Design and Implementation of IEEE-754 Addition and Subtraction for Floating Point Arithmetic Logic Unit

V.vinay chamkur
Vinayv.carrier@gmail.com
M.Tech ,VLSI DESIGN and EMBEDDED SYSTEMS
Chethana.R
LecturerECE Dept, SJBIT Bengaluru-60.

***ABSTRACT---* This paper describes the FPGA implementation of a Decimal Floating Point (DFP) adder/subtractor using IEEE 754-2008 format. In this paper we describe an efficient implementation of an IEEE 754 single precision Standard for Binary Floating-Point Arithmetic to include specifications for decimal floating-point arithmetic. As processor support for decimal floating-point arithmetic emerges, it is important to investigate efficient algorithms and hardware designs for common decimal floating-point arithmetic algorithms. This paper presents novel designs for a decimal floating-point addition and subtraction. They are fully synthesizable hardware descriptions in VERILOG. Each one is presented for high speed computing.**

***Keywords-* IEEE-754 Floating Point Standard; Addition and Subtraction Algorithm.**

I.INTRODUCTION

Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [1] standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. This paper focuses only on single precision normalized binary interchange format. Fig. 1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the significand[1]. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by (1)

The IEEE-754 standard specifies six numerical operations: addition, subtraction, multiplication, division, remainder, and square root. The standard also specifies rules for converting to and from the different floating-point formats (e.g short/integer/ long to /from single/double/quad-precision), and conversion between the different floating-point formats.

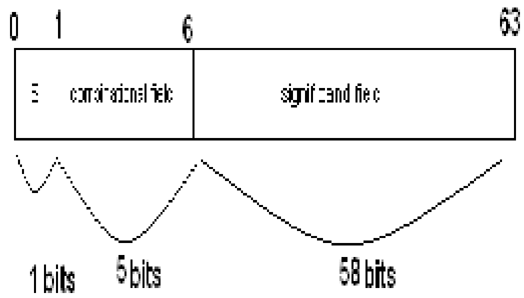


Figure 1.IEEE floating point format

$$Z = (-1^{S}) * 2^{(E - \underline{Bias})} * (1.M) \quad (1)$$

Where $M = m_{22} 2^{-1} + m_{21} 2^{-2} + m_{20} 2^{-3} + \ldots + m_1 2^{-22} + m_0 2^{-23}$;

$\underline{Bias} = 127$.

FIG : -1

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 × 10bits and contains the bulk of the significand. J represents the number of depletes.

When this field is combined with the leading significand bits from the combination field, the format encodes a total of p = 3 × J + 1 decimal digits. 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 IEEE-754 standard specifies six numerical operations: addition, subtraction, multiplication, division, remainder, and square root. The standard also specifies rules for converting to and from the different floating-point formats (e.g short/integer/long to/from single/double/quad-precision), and conversion between the different floating-point formats.

, AX and BX are the significands and EAX, EBX and EX are the exponents respectively. X is a digit that denotes the outputs of different units. The symbol $(N)_Z{}^T$ refers to T^th bit of the Z^th digit

in a number N, where the least significant bit and the least significant digit have index 0. For example, (A1)2
5 is the fifth bit of the second BCD digit in A1.

II. DECIMAL FLOATING POINT IN IEEE 754-2008:-

The primary difference between two formats, besides the radix, is the normalization of the significands (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 mantissa is encoded in densely packed decimal,
The exponent must be in the range [emin, emax], when biased by bias. Representations for infinity and not-a number (NaN) are also provided.

$$D - 1^s \times C \times 10^q,\ q - E - \text{bias}$$

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 [3], 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 are encoded in k bits in the following three fields (Fig1):

III. DECIMAL FLOATING-POINT ADDER/SUBTRACTOR IMPLEMENTATION

A general overview of proposed adder/subtractor is
described below. For the best performance, the design presents eight pipelined stages as is exhibited in the Fig. 2. Arrows are used to show the direction of data flow, the dashed blocks indicate the main stages of the design, and the dotted line indicates the pipeline.

