A Combined Interval and Floating Point Multiplier

James E. Stine and Michael J. Schulte

Computer Architecture and Arithmetic Laboratory
Electrical Engineering and Computer Science Department
Lehigh University
Bethlehem, PA 18015

Abstract

Interval arithmetic provides an efficient method for monitoring and controlling errors in numerical calculations. However, existing software packages for interval arithmetic are often too slow for numerically intensive computations. This paper presents the design of a multiplier that performs either interval or floating point multiplication. This multiplier requires only slightly more area and delay than a conventional floating point multiplier, and is one to two orders of magnitude faster than software implementations of interval multiplication.

1 Introduction

The performance of conventional microprocessors currently increases at a rate of approximately 55 percent per year and is expected to increase by a factor of 50 over the next ten years [1]. This rapid increase in computing power has led to a greater reliance on results produced by computer simulation and modeling. Although many areas depend on computer generated results for reliable information, roundoff error and catastrophic cancellation in floating point computations can cause results to be highly inaccurate, with little or no indication [2].

Interval arithmetic provides an efficient method for monitoring and controlling errors in floating point computations, by producing two values for each result [3]. The two values correspond to the lower and upper endpoints of an interval, which contains the true result. The width of the interval indicates the accuracy of the result. If the interval endpoints are not representable, they are outward rounded. For example, if each interval endpoint uses three decimal digits, the interval multiplication $[1.23, 1.24] \times [2.56, 2.57] = [3.1488, 3.1868]$ is outward rounded to $[3.14, 3.19]$. Although naive use of interval arithmetic can result in wide intervals, algorithms have been developed that produce narrow intervals for many applications [4].

Because of its ability to provide reliable results, several interval software packages been developed [5], [6]. Although these packages improve the reliability of arithmetic computations, they are often too slow for numerically intensive applications. As reported in [7], interval arithmetic operations implemented in software are often tens to hundreds of times slower than corresponding floating point operations. This is because interval arithmetic has overhead due to function calls, changing the rounding mode, selecting the appropriate interval endpoints, and testing for special cases and exceptions.

Many researchers feel that it is necessary for the performance of interval arithmetic to be within a factor of five of floating point arithmetic for it to gain general acceptance [7]. To this end, the GNU Fortran Compiler is being modified to provide support for an interval data type [8], based on the Interval Arithmetic Specification [9]. However, to achieve high performance, hardware support for interval arithmetic is also required. Previous hardware designs for interval arithmetic employ special-purpose coprocessors [10], [11], or functional units [12], [13]. Although these designs improve the performance of interval arithmetic, the cost of adding specialized interval hardware can be prohibitive.

This paper presents the design of a combined interval and floating point multiplier, which is constructed by adding a small amount of hardware to a conventional floating point multiplier. This approach offers the performance benefits of a dedicated interval multiplier, but requires significantly less hardware. Section 2 gives an overview of previous software and hardware techniques for interval multiplication. Section 3 shows how a floating point multiplier is modified to enable it to also perform interval multiplication. Section 4 provides area and delay estimates for the combined interval and floating point multiplier. Section 5 gives our conclusions.

0-8186-8409-7/98 $10.00

2 Interval multiplication

As presented in [3], multiplication of the intervals $X = [x_l, x_u]$ and $Y = [y_l, y_u]$ is defined as:

$$Z = X \cdot Y = [\min(x_l y_l, x_l y_u, x_u y_l, x_u y_u), \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)]$$

If the floating point multiplier is capable of producing a product that is at least twice as precise as its input operands, the interval product can be computed as:

$$Z = X \cdot Y = [\nabla \min(x_l y_l, x_l y_u, x_u y_l, x_u y_u), \Delta \max(x_l y_l, x_l y_u, x_u y_l, x_u y_u)]$$

where ∇ denotes rounding down toward negative infinity and Δ denotes rounding up toward positive infinity. Based on this definition, computing the interval endpoints of Z requires four multiplications (to obtain the double length products $x_l y_l$, $x_l y_u$, $x_u y_l$, and $x_u y_u$), four comparison operations (to obtain the minimum and maximum values), and two directed roundings [12].

