167 MHz Radix-4 Floating Point Multiplier

Robert K. Yu and Gregory B. Zyner

SPARC Technology Business, Sun Microsystems Inc.
Sunnyvale, California

Abstract - **An IEEE floating point multiplier with partial support for subnormal operands and results is presented. Radix-4 or modified Booth encoding and a binary tree of 4:2 compressors are used to generate the 53x53 double-precision product. Delay matching techniques were used in the binary tree stage and in the final addition stage to reduce cycle time. New techniques in rounding and sticky-bit generation were also used to reduce area and timing. The overall multiplier has a latency of 3 cycles, a throughput of 1 cycle, and a cycle time of 6.0ns. This multiplier has been implemented in a 0.5um static CMOS technology in the *Ultra*SPARC RISC microprocessor.**

I. Introduction

Multiplier units are commonly found in digital signal processors and, more recently, in RISC-based processors[1]. Double-precision floating point operations involve the inherently slow operation of summing 53 partial products together to produce the product. IEEE-compliant multiplication also involves the correct rounding of the product, adjustment to the exponent, and generation of correct exception flags. Multiplier units embedded in modern RISC-based processors must also be pipelined, small, and *fast*. Judicious functional and physical partitioning are needed to meet all these requirements[5][6][14]. In this paper, Section II describes the partial subnormal support. Section III describes details of the implementation, including the 4:2 compressor design, the binary tree composition, final addition, rounding, and sticky-bit generation. Finally, Section IV concludes this paper.

II. Subnormal Operations

Multiplier units that handle subnormal or denormal operands and results often require the determination of leading zeros, adjustments to mantissas (shifting) and exponent, and rounding. Overall timing and area are affected when subnormal operations are fully supported. However, by introducing a modest amount of hardware to compare only the value of the exponents, we can provide support for a large subset of the subnormal operations. This partial subnormal support does not require detection of leading zeros, adjustments of subnormal mantissas, and does not introduce an extra cycle penalty.

Figure 1 shows the floating point data formats, and Figure 2 shows their value definitions according to IEEE standards. The value of the mantissa is formed by concatenating the implicit bit with the fraction. For normalized values, the implicit bit is one, and for subnormal values, the implicit bit is zero. Note that for subnormal values, the convention is to have the exponent field zeroed, but the *value of the exponent* is taken to be one.

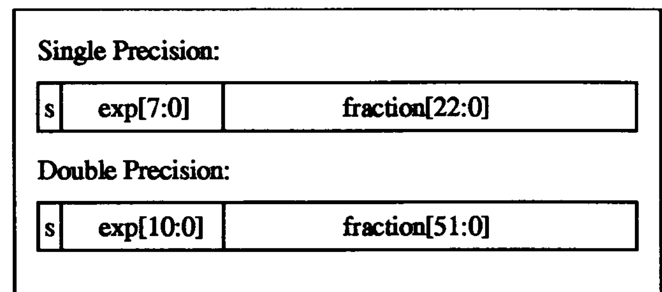

Fig. 1. Floating point formats.

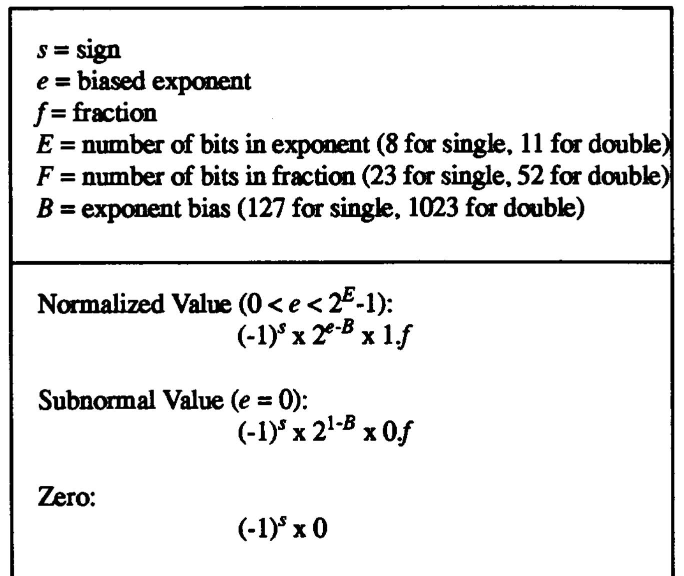

Fig. 2. Floating point format definition.

