AN ANALYSIS OF DIVISION ALGORITHMS AND IMPLEMENTATIONS

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-95-675

July 1995

This work was supported by NSF under contract MIP93-13701.

AN ANALYSIS OF DIVISION ALGORITHMS AND IMPLEMENTATIONS

by

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-95-675

July 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055
pubs@shasta.stanford.edu

Abstract

Floating-point division is generally regarded as a low frequency, high latency operation in typical floating-point applications. However, the increasing emphasis on high performance graphics and the industry-wide usage of performance benchmarks forces processor designers to pay close attention to all aspects of floating-point computation. Many algorithms are suitable for implementing division in hardware. This paper presents four major classes of algorithms in a unified framework, namely digit recurrence, functional iteration, very high radix, and variable latency. Digit recurrence algorithms, the most common of which is SRT, use subtraction as the fundamental operator, and they converge to a quotient linearly. Division by functional iteration converges to a quotient quadratically using multiplication. Very high radix division algorithms are similar to digit recurrence algorithms, but they incorporate multiplication to reduce the latency. Variable latency division algorithms reduce the average latency to form the quotient. These algorithms are explained and compared in this work. It is found that for low-cost implementations where chip area must be minimized, digit recurrence algorithms are suitable. An implementation of division by functional iteration can provide the lowest latency for typical multiplier latencies. Variable latency algorithms show promise for simultaneously minimizing average latency while also minimizing area.

Key Words and Phrases: Floating-point, division, algorithms, SRT, functional iteration, very high radix, variable latency, computer arithmetic

Contents

List of Figures

List of Tables

1 Introduction

In recent years computer applications have increased in their computational complexity. The industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay particular attention to implementation of the floating-point unit, or FPU. Special purpose applications, such as high performance graphics rendering systems, have placed further demands on processors. High speed floating-point hardware is a requirement to meet these increasing demands.

Modern applications comprise several floating point operations including addition, multiplication, and division. In recent FPUs, emphasis has been placed on designing ever-faster adders and multipliers, with division receiving less attention. Typically, the range for addition latency is 2 to 4 cycles, and the range for multiplication is 2 to 8 cycles. In contrast, the latency for double precision division in modern FPUs ranges from less than 8 cycles to over 60 cycles [24]. A common perception of division is that it is an infrequent operations whose implementation need not receive high priority. However, it has been shown that ignoring its implementation can result in significant system performance degradation for certain applications [27]. While the methodology for designing efficient high-performance adders and multipliers is well-understood, the design of dividers still remains a serious design challenge, often viewed as a "black-art" among system designers. Extensive theory exists describing the theory of division. However, the design space of the algorithms and implementations is large, due to the large number of parameters involved. Furthermore, deciding upon an optimal design depends heavily on its requirements.

Division algorithms can roughly be divided into four classes. The first and most common class is digit recurrence. The majority of commercial implementations are based on this class. Digit recurrence algorithms form a quotient one digit at a time, in a manner similar to traditional paper-and-pencil division. SRT division is a widely used variation of digit recurrence division, named for Sweeney, Robinson, and Tocher who independently proposed the algorithm. In each step of the algorithm a multiple of the divisor is subtracted from the dividend or partial remainder. The latency of forming a quotient using digit recurrence is linear with the length of the operands. The major advantages of digit recurrence are simplicity of implementation and the availability of a final remainder at the completion of the computation. The disadvantage is the linear convergence.

The second class of algorithms is functional iteration. These algorithms represent the division or reciprocal operation as a function, and use function-solving techniques such as the Newton-Raphson equation to converge to the quotient or reciprocal. Several commercial implementations have been based on this class of algorithm. Implementations of the Newton-Raphson equation require approximately two multiplications per step of the iteration, rather than a simple subtraction. The advantage of functional iteration is that it converges to the quotient or reciprocal faster than linearly, with typical implementations converging quadratically. The disadvantages are that the complexity per step of the iteration is complex and that a final remainder is not readily available.

The third class of algorithms is known as very high radix division. These algorithms are essentially extensions of simple digit recurrence algorithms to higher radices. The term very high radix typically refers to division implementations that retire more than 10 quotient bits per iteration step. These algorithms incorporate multiplication into the iteration step to simplify the formation of quotient digits and divisor multiples. The advantages of very high radix division are that more quotient bits are retired per step than simple digit recurrence and a final remainder is available. The iteration step requires more complexity than simple digit recurrence, but can require less than functional iteration. The disadvantage is linear convergence to the quotient. Only one commercial implementation has been based on this class of algorithm, the Cyrix 83D87 arithmetic coprocessor [2]. As the desire for higher performance division increases, this class could become more common.

The fourth class of algorithms is variable latency algorithms. Whereas the first three classes always have a fixed latency regardless of the input operands, variable latency algorithms have the ability to produce a result faster than the worst case latency, depending on the operands. These algorithms allow for increased average division performance without all of the complexity required for increased worst case division performance. However, a system that incorporates such a divider must be able to manage variable latency functional units. Therefore, such a system is typically more complex. One commercial processor implementation uses a variable latency divider, the self-timed divider in the Hal Sparc64 microprocessor [24]. However, variable latency functional units hold promise for future processors. As systems become more complex and allow for out-of-order execution and completion, variable latency dividers will be a cost-effective method for improving floating-point performance.

In the past, others have presented summaries of specific classes of division algorithms and implementations. Flynn [19] discusses the theory and methodology of multiplication-based division algorithms. Atkins [1] is the first major analysis of SRT algorithms. Tan [38] derives and presents the theory of high-radix SRT division, along with an analytic method of implementing SRT look-up tables. Soderquist [35] presents performance and area tradeoffs in divider design in the context of a specialized application. Ercegovac and Lang [9] present a thorough coverage of SRT algorithms. This study synthesizes the fundamental aspects of these and other works, in order to clarify the division design space. The four classes of division algorithms are presented and analyzed in terms of the three major design parameters: latency in system clock cycles, cycle time, and area. Other issues related to the implementation of division in actual systems are also presented. Throughout this work, the majority of the discussion is devoted to division, but the theory of square-root computation is an extension of the theory of division, and most of the analyses and conclusions for division can also be applied to the design of square-root units. Further details regarding square-root computation are presented in appendix B.

The remainder of this paper is organized as follows. Section 2 presents digit recurrence algorithms. Section 3 presents functional iteration. Sections 4 and 5 discuss very high radix and variable latency algorithms. Section 6 compares the algorithm classes. Section 7 is the conclusion.

2 Digit Recurrence Algorithms

The simplest and most widely implemented class of division algorithms is digit recurrence. Digit recurrence algorithms retire a fixed number of quotient bits in every iteration. Implementations of digit recurrence algorithms are typically of low complexity, utilize small area, and have relatively large latencies. The fundamental issues in the design of a digit recurrence divider are the radix, the choice of allowed quotient digits, and the representation of the intermediate remainder. The radix determines how many bits of quotient are retired in an iteration, which fixes the division latency. Larger radices can reduce the latency, but increase the time for each iteration. Judicious choice of the allowed quotient digits can reduce the time for each iteration, but with a corresponding increase in complexity and hardware. Similarly, different representations of the intermediate remainder can reduce iteration time, with corresponding increases in complexity.

Various techniques have been proposed for further increasing division performance, including staging of simple low-radix stages, overlapping sections of one stage with another stage, and prescaling the input operands. All of these methods introduce tradeoffs in the time/area design space. This section introduces the principles of digit recurrence division, along with an analysis of methods for increasing the performance of digit recurrence implementations. Also presented are techniques for handling final rounding and conversion in an efficient manner.

2.1 Definitions

Digit recurrence algorithms use subtractive methods to calculate quotients one digit per iteration. For the purposes of this paper, the input operands are assumed to be represented in a normalized floating-point format with fractional significands of n radix-r digits in sign-magnitude representation. The algorithms presented in this work are applied only to the magnitudes of the significands of the input operands. Techniques for computing the resulting exponent and sign are straightforward and are not discussed here. The most common format found in modern computers is the IEEE 754 standard for binary floating-point arithmetic [20]. This standard defines single and double precision formats, where n=24 for single precision and n=53 for double precision. The significand consists of a normalized quantity, with an explicit or implicit leading bit to the left of the implied binary point.

Digit recurrence algorithms can be further divided into *restoring* and *nonrestoring* division. Restoring division is similar to the familiar paper and pencil division. When dividing two n-bit numbers, the division can require up to 2n + 1 adds. Nonrestoring division eliminates the restoration cycles, and thus only requires up to n adds. This can be accomplished by allowing negative values of the quotient as well as positive values. In this way, small errors in one iteration can be corrected in subsequent iterations. *SRT division* is the name of the most common form of nonrestoring division. This class of division was named for Sweeney, Robertson, and Tocher, who independently proposed similar nonrestoring division algorithms [40]. The remainder of this section presents aspects relevant to SRT division.

2.2 Recurrence

For division, the quotient can be computed as follows:

$$q = \frac{dividend}{divisor}$$

Accordingly, this expression can be rewritten as:

$$dividend = q \times divisor + remainder$$

such that

$$|remainder| < |divisor| \times ulp \quad \text{and} \quad sign(remainder) = sign(dividend)$$

where the input operands are given by *dividend* and *divisor*, and the results are q and *remainder*. The precision of the quotient is defined by the unit in the last position (ulp), where for an integer quotient $ulp = 1$, and for a fractional quotient $ulp = r^{-n}$, assuming a radix-r representation and an n-digit quotient.

The following recurrence is used at every iteration:

$$P_0 = dividend \tag{1}$$
$$P_{j+1} = rP_j - q_{j+1} divisor \tag{2}$$

where P_j is the partial remainder, or residual, at iteration j.

In each iteration, one digit of the quotient is determined by the quotient-digit selection function:

$$q_{j+1} = SEL(P_j, divisor) \tag{3}$$

In order for the next residual P_{j+1} to be bounded, the value of the quotient digit is chosen such that

$$|P_{j+1}| < divisor$$

The *remainder* can be computed from the final residual by:

$$remainder = \begin{cases} P_n \times r^{-n} & \text{if } P_n \geq 0 \\ (P_n + divisor) \times r^{-n} & \text{if } P_n < 0 \end{cases}$$

Furthermore, the quotient has to be adjusted when $P_n < 0$ by subtracting r^{-n}.

2.3 Implementation of Basic Scheme

A block diagram of an implementation of the basic recurrence is shown in figure 1. The critical path of the topology is shown by the dotted line.

As can be noted from equations 1 and 2, each iteration of the recurrence comprises the following steps:

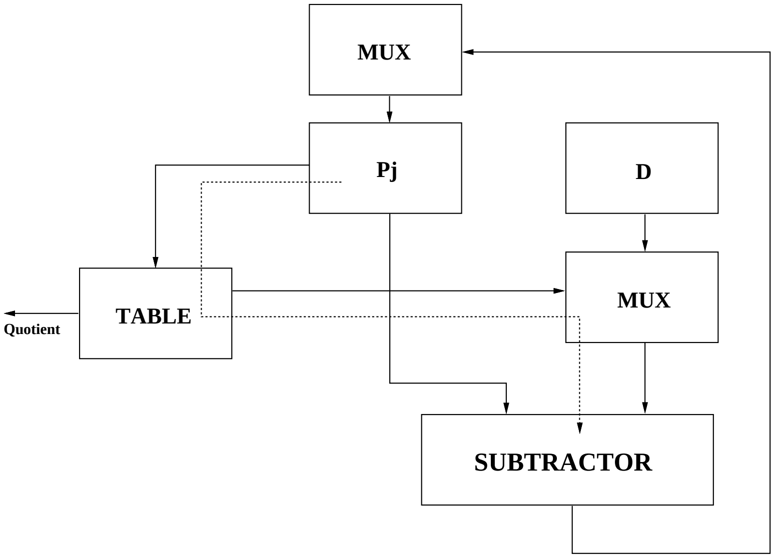


Figure 1: Basic SRT Topology

- Determine next quotient digit q_{j+1} by the quotient-digit selection function.
- Generate the product $q_{j+1} \times divisor$.
- Subtract $q_{j+1} \times divisor$ from $r \times P_j$

Each of these components can contribute to the overall cost and performance of the algorithm. Depending on certain parameters of the algorithm, the execution time can vary widely. To tradeoff cost for performance, these parameters can be studied and appropriately chosen.

2.3.1 Choice of Radix

The fundamental method of decreasing the overall latency (in machine cycles) of the algorithm is to increase the radix r of the algorithm. It is convenient to choose the radix to be a power of 2. In this way, the product of the radix and the partial remainder can be formed by shifting. Assuming the same quotient precision, the number of iterations of the algorithm required to compute the quotient is reduced by a factor k when the radix is increased from r to r^k. For example, a radix 4 algorithm retires 2 bits of quotient in every iteration. Increasing to a radix 16 algorithm will allow for retiring 4 bits in every iteration, for a 2X reduction in latency. This reduction does not come for free. As the radix increases, the quotient-digit selection becomes more complicated. It can be seen from figure 1 that quotient selection is on the critical path of the basic algorithm. The cycle time of the divider is defined as the minimum time to complete this critical path. The result of this is that the number of cycles may have been reduced due to the increased radix. However, the time per cycle may have increased. As a result, the total time required to compute an n bit quotient will not be reduced by the factor k. Additionally, the generation of all required divisor multiples may become impractical or infeasible for higher radices. Thus, these two factors can offset some or possibly all of the performance gained by increasing the radix.

2.3.2 Choice of Quotient Digit Set

In digit recurrence algorithms, some range of digits is decided upon for the allowed values of the quotient in each iteration. The simplest case is where, for radix r, there are exactly r allowed values of the quotient. However, to increase the performance of the algorithm, it is desirable to utilize a *redundant digit set*. Such a digit set can be composed of symmetric signed-digit consecutive integers, where the maximum digit is a. The digit set is made redundant by having more than r digits in the set. In particular,

$$q_j \in \mathcal{D}_a = \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a-1, a\}$$

Thus, to make a digit set redundant, it must contain more than r consecutive integer values including zero, and thus a must satisfy

$$a \geq \lceil r/2 \rceil$$

The redundancy of a digit set is determined by the value of the redundancy factor ρ, which is defined as

$$\rho \;=\; \frac{a}{r-1}, \quad \rho > \frac{1}{2}$$

Typically, signed-digit representations have $a < r-1$. When $a = \lceil \frac{r}{2} \rceil$, the representation is called *minimally redundant*, while that with $a = r-1$ is called *maximally redundant*, with $\rho = 1$. A representation is known as *non-redundant* if $a = (r-1)/2$, while a representation where $a > r-1$ is called *over-redundant*. For the next residual P_{j+1} to be bounded when a redundant quotient digit set is used, the value of the quotient digit must be chosen such that

$$|P_{j+1}| \;<\; \rho \times \mathit{divisor}$$

The design tradeoff can be noted from this discussion. By using a large number of allowed quotient digits a, and thus a large value for ρ, the complexity and latency of the quotient selection function can be reduced. However, choosing a smaller number of allowed digits for the quotient simplifies the generation of the multiple of the divisor. Multiples that are powers of two can be formed by simply shifting. If a multiple is required that is not a power of two (e.g. three), an additional operation such as addition may also be required. This can add to the complexity and latency of generating the divisor multiple. The complexity of the quotient selection function and that of generating multiples of the divisor must be balanced.

