ISTANBUL UNIVERSITY –
JOURNAL OF ELECTRICAL & ELECTRONICS ENGINEERING

YEAR : 2006 (169 - 174)
VOLUME : 6
NUMBER : 2

A FAST MULTIPLIER HARDWARE DESIGN FOR INTERVAL ARITHMETIC

[1]Ahmet SERTBAŞ [2]Hani EL-ABDALLAH [3]Fethullah KARABIBER

[1,2,3]Istanbul University, Engineering Faculty, Computer Engineering Department
34320, Avcilar, Istanbul, Turkey

[1]E-mail: asertbas@istanbul.edu.tr [2]E-mail: hani956@yahoo.com

[3]E-mail: fetullah@istanbul.edu.tr

ABSTRACT

In this paper, a new parallel hardware unit for interval multiplication is presented. Using the VHDL synthesis results, the area and delay estimates for the new design are given. Compared to previous hardware interval multipliers, our design is faster, but, requires more area.

Keywords: *Interval arithmetic, multipliers, VHDL synthesis, performance analysis*

1. INTRODUCTION

As computer applications have led to rapid increase in computing power, reliable computation requires results to be highly accurate. In most cases, computations that include real-valued numbers contain inaccuracy and results are almost unreliable due to catastrophic cancellations and rounding off. On the other hand, arithmetic errors in embedded systems can lead to disaster. For example, a plane may crash, a rocket may explode, or an engine may fail to operate.

The fundamental problem with most real-number computations is that their accuracy is not guaranteed. Increasing precision does not prevent this. Small errors can accumulate rapidly and limitations in the representation of numbers may lead to totally wrong results. For example, equation (1) shows an equation causing numerical error:

$$f = 333.75\, b^6 + a^2 (11a^2 b^2 - b^6 - 121b^4 - 2.0) + 5.5\, b^8 + a / (2b) \quad (1)$$

For a = 77617.0 and b = 33096.0, this equation yields f = 1.17260 when solved using single precision, double precision, and extended precision arithmetic. Increasing the precision seems to validate the results. However, the correct answer is actually f = -0.827396 x 10-17.

As conventional real-valued computations contain inaccuracy that makes results unreliable due to catastrophic cancellation and rounding off results, for the reliable and accurate computations, interval arithmetic can produce good results. Interval arithmetic which deals with sets of intervals provides reliability and accuracy needed by computing the lower and upper bounds xl, xu in which the true result x-true relies. So, the interval X = [xl, xu] bounds the true result such that: xl < = x-true < = xu. When one or both end point could not be represented during computation, outward

Received Date : *03.03.2005*
Accepted Date: *23.06.2006*

rounding towards –infinity, +infinity for each xl, xu respectively, guaranties that the resulting interval includes the true result. For example, the interval [1.56, 2.34] is outward rounded to two decimal digits resulting the interval [1.5, 2.4].
In the literature, interval arithmetic (and interval analysis) was firstly defined by Ramon E. Moore[1]. Some software packages are developed to support interval arithmetic and can provide the method of bounding the true result like C-XSC [1], PROFL [2], INTLIB [3] and recently, a built-in support is added for interval arithmetic to FORTRAN [4], but yet, enough performance is not achieved yet. The performance is considered acceptable if it does not exceed a factor of five to conventional real-world arithmetic [5]. Unfortunately software implementations achieve a factor of 20 to 100 of a conventional floating point arithmetic.

Regarding to inefficiency in achieving performance in determining the suitable case and rounding process by software implementations for interval arithmetic, it became necessary to search for hardware structures that can automatically select the interval endpoints and serve the rounding process correctly.

In recent years, to improve the performance of interval arithmetic, some hardware designs are proposed [6-10]. In [2], the serial and parallel interval multipliers that lead to a considerable increase in performance were presented. Many hardware structures, that improve interval multiplication and handle efficiently rounding of endpoints are implemented. An optimization of delay is achieved but with a slight overhead in area either by improving existing multiplication structures by adding some registers, multiplexers and some sort of comparing units, with some change in the previous logic or implementing new ones with their own hardware structure and logic. Selecting the right case by examining the sign bits in hardware units is based on the same method used in software.

