Floating-Point Division Algorithms for an x86 Microprocessor with a Rectangular Multiplier

Michael J. Schulte
University of Wisconsin
Schulte@engr.wisc.edu

Dimitri Tan
Advanced Micro Devices
Dimitri.Tan@amd.com

Carl E. Lemonds
Advanced Micro Devices
Carl.Lemonds@amd.com

Abstract

Floating-point division is an important operation in scientific computing and multimedia applications. This paper presents and compares two division algorithms for an x86 microprocessor, which utilizes a rectangular multiplier that is optimized for multimedia applications. The proposed division algorithms are based on Goldschmidt's division algorithm and provide correctly rounded results for IEEE 754 single, double, and extended precision floating-point numbers. Compared to a previous Goldschmidt division algorithm, the fastest proposed algorithm requires 25% to 37% fewer cycles, while utilizing a multiplier that is roughly 2.5 times smaller.

1. Introduction

In an x86 microprocessor, the floating-point unit (FPU) has undergone considerable change in recent years. Much of this change is due to the advent of Streaming SIMD Extensions (SSE) [1]. These extensions, mainly driven by multimedia applications (3D graphics, video, etc.), have added complexity to recent FPU designs. Prior to the addition of SSE, the FPU in x86 microprocessors only had to support x87 scientific floating-point instructions. In x87 mode, the FPU performs arithmetic operations on 80-bit extended-precision floating-point numbers, and then rounds the results to 32-bit single, 64-bit double, or 80-bit extended precision floating-point numbers [2]. Floating-point arithmetic in x86 microprocessors complies with the specifications given in the IEEE-754 Standard for Binary Floating-Point Arithmetic [3].

With the growing importance of multimedia applications, the FPU is now required to support both x87 instructions and SSE instructions. In 1999, Intel introduced SSE instructions that perform multiple floating-point arithmetic operations on single-precision floating-point data types [1]. For example, a single SSE instruction, DIVPS, performs four single-precision floating-point divide operations. A few years later, SSE2 introduced new instructions for parallel double-precision operations. Recently, SSE3 added horizontal arithmetic and asymmetric arithmetic operations, but no new data formats. Multimedia applications are placing a greater emphasis on SSE performance over x87. Hence, the FPU workload is shifting from engineering and scientific computing to multimedia applications.

We are designing an FPU that utilizes a 27-bit by 76-bit rectangular multiplier, in which the length of the multiplier operand is less than the length of the multiplicand operand. This reduces the area of the multiplier, but requires multiple passes through the multiplier to produce a full-precision result.

Our multiplier is optimized for single-precision SSE instructions, which are widely used in multimedia applications [1, 4]. The multiplier can perform two parallel single-precision multiplies each cycle with a latency of two cycles. It can perform one double-precision multiply every other cycle with a latency of three cycles or one extended-precision multiply every three cycles with a latency of four cycles. Compared to a fully-pipelined multiplier, the rectangular multiplier improves the latency of single precision multiplies and reduces the area of the FPU. It also has the potential to reduce power dissipation for multimedia applications. In addition to performing multiplication, the rectangular multiplier is used to perform division, square root, and elementary function computations.

Due to its importance in scientific computing and multimedia applications, several algorithms for floating-point division have been developed [5]. These algorithms can be divided into three main categories; digit recurrence, very high-radix, and functional iteration. Digit recurrence algorithms, such as restoring division, non-restoring division, and SRT division, compute a fixed number of quotient bits each iteration [6]. Very high-radix division algorithms, including accurate quotient approximations [7], the short reciprocal algorithm [8, 9, 10], and prescaling

1-4244-1258-7/07/$25.00 ©2007 IEEE

and selection by rounding algorithms [11, 12], are digit recurrence algorithms that compute a large number of quotient bits (e.g., 8 or more) each iteration. Functional iteration algorithms, such as Goldschmidt's algorithm [13] and Newton-Raphson iteration [14], typically obtain an estimate of the divisor's reciprocal, and then use multiplication and subtraction to double the number of accurate quotient bits each iteration.

In this paper, we present and compare two division algorithms for an x86 microprocessor with a rectangular multiplier. These algorithms are based on Goldschmidt's division algorithm and provide support for single, double, and extended precision floating-point numbers. The algorithms are also compared to the algorithm and implementation used on the AMD-K7 FPU [15], which employ Goldschmidt's algorithm to perform division, but uses a fully pipelined multiplier.

