RANGE CHECK AND RELU

1. RANGE CHECK FOR NONNEGATIVE INTEGERS: 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 $b^\kappa \leqslant p$. The key idea is that if a is completely determined by its κ least significant base-b digits, then it must be strictly less than b^κ. To enforce this, we impose the constraint

$$a \equiv r_{\kappa-1}b^{\kappa-1} + \cdots + r_0 b^0 \bmod p$$

where each r_i represents a base-b digit. Since we assume $0 \leqslant a < p$ and $b^\kappa \leqslant p$, this congruence actually enforces equality over the integers rather than just modulo p.

To ensure correctness, we must also verify that each r_i belongs to the set $\{0, 1, \ldots, b-1\}$ (or at least is congruent to one of these values modulo p). One way to enforce this is via the polynomial constraint:

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

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 r_i to the valid digit set.

It is important to note that this method only verifies that $0 \leqslant a < B$ when B is a power of b. If we need to check $0 \leqslant a < B$ for an arbitrary bound $B \leqslant p$, we must choose b and κ such that $b^\kappa \leqslant B$ and then verify $0 \leqslant a < b^\kappa$. However, this fails if $b^{\kappa-1} \leqslant a < B < b^\kappa$. If greater flexibility is required, the process can be extended to implement a more general range check, which we leave as potential future work.

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

$$0 \leqslant a < p.$$

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

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

We wish to verify that $0 \leqslant a < b^\kappa$. To minimize circuit size, κ should be chosen as small as possible: it suffices to choose κ such that $b^{\kappa-1} \leqslant a < b^\kappa$.

(3) Compute the κ least significant base-b digits of a [Proposition 3.3, Algorithm 4.1, Listing 7]:

$$a = b^\kappa q_\kappa + b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0, \quad q_\kappa \in \mathbb{Z}, \quad r_i \in \{0, 1, \ldots, b-1\}, \quad 0 \leqslant i < \kappa.$$

(4) Impose constraints in the arithmetic circuit [Algorithm 4.3, Listing 9]:

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

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

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

- Require [Algorithm 4.3, Listing 9]

$$a \equiv b^{\kappa-1} r_{\kappa-1} + \cdots + b^0 r_0 \bmod p.$$

(5) Assuming $0 \leqslant a < p$ in the first place and $b^\kappa \leqslant p$, these constraints guarantee [Proposition 3.1(b)]

$$a = b^{\kappa-1} r_{\kappa-1} + \cdots + b^0 r_0,$$

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

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

2. Signed Range Check and ReLU: The Process

Given an integer a in the range $-p/2 < a \leqslant p/2$ (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

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

where $\kappa \leqslant n-1$ and

$$2(b-1)b^{n-2} < p < b^n.$$

This may seem strange at first, but it is a natural generalization of the case $b = 2$, in which we verify that

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

under the assumption $\kappa \leqslant n-1$, where p is an n-bit prime satisfying

$$2^{n-1} < p < 2^n.$$

We begin by discussing this special case. Since arithmetic circuits operate on least residues modulo p, we must translate the desired signed inequality into one that applies to elements in the interval $[0, p)$. Observe that

$$-2^{\kappa-1} \leqslant a < 2^{\kappa-1} \quad \Longleftrightarrow \quad 0 \leqslant a + 2^{\kappa-1} < 2^\kappa.$$

Assuming $2^\kappa \leqslant p$, we can then perform a range check, following the method described in Section 1, on the shifted value $a + 2^{\kappa-1}$. If $(r_{\kappa-1}, \ldots, r_0)$ is the binary representation of $a + 2^{\kappa-1}$, i.e.,

$$a + 2^{\kappa-1} = 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0,$$

with each $r_i \in \{0, 1\}$, then $a \geqslant 0$, or equivalently, $a + 2^{\kappa-1} \geqslant 2^{\kappa-1}$, if and only if $r_{\kappa-1} = 1$. This tells us that

$$\operatorname{ReLU}(a) = r_{\kappa-1} a,$$

or equivalently (since $-p/2 \leqslant a < p/2$),

$$\operatorname{ReLU}(a) = r_{\kappa-1} r,$$

where r is the least residue of a modulo p.

In general, the interval

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

has length b^κ, but is not symmetric about 0 when $b \geqslant 3$. This asymmetry is not an issue if our main goal is simply to determine whether $a < 0$, and hence whether $\operatorname{ReLU}(a)$ equals 0 or a. As in the binary case, we observe that

$$-(b-1)b^{\kappa-1} \leqslant a < b^{\kappa-1} \quad \Longleftrightarrow \quad 0 \leqslant a + (b-1)b^{\kappa-1} < b^\kappa.$$

Once again, we perform a range check on the shifted value $a + (b-1)b^{\kappa-1}$ (or, more precisely, its least residue modulo p), assuming $b^\kappa \leqslant p$. However, because we assume that a lies in the balanced residue range $[-p/2, p/2)$, we need to ensure that congruences modulo p correspond to actual integer equalities. This motivates the assumption that

$$2(b-1)b^{n-2} < p < b^n,$$

with $\kappa \leqslant n-1$. See Proposition 3.1 for the justification.

