基于 CORDIC 算法高精度浮点超越函数的 FPGA 实现

李全　　李晓欢　　陈石平
(桂林电子科技大学　信息与通信学院　541004)

摘要：如何以合理的硬件代价来实现高精度浮点超越函数计算，成为了微处理器设计过程当中的一个非常重要的问题。本论文提出了一种新的输入输出浮点处理单元硬件架构，它能将数据从 CORDIC 算法内部格式转变为处理器能够支持的 IEEE754 标准浮点数据格式。并且输入数据支持两种不同的角度单位浮点数据直接输入，即以度为单位和以弧度为单位。同时，硬件模块还直接支持超过 360 度(2π弧度)的大角度数据输入，这样就不需要用软件来对输入角度进行预处理，极大地减少了超越函数的计算时间。最后，该浮点硬件计算模块在 Altera 公司 Nios II 处理器系统中以用户自定义指令的形式完成了实现。通过用 C 语言程序来验证了浮点 CORDIC 模块的正确性。
关键词：CORDIC; 超越函数; 浮点数据; Nios II

The Implementation of High-precision Floating Transcendental Functions on FPGA Based on the CORDIC Algorithm

Li Quan　Li Xiao Huan　Cheng Shi Ping
(GuiLin University of Electronic Technology, Information and Communication Institute 541004)

Abstract: It is very important to implement floating-point transcendental functions of high precision with proper hardware cost for high performance microprocessor design. This paper present a novel architecture of input and output floating-point execution unit,it can converts data from the format of CORDIC algorithm system to the format accepted by processors.Two different unit of angle can be directly supported by the floating-point CORDIC module,these are:degrees and radians.Also,the wide angle even above 360 degrees(2π radians) can be directly supported by the hardware module.So we need not any software processing with the input angle,it greatly reduce the time spent in calculating the transcendental functions. Finally,we accomplish the hardware module by adding custom instruction on the Nios II processor system of Altera corporation.The floating-point CORDIC module's correctness was proved by C program running in the Nios II processor.

1　引言

本文主要完成了 J.S.Walther 的基本 CORDIC 算法分析，分析了圆周系统，线性系统，双曲系统三种 CORDIC 算法旋转系统。研究了传统结构下 CORDIC 算法的硬件实现以及流水线结构下 CORIDC 算法的硬件实现。实现了一些基本的浮点超越函数的计算，如 sin、cos、atan、以及平方根运算等。

主要工作包括：

1. 对 CORDIC 算法的原理进行了分析和推导，研究了圆周系统，线性系统，双曲系统三种 CORDIC 算法旋转系统。对每一旋转系统所能计算的函数进行了分析。

2. 研究了浮点运算的基本原理，对 IEEE-754 浮点数据国际标准进行了分析，对常见的浮点运算如：浮点加减法、浮点乘除的原理进行了研究。
3. 设计了经典 CORDIC 算法的硬件电路，以及基于流水线结构的 CORDIC 算法硬件电路，用 ModelSim 进行了电路仿真和测试，并且在 FPGA 中进行了硬件实现和验证。
4. 详细分析和研究了利用 CORDIC 算法计算 sin、cos 以及 atan 等函数的计算过程。在经典 CORDIC 算法原理的基础之上增加了计算的迭代精度，增加了 CORDIC 算法在计算正余弦函数时的角度输入范围，提高了输出结果的计算精度。
5. 提出了一种新的解决方案，实现了利用 CORDIC 算法计算 sin、cos 时，输入输出均兼容 IEEE-754 单精度浮点数据格式。并且进行了硬件实现和验证。使得设计出来的 CORDIC 算法硬件电路能够直接支持大多数微处理器接口，能够作为硬件 IP 核灵活的嵌入到各种 SOC 系统当中，提高系统性能。
6. 提出了一种新的设计方法，使得硬件超越函数计算模块可以直接支持两种不同角度单位数据的直接输入。目前可以直接支持以度为单位和以弧度为单位的浮点角度输入，这样可以提高在数字信号处理中某些算法的计算效率。
7. 本文提出了一种新的算法，使得硬件超越函数计算模块可以直接支持超 2π即 360 度的大角度数据输入，从而不需要使用软件来进行角度预处理，加快了算法的执行速度，拓宽了硬件模块的使用范围。
8. 对 Altera 公司的软核处理器 Nios II 进行了性能分析和接口电路设计，包括自定义指令接口和自定义外设接口。将兼容 IEEE-754 单精度浮点数据标准的 CORDIC 计算模块作为外接浮点处理单元融入 Nios II 处理器的指令集当中，使得用户能够直接在 C 语言环境下使用 CORDIC 模块进行超越函数的计算。

2 CORDIC 算法的原理简介

CORDIC 算法包含圆周系统，线性系统，双曲系统三种旋转系统，每种系统又分别具有两种运算模式，即旋转模式和向量模式，下面以 CORDIC 算法的圆周系统为例进行介绍。CORDIC 算法的圆周系统完成的是一个平面坐标旋转，如图 1 所示：

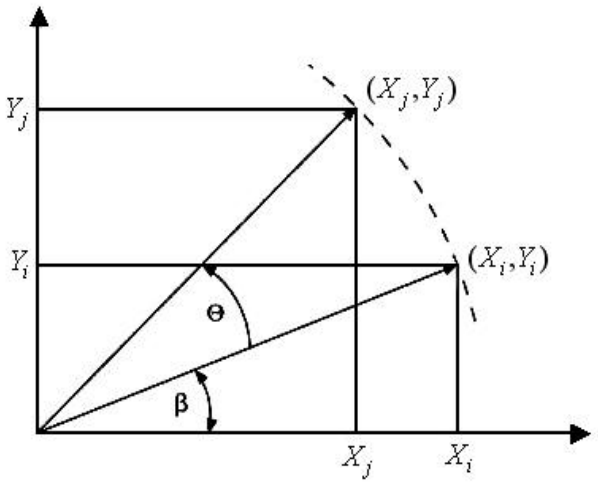

图 1　坐标旋转图

从图中可以看出，将向量(Xi,Yi)旋转 θ 角得到一个新的向量(Xj,Yj)，那么有：

$$X_j = R\cos(\theta+\beta) = X_i\cos(\theta) - Y_i\sin(\theta)$$

$$Y_j = R\sin(\theta+\beta) = X_i\sin(\theta) + Y_i\cos(\theta)$$

其中 R 为圆周的半径，把上式化为矩阵形式，平面旋转定义如下：

$$\begin{pmatrix} X_j \\ Y_j \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \end{pmatrix}$$

如果使用迭代的方法，则旋转的角度可以在多步之内完成，每一步的旋转完成其中的一小部分， 多步之后将会完成一个平面坐标的旋转。这时的迭代公式如下：

$$\begin{pmatrix} X_{n+1} \\ Y_{n+1} \end{pmatrix} = \begin{pmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & \cos\theta_n \end{pmatrix} \begin{pmatrix} X_n \\ Y_n \end{pmatrix}$$

消除 cos 因子得到单步旋转等式为：

$$\begin{pmatrix} X_{n+1} \\ Y_{n+1} \end{pmatrix} = \cos\theta_n \begin{pmatrix} 1 & -\tan\theta_n \\ \tan\theta_n & 1 \end{pmatrix} \begin{pmatrix} X_n \\ Y_n \end{pmatrix}$$

如果规定每一步旋转的角度只能为： $\theta_n = \arctan(\frac{1}{2^n})$

且令： $\sum_{n=0}^{\infty} S_n\theta_n = \theta$

其中，Sn={-1;+1}，即所有迭代角之和必须等于旋转角度。则每一步旋转的角度 Zn 为： $Z_n = \theta - \sum_{i=0}^{n-1} S_i\theta_i$

Sn 为 Zn 的符号函数，在旋转模式下：

$$S_n = \begin{cases} -1 & \text{if } Z_n < 0 \\ +1 & \text{if } Z_n \geq 0 \end{cases}$$

综合以上各式得到：

$$\begin{pmatrix} X_{n+1} \\ Y_{n+1} \end{pmatrix} = \cos\theta_n \begin{pmatrix} 1 & -S_n 2^{-n} \\ S_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} X_n \\ Y_n \end{pmatrix}$$

通过 N 次迭代，也就是向量旋转 N 次以后：

$$\begin{pmatrix} X_j \\ Y_j \end{pmatrix} = \prod_{n=0}^{N} \cos\theta_n \begin{pmatrix} 1 & -S_n 2^{-n} \\ S_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \end{pmatrix}$$

从上式中可以看出，对于给定次数的向量旋转，所有的 cos 乘积项是一个常数，可以提出来，于是令：

$$K = \frac{1}{P} = \prod_{n=0}^{N} \cos\theta_n = \prod_{n=0}^{N} \cos(\arctan(\frac{1}{2^n})) = \prod_{n=0}^{N} \frac{1}{\sqrt{1+2^{-2n}}}$$

具体的数值 K 取决于迭代次数 N。 对于所有的初始向量和所有的旋转角度而言，K 是一个常数。当迭代次数 N 趋向于无穷大时，K 值收敛且 K≈0.607253。通常把 K 称作聚焦常数。聚焦常数的倒数 P≈1.64676，通常称为旋转增益。于是旋转向量又可以写为：

$$\begin{pmatrix} X_j \\ Y_j \end{pmatrix} = K \times \prod_{n=0}^{N} \begin{pmatrix} 1 & -S_n 2^{-n} \\ S_n 2^{-n} & 1 \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \end{pmatrix}$$

CORDIC 算法有旋转模式和向量模式两种计算模式。在旋转模式中，Z 初始化为需要旋转的角，当 Z 旋转变为 0 时，公式变化如下：

$$X_{n+1} = X_n - S_n Y_n 2^{-n}$$

$$Y_{n+1} = Y_n + S_n X_n 2^{-n}$$

$$Z_{n+1} = Z_n - S_n \arctan(2^{-n})$$

经过 N 次迭代后，CORDIC 公式的输出变为：

$$X_{n+1} = P[X_0 \cos(Z_0) - Y_0 \sin(Z_0)]$$

$$Y_{n+1} = P[Y_0 \cos(Z_0) + X_0 \sin(Z_0)]$$

$$Z_{n+1} = 0$$

如果 X_0=1/P ，Y_0=0，Z_0=α ,N 次迭代后 CORDIC 公式的输出变为：

$$[X_{n+1}, Y_{n+1}, Z_{n+1}] = [\cos\alpha, \sin\alpha, 0]$$

从以上的分析可以看出， CORDIC 算法在圆周系统的旋转模式可以用来计算一个输入角的正弦、余弦和正切，此外还可以将极坐标变换为平面坐标。

向量模式将输入向量通过一个特定的角使 Y 变为 0。这种模式下的 CORDIC 算法跟旋转模式差不多，区别是旋转的方向取决于 Y 的符号，而不是旋转模式下 Z 的符号。

$$S_n = \begin{cases} +1 & \text{if } Y_n < 0 \\ -1 & \text{if } Y_n \geq 0 \end{cases}$$

