RANGE CHECK AND RELU

1. RANGE CHECK FOR NONNEGATIVE INTEGERS: THE PROCESS

Let p be an odd prime, and let $b \geqslant 2$, $\kappa \geqslant 0$, and h be integers satisfying $b^\kappa \leqslant h \leqslant p$. Given an integer a known to lie in the range $h - p \leqslant a < h$, we describe a method to verify that $a \in [0, b^\kappa)$.

Since arithmetic circuits operate over the field $\mathbb{Z}/p\mathbb{Z}$, we work with the least residue of a modulo p, denoted $\bar{a}$. Observe:

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

Since $b^\kappa \leqslant h$, we conclude that $0 \leqslant a < b^\kappa$ if and only if $0 \leqslant \bar{a} < b^\kappa$. Thus, verifying that $\bar{a} < b^\kappa$ suffices to prove the desired range condition on a, under the assumption that $a \in [h - p, h)$.

The key idea is that any nonnegative integer strictly less than b^κ is uniquely determined by its κ least significant base-b digits. Therefore, if the prover can supply such digits $d_0, \ldots, d_{\kappa-1} \in \{0, 1, \ldots, b-1\}$ satisfying

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

then it follows that $\bar{a} < b^\kappa$, and hence $a \in [0, b^\kappa)$.

The circuit enforces this decomposition via the constraint

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

noting that both sides lie in $[0, b^\kappa) \subseteq [0, p)$, so the congruence in fact asserts equality over the integers.

To ensure that each $d_i \in \{0, \ldots, b-1\}$, we may either:

- impose the vanishing polynomial constraint

$$d_i(d_i - 1) \cdots (d_i - (b-1)) \equiv 0 \pmod{p},$$

  which is expensive for large b, or
- verify membership in a precomputed lookup table (LUT) of valid base-b digits.

This method verifies that $a \in [0, B)$ when $B = b^\kappa$. To verify $a \in [0, B)$ for arbitrary $B < p$, one may select b and κ so that $b^\kappa \leqslant B$, and check $a < b^\kappa$. This approach works only when $B \leqslant b^\kappa$; otherwise, it fails for values $a \in [b^{\kappa-1}, B)$ if $B < b^\kappa$. More flexible range checks are possible but beyond the scope of this section.

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

(2) Let $b \geqslant 2$ be an integer base, κ a nonnegative integer, and h an integer satisfying

$$b^\kappa \leqslant h \leqslant p.$$

(3) Assume integer a lies in the range

$$h - p \leqslant a < h.$$

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

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

(5) As most frameworks work exclusively with least residue representations, let $\bar{a}$ denote the least residue of a modulo p:

$$\bar{a} \equiv a \bmod p \quad \text{and} \quad 0 \leqslant \bar{a} < p.$$

The prover supplies $\bar{a}$ to the circuit as a witness.

(6) The prover computes the κ least significant base-b digits $d_{\kappa-1}, \ldots, d_0$ of $\bar{a}$ [Proposition 3.3, Algorithm 4.1, Listing 2] and supplies them to the circuit as a witness:

$$a = b^{\kappa} q_{\kappa} + b^{\kappa-1} d_{\kappa-1} + \cdots + b^0 d_0, \quad q_{\kappa} \in \mathbb{Z}, \quad d_i \in \{0, 1, \ldots, b-1\}, \quad 0 \leqslant i < \kappa.$$

(7) The arithmetic circuit imposes the following constraints [Algorithm 4.3, Listing 4]:

- For $0 \leqslant i < \kappa$, ensure $d_i \equiv c_i \bmod p$ with $c_i \in \{0, 1, \ldots, b-1\}$ *either* by requiring

$$d_i(d_i - 1) \cdots (d_i - (b-1)) \equiv 0 \bmod p,$$

or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0, 1, \ldots, b-1\}$.

- Require [Algorithm 4.3, Listing 4]

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

(8) Assuming $h - p \leqslant a < h$ and $b^{\kappa} \leqslant h \leqslant p$, these constraints guarantee

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

provided each d_i is taken as a least residue, and hence that

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

2. Signed Range Check and ReLU: The Process

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) Let $b \geqslant 2$ be an integer base, κ a nonnegative integer, and h an integer, satisfying

$$b^{\kappa} \leqslant h + (b-1)b^{\kappa-1} \leqslant p. \tag{2.1}$$

(3) Assume integer a lies in the range

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

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

$$-(b-1)b^{\kappa-1} \leqslant a < b^{\kappa-1}, \tag{2.3}$$

and (optionally) compute $\mathrm{ReLU}(a)$.

(5) As most frameworks work exclusively with least residue representations, let $\bar{z}$ denote the least residue modulo p of an integer z. The circuit receives $\bar{a}$ as an input:

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

(6) The circuit computes the least residue $\overline{a + (b-1)b^{\kappa-1}}$ of $a + (b-1)b^{\kappa-1}$ modulo p:

$$\overline{a + (b-1)b^{\kappa-1}} \equiv a + (b-1)b^{\kappa-1} \bmod p \quad \text{and} \quad 0 \leqslant \overline{a + (b-1)b^{\kappa-1}} < p. \tag{2.5}$$

(7) The prover computes the κ least significant base-b digits $d_{\kappa-1}, \ldots, d_0$ of $\overline{a + (b-1)b^{\kappa-1}}$ [Proposition 3.3, Algorithm 4.1, Listing 2], and supplies them to the circuit as a witness:

$$\overline{a + (b-1)b^{\kappa-1}} = b^{\kappa} q_{\kappa} + b^{\kappa-1} d_{\kappa-1} + \cdots + b^0 d_0, \quad q_{\kappa} \in \mathbb{Z}, \quad d_i \in \{0, 1, \ldots, b-1\}, \quad 0 \leqslant i < \kappa. \tag{2.6}$$