Finally, suppose

$$a + (b-1)b^{\kappa-1} = r_{\kappa-1} b^{\kappa-1} + \cdots + r_0 b^0,$$

where each $r_i \in \{0, 1, \ldots, b-1\}$. Then $a \geqslant 0$, or equivalently, $a + (b-1)b^{\kappa-1} \geqslant (b-1)b^{\kappa-1}$, if and only if $r_{\kappa-1} = b-1$. This tells us that

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

which is tantamount to computing $\operatorname{ReLU}(a)$.

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. Let $b \geqslant 2$ be an integer base satisfying

$$2(b-1)b^{n-2} < p < b^n \tag{2.1}$$

for some integer $n \geqslant 2$.

(2) Assume integer a lies in the range

$$-p/2 < a \leqslant p/2.$$

As most frameworks work exclusively with least residue representations, we define r as the least residue of a modulo p:

$$a \equiv r \bmod p, \quad 0 \leqslant r < p.$$

(3) Fix $\kappa \leqslant n-1$. We wish to show that a lies in the interval $\left[-(b-1)b^{\kappa-1}, b^{\kappa-1}\right)$ of length b^κ. To minimize circuit size, κ should be just large enough so that this interval contains a.

(4) Compute the least residue $r^\sharp$ of $r+(b-1)b^{\kappa-1}$ modulo p, i.e. [Algorithm 4.5 Step 2, Listing 11]

$$r^\sharp \equiv r + (b-1)b^{\kappa-1} \bmod p, \quad 0 \leqslant r^\sharp < p.$$

(5) Compute the κ least significant base-b digits of $r^\sharp$ [Proposition 3.3, Algorithm 4.1, Listing 7]:

$$r^\sharp = b^\kappa q_\kappa + b^{\kappa-1} r_{\kappa-1} + \cdots + b^0 r_0, \quad q_\kappa \in \mathbb{Z}, \quad r_i \in \{0,1,\ldots,b-1\}, \quad 0 \leqslant i < \kappa.$$

(6) Impose constraints in the arithmetic circuit [Algorithm 4.3, Listing 9]:

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

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

  or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0,1,\ldots,b-1\}$.
- Require [Algorithm 4.3, Listing 9; Algorithm 4.5 Step 4, Listing 11]

  $$r^\sharp \equiv b^{\kappa-1} r_{\kappa-1} + \cdots + b^0 r_0 \bmod p.$$

(7) Assuming $-p/2 < a \leqslant p/2$ in the first place and $\kappa \leqslant n-1$, these constraints guarantee [Proposition 3.1(c)]

$$a + (b-1)b^{\kappa-1} = b^{\kappa-1} r_{\kappa-1} + \cdots + b^0 r_0,$$

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

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

(8) Finally, compute [Algorithm 4.5 Steps 5 and 6, Listing 11]

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

Consequently,

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

2.2. Commentary on each step.

(1) For $b = 2$, (2.1) simplifies to $2^{n-1} < p < 2^n$. More generally, since $2 \leqslant b \leqslant 2(b-1)$, it follows from (2.1) that

$$b^{n-1} < p < b^n.$$

Thus, p requires exactly n base-b digits for its representation, and no fewer. Moreover, if such an n exists, it is unique.

However, for $b \geqslant 3$, the strict inequality $b^{n-1} < 2(b-1)b^{n-2}$ holds for all $n \geqslant 2$. This means that the intervals $[2(b-1)b^{n-2}, b^{n-1})$, for $n \geqslant 2$, do not partition the positive integers into disjoint intervals indexed by n. Consequently, for an arbitrary prime p and base $b \geqslant 3$, there may be no integer n satisfying (2.1), as it is possible that $b^{n-1} < p < 2(b-1)b^{n-2}$ for some n.

Since p is fixed in advance, we define b to be *admissible* with respect to p if there exists an integer $n \geqslant 2$ such that (2.1) holds, and *inadmissible* otherwise. Listing 1 provides Python code for computing the list of inadmissible bases b, for $2 \leqslant b \leqslant B_{\max}$. Additionally, given an admissible b, the code determines the unique n satisfying (2.1).

Listings 2 – 5 apply these functions to two cases: the 31st Mersenne prime, $2^{31}-1$, and the prime that defines the order of the scalar field of the BN254 curve. In both cases, base $b = 10$ is admissible. Specifically, we find that $n = 10$ for the Mersenne prime $2^{31}-1$, and $n = 77$ for the BN254 prime.

