Array-of-arrays Architecture for Parallel Floating Point Multiplication

Hema Dhanesha, Katayoun Falakshahi and Mark Horowitz
Center for Integrated Systems, Stanford University
Stanford, CA 94305

Abstract

This paper presents a new architecture style for the design of a parallel floating point multiplier. The proposed architecture is a synergy of trees and arrays. Architectural models were designed to implement the 53-bit mantissa path of the IEEE standard 754 for floating point multiplication, and tested for functionality in Verilog. The design, which was done in dual-rail domino, was simulated in HSpice with estimated capacitive load models in a 1μm CMOS technology. Multiplication latency of 10ns (23.3 FO4) at 4.3V supply and 120^0C can be achieved with the best topology of the array-of-arrays architecture. The estimated multiplier area is 3mm x 6mm.

1: Introduction

Floating point multiplication involves a large number of additions to compress many partial products into the final product. Current floating point units use a variety of circuit structures to implement the floating point multiplier. These designs have chosen different trade-offs between architectural complexity, and circuit style. Some designs use simple full arrays with fast circuit technology to achieve high performance [5][8]. Other designs use complex tree adder structures to minimize the gate count in the critical path [4][6][7]. This paper looks at the constraints and trade-offs involved in designing a multiplier architecture and explains why the performance difference between trees and arrays is smaller than one might expect. Using this information, we propose some new multiplier arrangements that combine the virtues of arrays and trees.

Tree adder structures employ parallelism in adding partial products to reduce the critical path. To exploit this parallelism requires a fairly complex wiring network to move the partial sums to the next adder in the tree. This wiring makes it tough to layout a full Wallace tree, and even harder to place this tree into a datapath. There has been some work on reducing this complexity by using modified trees (usually 4:2 trees) [9][7] or by automation of the layout [1]. The fundamental problem with trees is that of routing a large number of wires, which still remains.

The wiring required in a tree increases the area and adds capacitive loading to the critical path. For a reasonable bit pitch, the number of wires crossing over cells requires use of single ended signals, which in turn forces the use of static CMOS circuits. Thus the wires slow down the circuit in two ways: larger loading on the cells, and forcing the use of slower base cells.

In contrast to trees, full arrays have extremely simple wiring and layout. Wires run only to neighboring cells, and the cells fit nicely into a general datapath layout. The cost of this simplicity is a larger number of gates in the critical path -- the number of partial product additions in the critical path is about three times larger as compared to a full tree. However, because of sparse wiring, arrays can use dual-rail dynamic adders with relatively light output loading. Our results show that a dynamic adder is more than twice as fast as the static version of the cell. This difference in circuit speeds is what makes the choice of multiplier architecture difficult. Both circuit and architecture issues need to be considered simultaneously.

Arrays and trees represent two extremes in multiplier organizations. To address some of the drawbacks of each, combinations of arrays and trees have been proposed. The goal is to reduce the number of adders in the critical path in an array-like architecture while maintaining sparse wiring and regular layout. The most common solution is to split the full array into odd and even partial products and interleave the additions [3]. This reduces the critical path by almost half as compared to a full array. The wiring is similar to the wiring of full arrays. This paper is an extension of this work, focusing on different architectures that break up the full array into smaller parts and use a combining network (another array or a tree) to produce the final result.

Section 2 presents a quick review of some of the characteristics of multiplication with focus on carry save arrays. Section 3 then describes different sub-array multiplier organizations, focusing on three architectures in detail. Finally some performance results from these designs are presented and compared with the existing architectures.

2: Floating Point Multiplication

The mantissa part of IEEE floating point multiplication takes two 53-bit unsigned binary numbers and generates a 53-bit result rounded to nearest/even, and an overflow bit used to normalize the result. Modified Booth encoding [2] is generally used to reduce the number of partial products to 27, each 54-bits wide (53 + sign extension bit due to possible negation after Booth encoding). These partial products are added in a carry save array to produce a 106-bit output in carry save form. This is then added in a carry propagate adder to give the final result.

A carry save array is a cascade of full adders (CSAs). For m-bit multiplication, each stage of the array passes on m-2 most significant bits of its sum and carry outputs to the next stage, shifted down by two bit significance. The two least significant outputs go out of the array as two bits of the final output. The first stage of an array can add 3 partial products together. Each subsequent stage adds one new partial product with the sum of the previous partial products. Thus, an n-stage array adds n+2 partial products. The array has one Booth encoder per partial product and one Booth mux per partial product bit to locally generate the new partial product going to the CSAs.

