To appear in IEEE Transactions on Computers 1997

Division Algorithms and Implementations

Stuart F. Oberman and Michael J. Flynn

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Abstract

Many algorithms have been developed for implementing division in hardware. These algorithms differ in many aspects, including quotient convergence rate, fundamental hardware primitives, and mathematical formulations. This paper presents a taxonomy of division algorithms which classifies the algorithms based upon their hardware implementations and impact on system design. Division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. Many practical division algorithms are hybrids of several of these classes. 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.

Index Terms: Computer arithmetic, division, floating point, functional iteration, SRT, table look-up, variable latency, very high radix

1 Introduction

In recent years computer applications have increased in their computational complexity. The industry-wide usage of performance benchmarks, such as SPECmarks [1], forces designers of general-purpose microprocessors 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 [2]. A common perception of division is that it is an infrequent operation whose implementation need not receive high priority. However, it has been shown that ignoring its implementation can result in significant system performance degradation for many applications [3]. Extensive literature 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 be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. The basis for these classes is the obvious differences in the hardware operations used in their implementations, such as multiplication, subtraction, and table look-up. Many practical division algorithms are not pure forms of a particular class, but rather are combinations of multiple classes. For example, a high performance algorithm may use a table look-up to gain an accurate initial approximation to the reciprocal, use a functional iteration algorithm to converge quadratically to the quotient, and complete in variable time using a variable latency technique.

In the past, others have presented summaries of specific classes of division algorithms and implementations. Digit recurrence is the oldest class of high speed division algorithms and as a result, a significant quantity of literature has been written proposing digit recurrence algorithms, implementations, and techniques. The most common implementation of digit recurrence division in modern processors has been named *SRT* division by Freiman [4], taking its name from the initials of Sweeney, Robertson [5] and Tocher [6], who discovered the algorithm independently in approximately the same time period. Two fundamental works on division by digit recurrence are Atkins [7], which is the first major analysis of SRT algorithms, and Tan [8], which derives and presents the theory of high-radix SRT division and an analytic method of implementing SRT look-up tables. Ercegovac and Lang [9] is a comprehensive treatment of division by digit recurrence, and it is recommended that the interested reader consult [9] for a more complete bibliography on digit recurrence division. The theory and methodology of division by functional iteration is described in detail in Flynn [10]. Soderquist [11] presents a survey of division by digit recurrence and functional iteration along with performance and area tradeoffs in divider and square root design in the context of a specialized application. Oberman [3] analyzes system level issues in divider design in the context of the SPECfp92 applications.

This study synthesizes the fundamental aspects of these and other works, in order to clarify the division design space. The five 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. The theory of square root computation is an extension of the theory of division. It is shown in [3] that square root operations occur nearly ten times less frequently than division in typical scientific applications, suggesting that fast square root implementations are not crucial to achieving high system performance. However, most of the analyses and conclusions for division can also be applied to the design of square root units. In the following sections, we present the five approaches and then compare these algorithm classes.

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 choices in the design of a digit recurrence divider are the radix, the allowed quotient digits, and the representation of the partial 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 partial 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.

2.1 Definitions

Digit recurrence algorithms use subtractive methods to calculate quotients one digit per iteration. *SRT division* is the name of the most common form of digit recurrence division algorithm. The input operands are assumed to be represented in a normalized floating point format with n bit significands in sign-and-magnitude representation. The algorithms presented here are applied only to the magnitudes of the significands of the input operands. Techniques for computing the resulting exponent and sign are straightforward. The most common format found in modern computers is the IEEE 754 standard for binary floating point arithmetic [12]. 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, and the magnitude of the significand is in the range [1,2). To simplify the presentation, this analysis assumes fractional quotients normalized to the range [0.5,1).

The quotient is defined to comprise k radix-r digits with

$$r = 2^b \tag{1}$$

$$k = \frac{n}{b} \tag{2}$$

where a division algorithm that retires b bits of quotient in each iteration is said to be a radix-r algorithm. Such an algorithm requires k iterations to compute the final n bit result. For a fractional quotient, 1 unit in the last place (ulp) $= r^{-k}$. Note that the radix of the algorithm need not be the same as that of the floating point representation nor the underlying physical implementation. Rather, they are independent quantities. For typical microprocessors that are IEEE 754 conforming, both the physical implementation and the floating point format are radix-2.

The following recurrence is used in every iteration of the SRT algorithm:

$$rP_0 = dividend \tag{3}$$

$$P_{j+1} = rP_j - q_{j+1}divisor \tag{4}$$

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:

$$q_{j+1} = SEL(rP_j, divisor) \tag{5}$$

The final quotient after k iterations is then

$$q = \sum_{j=1}^{k} q_j r^{-j} \tag{6}$$

The *remainder* is computed from the final residual by:

$$\text{remainder} = \begin{cases} P_n & \text{if } P_n \geq 0 \\ P_n + \text{divisor} & \text{if } P_n < 0 \end{cases}$$

Furthermore, the quotient has to be adjusted when $P_n < 0$ by subtracting 1 ulp.

2.2 Implementation of Basic Scheme

A block diagram of a practical implementation of the basic SRT recurrence is shown in fig. 1. The use of the bit-widths in this figure is discussed in more detail in section 2.2.4. The critical path of the implementation is shown by the dotted line.

From equations (4) and (5), each iteration of the recurrence comprises the following steps:

- Determine next quotient digit q_{j+1} by the quotient-digit selection function, a look-up table typically implemented as a PLA or combinational logic.
- Generate the product $q_{j+1}\times$ *divisor*.
- Subtract $q_{j+1}\times$ *divisor* from the shifted partial remainder $r \times P_j$

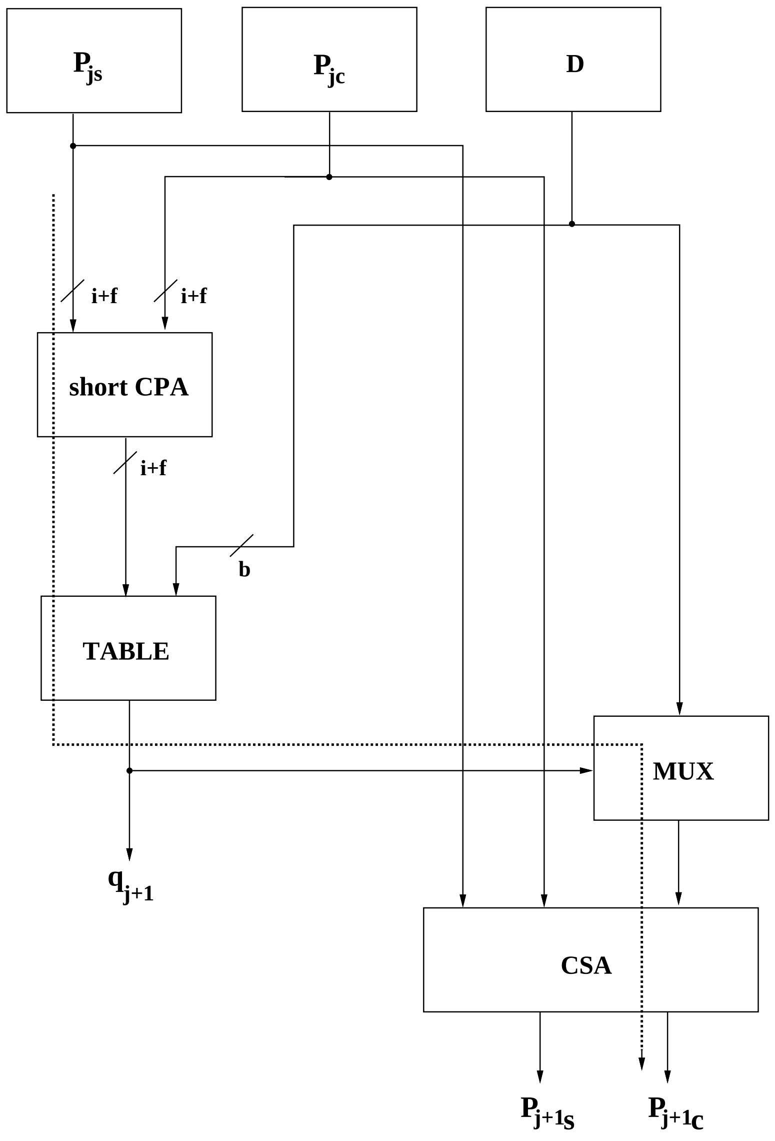

Figure 1: Basic SRT Divider Topology

Each of these components contributes to the overall cost and performance of the algorithm. Depending on certain parameters of the algorithm, the execution time and corresponding cost can vary widely.

