×8。表 2-8 给出了基于 EAB 的乘法器的数据。将表 2-8 中的数据与图 2-15 和 2-16 中的数据进行对比，就可以看出基于 EAB 的乘法器降低了 LC 的数量，但是没有提高 Registered Performance。

表 2-8　基于 EAB 的乘法器的数据

规　模	是否使用 I/O 寄存器	LC	EAB	流 水 线 级	Registered Performance
4×4	√	0	1	0	35.58MHz
4×4	√	0	1	1	51.54MHz
4×4	√	0	1	2	76.33MHz
4×4	—	16	1	0	37.87MHz
4×4	—	16	1	1	51.81MHz
4×4	—	16	1	2	76.33MHz
8×8	√	29	4	0	23.80MHz
8×8	√	29	4	1	40.81MHz
8×8	√	29	4	2	40.81MHz
8×8	—	49	4	0	24.15MHz
8×8	—	49	4	1	40.81MHz
8×8	—	49	4	2	40.81MHz

2.5　乘-累加器(Multiply-Accumulator, MAC)与乘积之和(Sum of Product, SOP)

众所周知，DSP 算法是非常强调乘-累加(multiply-accumulate, MAC)的。为了加以说明，我们来看一下：

$$y[n] = f[n] * x[n] = \sum_{k=0}^{L-1} f[k]x[n-k] \tag{2.30}$$

给出的线性卷积和。对于每次采样 $y[n]$都需要进行 L 次连续乘法和 $L-1$ 次加法操作来计算乘积之和。这就说明 $N\times N$ 位乘法器需要与累加器焊接在一起。全精度 $N\times N$ 位乘积位宽是 $2N$。如果两个操作数都是(对称的)有符号数，则乘积只有 $2N-1$ 个有效位，也就是说有两个符号位。为了保持累加器有效的动态范围，通常将其在位宽上多设计出额外的 K 位，就像下面的例子。

例 2.16： 模拟器件 PDSP 系列 ADSP21xx 含有一个 16×16 的阵列乘法器和一个具有额外 8 位位宽的累加器(累加器位宽总和是 32+8＝40 位)。有了这 8 个额外的位，不用牺牲输出就可以实现至少 2^8 次累加。如果两个操作数都是有符号数，就能够执行 2^9 次累加。为了产生需要的输出格式，像现代的 PDSP 还包括了一个筒状移位器，可以在一个时钟周期内完成需要的调整。

对于主流数字信号处理来讲，考虑在定点 PDSP 中的溢出是非常重要的，需要对 DSP 对象进行实时计算而不希望有中断。可以想到，检查和维护累加器溢出就会中断数据流，并带来一个明显的暂时负担。通过正确地选择数据的保护位就可以消除这一负担。

对于传统 PDSP 的 MAC，计算乘积和还有另一种可选方法，我们将在下面的部分加以讨论。

2.5.1 分布式算法基础

分布式算法(distributed arithmetic, DA)是一项重要的 FPGA 技术，广泛地应用在计算乘积和

$$y = < c, x > = \sum_{n=0}^{N-1} c[n] \cdot x[n] \tag{2.31}$$

之中。除了卷积之外，相关、DFT 计算和前面讨论过的 RNS 反演映射也可以阐述成“乘积和(sum of products, SOP)”。当使用传统的算法单元完成一个滤波周期时，大约需要用 N 个 MAC 循环。使用流水线可以缩短这一数量，但是也非常有限，仍旧非常长。当使用通用乘法器时，这就成了一个简单问题了。

在许多 DSP 应用领域中，在技术上是不需要通用的乘法算法的。如果滤波系数 $c[n]$可以通过演绎得到，那么在技术上部分乘积项 $c[n]x[n]$就变成了一个常数乘法(也就是缩放)。这是一个重要的差别，也是 DA 设计的一个先决条件。

有关 DA 的讨论最初可以追溯到 1973 年 Croisier[43] 发表的论文，而 DA 的推广工作则是由 Peled 和 Liu[47] 进行的。Yiu[48] 将 DA 扩展到有符号数，Kammeyer[49] 和 Taylor[50] 研究了 DA 系统中的量化效应。目前 DA 已经引入到教科书中[53,54]。为了理解 DA 设计范例，考虑“乘积和”内积如下：

$$\begin{aligned} y = < c, x > &= \sum_{n=0}^{N-1} c[n] \cdot x[n] \\ &= c[0]x[0]+c[1]x[1]+\ldots+c[N\text{-}1]x[N\text{-}1] \end{aligned} \tag{2.32}$$

进一步假设系数 $c[n]$是已知常数，$x[n]$是变量。无符号 DA 系统假设变量 $x[n]$的表达式如下：

$$x[n] = \sum_{b=0}^{B-1} x_b[n] \times 2^b, \quad x_b[n] \in [0, 1] \tag{2.33}$$

其中 $x_b[n]$表示 $x[n]$的第 b 位，而 $x[n]$也就是 x 的第 n 次采样，而内积 y 可以表示为：

$$y=\sum_{n=0}^{N-1}c[n]\cdot\sum_{b=0}^{B-1}x_b[k]\cdot 2^b \tag{2.34}$$

重新分别求和(也就是“分布式算法名称的由来”)，其结果如下：

$$\begin{aligned}y=&c[0](x_{B-1}[0]2^{B-1}+x_{B-2}[0]2^{B-2}+\ldots+x_0[0]2^0)\\&+c[1](x_{B-1}[1]2^{B-1}+x_{B-2}[1]2^{B-2}+\ldots+x_0[1]2^0)\\&\vdots\\&+c[N-1](x_{B-1}[N-1]2^{B-1}+\ldots+x_0[N-1]2^0)\\=&(c[0]x_{B-1}[0]+c[1]x_{B-1}[1]+\ldots+c[N-1]x_{B-1}[N-1])2^{B-1}\\&+(c[0]x_{B-2}[0]+c[1]x_{B-2}[1]+\ldots+c[N-1]x_{B-2}[N-1])2^{B-2}\\&\vdots\\&+(c[0]x_0[0]+c[1]x_0[1]+\ldots+c[N-1]x_0[N-1])2^0\end{aligned}$$

或者可以写成更为简洁的如下形式：

$$y=\sum_{b=0}^{B-1}2^b\cdot\sum_{n=0}^{N-1}\underbrace{c[n]\cdot x_b[n]}_{f(c[n],x_b[n])}=\sum_{b=0}^{B-1}2^b\cdot\sum_{n=0}^{N-1}f(c[n],x_b[n]) \tag{2.35}$$

函数 $f(c[n], x_b[n])$的实现需要特别的注意。所指的实现方法就是利用一个 LUT 实现映射 $f(c[n], x_b[n])$。也就是说 2^N-字宽、预先设定程序的 LUT 接收一个 N 位输入向量 $x_b=[x_b[0], x_b[1], \ldots, x_b[N-1]]$，输出为 $f(c[n], x_b[n])$。各个映射 $f(c[n], x_b[n])$都由相应的二次幂加权并累加。利用如图 2-17(b)所示的移位加法器就能够有效地实现累加。在 N 次查询循环后就完成了对内积 y 的计算。

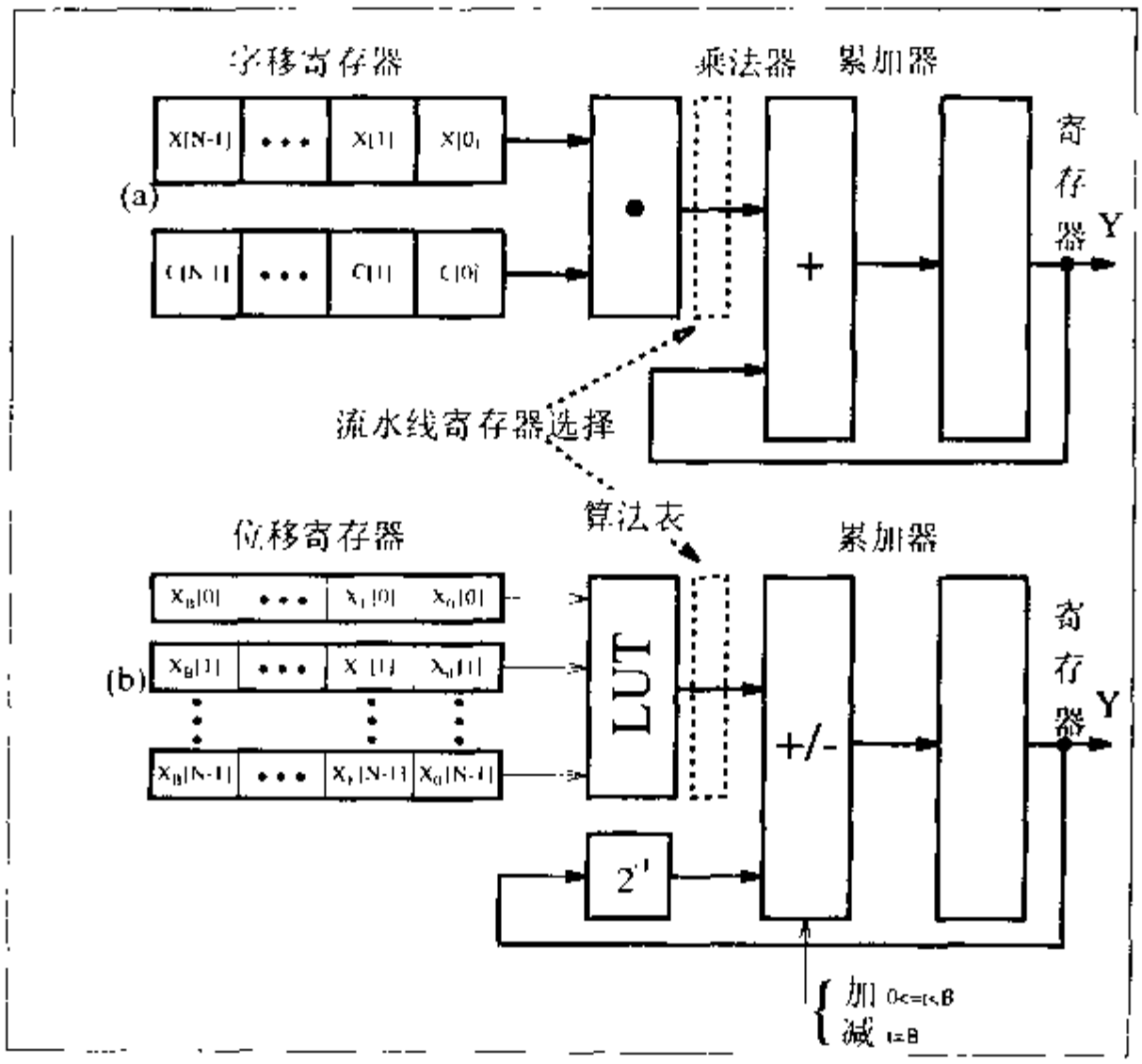


图 2-17　传统的 PDSP 和移位加法器 DA 结构

例 2.17：无符号 DA 卷积

某一根据内积方程 $y=<c, x>=\sum_{n=0}^{2} c[n]x[n]$ 来定义的三阶内积，假设 3 位系数其值分别是 $c[0]=2$，$c[1]=3$ 和 $c[2]=1$，最后得到执行 $f(c[n], x_b[n])$ 的 LUT 如下：

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[n], x[n]$
0	0	0	$1\times0+3\times0+2\times0=0_{10}=000_2$
0	0	1	$1\times0+3\times0+2\times1=2_{10}=010_2$
0	1	0	$1\times0+3\times1+2\times0=3_{10}=011_2$
0	1	1	$1\times0+3\times1+2\times1=5_{10}=101_2$
1	0	0	$1\times1+3\times0+2\times0=1_{10}=001_2$
1	0	1	$1\times1+3\times0+2\times1=3_{10}=011_2$
1	1	0	$1\times1+3\times1+2\times0=4_{10}=100_2$
1	1	1	$1\times1+3\times1+2\times1=6_{10}=110_2$

所得代表 $x[n]=\{x[0]=1_{10}=001_2, x[1]=3_{10}=011_2, x[2]=7_{10}=111_2\}$ 的内积如下：

步 骤 t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t]+ACC[t-1]=ACC[t]$
0	1	1	1	$6\times2^0+\quad 0\quad =6$
1	1	1	0	$4\times2^1+\quad 6\quad =14$
2	1	0	0	$1\times2^2+\quad 14\quad =18$

进行数值校验可以看到：

$$y=<c, x>=c[0]x[0]+c[1]x[1]+c[2]x[2]$$
$$=2\times1+3\times3+1\times7=18 \quad \surd$$

对于其硬件实现，与用 b 转移每个中间值(这需要一个昂贵的筒状移位器)相比，更为适合的是将寄存器内容本身在每一次迭代时逐位右移。很容易就可以证明这样做也会得到同样的结果。

可以比较一下分别采用通用 MAC 和 DA 硬件的 N 阶 B 位线性卷积的带宽。图 2-17 中给出了传统的 PDSP 结构和采用分布式算法的相同实现。

假设 LUT 和通用乘法器的延迟时间相同，$\tau=\tau(\text{LUT})=\tau(\text{MUL})$。这样，计算的等待时间就是 DA 的 $B\tau(\text{LUT})$和 PDSP 的 $N\tau(\text{MUL})$。就小位宽 B 来讲，DA 设计的速度可以显著地超过基于 MAC 的设计。在第 3 章中将就具体的滤波器设计示例来进行比较。

2.5.2　有符号的 DA 数制

接下来我们将讨论如何修改(2.32)，使之能够处理有符号补码数。在补码中，最高有效位是用来区别正数和负数的。例如，从表 2-1 中就可以看到十进制的 -3 编码后就是 $101_2=-4+0+1=-3_{10}$。所以我们将采用下面的$(B+1)$位表达式：

$$x[n]=-2^{b}\cdot x_{B}[n]+\sum_{b=0}^{B-1}x_{b}[n]\cdot 2^{b} \tag{2.36}$$

与(2.34)联立得到输出 y 如下：

$$y=-2^{b}\cdot f(c[n],\ x_{B}\ [n])+\ \sum_{b=0}^{B-1}2^{n}\cdot\sum_{n=0}^{N-1}f(c[n],x_{b}[n]) \tag{2.37}$$

要实现有符号 DA 系统，我们可以有两种选择来修改无符号 DA 系统。这就是：

- 带有加/减控制的累加器
- 采用具有一个额外输入的 ROM

在此推荐使用最常见的可转换式累加器，因为表中额外的输入位还需要一个两倍字长的表。下面的示例证明了加/减转换设计的处理步骤。

例 2.18：　有符号 DA 的内积

再来研究一下由卷积和 $y=<c,\ x>=\sum_{n=0}^{2}c[n]x[n]$ 定义的三阶内积。假设给定的数据是 N=4 位二进制补码编码形式的，系数分别是 $c[0]=-2$，$c[1]=3$ 和 $c[2]=1$，相应的 LUT 表如下：

$x_b[2]$	$x_b[1]$	$x_b[0]$	$f(c[k],\ x[n])$
0	0	0	$1\times0+3\times0-2\times0=0_{10}$
0	0	1	$1\times0+3\times0-2\times1=-2_{10}$
0	1	0	$1\times0+3\times1-2\times0=3_{10}$
0	1	1	$1\times0+3\times1-2\times1=1_{10}$
1	0	0	$1\times1+3\times0-2\times0=1_{10}$
1	0	1	$1\times1+3\times0-2\times1=-1_{10}$
1	1	0	$1\times1+3\times1-2\times0=4_{10}$
1	1	1	$1\times1+3\times1-2\times1=2_{10}$

$x[k]$的值是 $x[0]=1_{10}=0001_{2C}$，$x[1]=-3_{10}=1101_{2C}$，$x[2]=7_{10}=0111_{2C}$。采样下标 k 下的输出，也就是 y，其定义如下：

步 骤 t	$x_t[2]$	$x_t[1]$	$x_t[0]$	$f[t]\cdot2^{t}+Y[t-1]=Y[t]$
0	1	1	1	$2\times2^{0}\ +\ 0\ =2$
1	1	0	0	$1\times2^{1}\ +\ 2\ =4$
2	1	1	0	$4\times2^{2}\ +\ 4\ =20$
	$x_t[2]$	$x_t[1]$	$x_t[0]$	$-f[t]\cdot2^{t}+Y[t-1]=Y[t]$
3	0	1	0	$-3\times2^{3}\ +\ 20\ =-4$

数值校验结果是：$c[0]x[0]+c[1]x[1]+c[2]x[2]=-2\times1+3\times(-3)+1\times7=-4$　　√

2.5.3　改进的DA解决方案

接下来我们要讨论对基本DA概念进行的两项有趣的改进，其中第一个改进是缩小规模，而第二个改进则是提高速度。

如果系数N过多，用单个LUT不能够执行全字(复习：输入LUT位宽=系数的数量)，就可以利用部分表并将结果相加。如果我们再加上流水线寄存器，这一改进并没有降低速度，但是却可以极大地减小设计规模，因为LUT的规模随着地址空间，也就是输入系数N的增加而呈指数增加。假定长度为LN的内积：

$$y=<c, x>=\sum_{n=0}^{LN-1} c[n]x[n] \tag{2.38}$$

可以用一个DA结构实现。将和分配到L个独立的N阶并行DA的LUT之中，结果如下：

$$y=<c, x>=\sum_{l=0}^{L-1}\sum_{n=0}^{N-1} c[Ll+n]x[Ll+n] \tag{2.39}$$

如图2-18所示，实现一个$4N$的DA设计需要3个次辅助加法器。表格的规模从一个$4N\times 2^B$的LUT降低到4个$N\times 2^B$表。

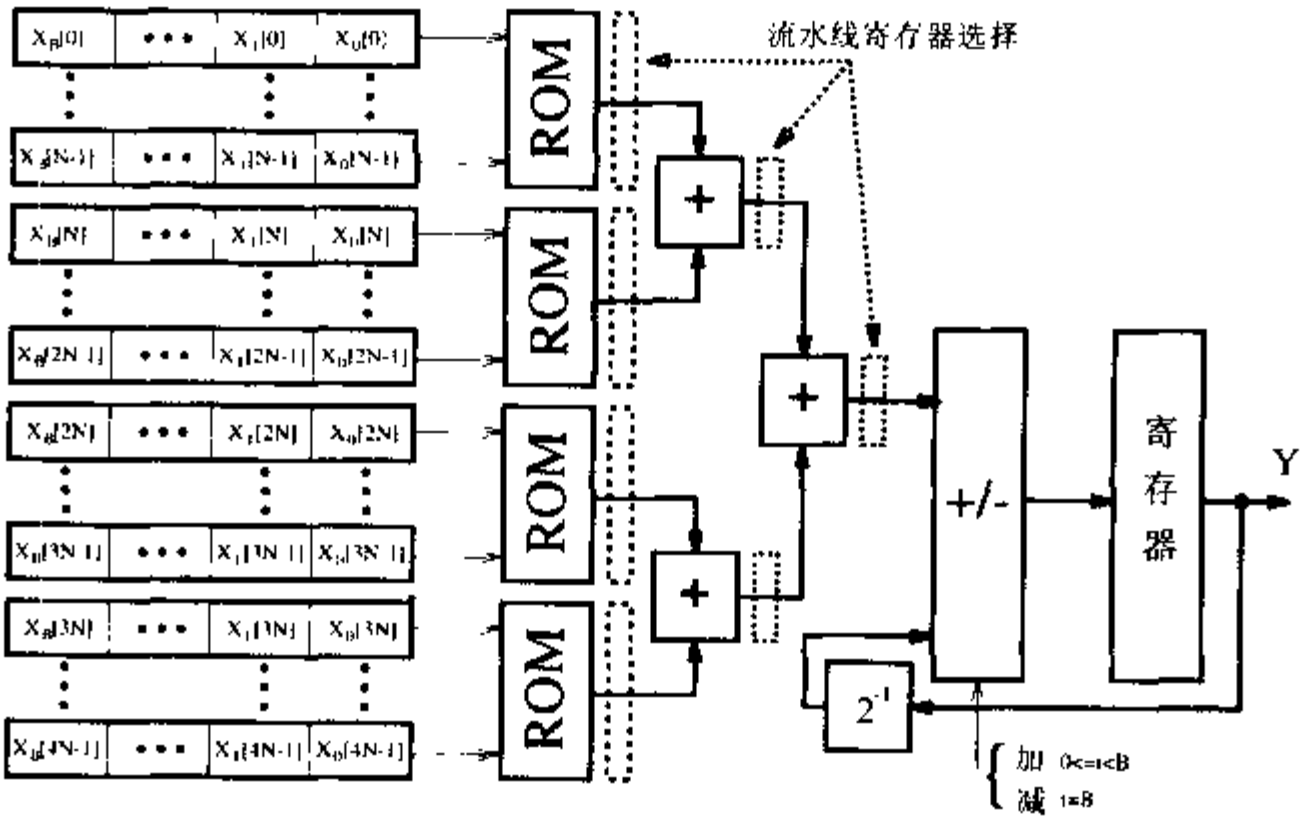


图2-18　将表分割以产生简化规模的分布式算法

另一个DA结构的改进以增加额外的LUT、寄存器和加法器为代价提高了速度。一个N阶乘积和计算的基本DA结构接收N个字中每个字内的一位。如果每个字中有两个位得以接收，则计算速度就可以从根本上翻倍。能够达到的最大速度就是使用如图2-19所示的完全流水线式字并行结构。在此，为4位有符号系数在每个LUT循环内计算出了长度为4的乘积和的新结果。对于最大速度，我们还必须为向量$x_b[n]$的每一位提供一个单独的ROM(具有相同的内容)。但是这样最大速度的代价就变得非常昂贵了：如果我们将输入位宽加倍，就需要两倍的LUT、寄存器和加法器。如果系数N的数量限制在4个或者8个，这一改进就有了吸引人的性能，特别是优于所有商业上可行的PDSP，就像我们将在第3章看到的一样。

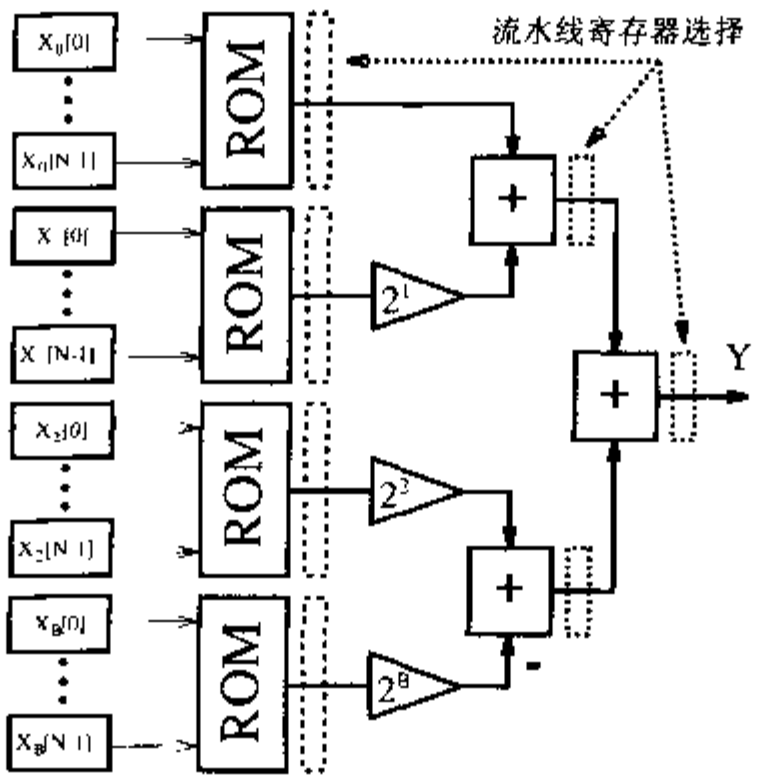


图 2-19　速度最优的高阶分布式算法

2.6　利用 CORDIC 计算特殊函数

如果利用 FPGA 实现某种数字信号处理算法，并且算法使用了一个非普通的(超越)代数函数比如 $x^{1/2}$ 或 arctan y/x，我们可以利用泰勒级数来近似这个函数，也就是：

$$f(x_0)=\sum_{k=0}^{K}\frac{df^{(k)}(x-x_0)}{dx}x^k\Big|_{x=x_0} \tag{2.40}$$

这样问题就化简成一系列乘法和加法运算了。一种可供考虑的更为有效的方法就是基于坐标旋转数字式计算机(Coordinate Rotation Digital Computer, CORDIC)的算法。CORDIC 算法是建立在众多应用基础之上的，例如：便携式计算器[55]和主流 DSP 对象，如适应性滤波器、FFT、DCT[56]、解调器[57]和神经网络[36]。基础 CORDIC 算法可以在 Volder[58]和 Walther[59]发表的两篇经典论文中找到。目前已经做了理论上的拓展，例如在双曲线模式中的范围或是 Hu 等人[60]和 Meyer-baese 等人[57]的量化误差分析等方面的扩展。VLSI 的实现已经在博士论文中讨论过了，例如 Timmermann[61]或 Hahn[62]所著的论文。第一次利用 FPGA 实现是由 Meyer-baese 等人[4, 57]研究的。CORDIC 算法在分布式算法中的实现是由 Ma[63]首创的。Y. Hu 在 1992 年 IEEE《信号处理》期刊评论文章中给出了相关的详细回顾，还包括一些应用细节。

Volder[58]提出最初的 CORDIC 算法是计算在平面直角坐标系(x,y)和极坐标系(R,θ)之间进行自由坐标变换的乘法。Walther[59]推广了 CORDIC 算法，将圆周(m=1)、线性(m=0)和双曲线(m=－1)变换都包括进来。对于每种模式两个旋转方向都是确定的。对于向量化而言，具有原点(X_0,Y_0)的向量按着如下方式旋转：通过将 Y_K 迭代收敛到 0，使得向量最后落在横坐标(也就是 x 轴)上。所谓**旋转**，就是具有原点(X_0,Y_0)的向量旋转一个角度 θ_0，被称为 Z 的角度寄存器的最终值收敛到 0。角 θ_k 闭合，这样每次迭代就只需要一次加法和一次二进制转换。表 2-9 中第二列给出了 3 种模式 m=1,0 和－1 旋转角度的选择。

表 2-9　CORDIC 算法模式

Modus	Angle θ_k	Shift-Sequence	Radius-Factor
circular m=1	$\tan^{-1}(2^{-k})$	0,1,2,…	K_1=1.65
linear m=0	2^{-k}	1,2,…	K_0=1.0
hyperbolic m - - 1	$\tanh^{-1}(2^{-k})$	1,2,3,4,4,…	K_{-1}=0.80

现在我们正式定义 CORDIC 算法，如下所示：

算法 2.19：CORDIC 算法

在每次迭代中，CORDIC 算法执行映射：

$$\begin{bmatrix} X_{k+1} \\ Y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & m\delta_k 2^{-k} \\ m\delta_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} X_k \\ Y_k \end{bmatrix} \tag{2.41}$$

$$Z_{k+1} = Z_k + \delta_k \theta_k$$

其中两个旋转方向是 $Z_K \to 0$ 和 $Y_K \to 0$。

这就意味着存在 6 种操作模式，在表 2-10 中给出了相应的总结。结果是差不多所有的超越函数都可以利用 CORDIC 算法进行计算。正确地选择初始值，就可以直接计算函数 $X \cdot Y$，Y / X，$\sin(Z)$，$\cos(Z)$，$\tan^{-1}(Z)$，$\sinh(Z)$，$\coh(Z)$和 $\tanh(Z)$。其他的函数可以通过选择适当的初始化来推广而得，经常是将多种操作模式组合起来，就如下面的列表所示：

$\tan(Z)=\sin(Z) / \cos(Z)$　　模式：m＝1,0

$\tanh(Z)=\sinh(Z) / \cosh(Z)$　　模式：m＝－1,0

$\exp(Z)=\sinh(Z)+\cosh(Z)$　　模式：m＝－1；$x=y=1$

$\log_e(W)=2\tanh^{-1}(Y/X)$　　模式：m＝－1

$X=W+1, Y=W-1$

$\sqrt{W}=\sqrt{X^2-Y^2}$　　模式：m＝1

$X=W+\frac{1}{4}, Y=W-\frac{1}{4}$

表 2-10　CORDIC 算法的操作模式 m

m	$Z_K \to 0$	$Y_K \to 0$
1	$X_K=K_1(X_0\cos(Z_0)-Y_0\sin(Z_0))$ $Y_K=K_1(X_0\cos(Z_0)+Y_0\sin(Z_0))$	$X_K=K_1\sqrt{X_0^2+Y_0^2}$ $Z_K=Z_0+\arctan(Y_0/X_0)$
0	$X_K=X_0$ $Y_K=Y_0+X_0 \cdot Z_0$	$X_K=X_0$ $Z_K=Z_0+Y_0/X_0$
－1	$X_K=K_{-1}(X_0\cosh(Z_0)-Y_0\sinh(Z_0))$ $Y_K=K_{-1}(X_0\cosh(Z_0)+Y_0\sinh(Z_0))$	$X_K=K_{-1}\sqrt{X_0^2+Y_0^2}$ $Z_K=Z_0+\tanh^{-1}(Y_0/X_0)$

对(2.41)的详细分析显示迭代向量只适用于如图 2-20(a)所示的曲线。向量的长度随着每次迭代发生变化，如图 2-20(b)所示。这种在长度上的变化并不依赖于起始角度，并且在 K 次迭代之后总会出现相同的变化。在表 2-9 的最后一列中给出了半径因子。为了确保 CORDIC 算法收敛，其余所有旋转角之和必须要大于实际旋转的角。下面是线性和圆周变换的情况。对于双曲线模式来说，形如 $n_{k+1}=3n_k+1$ 的迭代必须重复。这就是迭代 4、13、40、121…。

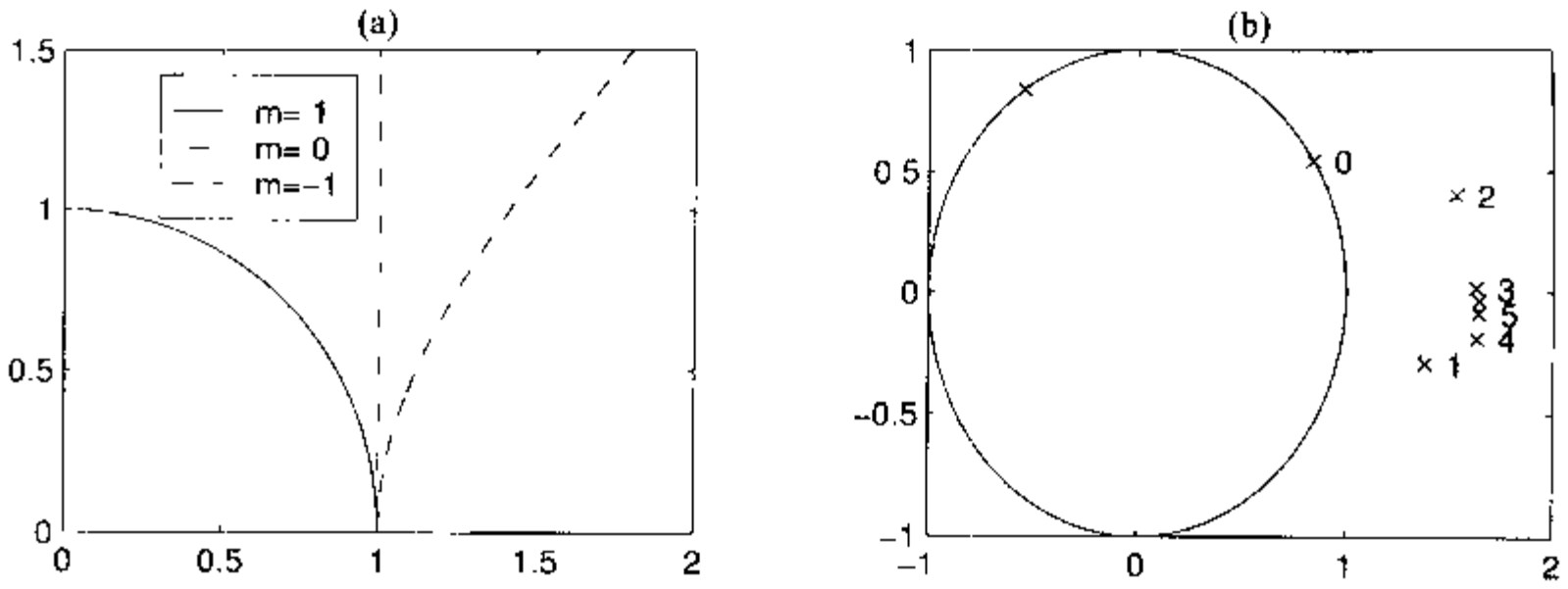


图 2-20　CORDIC (a) 模式 (b) 圆周向量化的示例

输出精度可以利用 Hu 开发的程序进行估算，如图 2-21 所示。曲线图显示圆周模式的有效位精度依赖于 X 和 Y 路径宽度和迭代的次数。如果需要 b 位的输出精度，“经验法则”建议 X 和 Y 路径需要 $\log_2(b)$个额外的保护位。从图 2-22 中也可以看出，Z 路径的位宽具有与 X 和 Y 一样的精度。与圆周 CORDIC 算法相对照，双曲线 CORDIC 算法的有效解决方案不是根据分析就可以计算的，因为精度依赖于第 K 次迭代时 $z(k)$的角度值。此外双曲精度还可以利用仿真进行评估。图 2-23 给出了对每一位宽/数的组合的可能测定值进行 1000 次迭代所得的最小精确度。3D 表示法给出了迭代的数量、X/Y 路径的位宽和有效位形式的最终最低精确度。等高线给出了迭代次数和位宽之间的一种交换。例如要获得 10 位的精确度，可以采用 21 位 X/Y 路径和 18 次迭代，也可以是 24 位 X/Y 路径和 14 次迭代。

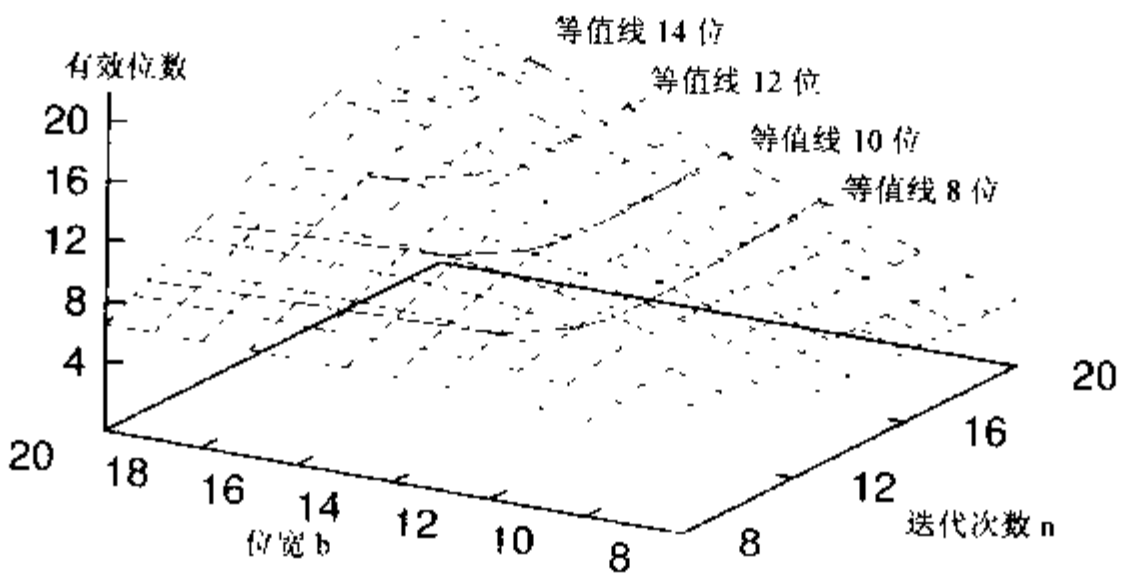


图 2-21　圆周模式中的有效位

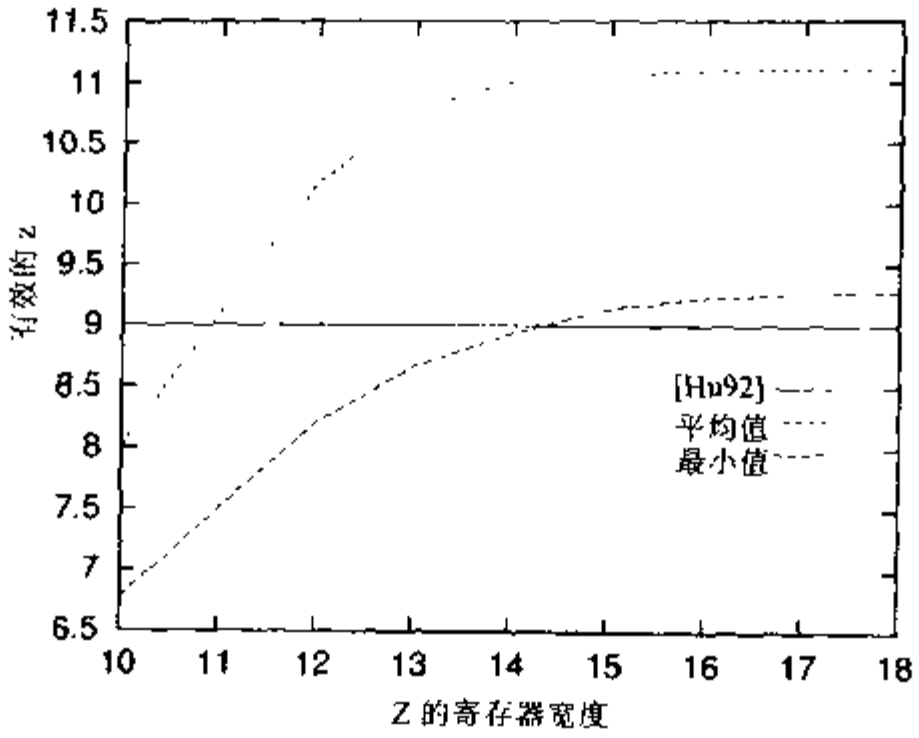


图 2-22　圆周模式的相位分解

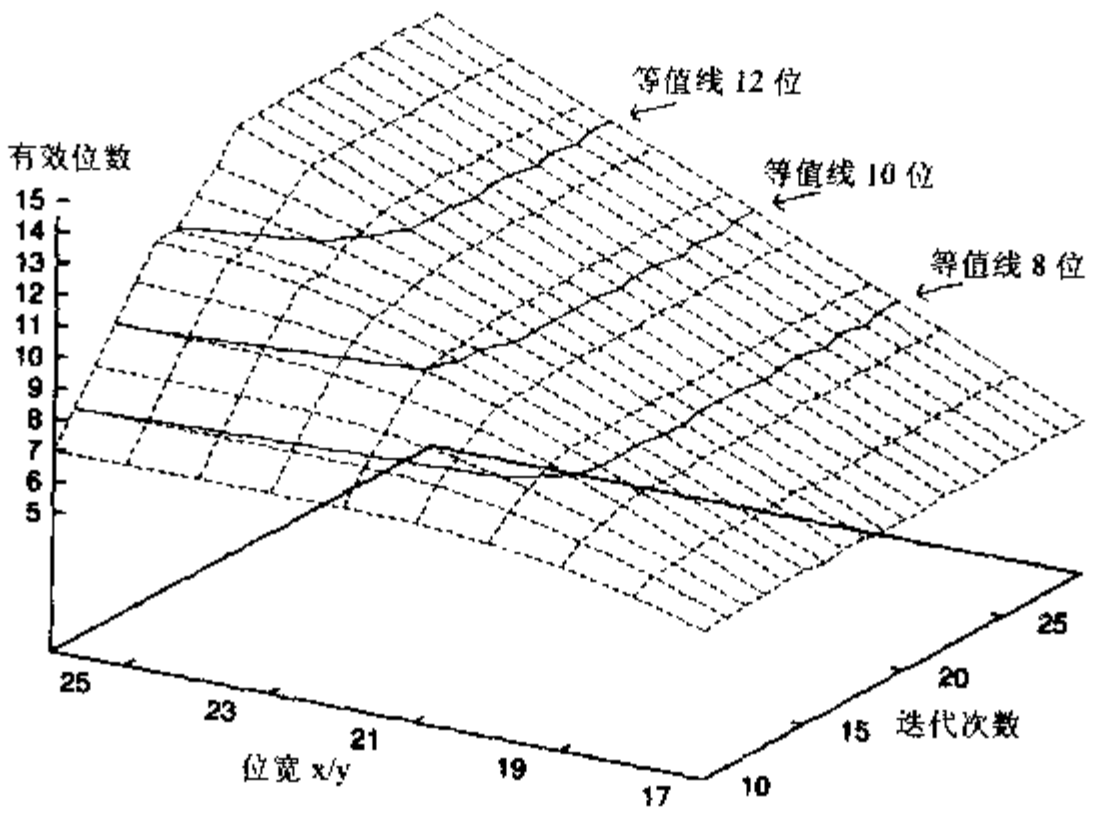


图 2-23　双曲线模式的有效位

CORDIC 结构

实现 CORDIC 结构可以采用两种基本结构：较为简洁的状态机和高速全流水线处理器。

如果计算时间不严格的话，就可以采用图 2-24 所示的状态机。在每个周期内都将精确地计算一次(2.41)的迭代。这一设计中最复杂的就是两个筒状移位器。这两个筒状移位器可以由一个单一筒状移位器代替，采用一个图 2-25 所示的多路转换器或者一个串行(右移或者是左/右)移位器。表 2-11 给出了采用 Xilinx XC3K FPGA 的 13 位实现的不同设计选择之间的比较。

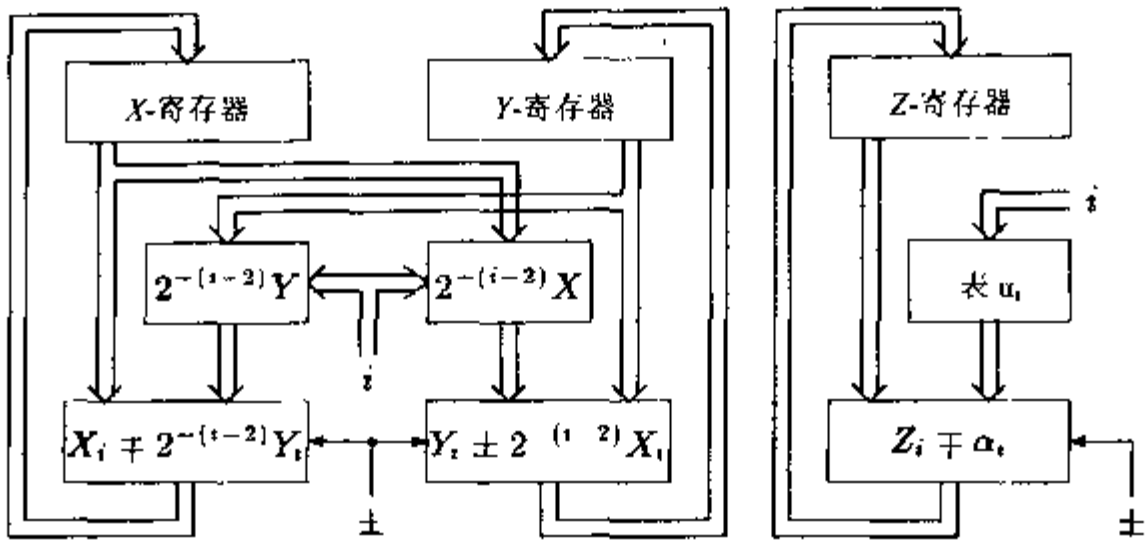


图 2-24　CORDIC 状态机

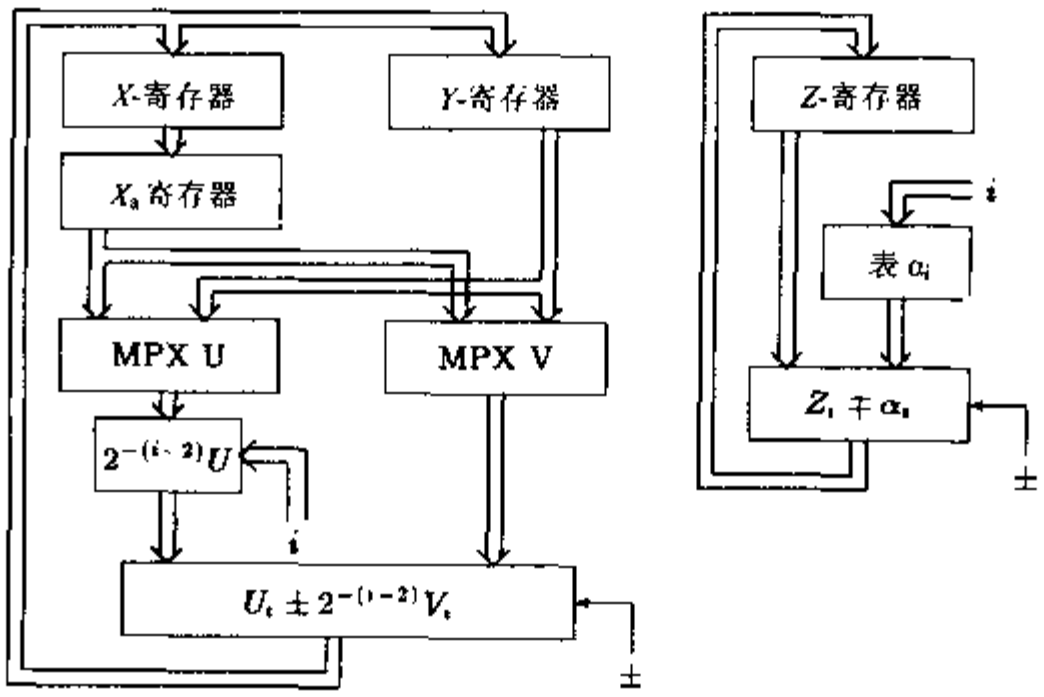


图 2-25　降低复杂性的 CORDIC 状态机

表 2-11　13 位外加 X/Y 路径符号位的 CORDIC 状态机的效率评估(Xilinx XC3K)

(缩写：Ac=accumulator(累加器)；BS=barrelshifter(筒状移位器)；RS=serial right shifter(串行右移移位器)；LRS=serial left right shifter(串行左/右移位器))

结　构	寄　存　器	多路复用器	加　法　器	移　位　器	∑LE	循　环
2BS+2Ac	2×7	0	2×14	1×19.5	81	12
2RS+2Ac	2×7	0	2×14	1×6.5	55	46
2LRS+2Ac	2×7	0	2×14	2×8	58	39
1BS+2Ac	7	3×7	2×14	19.5	75.5	20
1RS+2Ac	7	3×7	2×14	6.5	62.5	56
1LRS+2Ac	7	3×7	2×14	8	64	74
1BS+1Ac	3×7	2×7	14	19.5	68.5	20
1RS+1Ac	3×7	2×7	14	6.5	55.5	92
1LRS+1Ac	3×7	2×7	14	8	57	74

如果需要高速的话，可以采用如图 2-26 所示的全流水线处理器设计版本。图 2-26 给出了

个圆周 CORDIC 的 8 次迭代。在 K 循环起始延迟之后，在每次新的循环之后就会生成一个新的输出值。正如阵列乘法器一样，CORDIC 的实现在 LE 复杂性方面随着位宽的提高而呈平方的增加(见表 2-9)。

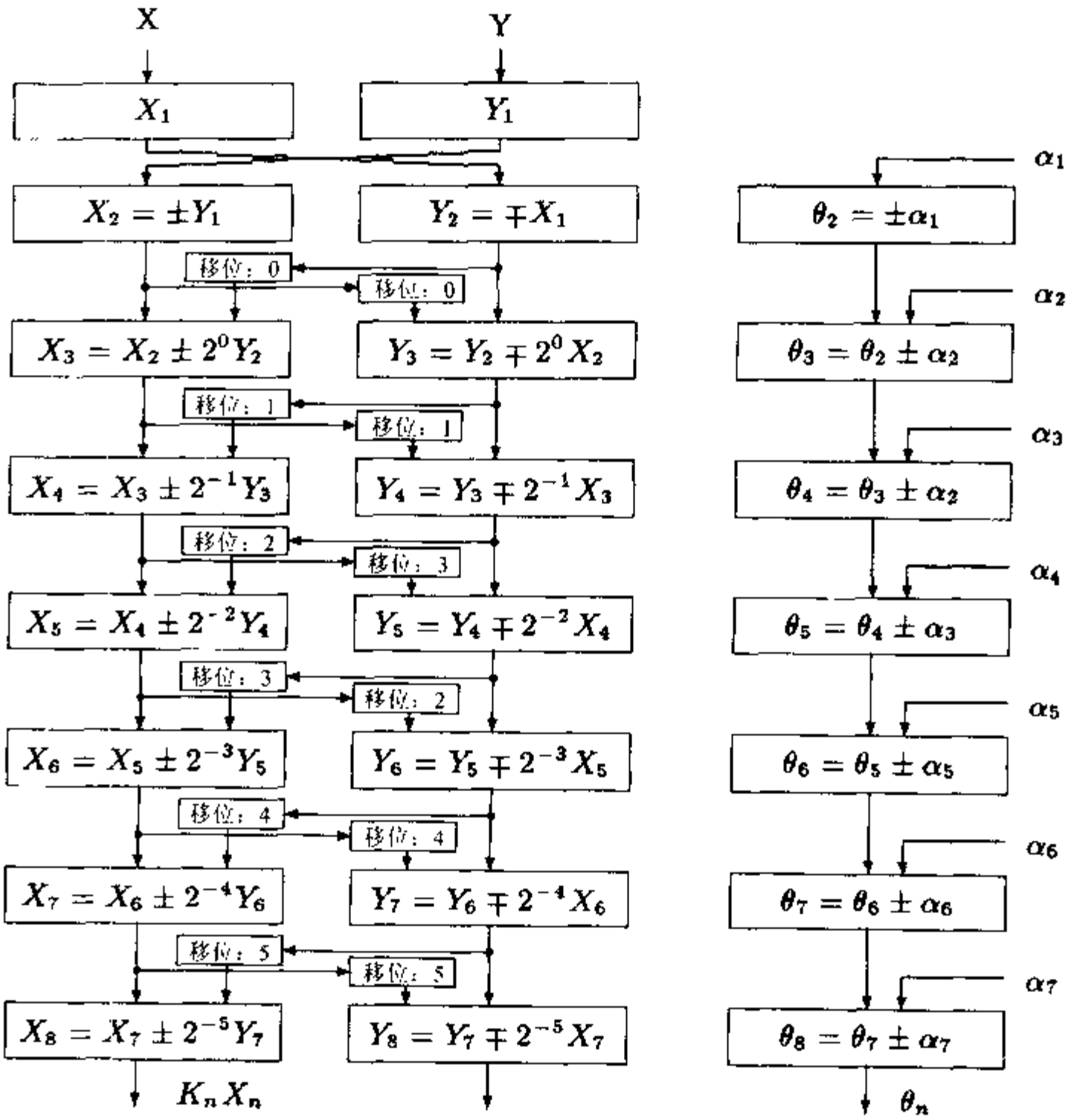


图 2-26　快速 CORDIC 流水线结构

下面的例题给出了一个圆周-向量化全流水线设计的前 4 个步骤。

例 2.20：　向量化模式中的圆周 CORDIC

第一次迭代旋转，向量分别从第二或者第三象限旋转到第一或者第四象限。移位序列是 0,0,1 和 2。前 4 个步骤的旋转角度是：$\arctan(\infty)=90°$，$\arctan(2^0)=45°$，$\arctan(2^{-1})=26.5°$和 $\arctan(2^{-2})=14°$。可以执行 8 位数据的 VHDL 代码[4]如下：

```
PACKAGE eight_bit_int IS      -- User defined types
  SUBTYPE BYTE IS INTEGER RANGE  - 128 TO 127;
  TYPE ARRAY_BYTE IS ARRAY (0 TO 3) OF BYTE;
END eight_bit_int;
```

注 4：这一例子相应的 Verilog 代码文件 cordic.v 可以在附录 A 中找到。

```
LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY cordic IS                                    ------> Interface
        PORT (clk          : IN   STD_LOGIC;
              x_in , y_in : IN   BYTE;
              r, phi, eps  : OUT BYTE);
END cordic;

ARCHITECTURE flex OF cordic IS
  SIGNAL   x, y, z   :    ARRAY_BYTE; -- Array of Bytes
BEGIN

  PROCESS                          ------> Behavioral Style
  BEGIN
    WAIT UNTIL clk = '1'; -- Compute last value first in
    r <= x(3);                    -- sequential VHDL statements !!
    phi <= z(3);
    eps <= y(3);

    IF y(2) > 0 THEN                    -- Rotate 14 degrees
      x(3) <= x(2) + y(2) /4;
      y(3) <= y(2) - x(2) /4;
      z(3) <= z(2) + 14;
    ELSE
      x(3) <= x(2) - y(2) /4;
      y(3) <= y(2) + x(2) /4;
      z(3) <= z(2) - 14;
    END IF;

    IF y(1) > 0 THEN                    -- Rotate 26 degrees
      x(2) <= x(1) + y(1) /2;
      y(2) <= y(1) - x(1) /2;
      z(2) <= z(1) + 26;
    ELSE
      x(2) <= x(1) - y(1) /2;
      y(2) <= y(1) + x(1) /2;
      z(2) <= z(1) - 26;
    END IF;
```

```
    IF y(0) > 0 THEN                    -- Rotate   45 degrees
      x(1) <= x(0) + y(0);
      y(1) <= y(0) - x(0);
      z(1) <= z(0) + 45;
    ELSE
      x(1) <= x(0) - y(0);
      y(1) <= y(0) + x(0);
      z(1) <= z(0) - 45;
    END IF;

-- Test for x_in < 0 rotate 0,+90, or - 90 degrees
    IF x_in > 0 THEN
      x(0) <= x_in;             -- Input in register 0
      y(0) <= y_in;
      z(0) <= 0;
    ELSIF y_in > 0 THEN
      x(0) <= y_in;
      y(0) <= - x_in;
      z(0) <= 90;
    ELSE
      x(0) <= - y_in;
      y(0) <= x_in;
      z(0) <= - 90;
    END IF;
  END PROCESS;

END flex;
```

图 2-27 给出了 X_0=215= - 41mod256 和 Y_0=55 的转换的仿真结果。注意：半径扩大到 $R=X_K=111=1.618\sqrt{X_0^2+Y_0^2}$，并且积累的角以角度形式表示是 $\arctan(Y_0/X_0)=123°$。设计需要 256 个 LC，并且在 fast 合成选项中选择采用 I/ O 单元寄存器时，该设计运行的速度为 43.47MHz，如果不采用 I/O 寄存器，设计需要 280 个寄存器，运行的速度为 48.54MHz。

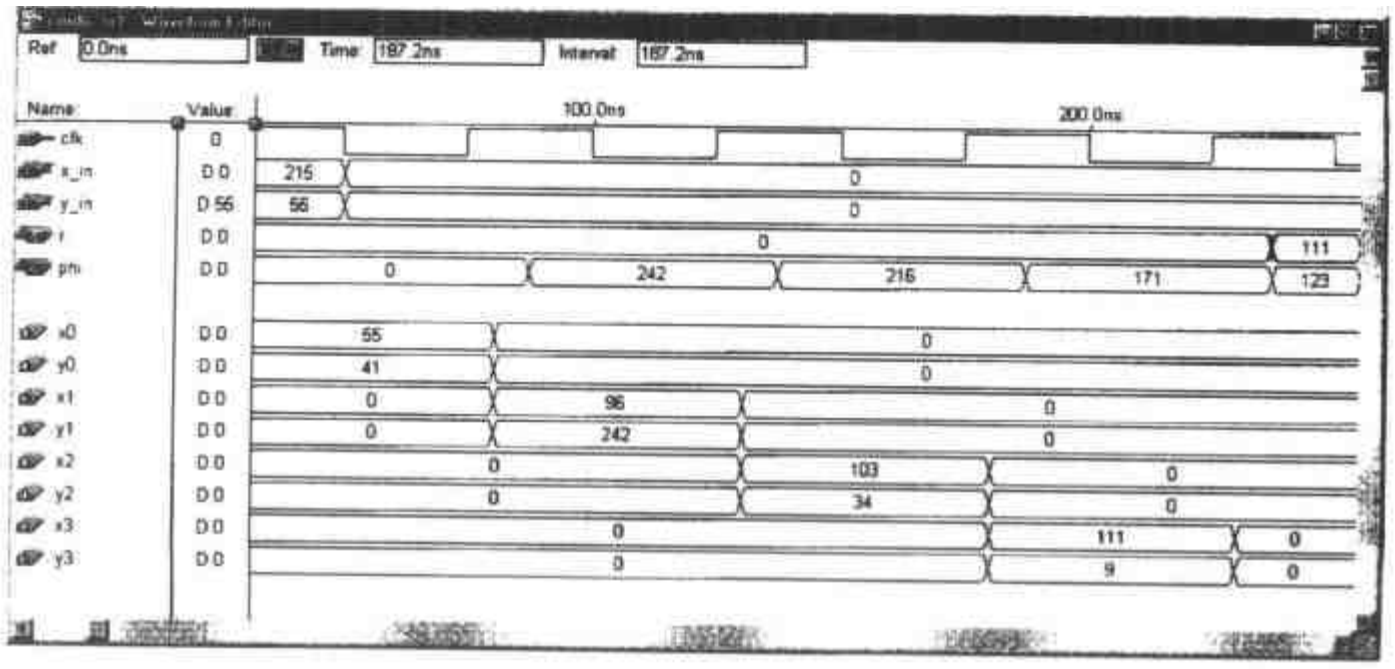


图 2-27　CORDIC 仿真结果

前面的例题中实际的 LC 数量要比期望的 4 阶段 8 位流水线设计需要的 5×8×3=120 个 LC 要多。增加的两个因素之一来源于 FLEX 元器件采用了需要 2*N* 个 LC 的 *N* 位可转换 LPM_ADD_SUB 兆函数。之所以需要 2*N* 个 LC，是因为 LC 在快速算法模式中仅有 3 个输入，而开关模式需要 4 个输入 LUT。要降低两个因素之一造成的 LC 数量的增加，需要 Xilinx XC4K 系列的每个 LC 具有 4 个输入的元器件。

2.7　练习

2.1：Wallace 已经为快速乘法器引入了一种替代结构。这种乘法器类型的基本构造模块是进位储存加法器(carry-save adder, CSA)。CSA 需要 3 个 *n* 位操作数并生成 2 个 *n* 位输出。由于没有进位的传送，这类加法器通常称为 3∶2 压缩或计数器。对于 *n*×*n* 位乘法器，我们总计需要 *n*－2 个 CSA 将输出简化到 2 个操作数。然后将这些操作数用一个(快速)2*n* 位并行进位加法器相加，计算乘法器的最终结果。

(a) CSA 计算可以以并行方式进行。确定一个 *n*×*n* 位乘法器的最小层数，*n*∈[0,16]。

(b) 试解释一下为什么用 FPGA 实现的带有快速二进制补码加法器的乘法器没有普通的阵列乘法器具有吸引力。

(c) 试解释一下最后加法器级中的流水线加法器如何实现快速乘法器？用表 2-7 中的数据估算一下。

(c1) 8×8 位乘法器

(c2) 16×16 位乘法器

需要的 LC 和可能的运行速度。

2.2：布思乘法器采用经典 CSD 码降低了必需的加法/减法操作数量。由 LSB 开始，通常是一步处理 2 或 3 位(称作 radix-4 和 radix-8 算法)。下面的表给出了可能的 radix-4 结构和操作。

x_{k+1}	x_k	x_{k-1}	累加器操作	注　释
0	0	0	ACC→ACC+R*(0)	在一串“0”内
0	0	1	ACC→ACC+R*(X)	以一串“1”结束
0	1	0	ACC→ACC+R*(X)	无
0	1	1	ACC→ACC+R*(2X)	以一串“1”结束
1	0	0	ACC→ACC+R*(－2X)	由一串“1”开始
1	0	1	ACC→ACC+R*(－X)	无
1	1	0	ACC→ACC+R*(－X)	由一串“1”开始
1	1	1	ACC→ACC+R*(0)	在一串“1”内

状态机的硬件要求是一个累加器和一个二进制补码移位器。

(a) 令 *X* 是一个有符号 6 位二进制补码表达式 －10＝110110_{2C}。完成下面的布思乘积 *P*＝*XY*＝－10*Y* 表，并且指出每个步骤中累加器的操作。

步骤	x_5	x_4	x_3	x_2	x_1	x_0	x_{-1}	ACC	ACC+布思法则
开始	1	1	0	1	1	0	0		
0									(2.42)
1									
2									

(b) 比较 radix-4 和 radix-8 算法的布思乘法器与例 2.15 中串行/并行乘法器的执行时间。

练习使用 MaxPlusII

2.3：(a) 用 MaxPlusII 的编译器编译文件 add_2p.vhd，选择速度和规模最优，需要多少 LC？并解释结果。

(b) 用 15+102 进行仿真。

2.4：解释对于减法应该如何修改设计 add_2p.vhd。修改设计并且对 3－2 进行仿真。

2.5：(a) 用 MaxPlusII 的编译器编译文件 mul_ser.vhd。

(b) 确定 8 位设计的 Registered Performance 和规模。总乘法执行时间是多少？

2.6：将设计文件 mul_ser.vhd 修改为 12×12 位数的乘法。

(a) 用 1000×2000 仿真新的设计。

(b) 评估新设计的 Registered Performance 和规模。

(c) 12×12 位乘法器的总执行时间是多少？

2.7：(a) 在 MaxPlusII 中设计一个级状态机，实现 6×6 位有符号输入的布思乘法器(请参阅练习 2.2)。

(b) 仿真 4 个数据±5×(±9)。

(c) 确定其 Registered Performance。

(d) 确定在最大速度时的 LC 的利用。

2.8：(a) 设计一个 generic CSA 组件，用来给 8×8 位乘法器构造一个 Wallace 树乘法器。

(b) 用 MaxPlusII 实现 8×8 Wallace 树。

(c) 用最后的加法器计算乘积，并且用 100×63 检验您的乘法器。

(d) 流水线设计 Wallace 树。这种流水线设计的最大吞吐量是多少？

(e) 用流水线加法器代替 16 位输出加法器。这一设计的 Registered Performance 如何？

2.9：(a) 练习使用组件实例原则，用预定义宏 LPM_ADD_SUB 和 LPM_MULT 为 8 位复数乘法器编写 VHDL 代码(也就是$(a+jb)(c+jd)=ac-bd+j(ad+bd)$)，所有的操作数 a，b，c 和 d 都是 8 位。

(b) 确定其 Registered Performance。

(c) 确定最大速度合成的 LC 的利用。

(d) 最优单 LPM_MULT 乘法器有多少流水线级？

(e) 最优复数乘法器总计有多少流水线级？

2.10：复数乘法器的一个可行算法如下：

$$\begin{aligned} s[1]&=a-b & s[2]&c-d & s[3]&=c+d \\ m[1]&=s[1]d & m[2]&=s[2]a & m[3]&=s[3]b \\ s[4]&=m[1]+m[2] & s[5]&=m[1]+m[3] \\ (a+jb)(c+jd)&=s[4]+js[5] \end{aligned} \tag{2.43}$$

这种算法一般需要 5 个加法器和 3 个乘法器。确证：如果一个系数，比方说 $c+jd$ 已知，那么 $s[2]$、$s[3]$和 d 就可以预先储存，运算就简化成 3 次加法和 3 次乘法。还可以：

(a) 利用上面的算法以及预定义宏 LPM_ADD_SUB 和 LPM_MULT 为 8 位有符号输入设计一个流水线 5/3 复数乘法器。

(b) 评估最大速度合成的 Registered Performance 和规模。

(c) 最优的单 LPM_MULT 乘法器有多少流水线级？

(d) 最优复数乘法器总计有多少流水线级？

2.11：用 MaxPlusII 的编译器编译文件 cordic.vhd，并且：

(a) 用 x_in=±50 和 y_in=±70 进行仿真(采用波形文件 cordic.scf)。为 4 个仿真确定半径因子。

(b) 确定半径和相位的最大误差，并与未量化的计算进行比较。

2.12：修改设计文件 cordic.vhd，实现 4 级和 5 级 CORDIC 流水线设计。

(a) 计算旋转角度，并编译 HVDL 代码。

(b) 用 x_in=±50 和 y_in=±70 进行仿真。

(c) 与未量化的计算进行比较，半径和相位的最大误差是多少？

第3章　有限脉冲响应(FIR)数字滤波器

3.1　数字滤波器

数字滤波器通常都是应用于修正或改变时域或频域中信号的属性。最为普通的数字滤波器就是线性时间不变量(linear time-invariant, LTI)滤波器。LTI 与其输入信号之间相互作用，经过个称为线性卷积的过程。表示为 $y=f * x$，其中 f 是滤波器的脉冲响应，x 是输入信号，而 y 是卷积输出。线性卷积过程的正式定义如下：

$$y[n]=x[n]*f[n]=\sum_{k}x[k]f[n-k]=\sum_{k}f[k]x[n-k] \tag{3.1}$$

LTI 数字滤波器通常分成**有限脉冲响应**(finite impulse response, 也就是 FIR)和**无限脉冲响应**(infinite impulse response, 也就是 IIR)两大类。顾名思义，FIR 滤波器由有限个采样值组成，将上述卷积的数量降低到在每个采样时刻为有限个。而 IIR 滤波器需要执行无限数量次卷积。在这一章将要讨论 FIR 滤波器的设计和实现方法，而有关 IIR 滤波器的问题将要在第 4 章讨论。

研究数字滤波器的动机就在于它们正日益成为一种主要的 DSP 操作。数字滤波器正在迅速地代替传统的模拟滤波器，后者是利用 RLC 元器件和运算放大器实现的。模拟滤波器采用拉普拉斯变换的普通微分方程进行数学模拟，是在时域或 s(也叫作拉普拉斯)域内进行分析的。模拟原型设计只能够应用在 IIR 设计之中，而 FIR 通常采用直接的计算机规范和算法进行设计。

在本章假定一个数字滤波器，该数字滤波器是一个已经设计出来并被选来用于实现的 FTP 滤波器。首先简要回顾一下 FIR 的设计过程，接下来讨论利用 FPGA 实现改进。

3.2　FIR 理论

带有常系数的 FIR 滤波器是一种 LTI 数字滤波器。L 阶或者长度为 L 的 FIR 输出对应于输入时间序列 $x[n]$的关系由一种有限卷积数量形式给出，具体形式如下：

$$y[n]=x[n]*f[n]=\sum_{k=0}^{L-1}x[k]f[n-k] \tag{3.2}$$

其中从 $f[0]\neq 0$ 一直到 $f[L-1]\neq 0$ 均是滤波器的 L 阶的系数，同时也对应于 FIR 的脉冲响应。对于 LTI 系统可以更为方便地将(3.2)表达成 z 域内的形式：

$$Y(z)=F(z)X(z) \tag{3.3}$$

其中 $F(z)$是 FIR 的传递函数，其 z 域内的定义形式如下：

$$F(z)=\sum_{k=0}^{L-1} f[k]z^{-k} \tag{3.4}$$

图 3-1 给出了 L 阶 LTI 型 FIR 滤波器的图解。可以看出，FIR 滤波器是由一个“抽头延迟线”加法器和乘法器的集合构成的。传给每个乘法器的操作数就是一个 FIR 系数，显然也可以称作“抽头权重”。过去也有人将 FIR 滤波器称为“横向滤波器”，就是说它的“抽头延迟线”结构。

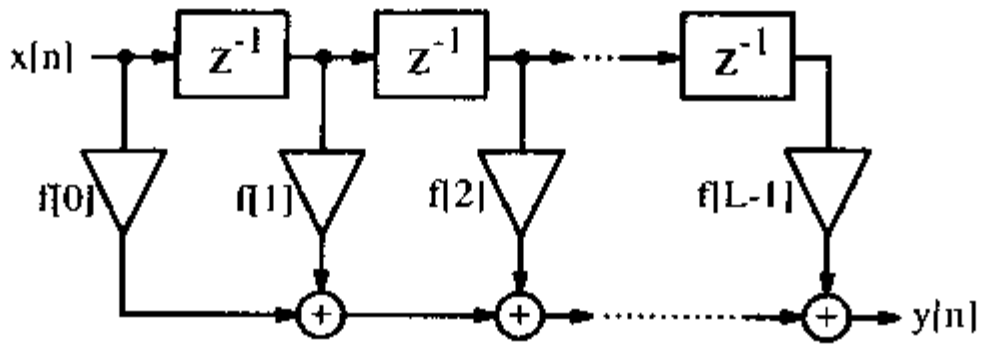


图 3-1　直接形式的 FIR 滤波器

(3.4)中多项式 $F(z)$的根确定了滤波器的零点。仅有零点存在也就是 FIR 经常被称作“全零点滤波器”的原因。在第 5 章我们将讨论 FIR 滤波器中重要的一类(叫作 CIC 滤波器)，它是递归的，但也是 FIR。这是可能的，因为递归部分产生的极点已经被滤波器的非递归部分消除了。有效极点/零点图就变得只有极点了，也就是全零点滤波器或者是 FIR。注意：非递归滤波器均是 FIR，但是递归滤波器却可以是 FIR 或者是 IIR。图 3-2 说明了这一关系。

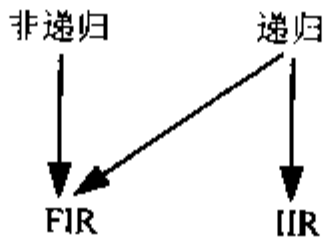


图 3-2　结构和脉冲长度之间的关系

3.2.1　具有转置结构的 FIR 滤波器

直接 FIR 模型的一个变种称为**转置式 FIR 滤波器**，可以根据图 3-1 中的 FIR 滤波器来构造：

- 输入输出互换
- 颠倒信号流的方向
- 用一个差分放大器代替一个加法器，反之亦然

转置式滤波器如图 3-3 所示，通常是指 FIR 滤波器的实现。该滤波器的优点在于我们不再需要给 $x[n]$提供额外的移位寄存器，而且也没有必要为达到高吞吐量给乘积的加法器(树)添加额外的流水线级。

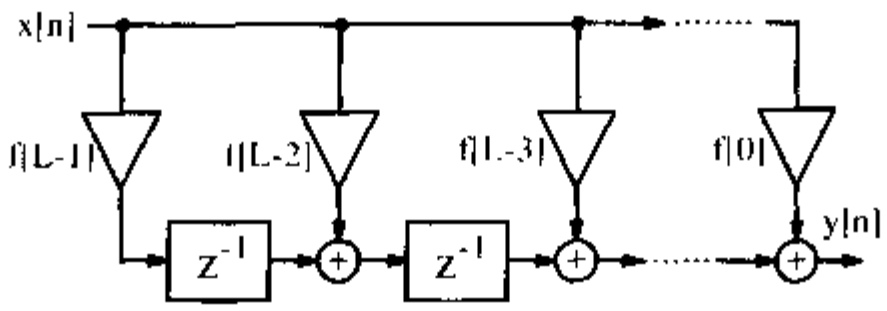


图 3-3　转置结构的 FIR 滤波器

下面的例题给出了转置式滤波器的一个直接实现。

例 3.1：可编程 FIR 滤波器

我们先来回顾一下采用 PDSP 进行乘积和计算的讨论(请参阅 2.5 节)，对于 B_x 数据/系数位宽和滤波长度 L，还必须提供额外的无符号 SOP(乘积和)的 $\log_2(L)$个数位和有符号算法的 $\log_2(L)$ - 1 个保护位。对于 9 位有符号数据/系数和 L=4 而言，加法器带宽必须是 $9+9+\log_2^{(4)}-1=19$。

下面的 VHDL 代码[1]给出了实现长为 4 的滤波器的通用规范。

```
-- This is a generic FIR filter generator
-- It uses W1 bit data/coefficients bits
LIBRARY lpm;                                    -- Using predefined packages
USE lpm.lpm_components.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY fir_gen IS                               ------> Interface
  GENERIC (W1 : integer := 9;  -- Input bit width
           W2 : integer := 18;-- Multiplier bit width 2*W1
           W3 : integer := 19;-- Adder width = W2+log2(L) - 1
           W4 : integer := 11;-- Output bit width
           L  : integer := 4;  -- Filter length
       Mpipe : integer := 3   -- Pipeline steps of multiplier
           );
  PORT ( clk    : IN STD_LOGIC;
         Load_x : IN  STD_LOGIC;
         x_in   : IN  STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
         c_in   : IN  STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
         y_out  : OUT STD_LOGIC_VECTOR(W4 - 1 DOWNTO 0));
END fir_gen;

ARCHITECTURE flex OF fir_gen IS
```

注 1：这一例子相应的 Verilog 代码文件 fir_gen.v 可以在附录 A 中找到。

```
  SUBTYPE N1BIT IS STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
  SUBTYPE N2BIT IS STD_LOGIC_VECTOR(W2 - 1 DOWNTO 0);
  SUBTYPE N3BIT IS STD_LOGIC_VECTOR(W3 - 1 DOWNTO 0);
  TYPE ARRAY_N1BIT IS ARRAY (0 TO L - 1) OF N1BIT;
  TYPE ARRAY_N2BIT IS ARRAY (0 TO L - 1) OF N2BIT;
  TYPE ARRAY_N3BIT IS ARRAY (0 TO L - 1) OF N3BIT;

  SIGNAL  x  :  N1BIT;
  SIGNAL  y  :  N3BIT;
  SIGNAL  c  :  ARRAY_N1BIT; -- Coefficient array
  SIGNAL  p  :  ARRAY_N2BIT; -- Product array
  SIGNAL  a  :  ARRAY_N3BIT; -- Adder array

BEGIN

  Load: PROCESS                  ------> Load data or coefficient
  BEGIN
    WAIT UNTIL clk = '1';
    IF (Load_x = '0') THEN
      c(L - 1) <= c_in;          -- Store coefficient in register
      FOR I IN L - 2 DOWNTO 0 LOOP   -- Coefficients shift one
        c(I) <= c(I+1);
      END LOOP;
    ELSE
      x <= x_in;                 -- Get one data sample at a time
    END IF;
  END PROCESS Load;


  SOP: PROCESS (clk)             ------> Compute sum-of-products
  BEGIN
    IF clk'event and (clk = '1') THEN
    FOR I IN 0 TO L - 2   LOOP         -- Compute the transposed
      a(I) <= (p(I)(W2 - 1) & p(I)) + a(I+1); -- filter adds
    END LOOP;
    a(L - 1) <= p(L - 1)(W2 - 1) & p(L - 1);       -- First TAP has
    END IF;                                        -- only a register
    y <= a(0);
  END PROCESS SOP;

  -- Instantiate L pipelined multiplier
  MulGen: FOR I IN 0 TO L - 1 GENERATE
  Muls: lpm_mult                   -- Multiply p(i) = c(i) * x;
        GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W1,
                      LPM_PIPELINE => Mpipe,
```

```
                    LPM_REPRESENTATION => "SIGNED",
                    LPM_WIDTHP => W2,
                    LPM_WIDTHS => W2)
        PORT MAP ( clock => clk, dataa => x,
                   datab => c(I), result => p(I));
        END GENERATE;

    y_out <= y(W3 - 1 DOWNTO W3 - W4);

END flex;
```

处理过程的第一步是 Load，如果 Load_x=0，就将系数下载到抽头延迟线上。否则就将数据字下载到 x 寄存器中，第二步称为 SOP，执行乘积和的计算，对乘积 p(I)进行一位有符号扩展，并加到前面的部分乘积和上。还要注意所有的乘法器都是由 generate 声明来举例说明的，这一声明允许额外流水线级的分配。最后，输出 y_out 被赋以 SOP 除以 256 的值，因为事先假定的系数都是分数形式的(也就是$|f[k]| \leqslant 1.0$)。设计使用了 890 个 LC，以 46.72MHz 的 Registered Performance 运行。

要仿真这一长度为 4 的滤波器，先来研究一下 Daubechies DB4 滤波器系数：

$$G(z) = \left((1+\sqrt{3}) + (3+\sqrt{3})z^{-1} + (3-\sqrt{3})z^{-2} + (1-\sqrt{3})z^{-3}\right)\frac{1}{4\sqrt{2}}$$

$$G(z) = 0.48301 + 0.8365z^{-1} + 0.2241z^{-2} - 0.1294z^{-3}$$

将系数量化成 8 位(加上符号位) 精度模式，结果如下：

$$G(z) = (124 + 214z^{-1} + 57z^{-2} - 33z^{-3})/256$$

$$= \frac{124}{256} + \frac{214}{256}z^{-1} + \frac{57}{256}z^{-2} + \left(-\frac{33}{256}z^{-3}\right)$$

从图 3-4 可以看出在前面 4 个阶段，我们将系数{124，214，57，－33}下载到抽头延迟线。注意：MaxPlusII 将－33 显示成无符号数，也就是 512－33=479。接下来通过将 100 下载到 x 寄存器中来核对滤波器的脉冲响应。首次有效输出出现在 450ns 之后，就像我们在图 3-4 中所见到的一样。

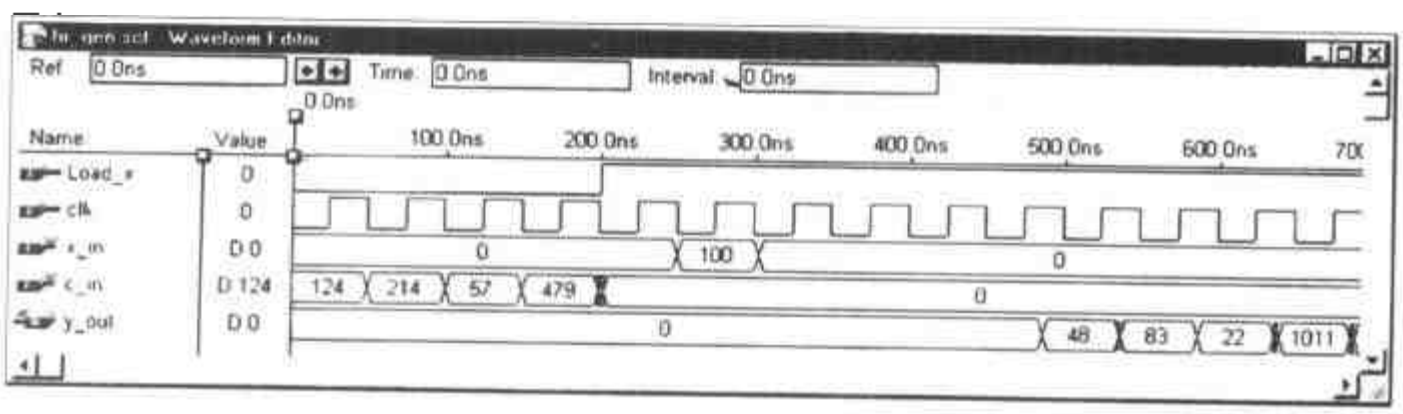


图 3-4 下载了 Daubechies 滤波器系数的 4 抽头可编程 FIR 滤波器的仿真

3.2.2 FIR 滤波器的对称性

FIR 脉冲响应的中心是对称性的一个重要的点。为了方便起见，经常可以将这一点定义成第 0 次采样时刻，这样的滤波器描述就是 a-causal(中心符号)。对于奇数长度的 FIR，a-causal

滤波器模型如下：

$$F(z)=\sum_{k=-(L-1)\ 2}^{(L-1)\ 2} f[k]z^{-k} \tag{3.5}$$

FIR 的周期响应可以根据求滤波器对单一周期边缘的传递函数来计算。令 $z=e^{\omega T}$，得到如下公式：

$$F(\omega)=F(e^{j\omega T})=\sum_{k} f[k]e^{-j\omega kT} \tag{3.6}$$

接下来我们用$|F(\omega)|$表示滤波器的绝对值周期响应，用$|\phi(\omega)|$表示相位响应，且满足：

$$\phi(\omega)=\arctan\left(\frac{\Im(F(\omega))}{\Re(F(\omega))}\right) \tag{3.7}$$

数字滤波器更多是利用相位和绝对值来描述，而较少使用 z 域、传递函数或者是复频率变换。

3.2.3　线性相位 FIR 滤波器

在许多应用领域，例如通信和图像处理中，在一定频率范围内维持相位的完整性是一个期望的系统属性。因此，设计能够建立线性相位-频率性能的滤波器是必须遵循的规范。系统相位线性度的标准尺度就是“**组延迟**”，其定义为：

$$\tau(\omega)=\frac{d\phi(\omega)}{d\omega} \tag{3.8}$$

完全理想的线性相位滤波器对于一定频率范围的组延迟是一个常数。可以看到，如果滤波器是对称或者反对称的，就可以实现线性相位，因此更倾向于采用(3.5)的 a-causal 结构。从(3.7)中可以看出，如果频率响应 $F(\omega)$是一个纯实或者纯虚函数，就可以实现固定的组延迟。这就意味着滤波器的脉冲响应必须保持偶对称或者奇对称，也就是：

$$f[n]=f[-n] \quad 或 \quad f[n]=-f[-n] \tag{3.9}$$

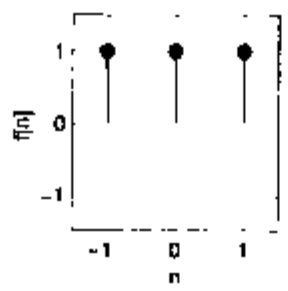
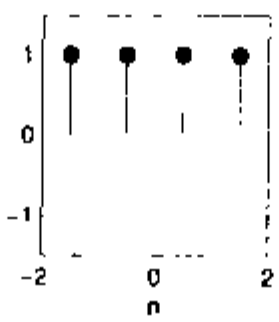
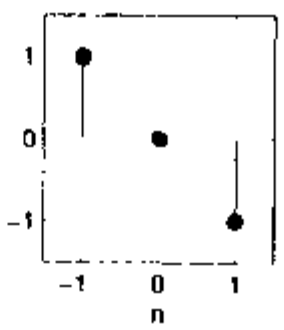
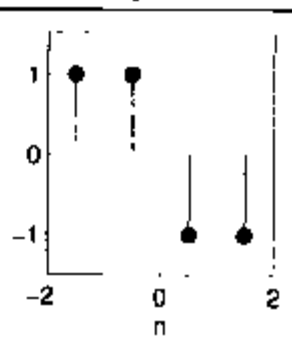
例如：一个奇数阶偶对称 FIR 滤波器的频率响应如下：

$$F(\omega)=f[0]+\sum_{k} f[k]e^{-jk\omega T}+f[-k]\ \ e^{jk\omega T} \tag{3.10}$$

$$=f[0]+2\sum_{k>0} f[k]\cos(k\omega T) \tag{3.11}$$

可以看到频率响应是频率的纯实函数。表 3-1 总结了对称、反对称、偶数阶和奇数阶的 4 种可能选择。此外，表 3-1 还以图形的形式显示了每类线性相位 FIR 的例子。线性相位 FIR 的固有对称属性还可以降低所需要的乘法器 L 的数量，如图 3-1 所示。研究一下图 3-5 中的线性相位 FIR(假定是偶对称的)，这是完全采用系数对称的滤波器。可以看到“对称”结构在每个滤波周期内都提供了一个乘法器预算资源。正好是图 3-1 中给出的直接结构的一半(L 比 $L/2$)，而加法器的数量保持不变，还是 $L-1$ 个。

表 3-1　4 种可能的线性相位 FIR 滤波器 $F(z)=\sum_k f[k]z^{-k}$

对称性	$f[n]=f[-n]$	$f[n]=f[-n]$	$f[n]=-f[-n]$	$f[n]=-f[-n]$
L	奇	偶	奇	偶
零点位置		π	0 和 π	0
示例				

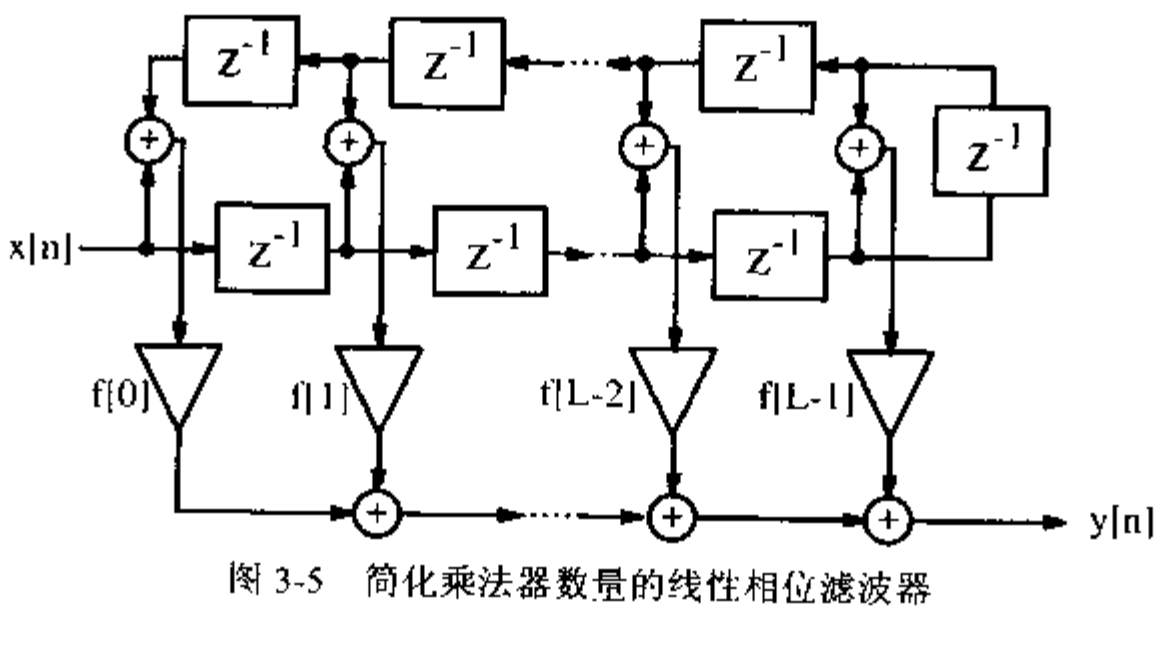


图 3-5　简化乘法器数量的线性相位滤波器

3.3　设计 FIR 滤波器

现代数字 FIR 滤波器都是采用计算机辅助工程(computer-aid engineering, CAE)工具进行设计的。本章所使用的滤波器是利用 MATLAB 软件中的信号处理工具箱设计的。这个工具箱包括一个“交互式低通滤波器设计”演示示例，覆盖了许多典型的数字滤波器的设计，包括：

- 等同纹波(Equirriple，也称作极小化极大)FIR 设计，该设计采用 ParksMcClellan 和 Remez 交换理论作为设计线性相位(对称)的 equirriple FIR 的依据。这一 equirriple 设计还可以用来设计微分器或者 Hilbert 变换器。
- Kaiser 窗函数设计采用了由 Kaiser 窗函数加权的反离散傅立叶变换理论。
- 最小二乘法 FIR 理论。这种滤波器的设计在通频带和抑止频带之中都存在纹波，但是通过最小二乘法的方法就可以将误差减到最小。
- 在第 4 章中将要讨论 4 种 IIR 滤波器设计方法(Butterworth、Chebyshev I 和 II 与 Elliptic 方法)。

在本节我们将专门研究一下 FIR 理论。大多数期望的滤波器传递函数(也就是幅频特性)我们是知道的，例如低通滤波器的规格说明就包括：通频带$[0...\omega_p]$、过渡频带$[\omega_p...\omega_s]$和抑止频带$[\omega_s...\pi]$的规格说明，其中假定采样频率为 2π。要计算滤波器系数，我们可以采用接下来要讨

论的直接频率方法。

3.3.1　直接窗函数设计方法

离散傅立叶变换(Discrete Fourier Transform, DFT)在频域和时域之间建立了一种直接关系。由于频域是滤波器定义的域，所以 DFT 可以用来计算一组可以生成与目标滤波器的频率响应相接近的滤波器的 FIR 滤波器系数。如此设计的滤波器就称为**直接 FIR 滤波器**。直接 FIR 滤波器定义如下：

$$f[n]=\mathrm{IDFT}(F[k])=\sum_{k}F[k]e^{j2\pi kn/L} \tag{3.12}$$

根据基础信号与系统理论可以得出：真实信号频谱都是**厄密共轭**(Hermitian)的。也就是说实频谱具有偶对称特性，而虚频谱则是奇对称的。如果合成的滤波器仅有实系数，则目标 DFT 设计频谱必然是厄密共轭或者是 $F[k]=F^*[-k]$的。其中*表示共轭复数。

利用矩形窗函数来研究长度为 16 的直接 FIR 滤波器，如图 3-6(a)所示，通频带纹波如图 3-6(b)所示。注意：滤波器给出了一个与理想低通滤波器接近的合理近似，最大不符合出现在过渡频带的边缘。观察到的“阻尼振荡”归因于**吉布斯**(Gibbs)现象，后者与有限傅立叶频谱再现陡沿的不稳定性有关。吉布斯阻尼振荡是在直接逆 DFT 方法中隐含的，预期能够达到滤波器阶的宽量程±7%以内。为了阐明这一点，我们来研究一下图 3-6(c)中长度为 128 的滤波器的例子，通频带纹波如图 3-6(d)所示。尽管滤波长度明显提高了(从 16 到 128)，但是边缘还是有差不多相同的阻尼振荡。阻尼振荡效应只能够采用数据“窗口”才可以抑制，在双边光滑地逐渐减小到 0。数据窗口覆盖了 FIR 的脉冲响应，伴随着过渡频带的加宽，生成一个更加“光滑的”幅频特性。例如：如果将 Kaiser 窗函数应用到 FIR 上，就可以减小吉布斯阻尼振荡，如图 3-7(上图)所示。在下面总结并给出了其他经典窗函数，它们之间的差别就在于它们在阻尼振荡和过渡频带宽度扩展之间的折衷的能力上有所不同。已经得到认可并发表的窗函数的数量是非常大的，最为常用的窗函数(用 $w[n]$表示)有：

- 矩形：$w[n]=1$
- Bartlett(三角形)：$w[n]=2n/N$
- 汉宁(Hanning)：$w[n]=0.5(1-\cos(2\pi n/L))$
- 加重平均(Hamming)：$w[n]=0.54-0.46\cos(2\pi n/L)$
- Blackman：$w[n]=0.42-0.5\cos(2\pi n/L)+0.08\cos(4\pi n/L)$
- Kaiser：$w[n]=\mathrm{I}_0\left(\beta\sqrt{1-(n-L/2)^2/(L/2)^2}\right)$

表 3-2 给出了这些窗函数的一些最重要的参数。

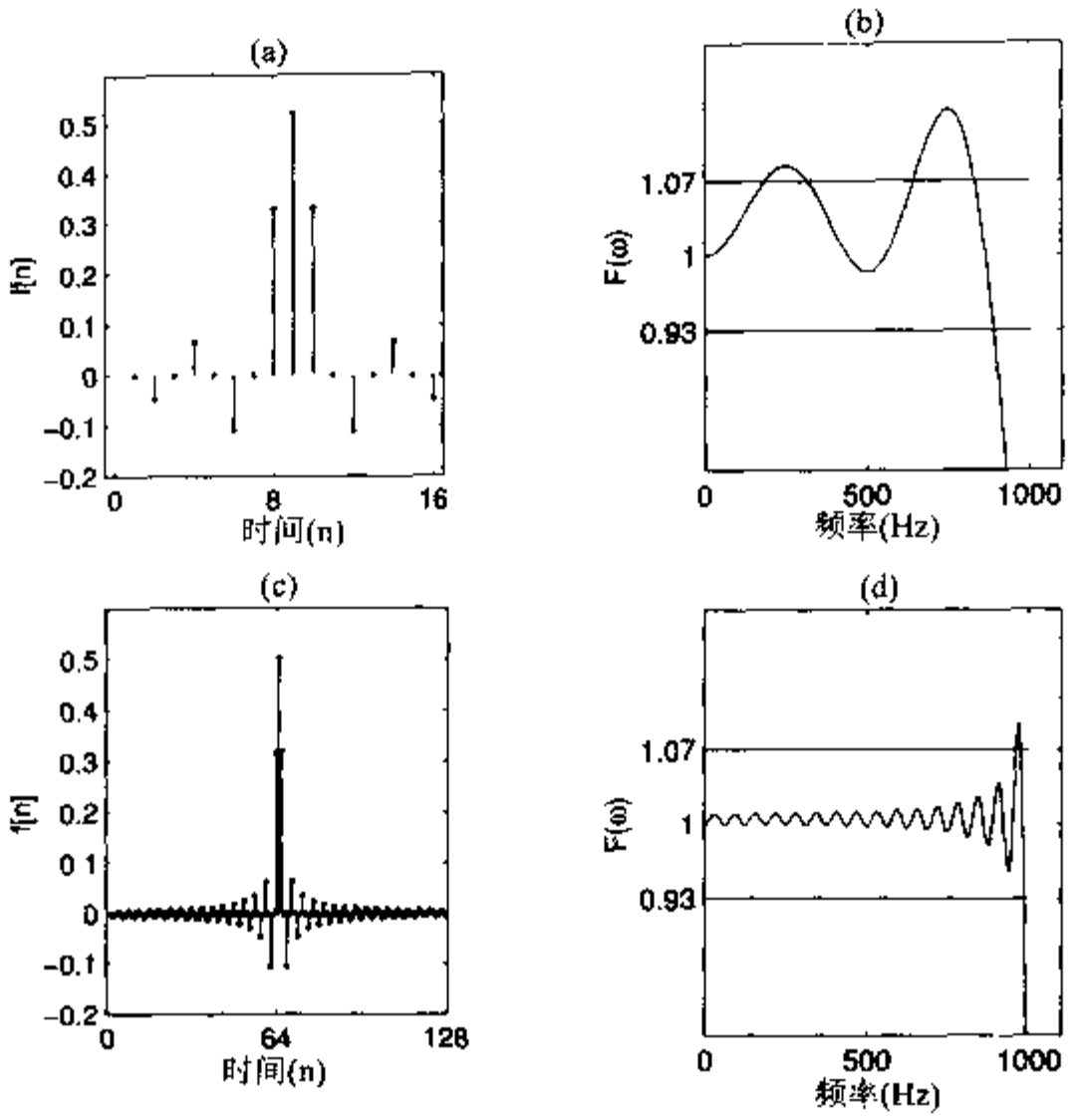


图 3-6　吉布斯(Gibbs)现象 (a) $L=16$ 的 FIR 低通滤波器的脉冲响应 (b) 传递函数 $L=16$ 的通频带。(c) $L=128$ 的 FIR 低通滤波器的脉冲响应 (d) 传递函数 $L=128$ 的通频带

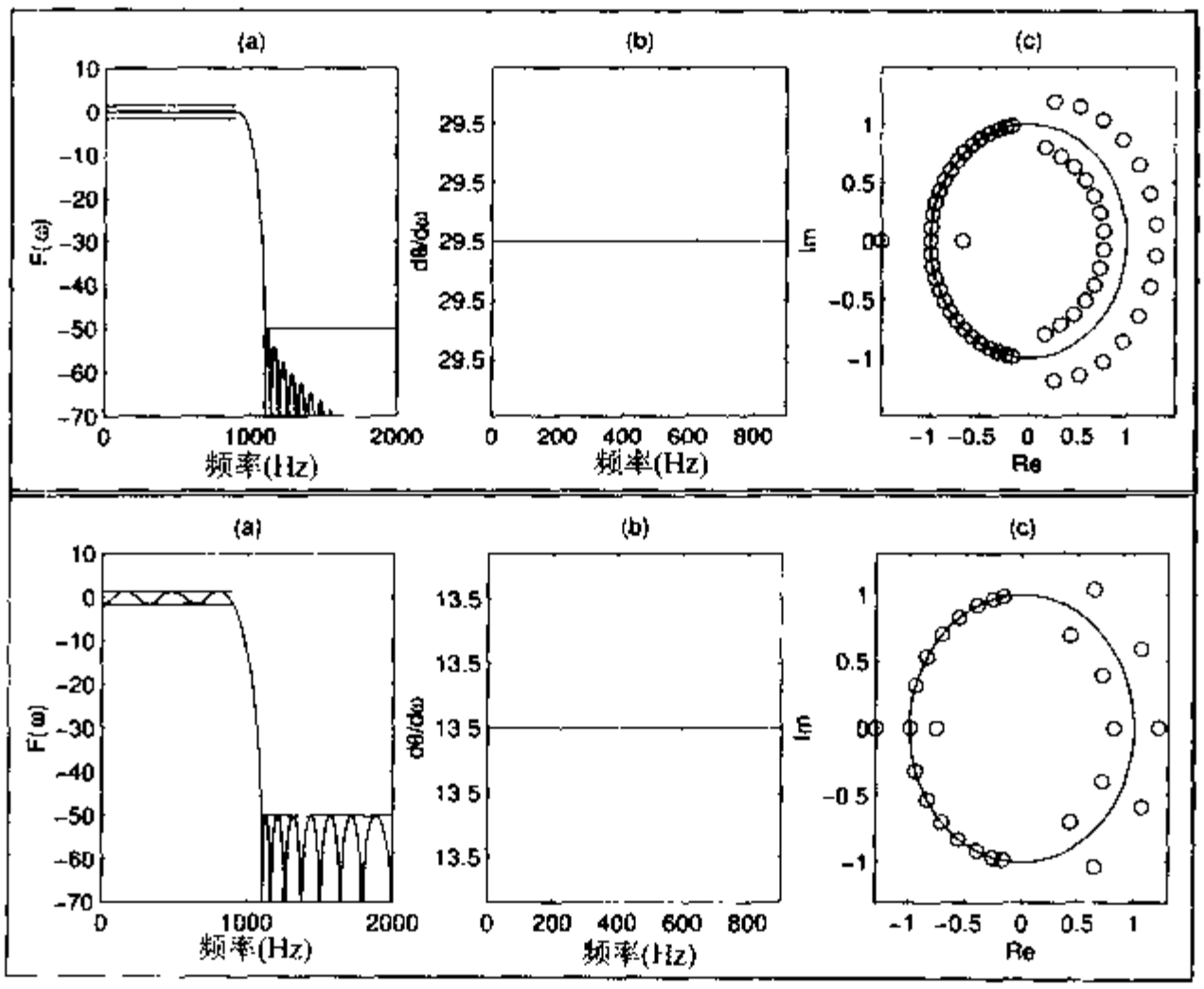


图 3-7　(上) $L=59$ 的 Kaiser 窗函数设计(下) $L=27$ 的 Parks—McClellan 设计 (a) 传递函数 (b) 通频带的组延迟 (c) 零极点

表 3-2　常用窗函数的参数

名　　称	3dB 带 宽	第一个零点	最 大 旁 瓣	每倍频程的旁瓣衰减	等效 Kaiserβ
矩形	0.89/T	1/T	−13dB	−6dB	0
Bartlett(三角形)	1.28/ T	2/ T	−27dB	−12dB	1.33
汉宁(Hanning)	1.44/ T	2/ T	−32dB	−18dB	3.86
加重平均 (Hamming)	1.33/ T	2/ T	−42dB	−6dB	4.86
Blackman	1.79/ T	3/ T	−74dB	−6dB	7.04
Kaiser	1.44/ T	2/ T	−38dB	−18dB	3

表 3-2 中的 3dB 带宽是传递函数从直流衰减到 3dB，也就是$\approx 1/\sqrt{2}$时的带宽。数据窗口也会产生旁瓣，有远离第 0 次谐波的各种各样的角度。表 3-2 的第三列依照窗口的光滑程度给出了在第一个或者第二个零 DFT 频率 1/T 没有零点的一些窗函数。根据测量，最大旁瓣增益与第 0 次谐波值有关。第五列描述了窗函数每倍频程渐进的减少。最后一列仿照相应的窗函数特性为 Kaiser 窗函数给出了值 β。建立在一阶贝塞尔函数 I_0 基础上的 Kaiser 窗函数在两个方面有所特别。就“阻尼振荡”抑制和过渡频带之间的宽度而言，它是非常接近最优的，第二就是可以由 β 调谐，后者决定滤波器的阻尼振荡。可以从接下来归于 Kaiser 的方程看出：

$$\beta=\begin{cases}0.1102(A-8.7) & A>50,\\ 0.5842(A-21)^{0.4}+0.07886(A-21) & 21\le A\le 50\\ 0 & A<21\end{cases} \tag{3.13}$$

其中 $A=20\log_{10}\varepsilon$，既是抑止频带衰减，也是通频带纹波的 dB 数。Kaiser 窗函数要得到一个需要的衰减层次，可以根据如下公式估算：

$$L=\frac{A-8}{2.285(\omega_s-\omega_p)}+1 \tag{3.14}$$

通常的长度误差是±2 个脉冲线。

3.3.2　等同纹波设计方法

典型的滤波器规格说明不仅仅包括通频带 ω_p 和抑止频带 ω_s 的频率以及理想增益的规格说明，还应该考虑到与期望传递函数之间的偏差(或者是纹波)。最为常见的就是假定在纹波方面，过渡频带是任意的。一类非常有效的符合这些规格说明的 FIR 滤波器就称作**等同纹波 FIR 滤波器**。等同纹波设计协议最小化了与理想传递函数之间的最大偏差(纹波误差)。等同纹波算法适用于很多 FIR 设计情况。最为流行的就是：

- 低通滤波器设计(在 MATLAB 中用 remez(L，F，A，W)表示)，公差设计方案如图 3-8(a)所示。

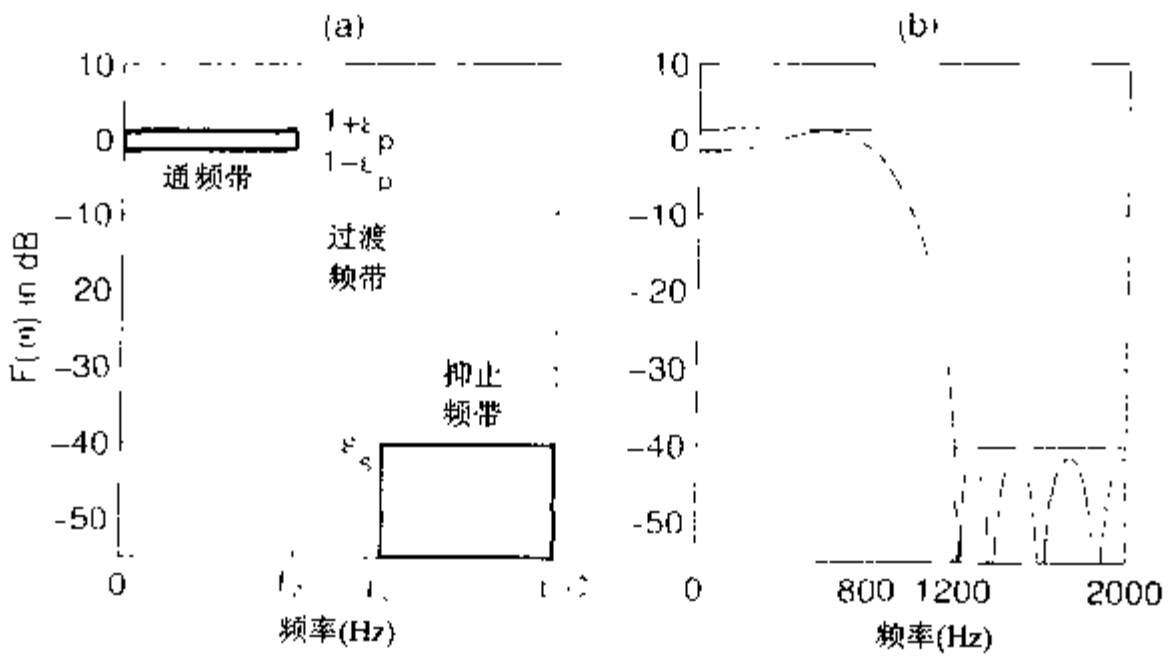


图 3-8　滤波器设计的参数 (a) 公差设计方案 (b) 满足公差设计方案的示例函数

● 希尔伯特(Hilbert)滤波器，也就是一种对通频带内所有频率都产生 90° 相位移的单元幅值滤波器(在 MATLAB 中用 remez(L，F，A，‘Hilbert’)表示)。

● 微分滤波器、频率和幅值随着 ω 成正比例地线性增加(在 MATLAB 中用 remez(L，F，A，‘differentiator’)表示)。

等同纹波或最小-最大算法通常都是采用 Parks-McClellan 迭代方法来实现的。可以利用 Parks-McClellan 方法生成适用于频域中的等同纹波或者极小化极大数据。这是根据“交错定理”得来的，交错定理认为存在符合公差设计方案的惟一 Chebyshev 多项式，且该各项式具有最小的长度。图 3-8(a)给出了这一公差设计方案，并且图 3-8(b)给出了满足这一方案的多项式。对于低通而言，多项式的长度，也就是滤波器的长度，可以根据下面的公式来计算：

$$L=\frac{-10\log_{10}(\varepsilon_p\varepsilon_s)-13}{2.324(\omega_s-\omega_p)}+1 \tag{3.15}$$

其中 ε_p 是通频带纹波，ε_s 是抑止频带纹波。

迭代的算法可以发现偏离正常值的局部最大误差的位置，降低每次迭代最大误差的规模，直到所有的偏离误差都具有相同值。更为常见的是，Remez 方法经常用来通过选出两次迭代之间误差曲线的最大尖峰的频率集合来选择新频率(请参阅【65】)。这就是为什么 MATLAB 中等同纹波函数称作 remez 的缘故。

与直接频率方法相比较，区别是有或者没有数据窗口，等同纹波设计方法的优点在于通频带和抑止频带偏差可以分别指定。例如，这在音频领域中是非常有用的，通频带中的纹波可以规定的更高一些，因为人耳只能够觉察到大于 3dB 的差别。

从图 3-7(下图)中可以看到，等同纹波设计与 Kaiser 窗函数设计倾向于明显降低滤波器阶数(也就是 27 比 59)一样，也具有同样的公差要求。

3.4 常系数 FIR 设计

仅在极少数应用场合(例如：自适应滤波器)才需要像例 3.1 中给出的通用可编程滤波器结构。在很多应用场合，滤波器都是 LTI 的(也就是线性时间不变量，linear time-invariant)，系数

不随时间变化，在这种情况下，硬件方面的工作基本上就降低到开发实现 FIR 滤波器算法所需要的乘法器和加法器(树)了。

利用可行的数字滤波器设计软件，FIR 系数的生成就是一种直接了当的过程。直接或者转置形式都是根据最大速度和最低资源的使用而定的。网格滤波用在自适应滤波器之中，这是因为滤波器不需要重新计算以前的网格节就可以一节一节的增加，但是这一特性只适用于 PDSP，而对于 FPGA 则很少采用。所以我们要将注意力集中在直接和转置形式的滤波器实现上。首先将对直接形式做可行性改进，随后就转到转置形式上来。在本节的结尾，我们将讨论一个采用分布式算法的另一设计法。

3.4.1　直接 FIR 设计

图 3-1 所示的直接 FIR 滤波器在 VHDL 中使用(顺序)PROCESS 声明或者是加法器和乘法器的“组件实例”来实现。PROCESS 设计为合成器提供了更多的自由，而组件实例则被设计者可以完全控制。为了说明这一点，下面将要给出一个长度为 4 的 FIR 滤波器作为 PROCESS 设计。尽管长度为 4 的 FIR 对于大多数实际应用来讲都太短了，但是它可以很容易地扩展到更高阶，并且其优点在于编辑时间比较短。线性相位(也就是对称)FIR 脉冲响应假定如下：

$$f[k] = \{-1.0, 3.75, 3.75, -1.0\} \tag{3.16}$$

这些系数可以直接编码成 4 位分数。例如：3.75_{10} 的 4 位二进制表达式是 11.11_2，其中“.”表示二进制小数点的位置。注意：通常仅仅实现正 CSD 系数时会更有效，因为正 CSD 系数具有更少的非零项，当计算乘积的累加时可以将系数的符号考虑进来。还请参阅后面 3.4 中将要讨论的 RAG 算法的前两个步骤。

在实际情况中，FIR 可以从计算机设计工具中得到，是以浮点数形式提供给设计者的。建立在浮点数系数基础之上的定点数 FIR 的实现，需要通过仿真或者代数分析进行确认，以保证设计规范仍然是令人满意的。在上述例子中，浮点数是 3.75 和 1.0，它们都是用定点数来精确地表示的，所以这种核查就可以跳过。

当用定点数设计时需要提出的另一个问题就是保护系统不要动态范围溢出。幸运的是，第 L 阶 FIR 的动态范围级数 G 的最坏情况可以很容易地计算出来，就是：

$$G \le \log_2\left(\sum_{k=0}^{L-1} |f[k]|\right) \tag{3.17}$$

总位宽就是输入位宽与级数 G 的位宽之和。对于上面(3.16)中的滤波器 $G=\log_2(9.5)<4$，这说明系统内部数据寄存器需要至少比输入数据多 4 个以上的整数位以保证不溢出。如果采用的是 8 位内部运算，输入数据就应该限制在 $\pm 128/9.5=\pm 13$ 之内。

例 3.2　4 抽头直接 FIR 滤波器

系数为{－1,3.75,3.75,－1}的滤波器的 VHDL 设计[2]如下：

```
PACKAGE eight_bit_int IS      -- User defined types
```

注 2：这一例子相应的 Verilog 代码文件 fir_srg.v 可以在附录 A 中找到。

```
  SUBTYPE BYTE IS INTEGER RANGE  - 128 TO 127;
  TYPE ARRAY_BYTE IS ARRAY (0 TO 3) OF BYTE;
END eight_bit_int;

LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY fir_srg IS                              ------> Interface
  PORT (clk     :   IN   STD_LOGIC;
        x       :   IN   BYTE;
        y       :   OUT BYTE);
END fir_srg;

ARCHITECTURE flex OF fir_srg IS

  SIGNAL tap : ARRAY_BYTE;     -- Tapped delay line of bytes

BEGIN

  p1: PROCESS                  ------> Behavioral Style
  BEGIN
    WAIT UNTIL clk = '1';
  -- Compute output y with the filter coefficients weight.
  -- The coefficients are [ - 1   3.75   3.75   - 1].
  -- Multiplication and division for Altera VHDL are only
  -- allowed for powers of two!
    y <= 2 * tap(1) + tap(1) + tap(1) / 2 + tap(1) / 4
         + 2 * tap(2) + tap(2) + tap(2) / 2 + tap(2) / 4
         - tap(3)  - tap(0);
    FOR I IN 3 DOWNTO 1 LOOP
      tap(I) <= tap(I - 1); -- Tapped delay line: shift one
    END LOOP;
    tap(0) <= x;                       -- Input in register 0
  END PROCESS;

END flex;
```

这一设计是对图 3-1 中直接 FIR 滤波器结构的文字解释，这种设计对对称和非对称滤波器都是适用的。抽头延迟线每个抽头的输出分别乘以相应加权的二进制值，再将结果相加。对应脉冲 10 的滤波器脉冲响应 y 如图 3-9 所示。注意：MaxPlusII 是以无符号数显示 - 10 的，也就是 256 - 10=246。

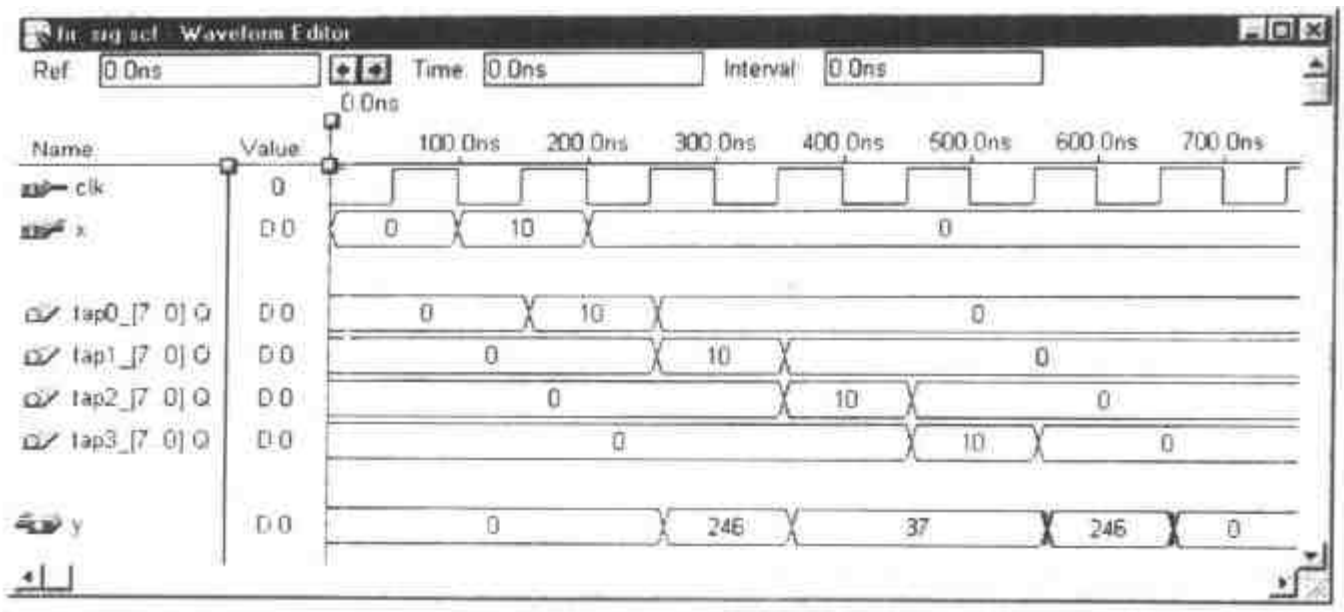


图 3-9　脉冲输入为 10 时 FIR 滤波器的 VHDL 仿真结果

有 3 种显而易见的措施可以改进这一设计：

(1) 用最优 CSD 码(请参阅第 2 章例 2.1)实现每个滤波器系数。

(2) 通过流水线来提高有效的乘法器速度。输出加法器应该安排在流水线平衡树中。如果系数被编码成“2 的幂”形式，流水线乘法器和加法器树就可以合并。流水线技术具有较低的成本，这是因为通常不应用 LC 寄存器的缘故。如果在树中相加项的数目不是“2 的幂”形式，可能还会需要少量额外的流水线寄存器。

(3) 对于对称系数而言，乘法的复杂性可以降低到如图 3-5 所示的程度。

前两项措施对所有 FIR 寄存器都是适用的，而第三种措施只对线性相位(对称)滤波器适用。下面通过举设计示例来说明这些想法。

例 3.3　改进的 4 抽头直接 FIR 滤波器

从前面示例中改进而得的设计对系数采用了 CSD 码，就是 $3.75=2^2-2^{-2}$。此外还可以采用对称和流水线来提高滤波器的性能。表 3-3 给出了每种不同设计的期望的最大吞吐量。CSD 编码与对称都生成了更小更为简洁的设计。Registered Performance 的改进可以通过流水线技术处理乘法器，并为输出累加提供一个加法器树来实现。但是还需要两个额外的流水线寄存器(也就是 16 个 LC)。最为简洁的设计是采用对称性和 CSD 编码，并且没有加法器树实现的。产生滤波器输出 y 的部分 VHDL 代码如下：

```
WAIT UNTIL clk = '1';
t1 <= tap(1) + tap(2);--Using symmetry
t2 <= tap(0) + tap(3);

y <= 4* t1 – t1 / 4 –t2;--Apply CSD code and add
…
```

最快速的设计是采用所有 3 种提高措施实现的。在这种情况下，部分 VHDL 代码变成如下形式：

```
WAIT UNTIL clk = '1';
t1 <= tap(1) + tap(2);   --Using symmetry of coefficients
t2 <= tap(0) + tap(3);
```

```
t3 <= 4 * t1 – t1 / 4;      --Pipelinged multiplier
t4 <=  - t2;                --Build a binary tree and add delay
y <= t3 + t4;
…
```

表 3-3　改进的 FIR 滤波器

对称性	否	是	否	否	是	是
CSD	否	否	是	否	是	是
树	否	否	否	是	否	是
速度/MHz	18.05	37.17	37.74	100.00	56.81	104.16
规模/LC	104	80	70	120	64	72

1. FIR 滤波器中的再定相流水线乘法器

经常是个别系数要比其他所有系数的流水线延迟都多。我们可以用 $f[n]z^{-d}$ 来模拟这种延迟。如果我们现在加上一个正延迟：

$$f[n]=z^{d}f[n]z^{-d} \tag{3.18}$$

两个延迟就可以相互抵消了。将此转换成硬件形式就意味着对于直接形式的 FIR 滤波器，我们必须使用寄存器前面第 d 个位置的输出。

这一原则如图 3-10(a)所示。图 3-10(b)给出了具有两个延迟的再定相流水线乘法器的一个示例。

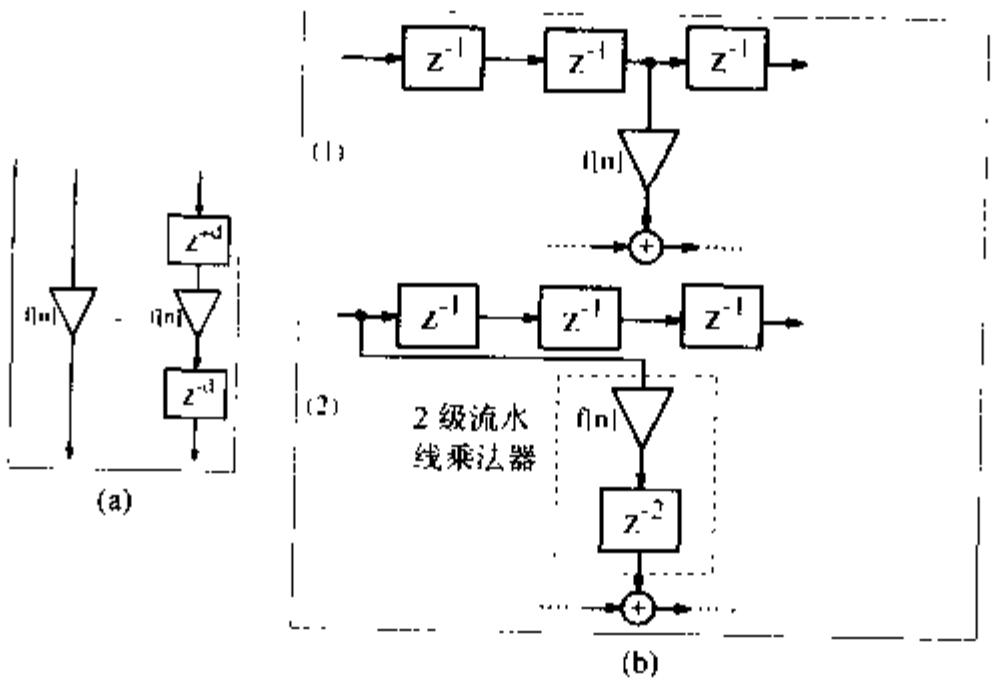


图 3-10　再定相 FIR 滤波器(a)原则 (b)再定相一个乘法器
(1)无流水线结构 (2)带有二阶流水线结构

3.4.2　具有转置结构的 FIR 滤波器

直接 FIR 滤波器的一种变化称为**转置滤波器**，我们已经在 3.2.1 节加以讨论了。就常系数滤波器来讲，与直接形式的滤波器相比较，转置滤波器在以下两个方面有所改进：

- 重复系数的多重使用,采用了简化加法器图(Reduced Adder Graph, RAG)算法[27,28,29,30]
- 流水线加法器采用的是进位存储加法器

流水线加法器可以提高速度，但是需要额外的加法器和寄存器成本，而 RAG 原则会降低滤波器的规模(也就是 LC 的数量)，而且通常也会提高速度。我们已经在第 2 章讨论过流水线加法器原则了，这里我们将集中研究 RAG 算法。

在第 2 章中已经提到，与直接实现 CSD 编码相比，实现常系数的因子经常更容易一些。例如：常数乘法器系数 45 的 CSD 编码的实现需要 3 个加法器，而因子 5×9 只需要 2 个加法器。对于转置滤波器而言，通常所有系数都具有几个因子的可能性是非常大的。例如系数 9 和 11，可以用 8+1 构造成 9，用 9+2 构成 11。这样就将总工作量降低到了一个加法器。但是，通常找到最优 RAG 是一件非常艰苦的事情，最终还要依靠直观判断。

当然还是有一些显而易见的措施可以降低工作量的，特别是：

算法 3.4　简化加法器图

(1) 去掉系数的符号，因为符号可以通过滤波器的抽头延迟线上的减法来实现。
(2) 去掉所有是 2 的幂的系数和因子，因为可以通过硬连线的数据移位来实现。
(3) 实现所有值为 1 的系数。
(4) 用值为 1 的系数构造更高值的乘法器。

步骤(1)~(3)均可以直接实现，但是步骤(4)可能比较复杂，因为理论上图的数量是呈指数增加的。为了简化这一过程，使用表 2-3 给出的 CSD 编码算法会有所帮助。为了说明 RAG 算法，我们来研究 Goodman 和 Carey[66] 定义的 F6 半频带 FIR 滤波器的系数的编码。

例 3.5　F6 半频带滤波器的简化加法器图

F6 半频带滤波器有 4 个非零系数，称为 $f[0]$、$f[1]$、$f[3]$和 $f[5]$，它们分别是 346、208、-44 和 9。为了首次值的评估，我们将十进制值(下标为 10)转换成二进制表达式，并查找来自表 2-3 的系数的值，其过程如下：

	$f[k]$			值
$f[0]=346_{10}$	$=2\times173$	$=$	101011010_2	4
$f[1]=208_{10}$	$=2^4\times13$	$=$	11010000_2	2
$f[3]=-44_{10}$	$=-2^2\times11$	$=$	-1011010_2	2
$f[5]=9_{10}$	$=3^2$	$=$	1001_2	1
	总和			9

对于直接 CSD 码的实现，需要 9 个加法器，其 RAG 算法处理如下：

步骤	要实现的	已经实现的	措施
(0)	{346,208, - 44,9}	{—}	初始化
(1)	{346,208,44,9}	{—}	非负系数
(2)	{346,208,44,9}	{—}	除去 2^k 系数
(3)	{173,13,11,9}	{—}	从系数中除去 2^k 因子
(4)	{173,13,11}	{9}	实现值为 1 的系数

(5)　{173,13,11}　　{9}　　其他系数是质数

对剩余的系数进行直观判断，由最低成本和最小值的系数开始，具体如下：

步骤	实现	已经实现	寻找表达式的措施
(6)	{173,13}	{9,11}	11=9+2
(7)	{173}	{9,11,13}	13=9+4
(8)	{—}	{9,11,13,173}	173=(4+1)9+128

图 3-11 给出了最终的已简化的加法器图。加法器的数量从 9 降低到 5，加法器路径延迟也从 4 降低到 3。

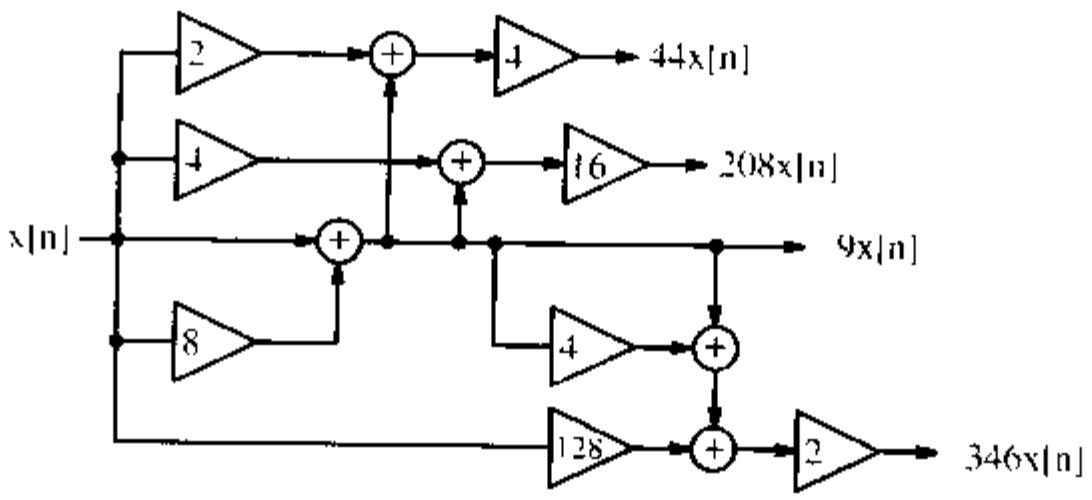


图 3-11　应用 RAG 算法的 F6 的实现

3.4.3　采用分布式算法的 FIR 滤波器

这种完全不同的 FIR 滤波器结构是建立在 2.5.1 节中介绍的分布式算法(DA)概念基础上的。与传统的乘积-和结构相比，在分布式算法中，我们总是计算具体位 b 在一个步骤中通过所有系数的乘积和。这种计算只需要一个小表和一个附带移位器的累加器即可。

为了说明该问题，我们来研究例 2.17 中 3 个系数的 FIR 滤波器。

例 3.6：作为状态机的分布式算法滤波器

在 VHDL 代码[3]中，分布式算法滤波器可以利用下面的状态机描述：

```
LIBRARY ieee;                          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
PACKAGE da_package IS                  -- User defined component
  COMPONENT case3
    PORT ( table_in    : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out   : OUT   integer RANGE 0 TO 6);
  END COMPONENT;
END da_package;

LIBRARY work;
```

注 3：　这一例子相应的 Verilog 代码文件 dafsm.v 可以在附录 A 中查到。

```
USE work.da_package.ALL;

LIBRARY ieee;                    -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
ENTITY dafsm IS                          ------> Interface
        PORT (clk   : IN STD_LOGIC;
              x_in0, x_in1, x_in2 :
                        IN   STD_LOGIC_VECTOR(2 DOWNTO 0);
              y     : OUT INTEGER RANGE 0 TO 63);
END dafsm;

ARCHITECTURE flex OF dafsm IS
  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state      : STATE_TYPE;
  SIGNAL x0, x1, x2, table_in
                              : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL table_out : integer RANGE 0 TO 7;
BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

PROCESS                    ------> DA in behavioral style
  VARIABLE p      : integer RANGE 0 TO 63;-- temp. register
  VARIABLE count : integer RANGE 0 TO 3; -- counts shifts
BEGIN
  WAIT UNTIL clk = '1';
  CASE state IS
    WHEN s0 =>             -- Initialization step
      state <= s1;
      count := 0;
      p := 0;
      x0 <= x_in0;
      x1 <= x_in1;
      x2 <= x_in2;
    WHEN s1 =>             -- Processing step
      IF count = 3 THEN    -- Is sum of product done ?
        y <= p;            -- Output of result to y and
        state <= s0;       -- start next sum of product
      ELSE
        p := p / 2 + table_out * 4;
        x0(0) <= x0(1);
        x0(1) <= x0(2);
```

```
            x1(0) <= x1(1);
            x1(1) <= x1(2);
            x2(0) <= x2(1);
            x2(1) <= x2(2);
            count := count + 1;
            state <= s1;
          END IF;
      END CASE;
    END PROCESS;

    LC_Table0: case3
      PORT MAP(table_in => table_in, table_out => table_out);

  END flex;
```

定义成 CASE 组件的 LC 表[4]由实用程序 dagen.exe 生成。其输出如下：

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case3 IS
        PORT ( table_in   : IN   STD_LOGIC_VECTOR(2 DOWNTO 0);
               table_out : OUT INTEGER RANGE 0 TO 6);
END case3;

ARCHITECTURE LCs OF case3 IS
BEGIN

-- This is the DA CASE table for
-- the 3 coefficients: 2, 3, 1
-- automatically generated with dagen.exe -- DO NOT EDIT!

