Decoding-free Two-Input Arithmetic for Low-Precision Real Numbers*

John L. Gustafson[xxxx−xxxx−xxxx−xxxx], Marco Cococcioni[1][0000−0002−7020−1524], Federico Rossi[1][0000−0002−4906−6997], Emanuele Ruffaldi[2][0000−0001−6084−6938], and Sergio Saponara[1][0000−0001−6724−4219]

[1] University of Pisa {marco.cococcioni, federico.rossi, sergio.saponara}@unipi.it
[2] MMI s.p.a eruffaldi@mmimicro.com

Abstract. In this work, we present a novel method for directly computing functions of two real numbers using logic circuits without decoding; the real numbers are mapped to a particularly-chosen set of integer numbers. We theoretically prove that this mapping always exists and that we can implement any kind of binary operation between real numbers regardless of the encoding format. While the real numbers in the set can be arbitrary (rational, irrational, transcendental), we find practical applications to low-precision posit™ number arithmetic. We finally provide examples for decoding-free 4-bit Posit arithmetic operations, showing a reduction in gate count up to a factor of 7.6× (and never below 4.4×) compared to a standard two-dimensional tabulation.

Keywords: decoding-free arithmetic · posit format · low-precision arithmetic · tabulated functions.

1 Introduction

For nearly a century, the method to expressing real numbers on digital computers has been with scientific notation: some form of significant digits (fixed-size storage representing a signed integer) scaled by a base number raised to a signed integer power, also in fixed-size storage. The IEEE 754 standard gave guidance for the details of this two-integer approach.

The artificial intelligence (AI) sector has been pushing the boundaries of Machine Learning (ML) and inference, which has reignited the debate over what is the appropriate representation for real numbers. The bandwidth and storage requirements of 32-bit IEEE standard floats, in particular, have prompted academics to consider 16-bit (and smaller, even down to 2-3-4 bits for extremely quantized neural networks [1]) alternatives to represent the numbers required for AI. According to the IEEE 754 standard, the half precision (binary16) format has 5 exponent bits and 10 fraction bits.

The posit™ number system, which was introduced in 2017, deviates from all previous fixed-field floating-point forms. It features the quire fixed-point accumulator, which is comparable to the Kulisch accumulator [2–4]. The AI Group on Facebook employs posits with the Logarithmic Number System kind of binade [5]. We focus on posits in this paper, but the method is applicable to any collection of $2N$ real-valued values represented by N bits. Posits are particularly well-suited to the approach, as we shall demonstrate in the next sections.

We propose an optimum method for mapping real numbers to integers, allowing us to execute exact two-input arithmetic operations on real numbers with simply integer addition. This significantly reduces hardware complexity (in terms of AND-OR gates), particularly when just a few bits are required to describe the two inputs. We employ a non-linear variant of integer linear programming to get the best mapping. Unlike traditional circuit designs that require decoding a format bit string into the scale (exponent) and significand in order to operate on floats and their variations, our solution just requires an integer mapping (two logic levels), an unsigned integer addition, and another integer mapping. The approach reaches its limit when the integer sizes get too large, but we demonstrate that it works for posit precision adequate for ML and inference.

The paper is organised as follows: i) In section 2 we summarise the posit format and its key properties, ii) in section 3 we recap the standard way to perform binary mathematical operations between real numbers, iii) in section 4 we present the mathematical foundation for the proposed approach, iv) in section 5 we present the problem formulation and the feasibility of finding a solution for such problem, iv) in section 6 we show the application of the proposed approach to the Posit$\langle 4, 0\rangle$ format and we report some quality metrics for the provided solution.

2 The Posit Format

The mapping method we describe in this paper can be applied to any set of real values, including algebraic and transcendental values, simply by assigning each real value to a natural number. Our method can be applied to

* Research supported by Horizon H2020 projects EPI-SGA2 and TextaRossa.

the legacy floating-point formats (floats), but IEEE Standard floats lack a mapping to integers that is one-to-one and onto, and redundant bit patterns make them inefficient at low precision. The IEEE Standard also specifies ten different exception categories and makes asymmetric use of tapered precision, complicating the use of our approach. For these reasons, we will focus on the *posit* format for encoding real values.

The posit format for real numbers was introduced in 2017 [4]. The format is n bits in length, $n \geqslant 2$. There are only two exception values, 0 represented by $00\cdots0$ and Not-a-Real (NaR) represented by $10\cdots0$. Non-exception values have four fields as shown in Figures 1 and 2, with color coding for clarity:

- Sign field: A single bit with digit value s
- Regime field: variable length, composed of a run of $k+1$ 1 bits or $-k$ 0 bits, ended by the opposite bit or by the end of the number
- Exponent field: es bits (bits beyond the end have value 0) representing an exponent e as an unsigned integer
- Fraction field: fraction f with up to $n - es - 3$ significant bits.

