A New Iterative Structure for Hardware Division: the Parallel Paths Algorithm

Eric Rice Richard Hughey

Department of Computer Engineering
University of California, Santa Cruz, CA 95064
E-mail: elrice,rph@soe.ucsc.edu

Abstract

This paper presents a new approach to hardware division—the parallel paths algorithm. In this approach, prescaling allows the division recurrence to be implemented by three processes which can be calculated in parallel during iterations. While two of the processes must complete in a single iteration, the third—which includes the most expensive division operations—can be calculated over multiple iterations. Iteration latency is determined by the slowest of the three paths, and in many cases can be limited to that of carry-save addition and latching. A radix-4 implementation of the algorithm is shown to achieve better performance than other commonly used methods while requiring a modest increase in area.

Index Terms: Computer arithmetic, hardware division, prescaling, linear convergence.

1 Introduction

Digit-recurrence algorithms used in special-purpose division hardware involve several steps, the most problematic of which is quotient selection. In basic SRT division [1, 2], quotient selection represents half or more of iteration latency [3]. As a result, considerable effort has been made to reduce its impact on division latency.

Two strategies are of particular interest to speed quotient selection. The first is to overlap multiple SRT stages in a single iteration, making use of multiple speculative calculations that can be selected among quickly when a key result becomes available. This overlap can involve quotient selection (used in the Ultra Sparc64 processor [4]), partial remainder formation (used in the Hal Sparc64 processor [5]), or both [6]. These are currently the best approaches for accelerating division without significant increase in area.

A second strategy to reduce the impact of quotient selection is to prescale the divisor close to unity. When prescaled with sufficient precision, this allows quotient selection to be performed by simply rounding or truncating the partial remainder [7]. While prescaling has been proposed for low-radix algorithms, the overhead associated with prescaling prevents them from being as efficient as overlapped SRT. High-radix prescaling algorithms proposed by Ercegovac et. al. [8] and by Wong and Flynn [9] lead to faster division than overlapped SRT but require significant increases in area.

A prescaling algorithm proposed by Ercegovac and Lang [10] uses prescaling in a different way. Although it does not quite achieve sufficient compensation for the prescaling overhead, it is of interest because of its similarity to our proposed algorithm in that (1) prescaling is performed to greater precision than needed by the iteration radix, and (2) this added precision is used to allow quotient selection to be calculated over multiple iterations (this is fixed at two iterations in the cited paper).

This paper presents a new strategy that can eliminate quotient selection from the critical path altogether. In this approach, prescaling allows the division recurrence to be implemented by three separate processes which can be calculated in parallel during iterations. Iteration latency is determined by the slowest of the three paths, and in many cases can be limited to that of carry-save addition and latching. We call this approach the parallel paths algorithm.

This paper is organized as follows. After introducing basic terms and concepts in Section 2, Section 3 describes the Parallel Paths algorithm. A radix-4 implementation is described in Section 4, followed by a comparison of the implementation with other efficient algorithms in Section 5.

2 Division Basics and Notation

We address the problem of calculating $Q{=}a/b$, where the standard recurrence for an iterative division algorithm is:

$$P_{i+1} = P_i - bq_i. \tag{1}$$

for partial remainder P_i (where $P_0{=}a$) and partial quotient q_i. Calculating equation (1) involves several steps:

- SEL: Select next quotient digit q_i.
- DRV: Drive selected digit (encoded) to array of full adders that will be used for partial remainder update.

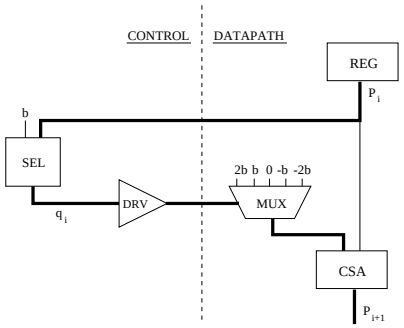

Figure 1. Basic radix-4 SRT algorithm.

- MUX: Use driven signals to multiplex appropriate (precalculated) bq_i.
- CSA: Perform partial remainder update via array of carry-save adders.
- REG: Latch new partial remainder.

Figures 1 and 2 illustrate how these basic blocks are used in a basic radix-4 SRT algorithm and a radix-4 SRT algorithm using overlapped radix-2 stages. In both cases, SEL is performed via table lookup based on b and the most significant bits of P_i after a short carry-propagate addition (CPA).

