An Area/Performance Comparison of Subtractive and Multiplicative Divide/Square Root Implementations

Peter Soderquist and Miriam Leeser
Cornell School of Electrical Engineering
Ithaca, NY 14853

Abstract

The implementations of division and square root in the FPU's of current microprocessors are based on one of two categories of algorithms. Multiplicative techniques, exemplified by the Newton-Raphson method and Goldschmidt's algorithm, share functionality with the floating-point multiplier. Subtractive methods, such as the many variations of radix-4 SRT, generally use dedicated, parallel hardware. These different approaches give rise to the distinct area and performance characteristics which are explored in this paper. Area comparisons are derived from measurements of commercial and academic hardware implementations. Representative divide/square root implementations are paired with typical add-multiply structures and simulated, using data from current microprocessor and arithmetic coprocessor designs, to obtain performance estimates. The results suggest that subtractive implementations offer a superior balance of area and performance, and stand to benefit most decisively from improvements in technology and growing transistor budgets due to their parallel operation. Multiplicative methods lend themselves best to situations where hardware re-use is mandated due to area or architectural constraints.

1 Introduction

1.1 Motivation

Every general-purpose microprocessor of recent design provides hardware support for division, and most implement square root as well. These implementations are split between two different types of algorithms. The IBM RS/6000 series and Mips R8000 employ multiplicative techniques. The DEC 21164 Alpha, Hewlett-Packard PA8000, IBM/Motorola PowerPC 604, Intel P6, Mips R4000 series, and Sun UltraSPARC, on the other hand, use hardware based on subtractive methods.

The different algorithms lead to distinct types of implementations. Multiplicative divide/square root designs make extensive use of the floating-point multiplier. They frequently require little additional hardware and yield low latencies. However, using a single functional unit for multiply, divide, and square root operations risks creating a performance bottleneck and complicates the design of the multiplier. Subtractive implementations require dedicated hardware, which is often more costly, and tend to have higher latencies. On the other hand, having a separate functional unit provides for parallel operation and de-couples the design of multiply and divide/square root hardware.

It is not obvious how these various advantages and disadvantages play themselves out in actual floating-point units. We examine the relative area of typical implementations, both qualitatively and quantitatively when possible. To compare performance, selected divide/square root implementations are paired with representative add-multiply structures and simulated, using data from commercial hardware. We also attempt to anticipate how different types of implementations might be affected by future improvements in technology.

1.2 Related work

Other researchers have performed comparisons of different division and square root implementations, usually focussing on a small subset of the possible alternatives. In his seminal paper on higher-radix division, Atkins [2] considers how to compute the cost of different types of SRT implementations. Stearns [17] discusses the merits and demerits of multiplicative and subtractive algorithms, and presents a design for SRT division and square root which synthesizes "the best of the ideas presented over the last thirty years." While quickly dismissing multiplicative methods, Peng et. al. [13] give a detailed but largely qualitative discussion of the area and performance properties of a variety of subtractive division methods. Taylor [18] performs a systematic study of ten related but different SRT divider designs with a quantitative comparison of cost and performance. Ercegovac and Lang [5] present a time and area analysis for a smaller but more diverse group of SRT dividers; Ercegovac, Lang, and Montuschi [4] perform a similar study of very high radix division methods. Using the SPECfp92 benchmarks, Oberman and Flynn [12] examine the tradeoffs of different division implementations at the system level.

This study attempts to address and, where possible, quantify the area and performance tradeoffs of division *and* square root implementations at the floating-point unit level. It uses a single, carefully chosen benchmark taken from real applications and focusses on the type of implementations used in current microprocessors.

2 Multiplicative divide/square root

2.1 Algorithms

Multiplicative algorithms use iterative refinement to achieve successively more accurate estimates of the quotient or square root. The two techniques used in current machines are the Newton-Raphson method and Goldschmidt's algorithm. Each technique transforms divide/square root computation into a series of multiplications, subtractions from a constant, and bit shifts. Both algorithms also have