In this paper, a fast interval multiplier is designed. It is the improved design of the parallel interval multiplier given in[2]. Section 2 describes an interval multiplication analytically. In Section 3, the new interval multiplier is presented. In Section 4, we give area and delay estimates for the new design and compare it to the previous interval multipliers in [2]. Section 5 presents the conclusions.

2. INTERVAL MULTIPLICATION

Multiplication could be performed on computers that support either single or double precision. The result have to be outward rounded towards –infinity, +infinity for both bounds. If double length is supported, the interval product can be computed as:

$$Z=[\blacktriangledown(\min(xl*yl, xl*yu, xu*yl, xu*yu)), \blacktriangle(\max(xl*yl, xl*yu, xu*yl, xu*yu))] \quad (2)$$

This process needs four multiplications(for the endpoint products) and four comparisons (to get min and max values) to obtain the result.
If double length is not supported, the interval product Z = X * Y can be computed as:

$$Z=[\min(\blacktriangledown(xl*yl), rd(xl*yu), \blacktriangledown(xu*yl), rd(xu*yu)), \max(\blacktriangle(xl*yl), \blacktriangle(xl*yu), \blacktriangle(xu*yl), \blacktriangle(xu*yu))]. \quad (3)$$

where ▼ and ▲ represent rounding downward toward negative infinity and upward toward positive infinity, respectively.

This also needs eight multiplications for the rounded products and six comparisons to obtain the min and max values [11]. In order to reduce the number of multiplications, the sign bit of the endpoint xl, xu, yl, yu can be examined in advance to determine the product of the right result of zl, zu.

The sign bits indicate weather multiplied intervals X, Y are greater than, less than or contain zero, so, 9 cases for multiplication can be classified. The first eight cases need only two rounded multiplications to determine zl, zu, where the last case when both intervals X, Y contain zero needs four rounded multiplications and two comparisons to determine zl, zu.

All these procedures suffer from conditional statements for achieving the right choice of endpoints to be multiplied. The algorithm above declares the conditional branches needed to perform multiplication. which greatly increases the time. Rounding the results downward towards –infinity and upward towards +infinity in computation of results for each case decrease performance of multiplication.

3. THE INTERVAL MULTIPLIER DESIGN

In this paper, the method in [2], which provides a considerable increase in performance, is employed for deciding to the interval endpoints and rounding the results. To produce the multiplication results z_l and z_u are determined the interval endpoints $\{x_l, x_u, y_l, y_u\}$ to be multiplied together by examining their sign bits (Sxl, Sxu, Syl, Syu), as shown in Table 1.

Shown in Table 1, the control bits Zc, tx1, tx2, ty1, ty2 select the endpoints to be multiplied at each multiplier. Fig. 1 shows the block diagram of the new design for the parallel interval multiplier that uses 4 IEEE standart multipliers, 2 min/max units(fig.2), six registers, 6 multiplexers for choosing the inputs to be multiplied and 2 multiplexers for selecting the results from the multipliers or the min/max units.

```
let mn = min(▼(xl * yu),  ▼ (xu * yl)), mx = max(▼ (xl * yl),  ▼ (xu * yu)),
if (xl >= 0) { if (yl >= 0) { zl = ▼ (xl * yl); zu = ▲ (xu * yu) ; }
else  if (yu < 0){ zl = ▼ (xu * yl); zu = ▲ (xl * yu)
else { zl = ▼ (xu * yl) ;  zu = ▲ (xu * yu) ; }
else if (xu < 0){ if (yl >= 0){ zl = ▼ (xl * yu); zu = ▲ (xu * yl) ;}
else if (yu < 0) { zl =▼ (xu * yu) ; zu = ▲ (xl * yl) ; }
else { zl = ▼ (xl * yu) ; zu =▲ (xl * yl); }
else { if (yl >= 0) { zl = ▼ (xl * yu) ; zu = ▲ (xu * yu) ;}
else if (yu < 0) { zl = ▼ (xu * yl) ; zu = ▲ (xl * yl) ;}
else {a = xl * yu ; b = xu * yl ; zl = min(a, b) ; c = xl * yl ; d = xu * yu ; zu = max(c, d) ;
```

