FPGA Implementation of Optimized Decimal Floating Point Multiplier using Binary Integer Decimal Encoding

[1]Rahul Shrivastava , [2]Prof. S.G. Kelarkar, [3]Prof. N.K. Mittal
[1]Dept. of Electronics and Communication Engineering, Oriental Institute of Science and Technology,Bhopal, India
[2]HOD, Electronics and Communication Engineering, Oriental Institute of Science and Technology,Bhopal, India
[3]Principal, Oriental Institute of Science and Technology,Bhopal, India
Email: rahul.stva@gmail.com

Abstract — **Binary floating point arithmetic is popularly used in hardware designs. But there are certain flaws in binary floating point arithmetic (BFP) namely; rounding error, error is representing decimal fractions etc. To reduce these errors decimal floating point arithmetic is used. In this paper an optimized approach to implement IEEE complaint binary integer decimal encoding based multiplier unit is presented. The proposed optimizations are given to reduce the delay and dynamic power consumption.**

Key Words—**BID Encoding, Floating point multiplication, low power, rounding**

I. INTRODUCTION

The binary floating point (BFP) arithmetic has certain flaws namely; it cannot provide correct decimal rounding and cannot precisely represent some decimal fractions such as 0.001, 0.0475 etc [1]. There are many applications where a precision is required such as billing, insurance, currency conversion, banking and some scientific applications. European Union requires that currency conversion to and from EURO is to be calculated to six decimal digits [2]. One study estimates that errors generating from BFP arithmetic can sum up to a yearly billing of over dollar 5 million for a large billing organization [3]. Therefore decimal floating point (DFP) arithmetic becomes very important in many current and future applications as it has ability to represent decimal fractions precisely. DFP arithmetic also has the ability to provide correct decimal rounding that will mimic the manual rounding.

Applications which cannot tolerate errors generating from BFP arithmetic, these application use software platforms to perform DFP arithmetic [1]. There are many software packages which are available for example: the java BigDecimal library [5] and IBM's decNumber library [4]. Also Intel published results for a decimal arithmetic library which uses Binary integer decimal (BID) encoding. These software packages are good enough for current applications, but trends towards globalization and e-commerce are increasing, so faster response of these systems is required. Software designs to these systems may be inadequate with the increasing performance demands of future systems. So hardware implementation of these systems is the need of the hour.

In 2008, the IEEE 754-1985 floating point standard has been revised and the new standard called the IEEE 754-2008 floating point standard was setup [6], which includes specifications for DFP formats, encoding and operations. The IEEE 754-2008 standard includes an encoding format for DFP numbers in which the significand and the exponent (and the payloads of NaNs) can be encoded in two ways namely; binary encoding and decimal encoding. [7]

Both the encoding formats break a number into a sign bit *s*, an exponent *E*, and a *p*-digit significand *c*. The value encoded is $(-1)^s \times 10^E \times c$. In both formats the range of possible values is identical, but the significand *c* is encoded differently. In the decimal encoding, it is encoded as a series of *p* decimal digits using the densely packed decimal encoding (DPD). This makes conversion to decimal form efficient, but it requires a special decimal ALU to process and it increases the hardware cost and delay of design. In the binary encoding also known as binary integer decimal (BID) encoding, it is encoded as a binary number. The BID encoding method is simple to implement as compared to DPD. It also requires less hardware also the delay performance of the design improves. [7]

In this paper a floating point multiplier unit is proposed. This floating point multiplier unit is IEEE P754 - 2008 complaint and based on binary integer decimal (BID) encoding for DFP arithmetic. The proposed floating point multiplier unit uses reciprocal multiplication to perform rounding and it is optimized for reduced delay and power consumption.

II. DECIMAL FLOATING POINT REPRESENTATION

In IEEE P754-2008, the value of a finite decimal floating point number is given by:

$$(-1)^s \times 10^{E-bias} \times C$$

Where S is the sign bit, '0' represents positive (+) number and '1' represents negative (-) number, E is the biased exponent, bias is a constant value that makes E a non-negative value and C is the significand. The IEEE 754 – 2008 specifies two methods of representing the significands of these numbers namely: Binary Integer Decimal (BID) [8] and Densely Packed Decimal (DPD) [9]. In BID encoding the significand is represented by an unsigned binary integer and in case of DPD encoding the significand is represented by unsigned decimal integer, a group of 10 bits is used to represent three decimal digits [9]. Sometimes the BID is called binary number and DPD is called as decimal number. For example 9.94 is represented as 994 x 10^{-3} in floating point number system, now the significand 994 can be represented using either BID or DPD encoding.

