AXILOG

CI-MATH-001

Certifiable Inference Mathematical Specification

Final v1.0 L2 [D1]

Last updated 20 January 2026
Compliance DO-178C · IEC 62304 · ISO 26262
Changelog 1 entry

Contents

Traceability: All code in `src/` SHALL be a literal transcription of the mathematics herein.

---

§1 Introduction

§1.1 Purpose

This document specifies the mathematical foundations for deterministic neural network inference. The goal is bit-identical execution across all platforms (x86, ARM, RISC-V) regardless of compiler, optimization level, or hardware implementation.

§1.2 Scope

Certifiable-inference provides:

- Fixed-point arithmetic (Q16.16)
- Matrix operations (multiply, add, transpose)
- Convolution (2D, valid padding)
- Activation functions (ReLU, Leaky ReLU)
- Pooling operations (max pooling)
- Deterministic hash tables

§1.3 Notation

Symbol	Meaning
$\lfloor x \rfloor$	Floor function
$\lceil x \rceil$	Ceiling function
$\equiv$	Definition / equivalence
$\wedge$	Logical AND
$\forall$	For all
$\in$	Element of
[a, b]	Closed interval
mod	Modulo operation
$\ll$	Left bit shift
$\gg$	Arithmetic right shift

§1.4 Document Alignment

Topic	CI-MATH-001	CI-STRUCT-001	SRS
Fixed-point format	§2	§3	SRS-002
Rounding	§2.5	§3.3	SRS-002
Matrix operations	§3	§4	SRS-003
Activations	§4	§5	SRS-004
Convolution	§5	§6	SRS-006
Pooling	§6	§7	SRS-008
Hash table	§7	§8	SRS-001
Fault model	§8	§9	SRS-002

§2 Fixed-Point Arithmetic

§2.1 Format Specification

Definition: Q16.16 fixed-point format uses a signed 32-bit integer where:

- Bits [31:16]: Signed integer part (16 bits)
- Bits [15:0]: Fractional part (16 bits)

```
Value = raw_integer / 2^16
raw_integer = Value × 2^16
```

§2.2 Constants

Constant	Symbol	Value	Hex
One	FIXED_ONE	65536	0x00010000
Half	FIXED_HALF	32768	0x00008000
Zero	FIXED_ZERO	0	0x00000000
Epsilon	FIXED_EPS	1	0x00000001
Maximum	FIXED_MAX	2147483647	0x7FFFFFFF
Minimum	FIXED_MIN	-2147483648	0x80000000
Shift	FIXED_SHIFT	16	—

§2.3 Range and Precision

Property	Value
Minimum representable	-32768.0
Maximum representable	+32767.99998474...
Precision (1 LSB)	1/65536 ≈ 0.0000152588
Decimal digits	~4.8 significant digits

§2.4 Conversion Functions

Integer to Fixed:

```
fixed_from_int(i) ≡ i ≪ FIXED_SHIFT
```

Fixed to Integer (truncation):

```
fixed_to_int(f) ≡ f ≫ FIXED_SHIFT
```

Float to Fixed (initialization only):

```
fixed_from_float(fl) ≡ ⌊fl × FIXED_ONE + 0.5⌋
```

Note: Float conversion is permitted only during model loading/initialization. Runtime inference uses integer-only operations.

§2.5 Arithmetic Operations

Addition:

```
fixed_add(a, b) ≡ a + b
```

Direct integer addition. Overflow behavior defined in §8.

Subtraction:

```
fixed_sub(a, b) ≡ a - b
```

Direct integer subtraction.

Multiplication with Rounding:

```
fixed_mul(a, b) ≡ round_shift((int64_t)a × (int64_t)b, FIXED_SHIFT)
```

Where `round_shift` implements round-to-nearest:

```
round_shift(x, shift) ≡ (x + (1 ≪ (shift - 1))) ≫ shift
```

Rationale: 64-bit intermediate prevents overflow. Adding FIXED_HALF before shifting implements round-to-nearest, minimizing cumulative quantization error.

Division:

```
fixed_div(a, b) ≡
    if b = 0: return 0 (with fault flag)
    else: ((int64_t)a ≪ FIXED_SHIFT) / b
```

Absolute Value:

```
fixed_abs(f) ≡
    if f ≥ 0: f
    else: -f
```

Note: `fixed_abs(FIXED_MIN)` overflows; this is handled via fault flags.

