RANGE CHECK, MAX, MIN, AND RELU

1. INTRODUCTION

In many neural network architectures and computational frameworks, one often encounters simple yet fundamental operations such as range checks, comparisons between values, and elementwise functions like the Rectified Linear Unit (ReLU). Though conceptually elementary, verifying the correctness of these operations within an arithmetic circuit requires careful formulation over a finite field.

This document outlines circuit constructions for three such core operations: verifying that an integer lies within a specified nonnegative range, checking whether a value is the maximum or minimum among a set of integers, and enforcing the correctness of a ReLU activation. Since $\mathrm{ReLU}(a) = \max\{a, 0\}$ it may be viewed as a special case of the maximum function. Likewise, more complex operations such as max pooling and min pooling—common in convolutional neural networks—reduce to repeated applications of max and min over sliding windows. By focusing on the shared underlying structure of these operations, we develop unified and efficient verification strategies suitable for integration into larger circuit-based computations.

2. NONNEGATIVE RANGE CHECK: THE PROCESS

Given an integer a in the range $0 \leqslant a < p$ (where p is the prime modulus of the field over which our arithmetic circuit is defined), we outline a process for verifying that a lies within the narrower range $0 \leqslant a < b^\kappa$, where $b^\kappa \leqslant p$. In fact, the process allows for a slightly more general assumption that a lies in an interval of the form $h - p \leqslant a < h$, where $b^\kappa \leqslant h \leqslant p$, but we consider $h = p$ for simplicity in this introduction.

The key idea is that a is completely determined by its κ least significant base-b digits if and only if $a < b^\kappa$. To enforce this, we impose the constraint

$$a \equiv d_{\kappa-1} b^{\kappa-1} + \cdots + d_0 b^0 \bmod p$$

where each d_j is intended to be a base-b digit. Since we assume $0 \leqslant a < p$ and $b^\kappa \leqslant p$, this congruence lifts to an equality over the integers.

To ensure correctness, we must also verify that each $d_j \in \{0, 1, \ldots, b-1\}$. One way to enforce this is via the polynomial constraint:

$$d_j(d_j - 1)\cdots(d_j - (b-1)) \equiv 0 \bmod p.$$

However, this approach introduces $b-1$ multiplication gates per digit, making it inefficient for $b > 2$. A more efficient alternative is to use a lookup table (LUT) to constrain d_j to the valid digit set.

This method verifies $0 \leqslant a < B$ when $B = b^\kappa$. To verify $a < B$ for arbitrary bounds $B \leqslant p$, one can choose κ such that $b^\kappa \leqslant B$, and verify $a < b^\kappa$ instead. However, may reject valid values of a if $a \in [b^{\kappa-1}, B)$ and $B < b^\kappa$. Greater flexibility would require a more general technique, which we leave as potential future work.

2.1. Steps in the process. Each step in the process has a corresponding explanatory note in Subsection 2.2 that provides additional context and details.

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime.

(2) Choose an integer base $b \geqslant 2$, a nonnegative integer κ, and an integer h, such that

$$b^\kappa \leqslant h \leqslant p. \tag{2.1}$$

(3) Assume integer a lies in the interval of length p with left endpoint $h-p$, which defines a complete set of residues $\bmod p$:

$$h-p \leqslant a < h. \tag{2.2}$$

(4) The goal of the circuit is to verify that, in fact,

$$0 \leqslant a < b^\kappa. \tag{2.3}$$

(5) Most frameworks work exclusively with least residue representations. Accordingly, the prover computes the least residue $\bar{a}$ of $a \bmod p$ and supplies it to the circuit as a (public or private) witness:

$$\bar{a} \equiv a \bmod p, \quad 0 \leqslant \bar{a} < p. \tag{2.4}$$

(6) The prover computes the κ least significant base-b digits of $\bar{a}$ [Proposition 5.4, Algorithm 6.1, Listing 2], and supplies the ordered tuple $(d_{\kappa-1},\dots,d_0)$ to the circuit as a (public or private) witness:

$$\bar{a} = b^\kappa q_\kappa + b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0, \quad q_\kappa \in \mathbb{Z}, \quad d_j \in \{0,1,\dots,b-1\}, \quad 0 \leqslant j < \kappa. \tag{2.5}$$

(7) Impose constraints in the arithmetic circuit [Algorithm 6.3, Listing 4]:

- For $0 \leqslant j < \kappa$, ensure $d_j \equiv c_j \bmod p$ with $c_j \in \{0,1,\dots,b-1\}$, either by enforcing

$$d_j(d_j-1)\cdots(d_j-(b-1)) \equiv 0 \bmod p, \tag{2.6}$$

  or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0,1,\dots,b-1\}$.
- Require [Algorithm 6.3, Listing 4]

$$\bar{a} \equiv b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0 \bmod p. \tag{2.7}$$

(8) Assuming $h-p \leqslant a < h$ in the first place and $b^\kappa \leqslant h \leqslant p$, these constraints guarantee [Proposition 5.2]

$$a = b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0, \tag{2.8}$$

provided each d_j is taken as a least residue, and hence that a lies in the range (2.3).

2.2. Commentary on each step.

(1) Typically, p is an n-bit prime satisfying

$$2^{n-1} \leqslant p < 2^n,$$

with $n \approx 256$. In practice, we use the prime field associated with the scalar field of the BN254 elliptic curve, a 254-bit prime that offers a good balance between security and efficiency. This field is widely supported in cryptographic applications and zk-SNARK frameworks due to the curve's pairing-friendly properties and efficient arithmetic over $\mathbb{Z}/p\mathbb{Z}$.

Polyhedra Network's EXPANDER prover and EXPANDERCOMPILERCOLLECTION (ECC) [1, 2, 3] support both the BN254 prime and the 31-bit Mersenne prime $2^{31}-1$. Although the latter is far too small for cryptographic security on its own, ECC uses internal field extensions to enable meaningful production use over this field, trading off cryptographic hardness for performance and recursion flexibility in certain use cases.

(2) The parameters b, κ, h are chosen in preprocessing and may be regarded as circuit constants.

We need to make sure we don't exclude any valid a in this process. The goal is to verify that $0 \leqslant a < b^\kappa$ (see Step (4)), so it suffices to choose b and κ such that

$$b^{\kappa-1} \leqslant a < b^\kappa. \tag{2.9}$$

The larger b and κ are, the larger the circuit.