Some of our goals in developing these algorithms include (1) the algorithms should have a small impact on the architecture and performance of the multiplier, (2) they should be able to efficiently utilize the rectangular multiplier and high-speed reciprocal approximations, (3) they should have low latencies and not require unnecessary passes through the rectangular multiplier, (4) they should be optimized for single-precision numbers, but also be able to efficiently support double and extended-precision numbers, and (5) they should produce correctly rounded results, as specified in the IEEE 754 Standard for Binary Floating-Point Arithmetic.

The main contribution of this paper is the presentation of two new division algorithms that are designed to be implemented with a rectangular multiplier and provide support for x87 and SSE datatypes. The algorithms presented in this paper are based on Goldschmidt's division algorithm and are able to utilize the rectangular multiplier and high-speed reciprocal approximations. Our algorithms have low latencies, especially for single-precision numbers. Compared to very high-radix algorithms, our algorithms require fewer modifications to the multiplier architecture. They have lower latencies than equivalent Newton-Raphson-based division algorithms, since there are fewer dependencies between multiplications.

The remainder of this paper is organized as follows: Section 2 gives an overview of Goldschmidt's division algorithm. Section 3 presents the design of a 27-bit by 76-bit rectangular multiplier that provides high-performance single-precision multiplications and is extended to implement the proposed division algorithms. Section 4 discusses a previous implementation of Goldschmidt's division algorithm on the AMD-K7 FPU, and describes our proposed division algorithms. Section 5 compares the division algorithms, and Section 6 gives our conclusions.

In the following sections, upper case variables denote operands and lower-case variables denote bits within those operands. Individual bits are indexed by their bit position with the more significant bits having lower indices. For example, $X = x_0.x_1 \ldots x_{n-1}$ has the value:

$$VX = \sum_{i=0}^{n-1} x_i 2^{-i}$$

When bits i through j of X are accessed, we use the notation $X[i:j]$, where $X[i:j] = x_i x_{i+1} \ldots x_{j-1} x_j$ for $i < j$.

2. Goldschmidt's division algorithm

Goldschmidt's division algorithm is also known as division by multiplicative normalization, division by convergence, and division by series expansion. It has been implemented in the IBM 360/91 [16], the TMS390C602A [17], the IBM S/390 G4 [18], and the AMD-K7 microprocessor [15]. Various publications describe Goldschmidt's division algorithm [19, 20, 21], its error analysis [22], and its implementation using pipelined multipliers [23, 24].

Goldschmidt's division algorithm, computes the quotient $Q = A/B$ by starting with an initial approximation to the divisor's reciprocal; $X_0 \approx 1/B$. It then multiplies X_0 by the dividend, A, and divisor, B, to obtain:

$$N_0 = X_0 \times A \qquad (1)$$

$$D_0 = X_0 \times B \qquad (2)$$

$$R_0 = 2 - D_0 \qquad (3)$$

After this, m iterations are performed, where:

$$N_{i+1} = R_i \times N_i \qquad (4)$$

$$D_{i+1} = R_i \times D_i \qquad (5)$$

$$R_{i+1} = 2 - D_{i+1} \qquad (6)$$

Finally, N_m is multiplied by R_m to obtain Q. Each iteration requires two multiplications and one subtraction (or complement operation) and approximately doubles the number of accurate bits.

If X_0 has an absolute error of ε_{X_0} and computations are performed without rounding error then:

$$N_0 = X_0 \times A = \left(\frac{1}{B} + \varepsilon_{X0}\right) \times A = \frac{A}{B} + A\,\varepsilon_{X0} \qquad (7)$$

$$D_0 = X_0 \times B = \left(\frac{1}{B} + \varepsilon_{X0}\right) \times B = 1 + B\varepsilon_{X0} \qquad (8)$$

$$R_0 = 2 - D_0 = 2 - (1 + \varepsilon_{X0}) = 1 - B\varepsilon_{X0} \qquad (9)$$

