Hardware support for UNUM floating point arithmetic

Andrea Bocco, Yves Durand, Florent de Dinechin

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*—The Universal NUMber, or 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 IEEE 754 standard [1] for floating-point (FP) numbers [2] is widely used by the scientific community. 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).

UNUM is an alternative representation format introduced by John L. Gustafson [3]. It is a variable-length FP format with an interval-based semantics. It is originally meant for scientific computing, where its improved accuracy and interval semantics can be very precious, or for embedded application which may benefit from reduced memory footprint.

However, as far as we know, the only realizations of UNUM arithmetic have been implemented in software [3]. It is not clear if an hardware implementation of UNUM can achieve the claimed potential. The aim of this work is therefore to investigate 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:

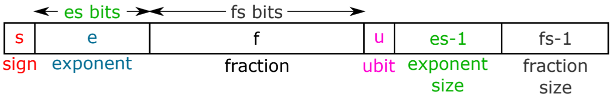

Figure 1. The UNUM format

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). For example, when using a (4,6) UE: 4 bits for *es-1* and 6 bits for *fs-1*, the largest UNUM numbers can count up to 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.

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, 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 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] e.g. 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 this interval size explosion.

Closer to this work, the promise of lower consumption has to be challenged against 1/ the increased power budget

978-1-5090-6508-0/17/$31.00 ©2017 IEEE

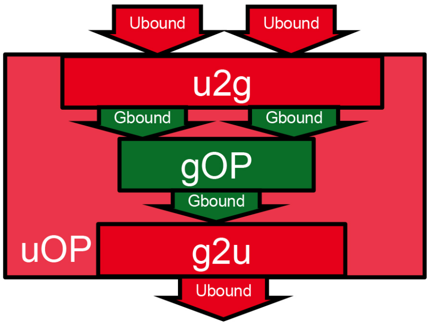

Figure 2. The u-layer operator general architecture

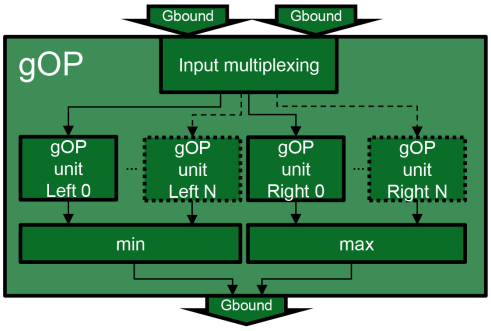

Figure 3. The g-layer operator general architecture

of the units themselves, 2/ the need to transfer intervals instead of scalar numbers, 3/ the additional complexity of managing variable-size data in the memory subsystem.

3. UNUM hardware implementation choices

Gustafson organizes the computation of UNUMs into three different layers [3]. Well-defined conversion functions are used to move numbers among layers. The *h-layer*, deals with the data format used for human interaction. We will not address it in this work. The *u-layer* is where the UNUM formats are defined, corresponds to the FP layer in existing systems. The *g-layer* is the underground layer where the operations are actually carried out (also called the scratchpad layer). Its format is not necessary identical to the u-layer i.e. numbers may be represented with 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]. Since the UNUM actually implements an algebra of intervals, 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.

As Figures 2 and 3 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.

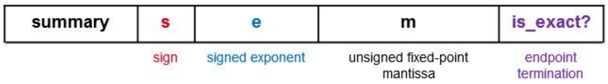

Figure 4. The g-number format

A second design choice is the g-number format adopted for the g-bound endpoints. Our solution is depicted on Figure 4. 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, 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 [1]) 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.

4. UNUM units architecture

4.1. Overview

Four u-bound operators (uOP) 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 (except u_cmp) share the same macro architecture depicted on Figure 2. The *U2G* sub-unit converts the input u-bounds into g-bounds. The obtained g-bounds are input to the g-bound operators (gOP) detailed below. The *G2U* sub-unit converts the obtained g-bound back in u-bound (not needed in the u_cmp since it directly outputs the comparison result flags computed by the g_cmp sub-unit).

