doi:10.11887/j.cn.202002005 http://journal.nudt.edu.cn

流水的浮点倒数近似值运算部件的设计与实现*

何 军,王 丽
(上海高性能集成电路设计中心,上海 201204)

摘 要:在部分低精度浮点运算应用中,需要流水的浮点倒数近似值运算。本文基于SRT-4算法设计并实现了一种流水的浮点倒数近似值运算部件。该部件采用6级流水线结构,运算结果精度至少为8位有效尾数。为了支持对非规格化浮点数的硬件处理,还设计并实现了改进版,有利于进一步提高浮点倒数近似值运算的性能。改进版采用8级流水线结构,新增了源操作数预规格化和结果后规格化功能模块,可以实现对非规格化浮点数的硬件处理。经过逻辑综合评估,改进版的硬件开销是面积在合理范围内增加19.23%,且对时序没有明显影响,可以满足预期的1.6 GHz频率设计目标。

关键词:浮点倒数;非规格化浮点数;流水
中图分类号:TP332.2 **文献标志码**:A **文章编号**:1001-2486(2020)02-041-06

Design and implementation of pipelined floating-point reciprocal approximation operation unit

HE Jun, *WANG Li*
(Shanghai High Performance Integrated Circuit Design Center, Shanghai 201204, China)

Abstract: In some low precision applications, pipelined floating-point reciprocal operation is required actually. Based on SRT-4 algorithm, a pipelined floating-point reciprocal operation unit was designed and implemented, which is constructed as a 6-stage pipeline unit, resulting in an 8-bit valid fractions. In order to support hardware process of denormal numbers, the unit was improved to get higher performance, which is constructed as a 8-stage pipeline unit, adding source operand pre-normalization and result post-normalization function components and supporting hardware process of denormal numbers. After logic synthesis, the area of the unit was increased by 19.23%, which is reasonable. The timing of the unit was not affected obviously and met the expected frequency goal of 1.6 GHz.

Keywords: floating-point reciprocal; denormal number; pipelined

浮点运算部件是微处理器的重要运算部件,与处理器的性能直接相关。常见的浮点运算包括浮点加、减、乘、乘加等。这些浮点运算在传统的科学计算和工程计算应用领域中应用十分广泛。相对于这些常见的浮点运算,浮点倒数运算并不很常用,但是在数字信号处理、多媒体、计算机图形计算等应用领域,以及部分科学计算应用领域,却比较常用,也是一种重要的运算[1]。此外,利用浮点倒数运算,还可以实现浮点除法运算。

常用的实现浮点倒数运算的算法与浮点除法类似,有基于减法运算的数字迭代算法[2]、SRT算法[3-4],也有基于乘法运算的Newton-Raphson算法和Goldschmidt算法[5-6]。基于这些算法,如果要实现浮点单精度或双精度浮点倒数运算,为了减少硬件开销,一般都是采用非流水的硬件迭代方法实现。其中基于减法的迭代算法是线性收敛的,每次可以迭代出1(基数为2)、2(基数为4)、3(基数为8)位结果,硬件开销小。基于乘法的迭代算法是二次收敛的,每次迭代后结果的精度翻倍,硬件开销大。

但在有的应用中,其实并不需要非流水的全精度的浮点倒数运算,而是能流水的部分精度(比如6~10位有效尾数)的浮点倒数近似值运算。再利用流水的快速浮点乘法运算,软件实现Newton-Raphson算法和Goldschmidt算法,就可以更高效地流水地实现任意精度的浮点除法运算等其他运算。这种方法尤其适用于低精度浮点运算应用。以单精度浮点(24位尾数,含隐含位)除法运算为例,目前硬件非流水实现,需要17拍。假设利用流水的浮点倒数近似值运算,得到8位精

* **收稿日期**:2019-09-30
基金项目:国家核高基重大专项资助项目(2018ZX01029101)
作者简介:何军(1980—),男,湖北汉川人,高级工程师,博士,E-mail:joyhejun@126.com

