Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7^{TM} Microprocessor

Stuart F. Oberman
California Microprocessor Division
Advanced Micro Devices
Sunnyvale, CA 94088
stuart.oberman@amd.com

Abstract

This paper presents the AMD-K7 IEEE 754 and x87 compliant floating point division and square root algorithms and implementation. The AMD-K7 processor employs an iterative implementation of a series expansion to converge quadratically to the quotient and square root. Highly accurate initial approximations and a high performance shared floating point multiplier assist in achieving low division and square root latencies at high operating frequencies. A novel time-sharing technique allows independent floating point multiplication operations to proceed while division or square root computation is in progress. Exact IEEE 754 rounding for all rounding modes and target precisions has been verified by conventional directed and random testing procedures, along with the formulation of a mechanically-checked formal proof using the ACL2 theorem prover.

1 Introduction

The AMD-K7 is an out-of-order, three-way superscalar x86 microprocessor with a 15-stage pipeline, organized to allow 500+ MHz operation. The processor can fetch, decode, and retire up to three x86 instructions per cycle, with independent integer and floating-point schedulers. Up to three operations per clock are dispatched from the 36-entry floating-point scheduler to three floating-point execution pipelines.

The AMD-K7 floating point unit [1] is an x87 compatible FPU. This extension defined a stack architecture, along with IEEE 754 [2] compliance for addition, multiplication, division, and square root operations. All four rounding modes of the standard are supported, along with three working precisions: single, double, and extended. All of the fundamental computations are calculated with an extended precision significand and exponent. Only at the conclusion of an operation is the significand conditionally rounded to a lower precision. The AMD-K7 processor also supports a wider internal precision format, with an 18 bit exponent and a 68 bit significand. Exactly rounded-to-nearest division is supported at this precision as a means for computing highly accurate transcendental functions.

Operation	Latency / Throughput (cycles)		
	Single	Double	Extended / Internal
Division	16/13	20/17	24/21
Square Root	19/16	27/24	35/32

Table 1. AMD-K7 division and square root performance

In order to achieve high overall floating point performance with high operating frequencies, we determined it was necessary to minimize the latencies of the fundamental operations. Reducing the number of logic levels per pipeline stage to increase operating frequency can easily cause functional unit latencies to increase by a proportional amount, with little net gain in overall performance. To realize a true performance gain, improvements in the algorithms for the operations are required to reduce the latency of the fundamental operations.

This paper discusses the division and square root implementation in the AMD-K7 FPU. These algorithms reduce the latency for these operations in a high clock-rate implementation, yielding the performance shown in Table 1. Allowing out-of-order execution of operations provides some amount of latency tolerance, since the machine can be kept busy with independent work whenever it is present. However, we determined that to achieve high system performance, the latency of all of the fundamental operations must be minimized. Many software applications have long

dependency chains in their computation, and thus performance becomes heavily dependent upon the latency of the functional units themselves. While division and square root are typically infrequent operations, it has been shown that ignoring their implementations can result in significant system performance degradation for many applications [3]. We confirmed this result in our target workloads, and we therefore designed high performance division and square root algorithms which could exploit our fast FP multiplier.

We discuss the theory of division and square root by functional iteration and present the algorithms used in the AMD-K7 FPU in Section 2. In Section 3, we examine the design of the shared FP multiplier and demonstrate how the division and square root algorithms use this hardware. We analyze the various design issues that arose throughout the project in Section 4. We discuss our verification strategies in Section 5. We present optimizations to the implementation that improve overall FP performance in Section 6. We draw final conclusions in Section 7.

2 Functional Iteration

2.1 Motivation

The most common algorithm used for division and square root in modern floating point units is digit recurrence. One implementation of digit recurrence is *SRT*, which has been used in many microprocessors. SRT division and square root use subtraction as the fundamental operator to retire a fixed number of quotient bits in every iteration. Unfortunately, it is this linear convergence to the result that makes this algorithm class undesirable for our application. For rapid computation of wide-precision results, we desire better than linear convergence.

Unlike digit recurrence, division and square root by functional iteration utilize multiplication as the fundamental operation. Multiplicative algorithms are able to take advantage of high-speed multipliers to converge to a result quadratically. Rather than retiring a fixed number of quotients and square root bits in every cycle, multiplication-based algorithms are able to at least double the number of correct result bits in every iteration. This class of algorithm has been widely discussed in the literature, including Flynn [4] and Goldschmidt [5].

There are two main forms of multiplicative iterations. The Newton-Raphson division and square root iteration uses a dependent chain of multiplications to converge to the reciprocal / reciprocal square root. This algorithm is self-correcting, in that any error in computing the approximation in a given iteration can be corrected in the subsequent iteration, since all operations are dependent. The series expansion iteration, often called *Goldschmidt's algorithm*, reorders the operations in the Newton-Raphson iteration to increase the parallelism and reduce the latency. However, in this implementation the result is computed as the product of independent terms, and the error in one of them is not corrected. To account for this error, the iteration multiplications require extra bits of precision to guard against the accumulation of uncorrected rounding error.