The purpose of the offset parameter h is to give the range check more flexibility. Prior to entering the circuit, we typically assume the input a is either:

- nonnegative, and in the least residue range $0 \leqslant a < p$, in which case we choose $h = p$;
- a signed integer in the balanced residue range $-p/2 < a \leqslant p/2$, in which case we choose $h = (p+1)/2$.

In any case, we must be able to justify (2.2) through off-circuit reasoning—see Comment (3).

Assumptions (2.1) and (2.2) are used in crucial ways in Steps (5) and (8): see Comments (5) and (8). They must be strictly adhered to in the circuit setup.

(3) The assumption that a lies in the interval (2.2) is crucial to Step (8): see Comment (8) for details. Since an arithmetic circuit over a field of order p cannot distinguish between integers that differ by a multiple of p, the bounds (2.2) cannot be enforced within the circuit and must instead be justified through off-circuit reasoning.

(4) As we explain below, we will actually show that $0 \leqslant \bar{a} < b^\kappa$, where $\bar{a}$ is the least residue of a mod p. Under our assumptions, this is equivalent to (2.3).

(5) Mathematically, any two integers that differ by a multiple of p are indistinguishable modulo p. In particular, an integer and its least residue representative modulo p can be freely interchanged in any congruence over $\mathbb{Z}/p\mathbb{Z}$. However, arithmetic circuits require inputs to be encoded as canonical representatives in the range $\{0,\dots,p-1\}$. Therefore, any integer intended for use in the circuit must be replaced by its least residue before entering the system.

In our setting, this means the least residue $\bar{a}$ of a mod p must be computed as part of preprocessing. As we're assuming $h-p \leqslant a < h$ (see (2.2)), where $0 \leqslant h \leqslant p$ (implied by assumption (2.1)), we may express the relationship between $\bar{a}$ and a as follows:

$$\bar{a} = \begin{cases} a & \text{if } 0 \leqslant a < h, \\ a+p & \text{if } h-p \leqslant a < 0 \end{cases} \tag{2.10}$$

$$a = \begin{cases} \bar{a} & \text{if } 0 \leqslant \bar{a} < h, \\ \bar{a}-p & \text{if } h \leqslant \bar{a} < p. \end{cases} \tag{2.11}$$

As we're also assuming that $b^\kappa \leqslant h$ (see (2.1)), we have (2.3) if and only if

$$\bar{a} < b^\kappa. \tag{2.12}$$

To see this, note that if $0 \leqslant a < b^\kappa$, then $a \geqslant 0$, and so $\bar{a} = a$ by (2.10). Hence $\bar{a} < b^\kappa$. Conversely, if $\bar{a} < b^\kappa$, then, since $b^\kappa \leqslant h$, we have $\bar{a} < h$, and so $a = \bar{a}$ by (2.11). Hence $0 \leqslant a < b^\kappa$.

Importantly, since showing $0 \leqslant a < b^\kappa$ is equivalent to showing that $0 \leqslant \bar{a} < b^\kappa$, Steps (6) and (7)—which really enforce the latter—do not exclude any valid a.

(6) To check that $0 \leqslant \bar{a} < b^\kappa$, we obtain the κ least significant base-b digits $d_0,\dots,d_{\kappa-1}$ of $\bar{a}$ via repeated division by b. This process also produces a quotient q_κ, which will equal zero if and only if $0 \leqslant \bar{a} < b^\kappa$. In that case, $\bar{a}$ is completely determined by the digits d_j, $0 \leqslant j < \kappa$. This much is true of any integer $\bar{a}$, positive or negative.

(7) For bases $b \geqslant 3$, a lookup table (LUT) is generally the more efficient method for enforcing digit constraints, as compared to multiplicative zero-testing polynomials (2.6).

If $\bar{a} \in [0, b^\kappa)$, then it is determined by its κ base-b digits, i.e. (2.5) holds, and so it satisfies the congruence (2.7), which is equivalent to

$$a \equiv b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0 \bmod p. \tag{2.13}$$

(8) Conversely, suppose (2.7) holds. As this is equivalent to (2.13), there exists an integer t such that

$$a = \left(b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0\right) + tp.$$

If $t \geqslant 1$, this implies $a \geqslant p$. This contradicts assumptions (2.1) and (2.2), which together imply that $a < p$. If $t \leqslant -1$, then

$$a \leqslant (b^\kappa - 1) - p \leqslant (h-1) - p < h - p,$$

since $b^\kappa \leqslant h$ by assumption (2.1). This contradicts assumption (2.2), which asserts that $a \geqslant h - p$.

We conclude that $t = 0$, and hence the desired integer equality (2.8) holds, which in turn yields (2.3).

3. Maxima and Minima: The Process

Let $x, a_0, \dots, a_{\ell-1}$ be integers such that all differences $x - a_i$ all lie in the balanced residue range $(-p/2, p/2]$:

$$-\frac{p}{2} < x - a_i \leqslant \frac{p}{2} \quad \text{for} \quad 0 \leqslant i < \ell, \tag{3.1}$$

where p is the prime modulus of the field over which our arithmetic circuit is defined. We describe how to verify, within the circuit, that

$$x = \max\{a_0,\dots,a_{\ell-1}\}.$$

The claim that x is the maximum of the set $\{a_0,\dots,a_{\ell-1}\}$ is equivalent to the conjunction of:

(1) $x = a_j$ for at least one index $j \in \{0, \ldots, \ell-1\}$, and

(2) $x \geqslant a_i$ for all $i \in \{0, \ldots, \ell-1\}$.

Given the assumption (3.1), Condition (1) holds if and only if $x - a_j \equiv 0 \bmod p$ for some j. Equivalently,

$$(x-a_0)(x-a_1)\cdots(x-a_{\ell-1}) \equiv 0 \bmod p.$$

Condition (2) is equivalent to:

$$0 \leqslant x - a_i \quad \text{for} \quad 0 \leqslant i < \ell.$$

If each difference is known to satisfy $x - a_i < b^\kappa \leqslant p/2$, then we may apply a nonnegative range check as described in Section 2, using the offset $h = (p+1)/2$, to enforce

$$0 \leqslant x - a_i < b^\kappa$$

without excluding any valid value of x.

To verify that

$$x = \min\{a_0, \ldots, a_{\ell-1}\}$$

the same strategy applies, except Condition (2) is replaced by: $x \leqslant a_i$ for all i, which corresponds to checking

$$0 \leqslant a_i - x < b^\kappa$$

for each i, using the same range check procedure.

3.1. Steps in the process. We describe the process for verifying a maximum in detail. We will then describe the changes required to verify a minimum. Each step in the process has a corresponding explanatory note in Subsection 3.2 that provides additional context and details.