1063-6889/95 $4.00 © 1995 IEEE

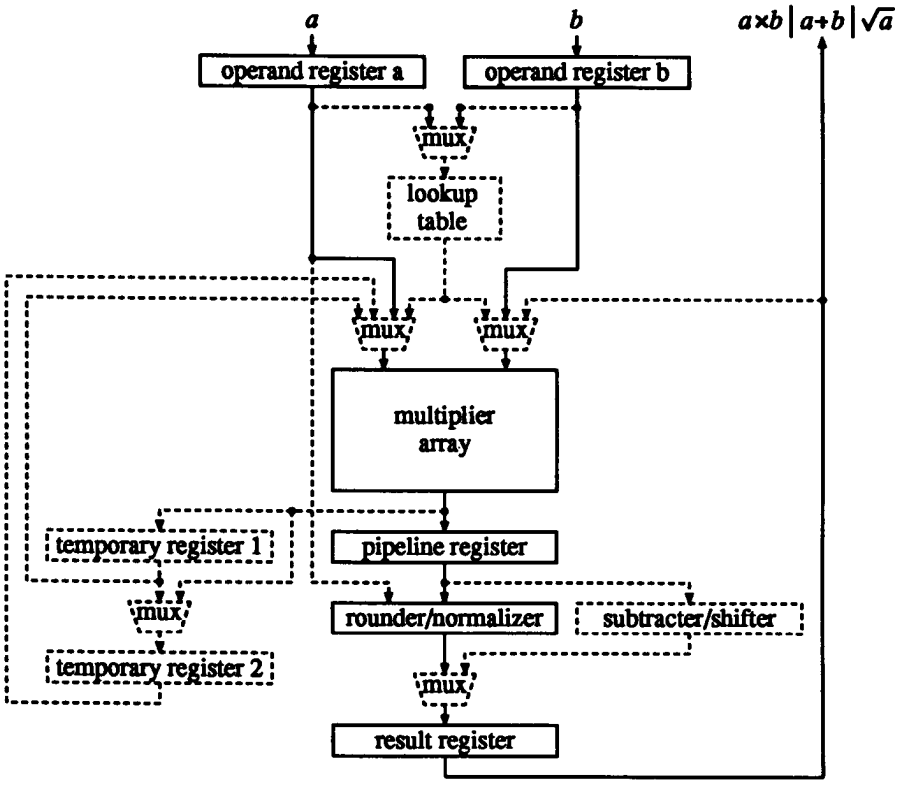


Figure 1: A floating-point multiplier enhanced for multiplicative divide/square root computation

quadratic convergence, which means that the number of accurate digits in the partial result doubles at every iteration, and both use reciprocal estimates to improve performance. The two methods differ primarily in the arrangement of operations, which has consequences for hardware implementation.

The Newton-Raphson method. A popular algorithm for several centuries, the Newton-Raphson method [14, 8] has been implemented in many designs, including the IBM RS/6000 series [9]. To compute the quotient a/b, let x_0 be an initial estimate or *seed value* close to $1/b$, and iterate over the formula

$$x_{i+1} = x_i \times (2 - b \times x_i)$$

until x_{i+1} is sufficiently close to $1/b$. Multiplying by a yields a/b within the desired accuracy. The square root $\sqrt{a}$ is computed by refining an estimate of $1/\sqrt{a}$ using the iteration

$$x_{i+1} = \frac{1}{2} \times x_i \times (3 - a \times x_i^2).$$

Calculating $a \times 1/\sqrt{a}$ produces the final estimate of $\sqrt{a}$.

Goldschmidt's algorithm. Goldschmidt's algorithm [8] performs the same operations as the Newton-Raphson method but in a different order. It has been implemented in the Sun SuperSPARC [3] and arithmetic coprocessors from Texas Instruments. Computing the quotient $a/b = x_0/y_0$ with Goldschmidt's algorithm involves multiplying both the numerator and denominator by a value r_i such that $x_{i+1} = x_i \times r_i$ and $y_{i+1} = y_i \times r_i$. With $r_i = 2 - y_i$, $y_i \to 1$, and therefore $x_i \to a/b$. To insure rapid convergence, both numerator and denominator are prescaled by a seed value close to $1/b$. Square root calculation is similar. To find $\sqrt{a}$, let $x_0 = y_0 = a$ and iterate over $x_{i+1} = x_i \times r_i^2$ and $y_{i+1} = y_i \times r_i$ so $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$. Let $r_i = (3 - y_i)/2$; then $x_i \to 1$, and consequently $y_i \to \sqrt{a}$. The prescaling operation uses an estimate of $1/\sqrt{a}$.

2.2 Implementations

Multiplicative implementations consist primarily of modifications to the floating-point multiplier. The choice of algorithm depends partly on the topology of the multiplier hardware. The IBM RS/6000 FPU is based on an atomic multiply-accumulate structure and uses specialized versions of the Newton-Raphson iterations to compute division and square root. For more conventional multipliers, Goldschmidt's algorithm has the advantage that the numerator and denominator operations in each iteration are independent, and can therefore be efficiently pipelined. Newton-Raphson implementations incur dependencies at every step of the computation, but may require less hardware because fewer concurrently existing values need to be maintained [16].

