# Implementation and Analysis of Modified Double Precision Interval Arithmetic Array Multiplication

Krutika Ranjan Kumar Bhagwat<sup>#1</sup>, Prof. Tejas V. Shah<sup>\*2</sup>, Prof. Deepali H. Shah<sup>#3</sup>

<sup>#</sup> Instrumentation & Control Engineering Department,

\*Instrumentation & Control Engineering Department, Gujarat Technological University

L. D. College of Engineering, Ahmedabad-380015, Gujarat, India

<sup>\*</sup>S.S College of Engineering, Bhavnagar - 364060, Gujarat, India

Abstract— This paper presents the design of a 64 bit array multiplier that performs interval multiplication. This multiplier requires carry save adders instead of full adders that reduces the delay in respect of conventional array multiplier. The 64 bit multiplication requires 53 x 53 multiplication which is done by array multiplier it  $n^{*}(n-1)$  CSA, where n=53 so,  $n^{*}(n-1) = 53$  \*52= has 2756 CSA is used. Arrangement of 2756 CSA is used to add partial products of multiplier. This multiplier is provides the better based on interval arithmetic which accuracy, by removing rounding off error over conventional floating point multiplier. There is performance improvement over software implementation of interval arithmetic, but it requires slightly more area rather than conventional floating point unit.

**Keywords-** Double Precision, Interval Multiplication, Significand multiplier, Array Multiplier.

## I. DOUBLE PRECISION

IEEE 754 standard defines double precision as 1 sign bit, 11 bits for exponent ,53 bits for (52 explicitly stored) significand precision.

The format is written with the significand having an implicit integer bit of value 1, unless the written exponent is all Zeros. With the 52 bits of the fraction significand appearing in the memory format, the Total precision is therefore 53 bits (approximately 16 decimal digits, 53  $\log_{10} (2) \approx 15.955$ ). [4]

## II. INTERVAL MULTIPLICATION

Multiplication of the intervals  $x = [x_1, x_u]$  and  $y = [y_1, y_u]$  is defined as:

Z = x \* y

=  $[min(x_1y_1, x_1y_u, x_uy_1, x_uy_u), max(x_1y_1, x_1y_u, x_uy_l, x_uy_u)]$ 

The interval multiplier shown in figure 2 has input and output registers, sign logic, an exponent adder and a significand multiplier with rounding and normalization logic. The input and output registers are each 64 bits and two multiplexer with control signal  $t_x$ ,  $t_y$  are used.



Fig. 1 Interval multiplier

The sign logic computes the sign of the result by performing the exclusive-or of the sign bits of the input operands. The exponent adder performs an 11-bit addition of the two exponents and subtracts the exponent bias of 1023. The significand multiplier performs a 53-bit by 53-bit array multiplication. If the most signicant bit of the product is one, the normalization logic shifts the product right one bit and increments the exponent. The rounding logic rounds the product to 53 bits based on a rounding mode ( $r_{m}$ ) is round to nearest even. [10]

# III. SIGNIFICAND MULTIPLIER(ARRAY MULTIPLIER USING CSA)

m x n bit multiplication can be viewed as forming n partial products of m bits each, and then summing the appropriately shifted partial products to produce an m+n bit result p. Therefore, generating partial products consist of the logical Anding of the appropriate bits of the multiplier and multiplicand. Each column of partial products must then be added and, if necessary, any carry values passed to the Next column. Simple array multiplication using full adder is shown in figure 2. [4]

Partial products are added using carry save adder instead of full adder which reduces delay. Full adder is replaced with CSA given in figure 3. The idea is to take 3 numbers that we want to add together, x + y+ z, and convert it into 2 numbers c + s such that x +c + s. In carry save addition, we y + z =refrain from directly passing on the carry information until the very last step. [13]



Figure 2: simple array multiplication



Figure 3: Full adder is replaced with CSA

The significand multiplier performs a 53-bit by 53-bit multiplication. If the most significant bit of the product is one, the normalization logic shifts the product right one bit and increments the exponent. Arrangement of CSA for

p [22:0] is shown in figure 5.

