



**UNIVERSITÀ DI PISA**

SYMBOLIC AND EVOLUTIONARY  
ARTIFICIAL INTELLIGENCE

A.A. 2024/2025

**Decoding-free Two-Input Arithmetic:  
Case Posit8**

Andrea Sottile   Gianluca A. Cometa

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Quick Recap: The Posit Format</b>	<b>4</b>
2.1	Posit8 and Assignment . . . . .	5
<b>3</b>	<b>Mapping Method and Problem Formulation</b>	<b>7</b>
3.1	Problem Overview . . . . .	9
<b>4</b>	<b>Initial State of the Art</b>	<b>11</b>
<b>5</b>	<b>Practical Implementation for the Problem</b>	<b>12</b>
5.1	Base Implementation: <code>solveMapping.m</code> . . . . .	13
5.1.1	Objective Function . . . . .	13
5.1.2	Constraints . . . . .	13
5.1.3	Solving the ILP Problem . . . . .	15
5.1.4	Advantages and Disadvantages . . . . .	16
5.2	Optimized Implementation: <code>solveMapping_speed.m</code> . . . . .	17
5.2.1	Key Features . . . . .	17
5.2.2	Advantages and Disadvantages . . . . .	18
5.3	Parallelized Implementation: <code>solveMapping_parallel.m</code> . . . . .	20
5.3.1	Key Features . . . . .	20
5.3.2	Advantages and Disadvantages . . . . .	22
5.4	Comparison of Approaches . . . . .	23
5.4.1	Summary of Implementations . . . . .	23
5.5	Interactive Main Menu . . . . .	24
<b>6</b>	<b>Experimental Environment</b>	<b>25</b>
6.1	Profiling Results and Considerations . . . . .	26
<b>7</b>	<b>Experimental Result</b>	<b>31</b>
7.1	Posit (4,0) Implementation Results . . . . .	31
7.2	Posit (8,0) Implementation Results . . . . .	32
<b>8</b>	<b>Future Improvements</b>	<b>33</b>
<b>9</b>	<b>Conclusions</b>	<b>33</b>
<b>10</b>	<b>Bibliography</b>	<b>34</b>

# 1 Introduction

In this paper, we present an extension to the implementation of Posit8 to implement decoding-free arithmetic, taking advantage of the optimization of linear integers.

Starting from the work done in the paper *"Decoding-free Two-Input Arithmetic for Low-Precision Real Numbers"* [1], our main goal was to review and improve the implementation part of the Matlab, to adapt it to the case of **Posit(8,0)**, performing a complete update, fixing bugs, and adopting parallel computing strategies to maximize the performance. The use of real numbers in computers has traditionally followed the representation in scientific notation, based on a combination of a mantissa and an exponent. The IEEE 754 standard, which formalized this representation, dominated the industry for decades.

However, with the evolution of artificial intelligence (AI) and machine learning (ML), the search for alternative number formats has become increasingly relevant.

The use of traditional floating-point numbers, in particular 32-bit numbers, requires high bandwidth and storage requirements, prompting researchers and academics to consider more compact representations, such as 16-bit or even only 2-4-bit variants for highly quantized applications.

The **Posit** numbering system, introduced in 2017, departs from traditional floating-point representations by providing a more efficient and robust structure for the representation of real numbers. In this paper, we focus on the application of **Posit8** to decoding-free arithmetic, an approach that allows mathematical operations to be performed without the need to translate the numerical representation into intermediate formats.

This method represents an optimal solution for mapping real numbers onto integers, allowing arithmetic operations to be performed exclusively by integer sums. Unlike traditional circuits, which require decoding/encoding, the proposed solution is based on a simple logical mapping operation, followed by an integer sum and remapping. It also significantly reduces hardware complexity by optimizing the number of logic gates required for mathematical operations, thereby improving efficiency in low-precision calculations. This approach could significantly reduce computational latency and lower power consumption compared to conventional methods, improving efficiency in low-precision calculations.

## 2 Quick Recap: The Posit Format

The *posit* format is a representation system for real numbers introduced in 2017, designed as an alternative to traditional floating-point formats (IEEE 754). It is particularly effective for low-precision operations and also allows for a symmetric, monotonic and bi-univocal representation of real numbers. The format defines only two special values: *0* and *NaR* (Not-a-Real), avoiding complications such as *zero negative* or the multiple exception categories of IEEE floats.

A posit number is encoded on a fixed number of bits,  $n$ , with  $n \geq 2$ , and presents:

- **Sign Field:** a single bit  $s$  indicating the sign of the number
- **Regime Field:** a sequence of variable length consisting of a run of  $(k+1)$  equal bits (all 1's or all 0's) ending with the opposite bit or the end of the number. The  $k$  value determines the scale
- **Exponent Field:** consisting of  $es$  bits representing the exponent as an unsigned integer (any missing bits are considered to be 0)
- **Fraction Field:** a fractional  $f$  bit, which occupies up to  $n - es - k$  bits and contributes to the precision of the number

The real value  $r$  represented is obtained from the following formula:

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

It combines sign, regime, exponent and fraction. Although not immediately intuitive, this formula guarantees a symmetrical range distribution and a precision that decreases in a controlled manner.

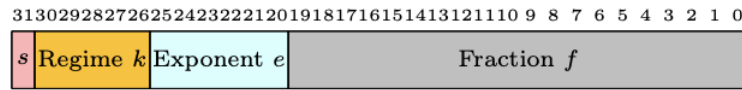


Figure 1: Bit fields of a Posit(32, 6) data type

## 2.1 Posit8 and Assignment

Based on the work[1] described in the introduction, our task was to continue the analysis carried out previously focusing in particular on the specific case represented by **Posit(8,0)**. We then re-performed all tests and evaluations by adopting this numerical format, with the aim of verifying its behavior, performance and reliability. As will be described in more detail later in the paper, during this testing phase we identified some significant issues in the original code, which led us to an almost complete re-implementation of the part in MATLAB. In this new version, we pay particular attention to optimizing performance and correct behavior in relation to the **Posit(8,0)** representation.

The following is a concrete example of a table for the case Posit(8,0), which contains the representation of the first twenty positive numbers:

Posit Bit Pattern	Value	Fraction
00000000	0.000000	0
00000001	0.015625	1/64
00000010	0.031250	1/32
00000011	0.046875	3/64
00000100	0.062500	1/16
00000101	0.078125	5/64
00000110	0.093750	3/32
00000111	0.109375	7/64
00001000	0.125000	1/8
00001001	0.140625	9/64
00001010	0.156250	5/32
00001011	0.171875	11/64
00001100	0.187500	3/16
00001101	0.203125	13/64
00001110	0.218750	7/32
00001111	0.234375	15/64
00010000	0.250000	1/4
00010001	0.265625	17/64
00010010	0.281250	9/32
<b>00010011</b>	<b>0.296875</b>	<b>19/64</b>
00010100	0.312500	5/16
.....	.....	.....

Table 1: Posit(8,0) binary representations (bitstrings) and corresponding real values

The following figure demonstrates the conversion of a Posit(8,0) number with binary representation 00010011 to its corresponding real value 0.296875.

In the Posit(8,0) format, the exponent size is explicitly set to 0, meaning no bits are allocated for an exponent field.

