An 826 MOPS, 210 uW/MHz Unum ALU in 65 nm

Florian Glaser*, Stefan Mach*, Abbas Rahimi*†, Frank K. Gürkaynak*, Qiuting Huang*, Luca Benini*‡
*Integrated System Laboratory (IIS), ETH Zurich, Zurich, Switzerland.
†EECS Department, University of California, Berkeley, USA.‡DEI, University of Bologna, Italy.
{glaser, mach, abbas, kgf, huang, benini}@iis.ee.ethz.ch

***Abstract*—To overcome the limitations of conventional floating-point number formats, an interval arithmetic and variable-width storage format called universal number (unum) has been recently introduced [1]. This paper presents the first (to the best of our knowledge) silicon implementation measurements of an application-specific integrated circuit (ASIC) for unum floating-point arithmetic. The designed chip includes a 128-bit wide unum arithmetic unit to execute additions and subtractions, while also supporting lossless (for intermediate results) and lossy (for external data movements) compression units to exploit the memory usage reduction potential of the unum format. Our chip, fabricated in a 65 nm CMOS process, achieves a maximum clock frequency of 413 MHz at 1.2 V with an average measured power of 210 uW/MHz.**

***Index Terms*—universal number (unum), floating-point, interval arithmetic, computing accuracy, ASIC, ALU**

I. Introduction

Large scale data analytics and numerical applications have very widely ranging requirements in terms of numerical precision. While approximate computing shows flexibility with low precision arithmetic and aggressive bit width reduction [2], the other side of the application spectrum adheres to the IEEE standard for floating-point arithmetic [3] (IEEE 754) in spite of its possible side effects e.g., accumulation of rounding errors [4] that can cause deviation from the exact value. To cover this wide range of demands, efficient hardware solutions that retain as much flexibility as possible, are highly desirable.

The IEEE 754 format mainly suffers from rigid allocation of bits to its sign, exponent and mantissa fields and lacks robustness to rounding errors [5]. The latter weakness is caused by the implicit rounding rules defined in the standard: When a desired value lies in between of two representable values, it will be forced to be rounded to the next best value producing an inevitable rounding error; across multiple calculations, such rounding error can be accumulated without allowing the application an explicit observation or control over the error. As an alternative, the universal number (unum) [6] format is proposed by John L. Gustafson to better control precision loss. The goal of unum is to overcome the limitations of the IEEE 754 format by introducing a variable-width storage format, and a *ubit* which determines whether a unum corresponds to an exact number or an interval between exact unums, hence explicitly representing when a calculation produces a value that is not exactly representable in the number system. Therefore, the *ubit* explicitly enables *encoding* the error bound. The unum format additionally defines two fields that make the number self-descriptive, as discussed briefly in Section II and deailed in [1].

The unum format, so far, has been supported in various programming environments including Julia [7], Matlab [8], Python [9], J and Mathematica [6] languages. VHDL code of three operators (addition, multiplication, and comparison) has been synthesized [10], and an FPGA implementation targets four operators (addition, subtraction, multiplication and division) [11]. To clearly evaluate the benefits and challenges of unum hardware design in silicon, we present – to the best of our knowledge – the first ASIC as a fully operational unum processor capable of performing additions and subtractions as well as format-specific functions for lossless and lossy compressions. This paper makes the following contributions:

- We present an ASIC integrating a unum arithmetic unit (ALU), supporting addition, subtraction, implicit lossless and explicit lossy compression, measuring 0.07 mm^2 in 65 nm CMOS.
- We report measurement results of the fabricated chip, achieving a maximum clock frequency of 413 MHz at 1.2 V with an average power of 210 µW/MHz.
- We critically analyze advantages and shortcomings in supporting the unum format in hardware.

The rest of the paper is organized as follows. Section II provides background on the unum format, how to perform computations with it and discusses associated advantages and shortcomings in terms of precision and memory footprint. Section III presents synthesis experiments for the IEEE 754 and unum compatible arithmetic units, followed by the design and optimization of the implemented ALU. In Section IV, we present the chip implementation and experimental results. Finally, Section V concludes the paper with a discussion of results.