Negation:

```
fixed_neg(f) ≡ -f
```

§2.6 Determinism Guarantee

Theorem (Bit-Perfect Arithmetic): For any two DVM-compliant platforms A and B:

```
∀ a, b ∈ int32_t:
    fixed_mul(a, b) on A = fixed_mul(a, b) on B
```

Proof: All operations use:

1. Integer arithmetic only (no floating-point)
2. Explicit widening to 64-bit before overflow-prone operations
3. Deterministic rounding (round-to-nearest via addition)
4. No platform-dependent behavior (no FMA, no compiler optimization variability)

---

§3 Linear Algebra

§3.1 Matrix Representation

A matrix M with dimensions (rows × cols) is stored in row-major order:

```
M[i][j] = data[i × cols + j]
```

Where:

- i ∈ [0, rows)
- j ∈ [0, cols)

§3.2 Matrix Initialization

```
fx_matrix_init(m, buffer, rows, cols):
    m.data ← buffer
    m.rows ← rows
    m.cols ← cols
```

Precondition: `buffer` must have at least `rows × cols` elements.

§3.3 Matrix Multiplication

For matrices A (m×k) and B (k×n), compute C = A × B where C is (m×n):

```
fx_matrix_mul(A, B, C):
    PRECONDITION: A.cols = B.rows
    PRECONDITION: C.rows = A.rows ∧ C.cols = B.cols

    FOR i ← 0 TO A.rows - 1:
        FOR j ← 0 TO B.cols - 1:
            acc ← 0 (int64_t)
            FOR k ← 0 TO A.cols - 1:
                acc ← acc + (int64_t)A[i,k] × (int64_t)B[k,j]
            C[i,j] ← round_shift(acc, FIXED_SHIFT)
```

Complexity: O(m × n × k) time, O(1) auxiliary space.

Determinism: 64-bit accumulator prevents intermediate overflow. Final rounding is deterministic.

§3.4 Matrix Addition

```
fx_matrix_add(A, B, C):
    PRECONDITION: A.rows = B.rows = C.rows
    PRECONDITION: A.cols = B.cols = C.cols

    FOR i ← 0 TO A.rows × A.cols - 1:
        C.data[i] ← A.data[i] + B.data[i]
```

§3.5 Vector Dot Product

```
fx_dot(a, b, n):
    acc ← 0 (int64_t)
    FOR i ← 0 TO n - 1:
        acc ← acc + (int64_t)a[i] × (int64_t)b[i]
    RETURN round_shift(acc, FIXED_SHIFT)
```

§3.6 Address Independence

Theorem: Matrix operations produce identical results regardless of memory addresses.

Proof: All operations depend only on:

1. Element values (not addresses)
2. Dimension metadata
3. Index arithmetic (deterministic)

No pointer comparisons or address-dependent behavior exists.

---

§4 Activation Functions

§4.1 ReLU

```
ReLU(x) ≡ max(0, x)
```

Implementation:

```
fx_relu(mat):
    FOR i ← 0 TO mat.rows × mat.cols - 1:
        IF mat.data[i] < 0:
            mat.data[i] ← 0
```

Properties:

- In-place operation (modifies input)
- O(n) time, O(1) space
- Deterministic (simple comparison)

§4.2 Leaky ReLU

```
LeakyReLU(x, α) ≡
    if x ≥ 0: x
    else: α × x
```

Implementation:

```
fx_leaky_relu(mat, alpha):
    FOR i ← 0 TO mat.rows × mat.cols - 1:
        IF mat.data[i] < 0:
            mat.data[i] ← fixed_mul(mat.data[i], alpha)
```

Note: α is typically 0.01 (Q16.16: 655).

§4.3 Bias Addition

```
fx_add_bias(mat, bias):
    PRECONDITION: bias.cols = mat.cols

    FOR i ← 0 TO mat.rows - 1:
        FOR j ← 0 TO mat.cols - 1:
            mat[i,j] ← mat[i,j] + bias[0,j]
```

§4.4 Dense Layer Forward Pass

A complete dense layer: Y = ReLU(X × W + b)

```
fx_dense_forward(X, W, b, Y):
    fx_matrix_mul(X, W, Y)
    fx_add_bias(Y, b)
    fx_relu(Y)
```

---

§5 Convolution

§5.1 2D Convolution (Valid Padding)

For input I (H_in × W_in) and kernel K (K_h × K_w), output O has dimensions:

```
H_out = H_in - K_h + 1
W_out = W_in - K_w + 1
```

Definition:

```
O[i,j] = Σ_{di=0}^{K_h-1} Σ_{dj=0}^{K_w-1} I[i+di, j+dj] × K[di, dj]
```

§5.2 Implementation

```
fx_conv2d(input, kernel, output):
    PRECONDITION: output.rows = input.rows - kernel.rows + 1
    PRECONDITION: output.cols = input.cols - kernel.cols + 1

    FOR i ← 0 TO output.rows - 1:
        FOR j ← 0 TO output.cols - 1:
            acc ← 0 (int64_t)
            FOR di ← 0 TO kernel.rows - 1:
                FOR dj ← 0 TO kernel.cols - 1:
                    in_val ← input[i + di, j + dj]
                    k_val ← kernel[di, dj]
                    acc ← acc + (int64_t)in_val × (int64_t)k_val
            output[i,j] ← round_shift(acc, FIXED_SHIFT)
```

