密级:_____

中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

参数化半精度浮点 CORDIC 处理器的研究

作者姓名：李尚应

指导教师：陈云霁 研究员

中国科学院计算技术研究所

学位类别：工学硕士

学科专业：计算机体系结构

培养单位：中国科学院计算技术研究所

2017 年 05 月

Research on Parameterized Half-Precision Floating-Point CORDIC Processor

by

Shangying Li

A dissertation submitted to
The University of Chinese Academy of Sciences
in partial fulfillment of the requirements
for the degree of
Master of Computer Architecture

Institute of Computing Technology, Chinese Academy of Sciences

May, 2017

学位论文独创性声明

本人郑重声明：我所呈交的学位论文是本人在导师指导下进行的研究工作及所取得的研究成果。尽我所知，除了文中已经标注引用的内容外，本论文中不含其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中作了明确的说明或致谢。本人知道本声明的法律结果由自己承担。

作者签名：李尚应　日期：2017-5-31

关于学位论文使用授权的说明

本人完全了解中国科学院计算技术研究所有关保留、使用学位论文的规定，即：中国科学院计算技术研究所有权保留送交论文的复印件，允许论文被查阅和借阅；可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

作者签名：李尚应　导师签名：　日期：2017-5-31

摘 要

基本超越函数，如 sin，cos，tan，$\sin^{-1}$，$\cos^{-1}$，$\tan^{-1}$，sinh，cosh，tanh，$\sinh^{-1}$，$\cosh^{-1}$，$\tanh^{-1}$，指数，对数的计算在科学计算中占有重要地位。计算它们的方法大体上分为基于多项式近似的方法，基于查表的方法和移位 -加法类算法。

移位 -加法类算法可以在硬件上实现基本超越函数。CORDIC 算法就属于移位 -加法算法类，它是一个计算超越函数的一个简单有效的算法。目前各方研究已经提出了很多 CORDIC 算法的变种，来弥补 CORDIC 算法串行性的缺陷，提高计算速度。遗憾的是，这些变种并未涉及浮点数领域。

本文提出了一个半精度浮点 CORDIC 处理器，提供对常见 CORDIC 算法中函数的支持。

本文的贡献在于：

1）将 CORDIC 处理器拓展至浮点数领域，将输入转换为定点数进行处理，并将算法的输出转换为浮点数；

2）使用参数缩减算法拓展 CORDIC 算法的收敛域来支持所有半精度浮点机器数，通过预处理将输入分割成 CORDIC 算法收敛范围内的数和预处理信息，在得到原始结果后通过后处理将原始结果和预处理信息重建成计算结果；

3）通过数学等式使用分步手段支持反正弦和反双曲正弦函数。

我们的处理器对所有 CORDIC 函数的平均相对误差小于 0.05%，并得到相对于 CPU 平均 9.43 的加速比，意味着它可以被用作科学计算的一个加速器。

4）本文还提出了一个基于查表的非线性函数运算装置，以在精度要求不高的时候进一步加快运算速度，降低芯片面积和功耗。

关键词： 基本函数，科学计算，CORDIC 算法，硬件加速器

Abstract

Elementary transcendental functions, such as $\sin, \cos, \tan, \sin^{-1}, \cos^{-1}, \tan^{-1}$, $\sinh, \cosh, \tanh, \sinh^{-1}, \cosh^{-1}, \tanh^{-1}$,, exponents, logarithms, play a crucial role in scientific computing. The method for computing them can generally be classified as polynomial approximation and/or table lookup, and shift-and-add algorithms.

Shift-and-add algorithms can implement elementary functions in hardware. CORDIC algorithm is one of them, it is an efficient way to compute transcendental functions. To compensate for CORDIC algorithm's sequential nature and accelerate computing, there has been many variation of CORDIC proposed. Unfortunately these variations didn't touch floating-point numbers.

This paper proposes a half-precision floating point CORDIC processor, provides support for usual CORDIC functions. The contribution of this paper is the following:

1) expand CORDIC processor to floating-point numbers, convert floating-point numbers to fix-point numbers for CORDIC processing, and convert result back to floating-point numbers;

2) expand convergence range to all half-precision floating point numbers, split input into a number within CORDIC convergence range and pre-processing information via pre-processing, and combine original result and pre-processing information via post-preprocessing;

3) support inverse sin and inverse sinh functions use mathematical identities and a two-step algorithms.

On average, the supported functions achieve precision of less than 0.05% on our processor. Our processor achieve an average speedup of 9.43 than CPU, meaning it could be used as an accelerator for scientific computing.

4) This paper also proposes a device for computing nonlinear functions, which significantly reduces computing time, silicon area and power consumption, at a mild expense of accuracy.

Keywords: Elementary functions, Scientific computing, CORDIC algorithm, Hardware accelerators

目 录

图形列表

表格列表

第一章 引言

1.1 CORDIC 算法的背景

CORDIC(COordinate Rotation DIgital Computer)，也被称作 Volder 算法，是一个计算三角函数和双曲函数的一个简单有效的算法。CORDIC 算术的基本概念是基于简单古老的二维几何原理。但它作为计算机算法的迭代实现是由 Jack E. Volder 在 1959 年第一次描述 [1] [2]，用于三角函数、乘法、除法的计算。直到今天，CORDIC 算法完成已有 58 年。在这 58 年里，不仅兴起了 CORDIC 算法的多种多样的应用，而且在针对这些应用的算法的设计实现领域也有了很大进展。在 1971 年，John Walther 阐述了 [3][4] 通过改变一些参数得到一个基本超越函数的统一算法，这个算法在 Volder 所能做到的 [1] 应用范围之外还包括对数、指数、开平方，从而使得 CORDIC 算法受到了更多的关注。同时，Cochran 对多种算法的性能进行了测评 [5]，并展示了 CORDIC 技术对科学计算应用是上佳的选择。

从此以后，CORDIC 的受青睐程度大大提高，主要是因为它支持对广泛的应用的高效低功耗实现：生成三角、对数、超越函数；复数乘法、特征值计算、矩阵取逆、解线性系统以及奇异值分解 (SVD)，用于信号处理、图像处理以及一般的科学计算。其他流行与将来的应用有：

1) 用于语音/音乐合成和通信的直接频率合成、数字调制和数字编码；

2) 用于机器人操作的直接运动学和逆运动学计算；

3) 用于图形和动画的平面和三维向量旋转。

尽管 CORDIC 在进行这些操作的技术中可能不全是最快的，但其算法的吸引力在于硬件实现简单——使用移位 -加法形式 $a \pm b \times 2^{-i}$ 操作的应用。

在保持不同应用的要求和约束环境的视角时，CORDIC 算法和架构的发展已经实现高吞吐率，低硬件复杂性和低实现时的延迟。一些用于降低实现复杂性的典型难题在于包括最小化缩放实现的复杂性和 CORDIC 引擎中桶形移位器的复杂性。常规 CORDIC 算法的一个固有缺点是实现时的延迟。角度重编码方案、混合粒度旋转和高基数 CORDIC 是几种试图降低延迟的方法。在提高吞吐率方面，一般采用并行和流水线 CORDIC 等方法解决。

本文的目的是对 CORDIC 算法、架构和应用的发展进行详细的调查研究，而后提出一个改进型流水线 CORDIC 算法并将其实现。

1.2 CORDIC 算法最新的研究进展

目前 CORDIC 算法领域最新算法的进展如下：

文献 [6] 介绍了一种增强的 CORDIC 双旋转方法，适用于椭圆和双曲模式，避免了比例因子和适用于冗余算术。

文献 [7] 将 CORDIC 算法分为两步，第一步粗糙地近似正弦和余弦函数，第二步在 CORDIC 旋转时对值进行微调。结果是 CORDIC 的算法步骤得到减小，硬件开销得到减少。

文献 [8] 推荐了一种新的基数 -4 双步分支混合 CORDIC 算法，可将迭代次数（包括计算比例因子和补偿）减少到 $(3n/8)+1$，其中 n 是精度。

文献 [9] 提出了一个新的角度集，与典型 CORDIC 使用的角度集不同，以达到更快的收敛并减少加法器的数量。

文献 [10] 在 CORDIC 算法上运用规范符号数位算术 (CSD)，模拟结果显示如果旋转角度被合适地分解，则可显著提高精度。VLSI 实现结果显示该基于 CSD 的 CORDIC 算法显著减少典型 CORDIC 算法的运行时间。

文献 [11] 使用快速量纲估计和前处理缩放来降低使用 CORDIC 计算反正切函数的误差。

1.3 CORDIC 算法硬件架构实现的近期进展

文献 [12] 在 Xilinx FPGA 上实现 CORDIC 算法来计算正弦和余弦函数。

文献 [13] 介绍了一种计算 -跳过 (computation-skip) 方案来达到积极的电压和频率缩放，降低了 CORDIC 处理器的能耗。

文献 [14] 使用流水线和复用器来分别减少 CORDIC 算法关键路径的延迟和面积开销。

文献 [15] 将 CORDIC 算法的圆模式和双曲模式合并为在一个电路中的可重构 CORDIC，与分别存在两个模式相比达到了显著的面积减少。

文献 [16] 考虑基于 CORDIC 的离散余弦变换 (DCT)。DCT 中不是所有的计算都具有同样的重要程度，因此 CORDIC 的迭代次数可被动态调整。这样，可在不影响图片质量的情况下显著降低运算能耗。

文献 [17] 提出了一个混合 FPGA 架构，能够以一个小的面积开销为代价降低延迟 (42%)，准确度和典型 CORDIC 相似。

文献 [18] 在一个全数字 FM 调制解调器内实现了一个流水线式 CORDIC 架

构，整个系统（包括 FM 调制解调器）具有 3911 个逻辑单元，233.33 纳秒延迟和最大频率 60MHz。

文献 [19] 介绍了基于扩展过的双曲 CORDIC 算法的浮点算术乘方计算。其带参数的硬件实现允许他们找到设计参数（数字格式，迭代次数等），资源使用，精度和运行时间之间的平衡。

文献 [20] 提出了一个低延迟混合 CORDIC 架构，首先通过自适应角度选择减少 50% 的迭代次数，其次使用定点数输入和浮点数输出的混合架构减少总硬件开销和提升最终结果的动态范围，最后通过并行和流水线过程和共享资源技术来达到 175.7MHz 和 1139LUT、489 寄存器的低资源消费。

文献 [21] 专注于在比特 -并行的展开 CORDIC 架构中减少功耗。他们对切换活动和关键路径上的电容的充电 -放电进行建模，通过 1）将频繁活动的节点隐藏于查表中来减少切换活动 2）重定时结构来减少关键路径和与之相关的电容充电 -放电，达到了动态功耗百分之 10 至 20 的减少和总功耗百分之 35 至 40 的减少。

1.4 本文 CORDIC 处理器的背景

现场可编程门阵列 (Field-Programmable Gate Arrays, FPGAs) 在各种商业高性能计算和嵌入式系统中日趋频繁地被使用。为了提高其计算性能性能，浮点数操作已被深入研究和优化。另外，更高级的操作例如三角函数，已得到了更多的关注并广泛使用。因此，一个可以在浮点数下计算各种超越函数的装置十分必要。浮点 CORDIC 处理器已被证明是一个有效的解决方案，近来研究表明，单精度 [22]，双精度 [23]，甚至四精度 [24]CORDIC 处理器能在一个芯片上实现。然而，它们中的大多数只考虑 CORDIC 算法的一种模式下超越函数的计算，并具有高延迟和高硬件开销。实际上，在嵌入式系统应用中，很多时候使用半精度浮点表示以减少硬件面积和延迟，导致以上高精度 CORDIC 处理器没有多少用武之地。

为了解决这个问题，本文提出一种新的半精度浮点 CORDIC 处理器，支持所有已知的 CORDIC 函数，具有三个模组：预处理器，核心单元（微旋转），后处理器。预处理器转换任何 16 位浮点输入为一定点格式，接下来使用参数缩放来拓展收敛域。带参数的核心单元由一系列流水组成，以实现典型 CORDIC 旋转。后处理器接受预处理器的参数缩放信息来得到最终结果，并将其正则化为 IEEE 754 半精度浮点格式。更进一步地，我们拓展 CORDIC 算法来支持反正弦函数和反双曲正弦函数。我们还在设计中引入线性近似技术来减少数据宽度和削减迭代次数而又不损失精度。

我们的 CORDIC 处理器的目标是以如下的特点计算超越函数：

1）一个完全流水化架构来降低延迟。

2）支持所有已知 CORDIC 函数，包括反正弦和反双曲正弦函数，并不限制输入所在域。

3）采用线性近似，用最小的数据宽度和迭代次数来得到所需精度

4）带参数的设计理念，在硬件空间，时钟频率和功耗之间达到平衡。

1.5 本文结构

本文的第一章介绍本文工作的背景；第二章将介绍 CORDIC 之外的基本函数的各种算法以便与 CORDIC 比较；第三章描述 CORDIC 算法的基本思想和 CORDIC 算法在目前他人研究下的各种变种；第四章介绍我们的工作——半精度浮点 CORDIC 处理器；第五章介绍我们的另一项工作——基于查表的非线性函数运算装置；第六章为结论。

第二章　基本函数的各种算法

本文着重讨论基本函数的运算。这里，我们称基本函数为常用的数学函数：$\sin, \cos, \tan, \sin^{-1}, \cos^{-1}, \tan^{-1}, \sinh, \cosh, \tanh, \sinh^{-1}, \cosh^{-1}, \tanh^{-1}$, 指数，对数。我们应当说“基本超越函数“，从数学上讲，$1/x$ 和 e^x 一样都是基本函数；本文不讨论基本算术函数。理论上，这些基本函数并不比除法更难算得多：Alt[25] 展示了这些函数在布尔电路深度方面与除法等价。这意味着，一个电路可以在与 $\log n$ 等比的时间内输出 n 位的正弦，余弦或对数。然而在现实的使用场景中，如果我们需要快速并精确地计算基本函数，我们就需要使用特别的优化。

上述话题在早期已有 Cody 和 Waite[26] 等人的研究，然而当时这些函数只是在软件里被实现，也没有浮点数算术的标准。在 Intel 8087 浮点数单元出现之后，基本函数开始被在硬件中实现（至少部分如此）。在通用计算领域，虽然软件有着灵活性较高的特点，但针对基本超越函数的专用硬件架构也十分需要。

更进一步地，随着高质量算术标准的出现（例如 IEEE-754 浮点数算术标准），和数学家和计算机科学家们如 W.Kahan, W.Cody, H.Kuki, P.Markstein, P.Tang 的决定性工作，用户已习惯于非常精确的结果。二十年前一个提供基本函数的一位或两位精度结果的库被认为足够了 [27]，但是现在的电路或库设计者必须构建有更精确保证的算法和架构（至少对通用式系统）。一些库甚至提供舍入正确的函数：返回的结果总是等于最接近精确结果的机器数。我们可以列出以下几种目前基本超越函数运算方面的需求：

1）速度；

2）准确度；

3）合理数量的资源（例如 ROM/RAM，使用硬件的面积）

4）保留重要的数学性质，例如单调性和对称。Silverstein 等人的文献 [28] 中指出，未保留单调性可导致微分计算的问题；

5）保留舍入方向。例如，若选择的舍入函数是向 $-\infty$ 舍入，返回值必须小于等于精确值。

6）范围限制：得到一个大于 1 的正弦函数可导致错误，例如，计算 [28]

$$\sqrt{1-\sin^2 x}$$

文献 [29] 详尽地介绍了与基本函数有关的各种概念和方法。

2.1 浮点运算的基本概念

这里我们主要关注 2008 年发布的 IEEE 754 浮点数标准指定的二进制格式。IEEE754 在 1985 年的第一次发布是提升对程序员所用的计算环境的重要原因。在此标准以前，浮点数算术无法付诸实用，有时能工作得很好，有时根本不可行。

众所周知一个基数 β，精度 p 的浮点数是一个如下形式的数字

$$\pm m \times \beta^e$$

其中 m 用 p 位和基数 β 表示，$m < \beta$，而且 e 是一个整数。然而，构造可信任的算法和证明需要一个更正式的定义。

一个浮点数格式可以部分地以四个整数为特征：

1）一个基数（基底）$\beta \geq 2$；

2）一个精度 $p \geq 2$（p 是该表示里的显著位）；

3）两个边界指数 e_{min} 和 e_{max} 其中 $e_{min} < e_{max}$。在所有实用情况下，$e_{min} < 0 < e_{max}$。

在这种格式下一个有限的浮点数是一个数 x，对它至少存在一种表示 (M, e) 满足

$$x = M \cdot \beta^{e-p+1},$$

这里

1）M 是一个绝对值小于等于 $\beta^p - 1$ 的数。它被叫做 x 在该表示下的整数有效位；

2）e 是一个整数，$e_{min} \leq e \leq e_{max}$ 叫做 x 在该表示下的指数。

我们现在回到前面的表示，注意到我们可以定义 $m = |M| \cdot \beta 1 - p$ 和 $s = 0$ 若 $x \geq 0$，1 若不然，那么

$$x = (-1)^s \cdot m \cdot \beta^e.$$

1）m 被叫做实有效位（或更简单地，该表示的有效位）。它在基数点前有一位，在基数点后最多有 $p - 1$ 位；

2）s 是 x 的符号。

注意对有些数 x，可能存在好几种不同的表示 (M, e) 或 (s, m, e)。例如 $\beta = 10$ 和 $p = 4$。在这种格式下 $M = 4560$ 和 $e = -1$，和 $M = 0456$ 和 $e = 0$ 都是 0.456 的有效表示。

为了得到唯一表示，我们可能想要标准化有限非零浮点数，这通过选择指数最小的一个表示来实现（然而指数仍然需要大于 e_min）。得到的表示为标准化表示。

1）总体上，若这样的表示满足 $1 \leq |m| < \beta$，或等价地，$\beta^{p-1} \leq |M| \leq \beta^p$。在这种情况下，我们称 x 为一个标准数。

2）若不满足，必有 $e = e_{min}$。对应的 x 值称为亚标准数（也常被称作非标准数）。在那种情况下，$|m| < 1$，或等价地，$|M| \leq \beta p - 1 - 1$。注意一个亚标准数的绝对值小于 $\beta^{e_{min}}$，亚标准数是非常小的数。

一个标准化带来的有趣结果是，当基数 β 等于 2 时，一个标准数的有效位的第一个比特必须是”1”，一个亚标准数的第一个比特必须是”0”。这样如果我们有 x 是否标准的信息（实际应用中这信息是在指数位中编码的），就没有必要存储它的第一个有效位；在很多计算机系统里它是不存储的（这叫做”隐含比特”惯例）。例如，在 IEEE754 双精度/binary64 格式下能表示的最大有限数为

$$(2 - 2^{-52}) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308}$$

最小的正数为

$$2^{-1074} \approx 4.940656458412465 \times 10^{-324}$$

最小的标准正数为

$$2^{-1022} \approx 2.225073858507201 \times 10^{-308}$$

基于基数 10 的算术经常被用于便携式计算器中。并且，它被用于财经计算，同时 2008 版的 IEEE 754 制定了几个十进制格式。十进制算术仍是一个活跃的研究领域。一个叫做 SETUN 的俄罗斯计算机使用基数 3，每位可取 -1,0,1（这叫做平衡三进制系统）。它在 19 世纪 60 年代诞生于莫斯科大学。几乎所有其他的现代计算系统使用基数 2。各方研究 [30][31][32] 表明基数 2 在有隐含比特惯例的情况下给出比其他基数更高的准确率。

2.1.1 IEEE 754 半精度二进制浮点数格式

IEEE 754 标准 [33] 制定了一个二进制数，具有三个组成部分：一个符号位，一部分指数位和一部分尾数位。符号位指示该数的符号，0 代表正数、1 代表负数。指数位经过一个常数修正后表示大小的指数阶，尾数决定该数的具体大小。正则表达为：

$$(-1)^{sign} \times 2^{exponents-bias} \times 1.mantissa$$

根据这个标准，IEEE754 半精度浮点数具有 1 个符号位，5 个指数位（修正常数 bias 为 15），和 10 个尾数位。

并且，指数位为 $(00000)_2$ 和 $(11111)_2$ 时被解释为亚正常数和非数 (NaN)。亚正常数如下表示：

$$(-1)^{sign} \times 2^{-14} \times 0.mantissa$$

最小的严格正数（亚正常）为 2^{-24}。最小的正正常数为 2^{-14}。最大绝对值为 $(2-2^{-10})2^{15} = 65504$。

2.2 冗余数字系统

通常，我们表示一个基数 r 的数时，我们使用 $0, 1, 2, ..., r-1$。然而，有时自然出现使用不同集合位数的数字系统。1840 年，柯西建议在基数 10 中使用数位 -5 到 +5 来简化乘法。Booth 重编码（一个有时被乘法器设计者采用的基数）生成以基数 2 表示的数，数位取 -1,0,+1。除法和开方的数位 -重现算法 [34] 同样生成一个“符号数位”表示。

其中有些特别的数字系统允许无进位的加法，这是本部分需要研究的内容。

假设我们要计算两个整数 $x = x_{n-1}x_{n-2}...x_0$ 和 $y = y_{n-1}y_{n-2}...y_0$ 的和 $s = s_n s_{n-1} s_{n-2}...s_0$，以传统二进制数字系统表示。通过检查众所周知的描述加法过程的方程（"$\vee$" 是布尔"OR"，$\oplus$ 是布尔"XOR"）

$$c_0 = 0 \tag{2.1}$$

$$s_i = x_i \oplus y_i \oplus c_i \tag{2.2}$$

$$c_{i+1} = x_i y_i \vee x_i c_i \vee y_i c_i \tag{2.3}$$

我们发现在 i 位置的输入进位 c_i 和 c_{i+1} 之间有依赖关系。这不意味着加法过程是本质上串行的，或者两个数的和必须在与操作数长度成线性关系的时间内完成：文献 [35] [36] 等中提出的加法算法和架构比直接显然的纯串行实现要快得多。尽管如此，进位之间的依赖关系使得一个完全并行的加法在传统数字系统下变得不可能。

2.2.1 符号数位数字系统 [29]

在 1961 年，Avizienis[37] 研究了叫做符号数位数字系统的不同数字系统。假设我们使用基数 r。在一个符号数位数字系统里，数字不再以 0 至 $r-1$ 的数位表示，而是以 $-a$ 至 a，其中 $a \leq r-1$。如果 $2a \geq r-1$，每个数

都能被这系统表示。例如，在基数 10 下使用 -5 至 $+5$，每个数都是可表示的。数字 15725 可被表示成数位链 $2\bar{4}325$(我们使用 $\bar{4}$ 表示数位 -4)。即，$15725 = 2 \times 10^4 + (-4) \times 10^3 + (-3) \times 10^2 + 2 \times 10^1 + 5$.