The K7 implements a variant of Goldschmidt's algorithm to compute division and square root. Unlike the previously-mentioned commercial implementations of functional iteration, the K7 FPU has additional features to support the constraints imposed by IEEE 754 and x87 compatibility, as well as extra K7 functionality. These include 1) Exactly-rounded IEEE 754 extended-precision results, where the result may be also down-rounded to single or double precision, 2) Exactly rounded-to-nearest internal precision division, with a 68 bit significand, and 3) Producing the correct C1 bit to indicate whether roundup occurred.

2.2 AMD-K7 Division and Square Root Algorithms

A division or square root operation in the K7 can be defined by the function

DIV-SQRT(op,pc,rc,a,b,z),

with inputs as follows:

(a) $op \in \{\texttt{OP-DIV}, \texttt{OP-SQRT}\}$;

(b) pc is an external precision control specifier;

(c) rc is a rounding control specifier;

(d) a and b are floating point input operands.

In the case $op = \texttt{OP-DIV}$, the output z represents an appropriately rounded approximation of the quotient a/b; when $op = \texttt{OP-SQRT}$, b is ignored and an approximation of $\sqrt{a}$ is returned.

A floating point representation for an arbitrary number x comprises three bit vectors, corresponding to the sign, significand, and exponent of x. A format is defined by the number of bits allocated to the significand, $sig(x)$, and the exponent, $expo(x)$, expressed together as $(sig(x), expo(x))$. The formats that are supported by the K7 floating point division and square root operations include $(24, 8)$, $(53, 11)$, and $(64, 15)$, which correspond to *single*, *double*, and *extended* precision as specified by IEEE 754. A wider *internal* precision, $(68, 18)$, is also supported for division. A combination of op and pc determines the target precision for the operation.

The rounding modes supported in the K7 are RN, RZ, RM, and RP, which correspond to the rounding modes round-to-nearest-even, truncation, round-to-minus-infinity, and round-to-positive-infinity, respectively. A combination

Program *Division*:

BEGIN_DIVISION: Input = (a,b,pc,rc) Output = (q_f)

x_0 = recip_estimate(b)

d_0 = ITERMUL(x_0,b), r_0 = comp1(d_0)
n_0 = ITERMUL(x_0,a)
if (PC == SINGLE)
  n_f = n_0, r_f = r_0
  goto END_DIVISION

d_1 = ITERMUL(d_0,r_0), r_1 = comp1(d_1)
n_1 = ITERMUL(n_0,r_0)
if (PC == DOUBLE)
  n_f = n_1, r_f = r_1
  goto END_DIVISION

d_2 = ITERMUL(d_1,r_1), r_2 = comp1(d_2)
n_2 = ITERMUL(n_1,r_1)
n_f = n_2, r_f = r_2

END_DIVISION:
q_i = LASTMUL(n_f,r_f,pc)

rem = BACKMUL(q_i,b,a), q_f = round(q_i,rem,pc,rc)

Figure 1. Program Division

of op and rc determines the rounding mode for the operation.

The K7 algorithm for division and square root is represented by the programs *Division* and *Square Root*, shown in Figures 1 and 2. The division and square root programs employ several K7-specific hardware functions which are discussed in detail in the next section.

3 Hardware Organization

The division and square root programs are implemented in hardware within a single execution pipeline. A state machine within the FP multiplier pipeline detects if a valid division or square root opcode is dispatched to the pipeline. Upon receipt of an appropriate opcode, the state machine begins sequencing through the appropriate states to realize the programs of Figures 1 and 2. The state machine controls several muxes within the multiplier pipeline, along with enable signals to conditionally advance several registers. These programs could be implemented in K7 microcode instead of through a hardware state machine. However, all of the microcode ops would need to go through the decoders and central scheduler, greatly reducing the throughput of the

Program *Square Root*:

BEGIN_SQRT: Input = (a,pc,rc) Output = (s_f)

x_0 = recipsqrt_estimate(a)

t_0 = ITERMUL(x_0,x_0)

if (PC == SINGLE)
  d_f = ITERMUL(x_0,a)
  n_0 = ITERMUL(t_0,a), r_f = comp3(n_0)
  GOTO END_SQRT

n_0 = ITERMUL(t_0,a), r_0 = comp3(n_0)
d_0 = ITERMUL(x_0,a)

t_1 = ITERMUL(r_0,r_0)
d_1 = ITERMUL(d_0,r_0)

n_1 = ITERMUL(n_0,t_1), r_1 = comp3(n_1)

if (PC == DOUBLE)
  d_f = d_1, r_f = r_1
  GOTO END_SQRT

t_2 = ITERMUL(r_1,r_1)
d_2 = ITERMUL(d_1,r_1)

n_2 = ITERMUL(n_1,t_2), r_2 = comp3(n_2)
d_f = d_2, r_f = r_2

END_SQRT:
s_i = LASTMUL(d_f,r_f,pc)

rem = BACKMUL(s_i,s_i,a), s_f = round(s_i,rem,pc,rc)

Figure 2. Program Square Root

machine for independent operations. Thus, we chose to support maximum floating point throughput by implementing a dedicated hardware state machine which only requires a single opcode to be decoded and scheduled. Figure 3 shows a block diagram of the FP significand multiplier pipeline including additional hardware to support division and square root computation.

