Faster Inversion Modulo $2^{255} - 19$: Constant-time modular inversion on Raspberry Pi 4 and Pi 5

Cesare Huang **Bo-Yin Yang**

Institute of Information Science, Academia Sinica
Taipei, Taiwan

Cesarehuang@icloud.com • byyang@iis.sinica.edu.tw

3,272
cycles / inversion
Raspberry Pi 5 / A76

3,917
cycles / inversion
Pi 400 / A72

01 BACKGROUND: BERNSTEIN–YANG DIVSTEPS

How do divsteps compute an inverse?

An inverse is an x such that $ax \equiv 1 \pmod{p}$. We will find the inverse of 3 modulo 7 using swaps, additions and halvings. In general, assume p is an odd prime and $0 < a < p$.

Start with $(F, G) = (p, a)$ and $\delta = \frac{1}{2}$. δ controls swaps; ϵ is 0 for even G, 1 for odd G. One **half-delta divstep** is

$$(\delta', F', G') = \begin{cases} (1-\delta,\ G,\ (G-F)/2), & \delta > 0,\ \epsilon = 1, \\ (1+\delta,\ F,\ (G+\epsilon F)/2), & \text{otherwise.} \end{cases}$$

These steps preserve the greatest common divisor. Track $(V, S) = (0, 1)$ with the same updates so $F \equiv aV$, $G \equiv aS \pmod{p}$. For V, S, halve modulo p: add p to an odd residue, then divide by two.

The first step, by hand. With $F = 7$, $G = 3$ and $\delta = 1/2$, the swap branch gives $F' = 3$ and $G' = (3-7)/2 = -2$. Likewise $V' = S = 1$, while $S' = (1-0)/2 \pmod 7$ is 4: add 7 to the odd numerator 1, then halve 8. Check: $3 \cdot 4 \equiv -2 \pmod 7$.

Example: $p = 7$, $a = 3$	δ	F	G	V	S
Start	1/2	7	3	0	1
1: swap, subtract, halve	1/2	3	−2	1	4
2: halve even G	3/2	3	−1	1	2
3: swap, subtract, halve	−1/2	−1	−2	2	4
4: halve even G	1/2	−1	−1	2	2
5: swap, subtract, halve	1/2	−1	0	2	0

At $G = 0$, $F = -1 \equiv 3 \cdot 2 \pmod 7$. Negate $V = 2$ to get $3^{-1} \equiv 5 \pmod 7$: indeed, $3 \cdot 5 = 2 \cdot 7 + 1$. Generally, $F = \pm 1$ gives $a^{-1} \equiv FV \pmod p$.

For $2^{255} - 19$, we use 588 fixed steps and branchless selections. The table stops early only to illustrate how the inverse emerges.

02 BACKGROUND: LINEAR TRANSFORMATIONS

Why can we batch divsteps?

1. Write a chosen step as a matrix

The swap branch gives $F' = G$, $G' = (G-F)/2$: both are **linear combinations** of the old F, G. Each chosen step applies $A/2$, where

$$A_{\text{swap}} = \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix}, \qquad A_{\text{other}} = \begin{pmatrix} 2 & 0 \\ \epsilon & 1 \end{pmatrix}.$$

The branch and $\epsilon = G \bmod 2$ still depend on the state; the matrix describes the update *after* that choice.

2. Multiply matrices instead of updating big integers

In the example above, step 1 swaps; step 2 halves an even G. Their combined integer matrix is

$$M_2 = A_2 A_1 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 4 \\ -1 & 1 \end{pmatrix}.$$

Apply it once, dividing by 2^2 for the two halvings:

$$\frac{1}{4} M_2 \begin{pmatrix} 7 \\ 3 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 12 \\ -4 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \end{pmatrix}.$$

This is exactly row 2 of the table. In general,

$$\begin{pmatrix} F_n \\ G_n \end{pmatrix} = 2^{-n} \underbrace{A_n \cdots A_1}_{M} \begin{pmatrix} F_0 \\ G_0 \end{pmatrix}.$$

The first step is the *rightmost* factor. The entries of M summarize all n steps.

3. Low bits choose the steps; NEON applies the batch

Given δ, each choice needs only parity. One halving consumes one bit of precision, so the starting low n bits suffice for n choices. Scalar code follows these small values and accumulates M; it need not update all limbs at every step.

In our two-step example, $(7, 3) \bmod 4 = (3, 3)$. Its first small update is $(3, 0)$: G is even, so it chooses the same A_2.