同样一个数字也可以表示为 $2\bar{4}3\bar{3}5$。如果 $2a \geq r$，那么有些数字有多种可能的表示，意味着系统是冗余的。2.2.3 节将展示，这是一个很重要的性质。

Avizienis 也提出了对这些数字系统的加法算法。算法 1 执行两个 n 位数字 $x = x_{n-1}x_{n-2}...x_0$ 和 $y = y_{n-1}y_{n-2}...y_0$ 在基数 r 下表示使用数位 $-a$ 至 a 的加法，其中 $a \leq r-1$ 和 $2a \geq r+1$。

算法 1 Avizienis 算法

输入：$x = x_{n-1}x_{n-2}...x_0$ 和 $y = y_{n-1}y_{n-2}...y_0$

输出：$s = s_n s_{n-1} s_{n-2}...s_0 = x + y$

1）并行地，对 $i = 0, ..., n-1$，计算进位 t_{i+1} 和中间和 w_i，满足：

$$t_{i+1} = +1, if x_i + y_i \geq a, \quad = 0, if -a+1 \leq x_i + y_i \leq a-1, = -1, if x_i + y_i \leq -a w_i \quad = x_i + y_i - \tag{2.4}$$

2）并行地，对 $i = 0, ..., n-1$，计算 $s_i = w_i + t_i$，其中 $w_n = t_0 = 0$。

检查这个算法后，我们的结论是进位 t_{i+1} 并不依赖于 t_i。不再有任何的进位传递：结果的所有数位都可以同时生成。条件 $2a \geq r+1$ 和 $a \leq r-1$ 不能同时对基数 2 实现。然而，仍有可能实施并行的，无进位的基数 2 下加法，其中数位等于 -1,0,1。这是使用 Avizienis 的另一种算法实现的（或者使用下文描述的借位-储存加法器）。

冗余数字系统在很多场合下被使用：乘法器的重编码，除法或类除法操作中的余数，在线算术等。算术操作器例如乘法器和除法器经常在内部使用冗余加法（这些操作器的输入和输出数据以一种非冗余的数字表示，但内部计算在冗余数字表示上进行）。例如，大多数乘法器使用（至少隐含地）进位 -保存数字系统，而数位 -重现除法器实际上使用两种不同的数字系统：大体上，部分余数以进位 -保存表示，并且商的数位以基数 2^k 的符号数位数字系统表示，其中 k 是一个小整数。[34]

2.2.2 进位 -保存和借位 -保存数字系统 [29]

我们着重讨论基数 2 的情况。在这个基数下，两个常见的冗余数字系统是进位 -保存（Carry-Save, CS）数字系统，和符号数位数字系统。在进位 -保存数字

系统里，数字以 0,1,2 数位表示，而且每个位 d 以两个比特 $d^{(1)}$ 和 $d^{(2)}$ 表示，它们的和等于 d。在符号数位数字系统里，数字以数位 -1,0,1 表示。在那种系统里，我们可以以借位 -保存（borrow-save, BS）编码表示数位，也叫做 (p,n) 编码：每个数位 d 以两个比特 d^+ 和 d^- 其中 $d^+ - d^- = d$（不同的数位编码也能导出快速简单的算术操作器 [38]）。这两个数字系统下可以进行非常快的加法和减法。进位 -保存加法器是个很有名的例子，它用于在一个进位 -保存系统表示的数字和一个用典型二进制系统表示的数字上进行加法。它由一行全加器单元组成，一个全加器由三个比特 x,y,z 计算两个比特 t 和 u，结果是 $2t+u$ 等于 $x+y+z$。

一个用于借位 -保存数字系统的加法器结构可以由略不同于全加器单元的基本单元来建造。以下算法将两个借位 -保存数字相加。

算法：借位 -保存相加

输入：两个借位 -保存数字 $a = a_{n-1}a_{n-2}...a_0$ 和 $b = b_{n-1}b_{n-2}...b_0$ 其中数位 a_i 和 b_i 属于 -1,0,1，每个数位 d 由两个比特 d^+ 和 d^- 表示其中 $d^+ - d^- = d$。

输出：一个借位 -保存数字 $s = s_n s_{n-1}...s_0$ 满足 $s = a + b$。

对每一个 $i = 0,...,n-1$，计算两个比特 c_{i+1}^+ 和 c_i^- 使得 $2c_{i+1}^+ - c_i^- = a_i^+ + b_i^+ - a_i^-$;

对每一个 $i = 0,...,n-1$，计算 s_{i+1}^- 和 s_i^+ 使得 $2s_{i+1}^- - s_i^+ = c_i^- + b_i^- - c_i^+$（其中 $c_0^+ = c_n^- = 0, s_n^+ = c_n^+$）

这个算法的两步都需要同样的基本运算：从三个比特 x,y,z 找到两个比特 t 和 u 使得 $2t - u = x + y - z$. 这可以通过一个 PPM(Plus Plus Minus) 单元完成，十分相似于全加器。使用 PPM 单元，可以简单地构造借位 -保存加法器。有办法将一个由借位 -保存系统表示的数字和一个典型非冗余二进制系统下表示的数字相加，且只需要一行 PPM 单元。文献 [39] 提供了更多借位 -保存算术操作器的信息。

2.3 多项式近似

使用有限次的加法、减法、乘法和比较，我们只能计算一个变量的分段多项式函数。如果我们加入除法，我们只能计算分段有理函数。这样，自然我们试图用多项式或有理函数来逼近基本函数。立即涌现的问题有：

我们如何计算这样的多项式或有理近似？

什么是最好的方式（以精度或者速度度量）来计算一个多项式或者有理函数？

最终误差将会是两个误差之和：近似误差（即，要近似的函数和多项式或有

理近似之间的距离），和计算误差（因该多项式或有理函数是以有限精度浮点算术进行计算），我们能否计算这些误差的边界？

2.3.1 插值

若我们试图用多项式近似一个连续函数，首先涌现的两个想法是，大体上使用泰勒级数或者在某些点对函数插值。后面可以看到，总体上泰勒级数不是一个好的解决方案（至少对定精度来说）。在聪明选择地点上插值可能是一个好想法：在 Chebyshev 点上对函数进行插值能得到一个几乎和 minimax 多项式一样好的多项式。Chebyshev 插值是一个很有趣的工具，具有良好的数学性质和很多有用的应用 [40]。Trefethen 在文献 [41] 中注意到对特别大的度数，在一个函数上进行 Chebyshev 插值比计算 minimax 近似快得多。这解释了大多数数值应用倾向于插值。我们的研究场景倾向于构造一劳永逸的微调近似，并将此算法应用实现百万次。

虽然在 Chebyshev 点上进行插值是一个有意义的选项，在等间距的点上进行插值却是一个灾难性的解决方案，总体上会得到很差的近似。

2.3.2 最小二乘多项式近似

我们在寻找一个小于 n 次的多项式

$$p^*(x) = p_n^* x^n + p_{n-1}^* x^{n-1} + ... + p_1^* x + p_0^*$$

满足

$$\int_a^b w(x)(f(x) - p^*(x))^2 dx = \min_{p \in \mathcal{P}_k} \int_a^b w(x)(f(x) - p(x))^2 dx$$

. 定义 $\langle f, g \rangle$ 为

$$\langle f, g \rangle = \int_a^b w(x) f(x) g(x) dx$$

. 近似 p^* 可被这样计算：

1）构造一列 $(T_m), m < n$ 多项式使得 (T_m) 是 m 次的，而且 $\langle T_i, T_j \rangle = 0$ 对 $i \neq j$。这样的多项式叫做正交多项式。

2）计算系数：

$$a_i = \frac{\langle f, T_i \rangle}{\langle T_i, T_i \rangle}$$

3）计算

$$p^* = \sum_{i=0}^{n} a_i T_i$$

证明显而易见，可在大多数数值分析的书中找到 [42]。下面就来介绍其中几种正交多项式。文献 [43] 提供了更多正交多项式的信息。

2.3.2.1 勒让德多项式 [29]

权重函数：$w(x)=1$；区间 $[a,b]=[-1,1]$；定义：$T_0(x)=1, T_1(x)=x$，$T_n(x)=\frac{2n-1}{n}xT_{n-1}(x)-\frac{n-1}{n}T_{n-2}(x)$；标量积的值：

$$\langle T_i,T_j\rangle = 0 if i\neq j$$

$$\langle T_i,T_i\rangle = \frac{2}{2i+1}$$

2.3.2.2 Chebyshev 多项式 [29]

权重函数：$w(x)=1/\sqrt{1-x^2}$；区间 $[a,b]=[-1,1]$；定义：$T_0(x)=1, T_1(x)=x$，$T_n(x)=2T_{n-1}(x)-T_{n-2}(x)=\cos(n\cos^{-1}x)$；标量积的值：

$$\langle T_i,T_j\rangle = 0 if i\neq j$$

$$\langle T_i,T_i\rangle = \pi if i=0$$

$$\langle T_i,T_i\rangle = \pi/2 if i\neq 0$$

Chebyshev 多项式在逼近理论中占有重要地位。更详细的对 Chebyshev 多项式性质的展示可在文献 [44] 中找到。

2.3.2.3 Jacobi 多项式 [29]

权重函数：$w(x)=(1-x)^{\alpha}(1+x)^{\beta},(\alpha,\beta>1)$；区间 $[a,b]=[-1,1]$；定义：

$$T_n(x)=\frac{1}{2^n}\sum_{m=0}^{n}\binom{n+\alpha}{m}\binom{n+\beta}{n-m}(x-1)^{n-m}(x+1)^m$$

标量积的值：

$$\langle T_i,T_j\rangle = 0 if i\neq j$$

$$\langle T_i,T_i\rangle = h_i$$

$$h_i=\frac{2^{\alpha+\beta+1}}{2i+\alpha+\beta+1}\frac{\Gamma(i+\alpha+1)\Gamma(i+\beta+1)}{i!\Gamma(i+\alpha+\beta+1)}$$

2.3.2.4 Laguerre 多项式 [29]

权重函数：$w(x) = e^{-x}$；区间 $[a,b] = [0,+\infty]$；定义：

$$T_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}(x^n e^{-x})$$

标量积的值：

$$\langle T_i, T_j\rangle = 0 if i \neq j$$

$$\langle T_i, T_i\rangle = 1$$

2.3.2.5 在任意区间上使用上述多项式

除了 Laguerre 多项式以外（区间 $[a,b] = [0,+\infty]$），我们给出的正交多项式都在区间 [-1,1] 间。得到另一个区间 [a,b] 上的逼近是直截了当的：

1）对 $u \in [-1,1]$，定义

$$g(u) = f(\frac{b-a}{2}u + \frac{a+b}{2});$$

注意到 $x = ((b-a)/2)u + ((a+b)/2) \in [a,b]$;

2）计算 g 的在 $[-1,1]$ 上的最小二乘多项式逼近 q^*。

3）得到 f 的最小二乘多项式逼近 p^* 为

$$p^*(x) = q^*(\frac{2}{b-a}x - \frac{a+b}{b-a}).$$

2.3.3 最小最大值多项式逼近

和前面段落一样，我们希望用一个至多 n 次的多项式 p^* 在一个闭区间 $[a,b]$ 上逼近一个连续函数 f。假设权重函数 $w(x)$ 等于 1。以下，$\|f-p\|_{\infty,[a,b]}$ 表示如下距离：

$$\|f-p\|_{\infty,[a,b]} = \max_{a\leq x\leq b}|f(x)-p(x)|.$$

我们在寻找 p^* 满足 $\|f-p\|_{\infty,[a,b]}$ 在至多 n 次的多项式中最小。该多项式 p^* 存在且唯一。在 1885 年，Weierstrass 证明了一个连续函数可被多项式以任意需要的精度逼近。另一 Chebyshev 的定理给出了最小最大值多项式逼近函数的性质。

定理 2-2(Chebyshev) p^* 是 n 次 f 的最小最大值逼近当且仅当存在至少 $n+2$ 个值

$$a \leq x_0 < x_1 < x_2 < ... < x_{n+1} \leq b$$

使得

$$p^*(x_i) - f(x_i) = (-1)^i[p^*(x_0) - f(x_0)] = \pm\|f - p\|_\infty.$$

证明可见文献 [41]。

Chebyshev 的定理展示了如果 p^* 是 f 的 n 次最小最大值逼近，那么最大逼近偏差被至少 $n+2$ 次达到，并且偏差的符号交替达到。这种“等震荡”性质导致 p 可以在某些情况下直接找出。

2.3.4 最小二乘逼近

下面我们将举例说明：在 [-1,1] 上，用 2 次多项式逼近 e^x。

使用勒让德多项式的最小二乘逼近 [29]：

前三个勒让德多项式为

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$$

与勒让德近似相关的标量积为

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)dx.$$

我们容易得到

$$\langle e^x, T_0\rangle = e - 1/e$$

$$\langle e^x, T_1\rangle = e/2$$

$$\langle e^x, T_2\rangle = e - 7/e$$

$$\langle T_0, T_0\rangle = 2$$

$$\langle T_1, T_1 \rangle = 2/3$$

$$\langle T_2, T_2 \rangle = 2/5$$

系数易由此得出:$a_0 = (1/2)(e - 1/e), a_1 = 3/e, a_2 = (5/2)(e - 7/e)$，多项式 $p^* = a_0T_0 + a_1T_1 + a_2T_2$ 等于

$$\frac{15}{4}(e - 7/e)x^2 + \frac{3}{e}x + \frac{33}{4e} - \frac{3e}{4} \approx 0.5367x^2 + 1.1037x + 0.9963$$

使用 Chebyshev 多项式的最小二乘逼近 ：

前三个 Chebyshev 多项式为

$$T_0(x) = 1$$

$$T_1(x) = x$$

$$T_2(x) = 2x^2 - 1$$

与勒让德近似相关的标量积为

$$\langle f, g \rangle = \int_{-1}^{1} \frac{f(x)g(x)}{\sqrt{1 - x^2}} dx.$$

使用任何数值积分算法，得到

$$\langle e^x, T_0 \rangle = 3.977463261...$$

$$\langle e^x, T_1 \rangle = 1.775499689...$$

$$\langle e^x, T_2 \rangle = 0.426463882...$$

$$\langle T_0, T_0 \rangle = \pi$$

$$\langle T_1, T_1 \rangle = \pi/2$$

$$\langle T_2, T_2 \rangle = \pi/2$$

系数易由此得出:$a_0 = 1.266065878..., a_1 = 1.130318208..., a_2 = 0.2714953395...$，多项式 $p^* = a_0T_0 + a_1T_1 + a_2T_2$ 约等于

$$0.5429906776x^2 + 1.130318208x + 0.9945705392.$$

最小最大值逼近 [29]：假设 $p^*(x) = a_0 + a_1x + a_2x^2$ 是 e^x 在 $[-1, 1]$ 上的最小最大值逼近。由定理 2-2，至少存在四个值 x_0, x_1, x_2, x_3，在那些地方最大逼近误差以交替符号达到。指数函数的凸性意味着 $x_0 = -1, x_3 = +1$。而且，$e^x - p^*(x)$ 的导数对 x_1, x_2 等于 0。这样我们得到

$$a_0 - a_1 + a_2 - 1/e = \epsilon \tag{2.5}$$

$$a_0 + a_1x_1 + a_2x_1^2 - e^{x_1} = -\epsilon \tag{2.6}$$

$$a_0 + a_1x_2 + a_2x_2^2 - e^{x_2} = \epsilon \tag{2.7}$$

$$a_0 + a_1 + a_2 - e = -\epsilon \tag{2.8}$$

$$a_1 + 2a_2x_1 - e^{x_1} = 0 \tag{2.9}$$

$$a_1 + 2a_2x_2 - e^{x_2} = 0 \tag{2.10}$$

这非线性方程组的解是

$$a_0 = 0.98903973... \tag{2.11}$$

$$a_1 = 1.13018381... \tag{2.12}$$

$$a_2 = 0.55404091... \tag{2.13}$$

$$x_1 = -0.43695806... \tag{2.14}$$

$$x_2 = 0.56005776... \tag{2.15}$$

$$\epsilon = 0.04501739... \tag{2.16}$$

这样，e^x 的最好的最小最大二次多项式近似在 [-1,1] 上为 $0.98903973 + 1.13018381x + 0.55404091x^2$，最大近似误差约为 0.045.