  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN  "000" =>     table_out <=  0;
      WHEN  "001" =>     table_out <=  2;
      WHEN  "010" =>     table_out <=  3;
      WHEN  "011" =>     table_out <=  5;
      WHEN  "100" =>     table_out <=  1;
      WHEN  "101" =>     table_out <=  3;
      WHEN  "110" =>     table_out <=  4;
      WHEN  "111" =>     table_out <=  6;
```

注 4：这一例子相应的 Verilog 代码文件 case3.v 可以在附录 A 中查到。

```
        WHEN    OTHERS   =>      table_out <=   0;
    END CASE;
  END PROCESS;
END LCs;
```

正如第 2 章所提到的，采用了一个移位/累加器，每一步骤只向右移动一个位置，而不是向左移动 k 个位置。图 3-12 给出了仿真结果以及对应输入序列{1,3,7}的正确结果(y=18)。这一设计以 61.72MHz 的 Registered Performance 运行，使用了 37 个 LC，没有采用 EAB。

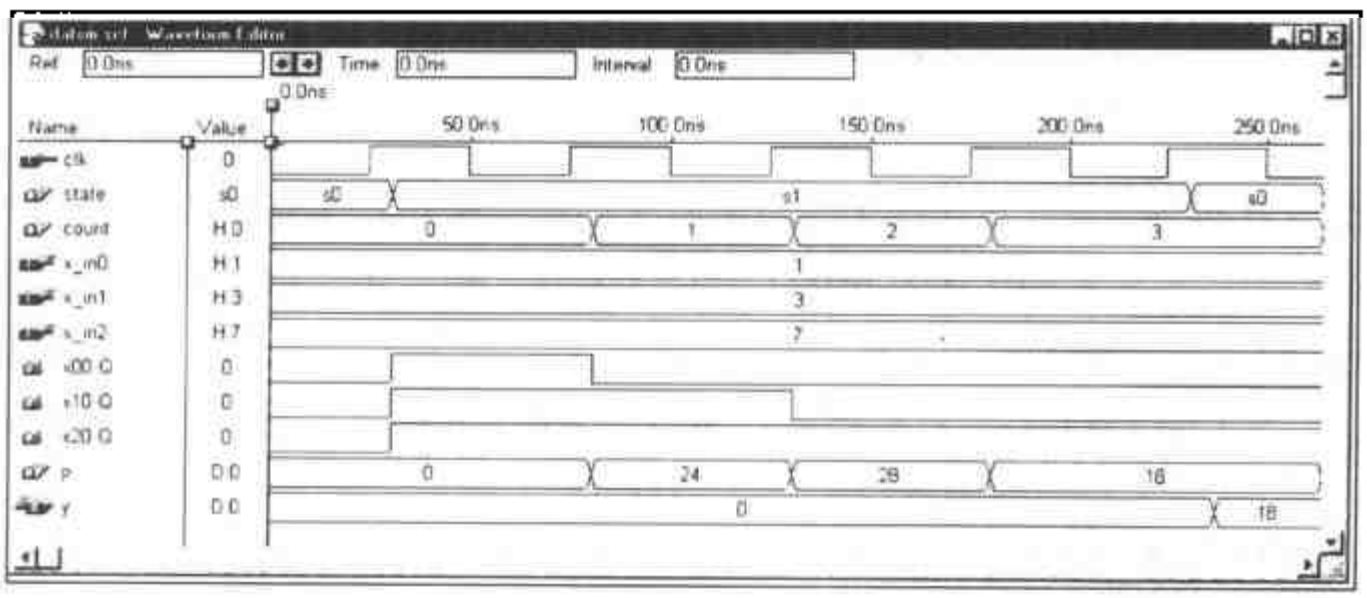


图 3-12　3 抽头 FIR 滤波器在输入{1,3,7}时的仿真结果

通过利用 CASE 声明来定义分布式算法表，合成器可以使用逻辑单元来实现 LUT 表。如果表比较小，就可以生成快捷有效的设计。对于较大的表，必须找到可替代的方法。在此情况下，我们可以使用 2KB 嵌入式阵列模块(embedded array block, EAB，我们已经在第 1 章中讨论过)，可以将 EAB 设定成 $2^8\times8$，$2^9\times4$，$2^{10}\times2$ 或者是 $2^{11}\times1$ 的表。我们将在下面详细地讨论这两个设计。

1. 采用逻辑单元的分布式算法

对于低阶情形而言，由于 LUT 表地址空间所限(例如：$L\leq4$)，FIR 滤波器的 DA 实现是非常具有吸引力的。应该记住，无论如何，FIR 滤波器都是线性滤波器。这就意味着低阶滤波器输出的集合可以相加，并由此定义一个高阶 FIR 滤波器的输出，如图 2-19 所示。建立在 FLEX10K 元器件的 LC 之上即 $2^4\times1$ 位的 DA 表，可以实现 4 个系数。需要的 LC 数量随着阶数呈指数增加。通常，LC 的数量要比 EAB 的数量多得多。例如：EPF10K20RC240-4 包含 1152 个 LC，但是只包含 6 个 EAB。而且 EAB 还可以用来非常有效地实现 RAM 和 FIFO 以及其他高位值函数。所以经常要节约使用 EAB。在未注册的模式下使用 EAB，会降低设计的最大带宽。如果要实现的设计采用了更大的表和 $2^b\times b$ 的 CASE 声明，可能会导致无效的设计。即使选择了“Global Project Logic Synthesis”中的“Optimize Area”选项，以及禁用了“Reduce Logic”和“Duplicate Logic Extraction”选项，这样做可以为 CASE 表给出最优区域，但还是要生成一个比期望大得多的设计。例如：$2^7\times7$ 的表需要 394 个 LC。图 3-13 给出了采用 CASE 声明具有 3 到 8 个输入的表所需要的 LC 数量。最小规模曲线是从 FLEX10K 的一个 4→1 多路复用器可以用 2 个 LC 构成(采用级联与门)这一实践中获得的。而一个 16→1 的多路复用器则是由 10 个 LC 构成的(请参阅 MaxPlusII 有关 busmux 的帮助)。

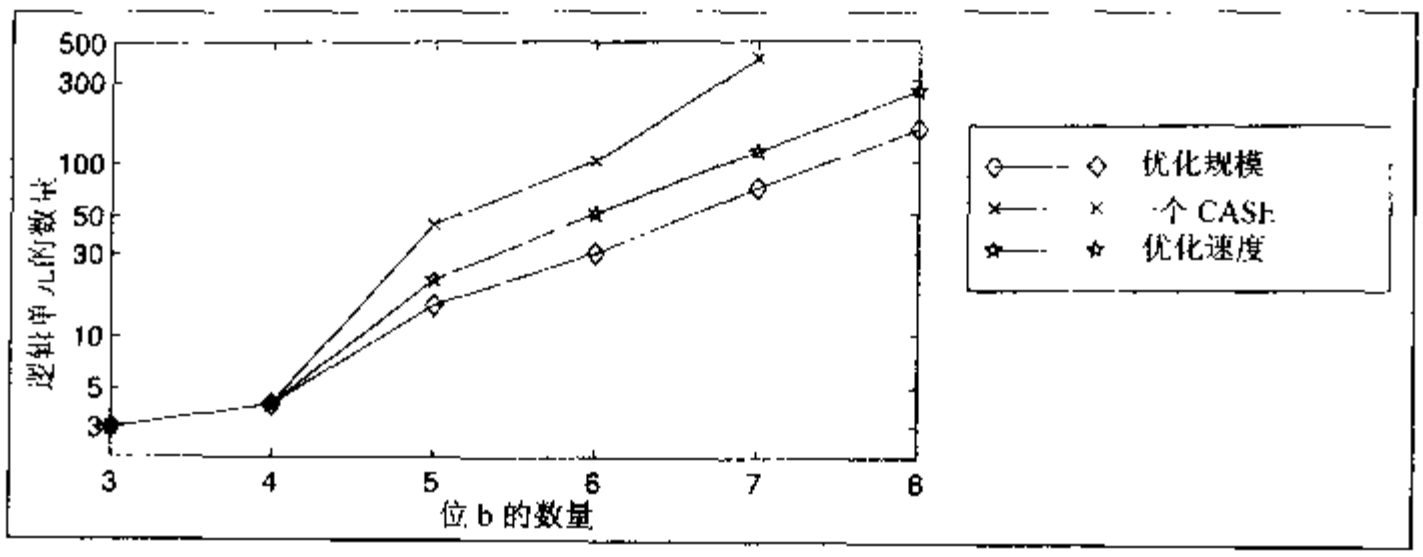


图 3-13　采用 CASE 声明的不同编码的合成结果 6 位输入和输出的规模比较

典型的合成器都会尽量优化逻辑方程，但是不能识别结构。通常利用 CASE 声明实现附带一个(总线)多路复用器的 4 输入表是更为有效的。在这一模式中，也是直接加上流水线寄存器进行模块化设计的。为了获得最大速度，必须在每个 2→1 多路复用器后引入一个寄存器。当然，与用 2 个 LC 实现 1 个 4→1 多路复用器相比还是需要更多 LC 的。下面的示例就说明了 5 输入表的结构。

例 3.7　5 输入 DA 表

对于附带(总线)多路复用器的 4 输入表，实用程序 dagen.exe 接收其滤波长度和系数，并且返回需要的 PROCESS 声明。下面的程序清单给出了对应一个任意系数集合{1,3,5,7, 9}的 VHDL 代码[5]输出：

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case5p IS
        PORT ( clk          : IN   STD_LOGIC;
               table_in     : IN   STD_LOGIC_VECTOR(4 DOWNTO 0);
               table_out    : OUT INTEGER RANGE 0 TO 25);
END case5p;

ARCHITECTURE LCs OF case5p IS

  SIGNAL lsbs : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL msbs0 : STD_LOGIC_VECTOR(1 DOWNTO 0);
  SIGNAL table0out00, table0out01 : INTEGER RANGE 0 TO 25;

BEGIN

-- These are the distributed arithmetic CASE tables for
-- the 5 coefficients: 1, 3, 5, 7, 9
-- automatically generated with dagen.exe -- DO NOT EDIT!
```

注 5：这一例子相应的 Verilog 代码文件 case5p.v 可以在附录 A 中找到。

```
PROCESS
BEGIN
   WAIT UNTIL clk = '1';
   lsbs(0) <= table_in(0);
   lsbs(1) <= table_in(1);
   lsbs(2) <= table_in(2);
   lsbs(3) <= table_in(3);
   msbs0(0) <= table_in(4);
   msbs0(1) <= msbs0(0);
END PROCESS;

PROCESS      -- This is the final DA MPX stage.
BEGIN        -- Automatically generated with dagen.exe
   WAIT UNTIL clk = '1';
   CASE msbs0(1) IS
      WHEN  '0' =>     table_out <=  table0out00;
      WHEN  '1' =>     table_out <=  table0out01;
      WHEN   OTHERS  =>     table_out <=  0;
   END CASE;
END PROCESS;

PROCESS      -- This is the DA CASE table 00 out of 1.
BEGIN        -- Automatically generated with dagen.exe
   WAIT UNTIL clk = '1';
   CASE lsbs IS
      WHEN  "0000" =>     table0out00 <=  0;
      WHEN  "0001" =>     table0out00 <=  1;
      WHEN  "0010" =>     table0out00 <=  3;
      WHEN  "0011" =>     table0out00 <=  4;
      WHEN  "0100" =>     table0out00 <=  5;
      WHEN  "0101" =>     table0out00 <=  6;
      WHEN  "0110" =>     table0out00 <=  8;
      WHEN  "0111" =>     table0out00 <=  9;
      WHEN  "1000" =>     table0out00 <=  7;
      WHEN  "1001" =>     table0out00 <=  8;
      WHEN  "1010" =>     table0out00 <=  10;
      WHEN  "1011" =>     table0out00 <=  11;
      WHEN  "1100" =>     table0out00 <=  12;
      WHEN  "1101" =>     table0out00 <=  13;
      WHEN  "1110" =>     table0out00 <=  15;
      WHEN  "1111" =>     table0out00 <=  16;
      WHEN   OTHERS  =>     table0out00 <=  0;
   END CASE;
END PROCESS;

PROCESS      -- This is the DA CASE table 01 out of 1.
BEGIN        -- Automatically generated with dagen.exe
```

```
    WAIT UNTIL clk = '1';
    CASE lsbs IS
      WHEN   "0000" ->     table0out01 <=   9;
      WHEN   "0001" =>     table0out01 <=   10;
      WHEN   "0010" =>     table0out01 <=   12;
      WHEN   "0011" =>     table0out01 <=   13;
      WHEN   "0100" =>     table0out01 <=   14;
      WHEN   "0101" =>     table0out01 <=   15;
      WHEN   "0110" =>     table0out01 <=   17;
      WHEN   "0111" =>     table0out01 <=   18;
      WHEN   "1000" =>     table0out01 <=   16;
      WHEN   "1001" =>     table0out01 <=   17;
      WHEN   "1010" =>     table0out01 <=   19;
      WHEN   "1011" =>     table0out01 <=   20;
      WHEN   "1100" =>     table0out01 <=   21;
      WHEN   "1101" =>     table0out01 <=   22;
      WHEN   "1110" =>     table0out01 <=   24;
      WHEN   "1111" =>     table0out01 <=   25;
      WHEN    OTHERS   =>     table0out01 <=   0;
    END CASE;
  END PROCESS;
END LCs;
```

5 个输出生成 2 个 CASE 表和 1 个 2→1 总线多路复用器。多路复用器也可以由使用 LPM 的 busmux 函数组件实例来实现。dagen.exe 程序写入一个名为 caseX.vhd 的 VHDL 文件。其中 X 是滤波器的长度，也是输入位宽。文件 caseXp.vhd 除了额外的流水线寄存器以外，也是同样的表。Component 可以直接用在状态机设计或开式的滤波器结构之中。

参看图 3-13，可以看到结构化 VHDL 代码增加了所需要的 LC 数量。图 3-14 就速度方面对不同设计方法进行了比较。减少 LC 的数量也可以提高吞吐量，这是因为 LC 层的数量也相应降低了。尽管为了实现一个 $2^8 \times 8$ 的表，采用了 7 个流水线级，并以 88.49MHz 的较高 Registered Performance 运行，但是对于一些实际应用场合来讲，这一设计还是有些太大。我们还可以考虑采用图 2-18 中给出的分块技术(练习 3.6)，或者下面将要讨论的利用一个 EAB 来实现。

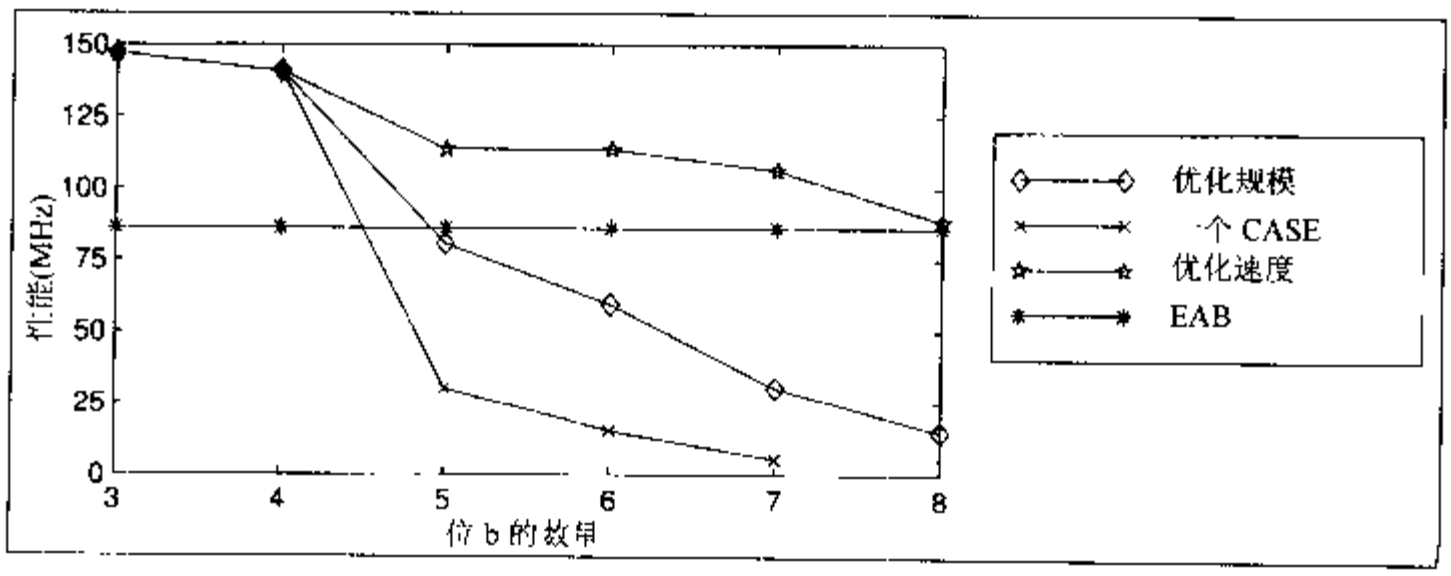


图 3-14　采用 CASE 声明的不同编码类型的速度比较

2. 采用嵌入式阵列模块的 DA

正如在下面最后一部分提到的，采用一个 2KB 的 EAB 来实现简单的滤波器是非常不经济的，主要是因为可用的 EAB 数量有限。还有就是 EAB 的最大 Registered Performance 是 76MHz，采用 LC 表实现可能会更快一些。下面的例题给出了采用 EAB 的组件例示的 DA 实现。

例 3.8　采用 EAB 的分布式算法滤波器

上一个例题中的 CASE 表可以由一个 EAB ROM 来代替。ROM 表是由文件 da3.mif 定义的。EAB 的默认输入输出配置是由“REGISTERED”给出的。如果不是所需要的配置，可以设置 LPM_ADDRESS_CONTROL =>“UNREGISTERED”和 LPM_OUTDATA=>“UNREGISTERED”。状态机设计的 VHDL 代码[6]如下：

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;                          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;   -- Contains conversion
                                       -- VECTOR -> INTEGER
ENTITY darom IS                                  ------> Interface
  PORT (clk   : IN STD_LOGIC;
        x_in0, x_in1, x_in2
                                  : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
        y     : OUT INTEGER RANGE 0 TO 63);
END darom;

ARCHITECTURE flex OF darom IS
  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state                          : STATE_TYPE;
  SIGNAL x0, x1, x2, table_in, mem
                                        : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL table_out                      : integer RANGE 0 TO 7;
BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

  PROCESS                               ------> DA in behavioral style
    VARIABLE   p     : integer RANGE 0 TO 63; --Temp. register
    VARIABLE count : integer RANGE 0 TO 3;
  BEGIN                                         -- Counts the shifts
```

注 6：这一例子相应的 Verilog 代码文件 darom.v 可以在附录 A 中找到。

```
WAIT UNTIL clk = '1';
CASE state IS
  WHEN s0 =>                    -- Initialization step
    state <= s1;
    count := 0;
    p := 0;
    x0 <= x_in0;
    x1 <= x_in1;
    x2 <= x_in2;
  WHEN s1 =>                    -- Processing step
    IF count = 3 THEN           -- Is sum of product done ?
      y <= p;                   -- Output of result to y and
      state <= s0;              -- start next sum of product
    ELSE
      p := p / 2 + table_out * 4;
      x0(0) <= x0(1);
      x0(1) <= x0(2);
      x1(0) <= x1(1);
      x1(1) <= x1(2);
      x2(0) <= x2(1);
      x2(1) <= x2(2);
      count := count + 1;
      state <= s1;
    END IF;
  END CASE;
END PROCESS;

rom_1: LPM_ROM
  GENERIC MAP ( LPM_WIDTH => 3,
                LPM_WIDTHAD => 3,
                LPM_OUTDATA => "UNREGISTERED",
                LPM_ADDRESS_CONTROL => "UNREGISTERED",
                LPM_FILE => "darom3.mif")
  PORT MAP ( address => table_in, q => mem);

table_out <= CONV_INTEGER(mem);

END flex;
```

与例 3.6 相比，现在已经得到了一个 LPM_ROM 的组件实例。由于需要在 ROM 的 STD_LOGIC_VECTOR 输出和整数之间进行一次转换，所以需要程序库 ieee 中的 std_logic_unsigned 软件包。ieee 中包括针对无符号 STD_LOGIC_VECTOR 的 CONV_INTEGER 函数。

包含文件 darom3.mif 也是由程序 dagen.exe 生成的，其内容如下：

```
-- This is the DA MIF table for 3coefficients:2, 3, 1
-- automatically generated with dagen.exe – DO NOT EDIT!
WIDTH = 3;
DEPTH = 8;
ADDRESS_RADIX = dec;
DATA_RADIX = dec;
CONTENT BEGIN
        0       :   0;
        1       :   2;
        2       :   3;
        3       :   5;
        4       :   1;
        5       :   3;
        6       :   4;
        7       :   6;
END;
```

这一设计是以 32.25MHz 的 Registered Performance 运行的，使用了 34 个 LC(比 CASE 方案减少了 3 个)和一个 EAB(更加准确地说是 3/8 个 EAB)。

但是 EAB 仅有一个单地址译码器，如果要实现一个 $2^4 \times 8$ 的表，整个 EAB 就被无谓地消耗了，而且也不能用于其他方面。然而，对于较长的滤波器而言，EAB 的使用还是非常有吸引力的，这是因为：

- EAB 的登记吞吐量是恒定的 76MHz。
- 降低了布线工作量。

3. 有符号 DA FIR 滤波器

有符号 DA 滤波器需要一个有符号的累加器。下面的例题给出了前面第 2 章例 2.18 中研究过的 3 个系数的示例的 VHDL 代码。

例 3.9　有符号 DA FIR 滤波器

对于有符号 DA 滤波器而言，还需要一个额外的状态。参照变量 count[7] 来处理符号位：

```
LIBRARY ieee;                       -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

PACKAGE da_package IS       -- User defined components
COMPONENT case3s
    PORT ( table_in     : IN    STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out    : OUT INTEGER RANGE - 2 TO 4);
END COMPONENT;
END da_package;
```

注 7：这一例子相应的 Verilog 代码文件 case3s.v 可以在附录 A 中找到。

```
LIBRARY work;
USE work.da_package.ALL;

LIBRARY ieee;                    -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY dasign IS                              ------> Interface
        PORT (clk   : IN STD_LOGIC;
              x_in0, x_in1, x_in2
                        : IN   STD_LOGIC_VECTOR(3 DOWNTO 0);
              y       : OUT INTEGER RANGE - 64 TO 63);
END dasign;

ARCHITECTURE flex OF dasign IS

  TYPE STATE_TYPE IS (s0, s1);
  SIGNAL state       : STATE_TYPE;
  SIGNAL table_in    : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL x0, x1, x2  : STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL table_out   : integer RANGE - 2 TO 4;

BEGIN

  table_in(0) <= x0(0);
  table_in(1) <= x1(0);
  table_in(2) <= x2(0);

  PROCESS                        ------> DA in behavioral style
    VARIABLE   p : integer RANGE - 64 TO 63;-- Temporary reg.
    VARIABLE count : integer RANGE 0 TO 4; -- Counts the
  BEGIN                                                -- shifts
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN s0 =>              -- Initialization step
        state <= s1;
        count := 0;
        p := 0;
        x0 <= x_in0;
        x1 <= x_in1;
        x2 <= x_in2;
      WHEN s1 =>              -- Processing step
        IF count = 4 THEN -- Is sum of product done?
          y <= p;          -- Output of result to y and
          state <= s0;     -- start next sum of product
```

```
            ELSE
              IF count = 3 THEN          -- Subtract for last
              p := p / 2 - table_out * 8;    -- accumulator step
              ELSE
              p := p / 2 + table_out * 8;    -- Accumulation for
              END IF;                        -- all other steps
                FOR k IN 0 TO 2 LOOP  -- Shift bits
                  x0(k) <= x0(k+1);
                  x1(k) <= x1(k+1);
                  x2(k) <= x2(k+1);
                END LOOP;
              count := count + 1;
              state <= s1;
            END IF;
        END CASE;
    END PROCESS;

    LC_Table0: case3s
      PORT MAP(table_in => table_in, table_out => table_out);

END flex;
```

LC 表(组件 case3s.vhd)是由程序 dagen.exe 生成的。其 VHDL 代码[8]如下：

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY case3s IS
  PORT ( table_in     : IN     STD_LOGIC_VECTOR(2 DOWNTO 0);
         table_out    : OUT    INTEGER RANGE - 2 TO 4);
END case3s;

ARCHITECTURE LCs OF case3s IS
BEGIN

-- This is the DA CASE table for
-- the 3 coefficients:  - 2, 3, 1
-- automatically generated with dagen.exe -- DO NOT EDIT!

  PROCESS (table_in)
  BEGIN
    CASE table_in IS
      WHEN  "000" =>     table_out <=  0;
      WHEN  "001" =>     table_out <=  - 2;
```

注 8：这一例子相应的 Verilog 代码文件 case3s.v 可以在附录 A 中找到。

```
        WHEN   "010" =>      table_out <=   3;
        WHEN   "011" =>      table_out <=   1;
        WHEN   "100" =>      table_out <=   1;
        WHEN   "101" =>      table_out <=  - 1;
        WHEN   "110" =>      table_out <=   4;
        WHEN   "111" =>      table_out <=   2;
        WHEN    OTHERS   =>      table_out <=   0;
    END CASE;
  END PROCESS;
END LCs;
```

图 3-15 给出了输入序列{1, - 3,7}的仿真结果，正如我们所希望的，输出 $y= -4_{10}=1111100_{2C}$。

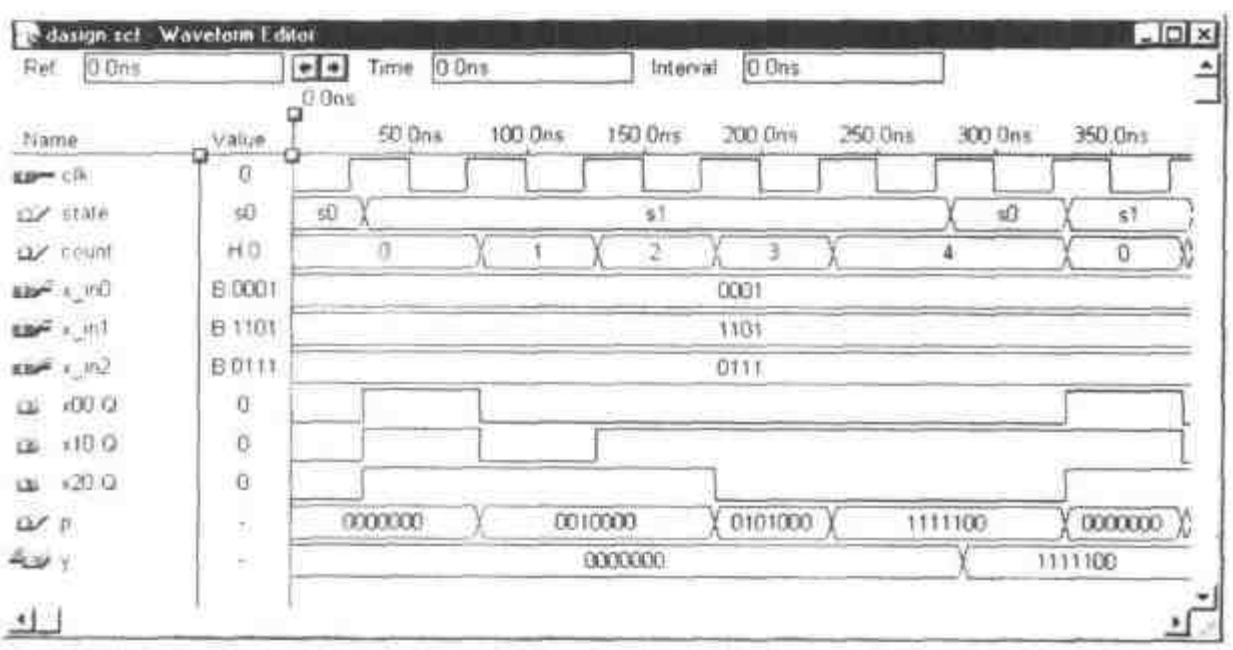


图 3-15　输入{1, - 3,7}的 3 抽头有符号 FIR 滤波器的仿真

为了提高 DA 滤波器的速度，可以采用开式环。输入采用逐次采样(每次一个字)、位并行形式。在这种情况下，对于输入的每一位都需要配置相应单独的表。且表的规模不固定(输入位宽等于滤波器抽头的数量)，表的内容是相同的。如果我们像前面提到的，采用组件定义 LC 表的话，明显的优点就是降低了 VHDL 代码的规模。为了加以说明，下面来研究一下前面的 3 系数，4 位输入的示例的开式环形式。

例 3.10　DA FIR 滤波器的开式环

在典型的 FIR 应用中，输入值是按照字并行形式处理的(请参阅图 3-16)。下面的 VHDL 代码[9]就是根据图 3-16 给出的开环 DA 代码。

```
LIBRARY ieee;                          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

PACKAGE da_package IS                  -- User defined components
  COMPONENT case3s
    PORT ( table_in    : IN   STD_LOGIC_VECTOR(2 DOWNTO 0);
           table_out   : OUT INTEGER RANGE -2 TO 4);
  END COMPONENT;
```

注 9：这一例子相应的 Verilog 代码文件 dapara.v 可以在附录 A 中找到。

```
END da_package;

LIBRARY work;
USE work.da_package.ALL;

LIBRARY ieee;                          -- Using predefined packages
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY dapara IS                                    ------> Interface
        PORT (clk   : IN STD_LOGIC;
              x_in  : IN   STD_LOGIC_VECTOR(3 DOWNTO 0);
              y     : OUT INTEGER RANGE - 46 TO 44);
END dapara;

ARCHITECTURE flex OF dapara IS
  SIGNAL x0, x1, x2, x3 : STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL y0, y1, y2, y3 : integer RANGE - 2 TO 4;
  SIGNAL s0 : integer RANGE - 6 TO 12;
  SIGNAL s1 : integer RANGE - 10 TO 8;
  SIGNAL t0, t1, t2, t3 : integer RANGE - 2 TO 4;

BEGIN

  PROCESS                              ------> DA in behavioral style
  BEGIN
    WAIT UNTIL clk = '1';
    FOR k IN 0 TO 1 LOOP    -- Shift all four bits
      x0(k) <= x0(k+1);
      x1(k) <= x1(k+1);
      x2(k) <= x2(k+1);
      x3(k) <= x3(k+1);
    END LOOP;
    x0(2) <= x_in(0); -- Load x_in in the
    x1(2) <= x_in(1); -- MSBs of register 2
    x2(2) <= x_in(2);
    x3(2) <= x_in(3);
    y <= y0 + 2 * y1 + 4 * y2 - 8 * y3;
-- Pipeline register and adder tree
--   t0 <= y0; t1 <= y1; t2 <= y2; t3 <= y3;
--   s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
--   y <= s0 + 4 * s1;
  END PROCESS;

  LC_Table0: case3s
                PORT MAP(table_in => x0, table_out => y0);
  LC_Table1: case3s
                PORT MAP(table_in => x1, table_out => y1);
  LC_Table2: case3s
```

```
                PORT MAP(table_in => x2, table_out => y2);
    LC_Table3: case3s
                PORT MAP(table_in => x3, table_out => y3);

END flex;
```

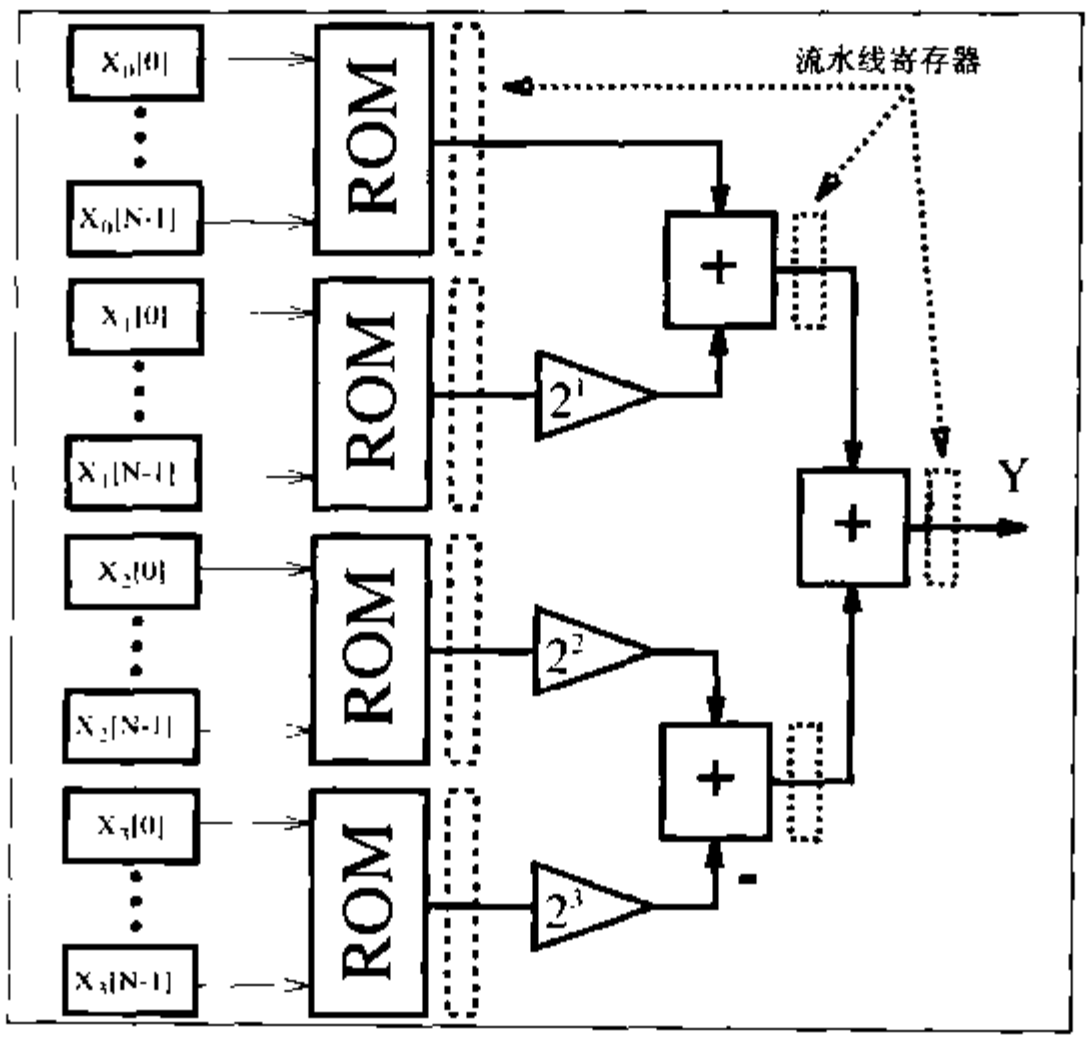


图 3-16　分布式算法 FIR 滤波器的并行实现

这一设计采用了 4 个规模为 $2^3 \times 4$ 的表，并且所有表的内容都与例 3.9 中表的内容相同。图 3-17 给出了输入序列{1, - 3,7}的仿真。由于输入是采用串行(和位并行)的，所以所期望的结果 $-4_{10}=1111100_{2C}$ 是在 400ns 的时间间隔内计算完成的。

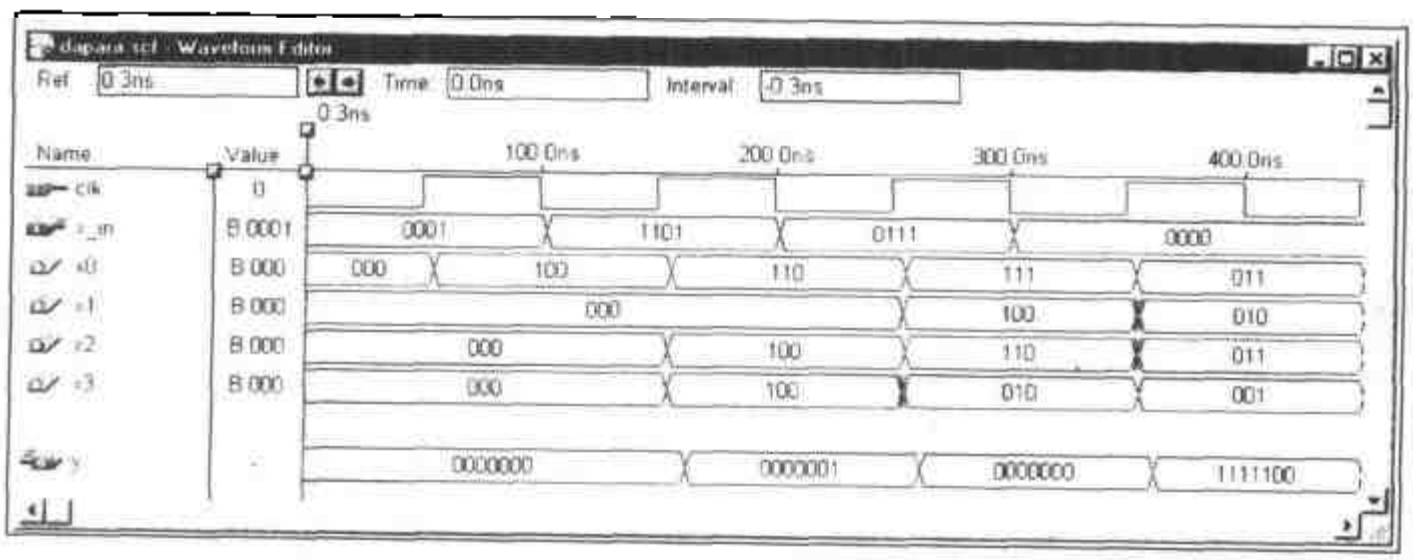


图 3-17　并行分布式算法 FIR 滤波器的仿真结果

上面的设计需要 39 个 LC，运行速度为 42.19MHz。与通用 MAC 设计相比，DA 概念的一个重要优点就是可以很容易地实现流水线处理。我们可以在表的输出和加法器树的输出上增加额外的流水线寄存器，而且不需要增加成本。为了计算 y，也就是代替：

```
y <= y0 + 2 * y1 + 4 * y2 - 8 * y3;
```

在 PROCESS 声明中，我们给流水线形式采用 t0 到 t1 的信号：

```
t0 <= y0; t1 <= y1; t2 <= y2; t3 <= y3;
s0 <= t0 + 2 * t1; s1 <= t2 - 2 * t3;
y <= s0 + 4 * s1;
```

由于没有使用额外的 LC 表寄存器和加法器，所以设计规模没有增加。但是 Registered Performance 却从 42.19MHz 增加到 106.38MHz!

3.5　练习

3.1：某一滤波器规格如下：采样频率 2KHz；通频带 0-0.4KHz；抑止频带 0.5-1KHz；通频带纹波 3dB；抑止频带纹波 48dB；利用 MATLAB 软件和“交互式低通滤波器设计”演示示例进行这一滤波器的设计。

(a1) 用 Kaiser 窗函数设计直接滤波器。

(a2) 确定滤波器长度和通频带内的绝对纹波。

(b1) 设计一个等同纹波滤波器。

(b2) 确定滤波器长度和通频带内的绝对纹波。

练习使用 MaxPlusII

3.2：(a) 计算长度为 11 的半波带滤波器 F5 的 RAG，这一滤波器有非零系数 $f[0]=256$, $f[\pm1]=150$, $f[\pm3]=-25$, $f[\pm5]=3$。

(b) 如果输入位宽是 8，则滤波器的最小输出位宽是多少？

(c1) 编写并(用 MaxPlusII 编译器)编译滤波器的 VHDL 代码。

(c2) 仿真滤波器的脉冲和阶跃响应。

(c3) 用状态机方法和已经实现的表 LPM_ROM 编写分布式算法滤波器的 VHDL 代码。

3.3：(a) 计算长度为 11 的半波带滤波器 F7 的 RAG，这一滤波器有非零系数 $f[0]=521$, $f[\pm1]=302$, $f[\pm3]=-53$, $f[\pm5]=7$。

(b) 如果输入位宽是 8，则滤波器的最小输出位宽是多少？

(c1) 编写并(用 MaxPlusII 编译器)编译滤波器的 VHDL 代码。

(c2) 仿真滤波器的脉冲和阶跃响应。

3.4：Hartley[67]已经引入了一种通过采用公共子表达式交叉系数实现常系数滤波器的概念，例如：滤波器

$$y[n]=\sum_{k=0}^{L-1}a[k]x[n-k] \tag{3.19}$$

其中 3 个系数 $a[k]=\{480,-302,31\}$。3 个系数的 CSD 码如下：

	512	256	128	64	32	16	8	4	2	1
480:	1	0	0	0	-1	0	0	0	0	0
-302:	0	-1	0	-1	0	1	0	0	1	0
31:	0	0	0	0	1	0	0	0	0	-1

从表中可以注意到 $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ 结构出现了 4 次。如果构造一个临时变量 $h[n]=2x[n]-x[n-1]$ 就可以用

$$y[n] = 256h[n] - 32h[n] - 16h[n-1] + h[n-1] \tag{3.20}$$

计算滤波器的输出。

(a) 代入 $h[n]=2x[n]-x[n-1]$验证(3.20)。

(b) 得到(3.19)的直接 CSD 实现和子表达式共享的实现分别需要多少个加法器?

(c1) 用 MaxPlusII 实现 8 位输入的子表达式共享滤波器。

(c2) 仿真滤波器的脉冲响应。

(c3) 确定 LC 的使用量和 Registered Performance。

3.5：用练习 3.4 的子表达式方法实现一个 4 抽头滤波器，系数 $a[k]=\{-1046,-1109,-894,2072\}$。

(a) 找出最频繁出现结构的 CSD 代码和子表达式。

(b) 分别用 2 或 -2 代入子表达式。用一次或多次子表达式共享化简集合。

(c) 确定临时方程并代回(3.19)检验。

(d1) 用 MaxPlusII 实现 8 位输入的子表达式共享滤波器。

(d2) 仿真滤波器的脉冲响应。

(d3) 确定 LC 的使用量和 Registered Performance。

3.6：(a) 利用程序 dagen.exe 和多级 CASE 声明为系数{20,24,21,100,13,11,19,7}编译一个 DA 表。针对最大速度合成设计，并确定规模和 Registered Performance。

(b) 利用分块技术，使用两个分别为{20,24,21,100}和{13,11,19,7}的集合和一个额外的加法器实现同样的表。针对最大速度合成设计，并确定规模和 Registered Performance。

(c) 比较(a)和(b)的设计。

第4章　无限脉冲响应(IIR)数字滤波器

概述

在第 3 章中我们介绍了 FIR 滤波器。在选择应用中，FIR 具有吸引力(+)或无吸引力(－)的最为重要的属性包括：

- +FIR 线性相位的性能很容易实现。
- +多频带滤波器是可行的。
- +Kaiser 窗函数方法允许自由迭代设计。
- +FIR 具有结构简单的抽取器和插入器(请参阅第 5 章)。
- +非递归滤波器总是稳定的，并且没有极限环。
- +可以很容易得到高速流水线式的设计。
- +典型的 FIR 都具有较低的系数和算法四舍五入误差预算以及定义明确的量化噪声。
- －由于极点/零点消除的不完整，递归滤波器可能会不稳定。
- －复杂的 Parks-McClellan 算法必须是针对极小化极大滤波器设计才是可行的。
- －高滤波长度需要很多的工作量。

在给定滤波器阶数时，与 FIR 滤波器相比，无限脉冲响应(Infinite Impulse Response, IIR)滤波器在达到某种性能属性方面经常可以更有效率。这是因为 IIR 滤波器引入了反馈，且适合于系统传递函数的零点和极点的实现，相比之下，FIR 滤波器则是一种全零滤波器。在本章中我们将研究 IIR 滤波器设计的基本原理。IIR 滤波器设计的传统方法包括将定义反馈类型的模拟滤波器转换到数字域。这是一个合理的途径，主要是因为设计模拟滤波器的技巧是相当先进的，也就是说有许多现成的标准表可供选用[68]。在本章中我们首先回顾一下 4 类最为重要的模拟原型滤波器，它们分别是 Butterworth、Chebyshev I 和 II 以及椭圆滤波器。

我们将会看到 IIR 滤波器可以克服 FIR 滤波器的很多缺点，当然也还有一些不尽如人意的性质。IIR 滤波器常规的具有吸引力(+)和无吸引力(－)的属性包括：

- +采用模拟原型滤波器的标准设计是很容易理解的。
- +高级选择性滤波器可以用低阶设计实现，并且可以以很高的速度运行。
- +设计时可以采用表和袖珍型计算器。
- +对于相同的公差设计方案，与 FIR 滤波器相比，IIR 滤波器较短。
- +可以采用闭环设计算法。
- －非线性相位响应是典型的，也就是说要得到线性相位响应是非常困难的(采用全通滤波器做相位补偿会使复杂性成倍增加)。
- －对于整数实现而言可能会出现极限环。
- －多频带设计非常困难，只能够设计低通、高通和带通滤波器。
- －反馈会引入不稳定性(大多数时候，极点到单位圆的镜像可以用来生成同样的幅度响应，

滤波器是稳定的)。

● - 要想得到高速流水线设计就更加困难了。

为了说明采用 IIR 滤波器的可能益处，下面我们将讨论一个一阶 IIR 滤波器的例子。

例 4.1　有耗积分器 I

滤波器的一个基本功能就是使得有干扰的信号平滑。假定信号 $x[n]$是以含有宽频带零平均值随机噪声的形式接收到的。从数学的角度讲，可以采用积分器来消除噪声的影响。如果输入信号的平均值能够保持的时间间隔是有限长，就可以采用有耗积分器处理含有额外噪声的信号。图 4-1 显示了一个简单的一阶有耗积分器，它满足离散时间差分方程：

$$y[n+1]=\frac{3}{4}y[n]+x[n] \tag{4.1}$$

图 4-1　用作有耗积分器的一阶 IIR 滤波器

正如我们可以从图 4-2(a)中的脉冲响应可以看到的，要实现与一阶有耗积分器同样的功能，需要 15 抽头的 FIR 滤波器才能够做到。有耗积分器的阶跃响应如图 4-2(b)所示。

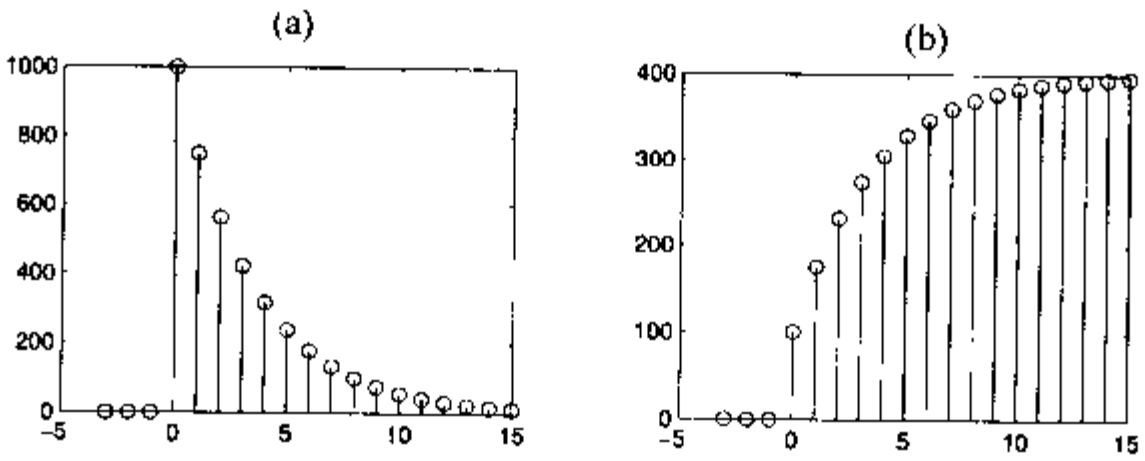


图 4-2　$a=3/4$ 的有耗积分器的仿真 (a) $1000\delta[n]$的脉冲响应 (b) $100\sigma[n]$的阶跃响应

下面的 VHDL 代码[1]给出了 IIR 滤波器的一种可能实现。

```
PACKAGE n_bit_int IS                    -- User defined type
  SUBTYPE BITS15 IS INTEGER RANGE - 2**14 TO 2**14 - 1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

注 1：这一例子相应的 Verilog 代码文件 iir.v 可以在附录 A 中查到。

```
ENTITY iir IS
  PORT (x_in   : IN  BITS15;        -- Input
        y_out  : OUT BITS15;        -- Result
        clk    : IN  STD_LOGIC);
END iir;

ARCHITECTURE flex OF iir IS

  SIGNAL x, y : BITS15;

BEGIN

  PROCESS       -- Use FF for input and recursive part
  BEGIN
  WAIT UNTIL clk = '1';
    x  <= x_in;
    y  <= x + y / 4 + y / 2;
  end process;

  y_out <= y;                -- Connect y to output pins

END flex;
```

在 PROCESS 模块内采用 WAIT 声明实现寄存器，而乘法和加法的实现所使用的是 CSD 码。设计采用了 31 个逻辑单元，运行速度是 55.86MHz。如果与选项 Optimize Speed=10 合成，则滤波器对于幅值为 1000 的脉冲的响应如图 4-3 所示，图 4-2(a) 给出了相应的仿真结果。

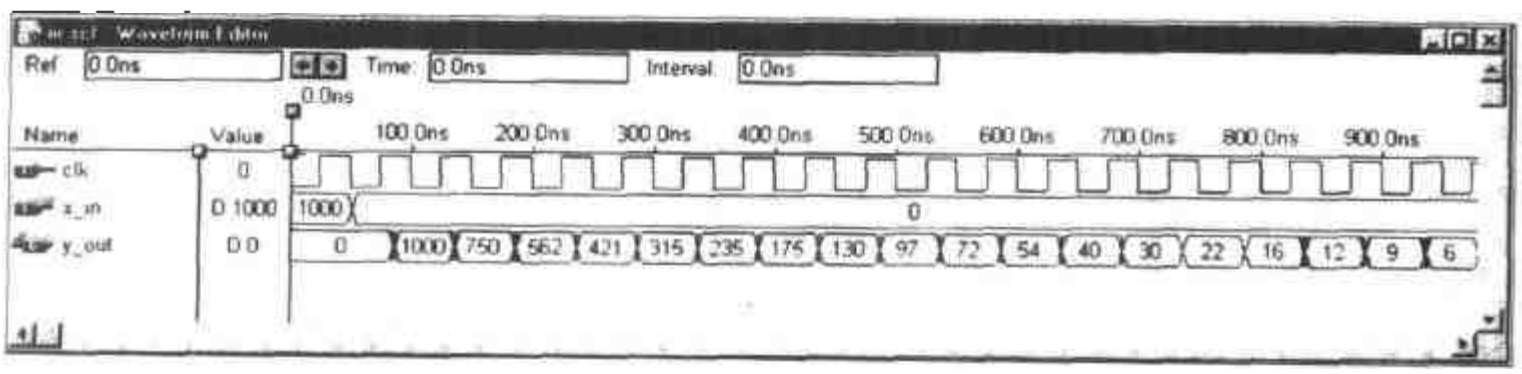


图 4-3　有耗积分器脉冲响应的 MaxPlusII 仿真

另一种可选设计方案采用了“标准逻辑矢量”数据类型和 LPM_ADD_SUB 宏函数，我们将在练习 4.4 中讨论这种设计。第二种方法生成的 VHDL 代码更长，但是其优点是可以在位的级别上对符号的扩展和乘法器进行直接控制。

4.1　IIR 理论

顾名思义，非递归滤波器，也就是 FIR 滤波器，没有引入反馈。这种滤波器的脉冲响应是有限的。相反，递归滤波器，也就是 IIR 滤波器，具有反馈，一般认为具有无限的脉冲响应。图4-4(a)分别给出了具有递归部分和没有递归部分的滤波器。如果将这些递归和非递归部分组

合起来就生成了典型的如图 4-4(b)所示的滤波器。图 4-4 中滤波器的传递函数可以写成：

$$F(z)=\frac{\sum_{l=0}^{L-1}a[l]z^{-l}}{1-\sum_{l=1}^{L-1}b[l]z^{-l}} \tag{4.2}$$

系统输出的差分方程如下：

$$y[n]=\sum_{l=0}^{L-1}a[l]x[n-l]+\sum_{l=1}^{L-1}b[l]y[n-k] \tag{4.3}$$

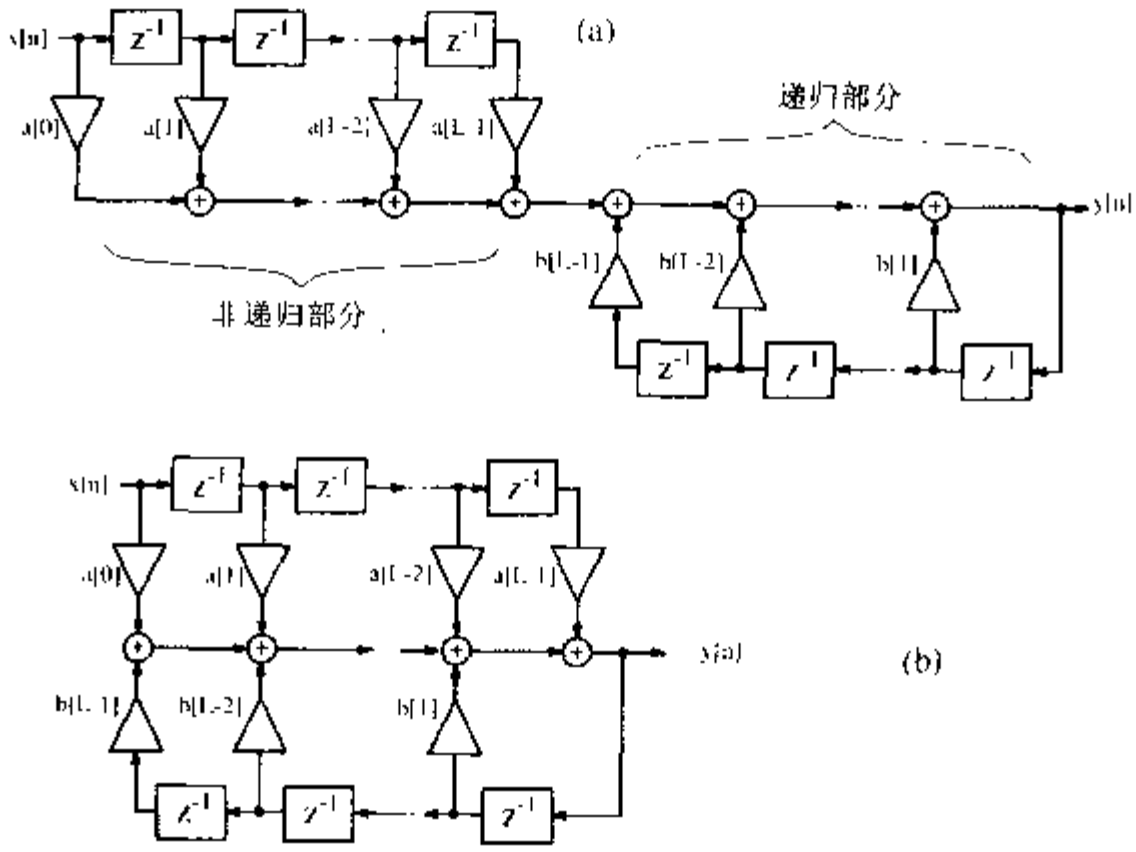


图 4-4　带有反馈的滤波器

与 FIR 滤波器的差分方程(3.2)进行比较就会发现，递归系统的差分方程不仅仅依赖于输入序列 $x[n]$的前 L 个值，而且与 $y[n]$的前 $L-1$ 个值有关。

如果计算 $F(z)$的极点和零点，就可以看到非递归部分，也就是 $F(z)$的分子具有零点 p_{0l}，而 $F(z)$的分母具有极点 $p_{\infty l}$。

对于传递函数而言，极点/零点图可以用来查找滤波器最重要的属性。在 z 域的传递函数中如果用 $z=e^{j\omega T}$进行替换，就可以通过图像的方法构造傅立叶传递函数：

$$F(\omega)=|F(\omega)|e^{\theta(\omega)}=\frac{\prod_{l=0}^{L-2}p_{0l}-e^{j\omega T}}{\prod_{l=0}^{L-2}p_{\infty l}-e^{j\omega T}}=\frac{e\sum_l\alpha_l\prod_{l=0}^{L-2}v_l}{e\sum_l\beta_l\prod_{l=0}^{L-2}u_l} \tag{4.4}$$

如图 4-5 所示，给出了具体的幅值(也就是增益)和相位值。对应于具体频率 ω_0，增益是零点向量 v_l 与极点向量 u_l 的商。这些向量分别起始于各自的零点或极点，终止于增益的频率点 $e^{j\omega_0 T}$。图 4-5 中示例的相位增益就是 $\theta(\omega_0)=\alpha_0+\alpha_1-\beta_0$。

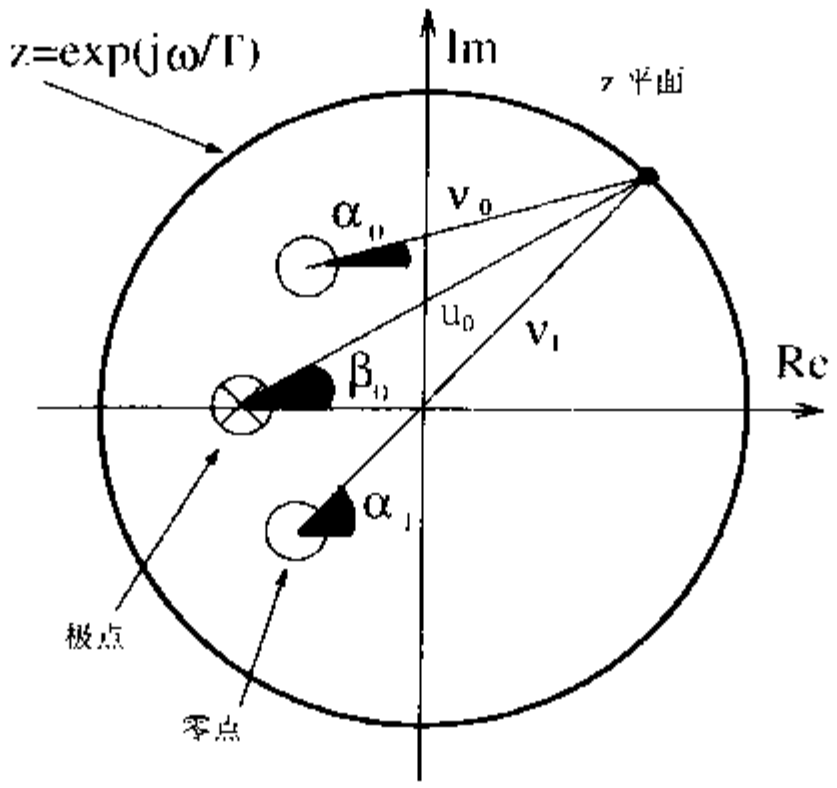


图 4-5　用极点/零点分布图计算传递函数。幅值增益 = $\nu_0\nu_1/u_0$，相位增益 $=\alpha_0+\alpha_1-\beta_0$

利用傅立叶域内的传递函数与极点/零点图之间的联系，我们可以推导出下列几个属性：

(1) 单位圆上的零点 $p_0=e^{j\omega_0 T}$ (无与之抵消的极点)生成傅立叶域内传递函数在频率 ω_0 的一个零点。

(2) 单位圆上的极点 $p_\infty=e^{j\omega_0 T}$ (无与之抵消的零点)生成傅立叶域内传递函数在频率 ω_0 的无穷增益。

(3) 独立于输入信号的稳定滤波器的所有极点都位于单位圆内部。

(4) 实际的滤波器在单位圆上具有单个极点和零点，而复数极点和零点总是成对出现，也就是说如果 $\alpha_0+j\alpha_1$ 是一个极点或者零点，则 $\alpha_0-j\alpha_1$ 也必定是极点或者零点。

(5) 线性相位(也就是常数组延迟)滤波器所有的极点和零点都是关于单位圆对称的，或者它们位于 $z=0$。

如果将 3 与 5 组合起来，就会发现对于稳定的线性相位系统而言，所有零点都关于单位圆对称，并且只允许极点位于 $z=0$。

滤波因此，IIR 滤波器(极点 $z\neq 0$)只能是近似的线性相位。为了实现近似，需要采用一种来自模拟滤波设计的著名原则：全通滤波器具有单位增益，引入非零相位增益，用来实现增益周期范围(也就是通频带)内的线性化。

4.2　IIR 系数的计算

在经典的 IIR 滤波器设计中，数字滤波器的设计是非常接近理想滤波器的。理想数字滤波器模型规范在数学上转换成一组模拟滤波器模型的规范，所采用的方法为下面公式给出的双边 z 变换：

$$s=\frac{z-1}{z+1} \tag{4.5}$$

经典模拟 Butterworth、Chebyshev 或者是椭圆模型都可以由这些规范合成。然后利用双边线性 z 变换映射到数字 IIR 滤波器中。

模拟 Butterworth 滤波器的幅值平方的频率响应如下：

$$|F(\omega)|^2=\frac{1}{1+\left(\frac{\omega}{\omega_s}\right)^{2N}} \tag{4.6}$$

$|F(\omega)|^2$ 的极点沿着圆周分布，分别相隔 π/N 弧度。再具体地说就是传递函数在 $\omega=0$ 处 N 次可微。这一结论说明传递函数在 0 Hz 附近是局部光滑的。图 4-6 (上图)给出了一个 Butterworth 滤波器模型的例子。注意，这一设计的公差设计方案与 Kaiser 窗函数和图 3-7 给出的 equiripple 设计是一致的。

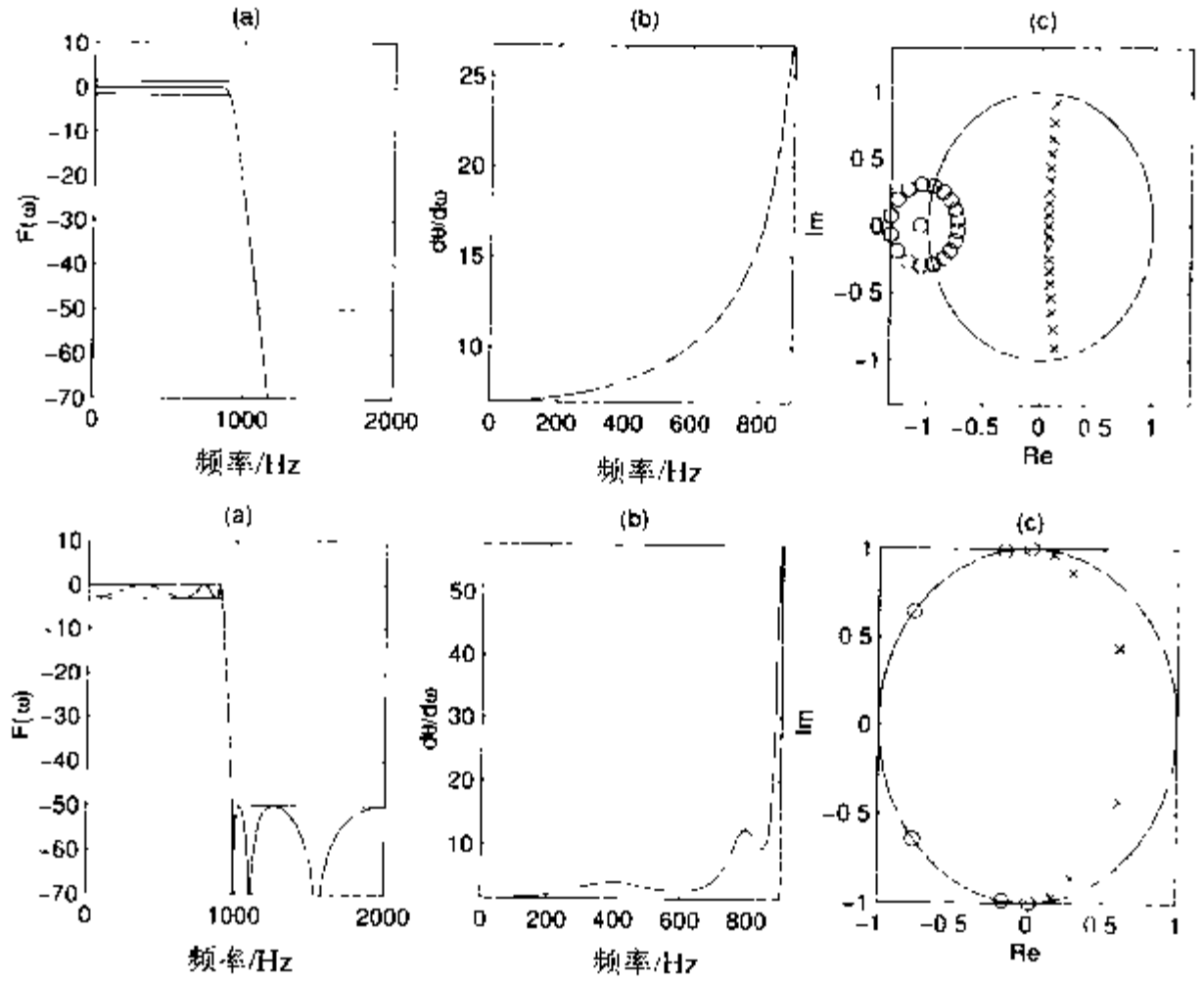


图 4-6　利用 MATLAB 工具箱的滤波器设计。(上) Butterworth 滤波器和(下)椭圆滤波器。
(a) 传递函数 (b) 通频带组延迟 (c) 极点/零点分布图(×＝极点，o＝零点)

I 型或者 II 型模拟 Chebyshev 滤波器是按照 Chebyshev 多项式 $V_M(\omega)=\cos(N\cos(\omega))$定义的，这就要求滤波器的极点必须驻留在一个椭圆上。I 型滤波器的幅值平方频率响应表述如下：

$$|F(\omega)|^2=\frac{1}{1+\varepsilon^2 V_N^2\left(\frac{\omega}{\omega_c}\right)} \tag{4.7}$$

典型的 I 型滤波器示例的幅频和脉冲响应如图 4-7(上图)所示。注意观察通带的纹波和光滑的抑止频带。

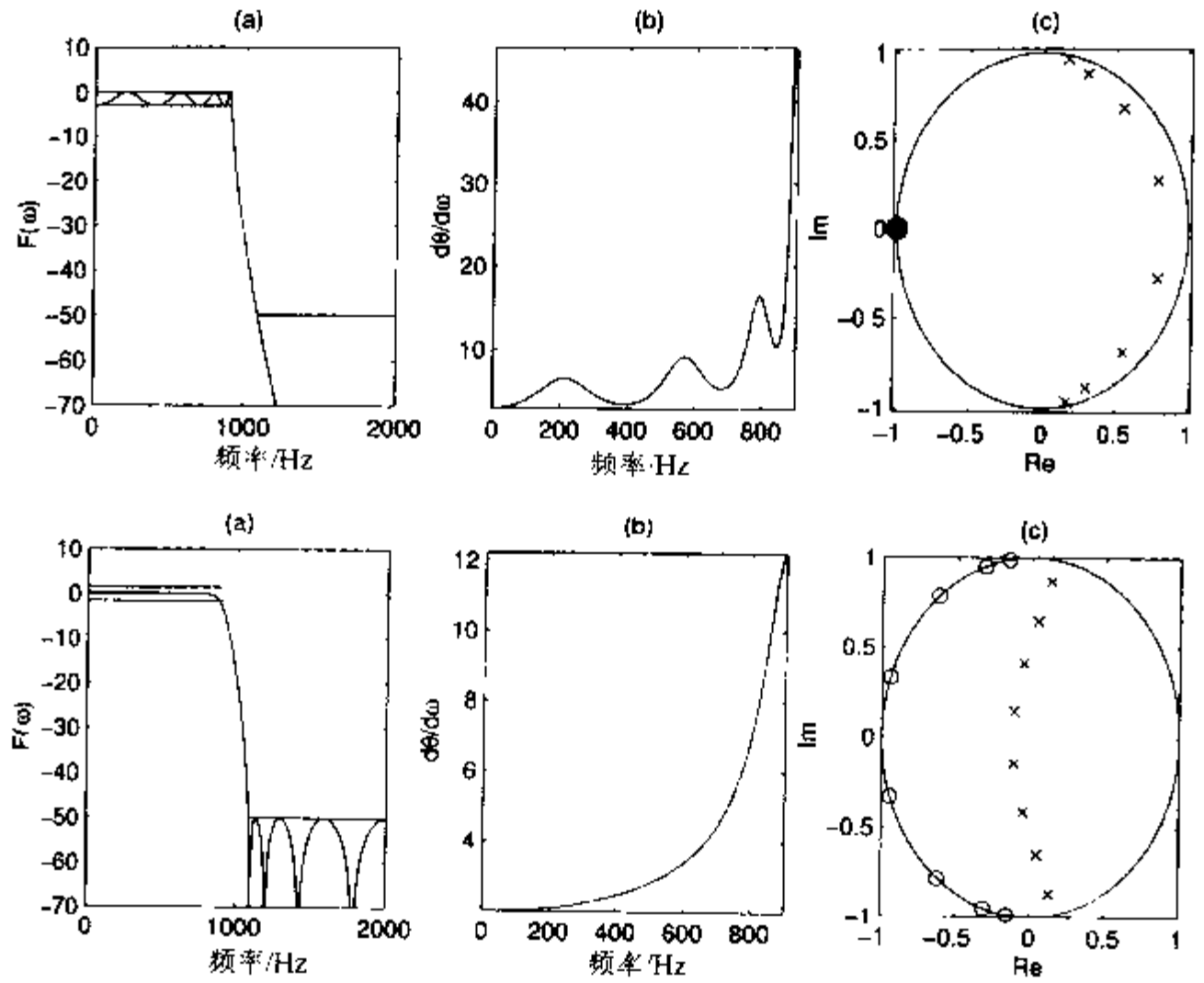


图 4-7　利用 MATLAB 工具箱的 Chebyshev 滤波器设计。(上) Chebyshev I 型滤波器和(下) Chebyshev II 型滤波器。(a) 传递函数 (b) 通频带组延迟 (c) 极点/零点分布图(×＝极点，o＝零点)

II 型滤波器幅值平方频率响应表述如下：

$$|F(\omega)|^2=\frac{1}{1+\left(\varepsilon^2 V_N^2\left(\frac{\omega}{\omega_s}\right)\right)^{-1}} \tag{4.8}$$

典型 II 型的一个示例的幅值频率和脉冲响应如图 4-7(下图)所示。注意：在该情况下会产生一个光滑的通频带，而抑止频带则呈现纹波特性。

模拟椭圆原型滤波器是根据 Jacobian 椭圆方程 $U_N(\omega)$的解定义的。幅值平方频率响应的表述如下：

$$|F(\omega)|^2=\frac{1}{1+\varepsilon^2 U_N^2\left(\frac{\omega}{\omega_s}\right)^{-1}} \tag{4.9}$$

典型椭圆滤波器的幅值平方和脉冲响应如图 4-6 的下图所示。注意观察椭圆滤波器在通频带和抑止频带表现出的纹波。

如果我们将 4 种不同的 IIR 滤波器的实现进行比较就不难发现：在图 3-8 给出的相同公差设计方案下，Butterworth 滤波器有 19 阶，Chebyshev 滤波器有 8 阶，而椭圆型滤波器设计有 6 阶。将图 4-6 和 4-7 进行比较，就会发现，随着滤波器阶数的减少，纹波就会增加，并且组延迟的非线性就变得非常严重。大多数情况下，良好的折衷就是 Chebyshev II 型滤波器，它具有中等阶数、平坦的通频带和允许的公差组延迟。

主要 IIR 滤波器设计属性的总结

在前面部分介绍了经典的 IIR 滤波器类型。每种模式都为设计者提供了折衷的选择。经典 IIR 型滤波器的属性总结如下：

- Butterworth：最平的通频带、平坦的抑止频带、宽的过渡频带
- Chebyshev I：等纹波的通频带、平坦的抑止频带、适中的过渡频带
- Chebyshev II：平坦的通频带、等纹波的抑止频带、适中的过渡频带
- 椭圆型：等纹波的通频带、等纹波的抑止频带、狭窄的过渡频带

在给定滤波器要求的情况下，一般要把握下列观察特性：

- 滤波器阶数
  - 一最低：椭圆型
  - 一中间：Chebyshev I 或 II 型
  - 一最高：Butterworth
- 通频带特性
  - 一等纹波：椭圆型、Chebyshev I 型
  - 一平坦度：Butterworth 和 Chebyshev II 型
- 抑止频带特性
  - 一等纹波：椭圆型、Chebyshev II 型
  - 一平坦度：Butterworth 和 Chebyshev I 型
- 过渡频带特性
  - 一最窄：椭圆型
  - 一中间：Chebyshev I 和 II 型
  - 一最宽：Butterworth

4.3 IIR 滤波器的实现

获得 IIR 滤波器的传递函数一般都是依靠直觉的经验，特别是使用像 MATLAB 这样的设计软件。IIR 滤波器可以根据其前后关系设计成多种体系结构，其中最重要的结构总结如下：

- 直接 I 形式(参阅图 4-8)
- 直接 II 形式(参阅图 4-9)
- 一阶或者二阶系统的级联(参阅图 4-10(a))
- 一阶或者二阶系统的并联实现(参阅图 4-10(b))
- 简单级联或者并联设计中典型二阶部分的双四边形实现(参阅图 4-11)
- 正交形式[69]，也就是一阶或者二阶静态变量系统的级联(参阅图 4-10(a))
- 并行正交的，也就是并联的一阶或者二阶静态变量系统的级联(参阅图 4-10(b))
- 连分数结构
- 网格滤波器(在 Gray-Markel 之后，参阅图 4-12)
- 波动的数字实现(在 Fettweis 之后[70])
- 一般状态空间滤波器

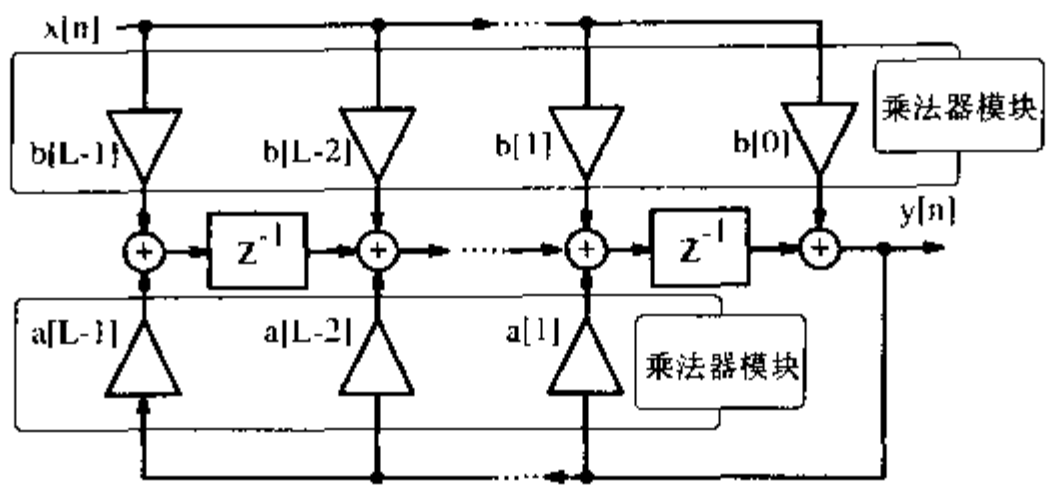


图 4-8　采用乘法器模块的直接 I 型 IIR 滤波器

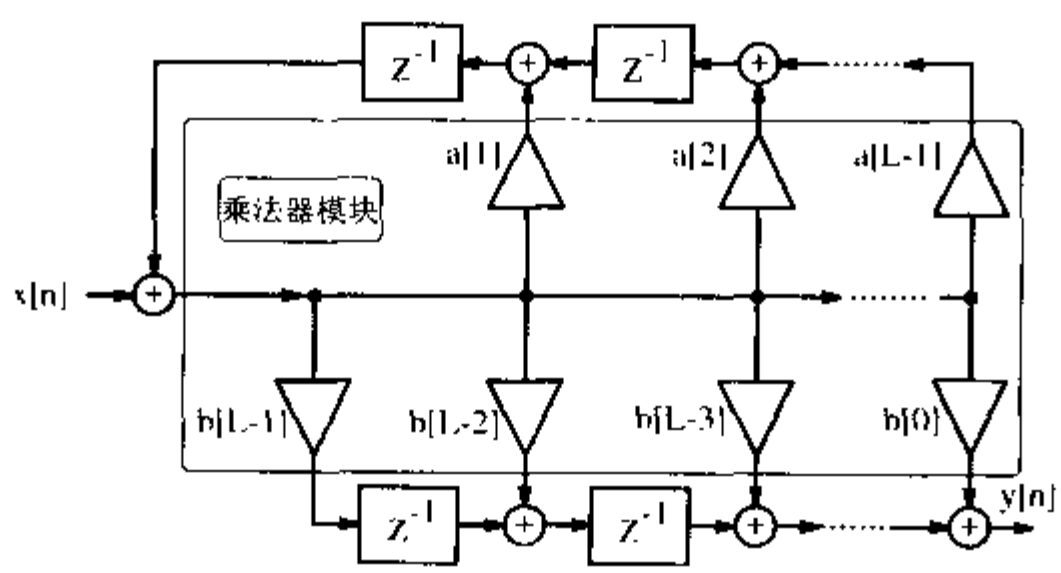


图 4-9　采用乘法器模块的直接 II 型 IIR 滤波器

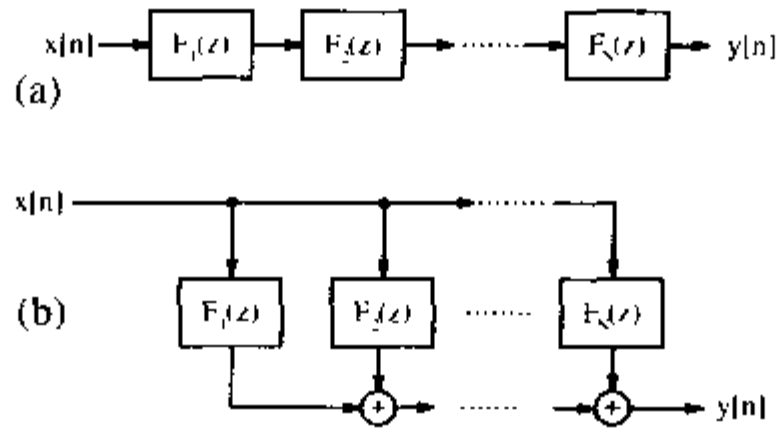


图 4-10　(a) $F(z)=\prod_{k=1}^{N}F_k(z)$ 的串行实现　(b) $F(z)=\sum_{k=1}^{N}F_k(z)$ 的并行实现

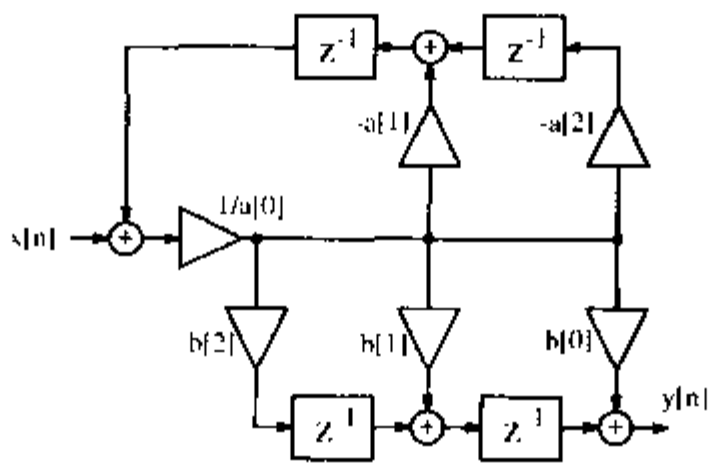


图 4-11　传递函数 $F(z)=(b[0]+b[1]z^{-1}+b[2]z^{-2})/(a[0]+a[1]z^{-1}+a[2]z^{-2})$ 的可能二阶部分的双四边形

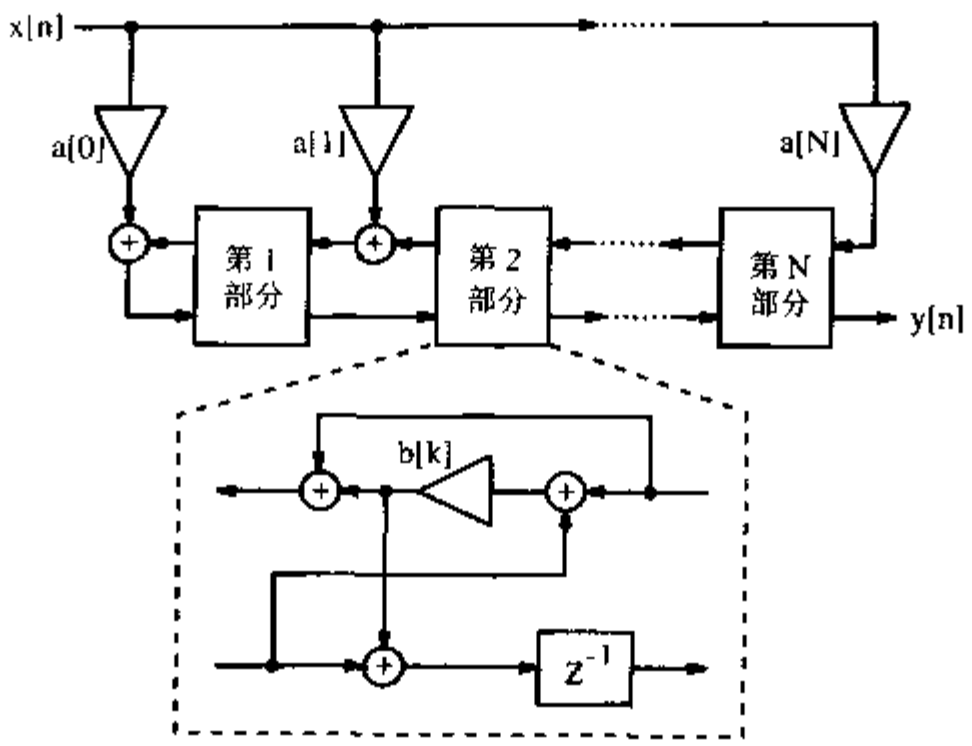


图 4-12　网格滤波器

每种结构都有其应用场合。下面给出了一些常规的选择规律：

- 速度
  - 一高速：直接 I 和 II
  - 一低速：波动
- 定点算法四舍五入误差的灵敏度
  - 一高：直接 I 和 II
  - 一低：正交、网格
- 定点系数四舍五入误差的灵敏度
  - 一高：直接 I 和 II
  - 一低：并行、波动
- 特殊属性
  - 一正交加权输出：网格
  - 一最佳二阶部分：正交
  - 一随机 IIR 技术要求：静态变量

借助像 MATLAB 之类的软件工具，可以很容易地将系数从一种结构转换为另一种结构，下面将举例说明。

例 4.2　Butterworth 二阶系统

假定我们要设计一种用二阶系统实现的 Butterworth 滤波器(阶数=10，通频带=0.3Fs)，可以利用下面的 MATLAB 代码生成系数：

```
N=10; Fp=0.3;
[B, A]=butter(N, Fp)
[sos, gain]=tf2sos(B,A)
```

也就是首先使用函数 butter()计算 Butterworth 滤波器的系数，然后利用“二阶部分传递函数” tf2sos 转换这一滤波器系数并计算双四边形的系数。

利用 MATLAB 可以得到下列结果：

B = 0.0001，0.0012，0.0048，0.0112，0.0168，

　　0.0168，0.0112，0.0048，0.0012，0.0001

A = 1.0000，－3.5863，6.5587，－7.5520，5.9363

　　－3.2606，1.2421，－.03146，0.0479，－0.0033

最后我们可以得到的二阶部分系数如下：

b[0,i]	b[1,i]	b[2,i]	a[0,i]	a[1,i]	a[2,i]
1.0000	2.0079	0.9997	1.0000	－0.4709	0.0610
1.0000	2.0762	1.0784	1.0000	－0.4936	0.1121
1.0000	2.0274	1.0296	1.0000	－0.5434	0.2243
1.0000	1.9695	0.9717	1.0000	－0.6310	0.4216
1.0000	1.9250	0.9271	1.0000	－0.7786	0.7541

图 4-13 给出了滤波器的传递函数、组延迟和极点/零点图。注意，图中所有的零点都位于 z_{0i}=－1 处，这一点从二阶系统分子的系数可以看出来。同时还要注意 b[1, i]=2 和 b[0, i]=b[2, i]=1 中的四舍五入误差。

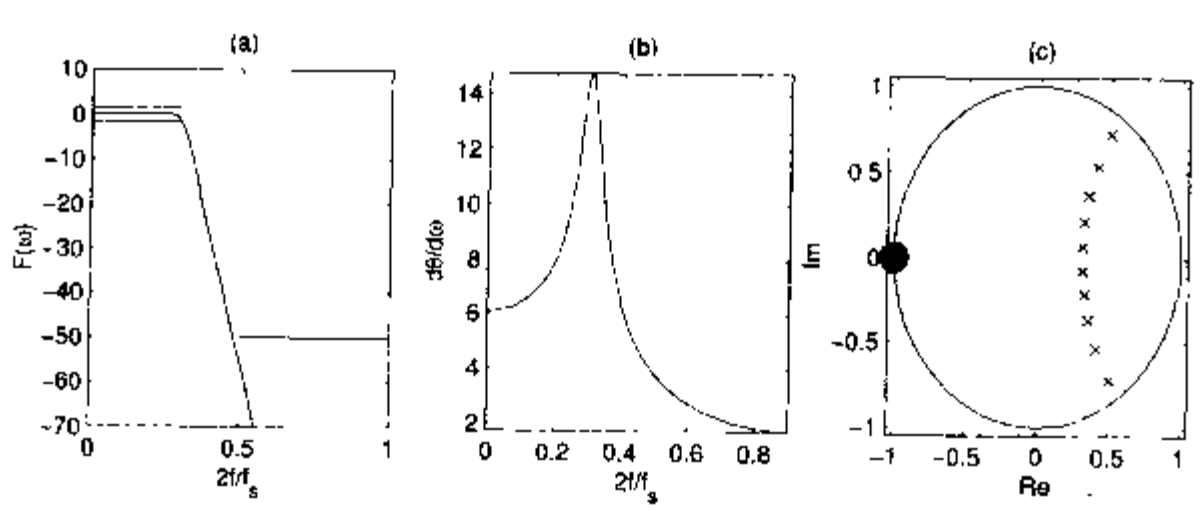


图 4-13　10 阶 Butterworth 滤波器表现的 (a) 幅值 (b) 相位 (c) 组延迟响应

4.3.1　有限字长效应

Crochiere 和 Oppenheim[71] 已经指出：数字滤波器需要的系数字长与系数灵敏度密切相关。也就是说同样的 IIR 滤波器的实现需要宽泛的字长范围。为了说明这一问题的一些动态特性，我们来研究一下 Crochiere 和 Oppenheim 分析的 8 阶椭圆滤波器[71]。这一 8 阶传递函数的实现需要波动、级联、并联、网格、直接 I 和 II 以及连分数结构。表 4-1 的第 2 列中给出了满足具体最大通频带误差准则所预计的系数字长的保守估计。由此可以看出直接形式需要比波动或者并行结构更多的字长。进而得出下面的结论：就位宽(W)乘法器输出(M)而论，波动结构给出了最佳的复杂性(MW)，这一点我们从表 4-1 的第 6 列也可以看出来。

表 4-1 Crochiere 和 Oppenheim 给出的 8 阶椭圆滤波器的数据(根据成本 $M \cdot W$ 分类)

类 型	字 长 W	乘 积 M	加 法	延 迟	成本 $M \cdot W$
波动	11.35	12	31	10	136
级联	11.33	13	16	8	147
并联	10.12	18	16	8	182
网格	13.97	17	32	8	238
直接 I 型	20.86	16	16	16	334
直接 II 型	20.86	16	16	8	334
连分数结构	22.61	18	16	8	408

参照 FIR 滤波器(请参阅第 3 章)，为了简化包含若干乘法器的模块的设计[72,73]，需要引入简化加法器图(reduced adder graph, RAG)技术。Dempster 和 Macleod 已经从 RAG 乘法器实现策略的角度计算了 8 阶椭圆滤波器。表 4-2 给出了比较结果。第 2 列是乘法器模块的规模，直接 II 型结构需要两个乘法器模块，规模分别是 9 和 7。波动结构没有两个具有相同输入的系数，所以也就不需要设计乘法器模块，不过需要实现 11 个单独的乘法器。第 3 列是实现乘法器模块需要的、规范的有符号位设计的加法器/减法器 B 的数量。第 4 列给出单个最佳的乘法器加法器图(multiplier adder graph, MAG)的相同结果。第 5 列是简化加法器图。第 6 列是 RAG 设计的所有加法器/字宽乘积。表 4-2 给出了级联形式和并联形式与波动数字滤波器相比，可比较的或者是更好一些的结果，因为采用 RAG 算法时，乘法器模块的规模是一个基本准则。没有为 FPGA 设计特别考虑延迟，这是因为所有的逻辑单元都有一个相关的触发器。

表 4-2 采用 CSD、MAG 和 RAG 策略实现 8 阶椭圆滤波器的数据

类 型	模 块 规 模	CSD	MAG	RAG	
		B	B	B	$W(B+A)$
级联	4×3,2×1	26	26	24	453
并行	11×9,4×2,1×1	31	30	29	455
波动	11×1	58	63	22	602
网格	1×9,8×1	33	31	29	852
直接 I	1×16	103	83	36	1085
直接 II	1×9,1×7	103	83	41	1189
连分数	18×1	118	117	88	2351

4.3.2 滤波器增益系数的最优化

一般情况下，我们是从浮点数滤波器系数来导出 IIR 整数系数的，首先规格化最大系数，接下来乘以需要的增益因数，例如：位宽 $2^{\text{round}(W)}$。但是，大多数情况下，在 $2^{\lfloor W \rfloor} \ldots 2^{\lceil W \rceil}$ 范围内选择增益系数更为有效。而且在传递函数中基本没有变化，这是因为在乘以增益系数之后，滤波器的系数就已经被四舍五入舍掉了。例如：我们在 $2^{\lfloor W \rfloor} \ldots 2^{\lceil W \rceil}$ 的范围中寻找前面提到的

Crochiere 和 Oppenheim 设计示例中的级联滤波器(采用表 4-1 中给出的增益：$2^{\lfloor 11.33 \rfloor+1}=1024$)的系数，表 4-3 给出了相应的数据。

表 4-3　最小级联滤波器复杂性的增益因子变量

	CSD	MAG	RAG
最优增益	1122	1121	1121
#最优增益的加法器	23	21	18
#增益=1024 的加法器	26	26	24
提高	12%	19%	25%

通过表 4-3 中的比较，我们可以看到加法器数量的真正改进需要乘法器的实现。尽管在本例中 MAG 和 RAG 的最佳增益系数相同，但是它们是可以不同的。

4.4　快速 IIR 滤波器

在第 3 章中 FIR 滤波器的 Registered Performance 是通过采用流水线技术(参阅图 3-6)来提高的。就 FIR 滤波器而言，实现流水线技术基本上不需要额外成本。但是流水线 IIR 滤波器就比较复杂了，而且一般还需要相应的成本。

提高 IIR 滤波器吞吐量的分析报告如下：

- 在时域中预先考虑交叉[75]
- 群集地预先考虑极点/零点的设置[76, 77]
- 分散地预先考虑极点/零点的设置[75, 78]
- IIR 抽取滤波器的设计[79]
- 并行处理[80]
- RNS 的实现[35] [44]

前 5 种方法都是依据滤波器结构或者是信号流程技术，而最后一种方法则是建立在算法基础上的(参阅第 2 章)。这些技术都需要示例来说明。为了简化每一个例子的 VHDL 表述，在此只考虑一阶 IIR 滤波器的情形，但是同样的思路对高阶 IIR 滤波器也是适用的，可以查阅相应的参考文献书目。

4.4.1　时域交叉

下面来研究一下一阶 IIR 系统的差分方程：

$$y[n+1]=ay[n]+bx[n] \tag{4.10}$$

一阶系统的输出，就是 $y[n+1]$，可以采用预先考虑的方法计算，将 $y[n+1]$带入 $y[n+2]$的差分方程。就是：

$$y[n+2]=ay[n+1]+bx[n+1]=a^2y[n]+abx[n]+bx[n+1] \tag{4.11}$$

等价的系统如图 4-14 所示。

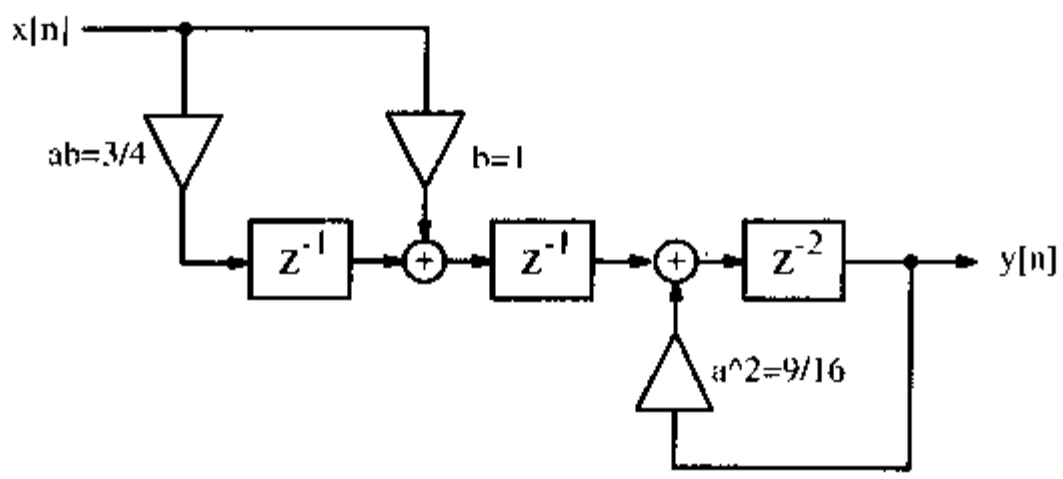


图 4-14　采用预先考虑算法的有耗积分器

通过采用预先考虑$(S-1)$步骤的转换，就可以生成这一概念，结果如下：

$$y[n+S]=a^S y[n]+\underbrace{\sum_{k=0}^{S-1}a^k bx[n+S-1-k]}_{(\eta)} \tag{4.12}$$

从中可以看到：(η)项定义了 FIR 滤波器的系数$\{b, ab, a^2b,\ldots, a^{S-1}b\}$，后者可以采用第 3 章提到的流水线技术(例如：流水线乘法器和流水线加法器树)。(4.12)中的递归部分也可以利用系数为 a^S 的 S 阶流水线乘法器来实现。下面用一个示例来加以说明。

例 4.3 有耗积分器 II

再来研究一下例 4.1 中的有耗积分器，但是这次要加上预先考虑。图 4-14 给出了预先考虑的有耗积分器，它是由一个非递归部分(例如：x 的 FIR 滤波器)和一个具有延迟为 2 和系数为 9/16 的递归部分构成的。

$$y[n+2]=\frac{3}{4}y[n+1]+x[n+1]=\frac{3}{4}(\frac{3}{4}y[n]+x[n])+x[n+1]$$

$$=\frac{9}{16}y[n]+\frac{3}{4}x[n]+x[n+1] \tag{4.13}$$

实现这种 IIR 滤波器的 VHDL 代码[2]如下：

```
PACKAGE n_bit_int IS                    -- User defined type
  SUBTYPE BITS15 IS INTEGER RANGE - 2**14 TO 2**14 - 1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY iir_pipe IS
  PORT ( x_in   : IN    BITS15;    -- Input
```

注 2：这一例子相应的 Verilog 代码文件 iir_pipe.v 可以在附录 A 中找到。

```
        y_out : OUT   BITS15;   -- Result
        clk   : IN    STD_LOGIC);
END iir_pipe;

ARCHITECTURE flex OF iir_pipe IS

  SIGNAL   x, x3, sx, y, y9 : BITS15;

BEGIN

  PROCESS   -- Use FFs for input, output and pipeline stages
  BEGIN
    WAIT UNTIL clk = '1';
    x    <= x_in;
    x3   <= x / 2 + x / 4;     -- Compute x*3/4
    sx   <=   x + x3; -- Sum of x element i.e. output FIR part
    y9   <= y / 2 + y / 16;   -- Compute y*9/16
    y    <= sx + y9;          -- Compute output
  END PROCESS;

  y_out <= y ;       -- Connect register y to output pins
```

END flex;

在这个例子中流水线加法器和乘法器是在两个步骤里实现的。第一步计算$\frac{9}{16}*y[n]$；第二步计算$\frac{3}{4}x[n]+x[n+1]$，再加上$\frac{9}{16}*y[n]$。这一设计使用了 64 个逻辑单元，以 73.52MH$_z$ 的 Registered Performance 运行。滤波器对于幅值为 1000 的脉冲的响应如图 4-15 所示。

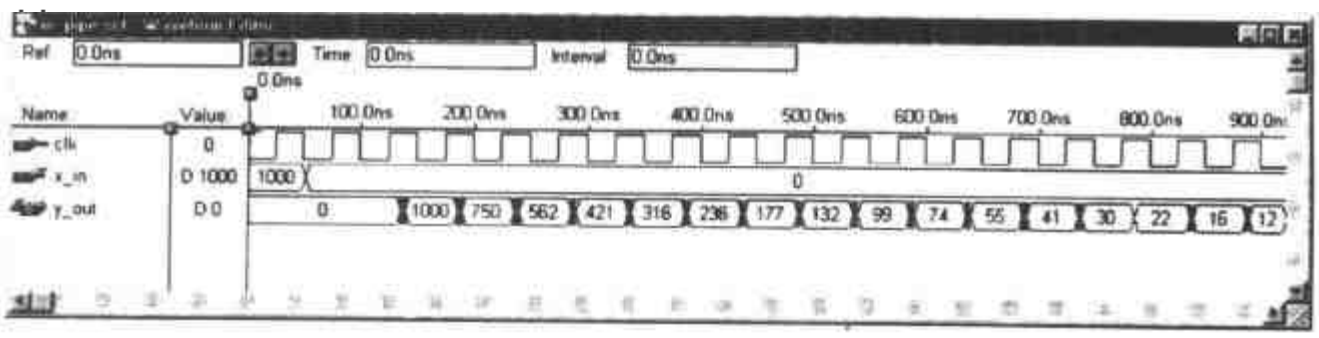


图 4-15　预先考虑有耗积分器脉冲响应的 VHDL 仿真

与例 4.1 中用 31 个 LC、运行速度为 55.86MHz 的预先考虑方案相比，就会发现预先考虑的流水线技术在设计的复杂程度上增加了一倍，速度提高了 31%。图 4-2 给出了两种滤波器对于幅值为 1000 的脉冲的响应，图 4-15 揭示了预先考虑结构有额外的总延迟。两种方法的量化效应差别将为±2。

在练习 4.5 中将讨论一种采用标准逻辑向量数据类型和 LPM_ADD_SUB 兆函数的第二种可选设计方案。这种方法生成的 VHDL 代码较长，但是优点是可以在符号扩展和乘法器的位层次上直接控制。

4.4.2 群集和分散预先考虑的流水线技术

群集和分散预先考虑结构为设计添加了极点和零点的自消除，进而简化滤波器递归部分的流水线。群集方法是在传递函数的分母中引入了额外的极点/零点，令 z^{-1}，z^{-2}，…，$z^{-(S-1)}$ 的系数为 0，下面给出一个二阶滤波器的群集示例。

例 4.4 群集方法

假定二阶传递函数的两个极点分别位于 1/2 和 3/4 处，其传递函数如下：

$$F(z)=\frac{1}{1-1.25z^{-1}+0.375z^{-2}}=\frac{1}{(1\quad 0.5z^{-1})(1-0.75z^{-1})} \tag{4.14}$$

在 $z=-1.25$ 处增加一个抵消极点/零点，生成新的传递函数：

$$F(z)=\frac{1+1.25z^{-1}}{1-1.1875z^{-2}+0.4688z^{-3}} \tag{4.15}$$

滤波器的递归部分可以用额外的流水线级实现。

就上面的示例(例如：$z_\infty=-1.25$)来说，群集的问题是被抵消的极点/零点对可能位于单位圆之外。如果极点/零点的抵消不完全，就会在设计中引入不稳定性。一般来说，二阶系统的极点是在 r_1，r_2 处，有一个额外的抵消对，而且有一个极点处在 $1/(r_1+r_2)$，也就是单位圆外(因为 $(r_1+r_2)<1$)。Soderstrand 等人[77]已经给出了一种稳定的群集方法，一般就是引入多于一对的极点/零点对。

分散预先考虑方法没有引入稳定性问题。对于一个在 P 点有极点的原始滤波器而言，引入了$(S-1)$个抵消位于 $z_k=pe^{j\pi k/S}$ 的极点/零点对。结果是在传递函数的分母中只有 z^0，z^S，z^{-2S} 等项有零系数。

例 4.5 分散预先考虑方法

考虑用两个额外的流水线级实现一个极点位于 $z_{\infty 1}=-0.5$ 和 $z_{\infty 2}=-0.25$ 的二阶系统。在 1/2 和 1/4 有极点的二阶滤波器的传递函数形式如下：

$$F(z)=\frac{1}{1-3/4z^{-1}+1/8z^{-2}} \tag{4.16}$$

分散预先考虑方法通过在 $0.25e^{\pm j2\pi/3}$ 和 $0.5e^{\pm j2\pi/3}$ 增加极点/零点对引入了两个额外的流水线级。在这一位置添加极点/零点对后，传递函数就变成：

$$\begin{aligned}F(z)&=\frac{1}{1+0.5z^{-1}+0.25z^{-2}}\cdot\frac{(1+0.5z^{-1}+0.25z^{-2})(1+0.75z^{-1}+0.5625z^{-2})}{(1+0.75z^{-1}+0.5625z^{-2})(1-0.75z^{-1}+0.125z^{-2})}\\&=\frac{1+1.25z^{-1}+0.1875z^{-2}+0.4687z^{-3}+0.1406z^{-4}}{1-0.5469z^{-3}+0.0527z^{-6}}\\&=\frac{512+640z^{-1}+608z^{-2}+240z^{-3}}{512-280z^{-3}+27z^{-6}}\end{aligned}$$

递归部分可以采用两个额外的流水线级来实现。

有趣的是我们可以看到，对于一阶 IIR 系统而言，群集和分散预先考虑方法引入的是相同的极点/零点抵消对，位置处于以原点为圆心的圆上，角度相差 $2\pi/S$。非递归部分可以根据下式以“2 的幂分解”的形式实现：

$$(1+az^{-1})(1+a^2z^{-2})(1+a^4z^{-4})\cdots \tag{4.17}$$

图 4-16 给出了一阶部分的一对极点/零点图像，其中的一阶递归部分需要 4 个流水线级来实现。

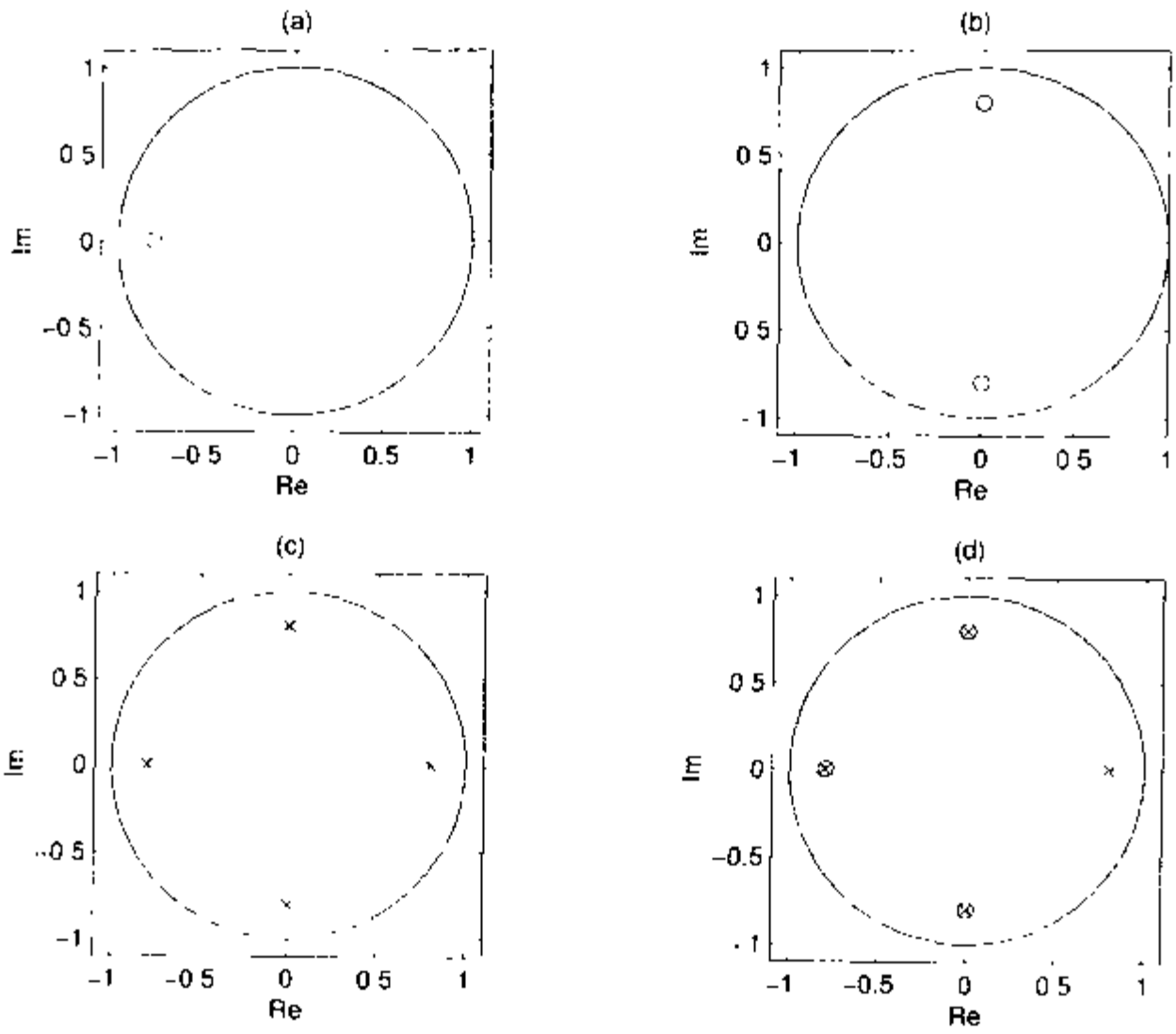


图 4-16　分散预先考虑的一阶 IIR 滤波器的极点/零点分布图

(a) $F_1(z)=(1+az^{-1})$ (b) $F_2(z)=1+a^2z^{-2}$ (c) $F_3(z)=(1-a^4z^{-4})$

(d) $F(z)=\prod_k F_k(z)=(1+az^{-1})(1+a^2z^{-2})/(1-a^4z^{-4})=1/(1-az^{-1})$

4.4.3　IIR 抽取设计

Martinez 和 Parks 已经提出了一种基于极小化极大方法的滤波器设计算法(参阅第 5 章)。最终的传递函数满足：

$$F(z)=\frac{\sum_{l=0}^{L}b[l]z^{-l}}{1-\sum_{n=0}^{N/S}a[n]z^{-nS}} \tag{4.18}$$

也就是在分母中所有其他 S 个系数都是非零的。由此，递归部分(也就是分母)可以采用 S 级流水线。可以看到，在最终的极点/零点分布中所有的零点都位于单位圆上，就像一般情况下的椭圆滤波器一样，而极点位于主轴有一个 $2\pi/S$ 角度差的圆上，请参阅图 4-17(b)。

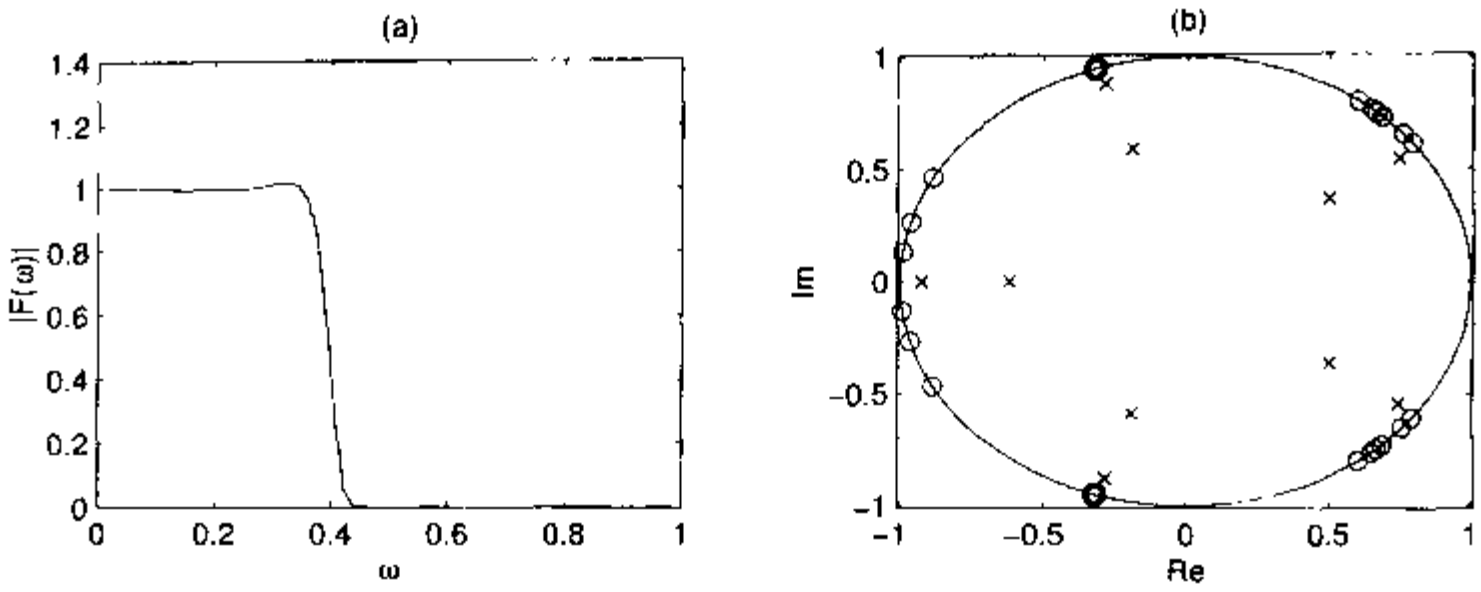


图 4-17　$S=5$ 的 37 阶 Martinez-Parks IIR 滤波器的 (a) 传递函数 (b) 极点/零点分布图

4.4.4　并行处理

并行处理滤波器的实现[80]是由 P 个并行 IIR 通路构成的，每个信道都以 $1/P$ 个输入采样速率运行，它们在输出位置靠多路复用器合成在一起，如图 4-18 所示。一般情况下，由于多路复用器要比乘法器和/或加法器速度快，所以并行方法速度也就更快。进一步讲，每个信道 P 都有一个 P 因子来计算其指定的输出。

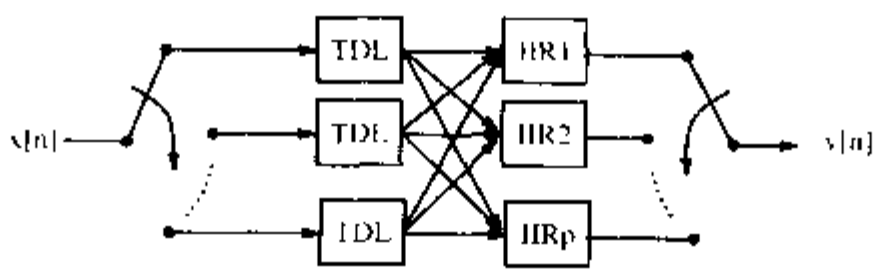


图 4-18　并行 IIR 滤波器的实现。抽头延迟线(TDL)以 1/P 个输入采样速率运行

为了方便说明。再来考虑一下 $P=2$ 的一阶系统，预先考虑结构与式(4.11)相同：

$$y[n+2]=ay[n+1]+x[n+1]=a^2y[n]+ax[n]+x[n+1] \tag{4.19}$$

现在将其分成偶数 $n=2k$ 和奇数 $n=2k-1$ 输出序列，得到：

$$y[n+2]=\begin{cases}y[2k+2]=a^2y[2k]+ax[2k]+x[2k+1]\\ y[2k+1]=a^2y[2k-1]+ax[2k-1]+x[2k]\end{cases} \tag{4.20}$$

其中 n，$k\in Z$。这两个方程是下面并行 IIR 滤波器 FPGA 实现的基础。

例 4.6　有耗积分器 III

研究一下 $a=3/4$ 的并行有耗积分器的实现，就好比是例 4.1 和例 4.3 中给出的方法的拓展。如图 4-19 所示，双信道并行有耗积分器是两个非递归部分(也就是 x 的 FIR 滤波器)和两个延迟为 2、系数为 9/16 的递归部分的组合。这一设计的 VHDL 代码[3]如下：

```
PACKAGE eight_bit_int IS                -- User defined type
  SUBTYPE BITS15 IS INTEGER RANGE - 2**14 TO 2**14 - 1;
END eight_bit_int;
```

注 3：这一例子相应的 Verilog 代码文件 iir_par.v 可以在附录 A 中找到。

```
LIBRARY work;
USE work.eight_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY iir_par IS                              ------> Interface
  PORT ( clk      : IN  STD_LOGIC;
         x_in     : IN  BITS15;
         clk2     : OUT STD_LOGIC;
         y_out    : OUT BITS15);
END iir_par;

ARCHITECTURE flex OF iir_par IS

  TYPE STATE_TYPE IS (even, odd);
  SIGNAL  state                          : STATE_TYPE;
  SIGNAL  x_even, x_odd, xd_odd, x_wait  : BITS15;
  SIGNAL  y_even, y_odd, y_wait, y       : BITS15;
  SIGNAL  x_e, x_o, y_e, y_o             : BITS15;
  SIGNAL  sum_x_even, sum_x_odd          : BITS15;
  SIGNAL  clk_div2                       : STD_LOGIC;

BEGIN

  Multiplex: PROCESS      --> Split x into even and odd samples
  BEGIN                   --> recombine y at clk rate
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN even =>
          x_even <= x_in;
          x_odd <= x_wait;
          clk_div2 <= '1';
          y <= y_wait;
          state <= odd;
      WHEN odd =>
          x_wait <= x_in;
          y <= y_odd;
          y_wait <= y_even;
          clk_div2 <= '0';
          state <= even;
    END CASE;
 END PROCESS Multiplex;
```

```
    y_out <= y;
    clk2   <= clk_div2;

    Arithmetic: PROCESS
    BEGIN
      WAIT UNTIL clk_div2 = '0';
      sum_x_even <= (x_even * 2 + x_even) /4 + x_odd;
      y_even <= (y_even * 8 + y_even ) /16 + sum_x_even;
      xd_odd <= x_odd;
      sum_x_odd <= (xd_odd * 2 + xd_odd) /4 + x_even;
      y_odd   <= (y_odd * 8 + y_odd) / 16 + sum_x_odd;
    END PROCESS Arithmetic;

END flex;
```

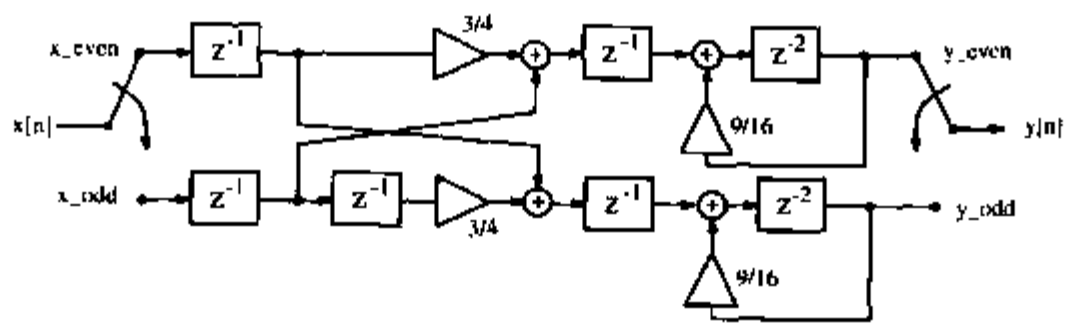


图 4-19　有两个通路的并行 IIR 滤波器的实现

本设计采用了两个 PROCESS 声明来实现。第一个是 PROCESS Multiplex，x 被分成偶数和奇数部分。输出 y 是以 clk 速率重新组合的。另外，第一个 PROCESS 声明生成了第二个时钟信号，运行速率是 clk/2。第二个模块是根据式(4.20)来实现滤波器的算法。用 MaxPlusII 软件测试设计的 Registered Performance 时就会出现一个问题，MaxPlusII 软件不能够在多时钟域内计算 Registered Performance。所以要假定输入多路复用器以两倍算法的速度运行，预计 58.13MHz 就会产生多于 100MHz 的输入速率。这就是 y_odd 和 y_even 滤波器退到 clk_div2 时钟的结果。可以通过图 4-20 给出的仿真对此进行确证。选择的 clk 频率要比计算的 Registered Performance 高一些。

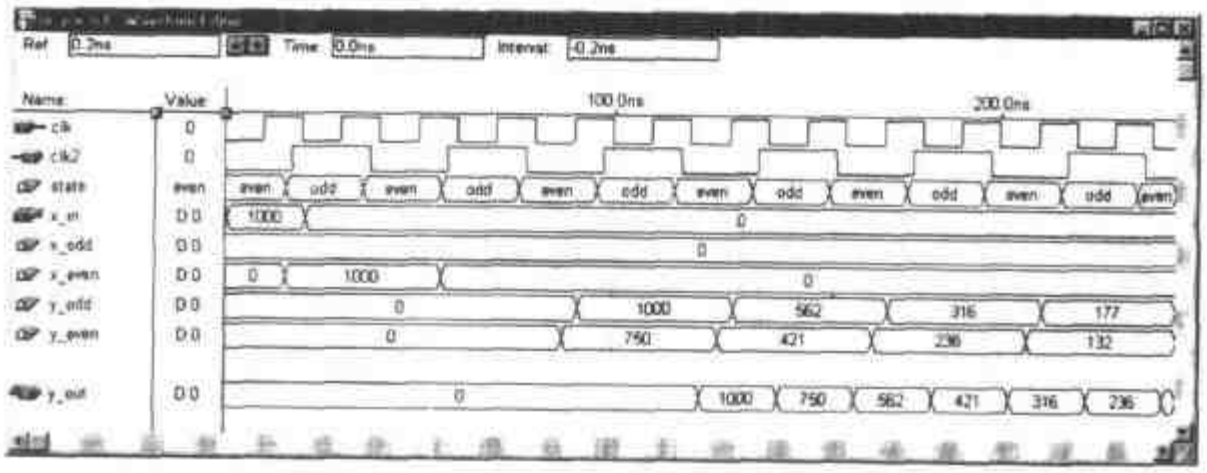


图 4-20　并行 IIR 滤波器对脉冲 1000 的响应的 VHDL 仿真

与前面给出的其他方法相比，并行实现的缺点在于需要较高的实现成本，本例中用了 213 个 LC。

4.4.5　采用 RNS 的 IIR 设计

由于余数系统(Residue Number System, RNS)使用的是固有的短字长，所以是实现快速(递归)IIR 滤波器的一个优先候选方案。在典型的 IIR-RNS 设计中，系统是作为递归和非递归系统的一个集合被实现的，每个系统都是按着 FIR 结构(请参阅图 4-21)定义的。每个 FIR 都可以采用四分之一平方乘法器或者是像第 2 章给出的索引域一样在 RNS-DA 中实现。

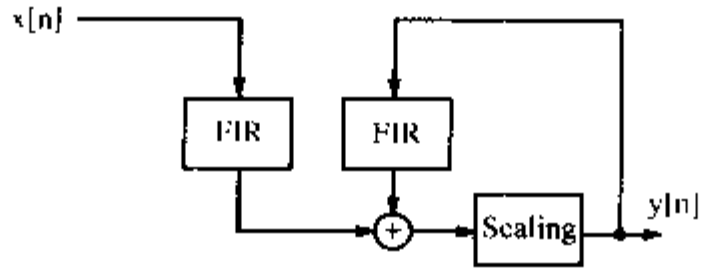


图 4-21　采用两个 FIR 部分和 Scaling 的 IIR 滤波器的 RNS 实现

对于稳定的滤波器而言，递归部分应该与控制动态范围增长成比例。比例运算可以用混合基数转换(mixed radix conversion)、中国余数定理(Chinese Remainder Theorem, CRT)或者是 ϵ—CRT 方法实现。对于高速设计，推荐在群集或者分散预先考虑采用流水线技术[35]的基础上增加一个额外的流水线延迟。我们将在 5.3 节详细研究 RNS 递归滤波器的设计，并可以看到 RNS 设计可以将速度从 50MHz 提高到 70MHz 以上。

4.5　练习

练习使用 MaxPlusII 软件

4.1：(a) 用 MaxPlusII 实现一个极点位于 $z_{=0}=3/8$ 处且输入宽度为 12 位的一阶 IIR 滤波器。

(b) 确定 LC 的数量和 Registered Performance。

(c) 用输入为 100 的脉冲仿真设计。

4.2：(a) 用 MaxPlusII 实现一个极点位于 $z_{=0}=3/8$ 处、输入宽度为 12 位且预先考虑一步的一阶 IIR 滤波器。

(b) 确定 LC 的数量和 Registered Performance。

(c) 用输入为 100 的脉冲仿真设计。

4.3：(a) 用 MaxPlusII 实现一个极点位于 $z_{=0}=3/8$ 处输入宽度为 12 位且采用双信道并行设计的一阶 IIR 滤波器。

(b) 确定 LC 的数量和 Registered Performance。

(c) 用输入为 100 的脉冲仿真设计。

4.4：(a) 用 MaxPlusII 实现例 4.1 中的一阶 IIR 滤波器，使用 15 位 std_logic_vector，并且用 2 个 lpm_add_sub 宏函数实现加法器。

(b) 确定 LC 的数量和 Registered Performance。

(c) 用输入为 1000 的脉冲仿真设计，并将结果与图 4-3 进行比较。

4.5：(a) 用 MaxPlusII 实现例 4.3 中的一阶流水线 IIR 滤波器，使用 15 位 std_logic_vector，并且用 4 个 lpm_add_sub 宏函数实现加法器。

(b) 确定 LC 的数量和 Registered Performance。

(c) 用输入为 1000 的脉冲仿真设计，并将结果与图 4-15 进行比较。

4.6：Shajaan 和 Sorensen 已经指出，可以通过将系数实现为“有符号 2 的幂(signed-power-of-two, SPT)”的值来有效地设计 IIR Butterworth 滤波器[81]。N 节级联滤波器的传递函数：

$$F(z)=\prod_{l=1}^{N} S[l]\frac{b[l,0]+b[l,1]z^{-1}+b[l,2]z^{-2}}{a[l,0]+a[l,1]z^{-1}+a[l,2]z^{-2}} \tag{4.21}$$

可以用图 4-11 中的二阶部分实现。例 4.2 中的 10 阶滤波器可以用下面的 SPT 滤波器系数实现[81]：

l	S[l]	1/a[l,0]	a[l,1]	a[l,2]
1	2^{-1}	2^{-1}	$-1-2^{-4}$	$1-2^{-2}$
2	2^{-1}	2^{-1}	$-1-2^{-1}$	$1-2^{-5}$
3	2^{-1}	2^{-1}	$-1-2^{-1}$	$2^{-1}+2^{-6}$
4	1	2^{-1}	$-1-2^{-2}$	$2^{-2}+2^{-5}$
5	2^{-1}	2^{-1}	$-1-2^{-1}$	$2^{-2}+2^{-4}$

由于 Butterworth 滤波器的所有零点都位于 $z=-1$ 处，所以我们选择 $b[0]=b[2]=0.5$ 和 $b[1]=1$。

(a) 计算并画出第一级双四边形的传递函数，并且完成滤波器。

(b) 实现并且仿真 8 位输入时的第一级双四边形。

(c) 用 MaxPlusII 构造并仿真 5 阶滤波器。

(d) 确定滤波器 LC 的数量和 Registered Performance。

第5章　多级信号处理

概述

数字信号处理中的经常性任务就是根据有用信号来调整采样速率。具有不同采样速率的系统就称之为多级系统。在本章，我们将列举两个典型的例子来说明多级 DSP 系统中的抽取和插入。然后将介绍多相符号，并研究一些有效的抽样设计。在本章的结尾，还要讨论滤波器组和一个崭新的非常著名的 DSP 工具箱的补充——小波分析。

5.1　抽取和插值

如果在 A/D 转换之后，可以在一个非常窄的频带(通常是低通或带通)中找到有用的信号,那样就可以合理地利用低通或者带通滤波器进行滤波，从而降低采样速率。向下采样前的窄带滤波器通常称之为抽取器[65]1。滤波、向下采样和以及对频谱的影响请参阅图 5-1。

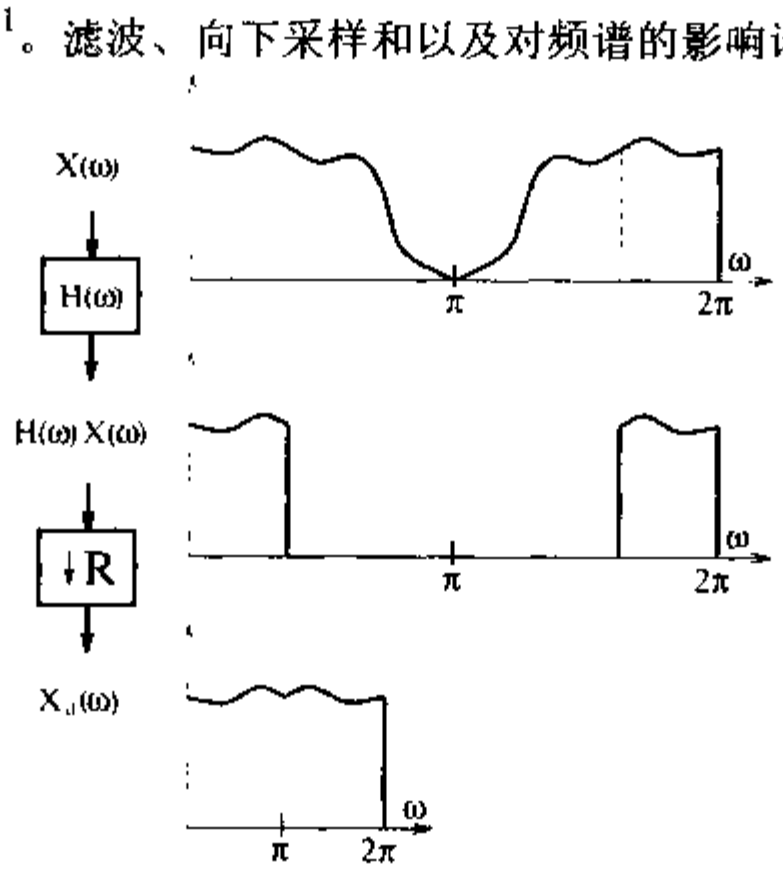


图 5-1　信号 $x(n) \circ\!-\!\bullet X(\omega)$ 的抽取

我们可以把采样速率降低到称之为“奈魁斯特速率”的极限，也就是说为了避免出现假信号，采样速率必须高于信号的带宽。低通滤波器的假信号如图 5-2 所示。假信号是不可修补的，所以必须不惜任何代价消除假信号。

注 1：一些作者将向下采样称为抽取器。

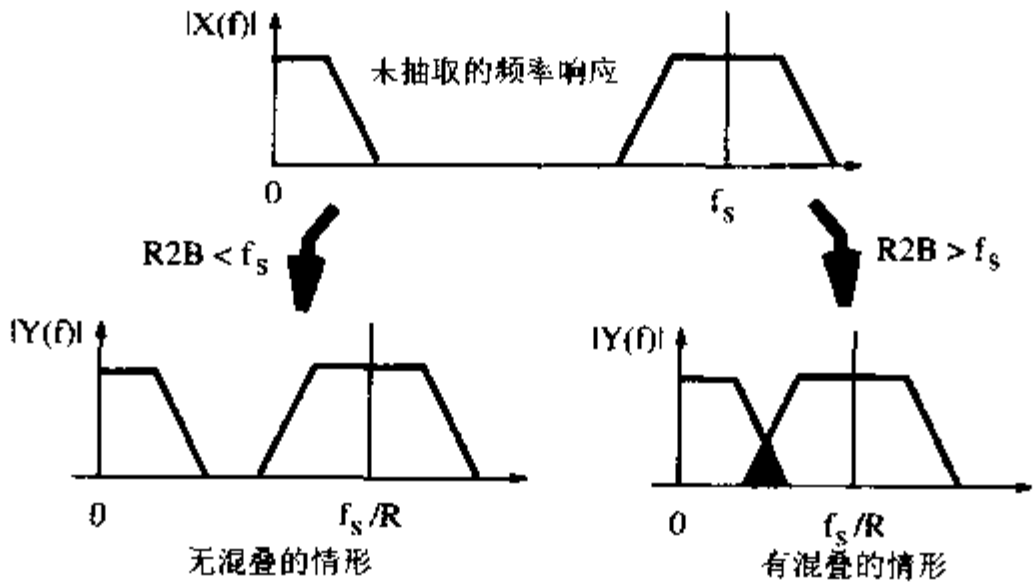


图 5-2　无混叠抽取和有混叠抽取的情形

对于带通信号而言，有用频带必须落在完整的频带之内。如果 f_s 是采样速率，R 是所要的向下采样因数，则有用带宽必须落在：

$$k\frac{f_s}{2R} < f < (k+1)\frac{f_s}{2R} \qquad k \in N \tag{5.1}$$

如果不这样的话，即使采样速率高于奈魁斯特频率，也还是有可能由于来自负频带的“复制”而出现混叠，如图 5-3 所示。

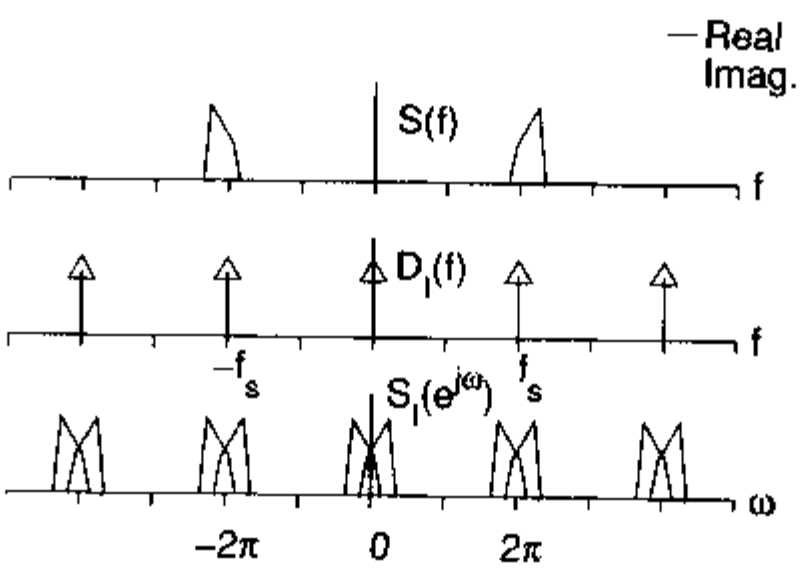


图 5-3　整数频带的扰乱(©1995 VDI 出版社[4])

提高采样速率可以起到一定的作用，例如：在 D/A 转换过程中，典型的 D/A 转换器在输出位置采用一阶采样-保持，生成阶梯状输出函数。这可以利用模拟 1/sinc(x)补偿滤波器进行补偿，但是在大多数情况下采用数字的解决方案会更加有效。在数字域中，我们可以采用一个扩展器和一个附加滤波器来获得需要的频带。从图 5-4 中可以看到引入的零点生成了基带频谱的额外副本，后者必须在信号进入 D/A 转换器处理之前消除。插值[2]越多，输出信号也就越光滑，请参阅图 5-5。

注 2:　一些设计者将扩展器称为插值。

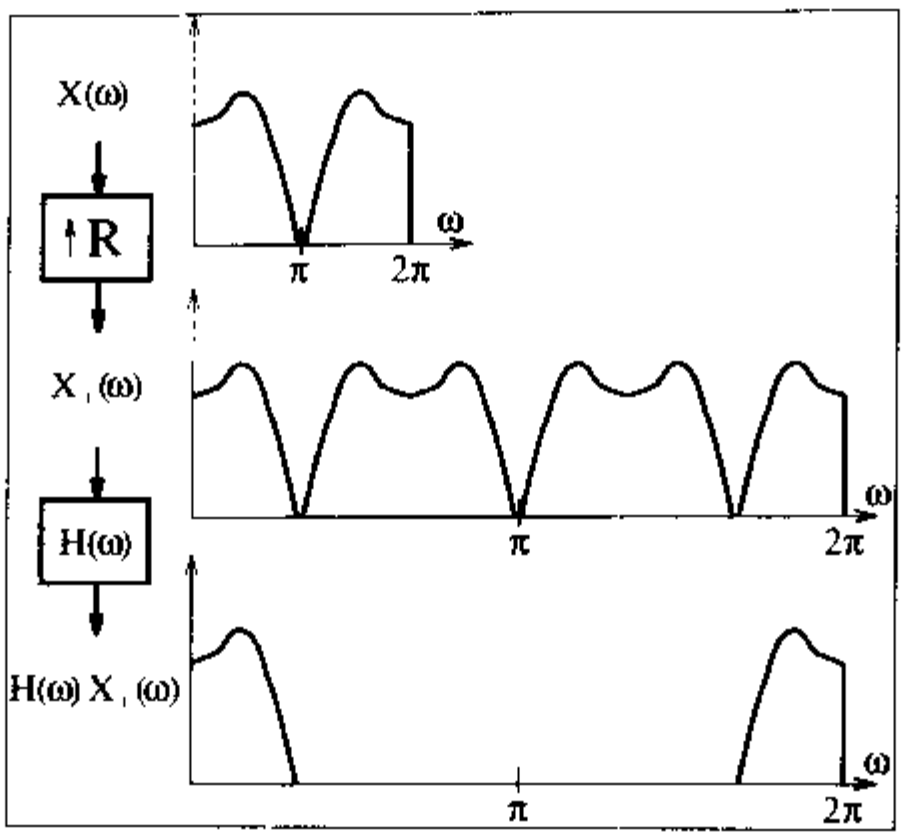


图 5-4　插值示例 $x[n] \circ\!-\!\bullet\, X(\omega)$，$R=3$

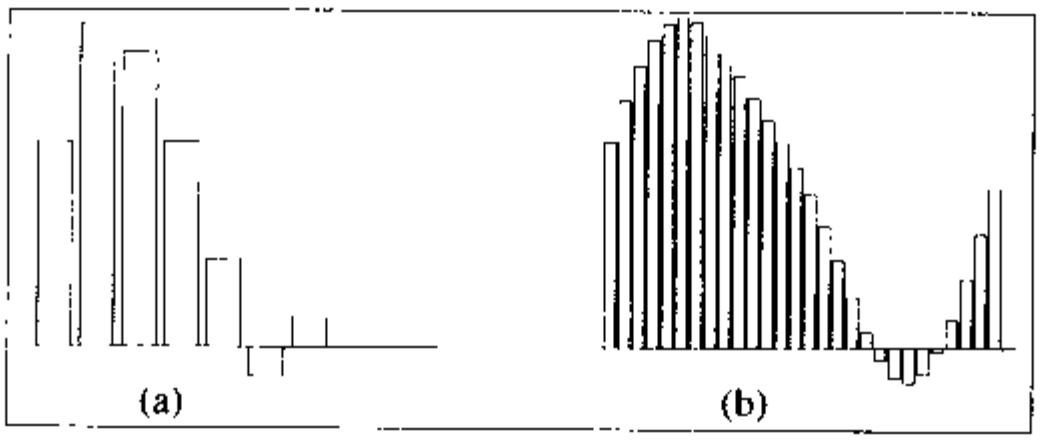


图 5-5　D/A 转换 (a) 低采样过密、高递降 (b) 高采样过密、低递降

5.1.1　Noble 恒等式

当处理多级系统的信号流程图时，经常可以像图 5-6 一样重新安排滤波器和向下采样/扩展器，这是非常有用的。这就是所谓的“Noble”关系式[82]。对于抽取器而言，后续跟随着：

$$(\downarrow R)F(z) = F(z^R)(\downarrow R) \tag{5.2}$$

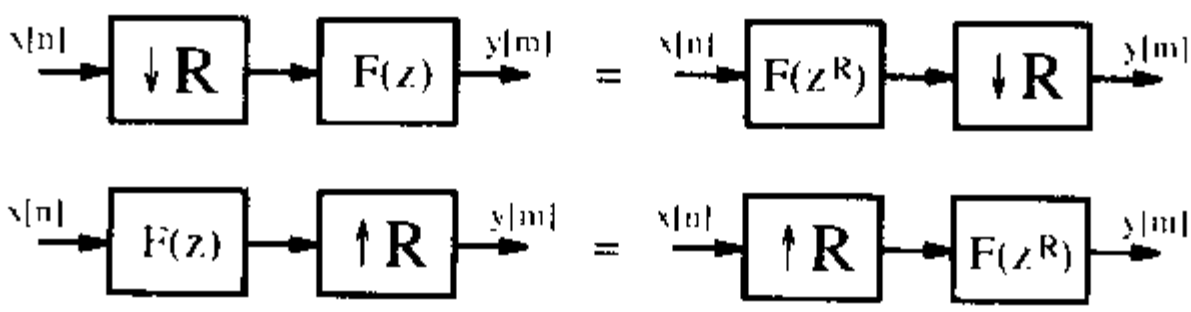


图 5-6　等价多级系统(Noble 关系)

也就是说首先进行向下采样，就可以将滤波器的长度 $F(z^R)$ 降低一个 R 因子。

就插值而言，Noble 关系式可以定义成：

$$F(z)(\uparrow R) = (\uparrow R)F(z^R) \tag{5.3}$$

也就是说，在插值中将滤波器放置在扩展器之前，就可以得到降低了 R 次的滤波器。

我们在 5.2 节中讨论多相实现的时候，这两个关系式是非常有用的。

5.1.2 用有理数因子进行采样速率转换

如果多级系统的输入输出速率不是一个整数因子的话，在采样速率中就需要采用有理数因子 R_1/R_2。要想更加精确的话，首先可以采用插值将采样速率提高 R_1 倍，然后利用抽取器将向下采样降低 R_2 倍。由于插值和抽取所采用的都是低通滤波器，因此根据图 5-7 上面的结构，我们只需要实现使用较小通频带频率的低通滤波器即可，也就是：

$$f_p = \min(\frac{\pi}{R_1}, \frac{\pi}{R_2}) \tag{5.4}$$

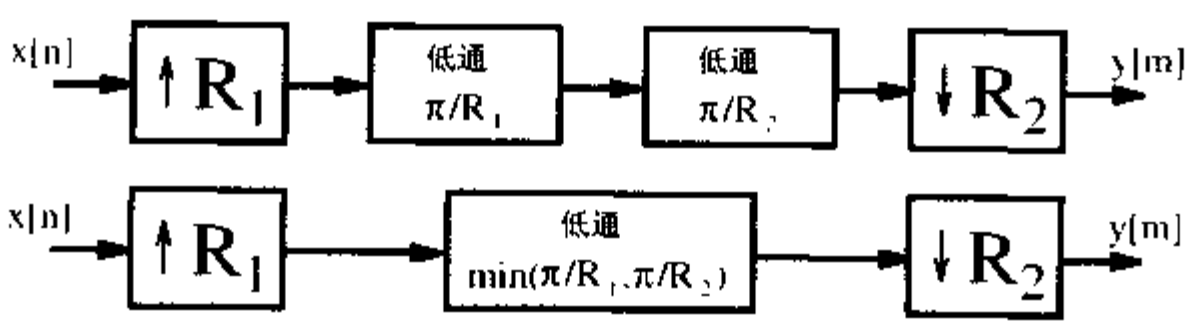


图 5-7　非整数抽取系统。(上) 插入器和抽取器的级联 (下) 最终合成的低通滤波器

图 5-7 下面的结构给出了相应的图形化解释。

5.2 多相分解

当在 IIR 或者 FIR 滤波器和滤波器组中实现抽取或插值时，多相分解是非常有用的。为了说明这一点，我们来研究一下 FIR 抽取滤波器的多相分解。如果在图 3-1 的 FIR 滤波器结构中添加 R 因子倍的向下采样，就会发现只需要计算时间情况中的输出 $y[n]$即可：

$$y[0], y[R], y[2R], \cdots \tag{5.5}$$

这样就不需要计算卷积的所有乘积和 $x[n]$f$[n-k]$了。例如：$x[0]$只需要乘以

$$f[0], f[R], f[2R], \cdots \tag{5.6}$$

除了 $x[0]$之外，其余的系数只需要乘以

$$x[R], x[2R], \cdots \tag{5.7}$$

由此就可以合理地将输入信号按下面的公式分成 R 个独立的序列：

$$x[n]=\sum_{r=0}^{R-1}x_r[n]$$
$$x_0[n]=\{x[0],x[R],\ldots\}$$
$$x_1[n]=\{x[1],x[R+1],\ldots\}$$
$$\vdots$$
$$x_{R-1}[n]=\{x[R-1],x[2R-1],\ldots\}$$

也可以将滤波器 f(n)分成 R 个序列：

$$f[n]=\sum_{r=0}^{R-1}f_r[n]$$
$$f_0[n]=\{f[0],f[R],\ldots\}$$
$$f_1[n]=\{f[1],f[R+1],\ldots\}$$
$$\vdots$$
$$f_{R-1}[n]=\{f[R-1],f[2R-1],\ldots\}$$

图 5-8 给出了采用多相分解的抽取器滤波器的实现。该抽取器的运行速度可以比后面跟随向下采样的通常 FIR 滤波器的速度快 R 倍。滤波器 $f_r[n]$就称为**多相滤波器**，因为他们都具有相同的幅值传递函数，但是它们被一个采样延迟分隔开，也就引入了相位偏移。

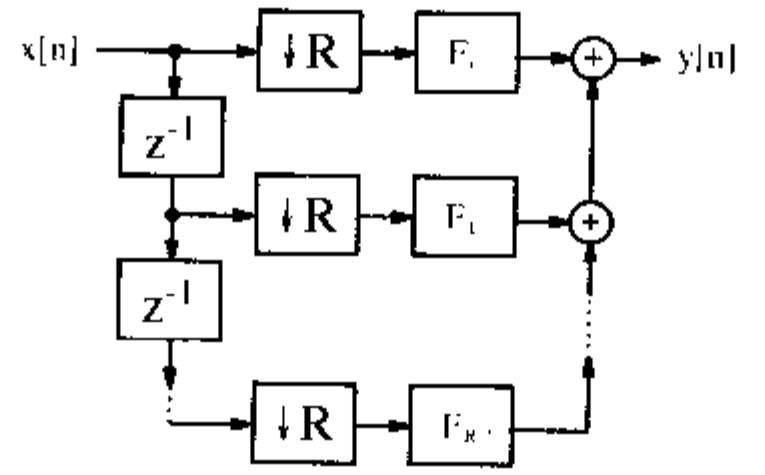


图 5-8　抽取滤波器的多相实现

下面给出说明多相分解的最终示例。

例 5.1　多相抽取器滤波器

考虑一个 Daubechies 长度是 4 的 $G(z)$滤波器，并且 R=2。

$$G(z)=((1+\sqrt{3})+(3+\sqrt{3})z^{-1}+(3-\sqrt{3})z^{-2}+(1-\sqrt{3})z^{-3})\frac{1}{4\sqrt{2}}$$

$$G(z)=0.48301+0.8365z^{-1}+0.2241z^{-2}-0.1294z^{-3}$$

将滤波器量化精确到 8 位精度，就得到：

$$G(z)=(124+214z^{-1}+57z^{-2}-33z^{-3})/256$$
$$G(z)=G_0(z^2)+z^{-1}G_1(z^2)$$

$$=\underbrace{(\frac{124}{256}+\frac{57}{256}z^{-2})}_{G_0(z^2)}+z^{-1}\underbrace{(\frac{214}{256}-\frac{33}{256}z^{-2})}_{G_1(z^2)}$$

就有

$$G_0(z)=\frac{124}{256}+\frac{57}{256}z^{-1}\qquad G_1(z)=\frac{214}{256}-\frac{33}{256}z^{-1} \tag{5.8}$$

下面的 VHDL 代码[3]给出 DB4 的多相实现。

