Hardware support for UNUM floating point arithmetic

Andrea Bocco
CEA-LETI
Grenoble, France
Email: andrea.bocco@cea.fr

Yves Durand
CEA-LETI
Grenoble, France
Email: yves.durand@cea.fr

Florent de Dinechin
INSA-Lyon
Lyon, France
Email: Florent.de-Dinechin@insa-lyon.fr

***Abstract*—UNUM is a variable length floating-point format conceived to substitute the current one defined in the IEEE 754 standard. UNUM is able, through an internal algebra based on interval arithmetic, to keep track of the precision during operations, offering better result reliability than IEEE 754.**

This work discusses the implementation of UNUM arithmetic and reports hardware implementation results of some of the UNUM operators.

1. Introduction

The floating-point (FP) format [1] is widely used by the scientific community to represent real numbers, and has been standardized in the IEEE 754 standard [2]. However, FP representation intrinsically introduces several problems, such as: accuracy in iterative algorithms (due to cancellation and accumulation errors), memory footprint and energy consumption (as even low-precision applications require long IEEE 754 FP numbers).

The Universal NUMber, or UNUM, is an alternative representation format introduced by John L. Gustafson [3]. It is a variable-length FP format with an interval-based semantics. These two features could improve both accuracy and memory footprint of application code.

However, as far as we know, the only realizations of UNUM arithmetic are done in software, using dynamically allocated data structures [3]. It is not clear if an hardware implementation of UNUM can achieve the claimed potential. The aim of this work is therefore to study such an hardware implementation of UNUM, and in particular assess its cost compared to a conventional IEEE 754 FPU.

After a presentation of the UNUM format and algebra in Sections 2, implementation choices left open in [3] are discussed in Section 3. The proposed implementation is described in Section 4 and evaluated and compared to classical FP in Section 5.

2. The UNUM format and algebra

As Figure 1 shows, a UNUM number is composed of several fields. Compared to standard floating points, it has two distinctive features:

First, it is a *variable-size, self-descriptive format*: the *es-1* and *fs-1* fields contain the bit lengths of the e (exponent) and f (fraction) fields, minus one. The size of these two fields is itself defined by the *UNUM environment* [3] (UE). In the present work, for instance, we use a (4,6) UE: 4 bits for `es-1` and 6 bits for `fs-1`. The largest UNUM numbers in this UE have 16 bits of exponent and 64 bits of fraction (more than double-precision numbers). The smallest ones have 1 bit of fraction and 1 bit of exponent, but are still encoded on 14 bits because of the other fields.

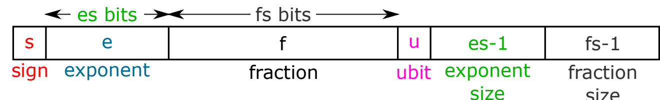

Figure 1. The UNUM format

The second distinctive feature is the u field (or ubit, for "uncertainty" bit). When 0, the UNUM represents a *scalar number*, whose value encoded into the three fields (s, e, f). When set, it represents an *open interval* defined between the scalar value obtained when the ubit is 0, and the one with the fraction field incremented by one Unit in the Last Position (ULP). A u-bit of 1 can be interpreted as "the fraction begins with the provided bits, but there are more and they are unknown".

Finally, to obtain a number representation closed under the basic operations, a real number can also be represented as an interval defined as a tuple of two UNUMs. Such an interval is called a *u-bound* [3]. It can be open or closed at each endpoint, depending on the corresponding ubit. The set of u-bounds is closed under the basic operations. During such operations, the resulting interval endpoints are rounded toward $\pm\infty$, to ensure the property that the real result is included in the resulting interval in spite of rounding error.

The UNUM algebra [3] is based upon three different layers. Well-defined conversion functions are used to move numbers among layers.

The *u-layer* is where the UNUM formats are defined, corresponds to the FP layer in existing systems.

There is one layer above it, the *h-layer*, where the data is reworded for humans. We will not address it in this work.

There is also a lower layer, the *g-layer*. This is the underground layer where the operations are actually carried out (also called the scratchpad layer). Contrary to FP, it is not necessary identical to the u-layer. For instance, it can

Figure 2. The g-number format

use more precision. Like in this work, numbers may have a fixed size in the g-layer, to match the fact that hardware is fixed. Finally, *fused operations* can be embedded inside the g-layer to compute complex functions with high accuracy [3].

Again for closure, the g-layer actually handles *g-bounds* [3] (the g-layer version of u-bounds). What we will call g-numbers in this work are g-bound endpoints.

The longer-term objective of this work is to propose a parametric hardware+software implementation of a complete UNUM system that addresses the dark spots of [3]. In particular, the promise of automatically improved accuracy will be confronted to the interval size explosion that plagues interval arithmetic. We are interested in studying how UNUM arithmetic can help deploying known (non-automatic) techniques to contain interval size explosion.