以上例子中勒让德近似的最大误差为 0.081，Chebyshev 近似的最大误差为 0.050，最小最大近似的误差为 0.045。同时，以指数函数在 0 的二次泰勒展开近似：

$$e^x \approx 1 + x + \frac{x^2}{2}$$

的最大误差为 0.218。我们看到泰勒展开比其他近似差得多。这情况经常发生：泰勒展开只能给出本地的（也就是一个值附近的）近似，一般不应用于全球的（也就是在一个区间上）的近似。我们看到勒让德近似在平均意义下最好，最小最大近似在最坏情况下最好，Chebyshev 近似非常接近于最小最大近似。

然而，我们必须注意到，对一个足够不规则的函数，Chebyshev 近似没那么接近于最小最大近似。下面我们将举例说明：

对 $|x|$ 的二次多项式近似

在前面的例子中，我们试图用多项式接近一个非常规则的函数（指数函数）。我们发现即便是 2 次如此低的多项式，逼近结果很好，而不规则函数难以近似。我们来研究对 $|x|$ 在 [-1,1] 上的二次多项式逼近。

勒让德近似：

$$\frac{15}{16}x^2 + \frac{3}{16} = 0.9375x^2 + 0.1875$$

Chebyshev 近似：

$$\frac{8}{3\pi}x^2 + \frac{2}{3\pi} \approx 0.8488263x^2 + 0.21220659$$

最小最大近似：

$$x^2 + \frac{1}{8}$$

最大误差勒让德为 0.1875，Chebyshev 为 0.2122，最小最大为 0.125。在 $|x|$ 的情况，Chebyshev 近似和最大最小近似很有不同。

2.3.5 收敛速度

我们在前些段落中看到任何连续函数可被多项式以任意精度逼近。不幸的是，当函数不够规则时，为了一个逼近误差而需要的多项式次数可能很高。Bernstein 的一个定理 [45] 展示 n 次最小最大近似的收敛可能很慢。如果我们选择一个“收敛速度”（即选取一个递减正实数序列 $\epsilon_n \to 0$），存在一个连续函数 f 使得最小最大 n 次多项式的近似误差等于 ϵ_n。

表 2-3 展示了一些常用函数的多项式逼近的收敛速度。我们可以看到收敛速度看起来难以预测（有结果表明，收敛速度依赖于函数是否解析或全纯 [41]）

表 2.1: 各种函数的 2-8 次多项式最小最大近似的有效数位。

函数、次数	2	3	4	5	6	7	8	9
sin	7.8	12.7	16.1	21.6	25.5	31.3	35.7	41.9
exp	6.8	10.8	15.1	19.8	24.6	29.6	34.7	40.1
ln(1+x)	8.2	11.1	14.0	16.8	19.6	22.3	25.0	27.7
$(x+1)^x$	6.3	8.5	11.9	14.4	18.1	20.0	22.7	25.1
arctan	8.7	9.8	13.2	15.5	17.2	21.2	22.3	24.5
tan	4.8	6.9	8.9	10.9	12.9	14.9	16.9	19.0
sqrt	3.9	4.4	4.8	5.2	5.4	5.6	5.8	6.0
arcsin	3.4	4.0	4.4	4.7	4.9	5.1	5.3	5.5

2.4 基于查表的方法

2.4.1 介绍

在 2.1 节已解释过，为了计算一个大区域上的函数（可能是所有某格式下的浮点数），我们首先进行一步或多步的范围缩减 (range reduction)，来将我们的原问题缩减为在一个（或多个）小区间上计算该函数。在每个这些区间上，要么函数被一个系数存储在表中的多项式逼近，要么我们存储某些点上的函数值，并用等式例如 $\exp(a+b) = \exp(a)\exp(b)$ 或三角等式来将问题缩减为在 0 周围计算这些函数。选取这些区间不是一个简单地任务：

1）大的区间会使得范围缩减更简单，需求更小的表格。然而，它们需要很大度数的多项式，计算它们会耗费更多的时钟周期，而且保证紧凑的逼近误差上限更为困难；

2）小的区间会有只要求小度数的多项式从而快速计算的优势。然而，可能的缓存不命中（因它们需求的大表格）可能会完全摧毁性能。文献 [46] 讨论了这些问题。

2.4.2 表格 -驱动的算法

在文献 [47][48][49][50] 中，P.T.P.Tang 提出了使用查表算法实现基本函数的一些指引。为了计算 $f(x)$，他的算法使用三个基本步骤：

缩减：从输入变量 x（可能已经被预缩减），得到一个变量 y，y 属于一个很小的区域，使得 $f(x)$ 可容易地由 $f(y)$ 或一个其他函数 $g(y)$ 得到。此步可与预缩

减步合并；

逼近：$f(y)$ 或 $g(y)$ 由一个低阶多项式逼近；

重建：$f(x)$ 由 $f(y)$ 或 $g(y)$ 得出。

2.4.2.1 IEEE 浮点数下 exp(x) 的 Tang 算法

假设我们想要在 IEEE 双精度浮点数下计算 exp(x)。Tang[447] 建议先缩减自变量到一个值 r，它处在区间内

$$[-\frac{\ln(2)}{64}, +\frac{\ln(2)}{64}]$$

第二步用多项式 $p(r)$ 逼近 $\exp(r)-1$，最后重建 $\exp(x)$ 使用以下方程：

$$\exp(x) = 2^m(2^{j/32} + 2^{j/32}p(r)),$$

其中 j 和 m 为

$$x = (32m + j)\frac{ln(2)}{32} + r, 0 \leq j \leq 31.$$

这些步骤实现如下。

缩减 ：为了使计算更精确，Tang 用两个浮点数 r_1 和 r_2 之和表示 r 其中

$$r_2 \ll r_1$$

并且 $r_1 + r_2$ 以高于工作精度的精度近似于 r。为了做到这样，Tang 使用一种受 Cody 和 White 的参数缩减算法启发的方法：他定义三个浮点数 $L^{left}, L^{right}, \Lambda$，其中：

Λ 是 $32/\ln(2)$ 舍入到双精度；

L^{left} 含有一些尾部的 0；

$L^{right} \ll L^{left}$，并且 $L^{left} + L^{right}$ 以远高于工作精度的精度近似于 $\ln(2)/32$。

r_1, r_2 如下计算。计算 N 为 $x \times \Lambda$ 舍入至最近整数。定义 $N_2 = N \mod 32, N_1 = N - N_2$. 我们在工作精度下计算：

$$r_1 = x - N \times L^{left}$$

$$r_2 = -N \times L^{right}$$

上述 $m = N_1/32$ 和 $j = N_2$。这种缩减算法的分析可在文献 [186] 中找到。

逼近 ：$p(r)$ 如下计算。首先，我们在工作精度下计算 $r = r_1 + r_2$。其次，我们计算

$$Q = r \times r \times (a_1 + r \times (a_2 + r \times (a_3 + r \times (a_4 + r \times a_5))))$$

其中 a_i 是最小最大逼近的系数。最后，我们得到

$p(r) = r_1 + (r_2 + Q)$.

r_2 只在 1 次项中被使用。

重建 ：值 $s_j = 2^{j/32}$，$j = 0, ..., 31$ 被以更高精度预计算并且由两个双精度数 s_j^{left} 和 s_j^{right} 使得：

$s_j^{right} \ll s_j^{left}$;

s_j^{left} 的六个尾部位等于零;

$s_j = s_j^{left} + s_j^{right}$ 在约 100 位的精度。

用 S_j 表示 s_j 的双精度逼近。我们计算

$$exp(x) = 2^m \times (s_j^{left} + (s_j^{right} + S_j \times p(r))).$$

2.4.2.2 ln(x) 在 [1,2] 上

缩减 ：如果 $x - 1$ 非常小（Tang 建议 $e^{1/16}$ 的阈值），那么我们用多项式逼近 $\ln x$。否则，我们寻找“断点” $c_k = 1 + k/64, k = 1, 2, ..., 64$，使得

$$|x - c_k| \leq \frac{1}{128}.$$

我们定义 $r = 2(x - c_k)/(x + c_k)$。这样 $|r| \leq 1/128$。

逼近 ：我们使用多项式 $p(r)$ 逼近

$$\ln(\frac{x}{c_k}) = \ln(\frac{1 + r/2}{1 - r/2})$$

在 $r \in [0, 1/128]$ 上。根据所求的精度，可以使用以下多项式（具有 binary64 的系数）。

次数 3:$r + 6004845316577347 \times 2^{-56} r^3$，相对误差 8.805×10^{-12}。次数 5:$r + 6004799502898513 \times 2^{-56} r^3 + 900733669546681 \times 2^{-56} r^5$，相对误差 2.005×10^{-17}。次

数 7:$r+6004799503160663\times2^{-56}r^3+450359962663711\times2^{-55}r^5+5147094262912473\times2^{-61}r^7$，相对误差 8.036×10^{-23}。次数 9:$r+6004799503160661\times2^{-56}r^3+1801439851004373\times2^{-57}r^5+5146948095108811\times2^{-61}r^7+4731161337446337\times2^{-63}r^9$，相对误差 1.685×10^{-23}。

重建 ：我们从以下式子得到 $\ln(x)$：

$$\ln(x) = \ln(c_k) + \ln(x/c_k) \approx \ln(c_k) + p(r).$$

。

其中 $\ln(c_k)$ 查表得到。

如果我们想要更高的精度，我们可以使用更多的断点：256 个断点和一个 7 次多项式（binary64 系数）将使得逼近误差为 1.803×10^{-24}。使用 512 个断点，逼近误差减少为 4.503×10^{-25}，若使用 9 次多项式则为 2.632×10^{-25}。这些提升并不显著，因为规定系数为 binary64 数是一个显著的限制。若允许三次以上的系数为双词数则可得到显著更好的精度。

2.5 移位 -加法算法

移位 -加法算法允许用十分简单的基本操作计算基本函数：加法，与数字系统中的基数的次方相乘的乘法（在定点算术下，这样的乘法以简单的移位得到），以及与一个基数 -r 数位相乘的乘法。CORDIC 算法就属于这一类。

2.5.1 恢复和非恢复算法

算法 1：指数函数算法 1

输入：t,N(N 是步数)

输入：E_N

定义 $t_0 = 0, E_0 = 1$;

构建两个序列 t_n, E_n 如下

$$t_{n+1} = t_n + d_n \ln(1 + 2^{-n})$$

$$E_{n+1} = E_n(1 + d_n 2^{-n}) = E_n + d_n E_n 2^{-n}$$

$$d_n = 1 \ (if \ t_n + \ln(1 + 2^{-n}) \leq t), 0 \ (otherwise)$$

该算法只需要加法和与 2 的次方相乘的乘法。这样的乘法在基数 -2 算术下变为移位。在区间 $I \approx [0, 1.56]$ 内 E_n 看起来收敛到指数函数。我们很容易验证 $E_n = e^{t_n}$。这样，我们假定的结果是

$$\lim_{n \to +\infty} t_n = t$$

也就意味着

$$t = \sum_{i=0}^{\infty} d_n \ln(1 + 2^{-n})$$

这样，我们的算法看起来可以分解 I 上的任意 t 成为和

$$t = d_0 w_0 + d_1 w_1 + d_2 w_2 + ...,$$

其中 $w_i = \ln(1 + 2^{-i})$。现在我们试图描述序列 (w_i) 的特点来为类似的拆分铺路。我们已经知道一个这样的序列：$w_i = 2^{-i}$，那么 t 的二进制展开让我们能得到 d_i 对任意的 $t \in [0, 2)$。

下面的定理给出了计算这样的拆分的一种算法，只要序列 (w_i) 满足某些简单性质。

定理 2-3：可恢复拆分算法 令 (w_n) 为一个递减的正实数序列使得级数 $\sum_{i=0}^{\infty}$ 收敛。若

$$\forall n, w_n \leq \sum_{k=n+1}^{\infty} w_k$$

那么对任意 $t \in [0, \sum_{k=0}^{\infty} w_k]$，序列 (t_n) 和 (d_n) 定义为

$$t_0 = 0, t_{n+1} = t_n + d_n w_n$$

$$d_n = 1 if t_n + w_n \leq t, 0 otherwise$$

满足

$$t = \sum_{n=0}^{\infty} d_n w_n = \lim_{n \to \infty} t_n$$

以上算法叫做“可恢复算法”，因它类似于可恢复除法算法 [34]。就我们所知而言，一些除法算法和大多数移位 -加法算法之间的相似性首先在 Meggitt[51] 中

指出。另一些“伪除法”算法由 Sarkar 和 Krishnamuthy 提出 [52]。下面的定理介绍了另一种算法，它给出 d_i 等于 -1 或 +1 的拆分。这算法，由它与非恢复除法算法之间的相似性叫做“非恢复算法”，在 CORDIC 算法中被使用。

定理 2-4：非恢复拆分算法 令 (w_n) 为一个递减的正实数序列使得级数 $\sum_{i=0}^{\infty}$ 收敛。若

$$\forall n, w_n \leq \sum_{k=n+1}^{\infty} w_k$$

那么对任意 $t \in [-\sum_{k=0}^{\infty} w_k, \sum_{k=0}^{\infty} w_k]$，序列 (t_n) 和 (d_n) 定义为

$$t_0 = 0, t_{n+1} = t_n + d_n w_n$$

$$d_n = 1 if t_n \leq t, -1 otherwise$$

满足

$$t = \sum_{n=0}^{\infty} d_n w_n = \lim_{n\to\infty} t_n$$

2.5.2 指数和对数的简单算法

2.5.2.1 指数的可恢复算法

我们承认序列 $\ln(1+2^{-n})$ 和 $\arctan 2^{-n}$ 是离散基，也就是它们满足定理 2-3 的条件（证明见 [53]）。我们使用离散基 $w_n = \ln(1+2^{-n})$。令 $t \in [0, \sum_{k=0}^{\infty} w_k] \approx [0, 1.56202...]$。由定理 2-3，序列 (t_n) 和 (d_n) 定义为

$$t_0 = 0, t_{n+1} = t_n + d_n \ln(1+2^{-n})$$

$$d_n = 1 if t_n + \ln(1+2^{-n}) \leq t, 0 otherwise$$

满足

$$t = \sum_{n=0}^{\infty} d_n \ln(1+2^{-n}) = \lim_{n\to\infty} t_n$$

现在我们来建造一个序列 E_n 使得在算法的任何第 n 步，

$$E_n = \exp(t_n).$$

因 $t_0 = 0, E_0$ 必须等于 1。如果 $d_n = 1$，我们必须在第 n 步以 $\exp\ln(1+2^{-n}) = 1 + 2^{-n}$ 乘 E_n 才能保证上述关系不变。因 t_n 收敛于 t，E_n 收敛到 e^t。

误差分析 若我们在算法的第 n 步停止，我们由定理 2-3 的证明有 $0 \leq t - t_n \leq \sum_{k=n}^{\infty} \ln(1 + 2^{-k}) < \sum_{k=n}^{\infty} 2^{-k} = 2^{-n+1}$。这样我们有 $1 \leq \exp(t - t_n) < \exp(2^{-n+1})$

这样

$$|\frac{e^t - E_n}{e^t}| \leq 1 - e^{-2^{-n+1}} \leq 2^{-n+1}$$

如此，我们在第 n 步停止时，相对误差以 2^{-n+1} 为界（我们大致有 n-1 个有效位）。

2.5.2.2 对数的可恢复算法

假设我们想要计算 $l = \ln(x)$。首先，假定 l 已知，我们使用上面算法计算它的指数 x：$t_0 = 0, E_1 = 1, t_{n+1} = t_n + d_n \ln(1 + 2^{-n}), E_{n+1} = E_n + d_n E_n 2^{-n}$。

其中 $d_n = 1, if t_n + \ln(1 + 2^{-n}) \leq l, 0 otherwise.$ 然而，实际上 l 未知，我们利用 E_n 和 t_n 的关系变换此条件为 $d_n = 1, if E_n \times (1 + 2^{-n}) \leq x, 0 otherwise$。这样我们得到 E_n 收敛于 x 和 t_n 收敛于 l。

误差分析 与上算法类似地有

$$0 \leq l - t_n \leq \sum_{k=n}^{\infty} \ln(1 + 2^{-k}) \leq 2^{-n+1}$$

也就是我们在第 n 步停止时，绝对误差以 2^{-n+1} 为界。

2.6 本章小结

本章讨论了基本函数的各种算法，包括各种多项式近似，基于查表的方法，和我们在硬件上实现的移位 -加法算法。我们还讨论了浮点运算的基本概念和冗余数字系统，这些概念和 CORDIC 算法的发展历程是相关的。

第三章　CORDIC 算法

3.1　CORDIC 算法的发展历程

与 CORDIC 类似的数学方法早在 1624 年就被 Henry Briggs 提出过 [54]，但 CORDIC 是对低复杂度有限状态 CPU 的优化。

在 1956 年，Jack E. Volder 在 Convair 的航空电子学部门构想了 CORDIC 算法。当时 B-58 轰炸机的导航电脑使用的是模拟分解器，已显出不足，需要更精确实时的数字解法。在他的研究中，1946 年版的 CRC 物理与化学手册中的一个方程给了他启发：

$$K_n R\sin\theta \pm \varphi = R\sin\theta \pm 2^{-n} R\cos\theta \tag{3.1}$$

$$K_n R\cos\theta \pm \varphi = R\cos\theta \mp 2^{-n} R\sin\theta \tag{3.2}$$

其中 $K_n = \sqrt{1+2^{-2n}}, \tan\varphi = 2^{-n}$.

他的研究导致了一份内部技术报告，提出用 CORDIC 算法计算正弦和余弦函数，以及一个实现它的原型计算机。[55] 该报告同时讨论了用改进过的 CORDIC 算法计算双曲坐标旋转，对数和指数的可能性。用 CORDIC 算法进行乘法和除法的构想也在这时出现。根据 CORDIC 原理，Volder 在 Convair 的同事 Dan H. Daggett 研发了二进制与二进制编码的十进制之间的转换算法。[56]

在 1958 年，Convair 终于开始建造一个用于解决雷达固定问题的展示系统，叫做 CORDIC I，1960 年完成，并无 Volder 的参与（他已经离开此公司）。1962 年，Daggett 和 Harry Schuss 建造了更泛用的 CORDIC II 模型 A（固定）和 B（空运）。

1959 年，Volder 的 CORDIC 算法首次向公众公开，导致它很快被并入 Martin-Orlando, Computer Conrol, Litton, Kearfott, Lear-Siegler, Sperry, Raytheon, Collins Radio 等公司的导航计算机里。

Volder 与 Malcolm MacMillan 合作建造了一个定点桌面计算器 Athena，使用他的二进制 CORDIC 算法。该设计在 1965 年 6 月被推荐到惠普，但没有被接受。尽管这样，MacMillan 向 David S. Cochran 介绍了 Volder 的算法，当 Cochran 稍后见到 Volder 时向他介绍了类似的方法：John E. Meggitt 在 1961 构想的伪乘法和伪除法。[51] Meggitt 的方法建议使用十进制而不是二进制。这些努力在

1966 年导致了一个惠普十进制 CORDIC 原型的 ROMable 逻辑实现，它是基于 Thomas E. Osborne 的原型 Green Machine——一个四函数浮点桌面计算器，它在 1964 年由他从 DTL 逻辑完成。这个项目的结果是在 1968 年 3 月完成了第一个公开展示的带科学函数的桌面计算器，HP 9100A。[57]

John Stephen Walter 1971 年在惠普一般化了 CORDIC 算法，称为统一 CORDIC 算法，使得它能够计算双曲和指数函数，对数，乘法，除法和开方。三角函数和双曲函数的 CORDIC 子程序可以共享其大部分代码。1972 年诞生了第一个科学手持计算器，HP-35。[5]

最初，CORDIC 只用二进制来实现。即使 Meggitt 对他的伪乘法建议使用十进制系统，十进制 CORDIC 沉寂了数年之久。Hermann Schmid 和 Anthony Bogacki 在 1973 年仍将它当做一个新发明 [58]，后来才发现 1966 年惠普已经将其实现了。

十进制 CORDIC 成为了口袋计算器的广泛选择，大多数这类计算器采用二进制编码的十进制而不是二进制。这种输入输出格式的变化并没有改变 CORDIC 的核心算法。

CORDIC 在 Intel 8087, 80287, 80387, 80486 协处理器系列中被实现 [59]，也在 Motorola 68881 和 68882 中对某些浮点操作实现，主要是作为一种减少浮点计算单元的逻辑门数量（以及复杂性）的手段。

文献 [60] 对 CORDIC 算法的发展历程，架构和应用有更详细的阐述。

3.1.1 CORDIC 架构

本节展示一些将 CORDIC 算法映射到硬件上的体系结构。大体上，这些结构可以分为折叠式和展开式，如图 3.1 所示。折叠式架构由在硬件上重复 CORDIC 算法中每一个差分方程然后在单个功能单元上时间上复用所有迭代获得。在信号处理架构中，折叠提供了一种用时间换取面积的方法。这些折叠架构可以被分类为比特序列和单词序列，分类的标准是运算单元在一比特还是一单词上实现 CORDIC 算法每次迭代的逻辑。传统上 CORDIC 算法是用比特序列实现的，所有的迭代都在同一硬件上完成。这减慢了计算设备的速度，不适用于高速实现。单词序列架构是一种迭代式 CORDIC 架构，由实现迭代方程得到。在这种架构中，移位器在每次迭代中被修改得到该迭代中需要的移位数。合适的基本角度 α_i 通过查表来访问。在单词序列架构的迭代中最重要的速度因子是进位/借位加法/减法和变量移位操作，这使得传统 CORDIC 实现对高速应用来说显得速度不足。这些缺点可以由展开迭代过程来克服，使得每个处理元件总是执行同样的迭

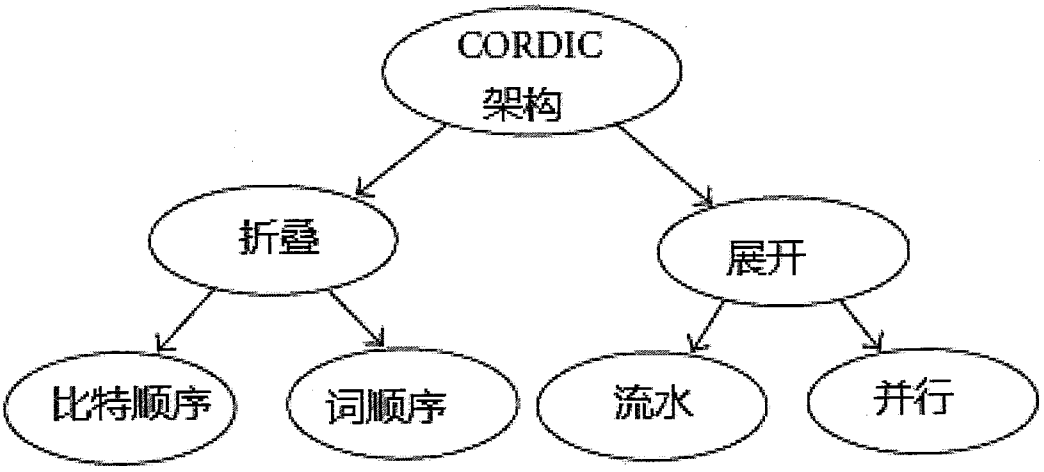

图 3.1: CORDIC 架构分类

代。展开式流水线架构相对于折叠式架构的优势是硬连线移位的高吞吐量而不是消耗时间和面积的桶移位器，以及 ROM 的消除。注意，流水线架构在 n 位精度时提供了 n 倍吞吐量的提高，代价是少于 n 倍地增加硬件。