The block diagram in Figure 1 shows a conventional, pipelined floating-point multiplier tailored for Goldschmidt's algorithm. Dotted lines indicate datapath components and routing not required for multiplication alone. Details will vary depending on the structure of the multi-

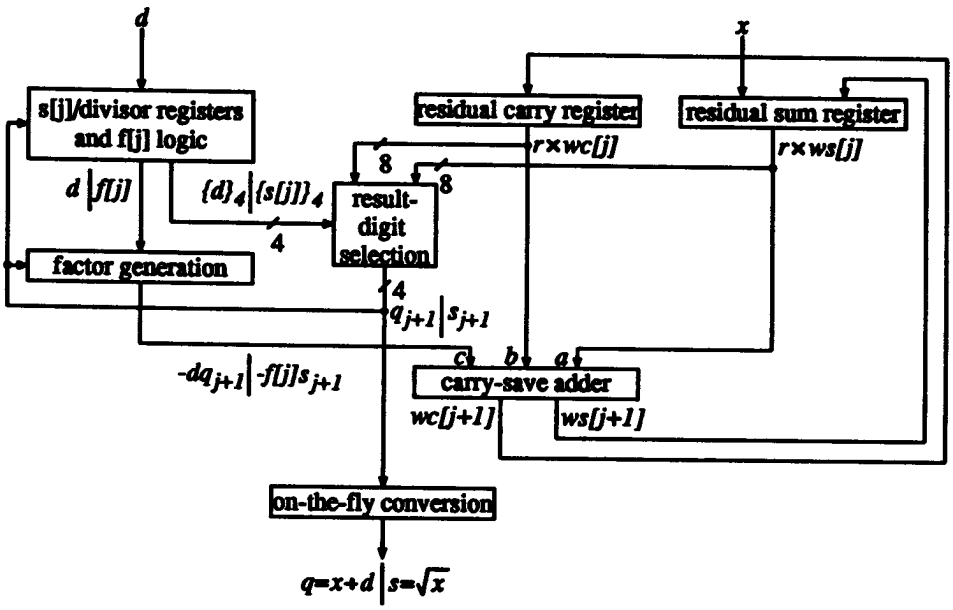


Figure 2: Radix-4 SRT divide/square root unit

plier, but the following elements are common to nearly all implementations:

- extra routing and storage
- lookup table for seed values
- hardware for constant subtraction/shifting
- last-digit rounding logic

The routing and storage are required to make divide/square root into atomic operations, bypassing the register file; the cost and complexity are highly implementation-dependent. The lookup table provides a starting approximation for the iterations; the more accurate the seed, the faster the execution of the algorithm. With quadratic convergence, an 8-bit seed requires 3 iterations to produce an accurate double-precision value, while a 16-bit seed only needs 2. However, the more accurate table will be 512 times larger than the less accurate one. Hardware to support the constant subtraction/shifting of the iterations provides additional speedup. Finally, multiplicative algorithms require special logic to insure accuracy in the last digit; this is a nontrivial problem, with several available solutions varying in cost, performance, and complexity [16]. It should be noted that divide/square root enhancements may negatively impact the latency of multiplication, since some of them lie on the critical path of the circuit.

3 Subtractive divide/square root

3.1 Algorithms

The subtractive divide/square root algorithms used in current microprocessors fall into the broad category of SRT methods. These algorithms use redundant representations of values and treat groups of consecutive bits as single higher-radix digits in order to enhance performance [5, 14].

Subtractive division computes the quotient q one digit at a time; the procedure for conventional long division is an algorithm of this type. Let $q[j]$ be the partial quotient at step j (where $q[n] = q$), and $w[j]$ the *residual* or *partial remainder*. The goal of the algorithm is to find the sequence of quotient bits which minimizes the residual. To compute $q = x \div d$ for n digit, radix-r values, set $w[0] = x$ and evaluate the recurrence

$$w[j+1] = rw[j] - dq_{j+1},$$

where q_{j+1} is the $j+1$st quotient digit. For square root computation, let the jth partial root be denoted by $s[j]$. To find $s = s[n] = \sqrt{x}$, set $w[0] = x$ and evaluate

$$w[j+1] = rw[j] - 2s[j]s_{j+1} - s_{j+1}^2 r^{-(j+1)}.$$

Define $f[j] = 2s[j] - s_{j+1}r^{-(j+1)}$; then

$$w[j+1] = rw[j] - f[j]s_{j+1}$$

has the same form as the division recurrence. In practice, $f[j]$ is simple to generate, which facilitates combined division and square root implementations.

3.2 Implementations

The efficient implementation of subtractive algorithms requires dedicated hardware resources. Figure 2 shows a basic radix-4 divide/square root unit. Most of the features enabling high performance are visible in the diagram. The residual is stored in redundant form as vectors of sum and carry bits; this enables the use of a low-latency carry-save adder to calculate the subtraction in the recurrence. The result-digit selection table returns the next quotient/square root digit on the basis of the residual and divisor/partial root values; the redundant result digit set allows truncated values to be used, keeping the table small and fast. Factor generation logic keeps all possible multiples of d or $f[j]$ and every result digit available at all times for multiplexer selection. Finally, on-the-fly conversion logic [5],

operating concurrently with computation and off of the critical path, is used to maintain updated values of $s[j]$ and $f[j]$, and to incrementally convert the partial result from redundant into conventional representation.