Closer to this work, the promise of lower consumption has to be challenged against the increased consumption of units themselves, the need to transfer intervals instead of scalar numbers, the additional complexity of managing variable-size data in the memory subsystem, etc.

3. UNUM hardware implementation choices

As Figures 3 and 4 show, the u-layer and g-layer operators are composed of different sub-units, interfaced each other using u-bound and g-bound formats.

A first design choice is to expand variable-size UNUMs into a hardware-friendly fixed-size *internal UNUM* format which retains the *es-1* and *fs-1* fields, but pads both exponent and mantissa with zeroes to the maximum size allowed by the UE. The conversions between variable-size UNUMs and this internal format involve shifts (computed out of the es-1 and fs-1 fields), but no rounding or special case management.

A second design choice is the g-number format adopted for the g-bound endpoints. Depicted on Figure 2, it is a large classical FP format, with two additions: 1/ five *summary bits* encode exceptional cases, in order to avoid area and latency of decoding them from the exponent field. 2/ an endpoint termination flag is true if the interval endpoint is closed, false if it is open. Like for internal UNUMs, the size of this g-number format is chosen large enough to handle the full set of all possible UNUMs for the chosen UE. The leading bit of the mantissa, which is implicit in IEEE754 formats, is here explicit. A signed exponent (instead of the usual biased one [2]) simplifies the conversion functions algorithm.

Currently, our internal g-numbers can be subnormals, in which case the leading mantissa bit 0. This conservative choice will be discussed in the conclusion.

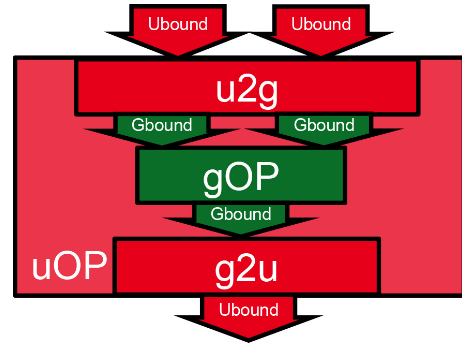

Figure 3. The u-layer operator general architecture

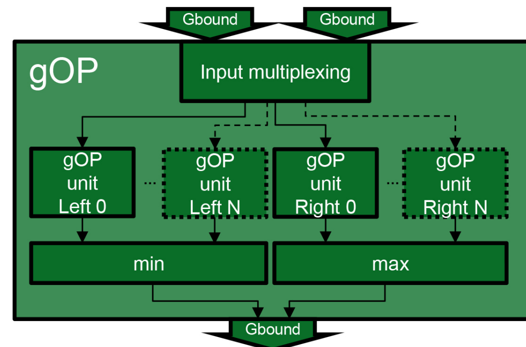

Figure 4. The g-layer operator general architecture

4. UNUM units architecture

4.1. Overview

Four UNUM units have been implemented: an u-bound adder *u_add*, an u-bound comparator *u_cmp*, an u-bound multiplier *u_mul*, and an u-bound divider *u_div*.

They all share the same macro architecture depicted on Figure 3. The input u-bounds are converted into g-bounds thanks to the *U2G* sub-unit. Then the obtained g-bounds are input to the g-bound operators (gOp) detailed below. These output a g-bound, which is converted back in u-bound through the *G2U* sub-unit. The only exception is the u_cmp unit, which doesn't need a G2U unit since it directly outputs the comparison result flags computed by the g_cmp sub-unit.

The conversion functions are identical among all the u-bound operators, only the gOP sub-unit (*g_add*, *g_cmp*, *g_mul*, and *g_div*) changes among them.

With the exception of g_cmp, each gOP unit has the same interval processing [4] macro architecture depicted on Figure 4. The endpoints of the two input g-bounds are extracted. The relevant operations are performed on these endpoints, rounded to the outside of the interval. The maximum and minimum values are then selected as the endpoints of the output g-bound.

Finally, all the endpoint computations in gOP units follow a classical floating-point scheme [1]: the input mantissa are aligned to a common format if needed, the operation (+,-,*,/) is computed on the two aligned operands, the result

is normalized and rounded, and the input exceptions [3] are handled.

In the present work the designed units are not pipelined, this will be the object of future work.

The next subsections describe the g-bound operators algorithms. The symbols used are: $G1$ and $G2$ for the two input g-bounds that the gOP units receive, Z for the output g-bound. For a g-bound A, we note $\underline{A}$ and $\overline{A}$ its lower and upper endpoints. For a real x we also note $\bigtriangledown(x)$ (resp. $\bigtriangleup(x)$) the rounding of x to the g-format in the to $-\infty$ (respto $+\infty$) direction.

