1-GHz HAL SPARC64® Dual Floating Point Unit with RAS Features

Ajay Naini, Atul Dhablania
Warren James, Debjit Das Sarma
HAL Computer Systems
1315 Dell Avenue
Campbell, CA 95008, USA

Abstract

An IEEE compliant, 1 GHz Sparc64-V Floating-Point Unit (FPU) with reliability-accessibility-serviceability (RAS) features and partial support for denormal operands and results is presented. The FPU has two functional units, each with an adder (FADD) and a multiplier (FMUL). Additionally, one of the functional units contains a graphics unit (VIS). Two floating-point instructions can be scheduled out of order each cycle, one to each functional unit. A peak performance of 4 GFLOP is achieved by scheduling two floating-point multiply add (FMA) instructions each cycle. The FADD unit is fully pipelined and can execute an addition, subtraction, conversion, or compare instruction every cycle. The FMUL unit executes pipelined multiply instructions. Divide and square-root instructions are executed with multiple iterations through the multiplier pipeline. The VIS unit is also pipelined and executes SIMD fixed-point graphics instructions. The adder and multiplier have latencies of 3 and 4 cycles respectively. Novel techniques are presented in the adder and multiplier implementations to reduce area and cycle time. The FPU provides RAS support for enhanced server reliability by using selective parity error detection. The FPU has been implemented in 0.15u, 6-layer metal CMOS technology.

1. Introduction

The HAL Sparc64 V (Sparc64-V) microprocessor targeted for large-scale SMP systems is SPARC V9 [1] compatible with the Visual Instruction Set (VIS2.0) extensions [2]. The SPARC64-V processor is an eight issue superscalar out-of-order chip with 1 GHz clock cycle in 0.15 micron 6-layer metal CMOS technology.

The Sparc64-V Floating-point Unit (FPU) has two identical floating-point functional units, one graphics (VIS) unit and one Reservation Station (FRS). Every cycle, two FPU/VIS instructions can be issued to the 16-entry FRS. The FRS can schedule two FPU instructions or one FPU and one VIS instruction per cycle. Each FPU has an Add unit (FADD) and a Multiplier unit (FMUL). The FADD unit is fully pipelined and can execute one Addition, Subtract, Conversion, Compare, or Move instruction every cycle. The FMUL unit implements Multiply, Divide, and Square-root instructions. The Multiply instruction is fully pipelined, but the Divide and Square-root instructions are not pipelined. The FMUL unit sends a *done* signal to the Scheduler after the Divide/Square-root instruction is completed so new instruction can be scheduled. The VIS unit is fully pipelined and implements all the VIS2.0 instructions.

Sparc64-V supports two additional non-SPARC instructions, unfused-FMADD (uFMADD) and unfused-FMSUB (uFMSUB). Unlike fused multiply-add where rounding is performed only once after the add operations, rounding is done twice for these unfused instructions (multiply, round, add/subtract, round). The rounding after the multiply operation enables us to generate a result that is compliant with IEEE-754 rounding and detect exceptions precisely. If multiply operation generates an exception, the add/subtract operation is not executed and the exception state of the multiply operation is forwarded to the trap handler. These instructions need three operands; *src1* and *src2* go the FMUL unit, *src3* goes to the FADD unit. The FMUL result is forwarded to the FADD unit via an internal FMA bus as shown in Figure 1. The add operation is performed on the multiply result and *src3*. New instructions (add, subtract, conversion, compare, and move) are not scheduled to the FADD unit during this cycle. The architecture supports the issue and execution of two uFMADD/uFMSUB (two FMUL and two FADD/FSUB) every cycle for a peak execution rate of 4 GFLOPS.

The block diagram of floating-point unit is shown in Figure 1. The FRS has six read ports, three for Funit-A and three for Funit-B. The three sources are needed for uFMADD/uFMSUB and PDIST (VIS) instructions. The FADD/FMUL (referred to as Funit-A) results are distributed on FPA result bus and FADD/FMUL/VIS (referred to as Funit-B) results are distributed on FPB result bus. Since the result bus wire lengths are long and back-to-back instruction execution is important for performance, one-half cycle is reserved for result distribution. For example, if the ADD instruction latency

0-7695-1150-3/01 $10.00 © 2001 IEEE

is three cycles, the ADD operation is done in two-and-half cycle and the last half cycle is used for result distribution. This allows for the dependent instruction to start execution in the fourth cycle. The FADD unit and FMUL unit implementation details are explained in their respective sections.

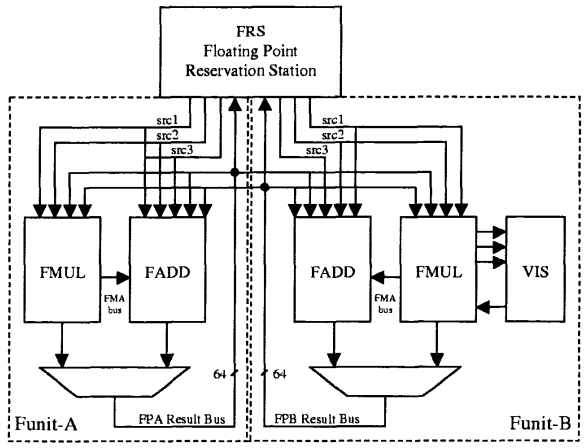

Figure 1: FPU block diagram

The VIS unit implements a subset of the VIS2.0 instructions that access the floating-point register file. Other VIS2.0 instructions are implemented in the Integer unit. VIS2.0 supports all the standard fixed-point graphics instructions (logical, alignment, arithmetic, compare, etc). All VIS instructions are SIMD type with 8-bit, 16-bit or 32-bit operands. The VIS unit gets operands from the FMUL unit and sends the result back to the FMUL unit. This decouples the VIS unit from the source and result distribution paths, ensuring that the FPA and FPB result bus distribution latency for FADD and FMUL units is unaffected. The data distribution from FMUL unit to VIS unit takes one-half cycle and the result distribution from VIS unit to FMUL unit takes another one-half cycle. So the minimum latency for a VIS instruction is two cycles, of which, only one-half cycle is allocated for execution as outlined in Figure 2.

