A FPGA IEEE-754-2008 DECIMAL64 FLOATING-POINT MULTIPLIER

Carlos Minchola
School of Engineering
Universidad Autónoma de Madrid
Madrid, Spain
e-mail: car.minchola@estudiante.uam.es

Gustavo Sutter
School of Engineering
Universidad Autónoma de Madrid
Madrid, Spain
e-mail: gustavo.sutter@uam.es

***Abstract*—this paper describes the design and implementation of a hardware module to calculate the decimal floating-point (DFP) multiplication compliant with the current IEEE-754-2008 standard. The design proposed is made up of independent stages: IEEE-754 coder / decoder, decimal multiplier and rounding. The decimal multiplication is based on a previously designed BCD multiplier. The novelty is the design of a combinational and sequential architecture for rounding stage. Time performances and hardware requirements results are reported and evaluated. A decimal64 multiplication is able to be performed in 66 ns in a Virtex 4. The DFP multiplication presented supports operations on the decimal64 format and it is easily extendable for the decimal128 format. To the best of author's knowledge, this is the first publication to present an IEEE 754-2008 multiplier in FPGA.**

Keywords: decimal arithmetic; multiplication; IEEE 754-2008; decimal floating point

I. INTRODUCTION

Decimal arithmetic is useful in many commercial and financial applications, which process decimal values and perform decimal rounding. However, current software implementations are prohibitively slow [1], prompting hardware manufacturers such as IBM to add decimal floating point (DFP) arithmetic support to their microprocessors [2]. Furthermore, the IEEE has developed the newly IEEE 754-2008 [3] standard for Floating-Point Arithmetic adding the decimal representation to the IEEE 754-1985 standard.

There are several recent works focused on fixed-point multiplication and DFP multiplication in ASICs. Generally, this designs use a sequential approach of iterating over the digits of the multiplier and others using parallel implementations [4,5,6,7]. Nevertheless, recently appears the first publications in decimal arithmetic applications [8,9] in FPGA.

The outline of the paper is as follows. In the next section, we describe the background information on decimal floating-point and the decimal multiplication is described. In section III the general structure of the multiplier is explained. In Section IV presents the design to IEEE-754 coder / decoder following the IEEE-754-2008 standard. The design is fully combinational. In Section V deals with the multiplier hardware which performs a high speed computation based on pipelined techniques. In Section VI proposes combinational and sequential design of shifting significand and rounding. Finally, in Section VII presents the results and finally, Section VIII briefly sums up some conclusion.

II. DECIMAL FLOATING POINT IN IEEE 754-2008

The new IEEE 754-2008 [3] standard specifies formats for both binary floating-point (BFP) and decimal floating-point (DFP) numbers. The primary difference between the two formats, besides the radix, is the normalization of the significands (also coefficient or mantissa). BFP significands are normalized with the radix point to the right of the most significant bit (MSB), while DFP mantissa are not required to be normalized and are represented as integers.

The IEEE 754-2008 standard specifies DFP formats of 64, and 128 bits. A DFP number contains a sign bit, an integer significand with a precision of p digits, and a biased exponent q. The value of a finite DFP number is:

$$D = -1^{S} * C * 10^{q}, q = E - bias \ . \qquad (1)$$

where s is the sign bit, C is the non-negative integer significand, and q the exponent. The exponent q is obtained as a function of biased non-negative integer exponent E.

The mantissa is encoded in Densely Packed Decimal (DPD) ([3] section 3.5.2), the exponent must be in the range [emin, emax], when biased by bias. Representations for infinity and Not-a-Number (NaN) are also provided.

Representations of floating-point numbers in the decimal interchange formats [3] are encoded in k bits in the following three fields (figure 1):

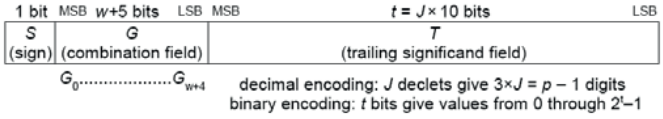

Figure 1. Decimal interchange floating-point format (from [3])

a) 1-bit sign s. b) A w + 5 bit combination field G encoding classification and, if the encoded datum is a finite number, the exponent q and four significand bits (1 or 3 of which are implied). The biased exponent E is a w + 2 bit quantity q + bias, where the value of the first two bits of the biased exponent taken together is either 0, 1, or 2. c) A t-bit trailing significand field T that contains $J \times 10$ bits and contains the bulk of the significand. J represents the number of declets.