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime.

(2) Choose an integer base $b \geqslant 2$ and a nonnegative integer κ such that

$$b^\kappa \leqslant \frac{p}{2}. \tag{3.2}$$

(3) Assume integer $x, a_0, \ldots, a_{\ell-1}$ are such that the differences $x - a_i$ all lie in the balanced residue range $(-p/2, p/2]$:

$$-\frac{p}{2} < x - a_i \leqslant \frac{p}{2} \quad \text{for} \quad 0 \leqslant i < \ell. \tag{3.3}$$

(4) The goal of the circuit is to verify that

$$x = \max\{a_0, \ldots, a_{\ell-1}\}. \tag{3.4}$$

(5) Most frameworks work exclusively with least residue representations. Accordingly, the prover computes the least residues $\bar{x}, \bar{a}_i$ of $x, a_i \bmod p$ and supplies them to the circuit as a (public or private) witnesses:

$$\bar{x} \equiv x \bmod p, \quad \bar{a}_i \equiv a_i \bmod p, \quad 0 \leqslant \bar{x}, \bar{a}_i < p \quad (0 \leqslant i < \ell). \tag{3.5}$$

(6) Within the circuit, compute the least residue $\bar{\delta}_i$ of $x - a_i$, for each i:

$$\bar{\delta}_i \equiv x - a_i \bmod p, \quad 0 \leqslant \bar{\delta}_i < p \quad (0 \leqslant i < \ell). \tag{3.6}$$

(7) Impose the following constraint, which ensures—under assumption (3.3)—that $x = a_j$ for some index j with $0 \leqslant j < \ell$:

$$\bar{\delta}_0 \bar{\delta}_1 \cdots \bar{\delta}_{\ell-1} \equiv 0 \bmod p. \tag{3.7}$$

(8) For each i, perform a range check (as in Section 2) to enforce $\bar{\delta}_i \in [0, b^\kappa)$:

$$0 \leqslant \bar{\delta}_i < b^\kappa \quad (0 \leqslant i < \ell). \tag{3.8}$$

This ensures—under assumption (2.2)—that $x \geqslant a_i$ for each i. The key steps are as follows:

(8a) The prover computes the κ least significant base-b digits of $\bar{\delta}_i$ [Proposition 5.4, Algorithm 6.1, Listing 2], and supplies the ordered tuple $(d_{\kappa-1}^{(i)}, \ldots, d_0^{(i)})$ to the circuit as a (public or private) witness:

$$\bar{\delta}_i = b^\kappa q_\kappa^{(i)} + b^{\kappa-1} d_{\kappa-1}^{(i)} + \cdots + b^0 d_0^{(i)}, \quad q_\kappa^{(i)} \in \mathbb{Z}, \quad d_j^{(i)} \in \{0, 1, \ldots, b-1\}, \quad 0 \leqslant j < \kappa. \tag{3.9}$$

(8b) Impose constraints in the arithmetic circuit [Algorithm 6.3, Listing 4]:

- For $0 \leqslant j < \kappa$, ensure $d_j^{(i)} \equiv c_j^{(i)} \bmod p$ with $c_j^{(i)} \in \{0, 1, \ldots, b-1\}$, either by enforcing

$$d_j^{(i)}(d_j^{(i)} - 1) \cdots (d_j^{(i)} - (b-1)) \equiv 0 \bmod p, \tag{3.10}$$

  or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0, 1, \ldots, b-1\}$.
- Require [Algorithm 6.3, Listing 4]

$$\bar{\delta}_i \equiv b^{\kappa-1} d_{\kappa-1}^{(i)} + \cdots + b^0 d_0^{(i)} \bmod p. \tag{3.11}$$

(8c) Assuming $-p/2 < x - a_i \leqslant p/2$ in the first place and $b^\kappa \leqslant p/2$, these constraints guarantee [Proposition 5.2, $h = (p+1)/2$]

$$x - a_i = b^{\kappa-1} d_{\kappa-1}^{(i)} + \cdots + b^0 d_0^{(i)}, \tag{3.12}$$

provided each $d_j^{(i)}$ is taken as a least residue, and hence that $x - a_i$ lies in the range

$$0 \leqslant x - a_i < b^\kappa.$$

(9) In particular, $x \geqslant a_i$ for each i. Since $x = a_j$ for some j by Step (7), we conclude that (3.4) holds.

To verify that

$$x = \min\{a_0, \ldots, a_{\ell-1}\}, \tag{3.13}$$

simply replace Step (6) by the following:

(6) Within the circuit, compute the least residue $\bar{\delta}_i$ of $a_i - x$, for each i:

$$\bar{\delta}_i \equiv a_i - x \bmod p, \quad 0 \leqslant \bar{\delta}_i < p \quad (0 \leqslant i < \ell). \tag{3.14}$$

That is, use $a_i - x$ instead of $x - a_i$. The range check in Step (8) will then ensure that $x \leqslant a_i$ for each i.

3.2. Commentary on each step.

(1) See Comment (1) of Subsection 2.2.

(2) Step (2) corresponds to Step (2) of the Range Check process (Subsection 2.1) with $h = (p+1)/2$.

The parameters b, κ are chosen in preprocessing and may be regarded as circuit constants.

We need to make sure we don't exclude any valid x in this process. The goal is to verify that $x = \max\{a_0, \ldots, a_{\ell-1}\}$, which entails verifying that $x - a_i \geqslant 0$ for all i. We go further and verify that $0 \leqslant x - a_i < b^\kappa$, so it suffices to choose b and κ such that $x - a_i$ is bounded above by b^κ uniformly for all i, but not much larger—the larger b and κ are, the larger the circuit.

(3) The assumption that each difference $x - a_i$ lies in the interval $(-p/2, p/2]$ is crucial to Step (8c). This holds, for instance, if

$$-\frac{p}{4} < x, a_0, \ldots, a_{\ell-1} \leqslant \frac{p}{4}.$$

In any case, since an arithmetic circuit over a field of order p cannot distinguish between integers that differ by a multiple of p, the bounds (3.3) cannot be enforced within the circuit and must instead be justified through off-circuit reasoning.

(4) Equality (3.4) holds if and only if $x = a_j$ for some j and $x \geqslant a_i$ for all i, $0 \leqslant i, j < \ell$. We verify that $x = a_j$ for some j in Step (7), and that $x \geqslant a_i$ for all i in Step (8).