(2) The assumption that a lies in the balanced residue range, i.e. $-p/2 < a \leqslant p/2$, is a hypothesis of Proposition 3.1(c), which we invoke in Step (7). Since an arithmetic circuit over a field of order p cannot distinguish between integers that differ by a multiple of p, this assumption cannot be enforced within the circuit and must instead be justified through off-circuit reasoning. If it fails to hold, the main conclusions of Steps (7) and (8) may not follow—see the end of each example in Section 5.

Since arithmetic circuit wires are represented by canonical elements in $\{0,\ldots,p-1\}$, we must translate conditions on a into equivalent statements about its least residue representative r, and vice versa. This is particularly relevant when performing signed comparisons or encoding operations. Mathematically, a and r are interchangeable in any congruence modulo p.

(3) The assumption that $\kappa \leqslant n-1$ means that integers cannot have multiple base-b representations modulo p. There are b^κ base-b strings of length κ, and we want this to be less than p. For example, in base 3, $0 \equiv 3^4(1) + 3^0(2) \bmod 83$, so 00000 and 10002 both represent 0 modulo 83 (which can only be represented with 5 base-3 digits). With multiple base$-b$ representations of an integer modulo p, the constraints imposed in Step (6) do not imply that a lies in the desired range or that $\mathrm{ReLU}(a)$ is correct.

(4) Since arithmetic circuit wires are represented by canonical elements in $\{0,\ldots,p-1\}$, we must take the least residue $r^\sharp$ of $r + (b-1)b^{\kappa-1}$ modulo p. Since r is itself the least residue of a modulo p,

$$r^\sharp \equiv a + (b-1)b^{\kappa-1} \bmod p.$$

(5) If a is in the expected range, i.e. $-(b-1)b^{\kappa-1} \leqslant a < b^{\kappa-1}$, then $r^\sharp = a + (b-1)b^{\kappa-1}$, and $q_\kappa = 0$.

(6) The constraint

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

introduces b multiplication gates to the circuit for each of the κ base-b digits, so choosing $b \geqslant 3$ is unlikely to be advantageous over $b = 2$ unless look up tables are used instead.

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

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

In the binary case ($b = 2$), representing numbers in this range requires $\kappa = 22$ bits (since $-2^{21} \leqslant a < 2^{21}$ has 2^{22} distinct values), meaning that 22 table lookups are needed with the digit set $\{0,1\}$.

In contrast, if we choose a larger basis, say $b = 10$, we need fewer digits to cover the same range. Since

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

we require 7 digits for the magnitude. However, because the signed range is represented as

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

we must choose $\eta = 8$, which means 8 table lookups in the lookup table $\{0,1,\ldots,9\}$.

In general, to represent the range

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

using base b, one can choose

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

so that the interval $[-2^{\kappa-1}, 2^{\kappa-1})$ is contained within $[-(b-1)b^{\eta-1}, b^{\eta-1})$. Thus, while a larger basis reduces the number of table lookups from κ to η, the trade-off is that the lookup tables themselves become larger (containing b elements instead of 2).

While larger bases b reduce the number of digits (and hence the number of table lookups), the cost of each lookup generally scales *sub-linearly* in b—both in theory and in practice. Since lookup constraints typically require fewer multiplication gates than enforcing digit validity algebraically (e.g., via a degree-b polynomial), it may be worthwhile to experiment with different bases when optimizing *circuit size*, understood here as the total number of multiplication gates.

(7) The congruence

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

does not imply that $r_i \in \{0,1,\ldots,b-1\}$: it only implies that $r_i \equiv c_i \bmod p$ for some $c_i \in \{0,1,\ldots,b-1\}$. However, in practice, wires in an arithmetic circuit are represented by canonical elements in $\{0,\ldots,p-1\}$. Hence, if $r_i \equiv c_i$ with $c_i \in \{0,1,\ldots,b-1\}$, we indeed get $r_i \in \{0,1,\ldots,b-1\}$.

(8) No additional constraint is required to ensure that $\mathrm{ReLU}(a) = \mathrm{sign}(a)\cdot r$, where r is the least residue of a modulo p. This follows from the base-b decomposition: the most significant digit $r_{\kappa-1}$ must equal $b-1$ if and only if $a \geqslant 0$, and must be strictly less than $b-1$ otherwise. These conditions are enforced by the digit constraints and the congruence

$$r^{\sharp} \equiv r_{\kappa-1}b^{\kappa-1} + \cdots + r_0 b^0 \bmod p,$$

which ensures that the base-b digits are consistent with the shifted version of a.

If, however, we wish to prove that $\mathrm{ReLU}(a)$ matches a *public input* or an externally provided value, we must include an explicit equality constraint. In that case, we also assume that the external value lies in the range $[0,p)$.

3. The maths

Proposition 3.1. *Fix an integer modulus $p \geqslant 2$ (not necessarily prime), and let $b \geqslant 2$ be an integer base. Suppose we express integers in base b using at most κ digits, where*

$$b^{\kappa} \leqslant p \tag{3.1}$$

For $0 \leqslant i \leqslant \kappa-1$, let the digits $r_i \in \{0,1,\ldots,b-1\}$ be given, and for any integer x denote by r_x its least residue modulo p.