The binary structure consists of:

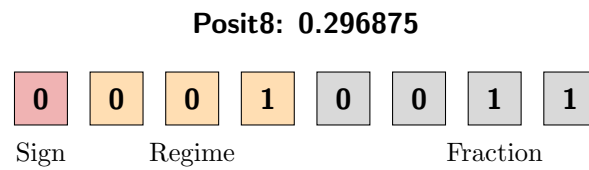


Figure 2: Binary representations of Posit(8,0): 00010011  $\Rightarrow$  Real value: 0.296875

### 3 Mapping Method and Problem Formulation

In the work[1] a method was introduced to represent binary operations between real numbers using a completely natural integer-based approach. The core of the method consists of replacing the direct calculation of an operation between reals by a sum between integers, followed by a final decoding function.

Let  $X, Y \subset \mathbb{R}$  be two finite sets of real numbers, and  $X^*, Y^* \subset \mathbb{N}$  the corresponding sets of digital representations (bitstrings) that encode them. The encodings are bijective, i.e. each real value corresponds to a single binary representation and vice versa.

Let  $\nabla$  be an arbitrary binary operation between an element of  $X$  and one of  $Y$ . Let us denote by  $Z \subset \mathbb{R}$  the set of values obtainable as:

$$z_{i,j} = x_i \nabla y_j, \quad \text{con } x_i \in X, y_j \in Y$$

The maximum number of distinct elements in  $Z$  is  $|X| \cdot |Y|$ , in the case where each combination yields a unique value, but in general there is:

$$1 \leq |Z| \leq |X| \cdot |Y|$$

Let us now introduce two ordered sets of distinct natural numbers:  $L^x = \{L_i^x\}$  e  $L^y = \{L_j^y\}$ , such that there exist two bijective functions:

$$f^x : X \rightarrow L^x, \quad f^y : Y \rightarrow L^y$$

which associate each real element of the initial sets  $X, Y$  with a natural number defined in:  $X^*, Y^*$ . Under this assumption, each  $x_i \in X$  and  $y_j \in Y$  is uniquely associated with  $L_i^x \in L^x$  and  $L_j^y \in L^y$  respectively.

Finally, we define:

$$L^z = \text{distinct}\{L_i^x + L_j^y\}, \quad \text{con } L_{i,j}^z = L_i^x + L_j^y$$

and a function:

$$f^z : L^z \rightarrow Z$$

which associates each integer sum with a real result. For this mapping to be correct, it is essential to ensure that different combinations between elements of  $X$  and  $Y$  that produce different results also generate distinct integer sums:

$$x_i \nabla y_j \neq x_p \nabla y_q \quad \Rightarrow \quad L_i^x + L_j^y \neq L_p^x + L_q^y \quad (1)$$

In the following section, we show how to formulate this problem as an instance of *integer optimization*, to define the functions  $f^x$ ,  $f^y$  e  $f^z$  (i.e. the sets  $L^x$ ,  $L^y$  and the decoding table  $f^z$ ).

When condition (1) is verified, we can always calculate the real result as:

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

In this formulation, the only operation performed at the computational level is a sum between natural integers, an operation that computers can perform efficiently and accurately via the arithmetic-logic unit (ALU). The Figure 3 graphically summarizes this process, highlighting the transition from the real domain to the integer domain, up to the final reconstruction of the result.

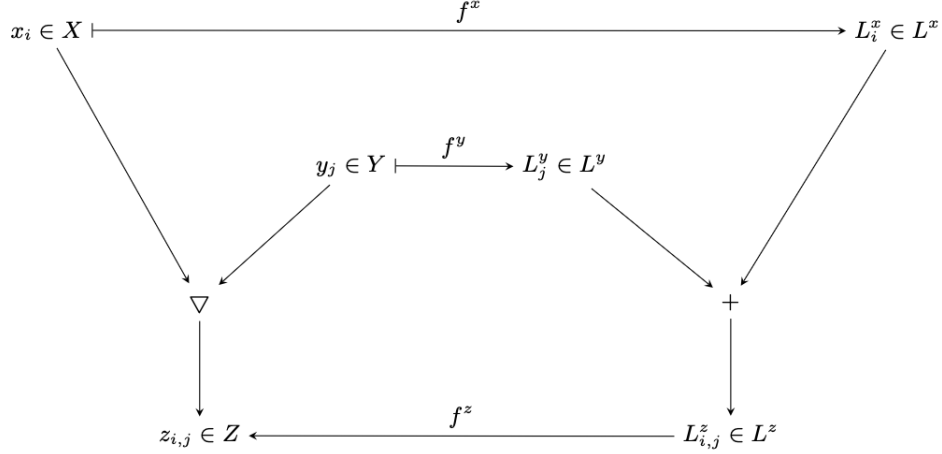


Figure 3: Diagram of the mapping process between operations on reals and sums between natural integers

Figure 3 represents the flow of reasoning in the form of a commutative diagram. The left-hand side shows the direct calculation between reals ( $x_i \nabla y_j$ ), while the right-hand side shows the proposed alternative path:

- $x_i \mapsto L_i^x, y_j \mapsto L_j^y$  through the functions  $f^x$  e  $f^y$
- Calculation of the sum:  $L_{i,j}^z = L_i^x + L_j^y$
- Final conversion by:  $f^z: z_{i,j} = f^z(L_{i,j}^z)$

This approach reduces circuit complexity, particularly for low-precision formats such as  $Posit(4,0)$  or  $Posit(8,0)$ , where arithmetic can be implemented using only integer sums and accesses to pre-computed tables. In the following section, the formulation of the integer optimization problem for finding optimal mappings will be described.



### 3.1 Problem Overview

The problem addressed in this chapter involves finding an optimal mapping for the variables  $L^x$  and  $L^y$  that minimizes the total sum of  $L_i^x$  and  $L_j^y$ , while respecting a set of constraints related to monotonicity and uniqueness. The constraints ensure that the mapping is consistent with the results of a given operation  $\nabla$  (which can be addition, multiplication, subtraction, or division) applied to the elements of two input vectors  $X$  and  $Y$ . The problem is formulated as an Integer Linear Programming (ILP) problem, where the objective is to minimize the sum of the decision variables  $L_i^x$  and  $L_j^y$ , subject to linear constraints.

The primary formulation problem and the ILP problem for the cases of addition and multiplication as in the previous publication[1] are therefore given below, while the ILP problem for the cases of division and subtraction will be shown in 3 .