```
PACKAGE n_bits_int IS                -- User defined types
  SUBTYPE BITS8 IS INTEGER RANGE - 128 TO 127;
  SUBTYPE BITS9 IS INTEGER RANGE - 2**8 TO 2**8 - 1;
  SUBTYPE BITS17 IS INTEGER RANGE - 2**16 TO 2**16 - 1;
  TYPE ARRAY_BITS17_4 IS ARRAY (0 TO 3) of BITS17;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY db4poly IS                              ------> Interface
  PORT (clk                 : IN  STD_LOGIC;
        x_in                : IN  BITS8;
        clk2                : OUT STD_LOGIC;
        x_e, x_o, g0, g1    : OUT BITS17;
        y_out               : OUT BITS9);
END db4poly;

ARCHITECTURE flex OF db4poly IS

  TYPE STATE_TYPE IS (even, odd);
  SIGNAL state                    : STATE_TYPE;
  SIGNAL x_odd, x_even, x_wait    : BITS8;
  SIGNAL clk_div2                 : STD_LOGIC;
  -- Arrays for multiplier and taps:
  SIGNAL r   : ARRAY_BITS17_4 ;
  SIGNAL x33, x99, x107, y        : BITS17;

BEGIN

  Multiplex: PROCESS      ------> Split into even and odd
  BEGIN                           -- samples at clk rate
```

注 3：这一例子相应的 Verilog 代码文件 db4poly.v 可以在附录 A 中查到。

```
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN even =>
        x_even <= x_in;
        x_odd  <= x_wait;
        clk_div2 <= '1';
        state <= odd;
      WHEN odd =>
        x_wait <= x_in;
        clk_div2 <= '0';
        state <= even;
    END CASE;
  END PROCESS Multiplex;

  AddPolyphase    : PROCESS (clk_div2,x_odd,x_even)
  VARIABLE m      : ARRAY_BITS17_4 ;
  BEGIN
-- Compute auxiliary multiplications of the filter
    x33   <= x_odd * 32 + x_odd;
    x99   <= x33 * 2 + x33;
    x107  <= x99 + 8 * x_odd;
-- Compute all coefficients for the transposed filter
    m(0) := 4 * (32 * x_even - x_even);          -- m[0] = 127
    m(1) := 2 * x107;                            -- m[1] = 214
    m(2) := 8 * (8 * x_even - x_even) + x_even;  -- m[2] = 57
    m(3) := x33;                                 -- m[3] =  - 33
------> Compute the filters and infer registers
    IF clk_div2'event and (clk_div2 = '0') THEN
---------- Compute filter G0
      r(0) <=  r(2) + m(0);      -- g[0] = 127
      r(2) <=  m(2);             -- g[2] = 57
---------- Compute filter G1
      r(1) <=  - r(3) + m(1);    -- g[1] = 214
      r(3) <=  m(3);             -- g[3] = - 33
---------- Add the polyphase components
      y <= r(0) + r(1);
    END IF;
  END PROCESS AddPolyphase;

  x_e <= x_even; -- Provide some test signal as outputs
  x_o <= x_odd;
  clk2 <= clk_div2;
  g0 <= r(0);
  g1 <= r(1);
```

```
    y_out <= y / 256; -- Connect to output

END flex;
```

第一个 PROCESS 声明是 FSM，包括控制流程和按着采样速率将输入流分解成奇数和偶数采样。第二个 PROCESS 声明包括简化的加法器图(reduced adder graph, RAG)乘法器，最后一个 PROCESS 声明在换位结构中安置了两个滤波器。尽管输出是成比例的，但还是存在潜在的增长，总量 $\Sigma|g_k|=1.673<2^1$。这样输出 y_out 还可以选择一个额外的保护位。本设计需要使用 208 个 LC，运行的 Registered Performance 为 90.90MHz。图 5-9 给出了滤波器的仿真结果。前 4 个输入样本是一个三角函数，用来证明奇偶采样的分离。采用幅值为 100 的脉冲可以验证两个多相滤波器的系数。注意：这里的滤波器不是移不变的！

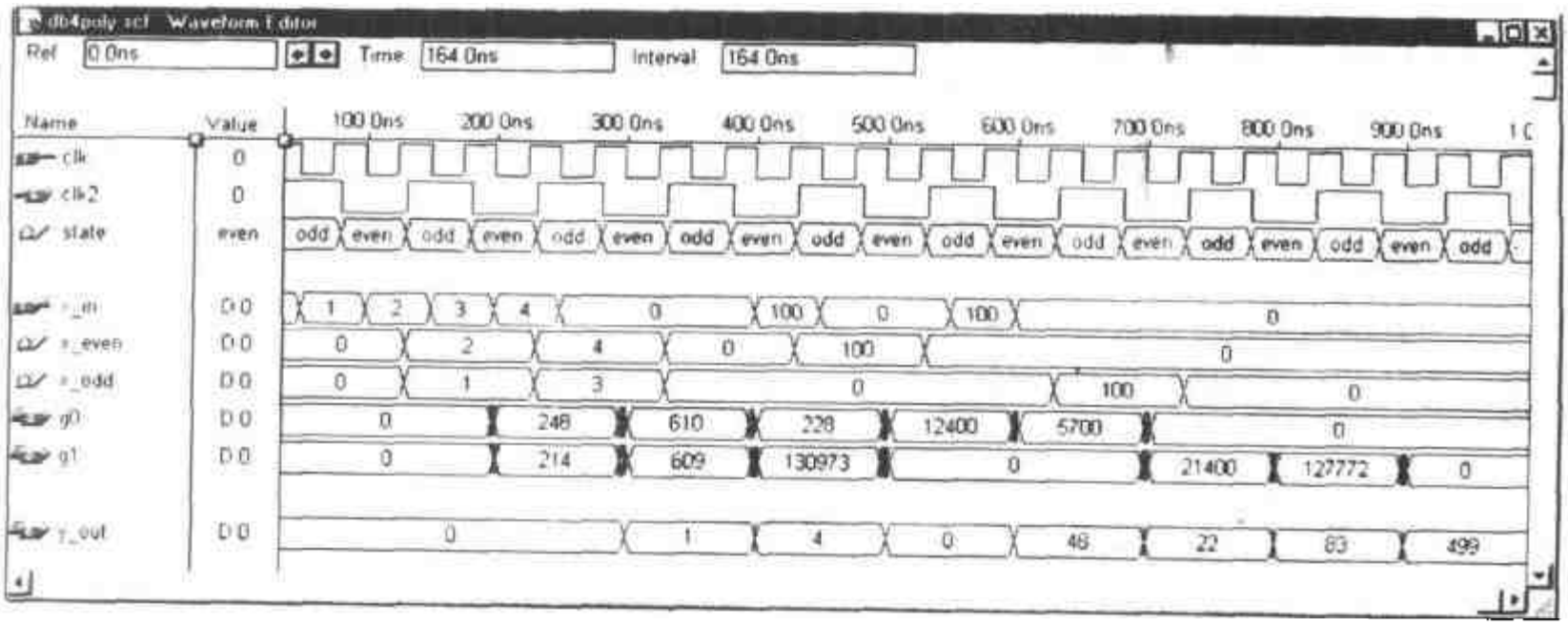


图 5-9　长度为 4 的 Daubechies 滤波器多相实现的 VHDL 仿真

从图 5-9 中 VHDL 的仿真结果可以看出，这样的抽取不再是移不变的，产生了技术上的非线性系统。可以采用一个单脉冲来验证。在偶数编址采样的初始时刻，响应是 $G_0(z)$，而在奇数编址采样的初始时刻，响应是 $G_1(z)$。

5.2.1　递归 IIR 抽取器

将多项分解应用到递归滤波器中也是可行的，并且对速度十分有利。按着 Martinez 和 Parks[79] 的构想，在传递函数：

$$F(z)=\frac{\sum_{l=0}^{L-1}a[l]z^{-l}}{1-\sum_{l=0}^{K-1}b[l]z^{-lR}} \tag{5.9}$$

中，也就是递归部分，只有各自的第 R 个系数。我们在前面的 IIR 滤波器(图 4-17)中已经讨论过这样的设计。图 5-10 指出，与 FIR 抽取器相比，IIR 抽取器依靠滤波器的过渡宽度ΔF，可以节省物理硬件。

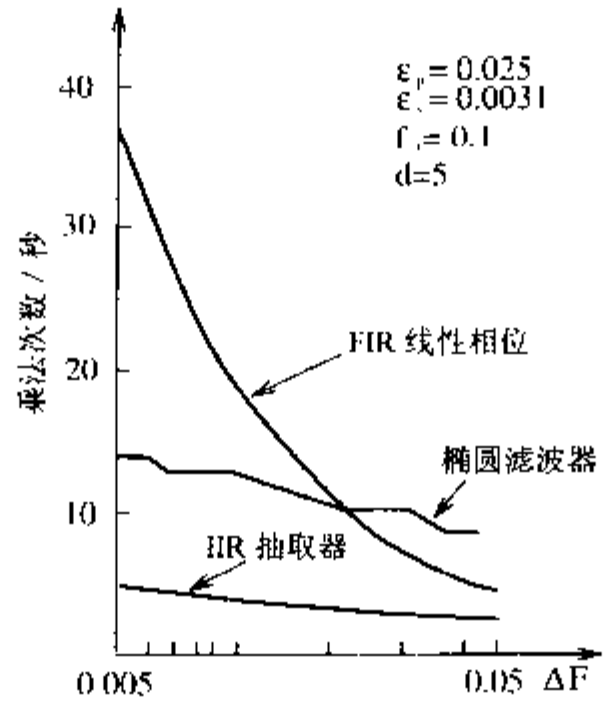


图 5-10　抽取器 $\Delta F = f_p - f_s$ 计算量之比较

5.2.2　快行 FIR 滤波器

多项分解的一个有趣的应用就是所谓的**快行 FIR 滤波器**。这种滤波器的基本构想如下：如果我们将输入信号 $x[n]$分解成 R 个多相组成部分，就可以采用 Winograd 的短卷积算法来实现快行滤波器。下面用 R=2 的示例来加以说明。

例 5.2　快行 FIR 滤波器

将信号 $X(z)$和滤波器 $F(z)$分解成偶数和奇数的多相组成部分，也就是：

$$X(z) = \sum_n x[n]z^{-n} = X_0(z^2) + z^{-1}X_1(z^2) \tag{5.10}$$

$$F(z) = \sum_n f[n]z^{-n} = F_0(z^2) + z^{-1}F_1(z^2) \tag{5.11}$$

$x[n]$和 $f[n]$在时域内的卷积产生了 z 域内的多项式乘法。由此，输出信号：

$$Y(z) = Y_0(z^2) + z^{-1}Y_1(z^2) \tag{5.12}$$

$$= (X_0(z^2) + z^{-1}X_1(z^2))(F_0(z^2) + z^{-1}F_1(z^2)) \tag{5.13}$$

如果将(5.13)分解成多相组成部分 $Y_0(z)$和 $Y_1(z)$，就可以得到：

$$Y_0(z) = X_0(z)F_0(z) + z^{-1}X_1(z)F_1(z) \tag{5.14}$$

$$Y_1(z) = X_1(z)F_0(z) + X_0(z)F_1(z) \tag{5.15}$$

现在将(5.13)与一个 2×2 线性卷积：

$$A(z)\cdot B(z) = (a[0] + z^{-1}a[1])(b[0] + z^{-1}b[1]) \tag{5.16}$$

$$= a[0]b[0] + z^{-1}(a[0]b[1] + a[1]b[0]) + a[1]b[1]z^{-2} \tag{5.17}$$

进行比较。就可以注意到 z^{-1} 的因子是相同的。但是对于 $Y_0(z)$而言，我们必须计算一个额外的

加法来获得正确的相位关系。Winograd[83]已经编译了一个短卷积算法列表，可以用 3 次乘法和下面的 6 次加法计算线性 2×2 卷积：

$$\begin{aligned} &a[0]=x[0]-x[1] \quad a[1]=x[0] \quad a[2]=x[1]-x[0] \\ &b[0]=f[0]-f[1] \quad b[1]=f[0] \quad b[2]=f[1]-f[0] \\ &c[k]=a[k]b[k] \quad k=0,1,2 \\ &y[0]=c[1]+c[2] \quad y[1]=c[1]-c[0] \end{aligned} \tag{5.18}$$

现在在短卷积算法的帮助下，我们可以按如下方式定义快行滤波器：

$$\begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} F_0 & 0 & 0 \\ 0 & F_0+F_1 & 0 \\ 0 & 0 & F_1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 0 \\ 1 & z^{-1} \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} \tag{5.19}$$

图 5-11 给出了图形化解释。

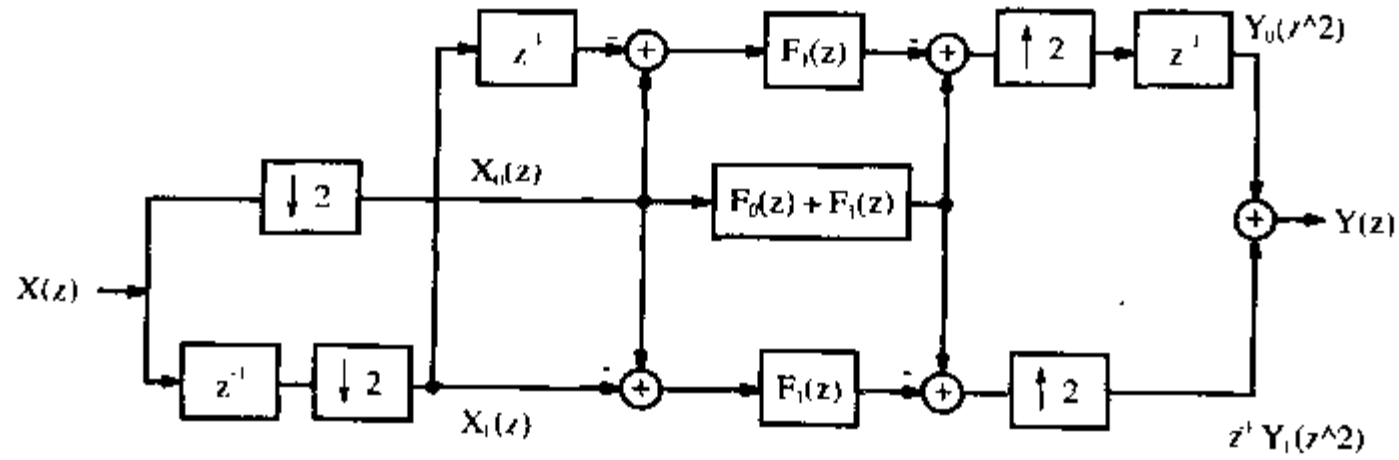


图 5-11　$R=2$ 的快行 FIR 滤波器

如果将直接滤波器的实现与快行 FIR 滤波器加以比较，我们就会在硬件效率与加法器和乘法器操作的平均次数之间作出辨别。直接实现需要 L 个乘法器和 $L-1$ 个加法器全速运行。而快行滤波器有 3 个 $L/2$ 长度、半速运行的滤波器，对于整个滤波器而言，每个输出样本有 $3L/4$ 次乘法和 $(2+2)/2+3/2(L/2-1)=3L/4+1/2$ 次加法，也就是说运算量比直接实现要少 25%左右。从实现的角度来看，我们需要 $3L/2$ 个乘法器和 $4+3(L/2-1)=3L/2+1$ 个加法器，也就是说，工作量要比直接实现多大约 50%。图 5-11 的重要特征是快行滤波器的运行速度基本上是直接实现的 2 倍。使用更多数量的分解数 R 可以进一步提高最大吞吐量。处理 R 个作为输入的 f_a 频率的多相信号的一般方法如下：

算法 5.3　快行 FIR 滤波器

(1) 将输入信号分解成 R 个多相信号，利用 A_e 个加法器以 f_a/R 的速率构成 R 个序列。

(2) 用 R 个长度为 L/R 的滤波器对 R 个序列进行滤波。

(3) 用 A_e 次加法计算输出 $Y_k(z)$的多相表达式。用最后的多路复用器生成输出信号 $Y(z)$。

注意：

已经计算过的长度为 L/R 的部分滤波器可以用算法 5.3 再次分解。这样问题就出现了：我们应该在什么时候停止这种反复的分解？Mou 和 Duhamel[84]已经编译了一个表，其目标就是最小化平均运算数量。表 5-1 给出了最优分解。所采用的准则就是最小化乘法和加法运算的总

量，它是基于 MAC 设计的典型。在表 5-1 中，在所有基于算法 5.3 实现的部分滤波器下面都加了下划线。

表 5-1　递归 FIR 分解的计算量[84, 85]

L	因子	$M+A$	$\frac{M+A}{L}$	L	因子	$M+A$	$\frac{M+A}{L}$
2	直接	6	3	22	11×2	668	30.4
3	直接	15	5	24	2^2×3×2	624	26
4	2×2	26	6.5	25	5×5	740	29.6
5	直接	45	9	26	13×2	750	28.6
6	3×2	56	9.33	27	3^2×3	810	30
8	2^2×2	94	11.75	30	5×3×2	912	30.4
9	3×3	120	13.33	32	2^4×2	1006	31.44
10	5×2	152	15.2	33	11×3	1248	37.8
12	2×3×2	192	16	35	7×5	1405	40.1
14	7×2	310	22.1	36	2^2×3×3	1260	35
15	5×3	300	20	39	13×3	1419	36.4
16	2^3×2	314	19.63	55	11×5	2900	52.7
18	2×3×3	396	22	60	5×2×3×2	2784	46.4
20	5×2×2	472	23.6	65	13×5	3345	51.46
21	7×3	591	28.1				

对于长度大于 60 的快速卷积，使用 FFT 会更加有效，我们将在第 6 章加以讨论。

5.3　Hogenauer CIC 滤波器

高分解速率滤波器的一种非常有效的结构就是由 Hogenauer[86] 引入的“**级联积分器梳状** (cascade integrator comb, CIC)” **滤波器**。CIC 滤波器(也称为 Hogenauer 滤波器)已经被证明是在高速抽取或插值系统中非常有效的单元。一种应用就是无线通信，其中以 RF(Radio Frequency，射频)或者 IF(Intermediate Frequency，中频)为采样速率的信号需要降低到基带。在窄带应用中(例如：蜂窝式无线电通信)，抽取比率经常需要超过 1000。这样的系统通常称作**信道器**[87]。另一个应用领域就是∑Δ数据转换[88]。

CIC 滤波器的基础是完美的极点零点抵消，要实现这样一个事实，只有使用精确的积分算法才是惟一可行的。二进制补码和余数系统都具有支持无误差算法的能力。在二进制补码情形中，算法是以模 2^b 执行的，而在余数系统中算法是以模 M 执行的。

接下来要给出一个介绍性的案例研究加以说明。

5.3.1　单级 CIC 案例研究

图 5-12 给出了一个无抽取 4 位运算的一阶 CIC 滤波器。这个滤波器包括一个(递归)积分器(I 部分)，接下来是一个 4 位微分器或梳状部分(C 部分)。这个滤波器是用 4 位数值以二进制补码算法实现的，数值边界是 $-8_{10}=1000_{2C}$ 和 $7_{10}=0111_{2C}$。

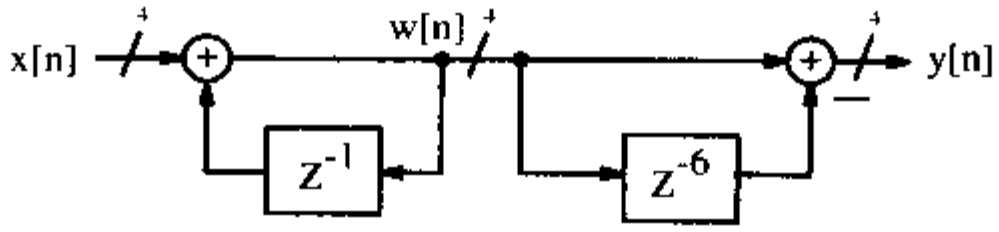


图 5-12　4 位运算中的位移平均值

图 5-13 给出了滤波器的脉冲响应。尽管滤波器是递归的，但是脉冲响应还是有限的，也就是说它是一个递归式 FIR 滤波器。这是不正常的，因为我们一般都希望递归式滤波器是 IIR 滤波器。脉冲响应给出了滤波器计算的和

$$y[n]=\sum_{k=0}^{D-1}x[n-k] \tag{5.20}$$

其中 D 是梳状部分的延迟。滤波器的响应是一个定义在 D 个连续采样值之上的位移平均值。这样一个位移平均值是低通滤波器的一个非常简单的形式。相同位移平均值的滤波器是作为非递归 FIR 滤波器实现的，需要 5 个加法器，而 CIC 设计只需要 1 个加法器和 1 个减法器。

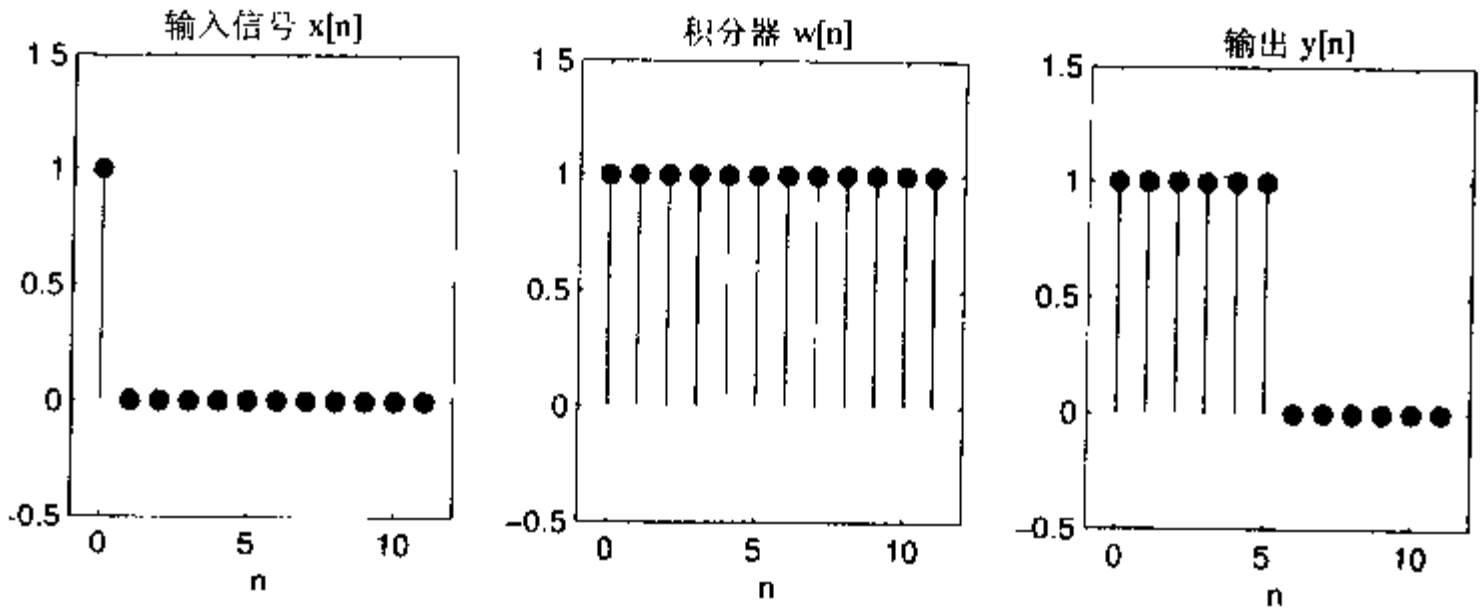


图 5-13　图 5-12 的滤波器的脉冲响应

已知一个极点位置的递归滤波器，当输入是一个极点与递归式滤波器极点直接重合的“本征频率”信号时，该滤波器具有最大稳定状态的正弦输出信号。对于 CIC 部分，本征频率对应频率 $\omega=0$，也就是阶跃信号输入。(5.20)式给出的一阶平均值的阶跃响应是前 D 次采样的一个斜坡，此后是一个常数 $y[n]=D=6$，如图 5-14 所示。注意：尽管积分器 $w[n]$显示了频繁的溢出，但输出还是正确的。这是因为梳状减法也采用的是二进制补码算法。例如：在第一次回绕时，实际的积分器信号是 $w[n]=-8_{10}=1000_{2C}$，延迟信号是 $w[n-6]=2_{10}=0010_{2C}$。这样的话，正像所希望的：$y[n]=-8_{10}-2_{10}=1000_{2C}-0010_{2C}=0110_{2C}=6_{10}$。累加器会继续向上计数，直到再一

次达到 $w[n]=-8_{10}=1000_{2C}$。这样的特性会继续下去，直到给出阶跃输入为止。实际上只要输出 $y[n]$是一个有效的 4 位二进制补码数，并且在[－8，7]范围内，二进制补码系统的精确算法就会自动地对积分器溢出作出补偿。

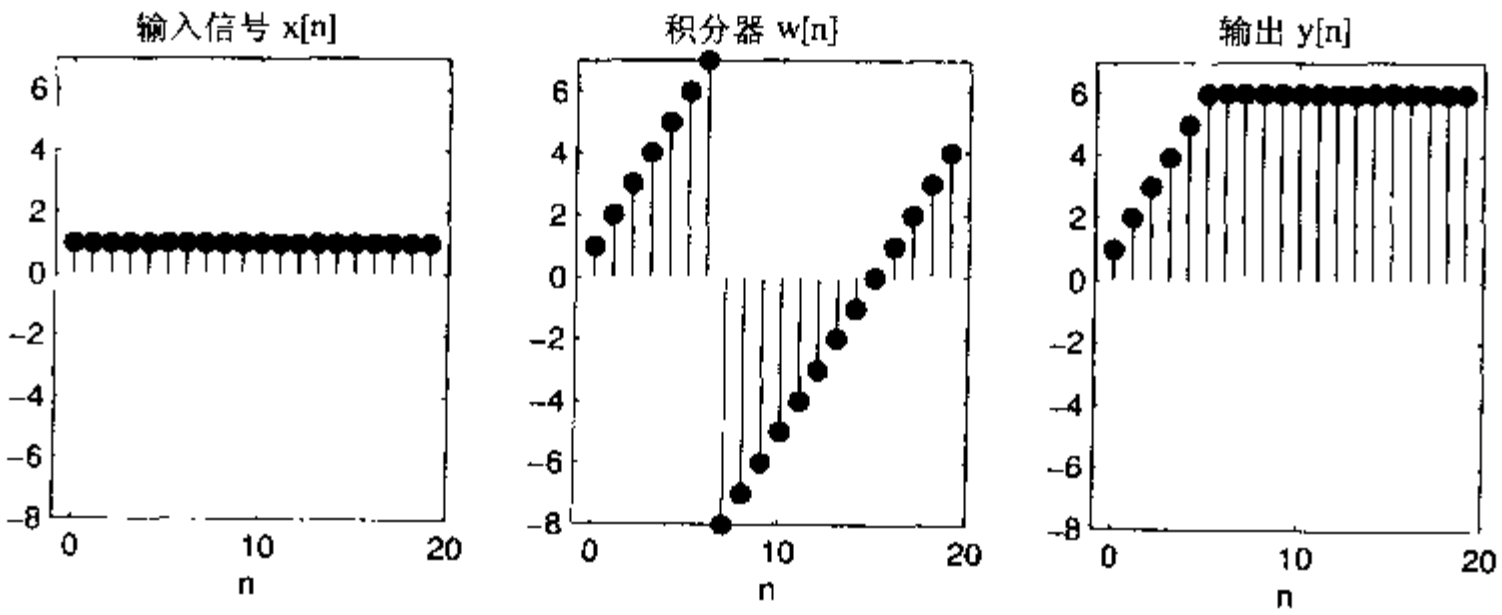


图 5-14　图 5-12 的滤波器的阶跃响应(本征频率测试)

一般情况下，4 位滤波器的宽度对典型应用而言通常是太小了。例如：Harris IC HSP43220 有 5 级，使用了 66 位积分器宽度。为了降低加法器的等待时间应该合理地使用多库 RNS 系统。例如：我们采用集合 $Z_{30}=\{2, 3, 5\}$，就可以从表 5-2 中看到总共可以表示 2·3·5＝30 个惟一值。映射是惟一的(双映射)，并且可以被中国余数定理证明。

表 5-2　集合{2,3,5}的 RNS 映射

$a=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a mod 2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
a mod 3	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
a mod 5	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0
$a=$	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
a mod 2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
a mod 3	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	
a mod 5	1	2	3	4	0	1	2	3	4	0	1	2	3	4	0	

图 5-15 给出了说明 RNS 实现的阶跃响应。滤波器的输出 $y[n]$已经用表 5-2 中的数据重新构建了。输出响应与在二进制补码情况下获得的采样值相同(请参阅图 5-14)。保持结构的映射称为**同态象**，双映射同态象就称为**同构**(符号≅)，可以表示成：

$$Z_{30}\cong Z_2\times Z_3\times Z_5 \tag{5.21}$$

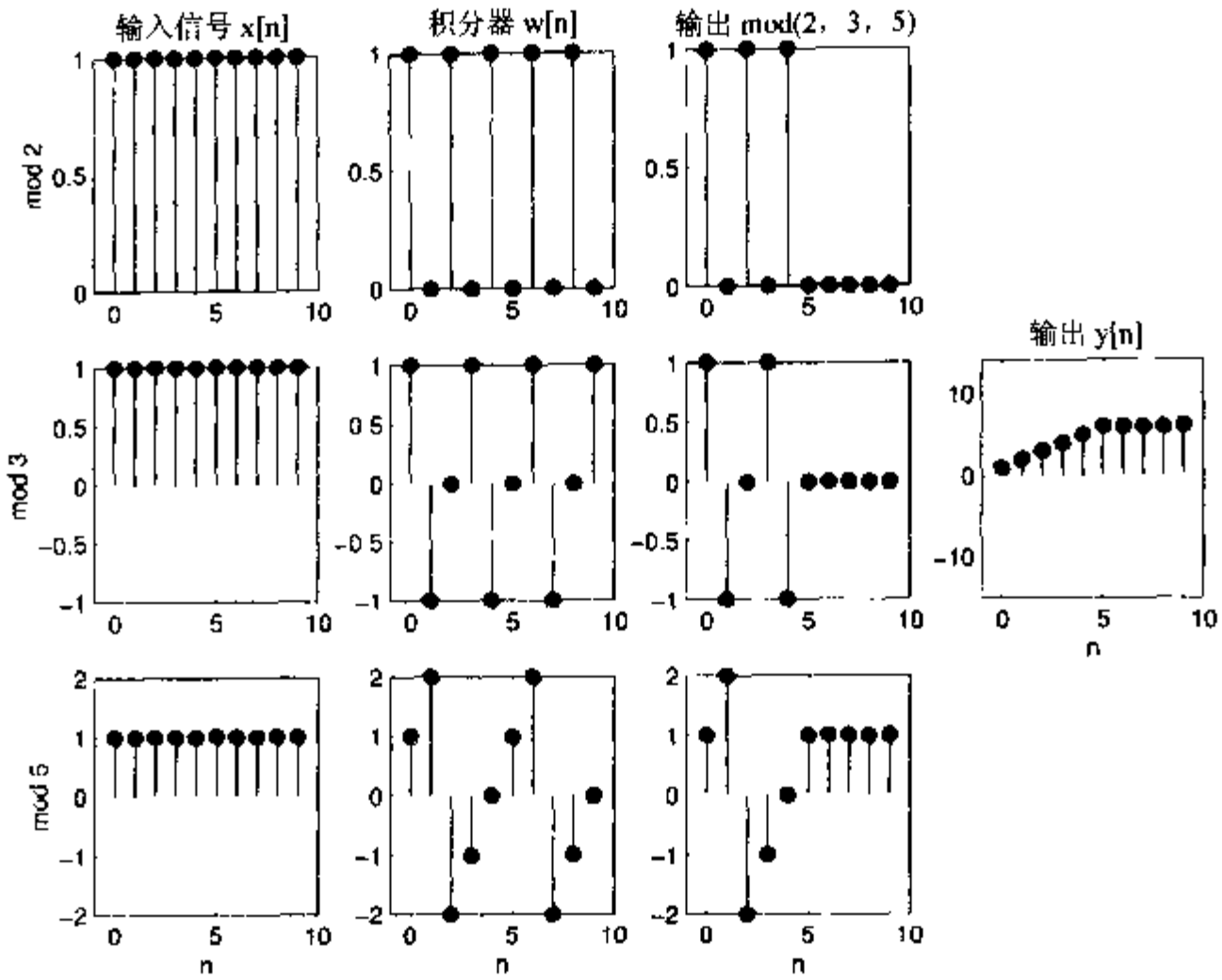


图 5-15　RNS 算法中一阶 CIC 的阶跃响应

5.3.2　多级 CIC 滤波器理论

一般包括 S 个级的 CIC 系统的传递函数如下：

$$F(z)=\left(\frac{1-z^{-RD}}{1-z^{-1}}\right)^{S} \tag{5.22}$$

其中 D 是梳状部分中延迟的数量，R 是向下采样(抽取)因子。从(5.22)式中可以看到，$F(z)$ 有 RDS 个零点和 S 个极点，RD 个零点是由数字项$(1-z^{-RD})$产生的，处于 $2\pi/(RD)$弧度处，圆心起始于 $z=1$。每个不同的零点都重复出现 S 次。$F(z)$的 S 个极点位于 $z=1$ 处，也就是说位于零频率(DC)位置。可以立即就看到这些极点已经被 CIC 滤波器的 S 个零点抵消掉了。进而得到 1 个 S 阶位移平均滤波器。最大动态范围增长出现在 DC 频率(也就是 $z=1$)。最大动态范围的增长是：

$$B_{\text{grow}}=(RD)^{S} \text{ 或 } b_{\text{grow}}=\log_2(B_{\text{grow}}) \text{ 位} \tag{5.23}$$

在设计 CIC 滤波器时，知道这个值是非常重要的，因为单态 CIC 实例需要精确的算法。在实际应用中，最坏情况下的增益是非常重要的，有构造的 66 位动态范围商用滤波器(例如：哈利斯 HSP43220 渠化器)为证，典型的设计都是采用二进制补码算法。

图 5-16 给出了一个三阶 CIC 滤波器，该滤波器包括一个三阶积分器和一个三阶梳状部分，并且采样速率降低了 R 倍。注意：首先要实现所有的积分器，然后是抽取器，最后再实现梳状部分。这种重新安排在梳状部分节省了一个因数 R 的延迟单元。高抽取速率滤波器的延迟数量 D 的典型值是 1 或 2。

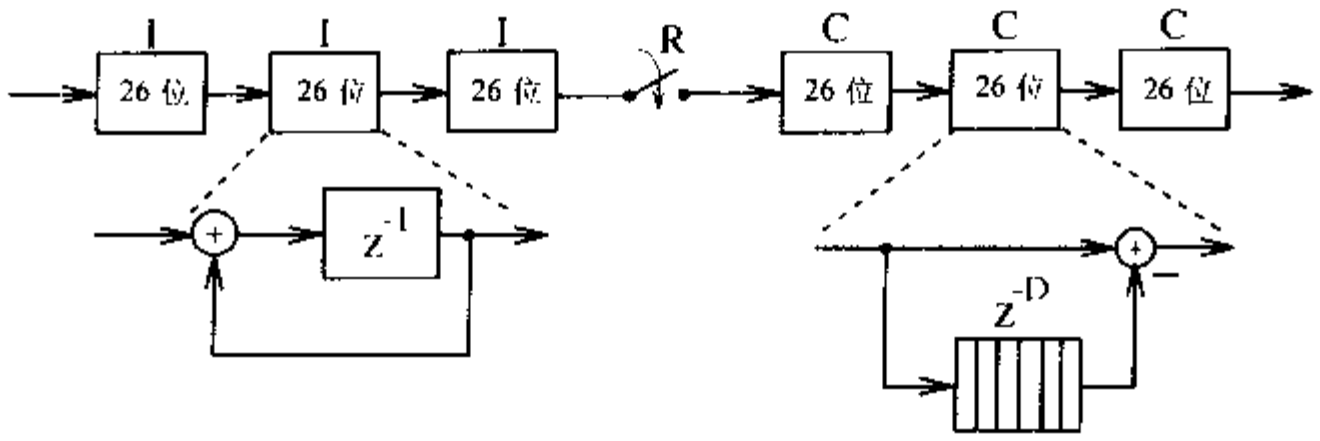


图 5-16　CIC 滤波器，每级 26 位

一个具有 8 位输入字宽的三阶 RNS CIC 滤波器，$D=2$，$R=32$ 或 $DR=2\times32=64$，需要的内部字宽为 $W=8+3\log_2(64)=26$ 位，保证不会产生运行时间溢出。正常的输出字宽一般都明显地比 W 小，比如 9 位。比方说，如果信号的带宽与采样频率的比率是 1/32，混叠抑制就是 89.6dB，通频带衰减是 0.17[86. & I+II]。

例 5.4　三阶 CIC 抽取器 I

最坏情况下的增益情况就是给 CIC 滤波器输入一个阶跃信号。图 5-17(a)给出了一个幅值为 127 的阶跃输入信号。图 5-17(b)显示了第三个积分器部分的输出。注意观察，运行时间溢出是以一个规则的速率出现的。图 5-17(c) 显示的 CIC 输出是以输入采样的速率被插入(光滑)的。图 5-17(d)的输出已经被刻度成 9 位精度，以抽取采样的速率显示。

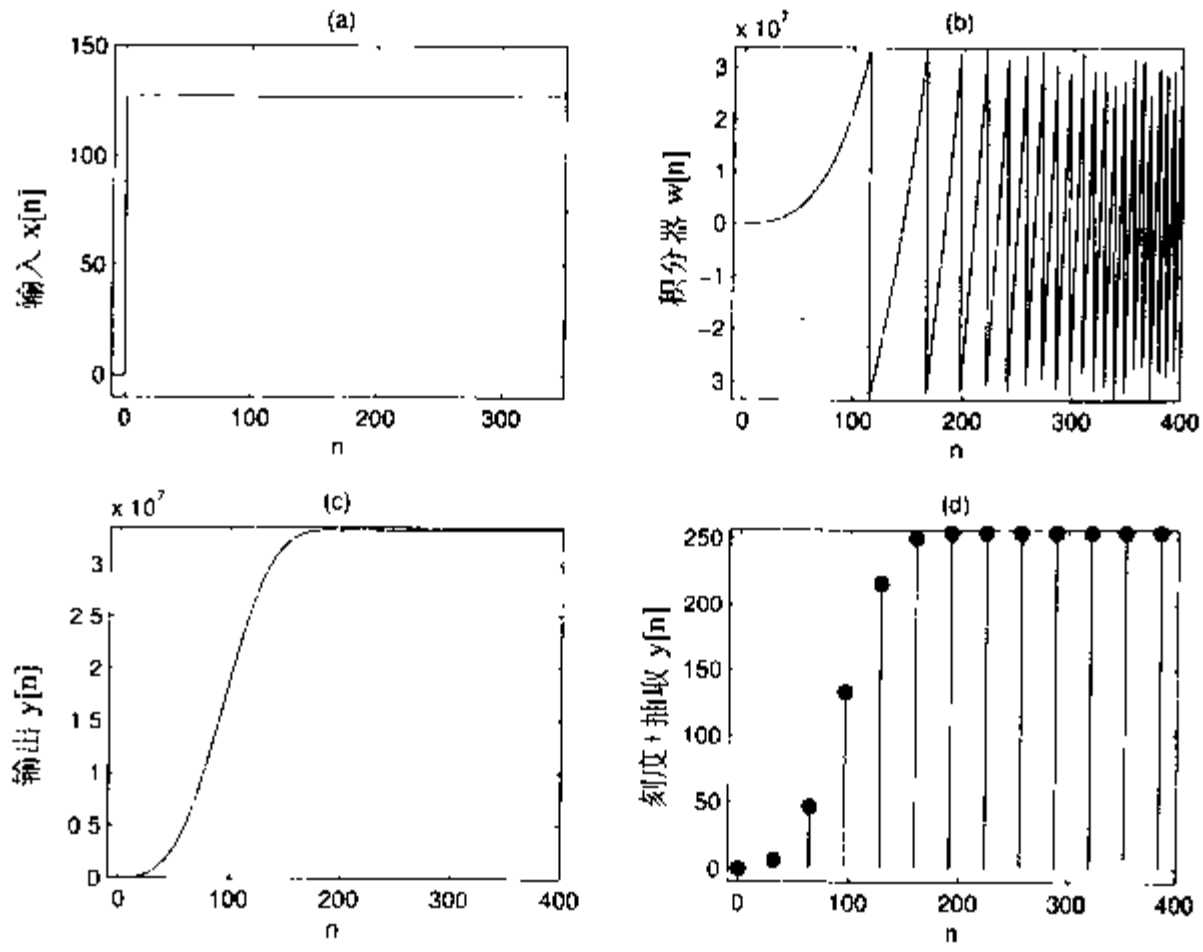


图 5-17　图 5-16 中的 3 阶 CIC 滤波器的仿真

下面的 VHDL 代码[5]给出了 CIC 实例的设计。

```
PACKAGE n_bit_int IS      -- User defined types
```

注 5：这一例子相应的 Verilog 代码文件 cic3r32.v 可以在附录 A 中找到。

```
    SUBTYPE word26 IS INTEGER RANGE 0 TO 2**26 - 1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY cic3r32 IS
        PORT ( clk   :    IN   STD_LOGIC;
                 x_in :    IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                y_out :    OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END cic3r32;

ARCHITECTURE flex OF cic3r32 IS
  TYPE     STATE_TYPE IS (hold, sample);
  SIGNAL   state     : STATE_TYPE ;
  SIGNAL   count     : integer RANGE 0 TO 31;
  SIGNAL   clk2      : STD_LOGIC;
  SIGNAL   x : STD_LOGIC_VECTOR(7 DOWNTO 0);
                                          -- Registered input
  SIGNAL   sxtx : STD_LOGIC_VECTOR(25 DOWNTO 0);
                                          -- Sign extended input
  SIGNAL   i0, i1 , i2 : word26;     -- I section   0, 1, and 2
  SIGNAL   i2d1, i2d2, i2d3, i2d4, c1, c0 : word26;
                                          -- I and COMB section 0
  SIGNAL   c1d1, c1d2, c1d3, c1d4, c2 : word26;      -- COMB 1
  SIGNAL   c2d1, c2d2, c2d3, c2d4, c3 : word26;      -- COMB 2

BEGIN

  FSM: PROCESS
  BEGIN
    WAIT UNTIL clk = '0';
    CASE state IS
      WHEN hold =>
        IF count < 31 THEN
            state <= hold;
        ELSE
            state <= sample;
        END IF;
      WHEN OTHERS =>
```

```
        state <= hold;
    END CASE;
END PROCESS FSM;

sxt: PROCESS (x)
BEGIN
    sxtx(7 DOWNTO 0) <= x;
    FOR k IN 25 DOWNTO 8 LOOP
        sxtx(k) <= x(x'high);
    END LOOP;
END PROCESS sxt;

Int: PROCESS
BEGIN
    WAIT UNTIL clk = '1';
        x    <= x_in;
        i0   <= i0 + CONV_INTEGER(sxtx);
        i1   <= i1 + i0 ;
        i2   <= i2 + i1 ;
    CASE state IS
        WHEN sample =>
            c0       <= i2;
            count    <= 0;
        WHEN OTHERS =>
            count    <= count + 1;
    END CASE;
    IF (count > 8) and (count <16) THEN
        clk2  <= '1';
    ELSE
        clk2  <= '0';
    END IF;
END PROCESS Int;

Comb: PROCESS
BEGIN
    WAIT UNTIL clk2 = '1';
        i2d1  <= c0;
        i2d2  <= i2d1;
        c1    <= c0 - i2d2;
        c1d1  <= c1;
        c1d2  <= c1d1;
        c2    <= c1   - c1d2;
        c2d1  <= c2;
        c2d2  <= c2d1;
        c3    <= c2   - c2d2;
```

```
    END PROCESS Comb;

    y_out <= CONV_STD_LOGIC_VECTOR(c3 / 2**17 , 9);

END flex;
```

设计的滤波器包括一个有限状态机(Finite State Machine，FSM)；符号扩展 sxt：PROCESS 和两个“算法”PROCESS 模块。Int：PROCESS 实现了 3 个积分器和梳状部分的时钟分频器。Comb：PROCESS 包括 3 个梳状滤波器，其中每个都有两次采样的延迟。滤波器运行速度为 38.46MHz，采用了 332 个 LC。注意：如果没有前面的向下采样，滤波器就不能够与目标元器件配合。前面的向下采样节省了 3×32×27＝2592 个寄存器或者 LC！

如果将滤波器的输出(图 5-18 给出 VHDL 输出 y_out)与图 5-17 给出的 MATLAB 仿真结果的响应 $y[n]$加以比较，就会看到滤波器的工作状态正是我们所希望的。

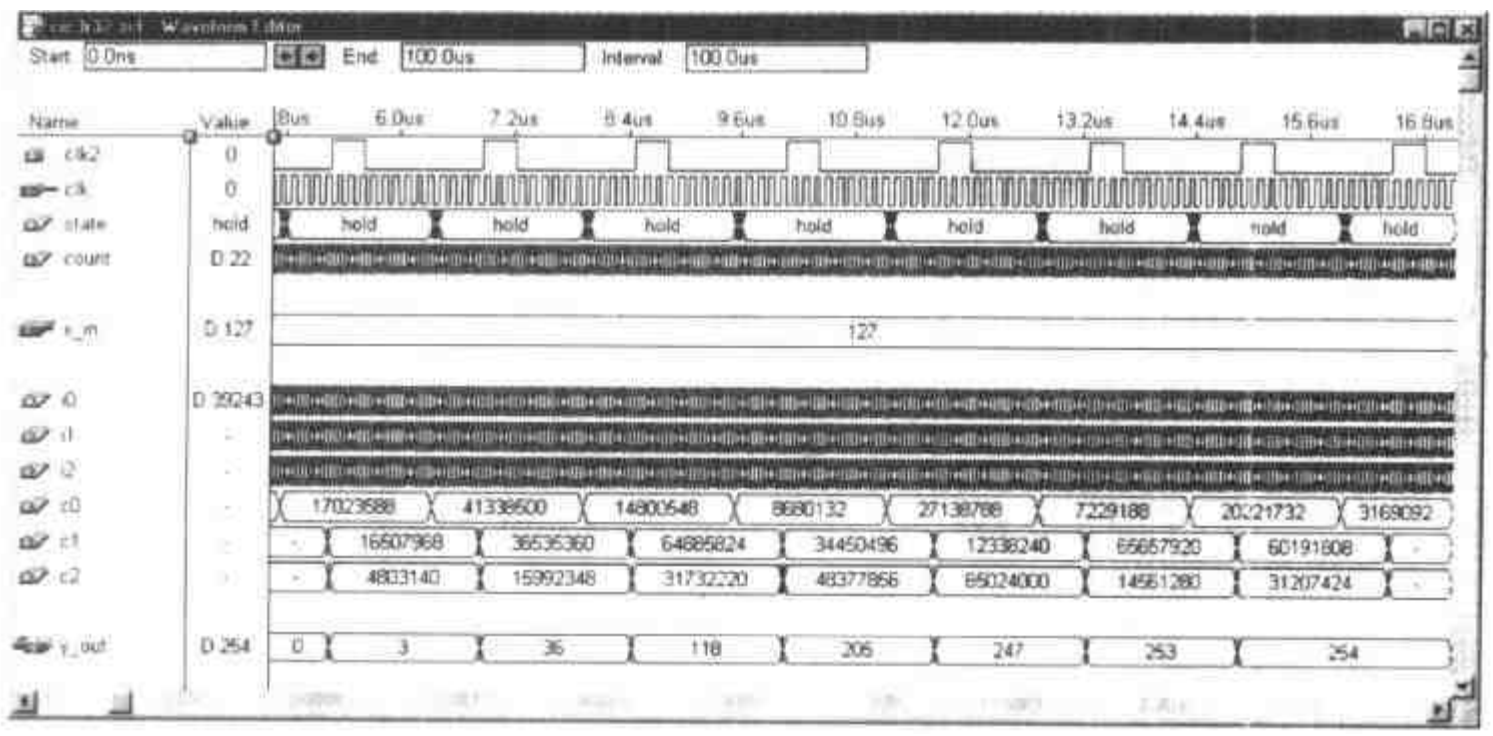


图 5-18　图 5-16 中的 3 阶 CIC 滤波器的 VHDL 仿真

Hogenaur[86]指出：在仔细分析的基础上，来自前级的一些较低有效位可以删掉，而且不影响系统的完整性。图 5-19 给出了一个所有级均采用全字宽(最坏情况)的系统的幅频响应，这个系统同时也应用了 Hogenaur 建议的“剪除”策略。

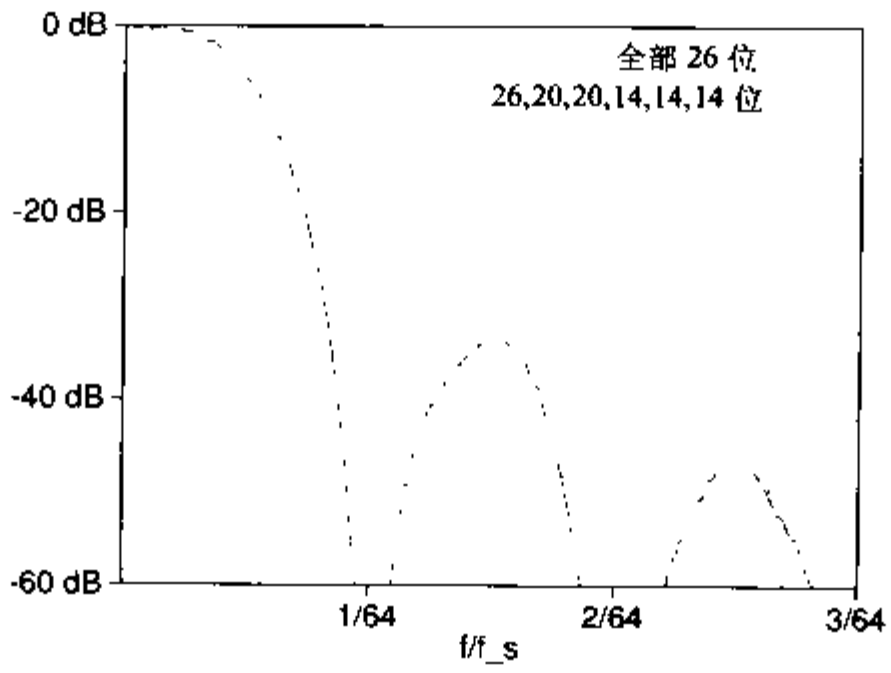


图 5-19　CIC 传递函数(f_s 是输入端的采样频率)

5.3.3　幅值与混叠畸变

S 阶 CIC 滤波器的传递函数如下：$z=\mathrm{e}^{j2\pi fT}$

$$F(z)=\left(\frac{1-z^{RD}}{1-z^{-1}}\right)^{S} \tag{5.24}$$

在频域内沿着弧 $z=\mathrm{e}^{j2\pi f/T}$ 求 $F(z)$，可以计算幅值畸变和最大混叠的成份。幅值响应变成：

$$|F(f)|=\left(\frac{\sin(2\pi fTRD/2)}{\sin(2\pi fT/2)}\right)^{S} \tag{5.25}$$

这一公式可以用来直接计算通频带边缘 ω_p 处的幅值畸变。图 5-20 给出了 $R=3$、$D=2$ 和 $RD=6$ 的三阶 CIC 滤波器的$|F(f-k/(2R))|$。观察 CIC 滤波器低频响应在通频带发生混叠的几个副本。

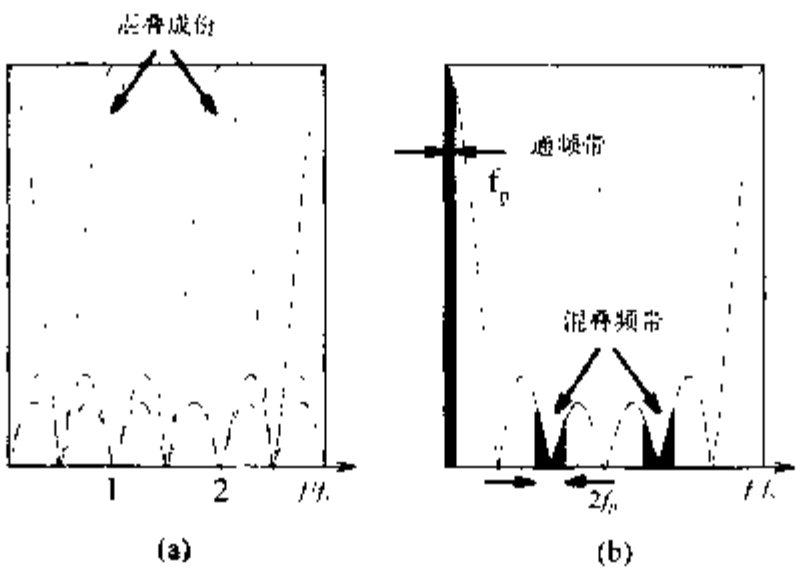


图 5-20　3 阶 CIC 抽取器的传递函数。注意：f_s 是低速的采样频率

可以看到在频率：

$$f|_{\text{最大混叠}}=1/(2R)-f_p \tag{5.26}$$

的最大混叠成份可以根据$|F(f)|$计算。一般情况下，只考虑第一个混叠成份，因为第二个混叠成份非常小。图 5-21 给出了在频率 f_p 处 $f_p/(Df_s)$不同比率下的幅值畸变。

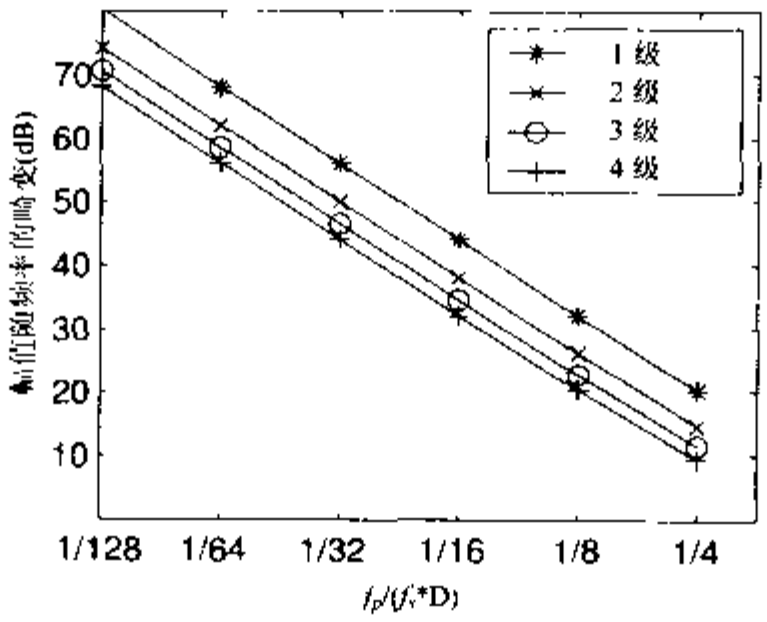


图 5-21　CIC 抽取器的幅值畸变

图 5-22 给出了具体通频带频率与采样频率之间的比率 f_p/f_s 在不同 S、R 和 D 值的情况下的最大混叠成份。

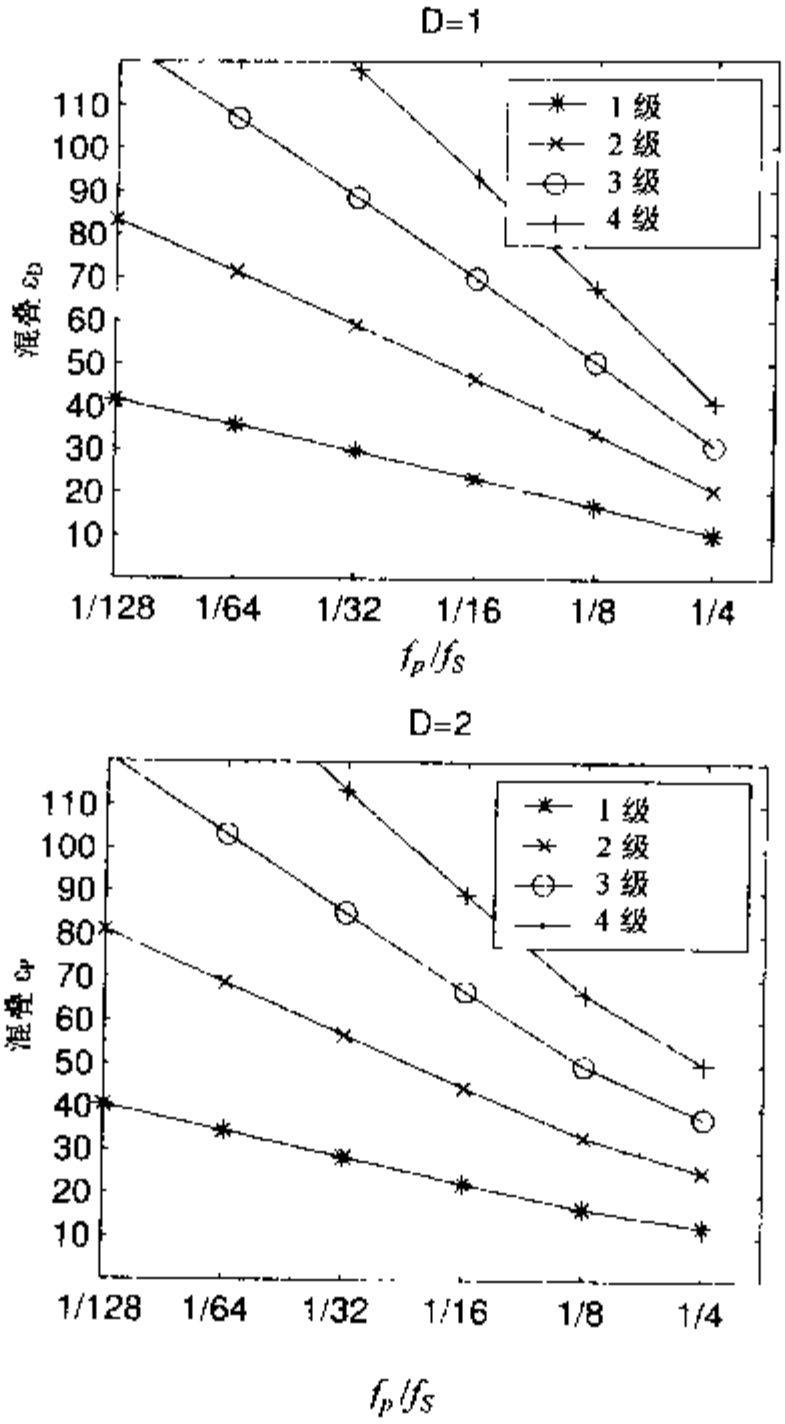


图 5-22　从 1 级到 4 级 CIC 抽取器的最大混叠成份

可以认为幅值畸变能够通过级联的 FIR 补偿滤波器得到校正，这种补偿滤波器在通频带的传递函数是 $1/F(z)$，但是混叠畸变是不能够被修复的。这样一来，通常可以接受的混叠畸变就成了主要的设计参数。

5.3.4　Hogenaur“剪除”理论

总内部位宽可以解释为输入字宽与最大动态增长需要(5.23)的和或用代数形式表示成：

$$B_{\text{intern}}=B_{\text{input}}+B_{\text{growth}} \tag{5.27}$$

如果要设计的滤波器是用在所有级别上的这一位宽执行精确的算法，那么在输出端就会产生非运行时间溢出。一般来讲，CIC 滤波器的输入和输出位宽都处于同一范围内。我们可以看到通过剪除在输出引入的量化通常要大于在前级剪除一些最低有效位带来的量化。如果 $\sigma^2_{T,2S+1}$ 是在输出通过剪除引入的量化噪声，则 Hogenauer 提出设其等于前面所有部分引入的噪声 σ^2_k 之和。对于包含 S 个积分器和 S 个梳状部分的 CIC 滤波器而言，就是：

$$\sum_{k=1}^{2S}\sigma_{T,k}^2=\sum_{k=1}^{2N}\sigma_k^2P_k^2\le\sigma_{T,2S+1}^2 \tag{5.28}$$

$$\sigma_{T,k}^2=\frac{1}{2S}\,\sigma_{T,2S+1}^2 \tag{5.29}$$

$$P_k^2=\sum_n(h_k[n])^2\quad k=1,2,\ldots2S \tag{5.30}$$

其中 P_k^2 是从第 k 级到输出的功率增益。计算下一个数时，被剪除位 B_k 是：

$$B_k=\left\lfloor 0{,}5\log_2(P_k^{-2}\cdot\frac{6}{N}\cdot\sigma_{T,2S+1}^2)\right\rfloor \tag{5.31}$$

$$\sigma_{T,k}^2\Big|_{j=2N-1}=\frac{1}{12}2^{2B_k}=\frac{1}{12}2^{2(B_{\text{in}}-B_{\text{out}}+B_{\text{growth}})} \tag{5.32}$$

梳状部分的功率增益 P_k^2，$k=1$，2，…，S，可以采用下面的二项式系数计算：

$$H_k(z)=\sum_{m=0}^{2S+1-k}(-1)^n\binom{2S+1-k}{n}z^{-kRD}\qquad k=S,S+1,\ldots,2S \tag{5.33}$$

计算第一项因数 P_k^2，$k=1$，2，…S 时，要注意每个积分器/梳状部分对都会生成一个有限(位移平均数)的脉冲响应。这样，第 k 级的最终系统就是一系列 $S-k+1$ 个积分器/梳状部分对，后面跟有 $k-1$ 个梳状部分。图 5-23 给出简化计算 P_k^2 的这一重新安排。

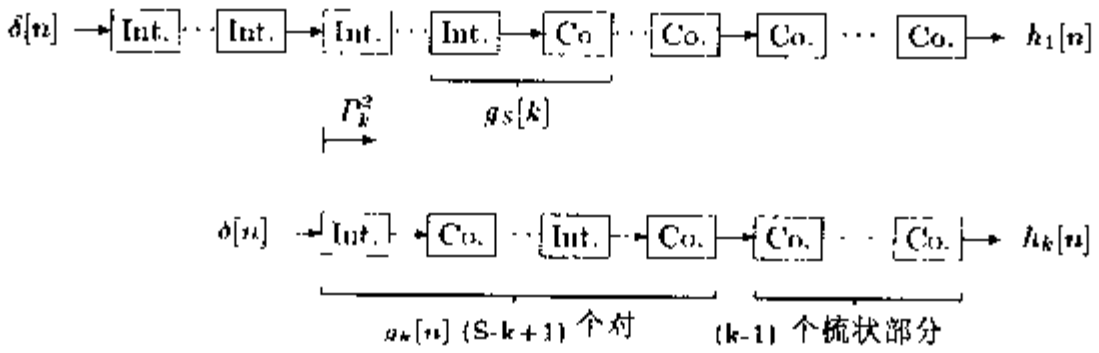


图 5-23　重构简化 P_k^2 的计算(©1995VDI 出版社[4])

例 5.5　三阶 CIC 抽取器 II

Garcia 等人已经设计了一种采用三阶 CIC 滤波器的 I/Q 解调器。抽取器的行数据是：$B_{\text{input}}=8$，$B_{\text{output}}=9$，位 $R=32$，$D=2$。很明显，位增长是：

$$B_{\text{growth}}=\left\lceil\log_2(RD^S)\right\rceil=\log_2(64^3)\lceil3\cdot6\rceil=18 \tag{5.34}$$

内部总位宽就是

$$B_{\text{intern}}=B_{\text{input}}+B_{\text{growth}}=8+18=26 \tag{5.35}$$

程序 cic.exe 给出了下面的结果：

```
-- ------------------------------------------------------------
-- Program for the design of a CIC decimator
-- ------------------------------------------------------------
--      Input bir width          Bin  =     8
--      Output bit width         Bout =     9
```

```
--        Number of stages          S    =     3
--        Decimation factor         R    =    32
--        COMB delay                D    =     2
--        Frequency resolution      DR   =    64
--        Passband freq. ratio      P    =     4
-- ---------------------------------------------------------------------
-- ---------------- Rseults of the Design ------------------------------
-- ---------------------------------------------------------------------
-- ------- Computed bit width:
-- Maximum bit growth over all stages                    =    18
-- Maximum bit width including sign Bmax+1               =    26
-- Stage    1 INTEGRATOR. Bit width  : 26
-- Stage    2 INTEGRATOR. Bit width  : 21
-- Stage    3 INTEGRATOR. Bit width  : 16
-- Stage    1 COMB.       Bit width  : 14
-- Stage    2 COMB.       Bit width  : 13
-- Stage    3 COMB.       Bit width  : 12
-- ------- Maximum aliasing component  : 0.009682 = 40.37   dB
-- ------- Amplitude distortion        : 0.258012 = 11.77   dB
```

图 5-21 和图 5-22 给出的设计图表也可以用来计算最大混叠成份和幅值畸变。下面的设计示例说明了位宽设计的详细细节，使用的软件是 MaxPlusII。

例 5.6　三阶 CIC 抽取器 III

设计所采用的数据与例 5.4.中的数据相同。但是现在考虑了例 5.5 中所计算的剪除。下面的 VHDL 代码[6]给出了带有剪除的 CIC 示例设计。

```
PACKAGE n_bit_int IS                -- User defined types
  SUBTYPE word26 IS INTEGER RANGE - 2**25 TO 2**25 - 1;
  SUBTYPE word21 IS INTEGER RANGE - 2**20 TO 2**20 - 1;
  SUBTYPE word16 IS INTEGER RANGE - 2**15 TO 2**15 - 1;
  SUBTYPE word14 IS INTEGER RANGE - 2**14 TO 2**14 - 1;
  SUBTYPE word13 IS INTEGER RANGE - 2**13 TO 2**13 - 1;
  SUBTYPE word12 IS INTEGER RANGE - 2**12 TO 2**12 - 1;
END n_bit_int;

LIBRARY work;
USE work.n_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

注 6：这一例子相应的 Verilog 代码文件 cic3s32.v 可以在附录 A 中找到。

```
USE ieee.std_logic_unsigned.ALL;


ENTITY cic3s32 IS
  PORT ( clk    :  IN   STD_LOGIC;
         x_in   :  IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
         y_out  :  OUT STD_LOGIC_VECTOR(8 DOWNTO 0));
END cic3s32;

ARCHITECTURE flex OF cic3s32 IS
  TYPE     STATE_TYPE IS (hold, sample);
  SIGNAL   state      : STATE_TYPE ;
  SIGNAL   count      : integer RANGE 0 TO 31;
  SIGNAL   clk2       : STD_LOGIC;
  SIGNAL   x          : STD_LOGIC_VECTOR(7 DOWNTO 0);
                                          -- Registered input
  SIGNAL   sxtx : STD_LOGIC_VECTOR(25 DOWNTO 0);
                                        -- Sign extended input
  SIGNAL   i0 :   word26;                       -- I section 0
  SIGNAL   i1 :   word21;                       -- I section 1
  SIGNAL   i2 :   word16;                       -- I section 2
  SIGNAL   i2d1, i2d2, i2d3, i2d4, c1, c0 : word14;
                                        -- I and COMB section 0
  SIGNAL   c1d1, c1d2, c1d3, c1d4, c2 : word13;    -- COMB   1
  SIGNAL   c2d1, c2d2, c2d3, c2d4, c3 : word12;    -- COMB   2

BEGIN

  FSM: PROCESS
  BEGIN
    WAIT UNTIL clk = '0';
    CASE state IS
      WHEN hold =>
        IF count < 31 THEN
            state <= hold;
        ELSE
            state <= sample;
        END IF;
      WHEN OTHERS =>
        state <= hold;
    END CASE;
  END PROCESS FSM;
```

```
Sxt: PROCESS (x)
BEGIN
  sxtx(7 DOWNTO 0) <= x;
  FOR k IN 25 DOWNTO 8 LOOP
    sxtx(k) <= x(x'high);
  END LOOP;
END PROCESS Sxt;

Int: PROCESS
BEGIN
WAIT
  UNTIL clk = '1';
    x    <= x_in;
    i0   <= i0 + CONV_INTEGER(sxtx);
    i1   <= i1 + i0 / 32;
    i2   <= i2 + i1 / 32;
  CASE state IS
    WHEN sample =>
      c0    <= i2 /4;
      count <= 0;
    WHEN OTHERS =>
      count <= count + 1;
  END CASE;
  IF (count > 8) and (count <16) THEN
    clk2   <= '1';
  ELSE
    clk2   <= '0';
  END IF;
END PROCESS Int;

Comb: PROCESS
BEGIN
  WAIT UNTIL clk2 = '1';
    i2d1  <= c0;
    i2d2  <= i2d1;
    c1    <= c0 - i2d2;
    c1d1  <= c1 / 2;
    c1d2  <= c1d1;
    c2    <= c1 / 2 - c1d2;
    c2d1  <= c2 / 2;
    c2d2  <= c2d1;
```

```
        c3    <= c2 / 2 - c2d2;
    END PROCESS Comb;

    y_out <= CONV_STD_LOGIC_VECTOR(c3 / 8, 9);

END flex;
```

这一设计具有与例 5.4 中的 CIC 相同的结构，包括一个有限状态机(finite state machine, FSM)，符号位扩展 sxt：PROCESS 和两个“算法”PROCESS 模块。Int：PROCESS 实现 3 个积分器和梳状部分的时钟分频器。Comb：PROCESS 包括 3 个梳状部分，每个梳状部分都有两个延迟。但是所有的积分器和梳状部分都设计成带有 Hogenaur 提出的剪除技术的位宽。这样就将设计规模降低到 199 个 LC，运行速度是 41.66MHz。

这样的设计将速度从 38MHz 提高到 41MHz，同时也节省了 133 个 LC，与例 5.4 中的设计相比也就是节省了 40%的 LC。与图 5-24 和图 5-18 中给出的 VHDL 仿真结果的滤波器输出相比，需要注意不同的最低有效位的量化效应(请参阅练习 5.11)。在剪除式设计中，“噪声”具有 LSB 的渐近线形式。

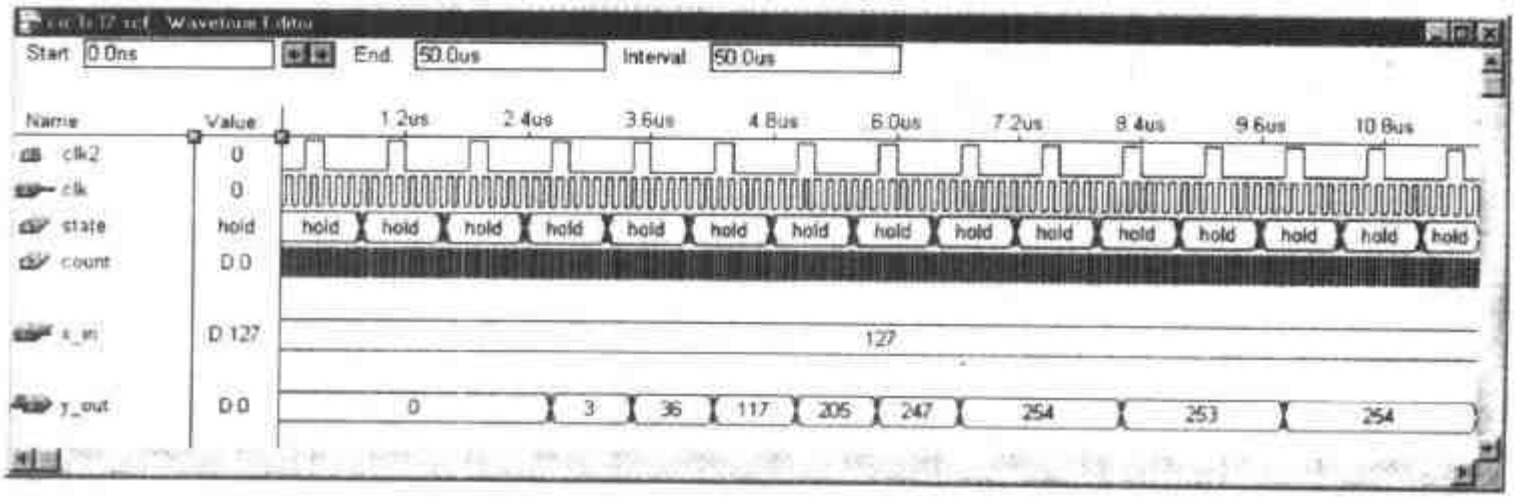


图 5-24　采用位剪除的三级 CIC 滤波器的 VHDL 仿真

5.3.5　CIC RNS 设计

采用 RNS 的 CIC 滤波器的设计应该归功于 Garcia、Meyer Baese 和 Taylor[44]。他们实现了具有 8 位输入、10 位输出、$D=2$ 和 $R=32$ 的三阶 CIC 滤波器。最大字宽是 26 位。对于 RNS 的实现，4-模集合{256，63，61，59}，也就是一个 8 位二进制补码和 3 个 6 位模，覆盖了这一范围(请参阅图 5-25)。输出采用 ε-CRT 刻度，为了实现所有的 5 个模加法器和 9 个 ROM 表或 7 个表(如果乘法逆运算 ROM 和 ε-CRT 是组合的)，需要 8 个表和 3 个二进制加法器[39, 图 1]或者(如图 5-26 所示)采用基于两个 6 位模的基本迁移刻度(base removal scaling, BRS)算法和余下两个模的 ε-CRT。下面的表给出了在 MSPS(Multiphase Serial-Parallel-Serial Storage，多相串行-并行-串行存储器)中的速度和 3 个刻度结构所使用的 LE 和 EAB 的数量。

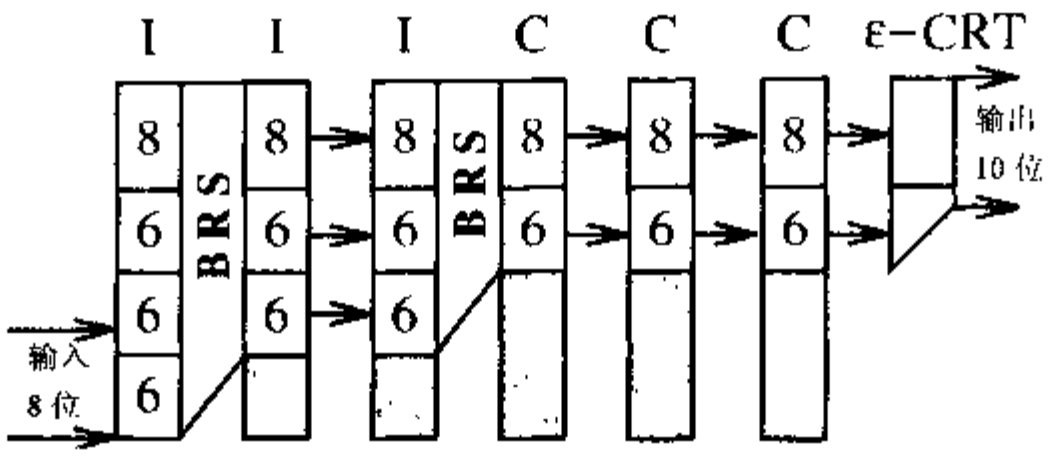


图 5-25　CIC 滤波器。基本迁移刻度(base removal scaling, BRS)的详尽设计

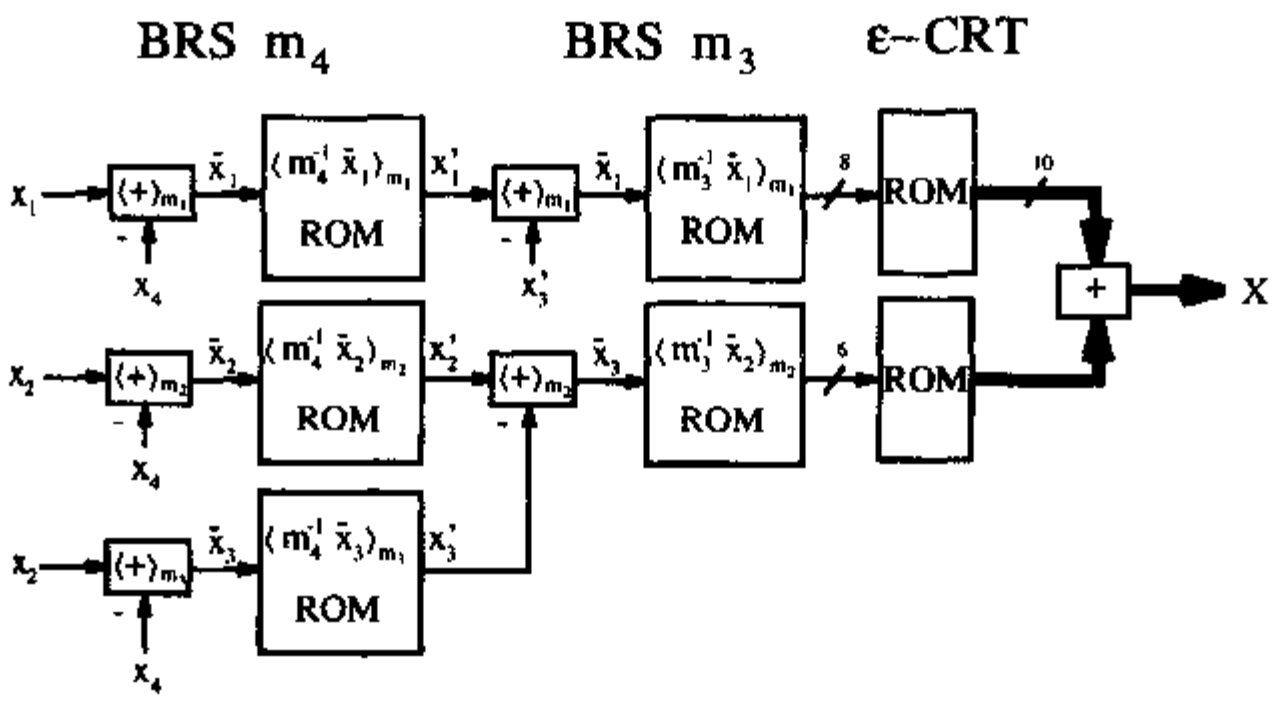


图 5-26　BRS 和 ε-CRT 转换步骤

类　　型	ε-CRT	(只适用于 BRS m_4 的速度数据)BRS ε-CRT	与 ROM 组合的 BRS ε-CRT
MSPS	58.8	70.4	58.8
#LE	34	87	87
#Table(EAB)	8	9	7

刻度结构 1 和 3 在速度方面降低到 58.8 个 MSPS，这就是需要 10 位 ε-CRT 的原因。需要注意的是：这并没有降低系统的速度，因为刻度是应用在较低的(输出)采样速率上。对于 BRS ε-CRT 而言，假定只有 BRS m_4 部分(请参阅图 5-13)必须以输入采样速率运行，而 BRS m_3 和 ε-CRT 以输出采样速率运行。

如果采用的刻度结构与例 5.5 和图 5-25 中介绍的相似，就可以节省一些资源。采用这种结构的话，在滤波器的前面部分就必须采用 BRS ε-CRT 结构来降低位宽。ROM 的较早使用将可能的吞吐量从 76.3 降低到 70.4MSPS，后者是 BRS m_4 的最大速度。在输出部分采用了高效的 ε-CRT 结构。

下面的表总结了滤波器设计的 3 种实现，这里没有包括刻度数据。

类型	2C 26 位	RNS 8，6，6，6 位	详细位宽 RNS 设计
MSPS	49.3	76.3	70.4
#LEs	343	559	355

5.4　多级抽取器

如果抽取的速率 R 较大，就会发现与单级转换器的实现相比，用较少的工作量就可以实现多级设计。特别是 S 级中每一级都具有 R_k 的抽取能力，总的向下采样速率是 $R=R_1R_2...R_S$。可惜的是，通频带不是完美的，例如纹波偏差从一级到下一级的积累，这样就导致 ε_p 的通频带偏差目标必须固定在 $\varepsilon'_p=\varepsilon_p/S$ 阶上，以满足整体设计规范。这是一种明显的最坏情况假设，其中所有的短滤波器在同一频率上都具有最大的纹波，一般情况下这也是最不利的。通常尝试使用接近给定通频带规定的 ε_p 的初始值会更加合理一些，然后再根据需要有选择地降低它。

使用 Goodman-Carey 半波带滤波器的多级抽取器设计

Goodman 和 Carey 提出[66]，在使用 CIC 和半波带滤波器的基础上设计多级系统。正如其名称所暗示的，半波带滤波器的通频带和抑止频带位于 $\omega_s=\omega_p=\pi/2$，也就是基带的中间。半波带滤波器可以用因数 2 来改变采样速率。如果半波带滤波器具有关于 $\omega=\pi/2$ 的点对称，则所有的偶系数(中心抽头除外)都变成了 0。

定义 5.7：半波带滤波器

半波带滤波器的脉冲响应关于 $k=d$ 对称，遵循下列法则

$$f[k]=0，\ k \text{ 为偶数，但不包括 } k=d \tag{5.36}$$

同样的情况转换到 Z 域内时，就变成

$$F(z)+F(-z)=c^*z^{-d} \tag{5.37}$$

其中 $c\in C$，是一个常数，$d\in N_0$

Goodman 和 Carey 已经编译了一列整数半波带滤波器，可以看到，随着长度的增加，幅值畸变逐渐变小。表 5-3 给出了这些半波带滤波器的系数。为了简化表达式，所有的系数都是以中心抽头位置 $d=0$ 来确定的。F1 是长度为 L 的移动平均滤波器，也就是 Hogenauer CIC 滤波器，而且也可以用在第一级，将速率改变成除了 2 以外的一个因数。图 5-27 给出了 9 种不同滤波器的传递函数。注意：在图 5-27 中的对数图内，不能够观察到对称点(通常的半波带滤波器都有)。

表 5-3　Goodman 和 Carey 给出的半波带滤波器 F1 到 F9 的系数

名　称	L	纹波	f[0]	f[1]	f[3]	f[5]	f[7]	f[9]
F1	3	—	1	1				
F2	3	—	2	1				
F3	7	—	16	9	-1			
F4	7	36dB	32	19	-3			

(续表)

名　称	L	纹波	f[0]	f[1]	f[3]	f[5]	f[7]	f[9]
F5	11	—	256	150	- 25	3		
F6	11	49dB	346	208	- 44	9		
F7	11	77 dB	521	302	- 53	7		
F8	15	65 dB	802	490	- 116	33	- 6	
F9	19	78 dB	8192	5042	- 1277	429	- 116	18

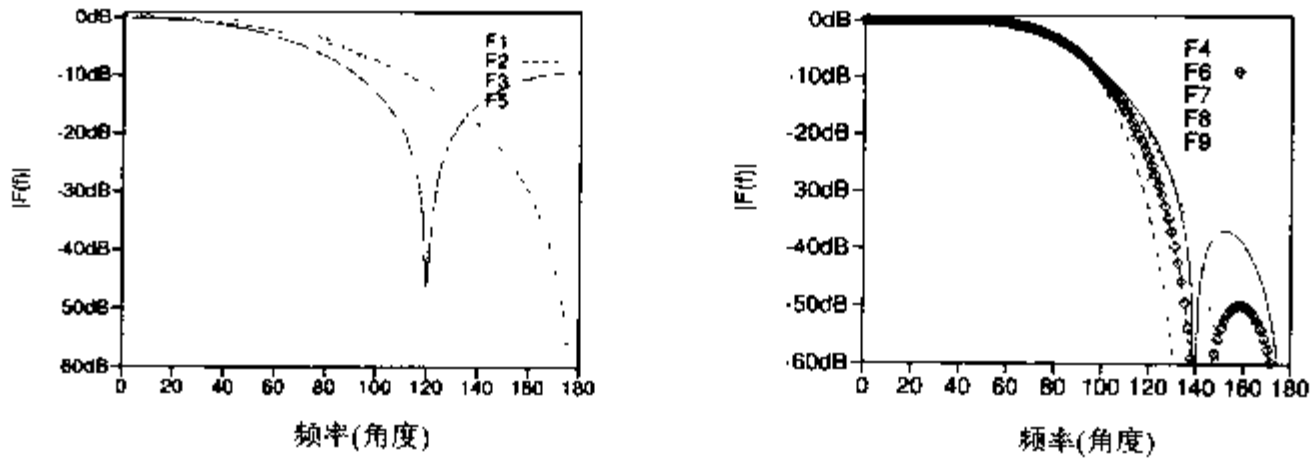


图 5-27　半波带滤波器 F1 至 F9 的传递函数

Goodman 和 Carey 多级抽取器设计的基本思想就是：在第一级中，可以采用较大纹波和较小复杂性的滤波器，因为通频带与采样频率的比率是相当小的。随着通频带与采样频率的比率的增加，我们就必须采用畸变较小的滤波器了。算法停止在 $R=2$ 处。对于最后的抽取($R=2$ 和 $R=1$)，则必须设计一个更长的半波带滤波器。

Goodman 和 Carey 已经提供了如图 5-28 所示的设计图表。首先，必须计算输入重复采样因数 R 与通频带和抑止频带中必要的衰减 $A=A_s=A_p$。由此开始，R、$R/2$，$R/4$...所需要的滤波器就排列成一条水平直线(在相同的抑止频带衰减上)。滤波器 F4 和 F6-F9 在通频带有纹波(请参阅练习 5.8)，如果需要使用几个这样的滤波器的话，就有必要调节 ε_p 了。这样就需要考虑做如下调整：

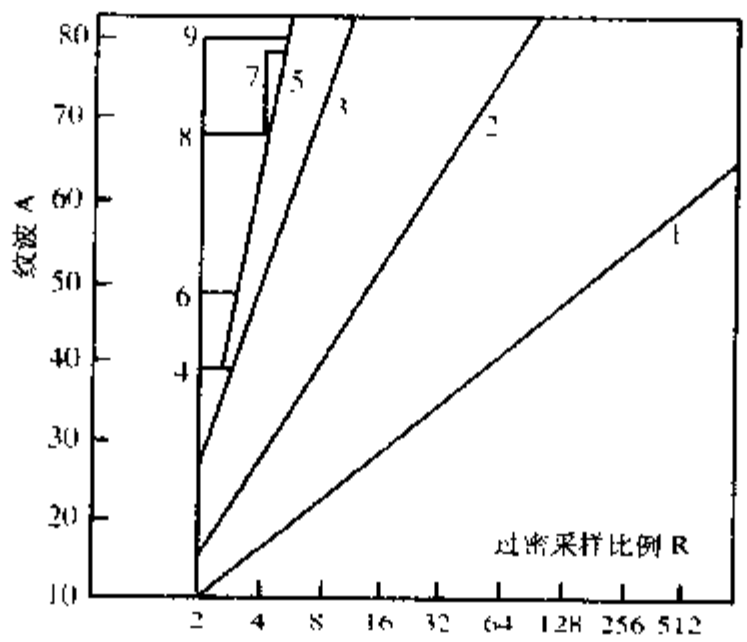


图 5-28　Goodman 和 Carey 设计图表

对于 F1-F3，F5　有

$$A = -20\log_{10}\varepsilon_p \tag{5.38}$$

对于 F4，F6-F9　有

$$A = -20\log_{10}\min(\frac{\varepsilon_p}{S'}, \varepsilon_s) \tag{5.39}$$

其中 S' 是带有纹波的级的数量。

接下来用一个示例来说明多级设计。

例 5.8　多级半波带滤波器抽取器

我们要设计一个采用 Goodman 和 Carey 设计方法的 $R=160$，$\varepsilon_p=0.015$ 和 $\varepsilon_s=0.031=30$dB 的抽取器。

粗略地一看，就会了解到总计需要 5 个滤波器，并且起始点定在 $R=160$ 和 30dB 处，如图 5-29 所示。从 160 到 32 采用了一个长度 $L=5$ 的 CIC 滤波器，在 CIC 滤波器之后跟有两个 F2 滤波器和 1 个 F3 滤波器，借此达到 $R=8$。现在需要 1 个处理纹波的滤波器，它满足：

$$A = -20\log_{10}\min(\frac{0.015}{1}, 0.031) = 36.48\ \text{dB} \tag{5.40}$$

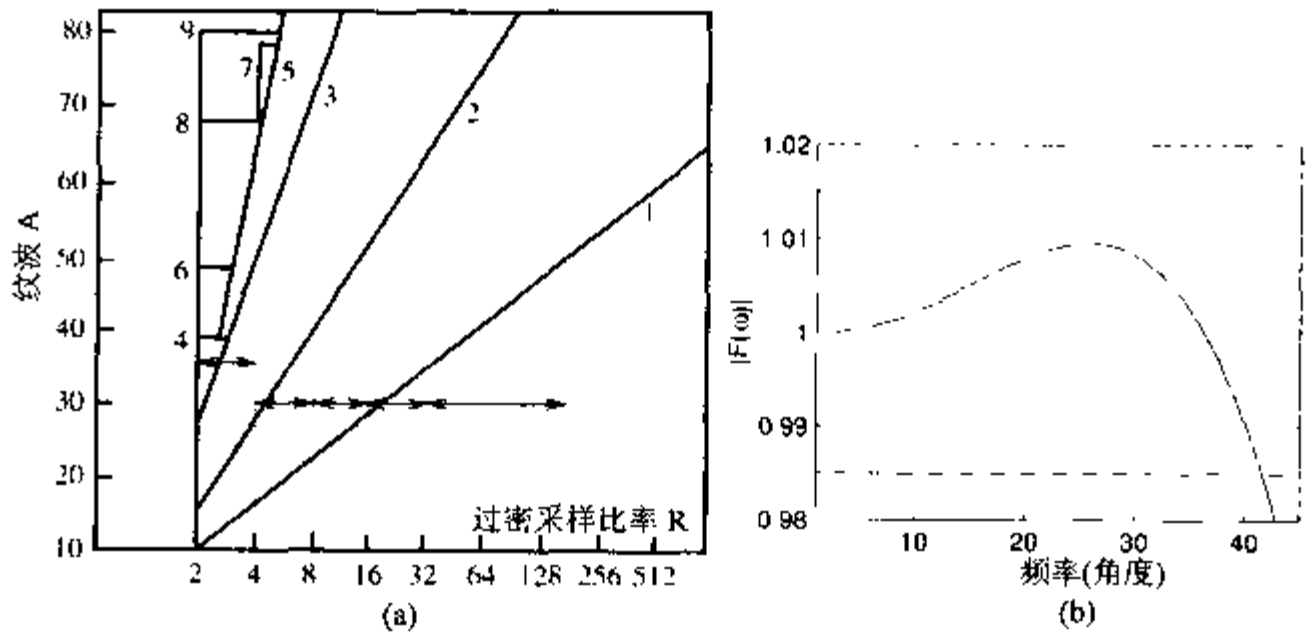


图 5-29　Goodman 和 Carey 半波带滤波器的设计示例。(a) 设计图表 (b) 传递函数$|F(\omega)|$

从图 5-28 可以看出 F4 滤波器适合 36dB。现在我们就可以计算整个滤波器的传递函数$|F(\omega)|$了，采用的是 Noble 关系式 $F(z)=F1(z)\ F2(z^5)\ F2(z^{10})F3(z^{20})\ F4(z^{40})$ (请参阅图 5-6)，如图 5-29(b)所示。图 5-29(a)给出了采用图 5-28 中的设计图表的设计算法。

例 5.8 说明，在(5.39)中仅仅考虑有纹波的滤波器就足够了。采用一种 $S=6$ 的更为不利的方法，就会得到 $A=-20\log(0.015/6)=52$dB，这样就需要 F8 滤波器才会有更好的效果。所以最好从一个最优假设出发，到后来很有可能就是正确的。

5.5　作为通频带抽取器的频率采样滤波器

5.3 节中讨论的 CIC 滤波器属于一个更大的称作**频率采样滤波器**(frequency sampling filter，FSF)的系统。频率采样滤波器可以作为梁化器或抽取滤波器，将信息波谱分解成若干个离散的

次能带，例如在多用户通讯系统中便是如此。经典 FSF 由一个梳状滤波器与一组频率选择谐振器级联[4.53]。这些谐振器各自产生一个极点集合，分别有选择地抵消梳状部分的预滤波器产生的零点。对诸振器的输出进行增益调整，从而改变整个滤波器的幅值频率响应。FSF 也可以通过级联全极点滤波器部分和全零点滤波器(梳状)部分来建立，如图 5-30 所示。通过选择梳状部分的延迟 $1\pm z^{-D}$ 就可以用其零点抵消全极点预滤波器的极点,如图 5-31 所示。无论在哪里有一个复杂的极点，都会有一个相应的复杂零点将其抵消，最终生成全零 FIR 滤波器，而且具有通常的线性相位和常系数组延迟属性。

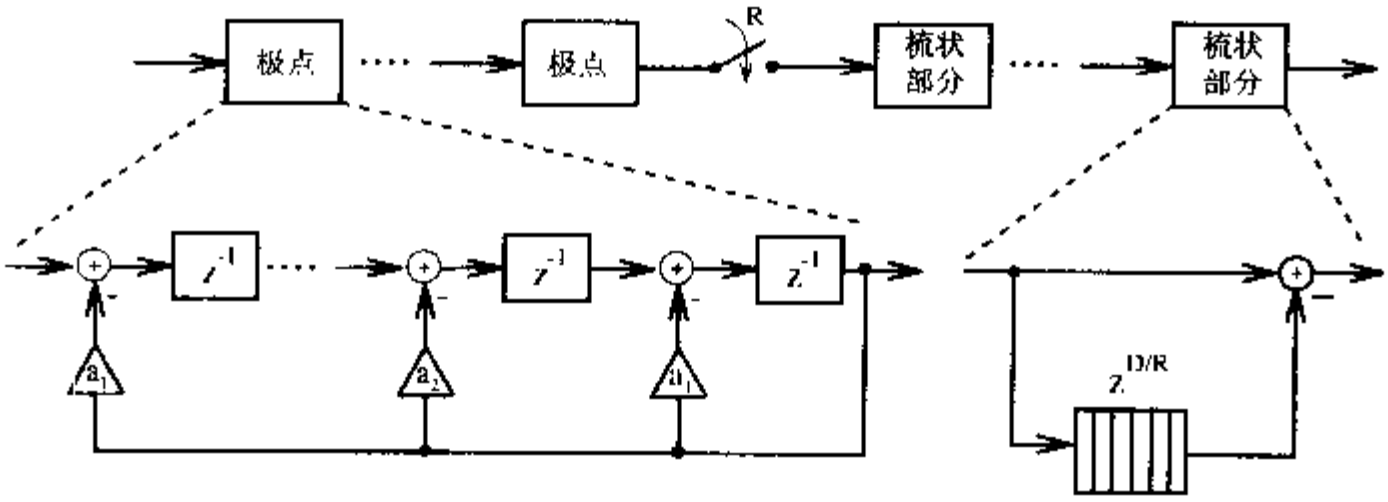
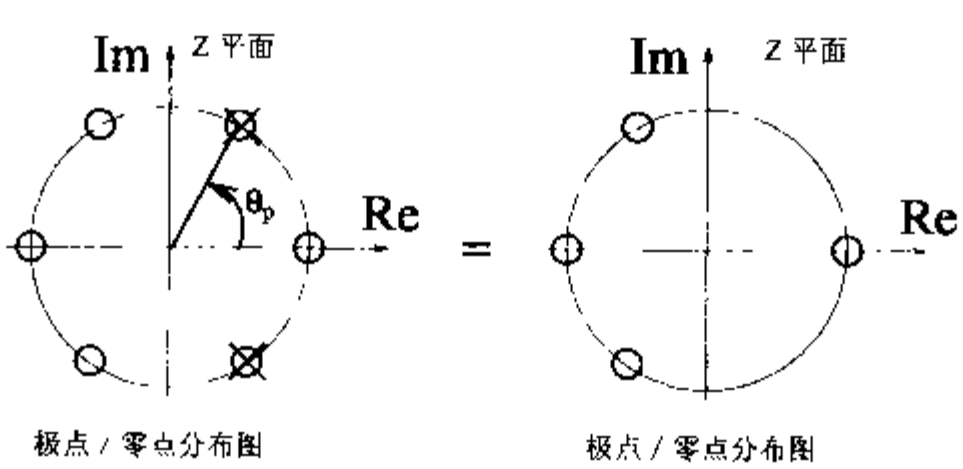


图 5-30　为节省多级信号处理一个因子 R 的延迟而级联频率采样滤波器[4.3.4 b]



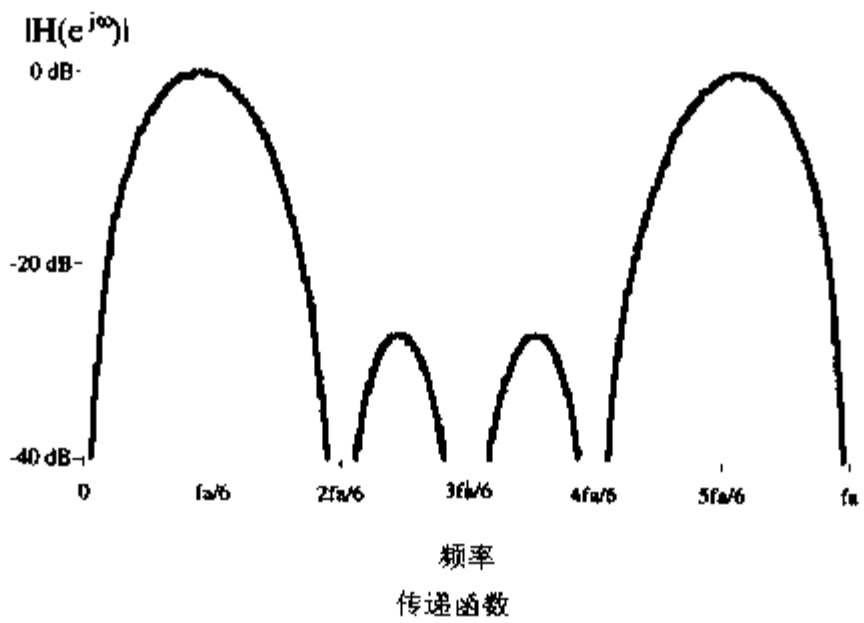


图 5-31　极角度 60° 和梳状延迟 $D=6$ 的极点/零点抵消的示例

频率采样滤波器令多级滤波器组设计者感兴趣的，部分是因为其固有的低复杂性和线性相位性能。FSF 设计依赖于精确的极点-零点抵消，经常在嵌入式场合得到应用。精确的 FSF 极点

-零点抵消可以通过使用二进制补码或 RNS 的整数环定义的多项式滤波器来保证。这样开发的 FSF 滤波器的极点可以驻留在单位圆的边缘。这种有条件地不稳定位置是可以接受的，主要是由于精确的极点-零点抵消的保证。如果没有这种保证的话，设计者就必须将谐振器的极点安置在单位圆内部，这在性能方面会有所损失。此外，允许 FSF 极点和零点驻留在单位圆上，可以建立一个带有较少乘法器的 FSF，而且降低了复杂性，增加了数据带宽。

下面来研究图 5-30 所示的滤波器。可以看到第一阶(整系数)滤波部分产生的极点位于的角度为 0° 和 180°，第二阶(整系数)滤波部分产生的极点根据关系式 $2\cos(2\pi K/D)=1$，0 和 -1 得到，分别位于的角度为 60°，90° 和 120°。表 5-4 给出了高阶部分的频率选择性，给出了所有具有整系数、并且根在单位圆上且不超过 6 阶的多项式的角频率。表 5-4 给出的基本构件可以有效地设计和实现这样的 FSF 滤波器。例如：可以开发二进制补码(也就是 RNS 的模)滤波器组作为常系数 Q 的语音处理滤波器组。其频率覆盖范围从 900 到 8000Hz[89, 90]，采用 16KHz 的采样频率。接下来在设计中添加整系数半波带滤波的 HB6[66] 抑畸变滤波器和二阶自由乘法 CIC 滤波器(也称作 Hogenauer 滤波器，请参阅 5.3 节)，进一步抑制不需要的频率成份，如图 5-32 所示。每个谐振器的带宽都可以通过梳状部分的级数和延迟数量独立地调节。满足预期带宽要求的级数和延迟的数量就是最优的。所有的频率选择滤波器都有两个级和延迟。

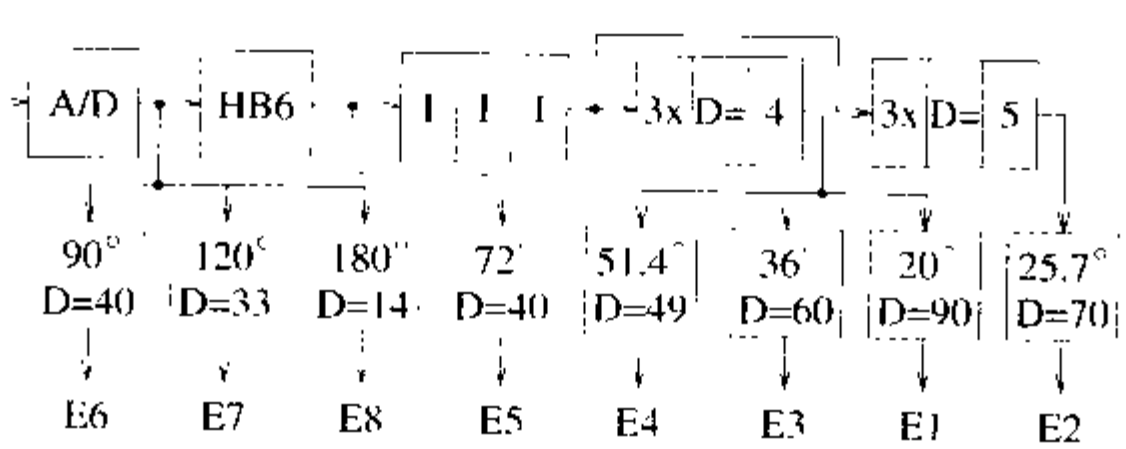


图 5-32　半波带和 CIC 预滤波器以及 FSF 梳状谐振器部分组成的滤波器组的设计

表 5-4 显示的是(到六阶为止)整系数滤波器生成惟一的角极点位置。给出的是滤波器系数和单位圆上根的非冗余角位置。

表 5-4　滤波器系数及单位圆上根的非冗余角位置

$C_K(z)$	阶数	a_0	a_1	a_2	a_3	a_4	a_5	a_6	θ_1	θ_2	θ_3
$-C_1(z)$	1	1	-1						0°		
$C_2(z)$	1	1	1						180°		
$C_6(z)$	2	1	-1	1					60°		
$C_4(z)$	2	1	0	1					90°		
$C_3(z)$	2	1	1	1					120°		
$C_{12}(z)$	4	1	0	-1	0	1			30°	150°	
$C_{10}(z)$	4	1	-1	1	-1	1			36°	108°	
$C_8(z)$	4	1	0	0	0	1			45°	135°	
$C_5(z)$	4	1	1	1	1	1			72°	144°	

(续表)

$C_k(z)$	阶数	a_0	a_1	a_2	a_3	a_4	a_5	a_6	θ_1	θ_2	θ_2
$C_{16}(z)$	6	1	0	0	−1	0	0	1	20.00°	100.00°	140.00°
$C_{14}(z)$	6	1	−1	1	−1	1	−1	1	25.71°	77.14°	128.57°
$C_7(z)$	6	1	1	1	1	1	1	1	51.42°	102.86°	154.29°
$C_9(z)$	6	1	0	0	1	0	0	1	40.00°	80.00°	160.00°

表 5-5 给出了采用 Xilinx XC4000 FPGA 作原型设计的滤波器组。采用高级设计工具(Xilinx 提供的 XBLOCKS)，则使用 CLB 数量的典型值要比理论上由统计加法器、触发器、ROM 和 RAM 得到的预期数值高 20%左右。

表 5-5 显示的是 Xilinx XC4000 FPGA 所使用的 CLB 数量(注意：F20D90 的含义是滤波器极角度为 20.00°，梳状延迟 $D=90$)。总计：实际 1572 个 CLB，非递归 FIR：11292 个 CLB。

表 5-5　Xilinx XC4000 FPGA 所使用的 CLB 数量

	F20 D90	F25 D70	F36 D60	F51 D49	F72 D40	F90 D40	F120 D33	F180 D14	FHB6	III	D4	D5
理论数量	122	184	128	164	124	65	86	35	122	31	24	24
实际数量	160	271	190	240	190	93	120	53	153	36	33	33
非递归 FIR 数量	2256	1836	1039	1140	1039	1287	1260	550				

FSF 的设计可以通过改变梳状部分的延迟、信道幅值或者阶的数量来操控。例如：梳状部分延迟的调整就很容易实现，因为 CLB 用作 32×1 的存储器单元，而计数器则利用作为存储器单元的 CLB 实现给定的梳状部分延迟。

5.6　滤波器组

数字滤波器组是一组具有公共输入或输出的滤波器，如图 5-33 所示。图 5-33(a)中的分析滤波器组经常用于频谱分析，也就是将输入信号分成 R 个不同的次能带信号。图 5-33(b)中将多个信号合成到公共的输出信号中，就称作合成滤波器组。分析滤波器可以是不相重叠的、稍有重叠或者是完全重叠的。图 5-34 给出了一个最为常见的稍有重叠滤波器组的示例。

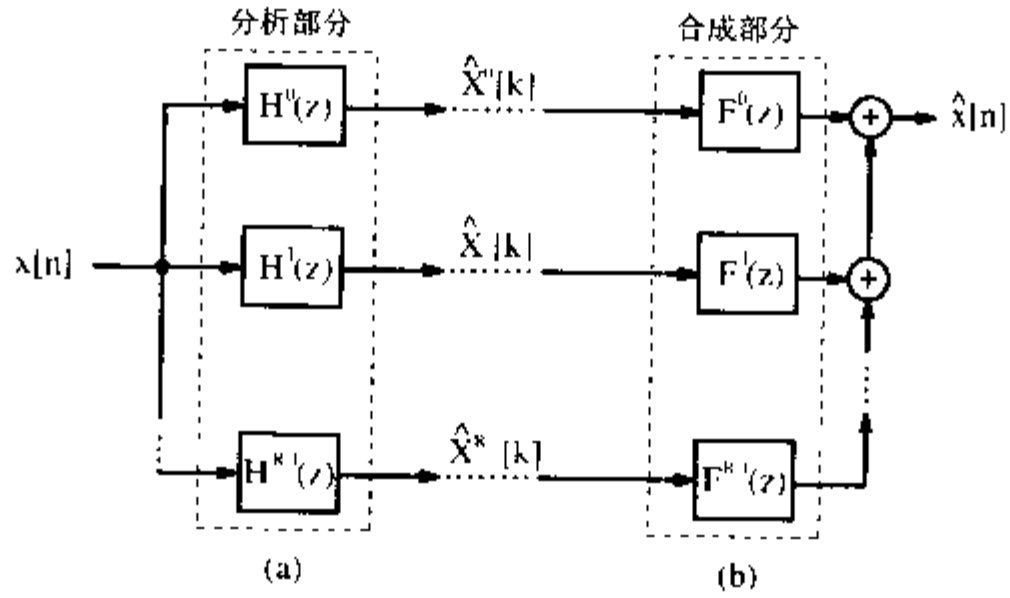


图 5-33　典型滤波器组的分解系统展示 (a) 分析滤波器 (b) 合成滤波器

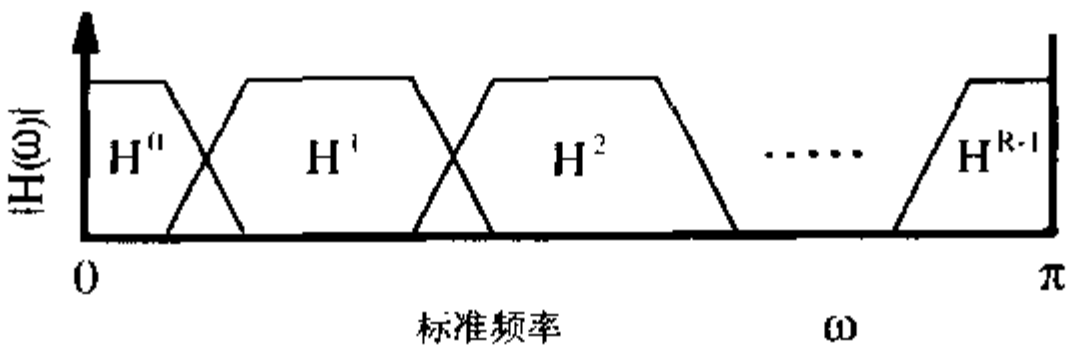


图 5-34　有少量混叠的 R 信道滤波器组

区别不同滤波器组的另一个重要特征就是带宽和各个滤波器中心频率之间的间隔。非均匀滤波器组的一个例子就是倍频间隔或小波滤波器组，我们将在 5.7 节中加以讨论。在均匀滤波器组中，所有的滤波器都具有同样的带宽和采样速率。从实现的角度来讲，比较倾向于均匀的、最大抽取滤波器组，这是因为它们在 FFT 算法的帮助下可以得以实现，我们将在下一节详细讨论。

5.6.1　均匀 DFT 滤波器组

在最大抽取或精密采样滤波器组中，抽取或插入数 R 与频带的数量 K 相等。如果第 r 个频带滤波器 $h^r[n]$是由单个原型滤波器 $h[n]$依据下面的公式计算得来的，那么我们就称之为 DFT 滤波器组。

$$h^r[n] = h[n]W_R^{rn} = h[n]e^{-j2\pi rn/R} \tag{5.41}$$

如果我们采用滤波器 $h^r[n]$和输入信号 $x[n]$的多相分解，就能够得到 R 个信道的滤波器组的一个有效实现。因为每个带通滤波器都是精密采样的，我们采用 R 个多相信号的分解，公式如下：

$$h[n] = \sum_{k=0}^{R-1} h_k[n] \leftrightarrow h_k[m] = h[mR-k] \tag{5.42}$$

$$x[n] = \sum_{k=0}^{R-1} x_k[n] \leftrightarrow x_k[m] = x[mR-k] \tag{5.43}$$

现在将(5.42)式代入(5.41)式，就会发现，所有带通滤波器 $h^r[n]$共用同一个相位滤波器 $h_k[n]$，而每个滤波器的“旋转因子”是不同的。图 5-35(a)给出了 $h^r[n]$的第 r 个滤波器的结构。很明显，$h^r[n]$ 的“旋转乘法”在输入向量为 $\hat{x}_0[n]$、$\hat{x}_1[n]$、…$\hat{x}_{R-1}[n]$时，与第 r 个 DFT 组件相关。对整个分析频带的计算就可以简化为用 R 个多相滤波器的滤波，如图 5-35(b)所示，跟随的是这 R 个滤波组件的 DFT(或 FFT)。很明显，这种方法要比(5.41)式中定义的滤波器的直接计算(参阅 5.6 练习)更加有效。

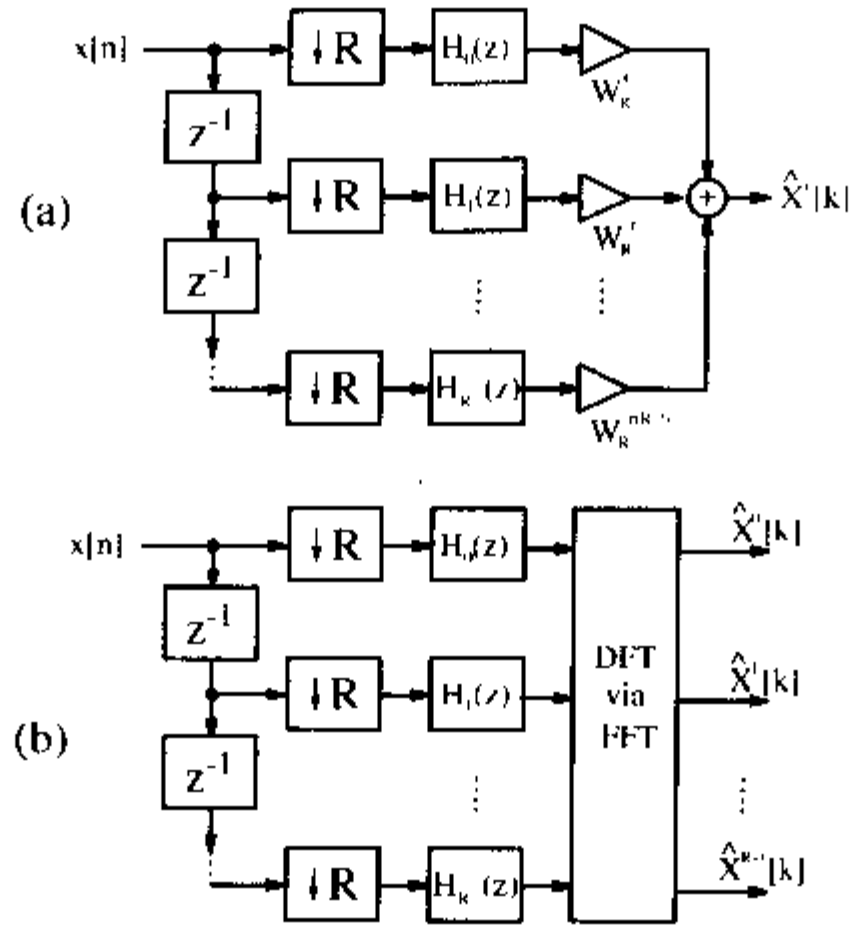


图 5-35　(a) 信道 k 的分析 DFT 滤波器组 (b) 完整的分析 DFT 滤波器组

均匀 DFT 合成滤波器的多相滤波器组的开发过程正好与分析滤波器组的开发过程相反，也就是说，我们可以用 R 个频谱成份 $\hat{X}^r[k]$ 作为逆 DFT(或 FFT)的输入，并用多相插入结构重构输出信号，如图 5-36 所示，重构的带通滤波器就变成：

$$f^r[n]=\frac{1}{R}f[n]W_R^{-rn}=f[n]e^{j2\pi rn/R} \tag{5.44}$$

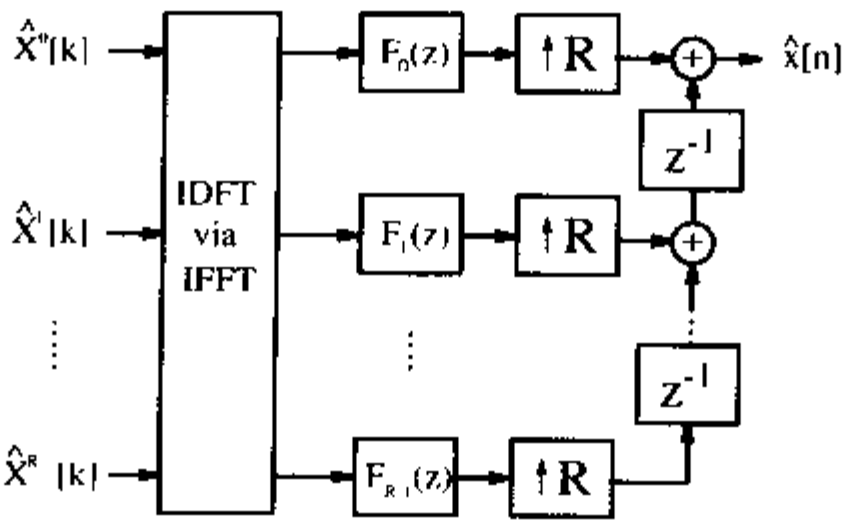


图 5-36　DFT 合成滤波器组

现在将分析滤波器组与合成滤波器组加以组合，就会看到 DFT 与 IDFT 相互抵消，如果包含多相滤波器的卷积给出一个单位采样函数，就会得到一些完美的重构结构。这个采样函数如下：

$$h_r[n]*f_r[n]=\begin{cases}1 & n=d\\ 0 & \text{其他}\end{cases} \tag{5.45}$$

换句话说，就是两个多相函数必须是互逆滤波器，也就是：

$$H_r(z)*F_r(z)=z^{-d}$$

$$F_r(z)=\frac{z^{-d}}{H_r(z)}$$

其中为了得到(可实现的)因果滤波器，我们提供了一个延迟 d。在实际设计中，这些理想情况是不能用两个 FIR 滤波器来得到精确满足的。可以采用两个 FIR 滤波器或者是将一个 FIR 与 IIR 滤波器组合起来得到近似的满足，请参阅下面的示例。

例 5.9　DFT 滤波器组

例 4.3 中讨论过的有耗积分器在结构上可以解释成一个 $R=2$ 的 DFT 滤波器组。差分方程如下：

$$y[n+1]=\frac{3}{4}y[n]+x[n] \tag{5.46}$$

这一滤波器在 z 域内的脉冲响应是：

$$F(z)\frac{z^{-1}}{1-0.75z^{-1}} \tag{5.47}$$

要得到两个多相滤波器，我们可以采用相近的结构对“分散预先考虑”(请参阅例 4.5)作改进。也就是说在镜像位置引入一个额外的极点/零点对。用$(1+0.75z^{-1})$分别乘以分母和分子，得到：

$$F(z)=\underbrace{\frac{0.75z^{-2}}{1-0.75^2z^{-2}}}_{H_0(z^2)}+z^{-1}\underbrace{\frac{1}{1-0.75^2z^{-2}}}_{H_1(z^2)} \tag{5.48}$$

$$=H_0(z^2)+z^{-1}H_1(z^2) \tag{5.49}$$

后者给出两个多相滤波器：

$$H_0(z)=\frac{0.75z^{-1}}{1-0.75^2z^{-1}}=0.75z^{-1}+0.4219z^{-2}+0.2373z^{-3}+\ldots \tag{5.50}$$

$$H_1(z)=\frac{1}{1-0.75^2z^{-1}}=1+0.5625z^{-1}+0.3164z^{-2}+\ldots \tag{5.51}$$