The real value r represented by the encoding is

$$r = (1 - 3s + f) \times 2^{(1-2s)\times(2^{es}k+e+s)}.$$

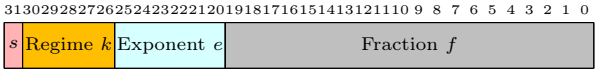

Fig. 1: Bit fields of a posit$\langle 32, 6\rangle$ data type.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	k					e		f							
1	1	1	0			0	0	1	0	0	0	1	0	0	1	1	1

Fig. 2: An example of a 16-bit posit with 3 bits for the exponent size ($n = 16$, $es = 3$). The sign is simply the bit value, $s = 1$. The regime has $(k + 1) = 2$ bits equal to 1 (pair 11) in its run before terminating in a 0 bit, so $k = 1$. The exponent value (unsigned integer) is $e = 1$. The nine fraction bits represent $39/2^9 = 39/512$. The associated real value is therefore $(1 - 3 \cdot 1 + 39/512) \times 2^{(1-2\cdot1)\times(8\cdot1+1+1)} = -1.923828125 \times 2^{-10} \approx -0.0018787$.

While the formula may be non-intuitive, the posit format provides a monotonic mapping of reals to 2's complement signed integers with symmetric dynamic range and symmetric accuracy tapering. It also eliminates non-mathematical complications like "negative zero."

3 Standard two-input arithmetic for reals

Consider a very simple case, that of Posit$\langle 4, 0\rangle$ format. The sixteen values are shown in Table 1.

Table 1: Posit$\langle 4, 0\rangle$ binary representations (bistrings) and corresponding real values

Posit	Value	Posit	Value
1000	NaR	0000	0
1001	-4	0001	1/4
1010	-2	0010	1/2
1011	$-3/2$	0011	3/4
1100	-1	0100	1
1101	$-3/4$	0101	3/2
1110	$-1/2$	0110	2
1111	$-1/4$	0111	4

Notice that the mapping is a *bijection*, and if the posit representation is interpreted as a 2's complement integer, the mapping is also monotone. The Posit Standard treats NaR as "less than" any real value, so posits are ordered. We focus on posit format instead of float format because float format is not a bijection, not monotone, and not ordered, which makes mathematical formalizations awkward and complicated.

Table 2 shows the multiplication table for positive values from the Posit$\langle 4, 0\rangle$ set. Note that the table entries are exact (not rounded to the nearest posit value).

Table 2: Multiplication table, positive Posit$\langle 4, 0\rangle$ values.

×	1/4	1/2	3/4	1	3/2	2	4
1/4	1/16	1/8	3/16	1/4	3/8	1/2	1
1/2	1/8	1/4	3/8	1/2	3/4	1	2
3/4	3/16	3/8	9/16	3/4	9/8	3/2	3
1	1/4	1/2	3/4	1	3/2	2	4
3/2	3/8	3/4	9/8	3/2	9/4	3	6
2	1/2	1	3/2	2	3	4	8
4	1	2	3	4	6	8	16

The table is symmetric because multiplication is commutative and the input row and input column are the same set; the method described here generalizes to non-commutative functions (like division, as shown later) and to inputs from different sets of real values. Two of the entry values are colour-coded (3/8 and 4) to make clear that arithmetic tables can be many-to-one, where several pairs of inputs result in the same value. This is key to understanding the mathematical formalization in the Sections that follow.

The classical hardware implementation of an arithmetic operation on two real arguments in binary float or posit format consists of the following four steps:

1. Test for exception cases using OR or AND trees on the bit fields, and trap to the appropriate output if an exceptional case is detected.
2. Otherwise, decode each argument into its significand and scale factor (exponent), each stored as a signed integer using the usual positional notation.
3. Operate, using traditional circuits for integer operations such as shift, add/subtract, multiply, and count leading zeros. For example, argument multiplication involves the addition of the integer scale factors and integer multiplication of the significands.
4. Encode the result into the format using rounding rules, which for round-to-nearest, tie-to-even requires an OR tree of some of the truncated bits and other logic, and an integer increment if the rounding is upward.

The decoding and encoding are costly for time, circuit resources, and electrical energy compared to the task of simply adding two unsigned integers. The first two steps can be done concurrently (speculatively) to save time, at the cost of wasting additional energy on the path not needed. The stages lend themselves to pipelining to improve throughput, but pipelining slightly increases the latency because of latching the result of each stage.

4 Mapping method and mathematical formalization

Let $X, Y \subset \mathbb{R}$ be two finite sets of real numbers, and $X^*, Y^* \subset \mathbb{N}$ be the sets of bit strings that digitally encode them. The encodings are bijective maps (as an example taken from Table 1, $x_i = \frac{3}{4} \in X$ is encoded as $x_i^* = 0011 \in X^*$). Let $\triangledown$ be any operation on an element of X and an element of Y. Let $Z \subset \mathbb{R}$ be the set of values obtainable as $z_{i,j} = x_i \triangledown y_j$, where $x_i \in X,\ y_j \in Y$. The number of elements in Z, $|Z|$, can be as high as $|X| \cdot |Y|$, when every $x_i \triangledown y_j$ is unique. In general, $1 \leqslant |Z| \leqslant |X| \cdot |Y|$.

Let us introduce the ordered sets of distinct natural numbers $L^x \equiv \{L_i^x\}, L^y \equiv \{L_j^y\}$, (hence $L^x, L^y \subset \mathbb{N}$) and let us suppose that $\exists f^x : X \longmapsto L^x$, f^x being a bijective mapping from the reals in X encoded by X^* to the naturals in L^x. Similarly, let us suppose that $\exists f^y : Y \longmapsto L^y$, f^y being a bijective mapping from the reals in Y encoded by Y^* to the naturals in L^y. Under such hypotheses, each x_i will be uniquely mapped into the corresponding value L_i^x. The same happens for the y_i, which are uniquely associated to L_i^y.

