SIGMA MALES : IMPLEMENTATION OF SLOW AND FAST

DIVISION ALGORITHMS IN COMPUTER ARCHITECTURE

**Vishnu.s** *Computer Science Rajalakshmi Institute of Technology Chennai, India* [vishnu.s.2021.cse@ritcennai.edu.in](mailto:vishnu.s.2021.cse@ritcennai.edu.in)

**Veerendran.s**

*Computer Science Rajalakshmi Institute of Technology Chennai, India* [veerendran.s.2021.cse@ritchennai.edu.in](mailto:veerendran.s.2021.cse@ritchennai.edu.in)

**Sujith Kumar.T**

*Computer Science Rajalakshmi Institute of Technology Chennai, India* [sujithkumar.t.2021.cse@ritchennai.edu.in](mailto:sujithkumar.t.2021.cse@ritchennai.edu.in)

**O.pandithurai**

Department of Computer Science and Engineering

Rajalakshmi Institute of Technology [pandics@ritchennai.edu.in](mailto:pandics@ritchennai.edu.in)

**G.Saikrishna**

Associate professor

Department of Mechanical Engineering Rajalakshmi Institute of Technology Saikrishna.g@ritchennai.edu.in

**S.vivek**

Assistant professor

Department of Mechanical Engineering Rajalakshmi Institute of Technology vivek.s@ritchennai.edu.in

**ABSTRACT:**

Many algorithms have been developed for implementing division in hardware. These algorithms differ in many aspects, including quotient convergence rate, fundamental hardware primitives, and mathematical formulations. This paper presents a taxonomy of division algorithms which classifies the algorithms based upon their hardware implementations and impact on system design. Division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. Many practical division algorithms are hybrids of several of these classes. These algorithms are explained and compared in this work. It is found that for low-cost implementations where chip area must be minimized, digit recurrence algorithms are suitable. An implementation of division by functional iteration can provide the lowest latency for typical multiplier latencies. Variable latency algorithms show promise for simultaneously minimizing average latency while also minimizing area.

**INTRODUCTION:**

A division algorithm provides a quotient and a remainder when we divide two numbers. They are generally of two types: slow algorithm and fast algorithm. Slow division algorithm are restoring, non-restoring, non-performing restoring, SRT algorithm and under fast comes Newton–Raphson and Goldschmidt. When we perform division operations on two numbers, the division algorithm will give us two things, i.e., quotient and remainder. This algorithm is based on the assumption that 0 < D < N. With the help of digit set {0, 1}, the quotient digit q will be formed in the restoring division algorithm. Division algorithms can be divided into classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. . A common perception of division is that it is an infrequent operation whose implementation need not receive high priority. However, it has been shown that ignoring its implementation can result in significant system performance degradation for many applications. This study synthesizes the fundamental aspects of these and other works, in order to clarify the division design space. The five classes of division algorithms are presented and analyzed in terms of the three major design parameters: latency in system clock cycles, cycle time, and area. Other issues related to the implementation of division in actual systems are also presented. Throughout this work, the majority of the discussion is devoted to division. The theory of square root computation is an extension of the theory of division.

**LITERATURE SURVEY:**

1. Slow Division Algorithms:

\* Long Division: The traditional method of division, which involves repeated subtraction and brings out one digit of the quotient at a time.

\* Restoring Division: An iterative algorithm that performs multiple subtractions to find the quotient and remainder.

\* Non-Restoring Division: Similar to restoring division, but instead of correcting for over-subtractions, it corrects for under-subtractions. Restoring division is an iterative algorithm used to perform division operations. It was commonly used in early computer systems due to its simplicity.

1. Overview of Restoring Division:

\* Explanation of the basic restoring division algorithm.

\* Description of the steps involved in the algorithm, including initialization, iterative subtraction, and quotient generation.

\* Representation of numbers (dividend, divisor, quotient, and remainder) in the algorithm.

2. Variants and Optimizations:

\* Non-Restoring Division: Comparison with restoring division, highlighting the differences in the correction process.

\* Bit-Pair Recoding: A technique that reduces the number of iterations required in restoring division.

\* Signed-Digit Representation: Using signed digits to speed up the division process.

\* Pipelined Restoring Division: Optimizations for parallel processing and improved throughput.

3. Performance Evaluation and Trade-offs:

\* Complexity analysis of restoring division in terms of time and hardware resources.

\* Comparison of restoring division with other division algorithms, such as non-restoring division, SRT division, or Goldschmidt division, in terms of speed and efficiency.