3.1 Multiplier Overview

The multiplier pipeline is designed in static CMOS logic to expedite design-time and circuit verification. The multiplier has a latency of four cycles, and it is fully-pipelined. It is configured to operate on a maximum of 76 bit operands. This maximum width is required to support exactly-rounded

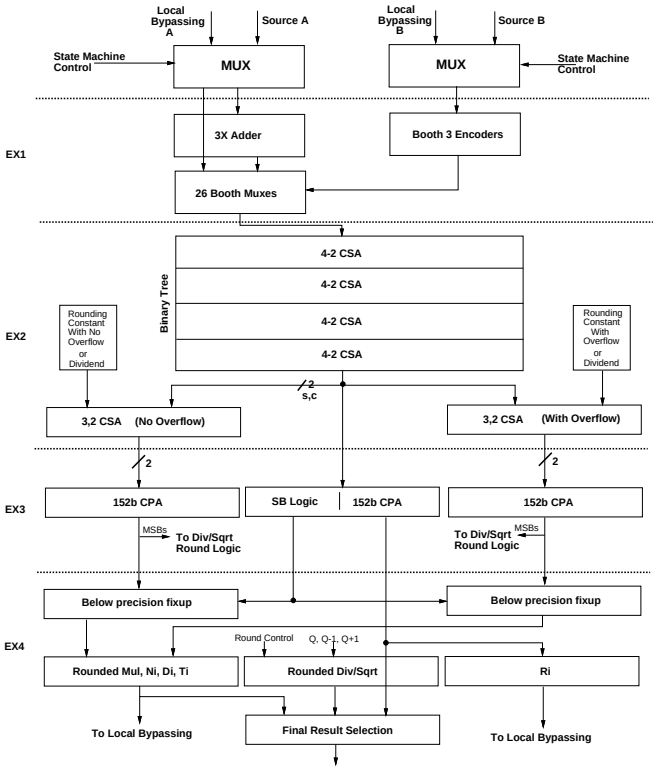

Figure 3. Multiplier pipeline

68 bit division and this choice is discussed later. The multiplier is also configured to compute all AMD 3DNow!^{TM} SIMD FP multiplications [6]; the algorithms to support this are not discussed here.

In the first cycle of the multiplier, EX1, overlapping groups of four bits of the multiplier operand are inspected as per the Booth 3 multiplier algorithm [7]. In parallel, a special 78 bit adder generates the 3x multiple of the multiplicand. The Booth encoders generate 26 control signals which control the selection of the 26 Booth muxes to form the appropriately signed multiples of the multiplicand.

In the second cycle, EX2, the 26 partial products are reduced to two through a binary tree of 4-2 compressors. The individual 4-2 compressors are designed in static CMOS with single-rail inputs and outputs. While a static dual-rail compressor with dual-rail inputs and outputs is typically faster than single-rail, it is also larger and requires twice the routing resources. In our implementation, the increased routing congestion and larger cells increased the wire-lengths between compressors, increasing overall delay compared with the single-rail design. The reduction tree is fundamentally a parallelogram. However, we implemented several folding and interleaving techniques which resulted in a rectangular tree and left internal wire lengths constant. The first portion of the multiplier rounding algorithm is then applied. This algorithm involves adding a rounding constant to the sum and carry outputs of the tree. Since the normalization of the assimilated result is unknown at this point, the addition is performed both assuming no overflow occurs and assuming overflow occurs. The additions themselves are implemented using an additional level of (3,2) carry-save adders. These (3,2) adders are also used as part of the back-multiply and subtract operation BACKMUL which forms the remainder required for quotient and square root rounding. The rounding constants used for the different precisions and rounding modes are shown in Table 2. The first four rows correspond to constants for regular multiplication operations, while the last three are for division / square root specific operations. In this table, $!x$ implies the bit inversion of x, and 24'b0 implies 24 zero bits.

The third cycle, EX3, forms the three carry-assimilated results. Two of these results are rounded results assuming that either overflow or no overflow occurs. The third is the raw unrounded result. In parallel, sticky-bit logic examines the low order bits of the sum and carry vectors, as a function of pc, to determine whether or not $S + C = 0$, using an optimized technique similar to those previously reported [8].

