AXILOG

SRS-005

DVM Arithmetic, Comparison & Mathematical Primitives

Locked v1.1 L1 [D1]

Last updated 27 March 2026
Supersedes SRS-005 v1.0
Depends on SRS-001 · DVEC-001 · DVM-SPEC-001
Changelog 1 entry

Contents

Depends on: SRS-001 v0.3 · DVEC-001 v1.3 · DVM-SPEC-001 v1.0-rc1

---

0. GOVERNING PRINCIPLE

> **L7 can certify policy and evidence only if L1 fixes the arithmetic semantics those policies and proofs depend on.**

Without mathematically closed L1 primitives:

- L4 velocity comparisons are semantically undefined
- L2 activation functions are implementation-dependent
- L3 scoring arithmetic is not reproducible
- L6 audit proofs rest on an unclosed substrate

This document closes L1.

---

1. PURPOSE

This document defines the **DVM Arithmetic, Comparison & Mathematical Primitives Specification** for the Axilog substrate (libaxilog).

It specifies:

- Q16.16 representation model
- Conversion semantics (integer ↔ Q16.16)
- Core arithmetic (add, sub, mul, div, negate, abs, min, max, clamp)
- Comparison and ordering
- Fault contract
- Transcendental approximations (exp, tanh)
- Lookup-table / polynomial rules
- Purity and boundedness
- Cross-platform identity
- Forbidden dependencies

Objective: Every arithmetic operation used by L2–L7 SHALL be mathematically total, bit-identical across platforms, and formally verifiable.

---

2. CONFORMANCE

A software component is conformant only if:

- It implements all applicable SHALL statements
- Each SHALL has a declared verification method
- All verification methods pass
- All associated DVEC-001 v1.3 constraints are satisfied

Non-conformance is a system integrity failure.

---

3. REQUIREMENT FORMAT

Each requirement is identified as:

```
SRS-005-SHALL-NNN
```

Each requirement includes:

- ID
- Statement
- Rationale (where applicable)
- Verification method

Allowed verification methods:

Method	Description
`test`	Deterministic unit test
`property test`	Property-based test over input domain
`static analysis`	Automated static analysis tool
`inspection`	Manual code or design review
`cross-platform harness`	certifiable-harness compatible golden reference
`golden vector`	Reference input/output vectors for bit-identity
`fault injection test`	Induced fault condition verification
`schema test`	Evidence schema conformance test
`RTM generation`	Requirements traceability matrix tooling

---

4. REPRESENTATION MODEL

4.1 Fixed-Point Type

SRS-005-SHALL-001

All deterministic arithmetic SHALL use Q16.16 fixed-point representation stored in `int32_t`.

Rationale: Q16.16 provides 16 bits of integer range [-32768, 32767] and 16 bits of fractional precision (1/65536 ≈ 0.0000153), sufficient for weights, activations, and policy thresholds.

Verification: inspection, static analysis

4.2 Unity Constant

SRS-005-SHALL-002

The value 1.0 SHALL be exactly 65536.

```
#define Q16_ONE  65536      /* 0x00010000 */
#define Q16_HALF 32768      /* 0x00008000 */
```

Verification: test, inspection

4.3 Representation Bounds

SRS-005-SHALL-003

The Q16.16 representation bounds SHALL be:

Constant	Value	Hexadecimal
`Q16_MAX`	INT32_MAX	0x7FFFFFFF
`Q16_MIN`	INT32_MIN	0x80000000
`Q16_EPS`	1	0x00000001

Verification: inspection, test

4.4 Two’s-Complement Requirement

SRS-005-SHALL-004

The implementation SHALL assume two’s-complement signed integer representation. The target compiler and platform MUST guarantee two’s-complement semantics.

Verification: inspection, static analysis, platform documentation

---

5. CONVERSION SEMANTICS

5.1 Integer to Q16.16 Conversion

SRS-005-SHALL-005

Integer-to-Q16.16 conversion SHALL use multiplication by 65536:

```
q16_16_t ax_int_to_q16(int16_t n, ct_fault_flags_t *faults)
{
    return (int32_t)n * Q16_ONE;
}
```

Constraint: Input SHALL be `int16_t` to guarantee no overflow.