度尾数需要6拍,再加上一次乘法运算(6拍流水执行),如果低精度应用只需要8位精度即可,则一次除法只需要12拍,且可流水执行。假设有 n 次这样的除法运算,使用非流水硬件实现,需要 $17n$ 拍,利用流水的浮点倒数近似值运算加乘法运算实现,只需要 $12+n-1$ 拍。当 n 越大时,流水硬件实现延迟更短。

因此,综合考虑运算延迟和硬件开销,采用基数为4的SRT算法[7-9](简称"SRT-4算法")设计并实现了一种浮点倒数近似值运算部件(简称"FREC部件"),6级流水线结构,运算结果精度至少为8位有效尾数。为了硬件支持浮点非规格化浮点数[10-11],经过进一步改进,增加源操作数预规格化和结果后规格化功能,可以实现对浮点非规格化浮点数的硬件处理,有利于进一步提高浮点倒数近似值运算的性能[12]。

1　设计与实现

1.1　SRT-4算法

基本的SRT算法迭代过程采用数学公式描述如下(其中,X 为被除数,D 为除数,R 为算法的基数):

$$\begin{aligned} RP_0 &= X \\ P_{i+1} &= RP_i - q_{i+1}D \end{aligned} \tag{1}$$

这里 P_i 是第 i 次迭代后的部分余数,在每次迭代中,商值 q_{i+1} 由商值选择函数 δ 决定:$q_{i+1}=\delta(RP_i,D)$,经 k 次迭代后最后的商 Q 和余数 P_r(P_k 为 k 次迭代后的部分余数)分别为:

$$\begin{aligned} Q &= \sum_{i=0}^{k} q_i R^{-i} \\ Pr &= \begin{cases} P_k + D & P_k < 0 \\ P_k & \text{else} \end{cases} \end{aligned} \tag{2}$$

具体实现时,SRT算法步骤如下:

1)将部分余数的初始值设为被除数 X,在第 i 步,通过将除数 D 与当前部分余数比较,选择下一位商数 q_{i+1}(由商值选择函数 δ 决定);

2)按照递推式(1)计算出下一个部分余数 P_{i+1};

3)按照前两步,经 k 次迭代后,将所得的商数加权求和就得到最终的商 Q,见式(2)。

一般基数 $R=2^n$,每次迭代可以得到的 n 位数商数。需要迭代的次数 k 与需要得到的结果精度有关。当 $R=4$ 时,每次迭代可以得到2位商数;对于单精度浮点数,$k=12$,对于双精度浮点数,$k=27$。

商值选择函数 δ 是SRT算法的关键部分,一般采用查询表的方式实现,即根据部分余数和除数的高若干位查表得到商。对于SRT-4算法,商数集合为$\{-2,-1,0,+1,+2\}$,可以采用表1所示的查询表[13],利用部分余数的高7位和除数的高4位(由于规格化的除数整数部分默认为1,因此这里的高4位实际是尾数的高4位)查表,即可得到商数。

表1　SRT-4商值查询表[13]

Tab.1　The SRT-4 quotient look-up table[13]

七位部分余数	四位除数															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00~02	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0
03	+1	+1	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0	+0
04	+1	+1	+1	+1	+1	+1	+1	+1	+0	+0	+0	+0	+0	+0	+0	+0
05	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+0	+0
06~0B	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
0C	+2	+2	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
0D	+2	+2	+2	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
0E	+2	+2	+2	+2	+2	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
0F	+2	+2	+2	+2	+2	+2	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
10	+2	+2	+2	+2	+2	+2	+2	+2	+1	+1	+1	+1	+1	+1	+1	+1
11	+2	+2	+2	+2	+2	+2	+2	+2	+2	+1	+1	+1	+1	+1	+1	+1
12	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+1	+1	+1	+1	+1
13	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+1	+1	+1	+1
14	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+1	+1
15	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+1
16~3F	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2	+2

表 1(续)