2.1 Prescaling algorithms

In a prescaling algorithm, a scaling factor M—an estimate of $1/b$—is used to scale a and b. The division problem then becomes:

$$Q = \frac{a}{b} = \frac{Ma}{Mb}.$$

and the recurrence becomes:

$$\begin{aligned} P_{i+1} &= P_i - q_i(Mb) && (2)\\ &= (P_i - q_i) + q_i(1 - Mb), && (3) \end{aligned}$$

where $P_0 = Ma$.

The reformulation in equation (3) shows the relationship between the precision of the scaling factor and the number of quotient bits that can be retired in each iteration by rounding or truncating. Since $(P_i - q_i)$ can have an arbitrarily large number of leading zeros (we could use a very over-redundant quotient digit set and select $q_i = P_i$), the magnitude of $q_i(1-Mb)$ determines the maximum number of bits that can be retired.

3 The Parallel Paths Algorithm

The parallel paths algorithm is based on equation (3), but aims at minimizing iteration latency rather than maximizing the number of bits that can be retired per iteration. The strategy is developed by studying the two terms of equation (3) and noticing two important differences between them.

The first difference is their complexity. While $(P_i - q_i)$ can be calculated quickly for some quotient selection methods (described below), adding $q_i(1-Mb)$ requires the combined latency of SEL→DRV→MUX→CSA, the entire standard digit-recurrence iteration.

The second difference between the terms of equation (3) is the relative magnitudes of the two terms. While subtracting q_i in $(P_i - q_i)$ zeroes the leading fractional bits and significantly affects P_{i+1}, the magnitude of $q_i(1-Mb)$ is determined by the accuracy of the scaling factor M.

The new approach uses this last fact to address the high latency of calculating q_i and adding $q_i(1-Mb)$ to the partial remainder. By prescaling with sufficient precision, $q_i(1-Mb)$ can be made small enough so that it will not affect quotient selection in one or more subsequent iteration(s). This allows postponing its introduction, spreading its latency over multiple iterations.

While this leads to partial remainders that are not fully adjusted with respect to previous quotient digits, it is easy to see the validity of the process:

$$\begin{aligned} Q &= \sum_{j=1}^{i} q_j + \frac{P_{i+1}}{Mb} \\ &= \sum_{j=1}^{i} q_j + \frac{P_i - q_i(Mb)}{Mb} \\ &= \sum_{j=1}^{i} q_j + \frac{(P_i - q_i)}{Mb} + \frac{q_i(1 - Mb)}{Mb}. \end{aligned}$$

Since $q_i(1 - Mb)$ will eventually be added to the partial remainder where its value will be divided by Mb, it does not matter that it has been postponed.

An iteration of the new approach involves three parallel processes. This is because addition of $q_i(1-Mb)$ is divided into two parts. While SEL→DRV→MUX can be distributed over multiple iterations, its subsequent addition to the partial remainder (CSA) must occur within a single iteration since each iteration will introduce new partial product(s) to the partial remainder. The three processes are thus:

- **SUB** Calculation of $(P_i - q_i)$. (Despite its name and function, an appropriate SUB will generally not involve subtraction, as discussed below.)
- **SEL→DRV→MUX** Preparation of the partial products for partial remainder update. This requires selection of q_i, driving such to the CSA array, and multiplexing between $q_i(1-Mb)$ values.
- **CSA** Partial remainder update through carry-save addition.

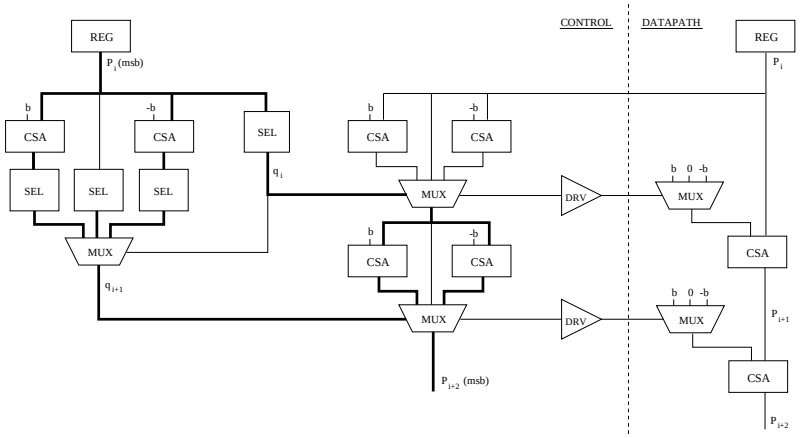

Figure 2. Radix-4 SRT using overlapped radix-2 stages (with overlapped quotient selection and overlapped partial remainder formation).