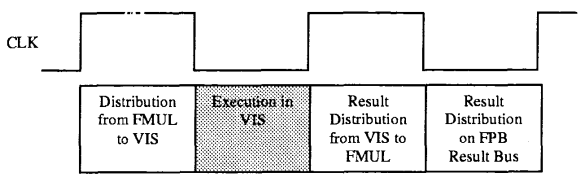

Figure 2. VIS pipeline for AND instruction

The FPU supports both single precision and double precision operations and conforms to the IEEE-754 standard. The latencies for single and double precision are the same for all instructions except for divide and square-root. The latencies of the important floating-point instructions are shown in Table 1. The Divide and Square-root instructions have a throughput less than their latencies because the new instruction can be started after the last iteration of the Divide/Square-root passes through the multiplier array. Since the instructions have different latencies, it is the scheduler's responsibility to avoid collisions on the result buses.

Table 1. Instruction latencies

Instruction(s)	Latency	Throughput
FADD/FSUB	3	1
FMUL	4	1
FDIVs	16	13
FDIVd	19	16
FSQRTs	22	19
FSQRTd	27	24
uFMADD/uFMSUB	7	1

Denormal operand and result handling/generation often require extra hardware and impact either cycle time or latency. A subset of denormal operations can be handled by manipulating only the exponent. Sparc64-V is able to support these operations with the addition of a minimal amount of hardware. Any denormal operation that involves mantissa normalization and rounding is not supported because handling them will not only require significant extra hardware but can affect the cycle time of the operation. Denormal operations in Sparc64-V are explained in detail in the denormal operations section.

In silicon, charged particles are generated by cosmic rays or alpha particles from radioactive decay of the chip materials. A high concentration of these charged particles can cause a temporary, localized circuit failure. This soft error behavior can be identified and fixed most of the time. Until recently, soft error detection and repair, referred to as RAS (Reliability, Accessibility, Serviceability), was featured only in mainframes because of the associated area and cycle time costs. The Sparc64-V microprocessor supports some select RAS features to achieve a target error rate of 240 fit (failure per 10^9 hours). This is achieved by protecting the major datapaths and arrays either with parity or ECC (Error Correcting Code). If an error is detected, the instruction is re-scheduled and soft error correction is attempted on the re-execution. All unresolved soft errors are logged into software accessible registers. In the floating-point unit, appropriate choices were made to support RAS without compromising the performance (latency and clock cycle), keeping the hardware cost to a minimum. The details of the RAS implementation are explained in the RAS section.

2. Addition unit

The overall structure of the floating-point adder unit is based on dual concurrent pipelines each requiring a shift in only one direction [3,4]. This is illustrated in Figure 1, the addition unit block diagram.

The Path 1 pipeline is used for effective subtraction operations with exponent difference of 0 or 1. Path 2 is used for effective addition and effective subtraction operations with exponent difference greater than 1. The approach described in [3] requires a rounding step in Path 1 for exponent difference of one because of a potential guard bit. However, by considering the MSB of the mantissa, we can avoid this rounding step in Path 1. By requiring the mantissa of the operand with the larger exponent to be less than 1.5, we have a minuend in the range [1, 1.5), and a one bit right shifted subtrahend in the range [0.5, 1), yielding a result that falls in the range (0, 1). This always requires a one bit left shift to normalize the result to the range [1, 2). The possible guard bit would be shifted to the LSB position removing the requirement for a rounding stage. The new selection criteria for Path 1 and Path 2 are described in Table 2.

Path 2 operations generally require a right shift and a rounding step. Addition and subtraction with an exponent difference larger than one require alignment of the smaller operand mantissa using a right shifter. Subtraction with exponent difference of 1 requires a single bit right shift of the mantissa of the smaller operand. Since the subtraction here is limited to a minuend in the range [1.5, 2) and a subtrahend in the range [0, 1), the result falls in the range (0.5, 2). No post-addition right shift is needed and the potential single bit left shift is handled during the first stage of the result selection.