In the next iteration:

$$N_1 = R_0 \times N_0 = \left(1 - B\varepsilon_{X0}\right) \times \left(\frac{A}{B} + A\varepsilon_{X0}\right) = \frac{A}{B} - AB\varepsilon^2_{x0} \quad (10)$$

$$D_1 = R_0 \times D_0 = \left(1 - B\varepsilon_{X0}\right) \times (1 + B\varepsilon_{X0}) = 1 - B^2\varepsilon^2_{x0} \quad (11)$$

$$R_1 = 2 - D_1 = 2 - (1 - B^2\varepsilon^2_{x0}) = 1 + B^2\varepsilon^2_{x0} \quad (12)$$

In general, when N_i is close to A/B, D_{i+1} *and* R_{i+1} converge towards 1.0 and N_{i+1} converges towards A/B. Each iteration roughly doubles the number of accurate bits in the quotient approximation, N_i.

Since R_i is close to 1.0, not all of the bits of R_i are needed to compute N_i and D_i. If $0 \le \varepsilon_{R_i} < 2^{-k}$, R_i has the form $1.0\ldots0r_{k+1}r_{k+2} \ldots r_{n-1}$. If $-2^{-k} \le \varepsilon_{R_i} < 0$, R_i has the form $0.1\ldots1r_{k+1}r_{k+2} \ldots r_{n-1}$. Consequently, the k most significant bits of R_i are not needed when computing N_i and D_i. Using the substitution $R'_i = R_i - 1$, Equations (4) to (6) can be rewritten as:

$$N_{i+1} = N_i + R'_i \times N_i \quad (13)$$

$$D_{i+1} = D_i + R'_i \times D_i \quad (14)$$

$$R'_{i+1} = 1 - D_{i+1} \quad (15)$$

Although this approach requires extra additions to implement Equation (13) and (14), it has the advantage that R'_i is close to zero, which lets $R'_i \times N_i$ and $R'_i \times D_i$ be computed with less precision. Instead of computing $R'_{i+1} = 1 - D_{i+1}$ directly, hardware computes R_i as the one's complement of D_{i+1} and then computes:

$$N_{i+1} = N_i + N_i \times R_i[k:j] \times 2^{-k}$$
$$= N_i + \{0'k, N_i\} \times R_i[k:j] \quad (16)$$

$$D_{i+1} = D_i + D_i \times R_i[k:j] \times 2^{-k}$$
$$= D_i + \{0'k, D_i\} \times R_i[k:j] \quad (17)$$

These computations multiply the appropriate bits from R_i by N_i or D_i right shifted by k bits and then adds this product to N_i or D_i, respectively.

3. Rectangular multiplier

The rectangular floating-point multiplier used to implement our proposed division algorithms has two pipeline stages, as shown in Figure 1. The first stage, EX1, consists of a 27-bit by 76-bit tree multiplier that accepts the two numbers to be multiplied, along with a 76-bit feedback term in carry-save format, and produces a 103-bit product in carry-save format. The second stage, EX2, consists of combined addition and rounding, result multiplexing, and forwarding to the register file and bypass networks.

The multiplier supports a range of precisions with wider precision multiplies achieved by multiple passes through the first stage, EX1. It supports operations on single precision numbers with 24-bit significands, double precision numbers with 53-bit significands, and extended precision numbers with 64-bit significands. Similar to the AMD-K7 multiplier design [15], our multiplier also provides a variety of other multiplication sizes to facilitate accurate division, square root, and elementary function computations. The multiplication sizes supported include 24x24, 25x24, 27x76, 53x53, 54x53, 54x76, 64x64, 68x68 and 76x76. The multiplier also performs two single precision (dual 24x24) multiplications in parallel, which is frequently used in multimedia applications.