因此 N 次迭代后 CORDIC 公式的输出变为:

$$X_{n+1} = P\sqrt{X_0^2 + Y_0^2}$$

$$Y_{n+1} = 0$$

$$Z_{n+1} = Z_0 + \arctan(\frac{Y_0}{X_0})$$

如果 Z_0=0，对于给定的 X_0 和 Y_0，N 次迭代后 CORDIC 公式的输出变为：

$$[X_{n+1}, Y_{n+1}, Z_{n+1}] = \left[P\sqrt{X_0^2 + Y_0^2}, 0, \arctan(\frac{Y_0}{X_0})\right]$$

从上式可以看出，CORDIC 算法在向量模式可以计算给定向量(X,Y)的长度和角度，即实现从平面坐标到极坐标的变换。

3 IEEE-754 浮点数据格式简介

IEEE-754 浮点数据标准有单精度、双精度和扩展精度三种浮点数表示格式。以单精度浮点数据格式为例，它由符号位、指数位和尾数位三部分组成，总共 32 位 。如图 2 所示：

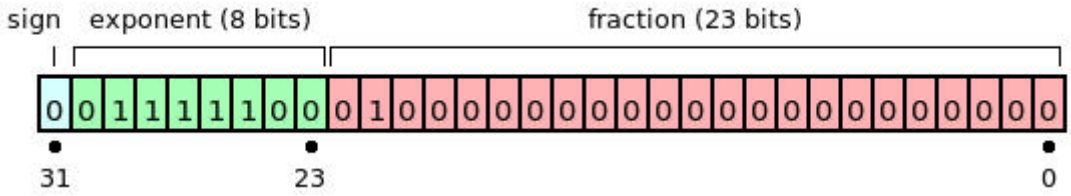

图 2　IEEE-754 单精度浮点数据格式

IEEE-754 单精度浮点数据格式包括了 1 位符号位、8 位指数位和 23 位小数位。并且在指数位和小数位之间还包含了一位隐含的“1”，所以总共有 24 位小数位。尾数通常表示的值在大于等于 1.0 到小于 2.0 之间。尾数的最高位就是那个隐含的“1”位，正好是在二进制小数点左边的第一个位，而余下的 23 个位的尾数则在小数点右边，即：1.mmmmmmmmm mmmmmmmmm mmmmmmmm

IEEE-754 单精度浮点数据的值由下式计算：

$$N = (-1)^S 2^{E-127}(1.M)$$

尽管在 1～2 之间有无限个实数，但是利用 IEEE-754 单精度浮点数据格式只能表示其中的八百万个(2^23)，因为我们使用的是 23 位的尾数（第 24 位永远是 1）。这就是浮点运算不准确的原因——使用单精度浮点数的计算精度只有 23 位。为了表示在 1.0 到小于 2.0 范围以外的数，就需要阶码了。浮点格式将计算 2 的由阶码的值指定的幂，并将尾数乘以这个数。对于 IEEE-754 单精度浮点数据格式来说，阶码是 8 位的，它使用余-127 格式，因此阶码 2^0 表示为 127。为了将一个阶码转换为余-127 格式，只需要加上 127 就可以了。例如 1.0 的 IEEE-754 单精度浮点表示为 0x3f800000，尾数是 1.0（包括隐含的“1”位），而阶码是 2^0，即在加上余-127 阶码值之和的 127。使用 24 位尾数，可以得到 6 个半十进制数字的精度，采用 8 位的余-127 阶码，单精度浮点数的动态范围大约是±2^128 或者大约±10^38。

为了在浮点计算过程中维持最大的精度，大多数的计算都使用规格化数。规格化浮点数就是尾数最高位是一的浮点数。保持浮点数为规格化是很有用的，因为这样就在计算过程中保持了最多的有效位。如果尾数的高端几个位全为零，那么在做计算时，尾数的有效位也就相应的减少了同样数量的位。因此，只用规格化数的浮点计算会更加精确。基本上所有的没有规格化的数都可以通过将尾数向左移位并递减阶码，直到尾数的最高位为一为止来进行规格化。如果需要进行规格化的数在二进制小数点左边有不止一位，那么只需要将尾数右移，并且递增阶码即可进行规格化。阶码是 2 的指数，每次递增阶码值，浮点数的值就被乘以二。类似地，递减阶码值浮点数就被除以二。将尾数左移一位的同时递减阶码就不会改变浮点数的值。

以下就是一个没有规格化的数：

0.10000 × 2^1

为了进行规格化，将尾数左移一位并递减阶码：

1.00000 × 2^0

在两种重要的情况下，浮点数是无法被规格化的。零就是其中一种情况。显然零是无法被规格化的，因为零的浮点表示就不包含一。无法规格化浮点数的第二种情况是尾数的高端几个位是零，但是移位阶码也是零，因此无法通过递减阶码来规格化尾数。IEEE 并没有禁止这些尾数高端位与移位阶码都是零的极小值，而是允许使用特殊的反向规格化数来表示它们。

4 迭代架构下 CORDIC 算法的硬件实现

实现 CORDIC 算法的硬件架构有几种不同的设计方案，理想的 CORDIC 算法硬件架构实现都是运行速度和资源面积的折衷。我们首先来分析一个迭代架构下的 CORDIC 算法实现，迭代架构下的 CORDIC 算法硬件结构是直接将 CORDIC 的计算公式进行展开得到的。从这一节里，我们可以看到一个耗费资源最小的 CORDIC 算法硬件实现，和一个最高性能的 CORDIC 算法硬件实现。

通过简单的把 CORDIC 算法的三个不同等式分别进行复用和实现，我们可以得到 CORDIC 算法的迭代硬件架构，迭代架构下 CORDIC 算法的硬件架构如图 3 所示：

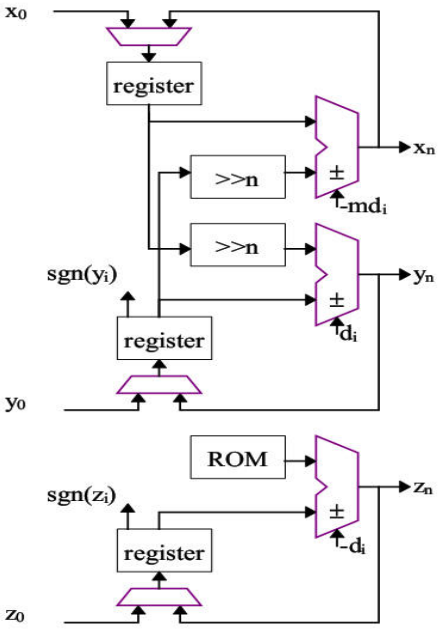

图 3 迭代架构下 CORDIC 算法的硬件架构

决策函数 d_i 由 y 或者是 z 寄存器的符号位来驱动。当计算模式是旋转模式时决策函数 d_i 由 z 寄存器的符号位来驱动；当计算模式是向量模式时决策函数 d_i 由 y 寄存器的符号位来驱动。在运算过程中，初始化值 X_0，Y_0 和 Z_0 通过多路选择器加载到 X，Y 和 Z 寄存器。在每一个迭代周期中，寄存器的值通过可变移位数移位器、加/减法器的运算后又写回到该寄存器当中。每一次迭代后可变移位数移位器都要变化移位数。与此相同，查找表 ROM 的地址在每一次迭代后也要增加，这样正确的 atan 基本角度数值才能送到 Z_n 加/减法器中进行运算。在最后一次迭代结束后，计算结果可以直接从加/减法器中读出。显然，需要一个简单的状态机来维持这样的迭代次序，以及为每一次迭代选择不同的移位数和不同的 ROM 表角度地址。

图 3 所设计的硬件架构使用以字为宽度(word-wide)的数据路径，也叫做比特位并行(bit-parallel)设计。但是其中的比特位并行可变移位数移位器在像是 ASIC 和 FPGA 的硬件实现中开销很大，并且运行速度很慢。特别是在 FPGA 中并没有一个标准的实现，直接实现往往效率很低，因为这样的移位器需要很高的扇入。通常，在 FPGA 中实现这样的移位器需要经过几层逻辑（例如：信号从输入端传到输出端要穿过好几层 LE）。结果就是耗费了大量的资源却还是得到了一个低速的设计。相比较而言，比特位串行(bit-serial)设计是更为紧凑的 CORDIC 算法硬件架构。

通过简化互联和逻辑，比特位串行设计相对于与比特位并行的设计来说可以工作在更高的时钟频率下。当然，比特位串行设计每一次迭代需要 w 个时钟周期（w 是计算过程中数据的位宽）。比特位串行设计包括了三个一位的串行加/减法器、三个移位寄存器

和一个串行的只读存储器（ROM）。每一个移位寄存器的长度都等于要计算数据的位宽。比特位串行的 CORDIC 硬件架构如图 4 所示：

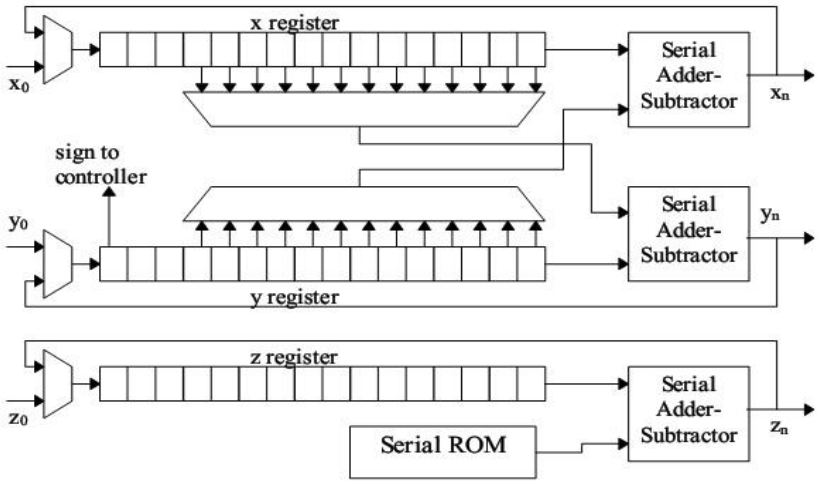

图 4　比特位串行的 CORDIC 硬件架构

在计算过程中，为了初始化 X_0，Y_0 和 Z_0 寄存器，左端的负责加载控制的多路选择器必须打开 w 个时钟周期（这些寄存器也可以设计成并行初始化结构）。一旦初始化加载完毕，数据右移通过串行的加/减法器，然后又回到左端的寄存器当中。每一次迭代都需要 w 个时钟周期才能返回正确的计算结果到寄存器当中。在每一次迭代开始时，负责控制的状态机读出 y 或者是 z 寄存器的符号位，然后直接用该符号位来设置加/减法器是执行加法运算还是减法运算。同样，在每一次迭代周期开始，负责交错的多路选择器要选择相应的寄存器位输出。迭代结束后，计算结果可以通过串行加/减法器的输出端读出，同时新一轮的初始化数值又被加载进寄存器中。