2.1. Path 1

An exponent difference of zero or one can be predicted

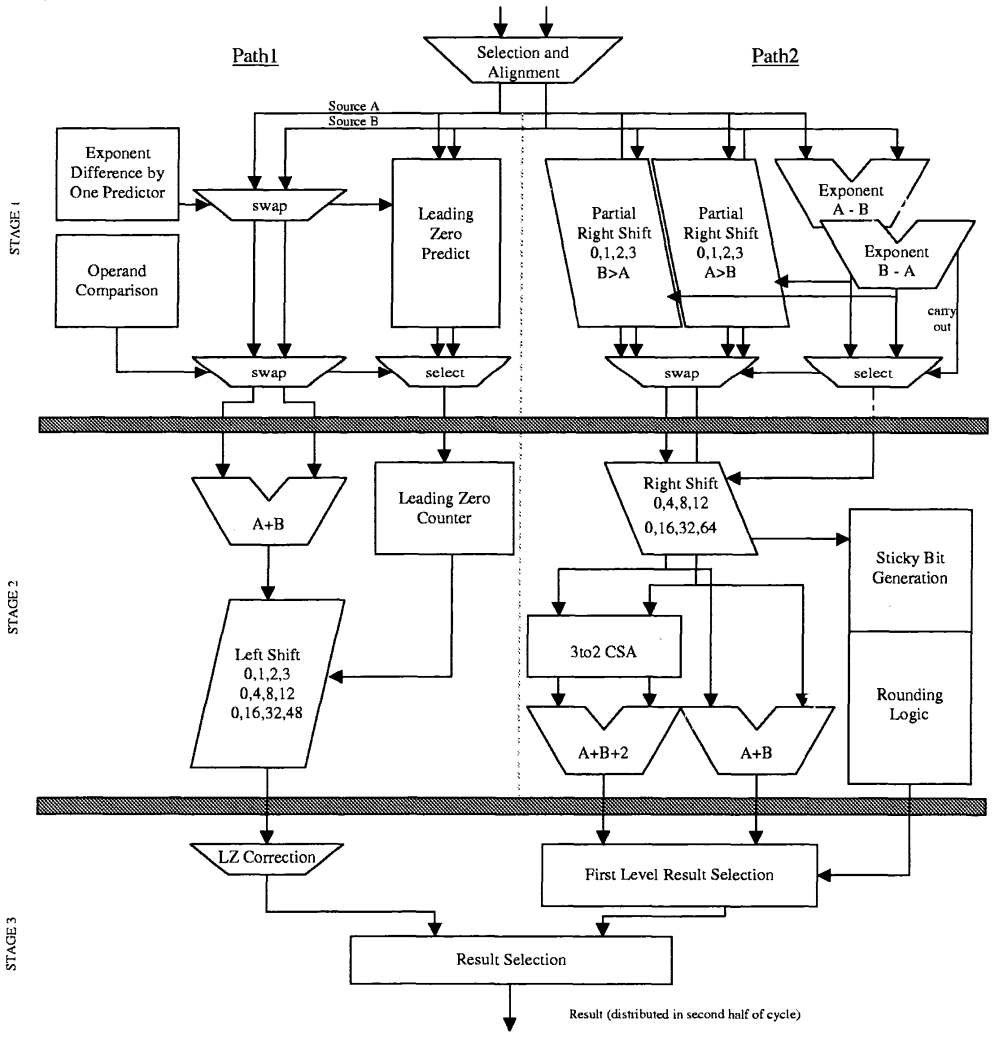

Figure 3. Addition unit

Table 2. Path selection

Path 1	Path 2
Subtraction with exponent difference of 0	Subtraction with exponent difference greater than 1
Subtraction with exponent difference of 1 and Mantissa of larger exponent less than 1.5	Subtraction with exponent difference of 1 and Mantissa of larger exponent greater than or equal to 1.5
	Addition

using the two least significant exponent bits of each operand [4]. This prediction is used to resolve the exponent precedence early in Path 1 so that the mantissa alignment and leading zero prediction [5,6] can both be completed in pipe stage 1.

In order to avoid a negative mantissa result that requires post-addition conversion from sign-magnitude format, the smaller mantissa should always be subtracted from the larger mantissa. For exponent difference of one, the above prediction can be used to indicate the precedence. For an exponent difference of zero, a full mantissa compare is done to determine the larger mantissa, and the mantissas are swapped accordingly.

Operands with an exponent difference of zero or one can generate a mantissa result that contains one or more leading zeros. The algorithm used to predict the leading zeros vector assumes that the smaller mantissa (B = $b_{63}...b_0$) is being subtracted from the larger (A = $a_{63}...a_0$) and implements the twos complement subtraction by inverting B. This precedence is guaranteed as above by using the difference by one prediction and the operand comparison result.

$$z_i = \overline{((a_i \oplus \overline{b_i}) + (\overline{a_{i-1}} \cdot b_{i-1}))}; i = 1 \text{ to } 63 \qquad (1)$$

The above prediction algorithm, which does not use a full carry chain, guarantees that the prediction vector (Z = $z_{63}...z_1$) has, at most, one too few leading zeros. This error is corrected in pipe stage 3 with a one bit left shift of the result.

In pipe stage two, the leading zeros in the prediction vector are counted (LZC) and encoded. In parallel, the mantissa subtraction is computed. The post-subtraction left shift is implemented as a series of muxes driven by the decoded LZC result.

The final step for Path 1 is leading zero error correction, which is handled in pipe stage three. Since only normalized results are supported in Sparc64-V, the leading zero prediction error can be detected by looking at the MSB (hidden bit) of the normalized result. If the MSB is zero, there was a prediction error, and a one bit left shift is required to complete the normalization. This result is sent for final selection between the two paths and driven onto the result bus.

2.2. Path 2

The pre-normalization right shift magnitude is determined from the exponent difference for both addition and subtraction operations in Path 2. Before the precedence of the operands is known, exponent subtractions of A – B and B – A are both performed. The low order results of the exponent subtraction are used to partially right shift both mantissas by 0, 1, 2 or 3. After the exponent subtraction is completed, the partially shifted mantissa of the smaller operand is selected. To avoid post-addition sign magnitude conversion for subtraction, the mantissa with the smaller exponent is swapped into the subtrahend position.

In pipe stage two of Path 2, the full result of the exponent subtraction is used to complete the right shift of the mantissa of the smaller operand. As data is right shifted out of the mantissa, the rounding information is computed in parallel (LSB+1, LSB, guard, round, sticky bit).

For proper IEEE-754 rounding [7], A+B, A+B+1 and A+B+2 are required. This can be achieved using only two adders to generate A+B and A+B+2; A+B+1 can be derived from these two results [3]. A Carry Save Adder (CSA) stage is inserted prior to the A+B+2 adder to add a 1 at the LSB+1 positions for single and double precision operations, bit 41 and bit 12 respectively. This row of CSA's is used in lieu of a flagged adder (as described in [4]) that would require a split carry chain to support insertion of rounding bits at different bit positions. Without a split carry chain, the topology of the adder can be simplified, leading to a faster adder design. Table 3 illustrates the selection of addition results with rounding and normalization based on the overflow bit (MSB + 1), round up bit and LSB. The round up bit indicates the conditional increment of the pre-normalized result to perform IEEE-754 rounding for all rounding modes, and is computed based on the rounding mode, sign, guard, round and sticky bits.

Path 2 subtraction results fall in the range (0.5, 2) which will require a one bit left shift for normalization when the result is less than 1. In the case of a left shift, the rounding position will change from the LSB to the guard bit. The two possible rounding positions can be handled using two *fill* bits at the LSB+1 and LSB positions, combined with A+B and A+B+2 results. For instance, when the LSB and guard bit are both one, a round up will carry through and be reflected by the selection of A+B+2 on the upper bits, zeroing out the LSB bit(s) in the process. Table 4 illustrates in more detail how the *fill* bits are combined with A+B and A+B+2 to generate the final result.

