OPTIMIZATION AND RAPID PROTOTYPING IN FPGA OF A FAST SINGLE PRECISION FLOATING POINT ADDER

Guilherme Godoi, André Luiz Aita

Microelectronics Group – Department of Electronics and Computing – DELC
Centro de Tecnologia, Universidade Federal de Santa Maria, UFSM, Brazil.

guilherme@mail.ufsm.br, aaita@inf.ufsm.br

ABSTRACT

This work presents the design of a fast single precision floating point FP adder. An 8-bit format, similar to the IEEE 754/85 standard format is implemented using the round-to-nearest rounding mode. First, the basic algorithm for addition of floating point numbers is reviewed. After, one modified algorithm, which increases the performance of the FP adder, is explained and proposed for implementation. In order to validate the algorithm and perform a rapid prototyping, the functional units were all described in VHDL, and then synthesized in FPGA. The VHDL language also allows the scalability necessary to attend the requirements of different users and applications. Modifying the exponent and mantissa sizes can set precision and range. Functional blocks were optimized to improve the adder in terms of area consumption and delay. The results obtained with the implementation and optimizations are shown and discussed.

RESUMO

Este trabalho apresenta o desenvolvimento de um somador rápido em ponto flutuante, precisão simples. Um formato de 8 bits, similar ao formato padrão da IEEE 754/85, é implementado utilizando o método arredondamento para o mais próximo. O algoritmo básico para soma de números em ponto flutuante é revisado. Após, um algoritmo modificado, que melhora o desempenho do somador em ponto flutuante é explicado e proposto para implementação. Para validar o algoritmo e realizar a prototipação rápida, as unidades funcionais foram descritas em VHDL, e então sintetizadas em FPGA. A linguagem VHDL também permite a escalabilidade necessária para atender demandas de diferentes usuários e aplicações. A precisão e intervalo de representação são ajustados modificando-se o tamanho da mantissa e do expoente.
Os blocos funcionais são otimizados para melhorar o somador em termos de área consumida e atraso. Os resultados obtidos com a implementação e otimizações são mostrados e discutidos.

OPTIMIZATION AND RAPID PROTOTYPING IN FPGA OF A FAST SINGLE PRECISION FLOATING POINT ADDER

Guilherme Godoi, André Luiz Aita

Microelectronics Group – Department of Electronics and Computing – DELC
Centro de Tecnologia, Universidade Federal de Santa Maria, UFSM, Brazil.

guilherme@mail.ufsm.br, aaita@inf.ufsm.br

ABSTRACT

This work presents the design of a fast single precision floating point adder. An 8-bit format, similar to the IEEE 754/85 standard format is implemented using the round-to-nearest rounding mode. First, the basic algorithm for addition of floating point numbers is reviewed. After, one modified algorithm, which increases the performance of the FP adder, is explained and proposed for implementation. In order to validate the algorithm and perform a rapid prototyping, the functional units were all described in VHDL, and then synthesized in FPGA. The VHDL language also allows the scalability necessary to attend the requirements of different users and applications. Modifying the exponent and mantissa sizes can set precision and range. Functional blocks were optimized to improve the adder in terms of area consumption and delay. The results obtained with the implementation and optimizations are shown and discussed.

1. INTRODUCTION

Many applications require numbers that are non-integers. If the representation interval of these numbers is wide and the required precision is high, floating point (FP) arithmetic is more suitable than fixed point one. The FP arithmetic has gained widespread use and today many general-purpose processors for digital signal processing (DSPs) implement FP arithmetic units. Unfortunately, these floating point units require excessive area and design time even for conventional implementations. Using the characteristics of VHDL that allow scalability or reusability, custom floating point formats can be derived from the IEEE 754/85 [1] format, for individual applications requirements, reducing the area consumption and increasing the speed. In [2], the floating point format was adapted to attend specifications like processor data path and memory width. The scalability allows the adjustment of different precisions and ranges, through the manipulation of the exponent and mantissa sizes.

The use of high-level languages allows the rapid prototyping in Field Programmable Gate Arrays (FPGAs). The high programmability and the increased density of FPGAs are important characteristics when referring to DSP implementations.