A radix-4 implementation produces 2 bits of the result for every iteration. Higher-radix units retire larger groups of bits at every step. Unfortunately, for radices greater than 8, the latency and cost of result digit selection and factor generation become prohibitive. One solution is to combine lower-radix stages into a single higher-radix unit. For example, two radix-4 dividers can be overlapped to enable radix-16 division [5], with only a modest increase in area and cycle time.

4 Case studies in divide/square root implementation

This section contains a series of experimental case studies which combine representative add-multiply structures with different practical implementations of division and square root, and presents the cost and performance impact of each choice. Area estimates and performance simulation are employed to achieve a quantitative comparison of the alternatives.

4.1 Selection of cases

The choice of add-multiply configuration largely determines the cost and performance properties of the FPU as a whole. The case studies are based on three representative configurations, as listed below, derived from actual machines in a sample of recent designs [16]. Every configuration reflects a different set of design prerogatives; in each case a maximum issue rate of one operation per cycle is assumed.

1. chained add and multiply
2. independent add and multiply
3. multiply-accumulate

Each add-multiply configuration is tested with four different divide/square root implementations. The basic list appears below, although there are a few exceptions for individual cases. Alternatives 1 and 3 represent practical implementations used in current systems, based on multiplicative and subtractive methods, respectively. Methods 2 and 4 are enhancements of these respective techniques. Together, these four implementations illustrate the current state-of-the-art and the effects of possible improvements. Performance figures are based as closely as possible on actual implementations, using data from the FPU designs informing the add-multiply models.

1. 8-bit seed Goldschmidt
2. 16-bit seed Goldschmidt
3. radix-4 SRT
4. radix-16 SRT

The Goldschmidt implementations are of the type presented in Section 2.2, utilizing hardware enhancements to the multiplier. Performance estimates are based on the Texas Instruments implementations [3], taking into account the particular topology of each multiplier. The multiply-accumulate case, which is based on the RS/6000, actually uses a Newton-Raphson iteration for division and square root as implemented by IBM.

Implementations of radix-4 and radix-16 divide/square root are of the type presented in Section 3.2. Most of the FPU's which form the basis of the add-multiply configurations implement radix-4 division. The latency and throughput of division and square root in these designs are used in simulation, since these figures reflect the constraints of the configuration and implementation technology. The radix-16 performance figures are also based on the given radix-4 figures. Again, the multiply-accumulate configuration is a special case since there is no actual radix-4 implementation available for reference.

4.2 Area comparisons

Estimating area in a way that holds true for different architectures is difficult because of basic differences in the implementation technology. Nevertheless, we shall attempt to give some basis for comparing the different implementations. Table 1 compares the size of the hardware required for division in the Weitek 3364 and Texas Instruments 8847 arithmetic coprocessors (not including shared components); the figures are based on measurements of microphotographs [8]. The chips have similar die sizes and device densities, and although the multiplication algorithms are different, both have two-pass arrays which take up approximately 22% of the chip area. In short, apart from their divide/square root implementation, these two chips have a lot in common. Now note how the area of the division hardware is no more than 5% of the chip size in either case. Also, the relative area requirements differ by little more than 1%. Although these figures represent only two particular designs, they suggest that 8-bit seed Goldschmidt and radix-4 SRT implementations are both economical, and that the area differences between them can be kept small.

Algorithm	radix-4 SRT	8-bit seed Goldschmidt
Device	Weitek 3364	TI 8847
Chip Area [mil^2]	148,000	156,000
Transistor Count	165,000	180,000
Mult. Area Share	22.1%	21.7%
Div./Sqrt. Area Share	3.9%	5.0%

Table 1: Area comparison of two divide/square root implementations

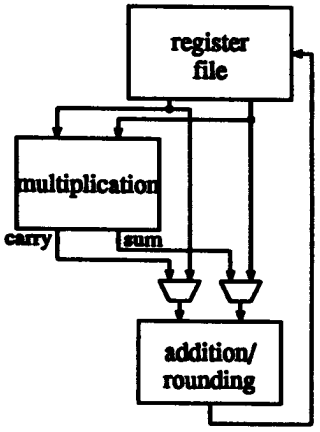
As for the more advanced implementations, case studies suggest that a radix-16 SRT unit need only be 45% larger than a radix-4 design in the same technology [5]. Determining the size of a 16-bit seed Goldschmidt implementation is more difficult given the available data, but a rough estimate indicates that it would be as much as *20 times larger* than the 8-bit seed implementation, given a straight ROM implementation of the seed table [16]. Even if the table size could be halved, the area would still be greater by more than a factor of 10. This calls into question the practicality of such an implementation.