Table 3. Path 2 addition result selection

Overflow	Round Up Bit	LSB	Result (Double Precision)	Result (Single Precision)
1	1	-	A + B + 2[64 : 12]	A + B + 2[64 : 41]
1	0	-	A + B[64 : 12]	A+B[64:41]
0	1	1	A+B+2[63:12], $\overline{A+B[11]}$	A+B+2[63:41], $\overline{A+B[40]}$
0	1	0	A+B[63:12], $\overline{A+B[11]}$	A+B[63:41], $\overline{A+B[40]}$
0	0	-	A+B[63:11]	A+B[63:40]

Table 4. Path 2 subtraction result selection

Round Up	MSB	LSB	Guard	Result (Double Precision)	Result (Single Precision)
0	0	-	-	A + B[62:10]	A + B[62:39]
1	0	0	0	A + B[62:12], 01	A + B[62:41], 01
1	0	0	1	A + B[62:12], 10	A + B[62:41], 10
1	0	1	0	A + B[62:12], 11	A + B[62:41], 11
1	0	1	1	A + B + 2[62:12], 00	A + B + 2[62:41], 00
0	1	-	-	A + B[63:11]	A + B[63:40]
1	1	0	0	A + B[63:12], 0	A + B[63:41], 0
1	1	0	1	A + B[63:12], 1	A + B[63:41], 1
1	1	1	0	A + B[63:12], 1	A + B[63:41], 1
1	1	1	1	A + B + 2[63:12], 0	A + B + 2[63:41], 0

The rounding algorithm used here supports all four IEEE-754 rounding modes. The rounded result selection mentioned above is implemented in pipe stage three followed by selection between the Path 1 and Path 2 results. The remainder of pipe stage three is used for result selection and distribution on the result bus.

2.3. Conversion operations

Overlapped on top of the framework for addition and subtraction are the ancillary floating-point instructions including floating-point to integer, integer to floating-point and floating-point to floating-point conversions.

Single to double precision floating-point and integer to floating-point conversions are handled in Path1. For single to double precision conversions, the left shifter is used for normalization if the incoming operand is denormal. Integer to floating-point conversions may require rounding after the normalization (left) shift to account for the extra bits of precision in the integer operand. Since subtraction does not require a rounding step in Path1, an extra cycle is used to round the floating-point result.

Double to single precision floating-point and floating-point to integer conversions are handled in Path2 since single precision conversions require rounding and floating-point to integer conversions use a right shift to align the fixed-point result based on the floating-point exponent.

3. Multiplier unit

The Multiplier Unit (FMUL) has a four-stage pipeline, which performs floating-point multiplication, division, and square-root instructions. Fixed point SIMD graphics multiply instructions (VIS2.0) are also performed in the FMUL unit. Floating-point multiply instructions are fully pipelined and require three and a half pipe stages (the last half stage is for result distribution). The division and square-root instructions are not pipelined and use multiple iterations through the multiplier array. New instructions cannot be scheduled to the FMUL unit until a division/square-root instruction completes.

The mantissa datapath of the multiplier pipeline is shown in Figure 4. The mantissa array is 60x60 to support division and square-root operations for correct IEEE-754 rounding. The multiplier array is also used to support four 8x16 signed SIMD graphics multiply operations.

Pipe stage 1 of the multiplier pipeline implements the operand alignment function as well as the Booth recoding of the multiplier. For 60-bit operands, both radix-4 and radix-8 Booth recoding require four levels of 4to2 CSA. Radix-4 is chosen over radix-8 recoding to avoid the extra delay in generating the partial products for Booth recode values of +3 and -3. Radix-4 modified Booth recoding [8] generates 31 partial products in 2's complement form to handle negative numbers that need to be sign extended. This sign extension is achieved by adding two extra bits [9], resulting in a 63-bit long partial product vector to

which the Booth sign is added, as shown in equations 2 and 3.

$$PP_0 = \overline{P_s}\, P_s\, P_s\, P_{60}\, P_{59}\, P_{58} \ldots\ldots\ldots P_1\, P_0 \quad (2)$$
$$P_{bs}$$

and for i = 1 to 30

$$PP_i = 1\,\overline{P_s}\, P_{60}\, P_{59}\, P_{58} \ldots\ldots\ldots P_1\, P_0 \quad (3)$$
$$P_{bs}$$

The Booth sign, P_{bs}, is 0 for Booth recode values of 0, 1, or 2, and P_{bs} is 1 for Booth recode values of -1 and -2. For unsigned multiplicand, P_s is 0 for Booth recode values of 0, 1, and 2, and P_s is 1 for Booth recode values of –1 and -2. For signed multiplicand, P_s is P_{59} for Booth recode values of 0, 1, and 2, and $P_s = \overline{P_{58}}$ for Booth recode values of –1 and –2.

In the pipe stage 2, the partial products are reduced using the Wallace tree technique [10] with 4to2 Carry-Save Adders (CSA). Thirty-two summands are formed by combining thirty-one partial products with the addend used for computing the residual for the division and square-root iterations. Each 4to2 CSA row reduces the number of summands by a factor of two. Four levels of 4to2 CSA rows are thus required to compress thirty-two summands to two terms (sum and carry) in redundant form.

In a conventional implementation of a Wallace tree, the outputs of the first three levels of CSA rows need alignment shifts of 8, 16, and 32 bits. Such a staggered array suffers long interconnect delays and increased width of the array. To reduce these problems, the multiplier array is folded as shown in Figure 5.

The multiplicand is routed diagonally and the partial products are aligned such that no shifting is required for the redundant sum and carry outputs of the CSA. Since the critical path is through the multiplier for partial product generation, the shifting on the multiplicand does not affect timing. After the partial products are generated, the critical path is through the CSA rows, and by not shifting the sum and carry outputs, wire delay is minimized. This is an improvement over other implementations, such as [11], where the sum and carry outputs, rather than the multiplicand, are shifted for proper alignment. The outputs from column b_{58} are used by the CSAs of column b_{59}. The routing of the wires from column b_{58} to column b_{59} is avoided by duplicating column's, $b_{58}..b_{52}$, in the lower half of the array. The array is also structured to facilitate the non-redundant result and sticky generation by grouping the upper (result) bits together and the lower (sticky) bits together. This grouping avoids the potential 64-bit shift of the final CSA outputs.

