ON THE IMPLEMENTATION OF SHIFTERS, MULTIPLIERS, AND DIVIDERS IN VLSI FLOATING POINT UNITS

Victor Peng - Sridhar Samudrala - Moshe Gavrielov

Digital Equipment Corporation
75 Reed Road
Hudson, MA01749

Abstract

Several options for the implementation of combinatorial shifters, multipliers, and dividers for a VLSI floating point unit are presented and compared. The comparisons are made in the context of a single chip implementation in light of the constraints imposed by currently available MOS technology.

Introduction

Several previous papers [1-4] have presented and compared various MOS implementations of carry acceleration algorithms. For basic floating point instructions the summation of the two mantissas is only one of a sequence of operations that are required for the implementation, several of which may constitute the computational bottleneck.

The purpose of this paper is to describe and compare options for the VLSI implementation of three other major building blocks: shifters, multipliers, and dividers. The emphasis is on solutions that are feasible for a single integrated circuit that is implemented in the MOS technology that is commercially available in the late 1980s. The instructions considered are the four basic arithmetic operations, i.e. addition, subtraction, multiplication, and division. The formats considered are single (32 bits) precision, and double (64 bits) precision with an IEEE floating point standard [5] or VAX standard [6] breakdown between the fields.

The limitation to a single integrated circuit, and the technology used, are the dominant factors in deciding which of the hardware options will be chosen. There are, however, other important factors such as the relative frequency of the instructions and the maximum execution time of a single instruction, that dictate how the available silicon area should be divided among the various hardware functional blocks. Due to the disparity of the applications and their accompanying constraints, there is no universal best solution, and the tradeoffs differ from case to case. This paper presents some of the solutions and points out their relative merits, but because of the the above reason it does not attempt to define an optimal solution for any of the units.

Combinatorial Shifters

There are two instances in the flow of the ADD and SUBTRACT floating point instructions where a large shift may be necessary [7]. The first occurs at the beginning of the operation and is necessary in order to align the mantissa of the operand with the smaller exponent with that of the other operand. The shift required is equal to the difference between the operands' exponents. The second occurs at the end of the operation if, due to mutual elimination of the mantissas' leading bits, normalization is required. The overall performance of the ADD and SUBTRACT instructions are strongly affected by the speed at which these large shifts are done. The hardware units that perform these massive shifts in one cycle are combinatorial shifters. Fully populated versions of these arrays can be utilized for performing rotation of the operands and are commonly called barrel shifters. This section describes four options for implementing the shift function. The first three options utilize full combinatorial shifters and differ in the way the array is partitioned. The fourth option uses a smaller array.

The simplest implementation of a shifter is as a one level array [8]. Figure 1 shows the diagram of an 8 bit right shifter which can be expanded to accomodate any number of bits. It shows the outputs rotated by 90 degrees in relation to the inputs. The layout of the mantissa data path usually dictates that the inputs and outputs of the shifter both point in the same direction. The additional 90 degree rotation which is required to satisfy the preceding criterion is usually implemented as part of the array and is not shown in the figures for the sake of simplicity. It should also be noted that the same hardware can be reutilized to perform both right and left shifts by reversing the direction of the inputs and the outputs. The operation of the array can be accelerated by using a precharged solution, as shown in figure 2. This scheme requires the data inputs to be ready ahead of time and the shift control lines to be predischarged. The advantages of the single level solution are its compactness and simplicity, which facilitate a dynamic array implementation. Its greatest disadvantage is the loading it induces on the data lines. This large loading is due to the fact that a combinatorial shifter for double precision formats