Three types of numbers are defined in IEEE P754 - 2008, decimal32 where the number is represented by 32 bits, decimal64 in this the number is represented by 64 bits and decimal128 where 128 bit are used to represented the number. In this paper decimal64 number is used, in this format 54 bits are used to encode the significand, remaining 10 bits are used to encode the sign bit and biased exponent. The maximum significand supported by decimal64 is $10^{16} – 1$, this number is less than 2^{54}. More details of BID encoding are given in [10].

III. BID MULTIPLICATION TECHNIQUE

A basic BID multiplication algorithm is shown in figure 1

```
Basic BID multiplication
1. Decode the inputs A and B to obtain ($A_S$, $A_E$, $A_C$) and ($B_S$, $B_E$, $B_C$)
2. Calculate IPC = $A_C$ x $B_C$, IPE = $A_E$ + $B_E$ - bias and $Z_S$ = $A_S$ XOR $B_S$
3. If IPC exceeds p digits, round the product and adjust the exponent
   Skip 3 if result is within p digits
4. Encode the result in BID format.
```

Figure 1: Basic floating point multiplication algorithm

To understand the hardware implementation of the BID multiplier, we first discuss a basic algorithm to multiply BID numbers. In IEEE 754-2008 format, the representation of a finite DFP number consists of three encoded values (S, E-bias, C), where S is the sign bit, E is a biased exponent, bias is a constant value that makes E nonnegative, and C is the significand. In the following discussion, A and B are the IEEE 754-2008 input operands, and (A_S, A_E, A_C) and (B_S, B_E, B_C) are their respective triples. The basic BID multiplication algorithm consists of four important steps and is shown in Figure 1. The IEEE 754-2008 BID-encoded operands are decoded to get their signs, biased exponents, and significands. Next, the significands, A_C and B_C, are multiplied to obtain the intermediate product, IPC. In parallel, the exponents are added and the bias is subtracted to produce the intermediate exponent, IPEXP. In the third step of the algorithm, IPC is tested to determine the need of rounding and the number of digits to be rounded. If the number of decimal digits in IPC is not greater than the result precision, p, rounding is not needed, and the multiplication operation is finished. Otherwise, IPC must be rounded and IPEXP must be adjusted. Finally, the sign, biased exponent, and significand of the result are packed to obtain the IEEE 754-2008 BID-encoded result of the multiplication. As an example of how BID multiplication is performed, consider the decimal64 BID-encoded operands of A = $32638D7EA4C68003_{16}$ and B = 3380000000000019_{16} as inputs to the multiplication, where the subscript 16 indicates hexadecimal digits. Numbers without subscripts have decimal digits. As described in the multiplication algorithm, the inputs are first decoded to obtain

$$A = (0,403 – 398,1,000,000,000,000,003)$$

$$= (1,000,000,000,000,003) \text{ X } 10^{403-398} \text{ and}$$

$$B = (0,412 – 398, 25) = 25 \text{ X } 10^{14}.$$

Next, the significands are multiplied to produce IPC = 25,000,000,000,000,075. In parallel, the biased exponents are added and the bias is subtracted to produce IP_{EXP} = 403 + 412 - 398 = 417. If IPC exceeds the result's precision, p, rounding is needed. In this example, IPC has 17 digits, and the precision for decimal64 is p = 16 digits, so it is necessary to round off one digit and increment IP_{EXP} by one. Depending on the rounding mode, the rounded result is either

$$Z = (0,418 - 398;2, 500,000,000,000,007)$$

$$= 2,500,000,000,000,007 \text{ x } 10^{20} \text{ or}$$

$$Z = (0,418 – 398;2,500,000,000,000,008)$$

$$= 2,500,000,000,000,008 \text{ x } 10^{20}.$$

Finally, Z is encoded to give the IEEE 754-2008 BID-encoded result of either

$3448E1BC9BF04007_{16}$ or $3448E1BC9BF04008_{16}$ depending upon the rounding mode.[10]