Table 1: All Cases for Interval Multiplication

Case	Interval X	Interval Y	Sxl	Sxu	Syl	Syu	Z = X * Y	zc	tx1	tx2	ty1	ty2
1	X>[0,0]	Y>[0,0]	0	0	0	0	[xl*yl, xu*yu]	0	1	1	1	1
2	X>[0,0]	Y<[0,0]	0	0	1	1	[xu*yl, xl*yu]	0	0	0	1	1
3	X<[0,0]	Y>[0,0]	1	1	0	0	[xl*yu, xu*yl]	0	1	1	0	0
4	X<[0,0]	Y<.0,0]	1	1	1	1	[xu*yu, xl*yl]	0	0	0	0	0
5	0 € X	Y>[0,0]	1	0	0	0	[xl*yu, xu*yu]	0	1	1	0	1
6	0 € X	Y<[0,0]	1	0	1	1	[xu*yl, xl*yl]	0	0	0	1	0
7	X>[0,0]	0 € Y	0	0	1	0	[xu*yl, xu*yu]	0	0	1	1	1
8	X<[0,0]	0 € Y	1	1	1	0	[xl*yu, xl*yl]	0	1	0	0	0
9	0 € X	0 € Y	1	0	1	0	*[mn, mx]	1	1	1	1	1

*mn=min(▼(xl*yu), ▼(xu*yl)), mx=max(▲(xl*yl), ▲(xu*yu))

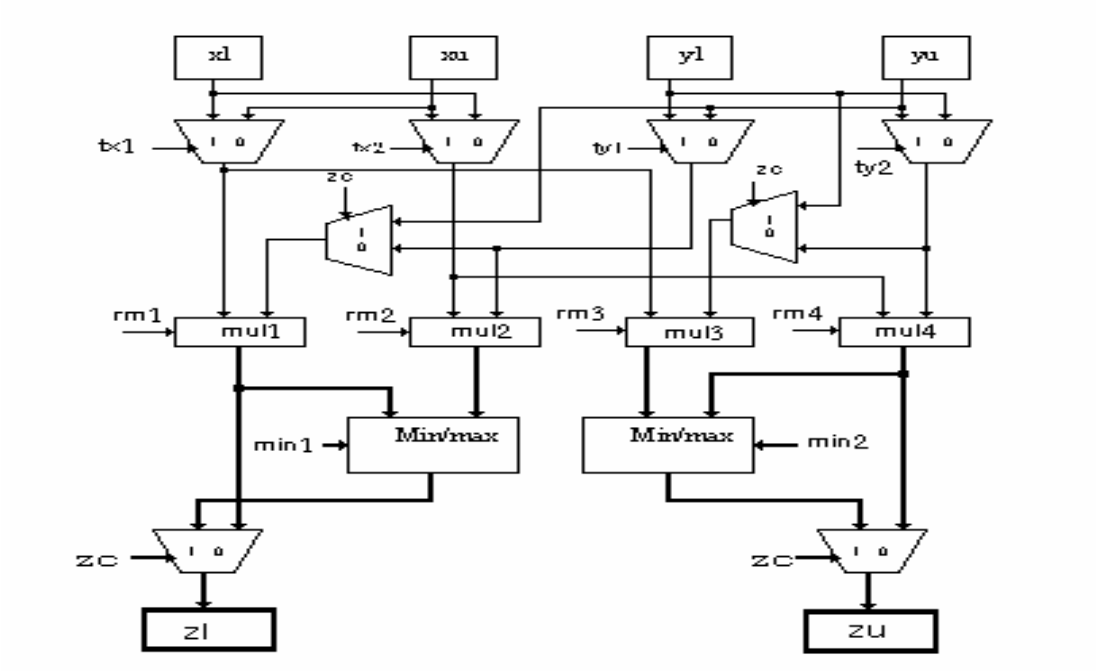

Fig.1. Proposed structure for interval parallel multiplier

Table 2. Execution steps for Case 9.

