167 MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages

J. Arjun Prabhu and Gregory B. Zyner

SPARC Technology Business, Sun Microsystems, Inc.
Mountain View, California

Abstract - ***Ultra*SPARC's IEEE-754 compliant floating point divide and square root implementation is presented. Three overlapping stages of SRT radix-2 quotient selection logic enable an effective radix-8 calculation at 167 MHz while only a single radix-2 quotient selection logic delay is seen in the critical path. Speculative partial remainder and quotient calculation in the main datapath also improves cycle time. The quotient selection logic is slightly modified to prevent the formation of a negative partial remainder for exact results. This saves latency and hardware as the partial remainder no longer needs to be restored before calculating the sticky bit for rounding.**

I. Introduction

The SRT algorithm provides a means of performing non-restoring division [1], [2]. More bits of quotient are developed per iteration with higher radices, but at a cost of greater complexity. A simple SRT radix-2 floating point implementation (Fig. 1) requires that the divisor and dividend both be positive and normalized, $1/2 \le D, Dividend < 1$. The initial shifted partial remainder, *2PR[0]*, is the dividend. Future partial remainders are developed according to the following equation,

$$PR_{i+1} = 2PR_i - q_{i+1}D. \qquad (1)$$

where q is the quotient digit {-1, 0, or +1} which is solely determined by the value of the previous partial remainder and is independent of the divisor, an attractive feature for square root. Discussion of the quotient digit selection function will be deferred until the next section. The partial remainder is often kept in redundant form so that carry-save adders can be used instead of slower and larger carry-propagate adders. The partial remainder is converted to non-redundant form after the desired precision is reached. The quotient digits can also be kept in redundant form and converted to non-redundant form at the end, or the quotient and quotient minus one (Q and $Q-1$) can be generated on the fly according to rules developed by Ercegovac [3].

The SRT algorithm can also be used for square root allowing utilization of existing division hardware. The simplified square root equation looks surprisingly similar to that of division [4]:

$$PR_{i+1} = 2PR_i - q_{i+1}(2Q_i + q_{i+1}2^{-(i+1)}). \qquad (2)$$

The terms in parentheses are the effective divisor. For square root, the so-called divisor is a function of the previous quotient bits (root bits to be more precise) [4], hence on-the-fly quotient generation is required.

Quotient selection logic (qslc) for a radix-2 SRT implementation will be discussed in Sections II and III with emphasis on how it can be modified to better constrain the partial remainder in the case of exact results. Preventing the partial remainder from unnecessarily going negative for exact results leads to a one cycle savings in generating the sticky bit.

It will be demonstrated in Section IV that overlapping radix-2 quotient selection logic stages can provide an effective timing solution for generating multiple bits of quotient per cycle. The timing benefits of speculative datapath calculations of the partial remainder, quotient, and next divisor occurring in conjunction with quotient digit selection will be shown. A comparison of overlapped radix-2 versus radix-4 implementations will also be discussed.

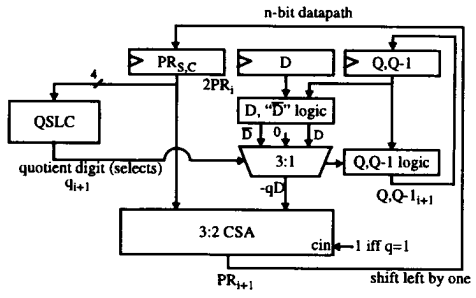


Fig. 1. Simple SRT Radix-2 Divide, Square Root Implementation

II. Quotient Selection Logic

The logic which generates quotient selection digits is the central element of an SRT division implementation.

Early research indicated that only the upper three bits of the redundant partial remainder are necessary inputs for a radix-2 quotient digit selection function [5], [6]. However, more recent studies have shown that four bits are required to correctly generate quotient digit selection digits and keep the partial remainder within prescribed bounds [7], [8], [9], [10]. The selection rules can be expressed as:

$$q_{i+1} = \begin{cases} 1, \text{ if } 0 \le 2PR_i \le 3/2 \\ 0, \text{ if } 2PR_i = -1/2 \\ -1, \text{ if } -5/2 \le 2PR_i \le -1 \end{cases} \quad (3)$$

The quotient selection logic is designed to guess correctly or overestimate the true quotient result. e.g. predicting 1 instead of 0, or 0 instead of -1. The SRT algorithm corrects itself later if the wrong quotient digit has been chosen.