In the fourth cycle, EX4, the bits below the target precision are appropriately cleared. While the rounding constant applied in EX2 insures that the result is correctly-rounded, it does not guarantee that the bits below the target LSB are cleared, as required for x87 compatibility. The LSB is then conditionally inverted for regular multiplication operations under RN as a function of the sticky-bit to correctly handle the nearest-even case. Finally, the MSB of the unrounded result determines whether or not overflow has occurred, and it selects between the no overflow and with overflow rounded results. For division and square root iterations, an extra result R_i is also provided which is the one's complement of the regular unrounded multiply result for division and an approximation to $\frac{3-N}{2}$ for square root. Both of the results are available for local storage and bypassing for use within the division and square root iterations. In the case of the last cycle of a division or square root BACKMUL op, the appropriately rounded result is chosen from the previously-computed target results: Q, $Q+1$, or $Q-1$.

The unrounded result is also used in the case of IEEE tiny results with the underflow exception masked; i.e. tininess occurs when the computed rounded result is below the minimum extended precision normal number. In this instance the unrounded result is passed to a microcode handler which can properly denormalize and round the result as required by x87 compatibility. The unrounded result is also used to determine whether roundup has occurred for proper setting of the C1 condition code. Roundup occurs when the rounded result differs from the unrounded result, and the C1 bit can be set appropriately given this informa-

	SINGLE	DOUBLE	EXTENDED	INTERNAL
RN	(24'b0,1'b1,126'b0)	(53'b0,1'b1,97'b0)	(64'b0,1'b1,86'b0)	(68'b0,1'b1,82'b0)
RZ	151'b0	151'b0	151'b0	-
RM	(24'b0,(127(Sign)))	(53'b0,(98(Sign)))	(64'b0,(87(Sign)))	-
RP	(24'b0,(127(!Sign)))	(53'b0,(98(!Sign)))	(64'b0,(87(!Sign)))	-
LASTMUL	(25'b0,1'b1,125'b0)	(54'b0,1'b1,96'b0)	(65'b0,1'b1,85'b0)	(69'b0,1'b1,81'b0)
ITERMUL	(76'b0,1'b1,74'b0)			
BACKMUL	(!Dividend[67:0],(83(1'b1)))			

Table 2. Rounding constants

tion. For division and square root, the unrounded result is synthesized by appropriately choosing between $Q - 1$ and Q.

The extra multiplier hardware required to support division and square root iterations that actually impacted total area included flip-flops to store intermediate results, an incrementer, and the state machine. We estimate that division and square root support accounts for about 10% of the total area of the multiplier unit. Designing a parallel SRT divider with similar performance would have required substantially more area and circuit design than our implementation required. We therefore conclude that this implementation represents a good balance of performance and area.

3.2 Special Iteration Operations

3.2.1 Recip_Estimate and RecipSqrt_Estimate

The initial approximation x_0 to the reciprocal of b, in the case $op =$ `OP-DIV`, is derived from a pair of tables, each containing 2^{10} entries, which we represent by the functions *recip-rom-p* and *recip-rom-q*. For $op =$ `OP-SQRT`, a separate pair of tables, each containing 2^{11} entries, represented by the functions *sqrt-rom-p* and *sqrt-rom-q*, is similarly used to derive an initial approximation to the reciprocal square root of a.

The basic theory of compressing reciprocal ROM tables using interpolation is discussed in [9]. We implemented an optimized and simplified reciprocal and reciprocal square root estimate interpolation unit which provides the best compression and performance for the requirements of our algorithms. This interpolation unit is also used to form the values for the AMD 3DNow! reciprocal and reciprocal square root functions PFRCP and PFRSQRT. Each p table entry is 16 bits, while the q table entries are each 7 bits. The total storage required for reciprocal and reciprocal square root initial approximations is 69 Kbits.

To form an estimate, the appropriate set of p and q tables is accessed in parallel. The results are added and the 16 bit sum is appended to a leading 1 to produce the reciprocal or reciprocal square root estimate. The latency for this process is three cycles. By exhaustive test we determined that the reciprocal estimate is accurate to at least 14.94 bits, while the reciprocal square root estimate is accurate to at least 15.84 bits.

3.2.2 ITERMUL(x,y)

This is a multiplication operation of x and y that forces the rounding to be round-to-nearest. It assumes that each of the input operands are 76 bits wide, and the 152 bit intermediate result is rounded to 76 bits. This wider precision is required to accommodate the uncorrected rounding errors which accumulate throughout the iterations.

3.2.3 LASTMUL(x,y,pc)

This is a multiplication operation of x and y that forces the rounding to be round-to-nearest. It performs the rounding to a precision one bit wider than the target precision specified by pc. For division, just prior to the rounding of this operation, the double-width product Q' is required to be accurate to at least

$$-2^{-(pc+2)} < Q - Q' < 2^{-(pc+2)} \tag{1}$$

where Q is the infinitely precise quotient, and pc is the target precision in bits, where 1 ulp for such a pc bit number is $2^{-(pc-1)}$ [10]. Exact rounding requires that the result have an error no worse than ± 0.5 ulp, or 2^{-pc}. Our rounding algorithm requires the result to be computed to an accuracy of at least 1 more bit. Further, since the final quotient result can be in the range of (0.5,2), 1 bit of normalization may be required, requiring 1 more bit of accuracy. For square root, just prior to the rounding of this operation, the double-width product S' is required to be accurate to at least

$$-2^{-(pc+1)} < S - S' < 2^{-(pc+1)} \tag{2}$$

where S is the infinitely precise square root, and pc is the target precision in bits. Since the square root result is in the range [1,2), it has a looser constraint on the accuracy of the input to this operation.