After the redundancy factor ρ is chosen, it is possible to derive the quotient selection function. A *containment condition* can be derived which allows for determining selection

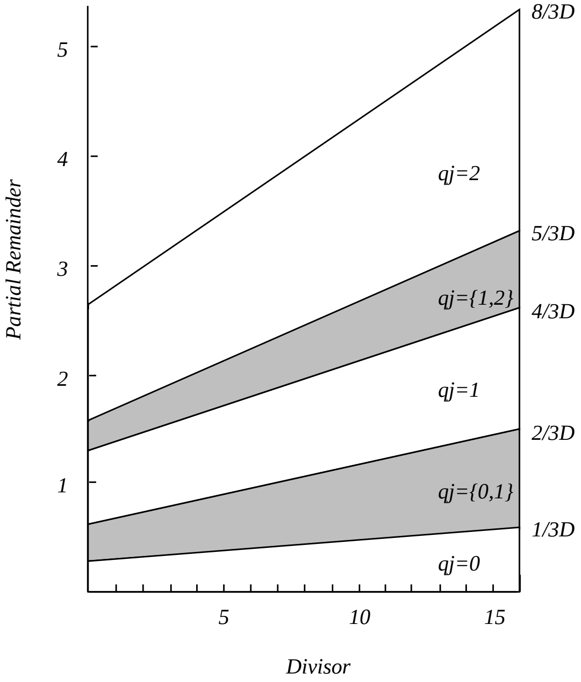


Figure 2: P-D diagram for radix-4

interval expressions. A selection interval is the region in which a particular quotient digit can be chosen. These expressions are given by

$$U_k = (\rho + k)d \qquad L_k = (-\rho + k)d$$

where U_k (L_k) is the largest (smallest) value of rP_j such that it is possible for $q_{j+1} = k$ to be chosen and still keep the next partial remainder bounded. The *P-D diagram* is a useful visual tool when designing a quotient-digit selection function. It has as axes the shifted partial remainder and the divisor. The selection interval bounds U_k and L_k are drawn as lines starting at the origin with slope $\rho + k$ and $-\rho + k$, respectively. A P-D diagram is shown in figure 2 with $r = 4$ and $a = 2$. The shaded regions are the overlap regions where more than one quotient digit may be selected.

2.3.3 Residual Representation

The residual can be represented in either of two different forms, either *redundant* or *nonredundant* forms. Conventional 2's complement representation is an example of a *nonredundant* form, while carry-save 2's complement representation is an example of a *redundant* form. Each iteration requires a subtraction to form the next residual. If this residual is in a nonredundant form, then this operation would require a full-width adder requiring carry propagation. Consequently, the cycle time would be large.

If the residual is computed in a redundant form, a carry-free adder can be used in the recurrence, minimizing the cycle time. However, the quotient-digit selection, which is

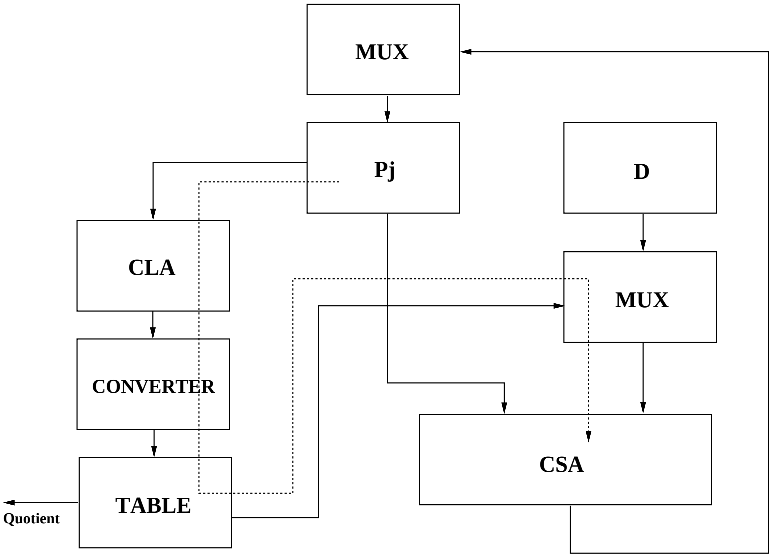


Figure 3: Enhanced SRT Topology

a function of the shifted residual, becomes more complex. Additionally, twice as many registers are required to store the residual between iterations. Finally, if the remainder is required from the divider, the last residual will have to be converted to a conventional representation. At a minimum, it is necessary to be able to determine the sign of the final remainder in order to implement a possible quotient correction step, as discussed previously. A block diagram of a divider with redundant residual and quotient-digit set is shown in figure 3.

2.3.4 Quotient-Digit Selection Function

Critical to the performance of a divider is the efficient implementation of the quotient selection function. If a redundant representation is chosen for the residual, the residual is not known exactly, and neither is the exact next quotient digit. However, by using a redundant quotient digit set, the residual does not need to be known exactly to select the next quotient digit. It is only necessary to know the residual well enough to know which range in figure 2 it lies. The selection function is realized by approximating the residual P_j and *divisor* to compute q_{j+1}. This is typically done by means of a lookup table. The challenge in the design is deciding how many bits of P_j and *divisor* are needed, while simultaneously minimizing the complexity of the table.

The standard method of designing SRT lookup tables is through the use of *selection constants*. The divisor range is separated into equal intervals $[d_i, d_{i+1})$ such that:

$$d_1 = \frac{1}{2}, \qquad d_{i+1} = d_i + 2^{-\delta}$$

The interval can be represented by the δ most significant bits of the divisor. Within each interval, a quotient digit is selected by the selection constants $m_k(i)$ as given by:

$$\text{for } d \in [d_i, d_{i+1}),\ q_{j+1} = k \text{ if } m_k(i) \leq rP_j \leq m_{k+1}(i) - r^{-n}$$

The set of selection constants for a given value of k form a series of steps that connect the overlap regions in figure 2. The greater the redundancy factor of the implementation, the wider the steps can be, and the fewer bits of divisor and/or residual that are needed.

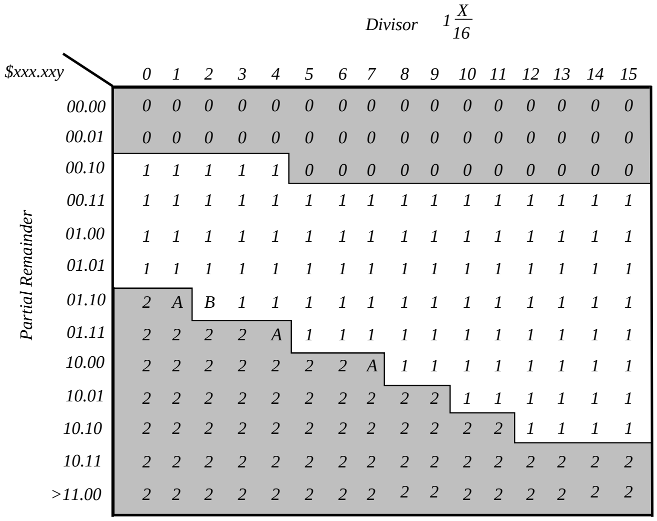
Let $\hat{d}$ be an estimate of the divisor using the δ most significant bits of the true divisor and $\hat{P}_j$ be an estimate of the partial remainder using the c most significant bits of the true partial remainder. To determine the minimum values for δ and c, it is necessary to consider the uncertainty region in the resulting estimates $\hat{d}$ and $\hat{P}_j$. The estimates will have errors ϵ_d and ϵ_p for the divisor and partial remainder estimates respectively. Because the estimates of both quantities are formed by truncation, ϵ_d and ϵ_p can each be 1 *ulp*. Additionally, if the partial remainder is kept in a carry-save form, ϵ_p can be as much as 2 *ulps*. This is due to the fact that both the sum and the carry values have been truncated, and each can have a 1 *ulp* error. When the two are summed to form a nonredundant estimate of the partial remainder, the actual error can be 2 *ulps*. The worst case ratios of $\hat{P}_j$ and $\hat{d}$ must be checked for all possible values of the estimates. For a two's complement representation of the partial remainder, ϵ_d and ϵ_p are always positive, and the maximum and minimum values of the ratio are given by:

$$maximum = \begin{cases} \frac{\hat{P}_j+\epsilon_p}{\hat{d}} & \text{if } P_j \geq 0 \\ \frac{\hat{P}_j}{\hat{d}} & \text{if } P_j < 0 \end{cases}$$

$$minimum = \begin{cases} \frac{\hat{P}_j}{\hat{d}+\epsilon_d} & \text{if } P_j \geq 0 \\ \frac{\hat{P}_j+\epsilon_p}{\hat{d}+\epsilon_d} & \text{if } P_j < 0 \end{cases}$$

It is necessary for the minimum and maximum values of the ratio to lie in regions such that both values can take on the same quotient digit. If the values require different quotient digits, then the uncertainty region is too large for the table configuration. Several iterations over the design space may be necessary to determine an optimal solution for the combination of radix, redundancy, values of δ and c, and error terms ϵ_p and ϵ_d.

Having performed the analysis to determine an optimal choice for the design parameters, the resulting table will be asymmetrical around zero. The asymmetry in the table is due to the asymmetry in the two's complement number system. To implement such a table in hardware, it is necessary to implement both the positive and negative halves of the table. This can lead to a large and slow implementation. An optimization can be made by carefully "folding" the negative portion of the table into the positive half [17]. For negative partial remainders, this requires conversion to a signed magnitude form. While the resulting table will be smaller and faster, the critical path of the divider increases by the delay of a two's complement to signed magnitude converter, which is essentially the delay of an exclusive-or gate. Figure 4 shows an implementation of a folded radix-4 next quotient-digit table that also supports shared square-root [18].



Divisor $1\frac{X}{16}$

Partial Remainder

$xxx.xxy	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
00.00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00.01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
00.10	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
00.11	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.01	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
01.10	2	A	B	1	1	1	1	1	1	1	1	1	1	1	1	1
01.11	2	2	2	2	A	1	1	1	1	1	1	1	1	1	1	1
10.00	2	2	2	2	2	2	2	A	1	1	1	1	1	1	1	1
10.01	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1
10.10	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
10.11	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
>11.00	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

A = 1 if negative and y bit = 1; A = 2 otherwise;

B = 2 if negative and y bit = 1; B = 1 otherwise.

Figure 4: Radix-4 next quotient/root selection table

2.4 Increasing Performance

2.4.1 Simple Staging

In order to retire more bits of quotient in every cycle, a simple low-radix divider can be replicated many times to form a higher radix divider, as shown in figure 5. In this implementation, the critical path is equal to:

$$t_{iter} = 2t_{qsel} + 2t_{qDsel} + 2t_{CSA}$$

In general, the implementation of divider hardware can range from totally sequential, as in the case of a single stage of hardware, to fully combinational, where the hardware is replicated enough such that the entire quotient can be determined combinationally in hardware. For totally or highly sequential implementations, the hardware requirements are small, saving chip area. This also leads to very fast cycle times, but the radix is typically low. Hardware replication can yield a very low latency in clock cycles due to the high radix but can occupy a large amount of chip area and have unacceptably slow cycle times.

One alternative to hardware replication to reduce division latency is to clock the divider at a faster frequency than the system clock. For example, in the HP PA7100, the very low cycle time of the radix-4 divider compared with the system clock allows it to retire 4 bits of quotient every machine cycle, effectively becoming a radix-16 divider [16]. The only additional hardware cost in this implementation is a few gates to generate the 2X clock.

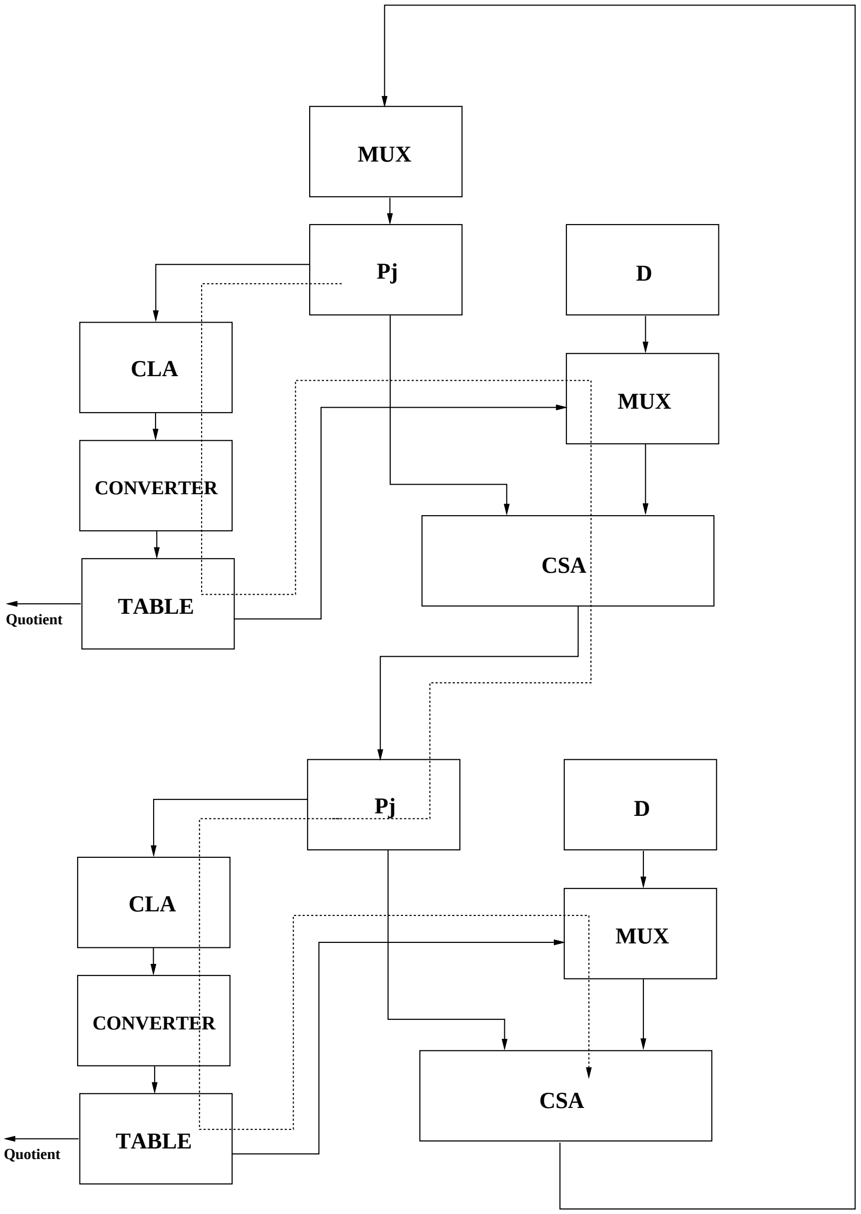


Figure 5: Higher radix using hardware replication

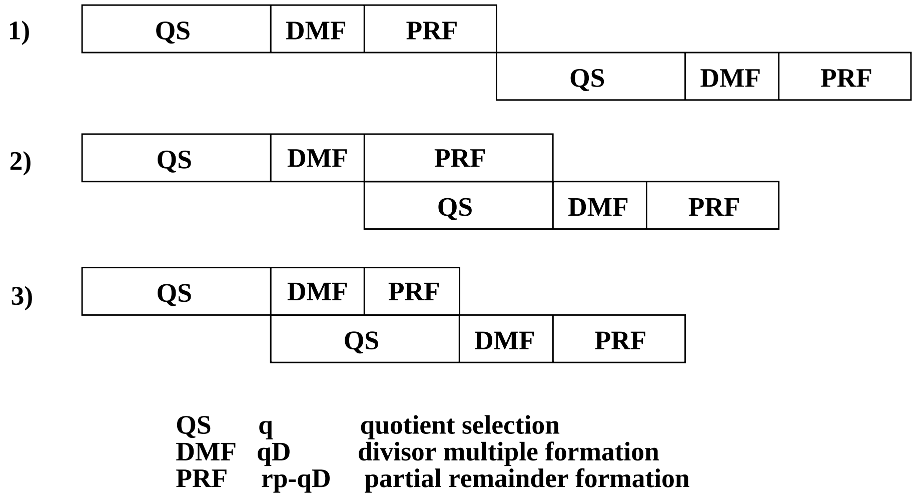


Figure 6: Three methods of overlapping division components

2.4.2 Overlapping Execution

It is possible to overlap or pipeline the components of the division step in order to reduce the cycle time of the division step [39]. This is illustrated in figure 6. The standard approach is represented in this figure by choice 1. Here, each quotient selection is dependent on the previous partial remainder, and this defines the cycle time. Depending upon the relative delays of the three components, choices 2 or 3 may be more desirable. Choice 2 is appropriate when the overlap is dominated by PRF time. This would be the case when the partial remainder is not kept in a redundant form. Choice 3 is appropriate when the overlap is dominated by QS, as it the case when a redundant partial remainder is used.