This paper is organized as follows. In Section 2, the floating point single precision representation is briefly reviewed. Section 3 describes the basic algorithm for addition of floating point numbers while in Section 4, a more time efficient algorithm is described. The algorithm implementation in FPGA is shown in Section 5. Improvements in the functional units to achieve a better performance of the adder are done in Section 6, and the results obtained are summarized in Section 7.

2. THE FLOATING POINT FORMAT

Figure 1 shows the single precision 32-bit floating point IEEE 754 format:

	s	*e*	*m*
Bit #:	31	30 23	22 0
Field:	Sign	exponent	mantissa

Figure 1. Floating Point IEEE 754 format

The sign bit (bit 31) represents the sign of the number. The 8-bit exponent is a signed number, using a bias of 127. The 23-bit mantissa has a leading bit implied on the representation (1.m). The value of the floating point number is given by:

$$-1^{(s)}\, 2^{(\boldsymbol{e}-127)}\, (1.\boldsymbol{m})$$

In this paper, another floating point format, similar to the IEEE 754, was used in order to allow a fast implementation and simulation of the developed algorithm shown in Section 4. This 8-bit format is composed of a 3-bit exponent, a 4-bit mantissa and the sign bit. Also, to compare the implementations of different representations, a 16-bit format with a 6-bit exponent and a 9-bit mantissa was implemented. This 16-bit format was used in [2] to a FIR filter application.

3. ADDITION/SUBTRACTION ALGORITHM

A typical floating point operation takes two inputs with *p* bits of precision and returns a *p*-bit result. According to the basic algorithm, the exact the result is first computed and then rounded to *p* bits. The rounding method used is the IEEE 754 default, the round-to-nearest mode. The addition of two floating point number (a_1 and a_2) can be divided in four stages, as follows. Figure 2 shows a block diagram of the algorithm.

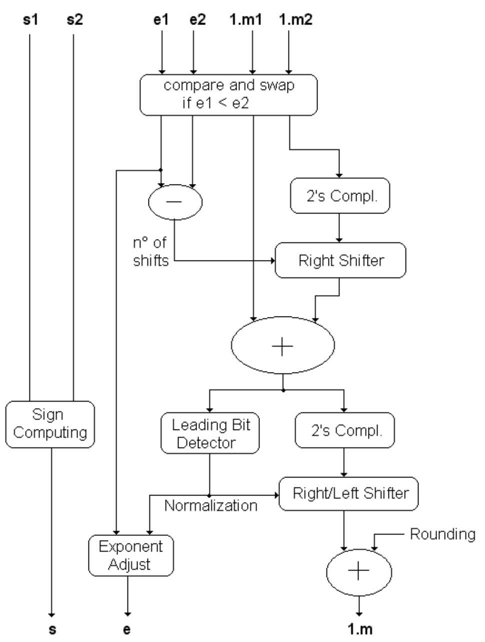

Figure 2. Block diagram of the basic algorithm

In Stage 1, the exponents are compared and if $e_1 < e_2$, the operands are swapped. This ensures the difference between the exponents to be a positive value. If the signs differ, the two's complement of m_2 is performed. After, m_2 is shifted (e_1-e_2) positions to the right, aligning the binary point. The first two bits shifted out are set to g (guard) and r (round), and the s (sticky) are set to the OR of the rest. The exponent of the result is equal to e_1.

In Stage 2, m_1 and m_2 are added (with the implied leading bit). If the result is negative, it should be replaced by its two's complement. In Stage 3, the result is shifted to the left until it's normalized. The exponent is adjusted according to the number of positions shifted. Finally, the result is rounded.

In Stage 4, the number is rounded using the round-to-nearest mode, which adds a one to the least significant bit according to the following rule: (LSB.r) + (r.s). If the rounding causes an overflow, replace the mantissa with zeros and add 1 to the exponent. Determine the sign of the result.

Although four 24-bit adders are present in the algorithm, only three sequential additions can even take place. Unfortunately, it will lead to a low performance of the floating point adder, and consume a large area (four 24-bit adders). The next section shows an improvement of this algorithm, in terms of speed and area.

4. THE MODIFIED ALGORITHM

In the modified algorithm, the three necessary additions (two's complement, mantissa sum and the rounding) are combined together and performed in just one step. Figure 3 sows a block diagram of the proposed algorithm).

