FPGA Implementations of BCD Multipliers

G. Sutter[1], E. Todorovich[1], G. Bioul[2,3], M. Vazquez,[2,3], J-P. Deschamps[2,4]
[1]Universidad Autónoma de Madrid, Spain [2]FASTA University, Mar del Plata, Argentina;
[3]UNCPBA University, Tandil, Argentina [4]Rovira I Virgili University, Tarragona, Spain;
gustavo.sutter@uam.es; etodorov@uam.es; gbioul@ufasta.edu.ar; mvazquez@exa.unicen.edu.ar;
jeanpierre.deschamps@urv.cat.

***Abstract*—This paper presents a number of approaches to implement decimal multiplication algorithms on Xilinx FPGA's. A variety of algorithms for basic one by one digit multiplication are proposed and FPGA implementations are presented. Later on N by one digit and N by M digit multiplications are studied. Time and area results for sequential and combinational implementations show better figures compared with previous published work. Comparisons against binary fully-optimized multipliers emphasize the interest of the proposed design techniques.**

Keywords-BCD arithmetic; IEEE-745 standard

I. INTRODUCTION

Recently, a growing attention has been paid to decimal computer arithmetic for its impact on scientific, commercial, and financial applications [1]. Moreover, the new revision of the IEEE-745 Standard for Floating-Point Arithmetic includes specifications for decimal formats [2]. The key advantage of decimal system rests upon a major precision than what binary can get representing decimal numbers. As a matter of fact, binary round up and truncate operations add up errors in the final results. With current technology, processing speed for decimal computations can be improved at reasonable hardware cost. This motivates the interest towards hardware realizations of decimal arithmetic algorithms and their implementations within the available binary technology. Hardware decimal arithmetic units serve as an integral part of some recently commercialized general purpose processors [3-4], where complex decimal arithmetic operations, such as multiplication or division, have been realized by iterative algorithms. Recent fast decimal arithmetic units are proposed in the literature [5-12]. As the electronic devices at hand are basically binary, the binary-coded decimal (*BCD*) encoding is the most popular, and still appears to be the best choice, though other binary inspired coding systems could be considered. Recently, *BCD* addition has been given some more attention in the literature [5-8], while other operations appear mainly open for further developments towards fast and affordable hardware designs.

This paper presents sequential and combinational approaches for *BCD* multiplier units implemented on reconfigurable devices such as Xilinx Virtex 4 [13] and Virtex 5 families. Comparative figures of merit are presented and discussed. Section 2 presents some algorithmic alternatives to implement a 1x1 *BCD* digit multiplier. A first technique is based on a binary multiplication followed by a correction stage [9]. Then some *ROM*-based designs are analyzed. Finally, multiplexer-based solutions are presented. Section 3 proposes an implementation for an *N*x1 *BCD* multiplier. This operation is later carried out iteratively to provide an *N*x*M* multiplier, but by itself, the *N*x1 multiplication appears to be a primitive for other algorithms, such as logarithm or exponential functions [5]. Section 4 deals with the *N*x*M* multiplication where combinational and sequential implementations are presented. In Section 5 performance and area comparisons are shown. Finally, Section 6 briefly sums up some conclusions.

II. ONE-DIGIT × ONE-DIGIT BCD MULTIPLICATION

A. Binary arithmetic with correction

The decimal product can be obtained through a binary product and a post correction stage. In [9] this multiplication is implemented through a combinational circuit. In what follows it is implemented on an FPGA platform.