TABLE I
TRUTH TABLE FOR RADIX-2 QUOTIENT SELECTION LOGIC

$2PR_{i,estimated}$	quotient $digit_{i+1}$	comments
100.0	don't care	2PR never < -5/2
100.1	don't care	2PR never < -5/2
101.0	-1*	2PR never < -5/2, but 2PR could be 101.1 when $2PR_{est}$ is 101.0
101.1	-1	
110.0	-1	
110.1	-1	
111.0	-1	2PR could = 111.1
111.1	0	2PR could = 000.0
000.0	+1	
000.1	+1	
001.0	+1	
001.1	+1	
010.0	don't care	2PR never > 3/2
010.1	don't care	2PR never > 3/2
011.0	don't care	2PR never > 3/2
011.1	don't care	2PR never > 3/2

The truth table for SRT radix-2 quotient selection logic has several don't care inputs because the partial remainder is constrained $-5/2 \le 2PR_i \le 3/2$. The estimated partial remainder is always less than or equal to the true partial remainder because the lower bits are ignored. There is a single case, $2PR_{est} = 101.0$, where the estimated partial remainder can appear to be out of bounds. By construction, the real partial remainder is within the negative bound, so -1 is the appropriate quotient digit to select. There are two other cases where the quotient digit selected based on the estimated partial remainder differs from what would be chosen based on the real partial remainder. However, in both of these instances of "incorrect" quotient digit selection, the quotient is not underestimated and the partial remainder is kept within prescribed bounds, so the final result will still be generated correctly.

For increased testability, most designs today are scannable. All registers are stitched together in a chain to allow a special test mode known as scan. During scan, external values can be optionally be shifted into these registers, the system can be clocked for one or more cycles, and the new register values can be shifted out and observed. These capabilities aid in functional and timing debug.

While loading the scan chain, the partial remainder flip flops can take on any value. Logic simplification for don't care cases should ensure that a unique quotient digit is always selected (i.e. the quotient digits selects are 1-hot) for all input combinations to prevent contention of multiplexer selects. The simplified truth table follows.

TABLE II
SIMPLIFIED QSLC TRUTH TABLE

$2PR_{i,estimated}$	quotient $digit_{i+1}$
0xx.x	+1
111.1	0
1xx.x	-1

III. STICKY BIT CALCULATION

Floating point operations generate a sticky bit along with the result in order to indicate whether the result is inexact. The sticky bit is also used with the guard and round bits for rounding according to IEEE Standard 754 [11]. For divide and square root operations, the sticky bit is determined by checking if the final partial remainder is non-zero. The final partial remainder is defined as the partial remainder after the desired number of quotients bits have been calculated. Since the partial remainder is in redundant form, a carry-propagate addition is performed prior to zero-detection (See Fig. 3a).

A. Exact Results

At first glance, the above solution seems perfectly reasonable for all final partial remainder possibilities, positive or negative. However, in the rare case where the result is exact, the final partial remainder will be equal to the negative divisor. For example, consider a number divided by itself (Fig.2). Since the dividend is positive and normalized, the quotient digit from the first iteration is one. For the next iteration, the partial remainder is zero which causes the second quotient digit to be one. For all subsequent iterations, the partial remainder will equal the negative divisor and quotient digits of minus one will be selected. After the last iteration, performing a sign detect on the final partial remainder indicates that *Q-1* should be chosen which is in fact the correct result. However, this same final partial remainder is non-zero which erroneously suggests an inexact result.

```
PR[0] = init dividend/2 =  D/2    q[1]   =+1    Q=1          Q-1=0
PR[1] = 2(D/2) - ( 1)D  =  0      q[2]   =+1    Q=11         Q-1=10
PR[2] = 2(0)   - ( 1)D  = - D     q[3]   =- 1   Q=101        Q-1=100
PR[3] = 2(-D)  - (-1)D  = - D     q[4]   =- 1   Q=1001       Q-1=1000
PR[n] = 2(-D)  - (-1)D  = - D     q[n+1] =- 1   Q=100..001   Q-1=100..000
```

Fig. 2. Divide iterations for a number divided by itself.

This problem extends to any division operation for which the result should be exact because the quotient selection logic is defined to guess positive for a zero partial remainder and correct for it later. The simplest solution is to restore negative final partial remainders by adding the divisor before performing zero-detection. Given the area expense of an additional carry-propagate adder, the solution should try to take advantage of existing hardware. Two ways to achieve this are shown in Figs. 3b and 3c.