4. Hardware Implementations:

\* Design considerations for hardware implementations of restoring division.

\* High-level architectures, including serial and parallel approaches.

5. Error Analysis and Numerical Stability:

\* Discussion on the impact of rounding errors and precision limitations in restoring division.

\* Analysis of error propagation and accumulation in iterative division algorithms.

\* Techniques to improve numerical stability and error control in restoring division.

2. Fast Division Algorithms:

\* Newton-Raphson Division: Based on iterative approximation, this algorithm uses Newton's method to refine the quotient estimation.

\* SRT Division (Sweeney, Robertson, and Tocher Division): A radix-2 algorithm that uses shift and add operations for faster division.

\* Goldschmidt Division: An iterative algorithm that uses multiplicative inverses to compute the reciprocal of the divisor and then performs multiplication.

\* Barrett Division: A technique that uses precomputed values to reduce the division problem into multiple smaller divisions, making it faster. Goldschmidt division is an iterative algorithm used for division computations. It is known for its speed and efficiency, particularly in hardware implementations.

1. Overview of Goldschmidt Division:

\* Explanation of the basic Goldschmidt division algorithm.

\* Description of the iterative steps involved, including initial approximation, reciprocal computation, and refinement.

\* Representation of numbers (dividend, divisor, quotient, and remainder) in the algorithm.

2. Variants and Optimizations:

\* Multiplier less Goldschmidt Division: Techniques to eliminate the use of multipliers, reducing hardware complexity.

\* Pipelined Goldschmidt Division: Optimizations for parallel processing and improved throughput.

\* Scaled Goldschmidt Division: Variations that employ scaling to handle numbers with large dynamic range.

3. Comparison with Other Division Algorithms:

\* Performance comparison of Goldschmidt division with other division algorithms like restoring division, non-restoring division, or SRT division.

4. Hardware Implementations:

\* Design considerations for hardware implementations of Goldschmidt division.

\* High-level architectures, including serial and parallel approaches.

\* Comparison of different hardware implementations in terms of performance metrics, area requirements, and power consumption.

5. Error Analysis and Numerical Stability:

\* Discussion on the impact of rounding errors and precision limitations in Goldschmidt division.

\* Analysis of error propagation and accumulation in iterative division algorithms.

\* Techniques to improve numerical stability and error control in Goldschmidt division.

6. Applications and Use Cases:

\* Exploration of practical applications where Goldschmidt division is advantageous, such as signal processing, cryptography, or scientific computing.

\* Case studies or examples showcasing the benefits of Goldschmidt division in real-world scenarios.

**OBEJECTIVE:**

\* The main objective of division is a mathematical operation that gives a quotient and a remainder. We will see the hardware implementation of the Division algorithm in computer architecture.

\*We use registers and counters while performing division.

**OUTCOMES:**

The outcomes or results of using slow and fast division algorithms can vary based on factors such as the specific algorithm used, the input data, and the implementation context (software or hardware). Here are some general outcomes associated with slow and fast division algorithms:

1. Slow Division Algorithm Outcomes:

\* Accuracy: Slow division algorithms typically provide accurate results with high precision, as they perform division iteratively using precise arithmetic operations.

\* Efficiency: Slow division algorithms tend to be less efficient in terms of speed and computational resources. They may require more iterations and have higher latency compared to fast division algorithms.

\* Resource Usage: Slow division algorithms may require more memory and computational resources, making them less suitable for applications with limited resources.

\* Versatility: Slow division algorithms are often applicable to a wide range of division scenarios, including both integer and floating-point divisions.

2. Fast Division Algorithm Outcomes:

\* Speed: Fast division algorithms are designed to reduce the number of iterations or operations required for division, resulting in faster computation compared to slow division algorithms.

\* Efficiency: Fast division algorithms can provide significant speed improvements, making them ideal for applications that require high-performance division operations.

\* Resource Usage: Fast division algorithms often require specialized hardware or optimized software implementations to achieve their speed advantages. They may require more complex circuitry or additional computational resources.

\* Accuracy: Depending on the specific algorithm and implementation, fast division algorithms may introduce some approximation or rounding errors, especially when dealing with floating-point division.

It's important to note that the choice between slow and fast division algorithms depends on the specific requirements and constraints of the application.

**CHALLANGES:**

Using both slow and fast division algorithms can present certain challenges depending on the specific algorithm, implementation context, and application requirements. Here are some challenges associated with using slow and fast division algorithms:

Challenges of Slow Division Algorithms:

1. Computational Efficiency: Slow division algorithms often require a large number of iterations and arithmetic operations, resulting in higher latency and increased computational complexity.