Three cases are identified, which cover all possibilities: (1) numbers with same signs, (2) numbers with different signs and same exponents, and (3) numbers with different signs and different exponents. The three possible situations are identified by two bits: the signal bit (F), which indicates if the signs are different; and difference bit (D), which indicates if the difference between the exponents is zero. The major problem is to identify which operations should be done in each case and how to define the necessary bits for the rounding step, since these bits depend on the format of this number after the mantissas sum.

In each case, two possibilities are again identified. In order to cover both simultaneously, two data-path are proposed. Two concurrent assumptions are performed in parallel, and the correct answer is selected at the end of the addition. To perform all the additions in one, all the information necessary for the round-up operation, like the least significant bit of the sum and the g, r and s bits, should be determined in advance. So, the round-up can be done with the mantissas sum. After the sum, the result can be normalized .

In the first case, the addition of the mantissas may cause a carry-out or not. The first datapath assumes that no carry-out has occurred. Since the number will be normalized, no left shifts will be necessary. The least significant bit of the sum can be achieved using the least significant bits of m_1 and m_2. The second datapath cover the possibility of carry-out. One right shift normalizes the numbers, and the LSB can be obtained in advance using the two least significant bits of m_1 and m_2.

The second case is simpler than the first one, since the rounding will never occur (no left shifts). The first datapath perform ($m_1 – m_2$) while the second perform ($m_2 – m_1$). The positive answer is selected at the end of the operation.

The third case combine the two's complement of Stage 1, the mantissa sum and the rounding. The first datapath assumes a normalized result, while the second assumes the leading bit in any other position of the mantissa. When the MSB of the sum is known, it selects the correct result. As in the first case, the LSB of the result can be obtained from the least significant bits of m_1 and m_2.

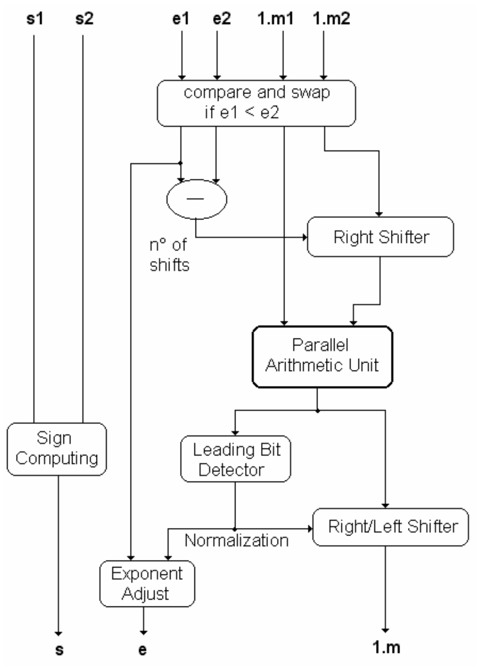

Figure 3. Block diagram of the modified algorithm

This modified algorithm performs the floating point sum in one addition, regardless of number format. Comparing to the first algorithm showed, the number of fixed-point adders was reduced, although some additional external logic was incorporated to select the operands of the two parallel adders.

5. IMPLEMENTATION

The circuits of the adder were described in VHDL, as structural components. Then a hierarchical structure was created. The scalability of VHDL allows a fast operand width adjustment, and different precisions and ranges can be easily obtained.

A magnitude comparator is employed to compare the exponents. So, the significand of the smallest number is always the right shifter input. Two subtractors are present in the FP adder. One performs exponent subtraction $d = e_1 - e_2$, determining the required number of right shifts, while the second adjusts the result exponent during the normalization step. The programmable right shifter performs the binary point alignment by shifting S_2 to the right. It also supplies the guard *g*, round *r* and sticky *s* bits (not shown in the figure) required for rounding.

The main functional unit of the FP adder is the Parallel Arithmetic Unity (*PAU*), shown in Figure 4. The *PAU* is composed of a duplicated 24-bit wide adder, a duplicated 2-bit adder and a duplicated carry-in generator. The significands to be added, and the guard, round and sticky bits from the programmable right shifter are the inputs of *PAU* . The signal $F = sig_1 \oplus sig_2$ and *D=OR-function of all bits from d* that *PAU* are used to identify the three cases. The 24-bit sum, the carry-out and the guard bit *G*, necessary to the normalization step, are the outputs of *PAU*.