Let L^z be the set of all distinct sums of elements in L^x and L^y: $L^z \equiv \{L_k^z\},\ L^z = \text{distinct}\{L_{i,j}^z\}, L_{i,j}^z = L_i^x + L_j^y$ and let f^z be a mapping between the natural numbers in L^z and Z: $f^z : L^z \longmapsto Z$.

When choosing the values for the sets L^x and L^y, we must ensure that whenever $x_i \triangledown y_j$ and $x_p \triangledown y_q$ differ, the same must happen for the values $L_i^x + L_j^y$ and $L_p^x + L_q^y$:

$$x_i \triangledown y_j \neq x_p \triangledown y_q \Rightarrow L_i^x + L_j^y \neq L_p^x + L_q^y \tag{1}$$

In section 5 we formulate the optimization problem to solve this task, i.e., we show a constructive way on how to build f^x, f^y and f^z (more precisely, on how to obtain the ordered sets L^x, L^y and the function f^z).
If (1) holds, then for any pair of elements $x_i \in X,\ \ y_j \in Y$, we have that:

$$z_{i,j} = x_i \triangledown y_j = f^z(f^x(x_i) + f^y(y_j)) \tag{2}$$

where $+$ is simply the addition between natural numbers, something digital computers can perform perfectly within a finite range using the Arithmetic Logic Unit. Figure 3 summarizes the approach.

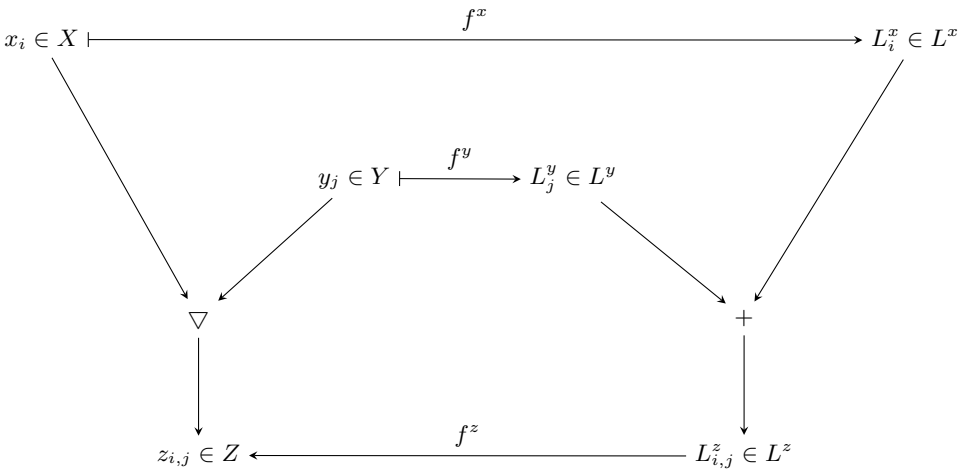

Fig. 3: Mapping between the product of reals and sum of natural numbers. Note that we can generalize it to any kind of operation if we are able to provide the appropriate $f^x()$, $f^y()$ and $f^z()$ functions. All the three functions can be implemented using one-dimensional look-up tables and $f^x(), f^y()$ are bijective functions.

As a recap, instead of implementing a full-fledged hardware processing unit for a given format (e.g., a Floating Point Unit), this approach aims to find **once** the three functions $f^x()$, $f^y()$ and $f^z()$ as shown before. These functions can be straightforwardly implemented as one-dimensional look-up tables and enables us to perform real number arithmetic using only the Arithmetic Logic Unit (ALU). Furthermore, these functions always exist and the optimal mapping can be obtained as shown in Section 5.

4.1 A note on the Z set

As said before, Z can be obtained as the set of values obtainable as $z_{i,j} = x_i \triangledown y_j$, where $x_i \in X,\ y_j \in Y$. Since we want to be able to represent $z_{i,j}, \forall i, j$ for the given format, we are limited by its dynamic range and decimal accuracy. The representable values of the format are equivalent to the set X (and Y, since in all practical implementations $Y \equiv X$).

This means that the following phenomena occur when constructing the Z set:

- Overflow: the result $z_{i,j} = x_i \triangledown y_j > \max(X) = \max(Y)$. Saturation can occur, forcing $z_{i,j} = \max(X) = \max(Y)$.
- Underflow: the result $0 < z_{i,j} < min(|x_i|, x_i \in X), z_{i,j} > 0$. The underflow can be forced to $z_{i,j} = min(|x_i|, x_i \in X)$. The same holds for $z_{i,j} < 0$ and the underflow occurs to $-min(|x_i|, x_i \in X)$.
- Rounding: in general $z_{i,j}$ may not be representable – i.e. $z_{i,j}$ does not belong to X. Depending on the format, a rounding scheme must be applied. As an example, the rounding scheme of Posit numbers is *round to nearest even*.

