HARDWARE INTEGRATION OF NFNS AND SBNS NUMERAL SYSTEMS IN A FAULT-TOLERANT NEUROMORPHIC PROCESSOR ARCHITECTURE

Andrey V. Voron (anvoron1@yandex.ru)

November 11, 2026

Abstract

This paper investigates the microarchitectural principles and technological prospects of designing a fundamentally new class of computing devices – a hybrid fault-tolerant neuromorphic processor. The feasibility of deep hardware integration of the New Fibonacci Numeral System (NFNS) and the Scaled Bergman Numeral System (SBNS) on a single semiconductor die is substantiated. It is demonstrated that due to the identical nature of their underlying mathematical principles (representation of weights via fractions, Fibonacci confluent/convoluted rules), both systems can share up to 85–90% of the physical area of the arithmetic logic unit (ALU) and utilize a unified hardware parity filter. A mechanism for dynamic on-the-fly switching of the computational basis within a single clock cycle via a configuration register depending on the specific application task is described. A technological forecast is formulated, according to which the integer NFNS mode ensures noise immunity and error control (bit-flips) in critical discrete environments (aerospace, cryptography), while the SBNS mode, leveraging the Truncation Symmetry Theorem, eliminates the accumulation of rounding errors and optimizes the bit-width of the significant register mantissa, serving as an ideal foundation for neuromorphic AI accelerators.

1 Introduction

Redundant positional numeral systems with irrational bases have been investigated at various times as a fundamental alternative to the binary basis [1, 2, 3]. The direct mathematical foundation for overcoming the limitations of classical logic was laid in the works of A.P. Stakhov on Fibonacci computer arithmetic [2, 3] and dates back to George Bergman's seminal concept of a numeral system with an irrational base [1]. However, the classical Bergman system and traditional Fibonacci codes suffered from significant drawbacks: chaotically varying redundancy, the "wandering carry" effect, and the problem of an "infinite information tail" when representing integers.

To overcome these limitations, the author previously developed and formalized two mathematical frameworks: the New Fibonacci Numeral System (NFNS) with a dual basis for noise-immune environments [4, 5], and the Scaled Bergman Numeral System (SBNS), which departs from the abstract unit base in favor of a central scale element – a weight scaling factor $k = 0.2$ [6].

The present work is dedicated to researching the principles of deep microarchitectural integration of the NFNS and SBNS systems at the single-chip level. The objective of this paper is to substantiate the possibility of constructing a universal fault-tolerant neuromorphic processor whose ALU is capable of dynamically switching the computational basis on the fly to achieve maximum security or extreme energy efficiency for AI algorithms depending on the nature of the tasks being executed.

2 The main part

For AI and continuous signal processing, SBNS is a more "powerful" tool than the integer-based NFNS due to the following reasons:

1. Natural Solution to the Floating-Point Problem for AI. Modern AI (neural networks, deep learning) relies on the continuous mathematics of real numbers (weights, probabilities, gradients). The binary system wastes colossal hardware resources approximating real numbers (FP32, FP16, INT8 formats). NFNS operates on integers from the Fibonacci sequence and is ideal for discrete, integer-based, ultra-reliable computations (e.g., onboard aerospace computers, cryptography). SBNS is inherently built on the irrational base of the golden ratio φ. This means that continuous processes are encoded harmoniously at the hardware level without artificial discretization.

2. Lifting the Curse of the "Infinite Fractional Tail" via Scale $k = 0.2$. The traditional Bergman code (with base $\varphi^0 = 1$) was impractical because the mundane representation of integers generated infinite fractions. Shifting the basis by a factor of 0.2 (the fraction 1/5) is an optimal engineering solution. Terminal fives and "mirror pairs" (e.g., ...5955 and ...9775) at the 10th–12th decimal places mutually cancel each other out during addition. The SBNS register features 54 bits (configurable); computational accuracy is maintained within a few least significant weights of the grid ($\sim 10^{-6}$). Bulky 64-bit coprocessors are no longer required.