2.2.1 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. We 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 f when the radix is increased from r to r^f. For example, a radix 4 algorithm retires 2 bits of quotient in every iteration. Increasing to a radix-16 algorithm allows for retiring 4 bits in every iteration, halving the latency. This reduction does not come for free. As the radix increases, the quotient-digit selection becomes more complicated. It is seen from fig. 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. While the number of cycles may have been reduced due to the increased radix, the time per cycle may have increased. As a result, the total time required to compute an n bit quotient is not reduced as expected. 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.

2.2.2 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, we use 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

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

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* determines the selection intervals. 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 plots the shifted partial remainder vs. 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 fig. 2 with $r = 4$ and $a = 2$. The shaded regions are the overlap regions where more than one quotient digit may be selected.

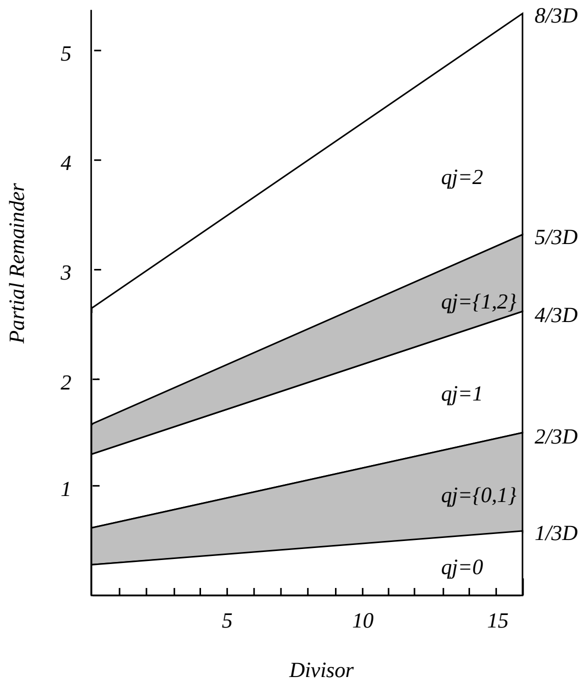

Figure 2: P-D diagram for radix-4

2.2.3 Residual Representation

The residual can be represented in two different forms, either *redundant* or *nonredundant* form. Conventional two's complement representation is an example of a *nonredundant* form, while carry-save two'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 requires a full-width adder requiring carry propagation, increasing the cycle time. If the residual is computed in a redundant form, a carry-free adder, such as a carry-save adder (CSA), can be used in the recurrence, minimizing the cycle time. However, the quotient-digit selection, which is 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 has to be converted to a conventional representation. At a minimum, it is necessary to determine the sign of the final remainder in order to implement a possible quotient correction step, as discussed previously.

2.2.4 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 exactly known to select the next quotient digit. It is only necessary to know the exact range of the residual in fig. 2. 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 small look-up 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.

Let $\hat{d}$ be an estimate of the divisor using the b most significant bits of the true divisor, excluding the implied leading one, such that $b = \delta - 1$. Let $\hat{P}_j$ be an estimate of the partial remainder using the c most significant bits of the true partial remainder, comprising i integer bits and f fractional bits. The usage of these estimates is illustrated in fig. 1. To determine the minimum values for b, i, and f, consider the uncertainty region in the resulting estimates $\hat{d}$ and $\hat{P}_j$. The estimates 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 are each strictly less than 1 ulp. Additionally, if the partial remainder is kept in a carry-save form, ϵ_p is strictly less than 2 ulps. This is due to the fact that both the sum and the carry values have been truncated, and each can have an error strictly less than 1 ulp. When the two are summed to form a nonredundant estimate of the partial remainder, the actual error is strictly less than 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 carry-save representation of the partial remainder, this involves calculating the maximum and minimum ratios of the shifted partial remainder and divisor, and ensuring that these ratios both lie in the same selection interval:

$$ratio_{max} = \begin{cases} \frac{r\hat{P}_j + \epsilon_{p(cs)}}{\hat{d}} & \text{if } P_j \geq 0 \\ \frac{r\hat{P}_j}{\hat{d}} & \text{if } P_j < 0 \end{cases} \tag{7}$$

$$ratio_{min} = \begin{cases} \frac{r\hat{P}_j}{\hat{d} + \epsilon_d} & \text{if } P_j \geq 0 \\ \frac{r\hat{P}_j + \epsilon_{p(cs)}}{\hat{d} + \epsilon_d} & \text{if } P_j < 0 \end{cases} \tag{8}$$

The minimum and maximum values of the ratio must 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 r, redundancy ρ, values of b, i and f, and error terms ϵ_p and ϵ_d. The table can take as input the partial remainder estimate directly in redundant form, or it can use the output of a short carry-propagate adder that converts the redundant partial remainder estimate to a nonredundant representation. The use of an external short adder reduces the complexity of the table implementation, as the number of partial remainder input bits are halved. However, the delay of the quotient-digit selection function increases by the delay of the adder. Such an adder is shown in fig. 1 as the CPA component.

Oberman [13] presents a methodology for performing this analysis along with several techniques for minimizing table complexity. Specifically, the use of Gray-coding for the quotient-digits is proposed to allow for the automatic minimization of the quotient-digit selection logic equations, achieving near optimal delay and area for a given set of table parameters. The use of a short carry-assimilating adder with a few more input bits than output bits to assimilate a redundant partial remainder before input into the table reduces the table complexity. Simply extending the short CPA before the table by two bits can reduce table delay by 16% and area by 9% for a minimally-redundant radix-4 table. Given the choice of using more divisor or partial remainder bits, reducing the number of remainder bits, while reducing the size of the short carry-assimilating adder, increases the size and delay of the table, offsetting the possible performance gain due to the shorter adder.

2.3 Increasing Performance

Several techniques have been reported for improving the performance of SRT division including [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Some of these approaches are discussed below.