Verification: property test, inspection

5.2 Q16.16 to Integer Conversion

SRS-005-SHALL-006

Q16.16-to-integer conversion SHALL use division by 65536 with truncation toward zero:

```
int32_t ax_q16_to_int(q16_16_t x, ct_fault_flags_t *faults)
{
    return x / Q16_ONE;  /* C99 truncation toward zero */
}
```

Verification: property test, inspection

5.3 Q16.16 to Integer with Rounding

SRS-005-SHALL-007

Q16.16-to-integer conversion with rounding SHALL use round-to-nearest-even:

```
int32_t ax_q16_to_int_rne(q16_16_t x, ct_fault_flags_t *faults)
{
    (void)faults;  /* No fault possible */

    int32_t base = x / Q16_ONE;              /* Integer part (truncates toward
zero) */
    int32_t frac = x - (base * Q16_ONE);     /* Fractional remainder */

    /* Handle negative fractional part */
    if (frac < 0) {
        frac = -frac;
    }

    if (frac > Q16_HALF) {
        /* Round away from zero */
        return (x >= 0) ? base + 1 : base - 1;
    } else if (frac < Q16_HALF) {
        /* Round toward zero */
        return base;
    } else {
        /* Tie: round to even */
        if (base & 1) {
            return (x >= 0) ? base + 1 : base - 1;
        }
        return base;
    }
}
```

Rationale: Uses division instead of right-shift to avoid implementation-defined behaviour for negative signed integers.

Verification: property test, golden vector

5.4 No Floating-Point Conversion

SRS-005-SHALL-008

No L1 primitive SHALL include a floating-point conversion path.

```
❌ (q16_16_t)(f * 65536.0f)
❌ (float)x / 65536.0f
```

Verification: static analysis, inspection

5.5 Decimal String Import

SRS-005-SHALL-009

Decimal string import (if provided) SHALL use deterministic fixed-point parsing with explicit rounding mode declaration and no floating-point intermediate.

Verification: property test, inspection

---

6. CORE ARITHMETIC

6.1 Saturated Addition

SRS-005-SHALL-010

Saturated addition SHALL widen operands to `int64_t`, compute the sum, and clamp to [Q16_MIN, Q16_MAX]:

```
q16_16_t ax_add_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    int64_t sum = (int64_t)a + (int64_t)b;

    if (sum > INT32_MAX) {
        faults->overflow = 1;
        return INT32_MAX;
    }
    if (sum < INT32_MIN) {
        faults->underflow = 1;
        return INT32_MIN;
    }
    return (int32_t)sum;
}
```

Verification: property test, fault injection test, cross-platform harness

---

6.2 Saturated Subtraction

SRS-005-SHALL-011

Saturated subtraction SHALL widen operands to `int64_t`, compute the difference, and clamp to [Q16_MIN, Q16_MAX]:

```
q16_16_t ax_sub_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    int64_t diff = (int64_t)a - (int64_t)b;

    if (diff > INT32_MAX) {
        faults→overflow = 1;
        return INT32_MAX;
    }
    if (diff < INT32_MIN) {
        faults→underflow = 1;
        return INT32_MIN;
    }
    return (int32_t)diff;
}
```

Verification: property test, fault injection test, cross-platform harness

---

6.3 Saturated Multiplication

SRS-005-SHALL-012

Saturated multiplication SHALL compute `(a × b) / 65536` using widened intermediate arithmetic with **truncation toward zero**:

```
q16_16_t ax_mul_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    int64_t prod = (int64_t)a * (int64_t)b;
    int64_t result = prod / Q16_ONE;  /* C99 truncation toward zero */

    if (result > INT32_MAX) {
        faults→overflow = 1;
        return INT32_MAX;
    }
    if (result < INT32_MIN) {
        faults→underflow = 1;
        return INT32_MIN;
    }
    return (int32_t)result;
}
```

Verification: property test, fault injection test, cross-platform harness

---

6.3.1 Multiplication Rounding Mode (Mandatory)

SRS-005-SHALL-067

All Q16.16 multiplication SHALL use truncation toward zero.

No alternative rounding mode is permitted in the core multiplication primitive. Applications requiring round-to-nearest-even multiplication SHALL use `ax_mul_q16_rne()` explicitly.

Rationale: Ensures bit-identical behaviour across all platforms and implementations. Truncation is the C99 default for integer division, eliminating any implementation-defined variance.

Verification: cross-platform harness, golden vector, inspection

---

6.4 Saturated Division

SRS-005-SHALL-013

Saturated division SHALL compute `(a × 65536) / b` using widened intermediate arithmetic and SHALL set `div_zero` if `b == 0`:

```
q16_16_t ax_div_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    if (b == 0) {
        faults->div_zero = 1;
        return 0;
    }

    int64_t num = (int64_t)a * Q16_ONE;
    int64_t result = num / (int64_t)b;

    if (result > INT32_MAX) {
        faults->overflow = 1;
        return INT32_MAX;
    }
    if (result < INT32_MIN) {
        faults->underflow = 1;
        return INT32_MIN;
    }
    return (int32_t)result;
}
```

Verification: property test, fault injection test, cross-platform harness

---

6.4.1 Division Overflow Edge Cases

SRS-005-SHALL-069

Division overflow cases, including `INT32_MIN / -1`, SHALL be treated as saturation with the `overflow` flag set.