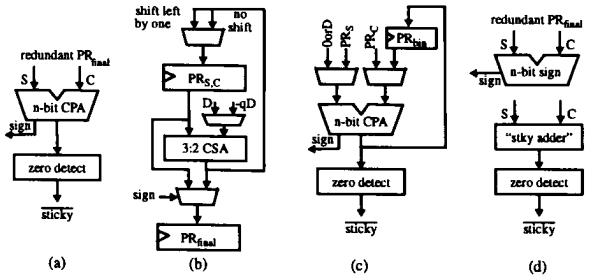


Fig. 3. Zero-detection and sign-detection on the final partial remainder.

Option 1 (Fig. 3b) takes advantage of existing csa hardware for restoration while option 2 (Fig. 3c) re-uses the carry-propagate adder. Both alternatives add extra multiplexer hardware and require the sticky bit calculation to take an additional cycle when the preliminary final partial remainder is negative. The first option especially impacts cycle time for basic iterations since the multiplexer is on the partial remainder formation critical path. Variable latency instructions in a pipelined superscalar processor make instruction scheduling and bypass control logic much more complex and is generally undesirable. Thus the net effect of restoring negative partial remainders is to add another cycle of latency for all divide and square root operations.

B. Improved Quotient Selection Algorithm

Enhancing the quotient digit selection function to prevent formation of a negative partial remainder for exact results is an ideal solution because it saves hardware and improves latency. This can be achieved by choosing a quotient digit of zero instead of one when the partial remainder is zero. This suggests choosing q=0 for $2PR_{est}$=000.0. Since the quotient digit selection function works on an estimated partial remainder, caution is required. An estimated partial remainder (shifted) could appear to be less than 1/2 when, in reality, adding the lower bits causes a 1 to propagate into the lowermost bit of the upper four bit partial remainder.

If the full partial remainder is 1/2 or above, q=1 should always be chosen since the divisor is constrained $1/2 \le D < 1$. The true quotient bit could be one. It will be corrected later if the divisor was greater than the partial remainder. There is no way to correct for under-estimation if q=0 is selected when q=1 was the correct quotient digit. The result will be irreversibly incorrect and the next partial remainder will be out of bounds.

Performing binary addition on the full partial remainder eliminates the estimation problem, but defeats the timing benefits of SRT division. q=0 could be chosen only when the full partial remainder is zero, but such detection would also be detrimental to timing.

A simple alternative is to detect a possible carry propagation into the least significant bit of the partial remainder. This can be done by looking at the fifth most significant bits of the redundant partial remainder, $PR_{S,msb-4}$ and $PR_{C,msb-4}$. If they are both zero, then propagation is not possible, and q=0 should be chosen; otherwise q=1 should be chosen. Even though lower bits of the partial remainder could be non-zero, the partial remainder is still within prescribed bounds and the correct result will be generated. As far as timing is concerned, the carry-propagate addition is still performed on four bits.

TABLE III
REFINED QSLC TRUTH TABLE

$2PR_{i,estimated}$	quotient $digit_{i+1}$
000.0 ($2PR_{S,C,msb-4}$ both 0)	0
0xx.x	+1
111.1	0
1xx.x	-1

Fig. 4 shows a logic implementation of this enhanced quotient digit selection function. In practice, the four bit adder and subsequent logic are merged into five stages of logic for more efficient timing and area utilization.

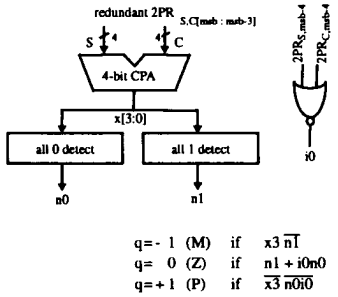


Fig. 4. Simple implementation of modified QSLC.

The number of additional gates needed to implement the new radix-2 quotient digit selection logic is relatively small. From Spice analysis, the impact on the qslc critical timing path was under five percent. There is an implementation dependent timing trade-off between slower quotient selection logic and eliminating the partial remainder restoration cycle at then end. Notice that if slowing down qslc doesn't limit the processor cycle time, there will always be a performance gain from saving one cycle of latency.