II. Unum Computing Background

A. The Unum Format

The unum format, depicted in Fig. 1 bears similarity to the IEEE 754 floating-point representation for real numbers with its *sign-exponent-mantissa* notation. The unum format extends this representation by adding three new fields that allow for the

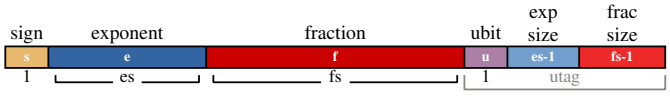

Fig. 1: The unum format, extending *sign-exponent-mantissa* floats with self-descriptive fields in the *utag*.

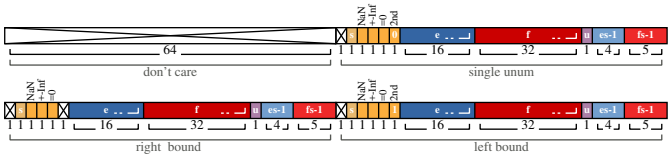

Fig. 2: Layout of the internal representation of single unums (top) and ubound values (bottom) in the 128-bit register file.

inclusion of self-descriptive information about the represented value. These additional fields are summarized under the name *utag*.

The last two fields in the utag denote the exponent size es and fraction size fs of the unum, making unum a variable-size format. Hence, floating-point values that can be represented with a small number of bits require fewer storage bits compared to a large fixed-size floating-point environment thanks to the self-descriptive nature of the utag.

Since it is practically not feasible to allow for unlimited exponent and mantissa sizes, the widths of the exponent size and fraction size fields in the utag are fixed, defining the maximum range of possible unum values. The chosen widths for the exponent size and fraction size fields then define a so-called unum *environment*. For example, setting the exponent size width to 4 bits and the fraction size width to 5 bits, the resulting environment can represent unums with *up to* 16 exponent and *up to* 32 fraction bits. Such unums are defined in a {4,5}-environment – the maximum possible size of a unum in an {a,b}-environment is given as $\text{maxubits} = 2 + 2^a + 2^b + a + b$.

The first field in the utag, called the *ubit*, can be set to denote that the represented value x is not an exact point on the real line, but rather an open interval $(x, x + ulp)$ with ulp being the unit in the last place for the current unum format. Explicitly encoding that the exact value cannot be represented in the current format sets unum apart from regular floating-point representations where all encoded values are considered as exact and approximation is completely implicit.

For describing general intervals more than one *ulp* apart, two unums can be connected to create a so-called *ubound*[1], each denoting one endpoint of an interval. In a ubound, each of the two ubits indicates whether the respective endpoint is part of the interval or not, i.e., whether the interval is closed or open there.

B. Unum Operations

In this work's implementation, we include the basic operations that are addition and subtraction. Unum addition is similar to floating-point addition, with more complex special cases involving infinities being dependent on both values and bound types. The left and right bound of ubounds can be handled independently, however.

One complexity of the floating-point arithmetic, namely rounding, is greatly simplified in unum: whenever the result of an operation on two exact values requires more precision than available in the unum environment, the ubit is set to mark the value as inexact. When handling bounds, the bound type of the result bound corresponds to the logical-OR of its operand ubits.

Since the bit-pattern representation of a value is not unique within a unum environment, there are additional unum-specific operations to be considered. Since implementations should strive to utilize as little bits as possible for a given value, we also define the lossless *optimize* operation, calculating the representation of a ubound with the smallest number of bits. Furthermore, Gustafson [1] specifies the *unify* operation that attempts to merge a ubound consisting of two unums into the smallest single unum that fully includes the interval. This operation can incur loss of precision, namely if the resulting inexact unum covers a larger interval than the initial ubound.

C. Considerations for Unum in Hardware

The interchange format for unums as shown in Fig. 1 is specified in [1]. Unum values reside in memory in this format, using only as much storage as mandated by the exponent size and fraction size fields – which can be drastically less than using a fixed-width floating-point representation. This departure from using uniformly sized and aligned operands however requires additional effort when handling unums in the memory system.

In order to illustrate the dynamic behavior of unum during calculations, *axpy* was run with input coefficients of rising complexity, calculating and accumulating the result using either floats or unum environments. The change of the relative error compared to a double precision reference as well as the bit-size over the iterations is shown in Fig. 3.

During phase I, only small coefficients are used, leading to results that can be exactly represented in all evaluated formats. The size of unum results is made up of the fixed size of the utag – 8 bit and 10 bit, respectively, for the {3,4} and {4,5} environments – and the dynamic number of bits needed to store the actual value.