If we apply the three previous rules (overflow handling, underflow handling and rounding scheme, overall named "`casting`") we get a new set $\hat{Z} = \{\text{distinct}(\texttt{cast}(z_{i,j}))\} = \{\hat{z}_t\},\ \hat{Z} \subseteq X$. Let $|\hat{Z}|$ be the cardinality of such set. Since it can be sorted, we will indicate it as $\tilde{Z} = \{\hat{z}_1, ..., \hat{z}_t, ..., \hat{z}_{|\hat{Z}|}\}$.

For the purposes of this work, this new set $\hat{Z}$ is even more relevant than Z itself, as shown in the following sections.

5 Obtaining the mapping: problem formulation and its solvability

The general problem of finding the mapping can be formulated as the integer programming problem in (3).

$$\begin{aligned}
\min \quad & \sum_i L_i^x + \sum_j L_j^y \\
\text{s.t.} \quad & L_1^x \geq 0 \\
& L_1^y \geq 0 \\
& L_{i_1}^x \neq L_{i_2}^x \qquad \forall i_1 \neq i_2 \\
& L_{j_1}^y \neq L_{i_2}^y \qquad \forall j_1 \neq j_2 \\
& L_i^x + L_j^y \neq L_p^x + L_q^y \quad \forall i,j,p,q \ s.t.\ x_i \triangledown y_j \neq x_p \triangledown y_q \\
& L_i^x, L_j^y \in \mathbb{Z} \qquad \forall i, \forall j
\end{aligned} \tag{3}$$

Since the *not-equals* constraints introduce disjoint domains for the solution, to ease the solver computation, we can exploit characteristics of the specific operation to specialize said constraints.

Let us consider an operation that is non-decreasing monotonic and commutative (e.g. sum and multiplication).

$$\begin{aligned}
\min \quad & \sum_i L_i^x + \sum_j L_j^y \\
\text{s.t.} \quad & L_1^x \geq 0 \\
& L_1^y \geq 0 \\
& L_i^x \geq L_j^x + 1 \qquad i > j \\
& L_i^y \geq L_j^y + 1 \qquad i > j \\
& L_i^x + L_j^y = L_j^x + L_i^y \quad \forall i, \forall j \\
& L_i^x + L_j^y + 1 \leq L_p^x + L_q^y \quad \forall i,j,p,q \ s.t.\ x_i \triangledown y_j < x_p \triangledown y_q \\
& L_i^x, L_j^y \in \mathbb{Z} \qquad \forall i, \forall j
\end{aligned} \tag{4}$$

This integer programming formulation is more tractable, since now the domain is a single polyhedron and not the disjunction of multiple ones. In addition, it can be constructively proved that its feasible region is not empty (see the procedure described in the Appendix 7). Furthermore, the minimization problem is bounded from below, since the variables must stay on the first quadrant and the coefficients of the objective function are all positive.

Under these assumptions, *the problem always admits a minimum for its objective value* [6], although its solution is not guaranteed to be unique. The latter means that different optimal sets L^x and L^y might exist, but they will be associated to the same (optimal) value of the objective function $\sum_i L_i^x + \sum_j L_j^y$.

6 Application: Posit$\langle 4, 0 \rangle$

In this section we show an application of the aforementioned method with a low-precision Posit$\langle 4, 0 \rangle$ format. We applied the method to derive the mapping for the four algebraic operations: $+, -, \times, /$. For each operation we report the $f^x(\cdot)$,$f^y(\cdot)$ and $f^z(\cdot)$ mappings as well as the resulting look-up tables and the respective logic functions that implement the mapping. The four different problems were solved enforcing different policies on the values that L^x and L^y sets can contain. These policies help the solving algorithm to converge to the solution faster. Table 3 summarise the policies adopted for the solution.

Table 3: Policies for the solver algorithm. All policies are to be intended as monotonic.

	L^x	L^y
SUM	Increasing	Increasing
MUL	Increasing	Increasing
SUB	Decreasing	Increasing
DIV	Increasing	Decreasing

We run the solver for the problems defined in section 5 obtaining, for each operation, the following outputs:

- The L^x, L^y and L^z sets.
- The f^x,f^y and f^z functions (or one-dimensional look-up tables) for that perform the mapping between the $X, Y, \hat{Z}$ and, respectively, the L^x, L^y, L^z sets.

Table 4: L^x and L^y sets for Posit$\langle 4, 0\rangle$ ($X \equiv Y \equiv \{ \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 4\}$)

operation	L^x	L^y
$+$	$\{0, 1, 2, 3, 5, 6, 11\}$	$\{0, 1, 2, 3, 5, 6, 11\}$
$\times$	$\{0, 2, 3, 4, 5, 6, 8\}$	$\{0, 2, 3, 4, 5, 6, 8\}$
$-$	$\{0, 1, 2, 3, 5, 6, 7\}$	$\{15, 14, 13, 12, 10, 8, 0\}$
$/$	$\{0, 2, 3, 4, 5, 6, 8\}$	$\{8, 6, 5, 4, 3, 2, 0\}$

The first step consists in obtaining the mapping between the two Posit$\langle 4, 0\rangle$ operands and, respectively, the L^x and L^y sets. Table 4 shows the obtained mapping $(f^x(\cdot), f^y(\cdot))$ for the four different operations. As stated at the beginning of this section, we enforced different L^x, L^y policies for the different operations. Indeed, we can see that, for addition and multiplication, the L^x, L^y sets are identical and monotonic increasing. This reflects the commutative properties of the addition and multiplication. On the other hand, for subtraction and division, the L^x, L^y sets are different: for the division, the two sets have the same elements, but the ordering is different, with L^x being monotonic increasing and L^y monotonic decreasing; for the subtraction the two sets are different, while preserving the same properties of the division ones – i.e. one being monotonic increasing and the other being monotonic decreasing.