In practice, however, interval arithmetic is typically performed on double precision operands, and a double precision result is produced. In this case, $X \cdot Y$ can be computed as:

$$Z = X \cdot Y = [\min(\nabla x_l y_l, \nabla x_l y_u, \nabla x_u y_l, \nabla x_u y_u), \max(\Delta x_l y_l, \Delta x_l y_u, \Delta x_u y_l, \Delta x_u y_u)]$$

Based on this definition, the interval endpoints of Z are computed by performing eight multiplications (with rounded products $\nabla x_l y_l$, $\nabla x_l y_u$, $\nabla x_u y_l$, $\nabla x_u y_u$, $\Delta x_l y_l$ $\Delta x_l y_u$, $\Delta x_u y_l$, and $\Delta x_u y_u$), and six comparison operations (three to obtain the maximum value and three to obtain the minimum value) [12].

To reduce the number of multiplications, a technique originally presented in [3] can be employed. With this technique, the signs of the interval endpoints are examined to determine the endpoints to multiply together to produce the interval product. 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 Table 1. In Case 9, when both X and Y contain zero, $mn = \min(\nabla x_l y_u, \nabla x_u y_l)$ and $mx = \max(\Delta x_l y_l, \Delta x_u y_u)$. For this case, the endpoints to be multiplied cannot be determined based solely on their signs. Instead, it is necessary to perform four multiplications and two comparisons to determine the lower and upper endpoints. In practice, Case 9 occurs infrequently, since intervals that contain zero have infinite relative error. The other eight cases require only two multiplications.

Although examining the signs of the interval endpoints decreases the number of multiplications, its software implementation requires several conditional branches to determine which interval endpoints should be multiplied. This is illustrated in Figure 1, which shows a software implementation of interval multiplication, similar to the one used in [5].

```
rm = get_round_mode();
round_down();
if (x_l > 0) {
  if (y_l > 0) {
    z_l = x_l · y_l; round_up(); z_u = x_u · y_u;
  } else if (y_u < 0) {
    z_l = x_u · y_l; round_up(); z_u = x_l · y_u;
  } else {
    z_l = x_u · y_l; round_up(); z_u = x_u · y_u;
  }
} else if (x_u < 0) {
  if (y_l > 0) {
    z_l = x_l · y_u; round_up(); z_u = x_u · y_l;
  } else if (y_u < 0) {
    z_l = x_u · y_u; round_up(); z_u = x_l · y_l;
  } else {
    z_l = x_l · y_u; round_up(); z_u = x_l · y_l;
  }
} else {
  if (y_l > 0) {
    z_l = x_l · y_u; round_up(); z_u = x_u · y_u;
  } else if (y_u < 0) {
    z_l = x_u · y_l; round_up(); z_u = x_l · y_l;
  } else {
    m_l = x_l · y_u; n_l = x_u · y_l;
    z_l = min(m_l, n_l);
    round_up();
    m_u = x_l · y_l; n_u = x_u · y_u;
    z_u = max(m_u, n_u);
  }
}
set_round_mode(rm);
```

Figure 1: Code for interval multiplication.

Software implementations of interval arithmetic are typically between one and two orders of magnitude slower than floating point multiplication. The large number of conditional statements used to select be-

Table 1: Nine cases for interval multiplication.

Case	Condition	Z	Example
1	$x_l > 0, y_l > 0$	$[x_l y_l, x_u y_u]$	$[1,2] \cdot [3,4] = [3,8]$
2	$x_l > 0, y_u < 0$	$[x_u y_l, x_l y_u]$	$[1,2] \cdot [-4,-3] = [-8,-3]$
3	$x_u < 0, y_l > 0$	$[x_l y_u, x_u y_l]$	$[-2,-1] \cdot [3,4] = [-8,-3]$
4	$x_u < 0, y_u < 0$	$[x_u y_u, x_l y_l]$	$[-2,-1] \cdot [-4,-3] = [3,8]$
5	$x_l < 0 < x_u, y_l > 0$	$[x_l y_u, x_u y_u]$	$[-1,2] \cdot [3,4] = [-4,8]$
6	$x_l < 0 < x_u, y_u < 0$	$[x_u y_l, x_l y_l]$	$[-1,2] \cdot [-4,-3] = [-8,4]$
7	$x_l > 0, y_l < 0 < y_u$	$[x_u y_l, x_u y_u]$	$[1,2] \cdot [-4,3] = [-8,6]$
8	$x_u < 0, y_l < 0 < y_u$	$[x_l y_u, x_l y_l]$	$[-2,-1] \cdot [-4,3] = [-6,8]$
9	$x_l < 0 < x_u, y_l < 0 < y_u$	$[mn, mx]$	$[-2,1] \cdot [-4,3] = [-6,8]$