This architecture was proposed for the IEEE 754-2008 decimal64 format and can be extended for the decimal128 format. The adder/subtractor on decimal64 is carried out as follows: The decoder unit takes the two 64-bit IEEE 754-2008 operands (OP1, OP2) to generate the sign bits (SA, SB), 16-digit BCD significant (A0, B0), 10-bit biased exponents (EA, EB), the effective operation (EOP) and flags for specials values of NaN or infinity. The signal EOP defines the effective operation (EOP = 0 for effective
addition and EOP = 1 for effective subtraction), this signal is calculated as:

EOP = SA xor SB xor OP-------- (2)

As soon as possible the decoded significant become
available, the leading zero detection unit (LZD) takes these results and computes the temporary exponents (EA1, EB1) and the normalized coefficients (A1, B1). The swapping unit swaps the operands (A1, B1) if EA1 < EB1 and Generates the BCD coefficients A2 (with higher exponent,max(EA1, EB1)) and B2 (with lower exponent, min(EA1,EB1)). In parallel with the above mentioned, this unit generates an exponent difference (Ed = |EA1 - EB1|), the exponent
E2 = max(EA1, EB1), the SWAP flag if a swapping process is carried out, and the right shift amount (RSA) which indicates how many digits B2 should be right shifted in order to guarantee that both coefficients (A2, B2) have the same exponent.

The RSA is computed as follows:

```
if (Ed <= p_max)
    RSA = Ed
else RSA = p_max
```

The value p_max = 18 digits, RSA is limited to this
value since B2 contains 16 digits plus two digits which will be processed to compute the guard and round digit.

Next, the Shifting unit receives as inputs the RSA, and the significand B2 generating a shifted B2 (B3) and a 2-bit signal called predicted sticky-bit (PSB) that will predict two initials sticky bits. PSB and B3 will be utilized as inputs in the decimal addition, control signals generation and post-correction units, respectively.
The outputs above mentioned plus two signals,
significand A2 and EOP, are taken as inputs in the control signals generation unit and generates the signals necessary to perform an addition or subtraction operation, these signals are described in the Sub-section 3.4 and are made up of a prior guard digit (RD2), the final partial sum (S2) and the corrected exponent (E3).
digit (GD1), a prior round digit (RD1), an extra digit (ED), a signal which verifies if A2 > B3(AGTB) and a carry into (CIN).

The significant BCD (A2, B3) and the CIN are inputs the decimal addition unit generating the partial sum of magnitude |S1| = |A2 + (-1) EOP B3| and a carry out (COUT), respectively.
At once, the 16-digit decimal addition unit takes the A2, B3, EOP and CIN and computes S1 as follows:

S1 = A2 + B3 if EOP = 0, S1 = A2 + cmp9 (B3)
if EOP = 1 and A2 >= B3, and

S1 = cmp9 A2+cmp9(B3)) if EOP = 1 and A2 < B3.
The symbol cmp9 means the 9`s complement.

The post-correction unit uses as inputs the PSB, the
Exponent E2, GD1, RD1, ED, the partial sum S1 and
COUT to verify, correct and compute the inputs signals if only the following two cases occur: 1) COUT=1 and EOP=0 and 2) (S1)15=0 and (GD1 > 0) and (EOP=1).

The analysis is explained in the Sub-section 3.5.
This unit generates the final sticky bit (FSB), the corrected guard digit (GD2) and round Next, the Rounding unit takes the outputs of the prior unit and rounds S2 to produce the result´s significand S3 and adjusts the exponent E3 to calculate the final exponent E4. Simultaneously the overflow, underflow and sign bit signals

The final sign bit is computed as:

FS = (SA ^ ~EOP) V (EOP ^ (AGTB ⊕ SA ⊕ SWAP))