The general problem of finding the mapping can be formulated as in (1)

$$\begin{aligned}
\min \quad & \sum_i L_i^x + \sum_j L_j^y \\
\text{s.t.} \quad & L_1^x \geq 0 \\
& L_1^y \geq 0 \\
& L_{i_1}^x \neq L_{i_2}^x \quad \forall i_1 \neq i_2 \\
& L_{j_1}^y \neq L_{j_2}^y \quad \forall j_1 \neq j_2 \\
& L_i^x + L_j^y \neq L_p^x + L_q^y \quad \forall i, j, p, q \text{ s.t. } x_i \nabla y_j \neq x_p \nabla y_q \\
& L_i^x, L_j^y \in \mathbb{Z} \quad \forall i, \forall j
\end{aligned} \tag{1}$$

Inequality constraints divide the solution space into several separate parts, making the problem more complex to solve. To simplify, we can adapt these constraints according to the characteristics of the operation we are modeling. Accordingly with the previous work we consider as the first case the one that involves non-decreasing, commutative monotone operations such as **sum** and **multiplication**. So, the problem will be:

$$\begin{aligned}
\min \quad & \sum_i L_i^x + \sum_j L_j^y \\
\text{s.t.} \quad & L_1^x \geq 0 \\
& L_1^y \geq 0 \\
& L_i^x \geq L_j^x + 1 \quad \forall i > j \\
& L_i^y \geq L_j^y + 1 \quad \forall i > j \\
& L_i^x + L_j^y = L_j^x + L_i^y \quad \forall i, \forall j \\
& L_i^x + L_j^y + 1 \leq L_p^x + L_q^y \quad \forall i, j, p, q \text{ s.t. } x_i \nabla y_j < x_p \nabla y_q \\
& L_i^x, L_j^y \in \mathbb{Z} \quad \forall i, \forall j
\end{aligned} \tag{2}$$

In the case of non-commutative operations - referred to as “*monoinv*” - such as subtraction and division, however, natural ordering is not sufficient. So the problem for division and subtraction operations is given below:

$$\begin{aligned}
\min \quad & \sum_i L_i^x + \sum_j L_j^y \\
\text{s.t.} \quad & L_1^x \geq 0 \\
& L_1^y \geq 0 \\
& L_i^x \geq L_j^x + 1 \quad \forall i > j \\
& L_i^y \leq L_j^y + 1 \quad \forall i > j \\
& L_i^x + L_j^y + 1 \leq L_p^x + L_q^y \quad \forall i, j, p, q \text{ s.t. } x_i \nabla y_j < x_p \nabla y_q \\
& L_i^x, L_j^y \in \mathbb{Z} \quad \forall i, \forall j
\end{aligned} \tag{3}$$

Note that under these assumptions starting from the mapping of each of the algebraic operations considered, the four different problems were solved by applying different policies on the values that the sets  $Lx$  and  $Ly$  can contain. For commutative and monotonic operations such as sum and multiplication, the result increases when both operands increase, so it is natural to choose both  $Lx$  and  $Ly$  in increasing order.

On the other hand, for non-commutative operations such as subtraction and division the behavior is asymmetric:

- For  $x - y$  the result increases as  $x$  increases but decreases as  $y$  increases
- For  $x/y$  the result increases with  $x$  and decreases with  $y$

Therefore, for the sum  $Lx(x) + Ly(y)$  (on which  $fz$  then operates) to reflect this property - that is, to be increasing in  $x$  and decreasing in  $y$  - the correct mapping should be:

- $Lx(x)$  “increasing”
- $Ly(y)$  “decreasing”

In fact, examining the results obtained for the format Posit(4,0) , by subtraction it turns out that:

- For  $x$  (minuend) we have  $Lx = [0, 1, 2, 3, 5, 6, 7]$  - that is, increasing
- For  $y$  (subtrahend) we have  $Ly = [15, 14, 13, 12, 10, 8, 0]$  - that is decreasing

For division the criterion is similar ( $Lx$  increasing,  $Ly$  decreasing) because again the operation is increasing with respect to the numerator and decreasing with respect to the denominator.

Contrary to the original work, we noticed an inconsistency between the policy given for the subtraction operation and the one that is most consistent with arithmetic properties: subtraction (**Minus** MATLAB operation) should follow the same logic as division (**RDivide** MATLAB operation), with  $Lx(x)$  increasing and  $Ly(y)$  decreasing. The contrary indication seems to be the result of a typo or a different convention, but in light of the monotonicity and experimental results, the most natural choice is the one **mirroring division’s bidirectional mapping**:

$$L_{i+1}^x > L_i^x \quad \text{and} \quad L_{j+1}^y < L_j^y \quad (4)$$

## 4 Initial State of the Art

At the beginning of our debugging and analysis activities of the previous work, we started from the code available in the **Conga23**[2] branch, which was the most recent and functional basis for the **Posit4** version.

In the initial phase, we also introduced some optimizations aimed at improving the efficiency of the execution for *Posit8*, which was definitely more time-consuming and burdensome.

We were able, thanks to these changes, to complete *plus* and *times* operations, although with extremely long computation times on the order of more than 120 hours.

Next, we attempted to run the *rdivide* operation, but the ILP optimization problem returned **no feasible solution** in the case of *Posit(8,0)*. In particular, we observed that the problem described as *genmonoinv* was unsolvable as defined, this in all probability due to the antisymmetry check not being passed in the case of *Posit8*.

For this reason, and because of the absence of the minus operation in the source code, we decided to reimplement from scratch the central part concerning the generation of the linear optimization problem, with the aim of having more control and flexibility in the definition of the operations. We started from the mathematical formulation mentioned in the previous section.

The following are the main reasons and challenges we faced in terms of optimization for the execution to reach completion in the case required to us for this work on *Posit(8,0)*.

## 5 Practical Implementation for the Problem

This chapter will present three similar code versions that finally return the same results even if however they differ in approach and execution time on the Posit(8,0) case. This is because practical execution showed us several issues, which led us to develop two additional versions to the initial one.

The original version was not sufficient in terms of performance to obtain a final result in a reasonable execution time window. A first approach we took was to redesign the constraint generation section using a better in-memory representation, which, however led intlinprog to a "maximum representation" error.

We then explored the possibility of using other solvers than MATLAB's intlinprog, such as IBM's CPLEX; however, it is a paid proprietary solver and the trial version was not sufficient for our number of constraints. So, in the end, we developed a version very similar to the first but, which adopts a parallel execution that thus allowed us to fully benefit from the performance of the Server that hosted the execution and allowed us to complete our runs.

The three MATLAB files discussed in this chapter represent distinct approaches to solving the same problem, each with its own pros and cons:

- `solveMapping.m`: this is the base implementation, which uses nested loops to generate the constraints. While this approach it is simple and easy to understand, it is also computationally expensive for large  $n$  due to its reliance on explicit loops
- `solveMapping_speed.m`: this implementation optimizes the base approach by leveraging vectorized operations and MATLAB's `meshgrid` function. It avoids nested loops and significantly reduces the execution time, making it suitable for medium to large-scale problems
- `solveMapping_parallel.m`: this implementation further improves performance by parallelizing the computation using MATLAB's `parfor` construct. It distributes the workload across multiple CPU cores, making it ideal for large-scale problems where execution time is critical

## 5.1 Base Implementation: solveMapping.m

The file `solveMapping.m` represents the base implementation of the Integer Linear Programming (ILP) problem.

The goal is to find an optimal mapping for the variables  $L^x$  and  $L^y$  that minimizes the total sum of  $L_i^x$  and  $L_j^y$ , while respecting monotonicity and uniqueness constraints.

The problem is formulated as an ILP problem, where the constraints are expressed as linear inequalities.

### 5.1.1 Objective Function

The objective is to minimize the sum of the decision variables  $L_i^x$  and  $L_j^y$ . This is represented in MATLAB as follows:

---

```
1 f = ones(numVars, 1); % Objective function: minimize sum(Lx) + sum(Ly)
```

---

Here, `numVars` is the total number of variables, which is  $2n$  (where  $n$  is the number of elements in  $X$  and  $Y$ ).