(5) See Comment (5) in Subsection 2.2 for a general discussion of least residue representations. In particular, most circuit frameworks require all values—whether public or private—to be expressed as least residues modulo p. As a result, even in unconstrained contexts (i.e., outside the circuit), computing the maximum or minimum of a set of integers cannot be performed directly on the input values if they are represented $\bmod p$. Instead, care must be taken to interpret these residues correctly and ensure the computation yields the correct least residue of the true extremum. See Proposition 5.1, Algorithms 6.4 and 6.5, and Listings 5 and 6 for a detailed and correct approach.

(6) Once inside the circuit, subtraction operations such as $\bar{x} - \bar{a}_i$ can be performed using native field arithmetic (e.g., via `api.sub` in ECC [2]), and the result will automatically be interpreted modulo p.

Since $\bar{x} \equiv x \bmod p$ and $\bar{a}_i \equiv a_i \bmod p$, we have $\bar{x} - \bar{a}_i \equiv x - a_i \bmod p$, and so the difference computed in-circuit is congruent to the true difference $x - a_i$. Thus, $\bar{\delta}_i$, which is by definition the least residue of $x - a_i \bmod p$, is given by computing $\bar{x} - \bar{a}_i$ within the circuit.

Under our assumption (3.3), we may express the relationship between $\bar{\delta}_i$ and $x - a_i$ as follows:

$$\bar{\delta}_i = \begin{cases} x - a_i & \text{if } 0 \leqslant x - a_i \leqslant p/2, \\ x - a_i + p & \text{if } -p/2 < x - a_i < 0 \end{cases} \tag{3.15}$$

$$x - a_i = \begin{cases} \bar{\delta}_i & \text{if } 0 \leqslant \bar{\delta}_i \leqslant p/2, \\ \bar{\delta}_i - p & \text{if } p/2 < \bar{\delta}_i < p. \end{cases} \tag{3.16}$$

As we're also assuming that $b^\kappa \leqslant p/2$ (see (3.2)), we have

$$0 \leqslant x - a_i \leqslant b^\kappa$$

if and only if

$$\bar{\delta}_i < b^\kappa$$

To see this, note that if $0 \leqslant x - a_i < b^\kappa$, then $x - a_i \geqslant 0$, and so $\bar{\delta}_i = x - a_i$ by (3.15). Hence $\bar{\delta}_i < b^\kappa$. Conversely, if $\bar{\delta}_i < b^\kappa$, then, since $b^\kappa \leqslant p/2$, we have $\bar{\delta}_i < p/2$, and so $x - a_i = \bar{\delta}_i$ by (3.16). Hence $0 \leqslant x - a_i < b^\kappa$.

Importantly, since showing $0 \leqslant x - a_i < b^\kappa$ is equivalent to showing that $0 \leqslant \bar{\delta}_i < b^\kappa$, Step (8)—which really enforces the latter—does not exclude any valid x.

(7) Since $\mathbb{Z}/p\mathbb{Z}$ is an integral domain, (3.7) implies that $\bar{\delta}_j = 0$ for at least one $j \in \{0, \ldots, \ell - 1\}$. By (3.16), this implies that $x - a_j = 0$, i.e. $x = a_j$.

(8) See Comments (6) – (8) of Subsection 2.1: replace $\bar{a}$ by $\bar{\delta}_i$, a by $x - a_i$, and let $h = (p+1)/2$.

(9) In conclusion, verifying that $x = \max\{a_0, \ldots, a_{\ell-1}\}$ reduces to a polynomial constraint (3.7) and ℓ range checks. The assumptions of Steps (2) and (3) are essential.

4. ReLU

The *Rectified Linear Unit (ReLU)* is one of the most commonly used activation functions in modern neural networks. Defined as

$$\mathrm{ReLU}(a) = \max\{a, 0\}, \tag{4.1}$$

it outputs the input value a when $a \geqslant 0$, and zero otherwise. ReLU introduces non-linearity into the model while maintaining computational simplicity, making it especially attractive in deep learning architectures.

ReLU activations are widely used in convolutional neural networks (CNNs), feedforward networks, and many deep learning frameworks due to their effectiveness in mitigating the vanishing gradient problem and enabling sparse activations. Their simplicity allows for efficient hardware implementation, and their behavior tends to promote better convergence during training. In circuit-based verifications of neural network inference (e.g., in zero-knowledge proofs), ReLU gates must be implemented via constraints that enforce the correct relationship between a and $\max\{a, 0\}$ over a finite field.

Since $x = \mathrm{ReLU}(a)$ is equivalent to $x = \max\{a, 0\}$, a ReLU computation may be verified ing the same procedure outlined in Section 3. This corresponds to the special case where $\ell = 2$, with inputs $a_1 = a$ and $a_1 = 0$.

5. The maths

Proposition 5.1. *Let p be a positive integer (not necessarily prime), and let $x_0, \ldots, x_{n-1}$ be integers lying in an interval of the form*

$$h - p \leqslant x_0, \ldots, x_{n-1} < h,$$

for some integer h. Let

$$M = \max\{x_0, \ldots, x_{n-1}\}.$$

Then the least residue $\bar{M}$ of M modulo p satisfies

$$\bar{M} = \overline{\max\{\overline{x_0 - h}, \ldots, \overline{x_{n-1} - h}\}} + h.$$

Similarly, if

$$m = \min\{x_0, \ldots, x_{n-1}\},$$

then

$$\bar{m} = \overline{\min\{\overline{x_0 - h}, \ldots, \overline{x_{n-1} - h}\} + h}.$$

Proof. For each i, since $h - p \leqslant x_i < h$, we have

$$0 \leqslant x_i - h + p < p \quad \text{and} \quad x_i - h + p \equiv x_i - h \bmod p,$$

that is,

$$\overline{x_i - h} = x_i - h + p.$$

Thus,

$$\begin{aligned} \max\{\overline{x_0 - h}, \ldots, \overline{x_{n-1} - h}\} &= \max\{x_0 - h + p, \ldots, x_{n-1} - h + p\} \\ &= \max\{x_0, \ldots, x_{n-1}\} - h + p \\ &= M - h + p. \end{aligned} \tag{5.1}$$

Since M is the maximum of a set of integers lying in $[h-p, h)$, it also lies in $[h-p, h)$. Therefore,

$$0 \leqslant M - h + p < p \quad \text{and} \quad M - h + p \equiv M - h \bmod p,$$

that is,

$$\overline{M - h} = M - h + p. \tag{5.2}$$

Substituting (5.2) into (5.1) yields

$$\overline{M - h} = \max\{\overline{x_0 - h}, \ldots, \overline{x_{n-1} - h}\}.$$