Let A and B be two *BCD* digits ($a_3\ a_2\ a_1\ a_0$) and ($b_3\ b_2\ b_1\ b_0$) respectively. The *BCD* coded product consists of two *BCD* digits D and C such that

$$A*B = D*10 + C \tag{1}$$

$A * B$ is first computed as a 7-bit binary number $P(6:0)$ such that

$$A*B = P = p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0 \tag{2}$$

Although a classic binary-to-*BCD* decoding algorithm can be used, it can be shown that the *BCD* code for P can be computed through binary sums of correcting terms described in Fig. 1. The first row in Fig. 1 shows the *BCD* weights. The weights of p_3, p_2, p_1 and p_0 are the same as those of the original binary number "$p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$". But weights 16, 32 and 64 of p_4, p_5, and p_6 have been respectively decomposed as (10, 4, 2), (20, 10, 2) and (40,

978-0-7695-3917-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ReConFig.2009.28

20, 4). Observe that "$p_3\ p_2\ p_1\ p_0$" could violate the interval [0, 9], then an additional adjust could be necessary.

First the additions of Row 1, 2, 3, and correction of "$p_3\ p_2\ p_1\ p_0$" are completed (least significant bit p_0 is not necessary in computations) are computed. Then the final correction is computed.

	80	40	20	10	8	4	2	1
		p_6	p_5	p_4	p_3	p_2	p_1	p_0
+			p_6	p_5		p_4	p_4	
						p_6	p_5	
	d_3	d_2	d_1	d_0	c_3	c_2	c_1	c_0

Fig. 1. Binary to BCD arithmetic reduction

One defines (Arithmetic I):

1. the binary product $A{\cdot}B = P = p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

2. the Boolean expression: $adj_1 = p_3 \wedge (p_2 \vee p_1)$,

3. the arithmetic sum: $dcp = p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0 + 0\ p_6\ p_5\ 0\ p_4\ p_4\ 0 + 0\ 0\ 0\ 0\ p_6\ p_5\ 0 + 0\ 0\ 0\ 0\ adj_1\ adj_1\ 0$,

4. the Boolean expression: $adj_2 = (dcp_3 \wedge (dcp_2 \vee dcp_1)) \vee (p_5 \wedge p_4 \wedge p_3)$.

One computes

$$dc = dcp + 0\ 0\ 0\ 0\ adj_2\ adj_2\ 0. \quad (3)$$

Then

$D = dc_7\ dc_6\ dc_5\ dc_4$ and $C = dc_3\ dc_2\ dc_1\ dc_0$

A better implementation can be achieved using the following relations (Arithmetic II):

1. the product $A{\cdot}B = P = p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

2. compute:

$cc = p_3\ p_2\ p_1\ p_0 + 0\ p_4\ p_4\ 0 + 0\ p_6\ p_5\ 0$,

$dd = p_6\ p_5\ p_4 + 0\ p_6\ p_5$

(cc is 5 bits, dd has 4 bits, computed in parallel)

3. define:

$cy_1 = 1$ iff $cc > 19$, $cy_0 = 1$ iff $9 < cc < 20$

(cy_1 y cy_2 are function of $cc_3\ cc_2\ cc_1\ cc_0$, and can be computed in parallel)

4. compute:

$C = cc_3\ cc_2\ cc_1\ cc_0 + cy_1\ (cy_1$ or $cy_0)\ cy_0\ 0$, (4)

$D = dd_3\ dd_2\ dd_1\ dd_0 + 0\ 0\ cy_1\ cy_0$

(C and D calculated in parallel)

Compared with the first approach, the second one requires smaller adders (5 and 4-bit vs 8-bit) and the adders can operate in parallel as well.

B. Using ROM

Actually a (100×8)-bit *ROM* can fit to store all the $A{\cdot}B$ multiplications. However, as A and B are two 4-bit operands the product can be mapped into a 2^8×8-bit *ROM*. In Xilinx devices two main possibilities are considered: *BRAM*'s or distributed *RAM*'s (LUT based implementation of *RAM*'s).

***BRAM*-based implementation**

Block *RAM*'s are 18-Kbit configurable and synchronous dual-port *RAM* blocks. They can be configured into different layouts. The 2^{11}×8-bit configuration option has been selected, though wasting some memory capacity. As *BRAM* is dual-port, two one-by-one digit multiplications can be implemented in a single *BRAM* and in a single clock cycle. However the main characteristic to cope with is that *BRAM*'s are synchronous, so either the address or the output should be registered.