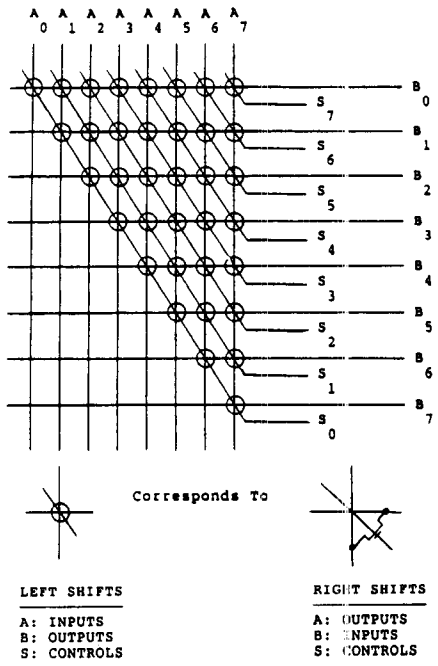


FIGURE 1 – A SINGLE LEVEL 8 BIT SHIFTER

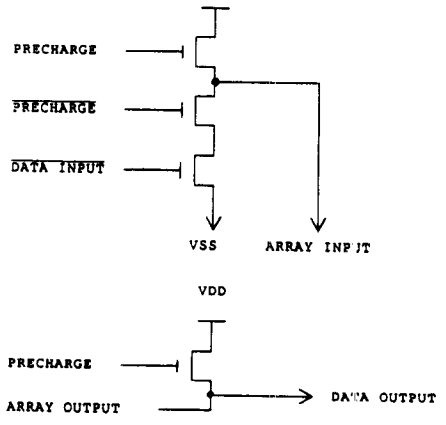


FIGURE 2 – PRECHARGED INPUT AND OUTPUT CIRCUITS FOR A SINGLE LEVEL SHIFTER

requires an array that can perform variable shifts from 0 to 55 bits. A single level implementation will load both the input and the output lines with up to 56 transistors. This will be the critical path in terms of shifter performance. An additional disadvantage of this approach is the need to fully decode all the shift control wires.

One alternative to the single level solution, that alleviates most of its problems, is to split it into two arrays. This is demonstrated in figure 3 for an 8 bit shifter. It has been split into one array that shifts by multiples of 4 bits, and another array that performs 0-3 bit shifts. This approach has the advantages of significantly decreasing

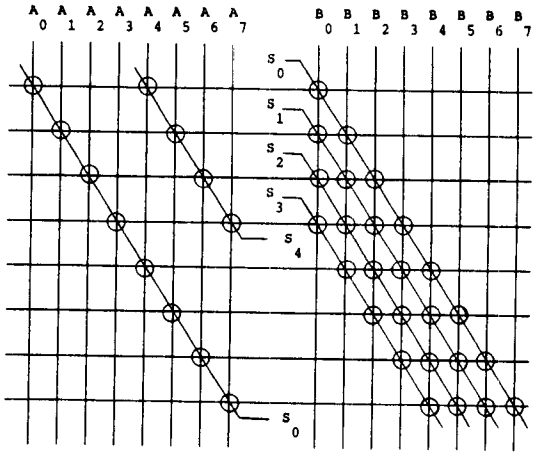


FIGURE 3 – A TWO LEVEL 8 BIT SHIFTER

the worst case loading on both the input and the output lines and does not require full decoding of the shift controls. Its obvious disadvantage is that the shifted data has to traverse two transistors in series instead of one. This may slow down the shifting and eliminate the performance gain attained from reducing the data line loading. It also complicates the implementation of dynamic solutions, and, since two arrays are used instead of one, the overall area required is usually larger.

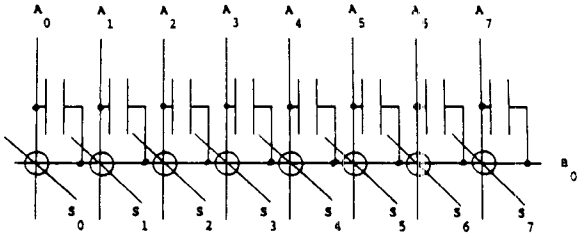
There are several ways of splitting the one level array into two smaller arrays. For example a 64 bit shifter can be split so that the first array shifts groups of either 2, 4, 8, 16, or 32 bits with the second array performing the complementary shift required to reach 64. Generally it is advantageous to try and minimize the total number of transistor drains and sources in any shift path. This leads to a symmetric partitioning of the arrays as an optimal solution in this context and has been used in the past [9]. Regardless of the partitioning scheme used, the second array is relatively empty. Significant area savings can be realized if it is rotated by 45 degrees, and only the active area is laid out.