Rationale: `INT32_MIN / -1` would produce `INT32_MAX + 1` which cannot be represented. This case SHALL saturate to `INT32_MAX` with the overflow flag set.

Verification: fault injection test, golden vector

6.4.2 No Signed Right-Shift for Division

SRS-005-SHALL-068

Right-shift operations on signed integers SHALL NOT be used to implement arithmetic division or scaling.

All scaling SHALL use explicit division or multiplication.

```
❌ int32_t base = x >> 16;
✅ int32_t base = x / Q16_ONE;
```

Rationale: Right-shift of negative signed integers is implementation-defined in C99 and violates cross-platform determinism (SRS-005-SHALL-054). Some compilers perform arithmetic shift (correct), others perform logical shift (incorrect for signed values).

Verification: static analysis, inspection, pre-commit scan

6.5 Negation

SRS-005-SHALL-014

Negation SHALL handle the asymmetric two's-complement range with saturation:

```
q16_16_t ax_neg_q16(q16_16_t x, ct_fault_flags_t *faults)
{
    if (x == INT32_MIN) {
        faults->overflow = 1;
        return INT32_MAX;
    }
    return -x;
}
```

Rationale: `-INT32_MIN` overflows in two's-complement.

Verification: property test, fault injection test

6.6 Absolute Value

SRS-005-SHALL-015

Absolute value SHALL handle INT32_MIN with saturation:

```
q16_16_t ax_abs_q16(q16_16_t x, ct_fault_flags_t *faults)
{
    if (x == INT32_MIN) {
        faults->overflow = 1;
        return INT32_MAX;
    }
    return (x < 0) ? -x : x;
}
```

Verification: property test, fault injection test

6.7 Minimum

SRS-005-SHALL-016

Minimum SHALL return the lesser of two values:

```
q16_16_t ax_min_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    (void)faults;  /* No fault possible */
    return (a < b) ? a : b;
}
```

Verification: property test

6.8 Maximum

SRS-005-SHALL-017

Maximum SHALL return the greater of two values:

```
q16_16_t ax_max_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults)
{
    (void)faults;  /* No fault possible */
    return (a > b) ? a : b;
}
```

Verification: property test

6.9 Clamp

SRS-005-SHALL-018

Clamp SHALL constrain a value to [min, max]:

```
q16_16_t ax_clamp_q16(q16_16_t x, q16_16_t lo, q16_16_t hi,
                      ct_fault_flags_t *faults)
{
    if (lo > hi) {
        faults→domain = 1;
        return lo;
    }
    if (x < lo) return lo;
    if (x > hi) return hi;
    return x;
}
```

Verification: property test, fault injection test

6.9.1 Clamp Invalid Bounds Behaviour

SRS-005-SHALL-070

If clamp bounds are invalid (`lo > hi`), the function SHALL:

1. Set the `domain` fault flag
2. Return `lo`
3. Perform no further evaluation

Rationale: Defines deterministic behaviour for invalid input. Returning `lo` provides a consistent, predictable result rather than undefined behaviour.

Verification: fault injection test, inspection

---

7. COMPARISON AND ORDERING

7.1 Signed Comparison

SRS-005-SHALL-019

All Q16.16 comparisons SHALL be signed, total, and bit-identical across platforms.

Verification: cross-platform harness, property test

7.2 No Floating-Point Comparison

SRS-005-SHALL-020

Comparison primitives SHALL NOT use floating-point conversion or host math libraries.

Verification: static analysis, inspection

7.3 Equality

SRS-005-SHALL-021

Equality SHALL be exact bit comparison:

```
bool ax_eq_q16(q16_16_t a, q16_16_t b)
{
    return a == b;
}
```

Verification: test, inspection

7.4 Less-Than

SRS-005-SHALL-022

Less-than SHALL use signed integer comparison:

```
bool ax_lt_q16(q16_16_t a, q16_16_t b)
{
    return a < b;
}
```

Verification: property test

7.5 Greater-Than

SRS-005-SHALL-023

Greater-than SHALL use signed integer comparison:

```
bool ax_gt_q16(q16_16_t a, q16_16_t b)
{
    return a > b;
}
```

Verification: property test

7.6 Less-Than-Or-Equal

SRS-005-SHALL-024

Less-than-or-equal SHALL use signed integer comparison:

```
bool ax_le_q16(q16_16_t a, q16_16_t b)
{
    return a ⩽ b;
}
```

Verification: property test

7.7 Greater-Than-Or-Equal

SRS-005-SHALL-025

Greater-than-or-equal SHALL use signed integer comparison:

```
bool ax_ge_q16(q16_16_t a, q16_16_t b)
{
    return a ⩾ b;
}
```