Table 5: Cross-sum of the L^x, L^y sets, producing the $L^z_{i,j}$ elements for multiplication (left) and addition (right)

L^x \ L^y	0	2	3	4	5	6	8
0	0	2	3	4	5	6	8
2	2	4	5	6	7	8	10
3	3	5	6	7	8	9	11
4	4	6	7	8	9	10	12
5	5	7	8	9	10	11	13
6	6	8	9	10	11	12	14
8	8	10	11	12	13	14	16

L^x \ L^y	0	1	2	3	5	6	11
0	0	1	2	3	5	6	11
1	1	2	3	4	6	7	12
2	2	3	4	5	7	8	13
3	3	4	5	6	8	9	14
5	5	6	7	8	10	11	16
6	6	7	8	9	11	12	17
11	11	12	13	14	16	17	22

Table 6: Cross-sum of the L^x, L^y sets, producing the $L^z_{i,j}$ elements for division (left) and subtraction (right)

L^x \ L^y	8	6	5	4	3	2	0
0	8	6	5	4	3	2	0
2	10	8	7	6	5	4	2
3	11	9	8	7	6	5	3
4	12	10	9	8	7	6	4
5	13	11	10	9	8	7	5
6	14	12	11	10	9	8	6
8	16	14	13	12	11	10	8

L^x \ L^y	15	14	13	12	10	8	0
0	15	14	13	12	10	8	0
1	16	15	14	13	11	9	1
2	17	16	15	14	12	10	2
3	18	17	16	15	13	11	3
5	20	19	18	17	15	13	5
6	21	20	19	18	16	14	6
7	22	21	20	19	17	15	7

Once we obtained the L^x, L^y sets, the $L^z_{i,j}$ elements can be obtained by computing the cross-sum between the two sets, with all the sum between each pair of elements of L^x and L^y. Table 5 shows the result for multiplication and addition. Due to symmetry properties of the two operations there are several duplicated values. The same holds for division and subtraction operations in 6.

As said above, the set L^z can be found as: $L^z = \text{distinct}\{L^z_{i,j}\}$. Such set will have $|L^z|$ entries, and therefore we can represent it as a ordered set of values in the following way:

$$L^z = \{L^z_1, ..., L^z_k, ..., L^z_{|L^z|}\}$$

having indicated with L^z_k its k-th element and $L^z_1 < L^z_2 < ... < L^z_{|L^z|}$. Let us now introduce the vector $\vec{w}$, having size equal to the cardinality of L^z. We will indicate it as $\vec{w} = \left(w_1, ..., w_k, ..., w_{|L^z|}\right)$. Each component of the vector belongs to $\hat{Z}$. In particular, $w_k = \hat{z}_{i,j}$, if i and j are such that $L^z_{i,j}$ is equal to the k-th value in L^z.

Hereafter we show what we obtained for the *Posit* $\langle 4, 0\rangle$ multiplication:

Ordered sets of real numbers

$X = \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 4\}$

$Y \equiv X$

$\hat{Z} = \{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 4\}$ (in general $\hat{Z} \subseteq X$, but in this case we obtained $\hat{Z} \equiv X$)

Ordered sets of natural numbers

$L^x = \{0, 2, 3, 4, 5, 6, 8\}$

$L^y = \{0, 2, 3, 4, 5, 6, 8\}$

$L^z = \{0,\ 2,\ 3, 4,\ 5,\ 6,\ 7, 8,\ 9, 10, 11, 12, 13, 14, 16\}$

Vector $\vec{w}$

$\vec{w} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2},\ 2,\ 2,\ 4,\ 4,\ 4,\ 4)$

The correspondence among the values of $z_{i,j}$, $\hat{z}_{i,j}$, $L^z_{i,j}$, L^z_k and w_k is shown in Table 7.

Table 7: *Posit* $\langle 4, 0\rangle$ *multiplication*: correspondence among $z_{i,j}$, $\hat{z}_{i,j}$, $L^z_{i,j}$, L^z_k and w_k values. On the left we report the first part, and on the right the second part (to save space).

$z_{i,j}$	$\hat{z}_{i,j}$	$L^z_{i,j}$ $(= L^x_i + L^y_j)$	L^z_k	w_k
1/16	1/4	0	**0**	**1/4**
1/8	1/4	2	**2**	**1/4**
1/8	1/4	2		
3/16	1/4	3	**3**	**1/4**
3/16	1/4	3		
1/4	1/4	4	**4**	**1/4**
1/4	1/4	4		
1/4	1/4	4		
3/8	1/4	5	**5**	**1/4**
3/8	1/4	5		
3/8	1/4	5		
3/8	1/4	5		
1/2	1/2	6	**6**	**1/2**
1/2	1/2	6		
1/2	1/2	6		
1/2	1/2	6		
9/16	1/2	6		
3/4	3/4	7	**7**	**3/4**
3/4	3/4	7		
3/4	3/4	7		
3/4	3/4	7		
1	1	8	**8**	**1**
1	1	8		
1	1	8		
1	1	8		
1	1	8		
9/8	1	8		
9/8	1	8		