3. The Truncation Symmetry Theorem [6] as a Substitute for Noise Smoothing. In AI architectures, number rounding (model quantization) always leads to the accumulation of errors (fractional noise), degrading prediction accuracy. The Truncation Symmetry Theorem [6] changes the rules. In the SBNS grid, truncation errors of a working pair of codes (Code A and Code B) are invariant and mutually compensate for each other, returning the result to the exact value. For AI, this implies computing without the accumulation of rounding errors beyond the resolution capacity of the grid.

4. Dynamic Bit-Width and Dense Data Streams. An algorithm is described where the processor understands in advance, based on the exponent value, how many digits the result will occupy and physically truncates unused memory cells. Neural networks contain billions of zero or low-significance weights. Utilizing SBNS would allow data to be packed into dense bit arrays without metadata and radix points, reducing the energy consumption of AI chips multi-fold.

Full utilization of both numeral systems (SBNS and NFNS) is possible on a single physical processor. From a microarchitectural perspective, this represents the most economically viable engineering solution. Since the foundational principles (representation of weights via fractions, rules for Fibonacci confluences/convolutions, and, crucially, the hardware parity filter) are identical in both systems, they can share over 85–90% of the entire physical area of the processor's arithmetic logic unit (ALU). Switching between systems on a single processor can be implemented via a hardware-software mechanism, dynamically adapting to current tasks.

How does this work in practice?

The processor can switch its mathematical basis on the fly (literally within a single clock cycle) by changing the value of a control flag in the configuration register (e.g., REG_MODE: 0 = NFNS, 1 = SBNS).

Scenario No. 1: AI and Signal Processing Mode Active (SBNS). The processor performs matrix multiplication, deep neural network training, or audio/video signal filtering. What does the processor do? The control flag switches the ALU to SBNS mode. The hardware parity filter ensures rapid bitwise comparison. During non-linear operations, a scale correction block (multiplier by 1/5) is automatically engaged. As a result, fractional noises and rounding errors mutually eliminate each other, allowing the neural network to deliver maximum computational accuracy with a smaller bit-width.

Scenario No. 2: Mission-Critical Mode or Data Transmission Over a Noisy Channel Active (NFNS). The system transmits telemetry, executes cryptographic operations, or processes control commands for an unmanned vehicle under radiation or electromagnetic interference. What does the processor do? The flag instantly switches the ALU to NFNS mode. The scale correction block is bypassed (the signal goes around it). Now, the exact same flip-flops and parity filter operate for strict redundancy control of the discrete Fibonacci code. As a result, single upsets (bit-flips) are detected by the filter and dual-form control at the 3rd stage.

To confirm the closure of the developed mathematical framework and to verify the microarchitectural blocks, a software model of the ALU named "VORON-NFNS/SBNS" was designed in Python 3.10+ using the Decimal library. Within a single structural class, the model implements a hardware state matrix of allowed codes, Voron's syntactic parity filter, and end-to-end algorithms for invariant splitting of "twos." It was experimentally verified that the ALU can switch its computational basis within one conventional clock cycle (REG_MODE). Bitwise arithmetic in SBNS mode demonstrated the operation of the Truncation Symmetry Theorem: fractional noise at the 10th–12th digits is mutually compensated, ensuring matrix operations are executed without accumulating quantization errors. The simulator code is fully verified and provided in APPENDIX A.

Architectural Advantages of the "Dual-System" Processor.

1. Maximum Hardware Utilization. Registers, adders, shift circuits, and the logic of the hardware parity filter never sit idle. They operate in both modes; only the interpretation of the codes and the logic of the final stage of non-linear operations change.

2. Flexibility for Software Developers. The programmer decides at which point to call the transition instruction to SBNS for heavy mathematics, and when to return the processor to the reliable NFNS mode.

3. Energy Efficiency. The mechanism of automatically halting the division cycle when the remainder reaches zero and the dynamic bit-width will equally and effectively save energy (by cutting off empty register cells) in both modes of chip operation.

Thus, the mathematical foundation of NFNS and SBNS enables the creation of a unique hybrid fault-tolerant neuromorphic processor that has no analogs in classical binary architecture.