4.2. g-bound adder

The g-bound adder has to compute the sum of two g-bound provided in input.

$$Z = \begin{cases} \underline{Z} = \bigtriangledown(\underline{G1} + \underline{G2}) \\ \overline{Z} = \bigtriangleup(\overline{G1} + \overline{G2}) \end{cases} \quad (1)$$

Here neither the input multiplexing unit, nor the minimum and maximum endpoint selection units are needed: the g-bound input endpoints are connected directly to the two endpoint computation units, and their output correspond to the final interval endpoints.

Inside each endpoint computation units, the final endpoint is computed. The algorithm chosen to sum the two g-numbers is the one shown in [1]: the two input mantissa are aligned to a common exponent value; the aligned mantissa are summed (or subtracted, depending on the input signs) together; The final result is normalized and rounded; The input exception are handled.

4.3. g-bound comparator

On intervals, the comparison $A < B$ may be true (if $\overline{A} < \underline{B}$), false (if $\underline{A} \geq \overline{B}$), or "I can't say" (in the other cases). As a side note, for UNUM to supplant FP, all the programming languages have to be modified to support this third option... But this is out of the scope of this paper.

To allow for that, UNUM comparison unit has to compute the six following comparison flags [3]: Lower Than (LT), Greater Than (GT), EQual (EQ), Not Nowhere EQual (NNEQ), Nowhere EQual (NEQ). In order to do that, it compares each g-bound input endpoint with the two endpoints of the other input. Depending how the two input g-bounds intersect (or not) each others, the four output flags are set or unset, also taking care of the exception values.

The hardware implementation of this function therefore consists of six greater-than comparators and six equal comparators. Their 8 results drive a combinational logic that generates the 6 output flags.

4.4. g-bound multiplier

The g-bound multiplier has to compute the multiplication of two g-bound provided in input.

$$\begin{cases} \underline{Z} = \min\left(\bigtriangledown(\underline{G1}\,\underline{G2}), \bigtriangledown(\underline{G1}\,\overline{G2}), \bigtriangledown(\overline{G1}\,\underline{G2}), \bigtriangledown(\overline{G1}\,\overline{G2})\right) \\ \overline{Z} = \max\left(\bigtriangleup(\underline{G1}\,\underline{G2}), \bigtriangleup(\underline{G1}\,\overline{G2}), \bigtriangleup(\overline{G1}\,\underline{G2}), \bigtriangleup(\overline{G1}\,\overline{G2})\right) \end{cases} \quad (2)$$

Fortunately, looking only at the signs allows to eliminate two of the 4 inputs to the max and min operations. As a consequence, with some multiplexing of the inputs endpoints, it is enough to instantiate four endpoint multiplication units, one two-input max, and one two-input min.

In addition, in 15 cases out of 16 input endpoint sign combinations, only one multiplication per endpoint is needed. The proposed implementation exploits this to reduce the average power consumption.

Again, the multiplication algorithm itself is a classical FP one [1]. In this case, there is no alignment of the mantissas, the two operand are directly multiplied each other. After that, the intermediate result is normalized and rounded.

4.5. g-bound divider

The g-bound divider has to compute the division of two g-bound provided in input.

$$\begin{cases} \underline{Z} = \min\left(\bigtriangledown(\underline{G1}/\underline{G2}), \bigtriangledown(\underline{G1}/\overline{G2}), \bigtriangledown(\overline{G1}/\underline{G2}), \bigtriangledown(\overline{G1}/\overline{G2})\right) \\ \overline{Z} = \max\left(\bigtriangleup(\underline{G1}/\underline{G2}), \bigtriangleup(\underline{G1}/\overline{G2}), \bigtriangleup(\overline{G1}/\underline{G2}), \bigtriangleup(\overline{G1}/\overline{G2})\right) \end{cases} \quad (3)$$

However, it is possible to demonstrate here that, looking at the sign of the input g-bound endpoints, at most one endpoint is a good candidate for each output endpoint. Thanks to that, multiplexing correctly the input g-bound endpoints to the endpoint computation units, it is possible to remove the minimum and maximum sub-units and output the endpoints directly.

The design was validated using a behavioral mantissa division (/ in VHDL). Unfortunately it is unsuitable for synthesis in this context. We now have to replace it with a synthesizeable one. Therefore there will be no synthesis result for division in this submission.

5. Synthesis results and comparison between UNUM and IEEE 754

The designed units were described in VHDL. They were validated using Mentor Questasim against a reference python library [5] on $2 \cdot 10^6$ pseudo-random input vectors. Then they were synthesized using Synopsys Design Compiler with the CORE65LPSVT library.