The most significant bit of the 106-bit result is the overflow bit, the next 53 bits are used to generate the IEEE compatible output. The 55th most significant bit is the round bit. The 51 least significant bits are added in a carry propagate adder to generate a carry and a sticky bit. The round bit and the carry output of the least significant bits are added to perform rounding to nearest/up[11]. Correction to nearest/even rounding is then performed by using the sticky bit.

Booth encoding can create negative partial products that need to be handled in the multiplier array. This area has been covered elsewhere [1] and will be briefly reviewed. Negative partial products cause two issues: sign extension, and the addition of the carry in. Sign extension is solved by placing an extra bit at the MSB of each partial product, which in effect is the ‘pre-added’ sign extension. The carry in is handled by adding the sign bit to the partial product. We could take care of this by adding only two partial products in the first CSA stage (rather than 3) and having one extra CSA delay in the critical path to add the sign bit. Instead, we shift the sign bit down to the next partial product so that the least significant sum output of a partial product is added with its sign bit while the next partial product addition is going on. This requires two more CSAs appended after the least significant CSA in the array. The last partial product generated by Booth encoding is always positive, so its sign bit is always zero and hence does not need to be added. If multiplication is done in parallel sub-arrays, the sign bit of the last partial product of an array (i) is sent down to the first partial product of the following array (i+1), along with the least significant sum output of array (i).

3: Sub-Array architectures

This section looks at architectures that address the trade-offs in multiplier design. The idea is to combine the concepts of arrays and trees, aimed at achieving low latency while keeping the wiring and area overheads small. This is achieved by breaking up a full array into smaller sub-arrays and then using combining arrays or trees to get the final result. Having more sub-arrays gives more parallelism in adding the partial products, but also adds wiring and layout complexity. Architectures with two and four sub-arrays are discussed below.

3.1: Two Half Arrays

The simplest option is to divide the full array into two half arrays and combine the results of the two. Figure 1 shows this architecture.

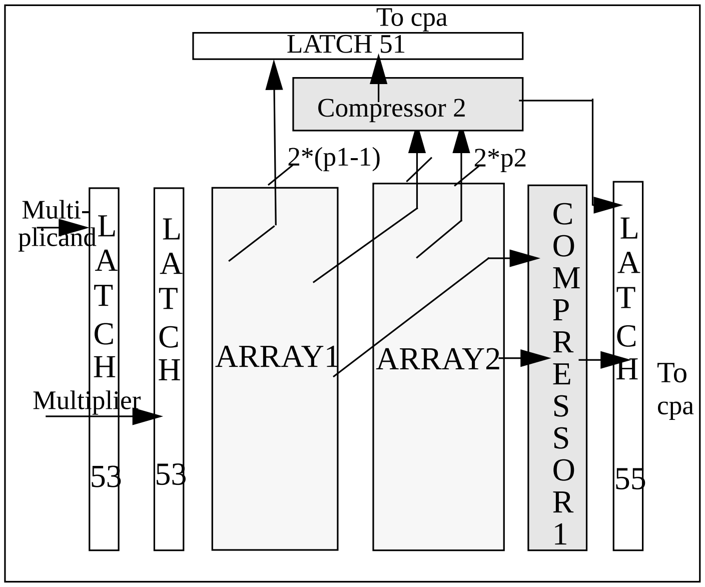

Figure 1. Two Half Arrays Architecture.

The 27 partial products are divided into two smaller groups of adjacent partial products. Arrays 1 and 2 are carry save arrays, adding *p1* and *p2* partial products respectively (*p1*+*p2*=27). Compressors 1 and 2 are 4-2 compressors, that combine the carry save outputs of the two arrays to produce the final output in carry save form. A 4-2 compressor is essentially a 2-stage carry save array.

The two least significant outputs of each partial product ‘fall off’ the datapath on top. When the output of array1 (*out1*) crosses over array2, it is shifted by two bits per partial product of array2 along with the inter-stage carry save signals of array2. Thus, *out1* crosses over array2 diagonally. This aligns the bit positions of the outputs of the two arrays at the top of the datapath as well as at the inputs of compressor1. Compressor 2 combines 2**p2* least significant outputs of the two arrays on top while compressor1 combines 54 most significant outputs in the datapath.