Distributed *RAM* (*LUT*-based) version 1

4-input *LUT*'s can be configured as 16×1-bit RAM's. Then the 2^8×8-bit ROM can be implemented using *LUT*'s.

Distributed *RAM* (*LUT*-based) version 2

The computation of the *BCD* final result components $C(0) = dc_0$ and $D(3) = dc_7$ are straightforward. Actually

$$C(0) = a_0 \wedge b_0 \quad (5)$$

and

$$D(3) = a_0 \wedge b_0 \wedge a_3 \wedge b_3. \quad (6)$$

Formula (5) is related to the parity while (6) emphasizes that bit $D(3)$ is set for one case only (decimal 9×9 = 81). It is thus possible to reduce the required memory size to a 2^8×6-bit *ROM* only.

Comments

1. *BRAM*-based design is fast but synchronous. It is useless for combinational implementations, but suitable for sequential and pipelined ones.

2. The existence of don't care conditions in the memory definition allows the synthesizer to reduce the effective memory request. Observe in Table 1 that only 71 *LUT*'s are effectively used. For a full 2^8×8-bit *ROM,* 128 (8×16-bit) *LUT*'s should be necessary.

C. Using multiplexers

Another alternative consists of a low-level definition of the component to be used (Fig. 2). The proposed circuit uses multiplexer *muxfx* with negligible interconnection delays.

For $C(0)$ and $D(3)$, only a 2-*LUT* and a 4-*LUT* respectively are necessary, according to formulas (5, 6). For the other 6 outputs, a circuit such as the one shown in Fig. 2 can be used. It requires one 2-*LUT*, two 4-*LUT*'s and one 6-*LUT*. If we use a 4-*LUT* based FPGA (all but the Virtex 5 and Virtex 6) the 6-*LUT* has to de decomposed into four 4-*LUT*'s, two *muxf5* and one *muxf6*.

The proposed decomposition is based on the following observation. If $a_3 = 1$ then $a_2 = a_1 = 0$. Similarly, if $b_3 = 1$ then $b_2 = b_1 = 0$. In some output-bit cases, the bit-functions

are simpler; hence some LUT's may be simplified further on.

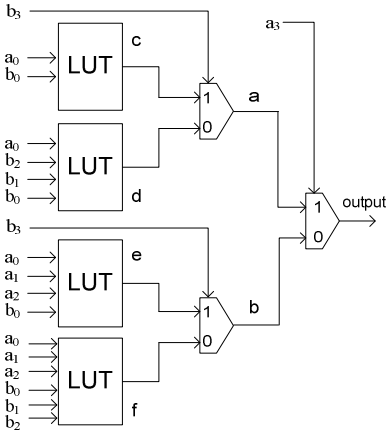

Fig. 2. Circuit to compute C(1), C(2), C(3), D(0), D(1) and D(2) of one-by-one *BCD* digit multiplier

D. Summary of one-by-one digit multiplication

Delay and area of the implementations presented in this section are summarized in Table 1 for a Virtex 4 FPGA, speed grade -11. The BRAM-based multiplier is the fastest cell for sequential and pipelined implementations. The *mux*-based cell is the best choice for combinational multipliers.

Table 1. Results of *BCD* 1×1 multiplier, Virtex 4 implementations

Implementation	Delay (ns)	Area	
		# *LUT*	# *BRAM*
Arithmetic I	7.8	47	-
Arithmetic II	4.3	37	-
BRAM ROM's	2.0	-	½
LUT based *ROM*	3.3	71	-
Mux	2.6	36	-

III. N×1 BCD DIGIT MULTIPLIER

An *N*×1 *BCD* digit multiplier is readily achieved through *N* 1×1-digit multiplications followed by a *BCD* decimal addition. Figure 3 shows how the partial products are arranged to feed the *BCD N*-digit adder stage. An efficient fast carry-chain adder circuit can be used [6, 7].

