Journal of Universal Computer Science, vol. 4, no. 2 (1998), 99-109
submitted: 25/9/97, accepted: 1/11/97, appeared: 28/2/98 © Springer Pub. Co.

The SNAP Project : Building Validated Floating Point Units

Hesham A. Al-twaijry
(Stuart F. Oberman
Steve T. Fu
Michael J. Flynn)
(Computer Systems Laboratory, Stanford University
Stanford, CA 94305
hesham@umunhum.stanford.edu, oberman@umunhum.stanford.edu,
fu@umunhum.stanford.edu, flynn@umunhum.stanford.edu)

Abstract: SNAP - The Stanford Sub-nanosecond arithmetic processor is an interdisciplinary effort to develop validated theory, and tools for realizing an arithmetic processor with execution rates under 1ns. The project has targeted the full spectrum of tradeoffs from algorithms, circuit optimizations, system issues, and development of metrics to characterize processors.

Key Words: Computer arithmetic, Validated Designs, Addition, Multiplication, Division

1 Introduction

The SNAP project has looked into many areas of the design of an arithmetic processor with the aim of producing validated designs that span the spectrum of operand lengths. In the area of algorithm improvement, SNAP work has introduced a new variable latency algorithm (VLA) for floating point addition, that can produce a result in one cycle 32% of the time. In floating point multiplication an algorithm for designing Wallace trees was developed. This algorithm is shown to be superior to binary trees.

With the realization that performance of an arithmetic processor is not simply dependent on the most advanced circuit techniques and algorithms, SNAP has addressed system issues by looking at the help that a compiler can provide. It has shown that divide though infrequent can have a big performance effect. With the best compiler technology a division latency of 10 cycles can be tolerated.

The increased advances in integrated circuit fabrication technology have resulted in integrated circuit fabrication technology have resulted in both smaller feature sizes and increased die areas. Together, these trends have provided a larger transistor buget for the processor designer, Therefore, it has become possible for the designer to implement more sophisticated arithmetic processors in hardware. Therefore, metrics that allow FPU designers to gauge their designs are of upmost importance. The SNAP project has addressed this problem by developing FUPA (**F**loating Point **U**nit Cost **P**erformance **A**nalysis Metric) that measures the efficiency of FP unit in terms of latency $\times$ area normalized to feature sizes.

2 Floating Point Addition

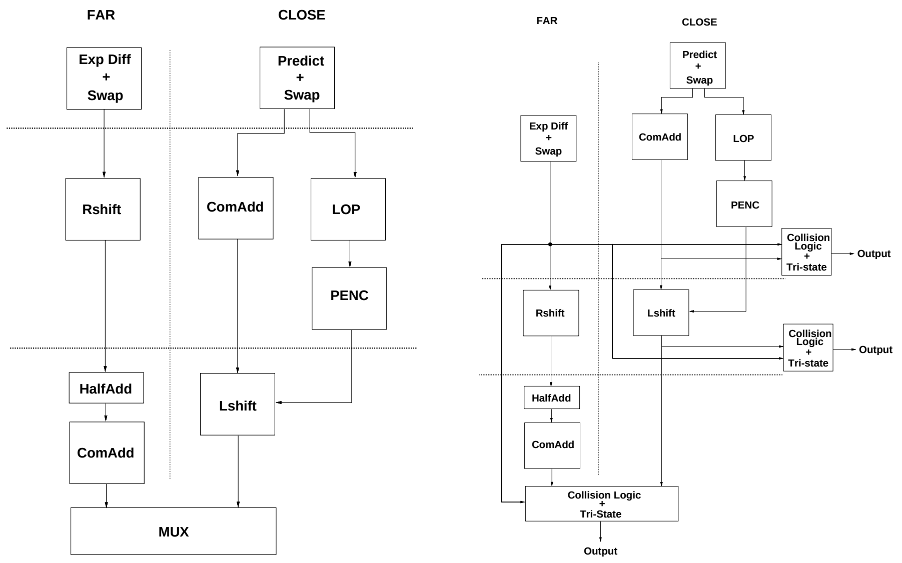
The most frequent FP operations are addition and subtraction, and together they account for over half of the total FP operations in typical scientific applications.

To reduce the latency, we observe that not all of the components are needed for all input operands. Two VLA techniques are proposed to take advantage of this to reduce the average addition latency. To effectively use average latency, the processor must be able to exploit a variable latency functional unit.

