IMPLEMENTATION OF BASIC ARITHMETIC OPERATIONS IN THE SCALED BERGMAN NUMBER SYSTEM (SBNS)

Andrey V. Voron (anvoron1@yandex.ru)

November 5, 2026

Abstract

The paper examines the algorithmic specifics of performing basic arithmetic operations in the Scaled Bergman Number System (SBNS) with a fixed weight scale factor of $k = 0.2$. The mechanisms for implementing linear operations (addition and subtraction) are analyzed, and the specific hardware phases for executing non-linear operations – multiplication and division under conditions of a scaled basis shift – are decomposed and described in detail. Using a concrete example of bitwise numerical multiplication, the operation of a two-phase arithmetic logic unit (ALU) utilizing a cyclic multiplier by a constant of 5 for scale code correction is demonstrated. A three-stage division cycle is described that employs a hardware parity filter to eliminate redundancy and ensure a fast, unambiguous bitwise comparison operation.

1 Introduction

Modern semiconductor microelectronics and computer engineering, based on the classical von Neumann architecture and binary coding, have closely approached a technological and physical dead end. As an alternative, redundant positional number systems have been developed at various times [1, 2, 3, 4, 5, 6]. To overcome the existing limitations of the binary number system, the author developed and formalized the Scaled Bergman Number System (SBNS [7]), which departs from being anchored to an abstract unit ($\varphi^0 = 1$) in favor of a central scale element – a fixed weight scale factor of $k = 0.2$. Any n-th member of the SBNS weight series is defined by the analytical formula $X_n = 0.2 \times \varphi^n$. The principles of classical Fibonacci computer arithmetic [2, 3], as well as design experience with the New Fibonacci Number System (NFNS) featuring a dual basis for fault-tolerant environments, were utilized as the theoretical foundation for overcoming redundancy [4, 5].

The present work is dedicated to investigating the basic mathematical logic of bit expansion and contraction at the level of the arithmetic logic unit (ALU) of an SBNS processor. The objective of this paper is to decompose the algorithms for addition, subtraction, bitwise multiplication with intermediate scale correction, and three-stage division, as well as to provide a mathematical justification for built-in trigger-level data integrity control based on the "parity rule".

2 The main part

From a mathematical standpoint, the basic logic of bit expansion and contraction in SBNS is inherited from George Bergman's classical number system [1]. Since both systems are based on the irrational golden ratio φ, the fundamental identities for carries (such as $\varphi^n = \varphi^{n-1} + \varphi^{n-2}$) remain unchanged.

Nevertheless, simply "rewriting" the algorithms of George Bergman's number system for SBNS is impossible. The introduction of a fixed scale factor ($k = 0.2$) creates a mathematical shift that requires a correction of the rules.

1. Linear Operations (Addition and Subtraction). Here, the transformation is trivial. Since the coefficient 0.2 can simply be factored out, the rules of addition and subtraction in George Bergman's number system are mapped onto SBNS one-to-one.

In George Bergman's number system, "contraction" takes the form of $1\varphi^n + 1\varphi^{n-1} \rightarrow 1\varphi^{n+1}$, whereas in SBNS it is $1 \times (0.2\varphi^n) + 1 \times (0.2\varphi^{n-1}) \rightarrow 1 \times (0.2\varphi^{n+1})$. The logic of carries and borrows in the ALU remains unchanged; only the physical "weight" assigned to each register trigger changes.

2. Non-linear Operations (Multiplication). In the classical George Bergman system, the multiplication of two base digits takes the form of $\varphi^a \times \varphi^b = \varphi^{a+b}$. In SBNS, however, multiplying the weights produces the square of the scale factor: $X_a \times X_b = (0.2 \times \varphi^a) \times (0.2 \times \varphi^b) = 0.04 \times \varphi^{a+b}$. The result effectively "drops out" of the system's base scale, since we need to obtain a value in the form of $0.2 \times \varphi^n$. Therefore, when representing multiplication, it is necessary to describe a scale correction algorithm. To achieve this, the processor must perform a shift of the resulting partial product code. A supplementary operation of dividing the intermediate result by 0.2 (which is equivalent to multiplying by the integer 5) is required to restore the original SBNS basis.