The inputs are a [52:0] and b [52:0] to generate product p [105:0]. Consider j as carry save adder and s as sum of CSA and c as carry of CSA and aobo as partial product of a[0] and b[0], and Kxy as partial product of a[x] and b[y] in modify array multiplication for half precision as reference to double precision arrangement shown in figure 4 which has product p[22:0] when the inputs are a[10:0] and b[10:0].

Arrangement of CSA for half precision the significand multiplier performs an 11-bit by 11-bit multiplication is given in figure4. If the most signicant bit of the product is one, the normalization logic shifts the product right one bit and increments the exponent.

N-bit unsigned array multiplier required

$$n^{*}(n-1)$$
 CSA= 53 \*52 = 2756.

#### **IV. IMPLEMENTATION**

The signs of the endpoints of the intervals x and y indicate whether x and y are greater than Zero, less than Zero, or contain Zero. This results in nine possible cases, as shown in Table1.

All 9 cases for interval multiplication for lower

Interval  $Z_1$  and upper interval  $Z_0$  are given in Table 1, when both x and y contain Zero,

> mn = min ( $\nabla$  xlyu, $\nabla$  xuyl) and  $mx=max(\Delta xlyl, \Delta xuyu)$ .

Take  $e_{xl}$  as  $(10101010101)_b = (555)_b$  and  $e_{xu}$  as  $(11001100110)_{b} = (666)_{h}$  and  $e_{vl}$  as  $(11100011100)_{b}$  $= (71c)_h$  and  $e_{yu}$  as  $(11110000111)_b = (787)_h$  and  $f_{xl}$ as  $(15555555555555)_h$  and  $fx_u$  as  $(19999999999999)_h$ and f<sub>vl</sub> as  $(1c71c71c71c71c)_{h}$ ,f<sub>vu</sub> as  $(1e1e1e1e1e1e1e)_h$ .

For case 1 Inputs are  $S_{x1} = 0$ ,  $S_{xu} = 0$ ,  $S_{y1} = 0$ ,  $S_{yu} = 0$ that generates outputs are  $S_{zl} = xor(S_{xl}, S_{yl}) = 0$  and output  $S_{zu} = xor (S_{xu}, S_{vu}) = 0$ . For exponent add internal wire  $e_{z1}$  is the addition of  $ex_1$  and  $e_{y1}$ . That generates output  $e_{zl} = (10001110001)_{b}$  and overflow\_flag1  $(o_f1) = 1$ . For bias 1023 output  $e_{zl} = e_{zl} - 1023 =$ (00001110010) b and flag1 =0. For exponent add Internal wire  $e_{zu}$  = exponent add( $e_{xu}$ ,  $e_{yu}$ ) =  $(10111101101)_{b}$  and overflow\_flag2(o\_f2) = 1. For bias 1023 output  $e_{zu} = e_{zu} - 1023 = 00111101110$  and flag1=0.And outputs are  $f_{zl} = (f_{xl}*f_{yl})$  is equals to  $(ecf684bda12f684c)_h$ ,  $f_{zu} = (f_{xu} * f_{yu})$  is equals to (2edededededee)<sub>h</sub>.

In figure 5 and 6 case1 simulation reports are given. Same calculation up to cases for 64 bit interval arithmetic array multiplication is given in Table 1.

For case nine inputs are  $S_{xl} = 1$ ,  $S_{xu} = 0$ ,  $S_{yl} = 1$ ,  $S_{yu}$ =0 generate outputs are  $S_{zl}$ = xor( $S_{xl}$ ,  $S_{yu}$ )=1,  $S_{zu}$ = xor  $(S_{xl}, S_{yl})= 0, e_{xl} = 3'h= 555, e_{xu} = 3'h= 666, e_{yl} =$ 3'h=71c,  $e_{vu}=3'h=787$ .