Both SUB and CSA update the partial remainder in each iteration and are performed simultaneously. SUB (based on q_i) adjusts the high-order bits of P_i, and CSA (based on q_{i-k} for k postponed stages) adjusts the lower-order bits. Since the latency of SEL→DRV→MUX can be 'hidden' by sufficiently postponing the introduction of partial products, iteration latency is often limited by the maximum of the SUB and CSA latencies.

Surprisingly, SUB is not the limiting factor when the right SEL is used. For example, by calculating quotient digits q_i via carry propagate addition (CPA) of the most significant bits of P_i followed by truncation, $(P_i - q_i)$ is simply the fractional result of the CPA plus the bits of P_i not included in the CPA (Figure 3). This SEL method thus allows $(P_i - q_i)$ to be calculated without first calculating q_i, and requires less time than a CSA.

Figure 4 illustrates how the parallel paths strategy can reduce iteration latency through increased prescaling precision and postponement of partial products. For every two additional bits of prescaling precision, each partial product can be postponed one additional iteration, since its magnitude relative to the 'local' partial remainder will be the same (ensuring the same bounds). The figure shows how this affects iteration latency for assumed latencies of 2 t_{xor} for CSA, 6 t_{xor} for SEL→DRV→MUX, and a SUB requiring 2 t_{xor} or less. Four possible places for adding the partial products $q_i(1-Mb)$ are shown, along with corresponding prescaling requirements and iteration latencies. Note that the 2nd, 3rd and 4th latencies are determined by the latency of SEL→DRV→MUX divided by the number of iterations the partial products are postponed, but that latency cannot be reduced further due to CSA latency. (Also, note that the latency estimates given are for illustrative purposes only, as are the two leading zeros in partial products.)

Designing a good implementation of the parallel paths algorithm is a complicated optimization problem. In addition to many of the same parameter choices available in previous digit-recurrence algorithms, there is the added structural choice of how many stages to postpone introduction of partial products. The three-way race inherent in iterations leads to different priorities for evaluating a given design, including the requirement for a fast SUB and increased importance of CSA latency. There is also a tradeoff between prescaling costs and iteration costs, since extra prescaling precision often leads to reduced iteration latency.

While an important purpose of this paper is to introduce the parallel paths strategy (rather than a particular implementation) our exploration of this design space has led to several promising designs, including the following radix-4 algorithm that provides improved performance over the best currently used algorithms without significant increase in area.

4 Parallel Paths Implementation: a Radix-4 Example

To develop a radix-4 parallel paths algorithm based on $q_i \in \{-2, \ldots, 2\}$, we begin with a simpler related algorithm based on $q_i \in \{0, \ldots, 4\}$, and then convert it to the preferred digit set (preferred primarily because allowing $q_i = 3$ requires calculating a non-redundant $3(1-Mb)$ for efficient partial product formation).

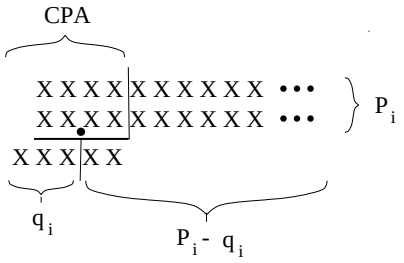

Figure 3. Implementing SEL via a CPA of the most significant bits of P_i allows fast calculation of $P_i - q_i$. (Each 'X' represents one bit.)