In multiplication, the resultant exponent e_r is calculated by:

$$e_r = e_1 + e_2 - B - z_1 - z_2 \quad (1)$$

where e_1 and e_2 are the biased exponents, and z_1 and z_2 are the leading zeros in the mantissas to the multiplicand and mul-

1063-6889/95 $4.00 © 1995 IEEE

tiplier, respectively. We can simplify Equation 1 by noting that if both operands are subnormal, then the value of e_r underflows to a value no greater than $-B$, and cannot be represented in the given precision[1]. We can therefore impose the constraint that only one operand can be subnormal without the loss of generality and rewrite this equation as:

$$e_r = e_1 + e_2 - B - z \quad (2)$$

where z is the number of leading zeros of the subnormal operand, if any. We further note that if the resulting value e_r is less than one, then the amount by which the resulting mantissa needs to be right-shifted to set $e_r = 1$ is:

$$rshift = 1 - e_r \quad (3)$$

If the *rshift* value is greater than the number of bits in the mantissa, then again we underflow beyond the dynamic range of the given precision. We define this condition as "extreme underflow."[2] Specifically, if

$$rshift \geq (F + 3) \quad (4)$$

then the mantissa will be right-shifted to the sticky-bit position or beyond and the resulting mantissa is either zero or the smallest subnormal number, depending on the rounding mode. The two cases where rounding will produce the smallest subnormal number are 1) rounding to plus infinity and the result is positive and 2) rounding to minus infinity and the result is negative. All other rounding modes including rounding to nearest and rounding to zero produces zero as the result.

Rewriting Equation 4 in terms of e_r, extreme underflow occurs when

$$e_r \leq -(F + 2) \quad (5)$$

We note from Equation 2 that e_r requires the detection of leading zeros z. If we ignore z altogether, which greatly simplifies the implementation, then Equation 5 becomes a conservative criterion for extreme underflow. That is, using only

$$e_r = e_1 + e_2 - B \quad (6)$$

to determine extreme underflow ignores those cases where extreme underflow would occur because of leading zeros present in the subnormal mantissa.

Equation 5 and Equation 6 form the basis used to provide partial subnormal support in this design. If Equation 5 is satisfied with the appropriate rounding mode, then the multiplier generates a zero result. The multiplier does not support the case where Equation 5 is not satisfied and a subnormal operand is encountered.

1. Does not apply to single precision multiplication resulting in a double precision result, or "fsmuld".
2. Sometimes loosely referred to as "gross underflow".

III. Implementation

A. Folded 3-Stage Pipeline

The multiplier operates over three stages. In the first stage, the multiplicand and multiplier operands go through the radix-4 encoding and multiply tree[3][4][7]. The intermediate summations of partial products and the result of the tree are in carry-save format. In the second stage, a conditional-sum adder is used to determine the product, converting the result from carry-save to binary form. In the third stage, rounding and flag generation is performed. The multiplier has a single-cycle throughput and a three-cycle latency.

B. Stage 1: Multiply Tree

1) Interleaved Binary Tree with Delay Matching

The first stage of a 53x53 bit multiplication of the mantissas is performed by a radix-4 Booth encoded binary tree of 4:2 compressors, or 5:3 counters. Figure 3 shows a schematic of the binary tree. The encoding scheme produces 27 partial products which are generated at blocks 0,1,3,4,7,8, and 10. Since 4:2 compressors are used, each block generates 4 partial products, except for block 10 which generates 3.