IV. ROUNDING MECHANISM

Five rounding modes are specified in IEEE 754 – 2008 specification namely; round ties to even (RTE), round ties to away (RTA), round towards zero (RTZ), round towards negative (RTN) and round towards positive (RTP). An example is explained here to easily explain the concept of rounding. Consider an input 1234 x 10^{-2}, which is 12.34 in simple decimal arithmetic, after rounding the resulting value is 13 x 10^0 in RTP rounding mode and 12 x 10^0 in all other rounding modes. The BID encoded significand is represented as 1234_{10} = 10011010010_2, the subscript represents radix of the number system and the leading zeros are not shown. Now after rounding the BID significand is either 13_{10} = 1101_2 or 12_{10} = 1100_2 depending on the rounding mode.

Decimal rounding can be implemented by dividing the number by 10^d to truncate d digits, the hardware should also have the intelligence to decide that the resulting number should be incremented or decremented depending upon the specified rounding mode. Now in this method the remainder is used to determine that whether increment or decrement is to be done. However this method is costly in terms of delay, power consumption and area. One other technique can be used which uses comparatively less area and greater delay performance, the method is called reciprocal multiplication.

Reciprocal Multiplication

In this method we multiply the value with a pre-calculated approximation of $W_d = 10^{-d}$ to achieve the division by $m = 10^d$.

This method is well suited when the divisors are already known. We have this situation and took the benefit from here. We need only p divisors in a format with p decimal digits. Here we have considered decimal64 format and in this format p is 16, so when we multiply by 10^{-d}, the value of d is from 1 to 16.

Now to perform correct decimal rounding, it is needed to determine that the pre-rounded result lies exactly halfway between two consecutive floating point numbers. To understand the importance of this, consider an input 6754500 x 10^{-3}, in this case the significand is 6754500 and the pre-rounded result is 6754.500. The correct rounding result is 6754 x 10^0 for RTE, RTZ and RTN rounding modes, but 6755 x 10^0 in RTA and RTP rounding modes.

Theorem 1: Let Ci and m = 10^d be positive integers such that 0 < Ci < 2^u and 0 < m < 2^v. Let q and r be the integer quotient and remainder of Ci divided by m. Thus, Ci = q·m + r, 0 ≤ r ≤ m-1. Define wd = ceil(2^{u+v}/m) and let P = Ci × wd be expressed in the form

$$P = 2^{u+v}Q + 2^uR + D$$

where Q, R, and D are non-negative integers, R < 2^v, and D < 2^u. Then Q = q. Furthermore r = 0 iff R = 0; r ≤ (m/2)-1 iff R ≤ 2^{v-1}-1; r = m/2 iff R = 2^{v-1}; and r ≥ (m/2)+1 iff R ≥ 2^{v-1}+1.

On the basis of theorem 1, Q provides the truncated product, q = Ci/10^d and R provides all information needed to determine the correctly rounded result. u is the number of bits required to represent the significand which is fixed to 54 in decimal64 format. The number of bits needed to represent 10^d, is v, given by: $d\log_2 10$, and this value is variable and depends on the value of d.

The 2u + 1 bit product, p = Ci x Wd shown in figure 2 is divided into three fields Q, R and D

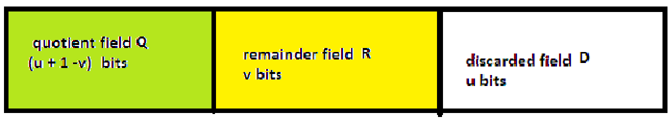

Figure 2: Product Fields

The D (u bits) field is discarded as it does not contain any useful information. The R (v bits variable in length) field is inspected to determine that the fraction is exactly zero, exactly one half, lies between zero and one half or above one half. To gather all the information above we will use a logic to determine r_star and s_star. Here r_star is the most significant bit of the R field and s_star is set if any of the remaining bits of R field is one or not. Table 1 summarizes the result gathered from r_star and s_star and their usage.