For case nine consider four different conditions

1.  $f_{xl} > f_{xu}$  and  $f_{yl} > f_{yu}$ ,  $f_{xl} > f_{xu}$  and  $f_{yl} < f_{yu}$ , 3.  $f_{xl} < f_{xu}$  and  $f_{yl} < f_{yu}$ , 4.  $f_{xl} < f_{xu}$  and  $f_{yl} > f_{yu}$ 

VERILOG HDL is used for programming by Xilinx ISE Design Suite 13.2 for synthesis and schematic and simulation is done here with the help of ISIM 13.2.

p[0]	p[1]	p[2]	p[3]	p[4]	p[5]	p[6]	p[7]	p[8]	p[9]	p[10]	p[11]	p[12]	p[13	]p[14]	p[15]	p[16]	p[17]	p[18]	p[19	]p[20]	p[21]
a0b0	j1	j2	j4	j7	j11	j16	j22	j29	j37	j46	j56	j66	j75	j84	j92	j99	jA5	jA10	jA14	jA17	jA19
	k01	k02	k03	c4	с7	c11	c16	c22	c29	c37	c46	c56	c66	c75	c84	c92	c99	cA5	cA10	cA14	cA17
	k10	k20	k12	c5	c8	c12	c17	c23	c30	c38	c47	c57	c67	c76	c85	c93	c100	cA6	cA11	cA15	cA18
	cin	k11	k21	<b>c6</b>	c9	c13	c18	c24	c31	c39	c48	c58	c68	c77	c86	c94	cA1	cA7	cA12	cA16	cin
		j3	j5	j8	j12	j17	j23	j30	j38	j47	j57	j67	j76	j85	j93	j100	jA6	jA11	jA15	jA18	.
		s2	k30	k04	c10	c14	c20	c25	c32	c40	c49	c59	c69	c78	c87	c95	cA2	cA8	cA13	sA17	
		c1	c2	k13	k05	c15	c19	c26	c33	c41	c50	c60	c70	c79	c88	c96	cA3	cA9	k9a	kaa	
		cin	c1	k22	k14	k06	c21	c27	c34	c42	c51	c61	c71	c80	c89	c97	cA4	k8a	ka9	cin	$\mathbf{V}$
			j6	j9	j13	j18	j24	j31	j39	j48	j58	j68	j77	j86	j94	jA1	jA7	jA12	jA16	,	p[22]
			s4	k31	k23	k15	k07	k08	c35	c43	c52	c62	c72	c81	c90	c98	k7a	sA10	sA14		cA19
			s5	k40	k32	k24	k16	k17	c36	c44	c53	c63	c73	c82	c91	k6a	k89	sA11	sA15		
			cin	s7	k41	k33	k25	c28	k09	c45	c54	c64	c74	c83	k5a	k79	k98	k99	cin		
			1	j10	j14	j19	j25	j32	j40	j49	j59	j69	j78	j87	j95	jA2	jA8	jA13	т		
				s8	k50	k42	k34	k26	k18	k19	c55	c65	k3a	k4a	k69	k97	sA5	ka8			
				s9	s12	k51	k43	k35	k27	k28	k1a	k2a	k49	k59	k78	ka6	sA6	sA12			
			ļ	cin	s13	k60	k52	k44	k36	koa	k29	k39	k58	k68	k87	k88	sA7	cin	I		
					j15	j20	j26	J33	j41	j50	j60	j70	j79	j88	<u>j96</u>	JA3	JA9	r			
					s14	s16	k61	k53	k45	k55	k38	k48	k67	k77	k96	s99	sA8				
					s11	s1/	k/0	k62	k54	k46	k47	k57	k/8	K86	ka5	s100	ka/				
				I	cin	S18	cin	K/1	K63	K37	K56	K66	K85	K95	IS95	SA1	cin	l			
					1	121	J27	J34	J42	J51 -21	161	J/1	180	J89	J97	JA4					
						s20	s22	S32	K72	CZ1	K65	K75	K94	584 - 95	s92	SAZ					
						519	523	533	KØ1	K73	K74	K84	Ka3	585	593	SA3					
					I	cin	524	KOU ;2	K90	K0Z	ко <u>э</u> ;сэ	K95	:01	:00	:00	cin					
						ſ	J20	ر د20	J45 627	J52	102	572	JO1	190	198						
							525	523	537		K92	s00 s67	\$75	507 c 8 8	s90 c07						
							s20	53U	s30	cin	Ka I	507	\$70	500 ka4	s97						
						I	321	i36	335 іЛЛ	153	i63	i73	i82	iQ1	CIII						
								s34	<del>540</del>	s46	\$56	569	578	589	1						
								s354	s40	s47	s57	s70	\$79	\$90							
								cin	\$42	s48	\$58	s71	\$80	cin							
									i45	i54	i64	i74	i83		I						
									s43	s49	s59	s72	s81	1							
									s44	s50	s60	s73	s82								
									cin	s51	s61	ka2	cin								
										j55	j65			1							
										s52	s62	ſ									
										s53	s63										
										s54	s64										