2.1 Current Algorithms

FP addition comprises several individual operations. Higher performance is achieved by reducing the maximum number of serial operations in the critical path of the algorithm.

A block diagram of a state-of-the-art FP adder is shown in figure 1a.



(a) Three Cycle Pipelined Adder (b) Variable Latency Adder

Figure 1: Adders

This architecture exploits many aspects of the FP addition dataflow. It implements the significand datapath in two parts: the *CLOSE* path and *FAR* path.For subtraction, when the exponents differ by more than 1 (FAR path), massive cancellation can not occur. Rather, there can be at most a 1 bit left-shift. Similarly, when the exponents differ by at most 1 (CLOSE path), massive cancellation may occur requiring a large normalizing left-shift, but no initial large aligning right shift is required. This allows the aligning right shift and the normalizing left-shift to be mutually exclusive, with only one such shift ever appearing on the critical path [5].

Another optimization made in this algorithm reduces the number of serial operations. In a straightforward implementation of the addition dataflow, rounding would be implemented by a separate series incrementer after all other operations. However, the realization can be made that the rounding step occurs very late in the computation, and it only modifies the result by a small amount. By precomputing all possible required results in advance, rounding and conversion can be reduced to the selection of the correct result [S4]. For the IEEE *round to nearest* (RN) rounding mode, the computation of $A + B$ and $A + B + 1$ is sufficient to account for all possible rounding and conversion possibilities. Incorporating this optimization into the algorithm requires that the significand adders in each path compute both *sum* and *sum+1*, typically through the use of a compound adder (ComAdd). Selection of the true result is accomplished by analyzing the rounding bits, and then selecting either of the two results. This optimization removes one significand addition step.

Assuming that the mantissas are conditionally swapped based upon the true exponent difference, the smaller mantissa is always subtracted from the larger mantissa, except possibly in the $CLOSE$ path for cases where the exponents are equal. However, in these cases, since there is no initial aligning right shift, the result is exact and no rounding is required. Further, by again precomputing both *sum* and *sum+1* in the significand adder, recomplementation can also be reduced to selection. The true subtraction of $A - B$ is accomplished by selecting *sum+1*, as the subtraction is implemented by $A + \overline{B} + 1$. If the carry-out of this addition is 0, then the result is negative requiring recomplementation. The complemented result is formed by bitwise inversion of *sum*, as

$$-(A - B) = \overline{A + \overline{B}}$$

Accordingly, recomplementation is reduced to a MUX and bitwise inversion.

Further performance improvement is achieved by computing the normalizing left-shift distance in the $CLOSE$ path in parallel with the compound adder, rather than in series, using leading-one-prediction (LOP) and priority-encoding (PENC). An adder employing all of these optimizations in a high clock-rate microprocessor typically has a latency of three cycles. The critical path in this implementation is in the third stage consisting of the delays of the half-adder, compound adder, multiplexor, and drivers.

2.2 Variable Latency Algorithm

From figure 1a, the long latency operation in the first cycle occurs in the FAR path. It contains hardware to compute the absolute difference of two exponents and to conditionally swap the mantissas. For IEEE double precision operands, the minimum latency in this path comprises the delay of an 11 bit adder and two multiplexors. The $CLOSE$ path, in contrast, has relatively little computation. A few gates are required to inspect the low-order 2 bits of the exponents to determine whether or not to swap the mantissas, and a multiplexor is required to perform the swap.

Rather than letting the $CLOSE$ path hardware sit idle during the first cycle, it is possible to take advantage of the duplicated hardware and initiate $CLOSE$ path computation one cycle earlier. This is accomplished by moving both the second and third stage $CLOSE$ path hardware up to their preceding stages. Since the first stage in the *CLOSE* path completes very early relative to the *FAR* path, the addition of the second stage hardware need not result in an increase in cycle time.

The operation of the proposed algorithm is as follows. Both paths begin speculative execution in the first cycle. At the end of the first cycle, the true exponent difference is known from the *FAR* path. If the exponent difference dictates that the *FAR* path is the correct path, then computation continues in that path for two more cycles, for a total latency of three cycles. However, if the *CLOSE* path is chosen, then computation continues for one more cycle, with the result available after a total of two cycles.