七位部分余数	四位除数															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40 ~ 68	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
69	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-1
6A	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-1	-1
6B	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-1	-1	-1	-1
6C	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-1	-1	-1	-1	-1
6D	-2	-2	-2	-2	-2	-2	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1
6E	-2	-2	-2	-2	-2	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1	-1
6F	-2	-2	-2	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
70	-2	-2	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
71	-2	-2	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
72	-2	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
73 ~ 78	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
79	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-0	-0
7A	-1	-1	-1	-1	-1	-1	-1	-1	-0	-0	-0	-0	-0	-0	-0	-0
7B	-1	-1	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0
7C ~ 7F	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0	-0

为了减少延迟,部分余数一般采用保留进位(carry save)的冗余形式保存在两个寄存器 Carry 和 Sum 中,避免迭代过程中的进位加法运算。所以在查表前,需要一个普通的 7 位进位传递加法器(Carry Propagate Adder, CPA),得到实际的部分余数,然后再查表,得到商数,如图 1 所示。

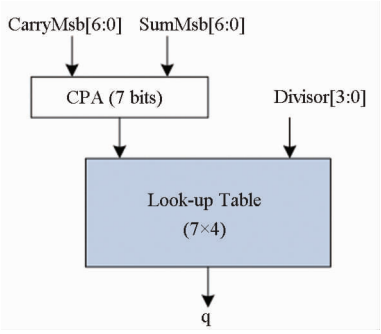

图 1　SRT - 4 商值选择函数的实现

Fig. 1　Implement of SRT - 4 quotient selection function

对于一次 SRT - 4 迭代来说,其实现原理如图 2 所示。查表得到的商数 q 采用 3 位编码表示。根据商数 q,选择得到多倍除数,并利用进位保留加法器(CSA3B2)得到更新后的部分余数 *Carry* 和 *Sum*;同时并行得到最新的商。这里商也采用了冗余形式保存在两个寄存器 Q 和 Qm 中,两者始终相差 1,即 $Qm = Q - 1$。利用飞速转换(on-the-fly)技术[14],每得到商数 q,就对当前的寄存器 Q 和 Qm 值进行更新,得到新的商。其转换表参见表 2,由 SRT4_QQM 模块实现。

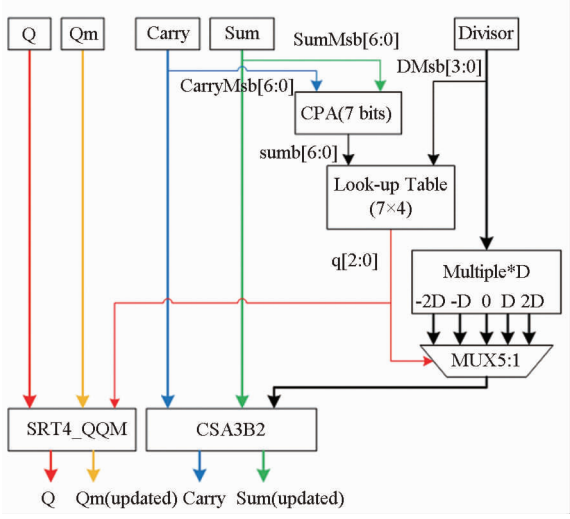

图 2　SRT - 4 算法迭代部分原理图

Fig. 2　Schematic diagram of iterative part of SRT - 4 algorithm

表 2　SRT - 4 飞速转换

Tab. 2　The SRT - 4 on-the-fly conversion

q	商数	$Q[9:0]$	$Qm[9:0]$
000	0	$\{Q[7:0],2'b00\}$	$\{Qm[7:0],2'b11\}$
001	+1	$\{Q[7:0],2'b01\}$	$\{Q[7:0],2'b00\}$
010	+2	$\{Q[7:0],2'b10\}$	$\{Q[7:0],2'b01\}$
101	-1	$\{Qm[7:0],2'b11\}$	$\{Qm[7:0],2'b10\}$
110	-1	$\{Qm[7:0],2'b10\}$	$\{Qm[7:0],2'b01\}$