978-0-7695-3917-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ReConFig.2009.34

When this field is combined with the leading significand bits from the combination field, the format encodes a total of $p = 3 \times J + 1$ decimal digits.

In this paper the decimal64 multiplication is developed, the values of *k*, *p*, *t, w*, and *bias* for decimal64 interchange formats are 16, 50, 12, and 398 respectively. That means that number has p = 16 decimal digits of precision in the significand, an unbiased exponent range of [−383, 384], and a bias of 398.

The non-normalized significand of DFP allows multiple (redundant or cohort) representations of numbers.

For example, multiplying 50×10^{12} by 23×10^{-5} yields a result that could be represented as or 115×10^8, 1150×10^7, 11500×10^6, 115000×10^5, etc. Because of the possibility of multiple representations, IEEE 754-2008 defines a preferred exponent, which for multiplication is $qE = qA + qB$. In example the correct result is 1150×10^7.

The multiplier uses the preferred exponent when encoding the result of a multiplication, so long as this does not lead to a loss of precision. However, in the case of multiplying 7654321×10^{10} by 2×10^5 with a precision of $p = 7$ digits, the result of 1530864×10^{16} is chosen to ensure there is no loss of precision.

III. Floating Point Multiplier Design

A general overview of proposed multiplier is presented in figure 2. Arrows are used to show the direction of data flow and the dashed blocks indicate the main stages of the design.

Initially, the IEEE-754 decoder reads two IEEE-754 standard operands (A, B) to produce the corresponding signs (SA, SB), binary exponents (EA, EB) and 16 digits binary coded decimal (BCD) significands (CA, CB). In parallel detects the special cases (SC) represented by: infinity and non-a-number (signaling NaN, and quiet NaN). As soon the decoded significands (CA, CB) become available the multiplication begins. If any operand is NaN then straightaway the DFP multiplication is NaN.

Simultaneously the sign and intermediate exponent (IE) are calculated, the first of them comes from a XOR gate operation (S = SA xor SB) and the second one results to add the biased exponent (IE = EA + EB – DECBIAS).

In FPGA implementation the multiplier is a sequential circuit that needs several clock cycles. At the end, the multiplier stage generates a result of 32-digit BCD called intermediate product (IP), this is reduced to 16-digit BCD according to the IEEE-754 standard.

Next, the leading zero detection (LZD) process removes the zeros located before of the significand digits of IP. A shifting process is carried out to remove shifting-left amount (SLA) zeros and as result is generated the shifted IP (SIP) that consists of the significant digits of IP.

Considering the rounding, the next step is to calculate the guard digit (GD) and the sticky-bit (SB). The GD is the *p*+1 position of SIP, in our case the seventeenth position. The SB is generated by means of OR gates operations of the remaining digits since seventeenth digit of SIP.

The rounding process is a combinational circuit that depending on the rounding strategy, the GD digit, and the SB bit generate an *add_one* signal in order to conditionally add one to the SIP significand.

Finally, the final product (FP), the intermediate exponent (IE), sign (S), and special cases (SC) are processed by the Coder to generate the IEEE-754-2008 format final output. In that final process the overflow and underflow should be analyzed and signaled.