Further reductions in the latency of the *CLOSE* path can be made after certain observations. First, the normalizing left shift in the second cycle is not required for all operations. Second, in the case of effective subtractions, small normalizing shifts, such as those of two bits or less, can be separated from longer shifts. Both of these cases have a latency of only one cycle, with little or no impact on cycle time. A block diagram of the variable latency adder is shown in figure 1b.

2.3 Performance

This algorithm was simulated using operands from actual applications to determine its effectiveness. The data for the study was acquired using the ATOM instrumentation system [6]. ATOM was used to instrument 10 applications from the SPECfp92 [7] benchmark suite which were then executed on a DEC Alpha 3000/500 workstation. The benchmarks used the standard input data sets. All double precision floating point addition and subtraction operations were instrumented. The operands from each operation were used as input to a custom FP adder simulator. The simulator recorded the effective operation, exponent difference, and normalizing distance for each set of operands.

The results show that 57% of the operations are in the *FAR* path and require three cycles, while 43% are in the *CLOSE* path and require at most two cycles. A comparison with a different study of floating point addition operands [8] on a much different architecture using different applications provides validation for these results. In that study over 30 years ago, six problems were traced on an IBM 704, tracking the aligning and normalizing shift distances. There 45% of the operands required aligning right shifts of 0 or 1 bit, while 55% required more than a 1 bit right shift. The similarity in the results suggests a fundamental distribution of floating point addition operands in scientific applications.

An analysis of the effective operations in the *CLOSE* path shows that the total of 43% can be broken down into 20% effective addition and 23% effective subtraction. A left shift less than or equal to 2 bits is required for 52.5% of the *CLOSE* path subtractions. In total, $20\% + (0.525) \times 23\% = 32\%$ of the operations can complete in the first cycle. The performance of the proposed techniques is summarized in table 1.

For each technique, the average latency is shown, along with the speedup provided over the base *Two Path* FP adder with a fixed latency of three cycles. By allowing effective additions in the *CLOSE* path to complete in the first cycle (adds), a speedup of 1.27 is achieved. For even higher performance, the most aggressive implementation (subs2) achieves a speedup of 1.33 by allowing all effective addition and those effective subtractions requiring normalizing shifts

Algorithm	Average Latency	Speed Up
Two Path	3.00	1.00
Two Cycle	2.57	1.17
Adds	2.37	1.27
Subs0	2.36	1.27
Subs1	2.31	1.30
Subs2	2.25	1.33

Table 1: FP addition Performance

of two bits or less to complete in the first cycle. These techniques do not add significant hardware, nor do they impact cycle time. They demonstrate how a VLA architecture can provide a reduction in average latency while maintaining single cycle throughput.

3 Multiplication

3.1 Background

Multiplication is the process of adding the partial products. Multiplication algorithms differ in how they generate the partial products and how the partial products are added together to produce the final result.

Research on multiplier design has included techniques for partial product generation [9] and partial product reduction [10], [11], [12], [13], [14]. Most previous analyses of the partial product reduction trees use as the basis for their design a simple compressor delay model where the delay from each input of a compressor to each output is equal. Also, the delay due to interconnection is typically ignored. Unfortunately, such simple models do not accurately reflect the performance of actual implementations where not all inputs have the same delay and where the added delay due to interconnect is significant, especially for minimum feature sizes below 0.5μm. However, a simple delay model is sufficient for the design of a binary tree using 4-2 compressors, as the delay for all inputs of a 4-2 compressor are approximately equal.

Designing an optimized partial product array using (3,2) counters requires taking into account all delay components. Further, organizing the counters in order to minimize worst-case delay is not trivial. Therefore, an algorithmic approach to the design, using a sophisticated delay model that takes into account the interconnect delay due to counter placement and the different path delays, is extremely useful. We have implemented such an algorithm, based upon the approach of Oklobdzija [15]. The algorithm takes into account interconnect delay due to counter placement and the different path delays. Our algorithm uses a complex delay model for the (3,2) counter, and it is further constrained by the availability of wiring tracks for the routing of each column of the partial product array [S5]. The number of wiring tracks available in a column is a function of the fabrication process and the floorplan of the multiplier. It is a fixed parameter for each column, and it limits the possible interconnections.