After rounding and normalizing to $pc + 1$ bits through the LASTMUL op, the resulting value R'' satisfies

$$-2^{-pc} < R - R'' < 2^{-pc}, \tag{3}$$

where R is either the infinitely precise quotient or square root as appropriate. Thus, the value can have an error of (-0.5,+0.5) ulp with respect to the final pc bit number.

3.2.4 BACKMUL(b,q,a)

This op is a multiplication operation that operates on the two source operands b and q, and it also accepts a third operand a. The 152 bit intermediate product of the two sources in carry-save form is added with an inverted version of the third operand, with the low-order bits filled with 1's as shown in Table 2. These three values are then input into the rounding carry-save adders in EX2, with the unused lsb carry bit set to one, realizing the function of $b \times q + TwosComp(a)$. This implements the negative version of the back multiply and subtraction operation to form the remainder, i.e.

$$b \times q - a$$

is implemented instead of the desired

$$a - b \times q.$$

The sign of this remainder is thus the negative of the true remainder sign. This operation returns two bits of status: whether the sign of the remainder is negative, taken from a high order bit of the result, and whether the remainder is exactly zero, using fast sticky-bit logic. Since q could be rounded to any of four precisions, the high order bit is chosen high enough to allow it to suffice for all of the precisions.

3.2.5 comp1(x)

This operation returns the one's complement by bit inversion of the unrounded version of the currently-computed product. The unbiased exponent is forced to either -1 or 0 depending upon whether the result should be in the binade [0.5,1) or [1,2). Using the one's complement instead of the two's complement in the iterations adds a small amount of error. However, this error is taken into account when designing the width of the multiplier and the required accuracy of the initial approximations.

3.2.6 comp3(x)

This operation returns an approximation to $\frac{3-x}{2}$ of the unrounded version of the currently-computed product. This approximation is formed by bit inversion and shifting.

3.2.7 round(q_i,rem,pc,rc)

The rounding function assumes that a biased trial result q_i has been computed with $pc+1$ bits, which is known to have an error of (-0.5,+0.5) ulp with respect to pc bits. The extra bit, or guard bit, is used along with the sign of the remainder, a bit stating whether the remainder is exactly zero, and rc to choose from three possible results, either q, $q-1$, or $q+1$ which are equal to q_i truncated to pc bits and decremented or incremented appropriately. The rounding details are shown in Table 3.

Guard Bit	Rem-ainder	RN	RP (+/-)	RM (+/-)	RZ
0	=0	trunc	trunc	trunc	trunc
0	-	trunc	trunc/dec	dec/trunc	dec
0	+	trunc	inc/trunc	trunc/inc	trunc
1	= 0	RNE	inc/trunc	trunc/inc	trunc
1	-	trunc	inc/trunc	trunc/inc	trunc
1	+	inc	inc/trunc	trunc/inc	trunc

Table 3. Action table for round function

For RN, in the case of an exact halfway case, it is necessary to inspect L, the lsb of q to determine the action. If $L = 0$, then the result is correctly rounded to nearest-even. Otherwise, the result is incremented to the closest even result. It should be noted that for division where the precision of the input operands is the same as that of the result, the exact halfway case can not occur. Only when the result precision is smaller than the inputs can such a result occur and rounding be required. For the directed rounding modes RP and RM, the action may depend upon the sign of the quotient estimate. Those entries that contain two operations such as *pos/neg* are for the sign of the final result itself being positive and negative respectively.

This function, along with the computation of $q-1$ and $q+1$, is implemented in parallel with the BACKMUL operation. The status information from the BACKMUL op is used as input to the ROUND function to quickly choose and return the correctly-rounded result.

3.3 Performance

The AMD-K7 implementation of the programs *Division* and *Square Root* through a hardware state machine yields the performance summarized in Table 1. The latencies include the time to form the initial approximation and the appropriate number of iterations. The scheduling of the special iteration multiplications is optimized to simultaneously minimize the latency of the operations and the number of extra storage registers required to hold intermediate results, as well as maximize the throughput.

4 Design Issues

The two primary design variables for the implementation of iterative division and square root are the minimum accuracy of the initial approximation and the minimum multiplier dimensions. We performed an initial loose error analysis using Mathematica to bound the worst case errors, and

we based the design upon this analysis. We used formal verification later to confirm the validity of our results.