Cycle	C	Action
1	0	r1 =▼(xl*yu), r2 = ▼(xu*yl), r3 =▲(xl*yl), r4 =▲(xu*yu)
2	1	Set *zl*=min(r1, r2) , Set *zu*=max(r3, r4)

Table 3. Performance comparisons of interval multipliers

Performance Metrics	Serial (Schulte)	Parallel (Schulte)	Proposed Interval Multiplier
Cycle_count1 (Zc=0)	2	1	1
Cycle_count2 (Zc=1)	5	3	2
Total Logic Cells	866	1373	2516
Chip Area (Estm.)/mm^2	60	96	173
Clock Frequency/MHz.	75.75	80.65	77.65
Estm. Total Delay- ns	30.8	15.15	14.31

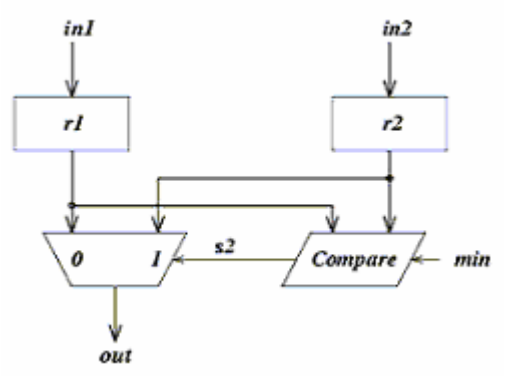

Fig. 2. Min/Max unit

The Boolean equations for the control and rounding mode bits are given as follows:

$$\begin{aligned} tx1 &= \overline{Syl} + Sxl * \overline{Syu} + Zc, \\ tx2 &= \overline{Syl} + \overline{Sxl} * \overline{Syu} + Zc, \\ ty1 &= \overline{Sxl} + Syl * \overline{Sxu} + Zc, \\ ty2 &= \overline{Sxl} + \overline{Syl} * \overline{Sxu} + Zc, \\ Zc &= Sxl * \overline{Sxu} * Syl * \overline{Syu}, \\ rm1 &= rm2 = c, \quad rm3 = rm4 = \overline{c} \end{aligned} \tag{4}$$

Where, the rm1, rm2 mode bits are for rounding towards – infinity and the rm3, rm4 are for rounding towards + infinity, also, c is for the clock cycle.

If X and Y do not both contain zero (the first 8 cases), a single cycle is required to compute the lower and upper interval endpoints of the product. On the other hand, if X and Y both contain zero (Case 9), only two cycles are sufficient to perform the interval computation, as shown in Table 2. On Case 9, at the first cycle, productions of the endpoints are done in parallel with the 4 multipliers. At the second cycle, the 1st min/max unit determine the minimum value of the 1st and 2nd multiplier's output according to the min1 control signal, while the 2nd min/max unit determines the maximum value of the 3rd and 4th multipliers output due to min2, the output registers are loaded with products required for the lower and upper endpoints comprising the result. Table 2 shows the steps for the parallel interval multiplier for Case 9.

The lower bound *zl* is selected from the multipliers' outputs r1, r2, according to the control bit min1. When min1=1, if r1< r2, then *zl*=r1 else *zl* = r2. The upper bound *zu* is selected from the multipliers' outputs r3, r4, according to the control bit min2. When min2=1, if r3 < r4, then *zu* = r3 else *zl* = r4.

4. COMPARISON

The all architectures and behaviors of serial, parallel and proposed interval multipliers are simulated for functionality at the logic level using using Model-Sim and synthesized for obtaining total logic cells, estimated chip areas, clock cycle frequency and total operation delays using the Quartus VHDL (Version II). Here, the total operation delay is computed as follows:

Estimated Total Delay= [Cycle_count1(8/9)+Cycle_count2*(1/9)] /Clock_frequency* ***(5)***

As shown in Fig.3, compared to the serial and parallel interval multipliers our design requires roughly 190 and 80 percent more area, respectively. Also, the new interval multiplier has cycle time approximately 3 percent shorter than the serial, but 3 percent longer than the parallel multiplier. Fortunately, with respect to the estimated total delay, our interval multiplier is 6 and 105 percent faster than the parallel and the serial multipliers, respectively. The total results obtained by VHDL simulations are given in Table 3.