Phase II applies large coefficients, significantly increasing the accumulated values. Unum formats start increasing in size to still accurately store the result. Once the exact value requires more fraction bits than available in the format, error

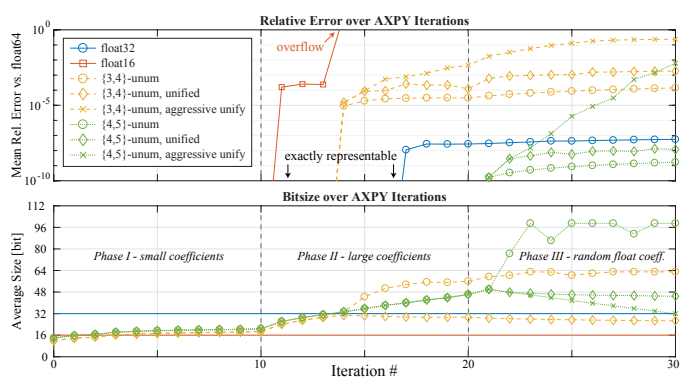

Fig. 3: Relative error of *axpy* iterations using floating-point and unum formats (top) and the bit-size of the results (bottom).

[1]This definition deviates from Gustafson's definition in [1], where the term *ubound* can also denote a single unum with the *ubit* set.

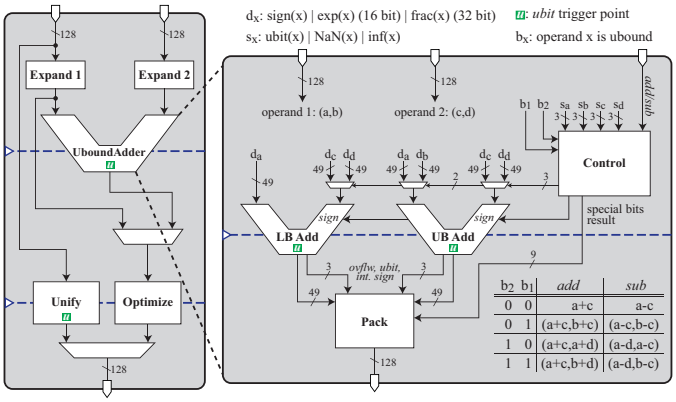

Fig. 4: Data path of the proposed, 128-bit wide ALU and architecture of the unum adder along with supported operations. Blue lines indicate automatically retimed pipeline stages.

proportional to the format-specific minimal *ulp*-width appears and unum starts using ubounds to accurately represent the uncertainty of the results.

In phase III more error is introduced by using random floats as coefficients, causing also the {4,5}-unum's 32 fraction bits to be insufficient for exact results.

The ubounds used for unum results would require significantly more storage space than floats, thus they should stay contained within the processing unit registers if possible. Before storing to main memory, *unify* can be used to reduce storage size at the cost of increasing the error bound. *Unify*ing excessively, for example after each iteration as shown in Fig. 3, causes the additional error introduced by each unification to quickly accumulate.

We notice in this example that there is a range where unum provides lower memory footprints than float32 with equivalent accuracy, while float16 error already grows rampant. *Unified* {3,4}-unums require 7% less memory than float32 at the price of a significant error increase similar to float16 – while remaining usable long after float16 overflows due to insufficient range. *Unified* {4,5}-unums require around 45% more storage than float32 values mostly due to utag overhead – albeit at around 5× lower error and explicitly denoting this error. Using float32 interval arithmetic to store the error bound would cost 39% more memory compared to unum in this example.

Since arithmetic units and register files must be provisioned for handling all possible unums in a given environment, this incurs a relative hardware overhead for those unums that do not use the maximum width of the environment. Unpacking of unum values in the register file and the storage of additional meta-information, called *summary bits* in [1], can simplify the implementation of unum operations, especially the handling of bounds and special cases such as NaN and infinity operands. As our ALU is targeted to extend embedded processing systems, we follow this approach in our implementation.

III. Unum ALU Design

We present a fully unum-{4,5} compatible ALU with support for a subset of the arithmetic and unum-specific operations proposed in [1]. The design is targeted for integration into embedded parallel processing systems as a tightly memory-coupled accelerator, or a core data path extension. We thus follow the hardware-oriented unpacked data format for representing unums proposed in [1] to a large extent; details of the employed format are shown in Fig. 2. One single unum operand in this internal format is 64 bit wide.