C. Parallel Sign Calculation and Zero-detection

It is possible to save hardware while also performing sign detection and sticky bit calculation in parallel as shown in Fig. 3d. Instead of using a full 59 bit adder, a 59 bit sign detect adder can be used, slightly improving timing, but mainly saving area. Zero-detection can be done without an explicit addition to convert the redundant partial remainder into binary.

$$t_i = \left(s_i \oplus c_i\right) \oplus \left(s_{i-1} + c_{i-1}\right) \tag{4}$$

where s_i and c_i are the sum and carry values of the final partial remainder. The sticky bit is computed by:

$$sticky = t_0 + t_1 + \ldots + t_n \tag{5}$$

This method generates inputs to the zero-detector with a 3-input xor delay instead of the delay of a 59 bit carry-propagate adder, a significant net savings.

IV. Overlapping Radix-2 Stages

The biggest performance gain is achieved by maximizing the number of iterations per cycle. As the number of result bits formed per cycle increases, the relative importance of saving one cycle of latency, as described in Section III, grows. A straightforward way of generating n result bits per cycle is to serialize the basic SRT radix-2 implementation. This solution is not attractive since the critical path includes n quotient selection logic delays and n carry save adder delays.

A. Optimal Timing via QSLC Overlapping

Overlapping quotient selection logic for the first and second iterations [12] as shown in Fig. 5 yields better timing results since only one qslc is in the critical path. Overlapping is achieved by performing $+D$ and $-D$ operations on the upper bits of the partial remainder while the first quotient selection bit is being determined. In this way, the second quotient digit selection can start before the first is finished.

Maximal overlapping can be extended to three bits per cycle to have an effective radix-8 implementation.

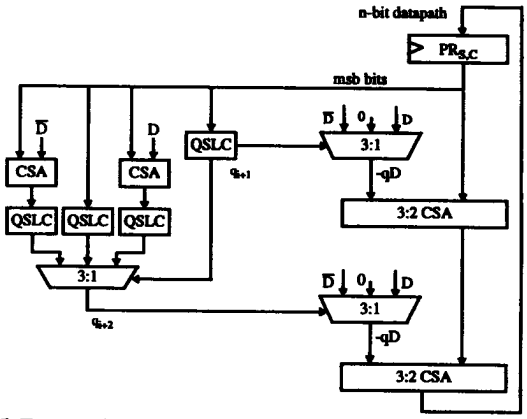


Fig. 5. Two overlapped SRT radix-2 QSLC stages.

Analyzing the possible partial remainders needed for the third quotient digit selection, as depicted in Fig. 6, shows that only seven csa's and qslc's are needed rather than nine. Since quotient selection logic is area intensive, a 22% reduction is quite beneficial.

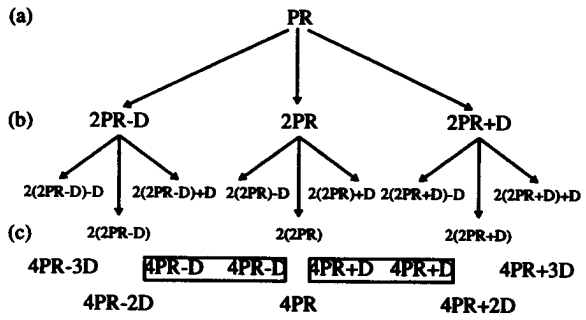


Fig. 6. Possible partial remainder values after the first and second iterations. (a) initial PR_i, (b) three possible PR_{i+1}, (c) seven possible PR_{i+2}.

Further timing improvement is also possible. By speculatively calculating the next partial remainder, quotient, and divisors for each possible quotient selection digit, the delay of a datapath carry-save adder can be masked by the longer qslc operation occurring in parallel. Fig. 7 shows the overall *Ultra*SPARC floating point divide, square root implementation. The critical path is reduced to 1 qslc, 2 csa's, and 3 muxes.

There is a timing, area trade-off. Overlapping improves timing at a cost of additional speculative hardware. The focus of this study is to optimize timing to the utmost with area minimization as a secondary goal.