### 5.1.2 Constraints

#### Monotonicity of $L^x$

The first constraint ensures that  $L^x$  is monotonically increasing. This is enforced by the condition:

$$L_{i+1}^x - L_i^x \geq 1 \quad \forall i \in \{1, \dots, n-1\}.$$

In MATLAB, this constraint is implemented as:

---

```
1 for i = 1:(n-1)
2     row = zeros(1, numVars);
3     row(i) = 1;           % Lx(i)
4     row(i+1) = -1;        % Lx(i+1)
5     A = [A; row];         % Add the row to the constraint matrix A
6     b = [b; -1];          % Add the corresponding value to the right-hand side vector b
7 end
```

---

#### Monotonicity of $L^y$ (Depending on the Operation)

The second constraint ensures that  $L^y$  is either monotonically increasing or decreasing, depending on the operation  $\nabla$ .

For example, if the operation is addition or multiplication,  $L^y$  must be increasing. If the operation is subtraction or division,  $L^y$  must be decreasing.

This is implemented as follows:

---

```
1 if isequal(op, @rdivide) || isequal(op, @minus)
2     % Ly must be decreasing
3     for j = 1:(n-1)
4         row = zeros(1, numVars);
```

```

5         row(n+j) = -1; % Ly(j)
6         row(n+j+1) = 1; % Ly(j+1)
7         A = [A; row];
8         b = [b; -1];
9     end
10 elseif isequal(op, @times) || isequal(op, @plus)
11     % Ly must be increasing
12     for j = 1:(n-1)
13         row = zeros(1, numVars);
14         row(n+j) = 1; % Ly(j)
15         row(n+j+1) = -1; % Ly(j+1)
16         A = [A; row];
17         b = [b; -1];
18     end
19 end

```

---

### Uniqueness and Ordering Constraints

The third constraint ensures that for every pair  $(i, j)$  and  $(p, q)$  such that  $X(i)\nabla Y(j) < X(p)\nabla Y(q)$ , the following condition holds:

$$L_i^x + L_j^y + 1 \leq L_p^x + L_q^y.$$

This is implemented using nested loops to iterate over all possible combinations of  $(i, j)$  and  $(p, q)$ . For simplicity, only sum and multiplication functions will be reported:

---

```

1 for i = 1:n
2     for j = 1:n
3         if j > i
4             continue ; % Skip iterations where j > i
5         end
6         for p = 1:n
7             if p == i
8                 continue; % Skip iterations where p == i
9             end
10            for q = 1:n
11                if isequal(op, @times)
12                    if q > p
13                        continue; % Skip iterations where q > p
14                    end
15                    if X(i) * Y(j) < X(p) * Y(q)
16                        row = zeros(1, numVars);
17                        row(i) = 1; % Lx(i)
18                        row(n+j) = 1; % Ly(j)
19                        row(p) = -1; % Lx(p)
20                        row(n+q) = -1; % Ly(q)
21                        A = [A; row];
22                        b = [b; -1];
23                    end
24                elseif isequal(op, @plus)
25                    if q > p
26                        continue; % Skip iterations where q > p

```

```

27         end
28         if X(j) + Y(i) < X(p) + Y(q)
29             row = zeros(1, numVars);
30             row(i) = 1;           % Lx(i)
31             row(n+j) = 1;        % Ly(j)
32             row(p) = -1;         % Lx(p)
33             row(n+q) = -1;       % Ly(q)
34             A = [A; row];
35             b = [b; -1];
36         end
37     end
38 end
39 end
40 end
41 end

```

---

### Equality Constraints (Optional)

In some cases, equality constraints are added to enforce specific relationships between  $L^x$  and  $L^y$ .

For example, if the operation is addition or multiplication, equality constraints may be added to ensure symmetry:

```

1 if isequal(op, @times) || isequal(op, @plus)
2     for r = 2:Nx
3         for c = 1:r-1
4             constrRow = zeros(numVars, 1)';
5             constrRow(r) = 1;
6             constrRow(c + Nx) = 1;
7             constrRow(c) = -1;
8             constrRow(r + Nx) = -1;
9             Aeq = [Aeq; constrRow];
10            beq = [beq; 0];
11        end
12    end
13 end

```

---

### 5.1.3 Solving the ILP Problem

Once the constraints are defined, the problem is solved using MATLAB's **intlinprog** function:

```

1 [x_opt, fval, exitflag, output] = intlinprog(f, intcon, A, b, Aeq, beq, lb, ub,
      options);

```

---

Here, **x\_opt** contains the optimal values for  $L^x$  and  $L^y$ , and **fval** is the minimized objective function value.

### 5.1.4 Advantages and Disadvantages

#### Advantages

- **Simplicity:** The implementation is straightforward and easy to understand
- **Flexibility:** It can handle any operation  $\nabla$  (addition, multiplication, subtraction, division) without significant modifications

#### Disadvantages

- **Efficiency:** The use of nested loops to generate constraints makes this implementation slow for large values of  $n$
- **Scalability:** Due to its computational complexity, this approach is not suitable for large-scale problems

The basic implementation in `solveMapping.m` is ideal for small-scale problems or prototyping, in fact it is fully functional as in the `Posit(4,0)` case. However, for larger problems, more optimised approaches such as *`solveMapping_speed.m`* or *`solveMapping_parallel.m`* are recommended to improve performance and scalability.



## 5.2 Optimized Implementation: solveMapping\_speed.m

The `solveMapping_speed.m` file represents an optimized version of the basic implementation, designed to improve performance through the use of vector operations and MATLAB's `meshgrid` function, which enables the efficient generation of the index combinations required for problem formulation.

We imagine that this is the version among 'solvemappings' with the best computational performance, as it avoids the use of explicit loops and makes full use of matrix processing. However, we were unable to test it due to the input size limit imposed by the `intlinprog` solver. In particular, the fully vector formulation generates very large matrices, and the number of binary variables to be optimised exceeds the internal configured vector size limit, generating the '*vector too long*' error.

From the documentation we have been able to deduce that this happens because `intlinprog` has a limited capacity to handle problems with an extremely large number of variables or matrices (typically exceeding a few million as in our case), the '*vector too long*' error is directly caused by attempting to create or pass to `intlinprog` a vector or matrix that exceeds the solver's internal capacity.

To try to solve this problem, we identified IBM's CPLEX as a possible solution, but due to the problems we mentioned in the previous section, it was not possible for us to test it in its trial version.

### 5.2.1 Key Features

#### Vectorized Operations with `meshgrid`

The core optimization in this implementation is the use of `meshgrid` to generate all combinations of  $X(i)$  and  $Y(j)$  in a vectorized manner.

This avoids the need for nested loops and significantly speeds up the computation. For example, for the addition operation, the code looks like this:

---

```
1 [X_i, Y_j] = meshgrid(X, Y); % Generate all combinations of X(i) and Y(j)
2 sum_ij = X_i(:) + Y_j(:);    % Compute X(i) + Y(j) for all combinations
```

---

Similarly, for multiplication, subtraction, or division, the same approach is used:

---

```
1 if isequal(op, @times)
2     prod_ij = X_i(:) .* Y_j(:); % Element-wise multiplication
3 elseif isequal(op, @minus)
4     diff_ij = X_i(:) - Y_j(:);  % Element-wise subtraction
5 elseif isequal(op, @rdivide)
6     div_ij = X_i(:) ./ Y_j(:);  % Element-wise division
7 end
```

---

### Efficient Constraint Generation with `ind2sub`

After computing the results of the operation  $\nabla$  for all combinations, the constraints are generated efficiently using `ind2sub`.