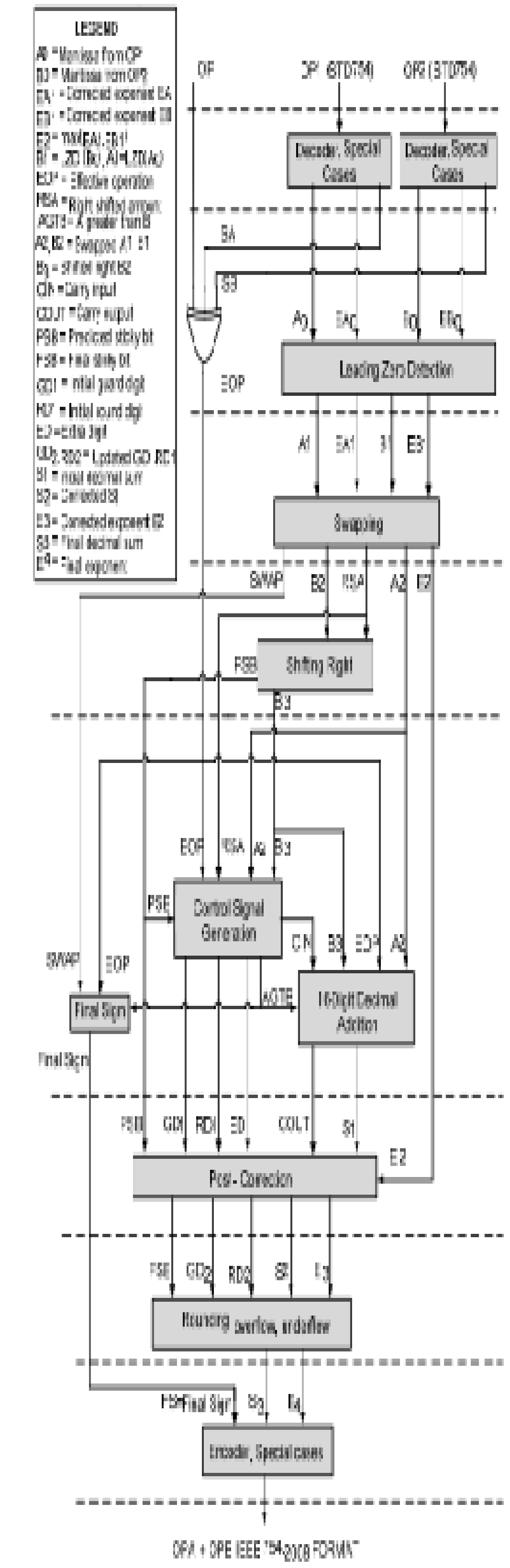


Figure 2: implementaition diagram are generated.

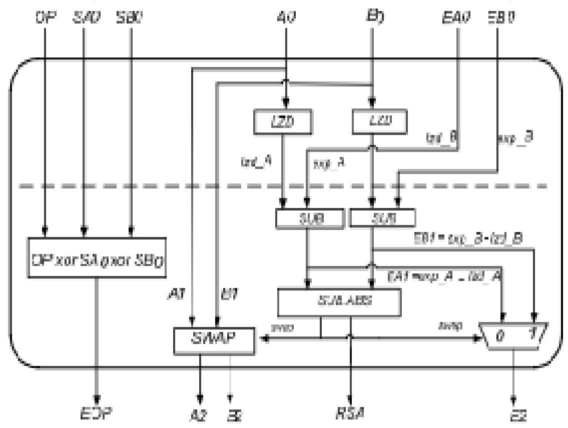


Fig. 3. Alignment and swapping unit

IV. PROBLEMS ASSOCIATED WITH FLOATING POINT ADDITION & SUBTRACTION

For the input the exponent of the number may be dissimilar. And dissimilar exponent can't be added directly. So the first problem is equalizing the exponent. To equalize the exponent the smaller number must be increased until it equals to that of the larger number. Then significant are added. Because of fixed size of mantissa and exponent of the floating-point number cause many problems to arise during addition and subtraction. The second problem associated with overflow of mantissa. It can be solved by using the rounding of the result. The third problem is associated with overflow and underflow of the exponent. The former occurs when mantissa overflow and an adjustment in the exponent is attempted the underflow can occur while normalizing a small result. Unlike the case in the fixed-point addition, an overflow in the mantissa is not disabling; simply shifting the mantissa and increasing the exponent can compensate for such an overflow. Another problem is associated with normalization of addition and subtraction. The sum or difference of two significant may be a number, which is not in normalized form. So it should be normalized before returning results.