(8) The arithmetic circuit imposes the following constraints [Algorithm 4.3, Listing 4]:

- For $0 \leqslant i < \kappa$, ensure $d_i \equiv c_i \bmod p$ with $c_i \in \{0, 1, \ldots, b-1\}$ *either* by requiring

$$d_i(d_i - 1) \cdots (d_i - (b-1)) \equiv 0 \bmod p, \tag{2.7}$$

  or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0, 1, \ldots, b-1\}$.
- Require [Algorithm 4.3, Listing 4; Algorithm 4.5 Step 4, Listing 6]

$$\overline{a + (b-1)b^{\kappa-1}} \equiv b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0 \bmod p. \tag{2.8}$$

(9) Assuming (2.1) and (2.2), these constraints guarantee [Proposition 3.1(a)] that

$$a + (b-1)b^{\kappa-1} = b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0, \tag{2.9}$$

provided each d_i is taken as a least residue, and hence that (2.3) holds. Conversely, if (2.3) holds then so does (2.8) [Proposition 3.1(b)], i.e. no valid a is excluded by the constraints.

(10) The circuit computes [Algorithm 4.5 Steps 5 and 6, Listing 6]

$$\operatorname{sign}(a) = \begin{cases} 1 & \text{if } d_{\kappa-1} = b-1 \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

Finally, the circuit computes

$$\operatorname{ReLU}(a) = \operatorname{sign}(a) \cdot \bar{a} \tag{2.11}$$

using standard multiplication over $\mathbb{Z}/p\mathbb{Z}$.

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, and h are fixed constants of the circuit and are therefore not subject to verification through constraints. Accordingly, the condition (2.1) is assumed to hold by construction.

When lookup tables are not employed in Step (8), it is common to set $b = 2$ to minimize the cost of enforcing digit validity. Nonetheless, there is no inherent restriction preventing one from selecting $\kappa = 1$ and a correspondingly large value of b.

(3) A natural and common choice is $h = (p+1)/2$, in which case the assumption (2.2) simplifies to the balanced interval

$$-\frac{p-1}{2} \leqslant a \leqslant \frac{p-1}{2}. \tag{2.12}$$

This is appropriate when off-circuit reasoning does not support a more specific assumption about the sign or magnitude of a.

The assumption (2.2) is essential and must be justified externally, since the circuit itself can only constrain the least residue representative $\bar{a}$ of $a \bmod p$. To draw conclusions about the original integer a, it is necessary to exclude all other integers congruent to $\bar{a}$ modulo p. Under the assumption (2.2), and provided $0 \leqslant h \leqslant p$ (which is ensured by (2.1)), the relationship between a and $\bar{a}$ is unambiguous:

$$\bar{a} = \begin{cases} a & \text{if } 0 \leqslant a < h, \\ a + p & \text{if } h - p \leqslant a < 0, \end{cases} \quad \text{and} \quad a = \begin{cases} \bar{a} & \text{if } 0 \leqslant \bar{a} < h, \\ \bar{a} - p & \text{if } h \leqslant \bar{a} < p. \end{cases} \tag{2.13}$$

(4) The target interval in (2.3) is of a highly structured form, which limits flexibility but is well-suited for our purposes. In particular, it suffices to determine whether $a < 0$, and thus to compute $\mathrm{ReLU}(a) = \max\{a, 0\}$.

The interval has length b^κ. When $b = 2$, it becomes the symmetric range $[-2^{\kappa-1}, 2^{\kappa-1})$. For larger values of b, the interval becomes increasingly skewed toward negative values, but it still provides a natural generalization of the binary case that supports a clean digit decomposition after a constant shift.

(5) The circuit receives $\bar{a}$ as an input—either computed internally as the output of a preceding subcircuit, or provided directly on an input wire. In either case, $\bar{a}$ denotes the least residue of a modulo p, which is the only representation accessible within the circuit.

The assumption $a \in [h - p, h)$ (see Comment (3)) ensures that $\bar{a}$ uniquely determines a, enabling sound reasoning about the underlying signed integer based solely on its canonical representative in $\mathbb{Z}/p\mathbb{Z}$.

(6) The circuit receives $\bar{a}$ as input. Since $(b-1)b^{\kappa-1}$ is a constant satisfying $(b-1)b^{\kappa-1} < p$ by assumption (2.1), it is already a valid least residue modulo p.

The shifted value

$$\bar{a} + (b-1)b^{\kappa-1}$$

is computed directly within the circuit. Because arithmetic is performed over $\mathbb{Z}/p\mathbb{Z}$, the result is automatically interpreted as the least residue:

$$\overline{\bar{a} + (b-1)b^{\kappa-1}} = \overline{a + (b-1)b^{\kappa-1}}.$$

(7) If a lies within the target range (2.3), then the shifted quantity $a + (b-1)b^{\kappa-1}$ is nonnegative and strictly less than $b^\kappa \leqslant p$. Consequently, its least residue equals the integer itself:

$$\overline{a + (b-1)b^{\kappa-1}} = a + (b-1)b^{\kappa-1}.$$

In this case, the base-b expansion of the shifted value involves no higher-order terms, so $q_\kappa = 0$ in (2.6).

(8) The constraint

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

requires evaluating a degree-b polynomial. Naïvely, this introduces $b-1$ multiplication gates per digit, which quickly becomes expensive when $b \geqslant 3$. As a result, it is generally preferable to use lookup tables (LUTs) for digit validity checks when working in higher bases.