可以用非递归 FIR 近似这些脉冲响应，但是要达到误差小于 1%，就必须要使用 16 个系数。所以，如果采用两个分别由(5.50)式和(5.51)式定义的递归多相 IIR 滤波器就会更加有效。在用多相滤波器分解之后，就可以采用一个由下面矩阵给出的 2-点 DFT：

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}$$

整个分析滤波器组现在就被构造成图 5-37(a)的形式。对于合成滤波器组而言首先必须用下面的矩阵计算逆 DFT：

$$\boldsymbol{W}^{-1}=\frac{1}{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

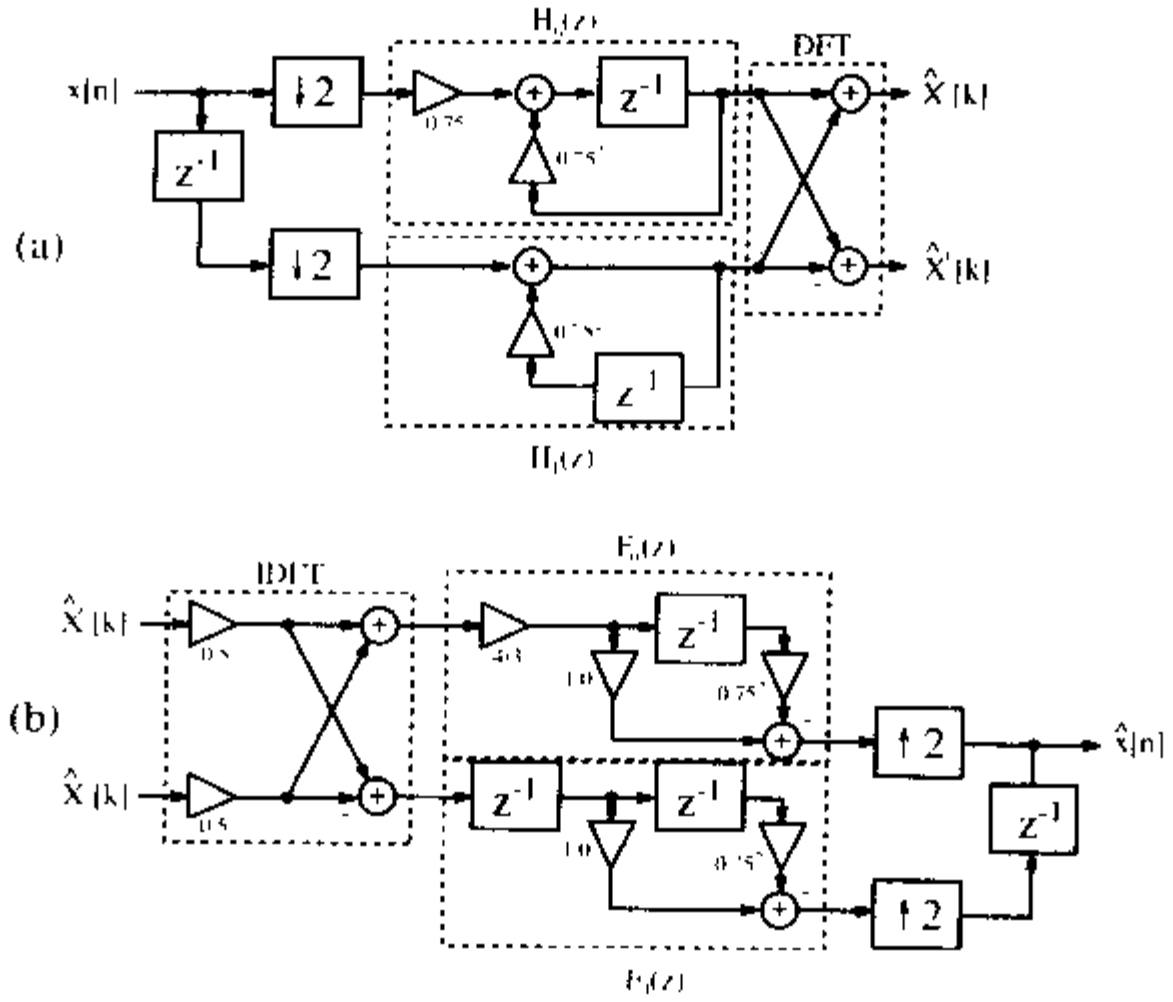


图 5-37　$R=2$ 的严格采样的均匀 DFT 滤波器组 (a) 分析滤波器组 (b) 合成滤波器组

为了获得完美的重构，就必须找到 $h_0[n]$和 $h_1[n]$的逆滤波器。这并不困难，因为 $H_r(z)'$是单极点 IIR 滤波器，而 $F_r(z)=z^{-d}/\ H_r(z)$就必然是两抽头 FIR 滤波器。利用(5.50)式和(5.51)式就可以得到 $d=1$，这对于得到因果滤波器而言就已经足够了，就是：

$$F_0[n]=\frac{4}{3}(1-0.75^2z^{-1}) \tag{5.52}$$

$$F_1[n]=z^{-1}-0.75^2z^{-2} \tag{5.53}$$

图 5-37(b)给出了合成滤波器组的图形化解释。

5.6.2　双信道滤波器组

双信道滤波器组是设计通用滤波器组和小波分析的一个重要工具。图 5-38 给出了一个双信道滤波器组的例子，它将输入信号用低通($G(z)$)和高通($H(z)$)“分析”滤波器分开。在分析和合成部分之间是两个单位的抽取和插入。抽取器和插入器之间的信号通常都是量化的，为了增强或者压缩而做过非线性化处理的。

习惯做法是只定义低通滤波器 $G(z)$，然后用其定义来规定高通滤波器 $H(z)$。正常的构造规则由下面的公式给出：

$$h[n]=(-1)^n g[n]\circ\!-\!\bullet\ H(z)=G(-z) \tag{5.54}$$

这一公式确定了滤波器将是成对镜像的，特别是在频域中，$|H(e^{j\omega})|=|G(e^{j(\omega-\pi)}|$。这就是**正交镜像滤波器**(Quadrature Mirror Filter, QMF)**组**，因为两组滤波器都是关于 $\pi/2$ 镜像对称的。

对于图 5-38 给出的合成形式，我们首先采用一个扩展器(采样速率提高 2 倍)，接下来是两组独立重构的滤波器 $\hat{G}(z)$ 和 $\hat{H}(z)$，共同重构 $\hat{x}(n)$。现在面临的一个挑战性的问题就是，输入信号能够完美地重构吗？也就是能否满足：

$$\hat{x}(n) = x[n-d] \tag{5.55}$$

这就是说完美的重构信号具有与最初信号相同的形状，即相当于相(时)移。因为 $G(z)$和 $H(z)$都不是理想的方波滤波器，实现完美的重构不是一个简单的问题。两组滤波器在 2 倍向下采样之后都会产生主要的畸变成份，如图 5-38 所示。满足(5.55)的简单正交滤波器组是由 Alfred Haar 提出来的(大约在 1910 年)[91]。

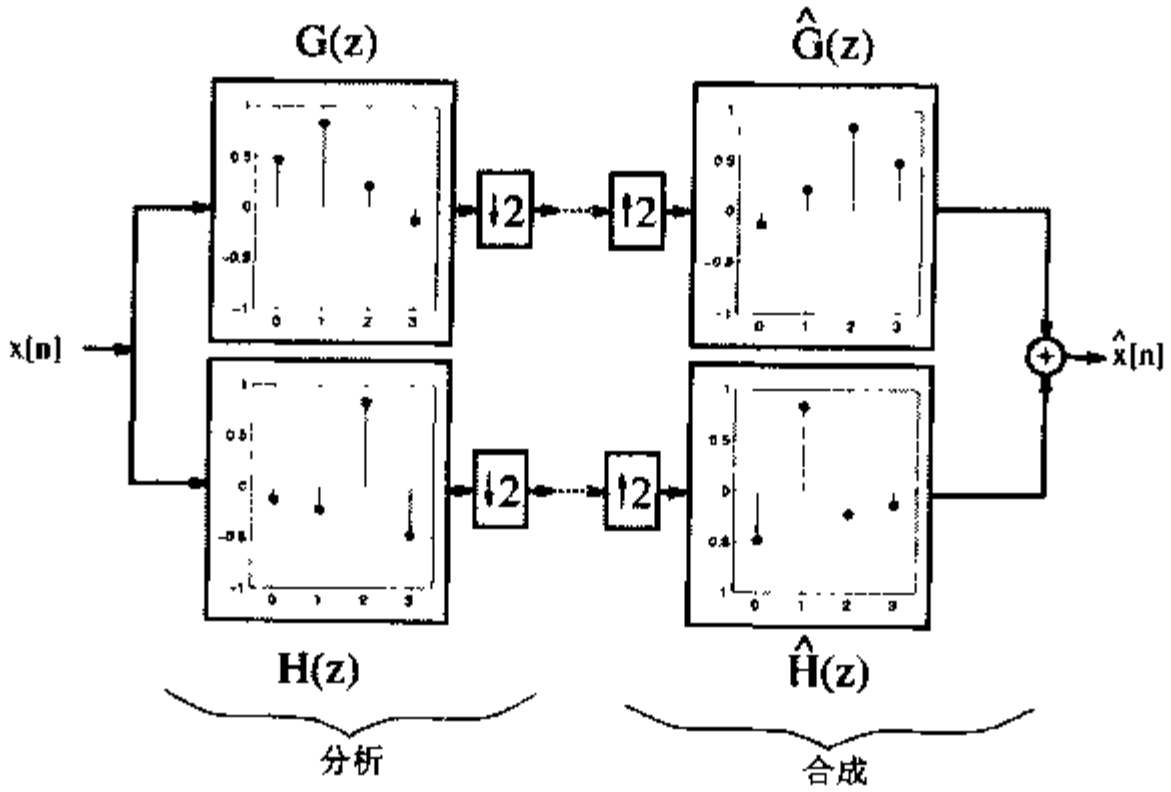


图 5-38　采用长度为 4 的 Daubechies 滤波器的双信道滤波器组

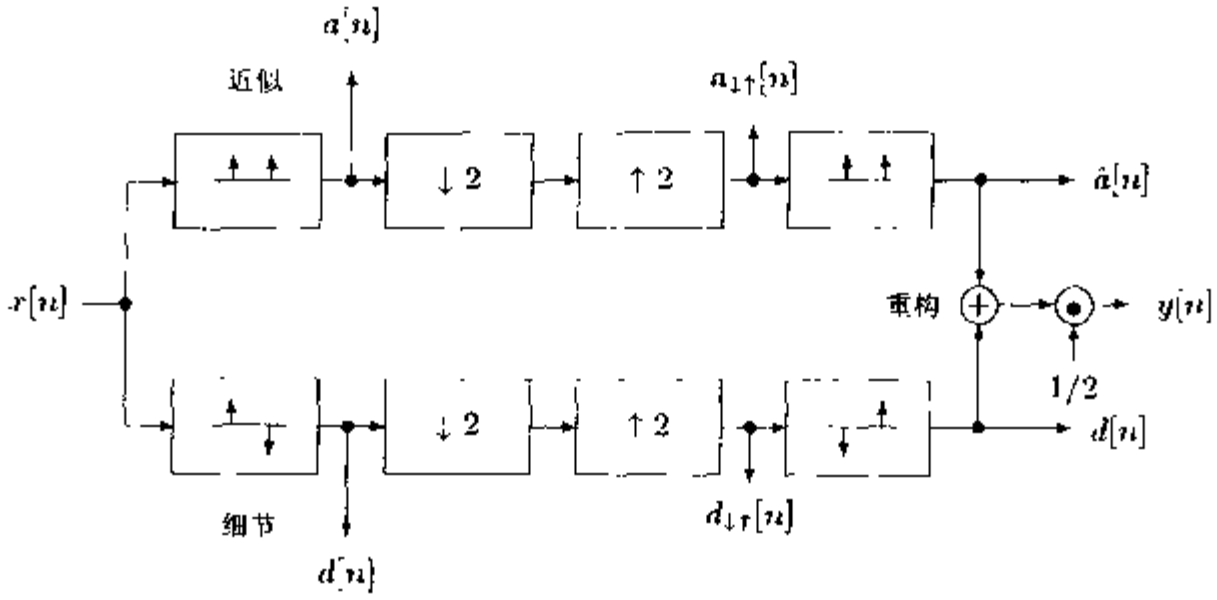
例 5.10　双信道 Haar 滤波器组 I

图 5-39 中双信道 QMF 滤波器组的滤波传递函数是[8]

$$G(z) = 1 + z^{-1} \qquad H(z) = 1 - z^{-1}$$

$$\hat{G}(z) = \frac{1}{2}(1 + z^{-1}) \qquad \hat{H}(z) = \frac{1}{2}(-1 + z^{-1})$$

注 8：通常，幅值因子是用获得正交滤波器的方法选择的，也就是$\sum_n |h[n]|^2 = 1$。这样，滤波器的幅值因子就是$1/\sqrt{2}$。这就使得硬件设计虽然变得更复杂了。



	Time step n				
	0	1	2	3	4
$x[n]$	$x[0]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$
$a[n]$	$x[0]$	$x[0]+x[1]$	$x[1]+x[2]$	$x[2]+x[3]$	$x[3]+x[4]$
$d[n]$	$x[0]$	$x[1]-x[0]$	$x[2]-x[1]$	$x[3]-x[2]$	$x[4]-x[3]$
$a_{\downarrow\uparrow}[n]$	0	$x[0]+x[1]$	0	$x[2]+x[3]$	0
$d_{\downarrow\uparrow}[n]$	0	$x[1]-x[0]$	0	$x[3]-x[2]$	0
$\hat{a}[n]$	0	$x[0]+x[1]$	$x[0]+x[1]$	$x[2]+x[3]$	$x[2]+x[3]$
$\hat{d}[n]$	0	$x[0]-x[1]$	$x[1]-x[0]$	$x[2]-x[3]$	$x[3]-x[2]$
$\hat{x}[n]$	0	$x[0]$	$x[1]$	$x[2]$	$x[3]$

图 5-39　双信道 Haar-QMF 滤波器组(©1999 Springer 出版社[5])

使用图 5-39 中表里的数据，就可以确认滤波器生成了输入信号的完美的重构信号。输入序列 $x[0]$、$x[1]$、$x[2]$、…，经过 $G(z)$和 $H(z)$的处理，分别生成和 $x[n]+x[n-1]$与差 $x[n]-x[n-1]$。向上采样前面的向下采样迫使每隔一个值的值都归零。在应用合成滤波器并将输出组合后，再一次得到有一个延迟的输入序列，也就是 $\hat{x}(n)=x[n-1]$，这是一个完美的重构，其中 $d=1$。

接下来我们要讨论 4 个滤波器组，要获得完美重构必须遵循的一般关系。有必要记住抽取和插入为 2 的信号 $s[k]$等价于 $S(z)$乘以序列{1，0，1，0，…，}。这在 z 域内就转换成：

$$S_{\downarrow\uparrow}(z)=\frac{1}{2}(S(z)+S(-z)) \tag{5.56}$$

如果将这一信号应用到双信道滤波器组上，低通通路 $X_{\downarrow\uparrow G}(z)$和高通信道 $X_{\downarrow\uparrow H}(z)$就变成：

$$X_{\downarrow\uparrow G}(z)=\frac{1}{2}(X(z)G(z)+X(-z)G(-z)) \tag{5.57}$$

$$X_{\downarrow\uparrow H}(z)=\frac{1}{2}(X(z)H(z)+X(-z)H(-z)) \tag{5.58}$$

在与合成滤波器 $\hat{G}(z)$ 和 $\hat{H}(z)$ 相乘后，总结最终结果，得到 $\hat{X}(z)$ 为：

$$\begin{aligned}\hat{X}(z)&=X_{\downarrow\uparrow G}(z)\hat{G}(z)+X_{\downarrow\uparrow H}(z)\hat{H}(z)\\&=\frac{1}{2}(G(z)\hat{G}(z)+H(z)\hat{H}(z))X(z)\\&\quad+\frac{1}{2}(G(-z)\hat{G}(z)+H(-z)\hat{H}(z))X(-z)\end{aligned} \tag{5.59}$$

$X(-z)$的因子表示混叠部分，$X(z)$的项表示幅值畸变。要想得到完美的重构，这一转换需要遵循：

法则 5.11　完美的重构

如图 5-38 所示，双信道滤波器组完美重构的条件是：

(1) $G(-z)\hat{G}(z)+H(-z)\hat{H}(z)=0$，也就是重构与混叠无关。

(2) $G(z)\hat{G}(z)+H(z)\hat{H}(z)=2z^{-d}$，也就是幅值畸变的幅度是 1。

接下来核对一下 Haar 滤波器组的情形。

例 5.12　双信道 Haar 滤波器组 II

双信道 QMF Haar 滤波器组的定义如下：

$$G(z)=1+z^{-1} \qquad H(z)=1-z^{-1}$$

$$\hat{G}(z)=\frac{1}{2}(1+z^{-1}) \qquad \hat{H}(z)=\frac{1}{2}(-1+z^{-1})$$

法则 5.11 的两种情形的证明如下：

(1) $G(-z)\hat{G}(z)+H(-z)\hat{H}(z)$

$$=\frac{1}{2}(1-z^{-1})(1+z^{-1})+\frac{1}{2}(1+z^{-1})(-1+z^{-1})$$

$$=\frac{1}{2}(1-z^{-2})+\frac{1}{2}(-1+z^{-2})=0 \qquad \surd$$

(2) $G(z)\hat{G}(z)+H(z)\hat{H}(z)$

$$=\frac{1}{2}(1+z^{-1})^2+\frac{1}{2}(1-z^{-1})(-1+z^{-1})$$

$$=\frac{1}{2}((1+2z^{-1}+z^{-2})+(-1+2z^{-1}-z^{-2}))=2z^{-1} \qquad \surd$$

采用法则 5.11 作为依据，可以注意到，如果我们将输入滤波器和输出滤波器调换，不会影响完美重构的情形。

下面我们要讨论一下在滤波器设计中的一些限制，以满足 5.11 的情形，且更容易实现的重构。

首先，利用下面的法则限制滤波器的选择：

法则 5.13　与混叠无关的双信道滤波器组

如果 $G(-z)=-\hat{H}(z)$ 和 $H(-z)=\hat{G}(z)$ (5.60)

则此双信道滤波器组与混叠无关。

如果将(5.60)式用于法则 5.11 中的第一种情形，就可以对这一法则进行核对。

使用长度为 4 的滤波器时，两种情形可以分别解释成：

$$g[n]=\{g[0],g[1],g[2],g[3]\} \rightarrow \hat{h}[n]=\{-g[0],g[1],-g[2],g[3]\}$$

$$h[n]=\{h[0],h[1],h[2],h[3]\}\rightarrow \hat{g}[n]=\{h[0],-h[1],h[2],-h[3]\}$$

有了法则 5.13 对滤波器的约束，现在就可以简化法则 5.11 中的第二种情形。有必要首先定义一个辅助乘积滤波器 $F(z)=G(z)\hat{G}(z)$。法则 5.11 中的第二种情形变为：

$$G(z)\hat{G}(z)+H(-z)\hat{H}(-z)=F(z)+\hat{G}(-z)G(-z)=F(z)-F(-z) \tag{5.61}$$

最终得到：

$$F(z)-F(-z)=2z^{-d} \tag{5.62}$$

也就是说乘积滤波器必须是一个半波带滤波器[9]。完美重构滤波器组的构造分下面 3 个步骤：

算法 5.14　完美重构双信道滤波器组

(1) 根据方程(5.62)定义一个标准半波带滤波器。

(2) 将滤波器 $F(z)$分解成 $F(z)=G(z)\hat{G}(z)$。

(3) 利用方程(5.60)计算 $H(z)$和 $\hat{H}(z)$，也就是 $\hat{H}(z)=-G(-z)$和 $H(z)=\hat{G}(-z)$

接下来用示例说明算法 5.14。为了简化符号表示法，在下面的示例中编写一个长度为 L 的滤波器作为 $G(z)$，长度为 N 的滤波器作为 $\hat{G}(z)$，并将两者组合作为一个 L/N 滤波器。

例 5.15　采用 F3 的完美重构滤波器组

长度为 7 的(标准)半波带滤波器 F3 具有如下传递函数：

$$\mathrm{F3}(z)=\frac{1}{16}(-1+9z^{-2}+16z^{-3}+9z^{-4}-z^{-6}) \tag{5.63}$$

传递函数的零点位于 $z_{01-4}=-1$，$z_{05}=2+\sqrt{3}=3.7321$和 $z_{06}=2-\sqrt{3}=0.2379=1/z_{05}$。分解 $F(z)=G(z)\hat{G}(z)$ 可以有不同的选择。例如，一个 5/3 滤波器可以是：

(1) $G(z)=(-1+2z^{-1}+6z^{-2}+2z^{-3}-z^{-4})/8$ 和 $\hat{G}(z)=(1+2z^{-1}+z^{-2})/2$

可以将 4/4 滤波器设计成：

(2) $G(z)=\frac{1}{4}(1+z^{-1})^3$ 和 $\hat{G}(z)=\frac{1}{4}(-1+3z^{-1}+3z^{-2}-z^{-3})$

4/4 结构的另一种形式采用了 Daubechies 滤波器结构，后者经常用于小波分析，其形式如下：

(3) $G(z)=\frac{1-\sqrt{3}}{4\sqrt{2}}(1+z^{-1})^2(-z_{05}+z^{-1})$ 和 $\hat{G}(z)=-\frac{1+\sqrt{3}}{4\sqrt{2}}(1+z^{-1})^2(-z_{06}+z^{-1})$

图 5-40 给出了这 3 种组合以及相应的极点/零点图。

注 9：有关半波带滤波器的定义请参阅 5.4.1 节的“定义 5.7”。

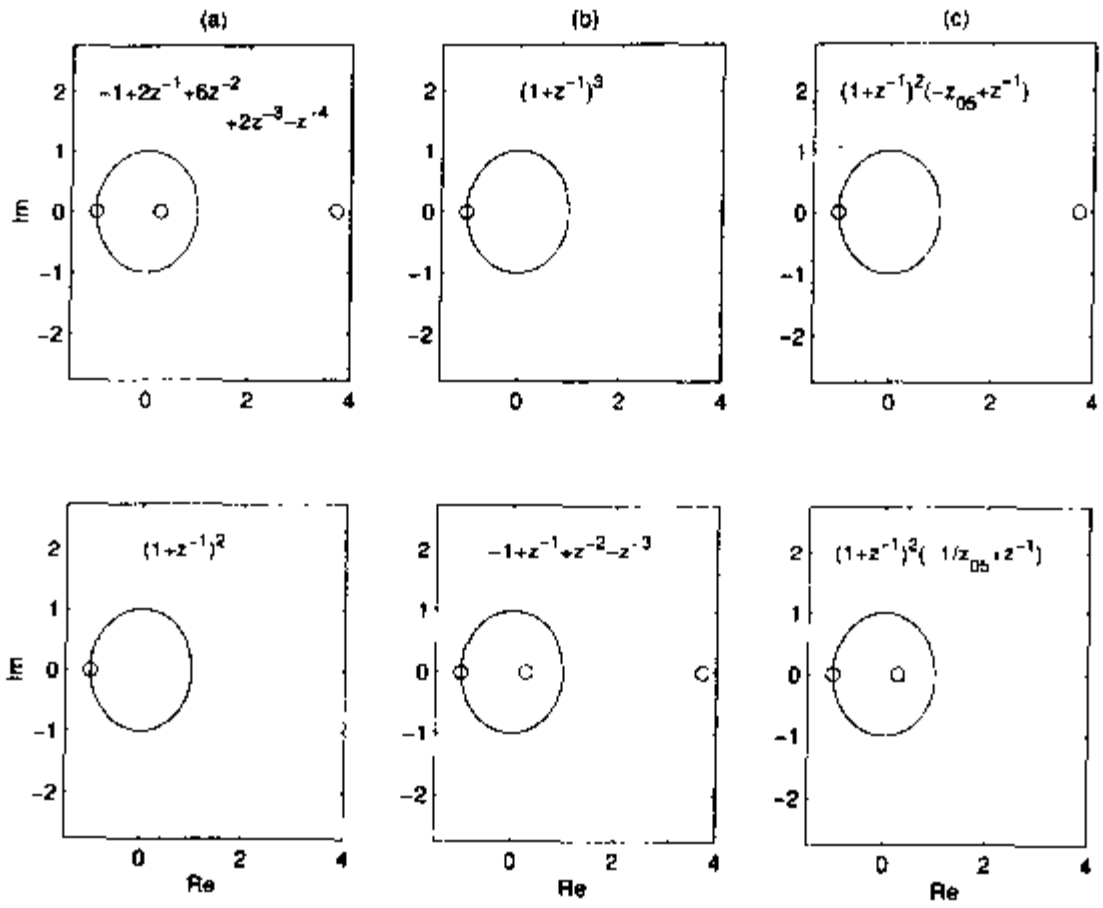


图 5-40　不同因子的半波带滤波器 F3 的极点/零点分布图。上排是 $G(z)$，下排是 $\hat{G}(z)$。

(a) 线性相位 5/3 滤波器 (b) 线性相位 4/4 滤波器 (c) 4/4 Daubechies 滤波器

对于 Daubechies 滤波器而言，还有条件 $H(z)=-z^{-N}G(-z^{-1})$的约束。也就是说，高通和低通多项式彼此互为镜像。这是正交滤波器的典型性能。

从图 5-40 给出 $F(z)=G(z)\hat{G}(z)$的极点/零点图可以得出下列结论：

推论 5.16　半波带滤波器的因式分解

(1) 要构造一个实数滤波器，在同一滤波器中必须保证组成位于(z_0 和 z_0*)的共轭对称零点。

(2) 对于线性相位滤波器，极点/零点图必须要关于单位圆(z=1)对称。在(z_0 和 $1/z_0$)的零点对必须安置在同一滤波器中。

(3) 要得到彼此镜像正交的正交滤波器($(F(z)=U(z)U(z^{-1})$)，所有的 z_0 和 $1/z_0$ 对都必须安置在不同的滤波器组中。

我们可以看到，上面的条件中有一些是不能够同时满足的。特别是第(2)条和第(3)条就表述了一个矛盾。正交、线性相位滤波器一般是不可能的，除非所有的零点都在单位圆上，就像 Haar 滤波器组一样。

如果将例 5.15 中的滤波器组分类，结构(a)和(b)是实线性相位滤波器，而(c)是实正交滤波器。

实现双信道滤波器组

现在要讨论实现双信道滤波器组的不同选择。首先讨论一般的情形。然后再根据滤波器是 QMF、线性相位还是正交，再具体地简化。这里只讨论分析滤波器组，因为合成滤波器组只需要将图交换位置就可以得到了。

1. **多相双信道滤波器组**。一般情况下，对于两个滤波器 $G(z)$和$H(z)$，可以认为每个滤波器都是由多相滤波器：

$$H(z)=H_0(z^2)+z^{-1}H_1(z^2) \qquad G(z)=G_0(z^2)+z^{-1}G_1(z^2) \tag{5.64}$$

实现的，如图 5-41 所示。这样并没有降低硬件的工作量(还是使用了 $2L$ 个乘法器和 $2(L-1)$ 个加法器)。但是这种设计的运行速度是通常采样频率的两倍，就是 $2f_s$。

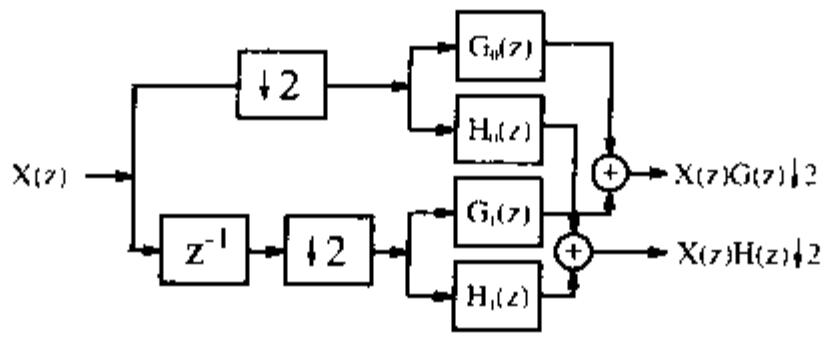


图 5-41　双信道滤波器组的多相实现

有 4 个多相滤波器只有通常滤波器一半的长度。我们可以直接实现这些长度为 $L/2$ 的滤波器，也可以用下面的方法：

(1) 游程长度滤波器采用 5.2.2 节讨论的短 Winograd 卷积算法[84]。

(2) 快速卷积采用 FFT(将在第 6 章讨论)或者 NFT(将在第 7 章讨论)。

(3) 采用第 3 章讨论过的高级算法概念，比如：分布式算法、简化加法器图或 RNS。

采用快速卷积 FFT/NTT 技术还具有额外的优点，就是每个多相滤波器的正向传输只需要进行一次即可。而且反向传输可以应用于两组成份的频谱和，如图 5-42 所示。但是一般情况下，FFT 方法只对较长的滤波器有提高作用，典型的长度值大于 32，然而典型的双信道滤波器长度值都是小于 32 的。

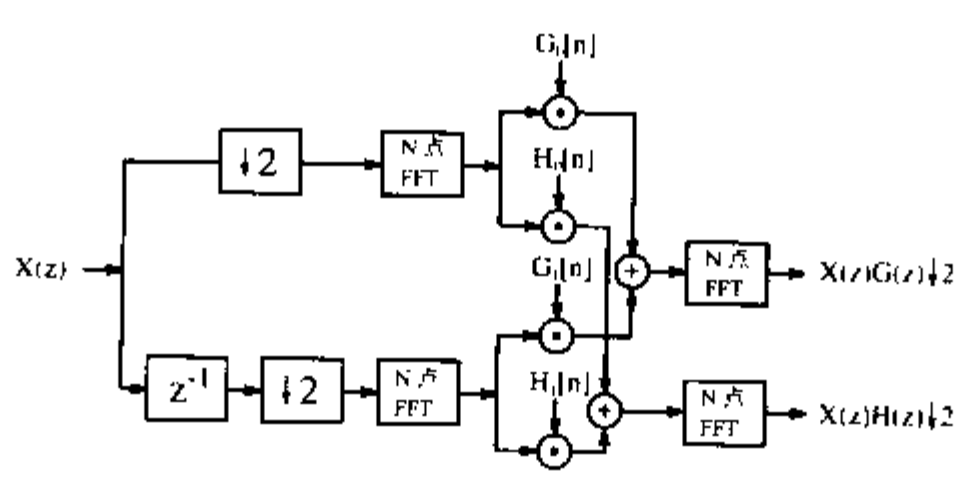


图 5-42　采用 FFT 进行多相分解和快速卷积的双信道滤波器组(©1999 Springer 出版社[5])

2. 提升。另一种常见的构造快速、高效双信道滤波器组的方法就是近期由 Swelden 和 Herley 提出的**提升模式**[92, 93]。其基本思路就是采用交叉项(称为提升和双重提升)，就像网格滤波器一样，要根据短滤波器来构造较长的滤波器，并且保证完美重构条件。基本结构如图 5-43 所示。

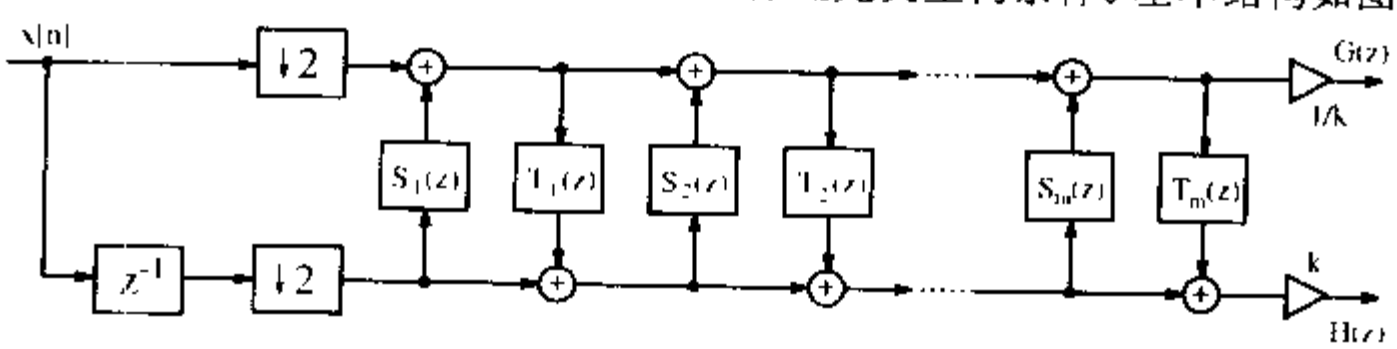


图 5-43　采用提升和双重提升步骤的双信道滤波器实现

设计提升模式，一般从“惰性滤波器组” $G(z)=\hat{H}(z)=1$ 和 $H(z)=\hat{G}(z)=z^{-1}$ 开始，这种新的滤波器组满足法则 5.11 的两个条件，也就是说，这是一种完美的重构滤波器组。随之而来的问题就出现了：如果保持一个滤波器不动，$S(z)$和 $T(z)$如何变化才能够保证滤波器组仍然

是完美的重构滤波器组呢？答案是十分重要的，而且也并不简单：

提升：对于任意 $S(z^2)$，有 $H'(z)=H(z)+G(-z)S(z^2)$　　(5.65)

双重提升：对于任意 $T(z^2)$，有 $G'(z)=G(z)+H(-z)T(z^2)$　　(5.66)

将提升公式代入法则 5.11 的完美重构条件，并进行检验，就会发现，如果 $\hat{G}(z)$ 和 $\hat{H}(z)$ 仍然符合法则 5.13 中关于与混叠无关的滤波器组(练习 5.9)的条件，则两种情况都能够得到满足。

接下来用在提升步骤中的 Daubechies 长度为 4 的滤波器的卷积来说明这种设计。

例 5.17　DB4 滤波器的提升实现

例 5.15 中的滤波器就是一个 Daubechies 长度为 4 的滤波器[94]，滤波器系数是：

$$G(z)=((1+\sqrt{3})+(3+\sqrt{3})z^{-1}+(3-\sqrt{3})z^{-2}+(1-\sqrt{3})z^{-3})\frac{1}{4\sqrt{2}}$$

$$H(z)=(-(1-\sqrt{3})+(3-\sqrt{3})z^{-1}-(3+\sqrt{3})z^{-2}+(1+\sqrt{3})z^{-3})\frac{1}{4\sqrt{2}}$$

一种可行的实现是采用两个提升步骤和一个双重提升步骤。根据上面公式得到的产生双信道滤波器组的差分方程是：

$$h_1[n]=x[2n+1]-\sqrt{3}x[2n]$$

$$g_1[n]=x[2n]+\frac{\sqrt{3}}{4}h_1[n]+\frac{\sqrt{3}-2}{4}h_1[n-1]$$

$$h_2[n]=h_1[n]+g_1[n+1]$$

$$g[n]=\frac{\sqrt{3}+1}{\sqrt{2}}g_1[n]$$

$$h[n]=\frac{\sqrt{3}-1}{\sqrt{2}}h_1[n]$$

注意：

前面的信号抽取和输出分离成偶数$x[2n]$和奇数$x[2n-1]$的序列允许滤波器以 $2f_s$ 的速度运行。这种结构可以直接转换成硬件，并且可以用 MaxPlusII 实现(练习 5.10)，这种实现需要使用 5 个乘法器和 4 个加法器。重构滤波器组可以根据图形的互换来构造，差分方程的形式就是操作的反向和符号的变化。

Daubechies 等人[95]已经指出：任何(双)正交小波滤波器组都能够转换成一个提升和双重提升的步骤序列。所需要的乘法器和加法器的数量依赖于提升步骤的数量(步骤越多就越简单)，与直接多相实现相比，其乘法器和加法器的数量减少了 50%。这种方法在乘法器的位宽较小时似乎特别有前景[96]。但是另一方面，因为类似网格的结构不允许使用 RAG 技术，所以对较长的滤波器来讲，直接多相方法通常效率更高一些。

迄今为止，尽管讨论过的技术(多相分解和提升)已经提高了速度和规模，并且涵盖了所有类型的双信道滤波器，但是如果采用 QMF、线性相位或正交滤波器还是可以获得额外的节省。接下来就要讨论这些方法。

3. **QMF 实现**。对于 QMF，根据(5.54)我们就会发现：

$$h[n](-1)^n g[n] \circ - \bullet H(z) = G(-z) \tag{5.67}$$

这意味着与多相滤波器是一样的(符号除外)，也就是：

$$G_0(z) = H_0(z) \qquad G_1(z) = -H_1(z) \tag{5.68}$$

取代图 5-41 中的 4 个滤波器，在 QMF 中我们只需要 2 个滤波器和一个额外的蝶形”，如图 5-44 所示。这就节省了大约一半的成本。QMF 滤波器需要：

L 个实加法器　　L 个实乘法器 (5.69)

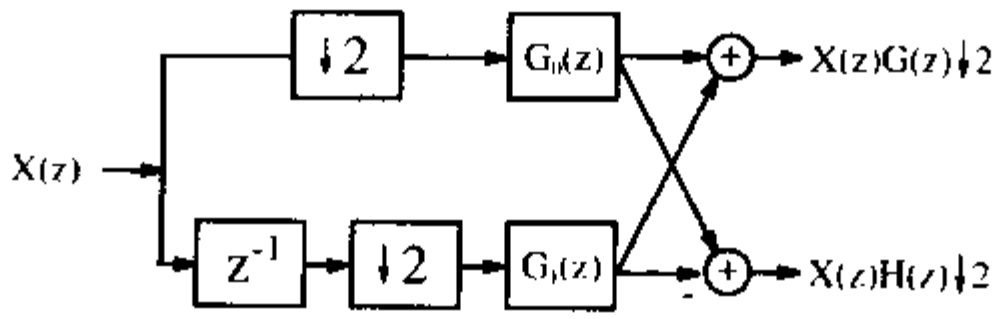


图 5-44　双信道 QMF 滤波器组的多相实现(©1999 Springer 出版社[51])

滤波器的运行速度可以是通常输入采样速率的 2 倍。

4. **正交滤波器组**。正交滤波器对[10]遵循共轭镜像滤波器(CQF)条件：

$$H(z) = z^{-N} G(-z^{-1}) \tag{5.70}$$

如果采用图 5-45 所示的转置 FIR 滤波器，就只需要一半数量的乘法器。不过，其缺点是不能够从多相分解中受益，也就是不能够将速度提高到两倍。

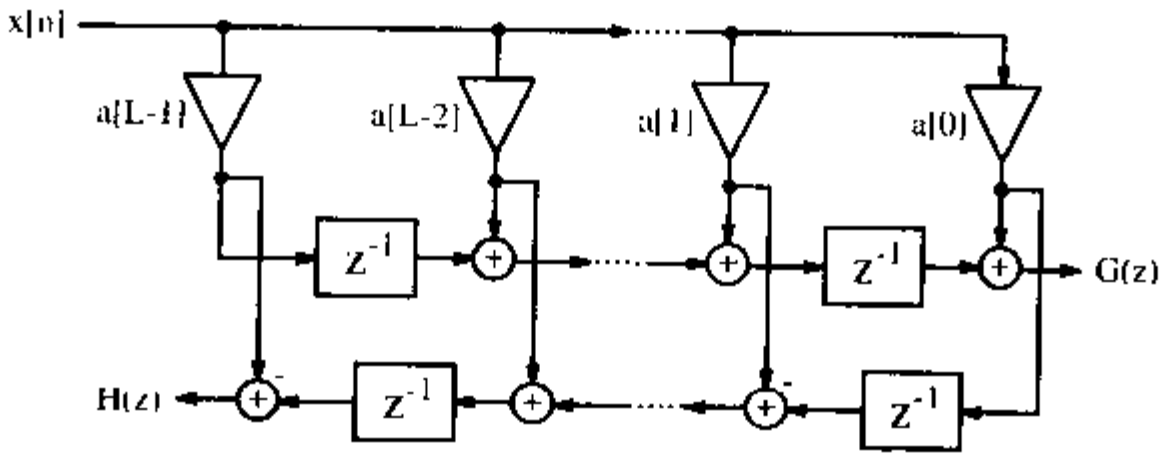


图 5-45　采用转置 FIR 结构的正交双信道滤波器组

实现 CQF 滤波器组的另一个选择方案就是采用图 5-46 所示的网格滤波器。下面的示例演示了直接 FIR 滤波器到网格滤波器的转换。

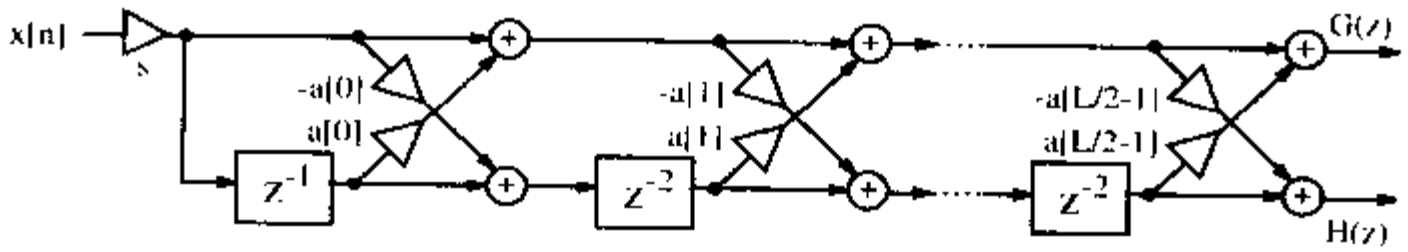


图 5-46　正交双信道滤波器组的网格实现(©1999 Springer 出版社[51])

注 10：正交滤波器的名称来源于滤波器的标量乘积相对于移位为 2(也就是$\sum g[k]h[k-2l]=0, k,l \in Z$)等于零这一事实。

例 5.18　网格 Daubechies L=4 滤波器的实现

例 5.15 中的滤波器结构是 Daubechies 长度为 4 的滤波器[94]。滤波器系数是：

$$G(z)=\frac{(1+\sqrt{3})+(3+\sqrt{3})z^{-1}+(3-\sqrt{3})z^{-2}+(1-\sqrt{3})z^{-3}}{4\sqrt{2}}$$
$$=0.48301+0.8365z^{-1}+0.2241z^{-2}-0.1294z^{-3} \tag{5.71}$$

$$H(z)=\frac{-(1-\sqrt{3})+(3-\sqrt{3})z^{-1}-(3+\sqrt{3})z^{-2}+(1+\sqrt{3})z^{-3}}{4\sqrt{2}}$$
$$=0.1294+0.2241z^{-1}-0.8365z^{-2}+0.48301z^{-3} \tag{5.72}$$

2 阶双信道网格滤波器的传递函数是：

$$G(z)=(1+a[0]z^{-1}-a[0]a[1]z^{-2}+a[1]z^{-3})s \tag{5.73}$$
$$H(z)=(-a[1]-a[0]a[1]z^{-1}-a[0]z^{-2}+z^{-3})s \tag{5.74}$$

现在，将(5.71)和(5.73)加以比较，就会得到：

$$s=\frac{1+\sqrt{3}}{4\sqrt{2}} \qquad a[0]=\frac{3+\sqrt{3}}{4\sqrt{2}s} \qquad a[1]=\frac{1-\sqrt{3}}{4\sqrt{2}s} \tag{5.75}$$

现在就可以直接将这种结构转换成硬件，下面给出了用 MaxPlusII 实现滤波器组的 VHDL 代码[11]。

```
PACKAGE n_bits_int IS                    -- User defined types
  SUBTYPE BITS8 IS INTEGER RANGE - 128 TO 127;
  SUBTYPE BITS9 IS INTEGER RANGE - 2**8 TO 2**8 - 1;
  SUBTYPE BITS17 IS INTEGER RANGE - 2**16 TO 2**16 - 1;
  TYPE ARRAY_BITS17_4 IS ARRAY (0 TO 3) OF BITS17;
END n_bits_int;

LIBRARY work;
USE work.n_bits_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY db4latti IS                              ------> Interface
  PORT (clk          : IN   STD_LOGIC;
        clk2         : OUT STD_LOGIC;
        x_in         : IN   BITS8;
```

注 11：这一例子相应的 Verilog 代码文件 db4latti.v 可以在附录 A 中找到。

```
        x_e, x_o      : OUT BITS17;
        g, h          : OUT BITS9);
END db4latti;

ARCHITECTURE flex OF db4latti IS

  TYPE STATE_TYPE IS (even, odd);
  SIGNAL state                      : STATE_TYPE;
  SIGNAL sx_up, sx_low, x_wait      : BITS17;
  SIGNAL clk_div2                   : STD_LOGIC;
  SIGNAL sxa0_up, sxa0_low          : BITS17;
  SIGNAL up0, up1, low0, low1       : BITS17;

BEGIN

  Multiplex: PROCESS ------> Split into even and odd
  BEGIN                          --      samples at clk rate
    WAIT UNTIL clk = '1';
    CASE state IS
      WHEN even =>
        -- Multiply with 256*s=124
        sx_up    <= 4 * (32 *    x_in - x_in);
        sx_low   <= 4 * (32 * x_wait - x_wait);
        clk_div2 <= '1';
        state <= odd;
      WHEN odd =>
        x_wait <= x_in;
        clk_div2 <= '0';
        state <= even;
    END CASE;
  END PROCESS;

--------- Multipy a[0] = 1.7321
  sxa0_up   <= (2*sx_up  - sx_up /4)
                                        - (sx_up /64 + sx_up/256);
  sxa0_low <= (2*sx_low - sx_low/4)
                                        - (sx_low/64 + sx_low/256);
--------- First stage -- FF in lower tree
  up0   <= sxa0_low + sx_up;
  LowerTreeFF: PROCESS
  BEGIN
    WAIT UNTIL clk_div2 = '0';
    low0 <= sx_low - sxa0_up;
  END PROCESS;
```

```
---------- Second stage   a[1]=0.2679
   up1   <= (up0 - low0/4) - (low0/64 + low0/256);
   low1  <= (low0 + up0/4) + (up0/64   +   up0/256);

   x_e   <= sx_up;     -- Provide some extra test signals
   x_o   <= sx_low;
   clk2  <= clk_div2;

   OutputScale: PROCESS
   BEGIN
      WAIT UNTIL clk_div2 = '0';
      g <=   up1 / 256;
      h <= low1 / 256;
   END PROCESS;

END flex;
```

这些 VHDL 代码是图 5-46 给出的网格滤波器的一个直接转换。输入流乘以 $s=0.48\approx 124/256$。接下来计算第一阶 $a[0]=1.73\approx(2-2^{-2}-2^{-6}-2^{-8})$的交叉项乘积。由此得出结论，在这一级，加法和底层树信号必然要延迟一个采样周期。在第二级，实现 $a[0]=0.27\approx(2^{-2}+2^{-6}+2^{-8})$的交叉乘法和最后的输出加法。这一设计使用了 334 个 LC，运行速度是 60.60MHz。图 5-47 给出了 VHDL 仿真结果。结果显示了对于幅值为 100 的脉冲分别在偶数和奇数位置滤波器 $G(z)$和 $H(z)$的响应。

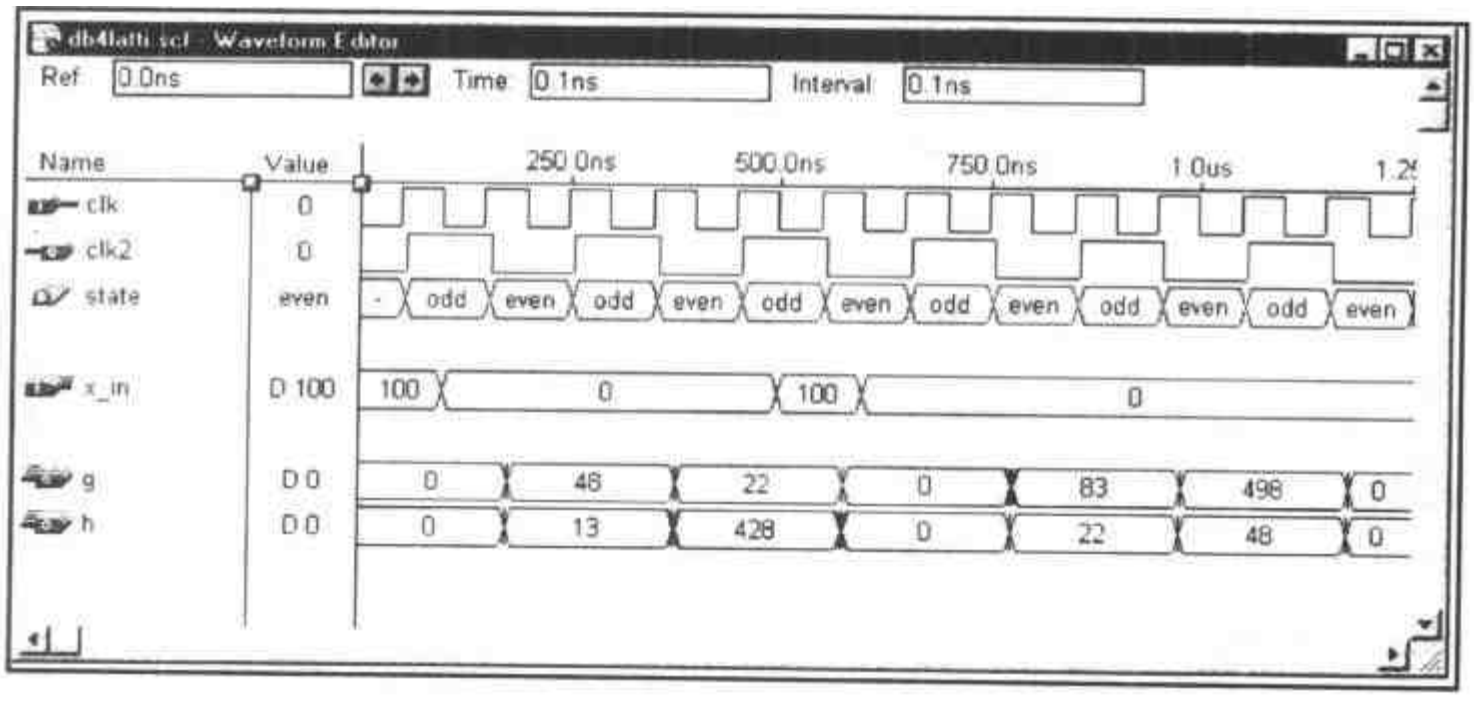


图 5-47　Daubechies 长度为 4 的网格滤波器组的 VHDL 仿真

如果将网格滤波器的规模与例 5.1 中给出的 $G(z)$的直接多相的实现(LC 数量乘以 2)加以比较，就会发现，两种设计的规模大致相同(208×2＝416 个 LC 和 334 个 LC)。尽管网格滤波器的实现只需要 5 个乘法器，与 8 个乘法器的多相实现相比，就可以看出在多相实现中我们能够使用 RAG 实现转置滤波器的系数，而在网格滤波器中，我们必须要实现单乘法器，一般情况下，单乘法器的效率较低。

5. 线性相位双信道滤波器组。在第 3 章中我们已经了解到：如果线性滤波器是偶对称或者

奇对称的，就可以节省 50%的乘法器资源。如果滤波器的长度为偶数，同样的对称也可以应用到滤波器的多相分解中。此外，这些滤波器的速度也可以提高一倍。

如果 $G(z)$和 $H(z)$具有相同的长度，使用网格滤波器的实现就会进一步减少实现的工作量，如图 5-48 所示。注意，在这里所使用的网格与图 5-46 给出的正交滤波器组中所使用的网格是不同的。

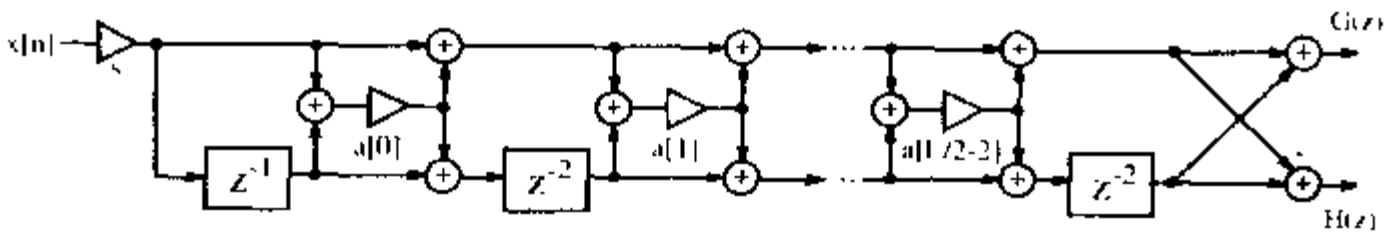


图 5-48　实现线性相位双信道滤波器组的网格滤波器(©1999 Springer 出版社[51])

下面的示例说明了如何将一个直接结构转换到网格滤波器中。

例 5.19　$L=4$ 线性相位滤波器的网格

例 5.15 中的滤波器结构是一个线性相位滤波器对，两个滤波器的长度均是 4，滤波器是：

$$G(z)=\frac{1}{4}(1+3z^{-1}+3z^{-2}+1z^{-3}) \tag{5.76}$$

$$H(z)=\frac{1}{4}(-1-3z^{-1}+3z^{-2}+1z^{-3}) \tag{5.77}$$

双信道长度 $L=4$ 的线性相位网格滤波器的传递函数是：

$$G(z)=((1+a[0])+a[0]z^{-1}+a[0]z^{-2}+(1+a[0])z^{-3})s \tag{5.78}$$

$$H(z)=(-(1+a[0])-a[0]z^{-1}+a[0]z^{-2}+(1+a[0])z^{-3})s \tag{5.79}$$

将(5.76)和(5.78)加以比较，就会得到：

$$s=-2 \qquad a[0]=-1,5 \tag{5.80}$$

线性相位网格滤波器的缺点就在于不是所有的线性相位滤波器都可以实现，具体地说就是 $G(z)$必须是偶对称，而 $H(z)$必须是奇对称，并且两个滤波器长度相同，且采样为偶数。

6. 实现选择的比较。最后，表 5-6 对不同的选择进行了比较，其中包括一般情形和具体类型，例如 QMF、线性相位和正交。

表 5-6 给出了需要的加法器和乘法器的数量、参考图、最大输入速率和构造上的重要问题：系数能否用 RAG 技术实现，或者作为单乘法器系数对于较短的滤波器，网格结构似乎比较有吸引力，而对于长滤波器，大多数情况下 RAG 可以生成更小更快的设计。注意：表 5-6 中乘法器和加法器的数量是对滤波器所需要的硬件工作量的一个估计，而不是文献中给出的在 PDSP/μP 解决方案[84, 99]中每次输入采样计算工作量的典型值。

其他关于双信道滤波器组的优秀文献还有很多(参阅[82, 99, 96, 100])，在此就不再赘述。

表 5-6　两个滤波器长度为 L 的双信道滤波器组的工作量

类　型	实乘法器数量	实加法器数量	参 考 图	速　度	是否可用 RAG
任意系数多相					
直接 FIR 滤波器	$2L$	$2L-2$	5.41	$2f_s$	√
提升	$\approx L$	$\approx L$	5.43	$2f_s$	—
正交镜像滤波器(QMF)					
恒等多相滤波器	L	L	5.44	$2f_s$	√
正交滤波器					
转置 FIR 滤波器	L	$2L-2$	5.45	f_s	√
网格	$L+1$	$3L/4$	5.46	$2f_s$	—
线性相位滤波器					
对称滤波器	L	$2L-2$	3.5	$2f_s$	√
网格	$L/2$	$3L/2-1$	5.48	$2f_s$	—

5.7　小波分析

通过转换方法处理的信号的时间频率表达式已经被证明对音频和图像处理【101，96，102】有利。对于短时帧(例如语音或音频信号)而言，许多从属于分析的信号都具有统计学上不变的属性，如图 5-49 所示。因此有必要在短窗函数内分析这些信号、计算信号参数并且向前滑动窗函数分析下一帧。如果这种分析是基于傅立叶变换的，就称之为**短项傅立叶变换**(short term Fourier transform, STFT)。

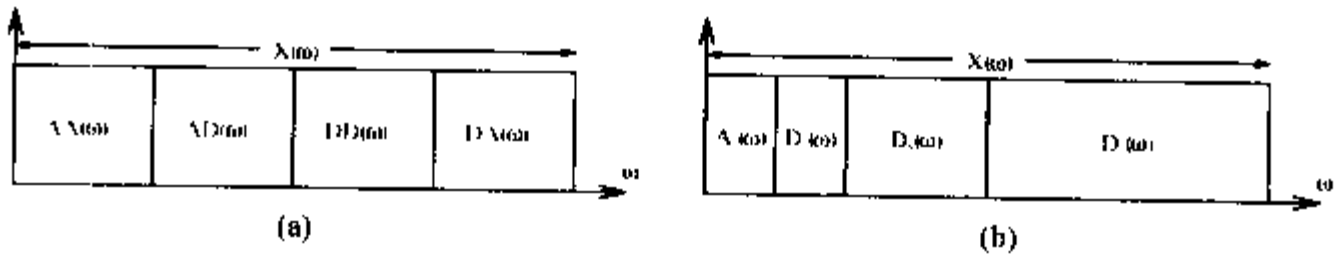


图 5-49　(a) 傅立叶(常系数带宽)和 (b) 常数 Q 的频率分布图

短项傅立叶变换的正式定义如下：

$$X(\tau,f)=\int_{-\infty}^{\infty}x(t)w(t-\tau)e^{-j2\pi ft}dt \tag{5.81}$$

也就是在信号 $x(t)$上滑动一个窗函数 $w(t-\tau)$，并且生成一系列时间频率映射。在频域和时域中，窗函数均会逐渐变小一直到 0，以确保映射在频率 Δf 和时间 Δt 之内的定位。在这种意义上，一种权重函数—— 高斯函数($g(t)=e^{-t^2}$)就是最佳选择，正如 Gabor 在 1949 年提议的，它能够提供最小的(Heisenberg 准则)乘积 $\Delta f\Delta t$ (也就是最优定位)[103]。Gabor 变换的离散化就产生了离散 Gabor 变换(discrete Gabor Transform, DGT)。Gabor 变换在整个时间和频率平面采用的是同一分辨率的窗口(请参阅图 5-50(a))。在图 5-50(a)中的每个长方形都有精度相同的形状，但是通常情况下，特别是在音频和图像处理领域中还需要一个常数 Q(也就是带宽与中心频率的商)。

换句话说，对于高频，我们希望有宽带滤波器和短的采样间隔，而对于低频，希望窄带宽和长的采样间隔。可以用 Morlet[104]提出的**连续小波变换**(continuous wavelet transform, CWT)来实现：

$$\mathrm{CWT}(\tau, f) = \int_{-\infty}^{\infty} x(t) h(\frac{t-\tau}{s}) dt \tag{5.82}$$

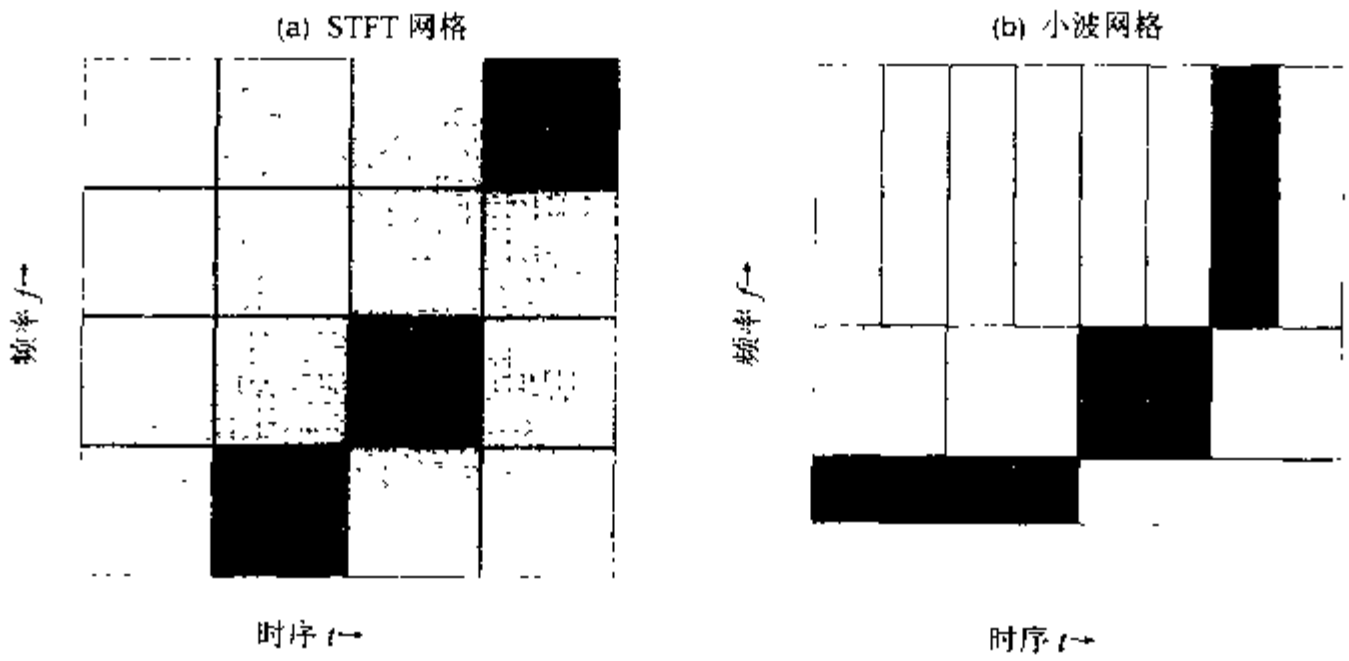


图 5-50　音频信号的时间频率栅格 (a) 短项傅立叶变换 (b) 小波变换

其中根据物理中的 Heugens 准则，$h(t)$称为**小波或子波**。图 5-51 给出了一些典型的小波。假定现在使用的小波是：

$$h(t) = (e^{j2\pi kt} - e^{k^2/2}) e^{-t^2/2} \tag{5.83}$$

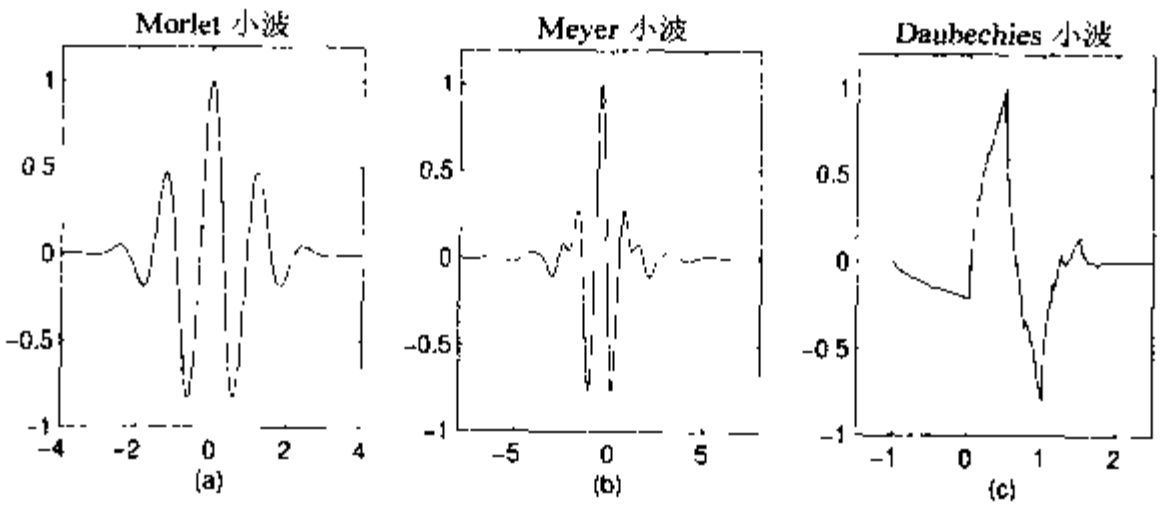


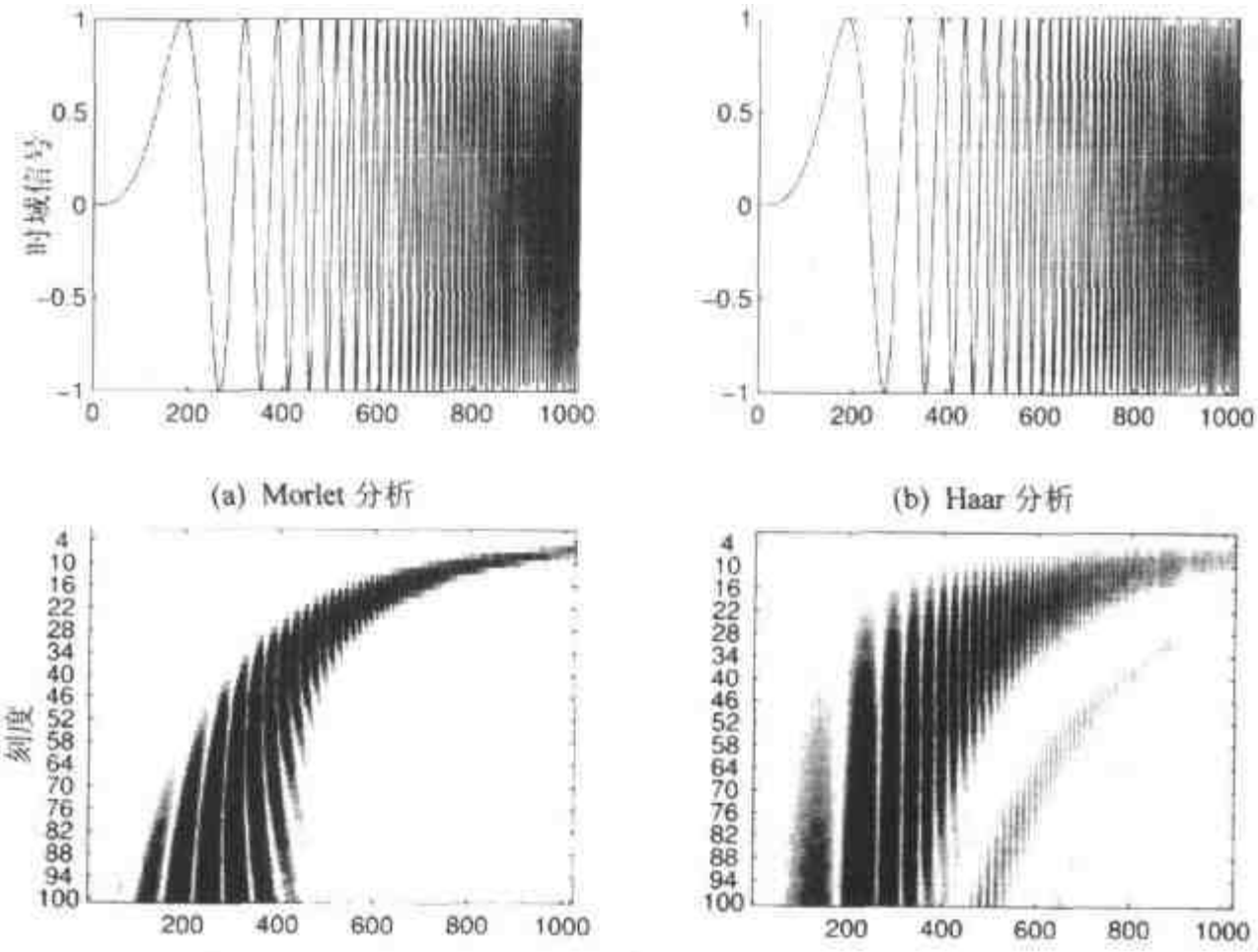
图 5-51　Morlet、Meyer 和 Daubechies 的一些典型小波

我们还是表情于 Gaussian 窗函数的“最优”特性，不过现在使用的是不同的时间和频率刻度。这就是 Morlet 变换，也从属于量化，因此称之为离散 Morlet 变换(discrete Morlet Transform, DMT)[105]。图 5-50(b)给出了在离散情形下时间和频率内的网格点。由于在(5.83)中引入了指数项$e^{-k^2/2}$，所以小波就不需要 DC 了。接下来的示例给出了 Gaussian 窗函数的优越性能。

例 5.20　线性调频信号的分析

图 5-52 给出了频率增加而幅值恒定的信号的分析。这样的信号就称为**线性调频信号**。如果采用傅立叶变换就会得到一个均匀的频谱，因为所有的频率都给出了，傅立叶频谱没有保留与

时间相关的信息。如果用带有 Gaussian 窗函数的 STFT，也就是 Morlet 变换代替它，就可以清晰地看到频率的增加，如图 5-52(a)所示。而且 Gaussian 窗函数给出了所有窗的最佳位置，另一方面，用 Haar 窗函数会减少计算的工作量，但是使用 Haar 窗函数会降低信号的时间-频率定位的精确度，如图 5-52(b)所示。



(a) Morlet 分析　　(b) Haar 分析

图 5-52　采用 (a) 离散 Morlet 变换、(b) Haar 变换的线性调频信号分析

DGT 和 DMT 都通过采用 Gaussian 窗函数提供了良好的定位，但是计算很密集。有一种基于这两种思路的无乘法器的实现。第一步， Gaussian 窗函数可以有效地近似成一个($\geq$3)的长方形函数的卷积，第二步是通过在多项式环上定义代数整数来有效地实现单通带**频率-采样滤波器**(frequency-sampling filter, FSF)，就如【105】中所介绍的一样。

接下来，我们要重点研究一种新近流行的分析方法，就是离散小波变换，这种方法更好地利用了听觉和视觉的人类感知模式(也就是常数 Q)，而且采用$O(n)$复杂算法还可以更加有效地进行计算。

离散小波变换

模拟模型的离散时间形式就产生了**离散小波变换**(discrete wavelet transform, DWT)。在实际应用中，DWT 受 $a=2$ 的离散时间二重 DWT 的限制，接下来我们就来研究这个问题。图 5-49(b)和图 5-50(b)给出的 DWT 实现的常数 Q，且带宽分布总是在滤波器树中对低通信号使用双信道滤波器组，如图 5-53 所示。

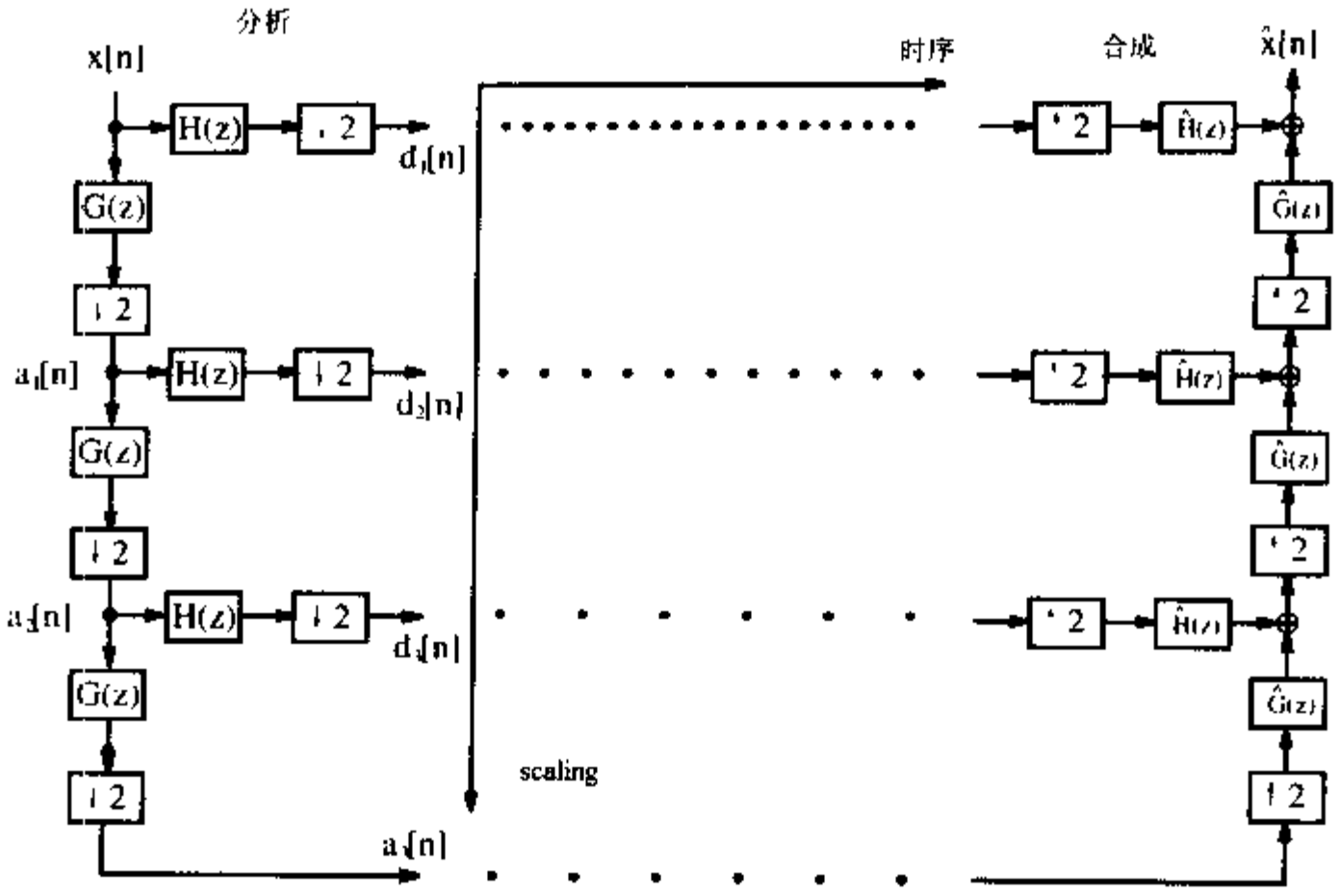


图 5-53　三倍频程的小波树分解(©1999 Springer 出版社[5])

现在要集中考虑在何种情况下可以用双信道 DWT 滤波器组实现 CWT 小波。我们认为，如果以合适的速率(高于奈魁斯特速率)对连续的小波进行采样，就可以称这种采样方式为 DWT。但是在一般情况下，只有那些可以用双信道滤波器组实现的连续小波变换才称作 DWT。

与一个连续小波 ψ (t)是否能够用双信道 DWT 实现密切相关的问题是刻度方程(scaling equation)：

$$\phi(t) = \sum_n g[n]\phi(2t - n) \tag{5.84}$$

是否存在，其中实际的小波是用：

$$\psi(t) = \sum_n h[n]\phi(2t - n) \tag{5.85}$$

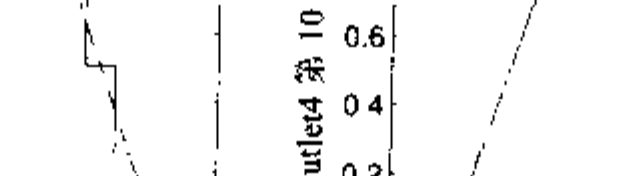
计算的。其中 $g[n]$是低通滤波器，而 $h[n]$是高通滤波器。要注意的是 $\phi(t)$和 $\psi(t)$是连续函数，而 $g[n]$和 $h[n]$是采样序列(当然也可以是 IIR 滤波器)。注意：(5.84)式与分形表现出的自相似性($\phi(t) = \phi(at)$)类似。实际上，刻度方程就可以迭代出分形，但是在一般情况下不用于设计，因为大多数情况下都是需要光滑的小波。如果采用零点的最大数量位于 π 处的滤波器，就可以提高光滑程度。

我们现在从前向后考虑重构问题：由滤波器 $g[n]$开始构造相应的小波。这是最常见的情形，特别是在采用算法 5.14 的半波带设计来生成所需要长度和性能的完美重构滤波器对时。

为了得到小波的图形化解释，可以由方波函数(逻辑框函数)开始，根据(5.84)构造下面的图形化迭代：

$$\phi^{(k+1)}(t)=\sum_n g[n]\phi^{(k)}(2t-n) \tag{5.86}$$

如果这种转换可以得到一个稳定的$\phi(t)$，则(新的)小波就找到了。因为两个逻辑框函数标定与相加的和仍然是逻辑框函数，也就是：



所以在第一次迭代之后，这种迭代就很明显地迅速收敛于 Haar 滤波器{1,1}。

现在我们用图形的方式构造属于滤波器 $g[n]=\{1,1,1,1\}$的小波，也称作 Hutlet4 滤波器[106]。

例 5.21　长度为 4 的 Hutlet 滤波器

我们首先介绍由经过$g[n]=\{1,1,1,1\}$加权的逻辑框函数，图 5-54(a)给出了$\phi^{(1)}(t)$起始点的和。此函数的刻度是 2，该和给出了一个 2 阶函数。在 10 次迭代之后就可以得到一个非常光滑的梯形函数。如果现在利用(5.54)中的 QMF 关系构造实际的小波，就可以得到有两个三角形的 Hutlet4 滤波器。

图 5-54　Hutlet4 的迭代步骤 1,2,3 和 10(实线：$\phi^{(k+1)}(t)$；点虚线：$\phi^{(k)}(2t-n)$；短划线：理想的 Hut-函数)

可以看到 $g[n]$是移动平均滤波器的脉冲响应，可以用一个一阶 CIC 滤波器[107]。图 5-55 给出了这种带有偶数长度系数类型小波的所有刻度的函数和小波。

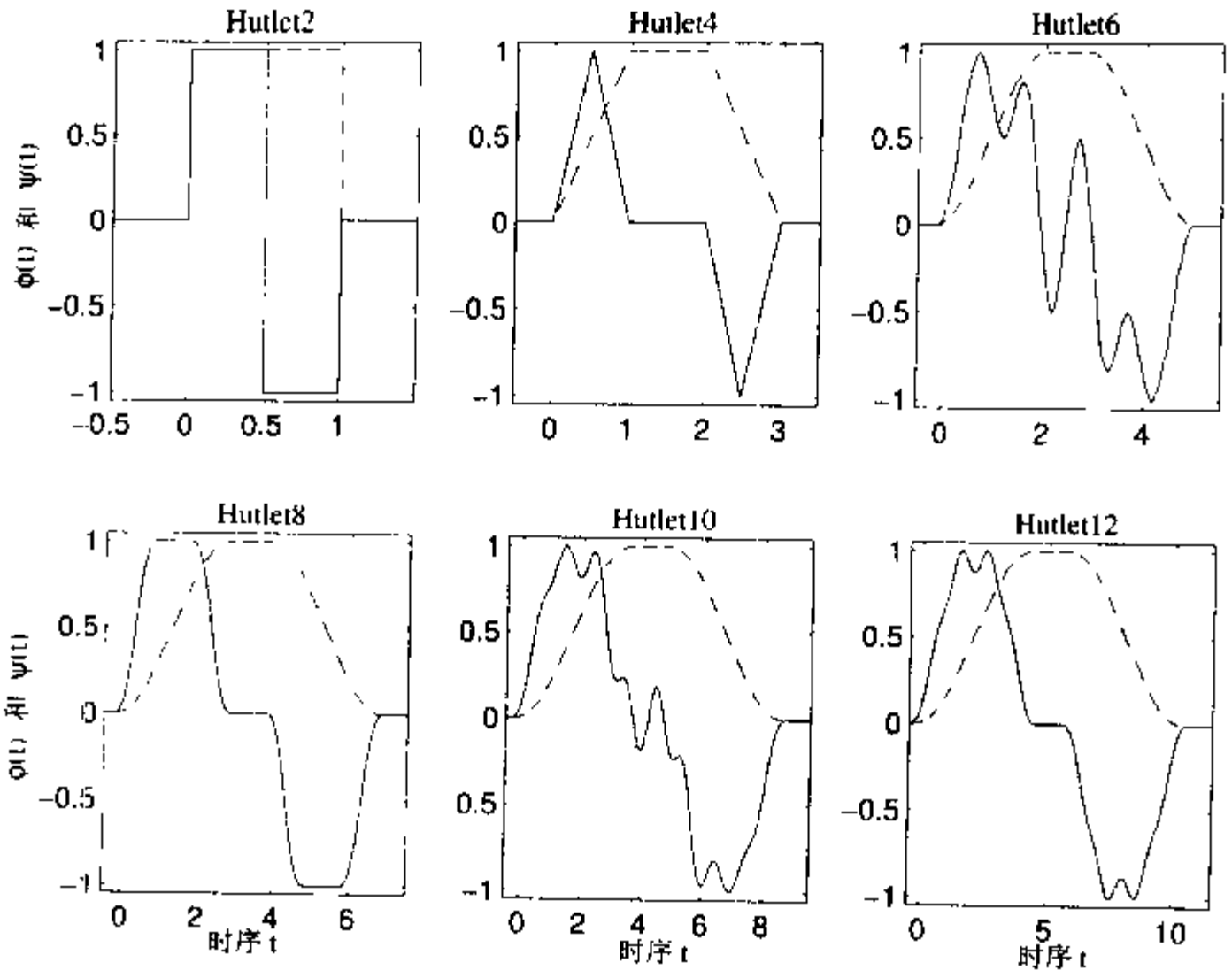


图 5-55　10 次迭代后的 Hutlet 小波系列(实线)和刻度函数(短划线) (©1999 Springer 出版社[51])

正像前面所提及的，(5.86)定义的迭代也收敛于某个分形。图 5-56 给出了相应的示例，这是长度为 5 的移动平均滤波器的小波。这个例子简要地说明了滤波器 $g[n]$选择的复杂性：它可以收敛成一个光滑的或者不规则的函数，这仅取决于看起来不重要的性质，比如滤波器的长度！

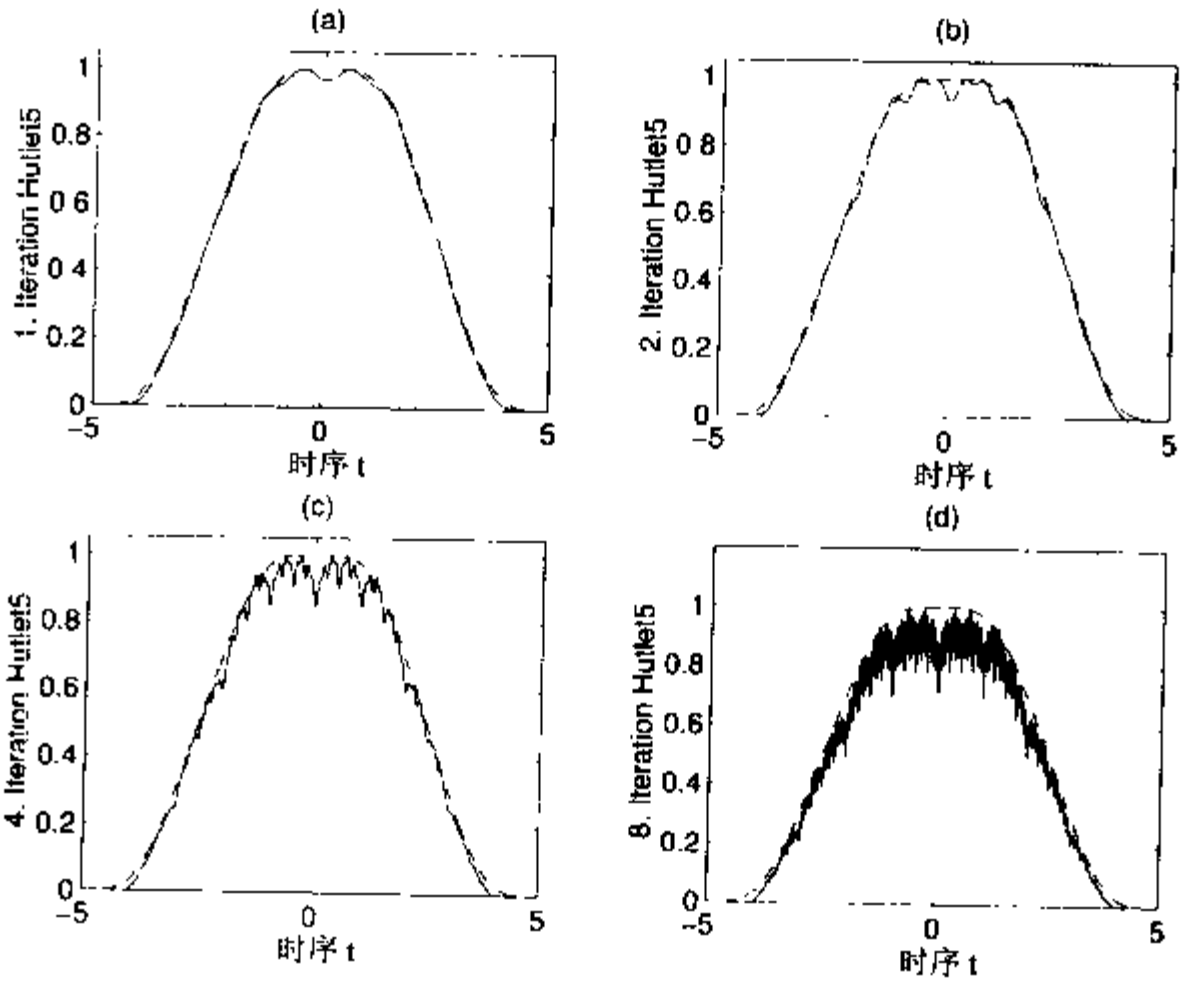


图 5-56　Hutlet5 的迭代步骤 1,2,4 和 8 序列集中到一个分形

到目前为止，还没有解释为什么 2-刻度方程(5.84)对 DWT 如此重要呢。如果在 DWT 的分析部分利用 5.1.1 提到的“Noble”关系：

$$(\downarrow M)H(z) = H(z^M)(\downarrow M) \tag{5.87}$$

重新组织向下采样(压缩)和滤波器，就可以很容易理解这一点了。图 5-57 给出了三阶滤波器的结果。如果计算级联序列，也就是：

$$H(z) \leftrightarrow d_1[k/2]$$
$$G(z)H(z^2) \leftrightarrow d_2[k/4]$$
$$G(z)G(z^2)H(z^4) \leftrightarrow d_3[k/8]$$
$$G(z)G(z^2)G(z^4) \leftrightarrow a_3[k/8]$$

的脉冲响应，将图形与图 5-51 给出的连续小波进行比较，就会看到 a_3 是对刻度函数的一种近似，而 d_3 则是对母小波的一种近似。

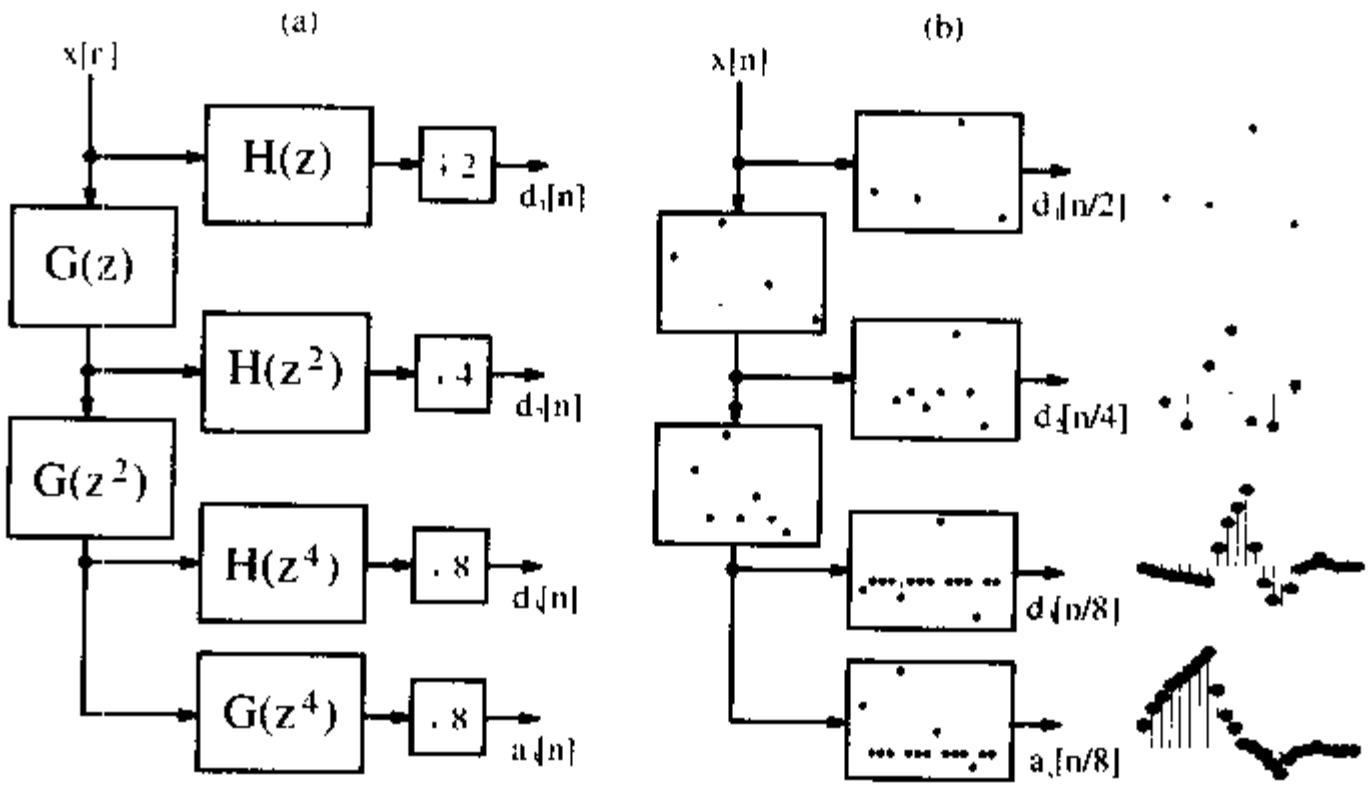


图 5-57　用 Noble 关系式重构 DWT 滤波器组　(a) z 域内的传递函数

(b) 长度为 4 的 Daubechies 滤波器的脉冲响应

不过这也不总是可行的。例如：对于图 5-51 给出的 Morlet 小波，就找不到相应的刻度函数，也就不可能用 DWT 实现了。

Daubechies 长度为 4 的滤波器的双信道 DWT 设计示例已经在多相表达式(例 5.1)和正交滤波器网格实现(例 5.18)的组合里讨论过了。

5.8　练习

5.1：令 $F(z) = 1 + z^{-d}$，根据定义 5.7，d 为何值可以得到半波带滤波器？

5.2：令 $F(z) = 1 + z^{-5}$ 是一个半波带滤波器。

(a) 绘出 $|F(\omega)|$。这一滤波器具有什么样的对称性？

(b) 用算法 5.14 计算完美重构的实数滤波器组。滤波器组的总延迟是多少?

5.3：用例 5.15 中的半波带滤波器 F3 构造完美重构的滤波器组，分别使用长度为：

(a) 1/7

(b) 2/6 的算法 5.14。

5.4：如果两个滤波器分别是：

(a) 复数

(b) 实数

(c) 线性相位

(d) 正交滤波器组

用例 5.15 的 F3 滤波器可以构造多少种不同的滤波器对?

5.5：用半波带滤波器 $F2(z)=1+z^{-1}+z^{-2}$ 计算基于算法 5.14 的所有可能的完美重构的滤波器组。

5.6：(a) 计算图 5-33 中严格采样均匀 DFT 滤波器实现的实数加法和乘法的数量。假定长度为 L 的分析和合成滤波器具有复数系数，且输入是实数值。

(b) 假定某种 FFT 算法需要$(15N\log_2(N))$次实数加法和乘法。使用图 5-35 和图 5-36 中长度为 R 的 L 个复数滤波器的多相表达式，计算均匀 DFT 滤波器组的总工作量。

(c) 用(a)和(b)的结果计算 $L=64$ 和 $R=16$ 的严格采样 DFT 滤波器组的工作量。

5.7：用例 5.9 的有耗积分器实现一个 $R=4$ 的均匀 DFT 滤波器组。

(a) 计算分析多相滤波器 $H_k(z)$。

(b) 为完美重构确定合成滤波器 $F_k(z)$。

(c) 确定 4×4DFT 矩阵。计算 DFT 需要多少次实数加法和乘法?

(d) 按照每次输入采样的实数加法和乘法来计算整个滤波器组的总计算量。

5.8：分析表 5-3 中每个 Goodman 和 Carey 半波带滤波器的频率响应。放大通频带以消除滤波器的纹波。

5.9：证明(5.65)和(5.66)中提升和双重提升结构的完美重构。

练习使用 MaxPlusII

5.10：(a)用例 5.17 中的提升结构实现 Daubechies 长度为 4 的滤波器，8 位的输入和系数，9 位的输出量化。

(b) 用类似图 5-47 的两个幅值为 100 的脉冲，对设计进行仿真。

(c) 确定 LC 的使用量和 Registered Performance。

(d) 就速度和规模方面比较直接多相实现(例 5.1)和网格实现(例 5.18)的提升设计。

5.11：用例 5.4 和例 5.6 的两个设计组件实例计算两个滤波器输出的不同，确定最大正偏差和负偏差。

5.12：(a) 用图 3.11 的简化加法器图设计一个利用 MaxPlusII 构造的 8 位输入的半波带滤波器 F6(请参阅表 5-3)。用转置 FIR 结构(图 3.3)作为滤波器的体系结构。

(b) 确定 F6 设计以 LC 表示的规模和 Registered Performance。

5.13：(a) 计算表 5-3 中 F6 滤波器的多相表达式。

(b) 用 MaxPlusII 实现 8 位输入的多相滤波器 F6，抽取 $R=2$。

(c) 确定多相设计以 LC 表示的规模和 Registered Performance。

(d) 就速度和规模方面与练习 5.12 中的直接实现相比较，多相设计的优缺点是什么？

第6章　傅立叶变换

离散傅立叶变换(Discrete Fourier Transform, DFT)及其快速实现，即快速傅立叶变换(Fast Fourier Transform，FFT)，在数字信号处理中扮演着重要的角色。

目前已经以多种形式发明(和再发明)了多种 DFT 和 FFT 算法。正如 Heideman 等人[108]所指出的，我们知道高斯就用过一种我们今天称之为 Cooley-Tukey FFT 的 FFT 类型算法。在本章中，我们将要简要地讨论图 6-1 中总结的最重要的算法。

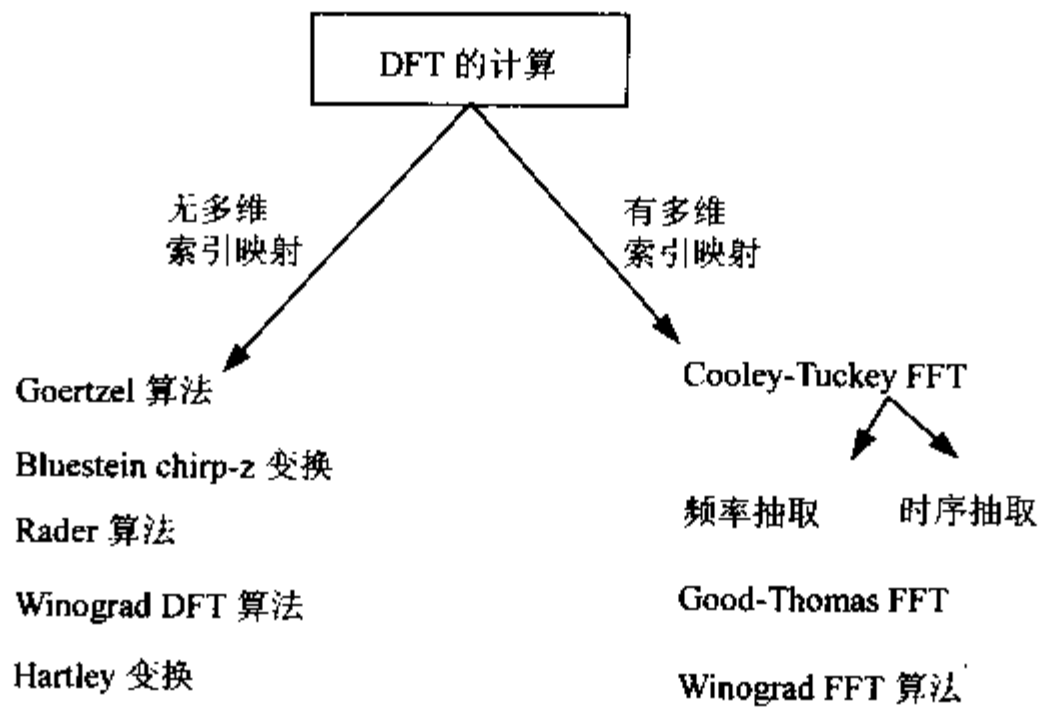


图 6-1　DFT 和 FFT 算法的分类

在此要沿用 Burrus【109】提出的术语学体系，Burrus 简单地根据 FFT 算法的输入输出序列之间的(多维)索引映射关系对之进行了分类。所以我们将所有(没有使用多维索引映射的)算法都称为 DFT 算法，尽管其中一些算法具有非常简单的计算量，如 Winograd DFT 算法。DFT 和 FFT 算法不是“独立”的：大多数算法的有效实现通常都是 DFT 和 FFT 算法组合的结果。例如：Rader 质数算法和 Good-Thomas FFT 的组合就产生了著名的 VLSI 实现。该文献提供了许多 FFT 设计的示例。我们发现，用 PDSP 和 ASIC 的 FFT 实现[110,111,112,113,114,115]已经发展到可以用 FPGA 实现一维[116,117,118]和二维[42,119]变换了。

在本章将要讨论 4 种最重要的 DFT 算法和 3 种最常用的 FFT 算法，并且依照计算量比较不同的实现结果。在本章的结尾，还要讨论傅立叶相关的变换，例如：DCT，它是图像压缩的一种重要工具(例如 JPEG、MPEG)。首先来简要地复习一下 DFT 的定义和一些重要属性。

如果要进一步详细地研究，还应该了解 DFT 算法。在基础 DSP 书籍[120,65,121,5]和多种 FFT 书籍[122,123,124,125,126,54]中都可以找到对 DFT 算法的详尽叙述。

6.1　离散傅立叶变换算法

我们首先来复习一下 DFT 的一些最重要的属性，然后复习 Bluestein、Goertzel、Rader 和 Winograd 提出的一些 DFT 基础算法。

6.1.1　用 DFT 近似傅立叶变换

傅立叶变换对的定义如下：

$$X(f)=\int_{-\infty}^{\infty}x(t)e^{-j2\pi ft}\mathrm{d}t \leftrightarrow x(t)=\int_{-\infty}^{\infty}X(f)e^{j2\pi ft}\mathrm{d}f \tag{6.1}$$

公式假定了一个无限持续时间和带宽的连续信号。对于实际的表达式还需要在时间和频率上采样，并且对幅值进行量化。从实现的角度来讲，我们更希望在时间和频率上使用有限数量的采样。这样就产生了离散傅立叶变换(discrete Fourier Transform, DFT)，其中在时间和频率上采用了 N 次采样，根据：

$$X[k]=\sum_{n=0}^{N-1}x[n]e^{-j2\pi kn/N}=\sum_{n=0}^{N-1}x[n]W_N^{kn} \tag{6.2}$$

离散傅立叶反变换定义如下：

$$x[n]=\frac{1}{N}\sum_{k=0}^{N-1}X[k]e^{j2\pi kn/N}=\frac{1}{N}\sum_{k=0}^{N-1}X[k]W_N^{-kn} \tag{6.3}$$

或者用向量/矩阵表示，就是：

$$\boldsymbol{X}=\boldsymbol{W}\boldsymbol{x} \leftrightarrow \boldsymbol{x}=\frac{1}{N}\boldsymbol{W}^{*}\boldsymbol{X} \tag{6.4}$$

如果用 DFT 对傅立叶频谱进行近似，就必须记住在时间和频率上采样的影响，分别是：

● 通过在时域上的采样，可以得到采样频率为 f_S 的周期性频谱。正如“Shannon 采样定理”所陈述的：只有在 $x(t)$ 的频率成份集中在一个低于奈魁斯特频率 $f_S/2$ 的狭窄范围内的情况下，用 DFT 近似傅立叶变换才是合理的。

● 通过在频域上的采样，时间函数就变成了周期性的，也就是说 DFT 假定时间序列是周期性的。如果对一个信号采用 N 次采样 DFT，没有在一个 N 次采样窗函数内完成整数个循环，就会产生一种称为泄漏的现象。所以，如果可能的话，并且 $x(t)$ 是周期性信号，就应该选择可以覆盖整数个 $x(t)$ 的周期采样频率和分析函数。

一种更为实用的降低泄漏的选择方案就是采用在两边逐渐衰减为 0 的窗函数。这类窗函数已经在第 3 章的 FIR 滤波器设计中讨论过了(请参阅表 3.2)。图 6-2 给出了一些典型窗函数的时间和频率特性[87, 127]。

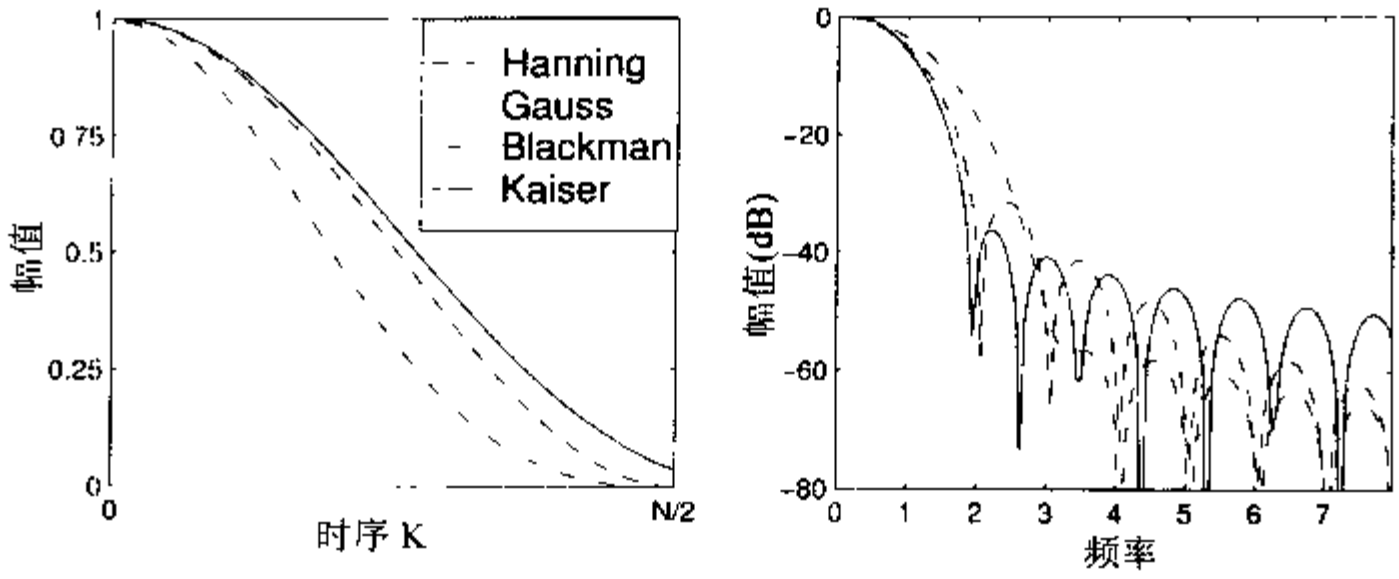


图 6-2　时域和频域内的窗函数

下面的一个示例说明窗函数的使用。

例 6.1　开窗操作

图 6-3(a)给出了在其采样窗口内不能完成的整数周期的正弦信号。该信号理想的傅立叶变换应该只包括两个在$\pm\omega_0$处的单位脉冲函数，如图 6-3(b)所示。图 6-3(c)和 6-3(d)分别给出了用不同窗函数进行 DFT 分析的结果。可以看到，用逻辑框函数的分析比用 Hanning 窗函数的分析多一些纹波。精确的分析也表明，用 Hanning 分析的主平稳宽度要大于用逻辑框函数(也就是无窗函数)分析所达到的宽度。

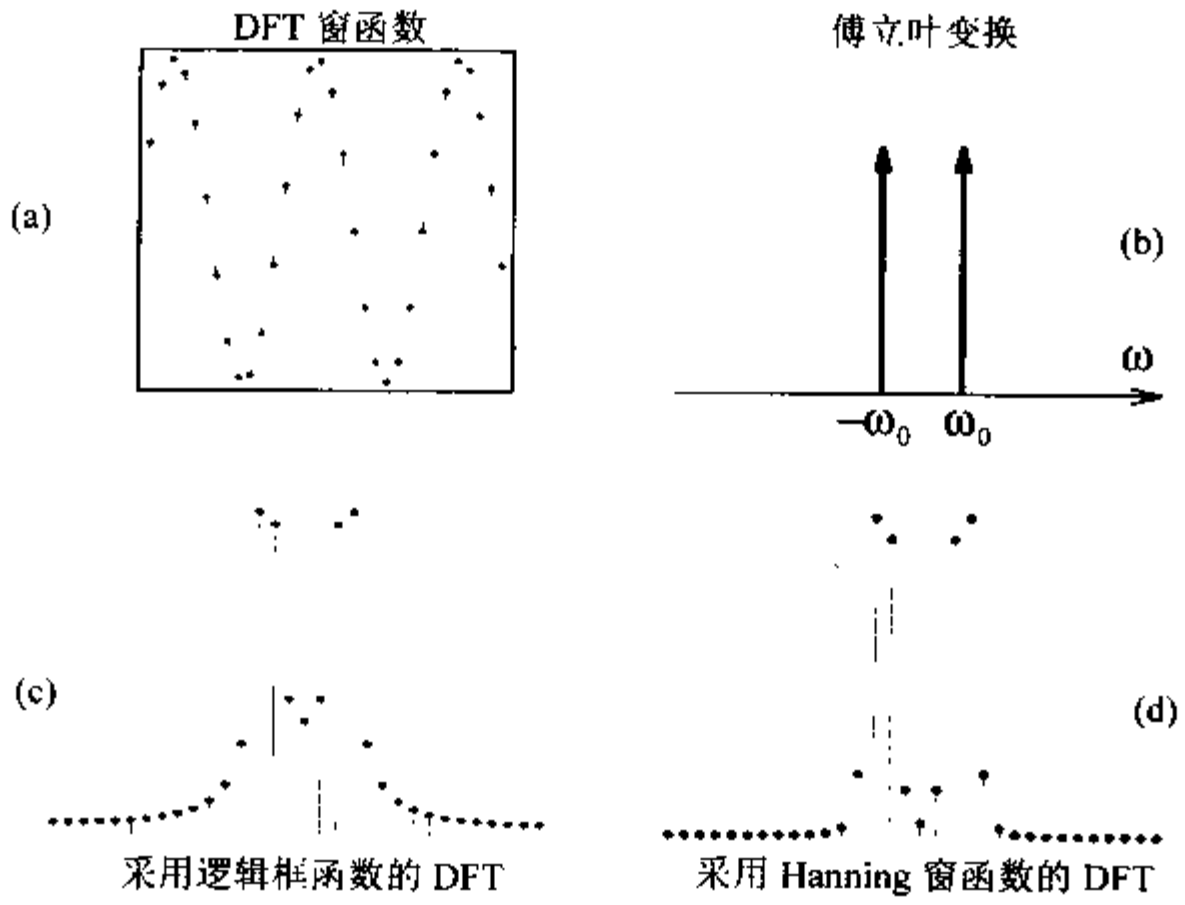


图 6-3　利用窗函数分析通过 DFT 的周期函数

6.1.2　DFT 的属性

表 6-1 总结了 DFT 最重要的属性。许多属性与傅立叶变换一致，例如：变换是惟一(双射)的、重叠的使用，以及实部与虚部通过 Hilbert 变换联系在一起。

前向和反向变换的相似性产生了一种可选的反演算法。利用 DFT 的向量/矩阵表示式：

$$\boldsymbol{X}=\boldsymbol{W}\boldsymbol{x}\leftrightarrow\boldsymbol{x}=\frac{1}{N}\boldsymbol{W}^{*}\boldsymbol{X} \tag{6.5}$$

可以得到：

$$\boldsymbol{x}=\frac{1}{N}(\boldsymbol{W}^{*}\boldsymbol{X})^{*}=\boldsymbol{W}\boldsymbol{X}^{*} \tag{6.6}$$

也就是可以利用刻度为 $1/N$ 的 $\boldsymbol{X}^{*}$的 DFT 计算离散傅立叶反变换。

表 6-1　DFT　定　理

定　　理	$x(n)$	$X(k)$
变换	$x(n)$	$\sum_{n=0}^{N-1}x[n]e^{-j2\pi nk/N}$
反变换	$\frac{1}{N}\sum_{n=0}^{N-1}X[k]e^{j2\pi nk/N}$	$X[k]$
重叠	$s_1x_1[n]+s_2x_2[n]$	$s_1X_1[k]+s_2X_2[k]$
时间反向	$x[-n]$	$X[-k]$
共轭复数拆分	$x^{*}[n]$	$X^{*}[-k]$
实部	$\Re(x[n])$	$(X[k]+X^{*}[-k])/2$
虚部	$\Im(x[n])$	$(X[k]+X^{*}[-k])/(2j)$
实偶数部分	$x_e[n]=(x[n]+x[-n])/2$	$\Re(X[k])$
实奇数部分	$x_o[n]=(x[n]-x[-n])/2$	$j\Im(X[k])$
对称性	$X[n]$	$Nx[-k]$
循环卷积	$x[n]\circledcirc f[n]$	$X[k]F[k]$
乘法	$x[n]\cdot f[n]$	$\frac{1}{N}X[k]\circledcirc F[k]$
周期平移	$x[n-d \bmod N]$	$X[k]e^{-j2\pi dk/N}$
Parseval 定理	$\sum_{n=0}^{N-1}\lvert x[n]\rvert^2$	$\frac{1}{N}\sum_{k=0}^{N-1}\lvert X[k]\rvert^2$

1. 实序列的 DFT

现在来研究一下当输入序列是实数时，一些 DFT(和 FFT)计算的额外简化计算。在这种情况下，我们有两种选择：一种是可以用一个 N 点 DFT 计算两个 N 点序列的 DFT；另一种是可以用一个 N 点 DFT 计算一个长度为 $2N$ 的实序列的 DFT。

如果利用表 6-1 给出的 Hilbert 属性，也就是实序列具有偶对称的实频谱和奇对称的虚频谱，下面的算法就可以合成起来[1122]。

算法 6.2　用一个 N 点 DFT 计算长度为 $2N$ 的 DFT

从时间序列 $x[n]$计算 $2N$ 点 DFT $X[k]=X_r[k]+jX_i[k]$的算法如下：

(1) 构造一个 N 点序列 $y[n]=x[2n]+jx[2n+1]$，其中 $n=0,1,\ldots N-1$。

(2) 计算 $y[n]\circ-\bullet Y[k]=Y_r[k]+jY_i[k]$，其中 $\Re(Y[k])=Y_r[k]$ 和 $\Im(Y_i[k])=Y_i(k)$ 分别是 $Y[k]$的实部和虚部。

(3) 计算
$$X_r[k]=\frac{Y_r[k]+Y_r[-k]}{2}+\cos(\pi k/N)\frac{Y_i[k]+Y_i[-k]}{2}-\sin(\pi k/N)\frac{Y_r[k]-Y_r[-k]}{2}$$
$$X_i[k]=\frac{Y_i[k]-Y_i[-k]}{2}-\sin(\pi k/N)\frac{Y_i[k]+Y_i[-k]}{2}-\cos(\pi k/N)\frac{Y_r[k]-Y_r[-k]}{2}$$

其中 $k=0,1,\ldots N-1$。

因此，除了一个 N 点 DFT(或 FFT)之外的计算量就是来自旋转因子$\pm\exp(j\pi k/N)$的 $4N$ 次实数加法和乘法。

为了用一个长度为 N 的 DFT 变换两个长度为 N 的序列，我们可以运用实序列具有一个偶频谱而纯虚序列的频谱为奇数这一事实(请参阅表 6-1)。这也是下面算法的基础。

算法 6.3　用一个 N 点 DFT 计算两个长度为 N 的 DFT

计算 N 点 DFT $g[n]\circ-\bullet G[k]$和$h[n]\circ-\bullet H[k]$的算法如下：

(1) 构造一个 N 点序列 $y[n]=h[n]+jg[n]$，其中 $n=0,1,\ldots N-1$。

(2) 计算 $y[n]\circ-\bullet Y[k]=Y_r[k]+jY_i[k]$，其中 $\Re(Y[k])=Y_r[k]$ 和 $\Im(Y_i[k])$分别是 $Y[k]$的实部和虚部

(3) 最后计算
$$H[k]=\frac{Y_r[k]+Y_r[-k]}{2}+j\frac{Y_i[k]-Y_i[-k]}{2}$$
$$G[k]=\frac{Y_i[k]+Y_i[-k]}{2}-j\frac{Y_r[k]-Y_r[-k]}{2}$$

其中 $k=0,1,\ldots N-1$。

因此，除了一个 N 点 DFT(或 FFT)之外的计算量就是为了构成正确的 2 个 N 点 DFT 而进行的 $2N$ 次实数乘法。

2. 利用 DFT 的快速卷积

最常用到 DFT(或 FFT)的一个领域就是计算卷积。如同傅立叶变换一样，时域内的卷积就是将两个变换的序列相乘：两个时间序列在频域内变换，计算一个(标量)逐点乘积，再将结果返回到时域中。与傅立叶变换相比，主要的区别是 DFT 计算了一个循环的而不是线性的卷积。这一点在用 FFT 实现快速卷积的时候必须加以考虑。也就产生了两种方法，分别是“重叠节约”和“重叠增加”。在重叠节约方法中，我们只需要简单地放弃在边界被循环卷积破坏的采样即可。在重叠增加方法中，通过在公共乘积流上直接加上部分序列的方法在滤波器和信号中填充 0。

对于快速卷积而言，最常见的输入序列都是实序列。因此，有效卷积可以用实变换来实现，例如将要在练习 6.16 中讨论的 Hartley 变换。我们还可以为 Hartley 变换构造一个类似 FFT 的算法，与复变换[128]相比较，可以将性能提高两倍。

如果要利用可行的 FFT 程序，我们需要使用前面讨论过的实序列算法 6.2 或算法 6.3。图 6-4 给出了一个与算法 6.2 相似的可选方法，该方法用一个 N 点 DFT 来实现两个 N 点变换，不过在这种情况下，实部用作 DFT，虚部用作 IDFT，根据卷积理论，在反变换的时候需要用到虚部。

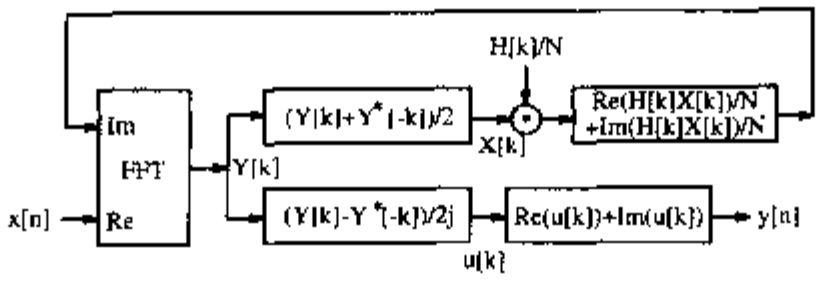


图 6-4　采用复数 FFT 的实数卷积[54]

假设实数值滤波器(也就是 $F[k]=F[-k]^*$)的 DFT 已经被离线计算过，那么在频域内就只需要 $N/2$ 次乘法来计算 $X[k]F[k]$。

6.1.3　Goertzel 算法

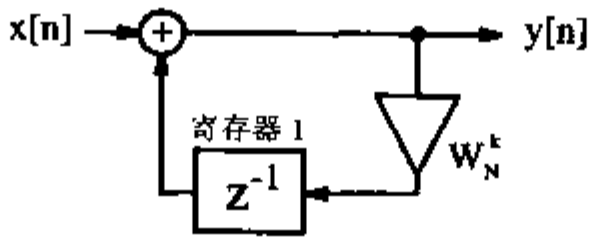
DFT 计算中的单个频谱成份是由：

$$X[k]=x[0]+x[1]W_N^k+x[2]W_N^{2k}+\mathrm{K}+x[N-1]W_N^{(N-1)k}$$

给出的。我们将所有的 $x[n]$用同一个公共因子 W_N^k 组合起来，就得到：

$$X[k]=x[0]+W_N^k(x[1]+W_N^k(x[2]+\mathrm{K}+W_N^k x[N-1])\mathrm{K})$$

可以看到这一结果是 $X[k]$的可行递归计算。这就是 Goertzel 算法，图 6-5 给出了相应的图形化解释。$y[n]$的计算由输入序列的最后一个值 $x[N-1]$开始。在步骤 3 之后，$X[k]$的一个频谱值就在输出端给出了。



Step	$x[n]$	Register 1	$y[n]$
0	$x[3]$	0	$x[3]$
1	$x[2]$	$W_4^k x[3]$	$x[2]+W_4^k x[3]$
2	$x[1]$	$W_4^k x[2]+W_4^{2k} x[3]$	$x[1]+W_4^k x[2]+W_4^{2k} x[3]$
3	$x[0]$	$W_4^k x[1]$ $+W_4^{2k} x[2]+W_4^{3k} x[3]$	$x[0]+W_4^k x[1]$ $+W_4^{2k} x[2]+W_4^{3k} x[3]$

图 6-5　长度为 4 的 Goertzel 算法

如果已经计算了几个频谱成份，将 $e^{\pm j2\pi n/N}$ 类型的因子组合就会降低复杂程度。这样就根据练习 6.4 中讨论的：

$$z^2-2z\cos(\frac{2\pi n}{N})+1$$

得到一个有分母的二阶系统。这样，所有的复数乘法就都简化成实数乘法了。

一般情况下，如果只有少量频谱成份需要计算的话，Goertzel 算法是很有吸引力的。对于整个 DFT 而言，计算量是 N^2 量级的，与直接 DFT 计算相比较就没有优势可言了。

6.1.4　Bluestein Chirp-z 变换

在 Bluestein Chirp-z 变换(CZT)算法中，DFT 指数 nk 可以量化展开成：

$$nk = -(k-n)^2/2 + n^2/2 + k^2/2 \tag{6.7}$$

因此 DFT 就变成了：

$$X[k] = W^{k^2/2}\sum_{n=0}^{N-1}(x[n]W^{n^2/2})W^{-(k-n)^2/2} \tag{6.8}$$

图 6-6 给出了算法的图形化解释。由此可以得到：

算法 6.4　Bluestein Chirp-z 算法

DFT 的计算分为 3 个步骤，分别是：

(1)　$x[n]$与$W_N^{n^2/2}$的 N 次乘法。

(2)　$x[n]W_N^{n^2/2} * W_V^{n^2/2}$的线性卷积。

(3)　$W_N^{k^2/2}$的 N 次乘法。

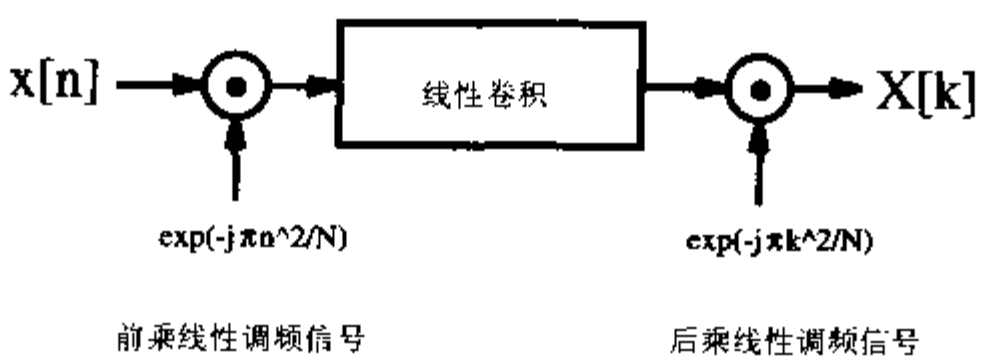


图 6-6　Bluestein Chirp-z 算法

要完成一次变换，就需要一个长度为 N 的卷积和 $2N$ 次复数乘法。与 Rader 算法相比较，其优点是变换长度 N 不需要限制在质数范围内。CZT 可以定义成任意长度。

Narasinha[129] 和其他人已经注意到，在 CZT 算法中，FIR 滤波器部分的许多系数是无关紧要或是相同的。例如：长度为 8 的 CZT 的 FIR 滤波器长度为 16，但是在图 6-7 中只给出了 4 个不同的复数系数。这 4 个系数分别是 $1, j$ 和 $\pm e^{22.5°}$，也就是只需要实现两个不可或缺的实系数。

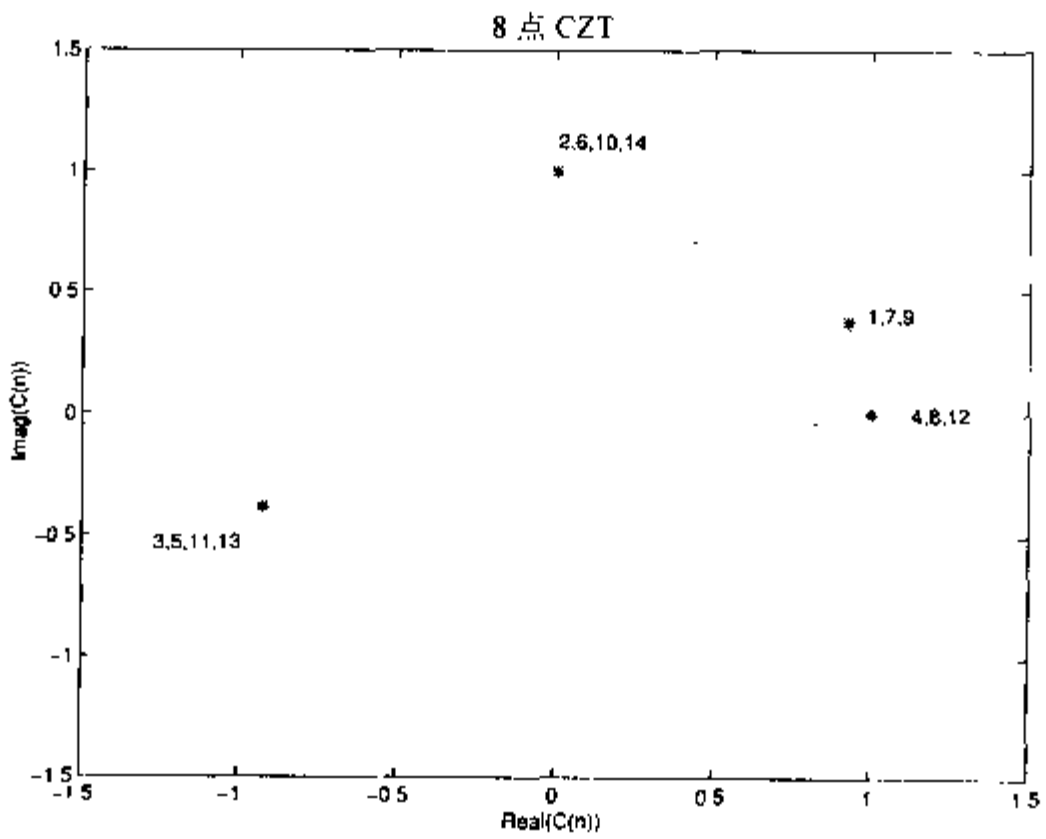


图 6-7　CZT 系数 $C(n) = e^{j2\pi\frac{n^2 \cdot 2 \bmod 8}{8}}$：$n = 1,2,\ldots,16$

相对于定点数 C_N 的系数而言，最大 DFT 的长度是多少应该是通常最感兴趣的。下面的表就给出了相应的数据。

DFT 长度	8	12	16	24	40	48	72	80	120	144	168	180	240	360	504
CN	4	6	7	8	12	14	16	21	24	28	32	36	42	48	64

正如前面所提到的，不同复数系数的数量与实现的计算量之间没有直接的联系，因为其中一些系数可能是无关紧要的(例如±1 或±j)，或者是对称的。特别是 2 的幂的变换就具有许多对称性，如图 6-8 所示。如果要为具体数量的不可或缺的实系数计算最大 DFT 长度，就会看到作为最大长度变换：

DFT 长度	10	16	20	32	40	48	50	80	96	160	192
sin/cos	2	3	5	6	8	9	10	11	14	20	25

长度 16 和 32 是分别只需要 3 个和 6 个实数乘法器的最大长度！

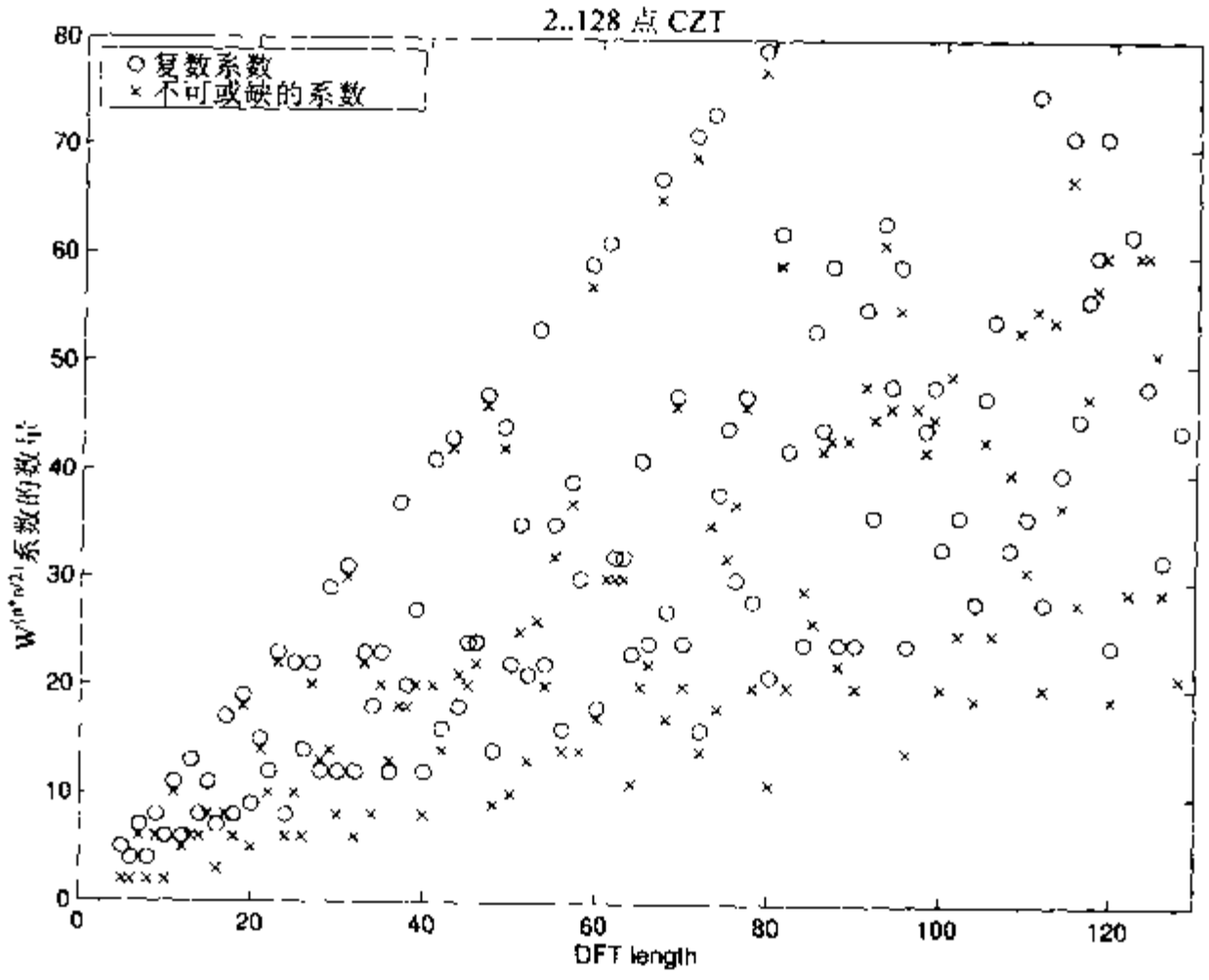


图 6-8　CZT 的复数系数和不可或缺的实数乘法的数量

一般情况下，2 的幂是受欢迎的 FFT 构造模块，下面的表就给出了在转置形式中，实现 CZT 滤波器时长度 $N=2^n$ 的工作量。

DFT 长度	8	16	32	64	128	256	512	1024
C_N	4	7	12	23	44	87	172	343
sin/cos	2	3	6	11	22	43	86	171
CSD	7	13	24	48	90	188	355	741
MAG	7	10	21	40	77	149	295	586
RAG	7	11	13	23	41	63	95	169

第 1 行是 DFT 的长度 N。第 2 行是复指数 C_N 的总数。复数系数 C_N 最坏的情况就是 C_N 有 2 个不可或缺的实数系数要实现。第 3 行给出了实际情况下不同的不可缺少的实系数的数量。将第 2 行与第 3 行加以对比，就会看到，对于 2 的幂长度，对称的和无关紧要的系数减少了不可缺少的系数的数量。最后 3 行给出了对于长度达到 256 时候的 CZT DFT，分别采用(第 2 章讨论过的)CSD、MAG、或 RAG 算法的 16 位(15 位无符号位和 1 个符号位)系数精度实现的工作量(也就是加法器的数量)。可以看到与 CSD 算法相比，RAG 算法可以从根本上将 DFT 长度的工作量减少 32 以上。

6.1.5 Rader 算法

用 Rader 算法[130, 131]计算 DFT：

$$X[k]=\sum_{n=0}^{N-1}x[n]W_N^{nk} \quad k,n\in Z_N;\ \text{ord}\,(W_N)=N \tag{6.9}$$