This function converts linear indices into multidimensional subscripts, allowing the constraints to be constructed without explicit nested loops.

For example:

---

```
1 condition = prod_ij < prod_pq; % Compare all combinations
2 [i_idx, j_idx, p_idx, q_idx] = ind2sub([n, n, n, n], find(condition));
```

---

Here, `find(condition)` returns the linear indices of the elements that satisfy the condition  $X(i)\nabla Y(j) < X(p)\nabla Y(q)$ .

The function `ind2sub` then converts these linear indices into the corresponding multidimensional indices  $(i, j, p, q)$ , which are used to construct the constraints.

### Building the Constraint Matrices

The constraints are built in a vectorized manner using the indices obtained from `ind2sub`:

---

```
1 numRows = numel(i_idx);
2 tempA = zeros(numRows, numVars);
3 tempB = -ones(numRows, 1);
4
5 for k = 1:numRows
6     row = zeros(1, numVars);
7     row(i_idx(k)) = 1; % Lx(i)
8     row(n + j_idx(k)) = 1; % Ly(j)
9     row(p_idx(k)) = -1; % Lx(p)
10    row(n + q_idx(k)) = -1; % Ly(q)
11    tempA(k, :) = row;
12 end
13
14 A = [A; tempA]; % Concatenate the new constraints
15 b = [b; tempB]; % Concatenate the right-hand side values
```

---

## 5.2.2 Advantages and Disadvantages

### Advantages

- **Speed and Efficiency:** The use of vectorized operations and `meshgrid` significantly reduces the execution time compared to the base implementation and the constraints too are generated in a highly efficient manner, avoiding the nested loop
- **Scalability:** This approach is better suited for larger problems due to its optimized computational complexity

## Disadvantages

- **Complexity and Memory Usage:** The implementation is more complex than the base version and the use of `meshgrid` and large matrices can increase memory consumption, for our work in the order of 384GB of RAM
- **Limit of Intlinprog:** This implementation is not a possible solution for `intlingprog` as it overcomes technical limitations of the solver which shows error *'vector too long'* without a workaround

The implementation in `solveMapping_speed.m` is ideal for medium to large-scale problems where performance is critical.

However for our case study of Posit8 the result produced a model of:

- 127.7 million constraints
- 254 integer variables
- 508.9 million coefficients in the constraints matrix

Unfortunately MATLAB's **intlinprog** solver was not sufficient to solve the resulting matrix as the error message *'Vector Too Long'* was returned. We tried various workarounds but with no solution was found according to the documentation, we also explored the possibility of using other solvers such as IBM's **CPLEX** but that is a paid model and, the trial version was not sufficient in the number of constraints to our work on *Posit(8,0)*.

### 5.3 Parallelized Implementation: solveMapping\_parallel.m

The file `solveMapping_parallel.m` represents a further optimization of the problem, designed to leverage parallel computing using MATLAB's `parfor` construct.

The goal is to distribute the workload across multiple CPU cores, reducing the overall execution time for large-scale problems.

#### 5.3.1 Key Features

##### Parallelization with `parfor`

The primary feature of this implementation is the use of `parfor` to parallelize the outer loop over the variable  $i$ . Each iteration of the loop is executed independently by a different worker, allowing the computation to be distributed across multiple cores.

The code structure is as follows, for simplicity only sum and multiplication functions will be reported:

---

```
1 parfor i = 1:n
2     tempA = []; % Temporary matrix for A (local to each worker)
3     tempB = []; % Temporary matrix for b (local to each worker)
4
5     for j = 1:n
6         if j > i
7             continue; % Skip iterations where j > i
8         end
9         for p = 1:n
10            if p == i
11                continue; % Skip iterations where p == i
12            end
13            for q = 1:n
14                if isequal(op, @times)
15                    if q > p
16                        continue; % Skip iterations where q > p
17                    end
18                    if X(i) * Y(j) < X(p) * Y(q)
19                        row = zeros(1, numVars);
20                        row(i) = 1; % Lx(i)
21                        row(n + j) = 1; % Ly(j)
22                        row(p) = -1; % Lx(p)
23                        row(n + q) = -1; % Ly(q)
24                        if i == p
25                            row(p) = 0;
26                        elseif j == q
27                            row(n + q) = 0;
28                        end
29                        tempA = [tempA; row];
30                        tempB = [tempB; -1];
31                    end
32                elseif isequal(op, @plus)
33                    if q > p
```

```

34         continue; % Skip iterations where q > p
35     end
36     if X(j) + Y(i) < X(p) + Y(q)
37         row = zeros(1, numVars);
38         row(i) = 1; % Lx(i)
39         row(n + j) = 1; % Ly(j)
40         row(p) = -1; % Lx(p)
41         row(n + q) = -1; % Ly(q)
42         if i == p
43             row(p) = 0;
44         elseif j == q
45             row(n + q) = 0;
46         end
47         tempA = [tempA; row];
48         tempB = [tempB; -1];
49     end
50 end
51 end
52 end
53 end
54 end
55
56 % Store the local results in a cell array
57 tempA_cell{i} = tempA;
58 tempB_cell{i} = tempB;
59 end

```

---

### Combining Results from Workers

After the `parfor` loop completes, the results from each worker are combined into the global constraint matrices  $A$  and  $b$ :

```

1 rowIndex = 1;
2 for i = 1:numWorkers
3     if ~isempty(tempA_cell{i})
4         numRows = size(tempA_cell{i}, 1);
5         A(rowIndex:rowIndex + numRows - 1, :) = tempA_cell{i};
6         b(rowIndex:rowIndex + numRows - 1) = tempB_cell{i};
7         rowIndex = rowIndex + numRows;
8     end
9 end

```

---

### Solving the ILP Problem

Once the constraints are assembled, the problem is solved using `intlinprog`, similar to the other implementations:

```

1 [x_opt, fval, exitflag, output] = intlinprog(f, intcon, A, b, Aeq, beq, lb, ub,
        options);

```

---

### 5.3.2 Advantages and Disadvantages

#### Advantages

- **Speed:** The parallelization significantly reduces the execution time, especially for large-scale problems, by distributing the workload across multiple CPU cores
- **Scalability:** This approach is highly scalable and can take full advantage of modern multicore processors
- **Efficiency:** The use of `parfor` allows for efficient utilization of computational resources

#### Disadvantages

- **Complexity:** The implementation is more complex due to the need to manage parallel workers and combine their results
- **Memory Overhead:** Each worker requires its own memory, which can lead to increased memory usage, especially for large  $n$
- **Computing Overhead:** There is a small overhead associated with managing the parallel pool and combining results from different workers

The parallelized implementation in `solveMapping_parallel.m` is ideal for large-scale problems where execution time is critical. By leveraging `parfor`, this approach achieves significant performance improvements, making it suitable for high-performance computing environments. However, it requires careful management of memory and computational resources to avoid excessive overhead.