One recent SRT implementation using overlapped execution is reported by Williams [41]. This implementation differs from conventional designs in that it uses self-timing and dynamic logic to increase the divider's performance. It comprises five cascaded radix-2 stages as shown in figure 7. Because it uses self-timing, no explicit registers are required to store the intermediate residual. Accordingly, the critical path does not contain residual register clock-to-q or setup time delays. The adjacent stages overlap their computation by replicating the CPAs for each possible quotient digit from the previous stage. This allows each CPA to begin operation before the actual quotient digit arrives at a multiplexor to choose the correct branch. Two of the three CPAs in each stage are preceded by CSAs to speculatively compute a truncated version of $P_{i+1} - D$, $P_{i+1} + D$, and P_{i+1}. This overlapping of the execution between neighboring stages allows the delay through a stage to be the average, rather than the sum, of the propagation delays through the remainder and quotient-digit selection paths. This is illustrated in figure 7 by the two different drawn paths. The self-timing of the data path dynamically ensures that data always flow through the minimal critical path. This divider, implemented in a 1.2μm CMOS technology, is able to produce a 54-b result in 45 to 160ns, depending upon the particular data operands. The Hal SPARC V9 microprocessor, called the Sparc64 or PM1, also implements a version of

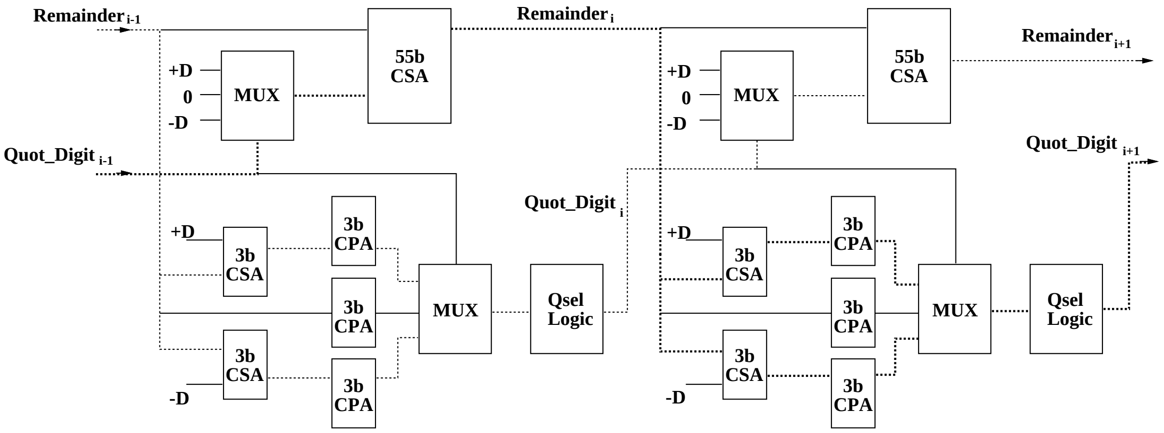


Figure 7: Two stages of self-timed divider

this self-timed divider, producing IEEE double precision results in 8 or 9 cycles [24].

2.4.3 Overlapping Quotient Selection

To avoid the increase in cycle time that results from staging radix-r segments together in forming higher radix dividers, some additional quotient computation can proceed in parallel [39]. In such a scheme, the quotient-digit selection of stage $j+2$ is overlapped with the quotient-digit selection of stage $j+1$, as shown in figure 8. This is accomplished by calculating an estimate of the next residual and the quotient-digit selection for q_{j+2} conditionally for all $2a+1$ values of the previous quotient digit q_{j+1}. Once the true value of q_{j+1} is known, it can be used to select the correct value of q_{j+2}. As can be seen from figure 8, the critical path is equal to:

$$t_{iter} = t_{qsel} + t_{qDsel} + 2t_{CSA} + t_{mux(data)}$$

Accordingly, comparing the simple staging of two stages with the overlapped quotient selection method for staging, it can be seen that the critical path has been reduced by

$$\Delta t_{iter} = t_{qsel} + t_{qDsel} - t_{mux(data)}$$

This is a reduction of slightly more than the delay of one stage of quotient-digit selection, at the cost of replicating $2a+1$ quotient-digit selection functions. This scheme has diminishing returns when overlapping more than two stages. Each additional stage requires the calculation of an additional factor $(2a+1)$ of quotient-digit values. Thus the kth additional stage will require $(2a+1)^k$ replicated quotient-selection functions. Because of this exponential growth in hardware, only very small values of k are feasible in practice.

Prabhu [30] discusses a radix-8 shared square-root design that utilizes overlapping quotient selection in the Sun UltraSPARC microprocessor. In this implementation, three radix-2 stages are cascaded to form a radix-8 divider. The second stage conditionally computes all three possible quotient digits of the the first stage, and the third stage computes all

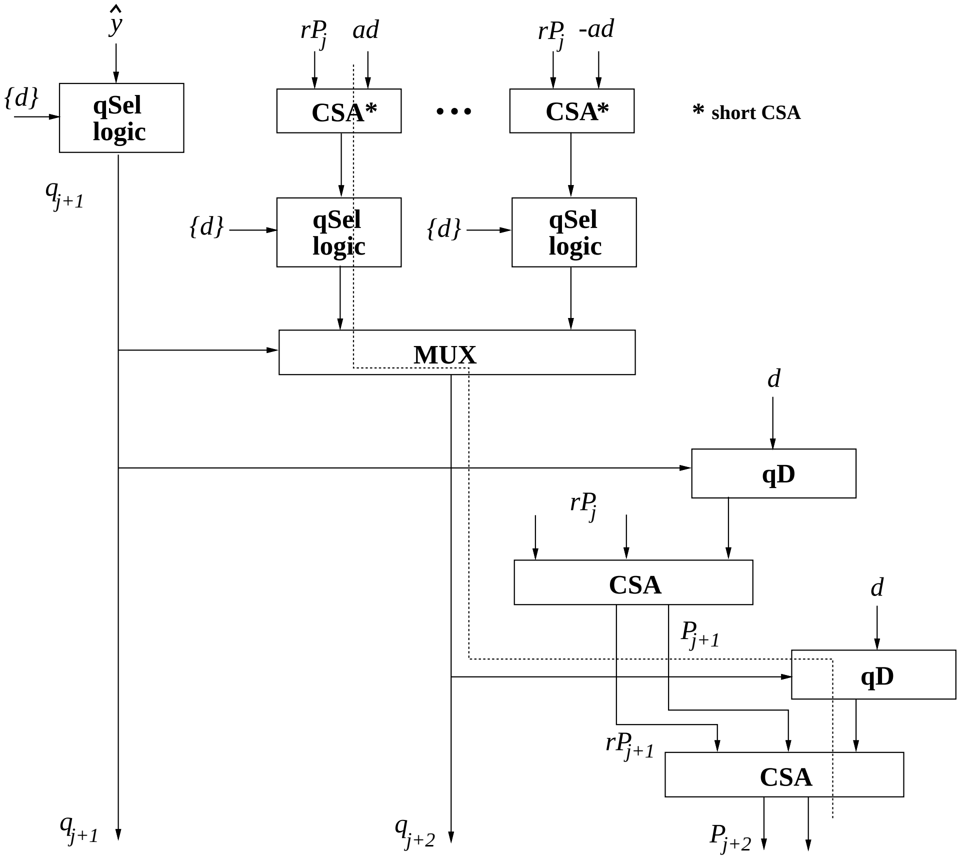


Figure 8: Higher radix by overlapping quotient selection

three possible quotient digits of the second stage. In the worst case, this would involve replication of three quotient-selection blocks for the second stage and nine blocks for the third stage. However, by recognizing that two of the nine blocks conditionally compute the identical quotient bits as another two blocks, only seven are needed.

2.4.4 Overlapping Residual Computation

A further optimization utilized both in the UltraSPARC and the Sparc64 is the overlapping of the residual computation. Both dividers implement a radix-2 digit set. Thus, the quotient can only take on the values of -1, 0, and +1. The divider's critical path after the quotient digit has been selected need not contain the delay of divisor multiple formation and the subtraction in the CSA. The divisor multiple formation and CSA hardware can be replicated three times, assuming one of the three quotient digit values for each replication. A multiplexor is then placed after these three stages so that the quotient digit can select the correct partial remainder. The delay is reduced by a CSA in the case of the UltraSPARC and by a CSA and a short CPA in the Sparc64 as shown in figure 7.

Quach [31] and Oberman [29] report similar optimizations for radix-4 implementations. For radix-4, it might initially seem that because of the five possible next quotient digits, five copies of residual computation hardware would be required. However, in the design of quotient-selection logic, the sign of the next quotient digit is known in advance, as it is just the sign of the previous partial remainder. This reduces the number of number of copies of residual computation hardware to three: 0, ±1, and ±2. However, by looking at a standard implementation of a radix-4 quotient-digit selection table, such as figure 4, it can be seen that the boundary between quotient digits 0 and 1 can be easily determined. To take advantage of this, the quotient digits are encoded as:

$$\begin{aligned} q(-2) &= Sq_2 \\ q(-1) &= Sq_1\overline{q}_2 \\ q(0) &= \overline{q}_1\overline{q}_2 \\ q(1) &= \overline{S}q_1\overline{q}_2 \\ q(2) &= \overline{S}q_2 \end{aligned}$$

In this way, the number of copies of residual computation hardware can be reduced to two: 0 or ±1, and ±2. A block diagram of a radix-4 divider with overlapped residual computation is shown in figure 9. The choice of 0 or ±1 is made by q_1 early, after only a few gate delays, by selecting the proper input of a multiplexor. Similarly, q_2 selects a multiplexor to choose which of the two banks of hardware is the correct one, either the 0 or ±1 bank, or the ±2 bank. The critical path of the divider becomes: $\max(tq_1, t_{CSA}) + 2t_{mux} + t_{shortCPA}$. Thus, at the expense of duplicating the residual computation hardware once, the cycle time of the standard radix-4 divider is nearly halved.

2.4.5 Range Reduction

Higher radix dividers can be designed by partitioning the implementation into lower radix segments, which are cascaded together. Unlike simple staging, in this scheme there is

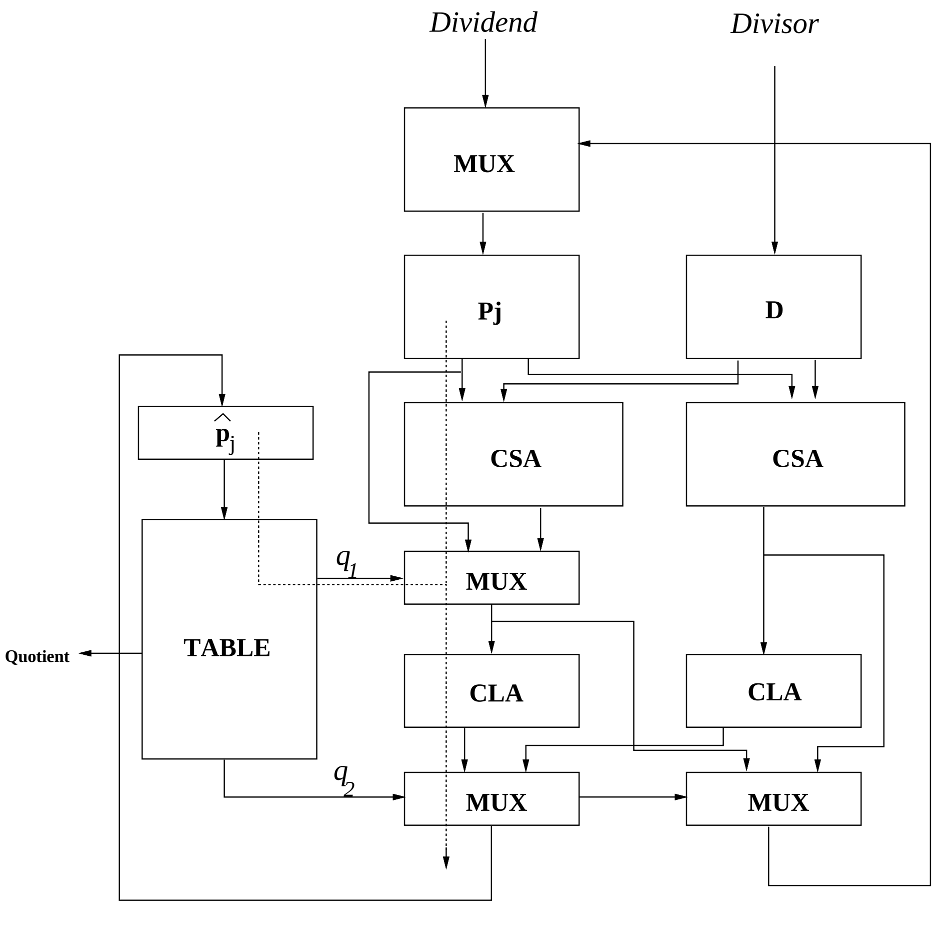


Figure 9: Radix-4 with overlapped residual computation

no shifting of the partial remainder between segments. Multiplication by the radix r is performed only between iterations of the step, but not between segments. The individual segments reduce the range of the partial remainder so that it is usable by the remaining segments [9, 18].

A radix-8 divider can be designed using a cascade of a radix-2 segment and a radix-4 segment. This decomposition is illustrated in figure 10. In this implementation the quotient digit sets are given by:

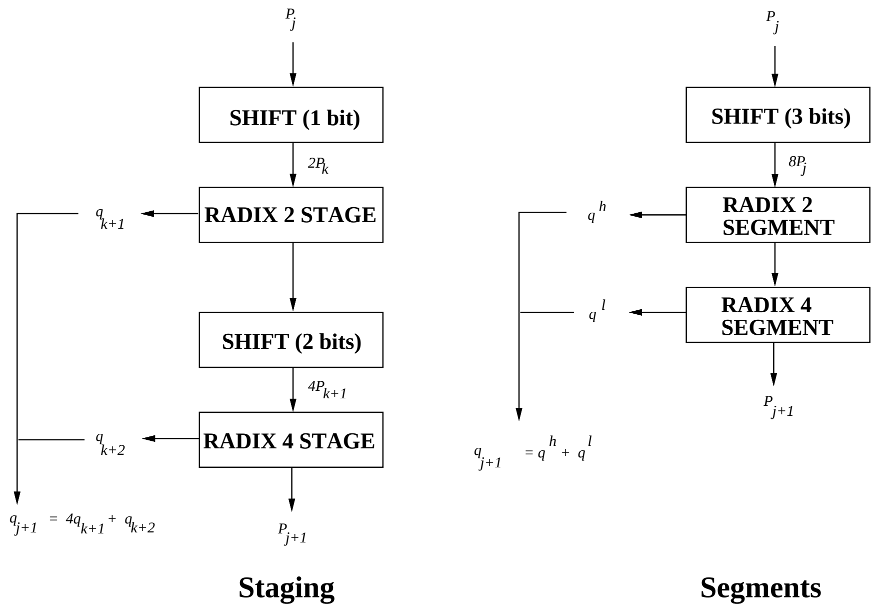


Figure 10: Decomposition into stages and segments

$$q_{j+1} = q^h + q^l \qquad q^h \in \{-4, 0, 4\}, \quad q^l \in \{-2, -1, 0, 1, 2\}$$

However, the resulting radix-8 digit set is given by:

$$q_{j+1} = \{-6, \ldots, 6\} \quad \rho = 6/7$$

When designing the quotient-digit selection hardware for both q_h and q_l, it should be realized that these are not standard radix-2 and radix-4 implementations, since the bounds on the step are set by the requirements for the radix-8 digit set. Additionally, to reduce the cycle time, the quotient-digit selections can be overlapped as discussed previously. In the worst case, this overlapping would involve two short CSAs, two short CPAs, and three instances of the radix-4 quotient-digit selection logic. However, it can be noticed that to distinguish the choices of $q^h = 4$ and $q^h = -4$, an estimate of the sign of the partial remainder is required, which can be done with only three bits of the carry-save representation of the partial remainder. Then, both $q^h = 4$ and $q^h = -4$ can share the same CSA, CPA and quotient-digit selection logic by muxing the input values. This overall reduction in hardware has the effect of increasing the cycle time by the delay of the sign detection logic and a mux.