只能够定义质数长度 N。首先利用：

$$X[0]=\sum_{n=0}^{N-1}x[n] \tag{6.10}$$

计算 DC 组成部分。由于 $N=p$ 是质数，根据第 2 章的讨论知道，需要一个本原元素和一个生成器，就可以产生 Z_p 域内除 0 之外的所有元素，也就是 $g^k\in Z_p/\{0\}$。这里用 g^n 模 N 通过 g^k 模 N 代替 n，就得到下面的索引变换：

$$X[g^k \bmod N]-x[0]=\sum_{n=0}^{N-2}x[g^n \bmod N]W_N^{g^{n+k \bmod (N-1)}} \tag{6.11}$$

其中 $k\in\{1,2,3,\ldots,N-1\}$。可以看到(6.11)的右侧是一个循环卷积，也就是：

$$[x[g^0 \bmod N],x[g^1 \bmod N],\ldots,x[g^{N-2} \bmod N]]\circledcirc[W_N,W_N^g,\ldots,W_N^{g^{N-2} \bmod (N-1)}] \tag{6.12}$$

接下来是一个 $N=7$ 的 Rader 算法的例题。

例 6.5　$N=7$ 的 Rader 算法

对于 $N=7$，有 $g=3$ 是一个本原元素(请参阅【5】，表 B-7)，其索引变换如下：

$$[3^0,3^1,3^2,3^3,3^4,3^5] \bmod 7\equiv[1,3,2,6,4,5] \tag{6.13}$$

首先计算 DC 组成部分：

$$X[0]=\sum_{n=0}^{6}x[n]=x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]$$

接下来是 $X[k]-x[0]$的循环卷积：

$$[x[1],x[3],x[2],x[6],x[4],x[5]]\circledcirc[W_7,W_7^3,W_7^2,W_7^6,W_7^4,W_7^5]$$

或者以矩阵表示：

$$\begin{bmatrix} X[1] \\ X[3] \\ X[2] \\ X[6] \\ X[4] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 \\ W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 \\ W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 \\ W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 \\ W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 \\ W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 \end{bmatrix} \begin{bmatrix} x[1] \\ x[3] \\ x[2] \\ x[6] \\ x[4] \\ x[5] \end{bmatrix} + \begin{bmatrix} x[0] \\ x[0] \\ x[0] \\ x[0] \\ x[0] \\ x[0] \end{bmatrix} \tag{6.14}$$

图 6-9 给出了相应的采用 FIR 滤波器的图形化解释。

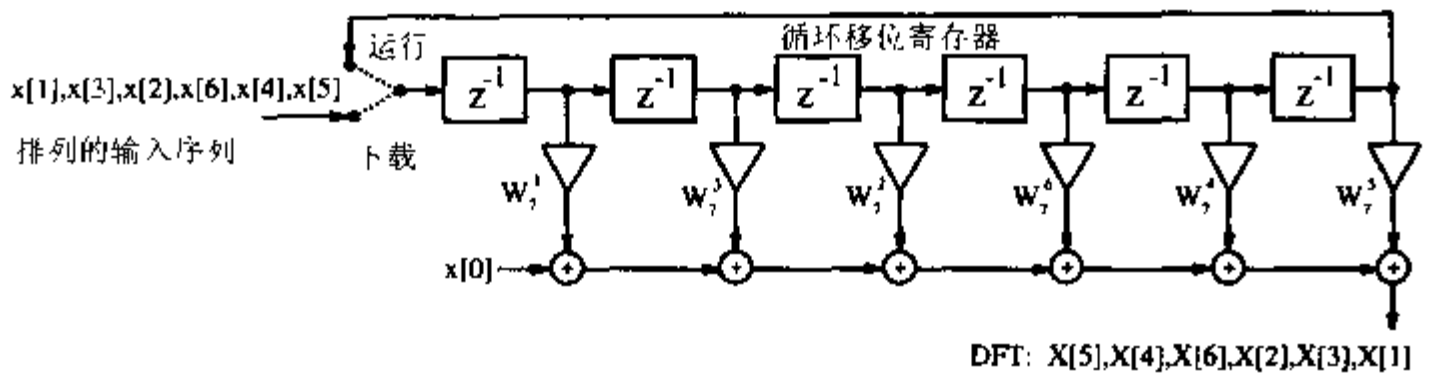


图 6-9　长度 p=7 的 Rader 质数因子 DFT 实现

现在可以用一个三角形信号 $x[n]=10\lambda[n]$(也就是步长为 10 的三角形)来检验 $p=7$ 的 Rader DFT 公式。直接解释(6.14)，就得到：

$$\begin{bmatrix} X[1] \\ X[3] \\ X[2] \\ X[6] \\ X[4] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 \\ W_7^3 & W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 \\ W_7^2 & W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 \\ W_7^6 & W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 \\ W_7^4 & W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 \\ W_7^5 & W_7^1 & W_7^3 & W_7^2 & W_7^6 & W_7^4 \end{bmatrix} \begin{bmatrix} 20 \\ 30 \\ 40 \\ 50 \\ 60 \\ 70 \end{bmatrix} + \begin{bmatrix} 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \end{bmatrix} = \begin{bmatrix} -35+j72 \\ -35+j8 \\ -35+j28 \\ -35-j72 \\ -35-j8 \\ -35-j28 \end{bmatrix}$$

$X[0]$的值就是时间级数的和，即 10+20+...+70=280。

此外，在 Rader 算法中，我们还可以使用复数对 $e^{\pm j2kn/N}$，$k \in [0, N/2]$的对称性来构造更为有效的 FIR 实现(练习 6.6)。实现 Rader 质数因子 DFT 与实现 FIR 滤波器是等价的，这一点我们已经在第 3 章讨论过了。为了实现快速 FIR 滤波器，有必要使用完全流水线 DA 或转置滤波器结构。下面就给出了一个 RAG FPGA 实现的示例。

例 6.6　Rader 算法的 FPGA 实现

长度为 7 的 Rader 算法的 RAG 实现过程如下。首先是对系数进行量化。假定输入值和系数都被表示成 8 位有符号数，量化后的系数是：

$k=$	0	1	2	3	4	5	6
$\mathrm{Re}\{256 \cdot W_7^k\}$	256	160	-57	-231	-231	-57	160
$\mathrm{Im}\{256 \cdot W_7^k\}$	0	-200	-250	-111	111	250	200

所有独立系数直接形式的实现(请查询表 2-3)都需要为常系数乘法器提供 24 个加法器。运

用转置结构，利用几个系数仅仅是符号不同这一事实，独立系数实现的工作量就可以降低到 11 个加法器。进一步优化(RAG，请参阅图 2-3)加法器的数量，就可以达到最小值 7(请参阅后面的 Factor: PROCESS 和 Coeffs: PROCESS)。对直接 FIR 结构有 3 倍以上的提高。接下来的 VHDL 代码[1]给出了运用转置 FIR 滤波器、长度为 7 的 Rader DFT 的一个可行实现。

```
PACKAGE B_bit_int IS      ------> User defined types
  SUBTYPE WORD8 IS INTEGER RANGE - 2**7 TO 2**7 - 1;
  SUBTYPE WORD11 IS INTEGER RANGE - 2**10 TO 2**10 - 1;
  SUBTYPE WORD19 IS INTEGER RANGE - 2**18 TO 2**18 - 1;
  TYPE ARRAY_WORD IS ARRAY (0 to 5) OF WORD19;
END B_bit_int;

LIBRARY work;
USE work.B_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY rader7 IS                              ------> Interface.
  PORT ( clk              : IN   STD_LOGIC;
         x_in             : IN   WORD8;
         y_real, y_imag   : OUT WORD11);
END rader7;

ARCHITECTURE flex OF rader7 IS

  SIGNAL  count     : integer RANGE 0 TO 15;
  TYPE    STATE_TYPE IS (Start, Load, Run);
  SIGNAL  state     : STATE_TYPE ;
  SIGNAL  accu      : WORD11;                   -- Signal for X[0]
  SIGNAL  real, imag : ARRAY_WORD;
                                          -- Tapped delay line array
  SIGNAL  x57, x111, x160, x200, x231, x250 : WORD19 ;
                          -- The (unsigned) filter coefficients
  SIGNAL  x5, x25, x110, x125, x256   : WORD19 ;
                                -- Auxiliary filter coefficients
  SIGNAL  x, x_0 : WORD8;   -- Signals for x[0]

BEGIN

  States: PROCESS      ------> State machine for RADER filter
```

注 1：这一例子相应的 Verilog 代码文件 rader7.v 可以在附录 A 中找到。

```
BEGIN
  WAIT UNTIL clk = '1';
  CASE state IS
    WHEN Start =>              -- Initialization step
      state <= Load;
      count <= 1;
      x_0 <= x_in;             -- Save x[0]
      accu <= 0 ;              -- Reset accumulator for X[0]
       y_real  <= 0;
       y_imag  <= 0;
    WHEN Load =>     -- Apply x[5],x[4],x[6],x[2],x[3],x[1]
      IF count = 8 THEN        -- Load phase done ?
        state <= Run;
      ELSE
        state <= Load;
        accu <= accu + x ;
      END IF;
      count <= count + 1;
    WHEN Run   =>   -- Apply again x[5],x[4],x[6],x[2],x[3]
      IF count = 15 THEN       -- Run phase done ?
        y_real  <= accu;       -- X[0]
        y_imag  <= 0;          -- Only re inputs i.e. Im(X[0])=0
        state <= Start;        -- Output of result
      ELSE                     -- and start again
        y_real  <= real(0) / 256 + x_0;
        y_imag  <= imag(0) / 256;
        state <= Run;
      END IF;
      count <= count + 1;
  END CASE;
END PROCESS States;

Structure: PROCESS             -- Structure of the two FIR
BEGIN                          -- filters in transposed form
  WAIT UNTIL clk = '1';
  x <= x_in;
  -- Real part of FIR filter in transposed form
  real(0) <= real(1) + x160    ;    -- W^1
  real(1) <= real(2) - x231    ;    -- W^3
  real(2) <= real(3) - x57     ;    -- W^2
  real(3) <= real(4) + x160    ;    -- W^6
  real(4) <= real(5) - x231    ;    -- W^4
```

```
    real(5) <=  - x57                ;      -- W^5

    -- Imaginary part of FIR filter in transposed form
    imag(0) <= imag(1) - x200    ;      -- W^1
    imag(1) <= imag(2) - x111    ;      -- W^3
    imag(2) <= imag(3) - x250    ;      -- W^2
    imag(3) <= imag(4) + x200    ;      -- W^6
    imag(4) <= imag(5) + x111    ;      -- W^4
    imag(5) <= x250;                    -- W^5
END PROCESS Structure;

Coeffs: PROCESS    -- Note that all signals
BEGIN              -- are globally defined
   WAIT UNTIL clk = '1';
-- Compute the filter coefficients and use FFs
   x160    <= x5 * 32;
   x200    <= x25 * 8;
   x250    <= x125 * 2;
   x57     <= x25 + x * 32;
   x111    <= x110 + x;
   x231    <= x256 - x25;
END PROCESS Coeffs;

Factors: PROCESS (x, x5, x25)        -- Note that all signals
BEGIN                                -- are globally defined
-- Compute the auxiliary factor for RAG without an FF
   x5      <= x * 4 + x;
   x25     <= x5 * 4 + x5;
   x110    <= x25 * 4 + x5 * 2;
   x125    <= x25 * 4 + x25;
   x256    <= x * 256;
END PROCESS Factors;

END flex;
```

本设计包括 4 个 PROCESSS 声明内的 4 个声明模块。第一个——“Stages：PROCESS”——是一个区分 3 个处理阶段：Start、Load 和 Run 的状态机。第二个——“Structure：PROCESS”——定义了两个 FIR 滤波器通路，分别是实和虚。第三项用 RAG 实现乘法器模块。第四个模块——“Factor：PROCESS”——实现 RAG 算法的未注册因子。可以看到，所有的系数都是由 6 个加法器和 1 个减法器实现的。本设计消耗了 494 个 LC，且以 29.94MHz 的 Registered Performance 运行。图 6-10 给出了 MaxPlussII 对三角波输入信号序列 $x[n]=\{10,20,30,40,50,60,70\}$ 的仿真结果。注意，输入和输出序列的起始点是 950ns，按交换的顺序作为无符号正数出现。最后，在 1.55μs 处 $X[0]$被发送到输出端，rader7 准备处理下一个输入帧。

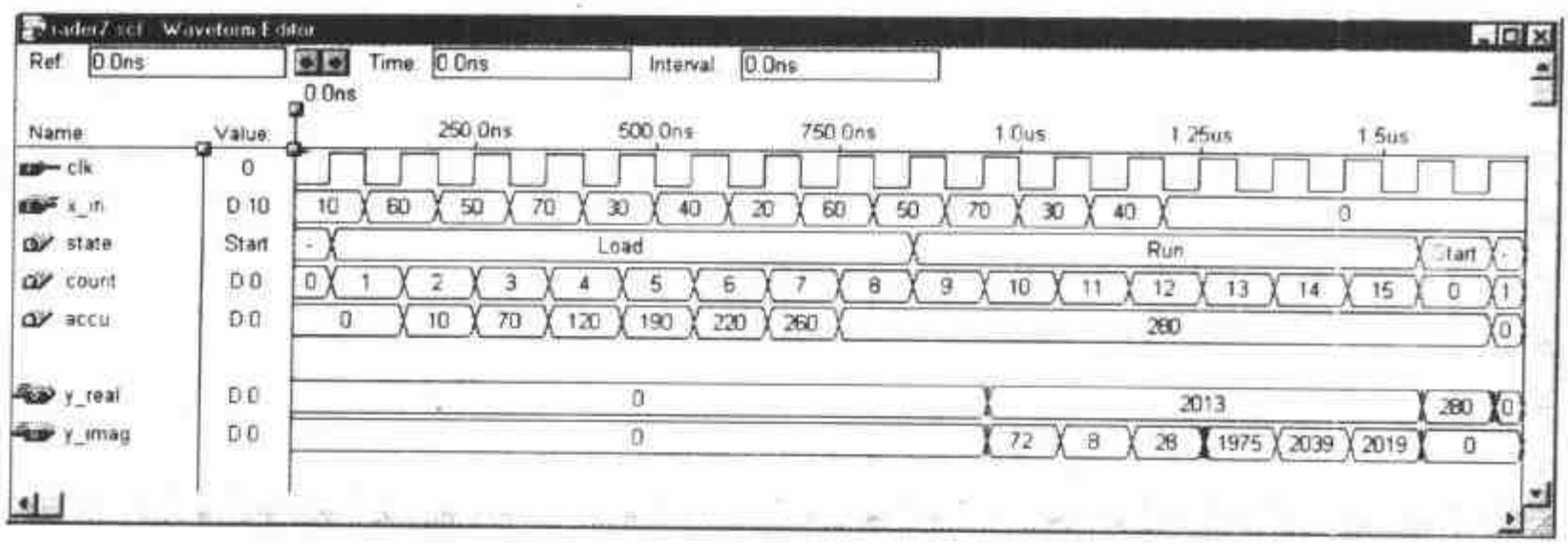


图 6-10　7 点 Rader 算法的 VHDL 仿真

由于 Rader 算法是受限于质数长度的，与 CZT 相比，在系数中就比较缺乏对称性。下面的表给出了质数长度为 $2^n \pm 1$ 时，转置形式的循环滤波器的实现工作量。

DFT 长度＃系数	7	17	31	61	127	257
sin/cos	6	16	30	60	124	253
CSD	28	76	126	265	553	1075
MAG	24	57	98	219	443	873
RAG	14	36	45	73	131	253

第 1 行给出了循环卷积长度 N，也就是复数系数的数量。将第 2 行与 $2N$ 个实 sin/cos 系数的最差情况相比较，就会看到，对称性和无关紧要的系数已经将不可或缺的系数降低了一半。最后 3 行分别给出了一个使用 CSD、MAG 或 RAG 算法的 16 位系数精度的实现工作量。注意 RAG 对较长滤波器的优势。从上面的表可以看到，CSD 类型的滤波器可以减少 $BN/4$ 的工作量，其中 B 是系数位宽(本表中是 16 位)，N 是滤波器长度。对于 RAG，工作量(也就是加法器的数量)仅是 N，也就是对长滤波器而言，比 CSD 提高了 $B/4$(B=16，提高了 16/4＝4)。对于长滤波器，RAG 只需要为额外的系数提供一个额外的加法器即可，因为已经合成的系数生成了一个“密集的”小系数栅格。

6.1.6　Winograd DFT 算法

我们要讨论的第一种精简必要乘法数量的算法就是 Winograd DFT 算法。Winograd DFT 算法是 Rader 算法(是将 DFT 转换成循环卷积)与我们在前面实现快速运行 FIR 滤波器时使用过的 Winograd[83] 短卷积算法(请参阅 5.2.2 节)的结合。

因而长度被限制在质数或质数的幂范围内。表 6-2 简要的给出了算法操作的必要数量。

表 6-2　带有实输入的 Winograd DFT 的效果表

组　长　度	实乘法总数量	非必要乘法总数量	实加法总数量
2	2	0	2
3	3	2	6
4	4	0	8
5	6	5	17
7	9	8	36
8	8	2	26
9	11	10	44
11	21	20	84
13	21	20	94
16	18	10	74
17	36	35	157
19	39	38	186

下面 $N=5$ 的示例详细地说明了构造 Winograd DFT 算法的步骤。

例 6.7　$N=5$ 的 Winograd DFT 算法

在由【5】给出的 Rader 算法的一个表达式中，用 $X[0]$代替 $x[0]$的形式如下：

$$X[0]=\sum_{n=0}^{4}x[n]=x[0]+x[1]+x[2]+x[3]+x[4]$$

$$X[k]-X[0]=[x[1],x[2],x[4],x[3]]\odot[W_5-1,W_5^2-1,W_5^4-1,W_5^3-1] \qquad k=1,2,3,4$$

如果用 Winograd 算法实现长度为 4 的循环卷积，只需要 5 次必不可少的乘法，我们就会得到下面的算法：

$$X[k]=\sum_{n=0}^{4}x[n]e^{-j2\pi kn/5} \qquad k=0,1,\cdots,4$$

$$\begin{bmatrix}X[0]\\X[1]\\X[2]\\X[3]\\X[4]\end{bmatrix}=\begin{bmatrix}1&0&0&0&0&0\\1&1&1&1&0&-1\\1&1&-1&1&1&0\\1&1&-1&-1&-1&0\\1&1&1&-1&0&1\end{bmatrix}$$

$$\bullet\,\mathrm{diag}\left(1,\frac{1}{2}(\cos(2\pi/5)+\cos(4\pi/5))-1,\right.$$

$$\left.\frac{1}{2}(\cos(2\pi/5)-\cos(4\pi/5)),\right.$$

$$\left.j\sin(2\pi/5),j(-\sin(2\pi/5)+\sin(4\pi/5)),j(\sin(2\pi/5)+\sin(4\pi/5))\right)$$

$$\bullet\begin{bmatrix}1&1&1&1&1\\0&1&1&1&1\\0&1&-1&-1&1\\0&1&-1&1&-1\\0&1&0&0&-1\\0&0&-1&1&0\end{bmatrix}\begin{bmatrix}x[0]\\x[1]\\x[2]\\x[3]\\x[4]\end{bmatrix}$$

对于实数或虚数输入序列 $x[n]$，总计算量分别只有 5 次或 10 次实数乘法。

用矩阵表示 Winograd DFT 算法是非常方便的：

$$\boldsymbol{W}_{N_I} = \boldsymbol{C}_I \cdot \boldsymbol{B}_I \cdot \boldsymbol{A}_I \tag{6.15}$$

其中 $\boldsymbol{A}_I$ 合并了输入加法，$\boldsymbol{B}_I$ 是傅立叶系数的对角矩阵，而 $\boldsymbol{C}_I$ 包括了输出加法。惟一的缺点就是不能够很容易地确定短卷积算法的精确步骤，这是因为计算的输入、输出加法所在的序列已经在这种矩阵表达式中消失了。

Rader 算法和短 Winograd 卷积的组合，也就是 Winograd DFT 算法，本算法将在后面与索引映射一道引入 Winograd FFT 算法。这种算法是目前已知的 FFT 算法中实数乘法次数最少的 FFT 算法。

6.2　快速傅立叶变换(Fast Fourier Transform, FFT)算法

正像这一章的概述中所提到的，我们使用的是 Burrus[109] 提出的术语，他将所有的 FFT 算法简单地根据不同的(多维)输入输出序列的索引映射进行分类。这是建立在长度为 N 的 DFT(6.2)：

$$X[k] = \sum_{n=0}^{N-1} x[n] \; W_N^{nk} \tag{6.16}$$

到多维 $N = \prod_i N_i$ 的表达式的变换基础之上的。一般情况下，只需要讨论两个因子的情形就足够了，因为更高的维数可以通过简单地反复迭代替换其中的一个因子就能够实现。为了简化表达式，我们在此只在二维索引映射变换内讨论 3 种 FFT 算法。

将(时域)索引 n 用：

$$n = An_1 + Bn_2 \bmod N \qquad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases} \tag{6.17}$$

进行交换，其中 $N = N_1 N_2$，且 $A, B \in Z$ 是以后必须定义的常数，利用这种索引变换，就可以根据下面的公式：

$$[x[0]x[1]x[2]\Lambda\; x[N-1]] = \begin{bmatrix} x[0,0] & x[0,1] & \Lambda & x[0,N_2-1] \\ x[1,0] & x[1,1] & \Lambda & x[1,N_2-1] \\ \mathrm{M} & \mathrm{M} & \mathrm{M} & \mathrm{M} \\ x[N_1-1,0] & x[N_1-1,1] & \Lambda & x[N_1-1,N_2-1] \end{bmatrix} \tag{6.18}$$

来构造数据的二维映射 $f : C^N \to C^{N_1 \times N_2}$。将另一个索引映射 k 应用到输出(频)域，就得到：

$$k = Ck_1 + Dk_2 \bmod N \qquad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases} \tag{6.19}$$

其中 $C, D \in Z$ 是以后必须定义的常数。由于 DFT 是双映射，所以我们必须选择 A,B,C 和 D，这样变换表达式才能仍然保持惟一，也就是惟一的双映射投影。Burrus[109] 已经确定了如何为具体的 N_1 和 N_2 选择 A,B,C 和 D 的一般情形，这样映射就是双映射了(参阅练习 6.8 和 6.9)。本章中给出的变换都是惟一的。

区别不同 FFT 算法的重要一点就是是否允许 N_1 和 N_2 具有公因数的问题，也就是 gcd(N_1, N_2)>1(gcd，greatest common divisor，最大公约数)，或者说 N_1 和 N_2 必须是互质的。通常，gcd(N_1, N_2)>1 的算法指的是公共因数算法(common factor algorithms, CFAs)，而 gcd(N_1, N_2)=1 就称为质数因数算法(prime factor algorithms, PFAs)。在接下来要讨论的 CFA 算法是 Cooley-Tukey FFT，而 Good-Thomas 和 Winograd FFT 则是 PFA 类型的。应该强调的是 Cooley-Tukey 算法可以真正地用两个因数 $N=N_1N_2$ 实现，彼此之间是互质的，并且对于 PFA，因子 N_1 和 N_2 必须是互质的，也就是说它们自身不一定是质数。例如：长度 $N=12$ 的变换因数分解成 $N_1=4$ 和 $N_2=3$，既可以用于 CFA FFT 也可以用作 PFA FFT！

6.2.1 Cooley-Tukey FFT 算法

Cooley-Tukey FFT 是所有 FFT 算法中最为通用的，因为 N 可以任意地进行因数分解。最流行的 Cooley-Tukey FFT 就是变换长度 N 是 r 基的幂的形式，也就是 $N=r^v$。这些算法通常称作 r 基算法。

Cooley 和 Tukey(更早是 Gauss)提出的索引变换也是最简单的索引映射。在(6.17)式中令 $A=N_2$ 和 $B=1$ 就得到下面的映射结果：

$$n = N_2 n_1 + n_2 \qquad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases} \tag{6.20}$$

从 n_1 和 n_2 的正确范围可以得出结论，(6.17)式给出的模简化不需要显式地计算。

对于(6.19)式的反映射，Cooley 和 Tukey 选择 $C=1$ 和 $D=N_1$，就得到下面的映射结果：

$$k = k_1 + N_1 k_2 \qquad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases} \tag{6.21}$$

在这种情况下模计算也可以省略。这时候如果根据(6.20)式和(6.21)式分别将 n 和 k 代入 W_N^{nk} 就会得到：

$$W_N^{nk} = W_N^{N_2 n_1 k_1 + N_1 N_2 n_1 k_2 + n_2 k_1 + N_1 n_2 k_2} \tag{6.22}$$

由于 W 是 $N=N_1N_2$ 阶的。就可以得到 $W_N^{N_1 N_2} = W_{N_2}$ 和 $W_N^{N_2} = W_{N_1}$，将(6.22)式简化成：

$$W_N^{nk} = W_N^{n_1 k_1} W_N^{n_2 k_1} W_{N_1}^{n_2 k_2} \tag{6.23}$$

这时候将(6.23)式代入(6.16)式的 DFT，就得到：

$$X[k_1, k_2] = \sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \underbrace{\left(W_N^{n_2 k_1} \underbrace{\sum_{n_1=0}^{N_1-1} x[n_1, n_2] W_{N_1}^{n_1 k_1}}_{N_1\text{点变换}} \right)}_{\bar{x}[n_2, k_1]} \tag{6.24}$$

$$= \underbrace{\sum_{n_2=0}^{N_2-1} W_{N_2}^{n_2 k_2} \bar{x}[n_2, k_1]}_{N_2\text{点变换}} \tag{6.25}$$

现在定义完整的 Cooley-Tukey 算法：

算法 6.8　Cooley-Tukey 算法

$N=N_1N_2$ 点 DFT 可以通过下列步骤进行：

(1) 根据(6.20)计算输入序列的索引变换。

(2) 计算长度 N_1 的 N_2 个 DFT。

(3) 在第一个变换级的输出上应用旋转因子 $W_N^{n_2k_1}$。

(4) 计算长度 N_2 的 N_1 个 DFT。

(5) 根据(6.21)计算输出序列的索引变换。

接下来用长度为 12 的变换来说明这些步骤。

例 6.9　N=12 的 Cooley-Tukey FFT

假设 $N_1=4$ 和 $N_2=3$。则有 $n=3n_1+n_2$ 和 $k=k_1+4k_2$，为索引映射计算下面的表：

n_2	n_1			
	0	1	2	3
0	$x[0]$	$x[3]$	$x[6]$	$x[9]$
1	$x[1]$	$x[4]$	$x[7]$	$x[10]$
2	$x[2]$	$x[5]$	$x[8]$	$x[11]$

k_2	k_1			
	0	1	2	3
0	$X[0]$	$X[1]$	$X[2]$	$X[3]$
1	$X[4]$	$X[5]$	$X[6]$	$X[7]$
2	$X[8]$	$X[9]$	$X[10]$	$X[11]$

在这个变换的帮助之下，可以构造如图 6-11 所示的信号流程图。可以看到，首先必须用 4 个点计算 3 个 DFT 中的每一个，然后乘以旋转因子，最后计算 4 个 DFT，其中每个的长度都是 3。

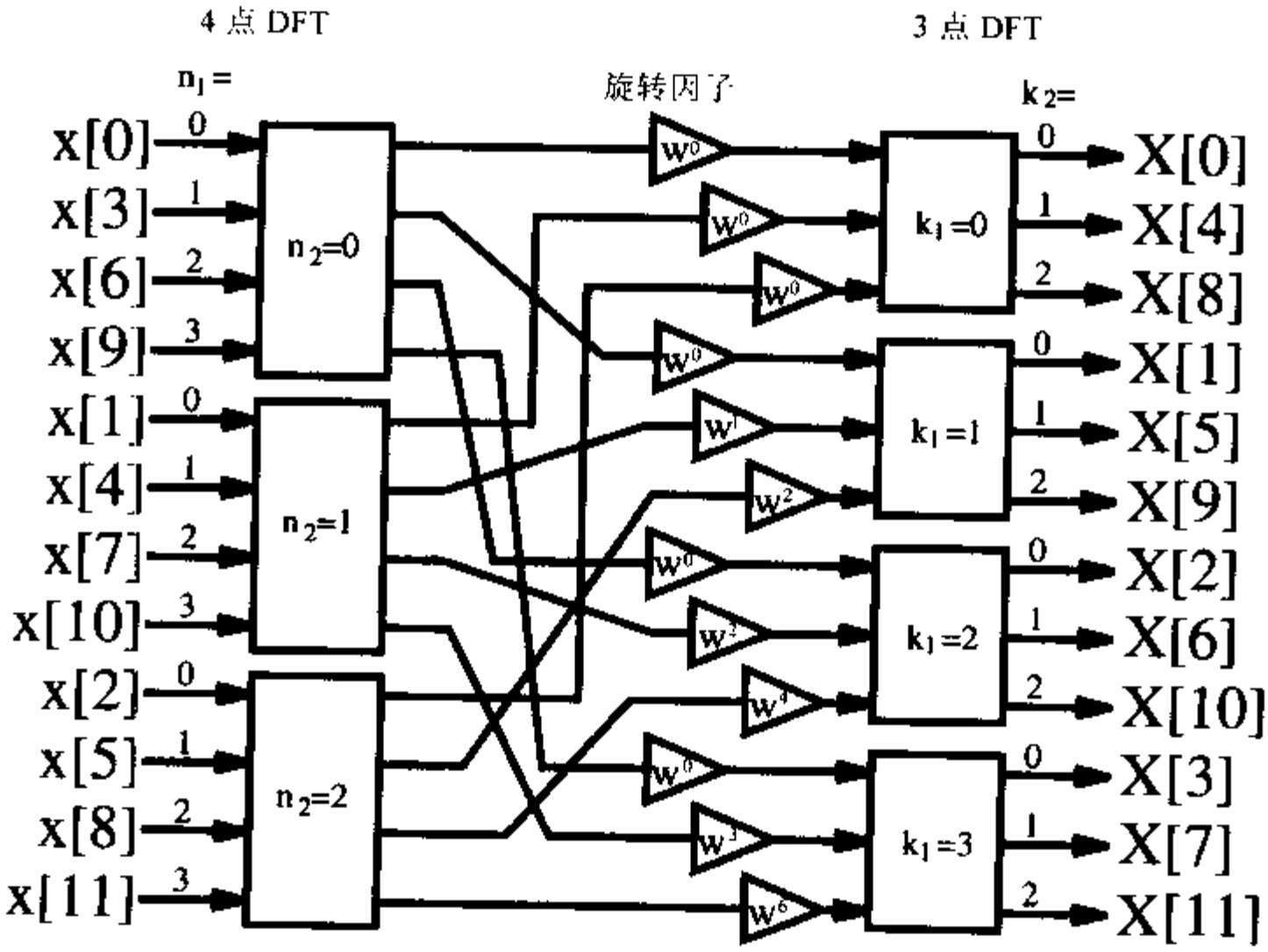


图 6-11　$N=12$ 的 Cooley-Tukey FFT

要直接计算 12 点的 DFT，总共需要 $12^2=144$ 次复数乘法和 $11^2=121$ 次复数加法。要计算同

样长度的 Cooley-Tukey FFT，旋转因子需要 12 次复数乘法，其中 8 次是无关紧要的乘法(也就是±1 或±j)。根据表 6-2，用 8 次实数加法就可以计算长度为 4 的 DFT，而且不需要乘法。要计算长度为 3 的 DFT，则需要 4 次乘法和 3 次加法。如果要用 3 次加法和 3 次乘法实现(固定系数的)复数乘法(请参阅算法 6.10)，12 点 Cooley-Tukey FFT 的总工作量是：

$$3\times 16+4\times 3+4\times 12=108 \text{ 次实数加法，和}$$

$$4\times 3+4\times 4=28 \text{ 次实数乘法}$$

而直接实现则需要 $2\times 11^2+12^2\times 3=674$ 次实数加法和 $12^2\times 3=432$ 次实数乘法。现在就应该非常清楚为什么将 Cooley-Tukey 算法叫作“快速傅立叶变换”(Fast Fourier Transform, FFT)的原因。

1. Radix-r Cooley-Tukey 算法

Cooley-Tukey 算法区别于其他 FFT 算法的一个重要事实就是 N 的因子可以任意选取。这样也就可以使用 $N=r^S$ 的 Radix-r 算法了。最流行的算法都是以 r=2 或 r=4 为基的，因为根据表 6-2，最简单的 DFT 不需要任何乘法就可以实现。例如：在 S 级且 r=2 的情形下，下列索引映射的结果是：

$$n=2^{S-1}n_1+\ldots+2n_{S-1}+n_S \tag{6.26}$$

$$k=k_1+2k_2+\ldots+2^{S-1}k_S \tag{6.27}$$

S>2 时的一个一般惯例是，在信号流程图中 2 点 DFT 是以蝶形图的形式绘出的，图 6-12 给出了 8 点变换的图示。信号流程图已经简化成用所有指向一个节点的箭头都代表加法的形式了，而常系数乘法则是在箭头上加一个因子表示。Radix-r 算法具有 $\log_r(N)$级，并且每组都有相同类型的旋转因子。

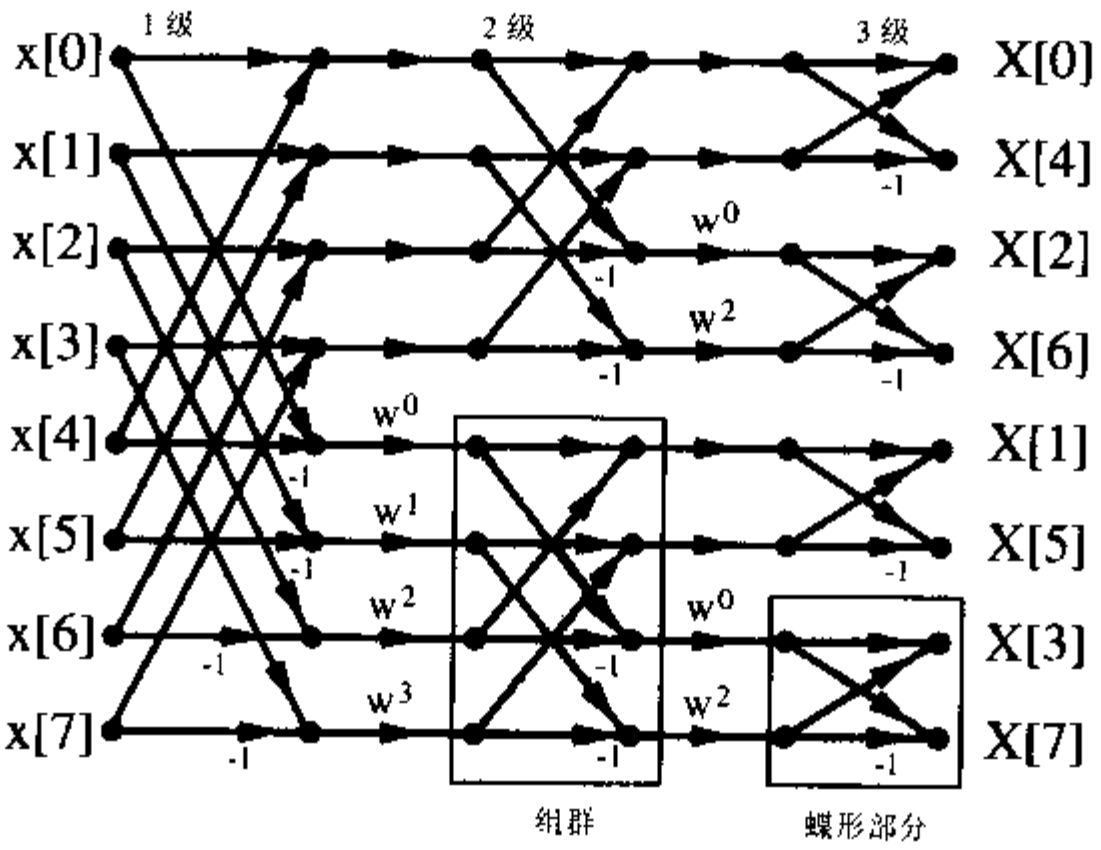


图 6-12　radix-2 的长度为 8 的频率抽取算法

从图 6-12 的信号流程图可以看出，计算可以“就地”完成，也就是蝶形所使用的存储位置可以被重写，因为数据在下一步的计算中已不再需要了。Radix-2 变换的旋转因子乘法总数是：

$$\log_2(N)N/2 \tag{6.28}$$

因为每两个箭头仅有一个旋转因子。

由于图 6-12 中的算法在频域中开始将最初的 DFT 分成更短的 DFT，所以这种算法就叫作频率抽取(decimation-in-frequency, DIF)算法。典型的输入值是按顺序出现的，而频率值的索引是按位逆序的。表 6-3 给出了 DIF Radix-2 算法的特征值。

表 6-3　频率抽取的 Radix-2 FFT

	第 1 级	第 2 级	第 3 级	…	第 $\log_2(N)$级
组数	1	2	4	…	$N/2$
每组的蝶形数量	$N/2$	$N/4$	$N/8$	…	1
增量指数旋转因子	1	2	4	…	$N/2$

我们还可以用时间抽取(decimation in time, DIT)构造一种算法。在该情况下，首先将输入序列分开，就会发现所有频率值都是按顺序出现的(练习 6.11)。

图 6-13 给出了索引 41 的 radix-2 和 radix-4 算法的必要索引变换。radix-2 算法需要位顺序的反转，也就是位逆序。而 radix-4 需要首先构造一个 2 位的“数字”然后再反转这些数字，这种操作就称为数字逆序。

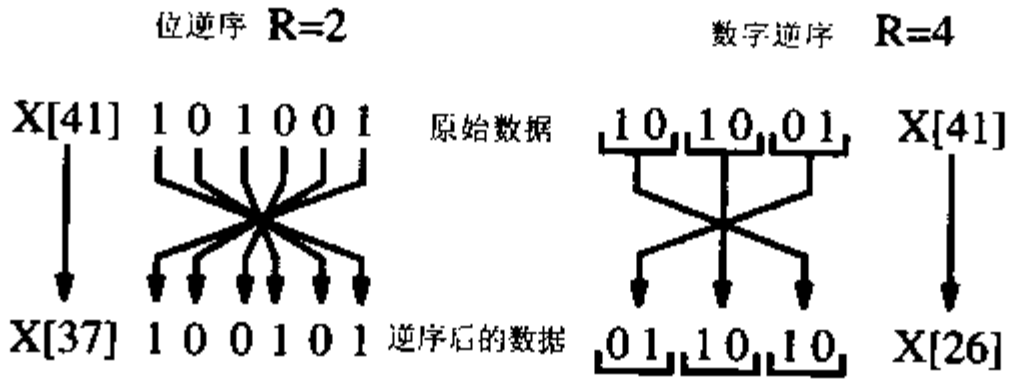


图 6-13　位逆序和数字逆序

2. Radix-2 Cooley-Tukey 算法的实现

radix-2 FFT 可以用蝶形处理器有效地实现，这种处理器除了蝶形本身外，还包括额外的旋转因子复数乘法器。

radix-2 蝶形处理器由一个复数加法器、一个复数减法器和一个旋转因子的复数乘法器组成。旋转因子的复数乘法通常由 4 次实数乘法和 2 次加/减法运算实现。但是只用 3 次实数乘法和 3 次加/减法运算构造复数乘法器也是可能的，因为一个操作数是可以预先计算的。算法如下：

算法 6.10　高效复数乘法器

复数旋转因子乘法 $R+jI=(X+jY)(C+jS)$是可以简化的，因为 C 和 S 可以预先计算，并储存在一个表中。而且还可以储存下面的 3 个系数：

$$C、C+S 和 C-S \tag{6.29}$$

有了这 3 个预先计算的因子，我们首先可以计算：

$$E=X-Y 和 Z=C*E=C*(X-Y) \tag{6.30}$$

然后用：

$$R=(C-S)*Y+Z \tag{6.31}$$

$$I=(C+S)*X-Z \tag{6.32}$$

计算最后的乘积。

检验：

$R=(C-S)Y+C(X-Y)$

$=CY-SY+CX-CY=CX-SY$　　√

$I=(C+S)X-C(X-Y)$

$=CX+SX-CX+CY=CY+SX$　　√

这种算法使用了 3 次乘法、1 次加法和 2 次减法，其代价是额外的第三个表。

下面的示例说明了这种旋转因子复数乘法器的实现过程。

例 6.11　旋转因子乘法器

我们首先给旋转因子乘法器选择一些具体的设计参数。假设有 8 位二进制输入数据，系数就应该有 8 位，再加上符号，并且乘以 $e^{j\pi/9}=e^{j20^\circ}$。量化成 8 位，旋转因子就变成了 $C+jS=256\times e^{j\pi/9}=121+j\,39$。如果输入值是 $70+j50$，则所期望的结果是：

$$(70+j50)e^{j\pi/9}=(70+j50)(121+j39)/256$$
$$=(6520+j8780)/256=25+j34$$

如果用算法 6.10 计算复数乘法，3 个因子就变成了：

$C=121$，$C+S=160$ 和 $C-S=82$

从上面可以看到，一般情况下 $C+S$ 和 $C-S$ 的表必须比 C 和 S 的表多一位精度。

下面的 VHDL 代码[2]实现了旋转因子乘法器。

```
LIBRARY lpm;
USE lpm lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

注 2：这一例子相应的 Verilog 代码文件 ccmul.v 可以在附录 A 中找到。

```
ENTITY ccmul IS
    GENERIC ( W2   : INTEGER := 17;      -- Multiplier bit width
              W1   : INTEGER := 9;       -- Bit width c+s sum
              W    : INTEGER := 8);      -- Input bit width
  PORT (clk   : STD_LOGIC;   -- Clock for the output register
        x_in, y_in, c_in                          -- Inputs
                  : IN   STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
        cps_in, cms_in                            -- Inputs
                  : IN   STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
        r_out, i_out                              -- Results
                  : OUT STD_LOGIC_VECTOR(W - 1 DOWNTO 0));
END ccmul;

ARCHITECTURE flex OF ccmul IS

  SIGNAL x, y, c : STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
                                    -- Inputs and outputs
  SIGNAL r, i, cmsy, cpsx, xmyc                -- Products
                        : STD_LOGIC_VECTOR(W2 - 1 DOWNTO 0);
  SIGNAL xmy, cps, cms, sxtx, sxty             -- x - y etc.
                        : STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);

BEGIN
    x     <= x_in;        -- x
    y     <= y_in;        -- j * y
    c     <= c_in;        -- cos
    cps   <= cps_in;      -- cos + sin
    cms   <= cms_in;      -- cos - sin

  PROCESS
  BEGIN
    WAIT UNTIL clk='1';
    r_out <= r(W2 - 3 DOWNTO W - 1);     -- Scaling and FF
    i_out <= i(W2 - 3 DOWNTO W - 1);     -- for output
  END PROCESS;
---------- ccmul with 3 mul. and 3 add/sub  ----------
  sxtx  <= x(x'high) & x;          -- Possible growth for
  sxty  <= y(y'high) & y;          -- sub_1 -> sign extension

  sub_1: lpm_add_sub                          -- Sub:  x - y;
    GENERIC MAP ( LPM_WIDTH => W1, LPM_DIRECTION => "SUB",
                  LPM_REPRESENTATION => "SIGNED")
```

```
    PORT MAP (dataa => sxtx, datab => sxty, result => xmy);

mul_1: lpm_mult                    -- Multiply   (x - y)*c = xmyc
  GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP ( dataa => xmy, datab => c, result => xmyc);

mul_2: lpm_mult                    -- Multiply (c - s)*y = cmsy
  GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP ( dataa => cms, datab => y, result => cmsy);

mul_3: lpm_mult                    -- Multiply (c+s)*x = cpsx
  GENERIC MAP ( LPM_WIDTHA => W1, LPM_WIDTHB => W,
                LPM_WIDTHP => W2, LPM_WIDTHS => W2,
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP ( dataa => cps, datab => x, result => cpsx);

sub_2: lpm_add_sub          -- Sub: i <= (c - s)*x - (x - y)*c;
  GENERIC MAP ( LPM_WIDTH => W2, LPM_DIRECTION => "SUB",
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP ( dataa => cpsx, datab => xmyc, result => i);

add_1: lpm_add_sub          -- Add: r <= (x - y)*c + (c+s)*y;
  GENERIC MAP ( LPM_WIDTH => W2, LPM_DIRECTION => "ADD",
                LPM_REPRESENTATION => "SIGNED")
  PORT MAP ( dataa => cmsy, datab => xmyc, result => r);

END flex;
```

旋转因子乘法器是用 3 个 lpm_mult 组件实例和 3 个 lpm_add_sub 模块来实现的。输出经过刻度，也具有与输入相同的数据格式。这是合理的，因为带有复数指数 $e^{j\phi}$ 的乘法不应该改变复数输入的幅值。(对于就地 FFT 而言)为了保证较短的延迟，复数乘法器只有输出寄存器，没有内部流水线寄存器。这个惟一的输出寄存器就有可能决定设计的 Registered Performance，但是从图 6-14 的仿真结果来看，可以忽略不计。这一设计使用了 493 个 LC，如果 lpm_mult 组件可以是流水线结构的话，运行速度还可以更快。

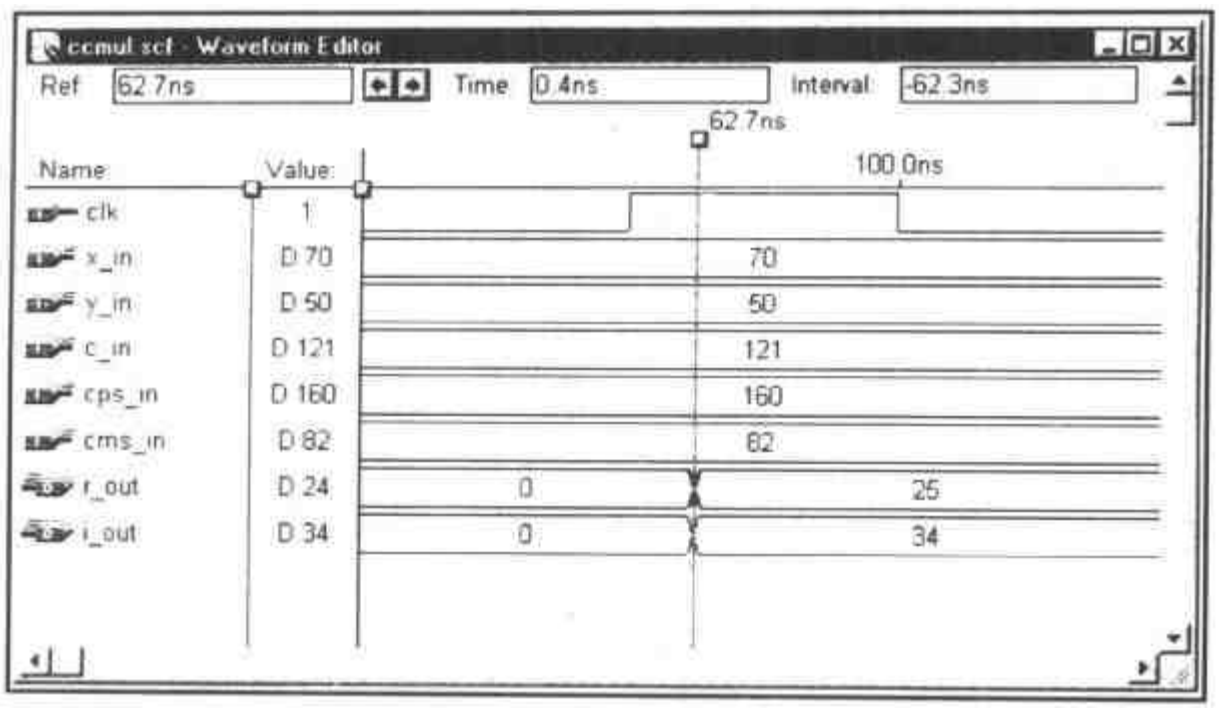


图 6-14　旋转因子乘法器的 VHDL 仿真

就地实现(也就是只有一个数据存储器)现在也是可行的，因为蝶形处理器可以被设计成没有流水线级的。如果引入额外的流水线级(一个给蝶形，3 个给乘法器)，设计规模就会增大一些(参阅练习 6.24)，不过可以提高速度。这种流水线设计的价格就是整个 FFT 的额外数据存储器的成本，这是因为数据读和写存储器现在必须被分开，也就是说非就地计算也可以实现。

用上面介绍的旋转因子乘法器，就可以为 Radix-2 Cooley-Tukey FFT 设计蝶形处理器了。

例 6.12　蝶形处理器

为了防止运算中的溢出，蝶形处理器需要计算两个(刻度的)蝶形方程：

$$D_{re}+j\,D_{im} = ((A_{re}+j\,A_{im})+(B_{re}+j\,B_{im}))/2$$
$$E_{re}+j\,E_{im} = ((A_{re}+j\,A_{im}) - (B_{re}+j\,B_{im}))/2$$

临时结果 $E_{re}+j\,E_{im}$ 必须乘以旋转因子。

整个蝶形处理器的 VHDL 代码[3] 如下：

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

PACKAGE mul_package IS       -- User defined components
  COMPONENT ccmul
    GENERIC ( W2   : INTEGER := 17;      -- Multiplier bit width
              W1   : INTEGER := 9;       -- Bit width c+s sum
```

注 3：这一例子相应的 Verilog 代码文件 bfproc.v 可以在附录 A 中找到。

```
                W     : INTEGER := 8);      -- Input bit width
    PORT
    (clk     : IN STD_LOGIC; -- Clock for the output register
     x_in, y_in, c_in: IN   STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
                                                    -- Inputs
     cps_in, cms_in   : IN   STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
                                                    -- Inputs
     r_out, i_out      : OUT STD_LOGIC_VECTOR(W - 1 DOWNTO 0));
                                                    -- Results
  END COMPONENT;
END mul_package;

LIBRARY work;
USE work.mul_package.ALL;


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_arith.ALL;

USE ieee.std_logic_unsigned.ALL;


ENTITY bfproc IS

   GENERIC ( W2   : INTEGER := 17;      -- Multiplier bit width

             W1   : INTEGER := 9;       -- Bit width c+s sum

             W    : INTEGER := 8);      -- Input bit width

  PORT

  (clk                : STD_LOGIC;
```

```
    Are_in, Aim_in, c_in,                    -- 8 bit inputs
    Bre_in, Bim_in     : IN   STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
    cps_in, cms_in     : IN   STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
                                             -- 9 bit coefficients
    Dre_out, Dim_out,                        -- 8 bit results
    Ere_out, Eim_out   : OUT STD_LOGIC_VECTOR(W - 1 DOWNTO 0));
END bfproc;

ARCHITECTURE flex OF bfproc IS

  SIGNAL   dif_re, dif_im                    -- Bf out
                          : STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
  SIGNAL   Are, Aim, Bre, Bim : integer RANGE  - 128 TO 127;
                                             -- Inputs as integers
  SIGNAL   c              : STD_LOGIC_VECTOR(W - 1 DOWNTO 0);
                                                    -- Input
  SIGNAL   cps, cms       : STD_LOGIC_VECTOR(W1 - 1 DOWNTO 0);
                                                    -- Coeff in
  SIGNAL   Cre, Cim       : STD_LOGIC_VECTOR(W-1 DOWNTO 0);
                                                    -- Results

BEGIN

  PROCESS    -- Compute the additions of the butterfly using
  BEGIN      -- integers and store inputs in flip-flops
    WAIT UNTIL clk = '1';
    Are   <= CONV_INTEGER(Are_in);
    Aim   <= CONV_INTEGER(Aim_in);
    Bre   <= CONV_INTEGER(Bre_in);
    Bim   <= CONV_INTEGER(Bim_in);
    c     <= c_in;              -- Load from memory cos
    cps   <= cps_in;            -- Load from memory cos+sin
    cms   <= cms_in;            -- Load from memory cos - sin
```

```
    Dre_out <= CONV_STD_LOGIC_VECTOR( (Are + Bre )/2, W);
    Dim_out <= CONV_STD_LOGIC_VECTOR( (Aim + Bim )/2, W);
  END PROCESS;

  -- No FF because butterfly difference "diff" is not an
  PROCESS (Are, Bre, Aim, Bim)                    -- output port
  BEGIN
    dif_re <= CONV_STD_LOGIC_VECTOR(Are/2 - Bre/2, 8);
    dif_im <= CONV_STD_LOGIC_VECTOR(Aim/2 - Bim/2, 8);
  END PROCESS;

---- Instantiate the complex twiddle factor multiplier ----
  ccmul_1: ccmul                                  -- Multiply (x+jy)(c+js)
    GENERIC MAP ( W2 => W2, W1 => W1, W => W)
    PORT MAP    ( clk => clk, x_in => dif_re, y_in => dif_im,
                  c_in => c, cps_in => cps, cms_in => cms,
                  r_out => Ere_out, i_out => Eim_out);

END flex;
```

蝶形处理器是由一个加法器、一个减法器和一个实例化为组件的旋转因子乘法器实现的。为了实现单输入/输出的已注册设计，还为输入 A、B、3 个表值和输出端 D 提供了触发器。这一设计采用了 533 个 LC，并以 18.38MHz 的 Registered Performance 运行。图 6-15 给出了零流水线设计的仿真，输入 A=100+j110，B=-40+j10 和 $W=e^{j\pi/9}$。

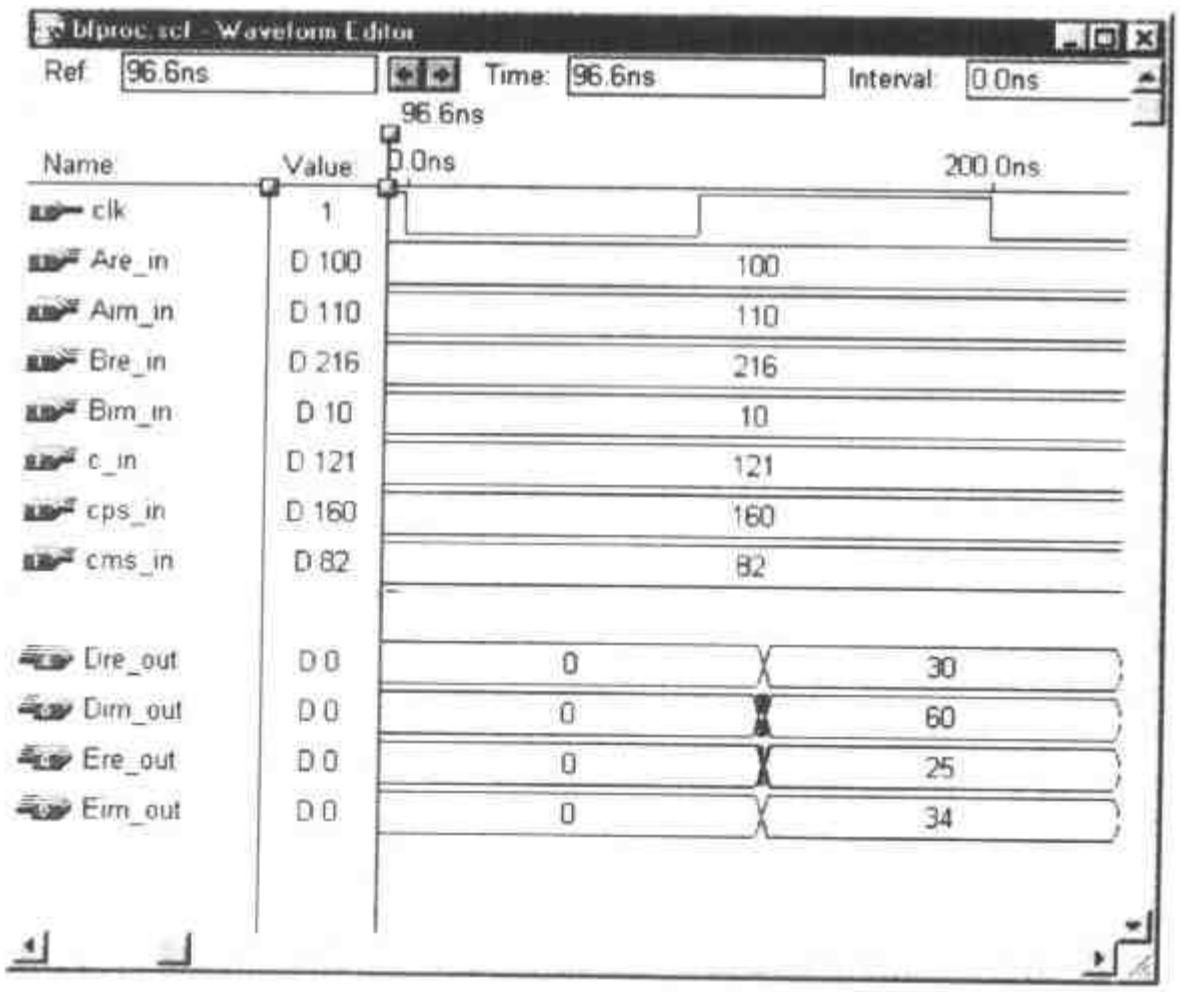


图 6-15　radix-2 蝶形处理器的 VHDL 仿真

6.2.2　Good-Thomas FFT 算法

Good[132]和 Thomas[133]提出的索引变换将一个长度 $N=N_1N_2$ 的 DFT 变换成“实际的”二维 DFT，也就是说没有 Cooley-Tukey FFT 中的旋转因子。无旋转因子流程的代价就是因子之间必须是互质的(也就是 $\gcd(N_k, N_l)=1$，$k\neq l$)，只要索引映射计算是在线进行的，而且没有预先计算好的表可供使用，那么索引映射就会变得更加复杂。

如果分别依据(6.17)式和(6.19)式，试图消除通过 n 和 k 的索引映射引入的旋转因子，就会有

$$\begin{aligned} W_N^{nk} &= W_N^{(An_1+Bn_2)(Ck_1+Dk_2)} \\ &= W_N^{ACn_1k_1+ADn_1k_2+BCk_1n_2+BDn_2k_2} \\ &= W_N^{N_2n_1k_1}W_N^{N_1k_2n_2} = W_{N_1}^{n_1k_1}W_{N_2}^{k_2n_2} \end{aligned} \tag{6.33}$$

前提是必须同时满足下列所有条件：

$$\langle AD\rangle_N = \langle BC\rangle_N = 0 \tag{6.34}$$

$$\langle AC\rangle_N = N_2 \tag{6.35}$$

$$\langle BD\rangle_N = N_1 \tag{6.36}$$

Good[132]和 Thomas[133]提出的映射要满足下面这一条件：

$$A=N_2 \quad B=N_1 \quad C=N_2\langle N_2^{-1}\rangle_{N_1} \quad D=N_1\langle N_1^{-1}\rangle_{N_2} \tag{6.37}$$

检验：因为 AD 和 BC 之中均包括因子 $N_1N_2=N$，所以也就验证了(6.34)式。有了 $\gcd(N_1, N_2)=1$ 以及欧拉定理，就可以写出逆运算 $N_2^{-1} \bmod N_1 = N_2^{\phi(N_1)-1} \bmod N_1$，其中 ϕ 是欧拉 ϕ 函数。条件(6.35)现在就可以改写成：

$$\langle AC\rangle_N = \langle N_2N_2\langle N_2^{\phi(N_1)-1}\rangle_{N_1}\rangle_N \tag{6.38}$$

这样就解决了内部模的简化。$v\in Z$ 且 $vN_1N_2 \bmod N=0$ 的最终形式是：

$$\langle AC\rangle_N = \langle N_2N_2(N_2^{\phi(N_1)-1}+vN_1)\rangle_N = N_2 \tag{6.39}$$

同样的理论也适用于条件(6.36)式，且如果采用 Good-Thomas 映射，从(6.34)式到(6.36)式的 3 个条件均可以满足。

最后，我们就可以得到下面的定理。

定理 6.13　Good-Thomas 索引映射

Good-Thomas 提出的 n 的索引映射是：

$$n = N_2n_1 + N_1n_2 \bmod N \quad \begin{cases} 0 \le n_1 \le N_1-1 \\ 0 \le n_2 \le N_2-1 \end{cases} \tag{6.40}$$

而 k 的索引映射结果是：

$$k = N_2\langle N_2^{-1}\rangle_{N_1}k_1 + N_1\langle N_1^{-1}\rangle_{N_2}k_2 \bmod N \quad \begin{cases} 0 \le k_1 \le N_1-1 \\ 0 \le k_2 \le N_2-1 \end{cases} \tag{6.41}$$

(6.41)式的变换与 2.2.12 中的中国余数定理是一致的。所以 k_1 和 k_2 可以简单地通过模简化来计算，也就是 $k_i=k \bmod N_i$。

如果在 DFT 矩阵方程(6.16)式中替换 Good-Thomas 索引映射，就有：

$$X[k_1,k_2]=\sum_{n_2=0}^{N_2-1}W_{N_2}^{n_2k_2}\underbrace{\left(\underbrace{\sum_{n_1=0}^{N_1-1}x[n_1,n_2]W_{N_1}^{n_1k_1}}_{N_1\text{点变换}}\right)}_{\bar{x}[n_2,k_1]} \tag{6.42}$$

$$=\underbrace{\sum_{n_2=0}^{N_2-1}W_{N_2}^{n_2k_2}\bar{x}[n_2,k_1]}_{N_2\text{点变换}}, \tag{6.43}$$

也就是像前面提到的，这是一种“实际的”二维 DFT 变换，没有 Colley 和 Tukey 提出的映射所引入的旋转因子。

下面就是 Good-Thomas 算法，虽然与 Colley-Tukey 算法 6.8 相似，但是索引映射不同，而且没有旋转因子。

算法 6.14　Good-Thomas FFT 算法

$N=N_1N_2$ 点的 DFT 可以遵循下面的步骤进行计算：

(1) 根据(6.40)进行输入序列的索引变换。

(2) 计算长度 N_1 的 N_2 次 DFT 的计算。

(3) 计算长度 N_2 的 N_1 次 DFT 的计算。

(4) 根据(6.41)进行输出序列的索引变换。

下面用 N=12 的变换的示例来说明这些步骤。

例 6.15　N=12 的 Godd-Thomas FFT 算法

假设 N_1=4，N_2=3，接下来根据 $n=3n_1+4n_2 \bmod 12$ 和 $k=9k_1+4k_2 \bmod 12$ 计算输入索引到输出索引结果的映射，并且也可以计算下面的索引映射表：

n_2	n_1			
	0	1	2	3
0	$x[0]$	$x[3]$	$x[6]$	$x[9]$
1	$x[4]$	$x[7]$	$x[10]$	$x[1]$
2	$x[8]$	$x[11]$	$x[2]$	$x[5]$

k_2	k_1			
	0	1	2	3
0	$X[0]$	$X[9]$	$X[6]$	$X[3]$
1	$X[4]$	$X[1]$	$X[10]$	$X[7]$
2	$X[8]$	$X[5]$	$X[2]$	$X[11]$

利用这些索引变换，我们就可以构造图 6-16 所示的信号流程图了。其中第一级有 3 个 DFT，每个 DFT 又有 4 个点，第二级有 4 个 DFT，每个 DFT 的长度为 3。级间不需要旋转因子乘法。

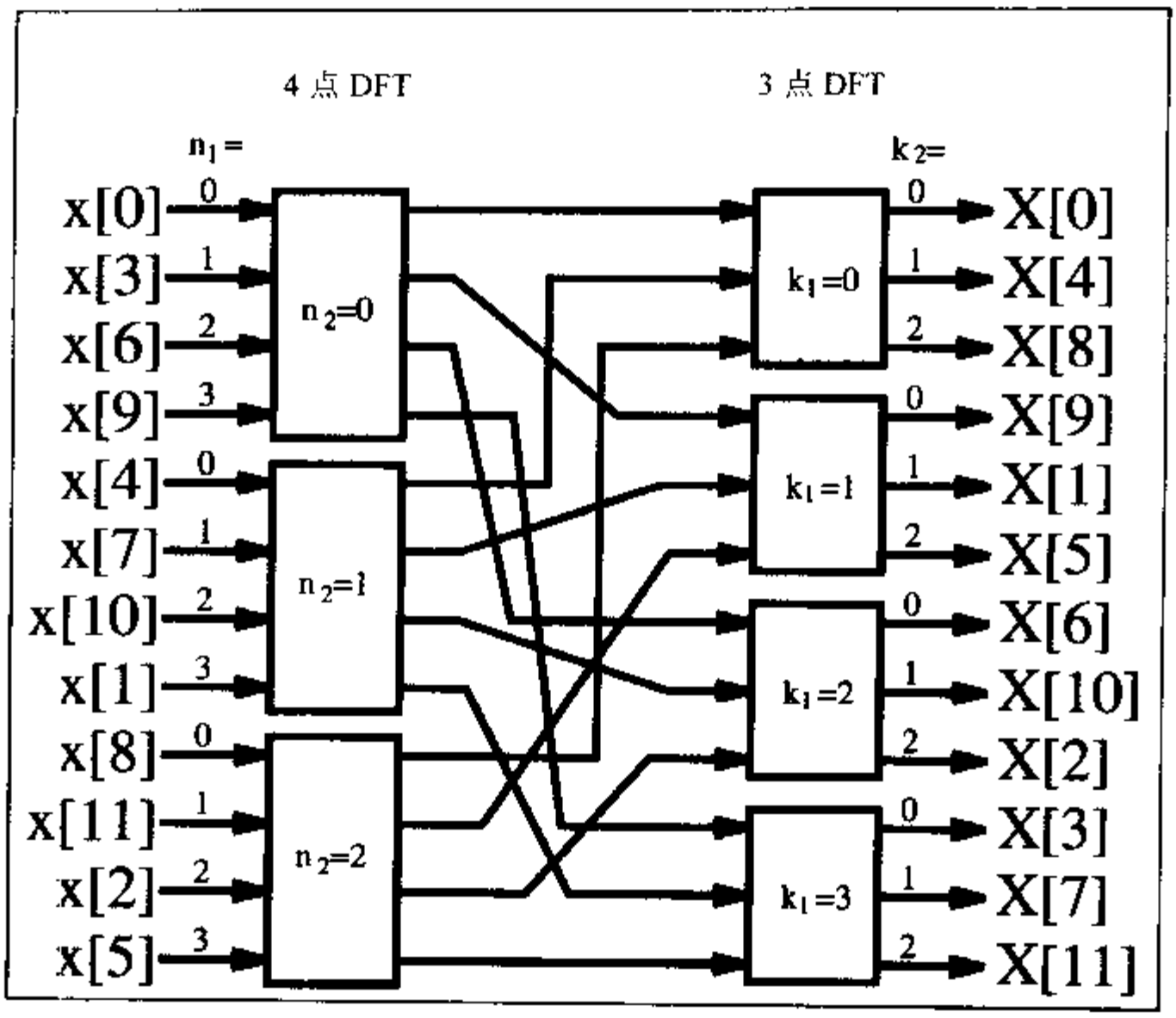


图 6-16　N=12 的 PFA FFT

6.2.3　Winograd FFT 算法

Winograd FFT 算法[83]是建立在对 $N_1 \times N_2$ 维逆 DFT 矩阵(没有前因子 N^{-1}) 的观察基础上，gcd(N_1，N_2)=1 也就是：

$$x[n] = \sum_{k=0}^{\lambda-1} X[k] W^{-nk} \tag{6.44}$$

$$\boldsymbol{x} = \boldsymbol{W}_N^* \boldsymbol{X} \tag{6.45}$$

这两个公式可以用两个分别是 N_1 和 N_2 维的二次 IDFT 矩阵的 Kronecker 乘积[4]重写。如同 Good-Thomas 算法的映射一样，我们必须将 $X[k]$和 $x[n]$的索引写成二维帧格式，然后逐行读出索引。下面给出一个 N=12 的示例来说明这些步骤。

例 6.16　使用 Kronecker 乘积且 N=12 的 IDFT

令 N_1=4 和 N_2=3，然后根据 Good-Thomas 索引映射进行输出索引变换 $k = 9k_1 + 4k_2 \bmod 12$

注 4：Kronecker 乘积定义是

$$\boldsymbol{A} \otimes \boldsymbol{B} = [a[i,j]]\boldsymbol{B} = \begin{bmatrix} a[0,0]\boldsymbol{B} & \cdots & a[0,L-1]\boldsymbol{B} \\ \vdots & & \vdots \\ a[k-1,0]\boldsymbol{B} & \cdots & a[K-1,L-1]\boldsymbol{B} \end{bmatrix}$$

其中 $\boldsymbol{A}$ 是一个 $K \times L$ 阶矩阵。

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ X[3] \\ X[4] \\ X[5] \\ X[6] \\ X[7] \\ X[8] \\ X[9] \\ X[10] \\ X[11] \end{bmatrix} \rightarrow \begin{array}{c|cccc} k_2 & k_1 & & & \\ & 0 & 1 & 2 & 3 \\ \hline 0 & X[0] & X[9] & X[6] & X[3] \\ 1 & X[4] & X[1] & X[10] & X[7] \\ 2 & X[8] & X[5] & X[2] & X[11] \end{array} \rightarrow \begin{bmatrix} X[0] \\ X[9] \\ X[6] \\ X[3] \\ X[4] \\ X[1] \\ X[10] \\ X[7] \\ X[8] \\ X[5] \\ X[2] \\ X[11] \end{bmatrix}$$

接下来构造长度为 12 的 IDFT

$$\begin{bmatrix} x[0] \\ x[9] \\ x[6] \\ x[3] \\ x[4] \\ x[1] \\ x[10] \\ x[7] \\ x[8] \\ x[5] \\ x[2] \\ x[11] \end{bmatrix} = \begin{bmatrix} W_{12}^{0} & W_{12}^{0} & W_{12}^{0} \\ W_{12}^{0} & W_{12}^{-4} & W_{12}^{-8} \\ W_{12}^{0} & W_{12}^{-8} & W_{12}^{-4} \end{bmatrix} \otimes \begin{bmatrix} W_{12}^{0} & W_{12}^{0} & W_{12}^{0} & W_{12}^{0} \\ W_{12}^{0} & W_{12}^{-3} & W_{12}^{-6} & W_{12}^{-9} \\ W_{12}^{0} & W_{12}^{-6} & W_{12}^{0} & W_{12}^{-6} \\ W_{12}^{0} & W_{12}^{9} & W_{12}^{-6} & W_{12}^{-3} \end{bmatrix} \begin{bmatrix} X[0] \\ X[9] \\ X[6] \\ X[3] \\ X[4] \\ X[1] \\ X[10] \\ X[7] \\ X[8] \\ X[5] \\ X[2] \\ X[11] \end{bmatrix}$$

到目前为止，我们已经用 Kronecker 乘积(重新)定义了 IDFT。利用速记符号 $\tilde{\boldsymbol{x}}$ 代替改变的序列 $\boldsymbol{x}$，就可以用下面的矩阵/向量表示：

$$\tilde{\boldsymbol{x}} = \boldsymbol{W}_{N_1} \otimes \boldsymbol{W}_{N_2} \tilde{\boldsymbol{X}} \tag{6.46}$$

对于短 DFT，我们使用 Winograd DFT 算法 6.7，也就是：

$$\boldsymbol{W}_{N_l} = \boldsymbol{C}_l \cdot \boldsymbol{B}_l \cdot \boldsymbol{A}_l \tag{6.47}$$

其中 $\boldsymbol{A}_l$ 合并了输入加法，$\boldsymbol{B}_l$ 是一个傅立叶系数的对角矩阵，而 $\boldsymbol{C}_l$ 包括了输出加法。现在将(6.47)代入(6.46)，运用这一结论就可以改变矩阵乘法和 Kronecker 乘积的计算顺序，变成：

$$\boldsymbol{W}_{N_1} \otimes \boldsymbol{W}_{N_2} = (\boldsymbol{C}_1 \cdot \boldsymbol{B}_1 \cdot \boldsymbol{A}_1) \otimes (\boldsymbol{C}_2 \cdot \boldsymbol{B}_2 \cdot \boldsymbol{A}_2) = (\boldsymbol{C}_1 \otimes \boldsymbol{C}_2)(\boldsymbol{B}_1 \otimes \boldsymbol{B}_2)(\boldsymbol{A}_1 \otimes \boldsymbol{A}_2) \tag{6.48}$$

由于矩阵 $\boldsymbol{A}_l$ 和 $\boldsymbol{C}_l$ 是简单的加法矩阵，所以也同样适用于其 Kronecker 乘积 $\boldsymbol{A}_1 \otimes \boldsymbol{A}_2$ 和 $\boldsymbol{C}_1 \otimes \boldsymbol{C}_2$。很明显，两个分别为 N_1 和 N_2 维的二次对角矩阵的 Kronecker 乘积也可以给出一个 $N_1 N_2$

维的对角矩阵。如果 M_1 和 M_2 分别是根据表 6-2 计算较小 Winograd DFT 的乘法的次数，则总计需要计算的乘法次数与 $\boldsymbol{B}=\boldsymbol{B}_1 \otimes \boldsymbol{B}_2$ 对角元素的数量，也就是 M_1M_2，是相同的。

现在我们将上述不同的步骤组合起来就构成了 Winograd FFT。

法则 6.17　Winograd FFT 的设计

$N=N_1N_2$ 点且 N_1 和 N_2 为互质数的变换可以按照下列步骤构造：

(1) 根据 Good-Thomas 映射，按索引的行读取对输入序列进行索引变换。

(2) 利用 Kronecker 乘积对 DFT 矩阵进行因数分解。

(3) 通过 Winograd DFT 算法代替长度为 N_1 和 N_2 的 DFT 矩阵。

(4) 集中乘法。

在 Winograd FFT 算法成功构造之后，我们就可以采取下面 3 个步骤来计算 Winograd FFT：

法则 6.18　Winograd FFT 算法

(1) 计算前置加法 $\boldsymbol{A}_1$ 和 $\boldsymbol{A}_2$。

(2) 根据矩阵 $\boldsymbol{B}_1 \otimes \boldsymbol{B}_2$ 计算 M_1M_2 次乘法。

(3) 根据 $\boldsymbol{C}_1$ 和 $\boldsymbol{C}_2$ 计算后置加法。

接下来我们看一个示例，详细地了解一下长度为 12 的 Winograd FFT 的构造。

例 6.19　长度为 12 的 Winograd FFT

要构造 Winograd FFT，我们要根据法则 6.17 计算变换中需要使用的矩阵。N_1=3 和 N_2=4 时，就可以得到下面的矩阵：

$$\boldsymbol{A}_1 \otimes \boldsymbol{A}_2 = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{6.49}$$

$$\boldsymbol{B}_1 \otimes \boldsymbol{B}_2 = \mathrm{diag}\,(1, -3/2, \sqrt{3}/2) \otimes \mathrm{diag}\,(1,1,1, -i) \tag{6.50}$$

$$\boldsymbol{C}_1 \otimes \boldsymbol{C}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & i \\ 1 & 1 & -i \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \tag{6.51}$$

根据(6.48)合并这些矩阵，得到 Winograd FFT 算法。输入和输出加法不需要乘法器就可以实现，实数乘法的总数是 2×3×4=24。

到目前为止，我们已经用 Winograd FFT 计算了 IDFT。如果要借助 IDFT 计算 DFT，可以采用(6.6)中用到的技术，并借助于 DFT 计算 IDFT。利用矩阵/向量表示就是：

$$x^* = (W_N^* X)^* \tag{6.52}$$

$$x^* = W_N X^* \tag{6.53}$$

如果 $W_N=[e^{2\pi jnk/N}]$(其中 n、$k \in Z_N$)是 DFT，则 DFT 就可以用 IDFT 按如下步骤计算：计算输

入序列的共轭复数，将序列用 IDFT 算法转换，计算输出序列的共轭复数。

也可以采用 Kronecker 乘积算法，也就是 Winograd FFT，直接计算 DFT。生成一个平滑修正的输出索引映射，如下例所示。

例 6.20　用 Kronecker 乘积公式计算 12 点 DFT：

$$\begin{bmatrix} X[0] \\ X[3] \\ X[6] \\ X[9] \\ X[4] \\ X[7] \\ X[10] \\ X[1] \\ X[8] \\ X[11] \\ X[2] \\ X[5] \end{bmatrix} = \begin{bmatrix} W_{12}^{0} & W_{12}^{0} & W_{12}^{0} \\ W_{12}^{0} & W_{12}^{4} & W_{12}^{8} \\ W_{12}^{0} & W_{12}^{8} & W_{12}^{4} \end{bmatrix} \otimes \begin{bmatrix} W_{12}^{0} & W_{12}^{0} & W_{12}^{0} & W_{12}^{0} \\ W_{12}^{0} & W_{12}^{3} & W_{12}^{6} & W_{12}^{9} \\ W_{12}^{0} & W_{12}^{6} & W_{12}^{0} & W_{12}^{6} \\ W_{12}^{0} & W_{12}^{9} & W_{12}^{6} & W_{12}^{3} \end{bmatrix} \begin{bmatrix} x[0] \\ x[9] \\ x[6] \\ x[3] \\ x[4] \\ x[1] \\ x[10] \\ x[7] \\ x[8] \\ x[5] \\ x[2] \\ x[11] \end{bmatrix} \tag{6.54}$$

输入序列 $x[n]$被看成是 Good-Thomas 映射的顺序，在 $X(k)$的(频率)输出索引映射中，与 Good-Thomas 映射相比，每第一个和第二个元素互换了位置。

6.2.4　DFT 和 FFT 算法的比较

很明显，目前已经有许多途径可以实现 DFT。现在就从图 6-1 中给出的算法中选定一种短 DFT 算法开始介绍。而且短 DFT 可以用 Cooley-Tukey、Good-Thomas 或 Winograd 提出的索引模式来开发长 DFT。选择实现的共同目标就是将乘法的复杂性降到最低。这是一种可行的准则，因为乘法的实现成本与其他运算，比如加法、数据访问或索引计算相比较而言要高得多。

图 6-17 给出了各种 FFT 长度所需要乘法的次数。从中可以得出结论，单纯从乘法复杂性准则考虑，Winograd FFT 是最有吸引力的。在本章中，给出了几种形式的 $N=4\times3=12$ 点 FFT 的设计。表 6-4 给出了直接算法、Rader 质数因子算法和用于简单 DFT 模块和 3 种分别称为 Cooley-Tukey、Good-Thomas 和 Winograd FFT 的不同索引映射的 Winograd FFT 算法。

表 6-4　长度为 12 复杂输入 FFT 算法的实数乘法的数量，假设一个复杂的乘法使用了 4 个实数乘法

DFT 方法	索引映射		
	Good-Thomas 参考图 6-16	Cooley-Tukey 参考图 6-2	Winograd 参考例 6.16
直接算法	$4 \cdot 12^2 = 4 \cdot 144 = 576$		
RPFA	$4(3(4-1)^2 + 4(3-1)^2) = 172$	$4(43+6) = 196$	—
WFTA	$3 \cdot 0 \cdot 2 + 4 \cdot 2 \cdot 2 = 16$	$16 + 4 \cdot 6 = 40$	$2 \cdot 3 \cdot 4 = 24$

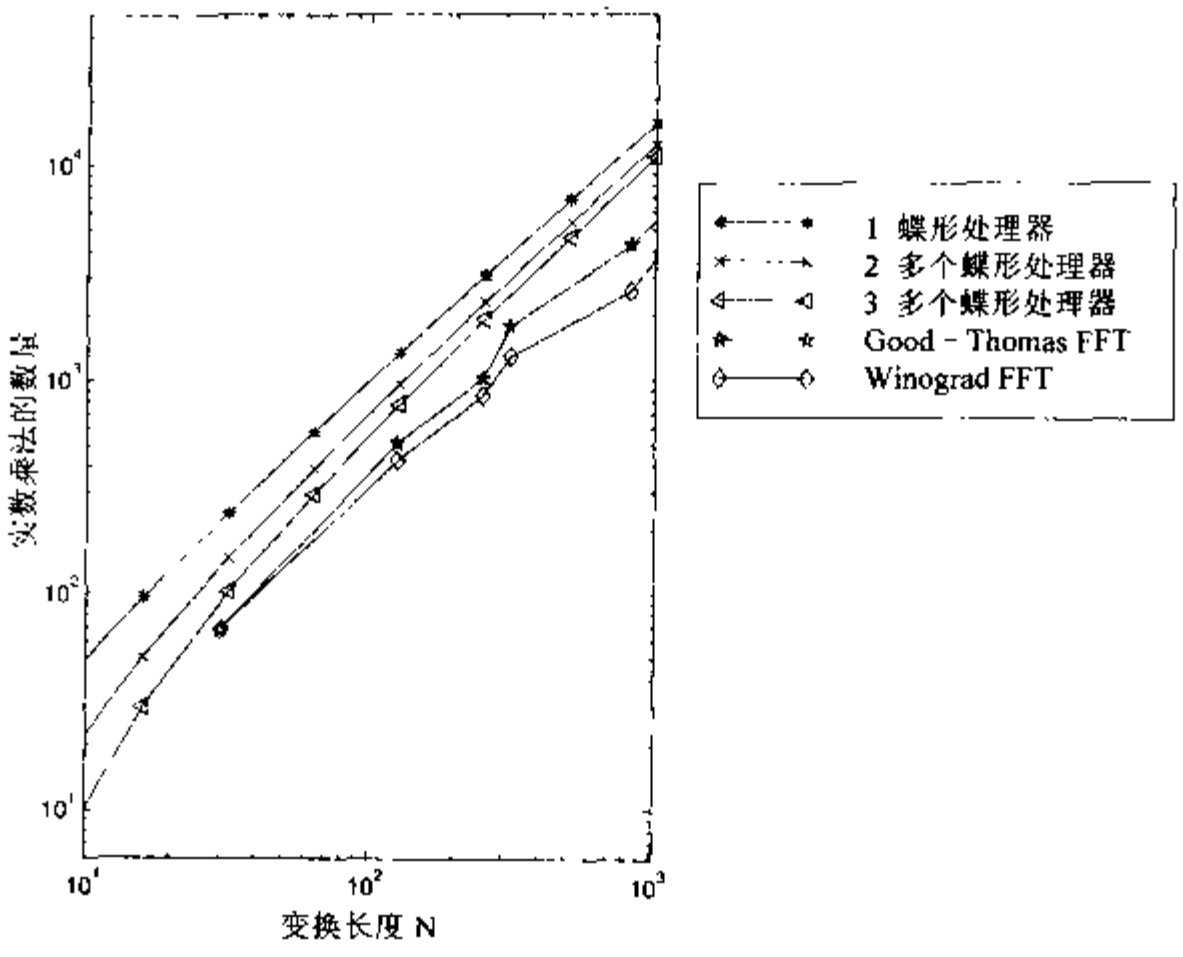


图 6-17　基于所需要的实数乘法次数的不同 FFT 算法的比较

除了乘法的次数以外，还需要考虑其他的约束条件，例如可能的变换长度、加法的次数、索引计算开销、系数或数据存储器规模和运行期间代码的长度。在许多情况下，Cooley-Tukey 方法提供了最佳总体解决方案，请参阅表 6-5。

表 6-5　长度 $N=\prod N_k$ 的 FFT 算法的重要属性

属　性	Cooley-Tukey	Good-Thomas	Winograd
任意变换长度	是	否 $\gcd(N_k, N_l)=1$	
W 的最大阶数	N	$\max(N_k)$	
是否需要旋转因子	是	否	否
#乘法	差	中	佳
#加法	中	中	中
#索引计算量	佳	中	差
就地数据	是	是	否
实现的优点	小规模蝶形处理器	可以使用 RPFA、快速、简单的 FIR 阵列	小规模的完全并行、中等规模 FFT(<50)

目前 FPGA 所达到的复杂性已经超过 1M 门，FFT 完全可以集成在单片 FPGA 中。由于这样的 FFT 模块设计是劳动密集的，通常使用大批供应的“知识产权”(intellectual property, IP)模块(有时候也称作虚拟组件(virtual component, VC))更有意义。例如可以访问 www.xilinx.com 或 www.altera.com，参阅 IP 合作程序。多数大批供应的设计都是基于 radix-2 或 radix-4 的。

表 6-6 总结了一些已公布的 FPGA 实现。Goslin[118] 的设计是基于 radix-2FFT 的，其中蝶

形部分是采用第 2 章讨论的分布式算法实现的。Dandalis 等人的设计是基于采用所谓的算术傅立叶变换的 DFT 近似，我们将在 7.1 节讨论这一内容。Meyer-Baese 等人[135]的 ERNS FFT 采用了 Rader 算法与将在第 7 章讨论的数论变换的组合。

表 6-6 一些 FPGA FFT 实现的比较

名 称	数据类型	FFT 类型	*N*-point FFT 时间	时钟速率 *P*	内部 RAM/ROM	设计目标/源
Xilinx FPGA	8Bit	Radix=2 FFT	*N*=256 102.4μs	70MHz 4.8W @3.3V	no	573 CLBs [118]
Xilinx FPGA	16Bit	AFT	*N*=256 82.48μs 42.08μs	50MHz 15.6W @3.3V 29.5W	no	[134] 2602 CLBs 4922 CLBs
Xilinx FPGA ERNSNTT	12.7Bit	FFT 使用 NTT	*N*=97 9.24μs	26MHz 3.5W @3.3V	no	1178 CLBs [135]

6.3 傅立叶相关的变换

离散余弦变换(discrete cosine transform, DCT)和离散正弦变换(discrete sine transform, DST)不是 DFT，但可以用 DFT 计算。不过 DCT 和 DST 不能够直接通过乘以变换的频谱和反变换来计算快速卷积，也就是卷积理论不成立，所以 DCT 和 DST 不像 FFT 一样得到广泛的应用，但是在图像压缩等一些应用领域中，DCT 是非常流行的(因为它们与 Kahunen-Loevé 变换非常接近)。由于 DCT 和 DST 是根据正弦和余弦“核函数”定义的，与 DFT 有着密切的关系，所以将在本章加以介绍。首先讨论 DCT 的 DST 的定义和属性，然后给出实现 DCT 的类似 FFT 的快速计算算法。所有的 DCT 都遵循下面的变换模式：

$$X[k]=\sum_{n} x[n]C_N^{nk} \leftrightarrow x[n]=\sum_{k} X[k]C_N^{n,k} \tag{6.55}$$

该模式是由 Wang[136]观察到的。4 种不同 DCT 实例的核函数 C_N^{nk} 分别由

DCT-I： $C_N^{n,k}=\sqrt{2/N}c[n]c[k]\cos(nk\frac{\pi}{N})$ $n,k=0,1,\ldots,N$

DCT-II： $C_N^{n,k}=\sqrt{2/N}c[n]\cos(k(n+\frac{1}{2})\frac{\pi}{N})$ $n,k=0,1,\ldots,N-1$

DCT-III： $C_N^{n,k}=\sqrt{2/N}c[k]\cos(n(k+\frac{1}{2})\frac{\pi}{N})$ $n,k=0,1,\ldots,N-1$

DCT-IV：$C_{\Lambda}^{n,k}=\sqrt{2/N}\cos((k+\frac{1}{2})(n+\frac{1}{2})\frac{\pi}{N})\quad n,k=0,1,\ldots,N-1$

定义。其中除了 $c[0]=1/\sqrt{2}$ 外，$c[p]=1$。DST 具有相同的结构，但是余弦项均由正弦项代替。DCT 的属性如下：

(1) 采用余弦基的 DCT 实现函数。

(2) 所有的变换均是正交的，也就是 $\boldsymbol{C}\cdot\boldsymbol{C}'=k[n]\boldsymbol{I}$。

(3) 与 DFT 不同的是，DCT 是一个实变换。

(4) DCT-I 是其本身的逆矩阵。

(5) DCT-II 是 DCT-III 的逆矩阵，反之也成立。

(6) DCT-IV 是其本身的逆矩阵，IV 是对称的，也就是 $\boldsymbol{C}=\boldsymbol{C}'$。

(7) DCT 的卷积属性与 DFT 中的卷积乘法关系不一样。

(8) DCT 是 Kahunen-Loevé 变换(KLT)的一种近似。

DCT-II 的二维 8×8 变换在图像压缩(也就是视频的 H.261、H.263 和 MPEG 标准和静态图像的 JPEG 标准)中经常用到。由于二维变换是分成二维的，所以我们可以先计算行变换再计算列变换，或者反之也可以(练习 6.18)。这样我们就可以只集中考虑一维变换的实现。

6.3.1　利用 DFT 计算 DCT

Narasimha 和 Peterson 引入了一种描述如何在 DFT 的帮助下计算 DCT 的结构[137,138]。DCT 到 DFT 的映射是非常具有吸引力的，因为我们可以利用 FFT 类型算法的多种变化。由于 DCT-II 最为常用，所以我们将进一步探讨 DFT 与 DCT-II 之间的关系。为了简化表达式，这里就省略了刻度操作，因为这一步骤可以包括在 DFT 或 FFT 计算的末尾。假定变换长度是偶数，用下面的置换：

$$y[n]=x[2n]\text{和 }y[N-n-1]=x[2n+1]\ \text{其中 }n=0,1,\ldots,N/2-1$$

重写 DCT-II 变换：

$$X[k]=\sum_{n=0}^{N-1}x[n]\cos\left(k(n+\frac{1}{2})\frac{\pi}{N}\right)\tag{6.56}$$

然后得到：

$$X[k]=\sum_{n=0}^{N/2-1}y[n]\cos\left(k(2n+\frac{1}{2})\frac{\pi}{N}\right)+\sum_{n=0}^{N/2-1}y[N-n-1]\cos\left(k(2n+\frac{3}{2})\frac{\pi}{N}\right)$$

$$X[k]=\sum_{n}y[n]\cos\left(k(2n+\frac{1}{2})\frac{\pi}{N}\right)\tag{6.57}$$

如果现在计算用 $Y[k]$ 表示的 $y[n]$ 的 DFT，可以看到：

$$X[k]=\Re(W_{4N}Y[k])\tag{6.58}$$

$$=\cos\left(\frac{\pi k}{2N}\right)\Re(Y[k])-\sin\left(\frac{\pi k}{2N}\right)\Im(Y[k])\tag{6.59}$$

这就很容易转换成 C 或 MATLAB 程序(参阅练习 6.18)，借助于 DFT 或 FFT 就可以计算 DCT。

6.3.2 快速直接 DCT 实现

DCT 的对称属性已经被 Byeong Lee[139]用来构造类似 FFT 的 DCT 算法。由于其与 radix-2 Cooley-Tukey FFT 的相似性，所以最终的算法称为快速 DCT 或简称 FCT。换句话说就是快速 DCT 算法可以用矩阵结构开发[140]。由于 DCT 是正交变换，所以可以通过转置逆 DCT(IDCT)得到 DCT。(6.55)式中引入的 IDCT-II 型有：

$$x[n]=\sum_{k=0}^{N-1}\hat{X}[k]C_N^{n,k}\ ,\quad n=0,1,\ldots,N-1 \tag{6.60}$$

注意 $\hat{X}[k]=c[k]X[k]$。将 $x[n]$分解成偶数和奇数部分，可以看到，$x[n]$可以由两个 $N/2$ DCT 重构，也就是：

$$G[k]=\hat{X}[2k]\ , \tag{6.61}$$

$$H[k]=\hat{X}[2k+1]+\hat{X}[2k-1]\ ,\quad k=0,1,\ldots,N/2-1 \tag{6.62}$$

在时域中，有：

$$g[n]=\sum_{k=0}^{N/2-1}G[k]C_{N/2}^{n,k}, \tag{6.63}$$

$$h[n]=\sum_{k=0}^{N/2-1}H[k]C_{N/2}^{n,k}\ ,\quad k=0,1,\ldots,N/2-1 \tag{6.64}$$

重构的形式变成：

$$x[n]=g[n]+1/(2C_N^{n,k})h[n] \tag{6.65}$$

$$x[N-1-n]=g[n]-1/(2C_N^{n,k})h[n]\ ,\quad n=0,1,\ldots,N/2-1 \tag{6.66}$$

重复这一过程就可以进一步分解 DCT。图 6-12 给出的(6.63)与 radix-2 FFT 旋转因子之间的比较表明，除法对 FCT 似乎是必要的。所以旋转因子$1/(2C_N^{n,k})$就应该预先被计算出来并储存的表中。这样的表方法对于 Cooley-Tukey FFT 也是适合的，因为在线计算三角函数一般是非常耗时间的。接下来用一个示例来说明 FCT。

例 6.21　8 点 FCT

对于 8 点 FCT，等式(6.61)至(6.66)式就变成：

$$G[k]=\hat{X}[2k] \tag{6.67}$$

$$H[k]=\hat{X}[2k+1]+\hat{X}[2k-1]\ ,\quad k=0,1,2,3 \tag{6.68}$$

在时域中就有：

$$g[n]=\sum_{k=0}^{3}G[k]C_4^{n,k} \tag{6.69}$$

$$h[n]=\sum_{k=0}^{3}H[k]C_4^{n,k}\ ,\quad n=0,1,2,3 \tag{6.70}$$

这样，重构就变成：

$$x[n] = g[n] + 1/(2C_8^{n,k})h[n], \tag{6.71}$$

$$x[N-1-n] = g[n] - 1/(2C_8^{n,k})h[n], \quad n=0, 1, 2, 3 \tag{6.72}$$

等式(6.67)和(6.68)构成了图 6-18 中流程图的第一级，而(6.71)式和(6.72)式构成了流程图的最后一级。

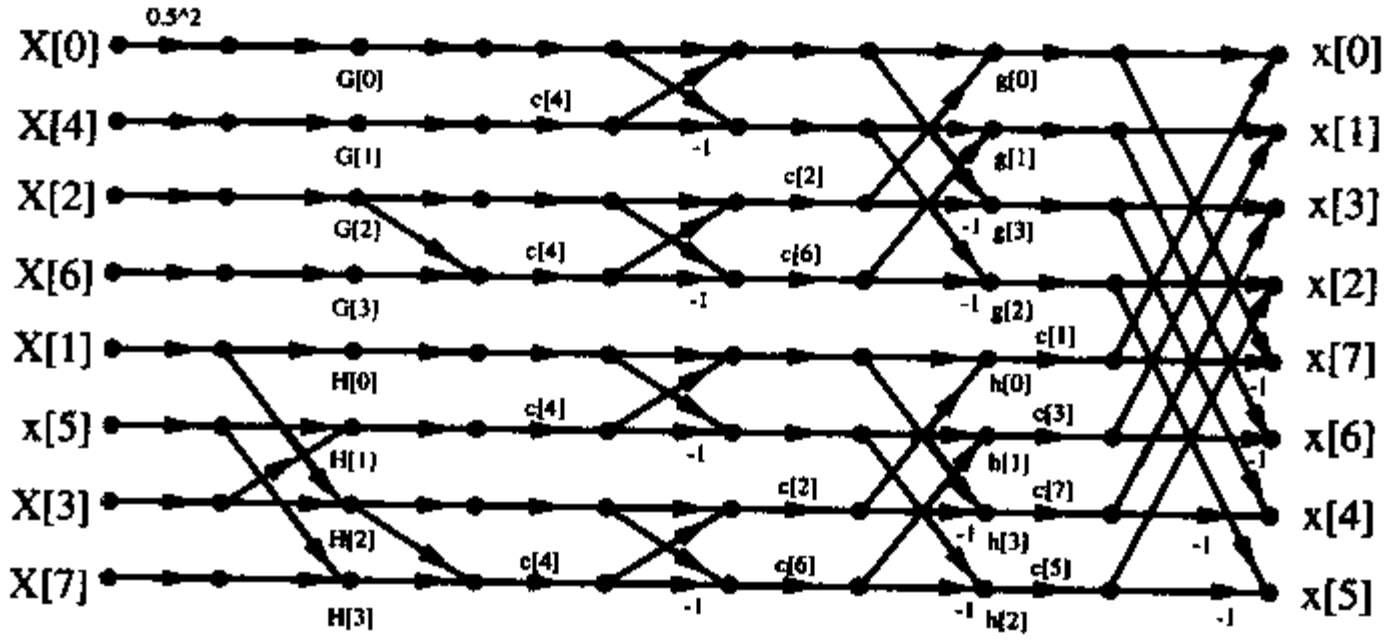


图 6-18　采用速记符号 $c[p] = 1/(2\cos(p\pi/16))$的 8 点快速 DCT 流程图

在图 6-18 中，输入序列 $\hat{x}[k]$是位逆序的。输出序列 $x[n]$的顺序按着下面的方式生成：由集合(0,1)开始通过增加一个前缀 0 和 1 形成新的集合。前缀是 1 时，前面格式中所有的位都是颠倒的。例如：从序列 10 得到两个子序列 010 和$1\bar{1}\bar{0}$=101。图 6-19 给出了这种帧格式的图解。

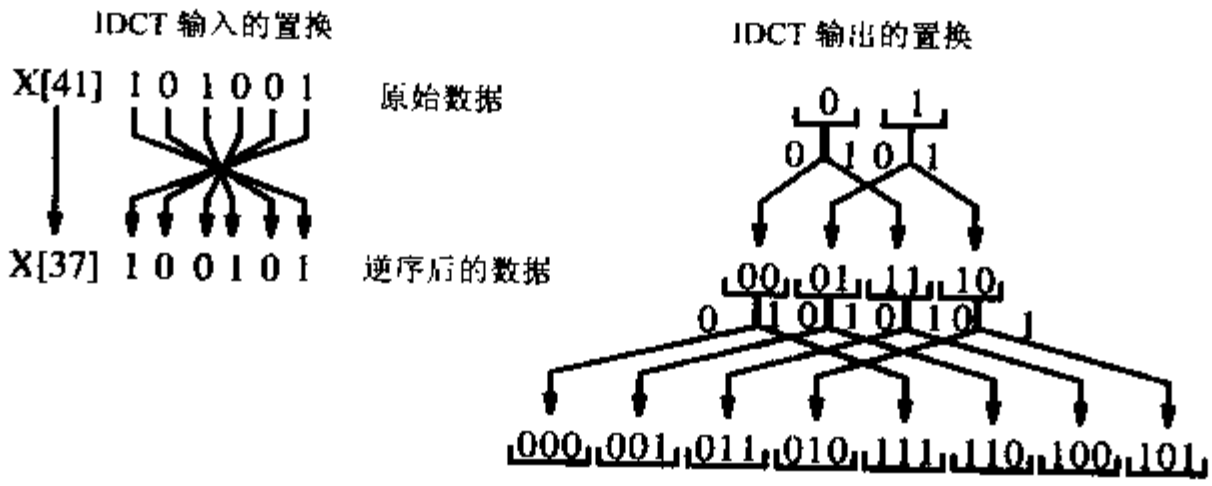


图 6-19　8 点快速 DCT 的输入输出的置换

6.4　练习

6.1：用傅立叶变换计算矩形和三角形窗函数的 3dB 带宽、第一个零点、最大旁瓣和每倍频程衰减。

6.2：(a) 计算 $x[n]=\{3, 1, -1\}$和$f[n]=\{2, 1, 5\}$的循环卷积。

(b) 计算 $N=3$ 的 DFT 矩阵 $\boldsymbol{W}_3$。

(c) 计算 $x[n]=\{3, 1, -1\}$和$f[n]=\{2, 1, 5\}$的 DFT。

(d) 根据 $\boldsymbol{y}=\boldsymbol{W}_3^{-1}\boldsymbol{Y}$，为来自(c)的信号计算 $Y[k]=X[k]F[k]$ (注意：(c)和(d)要用到 C 编译器或者 MATLAB)。

6.3：在 $X[k]=x[0]+x[1]W_N^k+x[2]W_N^{2k}+\mathrm{K}+x[N-1]W_N^{(N-1)k}$ 的 DFT 计算中，单边频谱成份 $X[k]$ 可以通过集合所有的公因子 W_N^k 进行重构，这样就得到 $X[k]=x[0]+W_N^k(x[1]+W_N^k(x[2]+\mathrm{K}+W_N^k x[N-1])\mathrm{K}))$。从而得到一个 $X[k]$的可行递归计算。称作 Goertzel 算法，图 6-5 给出了相应的图形化解释。如果只需要计算极少数频谱成份，则 Goertzel 算法是非常具有吸引力的。对于完整的 DFT 而言，工作量是 N^2 数量级的，与直接的 DFT 计算相比较没有什么优势。

(a) 构造包括输入和输出寄存器的递归信号流程图，计算 N=5 的单边 $X[k]$。

(b) N=5 和 k=1，计算脉冲 $x[n]=\delta[n]$时，设计(a)中可行的 FPGA 实现的所有寄存器的内容。

6.4：假设给定的二阶递归系统 $Y(z)=X(z)/(1+2\cos(2\pi k/N)z^{-1}+z^{-2})$，$k$ 和 N 固定。

(a) 绘出系统的信号流程图。

(b) 令 N=6 和 k=1，依靠 $x[0]$到 $x[5]$计算 $y[n]$，其中 $n=0,1,\ldots,N$-1。

(c) 判断该二阶系统与 6.1.3 节中的 Goertzel 算法之间的关系。

6.5：在 6.1.4 节定义了 Bluestein chirp-z 算法。图 6-6 给出了该算法相应的图形化解释。

(a) 求出 N=4 的 CZT 算法。

(b) 用 C 或者 MATLAB 计算三角形序列 $x[n]=\{0,1,2,3\}$的 CZT。

(c) 使用 C 或者 MATLAB 将长度扩展到 N=256，并使用相同长度的 FFT 进行检验。采用的三角形输入序列号 $x[n]=n$。

6.6：(a) 设计一个 N=7 的 Rader 算法的非递归滤波器的直接实现。

(b) 求出可以组合的系数。

(c) 就工作量方面对(a)和(b)的结果进行比较。

6.7：设计长度 N=3 的 Winograd DFT 算法。

6.8：(a) 用二维索引变换 $n=3n_1+2n_2 \bmod 6$ 求出映射(6.18)，这是不是一个双边映射？

(b) 用二维索引变换 $n=2n_1+2n_2 \bmod 6$ 求出映射(6.18)，这是不是一个双边映射？

(c) 对于 $\gcd(N_1, N_2)>1$，Burrus[1091] 发现在下列情形中的映射是双边的：

$A=aN_2$ 和 $B\neq bN_1$ 以及 $\gcd(a,N_1)=\gcd(B,N_2)=1$,或

$A\neq aN_2$ 和 $B=bN_1$ 以及 $\gcd(A,N_1)=\gcd(b,N_2)=1$，其中 $a, b\in Z$。假设 $N_1=9, N_2=15$ 计算 A=15，$B\in Z_{20}$时的所有可能值。

6.9：对于 $\gcd(N_1, N_2)=1$，Burrus[1091] 发现在下列情形中的映射是双边的：

$$A=aN_2 \text{ 和/或 } B=bN_1 \text{ 以及 } \gcd(A,N_1)=\gcd(B,N_2)=1 \tag{6.73}$$

其中 $a, b\in Z$。假设 $N_1=5$，$N_2=8$ 确定下列映射是不是可行的双边索引映射：

(a) A=8，B=5；

(b) A=8，B=10；

(c) A=24，B=15；

(d) A=7 时，计算所有有效的 $B \in Z_{40}$；

(e) A=8 时，计算所有有效的 $B \in Z_{40}$。

6.10：(a) 绘出 radix-2 DIF 算法的信号流程图，其中 N=16。

(b) 编写实现 DIF radix-2 FFT 的 C 或 MATLAB 程序。

(c) 用三角形输入 $x[n]=n+jn$, $n \in [0, N-1]$检验您的 FFT 程序。

6.11：(a) 绘出 radix-2 DIT 算法的信号流程图，其中 N=8。

(b) 编写实现 DIT radix-2 FFT 的 C 或 MATLAB 程序。

(c) 用三角形输入 $x[n]=n+jn$, $n \in [0, N-1]$检验您的 FFT 程序。

6.12：计算 N=16 的 radix-4 FFT 的索引映射。绘出 N=16 的 radix-4 FFT 的信号流程图。

6.13：绘出 N=12 的 Good-Thomas FFT 的信号流程图，在信号流程图中不要出现交叉(提示：可以采用行和列 DFT 的 3D 表示法)。

6.14：Burrus 和 Eschenbacher[141] 提出的 FFT 的索引变换如下：

$$n = N_2 n_1 + N_1 n_2 \bmod N \quad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases} \tag{6.74}$$

和

$$k = N_2 k_1 + N_1 k_2 \bmod N \quad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases}$$

(a) N_1=3 和 N_2=4，计算 n 和 k 的索引映射。

(b) 计算 W^{nk}。

(c) 将(b)的 W^{nk} 代入到 DFT 矩阵中。

(d) 这是什么类型的 FFT 算法？

(e) Rader 算法可以用来计算长度 N_1 或 N_2 的 DFT 吗？

6.15：(a) 计算 DFT 矩阵 $\boldsymbol{W}_2$ 和 $\boldsymbol{W}_3$。

(b) 计算 Kronecker 乘积 $\boldsymbol{W}_6' = \boldsymbol{W}_2 \otimes \boldsymbol{W}_3$。

(c) 计算向量 $\boldsymbol{X}$ 和 $\boldsymbol{x}$ 的索引，$\boldsymbol{X}=\boldsymbol{W}_6' \boldsymbol{x}$ 是长度为 6 的 DFT。

(d) 计算 $x[n]$和 $X[n]$的索引映射，$\boldsymbol{x} = \boldsymbol{W}_2^* \otimes \boldsymbol{W}_3^* \boldsymbol{X}$ 是 IDFT。

6.16：离散 Hartley 变换(DHT)是一种针对实数信号的变换。长度 N 的变换定义如下：

$$H[n] = \sum_{k=0}^{N-1} \mathrm{cas}(2\pi nk / N) h[k] \tag{6.76}$$

其中 cas(x)=sin(x)+cos(x)，与 DFT($f[k] \xleftrightarrow{\text{DFT}} F[n]$)的关系是：

$$H[n] = \Re\{F[n]\} - \Im\{F[n]\} \tag{6.77}$$

$$F[n] = E[n] - jO[n] \tag{6.78}$$

$$E[n]=\frac{1}{2}(H[n]+H[-n]) \tag{6.79}$$

$$O[n]=\frac{1}{2}(H[n]-H[-n]) \tag{6.80}$$

其中是 $\Re$ 实部，$\Im$ 是虚部，$E[n]$是 $H[n]$的偶数部分，而 $O[n]$是 $H[n]$的奇数部分。

(a) 给出计算逆 DHT 的方程。

(b) 给出用 DFT 计算卷积的步骤(采用 DFT 的频率卷积)。

(c) 如果输入序列是偶数，请给出(b)中算法的可能简化形式。

6.17：DCT-II 的形式如下：

$$X[k]=c[k]\sqrt{\frac{2}{N}}\sum_{n=0}^{N-1}x[n]\cos\left(\frac{2\pi}{4N}(2n+1)k\right) \tag{6.81}$$

$$c[k]=\begin{cases}\sqrt{1/2} & k=0\\ 1 & \text{其他}\end{cases} \tag{6.82}$$

(a) 给出计算逆变换的方程。

(b) 给出 N=4 的 DCT 矩阵。

(c) 计算 $x[n]$={1,2,2,1}和 $x[n]$={1,1, - 1, - 1}的变换。

(d) 试述奇对称或偶对称序列的 DCT 的一些特点。

6.18：下面的 MATLAB 代码借助于 radix-2 FFT(请参阅练习 6.10)可以计算 DCT-II 变换(假设偶数长度 $N=2^n$)。

```
function   X =DCTII(x)
N = length(x);                                  % get length
Y = [ x(1:2:N)]; x(N: - 2:2)];                  % re-order elements
Y = fft(y);                                     % Compute the FFT
w = 2*exp( - i*(0:N - 1)'*pi/(2*N))/sqrt(2*N);  % get weights
w(1) = w(1)/sqrt(2);                            % make it unitary
X = real(w.*Y);                                 % compute pointwise product
```

(a) 用 C 或 MATLAB 编译这个程序。

(b) 计算 $x[n]$={1,2,2,1}和 $x[n]$={1,1, - 1, - 1}的变换。

6.19：像 DFT 一样，DCT 也是一种可拆分的变换。这样我们就可以用一维 DCT 实现二维 DCT。二维 $N\times N$ 变换如下：

$$X[n_1,n_2]=\frac{c[n_1]c[n_2]}{2}\sum_{k=0}^{N-1}\sum_{l=0}^{N-1}x[k,l]\cos\left(n_1(k+\frac{1}{2})\frac{\pi}{N}\right)\cos\left(n_2(l+\frac{1}{2})\frac{\pi}{N}\right) \tag{6.83}$$

其中 $c[0]=1/\sqrt{2}$，$m\neq 1$ 时 $c[m]$=1。

用练习 6.18 中的程序分别按下面的步骤计算一个 8×8 DCT 变换：

(a) 首先进行行变换，然后进行列变换。

(b) 首先进行列变换，然后进行行变换。

(c) (6.83)的直接实现。

(d) 采用测试数据 $x[k, l]=k+l$, $k,l \in [0,7]$检验(a)和(b)，并比较结果。

练习使用 MaxPlusII 软件

6.20：(a) 使用 MaxPlusII 根据练习 6.3 实现一个一阶系统，计算 $N=5$ 和 $n=1$ 以及 8 位系数和输入数据的 Goertzel 算法。

(b) 给出 LC 的数量和 Registered Performance。

(c) 用 3 个输入序列{20,40,60,80,100}、{j20, j40, j60, j80, j100}和{20+j20,40+j40,60+j60, 80+j80,100+j100}对设计进行仿真。

6.21：(a) 用 MaxPlusII 设计一个 Component，计算(实数输入)4 点 Winograd DFT(练习 6.16)。输入和输出精度均是 8 位。

(b) 给出 LC 的数量和 Registered Performance。

(c) 用输入序列{40+j100,60+j80, 80+j60,100+j40}对设计进行仿真。

6.22：(a) 用 MaxPlusII 设计一个 Component，计算(实数输入)3 点复数 Winograd DFT(练习 6.16)。输入和输出精度均是 8 位。

(b) 给出 LC 的数量和 Registered Performance。

(c) 用输入序列{40+j100,60+j80,80+j60}对设计进行仿真。

6.23：(a) 使用 MaxPlusII，利用练习 6.21 和 6.22 中设计的 3 点和 4 点 Component，采用组件实例的方法设计一个与图 6-16 相似的完全并行的 12 点 Good-Thomas FFT。输入和输出精度均是 8 位。

(b) 给出 LC 的数量和 Registered Performance。

(c) 用输入序列 $x[n]=10n$, $0 \le n < 12$ 对设计进行仿真。

6.24：(a) 用 MaxPlusII 设计一个与例 6.11 相似的组件 ccmulp，计算旋转因子乘法。用 3 个流水线级作为乘法器，用一个流水线级作为输入减法 $X - Y$。输入和输出精度均是 8 位。

(b) 进行仿真，确保流水线乘法器可以正确地计算(70+j50)(121+j39)。

(c) 给出旋转因子乘法器的 LC 的数量和 Registered Performance。

(d) 实现完整的流水线蝶形处理器。

(e) 用例 6.12 中的数据进行仿真。

(f) 给出完整的流水线蝶形处理器的 LC 的数量和 Registered Performance。

第7章 前沿课题

由于 FPGA 可以逐位地实现，所以目前存在几种使得 FPGA 可以超出 PSDP 一个数量级的算法。讲述这样的应用就是本章的主旨。

从数论变换(Number Theoretic Transform, NTT)的角度来讲，FPGA 的根本优势就是它可以以任意位宽实现模运算。在 7.1 节将要详细地讨论 NTT。

对于差错控制和加密技术，需要用到两个基本构造模块：有限域算法(Galois field arithmetic)和线性反馈移位寄存器(linear feedback shift register, LFSR)。两者均可以用 FPGA 有效地实现，我们将在 7.2 节加以讨论。例如：如果 N 位 LFSR 用作 M 级多级数字发生器，就会给 FPGA 相对于 PDSP 或微处理器至少 MN 倍的速度优势。

最后，在 7.3 节中应用 FPGA 设计的通信系统将证明低系统成本、高吞吐量和快速原型设计的可行性。在本章结束部分将要综合讨论相干接收器和非相干接收器。

7.1 矩形变换和数论变换

卷积的快速实现和离散傅立叶变换(discrete Fourier transform, DFT)的计算都是信号和图像处理中经常遇到的问题。在实践中，这些操作通常都是用快速傅立叶变换(fast Fourier transform, FFT)算法实现的。NTT 在某些场合要优于基于 FFT 的系统。此外也有可能采用矩形变换，像 Walsh/Hadamard 或算法傅立叶变换，来得到 DFT 或卷积的近似，相关内容将在 7.1 节末尾加以讨论。

1971 年，Pollard【142】在有限群上定义了 NTT。由于存在变换对：

$$x[n] = N^{-1}\sum_{k=0}^{N-1} X[k]\alpha^{-nk} \bmod M \leftrightarrow X[k] = \sum_{n=0}^{N-1} x[k]\alpha^{kn} \bmod M \tag{7.1}$$

其中 $N \cdot N^{-1} \equiv 1$，$\alpha \in Z_M (Z_M = \{0,1,2,\ldots M-1\}$，$Z_M \cong Z/MZ)$是序列 N 的一个元素，也就是在有限群$(Z_M, \cdot)$中，对于所有 $k \in \{1,2,\ldots,N-1\}$有 $\alpha^N \equiv 1$ 和 $\alpha^k \neq 1$(请参阅练习 7.1)。

对于给定的多元组(α, M, N)，能够确保这样的变换对存在是非常重要的。很明显，α 必须是 N 阶的模 M。为了保证逆 NTT(INTT)的存在，其他的条件是：

(1) 乘法的逆 N^{-1} 模 M 必须存在，也就是说，方程 $x \cdot N \equiv 1 \bmod M$ 必须有一个解 $x \in Z_M$。

(2) 变换矩阵$|A|=|[\alpha^{kn}]|$的行列式必须是非零矩阵，这样矩阵才是可逆的，也就是说 A^{-1} 存在。

(3) 如果 α 和 M 没有公共因子，或简写成 $\gcd(\alpha, M)=0$，只能够得出乘法的逆存在。

(4) 对于第二种情况，要用到代数学中著名的事实：NTT 矩阵是 Vandermonde 矩阵的一个特例$(a[k]=\alpha_N^k)$，它满足下面的行列式：

$$\det(\boldsymbol{V})=\begin{vmatrix} 1 & a[0] & a[0]^2 & \cdots & a[0]^{L-1} \\ 1 & a[1] & a[1]^2 & \cdots & a[1]^{L-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a[L-1] & a[L-1]^2 & \cdots & a[L-1]^{L-1} \end{vmatrix}=\prod_{K>l}(a[k]-a[l]) \tag{7.2}$$

如果 $\det(\boldsymbol{V})\neq 0$，则需要 $a[k]\neq a[l]\ \forall k\neq l$。实际上，由于需要计算的是模 M，就又产生了第二个约束条件。特别是在约束条件(也就是 $\gcd(\prod_{k>l}a_k-a_l,M)=1$)中，不能够有一个 0 因数。

由此得出结论，要检验 NTT 的存在，必须要确认：

定理 7.1 Z_M 内 NTT 存在的条件

如果满足下面的条件，则在 Z_M 内定义的 α 的长度 N 的 NTT 存在：

(1) $\gcd(\alpha, M)=1$

(2) α 是 N 阶的，也就是：

$$\alpha^n \bmod M\begin{cases}=1 & n=N \\ \neq 1 & 1\leqslant n<N\end{cases} \tag{7.3}$$

(3) $\det(\boldsymbol{A})^{-1}$ 存在，也就是 $\gcd(\alpha^l-1, M)=1$，$l=1, 2, \ldots, N-1$

Z_p,p 为质数时，上面给出的所有条件均自动满足。在 Z_p 中可以找到 $p-1$ 阶的元素。但是变换长度 $p-1$ 一般还要受到实际应用的限制，因为就“一般”乘法和模运算化简而言还是需要的，而且在二进制或 QRNS 算法中更适合采用“标准” FFT[143]。

在环 $M=2^b$ 中没有有价值的变换。但使用邻近的 $2^b\pm 1$ 还是可能的。如果使用的是质数，则条件 1 和 3 均自动满足。接下来要讨论什么样的质数 $2^b\pm 1$ 是已知的。

默希尼和费尔马(Mersenne 和 Fermat)数 法国数学家马丁·默希尼(Martin Mersenne 1588-1648)首先研究了 2^b-1 形式的质数，用几何级数表示为：

$$(1+2^q+2^{2q}+\ldots+2^{qr-1})(2^q-1)=2^{qr}-1$$

可以看出来 Mersenne 质数的指数 b 也必须是质数，这是必要条件，但并不是充分条件，例如：$2^{11}-1=23\times 89$ 就不是质数。最初的 Mersenne 质数 2^b-1 的指数：

$$b=2, 3, 5, 7, 13, 17, 31, 61, 89, 107, 127, 521, 607, 1279 \tag{7.4}$$

2^b+1 类型的质数来自 Fermat 最初的理论。Fermat 推测所有的 $2^{(2^r)}+1$ 数都是质数，但是正像 Mersenne 质数一样，这是必要条件，但不是充分条件。因为如果 b 是奇数，也就是 $b=q2^r$，则有：

$$2^{q2^r}=(2^{(2^r)}+1)(2^{(q-1)2^r}-2^{(q-2)2^r}+2^{(q-3)2^r}-\cdots+1)$$

不是质数，就像 $(2^4+1)|(2^{12}+1)$ 也就是 17|4097。到目前为止已经知道了 5 个 Fermat 质数，分别是：

$$F_0=3 \qquad F_1=5 \qquad F_2=17 \qquad F_3=257 \qquad F_4=65537 \tag{7.5}$$

但是欧拉(1707-1783)指出 $F_5=2^{32}+1$ 可以被 641 除尽。到 F_{21} 为止还没有再出现 Fermat 质

数，这也就将 NTT 的可能质数中的 Fermat 质数减少到前 5 个。

7.1.1　算术模 $2^b \pm 1$

在第 2 章中复习了二进制反码(1C)和减 1 表示法(D1)的编码。参考表 2-1 的 1C 和 D1 编码，可以认为 1C 编码可以有效地表示算术模 $2^b - 1$，这是由 Rader[144] 提出的，用于构造 Mersenne NTT。D1 编码可以有效地表示算术模 2^b+1，而这是由 Leibowitz[145] 提出的，专门用于 Fermat NTT

下面的表再次说明了计算加法的 1C 和 D1 算法。

1C	D1
$s=a+b+c_N$	if((a==0)&&(b==0)) $s=0$ else $s = a + b + \overline{c_N}$

其中 a 和 b 是输入操作数，s 是和，而 c_N 是中间和 $a+b$ 没有经过模简化的进位位。要实现 1C 加法，首先要求出 B 位中间和，然后将 MSB c_N 加到 LSB 的进位位上。在 D1 算法中，在加到 LSB 之前必须将进位翻转。加上模 $2^B \pm 1$ 需要的硬件是两个加法器。第二个加法器应该用半加法器构造，因为在 LSB 中，操作数中除了进位位之外，就是 0。

例 7.2：计算 10+7mod *M*。

十　进　制	1C　$M=15$	D1　$M=17$
7 +10	0111 +1010	00110 +01001
17 校正	10001 +1=0010	01111 +1=1.0000
核对	$17_{10} \bmod 15=2$	$17_{10} \bmod 17=0$

减法是根据加性逆元素(additive inverse)定义的。具体说就是如果 $A+B=0$，就称 $B=-A$ 是 A 的加性逆元素。参照表 2-1 就可以很容易地构造加性逆元素。加性逆元素的产生如下：

1C	D1
$\overline{\alpha}$	if(zf(α)! =1) $\overline{\alpha}$

可以看出，首先必须进行逐位的求补计算。这对 1C 和 D1 中的非零元素的编码而言已经足够。但是对于 D1 中的 0，逐位求补是被禁止的。

例 7.3：2 的逆计算如下：

十　进　制	1C　$M=15$	D1　$M=17$
2 -2	0010 1101	0001 1110

可以用表 2-1 中的数据进行验证。

NTT 的最简单的 α 是 2。首先根据 $M=2^b\pm1$ 选择运算的编码(Mersenne 变换采用 C1，Fermat NTT 采用 D1)。仅有必须的乘法是那些带有 $\alpha^k=2^k$ 的乘法。在第 2 章中给出了这些乘法的实现，是通过二进制(左)旋转 k 位位置得到的。最左边的溢出位，也就是进位 c_N 被复制到 LSB 中。在 D1 编码中(除了 $A=0$ 以外)，必须计算进位位的补码，如下表所示：

1C	D1
shl(X, k, c_N)	if(X! $=0$) shl(X, k, $\overline{c_N}$)

下面的例子就说明了在 NTT 中最经常用到的 $\alpha^k=2^k$ 乘法。

例 7.4：1C 和 D1 编码的 2^k 乘法

下表给出了±2 乘以 2 的乘法，最后的 2 乘以 $8=2^3$ 的乘法给出了 1C 和 D1 编码的模运算的示范。

十　进　制	1C　$M=15$	D1　$M=17$
2×2^1	0010	0001
$=4$	0100	0011
-2×2^1	1101	1110
$=-4$	1011	1100
2×2^3	0010	0001
$=16$	0001	1111

可以用表 2-1 中的数据进行验证。

7.1.2　采用 NTT 的高效卷积

从上一节可以看出，α 是 2 的幂时，如果模是 $M=2^b\pm1$，乘法就简化成对数据的移位操作，该移位操作可以用 FPGA 高效快速地构造。很明显，可以将其扩展到复数 α 的形式 $2^u\pm j2^v$。复数 α 的乘法也相应地简化成简单的数据移位操作。

为了避免一般的乘法和一般的模运算，在构造 NTT 的时候需要考虑下面的约束条件：

定理 7.5　在实际中使用 NTT 的约束条件

NTT 只有在满足下面的约束条件时才具有实用价值：

(1) 算法是模 $M=2^b\pm1$。

(2) 所有的乘法 $x[k]\,\alpha^{kn}$ 均可以用 2 的模加法的最大值实现。

7.1.3　采用 NTT 的快速卷积

两个序列 x 和 h 的快速循环卷积可以通过将两个转置的序列相乘来执行【54，131，144】，

如下面的定理所述：

定理 7.6　用 NTT 实现的卷积

令 x 和 h 是模 M 定义的长度为 N 的序列，$z=<x \circledast y>_M$ 是 x 和 y 的循环卷积。令 $X=\mathrm{NTT}(x)$，$Y=\mathrm{NTT}(y)$为 x 和 y 的长度为 N、在 M 上计算的 NTT。则有：

$$z=\mathrm{NTT}^{-1}(X \odot Y) \tag{7.6}$$

要证明这一定理，首先要知道在环模 M 中必须遵循的交换、关联和分配的法则。由于 Z 是一个整数域(整数交换环)，所以显然要遵循这些特性[146, 147]。

具体而言，循环卷积的输出 $y[n]$是：

$$y[n]=\left\langle N^{-1}\sum_{l=0}^{N-1}\left(\sum_{m=0}^{N-1}x[m]\alpha^{ml}\right)\left(\sum_{k=0}^{N-1}h[k]\ \alpha^{kl}\right)\alpha^{-ln}\right\rangle_M \tag{7.7}$$

应用交换、关联和分配的性质，重新整理和以及乘积，就有：

$$y[n]=\left\langle\sum_{k=0}^{N-1}\sum_{m=0}^{N-1}x[m]h[k]\left(N^{-1}\sum_{l=0}^{N-1}\alpha^{(m+k-n)l}\right)\right\rangle_M \tag{7.8}$$

很显然，对于 m,n 和 k 的组合，当$\langle m+n-k\rangle \equiv 0 \bmod N$ 时，即$\sum_{k=0}^{N-1}\alpha^{(m-k-n)l}$等于 N。但当$\langle m+n-k\rangle_N \equiv r \neq 0$ 时，和就是：

$$\sum_{l=0}^{N-1}\alpha^{rl}=1+\alpha^{r}+\alpha^{2r}+\ldots+\alpha^{r(N-1)}=\frac{1-\alpha^{rN}}{1-\alpha^{r}}\equiv 0 \tag{7.9}$$

α 是 N 阶的且 $r<N$，则有 $\alpha^r \neq 1$。方程(7.8)变成：

$$N^{-1}\sum_{l=0}^{N-1}\alpha^{(m+k-n)l}=\begin{cases}\langle NN^{-1}\equiv 1\rangle_M & m+l-n\equiv 0 \bmod N\\ 0 & m+l-n\not\equiv 0 \bmod N\end{cases}$$

现在可以利用 $k\equiv\langle n-m\rangle$ 在和中消去 k，或者用 $m\equiv\langle n-k\rangle$ 消去 m。由第一种情形得到：

$$y[n]=\left\langle\sum_{m=0}^{N-1}x[m]h[\langle n-m\rangle_N]\right\rangle_M \tag{7.10}$$

由第二种情形得到：

$$y[n]=\left\langle\sum_{k=0}^{N-1}h[k]x[\langle n-k\rangle_N]\right\rangle_M \tag{7.11}$$

下面的例题说明了这种卷积。

例 7.7：长度为 4 的 Fermat NTT

利用 Fermat NTT 模 257 计算长度为 4 的时间序列 $x[n]=\{1, 1, 0, 0\}$和 $h[n]=\{1, 0, 0, 1\}$的循环卷积。

解：对于长度为4、模 $M=257$，元素 $\alpha=16$，阶数为4的NTT，且对称范围为[-128,...,128]，则需要 $4^{-1}\equiv -64 \bmod 257$ 和 $16^{-1}\equiv -64 \bmod 257$。变换和逆变换矩阵如下：

$$\boldsymbol{T}=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 16 & -1 & -16\\ 1 & -1 & 1 & -1\\ 1 & -16 & -1 & 16\end{bmatrix} \quad \boldsymbol{T}^{-1}=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -16 & -1 & 16\\ 1 & -1 & 1 & -1\\ 1 & 16 & -1 & -16\end{bmatrix}$$

$x[n]$和 $h[n]$的变换是变换的序列按照元素乘以元素实现的。用INTT得到 $y[n]$的结果如下：

$$\begin{aligned}
n,k &= \{0, \quad 1, \quad 2, \quad 3\}\\
x[n] &= \{1, \quad 1, \quad 0, \quad 0\}\\
X[k] &= \{2, \quad 17, \quad 0, \quad -15\}\\
h[n] &= \{1, \quad 0, \quad 0, \quad 1\}\\
H[k] &= \{2, \quad -15, \quad 0, \quad 17\}\\
X[k]\cdot H[k] &= \{4 \quad 2, \quad 0 \quad 2\}\\
y[n]=x[n]\circledast h[n] &= \{2 \quad 1 \quad 0 \quad 1\}
\end{aligned}$$

NTT的字长限制 用NTT计算卷积时需要记住，输出序列 $y[n]$的所有元素都必须在 M 范围之内(为简便起见，假定是无符号编码)如果：

$$x_{\max}h_{\max}L \le M \tag{7.13}$$

那么结果就为真，如果所采用的位宽 $B_x=\log_2(x_{\max})$、$B_h=\log_2(h_{\max})$、$B_L=\log_2(L)$和 $B_M=\log_2(M)$，就有 $B_x=B_h$，输入的最大位宽范围是：

$$B_x=\frac{B_M-B_L}{2} \tag{7.14}$$

另外的约束就是 $M=2^b\pm1$ 和 α 是2的幂。仅有极少数质数 M 变换存在。表7-1给出了 α 的大部分有用的选择以及Mersenne和Fermat NTT的变换长度(也就是 α 的阶数)。

表7-1 含有复数变换的质数 $M=2^b\pm1$ NTT

Mersenne $M=2^b-1$		Fermat $M=2^b+1$	
α	$\mathrm{ord}_M(\alpha)$	α	$\mathrm{ord}_M(\alpha)$
2	b	2	b
-2	$2b$	$\sqrt{2}$	$2b$
$\pm 2j$	$4b$	$1+j$	$4b$
$1\pm j$	$8b$		

如果将复数变换和非质数 M 也考虑进来，变换的数量和长度就增加了，复杂性也就随之增加了。一般而言，对于非质数模，定理7.1的情况需要检验。还可以利用下面的同余式：

$$a \bmod u \equiv (a \bmod (u\cdot v)) \bmod u \tag{7.15}$$

来使用 Mersenne 或 Fermat NTT 算法。这一公式规定了每次都要首先计算 $M = u \cdot v = 2^b \pm 1$，且只有输出序列需要计算模 u，它是与定理 7.1 有关的有效模。尽管 $M = u \cdot v = 2^b \pm 1$ 增加了内部位宽，但还是可以使用 1C 或 D1 算法。因为它们的复杂性比模算法模 u 更低，一般情况下，这样做会降低整个系统的工作量。

这样的非质数 NTT 就称作伪变换，也称作伪 Mersenne 变换或伪 Fermat 变换。下面的例题说明了伪 Fermat 变换的构造。

例 7.8：长度为 50 的 Fermat NTT

利用 MATLAB 实用程序 order.m(请参阅练习 7.1)，可以确定 $\alpha=2$ 是 50 阶的模 $2^{25}+1$ 的。根据定理 7.1，可以知道 $\gcd(\alpha^2 - 1, M) = 3$，长度为 50 的变换不存在模 $2^{25}+1$。所以必须确定 $M=(2^b \pm 1)$中不具有 50 阶的“坏”因子，并且用(7.15)式中的最终模算法来剔除这些因子。

解：利用标准 MATLAB 函数 factor(2^25+1)，M 的质因数是：

$$2^{25} + 1 = 3 \times 11 \times 251 \times 4051 \tag{7.16}$$

单个因子的 $\alpha=2$ 的阶数可以用练习 7.1 中给出的算法计算，分别是：

$$\mathrm{ord}_3(2)=2 \qquad \mathrm{ord}_{11}(2)=10 \qquad \mathrm{ord}_{251}(2)=50 \qquad \mathrm{ord}_{4051}(2)=50 \tag{7.17}$$

要得到长度为 50 的 NTT，就必须计算$(2^{25}+1)/33$ 的最终模化简。

比较 Fermat 和 Mersenne NTT 的实现，考虑：

- 有 b 个质数且长度为 b 的 Mersenne NTT，可以通过 CZT 变换或 Rader 质数因子定理(prime factor theorem, PFT)[130]转换成循环卷积，如图 7-1(a)所示。此外，如果用一个多 FPGA 实现[135]，还可以使用简化的总线结构。
- $M = 2^{(2^r)} + 1$的 Fermat NTT 具有 2 的幂的长度 $N=2^r$，可以用第 6 章讨论过的普通 Cooley-Tukey radix-2 类型 FFT 算法实现。