3.2 Methodology

In this study, we examined multiplier performance and area tradeoffs over combinations of several parameters: feature size (f=1.0μm to 0.2μm), counter configuration (3,2 and "4-2"), encoding scheme (non-Booth, Booth 2, and Booth 3), and significand precision (24b through 113b). For each category, we implemented a custom layout of a binary-tree multiplier using the MAGIC layout tool. Additionally, a unique (3,2) array was designed for every combination of feature size, encoding scheme, and significand precision. Using extracted parasitics, we performed SPICE timing simulations for each combination of parameters. Each simulation included delays due to transistors as well as interconnect. The scalable SPICE model of McFarland [S6] was used to project results down to 0.2μm.

Significand Length (bits)	Encoding Scheme		
	Non-Booth	Booth 2	Booth3
Single (24)	0.85	0.85	0.85
Double (53)	0.88	0.85	0.85
Extended (64)	0.96	0.82	0.88
Extended+4 (68)	0.79	0.77	0.88
Quad (113)	0.95	0.90	0.86

Table 2: Relative delay of Algorithmic Reduction to Binary tree for 0.3μm

Table 2 presents performance for several common significand precisions and possible encoding schemes for a 0.3μm process. In this table, the delays are for the algorithmic array relative to those of the binary tree. The results show that an algorithmically-designed array usually results in a lower latency than does the binary tree.

Significand Length (bits)	PP Reduction Method			
	Algorithmic		Binary Tree	
	Non-Booth	Booth 3	Non-Booth	Booth 3
Single (24)	1	1.15	0.98	1.12
Double (53)	1.18	1.14	1.14	1.15
Extended (64)	1.25	1.12	1.07	1.04
Extended+4 (68)	1.22	1.16	1.19	1.02
Quad (113)	1.23	1.13	1.18	1.19

Table 3: Relative latency of encoding scheme to Booth 2 for 0.3μm

Table 3 summarizes the performance of the different encoding schemes relative to the performance of Booth 2 for a $0.3\mu m$ process. From this table, as the length of the significand increases, Booth 2 becomes the choice which minimizes latency. In most of the cases, the reduction in the number of summands achieved when moving from Booth 2 to Booth 3 encoding is not large enough to offset the extra delay needed to generate the hard (3x) multiple required for Booth 3.

Significand Length (bits)	PP Reduction Method			
	Algorithmic		Binary Tree	
	Non-Booth	Booth 3	Non-Booth	Booth 3
Single (24)	1.02	1.11	0.99	1.15
Double (53)	1.50	0.99	1.35	1.02
Extended (64)	1.63	0.96	1.31	0.92
Extended+4 (68)	1.60	0.97	1.45	0.90
Quad (113)	1.73	0.95	1.54	1.04

Table 4: Relative latency $\times$ area product of encoding scheme to Booth 2 for $0.3\mu m$

Not all multiplier implementations require minimum latency. For these cases, an optimized design balances both latency and area. Table 4 summarizes the choice of encoding scheme which minimizes the latency $\times$ area product.

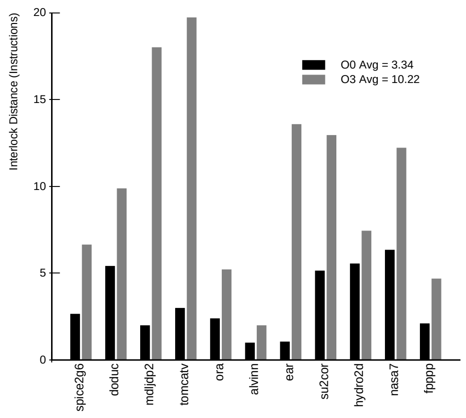
For single precision, both the latency and area of non-Booth and Booth 2 encoding are approximately the same. As a result, the delay $\times$ area product is the same for both. Non-Booth encoding is recommended in this case due to its simplicity of implementation. For other precisions, Booth 3 encoded multipliers are 10-15% smaller and 5-20% slower than Booth 2 encoded multipliers. Accordingly, if area is of primary concern, Booth 3 encoding is recommended for these

4 Floating Point Division

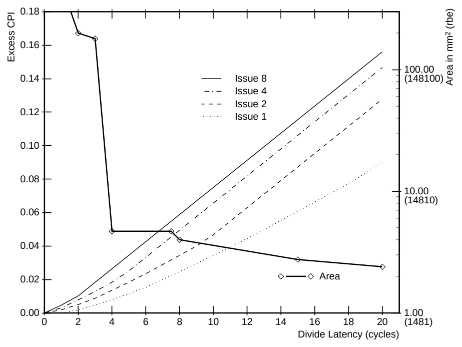
The emphasis in recent FPUs has been in designing ever-faster adders and multipliers, with division receiving less attention. Current applications and benchmarks are often written assuming that division is an inherently slow operation and should be used sparingly. While division is an infrequent operation even in floating point intensive applications, ignoring its implementation can result in system performance degradation. Choosing an optimal FP divider design in terms of performance and area is difficult, as the design space of FP dividers is large, comprising five different classes of division algorithms: digit recurrence, functional iteration, very high radix, table look-up, and variable latency [S7]. This section investigates the performance requirements of FP division and proposes several techniques for achieving them through a combination of FL and VLA techniques.

