

CIRCUITIZING RELU I

DON'T LOOK UP

1	THE PROCESS	1
2	THE MATHS	2
3	THE CODE	4
4	EXAMPLE	5
	REFERENCES	7

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) Assume¹ integer a lies in the range

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

and define r as the least residue² of a modulo p :

$$a \equiv r \pmod{p}, \quad 0 \leq r < p.$$

- (3) Fix³ $\kappa \leq n - 1$. 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]

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

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

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

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

- For $0 \leq i < \kappa$, ensure $r_i \equiv 0$ or $1 \pmod{p}$ by requiring

$$r_i(r_i - 1) \equiv 0 \pmod{p}.$$

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

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

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

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

provided each r_i is taken as a least residue.⁴

- (8) Finally, compute⁵ [Algorithm 3.3 Step 4, Listing 3]

$$\text{ReLU}(a) = r_{\kappa-1} r.$$

Remark 1.1. Although the outline above and code in §3 illustrate how to verify the ReLU of a *single* integer a , the procedure extends naturally to an array of integers. One simply applies the same sequence of steps to each element, treating each entry's ReLU independently. ■

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 $-2^{63} \leq a < 2^{63}$, every valid i64 value necessarily satisfies $-p/2 < a \leq p/2$, ensuring the assumption holds without requiring an explicit range check.

² Since arithmetic circuit wires are represented by canonical elements in $\{0, \dots, p-1\}$, 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 $\kappa \leq n - 1$ 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, $0 \equiv 2^4(1) + 2^0(1) \pmod{17}$, 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 $\{0, \dots, p-1\}$. Hence, if $r_i \equiv 0$ or $1 \pmod{p}$, we indeed get $r_i \in \{0, 1\}$.

⁵ No extra constraint is required to check $\text{ReLU}(a) = r_{\kappa-1} r$ because $r_{\kappa-1}$ must be 1 if $a \geq 0$ and 0 otherwise, as enforced by the binary decomposition constraints. However, if we need to *prove* that $\text{ReLU}(a)$ matches a *public input* or a previously computed *external* value, we can add a further constraint to ensure they coincide. We'd need to assume this external value lies in $[0, p)$ or else do a range check.

2. THE MATHS

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 $0 \leq i < \kappa$, 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 $1 \pmod m$ for each $0 \leq i < \kappa$ and $r + 2^{\kappa-1} \equiv 2^{\kappa-1}a_{\kappa-1} + \cdots + 2^0a_0 \pmod m$.
- (ii) $r_i \in \{0, 1\}$ for each $0 \leq i < \kappa$, and $r^\sharp = 2^{\kappa-1}r_{\kappa-1} + \cdots + 2^0r_0$.

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

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

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

Moreover, if $-m/2 < a \leq m/2$ and (i) holds, then by part (b), $\text{ReLU}(a) = r_{\kappa-1}r$. ■

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

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

Thus, (i) holds.

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

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

Thus, there exists integer t such that

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

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

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

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

contradicting the equality above.

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

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

again leading to a contradiction.

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

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

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

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

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

To determine t , we consider the cases:

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

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

This contradicts $a \leq m/2$.

- If $t \leq -1$, then

$$a \leq (2^{\kappa-2} + \dots + 2^0) - m < 2^{\kappa-1} - m \leq 2^{n-2} - m < m/2 - m = -m/2.$$

This contradicts $a > -m/2$.

Since both cases contradict the given bounds on a , we conclude that $t = 0$, yielding

$$r^\# = a + 2^{\kappa-1}.$$

Now, we analyze the sign of a :

- If $r_{\kappa-1} = 0$, then

$$a + 2^{\kappa-1} = 2^{\kappa-2}r_{\kappa-2} + \dots + 2^0r_0 \leq 2^{\kappa-2} + \dots + 2^0 < 2^{\kappa-1}.$$

Thus,

$$a < 2^{\kappa-1} - 2^{\kappa-1} = 0.$$

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

$$a + 2^{\kappa-1} = 2^{\kappa-1} + 2^{\kappa-2}r_{\kappa-2} + \dots + 2^0r_0 \geq 2^{\kappa-1}.$$

Hence

$$a \geq 0.$$

□

Proposition 2.3. Let $q_0 \in \mathbb{Z}$. For $0 \leq i < \kappa$, 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

$$q_0 = 2^\kappa q_\kappa + 2^{\kappa-1}a_{\kappa-1} + 2^{\kappa-2}a_{\kappa-2} + \dots + 2^0a_0. \quad (2.1)$$

(b) The following statements are equivalent: (i) $0 \leq q_0 < 2^\kappa$; (ii) $q_\kappa = 0$; (iii) $(a_{\kappa-1}, \dots, a_0)$ 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 $n \geq 0$. Now consider $\kappa = n + 1$. We have $q_i = 2q_{i+1} + a_i$, $a_i \in \{0, 1\}$ for $0 \leq i < n$ and also $i = n$. By inductive hypothesis,

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

(b) Suppose $0 \leq q_0 < 2^\kappa$. In view of (2.1), this implies

$$2^\kappa q_\kappa = q_0 - (2^{\kappa-1}a_{\kappa-1} + 2^{\kappa-2}a_{\kappa-2} + \dots + 2^0a_0) \leq q_0 < 2^\kappa.$$

Hence $q_\kappa < 1$. Also,

$$-2^\kappa q_\kappa = (2^{\kappa-1}a_{\kappa-1} + 2^{\kappa-2}a_{\kappa-2} + \dots + 2^0a_0) - q_0 \leq 2^{\kappa-1} + 2^{\kappa-2} + \dots + 2^0 < 2^\kappa.$$