The "partitioning" approach described above can be taken even further by splitting the shifter into more than two levels. These multi-level solutions have the same general advantages and disadvantages of the two level array but they are manifested to a greater extent. It should be noted that when the partitioning is taken to its limit, the resulting arrays are just large 2 to 1 multiplexers, each of which is controlled by one encoded shift bit.

A different approach to performing the shifts is by using a small array, capable of shifting the mantissa by only a few bits at a time. In this approach large shifts are implemented by sequencing through the small shifter several times. This seems like a potentially slow solution, but it has the obvious advantage of being the most efficient in terms of area. Moreover the performance degradation is

data dependent, and it has been shown [10] that for several applications the percentage of large alignments and normalizations is relatively low. This statistical approach has had widespread use [11]. It was especially prevalent in chips implemented in older technologies due to the lack of silicon real estate for a full fledged shifter. It is not a feasible solution for cases where consistent data independent performance is required, such as in fully pipelined applications.

The use of large regular arrays to implement shifters has some inherent problems. The first and most obvious is the large capacitance that the shift control lines have to drive. This is due to the number of transistor gates that are driven by each line, i.e. a maximum of 56 for a 0-55 bit shifter. The resulting capacitance normally precludes laying out the control lines in polysilicon due to the large RC delay. The second problem is caused by coupling between the lines and is especially severe in dynamic arrays. One instance of this problem, which is illustrated in figure 4, occurs if all the data outputs are initially precharged, the control lines are initially discharged and all the data inputs except for one bit are low. When the control goes high, the output nodes are discharged, and the output node that was meant to remain high may erroneously be pulled low by the capacitative coupling from the crossing wires.



SEQUENCE:

1. A INPUTS AND B OUTPUTS ARE PRECHARGED AND THEN $A_1 - A_7$ ARE DISCHARGED.

   SHIFT CONTROL INPUTS $S_0 - S_7$ ARE ALL INITIALLY LOW.

2. S_0 GOES HIGH, SIGNALING A SHIFT OF 0 BITS.

   $A_1 - A_7$ ARE DISCHARGED.

3. $B_1 - B_7$ ARE DIRECTLY DISCHARGED.

   B_0 IS PARTIALLY DISCHARGED DUE TO THE CAPACITATIVE COUPLING

FIGURE 4 - ONE INSTANCE OF THE SHIFTER COUPLING PROBLEM

Multiplier Array Alternatives

The relatively high frequency of multiply instructions clearly justifies a substantial investment in silicon area even for single chip implementations. This section describes four possible multiplier implementations and some of the advantages and disadvantages of each with respect to latency, area and complexity. The first option is a simple carry save adder (CSA) multiplier array which serves as the baseline for comparisons. The second and third options are radix 4 and radix 8 modified Booth multipliers respectively. The last set of implementations considered are tree-like structures for more optimal reduction of the partial products.

The straightforward implementation of a multiplier for n bit operands is an n row by n column array of AND gates and carry save adders (CSAs). This will be referred to as a simple CSA array multiplier, a segment of which is shown in figure 5. An interesting point to note is the direction of the carry and sum outputs from the cells. This is necessary in order to create a rectangular shaped array which does not waste silicon area. One consequence of this is that in successive rows of the array the same physical bit positions have different weights. The n least significant bits of the final sum are output on the right side of the array, while an n bit carry vector and an n bit sum vector are output at the bottom of the array. The n most significant bits of the product are formed by adding the corresponding bits of the final carry and sum outputs.