The uOP differ only by the gOP sub-unit (*g_add*, *g_cmp*, *g_mul*, *g_div*). They share the same conversion functions.

Each gOP unit (except g_cmp) has the same interval processing [4] macro architecture depicted on Figure 3. The endpoints of the two input g-bounds are used to perform the relevant endpoint operations, rounded to the outside of the interval. The maximum and minimum values are then selected as the endpoints of the output g-bound.

The endpoint computations in gOP units follow a classical floating-point scheme [2]: the input mantissas are aligned to a common format if needed, the operation (+,-,*,/) is computed on the aligned operands, the result is normalized and rounded, and the input exceptions [3] are handled.

The next subsections describe the g-bound operators algorithms. The symbols used are: for a gOP unit, $G1$ and

$G2$ for the two input g-bounds, and Z for the output g-bound. For a g-bound A, we note $\underline{A}$ and $\overline{A}$ as 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 $-\infty$ (resp. $+\infty$) direction.

4.2. g-bound adder

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

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

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 unit, the final endpoint is computed. The algorithm chosen to sum the two g-numbers is the one shown in [2]: 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 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.

$$\begin{cases} \underline{Z} = \min\big(\bigtriangledown\big(\underline{G1G2}\big), \bigtriangledown\big(\underline{G1}\overline{G2}\big), \bigtriangledown\big(\overline{G1}\underline{G2}\big), \bigtriangledown\big(\overline{G1G2}\big)\big) \\ \overline{Z} = \max\big(\bigtriangleup\big(\underline{G1G2}\big), \bigtriangleup\big(\underline{G1}\overline{G2}\big), \bigtriangleup\big(\overline{G1}\underline{G2}\big), \bigtriangleup\big(\overline{G1G2}\big)\big) \end{cases} \tag{2}$$

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 [2]. 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.

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

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.

5. Synthesis results and comparison between UNUM and IEEE 754

In this work we have described in hardware and validated the four basic floating point UNUM operators (+,-,*,/). The division was validated with the fixed point mantissa division described as behavioral ($/$ in VHDL). Unfortunately it is unsuitable for synthesis in this context. Therefore there will be no synthesis result for division in this submission.

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 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 each unit is showed the timing latency (ns), the area (number of cell/gate instances chosen by the synthesizer and μm^2), and the power consumption (mW). For each operator replicated inside the g-layer (g_add and g_mul),

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

	Timing	Area		Power
Unit	(ns)	(cells)	(μm^2)	(mW)
u_add	16.59	41852	132398	1.89
U2G in add	3.54	16573	52429	0.46
g_add	8.98	16113	50973	0.94
		= 2×8057	= 2×25486	= 2× 0.47
64-bit FP add	*8.23*	*7183*	*23068*	*1.75*
G2U in add	4.07	9166	28995	0.49
u_mul	16.55	83430	375472	8.37
U2G in mul	3.10	13933	62704	0.64
g_mul	9.20	63157	284232	6.86
		= 4×16790	= 4×71058	= 4×1.72
64-bit FP mul	*8.51*	*17635*	*77086*	*4.53*
G2U in mul	4.25	6341	28536	0.85
u_cmp	4.00	20273	66512	2.75
U2G in cmp	3.38	18651	61191	2.37
g_cmp	0.62	1622	5321	0.37

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.

Comparing gOP and IEEE units synthesis results, 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.

As highlighted in Table 1, gOP units individually are not so different with respect to standard IEEE ones. The main differences (and design complexities) are the conversion functions inside each uOP unit. In fact they 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 assessment of this trade-off. It also discusses some of the design choices that must be made.

The next steps are: to implement and synthesize the g-bound divider unit core, to pipeline the designed units, to complement the proposed UNUM units with a matching register file, to implement a full processor, in order to be able to compare the area and power overhead with the data data transfer with respect to IEEE 754 standard operations.

This requires some deeper rethinking, in particular in the control flow management instructions. For example comparisons require three outputs and not only two as in usual processors. This changes the usual control flow. 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] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985, 1985.

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

[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