3.1.2 CORDIC 变种的分类

CORDIC 算法的实现随着时代的变迁而发展，以适用于各种应用不同的要求——从非冗余到冗余。具有冗余算术的展开实现发起了解决传统 CORDIC 的高延迟的努力。此后，对冗余 CORDIC 的几个修改达到了减少迭代延迟、面积、功耗的效果。

CORDIC 根据数字系统的不同在大体上分为非冗余 CORDIC 和冗余 CORDIC。典型 CORDIC 算法的一大缺点是低吞吐量和高延迟，因在迭代方程的实现中需要进位传播加法器。这与 CORDIC 的新锐简单相违背，吸引了一些研究者开发加速实行的方法。解法一是减少每次迭代的时间，二是减少迭代次数，或者两者皆有。冗余算术已被用于减少典型 CORDIC 算法中每次迭代的时间。

文献 [61] 对 CORDIC 的变种进行了更详细的调查。

3.1.3 常数比例因子冗余基数 -2CORDIC

冗余基数 2CORDIC 方法可以基于比例因子对输入角度的依赖被分类为变量比例因子和常数比例因子方法。在冗余基数 -2CORDIC 中，$\sigma_i \in \{-1, 0, 1\}$ 因此

比例因子 K 是与数据相关的。这样，每次微旋转都需要计算 K，这些计算和纠正增加了计算时间和硬件开销。文献 [62][63] [64] 提供了几个常数比例因子的冗余 CORDIC 算法以解决比例因子数据相关的问题。在这些方法里，考虑了一个点基于 XY 平面中原点的迭代旋转。每次旋转的方向基于转向变量 z_i 的符号，它表示余下要旋转的角度。因为计算一个冗余数的符号需要更多时间，z_i 的估计值 $\hat{z}_i$ 被用来计算旋转的方向。该估计值由 z_i 的三个最显著位来计算。常数比例因子通过限制 σ_i 为 $\{-1,1\}$ 来达到，给更快的实现提供了条件。常数比例因子方法可以基于冗余基数 -2CORDIC 并有带符号位算术和进位借位算术中的算术来分类（见图 3.3）。

比例因子：在该段讨论的常数比例因子技术中不是所有实现都需要计算比例因子。在这些方法中，不考虑特别的比例因子补偿机制。注意到一个特别的补偿机制可以根据应用的不同而采用。

3.1.4 低延迟非冗余基数 - 2CORDIC[65]

文献 [66] 提出了一个对圆坐标系中的常规旋转 CORDIC 算法的显著改进，当剩余角度很小时使用线性近似。该剩余角度以如下方式选择：一阶泰勒级数 $\sin\theta_r \approx \theta_r$ 以及 $\cos\theta_r \approx 1$。用非冗余算术实现这种算法的架构在文献 [65] 中展示。前 $n/2+1$ 次迭代方程与典型 CORDIC 算法相同，前 $n/3$ 次迭代的 σ_i 值由角度累加器的 z_i 迭代决定。第 $n/3+1$ 以及之后迭代的旋转方向可以被并行决定，因为典型的圆模式反正切基数接近于 CORDIC 迭代次数的基数 -2 系数：

$$\lim_{k\to\infty} \frac{\tan 2^{-k}}{2^{-k}} = 1$$

在第 $(n/3+1) \le i \le (n/2+1)$ 次迭代中，所有的 σ_i 值由重编码的剩余角度 $w_{n/3+1}$ 决定。这些 σ_i 值被用于决定剩余角度 $w_{n/2+1}$。对 $i > (n/2+1)$，CORDIC 微旋转被替换为使用剩余角度 $w_{n/2+1}$ 的一次旋转：

$$x_{n/2+2} = k_{n/2+1}(x_{n/2+1} - \theta_r y_{n/2+1}),$$

$$y_{n/2+2} = k_{n/2+1}(y_{n/2+1} + \theta_r x_{n/2+1}),$$

其中 $\theta_r = wn/2+1$，$k_{n/2+1}$ 是第 $n/2+1$ 次迭代的比例因子。

3.1.5 使用 SD 算术的常比例因子冗余 CORDIC

使用 SD 算术的冗余基数 -2CORDIC 可基于达到常比例因子的技术被进一步分类（见图 3-5）。这些方法使用基本 CORDIC 迭代循环和必要的转换来实现。

修正旋转方法 [62] 用于计算正弦余弦函数时达到常比例因子的方法。这种方法避开 $\sigma_i = 0$ 的旋转，每次迭代基于估计值 $\hat{z}_i$ 进行一次旋转 -延长。更进一步地，在预定的空隙内有额外的旋转 -延长来纠正 $\sigma_i = 0$ 导致的错误并保证收敛。若用 b 分数位来估计 z_i，两次修正旋转之间的空隙应当等于或小于 $(b-2)$[67]。该方法在使用三到四个最显著位估计符号时也需要 50% 更多的迭代次数。这些旋转 CORDIC 中的双旋转和修正旋转带来的延迟可被枝节算法减少 [63]。

双旋转方法 [62] 双旋转方法在精度 n 下对前 $n/2$ 次迭代的每个基本角度进行两次旋转 - 延长。对大于 $n/2$ 次迭代的每个基本角度进行一次旋转 - 延长。一次负角度旋转通过两个负角度亚旋转获得。一次正角度旋转通过两个正角度亚旋转获得。一次不旋转通过一个负角度亚旋转和一个正角度亚旋转获得。这样，与典型的冗余 CORDIC 相比需要 50% 更多的迭代次数。

枝节方法 [63] 该方法用 SD 算术实现 CORDIC 算法，将旋转方向 σ_i 限制为 ± 1，并且不需要额外的旋转。这要求两个模组并行执行两个典型 CORDIC 旋转，每次旋转过后正确的结果被保留。如果 z_i 的符号可以被确定，两个模组在同样的方向执行旋转。若不然，枝节方法执行如下：一个 CORDIC 模组 (z^+) 在 $\sigma_i = +1$ 执行旋转，另一个模组 (z^-) 在 $\sigma_i = -1$ 执行旋转。下一次旋转的方向由哪一个模组中 z_i 的值小，就由该模组中 z_i 的符号来决定。在每次迭代中，角度累加器 $(z^+ \ or \ z^-)$ 计算剩余角度 $(z_i^+ \ or \ z_i^-)$ 来决定下次旋转的方向。旋转的方向由 $(z_i^+ \ or \ z_i^-)$ 的三个最显著位决定。

枝节方法的缺点是必须并行执行两个典型 CORDIC 迭代，在实现复杂性上几乎需要加倍的努力。另外，不执行枝节时其中一个模组就没有起到作用。然而，该方法比双旋转和修正旋转方法要快，因其在达到常比例因子时没有要求更多迭代。

两步枝节方法 [64] 枝节方法的性能可以通过两步枝节方法增加硬件利用率来提高。该方法涉及在每步中决定两个不同的 σ_i 值，比枝节方法多一些硬件，两个模组只在发生枝节的时候执行不同的计算。两步枝节方法在决定两个旋转方向时检查六个最显著位。这六个位被分为两个小组，每组三位，每组并行执行，使用归零模组生成要求的 σ_i。尽管两步枝节方法比枝节方法多一些硬件开销，它增加了 x/y 旋转模组的利用率。

3.1.6 使用 CS 算术的常比例因子冗余 CORDIC

有必要讨论另一类常比例因子冗余基数 -2CORDIC。相对于冗余基数 -2CORDIC，常比例因子冗余 CORDIC 使用符号算术的实现的结果是至少 50% 以上的面积和延迟增加 [62][63][64]。低延迟 CORDIC 算法 [68] 和微分 CORDIC 算法 [69][70] 这样使用 CS 算术的常比例因子冗余 CORDIC 已被提出，来解决这个开销，以下详述。

DCORDIC[69] 在 3.2.5 节的估计符号方法中，旋转 CORDIC 的 $x/y/z$ 数据路径有一半都花在修正可能出现的错误上，因为符号估计并不完美。该问题由高速比特流水技术和 CS 算术得到缓解 [70]。该算法涉及将典型 CORDIC 迭代方程转变为部分定格的迭代方程：

$$|\hat{z_{i+1}}| = ||\hat{z}_i| - \alpha_i|, \tag{3.3}$$

$$x_{i+1} = x_i - sign(z_i)2^{-i}y_i, \tag{3.4}$$

$$y_{i+1} = sign(z_i)2^{-i}x_i + y_i. \tag{3.5}$$

在这些表达式中显明的是计算 x 和 y 需要 z_i 的实际符号，但角度累加器 z 只需要 z_i 的绝对值。z_i 的实际符号可以由 z_0 的符号加上 $\hat{z}_i$ 绝对值计算中的符号改变信息决定。类似的，所有的 σ_i 值由递归算出。之后这种技术用 SD 算术实现并取名微分 CORDIC(DCORDIC) 被提出 [69]。因为在绝对值计算时计算转向变量 $\hat{z}_i$ 的符号要花长时间，采用了首个最显著位绝对值技术。这个技术替换了单词层面上的符号依赖，以比特层面上的依赖代替，减少了总计算时间。实现这些变换后的迭代序列的比特层面的流水线架构被提出，让高速操作成为可能。

低延迟冗余 CORDIC[68] 该算法用于解决冗余 CORDIC 中的延迟，提出将 n 次迭代分为不同的组，对不同的组使用不同的技术。对所有迭代，如果 $\sigma_i = \pm 1$，典型 CORDIC 迭代方程被使用。本方法在 $0 \leq i \leq (n-3)/4$ 次迭代时避免 $\sigma_i = 0$ 使用修正旋转方法。在 $(n-3)/4 < i \leq (n+1)/2$ 时 $\sigma_i = 0$ 成为可用的选择，因为在 n 位精度下有 $k_i = \sqrt{1+2^{-2i}} \approx 1 + 2^{-2i-1}$。向量不被旋转，但其长度以该比例因子 k_i 增加，因为最终坐标以常比例因子被缩放。在 $i > (n+1)/2$ 时，比例因子近似为 1，不需要对旋转进行修正。

3.2 CORDIC 算法简述

在本段里，我们讨论基于 CORDIC 算法计算的基本原理，并展示二维坐标下不同运算模式的迭代算法。在本段的最后，我们讨论二维旋转向高维的拓展。

3.2.1 典型 CORDIC 算法

由图 3.2 所示，二维向量 $p_0 = [x_0\ y_0]$ 基于原点角度 θ 的旋转得到的向量 $p_n = [x_n\ y_n]$ 有如下关系：

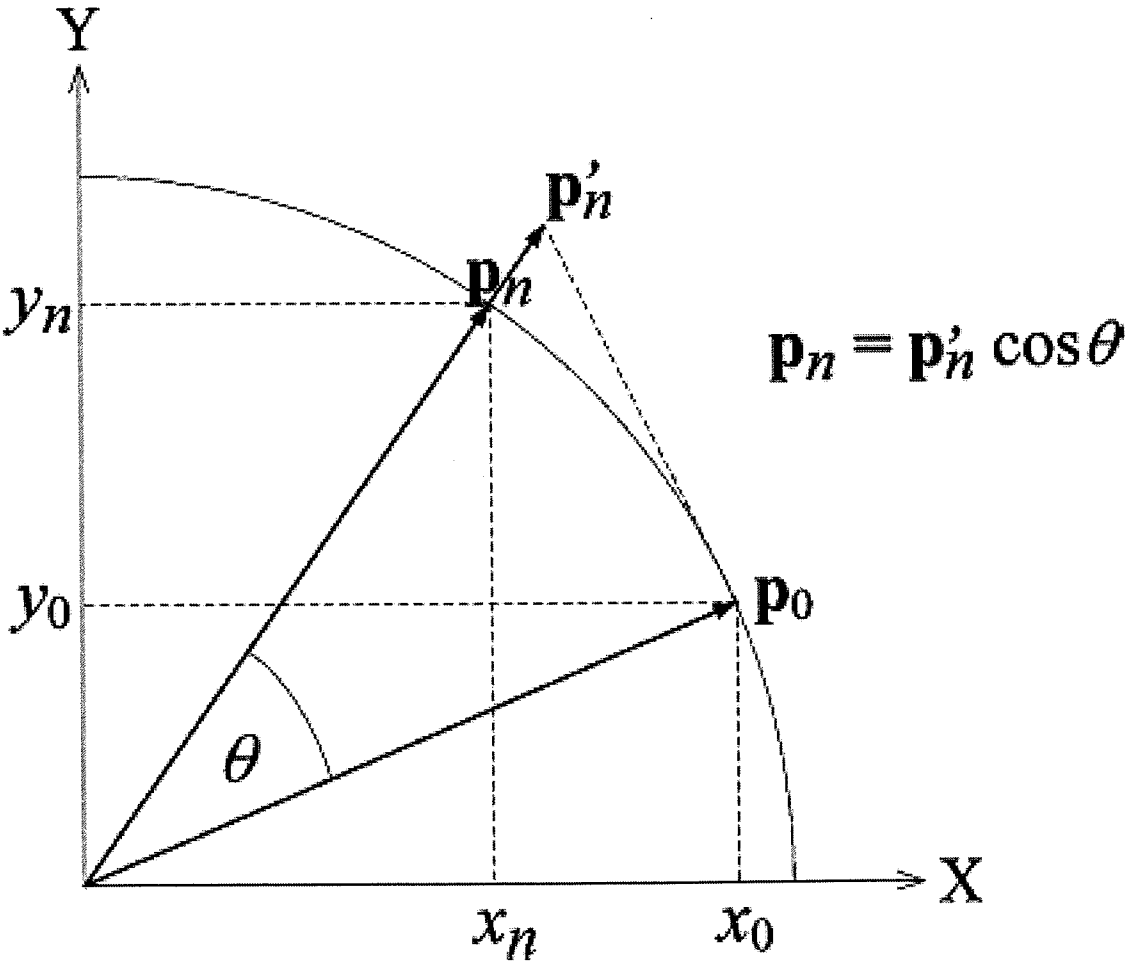

图 3.2: 在二维平面上做向量的旋转

$$x_n = \cos\theta x_0 - \sin\theta y_0$$

$$y_n = \sin\theta x_0 + \cos\theta y_0$$

或者写成矩阵形式：

$$p_n = Rp_0$$

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

为了计算方便，我们将 $\cos\theta$ 因子提出：

$$R = \cos\theta \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix}$$

这矩阵可以被解释为一个比例因子 $\cos\theta$ 和一个伪旋转矩阵 R_c 的乘积：

$$R_c = \begin{pmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{pmatrix}$$

这个伪旋转矩阵以 θ 角度旋转向量 p_0 并以因子 $K = (\cos\theta)^{-1}$ 改变其长度，得到一个伪旋转后向量 $p'_n = R_c p_0$.

为了实现硬件上的简便性，CORDIC 算法的重要思想有

1) 将整个旋转拆分为一系列预定好角度绝对值的基本旋转，这样的基本旋转在硬件上可被最小代价实现；

2) 避免比例因子的操作，这可能会涉及算术操作，如除法。第二个思想是基于比例因子只包含长度的信息但不包含旋转角度的信息。

3.2.1.1 旋转的迭代拆分

CORDIC 算法以迭代的方式完成旋转：旋转角度被拆分为一系列预定好的小角度，$\alpha_i = \arctan(2^{-i})$，这样，伪旋转中的 $\tan\alpha_i = 2^{-i}$ 就能在硬件上以移动 i 位来实现。CORDIC 算法不直接旋转角度 θ，而是通过一系列伪旋转 α_i，满足

$$\theta = \sum_{i=0}^{n-1} \sigma_i \alpha_i, \sigma_i = \pm 1$$

上述迭代收敛的条件为 $\alpha_i - \sum_{j=i+1}^{n-1} \alpha_j < \alpha_{n-1}, \forall i, i = 0, 1, ..., n-2$. 下面我们给出该条件为真的证明。

$$\because \arctan x = x - 1/3x^3 + 1/5x^5 - ... - 1/(4k-1)x^{4k-1} + 1/(4k+1)x^{4k+1} - ... \tag{3.6}$$

$$\therefore \alpha_i - \sum_{j=i+1}^{n-1} \alpha_j - \alpha_{n-1} \tag{3.7}$$

$$= \arctan 2^{-i} - \arctan 2^{-i-1} - ... - \arctan 2^{-n+1} - \arctan 2^{-n+1} \tag{3.8}$$

$$= 2^{-i} - 2^{-i-1} - ... - 2^{-n+1} - 2^{-n+1} \tag{3.9}$$

$$- ... \tag{3.10}$$

$$+ ... \tag{3.11}$$

$$- 1/(4k-1)(2^{(4k-1)-i} - 2^{(4k-1)-i-1} - ... - 2^{(4k-1)-n+1} - 2^{(4k-1)-n+1}) \tag{3.12}$$

$$+ 1/(4k+1)(2^{(4k+1)-i} - 2^{(4k+1)-i-1} - ... - 2^{(4k+1)-n+1} - 2^{(4k+1)-n+1}) \tag{3.13}$$

$$- ... \tag{3.14}$$

$$= 0 \tag{3.15}$$

$$- \frac{1}{3}\frac{2^{-3i} - 2 \times 2^{-3(n-1)} + 2^{-3n}}{1 - 2^{-3}} \tag{3.16}$$

$$+ \frac{1}{5}\frac{2^{-5i} - 2 \times 2^{-5(n-1)} + 2^{-5n}}{1 - 2^{-5}} \tag{3.17}$$

$$- ... \tag{3.18}$$

$$+ ... \tag{3.19}$$

$$- \frac{1}{4k-1}\frac{2^{-(4k-1)i} - 2 \times 2^{-(4k-1)(n-1)} + 2^{-(4k-1)n}}{1 - 2^{-(4k-1)}} \tag{3.20}$$

$$+ \frac{1}{4k+1}\frac{2^{-(4k+1)i} - 2 \times 2^{-(4k+1)(n-1)} + 2^{-(4k+1)n}}{1 - 2^{-(4k+1)}} \tag{3.21}$$

$$- ... \tag{3.22}$$

$$< 0 \tag{3.23}$$

但是，如此拆分只适用于 $|\theta| < 1.743286...$ 因为这是各微角度和 $\sum(\alpha_i)$ 的收敛范围。这个范围大于 $\pi/2$，因此对于第一和第四象限内的角度这种拆分适用。为了得到 θ 的拆分，σ 可以如下算出：

$$\sigma_j = 1, when\ \theta - \sum_{i=0}^{j-1} \sigma_i \alpha_i > 0,$$

$$\sigma_j = -1, when\ \theta - \sum_{i=0}^{j-1} \sigma_i \alpha_i < 0,$$

这样，对第 j 次旋转，旋转角度为 $\sigma_j\alpha_j$，旋转矩阵为

$$R = \cos\sigma_j\alpha_j \begin{pmatrix} 1 & -\tan\sigma_j\alpha_j \\ \tan\sigma_j\alpha_j & 1 \end{pmatrix}$$

比例因子 $\cos\sigma_j\alpha_j = 1/\sqrt{1+2^{-2i}}$，伪旋转矩阵

$$R_c = \begin{pmatrix} 1 & -\sigma_j 2^{-j} \\ \sigma_j 2^{-j} & 1 \end{pmatrix}$$

注意，伪旋转矩阵在第 j 次改变向量的长度 $1/\sqrt{1+2^{-2j}}$ 倍，与每次微旋转的方向 σ_j 无关。

3.2.1.2 移除比例因子

Volder 算法中的另一个简化是移除比例因子 $K_i = 1/\sqrt{1+2^{-2i}}$。移除后，我们得到的不是旋转后的向量 $p_n = KR_cp_0$ 而是伪旋转后的向量 $p'_n = R_cp_0 = \frac{1}{K}p_n$。比例因子 $A = \frac{1}{K}$ 为

$$A = \prod_{i=0}^{n}\frac{1}{K_i} = \prod_{i=0}^{n}\sqrt{1+2^{-2i}} \approx 1.6467605$$

这样，我们不再需要在每次迭代时都乘比例因子，而是将最终输出乘以 K，或者将输入乘以 $K = 1/A$。CORDIC 算法的基本迭代操作就是伪旋转向量 $p'_{i+1} = R_c(i)p_i$，同时将角度拆分为已选定角度 α_i，如下：

$$x_{i+1} = x_i - \sigma_i 2^{-i}y_i \tag{3.24}$$

$$y_{i+1} = y_i + \sigma_i 2^{-i}x_i \tag{3.25}$$

$$w_{i+1} = w_i - \sigma_i\alpha_i \tag{3.26}$$

上述迭代可以进行两种模式，旋转模式 (rotation mode) 和向量模式 (vector mode)。它们的不同在于如何选择旋转方向 σ_i。在旋转模式下，一个向量 p_0 被以角度 θ 旋转得到一个新的向量 p'_n。在这种模式下，每次微旋转的方向 σ_i 由 w_i 的符号决定：w_i 为正数时 σ_i 为 +1，否则 σ_i 为 -1。在向量模式下，向量 p_0 向着 x 轴旋转，其 y 坐标接近于零。所有微旋转的角度之和（输出角度 w_n）等于 p_0 被旋转的角度，输出的横坐标 x'_n 等于原向量的长度乘以比例因子。在这种模式下，每次微旋转的方向根据 y 的正负决定：y_i 为正时 $\sigma_i = -1$, 否则 $\sigma_i = 1$。CORDIC 的迭代无论在硬件上还是软件上都容易实现。在每次迭代之后移位的位数由一组移位器增加。为了达到 n 位精度，需要 $n+1$ 次 CORDIC 迭代。