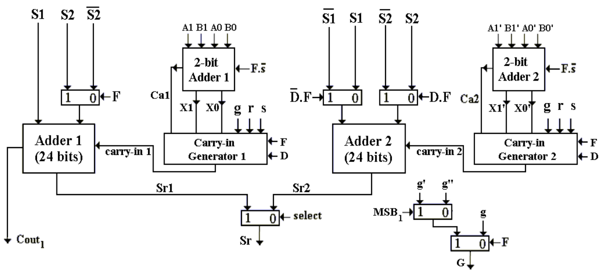

Figure 4. Parallel Arithmetic Unit - PAU

Inside the *PAU*, the 2-bit adder has an important function: generate all rounding information in advance. Depending on the case, the inputs *Ai* and *Bi* and outputs *Xi* can differ in meaning and value.

The programmable bi-directional shifter normalizes the result, after the significand addition. It can shift one bit to the right or a programmable number of bits to the left. When right shift is necessary, the high order bit of result should be filled with the carry-out. When left shift is necessary, the low order bit is filled with *g* bit in the first shift and with zeros in the next ones. The leading bit position encoder, which locates the position of the leading bit, generates the number of left shifts *N*. The exponent adjustment is performed together with the normalization, according to the number of bits shifted. If one right shift has occurred, the exponent should be incremented; otherwise, decremented of *N*.

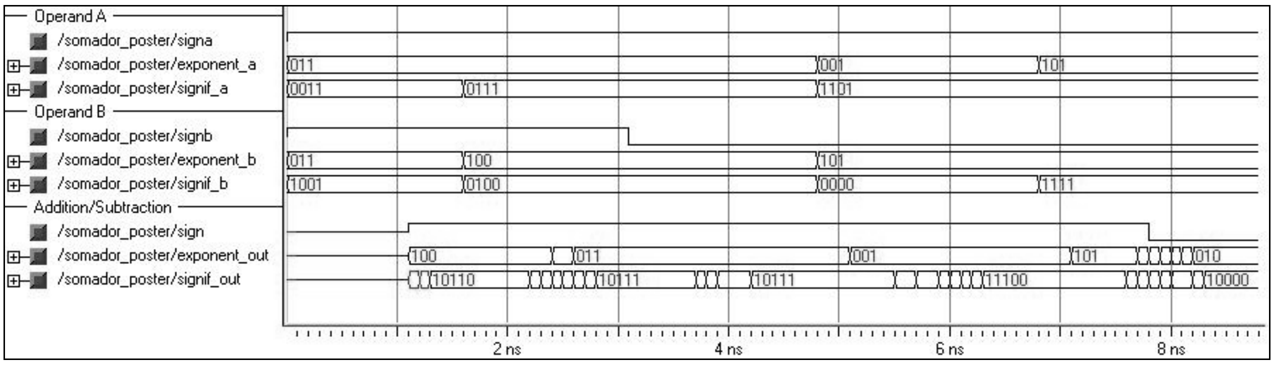

Figure 5. Post Place and Route Simulation

In the Figure 5, the complete functional simulation can be observed. The operands A and B are added/subtracted. No errors in the algorithm and in the implementation were detected.

6. OPTIMIZATIONS

When describing circuits in VHDL, careful about some aspects should be taken into account. The synthesis tools are sensitive to behavioral or structural descriptions. One example is the *for-loop* which consumes a large area, since the synthesis tool generates hardware for each iteration [3]. The *if* statement is preferred instead, increasing the speed and reducing the required area. Besides, depending on the type of adders, better synthesis may be possible, since the descriptions of each adder can be different.

The fixed-point adders represent the most time and area consuming in the floating point adder. In order to achieve faster and smaller circuits, different VHDL codes were implemented. The first implementation, taken as the reference, used a ripple-carry adder. The optimization alternatives used for fixed-point addition were: a carry-select adder and the adder provided by the synthesis tool when using the statement S<=A+B. Table 1 and 2 shows the results obtained.