For example, consider verifying $\mathrm{ReLU}(a)$ for an integer a satisfying

$$-2^{21} \leqslant a < 2^{21}.$$

In the binary case ($b = 2$), this interval contains 2^{22} integers, so $\kappa = 22$ bits are required, and the circuit must validate 22 base-2 digits using the digit set $\{0, 1\}$.

In contrast, for a larger base such as $b = 10$, the magnitude $|a|$ satisfies

$$10^6 < 2^{21} < 10^7,$$

so 7 digits suffice to express $|a|$ in base 10. However, since the signed interval we use is of the form

$$[-(b-1)b^{\eta-1},\ b^{\eta-1}),$$

we must set $\eta = 8$ to ensure full coverage. This results in 8 base-10 digits to be validated using the lookup table $\{0, 1, \ldots, 9\}$.

In general, to represent the signed range

$$-2^{\kappa-1} \leqslant a < 2^{\kappa-1}$$

using base b, it suffices to choose η such that

$$\eta - 1 = \left\lceil (\kappa - 1) \cdot \frac{\log 2}{\log b} \right\rceil.$$

This ensures that the interval $[-2^{\kappa-1}, 2^{\kappa-1})$ is contained in $[-(b-1)b^{\eta-1}, b^{\eta-1})$.

While increasing b reduces the number of digits (and hence the number of table lookups), the lookup table itself grows in size from 2 elements to b. Nevertheless, the cost of a lookup constraint typically scales sublinearly with b—both in theory and in practice. Moreover, lookup-based digit validation often requires significantly fewer multiplication gates than algebraic enforcement via a degree-b polynomial. It is therefore worthwhile to experiment with different bases when optimizing circuit size, understood here as the total number of multiplication gates.

(9) The congruence in (2.8) is equivalent to

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

However, we express the left-hand side using the notation $\overline{a+(b-1)b^{\kappa-1}}$ to emphasize that all quantities within the circuit are represented as least residues modulo p.

It is important to clarify that the digit validity constraint (2.7) does not directly enforce $d_i \in \{0,1,\ldots,b-1\}$; rather, it guarantees that $d_i \equiv c_i \bmod p$ for some c_i in that set. Nevertheless, since arithmetic circuits typically represent field elements as canonical representatives in $\{0,\ldots,p-1\}$, this congruence effectively ensures that each d_i lies in the valid digit range. In other words, if $d_i \equiv c_i \bmod p$ for some $c_i \in \{0,\ldots,b-1\}$, then necessarily $d_i = c_i$ in practice.

(10) The ReLU value can be derived within the circuit once the base-b digit decomposition of the shifted quantity $a+(b-1)b^{\kappa-1}$ is verified. Specifically, under the assumption that a lies in the range (2.2) and satisfies the range check (2.3), the sign of a is fully determined by the most significant digit $d_{\kappa-1}$:

$$\operatorname{sign}(a) = \begin{cases} 1 & \text{if } d_{\kappa-1} = b-1, \\ 0 & \text{otherwise.} \end{cases}$$

This is because $d_{\kappa-1} = b-1$ if and only if $a \geqslant 0$, and $d_{\kappa-1} < b-1$ otherwise, given the structure of the signed interval (2.3).

Once $\operatorname{sign}(a)$ is computed (e.g., via a boolean equality test or lookup), the circuit may define

$$\operatorname{ReLU}(a) := \operatorname{sign}(a) \cdot \bar{a}.$$

No additional constraint is needed to enforce correctness of this expression, provided that the digit constraints and base-b congruence

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

hold.

If we wish to prove that $\operatorname{ReLU}(a)$ matches an externally provided value or a public input, we must include an explicit equality constraint. In that case, we assume the provided value lies in $[h-p,h)$.

3. THE MATHS

Proposition 3.1. *Fix an integer modulus $p \geqslant 2$ (not necessarily prime). Let $b \geqslant 2$, $\kappa \geqslant 0$, and h be integers satisfying*

$$b^\kappa \leqslant h+(b-1)b^{\kappa-1} \leqslant p. \tag{3.1}$$

Assume integer a lies in the range

$$h-p \leqslant a < h. \tag{3.2}$$

(a) If there exist integers $d_i \in \{0,1,\ldots,b-1\}$ such that

$$a+(b-1)b^{\kappa-1} \equiv b^{\kappa-1}d_{\kappa-1}+\cdots+b^0 d_0 \bmod p, \tag{3.3}$$

then in fact

$$a+(b-1)b^{\kappa-1} = b^{\kappa-1}d_{\kappa-1}+\cdots+b^0 d_0. \tag{3.4}$$

Consequently, the value of $d_{\kappa-1}$ determines the sign of a:

$$d_{\kappa-1} = b-1 \quad\Longrightarrow\quad 0 \leqslant a < b^{\kappa-1}, \tag{3.5}$$

$$d_{\kappa-1} < b-1 \quad\Longrightarrow\quad -(b-1)b^{\kappa-1} \leqslant a < 0. \tag{3.6}$$

(b) If a lies in the range $[-(b-1)b^{\kappa-1}, b^{\kappa-1})$, then there exist integers $d_i \in \{0,1,\ldots,b-1\}$ such that (3.3) *holds.*

Remark 3.2. Equality (3.4) holds if and only if $(d_{\kappa-1}, d_{\kappa-2}, \ldots, d_0)$ is the κ-digit base-b representation of $a+(b-1)b^{\kappa-1}$. Equivalently, $(b-1-d_{\kappa-1}, d_{\kappa-2}, \ldots, d_0)$ is the κ-digit base-b radix complement representation of a. ■

Proof of Proposition 3.1. If (3.3) holds. Then there exists an integer t such that

$$tp = a+(b-1)b^{\kappa-1} - \left(b^{\kappa-1}d_{\kappa-1}+\cdots+b^0 d_0\right). \tag{3.7}$$

From (3.1) and (3.2), we see that

$$b^\kappa - p \leqslant a+(b-1)b^{\kappa-1} < p. \tag{3.8}$$

Since $0 \leqslant d_i \leqslant b-1$, we also have

$$0 \leqslant b^{\kappa-1}d_{\kappa-1} + \cdots + b^0 d_0 < b^\kappa. \tag{3.9}$$

Combining (3.7), (3.8), and (3.9), we see that

$$-p < tp < p. \tag{3.10}$$

Hence $t = 0$, and (3.4) follows.