To determine the accuracy of the Goldschmidt iterations, it is necessary to consider all of the sources of error in the final result. When converging to a result using Goldschmidt iterations, there are three possible sources of error in the pre-rounded result:

- Error due to initial approximation = ϵ_0
- Error due to use of one's complement, rather than true two's complement in the iterations = ϵ_{ones}
- Error due to using rounded results in the iteration multiplications = ϵ_{mul}

4.1 Division

4.1.1 Multiplier Dimensions

In this analysis, we consider the worst case output precision, internal precision with a 68 bit significand, such that three iterations of the algorithm are required to form an approximation to $Q = \frac{a}{b}$. Further, we assume that for performance reasons, the one's complement is used rather than the two's complement. The FP multiplier used in the iterations takes two inputs each of width n bits and produces an n bit result, such that 1 ulp for the result is $2^{-(n-1)}$.

We wrote a Mathematica program that implemented the *Division* program with three iterations. The errors due to multiplier rounding were separated in the iterations. This is to correctly account for the fact that the rounding error in the successive D_i refinements are self-correcting, while those in the N_i refinements are not. Rather, the N_i refinements suffer the additive error of the multiplies intrinsic to the N_i step itself as well as the error in the D_i step. Also, at each point where the rounding error is multiplied by the ratio of the two input operands, the worse case operand values in the range [1,2) were substituted in order to maximize the worst-case possible error.

From Mathematica, the final error in the final $2n$ bit pre-rounded quotient estimate q after three iterations is:

$$\epsilon_q \quad = \quad -ab^7\epsilon_0^8 + 10\epsilon_{mul} + 2\epsilon_{ones} \tag{4}$$

The initial approximation error is treated later. However, it is readily apparent that with an approximation accurate to at least 2^{-14}, the error in the final result due to the initial approximation after three iterations is on the order of 2^{-104}, and thus is not of concern. Instead, obtaining the required accuracy for exact rounding of internal precision results is dominated by the errors due to the multiplier dimension, ϵ_{mul} and ϵ_{ones}.

The value of ϵ_{ones} is a constant value of -1 ulp. The ulp error of multiplication rounding is ± 0.5 ulp. However, due to possible overflow before rounding, the mathematical value, with respect to the binade of the input operands for the specific multiplication, can be as much as ± 1 ulp. To provide the most conservative constraints, the ± 1 ulp value is used in the analysis for all of the rounding errors due to multiplication. Substituting these values into the expression for the pre-rounded quotient estimate yields a loose constraint on the final error in the result:

$$-2^{-(n-5)} \quad < \quad \epsilon_q < 2^{-(n-5)} \tag{5}$$

The design challenge becomes determining the minimum number of bits required for the multiplier. For the K7 FPU, the widest precision requiring exact division rounding is internal precision with $pc = 68$. For internal precision 1 ulp is 2^{-67}. The constraint for exact rounding of this precision is that the final double-width quotient estimate $N3$ formed in the LASTMUL op have error ϵ_{N3} given by

$$-2^{-70} < \epsilon_{N3} < 2^{-70} \tag{6}$$

as per Equation 1. By equating the required precision with the precision obtained through the iterations using an n by n bit multiplier:

$$\begin{aligned} 2^{-(n-5)} &= 2^{-70} \\ n &= 75 \end{aligned} \tag{7}$$

Thus, the minimum required number of bits n to guarantee exactly-rounded internal precision quotients is 75. To provide even more margin in the design, we implemented a 76x76 bit multiplier.

Precision	Iterations	Error
Single	1	$ab\epsilon_0^2$
Double	2	$ab^3\epsilon_0^4$
Extended / Internal	3	$ab^7\epsilon_0^8$

Table 4. Initial approximation error for reciprocal

4.1.2 Table Design

Mathematica analysis was used to determine the contribution of the initial approximation error of the reciprocal to the final pre-rounded result for each of the target precisions. This is shown in Table 4.

For single precision division, the constraint on the error in the final double-width quotient estimate formed in the LASTMUL op for exact rounding as per Equation 1 is given by:

$$-2^{-26} < \epsilon_{N1} < 2^{-26} \tag{8}$$

Accordingly, for the worst case with $a = b = 2$, the maximum value for ϵ_0 can be determined:

$$\begin{aligned} 2 \times 2 \times \epsilon_0^2 &< 2^{-26} \\ \epsilon_0 &< 2^{-14} \end{aligned}$$

For double precision division, the constraint for exact rounding is:

$$-2^{-55} < \epsilon_{N2} < 2^{-55}$$

and thus the maximum value for ϵ_0 is

$$\epsilon_0 < 2^{-14.75}$$

For extended precision division, the constraint for exact rounding is:

$$-2^{-66} < \epsilon_{N3} < 2^{-66}$$

and thus the maximum value for ϵ_0 is

$$\epsilon_0 < 2^{-9.25}$$

The tightest constraint on the initial approximation to the reciprocal is from double precision, where $\epsilon_0 < 2^{-14.75}$. This guided the design of our reciprocal estimate which has an error of at most $2^{-14.94}$.

4.2 Square Root

We consider the worst case output precision, extended precision, such that three iterations of the algorithm are required to form an approximation to $\sqrt{a}$. Further, we assume that for performance reasons, a variant of the one's complement is used rather than a subtraction from three.