CORDIC 算法的比特位串行设计显然要比图 3 所示的比特位并行设计简单的多，但是即便是这样，架构里面用到的多路选择器移位抽头还是会给一些硬件实现带来麻烦（比如说 FPGA）。虽然如此，比特位串行设计的内部互联已经是最少，寄存器间的硬件逻辑也已经是最简单。这样的架构可以允许位时钟的频率最高可以接近硬件的开关速率。能够使用极高的时钟频率，弥补了比特位串行设计一次迭代需要多个时钟周期造成的速度上的影响。

CORDIC 算法在圆周系统下的旋转模式中可以计算出输入角度的正余弦。在定点 CORDIC 内核中需要对数据格式进行定义和对输入角度进行预处理。

定点 CORDIC 内核内部所有的角度、数据表示方法都是采用整数的数据类型，因为整数便于移位和加减运算，而浮点格式的加减运算则比较麻烦，一般不会采用浮点格式的数据类型来设计硬件 CORDIC 内核。CORDIC 内核内部有两种不同单位的数据格式：

1、角度（单位可以是度或者是弧度）

2、计算数值（对于正余弦函数来说计算结果绝对值总是小于 1）

在正余弦函数的硬件实现中，我们采用度来表示输入角度的单位，为了提高角度的表示精度，令 2^32 代表 360 度，则：

1 度 = (2^32)/360 ≈ 11930465(十进制) = 0xb60b61(16 进制)

有几个常见的角度用此方式表示的数值如下：

30 度	= 0x15555555	45 度	= 0x20000000
60 度	= 0x2aaaaaaa	90 度	= 0x40000000
180 度	= 0x80000000	270 度	= 0xc0000000

在计算 sin、cos 函数的 CORDIC 内核内部，数值的大小范围在[0,1]，因此令 2^30 代表 1，即用 30 位二进制数来表示数值 1，第 31 位作为保护位用来防止数据溢出。第 32 位用作符号位。

有几个常见的数值用此方式表示的值如下：

0.1 = 0x06666666　　0.2 = 0x0ccccccc

0.5 = 0x20000000　　1.0 = 0x40000000

CORDIC 算法在圆周系统下的旋转模式中角度的收敛范围小于 99 度，也就是说只能处理位于第一象限的角度。对于 0～360 度的角度输入需要预先进行处理，要把处于第 2、3、4 象限的角度统统转第到 1 象限。这就需要判断一个给定的角度是处于哪个象限。这样的判断很简单，因为根据 CORDIC 内核内部的角度表示法：

$$\begin{cases} 90度=0x40000000 \\ 180度=0x80000000 \\ 270度=0xc0000000 \end{cases}$$

因此，只需要处理角度数值的最高 2 位，就可以知道该角度所处的象限了。 用 VerilogHDL 语言来描述，于是就可以得到：

$$\begin{cases} iAngle[31:30] == 2'b00 \quad 第1象限 \\ iAngle[31:30] == 2'b01 \quad 第2象限 \\ iAngle[31:30] == 2'b10 \quad 第3象限 \\ iAngle[31:30] == 2'b11 \quad 第4象限 \end{cases}$$

假设角度 α 位于 0～90 度，是象限处理以后的角度数值。

当 iAngle 在第 1 象限时，不需要进行正余弦交换：

$$\begin{cases} \sin(iAngle) = \sin(\alpha) \\ \cos(iAngle) = \cos(\alpha) \end{cases}$$

当 iAngle 在第 2 象限时，需要进行正余弦交换：

$$\begin{cases} \sin(iAngle) = \sin(90+\alpha) = \cos(\alpha) \\ \cos(iAngle) = \cos(90+\alpha) = -\sin(\alpha) \end{cases}$$

当 iAngle 在第 3 象限时，不需要进行正余弦交换：

$$\begin{cases} \sin(iAngle) = \sin(180+\alpha) = -\sin(\alpha) \\ \cos(iAngle) = \cos(180+\alpha) = -\cos(\alpha) \end{cases}$$

当 iAngle 在第 4 象限时，需要进行正余弦交换：

$$\begin{cases} \sin(iAngle) = \sin(270+\alpha) = -\cos(\alpha) \\ \cos(iAngle) = \cos(270+\alpha) = \sin(\alpha) \end{cases}$$

接下来还要进行符号位判断，由以上的正余弦交换公式可以立即判断出各个象限 sin 和 cos 的符号位，如下表所示。

象限	1	2	3	4
sin	+	+	−	−
cos	+	−	−	+

硬件采用 Verilog HDL 语言进行描述，为了提高计算精度，使用了 32 次迭代运算，角度以及计算结果均用 32 位的寄存器来表示。因此每一次计算需要花费大约 32 个时钟周期才能完成，CORDIC 硬件架构是比特位并行架构。使用 ModelSim SE 6.1f 对本模块进行功能仿真，我们选取一些特定的角度输入，以及任意的角度输入来对硬件进行测试。

角度输入是 30 度，也就是输入角度 iAngle=0x15555555 时的计算结果如图 5 所示：

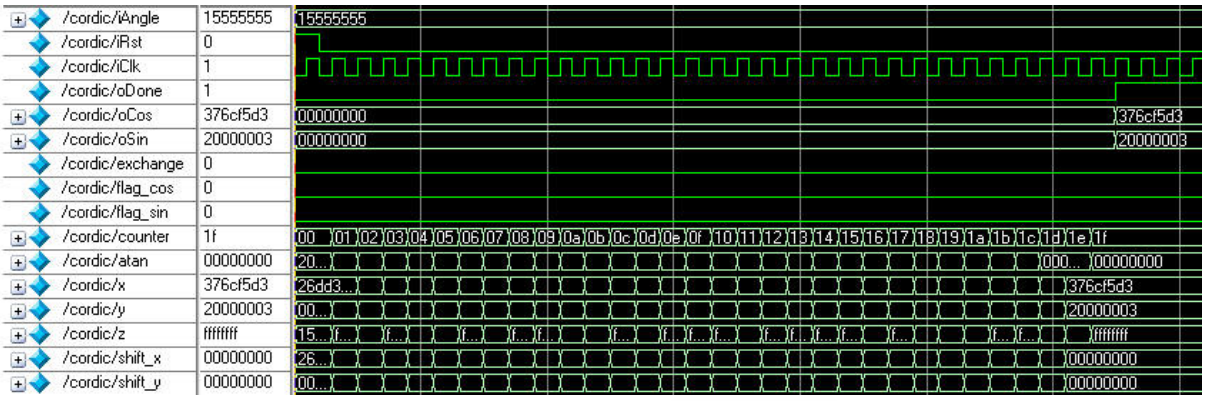

图 5　输入角度为 30 度时的计算结果

在仿真波形图中，iAngle 信号为输入的需要计算的角度，iRst 信号为模块的复位信号，iClk 信号为时钟周期信号，oCos 信号为余弦计算结果输出，oSin 信号为正弦计算结果输出，oDone 信号为 1 表示正余弦输出结果有效。从复位结束后，一共用了 32 个时钟周期得到计算结果。我们知道：

$$\begin{cases} \cos(30) = 0.866025(\text{浮点表示}) = 0x376cf5d0(\text{定点表示}) \\ \sin(30) = 0.500000(\text{浮点表示}) = 0x20000000(\text{定点表示}) \end{cases}$$

这是理想的计算结果，从仿真波形图中我们可以看出通过 32 次迭代实际的计算结果为：

$$\begin{cases} \cos(30) = 0x376cf5d3(\text{定点表示}) \\ \sin(30) = 0x20000003(\text{定点表示}) \end{cases}$$

本模块在 Altera 公司的 FPGA 器件中进行了硬件验证，采用的综合工具是 Quartus II 7.1+sp1，选用的芯片是 Cyclone 系列 EP1C6Q240C8。综合以后，一共需要 603 个逻辑单元，约占用 10%的芯片资源。综合后的时序分析报告如图 6 所示：

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock
1	Worst-case tsu	N/A	None	7.144 ns	iAngle[30]	oSin[23]~reg0	--	iClk
2	Worst-case tco	N/A	None	8.867 ns	oSin[5]~reg0	oSin[5]	iClk	--
3	Worst-case th	N/A	None	-4.917 ns	iAngle[30]	oSin[30]~reg0	--	iClk
4	Clock Setup: 'iClk'	N/A	None	77.37 MHz (period = 12.925 ns)	counter[1]	y[30]	iClk	iClk
5	Total number of failed paths							

图 6　综合后的时序分析报告

由时序分析报告中我们可以看出，在默认的约束条件下，综合以后的寄存器间的最长路径延迟时间为 12.925ns，　因此最高的工作频率 Fmax = 77.37MHz，这是在比特位并行迭代架构下的工作频率。

输入角度为 30 度的 Quartus II 时序仿真波形分别如图 7 所示：

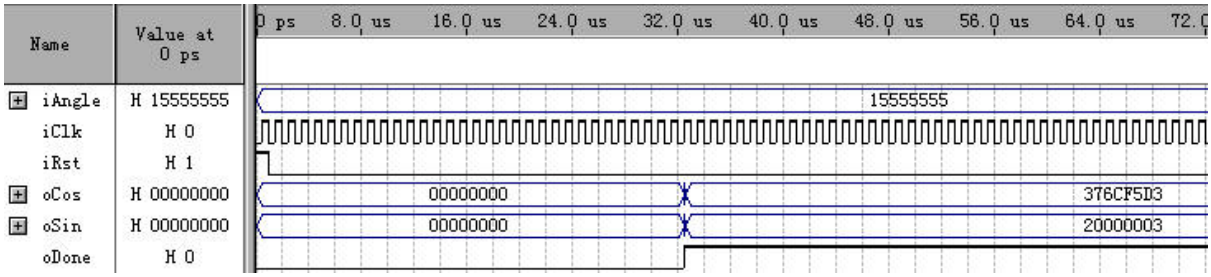

图 7　输入角度为 30 度的 Quartus II 时序仿真波形

在这样的硬件架构下，CORDIC 算法的计算相对误差不超过 5.6×10^-9，计算精度是比较高的。这主要是因为迭代次数达到 32 次，此时的角度分辨率可以精确到 arctan(2^-32)≈1.334 ×10^-8(度)。由于 CORDIC 算法的误差主要由角度迭代误差产生，角度分辨率越高于是计算精度就越高。