Now we establish (3.5) and (3.6):

- If $d_{\kappa-1} = b-1$, then

$$\begin{aligned} a + (b-1)b^{\kappa-1} &= (b-1)b^{\kappa-1} + b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0 \\ &\geqslant (b-1)b^{\kappa-1}. \end{aligned}$$

  Hence

$$a \geqslant 0.$$

  Also,

$$\begin{aligned} a + (b-1)b^{\kappa-1} &= (b-1)b^{\kappa-1} + \left[b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0\right] \\ &\leqslant (b-1)\left(b^{\kappa-1} + \cdots + b^0\right) \\ &= b^\kappa - 1. \end{aligned}$$

  Hence

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

- If $d_{\kappa-1} \leqslant b-2$, then

$$\begin{aligned} a + (b-1)b^{\kappa-1} &\leqslant (b-2)b^{\kappa-1} + \left[b^{\kappa-2}d_{\kappa-2} + \cdots + b^0 d_0\right] \\ &\leqslant (b-2)b^{\kappa-1} + \left[(b-1)(b^{\kappa-1} + \cdots + b^0)\right] \\ &= (b-2)b^{\kappa-1} + \left[b^{\kappa-1} - 1\right] \\ &= (b-1)b^{\kappa-1} - b^{\kappa-1} + b^\kappa - 1 \\ &= (b-1)b^{\kappa-1} - 1. \end{aligned}$$

  Thus,

$$a \leqslant (b-1)b^{\kappa-1} - 1 - (b-1)b^{\kappa-1} = -1.$$

  Also,

$$\begin{aligned} a + (b-1)b^{\kappa-1} &= d_{\kappa-1}b^{\kappa-1} + \cdots + b^0 d_0 \\ &\geqslant 0 \end{aligned}$$

  Hence

$$a \geqslant -(b-1)b^{\kappa-1}.$$

(b) If $a \in \left[-(b-1)b^{\kappa-1}, b^{\kappa-1}\right)$, then $a + (b-1)b^{\kappa-1}$ lies in $[0, b^\kappa)$, and hence has a unique κ-digit base-b representation as in (3.4). This integer equality implies the congruence (3.3). □

Proposition 3.3. *Fix an integer $b > 1$. Let $q_0 \in \mathbb{Z}$. For $0 \leqslant i < \kappa$, let q_{i+1} and a_i be the unique integers satisfying $q_i = bq_{i+1} + a_i$ and $a_i \in \{0, 1, \ldots, b-1\}$.*

(a) Then

$$q_0 = b^\kappa q_\kappa + b^{\kappa-1}a_{\kappa-1} + b^{\kappa-2}a_{\kappa-2} + \cdots + b^0 a_0. \tag{3.11}$$

(b) The following statements are equivalent: (i) $0 \leqslant q_0 < b^\kappa$; (ii) $q_\kappa = 0$; (iii) $(a_{\kappa-1}, \ldots, a_0)$ 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 $q_i = bq_{i+1} + a_i$, $a_i \in \{0, 1, \ldots, b-1\}$ for $0 \leqslant i < n$ and also $i = n$. By inductive hypothesis,

$$\begin{aligned} q_0 &= b^n q_n + b^{n-1}a_{n-1} + b^{n-2}a_{n-2} + \cdots + b^0 a_0 \\ &= b^n(bq_{n+1} + a_n) + b^{n-1}a_{n-1} + b^{n-2}a_{n-2} + \cdots + b^0 a_0 \\ &= b^{n+1}q_{n+1} + b^n a_n + \cdots + b^0 a_0. \end{aligned}$$

(b) Suppose $0 \leqslant q_0 < b^\kappa$. In view of (3.11), this implies

$$b^\kappa q_\kappa = q_0 - \left(b^{\kappa-1}a_{\kappa-1} + b^{\kappa-2}a_{\kappa-2} + \cdots + b^0 a_0\right) \leqslant q_0 < b^\kappa.$$

Hence $q_\kappa < 1$. Also,

$$-b^\kappa q_\kappa = \left(b^{\kappa-1}a_{\kappa-1} + b^{\kappa-2}a_{\kappa-2} + \cdots + b^0 a_0\right) - q_0 \leqslant b^{\kappa-1} + b^{\kappa-2} + \cdots + b^0 < b^\kappa.$$

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

$$q_0 = b^{\kappa-1}a_{\kappa-1} + b^{\kappa-2}a_{\kappa-2} + \cdots + b^0 a_0,$$

i.e., $(a_{\kappa-1}, \ldots, a_0)$ 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 $(b-1)(b^{\kappa-1} + b^{\kappa-2} + \cdots + b^0) = b^\kappa - 1$. □

4. 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.

4.1. The κ least significant base-b digits of a nonnegative integer. Whether performing a range check for nonnegative integers, a signed range check, or verifying a ReLU computation, 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 3.3, followed by Rust implementations for both the special case $b = 2$ and the general case $b \geqslant 2$.