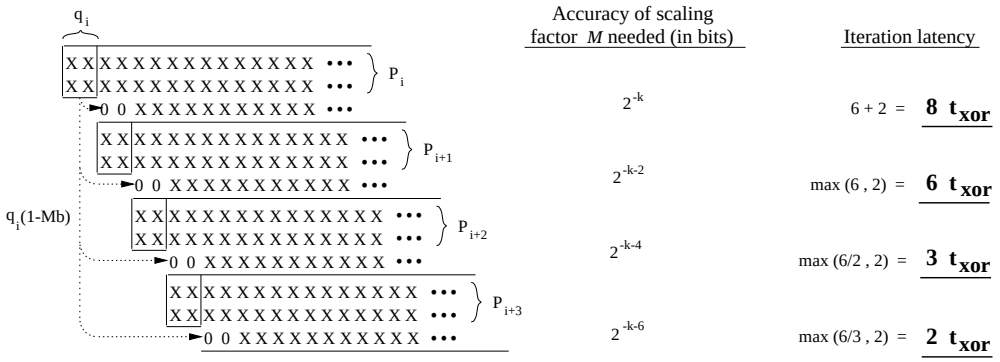

Figure 4. Reducing iteration latency in a radix-4 algorithm with the parallel paths algorithm.

4.1 Algorithm based on $q_i \in \{0, \ldots, 4\}$

The basic recurrence of the $q_i \in \{0, \ldots, 4\}$ algorithm is shown in Figure 5(a). The SEL and SUB functions are implemented by a CPA of P_i that incorporates two fractional bits (shaded region), with the integer result becoming q_i (implementing SEL). The fractional result (digits A and B) plus the bits of P_i not incorporated in the CPA, becomes $P_i - q_i$ (implementing SUB). P_{i+1} is formed through CSA addition of the less significant bits of P_i and $q_{i-2}(1-Mb)$, with digits A and B being forwarded directly to P_{i+1}.

Algorithm bounds are ensured by requiring that each partial product have four leading fractional zeros relative to the partial remainder to which it is added (Figure 5(b)). This ensures that the bits of P_{i+1} which determine q_{i+1} (shaded region) have a maximum value of 4.75, so that $q_{i+1} \leq 4$. Note that for the parallel paths algorithm one must assume that a worst-case partial product will be added to each P_i when checking bounds, since it is not based on q_i.

A bit-wise analysis of the CPA as done in Figure 5(b) is necessary. For example, one could instead calculate the maximum allowable P_i:

$$\begin{aligned} (P_{max} - q_{max} + [q(1-Mb)]_{max}) \times radix &= P_{max} \\ (P_{max} - 4 + 1/16) \times 4 &= P_{max} \\ P_{max} &= 5.25, \end{aligned}$$

but while this is in fact the value of P_{i+1} in Figure 5(b), a smaller P_i could cause CPA overflow. For example:

1	1	1	1	0	0	0	0	...
	1	.0	1	0	0	0	0	...

would lead to the q_i=5. While it would be possible to accomodate this by selecting q_i=4 and leaving the remaining '1' for P_{i+1}, such would require a more complex (slow) SEL. Fortunately, as Figure 5(b) shows, the present implementation avoids partial remainders that would cause such CPA overflow.

For perspective, there are many ways to design a CPA-based radix-4 algorithm. For example, if SEL were implemented using a CPA that incorporated 3 fractional bits of P_i (instead of 2) just three leading fractional zeros would be needed in partial products (Figure 6(a)). In fact, just two leading zeros would be needed if the 3rd fractional bit of the partial product were also incorporated in the CPA (Fig-

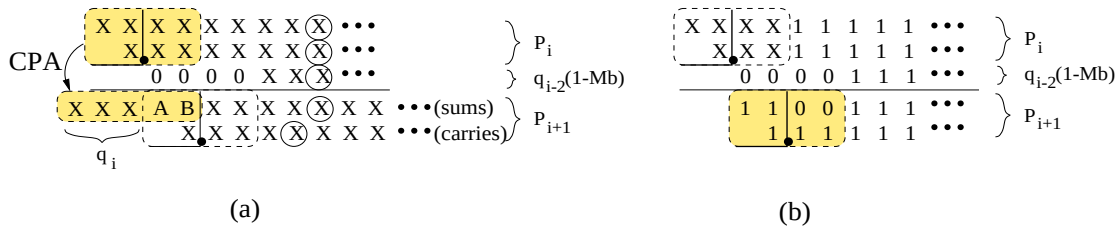

Figure 5. (a) Recurrence for $q_i \in \{0, \ldots, 4\}$ algorithm. (The circled X's show the inputs and outputs of one full adder.) (b) Demonstrating algorithm bounds.