注:1)对于浮点倒数近似值运算,只需要保留 10 位商即可。

2)$\{Q[7:0],2'b00\}$ 是按位拼接运算,2'b00 表示 2 位二进制数,下同。

1.2 FREC 部件结构

FREC 部件采用了 SRT－4 算法，为了得到至少 8 位有效尾数精度，考虑到首次商的最高位可能为 0，因此需要经过 5 次迭代，得到 10 位有效尾数。FREC 部件结构如图 3 所示。采用全流水实现，执行延迟为 6 拍，前 5 拍（ST0 ~ ST4）分别进行 5 次 SRT－4 迭代（流水进行），最后一拍（ST5）进行结果输出处理。与其他浮点运算一样，实际包括两条数据通路，分别进行输入异常数据和正常数据的处理，最后两条通路数据二选一输出（输入异常优先）。其中 F_REC_EXCEP 模块进行输入异常数据处理，主要进行无效操作（INV）、除数为零（DBZ）、非规格化浮点数（DNO）三种输入异常检测和处理。F_REC_EXP 模块主要进行阶码计算，并检测是否发生上、下溢出异常。

第一次迭代时，商值寄存器 Q、Qm 初始化为 0，部分余数寄存器 Carry 初始化为 0，部分余数寄存器 Sum 初始化为浮点数 1.0（即 0 × 3ff0_0000_0000_0000）规格化的尾数 *Sum_norm*，除数寄存器 Divisor 初始化为规格化的除数尾数 *Divisor_norm*，这里：

1）*Sum_norm* = {3′b000，1′b1，52′b0，1′b0}，即浮点数 1.0 的尾数；

2）*Divisor_norm* = {2′b00，1′b1，*Divisor*[51：0]，2′b00}。

结果输出处理主要完成如下功能：

1）利用普通加法器计算出实际的余数，根据余数判断是否发生非精确结果异常，并根据余数的符号决定是否需要恢复余数（再加上除数即可），并选择 *Q*、*Qm* 之一为最终的商。