4.3 Simulation

The benchmark used for performance is based on Givens rotation [6], chosen for its richness of divide and square root operations and importance in solving real problems. Scientific applications frequently require the solution of partial differential equations, and Givens rotations are a

key element of many such algorithms. They are also employed in signal processing, and the rotation and projection algorithms used in graphics and solid object modeling feature similar patterns of operations. Our FPU-level simulator computes the number of cycles required for each multiply-add and divide/square root combination to transform an arbitrary matrix into an upper triangular one using Givens rotations [16]. The schedule of operations has been optimized for each configuration. The majority of numerical applications for which this algorithm is suited feature square or overdetermined systems, and matrices having 100 or fewer rows. The test data, which are chosen accordingly, consist of 8 matrices ranging from 10-by-10 to 200-by-100 elements in size.

Each case study assumes that every one of the four divide/square root alternatives can be successfully incorporated into the existing add-multiply structure. In reality, some of the options may be precluded by limitations in the processor architecture or implementation. For example, the radix-16 divide/square root unit has a 20% longer cycle time than radix-4 design [5]. If the radix-4 unit only requires 80% of the available cycle time per iteration, then a radix-16 implementation may be feasible. If not, the required lengthening of processor cycle time will probably not be acceptable. Finally, some alternatives may be proscribed by area limitations.



Operation	Latency	Throughput
addition	4	3
multiplication	8	4

Figure 3: Chained add-multiply configuration

4.4 Case 1: chained add and multiply

The first case to be examined is typical of so-called *chained* add-multiply configurations, where the adder performs the final stages of the multiply operation. A block diagram of this structure and the latency and throughput of addition and multiplication appear in Figure 3. This configuration is usually associated with designs where economy of area is valued over raw floating-point performance. This motivates the re-use of hardware which makes the multiplier dependent on the adder. Typically, neither multiplication nor addition are fully pipelined, another economizing measure. With increasing transistor budgets, higher degrees of scalarity, and an increased emphasis on floating-point performance, the chained topology appears obsolescent, but is still represented by the Mips R4400 and DEC 21064. This particular example is inspired by the R4400 [10, 15].

Implementation	Latency	
	Divide	Square Root
8-bit seed Goldschmidt	35	51
16-bit seed Goldschmidt	28	40
radix-4 SRT	36	36
radix-16 SRT	23	23

Table 2: Divide/square root performance of chained implementations

The latencies of division and square root for the different implementation alternatives are given in Table 2. The third implementation (in boldface), is closest to the actual configuration of the Mips R4400. In actuality, division is performed by a radix-4 divider, while square root computation occurs in the floating-point adder using a radix-2 algorithm. Needless to say, this is a stumbling block for algorithms using square root at all. For the sake of uniform comparison with other configurations, we use the latency for radix-4 division from the Mips R4400 for both operations.

The radix-16 latencies are computed as follows. Computing 53 quotient/root bits in radix-4 requires a minimum $\lceil 53/2 \rceil = 27$ cycles; since the actual latency is 36 cycles, there is a 9 cycle overhead which is an artifact of the particular technology and FPU configuration of the Mips R4400. To estimate the performance of a radix-16 design, we find the minimum number of cycles required, $\lceil 53/4 \rceil = 14$, and add the 9 cycle overhead from the radix-4 case to obtain a latency of 23 cycles.

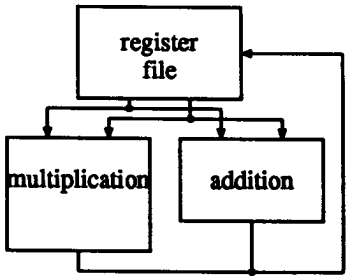
Implementation	Max	Min	Avg
8-bit seed Goldschmidt	0.0	0.0	0.0
16-bit seed Goldschmidt	11.0	2.1	5.8
radix-4 SRT	56.2	12.9	34.2
radix-16 SRT	82.6	12.9	42.3

Table 3: Improvement in execution time [%], by implementation, for chained configuration

Table 3 shows the improvement in execution time of the Givens rotation benchmark for each divide/square root implementation. The 8-bit seed Goldschmidt implementation is used as a performance baseline. Note the modest improvement effected by the transition from 8-bit seed to 16-bit seed Goldschmidt, as compared to the dramatic difference provided by the radix-4 and radix-16 techniques. This is due to the enhanced parallelism of the latter designs and the ability to overlap steps in the computation.

