RANGE CHECK

1. THE PROCESS

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is an n-bit prime (typically, $n \approx 256$ in practice):

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

(2) Assume[1] integer a lies in the range

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

and define r as the least residue[2] of a modulo p:

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

(3) Fix[3] $\kappa \leqslant n-1$. In practice, κ is typically at most 64 and is chosen as small as possible to minimize circuit size.

(4) Compute the least residue $r^\sharp$ of $r + 2^{\kappa-1}$ modulo p, i.e. [Algorithm 3.3 Step 2, Listing 3]

$$r^\sharp \equiv r + 2^{\kappa-1} \bmod p, \quad 0 \leqslant r^\sharp < p.$$

(5) Compute the κ least significant bits of $r^\sharp$ [Proposition 2.3, Algorithm 3.1, Listing 1]:

$$r^\sharp = 2^\kappa q_\kappa + 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0, \quad q_\kappa \in \mathbb{Z}, \quad r_i \in \{0,1\}, \quad 0 \leqslant i < \kappa.$$

(6) Impose constraints in the arithmetic circuit [Algorithm 3.2, Listing 2]:

- For $0 \leqslant i < \kappa$, ensure $r_i \equiv 0$ or $1 \bmod p$ by requiring

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

- Require [Algorithm 3.2, Listing 2; Algorithm 3.3 Step 3, Listing 3]

$$r + 2^{\kappa-1} \equiv 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0 \bmod p.$$

(7) Assuming $-p/2 < a \leqslant p/2$ in the first place and $\kappa \leqslant n-1$, these constraints guarantee [Proposition 2.1]

$$-2^{\kappa-1} \leqslant a < 2^{\kappa-1} \quad \text{and} \quad a + 2^{\kappa-1} = 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0,$$

provided each r_i is taken as a least residue.[4]

Remark 1.1. Although the outline above and code in §3 illustrate how to verify whether a *single* integer a lies in the range $[-2^{\kappa-1}, 2^{\kappa-1})$, the procedure extends naturally to an array of integers. One simply applies the same sequence of steps to each element. ■

Remark 1.2. Suppose we wish to verify that an integer a lies in the interval $[-S, S)$ where S is not a power of two. A common strategy is to choose κ such that $2^{\kappa-1} \leqslant S$, and then to verify that $a \in [-2^{\kappa-1}, 2^{\kappa-1})$, which is a subinterval of $[-S, S)$. This check is sufficient when a naturally falls within $[-2^{\kappa-1}, 2^{\kappa-1})$. However, if a lies in the remainder of the interval (i.e., in $[-S, -2^{\kappa-1})$ or in $[2^{\kappa-1}, S)$), then the κ-bit range check alone will fail to capture the full range. In such cases, one would need to adjust by, for example, verifying that $a - 2^{\kappa-1}$ (or $a + 2^{\kappa-1}$) lies in a correspondingly shifted range. This additional shifting complicates the procedure, so in practice it is often preferable either to choose S to be a power of two or to restrict a to a subinterval for which the simple κ-bit check suffices. ■

[1] This is a hypothesis of Proposition 2.1(b), which we invoke in (7). If it cannot be justified by off-circuit reasoning, then the result does not hold in general. One possible off-circuit justification is that a is represented as a signed 64-bit integer (`i64`) in the Rust framework. If $p > 2^{65}$ then, since `i64` constrains values to $-2^{63} \leqslant a < 2^{63}$, every valid `i64` value necessarily satisfies $-p/2 < a \leqslant p/2$, ensuring the assumption holds without requiring an explicit range check.

[2] Since arithmetic circuit wires are represented by canonical elements in $\{0, \ldots, p-1\}$, we must translate conditions on a into equivalent statements about r, and vice versa. This is particularly relevant when performing signed comparisons or encoding operations.

[3] The assumption that $\kappa \leqslant n-1$ means that integers cannot have multiple bitstring representations modulo p. There are 2^κ bitstrings of length κ, and we want this to be less than p. For example, $0 \equiv 2^4(1) + 2^0(1) \bmod 17$, so 00000 and 10001 both represent 0 modulo the 5-bit integer 17.

[4] In practice, wires in an arithmetic circuit are represented by canonical elements in $\{0, \ldots, p-1\}$. Hence, if $r_i \equiv 0$ or $1 \bmod p$, we indeed get $r_i \in \{0,1\}$.