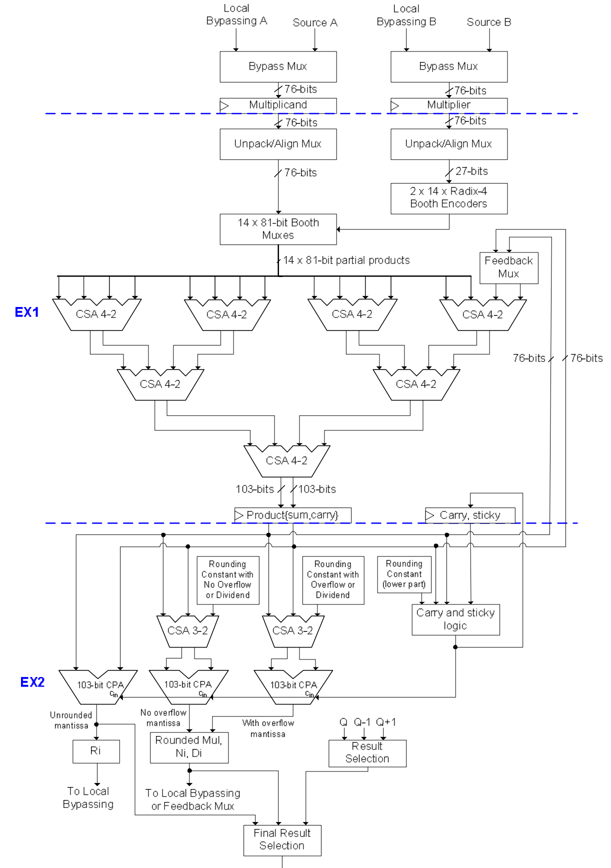

Figure 1. 27-bit by 76-bit multiplier

For each pass through the multiplier, the appropriate 27-bits of the multiplier operand are selected by the Unpack/Align Multiplexers. Two sets of radix-4 Booth encoders are required to support the dual 24x24 multiply. The Booth multiplexers produce fourteen 81-bit partial products, which are reduced, along with the two 76-bit feedback terms, using a partial product reduction tree implemented using three levels of 4-2 compressors. For the first pass, the feedback terms are all zeros. For subsequent passes, the feedback terms

are obtained from the upper 76-bits of the carry-save product from the previous pass.

The rounding scheme implemented in the second stage, EX2, involves adding rounding constants to the carry-save product using 3-2 carry-save adders (CSAs) prior to the final addition [15]. The rounding is performed prior to normalization using two additions, with one addition assuming rounding overflow occurs and one addition assuming rounding overflow does not occur. A third addition computes the un-rounded significand [15]. An appropriate rounding constant is provided for each of the first two additions and is omitted for the un-rounded significand. Since for wider precision multiplies, the product generation is split over multiple cycles, the lower 27-bits are processed after each pass to compute the sticky bit and the carry-in for the next pass. Table 1 shows the multiplier passes, latencies, and throughputs for supported multiplication sizes.

Table 1. Multiplier passes, latencies, and throughput for supported multiplication sizes

Multiplication Sizes	Multiplier Passes	Latencies (cycles)	Throughputs (mults/cycle)
Dual 24x24	1	2	2
24x24, 25x24, 27x76	1	2	1
53x53, 54x53, 54x76	2	3	1/2
64x64, 68x68, 76x76	3	4	1/3

4. Floating-point division algorithms

The division algorithms presented in this paper are derived from the AMD-K7 Goldschmidt division algorithm [15], which was designed for a fully-pipelined 76-bit by 76-bit multiplier. This section gives an overview of the AMD-K7 division algorithm [15]. It then presents our variations of Goldschmidt's division algorithm that are designed for an x86 microprocessor with the 76-bit by 27-bit rectangular multiplier presented in Section 3. The algorithms can be modified for other multiplier sizes.

Figure 2 shows the version of Goldschmidt's division algorithm implemented on the AMD-K7 and presented in [15]. This division algorithm only supports extended precision input operands with results rounded to single, double, extended, or internal precision. In Figure 2, A and B are the input operands. PC is the significand precision control, where PC is 24 for single precision, 53 for double precision, 64 for extend precision, and 68 for internal precision. Division with an internal precision of 68 bits is used to compute certain elementary functions. RC is the rounding control, which indicates if the final result is rounded to nearest even, toward zero, toward minus infinity, or toward plus infinity. Q_i is the initial quotient approximation and Q_f is the final correctly rounded quotient. REM is a 2-bit variable that indicates the sign of the remainder and if the remainder is zero. The cycles shown on the right assume that the initial reciprocal estimate takes three cycles and each multiplication takes four cycles [15]. The division algorithm takes 16 cycles for single precision (PC = 24), 20 cycles for double precision (PC = 53), and 24 cycles for extended and internal precision (PC = 64 and 68, respectively).