Consequently,

$$\overline{M - h} + h = \max\{\overline{x_0 - h}, \ldots, \overline{x_{n-1} - h}\} + h.$$

The result for the maximum follows on taking the least residue modulo p of both sides and noting that

$$M \equiv \overline{M - h} + h \bmod p.$$

The proof for the minimum m is entirely analogous. □

Proposition 5.2. *Let p be an integer (not necessarily prime), and let $b \geqslant 2$, $\kappa \geqslant 0$, and h be integers such that*

$$b^\kappa \leqslant h \leqslant p. \tag{5.3}$$

Suppose a is an integer satisfying

$$h - p \leqslant a < h, \tag{5.4}$$

and let $\bar{a}$ denote the least residue of $a \bmod p$. Let $d_0, \ldots, d_{\kappa-1} \in \{0, 1, \ldots, b-1\}$. Then

$$a = b^{\kappa-1} d_{\kappa-1} + \cdots + b^0 d_0 \tag{5.5}$$

if and only if

$$\bar{a} \equiv b^{\kappa-1} d_{\kappa-1} + \cdots + b^0 d_0 \bmod p. \tag{5.6}$$

In particular, if the congruence (5.6) *holds, then*

$$0 \leqslant a < b^\kappa. \tag{5.7}$$

Remark 5.3. (i) Although we may trivially replace $\bar{a}$ with a in (5.6), we intentionally express the congruence in terms of $\bar{a}$ to reflect the circuit-level perspective adopted in Subsection 2.1. Arithmetic circuits operate over the finite field $\mathbb{Z}/p\mathbb{Z}$ and thus only "see" least residue representatives of integers.

(ii) We do not explicitly impose the constraint

$$d_j(d_j - 1) \cdots (d_j - (b-1)) \equiv 0 \bmod p \tag{5.8}$$

in Proposition 5.2. Instead, we assume directly that each $d_j \in \{0, 1, \ldots, b-1\}$. This is because such membership can be enforced efficiently in a circuit via a lookup table (LUT), which avoids the higher multiplicative cost of (5.8).

It is worth noting, however, that if p is prime and d_j satisfies (5.8), then its least residue modulo p must lie in the digit set $\{0, 1, \ldots, b-1\}$, though d_j itself may fall outside that range. Hence, if we assume $0 \leqslant d_j < p$, or explicitly reduce d_j modulo p prior to circuit entry, then the multiplicative condition implies the desired digit constraint. Since we do not rely on (5.8) in the statement of the proposition, we also do not require p to be prime. ■

Proof of Proposition 5.2. If (5.5) holds, then (5.6) follows immediately, since $\bar{a} \equiv a \bmod p$ by definition. Conversely, suppose (5.6) holds. Then, using $\bar{a} \equiv a \bmod p$, we may write

$$a = \left(b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0\right) + tp \tag{5.9}$$

for some integer t. Since $0 \leqslant d_j \leqslant b-1$ for each j, we have

$$0 \leqslant b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0 \leqslant (b-1)(b^{\kappa-1} + \cdots + b^0) = b^\kappa - 1. \tag{5.10}$$

Now recall from assumption (5.4) that $h - p \leqslant a < h$, and from (5.3) that $b^\kappa \leqslant h \leqslant p$. From (5.9) and (5.10), we obtain:

$$a = (\text{value in } [0, b^\kappa - 1]) + tp < h \leqslant p \quad \Longrightarrow \quad tp < p \quad \Longrightarrow \quad t < 1,$$

and also:

$$a = (\text{value in } [0, b^\kappa - 1]) + tp \geqslant h - p \quad \Longrightarrow \quad tp \geqslant h - p - (b^\kappa - 1).$$

Since $b^\kappa \leqslant h$ implies $h - b^\kappa \geqslant 0$, we get

$$tp \geqslant -(p-1) \quad \Longrightarrow \quad t > -1.$$

Hence, t is an integer strictly between -1 and 1, so $t = 0$. Substituting back into (5.9) yields the desired integer equality (5.5). Finally, (5.10) implies that $0 \leqslant a < b^\kappa$, establishing (5.7). □

Proposition 5.4. *Fix an integer base $b \geqslant 2$. Let q_0 be any integer. For $0 \leqslant j < \kappa$, let q_{j+1} and d_j be the unique integers satisfying $q_j = bq_{j+1} + d_j$ and $d_j \in \{0, 1, \ldots, b-1\}$.*

(a) Then

$$q_0 = b^\kappa q_\kappa + b^{\kappa-1}d_{\kappa-1} + b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0. \tag{5.11}$$

(b) The following statements are equivalent: (i) $0 \leqslant q_0 < b^\kappa$; (ii) $q_\kappa = 0$; (iii) $(d_{\kappa-1}, \ldots, d_0)$ is the κ-digit base-b representation of q_0.

Proof. (a) We induct on κ. The result holds trivially for $\kappa = 0$. Suppose the result holds with $\kappa = n$ for some $n \geqslant 0$. Now consider $\kappa = n+1$. We have $q_j = bq_{j+1} + d_j$, $d_j \in \{0, 1, \ldots, b-1\}$ for $0 \leqslant j < n$ and also $j = n$. By inductive hypothesis,

$$\begin{aligned} q_0 &= b^n q_n + b^{n-1}d_{n-1} + b^{n-2}d_{n-2} + \cdots + b^0 d_0 \\ &= b^n(bq_{n+1} + d_n) + b^{n-1}d_{n-1} + b^{n-2}d_{n-2} + \cdots + b^0 d_0 \\ &= b^{n+1}q_{n+1} + b^n d_n + \cdots + b^0 d_0. \end{aligned}$$

(b) Suppose $0 \leqslant q_0 < b^\kappa$. In view of (5.11), this implies

$$b^\kappa q_\kappa = q_0 - \left(b^{\kappa-1}d_{\kappa-1} + b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0\right) \leqslant q_0 < b^\kappa.$$

Hence $q_\kappa < 1$. Also,

$$-b^\kappa q_\kappa = \left(b^{\kappa-1}d_{\kappa-1} + b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0\right) - q_0 \leqslant b^{\kappa-1} + b^{\kappa-2} + \cdots + b^0 < b^\kappa.$$

Hence $q_\kappa > -1$. We must therefore have $q_\kappa = 0$, which implies

$$q_0 = b^{\kappa-1}d_{\kappa-1} + b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0,$$

i.e., $(d_{\kappa-1}, \ldots, d_0)$ is the κ-digit base-b representation of q_0. Finally, if this holds, then $0 \leqslant q_0 < b^\kappa$, because the right-hand side lies between 0 and $(b-1)(b^{\kappa-1} + b^{\kappa-2} + \cdots + b^0) = b^\kappa - 1$. □

6. THE CODE

Rust code in this section is designed for implementation within the EXPANDERCOMPILERCOLLECTION (ECC) framework [3, 4]. This library provides a specialized interface for constructing and verifying arithmetic circuits.