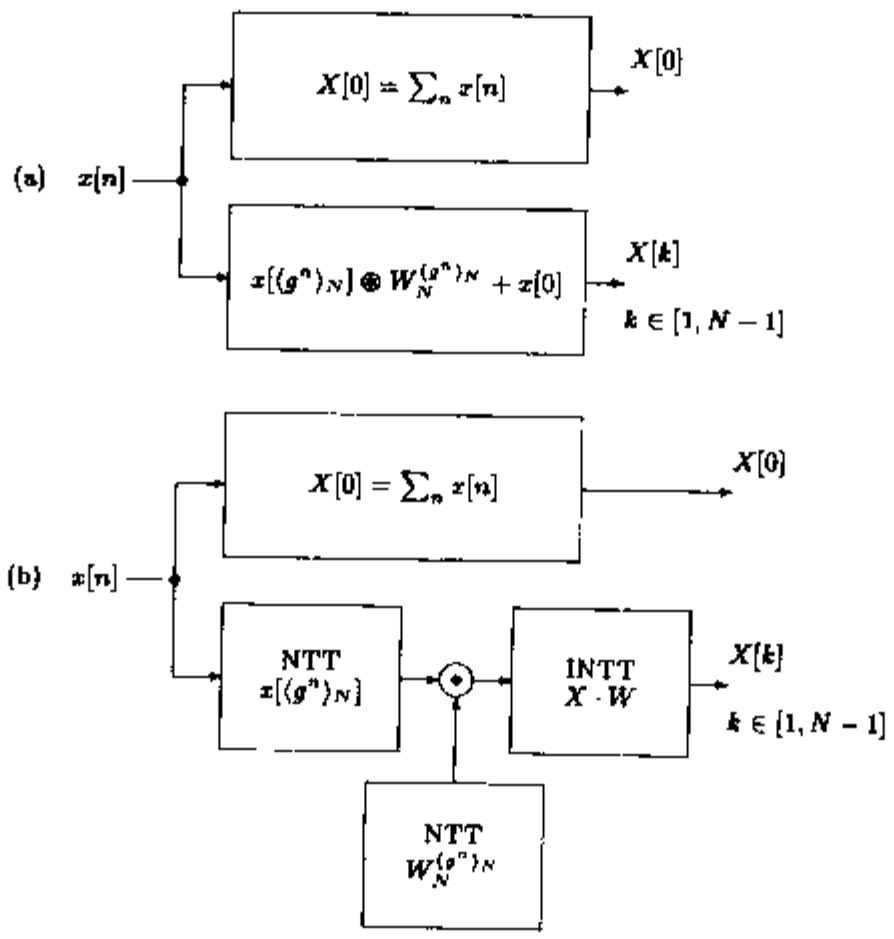


图 7-1　NTT 在计算 DFT 的 Rader 质数长度算法中的应用

(a) Rader 的初始算法 (b) 采用 NTT 的 Rader 质数算法的改进

7.1.4 NTT的多维索引映射和Agarwal-Burrus NTT

对于NTT，一般情况下，变换长度N与位宽是成比例的。这一约束使得构造长的(一维)变换不可能实现，因为需要的位宽将非常巨大。这时候可以尝试多维索引映射，也就是在第6章讨论过的Good-Thomas和Cooley-Tukey算法。如果将这些理论应用到NTT中，就会产生下面的问题：

- 在长度$N=N_1N_2$的Cooley-Tukey算法中，在旋转因子中需要一个N阶元素。这样，与一维情形相比较，虽然变换长度没有增加，但也会产生比较大的位宽，所以没有什么吸引力。
- 如果采用Good-Thomas映射，不需要长度为N的元素和长度为$N=N_1N_2$的变换。但是对于同一M还需要两个互质数长度变换N_1和N_2。如果采用表7-1中列出的变换，对NTT也是可行的。使得Good-Thomas NTT运作的惟一方法就是采用(发表在【135】中的)不同的扩展域，或是将其与Winograd短卷积算法组合使用，但是这会增加实现的复杂性。

Agarwal和Burrus[148]提出的一种理论看起来更有吸引力。在Agarwal-Burrus算法中，首先也是将一维阵列映射到二维阵列中，但是与Good-Thomas理论相反的是，长度为N_1和N_2一定不能是互质数。Agarwal-Burrus算法可以被理解成重叠存储理论的一个派生，利用该方法可以构造信号的周期延拓。如果采用$2L$阶的α，就可以构造一个：

$$N = 2L^2 \tag{7.18}$$

规模的卷积。从表7-2可以看出二维理论提高了变换的最大长度。

要计算Agarwal-Burrus-NTT，需要经过下面的5个步骤：

表7-2 Agarwal-Burrus - NTT的一些数据，用实Fermat NTT($b=2^t$, t=0至4)或伪Fermat NTT t=5,6计算循环卷积

模	α	1D	2D
2^b+1	2	$2b$	$2b^2$
2^b+1	$\sqrt{2}$	$4b$	$8b^2$

算法7.9： Agarwal-Burrus NTT

长度为$N=2L^2$的$x \odot h$与长度为L的NTT的循环卷积要经过下面的步骤实现：

(1) 根据

$$x=\begin{bmatrix} x[0] & x[L] & \cdots & x[N-1] \\ x[1] & x[L+1] & \cdots & x[N-L+1] \\ \vdots & \vdots & \vdots & \vdots \\ x[L-1] & x[2L-1] & \cdots & x[N-1] \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.19}$$

$$h=\begin{bmatrix} h[N-L+1] & h[L] & \cdots & h[N-2L+1] \\ \vdots & \vdots & \vdots & \vdots \\ h[N-1] & h[L-1] & \cdots & h[N-L-1] \\ h[0] & h[L] & \cdots & h[N-L] \\ h[1] & h[L+1] & \cdots & h[N-L+1] \\ \vdots & \vdots & \vdots & \vdots \\ h[L-1] & h[2L-1] & \cdots & h[N-1] \end{bmatrix} \tag{7.20}$$

进行一维序列到二维序列的索引变换。

(2) 在计算行变换 $\begin{bmatrix}\rightarrow \\ \rightarrow \\ \vdots\end{bmatrix}$ 后计算列变换 $\begin{bmatrix}\downarrow\downarrow\cdots\end{bmatrix}$。

(3) 计算逐个元素的矩阵乘法 $Y = H \odot X$。

(4) 在列 $\begin{bmatrix}\downarrow\downarrow\cdots\end{bmatrix}$ 的逆变换后进行行 $\begin{bmatrix}\rightarrow \\ \rightarrow \\ \vdots\end{bmatrix}$ 的逆变换。

(5) 根据

$$y=\begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ y[0] & y[L] & \cdots & y[N-L] \\ y[1] & y[L+1] & \cdots & y[N-L+1] \\ \vdots & \vdots & \vdots & \vdots \\ y[L-1] & y[2L-1] & \cdots & y[N-1] \end{bmatrix} \tag{7.21}$$

对 y 的低端输出序列进行重构。

可以用下面的例子说明 Agarwal-Burrus NTT。

例 7.10：长度为 8 的 Agarwal-Burrus-NTT

α=16 长度为 4 的 NTT 模 257 存在。用 Fermat NTT 模 257 计算 $X(z)=1+z^{-1}+z^{-2}+z^{-3}$ 与 $F(z)=1+2z^{-1}+3z^{-2}+4z^{-3}$ 的卷积。

解：首先计算 $x[n]$和 $f[n]$的索引映射和变换：

$$x=\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \leftrightarrow X=\begin{bmatrix} 4 & 34 & 0 & 227 \\ 34 & 32 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 227 & 2 & 0 & 225 \end{bmatrix} \tag{7.22}$$

$$f=\begin{bmatrix}0&0&0&0\\0&2&4&0\\1&3&0&0\\2&4&0&0\end{bmatrix}\leftrightarrow F=\begin{bmatrix}16&143&255&112\\253&114&66&206\\249&212&255&51\\253&45&195&145\end{bmatrix} \tag{7.23}$$

现在计算逐个元素的乘法，结果是：

$$Y=\begin{bmatrix}64&236&0&238\\121&50&0&155\\0&0&0&0\\120&90&0&243\end{bmatrix}\leftrightarrow y=\begin{bmatrix}2&6&4&0\\0&2&6&4\\1&6&9&4\\3&10&7&0\end{bmatrix} \tag{7.24}$$

从 y 的下半部分可以看到 $y[n]=\{1,3,6,10,9,7,4,0\}$ 的元素。

采用 Agarwal-Burrus-NTT，就需要提供两倍规模的中间存储器，但是可以计算更长的变换。二维索引变换的原理可以很容易扩展到三维索引变换，但是在大多数情况下，用二维理论达到的变换长度就足够了。例如：$\alpha=2$ 和 $b=32$，变换长度就从一维下的 64 增加到二维下的 $2^{11}=2048$。

7.1.5 用 NTT 计算 DFT 矩阵

大多数 DFT 和 NTT 都可以用来计算卷积，而且由于 IC 和 DI 可以有效地实现，所以采用 NTT 用 FPGA 计算卷积非常具有吸引力。但通常还是有必要计算 DFT 来估算傅立叶频谱。这样，问题就出现了：有没有可能用更加有效的 NTT 来计算 DFT？这个问题已经由 Siu 和 Constantinides 作出了详细的回答【149】。

其思路如下：对于长度为质数 p 的 DFT，可以采用 Rader 算法，该算法将任务转换成长度为 $p-1$ 的循环卷积。这种循环卷积是通过初始序列的 NTT 和 NTT 域、乘法逐位方式和向后变换中的 DFT 旋转因子计算的。图 7-1(b)给出了这些过程的步骤。在下面的例子中将说明这一原理。

例 7.11：$N=5$ 的 Rader 算法

$N=5$，生成元 $g=2$，给出如下的索引映射，$\{2^0, 2^1, 2^2, 2^3\} \bmod 5 \equiv \{1, 2, 4, 3\}$。首先，用：

$$X[0]=\sum_{n=0}^{4}x[n]=x[0]+x[1]+x[2]+x[3]+x[4]$$

计算 DC 部分，在第二步 $X[k]-x[0]$ 中，循环卷积：

$$\{x[1], x[2], x[4], x[3]\}\circledast\{W_5^1, W_5^2, W_5^4, W_5^3\}$$

现在对(重新排序的)序列 $x[n]$ 和 W_5^k 进行 NTT，如例 7.7 所示。然后在 NTT 域内对变换后的序列逐位相乘，最后计算 INTT。

Mersenne NTT 会产生一个问题，就是 NTT 本身的长度是质数，而长度增加 1 后就不再是质数了。但对于 Fermat NTT，长度是 2^t，而 $M=2^t+1$ 是一个质数。Siu 等人发现 8 个这样短长度的 DFT 构造模块是有用的。表 7-3 总结了这些基本构造模块。

表 7-3　用 Fermat NTT 计算 DFT 的构造模块

DFT 长度	数　字　环	α	实数乘法器的数量	实数移位加法器的数量
3	$F_1, F_2, F_3, F_4, F_5, F_6$	$2^2, 2^4, 2^8, 2^{16}, 2^{32}, 2^{64}$	2	6
5	$F_1, F_2, F_3, F_4, F_5, F_6$	$2, 2^2, 2^4, 2^8, 2^{16}, 2^{32}$	4	20
17	F_3, F_4, F_5, F_6	$2, 2^2, 2^4, 2^8$	16	144
257	F_6	$\sqrt{2}$	256	4544
13	$F_1, F_2, F_3, F_4, F_5, F_6$	$2, 2^2, 2^4, 2^8, 2^{16}, 2^{32}$	16	104
97	F_4, F_5, F_6	$2, 2^2, 2^3, 2^4$	128	1408
193	F_5, F_6	$2, 2^2$	256	3200
769	F_6	$\sqrt{2}$	1024	16448

表 7-3 的第一部分给出的是不需要索引映射的模块。第二部分是具有两个互质数因子的构造模块，分别是 13 - 1＝3×4，97 - 1＝3×32，193 - 1＝3×64 和 769 - 1＝3×256。两个因子的缺点是，在二维索引映射中，旋转因子的每两个变换只有一维可以变成 0。

在多维映射中，也可以实现类似 radix-2 FFT 的算法，或者是将 Fermat NTT 与其他 NTT 算法组合，例如：(伪)Fermat NTT 变换、(伪)Mersenne 变换、Lagrange 插值、Eisenstein NTT 或短卷积算法，如 Winograd 算法【149，54】。

接下来要复习一下使用长度 13 - 1＝3×4 多维索引映射的旋转因子的技术。这与第 6 章讨论过的 FFT 类似。

7.1.6　NTT 的索引映射

一般直接实现 NTT 矩阵其代价是非常高的。这个问题可以通过合适的多维技术加以解决。Burrus【109】给出了从一维到多维的不同公共因子和质数因子映射的系统综述，映射是以二维为例解释的，更高维的映射也是相同的。从(7.1)的一维循环长度为 N 的卷积到 $N=N_1N_2$ 的二维卷积的映射，以线性形式写出来就是：

$$n = M_1 n_1 + M_2 n_2 \bmod N \tag{7.25}$$

其中 $n_1 \in \{0,1,2,\ldots, N_1 - 1\}$ 和 $n_2 \in \{0,1,2,\ldots, N_2 - 1\}$。由于 $\gcd(N_1, N_2) \neq 1$，所以就可以使用著名的 Cooley-Tukey FFT 算法。Burrus[109]指出，当且仅当 N_1 和 N_2 是互质数(也就是 $\gcd(N_1, N_2) = 1$)时，在一维和二维内的映射均是循环的。要保证这一映射是一对一且是映成的(也就是双向映射)，映射常数 M_1 和 M_2 必须满足某些条件。在互为质数的前提下，使得映射为双向映射的条件是：

$$\begin{aligned} &M_1 = \beta N_1 \text{和/或} M_2 = \gamma N_1 \text{和} \\ &\gcd(M_1, N_1) = \gcd(M_2, N_2) = 1 \end{aligned} \tag{7.26}$$

举例说明，考虑 $N_1=3$ 和 $N_2=4$，$N=12$。根据(7.26)式可以知道，应该选择 M_1(N_2 的一个倍数)或 M_2(N_1 的一个倍数)，或两者都选。令 M_1 是 N_2 的最简单的倍数，也就是 $M_1=N_2=4$，也满足 $\gcd(M_1, N_1)=\gcd(4, 3)=1$。接下来可以看到 $\gcd(M_2, N_2)=\gcd(M_2, 4)=1$，$M_2$ 的可能值是 {1, 3, 5, 7, 9, 11}。做一个简单的选择，选 $M_2=N_1=3$。映射变成 $n=\langle 4n_1+3n_2\rangle_{12}$。现在用这一映

射来研究 12 点卷积的例子。一维循环阵列 $x[n]$到 3×4 二维阵列 $x[n_1,n_2]$的变换生成：

$$[x[0]x[1]x[2]\ldots x[11]]\leftrightarrow\begin{bmatrix}x[0] & x[3] & x[6] & x[9]\\ x[4] & x[7] & x[10] & x[1]\\ x[8] & x[11] & x[2] & x[5]\end{bmatrix} \tag{7.27}$$

要从 $X[k_1,k_2]$中恢复序列 $X[k]$，需要用到 Good[132] 提出的中国余数定理：

$$k=\langle(N_2^{-1}\bmod N_1)N_2k_1+(N_1^{-1}\bmod N_2)N_1k_2\rangle_N \tag{7.28}$$

α 矩阵可以改写成

$$X[k_1,k_2]=\sum_{n_1=0}^{N_1-1}\left(\sum_{n_2=0}^{N_2-1}x[n_1,n_2]\alpha_{N_2}^{n_2k_2}\right)\alpha_{N_1}^{n_1k_1} \tag{7.29}$$

其中 α_{N_i} 是第 N_i 阶的一个元素。将初始序列 $x[n]$映射到二维阵列 $x[n_1,n_2]$后，就可以用下面的两个步骤来计算所需要的矩阵：

(1) 在矩阵 $x[n_1,n_2]$的每一行执行一次 N_2 点 NTT。

(2) 在所得矩阵的每一列执行一次 N_1 点 NTT，得到 $X[k_1,k_2]$。

这些处理步骤如图 7-2 所示。输入映射由(7.27)给出，而输出映射可以根据(7.28)计算，

$$k=\langle\langle4^{-1}\rangle_3 4k_1+\langle3^{-1}\rangle_4 3k_2\rangle_{12}=\langle4k_1+9k_2\rangle_{12} \tag{7.30}$$

矩阵 $X[k_1,k_2]$可以如下面所示进行重新组织：

$$[X[0]X[1]X[2]\ldots X[11]]\leftrightarrow\begin{bmatrix}X[0] & X[9] & X[6] & X[3]\\ X[4] & X[1] & X[10] & X[7]\\ X[8] & X[5] & X[2] & X[11]\end{bmatrix} \tag{7.31}$$

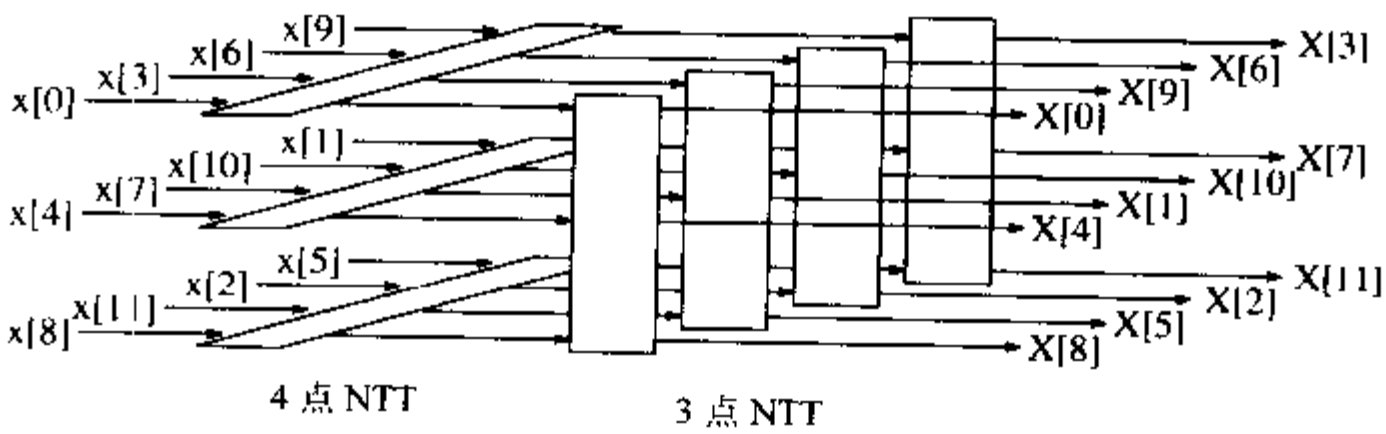


图 7-2　二维映射。第一级：3 个 4 点 NTT。第二级：4 个 3 点 NTT

长度为 97 的 DFT 案例研究　近些年来可编程数字信号处理器(PDSP)(如 TMS320; Motorola 56K; AT&T 32C)已经成为通过 FFT 算法实现快速卷积的主要工具。这些 PDSP 提供了一个典型周期时间值 10~50ns 的快速(实数)乘法器，还有一些 NTT 实现[150]，但需要模算法的 NTT 实现除外，这些需要模算法的 NTT 实现是通用的 PDSP 所不支持的。专用累加器，例如：J.McClellan 的 FNT[150] 就使用了 90 个标准的 ECL10K 集成电路。在近几年里，现场可编程门阵列(FPGA)的集成度足够高，速度也足够快，能够满足实现典型的高速 DSP 应用的要求[4, 119]。用一片 FPGA 实现几种算法核心、生成良好的程序包、速度和功率特性也成为了可能。FPGA

以其精细的颗粒度就能够有效地实现模算法，而不像在PDSP中一样会有性能损失。

在Fermat数制算法的NTT实现中，FPGA的实现与传统的FFT实现相比较，前面讨论过的速度和硬件优势就更加明显了。通过用Rader质数卷积方法实现DFT算法需要的I/O性能会得到进一步简化。

为了阐明NTT设计范例，下面将要给出一个Fermat数制系统中长度为97的DFT，F_4和F_5是为实数输入数据准备的。在本设计中还要用到Xilinx XC4K的多功能FPGA板来实现，正如在【135】中所报告的。

对于模Fermat数制算法(模2^n+1)而言，采用Leibowitz[145]的D1数制系统代替2C算法更能够发挥优势。负数与2C算法中的负数是相同的，正数则需要减1。0被编码成一个0字符串，MSB“ZERO-FLAG(零标志)”就是一个。这样，D1数制就包括一个ZERO-FLAG和整数位x_k。对于2C而言，MSB就是符号位，而对于D1而言，第二个MSB是符号位。这种通过2^m的2C↔D1的转换、求非、加法和乘法简单操作的编码是很容易计算的，如7.1.1节所示。

图7-1(b)给出了97点变换的简略处理步骤。单NTT的直接长度为96的实现需要至少96×2个筒状移位器和96×2个累加器以及大约96×(2×32+2×18)=9600个Xilinx组合逻辑模块(combinatorial logic blocks, CLB)。所以像上一节讨论的采用32×3索引映射应该是合理的。长度为32的FFT现在就变成了一个简单的长度为32的Fermat NTT，而长度为3的变换有$\alpha^k=1$；j和$-1-j$以及$j^2=j+1$。32点FNT可以用普通的radix-2 FFT类型的算法实现，而长度为3的变换可以用一个二抽头FIR滤波器实现。下面的表为F_4给出了Xilinx XC4000 FPGA的CLB利用率的估算：

长度为32的FNT	长度为3的FIR NTT	14个乘法器32位	长度为3的NTTS^{-1}	两个长度为32的FNT^{-1}
104	108	462	288	216

该设计共用掉了1178个CLB。为了获得更高的吞吐量，模块之间的缓冲存储器必须加倍。第一个FNT需要两个实数缓冲器和3个复数缓冲器。如果要在内部实现这些缓冲器，还需要额外的748个CLB，这样也可以使对I/O的要求达到最小化。假定利用率是80%，包括缓冲存储器在内，该设计就需要大约6个XC4010。

在该设计中，决定时间路径的就是长度为32的FNT。为了使吞吐量最大，在蝶形内部采用了三级流水线结构。对于5ns的FPGA，F_4的蝶形速度是28ns，F_5的是38.5ns。长度为32的FNT有5个级，每个级都有16个蝶形需要计算。这样长度为97的DFT总变换时间分别是：F_4为7.15μs，F_5为9.24μs。为了正确地设置这一结果，蝶形的计算时间给出了一个公平的结果。假定在0等待状态存储器情况下，50ns周期时间的TM320C50 PDSP需要17个周期，或是850ns才能完成蝶形的计算[151]。另一种“传统的”FPGA定点数算法设计[119]采用了4个串行/并行乘法器(2MHz)，因此，处理蝶形的时间是500ns。

7.1.7 用矩形变换计算DFT

矩形变换也可以将输入序列映射到图像域中，但是不一定具有DFT结构，也就是$A=[\alpha^{nk}]$。比如Haar变换[91]、Walsh/Hadamard变换[59]、轴环量化(ruff-quantized)DFT[5]、或算法傅立

叶变换[152, 153, 134]。通常这些矩形变换均不支持循环卷积，但是可以用来近似 DFT 谱[126]。矩形变换的优点在于系数均取自集合{－1，0，1}，不需要任何乘法。

图 7-3 给出了应该如何计算 DFT。为了得到一个有用的系统，假定计算矩形变换的工作量比较少，第二次变换采用了仅有极少非零元素的矩阵 $\boldsymbol{T}$，该矩阵将矩形变换映射到 DFT 向量。

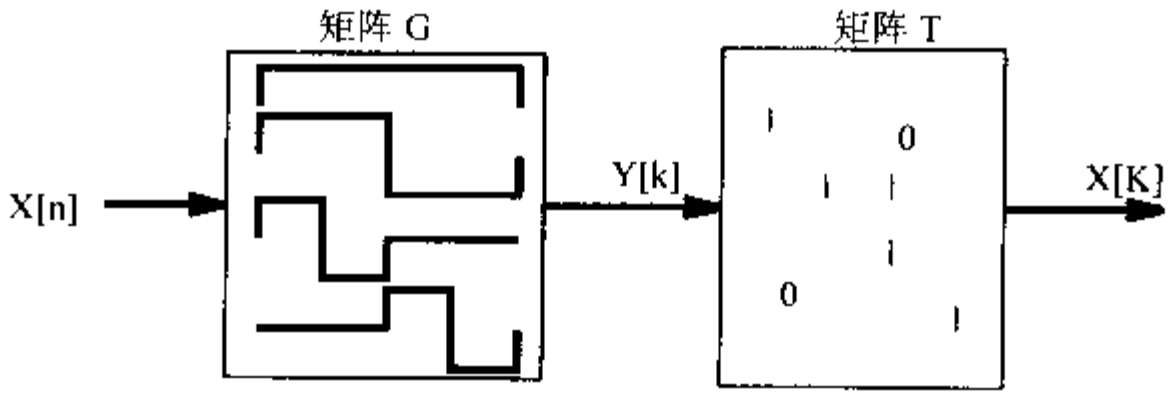


图 7-3　用矩形变换和映射矩阵 $\boldsymbol{T}$ 计算 DFT

表 7-4 比较了不同的实现。Walsh/Hadamard 和 Haar 变换的算法复杂性是最有趣的，但是从第二次变换 $\boldsymbol{T}$ 中零的数量可以得出结论：算法傅立叶变换和轴环量化 DFT 对于近似 DFT 更具吸引力。

表 7-4　近似 DFT 的不同变换的比较[5]

变　换	基 的 数 量	算法的复杂性	16×16T 矩阵中的 0
Walsh	N	$N\log_2(N)$	66
Hadamard	N	$N\log_2(N)$	66
Haar	N	$2N$	18
AFT	$N+1$	N^2	82
QDFT	$2N$	$(N/8)^2+3N$	86

7.2　差错控制和加密技术

现代通信系统，如寻呼机、移动电话或卫星传输系统，均需要使用算法校正传输差错。因为差错校正编码能够比专用的调制方案更好地利用受频带限制的信道容量(参阅图 7-4)。此外大多数系统还会用到加密技术算法，不仅仅是要保护信息防止未授权的侦听，还要保护信息防止未授权的改动。

如图 7-5 所示，在典型的传输图中，编码器(用于差错校正或加密技术)放置在数据源和实际的调制之间。在接收器端，译码器位于实际的解调和数据接收站(接收器)之间。通常，编码器和译码器是组合在一个电路中的，称作 CODEC(coder/decoder，编码/译码器)。

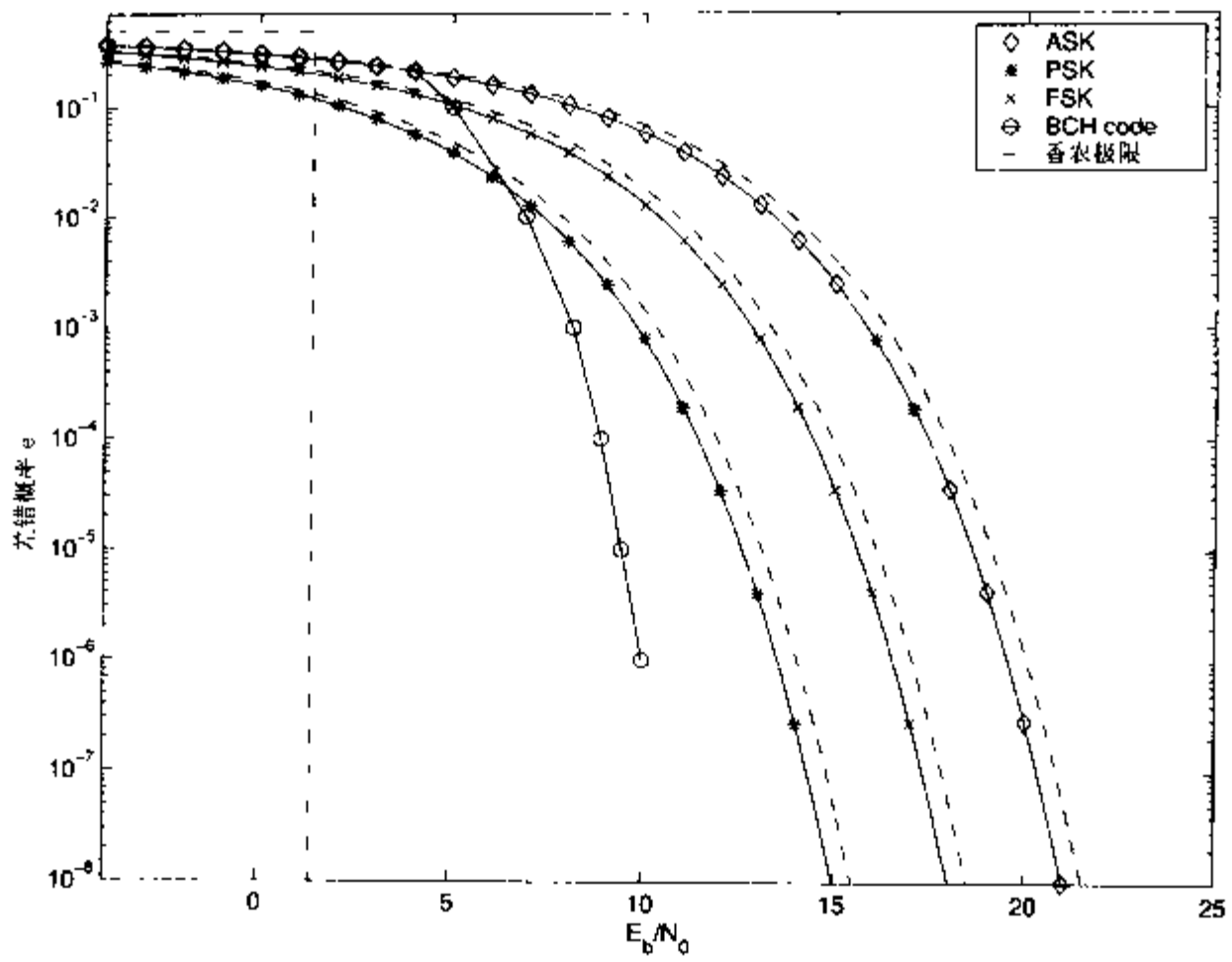


图 7-4　调制图的性能

图 7-5　典型的通信系统结构

典型的差错校正和加密技术算法均使用有限域算法，因而相对于 PDSP 而言，它们对于 FPGA 则更为适合[155]。使用 FPGA，可以非常有效地实现逐位运算或线性反馈移位寄存器(linear feedback shift register, LFSR)。一些 CODEC 图需要一些大规模的表，而且在为 FPGA 选择适合的算法时，也就会发现哪些算法是最适合的。在这一节中给出的算法主要来自于以前的出版物【4】，且这些算法目前已经用于开发低频寻呼系统[156, 157, 158, 159, 160]，和某种用于无线电控制监视器的[161, 162]差错校正图。

在一个很短的章节里讲解整个差错校正和加密技术是不可能的。我们会为进一步研究给出基本的理论和建议，以及在这一领域中非常优秀的教科书[124, 163, 92, 164, 165, 166, 167]。

7.2.1　源自编码理论的基本概念

保护数据传输防止随机错误的最简单方法就是将信息重复发送几次。称作重复码(repetition code)。例如 5 次重复，就是将信息发送 5 次，也就是：

$$0 \Leftrightarrow 00000 \tag{7.32}$$

$$1 \Leftrightarrow 11111 \tag{7.33}$$

其中左边是第 k 个信息位，右边是 n 位代码字。两个代码字之间的最短距离就是 n，也称作**海明距离**(Hamming distance)d^*，重复码的形式是$(n, k, d^*) = (5,1,5)$。这样的代码可以校正 $\lfloor (n-1)/2 \rfloor$次随机错误。但是从信道的效率方面来讲，这种代码是没有吸引力的。如果系统是双

路的，用奇偶校验和自动重复请求(automatic repeat request， ARQ)等技术检验任意奇偶差错则更有效。例如：在 PC 机内存中就应用了这种奇偶校验。

1. 用海明码进行差错控制

如果有几个奇偶校验位相加，就有可能用奇偶校验误差校正一个字。

如果奇偶校验 $P_{1,0}$，$P_{1,1}$，$P_{1,2}$和 $P_{1,3}$，采用的是模 2 运算，也就是 XOR，根据：

$$P_{1,0}=i_{21}\oplus i_{22}\oplus i_{23}\oplus i_{24}\oplus i_{25}\oplus i_{26}\oplus i_{27}$$

$$P_{1,1}=i_{21}\oplus i_{23}\oplus i_{25}\oplus i_{27}$$

$$P_{1,2}=i_{21}\oplus i_{22}\oplus i_{25}\oplus i_{26}$$

$$P_{1,3}=i_{21}\oplus i_{22}\oplus i_{23}\oplus i_{24}$$

奇偶校验器是 $i'_{28}(=P_{1,0})$，还需要 3 个放置差错出现位置的额外位。图 7-6(a)是编码器，而图 7-6(b)是包括差错校正逻辑的译码器。在译码器端引入的奇偶位是 XOR 最新计算的奇偶位。这种形式就是所谓的**伴随式**(Syndrome, $S_{1,0},\ldots,S_{1,3}$)。奇偶位的选择方式应该保证伴随式结构与二进制码中位的位置一致。也就是一个 3 → 7 分离乘法器可以用来译码差错位置。

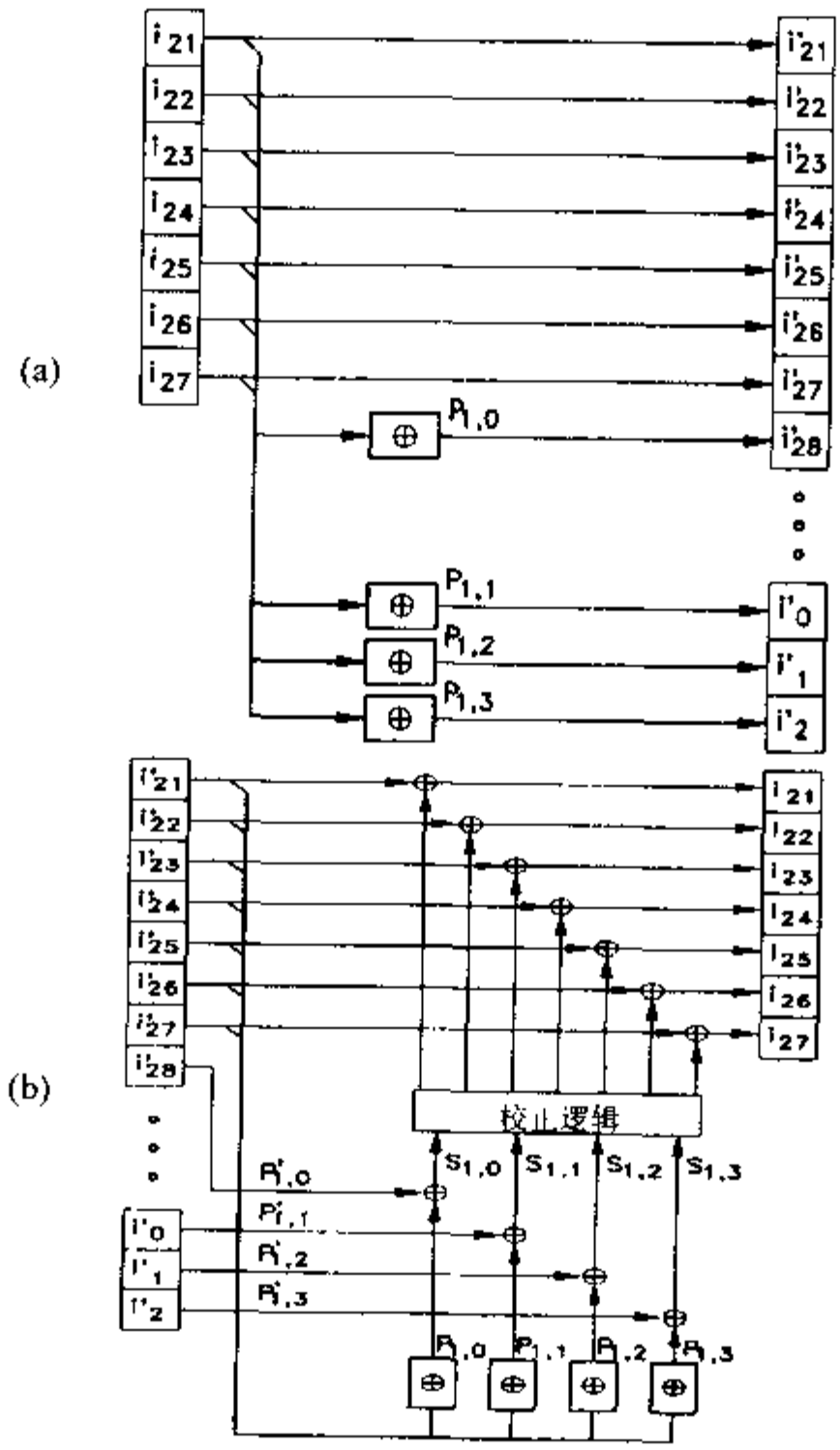


图 7-6 海明码的 (a) 编码器 (b) 译码器

如果要用更为简洁的表达式表示译码器，就可以采用下面的奇偶校验矩阵 $\boldsymbol{H}$：

$$\boldsymbol{H}=[\boldsymbol{P}^T:\boldsymbol{I}]=\begin{bmatrix}1&1&1&1&1&1&1&1&0&0&0\\1&0&1&0&1&0&1&0&1&0&0\\1&1&0&0&1&1&0&0&0&1&0\\1&1&1&1&0&0&0&0&0&0&1\end{bmatrix} \tag{7.34}$$

还可以用一个生成矩阵来描述编码器。$\boldsymbol{G}=[\boldsymbol{I}:\boldsymbol{P}]$，也就是说生成矩阵是由一个系统恒定的矩阵 $\boldsymbol{I}$ 和一个奇偶位矩阵 $\boldsymbol{P}$ 组成。代码字 $\boldsymbol{v}$ 由信息字 $\boldsymbol{i}$ 与生成矩阵 $\boldsymbol{G}$ 相乘 (模 2) 得到：

$$\boldsymbol{v}=\boldsymbol{i}\cdot\boldsymbol{G} \tag{7.35}$$

图 7-6 中给出的编码器是为(11,7,3)海明码提供的，它们能够检测并且校正一个差错。通常，可以看出对于 4 个奇偶位，最多可以使用 15 个信息位，也就是 (15,11,3) 海明码可以缩短成(11,7,3)码。

距离为 3 的海明码一般具有$(2^m-1, 2^m-m, 3)$的结构。例如：无线电控制监视器的天数就被编码成 22 位，而且(31, 26, 3)海明码可以缩短成(27, 22, 3)码，借此以获得单个差错校正码。奇偶校验矩阵又变成：

$$\boldsymbol{H}=\begin{bmatrix}1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&0&0&0\\1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&1&0&0&0\\1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0&1&0&0\\1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&1&0\\1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&1\end{bmatrix}$$

伴随式可以按照如下方式分类：校正逻辑是一个简单的 5→22 分离乘法器。

表 7-5 给出了采用 Xilinx XC3K FPGA 的无线电控制监视器差错校正单元以 CLB 计算的预计工作量，该单元由 3 个分别代表分钟、小时和天数的独立数据块组成。

表 7-5　海明码差错校正的预计工作量

分组海明码	CLB 的工作量		
	分钟(11,7,3)	小时(10,6,3)	天数(27,22,3)
寄存器	6	6	14
伴随式的计算	5	5	16
校正逻辑	4	4	22
输出寄存器	4	4	11
和	19	19	63
总计	101		

总之，对于分钟，使用额外的 3+3+5＝11 位和奇偶位(大约是 100 个 CLB)，就能够校正这三个数据块中的一个数据块中的一个差错。

2. 差错校正码的综述

在上一节介绍几个例子之后，接下来要讨论通用的代码和可行的编码器和译码器的实现。在大多数情况下，人们关注的更多的是译码器的工作量，因为像寻呼机或无线电通信等很多通信系统均具有一个发射器和多个接收器。图 7-7 给出了可行的译码器的框图。

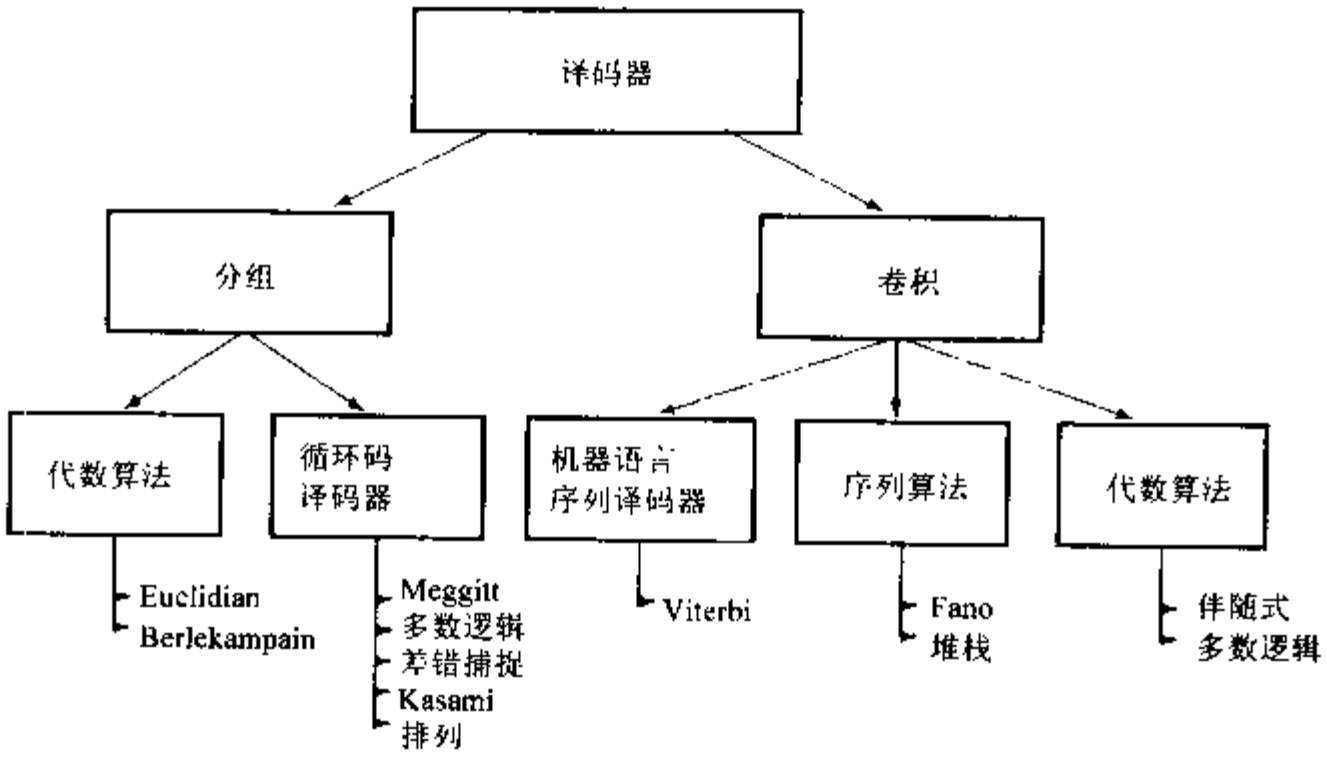


图 7-7　用于差错校正的译码器

一些接近最优的译码器都采用了大规模的表，在图 7-7 中没有包括这些种类。分组码和卷积码之间的区别就在于在代码产生中是否使用了“存储器”。两种方法都是以编码率 R 为特征的，R 是信息位与代码长度的商，也就是 $R=k/n$。对于采用存储器的树形码而言，实际输出的 n 位长度分组码不仅仅与当前的 k 个信息位有关，而且还依赖于前面 m 个符号位，如图 7-8 所示。卷积码的特征是存储长度 $v=m \cdot k$，距离轮廓、间隙 d_f 和最小距离 d_m 也是其特征(参阅【124】)。分组码通常用代数方法构造，运用的是有限域，而树形码一般只出现在计算机仿真中。

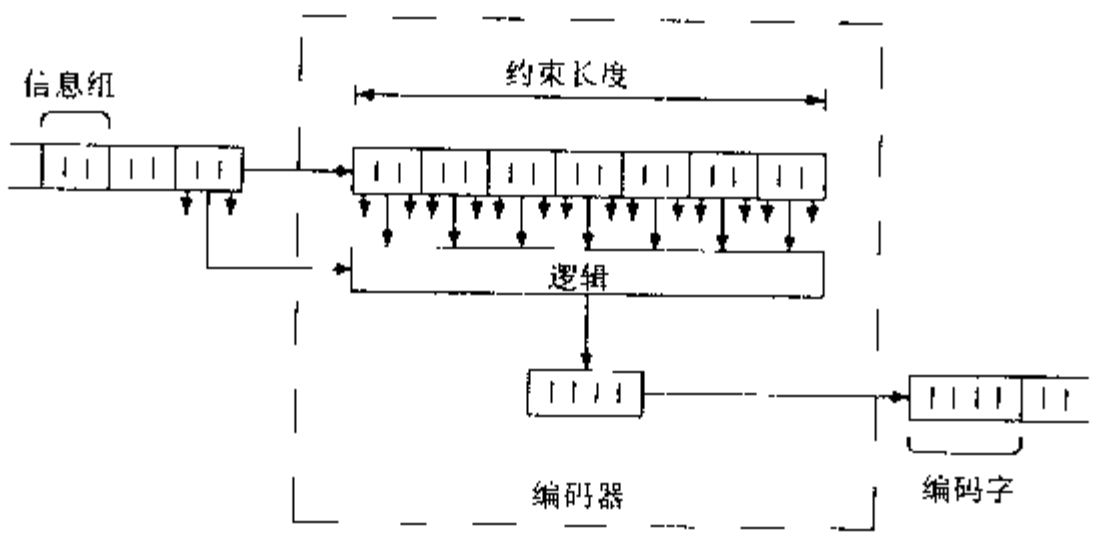


图 7-8　卷积编码器的参数

我们的讨论目前仅局限于线性码，也就是两个代码字的和仍然是一个代码字，因为这样就简化了译码器的实现。对于线性码，海明距离总可以按代码字和零字之间的差来计算。这就简化了代码性能的比较。线性树形码通常称为卷积码，这是因为可以用类似 FIR 的结构构造代码。卷积码可能是灾难性的，也可能是非灾难性的。如果是灾难性的代码，单个差错会被一直传播

下去。由此可以明白，系统的卷积码总是非灾难性的。此外，区别**随机差错**校正代码和**突发差错**校正代码也是经常事。在突发差错校正中，有可能是一长串突发差错(或是删除)。在随机差错校正代码中，校正差错的能力也不局限于在连续的位上——差错有可能出现在接收的代码字的随机位置上。

3. 编码边界

有了编码边界，我们就可以比较不同的编码方案。边界给出了代码的最大差错校正能力。译码器永远不可能超过编码的上限，通常，为了降低译码器的复杂程度，也需要译码低于理论边界。

一个简单但很好的粗略估算就是 Singleton 边界或海明边界。**Singleton 边界**规定：最小海明距离 d^*受奇偶位的数量$(n-k)$上限的限制。还可以知道的是：代码可以校正的差错的数量 t 和删掉的数量 e 均以海明距离为上限。这样就得到下面的约束条件：

$$e+2t+1 \le d^* \le n-k+1 \tag{7.36}$$

$d^*=n-k+1$ 的代码就称为最大距离可分码。但是除了重复码和奇偶校验码之外，均没有二进制最大距离可分码[124]。下面这个例子来自上一节的表 7-5，上限为 11 个奇偶位，最多可以校正 5 个差错。

下面的海明边界给出了一个 t 位差错校正二进制码的估算：

$$2^{n-k} \ge \sum_{m=0}^{t}\binom{n}{m} \tag{7.37}$$

(7.37)式说明了奇偶校验模式的可能数量(2^{n-k})必须要大于或等于差错模式的数量。如果(7.37)式中的等式成立，这样的代码就称为**理想码**或**完备码**(perfect code)。举例说明：上一节讨论的海明码就是一种理想码。如果需要找到一种代码来保护无线电控制监视器在一个分钟内传输的所有 44 个位，其中最多可以使用 13 个奇偶位，则必须有：

$$2^{13} > \binom{44}{0}+\binom{44}{1}+\binom{44}{2} \tag{7.38}$$

但是

$$2^{13} < \binom{44}{0}+\binom{44}{1}+\binom{44}{2}+\binom{44}{3} \tag{7.39}$$

也就是说，可以找到一种具有校正 2 个随机差错的代码，但是找不到能够校正 3 个随机差错的代码。在下一节将要复习一下分组编码器和译码器，然后讨论卷积编码器和译码器。

7.2.2 分组码

线性循环二进制 BCH (Boss，Chaudhuri 和 Hocquenghem)码和 Reed-Solomon 码由一大类分组码构成。BCH 码在时域和频域内具有多种知名的有效译码器。接下来简要说明一下(63, 50, 6)

到(57, 44, 6)的 BCH 码。算法的详细讨论是由 Blahut 给出的[1124]。

代码是以 GF(2^6)到 GF(2)的变换为基础的。描述 GF(2^6)需要一个 6 次的质数多项式，例如：$P(x)=x^6+x+1$。要计算生成多项式，必须要计算 GF(2^6)中第一个 $d-1=5$ 的最小多项式的最小公因子。如果 α 表示 GF(2^6)内的一个质数元素，则有 $\alpha^0=1$ 和 $m_{1(x)}=x-1$。α，α^2 和 α^4 的最小多项式均是同一个 $m_{\alpha(x)}=x^6+x+1$，α^3 的最小多项式是 $m_{\alpha^3(x)}=x^6+x^4+x^2+x+1$，这样就可以构造生成多项式 $g(x)$:

$$g(x)=m_{1(x)}\cdot m_{\alpha(x)}\cdot m_{\alpha^3(x)} \tag{7.40}$$

$$=x^{13}+x^{12}+x^{11}+x^{10}+x^9+x^8+x^6+x^3+x+1 \tag{7.41}$$

利用这个生成多项式，就可以向前构造编码器和译码器了。

1. 编码器

由于需要的是系统码，所以第一个代码字位应该与信息位相同。要获得系统码，可以根据：

$$p(x)=i(x)\cdot x^{n-k} \bmod g(x) \tag{7.42}$$

通过移位的信息位 $i(x)$的模简化来计算奇偶位 $p(x)$。这样的模简化可以用一个图 7-9 所示的递归移位寄存器实现。

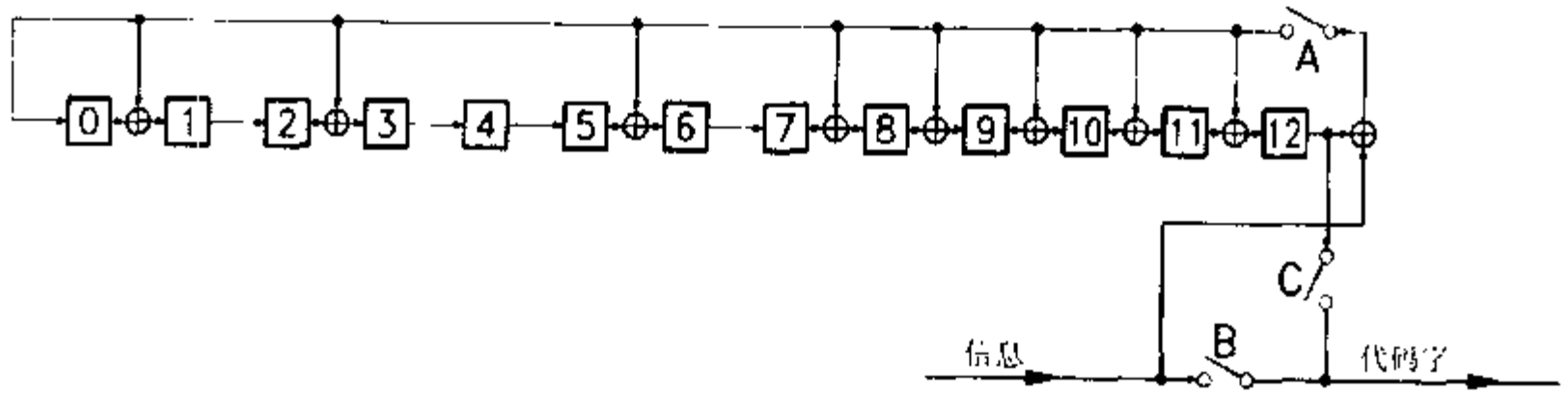


图 7-9 (57, 44, 6)BCH 码的编码器

电路工作流程如下：首先，开关 A 和 B 闭合，C 打开。接下来，接收信息位(首先是 MSB)并且直接变换成代码字。同时递归移位寄存器计算奇偶位。在信息位都被处理完之后，开关 A 和 B 打开，C 闭合。奇偶位就移位到了代码字中。

2. 译码器

通常译码器要比编码器复杂得多。Meggitt 译码器可以用在时域中译码，而且也可以用于频率译码，但是这需要对 BCH 码的代数性质有详尽的了解（【124，92，163】）。FPGA 的这类频率译码器已经作为知识产权(IP)模块了，通常也被称作“虚拟组件(virtual components, VC)”(请参阅【168，18，19】)。

Meggitt 译码器(如图 7-10 所示)对于仅有少数差错需要校验的代码是非常有效的，这是由于该译码器利用了 BCH 码的循环性质。由于只校正最高位位置的差错，然后计算循环移位，所以实际上所有被破坏的位都要通过 MSB 位置，并且得到校正。

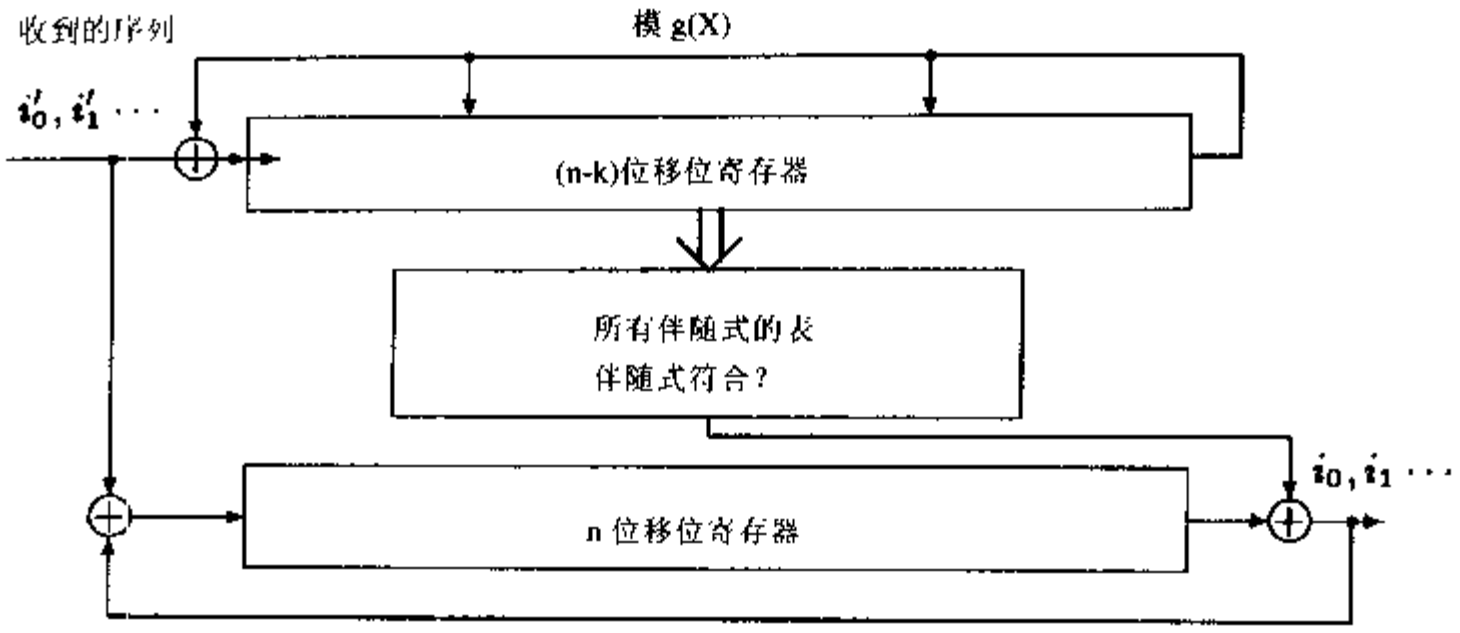


图 7-10　Meggitt 译码器的基本模块

为了使用缩短的代码并且重新得到代码的循环性质，必须计算所收到的数据 $a(x)$的前向耦合。这种条件可以通过利用下面的公式将代码缩短 b 位得到：

$$s[x] = a(x)i(x) \bmod g(x) = x^{n-k+b} i(x) \bmod g(x) \tag{7.43}$$

对于缩短的(57, 44, 6)BCH 码变成：

$$\begin{aligned} a(x) &= x^{63-50+6} \bmod g(x) = x^{19} \bmod g(x) \\ &= x^{19} \bmod (x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^3 + x + 1) \\ &= x^{10} + x^7 + x^6 + x^5 + x^3 + x + 1 \end{aligned}$$

得到的代码具有校正两个差错的能力。如果仅有 MSB 位置的差错需要校正，就需要储存总数为 $1+\binom{56}{1}=1+56=57$ 个不同的差错模式，如表 7-6 所示。通过仿真可以计算这 57 个伴随值，在【162】中列出了这些值。

表 7-6　可能的差错模式表

No.	差 错 模 式						
1	0	0	0	…	0	0	1
2	0	0	0	…	0	1	1
3	0	0	0	…	1	0	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
56	0	1	0	…	0	0	1
57	1	0	0	…	0	0	1

到目前为止，构成(57, 44, 6)BCH 码的 Meggitt 译码器的所有构造模块均有了。图 7-11 给出了这种译码器。

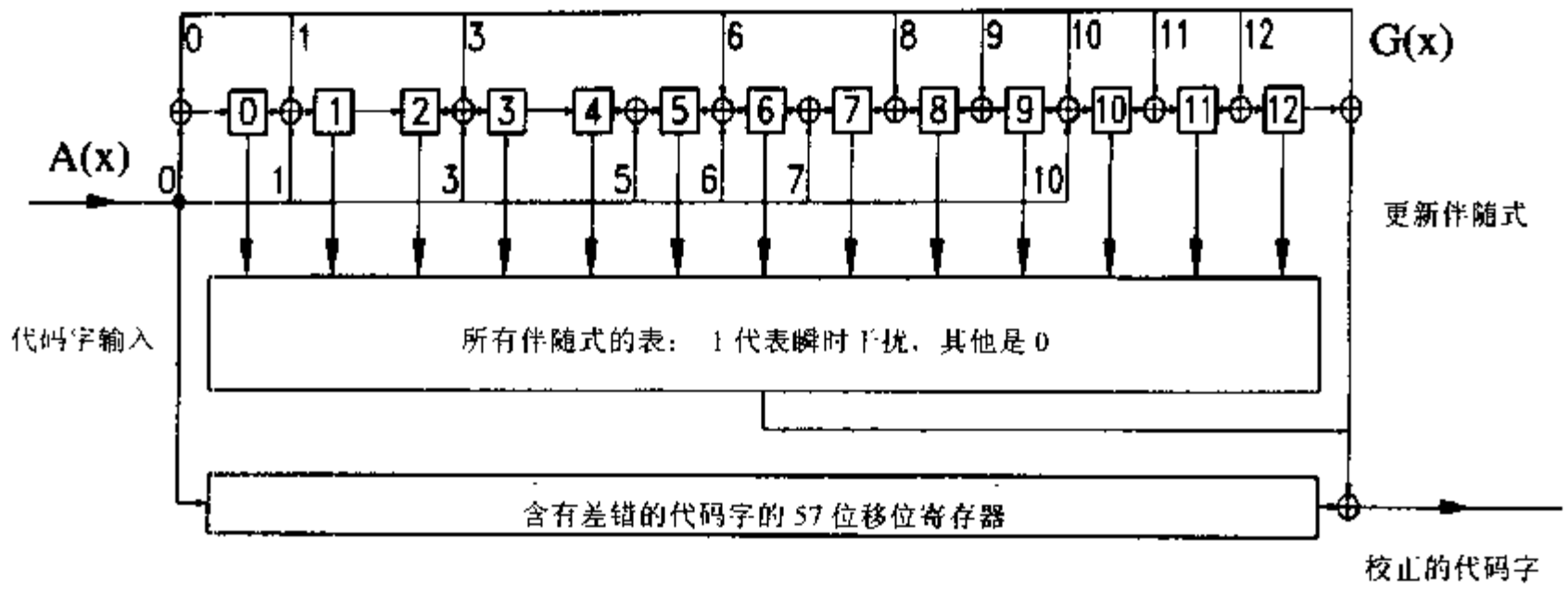


图 7-11　(57, 44, 6) BCH 码的 Meggitt 译码器

Meggitt 译码器具有两个级。在初始阶段，通过处理接收位模的生成多项式 $g(x)$计算伴随式，共用了 57 个循环。在第二阶段进行实际的差错校正。伴随式寄存器的内容与伴随式表的值一致。如果找到相应的表目，表就释放一个值，否则就释放 0。瞬时干扰位与接收位在移位寄存器中进行 XOR 运算，这样就将差错从移位寄存器中剔除了。瞬时干扰位也接入到伴随式寄存器上，剔除伴随式寄存器中的差错模式。伴随式寄存器和移位寄存器再一次受时钟驱动，就开始了下一次校正。最终，移位寄存器中应该包括校正过的字，而伴随式寄存器中应该包含全零字。如果伴随式不是零，就说明还有两个以上的差错，而这些差错是不能够用 BCH 码校正的。

对于 Meggitt 译码器的 FPGA 实现，我们仅关注的是伴随式表输入的大数(13)，因为典型 FPGA 的 LUT 均具有 4 到 8 个输入。可以用一个额外的 EPROM(仅适用于 Altera Flex 10K)或 4 个 2KB EAB 实现规模为 $2^{13}\times1$ 的表。伴随式寄存器连接到地址线上，它会给 57 个伴随式释放一个瞬时干扰(1)，否则就是 0。还可以使用逻辑合成工具和 FPGA 上的内部逻辑模块来计算表。Xilinx XNFOPT(用在【162】中)需要 132 个 LUT，每个 LUT 都是 $2^4\times2$ 位。如果采用现代的二元判定框图(Binary Decision Diagram，BDD)合成器类型[169, 170, 171]，其规模为 $2^4\times2$ 位，则这个数量(以额外的时间延迟为代价)可以减少到 58 个 LUT[172]。表 7-7 给出了采用 Flex 10K 作 Meggitt 译码器的不同种类伴随式表的预计工作量。

表 7-7　3 种基于 XC3K 实现的 Meggitt 译码器形式的 Altera FLEX 元器件的预计工作量[4]

(EAB 用作 $2^{11}\times1$ROM)

函　数　群	伴 随 式 表		
	使用 EAB 的情形	仅有 LC 的情形	BDD【172】
接口	36 个 LC	36 个 LC	36 个 LC
伴随式表	2 个 LC，4 个 EAB	264 个 LC	116 个 LC
64 位 FIFO	64 个 LC	64 个 LC	64 个 LC
Meggitt 译码器	12 个 LC	12 个 LC	12 个 LC
状态机	21 个 LC	21 个 LC	21 个 LC
总计	135 个 LC，4 个 EAB	397 个 LC	249 个 LC

7.2.3　卷积码

我们还要研究一下适合 FPGA 实现的卷积差错校正译码器。为了简化讨论，首先定义下面的约束，下面的约束条件均是通信系统的典型约束条件：

- 代码应该将译码器的复杂程度降低到最低，编码器的复杂程度倒是其次。
- 代码应该是线性系统的。
- 代码应该是卷积的。
- 代码应该提供随机差错校正。

规定系统代码允许提供一种断电模式，后者只接收没有差错校正的输入位[161]。如果信道是慢衰落的，就要规定随机差错校正代码。

图 7-12 给出了一个可行树形码的框图，而图 7-7 给出的则是可行的译码器框图。由于组织堆栈比较复杂，所以 Fano 译码器和堆栈译码器不是非常适合 FPGA 实现[160]。而卷积 $\mu P/\mu C$ 实现更加适合一些。在下一节中，将就硬件复杂性、Xilinx XC3K FPGA 中的 CLB 的使用和可实现的差错校正方面对维特比最大可能序列译码器和代数算法进行比较。

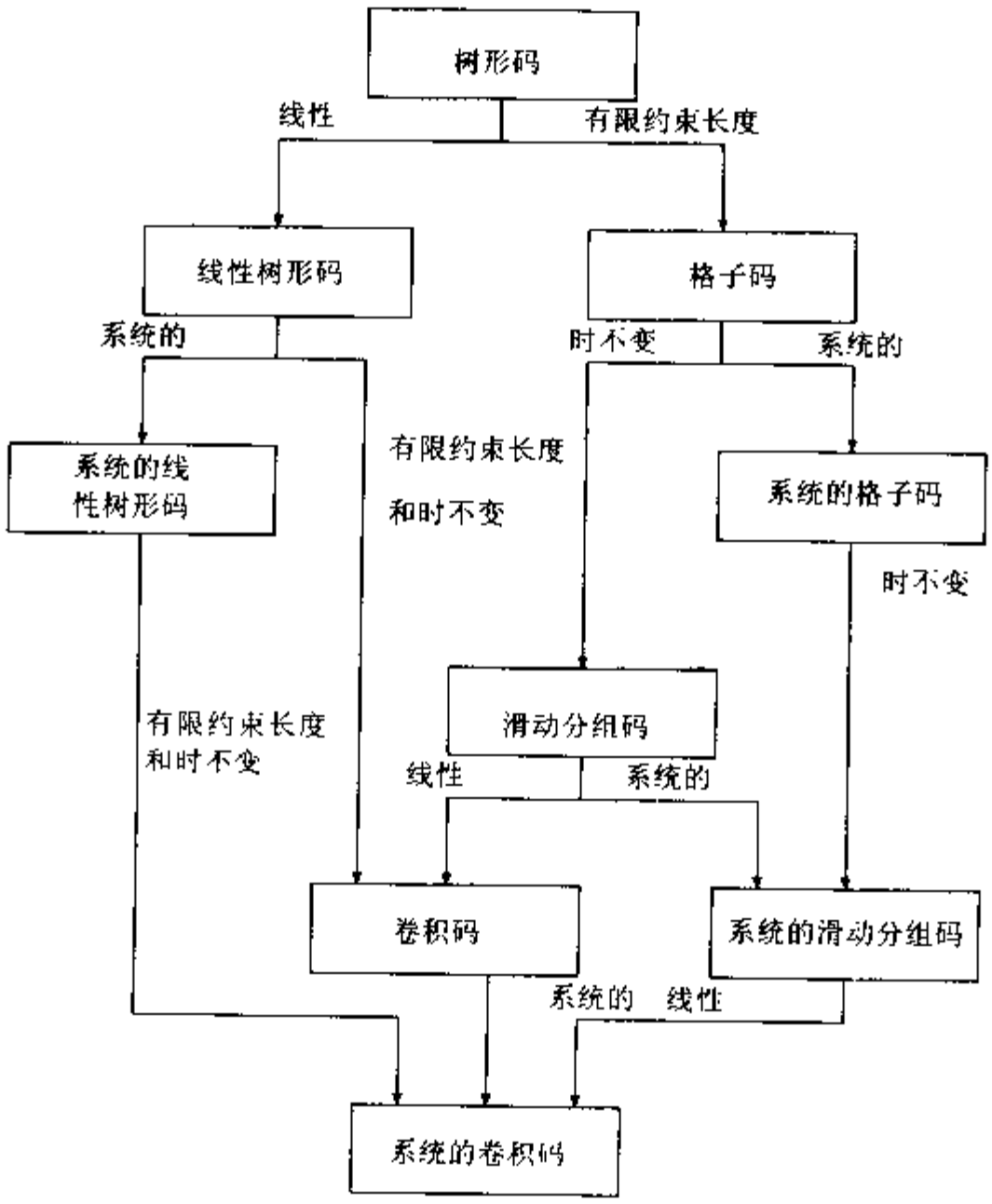


图 7-12　树形码的概观

1. 维特比最大可能序列译码器

维特比(Viterbi)译码器通过用最小海明距离计算相应发送器的序列来处理错误的序列。与之不同的是，代数算法通过格子图找出最优路径，也就是最优无记忆干扰序列的预估程序(memoryless noisy-sequence estimator, MLSE)。

维特比译码器的优点在于它的常数译码时间和 MLSE 最优性。而缺点就在于其较高的存储要求，并且代码最终限制在非常短的约束长度之内。图 7-13 和 7-14 给出了一个 $R=k/n=1/2$ 的译码器和伴随格子图。约束长度 $v=m\cdot k=2$，所以格子结构有 2^v 个节点，每个节点都有 $2^k=2$ 个引出边和至多 $2^k=2$ 个引入边。对于像这样的二进制格子结构($k=1$)而言，可以很方便地看出 0 作为上边，而 1 作为下边。

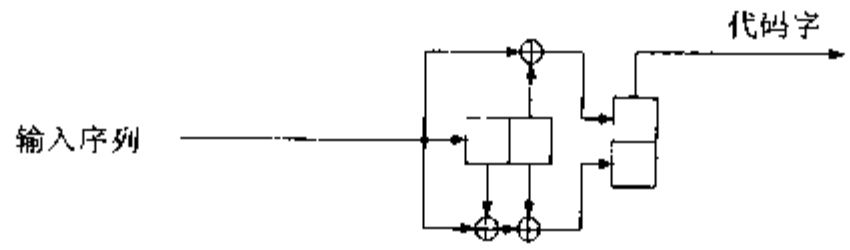


图 7-13　$R=1/2$ 卷积译码器的编码器

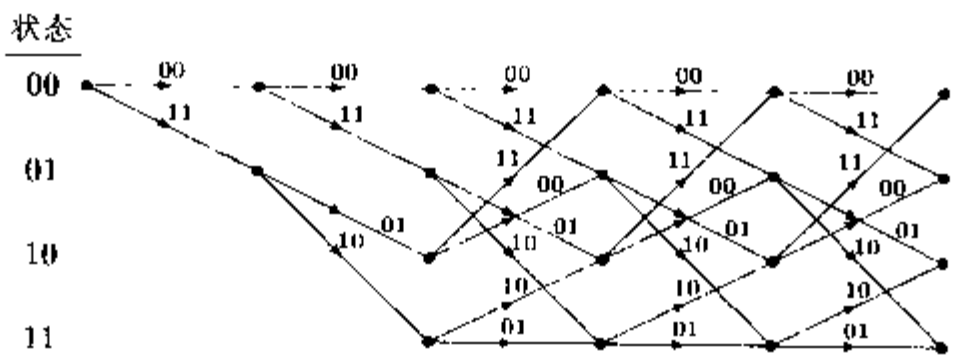


图 7-14　$R=1/2$ 卷积译码器的格子结构

对于 MLSE 译码而言，仅仅储存一个给定层面上通过节点的 2^v 个路径(及其度量标准)是足够的。因为 MLSE 路径必须要通过这些节点中的一个。在图中存在具有最高度量标准的残存部分，具有比残存部分更小的度量标准的引入路径就不需要储存了，因为这些路径永远也不会成为 MLSE 路径的一部分。然而，在任意给定时间内，如果最大度量标准是短差错序列的一部分，它也可能不是 MLSE 路径的一部分。否决这样一个局部信号是对用存储器解调数字 FM 信号的一种近似[173]。【124，381 页】和【92，120-3 页】的仿真结果显示，足够构造 4 到 5 倍约束长度的路径存储。无限路径存储也不会有明显的提高。

维特比译码器硬件包括 3 个主要部分：带有路径存储器的输出译码器(参阅图 7-15)、残存部分的计算和最大值检测(参阅图 7-16)。路径存储器有 $4v2^v$ 个位，使用了 $2v2^v$ 个 CLB。输出译码器采用了$(1+2+\ldots+2^{v-1})$个 2→1 多路复用器。每个度量标准更新加法器、寄存器和比较均是$(\lceil\log_2(v*n)\rceil+1)$个位宽。最大计算需要额外的比较、2→1 多路复用器和一个译码器。

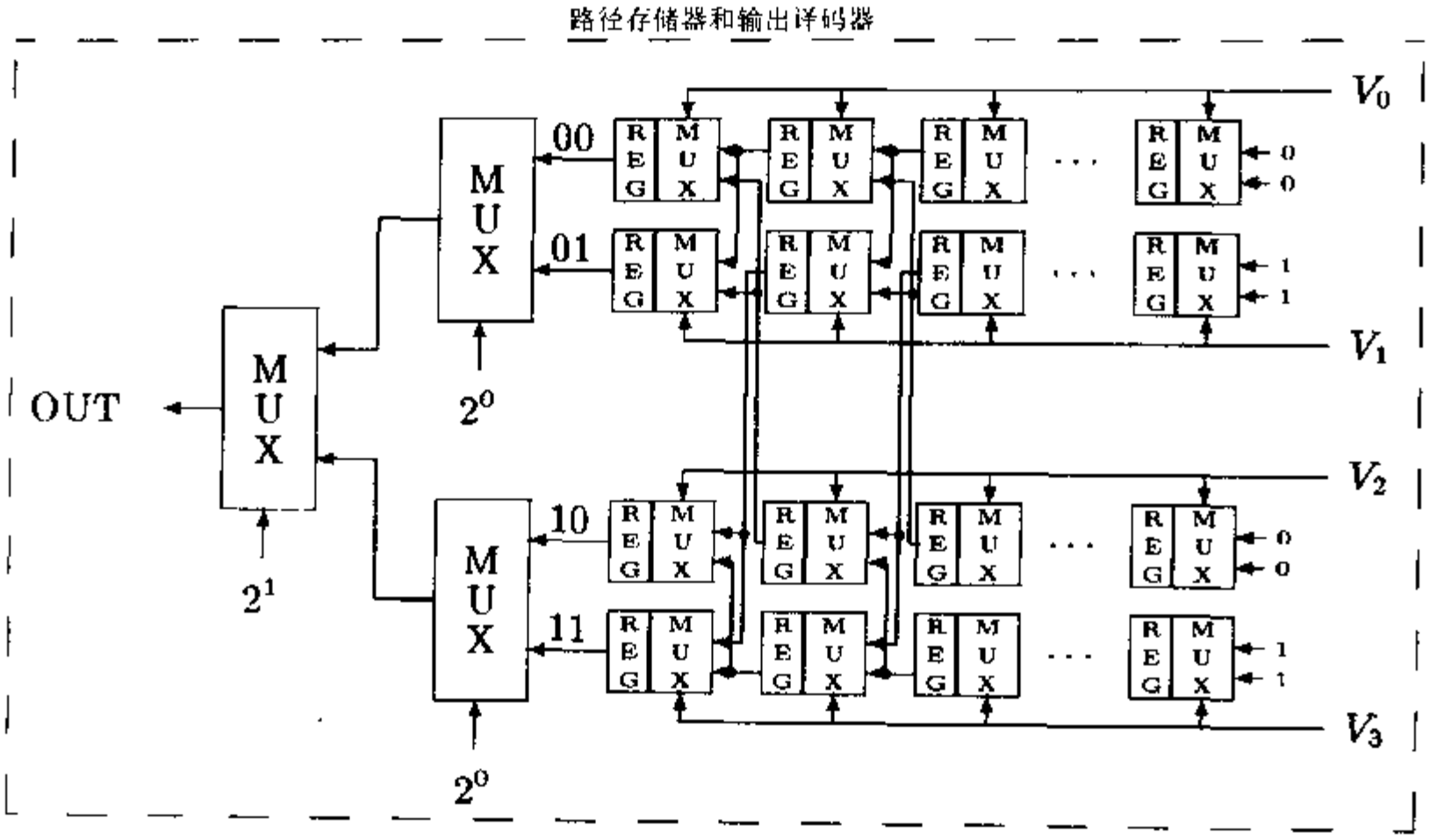


图 7-15　约束长度为 4v 和 2^{v-2} 个节点的维特比译码器：路径存储和输出译码器

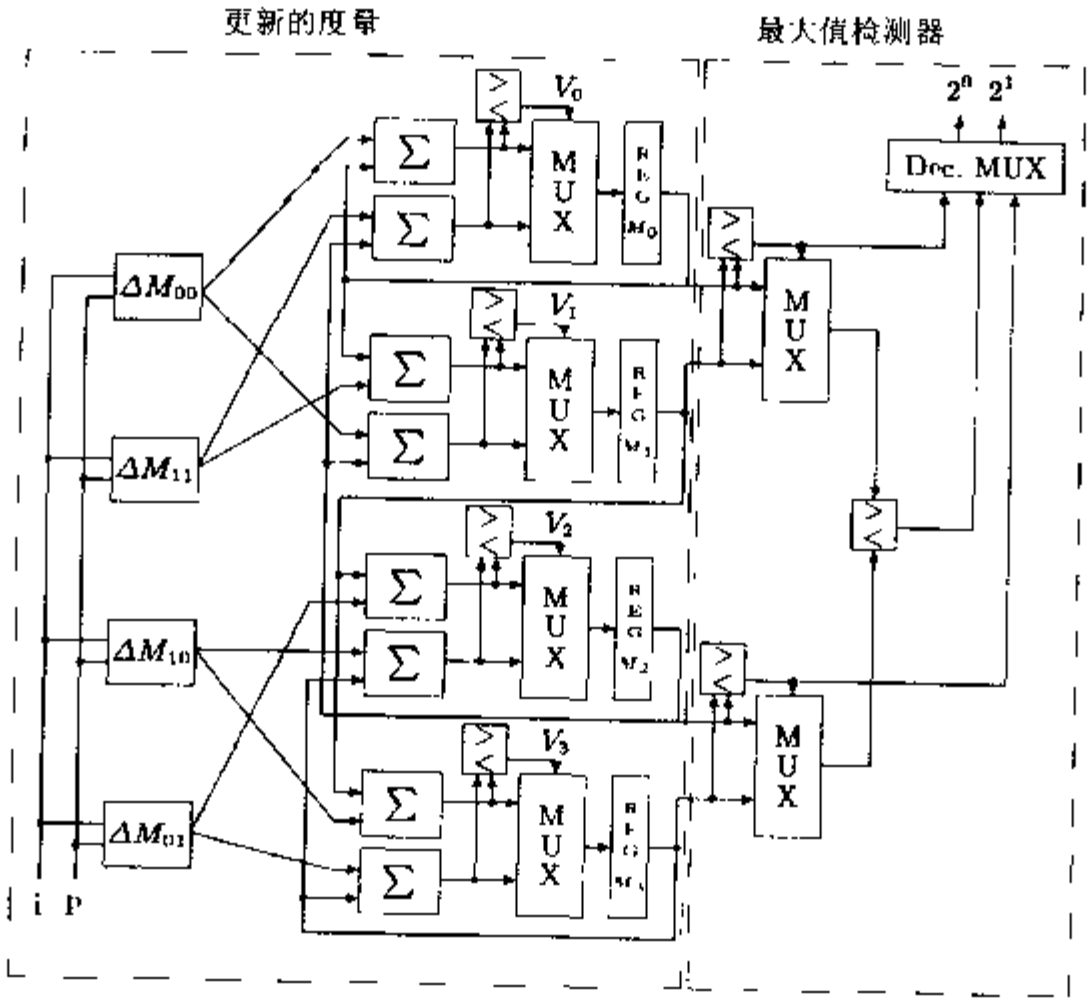


图 7-16　约束长度为 4v 和 2^{v-2} 个节点的维特比译码器：度量标准计算

k>1 的译码器的硬件就太复杂了，还不能够用今天的 FPGA 来实现。当 n>2 时，信息率 $R=1/n$ 太小，所以最适合的代码率是 $R=1/2$。表 7-8 列出了约束长度 v=2, 3, 4 和一般情况下 $R=1/2$ 时 XC3K FPGA 中的硬件复杂性(按 CLB 计算)。可以看出，复杂性随着约束长度 v 的增加而成指数增长，所以约束长度应该尽可能的短。尽管这样短的约束长度提供的短窗函数仅能够校正极少的差错，但 MLSE 算法还是保证了可以接受的性能。

表 7-8　$R=1/2$ 的维特比译码器 v=2, 3, 4 和一般情况下的硬件复杂性(按 CLB 计算)

函　数	v=2	v=3	v=4	$v\in N$
路径存储器	16	48	128	$4\cdot v\cdot 2^{v-1}$
输出译码器	1.5	3.5	6.5	$1+2+\ldots+2^{v-2}$
度量标准 ΔM	4	4	4	4
度量标准清除	1	2	4	$\lceil(2+4+\ldots+2^{v-1})/4\rceil$
度量标准加法器	24	64	128	$(\lceil\log_2(nv)\rceil+1)\cdot 2^{v+1}$
残存部分多路复用器	6	24	48	$(\lceil\log_2(nv)\rceil+1)\cdot 2^{v-1}$
度量标准比较	6	24	48	$(\lceil\log_2(nv)\rceil+1)\cdot 2^{v-1}$
最大比较	4.5	14	30	$(\lceil\log_2(nv)\rceil+1)\cdot\frac{1}{2}\cdot(1+2+\cdots+2^{v-1})$
多路复用器	3	12	28	$(2+\ldots+2^{v-1})\cdot\frac{1}{2}\cdot(\lceil\log_2(nv)\rceil+1)$
译码器	1	2	4	$\lceil(2+\ldots+2^{v-1})/4\rceil$
状态机	4	4	4	
总计：	67	197.5	428.5	

接下来就需要选择一个合适的生成多项式。在文献(【174，306-8 页】、【175，465 页】、【164，402-7 页】、【124，367 页】)中指出，对于给定的约束长度，非系统代码要比系统代码的性能要更好，但是应用非系统代码与使用没有差错校正的信息位的要求相违背。**快速搜索**(Quick Look In, QLI)代码就是 $R=1/2$ 的非系统卷积代码，它所提供的间隙值与任意已知约束长度 $v=2$ 至 4 的代码所提供的值相同[176]。QLI 代码的优点就是信息序列的重构只需要一个 XOR 门。v=2, 3, 4 的 QLI 分别具有 $d_f=5, 6, 7$ 的间隙[175, 465 页]。这似乎是对低功耗的一个良好的折衷。表 7-9 上面的部分给出了八进制表示形式的生成多项式。

表 7-9　采用 QLI 代码的维特比译码器 $v=2$ 到 4 的一致限权重

代码	O1＝7 O2＝5	O1＝74 O2＝54	O1＝66 O2＝46
约束长度	v=2	v=3	v=4
距　离	权　重 w_j		
0-4	0	0	0
5	1	0	0
6	4	2	0
7	12	7	4
8	32	18	12
9	80	49	26
10	192	130	74
11	448	333	205

(续表)

距　离	权　重 w_j		
12	1024	836	530
13	2304	2069	1369
14	5120	5060	3476
15	11264	12255	8470
16	24576	29444	19772
17	53079	64183	43062
18	109396	126260	83346
19	103665	223980	147474
20	262144	351956	244458

2. QLI 译码器的差错校正性能

为了计算 QLI 译码器的差错校正性能，运用一致限的理论是非常方便的。由于 QLI 代码是线性的，所以差错序列可以按与零序列的差进行计算。如果一个序列以零状态开始，MLSE 译码器就会作出错误的判断，并且不同于 j 个独立时间步中的空字，包含至少 $j/2$ 个 1。这种情形出现的概率是：

$$P_j = \begin{cases} \sum_{i=(j+1)/2}^{j} \binom{j}{i} p^i q^{j-i} & j \text{ 是奇数} \\ \frac{1}{2}\binom{j}{j/2} p^{j/2} q^{j/2} + \sum_{i=(j/2+1)}^{j} \binom{j}{i} p^i q^{j-i} j & j \text{ 是偶数} \end{cases} \tag{7.44}$$

位误码概率公式需要做的惟一的事情就是用代码的权重 j 计算路径的数量 w_j，这是一个很简单的编程任务[160, C.4]，由于 P_j 随着 j 的增加而呈指数减少，所以仅需计算前面几项 w_j 即可。表 7-9 给出了 j==0 到 20 的 w_j。总的差错概率可以用下面的公式计算：

$$P_b < \frac{1}{k}\sum_{j=0}^{x} w_j P_j \tag{7.45}$$

3. 伴随式代数译码器

用伴随式译码器和编码器(图 7-17)像标准分组译码器(图 7-18)一样计算数据序列中的若干奇偶位。译码器新近计算的奇偶位是 XOR 接收的用于创建“伴随”字的奇偶位，如果在传输中出现差错，伴随字就是非零的。差错位置和值是由伴随值确定的。与分组码只采用一个生成多项式相反的是，数据传输率 $R=k/n$ 时卷积代码具有 $k+1$ 个生成多项式。完成的生成元可以写成一个简洁的 $n\times k$ 阶生成元矩阵。图 7-18 中的编码器矩阵如下：

$$G(x) = [1 \quad x^{21} + x^{20} + x^{19} + x^{17} + x^{16} + x^{13} + x^{11} + 1] \tag{7.46}$$

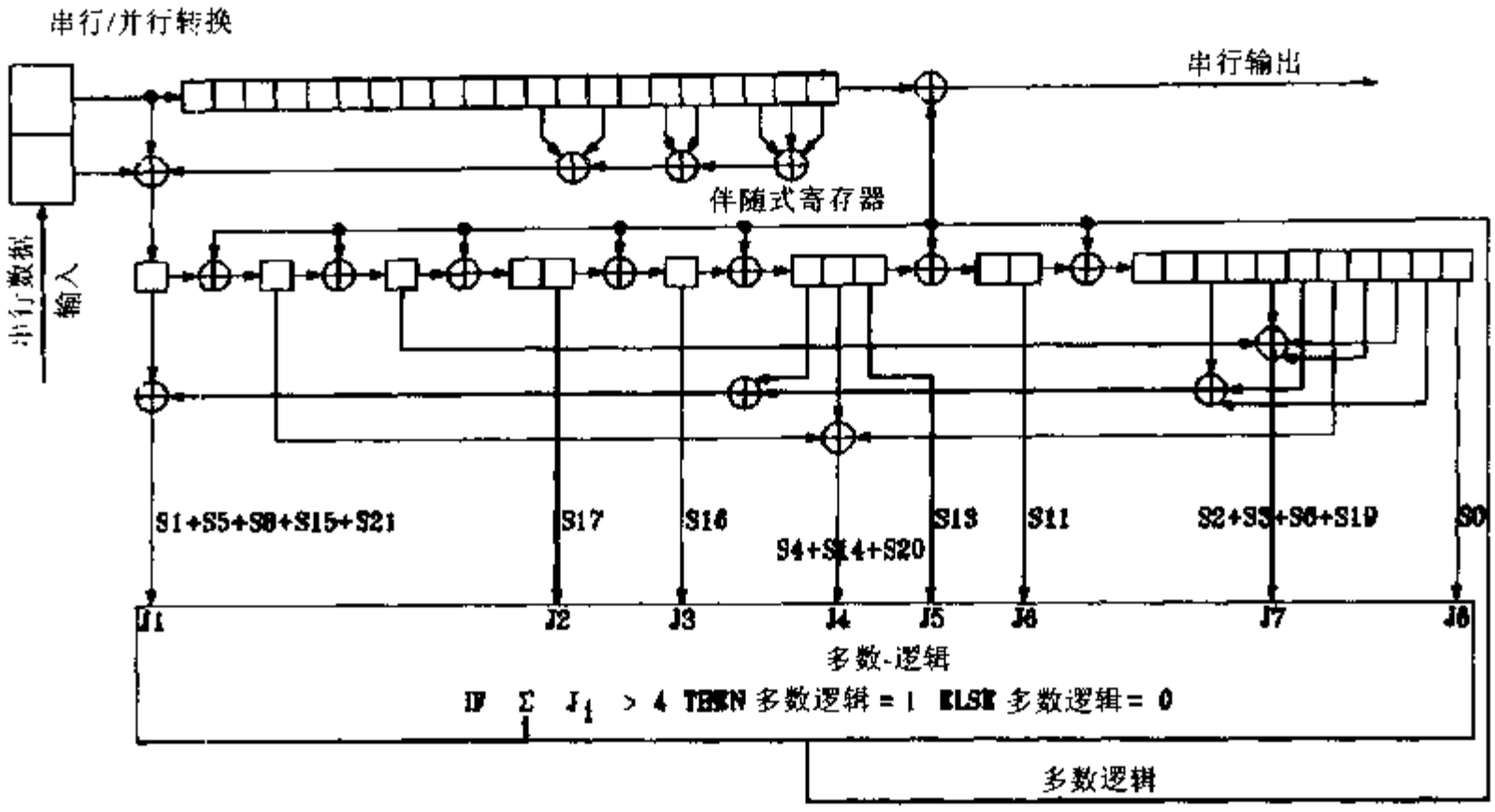


图 7-17　$J=8$ 的逐次逼近多数逻辑译码器

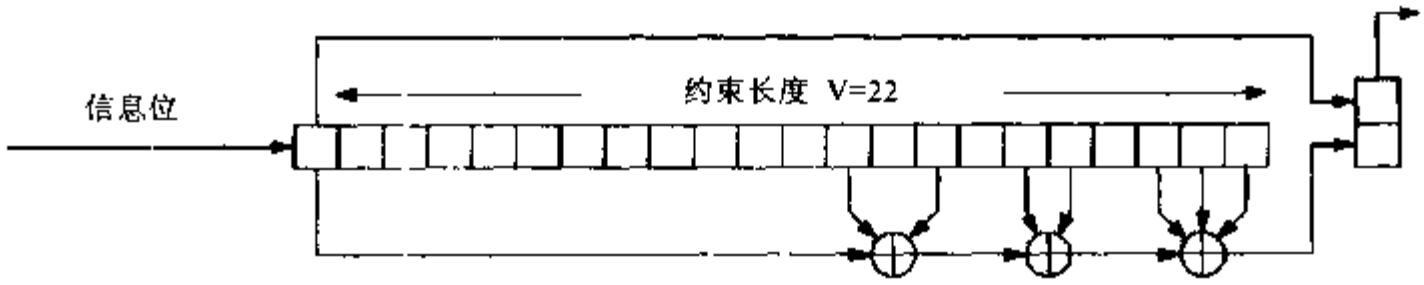


图 7-18　数据传输率 $R=1/2$ 且约束长度 $v=22$ 的系统(44, 22)编码器

系统代码的矩阵形式是 $\boldsymbol{G}(x)=[\boldsymbol{I}:\boldsymbol{P}(x)]$。已知 $\boldsymbol{G}\cdot\boldsymbol{H}^T=0$，奇偶校验矩阵 $\boldsymbol{H}(x)=[-\boldsymbol{P}(x)^T:\boldsymbol{I}]$ 很容易计算。所期望的伴随式向量就是 $\boldsymbol{S}=\boldsymbol{v}\cdot\boldsymbol{H}^T$，其中 $\boldsymbol{v}$ 是接收到的位序列。

接下来伴随式译码器在表中查询所计算的伴随式，并找到校正序列。为了限制表的规模，只允许包含第一个位的位置出现差错的序列。如果译码器需要校正的位多于一个，在校正之后就不要清除伴随式。必须将伴随式值从伴随式寄存器中减去(请参阅图 7-17 中的“多数逻辑”信号)。

标准的卷积译码器需要一个 22 位的表。但是很遗憾的是实现一个好的大于 4 到 11 位地址的 FPGA 查询表非常困难[161]。在此，多数逻辑代码——一类特殊的伴随式可译码的代码就表现出了优势。这种标准自成正交代码(Canonical Self-Orthogonal Code，CSOC)在奇偶校验矩阵 $\{\boldsymbol{A}_k\}$(其中 J 列用作计算伴随式的正交集)的第一行有专有的代码[164, 284 页]。这样，在第一个位的位置出现的每一个差错就会在伴随式寄存器中生成至少 $\lceil J/2\rceil$ 个代码。译码规则就是：

$$e_0'=\begin{cases}1 & \sum_{k=1}^{J}A_k>\lceil J/2\rceil \\ 0 & \text{其他}\end{cases} \tag{7.47}$$

“多数逻辑代码”名称的意义就是：取代昂贵的伴随式表，只需要多数投票即可。Massey[164, 289 页]已经设计了一类多数逻辑代码，称作逐次逼近代码，这种代码不是直接计算伴随式向量，而是操纵伴随位的组合来得到与 e_0' 正交的向量。以很小的硬件成本代价得到了比卷积 CSOC 代码更佳的差错校正性能。表 7-10 列出了一些数据传输率 $R=1/2$ 的逐次逼近代码。图 7-17 给出了一个 $J=8$ 的逐次逼近译码器。表 7-11 给出了 $J=4$ 至 10 的译码器按 CLB 计算的复杂性。

表 7-10　一些多数逻辑可译码的逐次逼近代码[164, 406 页]

J	t_{MD}	v	生成多项式	正交方程式
2	1	2	$1+x$	s_0, s_1
4	2	6	$1+x^3+x^4+x^5$	s_0, s_3, s_4, s_1+s_5
6	3	12	$1+x^6+x^7+x^9+x^{10}+x^{11}$	$s_0, s_6, s_7, s_9, s_1+s_3+s_{10}, s_4+s_8+s_{11}$
8	4	22	$1+x^{11}+x^{13}+x^{16}+x^{17}+x^{19}+x^{20}+x^{21}$	$s_0, s_{11}, s_{13}, s_{16}, s_{17}, s_2+s_3+s_6+s_{19}, s_4+s_{14}+s_{20}, s_1+s_5+s_8+s_{15}+s_{21}$
10	5	36	$1+x^{18}+x^{19}+x^{27}+x^{28}+x^{29}+x^{30}+x^{32}+x^{33}+x^{35}$	$s_0, s_{18}, s_{19}, s_{27}, s_1+s_9+s_{28}, s_{10}+s_{20}+s_{29}, s_{11}+s_{30}+s_{31}, s_{13}+s_{21}+s_{23}+s_{32}, s_{14}+s_{33}+s_{34}, s_2+s_3+s_{16}+s_{24}+s_{26}+s_{35}$

表 7-11　$J=4$ 至 10 多数逻辑译码器按 CLB 计算的复杂性

函　数	$J=4$	$J=6$	$J=8$	$J=10$
寄存器	6	12	22	36
XOR 门	2	4	7	11
多数逻辑电路	1	5	7	15
和	9	22	36	62

4. 逐次逼近译码器的差错校正能力

要计算逐次逼近代码的差错校正性能，首先必须注意在一个窗函数内将约束长度乘以 2，代码给出多达 $\lfloor J/2 \rfloor$ 位的差错[124, 440 页]：

$$P(J)=\sum_{k=0}^{\lfloor J/2 \rfloor}\binom{2v}{k}p^k(1-p)^{2v-k} \tag{7.48}$$

图 7-19 中的计算机仿真结果表明该仿真与这一等式是相一致的。(n, k)代码的等价单个差错概率 P_B 可以用下式计算：

$$P(J)=P(0)=(1-P_B)^k \tag{7.49}$$

$$\rightarrow P_B=1-e^{\ln(P(J))/k} \tag{7.50}$$

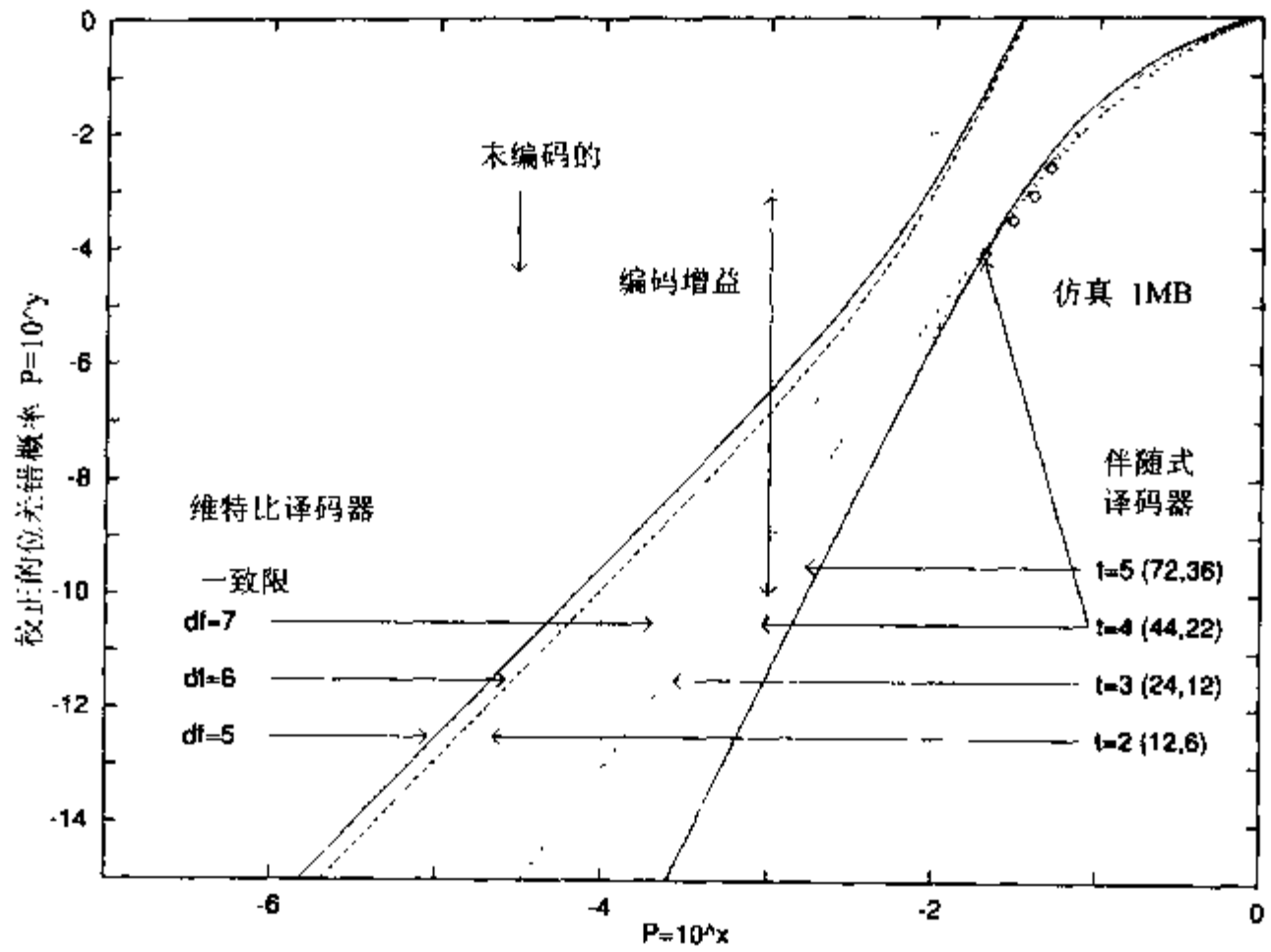


图 7-19 维特比和多数逻辑译码器的性能比较

5. 最终的比较

图 7-19 给出了维特比译码器和多数逻辑译码器的差错校正性能。对于可比较的硬件成本(维特比译码器：$v=2$，$d_f=5$，67 个 CLB，而逐次逼近译码器：$t=5$，62 个 CLB)，多数逻辑译码器的性能要更好，当允许更大的约束长度时，这一优势就更加明显。维特比算法的最优 MLSE 特性不能够补偿其短约束长度。

7.2.4 FPGA 的加密技术算法

大多数通信系统都采用数据流密码保护相关的信息，如图 7-20 所示。密钥顺序 K 大于或小于“伪随机序列”(为发送器和接收器所知)，利用 XOR 函数模 2 的特性，纯文本 P 可以在接收器端重构，这是因为：

$$P \oplus K \oplus K = P \oplus 0 = P \tag{7.51}$$

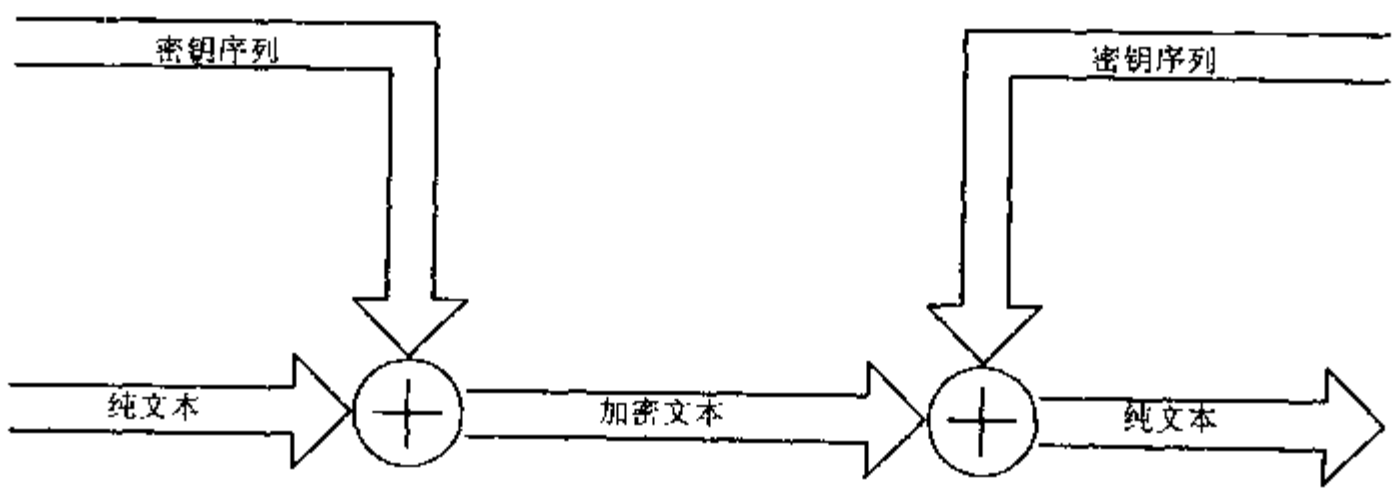


图 7-20 同步数据流密码的规则

接下来将要在一种基于线性反馈移位寄存器(Linear Feedback Shift Register, LSFR)的算法和一种“数据加密标准”(Data Encryption Standard, DES)的加密算法之间作一个比较。两种算法均不需要大规模的表，而且都非常适合用 FPGA 实现。

1. 线性反馈移位寄存器算法

最大序列长度 LFSR 对理想的安全密钥而言是一种非常好的方法，因为 LFSR 具有良好的统计学特性(参阅【177】、【178】)。换句话说就是很难在密码攻击中分析这个序列(是一种称作**密码分析学**的分析法)。由于可以用 PFGA 进行逐位的设计，所以这样的 LFSR 用 FPGA 实现会比用 PDSP 更加有效。图 7-21 给出了两种长度为 8 的 LFSR 的可行实现。

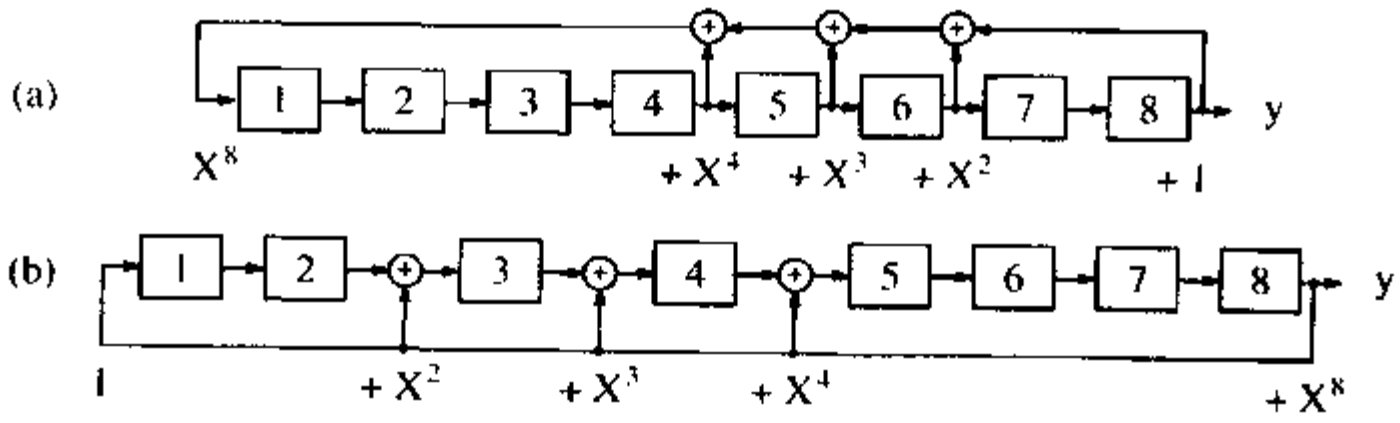


图 7-21　LFSR 的可行实现

对于 XOR LFSR，存在全是全零字的可能性，但是这种情况应该永远也不会出现。如果循环是从非零字开始的，则循环长度总是 $2^{l}-1$。通常，如果 FPGA 是在全零字状态被唤醒，就会更加方便地使用“镜像”或翻转的 LFSR 电路。如果全零字是一种正确模式，并且生成了精确的转置序列，就需要用一个“非 XOR”或 XNOR 门代替 XOR 门。这样的 LFSR 可以很容易用 VHDL 中的 PROCESS 声明来设计，如下例所示。

例 7.12　长度为 6 的 LFSR

下面就是长度为 6 的 LFSR 的 VHDL 代码[1]。

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY lfsr IS                                  ------> Interface
  PORT ( clk : IN   STD_LOGIC;
         y   : OUT STD_LOGIC_VECTOR(6 DOWNTO 1));
END lfsr;

ARCHITECTURE flex OF lfsr IS

  SIGNAL   ff  :   STD_LOGIC_VECTOR(6 DOWNTO 1);
```

注 1：这一例子相应的 Verilog 代码文件 lfsr.v 可以在附录 A 中找到。

```
BEGIN

  PROCESS              -- Implement length 6 LFSR with xnor
  BEGIN
    WAIT UNTIL clk = '1';
    ff(1) <= NOT (ff(5) XOR ff(6));
    FOR I IN 6 DOWNTO 2 LOOP        -- Tapped delay line:
      ff(I) <= ff(I - 1);           -- shift one
    END LOOP;
  END PROCESS ;

  PROCESS (ff)
  BEGIN                  -- Connect to I/O cell
    FOR k IN 1 TO 6 LOOP
      y(k) <= ff(k);
    END LOOP;
  END PROCESS;

END flex;
```

从图 7-22 的仿真结果可以得出结论，LFSR 通过所有可能的位模式时，生成的最大序列长度 $2^6 - 1 = 63 \approx 630\text{ns}/10\text{ns}$。时钟周期是 10ns，Registered Performance 是 100MHz。

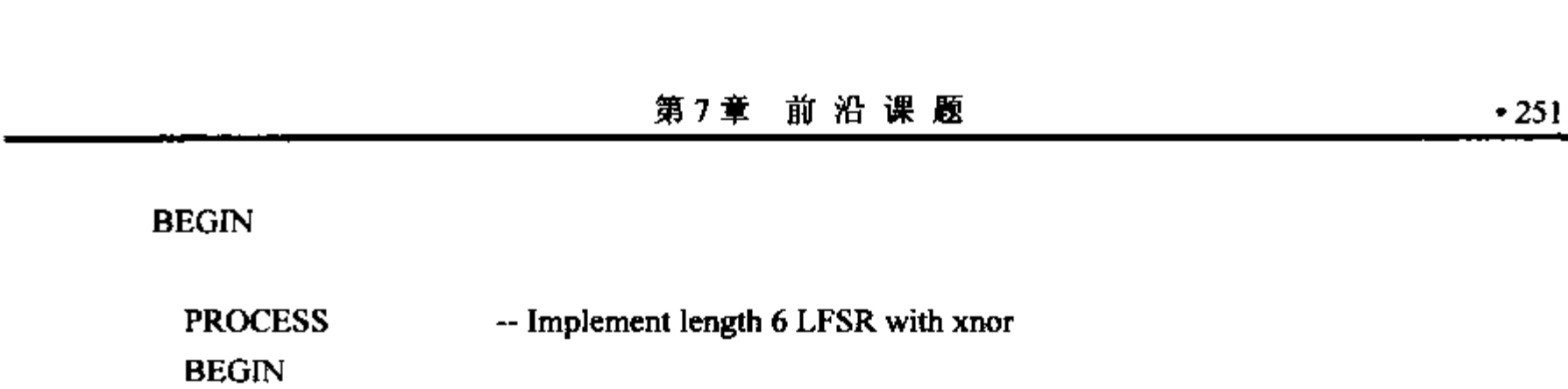


图 7-22 LFSR 的仿真

注意：

LFSR 序列的完整循环满足 Golomb 定义的最优长度 $2^l - 1$ 伪随机序列的 3 个准则[179,188页]：

(1) 在一个循环中，1 和 0 的数量之差不大于 1。

(2) 长度为 k 的运算(例如：111...序列，000...序列)是总运算的 $1/2^k$。

(3) 自相关函数 $C(T)$是常数，$T \in [1, n-1]$。

LFSR 通常用 GF(2)中的质数多项式来构造，并采用图 7-21 所示的电路。W. Stahnke[165]已经编辑了一列多达 168 阶的这样的多项式。这一论文可以在 http://www.jstor.org 在线阅读。用目前的可行数学软件包，例如：MAPLE、MUPAD 或 MAGMA 可以很容易扩展这个列表。下面就是用 MAPLE 计算 x^l+x^a+1 类型且 a 最小的质数多项式的代码例子。

```
with(numtheory):
```

```
for l from 2 by 1 to 45 do
  for a from 1 by 1 to l - 1 do
    if (Primitive(x^l+x^a+1) mod 2) then
    print(l,a);
    break;
  fi;
  od;
od;
```

表 7-12 根据图 7-21(a)给出了前 45 个最大长度 LFSR 所需要的 XOR 列表。例如第 14 个多项式(14, 12, 11, 9)的条目意思是这个质数多项式为：

$$p_{14}(x) = x^{14} + x^{14-13} + x^{14-11} + x^{14-9} + 1 = x^{14} + x^5 + x^3 + x + 1$$

当 l>2 时，这些质数多项式均是“双生子”，所有的质数多项式均是如此[180]。这也是“时域”可逆的形式 $x^l+x^{l-a}+1$。

表 7-12　前 45 个 LFSR 的列表

l	指　数	l	指　数	l	指　数
1	1	16	16，14，13，11	31	31，28
2	2，1	17	17，14	32	32，30，29，23
3	3，2	18	18，11	33	33，20
4	4，3	19	19，18，17，14	34	34，31，30，26
5	5，3	20	20，17	35	35，33
6	6，5	21	21，19	36	36，25
7	7，6	22	22，21	37	37，36，33，31
8	8，6，5，4	23	23，18	38	38，36，33，31
9	9，5	24	24，25，21，20	39	39，35
10	10，7	25	25，22	40	40，37，36，35
11	11，9	26	26，25，24，20	41	41，38
12	12，11，8，6	27	27，26，25，22	42	42，39，38，35
13	13，12，10，9	28	28，25	43	43，41，40，36
14	14，13，11，9	29	29，27	44	44，42，41，37
15	15，14	30	30，29，26，24	45	45，44，43，41

Stahnke[165] [18. XAPP52]已经计算了 x^l+x^a+1 类型的质数多项式。当 l<45 时，4 个元素的质数多项式，也就是$(x^l+x^b+x^a+1)$，不存在。但是可以找到 $x^l+x^{a+b}+x^b+x^a+1$ 类型的多项式，Stahnke 用这些多项式替代那些 x^l+x^a+1(l = 8, 12, 13 等)类型多项式不存在的情形。

表 7-12 中为抽头指数计算的 4 个元素的 LFSR 具有最大和(也就是 $a+b$)，一会就会看到，在多级 LFSR 实现中，这样会将复杂程度降到最低。

如果同时使用 n 位随机位，就有可能用时钟信号驱动 LFSR n 次，通常只使用 LFSR 的最低 n 位并不是一种好方法，因为这样会产生较弱的随机特性，也就是加密的安全性比较低。但是可以计算 n 位移位的等式。这样只需要一个时钟周期就可以生成 n 个新随机位。如果采用 LFSR 的“状态空间”表达式，就可以更加容易地计算所需要的等式。下面给出了一个例题。

例 7.13 三同步 LFSR

假定计算随机序列的质数多项式长度为 6，也就是 $p=x^6+x+1$。接下来的任务就是在一个时钟周期内计算 3 个“新”位。为了获得所需要的等式，首先必须计算 LFSR 的状态空间表达式，也就是 $\boldsymbol{x}(t+1)=\boldsymbol{A}\boldsymbol{x}(t)$：

$$\begin{bmatrix} x_6(t+1) \\ x_5(t+1) \\ x_4(t+1) \\ x_3(t+1) \\ x_2(t+1) \\ x_1(t+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_6(t) \\ x_5(t) \\ x_4(t) \\ x_3(t) \\ x_2(t) \\ x_1(t) \end{bmatrix} \tag{7.52}$$

有了这个状态空间表达式，就可以用实际值 $\boldsymbol{x}(t)$和传递矩阵 $\boldsymbol{A}$ 计算新的值 $\boldsymbol{x}(t+1)$。要计算 $\boldsymbol{x}(t+2)$的值，只需要计算 $\boldsymbol{x}(t+2)=\boldsymbol{A}\boldsymbol{x}(t+1)=\boldsymbol{A}^2\boldsymbol{x}(t)$，下一次迭代就得出 $\boldsymbol{x}(t+3)=\boldsymbol{A}^3\boldsymbol{x}(t)$。$n$ 同步 LFSR 的等式就可以通过计算 $\boldsymbol{A}^n \bmod 2$ 得到。当 $n=3$ 时有：

$$\boldsymbol{A}^3 \bmod 2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{7.53}$$

正像所期望的，对于寄存器 x_6 至 x_4 只需要 3 个位置的移位，而其他 3 个值 x_1 至 x_3 则需要用一个 EXOR 操作计算。下面就是实现三同步 LFSR 的 VHDL 代码[2]：

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY lfsr6s3 IS                              ------> Interface
  PORT ( clk : IN   STD_LOGIC;
         y   : OUT STD_LOGIC_VECTOR(6 DOWNTO 1));
END lfsr6s3;

ARCHITECTURE flex OF lfsr6s3 IS
```

注 2：这一例子相应的 Verilog 代码文件 lfsr6s3.v 可以在附录 A 中找到。

```
  SIGNAL ff : STD_LOGIC_VECTOR(6 DOWNTO 1);

BEGIN

  PROCESS    -- Implement three step length-6 LFSR with xnor
  BEGIN
    WAIT UNTIL clk = '1';
    ff(6) <= ff(3);
    ff(5) <= ff(2);
    ff(4) <= ff(1);
    ff(3) <= NOT (ff(5) XOR ff(6));
    ff(2) <= NOT (ff(4) XOR ff(5));
    ff(1) <= NOT (ff(3) XOR ff(4));
  END PROCESS ;

  PROCESS (ff)
  BEGIN                    -- Connect to I/O cell
    FOR k IN 1 TO 6 LOOP
      y(k) <= ff(k);
    END LOOP;
  END PROCESS;

END flex;
```

图 7-23 给出了三同步 LFSR 设计的仿真结果。将图 7-23 中 LFSR 的仿真结果与图 7-22 中单步 LFSR 的仿真结果相比较，就可以得出结论：每 3 列值出现一次，周期长度从 $2^6 - 1$ 降低到$(2^6 - 1)/3=21$。

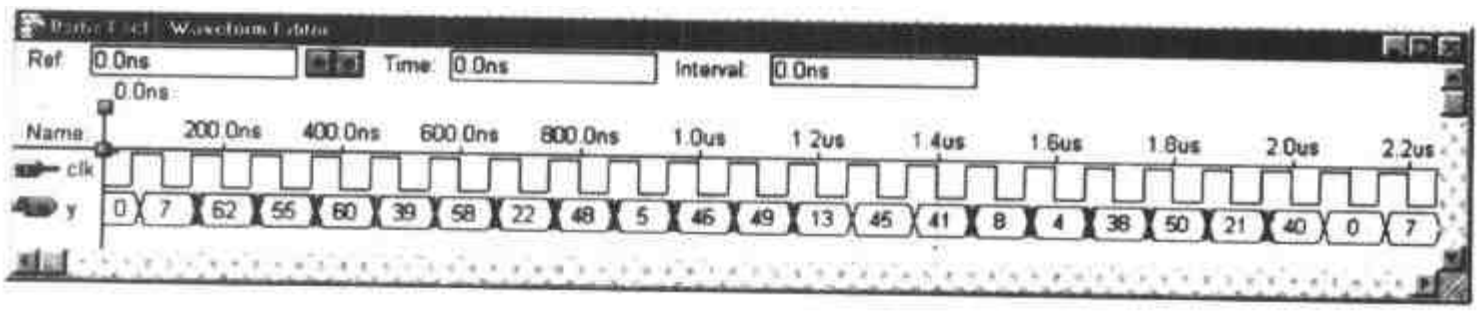


图 7-23　多步 LFSR 仿真结果

为了实现这样的多步 LFSR，我们想选择可以将电路工作量降到最低的质数多项式。这可以通过计数 $A^k \bmod 2$ 和 1 或寄存器的矩阵扇入端数中的非零项来计算，后者对应每一行中 1 的数量。当移位较少时，对于图 7-21 中的电路，扇入端数比较有优势。还可以看到[3]：如果反馈信号在矩阵 A 中闭合，那么矩阵 A_k 中的一些项就会由于模 2 运算而变成 0。正像前面所提到的，表 7-12 中 2 和 4 抽头 LFSR 数据经过计算就可以生成所有抽头的最大和。对于同样的和，会选

注 3：很明显，选择最小实现工作量的 LFSR 是不适合的，因为有 $\phi(2^l - 1)/l$ 个质数多项式，其中 $\phi(x)$是计算与 x 互质的数的欧拉函数。例如：一个 16 位寄存器具有 $\phi(2^{16} - 1)/16=2048$ 个不同的质数多项式[1801]！

择最小的抽头具有更大的值的质数多项式，例如：(11, 12)就比(10, 13)要好。之所以这样选择，是因为对于最大长度 LFSR 而言，抽头 *l* 是必须遵循的，其他值应该根据这个抽头的值来选择。

例如：如果采用 Stanke 的 $s_{14,a}(x)=x^{14}+x^{12}+x^{11}+x+1$ 质数多项式，就会在 $n=8$ 的多步 LFSR 中生成 58 个项，而如果采用表 7-12 中的 $p_{14}=x^{14}+x^5+x^3+x^1+1$ LFSR(也就是抽头 14,13,11,9)，A^8 mod 2 矩阵就只有 35 项(请参阅练习 7.6)。图 7-24 给出了采用图 7-21 中给出的两种不同实现的 LFSR 多项式中 1 的总数量，而图 7-25 给出这种 LSFR 的最大扇入端数(也就是一个 LC 需要的最大输入位宽)。根据两个图可以得出结论：仔细地选择多项式和 LFSR 结构，可以从根本上做到节省。从图 7-25 可以看到，对于多步 LFSR 合成而言，图 7-21(b)的 LFSR 具有更少的扇入端数(也就是更小的 LC 输入位宽)，但是对于较长的多步 *k* 而言，表 7-12 的质数多项式的工作量可以更少。

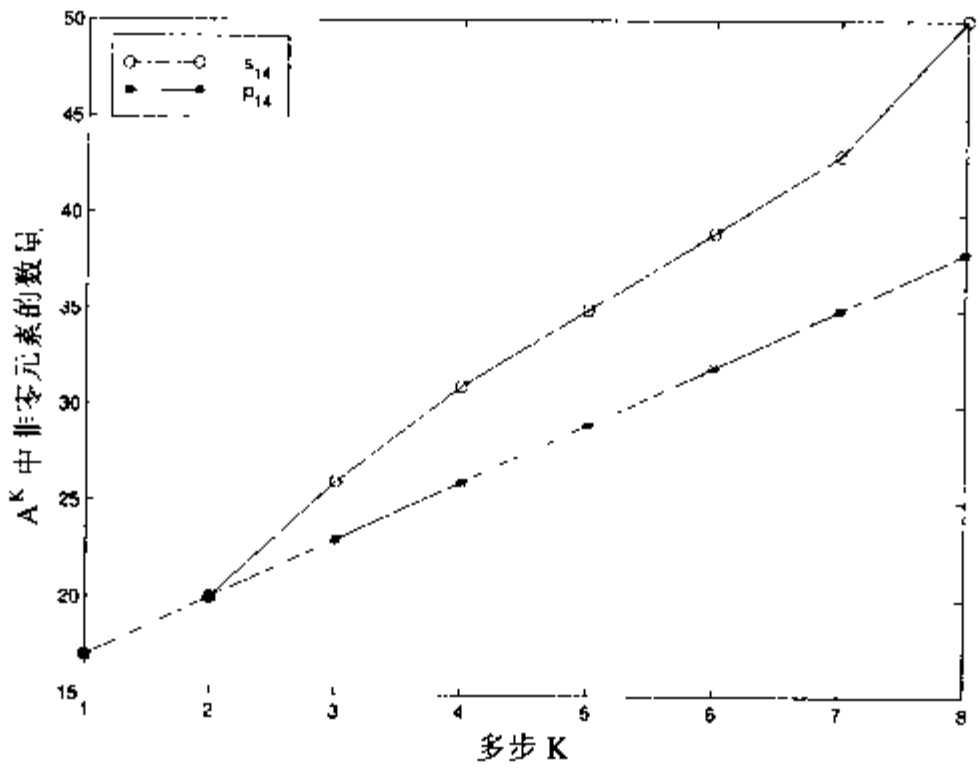


图 7-24　A_{14}^k 中 1 的数量

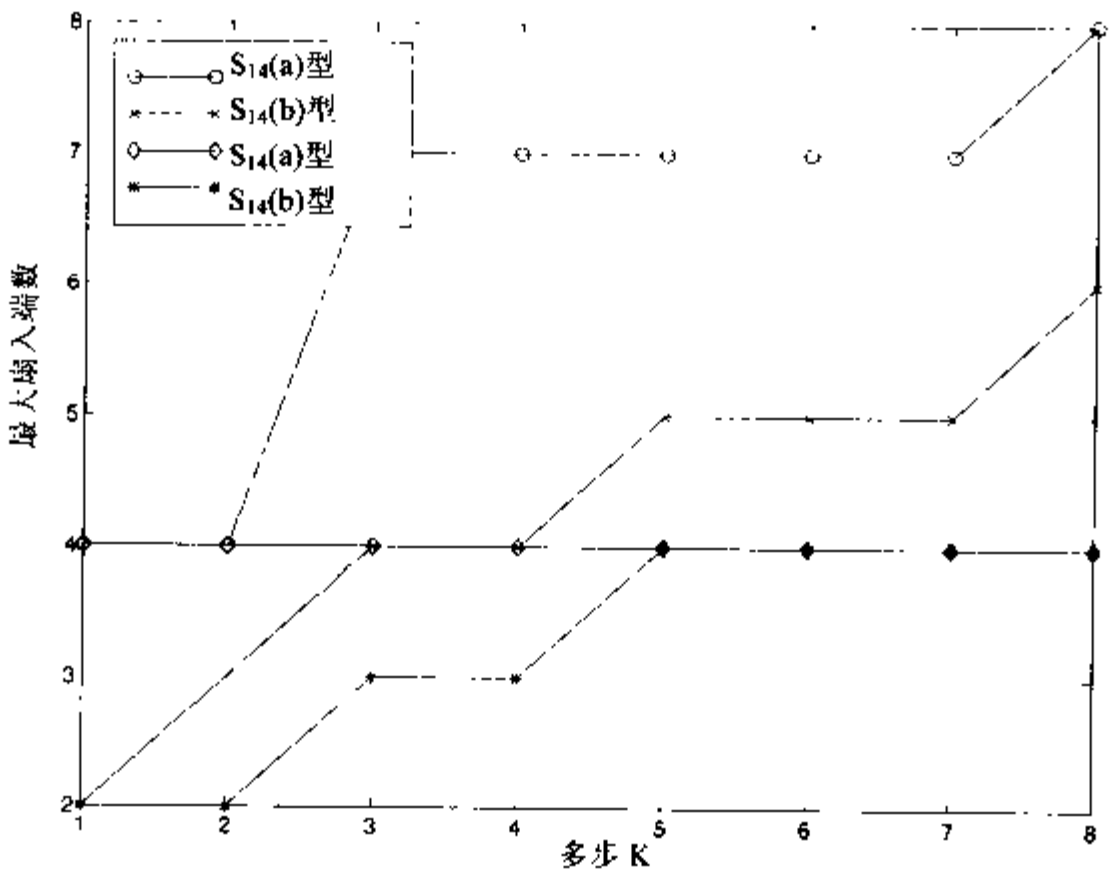


图 7-25　多步长度为 14 的 LFSR 的最大扇入端数

2. 组合 LFSR

如果将几个 LFSR 寄存器组合成一个密钥生成器就可以获得加密安全性性能方面的额外收益。目前存在几种线性和非线性的组合【167】、【166，150-173 页】。有实际意义实现的工作量和安全性是阈值的非线性组合。对于 3 种长度 L_1、L_2 和 L_3 的 3 种不同的 LFSR 的组合，其线性复杂性(是一个 LFSR 的等价长度)可以用 Berlekamp-Massey 算法合成，例如【166，141-9 页】，就有：

$$L_{ges} = L_1 \cdot L_2 + L_2 \cdot L_3 + L_1 \cdot L_3 \tag{7.54}$$

图 7-26 给出了这一结构的实现。

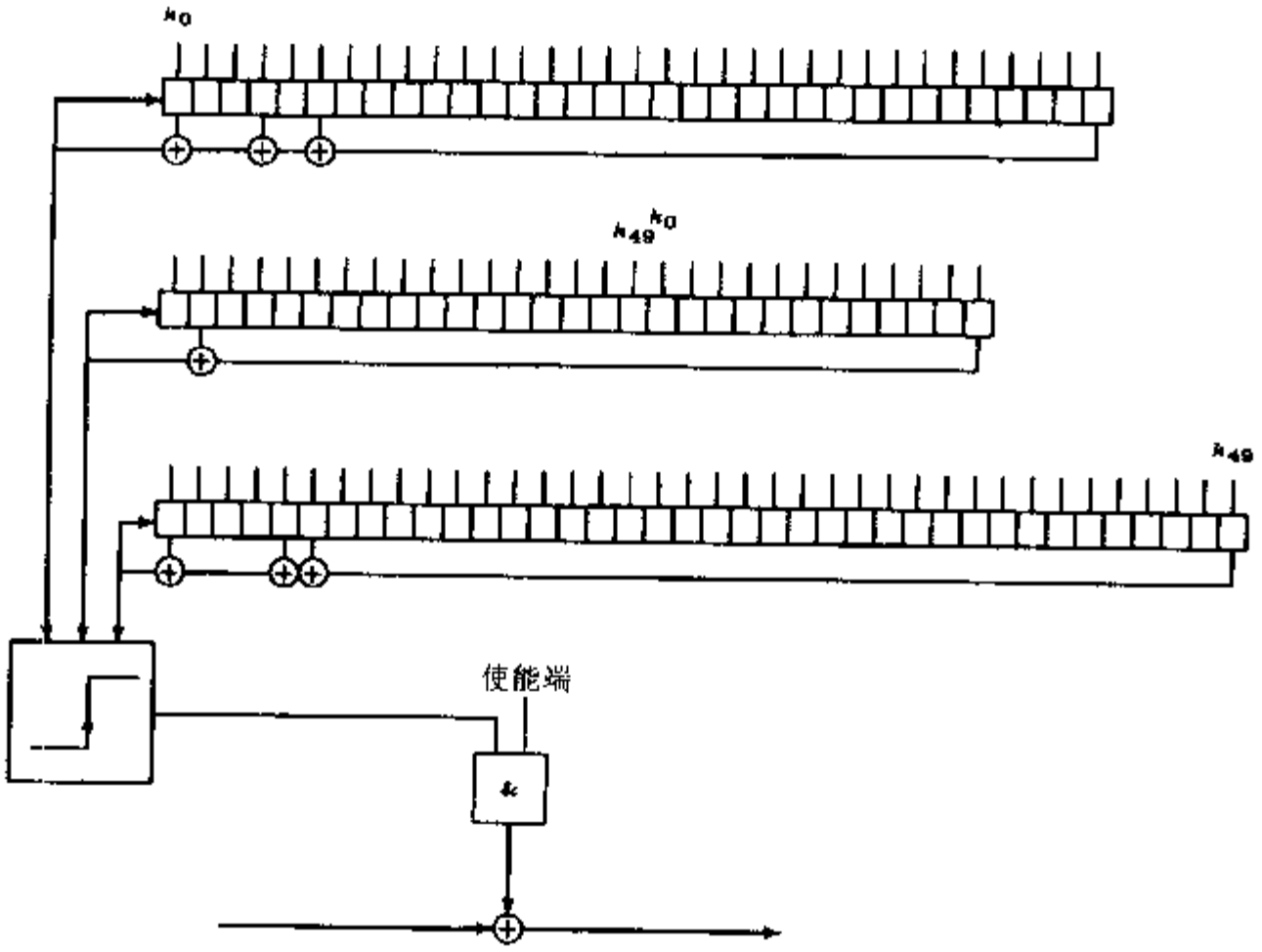


图 7-26　使用 3 个 LFSR 的数据流密码的实现

由于选定的分页格式中的密钥有 50 位，所以就要选择总长度为 2×50＝100 的寄存器，且 3 个反馈多项式分别是：

$$p_{33}(x) = x^{33} + x^6 + x^4 + x + 1 \tag{7.55}$$

$$p_{29}(x) = x^{29} + x^2 + 1 \tag{7.56}$$

$$p_{38}(x) = x^{38} + x^6 + x^5 + x + 1 \tag{7.57}$$

所有的多项式均是质数的。这就保证了所有 3 个移位寄存器序列的长度均是最大值。组合的线性复杂性如下：

$$L_1 = 33;\ L_2 = 29;\ L_3 = 38;$$

$$L_{\text{总}} = 33 \times 29 + 33 \times 28 + 29 \times 38 = 3313$$

在每次编码之后，密钥都会丢失，这就需要有额外的 50 个寄存器来保存密钥。这 50 个密钥需要用 2 次。表 7-13 给出了用 Xilinx 3K 系列 FPGA 所需要的硬件资源。