TABLE I: Post-layout area distribution of the proposed ALU

Overall ALU area	**50 kGE / 0.07 mm^2**
Lower, upper bound adder, each	14 %
Expand units, each	17 %
Unify unit	27 %
Optimize unit	7 %
Control, data routing	6 %

The maximum number of bits needed to represent these unums is $\mathrm{maxubits} = 59$ bits. We add the summary bits for *NaN*, $\pm\infty$, *=0* as well as the *2nd* flag to mark a unum part of a ubound, the ALU datapath that supports parallel operations for ubounds is therefore 128-bit wide.

A. ALU Architecture

The ALU is depicted in Fig. 4 and can perform additions and subtractions on either two ubounds, two unums or one ubound and one unum. Additionally, the format-specific functions *optimize* and *unify* were implemented: With *optimize*, lossless compression is provided on the one hand by calculating the representation with the smallest exponent and fraction size for a given unum or ubound. On the other hand, the (usually) lossy *unify* reduces a ubound to a unum whenever possible, saving potentially half of the storage at the expense of precision. Consequently, the adder and the *unify* unit can possibly output inexact results from exact operands; this behavior is deeply manifested in the unum format by a set *ubit*. All units with the capability of introducing this format-specific number property are marked with an inverted, green *u* in Fig. 4.

B. Unum Adder

The adder is internally split into separate data paths for the calculation of the resulting upper and lower bounds in case any of the operands is a ubound. The operands are denoted as (a, b) and (c, d) in case of ubounds and a and c in case of unums,

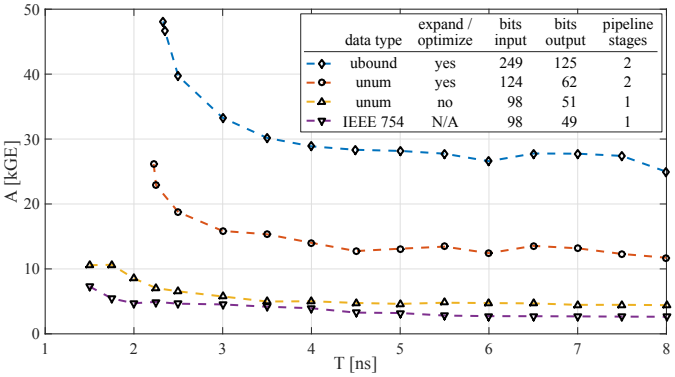

Fig. 5: Area and timing comparison of the proposed ubound adder and its sub-parts against an IEEE 754 compliant adder.

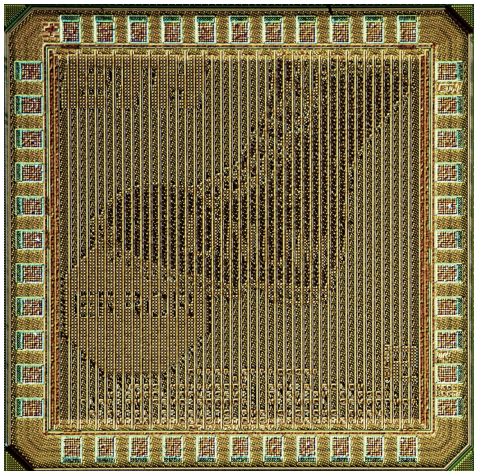

Fig. 6: Die micrograph of the taped-out ASIC.

respectively. In order to take advantage of the regularity of the floating-point arithmetic units, the operands are expanded to the maximum supported precision with 16 exponent and 32 fraction bits beforehand. The core of each adder then consists of a floating point adder of appropriate size with hidden bit, overflow and rounding support, complemented with checks for unum infinity, zero and NaN special cases. Most importantly, however, the adder detects if its result cannot be represented exactly and sets the *ubit* in such cases.

C. Optimized Compression

Particular focus was put on optimizing the routing of data through the available compression units during ALU design: The *optimize* operation is carried out both through a dedicated opcode as well as implicitly after every adder operation to leverage the storage-saving capabilities of the unum format; the *unify* operation can only be carried out with an explicit opcode to maintain controllability over all lossy operations. In a typical processing environment, intermediate results can then be successively *optimized* while *unified* only once and right before expensive data movements, e.g., DRAM transfers. This mechanism allows for maximum storage savings while not sacrificing desired intermediate precision.