2.3.1 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 fig. 3. In this implementation, the critical path is equal to:

$$t_{iter} \quad = \quad t_{reg} + 2(t_{qsel} + t_{qDsel} + t_{CSA}) \tag{9}$$

One advantage of unrolling the iterations by duplicating the lower-radix hardware is that the contribution of register overhead to total delay is reduced. The more the iterations are unrolled, the less of an impact register overhead has on total delay. However, the added area due to each stage must be carefully

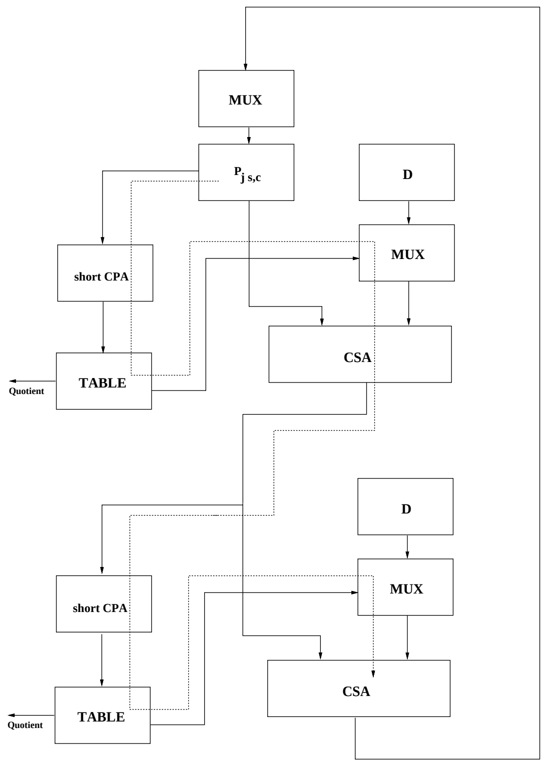

Figure 3: Higher radix using hardware replication

considered.

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. In the HP PA-7100, 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 [25]. In the succeeding generation HP PA-8000 processor, due to a higher system clock frequency, the divider is clocked at the same frequency as the rest of the CPU, increasing the latency (in cycles) by a factor of two [26].

The radix-16 divider in the AMD 29050 microprocessor [27] is another example of achieving higher radix by clocking a lower radix core at a higher frequency. In this implementation, a maximally-redundant

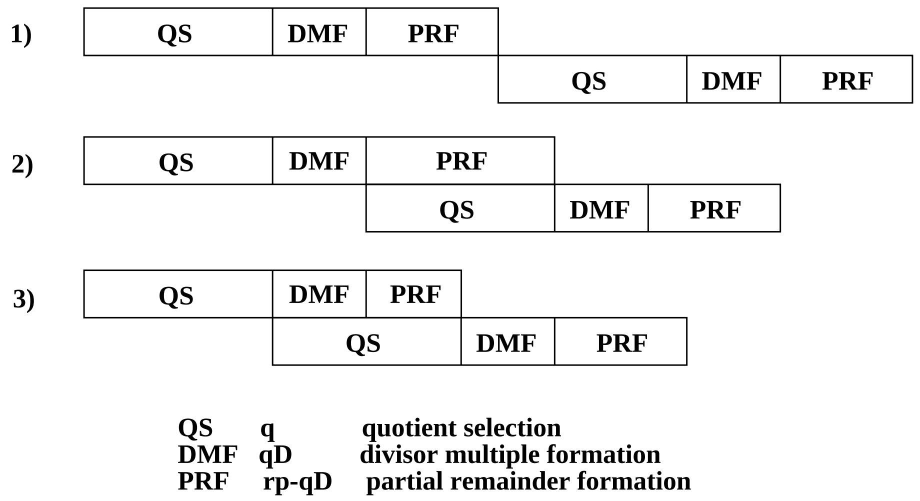

Figure 4: Three methods of overlapping division components

radix-4 stage is clocked at twice the system clock frequency to form a radix-16 divider. Two dynamic short CPAs are used in front of the quotient selection logic, such that in one clock phase the first CPA evaluates and the second CPA precharges, while in the other clock phase the first CPA precharges and the second CPA evaluates. In this way, one radix-4 iteration is completed in each phase of the clock, for a total of two iterations per clock cycle.

2.3.2 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 [23]. This is illustrated in fig. 4. The standard approach is represented in this figure by approach 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, approaches 2 or 3 may be more desirable. Approach 2 is appropriate when the overlap is dominated by partial remainder formation time. This would be the case when the partial remainder is not kept in a redundant form. Approach 3 is appropriate when the overlap is dominated by quotient selection, as is the case when a redundant partial remainder is used.

2.3.3 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 [9],[15],[23]. The quotient-

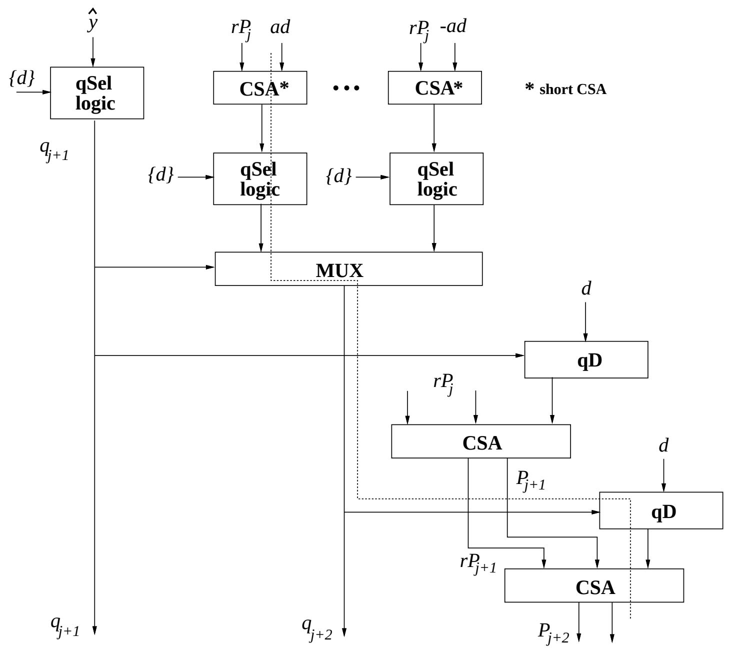

Figure 5: Higher radix by overlapping quotient selection

digit selection of stage $j + 2$ is overlapped with the quotient-digit selection of stage $j + 1$, as shown in fig. 5. This is accomplished by calculating an estimate of the next partial remainder and the quotient-digit selection for q_{j+2} conditionally for all $2a + 1$ values of the previous quotient digit q_{j+1}, where a is the maximum allowed quotient digit. Once the true value of q_{j+1} is known, it selects the correct value of q_{j+2}. As can be seen from fig. 5, the critical path is equal to:

$$t_{iter} = t_{reg} + t_{qsel} + t_{qDsel} + 2t_{CSA} + t_{mux(data)} \qquad (10)$$

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

$$\Delta t_{iter} = t_{qsel} + t_{qDsel} - t_{mux(data)} \qquad (11)$$

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 requires $(2a + 1)^k$ replicated quotient-selection functions. Because of this exponential growth in hardware, only very small values of k are feasible in practice.

Prabhu [28] discusses a radix-8 shared square root design with overlapped quotient selection used 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 three possible quotient digits of the second stage. In the worst case, this involves 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.

2.3.4 Overlapping Remainder Computation

A further optimization that can be implemented is the overlapping of the partial remainder computation, also used in the UltraSPARC. By replicating the hardware to compute the next partial remainder for each possible quotient digit, the latency of the recurrence is greatly reduced.