The critical path for generating q^l is given by:

$$t_{iter} = t_{signest} + t_{mux} + t_{CSA} + t_{shortCPA} + t_{qlsel} + t_{mux(data)}$$

In order to form P_{j+1}, q^l is used to select the proper divisor multiple which is then subtracted from the partial remainder from the radix-2 segment. The additional delay to form P_{j+1} is a mux select delay and a CSA. For increased performance, it is possible to precompute all partial remainders in parallel and use q^l to select the correct result. This reduces the additional delay after q^l to only a mux delay.

2.4.6 Simple Operands Scaling

Higher radix dividers generally have their cycle times dominated by the time for quotient-digit selection. The complexity of quotient-digit selection increases exponentially for increasing radix. To simultaneously decrease latency and cycle time, it is desirable to reduce the complexity of the quotient-digit selection function. By looking at a PD diagram such as figure 2, it is apparent that the maximum overlap occurs for the largest value of the divisor. Assuming a normalized divisor in the range $1/2 \leq d < 1$, the greatest amount of overlap occurs close to $d = 1$. To take advantage of this overlap, the divisor can be restricted to a range close to 1. This can be accomplished by *prescaling* the divisor [8, 37]. In order that the quotient be preserved, either the dividend also must be prescaled, or else the quotient must be postscaled. In the case of prescaling, if the true remainder is required after the computation, postscaling is required. The dividend and the divisor are prescaled by a factor M so that the scaled divisor z is

$$1 - \alpha \leq z = Md \leq 1 + \beta$$

where α and β are chosen in order to provide the same scaling factor for all divisor intervals and to ensure that the quotient-digit selection is independent of the divisor. The initial partial remainder is the scaled dividend. The smaller the range of z is, the simpler the quotient-digit selection function is. However, shrinking the range of z becomes more complex for smaller ranges. Thus, a design tradeoff exists between these two constraints.

By restricting the divisor to a range near 1, the quotient-digit selection function becomes independent of the actual divisor value, and thus is simpler to implement. The radix-4 implementation reported in [8] uses 6 digits of the redundant partial remainder as inputs to the quotient-digit selection function. This function assimilates the 6 input digits in a CPA, and the 6 bit result is used to consult a look-up table to provide the next quotient-digit. The scaling operation is achieved through the use of a 3-operand adder. In the case where a CSA is already being used for the division recurrence, no additional CSAs are required. Instead, the scalings proceed in sequence. To determine the scaling factor for each operand, a small table is typically consulted which yields the proper factors to add or subtract in the CSA to yield the scaled operand. Thus, prescaling requires a minimum of two additional cycles to the overall latency; one to scale the divisor, and one to assimilate the divisor in a carry-propagate adder. In parallel with the divisor assimilation, the dividend is scaled, and it can be used directly in redundant form as the initial partial remainder. The motivation for scaling is that reduction in cycle time due to the simpler quotient-selection logic should more than offset the addition of any required scaling cycles.

Enhancements to the basic prescaling algorithms have been reported by Montuschi [25] and Srinivas [36]. Montuschi reports how the use of an over-redundant digit set can be used in combination with operand prescaling. The proposed radix-4 implementation uses such an over-redundant digit set $\{\pm 4, \pm 3, \pm 2, \pm 1, 0\}$. The quotient-digit selection function uses a truncated redundant partial remainder that is in the range $[-6, 6]$, requiring four digits of the partial remainder as input. A 4-bit CPA is used to assimilate the four most significant digits of the partial remainder and to add a 1 in the least significant position. The resulting 4 bits in two's complement form represent the next quotient digit. The formation of the $\pm 3d$ divisor multiple is an added complication, and the solution for this implementation is to split the quotient digit into two separate stages, one with digit set $\{0, \pm 4\}$ and one with $\{0, \pm 1, -2\}$. This is the same methodology used in the range reduction techniques previously presented. Thus, the use of a redundant digit set simplifies the quotient-digit selection from requiring 6 bits of input to only 4 bits.

Srinivas reports an implementation of prescaling with a maximally redundant digit set. This implementation represents the partial remainder in radix-2 digits $\{-1, 0, +1\}$ rather than carry-save form. Each radix-2 digit is presented by 2 bits. Accordingly, the quotient-selection function need only observe 3 digits of the radix-2 encoded partial remainder. The resulting quotient digits produced by this algorithm belong to the maximally redundant digit set $\{-3, \cdots, +3\}$. This simpler quotient-digit selection function decreases the cycle time relative to a regular redundant digit set with prescaling implementation. Srinivas reports a 1.21 speedup over Ercegovac's regular redundant digit set implementation, and a 1.10 speedup over Montuschi's over-redundant digit set implementation, both for $n = 53$ IEEE double precision mantissas. However, due to the larger than regular redundant digit sets in the implementations of both Montuschi and Srinivas, each requires hardware to generate the $\pm 3d$ divisor multiple, which in these implementations results in requiring an additional n CSAs.

2.5 Other Issues

2.5.1 Quotient Conversion

As presented so far, the quotient has been collected in a redundant form, such that the positive values have been stored in one register, and the negative values in another. At the conclusion of the division computation, an additional cycle would be required to assimilate these two registers into a single quotient value using a carry-propagate adder for the subtraction. However, it is possible to convert the quotient digits as they are produced such that an extra addition cycle is not required. This scheme is known as on-the-fly conversion [9].

In on-the-fly conversion, two forms of the quotient are kept in separate registers throughout the iterations, Q_k and QM_k. QM_k is defined to be equal to $Q_k - r^{-k}$. The values of these two registers for step $k+1$ are defined by:

$$Q_{k+1} = \begin{cases} Q_k + q_{k+1} r^{-(k+1)} & \text{if } q_{k+1} \geq 0 \\ QM_k + (r - |q_{k+1}|) r^{-(k+1)} & \text{if } q_{k+1} < 0 \end{cases}$$

and

$$QM_{k+1} = \begin{cases} Q_k + (q_{k+1} - 1) r^{-(k+1)} & \text{if } q_{k+1} > 0 \\ QM_k + ((r-1) - |q_{k+1}|) r^{-(k+1)} & \text{if } q_{k+1} \leq 0 \end{cases}$$

From these conditions on the values of Q_k and QM_k, it can be seen that all of the additions can be implemented with concatenations. As a result, there is no carry or borrow propagation required. As every quotient digit is formed, each of these two registers is updated appropriately, either through register swapping or concatenation.

2.5.2 Rounding

The previously described on-the-fly conversion can be extended to also handle final rounding [9]. For floating-point representations such as the IEEE 754 standard, provisions for rounding are required. Traditionally, this is accomplished by computing an extra guard digit in the quotient and examining the final remainder. Based on the rounding mode selected and these two values, one *ulp* is conditionally added. The disadvantages in the traditional approach are that 1) the remainder may be negative and require a restoration step, and 2) the the addition of one *ulp* may require a full carry-propagate-addition. Accordingly, support for rounding can be expensive, both in terms of area and performance.

To extend the on-the-fly techniques, it is necessary to keep a third version of the quotient at all times QP_k, where $QP_k = Q_k + r^{-k}$. Correct rounding requires the computation of the sign of the final remainder. Sign detection logic requires at a minimum some form of carry-propagation detection network, such in standard carry-lookahead adders. The final quotient can be selected from the three available versions. For negative remainders, QM_k or Q_k can be chosen to appropriately reduce the quotient or round it. For positive remainder, either QP_k or Q_k is chosen, again depending on the rounding conditions.

2.6 Analysis

This section has presented the various tradeoffs in digit recurrence division. Fundamentally, to reduce division latency, more bits need to be retired in every cycle. However, directly increasing the radix can greatly increase the cycle time and the complexity of divisor multiple formation. The alternative is to stage lower radix stages together to form higher radix dividers, through simple staging or segments, and possibly overlapping one or both of the quotient selection logic and residual computation hardware. All of these alternatives lead to an increase in area, complexity and potentially cycle time. Given the continued industry demand for ever-lower cycle times, any increase must be managed.

Higher degrees of redundancy in the quotient digit set and operand prescaling are the two primary means of further reducing the recurrence cycle time. These two methods can be combined for an even greater reduction. For radix-4 division with operand prescaling, it has been shown that an over-redundant digit set can reduce the number of partial remainder bits required for quotient selection from 6 to 4. Choosing a maximally redundant set and a radix-2 encoding for the partial remainder can reduce the number of partial remainder bits required for quotient selection down to 3. However, each of these enhancements requires additional area and complexity for the implementation that must be considered.

Due to the cycle time constraints and area budgets of modern processors, these dividers are realistically limited to retiring fewer than 10 bits per cycle. However, a digit recurrence divider is an effective means of implementing a low cost division unit which operates in parallel with the rest of a processor.

3 Functional Iteration

Unlike digit recurrence division, division by functional iteration utilizes multiplication as the fundamental operation. The primary difficulty with subtractive division is the linear convergence to the quotient. Multiplicative division algorithms, though, are able to take advantage of high-speed multipliers to converge to a result quadratically. Rather than retiring a fixed number of quotients bits in every cycle, multiplication-based algorithms are able to double the number of correct quotient bits in every iteration. However, the tradeoff between the two classes is not only latency in terms of the number of iterations, but also the length of each iteration in cycles. Additionally, if the divider shares an existing multiplier, the performance ramifications on regular multiplication operations must be considered. Oberman [26] reports that in typical floating-point applications, the performance degradation due to a shared multiplier is small. Accordingly, if area must be minimized, an existing multiplier may be shared with the division unit with only minimal system performance degradation. This section presents the algorithms used in multiplication-based division, both of which are related to the Newton-Raphson equation. Additionally, it discusses issues related to increasing the performance of multiplication-based division algorithms, including methods for generating starting approximations, type of multiplier, and rounding methods.

3.1 Newton-Raphson

Division can be written as the product of the dividend and the reciprocal of the divisor, or

$$Q = a/b = a \times (1/b),$$

where Q is the quotient, a is the dividend, and b is the divisor. In this case, the challenge becomes how to efficiently compute the reciprocal of the divisor. In the Newton-Raphson algorithm, a *priming function* is chosen which has a root at the reciprocal [19]. In general, there are many root targets that could be used, including $\frac{1}{b}$, $\frac{1}{b^2}$, $\frac{a}{b}$, and $1 - \frac{1}{b}$. The choice of which root target to use is arbitrary. The selection is made based on convenience of the iterative form, its convergence rate, its lack of divisions, and the overhead involved when using a target root other than the true quotient.

The most widely used target root is the divisor reciprocal $\frac{1}{b}$, which is the root of the priming function

$$f(X) \quad = \quad 1/X - b = 0. \tag{4}$$

The well-known quadratically converging Newton-Raphson equation is given by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{5}$$

The Newton-Raphson equation of (5) is then applied to (4). The function and its first derivative are evaluated at X_0:

$$\begin{aligned} f(X_0) &= 1/X_0 - b \\ f'(X_0) &= -1/X_0^2. \end{aligned}$$

These results are then used to find an approximation to the reciprocal:

$$\begin{aligned} X_1 &= X_0 - \frac{f(X_0)}{f'(X_0)} \\ X_1 &= X_0 + \frac{(1/X_0 - b)}{(1/X_0^2)} \\ X_1 &= X_0 \times (2 - b \times X_0) & (6) \\ &\vdots \\ X_{i+1} &= X_i \times (2 - b \times X_i) & (7) \end{aligned}$$

The corresponding error term is given by

$$\epsilon_{i+1} = \epsilon_i^2(b),$$

and thus the error in the reciprocal decreases quadratically after each iteration. As can be seen from the general relationship expressed in (7), each iteration involves two multiplications and a subtraction. The subtraction is equivalent to the two's complement operation and is commonly replaced by it. Thus, two dependent multiplications and one two's complement operation are performed each iteration. The final quotient is obtained by multiplying the computed reciprocal with the dividend.

It can be seen that the number of operations per iteration and their order are intrinsic to the iterations themselves. However, the number of iterations required to obtain the reciprocal accurate to a particular number of bits is a function of the accuracy of the initial approximation X_0. By using a more accurate starting approximation, the total number of iterations required can be reduced. To achieve 53 bits of precision for the final reciprocal starting with only 1 bit, the algorithm will require 6 iterations:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 53$$

By using a more accurate starting approximation, for example 8 bits, the latency can be reduced to 3 iterations. By using at least 14 bits, the latency could be further reduced to only 2 iterations.

3.2 Series Expansion

A different method of deriving a division iteration is based on a series expansion. A name sometimes given to this method is *Goldschmidt's algorithm*. Consider the familiar Taylor series expansion of a function $g(y)$ at point a p,

$$g(y) \quad = \quad g(p) + (y-p)g'(p) + \frac{(y-p)^2}{2!}g''(p) + \cdots + \frac{(y-p)^n}{n!}g^{(n)}(p) + \cdots.$$

In the case of division, it is desired to find the expansion of the reciprocal of the divisor, such that

$$q = \frac{a}{b} = a \times g(y),$$

where $g(y)$ can be computed by an efficient iterative method. A straightforward approach might be to choose $g(y)$ equal to $1/y$ with $p = 1$, and then to evaluate the series. However, it is computationally easier to let $g(y) = 1/(1+y)$ with $p = 0$, which is just the Maclaurin series. Then, the function is

$$g(y) \quad = \quad \frac{1}{1+y} = 1 - y + y^2 - y^3 + y^4 - \cdots.$$

So that $g(y)$ is equal to 1/b, the substitution $y = b - 1$ must be made, where b is bit normalized such that $0.5 \leq b < 1$, and thus $|Y| \leq 0.5$. Then, the quotient can be written as

$$q \quad = \quad a \times \frac{1}{1 + (b-1)} = a \times \frac{1}{1+y} = a \times (1 - y + y^2 - y^3 + \cdots)$$

which, in factored form, can be written as

$$q \quad = \quad a \times [(1-y)(1+y^2)(1+y^4)(1+y^8)\cdots]. \tag{8}$$

This expansion can be implemented iteratively as follows. An approximate quotient can be written as

$$q_i = \frac{N_i}{D_i}$$

where N_i and D_i are iterative refinements of the numerator and denominator after step i of the algorithm. By forcing D_i to converge toward 1, N_i converges toward q. Effectively, each iteration of the algorithm provides a correction term $(1 + y^{2i})$ to the quotient, generating the expansion of (8).

Initially, let $N_0 = a$ and $D_0 = b$. To reduce the number of iterations, a and b should both be prescaled by a more accurate approximation of the reciprocal, and then the algorithm should be run on the scaled a' and b'. For the first iteration, let $N_1 = R_0 \times N_0$ and $D_1 = R_0 \times D_0$, where $R_0 = 1 - y = 2 - b$, or simply the two's complement of the divisor. Then,

$$D_1 = D_0 \times R_0 = b \times (1-y) = (1+y)(1-y) = 1 - y^2.$$

Similarly,

$$N_1 = N_0 \times R_0 = a \times (1-y).$$

For the next iteration, let $R_1 = 2 - D_1$, the two's complement of the new denominator. From this,

$$\begin{aligned} R_1 &= 2 - D_1 = 2 - (1 - y^2) = 1 + y^2 \\ N_2 &= N_1 \times R_1 = a \times [(1 - y)(1 + y^2)] \\ D_2 &= D_1 \times R_1 = (1 - y^2)(1 + y^2) = (1 - y^4) \end{aligned}$$

Continuing, a general relationship can be developed, such that each step of the iteration involves two multiplications

$$N_{i+1} = N_i \times R_i \quad \text{and} \quad D_{i+1} = D_i \times R_i$$

and a two's complement operation,

$$R_{i+1} = 2 - D_{i+1}$$

After i steps,

$$\begin{aligned} N_i &= a \times [(1 - y)(1 + y^2)(1 + y^4) \cdots (1 + y^{2i})] && (9) \\ D_i &= (1 - y^{2i}) && (10) \end{aligned}$$

Accordingly, N converges quadratically toward q and D converges toward 1. This can be seen in the similarity between the formation of N_i in (9) and the series expansion of q in (8). So long as b is normalized in the range $0.5 \leq b < 1$, then $y < 1$, each correction factor $(1 + y^{2i})$ doubles the precision of the quotient. This process continues as shown iteratively until the desired accuracy of q is obtained.