Verification: property test

7.8 Sign Classification

SRS-005-SHALL-026

Sign classification SHALL be deterministic:

```
typedef enum {
    AX_SIGN_NEGATIVE = -1,
    AX_SIGN_ZERO     =  0,
    AX_SIGN_POSITIVE =  1
} ax_sign_t;

ax_sign_t ax_sign_q16(q16_16_t x)
{
    if (x < 0) return AX_SIGN_NEGATIVE;
    if (x > 0) return AX_SIGN_POSITIVE;
    return AX_SIGN_ZERO;
}
```

Verification: property test

7.9 Total Ordering Guarantee

SRS-005-SHALL-027

The Q16.16 type SHALL define a total order over all 2^{32} values. No value is incomparable to any other.

Verification: inspection, property test

7.10 Monotonicity Classification

SRS-005-SHALL-028

All L1 functions SHALL declare their monotonicity classification in their documentation. The allowed classifications are:

Classification	Definition
`MONOTONE_INCREASING`	∀ a < b: f(a) ≤ f(b)
`MONOTONE_DECREASING`	∀ a < b: f(a) ≥ f(b)
`NON_MONOTONIC`	Neither of the above

Required declarations:

Function	Monotonicity
`ax_add_q16(a, k)` (fixed k)	MONOTONE_INCREASING
`ax_sub_q16(a, k)` (fixed k)	MONOTONE_INCREASING
`ax_mul_q16(a, k)` (k > 0)	MONOTONE_INCREASING
`ax_mul_q16(a, k)` (k < 0)	MONOTONE_DECREASING
`ax_neg_q16`	MONOTONE_DECREASING
`ax_abs_q16`	NON_MONOTONIC
`ax_exp_q16`	MONOTONE_INCREASING
`ax_tanh_q16`	MONOTONE_INCREASING

Verification: inspection, documentation audit

---

8. FAULT CONTRACT

8.1 Fault Context Acceptance

SRS-005-SHALL-029

Every public L1 primitive SHALL accept `ct_fault_flags_t *faults`.

Verification: static analysis, inspection

8.2 Arithmetic Fault Types

SRS-005-SHALL-030

The fault context SHALL include at minimum:

```
typedef struct {
    uint8_t overflow;    /* Result exceeded Q16_MAX */
    uint8_t underflow;   /* Result exceeded Q16_MIN */
    uint8_t div_zero;    /* Division by zero */
    uint8_t domain;      /* Input outside valid domain */
    uint8_t protocol;    /* Reserved: higher-layer integration (L5+) */
} ct_fault_flags_t;
```

Note on `protocol` field: The `protocol` flag is reserved for higher-layer integration (L5 agent state machines, L4 policy violations). L1 primitives SHALL NOT set this flag. It is included in the L1 struct definition to ensure a single fault type across all layers.

Verification: inspection

8.3 Saturation Fault Signalling

SRS-005-SHALL-031

Any arithmetic saturation SHALL set `overflow` or `underflow` as applicable.

Verification: fault injection test, property test

8.4 Totality Requirement

SRS-005-SHALL-032

All L1 primitives SHALL be total functions over their declared domain. No undefined output for any valid input.

Verification: property test, inspection

8.5 Purity Requirement

SRS-005-SHALL-033

All L1 primitives SHALL be pure, reentrant, and free of global mutable state.

Verification: static analysis, inspection

8.6 No Silent Failure

SRS-005-SHALL-034

No L1 primitive SHALL return a value without setting the appropriate fault flag if a fault condition occurred.

Verification: fault injection test, inspection

8.7 Fault Propagation

SRS-005-SHALL-035

Fault flags SHALL propagate: once set, they SHALL remain set until explicitly cleared by the caller.

Verification: property test

---

9. BOUNDEDNESS AND MEMORY

9.1 Bounded Time

SRS-005-SHALL-036

All L1 primitives SHALL execute in bounded time O(1).

Verification: static analysis, WCET analysis, inspection

9.2 Bounded Space

SRS-005-SHALL-037

All L1 primitives SHALL execute in bounded stack space. No recursion. No unbounded loops.

Verification: static analysis, `-fstack-usage`, inspection

9.3 No Dynamic Allocation

SRS-005-SHALL-038

No L1 primitive SHALL perform dynamic allocation (malloc, calloc, realloc, free).

Verification: static analysis, pre-commit scan

9.4 No Global Mutable State

SRS-005-SHALL-039

No L1 primitive SHALL read from or write to global mutable state.

Verification: static analysis, inspection

---

10. TRANSCENDENTAL APPROXIMATIONS

10.1 Exponential Function

SRS-005-SHALL-040

`ax_exp_q16` SHALL be implemented via a fixed polynomial approximation or CORDIC algorithm with frozen coefficients.