6.1. The κ least significant base-b digits of a nonnegative integer. In order to perform a range check as outlined in Subsection 2.1, it is essential to compute the κ least significant base-b digits of a nonnegative integer. We provide pseudocode for this task, with correctness justified by Proposition 5.4, followed by Rust implementations for both the special case $b = 2$ and the general case $b \geqslant 2$.

Algorithm 6.1 `to_base_b`: compute the κ least significant digits of base-b representation of a nonnegative integer

Require: nonnegative integers q_0 and κ
Ensure: a list d representing the κ least significant base-b digits of q_0

1: $d \leftarrow []$ ▷ Initialize an empty list
2: **for** $i \leftarrow 0$ to $\kappa - 1$ **do**
3: **append** $q_0 \bmod b$ **to** d
4: $q_0 \leftarrow \lfloor q_0/b \rfloor$ ▷ Shift q_0 to the right by one digit
5: **end for**
6: **return** d

```rust
fn to_binary<C: Config>(api: &mut API<C>, q_0: Variable, kappa: usize) -> Vec<Variable> {
    let mut d = Vec::with_capacity(kappa); // Preallocate vector
    let mut q = q_0; // Copy q_0 to modify iteratively

    for _ in 0..kappa {
        d.push(api.unconstrained_bit_and(q, 1)); // Extract least significant bit
        q = api.unconstrained_shift_r(q, 1); // Shift right by 1 bit
    }

    d
}
```

Listing 1: ECC Rust API: compute the κ least significant bits of binary representation of a nonnegative integer

```rust
fn to_base_b<C: Config, Builder: RootAPI<C>>(
    api: &mut Builder,
    q_0: Variable,
    b: u32,
    kappa: u32,
) -> Vec<Variable> {
    let mut d = Vec::with_capacity(kappa as usize); // Preallocate vector
    let mut q = q_0; // Copy q_0 to modify iteratively

    for _ in 0..kappa {
        d.push(api.unconstrained_mod(q, b)); // Extract least significant base-b digit
        q = api.unconstrained_int_div(q, b); // Shift q to remove the extracted digit
    }

    d
}
```

Listing 2: ECC Rust API: compute the κ least significant base-b digits of a nonnegative integer

6.2. Reconstructing an integer from its base-b representation and imposing constraints to ensure correctness. We provide pseudocode for reconstructing a nonnegative integer from its base-b representation. At the same time, we impose constraints to ensure the validity of the representation, with correctness justified by Proposition 5.2. In the pseudocode, each assertion represents a constraint to be enforced within the circuit.

We begin with the special case $b = 2$, where we impose the constraint $r_i(r_i - 1) \equiv 0 \bmod p$ to ensure that each r_i is a valid binary digit. For the general case $b \geqslant 2$, we use a lookup table to enforce that each digit lies in the valid set $\{0, 1, \ldots, b-1\}$. Finally, to complete the range check, we assert that the original integer is equal to its reconstructed value, thereby linking the digit representation to the actual input being verified.

Algorithm 6.2 `from_binary`: reconstruct and verify a nonnegative integer from at most κ least significant bits

Require: list of binary digits `d`, nonnegative integer κ, and original value $\bar{a}$
Ensure: `reconstructed_integer`: the integer represented by the first κ bits of `d`

`reconstructed_integer` $\leftarrow 0$
for $j \leftarrow 0$ to $\max\{\kappa - 1, \text{len}(d) - 1\}$ **do**
    `bit` $\leftarrow d[j]$ ▷ Binary digit check: ensure `bit` $\in \{0, 1\}$
    `bit_minus_one` $\leftarrow 1 -$ `bit`
    `bit_by_bit_minus_one` $\leftarrow$ `bit` $\times$ `bit_minus_one`
    assert `bit_by_bit_minus_one` $= 0$
    `bit_by_two_to_the_j` $\leftarrow$ `bit` $\times 2^j$
    `reconstructed_integer` $\leftarrow$ `reconstructed_integer` $+$ `bit_by_two_to_the_j`
end for
assert `reconstructed_integer` $= \bar{a}$ ▷ Final constraint: confirm correctness of reconstruction
return `reconstructed_integer`

Algorithm 6.3 `from_base_b`: reconstruct and verify a nonnegative integer from at most κ least significant base-b digits

Require: list of base-b digits `d`, nonnegative integer κ, and original value $\bar{a}$
Ensure: `reconstructed_integer`: the integer represented by the first κ base-b digits of `d`

`reconstructed_integer` $\leftarrow 0$
`LOOKUP_TABLE` $\leftarrow \{0, 1, \ldots, b-1\}$ ▷ Predefined valid digit set
for $j \leftarrow 0$ to $\max\{\kappa - 1, \text{len}(d) - 1\}$ **do**
    `digit` $\leftarrow d[j]$
    Enforce `digit` $\in$ `LOOKUP_TABLE` as a circuit constraint
    `digit_by_b_to_the_j` $\leftarrow$ `digit` $\times b^j$
    `reconstructed_integer` $\leftarrow$ `reconstructed_integer` $+$ `digit_by_b_to_the_j`
end for
assert `reconstructed_integer` $= \bar{a}$ ▷ Final constraint: ensure correctness of base-b representation
return `reconstructed_integer`

```rust
fn binary_digit_check<C: Config>(api: &mut API<C>, d: &[Variable]) {
    for &bit in d.iter() {
        let bit_minus_one = api.sub(1, bit);
        let bit_by_bit_minus_one = api.mul(bit, bit_minus_one);
        api.assert_is_zero(bit_by_bit_minus_one);
    }
}

fn from_binary<C: Config>(api: &mut API<C>, d: &[Variable], kappa: usize, a_bar: Variable) ->
    Variable {
    binary_digit_check(api, d);
    let mut reconstructed_integer = api.constant(0);

    for (j, &bit) in d.iter().take(kappa).enumerate() {
        let bit_by_two_to_the_j = api.mul(1 << j, bit);
        reconstructed_integer = api.add(reconstructed_integer, bit_by_two_to_the_j);
    }

    api.assert_is_equal(reconstructed_integer, a_bar);

    reconstructed_integer
}
```