2）根据商的最高位是否为 0，决定是否需要将商进行再规格化（左移 1 位），同时将结果的阶码减 1，得到正常数据通路的结果。

3）再跟输入异常处理通路的结果二选一输出（输入异常结果优先），作为最终的结果。

由于是求浮点倒数的近似值，为了简化硬件设计，仅支持向零舍入（即截断舍入）模式。

1.3 硬件支持非规格化浮点数的改进

根据 IEEE－754 标准[15]，规格化浮点数 $X=(s,e,f)$ 形式化表示如下：

$$X=(-1)^{s}\times 2^{e-bias}\times(1.f)$$

这里，$0<e<2^{bias}-1$，*bias* 为固定数，是浮点数据格式对应的阶码偏移值。

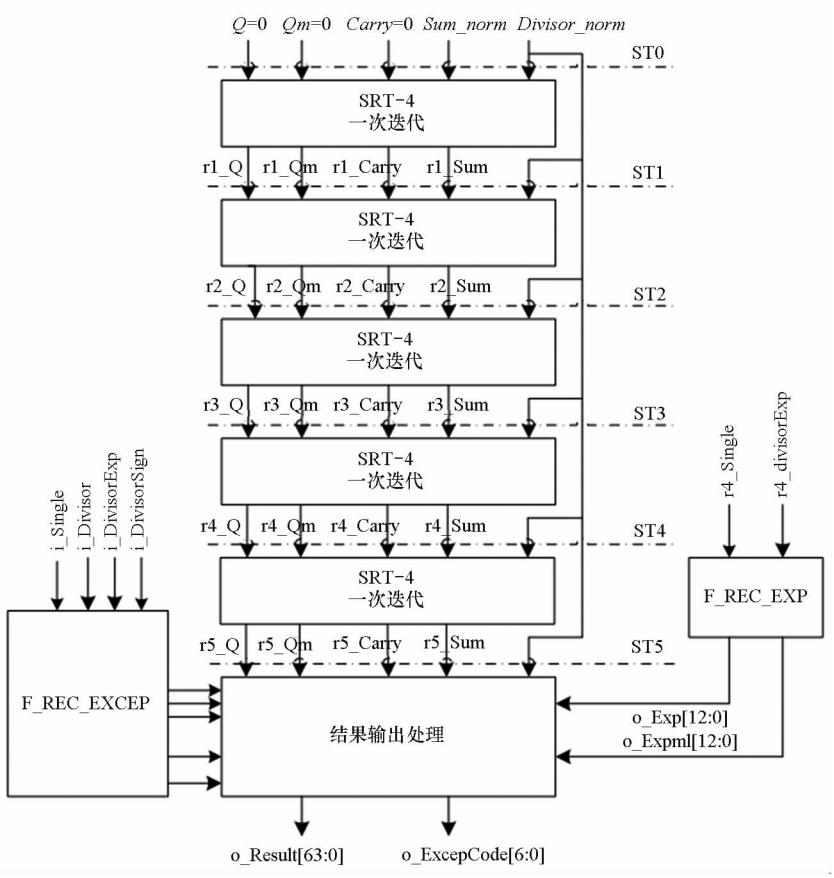

图 3 FREC 部件结构

Fig. 3 Implementation of FREC unit

非规格化浮点数 $X=(s,e,f)$ 形式化表示如下:

$$X=(-1)^s\times 2^{1-bias}\times(0.f)$$

这里,$e=0$,$f\neq 0$。两者可统一表示为:

$$X=(-1)^s\times 2^{e+\sim x_0-bias}\times(x_0+0.f)$$

这里 x_0 为尾数部分隐含整数位,对于规格化浮点数,其取值为 1;对于非规格化浮点数,其取值为 0。$\sim x_0$ 表示 x_0 取反。

一般浮点硬件仅支持规格化浮点数的运算,如果需要支持对非规格化浮点数的运算,需要进行特殊处理[10-11]:

1)对输入的非规格化浮点数要进行预规格化(pre-normalization)处理,检测尾数头 1 的位置,然后将尾数左移,直到最高位为 1,同时减少阶码;

2)当运算结果为非规格化浮点数时,需要进行后规格化(post-normalization)处理,常规的浮点运算需要对结果的尾数进行规格化左移,直到最高位是 1 为止。但是,如果发现此时的阶码出现了下溢(即小于最小阶码 $e_{\min}$)还需要再右移尾数,并增加阶码,直到阶码等于 $e_{\min}$。

1.3.1　源操作数预规格化

对非规格化浮点数的尾数(包括隐含的整数部分)进行头零检测(Leading Zero Detection,LZD),假定头零个数为 m(规格化数 $m=0$,非规格化数 $m>0$),将尾数左移 m 位,保证尾数最高位为 1。非规格化浮点操作数阶码为 0,而实际的阶码是 1,调整后的阶码为 $(1-m)$。对于规格化浮点数,不需要进行预规格化,直接进行运算即可(参见图 4)。

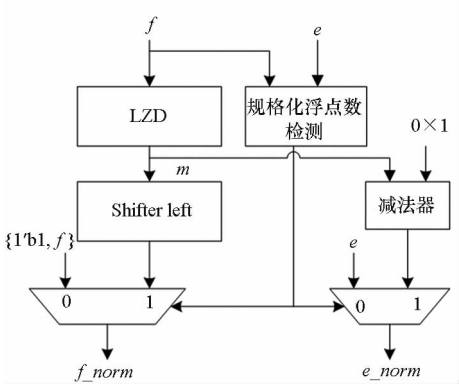

图 4　源操作数预规格化

Fig. 4　Source operand pre-normalization

1.3.2　结果后规格化

经过预规格化,非规格化浮点数的运算结果的尾数与规格化浮点数一样,头 0 的个数最多为 1,只需要规格化左移 1 位,同时将结果的阶码减 1,即可得到最终的尾数 f_r 和阶码 e_r。这里 e_r 可能小于等于 $e_{\min}$,即结果发生了下溢,可能是非规格化浮点数。