Verification: golden vector, cross-platform harness

10.2 Exponential Domain

SRS-005-SHALL-041

`ax_exp_q16` SHALL declare and enforce a finite valid input domain. Inputs outside this domain SHALL set `faults→domain` and return a defined saturated value.

Suggested domain: [-11.0, 11.0] in Q16.16 (approximately [-720896, 720896]).

Rationale: exp(11) ≈ 59874 fits in Q16.16; exp(12) ≈ 162755 overflows the integer portion.

Verification: property test, fault injection test, inspection

---

10.3 Exponential Accuracy

SRS-005-SHALL-042

`ax_exp_q16` SHALL produce results within ±2 ULP of the correctly rounded Q16.16 value over its valid domain.

Verification: golden vector, property test

10.4 Hyperbolic Tangent

SRS-005-SHALL-043

`ax_tanh_q16` SHALL be implemented via:

- The identity `tanh(x) = (exp(2x) - 1) / (exp(2x) + 1)`, OR
- A bit-perfect lookup table with linear interpolation, OR
- A fixed polynomial approximation

Verification: golden vector, cross-platform harness

10.5 Hyperbolic Tangent Domain

SRS-005-SHALL-044

`ax_tanh_q16` SHALL accept all Q16.16 values as input.

Rationale: tanh saturates smoothly; output is always in [-1, 1].

Verification: property test

10.6 Hyperbolic Tangent Accuracy

SRS-005-SHALL-045

`ax_tanh_q16` SHALL produce results within ±2 ULP of the correctly rounded Q16.16 value.

Verification: golden vector, property test

10.7 Sigmoid Function (Optional)

SRS-005-SHALL-046

If provided, `ax_sigmoid_q16` SHALL be implemented as:

```
sigmoid(x) = 1 / (1 + exp(-x))
```

using `ax_exp_q16` internally.

Verification: golden vector, cross-platform harness

10.8 Trigonometric Functions (Deferred)

SRS-005-SHALL-047

Trigonometric functions (`sin` , `cos`) SHALL NOT be included in L1 v1.0 unless a concrete L2 requirement justifies their inclusion.

Rationale: Reduces surface area and proof burden. exp/tanh are immediately relevant for inference; sin/cos are not.

Verification: inspection

---

11. LOOKUP-TABLE AND POLYNOMIAL RULES

11.1 Fixed Coefficients

SRS-005-SHALL-048

Any polynomial approximation SHALL use fixed coefficients frozen at compile time.

Verification: inspection, static analysis

11.2 Fixed Table Contents

SRS-005-SHALL-049

Any lookup table SHALL have fixed contents frozen at compile time.

Verification: inspection, static analysis

11.3 Compile-Time Generation

SRS-005-SHALL-050

Lookup tables and polynomial coefficients SHALL be generated at compile time or build time, not at runtime.

Verification: inspection, build system audit

11.4 No Runtime Synthesis

SRS-005-SHALL-051

No L1 primitive SHALL synthesise tables or coefficients at runtime.

Verification: static analysis, inspection

11.5 Endianness Neutrality

SRS-005-SHALL-052

Lookup tables stored in binary format SHALL be endianness-neutral or SHALL include compile-time byte-swapping for the target platform.

Verification: cross-platform harness, inspection

11.6 Table Provenance

SRS-005-SHALL-053

The generation method for any lookup table SHALL be documented, reproducible, and frozen. The generation script SHALL be included in the repository.

Verification: inspection, RTM generation

---

12. CROSS-PLATFORM IDENTITY

12.1 Platform Equivalence

SRS-005-SHALL-054

Identical inputs to any L1 primitive SHALL produce identical outputs and identical fault bits across supported architectures (x86_64, ARM64, RISC-V).

Verification: cross-platform harness, golden vector

12.2 Golden Vector Provision

SRS-005-SHALL-055

The implementation SHALL provide golden vectors for all arithmetic primitives covering:

- Normal operation
- Boundary values (0, ±1, Q16_MAX, Q16_MIN, Q16_EPS)
- Saturation conditions
- Fault conditions

Verification: golden vector suite

12.3 Cross-Platform Identity Tests

SRS-005-SHALL-056

The implementation SHALL provide cross-platform identity tests for all advanced primitives (exp, tanh, sigmoid).

Verification: CI harness, certifiable-harness integration

12.4 Compiler Independence

SRS-005-SHALL-057

The implementation SHALL produce identical results when compiled with GCC, Clang, and MSVC on each supported platform.

Verification: CI matrix, cross-platform harness

---

13. FORBIDDEN DEPENDENCIES

13.1 No libm

SRS-005-SHALL-058

No L1 primitive SHALL depend on `libm` or any host math library.