As reference for the designed UNUM units, a 64bit FPU compliant with the IEEE 754 standard is taken from OpenCores [6]. The UE chosen is the $(ess = 4, fss = 6)$: it is the smallest UE that includes all double-precision IEEE 754 FP numbers. Its g-layer operators have 64-bit

TABLE 1. SYNTHESIS RESULTS OF UNUM AND IEEE 754 UNITS.

Unit	Timing (ns)	Area (cells)	Power (mW)
u_add	16.43	39672	1.99
U2G in add	3.20	14599	0.44
g_add	9.08	16226	1.02
		= 2×8113	= 2× 0.51
64-bit FP add	*8.23*	*7183*	*1.75*
G2U in add	4.15	8926	0.53
u_mul	17.42	96153	8.61
U2G in mul	2.88	12115	0.47
g_mul	10.40	77307	7.31
		= 4×19327	= 4×1.83
64-bit FP mul	*8.51*	*17635*	*4.53*
G2U in mul	4.24	6731	0.82
u_cmp	4.30	34067	7.43
U2G in cmp	2.20	13320	2.16
g_cmp	2.10	20747	5.26

significands, and are therefore slightly more accurate than the double-precision ones (53-bit significands).

Table 1 shows the synthesis results of the proposed UNUM units, decomposed in their sub-units as per Section 4.

For operators replicated inside the g-layer (g_add and g_mul), we also factor out the replication. This enables comparison of g_add and g_mul with the corresponding IEEE-754 operators. Note however that the reported g_add and g_mul also involve input multiplexers and the minimum and maximum units.

In this table we have slightly different results for the U2G and G2U units, although these units are identical in each operator. Obviously the synthesizer optimizes them slightly differently in each context.

The lessons from this experiment are the following. Our g_add and g_mul are comparable to the IEEE ones (slightly slower and larger as expected, since their fraction size is 64 instead of 53). The real cost here is the x2 and x4 area overheads due to interval arithmetic. The observed power overhead is limited to a factor 2 in the multiplier, as explained in Section 4.4. The lower power consumption of the g_add unit is due to its simpler subnormal handling.

The conversion functions inside each u-layer operator almost double its latency. If we are to integrate an UNUM unit in a processor, this is not acceptable. It pleads for exposing the g-layer to the instruction set (with a g-layer register file and instructions that operate in the g-layer), and fusing operations in the g-layer as much as possible (as a compiler optimization). This would dilute the overhead of the conversion functions.

But then, we also need explicit conversions G2U instructions to the UNUM format. Program optimization would attempt to use as few as possible of such instructions, typically only when spilling to external memory. Currently, such conversions compute the smallest matching UNUM format. For iterated computations, the fraction size tends to grow and quickly saturates to its maximum. This will negate the potential power saving due to transmitting UNUM numbers stored on a few bits only. To address this problem, Gustafson suggests starting with a small UE and growing it if needed [3]. A more practical alternative in a hardware context would be that the G2U conversion instructions take as argument the number of significand bits to which to round. The issue, of course, is more burden on the programmer. This idea has to be evaluated on actual applications.

Let us now come back to one design choice: subnormal handling in the g-layer. One valid alternative would be to have only normal numbers in the g-layer. To ensure that all UNUM numbers are representable, we would need one more exponent bit than the maximum allowed by the UE, but the corresponding hardware overhead would be negligible. This would simplify and speed-up the g-layer operations, especially the multiplication. On the other hand it would make the conversions more complex since they would have to handle subnormalisation. However, this is consistent with the idea of a g-layer instruction set with as few as possible G2U instructions.

6. Conclusion

The new UNUM format is presented with two potential benefits. The first is safer numerical computing thanks to its well defined interval semantics. The second is power reduction in the data transfers thanks to a self-descriptive, variable-size format. However, these two benefits come at the cost of more expensive and more power-hungry computing unit. This work presents a first quantitative assessement of this trade-off. It also discusses some of the design choices that must be made.

The next step is to complement the proposed UNUM unit with a matching register file, then a full processor. This requires some deeper rethinking, in particular in the control flow management instructions: they must have three exits and not only two as in usual processors. The programming environment must also expose finer precision control than in classical systems.

Finally, the promises of the UNUM format remain to be demonstrated on full-scale applications. This is the long term objective of this research.

References

[1] J.-M. Muller et al., *Handbook of Floating-Point Arithmetic*, DOI 10.1007/978-0-8176-4705-6, Birkhäuser, 2010.

[2] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985, 1985.

[3] John L. Gustafson, “The End of Error - Unum Computing”, CRC Press, 2015.

[4] M. J. Schulte and E. E. Swartzlander, “A family of variable-precision interval arithmetic processors,” in *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 387-397, May 2000.

[5] Github project of the UNUM python library https://github.com/jrmuizel/pyunum.

[6] FPU 64 bit IEEE 754 compliant, Opencores, FPU 64 bit http://opencores.org/project,fpu_double