Apply $2^{-n}M$ once to full-width F, G using exact integer division, and to V, S *modulo* p using the modular inverse of 2^n. For the two-step example, $4^{-1} = 2 \pmod 7$, so $(V_2, S_2)^{\mathsf{T}} \equiv 2M_2(0, 1)^{\mathsf{T}} \equiv (1, 2)^{\mathsf{T}} \pmod 7$. **Same result, fewer full-width updates.**

03 THIS WORK: FROM x20 TO x24

Why pack f, u, v together?

Scalar f, g hold the low bits used to compute $M = \left(\begin{smallmatrix} u & v \\ r & s \end{smallmatrix}\right)$; NEON holds the full F, G, V, S state. Keep each low value together with its scaled matrix row in one 64-bit word:

$$\texttt{fuv} = f_{\text{low}} + 2^b \tilde{u} + 2^h \tilde{v},$$
$$\texttt{grs} = g_{\text{low}} + 2^b \tilde{r} + 2^h \tilde{s}.$$

Selection, addition/subtraction and shifting act on whole rows. Initialize coefficients at $-2^n I$: power-of-two scaling makes shifts exact; the negative sign fits the signed $(n+1)$-bit range $[-2^n, 2^n - 1]$.

After k steps, $C_k = -2^{n-k} T_k$ for integer transition matrix T_k. Signed extraction and negation recover T_n.

Our extension: one cleanup, 24 steps

Two $(n+1)$-bit coefficient spans and one top sign bit leave

$$b = 64 - 1 - 2(n+1), \qquad h = b + n + 1.$$

x20: 21-bit coefficients at bits 21 and 42; no intermediate cleanup.

x24: 25-bit coefficients at bits 13 and 38; one cleanup after step 12. The initial low24 overlaps the lower coefficient's nominal span.

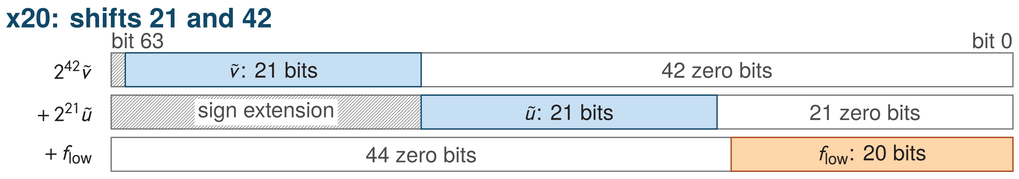
Continued at the top of the right column ⟶

03 THIS WORK: PACKING (CONTINUED)

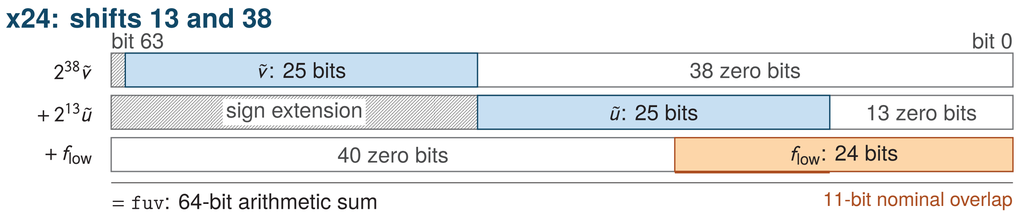
Packing as a 64-bit arithmetic sum

Each row is one addend. Blue: coefficients; ochre: low bits; hatching: sign extension. Bit 63 supplies top sign redundancy.

x20: shifts 21 and 42



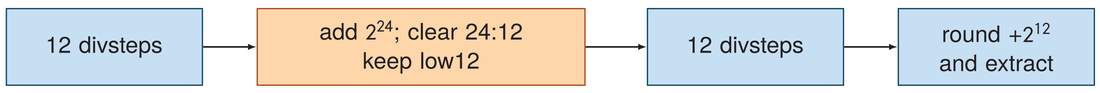
x24: shifts 13 and 38



grs uses the same layout with $(f, \tilde{u}, \tilde{v}) \mapsto (g, \tilde{r}, \tilde{s})$. Bars show initial residues; coefficients have $n - k$ trailing zeros after step k.

Why canonicalize after 12 steps? Only low12 is needed; higher residue bits could contaminate coefficient extraction.

Coefficients start at bit $13 + 12 = 25$; the signed residual is below 2^{24} in magnitude. Add 2^{24} and clear bits 24:12 to preserve coefficients and low12. This is packed-state cleanup, not reduction modulo p.



One cleanup buys four more steps per batch. The same 588 divsteps become $12 + 24 \cdot 24$ instead of $8 + 29 \cdot 20$, reducing full updates.