Verification: static analysis, linker audit

13.2 No Floating-Point Hardware State

SRS-005-SHALL-059

No L1 primitive SHALL depend on floating-point hardware state (rounding mode, exception flags, MXCSR, FPCR).

Verification: static analysis, inspection

13.3 No Locale Dependence

SRS-005-SHALL-060

No L1 primitive SHALL depend on locale settings.

Verification: static analysis, inspection

13.4 No System Clock

SRS-005-SHALL-061

No L1 primitive SHALL depend on system clock or wall time.

Verification: static analysis, pre-commit scan

13.5 No Randomness

SRS-005-SHALL-062

No L1 primitive SHALL depend on randomness or PRNG state.

Verification: static analysis, inspection

13.6 No External I/O

SRS-005-SHALL-063

No L1 primitive SHALL perform external I/O (file, network, console).

Verification: static analysis, inspection

---

14. HEADER TEMPLATE

14.1 Compliance Header

SRS-005-SHALL-064

Every L1 header SHALL declare:

```
/**
 * @file ax_arith.h
 * @brief DVM Q16.16 Arithmetic Primitives
 *
 * DVEC: v1.3
 * DETERMINISM: D1 — Strict Deterministic
 * MEMORY: Zero Dynamic Allocation
 * SRS: SRS-005
 *
 * Copyright © 2026 The Murray Family Innovation Trust
 * Patent: UK GB2521625.0
 */
```

Verification: static analysis, inspection

14.2 Function-Level Traceability

SRS-005-SHALL-065

Every public L1 function SHALL include SRS anchors:

```
/**
 * @brief Saturated Q16.16 addition.
 *
 * SRS-005-SHALL-010: Saturated addition
 * SRS-005-SHALL-029: Fault context acceptance
 * SRS-005-SHALL-031: Saturation fault signalling
 * SRS-005-SHALL-054: Cross-platform identity
 */
q16_16_t ax_add_q16(q16_16_t a, q16_16_t b, ct_fault_flags_t *faults);
```

Verification: RTM generation, static analysis

---

15. TYPE DEFINITIONS

15.1 Core Types Header

SRS-005-SHALL-066

The implementation SHALL provide `include/axilog/types.h`:

```
/**
 * @file types.h
 * @brief DVM Core Types
 *
 * DVEC: v1.3
 * DETERMINISM: D1 — Strict Deterministic
 * MEMORY: Zero Dynamic Allocation
 * SRS: SRS-005
 */

#ifndef AXILOG_TYPES_H
#define AXILOG_TYPES_H

#include <stdint.h>
#include <stdbool.h>

/**
 * @brief Q16.16 fixed-point type.
 * SRS-005-SHALL-001, SRS-005-SHALL-002, SRS-005-SHALL-003
 */
typedef int32_t q16_16_t;

#define Q16_SHIFT  16
#define Q16_ONE    (1 << Q16_SHIFT)       /* 65536 = 0x00010000 */
#define Q16_HALF   (1 << (Q16_SHIFT - 1)) /* 32768 = 0x00008000 */
#define Q16_MAX    INT32_MAX              /* 0x7FFFFFFF */
#define Q16_MIN    INT32_MIN              /* 0x80000000 */
#define Q16_EPS    1                      /* 0x00000001 */

/**
 * @brief Fault flags for arithmetic operations.
 * SRS-005-SHALL-029, SRS-005-SHALL-030
 *
 * Note: The 'protocol' field is reserved for higher-layer integration
 * (L5 agent state machines, L4 policy violations). L1 primitives SHALL
 * NOT set this flag. It is included here to ensure a single fault type
 * across all layers.
 */
typedef struct {
    uint8_t overflow;    /* Result exceeded Q16_MAX */
    uint8_t underflow;   /* Result exceeded Q16_MIN */
    uint8_t div_zero;    /* Division by zero */
    uint8_t domain;      /* Input outside valid domain */
    uint8_t protocol;    /* Reserved: higher-layer integration (L5+) */
} ct_fault_flags_t;

/**
 * @brief Check if any fault flag is set.
 * SRS-005-SHALL-034
```

```
 */
static inline bool ct_fault_any(const ct_fault_flags_t *f)
{
    return f->overflow || f->underflow || f->div_zero ||
           f->domain || f->protocol;
}

/**
 * @brief Sign classification.
 * SRS-005-SHALL-026
 */
typedef enum {
    AX_SIGN_NEGATIVE = -1,
    AX_SIGN_ZERO     =  0,
    AX_SIGN_POSITIVE =  1
} ax_sign_t;

#endif /* AXILOG_TYPES_H */
```