(a) The congruence

$$x \equiv b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 \bmod p \tag{3.2}$$

holds if and only if

$$r_x = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0. \tag{3.3}$$

(b) If (3.2) holds and $0 \leqslant x \leqslant p-1$, then

$$x = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0. \tag{3.4}$$

(c) Assume there exists an integer $n \geqslant 2$ such that

$$2(b-1)b^{n-2} < p < b^n, \tag{3.5}$$

and suppose further that $\kappa \leqslant n-1$. Let a be any integer satisfying

$$-\frac{p}{2} < a \leqslant \frac{p}{2}. \tag{3.6}$$

If

$$a + (b-1)b^{\kappa-1} \equiv b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 \bmod p, \tag{3.7}$$

then

$$a + (b-1)b^{\kappa-1} = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0. \tag{3.8}$$

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

$$r_{\kappa-1} = b-1 \implies 0 \leqslant a < b^{\kappa-1}, \tag{3.9}$$

$$r_{\kappa-1} < b-1 \implies -(b-1)b^{\kappa-1} \leqslant a < 0. \tag{3.10}$$

Remark 3.2. Equality (3.4) holds if and only if $(r_{\kappa-1}, r_{\kappa-2}, \ldots, r_0)$ is the κ-digit base-b representation of x. Equality (3.8) holds if and only if $(b-1-r_{\kappa-1}, r_{\kappa-2}, \ldots, r_0)$ is the κ-digit base-b radix complement representation of a. ■

Proof of Proposition 3.1. (a) Suppose the integer equality (3.3) holds. Then, since $x \equiv r_x \bmod p$, (3.2) holds. Conversely, suppose (3.2) holds. We may replace x by r_x in (3.2). Thus, there exists an integer t such that

$$r_x = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp. \tag{3.11}$$

Since r_x is a least residue modulo p, we have the bounds

$$0 \leqslant r_x \leqslant p-1. \tag{3.12}$$

Moreover, as $0 \leqslant r_i \leqslant b-1$ for $0 \leqslant i < \kappa$, we obtain

$$0 \leqslant b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 \leqslant (b-1)(b^{\kappa-1} + \cdots + b^0) \leqslant b^{\kappa} - 1 \leqslant p-1, \tag{3.13}$$

where the last inequality holds by (3.1). Applying the left-most inequality in (3.13) and the right inequality in (3.12) to (3.11), we get

$$0 + tp \leqslant b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp = r_x \leqslant p-1 \quad \Longrightarrow \quad t \leqslant \frac{p-1}{p} < 1. \tag{3.14}$$

Similarly, using the right-most inequality in (3.13) and the left inequality in (3.12), we obtain

$$0 \leqslant r_x = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp \leqslant p-1+tp \quad \Longrightarrow \quad t \geqslant -\frac{p-1}{p} > -1. \tag{3.15}$$

Since t is an integer satisfying $-1 < t < 1$, it must be $t = 0$. Substituting into (3.11) yields (3.3).

(b) This follows immediately from part (a) upon noting that if $0 \leqslant x \leqslant p-1$, then $x = r_x$.

(c) First, note that since $b \geqslant 2$, $b \leqslant 2(b-1)$, and hence

$$2(b-1)b^{n-2} \leqslant bb^{n-2} = b^{n-1}.$$

Thus, (3.5) implies that

$$b^{n-1} < p < b^n, \tag{3.16}$$

which, as we're assuming now that $\kappa \leqslant n-1$, implies (3.1). Thus, we may use (a) and the bounds in its proof.

Assume (3.7) holds (which is just (3.2) with $x = a + (b-1)b^{\kappa-1}$). Then there exists an integer t such that

$$a + (b-1)b^{\kappa-1} = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp. \tag{3.17}$$

Using the left inequality in (3.13), (3.17), the assumption $\kappa \leqslant n-1$, the left inequality in (3.5), and the right inequality in (3.6), we obtain

$$0 + tp \leqslant b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp = a + (b-1)b^{\kappa-1} \leqslant a + (b-1)b^{n-2} < \frac{p}{2} + \frac{p}{2} = p \quad \Longrightarrow \quad t < \frac{p}{p} = 1. \tag{3.18}$$

Similarly, using (3.17) and the right inequality in (3.13), we obtain

$$a + b^{\kappa} - b^{\kappa-1} = a + (b-1)b^{\kappa-1} = b^{\kappa-1}r_{\kappa-1} + \cdots + b^0 r_0 + tp \leqslant b^{\kappa} - 1 + tp,$$

which, together with the left inequality in (3.6), the assumption $\kappa \leqslant n-1$, and the left inequality in (3.16) (which implies $b^{n-2} < p/b \leqslant p/2$), yields

$$-\frac{p}{2} < a \leqslant b^{\kappa-1} - 1 + tp \leqslant b^{n-2} - 1 + tp < \frac{p}{2} - 1 + tp \quad \Longrightarrow \quad tp > -\frac{p}{2} - \frac{p}{2} + 1 \quad \Longrightarrow \quad t > \frac{1-p}{p} > -1.$$

Since t is an integer satisfying $-1 < t < 1$, we conclude $t = 0$. Substituting into (3.17) gives (3.8).