This implementation is noteworthy for its regularity and simplicity in terms of both logic and layout. In addition there is no need to preprocess either of the operands, i.e. only the multiplier and multiplicand are required to start the passage through the array. Its major disadvantages are that it is very costly in area, because it requires $O(n^2)$ cells, and in addition the resulting performance is not impressive, since it takes O(n) CSA delays. As a result of these two serious drawbacks it is not usually used for single chip floating point processors. However its simple structure lends itself to fully pipelined, dedicated multiplier, implementations where the minimization of the pipeline stage delay is the ultimate goal [12,13].

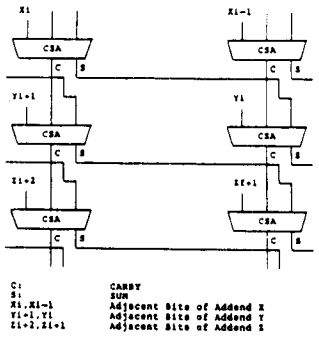


FIGURE 5 - A SEGMENT OF A SIMPLE CSA ARRAY MULTIPLIER

The use of the radix 4 version of the modified Booth algorithm [14] reduces the number of rows required in the multiplier array to O(n/2). Subsequently the propagation delay through the array is reduced to O(n/2) CSA delays as well. The additional hardware required for its implementation is the multiplier recode logic and a partial product selector per CSA cell. In order to retain an array with a rectangular outline, the carry bits propagate down and are right shifted by one bit, while the sum bits propagate down and are right shifted by two bits. This means that one carry bit and two sum bits are right shifted across the array boundary in each row. Since they can affect the most significant bits of the product via carry propagation, they cannot be discarded. An undesirable consequence of this is that at the end of the computation a 2n bit carry propagation addition of the final carry and sum vectors must be performed, instead of the n bit addition required by the simple array. One way to reduce the final carry propagate addition from 2n bits to n bits is by adding two bit adder cells to the right edge of each row of the array. These cells perform the carry propagate addition of the bits which are shifted out and inject the resulting carry back into the low order CSA in the next row. They will be referred to as LSB cells for the remainder of this section. In order to maintain the factor of two performance improvement of the radix 4 modified Booth algorithm, it is necessary to design the LSB cells with the same delay as a single bit CSA.

A more careful look at the implementation of the radix 4 modified Booth algorithm reveals a few additional drawbacks. There is a one time performance penalty at the beginning of the operation due to the fact that the multiplier must be recoded, and the outputs of the recoding logic driven across the array, before any valid carry and sum bits begin to propagate through it. The partial product selectors increase the propagation delay of the carry and sum bits in at least the first row of the array and, depending on the exact implementation, may do so in some of the subsequent rows too. The area gain is not as promising as it initially looks, because the partial product selectors and wiring interconnect are much larger than the corresponding logic in the simple CSA solution. Overall the area required for a radix 4 modified Booth multiplier which can multiply two mantissas with over 50 bits is still prohibitively large for single chip processors. One way to reduce this area is to use an array which has sufficent rows for single precision operands and "double pump" it for double precision operands. This is a viable solution, but it adds to the timing complexity of the design and requires extra vertical busses for feedback of the intermediate Carry and Sum results. Despite these drawbacks, the radix 4 modified Booth algorithm has been used very extensively [15–17]. The number of CSA rows required for the implementation of multiplication can be reduced even more by the use of a radix 8 modified Booth algorithm. Since it retires 3 multiplier bits per iteration it takes only O(n/3) CSA delays and requires $O(n^2/3)$ cells in the array. The advantages of the radix 8 solution are its higher speed and smaller area, which are a direct outcome of the reduced number of rows.