Program: *Goldschmidt's Division Algorithm in the AMD-K7 with a 76 by 76 Multiplier [15]*

Input = (A, B, PC, RC), Output = (Q_f)

Operations	**Cycles**
X_0 = recip_estimate(B)	1-3
D_0 = itermul_76x76(X_0, B), R_0 = comp1(D_0)	4-7
N_0 = itermul_76x76(X_0, A)	5-8
if (PC == 24)	
{N_f = N_0, R_f = R_0, goto END DIVISION }	
D_1 = itermul_76x76(D_0, R_0), R_1 = comp1(D_1)	8-11
N_1 = itermul_76x76(N_0, R_0)	9-12
if (PC == 53)	
{N_f = R_1, R_f = R_1, goto END DIVISION }	
D_2 = itermmul_76x76(D_1, R_1), R_2 = comp1(D_2)	12-15
N_2 = itermmul_76x76 (N_1, R_1)	13-16
R_f = N_2, R_f = R_2	
END DIVISION:	
Q_i = lastmul_76x76(N_f, R_f, $PC+1$)	See +
REM = backmul_76x76(Q_i, B, A),	
Q_f = round(Q_i, REM, PC, RC)	See *

+ 9-12 (PC = 24), 13-16 (PC = 53), 17-20 (PC = 64/68)
* 13-16 (PC = 24), 17-20 (PC = 53), 21-24 (PC = 64/68)

Figure 2: Goldschmidt's algorithm in the AMD-K7

The algorithm shown in Figure 2 includes several operations, which are discussed in detail by Oberman [15]. The *recip_estimate* operation uses 2^{10}-entry by 16-bit and 2^{10}-entry by 7-bit bipartite tables to provide a reciprocal estimate that is accurate to at least 14.94 bits [15, 25]. The *itermul_76x76* operation corresponds to a 76-bit by 76-bit multiply in which the result is rounded to 76 bits using round-to-nearest-even. The *comp1* operation produces the one's complement of D_i, which is a 76-bit value. The *lastmul_76x76* operation is a 76-bit by 76-bit multiply, which rounds its result to PC+1 bits of precision using round-to-nearest-even. PC+1 bits of precision are required in order to implement the AMD-K7 rounding technique [15]. The *backmul_76x76* operation performs a 76-bit by 76-bit multiplication of $Q_i \times B$ and subtracts A to determine the sign of the remainder and if the remainder is equal to zero. The *round* operation produces the correctly

rounded quotient using the AMD-K7 rounding technique [15].

To more efficiently implement Goldschmidt's division algorithm with a rectangular multiplier, our first version of Goldschmidt's algorithm (GS-1) uses a truncated version of R_i, in which the required precision of R_i is determined from a detailed error analysis. This analysis indicates correctly rounded results are still produced, when R_0 is truncated to 30 bits and R_1 is truncated to 60 bits. Since R_0 must be longer than 27 bits, it needs two passes through the 27-bit by 76-bit multiplier, so R_0 is instead truncated to 54 bits. Similarly, since R_1 is longer than 54 bits, it needs three passes through the multiplier, so all 76 bits are used.

Program: *Goldschmidt's Division Algorithm with Truncated R_i on a 27 x 76 Multiplier (GS-1)*

Input = (A, B, OT, PC, RC) Output = (Q_f)

Operations	Cycles
X_0 = recip_estimate(B)	1-3
D_0 = itermul_27x76(X_0, B), R_0 = comp1(D_0)	4-5
N_0 = itermul_27x76(X_0, A)	5-6
if (OT = = SINGLE) {	
Q_i = lastmul_54x76(R_0[0:53], N_0, *25*)	7-9
REM = backmul_25x24(Q_i, B, A),	
Q_f = round(Q_i, REM, *24*, RC)	10-11
goto END DIVISION }	
if (OT = = X87 and PC = = 24) goto X87 DIV	
D_1 = itermul_54x76(R_0[0:53], D_0),	
R_1 = comp1(D_1)	6-8
N_1 = itermul_54x76(R_0[0:53], N_0)	8-10
if (OT = = DOUBLE) {	
Q_i = lastmul_76x76(R_1, N_1, *54*)	11-14
REM = backmul_54x53(Q_i, B, A),	
Q_f = round(Q_i, REM, *53*, RC)	15-17
goto END DIVISION }	
X87 DIV:	
if (PC == 24) {	
Q_i = lastmul_54x76(R_0[0:53], N_0, *25*)	7-9
else if (PC == 53)	
Q_i = lastmul_76x76(R_1, N_1, *54*)	11-14
else {	
D_2 = itermul_76x76(R_1, D_1),	
R_2 = comp1(D_2)	11-14
N_2 = itermul_76x76(R_1, N_1)	14-17
Q_i = lastmul_76x76(R_2, N_2, $PC+1$) }	18-21
REM = backmul_76x76(Q_i, B, A),	
Q_f = round(Q_i, REM, PC, RC)	See *
END DIVISION:	