Finally, we analyze the sign of a:

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

  $$\begin{aligned} a + (b-1)b^{\kappa-1} &= (b-1)b^{\kappa-1} + b^{\kappa-2}r_{\kappa-2} + \cdots + b^0 r_0 \\ &\geqslant (b-1)b^{\kappa-1}. \end{aligned}$$

  Hence

  $$a \geqslant 0.$$

  Also,

  $$\begin{aligned} a + (b-1)b^{\kappa-1} &= (b-1)b^{\kappa-1} + \left[b^{\kappa-2}r_{\kappa-2} + \cdots + b^0 r_0\right] \\ &\leqslant (b-1)\left(b^{\kappa-1} + \cdots + b^0\right) \\ &= b^{\kappa} - 1. \end{aligned}$$

  Hence

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

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

$$\begin{aligned} a+(b-1)b^{\kappa-1} &\leqslant (b-2)b^{\kappa-1} + \left[b^{\kappa-2}r_{\kappa-2} + \cdots + b^0 r_0\right] \\ &\leqslant (b-2)b^{\kappa-1} + \left[(b-1)(b^{\kappa-1} + \cdots + b^0)\right] \\ &= (b-2)b^{\kappa-1} + \left[b^{\kappa-1} - 1\right] \\ &= (b-1)b^{\kappa-1} - b^{\kappa-1} + b^{\kappa} - 1 \\ &= (b-1)b^{\kappa-1} - 1. \end{aligned}$$

  Thus,

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

  Also,

$$\begin{aligned} a+(b-1)b^{\kappa-1} &= r_{\kappa-1}b^{\kappa-1} + \cdots + b^0 r_0 \\ &\geqslant 0 \end{aligned}$$

  Hence

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

□

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

$$q_0 = b^{\kappa}q_{\kappa} + b^{\kappa-1}a_{\kappa-1} + b^{\kappa-2}a_{\kappa-2} + \cdots + b^0 a_0. \tag{3.19}$$

(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 (3.19), 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 [1, 2]. This library provides a specialized interface for constructing and verifying arithmetic circuits.

4.1. Admissible bases. We provide Python code for determining which bases are inadmissible for a given modulus p (see Subsection 2.2, Comment (1)). This consideration applies specifically to signed range checks and ReLU verification, and is not relevant when performing range checks for nonnegative integers.

```python
def find_n(p, b):
    """Find the smallest n such that b^(n-1) < p < b^n."""
    n = 1
    while b**n <= p:
        n += 1
    return n

def is_admissible(p, b):
    """Check if b is admissible for prime p."""
    n = find_n(p, b)
    lower_bound = b**(n-1)
    upper_bound = b**n
    if lower_bound < p < upper_bound:
        threshold = 2 * (b - 1) * b**(n-2)
        return not (lower_bound < p < threshold)
    return False  # In case no such n exists

def find_inadmissible_bases(p, max_b=100):
    """Find all inadmissible bases for given p up to max_b."""
    inadmissible_bases = [b for b in range(2, max_b + 1) if not is_admissible(p, b)]
    return inadmissible_bases

def find_unique_n(p, b):
    """Find the unique n such that 2(b - 1)b^(n - 2) < p < b^n."""
    n = 2  # Start with n = 2 as per the problem statement
    while True:
        lower_bound = 2 * (b - 1) * b**(n - 2)
        upper_bound = b**n
        if lower_bound < p < upper_bound:
            return n
        n += 1  # Increment n until the condition is satisfied
```

Listing 1: Python code: given a modulus p, generate a list of inadmissible bases b up to 100

```
>>> find_inadmissible_bases(2**(31) - 1, max_b = 100)
[7, 14, 20, 21, 33, 34, 35, 65, 66, 67, 68, 69, 70, 71, 72, 73]
```

Listing 2: Inadmissible bases b up to 100 with respect to modulus $p = 2^{31} - 1$ (M31)

```
>>> find_unique_n(2**(31) - 1, 10)
10
```

Listing 3: Unique n such that $2(b-1)b^{n-2} < p < b^n$, for $p = 2^{31} - 1$ (M31) and $b = 10$

```
>>> p = 21888242871839275222246405745257275088548364400416034343698204186575808495617
>>> find_inadmissible_bases(p, max_b = 100)
[3, 5, 6, 9, 17, 23, 31, 36, 42, 49, 54, 59, 65, 72, 80, 81, 90, 101]
```