5　流水线架构下 CORDIC 算法的硬件实现

以上的硬件设计都是基于 CORDIC 算法迭代架构之下，然而在这样的迭代架构下，数据率是时钟频率的 1/n，n 是迭代次数。也就是说要经过 n 次迭代才能得到计算结果。显然这样的算法效率不高， 但是硬件规模确比较小。如果我们采用流水线的设计思想，把每一次的迭代处理并行起来， 使得每一个周期就能够进行 n 次运算， 那么计算的效率将会大大的提高。CORDIC 算法适合采用流水线架构来加速，因为 CORDIC 算法每个周期的运算输入都只跟前一周期的计算结果有关。CORDIC 算法采用流水线的思想来设计，就是利用增加硬件的规模来换取运行速度的提升，即面积与速度互换的设计思想。流水线架构下 CORDIC 算法的硬件实现如图 8 所示：

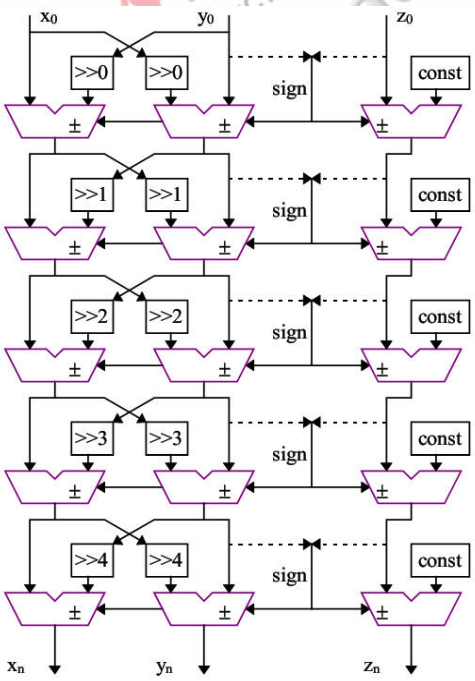

图 8　流水线架构下 CORDIC 算法的硬件实现

CORDIC 算法通过采用流水线结构，把每一次的迭代过程进行展开，不仅提高了运行速度，同时还可以带来两个方面硬件架构的明显简化。第一个方面就是在流水线架构中，每一级的移位器移位宽度都是固定的常数，这就意味着移位器可以简化到只需要连线就可以了。我们知道，在迭代架构的硬件实现中需要一个可变移位数移位器，这样的移位

器给硬件结构带来了很大的设计上的困难，并且还降低了整个系统的时钟运行频率。因为固定位数的移位在硬件实现上只需要进行重新连线就可以了，这就极大地简化了系统设计，增加了系统的运行速度。

第二个方面的简化体现在查找表上。CORIDC 算法的计算过程中的每一步都需要用到查找表，来查找相应的反正切角度数值。采用流水线架构以后，每一步的角度判断都分散开来执行并行计算，所以对于每一级的计算过程而言，所用到的角度数值是一个常数。这样就再也不需要一个像是迭代架构下的 ROM 表来进行查表运算，原先存储在 ROM 中的反正切角度数值，就也可以用电路的硬件连线来代替。这样的做法又可以显著的增加硬件的最大时钟运行频率。

对于流水线架构而言，整个 CORDIC 的计算过程可以归纳为一个互连的加/减法器阵列， 其中每一级加/减法器之间插入用于缓存的寄存器结构。它的时钟运行频率比起迭代架构来说要高很多。

流水线设计思想的引入，极大地提高了 CORDIC 算法的硬件性能。对于正余弦函数的硬件实现来说，原来需要 32 个周期才能得到的计算结果，现在一个周期就可以计算出来。硬件的功能仿真波形如图 9 所示：

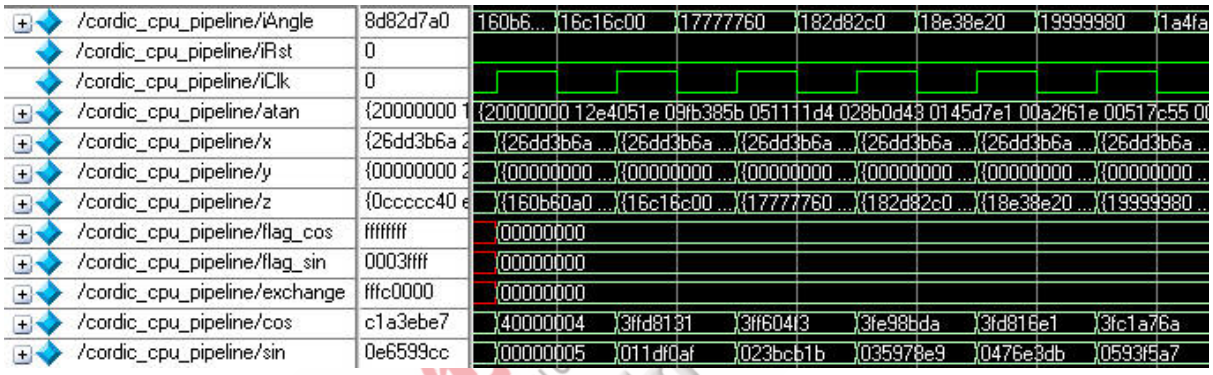

图 9　流水线架构下正余弦函数的硬件实现

在测试环境中，我们让角度值从 0 度开始递增， 递增的角度间隔为 1 度，图 9 中显示了输入角度为 0～5 度时的正余弦函数的计算结果。在理想情况下，正余弦函数的计算结果如下：

$$\begin{cases} \cos(0) = 1.000000(\text{浮点表示}) = 0x40000000(\text{定点表示}) \\ \cos(1) = 0.999848(\text{浮点表示}) = 0x3ffd812f(\text{定点表示}) \\ \cos(2) = 0.999391(\text{浮点表示}) = 0x3ff604f1(\text{定点表示}) \\ \cos(3) = 0.998629(\text{浮点表示}) = 0x3fe98bda(\text{定点表示}) \\ \cos(4) = 0.997564(\text{浮点表示}) = 0x3fd816e2(\text{定点表示}) \\ \cos(5) = 0.996195(\text{浮点表示}) = 0x3fc1a768(\text{定点表示}) \end{cases}$$

我们可以看出实际的计算结果为：

$$\begin{cases} \cos(0) = 0x40000004(\text{定点表示}) \\ \cos(1) = 0x3ffd8131(\text{定点表示}) \\ \cos(2) = 0x3ff604f3(\text{定点表示}) \\ \cos(3) = 0x3fe98bda(\text{定点表示}) \\ \cos(4) = 0x3fd816e1(\text{定点表示}) \\ \cos(5) = 0x3fc1a76a(\text{定点表示}) \end{cases}$$

计算结果的精度是比较高的，我们把正余弦函数的计算结果用连续的曲线来进行表

示，同时角度的递增由 0 度开始，以 1 度为单位线性增加，所得到的仿真波形如图 10 所示：

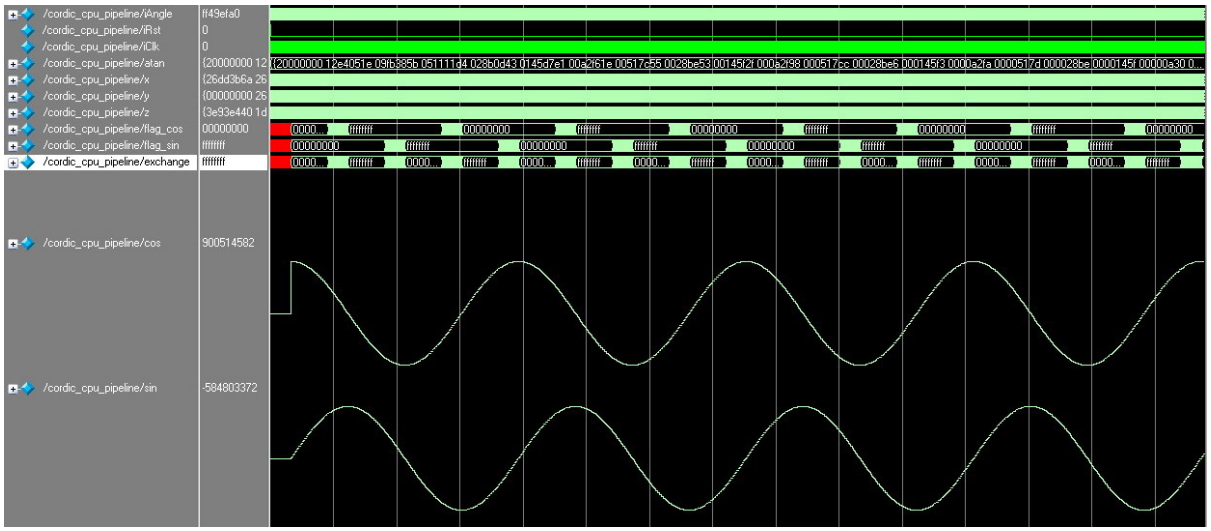

图 10　流水线架构下正余弦函数的硬件仿真

由于 CORDIC 算法的特性，正余弦函数可以同时计算出来，因此对于同一个输入角度来说，正余弦函数严格满足 90 度的相位关系，这点由图 10 也可以看出来。因此，基于流水线架构的 CORDIC 算法正余弦函数硬件计算模块可以用在比如直接数值频率合成（DDS）、软件无线电等诸多领域。

基于流水线结构的正余弦函数计算模块也在 Altera 公司的 FPGA 器件中进行了硬件验证。综合以后，一共需要 4287 个逻辑单元，约占用 72%的芯片资源。对比迭代架构下所花费的资源（603 个逻辑单元）来说，流水线下资源占用量增加了 7 倍。这是因为每一级的运算都要同时并行进行，需要大量的加/减法器，以及布线资源。我们再来看看时序分析的情况，综合后的时序分析报告如图 11 所示。

Timing Analyzer Summary

	Type	Slack	Required Time	Actual Time	From	To	From Clock	To Clock
1	Worst-case tsu	N/A	None	11.315 ns	iRst	z[2][17]	--	iClk
2	Worst-case tco	N/A	None	17.762 ns	altshift_tap...	cos[17]	iClk	--
3	Worst-case th	N/A	None	-2.424 ns	iAngle[11]	z[0][11]	--	iClk
4	Clock Setup: 'iClk'	N/A	None	134.10 MHz (period = 7.457 ns)	z[6][31]	x[7][30]	iClk	iClk
5	Total number of failed paths							

图 11　流水线架构综合后的时序分析报告

由时序分析报告中我们可以看出，在默认的约束条件下，综合以后的寄存器间的最长路径延迟时间为 7.457ns，因此最高的工作频率 Fmax = 134.10MHz，由于采用的是流水线架构，每一次计算平均而言只需要花费 1 个时钟周期，因此每一秒钟可以进行 1.34×10^8 次正余弦函数的计算，运算效率是迭代架构下的 55.37 倍。