tween the nine cases greatly increases the time required to perform interval multiplication. This is especially true for pipelined and super-scalar processors, for which the performance penalty due to conditional branches is greater. Another performance limitation is that changing the rounding mode in software often requires a large number of cycles. As noted in [14], changing the rounding mode on IEEE processors can take as long as executing six floating point additions, due to an inefficient user interface. On many processors, changing the rounding mode causes the entire floating point pipeline to be flushed, which results in a delay of several cycles and severely limits parallel execution. Furthermore, software implementations of interval multiplication are typically implemented as subroutines, which adds overhead for subroutine calls and returns.

To improve the performance of interval multiplication, hardware designs for dedicated interval multipliers have been developed [12], [13]. In [12], several hardware techniques for performing interval multiplication are presented and compared. These comparisons show that hardware interval multipliers that examine the sign bits to determine the endpoints to be multiplied typically require less hardware and have shorter execution times than designs that compute eight rounded products or four double length products and then determine the minimum and maximum values. In [13], hardware interval multipliers are introduced that require less hardware and have shorter execution times than those given in [12]. Although these hardware implementations of interval multiplication are much faster than software implementations, they require a significant amount of dedicated hardware. For example, area estimates presented in [13] indicated that serial and parallel interval multipliers require approximately 15 and 110 percent more area than an IEEE double precision multiplier implemented in the same technology.

3 Interval/floating point multiplier

Rather than using dedicated hardware to implement interval multiplication, the interval multiplier can share hardware with an existing floating point multiplier. With this approach the performance benefits of a dedicated hardware interval multiplier are achieved at a relatively low cost. This section presents the design of a combined interval and floating point multiplier. The design for this multiplier is based upon the design of the serial interval multiplier presented in [13], with modifications made to enable the same multiplier to perform either interval or floating point multiplication.

The interval multiplier presented in this section is designed to handle normalized numbers in the IEEE double precision format [15]. As in [12], [13], denormalized numbers and special IEEE values (i.e., zero, infinity, and not-a-number) are assumed to be handled separately. To provide tight intervals that contain the correct result, these special cases should be handled as specified in [16]. IEEE double precision numbers are 64 bits long, and have a sign bit (s), an 11-bit biased exponent (e), and a 52-bit significand (f). The significand contains a *hidden one*, which gives it an actual precision of 53 bits. The value of a normalized IEEE double precision number is

$$(-1)^s \times 1.f \times 2^{e-1023}$$

Figure 2 shows a block diagram of an IEEE double precision multiplier. The multiplier 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. 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 multiplication. If the most

significant 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 2-bit rounding mode (rm).

Figure 3 shows a block diagram of the combined interval/floating point multiplier. Compared to the floating point multiplier, the combined multiplier requires two additional input registers and one additional output register. It is also requires control logic and multiplexors to select the interval endpoints to be multiplied, and control logic to set the rounding mode appropriately for interval multiplication.

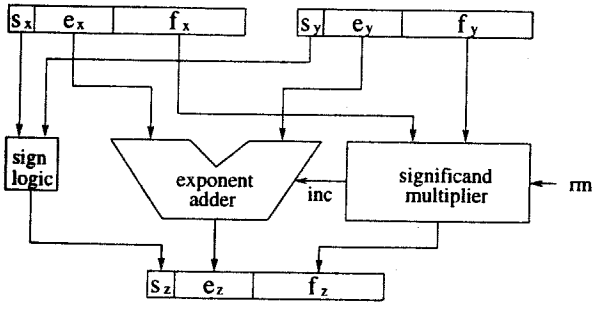

Figure 2: Floating point multiplier.