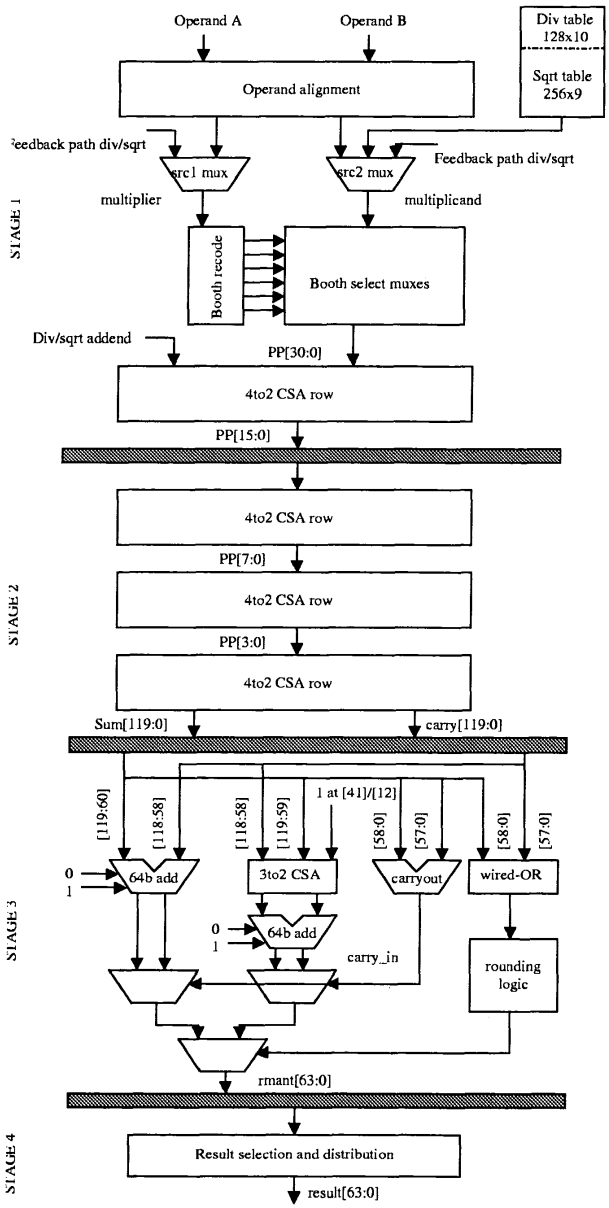

Figure 4. Multiplier unit

In pipe stage 3, the sum and carry vectors in redundant form are added using a group carry look-ahead adder with carry select to obtain the non-redundant sum. The carry-out from the low-order 60 bits is generated by special circuitry that provides the carry-in bit for the adder for the result selection. The sticky bit is generated in parallel to the generation of the carry-out, directly from redundant sum and carry bits, similar to the technique used in [11] as shown in equation 4.

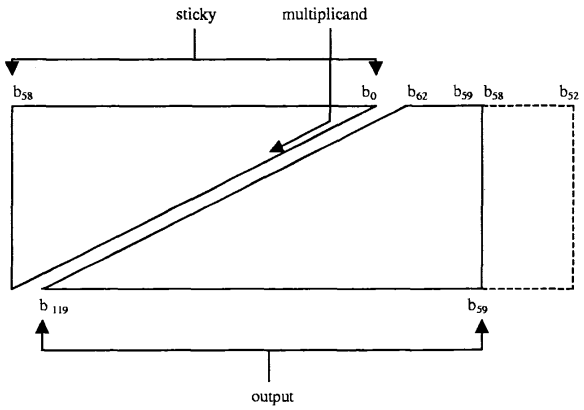

Figure 5. Multiplier array folding

$$p_i = s_i \oplus c_i \; ; \; k_i = \overline{s_i} \cdot \overline{c_i} \; ; z_i = \overline{p_i \oplus k_{i-1}}, \quad \text{for } i = 1 \text{ to } 58 \tag{4}$$

s_i and c_i are the sum and carry bits of the redundant sum and carry vectors. The propagate signal, p_i, and the kill signal, k_i, from the 60-bit carry-out logic are used to generate the vector z_i. The sticky bit is the logical OR of the z_i vector. Several rounding algorithms for IEEE multipliers exist in the literature [11,12]. The rounding algorithm implemented here can be described with reference to Figure 4. Sums with assumed overflow and non-overflow are generated with and without carry-in. Rounding is then performed by two levels of selection, first selection is based on the carry-in bit from the carry-out logic, and the second selection is based on the overflow bit, round bit, sticky bit, and the rounding mode. The selection of the rounded result is similar to the one used in the addition unit, illustrated in Table 3.

In pipe stage 4, the mantissa and exponent paths are properly aligned and merged, and the final result is driven onto the result bus. Partial support for handling denormal operands is provided by the multiplier hardware and is detailed in section 4.

3.1. Division and square-root

The division and square root operations are performed by iterating on the multiplier, and are not pipelined. The scheduler does not dispatch any multiply/division/square root/VIS instructions to the FMUL unit executing division/square root operation until the current division/square-root instruction completes. The division/square-root instruction sends a *done* signal to the scheduler 6 cycles before it is ready to drive the result on the result bus. The divider preempts the distribution bus in the 6th cycle after asserting *done* signal and drives the result distribution bus with the division/square-root result.

The division and square root instructions are implemented using the iterative Goldschmidt convergence algorithm [13,14]. Both operands, A and B, are normalized floating-point numbers with their mantissas in the range [1, 2) and a normalized value of A/B is produced in the range [1, 2) for division, and a normalized value of $\sqrt{B}$ in the range [1, 2) is produced by square root. For division, the seed value is an initial approximation of 1/B, obtained by indexing the leading 7 bits of B (excluding the implicit bit) into a 7-bits-in, 10-bits-out look-up table. Let F_0 be the result from the seed table. The iteration equations are shown in equations 5 and 6:

$$Q_1 = \lfloor A \cdot F_0 \rfloor, \; G_1 = \lceil B \cdot F_0 \rceil \tag{5}$$

$$F_i = \lfloor 2 - G_i \rfloor, \; Q_{i+1} = \lfloor Q_i \cdot F_i \rfloor, \; G_{i+1} = \lceil G_i \cdot F_i \rceil, \quad \text{for } i = 1, 2, \text{ and } 3. \tag{6}$$