Figure 6. Two other CPA-based radix-4 schemes. Different forms of CPA (shaded regions) require different number of leading fractional zeros in partial products to ensure $q_{i+1} \leq 4$.

ure 6(b)). Note that the CPA could not also incorporate the 2nd fractional bit of the partial product (i.e., so that partial products would require just 1 leading fractional zero) because then the CPA could produce two carries, leading to q_i=5.

While each of the SEL methods in Figure 6 would require increased latency, one could compensate for this by postponing partial products a third iteration. Using the SEL shown in Figure 6(b) would then require the same prescaling accuracy as in the proposed implementation, but SEL→DRV→MUX would have the latency of an additional CSA to complete.

Iteration latency of the proposed algorithm is designed to be based on CSA latency. By postponing introduction of partial products two iterations, SEL→DRV→MUX has at least 2 t_{FA} to finish without affecting iteration latency, and 2.5 t_{FA} if the CSA is designed to allow one input to arrive late (which we will assume).

4.2 Converting the algorithm to $q_i \in \{-2, \ldots, 2\}$

Converting the algorithm to $q_i \in \{-2, \ldots, 2\}$ can be accomplished by inserting a constant '1' in the fifth fractional position of each partial product and later subtracting it when it 'migrates' to an integer position (Figure 7). Because the constant represents a '2' when it migrates out in a partial quotient, subtracting it neatly converts $q_i \in \{0, \ldots, 4\}$ to $q_i \in \{-2, \ldots, 2\}$ [1]. (Note that while the figure indicates that there is a '1' in each constant position, this digit becomes '0' when the corresponding partial product is negative.)

Adding a constant to each $q_i(1-Mb)$—rather for example than adding all of them to P_0—is necessary to ensure that partial remainders remain positive, since we need $q_i \in \{0, \ldots, 4\}$ before subtracting 2. Thus we cannot 'save' one bit of prescaling precision by adding the constants to P_0. (For cases as in Figure 6(a) where the number of leading fractional zeros is odd—i.e. where the first non-zero position will represent a '1' when it migrates to an integer position—we can minimize prescaling requirements by adding half of each constant to the partial product and half to P_0.)

Prescaling requirements for this algorithm can be easily calculated. Since the 5th fractional digit of each partial product is used for the constant '1', each $q_i(1-Mb)$ requires 5 leading fractional zeros (if positive) or 5 leading fractional ones (if negative) relative to the partial remainder to which it is added. Since the radix point of this partial remainder has shifted four places from the partial remainder from which q_i is selected, we need $-2^{-(4+5)} < q_i(1-Mb) < 2^{-(4+5)}$, leading to:

$$-2^{-10} < (1 - Mb) < 2^{-10}$$
$$-2^{-10} < b(1/b - M) < 2^{-10}.$$

Since the value of b can approach 2, this seems to imply that we need:

$$-2^{-11} < (1/b - M) < 2^{-11}.$$

[1] As pointed out by Ercegovac, the process of adding and later subtracting these constants implements a sort of rounding—the previous two partial products cause the fractional bits of each partial remainder to be overvalued by 5/8. Without this or some other form of rounding, it would be theoretically impossible to design an algorithm based on truncation without using an overredundant quotient digit set [3].

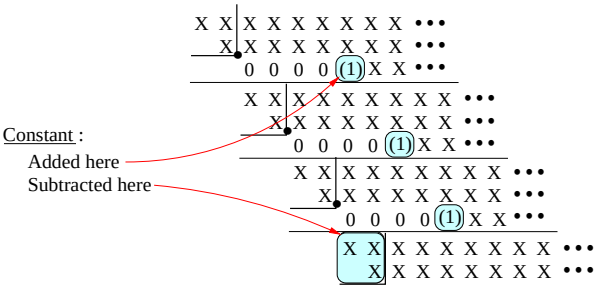

Figure 7. A $q_i \in \{0, \ldots, 4\}$ radix-4 algorithm is converted to a $q_i \in \{-2, \ldots, 2\}$ algorithm by adding a constant to each partial product and later subtracting it from each quotient digit.

However, because most methods for reciprocals produce estimates that are considerably more accurate for $b \approx 2$ than $b \approx 1$ (the curve $y = 1/x$ is flatter at $b \approx 2$), we can use methods aimed at just 2^{-10} reciprocal accuracy.