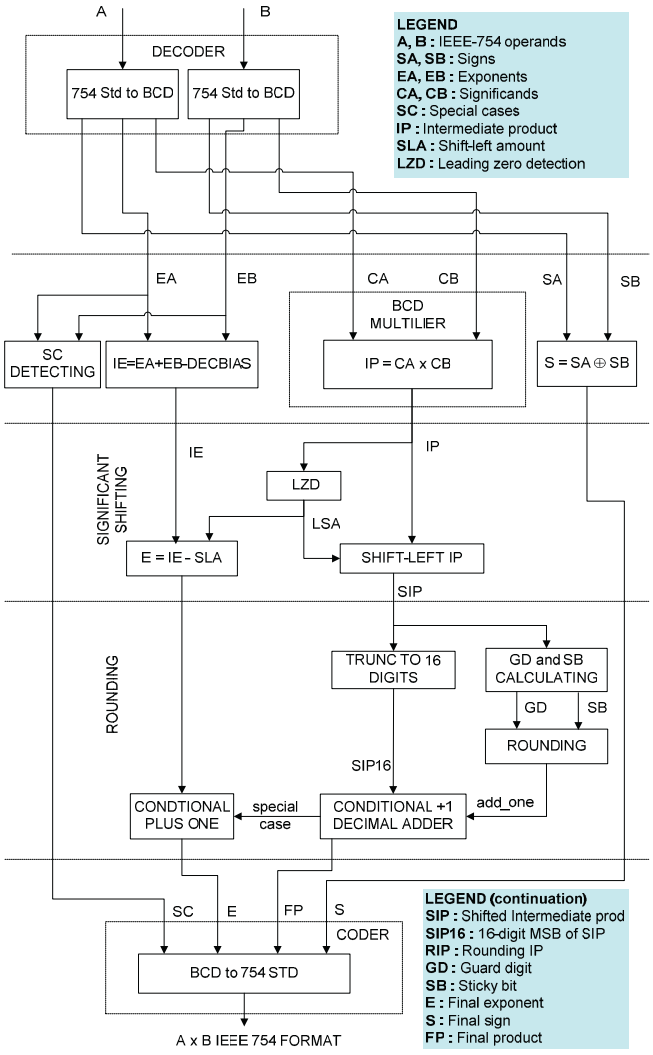

Figure 2. Decimal floating point multiplier block diagram

IV. IEEE 754-2008 Decoder / Coder

A decimal floating point number represented into the IEEE 754 interchange format (figure 1) is decomposed into the three fields: sign, exponent and significands.

A decimal64 number (64 bit interchange format) is decomposed into a 1 bit sign, a 10-bit exponent (E - bias), and 16 digits represented in BCD, i.e. 64 bits.

Then the decoder process is a combinational circuit that unpacks 64 bits into 75 bits (1+10+64). Into the decoder function the special cases (infinity and NANs) are detected and signaled.

The coder stage transforms the 75 bit representation of the result into the 64 bit interchange format. The combinational circuit takes into account the special cases that can be detected by the SC detection. Additionally in this stage the overflow and underflow conditions are detected and signaled.

V. BCD Fixed Point Multiplier

This stage reads the two significands (CA, CB) that comes from the decoder stage. The BCD multiplication calculate the final product that is IP = CA × CB.

The multiplier is based on [10] in this work several implementation techniques are proposed. Table 1 shows the implementation results in a Virtex 4 speed grade 12, the column BR stand for Xilinx Block Rams, T is period in ns, and #cy the number of cycles necessaries to complete the multiplication. The evaluated alternatives are four: 1) A fully combinational implementation (that finally is discarded because the amount of area necessary); 2 and 3) sequential implementation using LUTs and BRAMs to generate de digit by digit product. These alternatives need 17 clock cycles and present a good area and delay balance; and 4) a pipelined sequential multiplier, that consist in a 4-stage pipeline datapath using BRAMs to generate digit by digit product that requires only 8 clock-cycles.

TABLE I. Result of the 16 by 16 decimal digit multiplication on Virtex 4 -12

16x16 Multiplier	#slices	#FF	#LUT	BR	T (ns)	#Cy	Delay (ns)
Combinational	20,917	1,870	22,033	-	-	-	26.9
Sequential LUTs	1,486	519	1,669	-	8.5	17	144.5
Sequential BRAM	658	459	1,045	8	6.5	17	110.5
Sequential pipeline	1,969	1,384	3,007	32	5.4	8	43.2

VI. Shift Significand and Rounding

The shift significant and rounding stages of figure 2 were grouped in a single block. The first part transforms the 32 digit of IP (intermediate product) in a 16 digit value plus a guard digit and a sticky bit necessary to take the rounding decision. Additionally, at the shift significant and rounding stage the exponent will be adjusted subtracting the value SLA (shift left amount) to the previous computed IE (intermediate exponent).