To exploit the parallelism made available by two sub-arrays, we would like to design array1 and array2 so that their outputs arrive at the same time at the inputs of the compressors. *Out1* has a lot of wire loading capacitance, so array1 needs to be smaller than array2 in order to equalize the delays. From simulations, the increase in wire loading on the output of array1 is equivalent to 1.5 times CSA delay. The best balance is achieved for *p1* = 13 and *p2* = 14. Hence, array1 has 11 CSAs and array2 has 12. Our critical path for carry save addition is roughly 14.5 CSAs (array1 + wire loading + compressor). This is a significant reduction over the 25 CSAs long critical path of a full array, but is still far from 7 CSAs long critical path of a full tree.

We further divided the 27 partial products into four smaller groups of adjacent partial products that are added in parallel by four sub-arrays. The outputs of the sub arrays need to be combined by a 8-to-2 combining network. This can be a tree which has a critical path of 4 CSA delays or an array which has 6 CSA delays.

3.2: Four Quarter Arrays and another array:

To reduce the required wiring, we explored the option of a 8-to-2 array as the combining network. We call this the array-of-arrays architecture and it is shown in Figure 2. Arrays 1, 2, 3 and 4 add *p1*, *p2*, *p3* and *p4* partial products respectively, where *p1*+*p2*+*p3*+*p4* = 27. We divided the combining array into three 4-to-2 compressors placed closest to their inputs in the datapath; in order to preserve regularity in wiring and to minimize the number of wires crossing over arrays. Compressor1 combines the outputs of arrays 1 and 2. Compressor2 combines the combined output of arrays 1 and 2 coming from compressor1 with the output of array3. Similarly, compressor3 combines the combined output of arrays 1, 2 and 3 coming from compressor2 with the output of array4, to give the final result.

The least significant 2*(*p1*-1) bits of array1 fall off the datapath on top of array1. The 54 most significant bits of the outputs of arrays 1 and 2 (*out1* and *out2*) are aligned at the inputs of compressor1. The 2**p2* lower significant bits of the two arrays get aligned and fall off the datapath above array2. Similarly, we get signals with aligned bit significants at the inputs of compressors 3 and 4 and above arrays 3 and 4. Since the carry save outputs of an array cross over only the next array, a 4-to-2 compressor is required at the top of the datapath to combine least significant carry save outputs of two adjacent arrays. Compressor4 on the top combines these lower significant outputs from arrays 1 to 4.

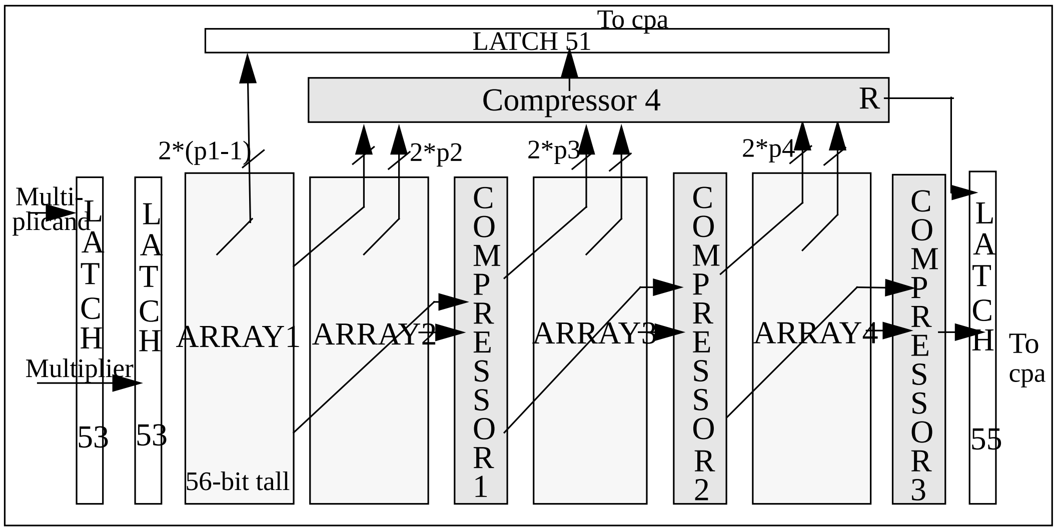

Figure 2. Array-of-arrays architecture

This architecture has very regular wiring and layout properties. All the signals logically flow from left to right in the datapath. The bit-pitch shift between two CSA stages is always 0-2 bits. At the most one pair of carry save output crosses over another array. This lack of dense wiring in a bit pitch allows us to use fast dual rail circuit style in this architecture. The height of the datapath is increased by compressor4 and the latch on top, and one Booth encoder at the bottom of the datapath. The architecture is very well suited for datapaths due to regular wiring and layout.

