Technical Report: Bit-Level "Grade-School" Arithmetic in Modern C++ by James Pate Williams, Jr. and Microsoft's Copilot

Restoring Unsigned Division + Shift-and-Add Multiplication with Statistical Test Harness

---

Publication Note

"Developed collaboratively with Microsoft Copilot; implementation, testing, and benchmarking performed by the author."

Abstract

This report documents the design and validation of two classic binary arithmetic algorithms implemented in C++: (1) **restoring division** for unsigned integers and (2) **shift-and-add multiplication** (the binary analog of grade-school long multiplication). Restoring division iteratively shifts in dividend bits, performs a trial subtraction of the divisor, and "restores" the remainder when the subtraction would go negative—an approach widely described in computer-architecture references. Shift-and-add multiplication decomposes the multiplier into bits and conditionally adds shifted copies of the multiplicand, mirroring long multiplication in base 2. [geeksforgeeks.org], [users.utcluj.ro] [users.utcluj.ro], [stackoverflow.com]

An initial exhaustive test strategy over all numerators and denominators for each bit-width *n* produced a combinatorial blow-up (approximately 2^{2n}operand pairs), making performance results meaningless for algorithm analysis. The harness was redesigned to run a **fixed number of randomized tests per n** (200,000), timed with **std::chrono**, yielding scalable verification through $n = 32$. The user-reported benchmark outcomes were: **division test runtime ≈ 1.044 s** and **multiplication test runtime ≈ 0.739 s** (both across n=1..32 with 200,000 trials per n).

1. Background and Motivation

Elementary "grade-school" arithmetic generalizes cleanly to binary. In the multi-precision setting, Donald Knuth describes classical arithmetic procedures (including "Algorithm M" for multiplication) in *The Art of Computer Programming, Volume 2*, section 4.3.1 ("The Classical Algorithms"). While TAOCP presents these procedures in a base-b digit framework, the same concepts map naturally to fixed-width binary words and to hardware-flavored bit-serial routines. [othedev.github.io], [pearson.de]

Two algorithms were selected because they are foundational, pedagogically clear, and implementable using simple operations:

1. **Restoring division (unsigned)**: repeated shift + subtract + conditional restore. [geeksforgeeks.org], [users.utcluj.ro]
2. **Shift-and-add multiplication (unsigned)**: repeated conditional add + shifts based on multiplier bits. [users.utcluj.ro], [stackoverflow.com]

2. Algorithms

2.1 Restoring Division (Unsigned)

Concept. Restoring division maintains a running remainder register. For each bit position (most significant to least), it shifts the remainder left, "brings down" the next dividend bit, then tries subtracting the divisor. If the result is negative, it restores the previous remainder (by undoing the subtraction) and writes a 0 quotient bit; otherwise it keeps the subtraction result and writes a 1 quotient bit. This "trial subtraction + restore on negative" behavior is the defining feature of restoring division. [geeksforgeeks.org], [users.utcluj.ro]

Implementation approach used here.

- Work in a conceptual width of **(n+1)** bits for the remainder so that negative detection can be represented via a sign bit (or borrow).
- Use two's complement addition to compute "trial subtraction" as $T = R - D$by adding $R + (-D)$, then test the top bit of the (n+1)-bit trial remainder to decide whether subtraction underflowed.
- For the $n = 32$edge case, promote to uint64_t so a 33-bit remainder mask is representable (avoiding undefined behavior in shifts).

This matches the standard restoring pattern (shift, subtract, decide, restore if needed), while expressing it with efficient bitwise operations. [geeksforgeeks.org], [users.utcluj.ro]

2.2 Shift-and-Add Multiplication (Unsigned)

Concept. Shift-and-add multiplication is literally grade-school multiplication in base 2: scan bits of the multiplier; when a bit is 1, add the appropriately shifted multiplicand to an accumulator. Architecture treatments explain it as repeated conditional addition, coupled with left shifting the multiplicand (or right shifting the multiplier) each iteration. [users.utcluj.ro], [userpages....s.umbc.edu]

A compact statement of the idea is: decompose the multiplier into powers of two and sum shifted copies of the multiplicand. [stackoverflow.com], [users.utcluj.ro]

Knuth connection. In TAOCP's classical multiplication (often referred to as "Algorithm M"), the same structure appears in radix-b form: multiply digit pairs and accumulate into an output array with carry propagation. In binary with single-bit "digits," this collapses to conditional adds and shifts. [othedev.github.io], [pearson.de]

3. Test Methodology and Harness Design

3.1 Why Exhaustive Testing Broke Down

The initial driver enumerated all inputs for each n-bit width:

- numerators: $0 \ldots 2^n - 1$
- denominators: $1 \ldots 2^n - 1$

That is $(2^n) \cdot (2^n - 1) \approx 2^{2n}$ divisions per n. Increasing n by 2 multiplies work by roughly $2^4 = 16$, which aligns with the observed jump from "seconds" to "hundreds of seconds." This is not an algorithm problem; it is a test-space explosion.

