**Advantages of Pipelined restoring Division:**

**Simpler Control Logic:**

- Pipelined restoring division uses a relatively straightforward sequence of additions, subtractions, and shifts, making the control logic easier to design compared to SRT or Goldschmidt, which involve more complex operations like partial quotient selection or multiplication.

**Hardware Friendly:**

- It is suitable for fixed-point arithmetic and can be implemented with simple hardware without the need for high-speed multipliers or floating-point units, which are required for Newton-Raphson and Goldschmidt methods.

**Disadvantages of pipelined restoring division:**

**Slower Convergence:**

- Pipelined restoring division is typically slower than iterative methods like Newton-Raphson and Goldschmidt, which converge quadratically and are better suited for high-performance systems.

**Not Ideal for Floating-Point:**

- It's less efficient or practical for floating-point division due to precision and normalization issues. Methods like SRT are more optimized for IEEE floating-point standards.

**More Clock Cycles:**

- Since Pipelined restoring division performs one operation per bit of the divisor, it can require more clock cycles than Goldschmidt or Newton-Raphson, which compute multiple bits per iteration using multiplication.

**The comparison table:**

Here is the comparison table that compares our method (pipelined restoring) with other methods such as SRT, Newton-Raphson and Goldschmidt.

**Comparison of Division Algorithms:**

| Feature / Method | Pipelined Restoring Division | SRT (e.g., Radix-4) | Newton-Raphson | Goldschmidt |
|---|---|---|---|---|
| **Basic Principle** | Repeated add/subtract & shift | Digit recurrence with lookup table | Iterative reciprocal approximation | Parallel scaling of numerator & denominator |
| **Latency (32-bit)** | ~32 cycles | ~10–20 cycles | ~4–6 cycles | ~3–5 cycles |
| **Convergence Rate** | Linear (1 bit per iteration) | Logarithmic (radix-dependent) | Quadratic | Quadratic |
| **Accuracy (per iteration)** | 1 bit | ~$\log_2$(radix) bits | Doubles each iteration | Doubles each iteration |
| **Iterations Required (32-bit)** | ~32 | ~5 (radix-4) | 3–4 iterations | 3–4 iterations |
| **Initial Approximation** | Not required | Table lookup | Required (LUT or estimate) | Required (LUT or estimate) |
| **Hardware Complexity** | Low | Medium | High (requires multipliers) | High (optimized for parallelism) |
| **Floating-Point Suitability** | Low | Moderate | Excellent | Excellent |
| **Typical Applications** | Simple processors, microcontrollers | General-purpose processors | High-performance CPUs | GPUs, SIMD/parallel processors |

**HANDWORKED EXAMPLE:**

# HAND WORKED EXAMPLE

## Assumptions :-

XLEN = 4

STAGE_LIST = 4'b1111 (all stages enabled)

DIVIDEND (a) = 13 = 1101

DIVISOR (b) = 3 = 0011

## cycle - wise computation Table :-

| cycle | dividend [i] | m (dividend >> shift) | n (divisor) | q(m >= n) | t(m-n) | d(new dividend) | quotient[i+1] |
|---|---|---|---|---|---|---|---|
| 0 | 1101 | 0001 | 0011 | 0 | 0000 | 0110 | 0000 |
| 1 | 0110 | 0011 | 0011 | 1 | 0000 | 0011 | 0010 |
| 2 | 0011 | 0010 | 0011 | 0 | 0000 | 1001 | 0010 |
| 3 | 1001 | 1001 | 0011 | 1 | 0110 | 0110 | 0011 |

## Final Result :-

Final quotient (quo) = 0100 = 4

Final remainder (rem) = 0001 = 1

## check :-

13 / 3 = 4, remainder = 1 ✓

**CODE AND IMPLEMENTATION:**

Github: https://github.com/AnanthNaik/EE311/tree/main

**Division Algorithms:**

## 1. SRT Division

➔ Type: Digit recurrence method.

➔ Process: Iteratively computes the quotient one digit at a time, allowing quotient digits to be in a redundant form (e.g., {-1, 0, 1}).

➔ Key Formula:

$$r_{i+1} = r_i - q_i \cdot d \cdot \beta^{-i}$$

$$Q = \sum_{i=1}^{n} q_i \cdot \beta^{-i}$$

➔ Where:

- `r_i` is the current remainder
- `q_i` is the quotient digit (can be -1, 0, or +1 in radix-2, or more digits in higher radices).
- `d` is the divisor.
- β is the radix (e.g., 2, 4, 8).
- `Q` is the final quotient.

➔ Process:

- Normalize `dividend` and `divisor` (make divisor in [0.5, 1]).
- Initialize `r_0 = dividend`, `Q = 0`.
- Use lookup tables or decision logic to determine `q_i` from `r_i` and `d`.
- Update `r_{i+1}` and accumulate `q_i` into the final quotient.
- Repeat for required precision.
- Convert redundant results to binary if needed.

➔ Features:

- Uses lookup tables or decision logic to select quotient digits.
- Handles one digit per cycle; moderate speed.

- Common in FPUs due to balanced performance and complexity.

## 2. Newton-Raphson Division

➔ Type: Iterative approximation method.

➔ Process: Converts division into reciprocal approximation using:

$$x_{n+1} = x_n(2 - B \cdot x_n)$$

$$Q = A \cdot x_n$$

Where, Q is the final quotient.

➔ Features:

- Fast convergence (doubles precision each step).
- Requires good initial estimate and multiple multipliers.
- Efficient for scalar floating-point units and CPUs.

## 3. Goldschmidt Division

➔ Type: Iterative approximation method.

➔ Process: Simultaneously scales both dividend and divisor to force the divisor toward 1:

Q = A/B, F = scaling factor to bring B →1.

$$F_i = 2 - B_i \cdot x_i$$

$$A_{i+1} = A_i \cdot F_i, \quad B_{i+1} = B_i \cdot F_i$$

Q = An.

Where , Q is the final quotient and x_0 is initial approximation.

➔ Features:

- High parallelism; suitable for SIMD/GPU architectures.
- Fully multiplication-based; no division or subtraction required in loop.
- Converges quickly and efficiently in hardware.