Technological Forecast for NFNS and SBNS.

Each system addresses pain points of computing technology, but in entirely different segments. Below is a forecast of task distribution based on efficiency for NFNS and SBNS.

1. Forecast for NFNS (New Fibonacci Numeral System).

The primary advantages are noise immunity, instant hardware data corruption control (bit-flips), and cryptographic resilience.

Most suitable tasks and industries:

1. Aerospace Engineering and Space Exploration. Cosmic radiation causes bit-flips in silicon. NFNS will detect a radiation-induced fault directly at the 3rd stage of an arithmetic operation, preventing a corrupted bit from distorting a trajectory or command.

2. Autonomous Vehicles. In critical systems (autopilots, medical robots), software-based error-checking delays can cost lives. NFNS is ideal for logical cores of controllers managing braking, steering, and collision avoidance systems.

3. High-Risk Industrial Automation (ICS/SCADA). Programmable logic controllers for nuclear power plants, chemical production facilities, and train control systems operating under severe electromagnetic interference.

2. Forecast for SBNS (Scaled Bergman Numeral System).

The primary advantages are harmonious encoding of continuous values, absence of rounding error accumulation, and hardware data compression.

Most suitable tasks and industries:

1. Training Large Neural Networks. Thanks to the Truncation Symmetry Theorem, SBNS chips can perform matrix multiplications without loss of precision while holding weights in short registers, thereby reducing the energy costs of data centers.

2. Edge AI and Mobile Processors. Local AI is heavily constrained by thermal limits and battery capacity. The algorithm for truncating empty memory cells when working with sparse weight matrices will allow large models to run locally on wearable devices.

3. Digital Signal Processing (DSP). Streaming processing of audio, video, and radio signals, where binary roundings typically create digital noise.

4. Analog-to-Digital and Optical (Photonic) Computing. Light and analog signals attenuate harmoniously; their physics align closely with the golden ratio.

3 Conclusion

1. Within a unified semiconductor architecture, the principles of hardware integration of two numerical bases – the New Fibonacci Numeral System (NFNS) and the Scaled Bergman Numeral System (SBNS) – have been theoretically substantiated and described. It is shown that NFNS is optimal as a numerical basis for noise-immune, fault-tolerant (bit-flip protected), and cryptographically secure discrete systems, whereas SBNS is optimal for analog-to-digital, neuromorphic, and optical AI processors. The practical feasibility and mathematical closure of the developed hybrid architecture were experimentally proven via end-to-end computer modeling. The created software core of the ALU confirmed the ability of a single set of flip-flops to perform addition, subtraction, division, scaled multiplication, and neural network layer emulation within a strictly bounded bit-width grid.

2. The presence of a shared mathematical foundation – the invariance of Fibonacci confluence/convolution rules, the representation of digit weights via fractions, and the end-to-end operation of the parity rule developed by the author – opens up the possibility of reusing 85% to 90% of the physical area of the ALU. A single hardware parity filter successfully solves two engineering challenges: it provides instantaneous integrity verification of integer codes in NFNS mode and guarantees rapid, unambiguous bitwise comparison of the remainder during division in SBNS mode.

3. The introduction of a weight scaling factor $k = 0.2$ eliminated the core issue of the Bergman system – the occurrence of infinite information tails. The Truncation Symmetry Theorem allows for the optimization of the bit-width of physical registers for AI computations. The process of mutual cancellation of fractional noises and rounding errors at the register flip-flop level replaces energy-expensive software-hardware methods of quantization and smoothing in training models.

4. The proposed algorithm for dynamic on-the-fly basis switching (within a single clock cycle) via the REG_MODE control flag allows for a departure from chip specialization. A universal dual system is formed on a single die. While the NFNS mode acts as a numerical basis for reliable, fault-tolerant discrete systems (aerospace onboard computers, cryptographic processors), the SBNS mode provides a ready foundation for energy-efficient neuromorphic, analog-to-digital, and optical artificial intelligence accelerators.

References

[1] Bergman G. A Number System with an Irrational Base. Mathematics Magazine, 1957, vol. 31, no. 2, pp. 98–110.