Consider the iterations for division. A comparison of equation (9) using the substitution $y = b - 1$ with equation (7) using $X_0 = 1$ shows that the results are identical iteration for iteration. Thus, the series expansion is mathematically identical to the Newton-Raphson iteration for $X_0 = 1$. Additionally, each algorithm can benefit from a more accurate starting approximation of the reciprocal of the divisor to reduce the number of required iterations. However, the implementations are not exactly the same. First, Newton-Raphson converges to a reciprocal, and then multiplies by the dividend to compute the quotient, whereas the series expansion first prescales the numerator and the denominator by the starting approximation and then converges directly to the quotient. Each iteration in both algorithms comprises two multiplications and a two's complement operation. From (7), it can be noted that the multiplications in Newton-Raphson are dependent operations. In the series expansion implementation, though, the two multiplications of the numerator and denominator are independent operations and may occur in parallel. As a result, the series expansion implementation can take advantage of an existing pipelined multiplier to obtain higher performance. Second, the Newton-Raphson iteration is self-correcting, in that any error in computing X_i can be corrected in the subsequent iteration, since all operations are dependent. However, in the series-expansion implementation, the result is computed as the product of independent terms, and the error in one of them will not be corrected. To account for this error, the calculations should use a few extra bits of precision. A performance enhancement that can be used for both iterations is to perform early computations in reduced precision. This is reasonable, because the early computations do not generate many correct bits. As the iterations continue, quadratically larger amounts of precision are required in the computation.

In practice, dividers based on functional iteration have used both versions. The Newton-Raphson algorithm was used in the Astronautics ZS-1 [4], Intel i860 [21], and the IBM RS/6000 [22]. The series expansion was used in the IBM 360/91 [15] and TMS390C602A [13]. Latencies for such dividers range from 11 cycles to more than 16 cycles, depending upon the precision of the initial approximation and the latency and throughput of the floating-point multiplier.

3.3 Starting Approximations

As mentioned previously, Newton-Raphson and series expansion division implementations can benefit from a more accurate initial reciprocal approximation. There is one standard methods and one newer method of forming starting approximations:

1. Look-up tables
2. Partial product arrays

3.3.1 Look-up tables

For modern division implementations, the most common method of generating starting approximations is through a look-up table. Such a table is typically implemented in the form of a ROM or a PLA. Typical implementations consist of a 1 kilobyte ROM which provides an approximation of 8 or 9 bits. An advantage of look-up tables is that they are fast, since no arithmetic calculations need be performed. The disadvantage is that a look-up table's size grows exponentially with each bit of added accuracy. Accordingly, a tradeoff exists between the precision of the table and its size.

To index into a reciprocal table, it is assumed that the operand is IEEE normalized $1.0 \leq b < 2$. Given such a normalized operand, k bits of the truncated operand are used to index into a table providing m output bits of the reciprocal approximation, which has the range $0.5 < recip \leq 1$. The truncated operand is represented as $1.b'_1 b'_2 \cdots b'_k$, and the output reciprocal approximation is $0.1b'_1 b'_2 \cdots b'_m$. Typically, the design of a reciprocal table starts with a specification for the minimum accuracy of the table, often expressed in bits. This value dictates the minimum size of each table entry. To determine the number of bits of the input operand required to index into the table, the following expression is evaluated:

$$|\frac{1}{b} - \frac{1}{b - 2^{-n}}| \leq \epsilon_0,$$

where ϵ_0 is defined as the error due to approximation. When truncating b at the nth bit, the reciprocal approximation must not differ from the true reciprocal by more than ϵ_0. A common method of designing the look-up table is to implement a piecewise-constant approximation of the reciprocal function. In this case, the approximation for each entry is found by taking the reciprocal of the mid-point between $1.b'_1 b'_2 \cdots b'_k$ and its successor where the mid-point is $1.b'_1b'_2\cdots b'_k1$. The reciprocal of the mid-point is rounded by adding $2^{-(m+1)}$, and then truncating the result to produce the reciprocal approximation $0.1b'_1b'_2\cdots b'_m$. As can be seen, all values will have a leading-one that can be implied and therefore do not need to be explicitly stored in the table.

Das Sarma [5] has shown that the piecewise-constant approximation method for generating reciprocal look-up tables minimizes the maximum relative error in the final result. He further describes how to generate optimal k-bits-in m-bits-out reciprocal tables. A k-bits-in k-bits-out reciprocal table will guarantee a precision of at least $k + 0.415$ bits. Also, it is shown that with $m = k + g$, where g is the number of output guard bits, a generated table with one, two, and three guard bits on the output are guaranteed precision of at least $k + 0.678$ bits, $k + 0.830$ bits, and $k + 0.912$ bits respectively.

Rather than using a constant approximation to the reciprocal, it is possible to use a linear or polynomial approximation. A polynomial approximation is expressed in the form of a truncated series of the form:

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots$$

To get a first order or linear approximation, the coefficients a_0 and a_1 are stored in a look-up table, and a multiplication and an addition are required. Schulte [14] developed methods for selecting constant and linear approximations which minimize the absolute error of the final result for Newton-Raphson implementations. Minimizing the maximum relative error in an initial approximation minimizes the maximum relative error in the final result. However, the initial approximation which minimizes the maximum absolute error of the final result depends on the number of iterations of the algorithm. Accordingly, they present the tradeoff between n, the number of iterations, and k, the number of bits used as input to the table, for constant and linear approximations, and the effects on the absolute error of the final result. In general, linear approximations guarantee more accuracy than constant approximations, but they require twice as many entries in the table and additional operations.

In a more recent study, Das Sarma [6] describes bipartite reciprocal tables. These tables utilize separate table lookup of the positive and negative portions of a reciprocal value in borrow-save form. This separation allows 4-9 bit reciprocal tables to be 2 to 4 times smaller than conventional tables. For 10-16 bit tables, bipartite tables can be 4 to more than 16 times smaller than conventional implementations. Using such bipartite tables may allow for larger starting approximations than would normally be considered.

3.3.2 Partial Product Arrays

Another alternative to look-up tables for starting approximation is the use of partial product arrays [33]. A partial product array can be derived which sums to an approximation of the reciprocal operation. Such an array is similar to the partial product array of a multiplier. As a result, an existing floating-point multiplier can be used to perform the summation.

A multiplier used to implement IEEE double precision numbers involves 53 rows of 53 elements per row. This entails a large array of 2809 elements. If Booth encoding is used in the multiplier, the bits of the partial products are recoded, decreasing the number of rows in the array by half. A Booth multiplier typically has only 27 in the partial product array.

A multiplier sums all of these boolean elements to form the product. However, each boolean element of the array can be replaced by a generalized boolean element. By back-solving the partial product array, it can be determined what elements are required to generate the appropriate function approximation. These elements are chosen carefully to provide a high-precision approximation and reduce maximum error. This can be viewed as analogous to the choosing of coefficients for a polynomial approximation. In this way, a partial product array is generated which reuses existing hardware to generate a high-precision approximation.

In the case of the reciprocal function, a 17 digit approximation can be chosen which utilizes 18 columns of a 53 row array. Less than 20% of the array is actually used. However, the implementation is restricted by the height of the array, which is the number of rows. The additional hardware for the multiplier is 484 boolean elements. It has been shown that such a function will yield a minimum of 12.003 correct bits, with an average of 15.18 correct bits. An equivalent ROM look-up table that generates 12 bits would require about 39 times more area. If a Booth multiplier is used with only 27 rows, a different implementation can be used. This version uses only 175 boolean elements. It generates an average of 12.71 correct bits and 9.17 bits in the worst case. This is about 9 times smaller than an equivalent ROM look-up table.

3.4 Rounding

The main disadvantage of using functional iteration for division is the difficulty in obtaining a correctly rounded result. With subtractive implementations, both a result and a remainder are generated, making rounding a straightforward procedure. Functional iteration which converges directly to the quotient, such as the series expansion implementation, only produces a result which is close to the correctly rounded quotient, and it does not produce a remainder. The Newton-Raphson algorithm has the additional disadvantage that it converges to the reciprocal, not the quotient. Even if the reciprocal can be correctly rounded, it does not guarantee that the quotient will be correctly rounded.

There are two main techniques used to compute a correctly rounded result when using series expansion functional iteration. The first method requires a datapath twice as wide as the final result. The quotient is computed to a little more than twice the precision of the final quotient, and then the extended result is rounded to the final precision. An explanation of this procedure is as follows. Consider that the dividend X and the divisor Y are both normalized and represented by b bits, and the final quotient $Q = X/Y$ is represented by b bits. It must be first noted that the exact halfway quotient can not occur when dividing two b bit normalized numbers. For an exact halfway case, the quotient would be represented by exactly a $b+1$ bit number with both its MSB and LSB equal to 1, and thus having exactly $b-1$ bits between its most significant and least significant 1's. The product of such a number with any non-zero finite binary number must also have the same property, and thus the dividend must have this property. But, the dividend is defined to be a normalized b bit number, and thus can it can have a maximum of $b-2$ bits between its most significant and least significant 1's.

To obtain b significant bits of the quotient, b bits are computed if the first quotient bit is 1, and $b+1$ bits if the first quotient bit is 0. At this point, because the exact halfway case can not occur, rounding can proceed based solely on the values of the next quotient bit and the sticky bit. The sticky bit is 0 if the remainder at this point is exactly zero. If any bit of the remainder is 1, then the sticky bit is 1. Let R_0 be the value of the remainder after this computation, assuming the first bit is 1:

$$X = Q_0 \times Y + R_0, \quad \text{with } R_0 < 2^{-b}$$

Then, compute another b bits of quotient, denoted Q_1.

$$R_0 = Q_1 \times Y + R_1, \quad \text{with } R_1 < 2^{-2b}$$

Q_1 is less than 2^{-b}, with an accuracy of 2^{-2b}, and Y is normalized to be accurate to 2^{-b}. Accordingly if $Q_1 = 0$, then $R_0 = R_1$. But, R_0 can equal R_1 if and only if $R_0 = R_1 = 0$. This is because $R_0 < 2^{-b}$ and $R_1 < 2^{-2b}$ and Y is a b bit quantity. Similarly, if $Q_1 \neq 0$, then the remainder R_0 can not equal 0. The computation proceeds in the same manner if the first quotient bit is 0, except that $b+1$ bits will have been computed for Q_0. From this analysis, it is apparent that by computing at most $2b+1$ bits, the quotient can be correctly rounded without requiring the actual remainder.

The principal disadvantage of this method is that it requires one additional full iteration, and it requires a datapath at least two times larger than is required for non-rounded results. A faster and smaller method has been shown to be possible that was first implemented on the TI 8847 and TMS390C602A [13]. This scheme does not require a two times larger datapath. Rather, the quotient is computed in a datapath with six extra guard bits. The quotient at that point is equal to

$$q = \frac{dividend}{divisor} + rem.$$

An extra multiplication is then performed to compute $q \times divisor$. This result is compared with the actual dividend, still with only six extra guard bits. From this low-precision comparison, the rounding direction can be readily obtained.

An additional method has been proposed for rounding in Newton-Raphson implementations that utilize a signed-digit multiplier [10]. The signed-digit representation allows for the removal of the subtraction or complement cycles of the iteration. In this scheme, it is possible to obtain a correctly rounded quotient in nine cycles, including the final multiplication and ROM access. The redundant binary recoding of the partial products in the multiplier allows for the simple generation of a correct sticky bit. Using this sticky bit and a special recode circuit in the multiplier, correct IEEE rounding is possible at the cost of only one additional cycle to the algorithm.

3.5 Analysis

Both the Newton-Raphson and series expansion iterations are effective means of implementing division in hardware. For both iterations, the cycle time is limited by two multiplications. In the Newton-Raphson iteration, these multiplications are dependent and must proceed in series, while in the series expansion, these multiplications may proceed in parallel. To reduce the latency of the iterations, an accurate initial approximation can be used.

This introduces a tradeoff between additional chip area for a look-up table and the latency of the divider. An alternative to a look-up table is the use of a partial product array, possibly by reusing an existing floating-point multiplier. Instead of requiring additional area, such an implementation could increase the cycle time through the multiplier. The primary advantage of division by functional iteration is the quadratic convergence to the quotient. Functional iteration does not readily provide a final remainder. Accordingly, correct rounding for functional iteration implementations is difficult. When a latency is required lower than can be provided by an SRT implementation, functional iteration is currently the primary alternative. It provides a way to achieve lower latencies without seriously impacting the cycle time of the processor and without a large amount of additional hardware.

4 Very High Radix Algorithms

Digit recurrence algorithms are readily applicable to low radix division and square-root implementations. As the radix increases, the quotient-digit selection hardware and divisor multiple process become more complex, increasing cycle time, area or both. To achieve very high radix division with acceptable cycle time, area, and means for precise rounding, it is necessary to use a variant of the digit recurrence algorithms, with simpler quotient-digit selection hardware. The term "very high radix" applies roughly to dividers which retire more than 10 bits of quotient in every iteration. The very high radix algorithms presented are similar in that they all use multiplication for divisor multiple formation and look-up tables to obtain an initial approximation to the reciprocal. They differ in the number and type of operations used in each iteration and the technique used for quotient-digit selection.

4.1 Accurate Quotient Approximations

This high radix algorithm proposed by Wong [42] is as follows. The quotient Q is defined in terms of the normalized dividend X and divisor Y as

$$Q = \frac{X}{Y}.$$

In the algorithm, truncated version of X and Y are used, denoted X_h and Y_h. X_h is defined as the high-order $m+1$ bits of X extended with 0's to get a n-bit number. Similarly, Y_h is defined as the high order m bits of Y extended with 1's to get a n-bit number. From these definitions, it is clear that X_h is always less than or equal to X and Y_h is always greater than or equal to Y. This implies that $1/Y_h$ is always less than or equal to $1/Y$, and therefore X_h/Y_h is always less than or equal to X/Y.

The algorithm is as follows:

1. Initially, set the estimated quotient Q and the variable j to 0. Then, get an approximation of $1/Y_h$ from a look-up table, using the top m bits of Y, returning an m bit approximation. However, only $m-1$ bits are actually required to index into the table, as the guaranteed leading one can be assumed. In parallel, perform the multiplication $X_h \times Y$.

2. Scale both the truncated divisor and the previously formed product by the reciprocal approximation. This involves two multiplications in parallel for maximum performance,

   $$(1/Y_h) \times Y \quad \text{and} \quad (1/Y_h) \times (X_h \times Y)$$

   The product $(1/Y_h) \times Y = Y'$ is invariant across the iterations, and therefore only needs to be performed once. Subsequent iterations need only compute one multiplication:

   $$Y' \times P_h,$$

   where P_h is the current truncated partial remainder. The product $P_h \times 1/Y_h$ can be viewed as the next quotient digit, while $(P_h \times 1/Y_h) \times Y$ is the effective divisor multiple formation.

3. Perform the general recurrence to obtain the next partial remainder:

   $$P' = P - P_h \times (1/Y_h) \times Y,$$

   where $P_0 = \mathrm{X}$. Since all products have already been formed, this step only involves a subtraction.

4. Compute the new quotient as

   $$\begin{aligned} Q' &= Q + (P_h/Y_h) \times (1/2^j) \\ &= Q + P_h \times (1/Y_h) \times (1/2^j) \end{aligned}$$

   The new quotient is then developed by forming the product $P_h \times (1/Y_h)$ and adding the shifted result to the old quotient Q.

