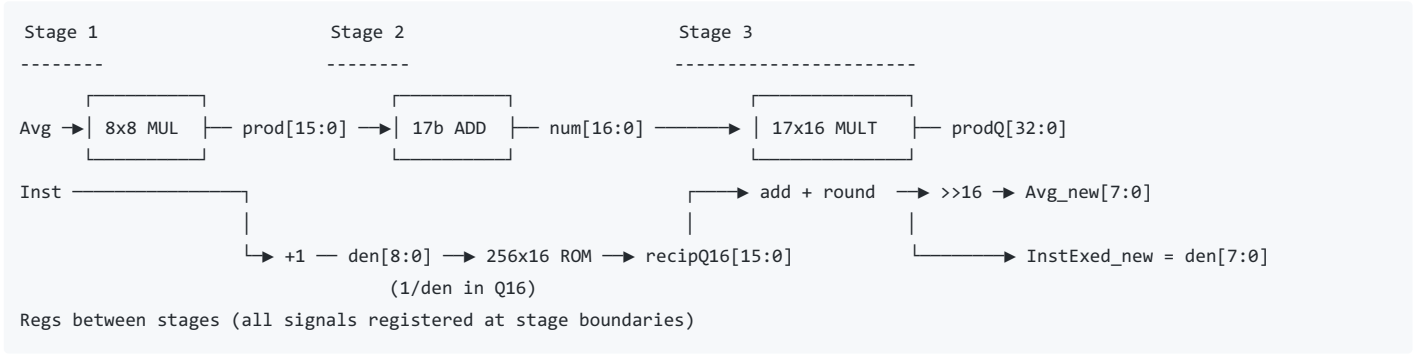
Performance-oriented (same 3-stage latency, 1 result/cycle)

Math: `AvgTxLen_new = (AvgTxLen*InstExed + CurTxLen) / (InstExed+1)` Replace `÷(InstExed+1)` with `× recipQ16(InstExed+1)` and a right shift by 16 (with rounding).

Schematic (ASCII)



Key blocks

- **Stage 1:** `mul_s1 = Avg×Inst` , `den_s1 = Inst+1` , pipeline Cur.
- **Stage 2:** `num = mul_s1 + Cur` .
- **Stage 3:** `q = ((num * recipQ16(den)) + 2^15) >> 16` (rounded), register outputs.

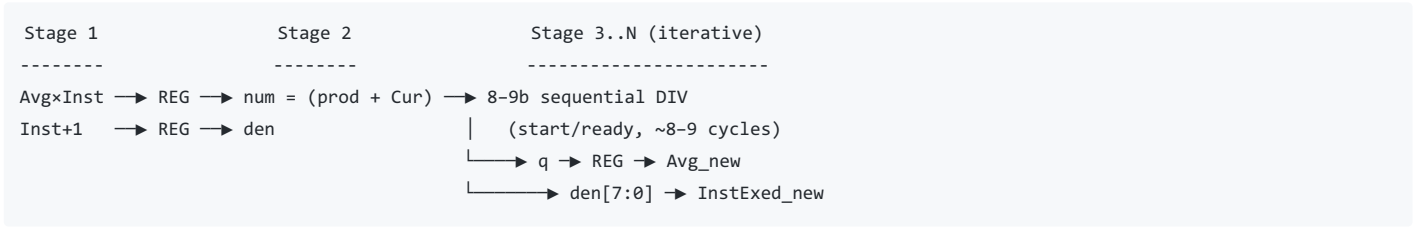
Why it helps: Pushing the slow, wide **divider** off the critical path eliminates the long carry-propagate network that caused GLS setup violations in the synthesized version with 100MHz Clock. In current design, we reduced the clock frequency to 25 MHz to meet timing. A **17×16 multiplier + add/shift** maps much better to std-cell multipliers (or DW02_mult) and retimes cleanly. The 256×16 ROM is tiny (≈4 Kb) and becomes simple combinational logic. In practice on 45 nm cells, this swap typically boosts Fmax (often 1.3–2× over a single-stage divider), reduces dynamic power (shorter chains toggle), and can even lower area because the divider is expensive while one small ROM + 1 extra multiplier is modest for 8-bit outputs.

The synthesized ALU's critical path is dominated by the Stage-3 integer divider, which is difficult to close at high frequencies in 45 nm std-cell flows. Replacing this block with a pipelined reciprocal×multiply (Q16 LUT + 17×16 multiplier + round/shift) preserves the arithmetic to 8-bit precision while transforming the critical path into multiplier and short adder logic that retimes and scales cleanly. The reciprocal LUT is tiny (256×16 ≈ 4 Kb), the pipeline depth remains three stages with **one result per cycle**, and the divider's long carry chains—and their associated timing violations and power—are eliminated. This redesign typically increases Fmax, reduces toggling on long paths, and can even lower total area versus a wide combinational divider, delivering a better PPA point without changing the external interface or latency.

Area-oriented (multi-cycle divider, smaller area)

If throughput (1 result/cycle) isn't mandatory, keep the original math but replace DIV with a small **sequential divider** (e.g., restoring or `DW_div_seq`) and add a tiny valid/ready handshake.

Schematic (ASCII)



Why it helps: A small iterative divider plus a controller is **much smaller** than a fast combinational divider. You trade CPI (e.g., one result every ~8–9 cycles) for area/power. Great for resource-constrained designs or when this ALU isn't on the top-throughput path.