We wrote another Mathematica program which implemented the *Square Root* program with three iterations. The analysis shows that the worst case error in the final $2n$ bit pre-rounded square root estimate s, formed in the LASTMUL op, due only to rounding and one's complement after three iterations for extended precision is:

$$\epsilon_s = 7\epsilon_{mul} + 2\epsilon_{ones} \quad (9)$$

This error is less than that produced by internal precision division, so it is not an influencing factor on the dimension of the multiplier.

The tabulation for initial approximation error is shown in Table 5.

For single precision, the constraint on the accuracy of the double-width square root result S formed in LASTMUL for exact rounding of square root, as per Equation 2, is given by:

$$-2^{-25} < \epsilon_S < 2^{-25} \quad (10)$$

Accordingly, the maximum value for ϵ_0 can be determined:

$$\begin{aligned} 12\epsilon_0^2 &< 2^{-25} \\ \epsilon_0 &< 2^{-14.3} \end{aligned}$$

Precision	Iterations	Error
Single	1	$\frac{3}{2}a^{1.5}\epsilon_0^2$
Double	2	$\frac{27}{8}a^{2.5}\epsilon_0^4$
Extended	3	$\frac{2187}{128}a^{4.5}\epsilon_0^8$

Table 5. Initial approximation error for reciprocal square root

For double precision, the constraint for exact rounding is given by

$$-2^{-54} < \epsilon_S < 2^{-54}$$

and the maximum value for ϵ_0 is

$$\epsilon_0 < 2^{-15.2}$$

For extended precision, the constraint for exact rounding is given by

$$-2^{-65} < \epsilon_S < 2^{-65}$$

and the maximum value for ϵ_0 is

$$\epsilon_0 < 2^{-9.8}$$

Accordingly, the tightest constraint on the initial approximation of the reciprocal square root is from double precision, where $\epsilon_0 < 2^{-15.2}$. This guided the design of our reciprocal square root estimate which has an error of at most $2^{-15.84}$.

5 Verification

Verification for the division and square root implementation was focused on whether we could compute exactly-rounded results at all target precisions with all required rounding modes, given our choice of algorithms, multiplier dimensions, and initial approximations. We followed two parallel paths for verification: directed random vector testing and formal proofs.

5.1 Directed Random Vectors

For division, we wrote a C program which would generate random input operands such that the results would be very close to rounding boundaries. The theory of this program is based upon work of Kahan [11], and it is similar to that available in the UCBtest suite [12]. The program

operates in either single, double, or extended precision significand modes. It generates random divisors at the target precision, and it uses appropriate equations to generate a dividend such that the resulting quotient lies very near the rounding point between two machine numbers. Hundreds of thousands of vectors were run both on the RTL and gate models at the three precisions to verify IEEE rounding compliance, with no rounding errors found. Internal precision division was not verified using this procedure.

Similarly, for square root we wrote another C program which would directly generate operands whose square roots were very near rounding points. The theory for this operand generation process is based upon other work of Kahan [13]. These operands were run on the RTL and gate models at the three IEEE precisions, with no rounding errors found.

While this process of directed random vectors simulation gave us confidence that our algorithms and design choices were correct, it did not provide a guarantee. With vector-based verification, only exhaustive simulation can guarantee correctness. Given our limited available design and verification time, exhaustive simulation was not feasible.

5.2 Formal Proofs

To gain more confidence in the correctness of all of our design choices, we developed a mechanically verified proof of the multiplication, division, and square root operations. The proof is based on a formal description of the hardware, derived from an executable model that was written in C and used for preliminary testing. The details of the proof are not included here due to space limitations, but they are available at [14].

Two main theorems were formed regarding the correctness of the computed results from our implementation; one for division and one for square root. The statements of correctness are based on the exact rounding requirements of IEEE 754. Every step in the proof of these theorems has been formally encoded in ACL2 logic and mechanically checked with the ACL2 theorem prover [15], in the interest of eliminating the possibility of human error. The input to the prover, culminating in formal versions of these two theorems, consisted of more than 250 definitions and 3000 lemmas.

The proof provides several results confirming the correctness of our algorithms and implementation. First it demonstrates that the accuracy of our initial approximations is sufficient for all of our target precisions. The number of bits of accuracy is not parameterized in the proof, so we can only confirm that our estimates are sufficient. However, the dimension of the multiplier is a parameter in the proof. In the formalization of the model, the multiplier width is represented as a parameter M, the minimum value of which was to be determined. The proof confirms what our earlier analysis showed, that correctness requires only $M \geq 75$.

6 AMD-K7 Optimizations

6.1 Multiply under Division / Square Root

One of the disadvantages to the use of multiplication-based division and square root with a single multiplier is the loss in throughput for regular multiplication operations. State-machine implementations of iterative division and square root typically require use of the shared multiplier resource for the duration of the iterations. Although the latencies for the resulting operations can be low, as shown by our implementation, the shared multiplier is blocked until the operations complete. In the K7 design, it was not acceptable to completely block the multiplier pipeline for every division or square root operation. While these operations are not very frequent, they occur with enough frequency in our target workloads that blocking the multiplier would significantly degrade overall performance.