2. Resource Usage: Slow division algorithms may consume more memory and computational resources, making them less suitable for applications with limited resources.

3. Real-Time Processing: The slower execution time of slow division algorithms may pose challenges in real-time systems or applications that demand fast response times.

4. Scalability: Slow division algorithms may not scale well for large input sizes or when dealing with complex mathematical operations.

5. Hardware Implementation: Translating slow division algorithms into hardware can be challenging due to their sequential nature and resource requirements.

Challenges of Fast Division Algorithms:

1. Complexity: Fast division algorithms can be more complex to implement compared to slow division algorithms. They may require specialized hardware or optimized software implementations.

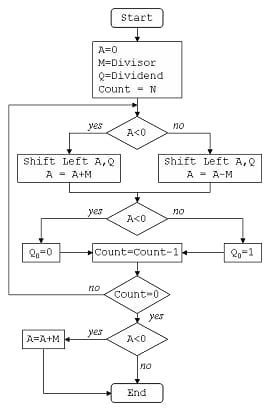
2. Accuracy and Precision: Some fast division algorithms introduce approximation techniques to achieve faster computation, which can lead to slight errors or loss of precision, especially in floating-point division.

3. Hardware Complexity: Fast division algorithms may require additional circuitry or computational resources, resulting in increased hardware complexity and potentially higher costs.

4. Algorithm Selection: Choosing the most suitable fast division algorithm for a specific application can be challenging, as different algorithms have varying performance characteristics and trade-offs.

5. Limited Applicability: Some fast division algorithms may have limitations in terms of the types of numbers they can handle or the specific division scenarios they are optimized for.

**ARCHITECTURE/SYSTEM MODEL**:

BLOCK DIAGRAM FOR SLOW DIVISION ALGORITHM

EXPLANATION OF ABOVE SLOW DIVISION ALGORITHM:

• SLOW division algorithms produce one digit of the final quotient per iteration.

• Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division.

• The restoring division algorithm is a slow division algorithm that calculates the quotient digit by digit.

• This algorithm will generate a quotient and a remainder after the division algorithm.

with the assumption that the dividend is greater than the divisor.

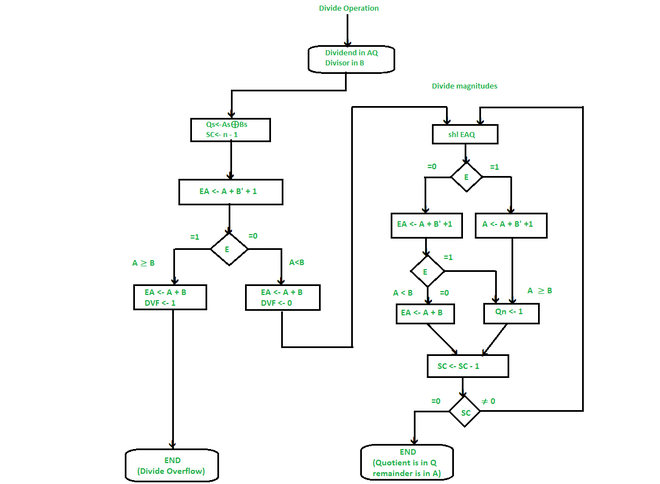
• This division algorithm in computer architecture uses three registers:

~Register A is initialized to 0

~Register Q stores the dividend

~Register M stores the divisor. N is used as a counter

BLOCK DIAGRAM FOR FAST DIVISION ALGORITHM



**EXPLANATION OF ABOVE FAST DIVISION ALGORITHM**

• The quotient is predicted to the closest approximation to the actual quotient, and then the calculation starts.

• The algorithms for fast division category are Newton- Raphson, and Goldschmid

• Newton-Raphson uses Newton's method

• Goldschmidt division is iterative division algorithm deployed in many processors

• Higher precision can be achieved by adding fraction bits for intermediate calculations or by having more iterations.

**HARDWARE/SOFTWARE MODEL GOING TO USE FOR IMPLEMENTATION:**

Intel Quartus Prime, which is a software suite used for designing and programming Intel Field-Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). Quartus Prime provides a comprehensive environment for designing digital systems, including RTL (Register Transfer Level) design, synthesis, simulation, and implementation.

Intel Quartus Prime offers various features and tools to facilitate FPGA design, including:

1. Design Entry: Quartus Prime supports different design entry methods, such as HDL (Hardware Description Language) coding using Verilog or VHDL, as well as schematic-based design entry.