表 7-13　Xilinx 3K 系列 FPGA 的成本(按 CLB 计算)

函　数　组	CLB
50 位密钥寄存器	25
100 位移位寄存器	50
反馈	3
阈值	0.5
带有信息的 XOR	0.5
总计	79

3. 以数据加密标准为基础的算法

图 7-27 中介绍的**数据加密标准**(Data Encryption Standard，DES)主要应用在分组密码中。通过选择“输出反馈模式”(output feedback mode, OFB)，还可以在数据流密码中采用改进的 DES(参阅图 7-28)。通常，DES 的其他模式(ECB、CBC 或 CFB)不适用于通信系统，这是由于“雪崩效应”引起的：在传输中的一个位的差错在后来就会导致一个组中所有位的一半都出现差错。

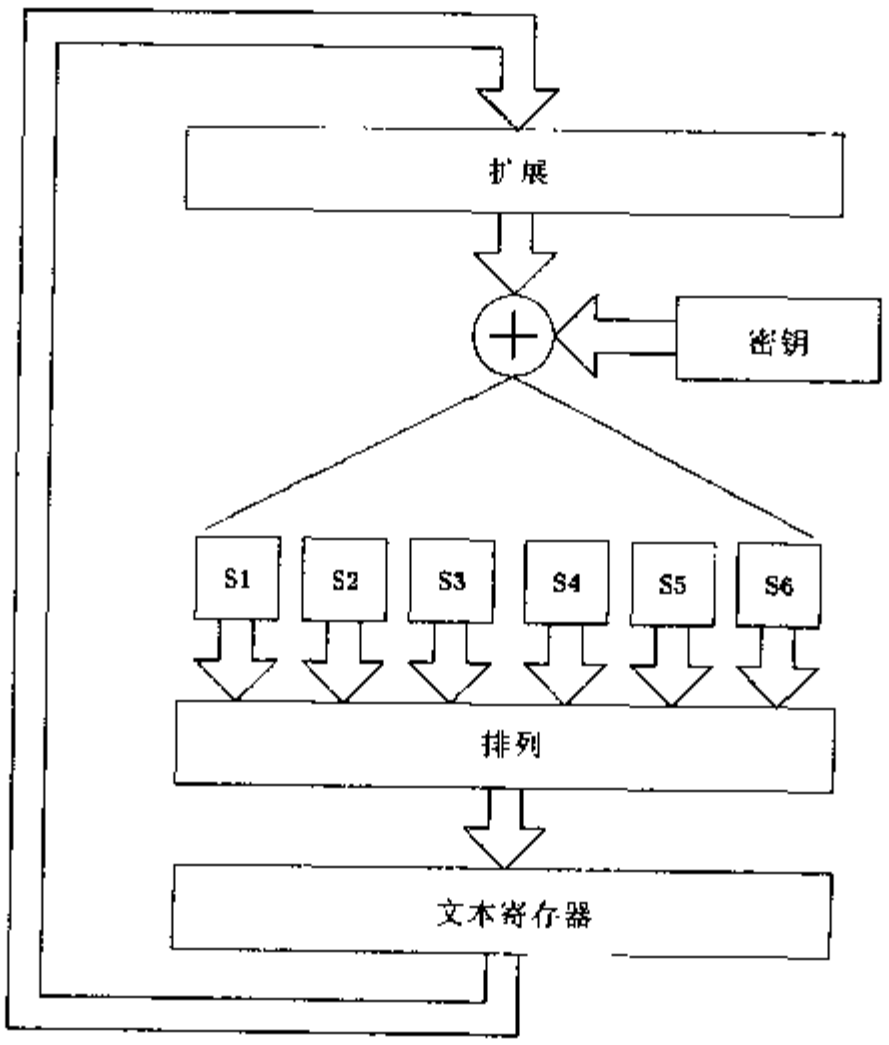


图 7-27　分组加密系统(DES)的状态机

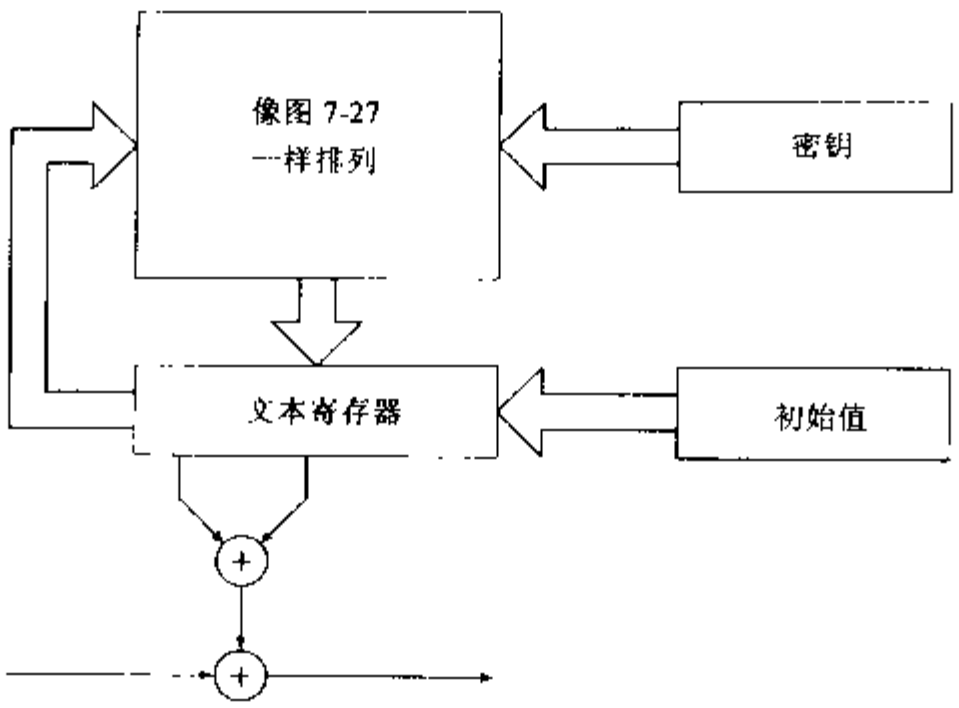


图 7-28　在 OFB 模式中用作数据流密码的分组密码

首先来复习一些 DES 算法的原理，然后讨论对于 FPGA 实现所作的一些适当的改进。

DES 包括一个将纯文本组转换成密码文本组的有限状态机，首先，将要代替的组下载到状态寄存器(32 位)中。接下来将其扩展(到 48 位)，与密钥(也是 48 位)组合，并代入 8 个 6→4 位宽的 S 方框。最后执行每一位的排列。这样的循环可以循环几次(如果需要的话，可以采用一个变更的密钥)。在 DES 中，密钥通常都是移动一到二位，所以在 16 个轮回之后，密钥就会返回到初始位置。由于 DES 可以看成是 Feistel 密码的迭代应用(请参阅图 7-29)，所以 S 方框必须是不能转置的。

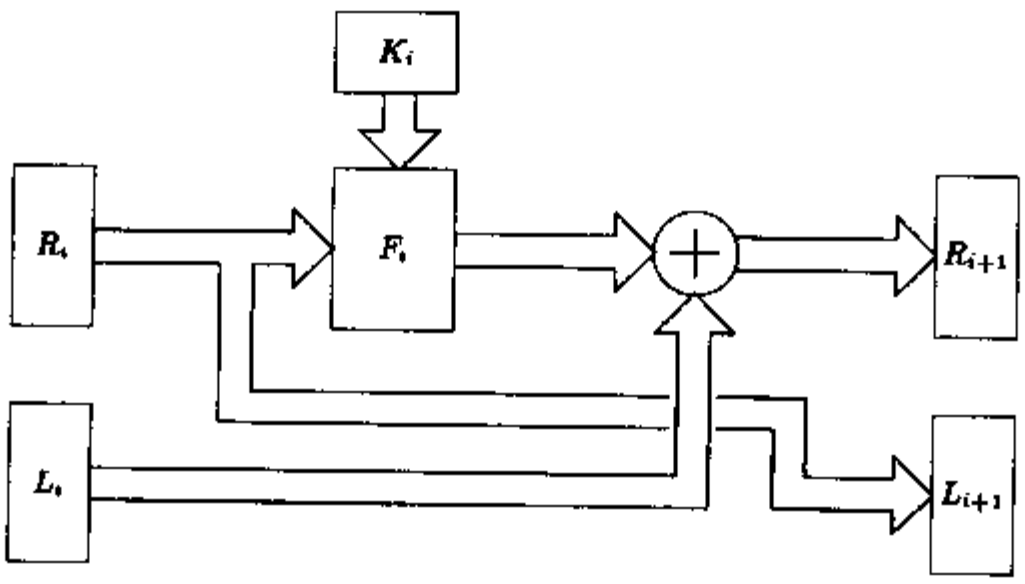


图 7-29　Feistel 网络的原理

为了简化 FPGA 实现，做一些改进还是非常有用的，例如：将状态寄存器的长度降低到 25 位，不采用扩展，而采用表 7-14 中列出的最终置换。

表 7-14　置　换　表

从第 n 位	0	1	2	3	4	5	6	7	8	9	10	11	12
到第 n 位	20	4	5	10	15	21	0	6	11	16	22	1	7
从第 n 位	13	14	15	16	17	18	19	20	21	22	23	24	
到第 n 位	12	17	23	2	8	13	18	24	3	9	14	19	

由于大多数 FPGA 仅有 4 到 5 个输入查询表(LUT)，所以表 7-15 给出了已设计的有 5 输入的 S 方框。

表 7-15 5 个新设计的代替方框(S 方框)

输入	方框 1	方框 2	方框 3	方框 4	方框 5	输入	方框 1	方框 2	方框 3	方框 4	方框 5
0	1E	F	14	19	6	10	19	B	1C	17	19
1	13	1	1D	14	E	11	16	1E	A	9	A
2	14	13	16	D	1A	12	7	18	1B	3	4
3	1	1F	B	4	3	13	1C	D	3	10	14
4	1A	19	5	1C	B	14	1D	5	19	A	13
5	1B	1C	E	1A	1E	15	5	14	D	16	11
6	E	12	8	1E	0	16	2	15	0	12	10
7	B	11	F	1	2	17	1F	9	2	1F	12
8	D	8	4	C	1D	18	F	3	15	B	5
9	10	7	C	F	C	19	11	10	6	2	F
A	3	1B	1E	1B	18	1A	C	6	7	6	8
B	0	0	13	1D	17	1B	18	17	12	18	16
C	4	1A	10	5	1	1C	9	4	1F	11	1C
B	6	C	1	15	15	1D	15	16	1A	8	7
E	A	1D	18	E	1B	1E	8	E	9	7	D
F	17	2	17	13	9	1F	12	A	11	0	1F

尽管我们的意图是仅使用 OFB 模式，表 7-16 中还是以可以转置的方式生成了 S 方框。这样，改进的 DES 还是可以用作标准的分组密码(Electronic Code Book，电子源码书)。

表 7-16 5 个代替方框的独立矩阵(理想值是 16)

方框 1

20	12	20	20	20
12	20	12	16	16
12	16	16	12	8
16	16	20	12	16
20	16	20	12	12

方框 2

20	16	20	12	20
20	20	20	16	16
12	20	20	16	8
16	24	12	16	12
16	20	16	20	20

方框 3

20	12	16	16	16
16	20	16	16	16
16	16	20	12	12
16	8	12	16	20
20	12	12	20	12

方框 4

20	16	20	20	16
12	16	12	16	20
20	16	16	20	16
20	16	16	20	24
16	12	28	20	16

方框 5

12	20	8	12	20
20	12	16	24	20
16	12	12	20	16
16	20	16	20	12
12	16	16	12	24

对于S方框的一个合理测试就是独立矩阵。这种矩阵为每个输入/输出组合给出了如果输入位发生变化，输出位所改变的概率。雪崩效应的理想概率是1/2。表7-16给出了5个新的S方框的独立矩阵。取代概率的是，该表给出了出现的绝对数值。由于对于每个S方框而言，都有$2^5=32$个可能的输入向量，所以理想值是16。可以用一个随机生成器生成S方框，一些值与理想值16较大有差别的原因就在于所需要的反转不一样。

表7-17总结了基于DES的算法的硬件工作量。

表7-17　基于改进的DES的算法的硬件工作量

函　数　组	CLB
25位密钥寄存器	12.5
25位附加位	12.5
25位状态寄存器	12.5
5个S方框5→5	25
排列	0
25位初始化向量	12.5
多路复用：初始化向量/S方框	12.5
带有信息的XOR	1
总计	87.5

4. 加密技术的性能比较

接下来要讨论LFSR和基于DSE算法的加密技术的性能分析。已经定义了几种安全测试，下面的比较给出了两个最明显的差别(其他的差别在两者之间表现得不太明显)。两个测试均生成100个随机密钥。

- 用不同的密钥对所生成的序列进行分析。在每个随机密钥中变化其中的一位，并记录纯文本中位变化的数量。平均起来应该有50%左右的位颠倒了(雪崩效应)。
- 与上面的测试相类似，但是这次是在输出序列中分析变化的数量，这主要取决于密钥所发生变化的位置。这次发生变化的位又是50%左右。

两个测试均采用了64位长的纯文本(再请参阅图7-20)。纯文本是任意的。对于第一种测试，累加了每个密钥位置的所有变化。对于第2种测试，累加了依赖于输出序列中位置的所有变化。两种测试均执行了100个随机密钥。第1种测试的理想值是$64\times0.5\times100=3,200$，第2种测试的最优值是$50\times0.5\times100=2,500$。图7-30和7-31分别给出了结果。两者都清楚地显示了DES-OFB模式对密钥的变化要比对带有3个LFSR的模式敏感。第2种测试的结论表明，当密钥的变化影响输出序列前，SR模式需要大约32步。而对于DES-OFB模式而言，只是最初的4次样本与理想值2500之间有明显的差别。

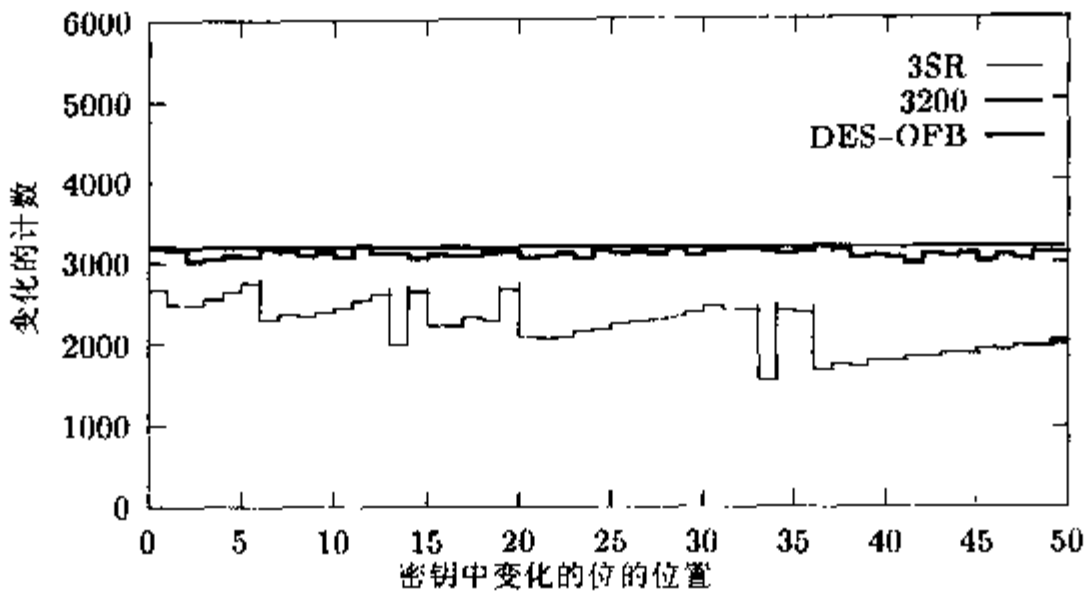


图 7-30 第 1 种测试的结果

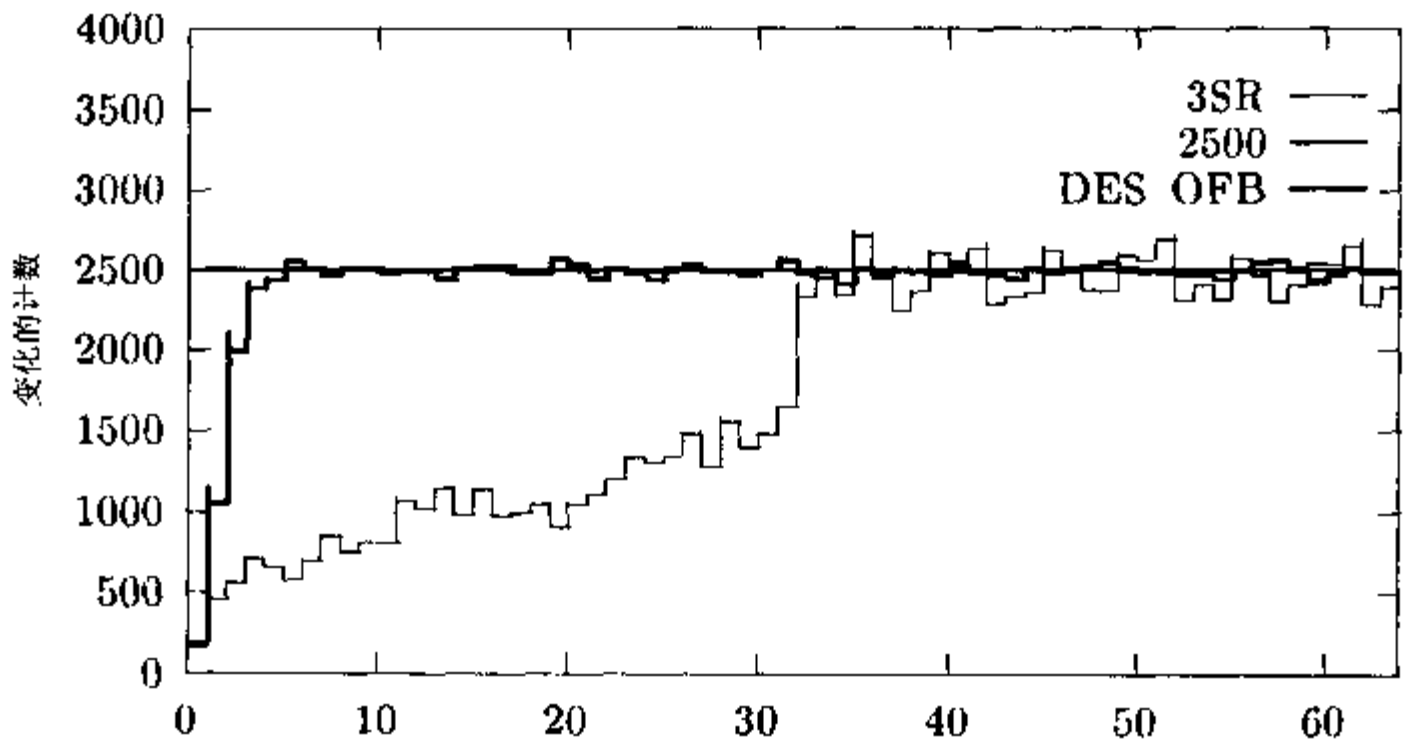


图 7-31 第 2 种测试的结果

鉴于良好的测试结果，优先选择 DES-OFB 模式而不是 LFSR 模式。

5. 加密安全的注意事项

通常很难得出结论说一个加密系统是否安全。除了密钥可以被窃取之外，快速解密算法目前还不知道的事实并不能够证明就没有这样的快速算法。还有就是采用强大的计算机进行“强力攻击”和/或并行攻击的问题。56 位密钥的 DES 算法就是一个典型的例子，这在过去很多年中，它一直被当作标准使用，但是最终在 1997 年还是被宣布是不安全的。DES 首先是被互联网上的一个志愿者计算机主的网络在 39 天内给破译了。后来在 1997 年 7 月 EEF(Electronic Frontier Foundation, 电子尖端基础)完成了破译机器的设计。目前已经被收录在一本书中[181]，其中包括所有的图和软件源代码，可以从 http://www.eff.org 网站下载该内容。这个破译机器执行一次穷举密钥搜索，就可以在不到 5 天的时间内破译所有的 56 位密钥。它全部是用定制芯片构成的，每个芯片都有 24 个破译单元。29 个板中的每个板都包括 64 个“深度破译”芯片，也就是说总共使用了 1,856 个芯片或是 44,544 个单元。该系统价值 250,000 美元。当 DES 在 1977 年推出该系统时估计其价值是二千万美金，也就是相当于今天的四千万美金。这也是摩尔律的

一个很好的近似，其中摩尔律的内容是：每 18 个月微处理器的规模或速度会提高一倍，而其价格却降低 1/2。从 1977 年到 1998 年，这样的机器价格降低了 $40\times10^6/2^{22/1.5}\approx1500$ 美金，也就是说，该价格在现在能够制造一个 DES 破译器(已经为 EFF 所证实)。

这样 56 位的 DES 就不再安全了，但是现在通用的均是采用三重 DES，如图 7-32 所示，或是 128 位的密钥系统。表 7-18 表明，这些系统似乎在今后的几年中还是很安全的，例如：EFF 破译者现在还需要大约 5×2^{112} 天或是 7×10^{31} 年来破译三重 DES。

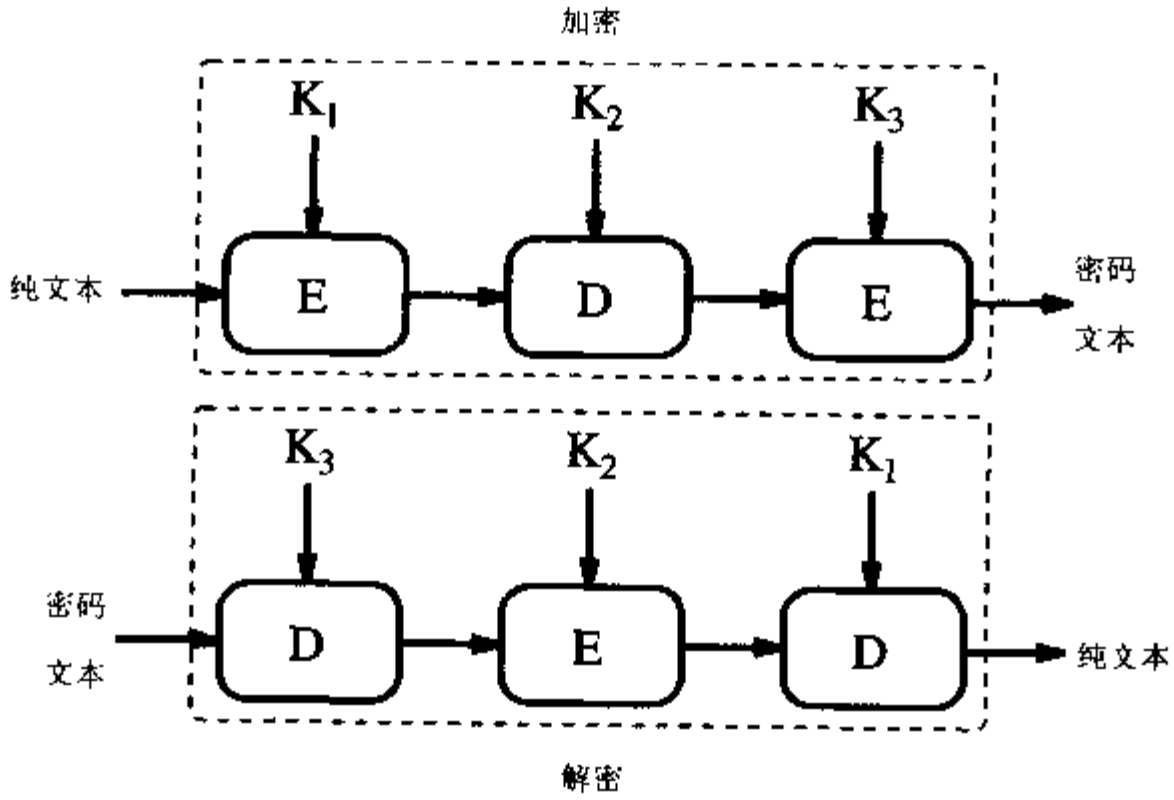


图 7-32　三重 DES (K_i=密钥；E=单个加密；D=单个解密)

表 7-18　加密算法【182】

算　法	密钥长度(位)	数学操作符/原理	对 称 性	开发者(年)
DES	56	XOR、固定 S 方框	s	IBM (1977)
三重 DES	122-168	XOR、固定 S 方框	s	无
RSA	变量	质因数	a	Rivest /Shamir /Adleman (1977)
IDEA	128	XOR、加、乘	s	Massey/Lai (1991)
Blowfish	<448	XOR、加、固定 S 方框	s	Schneider (1993)
RC5	<2048	XOR、加、旋转	s	Rivest (1994)
CAST-128	40-128	XOR、旋转、S 方框	s	Adams/Tavares (1997)

表 7-18 的第一列是通用的加密算法的缩略词。第二列和第三列包括算法的典型参数。对称算法(在第 4 列中用 s 指代)通常都基于 Feistel 算法，而非对称算法(在第 4 列中用 a 指代)可以用在公共/私有密钥系统中。最后一列给出了开发者的名称和算法第一次公布的年代。

7.3　调制和解调

长期以来，通信系统设计的一个目标就是实现全数字接收器，只包括一个天线和一个完全可编程电路的数字滤波器、用作差错校正的解调器和/或译码器以及单片可编程芯片上的加密技术。今天，FPGA 门的数量已经达到百万以上，这也就使得全数字接收器成为了现实。正像 W. Carter 所预言的那样："很显然，FPGA 将会是通信系统进入 21 世纪的关键技术"[183]。在本节中将开发用 FPGA 设计和实现的通信系统。

7.3.1　基本的调制概念

简单的通信系统在载波频率(表示成 f_0)上传输和接收信息。这种载波是用幅值、频率或是相位来调制的，具体情况视传输的信号 $x(t)$而定。图 7-33 给出了二进制传输的调制信号。对于二进制传输而言，调制可以称为幅移调制(amplitude shift keying, ASK)、相移调制(phase shift keying, PSK)和频移调制(frequency shift keying, FSK)。

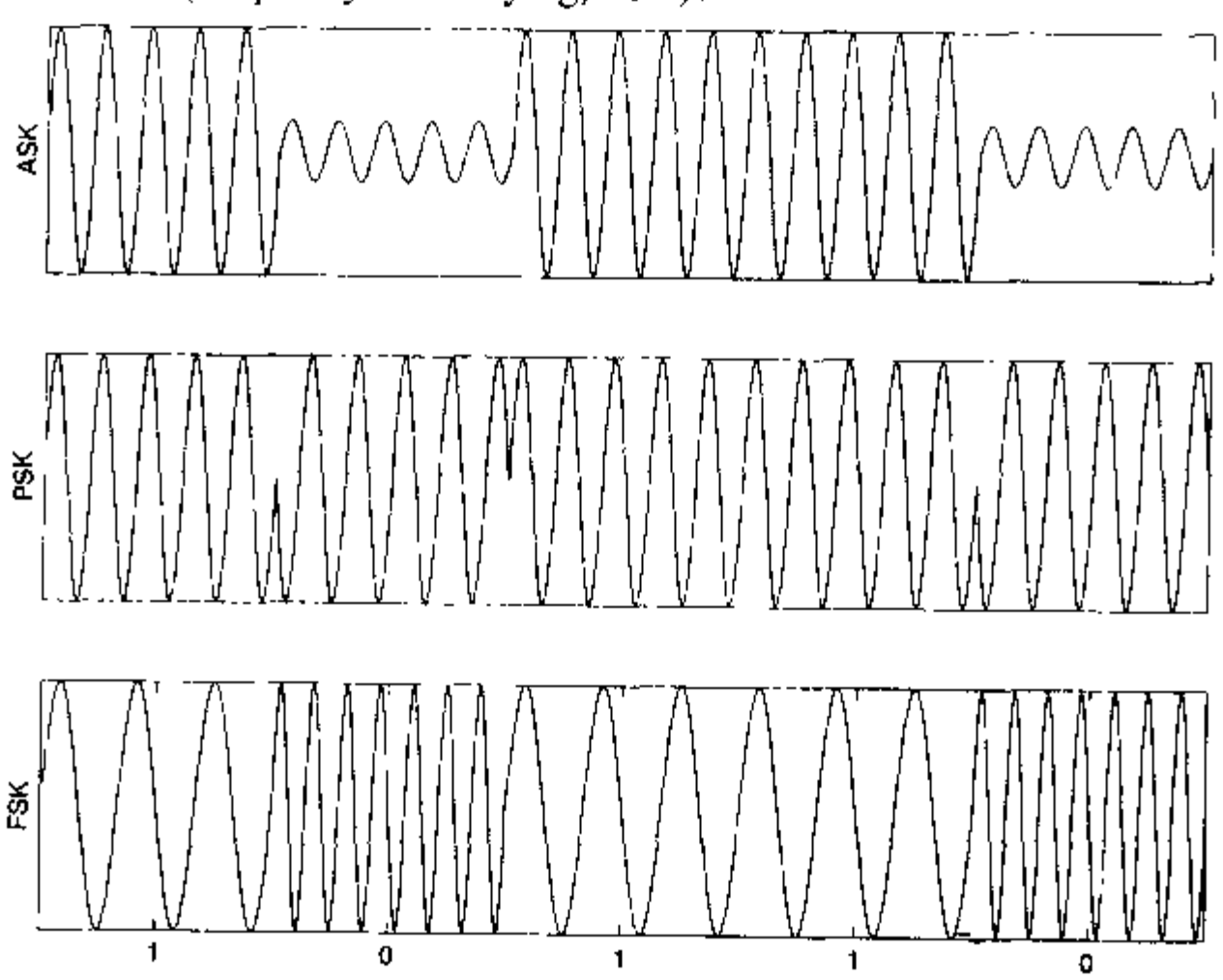


图 7-33　ASK、PSK 和 FSK 调制

通常用一个旋转箭头在水平轴上的投影可以更加有效地描述一个(实)调制信号，根据：

$$s(t) = \Re\{A(t)e^{\mathrm{j}2\pi f_0 t - \Delta\phi(t)+\phi_0)}\} = A(t)\cos(2\pi f_0 t + \Delta\phi(t) + \phi_0) \tag{7.58}$$

其中 ϕ_0 是一个(随机)相位偏移，$A(t)$描述的是幅值包络线，$\Delta\phi(t)$描述的是频率或相位调制的部分，如图 7-34 所示。从(7.58)式可以看到，AM 和 PM/FM 信号可以单独用来传输不同的信号。

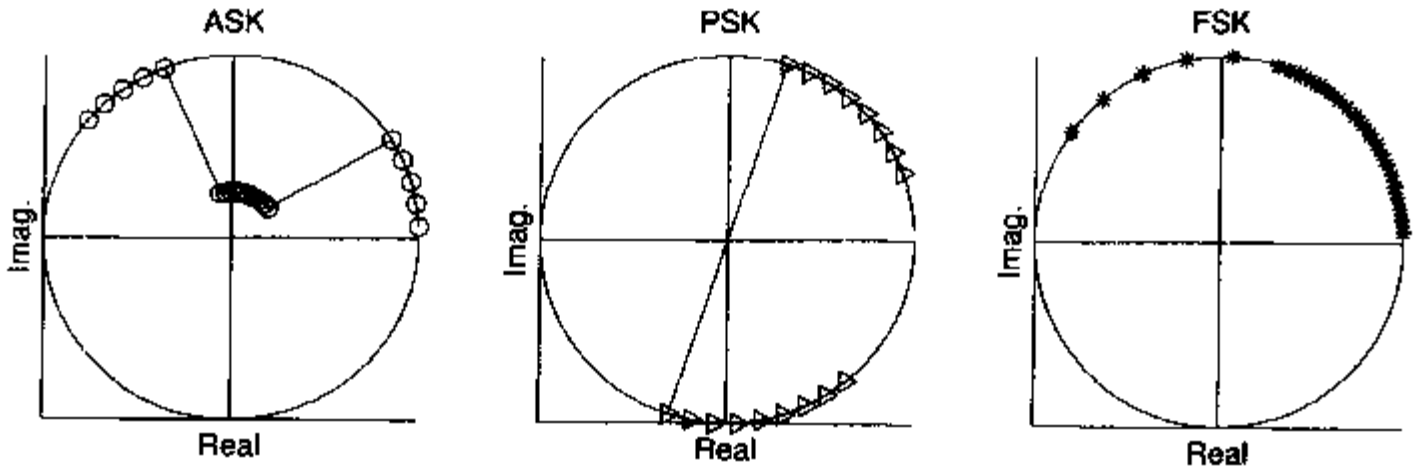


图 7-34　复平面中的调制

实现通用调制器的一个有效解决方案(不需要大规模的表)就是第 2 章中讨论过的 CORDIC 算法。CORDIC 算法用在旋转模式中，也就是一个$(R, \theta)\rightarrow(X, Y)$的坐标旋转器。图 7-35 给出了 AM、PM 和 FM 信号的完整调制器。

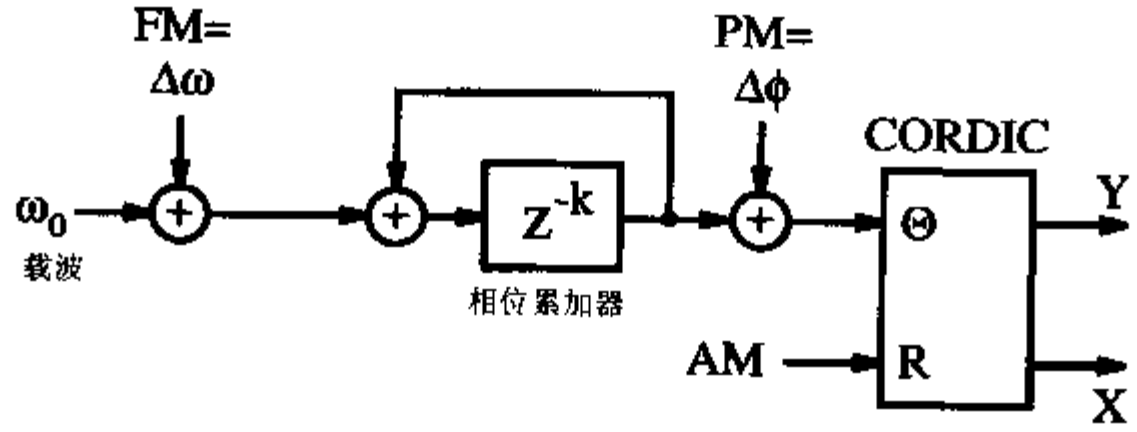


图 7-35　采用 CORDIC 的通用调制器

要实现幅度调制，需要将信号 $A(t)$直接与 CORDIC 的半径 R 输入连接。通常，旋转模式中的 CORDIC 算法有随着半径线性增加的趋势。这与放大器的增益变化相符，并且不需要在 AM 方案中加以考虑。倘若不希望线性增加半径(因子是 1.6468，参阅表 2-1)，可以使用 1/1.6468 的常系数乘法器对输入或输出进行刻度。

传输信号的相位 $\theta=2\pi f_0 t+\Delta\phi(t)$也必须计算。要生成恒定的载波频率，还要生成随 $2\pi f_0 t$ 线性增加的相位信号，这可以用一个累加器实现。如果需要生成 FM 信号，可以用Δf代替 f_0，或是再用一个累加器来计算 $2\pi\Delta f t$，然后将两个累加器的结果相加。对于 PM 信号，需要在信号的相位上增加一个常数偏移(不随时间增加)。这些相位信号相加起来作为 CORDIC 处理器的角输入 z 或 θ。Y 寄存器在迭代的初始时刻被设置成 0。

接下来的例子说明了 CORDIC 调制器的一个完整的流水线形式。

例 7.14：采用 CORDIC 的通用调制器

根据图 7-35，AM、PM 和 FM 信号的通用调制器可以用下面 CORDIC 部分的 VHDL 代码[4]来设计。

```
PACKAGE nine_bit_int IS      -- User defined types
```

注 4：这一例子相应的 Verilog 代码文件 ammod.v 可以在附录 A 中找到。

```
   SUBTYPE NINE_BIT IS INTEGER RANGE  - 256 TO 255;
   TYPE ARRAY_NINE_BIT IS ARRAY (0 TO 3) OF NINE_BIT;
END nine_bit_int;

LIBRARY work;
USE work.nine_bit_int.ALL;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY ammod IS                              ------> Interface
        PORT (clk                 : IN   STD_LOGIC;
              r_in , phi_in       : IN   NINE_BIT;
              x_out, y_out, eps   : OUT NINE_BIT);
END ammod;

ARCHITECTURE flex OF ammod IS

BEGIN
PROCESS                                      ------> Behavioral Style
  VARIABLE x, y, z : ARRAY_NINE_BIT; -- Tapped delay line
BEGIN
WAIT UNTIL clk = '1';       -- Compute last value first
  x_out <= x(3);            -- in sequential statements !!
  eps   <= z(3);
  y_out <= y(3);

  IF z(2) > 0 THEN                           -- Rotate 14 degrees
    x(3) := x(2) - y(2) /4;
    y(3) := y(2) + x(2) /4;
    z(3) := z(2) - 14;
  ELSE
    x(3) := x(2) + y(2) /4;
    y(3) := y(2) - x(2) /4;
    z(3) := z(2) + 14;
  END IF;

  IF z(1) > 0 THEN                           -- Rotate 26 degrees
    x(2) := x(1) - y(1) /2;
    y(2) := y(1) + x(1) /2;
    z(2) := z(1) - 26;
  ELSE
    x(2) := x(1) + y(1) /2;
```

```
    y(2) := y(1) - x(1) /2;
    z(2) := z(1) + 26;
END IF;

IF z(0) > 0 THEN                    -- Rotate   45 degrees
    x(1) := x(0) - y(0);
    y(1) := y(0) + x(0);
    z(1) := z(0) - 45;
ELSE
    x(1) := x(0) + y(0);
    y(1) := y(0) - x(0);
    z(1) := z(0) + 45;
END IF;

IF phi_in > 90      THEN            -- Test for |phi_in| > 90
    x(0) := 0;                      -- Rotate 90 degrees
    y(0) := r_in;                   -- Input in register 0
    z(0) := phi_in - 90;
ELSIF phi_in < - 90 THEN
    x(0) := 0;
    y(0) := - r_in;
    z(0) := phi_in + 90;
  ELSE
    x(0) := r_in;
    y(0) := 0;
    z(0) := phi_in;
  END IF;
END PROCESS;

END flex;
```

图 7-36 给出了 AM 信号的仿真。注意：Altera 仿真不生成有符号数据，而是生成无符号二进制数据(其中负值有一个偏移量 512)。可以看到 4 个步骤的流水线延迟，而值 100 被一个 1.6 的因子放大了。半径 r_in 从 100 到 25 的切换令最大值 x_out 从 163 降到了 42。CORDIC 调制器运行的速度为 40.65MHz，使用了 279 个 LC。

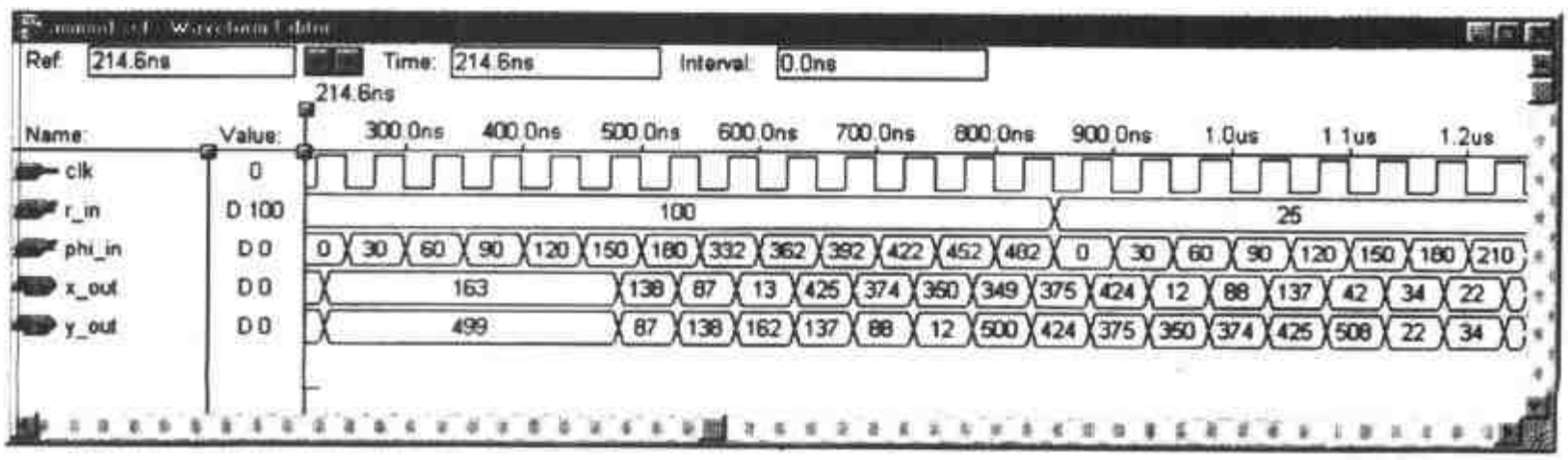


图 7-36　采用 CORDIC 算法的 AM 调制器的仿真

解调可以是**相干**的也可以是**非相干**的。相干接收器必须要再现未知的载波相位ϕ_0，而非相干接收器则不需要如此。如果接收器采用的是中频 (intermediate frequency, IF)带，这种接收器就称作超外差接收器或是双变频超外差(双 IF 频带)接收器。IF 接收器也经常称作外差接收器。如果不使用 IF 级，就是**零 IF 或零差接收器**。图 7-37 给出了不同种类接收器的一个系统的综述。有些接收器仅能采用一种调制方案，而其他接收器则可以采用多种模式(例如 AM、PM 和 FM)。后者就称为通用接收器。接下来首先讨论非相干接收器，然后再讨论相干接收器。

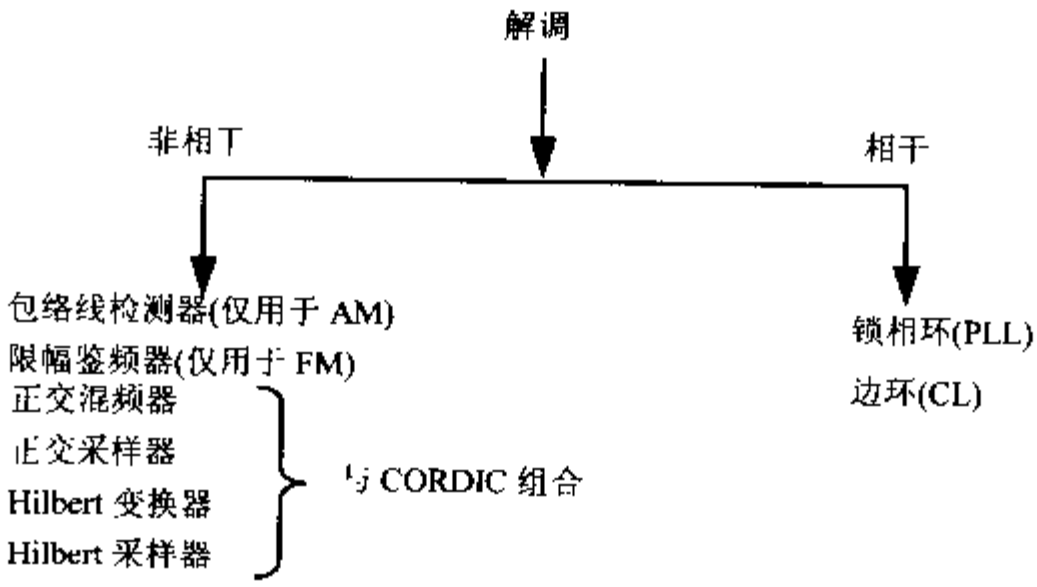


图 7-37　相干和非相干解调方案

为了只选择有用的信号成份，所有的接收器均采用集约型滤波器(如在第 3 章和第 4 章中讨论过的)。此外对于外差接收器，还需要用滤波器对镜像频率进行抑制，镜像频率来源于频移：

$$s(t)\cdot\cos(2\pi f_m t)\leftrightarrow S(f+f_m)+S(f-f_m) \tag{7.59}$$

7.3.2　非相干解调

在非相干调制方案中假定接收器已经知道精确的载波频率，但是初始相位ϕ_0未知。

如果用数字或模拟滤波将信号成份成功地选择出来，问题就出现了：是需要一种调制模式(例如：AM 或 FM)还是需要通用调制器？非相干 AM 调制器与一个全或半检波器以及一个额外的低通滤波器一样简单。如果仅是 AM 或 FM 解调的话，**限幅器/鉴相器**类型的解调器就是一种有效的实现。这种解调器构造了一个输入信号的阈值，并且把值限制在±1，然后简单地测量两个过零区间的距离。这些接收器可以很容易用 FPGA 来实现，但是经常会在相位信号中产生 2π 的跳跃(称作“咯吧声”，clicks[184, 185])，另外，还有其他的具有更好性能的解调器。

我们主要讨论采用同相和正交分量的通用接收器。这种类型的接收器只是简单地将与(7.58)相关的调制方案倒置。在第一步中，要从接收到的余弦信号中计算与发送器正弦成份(与载波成正交，名称是 Q 相)同相的成份。这样就用 I 和 Q 相重构复平面中的箭头(以载波频率旋转)。此时，这一调制恰好是图 7-35 电路的倒置。可以在向量模式中运用 CORDIC 算法，也就是用 $I=X$ 和 $Q=Y$ 进行坐标变换 $X,Y\rightarrow R,\theta$。这样输出 R 直接与 AM 部分成比例，PM/FM 部分可以根据 θ 信号(也就是 Z 寄存器)重构。

调制的不同部分是 I/Q 的生成，通常要用到两种理论：正交方案和 Hilbert 变换。

在正交方案中，输入信号与两个混频器信号 $2\cos(2\pi f_m t)$和 $-j2\sin(2\pi f_m t)$相乘。如果 IF 频带

$f_{IF} = f_0 - f_m$ 中的信号是由滤波器选择的，这些信号的复数和就是复平面上旋转箭头的重构。图 7-38 给出了这种方案，图 7-39 给出了 *I/Q* 生成的一个例子。从图 7-39 可以看到，最终的信号没有负的频谱成份。这就是典型的非相干接收器类型，这些信号就称作分析信号。

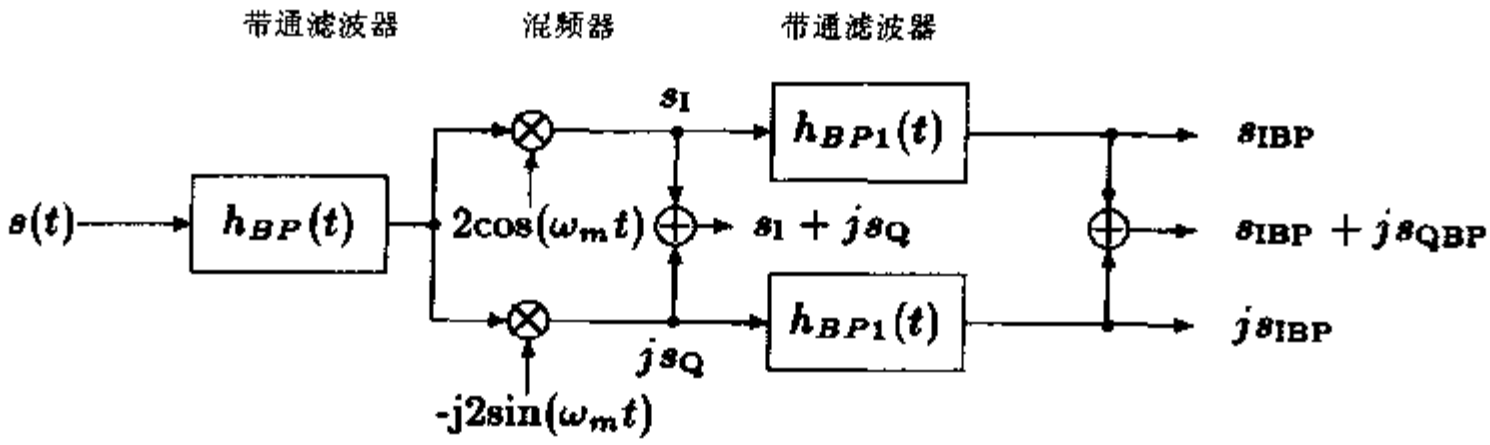


图 7-38　采用正交方案的 *I/Q* 相位的生成

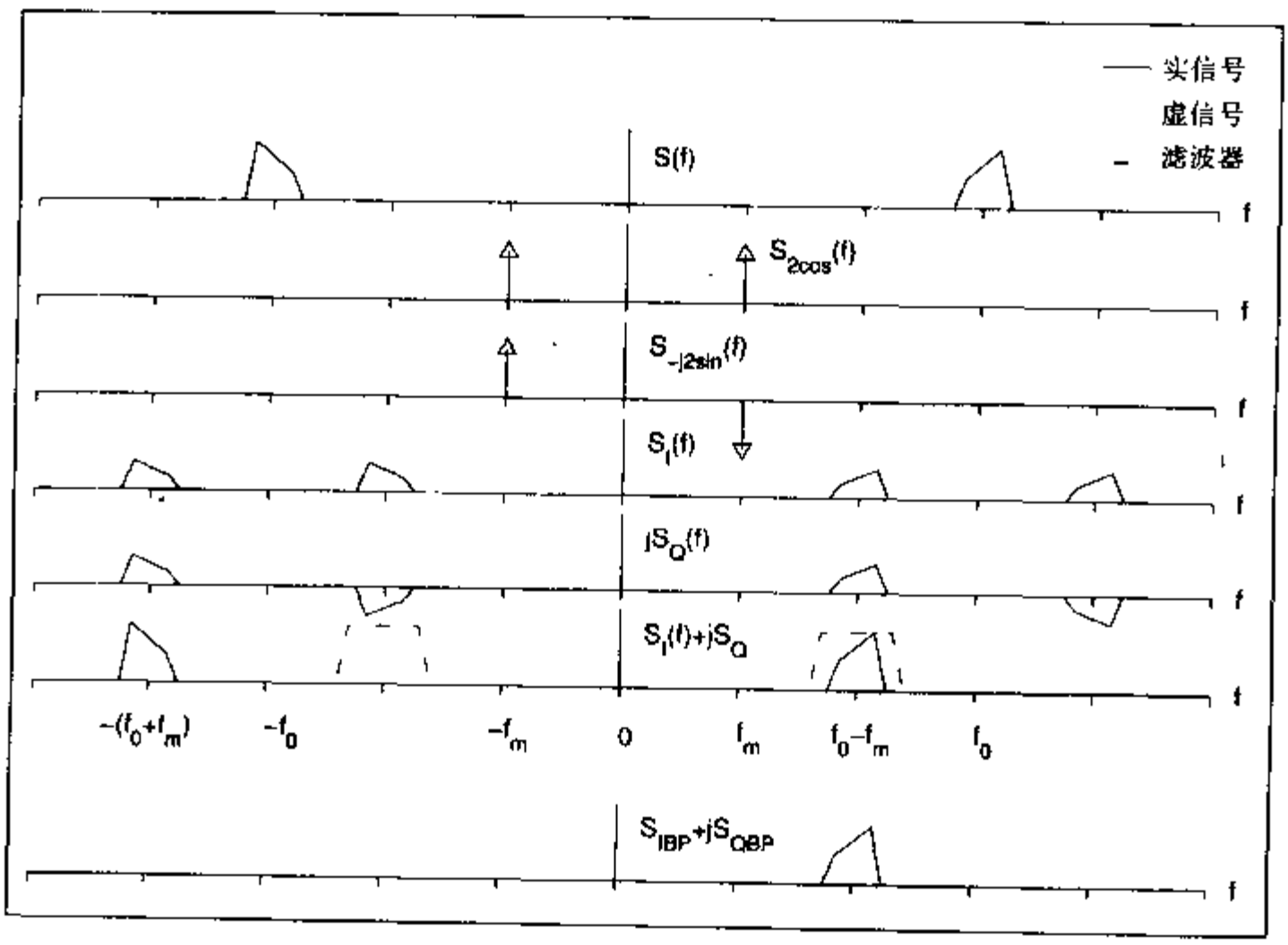


图 7-39　*I/Q* 生成的频谱示例

要减少滤波器的工作量，最好是采用接近零的 IF 频率。这在模拟方案中(特别是对于 AM)，经常会带来新的问题，放大器会漂移到饱和区。但是对于全数字接收器而言，就可以构造零差或零 IF 接收器。这样带通滤波器就简化成了低通滤波器。Hogenauer 的 CIC 滤波器(参阅第 5 章)就是这些高抽取滤波器的有效实现。图 7-40 给出了相应的频谱。实输入信号以 2π 的速率采样。然后这一信号分别与余弦信号 $S_{2\cos}(e^{j\omega})$和正弦信号 $S_{-j2\sin}(e^{j\omega})$相乘，就生成了同相成份 $S_I(e^{j\omega})$和正交成份 $jS_Q(e^{j\omega})$。现在这两个信号组合成复数分析信号 S_I+jS_Q。在最后的低通滤波之后，可以采用一个采样频率的抽取。

这样，LF 的完整全数字零 IF 接收器就用 FPGA 技术构造起来了[186]。

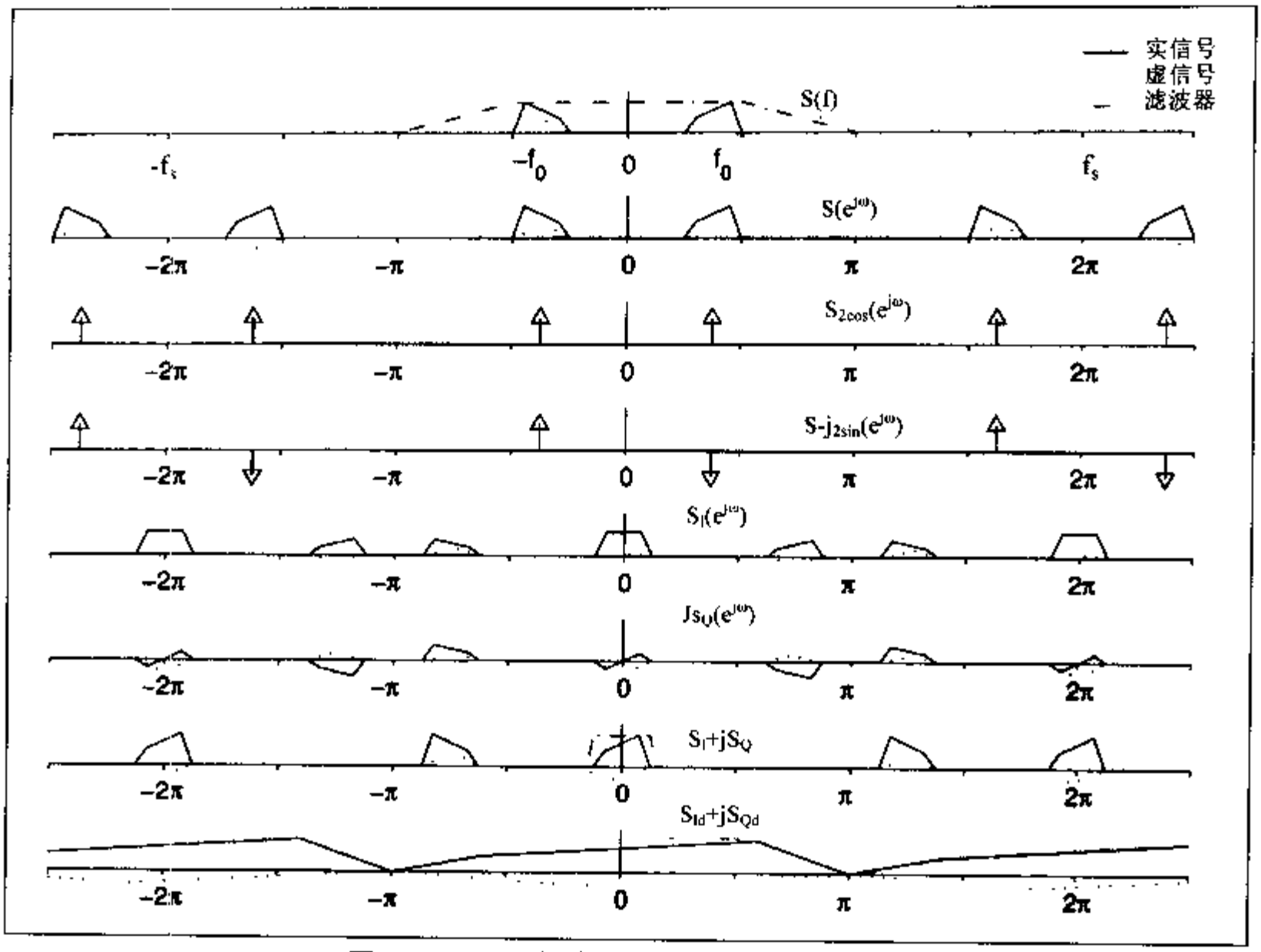


图 7-40 零 IF 接收器的频谱，采样频率是 2π

例 7.15：零 IF 接收器

接收器有一个天线、一个可编程增益调节器(programmable gain adjust, AGC)和一个 7 阶 Cauer 低通滤波器，后面还跟有 一个 8 位音频 A/D 转换器。接收器对输入范围从 50KHz 到 150KHz 的信号采用 8 倍过采样(0.4-1.2MHz)。正交乘法器是 8×8 位阵列乘法器。两级 CIC 滤波器的设计是 24 位和 19 位积分器精度，梳状部分是 17 位和 16 位精度。最终的采样比率就降低到 64。整个设计可以放置在一个单片 XC 3090 Xilinx FPGA 中。下面的表就给出了单个单元的工作量。

设 计 部 分	CLB
带有 sin/cos 表的混频器	74
两个 CIC 滤波器	168
状态机和 PDSP 接口	18
频率合成器	32
总计	292

对于可调的频率合成器，可以参照模拟的锁相环，利用一个累加器来构造[4]。图 7-41 给出了这种类型的频率合成器，图 7-42 是该合成器的实测性能。累加器合成器能够严格受时钟控制，其主要原因是只需要溢出。这就要采用一个逐位进位储存加法器。用作 PLL 参照的累加器能够生成 $F_{out}=M_2F'_{in}=M_1M_2F_{in}/2^N$。

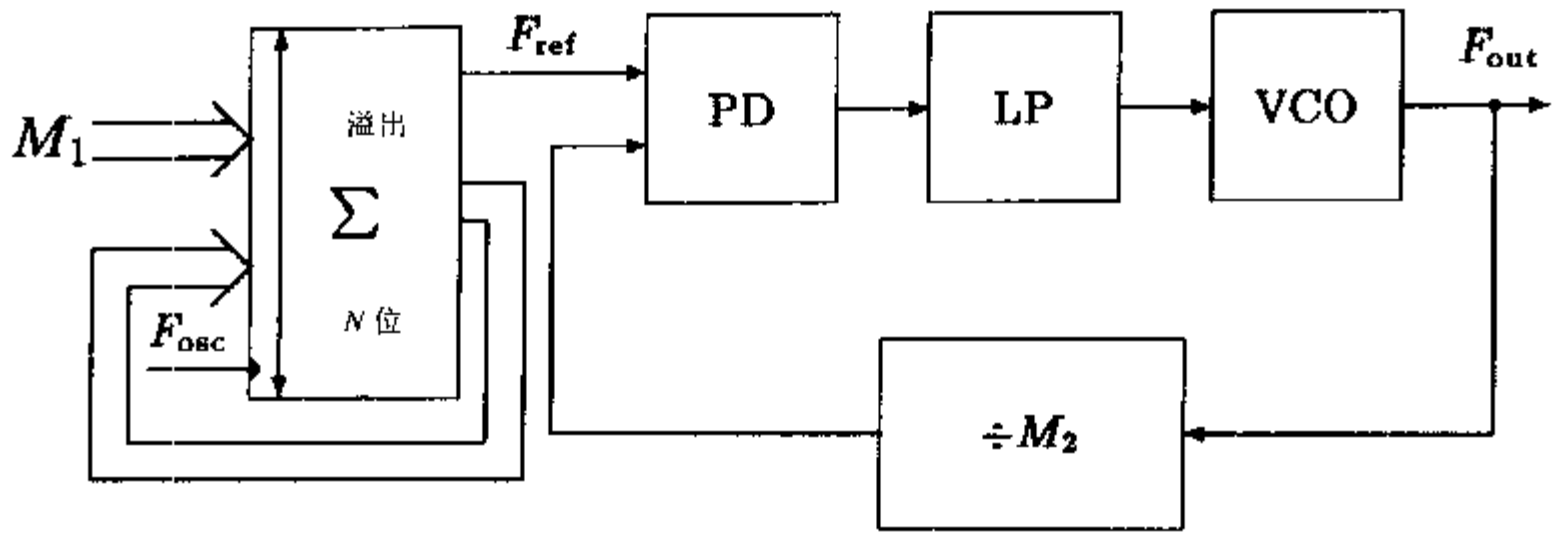
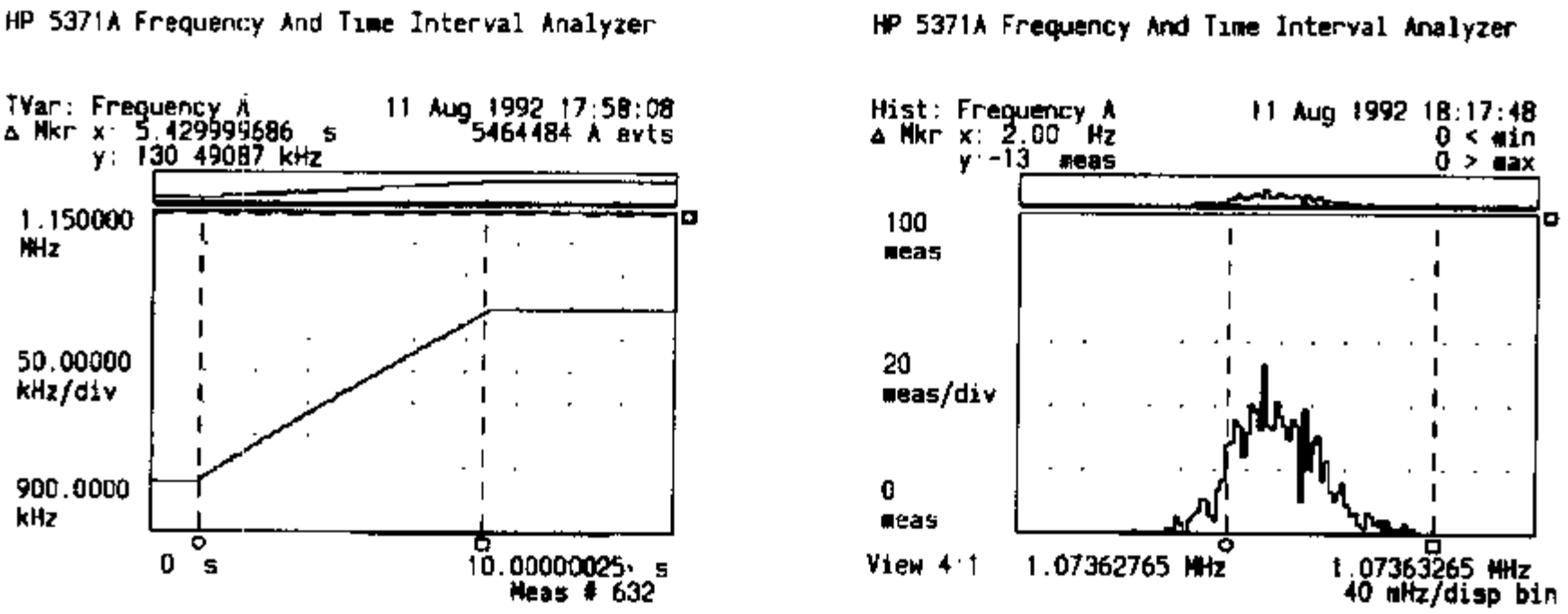


图 7-41　以累加器为参照的 PLL



(a)　　(b)

图 7-42　以累加器为参照的 PLL 合成器 (a) F_{out} 从 900KHz 切换到 1.2MHz 时合成器的运行状态

(b) 频率误差小于 2Hz 的柱状图

Hilbert 变换器方案的事实依据就是正弦信号可以通过将余弦信号相位延迟 90°来计算。如果用滤波器来生成这样的 Hilbert 变换器，如图 7-43 所示，滤波器的振幅必须是 1，而且相位对所有频率必须都是 90°。可以用傅立叶变换的定义找到其脉冲响应和传递函数，也就是：

$$h(t)=\frac{1}{\pi t}\leftrightarrow H(j\omega)=-j\gamma(\omega)=\begin{cases} j & -\infty<\omega<0 \\ -j & 0<\omega<\infty \end{cases} \tag{7.60}$$

其中 $\omega)=-1\forall\omega<0$ 和 $\omega)=-1\forall\omega<0$ 是作为符号函数。Hibert 滤波器只能够用 FIR 滤波器近似，最终的系数已在有关文献中给出(例如：请参阅【187】、【188】、【120，168-174 页】或【65，681 页】)。

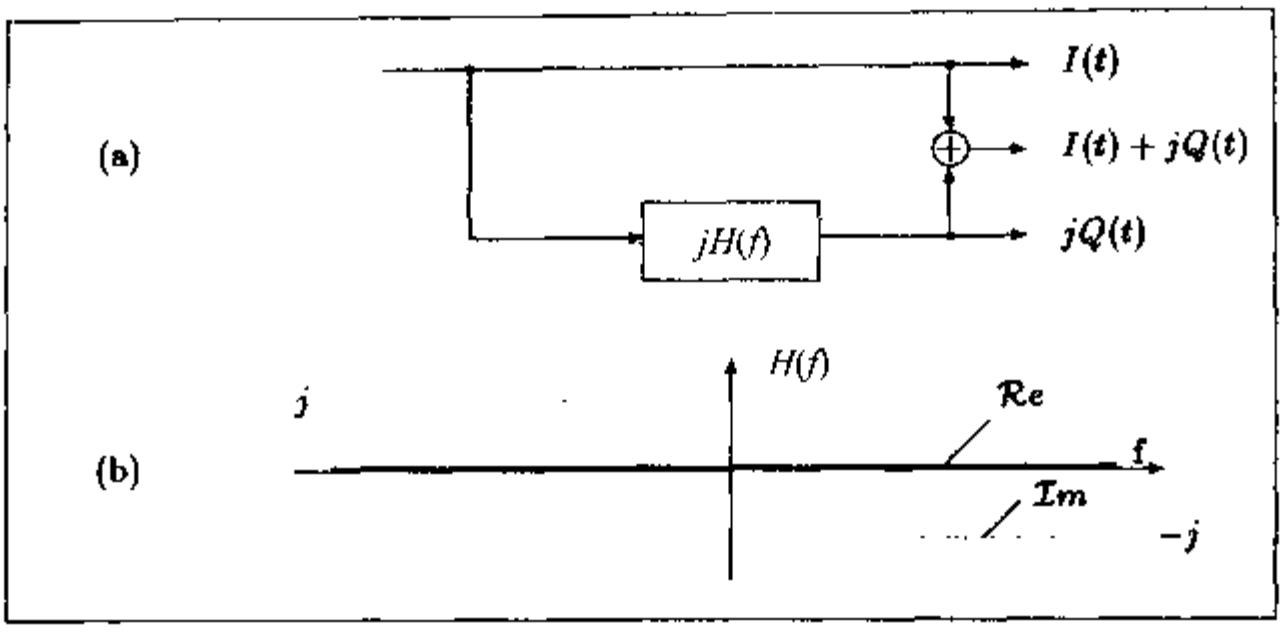


图 7-43　Hilbert 变换器 (a) 滤波器 (b) $H(f)$的频谱

窄带接收器的简化

如果输入信号是窄带信号，也就是传输的位的速率要比载波频率小得多，就可以在解调方案中做一些简化。在输入采样方案中可以以载波的速率进行采样，或是以载波的周期 $T_0=1/f_0$ 的一个因子进行采样，这样就保证采样信号与载波成份完全无关。

如果零 IF 接收器以 $4f_0$ 的速率采样，正交方案就变得不重要了。这种情况下正弦和余弦成份就是元素 0, 1, -1，载波相位是 0，90°，180°...。这在文献中通常称之为“复数采样”[189, 190]。也可以运用采样过疏，也就是用采样器每两个或三个载波周期计算一次，而且采样的信号仍然与载波频率无关。

如果信号以 $T_0/4$ 的速率采样，那么 Hilbert 变换器也可以被简化。可以采用 $Q_{-1}=1$ 的一阶或对称类型系数 $Q_1=-0.5$，$Q_{-1}=0.5$ 或非对称系数 $Q_{-1}=1.5$，$Q_{-3}=0.5$ 的二阶 Hilbert 采样器[191]。

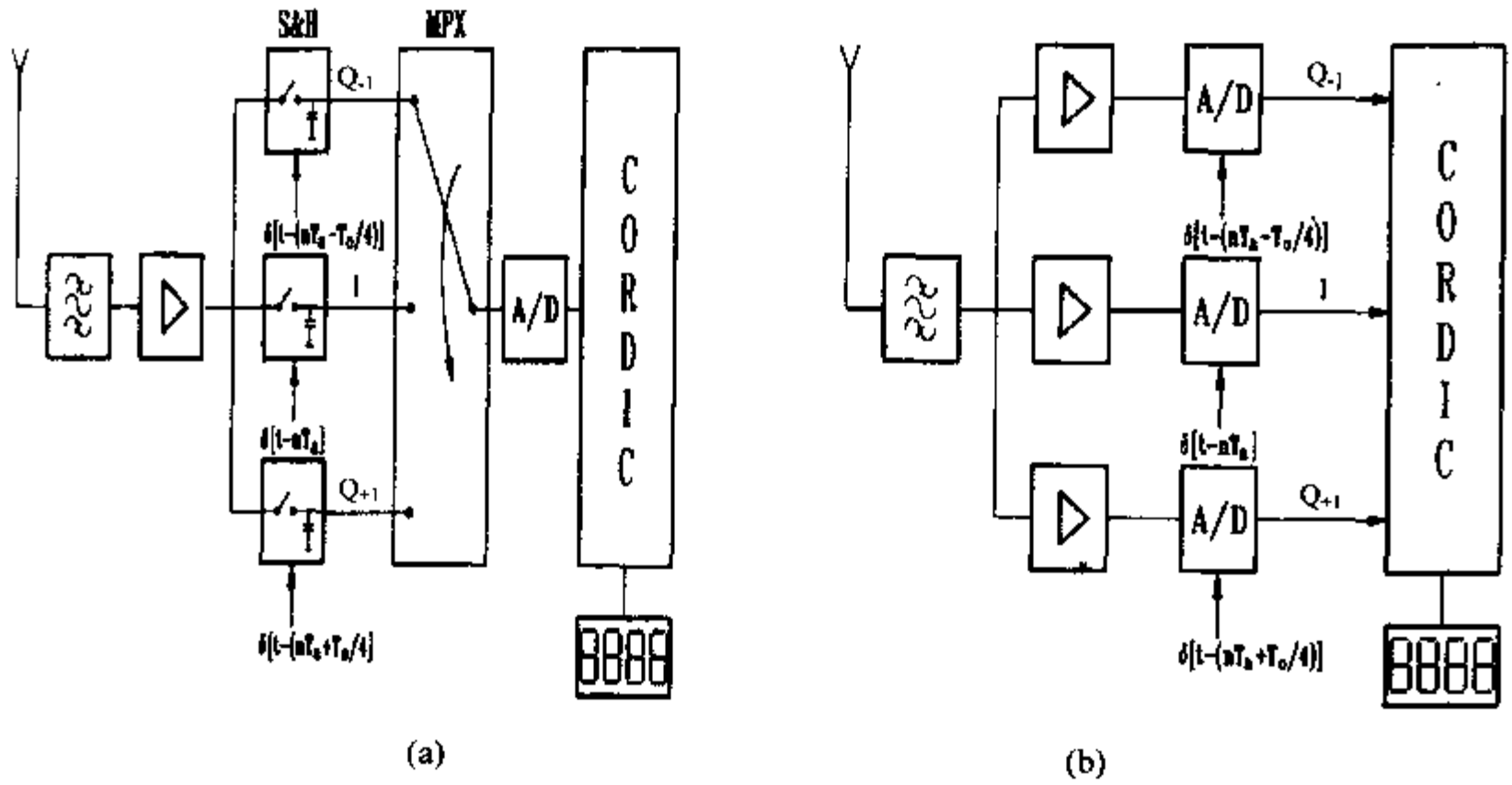
表 7-19 给出了 3 个短项 Hilbert 变换器系数和调制的最大允许频率偏移 Δf，Hilbert 滤波器提供了具体的精度。

表 7-19　Hilbert 采样器的系数

类　型	系　数	位	$\Delta f / f_0$
零阶	$Q_{-1}=1.0$	8	0.005069
	$Q_{-1}=1.0$	12	0.000320
	$Q_{-1}=1.0$	16	0.000020
二阶非对称	$Q_{-1}=1.5$；$Q_{-3}=0.5$	8	0.032805
	$Q_{-1}=1.5$；$Q_{-3}=0.5$	12	0.008238
	$Q_{-1}=1.5$；$Q_{-3}=0.5$	16	0.002069
二阶对称	$Q_1=-0.5$；$Q_{-1}=0.5$	8	0.056825
	$Q_1=-0.5$；$Q_{-1}=0.5$	12	0.014269
	$Q_1=-0.5$；$Q_{-1}=0.5$	16	0.003584

图 7-44 展示了用于解调无线电控制监视器信号的 Hilbert 采样器的两种可能实现[192, 193]。

第一种方法采用了 3 个采样——保持电路，第二种方法采用 3 个 A/D 转换器构造了一个一阶对称 Hilbert 采样器。



(a)　(b)

图 7-44　一阶 Hilbert 采样器的两种形式

图 7-45 给出了直接采样过疏 2 倍的 Hilbert 采样器的频谱性状。

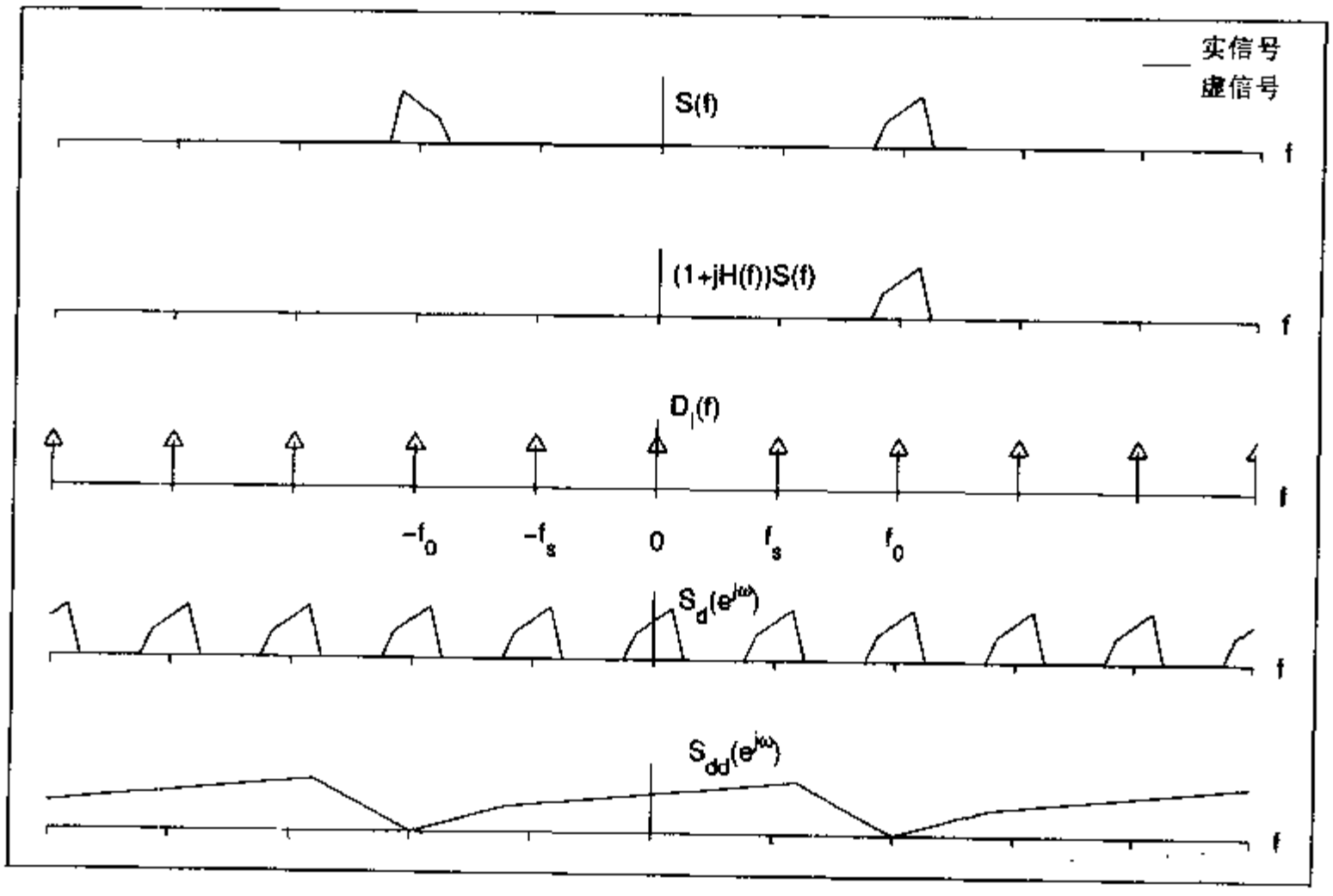


图 7-45　采样过疏的 Hilbert 采样器的频谱

7.3.3　相干解调

如果接收器的相位 ϕ_0 已知，则解调就可以用乘法和低通滤波器来实现。对于 AM，所接收到的信号 $s(t)$乘以 $2\cos(\omega_0 t+\phi_0)$，而 PM 或 FM 则乘以 $-2\sin(\omega_0 t+\phi_0)$，就有：

AM：

$$A(t)\cos(2\pi f_0 t+\phi_0)\cdot 2\cos(2\pi f_0 t+\phi_0)$$

$$=\underbrace{A(t)}_{\text{低通部分}}+A(t)\cos(4\pi f_0 t+2\phi_0)+ \tag{7.61}$$

$$s_{\mathrm{AM}}(t)=A(t)-A_0 \tag{7.62}$$

PM：

$$-2\sin(2\pi f_0 t+\phi_0)\cdot 2\cos(2\pi f_0 t+\Delta\phi(t))$$

$$=\underbrace{\sin(\Delta\phi(t))}_{\text{低通部分}}+\cos(4\pi f_0 t+2\phi_0+\Delta\phi(\mathrm{t})) \tag{7.63}$$

$$\sin(\Delta\phi(t))\approx\Delta\phi(t) \tag{7.64}$$

$$s_{\mathrm{PM}}(t)=\frac{1}{\eta}\Delta\phi(t) \tag{7.65}$$

FM：

$$s_{\mathrm{FM}}(t)=\frac{1}{\eta}\frac{d\ \Delta\phi(t)}{dt} \tag{7.66}$$

η 就是所谓的调制指数。

接下来将要讨论适合于用 FPGA 实现的相干接收器类型。通常情况下，相干接收器的信噪要比非相干接收器好 1dB 以上(参阅图 7-4)。同步或相干 FM 接收器用环中的压控振荡器(voltage-controlled oscillator, VCO)跟踪引入信号的载波相位。这个电压的直流(DC)部分与 FM 信号成正比。PM 信号解调需要 VCO 控制信号的积分，而 AM 解调需要另一个混频器与一个带有低通滤波器的 $\pi/2$ 移相器之间的加法。相干解调的风险就是，对于低信噪信道，环可能会开锁，且性能的下降十分惊人。

通用的相干接收器环有两种：锁相环(Phase-Locked Loop，PLL)和边环(Costas Loop，CL)。图 7-46 和 7-47 分别是 PLL 和 CL 的方框图，可以看出 CL 的复杂性几乎是 PLL 的两倍。每种环都可以用模拟电路(线性 PLL/CL)或全数字电路(ADPLL, ADCL)实现(参阅【194】、【195】、【196】、【197】)。这些环的稳定性分析已经超出本书所讨论的范围，在文献(【198】、【199】、【200】、【201】、【202】)中均有论述。第一种 PLL 是模拟 PLL 到 FPGA 技术的一个直接转换。

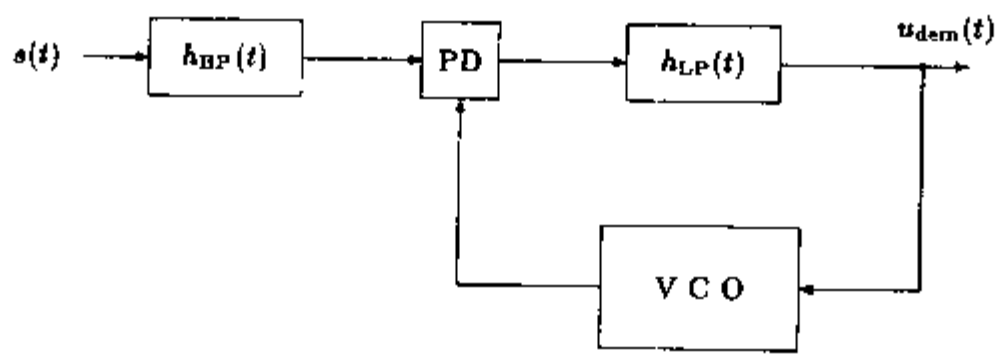


图 7-46 带有(必需的)带通滤波器($h_{\mathrm{BP}}(t)$)的锁相环(PLL)，相位检测器(PD)，低通滤波器($h_{\mathrm{LP}}(t)$)和压控振荡器(VCO)

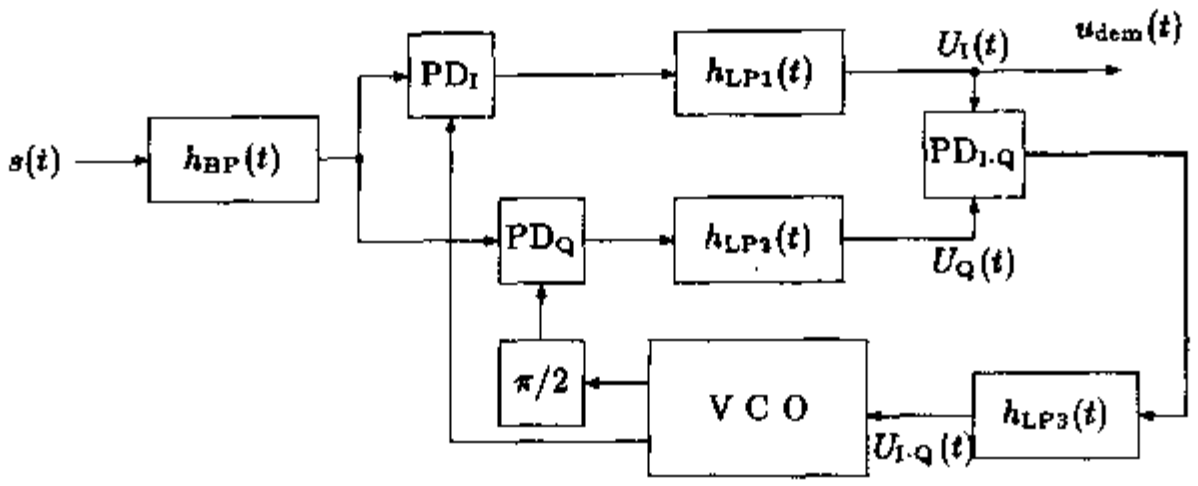


图 7-47　带有(必需的)带通滤波器($h_{BP}(t)$)的边环(CL)，3 个相位检测器(PD)，3 个低通滤波器($h_{LP}(t)$)和带有 $\pi/2$ 移相器的压控振荡器(VCO)

1. 线性锁相环

线性和数字环之间的区别就在于需要处理的输入信号的类型不同。线性 PLL 或 CL 运用快速乘法器作为相位检测器，为环提供可能的多电平输入信号。数字 PLL 或 CL 只能够处理二进制输入信号(数字在这里指的是输入信号的量化，而不是硬件的实现！)。

如图 7-46 所示，线性 PLL 具有 3 个主要模块：

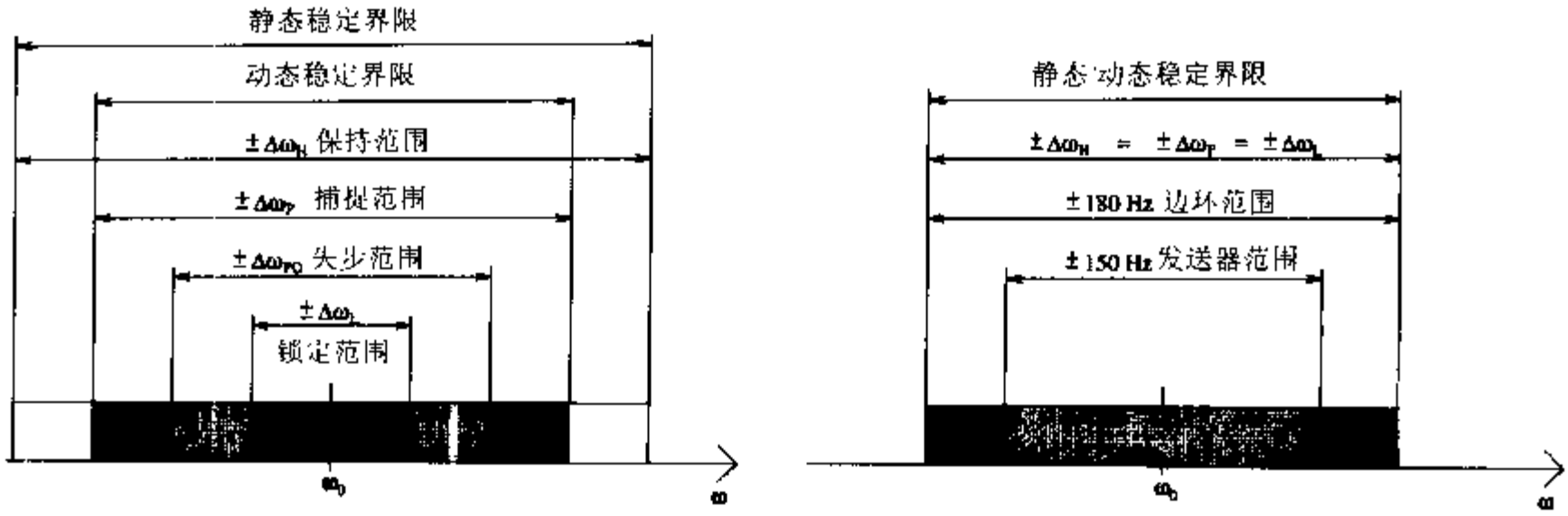
- 作为相位检测器的乘法器
- 环路滤波器
- VCO

为了保证环处于闭锁状态，要求环路输出信噪比要大于 4=6dB[197, 35 页]。由于通常天线的选择还不能够窄到满足这个条件，所以就需要在图 7-46 和 7-47 中添加一个额外的的窄带通滤波器作为解调器的补充[204]。“级联带通梳状(cascaded bandpass comb)”滤波器(参阅表 5.4)就是一个很有效的例子。但是，如果采用固定的 IF，则该滤波器的设计就更加简单，就像超外差或双变频超外差接收器一样。

VCO(或是 ADPLL 的数控振荡器，digitally controlled oscillator, DCO)以 $\omega_2=\omega_0+K_0*U_f(t)$的频率振荡，其中 ω_0 是支持点，K_0 是 VCO/VDO 的增益。对于正弦输入信号，在低通滤波器的输出端得到信号

$$u_{\text{dem}}(t) = K_d \sin(\Delta\phi(t)) \tag{7.67}$$

其中 $\Delta\phi(t)$是 DCO 输出和带通滤波器输入信号之间的相位差。对于小的差异，正弦可以用其自变量近似，由此得到 $u_{\text{dem}}(t)$与 $\Delta\phi(t)$成正比(环保持闭锁状态)。如果输入信号有一个非常突然的相位中断，环就会开锁。图 7-48 给出了环的不同运行区域。约束范围 $\omega_0\pm\Delta\omega_H$ 是静态运行界限(仅适用于频率合成器)，锁定范围是 PLL 在频率差 $\omega_1-\omega_2$ 的单个周期内闭锁的区域。在捕捉范围内，环将会在捕捉时间 T_L 内完成闭锁，T_L 延续的时间可以超过一个 $\omega_1-\omega_2$ 的周期。失步范围是环在不开锁的情况下可以维持的最大频率跳跃。$\omega_0\pm\Delta\omega_{PO}$ 是在调制中使用的动态运行界限。还有很多关于优化 PD、环路滤波器和 VCO 增益的文献，参阅【198】、【199】、【200】、【201】和【202】。



(a)　　(b)

图 7-48　(a) PLL/CL 的运行区域 (b) 图 7-51 中的 CL 的运行区域

与数字 PLL 和 CL 相比较，线性环的优点就是其噪声缩减的能力，但是在线性 PLL 中用作 PD 的快速乘法器的硬件成本非常高，而且这种 PD 在 π/2 处有一个不稳定的支持点相移(－π/2 是稳定点)，这就阻碍了锁相。表 7-20 计算了用同样的函数组作为 8 位非相干接收器(参阅例 7.15)，线性 PLL 的硬件成本(按 CLB 的数量计算)。在 AM 和 PM 调制的乘法中，采用硬件乘法器，将右边一列的电路成本降低了 58 个 CLB，这样也就可以将其放置在一个 320 个 CLB 的 Xilinx XC3090 元器件中了。

表 7-20　Xilinx XC3000 FPGA 的线性 PLL 通用解调器的成本(按 CLB 计算)

函　数　组	仅 有 FM	FM、AM 和 PM
相位检测器(8×8 位乘法器)	65	72
环路滤波器(两级 CIC)	84	168
频率合成器	34	34
DCO(*N* / *N*+*K* 个除法器，sin/cos 表)	16	16+2
PDSP 接口	15	15
总计	214	307

将这些成本与非相干接收器的成本加以比较，就可以看出，相干接收器没有采用 CORDIC 解调，共使用了 292 个 CLB，而一个额外的 CORDIC 处理器需要 367.5 个 CLB，其结果表明 CLB 的数量在线性 PLL 的实现中又有稍许改进。如果需要的仅仅是 FM 和 PM 解调，下面两个部分将要讨论的数字 PLL 或 CL 会明显地降低其复杂性。

这些设计均是为解调气象传真图片开发的，后者在欧洲中部由低频无线电台 DCF37 和 DCF54(载波 117.4KHz 和 134.2KHz；频率调制 FIC：±150MHz)发送。

2. 数字锁相环

正像上一节所解释的，数字 PLL 是用二进制输入信号工作的。数字 PLL 的相位检测器要比线性 PLL 中用到的快速乘法器简单；通常选择 XOR 门、边沿触发的 JK 触发器或是带有一些

附加门的成对的 RS 触发器[197, 60-65 ×]。图 7-49 给出的相位检测器是最复杂的，但是它提供了相位和频率灵敏度以及近似无限的约束范围。

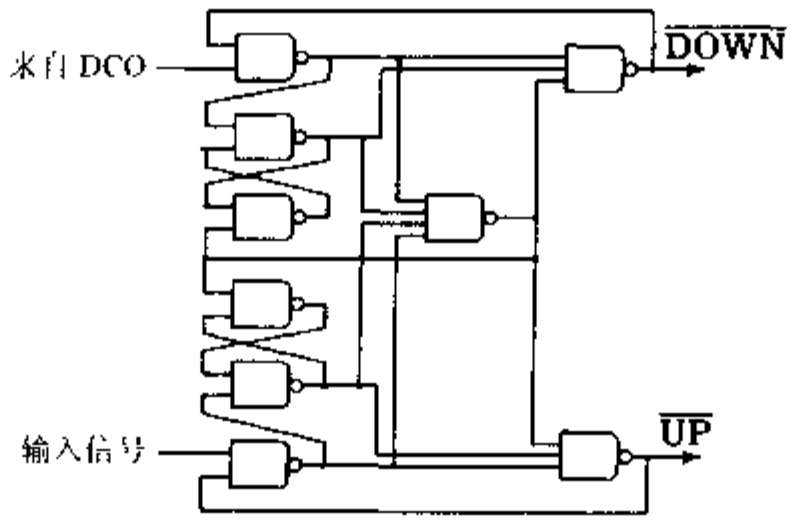


图 7-49　相位检测器[18, 8-127 ×]

改进的计数器可以用作 DPLL 的环路滤波器。可以是 $N/(N+K)$个计数器或多级计数器，例如 N/M 个除法器，分别采用的是向上和向下的计数器，第三个计数器计算两者之差。如果某个阈值不满足，这就可能会中断进一步的信号处理。对于 DCO，有可能会用到任何典型的全数字频率合成器、如：累加器、除法器或是倍增的生成器。最经常使用的合成器就是可调除法器，这是因为它具有较低的相位差。这种合成器的低分辨率可以通过采用一个低接收器 IF 来提高[205]。

一种非常简单的 DPLL 实现就是 74LS297 电路，它采用了一种“脉冲窃取”的设计方法。这种方案可以用相位和频率敏感的 J-K 触发器加以改进，如图 7-50 所示。PLL 运行方式为：“检测触发器”以静止频率：

$$F_{\text{comp}} = \frac{F_{\text{in}}}{N} = \frac{F_{\text{osc}}}{KM} \tag{7.68}$$

运行。

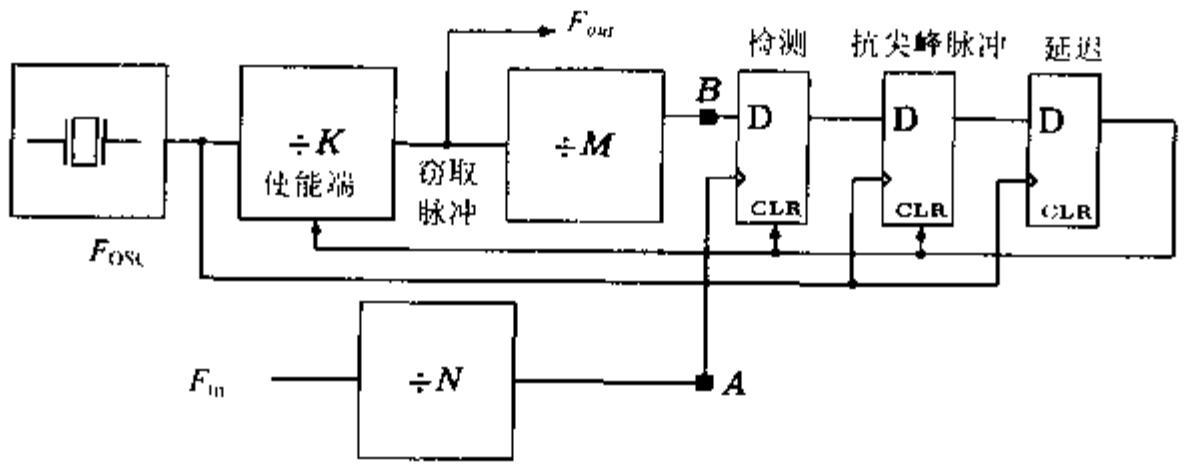


图 7-50　“脉冲窃取”PLL[206]

为了跟踪高于静止频率的输入频率，振荡器频率 $F_{\text{osc+}}$要设置的稍高一些：

$$T_{\text{comp}} - T_{\text{comp+}} = \frac{1}{2} T_{\text{osc}} \tag{7.69}$$

这样，信号在 B 点振荡就比信号 F_{osc} 快半个周期。在静止频率点大约两个周期之后，必然

在检测器触发器内闭锁。这一信号通过分离低频瞬态干扰装置和延迟触发器运行，接下来约束一个÷K 除法器(这就是“脉冲窃取”名称的由来)的脉冲。这样就在 B 处延迟了信号，从而 A 处的信号相位就跑到 B 之后，此过程循环往复。PLL 的闭锁范围有一个更低的边界 $F_{in}|_{min}$=0Hz。上限依赖于最大输出频率 F_{osc}+/K，这样，闭锁范围就变成：

$$\pm \Delta\omega_L = \pm N \cdot F_{osc} + /(K \cdot M) \tag{7.70}$$

接收器可以通过省略计数器 N 和 M 来简化。在气象传真图像译码应用中，双变频超外差接收器的第二个 IF 就是这样设置的，300Hz 频率调制(δf=0Hz→白色；δf=300Hz→黑色)严格地符合 32 个窃取脉冲，这样，窃取脉冲就与灰度级直接对应。我们设定一个 1920 波特的像素率和一个 16.6KHz 的 IF。对于每一个像素有 4 个“窃取值”(一个时间间隔内的窃取脉冲的数量)是确定的，这样总计需要用 $\log_2(2\times4)$=3 个位移来计算 16 个灰度级的值。表 7-21 给出了这种 PLL 类型的硬件复杂性。

表 7-21　脉冲窃取 DOLL 的硬件复杂性[205]

函　数　组	CLB
DCO	16
相位检测器	5
环路滤波器	6
求平均值	11
PC－接口	10
频率合成器	26
状态机	10
总计	84

3. 边环(Costas loop)

这是相干环的一个扩展类型，最初由约翰 P. Costas 在 1956 年提出，他将该环用于载波恢复。如图 7-47 所示，CL 具有同相和正交路径(下标分别是 I 和 Q)，采用了 π/2 移相器和第三个 PD 以及低通滤波器。CL 的复杂程度大约是 PLL 的两倍，但是对信号的闭锁也快了两倍。边环对同相和正交增益的微小变化十分敏感，因此一直是以全数字电路实现。FPGA 似乎就是一种理想的实现工具(【207】、【208】)。

对于信号 $U(t)=A(t)\sin(\omega_0 t+\Delta\phi(t))$，在混频器和低通滤波器之后，就得到：

$$U_I(t)=K_d A(t)\cos(\Delta\phi(t)) \tag{7.71}$$

$$U_Q(t)=K_d A(t)\sin(\Delta\phi(t)) \tag{7.72}$$

其中 $2K_d$ 是 PD 的增益。$U_I(t)$和 $U_Q(t)$在第三个 PD 中相乘，经过低通滤波，就得到 DCO 控制信号：

$$U_{IQ}(t)\sim K_d \sin(2\Delta\phi(t)) \tag{7.73}$$

比较(7.67)式和(7.73)式就可以看出，对于$\Delta\phi(t)$的微小调制，控制信号 $U_{I+Q}(t)$的斜率是 PLL 的两倍。与在 PLL 中一样，如果仅有 FM 或 PM 信号需要解调，PD 也可以是全数字的。

图 7-51 给出了 CL 的一个方框图。天线信号首先经过一个 4 阶 Butterworth 低通滤波器滤波和放大，然后通过一个 8 位转换器以 32 或 64 倍于载波基带频率的采样速率进行数字化。生成的信号是分离的，并进入到过零区间检测器和最小值/最大值检测器。两个相位检测器将信号与参考信号相比较，信号的 π/2 相移副本由时间常数较大的 PLL 与参照累加器合成[4]。每个相位检测器均有两个边沿检测器，总共可以生成 4 个 UP 信号和 4 个 DOWN 信号。如果 PD 生成的 UP 信号多于 DOWN 信号，就说明参照频率太低，反之就是太高。为一位像素在 13 位累加器作为环路滤波器中的延迟积累∑UP－∑DOWN 的差。环路滤波器数据被传递给像素转换器，像素转换器为 DCO 给出了正确值，如表 7-22 所示。累加的和也用作像素的灰度值，并且将该值传递给 PC，以储存并显示气象传真图片。

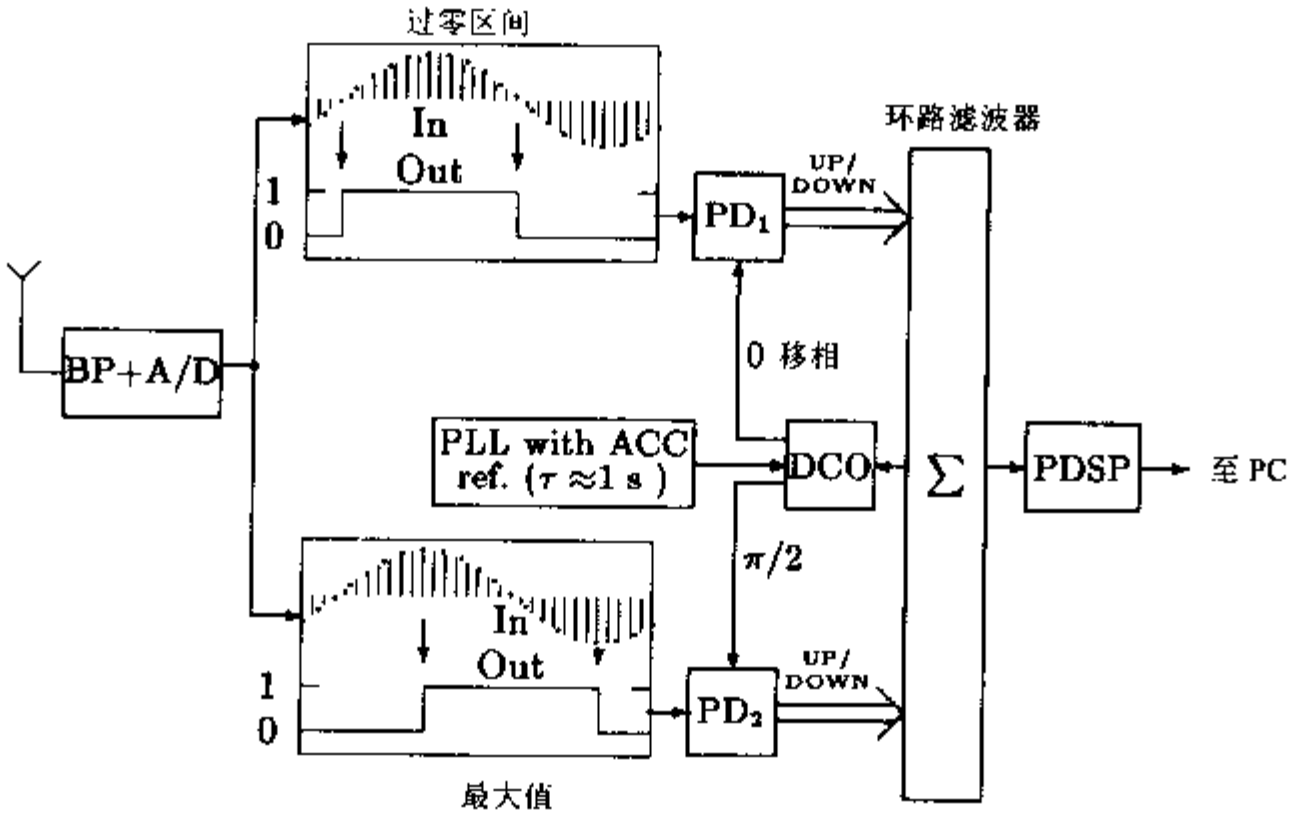


图 7-51　边环的结构

表 7-22　32 倍过采样时环路滤波器输出和 DCO 校正值

累加器					
下　溢	溢　出	和	DCO-IN	灰　度　值	$f_{carrier}\pm\delta f$
是	否	$s<-(2^{13}-1)$	3	0	+180Hz
否	否	$-(2^{13}-1)\le s<-2048$	2	0	+120Hz
否	否	$-2048\le s<-512$	1	4	+60Hz
否	否	$-512\le s<512$	0	8	+0Hz
否	否	$512\le s<2048$	-1	12	-60Hz
否	否	$2048\le s<2^{13}-1$	-2	15	-120Hz
否	是	$s\ge 2^{13}-1$	-3	15	-180Hz

用于 2K 波特像素率的最小可检测的相移是：

$$f_{\text{carrier+1}} = \frac{1}{1/f_{\text{carrier}} - t_{\text{ph37}} \cdot 2kBaud / f_{\text{carrier}}} = 117.46\text{KHz} \tag{7.74}$$

其中 t_{ph37}=1(32×117KHz)=266ns 是 32 倍过采样的采样周期。频率解是 117.46KHz - (f_{carrier} = 117.4KHz) = 60Hz。在 300Hz 的频率调制下，可以区分 5 个灰度值。由于 3164-4ns FPGA 和所用的 A/D 转换器的限制，对累加器使用更高采样速率是不可能的。如果采用快速 A/D 转换器和 Altera Flex 或 Xilinx XC4K FPGA，就可以使用 128 和 256 倍的过采样。

当最大相移为 π 时，环需要⌈16/3⌉=6 次采样的最大闭锁时间 T_L，或者说是 1.5μs。表 7-23 给出了 32 和 64 倍过采样时 CL 的复杂性。

表 7-23　全数字边环的复杂性[204, 80 页]

函　数　组	32 倍过采样时的 CLB	64 倍过采样时的 CLB
频率合成器	33	36
零点检测	42	42
最大值检测	16	16
四相位检测器	8	8
环路滤波器	51	51
DCO	12	15
TMS 接口	11	11
总计	173	179

7.4　练习

7.1：下面的 MATLAB 代码可以用来计算一个元素的阶。

```
function N = order(x, M)
% Compute the order of x modulo M
p=x; l=1;
while p ~=1
    l=l+1; p = p*x;
    re = real(p); im = imag(p);
    p =mod(re, M) + i* mod(im, M);
end:
N=l;
```

假设这个函数被 order(2,2^25+1)调用，其结果将是 50。要计算 2^{25}+1 的单因子，可以利用标准 MATLAB 函数 factor(2^25+1)。为：

(a) α=2 和 $M=2^{41}+1$

(b) α= - 2 和 $M=2^{29}-1$

(c) α=1+j 和 $M=2^{29}+1$

(d) $\alpha=1+j$ 和 $M=2^{26}-1$

计算变换长度、“坏”因子 v(也就是阶不等于 order(α, $2^B\pm1$))、所有“好”因子 M/v 和可能的输入位宽 $B_x=(\log_2(M/v)-\log_2(L))/2$。

7.2：为值 x 的 gcd(x, M)计算逆 $x^{-1}\bmod M$，可以利用丢番图方程(diophantic equation)的结果：

$$\gcd(x,M)=u\cdot x+v\cdot M \qquad u,v\in Z \tag{7.75}$$

(a) 解释如何利用 MATLAB 函数[g u v]=gcd(x,M)计算乘法逆元。

如果可能，请计算下列逆元：

(b) $3^{-1}\bmod 73$；

(c) $64^{-1}\bmod 2^{32}+1$；

(d) $31^{-1}\bmod 2^{31}-1$；

(e) $89^{-1}\bmod 2^{11}-1$；

(f) $641^{-1}\bmod 2^{32}+1$；

7.3：可以用下面的 MATLAB 代码计算长度为 2，4，8，且 16 模 257 的 Fermat NTT。

```
function Y = ntt(x)
% Compute Fermat NTT of length 2,4,8 and 16 modulo 257
l = length(x);
switch(l)
case 2, alpha= - 1;
case 4, alpha= 16;
case 8, alpha=4;
case 16, alpha=2;
otherwise, disp('NTT length not support')
end
A=ones(1,1);A(2,2)=alpja;
%*********Computing second column
for m=3: 1;
 A(m,2)=mod(A(m - 1,2)*alpha,257);
end
%*********Computing rest of matrix
for m=2: l
   for n=2: l-1
      A(m,n+1)=mod(A(m,n)*A(m,2),257);
   end
end
%*********Computing NTT A*x
for k=1: l
   C1=0;
for j=1: l
      C1=C1+A(k,j)*x(j);
   end
```

```
    X(k)=mod(C1,257);
  end
  Y=X;
```

(a) 计算 x={1,1,1,1,0,0,0,0}的 NTT X。

(b) 写出相应的 INTT 代码。计算(a)中 X 的 INTT。

(c) 计算逐位乘积 $Y=X\odot X$ 和 INTT(Y)=y。

(d) 扩展复杂 Fermat NTT 和 $\alpha=1+j$ 时 INTT 的代码。并用恒等式 x=INTT(NTT(x))验证您的程序。

7.4：$N=4$ 的 Walsh 变换式是由下面的矩阵给出的：

$$\boldsymbol{W}_4=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\\ 1 & -1 & 1 & -1\end{bmatrix}$$

(a) 计算行向量的内积。这种矩阵具有什么特性？

(b) 用(a)中的结果计算 $\boldsymbol{W}_4^{-1}$。

(c) 计算 8×8 Walsh 矩阵 $\boldsymbol{W}_8$，用 2 刻度初始行向量(也就是 $h[n/2]$)，并计算来自第 3 行和第 4 行额外的两个“子式” $h[n]+h[n-4]$和 $h[n]-h[n-4]$。在结果矩阵 $\boldsymbol{W}_8$ 中应该没有零。

(d) 绘出构造高阶 Walsh 矩阵的函数树。

7.5：可以用如下迭代计算 Hadamard 矩阵：

$$\boldsymbol{H}_{2^{l+1}}=\begin{bmatrix}\boldsymbol{H}_{2^l} & \boldsymbol{H}_{2^l}\\ \boldsymbol{H}_{2^l} & -\boldsymbol{H}_{2^l}\end{bmatrix} \tag{7.76}$$

其中 $\boldsymbol{H}_1=[1]$。

(a) 计算 $\boldsymbol{H}_2$、$\boldsymbol{H}_4$ 和 $\boldsymbol{H}_8$。

(b) 为 $\boldsymbol{H}_4$ 和 $\boldsymbol{H}_8$ 中的行找到合适的索引，并与练习 7.4 中的 Walsh 矩阵 $\boldsymbol{W}_4$ 和 $\boldsymbol{W}_8$ 相比较。

(c) 确定从 Walsh 矩阵到 Hadamard 矩阵的映射的一般规则。

提示：

首先以二进制表示法计算索引。

7.6：下面的 MATLAB 代码可以用来计算 $p_{14}=x^{14}+x^5+x^3+x^1+1$ 的状态空间表达式。非零元素用 nnz 表示，扇入端数为最大值。

```
p= input('Please define power of matrix = ')
A=zeros(14,14);
for m=1:13;
   A (m,m+1)=1;
```

```
end
A(14,14)=1;
A(14,13)=1;
A(14,11)=1;
A(14,9)=1;
Ap=mod(A^p,2);
nnz(Ap)
max(sum(Ap,2))
```

(a) 计算非零元素的数量以及 p=2 到 8 时的扇入端数。

(b) 修改代码，计算双生子 $p_{14}=x^{14}+x^{13}+x^{11}+x^9+1$。计算修改后的多项式在 $p=2$ 至 8 时的非零元素的数量。

(c) 修改初始代码，计算(a)和(b)的另一种 LFSR 实现(参考图 7-21)并计算 p=2 到 8 时的非零元素。

练习使用 MaxPlusII

7.7：(a) 用 MaxPlusII 编译例 7.12 中长度为 6 的 LFSR 的代码文件 lfsr.vhd。

(b) 用：

```
ff(1)<=ff(5) XNOR ff(6);
```

代替

```
ff(1)<=NOT (ff(5) XOR ff(6));
```

并用 MaxPlusII 编译。

(c) 接下来改变编译器设置 Interfaces→VHDL Netlist Reader Setting 从 VHDL 1987 到 VHDL 1993，再一次进行编译，并解释其结果。