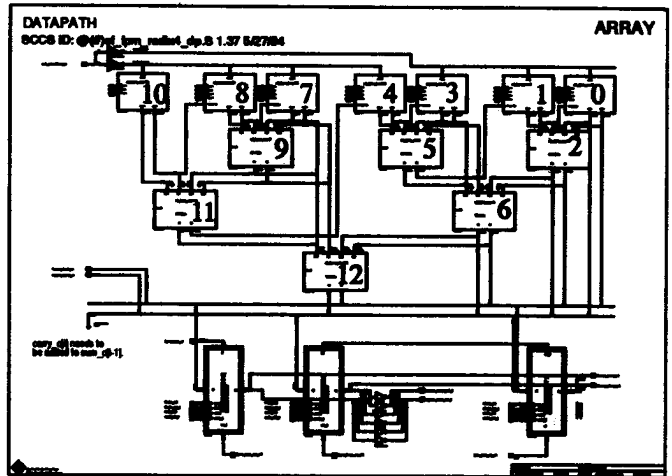

Fig. 3. Schematic of radix-4 binary tree.

Unlike traditional implementations, where inputs flow starting from the top and side of the tree to the bottom of the tree, this implementation has the multiplier and staged results placed on the same side of the tree. That is, pipeline register are embedded in the tree and are routed to the same side as the multiplicand, as shown in Figure 4. The advantage of this approach is to reduce interconnect lengths and to push some of the interconnect delay to the next stage.

The complexity of a binary tree does not lend itself to a straight-forward layout[2]. To minimize the delay through the tree due to interconnects, the placement of the rows of partial product generators and adders are done such that wire lengths are balanced among the rows. Both the vertical distance and

the horizontal distance, which comes about because the tree is "left-justified" and is significant in some cases, were taken into account.

Table I shows the vertical row distances and horizontal bit distances between cells on different rows. The critical path through the array involve the rows with large horizontal shifts, namely rows 0, 2, 6, and 12. These rows have been placed close together to reduce this path. Figure 4 shows the placement used[8].

TABLE I
DISTANCE BETWEEN CELLS

Row Transition	*Horizontal Distance*	*Vertical Distance*
0 -> 2	9	1
1 -> 2	1	2
3 -> 5	9	1
4 -> 5	1	3
7 -> 9	9	1
8 -> 9	1	2
2 -> 6	17	3
5 -> 6	1	3
9 -> 11	3	3
10 -> 11	0	7
6 -> 12	18	4
11 -> 12	0	4

2) Folded Adder Rows

Another problem presented by the irregular tree structure is the differing number of bits in each row, which varied from 61 to 76 bits. In order to reduce the area of the tree, some of the adders in the larger rows were "folded" to rows with fewer cells. The folding was done such that timing was not affected. More folding could be done but not without impacting the critical path through the tree.

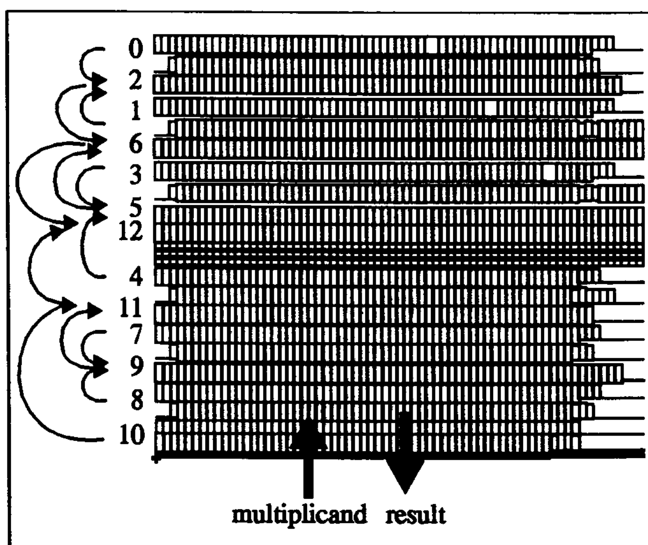

Fig. 4. Block diagram of binary tree showing the ordering of the rows used to balance the interconnect delays.