A. N×1 implementation results

Comparative figures of merit (min. clock period T, and area in number of LUTs) of the *N*×1 multiplier are shown in Table 2, for several values of *N*; Virtex 4 devices, speed grade -11 have been used. Results correspond to *mult by mux* cells (Section 2.3) and *BRAM*-based cells (Section 2.2.a.).

Table 2. Results of *BCD* N×1 multipliers using "mult by mux" cells and *BRAM* cells on Virtex 4

N	"*mult by mux*" cells		*BRAM*-based cells		
	T ns	# *LUT*	T ns	# *LUT*	# *BRAM*
4	6.4	237	5.4	80	2
8	6.9	471	6.0	158	4
16	7.8	946	6.5	320	8
32	8.4	1890	7.1	641	16

IV. N×M DIGITS MULTIPLIER

Full combinational and sequential implementations of *N*×*M* digits multipliers have been carried on.

For synthesis and implementation, XST [14] and Xilinx ISE 10.1 tool [15] have been used respectively. Same pin assignments, preserving hierarchy option, speed optimization and timing constraints, were part of the design strategy. Area and delay figures are presented in Tables 3 to 5, as provided by Xilinx reports for Virtex 4, speed grade -11 devices.

A. Combinational implementations

For the implementation of the *N*×*M*-digit multiplier, the *N*×1 *mux*-based multiplication stage has been replicated *M* times: it is the best choice because *BRAM*-based multipliers are synchronous. Partial products are inputs to an addition tree (Fig. 4). For all *BCD* additions the fast carry-chain adder of [6, 7] has been used.

Input and output registers have been included in the design. Delays include *FF* propagation and connections. The amount of *FF*'s actually used is greater than 8∗(*M*+*N*) because the ISE tools [18] replicate the input register in order to reduce fan-outs. The most useful data for area evaluation is the number of *LUT*'s.

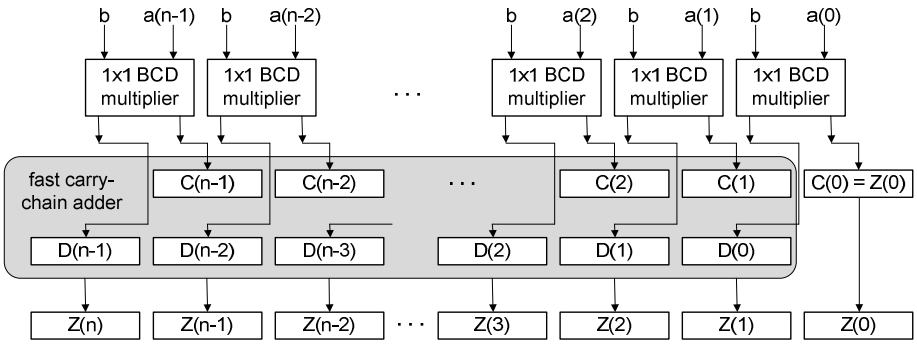

Fig 3. N by one digit circuit multiplication. It includes *N* one by one digit multipliers and a fast carry-chain adder