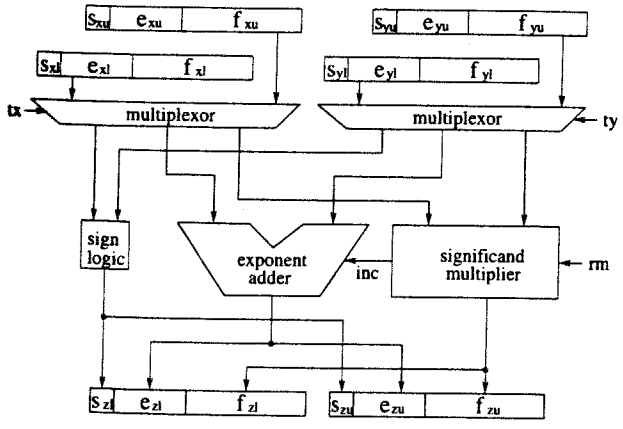

Figure 3: Interval/floating point multiplier.

Two multiplexors select the endpoints to be multiplied based on the toggle bits, tx and ty. If the toggle bit is one, the lower interval endpoint is selected. The values for the toggle bits are determined based on the sign bits of the interval endpoints, sxl, sxu, syl, and syu, and whether the lower or upper interval endpoint of the product is to be computed. The sign bits are one if the number is negative and zero if it is positive. A control bit le is set to one when the lower interval endpoint is computed and zero when the upper interval endpoint is computed.

Table 2 shows the value of the sign and toggle bits for each of the nine cases presented in Table 1. In Case 9, both intervals contain zero, and it is not possible to determine the interval endpoints to multiply based only on the sign bits. One approach for handling this case is to have the interval multiplier automatically take care of it [12], [13]. However, this approach requires a significant amount of additional hardware to store temporary values and compute the minimum and maximum. It also complicates the control logic, since three extra cycles are needed to perform interval multiplication when both intervals contain zero [13]. Since this case occurs infrequently, the approach taken in this paper is to have the interval multiplier detect that both intervals contain zero and signal a trap to software by setting the zero contained (zc) flag to one. For this case, the values for the toggle bits are irrelevant, which is indicated by X in Table 2.

To specify the operation performed by the multiplier a control bit fp is used. This bit is set to one for floating multiplication and zero for interval multiplication. It is assumed that when floating point multiplication is performed, the input operands are stored in the registers for x_l and y_l. In this situation, both toggle bits should to be one and the zero contained flag should be zero. Based on Table 2, the logic equations for the toggle bits and the zero contained flag are:

$$\begin{aligned} tx &= fp + le \cdot (syl + \overline{syu} \cdot sxl) + \overline{le} \cdot (syu + sxl \cdot syl) \\ ty &= fp + le \cdot (\overline{sxl} + syl \cdot \overline{sxu}) + \overline{le} \cdot (sxu + sxl \cdot syl) \\ zc &= \overline{fp} \cdot sxl \cdot \overline{sxu} \cdot syl \cdot \overline{syu} \end{aligned}$$

When floating point multiplication is performed, the two bits for the rounding mode come from the floating point status and control register (SCR), as is done on most IEEE compliant processors. These bits are denoted as fp_rm1 and fp_rm0. The design presented here uses the values for the rounding mode bits given in the Sparc Architecture Manual [17] and shown in Table 3. When interval multiplication is performed, the rounding mode from the SCR is altered to cause the multiplier to round toward negative infinity when computing the lower interval endpoint and toward positive infinity when computing the upper interval endpoint. The rounding mode bits, $rm0$ and $rm1$, for the combined interval and floating point multiplier are determined as follows:

$$\begin{aligned} rm0 &= fp\_rm0 \cdot fp + le \cdot \overline{fp} \\ rm1 &= fp\_rm1 + \overline{fp} \end{aligned}$$

Table 2: Setting of sign and toggle bits for interval multiplication.