The IEEE-754 standard describes several rounding model [3], the paper will present results for the nearest ties to even model but producing the other rounding models are easy and gives similar area and delay results.

In order to implement the shift significant and rounding stage, sequential and fully combinational implementations were analyzed in order to tradeoff area and delay.

A. Combinational implementation

The circuit detects and removes the zeros located to the left of the significant digits of the intermediate product (IP) up to a maximum of 16 zeros. Then the rounding procedure is made based on the shifted intermediate product, the guard digit and the sticky bit.

The combinational shift is based on the basic cell of figure 3 that shifts an *N* digits number from 0 up to *K* position to the left depending on the amount of leading zeros. A fully combinational shift circuit is suggested in figure 4. If the maximum amount of zeros to be deleted is *M*, then *M/K* cells should be used. In our implementation where the maximum amount of zeros to be deleted is 16, if we detect up to 2 zeros (*K*=2) we will need 8 shift-left basic cells.

Before rounding the shifted IP (SIP), the guard digit (GD) and sticky-bit (SB) are computed in parallel. The SIP just has the most significant digits; the GD is the next digits to the right of SIP digits, and finally the sticky bit is zero when all the rest of digits are equal to zero.

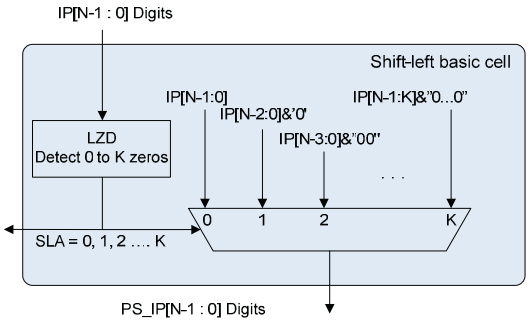

Figure 3. Shift significand basic cell. Shift from 0 to K digits

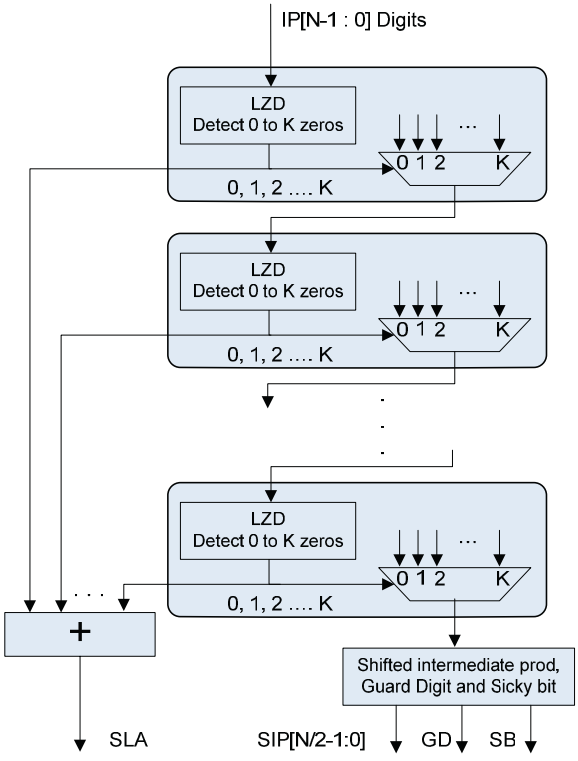

Figure 4. A simple combinational shift significand architecture.

In the rounding process, when no overflow or underflow are present, the rounding is accomplished by selecting either the shifted intermediate product (SIP) truncated to *p* digits or its incremented value. The decision to increment or not is made by the rounding module depending on the rounding strategy, the conditional plus one decimal adders adds one to the SIP or not depending on rounding *add_one*.