为了得到非规格化浮点数结果,需要对结果进行后规格化,即将结果的尾数右移,同时增加结果的阶码,直到阶码等于 $e_{\min}$。不妨设假定 $n=e_{\min}-e_r$($e_r\leqslant e_{\min}$),p 为浮点尾数有效位宽(对于单精度浮点数 $p=23$,对于双精度浮点数 $p=52$,对于这里倒数近似值运算来说,实际 $p=9$),反规格化右移位数 y 与结果尾数的精度 p 有关:

1)若 $n<(p+1)$,则 $y=n$(当 n 为 0 时无须右移);

2)若 $n\geqslant(p+1)$,则 $y=p+1$(此时如果再继续右移,实际上结果的尾数已经为 0,不必再右移了)。

当右移位数超出精度 $p+1$,导致移位后有效尾数为 0,此时阶码也应为 0,即最终结果为 0。当结果为非规格化浮点数时,也属于发生下溢,会报告下溢异常。结果后规格化功能的实现参见图 5。

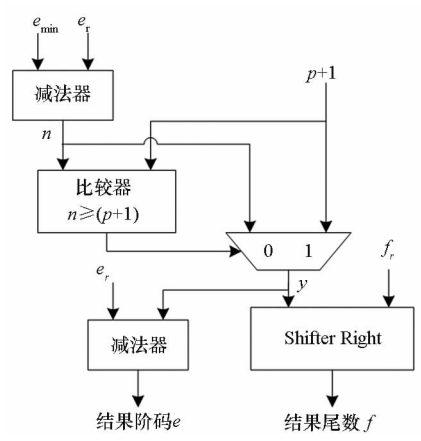

图 5　结果后规格化

Fig. 5　Result post-normalization

为了硬件支持非规格化浮点数处理,在 FREC 部件正常运算通路开始之前对源操作数进行预规格化(需要增加 1 拍),在结果输出处理模块之前,增加结果后规格化(也需要增加 1 拍),最终 FREC 部件新增 2 级流水线,执行延迟变为 8 拍。

此外,对输入异常数据处理模块 F_REC_EXCEP 也进行了适当修改,不再把非规格化浮点数直接当同符号的 0 进行处理。

2　正确性验证与逻辑综合

浮点倒数近似值运算的正确性验证,是以已经验证正确的浮点除法为参考模型,将被除数固

定为浮点数 1.0,除数与倒数运算的源操作数相同,然后截取除法结果的高 8 位有效尾数进行验证。源操作数采用浮点典型特殊值和随机值相结合的方法,进行大量模拟验证。此外,针对算法实现撰写了功能点,功能覆盖率可达到 100%。还进行了代码覆盖率分析,也可达到接近 100%(部分底层公共子模块存在覆盖不到的情况,也进行了确认)。结果表明,浮点倒数近似值运算部件的功能是正确的,结果的精度也符合预期。

利用 Synopsys 的 Design Compiler 工具,基于 16/14 nm 工艺条件,对 FREC 部件的 Verilog 设计代码进行了逻辑综合,包括硬件支持非规格化浮点数的改进版(FREC2)。逻辑综合的约束条件相同,时钟频率均为 1.6 GHz,两个版本均能满足频率设计目标,但是改进版有一定的面积和功耗开销,综合结果参见表 3。

表 3 逻辑综合结果

Tab. 3 Logic synthesis result

	面积/ μm^2	动态功耗/ mW	静态功耗/ μW
FREC	2348.11	13.44	7.89
FREC2	2799.56	15.25	9.34
增幅	19.23%	13.47%	18.38%