V.ADDITION AND SUBTRACRION ALGORITHM

Let a1 and a2 be the two numbers to be added. The notations ei and si are used for the exponent and significant of the addends ai. This means that the floating-point inputs have been unpacked and that si has an explicit leading bit. To add a1 and a2, perform these eight steps:

1. If e1 < e2, swap the operands. This ensures that the difference of the exponents satisfies

d = e1–e2 □ 0. Tentatively set the exponent of the result to e1.

2. If the sign of a1 and a2 differ, replace s2 by its two's complement.

3. Place s2 in a p-bit register and shift it d = e1-e2 places to the right (shifting in 1's if the s2 was complemented in previous step). From the bits shifted out, set g to the most- significant bit, r to the next most-significant bit, and set sticky bit s to the OR of the rest.

4. Compute a preliminary significant S = s1+s2 by adding s1 to the p-bit register containing s2. If the signs of a1

and a2 are different, the most-significant bit of S is 1, and there was no carry out then S is negative. Replace S with its two's complement. This can only happen when d = 0.

5. Shift S as follows. If the signs of a1 and a2 are same and there was a carry out in step 4, shift S right by one, filling the high order position with one (the carry out). Otherwise shift it left until it is normalized. When left shifting, on the first shift fill in the low order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.

6. Adjust r and s. If S was shifted right in step 5, set r: = low order bit of S before

shifting and s: = g or r or s. If there was no shift, set r: = g, s: = r. If there was a single left shift, don't change r and s. If there were two or more left shifts, set r: = 0, s: = 0. (In the last case, two or more shifts can only happen when a1 and a2 have opposite signs and the same exponent, in which case the computation s1 + s2 in step 4 will be exact.)

7. Round S using following rounding rules as in Table ;

Rounding	*Sign of result*	*Sign of result <0*
-∞		+1 if $r \vee s$
0		
Nearest	+1 if $r \wedge p_0$ or r	+1 if $r \wedge p_0$ or r

If a table entry is non empty, add 1 to the low order bit of S. If rounding causes carry out, shift S right and adjust the exponent. This is the significant of the result.

8.Compute the sign of the result. If a1 and a2 have the same sign, this is the sign of the result.If a1 and a2 have different signs, then the sign of the result depends on which of a1, a2 is negative, whether there was a swap in the step 1 and whether S was replaced by its two's complement in step 4. As in table below

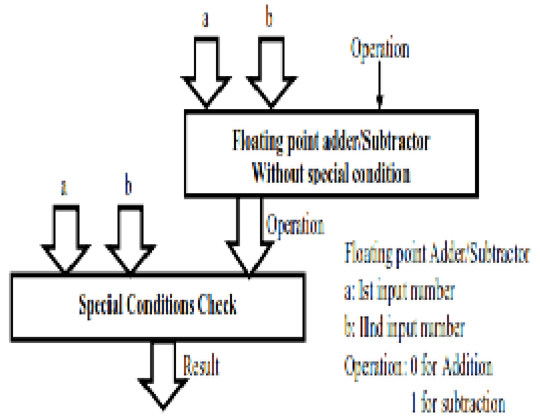
Swap	*Compleme*	*Sign (a_1)*	*Sign*	*Sign*
Y	∅	+	-	-
es	∅	-	+	+
Y		+	-	+
es	N	-	+	-
N	o	+	-	-

VI. SPECIAL CONDITIONS

Some special conditions are checked before processing. If any condition is met then we have no need to calculate the result by normal procedure. Results are directly calculated. So all the operations are bypassed when any such condition is met.

1. If a1 = 0 and a2 = 0 then result will be zero.
2. If a1 = a2 and sign of a1 □ sign of a2 then result will be again zero.
3. If a1 = 0 and a2 □ 0 then result will be equal to a2.
4. If a2 = 0 and a1 □ 0 then result will be equal to a1.
5. If d = |e1 – e2| > 24 then result will be equal to larger of a1 and a2.

VII.Hardware Approach


a
b
Operation
Floating point adder/Subtractor
Without special condition
a
b
Operation
Special Conditions Check
Result
Floating point Adder/Subtractor
a: Ist input number
b: IInd input number
Operation: 0 for Addition
1 for subtraction

The block diagrams of the architecture used for combinational adder is shown above in

Figure 4, step by step from the lower abstract level to the higher abstract level.