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

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

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

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  $q_0 \bmod 2$  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  $r$ 

```

```

1 fn to_binary<C: Config>(api: &mut API<C>, q_0: Variable, kappa: usize) -> Vec<Variable> {
2     let mut r = Vec::with_capacity(kappa); // Preallocate vector
3     let mut q = q_0; // Copy q_0 to modify iteratively
4
5     for _ in 0..kappa {
6         r.push(api.unconstrained_bit_and(q, 1)); // Extract least significant bit
7         q = api.unconstrained_shift_r(q, 1); // Shift right by 1 bit
8     }
9
10    r
11 }

```

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:    $\text{bit} \leftarrow r[i]$  ▷ Binary digit check: ensure  $\text{bit} \in \{0, 1\}$ 
4:    $\text{bit\_minus\_one} \leftarrow 1 - \text{bit}$ 
5:    $\text{bit\_by\_bit\_minus\_one} \leftarrow \text{bit} \times \text{bit\_minus\_one}$ 
6:   assert  $\text{bit\_by\_bit\_minus\_one} = 0$ 
7:    $\text{bit\_by\_two\_to\_the\_i} \leftarrow \text{bit} \times 2^i$ 
8:   reconstructed_integer  $\leftarrow \text{reconstructed\_integer} + \text{bit\_by\_two\_to\_the\_i}$ 
9: end for
10: return reconstructed_integer

```

```

1 fn binary_digit_check<C: Config>(api: &mut API<C>, r: &[Variable]) {
2     for &bit in r.iter() {
3         let bit_minus_one = api.sub(1, bit);
4         let bit_by_bit_minus_one = api.mul(bit, bit_minus_one);
5         api.assert_is_zero(bit_by_bit_minus_one);
6     }
7 }
8
9 fn from_binary<C: Config>(api: &mut API<C>, r: &[Variable], kappa: usize) -> Variable {
10    binary_digit_check(api, r);
11    let mut reconstructed_integer = api.constant(0);
12
13    for (i, &bit) in r.iter().take(kappa).enumerate() {
14        let bit_by_two_to_the_i = api.mul(1 << i, bit);
15        reconstructed_integer = api.add(reconstructed_integer, bit_by_two_to_the_i);
16    }
17
18    reconstructed_integer
19 }

```

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

Algorithm 3.3 `verify_relu`: verify $\text{ReLU}(a)$ in an arithmetic circuit

Require: p is an n -bit prime, a is an integer in $(-p/2, p/2]$, and $\kappa \leq n - 1$ is a nonnegative integer

Ensure: $\text{relu_of_a} = \max\{0, a\}$

- 1: $\text{lr_a} \leftarrow a \bmod p$ \triangleright Step 1: Compute least residue of a modulo p
 - 2: $\text{shift} \leftarrow 2^{\kappa-1}$ \triangleright Step 2: Shift lr_a by $2^{\kappa-1}$
 - 3: $\text{lr_a_shifted} \leftarrow \text{lr_a} + \text{shift} \bmod p$
 - 4: $\text{bits} \leftarrow \text{to_binary}(\text{lr_a_shifted}, \kappa)$ \triangleright Step 3: Convert lr_a_shifted to an unconstrained κ -bit representation
 - 5: $\text{reconstructed} \leftarrow \text{from_binary}(\text{bits}, \kappa)$ \triangleright Step 4: Impose binary constraints and reconstruct
 - 6: $\text{assert_equal}(\text{reconstructed}, \text{lr_a_shifted})$
 - 7: $\text{relu_of_a} \leftarrow \text{lr_a} \times \text{bits}[\kappa - 1]$ \triangleright Step 5: Final ReLU: $\text{ReLU}(a) = \text{lr_a} \times \text{bits}[\kappa - 1]$
 - 8: **return** relu_of_a
-

```
1 fn verify_relu<C: Config>(api: &mut API<C>, a: Variable, kappa: usize) -> Variable {
2   // Step 1: Shift a by 2^(kappa - 1) mod p
3   let shift = api.constant(1 << (kappa - 1));
4   let a_shifted = api.add(a, shift); // = a + 2^(kappa - 1) mod p
5
6   // Step 2: Convert a_shifted to binary (unconstrained)
7   let bits = to_binary(api, a_shifted, kappa);
8
9   // Step 3: From binary -> impose bit constraints, reconstruct the sum
10  let reconstructed = from_binary(api, &bits, kappa);
11
12  // Step 4: Enforce that reconstructed == a_shifted mod p
13  api.assert_is_equal(reconstructed, a_shifted);
14
15  // Step 5: Compute ReLU(a) = a * bits[kappa-1]
16  let relu_of_a = api.mul(a, bits[kappa - 1]);
17
18  relu_of_a
19 }
```

Listing 3: ECC Rust API: ensuring correctness of a ReLU computation

4. EXAMPLE

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

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

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

- We compute the least residue of the right-hand side of $(*)$.
- We compare with r^\sharp to determine whether $(*)$ is satisfied.
- We compute $r_{\kappa-1}r$, which is equal to $\text{ReLU}(a)$ if and only if the constraints are satisfied, or equivalently, if and only if $-2^{\kappa-1} \leq a < 2^{\kappa-1}$.

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

What if the assumption that $a \in (-p/2, p/2]$ does not hold? The constraints may not be satisfied, revealing that the integer lies outside of the range $[-2^{\kappa-1}, 2^\kappa)$, or $r_{\kappa-1}r$ may not be equal to $\text{ReLU}(a)$.

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

REFERENCES

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