Oberman [20] and Quach [21] report optimizations for radix-4 implementations. For radix-4, it might seem that because of the five possible next quotient digits, five copies of partial remainder computation hardware are 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 copies of partial remainder computation hardware to three: 0, ±1, and ±2. However, from an analysis of a radix-4 quotient-digit selection table, the boundary between quotient digits 0 and 1 is readily 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 partial remainder computation hardware can be reduced to two: 0 or ±1, and ±2. A block diagram of a radix-4 divider with overlapped remainder computation is shown

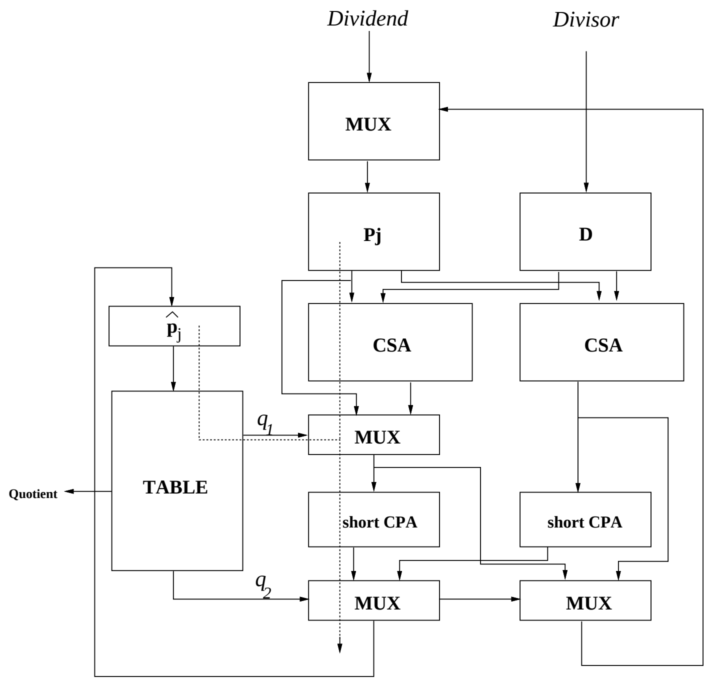

Figure 6: Radix-4 with overlapped remainder computation

in fig. 6. 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: $\max(tq_1, t_{CSA}) + 2t_{mux} + t_{shortCPA} + t_{reg}$. Thus, at the expense of duplicating the remainder computation hardware once, the cycle time of the standard radix-4 divider is nearly halved.

2.3.5 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 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],[15].

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

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

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, the quotient-digit selections can be overlapped as discussed previously to reduce the cycle time. In the worst case, this overlapping involves two short CSAs, two short CPAs, and three instances of the radix-4 quotient-digit selection logic. However, 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:

$$t_{iter} = t_{reg} + t_{signest} + t_{mux} + t_{CSA} + t_{shortCPA} + t_{qlsel} + t_{mux(data)} \tag{12}$$

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.

2.3.6 Operands Scaling

In higher radix dividers the cycle time is generally determined by quotient-digit selection. The complexity of quotient-digit selection increases exponentially for increasing radix. To decrease cycle time, it may be desirable to reduce the complexity of the quotient-digit selection function using techniques beyond those presented, at the cost of adding additional cycles to the algorithm. From an analysis of a quotient-digit selection function, the maximum overlap between allowed quotient digits 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 [14],[29]. 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

$$1 - \alpha \leq z = Md \leq 1 + \beta \tag{13}$$

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, the simpler the quotient-digit selection function. 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 Ercegovac [14] 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 uses a 3-operand adder. If a CSA is already used for the division recurrence, no additional CSAs are required and the scalings proceed in sequence. To determine the scaling factor for each operand, a small table yields the proper factors to add or subtract in the CSA to determine 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. The dividend is scaled in parallel with the divisor assimilation, and it can be used directly in redundant form as the initial partial remainder.

Enhancements to the basic prescaling algorithms have been reported by Montuschi [18] and Srinivas [22]. Montuschi uses an over-redundant digit set combination with operand prescaling. The proposed radix-4 implementation uses the 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 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 [22] 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 represented by 2 bits. Accordingly, the quotient-selection function uses only 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, using $n = 53$ 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 53 CSAs.

2.4 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 is 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 [30].

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:

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

and

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

with the initial conditions that $Q_0 = QM_0 = 0$. From these conditions on the values of Q_k and QM_k, all of the updating 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 Rounding

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. One ulp is conditionally added based on the rounding mode selected and these two values. 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.

The previously described on-the-fly conversion can be extended to perform on-the-fly rounding [31]. This technique requires keeping a third version of the quotient at all times QP_k, where $QP_k = Q_k + r^{-k}$. The values of this register for step $k+1$ is defined by:

$$QP_{k+1} = \begin{cases} (QP_k, 0) & \text{if } q_{k+1} = r - 1 \\ (Q_k, (q_{k+1} + 1)) & \text{if } -1 \leq q_{k+1} \leq r - 2 \\ (QM_k, (r - |q_{k+1}| + 1)) & \text{if } q_{k+1} < -1 \end{cases}$$

Correct rounding requires the computation of the sign of the final remainder and the determination of whether the final remainder is exactly zero. Sign detection logic can require some form of carry-propagation detection network, such as in standard carry-lookahead adders, while zero-remainder detection can require the logical ORing of the assimilated final remainder. Faster techniques for computing the sign and zero-remainder condition are presented in [9]. The final quotient is appropriately selected from the three available versions.

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 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 [3] 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.

3.1 Newton-Raphson

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

$$Q = a/b = a \times (1/b), \tag{14}$$

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 [10]. 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:

$$f(X) \quad = \quad 1/X - b = 0. \tag{15}$$

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

$$x_{i+1} \quad = \quad x_i - \frac{f(x_i)}{f'(x_i)} \tag{16}$$

The Newton-Raphson equation of (16) is then applied to (15), and this iteration is then used to find an approximation to the reciprocal:

$$X_{i+1} \quad = \quad X_i - \frac{f(X_i)}{f'(X_i)} = X_i + \frac{(1/X_i - b)}{(1/X_i^2)} = X_i \times (2 - b \times X_i) \tag{17}$$

The corresponding error term is given by

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

and thus the error in the reciprocal decreases quadratically after each iteration. As can be seen from (17), each iteration involves two multiplications and a subtraction. The subtraction is equivalent to forming the two's complement 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.

Rather than performing a complete two's complement operation at the end of each iteration to form $(2 - b \times X_i)$, it is possible instead to simply implement the one's complement of $b \times X_i$, as was done in the IBM 360/91 [32] and the Astronautics ZS-1 [33]. This only adds a small amount error, as the one's and two's complement operations differ only by 1 ulp. The one's complement avoids the latency penalty of carry-propagation across the entire result of the iteration, replacing it by a simple inverter delay.

While the number of operations per iteration is constant, 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. Section 5 explores in more detail the use of look-up tables to increase performance.

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* [34]. Consider the familiar Taylor series expansion of a function $g(y)$ at a point p,

$$g(y) \quad = \quad g(p) + (y-p)g'(p) + \frac{(y-p)^2}{2!}g''(p) + \cdots + \frac{(y-p)^n}{n!}g^{(n)}(p) + \cdots. \tag{18}$$

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

$$q \quad = \quad \frac{a}{b} = a \times g(y), \tag{19}$$

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

$$g(y) \quad = \quad \frac{1}{1+y} = 1 - y + y^2 - y^3 + y^4 - \cdots. \tag{20}$$

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

$$q \quad = \quad a \times [(1-y)(1+y^2)(1+y^4)(1+y^8)\cdots]. \tag{21}$$

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 (21).

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

$$N_{i+1} = N_i \times R_i \quad \text{and} \quad D_{i+1} = D_i \times R_i \tag{22}$$

and a two's complement operation,

$$R_{i+1} = 2 - D_{i+1} \tag{23}$$

After i steps,

$$N_i = a \times [(1-y)(1+y^2)(1+y^4)\cdots(1+y^{2i})] \tag{24}$$

$$D_i = (1-y^{2i}) \tag{25}$$

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 (24) and the series expansion of q in (21). So long as b is normalized in the range $0.5 \leq b < 1$, then $y < 1$, each correction factor $(1 + y^{2^i})$ 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 (24) using the substitution $y = b - 1$ with equation (17) 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. 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. Thus, the series expansion requires the overhead of one more multiplication operation as compared to Newton-Raphson. Each iteration in both algorithms comprises two multiplications and a two's complement operation. From (17), the multiplications in Newton-Raphson are dependent operations. In the series expansion iteration, the two multiplications of the numerator and denominator are independent operations and may occur concurrently. As a result, a series expansion implementation can take advantage of a pipelined multiplier to obtain higher performance in the form of lower latency per operation. In both iterations, unused cycles in the multiplier can be used to allow for more than one division operation to proceed concurrently. Specifically, a pipelined multiplier with a throughput of 1 and a latency of l can have l divisions operating simultaneously, each initiated at 1 per cycle. 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 [33], Intel i860 [35], and the IBM RS/6000 [36]. The series expansion was used in the IBM 360/91 [32] and TMS390C602A [37]. 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 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. The series expansion iteration, which converges directly to the quotient, 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 the quotient to be correctly rounded.