We have investigated in detail the relationship between FP division latency and system performance [S2]. System performance was evaluated using 11 applications from the SPECfp92 benchmark suite. The applications were each compiled on a DECstation 5000 using the MIPS C and Fortran compilers at O3 optimization.

In order to analyze the impact that the compiler can have on improving system performance, we measured the interlock distances of division results as a function of compiler optimization level. Figure 2a shows the average interlock distances for all of the applications at both O0 and O3 levels of optimization. By intelligent scheduling and loop unrolling, the compiler is able to expose instruction-level parallelism in the applications, increasing the interlock distances. Figure 2a shows that the average interlock distance can be increased by a factor of three by compiler optimization to over 10 instructions. Accordingly, for scalar processors, a division latency of 10 cycles or less can be tolerated.



(a) Interlock Distances



(b) CPI and area vs division latency

Figure 2: Divider

To determine the effects of division latency on overall system performance, the performance degradation due to division was determined. This degradation is expressed in terms of excess CPI, or the CPI due to the result interlock. The performance degradation due to division latency between 1 and 20 cycles is displayed in figure 2b.

In this figure, designs above 8 cycles are SRT implementations, the design between 4 and 8 cycles is a self-timed SRT design, and those designs below 4 cycles are very-high radix designs requiring large initial approximation tables.

Figure 2b also shows the effect of increasing the number of instructions issued per cycle on excess CPI due to division. To determine the effect of varying instruction issue rate on excess CPI due to division, a model of an underlying architecture must be assumed. In this study, an optimal superscalar processor is assumed, such that the maximum issue rate is sustainable. The issue rate is then used to appropriately reduce the interlock distances. Figure 2b also shows how area increases as the functional unit latency decreases. The estimation of area is based on several reported layouts, all of which have been normalized to

1.0μm scalable CMOS layout rules.

5 Floating Point Characterization

The emergence of metrics that allow floating point unit (FPU) designers to gauge their FPU designs is long overdue. The allocation of die area to FPUs remains an art based on engineering intuition and past experience. We present the **F**loating Point **U**nit Cost **P**erformance **A**nalysis Metric($FUPA$) to allow quantitative tradeoffs between performance and cost.

FPU design requires the underlying technology to meet the computation and communication complexity of the algorithm. From a cost perspective, the designer floorplans the available die area and divides the power budget by considering the performance benefit of allocating more die area to a specific operation. $FUPA$ integrates both cost and performance into simple formula for determining the optimality of FPU design.

We summarize the computation of FUPA as:

1. **Profile the applications** to obtain dynamic floating point operation (add-sub, multiply, and divide) distribution the application.
2. **Compute Effective Latency** (EL) from the clock rate, FPU latencies, and the dynamic FP operation distribution obtain in step 1.
3. **Measure the die area** ($Area$) of the FPU not including the register file.
4. **Compute Normalized Effective Latency** (NEL) **and Normalized Area** ($NArea$), removing the feature size dependency.
5. **Compute** $FUPA$ where $FUPA = \frac{(NEL)(NArea)}{100}$.

Processor	Effective Latency(ns)	Normalized Area(mm^2)	Normalized Effective Latency(ns)	FUPA (cm^2ns)
Intel P6	37.67	50.62	75.33	38.13
MIPS R10000	14.25	44.07	28.50	12.56
SUN UltraSparc	23.65	133.43	50.32	67.14
DEC21164	16.5	69.39	33	22.89
AMD K5	88	47.47	176	83.55
PA8000	22	81.16	44	35.71

Table 5: RelaFUPA components and results of recently announced processors

Lower FUPA represents a more efficient FPU design with the lowest FUPA setting a "PAR" for designer to achieve. Table 5 demonstrates the FUPA of some recent microprocessors. From Table 5. we observe a wide range of FUPA results. The R10000 exhibits the lowest FUPA with both the lowest NArea and NEL, an unexpected result contrary to the general assumption that decreased latency is achieved by adding parallelism and die area.

