Cheap Extension of Floating-Point Units for Interval Arithmetic

Jürgen Wolff von Gudenberg
Universität Würzburg
email: jwvg@acm.org

Abstract

In the paper we show how a double-precision floating-point unit can be extended to support interval arithmetic. Our current design starts from 64 bit double-precision format. We assume that memory and buses use this format. We then represent an interval as 2 single-precision numbers that fit into the same format. The unit is split into two tightly coupled single-precision arithmetic units in order to perform one interval, one doubled precision or two independent floating-point operations.

The design which is similar to the recently introduced multi-media architectures is applicable for any two related precisions. It can further be extended to support double-precision interval arithmetic. Then, 2 consecutive double-precision numbers representing the bounds are piped to our unit.

For the IEEE single and double formats detailed simulations have shown that this can be done with little overhead for addition and with moderate overhead for multiplication.

After a short introduction to interval arithmetic we specify our algorithms for interval addition/subtraction and multiplication. For interval addition we consider a fast carry look ahead adder which can be split into two adders by cutting the root of the tree and inserting two new appropriate carry-in signals. A tree-like comparator is divided into two parts by ignoring the final stage and delivering two independent output signals. Furthermore the shifter unit can also be separated into two units. Here we assume a fast shifter with $\log(n)$ stages. For each stage we require some new multiplexers. The logic to calculate the sticky bit has to be doubled. The integer unit performing the calculation of the exponent has to be extended, in our case for the IEEE format from 11 to 16 bit. This unit, then, is split into two 8 bit units.

For interval multiplication we implement a version of the 4-product method [11] which computes the 4 different double length products of the interval bounds, and then rounds the minimum and maximum. We have to extend the multiplier pipeline by two additional stages. Another option is to implement the 8-products method [11].

Interval addition can thus be carried out with the same latency as and interval multiplication uses 1 through 2 cycles (or steps) more than the corresponding floating-point operation.

1. Interval Arithmetic

Interval Arithmetic is defined as an arithmetic for compact, continuous sub-

sets $\mathbf{a}, \mathbf{b}$ of the real numbers. Let $\circ \in \{+, -, \cdot, /\}$ then

$$\mathbf{a} \circ \mathbf{b} = \{a \circ b | a \in \mathbf{a}, b \in \mathbf{b}\} \tag{1}$$

defines the corresponding interval operation. If the bounds are computer representable, the represented set still is continuous. The theoretical definition (1) can be implemented with efficient formulae depending only on the endpoints of the intervals. To ensure that the exact result lies within the resulting interval for every interval operation directed rounding towards $\pm\infty$ has to be applied. This outward rounding, indeed, has the consequence that interval arithmetic may be used to control rounding errors, because automatically verified bounds for the true solution or solution set are delivered. For a detailed treatment of machine interval arithmetic we refer the reader to [5]. We denote downwardly or upwardly directed rounded operations by enclosing the operator into ∇ or $\triangle$, respectively.

1.1. Addition/Subtraction

Interval addition/subtraction is very simple to perform by adding/subtracting the interval bounds and rounding the results appropriately

$$[a, b] + [c, d] = [a \mathbin{\nabla} c, b \mathbin{\triangle} d]$$

$$[a, b] - [c, d] = [a \mathbin{\nabla} d, b \mathbin{\triangle} c]$$

No extra conditionals are needed, and there are no specific exceptions.

1.2. Multiplication

$$[a, b] * [c, d] = [min(a \mathbin{\nabla} c, a \mathbin{\nabla} d, b \mathbin{\nabla} c, b \mathbin{\nabla} d), max(a \mathbin{\triangle} c, a \mathbin{\triangle} d, b \mathbin{\triangle} c, b \mathbin{\triangle} d)] \tag{2}$$

In [11, 7] several alternatives to support interval multiplication by hardware are discussed, some of those are considered in this paper:

The straight forward implementation of (2) is a simple but interesting alternative using nowadays processors, although 8 products and 6 comparisons have to be computed, see [3]. Another algorithm starts with the computation of the 4 products of the bounds in double-length format, i.e. doubled precision, determines minimum and maximum, and then rounds. This yields a total number of 4 multiplications, 4 comparisons and 2 roundings. Most frequently used in software implementations is the following approach, based on a case selection to save as many multiplications as possible. Table 1 displays the cases depending on the signs of the bounds of the input intervals.