Proposition 2.1. *Let m and n be integers satisfying $2^{n-1} < m < 2^n$. Let a be an integer, and let r denote its least residue modulo m. Suppose κ is a nonnegative integer such that $\kappa < n$, and for each $0 \leqslant i < \kappa$, let a_i be an integer with r_i as its least residue modulo m. Finally, define $r^\sharp$ as the least residue of $r + 2^{\kappa-1}$ modulo m.*

(a) The following statements are equivalent:

(i) $a_i \equiv 0$ or $1 \bmod m$ for each $0 \leqslant i < \kappa$ and $r + 2^{\kappa-1} \equiv 2^{\kappa-1}a_{\kappa-1} + \cdots + 2^0 a_0 \bmod m$.

(ii) $r_i \in \{0,1\}$ for each $0 \leqslant i < \kappa$, and $r^\sharp = 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0$.

(b) If $-m/2 < a \leqslant m/2$ and (i) holds, then $r^\sharp = a + 2^{\kappa-1}$. Consequently: if $r_{\kappa-1} = 0$, then $-2^{\kappa-1} \leqslant a < 0$, while if $r_{\kappa-1} = 1$, then $0 \leqslant a < 2^{\kappa-1}$.

Remark 2.2. In part (a), statement (i), the condition $a_i \equiv 0$ or $1 \bmod m$ is equivalent to $a_i(a_i - 1) \equiv 0 \bmod m$ when m is prime.

Part (b) may be restated as follows: If $-m/2 < a \leqslant m/2$ and (i) holds, then $-2^{\kappa-1} \leqslant a < 2^{\kappa-1}$, and $(1 - r_{\kappa-1}, r_{\kappa-2}, \ldots, r_0)$ is the κ-bit two's complement representation of a. ■

Proof of Proposition 2.1. (a) Suppose (ii) holds. Then, for each $0 \leqslant i < \kappa$, we have $r_i \in \{0,1\}$, which, since $a_i \equiv r_i \bmod m$, implies that $a_i \equiv 0$ or $1 \bmod m$. Furthermore,

$$r + 2^{\kappa-1} \equiv r^\sharp \equiv 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0 \equiv 2^{\kappa-1}a_{\kappa-1} + \cdots + 2^0 a_0 \bmod m.$$

Thus, (i) holds.

Conversely, suppose (i) holds. Then, since $a_i \equiv 0$ or $1 \bmod m$ and r_i is the least residue of $a_i \bmod m$, it follows that $r_i \in \{0,1\}$ for all $0 \leqslant i < \kappa$. Additionally,

$$r^\sharp \equiv r + 2^{\kappa-1} \equiv 2^{\kappa-1}a_{\kappa-1} + \cdots + 2^0 a_0 \equiv 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0 \bmod m.$$

Thus, there exists and integer t such that

$$r^\sharp + tm = 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0.$$

To show that $t = 0$, we consider the cases:

- If $t \geqslant 1$, then since $r^\sharp \geqslant 0$, $2^{n-1} < m$, and $\kappa \leqslant n-1$, we obtain

  $$r^\sharp + tm \geqslant m > 2^{n-1} \geqslant 2^\kappa > 2^{\kappa-1} + \cdots + 2^0 \geqslant 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0,$$

  contradicting the equality above.

- If $t \leqslant -1$, then since $r^\sharp < m$, we have

  $$r^\sharp + tm < 0 \leqslant 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0,$$

  again leading to a contradiction.

Thus, we must have $t = 0$, proving that (ii) holds.

(b) Since $a + 2^{\kappa-1} \equiv r + 2^{\kappa-1} \equiv r^\sharp \bmod m$, we can write

$$a + 2^{\kappa-1} = r^\sharp + tm$$

for some integer t. Assuming (i) holds, which is equivalent to (ii), we obtain

$$a + 2^{\kappa-1} = r^\sharp + tm = 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0 r_0 + tm.$$

To determine t, we consider two cases:

- If $t \geqslant 1$, then using $2^{n-1} < m$ and $\kappa \leqslant n-1$, we find

  $$a \geqslant m - 2^{\kappa-1} \geqslant m - 2^{n-2} > m - m/2 = m/2.$$

  This contradicts $a \leqslant m/2$.

- If $t \leqslant -1$, then
$$a \leqslant \left(2^{\kappa-2} + \cdots + 2^0\right) - m < 2^{\kappa-1} - m \leqslant 2^{n-2} - m < m/2 - m = -m/2.$$
This contradicts $a > -m/2$.