Acknowledgments

This work was supported by the NSF under grant MIP93-13701 and a fellowship from the Saudi National Guard.

List of Recent SNAP Publications

The following is a list of recent publications related to the SNAP project. These and other publications and information on the SNAP project and researchers may be obtained through the World Wide Web using the URL http://umunhum.stanford.edu.

[S1] S. F. Oberman, *Design Issues in High Performance Floating Point Arithmetic Units*, Ph.D. thesis, Stanford University, Nov. 1996.

[S2] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Computers*, vol. 46, no. 2, pp. 154–161, Feb. 1997.

[S3] S. F. Oberman and M. J. Flynn, "A variable latency pipelined floating-point adder," in *Proc. Euro-Par'96, Springer LNCS vol. 1124*, pp. 183–192, Aug. 1996.

[S4] N. T. Quach and M. J. Flynn, "An improved algorithm for high-speed floating-point addition," Technical Report No. CSL-TR-90-442, Stanford University, Aug. 1990.

[S5] H. Al-Twaijry and M. J. Flynn, "Optimum placement and routing of multiplier partial product trees," Technical Report: CSL-TR-96-706, Stanford University, Sept. 1996.

[S6] G. McFarland and M. Flynn, "Limits of scaling MOSFETs," Technical Report: CSL-TR-95-662 Revised, Stanford University, Nov. 1995.

[S7] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *to appear in IEEE Trans. Computers*, 1997.

[S8] D. L. Harris, S. F. Oberman and M. A. Horowitz, "SRT division architectures and implementations," in *Proc. 13th IEEE Symp. Computer Arithmetic*, this volume, July 1997.

[S9] S. F. Oberman and M. J. Flynn, "Reducing division latency with reciprocal caches," *Reliable Computing*, vol. 2, no. 2, pp. 147–153, Apr. 1996.

References

1. ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985.
2. D. Greenley et al., "UltraSPARC: the next generation superscalar 64-bit SPARC," in *Digest of Papers. COMPCON 95*, pp. 442–451, Mar. 1995.
3. L. Kohn and S. W. Fu, "A 1,000,000 transistor microprocessor," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 54–55, 1989.
4. J. A. Kowaleski et al., "A dual-execution pipelined floating-point CMOS processor," in *Slide Supplement to Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 287, 1996.

5. M. P. Farmwald, *On the Design of High Performance Digital Arithmetic Units*, Ph.D. thesis, Stanford University, Aug. 1981.
6. A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196–205, June 1994.
7. SPEC Benchmark Suite Release 2/92.
8. D. W. Sweeney, "An analysis of floating-point addition," *IBM Systems Journal*, vol. 4, pp. 31–42, 1965.
9. O. L. McSorley, "High speed arithmetic in binary computers," *Proc. IRE*, vol. 49, no. 1, pp. 67–91, Jan. 1961.
10. C. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. Electronic Computers*, pp. 14–17, Feb. 1964.
11. L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, pp. 349–356, Mar. 1965.
12. D. T. Shen and A. Weinberger, "4-2 carry-save adder implementation using send circuits," *IBM Technical Disclosure Bull.*, vol. 20, no. 9, Feb. 1978.
13. M. Santoro and M. Horowitz, "A pipelined 64X64b iterative array multiplier," in *Digest of Technical Papers, IEEE Int. Solid-State Circuits Conf.*, pp. 35–36, Feb. 1988.
14. N. Ohkubo et al., "A 4.4 ns CMOS 54*54-b multiplier using pass-transistor multiplexor," *IEEE J. Solid-State Circuits*, vol. SC-30, no. 3, pp. 251–257, Mar. 1995.
15. V. G. Oklobdzija, D. Villeger and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Trans. Computers*, vol. C-45, no.3, pp. 294–305, Mar. 1996.
16. D. DasSarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 12–25, July 1995.
17. M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proc. 12th IEEE Symp. Computer Arithmetic*, pp. 2–9, July 1995.
18. S. E. Richardson, "Exploiting trivial and redundant computation," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 220–227, July 1993.
19. D. Eisig et al., "The design of a 64-bit integer multiplier/divider unit," in *Proc. 11th IEEE Symp. Computer Arithmetic*, pp. 171–178, July 1993.
20. E. Schwarz, "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.