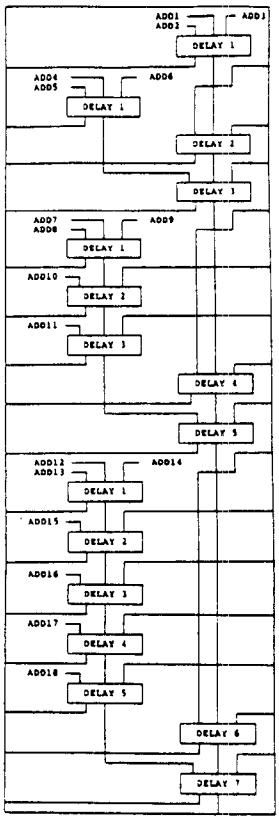
Most of the disadvantages of the radix 8 modified Booth multiplier are similiar to those of the radix 4 implementation. In fact, since the recode logic, the selectors, and the interconnect are more complex in the radix 8 case, the additional delays will probably be longer. In addition to these problems, which are common to both implementations, there are additional drawbacks which are unique to the radix 8 version. One of these is due to the need for the precalculation of 3 times the multiplicand for its implementation. Another problem is that the performance of the array is more severely limited by the propagation delay of the LSB cells rather than the CSAs. This is due to the fact that the LSB cell for the radix 8 implementation is actually a three bit carry propagate adder. This makes matching its delay to that of the CSA extremely difficult. However, it is always possible to avoid this problem by not using the LSB approach and implementing a 2n bit carry propagation addition for the final product formation. Radix-8 based multiplier arrays have been implemented in single chip floating point units [11]. However, for radices larger than 8, the problems involved in the implementation of the modified Booth algorithm are exacerbated and do not look promising for VLSI implementation with current technology.

A completely different approach to accelerating the multiplication process is by the use of Wallace [18] or Dadda [19] tree based algorithms. These approaches offer a minimial number of logic gate delays in the critical path of the array. However, the complexity and the irregularity of the interconnect make the implementation of these schemes difficult and extremely area consuming. An additional drawback is due to the relatively unordered way in which the partial products are summed. This precludes the efficient use of LSB cells and usually necessitates that the arrays be 2n bits wide for n bit operands. As a result of these area-consuming characteristics, tree solutions have been used almost exclusively in dedicated multiplier chips [20,21].

There are several ways in which the tree structure can be modified in order to attain higher layout oriented regularity without completely sacrificing its high performance. One of these [17] uses a "pseudo Wallace" tree to achieve a delay which is proportional to the square root of the number of operand bits. An illustration of a typical "slice" used for the summation of 16 bits appears in figure 6.

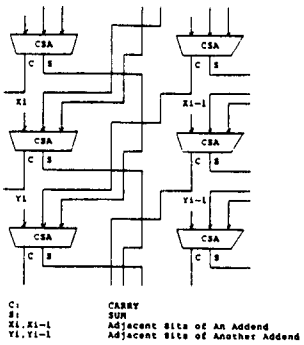
The analysis of this approach shows that the maximum propagation delay for adding n^2 summands is 2n-1 CSA delays. Furthermore, the resulting structure is very modular and consists of only three types of cells with both horizontal and vertical connectivity built into the slice. The disadvantage of this approach, relative to any of the highly ordered arrays already discussed, is that the interconnect between the cells gets longer as you proceed down the slice. Another drawback is that like most other tree based solutions, LSB cells cannot be used. This requires the array to be 2n bits wide for n bit operands.

Another technique, which has been used in a dedicated multiplier chip [22] and is even better adapted to VLSI implementation, breaks the regular array down into odd and even CSA rows. The outputs of the odd rows skip the even rows and connect to the next odd row The same is done with the outputs of the even rows and is illustrated in figure 7. This parallelism cuts down the number of CSA delays to about one half of what is required in a normal array without compromising its regularity. Retaining the regularity also enables the use of an LSB type solution in order to avoid having a 2n bit wide array.