TABLE 1: Information gathered from r_star and s_star

Conditions	r_star	s_star
Pre-rounded result is exact	0	0
Pre-rounded result is less than mid-point	0	1
Pre-rounded result is exactly mid-point	1	0
Pre-rounded result is greater than mid-point	1	1

V. MULTIPLIER UNIT DESIGN

In this section, we present the hardware design of a decimal64 BID multiplier. Figure 3 shows a high-level block diagram of the decimal64 BID multiplier. The multiplier has three inputs: two BID-encoded operands, A and B, and the rounding mode information, rounding_mode. The outputs are the BID-encoded result of the multiplication, Z, and the exception flags. External reset and clock signals are required by the multiplier/rounder block, but they are not depicted here. The main blocks of the BID multiplier design are two BID decoders and one BID encoder, a multiplier/rounder block, and a block to handle the exception flags.

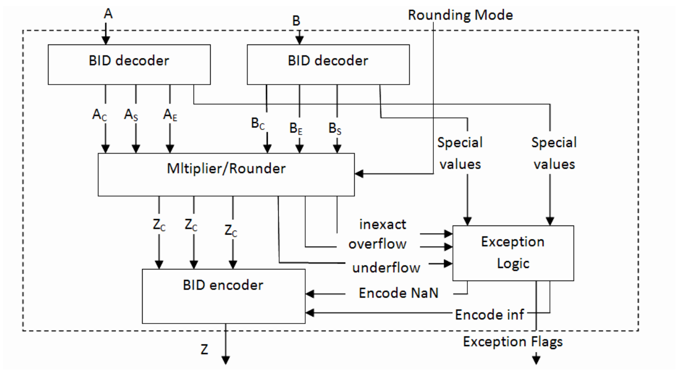

Figure 3: High level block

A. *Decoder and Encoder Blocks*

The multiplier includes two BID decoders to get the biased exponents, significands and signs from the two input operands. Each decodes an input into: 1 bit for the sign of operand, 10 bits for the exponent value, 54 bits for the significand, and 4 bits to indicate a special value (sNaN, qNaN, infinity (Inf), and zero).

The outputs from the BID decoders are sent to the multiplier/rounder block where the multiplication and rounding are performed. The special value bits are sent to a combinational exception logic block along with inexact, overflow, and underflow bits from the multiplier/rounder block to determine the exception flags and sNaN/Inf signals. The encoder produces the final IEEE 754-2008 result. This result is a finite or a special value (qNaN, sNaN, or Inf) depending on the encode signals. Although not shown in figure 4, when an overflow occurs, depending on the sign of the product and the rounding mode, the encoder produces as final output one of the following values: negative Inf, positive Inf, the maximum negative DFP number, or the maximum positive DFP number.

B. *Exception Logic Block*

When the multiplication result has been rounded and if some digits are rounded off, the floating-point number is inexact and, therefore, the multiplier generates a bit signaling this exception. In addition, the multiplication can produce an overflow or underflow, which must be signaled. These signals, shown in figure 3, are the inputs to the combinational exception logic block, along with the special value signals from the decoders. This exception logic block produces the invalid, underflow, overflow, and inexact flags. The invalid flag is raised when one of the operand inputs is zero and the other is Inf or when one of the operands is an sNaN. The underflow flag is raised when the underflow signal from the multiplier is set and neither input is NaN or Inf. The overflow flag is the overflow signal coming from the multiplier and indicates that a result may be too large to represent as a finite DFP number. The inexact flag is raised when an overflow or underflow result occurs or when the result is inexact and neither of the inputs is Inf nor NaN.

C. *Multiplier/ Rounding Block*

This section describes the multiplier/rounder block that multiplies and rounds BID-encoded decimal64 numbers.

To reduce area we have used a 53 X 53 bit array multiplier. The same multiplier is used in rounding unit for area conservation. The internal block diagram of the multiplier and rounder unit is shown in figure 4. The brief description of each block is given in following paragraphs.

C.1 LOD

After extracting A_c, B_c in binary form, the position of ‘1’ is found out using a LOD block, this will give us two values A_{top} and B_{top}. Now the two values A_{top} and B_{top} are added to calculate k. This will help us to evaluate the need of rounding, if $k < 53$ then the result is within the precision p of design. k is also used to determine the number of digits to be round off d.