Listing 4: Inadmissible bases b up to 100 with respect to p, where p is the order of the scalar field of the BN254 curve.

```
>>> p = 21888242871839275222246405745257275088548364400416034343698204186575808495617
>>> find_unique_n(p, 10)
77
```

Listing 5: Unique n such that $2(b-1)b^{n-2} < p < b^n$, for the BN254 prime and $b = 10$

4.2. 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 $\kappa$
Ensure: a list $r$ representing the $\kappa$ least significant base-$b$ digits of $q_0$
 1: $r \leftarrow []$                                  ▷ Initialize an empty list
 2: for $i \leftarrow 0$ to $\kappa - 1$ do
 3:     append $q_0 \bmod b$ to $r$
 4:     $q_0 \leftarrow \lfloor q_0/b \rfloor$          ▷ Shift $q_0$ to the right by one digit
 5: end for
 6: return $r$
```

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

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

    r
}
```

Listing 6: 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 r = Vec::with_capacity(kappa as usize); // Preallocate vector
    let mut q = q_0; // Copy q_0 to modify iteratively

    for _ in 0..kappa {
        r.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
    }

    r
}
```

Listing 7: ECC Rust API: compute the κ least significant base-b digits of a nonnegative integer

4.3. 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 $r_i(r_i - 1) \equiv 0 \bmod p$ 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 4.2 `from_binary`: reconstruct and verify a nonnegative integer from at most κ least significant bits

```
Require: list of binary digits r, nonnegative integer $\kappa$, and original value a
Ensure: reconstructed_integer: the integer represented by the first $\kappa$ bits of r
reconstructed_integer $\leftarrow 0$
for $i \leftarrow 0$ to $\max\{\kappa - 1, \mathrm{len}(r) - 1\}$ do
    bit $\leftarrow r[i]$                                    ▷ 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_i $\leftarrow$ bit $\times 2^i$
    reconstructed_integer $\leftarrow$ 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 r, nonnegative integer $\kappa$, and original value a
Ensure: reconstructed_integer: the integer represented by the first $\kappa$ base-$b$ digits of r
reconstructed_integer $\leftarrow 0$
LOOKUP_TABLE $\leftarrow \{0, 1, \ldots, b-1\}$                    ▷ Predefined valid digit set
for $i \leftarrow 0$ to $\max\{\kappa - 1, \mathrm{len}(r) - 1\}$ do
    digit $\leftarrow r[i]$
    Enforce digit $\in$ LOOKUP_TABLE as a circuit constraint
    digit_by_b_to_the_i $\leftarrow$ digit $\times\, b^i$
    reconstructed_integer $\leftarrow$ 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>, r: &[Variable]) {
    for &bit in r.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>, r: &[Variable], kappa: usize, a: Variable) -> Variable {
    binary_digit_check(api, r);
    let mut reconstructed_integer = api.constant(0);

    for (i, &bit) in r.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 8: 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 [3] 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 ‘r‘ is a valid base-b digit using the lookup table.
fn base_b_digit_check<C: Config, API: RootAPI<C>>(
    api: &mut API,
    r: &[Variable],
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable
) {
    for &digit in r.iter() {
        lookup_digit(api, digit, lookup_table);
    }
}

// Reconstruct an integer from the first ‘kappa‘ digits in ‘r‘ (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,
    r: &[Variable],
    kappa: usize,
    b: u32,
    lookup_table: &mut LogUpSingleKeyTable,
    a: Variable
) -> Variable {
    base_b_digit_check(api, r, b, lookup_table);

    let mut reconstructed_integer = api.constant(0);
    for (i, &digit) in r.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 9: ECC Rust API: reconstruct a nonnegative integer from at most κ least significant base-b digits and impose constraints

4.4. 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 r, where $r \equiv a \bmod p$ and $0 \leqslant r < p$. We then perform the shift on r, 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 $\texttt{sign}(a) \cdot r$, where r is the least residue of a modulo p.

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 (8), no additional constraint is required after verifying that the κ base-b digits determine the shifted value. Since $\mathrm{ReLU}(a)$ is computed directly within the circuit as an internal value, we are not asking the prover to supply it as a public input. If, however, we wished to prove that the circuit's output matches an externally computed or public value, we could impose one final constraint to enforce this equality.