§5.3 Complexity

Metric	Value
Time	O(H_out × W_out × K_h × K_w)
Space	O(1) auxiliary
Memory reads	O(H_out × W_out × K_h × K_w)
Memory writes	O(H_out × W_out)

§5.4 Common Kernels

Identity (3×3):

```
[0, 0, 0]
[0, 1, 0]
[0, 0, 0]
```

Sobel Horizontal (3×3):

```
[-1, 0, +1]
[-2, 0, +2]
[-1, 0, +1]
```

Sobel Vertical (3×3):

```
[-1, -2, -1]
[ 0,  0,  0]
[+1, +2, +1]
```

---

§6 Pooling

§6.1 Max Pooling 2×2

For input I (H × W), output O has dimensions (H/2 × W/2):

```
O[i,j] = max(I[2i, 2j], I[2i, 2j+1], I[2i+1, 2j], I[2i+1, 2j+1])
```

§6.2 Implementation

```
fx_maxpool_2x2(input, output):
    PRECONDITION: output.rows = input.rows / 2
    PRECONDITION: output.cols = input.cols / 2

    FOR i ← 0 TO output.rows - 1:
        FOR j ← 0 TO output.cols - 1:
            v00 ← input[2i, 2j]
            v01 ← input[2i, 2j + 1]
            v10 ← input[2i + 1, 2j]
            v11 ← input[2i + 1, 2j + 1]

            m1 ← max(v00, v01)
            m2 ← max(v10, v11)
            output[i,j] ← max(m1, m2)
```

§6.3 Properties

Property	Value
Dimension reduction	2× in each spatial dimension
Range preservation	output ∈ [min(input), max(input)]
Determinism	Comparison-only (no arithmetic rounding)

§6.4 Boundary Handling

If input dimensions are not divisible by 2, the rightmost column and/or bottom row are ignored. This is explicit in the precondition and must be handled at model design time.

---

§7 Deterministic Hash Table

§7.1 Purpose

A hash table with deterministic iteration order, required for reproducible model weight storage and lookup.

§7.2 Hash Function

Using FNV-1a (Fowler-Noll-Vo):

```
fnv1a_hash(key):
    hash ← 2166136261 (FNV offset basis)
    FOR each byte c in key:
        hash ← hash XOR c
        hash ← hash × 16777619 (FNV prime)
    RETURN hash
```

Properties:

- Deterministic across platforms
- No floating-point
- Good distribution for string keys

§7.3 Collision Resolution

Linear probing with insertion-order tracking:

```
probe_index(hash, i, capacity) ≡ (hash + i) mod capacity
```

§7.4 Iteration Order

Iteration follows insertion order, not hash order. This is maintained via a separate index array:

```
FOR i ← 0 TO insert_count - 1:
    yield entries[insertion_order[i]]
```

§7.5 Operations

Insert:

```
d_table_insert(table, key, value):
    IF table.count ≥ table.capacity:
        RETURN D_TABLE_FULL

    hash ← fnv1a_hash(key)
    FOR i ← 0 TO table.capacity - 1:
        idx ← probe_index(hash, i, table.capacity)
        IF table.entries[idx].occupied:
            IF table.entries[idx].key = key:
                RETURN D_TABLE_KEY_EXISTS
        ELSE:
            table.entries[idx] ← {key, value, occupied: true}
            table.insertion_order[table.count] ← idx
            table.count ← table.count + 1
            RETURN D_TABLE_OK
```

Get:

```
d_table_get(table, key, out_value):
    hash ← fnv1a_hash(key)
    FOR i ← 0 TO table.capacity - 1:
        idx ← probe_index(hash, i, table.capacity)
        IF NOT table.entries[idx].occupied:
            RETURN D_TABLE_NOT_FOUND
        IF table.entries[idx].key = key:
            *out_value ← table.entries[idx].value
            RETURN D_TABLE_OK
    RETURN D_TABLE_NOT_FOUND
```

---

§8 Fault Model

§8.1 Fault Flags

```
typedef struct {
    uint32_t overflow    : 1;   /* Result exceeded INT32_MAX */
    uint32_t underflow   : 1;   /* Result below INT32_MIN */
    uint32_t div_zero    : 1;   /* Division by zero attempted */
    uint32_t domain      : 1;   /* Input outside valid range */
    uint32_t precision   : 1;   /* Precision loss detected */
    uint32_t _reserved   : 27;
} ci_fault_flags_t;
```

§8.2 Fault Behavior

Condition	Action	Fault Flag
Multiplication overflow	Saturate to FIXED_MAX/MIN	overflow/underflow
Division by zero	Return 0	div_zero
Invalid dimension	Skip operation	domain
Precision loss in conversion	Continue with rounded value	precision

§8.3 Fault Propagation

Fault flags are **sticky**: once set, they remain set until explicitly cleared.

```
ci_has_fault(f) ≡ f.overflow ∨ f.underflow ∨ f.div_zero ∨ f.domain ∨ f.precision
```

§8.4 Safety Philosophy

Fail-safe behavior: Operations continue with defined (saturated/clamped) values rather than undefined behavior. The fault flag enables the caller to detect and handle the condition.