5. The new partial remainder P' is normalized by left-shifting to remove any leading 0's. It can be shown that the algorithm guarantees $m - 2$ leading 0's. The shift index j is revised by $j' = j + m - 2$.

6. All variables are adjusted such that $j = j'$, $Q = Q'$, and $P = P'$.

7. Repeat steps 2 through 6 of the algorithm until $j \geq q$.

8. After the completion of all iterations, the top n bits of Q form the true quotient. Similarly, the final remainder is formed by right-shifting P by $j - q$ bits. This remainder, though, assumes the use of the entire value of Q as the quotient. If only the top n bits of Q are used as the quotient, then the final remainder is calculated by adding $Q_l \times Y$ to P, where Q_l comprises the low order bits of Q after the top n bits.

This basic algorithm reduces the partial remainder P by $m - 2$ bits every iteration. Accordingly, an n bit quotient requires $\lceil n/(m-2) \rceil$ iterations.

An advanced version of this algorithm has also been proposed. Rather than representing the approximate reciprocal by a single constant term $1/Y_h$ obtained from a look-up table, more terms from a Taylor series approximation can be used. The Taylor series approximation equation for $1/Y$ at $Y = Y_h$ is:

$$1/Y = 1/Y_h - \Delta Y/Y_h^2 + (\Delta Y)^2/Y_h^3 \cdots$$

The advanced version uses the same iteration steps as in the basic algorithm presented earlier. However, in step 1, while $1/Y_h$ is obtained from a look-up table using the leading m bits of Y, in parallel approximations for $1/Y_h^2$, $1/Y_h^3$, etc. are obtained from additional look-up tables, all indexed using the leading m bits of Y. These additional tables have word widths of b_i given by

$$b_i = (m \times t - t) + \lceil \log_2 t \rceil - (m \times i - m - i).$$

where t is the number of terms of the series used, and thus the number of look-up tables. The value of t must be at least 2, but all subsequent terms are optional. The advanced version reduces P' by $m \times t - t - 1$ bits per iteration, and therefore the algorithm requires $\lceil n/(m \times t - t - 1) \rceil$ iterations.

As in low-radix SRT implementations, both versions of the algorithm can benefit by storing the partial remainder P in a redundant representation. However, before any of the multiplications using P as an operand take place, the top $m + 3$ bits of P must be carry-assimilated for the basic method, and the top $m + 5$ bits of P must be carry-assimilated for the advanced method. Similarly, the quotient Q can be kept in a redundant form until the final iteration. After the final iteration, full carry-propagate additions must be performed to calculate Q and P in normal, non-redundant form.

The hardware required for this algorithm is as follows. At least one look-up table is required of size $2^{m-1}m$ bits. Three multipliers are required: one multiplier with carry assimilation of size $(m + 1) \times n$ for the initial multiplications by the divisor Y, one carry-save multiplier with accumulation of size $(m + 1) \times (n + m)$ for the iterations, and one carry-save multiplier of size $(m + 1) \times m$ to compute the quotient segments. One carry-save adder is required to accumulate the quotient in each iteration. Two carry-propagate adders are required: one short adder at least of size $m + 3$ bits to assimilate the most significant bits of the partial remainder P, and one adder of size $n + m$ to assimilate the final quotient.

To calculate IEEE double precision quotients, where $n = 53$, several permutations of this algorithm are possible. A slower implementation might utilize the basic method with $m = 11$. The single look-up table would have $2^{11-1} = 1024$ entries, each 11 bits wide, for a total of 11K bits in the table. Assuming all multipliers with assimilation compute results in 1 cycle, it would take 2 initial cycles to perform the table look-up of $1/Y_h$ and perform the two initial multiplications and perform the first iteration, 5 cycles for the remaining iterations, 1 cycle to assimilate the quotient, and 1 cycle for rounding. This results in a total of 9 cycles. A faster implementation using the advanced method with $m = 15$ and $t = 2$ would require a total table size of 736K bits. It would require 2 initial cycles, 1 cycle for the additional iteration, 1 cycle for quotient assimilation, and 1 rounding cycle, for a total of 5 cycles. Thus, at the expense of several multipliers, adders, and two large look-up tables, the latency of division can be greatly reduced using this algorithm. In general, the algorithm requires at most $\lceil n/(m - 2) \rceil + 3$ cycles.

4.2 Short Reciprocal

The Cyrix 83D87 arithmetic coprocessor utilizes a short reciprocal algorithm similar to the accurate quotient approximation method to obtain a radix 2^{17} divider [23, 2]. Instead of having several multipliers of different sizes, the Cyrix divider has a single 18x69 rectangular multiplier with an additional adder port that can perform a fused multiply/add. It can, therefore, also act as a 19x69 multiplier. Otherwise, the general algorithm is nearly identical:

1. Initially, an estimate of the reciprocal $1/Y_h$ is obtained from a look-up table. In the Cyrix implementation, this approximation is of low precision. This approximation is refined through two iterations of the Newton-Raphson algorithm to achieve a 19 bit approximation. This method decreases the size of the look-up table at the expense of additional latency. Also, this approximation is chosen to be intentionally larger than the true reciprocal by an amount no greater than 2^{-18}. This differs from the accurate quotient method where the approximation is chosen to be intentionally smaller than the true reciprocal.

2. Perform the recurrence

$$\begin{aligned} P' &= P - P_h \times (1/Y_h) \times Y & (11) \\ Q' &= Q + P_h \times (1/Y_h) \times (1/2^j) & (12) \end{aligned}$$

   where P_0 is the dividend X. In this implementation, the two multiplications of (11) need to be performed separately in each iteration. One multiplication is required to compute $P_h \times (1/Y_h)$, and a subsequent multiply/add is required to multiply by Y and accumulate the new partial remainder. The product $P_h \times (1/Y_h)$ is a 19 bit high radix quotient digit. The multiplication by Y forms the divisor multiple required for subtraction. However, the multiplication $P_h \times (1/Y_h)$ required in (12) can be reused from the result computed for (11). Only one multiplication was required in the accurate quotient method because the product $(1/Y_h) \times Y$ was computed once at the beginning in full precision, and could be reused on every iteration. The Cyrix multiplier only produces limited precision results, 19 bits, and thus the multiplication by Y needs to be repeated at every iteration. Because of the specially chosen 19 bit short reciprocal, along with the 19 bit quotient digit and 18 bit accumulated partial remainder, this scheme guarantees that 17 bits of quotient are retired in every iteration.

3. After the iterations, one additional cycle is required for rounding and postcorrection. Unlike the accurate quotient method, on-the-fly conversion of the quotient digits is possible, as there is no overlapping of the quotient segments between iterations.

Thus, the short reciprocal algorithm is very similar to the accurate quotient algorithm. One difference is the method for generating the short reciprocal. However, either method could be used in both algorithms. The use of Newton-Raphson to increase the precision of a smaller initial approximation is chosen merely to reduce the size of the look-up table. The fundamental difference between the two methods is Cyrix's choice of a single rectangular fused multiplier/add unit with assimilation to perform all core operations. While this eliminates a majority of the hardware required in the accurate quotient method, it increases the iteration length from one multiplication to two due to the truncated results.

The short reciprocal unit can generate double precision results in 15 cycles: 6 cycles to generate the initial approximation by Newton-Raphson, 4 iterations with 2 cycles per iteration, and one cycle for postcorrection and rounding. As mentioned, with a larger table, the initial approximation could be obtained in as little as 1 cycle, reducing the total cycle count to 10 cycles. It should be noted that the radix of 17 was chosen due to the target format of IEEE double extended precision, where $n = 64$. This divider can generate double extended precision quotients as well as double precision in 10 cycles. In general, this algorithm requires at least $2\lceil n/b \rceil + 2$ cycles.

4.3 Rounding and Prescaling

Ercegovac and Lang [12] report a high radix division algorithm similar to the previously presented methods. Their algorithm involves obtaining an accurate initial approximation of the reciprocal, scaling both the dividend and divider by this approximation, and then performing multiple iterations of quotient-selection by rounding and partial remainder reduction by multiplication and subtraction. By retiring b bits of quotient in every iteration, it is a radix 2^b algorithm. The algorithm is as follows to compute X/Y:

1. Obtain an accurate approximation of the reciprocal from a table. Rather than using a constant piecewise approximation, this method uses the previously presented technique of linear approximation to the reciprocal. The look-up table stores two coefficients, c_1 and c_2, which have length $b + 5$ and $b + 6$ bits respectively. These entries are indexed using the most significant $\lfloor b/2 \rfloor + 1$ bits of Y. The scaling factor M, which is equivalent to the short reciprocal, is found from c_1, c_2 and the $b+6$ most significant bits of Y by
$$M = -c_1 \times Y_h + c_2$$
Thus, this computation utilizes a carry-save multiplier with an incorporated carry-save adder.

2. Scale Y by the scaling factor M. This involves the carry-save multiplication of the $b + 6$ bit value M and the n bit operand Y to form the $n + b + 5$ bit scaled quantity $Y \times M$.

3. Scale X by the scaling factor M, yielding an $n + b + 5$ bit quantity $X \times M$. This multiplication along with the multiplication of step 2 both can share the $(b + 6) \times (n + b + 5)$ multiplier used in the iterations. In parallel, the scaled divisor $M \times Y$ is assimilated. This involves an $(n + b + 5)$ bit carry-propagate adder.

4. Determine the next quotient digit, needed for the general recurrence:
$$P_{j+1} = rP_j - q_{j+1}(M \times Y)$$

where $P_0 = M \times X$. The choice of scaling factor was made for the purpose of greatly simplifying the quotient-digit selection function. In this scheme, the choice of scaling factor allows for quotient-digit selection to be implemented simply by rounding. Specifically, the next quotient digit is obtained by rounding the shifted partial remainder in carry-save form to the second fractional bit. This can be done using a short carry-save adder and a small amount of additional logic. The quotient-digit obtained through this rounding is in carry-save form, with one additional bit in the least-significant place. This quotient-digit is first recoded into a radix-4 signed-digit set (-2 to +3), then that result is recoded to a radix-4 signed-digit set (-2 to +2). The result of quotient-digit selection by rounding requires $2(b+1)$ bits.

5. Perform the multiplication $q_{j+1} \times z$, where z is the scaled divisor $M \times Y$, then subtract the result from rP_j. This can be performed in one step by a fused multiply/add unit.

6. Perform postcorrection and any required rounding. As discussed previously, postcorrection requires at a minimum sign detection of the last partial remainder and the correction of the quotient.

Throughout the iterations, on-the-fly quotient conversion is used.

The latency of the algorithm in cycles can be calculated as follows. At least one cycle is required to form the linear approximation M. One cycle is required to scale Y, and an additional cycle is required to scale X. $\lceil n/b \rceil$ cycles are needed for the iterations. Finally, one cycle is needed for the postcorrection and rounding. Therefore, the total number of cycles is given by

$$Cycles = \lceil n/b \rceil + 4$$

The hardware required for this algorithm is similar to the Cyrix implementation. One look-up table is required of size $2^{\lfloor b/2 \rfloor}(2b+11)$ bits to store the coefficients of the linear approximation. A $(b+6) \times (b+6)$ carry-save fused multiply/add unit is needed to generate the scaling factor M. One fused multiply/add unit is required of size $(b+6) \times (n+b+5)$ to perform the two scalings and the iterations. A recoder unit is necessary to recode both M and $q_j + 1$ to radix-4. Finally, combinational logic and a short CSA are required to implement the quotient-digit selection by rounding.

4.4 Multiplicative Iterative Division

An extension to the Cyrix short reciprocal non-restoring division algorithm can be made, as reported by Schwarz [33]. Due to the rectangular fused multiply/add unit in the Cyrix algorithm, intermediate results are produced in truncated form. Accordingly, each iteration requires two multiplications. However, by carrying out the iteration in full precision, the two multiplications can be reduced to one, in a manner similar to Wong's accurate quotient approximations. The algorithm is as follows:

1. Initially, obtain a b-bit approximation of the reciprocal, $1/Y_h$.

2. Form $Y' = 1 - (1/Y_h \times Y)$, which has n bits. This requires a multiplication and a subtraction. However, Y' is invariant throughout the iterations and only needs to be computed once. Set the scaled quotient $S_0 = 0$.

3. Perform the iteration:

   $$\begin{aligned} P_{j+1} &= (P_j \times Y')^{tr} \\ S_{j+1} &= S_j + P_j \end{aligned}$$

   where $P_0 = X$, the dividend. The truncation for the partial remainder recurrence requires $n + g$ bits, where g is the number of guard bits required. It can be shown that g is equal to the number of iterations plus one, or $\lceil (n+1)/b \rceil + 1$. Rather than accumulating the quotient estimates in each iteration, the partial remainders P_j are accumulated instead. The scaled quotient is formed by this accumulation. The latency of the iteration is the multiplication to form P_{j+1}, performed in parallel with the addition for the accumulation of the scaled quotient.

4. To form the final quotient and remainder:

   $$\begin{aligned} Q &= S_{last} \times (1/Y_h) \\ R &= X - Q \times Y \end{aligned}$$

This algorithm requires 1 cycle to obtain the initial approximation, 1 multiplication cycle to form the scaled dividend, and 1 cycle to subtract. Each iteration requires 1 cycle for a parallel multiplication and addition. Finalization requires 2 multiplication cycles, 1 subtraction cycle, and 1 cycle for postcorrection and rounding. In total, the algorithm requires $\lceil n/b \rceil + 7$ cycles to compute full precision quotients and remainders. In terms of hardware, it requires a multiplier of width $n+g$, a $2^b \times b$ bit look-up table, and a full-precision carry-propagate adder.

Although interesting from a theoretical aspect, this algorithm is not as attractive as the previously described very high radix algorithms, due to its additional latency in forming the quotient and remainder.

4.5 Analysis

The primary difference between these algorithms are the number and width of multipliers used. These have obvious effects on the latency of the algorithm and the size of the implementation. In the accurate quotient approximations and short-reciprocal algorithms, the next quotient digit is formed by a multiplication $P_h \times (1/Y_h)$ in each iteration. Because the Cyrix implementation only has one rectangular multiply/add unit, each iteration must perform this multiplication in series: first this product is formed as the next quotient digit, then the result is multiplied by Y and subtracted from the current partial remainder to form the next partial remainder, for a total of two multiplications. The accurate quotient approximations method computes $Y' = Y \times (1/Y_h)$ once at the beginning in full precision, and is able to used the result in every iteration. Each iteration still requires two multiplications, but these can be performed in parallel: $P_h \times Y'$ to form the next partial remainder, and $P_h \times (1/Y_h)$ to form the next quotient digit.

The rounding and prescaling algorithm, on the other hand, does not require a separate multiplication to form the next quotient digit. Instead, by scaling *both* the dividend and divisor by the initial reciprocal approximation, the quotient-digit selection function can be implemented by simple rounding logic directly from the redundant partial remainder. Each iteration only requires one multiplication, reducing the area required compared with the accurate quotient approximations algorithm, and decreasing the latency compared with the Cyrix short-reciprocal algorithm. However, because both input operands are prescaled, the final remainder is not directly usable. If a remainder is required, it must be postscaled. Overall, the rounding and prescaling algorithm achieves the lowest latency and cycle time with a reasonable area, while the Cyrix short-reciprocal algorithm achieves the smallest area.

5 Variable Latency Algorithms

Digit recurrence and very high radix algorithms all retire a fixed number of quotient bits in every iteration, while algorithms based on functional iteration retire a quadratically increasing number of bits every iteration. This section discusses methods for implementing dividers that compute results in a variable amount of time. Self-timing is one method of implementing a variable latency divider, as discussed in section 2. This section presents two additional techniques for reducing the average latency of division computation. These techniques take advantage of the fact that the computation for certain operands can be completed sooner than others, or reused from a previous computation. Reducing the worst case latency of a divider requires that all computations made using the functional unit will complete in less than a certain amount of time. In some cases, modern processors are able to use the results from functional units as soon as they are available. Providing a result as soon as it is ready can therefore increase overall system performance. For a given division implementation it may not be possible to reduce the latency for all computations. However, using these techniques, it is possible to reduce the latency for certain computations, with a corresponding increase in system performance.