3) 4:2 Compressor Design

The 4:2 compressor schematic is shown in Figure 5. This adder takes 5 inputs {*x3, x2, x1, x0,* and *cin*} and generates 3 outputs {*carry, cout,* and *sum*}. All inputs and the *sum* output have a weight of one, and two outputs *carry* and *cout* have a weight of two[10][12]. That is,

$$\begin{aligned} & 2^0 \cdot (x3 + x2 + x1 + x0 + cin) \\ & = 2^1 \cdot (carry + cout) + 2^0 \cdot sum \end{aligned} \quad (7)$$

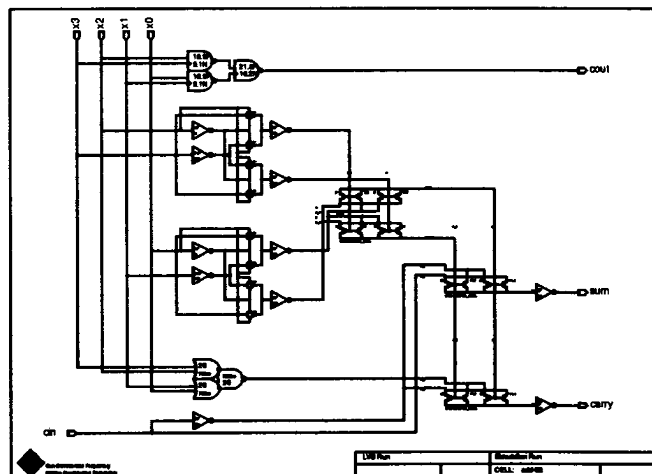

Fig. 5. Circuit schematic of 4:2 compressor.

The 4:2 compressor has been designed according to Table II. Since the *cout* signal is independent of the *cin* input and only dependent on the *x* inputs, a row of such adders hooked up together as shown in Figure 6 will not exhibit any rippling of carries from *cin* to *cout*. The adder has also been designed such that the delay from *xi* to *sum* or *carry* is approximately the same as the combined delay from *xi* to *cout*, and

TABLE II
TRUTH TABLE FOR 4:2 COMPRESSOR

x3	*x2*	*x1*	*x0*	*cout*	*carry*	*sum*
0	0	0	0	0	0	*cin*
0	0	0	1	0	*cin*	$\overline{cin}$
0	0	1	0	0	*cin*	$\overline{cin}$
0	0	1	1	1	0	*cin*
0	1	0	0	0	*cin*	$\overline{cin}$
0	1	0	1	0	1	*cin*
0	1	1	0	0	1	*cin*
0	1	1	1	1	*cin*	$\overline{cin}$
1	0	0	0	0	*cin*	$\overline{cin}$
1	0	0	1	0	1	*cin*
1	0	1	0	0	1	*cin*
1	0	1	1	1	*cin*	$\overline{cin}$
1	1	0	0	1	0	*cin*
1	1	0	1	1	*cin*	$\overline{cin}$
1	1	1	0	1	*cin*	$\overline{cin}$
1	1	1	1	1	1	$\overline{cin}$

cin to *sum* or *carry* of an adjacent adder.

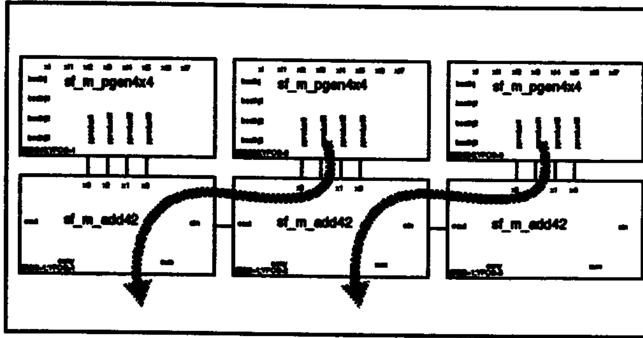

Fig. 6. Interconnect of 4:2 compressors with no horizontal ripple carry.

C. Stage 2: Final Addition

1) Optimized Conditional-Sum Adder

The final add stage of the multiplier makes use of a 52-bit conditional sum adder that is partitioned for minimum delay. Figure 7 shows a block diagram of the recursive structure of the conditional sum adder. As shown in the diagram, an N-bit conditional sum adder is made up of two smaller conditional sum adders, one that is j-bits, and one that is N-j bits wide. Two 2:1 muxes are used to output the upper sum and carry results; these outputs are selected by the carries from the lower j-bit adder. Typically the selects to the muxes are buffered up to handle the capacitive loading due to large fanouts. These smaller adders are, in turn, made up of smaller conditional sum adders[11].