2. Synthesis and Optimization: The software includes a synthesis tool that converts the high-level RTL description into a gate-level netlist. It also performs optimizations to improve design performance, area utilization, and power consumption.

3. Simulation: Quartus Prime supports simulation of the design using Model Sim, a widely used HDL simulator. This enables designers to verify the functionality of their designs before proceeding to the implementation phase.

4. Place and Route: Quartus Prime's implementation tools perform place and route algorithms to map the design onto the target FPGA device. This process determines the physical placement of logic elements, routing of interconnects, and optimization of timing constraints.

5. Timing Analysis and Optimization: The software offers timing analysis tools to verify and optimize the design's performance with respect to critical paths, setup/hold time requirements, and other timing constraints.

6. Programming and Configuration: Quartus Prime provides utilities to program the FPGA or CPLD devices, allowing the synthesized design to be loaded onto the hardware for testing and deployment.

7. Debugging and Verification: The software integrates debugging features and interfaces to facilitate the identification and resolution of design issues. It also supports advanced verification methodologies, such as System Verilog assertions and code coverage analysis.

Intel Quartus Prime supports a wide range of FPGA families, including Intel Arria, Intel Cyclone, and Intel Stratix series. The software is regularly updated with new features,

It's important to note that Quartus Prime is a powerful and complex tool suite, and utilizing its features effectively requires familiarity with digital design concepts, FPGA architectures, and the software itself. Intel provides comprehensive documentation, user guides, and tutorials to assist designers in using Quartus Prime for their FPGA design projects.

Model Sim provides support for simulating and testing designs described in HDL languages such as Verilog and VHDL. It offers a range of features for functional verification, including:

1. Simulation Environment: Model Sim provides a graphical user interface (GUI) where users can interactively create, manage, and run simulation testbenches. It allows users to define and control simulation stimuli, monitor signals, and analyze simulation results.

2. Language Support: Model Sim supports both Verilog and VHDL languages, enabling users to simulate and verify designs described in either language. It includes language-specific compilers and simulators for efficient simulation.

3. Waveform Viewer: Model Sim incorporates a waveform viewer, which allows users to visualize and analyze waveforms generated during simulation. The waveform viewer enables users to track and monitor signals, set breakpoints, and debug their designs.

4. Advanced Debugging Features: Model Sim provides advanced debugging capabilities, such as signal tracing, breakpoints, and single-step execution. These features help users identify and resolve issues within their designs.

5. Code Coverage Analysis: Model Sim offers code coverage analysis tools that assess the completeness of testbench coverage in terms of code execution. This helps users ensure thorough verification of their designs.

6. Support for Assertion-Based Verification: Model Sim supports System Verilog Assertions (SVA) and VHDL assertions, allowing users to specify properties and constraints to be verified during simulation.

7. Performance Optimization: Model Sim includes optimization features to enhance simulation performance. These optimizations can accelerate simulation speed and improve overall productivity.

When it comes to Verilog HDL, there are several online compilers and simulators available that allow you to write and simulate Verilog code without installing any specialized tools. Here are a few examples:

1. EDA Playground: EDA Playground (<https://www.edaplayground.com/>) is a widely used online platform for designing and simulating digital circuits. It supports Verilog as well as System Verilog and VHDL. EDA Playground provides a web-based IDE with simulation capabilities and allows you to write Verilog code, compile it, simulate the design, and view waveforms.

2. Circuit Verse: Circuit Verse (<https://circuitverse.org/>) is an online platform primarily focused on digital circuit design and simulation. It supports both block-based and text-based design methodologies, including Verilog. You can design and simulate Verilog circuits using Circuit Verse's intuitive web interface.

3. JDoodle: JDoodle (<https://www.jdoodle.com/>) is an online coding platform that supports various programming languages, including Verilog. It provides a simple web-based editor where you can write your Verilog code, and then compile and run it to see the output.

4. EDA.io: EDA.io (<https://eda.io/>) is a cloud-based development environment specifically designed for hardware engineers. It supports Verilog, System

Verilog, and VHDL, and provides an online IDE for writing, compiling, and simulating hardware designs.