Algorithm 4.1 `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 mod 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

4.2. 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 3.1. 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 $d_i(d_i - 1) \equiv 0 \bmod p$ to ensure that each d_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 4.2 `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 `a`
Ensure: `reconstructed_integer`: the integer represented by the first κ bits of `d`

```
reconstructed_integer ← 0
for i ← 0 to max{κ − 1, len(d) − 1} do
    bit ← d[i]                                   ▷ Binary digit check: ensure bit ∈ {0, 1}
    bit_minus_one ← 1 − bit
    bit_by_bit_minus_one ← bit × bit_minus_one
    assert bit_by_bit_minus_one = 0
    bit_by_two_to_the_i ← bit × 2^i
    reconstructed_integer ← reconstructed_integer + bit_by_two_to_the_i
end for
assert reconstructed_integer = a             ▷ Final constraint: confirm correctness of reconstruction
return reconstructed_integer
```

Algorithm 4.3 `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 `a`
Ensure: `reconstructed_integer`: the integer represented by the first κ base-b digits of `d`

```
reconstructed_integer ← 0
LOOKUP_TABLE ← {0, 1, ..., b − 1}            ▷ Predefined valid digit set
for i ← 0 to max{κ − 1, len(d) − 1} do
    digit ← d[i]
    Enforce digit ∈ LOOKUP_TABLE as a circuit constraint
    digit_by_b_to_the_i ← digit × b^i
    reconstructed_integer ← reconstructed_integer + digit_by_b_to_the_i
end for
assert reconstructed_integer = 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: Variable) -> Variable {
    binary_digit_check(api, d);
    let mut reconstructed_integer = api.constant(0);

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

    api.assert_is_equal(reconstructed_integer, a);

    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`.
fn from_base_b<C: Config, API: RootAPI<C>>(
    api: &mut API,
    d: &[Variable],
    kappa: usize,
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable,
    a: Variable
) -> Variable {
    base_b_digit_check(api, d, b, lookup_table);

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

    api.assert_is_equal(reconstructed_integer, a);

    reconstructed_integer
}
```

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

4.3. Signed range check and ReLU verification. To extend our range check to signed integers, one additional step is required: we shift the input a by $(b-1)b^{\kappa-1}$ to bring the interval $[-(b-1)b^{\kappa-1}, b^{\kappa-1})$ into the nonnegative range $[0, b^{\kappa})$. Since ECC's Rust API (like most circuit frameworks) operates over least residues modulo p, we assume in preprocessing that a is replaced by its least residue $\bar{a}$, where $\bar{a} \equiv a \bmod p$ and $0 \leqslant \bar{a} < p$. We then perform the shift on $\bar{a}$, reduce modulo p, and proceed with the base-b digit decomposition and constraint checks as in the nonnegative case.

For ReLU verification, one further step is needed. After extracting the κ base-b digits of the shifted value, we define a boolean flag $\texttt{sign}(a)$, which is equal to 1 if the most significant digit equals $b-1$, and 0 otherwise. This bit serves as an indicator of whether $a \geqslant 0$, and allows us to compute $\mathrm{ReLU}(a)$ as $\texttt{sign}(a) \cdot \bar{a}$.

We first present the special case $b = 2$, followed by the general case $b \geqslant 2$ using a lookup table for digit validity.

As explained in Comment (10), no additional constraint is needed if $\mathrm{ReLU}(a)$ is computed and used internally within the circuit, as its value is determined from the base-b digit decomposition of the shifted input. In this case, we are not asking the prover to supply $\mathrm{ReLU}(a)$ as a public input.

However, if we wish to prove that the circuit's output matches a public or externally computed value r, we can impose a final constraint:

$$r - \mathrm{sign}(a) \cdot \bar{a} \equiv 0 \mod p.$$

Under the assumption that $r \in [h-p,\ h)$ and $\bar{a} \in [0, h)$, the difference $r - \mathrm{sign}(a) \cdot \bar{a}$ lies in $(-p, h) \subseteq (-p, p)$, and hence the congruence implies actual equality over the integers. This ensures that $\mathrm{ReLU}(a) = r$ holds unambiguously, without confusion from modular equivalence classes.

Algorithm 4.4 `verify_relu`: verify $\mathrm{ReLU}(a)$ in an arithmetic circuit (binary case)

Require: p is an odd prime; $\kappa \geqslant 0$, h are integers such that $2^{\kappa} \leqslant h + 2^{\kappa} \leqslant p$, and a is an integer in $[h-p, h-1]$, replaced by its least residue $\bar{a}$ in preprocessing.
Ensure: $\texttt{relu\_of\_a} = \max\{0, a\}$

▷ Step 1: Replace a with its least residue modulo p
1: $\texttt{lr\_a} \leftarrow a \bmod p$
▷ Step 2: Shift $\texttt{lr\_a}$ by $2^{\kappa-1}$
2: $\texttt{shift} \leftarrow 2^{\kappa-1}$
3: $\texttt{lr\_a\_shifted} \leftarrow (\texttt{lr\_a} + \texttt{shift}) \bmod p$
▷ Step 3: Compute κ unconstrained binary digits of the shifted value
4: $\texttt{bits} \leftarrow \texttt{to\_binary}(\texttt{lr\_a\_shifted}, \texttt{kappa})$
▷ Step 4: Impose bit constraints and ensure reconstruction equals shifted input
5: $\_ \leftarrow \texttt{from\_binary}(\texttt{bits}, \texttt{kappa})$
▷ Step 5: Extract sign bit and compute $\mathrm{ReLU}(a) = \texttt{lr\_a} \times \texttt{sign}(a)$
6: $\texttt{relu\_of\_a} \leftarrow \texttt{lr\_a} \times \texttt{bits}[\texttt{kappa} - 1]$
7: **return** $\texttt{relu\_of\_a}$