For square-root, the seed value is an initial approximation of $1/\sqrt{B}$ or $1/\sqrt{2B}$, obtained by indexing into one of two separate lookup tables, one for the even exponent, and the other for the odd exponent of B. Each table has 128 entries, each entry being 9 bits wide. Let F_0 be the result from the seed table. The iteration equations are shown in equations 7 and 8:

$$Q_1 = \lfloor B \cdot F_0 \rfloor, \; G_1 = \lceil (B \cdot F_0) \cdot F_0 \rceil \tag{7}$$

$$F_i = \lfloor \tfrac{1}{2} \cdot (3 - G_i) \rfloor, \; Q_{i+1} = \lfloor Q_i \cdot F_i \rfloor, \; G_{i+1} = \lceil (G_i \cdot F_i) \cdot F_i \rceil, \quad \text{for } i = 1, 2, \text{ and } 3 \tag{8}$$

The lookup tables for division and square root are implemented to minimize their sizes for optimal double precision operations and to allow the table access to occur in one cycle. Bipartite seed table implementation for the same initial accuracy was investigated to be slower than the present single lookup table access. Four 10-bit tables and a 14-bit adder would be required to reduce the number of iterations through the multiplier by one. The area penalty for such an implementation far exceeded the potential improvement in latencies.

The seed values are always chosen to be less than the corresponding infinitely precise values, minimizing the maximum relative error in the interval [1,2) [15]. In the division and square-root iterations, each multiplication is 60 by 60 bits, where $\lfloor\rfloor$ and $\lceil\rceil$ represent round down and round up values of the 120-bit multiplication result to 60 bits, where the round up is approximated by unconditionally adding a one in LSB (bit 60). Each of the intermediate products is rounded to a higher internal precision (60 bits) to account for the accumulated rounding errors in the intermediate iterations. The iterative refinement factors, $2 - G_i$ and $\frac{1}{2} \cdot (3 - G_i)$ are implemented by complementing G_i and adding the constants (1 for division and 3/2 for square root) in the unused slots of the

Table 5. Single precision division pipeline

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Initial Seed	F_0															
1st Iteration		G_1	G_1	G_1												
			Q_1	Q_1	Q_1											
2nd Iteration					G_2	G_2	G_2									
						Q_2	Q_2	Q_2								
3rd Iteration									Q_3	Q_3	Q_3					
												-				
Residual and Result													C	C	C	Res

CSA rows. With the design parameters stated above, Q_3 has enough bits of accuracy to yield the correctly rounded single precision result. For double precision operations, Q_4 needs to be computed to yield the correctly rounded result. The iterations through the multiplier array use only the first three stages of the multiplier pipeline. The fourth stage is only used for the final IEEE rounding and result distribution.

Rounding is performed after the final iteration of the convergence algorithm on the 120-bit raw result Q (Q_3 for single precision, Q_4 for double precision) to produce the IEEE compliant result. The rounding algorithm used is similar to the methods used in [16,17]. The residual is first computed as $C = A - Q_c * B$ for division, and $C = B - Q_c^2$ for square root in one pass through the multiplier pipe, where Q_c is obtained by rounding up Q to ½ ulp (25 bits for single precision, and 54 bits for double precision). The correct result is then selected from *trunc*(Q), *trunc*(Q + ½ ulp), and *trunc*(Q + 1 ulp), where *trunc(x)* is the round-down value of x to 24 bits for single precision and 53 bits for double precision, and ulp = 2^{-23} for single precision, and ulp = 2^{-52} for double precision. The above three results are generated for the multiplier rounding and thus there is no extra penalty for division and square root rounding. The selection of the appropriate result is based on the round bit and LSB of Q, the residual C (there are three cases C>0, C=0, C<0), the sign of the expected result, and the rounding mode.

Single and double precision division have latencies of 16 and 19 cycles respectively, and single and double precision square root have latencies of 22 and 27 cycles respectively. The throughputs are three cycles shorter than the latency because a subsequent operation can enter the multiply pipe once the last iteration of the division or square root has moved to pipe stage 2. The latency of single precision division iterating on the multiplier pipeline is explained in Table 5. There is one cycle bubble in the pipeline between computation of Q_3 and the computation of residual C to generate the three possible results for subsequent selection for rounding. The latencies of double precision division and single and double precision square root can be explained similarly.

Correctness of the division and square-root implementations has been verified by deriving the upper bound on the maximum error. The two sources of error are the approximation error due to the algorithm itself, and the computational error due to finite table values and finite multiplier size. These errors are computed and accumulated over the required number of iterations. The initial table size and the seed values, size of the multiplier, and the intermediate directed rounding guarantee that the 120-bit raw result, Q, satisfies the bound, $q - ½ \text{ ulp} < Q < q$ for $q \geq 1$, and $q - ¼ \text{ ulp} < Q < q$ for $q < 1$ for division, and $q - ¼ \text{ ulp} + \text{ulp}^2 < Q < q$ for square-root, where ulp = 2^{-23} for single precision, and ulp = 2^{-52} for double precision, and q is the infinitely precise result. This bound is sufficient to yield correctly rounded result in all rounding modes.

For special operands such as Zero, QNAN, SNAN, Infinity etc., the division and square-root operations do not iterate through the multiplier. Results for these cases are generated by special control logic in the first pass through the multiplier. This early exit feature allows the latency of division and square-root operations to be only 10 cycles for special operands.

4. Denormal operations

In general, floating-point operations will encounter denormal operands and operands that will result in denormal results. Generating IEEE-754 compliant results for denormal operands and results often requires the determination of leading zeros, adjustment to mantissas (shifting) and exponent, and special rounding. Overall, timing and area are adversely affected when denormal operations are fully supported in hardware. However, by introducing a minimal amount of hardware to compare the value of the exponents, we can provide support for a large subset of the denormal operations.