C.2 d’ LUT and d calculation

The number of digits to be rounded off is d, first the value of d’ is determined by a d’ LUT (look up table) depending upon the value of k. the table is shown in table. Only k cannot always determine the value of d, d’ is determined by k, the exact value d is determined by d’ and a comparison unit of IPC and 10^n, if IPC > 10^n. then d = d’ + 1, else d = d’.

C.3 Wd LUT and 10^n

The value of W_d is calculated using a LUT. The value is stored in a LUT whose value is calculated manually using the formula $W_d = 2^{u+v/m.}$

C.4 Counter, Multiplexer Unit and Register Unit

The counter is used to count from 0 to 101(5 in decimal), this counter is used to reuse the multiplier. When count = 0, the significands Ac and Bc are multiplied. When count = 1, WdL and IPL are multiplied and result is stored in reg1, when count = 2, WdL and IPH are multiplied and result is stored in reg2, when count = 3, WdH and IPL are multiplied and stored in reg3, when count = 4, WdH and IPH are multiplied. When count = 5 the values are added to get a sum of 216 bit stored in product register.

C.5 Multiplier

A 54 X 54 multiplier is used. The same multiplier is used for both multiplication of significands and for rounding. This approach will save the very important area of design and makes the design more hardware cost effective.

C.6 Comparison and rounding needed

First k is compared with a constant value 53, if k < 53 then round_need = ‘0’. If k > 53 then roun_need = ‘1’. If k = 53 then IPL is compared with 10^{16}, if IPL < 10^{16} then round_need is ‘0’ else ‘1’.

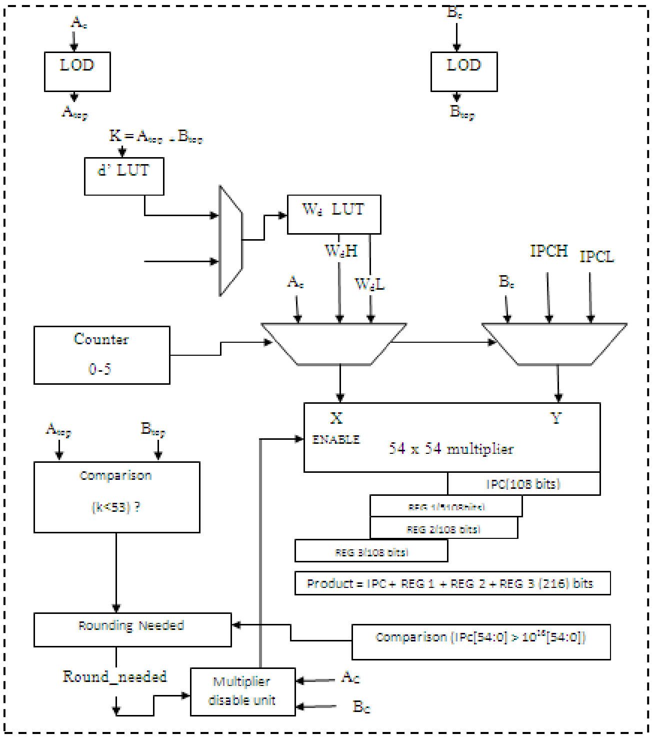

Figure 4: Multiplier/rounder block part A

C.7 Multiplier Disable unit

When round_need is ‘0’ then from count 1 to count 5 there is no need of multiplier, so we disabled the multiplier during this period. This will reduce the power consumption of the design. Also when A_C or B_C any of the two significands are zero then the result is zero. In this condition the multiplier is disabled.

C.8 Extract significand and rounder unit

This unit will determine the significand i.e. the output Z, also the r* and s* are calculated depending upon the values of d. the P field contains two fields the R and Q field. The MSB of R field is r* and if the remaining bits of R field is greater than 0 then s* is ‘1’ else ‘0’. The higher bits of P is Q field which is here C [108 : 54]. The remaining bits form R field.

C.9 Result Mux Logic and Increment

Depending upon the rounding values, r*, s* C_{tmp} [0] and Si the increment condition is determined. The table 1 shows increment condition.