4.5 Case 2: independent add and multiply

In the second type of add-multiply configuration, captured in Figure 4, addition and multiplication are completely independent of each other. Performance is the highest priority, and cost less of an object. Not only are the adder and multiplier independent and fully pipelined, but their latencies are matched and only two cycles long



Operation	Latency	Throughput
addition	2	1
multiplication	2	1

Figure 4: Independent add-multiply configuration

each. The particular chip which this configuration is based on, the HP PA7200 [1, 7], also has a very short cycle time. Other designs with similar topologies include the Sun UltraSPARC, the Mips 10000, Intel P6, and DEC 21164.

Implementation	Latency	
	Divide	Square Root
8-bit seed Goldschmidt	9	13
16-bit seed Goldschmidt	7	10
radix-4 SRT	**15**	**15**
radix-16 SRT	8	8

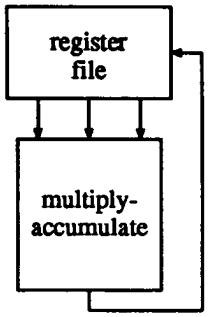
Table 4: Divide/square root performance of independent implementations

The division and square root latencies are shown in Figure 4. For radix-4 division, there is a simple but powerful optimization in effect. In the implementation technology of the HP PA7200, the cycle time of the divide/square root unit is so short compared to the latency of the multiplier array that its clock runs at twice the frequency of the rest of the system. Thus it requires only $\lceil 53/(2\times2) \rceil = 14$ cycles with one cycle of overhead. The radix-16 design, if it could implemented with a comparable iteration delay, would therefore require $\lceil 53/(4\times2) \rceil + 1 = 8$ cycles. Even with these optimizations, the extremely fast multiplication makes the Goldschmidt implementations competitive in latency with the subtractive ones.

Implementation	Max	Min	Avg
8-bit seed Goldschmidt	0.0	0.0	0.0
16-bit seed Goldschmidt	9.9	1.6	5.0
radix-4 SRT	20.9	7.2	15.2
radix-16 SRT	46.0	7.2	23.4

Table 5: Improvement in execution time [%], by implementation, for independent configuration

The execution time improvement figures shown in Table 5 reinforce the effects of enhanced parallelism noted earlier. Although the lower multiplication latency cuts into the benefits of the radix-4 and radix-16 implementations, the performance advantages are still significant. The shift in balance between multiplication and addition latency from the chained configuration mean that the difference between 16-bit seed Goldschmidt and 8-bit seed Goldschmidt is also smaller than before.



Operation	Latency	Throughput
multiply-accumulate	2	1

Figure 5: Multiply-accumulate configuration

4.6 Case 3: multiply-accumulate

The multiply-accumulate structure represents a bid for high-performance floating-point, but with a different design philosophy from the independent configuration. Multiplication and addition are coupled, not unlike in the chained configuration, but a large amount of hardware has been devoted to bring the latency of these operations to an absolute minimum. Furthermore, addition and multiplication are performed as a single operation. The multiply-accumulate unit in this example, shown in Figure 5, is based on the IBM RS/6000 series [11, 20, 19] and can perform a multiply-add instruction in the same number of cycles it takes the HP PA7200 to perform just one of the operations. This configuration is capable of very high performance, particularly for algorithms which can be arranged to suit its topology. The Mips 8000 and HP PA800 have similar add-multiply hardware.

Implementation	Latency	
	Divide	Square Root
8-bit seed Newton-Raphson	**19**	**22**
16-bit seed Newton-Raphson	14	17
radix-4 SRT	15	15
radix-16 SRT	8	8

Table 6: Divide/square root performance of multiply-accumulate implementations

The IBM RS/6000 series uses unique algorithms for the Newton-Raphson iterations to accommodate the structure of the multiply-accumulate unit. Division and square root latencies for the 8-bit seed Newton-Raphson implementation in Table 6 are identical to the actual processor. The 16-bit seed Newton-Raphson figures are obtained from estimates based on available information about the division and square root algorithms [9]. The POWER2 series of processors actually has two identical floating point units, each centered on a multiply-accumulate structure. We have decided to avoid the complexity of scheduling operations for two floating-point units and chosen to simulate the

behavior of one in isolation, as in the original POWER series.

When it comes to the subtractive implementations, there is a gap in the available data, since the IBM RS/6000 has only multiplicative division and square root. As an approximation, it has been assumed that the divide/square root circuits from the HP PA7200 can be implemented alongside the IBM RS/6000 multiply-accumulate unit with the same performance values; this seems reasonable since the cycle time of the RS/6000 is actually longer then for the PA7200.