流水线结构的正余弦函数计算的时序仿真波形如图 12 所示，时序仿真波形图中，角度输入也是从 0 度开始，以 1 度为单位进行递增。

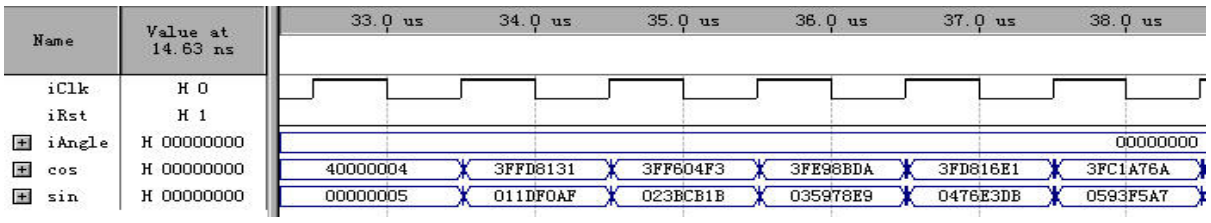

图 12　流水线结构的正余弦函数计算的时序仿真波

由时序仿真波形图中也可以看出，时序仿真完全满足设计要求，与功能仿真的结果是一样的。

6 输入输出均兼容 IEEE-754 标准的浮点正余弦函数的 CORDIC 实现

前面介绍的基于 CORDIC 算法正余弦函数的硬件实现，角度的输入以及最后计算结果的输出均采用的是 CORDIC 算法内部的数据格式。我们常用的数据单位必须经过转换才能被 CORDIC 内核使用，同时 CORDIC 内核输出的计算结果也要经过转换才能变为我们熟悉的表示方式。做为一个 CPU 的浮点协处理器，我们的模块主要与 CPU 进行数据传递，而目前几乎所有的 CPU 都使用 IEEE-754 标准来表示浮点数据，因此做到与 IEEE-754 标准兼容的数据输入输出对于基于 CORDIC 算法的正余弦函数计算协处理模块来说是非常重要的。这就包括了角度输入与 IEEE-754 标准兼容和计算结果输出与 IEEE-754 标准兼容两个部分。

对于正余弦函数的计算来说，角度的输入单位主要包括度和弧度两种，对于大部分超越函数的计算来说，角度表示无论是用度为单位还是弧度为单位都需要是一个浮点数据类型。本文所设计的基于 CORDIC 算法的正余弦函数计算协处理模块支持 IEEE-754 浮点数据格式的度和弧度两种数据直接输入。这样就需要对输入角度进行前处理，把 IEEE-754 格式的浮点角度输入转化为 CORDIC 内核能够使用的内部数据格式，并且要尽可能的减小转换所带来的误差。

首先要找到浮点角度表示和 CORDIC 内部角度表示之间的对应关系。通过前面章节对 IEEE-754 浮点数据类型及其基本运算的介绍，我们知道浮点的角度表示如果采用 IEEE-754 的单精度数据格式，那么它就带有 1 位符号位、8 位指数位和 23 位尾数位，其中尾数位带有一位隐含的 1，因此总共有 24 位尾数。比如角度 1.0 度的单精度浮点数据输入为 0x3f800000，这个值要对应到 CORDIC 内核的内部表示法的 0xb60b61。同理，角度 30.0 度的浮点输入 0x41f00000 要对应到内部的表示数值 0x15555555，可以看出这是一个乘法的对应关系。要用一个硬件乘法器结构来实现这样的线性映射关系。要实现一个浮点乘法器对于硬件设计来说开销是很大的，不仅会占用大量的芯片面积资源，还会降低系统的运行时钟频率。但是可以发现，对于一个确定的浮点角度单位而言（比如说度或者弧度单位），只需要进行一个浮点数与一个常数的相乘运算。更进一步，一个浮点数据乘以一个固定的常数我们可以优化为两个整数进行相乘。最简单的思路就是直接用该浮点数的尾数（要补上隐含的 1）与固定常数相乘，然后再通过移位和固定宽度的位选取来获得结果。这里先暂时不考虑该浮点角度的符号位，符号的判断会另外处理，因为 cos 是偶函数可以不用考虑符号位，sin 是奇函数符号位的判断也比较简单，后面会和正余弦交换统一处理。

前处理单元处理的是输入给 CORDIC 内核的数据，后处理单元处理的就是内核的输出数据。由对前处理单元的叙述中可知，在计算 sin、cos 函数的 CORDIC 内核内部，数值的大小范围在[0,1],因此令 2^30 代表 1，即用 30 位二进制数来表示数值 1，第 31 位作为保护位用来防止数据溢出。第 32 位用作符号位。后处理单元的作用就是要将 CORDIC 内核输出的数据格式转为标准的单精度 IEEE-754 浮点格式。后置处理单元硬件结构如图 13 所示。

图 13 中的 LZDC（Leading zero detection circuit）为前导零检测电路，其功能是用来统计数据从最高位开始连续 0 的个数，再根据它把计算出来的数值转换成 IEEE-754 标准格式。在转换过程中需要对尾数进行左移，此位移量再与前置处理器单元传来的参考指数值相减得到输出指数值。符号位直接由前置处理过程得到，最后将符号位、指数数值、小数按位连接得到 IEEE-754 格式的数据，并从硬件接口输出端输出。

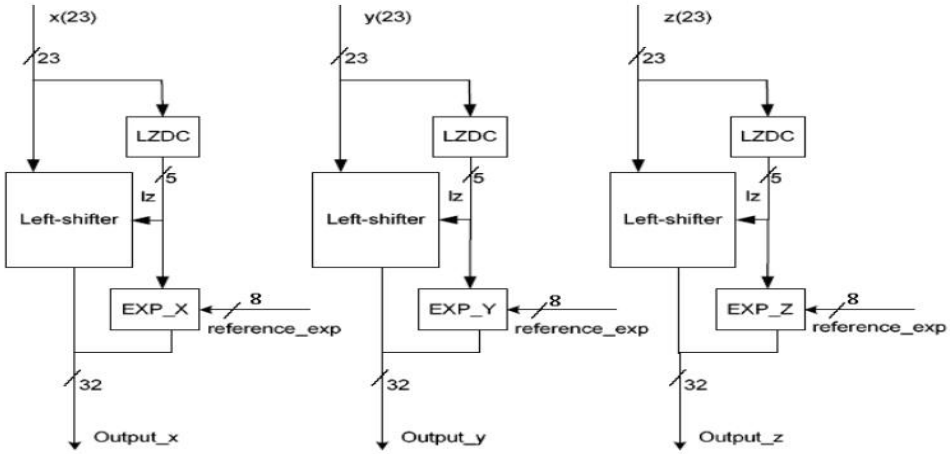

图 13　后置处理单元硬件结构

通过前面介绍的输入角度的前处理和输出数据的后处理过程，基本上完成了兼容 IEEE-754 标准的数据接口设计。硬件的接口模块如图 14 所示：

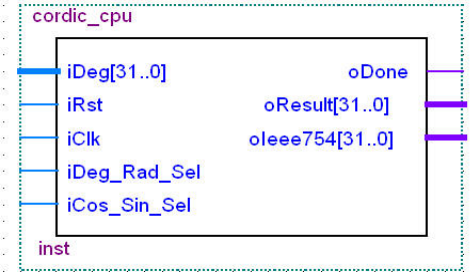

图 14　硬件的接口模块图

由图 14 可以看出，本模块一共有 5 个数据输入端口，3 个数据输出端口。数据输入端口分别是 iDeg、iRst、iClk、iDeg_Rad_Sel 和 iCos_Sin_Sel。数据输入端口 iDeg 输入数据格式直接使用浮点单精度 IEEE-754 标准，具有 32 位数据线，直接支持浮点角度单位输入以及浮点弧度单位的输入。通过直接硬件支持两种角度单位的输入，可以避免角度换算时带来的计算误差以及用软件进行换算时带来的计算速度减慢问题。因为对于某些算法来说，直接使用角度作为单位来计算比较直观方便。同时，对于另外一些算法来说，使用弧度作为单位来计算比使用角度来计算更容易。所以，本正余弦函数计算协处理模块设计上可以直接支持以上两种单位的浮点角度输入，不需要使用软件事先进行转换，节省了计算时间，提高了计算效率。

iRst 是模块的硬件复位信号，高电平有效。硬件模块在 iRst 信号有效期间初始化各个寄存器以及进行计算控制的状态机模块。同时 iRst 信号还是计算开始信号，当 iRst 由有效变为无效以后的第一个 iClk 时钟上升沿开始进行第一次迭代运算。iRst 信号需要保持有效至少一个 iClk 时钟周期，以确保初始化过程的正确完成。

iClk 是时钟周期输入信号。模块在每个 iClk 时钟的上升沿进行迭代运算。

iDeg_Rad_Sel 是角度和弧度单位选择信号。在 iDeg_Rad_Sel 输入为低电平时表示输入信号 iDeg 的单位为角度，在 iDeg_Rad_Sel 输入为高电平时表示 iDeg 单位为弧度。模块通过该信号来区分角度和弧度单位的输入。

iCos_Sin_Sel 是输出余弦正弦选择信号。当 iCos_Sin_Sel 为低电平时输出为输入角度的余弦值，当 iCos_Sin_Sel 为高电平时输出为输入角度的正弦值。因为由于 CORDIC 内核的特性，正余弦函数可以同时计算出来，所以用 iCos_Sin_Sel 信号来选择输出是

余弦还是正弦。

数据输出端口分别是 oDone、oResult 和 oIEEE-754 这三个端口。oDone 端口是输出有效信号， 当 oDone 为高电平时表示输出数据有效， 当 oDone 为低电平时表示数据还处于计算过程中。在 oDone 信号有效后就可以从 oResult 端口或者是 oIEEE-754 端口读出计算结果。

oResult 端口是计算结果输出端口。oResult 端口输出的是 iDeg 端口输入角度的正弦或者余弦函数，oResult 端口输出数据格式采用 CORDIC 内核的内部数据格式，不能直接与 CPU 接口。

oIEEE-754 端口是计算结果输出端口。oIEEE-754 端口输出的是 iDeg 端口输入角度的正弦或者余弦函数，oIEEE-754 端口输出数据格式采用 IEEE-754 单精度浮点数据格式，可以直接与各种 CPU 接口。

假设需要计算 30 度角的余弦函数，我们需要令 iDeg_Rad_Sel 为 0，iCos_Sin_Sel 为 0。因为 30.0 的 IEEE-754 单精度浮点表示为 0x41f00000，所以 iDeg 输入为 0x41f00000。硬件仿真测试波形如图 15 所示：