[2] Stakhov A.P. Introduction to the Algorithmic Theory of Measurement. Moscow, Sovetskoye Radio Publ., 1977. 288 p. (In Russian)

[3] Stakhov A.P. Codes of the Golden Proportion. Moscow, Radio i Svyaz Publ., 1984. 152 p. (In Russian)

[4] Voron A.V. Fibonacci Code Redundancy and the Design of a New Fibonacci Numeral System – NFNS. viXra e-Print archive, 2026, viXra:2607.0114. Available at: vixra.org.

[5] Voron A.V. Development of a Redundant Fibonacci Numeral System (NFNS) With a Dual Basis for Fault-tolerant Computing Environments. viXra e-Print archive, 2026, viXra:2609.0053. Available at: vixra.org.

[6] Voron A.V. Scaled Bergman Number System (SBNS) and the Voron Filter as a New Basis for Computer Technology. viXra e-Print archive, 2026, viXra:2610.0069. Available at: vixra.org.

APPENDICES

APPENDIX A

```
# voron_final_hybrid_processor.py
# Verification simulator of the hybrid ALU "VORON-NFNS/SBNS"
from decimal import Decimal, getcontext
getcontext().prec = 60
VERIFY_TOL_LSB = 3  # Verification tolerance for the ALU result, in LSB steps

class VoronHybridProcessorALU:
    def __init__(self, register_depth_nfns: int = 15, register_depth_sbns: int
= 54):
        self.nfns_depth = register_depth_nfns
        self.sbns_depth = register_depth_sbns
        self.REG_MODE = 0  # Default active mode: NFNS (noise immunity)
        self.phi = (Decimal(1) + Decimal(5).sqrt()) / 2
        self.k = Decimal('0.2')
        # NFNS Basis: F1 = F2 = 1 (article indexing); generalized indices
        # F0 = 0, F(-1) = 1, F(-2) = -1 are needed for splitting twos.
        self.fib = {i: 0 for i in range(-2, self.nfns_depth + 1)}
        self.fib[1] = self.fib[2] = 1
        for i in range(3, self.nfns_depth + 1):
            self.fib[i] = self.fib[i - 1] + self.fib[i - 2]
        self._nfns_suffix = [0] * (self.nfns_depth + 1)  # Suffix sums of
weights
        for p in range(1, self.nfns_depth + 1):
            self._nfns_suffix[p] = self._nfns_suffix[p - 1] + self.fib[p]
        # SBNS Grid: X_n = k * phi^n
        half = self.sbns_depth // 2
        self.sbns_high = self.sbns_depth - 1 - half
        self.sbns_low = -half
        self.LSB = self._weight(self.sbns_low)

    def _weight(self, n: int) -> Decimal:
        return self.k * (self.phi ** n)

    def set_reg_mode(self, mode: int):
        """On-the-fly computational basis switching (in a single clock
cycle)."""
        if mode not in (0, 1):
            raise ValueError("REG_MODE only accepts 0 (NFNS) or 1 (SBNS)")
        self.REG_MODE = mode
        print(f"[TACT]: REG_MODE = {mode} "
              f"({'SBNS (AI Accelerator)' if mode else 'NFNS (Noise
Immunity)'})")

    # ------------------ Voron Filter (Stage 3) ------------------
    def validate_voron_rule(self, code: dict) -> bool:
        """Unified syntactic filter: even zero intervals."""
        active = sorted((n for n, b in code.items() if b), reverse=True)
        for i in range(len(active) - 1):
            if (active[i] - active[i + 1] - 1) % 2 != 0:
                return False
        return True
```

```
    # ------------------ NFNS (REG_MODE = 0) ------------------
    def nfns_encode(self, number: int) -> dict:
        """Finite state machine with pruning of forbidden branches (Stage 3).
        Returns the working form (with the most significant 1)."""
        if self.REG_MODE != 0:
            raise RuntimeError("NFNS encoding called while SBNS is active
(REG_MODE=1)")
        if number < 0:
            raise ValueError("NFNS encodes non-negative integers")
        maxval = self._nfns_suffix[self.nfns_depth]
        if number > maxval:
            raise OverflowError(f"{number} is out of NFNS register grid
(>{maxval})")
        sols = []
        def dfs(pos, remaining, prev, chosen):
            if len(sols) >= 2:
                return
            if remaining == 0:
                sols.append(tuple(chosen))
                return
            if pos < 1 or remaining > self._nfns_suffix[pos]:
                return
            w = self.fib[pos]