Since both cases contradict the given bounds on a, we conclude that $t = 0$, yielding
$$r^\sharp = a + 2^{\kappa-1}.$$

Now, we analyze the sign of a:

- If $r_{\kappa-1} = 0$, then
$$a + 2^{\kappa-1} = 2^{\kappa-2} r_{\kappa-2} + \cdots + 2^0 r_0 \leqslant 2^{\kappa-2} + \cdots + 2^0 < 2^{\kappa-1}.$$
Thus,
$$a < 2^{\kappa-1} - 2^{\kappa-1} = 0.$$

- If $r_{\kappa-1} = 1$, then
$$a + 2^{\kappa-1} = 2^{\kappa-1} + 2^{\kappa-2} r_{\kappa-2} + \cdots + 2^0 r_0 \geqslant 2^{\kappa-1}.$$
Hence
$$a \geqslant 0.$$

$\square$

Proposition 2.3. *Let $q_0 \in \mathbb{Z}$. For $0 \leqslant i < \kappa$, let q_{i+1} and a_i be the unique integers satisfying $q_i = 2q_{i+1} + a_i$ and $a_i \in \{0, 1\}$.*

(a) Then
$$q_0 = 2^\kappa q_\kappa + 2^{\kappa-1} a_{\kappa-1} + 2^{\kappa-2} a_{\kappa-2} + \cdots + 2^0 a_0. \tag{2.1}$$

(b) The following statements are equivalent: (i) $0 \leqslant q_0 < 2^\kappa$; (ii) $q_\kappa = 0$; (iii) $(a_{\kappa-1}, \ldots, a_0)$ is the κ-bit binary 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 $q_i = 2q_{i+1} + a_i$, $a_i \in \{0, 1\}$ for $0 \leqslant i < n$ and also $i = n$. By inductive hypothesis,
$$\begin{aligned} q_0 &= 2^n q_n + 2^{n-1} a_{n-1} + 2^{n-2} a_{n-2} + \cdots + 2^0 a_0 \\ &= 2^n (2q_{n+1} + a_n) + 2^{n-1} a_{n-1} + 2^{n-2} a_{n-2} + \cdots + 2^0 a_0 \\ &= 2^{n+1} q_{n+1} + 2^n a_n + \cdots + 2^0 a_0. \end{aligned}$$

(b) Suppose $0 \leqslant q_0 < 2^\kappa$. In view of (2.1), this implies
$$2^\kappa q_\kappa = q_0 - \left(2^{\kappa-1} a_{\kappa-1} + 2^{\kappa-2} a_{\kappa-2} + \cdots + 2^0 a_0\right) \leqslant q_0 < 2^\kappa.$$

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

Hence $q_\kappa > -1$. We must therefore have $q_\kappa = 0$, which implies
$$q_0 = 2^{\kappa-1} a_{\kappa-1} + 2^{\kappa-2} a_{\kappa-2} + \cdots + 2^0 a_0,$$

i.e., $(a_{\kappa-1}, \ldots, a_0)$ is the κ-bit binary representation of q_0. Finally, if this holds, then $0 \leqslant q_0 < 2^\kappa$, because the right-hand side lies between 0 and $2^{\kappa-1} + 2^{\kappa-2} + \cdots + 2^0 = 2^\kappa - 1$. $\square$

3. THE CODE

The pseudocode and Rust code that follow are designed for implementation within the *ExpanderCompilerCollection* (ECC) framework [1]. This library provides a specialized interface for constructing and verifying arithmetic circuits.

Algorithm 3.1 `to_binary`: compute κ least significant bits of binary representation of a nonnegative integer

Require: nonnegative integers q_0 and κ
Ensure: a list r representing the κ least significant binary digits of q_0

1: $r \leftarrow []$ ▷ Initialize an empty list
2: **for** $i \leftarrow 0$ to $\kappa - 1$ **do**
3:     **append** q_0 mod 2 **to** r
4:     $q_0 \leftarrow \lfloor q_0/2 \rfloor$ ▷ Shift q_0 to the right by one bit
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 1: ECC Rust API: compute κ least significant bits of binary representation of a nonnegative integer

Algorithm 3.2 `from_binary`: reconstruct a nonnegative integer from at most κ least significant bits and impose constraints