The rounding decision *add_one* is made by the following algorithm:

Algorithm 1. Simple rounding strategy

```
GD = SIP[15]; --guard digit
SB = is zero if SIP[14:0]are zero;--sticky bit.
if GD < 5 then add_one := '0';
elsif GD >5 then add_one := '1';
else --GD = 5
  if SB = '0' then add_one := '0';
  else add_one := '1'; end if;
end if;
```

Comparative figure of merit of this implementation is given at the Table II. In order to observe area-delay tradeoffs four architectures were implemented: comparison that begins with 16-zeros, with 8-zeros, with 4-zeros, and with 2-zeros.

TABLE II. RESULTS OF THE COMBINATIONAL SHIFT SIGNIFICAND AND ROUNDING.

# K-Zeros	# slices	# LUT	Delay (ns)
2	1002	1298	12.2
4	885	1538	10.5
8	776	1392	8.8
16	526	950	8.3

B. *Sequential implementation*

Based on the basic cell of figure 3 a sequential implementation of shifting process could be implemented. A simple optimization that can improve period of the data path is simple translating the digits to bits following the next relation:

BR(i) := '1' if IP(i) = "0000" otherwise '0';

Then the Leading *K*-Zero Detection (LZD), and sticky bit (SB) computation can be simplified. Figure 5 shows the sequential implementation of shift significand.

The data path is controlled by a simple state machine that shifts the digits up to 16 positions. The number of iterations depends upon the number of *K*-zero detected per clock cycle and also by the input number IP (intermediate product). If no leading zero is present in IP the circuit uses a single clock cycle. As an example, consider the number of figure 6, a 15 leading zeros number (*n* is a nonzero number and *d* any digit), using a *K*=4 (detects up to four zeros per cycle). The worst case is then 16/*K* + 1 cycles.

After the shift procedure in one cycle using the sticky bit (SB) and the guard digit (GD) the *add_one* signal is computed. In the final cycle the conditional plus one is performed.

The obvious advantage of the sequential architecture is noticed in the area improvement in comparing the combinational design, the results are observed at Table III.

TABLE III. RESULTS OF THE SEQUENTIAL SHIFT SIGNIFICANT AND ROUNDING ON VIRTEX 4

#*K*-zeros	#slices	#FF	#LUT	T (ns)	# Cycles min-max-avg	Delay (ns) min-max-avg
2	372	256	708	3.0	4 / 11 / 7.5	12.0 / 33.1 / 22.6
4	584	263	1126	3.4	4 / 7 / 5.5	13.5 / 23.7 / 18.6
8	775	257	1478	5.0	4 / 5 / 4.5	20.1 / 25.1 / 22.6

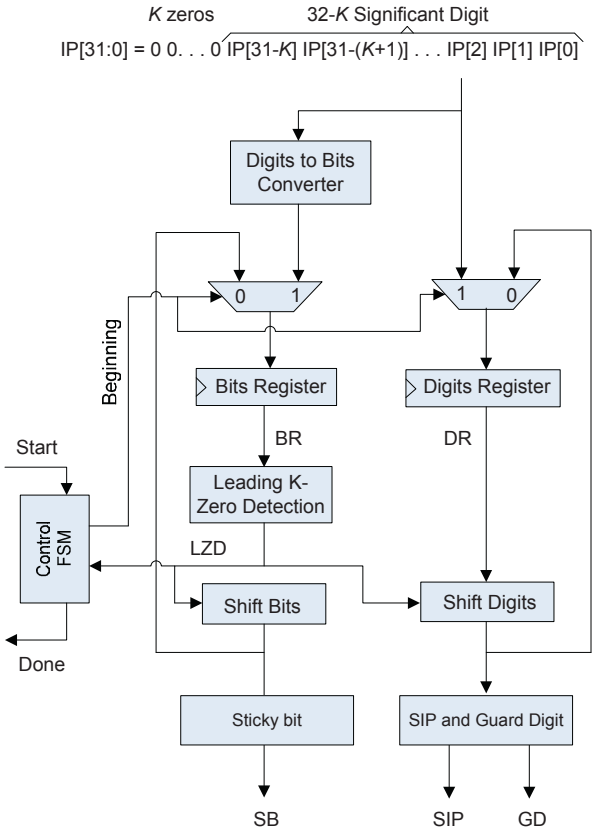

Figure 5. A sequential shift significand architecture.