B. Extension

In theory, n qslc stages can be overlapped. Assuming speculative datapath partial remainder calculations each iteration, the critical path for n bits per cycle is 1 qslc, (n-1) csa's, and n muxes. The incremental timing cost is one csa and one mux. There are 2^n-1 partial remainder

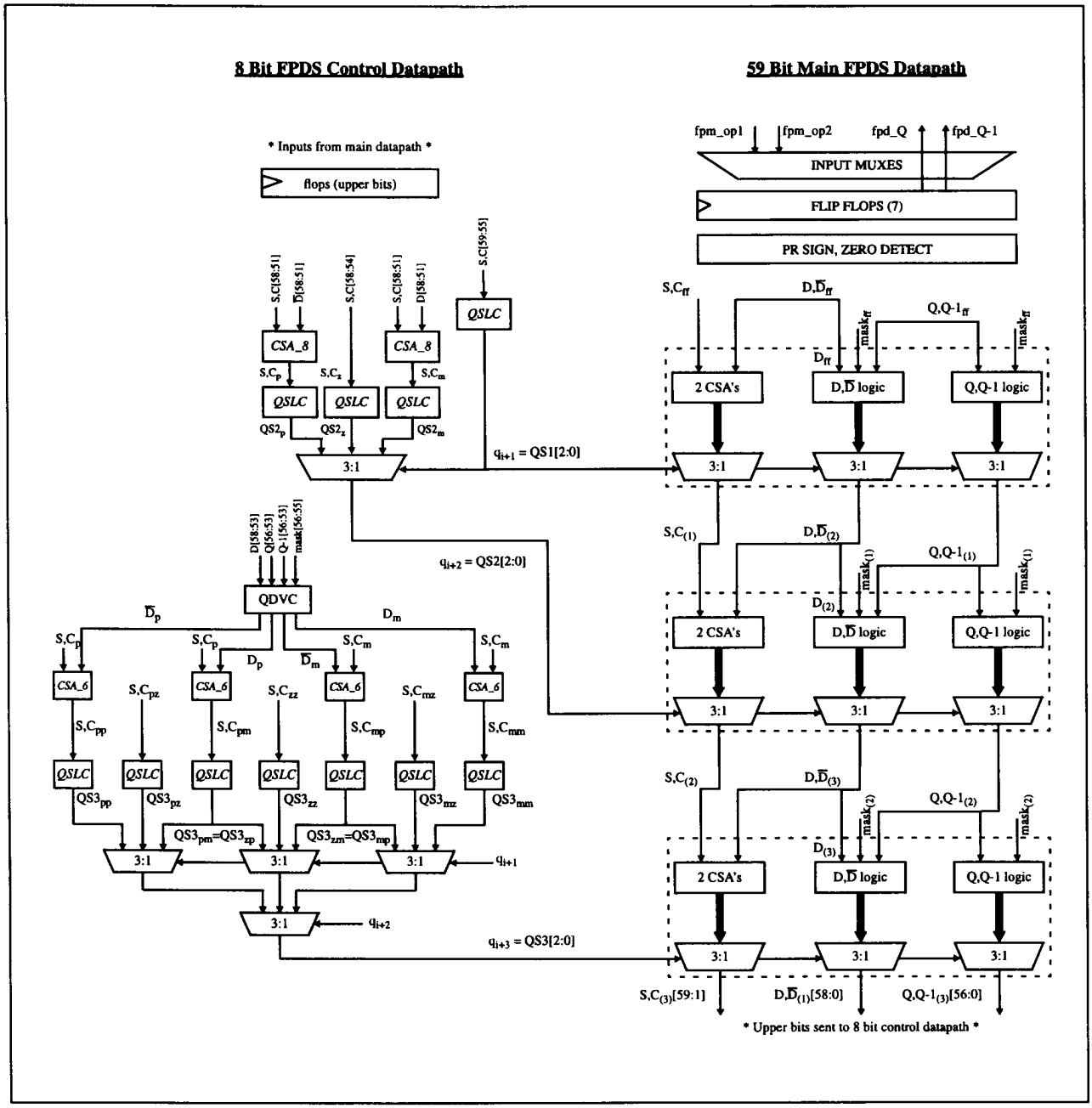


Fig. 7. *Ultra*SPARC radix-8 floating point divide, square root implementation with three overlapped radix-2 stages and speculative datapath calculation.

possibilities for the nth qslc stage. They are in the range:

$$2^{n-1}PR_i + [-(2^{n-1}-1)D,..,0,..,(2^{n-1}-1)D]. \quad (6)$$

Therefore, the incremental qslc cost for the nth overlapped stage is 2^n-1 rather than 3^{n-1} as suggested by Taylor [12]. In practice, overlapping two to four stages makes the most sense. As n gets higher, the number of qslc's grows exponentially making the area cost prohibitive. In addition, greater fanout leads to increased wire and gate loads which significantly lessen the timing benefits. Table IV summarizes timing and qslc cost considerations for overlapped radix-2 stages.