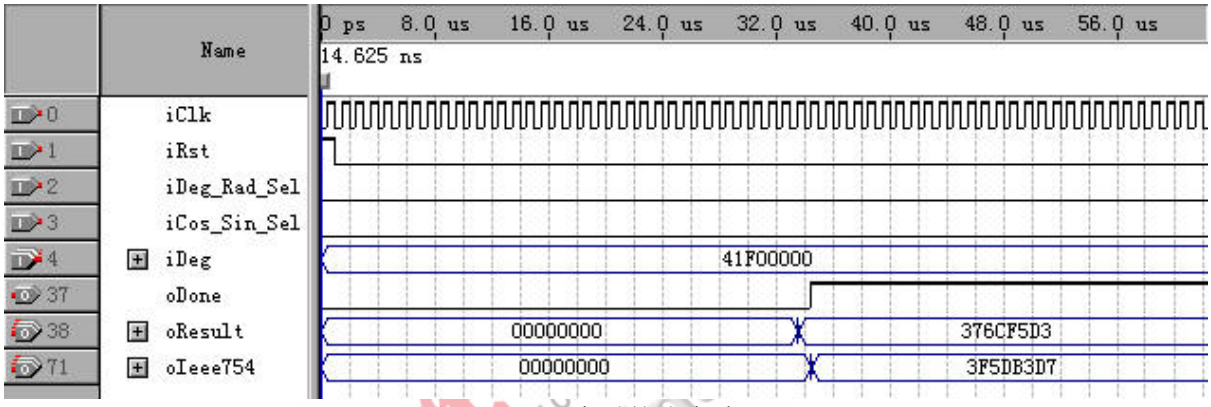

图 15　30.0 度时的仿真波形

我们知道 cos(30.0) = 0.866025，由图 15 可以看出计算得到结果为 0x3f5db3d7，这是 32 位的 IEEE-754 单精度浮点数据格式，它等于浮点数 0.866025。我们可以通过单精度浮点数据转换工具来进行这种数据类型的换算：

图 16　数据类型的换算

从这个计算中我们可以看出，本模块在单精度表示范围之内没有误差，因为单精度浮点数最多有 24 位的小数，而 CORDIC 模块内部的计算精度在 30 位以上，因此计算的结果是比较精确的，完全能够满足大部分计算的需要。同时每一次浮点三角函数运算都可以在 34 个时钟周期之内完成，具有很高的实用价值。

iDeg_Rad_Sel 为 1 时表示输入角度的单位是弧度，输入弧度为 0.5869 时的硬件测试波形如图 17 所示：

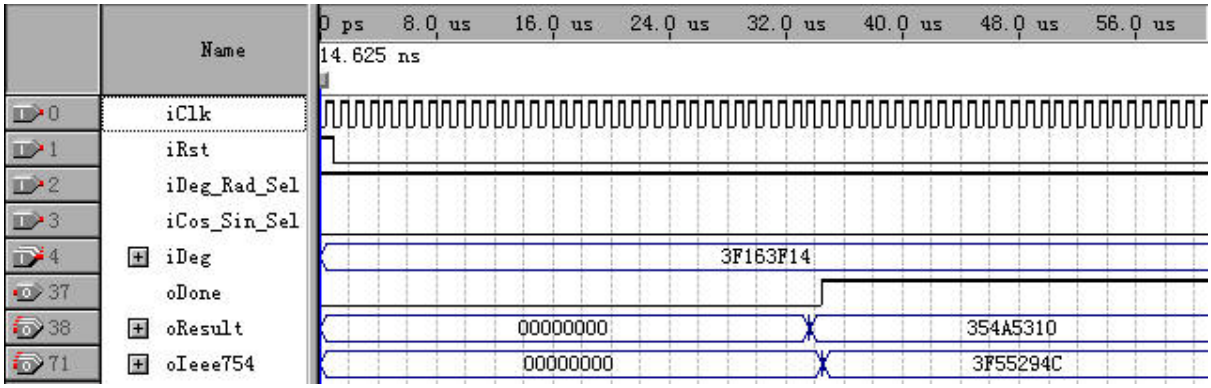

图 17　0.5869 弧度时的仿真波形

弧度 0.5869 的 IEEE-754 单精度表示为：0x3f163f14，通过计算可知：cos(0.5869) = 0.832661，实际硬件的计算结果为 0x3f55294c，它等于浮点数 0.832661。计算同样在单精度范围内不存在误差，而且计算仍然可以在 34 个时钟周期之内完成。

最后我们来进行输入弧度为负值且输入超过 2π 时的硬件工作情况测试。在输入数据时，不论角度的单位是度还是弧度，硬件都直接支持正数和负数角度的输入，具体的输入数据处理都是由统一的角度输入前处理单元来完成。当输入弧度值为：-20.8879 时的硬件测试波形如图 18 所示：

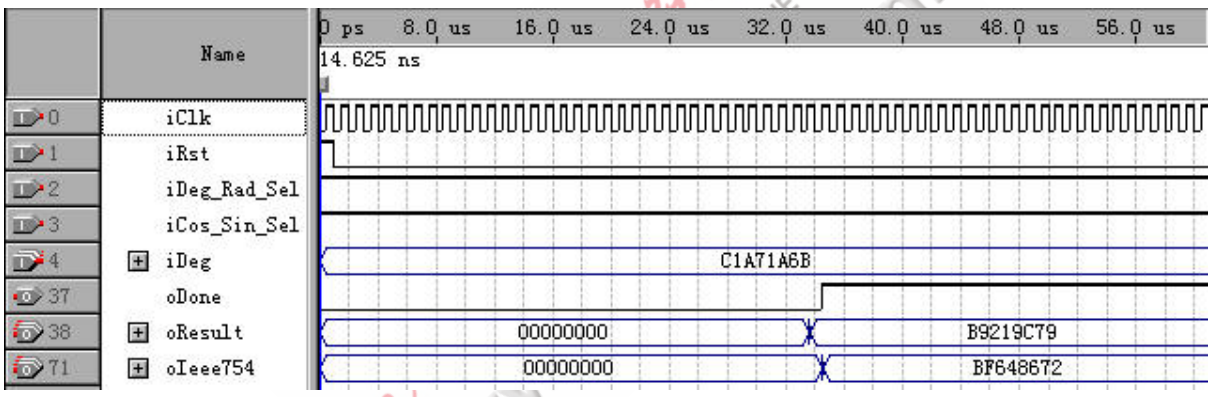

图 18　-20.8879 弧度时的仿真波形

弧度-20.8879 的 IEEE-754 单精度表示为：0xc1a71a6b，通过计算可知，在理想情况下：sin(-20.8879) = -0.892676，实际硬件的计算结果为 0xbf648672，它等于浮点数-0.892676。通过输入单位为弧度时的硬件测试，我们可以看出，本模块仍然提供了很高的计算精度，并且输入数据兼容性强，范围大，为实现与 CPU 的接口发挥了重要的作用。

7　浮点 CORDIC 计算模块与 Nios II 处理器的接口

Nios II 在 FPGA 上实现具有一系列不同于普通嵌入式 CPU 系统的特性，它的外设可以灵活选择或增加，可以自定制用户逻辑为外设，可以允许用户定制自己的指令集，设计者可以使用 Nios II 加上外部的 Flash、SRAM 来构成一个嵌入式处理器系统。

使用 Altera 公司的 Nios II 嵌入式处理器，设计师可以增加用户自定义指令到 Nios II 指令集中，加速软件算法中时间消耗最大的部分程序的执行。通过使用用户自定义指令，可以把一系列复杂的标准指令简化为单一的硬件执行指令。用户自定义指令的这些特性有着广泛的应用，例如：优化数字信号处理程序中的软件内部循环、数据包的帧头处理和计算密集型应用等等。通过 Quartus II 软件中的 SOPC Builder，可以向 Nios

II 处理器添加多达 256 条的用户自定义指令。用户自定义指令逻辑直接连接到 Nios II 处理器的算术逻辑单元（ALU）之上 ，如图 19 所示：

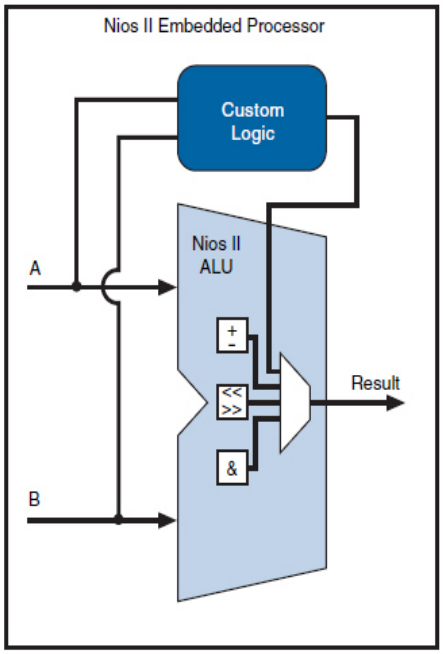

图 19　Nios II 处理器用户自定义指令

Nios II 处理器的用户自定义指令在数据路径上是与 ALU 相邻的硬件逻辑块，用户自定义指令使设计者能够通过定制处理器来满足实际应用的需要。用户自定义指令接口有四种：组合逻辑用户自定义指令接口、多周期用户自定义指令接口、扩展用户自定义指令接口和内部寄存器文件用户自定义指令接口。本设计使用的是扩展用户自定义指令接口。

扩展型用户自定义指令允许单一一条用户自定义指令执行几种不同的操作功能。扩展型用户自定义指令用一个 N 字段来指定逻辑模块是执行哪一个具体的操作，指令中的 N 字段最多可以达到 8bit，意味着单一一个用户逻辑模块最多可以实现 256 种不同的操作。图 20 是一个可以实现位交换、字节交换和半字节交换三种不同操作的扩展型用户自定义指令的框图。图 20 所示的硬件模块将 dataa 端口输入的操作数进行交换，它使用了 2bit 宽度的信号 n[1..0]来从多路选择器中选择输出信号。输入的选择信号 n[1..0]直接来自用户自定义指令的 N 字段。