            if w <= remaining and (prev is None or (prev - pos - 1) % 2 == 0):
                chosen.append(pos)
                dfs(pos - 1, remaining - w, pos, chosen)
                chosen.pop()
                if len(sols) >= 2:
                    return
            dfs(pos - 1, remaining, prev, chosen)
        dfs(self.nfns_depth, number, None, [])
        if not sols:
            raise ArithmeticError(
                f"Value {number} cannot be represented in a {self.nfns_depth}-
bit NFNS register")
        return {i: (1 if i in sols[0] else 0) for i in range(1, self.nfns_depth
+ 1)}

    def nfns_dual_forms(self, number: int):
        """Dual pair (working, control). Control = None out of duality range (<
F_(depth+1))."""
        if number < 0:
            raise ValueError
        maxval = self._nfns_suffix[self.nfns_depth]
        if number > maxval:
            raise OverflowError
        if number == 0:
            return {i: 0 for i in range(1, self.nfns_depth + 1)}, None
        sols = []
        def dfs(pos, remaining, prev, chosen):
            if len(sols) >= 2:
                return
            if remaining == 0:
                sols.append(tuple(chosen))
                return
```

```
            if pos < 1 or remaining > self._nfns_suffix[pos]:
                return
            w = self.fib[pos]
            if w <= remaining and (prev is None or (prev - pos - 1) % 2 == 0):
                chosen.append(pos)
                dfs(pos - 1, remaining - w, pos, chosen)
                chosen.pop()
                if len(sols) >= 2:
                    return
            dfs(pos - 1, remaining, prev, chosen)
        dfs(self.nfns_depth, number, None, [])
        work = {i: (1 if i in sols[0] else 0) for i in range(1, self.nfns_depth
+ 1)}
        ctrl = ({i: (1 if i in sols[1] else 0) for i in range(1,
self.nfns_depth + 1)}
                if len(sols) > 1 else None)
        return work, ctrl

    def nfns_decode(self, code: dict) -> int:
        return sum(self.fib[i] for i, b in code.items() if b)

    def nfns_add(self, a: dict, b: dict) -> dict:
        """Four-stage algorithm from the article: bitwise addition ->
        splitting twos -> FSM filtering -> working form fixation."""
        if self.REG_MODE != 0:
            raise RuntimeError("NFNS addition called while SBNS is active
(REG_MODE=1)")
        cnt = {i: 0 for i in range(0, self.nfns_depth + 1)}
        for src in (a, b):
            for i, bit in src.items():
                if bit:
                    cnt[i] = cnt.get(i, 0) + 1
        # Stage 2: Splitting twos. General identity: 2F_i = F_(i+1) + F_(i-2)
        # for i >= 3; corner cases: 2F_1 = F_3, 2F_2 = F_3 (2 = F_3).
        for _ in range(1000):
            doubles = [i for i, c in cnt.items() if c >= 2]
            if not doubles:
                break
            for i in doubles:
                while cnt[i] >= 2:
                    cnt[i] -= 2
                    if i <= 2:
                        if 3 > self.nfns_depth:
                            raise OverflowError("Carry overflow past MSB of
NFNS register")
                        cnt[3] += 1
                    else:
                        if i + 1 > self.nfns_depth:
                            raise OverflowError("Carry overflow past MSB of
NFNS register")
                        cnt[i + 1] += 1
                        cnt[i - 2] += 1
        else:
            raise ArithmeticError("Twos splitting limit exceeded")
        value = sum(c * self.fib[i] for i, c in cnt.items() if c)
        return self.nfns_encode(value)  # Stage 3: FSM; Stage 4: Fixation
```

```
    # ------------------ SBNS (REG_MODE = 1) ------------------
    def _check_sbns_mode(self):
        if self.REG_MODE != 1:
            raise RuntimeError("SBNS operation called while NFNS is active
(REG_MODE=0)")

    def sbns_encode(self, value) -> dict:
        """Alternating greedy encoder: constructively maintains even zero
intervals;