3. Division. Classical division in George Bergman's number system relies on the step-by-step subtraction of the divisor, shifted by a certain number of positions φ^{-i}. In SBNS, this shift is also displaced by the scale value. Each step of division or trigonometric calculation (via CORDIC) must now operate with constants scaled to a step of $0.2 \times \varphi^{-i}$. We supplement the arithmetic of George Bergman's number system with a scaled ALU controller that compensates for the shift ($k = 0.2$) during multiplication, yielding hardware-driven self-attenuation of fractional noise in return.

Methodology of Bitwise Multiplication in SBNS (Using the Example of 2 × 2 = 4).

Let us consider bitwise multiplication in the SBNS system through a concrete example, demonstrating how the classical George Bergman algorithm is modified due to the scale factor ($k = 0.2$). For simplicity and clarity, we will multiply two integers: $2 \times 2 = 4$.

Step 1. Conversion of numbers into SBNS codes. The number 2 in the SBNS system is represented by three active digits (for Filtering Working Code "A") (2 = 1.37082039325 + 0.52360679775 + 0.12360679775). In terms of weights, this takes the form $2 = X_4 + X_2 + X_{-1}$. Let us write our expression in terms of SBNS weights: $(X_4 + X_2 + X_{-1}) \times (X_4 + X_2 + X_{-1})$.

Step 2. Bitwise multiplication (matrix formation). We expand the brackets, multiplying every digit by every other digit, just like in a regular polynomial. In the classical George Bergman system, we would simply add the power indices ($\varphi^a \times \varphi^b = \varphi^{a+b}$). In SBNS, however, each weight includes 0.2. $X_a \times X_b = (0.2\varphi^a) \times (0.2\varphi^b) = 0.04\varphi^{a+b}$. Let us compose the matrix of partial products (the indices are simply added, but we keep the coefficient 0.2 in mind before each result).

$$X_4 \times X_4 = 0.2 \times X_8$$
$$X_4 \times X_2 = 0.2 \times X_6$$
$$X_4 \times X_{-1} = 0.2 \times X_3$$
$$X_2 \times X_4 = 0.2 \times X_6$$
$$X_2 \times X_2 = 0.2 \times X_4$$
$$X_2 \times X_{-1} = 0.2 \times X_1$$
$$X_{-1} \times X_4 = 0.2 \times X_3$$
$$X_{-1} \times X_2 = 0.2 \times X_1$$
$$X_{-1} \times X_{-1} = 0.2 \times X_{-2}$$

Step 3. Preliminary summation. We add identical digits (for instance, two instances of $0.2 \times X_6$ yield $0.4 \times X_6$). The intermediate result = $0.2 \times (X_8 + 2X_6 + 2X_3 + X_4 + 2X_1 + X_{-2})$. Now, using the baseline Fibonacci contraction rule of George Bergman's number system ($X_n = X_{n-1} + X_{n-2}$), the processor's ALU simplifies this combination of weights. For example, two X_6 terms can be expanded and then contracted with other digits. Following the execution of standard Bergman contractions/expansions, the combination in brackets reduces to a clean form: $(X_8 + 2X_6 + 2X_3 + X_4 + 2X_1 + X_{-2}) \rightarrow X_9 + X_7 + X_4 + X_2 + X_{-1} + X_{-3}$.

Let us plug in the actual numerical values of the weights from the preprint to verify accuracy: 15.2026311235 (X_9) + 5.80688837075 (X_7) + 1.37082039325 (X_4) + 0.52360679775 (X_2) + 0.12360679775 (X_{-1}) + 0.0472135955 (X_{-3}) = 23.0747670785.

Multiplying by the factored-out coefficient of 0.2: 23.0747670785 × 0.2 = 4.6149534157. This represents an intermediate value in the intermediate scale of $k = 0.04$ (when converted back to the base basis). To restore the true code for the number 4, a scale correction phase (Step 4) is required.