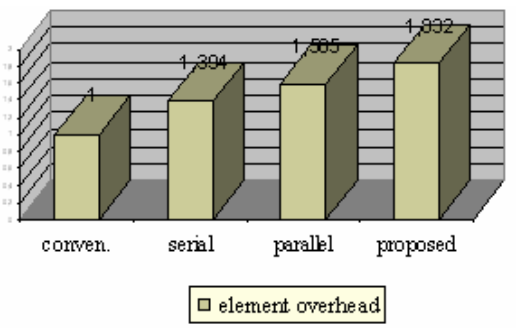

Fig.3. Relative compares of the interval multipliers with respect to the number of total logic gates

5. CONCLUSIONS

This paper proposes a hardware design for the interval multiplier. Serial interval multiplier consumes 2 cycles for executing cases 1-8 and 5 cycles for case 9 to handle interval multiplication, where parallel interval multiplier needs just

one cycle to execute the cases 1-8 and 3 cycles for case 9. The proposed parallel multiplier needs just 2 cycles for case 9, with a cycle not to be lost and only 1 cycle for cases 1-8.

According to total logic elements, the serial interval multiplier is more than conventional floating point multiplier of 1.35 where the parallel is of 2.16 and the proposed interval multiplier is, the higher, of 3.95. but improved parallel is nearly like conventional usage with only 1.083, where parallel is 1.16 and the serial is 2.25.

These estimations show that the new design gives a little more speedup (roughly 6 percent), but, 80 percent more area than the parallel interval multiplier referenced in [2]. Therefore, the proposed interval multiplier can be used for the need of fast interval computing. By adjusting the selection bits of the multiplexers (t_{x1}, t_{x2}, t_{y1}, t_{y2} and z_c), the new interval multiplier can also perform the interval structures chosen to compare in this work.

The next step, besides the search for efficient software implementations able to support interval arithmetic within permitable ranges of delay, is to work for monitoring other arithmetic functions like the exponent.

REFERENCES

[1] Moore R.E., 1966, *Interval Analysis*, Englewood Cliffs: Prentice Hall.

[2] Schulte M.J., Bıckersatff K.C., Schwartzlander E.Jr., 1996, "Hardware Units for Interval Multiplication" *Proceedings of the 2nd Workshop of Computer Arithmetic, Interval, and Symbolic Computations*, pp. 85-87.

[3] Williams G. S., "Processor Support For Interval Arithmetic" *thesis for master degree*, Lehigh University, May 1998.

.

[4] Goldberg D.,1991 "What Every Computer Scientist Should Know About Floating Point Arithmetic" *ACM Computing Surveys*, vol. 23, pp. 5-48.

[5] Arnold D.N.,1997 "Disasters Caused by Computer Arithmetic Errors" available from Internet URL http://www.ima.umn.edu/~arnold/455.f96/disasters.html February, 1997.

[6] Rumph S.M, 1988 "Algorithms for Verified Inclusions: Theory and Practice" *Reliability in Computing*, San Diego: Academic Press, 1988.

[7] Kearfott R.B. et al.,1996, "A Specific Proposal for Interval Arithmetic in FORTRAN" http://interval.louisiana.edu/F90/f96-pro.asc, March 96.

[8] Walster G.W., 1996, "Stimulating Hardware and Software Support for Interval Arithmetic" *Applications of Interval Computations*, pp. 405-416, 1996.

[9] Alefeld G. and Herzberger R.,1983 , *Introduction to Interval Computations*, New York: Academic Press.

[10] Walster G.W., "Interval Arithmetic: The New Floating-Point Arithmetic Paradigm," http://www.mscs.mu.edu/~globsol/readings.html, March, 1998

[11] Coriliss G.F.,1993 "Comparing Software Packages for Interval Arithmetic," in *Abstracts of the International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics.*