Consider the case of floating-point multiplication. If both operands are denormal, the result is guaranteed to underflow beyond the dynamic range irrespective of the number of leading zeros in the mantissa bits. If one of the operands is a denormal, and the resulting exponent (esrc1

Table 6. Denormal handling

Instruction(s)	1 Denormal Operand, 1 Normal Operand	2 Denormal Operands	Result Denormal
FADD{s,d}, FSUB{s,d}	Unfinished_FPop	Unfinished trap	Unfinished_FPop, if eres < 1
FMUL{s,d}	Unfinished_FPop, if -25 < (esrc1 + esrc2 - 126) (sp) -54 < (esrc1 + esrc2 - 1022) (dp)	Handled in hardware	Unfinished_FPop, if -25 < eres < 1 (sp) -54 < eres < 1 (dp)
FDIV{s,d}	Normal/Denormal: Unfinished_FPop, if (esrc1 – esrc2 – 1) < 128 (sp) (esrc1 – esrc2 – 1) < 1024 (dp) Denormal/Normal: Unfinished_FPop, if -25 < (esrc1 - esrc2 + 126) (sp) -54 < (esrc1 - esrc2 + 1022) (dp)	Unfinished FPop	Unfinished_FPop, if -25 < eres < 1 (sp) -54 < eres < 1 (dp)
FSQRT{s,d}	Unfinished_FPop for positive operand		

+ esrc2 – bias) is less than –25 for single precision or –54 for double precision, the result is guaranteed to underflow beyond the dynamic range in the given precision, irrespective of the number of leading zeros in mantissa bits of the denormal operand. The two cases where the rounding will produce the smallest denormal number are rounding to positive infinity and the result is positive, and rounding to negative infinity and the result is negative. All other rounding modes produce zero as a result. Such extreme underflow cases are efficiently handled by hardware. All other cases that are not handled by hardware are conservatively trapped as unfinished_FPop. Table 6 shows how denormals are handled for different operations, where the biased exponents of the source operands, and the result are denoted by esrc1, esrc2, and eres. The instructions with denormal operands/results are trapped by setting unfinished_FPop exception. The software emulates the trapping instruction and generates the correct results and flags.

5. RAS

The Floating-Point Unit supports RAS features without sacrificing significant area or performance. Data buses are parity protected on the word boundary (32-bits) instead of byte (8-bits) to reduce the number of parity bits distributed and stored. Since parity generation of 32-bits takes five levels of logic to implement, the parity distribution is done one cycle after the data is distributed. Soft errors in static circuits are recoverable most of the time. They cannot be recovered if the soft error propagates into a storage element such as a latch or a flip-flop. Since the soft errors are usually low swing, the propagation is two to three times slower than normal delay. Unless the soft error occurs on a gate very close to the storage gate, they are recoverable. Soft errors in dynamic circuits and storage elements such as latches are non-recoverable and their error rates are shown in Table 7. By providing RAS protection for dynamic circuits and all registers that hold state for more than one cycle, most of the soft errors can be detected.

Table 7. Circuit component soft error rates

Circuit Type	Soft Error Rate (fit)
Latch	0.00052
Dynamic circuit	0.0013

The FRS registers holds data for more than one cycle so it is parity protected. The parity is written to the FRS one cycle after the data is written. Similarly, the parity is read one cycle after the data is read. When the FRS is read, the parity is checked and errors, if any, are logged.

The parity prediction algorithms for the arithmetic functions are expensive both in terms of area and speed. Since 80% of the FADD unit uses dynamic circuits that need to be protected, a different approach is used. By implementing a dual-rail domino design, the true and complement signals at the last cone of logic can be XORed. An error is indicated if the XOR output is zero. This approach is used to detect errors on some of the major blocks (align, compare, add) of the FADD unit. Eighty percent of the domino logic is covered in this manner. An example of dual-rail domino with error check is shown in Figure 6. The example shows a dual-rail domino AND gate, but this concept can be generally applied to any dual-rail domino gate. In the case of an adder, the final carry and carry-complement can be

XORed as long as carry and carry-complement logic are independent.

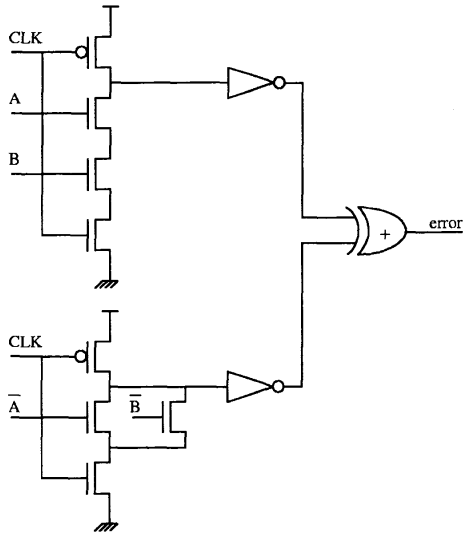

Figure 6. Dual-rail domino error detection

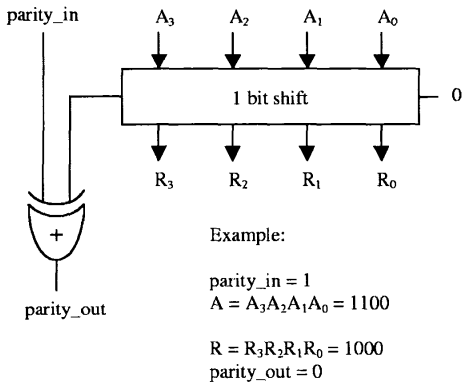

Figure 7. Parity prediction for 1 bit left shift function

In general, multiplier arrays are protected with residue checking because residue arithmetic takes less area. However, residue generation timing is more critical than the critical timing of the array itself. So for Sparc64-V, the multiplier array (pipe stages 1 and 2) is designed with static logic and RAS protection is not provided. For the next two pipe stages of the FMUL unit, single rail dynamic circuits are used and RAS protection is provided by parity prediction approach. The parity prediction lags the data by one-half cycle. Parity prediction was used because dual-rail domino was not used in these pipe stages (except adder) and there are registers that hold data for more than one cycle to support divide and square-root operations. An example of parity prediction for a 1 bit left shift operation is shown in Figure 7. The example illustrates how the parity value is affected when a zero is left shifted into the given input (A).

The VIS unit is implemented with static logic so no RAS protection is provided. The FPU RAS error reporting is synchronized with the floating-point exception reporting.