Verification: inspection

---

16. VERIFICATION MATRIX

Requirement	Function(s)	Verification
SRS-005-SHALL-001	All	inspection, static analysis
SRS-005-SHALL-010	`ax_add_q16`	property test, fault injection, golden vector
SRS-005-SHALL-011	`ax_sub_q16`	property test, fault injection, golden vector
SRS-005-SHALL-012	`ax_mul_q16`	property test, fault injection, golden vector
SRS-005-SHALL-013	`ax_div_q16`	property test, fault injection, golden vector
SRS-005-SHALL-014	`ax_neg_q16`	property test, fault injection
SRS-005-SHALL-015	`ax_abs_q16`	property test, fault injection
SRS-005-SHALL-019–028	`ax_eq/lt/gt/le/ge/sign_q16`	property test
SRS-005-SHALL-040–042	`ax_exp_q16`	golden vector, cross-platform harness
SRS-005-SHALL-043–045	`ax_tanh_q16`	golden vector, cross-platform harness
SRS-005-SHALL-054–057	All	cross-platform harness, CI matrix
SRS-005-SHALL-058–063	All	static analysis, linker audit

17. L1 SUBSTRATE MAPPING

By locking these L1 requirements, the Axioma framework achieves **Full Mathematical Closure**:

Logic Layer	Dependent Metric	L1 Requirement
L4 Policy	Velocity > Threshold	SRS-005-SHALL-019 (Signed Comparison)
L4 Policy	Value within bounds	SRS-005-SHALL-018 (Clamp)
L3 Oracle	Temperature scaling	SRS-005-SHALL-012 (Saturated Mul)
L2 Inference	Softmax/Activation	SRS-005-SHALL-040 (Deterministic Exp)
L2 Inference	tanh activation	SRS-005-SHALL-043 (Deterministic Tanh)
L6 Audit	Q16.16 in evidence	SRS-005-SHALL-001 (Representation)
L7 Governance	Proof-carrying policy	All comparison SHALLs

18. PUBLIC API SUMMARY

Function	SRS Coverage
`ax_add_q16`	SRS-005-SHALL-010, 029, 031, 054
`ax_sub_q16`	SRS-005-SHALL-011, 029, 031, 054
`ax_mul_q16`	SRS-005-SHALL-012, 029, 031, 054
`ax_div_q16`	SRS-005-SHALL-013, 029, 031, 054
`ax_neg_q16`	SRS-005-SHALL-014, 029, 031
`ax_abs_q16`	SRS-005-SHALL-015, 029, 031
`ax_min_q16`	SRS-005-SHALL-016, 029
`ax_max_q16`	SRS-005-SHALL-017, 029
`ax_clamp_q16`	SRS-005-SHALL-018, 029, 031
`ax_eq_q16`	SRS-005-SHALL-021
`ax_lt_q16`	SRS-005-SHALL-022, 019
`ax_gt_q16`	SRS-005-SHALL-023, 019
`ax_le_q16`	SRS-005-SHALL-024, 019
`ax_ge_q16`	SRS-005-SHALL-025, 019
`ax_sign_q16`	SRS-005-SHALL-026
`ax_int_to_q16`	SRS-005-SHALL-005
`ax_q16_to_int`	SRS-005-SHALL-006
`ax_q16_to_int_rne`	SRS-005-SHALL-007
`ax_exp_q16`	SRS-005-SHALL-040, 041, 042, 054
`ax_tanh_q16`	SRS-005-SHALL-043, 044, 045, 054
`ax_sigmoid_q16`	SRS-005-SHALL-046 (optional)

19. CLOSURE ASSESSMENT

Requirement Category	Status
Representation model	✅ Closed
Conversion semantics	✅ Closed
Core arithmetic	✅ Closed
Comparison and ordering	✅ Closed
Fault contract	✅ Closed
Transcendental approximations	✅ Closed (exp, tanh)
Lookup-table rules	✅ Closed
Purity and boundedness	✅ Closed
Cross-platform identity	✅ Closed
Forbidden dependencies	✅ Closed

Open Items (Deferred to L1 v1.1):

- Trigonometric functions (sin, cos) — pending L2 justification
- Extended precision accumulator (Q32.32) — if L2 requires

20. CI GATE REQUIREMENTS

The following CI gates SHALL pass before merge:

```
□ property-test-arith     — All arithmetic property tests
□ property-test-compare   — All comparison property tests
□ golden-vector           — All golden reference vectors match
□ fault-injection         — All fault paths exercised
□ cross-platform-x86      — x86_64 bit-identity
□ cross-platform-arm      — ARM64 bit-identity
□ static-analysis         — Zero warnings (clang-tidy, cppcheck)
□ forbidden-pattern-scan  — No float/double/malloc/time
□ signed-shift-scan       — No >> on signed integers (SRS-005-SHALL-068)
□ rtm-coverage            — All SHALLs traced to code
```

---

21. FINAL STATEMENT

> **The L1 substrate is not a convenience layer.**
>
> **It is the mathematical foundation upon which all higher-layer proofs rest.**

If L1 is not closed, L7 governance is meaningless.