3.2.2 CORDIC 算法的一般化

在 1971 年，Walther 发现 CORDIC 算法可以被改造并适用于双曲函数 [3]，他以一种一般化的方式重述了 CORDIC 算法，可以在圆、双曲、线性模式下实现旋转。这种一般化的表述包含了一个新变量 m，在不同的坐标系统下有不同的值。一般化 CORDIC 算法可以如下表述：

$$x_{i+1} = x_i - m\sigma_i 2^{-i} y_i \tag{3.27}$$

$$y_{i+1} = y_i + \sigma_i 2^{-i} x_i \tag{3.28}$$

$$w_{i+1} = w_i - \sigma_i \alpha_i \tag{3.29}$$

σ_i 的决定不变：旋转模式下 w_i 为正数时 σ_i 为 +1，否则 σ_i 为 -1。向量模式下 y_i 为正时 $\sigma_i = -1$, 否则 $\sigma_i = 1$。这样，每次迭代造成向量 $(x_n\ y_n)$ 的大小改变 $1/K_i = (1 + m\sigma_i^2)^{1/2}$ 倍，角度改变 $\alpha_i = m^{-1/2} \arctan m^{1/2}\sigma_i$。经过我们的调查，$m$ 取 1，0，-1 时可以得到三种不同的坐标模式：1- 圆，0- 线性，-1-双曲。m 取其他值也是可以的，但没有什么新的意义。

这样，迭代后的结果为

$$x_n = A(x_0 \cos(\alpha m^{1/2}) - y_0 m^{-1/2} \sin(\alpha m^{1/2})) \tag{3.30}$$

$$y_n = A(y_0 \cos(\alpha m^{1/2}) + x_0 m^{-1/2} \sin(\alpha m^{1/2})) \tag{3.31}$$

$$w_n = w_0 - \alpha \tag{3.32}$$

其中比例因子 $A = 1/K = \prod_{i=0}^{n} 1/K_i$。注意，

$$\alpha = \lim_{m \to 0} m^{-}1/2 \sin(\alpha m^{1/2}) \tag{3.33}$$

$$\alpha = \lim_{m \to 0} m^{-}1/2 \arctan(\alpha m^{1/2}) \tag{3.34}$$

$$\sinh z = -i \sin(iz), i = (-1)^{1/2} \tag{3.35}$$

$$\cosh z = \cos(iz) \tag{3.36}$$

$$\tanh^{-1} z = -i \tan^{-1}(iz) \tag{3.37}$$

这样，$m = 1$ 时进行的是经典 CORDIC 运算，旋转模式：

$$x_n = A(x_0 \cos w_0 - y_0 \sin w_0) \tag{3.38}$$

$$y_n = A(y_0 \cos w_0 + x_0 \sin w_0) \tag{3.39}$$

$$w_n = 0 \tag{3.40}$$

向量模式：

$$x_n = A\sqrt{x^2 + y^2} \tag{3.41}$$

$$y_n = 0 \tag{3.42}$$

$$w_n = w_0 + \tan^{-1}(y_0/x_0) \tag{3.43}$$

其中 $A = \prod_{i=0}^{n-1}(1 + 2^{-2i})^{1/2}$.

$m = 0$ 时将角度 α_i 取为 $2^{-i}, i = 1, 2, 3, 4, ...$ 则进行的是乘除法运算，旋转模式为乘法：

$$x_n = x_0 \tag{3.44}$$

$$y_n = y_0 + x_0 \times w_0 \tag{3.45}$$

$$w_n = 0 \tag{3.46}$$

向量模式为除法：

$$x_n = x_0 \tag{3.47}$$

$$y_n = 0 \tag{3.48}$$

$$w_n = w_n + y_0/x_0 \tag{3.49}$$

m=-1 时进行的是双曲函数运算：旋转模式：

$$x_n = B(x_0 \cosh w_0 + y_0 \sinh w_0) \tag{3.50}$$

$$y_n = B(y_0 \cosh w_0 + x_0 \sinh w_0) \tag{3.51}$$

$$w_n = 0 \tag{3.52}$$

向量模式：

$$x_n = B\sqrt{x^2 - y^2} \tag{3.53}$$

$$y_n = 0 \tag{3.54}$$

$$w_n = w_0 + \tanh^{-1}(y_0/x_0) \tag{3.55}$$

其中 $B = \prod_{i=1}^{n}(1 - 2^{-2i})^{1/2}$.

注意，双曲模式下 $i = 0$ 没有意义。而且将双曲模式下的角度 α_i 直接取为 $\tanh^{-1} 2^{-i}$ 不能满足 3.1.1.1 节的收敛条件 $\alpha_i - \sum_{j=i+1}^{n-1} \alpha_j < \alpha_{n-1}, \forall i, i = 0, 1, ..., n-2.$，需要重复第 $4, 13, 40, ..., k, 3k+1$ 次迭代才能满足。下面我们给出证明。

$$\because \tanh^{-1} x = x + 1/3x^3 + 1/5x^5 + ... + 1/(4k-1)x^{4k-1} + 1/(4k+1)x^{4k+1} + ... \tag{3.56}$$

$$\therefore \alpha_i - \sum_{j=i+1}^{n-1} \alpha_j - \alpha_{n-1} \tag{3.57}$$

$$= \tanh^{-1} 2^{-i} - \tanh^{-1} 2^{-i-1} - ... - \tanh^{-1} 2^{-n+1} - \tanh^{-1} 2^{-n+1} \tag{3.58}$$

$$= 2^{-i} - 2^{-i-1} - ... - 2^{-n+1} - 2^{-n+1} \tag{3.59}$$

$$+ ... \tag{3.60}$$

$$+ ... \tag{3.61}$$

$$+ 1/(4k-1)(2^{(4k-1)-i} - 2^{(4k-1)-i-1} - ... - 2^{(4k-1)-n+1} - 2^{(4k-1)-n+1}) \tag{3.62}$$

$$+ 1/(4k+1)(2^{(4k+1)-i} - 2^{(4k+1)-i-1} - ... - 2^{(4k+1)-n+1} - 2^{(4k+1)-n+1}) \tag{3.63}$$

$$+ ... \tag{3.64}$$

$$= 0 \tag{3.65}$$

$$+ \frac{1}{3} \frac{2^{-3i} - 2 \times 2^{-3(n-1)} + 2^{-3n}}{1 - 2^{-3}} \tag{3.66}$$

$$+ \frac{1}{5} \frac{2^{-5i} - 2 \times 2^{-5(n-1)} + 2^{-5n}}{1 - 2^{-5}} \tag{3.67}$$

$$+ ... \tag{3.68}$$

$$+ ... \tag{3.69}$$

$$+ \frac{1}{4k-1} \frac{2^{-(4k-1)i} - 2 \times 2^{-(4k-1)(n-1)} + 2^{-(4k-1)n}}{1 - 2^{-(4k-1)}} \tag{3.70}$$

$$+ \frac{1}{4k+1} \frac{2^{-(4k+1)i} - 2 \times 2^{-(4k+1)(n-1)} + 2^{-(4k+1)n}}{1 - 2^{-(4k+1)}} \tag{3.71}$$

$$+ ... \tag{3.72}$$

$$> 0 \tag{3.73}$$

因此直接取为 $\tanh^{-1}2^{-i}$ 不能满足 3.1.1.1 节的收敛条件。但观察上述余项可得

$$\frac{1}{3}\frac{2^{-3i}-2\times 2^{-3(n-1)}+2^{-3n}}{1-2^{-3}} \tag{3.74}$$

$$+\frac{1}{5}\frac{2^{-5i}-2\times 2^{-5(n-1)}+2^{-5n}}{1-2^{-5}} \tag{3.75}$$

$$+... \tag{3.76}$$

$$+... \tag{3.77}$$

$$+\frac{1}{4k-1}\frac{2^{-(4k-1)i}-2\times 2^{-(4k-1)(n-1)}+2^{-(4k-1)n}}{1-2^{-(4k-1)}} \tag{3.78}$$

$$+\frac{1}{4k+1}\frac{2^{-(4k+1)i}-2\times 2^{-(4k+1)(n-1)}+2^{-(4k+1)n}}{1-2^{-(4k+1)}} \tag{3.79}$$

$$+... \tag{3.80}$$

$$<\frac{2}{7}2^{-3i}+\frac{1}{5}2^{-5i}+\frac{1}{7}2^{-7i}+... \tag{3.81}$$

$$<\frac{2}{7}2^{-3i}+\frac{2}{7}2^{-5i}+\frac{2}{7}2^{-7i}+... \tag{3.82}$$

$$<\frac{2}{7}\times\frac{5}{4}2^{-3i} \tag{3.83}$$

$$<\frac{5}{14}2^{-3i} \tag{3.84}$$

$$<2^{-3i+1} \tag{3.85}$$

$$<\tanh^{-1}2^{-3i+1} \tag{3.86}$$

因此对任何一个 $i=1,2,3,4,....$，只要其后的第 $3i+1$ 次或更前的迭代被重复两次就能满足收敛条件。因此重复第 $4,13,40,...,k,3k+1$ 次迭代是保证收敛的充分条件。

但在我们的实验中，由于使用了 16 位浮点数，对高精确度 i 较大时有 $\tanh^{-1}2^{-i}\approx 2^{-i}$，因此不重复第 13 次迭代并不损失精度，我们的实验中只将第 4 次迭代重复。这样，比例因子 $B\approx 0.8281$。

3.2.3 高维 CORDIC 算法

文献 [71] 使用 Householder 镜像将 CORDIC 算法拓展到高维度。Householder 镜像矩阵定义如下：

$$H_m = I_m - 2\frac{uu^T}{u^Tu}$$

这里 $\mathbf{u}$ 是一个 m 维向量，I_m 是 $m\times m$ 单位矩阵。乘积 $(H_m v)$ 是将 m 维向量 $\mathbf{v}$ 作关于平面的镜像，该平面通过原点且法向量为 $\mathbf{u}$。基于 Householder 镜像的 CORDIC 算法将一个 m 维向量旋转到其中一个坐标轴上。

为了显明起见，我们考虑三维向量 $P_0 = [x_0\ y_0\ z_0]$ 变换至欧几里得空间的 x 轴上。三维情况下的旋转矩阵 $\mathbf{R}_{H3}(i)$ 是两个简单 Householder 镜像的乘积

$$\mathbf{R}_{H3}(i) = [\mathbf{I}_3 - 2\frac{e_1 e_1^T}{e_1^T e_1}][\mathbf{I}_3 - 2\frac{u_i u_i^T}{u_i^T u_i}]$$

这里 $e_1 = [1\ 0\ 0]^T$，$u_i = [1\ \sigma_{y_i} t_i\ \sigma_{z_i} t_i]^T$，$t_i$ 可取为 2^{-i}，旋转方向为 $\sigma_{y_i} = sign(x_i y_i)$ 和 $\sigma_{z_i} = sign(x_i z_i)$。

第 i 个旋转矩阵可以被写成伪旋转矩阵的形式 $\mathbf{R}_{H3}(i) = K_{Hi}\mathbf{R}_{HC3}(i)$，这里比例因子 $K_{Hi} = 1/\sqrt{1 + 2^{-2i+1}}$，$\mathbf{R}_{HC3}(i)$ 可以被表示成移位和抉择变量；

$$R_c = \begin{bmatrix} 1 & \sigma_{yi}2^{-i+1} & \sigma_{zi}2^{-i+1} \\ -\sigma_{yi}2^{-i+1} & 1 - 2^{-2i+1} & -\sigma_{yi}\sigma_{zi}2^{-2i+1} \\ -\sigma_{zi}2^{-i+1} & -\sigma_{yi}\sigma_{zi}2^{-2i+1} & 1 - 2^{-2i+1} \end{bmatrix}$$

这样，三维 CORDIC 旋转的第 i 次迭代得到 $\mathbf{p}_{i+1} = \mathbf{R}_{HC3}(i)\mathbf{p}_i$，向量被旋转到 x 轴，在 n 次迭代后长度放大了 $\prod_{i=0}^{n}(K_{Hi})$ 倍，精度为 $n-1$ 位 [65]。

3.3 并行 CORDIC 算法

到此为止讨论的 CORDIC 算法把角度 θ 表示成一系列叫做反正切基数集的基本角度

$$\theta = \sigma_0\alpha_0 + \sigma_1\alpha_1 + ... + \sigma_{n-1}\alpha_{n-1},$$

其中 $\alpha_i = \tan^{-1}(2^{-i})$ 以及 $\sigma_i \in \{-1, 1\}$，满足收敛定理

$$\alpha_i - \sum_{j=i+1}^{n-1} \alpha_j < \alpha_{n-1}$$

而不是用一般基数表示

$$\theta = \sigma_0 2^0 + \sigma_1 2^{-1} + ... + \sigma_{n-1}2^{-n+1}$$

第 i 次迭代的旋转方向 σ_i 在顺序计算完前 $i-1$ 次迭代后决定。从这基数系统的顺序依赖显而易见，CORDIC 算法的速度可被避免计算 σ_i 值或 x/y 坐标的顺序行为来提升。文献中提出的各种实施其中一种技术或两种都采用的冗余 CORDIC 算法在接下来的段落中讨论。

3.3.1 低延迟基数 -2CORDIC[68]

对旋转模式提出的低延迟并行基数 -2CORDIC 通过消除 z 路径的顺序依赖来预测 σ_i。为了最小化预测错误，每次只预测一组迭代的方向而不是所有迭代的方向。该架构不允许 $i=0$ 时旋转。这样，此架构的收敛域小于 $(-\pi/2,+\pi/2)$。另一方面，z 路径的中间结果需要从冗余表示转换到二进制表示，这限制了该架构的流水线实现。为了进一步降低该并行算法的延迟，停止算法和展位编码方法被提出。

3.3.2 P-CORDIC[72]

P-CORDIC 算法撤销了计算旋转方向时的顺序过程，并且维持了常比例因子。该算法在实际 CORDIC 旋转在 x/y 路径上迭代之前预先计算微旋转的方向。这由旋转方向 d 和角度 θ 的二进制表示之间的关系得到：

$$\sigma = 0.5\theta + 0.5c_1 + sign(\theta)\epsilon_0 + \delta$$

其中 $c_1 = 2 - \sum_{i=0}^{\infty}(2^{-i} - \tan^{-1}(2^{-i}))$，$\delta = \sum_{i=1}^{n/3}(\sigma_i\epsilon_i)$，$\epsilon_0 = 1 - \tan^{-1}(1)$，以及 $\epsilon_i = 2^{-i} - \tan^{-1}(2^{-i})$。这里，$\delta$ 在前 $n/3$ 次迭代时使用部分偏差 ϵ_i 和相应的方向比特 σ_i 来计算。在 $n/3$ 次以后，ϵ_i 的值每次缩小 8 倍。对任何二进制下的角度 θ，旋转的方向通过实现这个表达式从 ROM 中获取偏差变量 δ 获得。该算法的展开式实现去除了 z 路径，减少了实现的面积。该算法相对于基数 -2 展开式并行结构达到了延迟和硬件开销的减少。

比例因子：P-CORDIC 的该实现中比例因子维持常数，因为 $\sigma_i \in \{-1,1\}$ 在 x/y 路径的实现中得到保证。比例因子补偿通过常因子乘法技术来实现。

3.3.2.1 混合 CORDIC 算法

在圆坐标系下 n 位定点 CORDIC 处理器中，接近 $n/3$ 次迭代必须顺序进行。在精度不能受影响的情况下这对生成方向和旋转都为真。[73] 以后 $2n/3$ 次迭代的旋转方向可以并行生成，因为反正切系数逼近基数 -2 系数：

$$\lim_{k\to+\infty}\frac{\tan^{-1}2^{-k}}{2^{-k}} = 1.$$

这种行为可以被混合 CORDIC 算法利用，来加速典型 CORDIC 旋转器。该算法涉及拆分 θ 为 θ_H 和 θ_L。基于 θ_H 的旋转通过典型 CORDIC 算法来实施，涉及 θ_L 的迭代可以像线性坐标系中的被简化。该算法导致了几个并行 CORDIC 算

法的发展 [74][75] [76]，它们可以大体上被分类为糅合 -混合 CORDIC 和分离 -混合 CORDIC 算法。在糅合 -混合 CORDIC 算法中 [76]，输入角度 θ 和初始坐标 (x_{in}, y_{in}) 被用于计算前 $n/3$ 次迭代的旋转，与典型 CORDIC 相同。$n/3$ 次迭代之后余下的角度被用于计算后 $2n/3$ 次旋转的方向。该实现被设计得保持 x/y 路径的冗余算术的快时间特征。在分离 -混合 CORDIC 算法中 [74] [75]，θ 的前 $n/3$ 位用来生成前 $n/3$ 次旋转的方向，后 $2n/3$ 位用来推测后 $2n/3$ 次旋转的方向。

扁平 CORDIC[74] 该文献提出了扁平 CORDIC 算法来消去 x/y 路径的迭代特征以减少总计算时间。该算法通过连续替换将最终向量表示成初始向量的组合，将典型 CORDIC 的 x/y 迭代方程转化为一个并行版本，结果是一个 n 精度下单一方程。16 位精度下正弦/余弦生成器的最终坐标是：

$$
\begin{aligned}
x_{16} = [1 - \{(&\sigma_1\sigma_2 2^{-1}2^{-2} - ... - \sigma_1\sigma_{23}2^{-1}2^{-23} && (3.87)\\
&- \sigma_2\sigma_3 2^{-2}2^{-3} - ... - \sigma_9\sigma_{10}2^{-9}2^{-10}) && (3.88)\\
&+ (\sigma_1\sigma_2\sigma_3\sigma_4 2^{-1}2^{-2}2^{-3}2^{-4} + ... && (3.89)\\
&+ \sigma_2\sigma_3\sigma_4\sigma_5 2^{-2}2^{-3}2^{-4}2^{-5} + ... && (3.90)\\
&+ \sigma_3\sigma_4\sigma_6\sigma_7 2^{-3}2^{-4}2^{-6}2^{-7}) + E_{C-X}\}], && (3.91)\\
y_{16} = [&\sigma_1 2^{-1} + \sigma_2 2^{-2} + ... + \sigma_{16}2^{-16} && (3.92)\\
&- (\sigma_1\sigma_2\sigma_3 2^{-1}2^{-2}2^{-3} - ... - \sigma_5\sigma_7\sigma_8 2^{-5}2^{-7}2^{-8}) && (3.93)\\
&+ (\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5 2^{-1}2^{-2}2^{-3}2^{-4}2^{-5} + ... && (3.94)\\
&+ \sigma_2\sigma_3\sigma_4\sigma_5\sigma_6 2^{-2}2^{-3}2^{-4}2^{-5}2^{-6}) + E_{C-Y}], && (3.95)
\end{aligned}
$$

其中 E_{C-X} 和 E_{C-Y} 是错误补偿因子。x_{in} 和 y_{in} 被初始化为 $1/K$ 和 0。16 位精度下的 16 个符号位 $(\sigma_1, \sigma_2, ..., \sigma_{15}, \sigma_{16})$ 代表了要达到目标旋转的 16 个微旋转的方向。这些方程展示了典型 CORDIC 算法的完全并行化。该技术预先计算 σ_i，它在 $\{-1, 1\}$ 上取值以保证常比例因子。前 $n/3$ 次迭代的 σ_i 使用一项技术预计算，叫做分离分解算法 (Split Decomposition Algorithm, SDA)，它限制输入范围为 $(0, \pi/4)$。余下 $2n/3$ 个 σ_i 通过 $n/3$ 次迭代后余下的角度来预测。该架构和技术的内部单词长度被考虑为 $n + \log_2 n$，n 为外部精度。需要注意的是，x/y 迭代的完全并行化导致需要展平的项数呈指数级增长，影响了电路复杂性。另外，扁平 CORDIC 的实现需要复杂的组合电路硬件块，可扩展性差。

比例因子：扁平 CORDIC 算法中的比例因子是常数，因为 $\sigma_i \in \{-1, 1\}$。比例因子补偿通过一个使用 CS 加法树设计的乘法器来实现。