There have been three main techniques used in previous implementations to compute rounded results when using division by functional iteration. The IBM 360/91 implemented division using Goldschmidt's algorithm [32]. In this implementation, 10 extra bits of precision in the quotient were computed. A hot-one was added in the LSB of the guard bits. If all of the 10 guard bits were ones, then the quotient was rounded up. This implementation had the advantage of the fastest achievable rounding, as it did not require any additional operations after the completion of the iterations. However, while the results could be considered "somewhat round-to-nearest," they were definitely not IEEE compliant. There was no concept of exact rounding in this implementation.

Another method requires a datapath twice as wide as the final result, and it is the method used to implement division in the IBM RS/6000 [36]. 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. The RS/6000 implementation uses its fused multiply-accumulate for all of the operations to guarantee accuracy greater than $2n$ bits throughout the iterations. After the completion of the additional iteration,

$$Q' = \text{estimate of } Q = \frac{a}{b} \text{ accurate to } 2n \text{ bits}$$

A remainder is calculated as

$$R = a - b \times Q' \tag{26}$$

A rounded quotient is then computed as

$$Q'' = Q' + R \times b \tag{27}$$

where the final multiply-accumulate is carried in the desired rounding mode, providing the exactly rounded result. The principal disadvantage of this method is that it requires one additional full iteration of the algorithm, and it requires a datapath at least two times larger than is required for non-rounded results.

A third approach is that which was used in the TI 8847 FPU [37]. In this scheme, the quotient is also computed with some extra precision, but less than twice the desired final quotient width. To determine the sticky bit, the final remainder is directly computed from:

$$\begin{aligned} Q &= \frac{a}{b} - R \\ R &= a - b \times Q \end{aligned}$$

It is not necessary to compute the actual magnitude of the remainder; rather, its relationship to zero is required. In the worst case, a full-width subtraction may be used to form the true remainder R. Assuming sufficient precision is used throughout the iterations such that all intermediate multiplications are at least $2n$ bits wide for n bit input operands, the computed quotient is less than or equal to the infinitely precise quotient. Accordingly, the sticky bit is zero if the remainder is zero and one if it is nonzero. If truncated multiplications are used in the intermediate iterations, then the computed quotient converges toward the exactly rounded result, but it may be either above or below it. In this case, the sign of the remainder is also required to detect the position of the quotient estimate relative to the true quotient. Thus, to support exact rounding using this method, the latency of the algorithm increases by at least the multiplication delay necessary to form $Q \times b$, and possibly by a full-width subtraction delay as well as zero-detection and sign-detection logic on the final remainder. In the TI 8847, the relationship of the quotient estimate and the true quotient is determined using combinational logic on 6 bits of both a and $Q \times b$ without explicit computation of R.

In Schwarz [38], a technique is presented for avoiding the final remainder calculation for certain cases. By computing one extra bit of precision in the quotient, half of the cases can be completed by inspection of only the extra guard bit, with no explicit remainder calculation required. Oberman [39] shows that by using m guard bits in the quotient estimate, a back-multiplication and subtraction for forming the remainder are required for only 2^{-m} of all cases. Further, when a true remainder is required, the full-width subtraction can be replaced by simple combinational logic using only the LSB's of the dividend and the back product of the quotient estimate and the divisor, along with the sticky bit from the multiplier. The combination of these techniques allow for an increase in average division performance.

An additional method has been proposed for rounding in Newton-Raphson implementations that uses a signed-digit multiplier [40]. 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.

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, we 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 are similar in that they 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

In the high radix algorithm proposed by Wong [41], truncated versions of the normalized dividend X and divisor 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, 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 use only 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:

$$P' = P - P_h \times (1/Y_h) \times Y, \tag{28}$$

where P_0 = X. Since all products have already been formed, this step only involves a subtraction.

4. Compute the new quotient as

$$\begin{aligned} Q' &= Q + (P_h/Y_h) \times (1/2^j) \\ &= Q + P_h \times (1/Y_h) \times (1/2^j) \end{aligned} \tag{29}$$

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.

Wong also proposes an advanced version of this algorithm [41] using 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 higher order terms of the Taylor series 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

$$b_i = (m \times t - t) + \lceil \log_2 t \rceil - (m \times i - m - i) \qquad (30)$$

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. To form the final estimate of $1/Y_h$ from the t estimates, additional multiply and add hardware is required. Such a method for forming a more accurate initial approximation is investigated in more detail in section 5. 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 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.

A slower implementation of this algorithm might use 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, with a resulting latency 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, but with a latency of only 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 [42],[43]. 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 to Wong:

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

$$P' = P - P_h \times (1/Y_h) \times Y \tag{31}$$

$$Q' = Q + P_h \times (1/Y_h) \times (1/2^j) \tag{32}$$

where P_0 is the dividend X. In this implementation, the two multiplications of (31) 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 (32) can be reused from the result computed for (31). Only one multiplication is required in the accurate quotient method because the product $(1/Y_h) \times Y$ is computed once at the beginning in full precision, and can be reused on every iteration. The Cyrix multiplier only produces limited precision results, 19 bits, and thus the multiplication by Y is 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. With a larger table, the initial approximation can be obtained in as little as 1 cycle, reducing the total cycle count to 10 cycles. The radix of 2^{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 [44] report a high radix division algorithm which 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 to form the scaling factor M. Rather than using a simple table look-up, this method uses a linear approximation to the reciprocal, a technique discussed in the next section.

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:

$$P_{j+1} = rP_j - q_{j+1}(M \times Y) \tag{33}$$

   where $P_0 = M \times X$. 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.

5 Look-up Tables

Functional iteration and very high radix division implementations can both benefit from a more accurate initial reciprocal approximation. Further, when a only low-precision quotient is required, it may be sufficient to use a look-up table to provide the result without the subsequent use of a refining algorithm. Methods for forming a starting approximation include direct approximations and linear approximations. More recently, partial product arrays have been proposed as methods for generating starting approximations while reusing existing hardware.

5.1 Direct Approximations

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. 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 bits after the leading bit in the $m+1$ bit fraction reciprocal approximation R_0 which has the range $0.5 < recip \leq 1$. The total size of such a reciprocal table is $2^k m$ bits. The truncated operand is represented as $1.b'_1 b'_2 \cdots b'_k$, and the output reciprocal approximation is $0.1b'_1 b'_2 \cdots b'_m$. The only exception to this is when the input operand is exactly 1, in which case the output reciprocal should also be exactly 1. In this case, separate hardware is used to detect this. All input values have a leading-one that can be implied and does not index into the table. Similarly, all output values have a leading-one that is not explicitly stored in the table. The design of a reciprocal table starts with a specification for the minimum accuracy of the table, expressed in bits. This value dictates the number of bits (k) used in the truncated estimate of b as well as the minimum number of bits in each table entry (m). 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 $1.b'_1 b'_2 \cdots b'_k 1$. The reciprocal of the mid-point is rounded by adding $2^{-(m+2)}$ and then truncating the result to $m+1$ bits, producing a result of the form $0.1b'_1 b'_2 \cdots b'_m$. Thus, the approximation to the reciprocal is found for each entry i in the table from:

$$R_i = \left\lfloor 2^{m+1} \times \left(\frac{1}{\hat{b} + 2^{-(k+1)}} + 2^{-(m+2)} \right) \right\rfloor / 2^{m+1} \qquad (34)$$

where $\hat{b} = 1.b'_1 b'_2 \cdots b'_k$.