Figure 4. Modify array multiplication for half precision as reference to double precision arrangement

w.F	loat (	O.61xd)	) - [Def	ault.	wcfg*]															
w F	ile	Edit	View	Sim	ulation	Window	Layout	Help												
Ж	D		0	ß	⊂   #	in ↓	10	°a ⊟	•	P 15	) 🏓 🌶	ء 🕺 🦻	۵	•	: <b>2</b> r	t if	- AL	<b>G</b> •	₽X	1.00us
Æ															2,572.1	74 ns				
<u>چ</u>	Nai	ne		Va	_,	1,500 ns				2,000 ns				2,500	ns				3,0	00 ns
2		🔓 sxi		0																
~		ll <mark>e</mark> sxu		0																
6		ling syl		0																
$\odot$		l 🖬 syu		0																
<b>*</b>		ll <mark>a</mark> cin		0																
<b>⊅</b> r		📩 exi[1	10:0]	10							101010101	101								
-		📩 exu[	[10:0]	11							110011001	110								$\square$
Ĭ		🔓 eyl[1	10:0]	11							111000111	100								
r I	•	📩 eyu[	[10:0]	11							111100001	111								
1		🚡 fxl[5	2:0]	10				1010	10 10 10 10	10 10 10 10 1	01010101010	010101010	10 10 10 10	10101	0101					$ \rightarrow $
LSH		🚡 fxu[	52:0]	11				1100	11001100	110011001	100110011	1001100	11001100	11001	1001					
3		🚡 fyl[5	2:0]	11				11100	00111000	111000111	000111000	0111000	11100011	10001	1100					
5,10		🚡 fyu[	52:0]	11				11110	00001111	000011110	000111100	00011110	00001111	00001	1110					$\square$

Fig.5 case1 inputs waveforms

# International Journal of Computer & Organization Trends –Volume 2 Issue 2 March to April 2012

eer F	Float (O.61xd) - [Default.wcfg*]																						
202 I	File I	Edit V	iew	Sim	ulati	on	Win	dow	La	ayout	: He	elp											
Ж	D	6×	۲	5	୍ୱ	A	A	$\downarrow \\$	Î	3	5			6	۶	<b>k?</b>	€	P	ø	۶	2	1	-
Æ	•	fzl[105:0]	10	10010	11110	1101000	001001	011110	011010	0000100	101111	000001	111101	001000	0100001	111011	010000	100101	11101	1000000	1001	100	
0	► = ► =	EXzi[10:0]	00	11000	0011	,00000	110000	001100	00000	1100000	010110	000	01110	010	/11001	010111	10110	111101	10111	011011	1101		
8	•	EXzu[10:0]	00									001	11101	110									
2	16	szl	0		-											-							
	12	flag1	0																				
œ	16	flag2	0																				
$\Theta$	•	a1[52:0]	10					1	01010	101010	101010	101010	101010	010101	0101010	10 10 10	0101					$ \ge $	
1	► 📲	a2(52:0) b1[52:0]	11					1	110011	111000	111000	111000	111001	011100	01110011	111000	11001					$\dashv$	
<b>⊉</b> r	▶ 🐝	b2[52:0]	11					1	11100	001111	000011	110000	11110	000111	1000011	1100001	1110						
-	▶ 🐝	ezi[10:0]	10									100	01110	001								$ \ge $	
Í	16	o_f1	1									101	11101	101									
1	16	o_f2	1																				