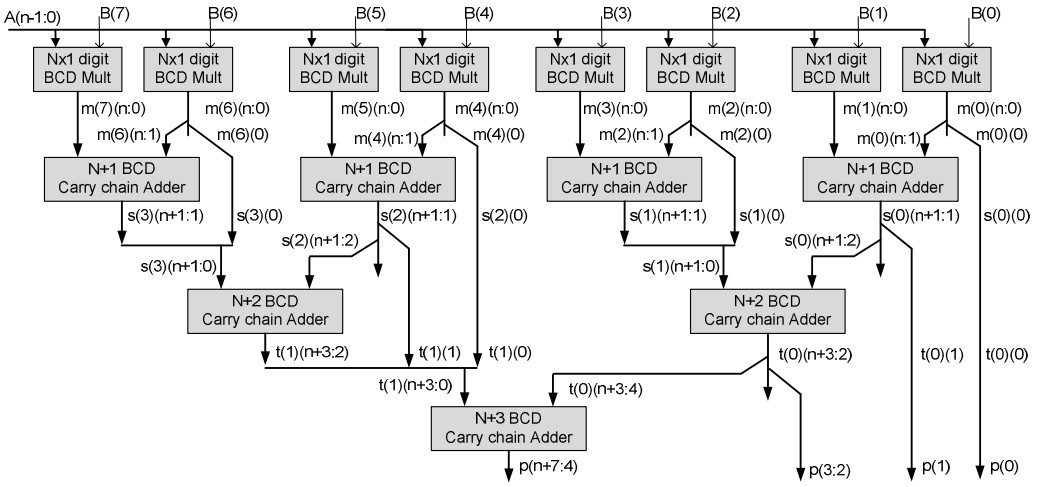

Fig 4. A example of a paralell $N \times M$ multiplier ($N \times 8$)

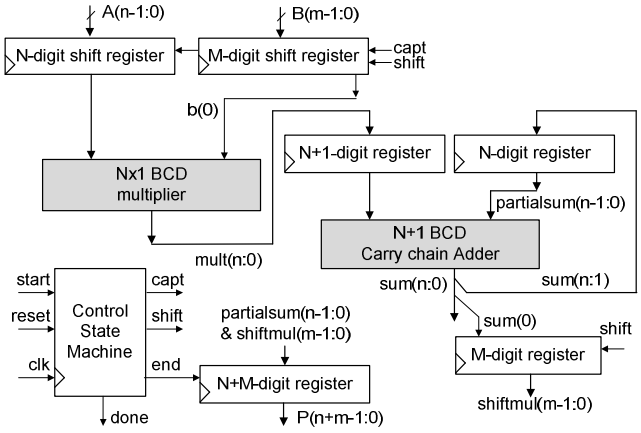

Fig. 5. A sequential $N \times M$ multiplier

Table 3. Results of combinational implementations of $N \times M$ multipliers (*mux*-based cell) on Virtex 4

N	M	Delay ns	# *FF*	# *LUT*	# slices
4	2	10.6	89	562	575
8	2	11.6	161	1,149	1,160
16	2	12.9	305	2,309	2,321
32	2	13.8	574	4,614	4,632
4	4	13.5	163	1,239	1,230
8	4	15.5	294	2,498	2,481
16	4	18.2	472	5,199	4,932
32	4	19.5	910	10,429	9,959
4	8	18.7	264	2,609	2,504
8	8	19.8	490	5,378	5,060
16	8	22.9	932	10,861	10,277
32	8	24.8	1,685	21,783	20,526
16	16	26.9	1,870	22,033	20,917
32	16	32.7	3,623	42,982	41,501

B. Sequential implementations

The sequential circuit multiplies N digits by 1 digit per clock cycle. In order to speed up the computation, the addition of partial results is processed in parallel, but one cycle later (Fig. 5).

Results for sequential implementation using "*mult by mux*" cells for 1×1 *BCD* digit multiplication are given in Table 4. If the *BRAM*-based cell is used, shorter periods (T), between 1.0 ns and 1.9 ns, can be achieved. Results for *BRAM*-based cells are provided in Table 5. As it can be observed, the fastest multipliers are the ones based on the fastest BRAM-based cell.

Table 4. Results of sequential implementations of $N \times M$ multipliers using *mult by mux* on Virtex 4.

N	M	T ns	# *FF*	# *LUT*	# slices	# cycles	delay ns
4	4	6.8	139	422	379	5	34.0
8	4	7.5	217	790	716	5	37.5
16	4	8.5	373	1,518	1,384	5	42.5
4	8	6.8	188	473	413	9	61.2
8	8	7.5	266	841	748	9	67.5
16	8	8.5	422	1,569	1,418	9	76.5
32	8	8.9	725	3,026	2,773	9	80.1
4	16	6.8	285	572	480	17	115.6
8	16	7.5	363	940	815	17	127.5
16	16	8.5	519	1,669	1,486	17	144.5
32	16	8.9	822	3,125	2,831	17	151.3
16	32	8.5	712	1,863	1,626	33	280.5
32	32	8.9	1,015	3,320	2,976	33	293.7