增加的面积主要为:源操作数预规格化中头零检测和左移移位器,结果后规格化中的右移移位器。增加的面积开销在合理范围内,主要是对时序没有影响,可以满足频率设计目标。

3 结论

基于 SRT - 4 算法,本文设计并实现了一种流水的 FREC 部件,6 级流水线,支持浮点倒数流水运算,结果精度至少 8 位有效尾数。基于该部件,利用流水的快速浮点乘法运算,软件基于 Newton-Raphson 算法和 Goldschmidt 算法,可以更高效地流水实现任意精度的浮点除法运算等其他运算。这种方法尤其适用于低精度浮点运算应用。

本文还设计并实现了改进版的 FREC 部件,增加源操作数预规格化和结果后规格化功能模块,可以实现对浮点非规格化浮点数的硬件处理,有利于进一步提高浮点倒数近似值运算的性能。改进版的 FREC 部件采用 8 级流水线结构,由于支持流水操作,增加了 2 级流水线对运算性能的影响很小。经过逻辑综合评估,改进版的 FREC 部件硬件开销是面积增加 19.23%。增加的面积开销在合理范围内,且对时序没有明显影响,可以满足预期的 1.6 GHz 频率设计目标。

参考文献(References)

[1] Pineiro J A, Bruguera J D. High-speed double-precision computation of reciprocal, division, square root, and inverse square root[J]. IEEE Transactions on Computers, 2002, 51(12): 1377 - 1388.

[2] Ercegovac M D, Lang T. Division and square root: digit recurrence algorithms and implementations[M]. MA, USA: Kluwer Academic Publishers, 1994.

[3] Harris D L, Oberman S, Horowitz M A. SRT division architectures and implementations[C]//Proceedings of the 13th IEEE International Symposium Computer Arithmetic, 1997: 18 - 25.

[4] Oberman S F, Flynn M J. Implementing division and other floating point operations: a system perspective[J]. Scientific Computing and Validated Numerics, 1970.

[5] Ercegovac M, Imbert L, Matula D W, et al. Improving goldschmidt division, square root and square root reciprocal[J]. IEEE Transactions Computers, 2000, 49(7): 759 - 763.

[6] Ercegovac M, Lang D, Muller J M, et al. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers[J]. IEEE Transactions Computers, 2000, 49(7): 628 - 637.

[7] Oberman S F. Floating point division and square root algorithms and implementation in the AMD - K7 microprocessor[C]//Proceedings of the 14th Symposium Computer Arithmetic, 1999: 106 - 115.

[8] Burgess N, Hinds C N. Design issues in Radix - 4 SRT square root and divide unit[C]//Proceedings of the 35th Asilomar Conference on Signals, Systems and Computers, 2001: 1646 - 1650.

[9] Burgess N, Hinds C N. Design of the ARM VFP - 11 divide and square root synthesisable macrocell[C]// Proceedings of the 18th IEEE Symposium on Computer Arithmetic, Montpellier, France, 2007: 87 - 94.

[10] Schwarz E M, Schmookler M S, Trong S D. Hardware implementations of denormalized numbers[C] // Proceedings of the 16th Symposium Computer Arithmetic, 2003: 70 - 78.

[11] Schwarz E M, Schmookler M, Trong S D. FPU implementations with denormalized numbers[J]. IEEE Transactions Computers, 2005, 54(7): 825 - 836.

[12] Lawlor O, Govind H, Dooley I, et al. Performance degradation in the presence of subnormal floating-point values[C]//Proceedings of the International Workshop on Operating System Interference in High Performance Applications, 2005.

[13] 付仲满. X 微处理器 FPU 的设计与实现[D]. 长沙: 国防科技大学, 2005.
FU Zhongman. Design and implementation of FPU for X Microprocessor[D]. Changsha: National University of Defense Technology, 2005. (in Chinese)

[14] Ercegovac M D, Lang T. On-the-fly conversion of redundant into conventional representations[J]. IEEE Transactions on Computers, 1987, 36(7): 895 - 897.

[15] ANSI/IEEE Standard 754 - 1985: IEEE standard for binary floating-point arithmetic[S/OL]. [2019 - 06 - 30]. https://standards.ieee.org/standard/754 - 1985.html