$z_{i,j}$	$\hat{z_{i,j}}$	$L^z_{i,j}$ $(= L^x_i + L^y_j)$	L^z_k	w_k
3/2	3/2	9	**9**	**3/2**
3/2	3/2	9		
3/2	3/2	9		
3/2	3/2	9		
2	2	10	**10**	**2**
2	2	10		
2	2	10		
2	2	10		
9/4	2	10		
3	2	11	**11**	**2**
3	2	11		
3	2	11		
3	2	11		
4	4	12	**12**	**4**
4	4	12		
4	4	12		
6	4	13	**13**	**4**
6	4	13		
8	4	14	**14**	**4**
8	4	14		
16	4	16	**16**	**4**

We also report some multiplications using 4-bit posit in Table 8 (we do not report all the combinations for the sake of the space). Note that we are employing the standard rounding scheme for posit numbers, therefore values that are not represented by the posit domain are rounded to the nearest value.

Table 8: Example of multiplication for *Posit* $\langle 4,0\rangle$ using the L^x, L^y, L^z values, for some (x_i, y_j) pairs.

x_i	L_i^x	y_j	L_j^y	$L_{i,j}^z$ $(= L_i^x + L_j^y)$	$z_{i,j}$ $(= x_i \times y_j))$	$\hat{z}_{i,j}$ $(= \texttt{cast}(x_i \times y_j)$
$\frac{1}{2}$	2	$\frac{1}{4}$	0	2	$\frac{1}{8}$	$\frac{1}{4}$
$\frac{1}{2}$	2	$\frac{1}{2}$	2	4	$\frac{1}{4}$	$\frac{1}{4}$
$\frac{1}{2}$	2	$\frac{3}{4}$	3	5	$\frac{3}{8}$	$\frac{1}{4}$
$\frac{1}{2}$	2	$\frac{3}{2}$	5	7	$\frac{3}{4}$	$\frac{3}{4}$
$\frac{1}{2}$	2	2	6	8	1	1
$\frac{1}{2}$	2	4	8	10	2	2

Finally we report in Table 9 the L^z sets for the different operations and the associated $\vec{w}$ vectors for *Posit* $\langle 4,0\rangle$.

Table 9: L^z ordered sets and the associated $\vec{w}$ vectors for Posit$\langle 4,0\rangle$ for the four arithmetic operations.

Op	L^z set	Associated $\vec{w}$ vector
$\times$	$\{0,2,3,4,5,6,7,8,9,10,11,12,13,14,16\}$	$(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{3}{4},1,\frac{3}{2},2,2,4,4,4,4)$
$+$	$\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,16,17,22\}$	$(\frac{1}{2},\frac{3}{4},1,1,\frac{3}{2},\frac{3}{2},2,2,2,2,2,4,4,4,4,4,4,4)$
$/$	$\{0,2,3,4,5,6,7,8,9,10,11,12,13,14,16\}$	$(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{2},\frac{3}{4},1,\frac{3}{2},2,2,4,4,4,4)$
$-$	$\{0,1,2,3,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22\}$	$(\{\frac{1}{4}$ repeated 14 times$\},2,\frac{1}{2},2,1,2,4,4,4)$

As a final remark, observe how the function $f^z()$ described in Section 4 (in particular, in equation (2)), is easily obtainable using the the values in the vector $\vec{w}$. Indeed, the $f^z : L^z \longmapsto \hat{Z}$ we are looking for is the one-dimensional lookup table having entries (L_k^z, w_k). An example of such a lookup table is given by the last two columns of Table 7.

6.1 Quality metrics

When evaluating the results produced by the solver we want to have a baseline benchmark to compare our results. Let us have a Posit$\langle N,E\rangle$, if we think about a look-up table to accommodate the results of an operation, the simplest approach we can have is a 2-dimensional look-up table indexed by the elements of the integer sets X^*,Z^* that digitally encode the respective real values contained in the set X, Y. Each cell of the table has N bits for the output while the address of the look-up table is just the concatenation of the integer representations, thus on $2 \cdot N$ bits. Therefore the table has $2^{2 \cdot N}$ entries of N bits. Table 10 shows this *naïve* approach for a 4-bit posit with the multiplication operation. As we said before, such table has $2^{2 \cdot N} = 2^8 = 256$ entries with 4-bit wide cells, totalling to $256 \cdot 4 =$ **1024 bits**. An optimized version of this table may operate just on the positive part of the domain, reducing the number of entries to $2^{2 \cdot (N-1)} = 2^6 = 64$ with 3-bit wide cells, totalling to $64 \cdot 3 =$ **192 bits**.

Table 10: Naïve look-up table for the multiplication for *Posit* ⟨4.0⟩.