TABLE IV
PERFORMANCE, COST TABLE FOR MAXIMUM OVERLAPPING

stages	critical path	tot qslc's	delta critical path	delta qslc's
1	qs + mux	1	-	-
2	qs + pr + 2mux	4	pr + mux	3
3	qs + 2pr + 3mux	11	pr + mux	7
4	qs + 3pr + 4mux	26	pr + mux	15
n	qs + (n-1)pr +(n)mux	$\sum_{i=1}^{n} 2^i - 1$	pr + mux	$2^n - 1$

There is a limit on how much overlapping is necessary to achieve the optimal radix-2 implementation critical path as illustrated in a four bit per cycle example. Suppose the quotient selection logic delay is three times a carry-save adder delay. Then the quotient digits from the first stage qslc will be ready at the same time as the third stage partial remainder bits are entering the fourth stage qslc's. The first level of three-to-one muxes following the fourth stage of qslc's can be moved before the quotient selection logic as shown in Fig. 8. Thus, the fourth stage number of qslc's is reduced from fifteen to seven while achieving the same timing as with maximum overlapping. In general, the optimal degree of overlapping will depend on the relative csa and qslc delays, and need only be sufficient to mask the delay of previous quotient selection logic stages.

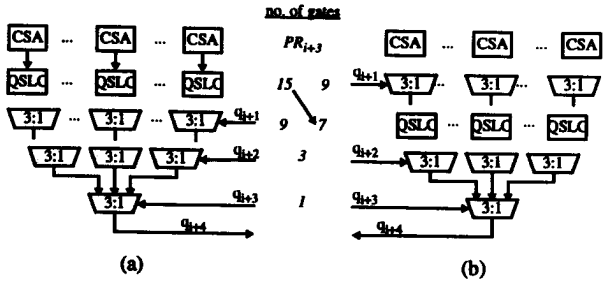


Fig. 8. Optimal radix-2 timing with (a) maximum overlapping, (b) reduced overlapping.

C. Overlapped Radix-2 versus Radix-4

Overlapping radix-2 stages yields better timing results than overlapping radix-4 stages. With radix-4, the upper eight bits of the redundant partial remainder and upper four bits of the divisor are sent to the quotient selection logic. The logic cannot be merged as with radix-2, so there is an explicit eight bit binary addition followed by a ten input, forty-four product term PLA [12].

Comparison of the critical path gate delays (gd) for two bits per cycle and four bits per cycle confirms better timing from a radix-2 based solution. The critical paths for two bits per cycle are as follows.

$$\underset{5}{qslc} + \underset{1.8}{csa} + \underset{(2\,x\,1)}{2\,mux3.} \quad (8.8gd) \quad (7)$$

$$\underset{6}{add8} + \underset{1}{pla} + mux5. \quad (7gd+pla) \quad (8)$$

Two overlapped radix-4 stages are implemented in the same way as depicted for radix-2 in Fig. 5. The critical paths for four bits per cycle are shown below.

$$\underset{5}{qslc} + \underset{(3x1.8)}{3\,csa} + \underset{(4\,x\,1)}{4\,mux3.} \quad (14.4gd) \quad (9)$$

$$\underset{1.8}{csa} + \underset{6}{add8} + pla + \underset{(2\,x\,1) = (9.8gd+pla)}{2\,mux5.} \quad (10)$$

A 10-input, 44-product term pla takes more than five gate delays, so for both two bits per cycle and four bits per cycle, overlapping radix-2 stages produces better timing. This analysis implies that a combined radix-4, radix-2 approach to yield an effective radix-8 solution is not faster than simply overlapping three radix-2 stages.

V. PROCESSOR IMPLEMENTATION

*Ultra*SPARC performs non-blocking divide and square root operations. Generating three result bits per cycle at 167 Mhz, the latency is twelve cycles for single precision (SP) and twenty-two cycles for double precision (DP). The divide and square root unit contains seventy-thousand transistors implemented in 0.5um CMOS technology (Fig. 9).