B Unsigned Adder (for exponent addition)

This unsigned adder is responsible for adding the exponent of the first input to the exponent of the second input and subtracting the Bias (127) from the addition result (i.e. A_exponent + B_exponent - Bias). The result of this stage is called the intermediate exponent. The add operation is done on 8 bits, and there is no need for a quick result because most of the calculation time is spent in the significand multiplication process (multiplying 24 bits by 24 bits); thus we need a moderate exponent adder and a fast significand multiplier.

An 8-bit ripple carry adder is used to add the two input exponents. As shown in Fig. 3 a ripple carry adder is a chain of cascaded full adders and one half adder; each full adder has three inputs (A, B, C_i) and two outputs (S, C_o). The carry out (C_o) of each adder is fed to the next full adder (i.e each carry bit "ripples" to the next full adder).

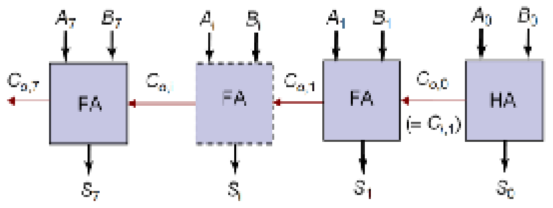


Figure 5 : - ripple carry adder

The addition process produces an 8 bit sum (S_7 to S_0) and a carry bit ($C_{o,7}$). These bits are concatenated to form a 9 bit addition result (S_8 to S_0) from which the Bias is subtracted. The Bias is subtracted using an array of ripple borrow subtractors .

A normal subtractor has three inputs (minuend (S), subtrahend (T),Borrow out (B_o)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant ($127|_{10} = 001111111|_2$).

S	T	B_i	Difference(R)	B_o
0	1	0	1	1
1	1	0	0	0
0	1	1	0	1
1	1	1	1	1

The above table shows one bit subtractor

VIII.SIMULATION RESULTS

FLOATING POINT ADDER

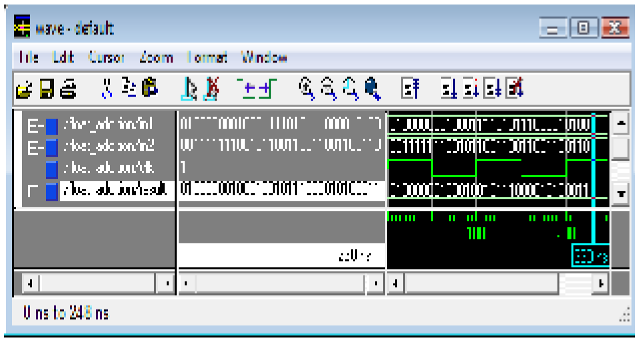
Input 1 = 01000000010001111010111000010100 (3.120 10)

Input 2 = 00111111100101100110011001100110 (1.175 10)

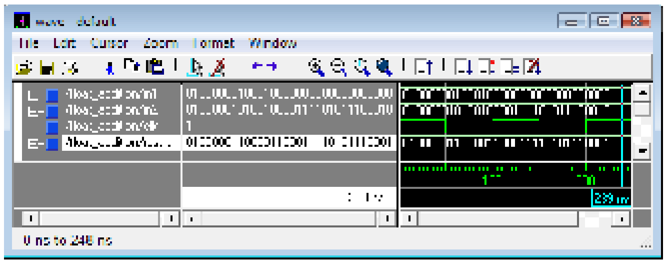
Required Result = 01000000100010010111000010100011

(4.295 10)

Obtained Result = 01000000100010010111000010100011 (4.295 10)