附录A Verilog 源代码

```
//*************************************************************
// IEEE STD 1364-1995 Verilog file: example.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
//'include "220model.v"      // Using predefined components

module example (clk, a, b, op1, sum, d);   //----> Interface

   parameter WIDTH =8;       // Bit width
   input   clk;
   input   [WIDTH - 1:0] a, b, op1;
   output [WIDTH - 1:0]  sum, d;

   wire [WIDTH - 1:0]    c;              // Auxiliary variables
   reg   [WIDTH - 1:0]    s;             // Infer FF with always
   wire [WIDTH - 1:0] op2, op3;

   wire    clkena, ADD, ena, aset, sclr, sset, aload, sload,
                        aclr, ovfl, cinl; // Auxiliary lpm signals

// Default for add:
   assign cinl=0; assign aclr=0; assign ADD=1;

   assign ena=1; assign aclr=0; assign aset=0;
   assign sclr=0; assign sset=0; assign aload=0;
   assign sload=0; assign clkena=0; // Default for FF

   assign op2 = b;            // Only one vector type in Verilog;
                  // no conversion int -> logic vector necessary

// Note when using 220model.v ALL component's signals
// must be defined, default values can only be used for
// the parameters.

   lpm_add_sub add1                //----> Component instantiation
   ( .result(op3), .dataa(op1), .datab(op2)); // Used ports
//    .cin(cinl),.cout(crl), .add_sub(ADD), .clken(clkena),
```

```
//    .clock(clk), .overflow(ovl1), .aclr(aclr)); // Unused
      defparam add1.lpm_width = WIDTH;
      defparam add1.lpm_representation = "SIGNED";

   lpm_ff reg1
   ( .data(op3), .q(sum), .clock(clk));   // Used ports
//   .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//   .sset(sset), .aload(aload), .sload(sload)); // Unused
      defparam reg1.lpm_width = WIDTH;

   assign c = a + b; //----> Continuous assignment statement

   always @(posedge clk)     //----> Behavioral style
   begin : p1                // Infer register
      s = c + s;             // Signal assignment statement
   end
   assign d = s;

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: fun_text.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
//   A 32 bit function generator using accumulator and ROM
//'include "220model.v"

module fun_text (M, sin, acc, clk); //----> Interface

   parameter WIDTH = 32;     // Bit width
   input   [WIDTH - 1:0]   M;
   output [7:0]   sin, acc;
   input                  clk;

   wire [WIDTH - 1:0] s, acc32;
   wire [7:0]       msbs;                    // Auxiliary vectors
   wire    ADD, ena, aset, sclr, sset;       // Auxiliary signals
   wire aload, sload, aclr, ovfl, cin1, clkena;

   // Default for add:
   assign clkena=0; assign cin1=0; assign ADD=1;
   //default for FF:
   assign ena=1; assign aclr=0; assign aset=0; assign sclr=0;
   assign sset=0; assign aload=0; assign sload=0;
```

```
    lpm_add_sub add_1                                  // Add M to acc32
    ( .result(s), .dataa(acc32), .datab(M));           // Used ports
//    .cout(cr1), .add_sub(ADD), .overflow(ovl1),      // Unused
//    .clock(clk),.cin(cin1), .clken(clkena), .aclr(aclr));
//
       defparam add_1.lpm_width = WIDTH;
       defparam add_1.lpm_representation = "UNSIGNED";


    lpm_ff reg_1                                       // Save accu
    ( .data(s), .q(acc32), .clock(clk));               // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset),          // Unused ports
//    .sset(sset), .aload(aload), .sload(sload),.sclr(sclr));
       defparam reg_1.lpm_width = WIDTH;

    assign msbs = acc32[WIDTH - 1:WIDTH - 8];
    assign acc  = msbs;

    lpm_rom rom1
    ( .q(sin), .inclock(clk), .outclock(clk),
                                  .address(msbs)); // Used ports
//                          .memenab(ena) ) ;        // Unused port
       defparam rom1.lpm_width = 8;
       defparam rom1.lpm_widthad = 8;
       defparam rom1.lpm_file = "sine.mif ";

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: add_1p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
//'include "220model.v"

module add_1p (x, y, sum, clk);
  parameter WIDTH  = 15, // Total bit width
            WIDTH1 = 7,  // Bit width of LSBs
            WIDTH2 = 8;  // Bit width of MSBs

  input  [WIDTH - 1:0] x,y;      // Inputs
  output [WIDTH - 1:0] sum;      // Result
  input                clk;      // Clock

  reg  [WIDTH1 - 1:0] l1, l2; // LSBs of inputs
  wire [WIDTH1 - 1:0] q1, r1; // LSBs of inputs
```

```
    reg   [WIDTH2 - 1:0] l3, l4; // MSBs of input
    wire [WIDTH2 - 1:0] r2, q2, u2; // MSBs of input
    reg   [WIDTH - 1:0]   s;       // Output register
    wire cr1,cq1; // LSBs carry signal
    wire [WIDTH2 - 1:0] h2; // Auxiliary MSBs of input

    wire   clkena, ADD, ena, aset, sclr;   // Auxiliary signals
    wire sset, aload, sload, aclr, ovf1, cin1;

// Default for add:
    assign cin1=0; assign aclr=0; assign ADD=1;

    assign ena=1; assign aclr=0;                    // Default for FF
    assign sclr=0; assign sset=0; assign aload=0;
    assign sload=0; assign clkena=0; assign aset=0;

    // Split in MSBs and LSBs and store in registers
    always @(posedge clk) begin
      // Split LSBs from input x,y
      l1[WIDTH1 - 1:0] <= x[WIDTH1 - 1:0];
      l2[WIDTH1 - 1:0] <= y[WIDTH1 - 1:0];
      // Split MSBs from input x,y
      l3[WIDTH2 - 1:0] <= x[WIDTH2 - 1+WIDTH1:WIDTH1];
      l4[WIDTH2 - 1:0] <= y[WIDTH2 - 1+WIDTH1:WIDTH1];
    end
/************** First stage of the adder   ******************/
    lpm_add_sub add_1                                    // Add LSBs of x and y
    ( .result(r1), .dataa(l1), .datab(l2), .cout(cr1));
                                                         // Used ports
//    .overflow(ovl1), .clken(clkena), .add_sub(ADD),
//    .cin(cin1), .clock(clk), .aclr(aclr)); // Unused ports
       defparam add_1.lpm_width = WIDTH1;
       defparam add_1.lpm_direction = "add";

    lpm_ff reg_1                                  // Save LSBs of x+y
    ( .data(r1), .q(q1), .clock(clk));            // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload));     // Unused
       defparam reg_1.lpm_width = WIDTH1;

    lpm_ff reg_2          // Save LSBs carry
    ( .data(cr1), .q(cq1), .clock(clk));          // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_2.lpm_width = 1;
```

```
    lpm_add_sub add_2                          // Add MSBs of x and y
    ( .dataa(l3), .datab(l4), .result(r2)        ); // Used ports
//    .add_sub(ADD), .cout(cout1), .cin(cin1), .clken(clkena),
//    .overflow(ovl1),    .clock(clk), .aclr(aclr)); // Unused
      defparam add_2.lpm_width = WIDTH2;
      defparam add_2.lpm_direction = "add";

  lpm_ff reg_3                                 // Save MSBs of x+y
  ( .data(r2), .q(q2), .clock(clk));           // Used ports
// .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
// .sset(sset), .aload(aload), .sload(sload));      // Unused
       defparam reg_3.lpm_width = WIDTH2;


/*************** Second stage of the adder ****************/
// One operand is zero
   assign h2 = {WIDTH2{1'b0}};

   lpm_add_sub add_3     // Add MSBs (x+y) and carry from LSBs
   ( .cin(cq1), .dataa(q2), .datab(h2), .result(u2));
                                                  // Used ports
//    .cout(cout1), .overflow(ovl1), .clken(clkena),      // Unused
//    .add_sub(ADD), .clock(clk), .aclr(aclr));           // ports
       defparam add_3.lpm_width = WIDTH2;
       defparam add_3.lpm_direction = "add";

    always @(posedge clk) begin    // Build a single registered
      s = {u2[WIDTH2 - 1:0],q1[WIDTH1 - 1:0]};        // output word
    end                                          // of WIDTH=WIDTH1+WIDHT2

    assign sum = s ;       // Connect s to output pins

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: add_2p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
// 22-bit adder with two pipeline stages
// uses four components: csa7.v; csa7cin.v;
//                       add_ff8.v; add_ff8cin.v

//'include "220model.v"
//'include "csa7.v"
//'include "csa7cin.v"
```

```
//'include "add_ff8.v"
//'include "add_ff8cin.v"

module add_2p (x, y, sum, clk);
  parameter WIDTH   = 22,  // Total bit width
            WIDTH1  = 7,   // Bit width of LSBs
            WIDTH2  = 7,   // Bit width of middle s
            WIDTH12= 14,  // Sum WIDTH1+WIDTH2
            WIDTH3 =   8;  // Bit width of MSBs

  input   [WIDTH - 1:0] x,y;      // Inputs
  output [WIDTH - 1:0] sum;       // Result
  input                clk;       // Clock

  reg   [WIDTH1 - 1:0] l1, l2; // LSBs of inputs
  wire [WIDTH1 - 1:0] q1, v1, s1; // LSBs of inputs
  reg   [WIDTH2 - 1:0] l3, l4; // Middle bits
  wire [WIDTH2 - 1:0] q2, h2, v2, s2; // Middle bits
  reg   [WIDTH3 - 1:0] l5, l6; // MSBs of input
  wire [WIDTH3 - 1:0] q3, h3, v3, s3; // MSBs of input
  wire [WIDTH - 1:0]   s;       // Output register
  wire cq1, cq2, cv2; // Carry signals
  wire ena, aset, sclr, sset, aload, sload, aclr;
                                          // Auxiliary FF signals
  assign ena=1; assign aclr=0; assign aset=0;
  assign sclr=0; assign sset=0; assign aload=0;
  assign sload=0;                         // Default for FF

  // Split in MSBs and LSBs and store in registers
  always @(posedge clk) begin
    // Split LSBs from input x,y
    l1[WIDTH1 - 1:0] <= x[WIDTH1 - 1:0];
    l2[WIDTH1 - 1:0] <= y[WIDTH1 - 1:0];
    // Split middle bits from input x,y
    l3[WIDTH2 - 1:0] <= x[WIDTH2 - 1+WIDTH1:WIDTH1];
    l4[WIDTH2 - 1:0] <= y[WIDTH2 - 1+WIDTH1:WIDTH1];
    // Split MSBs from input x,y
    l5[WIDTH3 - 1:0] <= x[WIDTH3 - 1+WIDTH12:WIDTH12];
    l6[WIDTH3 - 1:0] <= y[WIDTH3 - 1+WIDTH12:WIDTH12];
  end

//*************** First stage of the adder  ****************
  csa7 add_1          // Add LSBs of x and y
  ( .a(l1), .b(l2), .clock(clk), .s(q1), .c(cq1));
```

```
    csa7 add_2          // Add LSBs of x and y
    ( .a(l3), .b(l4), .clock(clk), .s(q2), .c(cq2) );

  add_ff8 add_3                 // Add MSBs of x and y
  ( .a(l5), .b(l6), .clock(clk), .s(q3));

//************** Second stage of the adder ******************
    // Two operands are zero
    assign h2 = {WIDTH2{1'b0}};
    assign h3 = {WIDTH3{1'b0}};

    lpm_ff reg_1                    // Save q1
    ( .data(q1), .q(v1), .clock(clk));  // Used ports
//        .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//        .sset(sset), .aload(aload), .sload(sload)); // Unused
    defparam reg_1.lpm_width = WIDTH1;

// Add result of middle bits (x+y) and carry from LSBs
    csa7cin add_4
    (.a(q2), .b(h2), .cin(cq1), .clock(clk), .s(v2),.c(cv2));

// Add result of MSBs bits (x+y) and carry from middle
    add_ff8cin add_5
    ( .a(q3), .b(h3), .cin(cq2), .clock(clk), .s(v3));

//************** Third stage of the adder ******************
    lpm_ff reg_2                    // Save v1
    ( .data(q1), .q(s1), .clock(clk));    // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_2.lpm_width = WIDTH1;

    lpm_ft reg_3                    // Save v2
    ( .data(v2), .q(s2), .clock(clk));    // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_3.lpm_width = WIDTH1;

// Add result of MSBs bits (x+y) and 2. carry from middle
    add_ff8cin add_6
    ( .a(v3), .b(h3), .cin(cv2), .clock(clk), .s(s3));

// Build a single output word of WIDTH=WIDTH1+WIDHT2+WIDTH3
    assign s ={s3[WIDTH3 - 1:0],s2[WIDTH2 - 1:0],s1[WIDTH1 - 1:0]};
```

```
    assign sum = s;       // Connect s to output pins

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: add_3p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
// 29-bit adder with three pipeline stage
// uses four components: csa7.v; csa7cin.v;
//                       add_ff8.v; add_ff8cin.v

//'include "220model.v"
//'include "csa7.v"
//'include "csa7cin.v"
//'include "add_ff8.v"
//'include "add_ff8cin.v"

module add_3p (x, y, sum, clk);
  parameter WIDTH      = 29,      // Total bit width
            WIDTH0     = 7,       // Bit width of LSBs
            WIDTH1     = 7,       // Bit width of 2. LSBs
            WIDTH01    = 14,      // Sum WIDTH0+WIDTH1
            WIDTH2     = 7,       // Bit width of 2. MSBs
            WIDTH012   = 21,      // Sum WIDTH0+WIDTH1+WIDTH2
            WIDTH3     = 8;       // Bit width of MSBs

input  [WIDTH - 1:0] x,y;         // Inputs
output [WIDTH - 1:0] sum;         // Result
input                clk;         // Clock

reg  [WIDTH0 - 1:0] l0, l1; // LSBs of inputs
wire [WIDTH0 - 1:0] q0, v0, r0, s0; // LSBs of inputs
reg  [WIDTH1 - 1:0] l2, l3; // 2. LSBs of input
wire [WIDTH1 - 1:0] q1, v1, r1, s1; // 2. LSBs of input
reg  [WIDTH2 - 1:0] l4, l5; // 2. MSBs bits
wire [WIDTH2 - 1:0] q2, v2, r2, s2, h7; // 2. MSBs bits
reg  [WIDTH3 - 1:0] l6, l7; // MSBs of input
wire [WIDTH3 - 1:0] q3, v3, r3, s3, h8; // MSBs of input
wire [WIDTH - 1:0]  s;      // Output register
wire cq0, cq1, cq2, cv1, cv2, cr2; // Carry signals
wire ena, aset, sclr, sset, aload, sload, aclr;
                                   // Auxiliary FF signals

assign ena=1; assign aclr=0; assign aset=0;
```

```
    assign sclr=0; assign sset=0; assign aload=0;
    assign sload=0;                                  // Default for FF

  // Split in MSBs and LSBs and store in registers
  always @(posedge clk) begin
    // Split LSBs from input x,y
    l0[WIDTH0 - 1:0] <= x[WIDTH0 - 1:0];
    l1[WIDTH0 - 1:0] <= y[WIDTH0 - 1:0];
    // Split 2. LSBs from input x,y
    l2[WIDTH1 - 1:0] <= x[WIDTH1 - 1+WIDTH0:WIDTH0];
    l3[WIDTH1 - 1:0] <= y[WIDTH1 - 1+WIDTH0:WIDTH0];
    // Split 2. MSBs from input x,y
    l4[WIDTH2 - 1:0] <= x[WIDTH2 - 1+WIDTH01:WIDTH01];
    l5[WIDTH2 - 1:0] <= y[WIDTH2 - 1+WIDTH01:WIDTH01];
    // Split MSBs from input x,y
    l6[WIDTH3 - 1:0] <= x[WIDTH3 - 1+WIDTH012:WIDTH012];
    l7[WIDTH3 - 1:0] <= y[WIDTH3 - 1+WIDTH012:WIDTH012];
  end

  //*************** First stage of the adder   ******************
    csa7 add_0                              // Add LSBs of x and y
    ( .a(l0), .b(l1), .clock(clk), .s(q0), .c(cq0));
    csa7 add_1                              // Add 2. LSBs of x and y
    ( .a(l2). .b(l3), .clock(clk), .s(q1), .c(cq1) );
    csa7 add_2                              // Add 2. MSBs of x and y
    ( .a(l4), .b(l5), .clock(clk), .s(q2), .c(cq2) );
    add_ff8 add_3                           // Add MSBs of x and y
    ( .a(l6), .b(l7), .clock(clk), .s(q3) );

  //************** Second stage of the adder ******************
    // Two operands are zero
    assign h7 = {WIDTH2{1'b0}};
    assign h8 = {WIDTH3{1'b0}};

    lpm_ff reg_1                            // Save q0
    ( .data(q0), .q(v0), .clock(clk));  // Used ports
  //   .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
  //   .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_1.lpm_width = WIDTH0;

  // Add result of 2. LSBs (x+y) and carry from LSBs
    csa7cin add_4
    (.a(q1), .b(h7), .cin(cq0), .clock(clk), .s(v1),.c(cv1));

  // Add result of 2. MSBs (x+y) and carry from 2. LSBs
```

```
    csa7cin add_5
  (.a(q2), .b(h7), .cin(cq1), .clock(clk), .s(v2), .c(cv2));

// Add result of MSBs (x+y) and carry from 2. MSBs
    add_ff8cin add_6
    ( .a(q3), .b(h8), .cin(cq2), .clock(clk), .s(v3));

//*************** Third stage of the adder ******************
lpm_ff reg_2                          // Save v0
( .data(v0), .q(r0), .clock(clk));    // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_2.lpm_width = WIDTH0;

   lpm_ff reg_3                       // Save v1
   ( .data(v1), .q(r1), .clock(clk)); // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); // Unused
       defparam reg_3.lpm_width = WIDTH1;

// Add result of 2. MSBs (x+y) and carry from 2. LSBs
csa7cin add_7
( .a(v2), .b(h7), .cin(cv1), .clock(clk), .s(r2), .c(cr2));

// Add result of MSBs (x+y) and carry from 2. MSBs
add_ff8cin add_8
( .a(v3), .b(h8), .cin(cv2), .clock(clk), .s(r3) );

//************** Fourth stage of the adder ******************
   lpm_ff reg_4                       // Save r0
   ( .data(r0), .q(s0), .clock(clk)); // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); //Unused
       defparam reg_4.lpm_width = WIDTH0;

   lpm_ff reg_5                       // Save r1
   ( .data(r1), .q(s1), .clock(clk)); // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); //Unused
       defparam reg_5.lpm_width = WIDTH1;

   lpm_ff reg_6                       // Save r2
   ( .data(r2), .q(s2), .clock(clk)); // Used ports
//    .enable(ena), .aclr(aclr), .aset(aset), .sclr(sclr),
//    .sset(sset), .aload(aload), .sload(sload)); //Unused
```

```
    defparam reg_6.lpm_width = WIDTH2;

// Add result of MSBs (x+y) and carry from 2. MSBs
add_ff8cin add_9
( .a(r3), .b(h8), .cin(cr2), .clock(clk), .s(s3));

// Build a single output word of
// WIDTH = WIDTH0 + WIDTH1 + WIDHT2 + WIDTH3
assign s = {s3[WIDTH3 - 1:0], s2[WIDTH2 - 1:0],
                              s1[WIDTH1 - 1:0],s0[WIDTH0 - 1:0]};

assign sum = s ;      // Connect s to output pins

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: mul_ser.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module mul_ser (clk, x, a, y);   //----> Interface

  input          clk;
  input  [7:0]   x, a;
  output [15:0]  y;
  reg    [15:0]  y;


always @(posedge clk) //-> Multiplier in behavioral style
begin : States
  parameter s0=0, s1=1, s2=2;
  reg [2:0] count;
  reg [1:0] state;
  reg  [15:0] p, t;          // Double bit width
  reg  [7:0] a_reg;
  case (state)
    s0 : begin               // Initialization step
      a_reg <= a;
      state  <= s1;
      count = 0;
      p <= 0;                  // Product register reset
      t <={{8{x[7]}},x};   // Set temporary shift register
    end                                                    // to x
    s1 : begin               // Processing step
      if (count == 7)        // Multiplication ready
        state <= s2;
```

```
            else              // Note that MaxPlusII does not does
              begin           // not allow variable bit selects,
              if (a_reg[0] == 1) // see (LRM Sec. 4.2.1)
                p <= p + t;          // Add 2^k
              a_reg <= a_reg >> 1;// Use LSB for the bit select
              t <= t << 1;
              count = count + 1;
              state <= s1;
            end
          end
          s2 : begin          // Output of result to y and
            y <= p;           // start next multiplication
            state <= s0;
          end
        endcase
    end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: cordic.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module cordic (clk, x_in , y_in, r, phi, eps);

  parameter W = 7;   // Bit width - 1
  input             clk;
  input     [W:0] x_in, y_in;
  output    [W:0] r, phi, eps;
  reg       [W:0] r, phi, eps;

// There is no bit access in 2D array types
// in Verilog, therefore use single vectors
  reg [W:0] x0, y0, z0;
  reg [W:0] x1, y1, z1;
  reg [W:0] x2, y2, z2;
  reg [W:0] x3, y3, z3;

  always @(posedge clk) begin     //----> Infer register
    if (x_in > 0)                 // Test for x_in < 0 rotate
      begin                       // 0, +90, or - 90 degrees
      x0 <= x_in; // Input in register 0
      y0 <= y_in;
      z0 <= 0;
      end
```

```
else if (y_in > 0)
   begin
   x0 <= y_in;
   y0 <= - x_in;
   z0 <= 90;
   end
else
   begin
   x0 <= - y_in;
   y0 <= x_in;
   z0 <= - 90;
   end

if (y0 > 0)                    // Rotate 45 degrees
   begin
   x1 <= x0 + y0;
   y1 <= y0 - x0;
   z1 <= z0 + 45;
   end
else
   begin
   x1 <= x0 - y0;
   y1 <= y0 + x0;
   z1 <= z0 - 45;
   end

if (y1 > 0)                    // Rotate 26 degrees
   begin
   x2 <= x1 + {y1[W],y1[W:1]}; // i.e. x1 + y1 /2
   y2 <= y1 - {x1[W],x1[W:1]}; // i.e. y1 - x1 /2
   z2 <= z1 + 26;
   end
else
   begin
   x2 <= x1 - {y1[W],y1[W:1]}; // i.e. x1 - y1 /2
   y2 <= y1 + {x1[W],x1[W:1]}; // i.e. y1 + x1 /2
   z2 <= z1 - 26;
   end

if (y2 > 0)                    // Rotate 14 degrees
   begin
      x3 <= x2 + {y2[W],y2[W],y2[W:2]}; // i.e. x2 + y2/4
      y3 <= y2 - {x2[W],x2[W],x2[W:2]}; // i.e. y2 - x2/4
      z3 <= z2 + 14;
   end
```

```
      else
        begin
          x3 <= x2 - {y2[W],y2[W],y2[W:2]}; // i.e. x2 - y2/4
          y3 <= y2 + {x2[W],x2[W],x2[W:2]}; // i.e. y2 + x2/4
          z3 <= z2 - 14;
        end

      r   <= x3;
      phi <= z3;
      eps <= y3;
    end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: fir_gen.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
// This is a generic FIR filter generator
// It uses W1 bit data/coefficients bits
module fir_gen (clk, Load_x, x_in, c_in, y_out);

  parameter W1 = 9,     // Input bit width
            W2 = 18,    // Multiplier bit width 2*W1
            W3 = 19,    // Adder width = W2+log2(L)-1
            W4 = 11,    // Output bit width
            L  = 4,     // Filter length
            Mpipe = 3; // Pipeline steps of multiplier
  input clk, Load_x;   // std_logic
  input [W1 - 1:0] x_in, c_in;      // Inputs
  output [W3 - 1:0] y_out;          // Results

  reg [W1 - 1:0]   x;
  wire [W3 - 1:0]   y;
// 2D array types i.e. memories not supported by MaxPlusII
// in Verilog, use therefore single vectors
  reg   [W1 - 1:0] c0, c1, c2, c3; // Coefficient array
  wire [W2 - 1:0] p0, p1, p2, p3; // Product array
  reg   [W3 - 1:0] a0, a1, a2, a3; // Adder array

  wire   [W2 - 1:0] sum;   // Auxilary signals
  wire   clken, aclr;

  assign sum=0; assign aclr=0; // Default for mult
  assign clken=0;
```

```
//----> Load Data or Coefficient
   always @(posedge clk)
      begin: Load
      if (! Load_x) begin
         c3 <= c_in;       // Store coefficient in register
         c2 <= c3;         // Coefficients shift one
         c1 <= c2;
         c0 <= c1;
         end
      else begin
         x <= x_in; // Get one data sample at a time
      end
   end

//----> Compute sum-of-products
   always @(posedge clk)
      begin: SOP
   // Compute the transposed filter additions
      a0 <= {p0[W2 - 1], p0} + a1;
      a1 <= {p1[W2 - 1], p1} + a2;
      a2 <= {p2[W2 - 1], p2} + a3;
      a3 <= {p3[W2 - 1], p3}; // First TAP has only a register
   end
   assign y = a0;

// Instantiate L pipelined multiplier
   lpm_mult mul_0                    // Multiply    x*c0 = p0
      (.clock(clk), .dataa(x), .datab(c0), .result(p0));
//    .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
      defparam mul_0.lpm_widtha = W1;
      defparam mul_0.lpm_widthb = W1;
      defparam mul_0.lpm_widthp = W2;
      defparam mul_0.lpm_widths = W2;
      defparam mul_0.lpm_pipeline = Mpipe;
      defparam mul_0.lpm_representation = "SIGNED";

   lpm_mult mul_1                    // Multiply    x*c1 = p1
      (.clock(clk), .dataa(x), .datab(c1), .result(p1));
//    .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
      defparam mul_1.lpm_widtha = W1;
      defparam mul_1.lpm_widthb = W1;
      defparam mul_1.lpm_widthp = W2;
      defparam mul_1.lpm_widths = W2;
      defparam mul_1.lpm_pipeline = Mpipe;
```

```
    defparam mul_1.lpm_representation = "SIGNED";

  lpm_mult mul_2                    // Multiply    x*c2 = p2
    (.clock(clk), .dataa(x), .datab(c2), .result(p2));
//    .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_2.lpm_widtha = W1;
    defparam mul_2.lpm_widthb = W1;
    defparam mul_2.lpm_widthp = W2;
    defparam mul_2.lpm_widths = W2;
    defparam mul_2.lpm_pipeline = Mpipe;
    defparam mul_2.lpm_representation = "SIGNED";

  lpm_mult mul_3                    // Multiply    x*c3 = p3
    (.clock(clk), .dataa(x), .datab(c3), .result(p3));
//    .sum(sum), .clken(clken), .aclr(aclr)); // Unused ports
    defparam mul_3.lpm_widtha = W1;
    defparam mul_3.lpm_widthb = W1;
    defparam mul_3.lpm_widthp = W2;
    defparam mul_3.lpm_widths = W2;
    defparam mul_3.lpm_pipeline = Mpipe;
    defparam mul_3.lpm_representation = "SIGNED";

    assign y_out = y[W3 - 1:W3 - W4];

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: fir_srg.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module fir_srg (clk, x, y);   //----> Interface

  input        clk;
  input  [7:0] x;
  output [7:0] y;
  reg    [7:0] y;
// Tapped delay line array of bytes
  reg    [7:0] tap0, tap1, tap2, tap3;
// For bit access use single vectors in Verilog

  always @(posedge clk)   //----> Behavioral Style
  begin : p1
   // Compute output y with the filter coefficients weight.
   // The coefficients are [ - 1   3.75   3.75   - 1].
   // Multiplication and division for Altera MaxPlusII can
```

```
    // be done in Verilog with sign extensions and shifts!
     y <= (tap1<<1) + tap1 + {tap1[7],tap1[7:1]}
            + {tap1[7],tap1[7],tap1[7:2]} + (tap2<<1) + tap2
            + {tap2[7],tap2[7:1]}
            + {tap2[7],tap2[7],tap2[7:2]} - tap3 - tap0;

     tap3 <= tap2;   // Tapped delay line: shift one
     tap2 <= tap1;
     tap1 <= tap0;
     tap0 <= x;      // Input in register 0
  end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: dafsm.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
'include "case3.v" // User defined component

module dafsm (clk, x_in0, x_in1, x_in2, y); //--> Interface
 input            clk;
 input   [2:0]    x_in0, x_in1, x_in2;
 output [5:0]     y;
 reg     [5:0]    y;
 reg     [2:0]    x0, x1, x2;
 wire    [2:0]    table_in, table_out;

 reg [5:0] p;   // temporary register

 assign table_in[0] = x0[0];
 assign table_in[1] = x1[0];
 assign table_in[2] = x2[0];

 always @(posedge clk)   //----> DA in behavioral style
 begin : DA
   parameter s0=0, s1=1;
   reg [0:0] state;
   reg [1:0] count;     // Counts the shifts
   case (state)
     s0 : begin           // Initialization
        state <= s1;
        count = 0;
        p   <= {6{1'b0}};
```

```
      x0 <= x_in0;
      x1 <= x_in1;
      x2 <= x_in2;
   end
   s1 : begin                    // Processing step
      if (count == 3) begin      // Is sum of product done?
         y <= p;                 // Output of result to y and
         state <= s0;            // start next sum of product
      end
      else begin
         p <= {p[5],p[5:1]} + {1'b0,table_out,2'b00};
         x0[0] <= x0[1];
         x0[1] <= x0[2];
         x1[0] <= x1[1];
         x1[1] <= x1[2];
         x2[0] <= x2[1];
         x2[1] <= x2[2];
         count = count + 1;
         state <= s1;
      end
   end
      endcase
   end

   case3 LC_Table0
   ( .table_in(table_in), .table_out(table_out));

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: case3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module case3 (table_in, table_out);
   input     [2:0] table_in;    //Three bit
   output    [2:0] table_out;   //Range 0 to 6

   reg [2:0] table_out;

// This is the DA CASE table for
// the 3 coefficients: 2, 3, 1

   always @(table_in)
   begin
      case (table_in)
```

```
        0 :     table_out = 0;
        1 :     table_out = 2;
        2 :     table_out = 3;
        3 :     table_out = 5;
        4 :     table_out = 1;
        5 :     table_out = 3;
        6 :     table_out = 4;
        7 :     table_out = 6;
        default : ;
      endcase
    end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: case5p.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module case5p (clk, table_in, table_out);

  input         clk;
  input  [4:0] table_in;
  output [4:0] table_out;        // range 0 to 25

  reg [4:0] table_out;
  reg [3:0] lsbs;
  reg [1:0] msbs0;
  reg [4:0] table0out00, table0out01;

// These are the distributed arithmetic CASE tables for
// the 5 coefficients: 1, 3, 5, 7, 9

  always @(posedge clk) begin
    lsbs[0] = table_in[0];
    lsbs[1] = table_in[1];
    lsbs[2] = table_in[2];
    lsbs[3] = table_in[3];
    msbs0[0] = table_in[4];
    msbs0[1] = msbs0[0];
  end

// This is the final DA MPX stage.
  always @(posedge clk) begin
    case (msbs0[1])
      0 : table_out <= table0out00;
```

```
            1 : table_out <= table0out01;
            default : ;
        endcase
    end

// This is the DA CASE table 00 out of 1.
    always @(posedge clk) begin
        case (lsbs)
            0   : table0out00 = 0;
            1   : table0out00 = 1;
            2   : table0out00 = 3;
            3   : table0out00 = 4;
            4   : table0out00 = 5;
            5   : table0out00 = 6;
            6   : table0out00 = 8;
            7   : table0out00 = 9;
            8   : table0out00 = 7;
            9   : table0out00 = 8;
            10 : table0out00 = 10;
            11 : table0out00 = 11;
            12 : table0out00 = 12;
            13 : table0out00 = 13;
            14 : table0out00 = 15;
            15 : table0out00 = 16;
            default ;
        endcase
    end

// This is the DA CASE table 01 out of 1.
    always @(posedge clk) begin
        case (lsbs)
            0   : table0out01 = 9;
            1   : table0out01 = 10;
            2   : table0out01 = 12;
            3   : table0out01 = 13;
            4   : table0out01 = 14;
            5   : table0out01 = 15;
            6   : table0out01 = 17;
            7   : table0out01 = 18;
            8   : table0out01 = 16;
            9   : table0out01 = 17;
            10 : table0out01 = 19;
            11 : table0out01 = 20;
            12 : table0out01 = 21;
            13 : table0out01 = 22;
```

```
        14 : table0out01 = 24;
        15 : table0out01 = 25;
        default ;
      endcase
    end

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: darom.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
//include "220model.v"

module darom (clk, x_in0, x_in1, x_in2, y); //--> Interface

   input           clk;
   input    [2:0]  x_in0, x_in1, x_in2;
   output   [5:0]  y;
   reg      [5:0]  y;
   reg      [2:0]  x0, x1, x2;
   wire     [2:0]  table_in, table_out;

   reg [5:0] p;   // Temporary register
   wire ena;

   assign ena=1;

   assign table_in[0] = x0[0];
   assign table_in[1] = x1[0];
   assign table_in[2] = x2[0];

   always @(posedge clk)   //----> DA in behavioral style
   begin : DA
      parameter s0=0, s1=1;
      reg [0:0] state;
      reg [1:0] count;     // Counts the shifts
      case (state)
        s0 : begin   // Initialization
           state <= s1;
           count = 0;
           p   <= 0;
           x0 <= x_in0;
           x1 <= x_in1;
           x2 <= x_in2;
```

```
        end
        s1 : begin                    // Processing step
          if (count == 3) begin       // Is sum of product done?
            y <= p;                   // Output of result to y and
            state <= s0;              // start next sum of product
          end
          else begin
            p <= {p[5],p[5:1]} + {1'b0,table_out,2'b00};
            x0[0] <= x0[1];
            x0[1] <= x0[2];
            x1[0] <= x1[1];
            x1[1] <= x1[2];
            x2[0] <= x2[1];
            x2[1] <= x2[2];
            count = count + 1;
            state <= s1;
          end
        end
        default : ;
      endcase
    end

      lpm_rom rom_1
      (.address(table_in), .q(table_out));          // Used ports
    // .inclock(clk), .outclock(clk), .memenab(ena)); // Unused
        defparam rom_1.lpm_width = 3;
        defparam rom_1.lpm_widthad = 3;
        defparam rom_1.lpm_outdata = "UNREGISTERED";
        defparam rom_1.lpm_address_control = "UNREGISTERED";
        defparam rom_1.lpm_file = "darom3.mif";

    endmodule

    //*****************************************************************
    // IEEE STD 1364-1995 Verilog file: dasign.v
    // Author-EMAIL: Uwe.Meyer-Baese@ieee.org
    //*****************************************************************
    'include "case3s.v" // User defined component

    module dasign (clk, x_in0, x_in1, x_in2, y); //-> Interface

      input          clk;
      input  [3:0]   x_in0, x_in1, x_in2;
      output [6:0]   y;
      reg    [6:0]   y;
```

```
reg     [3:0]    x0, x1, x2;
wire    [2:0]    table_in;
wire    [3:0]    table_out;

reg [6:0] p;   // Temporary register

assign table_in[0] = x0[0];
assign table_in[1] = x1[0];
assign table_in[2] = x2[0];

always @(posedge clk)   //----> DA in behavioral style
begin : DA
   parameter s0=0, s1=1;
   integer k;
   reg [0:0] state;
   reg [2:0] count;                  // Counts the shifts
   case (state)
      s0 : begin                     // Initialization step
         state <= s1;
         count = 0;
         p  <= 0;
         x0 <= x_in0;
         x1 <= x_in1;
         x2 <= x_in2;
      end
      s1 : begin                     // Processing step
         if (count == 4) begin       // Is sum of product done?
            y <= p;                  // Output of result to y and
            state <= s0;             // start next sum of product
         end
         else begin       // Subtract for last accumulator step
            if (count ==3)     // i.e. p/2 +/ - table_out * 8
               p <= {p[6],p[6:1]} - (table_out << 3);
            else               // Accumulation for all other steps
               p <= {p[6],p[6:1]} + (table_out << 3);
            for (k=0; k<=2; k= k+1) begin        // Shift bits
               x0[k] <= x0[k+1];
               x1[k] <= x1[k+1];
               x2[k] <= x2[k+1];
            end
            count = count + 1;
            state <= s1;
         end
      end
   endcase
```

```
    end

    case3s LC_Table0
    ( .table_in(table_in), .table_out(table_out));

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: case3s.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module case3s (table_in, table_out);

   input  [2:0] table_in;  // Three bit
   output [3:0] table_out;  // Range - 2 to 4 -> 3 + sign bit

   reg [3:0] table_out;

// This is the DA CASE table for
// the 3 coefficients: - 2, 3, 1
   always @(table_in)
   begin
     case (table_in)
        0 :      table_out =  0;
        1 :      table_out = - 2;
        2 :      table_out =  3;
        3 :      table_out =  1;
        4 :      table_out =  1;
        5 :      table_out = - 1;
        6 :      table_out =  4;
        7 :      table_out =  2;
       default : ;
     endcase
   end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: dapara.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
`include "case3s.v" // User defined component

module dapara (clk, x_in, y); //----> Interface
```

```
input          clk;
input  [3:0]   x_in;
output [6:0]   y;
reg    [6:0]   y;

reg    [2:0] x0, x1, x2, x3;
wire   [3:0] y0, y1, y2, y3;
reg    [4:0] s0, s1;
reg    [3:0] t0, t1, t2, t3;

always @(posedge clk)   //----> DA in behavioral style
begin : DA
  integer k;
  for (k=0; k<=1; k=k+1) begin        // Shift all four bits
      x0[k] <= x0[k+1];
      x1[k] <= x1[k+1];
      x2[k] <= x2[k+1];
      x3[k] <= x3[k+1];
  end
  x0[2] <= x_in[0];       // Load x_in in the
  x1[2] <= x_in[1];       // MSBs of register 2
      x2[2] <= x_in[2];
      x3[2] <= x_in[3];
      y <= {{3{y0[3]}},y0} + {{2{y1[3]}},y1,1'b0}
            + {y2[3],y2,2'b00} - (y3 << 3);
// Sign extensions, pipeline register, and adder tree:
//   t0 <= y0; t1 <= y1; t2 <= y2; t3 <= y3;
//   s0 <= {t0[3],t0} + (t1 << 1);
//   s1 <= {t2[3],t2} - (t3 <<1);
//   y  <= {{2{s0[4]}},s0} + (s1 << 2);
   end

   case3s LC_Table0 ( .table_in(x0), .table_out(y0));
   case3s LC_Table1 ( .table_in(x1), .table_out(y1));
   case3s LC_Table2 ( .table_in(x2), .table_out(y2));
   case3s LC_Table3 ( .table_in(x3), .table_out(y3));

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: iir.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
module iir ( x_in,         // Input
```

```
            y_out,      // Result
            clk);
  parameter W = 14;   // Bit width - 1
  input   [W:0] x_in;
  output [W:0] y_out;
  input          clk;

  reg [W:0] x, y;

// initial begin
//   y=0;
//   x=0;
// end

// Use FFs for input and recursive part
always @(posedge clk) begin       // Note: there is no signed
  x  <= x_in;                     // integer in Verilog
  y  <= x + {y[W],y[W:1]} + {{2{y[W]}},y[W:2]};
                                  // i.e. x + y / 2 + y / 4;
end
assign  y_out = y;                // Connect y to output pins

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: iir_pipe.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
module iir_pipe (x_in, y_out, clk); //----> Interface

  parameter W = 14; // Bit width - 1
  input          clk;
  input  [W:0]   x_in;    // Input
  output [W:0]   y_out;   // Result

  reg [W:0] x, x3, sx;
  reg [W:0] y, y9;

  always @(posedge clk)    // Infer FFs for input, output and
  begin                    // pipeline stages;
    x   <= x_in;           // use non-blocking FF assignments
    x3  <= {x[W],x[W:1]} + {x[W],x[W],x[W:2]};
                           // i.e. x / 2 + x / 4 = x*3/4
    sx  <= x + x3; // Sum of x element i.e. output FIR part
```

```
    y9  <= {y[W],y[W:1]} + {{4{y[W]}},y[W:4]};
                                    // i.e. y / 2 + y / 16 = y*9/16
    y   <= sx + y9;                                  // Compute output
  end

  assign y_out = y ;    // Connect register y to output pins

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: iir_par.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module iir_par (clk, x_in, clk2, y_out); //----> Interface

  parameter W = 14; // bit width - 1
  input          clk;
  input  [W:0]   x_in;
  output [W:0]   y_out;
  output         clk2;

reg [W:0] x_even, x_odd, xd_odd, x_wait;
reg [W:0] y_even, y_odd, y_wait, y;
reg [W:0] x_e, x_o, y_e, y_o;
reg [W:0] sum_x_even, sum_x_odd;
reg       clk_div2;

always @(posedge clk)              // Split x into even
begin : Multiplex                  // and odd samples;
  parameter even=0, odd=1;         // recombine y at clk rate
  reg [0:0] state;
  case (state)
    even : begin
           x_even <= x_in;
           x_odd <= x_wait;
           clk_div2 <= 1;
           y <= y_wait;
           state <= odd;
    end
    odd : begin
          x_wait <= x_in;
          y <= y_odd;
          y_wait <= y_even;
          clk_div2 <= 0;
          state <= even;
```

```
      end
    endcase
end

assign y_out = y;
assign clk2  = clk_div2;

always @(negedge clk_div2)
begin: Arithmetic
  sum_x_even <= x_odd + {x_even[W],x_even[W:1]}
                      + {x_even[W],x_even[W],x_even[W:2]};
                    // i.e. x_odd + x_even / 2 + x_even /4
  y_even <= sum_x_even + {y_even[W],y_even[W:1]}
                           + {{4{y_even[W]}},y_even[W:4]};
                // i.e. sum_x_even + y_even / 2 + y_even /16
  xd_odd <= x_odd;
  sum_x_odd <= x_even + {xd_odd[W],xd_odd[W:1]}
                      + {xd_odd[W],xd_odd[W],xd_odd[W:2]};
                    // i.e. x_even + xd_odd / 2 + xd_odd /4
  y_odd  <= sum_x_odd + {y_odd[W],y_odd[W:1]}
                           + {{4{y_odd[W]}},y_odd[W:4]};
                // i.e. sum_x_odd + y_odd / 2 + y_odd / 16
  end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: cic3r32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module cic3r32 (clk, x_in, y_out); //----> Interface

  input          clk;
  input  [7:0]   x_in;
  output [8:0]   y_out;
  reg    [7:0]   y;

  parameter hold=0, sample=1;
  reg [1:0] state;
  reg [4:0]  count;
  reg  clk2;
  reg  [7:0]  x;          // Registered input
  wire [25:0] sxtx;     // Sign extended input
  reg  [25:0] i0, i1 , i2;              // I section   0, 1, and 2
  reg  [25:0] i2d1, i2d2, i2d3, i2d4, c1, c0; // I + COMB 0
```

```
reg   [25:0] c1d1, c1d2, c1d3, c1d4, c2; // COMB section 1
reg   [25:0] c2d1, c2d2, c2d3, c2d4, c3; // COMB section 2

always @(negedge clk)
begin : FSM
  case (state)
    hold : begin
      if (count < 31)
          state <= hold;
      else
          state <= sample;
      end
    default :
      state <= hold;
  endcase
end

assign sxtx = {{18{x[7]}},x};

always @(posedge clk)
begin : Int
    x     <= x_in;
    i0    <= i0 + sxtx;
    i1    <= i1 + i0 ;
    i2    <= i2 + i1 ;
  case (state)
    sample : begin
      c0    <= i2;
      count <= 0;
    end
    default :
      count <= count + 1;
  endcase
  if ((count > 8) && (count < 16))
    clk2   <= 1;
  else
    clk2   <= 0;
end

always @(posedge clk2)
begin : Comb
    i2d1 <= c0;
    i2d2 <= i2d1;
    c1    <= c0 - i2d2;
    c1d1 <= c1;
```

```
        c1d2 <= c1d1;
        c2   <= c1 - c1d2;
        c2d1 <= c2;
        c2d2 <= c2d1;
        c3   <= c2 - c2d2;
    end

    assign y_out = c3[25:17];

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: cic3s32.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module cic3s32 (clk, x_in, y_out); //----> Interface

    input          clk;
    input  [7:0]   x_in;
    output [8:0]   y_out;
    reg    [7:0]   y;

    parameter hold=0, sample=1;
    reg [1:0] state;
    reg [4:0]   count;
    reg   clk2;
    reg   [7:0]   x;                          // Registered input
    wire  [25:0] sxtx;                        // Sign extended input
    reg   [25:0] i0;                          // I section 0
    reg   [20:0] i1;                          // I section 1
    reg   [15:0] i2;                          // I section 2
    reg   [13:0] i2d1, i2d2, i2d3, i2d4, c1, c0; // I + COMB 0
    reg   [12:0] c1d1, c1d2, c1d3, c1d4, c2;  // COMB section 1
    reg   [11:0] c2d1, c2d2, c2d3, c2d4, c3;  // COMB section 2

always @(negedge clk)
begin : FSM
    case (state)
      hold : begin
        if (count < 31)
            state <= hold;
        else
            state <= sample;
        end
      default :
```

```
            state <= hold;
        endcase
    end

    assign sxtx = {{18{x[7]}},x};

    always @(posedge clk)
    begin : Int
            x    <= x_in;
            i0   <= i0 + sxtx;
            i1   <= i1 + i0[25:5];
            i2   <= i2 + i1[20:5];
        case (state)
            sample : begin
                c0      <= i2[15:2];
                count <= 0;
            end
            default :
                count <= count + 1;
        endcase
        if ((count > 8) && (count < 16))
            clk2   <= 1;
        else
            clk2   <= 0;
    end

    always @(posedge clk2)
    begin : Comb
            i2d1 <= c0;
            i2d2 <= i2d1;
            c1     <= c0 - i2d2;
            c1d1 <= c1[13:1];
            c1d2 <= c1d1;
            c2     <= c1[13:1] - c1d2;
            c2d1 <= c2[12:1];
            c2d2 <= c2d1;
            c3     <= c2[12:1] - c2d2;
    end

    assign y_out = c3[11:3];

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: db4poly.v
```

```
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
module db4poly (clk, x_in, clk2, x_e, x_o, g0, g1, y_out);

  input          clk;
  output         clk2;
  input  [7:0]   x_in;
  output [16:0]  x_e, x_o, g0, g1; // Test signals
  output [8:0]   y_out;

  reg  [7:0] x_odd, x_even, x_wait;
  wire [16:0] x_odd_sxt, x_even_sxt;
  reg  clk_div2;

// Register for multiplier, coefficients, and taps
  reg  [16:0] m0, m1, m2, m3, r0, r1, r2, r3;
  reg  [16:0] x33, x99, x107;
  reg  [16:0] y;

  always @(posedge clk)   // Split into even and odd
  begin : Multiplex       // samples at clk rate
    parameter even=0, odd=1;
    reg [0:0] state;
    case (state)
      even : begin
        x_even <= x_in;
        x_odd  <= x_wait;
        clk_div2 = 1;
        state <= odd;
      end
      odd : begin
        x_wait <= x_in;
        clk_div2 = 0;
        state <= even;
      end
    endcase
  end

  assign x_odd_sxt = {{9{x_odd[7]}},x_odd};
  assign x_even_sxt = {{9{x_even[7]}},x_even};

  always @(x_odd_sxt or x_even_sxt)
  begin : RAG
// Compute auxiliary multiplications of the filter
    x33   = (x_odd_sxt << 5) + x_odd_sxt;
```

```
      x99   = (x33 << 1) + x33;
      x107 = x99 + (x_odd_sxt << 3);
// Compute all coefficients for the transposed filter
      m0 = (x_even_sxt << 7) - (x_even_sxt << 2);  // m0 = 124
      m1 = x107 << 1;                              // m1 = 214
      m2 = (x_even_sxt << 6) - (x_even_sxt << 3)
                                    + x_even_sxt;  // m2 =  57
      m3 = x33;                                    // m3 = - 33
   end

   always @(negedge clk_div2) // Infer registers;
   begin : AddPolyphase          // use non-blocking assignments
//---------- Compute filter G0
      r0 <=  r2 + m0;            // g0 = 128
      r2 <=  m2;                 // g2 = 57
//---------- Compute filter G1
      r1 <=  - r3 + m1;          // g1 = 214
      r3 <=  m3;                 // g3 = - 33
// Add the polyphase components
      y <= r0 + r1;
   end

// Provide some test signal as outputs
   assign x_e = x_even;
   assign x_o = x_odd;
   assign clk2 = clk_div2;
   assign g0 = r0;
   assign g1 = r1;

   assign y_out = y[16:8]; // Connect y / 256 to output

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: db4latti.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
module db4latti (clk, x_in, clk2, x_e, x_o, g, h);

   input          clk;
   output         clk2;
   input  [7:0]   x_in;
   output [16:0]  x_e, x_o;
   output [8:0]   g, h;
   reg    [8:0]   g, h;
```

```
reg   [7:0]  x_wait;
wire  [16:0] x_wait_sxt, x_in_sxt;
reg   [16:0] sx_up, sx_low;
wire  [24:0] sx_up_sxt, sx_low_sxt;
reg   clk_div2;
wire [16:0] sxa0_up, sxa0_low;
wire [16:0] up0, up1, low1;
reg  [16:0] low0;
wire [24:0] up0_sxt, low0_sxt;

assign x_in_sxt = {{9{x_in[7]}},x_in};
assign x_wait_sxt = {{9{x_wait[7]}},x_wait};

always @(posedge clk)       // Split into even and odd
begin : Multiplex           // samples at clk rate
  parameter even=0, odd=1;
  reg [0:0] state;
  case (state)
    even : begin
      // Multiply with 256*s=124
      sx_up    <= (x_in_sxt << 7) - (x_in_sxt << 2);
      sx_low   <= (x_wait_sxt << 7) - (x_wait_sxt << 2);
      clk_div2 <= 1;
      state <= odd;
    end
    odd : begin
      x_wait <= x_in;
      clk_div2 <= 0;
      state <= even;
    end
  endcase
end

//******** Multipy a[0] = 1.7321
  assign sx_up_sxt = {{8{sx_up[16]}},sx_up};
  assign sx_low_sxt = {{8{sx_low[16]}},sx_low};
                                          // i.e. sign extensions
// Compute: (2*sx_up   - sx_up /4)-(sx_up /64 + sx_up /256)
  assign sxa0_up   = ((sx_up_sxt << 1)  - (sx_up_sxt >> 2))
                   - ((sx_up_sxt >> 6) + (sx_up_sxt >> 8));
// Compute: (2*sx_low - sx_low/4) - (sx_low/64 + sx_low/256)
  assign sxa0_low = ((sx_low_sxt << 1) - (sx_low_sxt >> 2))
                  - ((sx_low_sxt >> 6) + (sx_low_sxt >> 8));
//******** First stage -- FF in lower tree
```

```
    assign up0 = sxa0_low + sx_up;
    always @(negedge clk_div2)
    begin: LowerTreeFF
        low0 <= sx_low - sxa0_up;
    end

//******** Second stage: a[1]=0.2679
// Compute:     (up0 - low0/4) - (low0/64 + low0/256);
    assign up0_sxt = {{8{up0[16]}},up0};
    assign low0_sxt = {{8{low0[16]}},low0};
    assign up1    = (up0_sxt - (low0_sxt >> 2))
                    - ((low0_sxt >> 6) + (low0_sxt >> 8));
// Compute: (low0 + up0/4) + (up0/64    +    up0/256)
    assign low1 = (low0_sxt + (up0_sxt >> 2))
                        + ((up0_sxt >> 6) + (up0_sxt >> 8));

    assign x_e    = sx_up;          // Provide some extra
    assign x_o    = sx_low;         // test signals
    assign clk2 = clk_div2;

    always @(negedge clk_div2)
    begin: OutputScale
        g <= up1[16:8];          // i.e. up1 / 256
        h <= low1[16:8];         // i.e. low1 / 256;
    end

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: rader7.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module rader7 (clk, x_in, y_real, y_imag); //---> Interface

    input           clk;
    input   [7:0]   x_in;
    output  [10:0]  y_real, y_imag;
    reg     [10:0]  y_real, y_imag;

    reg [10:0]   accu;                          // Signal for X[0]
    // Note: No direct bit access of 2D vector in Verilog
    // use auxiliary signal for this purpose
    reg [18:0]   imag0, imag1, imag2, imag3, imag4, imag5,
                 real0, real1, real2, real3, real4, real5;
                                        // Tapped delay line array
```

```
reg [18:0]   x57, x111, x160, x200, x231, x250 ;
                                          // The filter coefficients
reg [18:0]   x5, x25, x110, x125, x256;
                                       // Auxiliary filter coefficients
reg [7:0]    x, x_0;                   // Signals for x[0]
wire [18:0] x_sxt, x_0_sxt;

assign x_sxt = {{9{x[7]}},x}; // Sign extension of input
assign x_0_sxt = {{9{x_0[7]}},x_0};              // and x[0]

always @(posedge clk)   // State machine for RADER filter
begin : States
  paramcter Start=0, Load=1, Run=2;
  reg [1:0] state;
  reg [4:0] count;
  case (state)
    Start : begin            // Initialization step
      state <= Load;
      count <= 1;
      x_0 <= x_in;           // Save x[0]
      accu <= 0 ;            // Reset accumulator for X[0]
      y_real   <= 0;
      y_imag   <= 0;
end
Load : begin     // Apply x[5],x[4],x[6],x[2],x[3],x[1]
  if (count == 8)        // Load phase done ?
    state <= Run;
  else begin
    state <= Load;
    accu <= accu + x_sxt;
  end
  count <= count + 1;
end
Run : begin     // Apply again x[5],x[4],x[6],x[2],x[3]
  if (count == 15) begin // Run phase done ?
    y_real   <= accu;            // X[0]
    y_imag  <= 0;   // Only re inputs i.e. Im(X[0])=0
    state <= Start;          // Output of result
  end                                // and start again
  else begin
    y_real   <= (real0 >> 8) + x_0_sxt;
                                       // i.e. real[0]/256+x[0]
    y_imag   <= (imag0 >> 8);   // i.e. imag[0]/256
    state <= Run;
  end
```

```
        count <= count + 1;
      end
    endcase
end

always @(posedge clk)     // Structure of the two FIR
begin : Structure         // filters in transposed form
  x <= x_in;
  // Real part of FIR filter in transposed form
  real0 <= real1 + x160  ;     // W^1
  real1 <= real2 - x231  ;     // W^3
  real2 <= real3 - x57   ;     // W^2
  real3 <= real4 + x160  ;     // W^6
  real4 <= real5 - x231  ;     // W^4
  real5 <= -x57;               // W^5

  // Imaginary part of FIR filter in transposed form
  imag0 <= imag1 - x200  ;     // W^1
  imag1 <= imag2 - x111  ;     // W^3
  imag2 <= imag3 - x250  ;     // W^2
  imag3 <= imag4 + x200  ;     // W^6
  imag4 <= imag5 + x111  ;     // W^4
  imag5 <= x250;               // W^5
end

always @(posedge clk)
begin : Coeffs //Note that all signals are globally defined
// Compute the filter coefficients and use FFs
  x160   <= x5 << 5;           // i.e. 160 = 5 * 32;
  x200   <= x25 << 3;          // i.e. 200 = 25 * 8;
  x250   <= x125 << 1;         // i.e. 250 = 125 * 2;
  x57    <= x25 + (x << 5);    // i.e. 57 = 25 + 32;
  x111   <= x110 + x;          // i.e. 111 = 110 + 1;
  x231   <= x256 - x25;        // i.e. 231 = 256 - 25;
end

always  @(x_sxt or x5 or x25)      // Note that all signals
begin : Factors                    // are globally defined
// Compute the auxiliary factor for RAG without an FF
  x5     = (x_sxt << 2) + x_sxt;   // i.e. 5 = 4 + 1;
  x25    = (x5 << 2) + x5;         // i.e. 25 = 5*4 + 5;
  x110   = (x25 << 2) + (x5 << 2); // i.e. 110 = 25*4+5*4;
  x125   = (x25 << 2) + x25;       // i.e. 125 = 25*4+25;
  x256   = x_sxt << 8;             // i.e. 256 = 2 ** 8;
end
```

```
endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: ccmul.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
//`include "220model.v"

module ccmul (clk, x_in, y_in, c_in,
                  cps_in, cms_in, r_out, i_out);

  parameter W2 = 17,     // Multiplier bit width
            W1 = 9,      // Bit width c+s sum
            W  = 8;      // Input bit width
  input clk;   // Clock for the output register
  input [W - 1:0] x_in, y_in, c_in;       // Inputs
  input [W1 - 1:0]   cps_in, cms_in;      // Inputs
  output [W - 1:0]      r_out, i_out;     // Results
  reg    [W - 1:0]      r_out, i_out;     // Results


  wire [W - 1:0] x, y, c ;                        // Inputs and outputs
  wire [W2 - 1:0] r, i, cmsy, cpsx, xmyc, sum;    // Products
  wire [W1 - 1:0] xmy, cps, cms, sxtx, sxty;      // x-y etc.


  wire   clken, cr1, ovl1, cin1, aclr, ADD, SUB;
                                                  // Auxiliary signals
  assign cin1=0; assign aclr=0; assign ADD=1; assign SUB=0;
  assign cr1=0; assign sum=0; assign clken=0;
                                                  // Default for add
  assign x   = x_in;   // x
  assign y   = y_in;   // j * y
  assign c   = c_in;   // cos
  assign cps = cps_in; // cos + sin
  assign cms = cms_in; // cos - sin

  always @(posedge clk) begin
    r_out <= r[W2 - 2:W];          // Scaling and FF for output
    i_out <= i[W2 - 2:W];
  end

//********* ccmul with 3 mul. and 3 add/sub  **************
  assign sxtx  = {x[W - 1],x};     // Possible growth for
  assign sxty  = {y[W - 1],y};     // sub_1 -> sign extension
```

```
    lpm_add_sub sub_1                          // Sub:   x - y
    ( .result(xmy), .dataa(sxtx), .datab(sxty));// Used ports
//    .add_sub(SUB), .cout(cr1), .overflow(ovl1), .cin(cin1),
//    .clken(clken), .clock(clk), .aclr(aclr));   // Unused
        defparam sub_1.lpm_width = W1;
        defparam sub_1.lpm_representation = "SIGNED";
        defparam sub_1.lpm_direction = "sub";

    lpm_mult mul_1                             // Multiply   (x - y)*c = xmyc
    ( .dataa(xmy), .datab(c), .result(xmyc)); // Used ports
//    .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                                        // Unused ports
        defparam mul_1.lpm_widtha = W1;
        defparam mul_1.lpm_widthb = W;
        defparam mul_1.lpm_widthp = W2;
        defparam mul_1.lpm_widths = W2;
        defparam mul_1.lpm_representation = "SIGNED";

    lpm_mult mul_2                             // Multiply (c - s)*y = cmsy
    ( .dataa(cms), .datab(y), .result(cmsy)); // Used ports
//    .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                                        // Unused ports
        defparam mul_2.lpm_widtha = W1;
        defparam mul_2.lpm_widthb = W;
        defparam mul_2.lpm_widthp = W2;
        defparam mul_2.lpm_widths = W2;
        defparam mul_2.lpm_representation = "SIGNED";

    lpm_mult mul_3                             // Multiply (c+s)*x = cpsx
    ( .dataa(cps), .datab(x), .result(cpsx)); // Used ports
//    .sum(sum), .clock(clk), .clken(clken), .aclr(aclr));
                                                        // Unused ports
        defparam mul_3.lpm_widtha= W1;
        defparam mul_3.lpm_widthb = W;
        defparam mul_3.lpm_widthp = W2;
        defparam mul_3.lpm_widths = W2;
        defparam mul_3.lpm_representation = "SIGNED";

    lpm_add_sub add_1              // Add:   r <= (x - y)*c + (c-s)*y
    ( .dataa(cmsy), .datab(xmyc), .result(r));   // Used ports
//    .add_sub(ADD), .cout(cr1), .overflow(ovl1), .cin(cin1),
//    .clken(clken), .clock(clk), .aclr(aclr));   // Unused
        defparam add_1.lpm_width = W2;
        defparam add_1.lpm_representation = "SIGNED";
```

```
        defparam add_1.lpm_direction = "add";

    lpm_add_sub sub_2                  // Sub: i <= (c+s)*x - (x - y)*c
    ( .dataa(cpsx), .datab(xmyc), .result(i));    // Used ports
// .add_sub(SUB), .cout(cr1), .overflow(ovl1), .clock(clk),
// .cin(cin1),    .clken(clken), .aclr(aclr));    // Unused
        defparam sub_2.lpm_width = W2;
        defparam sub_2.lpm_representation = "SIGNED";
        defparam sub_2.lpm_direction = "sub";

endmodule

//*************************************************************
// IEEE STD 1364-1995 Verilog file: bfproc.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*************************************************************
//`include "220model.v"
//`include "ccmul.v"

module bfproc (clk, Are_in, Aim_in, Bre_in, Bim_in, c_in,
          cps_in, cms_in, Dre_out, Dim_out, Ere_out, Eim_out);

parameter W2 = 17,     // Multiplier bit width
          W1 = 9,      // Bit width c+s sum
          W  = 8;      // Input bit width
input clk;   // Clock for the output register
input [W - 1:0] Are_in, Aim_in;          // 8-bit inputs
input [W - 1:0] Bre_in, Bim_in, c_in;    // 8-bit inputs
input [W1 - 1:0]   cps_in, cms_in;       // 9-bit coefficients
output [W - 1:0]   Dre_out, Dim_out, Ere_out, Eim_out;
reg    [W - 1:0]   Dre_out, Dim_out;     // 8-bit registered
                                         // results
reg   [W   1:0] dif_re, dif_im;          // Bf out
reg   [W - 1:0] Are, Aim, Bre, Bim;      // Inputs as integers
reg   [W - 1:0] c;                         // Input
reg   [W1 - 1:0] cps, cms;                 // Coefficient in

always @(posedge clk)     // Compute the additions of the
begin                     // butterfly using integers
  Are     <= Are_in;      // and store inputs
  Aim     <= Aim_in;      // in flip-flops
  Bre     <= Bre_in;
  Bim     <= Bim_in;
  c       <= c_in;        // Load from memory cos
  cps     <= cps_in;      // Load from memory cos+sin
```

```
    cms      <= cms_in;       // Load from memory cos-sin
    Dre_out <= ({Are[W - 1],Are} + {Bre[W - 1],Bre}) >> 1;
                                                  // i.e. Are/2 + Bre/2
    Dim_out <= ({Aim[W - 1],Aim} + {Bim[W - 1],Bim}) >> 1;
end                                               // i.e. Aim/2 + Bim/2

     // No FF because butterfly difference "diff" is not an
always @(Are or Bre or Aim or Bim)                // output port
begin
    dif_re = ({Are[W - 1],Are} - {Bre[W - 1],Bre}) >> 1;
                                                  // i.e. Are/2 - Bre/2
    dif_im = ({Aim[W - 1],Aim} - {Bim[W - 1],Bim}) >> 1;
end                                               // i.e. Aim/2 - Bim/2

//*** Instantiate the complex twiddle factor multiplier
ccmul ccmul_1                                     // Multiply (x+jy)(c+js)
( .clk(clk), .x_in(dif_re), .y_in(dif_im),    .c_in(c),
  .cps_in(cps), .cms_in(cms), .r_out(Ere_out),
                                                      .i_out(Eim_out));

endmodule
```

```
//*****************************************************************
// IEEE STD 1364-1995 Verilog file: lfsr.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module lfsr (clk, y); //----> Interface

   input            clk;
   output [6:1]     y;   // Result

   reg [6:1] ff; // Note that reg is keyword in Verilog and
                                          // can not be variable name
   integer i;

   always @(posedge clk) begin // Length 6 LFSR with xnor
      ff[1] <= ff[5] ~^ ff[6]; // Use non-blocking assignment
      for (i=6; i>=2 ; i=i - 1)   // Tapped delay line: shift one
         ff[i] <= ff[i - 1];
   end

   assign     y = ff;              // Connect to I/O cell

endmodule
```

```
//*****************************************************************
// IEEE STD 1364-1995 Verilog file: lfsr6s3.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module lfsr6s3 (clk, y); //----> Interface

  input             clk;
  output [6:1]      y;   // Result

  reg [6:1] ff; // Note that reg is keyword in Verilog and
                                    // can not be variable name

  always @(posedge clk) begin       // Implement three-step
    ff[6] <= ff[3];                 // length-6 LFSR with xnor;
    ff[5] <= ff[2];                 // use non-blocking assignments
    ff[4] <= ff[1];
    ff[3] <= ff[5] ~^ ff[6];
    ff[2] <= ff[4] ~^ ff[5];
    ff[1] <= ff[3] ~^ ff[4];
  end

  assign   y = ff;

endmodule

//*****************************************************************
// IEEE STD 1364-1995 Verilog file: ammod.v
// Author-EMAIL: Uwe.Meyer-Baese@ieee.org
//*****************************************************************
module ammod (clk, r_in , phi_in,
                        x_out, y_out, eps); //----> Interface

  parameter W = 8;   // Bit width - 1
  input             clk;
  input   [W:0]  r_in, phi_in;
  output  [W:0]  x_out, y_out, eps;
  reg     [W:0]  x_out, y_out, eps;
  reg     [W:0]  r, phi;

  reg [W:0] x0, y0, z0; // There is no bit access in 2D
  reg [W:0] x1, y1, z1; // array types in Verilog,
  reg [W:0] x2, y2, z2; // therefore use single vectors
  reg [W:0] x3, y3, z3;

  always @(posedge clk) begin     //----> Infer register
```

```
 if (phi_in > 90)                    // Test for |phi_in| > 90
    begin                            // Rotate 90 degrees
    x0 <= 0;
    y0 <= r_in;                      // Input in register 0
    z0 <= phi_in -'d90;
    end
 else if ((phi_in > 331) && (phi_in < 423))
    begin
    x0 <= 0;
    y0 <= - r_in;
    z0 <= phi_in + 'd90;
    end
 else
    begin
    x0 <= r_in;
    y0 <= 0;
    z0 <= phi_in;
  end

 if (z0 > 0)                         // Rotate 45 degrees
   begin
    x1 <= x0 - y0;
    y1 <= y0 + x0;
    z1 <= z0 - 'd45;
    end
  else
    begin
    x1 <= x0 + y0;
    y1 <= y0 - x0;
    z1 <= z0 + 'd45;
    end

 if (z1 > 0)                         // Rotate 26 degrees
    begin
    x2 <= x1 - {y1[W],y1[W:1]}; // i.e. x1 - y1 /2
    y2 <= y1 + {x1[W],x1[W:1]}; // i.e. y1 + x1 /2
    z2 <= z1 - 'd26;
    end
 else
    begin
    x2 <= x1 + {y1[W],y1[W:1]}; // i.e. x1 + y1 /2
    y2 <= y1 - {x1[W],x1[W:1]}; // i.e. y1 - x1 /2
    z2 <= z1 + 'd26;
    end
```

```
        if (z2 > 0)                                    // Rotate 14 degrees
          begin
            x3 <= x2 - {y2[W],y2[W],y2[W:2]}; // i.e. x2 - y2/4
            y3 <= y2 + {x2[W],x2[W],x2[W:2]}; // i.e. y2 + x2/4
            z3 <= z2 - 'd14;
          end
        else
          begin
            x3 <= x2 + {y2[W],y2[W],y2[W:2]}; // i.e. x2 + y2/4
            y3 <= y2 - {x2[W],x2[W],x2[W:2]}; // i.e. y2 - x2/4
            z3 <= z2 + 'd14;
          end

        x_out <= x3;
        eps   <= z3;
        y_out <= y3;
      end

endmodule
```

附录B　VHDL和Verilog编码

遗憾的是，到目前为止我们看到只有两套流行的 HDL 语言。美国西海岸和亚洲倾向于使用 Verilog 语言，而美国东海岸和欧洲使用得更频繁的是 VHDL 语言。对于用 FPGA 进行数字信号处理而言，这两种语言都非常适用，但是 VHDL 示例更容易阅读一些，因为它支持 IEEE VHDL 1076-1987 和 1076-1993 标准中的有符号算法和乘/除运算。这种差异很快就会随着新的 Verilog IEEE 标准 1364-1999 得到批准而消失，这样 Verilog 也就包括了有符号算法。其他的约束条件就包括个人的偏好、EDA 库和可用的工具、数据类型、可读性、性能以及用 PLI 对语言进行扩展，还有就是商业和市场问题等。查阅 Smith 先生的著作[3]就可以看到详细的比较。工具提供商宣称目前对两种语言都给予支持。

这样，使用可以很容易转换成这两种语言的 HDL 代码就是一种非常好的想法。当命名变量、标签、常数和用户类型等时，在 HDL 代码中需要注意的一条重要规则就是要避免两种语言的关键字。IEEE 标准 VHDL 1076-1987 使用了 77 个关键字，而 VHDL1076-1993 中增加了 17 个关键字(请参阅 VHDL 1076-1993 语言参考手册，179 页) 。1076-1993 中新增的关键字有：

GROUP, IMPURE, INERTIAL, LITERAL, POSTPONED, PURE, REJECT, ROL, ROR, SHARED, SLA, SLL, SRA, SRL, UNAFFECTED, XNOR

遗憾的是这些关键字在 MaxPlusII 的编辑器中都没有加以强调。IEEE 标准 Verilog 1364-1995 有 102 个关键字(请参阅语言参考手册，604 页)。两种语言加起来共有 182 个关键字，其中包括 17 个共用的关键字。表 B-1 以大写字母的形式列出了 VHDL 1076-1993 的关键字，而 Verilog1364-1995 的关键字是以小写字母表示的，而共用的关键字则是以首字母大写的形式表示的。

表 B-1　VHDL 1076-1993 和 Verilog 1364-1995 的关键字

ABS	event	OF	OF
ACCESS	EXIT	ON	SLA
AFTER	FILE	OPEN	SLL
ALIAS	For	Or	small
ALL	force	OTHERS	specify
always	forever	OUT	specparam
And	fork	output	SRA
ARCHITECTURE	Function	PACKAGE	SRL
ARRAY	GENERATE	parameter	strong0
ASSERT	GENERIC	pmos	strong1
assign	GROUP	PORT	SUBTYPE
ATTRIBUTE	GUARDED	posedge	supply0
Begin	highz0	POSTPONED	supply1

（续表）

BLOCK	If	PROCEDURE	task
BODY	ifnone	PROCESS	THEN
buf	IMPURE	pull0	time
BUFFER	IN	pull1	TO
bufif0	INTERTIAL	pulldown	tran
bufif1	initial	pullup	tranif0
BUS	Inout	PURE	tranif1
Case	input	RANGE	TRANSPORT
casex	integer	rcmos	tri
casez	IS	real	tri0
cmos	join	realtime	tri1
COMPONENT	LABEL	RECORD	triand
CONFIGURATION	large	reg	trior
CONSTANT	LIBRARY	REGISTER	trireg
deassign	LINKAGE	REJECT	TYPE
default	LITERAL	release	UNAFFECTED
defparam	LOOP	REM	UNITS
disable	macromodule	repeat	UNTIL
DISCONNECT	MAP	REPORT	USE
DOWNTO	medium	RETURN	VARIABLE
edge	MOD	rnmos	vectored
Else	module	ROL	Wait
ELSIF	Nand	ROR	wand
End	negedge	rpmos	weak0
endcase	NEW	rtran	weak1
endfunction	NEXT	rtranif0	WHEN
endmodule	nmos	rtranif1	While
endprimitive	Nor	scalared	wire
endspecify	Not	SELECT	WITH
endtable	notif0	SEVERITY	wor
endtask	notif1	SHARED	Xnor
ENTITY	NULL	SIGNAL	Xor
event	OF	OF	

B.1　示例列表

下表给出了本书中用到的所有 VHDL 和 Verilog 示例结果。

设　计	VHDL 代码		Verilog 代码	
	LC	MHz	LC	MHz
add_1p	26	97.08	26	97.08
add_2p	58	94.33	51	94.33
add_3p	105	91.74	105	91.74
ammod	279	40.65	277	42.01
bfproc	533	18.38	542	18.65
ccmul	493	—	504	—
cic3r32	332	38.46	332	38.75
cic3s32	199	41.66	200	41.32
cordic	256	43.47	253	45.04
dafsm	37	61.72	37	58.13
dapara	39	42.19	39	42.19
draom	34	32.25	34	32.25
dasign	65	42.19	65	39.84
db4latti	334	60.60	324	63.69
db4poly	208	90.90	191	99.00
example	25	125.00	25	125.00
fir_gen	890	46.72	882	48.54
fir_srg	97	19.53	97	19.76
fun_text	32	59.17	32	59.17
iir	31	55.86	31	55.86
iir_par	213	58.13	212	60.24
iir_pipe	64	73.52	64	73.52
lfsr	6	100.00	6	100.00
lfsr6s3	6	95.23	6	95.23
mul_ser	111	45.87	137	45.87
rader7s	494	29.94	495	28.81

在本书中用到了 MaxPlusII 9.23 版的如下选项：

- Global Project Synthesis Style，选择 FAST 选项。
- Assign | Global Project Logic Synthesis 选项，选择 Optomize 10(速度)。
- Assign | Global Project Logic Synthesis | Automatic Fast I/O。
- Assign | Device，选择 Device Family。选择 FLEX10K 选项。

● Device | EPF10K20RC240-4

以 MHz 表示的数据是设计的 Registered Performance(从“定时分析器输出”文件，也就是 *.tao 文件中得到)。上表是按照如下方式构造的：第 1 列给出了设计的“实体”或是模型的名称。第 2 列到第 3 列是 VHDL 设计的数据：报告文件(*.rpt)中 LC 的数量以及 Registered Performance。第 4 列到第 5 列给出了 Verilog 设计的相同数据。比较 VHDL 和 Verilog 合成的结果，就可以看到一些数据不是完全相同的。

B.2 参数化的模块库(LPM)

在整本书中我们用到了 4 种不同的 LPM 兆函数(参阅图 B-1)，分别是：

- lpm_ff，触发器兆函数
- lpm_rom，ROM 兆函数
- lpm_add_sub，加法器/减法器兆函数
- lpm_mult，乘法器兆函数

下面给出了这些兆函数的解释及其端口的定义、参数和资源的使用。这些信息也可以使用 MaxPlusII 帮助在 VHDL→Megafunction/LPM 中找到。

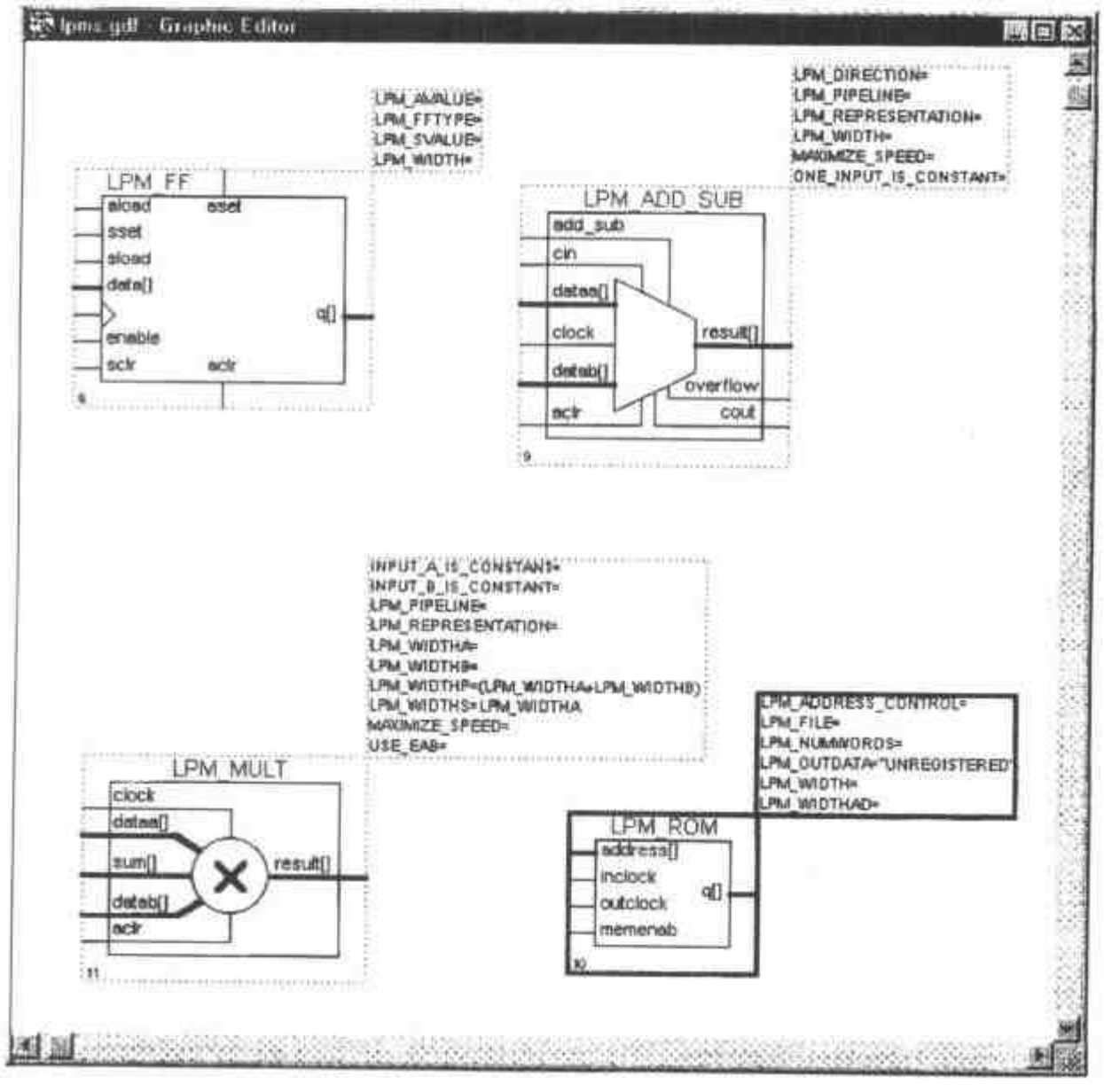


图 B-1　所使用的 4 种 LPM 兆函数

B.2.1　参数化的触发器兆函数(lpm_ff)

如果需要基本数据单元 DFF、DFFE、TFF 和 TFFE(如：同步和异步设置、清除和加载输入)中没有的特性时，就要用到 lpm_ff 函数。我们已经在 example1.1、fun_text、add_1p、add_2p、add_3p 以及组件 csa7 和 csa7cin 的设计中使用了这个兆函数。

Altera 推荐像“用 MegaWizard Plug-In Manager 创建定制兆函数变量”中所描述的那样来初始化这个函数。Verilog 中原型的端口名称和命令是：

```
module lpm_ff( q,
               data, clock, enable,
               aclr, aset,
               sclr, sset
               aload, sload);
```

VHDL 组件声明如下：

```
COMPONENT lpm_ff
    GENERIC (LPM_WIDTH:POSITIVE;
             LPM_AVALUE: STRING :="UNUSED";
             LPM_FFTYPE: STRING := "FFTYPE_DEF";
             LPM_TYPE: STRING := "L_FF";
             LPM_SVALUE: STRING := "UNUSED";
             LPM_HINT: STRING := "UNUSED";
    PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH - 1 DOWNTO 0);
          clock: IN STD_LOGIC;
          enable: IN STD_LOGIC := '1';
          sload: IN STD_LOGIC := '0';
          sclr: IN STD_LOGIC := '0';
          sset: IN STD_LOGIC := '0';
          aload: IN STD_LOGIC := '0';
          aclr: IN STD_LOGIC := '0';
          aset: IN STD_LOGIC := '0';
          q: OUT STD_LOGIC_VECTOR(LPM_WIDTH - 1 DOWNTO 0));
END COMPONENT;
```

1. 端口

下表给出了 lpm_ff 的所有 INPUT 端口。

端口名称	是否必需	描　　述	说　　明
data	否	T 类型触发器：触发器使能 DD 类型触发器：数据输入	输入端口 LPM_WIDTH 宽度。如果 data 输入未使用，则至少使用 aset、aclr、sset 或 sclr 端口中的　个。未使用的数据输入默认为 GND
clock	是	上升沿触发的时钟	无

(续表)

端口名称	是否必需	描　述	说　明
enable	否	时钟使能输入	默认值为 1
sclr	否	同步清除输入	如果 sset 和 sclr 均被使用且已经被声明，则 sclr 优先。在奇偶校验位被应用到端口前，sclr 信号影响输出 q 的值
sset	否	同步设置输入	如果值已经给出，或是设定 q 输出的值全是 1，则设定 q 输出的值为 LPM_SVALUE 指定的值。如果 sset 和 sclr 都已经使用且已经被声明，则 sclr 优先。在奇偶校验位被应用到端口前，sset 信号影响输出 q 的值
sload	否	同步加载输入，把下一次活动时钟沿上的 data 输入端的值加载到触发器	默认值为 0。如果使用 sload，则必须使用 data。加载操作时，sload 必须是高电平(1)且使能必须是高电平(1)或未连接。当 LPM_FFTYPE 参数设置为 DFF 时，就可以忽略 sload 端口
aclr	否	异步清除输入	如果 aset 和 aclr 均被使用，且已声明，则 aclr 优先。在奇偶校验位被应用到端口前，aclr 信号影响输出 q 值
asset	否	异步设置输入	如果值已经给出，或是设定 q 输出的值全是 1，则设置 q 输出的值为 LPM_AVALUE 指定的值
aload	否	异步加载输入。用数据输入端的值异步加载触发器	默认值为 0。如果 aload 已被使用，则 data 也必须被使用

下表给出了 lpm_ff 的所有 OUTPUT 端口。

端口名称	是否必需	描　述	说　明
q	是	来自 D 或 T 触发器的数据输出	输出端口 LPM_WIDTH 宽度

2. 参数

下表给出了 lpm_ff 组件的参数。

参　数	类　型	是否必需	说　明
LPM_WIDTH	整数	是	data 和 q 端的宽度
LPM_AVALUE	整数	否	当 aset 为高电平时，加载恒定的值。如果省略，默认值全部为 1。LPM_AVALUE 参数的最大值限制为 32 位
LPM_SVALUE	整数	否	当 sset 为高电平时，在 clock 的上升沿加载恒定的值。如果省略，默认值全部为 1

(续表)

参　数	类　型	是否必需	说　明
LPM_FFTYPE	字符串	否	值是“DFF”，“TFF”和“UNUSED”类型触发器。如果省略，则默认值是“DFF”。当 LPM_FFTYPE 参数被设置成“DFF”时，就忽略了 sload 端口
LPM_HINT	字符串	否	允许在 VHDL 设计文件中规定 Altera 特有的参数，默认值是“UNUSED”
LPM_TYPE	字符串	否	确定在 VHDL 设计文件中 LPM 的实体名称

注意：

对于 Verilog LPM 220 可合成代码(也就是 220model.v)，采用下面的参数命令：lpm_type，lpm_width，lpm_avalue，lpm_svalue，lpm_pvalue，lpm_fftype，lpm_hint。

3. 功能

下表就是 lpm_ff 中 T 型触发器运作的示例。

输入信号							输　出
aclr	aset	enable	clock	sclr	sset	sload	Q[LPM_WIDTH - 1..0]
1	X	X	X	X	X	X	000…
0	1	X	X	X	X	X	111…or LPM_AVALUE
0	0	0	X	X	X	X	q[LPM_WIDTH - 1..0]
0	0	1	∫	1	X	X	000…
0	0	1	∫	0	1	X	111…or LPM_SVALUE
0	0	1	∫	0	0	1	data[LPM_WIDTH - 1..0]
0	0	1	∫	0	0	0	q[LPM_WIDTH - 1..0] xor data[LPM_ WIDTH - 1..0]

4. 资源使用方法

此函数 lpm_ff 每位使用一个逻辑单元。

B.2.2　参数化的加法器/减法器兆函数(lpm_add_sub)

Altera 推荐使用 lpm_add_sub 函数代替所有其他类型的加法器/减法器函数，包括旧类型的加法器/减法器的宏函数。我们已经在 example1.1、fun_text、add_1p、ccmul 以及 add_ff8、add_ff8cin、csa7 和 csa7cin 组件的设计中使用了这个兆函数。

Altera 推荐像“用 MegaWizard Plug-In Manager 创建定制兆函数变量”中所描述的那样来初始化这个函数。

Verilog 中原型的端口名称和命令是：

```
module lpm_add_sub (cin,
                    dataa, datab,
                    add_sub, clock, aclr,
                    result, cout, overflow);
```

VHDL 组件的声明如下：

```
COMPONENT lpm_add_sub
  GENERIC (LPM_WIDTH:POSITIVE;
           LPM_REPRESENTATION:STRING:="SINGNED";
           LPM_DERECTION: STRING : = "UNUSED";
           LPM_HINT:STRING:= "UNUSED";
           LPM_PIPELINE:INTEGER:=0;
           LPM_TYPE:STRING:= "L_ADD_SUB");
  PORT(dataa, datab
           : IN STD_LOGIC_VECTOR(LPM_WIDTH - 1 DOWNTO 0);
      aclr,clken,clock,cin : IN STD_LOGIC:='0';
      add_sub              : IN STD_LOGIC:= '1';
      result:OUT STD_LOGIC_VECTOR(LPM_WIDTH - 1 DOWNTO 0);
      cout, overflow       : OUT STD_LOGIC);
END COMPONENT;
```

1. 端口

下表给出了 lpm_add_sub 的所有 INPUT 端口。

端口名称	是否必需	描　述	说　明
cin	否	到低阶位的进位。如果运算是“ADD”，则低＝0，高＝+1。如果运算是“SUB”，则低＝－1，高＝0	如果省略，默认值是 0(也就是如果运算是“ADD”，为低，如果运算是“SUB”，为高)
dataa	是	被加数/被减数	输入端口 LPM_WIDTH 宽度
datab	是	加数/减数	输入端口 LPM_WIDTH 宽度
add_sub	否	如果信号为高电平，则运算＝dataa+datab 如果信号为低电平，则运算＝dataa-datab	如果使用了 LPM_DIRECTION 参数，就不能使用 add_sub。如果省略，则默认值是“ADD”。Altera 推荐使用 LPM_DIRECTION 参数来指定 lpm_add_sub 函数的运算，而不是为 add_sub 端口指定一个常数
clock	否	用于流水线用法的时钟	时钟端口为 lpm_add_sub 函数提供流水线操作。如果 LPM_PIPELINE 值不为 0(默认值)，则必须连接 clock 端口

（续表）

端口名称	是否必需	描　　述	说　　明
clken	否	用于流水线用法的时钟使能端	仅适用于 VHDL
aclr	否	用于流水线用法的异步清除	流水线初始化成未定义的(X)逻辑电平。使用 aclr 端口可以在任一时刻将流水线设置成全 0，与 clock 信号异步

下表给出了 lpm_add_sub 的所有 OUTPUT 端口。

端口名称	是否必需	描　　述	说　　明
result	是	dataa+或－datab+或－cin	输出端口 LPM_WIDTH 宽度
cout	否	MSB 的进位输出或输入	如果使用了 overfolw，就不能使用 cout。cout 端口具有作为 MSB 的进位输出或输入的物理解释。cout 对在“UNSIGNED”运算中检测溢出的操作至关重要
overflow	否	结果超出可能的精确度	如果使用了 overfolw，就不能使用 cout。overflow 端口具有作为 MSB 的进位输出与 MSB 的进位输入的 XOR 运算的物理解释。只有在 LPM_REPRESENTATION 参数值是“SIGNED”时，overflow 才有意义

2. 参数

下表给出了 lpm_add_sub 组件的参数。

参　　数	类　型	是否必需	说　　明
LPM_WIDTH	整数	是	dataa、datab 和 result 端口的宽度
LPM_DIRECTION	字符串	否	值是“ADD”、“SUB”和“UNUSED”，如果忽略，则默认值是“DEFAULT”，该默认值指向从 add_sub 端口获得值的参数。如果使用 LPM_DIRECTION，就不能使用 add_sub 端口。Altera 推荐使用 LPM_DIRECTION 参数来规定 lpm_add_sub 函数的运算，而不是为 add_sub 端口指定一个常数
LPM_REPRES-ENTATION	字符串	否	执行的加法类型：“SINGED”、“UNSIGNED”或“UNUSED”。如果忽略，则默认值是“SINGED”
LPM_PIPELINE	整数	否	规定与 result 输出相关的延迟的时钟周期的数量。0 值表明没有延迟，将实例化一个纯组合函数。如果忽略，则默认值是 0(无流水线)
LPM_HINT	字符串	否	允许在 VHDL 设计文件中规定 Altera 特有的参数。默认值是“UNUSED”

(续表)

参　数	类　型	是否必需	说　明
LPM_TYPE	字符串	否	确定 VHDL 设计文件中的实体名称
ONE_INPUT_ _IS_CONSTANT	字符串	否	Altera 特有的参数。值是“YES”、“NO”和“UNUSED”。如果一个输入是常数，则提供更大的优化。如果忽略，则默认值是“NO”
MAXIMIZE_SPEED	整数	否	Altera 特有的参数。您可以指定 0 至 10 之间的值。如果使用该参数，MaxPlusII 就会试图为了速度而不是面积而优化 lpm_add_sub 函数的一个特殊实例，并覆盖 Global Project Logic Synthesis 对话框(Assign 菜单)中优化选项中的设置。如果不使用 MAXIMIZE_SPEED，就使用优化选项中的值。如果 MAXIMIZE_SPEED 的设置是 6 或更高，编译器就会优化 lpm_add_sub，得到更高速度，如果设置是 5 或更小，编译器就优化 lpm-add-sub，得到更小面积

注意：

对于 Verilog LPM 220 可合成代码(也就是 220model.v)，采用下面的参数命令：lpm_type，lpm_width，lpm_direction，lpm_representation，lpm_pipeline 和 lpm_hint。

3. 功能

下表是 lpm_add_sub 无符号运作的示例。

输　入			输　出	
add_sub	dataa	datab	Cout，result	overflow
1	a	b	a+b+cin	cout
0	a	b	a－b－cin	！cout

下表是 lpm_add_sub 有符号运作的示例。

输　入			输　出	
add_sub	dataa	datab	Cout，sum	overflow
1	a	b	a+b+cin	a≥0 和 b≥0 以及和<0 或 a<0 和 b<0 以及和≥0
0	a	b	a-b-cin	a≥0 和 b<0 以及和<0 或 a<0 和 b≥0 以及和≥0

4. 资源使用方法

下表总结了用 lpm_add_sub 兆函数实现带有进位输入和进位输出的 16 位无符号加法器的资源使用方法。逻辑单元与加法器宽度成线性正比关系。

设计目标		设计结果		
元器件系列	优　化	LC	速度(ns)	注　释
FLEX 6K、8K 和 10K	可布线性	45	53	EPF8282A-2 的速度
	速度	18	17	
MAX 5K、7K 和 9K	可布线性	28(22)	23	EPM7128E-7 的速度

圆括号中是所使用的共享扩展器的数量。

B.2.3　参数化的乘法器兆函数(lpm_mult)

Altera 推荐使用 lpm_mult 函数代替所有其他类型的乘法器函数，包括旧类型的乘法器宏函数。我们已经在 ccmul 和 fir_gen 设计中使用了这个兆函数。

Altera 推荐像“用 MegaWizard Plug-In Manager 创建定制兆函数变量”中所描述的那样来初始化这个函数。

Verilog 中原型的端口名称和命令是：

```
module lpm_mult (dataa,datab,sum,aclr,clock,
                 result);
```

VHDL 组件的声明如下：

```
COMPONENT lpm_mult
  GENERIC ( LPM_WIDTHA:POSITIVE;
            LPM_WIDTHB:POSITIVE;
            LPM_WIDTHS:POSITIVE;
            LPM_WIDTHP:POSITIVE;
            LPM_REPRESENTATION:STRING:="UNSIGNED";
            LPM_PIPELINE:INTEGER:=0;
            LPM_TYPE:STRING:= "L_MULT");
            LPM_HINT:STRING:= "UNUSED";
 PORT(dataa: IN STD_LOGIC_VECTOR(LPM_WIDTHA - 1 DOWNTO 0);
      datab: IN STD_LOGIC_VECTOR(LPM_WIDTHB - 1 DOWNTO 0)
      aclr,clken,clock: IN STD_LOGIC:='0';
      sum: IN STD_LOGIC_VECTOR(LPM_WIDTHS - 1 DOWNTO 0);
                                          : =(OTHERS=>'0');
      result:OUT STD_LOGIC_VECTOR(LPM_WIDTHP - 1 DOWNTO 0);
      );
END COMPONENT;
```

1. 端口

下表给出了 lpm_mult 的所有 INPUT 端口。

端口名称	是否必需	描　述	说　明
dataa	是	被乘数	输入端口 LPM_WIDTHA 宽度
datab	是	乘数	输入端口 LPM_WIDTHB 宽度
sum	否	部分和	输入端口 LPM_WIDTHS 宽度
clock	否	用于流水线用法的时钟	时钟端口为 lpm_mult 函数提供流水线操作。如果 LPM_PIPELINE 值不为 0，必须连接 clock 端口
clken	否	用于流水线用法的时钟使能端	仅适用于 VHDL
aclr	否	用于流水线用法的异步清除	流水线初始化成未定义的(X)逻辑电平。使用 aclr 端口可以在任一时刻将流水线设置成全 0，与 clock 信号异步

下表给出了 lpm_mult 的所有 OUTPUT 端口。

端口名称	是否必需	描　述	说　明
result	是	result=dataa*datab+sum 乘积 LSB 与和 LSB 对准	输出端口 LPM_WIDTHP 宽度。如果 LPM_WIDTHP< max(LPM_WIDTHA+LPM_WIDTHB，LPM_WIDTHS)或(LPM_WIDTHA +LPM_WIDTHS)，则只给出 LPM_WIDTHP 的 MSB

2. 参数

下表给出了 lpm_mult 组件的参数。

参　数	类　型	是否必需	说　明
LPM_WIDTHA	整数	是	dataa 端口的宽度
LPM_WIDTHB	整数	是	datab 端口的宽度
LPM_WIDTHP	整数	是	result 端口的宽度
LPM_WIDTHS	整数	是	sum 端口的宽度，即便没有使用 sum 端口，也需要该参数
LPM_REPRES-ENTATION	字符串	否	所执行的乘法类型为："SINGED"、"UNSIGNED"或"UNUSED"。如果忽略，则默认值是"SINGED"
LPM_PIPELINE	整数	否	指定与 result 输出相关的延迟的时钟周期的数量。0 值表明没有延迟，将实例化一个纯组合函数。如果忽略，则默认值是 0(无流水线)
LPM_HINT	字符串	否	允许在 VHDL 设计文件中指定 Altera 特有的参数。默认值是"UNUSED"

(续表)

参　数	类　型	是否必需	说　明
LPM_TYPE	字符串	否	确定 VHDL 设计文件中的实体名称
INPUT_A_ _IS_CONSTANT	字符串	否	Altera 特有的参数。值是“YES”、“NO”和“UNUSED”。如果 dataa 与一个恒定值连接在一起，就将 INPUT_A_IS_CONSTANT 设定为 YES，优化乘法器，以提高资源的使用及速度。如果忽略，则默认值是“NO”
INPUT_B _IS CONSTANT	字符串	否	Altera 特有的参数。值是“YES”、“NO”和“UNUSED”。如果 datab 与一个恒定值连接在一起，就将 INPUT_B__IS_CONSTANT 设定为 YES，优化乘法器，以提高资源的使用及速度。如果忽略，则默认值是“NO”
USE_EAB	字符串	否	Altera 特有的参数。值是“ON”、“OFF”和“UNUSED”。设定 USE_EAB 参数为“ON”，允许 MaxPlusII 使用 FLEX 10K 元器件中的 EAB 实现 4×4(或 8×常数值)的构造模块。Altera 推荐只有当缺乏 LCELL 的时候才将 USE-EAB 设置为“ON”。如果想使用这个参数，则在 GDF 中初始化函数时，必须用 Edit Ports/Parameters 对话框(符号菜单)手动输入参数名称和值。还可以在 TDF 或 Verilog 设计文件中使用这个参数名称。在 VHDL 设计文件中必须使用 LPM_HINT 参数来指定 USE_EAB 参数
LATENCY	整数	否	Altera 特有的参数。与 LPM_PIPELINE 参数一样(这个参数只为向后兼容的 MaxPlusII 7.0 版本设计。对于新的设计，可以使用 LPM_PIPELINE 参数代替
MAXIMIZE_SP EED	整数	否	Altera 特有的参数。可以指定 0 至 10 之间的值。如果使用该参数，MaxPlusII 就会试图对速度而不是面积优化 lpm_mult 函数的一个特殊实例，并覆盖 Global Project Logic Synthesis 对话框(Assign 菜单)中优化选项的设置。如果不使用 MAXIMIZE_SPEED，就使用优化选项中的值，如果 MAXIMIZE_SPEED 的设定是 6 或更高，编译器就会优化 lpm_add_sub，得到更高的速度，如果设定是 5 或更小，编译器就优化 lpm_add_sub，得到更小面积
LPM_HINT	字符串	否	允许在 VHDL 设计文件中指定 Altera 特有的参数。默认值是“UNUSED”

注意：

仅在 LPM_REPRESENTATION 被设置为“SIGNED”时，指定 MAXIMI2E-SPEED 的值才有效。

对于 Verilog LPM 220 可合成代码(也就是 220model.v)，可采用下面的参数命令：lpm_type，lpm_widtha，lpm_widthb，lpm_widths，lpm_widthp，lpm_representation，lpm_pipeline 和 lpm_hint。

3. 功能

下表是 lpm_mult 无符号运作的示例。

输　入			输　出
dataa	datab	sum	product
a	*b*	*s*	LPM_WIDTHP *a* **b*+*s* 的 MSB

4. 资源使用方法

下表总结了用 lpm_mult 兆函数实现 LPM_PIPELINE＝0 以及没有优化 sum 输入的 4 位和 8 位乘法器的使用方法。逻辑单元与乘法器宽度成线性正比关系。

设 计 目 标		设 计 结 果			
元器件系列	优　化	宽 度	LC	速度(ns)	注　释
FLEX 6K、8K 和 10K	可布线性	8	121	80	EPF8282A-2 的速度
	速度	8	163	52	
FLEX 6K、8K 和 10K	可布线性	4	29	34	EPF8282A-2 的速度
	速度	4	41	27	
MAX 5K、7K 和 9K	可布线性	4	26(11)	23	EPM7128E-7 的速度
	速度	4	27(4)	19	

圆括号中表示的是共享扩展器的数量。上面 FLEX 10K 系列 4 位×4 位乘法器例子可以在一个单片 EAB 中实现。

B.2.4　参数化的 ROM 兆函数(lpm_rom)

Altera 推荐使用 lpm_rom 实现所有的 ROM 函数。lpm_rom 函数只适用于 FLEX 10K 元器件。我们已经将这个兆函数用在了 fun_text 和 darom 设计中。MaxPlusII 编译器自动地在 FLEX 10K 元器件的 EAB 中实现这个函数的合适部分。这样就不需要为这个函数使用“Implement in EAB”逻辑选项，而且这样做的话，会出现警告信息。

Altera 推荐像“用 MegaWizard Plug-In Manager 创建定制兆函数变量”中所描述的那样来初始化这个函数，您也可以用 genmen.exe 实用程序为在第三方仿真器中使用的这个函数创建一个仿真模型。在 DOS 提示符中输入 genmem-h 就可以得到关于如何利用这个实用程序的信息。

Verilog 中原型的端口名称和命令是：

```
module lprn_rom (address,inclock,outclock,memenab,
                 q);
```

VHDL 组件的声明如下：

```
COMPONENT lpm_rom
```

```
    GENERIC (LPM_WIDTH              :POSITIVE;
             LPM_TYPE:              STRING      :="L_ROM"
             LPM_WIDTHAD            :POSITIVE;
             LPM_NUMWORDS           :POSITIVE;
             LPM_FILE               :STRING;
             LPM_ADDRESS_CONTROL:STRING := "REGISTERED";
             LPM_OUTDATA            :STRING := "REGISTERED";
             LPM_HINT               :STRING := "UNUSED";
  PORT(address      :IN STD_LOGIC_VECTOR(LPM_WIDTHAD - 1 DOWNTO 0);
       inclock      : IN STD_LOGIC:='1';
       outclock     : IN STD_LOGIC:= '1';
       memenab      : IN STD_LOGIC:= '1';
       q            : OUT STD_LOGIC_VECTOR(LPM_WIDTH - 1 DOWNTO 0)
       );
END COMPONENT;
```

1. 端口

下表给出了 lpm_rom 的所有 INPUT 端口。

端口名称	是否必需	描　　述	说　　明
address	是	输入到存储器的地址	输入端口 LPM_WIDTHAD 宽度
inclock	否	输入寄存器的时钟	当 inclock 端口被连接时，address 端口是同步的(已注册)，而当 inclock 端口未被连接时，address 端口是异步的(已注册)
outclock	否	输出寄存器的时钟	当 outclock 端口被连接时，编址的存储器到 q 的目录的响应是同步的，而 outclock 未被连接时则是异步的
memnab	否	存储器使能输入	High＝数据输出到 q，Low＝高阻抗输出

下表给出了 lpm_rom 的所有 OUTPUT 端口。

端 口 名 称	是 否 必 需	描　　述	说　　明
q	是	存储器输出	输出端口 LPM_WIDTH 宽度

2. 参数

下表给出了 lpm_rom 组件的参数。

参　数	类　型	是否必需	说　明
LPM_WIDTH	整数	是	q 端口的宽度
LPM_WIDTHAD	整数	是	address 端口的宽度。LPM_WIDTHAD 的宽度应该(但不是必须)等于 $\log_2$(LPM_NUMWORDS)。如果 LPM_WIDTHAD 太小，则某些存储器位置就不能够编址了。如果太大，地址就会返回未定义的逻辑电平
LPM_NUMWORDS	整数	是	存储器中储存的字的数量。通常，这个值应该(但不是必须)是 $2^{LPM\_WIDTHAD}-1<LPM\_NUMWORDS\leq 2^{LPM\_WIDTHAD}$。如果忽略该参数，则默认值是 $2^{LPM\_WIDTHAD}$
LPM_FILE	字符串	否	存储器初始化文件(*.mif)或十六进制(Intel 格式)文件(*.hex)的名称，包括 ROM 初始化数据(“<filename>”)或“UNUSED”
LPM_ADDRESS_CONTROL	字符串	否	值是“REGISTERED”、“UNREGISTERED”和“UNUSED”。表示地址端口是否已注册。如果忽略该参数，则默认值是“REGISTERED”
LPM_OUTDATA	字符串	否	值是“REGISTERED”、“UNREGISTERED”和“UNUSED”表示 q 和 eq 端口是否已注册。如果忽略该参数，则默认值是“REGISTERED”
LPM_HINT	字符串	否	允许指定在 VHDL 设计文件中 Altera 特有的参数。默认值是“UNUSED”
LPM_TYPE	字符串	否	确定 VHDL 设计文件的 LPM 实体名称

注意：

对于 Verilog LPM 220 可合成代码(也就是 220model.v)，可以采用下面的参数命令：lpm_type，lpm_width，lpm_widthad，lpm_numwords，lpm_address_control，lpm_outdata,lpm_file 和 lpm_hint。

3. 功能

下表给出了从 Nlpm_rom 的存储器中同步读取的方式。

OUTCLOCK	MEMENAB	功　能
X	L	q 输出是高阻抗(存储器未使能)
↑	H	在输出中无变化
↑	H	输出寄存器加载的是 address 所指向的存储器位置的内容。q 用于输出输出寄存器的内容

当 inclock 和 outclock 均未被连接时，存储器的操作是完全异步的。输出 q 是非对称的，并反应了存储器中 address 指向的数据。下表给出了 lpm_rom 的异步存储器操作方式。

MENEMAB	函　数
L	q 输出是高阻抗的(存储器未使能)
H	读出由 address 指向的存储器位置

4. 资源使用方法

lpm_rom 兆函数的每个存储器位使用一个嵌入式单元。

附录C 术语汇编

ACC	Accumulator，累加器
ACT	Actel FPGA family，Actel FPGA 系列
ADCL	All-digital CL，全数字 CL
ADPLL	All-Digital PLL，全数字锁相环
ADC	Analog-to-Digital Converter，模拟-数字转换器
ADSP	Analog Devices Digital Signal Processor family，模拟元器件数字信号处理器系列
AFT	Arithmetic Fourier Transform，算术傅立叶变换
AHDL	Altera HDL，Altera 硬件描述语言
AM	Amplitude Modulation，幅度调制
ALU	Arithmetic Logic Unit，算术逻辑单元
AMD	Advanced Micro Devices, Inc.，(美国)AMD 公司(生产半导体及芯片)
ASCII	American Standard Code for Information Interchange，美国信息交换标准代码
ASIC	Application Specific IC，专用集成电路
BDD	Binary Decision Diagram，二元判定框图
BP	Bandpass，通频带，带通
BRS	Base Removal Scaling，基本迁移缩放比例
BS	Barrel Shifter，筒状移位器
CAE	Computer-Aided Engineering，计算机辅助工程
CAST	Carlisle Adams and Stafford Tavares
CBC	Cipher Block Chaining，密码字组链接
CBIC	Cell-Based IC，单元基集成电路
CD	Compact Disc，只读光盘
CFA	Common Factor Algorithm，公因数算法
CFB	Cipher Feedback，密码反馈
CIC	Cascaded Integrator Comb，级联积分器梳状
CL	Costas Loop，边环
CLB	Configurable Logic Block，配置逻辑模块
CMOS	Complementary Metal-Oxide-Semiconductor，互补型金属氧化物半导体
CODEC	Coder/Decoder，编码器/译码器
CORDIC	COordinated Rotation DIgital Computer，坐标旋转数字计算机，协调旋转数字计算机

COTS	Commercial Off-The-Shelf technology，商用成品技术
CPLD	Complex PLD，复杂可编程逻辑电路
CPU	Central Processing Unit，中央处理器，(又称)中央处理机
CQF	Conjugate Quadrature Filter，共轭正交滤波器
CRNS	Complex Residue Number System，复数余数系统
CRT	Chinese Remainder Theorem，中国余数定理
CSOC	Canonical Self-Orthogonal Code，标准自成正交代码
CSD	Canonical Signed Digit，标准有符号数字量
CWT	Continuous Wavelet Transform，连续小波变换
CZT	Chirp-z Transform，线性调频脉冲 z 变换
DA	Distributed Arithmetic，分布式算法
DAC	Digital-to-Analog Converter，数字-模拟转换器
DB	DauBechies filter，DauBechies 滤波器
DC	Direct Current，直流
DCT	Discrete Cosine Transform，离散余弦变换
DCO	Digital Controlled Oscillator，数字控制振荡器
DES	Data Encryption Standard，数据加密标准
DFT	Discrete Fourier Transform，离散傅立叶变换
DHT	Discrete Hartley Transform，离散哈特利变换
DIF	Decimation In Frequency，频率抽取
DIT	Decimation In Time，时间抽取，时序抽取
DMT	Discrete Morlet Transform，离散 Morlet 变换
DPLL	Digital PLL，数字锁相环
DSP	Digital Signal Processing，数字信号处理
DWT	Discrete Wavelet Transform，离散小波变换
EAB	Embedded Array Block，嵌入式阵列模块
ECB	Electronic Code Book，电子源码书
ECL	Emitter Coupled Logic，射极耦合逻辑(电路)
EDIF	Electronic Design Interchange Format，电子设计互换格式
EFF	Electronic Frontier Foundation，电子尖端基础(一个专门从事破译密码工作的组织)
EPF	Altera FPGA family，Altera FPGA 系列
EPROM	Electrically Programmable ROM，电可编程序只读存储器
ERA	Plessey FPGA family，Plessey FPGA 系列
ERNS	Eisenstein RNS，Eisenstein 余数系统
ESA	European Space Agency，欧洲航天局

FCT	Fast Cosine Transform，快速余弦变换
FC2	FPGA Compiler II，FPGA 自动编码器 II
FF	Flip-Flop，触发器
FFT	Fast Fourier Transform，快速傅立叶变换
FIR	Finite Impulse Response，有限脉冲响应
FIFO	First-In First-Out，先入先出
FLEX	Altera FPGA family，Altera FPGA 系列
FM	Frequency Modulation，频率调制
FNT	Fermat NTT，Fermat Network Transfer Table，费尔马网络传输表
FPGA	Field Programmable Gate Array，现场可编程门阵列
FPL	Field-Programmable Logic(combines CPLD and FPGA)，现场可编程逻辑(CPLD 与 FPGA 的结合)
FSF	Frequency Sampling Filter，频率采样滤波器
FSK	Frequency Shift Keying，移频键控
FSM	Finite State Machine，有限状态自动机
GAL	Generic Array Logic，通用阵列逻辑(电路)
GF	Galois Field，伽罗瓦域，有限域
HB	HalfBand filter，半波带滤波器
HI	HIgh frequency，高频
HDL	Hardware Description Language，硬件描述语言
HSP	Harris Semiconductor DSP ICs，哈里森半导体 DSP 集成电路
IBM	International Business Machines Corp.，(美国)国际商用机器公司
IC	Integrated Circuit，集成电路
IDCT	Inverse DCT，离散傅立叶反变换、离散傅立叶逆变换、逆 DCT
IDEA	International Data Encryption Algorithm，国际数据加密算法
IDFT	Inverse DFT，离散傅立叶反变换、离散傅立叶逆变换、逆 DFT
IEEE	Institute of Electrical & Electronic Engineers，(美国)电气(和)电子工程师学会
IF	Inter Frequency，内部频率
IFFT	Inverse Fast Fourier Transform，快速傅立叶反变换、快速傅立叶逆变换、逆 FFT
IIR	Infinite Impulse Response，无限脉冲响应
INTT	Inverse NTT，数论反变换、数论逆变换、逆 NTT
JPEG	Joint Photographic Experts Group，ISO/IEC，联合图像专家组，一种压缩标准
LAB	Logic Array Block，逻辑阵列模块
LAN	Local Area Network，局域网，局部区域网，局域网路
LC	Logic Cell，逻辑单元

LE	Logic Element，逻辑元件
LF	Low Frequency，低频
LFSR	Linear Feedback Shift Register，线性反馈移位寄存器
LNS	Logarithmic Number System，对数记数系统
LO	LOw frequency，低频
LP	Low Pass，低通
LPM	Library of Parameterized Modules，参数化模块库
LRS	serial Left Right Shifter，串行左右移位器
LSB	Least Significant Bit，最低有效位
LSI	Large Scale Integration，大规模集成(电路)
LTI	Linear Time Invariant，线性时间不变量
LUT	Look-Up Table，查表
MAC	Multiplication and ACcumulate，乘-累加
MACH	AMD/Vantis FPGA family，AMD/Vantis FPGA 系列
MAG	Multiplier Adder Graph，乘法器-加法器图
MAX	Altera CPLD family，Altera CPLD 系列
MIF	Memory Initialization File，存储器初始化文件
MLSE	Maximum Likelihood Sequence Estimator，最大可能序列估计函数
MNT	Mersenne NTT，Mersenne 数论变换
MPEG	Motion Picture Experts Group，【ISO】运动图像专家组(全球影像/声音/系统压缩标准)
MPX	Multiplexer，多路复用器
MSPS	Millions of Sample Per Second，每秒钟几百万次采样
MRC	Mixed Radix Conversion，混合基数转换
MSB	Most Significant Bit，最高有效位
MUL	Multiplication 乘法
NP	Non Polynomial complex problem，非多项式复合问题
NRE	Non Reoccurring Engineering costs，不可重复的工程成本
NTT	Number Theoretic Transform，数论变换
OFB	Open FeedBack(mode)，开反馈(模式)
PC	Personal Computer，个人计算机
PD	Phase Detector，相位检测器
PFA	Prime Factor Algorithm，质数因子算法
PLA	Programmable Logic Array，可编程逻辑阵列
PLD	Programmable Logic Device，可编程逻辑器件
PLL	Phase-Locked Loop，锁相环

PM	Phase Modulation，调相制，相位调制
PREP	PRogrammable Electronic Performance Cooperation，可编程电子产品性能协议
PRNS	Polynomial RNS，多项式余数系统
PROM	Programmable ROM，可编程序只读存储器
PSK	Phase Shift Keying，相移键控
QDFT	Quantized DFT，量化离散傅立叶变换
QLI	Quick Look-In，快速搜索
QFFT	Quantized FFT，量化快速傅立叶变换
QMF	Quadrature Mirror Filter，正交镜像滤波器
QRNS	Quadratic RNS，二次余数系统
RAM	Random Access Memory，随机存取存储器
RC	Resistor/Capacity，电阻/电容
RF	Radio Frequency，射频
RISC	Reduced Instruction Set Computer，精简指令系统计算机
RNS	Residue Number System，余数系统
ROM	Read Only Memory，只读存储器
RPFA	Rader Prime Factor Algorithm，Rader 质数因子算法
RS	serial Right Shifter，串行右移移位器
RSA	Rivest-Shamir-Adleman，在数据保密技术中使用的一种通用密钥密码方法，它是基于大数作因子分解的难度而建立的方法，由 Rivest，Shamir 和 Adleman 提出
SD	Signed Digit，有符号数字量
SM	Signed Magnitude，有符号数值
SPLD	Simple PLD，简化 PLD
SPT	Signed Power of Two，有符号的 2 的幂
SR	Shift Register，移位寄存器
SRAM	Static Random Access Memory，静态随机存取存储器
STFT	Short Time Fourier Transform，瞬时傅立叶变换
TLU	Table Look-up，表查询
TMS	Texas Instruments DSP family，得克萨斯仪器 DSP 系列
TI	Texas Instruments，德克萨斯仪器(公司)
TTL	Transistor-Transistor Logic 晶体管-晶体管逻辑(电路)
UART	Universal Asynchronous Receiver Transmitter，通用异步收发器
VCO	Voltage Controlled Oscillator，压控振荡器
VHDL	VHSIC Hardware Description Language，VHSIC 硬件描述语言

VHSIC	Very High Speed Integrated Circuit，其全称是极高速集成电路
VLSI	Very Large Scale Integration，超大规模集成电路
WFTA	Winograd Fourier Transform Algorithm，Winograd 傅立叶变换算法
XC	Xilinx FPGA family，Xilinx FPGA 系列
XNOR	eXclusive NOR gate,“同”门

参 考 文 献

1. B. Dipert: "EDN's First Annual PLD Directory," EDN pp. 54-84 (2000) Http://www. ednmag. com/ednmag/reg/2000 / 08172000 / 17cs.htm
2. S. Brown, Z. Vranesic: Fundamentals of Digital Logic with VHDL Design(Prentice Hall, Englewood Cliffs, New Jersey, 1999)
3. D. Smith: HDL Chip Design (Doone Publications, Madison, Alabama, USA,1996)
4. U. Meyer-Bäse: The Use of Complex Algorithm in the Realization of Universal Sampling Receiver Using FPGAs (in German) (VDI/Springer, Düsseldorf,1995), Vol. 10, No. 404, 215 pages
5. U. Meyer-Bäse: Fast Digital Signal Processing (in German) (Springer, Heidel-berg, 1999), 370 pages
6. P. Lapsley, J. Bier, A. Shoham, E. Lee: DSP Processor Fundamentals (IEEE Press, New York, 1997)
7. D. Shear: "EDN's DSP Benchmarks," EDN 33, 126-148 (1988)
8. GEC Plessey Semiconductors: "Data Sheet," ERA60100 (1990)
9. J. Greene, E. Hamdy, S. Beal: "Antifuse Field-Programmable Gate Arrays,"Proceedings of the IEEE pp. 1042-56 (1993)
10. J. Rose, A. Gamal, A. Sangiovanni-Vincentelli: "Architecture of Field-Programmable Gate Arrays," Proceedings of the IEEE pp. 1013-29 (1993)
11. Xilinx: "PREP Benchmark Observations," in Xilinx-Seminar San Jose (1993)
12. Altera Corporation: "PREP Benchmarks Reveal FLEX 8000 is Biggest, MAX7000 is Fastest," in Altera Corporation News & Views San Jose (1993)
13. Actel: "PREP Benchmarks Confirm Cost Effectiveness of Field-Programmable Gate Arrays," in Actel-Seminar (1993)
14. E. Lee: "Programmable DSP Architectures: Part I," IEEE Trdnsactions on Acoustics, Speech and Signal Processing Magazine pp. 4-19 (1988)
15. E. Lee: "Programmable DSP Architectures: Part II," IEEE Transactions on Acoustics, Speech and Signal Processing Magazine pp. 4-14 (1989)
16. R. Petersen, B. Hutchings: "An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing," Lecture Notes in Computer Science 975,293-302 (Springer, Heidelberg, 1995)
17. J. Villasenor, B. Hutchings: "The Flexibility of Configurable Computing," IEEE Signal Processing Magazine pp. 67-84 (1998)
18. Xilinx: "Data Book," XC2000, XC3000 and XC4000 (1993)

19. Altera Corporation: "Data Sheet," FLEX 10K CPLD Family (1996)
20. Altera Corporation: "Manual," Max+PlusII Getting Started (1995)
21. O. Spaniol: Computer Arithmetic: Logic and Design (John Wiley & Sons, New York, 1981)
22. I. Koren: Computer Arithmetic Algorithms (Prentice Hall, Englewood Cliffs,New Jersey, 1993)
23. E.E. Swartzlander: Computer Arithmetic, Vol. I (Dowden, Hutchingon and Ross, Inc., Stroudsburg, Pennsylvania, 1980), also reprinted by IEEE Com-purer Society Press (1990)
24. E. Swartzlander: Computer Arithmetic, Vol. II (IEEE Computer Society Press, Stroudsburg, Pennsylvania, 1990)
25. K. Hwang: Computer Arithmetic: Principles, Architecture and Design (John Wiley & Sons, New York, 1979)
26. N. Takagi, H. Yasuura, S. Yajima: "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," IEEE Transactions on Computers 34(2) (1985)
27. D. Bull, D. Horrocks: "Reduced-Complexity Digital Filtering Structures Using Primitive Operations," Electronics Letters pp. 769-771 (1987)
28. D. Bull, D. Horrocks: "Primitive Operator Digital Filters," IEE Proceedings-G 138,401-411 (1991)
29. A. Dempster, M. Macleod: "Use of Minimum-Adder Multiplier Blocks in FIR. Digital Filters," IEEE Transactions on Circuits and Systems II 42, 569-577(1995)
30. A. Dempster, M. Macleod: "Comments on 'Minimum Number of Adders for Implementing a Multiplier and Its Application to the Design of Multiplierless Digital Filters'," IEEE Transactions on Circuits and Systems H 45,242-243(1998)
31. F. Taylor, R. Gill, J. Joseph, J. Radke: "A 20 Bit Logarithmic Number System Processor," IEEE Transactions on Computers 37(2) (1988)
32. P. Lee: "An FPGA Prototype for a Multiplierless FIR Filter Built Using the Logarithmic Number System," Lecture Notes in Computer Science 975,303-310 (Springer, Heidelberg, 1995)
33. J. Mitchell: "Computer Multiplication and Division Using Binary Logarithms," IRE Transactions on Electronic Computers EC-11,512-517 (1962)
34. N. Szabo, R. Tanaka: Residue Arithmetic and Its Applications to Computer Technology (McGraw-Hill, New York, 1967)
35. M. Soderstrand, W. Jenkins, G. Jullien, F. Taylor: Residue Number System Arithmetic: Modern Applications in Digital Signal Processing, IEEE Press Reprint Series (IEEE Press, New York, 1986)
36. U. Meyer-Bäse, A. Meyer-Bäse, J. Mellott, F. Taylor: "A Fast Modified CORDIC-Implementation of Radial Basis Neural Networks," Journal of VLSI Signal Processing pp. 211-218 (1998)
37. V. Hamann, M. Sprachmann: "Fast Residual Arithmetics with FPGAs," in Proceedings of

the Workshop on Design Methodologies for Microelectronics Smolenice Castle, Slovakia (1995), pp. 253-255

38. G. Jullien: "Residue Number Scaling and Other Operations Using ROM Arrays," IEEE Transactions on Communications 27, 325-336 (1978)
39. M. Griffin, M. Sousa, F. Taylor: "Efficient Scaling in the Residue Number System," in IEEE International Conference on Acoustics, Speech, and Signal Processing (1989), pp. 1075-1078
40. G. Zelniker, F. Taylor: "A Reduced-Complexity Finite Field ALU," IEEE Transactions on Circuits and Systems 38(12)(1991)
41. IEEE Task P754: "A Proposed Standard for Binary Floating-Point Arithmetic," IEEE Transactions on Computers 14(12) (1981)
42. N. Shirazi, P. Athanas, A. Abbott: "Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine," Lecture Notes in Computer Science 975,282-292 (Springer, Heidelberg; 1995)
43. M. Bayoumi, G. Jullien, W. Miller: "A VLSI Implementation of Residue Adders," IEEE Transactions on Circuits and Systems 34(3), 284-288 (1987)
44. A. Garcia, U. Meyer-Bäse, F. Taylor: "Pipelined Hogenauer CIC Filters Using Fidd-Programmable Logic and Residue Number System," in IEEE International Conference on Acoustics, Speech, and Signal Processing Vol. 5 (1998),pp. 3085-3088
45. L. Turner, P. Graumann, S. Gibb: "Bit-Serial FIR Filters with CSD Coefficients for FPGAs," Lecture Notes in Computer Science 975, 311-320 (Springer, Heidelberg, 1995)
46. A. Croisier, D. Esteban, M. Levilion, V. Rizo: (1973), "Digital Filter for PCM Encoded Signals," US Patent No. 3777130
47. A. Peled, B. Liu: "A New Realizattion of Digital Filters," IEEE Transactions on Acoustics, Speech and Signal Processing 22(6), 456-462 (1974)
48. K. Yiu: "Oil Sign-Bit. Assignment for a Vector Multiplier," Proceedings of the IEEE 64, 372-373 (1976)
49. K. Kammeyer: "Quantization Error on the Distributed Arithmetic," IEEE Transactions on Circuits and Systems 24(12), 681-689 (1981)
50. F. Taylor: "An Analysis of the Distributed-Arithmetic Digital Filter," IEEE Transactions on Acoustics, Speech and Signal Processing 35(5), 1165-1170 (1986)
51. S. White: "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," IEEE Transactions on Acoustics, Speech and Signal Processing Magazine pp. 4-19 (1989)
52. K. Kammeyer: "Digital Filter Realization in Distributed Arithmetic," in Proc. European Conf. on Circuit Theory and Design (1976), Genoa, Italy
53. F. Taylor: Digital Filter Design Handbook (Marcel Dekker, New York, 1983)
54. H. Nussbaumer: Fast Fourier Transform and Convolution Algorithms(Springer, Heidelberg,

1990)

55. H. Schmid: Decimal Computation (John Wiley & Sons, New York, 1974)
56. Y. Hu: "CORDIC-Based VLSI Architectures for Digital Signal Processing," IEEE Signal Processing Magazine pp. 16-35 (1992)
57. U. Meyer-Bäse, A. Meyer-Bäse, W. Hilberg: "COordinate Rotation DIgital Computer (CORDIC) Synthesis for FPGA," Lecture Notes in Computer Sci-ence 849, 397-408 (Springer, Heidelberg, 1994)
58. J.E. Voider: "The CORDIC Trigonometric Computing Technique," IRE Transactions on Electronics Computers 8(3), 330-4 (1959)
59. J. Walther: "A Unified Algorithm for Elementary Functions," Spring Joint Computer Conference pp. 379-385 (1971)
60. X. Hu, R. Huber, S. Bass: "Expanding the Range of Convergence of the CORDIC Algorithm," IEEE Transactions on Computers 40(1), 13-21 (1991)
61. D. Timmermann (1990): "CORDIC-Algorithmen, Architecturen und monolithische Realisierungen mit. Anwendungen in der Bildverarbeitung,"Ph.D. thesis, VDI/Springer, Du sseldorf, Vol. 10, No. 152
62. H. Hahn (1991): "Untersuchung und Integration von Berechnungsverfahren elementarer Funktionen auf CORDIC-Basis mit. Anwendungen in der adaptiven Signalverarbeitmlg," Ph.D. thesis, VDI/Springer, Düsseldorf Vol. 9, No. 125
63. G. Ma (1989): "A Systolic Distributed Arithmetic Computing Machine for Digital Signal Processing and Linear Algebra Applications," Ph.D. thesis, University of Florida, Gainesville
64. Y.H. Hu: "The Quantization Effects of the CORDIC Algorithm," IEEE Transactions on Signal Processing pp. 834-844 (1992)
65. A.V. Oppenheim, R.W. Schafer: Discrete-Time Signal Processing(Prentice Hall, Englewood Cliffs, New Jersey, 1992)
66. D.J. Goodman, M.J. Carey: "Nine Digital Filters for Decimation and Interpolation," IEEE Transactions on Acoustics, Speech and Signal Processing 25(2)121-126 (1977)
67. R. Hartley: "Subexpression Sharing in Filters Using Canonic Signed Digital Multiplier," IEEE Transactions on Circuits and Systems II 30(10), 677-688(1996)
68. R. Saal: Handbook of filter design (AEG-Telefunken, Frankfurt, Germany,1979)
69. C. Barnes, A. Fam: "Minimum Norm Recursive Digital Filters that Are Free of Overflow Limit Cycles," IEEE Transactions on Circuits and Systems pp. 569-574 (1977)
70. A. Fettweis: "Wave Digital Filters: Theorie and Practice," Proceedings of the IEEE pp. 270-327 (1986)
71. R. Crochiere, A. Oppenheim: "Analysis of Linear Digital Networks," Proceedings of the IEEE 63(4), 581-595 (1995)
72. A. Dempster, M. Macleod: "Multiplier Blocks and Complexity of IIR Structures,"

Electronics Letters 30(22), 1841-1842 (1994)

73. A. Dempster, M. Macleod: "IIR Digital Filter Design Using Minimum Adder Multiplier Blocks," IEEE Transactions on Circuits and Systems II 45, 761-763 (1998)
74. A. Dempster, M. Macleod: "Constant Integer Multiplication Using Minimum Adders," IEE Proceedings- Circuits, Devices & Systems 141, 407-413 (1994)
75. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining Using Scattered Look-Ahead and Decomposition," IEEE Transactions on Acoustics, Speech and Signal Processing 37(7), 1099-1117 (1989)
76. H. Loomis,B. Sinha: "High-Speed Recursive Digital Filter Realization," Circuits, Systems, Signal Processing 3(3), 267-294 (1984)
77. M. Soderstrand, A. de la Serna, H. Loomis: "New Approach to Clustered Look-Ahead Pipelined IIR Digital Filters," IEEE Transactions on Circuits and Systems II 42(4), 269-274(1995)
78. J. Living, B. Al-Hashimi: "Mixed Arithmetic Architecture: A Solution to tile Iteration Bound for Resource-Efficient FPGA and CPLD Recursive Digital Filters," in IEEE International Symposium on Circuits and Systems Vol. I (1999), pp. 478-481
79. H. Martinez, T. Parks: "A Class of Infinite-Duration Impulse Response Digital Filters for Sampling Rate Reduction," IEEE Transactions on Acoustics,Speech and Signal Processing 26(4), 154-162 (1979)
80. K. Parhi, D. Messerschmidt: "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part II: Pipelined Incremental Block Filtering," IEEE Transactions on Acoustics, Speech and Signal Processing 37(7), 1118-1134(1989)
81. M. Shajaan, J. Sorensen: "Time-Area Efficient Multiplier-Free Recursive Filter Architectures for FPGA ImplementafionT," in IEEE International Conference on Acoustics, Speech, and Signal Processing (1996), pp. 3269-3272
82. P. Vaidyanathan: Multirate Systems and Filter Banks (Prentice Hall, Englewood Cliffs, New Jersey, 1993)
83. S. Winograd: "On Computing the Discrete Fourier Transform," Mathematics of Computation 32, 175-199 (1978)
84. Z. Mou, P. Duhamel: "Short-Length FIR Filters and Their Use in Fast Non-recursive Filtering," IEEE Transactions on Signal Processing 39, 1322-1332 (1991)
85. P. Balla, A. Antoniou, S. Morgera: "Higher Radix Aperiodic-Convolution Algorithms," IEEE Transactions on Acoustics, Speech and Signal Processing 34(1), 60-68 (1986)
86. E.B. Hogenauer: "An Economical Class of Digital Filters for Decimation and Interpolation," IEEE Transactions on Acoustics, Speech and Signal Processing 29(2), 155-162 (1981)
87. Harris Semiconductor: "Data Sheet," HSP43220 Decimating Digital Filter(1992)
88. Motorola Inc.: "Pilkingtons Patent," Elektronik Praxis pp. 40-43 (1993)
89. O. Six (1996): "Design and Implementation of a Xilinx Universal XC-4000 FPGAs board,"

Master's thesis, Institute for Data Technics, Darmstadt University of Technology

90. S. Dworak (1996): "Design and Realization of a New Class of Frequency Sampling Filters for Speech Processing Using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
91. A. Haar: "Zur Theorie der orthogonalen Funktionensysteme," Mathematische Annalen 69,331-371 (1910). Dissertation Göttingen 1909
92. P. Sweeney: Error Control Coding (Prentice Hall, New York, 1991)
93. C. Herley, M. Vetterli: "Wavelets and Recursive Filter Banks," IEEE Transactions on Signal Processing 41, 2536-2556 (1993)
94. I. Daubechies: Ten Lectures on Wavelets (Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1992)
95. I. Daubechies, W. Sweldens: "Factoring Wavelet Transforms into Lifting Steps," The Journal of Fourier Analysis and Applications 4, 365-374 (1998)
96. G. Strang, T. Nguyen: Wavelets and Filter Banks (Wellesley-Cambridge Press, Wellesley MA, 1996)
97. D. Esteban, C. Galand: "Applications of Quadrature Mirror Filters to Split Band Voice Coding Schemes," in IEEE International Conference on Acoustics,Speech, and Signal Processing (1977), pp. 191-195
98. M. Smith, T. Barnwell: "Exact Reconstruction Techniques for Tree-Structured Subband Coders," IEEE Transactions on Acoustics, Speech and Signal Processing pp. 434-441 (1986)
99. M. Vetterli, J. Kovacevic: Wavelets and Subband Coding (Prentice Hall, Englewood Cliffs, New Jersey, 1995)
100. R. Crochiere, L. Rabiner: Multirate Digital Signal Processing (Prentice Hall, Englewood Cliffs, New Jersey, 1983)
101. M. Acheroy, J.M. Mangen, Y. Buhler.: "Progressive Wavelet Algorithm Versus JPEG for the Compression of METEOSAT Data," in SPIE, San Diego (1995)
102. T. Ebrahimi, M. Kunt: "Image Compression by Gabor Expansion," Optical Engineering 30, 873-880 (1991)
103. D. Gabor: "Theory of Communication," J. Inst. Elect. Eng (London)93,429-457 (1946)
104. A. Grossmann, J. Morlet: "Decomposition of Hardy Functions into Square Integrable Wavelets of Constant Shape," SIAM J. Math. Anal. 15, 723 736(1984)
105. U. Meyer-Bäse: "High-Speed Implementation of Gabor and Morlet Wavelet Filterbanks Using RNS Frequency Sampling Filters," in Aerosense 98 *SPIE*,Orlando (1998), pp. 522-533
106. U. Meyer-Bäse: "Die Hutlets-eine biorthogonale Wavelet-Familie: Effiziente Realisierung durch multiplizierenfreie, perfekt rekonstruierende Quadratur Mirror Filter," Frequenz pp. 39-49 (1997)
107. U. Meyer-Bäse, F. Taylor: "The Hutlets - a Biorthogonal Wavelet Family and Their

High-Speed Implementation with RNS, Multiplier-Free, Perfect Recon-struction QMF," in Aerosense 97 *SPIE*, Orlando (1997), pp. 670-681

108. M. Heidelnan, D. Johnson, C. Burrus: "Gauss and the History of the Fast Fourier Transform," IEEE Transactions on Acoustics, Speech and Signal Processing Magazine 34, 265-267 (1985)
109. C. Burrus: "Index Mappings for Multidimensional Formulation of the DFT and Convolution," IEEE Transactions on Acoustics, Speech and Signal Processing 25,239-242 (1977)
110. B. Baas: (1998), "SPIFFEE an Energy-Efficient Single-Chip 1024-Point FFT Processor," http://www.stanford.edu/bbaas/spiffeel.html
111. G. Sunada, J. Jin, M. Berzins, T. Chen: "COBRA: An 1.2 Million Transistor Exandable Column FFT Chip," in Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (IEEE Computer Society Press, Los Alamitos, California, USA, 1994), pp. 546-550
112. Texas Memory Systems: "TM-66 swiFFT Chip," http://www.texmexnsys.com (1996)
113. Sharp Microeletronics: "BDSP9124 Digital Signal Processor," http://www.butterflydsp.com (1997)
114. J. Mellott (1997): "Long Instruction Word Computer," Ph.D. thesis, University of Florida, Gainesville
115. P. Lavoie: "A High-Speed CMOS Implementation of the Winograd Fourier Transform Algorithm," IEEE Transactions on Signal Processing 44(8), 2121-2126 (1996). Http://www.dreo.dnd.ca/pages/electdev/ewd011.htm
116. G. Panneerselvam, P. Graumann, L. Turner: "Implementation of Fast Fourier Transforms and Discrete Cosine Transforms in FPGAs," in Lecture Notes in Computer Science 1142, 272-281 (Springer, Heidelberg, 1996),
117. Altera Corporation: "Fast Fourier Transform," in Solution Brief 12, Altera Corparation (1997)
118. G. Goslin: "Using Xilinx FPGAs to Design Custom Digital Signal Processing Devices," in Proceedings of the DSP^x (1995), pp. 595-604
119. C. Dick: "Computing 2-D DFTs Using FPGAs," Lecture Notes in Computer Science: Field-Programmable Logic 1142, pp. 96-105 (Springer, Heidelberg,1996)
120. S.D. Stearns, D.R. Hush: Digital Signal Analysis (Prentice Hall, Englewood Cliffs, New Jersey, 1990)
121. K. Kammeyer, K. Kroschel: Digitale Signalverarbeitung (Teubner Studienbücher, Stuttgart, 1989)
122. E. Brigham: FFT, 3rd edn. (Oldenbourg Verlag, München Wien, 1987)
123. R. Ramirez: The FFT: Fundamentals and Concepts (Prentice Hall, Englewood Cliffs, New Jersey, 1985)

124. R.E. Blahut: Theory and Practice of Error Control Codes (Addison-Wesley,Melo Park, California, 1984)
125. C. Burrus, T. Parks: DFT//FFT and Convolution Algorithms (John Wiley & Sons, New York, 1985)
126. D. Elliott, K. Rao: Fast Transforms: Algorithms, Analyses, Applications (Academic Press, New York, 1982)
127. A. Nuttall: "Some Windows with Very Good Sidelobe Behavior," IEEE Transactions on Acoustics, Speech and Signal Processing ASSP-29(1), 84-91 (1981)
128. U. Meyer-Bäse, K. Datum (1988): "Fast Fourier Transform Using Signal Processor," Master's thesis, Department of Information Science, Darmstadt University of Technology
129. M. Narasimha, K. Shenoi, A. Peterson: "Quadratic Residues: Application to Chirp Filters and Discrete Fourier Transforms," in IEEE International Conference on Acoustics, Speech, and Signal Processing (1976), pp. 12-14
130. C. Rader: "Discrete Fourier Transform When the Number of Data Samples is Prime," Proceedings of the IEEE 56, 1107-8 (1968)
131. J. McClellan, C. Rader: Number Theory in Digital Signal Processing (Prentice Hall, Englewood Cliffs, New Jersey, 1979)
132. I. Good: "Tile Relationship between Two Fast Fourier Transforms," IEEE Transactions on Computers 20,310-317 (1971)
133. L. Thomas: "Using a Computer to Solve Problems in Physics," in Applications of Digital Computers (Ginn, Dordrecht, 1963)
134. A. Dandalis, V. Prasanna: "Fast Parallel Implementation of DFT Using Configurable Devices," Lecture Notes in Computer Science 1304, 314-323 (Springer, Heidelberg, 1997)
135. U. Meyer-Bäse, S. Wolf, J. Mellott, F. Taylor: "High Performance Implemenration of Convolution on a Multi FPGA Board Using NTT's Defined Over the Eisenstein Residuen Number System," in Aerosense 97 *SPIE*, Orlando(1997), pp. 431-442
136. Z. Wang: "Fast Algorithms for the Discrete W Transfrom and for the discrete Fourier Transform," IEEE Transactions on Acoustics, Speech and Signal Processing pp. 803-816 (1984)
137. M. Narasimha, A. Peterson: "On the Computation of the Discrete Cosine Transform," IEEE Transaction on Communications 26(6), 934-936 (1978)
138. K. Rao, P. Yip: Discrete Cosine Transform (Academic Press, San Diego, California, 1990)
139. B. Lee: "A New Algorithm to Compute the Discrete Cosine Transform," IEEE Transactions on Acoustics, Speech and Signal Processing 32(6), 1243-1245(1984)
140. S. Ramachandran, S. Srinivasan, R. Chen: "EPLD-Based Architecture of Real Time 2D Discrete Cosine Transform and Quantization for Image Compression," in IEEE International Symposium on Circuits and Systems Vol. III(1999), pp. 375- 378
141. C. Burrus, P. Eschenbacher: "An In-Place, In-Order Prime Factor FFT Algorithm," IEEE

Transactions on Acoustics, Speech and Signal Processing 29(4),806-817 (1981)

142. J. Pollard: "The Fast Fourier Transform in a Finite Field," Mathematics of Computation 25,365-374 (1971)
143. F. Taylor: "An RNS Discrete Fourier Transform Implementation," IEEE Transactions on Acoustics, Speech and Signal Processing 38, 1386-1394 (1990)
144. C. Rader: "Discrete Convolutions via Mersenne Transforms," IEEE Transactions on Computers C-21, 1269-1273 (1972)
145. Leibowitz: "A Simplified Binary Arithmetic for the Fermat Number Transform," IEEE Transactions on Acoustics, Speech and Signal Processing 24,356-359 (1976)
146. N. Bloch: Abstract Algebra with Applications (Prentice Hall, Englewood Cliffs, New Jersey: 1987)
147. J. Lipson: Elements of Algebra and Algebraic Computing (Addison-Wesley, London, 1981)
148. R. Agrawal, C. Burrus: "Fast Convolution Using Fermat Number Transforms with Applications to Digital Filtering," IEEE Transactions on Acoustics, Speech and Signal Processing 22, 87-97 (1974)
149. W. Siu, A. Constantinides: "On the Computation of Discrete Fourier Transform Using Fermat Number Transform," IEE Proceedings F 131, 7-14 (1984)
150. J. McClellan: "Hardware Realization of the Fermat Number Transform,"IEEE Transactions on Acoustics, Speech and Signal Processing 24(3), 216-225 (1976)
151. Texas Instruments: "User's Guide," TMS320C50, p. 7-51 (1993)
152. I. Reed, D. Tufts, X. Yu, T. Truong, M.T. Shih, X. Yin: "Fourier Analysis and Signal Processing by Use of the Möbius Inversion Formula," IEEE Transactions on Acoustics, Speech and Signal Processing 38(3), 458-470 (1990)
153. H. Park, V. Prasanna: "Modular VLSI Arichitectures for Computing the Arithmetic Fourier Transform," IEEE Transactions on Signal Processing 41(6), 2236-2246 (1993)
154. H. Lüke: Signalübertragung (Springer, Heidelberg, 1988)
155. D. Herold, R. Huthinann (1990): "Decoder for the Radio Data System (RDS)Using Signal Processor TMS320C25," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
156. U. Meyer-Bäse, R. Watzel: "A Comparison of DES and LFSR-based FPGA Implementable Cryptography Algorithms," in 3rd International Symposium on Communication Theory & Applications (1995), pp. 291-298
157. U. Meyer-Bäse, R. Watzel: "An Optimized Format for Long Frequency Paging Systems," in 3rd International Symposium on Communication Theory &Applications (1995), pp. 78-79
158. U. Meyer-Bäse: "Convolutional Error Decoding with FPGAs," Lecture Notes in Computer Science 1142, 376-175 (Springer, Heidelberg, 1996)
159. R. Watzel (1993): "Design of Paging Scheme and Implementation of the Suitable

Cryto-Controller Using FPGAs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology

160. J. Maier, T. Schubert (1993): "Design of Convolutional Decoders Using FPGAs for Error Correction in a Paging System," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
161. Y. Gao, D. Herold, U. Meyer-Bäse: "Zum bestehenden Übertragungsprotokoll kompatible Fehlerkorrektur," in Funkuhren Zeitsignale Normalfrequenzen(1993), pp. 99-112
162. D. Herold (1991): "Investigation of Error Corrections Steps for DCF77 Signals Using Programmable Gate Arrays," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
163. D. Wiggert: Error-Control Coding and Applications (Artech House, Dedham, Mass., 1988)
164. G. Clark, J. Cain: Error-Correction Coding for Digital Communications(Plenum Press, New York, 1988)
165. W. Stahnke: "Primitive Binary Polynomials," Mathematics of Computation pp. 977-980 (1973)
166. W. Fumy, H. Riess: Kryptographie (R. Oldenbourg Verlag, München, 1988)
167. B. Schneier: Applied Cryptography (John Wiley & Sons, New York, 1996)
168. M. Langhammer: "Reed-Solomon Codec Design in Programmable Logic," Communication System Design (www.csdmag.com) pp. 31-37 (1998)
169. B. Akers: "Binary Decusion Diagrams," IEEE Transactions on Computers pp. 509-516 (1978)
170. R.Bryant: "Graph-Based Algorithms for Boolean Function Manipulation,"IEEE Transactions on Computers pp. 677-691 (1986)
171. A. Sangiovanni-Vincentelli, A. Gamal, J. Rose: "Synthesis Methods for Field Programmable Gate Arrays," Proceedings of the IEEE pp. 1057-83 (1993)
172. R. del Rio (1993): "Synthesis of boolean Functions for Field Programmable Gate Arrays," Master's thesis, Univerity of Frankfurt, FB Informatik
173. U. Meyer-Bäse: "Optimal Strategies for Incoherent Demodulation of Narrow Band FM Signals," in 3rd International Symposium on Communication Theory & Applications (1995), pp. 30-31
174. J. Proakis: Digital Communications (McGraw-Hill, New York, 1983)
175. R.Johannesson: "Robustly Optimal One-Half Binary Convolutional Codes,"IEEE Transactions on Information Theory pp. 464 8 (1975)
176. J. Massey, D. Costello: "Nonsystematic Convolutional Codes for Sequential Decoding in Space Applications," IEEE Transactions on Communications pp. 806-813 (1971)
177. F. MacWilliams, J. Sloane: "Pseudo-Random Sequences and Arrays," Proceedings of the IEEE pp. 1715-29 (1976)
178. T. Lewis, W. Payne: "Generalized Feedback Shift Register Pseudorandom Number

Algorithm," Journal of the Association for Computing Machinery pp. 456-458 (1973)

179. P. Bratley, B. Fox, L. Schrage: A Guide to Simulation (Springer-Lehrbuch, Heidelberg, 1983), pp. 186-190

180. M. Schroeder: Number Theory in Science and Communication (Springer, Heidelberg, 1990)

181. Electronic Frontier Foundation: Cracking DES (O'Reilly & Associates, Sebastopol, 1998)

182. W. Stallings: "Encryption Choices Beyond DES," Communication System Design (www. csdmag, com) pp. 37-43 (1998)

183. W. Carter: "FPGAs: Go Reconfigure," Communication System Design(www. csdmag, com) p. 56 (1998)

184. J. Anderson, T. Aulin, C.E. Sundberg: Digital Phase Modulation (Plelmm Press, New York, 1986)

185. U.Meyer-Bäse (1989): "Investigation of Threshold Improving Limiter/Discriminator Demodulator for FM Signals Through Computer Simulations," Master's thesis, Department of Information Science, Darmstadt University of Technology

186. E.Allmann, T. Wolf (1991): "Design and Implementation of a Fully Digital Zero IF Receiver Using Programmable Gate-Arrays and Floating-Point DSPs," Master's thesis, Institute for Data Technics, Darmstadt University of Technology

187. O. Herrmann: "Quadraturfilter mit rationalem Übertragungsfaktor," Archiv der elektrischen Übertragung pp. 77-84 (1969)

188. O. Herrmann: "Transversalfilter zur Hilbert-Transformation," Archiv der elektrischen Übertragung pp. 581-587 (1969)

189. V. Considine: "Digital Complex Sampling," Electronics Letters pp. 608-609(1983)

190. T.E. Thiel, G.J. Saulnier: "Simplified Complex Digital Sampling Demodulator," Electronics Letters pp. 419-421 (1990)

191. U. Meyer-Bäse, W. Hilberg: (1992), "Schmalbandempfänger für Digitalsignale," German Patent No. 4219417.2-31

192. B. Schlanske (1992): "Design and Implementation of a Universal Hilbert Sampling Receiver with CORDIC Demodulation for LF Fax Signals Using digital Signal Processor," Master's thesis, Institute for Data Technics, Darmstadt University of Technology

193. A. Dietrich (1992): "Realisation of a Hilbert Sampling Receiver with CORDIC Demodulation for DCF77 Signals Using Floating-Point Signal Processors," Master's thesis, Institute for Data Technics, Darmstadt University of Technology

194. A. Viterbi: Principles of Coherent Communication (McGraw-Hill, New York, 1966)

195. F. Gardner: Phaselock Techniques (John Wiley & Sons, New York, 1979)

196. H. Geschwinde: Einführung in die PLL-Technik (Vieweg, Braunschweig, 1984)

197. R. Best: Theorie und Anwendung des Phase-locked Loops (AT Press, Schwitzerland, 1987)

198. W. Lindsey, C. Chie: "A Survey of Digital Phase-Locked Loops," Proceedings of the IEEE

pp. 410-431 (1981)

199. R. Sanneman, J. Rowbotham: "Unlock Characteristics of the Optimum Type II Phase-Locked Loop," IEEE Transactions on Aerospace and Navigational Electronics pp. 15-24 (1964)
200. J. Stensby: "False Lock in Costas Loops," Proceedings of the 20th Southeastern Symposium on System Theory pp. 75-79 (1988)
201. A. Mararios, T. Tozer: "False-Lock Performance Improvement in Costas Loops," IEEE Transactions on Communications pp. 2285-88 (1982)
202. A. Makarios, T. Tozer: "False-Look Avoidance Scheme for Costas Loops,"Electronics Letters pp. 490-2 (1981)
203. U. Meyer-Bäse: "Coherent Demodulation with FPGAs," Lecture Notes in Computer Science 1142, 166-175 (Springer, Heidelberg, 1996)
204. J. Guyot, H. Schmitt (1993): "Design of a Fully Digital Costas Loop Using Programmable Gate Arrays for Coherent Demodulation of Low Frequency Signals," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
205. R. Resch, P. Schreiner (1993): "Design of Fully Digital Phase-Locked Loops Using programmable Gate-Arrys for a Low-Frequency Reciever," Master's thesis, Institute for Data Technics, Darmstadt University of Technology
206. D. McCarty: "Digital PLL suits FPGAs," Elektronic Design p. 81 (1992)
207. J. Holmes: "Tracking-Loop Bias Due to Costas Loop Arm Filter Imbalance,"IEEE Transactions on Communications pp. 2271-3 (1982)
208. H. Choi: "Effect of Gain and Phase Imbalance on the Performance of Lock Detector of Costas Loop," IEEE International Conference on Communications,Seattle pp. 218-222 (1987)

[General Information]
书名=数字信号处理的FPGA实现
作者=
页数=360
SS号=11063035
出版日期=