Table 1: Rounding Modes and Increment Conditions

Rounding mode	**Increment Condition**
roundTiesToAway	r*
roundTiesToEven	*r** & (*s** | *Ctmp*[0])
roundTowardZero	0
roundTowardPositive	~*Si* & (*r** | *s**)
roundTowardNegative	*Si* &(*r** | *s**)

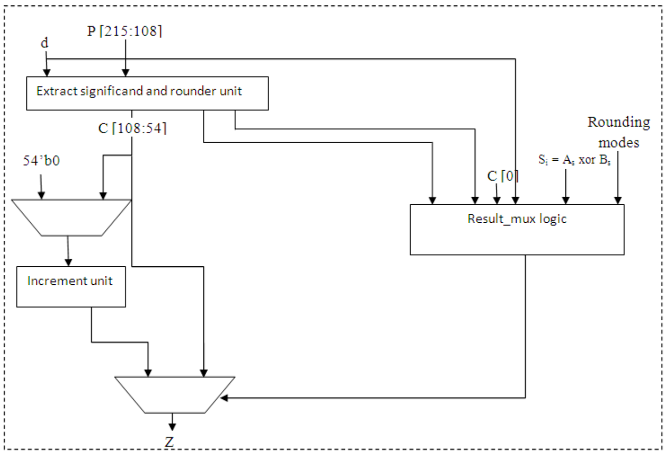

Figure 5: Rounder Block part B

Finally the encoding is accomplished. The table 2 below shows the values to be encoded. Also the exception values are encoded, when needed.

Table 2: Output Fields with round_need

round _need	Z_S	Z_E	Z_C
0	A_S XOR B_S	A_E + Be – 398	IPL
1	As XOR B_S	A_E + Be - 398 + d	Z

VI. RESULTS AND CONCLUSION

We have used Xilinx tool for simulation, synthesis and implementation of our design. Figure 6 shows the simulation result of the design. We have used ISIM tool by Xilinx to simulate the design. In the figure a[63:0] and b[63:0] (hexadecimal values) are the BID encoded input operands. Next the design decodes them and produces as, bs (sign bits), ae[9:0] and be[9:0] (bias exponent values, decimal equivalent) and ac[53:0] and bc[53:0] (multiplier and multiplicand respectively, decimal equivalent). After the decoding of the BID input operands the multiplication of ac[53:0] and bc[53 : 0] is accomplished and need of rounding is determined using signal k[6:0], here in this case rounding is not needed so the signal round_need is '0'. The exponent and sign values are generated namely zpe[9:0] (decimal equivalent) and si. As we can see from figure that multiplier is off during count 1 to count 5 (p = 'Z'), this will reduce the dynamic power consumption of the design. Next the intermediate results are encoded to produce en_output[63:0] (hexadecimal value), the final result.

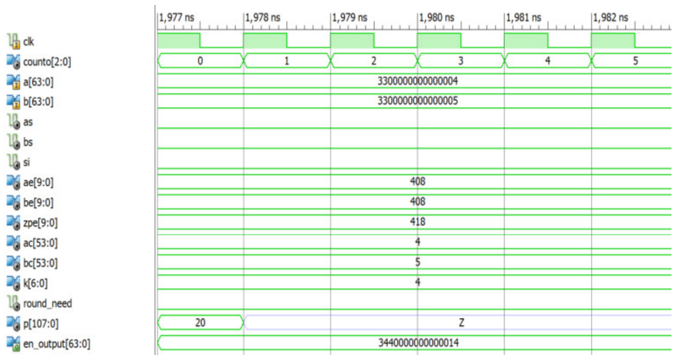

Figure 6: Simulation result using Xilinx ISIM

Here we have used Xilinx XST synthesizer to synthesize our design. Also Xilinx timing analyzer is used to calculate the delay of the design. The resource utilization summary is depicted in table.

Table 2: Resource Utilization Summary

Serial no	**Components used and Delay**	**Our design**
1	4 input LUT's	5907
2	Slices	3041
3	Flip flops	718
4	Cycles	6
5	Delay	26.06ns

We have used X-Power analyzer by Xilinx to calculate the static and dynamic power consumption of the design. The power consumption is summary is depicted in table.

Table 3: Power Consumption Summary

Serial no.	**Type**	**Our design**
1	Static power consumption	168.13 mW
2	Dynamic power consumption	88.14 mW
3	Total power consumption	256.28 mW

In this work we have used BID encoding instead of DPD encoding. This will reduce delay and hardware cost of the design. Also we have reused the multiplier for rounding purposes this will further decrease the hardware cost of the design, also power consumption of the system will be reduced. One more amendment we have incorporated here is we have disabled the multiplier when rounding is not needed, this will reduce the dynamic power consumption of the design.