Based on analysis in [11] (though eventually found empirically) we found a linear approximation that produces an 11-bit redundant reciprocal estimate after a short multiply-accumulate:

$$M = C_0 - C_1(b - p),$$

where p is b truncated after four fractional bits. This method requires two lookup tables for C_0 and C_1 of sizes $(2^4 \times 11)$ and $(2^4 \times 7)$, respectively. In the next section we describe how M can be calculated using iteration hardware.

4.3 Implementation Details

Figure 8 shows the hardware blocks needed for iterations of the proposed implementation, excluding the scaling factor lookup tables and hardware for on-the-fly conversion of partial quotients. Two copies of iteration hardware are used because of the sensitivity of the parallel paths algorithm to latching latency.

There are several subtle features of the implementation. To ensure maximum efficiency, REG C must be latched 1 t_{xor} later than the other registers (its output will be the late input to the CSA).

Primary iteration hardware can be used to calculate M and the prescaled operands $(1 - Mb)$ and Ma. To see how this works, consider a generic multiply-add:

$$xy + z,$$

where x has fewer digits than y. By storing x in the most significant positions of REG B and selecting y in the Q-MUX, each iteration will process two radix-4 digits of x—multiplying each by y and adding the results to the partial remainder. Since iteration hardware left-shifts partial remainders 2 places after each CSA, the various partial products $x_i y$ end up correctly aligned, and the digits of x are slowly shifted out of REG B. (Note that at the beginning we need a right-shifted y in Q-MUX to compensate for these shifts.) By loading a right-shifted z into P_0 at the beginning (adjacent to x), iteration hardware will calculate a redundant $(xy + z)$ in REG B.

Based on this method, the choices of inputs to the two MUXes at the top of Figure 8 allow iteration hardware to calculate:

1) $C_0 - C_1(b - p) = M$
2) $1 - Mb$
3) Ma

before beginning division iterations.

5 Comparison with Other Algorithms

We now compare the radix-4 parallel paths algorithm presented above with several of the best competing algorithms for double precision division. In particular, our comparison includes:

- Basic radix-4 SRT.
- Radix-16 SRT using two overlapped radix-4 stages. (One of the best approaches currently in use.)
- Radix-512 very high radix algorithm (VHR) of Ercegovac et. al. (We do not include higher radices of this approach due to their large area requirements.)

In Table I we evaluate area and latency requirements for these algorithms, relying primarily on the methodology and results of Montuschi et. al. [12], where parameterized area and latency estimates are provided for common hardware blocks. We use their results for basic radix-4 SRT and radix-16 overlapped SRT without modification. For each algorithm, we also evaluate latency when a faster MUX/REG design is used, as proposed in [13]. (Due to lack of data, we assume the same areas when using this design.)

Figure 9 plots speed versus area for these algorithms. As can be seen, the parallel paths algorithm is slightly faster