Das Sarma [45] has shown that the piecewise-constant approximation method for generating reciprocal look-up tables minimizes the maximum relative error in the final result. The maximum relative error in the reciprocal estimate ϵ_r for a k-bits-in k-bits-out reciprocal table is bounded by:

$$|\epsilon_r| \;=\; \left|R_0 - \frac{1}{b}\right| \leq 1.5 \times 2^{-(k+1)} \tag{35}$$

and thus the table guarantees a minimum precision of $k + 0.415$ bits. It is also shown that with $m = k + g$, where g is the number of output guard bits, the maximum relative error is bounded by:

$$|\epsilon_r| \;\leq\; 2^{-(k+1)} \times \left(1 + \frac{1}{2^{g+1}}\right) \tag{36}$$

Thus, the precision of a k-bits-in, $(k + g)$-bits-out reciprocal table for $k \geq 2$, $g \geq 0$, is at least $k + 1 - \log_2(1 + \frac{1}{2^{g+1}})$. As a result, generated tables 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.

In a more recent study, Das Sarma [46] describes bipartite reciprocal tables. These tables use separate table lookup of the positive and negative portions of a reciprocal value in borrow-save form, but with no additional multiply/accumulate operation required. Instead, it is assumed that the output of the table is used as input to a multiplier for subsequent operations. In this case, it is sufficient that the table produce output in a redundant form that is efficiently recoded for use by the multiplier. Thus, the required output rounding can be implemented in conjunction with multiplier recoding for little additional cost in terms of complexity or delay. This method is a form of linear interpolation on the reciprocal which allows for the use of significantly smaller tables. These bipartite tables are 2 to 4 times smaller than 4-9 bit conventional reciprocal tables. For 10-16 bit tables, bipartite tables can be 4 to more than 16 times smaller than conventional implementations.

5.2 Linear Approximations

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:

$$P(b) \;=\; C_0 + C_1 b + C_2 b^2 + C_3 b^3 + \cdots \tag{37}$$

To get a first order or linear approximation, the coefficients C_0 and C_1 are stored in a look-up table, and a multiplication and an addition are required. As an example, a linear function is chosen such as

$$P(b) \quad = \quad -C_1 \times b + C_0 \tag{38}$$

in order to approximate $1/b$ [47]. The two coefficients C_0 and C_1 are read from two look-up tables, each using the k most significant bits of b to index into the tables and each returning m bits. The total error of the linear approximation ϵ_{la} is the error due to indexing with only k bits plus the truncation error due to only storing m bit entries in each of the tables, or

$$|\epsilon_{la}| \quad < \quad 2^{-(2k+3)} + 2^{-m} \tag{39}$$

Setting $m = 2k + 3$ yields $|\epsilon_{la}| < 2^{-(2k+2)}$, and thus guaranteeing a precision of $2k + 2$ bits. The total size required for the tables is $2^k \times m \times 2$ bits, and a $m \times m$ bit multiply/accumulate unit is required.

Schulte [48] proposes 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. For constant and linear approximations, they present the tradeoff between n, the number of iterations, k, the number of bits used as input to the table, and the effects on the absolute error of the final result. In general, linear approximations guarantee more accuracy than constant approximations, but they require additional operations which may affect total delay and area.

Ito [47] proposes an improved initial approximation method similar to a linear approximation. A modified linear function

$$A_1 \times (2\hat{b} + 2^{-k} - b) + A_0 \tag{40}$$

is used instead of $-C_1 \times b + C_0$ for the approximation to $1/b$. In this way, appropriate table entries are chosen for A_0 and A_1. The total table size required is $2^k \times (3k + 5)$ bits, and the method guarantees a precision of $2.5k$ bits.

5.3 Partial Product Arrays

An alternative to look-up tables for starting approximation is the use of partial product arrays [49],[50]. 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.

6 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 an increasing number of bits every iteration. However, all of these algorithms complete in a fixed number of cycles. This section discusses methods for implementing dividers that compute results in a variable amount of time. The DEC Alpha 21164 [51] uses a

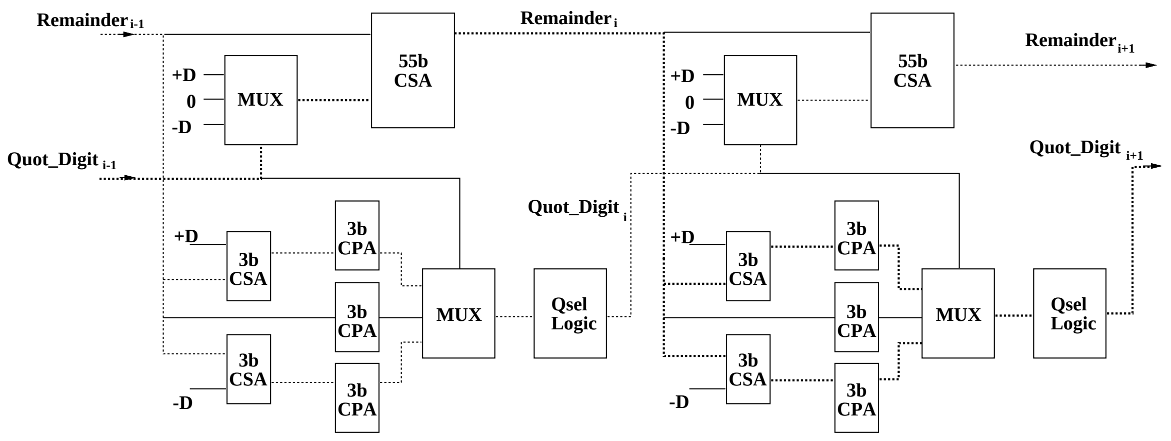

Figure 7: Two stages of self-timed divider

simple normalizing non-restoring division algorithm, which is a predecessor to fixed-latency SRT division. Whenever a consecutive sequence of zeros or ones is detected in the partial remainder, a similar number of quotient bits are also set to zero, all within the same cycle. In [51], it is reported that an average of 2.4 quotient bits are retired per cycle using this algorithm.

This section presents three 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 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.

6.1 Self-Timing

A recent SRT implementation returning quotients with variable latency is reported by Williams [24]. 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 fig. 7. Because it uses self-timing, no explicit registers are required to store the intermediate remainder. Accordingly, the critical path does not contain delay due to partial remainder register overhead. 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 fig. 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, the Sparc64, also implements a version of this self-timed divider, producing IEEE double precision results in about 7 cycles [52].

6.2 Result Caches

In typical 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 and decrease the effective latency of computations.