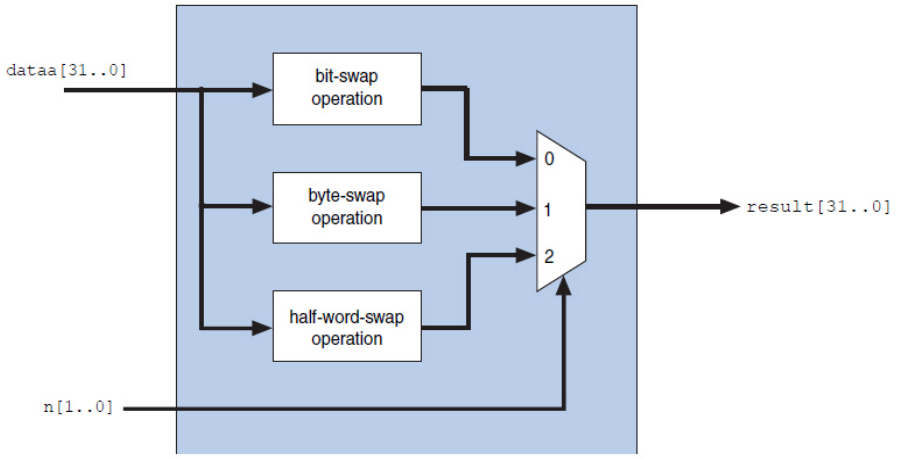

图 20　扩展型用户自定义指令

扩展型多周期用户自定义指令接口时序如图 21 所示：

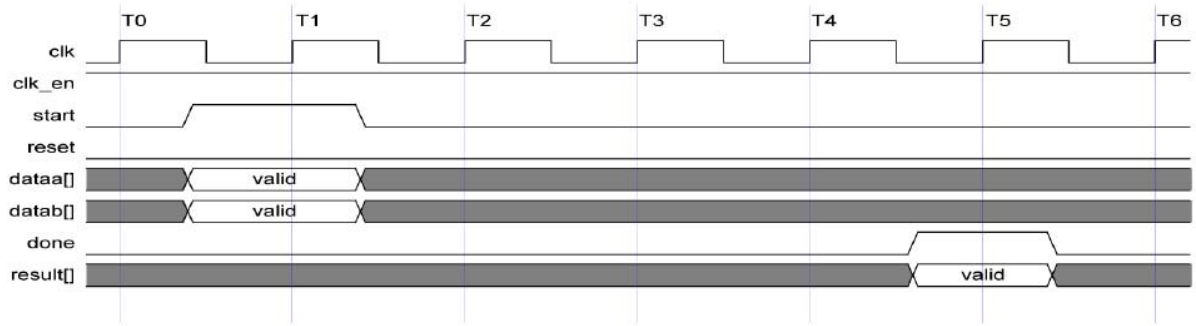

图 21　扩展型用户自定义指令接口时序

扩展型多周期用户自定义指令端口说明如下表：

端口名称	方向	必须	用途
clk	输入	是	系统时钟
clk_en	输入	是	时钟使能
reset	输入	是	同步复位
start	输入	否	指示用户指令逻辑开始执行
done	输出	否	用户指令逻辑指示处理器执行完毕
dataa[31..0]	输入	否	输入操作数
datab[31..0]	输入	否	输入操作数
result[31..0]	输出	否	输出结果

本论文设计的浮点 CORDIC 计算模块最终在 Nios II 处理器中完成了硬件实现，它作为 Nios II 处理器的一条硬件自定义指令可以很方便的由软件来调用，并且还提供了直接由 C/C++语言调用的宏。作为一条用户自定义指令，该浮点 CORDIC 计算模块与 Nios II 处理器接口采用的是扩展型多周期用户自定义指令接口形式。这是因为浮点 CORDIC 计算模块不是简单的组合逻辑电路，计算无法在一个时钟周期之内完成，所以不能采用组合逻辑用户自定义指令接口形式。同时由于 CORDIC 算法的特性，sin、cos 函数可以在一次计算中同时计算出来，为了节省硬件资源，提高硬件的利用率，本浮点 CORDIC 计算模块采用了扩展型用户自定义指令接口，并且本模块还直接支持两种不同单位的角度输入，这都是依靠扩展型用户自定义指令接口的特性来实现的。

浮点 CORDIC 计算模块的用户自定义指令接口的 HDL 语言描述程序如下：

```
module cordic_cpu_top
(
    clk,
    reset,
    clk_en,
    start,
    done,
    n,
    dataa,
    result,
);
```

```
input  clk;
input  reset;
input  clk_en;
input  start;
input  [1:0]  n;
input  [31:0] dataa;
output done;
output [31:0] result;

cordic_cpu u0 (
    .iClk(clk),
    .iRst(start),
    .iDeg(dataa),
    .iDeg_Rad_Sel(n[0]),
    .iCos_Sin_Sel(n[1]),
    .oDone(done),
    .oIEEE-754(result) );

endmodule
```

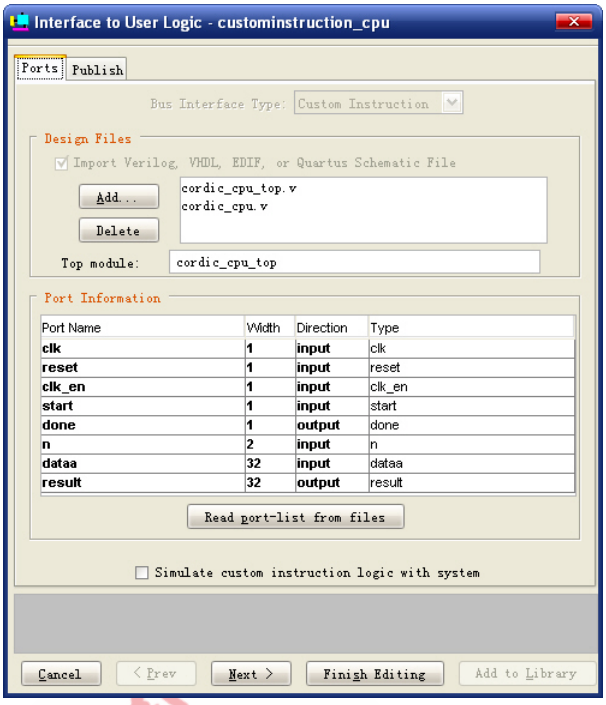

图 22　加入用户自定义指令

本浮点 CORDIC 计算模块采用扩展型用户自定义指令接口，一共支持四种输入输出计算模式，由参数 n 来进行选择，参数 n 一共有两位，其中 n[0]决定输入的角度单位是度还是弧度，n[1]决定输出的计算结果是正弦还是余弦。它们的计算模式组合如下表所示。

n[0]	n[1]	计算结果
0	0	计算输入角度单位为度的余弦函数
0	1	计算输入角度单位为度的正弦函数
1	0	计算输入角度单位为弧度的余弦函数
1	1	计算输入角度单位为弧度的正弦函数

start 信号直接接到的 cordic_cpu 模块的复位端口 iRst，用于每一次计算开始时的初始化。当 done 信号由低电平变为高电平时，处理器可以从 result 端口读出计算结果，计算结果直接使用 IEEE-754 单精度浮点数据格式。在 SOPC Builder 系统中以用户自定义指令的接口方式加入该模块。加入模块的过程主要是对接口类型进行匹配，如图 22 所示。

通过 Add...按钮加入本模块的 HDL 硬件语言描述文件和接口说明文件，并且以接口说明文件作为 top module。cordic_cpu.v 文件是浮点 CORDIC 计算模块的 HDL 硬件语言描述文件，cordic_cpu_top.v 文件是接口说明文件。用 read port-list from files 自动读入端口名称，并进行端口类型设置以后模块就基本上与 Nios II 系统连接完毕了。用户自定义指令加入完毕后 CPU 配置选项里的 Custom Instruction 栏如图 23 所示，从图中可以看出该模块的时钟周期为 Variable 类型， 对应多周期用户自定义指令，操作码扩展使用了 N[1:0]两个端口。

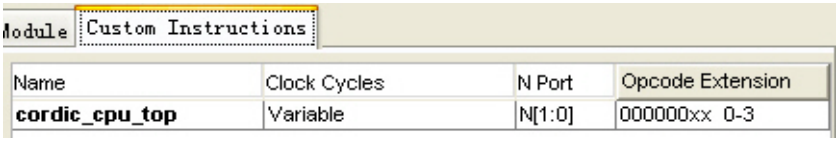

图 23　用户自定义指令

为了方便在C/C++语言中对用户自定义指令的调用，在Nios II IDE中可以生成直接调用用户自定义指令的宏：

```
#define COS_DEG(A) builtin_custom_fnf(ALT_CI_CORDIC_CPU_TOP_N+
(0&ALT_CI_CORDIC_CPU_TOP_N_MASK),(A))
#define COS_RAD(A) builtin_custom_fnf(ALT_CI_CORDIC_CPU_TOP_N+
(1& ALT_CI_CORDIC_CPU_TOP_N_MASK),(A))
#define SIN_DEG(A) builtin_custom_fnf(ALT_CI_CORDIC_CPU_TOP_N+
(2& ALT_CI_CORDIC_CPU_TOP_N_MASK),(A))
#define SIN_RAD(A) builtin_custom_fnf(ALT_CI_CORDIC_CPU_TOP_N+
(3& ALT_CI_CORDIC_CPU_TOP_N_MASK),(A))
```

只需要在C语言中直接使用COS_DEG()、COS_RAD()、SIN_DEG()、SIN_RAD()这四个宏调用就可以使用用户自定义硬件指令，来分别计算以度为单位和以弧度为单位的输入角度的正余弦函数。

验证计算结果的正确性和计算结果的精度主要是通过与软件仿真库的函数比较而得到的，C语言计算超越函数的软件函数声明主要在math.h文件中，只要包含math.h这个文件，我们就可以直接使用软件仿真库来计算超越函数。测量浮点CORDIC计算模块的计算速度是通过使用硬件定时器来实现的，我们特地将一个定时器指定为timestamp功能，用来测量一段程序执行所花费的时钟周期数，在本Nios II处理器系统中，使用的时钟周期频率是50MHz，使用的是经济型Nios II内核配置。

以角度作为输入单位时，本系统使用的C语言验证程序主要部分如下：

```
for(angle=0.0;angle<=360.0;angle+=10.0)
{
    time1  = alt_timestamp();
    result = COS_DEG(angle);
    time2  = alt_timestamp();
    printf(" COS_DEG(%f) = %f \n" ,angle,result)
    printf(" hardware use ticks : %l\n" ,time2-time1-time_overhead);

    time1  = alt_timestamp();
    result = cos(angle*Deg2Rad);
    time2  = alt_timestamp();
    printf(" cos(%f) = % \n" ,angle,result)
    printf(" software use ticks : %l\n" ,time2-time1-time_overhead);
}
```

以弧度作为输入单位时，本系统使用的C语言验证程序主要部分如下：

```
for(angle=0.0;angle<=2.0*pi;angle+=0.1)
{
    time1  = alt_timestamp();
    result = COS_RAD(angle);
    time2  = alt_timestamp();
    printf(" COS_RAD(%f) = %f \n" ,angle,result)
    printf(" hardware use ticks : %ld\n" ,time2-time1-time_overhead);

    time1  = alt_timestamp();
```

```
    result = cos(angle);
    time2  = alt_timestamp();
    printf(" cos(%f) = % \n",angle,result)
    printf(" software use ticks : %ld\n",time2-time1-time_overhead);
}
```

在这两段测试程序中，前半部分是直接使用硬件自定义指令来进行三角函数的计算，后半部分是使用软件仿真库函数来进行计算，并且分别测量出了计算所用的时钟周期数。因为使用软件仿真库函数来进行三角函数计算时，只支持角度单位为弧度的输入，所以在以度作为输入单位时，角度要乘以一个常数 Deg2Rad，该常数的值约为：0.0174533。