Implementation	Max	Min	Avg
8-bit seed Newton-Raphson	0.0	0.0	0.0
16-bit seed Newton-Raphson	19.7	4.9	11.6
radix-4 SRT	68.4	22.4	48.3
radix-16 SRT	125.7	22.5	68.8

Table 7: Improvement in execution time [%], by implementation, for multiply-accumulate configuration

From the performance figures in Table 7, it is clear that even the multiply-accumulate configuration can benefit from the parallelism of subtractive implementations. In fact, since the latency of multiplicative division and square root in cycles is slightly longer than for the independent configuration, the benefit is even more apparent.

5 Analysis

Using the data accumulated in the case studies, it is possible to draw some general conclusions about the area/performance efficiency of the different divide/square root implementations. It is important to tread lightly on the issue of comparing the performance of designs with different add-multiply configurations, for several reasons. The choice of a given configuration tends to place a design within a distinct cost/performance category. Different machines also draw the line between cycle time and cycle utility in different ways [16]. Finally, the machines in our sample represent different technology generations. Although the data in this study speak clearly across configuration boundaries, it is important to keep the above qualifications in mind when summarizing the results.

5.1 General trends

The single biggest factor in performance improvement, for all configurations, is the increased parallelism of the subtractive implementations. Across the various configurations, the radix-4 SRT implementation outperforms the 8-bit seed Goldschmidt version in every instance, even with inferior per-operation latencies. It also dominates the 16-bit seed Goldschmidt in the overwhelming majority of cases, in spite of a consistent latency disadvantage. Even more striking is the dramatic improvement in switching from radix-4 to radix-16, compared to the relatively paltry effect of speeding up the Goldschmidt iteration with a larger seed value.

5.2 Evaluation of specific methods

Table 8 shows the maximum, minimum, and average improvement of the benchmark execution time across all configurations. The maximum values are more important since they generally represent the more interesting types of problems -- namely small, overdetermined systems.

Implementation	% Improvement		
	Max	Min	Avg
8-bit seed multiplicative	0.0	0.0	0.0
16-bit seed multiplicative	19.7	1.6	7.5
radix-4 SRT	69.4	7.2	32.6
radix-16 SRT	125.7	7.2	44.8

Table 8: Cumulative execution time improvement [%] due to different divide/square root implementations

Until now, the relative cost of the implementations has been mentioned only in passing; Table 9 dramatizes the differences by displaying the approximate area of each configuration as a factor of the 8-bit seed multiplicative case. Recall that the multiplicative implementations for the multiply-accumulate configuration are actually versions of the Newton-Raphson method, not Goldschmidt's algorithm; although the cost of the Newton-Raphson implementations is likely to be slightly cheaper, we will treat them as equivalent for the sake of simplicity.

Implementation	Area Factor
8-bit seed multiplicative	1
16-bit seed multiplicative	20
radix-4 SRT	1.3
radix-16 SRT	1.8

Table 9: Relative cost of different divide/square root implementations

Clearly, the 8-bit seed multiplicative implementations have the lowest cost of the four alternatives for each case. However, the benchmark performance is also the worst of the implementations considered, from 19.7% to 1.6% slower than the next slowest alternative.

The 16-bit seed multiplicative implementations show a staggering increase in area. This is a result of the exponential growth of the seed lookup table with the number of bits of the initial guess. Unfortunately, the number of iterations required only decreases at a linear rate [16], which leads to a very modest performance improvement, less than 20% in the very best case and much lower on average. This type of implementation is an extremely cost-ineffective way to perform division and square root and is probably downright infeasible in many situations.

Radix-4 SRT divide/square root gives up to 69.4% better benchmark performance than the 8-bit seed multiplicative implementations, and never less than a 7.2% improvement. Yet the cost is only 30% greater. It also well outperforms the 16-bit seed multiplicative implementations on average. The radix-4 implementation type is arguably the most efficient balance of area and performance.

By far the swiftest of the implementation methods examined, radix-16 divide/square root has a maximum performance 125.7% to 46.0% faster than corresponding 8-bit multiplicative versions, but is still less then twice as costly in area, and more than ten times cheaper than the 16-bit seed multiplicative alternatives.

6 Recommendations

At the present time, radix-4 divide/square root appears to be the most solid choice of implementation, providing high performance at a reasonable cost. In fact, a significant proportion of the most recent microprocessor designs have chosen this alternative, with multiplicative designs in the minority. To the authors' knowledge, no current microprocessors implement radix-16 divide/square root, although with transistor budgets on the rise, this appears to be an increasingly appealing and feasible option for squeezing the most performance out of an FPU.

Implementations of 8-bit seed Goldschmidt or Newton-Raphson make sense in cases where building a separate divide/square root unit is not feasible for cost or other reasons. This applies either to very inexpensive designs or machines which invest a large amount of hardware in fast multiplication and division. The large table size required casts serious doubts on the practicality of 16-bit seed multiplicative implementations. No commercial designs known to the authors implement it, and even if the required area were available, it would be better spent on a parallel divider.