Listing 3: ECC Rust API: reconstruct a nonnegative integer from at most κ least significant bits and impose constraints

The code below integrates ECC's LogUp circuit; see [5] for documentation. Although untested and subject to revision, it provides a working draft to build upon through further experimentation and refinement.

```rust
fn lookup_digit<C: Config, API: RootAPI<C>>(
    api: &mut API,
    digit: Variable,
    lookup_table: &mut LogUpSingleKeyTable
) {
    // Use the lookup table (populated with valid digit constants) to constrain `digit`.
    // The second argument here is the associated value vector, which in this case we assume to be empty.
    lookup_table.query(digit, vec![]);
}

// Check that every digit in the slice `d` is a valid base-b digit using the lookup table.
fn base_b_digit_check<C: Config, API: RootAPI<C>>(
    api: &mut API,
    d: &[Variable],
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable
) {
    for &digit in d.iter() {
        lookup_digit(api, digit, lookup_table);
    }
}

// Reconstruct an integer from the first `kappa` digits in `d` (assumed little-endian)
// and enforce that each digit is a valid base-b digit via the lookup table.
// Finally, assert equality with a given input `a_bar`.
fn from_base_b<C: Config, API: RootAPI<C>>(
    api: &mut API,
    d: &[Variable],
    kappa: usize,
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable,
    a_bar: Variable
) -> Variable {
    base_b_digit_check(api, d, b, lookup_table);

    let mut reconstructed_integer = api.constant(0);
    for (j, &digit) in d.iter().take(kappa).enumerate() {
        let factor = api.constant(b.pow(j as u32));
        let term = api.mul(factor, digit);
        reconstructed_integer = api.add(reconstructed_integer, term);
    }

    api.assert_is_equal(reconstructed_integer, a_bar);

    reconstructed_integer
}
```

Listing 4: ECC Rust API: reconstruct a nonnegative integer from at most κ least significant base-b digits and impose constraints

6.3. The maximum (or minimum) of a set of integers. Following the process described in Section 3, verifying a maximum (or minimum) reduces to imposing a polynomial constraint followed by a series of range checks.

Since the maximum (or minimum) value must be provided by the prover, it must be computed consistently with the field arithmetic used in the circuit. Even in an unconstrained environment, values are represented as least residues modulo p, which complicates direct comparisons between integers. In particular, finding the maximum is not as simple as comparing the integers naively. The algorithm and Rust implementation below illustrate how to correctly compute the least residue of the maximum when all values lie in an interval of the form $[h-p,h)$. For a proof of correctness, see Proposition 5.1.

Algorithm 6.4 `max_with_offset`: compute the least residue modulo p of the maximum of a set of integers

Require: Integers $x_0,\ldots,x_{n-1}$ satisfying $x_i \in [h-p,h)$ for some integer h
Ensure: Returns $\bar{M}$, the least residue modulo p of $M = \max\{x_0,\ldots,x_{n-1}\}$

$M_0 \leftarrow \overline{x_0 - h}$ ▷ Shift into offset domain
for $i \leftarrow 1$ to $n-1$ **do**
  $c \leftarrow \overline{x_i - h}$ ▷ Candidate in offset domain
  if $c > M_0$ **then**
    $M_0 \leftarrow c$
  end if
end for
return $\overline{M_0 + h}$

Algorithm 6.5 `min_with_offset`: compute the least residue modulo p of the minimum of a set of integers

Require: Integers $x_0,\ldots,x_{n-1}$ satisfying $x_i \in [h-p,h)$ for some integer h
Ensure: Returns $\bar{m}$, the least residue modulo p of $m = \min\{x_0,\ldots,x_{n-1}\}$

$m_0 \leftarrow \overline{x_0 - h}$ ▷ Shift into offset domain
for $i \leftarrow 1$ to $n-1$ **do**
  $c \leftarrow \overline{x_i - h}$ ▷ Candidate in offset domain
  if $c < m_0$ **then**
    $m_0 \leftarrow c$
  end if
end for
return $\overline{m_0 + h}$

```rust
pub fn max_with_offset<C: Config, Builder: RootAPI<C>>(
    api: &mut Builder,
    values: Vec<Variable>,
    offset: Variable,
) -> Variable {
    // Shift first value into offset domain
    let mut max_offset = api.unconstrained_sub(values[0], offset);

    for &val in values.iter().skip(1) {
        let candidate = api.unconstrained_sub(val, offset);

        let is_greater = api.unconstrained_greater(candidate, max_offset);
        let not_greater = api.unconstrained_lesser_eq(candidate, max_offset);

        let take_candidate = api.unconstrained_mul(candidate, is_greater);
        let keep_current = api.unconstrained_mul(max_offset, not_greater);

        max_offset = api.unconstrained_add(take_candidate, keep_current);
    }

    // Return max = max_offset + offset
    api.unconstrained_add(max_offset, offset)
}
```

Listing 5: ECC Rust API: compute least residue modulo p of a maximum of a set of integers in unconstrained environment

```rust
pub fn min_with_offset<C: Config, Builder: RootAPI<C>>(
    api: &mut Builder,
    values: Vec<Variable>,
    offset: Variable,
) -> Variable {
    // Shift first value into offset domain
    let mut min_offset = api.unconstrained_sub(values[0], offset);

    for &val in values.iter().skip(1) {
        let candidate = api.unconstrained_sub(val, offset);

        let is_lesser = api.unconstrained_lesser(candidate, min_offset);
        let not_lesser = api.unconstrained_greater_eq(candidate, min_offset);

        let take_candidate = api.unconstrained_mul(candidate, is_lesser);
        let keep_current = api.unconstrained_mul(min_offset, not_lesser);

        min_offset = api.unconstrained_add(take_candidate, keep_current);
    }

    // Return min = min_offset + offset
    api.unconstrained_add(min_offset, offset)
}
```

Listing 6: ECC Rust API: compute least residue modulo p of a minimum of a set of integers in unconstrained environment

Algorithm 6.6 `verify_max`: verify that $x = \max\{a_0, \ldots, a_{\ell-1}\}$

Require: Integers $x, a_0, \ldots, a_{\ell-1}$, base $b \geqslant 2$, digits κ, modulus p
Ensure: Verifies that $x = \max\{a_0, \ldots, a_{\ell-1}\}$