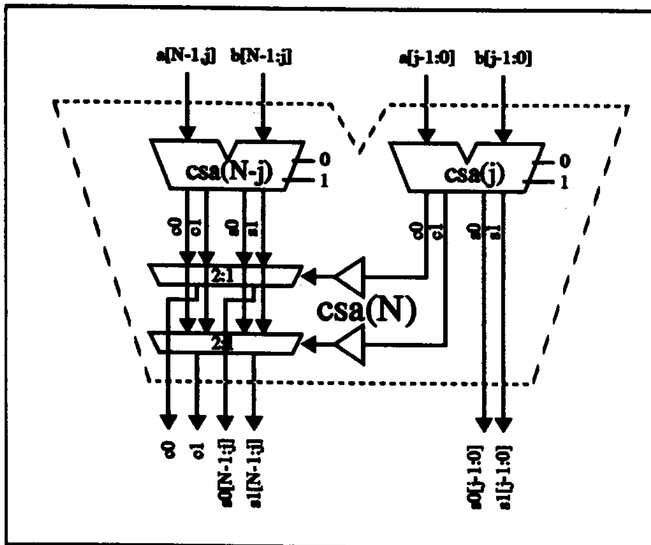

Fig. 7. Recursive structure of conditional sum adder.

The delay through this adder is affected by how the adders are partitioned. We define the delay for an N-bit adder partitioned at position *j* as *T(N,j)*:

$$T(N,j) = \max[T_{opt}(j) + T_{buf}(N-j) + T_{sel}, T_{opt}(N-j) + T_{mux}] \quad (8)$$

In Equation 8, $T_{opt}(i)$ is the optimal delay for an i-bit adder, T_{buf} is the buffer delay and is a function of the number of bits on the left adder, or N-j, T_{sel} is the select to out delay of the mux, and T_{mux} is the data to out delay of the mux. The minimum delay for an N-bit adder $T_{opt}(N)$ is simply:

$$T_{opt}(N) = \min[T(N,j)] \quad (9)$$

where j varies from 1 to $N-1$. The problem of finding $T_{opt}(N)$ is a recursive min-max problem and lends itself well to an efficient dynamic programming solution developed internally for this implementation.

2) Sticky-Bit Generation from Carry-Save Format

The *sticky* bit, which is needed to perform the correct rounding, is generated in the second stage. Typically, the lower 51 bits in carry-save format are summed and OR'ed together. However, in our technique, we are able to generate the *sticky* bit directly from the outputs of the tree in carry-save format without the need for any 51-bit adder to generate the sum beforehand, resulting in significant timing and area savings. We define:

$$p_i = s_i \oplus c_i \quad (10)$$

$$h_i = s_i + c_i \quad (11)$$

$$t_i = p_i \oplus h_{i-1} \quad (12)$$

where s_i and c_i are the sum and carry outputs from the tree. The *sticky* bit is then computed directly by using a ones-detector[9]:

$$sticky = t_0 + t_1 + \ldots + t_{50} \quad (13)$$

D. Stage 3: Rounding

1) Using Conditional Sum Adders

By using a conditional sum adder to generate both the sum and sum+1, we remove the need for an incrementer to perform the rounding operation. Only multiplexing is needed to select the correct result after rounding[13].

2) Overflow After Rounding

In double precision multiplication, after the 106-bit product (in carry-save format) has been generated by the array, the decimal point occurs between bits 104 and 103, and only the upper 53-bits are used for the mantissa result. The lower 53-bits are needed only to perform the correct rounding. After rounding is performed, either bits 105-53 or bits 104-52 are used depending on the value of bit 105 or MSB. If this MSB is set, then the mantissa is taken from bits 105-53, and the value

of the exponent is incremented. Otherwise, if the MSB is not set, then bits 104-52 are used, and the exponent is not incremented. Note that the rounding itself may propagate to set the MSB; this is the case of overflow *after* rounding. Figure 8 shows how the mantissa is selected from the array result depending on bit 105 after rounding. In the figure, L, G, R, and S represents the LSB, guard, round, and sticky bits, respectively.