	$A = [a, b]$	$B = [c, d]$	$A * B$
1	$A \geq [0,0]$	$B \geq [0,0]$	$[a \bigtriangledown c, b \bigtriangleup d]$
2	$A \geq [0,0]$	$B < [0,0]$	$[b \bigtriangledown c, a \bigtriangleup d]$
3	$A \geq [0,0]$	$0 \epsilon B$	$[b \bigtriangledown c, b \bigtriangleup d]$
4	$A < [0,0]$	$B \geq [0,0]$	$[a \bigtriangledown d, b \bigtriangleup c]$
5	$A < [0,0]$	$B < [0,0]$	$[b \bigtriangledown d, a \bigtriangleup c]$
6	$A < [0,0]$	$0 \epsilon B$	$[a \bigtriangledown d, a \bigtriangleup c]$
7	$0 \epsilon A$	$B \geq [0,0]$	$[a \bigtriangledown d, b \bigtriangleup d]$
8	$0 \epsilon A$	$B < [0,0]$	$[b \bigtriangledown c, a \bigtriangleup c]$
9	$0 \epsilon A$	$0 \epsilon B$	$[min(a \bigtriangledown d, b \bigtriangledown c), max(a \bigtriangleup c, b \bigtriangleup d)]$

Table 1: Interval Multiplication

The worst case is (9). In each other case only one multiplication for each bound is needed, but in (9) two directed products and one comparison per bound must be performed.

1.3. Software Implementations

Intervals need twice as much storage space as floating-point numbers. The operation times are theoretically twice as long for addition/subtraction, and there is a factor of 2.3 for multiplication. These factors, however, cannot be obtained in practice. Since intervals are no standard data types many optimizations cannot be performed. We measured a factor of 4 for addition/subtraction and a factor of 10 for multiplication in the fastest implementation of interval arithmetic known to us [6].

1.4. Applications

Applications of interval arithmetic reach beyond the range of safety critical problems where verified results are mandatory. In global optimization especially the discarding of large sections is supported. In computer graphics, rendering images, ray-tracing, e.g., interval algorithms have been proposed that are superior to the established floating-point versions in terms of efficiency and, of course, reliability [9]. Here often single precision suffices.

2. Architecture Extension

We propose a simple and cheap extension which preserves the existing bus band-widths and control mechanisms as far as possible.

Our main idea is to represent the 2 single precision endpoints of an interval in a double precision word and simultaneously to compute the 2 bounds in

parallel by an extended double precision FPU.

3. Interval adders

In this section, we will point out, how to extend a given double precision fpadder to perform single precision interval addition. For the sake of simplicity, we suppose the following simple interface of the adder: The 2 operands are input to the unit using two 64bit input buses, the output is delivered by a 64 bit result bus. Usually, a double precision adder is pipelined, and this can be done in many different ways. In order to illustrate, that such adders can be easily extended, we consider the following example of a floating-point addition pipeline:

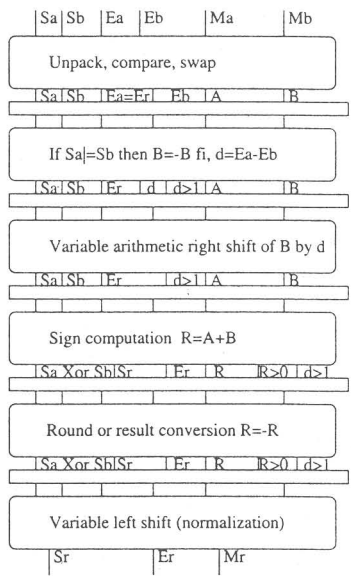

Figure 1: A Floating-Point Addition Pipeline of Latency 6

This unit is split to obtain an interval addition pipeline of the same latency 6. The basic idea is to extend all components of the exponent unit to 16 bit and then to split all integer and floating-point components of the unit:

1. We can cut off the last stage of a tree-like comparator by one multiplexer, thus delivering 2 results.

2. A converter that consists out of independent inverters and an incrementor can be split analogously to an adder.

3. Each stage of a variable length shifter is cut in the middle by $d+1$ multiplexers, if the stage performs a d-bit shift. A second sticky bit logic has to be supplied. Nevertheless this sums up to only 12% of the total cost of the shifter.

4. One single multiplexer switches the last stage of our carry-look-ahead adder in order to get two independent results.

5. Two independent rounding units have to be supplied.

A more detailed specification of the units is contained in [10, 3, 4].

4. Interval Multipliers

We suppose, that 2 operands are input to the double precision FPU using two 64bit input buses. In order to make the discussion more clear, we suppose, that the double precision multiplier pipeline has the following stages:

1. split mantissae and compute 4 partial products.

2. add partial products in carry-save encoding.

3. final carry-look-ahead addition of the product.

4. normalize and add exponents.

5. round and adjust exponent.

6. normalize again.

Mantissa multiplication in the first 3 stages can be split for single precision interval multiplication relatively easy:

1. compute 4 separate products of interval bounds.

2. circumvent the last stage of the carry-save addition.

3. split the final fast carry-look ahead adder into 2 parts and insert two more fast adders.

4. normalize 4 results and add exponents.

5. compute minimum and maximum.

6. round 2 bounds and adjust exponents.

7. normalize 2 bounds again.

For details, again, see [3, 4].

References

[1] G. Alefeld, J. Herzberger: *An Introduction to Interval Computations*, Academic Press, New York, 1983

[2] T. Duff: *Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry*, Computer Graphics, 26,2, 1992, pp 131-138

[3] R. Kolla, A. Vodopivec, J. Wolff v. Gudenberg: *The IAX Architecture - Interval Arithmetic Extension*, Universität Würzburg, Institut für Informatik, Techn. Report TR225, April 1999

[4] R. Kolla, A. Vodopivec, J. Wolff v. Gudenberg: *Splitting Double Precision FPUs for Single Precision Interval Arithmetic*, Proceedings of the Workshop (Mikroprozessoren zur Jahrtausendwende: Gestaltungsalternativen zukünftiger Prozessorarchitekturen), 15. GI/ITG-Fachtagung ARCS '99 Architektur von Rechensystemen 1999 - 1999, to appear

[5] U. Kulisch, W. L. Miranker: *The Arithmetic of the Digital Computer: A New Approach.* SIAM Review, Vol. 28, No. 1, March 1986

[6] M. Lerch: *fi_lib++*, Universität Würzburg, Institut für Informatik, Techn. Report, to appear

[7] M. Schulte et al: *Hardware Units for Interval Multiplication*, ANAIS Workshop, Recife Brasil, p. 85-87, 1996

[8] J. Snyder: *Interval Analysis for Computer Graphics*, Computer Graphics, 26,2, 1992, pp 121-130

[9] N. Stolte, R. Caubet: *Comparison between Different Rasterization Methods for Implicit Surfaces*, in Rae Earnshaw, John A. Vince and How Jones (eds), Visualization and Modeling, Chapter 10, pp 191-201, Academic Press, 1997.

[10] A. Vodopivec: *Entwurf einer Hardwareeinheit für Intervallarithmetik*, Universität Würzburg, Institut für Informatik, Diplomarbeit, 1998

[11] J Wolff v Gudenberg: *Hardware Support for Interval Arithmetic*, in Alefeld/ Frommer/ Lang (eds.) Scientific Computing and Validated Numerics, Akademie Verlag, p. 32-37, 1996