REFERENCE [17]

FIGURE 6 – PSEUDO WALLACE TREE STRUCTURE



REFERENCE [22]

FIGURE 7 – ODD AND EVEN ROW TREE STRUCTURE

Division Implementation Options

In most applications, division is the least frequently used of the basic floating point operations, and it is definitely the most difficult to implement. One consequence of these facts has been that, in the past, its execution time has usually been much longer than that of multiplication and addition. There are several applications for which significantly longer execution time is unacceptable. In some cases this may be due to system oriented requirements, such as the need to minimize the interrupt response time. In others, such as computer graphics, this may be driven by the need to avoid a computational bottleneck in a highly macro instruction pipelined environment. For these applications, which are sensitive to the speed of their slowest instruction, it is important to accelerate the floating point division operation.

There are two basic approaches that can be taken in a VLSI implementation to accelerate the computation of division. The first is to reutilize the fast hardware used for the implementation of the other operations, while adding a minimal amount of extra hardware for division acceleration. The second is to use a dedicated hardware divider. The intent of this section is to present variations of both options while keeping in mind the constraints of a single chip solution.

The most basic method is to implement an iterative, restoring or non-restoring, radix 2 algorithm. If this approach is taken, the extra area that would have been dedicated to the acceleration of division can be utilized for the implementation of an extremely fast adder. This has the significant advantage of accelerating all four basic floating point operations at a relatively low price in silicon real estate. Its main disadvantage is due to the fact that single and double precision floating point division require 26 and 55 iterations respectively. Thus, fast as the adder may be, division will remain at a significant performance disadvantage relative to other operations. However, due to the method's simplicity and the overall performance gain, it has been used extensively in the past [9].

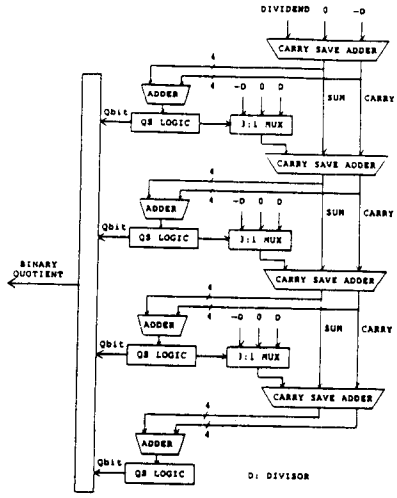
One interesting way to improve the performance is by implementing normalizing division [23]. This is done by examining the partial remainders' high order bits. If these are all 0s (in restoring algorithms), or are either all 0s or all 1s (in nonrestoring algorithms), the magnitude of the partial remainder is relatively small compared to that of the divisor. In these cases the partial remainder can be normalized by several bits in one iteration. These multi-bit normalizations correspond to the computation of several of the quotient bits in a single iteration. This algorithm's most striking advantage is the small amount of additional hardware required (detection logic on the most significant bits of the partial remainder, and a variable length shifter, which is usually necessary for the implementation of addition and subtraction anyway), while yielding a relatively significant performance boost. It has been utilized in a single chip floating point accelerator [11] with a reported average speedup of 50%. Its disadvantages are the data dependency of the performance speedup and the worst case execution time which remains the same.

One way of attaining data independent acceleration is by implementing higher radix divison. The brute force method of doing this is by the use of several adders that perform concurrent magnitude comparison of the partial remainder with all the relevant multiples of the divisor. This technique has been used to implement radix 8 division in a dedicated divider chip [24]. Its major disadvantage is that it requires n-1 adders and a large n to 1 selector in order to implement radix n division. Multiple fast adders are extremely expensive in terms of silicon area and effectively rule out this sort of implementation for a single chip solution.

A more attractive option for VLSI implementation is the SRT algorithm which eliminates the need for carry propagate addition during the iterative division steps. A radix 2 implementation of this scheme is illustrated in figure 8. In this particular implementation a carry save adder (CSA) is used instead of a carry propagate adder to form the partial remainder. A small, four bit carry propagate adder (CPA) is used to sum the corresponding most significant bits of the partial remainder. This sum is used to select the correct multiple of the divisor for the next step. The CSA adds the sum and carry portions of the partial remainder from the previous stage and the selected multiple of the divisor. This scheme attains relatively high performance at a low price in silicon real estate. The high performance is due to the relatively short critical path which consists of only the CSA delay, the four bit CPA delay and the selection logic delay. In one implementation of this algorithm [17] the total delay for the critical path is reported to be only about one eighth of a corresponding carry propagate adder. The additional hardware required per stage is relatively small and consists of a one row CSA, a 4 bit CPA, and a 3 to 1 selector.