This document closes L1.

---

22. REVISION HISTORY

Version	Date	Author	Changes
1.0-Frozen	2026-03-27	William Murray	Initial frozen release — 66 SHALL statements. Complete mathematical closure for L1 substrate.
1.1-Frozen	2026-03-27	William Murray	Audit closure — 4 new SHALLs (067–070). Fixed multiplication rounding ambiguity, removed signed right-shift dependency, added division overflow rule, formalised clamp invalid bounds behaviour, tightened monotonicity classification. Total: 70 SHALL statements.

23. DOCUMENT APPROVAL

Role	Name	Date	Signature
Author	William Murray	2026-03-27	
Reviewer			
Approver			

Appendix A — Golden Vector Examples

A.1 Addition

a (Q16.16)	b (Q16.16)	Expected	Fault
65536	65536	131072	none
INT32_MAX	1	INT32_MAX	overflow
INT32_MIN	-1	INT32_MIN	underflow
0	0	0	none
-65536	65536	0	none

A.2 Multiplication

a (Q16.16)	b (Q16.16)	Expected	Fault
65536 (1.0)	65536 (1.0)	65536 (1.0)	none
131072 (2.0)	131072 (2.0)	262144 (4.0)	none
INT32_MAX	2	INT32_MAX	overflow
32768 (0.5)	32768 (0.5)	16384 (0.25)	none

A.3 Division

a (Q16.16)	b (Q16.16)	Expected	Fault
65536 (1.0)	65536 (1.0)	65536 (1.0)	none
131072 (2.0)	65536 (1.0)	131072 (2.0)	none
65536 (1.0)	0	0	div_zero
INT32_MIN	-1	INT32_MAX	overflow
INT32_MAX	1	INT32_MAX	overflow

A.4 Exponential

x (Q16.16)	Expected (approx)	Fault
0	65536 (1.0)	none
65536 (1.0)	178145 (2.718)	none
-65536 (-1.0)	24109 (0.368)	none
720896 (11.0)	near Q16_MAX	none
786432 (12.0)	saturated	domain

Appendix B — Alignment with DVM-SPEC-001

DVM-SPEC-001 Section	SRS-005 Coverage
§3 Fixed-Point Model	SRS-005-SHALL-001–004
§4 Arithmetic Primitives	SRS-005-SHALL-010–018, 067–070
§5 Fault Model	SRS-005-SHALL-029–035
§6 Oracle Boundary	(L3 concern, not L1)
§7 Commitment	(L6 concern, not L1)

Appendix C — DVEC-001 v1.3 Conformance

DVEC Mandate	SRS-005 Implementation
§1.1 Bit-Identity	SRS-005-SHALL-054, 067, 068
§1.2 Forbidden Constructs	SRS-005-SHALL-058–063, 068
§2.1 Total Function	SRS-005-SHALL-032
§2.4 Fault Propagation	SRS-005-SHALL-029–035, 069, 070
§8.1 Minimal Footprint	SRS-005-SHALL-036–039
§12.2 Arithmetic Enforcement	All arithmetic SHALLs

Appendix D — v1.0 → v1.1 Audit Record

Finding	Severity	Resolution
Multiplication rounding mode under-specified	BLOCKING	SRS-005-SHALL-067: Mandates truncation toward zero
Signed right-shift is implementation-defined in C99	BLOCKING	SRS-005-SHALL-068: Prohibits signed right-shift for division
Division overflow edge case (INT32_MIN / -1) implicit	HIGH	SRS-005-SHALL-069: Explicit saturation rule
Clamp invalid bounds behaviour ambiguous	HIGH	SRS-005-SHALL-070: Formalised domain fault behaviour
Monotonicity requirement mathematically weak	MEDIUM	SRS-005-SHALL-028: Tightened to explicit classification
Protocol flag in fault struct is L5+ concern	LOW	Documentation: Explicitly marked as reserved for higher layers
RNE conversion used right-shift on negative values	BLOCKING	SRS-005-SHALL-007: Rewritten to use division

Audit Source: External review (GPT-4o, Gemini 2.5 Pro)

Audit Date: 27 March 2026

Audit Verdict: All blocking and high-severity findings resolved. Document status elevated to **Audit-Frozen FINAL**.

SRS-005 v1.1-Frozen — Audit-Frozen FINAL William Murray · SpeyTech · March 2026 Patent GB2521625.0