To minimize the delay, we designed the arrays so that two inputs to each compressor arrive at the same time. Since one of the inputs comes from another unit driving a long wire, the numbers of adds done in this unit should be less than the local array. This implies that array1 will have fewer adders than array2, array3 will have fewer adders than array2 and compressor1 combined, etc. Ignoring the wire loading, this would give a partition of 5,5,7,10 to add the 27 partial products (3,3,5,8 CSAs in arrays1-4 respectively) with a critical path of 9 CSAs (3 in array1 and 6 in compressors 1, 2, and 3). We refer to this as array-of-arrays1 architecture.

When wire loading is taken into account the situation is a little more complex. Our simulations show that the longest wire delay in this architecture is roughly equivalent to 1 CSA delay. The optimal partition is 4,4,8,11 partial products (2,2,6,9 CSAs in arrays1-4 respectively), and our simulations show that in this architecture the arrays and wire loadings are balanced such that the critical path is entirely in array4. The critical path is 11 CSA delays. This is referred to as array-of-arrays2 architecture.

3.3: Four Quarter Arrays and a Tree:

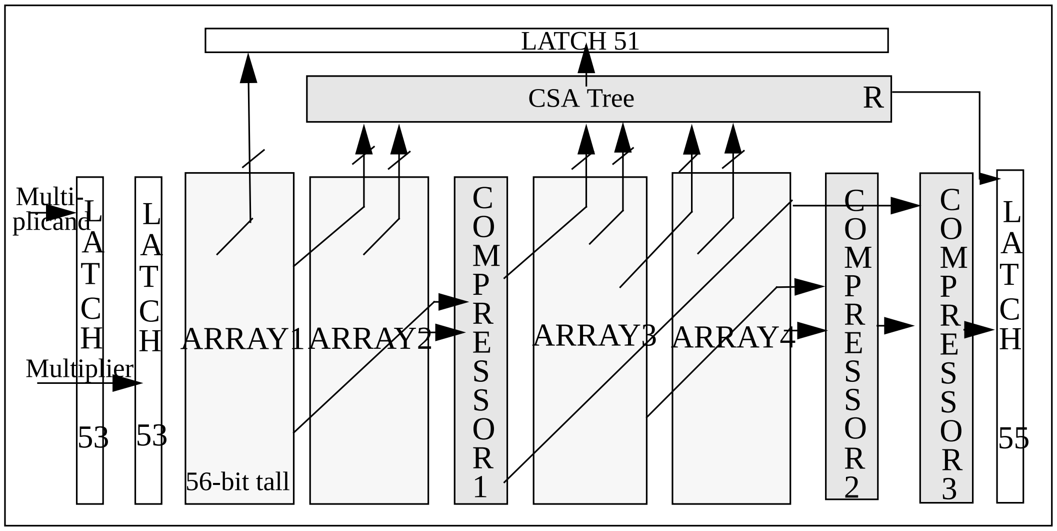

Figure 3. Four Quarter arrays and a Tree

We also explored the tree combining network, but the tree required many more wire tracks, longer wires and more complex routing. The simplest layout is very similar to the previous design and is shown in Figure 3. Here the outputs of arrays 1, 2, 3 and 4 are combined by a 8-2 tree formed by compressors 1 2 and 3 in the datapath and a 8-2 tree at the top. Note that the outputs of the first compressor and array3 need to route on top of array4. Since each bit is represented by 2 wires (sum and carry) and each wire is dual rail encoded, this is 8 extra lines that must run across a cell.

If wire loading is ignored, the arrays should all be the same size, so the partition would be 6,7,7,7 partial products. The critical path through this would be 9 CSAs (5 in the array, and then 4 in the combining tree). When the wire delays are considered, again the latter arrays should be slightly larger to equalize the delays. Since the long wire delays are roughly equivalent to one CSA, we get an optimal partition of 6,6,7,8 and a critical path of 10 CSAs. The height of the datapath increases both because of the increase in wires routing over each cell, and because the combining network at the top is now a tree.