Step 4. Scale correction (multiplication by 5). At this point, the classical George Bergman algorithm would have concluded, but in SBNS, the ALU is obligated to return the number to the original scale. To do this, the intermediate result must be divided by 0.2, which physically means multiplying by the integer 5. How can a code be multiplied by 5 at the hardware level using Fibonacci logic? The number 5 is the weight of the digit X_5 / 0.2 or the sum $X_4 + X_2 + X_{-1} + 1$. The processor executes a fast hardware multiplication of the intermediate code by the constant 5: 4.6149534157 × 5 = 23.0747670785.

Step 5. Final filtering and "Noise Attenuation". The obtained value of 23.0747670785 is mapped back into the SBNS weight grid by the hardware filter. 23.0747670785 precisely corresponds to the sum of the weights $X_9 + X_7 + X_4 + X_2 + X_{-1} + X_{-3}$. Looking at the author's code table [7] (Table A3) for the number 4, the Filtering Working Code "A" (FWC [7]) contains ones in the digits X_7, X_1, X_{-2}, X_{-4}, X_{-6}, X_{-8}. Upon transitioning to the Checking Code "B" (FCC [7]) or performing the final contraction, the fractional "tails" of the numbers (all those trailing fives and register truncation noises) mutually cancel each other out in accordance with the Symmetry of Truncation Theorem.

Multiplication in SBNS "works" according to the laws of George Bergman's number system but requires a two-phase ALU:

1. Phase No. 1: Addition of digit indices (George Bergman's number system) with an automatic shift of the result into the "intermediate" scale of $k = 0.04$.

2. Phase No. 2: A hardware cyclic multiplier by a fixed constant of 5 (scale corrector) before sending the code to the hardware parity filter.

While this complicates the ALU by a few logic gates compared to the classical George Bergman number system implementation, it completely eliminates the problem of infinite fractions during rounding.

The Division Operation in SBNS.

The division operation in the SBNS system is the most complex yet mathematically predictable process. Because the base digit weights are rigidly tied to the geometry of the golden ratio φ, division in SBNS inherits the classical division principles of George Bergman but is modified due to the scale factor $k = 0.2$.

In traditional processors, division is performed either as a sequential subtraction of the divisor from the dividend with a register shift (algorithms of the Restoring/Non-Restoring Division family) or via multiplication by the reciprocal (the Newton-Raphson method) [8]. In SBNS, the process is divided into three hardware stages.

Stage 1. Scale Alignment (Preprocessing). Since every base digit in SBNS has an embedded scale $X_n = 0.2 \times \varphi^n$, dividing one number by another in analytical form introduces a mathematical shift: $X_a / X_b = 0.2 \times \varphi^a / 0.2 \times \varphi^b = \varphi^{a-b}$. In this process, the scale factor of 0.2 in the numerator and denominator cancels out. On output, we obtain a "pure" power of the golden ratio φ^{a-b}, which drops out of the standard SBNS weight grid (where every digit is required to contain a factor of 0.2).

Before launching the division cycle, the processor's ALU must correct the dividend. It performs a hardware multiplication of the dividend by 0.2 (or divides by 5) so that the intermediate division result is immediately formed in the correct SBNS basis: $(0.2 \times X_a) / X_b = 0.2 \times (0.2 \times \varphi^a) / 0.2 \times \varphi^b = 0.2 \times \varphi^{a-b} = X_{a-b}$.

Stage 2. Subtraction and Shift Cycle (Analogue of George Bergman's Algorithm). After scale alignment, a classical "long division"-like procedure is launched, moving from the most significant bits to the least significant bits:

1. Sign determination and comparison: The processor compares the most significant bits of the dividend (remainder) and the divisor.

2. Subtraction: If the remainder is greater than or equal to the divisor, a one (1) is written into the corresponding bit of the quotient, and the divisor is subtracted from the remainder.

3. Shift: If the remainder is smaller, a zero (0) is written into the quotient bit, and the divisor is shifted one bit to the right (which is equivalent to dividing it by φ). In SBNS, shifting a code left or right by one trigger means an automatic multiplication or division of the weight by φ, since the link between adjacent cells is strictly geometric: $X_n \times \varphi = X_{n+1}$.