Higher radix implementations of the SRT algorithm [25] may have the potential of providing even higher performance, as has been demonstrated in a multi-chip ECL solution [26]. The additional hardware required for the higher radix implementation is relatively small. To demonstrate this, the radix 2 SRT implementation of figure 10 can be readily adapted to accomodate radix 4 by using: a 7 to 1 selector instead of the 3 to 1 selector, a 7 bit CPA instead of the 4 bit CPA, and more complex logic to select the correct multiple of the divisor. In order to realize the gains from the higher radix implementation, it is necessary to verify that its critical path takes less than twice the radix 2 critical path. In terms of area it is definitely more attractive than the radix 2 implementation. Both radix 2 and radix 4 SRT implementations require the insertion of an additional carry propagation adder and selection logic adjacent to the high order bits of the data path. This additional logic does not fit well into the data path and causes a "bulge" in the layout of the high order bits. This normally translates into a significant amount of unutilizable area in the low order bits.

The last algorithm considered for the implementation of division is multiplicative approximation. It has been used extensively in multi-chip and externally sequenced solutions [27,28]. However due to the area required for the seed lookup table and the large number of iterations required to attain the accuracy defined in both the IEEE [5] and VAX [6] standards, it does not appear to be a promising technique for single chip solutions.



REFERENCE [17]

FIGURE 8 - SRT DIVIDER USING CARRY SAVE ADDERS

Conclusion

This paper has described and compared options for the VLSI implementation of combinatorial shifters, multipliers, and dividers. The emphasis has been on a single chip MOS solution with the goal of attaining high overall floating point performance. At current integration levels it is not possible to use full sized arrays for the implementation of all three units. As the optimal solution is application dependent, the hardware tradeoffs are most heavily affected by the relative frequency of the instructions. As technology advances, due to larger chip sizes, smaller critical dimensions, and additional interconnect levels, it will become feasible to implement algorithms that are currently limited to multi-chip solutions. When this occurs it is expected that additional acceleration will be attained mostly by a higher degree of computational parallelism.

References

[1] On Fast Binary Addition in MOS Technologies
Jean Vuillemin and Leonidas Guibas
Proc. ICCC82, Pages 147-150.

[2] Some Optimal Schemes for ALU Implementation in VLSI Technology
Vojin G. Oklobzija and Earl R. Barnes
Proc. 7th Symposium on Computer Arithmetic
June 1985, Pages 2-8.

[3] A Comparison of ALU Structures for VLSI Technology
S. Ong and D.E. Atkins
Proc. 6th Symposium on Computer Arithmetic
June 1983, Pages 10-16.

[4] Regular, Area-Time Efficient Carry Look-Ahead Adders
Tin-Fook Ngai and Mary Jane Irwin
Proc. 7th Symposium on Computer Arithmetic
June 1985, Pages 9-15

[5] IEEE Standard for Binary Floating-Point Arithmetic
August 12th 1985

[6] VAX Architecture Reference Manual
September 23rd 1983
Digital Equipment Corporation.

[7] Some Tricks of the (Floating Point) Trade
J.B. Gosling
Proc. 6th Symposium on Computer Arithmetic
June 1983, Pages 218-220.

[8] The Design and Analysis of VLSI Circuits
Lance A. Glasser and Daniel W. Dobberpuhl
Addison-Wesley, 1985

[9] VLSI Floating-Point Processors
Jan Fandrianto, B.Y. Woo
Proc. 7th Symposium on Computer Arithmetic
June 1985, Pages 93-100

[10] An Analysis of Floating-Point Addition
D.W. Sweeney
IBM Systems Journal, Vol. 4, 1965.
Pages 31-42.