With a four cycle multiplier, we determined that the multiplier was only being utilized for approximately half of the cycles throughout the iterations. Each iteration of the algorithms requires two or three possibly parallel multiplications, but the next iteration can not begin until completion of the previous one. We configured the controlling state-machine to detect such unused cycles in the multiplier. It notifies the central scheduler several cycles in advance whenever this occurs. This provides the scheduler sufficient time to appropriately schedule an independent multiplication operation in the unused cycle. The only added overhead to support this mechanism is a small amount of storage in the pipeline to keep some status computed in the first few cycles until it is required during the BACKMUL operation at the conclusion of the division or square root operation. The properly-scheduled independent multiply operations can then use the available hardware seamlessly.

We modeled this mechanism as part of our performance analysis. We modeled 1) the pipeline being completely blocking for division / square root, 2) our implementation, and 3) a completely non-blocking division / square root unit. The implementation of our time-sharing scheme of the multiplier provides over 80% of the performance of a completely non-blocking division / square root unit. We deemed this level of performance more than satisfactory given the very small amount of overhead required.

6.2 Early Completion

In order to improve division performance further, we desired to reduce the latency for certain types of operands. FP division of a number by a power-of-two in theory amounts to just a reduction in the exponent, with the significand unchanged. Accordingly, such an operation could have a latency much lower than a typical division computation. In the case of x87 FP, the computation can be slightly more

complicated, as the significand itself needs to be conditionally rounded to a lower precision based upon the control word.

As the K7 can execute instructions out-of-order and tolerate variable latency, we allow division operations to have variable latencies. The controlling state machine notifies the central scheduler a fixed number of cycles in advance when the division or square root operation will complete. The scheduler uses this information to schedule dependent operations and subsequent division or square root instructions. So long as the latency is at least as long as the notification period, divides and square roots can be any number of cycles.

Thus, the K7 is able to complete the computation of quotients where the divisor is exactly a power-of-two in only 11 cycles, regardless of the target precision. The exponent adjustment occurs during the initial approximation lookup phase, while the rounding of the significand occurs using the LASTMUL and BACKMUL ops. Since the minimum required notification time for the scheduler in this instance is seven cycles, this latency is acceptable. Division operations where the dividend is exactly zero are also handled by this mechanism. To simplify the implementation, they use the same sequence of events, returning a zero result in 11 cycles, regardless of the target precision.

7 Conclusion

We have presented an x87 and IEEE compatible implementation of division and square root using quadratically-converging algorithms, demonstrating that they are a practical and potentially superior alternative to linear-converging algorithms. Using a high performance shared multiplier, we minimized the total die area required while maximizing system performance. Our formal proofs and directed testing show that both high performance and correctness of operation are simultaneously achievable.

8. Acknowledgements

The author wishes to thank James Fan for his contribution to the physical design of the multiplier pipeline, David Russinoff for his contribution to the verification of the algorithms, and Norbert Juffa for his assistance throughout the design process.

References

[1] A. Scherer, M. Golden, N. Juffa, S. Meier, S. Oberman, H. Partovi, F. Weber, "An out-of-order three-way superscalar multimedia floating-point unit," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, 1999.

[2] ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[3] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[4] M. Flynn, "On division by functional iteration," *IEEE Trans. Computers*, vol. C-19, no. 8, pp. 702–706, Aug. 1970.

[5] R. E. Goldschmidt, "Applications of division by convergence," M.S. thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1964.

[6] S. Oberman, F. Weber, N. Juffa, and G. Favor, "AMD 3DNow! technology and the K6-2 microprocessor," in *Proc. Hot Chips 10*, pp. 245–254, Aug. 1998.

[7] A. D. Booth, "A signed binary multiplication technique," *Quart. Journal Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[8] J. Cortadella and J. M. Llaberia, "Evaluation of A+B=K conditions without carry propagation," *IEEE Trans. Computers*, vol. 41, no. 11, pp. 1484–1488, Nov. 1992.

[9] D. DasSarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 12–25, July 1995.

[10] S. F. Oberman, *Design Issues in High Performance Floating Point Arithmetic Units*, Ph.D. thesis, Stanford University, Nov. 1996.

[11] W. Kahan, "Checking whether floating-point division is correctly rounded," Prof. Kahan's Lecture Notes, 1987.

[12] "UCBTEST Suite," available from http://www.netlib/org/fp.

[13] W. Kahan, "A test for correctly rounded SQRT," Prof. Kahan's Lecture Notes, 1996.

[14] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD K7 floating-point division and square root instructions," available at http://www.onr.com/user/russ/david/k7-div-sqrt.html.

[15] R. S. Boyer and J. Moore, *A Computational Logic Handbook*, Academic Press, Boston, MA, 1988.