6. Summary

We have presented the framework for a reliable, high performance IEEE-754 compliant Floating Point Unit with support for VIS instructions. Novel techniques presented in the Floating Point Add algorithm eliminate rounding in Path 1 thereby reducing area and enabling us to meet our 1GHz timing goal. Enhancements to the addition algorithm also result in eliminating the need for post addition data conversion in both paths. Use of the addition and addition + 2 results yield efficient rounding for add, subtract and multiply instructions. The multiplier folding technique presented results in a well-balanced Wallace tree, minimizing area and wire delay. The bypass path in the multiplier for divide and square-root iterations reduces the latency of these instructions by 3 and 5 cycles, respectively. RAS features provided in the floating-point unit make the soft errors failures due to alpha particles detectable and to some extend correctable. Partial support for denormals and early exit feature for special operands in the case of divide and square-root operations add to the overall performance of the FPU at minimal hardware cost and no impact to cycle time.

Table 8 highlights the salient features of the technology used and some relevant statistics gathered from the completed design. Figure 8 shows the routed layout of the Floating Point Unit.

Table 8. FPU design characteristics

Methodology	Full Custom
Process	0.15u, 6 layer Metal CMOS
Transistor Count	1.9 Million
Transistor Density	87 KT/mm^2
FPU Footprint	9.1 x 2.4 mm^2
Cycle Time	1ns @ 1.5V, 85 °C

7. Acknowledgments

The authors would like to acknowledge Dr. Ping Tak Peter Tang for completing the error analysis of the divide and square-root algorithms for the Hal specific implementation and for providing valuable feedback as to

the size of the seed tables for optimal implementation. Acknowledgments are also due to Dr. Allen Lyu for microarchitecting the divide and square root implementations, the design team (James Vinh, Pranjal Srivastava, Jeremy Minor, Sathyanath Subramanian), and D.S. Radhakrishnan for verifying the design.

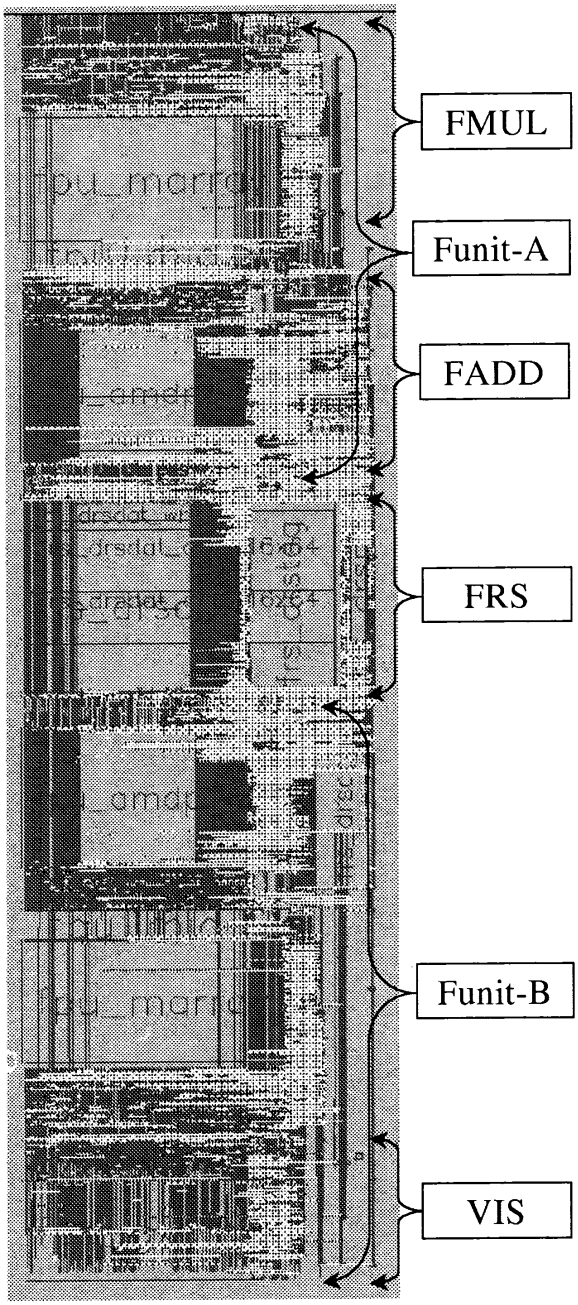

Figure 8. Layout

8. References

[1] "The SPARC Architecture Manual Version ", SPARC International, Inc., Menlo Park, CA 1994.

[2] "Visual Instruction Set Users Guide", Sun Microelectronics, Sun Microsystems, 1995.

[3] N. T. Quach and M. J. Flynn,"An Improved Algorithm for High-Speed Floating-Point Addition" Technical Report CSL-TR-90-442 Stanford University, August 1990.

[4] A. Beaumont-Smith, N. Burgess, S. Lefrere and C.C. Lim, "Reduced Latency IEEE Floating-Point Standard Adder Architectures," in *Proc. 14th Symp. on Computer Arithmetic,* pp 35-42, 1999.

[5] E. Hokenek, R. K. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit", in *IBM Journal of Research and Development*, pp 71-77, Jan. 1990.

[6] Suzuki et. al., "Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition" in *Journal of Solid State Circuits,* pp 1157-1164, Aug. 1996.

[7] "IEEE Standard for Binary Floating-Point Arithmetic", IEEE Standard 754-1985, IEEE Computer Society, New York 1985.

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

[9] I. Koren, "Computer Arithmetic Algorithms," Prentice Hall, 1993.

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

[11] R. K. Yu and G. B. Zyner, "167 MHz radix-4 floating point multiplier," in *Proc. 12th Symp. Comput. Arithmetic,* pp 149-154, 1995.

[12] M. R. Santoro, G. Bewick and M. A. Horowitz, "Rounding Algorithms for IEEE Multipliers," in *Proc. 9th Symp. Computer Arithmetic,* pp 176-183, 1989.

[13] R. E. Goldschmidt, "Applications of division by convergence," in *M. S. thesis,* Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1964.

[14] C. V. Ramamoorthy, J. R. Goodman and K. H. Kim, "Some properties of iterative square-rooting methods using high speed multiplication," in *IEEE Trans. Comput.*, C-21(8), pp 837-847, Aug. 1972.

[15] D DasSarma and David W. Matula, "Measuring the accuracy of ROM reciprocal tables," in *Proc. 11th Symp. Computer Arithmetic,* pp 95-102, 1993.

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

[17] Stuart Oberman, "Division Algorithms and Implementations" in *IEEE Transactions on Computers Vol. 46, No. 8,* pp 853-854, Aug. 1997.