3.2 Fixed-Work Statistical Testing (Adopted)

To measure algorithm behavior (not test enumeration growth), the harness was redesigned to run a **constant number of randomized trials per n**:

- For each $n \in [1,32]$: run **200,000** trials
- Generate operands masked to n bits: mask = (n==32) ? 0xFFFFFFFF : ((1u<<n)-1)
- Ensure denominator non-zero for division: denom |= 1 (or otherwise enforce denom != 0)
- Validate against hardware operators:
    - division: numer / denom, numer % denom
    - multiplication: (uint64_t)a * (uint64_t)b

This design gives broad coverage and predictable runtime while still detecting regressions with high probability.

3.3 Modern Timing

Timing was implemented using **std::chrono** (high-resolution clock), replacing legacy clock()/time.h. This avoids platform-dependent tick rates and provides straightforward wall-clock duration measurement.

4. Results (User-Reported)

With the redesigned harness (n = 1..32, 200,000 trials per n, Release build), the reported runtimes were:

- **Division test runtime**: **1.04426 seconds**
- **Multiplication test runtime**: **0.738832 seconds**

Both suites completed with **n-maximum = 32** and "All tests passed."

4.1 Interpretation

Given identical trial counts across n and consistent operand masking, these times reflect relative per-trial cost:

- Restoring division does per-bit work that includes trial subtraction and sign/borrow handling. [geeksforgeeks.org], [users.utcluj.ro]
- Shift-and-add multiplication typically performs at most one add per multiplier bit, but many iterations do "shift-only" when the current multiplier bit is 0; this can make it slightly faster in practice for random multipliers. [users.utcluj.ro], [stackoverflow.com]

5. Implementation Notes (Engineering Choices That Matter)

5.1 Use of Fixed-Width Types (`uint32_t`)

- The decision to standardize the driver and arithmetic code on `uint32_t` is correctness-driven: `unsigned int` is not guaranteed to be 32 bits on all platforms, while `uint32_t` is. This is particularly important for bit-accurate algorithms and masks.

5.2 Avoiding Undefined Shifts

- Shifting by the type width (e.g., `1u << 32`) is undefined behavior in C++. The harness uses explicit handling for $n = 32$masks. The division routine similarly promotes to `uint64_t` when modeling 33-bit intermediates (remainder width $n + 1$).

5.3 Clear Separation: Algorithm vs. Harness

A major outcome of this exercise is architectural: algorithms live in Arithmetic.*, while the driver organizes:

- menu selection (planned)
- per-algorithm test functions
- consistent RNG seeding
- reporting and timing

This separation makes it easy to extend into additional algorithms (non-restoring division, Booth multiplication, etc.) without destabilizing the baseline.

6.1 Limitations

- **Random testing is not exhaustive.** It provides high confidence but not formal proof.
- **Uniform RNG masking** is good for bit coverage but does not guarantee targeted stress on pathological cases unless those are added explicitly.

6.2 Recommended Next Steps

1. **Add edge-case test vectors** per n (0, 1, max, powers of two, alternating bit patterns).
2. **Per-n timing breakdown** to observe scaling curves directly.
3. Implement:
    - **Non-restoring division** (avoids explicit restoration each step; described as a common improvement over restoring in architecture texts). [users.utcluj.ro], [people.cs.pitt.edu]
    - **Booth multiplication** (reduces additions for runs of 1s). [userpages….s.umbc.edu]

Appendix A: References (Reading-Friendly Links)

- Knuth's TAOCP home page: The Art of Computer Programming (official page) [www-cs-fac…anford.edu]
- TAOCP Vol. 2 contents showing "Arithmetic / Classical Algorithms": TAOCP Vol. 2 table of contents (PDF) [pearson.de]
- Summary page referencing Knuth's "Algorithm M" and related classical algorithms: Algorithms summary (includes Knuth Algorithm M) [othedev.github.io]
- Shift-and-add multiplication explanation: Shift-and-Add Multiplication (PDF) [users.utcluj.ro]
- Restoring division overview: Restoring Division Algorithm (Unsigned) [geeksforgeeks.org]