	Naïve $L^z_{i,j}$ (=L^x_i+L^y_j)	Posit encoding of $\hat{z}_{i,j}$	$\hat{z}_{i,j}$
row 1	**0000***0000*	0000	0
	...	...	...
row 16	**0000***1111*	0000	0
row 17	**0001***0000*	0000	0
	...	...	...
row 32	**0001***1111*	1111	−1/4
row 33	**0010***0000*	0000	0
	...	...	...
row 48	**0010***1111*	1111	−1/4
row 49	**0011***0000*	0000	0
	...	...	...
row 64	**0011***1111*	1111	−1/4
row 65	**0100***0000*	0000	0
	...	...	...
row 80	**0100***1111*	1111	−1/4
row 81	**0101***0000*	0000	0
	...	...	...
row 96	**0101***1111*	1111	−1/4
row 97	**0110***0000*	0000	0
	...	...	...
row 113	**0110***1111*	1110	−1/2
	...	...	...
	...	...	...
row 240	**1111***0000*	0000	0
	...	...	...
row 256	**1111***1111*	0001	1/4

When we obtain a solution to the problem of mapping we need to compare it at least against to the baseline solution to see if it actually introduces an improvement. Such improvement can be in the number of entries of the look-up(s) table(s), in the number of output bits or in the combination of the two factors. Moreover, we may also consider that, for low-precision format, we can derive a combinatorial logic function that performs the f^x, f^y, f^z mappings.

Since we deal with a 4-bit format, it is worth to consider the gate count of a combinatorial solution to the problem. Suppose that, instead of implementing the mapping with a look-up table, we implement the mapping using a combinatorial logic function of the input bits. We can evaluate the cost of this solution in terms of number of AND-OR gates (ignoring the cost of *NOT* gates, as usually done). Table 11 shows the gate cost for each operation and the comparison with the naïve solution presented before as a baseline benchmark.

Table 11: Total gate count AND-OR for each operation for Posit⟨4, 0⟩.

	Total gates for L^x	Total gates for L^y	Total gates for L^z	**Grand total gates**	Grand total gates of the naïve solution	Gate reduction
+	10	10	11	**31**	138	4.4×
×	7	7	9	**23**	138	6×
−	8	5	5	**18**	138	7.6×
/	7	7	9	**23**	138	6×

7 Conclusions

In this paper, we described a novel method for directly computing functions of two real numbers without decoding using logic circuits; the real numbers are mapped to a specially chosen set of integer values. We demonstrated that this mapping exists all the time and that we can implement any type of binary operation between real numbers independent of encoding scheme. In particular, we applied this method to the 4-bit posit format, obtaining the mapping for all the 4 algebraic operations. Finally, we compared the obtained solution to a baseline benchmark in terms of number of look-up table entries and gates count. We showed how our approach can produce mapping tables that are smaller than a traditional look-up table solution with logic functions that implement the mappings having a lower AND-OR gate count when compared to the baseline solution.

Acknowledgments

Work partially supported by H2020 projects EPI2 (grant no. 101036168, `https://www.european-processor-initiative.eu/` and TextaRossa (grant no. 956831, `https://textarossa.eu/`).

Appendix: How to build an initial feasible solution

An initial feasible solution (useful to speedup Matlab `intlinprog` function) can be constructed as shown below. We will focus on the positive values in X, different both from NaR (notice that we are excluding the zero as well). Let us call this set $\mathcal{X}$. Let us indicate with $x_i^* \in \mathcal{X}$ the corresponding bistring (in the next we will refer to the *Posit* $\langle 4,0\rangle$ case, as an example). Therefore, the bistrings will be the ones of *Posit* $\langle 4,0\rangle$ without its most significant bit (see Table 1).

- Each $x_i \in \mathcal{X}$ is mapped to the natural number $L_i^x = x_i^* \cdot 2^n$, n being the maximum number of bits needed for representing the x_i (in the case of *Posit* $\langle 4,0\rangle$, $n = 3$)
- Each $y_j \in \mathcal{Y}$ is mapped to the natural number $L_j^y = y_j^*$

Therefore, we obtain the L^x, L^y sets: $L^x : \{x_1^* \cdot 2^n, \ldots, x_{|\mathcal{X}|}^* \cdot 2^n\}$ and $L^y : \{y_1^*, \ldots, y_{|\mathcal{Y}|}^*\}$. Each $L_{i,j}^z \in L^z$ is obtained by the concatenation of the bit strings x_i^*, y_j^* (or equivalently, as $L_{i,j}^z = L_i^x + L_j^y$, as shown in Table 12).

We now prove that this solution satisfies the constraint given in equation (1):

- Since there are no conflicting encodings of the real numbers in $\mathcal{X}$ and $\mathcal{Y}$, we can guarantee that different real numbers have different bit-strings that digitally encode them. Therefore, $L_i^x \neq L_j^x, \forall i \neq j$ and $L_p^y \neq L_l^y, \forall k \neq l$.
- Since all the encodings in L^x, L^y are different from each other, also the concatenation of any pair L_i^x, L_j^y is unique. Therefore, $L_{i,j}^z \neq L_{k,q}^z$, if $x_i \triangledown y_j \neq X_j \triangledown Y_q$(with $\triangledown$ we indicate the generic operation for which we are finding the mapping).
- Being the values $L_{i,j}^z$ unique (no duplicates), we can easily obtain the ordered set L^z, by sorting them.
- the k-element vector $\vec{w}$ can be trivially obtained as $\texttt{cast}(x_i \triangledown y_j)$, when $i \cdot 2^n + j = k$.

An example of feasible solution for the multiplication of *Posit* $\langle 4,0\rangle$ numbers is reported in Table 12.