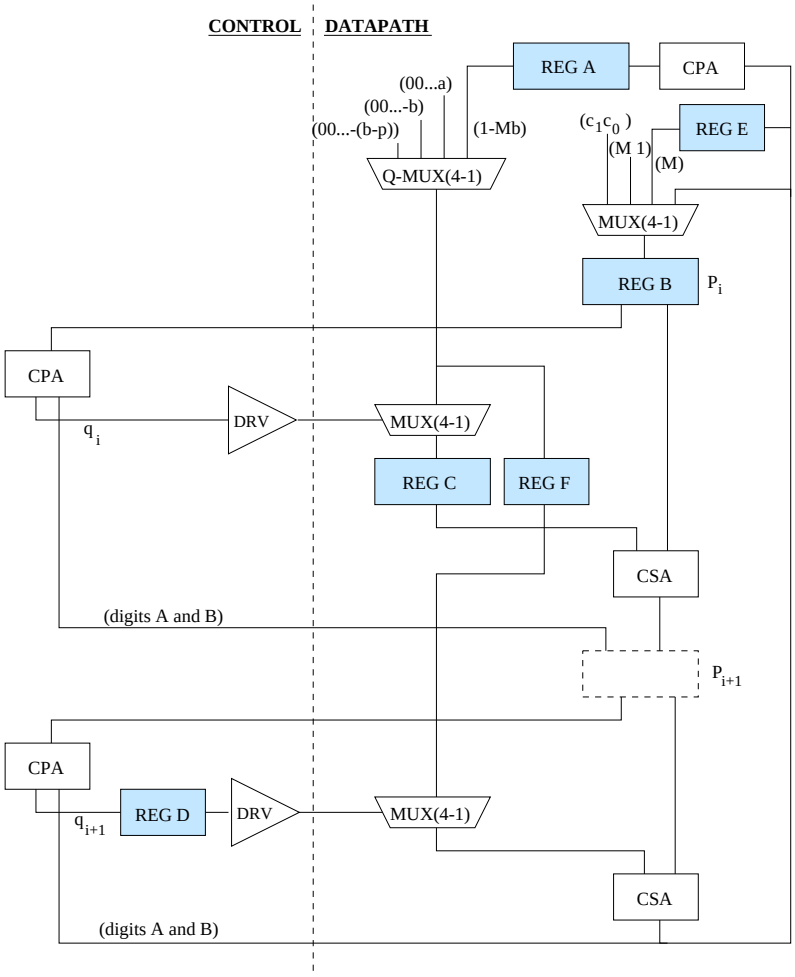

Figure 8. Primary hardware blocks needed for iterations.

Table I. Estimated component latency and area for parallel paths and Radix-512 VHR algorithms.

	Radix-4 Parallel Paths 2 copies	**R-512 VHR**
scaling factor M	50	150
REG A	.6(66)	.6(68)
Q-MUX	.8(72)	.8(68)
REG E	.6(11)	–
MUX (for REG B)	.8(72)	–
REG B	1.2(72)	1.2(68)
SEL	–	–
DRV	–	–
MUX (part. product)	.8(72)(2)	.8(68)(6)
REG C	.6(72)	–
REG F	.6(72)	–
CSA	72(2)	2.3(68)(3)
REG D	–	–
O-T-Fly	1.45(56)	1.45(56)
CPA	2.4(66)	2.4(68)
TOTAL AREA (A_{FA})	**883**	**1367**

	Radix-4 Parallel Paths 2 copies	**R-512 VHR**
scaling factor M	4	1
CSA(1-Mb)	3	1
CPA(1-Mb),CSA(Ma)	3	1
q_i's	15	7
finish	2	1
total cycles	26	11
cycle time	4/2.75	7.5/6.25
TOTAL LATENCY (t_{FA})	**104/71.5**	**82.5/70**

	R-4 SRT	**R-16 Overlapped SRT**
TOTAL AREA (A_{FA})	**300**	**550**
total cycles	28	15
TOTAL LATENCY (t_{FA})	**170/135**	**120/101.25**