Cycle	Digit Register (DR)
1	0000 0000 0000 000n dddd dddd dddd dddd
2	0000 0000 000n dddd dddd dddd dddd 0000
3	0000 000n dddd dddd dddd dddd 0000 0000
4	000n dddd dddd dddd dddd 0000 0000 0000
5	nddd dddd dddd dddd 0000 0000 0000 0000

Figure 6. Example of DR register evalution in a sequential shift significand with *K*=4.

An additional extra operation is potentially carried out at this point. If the shifted partial product is 99…99 and the *add_one* signal is asserted, the resulted decimal significand

is 10…0 but using one extra digit. The solution in that situation is that the conditional adder returns 10…0 with p digits and gives an additional signal in order to add one to the intermediate exponent.

VII. FPGA IMPLEMENTATION OF DECIMAL FLOATING POINT MULTIPLIER

All circuits were described in VHDL. Some parts of multiplier use low level component instantiation. For synthesis and implementation XST [11] and Xilinx ISE 10.1 tool [12] have been used respectively. The circuits were implemented in a Virtex-4 speed grade -12 [13] using timing constraints.

The final area and delay figure depends mainly on multiplier and the shift significand selected. At table IV the resulting DFP multiplier for minimum delay (low latency) and minimum area (low area) are presented. Those implementations use the nearest ties to even rounding and the special cases (SC) and overflow/underflow detection overhead is not considered.

For the fast circuit the pipeline multiplier is used with the combinational 16-zeros block for shift significand and rounding. Since the combinational rounding is the critical path, this operation is made in two clock cycle in order to speed up the clock frequency.

The minimum area use the sequential BRAM based multiplier and the sequential 2 zeros-block for shift significand and rounding stage. In this circuit the average case delay is presented.

TABLE IV. RESULTS OF THE DFP MULTIPLIER IMPLEMENTED IN VIRTEX 4 -12

DFP multiplier	#slices	# FF	# LUT	BR	T (ns)	# Cy	Delay (ns)
Low latency	3494	1713	5961	32	5.5	12	66.0
Low area	1222	1123	1966	8	6.2	26	161.2

Table V shows the area and delay breakdown DFP multiplier parts for both alternatives. Clearly the fixed point BCD multiplication and shift significant are the more area-time consuming task and where the optimization effort should be applied. In this breakdown the coder and decoder do not include special cases, overflow and underflow.

A. Verification

To validate the designs, large numbers of random vectors were applied to an automated testbench that tests the behavioral model using ModelSim. For the floating point multiplier, over 10,000 test cases were used. Since not all rounding modes and exception are supported in this version, only the supported featured were under test.

B. Future Inprovements

Several improvements could be studied in a future advanced version. Firstly, the fixed point decimal multipliers could be improved in clock speed and in latency. The use of carry save reducer trees instead of carry-chain adders is one of the alternatives. Secondly, the significand shifting could be also improved pre-computing the leading zeros of the intermediate product (IP) based on leading zeros of operands (CA and CB). A deeper analysis of shift significant and how to fit it inside the FPGA structure will give interesting results. Finally the complete support of rounding modes and exceptions should be studied and added.

TABLE V. AREA DELAY BREAKDOWN OF DFP MULTIPLIER

Area / delay breakdown	Low Latency				Low Area			
	delay		*area*		*delay*		*area*	
	#Cy	*%*	*Luts*	*%*	*#Cy*	*%*	*Luts*	*%*
decoder	1	8%	107	3%	1	4%	107	6%
multiplication	8	67%	3007	73%	17	65%	1045	54%
shift&round	2	17%	950	23%	7	27%	708	37%
coder	1	8%	68	2%	1	4%	68	4%

C. Results Comparisons

To the best of author's knowledge, this is the first publication to present an IEEE 754-2008 compliant multiplier in FPGA. In the other hand ASIC results for IEEE-754-2008 decimal multiplier were recently reported [14, 15] but is hard to establish a fair comparison.