Compute $\bar{\delta}_i \leftarrow x - a_i \bmod p$ for $i = 0$ to $\ell - 1$
$P \leftarrow 1$
for $i \leftarrow 0$ to $\ell - 1$ **do**
    $P \leftarrow P \cdot \bar{\delta}_i$ ▷ Accumulate product
end for
assert $P \equiv 0 \bmod p$ ▷ Ensures $x = a_j$ for some j
for $i \leftarrow 0$ to $\ell - 1$ **do**
    $d_i \leftarrow$ `to_base_b`$(\bar{\delta}_i, b, \kappa)$ ▷ Compute base-b digits
    `from_base_b`$(d_i, \kappa, b, \bar{\delta}_i)$ ▷ Range check: $0 \leqslant \bar{\delta}_i < b^\kappa$
end for

Algorithm 6.7 `verify_min`: verify that $x = \min\{a_0, \ldots, a_{\ell-1}\}$

Require: Integers $x, a_0, \ldots, a_{\ell-1}$, base $b \geqslant 2$, digits κ, modulus p
Ensure: Verifies that $x = \min\{a_0, \ldots, a_{\ell-1}\}$

Compute $\bar{\delta}_i \leftarrow a_i - x \bmod p$ for $i = 0$ to $\ell - 1$
$P \leftarrow 1$
for $i \leftarrow 0$ to $\ell - 1$ **do**
    $P \leftarrow P \cdot \bar{\delta}_i$ ▷ Accumulate product
end for
assert $P \equiv 0 \bmod p$ ▷ Ensures $x = a_j$ for some j
for $i \leftarrow 0$ to $\ell - 1$ **do**
    $d_i \leftarrow$ `to_base_b`$(\bar{\delta}_i, b, \kappa)$ ▷ Compute base-b digits
    `from_base_b`$(d_i, \kappa, b, \bar{\delta}_i)$ ▷ Range check: $0 \leqslant \bar{\delta}_i < b^\kappa$
end for

```rust
/*
Verifies that `x` is either the max or min of the list `[a_0, ..., a_{ell - 1}]`,
depending on the `is_max` flag. Performs a zero-product constraint and
`ell` range checks using base-`b` digits.

# Arguments
 - `api`: the circuit builder
 - `x`: (least residue of) the claimed extremal value
 - `a_vec`: (least residues of) list of candidate values
 - `is_max`: if true, verifies max; if false, verifies min
 - `b`: base for digit decomposition
 - `kappa`: number of digits (i.e., b^kappa is the upper bound)
 - `lookup_table`: lookup table for valid base-b digits
*/
pub fn verify_extreme<C: Config, API: RootAPI<C>>(
    api: &mut API,
    x: Variable,
    a_vec: &[Variable],
    is_max: bool,
    b: u32,
    kappa: usize,
    lookup_table: &mut LogUpSingleKeyTable,
) {
    let mut product = api.constant(1);
    for &a_i in a_vec.iter() {
        // delta_i = x - a_i (for max) or a_i - x (for min)
        let delta_i = if is_max {
            api.sub(x, a_i)
        } else {
            api.sub(a_i, x)
        };

        product = api.mul(product, delta_i); // Enforce x = a_j for some j

        let digits = to_base_b(api, delta_i, b, kappa as u32);
        from_base_b(api, &digits, kappa, b, lookup_table, delta_i);
    }

    api.assert_is_zero(product);
}
```

Listing 7: ECC Rust API: verify correctness of a maximum or minimum of a set of integers

6.4. ReLU of an integer. ReLU is simply a special case of the maximum function, requiring only minor adaptation of the general verification procedure.

Algorithm 6.8 `verify_ReLU`: verify that $x = \mathrm{ReLU}(a) = \max\{a, 0\}$

Require: Integer a, claimed output x, base $b \geqslant 2$, digits κ, modulus p
Ensure: Verifies that $x = \max\{a, 0\}$

Compute $\bar{\delta}_0 \leftarrow x - a \bmod p$
Compute $\bar{\delta}_1 \leftarrow x - 0 = x$
$P \leftarrow \bar{\delta}_0 \cdot \bar{\delta}_1$
assert $P \equiv 0 \bmod p$ ▷ Ensure $x = a$ or $x = 0$
for $i \in \{0, 1\}$ **do**
    $d_i \leftarrow$ `to_base_b`$(\bar{\delta}_i, b, \kappa)$
    `from_base_b`$(d_i, \kappa, b, \bar{\delta}_i)$
end for

```rust
/*
Verifies that `x = ReLU(a) = max(a, 0)` using the same zero-product + range check method
used for `verify_max`.

# Arguments
 - `api`: the circuit builder
 - `a`: (least residue of) the input value to the ReLU function
 - `x`: (least residue of) the claimed output (should equal max(a, 0))
 - `b`: the base for digit decomposition
 - `kappa`: number of digits (i.e., b^kappa is the upper bound)
 - `lookup_table`: lookup table for base-b digit validity
*/
pub fn verify_relu<C: Config, API: RootAPI<C>>(
    api: &mut API,
    a: Variable,
    x: Variable,
    b: u32,
    kappa: usize,
    lookup_table: &mut LogUpSingleKeyTable,
) {
    let delta_0 = api.sub(x, a);              // x - a
    let delta_1 = x;                          // x - 0 = x
    let product = api.mul(delta_0, delta_1); // (x - a)(x - 0)

    api.assert_is_zero(product);              // x equals a or 0

    for &delta in &[delta_0, delta_1] {
        let digits = to_base_b(api, delta, b, kappa as u32);
        from_base_b(api, &digits, kappa, b, lookup_table, delta);
    }
}
```

Listing 8: ECC Rust API: verify correctness of a ReLU computation

References

[1] Polyhedra Network. *Expander: Proof Backend for Layered Circuits*. `https://github.com/PolyhedraZK/Expander`. Accessed May 5, 2025.

[2] Polyhedra Network. *Expander: Proof System Documentation*. `https://docs.polyhedra.network/expander/`. Accessed May 5, 2025.

[3] Polyhedra Network. *ExpanderCompilerCollection: High-Level Circuit Compiler for the Expander Proof System*. `https://github.com/PolyhedraZK/ExpanderCompilerCollection`. Accessed May 5, 2025.

[4] Polyhedra Network. *ExpanderCompilerCollection: Rust Frontend Introduction*. `https://docs.polyhedra.network/expander/rust/intro`. Accessed May 5, 2025.

[5] Polyhedra Network. *LogUp: Lookup-Based Range Proofs in ExpanderCompilerCollection*. `https://docs.polyhedra.network/expander/std/logup`. Accessed May 5, 2025.