Require: list of binary digits `r` and nonnegative integer κ
Ensure: `reconstructed_integer`: the integer represented by the first κ bits of `r`

1: `reconstructed_integer` $\leftarrow 0$
2: **for** $i \leftarrow 0$ to $\max\{\kappa - 1, \text{len}(r) - 1\}$ **do**
3:     `bit` $\leftarrow r[i]$ ▷ Binary digit check: ensure `bit` $\in \{0,1\}$
4:     `bit_minus_one` $\leftarrow 1 -$ `bit`
5:     `bit_by_bit_minus_one` $\leftarrow$ `bit` $\times$ `bit_minus_one`
6:     **assert** `bit_by_bit_minus_one` $= 0$
7:     `bit_by_two_to_the_i` $\leftarrow$ `bit` $\times 2^i$
8:     `reconstructed_integer` $\leftarrow$ `reconstructed_integer` $+$ `bit_by_two_to_the_i`
9: **end for**
10: **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) -> 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);
    }

    reconstructed_integer
}
```

Listing 2: ECC Rust API: reconstruct nonnegative integer from at most κ least significant bits and impose constraints

Algorithm 3.3 `range_check`: verify whether a lies in $[-2^{\kappa-1}, 2^{\kappa-1})$

Require: p is an n-bit prime, a is an integer in $(-p/2, p/2]$, and `kappa` $\leqslant n-1$ is a nonnegative integer
Ensure: Returns 1 if $a \in [-2^{\kappa-1}, 2^{\kappa-1})$, and 0 otherwise.

▷ Step 1: Compute least residue of a 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` $\bmod p$

▷ Step 3: Convert `lr_a_shifted` to an unconstrained κ-bit representation

`bits` $\leftarrow$ `to_binary(lr_a_shifted, kappa)`

▷ Step 4: Impose binary constraints and reconstruct

`reconstructed` $\leftarrow$ `from_binary(bits, kappa)`
`assert_equal(reconstructed, lr_a_shifted)`

▷ Step 5: Return 1 if a is in the desired range, 0 otherwise

return `(lr_a == reconstructed_a)`

```rust
fn range_check<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 + 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, reconstruct the sum
    let reconstructed = from_binary(api, &bits, kappa);

  // Step 4: Compute the boolean check (1 if valid, 0 otherwise)
    let is_valid = api.is_equal(a_shifted, reconstructed);

  // Step 5: Return the boolean result (1 if in range, 0 otherwise)
    is_valid
}
```

Listing 3: ECC Rust API: verify whether a lies in $[-2^{\kappa-1}, 2^{\kappa-1})$

4. Example

Consider the prime $p = 31$, which is a 5-bit prime ($n = 5$), because $2^4 \leqslant 31 < 2^5$.

- We fix $\kappa = 4$, which satisfies $\kappa \leqslant n-1$.
- We let a range over $(-p/2, p/2] \cap \mathbb{Z} = \{-15, \ldots, 15\}$.
- We compute r, the least residue of $a \bmod p$.
- We compute $r + 2^{\kappa-1}$.
- We compute $r^\sharp$, the least residue of $r + 2^{\kappa-1} \bmod p$.
- We compute $(r_{\kappa-1}, \ldots, r_0)$, the κ least significant bits of $r^\sharp$.
- The constraints are $r_i(r_i - 1) \equiv 0 \bmod p$ and

$$r + 2^{\kappa-1} \equiv 2^{\kappa-1} r_{\kappa-1} + \cdots + 2^0 r_0 \bmod p. \qquad (*)$$

- We compute the least residue of the right-hand side of $(*)$.
- We compare with $r^\sharp$ to determine whether $(*)$ is satisfied. If it is, then a lies in $[-2^{\kappa-1}, 2^{\kappa-1})$. If not, then a lies outside of this range.

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

What if the assumption that $a \in (-p/2, p/2]$ does not hold? Then the constraints may be satisfied, even though a does not lie in $[-2^{\kappa-1}, 2^{\kappa-1})$. This shows that the underlying assumption that $a \in (-p/2, p/2]$ is crucial.

a	r	$r+2^{\kappa-1}$	$r^\sharp$	κ LSB $r^\sharp$	RHS(∗)	(∗) holds
−30	1	9	9	1001	9	✓
32	1	9	9	1001	9	✓

References

[1] Polyhedra Network. *ExpanderCompilerCollection*. GitHub repository. Accessed January 28, 2025.