In order to have a kind of comparison we have implemented the binary floating point multiplication [16] from Xilinx core generator using a binary64 format. The core generator has different architectures using more or less DSP blocks.

We have implemented in the same Virtex 4 -12 device the multiplier that uses more DSPs (called max usage) obtaining a multiplication in 23 cycles at 454 MHz (2.2 ns period), using 17 DSP blocks and 580 slices (768 FF, 524 LUTs). That is, multiplies in 50.6 ns. Additionally, we have implemented the binary multiplier that uses no DSP (called no usage). This circuit produces a result in 9 cycles working at 243 MHz (4.1 ns period) and needs 1381 slices (1381, 2308 LUTs). This multiplier needs a total of 36.9 ns for a binary FP multiplication.

Observe that results of table IV have comparative computation delay to the binary counterpart. Further more if several decimal operations should be carry out the decoding and coding process of DFP could be avoided saving two clock cycles per decimal operation.

VIII. SUMMARY

A first implementation of an IEEE 754-2008 decimal multiplier in FPGA was developed. The results are promising having a multiplication in 66 ns in a Virtex 4 speed grade 12. The computation time for a decimal64 is comparable as the necessary for a binary64 multiplication using the core provided for Xilinx.

The future improvements and speed up comes from modify the decimal fixed point multiplication, reducing whenever possible the clock cycles and improving the shift

significant process. Since the architecture is generated using generic parameters the decimal128 implementations is almost trivial.

ACKNOWLEDGMENT

This work is granted by the CICYT of Spain under contract TEC2007-68074-C02-02/MIC.

REFERENCES

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

[2] 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

[3] IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008), Revision of IEEE Std 754-1985. 29 August 2008.

[4] M. A. Erle and M. J. Schulte. “Decimal Multiplication Via Carry-Save Addition”. In International Conference on Application-Specific Systems, Architectures, and Processors, pages 348–358, June 2003.

[5] T. Lang and A. Nannarelli, “A Radix-10 Combinational Multiplier”. In Proc. of 40th Asilomar Conference on Signals, Systems, and Computers, pages 313–317, Oct 2006.

[6] A. Vazquez, E. Antelo, and P. Montuschi. “A New Family of High–Performance Parallel Decimal Multipliers”. In 18th IEEE Symposium on Computer Arithmetic, pages 195–204, June 2007.

[7] B. Hickmann, A. Krioukov, M. Schulte, M. Erle, “A parallel IEEE P754 decimal floating-point multiplier” In proc. of ICCD 2007, pp 296-303, Oct. 2007

[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. Bioul, G. Sutter, M. Vazquez, J-P. Deschamps, “Decimal Addition in FPGA” in proc of V southern conference of programmable logic, Sao Carlos Brazil, arpil 2009.

[10] G. Sutter, G. Bioul, E.Todorovich, “Decimal multipliers in FPGAs”, UAM Internal Report, January 2009.

[11] Xilinx Inc. XST User Guide 10.1. available at www.xilinx.com. June 2008.

[12] Xilinx Inc. Xilinx ISE Design Suite 10.1 Software Manuals. available at www.xilinx.com. 2008.

[13] Xilinx Inc. Virtex-4 Libraries Guide for VHDL design available at www.xilinx.com. 2008.

[14] Charles Tsen, Sonia González-Navarro, Michael Schulte, Brian Hickmann, Katherine Compton, “A Combined Decimal and Binary Floating-Point Multiplier”, ASAP 2009 20th IEEE Intern. Conf. on Application-specific Systems, Architectures, pp.8-15, July 2009.

[15] Mark A. Erle, Brian J. Hickmann, Michael J. Schulte, “Decimal Floating- Point Multiplication”, IEEE Transactions on Computers, vol. 58, no. 7, pp. 902- 916, July 2009.

[16] Xilinx inc. Floating-Point Operator v4.0, Product Specification DS335 April 25, 2008.