Current trends in microprocessor implementation include ever larger transistor budgets and increasing levels of parallelism. Designers are increasingly less likely to worry about conserving area than to puzzle over how to use available space efficiently. Subtractive methods, with their independent operation, are in a better position to exploit higher degrees of scalarity than multiplicative techniques, which serialize multiplication, division, and square root computation. Indeed, the latest generation of microprocessors is dominated by chips with subtractive implementations, including the Sun UltraSPARC, Mips R10000, Intel P6, and HP PA8000. As the need to conserve area and devices becomes less urgent, one of the primary motivations for multiplicative methods begins to recede.

Finally, multiplicative implementations are always more or less intimately linked to the design of the floating-point multiplier, possibly compromising its performance. Subtractive techniques decouple division and square root from multiplication and provide the possibility of independently optimizing the implementations. Even if multiplication, division, and square root are not a good combination, perhaps, as has been suggested, multiplication and addition are a natural pair. By this reasoning, a combination of multiply-accumulate units and separate divide/square root units, as in the HP PA8000, may represent the floating-point architecture of the future.

Acknowledgements

This research is supported in part by the National Science Foundation under contract CCR-9257280. Peter Soderquist was supported by a Fellowship from the National Science Foundation. Miriam Leeser is supported in part by an NSF Young Investigator Award. We would like to thank Bard Bloom and Adam Bojanczyk of Cornell University for reading and commenting on an early version of this paper.

References

[1] Tom Asprey et al. Performance features of the PA7100 microprocessor. *IEEE Micro*, pages 22--35, June 1993.

[2] Daniel E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Trans. on Computers*, pages 925--934, October 1968.

[3] Henry M. Darley et al. Floating-point/integer processor with divide and square root functions. U.S. Patent 4,878,190, October 1989.

[4] Milos D. Ercegovac, Tomas Lang, and Paolo Montuschi. Very high radix division with selection by rounding and prescaling. In *Proc. 11th IEEE Symposium on Computer Arithmetic*, pages 112--119. IEEE, June 1993.

[5] Milos D. Ercegovac and Tomas Lang. *Division and Square Root: Digit Recurrence Algorithms and Implementations*. Kluwer Academic Publishers; Boston, 1994.

[6] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins Univ. Press; Baltimore, second edition, 1989.

[7] Linley Gwennap. PA-7200 enables inexpensive MP systems: HP's next-generation PA-RISC also contains unique "assist" cache. *Microprocessor Report*, January 1994.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers; San Mateo, CA, 1990. Appendix A: Computer Arithmetic by David Goldberg.

[9] Peter W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Jour. of Res. and Dev.*, pages 111--119, January 1990.

[10] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The Mips R4000 processor. *IEEE Micro*, pages 10--22, April 1992.

[11] R. K. Montoye, Hokenek E., and S. L. Runyon. Design of the IBM RISC System/6000 floating-point execution unit. *IBM Jour. of Res. and Dev.*, pages 59--70, January 1990.

[12] Stewart F. Oberman and Michael J. Flynn. Design issues in floating-point division. Technical Report CSL-TR-94-647, Stanford University Departments of Electrical Engineering and Computer Science, Stanford, CA, December 1994.

[13] Victor Peng, Sridhar Samudrala, and Moshe Gavrielov. On the implementation of shifters, multipliers, and dividers in VLSI floating point units. In *Proc. 8th IEEE Symposium on Computer Arithmetic*, pages 95--102. IEEE, May 1987.

[14] Norman R. Scott. *Computer Number Systems and Arithmetic*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[15] Satya Simha. *R4400 Microprocessor: Product Information*. MIPS Technologies, Inc., Mountain View, CA, September 1993.

[16] Peter Soderquist and Miriam Leeser. Area and performance tradeoffs in floating-point division and square root implementations. Technical Report EE-CEG-94-5, Cornell School of Electrical Engineering, Ithaca, NY, December 1994.

[17] C. C. Stearns. Subtractive floating-point division and square root for VLSI DSP. In *European Conf. Circuit Theory and Design*, pages 405--409, September 1989.

[18] George S. Taylor. Radix 16 SRT dividers with overlapped quotient selection stages. In *Proc. 7th IEEE Symposium on Computer Arithmetic*, pages 64--71. IEEE, June 1985.

[19] Steven W. White. POWER2: Architecture and performance. In *Digest of Papers: COMPCON Spring 1994*, pages 384--388. IEEE, February 1994.

[20] Steven W. White et al. How does processor MHz relate to end-user performance? Part 1: Pipelines and functional units. *IEEE Micro*, pages 8--16, August 1993.