## 5.4 Comparison of Approaches

The three implementations differ primarily in their computational strategies and performance characteristics:

- **Complexity:** The base implementation (`solveMapping.m`) is the simplest but least efficient, while the parallelized implementation (`solveMapping_parallel.m`) is the most complex but offers the best performance for large problems and of course for our work
- **Scalability:** The base implementation is not suitable for large  $n$  due to its high computational complexity. In contrast, the optimized and parallelized implementations are designed to handle larger problems efficiently
- **Memory Usage:** The base implementation has the lowest memory overhead, while the optimized and parallelized implementations may require more memory due to the use of vectorized operations and parallel workers
- **Flexibility:** All three implementations are flexible and can handle different operations  $\nabla$  (addition, multiplication, subtraction, or division) with minimal modifications

### 5.4.1 Summary of Implementations

The following table provides a high-level comparison of the three implementations, summarizing their key features, advantages, and disadvantages:

Implementation	Key Features	Advantages	Disadvantages
<code>solveMapping.m</code>	Base implementation with nested loops for constraint generation.	Simple, flexible, easy to understand.	Slow for large $n$ , not scalable.
<code>solveMapping_speed.m</code>	Optimized with vectorized operations and <code>meshgrid</code> .	Faster than base implementation, efficient constraint generation.	More complex, higher memory usage, vector too long error.
<code>solveMapping_parallel.m</code>	Parallelized with <code>parfor</code> for distributed computation.	Fastest for large problems, scalable, efficient resource utilization.	Complex to implement, higher memory overhead, requires multicore hardware.

Table 2: Comparison of the three implementations.

## 5.5 Interactive Main Menu

For convenience and to facilitate testing, we have created an interactive menu that allows the user to quickly run the three versions of the developed code. The menu at first guides the user in choosing which implementation to test, making it easier to compare the various proposed solutions. Next the user is asked on which Posit he wants to test the execution between (4,0) (6,0) and (8,0), after that it will be possible to choose the operation among the 4 covered in this document (addition, multiplication, subtraction, or division) and then indicate the name of the JSON file in which to save the result of the execution.

Thanks to this interactive menu the user can quickly test the functionality by being able to run the implementations shown above.

Some screenshots of the main are shown below:

```

Command Window
=== [DFP8 SYSTEM CONSOLE] ===
:: Decoding-free Two-Input Arithmetic
:: Integer Linear Optimization Interface Initialized

:: Choose which Solve Mapping version to run

1. Classic
2. Parallel
3. Speed
4. Exit
Select a mode: 1
Selected mode: classic ---
Choose Posit type:
1. Posit(4,0)
2. Posit(6,0)
3. Posit(8,0)
Choice: 1
Choose the operation:
1. Addition
2. Subtraction
3. Multiplication
4. Division
Choice: 1
Output JSON file name (e.g., result.json): result.json

Command Window
0 (heuristics)

Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerance of the optimal value,
options.AbsoluteGapTolerance = 1e-06. The intcon variables are integer within tolerance, options.ConstraintTolerance = 1e-06.

Solver finished
Mapping Lx (numerator operand):
0 1 2 3 5 7 15

Mapping Ly (operand denominator):
0 1 2 3 5 7 15

Lookup table (Lz -> operation result):
Lz = 0 --> 0.500000
Lz = 1 --> 0.750000
Lz = 2 --> 1.000000
Lz = 3 --> 1.250000
Lz = 4 --> 1.500000
Lz = 5 --> 1.750000
Lz = 6 --> 2.000000
Lz = 7 --> 2.250000
Lz = 8 --> 2.500000
Lz = 9 --> 2.750000
Lz = 10 --> 3.000000
Lz = 12 --> 3.500000
Lz = 14 --> 4.000000
Lz = 15 --> 4.250000
Lz = 16 --> 4.500000
Lz = 17 --> 4.750000
Lz = 18 --> 5.000000
Lz = 20 --> 5.500000
Lz = 22 --> 6.000000
Lz = 30 --> 8.000000
Execution completed. Press any key to continue.

```

Figure 4: Interactive Main Menu



## 6 Experimental Environment

The workloads of our work were run on two separate server machines, which were used sequentially to meet the increasing computational demands and different problems we faced. The first machine is equipped with:

- 2 x Intel Xeon E5645 processor at 2.40 GHz, consisting of 12 physical cores and 24 logical threads
- 96 GB of RAM
- traditional hard disk drive (HDD)

This machine was initially used to run initial tests, but the computational resources proved insufficient to handle the entire workload, especially with memory-intensive operations and parallel computing.

To manage with these critical workload, a second higher-performance server machine was deployed, this is a **liquid-cooled system** with:

- 4 x Intel Xeon Gold 6418H CPUs, consisting of 96 physical cores and 192 logical threads
- 1 TB of RAM
- NVME disk drive

This machine provided a suitable environment for massive parallel execution and processing of the most time-consuming workloads, while maintaining thermal stability and energy efficiency through advanced cooling.

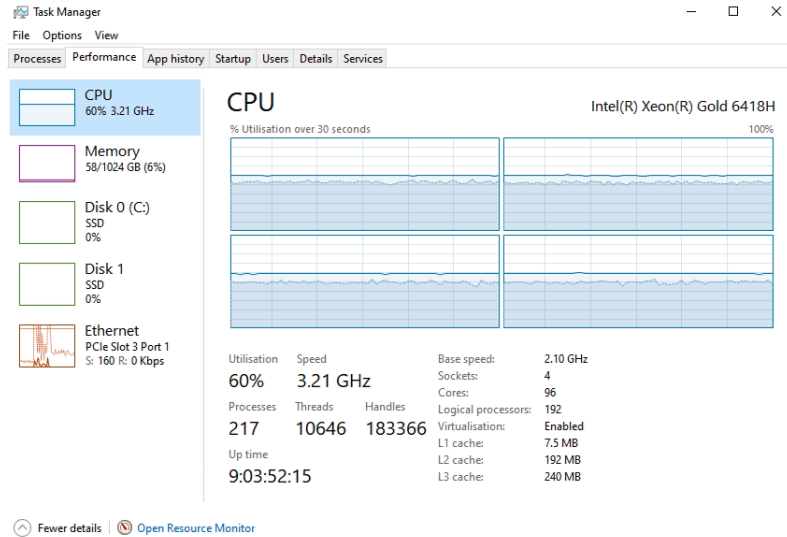


Figure 5: Server resources during execution (NUMA nodes)

## 6.1 Profiling Results and Considerations

The analysis of the execution profiles obtained using the MATLAB profiler for the **minus operation on Posit(8,0)** revealed significant behaviour with regard to the `solveMapping_parallel` function, particularly when using the *parfor* construct.

**Profile Summary**  
Generated 09-Apr-2025 18:30:06 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">eval_new</a>	1	213116.588 s	0.032 s	
<a href="#">solveMapping_parallel</a>	1	213116.553 s	953.479 s	
<a href="#">parallel_function</a>	1	210551.578 s	0.008 s	
<a href="#">parallel_function&gt;distributed_execution</a>	1	210551.530 s	0.020 s	
<a href="#">...Engine&gt;ParforEngine.getCompleteIntervals</a>	16	210551.444 s	210485.685 s	
<a href="#">intlinprog</a>	1	1518.008 s	0.004 s	
<a href="#">...gBranchAndCut&gt;IntlinprogBranchAndCut.run</a>	1	1517.973 s	0.003 s	
<a href="#">slbiClient</a>	1	1517.949 s	1517.590 s	

Figure 6: Profiler Summary Minus Operation in Posit8

The entire execution of this function took over **59 hours**, 98% of which was consumed within the *parfor* loop, with each iteration being computationally intensive.