Table 12: Example of a feasible solution for positive values of *Posit* $\langle 4, 0\rangle$. Notice that $L_i^x = x_i^* \cdot 2^3$ and $L_j^y = y_j^*$.

x_i	x_i^*	L_i^x	y_j	y_j^*	L_j^y	$L_{i,j}^z$	$L_{i,j}^z$	L_k^z	w_k
	(base 10)	(base 2)		(base 10)	(base 2)	(base 2)	(base 10)	(base 10)	
1/4	1	**001**000	1/4	1	*001*	**001***001*	9	9	1/4
1/4	1	**001**000	1/2	2	*010*	**001***010*	10	10	1/4
1/4	1	**001**000	3/4	3	*011*	**001***011*	11	11	1/4
1/4	1	**001**000	1	4	*100*	**001***100*	12	12	1/4
1/4	1	**001**000	3/2	5	*101*	**001***101*	13	13	1/4
1/4	1	**001**000	2	6	*110*	**001***110*	14	14	1/2
1/4	1	**001**000	4	7	*111*	**001***111*	15	15	1
1/2	2	**010**000	1/4	1	*001*	**010***001*	17	17	1/4
1/2	2	**010**000	1/2	2	*010*	**010***010*	18	18	1/4
1/2	2	**010**000	3/4	3	*011*	**010***011*	19	19	1
1/2	2	**010**000	1	4	*100*	**010***100*	20	20	1
1/2	2	**010**000	3/2	5	*101*	**010***101*	21	21	3/4
1/2	2	**010**000	2	6	*110*	**010***110*	22	22	1
1/2	2	**010**000	4	7	*111*	**010***111*	23	23	2
3/4	3	**011**000	1/4	1	*001*	**011***001*	25	25	1/4
3/4	3	**011**000	1/2	2	*010*	**011***010*	26	26	1/4
3/4	3	**011**000	3/4	3	*011*	**011***011*	27	27	1/2
3/4	3	**011**000	1	4	*100*	**011***100*	28	28	3/4
3/4	3	**011**000	3/2	5	*101*	**011***101*	29	29	1
3/4	3	**011**000	2	6	*110*	**011***110*	30	30	3/2
3/4	3	**011**000	4	7	*111*	**011***111*	31	31	2
1	4	**100**000	1/4	1	*001*	**100***001*	33	33	1/4
1	4	**100**000	1/2	2	*010*	**100***010*	34	34	1/2
1	4	**100**000	3/4	3	*011*	**100***011*	35	35	3/4
1	4	**100**000	1	4	*100*	**100***100*	36	36	1
1	4	**100**000	3/2	5	*101*	**100***101*	37	37	3/2
1	4	**100**000	2	6	*110*	**100***110*	38	38	2
1	4	**100**000	4	7	*111*	**100***111*	39	39	4
3/2	5	**101**000	1/4	1	*001*	**101***001*	41	41	1/4
3/2	5	**101**000	1/2	2	*010*	**101***010*	42	42	3/4
3/2	5	**101**000	3/4	3	*011*	**101***011*	43	43	1
3/2	5	**101**000	1	4	*100*	**101***100*	44	44	3/2
3/2	5	**101**000	3/2	5	*101*	**101***101*	45	45	2
3/2	5	**101**000	2	6	*110*	**101***110*	46	46	2
3/2	5	**101**000	4	7	*111*	**101***111*	47	47	4
2	6	**110**000	1/4	1	*001*	**110***001*	49	49	1/2
2	6	**110**000	1/2	2	*010*	**110***010*	50	50	1
2	6	**110**000	3/4	3	*011*	**110***011*	51	51	3/2
2	6	**110**000	1	4	*100*	**110***100*	52	52	2
2	6	**110**000	3/2	5	*101*	**110***101*	53	53	2
2	6	**110**000	2	6	*110*	**110***110*	54	54	4
2	6	**110**000	4	7	*111*	**110***111*	55	55	4
4	7	**111**000	1/4	1	*001*	**111***001*	57	57	1
4	7	**111**000	1/2	2	*010*	**111***010*	58	58	2
4	7	**111**000	3/4	3	*011*	**111***011*	59	59	2
4	7	**111**000	1	4	*100*	**111***100*	60	60	4
4	7	**111**000	3/2	5	*101*	**111***101*	61	61	4
4	7	**111**000	2	6	*110*	**111***110*	62	62	4
4	7	**111**000	4	7	*111*	**111***111*	63	63	4

References

1. M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara, "Small reals representations for deep learning at the edge: A comparison," in *Next Generation Arithmetic*, ser. Lecture Notes in Computer Science, J. Gustafson and V. Dimitrov, Eds., vol. 13253. Springer International Publishing, 2022, pp. 117–133, https://doi.org/10.1007/978-3-031-09779-9_8.
2. J. L. Gustafson, *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.

3. ——, "A radical approach to computation with real numbers," *Supercomputing Frontiers and Innovations*, vol. 3, no. 2, pp. 38–53, 2016.
4. J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.
5. J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018. [Online]. Available: http://arxiv.org/abs/1811.01721
6. M. Conforti, G. Cornuéjols, and G. Zambelli, *Integer Programming*, ser. Graduate Texts in Mathematics. Springer International Publishing, 2014.