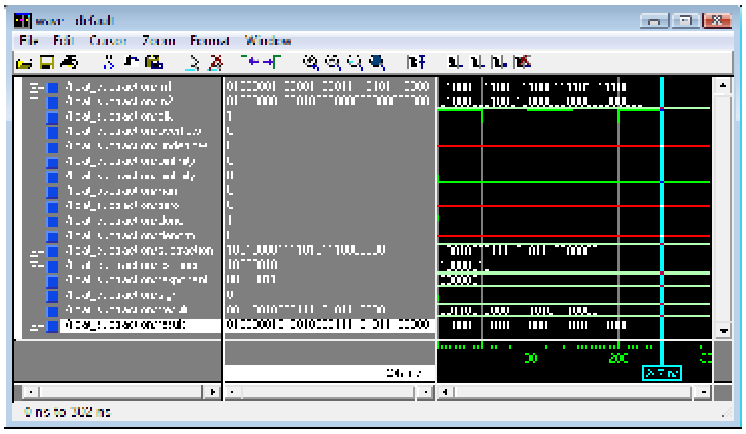


Input 1 = 01000000100010000000000000000000 (4.25 10)
Input 2 = 01000001010010000111101011100010 (12.53 10)
Required Result = 01000001100001100011110101110000 (16.78 10)
Obtained Result = 01000001100001100011110101110001 (16.78 10)

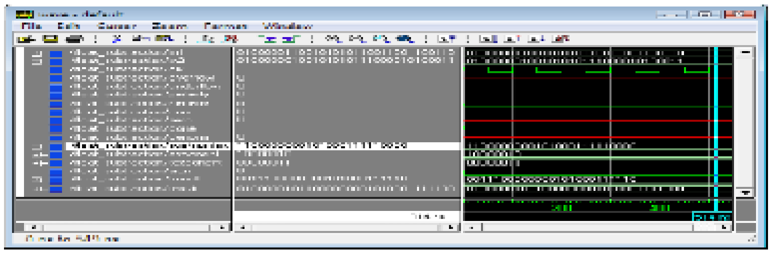


FLOATING POINT SUBTRACTOR

Input 1 = 01000001100001100011110101110000 (16.78 10)
Input 2 = 01000000100010000000000000000000 (4.25 10)
Required Result = 01000001010010000111101011100010 (12.53 10)
Obtained Result = 01000001010010000111101011100010 (12.53 10)



Input 1 = 01000001100101010110011001100110 (18.675 10)
Input 2 = 01000000100101010111000010100011 (4.670 10)
Required Result = 01000001011000000001010001111101 (14.005 10)
Obtained Result = 01000001011000000001010001111101 (14.005 10)



IX.IMPLEMENTATION AND TESTING

The whole adder (top unit) was tested against the Xilinx floating point adder core generated by Xilinx coregen. Xilinx core was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode.

A testbench is used to generate the stimulus and applies it to the implemented floating point adder and to the Xilinx core then compares the results. The floating point multiplier code was also checked using DesignChecker [7]. DesignChecker is a linting tool which helps in filtering design issues like gated clocks, unused/undriven logic, and combinational loops. The design was syn
5VFX200TFF1738 with a timing constraint of 300MHz. Post synthesis and place and route simulations were made to ensure the design functionality after synthesis and place and route. shows the resources and frequency of the implemented floating point multiplier and Xilinx core

	Technology	Clk (GHz)	Cycles	Delay (ns)	Mops/ sec
SW	Itanium2 [18]	1.4	219	156.4	6.4
	Xeon5100 [19]	3.0	133	44.3	22.6
	Xeon [18]	3.2	249	77.8	12.9
	Pentium M [21]	1.5	848	565.3	1.8
HW	Power6 [22]	5.0	17	3.4	294.1
	Z10 [23]	4.4	12	2.7	366.7
	BID 65nm [10]	1.3	3-13	10.0	100.0
	BID Virtex 5 [9]	0.16	13-18	109.8	9.1
	Proposed Virtex5	0.2	8	40	200.0

The area of Xilinx core is less than the implemented floating point adder because the latter doesn"t truncate/round the 48 bits result of the mantissa multiplier which is reflected in the amount of function generators and registers used to perform operations on the extra bits; also the speed of Xilinx core is affected by the fact that it implements the round to nearest rounding mode.

X. CONCLUSIONS AND FUTURE WORK

This paper deals with development of a Floating Point adder and subtractor for ALU in VHDL and verilog with the help of ModelSim and synthesized with Xilinx tools. Simulation results of all the designed programs have been carried out for various inputs with the help of ModelSim tool. Both are available in single cycle and pipeline architectures and fully synthesizable with performance comparable to other available high speed implementations. The design is described as graphical schematics and VHDL code. This dual representation is very valuable as allows for easy navigation over all the components of the units, which allows for a faster understanding of their interrelationships and the different aspects of a Floating Point operation.