* 10-13 (PC=24), 15-18 (PC=53), 22-25 (PC = 64/68)

Figure 3: Goldschmidt's algorithm with truncated R_i on a 27 x 76 multiplier (GS-1)

Utilizing a truncated version of R_0 allows some of the multiplications to be performed with fewer passes through the rectangular multiplier. The GS-1 algorithm also examines the operand type, OT, since SSE requires support for single and double precision input operands and operations on these types of operands require fewer passes through the rectangular multiplier than extended precision operands.

Figure 3 shows the GS-1 Algorithm. In this figure, the size of each multiplication is specified by the numbers after the "_". All of the *itermul_* operations, truncate their results to 76 bits, the *lastmul_* operations round their results to the precision specified in the last argument using round-to-nearest. The rest of the operations have the same functionality as the corresponding operations in Figure 2, except for the size of the input operands. For example,

Q_i = lastmul_54x76(R_0[0:53], N_0, *25*)

indicates that the 54 most significant bits of R_0 are multiplied by all 76 bits of N_0. The result is rounded to 25 bits using round-to-nearest. Since R_0[0:53] is 54 bits, this multiplication is performed with two passes through the rectangular multiplier.

For single precision operands (OT = SINGLE), all of the multiplications, except for *lastmul_54x76*, require only a single pass through the multiplier tree and the division has a latency of 11 cycles. For double precision operands, the multiplications require one to three passes through the multiplier tree and the division has a latency of 17 cycles. For x87 operands, the latency depends on the required precision of the final result and is 13 cycles for single precision, 18 cycles for double precision, and 25 cycles for extended or internal precision.

Our second version of Goldschmidt's algorithm (GS-2), shown in Figure 4, uses a truncated version of R_i and takes advantage of the fact that R_i is close to 1.0 to reduce the number of bits in R_i used for the iterative multiplications and reduce the number of passes through the multiplier. For example, since $|R_0 - 1| < 2^{-13}$, the thirteen most significant bits of R_i are not needed. Based on Equation (17), this allows the computation

D_1 = itermul_54x76(R_0[0:53], D_0) (18)

which requires two passes through the multiplier tree in GS-1 to be replaced by the computation

D_1 = itermuladd_27x76(R_0[13:39], D_0, 13) (19)

which corresponds to

$$\begin{aligned} D_1 &= D_0 + D_0 \times R_0[13:39] \times 2^{-13} \\ &= D_0 + \{0'13, D_0\} \times R_0[13:39] \end{aligned} \quad (20)$$

This operation requires only a single pass through the multiplier with D_0 right shifted by 13 bits, the lower 13 bits of D_0 truncated, and the un-shifted value of D_0 added to the product. This operation compensates for the fact that $1 - R_i$ is used instead of R_i, as described in Section 2. Similar optimizations are used throughout the algorithm to reduce the number of passes through

the multiplier and the latency of the division algorithm. The operations that use these types of optimizations are *itermuladd_* and *lastmuladd_*. They implement operand shifting, multiplication, and addition by using a modified version of the multiplier described in Section 3. The *lastmuladdd* operation is similar to the *itermuladd* algorithm, except that the result is rounded to the number of bits specified by its last argument using round-to-nearest.