Table 1. Area optimizations of the fixed-point adders (in slices)

operand size	Ripple Carry	Carry Select		Synthesis Tool	
		Slices	Opt. (%)	Slices	Opt. (%)
8	7	9	+28,57	7	0
16	15	19	+26,66	5	–66,67
32	36	46	+27,77	12	–66,67

Table 2. Delay optimizations of the fixed-point adders (in ns)

operand size	Ripple Carry	Carry Select		Synthesis Tool	
		delay	Opt. (%)	delay	Opt. (%)
8	12,577	11,982	–4,73	11,474	–8,76
16	19,763	15,318	–22,49	10,875	–44,97
32	40,595	29,094	–28,33	14,011	–65,48

The fixed-point adder provided by the synthesis tool has shown the best performance in terms of area and delay. This is due to the fact that the Spartan-II CLB supports carry logic [4]. The tool synthesizes the adders in a carry chain, optimizing the routing channel between the CLBs. However, for adders up to 7-bit wide, the synthesis tool didn't create the carry chain resulting in non-organized slices that consume an excessive area. This can be observed in Figure 6. For more than 7-bit wide, the carry chain produces an almost linear relation between area/delay and number of bits. The carry-select adder is another alternative, considering the trade-offs between consumed area and necessary speed, mainly in the case where there is no support for the carry logic.

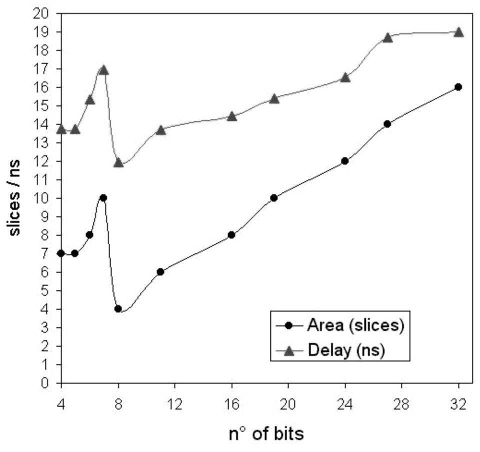

Figure 6. Area and delay of different sizes fixed-point adders provided by the synthesis tool

When implementing the adder with small size operands like the 8-bit format proposed, the difference between the ripple-carry or the synthesis adder implementation is not visible. The Figure 7 shows a comparison between different width ripple-carry and synthesis adder. For the standard IEEE 754 32-bit format, the optimization can reach 33% in area and 34% in delay.

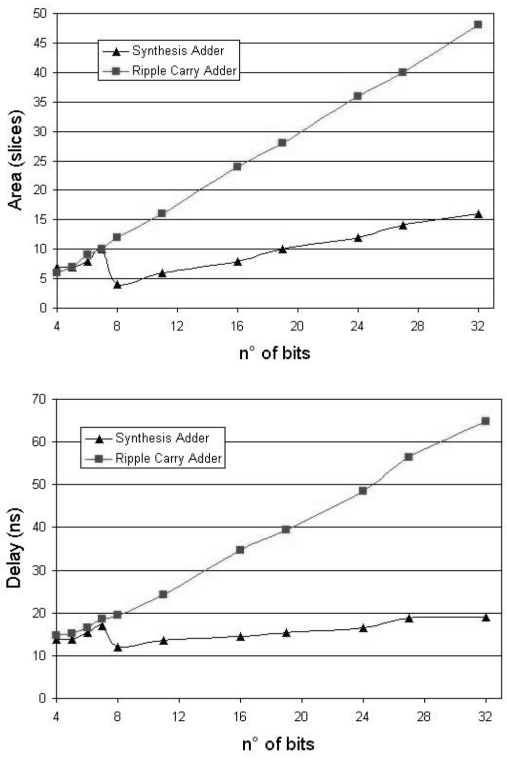

Figure 7. Area and delay comparison between ripple-carry and the synthesis tool adder

7. RESULTS

The 8-bit format was chosen for validation of the algorithm proposed in Section 5. This format reduces the number of inputs and outputs, which can ease the testing procedures. The FPGA targeted for implementation was the Xilinx xc2s200-5pq208 of the Spartan II family. The synthesis software used in this work was the FPGA Express from Synopsys, available into the Xilinx ISE 4.2i environment.

Due to the small size of operands used in the validation, improvements could not be observed. Optimization results start to appear for wider operands. Table 3 summarizes the implementation results.