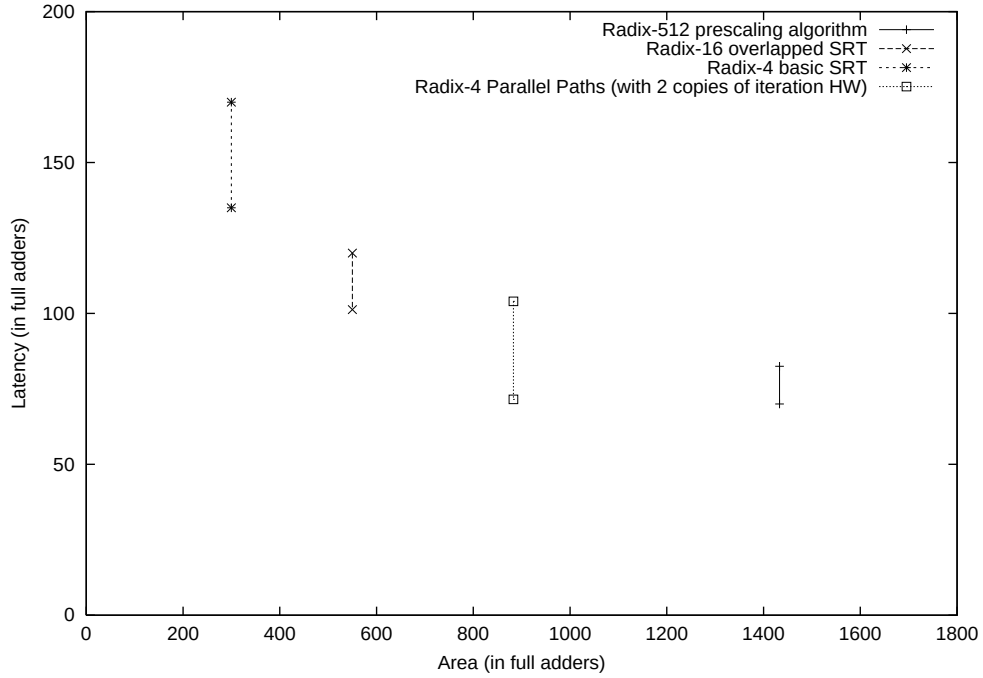

Figure 9. Speed versus area for several of the best division algorithms. The upper latency shown for each algorithm is based on component estimates from [12]; the reduced latencies assume use of a fast MUX/REG as described in [13].

than overlapped SRT when the slower registers are used, and significantly faster when using the faster registers.

5.1 Higher Precision Division

For higher-precision division, the parallel paths approach increases its advantage over the SRT algorithms. This is because, for example, if the division must be carried out to twice as many bits, the SRT algorithms will require fully twice as many iterations to perform the division. The parallel paths algorithms, on the other hand, while requiring twice as many quotient-digit producing iterations (representing 50-60% of its double precision latency) will see very little increase in latency in the other 40+% of the algorithm.

For *very* high-precision division, another variation of the Parallel Paths algorithm can attain speeds approaching 1 bit every $0.25t_{FA}$, though (not unexpectedly) this algorithm requires considerable area.

6 Conclusions/Future Work

This paper introduced the parallel paths algorithm, a new hardware division strategy that allows the latency of quotient selection and partial product formation to be 'hidden' by distributing their calculation over multiple iterations. While still looking for an optimal design—the parallel paths approach is in fact a class of algorithms—we examined a specific radix-4 implementation that can achieve an estimated speedup of 1.4 over radix-16 overlapped SRT while requiring a factor of 1.6 increase in area. These results indicate that the parallel paths strategy offers a new option in terms of area/performance tradeoff. Future work will include more detailed design and simulation, with eventual implementation in silicon.

7 Acknowledgments

This work was supported in part by NSF grant EIA-9905322 and an ARCS foundation scholarship.

References

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

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

[3] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-recurrence algorithms and applications*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1994.

[4] J. A. Prabhu and G. B. Zyner, "167 MHz radix-8 divide and square root using overlapped radix-2 stages," in *Proc. Int. Conf. Computer Design*, pp. 155–162, IEEE, 1995.

[5] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160-ns 54-b CMOS divider," *IEEE Journal of Solid-State Circuits*, vol. 26, pp. 1651–1661, Nov. 1991.

[6] D. Harris and M. A. Horowitz, "SRT division architectures and implementations," in *Proc. Int. Conf. Computer Design*, pp. 18–25, IEEE, 1997.

[7] A. Svoboda, "An algorithm for division," in *Proc. 9th Symp. Inform. Processing Machines*, pp. 25–34, 1963.

[8] M. D. Ercegovac, T. Lang, and P. Montuschi, "Very high radix division with selection by rounding and prescaling," *IEEE Trans. Computers*, vol. 43, pp. 909–918, Aug. 1994.

[9] D. Wong and M. Flynn, "Fast division using accurate quotient approximations to reduce the number of iterations," *IEEE Trans. Computers*, vol. 41, pp. 981–995, Aug. 1992.

[10] M. D. Ercegovac and T. Lang, "A division algorithm with prediction of quotient digits," in *Proc. Int. Conf. Computer Design*, pp. 51–56, IEEE, 1985.

[11] P.-M. Seidel, "High-speed redundant reciprocal approximation," *Integration, the VLSI Journal*, vol. 28, pp. 1–12, 1999.

[12] P. Montuschi and T. Lang, "Boosting very-high radix division with prescaling and selection by rounding," *IEEE Transactions on Computers*, vol. 50, pp. 13–27, Jan. 2001.

[13] F. Klass, C. Amir, A. Das, K. Aingaran, C. Truong, R. Wang, A. Mehta, R. Heald, and G. Yee, "A new family of semidynamic and dynamic flip-flops with embedded logic for high-performance processors," *IEEE Journal of Solid-State Circuits*, vol. 34, pp. 712–716, May 1999.