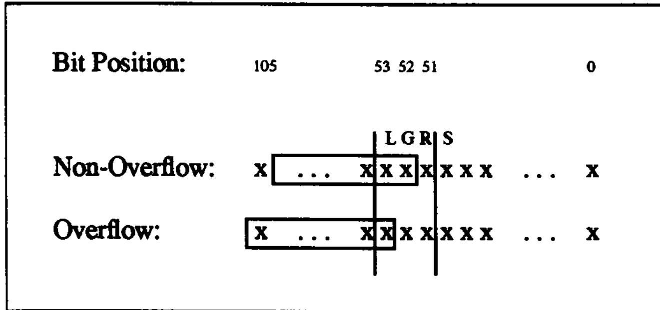

Fig. 8. Mantissa selection for overflow and non-overflow.

Figure 9 shows a block diagram of the rounding datapath and logic. The dotted line represents pipeline registers: the final add operation and rounding are done in separate stages. The lower 50 bits from the array are used to generate two signals: c51and S. The c51 signal is the carry into bit 51. Bits 53 through 51 along with c51 are added to create the L, G, and R bits, and the rest of the bits 105:54 are added using a conditional sum adder to form two results sum0[105:54] and sum1[105:54].

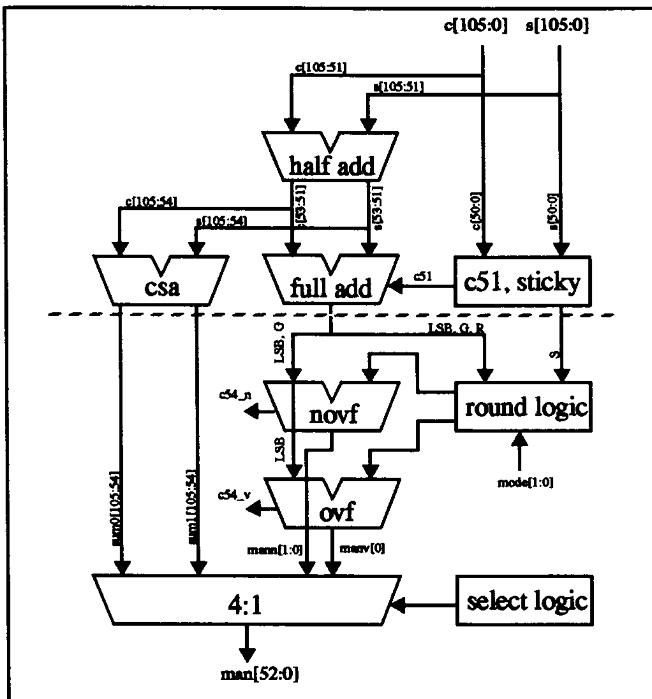

Fig. 9. Rounding.

Because of the c51 signal and a rounding that may occur at bits 53 or 52, there is a possibility of introducing two carries into bit position 54. To ensure that only one carry is propagated into bit 54, a row of half-adders is used at bits 105 through 51.

To correctly handle the case of overflow *after* rounding, our implementation makes use of two adders, ovf and novf, to generate the signals c54_v and c54_n, respectively, which are needed by the final selection logic. The c54_v and c54_n are the carries into bit position 54 assuming an overflow and non-overflow, respectively. The L, G, R, S, and rounding mode bits are used by the round logic to generate two rounding values. One value assumes a mantissa overflow, and the other assumes no mantissa overflow. These rounding bits are added to the L and {L,G} bits to form the lower one and two bits of the resulting mantissa for overflow (manv) and non-overflow (mann), respectively.