Richardson [53] 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 storage. Once a computation is calculated, it is stored in a *result cache*. 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 [54] investigates 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) area model of Mulder [55], a system performance vs. chip area relationship was derived for a fully-associative cache that targets only double precision division

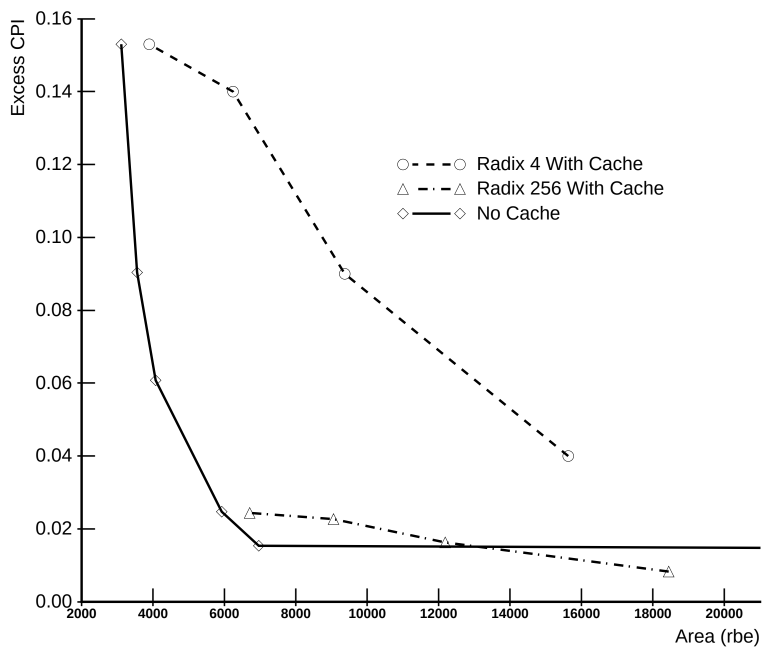

Figure 8: CPI vs area with and without division cache

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. Fig. 8 shows the relationship derived in comparison with the area required for various SRT dividers. From fig. 8, is is apparent that if an existing divider has a high latency, as in the case of a radix-4 SRT divider, the addition of a division cache is not area efficient. Rather, better performance per area can be achieved by improving the 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 division and reciprocal caches is shown in fig. 9 for finite sized caches. For

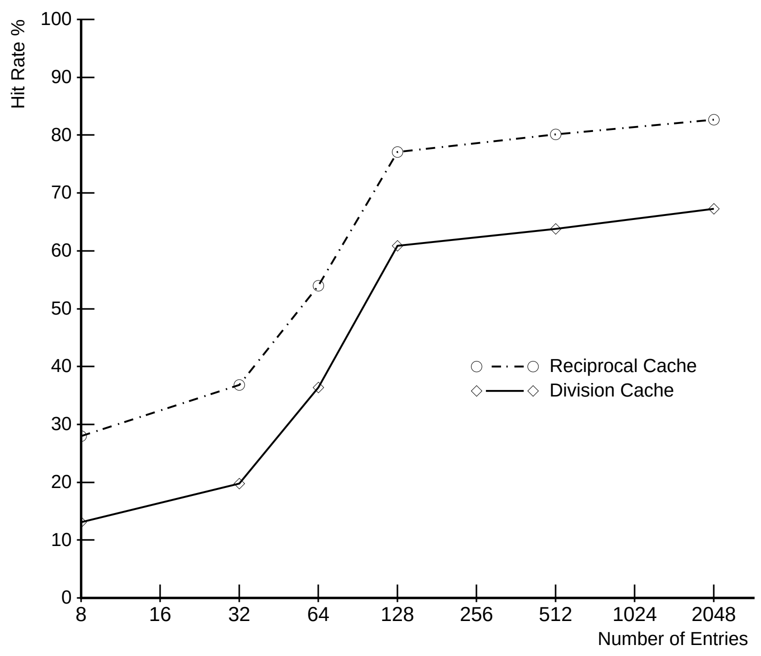

Figure 9: Hit rates for finite division and reciprocal caches

these applications, the reciprocal cache hit rates are consistently larger and less variable than the division cache hit rates. 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 about a two-times speedup. Furthermore, the variance of this speedup across different applications is low.

6.3 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 [56]. 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, at least one iteration is needed 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 offsets the increase in the number of iterations required.

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 30% faster in delay per bit 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, retiring a maximum of 4 bits per cycle, using the same radix-8 datapath was about 10% faster than a conventional radix-8 divider, with an area reduction of 25%.

7 Comparison

Five classes of division algorithms have been presented. In SRT division, 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, by simple staging or possibly overlapping one or both of the quotient selection logic and partial remainder 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, 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, SRT 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.

Very high radix dividers are used when it is desired to retire more than 10 bits per cycle. The primary difference between the presented 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.

Self-timing, result caches, bit-skipping, 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 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.

Both the Newton-Raphson and series expansion iterations are effective means of implementing faster division. 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

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	1 multiplication[1]	$(\lceil \log_2 \frac{n}{i} \rceil + 1)t_{mul} + 2$, $t_{mul} > 1$ $2\lceil \log_2 \frac{n}{i} \rceil + 3$, $t_{mul} = 1$	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

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.

A summary of the algorithms from these classes is shown in table 1. In this table, n is the number of bits in the input operands, i is the number of bits of accuracy from an initial approximation, and t_{mul} is the latency of the fused multiply/accumulate unit in cycles. None of the latencies include additional time required for rounding or normalization.

Table 2 provides a rough evaluation of the effects of algorithm, operand length, width of initial approximation, and multiplier latency on division latency. All operands are IEEE double precision mantissas, with $n = 53$. It is assumed that 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/accumulate latencies of 1, 2 and 3 cycles. Additionally, latencies are

[1]When a pipelined multiplier is used, the delay per iteration is t_{mul}, but one cycle is required at the end of the iterations to drain the pipeline.

Algorithm	Radix	Latency (cycles)
SRT	4	27
	8	18
	16	14
	256	7

Algorithm	Pipelined	Initial Approx	Latency (cycles)		
			$t_{mul} = 1$	$t_{mul} = 2$	$t_{mul} = 3$
Newton-Raphson	either	$i = 8$	8	15	22
	either	$i = 16$	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	either	$i = 8$ (basic)	8	16	24
	either	$i = 16$ (basic)	5	10	15
	either	$i = 13$ and $t = 2$ (adv)	3	6	9
short reciprocal	either	$i = 8$	15	29	
	either	$i = 16$	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

shown for pipelined as well as unpipelined multipliers. A pipelined unit can begin a new computation every cycle, while an unpipelined unit can only begin after the previous computation has completed.

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 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.

8 Conclusion

In this paper, the five major classes of division algorithms have been presented. The classes are determined by the differences in the fundamental operations used in the hardware implementations of the algorithms. 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 processor designers continue to seek 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. Latency is reduced in this implementation through the use of a reordering of the operations in the Newton-Raphson iteration in order to exploit single-cycle throughput of pipelined multipliers. In contrast, the Newton-Raphson iteration itself, with its serial multiplications, has a higher latency. However, if a pipelined multiplier is used throughout the iterations, more than one division operation can proceed in parallel. For implementations with high division throughput requirements, the Newton-Raphson iteration provides a means for trading latency for throughput.

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, although this effect is minimal in many applications. 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 for highly parallel division/multiplication applications, such as graphics and 3D rendering applications. 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 and the DEC Alpha 21164 divider are the only commercial implementations that generate quotients with variable latency depending upon the input operands. 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 continues to increase as die area increases and feature sizes decrease. The correspondingly larger amount of area available for floating point units allows for implementations of higher radix, lower latency algorithms.

9 Acknowledgments

The authors would like to thank Nhon Quach and Grant McFarland for their helpful discussions and comments. The authors also thank the anonymous reviewers for their valuable comments and suggestions. This research was supported by the National Science Foundation under grant MIP93-13701.

References

[1] SPEC benchmark suite release 2/92.

[2] *Microprocessor Report*, various issues, 1994-96.

[3] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *To appear in IEEE Trans. Computers*, 1997.

[4] C. V. Freiman, "Statistical analysis of certain binary division algorithms," *IRE Proc.*, vol. 49, pp. 91–103, 1961.