VII. FUTURE SCOPE

A low power multiplier can be used here instead of simple array multiplier and the concept of clock gating [12-18]; this will increase the power performance of the system. Also instead of using reciprocal multiplication as a rounding mechanism, injection based rounding can be used, this will reduce the delay.[19] Also optimization in coding is also possible, which will improve the performance of the design.

REFERENCES

[1] M. F. Cowlishaw, "Decimal Floating-Point : Algorism for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 2003, Santiago deCompostela, Spain, pp. 104-111.

[2] D.G. for Economic and F. A. C. from the Commission to the European Council, "Review of the Introduction of Euro Notes and Coins," EURO PAPERS, Apr. 2002.

[3] IBM Corporation, "The 'telco' benchmark," *Available at http://www2.hursley.ibm.com/decimal/telco.html*, 2002.

[4] M. F. Cowlishaw, "The decNumber Library," *Available at http://www2.hursley.ibm.com/decimal/decnumber/pdf 2006*.

[5] Sun Microsystems, "BigDecimal (Java 2 Platforms SE v1.4.0)," *URL: http://java.sun/com/products,* Sun Microsystems Inc., 2002.

[6] ANSI/IEEE 754-1985, "Standard for Binary Floating-Point Arithmetic".

[7] http://*en.**wikipedia**.org/**wiki**/Binary_Integer_Decimal*.

[8] P. Tang, "Binary-Integer Decimal Encoding for Decimal Floating-Point," *Intel Corporation, Available at http://754r.ucbtest.org/issues/decimal/bid_rationale.pdf.*

[9] M. F. Cowlishaw, "Densely Packed Decimal Encoding," *IEEE Proceedings – Computers and Digital Techniques, vol. 149*, May 2002, pp. 102-104.

[10] Ping Tak Peter Tang "BID – binary integer decimal encoding " Intel Corporation, july 2006.

[11] Sonia Gonzalez-Navarro,Charles Tsen, Member and Michael J. Schulte," Binary Integer Decimal-Based Floating-Point Multiplication" IEEE transactions on computers, vol. 62, no. 7, july 2013 pp 1460-1466.

[12]http://www.mitpublications.org/yellow_images/1315565167_logo_13.pdf.
[13]http://www.xilinx.com/support/documentation/white_papers/wp370_Intelligent_Clock_Gating.pdf

[14] J. Di and J. S. Yuan, "Power-aware pipelined multiplier design based on 2-dimensional pipeline gating," in *13th Great Lakes Symposium on VLSI.* ACM, 2003, pp. 64–67.

[15] Sunjoo Hong, Taehwan Roh and Hoi-Jun Yoo, "a 145w 8×8 parallel multiplier based on optimized bypassing architecture", department of electrical engineering, Korea advanced institute of science and technology (KAIST), Daejeon, Republic of Korea, IEEE, pp.1175-1178, 2011.

[16] Yin-Tsung Hwang, Jin-Fa Lin, Ming-Hwa Sheu and Chia-Jen Sheu, "low power multipliers using enhenced row bypassing schemes", department of electronic engineering, National Yunlin University of science & technology, Touliu, Yunlin, Taiwan, IEEE, pp.136-140, 2007.

[17] George Economakos, Dimitris Bekiaris and Kiamal Pekmestzi, "a mixed style architecture for low power multipliers based on a bypass technique", national technical University of Athens, school of electrical and computer engineering, heroon polytechniou 9, GR-15780 Athens, Greece, IEEE, pp.4-6, 2010.

[18] Meng-Lin Hsia and Oscal T.-C. Chen, "low power multiplier optimized by partial-product summation and adder cells", dept. of electrical engineering, national chung cheng University, chia-yi, 621, Taiwan, IEEE, pp.3042-3045, 2009.

[19] Guy Even, Silvia M. Mueller, Peter-Michael Seidel "A dual precision IEEE Floating-point multiplier" Elsevier INTEGRATION, the VLSI journal 29 (2000) 167-180.