							$le=1$		$le=0$		
Case	Condition	sxl	sxu	syl	syu	Z	tx	ty	tx	ty	zc
1	$x_l > 0, y_l > 0$	0	0	0	0	$[x_l y_l, x_u y_u]$	1	1	0	0	0
2	$x_l > 0, y_u < 0$	0	0	1	1	$[x_u y_l, x_l y_u]$	0	1	1	0	0
3	$x_u < 0, y_l > 0$	1	1	0	0	$[x_l y_u, x_u y_l]$	1	0	0	1	0
4	$x_u < 0, y_u < 0$	1	1	1	1	$[x_u y_u, x_l y_l]$	0	0	1	1	0
5	$x_l < 0 < x_u, y_l > 0$	1	0	0	0	$[x_l y_u, x_u y_u]$	1	0	0	0	0
6	$x_l < 0 < x_u, y_u < 0$	1	0	1	1	$[x_u y_l, x_l y_l]$	0	1	1	1	0
7	$x_l > 0, y_l < 0 < y_u$	0	0	1	0	$[x_u y_l, x_u y_u]$	0	1	0	0	0
8	$x_u < 0, y_l < 0 < y_u$	1	1	1	0	$[x_l y_u, x_l y_l]$	1	0	1	1	0
9	$x_l < 0 < x_u, y_l < 0 < y_u$	1	0	1	0	$[mn, mx]$	X	X	X	X	1

Table 3: Value of the rounding mode bits.

Rounding mode	fp_rm1	fp_rm0
Round to nearest even	0	0
Round toward 0	0	1
Round toward $+\infty$	1	0
Round toward $-\infty$	1	1

4 Area and delay estimates

Area and delay estimates are given in Table 4 for the combined interval and floating point multiplier, along with estimates for an IEEE double precision multiplier and the serial interval multiplier presented in [13]. The area and delay estimates are based on data from a 0.4 micron CMOS standard cell library [18]. These estimates assume that the significand multiplier is a Reduced Area Multiplier [19], followed by a carry select adder with rounding and normalization logic [20]. The area of each component is estimated by adding together the areas of all of the macrocells that make up the component and then adding an additional 50 percent for internal wiring. The total area is estimated as the sum of the component areas plus an additional 50 percent for global routing and unused space. The delay for each component is computed by taking the worst case delay of the critical path and adding 25 percent for process variations and clock skew.

Compared to the IEEE double precision multiplier, the combined interval and floating point multiplier requires three additional 64-bit registers, and the multiplexors and control logic for selecting the interval endpoints. It uses about seven percent more area and has a total delay that is eight percent longer. The serial interval multiplier requires a dedicated min/max unit, and it cannot perform floating point multiplication. Compared to the IEEE double precision multiplier, it requires fifteen percent more area and has a total delay that is about eight percent longer. Although the min/max unit of the serial interval multiplier causes the total area to increase by about 8 percent, it does not increase the cycle time, since pipeline registers are used to keep it off the critical path.

If the combined interval and floating point multiplier is not pipelined, it can perform floating point and interval multiplication in one and two cycles, respectively, with a cycle time of approximately 13 ns. To decrease the cycle time, the interval multiplier can be pipelined into two stages. The first stage performs interval endpoint selection, partial product generation, and partial product reduction. The second stages combines the carry and sum vectors from the reduced partial products, and normalizes and rounds the result. With this approach, floating point and interval multiplication require two and three cycles, respectively, and the cycle time is reduced to approximately 7 ns. For the pipelined implementation, the total area increases to 10.88 mm^2 due to additional registers needed for pipelining. Depending on the area and performance requirements, other design modifications can be made.

5 Conclusion

The combined interval and floating point multiplier is implemented by making minor modification to a conventional floating point multiplier. With this approach, interval multiplication requires only one more cycle than floating point multiplication, and is one to two orders of magnitude faster than software implementations of interval multiplication. By trapping to software in the uncommon case that both endpoints contain zero, a significant decrease in area is achieved. The design of the combined multiplier has been completed, and its VLSI implementation is currently in progress.