Instructions are both issued and completed through the pipelined floating point multiplier which formats operands, calculates the exponent, and performs rounding. The floating point multiplier and divider share the same register file read ports for operands, so divide and multiply instructions are never issued at the same time. Therefore the multiplier is always available for operand formatting and exponent calculation when a divide instruction is issued. Multiply instructions can be issued during subsequent cycles while the divider is iterating. Four cycles prior to division completing, a

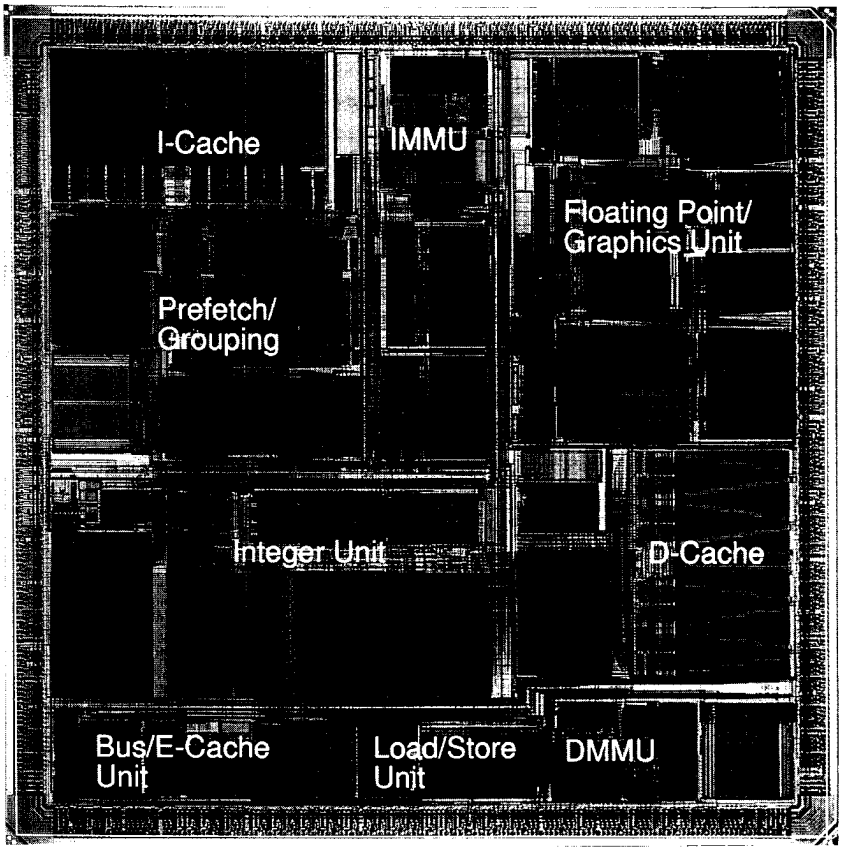
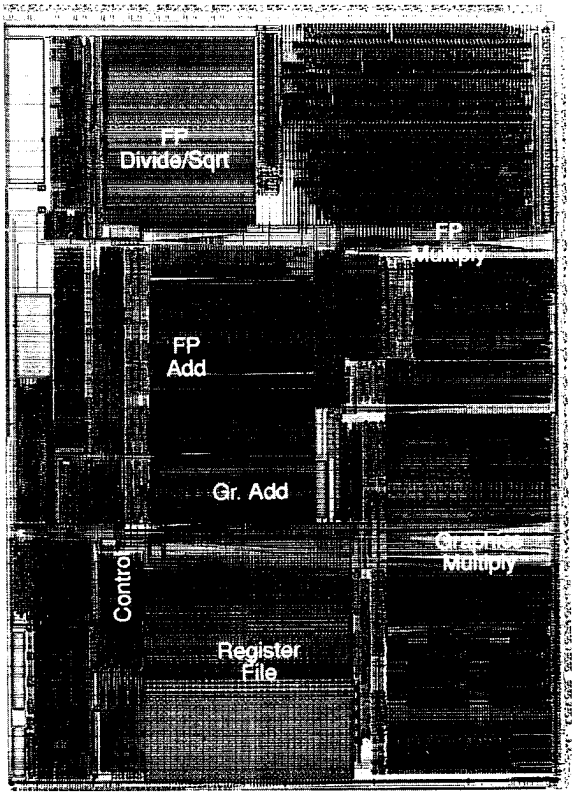


Fig. 9. *Ultra*SPARC (above), *Ultra*SPARC floating point, graphics unit (below).



multiply slot is reserved so that the divider can re-use the final stage multiplier rounding hardware. Since *Ultra*SPARC performs in-order execution, there will be a one cycle delay only in the unlikely event that the superscalar processor has been able to issue independent instructions for 8 cycles (SP) or 18 cycles (DP) and an independent multiply instruction is ready to be issued during this exact cycle. Thus making use of the multiplier yields significant hardware savings with near zero performance impact.