Fig.6 case1 outputs waveforms

				TABLE I		
CASES	FOR	64 BIT	INTERVAL	ARITHMETIC	ARRAY	MULTIPLICATION

		INP	UT	o/p	z	Ex ADD		Bias 102	3	
Case	Condition	S <sub>xl</sub>	S <sub>yl</sub>	Szi	$Z_I = X_I Y_I$	Internal wire ez	OF	E <sub>zi</sub> = e <sub>zi</sub> -1023	f	f <sub>Zl</sub> (16'h)
1	$X_1 > 0, Y_1 > 0$	0	0	0		10001110001	1	1110010	0	ECF684BDA12F684C
		Sxu	Syu	SZu	Zu=XuYu	Internal wire ezu	OF	E <sub>zu</sub> = eZ <sub>u</sub> -1023	f	f <sub>Zu</sub> (16'h)
		0	0	0		10111101101	1	111101110	0	2EDEDEDEDEDEDEE
2	X <sub>l</sub> >0,Y <sub>u</sub> < 0	0	1	1	Zl = XuYl	10110000010	1	110000011	0	82BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
		0	1	1	Zu=XlYu	10011011100	1	1101110	0	8275F5F5F5F5F5F6
3	X <sub>u</sub> <0,Y <sub>I</sub> >0	1	0	1	Zl=Xlyu	10011011100	1	11011101	0	8275F5F5F5F5F5F6
		1	0	1	Zu=XuYl	10110000010	1	110000011	0	82BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
4	X <sub>u</sub> <0,Y <sub>u</sub> <0	1	1	0	Zl=XuYu	10111101101	1	111101110	0	2EDEDEDEDEDEDEE
		1	1	0	Zu=XIYI	10001110001	1	1110010	0	ECF684BDA12F684C
5	X <sub>1</sub> <0 <x<sub>u,Y<sub>1</sub>&gt;0</x<sub>	1	0	1	Zl=XlYu	10011011100	1	110000011	0	8275F5F5F5F5F5F6
		0	0	0	Zu=XuYu	10111101101	1	111101110	0	2EDEDEDEDEDEDEE
6	X1<0 <x1,y1<0< td=""><td>1</td><td>1</td><td>1</td><td>Zl=XuYl</td><td>10110000010</td><td>1</td><td>110000011</td><td>0</td><td>82BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB</td></x1,y1<0<>	1	1	1	Zl=XuYl	10110000010	1	110000011	0	82BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
		0	1	0	Zu=XIYI	10001110001	1	1110010	0	ECF684BDA12F684C
7	X <sub>1</sub> >0,Y <sub>1</sub> <0 <y<sub>u</y<sub>	0	1	1	Zl=XuYl	10110000010	1	110000011	0	82BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
		0	0	0	Zu=XuYu	10111101101	1	111101110	0	2EDEDEDEDEDEDEE
8	X <sub>u</sub> <0,Y <sub>I</sub> <0 <y<sub>u</y<sub>	1	1	1	ZI=XIYu	10011011100	1	110000011	0	8275F5F5F5F5F5F6
		1	0	0	Zu=XIYI	10001110001	1	1110010	0	ECF684BDA12F684C
9	X <sub>1</sub> <0 <x<sub>u,Y<sub>1</sub>&lt;0<y<sub>u</y<sub></x<sub>	1	1	1	Zl=Xuyu	10111101101	1	111101110	0	2EDEDEDEDEDEDEE
		0	0	0	Zu=XIYu	10111101101	0	110000011	0	8275F5F5F5F5F5F6