Function Name	Function Type	Calls			
<a href="#">eval_new</a>	script	1			
Lines where the most time was spent					
Line Number	Code	Calls	Total Time	% Time	Time Plot
<a href="#">84</a>	parfor i = 1:n	1	210551.585 s	98.8%	
<a href="#">252</a>	[x_opt, fval, exitflag, output...	1	1518.009 s	0.7%	
<a href="#">179</a>	A = [A; tempA_cell{i}];	127	907.861 s	0.4%	
<a href="#">75</a>	parpool; % Avvia un pool di wo...	1	93.099 s	0.0%	
<a href="#">208</a>	An = -A;	1	12.914 s	0.0%	
All other lines			33.085 s	0.0%	
Totals			213116.553 s	100%	

Figure 7: Profiler Summary Minus Operation in Posit8

In line with what we expected, the use of *parfor* lead to a proportional reduction in execution time compared to the sequential approach, thanks to

the cores not used by the server in the original version, effectively parallelizing execution and constraint generation. Most of the time is used by *parfor* and, as we expected, the possible cause lies in the nature of MATLAB and its Java implementation of *parfor* for portability, which introduces a synchronization barrier between processes.

This effect is particularly evident in the context of *parfor*, where each worker must complete the assigned iteration before the next one is started, causing bottlenecks even in the presence of abundant computational resources. This can be solved by correctly configuring the *Parallel Computing Profile* and the related **Cluster Profile Manager** options.

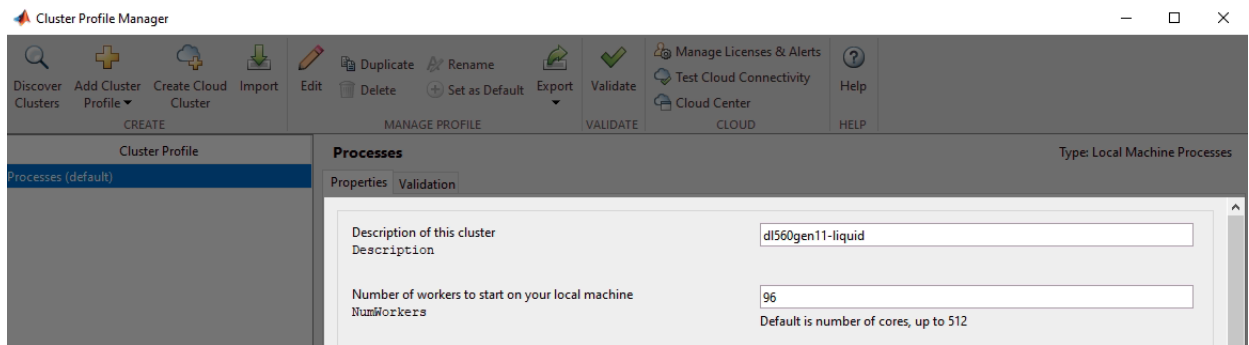


Figure 8: Production Cluster Profile Preferences

In addition, the *gcp* (get current parallel pool) function and its initial operations (setup phase) show negligible time in profiling, indicating that the real overhead comes mainly from synchronizing data and not from creating or managing the parallel pool.

The profiling analysis for the **plus operation Posit(8,0)** showed similar behavior to that found for the minus operation, with only one difference in terms of overall execution time. Again, almost all of the execution time was spent within the *parfor* loop of the `solveMapping_parallel` function, where the overall time exceeded **54.5 hours**, with approximately 99% of it consumed by the row alone that initialises the **parfor**.

Same goes for **rdivide operation** and **times operation**, which took respectively about **61.5 hours** and about **209 hours** to finish its execution and showed the same pattern in the distribution of execution times as the other operations. The table below summarizes the execution times:

Operation on Posit(8,0)	Execution Time (hours)
Minus	59.0
Plus	54.5
Rdivide	61.5
Times	209.0

Table 3: Execution time of operations on Posit(8,0)

Execution profilers for other operations are given below:

**Profile Summary**  
Generated 03-Apr-2025 19:45:51 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">eval_new</a>	1	193619.217 s	0.033 s	
<a href="#">solveMapping_parallel</a>	1	193619.181 s	974.216 s	
<a href="#">parallel_function</a>	1	191277.203 s	0.005 s	
<a href="#">parallel_function&gt;distributed_execution</a>	1	191277.170 s	0.014 s	
<a href="#">...Engine&gt;ParforEngine.getCompleteIntervals</a>	16	191277.068 s	191213.510 s	
<a href="#">intlinprog</a>	1	1294.375 s	0.005 s	
<a href="#">...gBranchAndCut&gt;IntlinprogBranchAndCut.run</a>	1	1294.336 s	0.002 s	
<a href="#">slibClient</a>	1	1294.313 s	1293.961 s	

Figure 9: Profiler Summary Plus Operation in Posit8

Function Name	Function Type	Calls			
<a href="#">eval_new</a>	script	1			
Lines where the most time was spent					
Line Number	Code	Calls	Total Time	% Time	Time Plot
<a href="#">84</a>	parfor i = 1:n	1	191277.207 s	98.8%	
<a href="#">252</a>	[x_opt, fval, exitflag, output...	1	1294.376 s	0.7%	
<a href="#">179</a>	A = [A; tempA_cell{i}];	126	910.397 s	0.5%	
<a href="#">75</a>	parpool; % Avvia un pool di wo...	1	73.060 s	0.0%	
<a href="#">198</a>	Aeq = [Aeq; constrRow];	8001	17.966 s	0.0%	
All other lines			46.176 s	0.0%	
Totals			193619.181 s	100%	

Figure 10: Profiler Summary Plus Operation in Posit8

#### Profile Summary

Generated 06-Apr-2025 23:28:24 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">eval_new</a>	3	220335.631 s	0.052 s	
<a href="#">solveMapping_parallel</a>	3	220335.566 s	826.433 s	
<a href="#">parallel_function</a>	3	215275.369 s	0.051 s	
<a href="#">parallel_function&gt;distributed_execution</a>	3	215275.259 s	0.025 s	
<a href="#">...Engine&gt;ParforEngine.getCompleteIntervals</a>	20	215274.931 s	215202.799 s	
<a href="#">intlinprog</a>	1	4031.605 s	0.004 s	
<a href="#">...gBranchAndCut&gt;IntlinprogBranchAndCut.run</a>	1	4031.571 s	0.002 s	
<a href="#">slbiClient</a>	1	4031.550 s	4031.195 s	

Figure 11: Profiler Summary Rdivide Operation in Posit8

Function Name	Function Type	Calls
<a href="#">eval_new</a>	script	3

**Lines where the most time was spent**

Line Number	Code	Calls	Total Time	% Time	Time Plot
<a href="#">84</a>	parfor i = 1:n	3	215275.375 s	97.7%	
<a href="#">252</a>	[x_opt, fval, exitflag, output...	1	4031.606 s	1.8%	
<a href="#">179</a>	A = [A; tempA_cell{i}];	127	782.136 s	0.4%	
<a href="#">75</a>	parpool; % Avvia un pool di wo...	3	201.163 s	0.1%	
<a href="#">208</a>	An = -A;	1	12.395 s	0.0%	
All other lines			32.890 s	0.0%	
Totals			220335.566 s	100%	