        truncation remainder < ~phi*LSB (Truncation Symmetry Theorem: code pair
offsets error)."""
        self._check_sbns_mode()
        target = value if isinstance(value, Decimal) else Decimal(str(value))
        if target < 0:
            raise ValueError("SBNS encodes non-negative values")
        maxval = sum(self._weight(n) for n in range(self.sbns_low,
self.sbns_high + 1))
        if target > maxval:
            raise OverflowError("Value is out of SBNS register grid")
        code = {}
        zeros = 0
        rem = target
        for n in range(self.sbns_high, self.sbns_low - 1, -1):
            w = self._weight(n)
            if rem >= w and zeros % 2 == 0:
                code[n] = 1
                rem -= w
                zeros = 0
            else:
                zeros += 1
        return code

    def sbns_decode(self, code: dict) -> Decimal:
        return sum((self._weight(n) for n, b in code.items() if b), Decimal(0))

    def _sbns_finalize(self, cnt: dict) -> dict:
        """Counter normalization: exact value -> alternating greedy code."""
        exact = sum((c * self._weight(n) for n, c in cnt.items() if c),
Decimal(0))
        if exact < 0:
            raise ArithmeticError("Negative result in SBNS")
        return self.sbns_encode(exact)

    def sbns_add(self, a: dict, b: dict) -> dict:
        self._check_sbns_mode()
        cnt = {}
        for src in (a, b):
            for n, bit in src.items():
                if bit:
                    cnt[n] = cnt.get(n, 0) + 1
        # Upward splitting of twos: 2X_n = X_(n+1) + X_(n-2)
        for _ in range(1000):
            doubles = [n for n, c in cnt.items() if c >= 2]
            if not doubles:
                break
```

```
            for n in doubles:
                while cnt[n] >= 2:
                    cnt[n] -= 2
                    if n + 1 > self.sbns_high:
                        raise OverflowError("Carry overflow past MSB of SBNS
grid")
                    cnt[n + 1] = cnt.get(n + 1, 0) + 1
                    cnt[n - 2] = cnt.get(n - 2, 0) + 1
        else:
            raise ArithmeticError("Splitting limit exceeded")
        res = self._sbns_finalize(cnt)
        exact = self.sbns_decode(a) + self.sbns_decode(b)
        if abs(self.sbns_decode(res) - exact) > VERIFY_TOL_LSB * self.LSB:
            raise ArithmeticError("SBNS addition verification failed")
        return res

    def sbns_sub(self, a: dict, b: dict) -> dict:
        self._check_sbns_mode()
        cnt = {}
        for n, bit in a.items():
            if bit:
                cnt[n] = cnt.get(n, 0) + 1
        for n, bit in b.items():
            if bit:
                cnt[n] = cnt.get(n, 0) - 1
        # Upward borrowing: X_(n-1) = X_(n+1) - X_n  =>  -X_n = X_(n-1) -
X_(n+1)
        for _ in range(1000):
            negs = [n for n, c in cnt.items() if c < 0]
            if not negs:
                break
            for n in negs:
                while cnt[n] < 0:
                    cnt[n] += 1
                    if n + 1 > self.sbns_high:
                        raise OverflowError("Borrow overflow past MSB of SBNS
grid")
                    cnt[n + 1] = cnt.get(n + 1, 0) - 1
                    cnt[n - 1] = cnt.get(n - 1, 0) + 1
        else:
            raise ArithmeticError("Borrowing limit exceeded")
        for _ in range(1000):  # Clean up remaining doubles after borrows
            doubles = [n for n, c in cnt.items() if c >= 2]
            if not doubles:
                break
            for n in doubles:
                while cnt[n] >= 2:
                    cnt[n] -= 2
                    if n + 1 > self.sbns_high:
                        raise OverflowError("Carry overflow past MSB of SBNS
grid")
                    cnt[n + 1] = cnt.get(n + 1, 0) + 1
                    cnt[n - 2] = cnt.get(n - 2, 0) + 1
        res = self._sbns_finalize(cnt)
        exact = self.sbns_decode(a) - self.sbns_decode(b)
        if abs(self.sbns_decode(res) - exact) > VERIFY_TOL_LSB * self.LSB:
```

```