[11] A High Performance Floating Point Coprocessor
Gil Wolrich, Edward McLellan, Larry Harada,
James Montanaro and Robert A. J. Yodlowski
IEEE Journal of Solid State Circuits,Vol.SC-19,
No. 5., October 1984, Pages 690-696.

[12] A Pipelined 330-MHz Multiplier
Tobias G. Noll, Doris Schmitt-Landsiedel,
Heinrich Klahr and Gerhard Enders
IEEE Journal of Solid State Circuits,Vol.SC-21,
No. 3., June 1986, Pages 411-416

[13] A 70-MHz 8-bit by 8-bit Parallel Pipelined Multiplier in 2.5-uM CMOS
Mehdi Hatamian and Glenn L. Cash
IEEE Journal of Solid State Circuits,Vol.SC-21,
No.4., August 1986, Pages 505-513

[14] A Proof of the Modified Booth's Algorithm for Multiplication
L.P. Rubinfield
IEEE Transactions on Computers, Vol. C-24,
No. 10, October 1975, Pages 1014-1015

[15] A CMOS Floating Point Multiplier
Masaru Uya, Katsuyuki Kaneko, and Juro Yasui
IEEE Journal of Solid-State Circuits,Vol.SC-19,
No.5., October 1984, Pages 697-701

[16] A Single Chip 80b Floating Point Processor
Kazumitsu Takeda,Fumiaki Ishino,Yoshitaka Ito,
Ryota Kasai,and Takayoshi Nakashima
1985 IEEE ISSCC
Digest of Technical Papers, Pages 16-17

[17] An NMOS 64b Floating-Point Chip Set
William M. McAllister, Dan Zuras
1986 IEEE ISSCC
Digest of Technical Papers
Pages 34-35, 294-295

[18] A Suggestion for a Fast Multiplier
C.S. Wallace
IEEE Trans. Electron. Comput., EC-13, 1964
Pages 14-17

[19] On Parallel Digital Multipliers, L. Dadda
Alta Frequenza, Vol. 45, 1976, Pages 574-580

[20] A 45ns 16x16 CMOS Multiplier
Yoshio Kaji, Nobuyuki Sugiyama, Yoshishige Kitamura, Suichi Ohya and Masanori Kikuchi
1984 IEEE ISSCC
Digest of Technical Papers, Pages 84,85

[21] A CMOS 32b Wallace Tree Multiplier-Accumulator
Abbas El Gamal, David Gluss, Peng-Huat Ang, Jonathan Greene and Justin Reyneri
1986 ISSCC
Digest of Technical Papers, Pages 194,195.

[22] A 16-Bit CMOS/SOS Multiplier-Accumulator
Jun Iwamura, Kazuo Suganuma, Sinji Taguchi, Minoru Kimura and Kenji Maeguchi
Proceedings ICCC82, Pages 151-154

[23] High-speed Arithmetic in Binary Computers
O. L. MacSorley
Proc. IRE, vol. 4, 1961.
Pages 67-91.

[24] Floating-Point Chip Set Speeds Real-Time Computer Operation
William H. McAllister and John R. Carlson
Hewlett-Packard Journal
February 1984, Pages 17-23

[25] Higher Radix Division Using Estimates of the Divisor and Partial Remainders.
Daniel E. Atkins
IEEE Trans. Computers, Vol. C-17,
October 1968, Pages 925-934

[26] Radix 16 SRT Dividers with Overlapped Quotient Selection Stages
George S. Taylor
Proc. 7th Symposium on Computer Arithmetic
June 1985, Pages 64-71

[27] Performing Floating Point Division with the WTL 1032/1033
Application Note
Weitek Corporation, 1984

[28] Am29325 32-Bit Floating Point Processor
Advanced Micro Devices
February, 1986