时间间隔的测量通过调用 alt_timestamp()函数来实现，将需要进行时间间隔测量的代码放置在两个 alt_timestamp()函数调用之间，就可以测出代码的执行时间。在程序中 timer_overhead 是一次空操作的时间，引入 timer_overhead 的目的是为了让时间间隔测量更加精确，它的数值更系统的配置和运行情况有关，测量 timer_overhead 的程序代码如下：

```
    time1 = alt_timestamp();
    time2 = alt_timestamp();
    timer_overhead = time2 - time1;
    printf(" tim_overhead = %ld\n", timer_overhead);
```

最后的程序运行结果可以直接从 Nios II IDE 的 console 窗口中得到，该窗口显示的是 Nios II 处理器运行时通过调用 Printf()函数发送过来的数据，它是对 Nios II 系统进行硬件调试的一个重要工具。当输入角度单位为度时的余弦函数计算结果如图 24 所示：

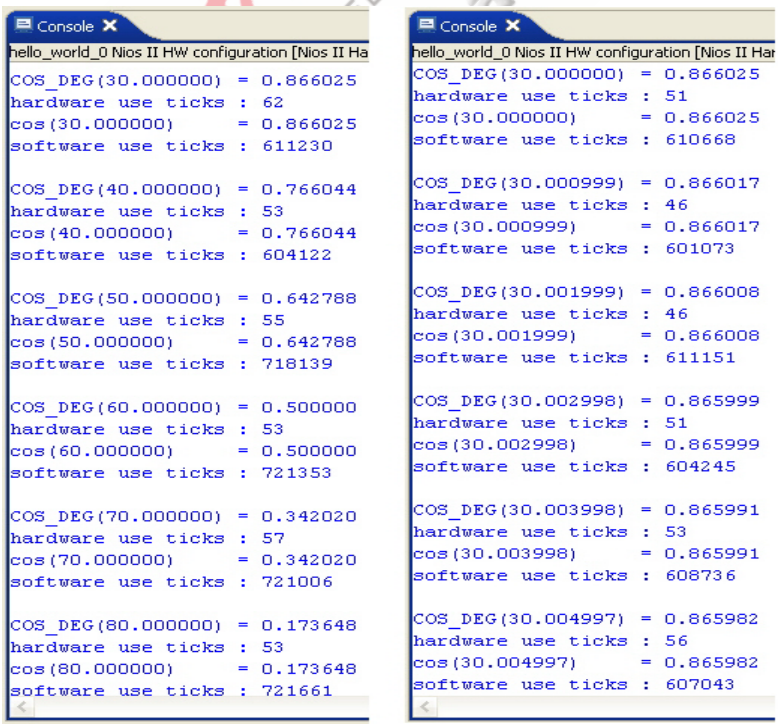

图 24　输入角度单位为度时的余弦函数计算结果

图 24 左窗口显示的是角度递增为 10 度时的计算结果，右窗口显示的是角度递增为

0.001 度时的计算结果。从中我们可以看出，在浮点单精度表示范围之内，也就是 6～7 位有效数字时，本论文设计的硬件浮点 CORDIC 计算模块与软件仿真算法的计算结果完全一致。并且输入输出均是 IEEE-754 单精度浮点数据格式，不需要做任何转换。

在计算速度方面硬件浮点 CORDIC 计算模块的优势更是明显。硬件计算的平均耗时在 50～60 个时钟周期，而软件仿真算法的平均耗时在 600000～700000 个时钟周期之间，硬件算法比软件快 12000 倍以上。并且软件算法随着计算输入角度数值的不同，所耗费的时钟周期数差别也很大，可以达到 100000 个时钟周期以上。相比较而言，硬件算法的时钟周期数在不同的输入角度时差别要小的多。

当输入角度单位为弧度时的正弦函数计算结果如图 25 所示：

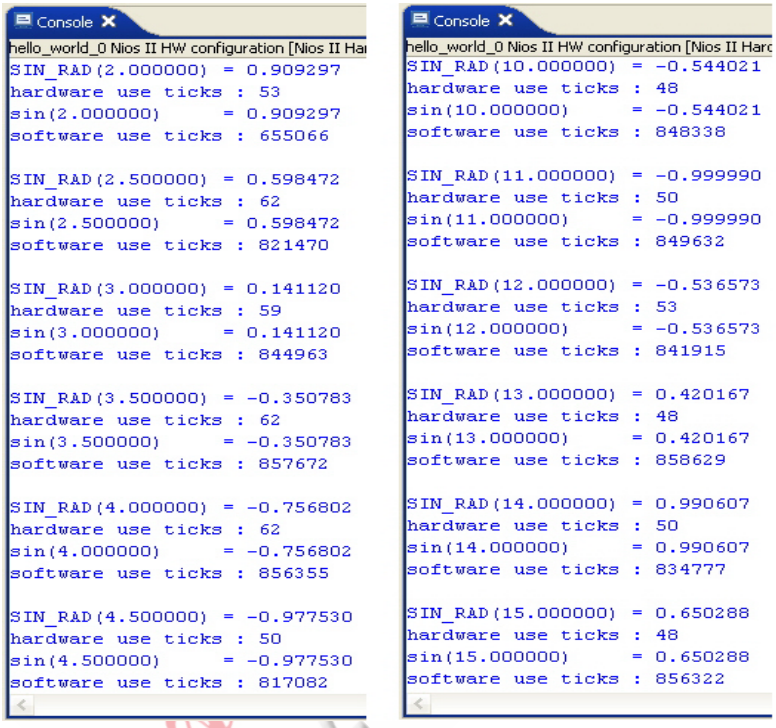

图 25　输入角度单位为弧度时的正弦函数计算结果

图 25 左窗口显示的是角度递增为 0.5 个弧度时的计算结果，右窗口显示的是角度递增为 1 个弧度时的计算结果。这幅图片还同时展示了硬件浮点 CORDIC 计算模块在大角度输入时的计算情况，因为当输入以弧度为单位时：$2\pi \approx 6.28$，右窗口中一些角度的输入已经大于一个圆周了。硬件计算的平均耗时也是在 50～60 个时钟周期，而软件仿真算法的平均耗时在 600000～800000 个时钟周期之间，这是因为对于大角度的输入而言，软件仿真算法需要进行一次取模运算，这就极大地增加了计算所花费的时钟周期数。而硬件浮点 CORDIC 计算模块在大角度输入时仍然能够有很快的计算速度，这取决于硬件模块设计上对于大角度浮点数据输入的良好支持。在以弧度为单位时的大角度输入三角函数计算过程中，硬件算法比软件平均快 17000 倍以上。

同时，在计算结果的精度上，硬件浮点 CORDIC 计算模块的结算结果与软件仿真算法的计算结果也是完全一致的。这也就是说，硬件算法在以弧度为单位大角度直接输入时仍然具有浮点单精度范围内没有误差的计算性能。这就极大地拓宽了硬件的应用范围和应用的领域。

最后一幅图显示的是以度为输入单位，并且输入角度范围更大时余弦函数的计算测试结果。

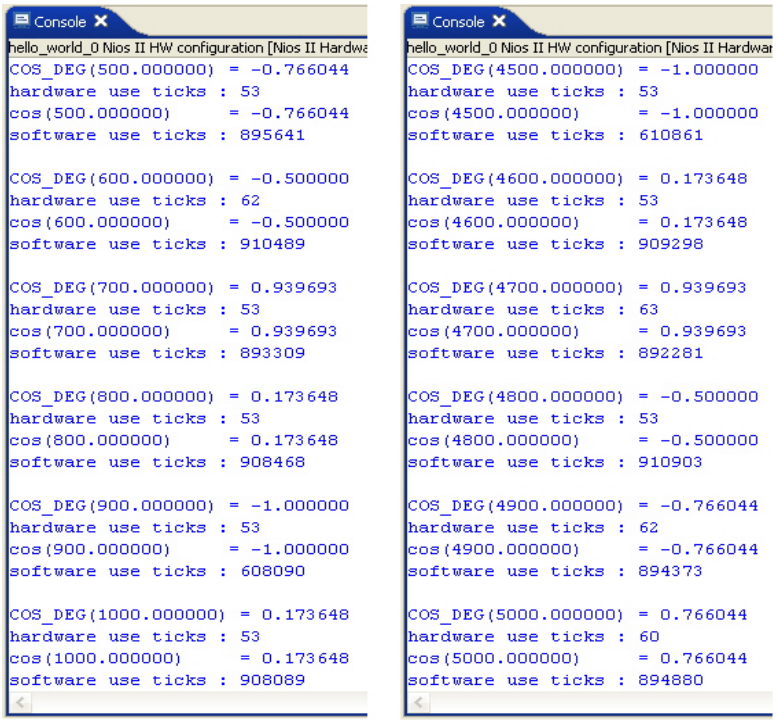

图 26　输入角度单位为度时的余弦函数计算结果

从图 6.14 中我们可以看出，硬件计算的平均耗时是在 50～60 个时钟周期，而软件仿真算法的平均耗时则达到了 800000～900000 个时钟周期，一方面这是由于软件仿真算法不直接支持以度为单位的计算；另一方面也是由于大角度输入时需要进行软件取模运算所致。

8　结束语

本论文设计的硬件浮点 CORDIC 计算模块通过与 Nios II 处理器系统进行连接，作为 Nios II 处理器的一条自定义指令，通过软件来对硬件进行了全面的测试，测试结果显示了硬件设计的正确性。其计算精度和计算速度完全满足原先设计的要求，并且本论文设计的硬件浮点 CORDIC 计算模块的特性在于：一是直接支持以度为单位和以弧度为单位的浮点角度输入；二是直接支持超过 360 度的大角度数据输入。今后可以在本论文工作的基础上，开展更高层次的基于 CORDIC 方面的算法研究和硬件实现。

参考文献

[1] Volder.The CORDIC trigonometric computing technique[J].IRE Trans,1959:334-334.

[2] Walther.J.S.A unified algorithm for elementary functions[J].Spring Joint Computer Conf,1971:379-385.

[3] Andraka.A survey of CORDIC algorithms for FPGA based computers[A].1998:191-200.

[4] 李全,陈石平,付佃华.用 FPGA 实现 CORDIC 算法的 32 位浮点三角超越函数之正余弦函数[J].电子产品世界，2006，217：150-151.

作者简介：
李　全 （1982－）男，桂林电子科技大学通信与信息工程系研究生，主要从事FPGA及嵌入式系统方面的研究。
李晓欢 （1982－）男，桂林电子科技大学通信与信息工程系研究生，主要从事FPGA通信方面的研究。
陈石平 （1981－）男，桂林电子科技大学通信与信息工程系研究生，主要从事FPGA通信方面的研究。

联系方式

联系人：　李全
地址：　广西桂林市府后里10号3楼2单元
邮编 ：　541001
电话 ：　13907731799
Email:　liquan_lq@163.com

原创性声明

我们声明所呈交的论文是在导师指导下进行的研究工作及取得的研究成果。尽我们所知，除了参考文献外，论文中不包含其他人已经发表或撰写过的研究成果。

李全　李晓欢　陈石平

2009-03-16