            raise ArithmeticError("SBNS subtraction verification failed")
        return res

    def sbns_mul(self, a: dict, b: dict) -> dict:
        """Hardware multiplier: Weight pair matrix -> Pair adder ->
        V_SCALE scaling gate (multiplication by k = 0.2)."""
        self._check_sbns_mode()
        cnt = {}
        for i, bi in a.items():
            if not bi:
                continue
            for j, bj in b.items():
                if bj:
                    p = i + j
                    if p > self.sbns_high:
                        raise OverflowError("Product is out of SBNS register
grid")
                    cnt[p] = cnt.get(p, 0) + 1
        exact = sum((c * self._weight(n) for n, c in cnt.items() if c),
Decimal(0))
        exact *= self.k  # V_SCALE: multiplier by 1/5
        res = self.sbns_encode(exact)
        if abs(self.sbns_decode(res) - exact) > VERIFY_TOL_LSB * self.LSB:
            raise ArithmeticError("SBNS multiplication verification failed")
        return res

    def sbns_div(self, a: dict, b: dict) -> dict:
        """Bitwise division with auto-stop upon zero remainder; dropped
remainder < LSB."""
        self._check_sbns_mode()
        A, B = self.sbns_decode(a), self.sbns_decode(b)
        if B == 0:
            raise ZeroDivisionError("Division by zero in SBNS")
        q = {}
        rem = A
        for p in range(self.sbns_high, self.sbns_low - 1, -1):
            w = self._weight(p)
            if rem >= w:
                q[p] = 1
                rem -= w
        if rem >= self.LSB:
            raise ArithmeticError("Division remainder exceeds the grid LSB")
        exact = A / B
        maxval = sum(self._weight(n) for n in range(self.sbns_low,
self.sbns_high + 1))
        if exact > maxval:
            raise OverflowError("Quotient out of SBNS grid bounds")
        res = self.sbns_encode(exact)
        if abs(self.sbns_decode(res) - exact) > VERIFY_TOL_LSB * self.LSB:
            raise ArithmeticError("SBNS division verification failed")
        return res

# ------------------ Matrix Multiplier ------------------
    def matrix_multiply(self, A, B):
        """Matrix multiplication of floats in SBNS mode; returns a float matrix
        and Decimal baseline for verification."""
```

```
        self._check_sbns_mode()
        n = len(A)
        m = len(B[0])
        K = len(B)
        R = []
        for i in range(n):
            row = []
            for j in range(m):
                acc = Decimal(0)
                for t in range(K):
                    ca = self.sbns_encode(Decimal(str(A[i][t])))
                    cb = self.sbns_encode(Decimal(str(B[t][j])))
                    pr = self.sbns_mul(ca, cb)
                    exact = self.sbns_decode(pr)
                    if abs(exact - Decimal(str(A[i][t])) *
Decimal(str(B[t][j]))) > VERIFY_TOL_LSB * self.LSB:
                        raise ArithmeticError("Product verification failed")
                    acc += exact
                row.append(acc)
            R.append(row)
        return R

# ---------------- DEMO SCENARIO ----------------
if __name__ == "__main__":
    alu = VoronHybridProcessorALU()
    print("=== Scenario 2: NFNS (REG_MODE=0) ===")
    alu.set_reg_mode(0)
    a, b = alu.nfns_encode(7), alu.nfns_encode(8)
    print("7  ->", a, "=", alu.nfns_decode(a))
    print("8  ->", b, "=", alu.nfns_decode(b))
    s = alu.nfns_add(a, b)
    print("7+8 ->", s, "=", alu.nfns_decode(s))
    print("=== Scenario 1: SBNS (REG_MODE=1) ===")
    alu.set_reg_mode(1)
    x, y = alu.sbns_encode(Decimal('0.755')),
alu.sbns_encode(Decimal('0.235123'))
    print("0.755    ->", alu.sbns_decode(x))
    print("0.235123 ->", alu.sbns_decode(y))
    print("add  ->", alu.sbns_decode(alu.sbns_add(x, y)))
    print("sub  ->", alu.sbns_decode(alu.sbns_sub(x, y)))
    print("mul  ->", alu.sbns_decode(alu.sbns_mul(x, y)))
    print("div  ->", alu.sbns_decode(alu.sbns_div(x, y)))
    R = alu.matrix_multiply([[0.2, 0.4], [0.6, 0.8]], [[0.25, 0.5], [0.75,
0.3]])
    print("matrix ->", R)
```