Para-CORDIC[75] Para-CORDIC 通过二进制至双极表示 (binary to bipolar representation, BBR) 和微旋转角度重编码 (microrotation angle recoding, MAR) 技术并行化旋转方向的生成。该算法迭代计算 x/y 坐标，并完全移除 z 路径。输入角度 θ 被拆分成为高位 θ_H 和低位 θ_L。二进制表示输入角度 θ 为它的两个构成部分是

$$\theta = (-d_0) + \sum_{i=1}^{l-1} d_i 2^{-i} + \sum_{i=l}^{n} d_i 2^{-i},$$

其中 $d_i \in \{0, 1\}$ 和 $l = (n - \log_2 3)/3$。$(l-1)$ 位的输入角度被转换为 BBR，而且旋转方向 σ_1 至 σ_{l-1} 通过 MAR 技术决定。因为 $\tan^{-1} 2^{-i} \neq 2^{-i}$，该方法对每次迭代根据每个二维位置权重 2^{-i} 进行额外的微旋转。在前 $(l-1)$ 次旋转之后剩余的角度被加至 θ_L。σ_1 至 σ_{n+1} 的值通过修正后的 θ_L 的 BBR 得到。该方法消除了储存预计算旋转方向的 ROM。然而，它需要更多的 x/y 阶段来重复某些微旋转，以及一组加法器来计算修正后的 θ_L。

半扁平 CORDIC[76] 典型 CORDIC 算法中的迭代特征可被半扁平 CORDIC 部分消除。该方法涉及为 $x/y/z$ 循环的半并行，来提升旋转展开式 CORDIC 的速度同时又不增加面积要求。内部精度比要求的外部精度高以避免 CORDIC 算法中的误差累积。对 σ_i 的最初 λ 位，x/y 循环使用双旋转方法 [62] 迭代计算，得到 $x_{\lambda-1}/y_{\lambda-1}$。接下来，如果所有的 σ_i 能被推测出来，$x_{n-1}/yn-1$ 可以被含 $x_{\lambda-1}/y_{\lambda-1}$ 的算式表达出。输入角度的接下来 $(n_{int}/3 - \lambda)$ 位的 σ_i 被预计算并放在 ROM 中。n_{int} 为内部精度。剩余的 $(2n_{int}/3)$ 个 σ_i 通过旋转角度来预测。值得注意的是，用于预计算 $(n_{int}/3 - \lambda)$ 个 σ_i 的分离分解算法没有给出描述或参考。

文献 [76] 中的模拟结果清晰表明，计算时间和芯片面积会被 λ 的选择所影响。这些测试的结果是，最好的平衡在 16 位 CORDIC （内部精度 22 位）下为 $\lambda = 6$，而在 32 位 CORDIC（内部精度 39 位）下为 $\lambda = 8$。在 λ 次迭代之后，所有 (x_n/y_n) 的项使用 Wallace 树相加，展平了 x/y 路径。然而，该架构的可扩展性差。

比例因子：该算法达到常比例因子，因 σ_i 在 $\{-1, 1\}$ 上取值。

3.4 高基数冗余 CORDIC

如 3.2 节所示，吞吐量和延迟是基于 CORDIC 系统的重要性能属性。到此为止摆上的各种基数 -2CORDIC 算法通过常比例因子可以被用于降低迭代延迟，导致增加吞吐量。在减低延迟方面，使用 SD 算术的 [77] 和 CS 算术的 [78] 高基数

CORDIC 算法被提出。这成为可能因高基数表示减少了迭代次数。基数 -4 旋转的实施最初是在文献 [67] 中构想的，用来加速基数 -2 算法。

在这段将要讨论的常比例因子算法中，比例因子不需要每步计算。这些方法不涉及具体的比例因子补偿机制，补偿机制可根据应用的不同而采用。

3.4.1 流水线基数 -4CORDIC[77]

文献 [77] 讨论了一般化的 CORDIC 算法：三种坐标系的任意基数及其在圆坐标系旋转模式下的基数 -4 流水线 CORDIC 处理器实现。这个算法进行先后两次基数 -2 微旋转，微旋转的角度相同，使用如下迭代公式

$$x_{i+1} = x_i - (\sigma_{i,1} + \sigma_{i,2})4^{-i}y_i - \sigma_{i,1}\sigma_{i,2}4^{-2i}x_i, \tag{3.96}$$

$$y_{i+1} = (\sigma_{i,1} + \sigma i, 2)4^{-i}x_i + y_i - \sigma_{i,1}\sigma_{i,2}4^{-2i}y_i, \tag{3.97}$$

$$z_{i+1} = z_i - (\sigma_{i,1} + \sigma_{i,2})\alpha_i, \tag{3.98}$$

这里 $\sigma_{i,1}$ 和 $\sigma_{i,2}$ 是两个不同的基数 -2 系数，用于解码基数 -4 系数 $\sigma_i \in \{-2,-1,0,+1,+2\}$ 满足关系 $(\sigma_i = \sigma_{i,1} + \sigma_{i,2})$。角度 α_i 选定为 $\alpha_0 = 2^{-1}$ 和 $\alpha_i = 4^{-i}$ 对 $1 \le i \le n-1$。选择 σ_i 的函数由 z-坐标的前五个最显著位决定，保证了该算法的收敛。该算法是用 SD 算术设计的，对 x/y 数据路径的 $i < n/4$ 的每个阶段需要两个加/减法器，相比之下基数 -2CORDIC 只需要一个。然而，所需加法数量在最后 $n/4$ 个阶段有所减少。

比例因子计算：在基数 -4CORDIC 算法中比例因子 K 是变量，因 σ_i 在 $\{-2,-1,0,+1,+2\}$ 上取值。每次迭代时 K 用组合电路计算，实现表达式

$$K = \prod_{i=0}^{n/2-1} k_i = \prod_{i=0}^{n/2-1} (1 + |\sigma_{i,1}|4^{-2i})^{1/2}(1 + |\sigma_{i,2}|4^{-2i})^{1/2}.$$

3.4.2 冗余基数 -2-4CORDIC[78]

在一个冗余基数 -2CORDIC 旋转单元中，旋转次数可通过将旋转方向表示为基数 2 和基数 4 的形式来减少 25%。该算法对迭代的不同子集采用不同的使用 CS 算术的 CORDIC 算法变种。对迭代 $1 \le i < n/4$，非冗余基数 -2CORDIC 算法被采用，其中 $\sigma_i \in \{-1,1\}$。对 $n/4 \le i \le (n/2+1)$，3.2.5 节的修正旋转方法被采用。对 $i > (n/2+1)$，冗余基数 -4CORDIC 算法被采用，这样所需迭代次数缩减一半。一个实现该算法在旋转/向量模式和圆/双曲坐标系下操作的统一架构在该文献中被提出。

比例因子计算：该算法有常比例因子，因为 $i < n/2+1$ 时 $\sigma_i = 0$ 被避免。$i > n/2+1$ 时比例因子为 $k_i = \sqrt{1+4^{-2i}}$，它太接近于 1 了，不需要被计算。

3.4.2.1 基数 -4CORDIC[79]

一个使用 CS 算术的冗余基数 -4CORDIC 被提出，用于减少冗余基数 -2CORDIC 的延迟。该算法使用两种不同技术计算 σ_i。对 $0 \leq i < (n/6)$，σ_i 顺序地由角度累加器决定。对 $i \geq (n/6)$ 次微旋转，σ_i 由前 $n/6$ 次旋转后余下的角度推测 [73]。这样，w 路径的复杂度是 $n/6$，而不是 3.3 节各种架构中的 n。对 $0 \leq i < (n/6)$，微旋转做成两步流水来提高吞吐量。文献中提出了一个 32 位流水结构来实现使用 CS 算术的基数 -4CORDIC 算法。

比例因子计算：可能出现的比例因子被预先计算并存放在 ROM 中。对 $\sigma_i^2 \in \{0,1,4\}$，可能的比例因子有 $3^{n/4+1}$ 个。随着 n 的增加，ROM 的大小和存取时间变长。因此，只对一部分迭代存储比例因子，这些值用来通过组合逻辑计算余下迭代的比例因子。这是通过实施比例因子的泰勒级数展开的前几项来设计的。在这基数 -4 实现中，迭代次数减少，代价是计算比例因子的硬件增加。

3.5 CORDIC 算法的缺陷和改进

典型 CORDIC 算法受到四种缺陷的掣肘。第一种缺陷为迭代方程导致的有限收敛域；第二种缺陷是计算的内在顺序性，(n+1) 次迭代才能达到 n 位精度，从而延迟变高；第三种缺陷是没有提供计算反正弦和反双曲正弦函数的方法。第四种缺陷为对浮点 CORDIC 算法，为了保证精度需有完全的数据宽度和更多的迭代，这意味着更多的资源消耗。

在本段中，我们首先阐述文献 [3] 提出的参数缩减算法来拓展收敛域；接下来我们拓展 CORDIC 算法来支持反正弦和反双曲正弦函数。最后，我们引入线性近似技术来减少数据宽度和迭代次数，又不损失精确度。用于减少延迟的流水线架构将在下一段中提出。

3.5.1 通过参数缩减算法拓展收敛域

典型 CORDIC 算法只有有限的收敛域，对双曲函数尤甚。解决这问题的有效方法主要有两种：引入一系列特定角度 [80] 以及通过数学等式重新映射角度，也叫做参数缩减算法。我们选择参数缩减算法来拓展收敛域，因其相对前一种方法精度较高。表 3-1 展示了对每个 CORDIC 的函数的参数缩减操作。

函数	收敛域	前处理	模式	输入 (x_0, y_0, z_0) 输出 (x_n, y_n, z_n)	后处理
$\sin a$	(-1.74,1.74)	$a = k\pi + b$ $b \in [-\pi/2, \pi/2]$	C-R	$(K_1^{-1}, 0, b)$ $(\cos b, \sin b, 0)$	$\sin a = y_n$ k 偶数 $\sin a = -y_n$ k 奇数
$\cos a$	(-1.74,1.74)	$a = k\pi + b$ $b \in [-\pi/2, \pi/2]$	C-R	$(K_1^{-1}, 0, b)$ $(\cos b, \sin b, 0)$	$\sin a = x_n$ k 偶数 $\sin a = -x_n$ k 奇数
$\arctan a$	$(-\infty, +\infty)$	无	C-V	$(1, a, 0)$ $(K_1\sqrt{1+a^2}, 0, \arctan a)$	$\arctan a = z_n$
$\sinh a$	(-1.13,1.13)	$a = k\ln 2 + b$ $b \in [0, \ln 2]$	H-R	$(K_{-1}^{-1}, 0, b)$ $(\cosh b, \sinh b, 0)$	$\sinh a = 2^{k-1}(x_n + y_n) - 2^{-k-1}(x_n - y_n)$
$\cosh a$	(-1.13,1.13)	$a = k\ln 2 + b$ $b \in [0, \ln 2]$	H-R	$(K_{-1}^{-1}, 0, b)$ $(\cosh b, \sinh b, 0)$	$\cosh a = 2^{k-1}(x_n + y_n) + 2^{-k-1}(x_n - y_n)$
$\exp(a)$	(-1.13,1.13)	$a = k\ln 2 + b$ $b \in [0, \ln 2]$	H-R	$(K_{-1}^{-1}, 0, b)$ $(\cosh b, \sinh b, 0)$	$\exp(a) = 2^k(x_n + y_n)$
$\tanh^{-1} a$	(-0.81,0.81)	$1 - \lvert a\rvert = b2^{-2k}$ $b \in [1/4, 1]$	H-V	$(1 + \lvert a\rvert + b, 1 + \lvert a\rvert - b, 0)$ $(2^{k+1}K_{-1}\sqrt{1-a^2}, 0, \tanh^{-1}\lvert a\rvert - k\ln 2)$	$a > 0$: $\tanh^{-1} a = z_n + k\ln 2$ $a \le 0$: $\tanh^{-1} a = -z_n - k\ln 2$
$\ln a$	(0.10,9.58)	$a = b2^{-2k}$ $b \in [1/4, 1]$	H-V	$(b+1, b-1, 0)$ $(2^{k+1}K_{-1}\sqrt{b}, 0, 1/2\ln b)$	$\ln a = 2k\ln 2 + 2z_n$
$\sqrt{a}$	(0.03,2.42)	$a = b2^{-2k}$ $b \in [1/4, 1]$	H-V	$(b+1, b-1, 0)$ $(2^{k+1}K_{-1}\sqrt{b}, 0, 1/2\ln b)$	$\sqrt{a} = K_{-1}^{-1}2^{-k-1}x_n$

表 3.1: CORDIC 算法在不同函数下使用参数缩减的情况

为了更清楚起见，我们假设以下符号：a 是输入角度，k 是补偿系数，b 是处于收敛范围内的目标角度。CORDIC 核心算法的输入向量是 (x_0, y_0, z_0)，输出向量是 (x_n, y_n, z_n)。我们同时引入核心单元的四种操作模式 C-R,C-V,H-R,H-V。C 和 H 各表示圆坐标模式和双曲坐标模式，R 和 V 各表示旋转模式和向量模式。

$\sin a$ 和 $\cos a$ 的计算 正弦和余弦使用 C-R 模式计算。输入角度 a 被映射到 $[-\pi/2, \pi/2]$ 上。给定 $a = k\pi + b$ 其中 k 是整数以及 $b \in [-\pi/2, \pi/2]$，k 和 b 可以如下计算：$k = \lfloor (a + \pi/2)/\pi \rfloor$ 和 $b = a - k\pi/2$。接下来我们设定输入向量 (x_0, y_0, z_0) 为 $(1/K_1, 0, b)$，经过核心单元的运算，输出向量为 $(\cos b, \sin b, 0)$。最终的结果 $\cos a$ 和 $\sin a$ 可分别由 $\cos b$ 和 $\sin b$ 根据 k 得出。

$\arctan a$ 的计算 反正切函数使用 C-V 模式计算。它不需要参数缩减操作。输入向量初始化为 $(1, a, 0)$，输出向量的 z_n 即反正切函数的最终结果。

$\sinh a, \cosh a, e^a$ 的计算 CORDIC 算法可以通过 H-R 模式得到这些函数。输入角度 a 应被映射到 $[0, \ln 2]$ 上。给定 $a = k \ln 2 + b$ 其中 k 是整数以及 $b \in [0, \ln 2]$，k

$\arcsin a$ 第一阶段	[-1,1]	$1 - \lvert a \rvert = b2^{-2k}$ $b \in [1/4, 1]$	H-V	$(1 + \lvert a \rvert + b, 1 + \lvert a \rvert - b, 0)$ $(2^{k+1}K_{-1}\sqrt{1-a^2}, 0, \tanh^{-1}\lvert a \rvert - k \ln 2)$	$\sqrt{1-a^2} = 2^{-k-1}K_{-1}^{-1}x_n$
$\arcsin a$ 第二阶段	$(-\infty, +\infty)$	无	C-V	$(\sqrt{1-a^2}, a, 0)$ $(K_1, 0, \arctan a/\sqrt{1-a^2})$	$\arcsin a = z_n$
$\sinh^{-1} a$ 第一阶段	$(-\infty, +\infty)$	无	C-V	$(1, a, 0)$ $(K_1\sqrt{1+a^2}, 0, \arctan a)$	$\sqrt{1+a^2} = K_1^{-1}x_n$
$\sinh^{-1} a$ 第二阶段	$(-\infty, +\infty)$	$a + \sqrt{1-a^2} = b2^{-2k}$ $b \in [1/4, 1]$	H-V	$(b + 1, b - 1, 0$ $(2^{k+1}K_{-1}\sqrt{b}, 0, 1/2 \ln b)$	$\sinh^{-1} a = 2k \ln 2 + 2z_n$

表 3.2: 续表：CORDIC 算法在不同函数下使用参数缩减的情况

和 b 可以如下计算：$k = \lfloor a/\ln 2 \rfloor$ 和 $b = a - k\ln 2$。输入向量初始化为 $(K_{-1}^{-1}, 0, b)$，经过核心单元的运算，输出向量为 $(\cosh b, \sinh b, 0)$。最终结果通过表 5-1 中的补偿系数由 $\cosh b$ 和 $\sinh b$ 得出。

$\ln a, \sqrt{a}, \tanh^{-1} a$ 的计算 这三个函数都使用 H-V 模式计算。前处理的核心操作可以被表示为 $a = b4^k$ 其中 $b \in [1/4, 1]$。在硬件实现上，这操作简化为寻找 b 的第一个零的位数。

3.5.2 用于反正弦和反双曲正弦的改良 CORDIC 算法

典型 CORDIC 算法并不支持反正弦和反双曲正弦的计算。在本段中，我们将拓展 CORDIC 算法来支持上述两个函数：使用如下数学公式

$$\arcsin a = \arctan(a/\sqrt{1-a^2})$$

$$\sinh^{-1} a = \ln(a + \sqrt{1+a^2})$$

$\arcsin a = \arctan(a/\sqrt{1-a^2})$ 的计算 反正弦函数可通过两个阶段来完成。第一阶段，我们设定 CORDIC 算法为 H-V 模式，输入向量 $(x_0, y_0, z_0) = (1, a, 0)$ 来得到输出向量 $x_n = \sqrt{1-a^2}$。在第二阶段，我们在 C-V 模式下旋转，设定输入向量为 $(x_0, y_0, z_0) = (\sqrt{1-a^2}, a, 0)$，最后结果为 $z_n = \arctan(a/\sqrt{1-a^2})$。

$\sinh^{-1} a = \ln(a + \sqrt{1+a^2})$ 的计算 反双曲正弦函数也可以通过两个阶段来完成。在第一阶段中，我们设定 CORDIC 算法为 C-V 模式，输入向量 $(x_0, y_0, z_0) = (1, a, 0)$ 来得到输出向量 $x_n = \sqrt{1+a^2}$。在第二阶段，我们在 H-V 模式下旋转，设定输入向量为 $(x_0, y_0, z_0) = (\sqrt{1+a^2}+a+1, \sqrt{1+a^2}+a-1, 0)$，最后结果为 $2z_n = \ln(a + \sqrt{1+a^2})$。

3.5.3 使用线性近似减少数据宽度和迭代次数

CORDIC 算法在本质上是串行的，为了达到 n 位精度需要 $n+1$ 次迭代。如果我们使用定点算术，为了达到 n 位精度需要 x, y 的数据宽度各为 $n+2+\log_2 n$，z 的输入宽度为 $n + \log_2 n$，然而在浮点数算术中，需要更多的数据宽度和迭代次数来达到期望的精度。

2.1 节讨论 IEEE754 时已提到，对半精度浮点表示，最小的严格正数值为 2^{-24}，意味着需要 25 次迭代，32 位 x 和 y 数据宽度，30 位 z 数据宽度。这会导致长延迟和大面积硬件开销。

为了解决这个问题，我们提出线性近似来减少数据宽度和迭代次数又不损失精度。简要介绍这种技术：以圆坐标系下的旋转模式为例（它输出正弦和余弦函数），正弦和余弦在 0 附近的泰勒展开分别为：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots$$

对于一个 n 精度的角度 θ 满足 $\theta < 2^{-n/2}$，第二项以后的泰勒级数在 n 精度下没有作用，最终结果就是 $\sin(\theta) = \theta$ 和 $\cos(\theta) = 1$。同样，在 0 周围有 $\arctan(\theta), \arcsin(\theta)$, $\sinh(\theta), \tanh^{-1}(\theta), \sinh^{-1}(\theta) \approx \theta$，$\exp(\theta) \approx 1 + \theta$，$\cos(\theta), \cosh(\theta) \approx 1$。余下的自然对数和开方函数在参数缩减之后有 $\theta \in [1/4, 1]$，线性近似不需要。

对于半精度浮点算术，尾数部分为 10 位也就是需要 11 位精度。所以当 $\theta < 2^{-5}$ 时可以直接得到结果不需要进行 CORDIC 运算。相比之下，当 $\theta > 2^{-5}$ 时前 5 位不都是 0，意味着尾数部分需要额外四个位。结论是为了保持浮点数准确，只需要 15 位精度，也就是 16 次迭代，21 位 x,y 数据宽度和 19 位 z 数据宽度。

为了防止溢出，我们对 x,y,z 数据路径添加三位。最终的数据宽度为 x,y24 位，z22 位。

第四章　半精度浮点 CORDIC 处理器

4.1　设计理念

我们从基本 CORDIC 运算的实现需要一定的迭代次数发现，采用流水线技术可以大幅度增加基本 CORDIC 单元的吞吐量。然而，基本 CORDIC 单元的输入格式是直角坐标系下的长度和角度，并且其角度必须在一定的收敛范围内（圆模式和双曲模式拥有不同的收敛范围）才能正常计算。再加上我们需要浮点数的输入和输出，因此需要前处理器和后处理器。基本 CORDIC 单元的流水线技术倒逼我们在前处理器和后处理器上也实施流水线技术，来提高吞吐量。不幸的是，不同函数需要的前处理和后处理操作不同，为了避免冲突，我们暂时取最长的流水路径，代价是增加部分函数的计算延迟。