Please note that while online compilers are convenient for quick testing and experimentation, they may have limitations in terms of the size and complexity of the designs they can handle,

**CONCLUSION:**

In this article, we learned about the division algorithm in computer architecture which is the Restoring Algorithm. The classes are determined by the differences in the fundamental operations used in the hardware implementations of the algorithms. The simplest and most common class found in the majority of modern processors that have hardware division support is digit recurrence, specically SRT. Additionally, for small radices, it has been possible to meet the tight cycle-time requirements of high performance processors without requiring large amounts of die area. The disadvantage of these SRT implementations is their relatively high latency, as they only retire 1-3 bits of result per cycle. As processor designers continue to seek an ever-increasing amount of system performance, it becomes necessary to reduce the latency of all functional units, including division. However, if a pipelined multiplier is used throughout the iterations, more than one division operation can proceed in parallel. For implementations with high division throughput requirements, the Newton-Raphson iteration provides a means for trading latency for throughput.

**REFERENCES**

[1] SPEC benchmark suite release 2/92.

[2] Microprocessor Report, various issues, 1994-96.

[3] S. F. Oberman and M. J. Flynn, “Design issues in division and other floating-point operations, "To appear in IEEE Trans. Computers, 1997.

[4] C. V. Freiman, “Statistical analysis of certain binary division algorithms,” IRE Proc., vol. 49, pp.91-103, 1961.

[5] J. E. Robertson, “A new class of digital division methods, "IRE Trans. Electronic Computers, vol.EC-7, pp. 218-222, Sept. 1958.

[6] K. D. Tocher, “Techniques of multiplication and division for automatic binary computers,” Quart. J.Mech. Appl. Math., vol. 11, pt. 3, pp. 364-384, 1958.

[7] D. E. Atkins, \Higher-radix division using estimates of the divisor and partial remainders," IEEETrans. Computers, vol. C-17, no. 10, Oct. 1968.[8] K. G. Tan, “The theory and implementation of high-radix division,” in Proc. 4th IEEE Symp.Computer Arithmetic, pp. 154-163, June 1978.

[9] M. D. Ercegovac and T. Lang, Division and Square Root: Digit-Recurrence Algorithms and implementations, Boston: Kluwer Academic Publishers, 1994.

[10] M. Flynn, “On division by functional iteration,” IEEE Trans. Computers, vol. C-19, no. 8, Aug. 1970.

[11] P. Soderquist and M. Leeser, “An area/performance comparison of subtractive and multiplicative divide/square root implementations,” in Proc. 12th IEEE Symp. Computer Arithmetic, pp. 132-139,July 1995.

[12] “IEEE standard for binary floating point arithmetic,” ANSI/IEEE Std 754-1985, New York, The institute of Electrical and Electronic Engineers, Inc., 1985.

[13] S. Oberman and M. Flynn, “Measuring the complexity of SRT tables,” Tech. Rep. CSL-TR-95-679,Comput. Syst. Lab., Stanford Univ., Stanford, CA, Nov. 1995.

[14] M. D. Ercegovac and T. Lang, “Simple radix-4 division with operands scaling,” IEEE Trans. Computers, vol. C-39, no. 9, pp. 1204-1207, Sept. 1990.

[15] J. Fand Rianto , “Algorithm for high-speed shared radix 8 division and radix 8 square root,” in Proc.9th IEEE Symp. Computer Arithmetic, pp. 68-75, July 1989.

[16] S. E. McQuillan, J. V. McCanny, and R. Hamill, “New algorithms and VLSI architectures for SRT division and square root,”Proc. 11th IEEE Symp. Computer Arithmetic, pp. 80-86, July 1993

[17] P.Montushi and L. Ciminiera, “Reducing iteration time when result digit is zero for radix 2 SRT division and square root with redundant remainders,” IEEE Trans. Computers, vol. 42, no. 2, pp.239-246, Feb. 1993.

[18] P.Montuschi and L.Ciminiera, “Over-redundant digit sets and the design of digit-by-digit division units,” IEEE Trans. Computers, vol. 43, no. 3, pp. 269-277, Mar. 1994.49To appear in IEEE Transactions on Computers 1997

[19] P. Montuschi and L. Ciminiera, “Radix-8 division with over-redundant digit set,” J. VLSI SignalProcessing, vol. 7, no. 3, pp. 259{270, May 1994.

[20] S. Oberman, N. Quach, and M. Flynn, “The design and implementation of a high-performance floating-point divider,”Tech. Rep. CSL-TR-94-599, Comput. Syst. Lab., Stanford Univ., Stanford,CA, Jan. 1994.

[21] N. Quach and M. Flynn, “A radix-64 floating-point divider,” Tech. Rep. CSL-TR-92-529, Comput.Syst. Lab., Stanford Univ., Stanford, CA, June 1992.

[22] H. Srinivas and K. Parhi, “A fast radix-4 division algorithm and its architecture,” IEEE Trans.Computers, vol. 44, no. 6, pp. 826-831, June 1995.