Algorithm 4.5 `verify_relu`: verify whether a lies in $[-(b-1)b^{\kappa-1}, b^{\kappa-1})$, then verify $\mathrm{ReLU}(a)$

Require: p is an odd prime; $b \geqslant 2$, $\kappa \geqslant 0$, h are integers such that $b^{\kappa} \leqslant h + (b-1)b^{\kappa} \leqslant p$, and a is an integer in $[h-p, h-1]$, replaced by its least residue $\bar{a}$ in preprocessing.
Ensure: Returns $\mathrm{ReLU}(a) = \texttt{sign\_a} \times \texttt{lr\_a}$

▷ Step 1: Compute the least residue of a modulo p
1: $\texttt{lr\_a} \leftarrow a \bmod p$
▷ Step 2: Shift $\texttt{lr\_a}$ by $(b-1)b^{\kappa-1}$
2: $\texttt{shift} \leftarrow (b-1)b^{\kappa-1}$
3: $\texttt{lr\_a\_shifted} \leftarrow (\texttt{lr\_a} + \texttt{shift}) \bmod p$
▷ Step 3: Compute κ unconstrained base-b digits of the shifted value
4: $\texttt{digits} \leftarrow \texttt{to\_base\_b}(\texttt{lr\_a\_shifted}, \kappa, b)$
▷ Step 4: Enforce digit constraints and reconstruct using lookup table
5: $\_ \leftarrow \texttt{from\_base\_b}(\texttt{digits}, \kappa, b)$ ▷ This enforces digit validity and ensures reconstruction equals $\texttt{lr\_a\_shifted}$
▷ Step 5: Extract sign bit
6: $\texttt{sign\_a} \leftarrow \texttt{is\_equal}(\texttt{digits}[\kappa - 1], b - 1)$
▷ Step 6: Return $\mathrm{ReLU}(a)$
7: **return** $\texttt{sign\_a} \times \texttt{lr\_a}$

```rust
fn verify_relu<C: Config>(api: &mut API<C>, a: Variable, kappa: usize) -> Variable {
    // Step 1: Shift a by 2^(kappa - 1) mod p
    let shift = api.constant(1 << (kappa - 1));
    let a_shifted = api.add(a, shift); // a_shifted = a + 2^(kappa - 1) mod p

    // Step 2: Convert a_shifted to binary (unconstrained)
    let bits = to_binary(api, a_shifted, kappa);

    // Step 3: From binary -> impose bit constraints and check reconstruction
    // (from_binary includes the assertion that reconstructed == a_shifted)
    let _ = from_binary(api, &bits, kappa);

    // Step 4: Compute ReLU(a) = a * bits[kappa - 1] (i.e., sign(a) * a)
    let relu_of_a = api.mul(a, bits[kappa - 1]);

    relu_of_a
}
```

Listing 5: ECC Rust API: verifying $\mathrm{ReLU}(a)$ in the binary case

```rust
// Verifies whether a lies in [-(b - 1)b^(kappa - 1), b^(kappa - 1)) and computes ReLU(a)
fn range_check<C: Config, API: RootAPI<C>>(
    api: &mut API,
    a: Variable,
    kappa: usize,
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable,
) -> Variable {
    // Step 1: Assume a is already in its least residue form modulo p

    // Step 2: Shift a by (b - 1) * b^(kappa - 1)
    let b_minus_one = api.constant(b - 1);
    let b_to_kappa_minus_one = api.constant(b.pow((kappa - 1) as u32));
    let shift = api.mul(b_minus_one, b_to_kappa_minus_one);
    let a_shifted = api.add(a, shift); // a_shifted = a + (b - 1) * b^(kappa - 1) mod p

    // Step 3: Convert a_shifted to a base-b representation with kappa digits (unconstrained)
    let digits = to_base_b(api, a_shifted, kappa, b);

    // Step 4: Enforce digit constraints and reconstruction using lookup table
    // This internally asserts that the digits reconstruct a_shifted
    let _ = from_base_b(api, &digits, kappa, b, lookup_table);

    // Step 5: Compute the sign_a flag for ReLU(a)
    let sign_a = api.is_equal(digits[kappa - 1], b - 1);

    // Step 6: Return ReLU(a) = sign_a * a
    let relu_of_a = api.mul(sign_a, a); // relu_of_a = sign_a * lr_a
    relu_of_a
}
```

Listing 6: ECC Rust API: verify whether a lies in $[-(b-1)b^{\kappa-1}, b^{\kappa-1})$, then verify $\mathrm{ReLU}(a)$

5. EXAMPLES

Example 5.1 (Binary representation)**.** Consider the prime $p = 31$, which is a 5-bit prime ($n = 5$), since

$$2^4 \leqslant 31 < 2^5.$$

We take $b = 2, \kappa = 4$, and $h = (p+1)/2 = 16$, which satisfy $b^\kappa \leqslant h + (b-1)b^{\kappa-1} \leqslant p$. We let a range over the balanced residue interval

$$[h-p, h) \cap \mathbb{Z} = \{-15, \ldots, 15\},$$

and proceed as follows:

- Compute $\bar{a}$, the least residue of a mod p. *Example:* if $a = -15$, then $\bar{a} = 16$.
- Compute $\bar{a} + 2^{\kappa-1}$. *Example:* $\bar{a} + 2^3 = 16 + 8 = 24$.
- Compute $\overline{a + 2^{\kappa-1}}$, the least residue of $a + 2^{\kappa-1}$ modulo p. *Example:* $\overline{a + 2^{\kappa-1}} = 24$.
- Compute $(d_{\kappa-1}, \ldots, d_0)$, the κ least significant bits of $\overline{a + 2^{\kappa-1}}$. *Example:* $\overline{a + 2^{\kappa-1}} = 24$ has binary representation 11000, so
$$(d_3, d_2, d_1, d_0) = (1, 0, 0, 0).$$
Note that we discard the leading bit; only the κ least significant bits are retained.
- Impose the constraints:
  - $d_i(d_i - 1) \equiv 0 \bmod p$ for each i.
  - The reconstruction constraint:
$$a + 2^{\kappa-1} \equiv 2^{\kappa-1} d_{\kappa-1} + \cdots + 2^0 d_0 \bmod p. \qquad (*)$$
Example: The first constraint is satisfied: each $d_i \in \{0, 1\}$. But the right-hand side of $(*)$ is:
$$2^3(1) + 2^2(0) + 2^1(0) + 2^0(0) = 8,$$
while $a + 2^{\kappa-1} \equiv 24 \bmod p$. Hence, the constraint $(*)$ fails.
- Compute the least residue of the right-hand side of $(*)$. *Example:* $8 \bmod 31 = 8$.
- Compare this with $\overline{a + 2^{\kappa-1}}$ to check whether $(*)$ is satisfied. *Example:* $\overline{a + 2^{\kappa-1}} = 24 \neq 8$, so the constraint fails. This tells us that $a = -15$ lies outside the valid range $[-2^{\kappa-1}, 2^{\kappa-1}) = [-8, 8)$.
- Compute $d_{\kappa-1} \cdot \bar{a}$, which equals $\mathrm{ReLU}(a)$ when the constraints are satisfied. *Example:* $d_3 = 1$ and $\bar{a} = 16$, so $d_3 \cdot \bar{a} = 16$. But $\mathrm{ReLU}(a) = \max\{-15, 0\} = 0$. We would not expect equality in this instance, because a did not pass the range check.

The table below shows the results for all $a \in [h-p, h)$. When $(*)$ holds and the constraints are satisfied, the output of $d_{\kappa-1}\bar{a}$ correctly recovers $\mathrm{ReLU}(a)$.

a	$\bar{a}$	$\bar{a}+2^{\kappa-1}$	$\overline{a+2^{\kappa-1}}$	κ LSBs of $\overline{a+2^{\kappa-1}}$	RHS$(*)$	$(*)$ holds	$d_{\kappa-1}\bar{a}$
-15	16	24	24	1000	8	✗	16
-14	17	25	25	1001	9	✗	17
-13	18	26	26	1010	10	✗	18
-12	19	27	27	1011	11	✗	19
-11	20	28	28	1100	12	✗	20
-10	21	29	29	1101	13	✗	21
-9	22	30	30	1110	14	✗	22
-8	23	31	0	0000	0	✓	0
-7	24	32	1	0001	1	✓	0
-6	25	33	2	0010	2	✓	0
-5	26	34	3	0011	3	✓	0
-4	27	35	4	0100	4	✓	0
-3	28	36	5	0101	5	✓	0
-2	29	37	6	0110	6	✓	0
-1	30	38	7	0111	7	✓	0
0	0	8	8	1000	8	✓	0
1	1	9	9	1001	9	✓	1
2	2	10	10	1010	10	✓	2
3	3	11	11	1011	11	✓	3
4	4	12	12	1100	12	✓	4
5	5	13	13	1101	13	✓	5
6	6	14	14	1110	14	✓	6
7	7	15	15	1111	15	✓	7
8	8	16	16	0000	0	✗	0
9	9	17	17	0001	1	✗	0
10	10	18	18	0010	2	✗	0
11	11	19	19	0011	3	✗	0
12	12	20	20	0100	4	✗	0
13	13	21	21	0101	5	✗	0
14	14	22	22	0110	6	✗	0
15	15	23	23	0111	7	✗	0

Table 1: Range check and ReLU verification for $p = 31$, $b = 2$, and $\kappa = 4$.

We also include two additional values outside the assumed range of a. When $a \notin [h-p, h)$, the constraints may fail to hold, revealing that a lies outside the valid range. Even if the constraints are satisfied, the final value $d_{\kappa-1}\bar{a}$ may no longer equal $\mathrm{ReLU}(a)$, since the sign bit no longer reliably encodes the correct comparison.

a	$\bar{a}$	$\bar{a}+2^{\kappa-1}$	$\overline{a+2^{\kappa-1}}$	κ LSBs of $\overline{a+2^{\kappa-1}}$	RHS$(*)$	$(*)$ holds	$d_{\kappa-1}\bar{a}$
-30	1	9	9	1001	9	✓	1
16	16	24	24	1000	8	✗	16

Table 2: Examples where $a \notin [h-p,h)$: constraints may not be satisfied, or they may be while $\mathrm{ReLU}(a) \neq d_{\kappa-1}\bar{a}$.

■

Example 5.2 (Base-3 representation)**.** Consider the prime $p = 37$, base $b = 3$, $\kappa = 3$, $h = (p+1)/2 = 19$, which satisfy $b^\kappa \leqslant h + (b-1)b^{\kappa-1} \leqslant p$. We let a range over the balanced residue interval

$$[h-p,h) \cap \mathbb{Z} = \{-18,\ldots,18\},$$

and proceed as follows:

- Compute $\bar{a}$, the least residue of a mod p. *Example:* if $a = -18$, then $\bar{a} = 19$.
- Compute the shifted value
$$\bar{a} + (b-1)b^{\kappa-1} = \bar{a} + 2 \cdot 3^2 = \bar{a} + 18.$$
Example: $\bar{a} + 18 = 19 + 18 = 37$.
- Compute $\overline{a+18}$, the least residue of $\bar{a}+18$ mod p. *Example:* $\overline{a+18} = 37 \bmod 37 = 0$.
- Compute the κ least significant base-3 digits of $\overline{a+18}$, denoted (d_2,d_1,d_0). *Example:* $\overline{a+18} = 0$ has ternary representation $(0,0,0)$.
- Impose the constraints:
  - For each i, require
$$d_i(d_i-1)(d_i-2) \equiv 0 \bmod p,$$
ensuring $d_i \in \{0,1,2\}$.
  - Enforce the reconstruction constraint
$$\bar{a} + 18 \equiv 3^2 d_2 + 3^1 d_1 + 3^0 d_0 \bmod p. \qquad (*)$$
Example: RHS$(*) = 3^2 \cdot 0 + 3^1 \cdot 0 + 3^0 \cdot 0 = 0$. Since $\bar{a} + 18 = 37$, which is congruent to 0 mod 37, the constraint $(*)$ is satisfied.
- Compute the least residue of the right-hand side of $(*)$. *Example:* $0 \bmod 37 = 0$.
- Compare with $\overline{a+18}$ to determine whether $(*)$ is satisfied. *Example:* $\overline{a+18} = 0$ and RHS$(*) = 0$, so the constraint holds. This confirms that $a = -18$ lies in the valid range $[-18,9)$.
- Compute the sign flag
$$\mathrm{sign}(a) = \begin{cases} 1 & \text{if } d_{\kappa-1} = b-1, \\ 0 & \text{otherwise.} \end{cases}$$
Example: $d_2 = 0 \neq 2$, so $\mathrm{sign}(a) = 0$.
- Finally, compute
$$\mathrm{ReLU}(a) = \mathrm{sign}(a) \cdot \bar{a}.$$
Example: $0 \cdot 19 = 0 = \mathrm{ReLU}(-18)$, as expected.

The table below shows values for all $a \in [h-p,h)$. For each value, it displays whether the constraint $(*)$ is satisfied, and whether the computed value $\mathrm{sign}(a) \cdot \bar{a}$ matches $\mathrm{ReLU}(a)$.

a	$\bar{a}$	$\bar{a}+18$	$\overline{a+18}$	κ LSDs of $\overline{a+18}$	RHS$(*)$	$(*)$ holds	sign(a)	sign$(a)\bar{a}$
-18	19	37	0	000	0	✓	0	0
-17	20	38	1	001	1	✓	0	0
-16	21	39	2	002	2	✓	0	0
-15	22	40	3	010	3	✓	0	0
-14	23	41	4	011	4	✓	0	0
-13	24	42	5	012	5	✓	0	0
-12	25	43	6	020	6	✓	0	0
-11	26	44	7	021	7	✓	0	0
-10	27	45	8	022	8	✓	0	0
-9	28	46	9	100	9	✓	0	0
-8	29	47	10	101	10	✓	0	0
-7	30	48	11	102	11	✓	0	0
-6	31	49	12	110	12	✓	0	0
-5	32	50	13	111	13	✓	0	0
-4	33	51	14	112	14	✓	0	0
-3	34	52	15	120	15	✓	0	0
-2	35	53	16	121	16	✓	0	0
-1	36	54	17	122	17	✓	0	0
0	0	18	18	200	18	✓	1	0
1	1	19	19	201	19	✓	1	1
2	2	20	20	202	20	✓	1	2
3	3	21	21	210	21	✓	1	3
4	4	22	22	211	22	✓	1	4
5	5	23	23	212	23	✓	1	5
6	6	24	24	220	24	✓	1	6
7	7	25	25	221	25	✓	1	7
8	8	26	26	222	26	✓	1	8
9	9	27	27	000	0	✗	0	0
10	10	28	28	001	1	✗	0	0
11	11	29	29	002	2	✗	0	0
12	12	30	30	010	3	✗	0	0
13	13	31	31	011	4	✗	0	0
14	14	32	32	012	5	✗	0	0
15	15	33	33	100	6	✗	0	0
16	16	34	34	101	7	✗	0	0
17	17	35	35	102	8	✗	0	0
18	18	36	36	110	9	✗	0	0

Table 3: Range check and ReLU verification for $p = 37$, $b = 3$, and $\kappa = 3$.

As with the binary case, if the assumption $h - p \leqslant a < h$ does not hold, the result is not guaranteed to be correct. For example, even if the constraints are satisfied, the computed value $\text{sign}(a) \cdot \bar{a}$ may not equal $\text{ReLU}(a)$.

a	$\bar{a}$	$\bar{a}+18$	$\overline{a+18}$	κ LSDs of $\overline{a+18}$	RHS$(*)$	$(*)$ holds	$\text{sign}(a)$	$\text{sign}(a)\bar{a}$
-36	1	19	19	**2**01	19	✓	1	1
19	19	37	0	**0**00	0	✓	0	0

Table 4: Examples where $a \notin [h-p, h)$: constraints may not be satisfied, or they may be while $\text{ReLU}(a) \neq \text{sign}(a) \cdot \bar{a}$.

■

REFERENCES

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

[2] Polyhedra Network. *Expander: Proof System Documentation*. `https://docs.polyhedra.network/expander/`. Accessed June 1, 2025.

[3] Polyhedra Network. *ExpanderCompilerCollection: High-Level Circuit Compiler for the Expander Proof System*. `https://github.com/PolyhedraZK/ExpanderCompilerCollection`. Accessed June 1, 2025.

[4] Polyhedra Network. *ExpanderCompilerCollection: Rust Frontend Introduction*. `https://docs.polyhedra.network/expander/rust/intro`. Accessed June 1, 2025.

[5] Polyhedra Network. *LogUp: Lookup-Based Range Proofs in ExpanderCompilerCollection*. `https://docs.polyhedra.network/expander/std/logup`. Accessed June 1, 2025.