5.1 Result Caches

Computer applications typically perform computations on input data, and produce final output data based on the results of the computation. Due to the nature of applications, the input operands for one calculation are often the same as those in a previous calculation. For example, in matrix inversion, each entry of the matrix must be divided by the determinant. By recognizing that such redundant behavior exists in applications, it is possible to take advantage of this fact and decrease the effective latency of computations.

Richardson [32] presents the technique of result caching as a means of decreasing the latency of otherwise high-latency operation, such as division. This technique exploits the redundant nature of certain computations by trading execution time for increased memory

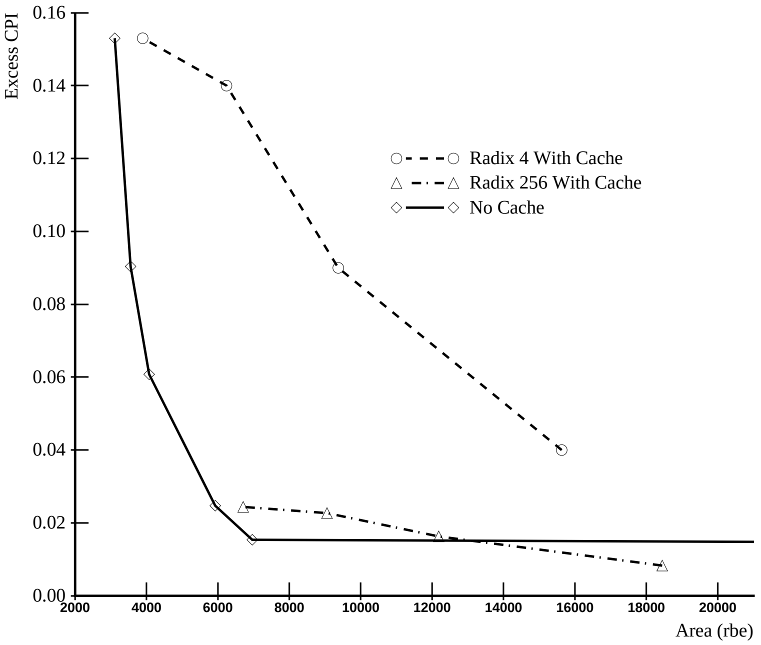


Figure 11: CPI vs area with and without division cache

storage. Once a computation is calculated, it is stored in a *result cache*. The operation of a result cache is as follows. When a targeted operation is issued by the processor, access to the result cache is initiated simultaneously. If the cache access results in a hit, then that result is used, and the operation is halted. If the access misses in the cache, then the operation writes the result into the cache upon completion. Various sized direct-mapped result caches were simulated which stored the results of double precision multiplies, divides, and square roots. The applications surveyed included several from the Spec92 and Perfect Club benchmark suites. Significant reductions in latency were obtained in these benchmarks by the use of a result cache. However, the standard deviation of the resulting latencies across the applications was large.

In another study, Oberman [28] investigated in more detail the performance and area effects of caches that target division, square-root, and reciprocal operations in applications from the SPECfp92 and NAS benchmark suites. Using the register bit equivalent (rbe) model of Mulder [11], a system performance vs. chip area relationship was derived for a cache that targets only double precision division operations. Each cache entry stores a 55 bit mantissa, indexed by the dividend's and divisor's mantissas with a valid bit, for a total of a 105 bit tag. The total storage required for each cache entry is therefore approximately 160 bits. The caches were fully-associative, using random replacement on a miss. Figure 11 shows the relationship derived. From figure 11, is is apparent that if an existing divider has a high latency, as in the case of a radix-4 divider, the addition of a division cache is not area efficient. Rather, better performance per area can be achieved by improving the

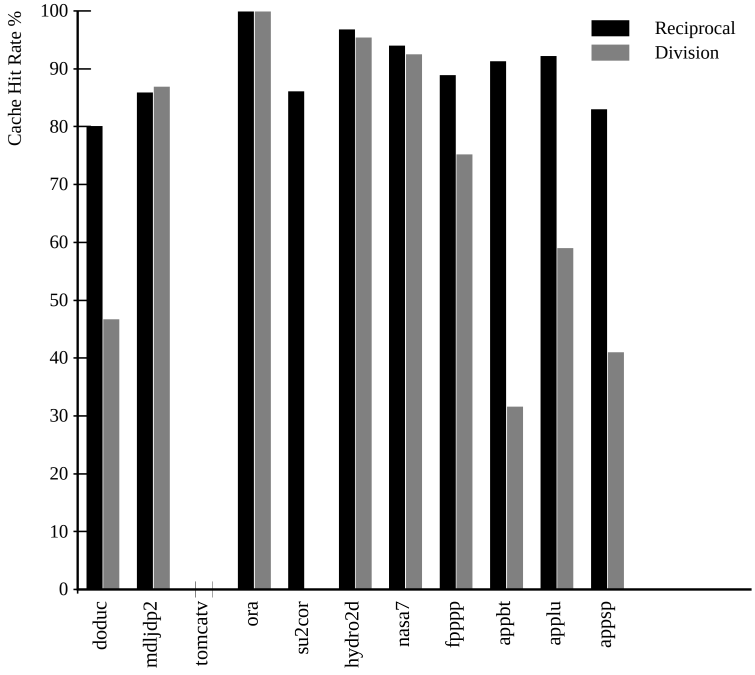


Figure 12: Hit rates for infinite division and reciprocal caches

divider itself, by any of the means discussed previously. Only when the base divider already has a very low latency can the use of division cache be as efficient as simply improving the divider itself.

An alternative to the caching of quotients is a reciprocal cache, where only the reciprocal is stored in the cache. Such a cache can be used when the division algorithm first computes a reciprocal, then multiplies by the dividend to form a quotient, as in the case of the Newton-Raphson algorithm. A reciprocal cache has two distinct advantages over a division cache. First, the tag for each cache entry is smaller, as only the mantissa of the divisor needs to be stored. Accordingly, the total size for each cache entry would be approximately 108 bits, compared with the approximately 160 bits required for a division cache entry. Second, the hit rates are larger, as each entry only needs to match on one operand, not two. A comparison of the hit rates obtained for infinite division and reciprocal caches is shown in figure 12. Similar results are shown in figure 13 for finite sized caches. It is readily apparent that for these applications, the reciprocal cache hit rates are consistently larger and less variable than the division cache hit rates. This study showed that a divider using a reciprocal cache with a size of about eight times that of an 8-bits-in, 8-bits-out ROM look-up table can achieve a speedup of 1.86. Furthermore, the variance of this speedup across different applications is low.

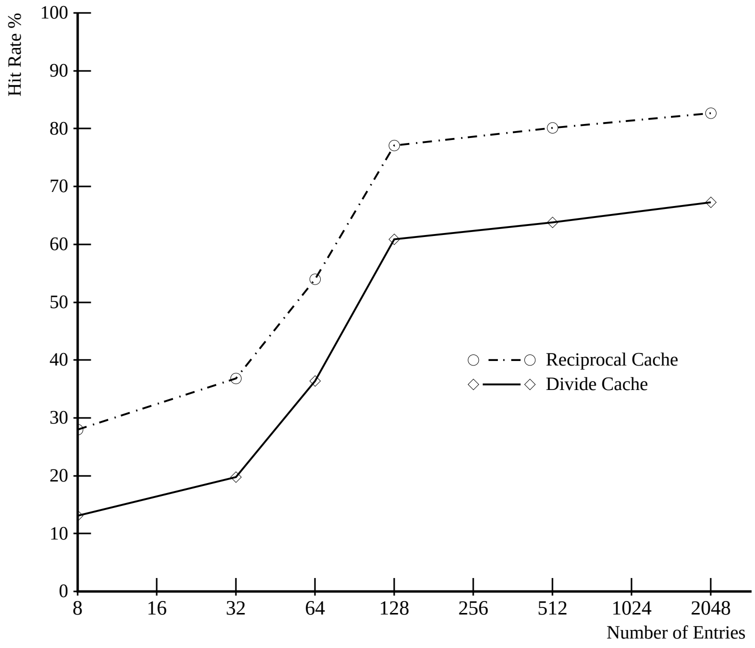


Figure 13: Hit rates for finite division and reciprocal caches

5.2 Speculation of Quotient Digits

A method for implementing an SRT divider that retires a variable number of quotient bits every cycle is reported by Cortadella [3]. The goal of this algorithm is to use a simpler quotient-digit selection function than would normally be possible for a given radix by using fewer bits of the partial remainder and divisor than are specified for the given radix and quotient digit set. This new function does not give the correct quotient digit all of the time. When an incorrect speculation occurs, it is necessary to use at least one iteration to fix the incorrect quotient digit. However, if the probability of a correct digit is high enough, then the resulting lower cycle time due to the simple selection function will offset the increase in the number of iterations required. In this implementation, a variation of the standard SRT recurrence is used:

$$P^s_{j+1} \quad = \quad rP_j - q^s_{j+1}d \tag{13}$$

where q^s_{j+1} is the speculated next quotient digit. After the iteration completes, it is necessary to determine whether the new partial remainder is within the allowable range as set by the redundancy of the quotient digit set. If it is within the allowable range, then $P_{j+1} = P^s_{j+1}$, and the algorithm continues. Otherwise, the partial remainder needs to be corrected. The correction cycle performs:

$$P_{j+1} \quad = \quad P^s_{j+1} - q^c_{j+1}d \tag{14}$$

Accordingly, q^c_{j+1} has the same weight as q^s_{j+1}, and the true next quotient digit is given by

$$q_{j+1} = q^s_{j+1} + q^c_{j+1}$$

An enhancement to this basic scheme is to allow another option in the case of an incorrectly speculated quotient digit. If the error is small, it is possible to advance by a number of quotient bits which is less than a whole digit, and this is known as a *partial advance*. The iteration used for partial-advance is

$$P_{j+1} = pP_j - q^p_{j+1}d \tag{15}$$

where $\log_2 p$ is the number of partially advanced bits. Accordingly, the partially advance quotient digit q^p_{j+1} overlaps with the previous digit by $\log_2 \frac{r}{p}$ bits.

The speculation/quotient-digit selection tables were designed iteratively, performing thousands of random division simulations to determine which quotient values should be returned for each input. These simulations were repeated for various radices using a varying number of divisor and partial remainder bits as input. It is also necessary to determine when an incorrect speculation occurs. While this could be done by implementing a true radix-r quotient-digit selection function in parallel, it is faster and less complex to simply detect whether or not the next partial remainder is within the allowable range. The comparisons that must be performed every iteration are

$$-\rho\hat{d}^c - 2^{-f+1} \leq \hat{P}^c \leq \rho\hat{d}^c - 2^{-f+1} \tag{16}$$

where $\hat{d}^c$ and $\hat{P}^c$ are truncated, carry-assimilated forms of d and P, each with f fractional bits. Because this comparison uses truncated values of d and P, it is possible for the comparison to fail when the speculation is, in fact, correct. This comparison is conservative, so that all incorrect speculations are detected as incorrect, but some correct speculations can also be detected as incorrect. The probability of correctly detecting correct speculations increases with f. The hardware required for digit speculation for both schemes is shown in figure 14.

Several different variations of this implementation were considered for different radices and base divider configurations. A radix-64 implementation was considered which could retire up to 6 bits per iteration. It was found to be 30% faster than the fastest conventional implementation of the same base datapath, which was a radix-8 divider using segments. However, because of the duplication of the quotient-selection logic for speculation, it required about 44% more area than a conventional implementation. A radix-16 implementation (maximum 4 bits per cycle) implementation using the same radix-8 datapath was about 10% faster than a conventional radix-8 divider, with an area reduction of 25%.

5.3 Analysis

Self-timing, result caches, and quotient digit speculation have been shown to be effective methods for reducing the average latency of division computation. A reciprocal cache is an efficient way to reduce the latency for division algorithms that first calculate a reciprocal. While reciprocal caches do require additional area, they require less than that required

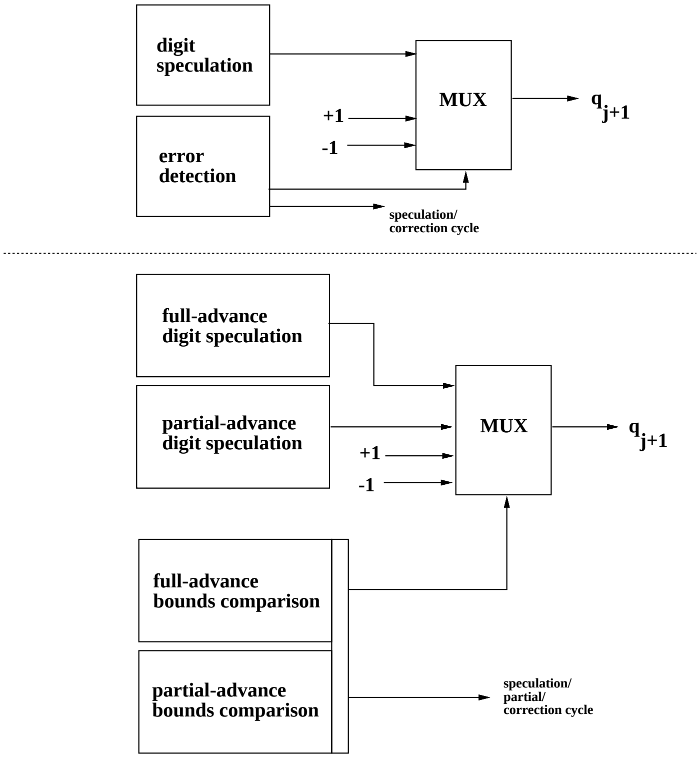

digit speculation
error detection
+1
-1
MUX
q_{j+1}
speculation/ correction cycle
full-advance digit speculation
partial-advance digit speculation
+1
-1
MUX
q_{j+1}
full-advance bounds comparison
partial-advance bounds comparison
speculation/ partial/ correction cycle

Figure 14: Units for digit speculation

by much larger initial approximation look-up tables, while providing a better reduction in latency. Self-timed implementations use circuit techniques to generate results in variable time. The disadvantage of self-timing is the complexity in the clocking, circuits, and testing required for correct operation. Quotient digit speculation is one example of reducing the complexity of SRT quotient-digit selection logic for higher radix implementations.

As processors become more complex and better able to utilize functional units that complete in variable time, variable latency dividers will become more important. The techniques presented in the first sections can be used to minimize the cycle time and worst case latency of the divider implementation. Variable latency techniques can then be included to further reduce the average division latency. This combination of algorithms allows for achieving maximum system performance.

6 Comparison

Four classes of division algorithms have been presented. A summary of several algorithms from these classes is shown in table 1.

Algorithm	Iteration Time	Latency (cycles)	Approximate Area
SRT	Qsel table + (multiple form + subtraction)	$\lceil \frac{n}{r} \rceil$ (+ scale)	Qsel table + CSA
Newton-Raphson	2 serial multiplications	$(2\lceil \log_2 \frac{n}{i} \rceil + 1)t_{mul} + 1$	1 multiplier + table + control
series expansion	2 parallel multiplications	$(\lceil \log_2 \frac{n}{i} \rceil + 2)t_{mul} + 1$	1 multiplier + table + control
accurate quotient approx	1 multiplication	$(\lceil \frac{n}{i} \rceil + 1)t_{mul}$	3 multipliers + table + control
short reciprocal	2 serial multiplications	$2\lceil \frac{n}{i} \rceil t_{mul} + 1$	1 short multiplier + table + control
round/prescale	1 multiplication	$(\lceil \frac{n}{i} \rceil + 2)t_{mul} + 1$	1 multiplier + table + control

Table 1: Summary of algorithms

In this table, n is the number of bits in the input operands, $r = \log_2 radix$ for SRT algorithms, i is the number of bits of accuracy from an initial approximation, and t_{mul} is the latency of the a fused multiply/add unit in cycles. None of the latencies include any additional time required for rounding or normalization.