References

[1] M. F. Cowlishaw, Decimal floating-point: algorism for computers, in *Proc. 16th IEEE Symp. Computer Arithmetic*, 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, 2009, iBM Journal of Research and Development.

[3] IEEE Standard for Floating-Point Arithmetic, pp. 1 58, 2008, iEEE Std 754-2008.

[4] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, The IBM z900 decimal arithmetic unit, in *Proc. Conf Signals, S ystems and Computers Record of the Thirty-Fifth Asilomar Conf*, vol. 2, 2001, pp. 1335 1339.

[5] W. Haller, K. Ulrich, L. Thomas, and H. Wetter, Combined binary/decimal adder unit, in *International Business Machines Corporation (Armonk, NY)*, 1999.

[6] G. Bohlender and T. Teufel, BAP-SC: A Decimal Floating-Point Processors for Optimal Arithmetic, in *Computerarithmetic: Scientific Computation*

[7] J. Thompson, N. Karra, and M. J. Schulte, A 64-bit decimal floating-point adder, in *Proc. IEEE Computer society Annual Symp. VLSI*, 2004, pp. 297 298.

[8] M. S. Cohen, T. E. Hull, and V. C. Hamacher, CADAC: A Controlled-Precision Decimal Arithmetic Unit, no. 4, pp. 370 377, 1983.

[9] A. Farmahini-Farahani, C. Tsen, and K. Compton, FPGA implementation of a 64-Bit BID-based decimal floating-point adder/subtractor, in *Proc. Int. Conf. F ield- Programmable Technology FPT 2009*, 2009, pp. 518 521.

[10] C. Tsen, S. Gonzalez-Navarro, and M. Schulte, Hardware design of a Binary Integer Decimal-based floating-point adder, pp. 288 295, 2007, computer Design, 2007. ICCD 2007. 25th International Conference on.

[11] C. Minchola and G. Sutter, A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier, in *Proc. Int. Conf. Reconfigurable Computing and FPGAs ReConFig 09*, 2009, pp. 59 64.

[12] L.-K. Wang and M. J. Schulte, Decimal Floating-Point Adder and Multifunction Unit with Injection-Based Rounding, in *Proc. 18th IEEE Symp. Computer Arithmetic ARITH 07*, 2007, pp. 56 68.

[13] M. Vazquez, G. Sutter, G. Bioul, and J. P Deschamps,
Decimal Adders/Subtractors in FPGA: Efficient 6-input
LUT Implementations, in *Proc. Int. Conf.
Reconfig urable Computing and FPGAs ReConFig 09*, 2009, pp. 42 47.

[14] *Xilinx Inc. XST User G uide 12.1*, v12.1 ed., Xilinx Inc., June 2009. [Online]. Available: http://www.xilinx.com

[15] *Xilinx Inc. Xilinx ISE D esign Suite 12.1 Software Manuals*, v12.1 ed., Xilinx Inc., June 2009. [Online]. Available:
http://www.xilinx.com

[16] *Xilinx Inc. Virtex-5 Li braries Guide for VHDL design*, v12.1 ed., Xilinx Inc., June 2009. [Online]. Available:
http://www.xilinx.com

[17] Xilinx Inc, *DS335: Floating-Point Operator v5.0*June 2009.

[18] M. Cornea, J. Harrison, C. Anderson, P. Tang, E. Schneider, and E. Gvozdev, A software implementation of the ieee 754r decimal floating-point arithmetic using the binary encoding format, pp. 148 162, 2009, computers, IEEE Transactions on.

[19] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen, A software implementation of
the ieee 754r decimal floating-point arithmetic using the binary encoding format, pp. 29 37, 2007, computer Arithmetic, 2007. ARITH 07. 18th IEEE Symposium on.

[20] *The decNumber C library*, v3.68 ed., IBM UK Laboratories, January 2010. [Online]. Available: http://- speleotrove.com/decimal/decnumber.pdf

[21] M. Cowlishaw. (2009) Decimal library performance. [Online]. Available: http://speleotrove.com/decimal/-
decperf.
pdf