The main hardware challenge of this stage lies in the fact that while determining which number is larger is trivial in a binary system – simply comparing the first non-matching bits from left to right – in redundant systems (George Bergman [1], Fibonacci [2, 3], NFNS [4, 5], SBNS [7]), a single number can be written in a multitude of ways [6]. For example, a number can be represented as a leading digit 100... or as a sum of lower digits 011.... In this regard, SBNS (as well as NFNS) possesses a significant advantage: the hardware parity filter [7]. Because the filter strictly constrains redundancy and confines the code to only two deterministic states (Working Code "A" or Checking Code "B"), the processor does not need to expend clock cycles on intermediate recoding. The filter makes the code comparison operation fast and unambiguous.

Stage 3. Automatic Cycle Termination and Remainder Attenuation. In binary processors, the division of fractional numbers often results in an infinite repeating fraction (e.g., 1 / 3 = 0.3333...), and the cycle is forcibly terminated when the register capacity is exhausted, generating a truncation error.

In SBNS, when dividing integers or commensurable fractions within a given range, the property of the "mutual destruction of trailing fives" causes the division remainder at a certain step (typically at the 10th – 12th decimal place) to generate a microstructure that evaluates to an exact zero during a subsequent Fibonacci contraction. The processor physically "sees" that there is nothing further to compute (the remainder equals 0) and instantly terminates the division cycle. At that exact moment, a dynamic bit-width algorithm is triggered, cutting off empty register cells to conserve energy.

Hardware Data Integrity Control Operation.

The hardware integrity control operation in the SBNS system is a specialized logical procedure executed directly at the flip-flop level of the arithmetic logic unit (ALU). It serves to instantly detect hardware faults (such as those induced by cosmic radiation in aerospace systems) without utilizing software-level redundant codes (like CRC or checksums) [9]. This operation is based on the theorem formulated by the author regarding parity invariance under the operation of a hardware filter [7]. The control logic relies on a unique property of SBNS (and NFNS [4, 5]): any number inside the processor is always "clamped" by the hardware filter between two valid code states:

1. Working Code "A" (The expanded form of the number representation).

2. Checking Code "B" (The contracted form of the number representation).

The positions of any two adjacent active ones in a "correct" code must have opposing parity if they stand next to each other or are separated exclusively by an even number of zeros. The sum of the truncated weights of active digits for codes "A" and "B" is strictly invariant (equal to a single whole value). Upon transitioning from form "A" to form "B" and vice versa, the transit error is absolutely zero (EpsilonTransition≡0)

3 Conclusion

1. While linear operations (addition, subtraction) map from the classical George Bergman number system algorithm to SBNS one-to-one due to factoring out the 0.2 coefficient, non-linear operations require the introduction of a scaled ALU controller. Multiplication is accompanied by the emergence of a scale square $0.04 \times \varphi^{a+b}$, which is resolved by introducing a second phase of computation – a fast hardware-level multiplication of the partial product code by a fixed constant of 5. This is performed to restore the system's original basis.

2. The traditional problem of slow bitwise comparison of the remainder and the divisor, inherent to redundant irrational systems, is fully resolved in SBNS through the application of a hardware parity filter (the parity rule). The filter rigidly limits coding redundancy, maintaining the number strictly within the bounds of two deterministic code states – Working Code "A" and Checking Code "B" – which renders bit comparison during step-by-step subtraction unambiguous and fast.

3. Drawing upon the parity invariance theorem, the ALU is capable of performing instant, end-to-end data integrity verification directly at the register flip-flop level without relying on software redundancy (CRC). The transit error during the transition from form "A" to form "B" is absolutely zero, ensuring high fault tolerance of the computing environment under the influence of external negative factors.

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] Zeckendorf E. Representation of natural numbers as a sum of Fibonacci numbers or Lucas numbers (Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas). Bulletin de la Société Royale des Sciences de Liège, 1972, vol. 41, pp. 179–182.

[7] 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.

[8] Koren I. Computer Arithmetic Algorithms. 3rd ed. Boca Raton, AK Peters/CRC Press, 2018. 312 p.

[9] Parhami B. Computer Arithmetic: Complexity and Design of Fault-Tolerant Systems. New York, Oxford University Press, 2010. 496 p.