Device utilization summary for 64 bit multiplication										
Area analysis	floating point	interval arithmetic								
Number of Slices	4212	8603								
Number of Slice Flip Flops		234								
Number of 4 input LUTs	7326	14967								
umber of Ios	261	499								
Number of bonded IOBs	261	499								
IOB Flip Flops		2								
Number of GCLKs	1	1								
Real time delay	220 ns	358 ns								
Maximum combination	nal path delay a	nalysis in ns								
Critical path delay	465.005	421.563								
Memory	y in kilobytes									
Total memory usage	287584	334140								

TABLE II ANALYSIS REPORT

# V. CONCLUSION

Interval arithmetic provides reliability and accuracy by computing a lower and upper bound in which result is guaranteed to reside. Concept of carry look ahead for 11 bit exponent adder is used which reduces the delay. Concept of carry save adder in array multiplication is used instead of half adders and full adders which reduces the number of gates and delay. 334140 kilobytes memory is required. 421.563 ns is the maximum critical path delay for 64 bit interval arithmetic array multiplier.

64 bit interval arithmetic array multiplier requires almost twice real time delay compared to 64 bit floating point array multiplier. So speed of interval arithmetic array multiplier decreases and area increases.

#### REFERENCES

- Josh Milthorpe and Alistair Rendell "Learning to live with errors: A fresh look at floating-point computation", Australian National University, Computing Conference 2005
- [2] Gupte, ruchir "Interval arithmetic logic unit for dsp and control applications", Electrical and Computer Engineering, Raleigh 2006
- [3] Rajashekar Shettar, Dr.R.M.Banakar and Dr. P.S.V.Nataraj, "Design and Implementation of Interval Arithmetic Algorithms", First International Conference on Industrial and Information Systems, ICIIS 2006, 8 - 11 August 2006, Sri Lanka
- [4] Wikipedia, the free encyclopedia
- [5] Rajashekar Shettar, Dr.R.M.Banakar and Dr. P.S.V.Nataraj, "Implementation of Interval Arithmetic Algorithms on FPGAS", International Conference on Computational Intelligence and Multimedia Applications 2007, © 2007 IEEE
- [6] Alexandru Amaricai Mircea Vladuaiu Lucian Prodan Mihai Udrescu Oana Boncalo "Design of Addition and Multiplication Units for High Performance Interval Arithmetic Processor",

Computer Science and Engineering Department, ©2007 IEEE

- [7] Michael J. Schulte, Member, IEEE, and Earl E. Swart Zlander Jr., Fellow, IEEE, "A Performance Comparison Study on Multiplier Designs", IEEE Transaction On Computers, May 2000
- [8] Yong Dou S. Vassiliadis G. K. KuZmanov G. N. Gaydadjiev, "64-bit Floating-Point FPGA Matrix Multiplication", National Laboratory for Computer Engineering, FPGA'05, Monterey, California, USA, February 20–22, 2005
- [9] Anane Nadjia, Anane Mohamed, Bessalah Hamid, Issad Mohamed & Messaoudi khadidja, "Hardware Algorithm for Variable Precision Multiplication on FPGA" © 2009 IEEE
- [10] James E. Stine and Michael J. Schulte "A Combined Interval and Floating Point Multiplier", Computer Architecture and Arithmetic Laboratory, Electrical Engineering and Computer Science Department, Lehigh University, Bethlehem, PA 18015
- [11] Sparc Architecture Manual
- [12] Lab-Report ECAD, "4bit multiplier using Mentor Graphics"
- [13] Prof. LohCS3220- Processor Design "Carry-Save Addition" - Spring 2005, February 2, 2005
- [14] "Carry Save Adder Trees in Multipliers" E C E N 6 2
  6 3 A d v a n c e d V L S I D e s i g n November 3, 1999
- [15] C. N. Marimuthu1, P. Thangaraj "Low Power High Performance Multiplier", Anna University, Tamil nadu, India