D. Comparison with IEEE 754

Fig. 5 shows synthesis experiments in 65 nm, comparing different unum-enabled arithmetic units with an IEEE 754 compliant floating-point adder with corresponding exponent and fraction sizes. A first observation is a modest area increase (27% or 1.08 kGE with a 4 ns period constraint) when only considering the unum adder. However, complementing the adder with the expand and *optimize* units to take advantage of on-the-fly data compression comes with an area increase of more than 3.5×. The implemented, fully-parallel ubound adder adds roughly another factor of two while also providing double the throughput. The second important observation is the limitation in terms of minimum clock period for the compression-enabled unum units, even with an additional pipeline stage. Table I confirms the findings that compression-related blocks consume a significant part of the overall ALU area; they however can be reused and shared between arithmetic operations.

IV. ASIC Implementation

For silicon verification and characterization, we embedded the proposed ALU into a test-bed consisting of instruction SRAM, register file and control state machine. A maximum of 1024 instructions can be executed sequentially once or repeatedly, hiding IO delays to emulate operating conditions resulting from integration into embedded processing systems.

A. Experimental Setup

Both SRAM and register file are accessible for writing and reading through dedicated commands to a memory controller block; consequently, the maximum ALU speed can be determined after preloading instruction memory and register file with suitable instructions and data, respectively. Results from the register file are then read out and verified against a golden model implementation [9]. The design nets 0.258 mm^2 of circuit area within the ASIC die pictured in Fig. 6.

B. Experimental Results

The fabricated prototypes were characterized on a commercial *Advantest SoC V93000* ASIC tester, using full-range data generated in a directed random fashion. The findings with further ASIC properties are summarized in Table II.

V. Conclusion

We presented measurement results of the first unum-{4,5} ALU ASIC implementation. Our 128-bit wide ALU supports addition and subtraction of ubounds and the unum-specific operations *optimize* and *unify* at up to 413 MHz, allowing up to 826 M unum additions or subtractions per second. We discussed synthesis experiments for the comparison of unum-enabled arithmetics with the IEEE 754 counterparts and conclude that it must be carefully analyzed whether memory accesses are expensive enough for the significant (de)compression overhead linked to variable-width number formats to pay off. Furthermore, we touched on the possible storage-saving capabilities of the unum format through an example, concluding that unum formats provide moderate memory footprint advantage (7%) with respect to the standard FP32 and wider range than FP16, at a price of a significant increase in datapath complexity and requiring special care in avoiding aggressive unification to prevent error blow-up.

TABLE II: Measured characteristics of unum-{4,5} ASIC, all numbers acquired from measurements at 1.2 V at room temperature

Technology / Supply		umcL65 / 1.2 V
Circuit Area		0.258 mm^2
Measured Leakage Power		1.3 mW
Measured Dynamic Power		210 µW/MHz
Maximum Speed	Add/Subtract	413 MHz
	Unify	468 MHz
	Optimize	471 MHz

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of David Oelen and Lucas Mayrhofer during ASIC design and testing. Support received from the ETH Zurich Postdoctoral Fellowship program and the Marie Curie Actions for People COFUND Program.

REFERENCES

[1] J. L. Gustafson, *The End of Error: Unum Computing*. CRC Press, 2017.

[2] N. M. Ho, E. Manogaran, W. F. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 63–68.

[3] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[4] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 12:1–12:41, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1353445.1353446

[5] J. L. Gustafson, "A radical approach to computation with real numbers," *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, 2016. [Online]. Available: http://superfri.org/superfri/article/view/94

[6] W. Tichy, "The end of (numeric) error: An interview with John L. Gustafson," *Ubiquity*, vol. 2016, no. April, pp. 1:1–1:14, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2913029

[7] "Unum arithmetic in Julia," 2017. [Online]. Available: https://github.com/JuliaComputing/Unums.jl

[8] M. Kvasnica, "munum: Matlab(R) library for universal numbers," 2017. [Online]. Available: https://bitbucket.org/kvasnica/munum

[9] J. Muizelaar, "Python port of the Mathematica unum prototype from 'The End of Error'," 2017. [Online]. Available: https://github.com/jrmuizel/pyunum

[10] A. Bocco, Y. Durand, and F. de Dinechin, "Hardware support for unum floating point arithmetic," in *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, June 2017, pp. 93–96.

[11] J. Hou, Y. Zhu, Y. Shen, M. Li, Q. Wu, and H. Wu, "Enhancing precision and bandwidth in cloud computing: Implementation of a novel floating-point format on fpga," in *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, June 2017, pp. 310–315.