Figure 12: Profiler Summary Rdivide Operation in Posit8

#### Profile Summary

Generated 18-Apr-2025 13:02:10 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">eval_new</a>	1	755298.584 s	0.037 s	
<a href="#">solveMapping_parallel</a>	1	755298.542 s	2811.150 s	
<a href="#">parallel_function</a>	1	745144.775 s	0.005 s	
<a href="#">parallel_function&gt;distributed_execution</a>	1	745144.742 s	0.015 s	
<a href="#">...Engine&gt;ParforEngine.getCompleteIntervals</a>	16	745144.608 s	744976.600 s	
<a href="#">intlinprog</a>	1	7241.405 s	0.004 s	
<a href="#">...gBranchAndCut&gt;IntlinprogBranchAndCut.run</a>	1	7241.375 s	0.002 s	
<a href="#">slbiClient</a>	1	7241.351 s	7240.602 s	

Figure 13: Profiler Summary Times Operation in Posit8



Function Name	Function Type	Calls			
<a href="#">eval_new</a>	script	1			
Lines where the most time was spent					
Line Number	Code	Calls	Total Time	% Time	Time Plot
<a href="#">84</a>	parfor i = 1:n	1	745144.779 s	98.7%	
<a href="#">252</a>	[x_opt, fval, exitflag, output...	1	7241.406 s	1.0%	
<a href="#">179</a>	A = [A; tempA_cell{i}];	127	2699.617 s	0.4%	
<a href="#">75</a>	parpool; % Avvia un pool di wo...	1	100.627 s	0.0%	
<a href="#">208</a>	An = -A;	1	24.311 s	0.0%	
All other lines			87.802 s	0.0%	
Totals			755298.542 s	100%	

Figure 14: Profiler Summary Times Operation in Posit8

This evidence suggests that, although parallel computing correctly distributes iterations across multiple workers, the benefits are severely limited by internal pool management mechanisms in MATLAB. In particular, the presence of synchronization barriers between workers can introduce latency, especially in the presence of iterations that are not homogeneous in terms of execution time. This type of synchronization is managed by a layer in Java, which may introduce additional overhead, especially when the number of workers increases and a multi-socket machine is used.

In summary, the profiling of the plus operation confirms that, the results are similar to those of the minus operation. In heavily parallelized workload contexts, efficiency depends not only on the available hardware resources but also on the design of the MATLAB code and the nature of the parallelism primitives used. However, even in this case, parallelization enabled the execution to be concluded correctly.

## 7 Experimental Result

In this section we continue the work of quality metrics and use the same approach as in the previous work[1], reporting the data and making a comparison for our solutions as well. Specifically, the script `Post_solve.py` is able to take as input the JSON file produced by the Matlab execution and returns as output the logical gates summary for the lookup-tables of each of the operations (sum, multiplication, division and subtraction) on Posit numbers.

### 7.1 Posit (4,0) Implementation Results

This section presents the gate counting results obtained for our solution and related logic gate synthesis for Posit (4,0) formats by comparing them with both the naive implementation and the results of previous work. This served us as a general baseline for comparability between the approaches.

Operation	Total Gates			Grand Total Gates	Naive Total Gates	Gate Reduction
	$L^x$	$L^y$	$L^z$			
+	4	4	10	18	138	$7.67\times$
$\times$	4	4	12	20	138	$6.90\times$
-	4	5	11	20	138	$6.90\times$
/	4	10	13	27	138	$5.11\times$

Table 4: Proposed Method - Total AND-OR Gate Count for Posit(4,0)

Operation	Total Gates			Grand Total Gates	Naive Total Gates	Gate Reduction
	$L^x$	$L^y$	$L^z$			
+	10	10	11	31	138	$4.45\times$
$\times$	7	7	9	23	138	$6.00\times$
-	8	5	5	18	138	$7.67\times$
/	7	7	9	23	138	$6.00\times$

Table 5: Reference Paper - Total AND-OR Gate Count for Posit(4,0)

Operation	Proposed	Paper	Naive
Addition	18	31	138
Subtraction	20	18	138
Multiplication	20	23	138
Division	27	23	138

Table 6: Grand total gate comparison for Posit (4,0)

The results demonstrate comparable gate counts between our implementation and the reference paper (Table 6), with variations attributed to different *ESPRESSO heuristic* outcomes in PyEDA. The maximum variation of  $4.4\text{-}7.6\times$  improvement over naive implementation aligns with previous work.

## 7.2 Posit (8,0) Implementation Results

Below this section presents the gate count results obtained for our solution and related synthesis to logic gates for Posit (8,0) formats, Table 7 represents our proposed method with its counts, Table 8 shows instead the results of the only two operations possible on the reference implementation[2].

Operation	Total Gates			Grand Total Gates	Naive Total Gates	Gate Reduction
	$L^x$	$L^y$	$L^z$			
+	59	59	182	300	15782	$52.61\times$
$\times$	859	859	4	1722	15782	$9.16\times$
-	59	32	137	228	15782	$69.22\times$
/	767	868	7	1642	15661	$9.54\times$

Table 7: Proposed Method - Total AND-OR Gate Count for Posit(8,0)

Operation	Proposed		Reference	
	Total	$L^z$	Total	$L^z$
Addition	300	182	268	96
Multiplication	1722	4	1106	105

Table 8: Comparison with reference implementation (+,  $\times$  only)

Our implementation achieves significant improvements over naive solutions (15782 ports) while maintaining comparability with previous work.



## 8 Future Improvements

Among possible future developments, one of the main objectives will be to extend the presented approach to the **Float8** format, with the intention of comparing its performance and efficiency with respect to the Posit8 representation. This would make it possible to explore the generalizability of the method and assess its effectiveness on alternative numerical formats with reduced precision.

A further improvement concerns solving the optimization problem using advanced solvers. In particular, it is intended to explore the integration of **IBM CPLEX**, which could overcome the *intlinprog* limitations encountered when running the optimized version with *meshgrids*, especially in the Posit(8,0) case, where the number of constraints and variables reached orders of magnitude that could not be handled by the MATLAB solver.

Finally, further optimizations could concern the reduction of execution time through truth table compression techniques and the adoption of more compact mapping strategies, as well as potential direct synthesis in hardware to concretely verify reductions in consumption and circuit complexity.

## 9 Conclusions

Our work extended and optimized a decoding-free approach to arithmetic on real numbers, applied to the Posit(8,0) format, through the formulation of an integer linear optimization problem. Starting from the analysis of the limitations of the previous version, three different approaches for generating the mappings were developed and compared: a basic version, an optimized version and a parallelized version.

The results obtained demonstrate the validity of the method, with a significant reduction in logical complexity (in terms of number of logic gates) compared to the naïve approach. In particular, parallelization proved critical in completing the executions of the four operations, overcoming the computational constraints encountered in the other versions.

Despite of the difficulties related to the scalability of the problem and the limitations of the MATLAB *intlinprog* solver, the method showed good generalizability, allowing the complete implementation of the four fundamental arithmetic operations. In addition, possible directions for further performance improvement emerged, both at the algorithm and hardware implementation level.

This work thus represents a concrete step towards the use of efficient and lightweight arithmetic units, potentially applicable in contexts where resources are limited but precise and compact handling of real numbers is required.

## 10 Bibliography

### References

- [1] John L. Gustafson, Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. *Decoding-free Two-Input Arithmetic for Low-Precision Real Numbers*.
- [2] <https://github.com/unipi-dii-compressedarith/dfarc/tree/conga23>
- [3] <https://github.com/andreaSottile/SEAI-Project-2025>