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

```
Require: $p$ is an $n$-bit prime; $a$ is an integer in $(-p/2, p/2]$, replaced by its least residue $r$ in preprocessing; kappa $\leqslant n-1$ is a nonnegative integer
Ensure: relu_of_a $= \max\{0, a\}$
                                        ▷ Step 1: Replace $a$ with its least residue modulo $p$
lr_a $\leftarrow a \bmod p$
                                        ▷ Step 2: Shift lr_a by $2^{\kappa-1}$
shift $\leftarrow 2^{\kappa-1}$
lr_a_shifted $\leftarrow$ (lr_a + shift) mod $p$
                                        ▷ Step 3: Compute $\kappa$ unconstrained binary digits of the shifted value
bits $\leftarrow$ to_binary(lr_a_shifted, kappa)
                                        ▷ Step 4: Impose bit constraints and ensure reconstruction equals shifted input
_ $\leftarrow$ from_binary(bits, kappa)
                                        ▷ Step 5: Extract sign bit and compute ReLU($a$) = lr_a × sign($a$)
relu_of_a $\leftarrow$ lr_a × bits[kappa − 1]
return 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 a prime; $b \geqslant 2$ is an integer base such that $2(b-1)b^{n-2} < p < b^n$; $a$ is an integer in $(-p/2, p/2]$, and $\kappa \leqslant n-1$
Ensure: Returns ReLU($a$) = sign_a × lr_a
                                        ▷ Step 1: Compute the least residue of $a$ modulo $p$
lr_a $\leftarrow a \bmod p$
                                        ▷ Step 2: Shift lr_a by $(b-1)b^{\kappa-1}$
shift $\leftarrow (b-1)b^{\kappa-1}$
lr_a_shifted $\leftarrow$ (lr_a + shift) mod $p$
                                        ▷ Step 3: Compute $\kappa$ unconstrained base-$b$ digits of the shifted value
digits $\leftarrow$ to_base_b(lr_a_shifted, $\kappa$, $b$)
                                        ▷ Step 4: Enforce digit constraints and reconstruct using lookup table
_ $\leftarrow$ from_base_b(digits, $\kappa$, $b$)   ▷ This enforces digit validity and ensures reconstruction equals lr_a_shifted
                                        ▷ Step 5: Extract sign bit
sign_a $\leftarrow$ is_equal(digits[$\kappa - 1$], $b - 1$)
                                        ▷ Step 6: Return ReLU($a$)
return sign_a × 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 10: ECC Rust API: verifying 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 11: ECC Rust API: verify whether a lies in $[-(b-1)b^{\kappa-1}, b^{\kappa-1})$, then verify ReLU($a$)

5. Examples

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 $b = 2$ and fix $\kappa = 4$, which satisfies $\kappa \leqslant n-1$. We let a range over the balanced residue interval

$$(-p/2, p/2] \cap \mathbb{Z} = \{-15, \ldots, 15\},$$

and proceed as follows:

- Compute r, the least residue of a mod p. *Example:* if $a = -15$, then $r = 16$.
- Compute $r + 2^{\kappa-1}$. *Example:* $r + 2^3 = 16 + 8 = 24$.
- Compute $r^\sharp$, the least residue of $r + 2^{\kappa-1}$ modulo p. *Example:* $r^\sharp = 24$.
- Compute $(r_{\kappa-1}, \ldots, r_0)$, the κ least significant bits of $r^\sharp$. *Example:* $r^\sharp = 24$ has binary representation 11000, so

  $$(r_3, r_2, r_1, r_0) = (1, 0, 0, 0).$$

  Note that we discard the leading bit; only the κ least significant bits are retained.
- Impose the constraints:
  - $r_i(r_i - 1) \equiv 0 \bmod p$ for each i.
  - The reconstruction constraint:
    $$r + 2^{\kappa-1} \equiv 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0 \bmod p. \tag{$*$}$$

  Example: The first constraint is satisfied: each $r_i \in \{0, 1\}$. But the right-hand side of $(*)$ is:

  $$2^3(1) + 2^2(0) + 2^1(0) + 2^0(0) = 8,$$

  while $r + 2^{\kappa-1} \equiv 24 \bmod p$. Hence, the constraint $(*)$ fails.
- Compute the least residue of the right-hand side of $(*)$. *Example:* $8 \bmod 31 = 8$.
- Compare this with $r^\sharp$ to check whether $(*)$ is satisfied. *Example:* $r^\sharp = 24 \neq 8$, so the constraint fails. This tells us that $a = -15$ lies outside the valid range $[-2^{\kappa-1}, 2^{\kappa-1}) = [-8, 8)$.