The increase in area, additional wiring required and its modest delay difference caused us to drop the tree combiner, and we focused on the array-of-arrays2 architecture. The next sections discuss our model of the critical path and gives simulation results.

4: Implementation:

The first three sub-array architectures were designed with 1µm dual rail domino logic because it is very well suited circuit technology for arithmetic functions. We fixed the vertical bit pitch to be 80 lambda (40µ), which limits the number of wiring tracks per bit pitch to 11 of which we always have the following in any architecture: 1 power/ground + 4 sum, carry outputs (carry save output of addition) + 1 partial product bit. This leaves us with 5 extra tracks per bit pitch, which means we can run only one extra set of CSA outputs over another CSA stage. The sub-array architectures with arrays in the combining networks can thus be used with dual rail circuit style. We also laid out the four array-of-arrays architecture in MAGIC to verify our area and capacitance estimates. Table 1. summarizes these areas.

Table 1. Area

Description	# cells in a bit slice	Area in width µm
Input Latch	2	101
Booth Mux	27	81
CSA	19 + 6	72
Output Latch	2	22
Area of cells	-	3400
Area of wiring channels	-	2600

Total Area = 3mm high + 6mm width. As can be seen from the relative areas of wiring and cells, 44% of the total area is spent in wiring channels. This area can not be reduced as the number of wires that need to be routed in an architecture as well as the wire pitch is constant. Routing the additional set of carry-save output diagonally across the array increases the size of the multiplier by about 30% in this technology. The difference in area between the split array organizations (2 half arrays, interleaved arrays, 4 quarter arrays) is small. This area overhead for wires grows much larger if a tree organization is used.

5: Simulation Results:

We broke the critical path of the multiplier into three basic pieces: setup, addition, output. The setup phase is the time required to get the data from the input latches to drive the wires to the Booth Encoders located along the bottom edge of the array, through the encoders to drive the control lines to the Booth Muxes, and then to have the data arrive at the inputs of the CSAs. The addition phase includes all the time needed to combine the partial products into a single carry-save result. The output phase is simply the time to latch the data and drive it to the carry propagation adder. The setup and output part of the critical path are the same in any architecture. Addition time depends on the way the array is designed and we explored a number of sub-array organizations.

We modelled the wiring capacitances using aggressive formulae with known areas and process parameters for unit capacitance over length for a wire travelling over active, and over metal. The circuits were simulated with all the fan-outs and wiring loads modelled in HSPICE with worst-case operating conditions - 4.3v supply voltage and 120^{o}C temperature. The delays were measured between 50% to 50% points.

To normalize out the technology, the delay numbers are also measured relative to an inverter driving a load that is 4x its size. We have found that this number of ‘fanout of 4’ (FO4) inverter delays is relatively insensitive to technology and operating conditions, and is a good metric for comparing designs across different technologies. In our technology an inverter in a fanout of 4 inverter chain has a delay of 430 ps under worst-case operating conditions. Table 2. shows the initial and output delays that are common among all the array architectures. Table 3.-Table 7. show the addition delays for the different array architectures discussed in this paper.

Table 2. Common Delays

Component	Delay nsec	FO4 Delay
Multiplier latch	0.77	1.79
Booth Encoder	1.74	4.04
Booth Mux	0.452	1.05
Total Initial Delay		
Output Latch	0.43	1.00
Initial + Output Delay	**3.39**	**7.87**

Table 5. Interleaved Array

Component	Delay nsec	FO4 Delay
CSA within the array	0.714	1.66
Output of 12-CSA array	8.6	20
Output of Compressor1	0.886	2.06
Addition Delay	**9.486**	**22.06**

Table 3. Full Array

Component	Delay nsec	FO4 Delay
CSA within the array	0.576	1.34
Output of 25-CSA array	14.41	33.5
Output of Compressor1	0.886	2.06
Addition Delay	**15.29**	**35.56**

Table 6. Array-of-arrays 2

Component	Delay nsec	FO4 Delay
Output of 9-CSAArray4	5.19	12.07
Output of Compressor3	0.886	2.06
Addition Delay	**6.076**	**14.13**

Table 4. Two Half Arrays

Component	Delay nsec	FO4 Delay
CSA within the array	0.576	1.34
Output of 11-CSA array	7.39	17.2
Output of Compressor1	0.886	2.06
Addition Delay	**8.86**	**20.6**