4.2　硬件实现和实验

本文提出了的半精度浮点数 CORDIC 架构的设计和实现，它使用前段阐述过的 IEEE 754 半精度浮点表示。

大体上，CORDIC 处理器由三个模块构成：预处理器，参数微旋转处理器和后处理器。在本段中将详述每个模块的结构。

4.2.1　总体 RTL 结构

输入输出为 clock, reset, in_a, in_f, in_valid, out_result, out_valid。

其中本地参数为内部数据宽度 data_width（设定为 20）

其中输入为 clock（系统时钟）一比特，reset（系统重置）一比特，in_a（原始数据）16 比特，in_f（函数种类）4 比特，in_valid （in_a 是否有效）一比特。

其中输出为 out_result（浮点数输出）16 比特，out_valid（out_result 是否有效）1 比特。

内部 wire 和 reg 定义如下：

连接预处理和 CORDIC 流水线之间的 wire：x1，y1，z1 各 data_width 比特；mode1，3 比特；k1，6 比特；f1，4 比特；valid1，1 比特；sign1，1 比特；

连接 CORDIC 流水线和后处理器之间的 wire：x2，y2，z2 各 data_width 比特；mode2，3 比特；k2，6 比特；f2，4 比特；valid2，1 比特；sign2，1 比特；

内部包含预处理模块：clock 为 clock，reset 为 reset，in_a 为 in_a，in_f 为 in_f，in_valid 为 in_valid，out_x 为 x1，out_y 为 y1，out_z 为 z1，out_mode 为 mode1，out_k 为 k1，out_f 为 f1，out_valid 为 valid1，out_sign 为 sign1。

内部包含 CORDIC 流水线模块：clock 为 clock，reset 为 reset，in_x 为 x1，in_y 为 y1，in_z 为 z1，in_mode 为 mode1，in_k 为 k1，in_f 为 f1，in_vaild 为 valid1，in_sign 为 sign1，out_x 为 x2，out_y 为 y2，out_z 为 z2，out_mode 为 mode2，out_k 为 k2，out_f 为 f2，out_vaild 为 valid2，out_sign 为 sign2。

内部包含后处理模块：clock 为 clock，reset 为 reset，in_x 为 x2，in_y 为 y2，in_z 为 z2，in_mode 为 mode2，in_k 为 k2，in_f 为 f2，in_valid 为 valid2，in_sign 为 sign2，out_result 为 out_result，out_valid 为 out_valid。

图 4.1 表示了 CORDIC 处理器在 RTL 方面的结构。

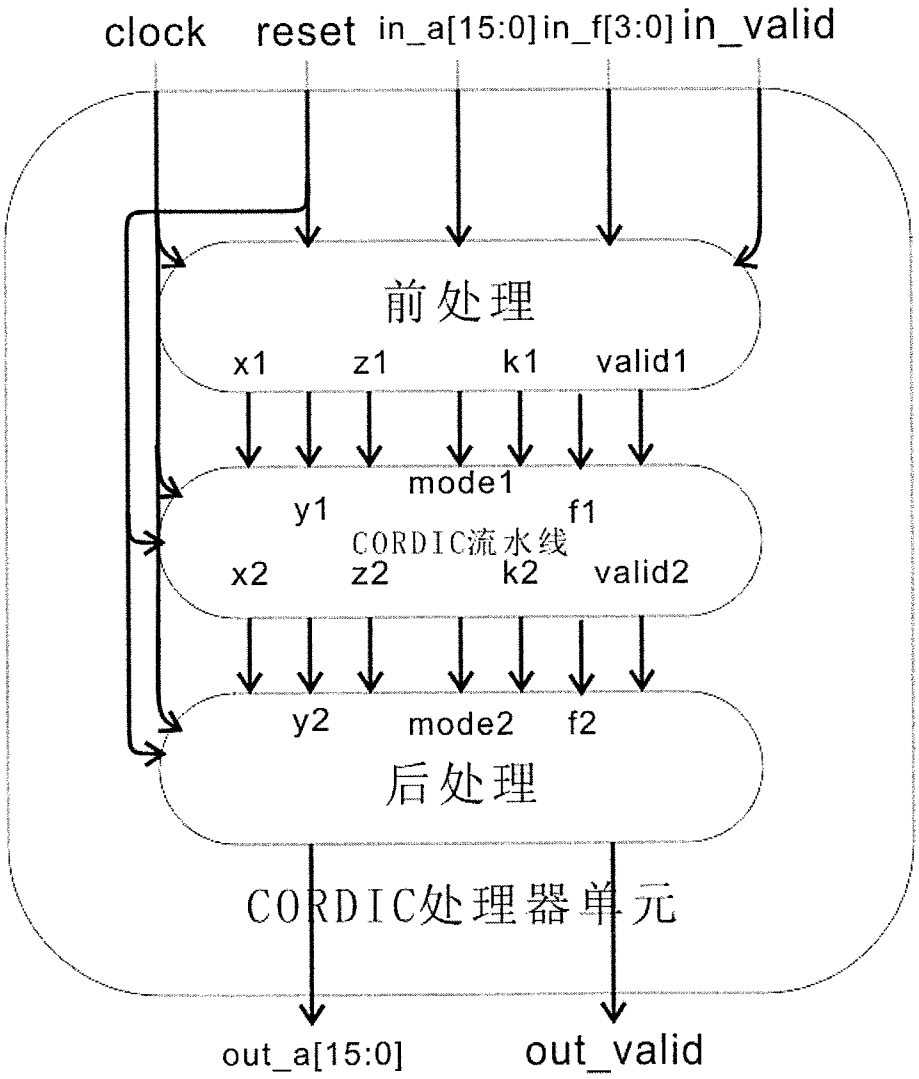

图 4.1: CORDIC 处理器的 RTL 结构

4.2.2 预处理器概览

预处理器负责将半精度浮点格式转换为定点格式并且映射角度到 CORDIC 算法的收敛域内。它包括 9 个流水线阶段。我们以记号 Sn 来表示第 n 个流水阶段。

S1 根据 IEEE 754 标准转换半精度浮点数到定点数。参数缩减技术在 S2 至 S6 中进行，来扩展收敛域。最重要的路径为对正弦和余弦的角度重映射：S2,S3,S4 进行与 $1/\pi$ 的乘法来得到 k。然后 S5,S6,S7 完成与 π 的乘法得到 $k\pi$，之后 S8 进行减法得到 b。S9 进行小角度时的线性近似，来减少迭代次数和数据宽度。

4.2.3 预处理器的 RTL 结构

预处理器的输入输出为 clock, reset, in_a, in_f, in_valid, out_x, out_y, out_z, out_mode, out_k, out_f, out_valid, out_sign。

其中本地参数为内部数据宽度 data_width（设定为 20）

其中输入为 clock（系统时钟）一比特，reset（系统重置）一比特，in_a（需要做预处理的原始数据）16 比特，in_f（函数种类）4 比特，in_valid（in_a 是否有效）一比特。

其中函数种类如下规定：

sin-0000，cos-0001，tan-0010，arctan-0011，sinh-1000，cosh-1001，tanh-1010，exp-1011，log-1101，sqrt-1110，arctanh-1111。

其中输出为 out_x, out_y, out_z（为 CORDIC 核心模块需要的 x,y,z）均为 data_width 比特，out_mode（CORDIC 核心模块的模式选择）3 比特，out_k（供后处理模块使用）6 比特，out_f（函数种类）4 比特，out_valid（对后处理模块的输出有效）一比特，out_sign （对后处理模块的输出符号）一比特。

其中 CORDIC 核心模块的模式选择如下规定：

000 圆 + 旋转模式 001 圆 + 向量模式 010 双曲 + 旋转模式 011 双曲 + 向量模式 100 输入在前处理模块为小值 out_z 为浮点 111 输入溢出范围

模块中 wire 和 reg 定义如下：

wire 常量：math_inv_pi（$1/\pi$）18 比特，math_pi（π）24 比特，inv_A（圆模式下的比例因子的倒数）20 比特，math_inv_ln2 （$1/\ln 2$）12 比特，math_ln2（$\ln 2$）21 比特，inv_B（双曲模式下的比例因子的倒数）20 比特。

输入的表示：sign 一比特为输入的符号 in_a[15]，exp5 比特为输入的指数部分 in_a[14:10]，res10 比特为输入的尾数部分 in_a[9:0]。

计算的中间结果：

第一阶段：reg1_stage1，29 比特；valid_stage1，1 比特；f_stage1，4 比特；sign_stage1，1 比特；exp_stage1，5 比特；res_stage1，10 比特；

第二阶段：wire1_stage2，29 比特；wire2_stage2，29 比特；reg1_stage2，16 比特；valid_stage2，1 比特；f_stage2，4 比特；sign_stage2，1 比特；exp_stage2，5 比特；res_stage2，10 比特；

第三阶段：wire1_stage3，25 比特；wire2_stage3，25 比特；wire3_stage3，16 比特；wire4_stage3，25 比特；wire5_stage3，25 比特；wire6_stage3，11 比特；reg1_stage3，20 比特；reg2_stage3，20 比特；reg3_stage3，6 比特带符号；valid_stage3，1 比特；f_stage3，4 比特；sign_stage3，1 比特；exp_stage3，5 比特；res_stage3，10 比特；

第四阶段：wire1_stage4，20 比特；wire2_stage4，20 比特。

4.2.4 带参数微旋转处理器概览

参数微旋转处理器执行 CORDIC 旋转来完成所需函数的计算。它包括三个阶段，分别叫做第一旋转阶段，暂存层阶段和第二旋转阶段。参数为基本 CORDIC 单元数量 M 和每个单元执行的迭代次数 N。M 和 N 满足 MN=32。总的流水线阶段为 M+5。该参数允许我们在面积，时钟频率和执行速度上找到平衡（将在后面段落中讨论）。

第一旋转阶段包括 M/2 流水线阶段，每个阶段执行 N 个 CORDIC 迭代，总迭代次数为 MN/2=16。每个流水线阶段是图 2 所示的基本 CORDIC 单元，拥有 3 个定点加法器，2 个移位器和一个包括 N 个迭代角度的查表 (Look-Up Table, LUT)。在此阶段之后，大多数 CORDIC 函数就已经计算完毕，但反正弦函数和反双曲正弦函数除外，它们需要两步来得到最终结果。

暂存层特别设计用于为反正弦函数和反双曲正弦函数完成 $\sqrt{1-a^2}$ 和 $a+\sqrt{1+a^2}$ 的计算，拥有 6 个流水线阶段，详细操作已在表 3.2 中给出。最重要路径为 $a+\sqrt{1+a^2}$ 的计算。前三步完成与 K_{-1}^{-1} 的乘法来得到 $\sqrt{1+a^2}$，第四步计算 $a+\sqrt{1+a^2}$ 进行加法运算。第五步计算最初出现的 0 来得到 k 和 b。第六步进行 +1 和 -1 来得到第二步的输入。

第二旋转阶段用于计算反正弦函数和反双曲正弦函数，与第一旋转阶段相同。

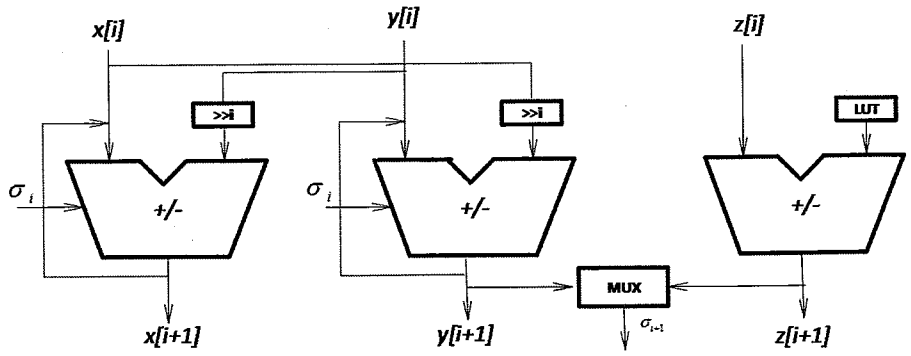

图 4.2: 微旋转处理器

4.2.5 带参数微旋转处理器的流水线 RTL 结构

CORDIC 流水线的参数为 data_width（内部数据长度）取为 20。

CORDIC 流水线的输入为 clock（系统时钟）1 比特，reset（系统重置）1 比特，in_x（输入 x）带符号 data_width 比特，in_y（输入 y）带符号 data_width 比特，in_z（输入 z）带符号 data_width 比特，in_mode（操作模式）3 比特，in_k（预处理信息）6 比特，in_f （预处理传来的函数种类）4 比特，in_valid（输入是否有效）1 比特，in_sign(预处理信息）1 比特。

CORDIC 流水线的输出为 out_x（输入 x）带符号 data_width 比特，out_y（输入 y）带符号 data_width 比特，out_z（输入 z）带符号 data_width 比特，out_mode（操作模式）3 比特，out_k（预处理信息）6 比特，out_f （预处理传来的函数种类）4 比特，out_valid （输入是否有效）1 比特，out_sign(预处理信息）1 比特。

CORDIC 流水线内部的 reg 和 wire 定义：

本地参数 stages（取为 16）；wire x 为长度 stage+1 的数组，每个成员为带符号 data_width 比特。wire y，wire z 同样。mode，k，f，valid，sign 也是长度 stage+1 的数组，每个成员分别为 3，6，4，1，1 比特。

x，y，z，mode，k，f，valid，sign 的第 0 个成员分别被赋值为 in_x，in_y，in_z，in_mode，in_k，in_f，in_valid，in_sign。

同样，x，y，z，mode，k，f，valid，sign 的第 stage 个成员分别被赋值为 out_x，out_y，out_z，out_mode，out_k，out_f，out_valid，out_sign。

x，y，z，mode，k，f，valid，sign 的第 i 个和第 i+1 个成员被 CORDIC 单元的第 i 阶段连通，将在下一节描述。

4.2.6 带参数微旋转处理器的流水线内单元 RTL 结构

流水线内单元的输入输出包括 clock，reset，in_x，in_y，in_z，in_mode，in_k，in_f，in_valid，in_sign，out_x，out_y，out_z，out_mode，out_k，out_f，out_valid，out_sign。

流水线内单元含有本地参数 data_width 和参数 stage；以及 reg：x，y，z 分别为 data_width 比特和旋转方向 d。

流水线内单元还存储有 2^{-i} 的反正切和反双曲正切表。

计算时流水线内单元对 in_k，in_f，in_valid，in_sign 不做处理直接分别输出到 out_k，out_f，out_valid，out_sign。同时，单元根据 in_mode 决定旋转方向 d 和 x，y，z 从 in_x，in_y，in_z 的计算，将 x，y，z 输出为 out_x，out_y，out_z。

4.2.7 后处理器概览

后处理器对各个 CORDIC 函数执行表 5-2 中的后处理操作来得到最终结果，之后存在一个标准化阶段来将定点数转换为 IEEE 754 半精度浮点数格式。后处理器由 5 个流水线阶段组成，前 4 个阶段通过补偿参数 k 来得到最终定点结果，最后一个流水线阶段为标准化阶段。最重要的数据路径为前三个流水线阶段计算 k 和 $2\ln 2$ 的乘积，第四个阶段通过移位和加法计算 $\ln a = 2k\ln 2 + 2z_n$。

4.2.8 后处理器的 RTL 结构

后处理器的输入输出包括：clock，reset，in_x，in_y，in_z（分别为接受 CORDIC 流水线的 x，y，z），in_mode，in_k，in_f，in_valid，in_sign，out_result，out_valid。

后处理器包括常量 math_ln2，20 比特；inv_B，12 比特. 并有 reg：temp_sign，1 比特；temp_exp，带符号 6 比特；temp_res，10 比特；temp_floata，16 比特；这些 reg 为输出用。

后处理器的第一阶段包括 wire1_stage1（以下后缀均为 stage1），wire2，wire3，wire4 分别为 24，24，6，32 比特；reg1，reg2，reg3，均为 24 比特；sign，1 比特；k，6 比特；shift （移位量），5 比特，shift_dir（移位方向），1 比特，mode，3 比特；f，4 比特；valid，1 比特。

后处理器的第二阶段包括 wire1_stage2（以下后缀均为 stage2），为 24 比特；reg1，reg2，reg3，均为 24 比特；sign，1 比特；k，6 比特；shift （移位量），5

比特，shift_dir（移位方向），1 比特，mode，3 比特；f，4 比特；valid，1 比特。

后处理器的第三阶段包括 wire1_stage3（以下后缀均为 stage3），wire2，wire3，wire4 各为 24 比特；reg1，reg2，reg3，均为 24 比特；sign，1 比特；k，6 比特；shift （移位量），5 比特，shift_dir（移位方向），1 比特，mode，3 比特；f，4 比特；valid，1 比特。

4.3 实验结果和讨论

我们将半精度浮点 CORDIC 处理器在 FPGA Xilinx Virtex7 xc7v2000 上实现了。实验中所有的模组用 Verilog 编码并用 Vivado 2014.4 开发工具合成。在合成中我们使用不同的两个参数组合 M(基本 CORDIC 单元数) 和 N(每个单元进行的迭代数) 来寻找面积、时钟频率和功耗的平衡。然后我们评估处理器的性能和精度，与 FPGA、CPU、GPU 进行比较得到相对误差和加速比。

4.3.1 探寻参数平衡

我们提出的微旋转处理器的带参数设计允许我们探索面积、时钟频率和功耗的平衡以选择符合不同要求的最优的硬件实现。结果列于表 4-1。

随着 M 的降低，资源使用减少意味着更少的基本 CORDIC 单元被整合至 FPGA。随着 M 的减少，总功耗降低，其中静态功耗不变，动态功耗降低。当每个 CORDIC 单元的迭代次数增加时，FPGA 设计中的最重要路径以线性增长，最终导致执行频率以线性减慢。结果表明我们的 CORDIC 处理器最高时钟频率可达到 266.7MHz 同时功耗低至 0.979W。

表 4.1: 面积、时钟频率和功耗的平衡

M	N	资源			功耗 (W)			时钟频率 (MHz)	时钟周期 (ns)
		LUT	REG	DSP	动态	静态	综合		
32	1	6716	2574	8	0.338	0.641	0.979	266.7	3.75
16	2	3745	1534	8	0.195	0.639	0.834	166.7	6.0
8	4	2706	1014	8	0.130	0.638	0.768	90.9	11.0
2	16	2136	754	8	0.098	0.637	0.735	47.6	21.0
1	32	1792	633	8	0.075	0.637	0.712	25.0	40.0

表 4.2: 不同函数的精度和加速比

函数	平均相对误差 (%)	最大相对误差 (%)	加速比 (CPU)	加速比 (GPU)
sin	0.0449	0.229	10.8	0.0161
cos	0.0454	0.235	9.24	0.0160
arctan	0.0397	0.136	8.23	0.0186
sinh	0.0172	0.095	8.97	0.0164
cosh	0.0244	0.195	8.58	0.0163
exp	0.0460	0.276	5.33	0.0144
arctanh	0.0154	0.131	7.91	0.0163
ln	0.0358	0.136	11.51	0.0164
sqrt	0.0443	0.132	2.48	0.0143
arcsin	0.0425	0.235	18.3	0.0161
arcsinh	0.0412	0.228	12.35	0.0239
average	N/A	N/A	9.43	0.0168

4.3.2 性能和精度

我们考虑半精度浮点数的所有 65536 个不同数字然后比较各种 CORDIC 函数在三个不同平台上的结果和运行时间。一个是 Intel Xeon ES-4640CPU，拥有 2.20GHz 时钟频率，源码在 C 下编写并用 gcc4.7 在 O3 优化下编译。另一个是 Xilinx Virtex7 xc7v2000 FPGA 芯片使用半精度浮点 CORDIC 处理器在 266.7MHz 下运行。最终是 NVIDIA K40 GPU 包括 15 个 SMX（流媒体多处理器），每一个包括 192 个单精度 CUDA 核心，64 个双精度单元，32 个特殊函数单元 (SFU)，32 个存取单元，在 745MHz 下运行。表 4-2 展示了平均相对误差，最高相对误差和加速比。

我们得到的各函数对 CPU 的加速比中，反正弦函数的加速比最高为 18.3，另外较高的有反双曲正弦函数，对数函数和正弦函数，对 CPU 的加速比都在 10 以上。加速比最差的是开方函数，仅为 2.48，这是由于开方函数可以以不涉及三角或双曲函数的方式计算（即仅通过试商就可算得），而我们为了架构的统一运用数学等式使用与双曲函数相关的变换计算开方函数。所有 CORDIC 函数的平均相对误差小于 0.05%，也意味着我们的设计可以得到半精度浮点数要求的 11 位精度；最高相对误差小于 0.3%。相对于 GPU 我们的处理器得到小于 0.02 的加速

比，因 GPU 非常强大，包括 2880 个 SP 在 745MHz 下同时计算函数。与 CPU 相比，反正弦函数得到最大的加速比 18.3，开方函数得到最差的加速比 2.48。我们的处理器得到相对于 CPU 平均 9.43 的加速比，意味着它可以被用作科学计算的一个加速器。