[5] J. E. Robertson, "A new class of digital division methods," *IRE Trans. Electronic Computers*, vol. EC-7, pp. 218–222, Sept. 1958.

[6] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quart. J. Mech. Appl. Math.*, vol. 11, pt. 3, pp. 364–384, 1958.

[7] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainders," *IEEE Trans. Computers*, vol. C-17, no. 10, Oct. 1968.

[8] K. G. Tan, "The theory and implementation of high-radix division," in *Proc. 4th IEEE Symp. Computer Arithmetic*, pp. 154–163, June 1978.

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

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

[11] P. Soderquist and M. Leeser, "An area/performance comparison of subtractive and multiplicative divide/square root implementations," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 132–139, July 1995.

[12] "IEEE standard for binary floating point arithmetic," ANSI/IEEE Std 754-1985, New York, The Institute of Electrical and Electronic Engineers, Inc., 1985.

[13] S. Oberman and M. Flynn, "Measuring the complexity of SRT tables," Tech. Rep. CSL-TR-95-679, Comput. Syst. Lab., Stanford Univ., Stanford, CA, Nov. 1995.

[14] M. D. Ercegovac and T. Lang, "Simple radix-4 division with operands scaling," *IEEE Trans. Computers*, vol. C-39, no. 9, pp. 1204–1207, Sept. 1990.

[15] J. Fandrianto, "Algorithm for high-speed shared radix 8 division and radix 8 square root," in *Proc. 9th IEEE Symp. Computer Arithmetic*, pp. 68–75, July 1989.

[16] S. E. McQuillan, J. V. McCanny, and R. Hamill, "New algorithms and VLSI architectures for SRT division and square root," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 80–86, July 1993.

[17] P. Montuschi and L. Ciminiera, "Reducing iteration time when result digit is zero for radix 2 SRT division and square root with redundant remainders," *IEEE Trans. Computers*, vol. 42, no. 2, pp. 239–246, Feb. 1993.

[18] P. Montuschi and L. Ciminiera, "Over-redundant digit sets and the design of digit-by-digit division units," *IEEE Trans. Computers*, vol. 43, no. 3, pp. 269–277, Mar. 1994.

[19] P. Montuschi and L. Ciminiera, "Radix-8 division with over-redundant digit set," *J. VLSI Signal Processing*, vol. 7, no. 3, pp. 259–270, May 1994.

[20] S. Oberman, N. Quach, and M. Flynn, "The design and implementation of a high-performance floating-point divider," Tech. Rep. CSL-TR-94-599, Comput. Syst. Lab., Stanford Univ., Stanford, CA, Jan. 1994.

[21] N. Quach and M. Flynn, "A radix-64 floating-point divider," Tech. Rep. CSL-TR-92-529, Comput. Syst. Lab., Stanford Univ., Stanford, CA, June 1992.

[22] H. Srinivas and K. Parhi, "A fast radix-4 division algorithm and its architecture," *IEEE Trans. Computers*, vol. 44, no. 6, pp. 826–831, June 1995.

[23] G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," in *Proc. 7th IEEE Symp. Computer Arithmetic*, pp. 64–71, June 1985.

[24] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, Nov. 1991.

[25] T. Asprey, G. S. Averill, E. DeLano, R. Mason, B. Weiner, and J. Yetter, "Performance features of the PA7100 microprocessor," *IEEE Micro*, vol. 13, no. 3, pp. 22–35, June 1993.

[26] D. Hunt, "Advanced performance features of the 64-bit PA-8000," in *Digest of Papers COMPCON '95*, pp. 123–128, Mar. 1995.

[27] T. Lynch, S. McIntyre, K. Tseng, S. Shaw, and T. Hurson, "High speed divider with square root capability," U.S. Patent No. 5,128,891, 1992.

[28] J. A. Prabhu and G. B. Zyner, "167 MHz Radix-8 floating point divide and square root using overlapped radix-2 stages," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 155–162, July 1995.

[29] A. Svoboda, "An algorithm for division," *Info. Process. Machines*, vol. 9, pp. 29–34, 1963.

[30] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations," *IEEE Trans. Computers*, vol. C-36, no. 7, pp. 895–897, July 1987.

[31] M. D. Ercegovac and T. Lang, "On-the-fly rounding," *IEEE Trans. Computers*, vol. 41, no. 12, pp. 1497–1503, Dec. 1992.

[32] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers, "The IBM System/360 Model 91: Floating-point execution unit," *IBM J. Res. Develop.*, vol. 11, pp. 34–53, Jan. 1967.

[33] D. L. Fowler and J. E. Smith., "An accurate, high speed implementation of division by reciprocal approximation," in *Proc. 9th IEEE Symp. Computer Arithmetic*, pp. 60–67, Sept. 1989.

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

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

[36] P. W. Markstein, "Computation of elementary function on the IBM RISC System/6000 processor," *IBM J. Res. Develop.*, pp. 111–119, Jan. 1990.

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

[38] E. Schwarz, "Rounding for quadratically converging algorithms for division and square root," in *Proc. 29th Asilomar Conf. Signals, Systems, and Computers*, pp. 600–603, Oct. 1995.

[39] S. F. Oberman and M. J. Flynn, "Fast IEEE rounding for division by functional iteration," Tech. Rep. CSL-TR-96-700, Comput. Syst. Lab., Stanford Univ., Stanford, CA, July 1996.

[40] H. Kabuo, T. Taniguchi, A. Miyoshi, H. Yamashita, M. Urano, H. Edamatsu, and S. Kuninobu, "Accurate rounding scheme for the Newton-Raphson method using redundant binary representation," *IEEE Trans. Computers*, vol. 43, no. 1, pp. 43–51, Jan. 1994.

[41] D. Wong and M. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 981–995, Aug. 1992.

[42] W. S. Briggs and D. W. Matula, "A 17x69 Bit multiply and add unit with redundant binary feedback and single cycle latency," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 163–170, July 1993.

[43] D. Matula, "Highly parallel divide and square root algorithms for a new generation floating point processor," extended abstract presented at SCAN-89 Symp. Comput. Arithmetic and Self-Validating Numerical Methods, Oct. 1989.

[44] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very high radix division with selection by rounding and prescaling," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 909–918, Aug. 1994.

[45] D. DasSarma and D. Matula, "Measuring the accuracy of ROM reciprocal tables," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 932–940, Aug. 1994.

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

[47] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 2–9, July 1995.

[48] M. J. Schulte, J. Omar, and E. E. Swartlander, "Optimal initial approximations for the Newton-Raphson division algorithm," *Computing*, vol. 53, pp. 233–242, 1994.

[49] E. Schwarz, "High-radix algorithms for high-order arithmetic operations," Tech. Rep. CSL-TR-93-559, Comput. Syst. Lab., Stanford Univ., Stanford, Ca, Jan. 1993.

[50] E. Schwarz and M. Flynn, "Hardware starting approximation for the square root operation," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 103-111, July 1993.

[51] P. Bannon and J. Keller, "Internal architecture of Alpha 21164 microprocessor," in *Digest of Papers COMPCON '95*, pp. 79–87, Mar. 1995.

[52] T. Williams, N. Patkar, and G. Shen, "SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor," *IEEE J. Solid-State Circuits*, vol. 30, no. 11, pp. 1215–1226, Nov. 1995.

[53] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 220–227, July 1993.

[54] S. Oberman and M. Flynn, "Reducing division latency with reciprocal caches," *Reliable Computing*, vol. 2, no. 2, Apr. 1996.

[55] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories and its application," *IEEE J. Solid-State Circuits*, vol. 26, no. 2, Feb. 1991.

[56] J. Cortadella and T. Lang, "High-radix division and square root with speculation," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 919–931, Aug. 1994.