Table 7. Array-of-arrays 1

Component	Delay nsec	FO4 Delay
Output of 3-CSA Array1	2.61	6.07
Output of Compressor1	1.75	4.07
Output of Compressor2	1.815	4.22
Output of Compressor3	0.886	2.06
Addition Delay	**7.061**	**16.42**

Interleaved arrays reduce the addition time of the full arrays by 38%. The carry save outputs within an interleaved array have almost twice as much wire capacitance as the ones in a full array. Two half arrays get rid of this internal wire loading, hence each CSA stage is faster as compared interleaved arrays. This makes two half arrays faster by 6.6% in spite of the fact that it has more carry save adders in the critical path than interleaved array. Table 7. shows that even though the number of carry save additions is minimized in array-of-arrays1, due to the large wire loading at three places in the critical path, 31.4% of the total addition time in this architecture is spent in wire loadings. This illustrates the fact that minimizing the CSA count in the critical path is not the only criteria for designing an optimal architecture, wiring plays an important role in the critical path.

The best array architecture is array-of-arrays2 (Table 6.) which has no long running wires in the critical path. The array-of-arrays2 architecture gives 26% improvement over two half arrays (Table 4.). The total critical path latency for this optimal architecture is 10.03ns, which is 23.33FO4 delay. As can be seen from tables 2 and 6, only 66% of the total latency is spent in carry save additions in this architecture, as compared to 82% in a full array.

6: Conclusions

A new architecture style is presented for the design of parallel floating point multipliers. The idea is to break up the partial product addition into groups of sub-arrays and combine the results of the sub-arrays using another array. This architecture requires a modest increase in area over a full array and can be implemented without requiring additional wire pitches over an interleaved array. The partitioning of partial products depends on the size of the multiplier, as well as wire loadings. It is shown that minimizing the number of CSAs in the critical path at the expense of wire loading does not improve performance. Dividing a 53-bit multiplication into two half arrays reduces the critical path for partial product addition by 42.1%. Dividing this further into four sub arrays gives 18.2% further improvement. Since the sizes of the two array and four array multiplier are nearly the same, the four array architecture seems attractive for future multiplier designs.

7: References

[1]Gary Bewick. “Fast Multiplication: Algorithms and Implementation,” PhD Stanford University, 1994

[2]A D Booth, “A signed binary multiplication technique”, Qt. J. Mech. Appl. Math., vol. 4, Part2, 1951.5.

[3]D. Dobberpuhl, “A 200MHz 64bit Dual-Issue CMOS Microprocessor.,” 1992 IEEE ISSCC digest of technical papers, pp. 106-107.

[4]Gensuke Goto et al., “A 54x54-b Regularly Structured Tree Multiplier”, IEEE J. Solid-State Circuits, vol. 27, No. 9, Sept 19892, pp 1229-1236.

[5]Craig Heikes, “A 4.5mm^2 Multiplier Array for a 200MFLOP Pipelined Coprocessor”, 1994 IEEE ISSCC digest of technical papers, pp. 290-291.

[6]Scott Hilker, Nghia Phan and Dan Rainey, “A 3.4ns 0.8µm BiCMOS 53x53b Multiplier Tree”, 1994 IEEE ISSCC digest of technical papers, pp. 292-293.

[7]Leslie Kohn and Sai-Wai Fu, “A 1,000,000 Transistor Microprocessor”, 1989 IEEE ISSCC digest of technical papers, pp. 54-55.

[8]Junji Mori et al., “A 10-ns 54x54-b Parallel Structured Full Array Multiplier with 0.5-µm CMOS Technology”, IEEE J. Solid-State Circuits, vol. 26, No. 4, April 1991, pp 600-606.

[9]Mark R Santoro and Mark Horowitz, “SPIM: A Pipelined 64x64-bit Iterative Multiplier”, IEEE J. Solid-State Circuits, vol. 24, No. 2, april 1989, pp 491-493.

[10]C S Wallace, “A suggestion for fast multipliers”, IEEE Trans. Electron. Computers, vol EC-13, pp. 14-17, Feb. 1964.

[11]Mark R Santoro, Gary Bewick, Mark Horowitz, “Rounding Algorithms for IEEE multipliers”, Proceedings of 9th symposium on computer arithmetic, Santa Monica, CA, Sept 1989. pp 176-183.