Table 2 illustrates the effects of algorithm, operand length, width of initial approximation, and multiplier latency on division latency. All operands are IEEE double precision mantissas, with $n = 53$. Table look-ups for initial approximations require 1 cycle. The SRT latencies are separate from the others in that they do not depend on multiplier latency, and they are only a function of the radix of the algorithm for the purpose of this table. For the multiplication-based division algorithms, latencies are shown for multiplier latencies of 1, 2 and 3 cycles. Additionally, latencies are shown for pipelined as well as unpipelined multipliers. A pipelined multiplier can begin a new computation every cycle, while an unpipelined multiplier can only begin after the previous computation has completed.

Algorithm	Pipelined	Initial Approx	Latency (cycles)		
			$t_{mul} = 1$	$t_{mul} = 2$	$t_{mul} = 3$
SRT	-	$r = 2$	27	-	-
		$r = 3$	18	-	-
		$r = 4$	14	-	-
		$r = 8$	7	-	-
Newton-Raphson	no/yes	$i = 8$	8	15	22
	no/yes	$i = 16$	6	11	16
series expansion	no	$i = 8$	9	17	25
	no	$i = 16$	7	13	19
	yes	$i = 8$	9	10	14
	yes	$i = 16$	7	8	11
accurate quotient approximations	no/yes	$i = 8$ (basic)	8	16	24
	no/yes	$i = 16$ (basic)	5	10	15
	no/yes	$i = 13$ and $t = 2$ (adv)	3	6	9
short reciprocal	no/yes	$i = 8$	15	29	
	no/yes	$i = 16$	9	17	
round/prescale	no	$i = 8$	10	19	28
	no	$i = 16$	7	13	19
	yes	$i = 8$	10	18	26
	yes	$i = 16$	7	10	14

Table 2: Latencies for different configurations

From table 2, the advanced version of the accurate quotient approximations algorithm provides the lowest latency. However, the area requirement for this implementation is tremendous, as it requires at least a 736K bits look-up table and three multipliers. For realistic implementations, with $t_{mul} = 2$ or $t_{mul} = 3$ and $i = 8$, the lowest latency is achieved through a series expansion implementation. However, all of the multiplication-based implementations are very close in performance. This analysis shows the extreme dependence of division latency on the multiplier's latency and throughput. A factor of three difference in multiplier latency can result in nearly a factor of three difference in division latency for several of the implementations.

It is difficult for an SRT implementation to perform better than the multiplication-based implementations due to infeasibility of high radix at similar cycle times. However, through the use of scaling and higher redundancy, it may be possible to implement a radix 256 SRT divider that is competitive with the multiplication-based schemes in terms of cycle time and latency. The use of variable latency techniques, such as self-timing, can provide further means for matching the performance of the multiplication-based schemes, without the difficulty in rounding that is intrinsic to the functional iteration implementations.

7 Conclusion

In this paper, the four major classes of division algorithms have been presented. The simplest and most common class found in the majority of modern processors that have hardware division support is digit recurrence, specifically SRT. Recent commercial SRT implementations have included radix 2, radix 4, and radix 8. These implementations have been chosen in part because they operate in parallel with the rest of the floating-point hardware and do not create contention for other units. Additionally, for small radices, it has been possible to meet the tight cycle-time requirements of high performance processors without requiring large amounts of die area. The disadvantage of these SRT implementations is their relatively high latency, as they only retire 1-3 bits of result per cycle. As processors continue to seek to provide an ever-increasing amount of system performance, it becomes necessary to reduce the latency of all functional units, including division.

An alternative to SRT implementations is functional iteration, with the series expansion implementation the most common form. The latency of this implementation is directly coupled to the latency and throughput of the multiplier and the accuracy of the initial approximation. The analysis presented shows that a series expansion implementation provides the lowest latency for reasonable areas and multiplier latencies. If minimizing area is of primary importance, then such an implementation typically shares an existing floating-point multiplier. This has the effect of creating additional contention for the multiplier, possibly reducing the performance of multiplication. An alternative is to provide an additional multiplier, dedicated for division. This can be an acceptable tradeoff if a large quantity of area is available and maximum performance is desired. The main disadvantage with functional iteration is the lack of remainder and the corresponding difficulty in rounding.

Very high radix algorithms are an attractive means of achieving low latency while also providing a true remainder. The only commercial implementation of a very high radix algorithm is the Cyrix short-reciprocal unit. This implementation makes efficient use of a single rectangular multiply/add unit to achieve lower latency than most SRT implementations while still providing a remainder. Further reductions in latency could be possible by using a full-width multiplier, as in the rounding and prescaling algorithm.

The Hal Sparc64 self-timed divider is the only commercial implementation that generates quotients with variable latency depending upon the input operands. It is the circuit design that provides for the lower and variable latency in that implementation. Reciprocal caches have been shown to be an effective means of reducing the latency of division for implementations that generate a reciprocal. Quotient digit speculation is also a reasonable method for reducing SRT division latency.

The importance of division implementations will continue to increase as die area increases and feature sizes decrease. The correspondingly larger amount of area available for floating-point units will allow for implementations of higher radix, lower latency algorithms.

8 Acknowledgements

The authors wish to thank N. Quach for his helpful discussions throughout this work, and G. McFarland for reading and commenting on an early version of this paper.

A Pentium Bug

It is interesting to inspect the radix-4 table of figure 4 somewhat more carefully. It is similar to that used in the radix-4 divider of the Intel Pentium [34]. In such a radix-4 scheme, the table is designed such that the shifted next partial remainder will never be greater than $8/3 \times d$, and therefore, it is always bounded. In the first release of the Pentium, the PLA implementing the quotient-selection logic was missing five entries along the $8/3 \times d$ line, corresponding to the truncated divisor values of 1.0001, 1.0100, 1.0111, 1.1010, and 1.1101. These entries should have contained the next-quotient value of 2. Instead, due to an error in the process of programming the PLA, these five values were set to 0. Accordingly, it was possible for the Pentium divider to produce erroneous results, depending upon whether or not in the process of calculating the quotient those entries were ever used. This gave rise to a large debate with regard to the frequency of erroneous results, which Intel has claimed to be approximately 1 in 9 billion random divisions. Such an experience demonstrates the importance of thorough and correct testing of all components of a CPU, at all stages of the design process.

B Square-Root

B.1 SRT

The recurrence for square-root is conceptually similar to that of division. However, for floating-point representation and normalized operand, an additional requirement is made that the operand must have an even exponent in order to permit the computation of the resulting exponent. Accordingly, it may be necessary to prescale the operand to meet this requirement.

The following recurrence is used at every iteration:

$$\begin{aligned} P_0 &= operand - 1 && (17) \\ P_{j+1} &= rP_j - 2S[j]s_{j+1} - s_{j+1}^2 r^{-(j+1)} && (18) \end{aligned}$$

where P_j is the residual, at iteration j. Additionally, the result is denoted by s and S such that,

$$s = S[n] = \sum_{i=0}^{n} s_i r^{-i}$$

The next result digit is chosen using the following function:

$$s_{j+1} = SEL(\hat{y}_j, \hat{S}[j]) \qquad (19)$$

where $\hat{y}_j$ and $\hat{S}[j]$ are estimates of rP_j and $S[j]$ respectively.

From equations (2) and (18), the similarity between the recurrences for division and square root can be seen. The digit sets and residual representation can be formed in the same manner as is done for division. Additionally, the function used to generate the next result digit is similar to that of division. Thus, division and square-root can and usually do share the same hardware.

Typically, though, the first few bits of the square-root result require a special result-selection function, different than the standard look-up table used for the rest of the iterations. However, Ercegovac [7] shows a method for generating the first few bits of the result without requiring a separate initial table. Specifically, they show a radix-4 square root implementation that generates the first result digit directly from the same table by the addition of three extra gates. This allows for a simplified implementation with no loss of performance.

B.2 Functional Iteration

Square-root can be computed using the Newton-Raphson equation (5) in a manner similar to division. The choice of priming function to solve must be selected carefully. An initial choice might be

$$f(X) \;=\; X^2 - b = 0 \tag{20}$$

which has a root at $X = \sqrt{b}$. However, a direct application of (5) to (20) will result in

$$\begin{aligned} X_1 &= X_0 - \frac{f(X_0)}{f'(X_0)} \\ X_1 &= X_0 - \frac{(X_0^2 - b)}{(2X)} \\ &\vdots \\ X_{i+1} &= \frac{1}{2}(X_i + \frac{b}{X_i}) \end{aligned} \tag{21}$$

Unfortunately, this direct application leads to an iteration which contains a division, which itself is an operation that requires iteration in order to be computed. A better method for computing square-root is to rewrite the the computation as

$$\sqrt{b} = b \times \frac{1}{\sqrt{b}}.$$

This leads to a priming function of

$$f(X) \;=\; \frac{1}{X^2} - b = 0. \tag{22}$$

When the Newton-Raphson equation is applied, the iteration becomes

$$\begin{aligned} X_1 &= X_0 - \frac{f(X_0)}{f'(X_0)} \\ X_1 &= X_0 - \frac{1/X_0^2 - b}{-2/X_0^3} \\ &\vdots \\ X_{i+1} &= \frac{1}{2} \times X_i \times (3 - b \times X_i^2) \end{aligned} \tag{23}$$

This analysis shows that the reciprocal square-root Newton-Raphson iteration requires three multiplications, a subtraction, and a division by two, which can be implemented as a shift. The final square-root is obtained by a multiplying the inverse square-root result with the input operand. By noting that the division operation in (21) can be roughly be replaced by the three multiplications of (23), it can be inferred that the latency of a division operation should be designed to be no worse than a factor of three of the multiplication latency.

The development for square-root calculation using a series expansion implementation proceeds similarly. Initially let $N_0 = b$ and $D_0 = b$. In square-root, like division, the number of iterations can be reduced by prescaling. Accordingly, b should be prescaled by a more accurate approximation of the reciprocal square-root of b, and then the algorithm should be run on the scaled b'. A general relationship can be developed, such that each step of the iteration involves three multiplications

$$N_{i+1} = N_i \times R_i^2 \quad \text{and} \quad D_{i+1} = D_i \times R_i,$$

a subtraction, and a shift,

$$R_{i+1} = \frac{3 - D_i}{2}.$$

From this iteration, N converges quadratically towards 1, and D converges toward $\sqrt{b}$.

References

[1] D. E. Atkins. Higher-radix division using estimates of the divisor and partial remainders. *IEEE Transactions on Computers*, C-17(10), October 1968.

[2] W. S. Briggs and D. W. Matula. A 17x69 Bit multiply and add unit with redundant binary feedback and single cycle latency. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 163–170, July 1993.

[3] J. Cortadella and T. Lang. Division with speculation of quotient digits. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 87–94, July 1993.

[4] D. L. Fowler et al. An accurate, high speed implementation of division by reciprocal approximation. In *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 60–67, September 1989.

[5] D. DasSarma and D. Matula. Measuring the accuracy of ROM reciprocal tables. *IEEE Transactions on Computers*, 43(8):932–940, August 1994.

[6] D. DasSarma and D. Matula. Faithful bipartite ROM reciprocal tables. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 12–25, July 1995.

[7] M. D. Ercegovac and T. Lang. Radix-4 square root without initial PLA. *IEEE Transactions on Computers*, 39(8):1016–1024, August 1990.

[8] M. D. Ercegovac and T. Lang. Simple radix-4 division with operands scaling. *IEEE Transactions on Computers*, C-39(9):1204–1207, September 1990.

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

[10] H. Kabuo et al. Accurate rounding scheme for the Newton-Raphson method using redundant binary representation. *IEEE Transactions on Computers*, 43(1):43–51, January 1994.

[11] J. Mulder et al. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2), February 1991.

[12] M. D. Ercegovac et al. Very high radix division with selection by rounding and prescaling. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 112–119, July 1993.

[13] M. Darley et al. The TMS390C602A floating-point coprocessor for Sparc systems. *IEEE Micro*, 10(3):36–47, June 1990.

[14] M. J. Schulte et al. Optimal initial approximations for the Newton-Raphson division algorithm. *Computing*, 53:233–242, 1994.

[15] S. F. Anderson et al. The IBM System/360 Model 91: Floating-point execution unit. *IBM Journal of Research and Development*, 11:34–53, January 1967.

[16] T. Asprey et al. Performance features of the PA7100 microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[17] J. Fandrianto. Algorithm for high-speed shared radix 4 division and radix 4 square root. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, pages 73–79, May 1987.

[18] J. Fandrianto. Algorithm for high-speed shared radix 8 division and radix 8 square root. In *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 68–75, July 1989.

[19] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8), August 1970.

[20] ANSI/IEEE std 754-1985, IEEE standard for binary floating-point arithmetic.

[21] Intel, i860 64-bit microprocessor programmer's reference manual, 1989.

[22] P. W. Markstein. Computation of elementary function on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, pages 111–119, January 1990.

[23] D. Matula. Highly parallel divide and square root algorithms for a new generation floating point processor. In *SCAN-89, International Symposium on Scientific Computing, Computer Arithmetic, and Numeric Validation*, October 1989.

[24] *Microprocessor Report*, Various issues, 1994.

[25] P. Montuschi and L. Ciminiera. Over-redundant digit sets and the design of digit-by-digit division units. *IEEE Transactions on Computers*, 43(3):269–277, March 1994.

[26] S. Oberman and M. Flynn. Design issues in floating-point division. Technical Report No. CSL-TR-94-647, Computer Systems Laboratory, Stanford University, December 1994.

[27] S. Oberman and M. Flynn. Implementing division and other floating-point operations: a system perspective. To be presented at *SCAN-95, International Symposium on Scientific Computing, Computer Arithmetic, and Numeric Validation*, Wuppertal, Germany, September 1995.

[28] S. Oberman and M. Flynn. On division and reciprocal caches. Technical Report No. CSL-TR-95-666, Computer Systems Laboratory, Stanford University, April 1995.

[29] S. Oberman, N. Quach, and M. Flynn. The design and implementation of a high-performance floating-point divider. Technical Report No. CSL-TR-94-599, Computer Systems Laboratory, Stanford University, January 1994.

[30] J. A. Prabhu and G. B. Zyner. 167 MHz Radix-8 floating point divide and square root using overlapped radix-2 stages. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 155–162, July 1995.

[31] N. Quach and M. Flynn. A radix-64 floating-point divider. Technical Report No. CSL-TR-92-529, Computer Systems Laboratory, Stanford University, June 1992.

[32] S. E. Richardson. Exploiting trivial and redundant computation. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, July 1993.

[33] E. Schwarz. High-radix algorithms for high-order arithmetic operations. Technical Report No. CSL-TR-93-559, Computer Systems Laboratory, Stanford University, January 1993.

[34] H. P. Sharangpani and M. L. Barton. Statistical analysis of floating point flaw in the pentium processor, November 1994.

[35] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point division and square root implementations. Technical Report EE-CEG-94-5, Cornell School of Electrical Engineering, December 1994.

[36] H. Srinivas and K. Parhi. A fast radix-4 division algorithm and its architecture. *IEEE Transactions on Computers*, 44(6):826–831, June 1995.

[37] A. Svoboda. An algorithm for division. *Information Processing Machines*, 9:29–34, 1963.

[38] K. G. Tan. The theory and implementation of high-radix division. In *Proceedings of the 4th IEEE Symposium on Computer Arithmetic*, pages 154–163, June 1978.

[39] G. S. Taylor. Radix 16 SRT dividers with overlapped quotient selection stages. In *Proceedings of the 7th IEEE Symposium on Computer Arithmetic*, pages 64–71, June 1985.

[40] S. Waser and M. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Holt, Rinehart, and Winston, 1982.

[41] T. E. Williams and M. A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.

[42] D. Wong and M. Flynn. Fast division using accurate quotient approximations to reduce the number of iterations. *IEEE Transactions on Computers*, 41(8):981–995, August 1992.