3) Final Selection

The final select logic combines the appropriate sum0 and sum1 from the conditional sum adder with either manv or mann to form the final mantissa. Table III shows the truth table to the selection logic. The key to the table is the expression for the *Overflow* signal, shown in Equation 14. The first expression refers to the case where the MSB is set as a result of a carry within the addition of the 51 bits without a carry into bit 54. The second expression refers to the case where the MSB is set due to some carry into bit 54 in the non-overflow case. This carry may be due to rounding itself, or the case of overflow after rounding.

$$Overflow = \text{sum0}[105] + (\text{c54_n} \cdot \text{sum1}[105]) \quad (14)$$

TABLE III
SELECTION LOGIC

Overflow	*c54_n*	*c54_v*	*Select*
0	0	x	sum0[104:54], mann[1:0]
0	1	x	sum1[104:54], mann[1:0]
1	x	0	sum0[105:54], manv[0]
1	x	1	sum1[105:54], manv[0]

4) Shared Hardware with Divide and Square Root

In our implementation of the floating point unit, the rounding for multiplication is similar to that needed for division and square root. In the interest of saving area, multiplication, division, and square root all share the same rounding hardware. Only additional muxing between the multiply, divide, or square root results is required before the inputs to the block shown in Figure 9.

One difference, however, between multiplication, division, and square root is the handling of the mantissas overflow. In

multiplication, the incremented exponent is used if an overflow occurs. In division and square root, however, the decimal point is taken to be immediately to the right of the MSB. Therefore, if the mantissa's MSB is zero, then the decremented exponent is selected. Table IV shows how the exponent is selected for multiplication, division, and square root.

TABLE IV
EXPONENT SELECTION FOR MULTIPLY AND DIVIDE

Mantissa	*Multiply*	*Divide/Sqrt*
Overflow	$e_r + 1$	e_r
Non-overflow	e_r	$e_r - 1$

IV. CONCLUSION

We have presented the design and implementation of a high-speed floating point multiplier. Partial subnormal support has been implemented with minimal addition to hardware and penalty on performance. Delay matching techniques were used in the multiplier tree and in the final addition stages. The rounding hardware is shared with the divide and square root units.

V. ACKNOWLEDGEMENT

The authors would like to thank Marc Tremblay, Arjun Prabhu, and the Program Committee for their valuable suggestions, and Nasima Parveen for her contributions in verification.

REFERENCES

[1] M. Mehta, et al, "High-speed multiplier design using multi-input counter and compressor circuits," *Proceedings 10th Symposium on Computer Arithmetic*, pp. 43-50, 1991.

[2] M. Nagamatsu, et al, "A 15 ns 32x32 bit cmos multiplier with an improved parallel structure," *Custom Integrated Circuits Conference*, pp. 10.3.1-10.3.4, 1989

[3] L. P. Rubinfield, "A proof of the modified Booth's algorithm for multiplication," *IEEE Trans. Comput.*, pp. 1014-1015, Oct. 1975

[4] C. S. Wallace, "A suggestion for parallel multipliers," *IEEE Trans. Electron. Comput.*, vol. EC-13, pp. 14-17, Feb. 1964

[5] L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349-356, 1965

[6] L. Dadda, "On parallel digital multipliers," *Alta Frequenza*, vol. 45, pp. 574-580, 1976.

[7] A. D. Booth, "A signed multiplication technique," *Quarterly J. Mechan. Appl. Math*, vol. 4, pt. 2, pp. 236-240, 1951

[8] 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

[9] Email correspondence with Vojin Oklobdzija, 1993.

[10] D. T. Shen and A. Weinberger, "4-2 carry-save adder implementation using send circuits", *IBM Technical Disclosure Bulletin*, vol. 20, no. 9, Feb 1978.

[11] J. Sklansky, "Conditional sum addition logic", *Trans. IRE*, vol. EC-9, no. 2, pp. 226-230, June 1960.

[12] M. Santoro and M. Horowitz, "A pipelined 64x64b iterative array multiplier", *IEEE Int. Solid-State Circuits* conf., pp.35-36, Feb. 1988.

[13] M. Santoro, G. Bewick, and M. Horowitz, "Rounding algorithms for IEEE multipliers", *IEEE 9th Symposium on Computer Arithmetic* proceedings, pp. 176-183, Sept. 1989.

[14] V. Peng, S. Sanudrala, M. Gavrielov, "On the implementation of shifters, multipliers, and dividers in VLSI floating point units", *IEEE 8th Symposium on Computer Arithmetic* proceedings, pp. 95-102, May 1987.