V. RESULTS COMPARISON

Not too many results have been reported recently for decimal operations in *FPGA*. Neto *et al*. [8] present results for 3×3-, 4×4- and 5×5-digit multipliers: in particular a 5×5 multiplier has a 10-cycle latency at 200MHz, i.e. 50 ns for an implementation using 811 LUT's and one embedded

binary multiplier in a Virtex 4, speed grade -12. In almost the same time and requiring less LUT's the BRAM-based sequential implementation presented in this paper can multiply 8×8 digits.

Table 5. Results of sequential implementations of *N*×*M* multipliers using *BRAM* cell on Virtex 4

N	M	T ns	# *FF*	# *LUT*	# slices	# *BR*	Cycles	Delay ns
4	4	5.7	121	269	168	2	5	25.5
8	4	5.9	185	480	300	4	5	29,5
16	4	6.5	313	896	556	8	5	32,5
4	8	5.7	170	320	202	2	5	28,5
8	8	5.9	234	531	334	4	9	53.1
16	8	6.5	362	947	591	8	9	58.5
32	8	7.0	618	1,780	1,122	16	9	63,0
4	16	5.7	267	418	269	2	17	96,9
8	16	5.9	331	629	400	4	17	100.3
16	16	6.5	459	1,045	658	8	17	110.5
32	16	7.0	715	1,878	1,188	16	17	119.0
16	32	6.5	652	1,242	791	8	33	214.5
32	32	7.0	908	2,074	1,317	16	33	231.0

Table 6. Results comparison against a binary multiplier core in Virtex 4

Multiplier	# slices	Other Resources	Cycles	Delay ns
Core Generator, 53×53 bits	279	16 DSP	18@400MHz	45.0
Sequential 16×16, *BRAM* cell	658	8 BRAM	17@153MHz	110.5
Combinational 16×16, *mux*-based cell	20,917	-	-	26.9

Another way to appraise the results is by matching them against a binary multiplier. Xilinx provides in Core Generator [16] a binary multiplier for up to 64×64 bits. One can compare the 16×16 BCD digits multipliers of Table 3 and 5 to a 53×53 bits binary multiplier covering the same operational range. Note that additional BCD to binary and binary to BCD converters are required for the binary multiplier. These converters are not included in the time and area results.

In order to observe area-delay trade-off Fig. 6 and 7 show results for 4×4 and 16×16 operand sizes respectively. Area is expressed in slices; neither *BRAM*'s nor *DSP* blocks are represented. Fig. 6 presents combinational (comb), sequential implementation based on *mult by mux* (seq_mux), sequential based on *BRAM* multiplication (seq_bram), and Neto *et al*. [8] implementations for 4×4 digits multiplications. Fig. 7 shows results for 16×16 multiplier circuits.

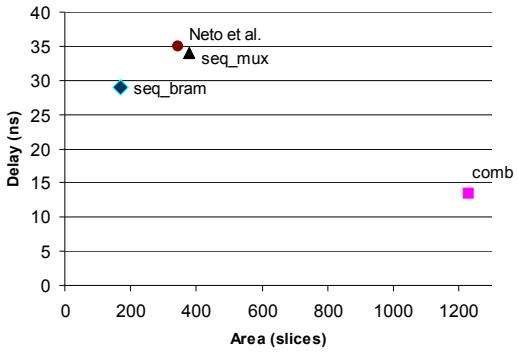

Fig. 6. Area–delay for diferent 4×4 digits multiplier

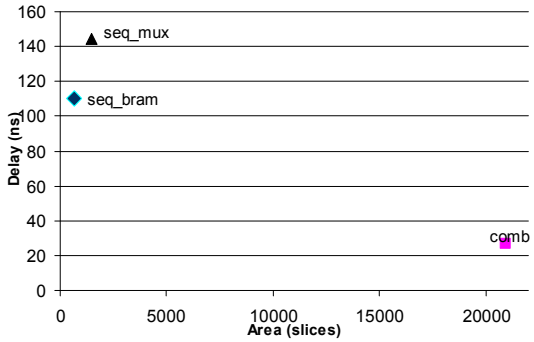

Fig. 7. Area–delay for diferent 16×16 digits multiplier

VI. Conclusion

This paper has reviewed a number of approaches to implement decimal multiplication algorithms on Xilinx devices.