STRESS TEST LOG — Verification simulator of the hybrid ALU "VORON-NFNS/SBNS", Revision 2.2

Register Parameters: NFNS depth=15 (network 1..1596, duality up to 986 = F_{16}-1); SBNS depth=54 (indices -27..+26, X_n = k * phi^n, k=0.2, CSE X_0=0.2).
LSB (least significant weight of the SBNS grid) = 4.554e-07.

=== NFNS MODE (REG_MODE=0) ===
1. Duality on the grid 1..986: decoding errors: 0; values missing a control form: 0 (out of 986).
2. Range 987..1596: unrepresentable values in the register: 0.
Examples of dual pairs (matching the tables in the paper):
2: working `100` | control `11`
* 4: working `1001` | control `111`
* 7: working `10011` | control `1111`
* 8: working `100000` | control `11000`
* 9: working `100001` | control `11001`
* 25: working `10000111` | control `1100111`
3. NFNS addition, 300 random pairs: arithmetic errors: 0; true overflows: 0.
4. SEU-intercept (1500 single bit-flips in a register pair): syntax filter caught 1004/1500 (67%); filter + value verification across working/control forms caught 1500/1500 (100%).

=== SBNS MODE (REG_MODE=1) ===
5. SBNS encoding, 200 random values within [0..100]: invalid codes: 0; error: 1.61 LSB (theoretical upper bound $\phi \cdot$ LSB ≈ 1.62).
6. Addition/subtraction, 200 pairs: verification errors: 0; overflows (incl. negative subtraction result): 0; error for add: 4.05 LSB, sub: 2.71 LSB.
7. Multiplication (V_SCALE, k=0.2), 100 pairs: exceptions: 0; RELATIVE error: 4.99e-07. 100 pairs: exceptions: 0; RELATIVE error: 4.99e-07.
8. Division (bitwise, auto-stop upon zero remainder), 100 pairs: exceptions: 0; relative error: 1.04e-04.
9. Matrix Multiplier 2x2: result `[[0.3499985532686482, 0.2199992739468716], [0.7499980978993893, 0.5399983202091138]]`; relative error of elements < 1e-6.
10. SEU-intercept SBNS (300 bit-flips): syntax filter caught 261/300 (87%); filter + value verification caught 300/300 (100%).

Paper demo scenario summary: `REG_MODE 0`: 7+8 = 15 (1000011); `REG_MODE 1`: 0.755 and 0.235123, add: 0.990123 / sub: 0.519877 / mul: 0.177518 / div: 3.211085 — all metrics remain strictly within the verification tolerance bounds.

All pipelines are deterministic (iteration limits applied); register boundaries throw explicit exceptions.

Notes: In step 6, the error accumulation up to ~ 4 LSB is compounded by the encoding errors of the operands (≤ 1.6 LSB each) and normalization—an inherent property of the grid; internal verification matches the result against the decoded operands. In step 8, the maximum relative error of 1.04e-04 occurs on quotients smaller than the least significant grid weight (truncated to 0); the absolute error across all 100 pairs stays strictly within tolerance limits.