Program: *Goldschmidt's Division Algorithm with Reduced R_i on a 27 x 76 Multiplier (GS-2)*

Input = (A,B,OT, PC,RC), Output = (Q_f)

Operations	Cycles
X_0 = recip_estimate(B)	1-3
D_0 = itermul_27x76(X_0, B), R_0 = comp1(D_0)	4-5
N_0 = itermul_27x76(X_0, A)	5-6
if (OT == SINGLE) {	
Q_i = lastmuladd_27x76(R_0[13:39], N_0, 13, *25*)	7-8
REM = backmul_25x24(Q_i, B, A),	
Q_f = round(Q_i, *REM*, *24*, *RC*)	9-10
goto END DIVISION }	
if (OT == X87 and PC = 24) goto X87 DIV	
D_1 = itermuladd_27x76(R_0[13:39], D_0, 13),	
R_1 = comp1(D_1)	6-7
N_1 = itermuladd_27x76(R_0[13:39], N_0, 13)	7-8
if (OT == DOUBLE) {	
Q_i = lastmuladd_54x76({R_1[26,75], N_1, 26, *54*)	9-11
REM = backmul_54x53(Q_i, B, A),	
Q_f = round(Q_i, *REM*, *53*, *RC*)	12-14
goto END DIVISION }	
X87 DIV:	
if (PC == 24)	
Q_i = lastmuladd_27x76(R_0[13,39], N_0, 13, *25*)	7-8
else if (PC == 53)	
Q_i = lastmuladd_54x76(R_1[26,75], N_1, 26, *54*)	9-11
else {	
D_2 = itermuladd_27x76(R_1[26:52], D_1, *26*),	
R_2 = comp1(D_2)	8-9
N_2 = itermuladd_27x76(R_1[26:52], N_1, 26)	9-10
Q_i = lastmuladd_27x76(R_2[52:75], N_2, 52, *PC+1*)}	11-12
REM = backmul_65x64(Q_i, B, A),	
Q_f = round(Q_i, *REM*, *PC*, *RC*)	See *
END DIVISION:	

* 9-12 (*PC*=24) , 12-15 (*PC*=53), 13-16 (*PC*=64/68)

Figure 4: Goldschmidt's algorithm with reduced R_i on a 27 x 76 multiplier (GS-2)

5. Algorithm comparison

Table 2 compares the latency in cycles for each division algorithm, based on the multiplication latencies given in Table 1. In Table 2, (S), (D), and (E) indicate results are rounded to single, double, or extended precision, respectively. For completeness, the latency of the original division algorithm [15] on the AMD-K7 microprocessor with a 76x76 multiplier is also given, and denoted as K7 (76x76). The 76x76 multiplier is roughly 2.5 times larger than our 27x76 multiplier. Table 2 also shows the latency for the K7 division algorithm [15], when it has minor modification to work with our rectangular multiplier. This modified algorithm is denoted as K7 (27x76). As shown in Table 2, the two proposed algorithms have better latency than the AMD-K7 (27x76) algorithm for all operand types and precisions. The GS-2 (27x76) algorithm has the lowest overall latency for all operand types and precisions. Compared to the GS-1 (27x76) algorithm, the GS-2 (27x76) algorithm reduces the latency by one cycle for single precision, three cycles for double precision, and nine cycles for extended precision, when the input and output operands have the same precision.

Table 3 shows the number of passes through the multiplier for each division algorithm, based on the number of multiplier passes for the various multiplication sizes given in Table 1. For example, a 27x76 multiplication only requires a single pass through the multiplier and a 76x76 multiplication requires 3 passes through the multiplier. As shown in Table 3, the GS-2 algorithm has the fewest passes through the multiplier for all operand types and precisions. The number of passes through the multiplier is important since it impacts the power dissipated by the division algorithm and also indicates how available the multiplier is for implementing other operations.

Table 2: Latency of division algorithms (cycles)

Algorithm	Single (S)	Double (D)	Ext. (S)	Ext. (D)	Ext. (E)
K7 (76x76)	16	20	16	20	24
K7 (27x76)	14	20	14	20	26
GS-1 (27x76)	11	17	13	18	25
GS-2 (27x76)	10	14	12	15	16

Table 3: Multiplier passes of division algorithms

Algorithm	Single (S)	Double (D)	Ext. (S)	Ext. (D)	Ext. (E)
K7 (76x76)	12	18	12	18	24
K7 (27x76)	8	14	8	14	20
GS-1 (27x76)	5	11	7	12	18
GS-2 (27x76)	4	8	6	9	10

Compared to the K7 (27x76) algorithm, the GS-1 (27x76) algorithm has roughly the same hardware complexity, but more complex control logic to handle the different multiplication sizes. The GS-2 algorithm has the most complexity, since it has additional multiplexers to shift R_i, N_i, and D_i and it has modifications to the multiplier tree to perform multiply-add operations. For our implementation, the relatively small increase in hardware complexity of the GS-2 algorithm is less important than the reduced latency and passes through the rectangular multiplier.