The developed implementations take benefit from two important features: (i) the efficient multiplication of digit by digit using either embedded *BRAM*'s (Section 2.2) or a low level implementation with *LUT*'s and *muxfx* multiplexers (Section 2.3), and (ii) the use of fast adders [5, 6] for the adding tree.

Combinational and/or sequential approaches have to be considered to cope with the traditional area-delay trade-off. If small size operands are at hand, a combinational implementation is a reasonable option; for large operands, sequential circuits reduce area with a generally acceptable delay penalty within design constraints.

Acknowledges

This work has been partially granted by the CICYT of Spain under contract TEC2007-68074-C02-02/MIC. EDA Tools and development boards were provided by Xilinx Inc. and Mentor Graphics through University Program agreements.

References

[1] M.F. Cowlishaw: "Decimal floating-point: algorithm for computers," Proc. 16th IEEE Symposium on Computer Arithmetic, June 2003, pp. 104–111.

[2] IEEE Standard for Floating–point Arithmetic. IEEE Standards Committee, Oct. 2006.

[3] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw: "Decimal floating-point support on the IBM System z10 processor", IBM Journal of Research and Development, Volume 53, Number 1, 2009.

[4] F.Y. Busaba, C.A. Krygowski, W.H. Li, E.M. Schwarz and S.R. Carlough: "The IBM Z900 decimal arithmetic unit," Asilomar Conf. on Signals, Systems and Computers, November 2001, vol. 2, pp. 1335–1339.

[5] J.-P. Deschamps, G. Bioul and G. Sutter "Synthesis of Arithmetic Circuits, - ASIC, FPGA, Embedded Systems," John Wiley & Sons, New-York, February, 2006.

[6] G. Bioul, M. Vazquez, J.-P. Deschamps, G. Sutter: "Decimal Addition in FPGA", V Southern Conference on Programmable Logic (SPL09). Sao Carlos, Brazil, April 1-3, 2009.

[7] M. Vazquez, G. Sutter, G. Bioul, J P. Deschamps, "FPGA Xilinx 6-LUT implementations of BCD adders/subtractors," internal Report, Univ. Fasta, Mar del Plata, Argentina, feb. 2009.

[8] H. C. Neto and M. P. Véstias, "Decimal multiplier on FPGA, using embedded binary multipliers," International Conference on Field Programmable Logic and Applications, 2008, pp.197-202, Sept. 2008.

[9] G. Jaberipur and A. Kaivani, "Binary-coded decimal digit multiplier," IET Comput. Digit. Tech. 2007,1, (4), pp377-381.

[10] M. A. Erle and M. J. Schulte, "Decimal multiplication via carry-save addition," in Proc. IEEE 14th IEEE Int. Conf. Application Specific Systems, June 2003, pp. 348–358.

[11] M. J. S. R. D. Kenney and M. A. Erle, "High-frequency decimal multiplier," in Proc. IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, Oct. 2004, pp. 26–29.

[12] T. Lang and A. Nannarelli, "A radix-10 combinational multiplier," in Proc. IEEE 40th Int. Asilomar Conf. on Signals, Systems, and Computers, Oct. 2006, pp. 313–317.

[13] Xilinx Inc. "Virtex 4 User Guide," 2008, available in http://www.xilinx.com.

[14] Xilinx Inc. "XST User Guide 10.1i," 2008, available in http://www.xilinx.com.

[15] Xilinx Inc. "ISE 10.1 Documentation," 2008, available in http://www.xilinx.com.

[16] Xilinx inc. Multiplier v10.1 DS255 April 25, 2008