04 THIS WORK: FAST-FORWARDING

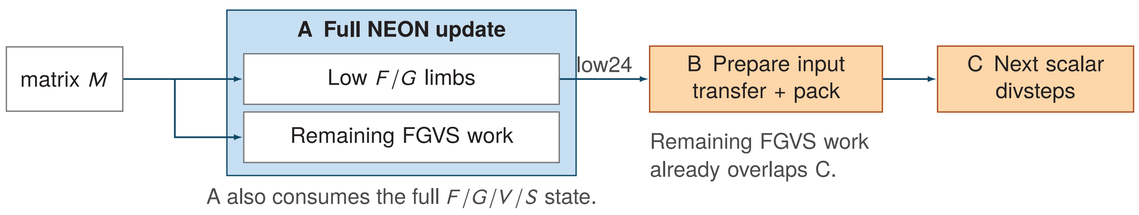
Shorten the divstep feedback path

The next scalar batch needs the new low24 before NEON finishes all limbs. Compute it directly from $f_{48} = F \bmod 2^{48}$, $g_{48} = G \bmod 2^{48}$:

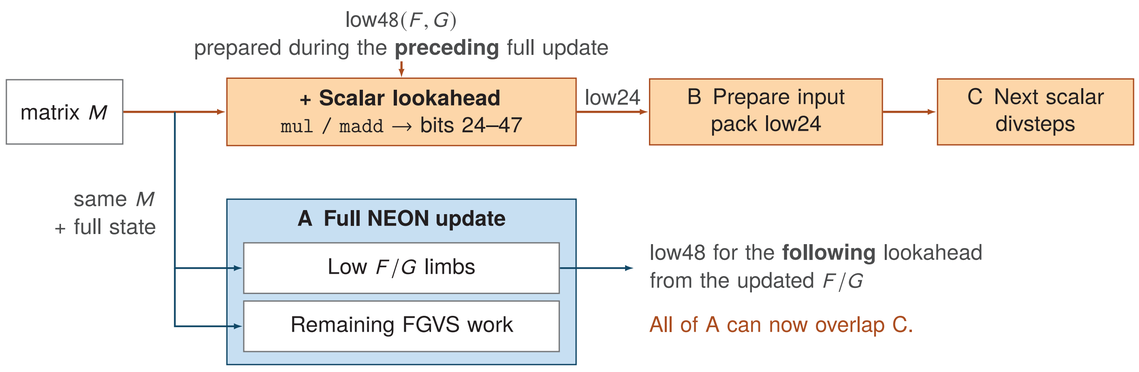
$$F' \bmod 2^{24} = \frac{(uf_{48} + vg_{48}) \bmod 2^{48}}{2^{24}}.$$

Higher bits cannot affect product bits 24–47. Each row uses scalar `mul`, `madd`, and extraction; the G' row is analogous.

ORIGINAL LOW-BIT FEEDBACK COMES FROM NEON



FF ADD A SCALAR PATH FOR THE SAME LOW BITS



Each panel follows one matrix M; arrows show data dependencies.

One matrix, two paths. Lookahead supplies the next low24; NEON still updates all limbs. Its fresh low48 feeds the *following* lookahead.

SLOTHY: register allocation and scheduling

We supply the symbolic assembly; SLOTHY first maps its named values to scalar and vector registers without reordering. A window heuristic then solves smaller scheduling problems over overlapping instruction windows. Using the specified target's latency and execution-resource model, it interleaves independent scalar divsteps/lookahead with NEON limb updates while preserving data dependencies.

05 NATIVE MEASUREMENTS

Measured cycles / inversion

12-version comparison, 2026-09-14; separate from the repo rerun.

	A72	A76
FF tuned for A72	**3,917**	3,316
FF tuned for A76	3,998	**3,272**
s2n-bignum `inv_p25519`	6,095	5,550
lib25519 `safegcdneon`	7,942	5,522

Pi 400 / A72: 1.8 GHz; Pi 5 / A76: 2.4 GHz. Native GCC 14.2. Median of six run medians; 501 samples × 1,000 calls; paired empty subtraction; two link orders; CPU 3. Warm cache, common nonzero input below 2^{255}; original wrappers included.

Reference. D. J. Bernstein et al. "Accelerating and verifying constant-time modular inversion." *EUROCRYPT 2026*, vol. 7, pp. 598–627. doi:10.1007/978-3-032-25336-1_21.
Reference. D. J. Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion." *TCHES* 2019(3), pp. 340–398.
SLOTHY. A. Abdulrahman et al. "Fast and Clean: Auditable high-performance assembly via constraint solving." *TCHES* 2024(1), pp. 87–132.
Article and code: github.com/cheng-wei-huang0612/cesare_invert_2p255m19.git

Benchmark note. Poster numbers are from a 12-version comparison (best A76: 3,272 cycles), not the public repo's three-version rerun (3,275 cycles).

EUROCRYPT 2026 TCHES 2019 SLOTHY GitHub Article & code