- Compute $r_{\kappa-1} \cdot r$, which equals $\mathrm{ReLU}(a)$ when the constraints are satisfied. *Example:* $r_3 = 1$ and $r = 16$, so $r_3 \cdot r = 16$. 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 $a \in (-p/2, p/2]$. When $(*)$ holds and the constraints are satisfied, the output of $r_{\kappa-1} r$ correctly recovers $\mathrm{ReLU}(a)$.

a	r	$r+2^{\kappa-1}$	$r^\sharp$	κ LSB $r^\sharp$	RHS($*$)	($*$) holds	$r_{\kappa-1}r$
-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 $a \notin (-p/2, p/2]$, the constraints may fail to hold, revealing that a lies outside the valid range. Even if the constraints are satisfied, the final value $r_{\kappa-1}r$ may no longer equal $\mathrm{ReLU}(a)$, since the sign bit no longer reliably encodes the correct comparison.

a	r	$r+2^{\kappa-1}$	$r^\sharp$	κ LSB $r^\sharp$	RHS$(*)$	$(*)$ holds	$r_{\kappa-1}r$
-30	1	9	9	1001	9	✓	1
16	16	24	24	1000	8	✗	16

Table 2: Examples where $a \notin (-p/2, p/2]$: constraints may not be satisfied, or they may be while $\mathrm{ReLU}(a) \neq r_{\kappa-1}r$.

5.2. Base-3 representation. Consider the prime $p = 37$, base $b = 3$, and integer $n = 4$, which satisfy

$$2(b-1)b^{n-2} = 2 \cdot 2 \cdot 3^2 = 36 < 37 < 81 = 3^4.$$

We fix $\kappa = 3$, which satisfies the condition $\kappa \leqslant n-1$. We let a range over the balanced residue interval

$$(-p/2, p/2] \cap \mathbb{Z} = \{-18, \ldots, 18\},$$

and proceed as follows:

- Compute r, the least residue of a mod p. *Example:* if $a = -18$, then $r = 19$.
- Compute the shifted value
$$r + (b-1)b^{\kappa-1} = r + 2 \cdot 3^2 = r + 18.$$
Example: $r + 18 = 19 + 18 = 37$.
- Compute $r^\sharp$, the least residue of $r + 18$ mod p. *Example:* $r^\sharp = 37$ mod $37 = 0$.
- Compute the κ least significant base-3 digits of $r^\sharp$, denoted (r_2, r_1, r_0). *Example:* $r^\sharp = 0$ has ternary representation $(0,0,0)$.
- Impose the constraints:
  - For each i, require
$$r_i(r_i - 1)(r_i - 2) \equiv 0 \bmod p,$$
ensuring $r_i \in \{0, 1, 2\}$.
  - Enforce the reconstruction constraint
$$r + 18 \equiv 3^2 r_2 + 3^1 r_1 + 3^0 r_0 \bmod p. \tag{$*$}$$

  Example: RHS$(*) = 3^2 \cdot 0 + 3^1 \cdot 0 + 3^0 \cdot 0 = 0$. Since $r + 18 = 37$, which is congruent to 0 mod 37, the constraint $(*)$ is satisfied.
- Compute the least residue of the right-hand side of $(*)$. *Example:* 0 mod $37 = 0$.
- Compare with $r^\sharp$ to determine whether $(*)$ is satisfied. *Example:* $r^\sharp = 0$ and RHS$(*) = 0$, so the constraint holds. This confirms that $a = -18$ lies in the valid range $[-18, 9)$.
- Compute the sign flag
$$\mathrm{sign}(a) = \begin{cases} 1 & \text{if } r_{\kappa-1} = b-1, \\ 0 & \text{otherwise.} \end{cases}$$
Example: $r_2 = 0 \neq 2$, so $\mathrm{sign}(a) = 0$.
- Finally, compute
$$\mathrm{ReLU}(a) = \mathrm{sign}(a) \cdot r.$$
Example: $0 \cdot 19 = 0 = \mathrm{ReLU}(-18)$, as expected.

The table below shows values for all $a \in (-p/2, p/2]$. For each value, it displays whether the constraint $(*)$ is satisfied, and whether the computed value $\mathrm{sign}(a) \cdot r$ matches $\mathrm{ReLU}(a)$.

a	r	$r+18$	$r^\sharp$	κ LSD $r^\sharp$	RHS($*$)	($*$) holds	sign(a)	sign(a)r
−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 $-p/2 < a \leqslant p/2$ does not hold, the result is not guaranteed to be correct. For example, even if the constraints are satisfied, the computed value $\text{sign}(a) \cdot r$ may not equal $\text{ReLU}(a)$.

a	r	$r+18$	$r^\sharp$	κ LSD $r^\sharp$	RHS($*$)	($*$) holds	sign(a)	sign(a)r
-36	1	19	19	201	19	✓	1	-36
19	19	37	0	000	0	✓	0	0

Table 4: Examples where $a \notin (-p/2, p/2]$: constraints may not be satisfied, or they may be while $\text{ReLU}(a) \neq \text{sign}(a) \cdot r$.

References

[1] Polyhedra Network. *ExpanderCompilerCollection: High-Level Circuit Compiler for the Expander Proof System*. `https://github.com/PolyhedraZK/ExpanderCompilerCollection`. Accessed April 11, 2025.

[2] Polyhedra Network. *ExpanderCompilerCollection: Rust Frontend Introduction*. `https://docs.polyhedra.network/expander/rust/intro`. Accessed April 11, 2025.

[3] Polyhedra Network. *LogUp: Lookup-Based Range Proofs in ExpanderCompilerCollection*. `https://docs.polyhedra.network/expander/std/logup`. Accessed April 11, 2025.