Table 3. Optimizations of the 8-bit format Floating Point adder

	Non-Optimized	Optimized	Optim. (%)
Area (slices)	81	81	0
Delay (ns)	49,069	50,046	+1,95
Type Delay (ns)	22.152 logic 26.917 route	20.846 logic 29.200 route	–5,895 +8,481
Frequency (MHz)	20,37	19,98	+1,90
Logic Levels	26	24	–7,69
Critical Path	exp_A(0) to signif_out(4)	exp_A(0) to signif_out(1)	-

8. CONCLUSIONS

A fast single precision floating point adder was developed and prototyped, using a reduced format of 8-bit operands, similar to the standard IEEE 754 32-bit operand. An optimization at algorithm-level was adopted, reducing the number of sequential fixed-point additions. The proposed algorithm uses a Parallel Arithmetic Unit, which employs two parallel datapaths. So, the four fixed-point adders present in the first algorithm showed are reduced to two adders and the worst case delay was reduced from three to one fixed-point addition.

The circuits of the floating point adder were described in VHDL as structural blocks and placed in a hierarchical structure. Using the scalability of a high-level language, different floating point formats can be achieved to fit the needs of different applications. The exponent and the mantissa sizes are adjusted to obtain different precisions and ranges. Optimizations in the VHDL codes were done to increase the speed and reduce the area consumed. One of the most time and area consumption blocks is the fixed-point adder. Different architectures were tested, some of them reducing the area and others reducing the delay. The adder with better performance is the adder produced by the synthesis tool. The Spartan II FPGA supports carry logic, resulting in small and fast adders. The optimizations in the adders reduced the logic delay and increased the route delay, resulting in almost the same total delay. For larger operands sizes, the optimizations in area and delay can reach reasonable values.

9. REFERENCES

[1] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE754-1985, New York, 1985.

[2] N. Shirazi, A. Walters, and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines," *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, 1995.

[3] I. O. Flores, M. Jimenez and D. Rodriguez, "Optimizing the Implemntation of Floating Point Units for FPGA Synthesis", Computing Research Conference CRC2002, Mayagüez, Puerto Rico, 2002.

[4] Xilinx Inc. *Spartan-II 2.5V FPGA Family: Functional Description*, 2001.

[5] C. H. Ho, M. P. Leong, J. Becker, M. Glesner, "Rapid Prototyping of FPGA Based Floating Point DSP Systems", 13th IEEE International Workshop on Rapid System Prototyping (RSP'02), 2002 Darmstadt, Germany, 2002.

[6] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, D. Poirier, "A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs", *FPGA`02*, Monterey, California, USA, 2002.

[7] G. Godoi, F. D. Franke and A. L. Aita, "Design of a Fast Single Precision Round-to-Nearest Floating-Point Adder",*XVII SIM – Simpósio Sul de Microeletrônica*, Canela, RS, 2002.

[8] M. Glesner and A. Kirschbaum, "State-of-the-Art in Rapid Prototyping", *SBCCI 98 – XI Brazilian Symposium on Integrated Circuit Design*, Búzios, RJ, Brazil, 1998.

[9] A.Beaumont-Smith, N.Burgess, S. Lefrere and C.C. Lim, in "Reduced Latency IEEE Floating-Point Standard Adder Architectures", pp. 35-42, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia 1999.

[10] M. R. Santoro, G. Bewick, and M. A. Horowitz, in "Rounding Algorithms for IEEE Multipliers", pp. 176-183, *Proceedings of 9th Symposium on Computer Arithmetic*, 1989.

[11] L. Kohn and S.-W. Fu, in "A 1,000,000 transistor processor", pp. 54-55. *IEEE Int'l Solid-State Circuits Conf.*, 1989.

[12] D. A. Patterson, in "Computer Architecture: A Quantitative Approach, J. L. Hennessy", 2nd. Edition, pp. A.13-28, Academic Press, 1996.

[13] D. A. Patterson, in Organização e Projeto de Computadores - A Interface Hardware/ Software, J. L. Hennessy, 2nd Edition, pp. 160-166, Ed. Livros Técnicos e Científicos,(2000.

[14] P. M. Seidel, in "How to half the latency of IEEE compliant Floating-point Multiplication". In *Proceedings of the 24th Euromicro Conference*, volume 24. IEEE,