§9 Execution Timing

§9.1 Bounded Execution

All operations have bounded worst-case execution time (WCET):

Operation	Complexity	Bounded
fixed_mul	O(1)	✓
fx_matrix_mul	O(m×n×k)	✓ (dimensions known)
fx_conv2d	O(H×W×Kh×Kw)	✓ (dimensions known)
fx_maxpool	O(H×W)	✓
d_table_get	O(capacity) worst	✓ (capacity bounded)

§9.2 No Unbounded Loops

All loops have statically-known bounds derived from:

- Matrix dimensions (provided at initialization)
- Hash table capacity (fixed at creation)
- Kernel dimensions (fixed at model load)

§9.3 No Recursion

No recursive function calls. All algorithms are iterative.

§9.4 No Dynamic Allocation

Zero calls to malloc/free/realloc during inference. All memory provided by caller.

§10 Determinism Proofs

§10.1 Platform Independence Theorem

Statement: For any valid input state S and any two platforms A, B implementing the DVM specification:

```
inference(S) on A ≡ inference(S) on B
```

Proof sketch:

1. **Integer arithmetic identity:** All operations reduce to standard integer operations (+, -, ×, /, shifts) which are platform-independent for fixed-width integers.
2. **No floating-point:** All numerical computation uses int32_t or int64_t. No IEEE-754 operations.
3. **Deterministic rounding:** round_shift uses integer addition and arithmetic shift, both platform-independent.
4. **Memory layout independence:** Operations depend on values, not addresses.
5. **Iteration order determinism:** Hash table iteration follows insertion order, not hash order.

§10.2 Reproducibility Corollary

Given identical:

- Model weights (Q16.16)
- Input data (Q16.16)
- Network topology

The output is bit-identical across:

- x86_64, ARM64, RISC-V
- GCC, Clang, MSVC
- -O0, -O2, -O3, -Ofast
- Debug and release builds

Appendix A: Test Vectors

A.1 Fixed-Point Multiplication

```
Input: a = 0x00020000 (2.0), b = 0x00030000 (3.0)
Expected: 0x00060000 (6.0)

Input: a = 0x00018000 (1.5), b = 0x00028000 (2.5)
Expected: 0x0003C000 (3.75)

Input: a = 0x7FFF0000 (32767.0), b = 0x00020000 (2.0)
Expected: Overflow (saturate to FIXED_MAX)
```

A.2 Matrix Multiplication (2×2)

```
A = [1.0  2.0]    B = [5.0  6.0]    Expected C = [19.0  22.0]
    [3.0  4.0]        [7.0  8.0]                 [43.0  50.0]

Q16.16:
A.data = [0x10000, 0x20000, 0x30000, 0x40000]
B.data = [0x50000, 0x60000, 0x70000, 0x80000]
C.data = [0x130000, 0x160000, 0x2B0000, 0x320000]
```

A.3 Convolution (Identity Kernel)

```
Input (3×3):        Kernel (3×3):       Output (1×1):
[1  2  3]          [0  0  0]           [5]
[4  5  6]          [0  1  0]
[7  8  9]          [0  0  0]
```

Appendix B: Compliance Mapping

B.1 DO-178C Objectives

Objective	Section	Compliance
Requirements traceability	§1.4	✓ SRS mapping
Deterministic execution	§10	✓ Proved
Bounded resources	§9	✓ O(1) memory
WCET analysis	§9.1	✓ All bounded

B.2 IEC 62304 Objectives

Objective	Section	Compliance
Software architecture	§1-§7	✓ Documented
Risk analysis	§8	✓ Fault model
Unit verification	Appendix A	✓ Test vectors

B.3 ISO 26262 Objectives

Objective	Section	Compliance
ASIL capability	§8, §10	✓ Deterministic, fault-tolerant
Defensive programming	§8.4	✓ Saturation behavior
Traceability	§1.4	✓ Full chain

Revision History

Version	Date	Author	Changes
1.0.0	2026-01-20	William Murray	Initial release

Document Status: Final

Traceability: All code in `certifiable-inference/src/` SHALL reference this document via `@traceability CI-MATH-001 §N.N` comments.