Table 4: Multiplier area and delay estimates.

	IEEE Double		Combined Interval		Serial Interval [13]	
Component	Area (mm^2)	Delay (ns)	Area (mm^2)	Delay (ns)	Area (mm^2)	Delay (ns)
Multiplier	6.05	11.53	6.05	11.53	6.05	11.53
Registers	0.35	0.51	0.70	0.51	0.70	0.51
Multiplexors	–	–	0.08	0.44	0.08	0.44
Control Logic	–	–	0.01	0.50	0.01	0.50
Min/Max Unit	–	–	–	–	0.53	6.27
Route and Space	3.20	–	3.42	–	3.69	–
Total	9.60	12.04	10.26	12.98	11.06	12.98
Percent Overhead	–	–	6.88	7.81	15.21	7.81

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. MIP-9703421.

References

[1] A. Yu, "The Future of Microprocessors," *IEEE Micro*, vol. 16, no. 6, pp. 46–53, 1996.

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

[3] R. E. Moore, *Interval Analysis*. Prentice Hall, 1966.

[4] G. F. Corliss, "Industrial Applications of Interval Techniques," in *Computer Arithmetic and Self-Validating Numerical Methods*, (C. Ullrich, ed.), pp. 91–113, Academic Press, 1990.

[5] O. Knuppel, "PROFIL/BIAS - A Fast Interval Library," *Computing*, vol. 53, pp. 277–288, 1994.

[6] R. B. Kearfott and M. Novoa, "INTBIS, A Portable Interval Newton Bisection Package," *ACM Transactions on Mathematical Software*, vol. 16, pp. 152–157, 1990.

[7] R. B. Kearfott *et al.*, "A Specific Proposal for Interval Arithmetic in FORTRAN," Manuscript, University of Southwestern Louisiana, 1996.

[8] M. J. Schulte, V. Zelov, A. Akkas, and J. C. Burley, "Adding Interval Support to the GNU Fortran Compiler," Manuscript, Lehigh University, 1997.

[9] D. Chiriaev and G. W. Walster, "Interval Arithmetic Specification," Manuscript, 1997.

[10] M. J. Schulte and E. E. Swartzlander, Jr., "Variable Precision, Interval Arithmetic Coprocessors," *Reliable Computing*, vol. 2, pp. 47-62, 1996.

[11] C. Baumhof, "A New VLSI Vector Arithmetic Coprocessor for the PC," *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 210–215, 1995.

[12] J. Wolff von Gudenberg, "Hardware Support for Interval Arithmetic", in *Scientific Computing and Validated Numerics*, (G. Alefeld, A. Frommer, and B. Lang, eds.), pp. 32–37. Akademie Verlag, 1996.

[13] M. J. Schulte, K. C. Bickerstaff, and E. E. Swartzlander, Jr., "Hardware Interval Multipliers," *Journal of Theoretical and Applied Informatics*, vol. 3, no. 2, pp. 73–90, 1996.

[14] A. Knofel, "Hardware Kernel for Scientific/Engineering Computations," *Scientific Computing with Automatic Result Verification*, (E. Adams and U. Kulisch, eds.), pp. 549–570, Academic Press, Inc., 1993.

[15] *IEEE Standard 754 for Binary Floating Point Arithmetic*. IEEE, 1985.

[16] G. W. Walster, "The Extended Real Interval System," Manuscript, 1997.

[17] D. L. Weaver and T. Germond, *The Sparc Architecture Manual*. Sparc International, Inc., 1992.

[18] *G10-p Cell-Based ASIC Products Databook*. LSI Logic Corporation, 1997.

[19] K. C. Bickerstaff, M. J. Schulte, and E. E. Swartzlander, Jr., "Parallel Reduced Area Multipliers," *Journal of VLSI Signal Processing*, vol. 9, pp. 181–192, April 1995.

[20] M. R. Santoro, G. Bewick, and M. A. Horowitz, "Rounding Algorithms for IEEE Multipliers," in *Proceedings of the 9th Symposium on Computer Arithmetic*, pp. 176–183, 1989.