6. Conclusions

This paper presents and compares variations of Goldschmidt's division algorithm for an x86 microprocessor that utilizes a rectangular multiplier. Of the algorithms presented in this paper, the GS-2 algorithm has the lowest latency and requires the fewest passes through the rectangular multiplier. All of the algorithms presented in this paper have been verified through extensive error analysis. The GS-2 algorithm has been modeled in Verilog and simulated using over 100 million test vectors for the supported operand types and result precisions.

References

[1] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, vol. 20, no. 4, pp. 47-57, July 2000.

[2] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions*, Revision 3.07, September 2006.

[3] ANSI and IEEE, *IEEE Standard for Binary Floating-point Arithmetic*, 1985.

[4] W.-C. Ma and C.-L. Yang, "Using Intel Streaming SIMD Extensions for 3D Geometry Processing," *Proceedings of the 3rd IEEE Pacific-Rim Conference on Multimedia*, pp. 1080-1087, December 2002

[5] S. F. Oberman and M. J. Flynn, "Division Algorithms and Implementations," *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833-854, August 1997.

[6] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*, Kluwer Academic Publishers, 1994.

[7] D. Wong and M. Flynn, "Fast Division Using Accurate Quotient Approximations to Reduce the Number of Iterations," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 981-995, August 1992.

[8] W. S. Briggs and D. W. Matula, "A 17 × 69 Bit Multiply and Add Unit with Redundant Binary Feedback and Single Cycle Latency," *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pp. 163-170, July 1993.

[9] W. S. Briggs and D. W. Matula, "Method and Apparatus for Performing Division Using a Rectangular Aspect Ratio, Multiplier," U.S. Patent No. 5,046,038, 1989.

[10] W. S. Briggs and D. W. Matula, "Method and Apparatus for Performing Prescaled Division," U.S. Patent No. 5,475,630, 1995.

[11] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very High Radix Division with Prescaling and Selection by Rounding," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 909-918, August 1994.

[12] T. Lang and P. Montuschi, "Boosting Very High Radix Division with Prescaling and Selection by Rounding," *IEEE Transactions on Computers*, vol. 50, no. 1, pp. 13-27, January 2001.

[13] R. E. Goldschmidt, *Applications of Division by Convergence*, M.S. thesis, Dept. of Electrical Engineering, MIT, Cambridge, MA, June 1964.

[14] M. Flynn, "On Division by Functional Iteration," *IEEE Transactions on Computers*, vol. 19, no. 8, pp. 702-706, August 1970.

[15] S. F. Oberman, "Floating-point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor," *In Proceedings of the 14th IEEE Symposium on Computer Arithmetic,* pg. 106-115, 1999.

[16] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers,"The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM Journal of Research and Development*, vol. 11, pp. 34-53, Jan. 1967.

[17] H. Darley, M. Gill, D. Earl, D. Ngo, P. Wang, M. Hipona, and J. Dodrill, "Floating Point/Integer Processor with Divide and Square Root Functions," U.S. Patent No. 4,878,190, 1989.

[18] E. M. Schwarz, L. Sigal, and T. J. McPherson, "CMOS Floating-point Unit for the S/390 Parallel Enterprise Server G4," *IBM Journal of Research and Development*, vol. 41, no. 4/5, pp. 475-488, July/September 1997.

[19] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann Publishers, 2004.

[20] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.

[21] I. Koren, *Computer Arithmetic Algorithms*, A. K. Peters, 2002.

[22] G. Even, P.-M Seidel, and W. E. Ferguson, "A Parametric Error Analysis of Goldschmidt's Division Algorithm," *16th IEEE Symposium on Computer Arithmetic*, pp. 165-171, June 2003.

[23] G. Even and P.-M. Seidel, "Pipelined Multiplicative Division with IEEE Rounding," *IEEE International Conference on Computer Design*, pp. 240-245, 2003.

[24] G. Even and P.-M. Seidel, "Pipelined Multiplicative Division with IEEE Rounding," U.S. Patent No. 2004/0128338, July, 2004.

[25] S. F. Oberman, "Bipartite Look-up Table with Output Values Having Minimized Absolute Error," U.S. Patent No. 6,223,192, April, 2001.