The modified quotient selection logic algorithm slowed the internal floating point divider critical path by approximately 100ps, or by less than two percent. The limiting timing path on the processor was slower, so there was no negative impact from the improved qslc. The full benefit of eliminating the cycle for restoring the partial remainder prior to sticky detection was realized.

Division and square root operations were thoroughly tested with 100% toggle coverage and 100% finite state machine arc coverage in a stand-alone test (SAT) environment, at the cpu level, and in silicon. Over 860,000 directed vectors and 140,000 random vectors were applied at the SAT level. Included were pseudo-exhaustive double precision tests for division in which all combinations of the upper eight bits of the dividend and divisor were sequenced (64K vectors). The same was done for square root in which all combinations of the upper thirteen bits of square root operands for odd and even exponents were sequenced (16K vectors).

A radix-2 solution was preferable over radix-4 for a number of reasons including timing, as described earlier, and greater flexibility. From the outset, it was known that it might be necessary to scale back the number of bits per cycle due to timing or area considerations. That raised the specter of having to reduce to two bits per cycle in a radix-4 implementation. Going with radix-2 provided a better safety option since three bits per cycle yields 6% better overall floating point performance (SPECfp92) than two bits per cycle [13].

Implementing square root did not come for free. While it is true that square root completely re-uses existing divide hardware, additional logic was required to generate the so-called divisors used in square root. In addition, the quotient had to be developed on the fly, as opposed to using a shift register and assimilating the redundant bits during the final cycle, since it was needed for the divisor calculation. Additional datapath feedthrough tracks were also necessary. An estimated 15% of the area went to support square root. With careful design, the logical critical path for square root was made the same as for divide. The additional area required to support square root, though, did impact timing since wire delays were greater.

VI. Conclusion

High-speed floating point division and square root is achieved by overlapping radix-2 quotient selection logic stages and speculatively calculating the partial remainder, quotient, and next divisor. An enhanced quotient digit selection function prevents the working partial remainder from becoming negative if the result is exact. This translates into a one cycle savings since negative partial remainders no longer need to be restored before calculating the sticky bit.

Acknowledgment

The authors would like to thank Marc Tremblay and Guy Steele for reviewing the preliminary draft, and Nasima Parveen and Richard Landes for their contributions to verification and physical design.

References

[1] J. E. Robertson, "A new class of digital division methods," *IEEE Trans. Comput.*, vol. C-7, pp. 218-222, Sept. 1958.

[2] 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.

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

[4] M. D. Ercegovac and T. Lang, "Radix-4 square root without initial PLA," *IEEE Trans. Comput.*, vol. 39, no. 8, pp. 1016-1024, Aug. 1990.

[5] S. Majerski, "Square root algorithms for high-speed digital circuits," *Proc. Sixth IEEE Symp. Comput. Arithmetic.* pp. 99-102, 1983.

[6] D. Zuras and W. McAllister, "Balanced delay trees and combinatorial division in VLSI," *IEEE J. Solid-State Circuits*, vol. SC-21, no. 5, pp. 814-819, Oct. 1986.

[7] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994, ch 3.

[8] S. Majerski, "Square-rooting algorithms for high-speed digital circuits," *IEEE Trans. Comput.*, vol. C-34, no. 8, pp. 724-733, Aug. 1985.

[9] P. Montuschi and L. Ciminiera, "Simple radix 2 division and square root with skipping of some addition Steps," *Proc. Tenth IEEE Symp. Comput. Arithmetic.* pp. 202-209, 1991.

[10] V. Peng, S. Samudrala, and M. Gavrielov, "On the implementation of shifters, multipliers, and dividers in floating point units," *Proc. Eighth IEEE Symp. Comput. Arithmetic.* pp. 95-101, 1987.

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

[12] G. S. Taylor, "Radix 16 SRT dividers with overlapped quotient selection stages," *Proc. Seventh IEEE Symp. Comput. Arithmetic.* pp. 95-101, 1985.

[13] M. Tremblay, "A fast and flexible performance simulator for micro-architecture trade-off analysis on UltraSPARC," Submitted to the 1995 *Design Automation Conference*.