4.4 未来展望

我们计划

1）继续对前处理器和后处理器进行优化，降低流水线阶段数目；

2）探求更快计算反正弦函数和反三角正弦函数的方法；

3）在前后处理器时，至少对部分函数如对数函数，直接处理浮点数的尾数部分和指数部分，而非转换为定点数再将定点结果转换为浮点数；

4）寻找处理冲突的方法，让计算简单的函数少等待计算繁杂的函数。

第五章　非线性函数基于查表的运算装置和方法

第四章介绍了我们的 CORDIC 处理器，它能以较高的精度计算各种基本超越函数。然而，如果我们愿意牺牲精度来换取更快的速度和更小的芯片面积和功耗（例如计算一个神经网络的激活函数），那么第二章所述的多项式近似（在对精度要求不高的情况下，我们可以简化为线性近似）也是一种方案。

本章提供一种非线性函数（不仅限于基本超越函数）运算装置及方法，解决现有技术在计算非线性函数时运算速度慢、运算装置面积大、功耗高等问题。

5.1　非线性函数运算装置介绍

我们提出一种非线性函数运算装置，用于根据一输入的浮点数计算非线性函数的函数值。装置包括：

5.1.1　配置模块

该模块用于将非线性函数的自变量分段为 N 个区间，在每个区间内，将非线性函数拟合为一个线性函数，分别得到 N 个线性函数，并获取 N 个线性函数的斜率值和截距值，其中，将 N 个线性函数的斜率值和截距值存储于斜率截距存储模块，每组斜率值和截距值一一对应于 N 个区间中一个区间的序号 index，并将序号 index 存储于选择模块，其中，序号 index 的取值范围为 [0, N-1]。因此，选择模块根据浮点数落入哪个区间，获取相应的区间的序号 index，并根据序号 index 在斜率截距存储模块得到相应的斜率值 k 和截距值 b。

配置模块还设定非线性函数自变量取值范围为 (-r，r)，并将边界值 r 的指数部分作为一偏移值 bias 输入至选择模块，所述选择模块根据浮点数及偏移值 bias，确定序号 index，并根据所述序号 index 得到对应的斜率值和截距值。需要说明的是，线性函数不可能覆盖所有非线性函数的取值，故可设定非线性函数自变量取值范围为 (-r，r)，以使得在 (-r，r) 中进行线性拟合。在设定好后，输入的浮点数落入 (-r，r) 中，这样只需根据浮点数所在区间就可得到相应的序号 index，但是，输入的浮点数也有可能不落入 (-r，r) 中，为了得到相应的序号 index，我们通过引入一偏移值 bias，配合浮点数，可在浮点数落入或者不落入取值范围 (-r，r) 时，都能得到相应的序号 index，具体包括：

当 bias-exp<0 时，在所述浮点数为正数情况下，index 取 N-1，在所述浮点数为负数情况下序号 index 取 0，其中，exp 为所述浮点数的指数部分；

当 0≤bias-exp<W-1 时，

$$index = 2^{W-1} + 2^{W-1-m-1} + frac[F-1:F-(W-1-m-1)+1]$$

其中，frac 为浮点数的尾数部分，W 为序号 index 的位宽 m=bias–exp，F 为所述浮点数的尾数的位宽，然后将 index 的每一位和所述浮点数的符号位进行异或运算；

当 bias-exp≥W-1，index 最高位为浮点数的符号位取反，低 W-1 位均为浮点数的符号位。根据本发明的一种实施方式，线性拟合模块包括乘法器和加法器，其中，乘法器用于将查表得到的斜率值 k 与浮点数相乘，得到相乘结果，加法器用于将乘法器得到相乘结果与查表得到的截距值 b 相加，得到线性函数的函数值 y。

5.1.2 查表模块

该模块存储有多个线性函数的斜率值和截距值，其中，多个线性函数由非线性函数分段线性拟合而得到，并且，查表模块根据浮点数获取相应的斜率值 k 和截距值 b；由于通过一组斜率值和截距值可以确定一个线性函数，因此，斜率值和截距值在存储时要有对应关系；

查表模块包括斜率截距存储模块和选择模块，其中，斜率截距存储模块用于存储多个线性函数所对应的斜率值和截距值，选择模块用于根据浮点数在斜率截距存储模块中选择并获取相应的斜率值 k 和截距值 b。

5.1.3 线性拟合模块

该模块用于根据查表模块得到的斜率值 k 和截距值 b，得到相应的线性函数 y=k×x+b，并将浮点数代入线性函数，得到线性函数的函数值，以作为浮点数在所述非线性函数中的函数值。

本装置的原理在于，将复杂的非线性函数拟合为多段线性函数，应该可知，分段的区间越小，线性函数与非线性函数的函数值越接近，亦即精度越高。确定输入的浮点数落入分段中的哪一段，由此确定出此段对应的线性函数，并将浮点数代入线性函数中，得到相应的函数值。

5.2 非线性函数运算方法

本段叙述通过上节装置计算非线性函数的方法，包括：

S0，用于将所述非线性函数的自变量分段为 N 个区间，在每个区间内，将非线性函数拟合为一个线性函数，分别得到 N 个线性函数，并获取所述 N 个线性函数的斜率值和截距值，其中，每组斜率值和截距值一一对应于 N 个区间中一个区间的序号 index，序号 index 的取值范围为 [0, N-1]。

S0 还包括，设定非线性函数自变量取值范围为 (-r，r)，并将边界值 r 的指数部分作为一偏移值 bias；

S1，根据浮点数从多个线性函数中获取一个线性函数的斜率值 k 和截距值 b，其中，多个线性函数由所述非线性函数分段线性拟合而得到；

S1 还包括，根据浮点数及 S0 的偏移值 bias，确定，并根据得到对应的斜率值和截距值。

S1 中，根据浮点数及所述偏移值 bias，确定序号 index，包括：

当 bias-exp<0 时，在所述浮点数为正数情况下，index 取 N-1，在所述浮点数为负数情况下序号 index 取 0，其中，exp 为所述浮点数的指数部分；

当 0 bias-exp< -1 时，

$$index = 2^{W-1} + 2^{W-1-m-1} + frac[F-1:F-(W-1-m-1)+1]$$

其中，frac 为浮点数的尾数部分，W 为序号 index 的位宽 m=bias-exp，F 为所述浮点数的尾数的位宽，然后将 index 的每一位和所述浮点数的符号位进行异或运算；

当 bias-exp W-1，index 最高位为浮点数的符号位取反，低 W-1 位均为浮点数的符号位。根据本发明的一种实施方式，线性拟合模块包括乘法器和加法器，其中，乘法器用于将查表得到的斜率值 k 与浮点数相乘，得到相乘结果，加法器用于将乘法器得到相乘结果与查表得到的截距值 b 相加，得到线性函数的函数值 y。

S2，根据得到的斜率值 k 和截距值 b，得到相应的线性函数 y=k×x+b，并将所述浮点数代入线性函数，得到线性函数的函数值，以作为浮点数在所述非线性函数中的函数值。

5.3 非线性函数运算装置的图示

图 5-1 是本文提出的非线性函数运算装置的结构图，如图 1 所示，装置包括查表模块 5 和线性拟合模块 6，其中，查表模块 5 用于根据由输入的自变量的值 x，以及外部配置进来的偏移量 bias，查找到对应的分段线性拟合的斜率和截距。

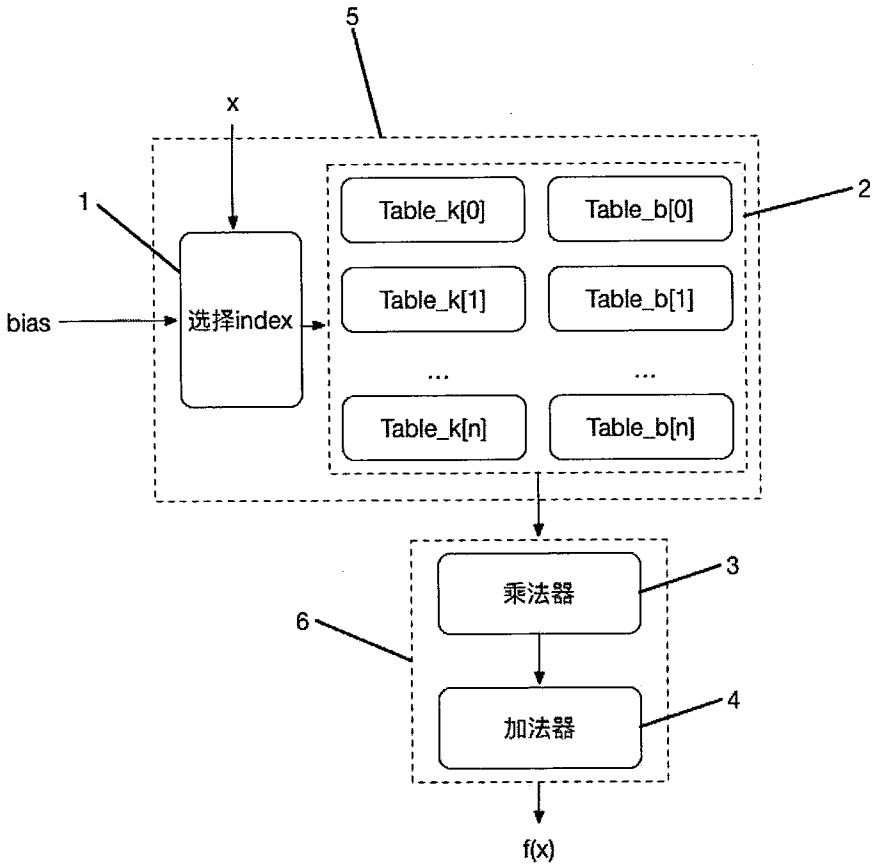

图 5.1: 非线性函数运算装置的结构图

查表模块 5 包括序号选择模块 1 和斜率截距存储模块 2，序号选择模块 1 用于根据输入的自变量值 x 和配置的偏移量 bias 计算出 index，斜率截距存储模块 2 用于根据序号选择模块 1 计算出的 index，选出斜率及截距。

线性拟合模块 6 用于根据查表模块 5 得到的斜率和截距通过线性拟合的方法得到最后结果。线性拟合模块 6 包括乘法器 3 和加法器 4，其中，乘法器 3 用于计算 k*x，加法器 4 用于计算 k*x+b。

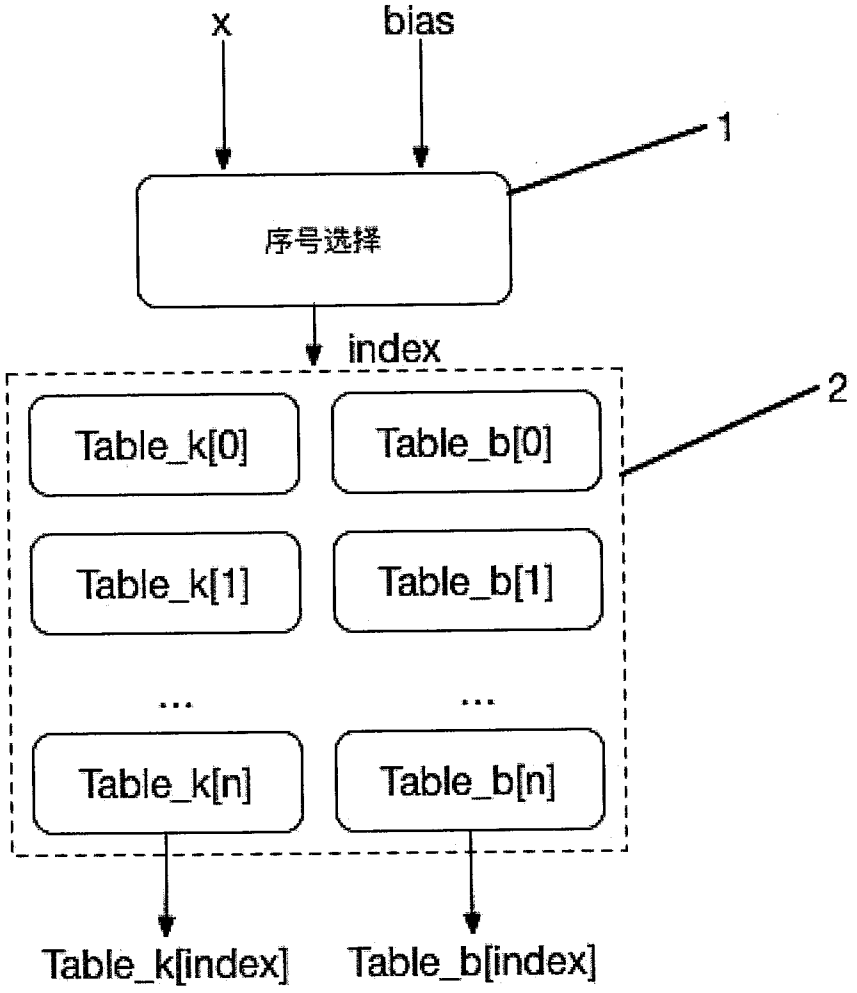

图 5.2: 非线性函数运算装置的内部结构图

图 5-2 是本文提出的非线性函数运算装置的内部结构图，如图 2 所示，查表模块 5 的输入值是非线性函数的自变量，以及偏移值。序号选择模块 1 根据所述自变量 x 和偏移量计算出 index。

斜率截距存储模块 2 中，Table_k 和 Table_b 里存储了非线性函数分段线性拟合的直线斜率和截距，Table_k 和 Table_b 里的值是可以配置的，在开始计算

之前，它们的值应该已完成配置。根据以上所述算出的 index，可以选出要使用的斜率 Table_k[index], 和截距 Table_b[index]。

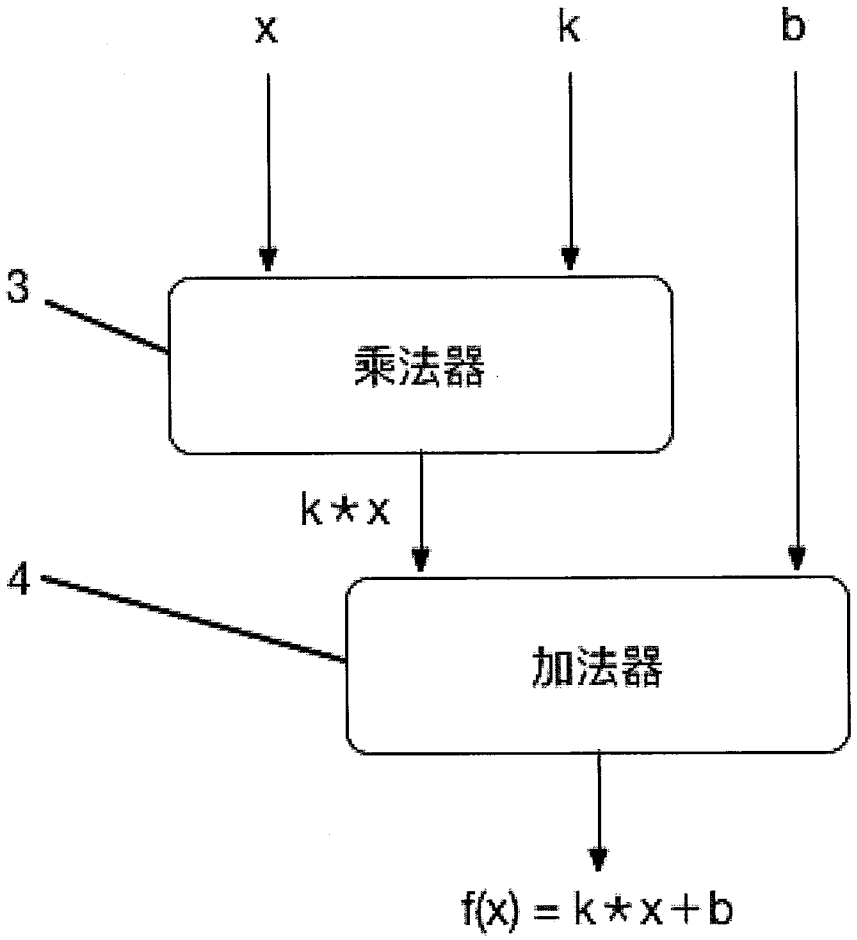

图 5.3: 线性拟合模块的内部结构图

图 5-3 是本文提出的非线性函数运算装置中线性拟合模块的内部结构图，如图 3 所示，线性拟合模块 6 有三个输入，x 表示自变量，即外部输入需要进行非线性变换的值，k 和 b 是查表得到的截距和斜率，输出是最终的结果 f(x)，线性拟合模块 6 实现的运算是：f(x) = k * x + b。

图 5-4 是本文提出的非线性函数运算装置实施的运算的原理图，如图 4 所示，查表模块 3 的输入是自变量 x，查找部件 3 根据 x 的值找到对应的斜率 k 和截距 b，并将 k 和 b 输出，在乘法器 4 中计算 k*x，并将结果和 b 输出，在加法器 5 中计算 k*x+b，计算得到最终的结果。

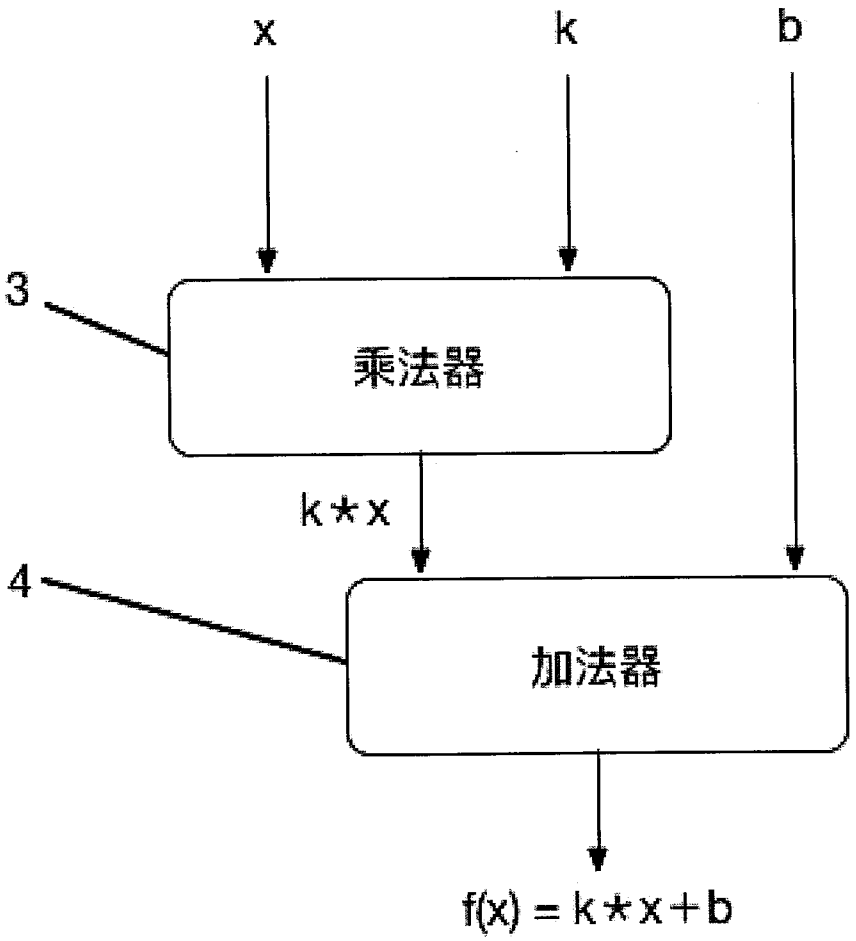

图 5.4: 非线性函数运算的原理图

5.4 有益效果

本章所述的非线性函数运算装置和方法将非线性函数拟合为多个线性函数，只需针对不同的自变量选择相应的线性函数，故在运算时只需要进行简单的加法和乘法运算，因此简化硬件设计，并提高运算速度，同时降低芯片功耗和面积。

第六章　结论

现代超越函数如正弦，余弦，双曲正弦，双曲余弦，指数，对数甚至反正弦等函数是很多常见科学和技术计算中不可或缺的部分。本文调查了各种基于CORDIC 算法的运算装置的研究现状，并实现一个半精度浮点 CORDIC 处理器，由三部分组成：预处理器，微旋转处理器和后处理器。我们提出四种方法来改进经典 CORDIC：1）参数缩减，2）拓展至反正弦和反双曲正弦，3）线性近似，4）流水线技术。另外，微旋转处理器中的参数设计允许我们寻找资源面积、时钟频率和执行时间中的平衡，以选择符合硬件资源和时钟频率要求的最佳的硬件实现。结果表明我们的设计可以达到最快 266.7MHz 得到平均对 CPU 加速比 9.43 同时功耗低至 0.979W。

另外，本文还讨论了基于查表的非线性函数运算装置，它可以在牺牲精度的情况下进一步提升运算速度，降低芯片面积与功耗。