Design and Implementation of Slow and Fast Division algorithm in Computer Architecture using verilog

Mr .Franklin Telfer
Assistant Professor
Department of Electronic andCommunication
Rajalaskmi Institute of Technology
Chennai,India
franklintelfer@ritchennai.edu.in

Vishalakshi.V
Electronics and communication
Rajalakshmi Institute Of Technology
Chennai,India
vishalakshi.v.2021.ece@ritchennai.edu.in

Vishnupriya.V
Electronics and communication
Rajalakshmi Institute of Technology
Chennai, India
vishnupriya.v.2021.ece@ritchennai.edu.in

Yuvaraj.D
Electronics and communication
Rajalakshmi Institute of Technology
Chennai,India
yuvarajd.2021.ece@ritchennai.edu.in

Abstract— Many algorithms have been developed for implemening division in hardware. These algorithm in many aspects, including quoent convergence rate, fundamental hardware Many algorithms have been developed for implemen ng division in hardware. These algorithms in many aspects, including quoent convergence rate, fundamental hardware primives, and mathema cal formula ons. This paper presents a taxonomy of division algorithms which classi the algorithms based upon their hardware implementa ons and impact on system design. Division algorithms can be divided into classes: digit recurrence, func onal itera on, very high radix, table lookup, and variable latency. Many prac cal division algorithms are hybrids of several of these classes. These algorithms are explained and compared in this work. It is found that for lowcost implementaions where chip area must be minimized, digit recurrence algorithms are suitabl.

Index Terms: Computer arithmetic, division, oa ng point, func onal itera on, SRT, table look-up, variable latency, very high radix .

I. INTRODUCTION

In recent years computer applica ons have increased in their computa onal complexity. The industry-wide usage of performance benchmarks, such as SPECmarks
[1] .Forces designers of general-purpose microprocessors to pay par cular a en on to implementa on of the oa ng point unit, or FPU. Special purpose applica ons, such as high performance graphics rendering systems, have placed further demands on processors. High speed oa ng point hardware is a requirement to meet these increasing demands. Modern applica ons comprise several oa ng point opera ons including addi on, mul plica on, and division. In recent FPUs, emphasis has been placed on designing everfaster adders and mul pliers, with division receiving less a en on. Typically, the range for addi on latency is 2 to 4 cycles, and the range for mul plica on is 2 to 8 cycles. In contrast, the latency for double precision division in modern FPUs ranges frs than 8 cycles to over 60 cycles . [2]. A common percep on of division is that it is an infrequent opera on whose implementa on need not receive high priority. However, it has been shown that ignoring its implementa on can result in signi system performance degrada on for many applica ons.
[3]. Extensive literature exists describing the theory of division. However, the design space of the algorithms and implementa ons is large due to the large number of parameters involved. Furthermore, deciding upon an op mal design depends heavily on its requirements. studies on Fashion MNIST and analyse their methodologies, procedures, and performance indicators. This review presents the framework for our inves ga on and an indepth analysis of the state-of-theart for defea ng the Fashion
The most common implementa on of digit recurrence division in modern processors has been named SRT division by Freima taking its name from the ini als of Sweeney, Robertso and Tocher who discovered the algorithm independently in approximately the same me period.

II. LITERATURE SURVEY

A literature survey on the implementa on of slow and fast division algorithms in computer architecture using Verilog can involve exploring research papers, conference proceedings,

and relevant publica ons in the field. While I cannot provide an exhaus ve review of all the exis ng literature, I can give you an overview of common slow and fast division algorithms and suggest poten al sources to delve into further.

1. Slow Division Algorithms:

a. Restoring Division: Restoring division is a straigh orward algorithm that repeatedly subtracts the divisor from the dividend un l the remainder becomes nega ve. The quo ent is obtained by coun ng the number of subtrac ons performed. You can refer to "Computer Organiza on and Design" by David A. Pa erson and John L. Hennessy for an understanding of restoring division algorithms in computer architecture.

b. Non-Restoring Division: Non-restoring division is an improvement over restoring division that eliminates the need to adjust the remainder. It uses a series of condi onal addi ons and subtrac ons to compute the quo ent. "Digital Design and Computer Architecture" by David Harris and Sarah Harris provides insights into non-restoring division algorithms.

2. Fast Division Algorithms:

a. SRT Division: SRT (Sweeney, Robertson, and Tocher) division algorithm is an itera ve method that provides faster division by using a precomputed table of par al remainders. This algorithm reduces the number of itera ons required to compute the quo ent. The paper "Radix-4 SRT Division Algorithms" by M. N. Hossain and A. V. Deshmukh explains radix-4 SRT division in the context of computer architecture and Verilog implementa on.

b. Newton-Raphson Division: Newton-Raphson division is a non-itera ve algorithm that uses an itera ve approxima on technique to compute the reciprocal of the divisor. The reciprocal is then mul plied by the dividend to obtain the quo ent. "Digital Computer Electronics" by Albert Paul Malvino and Jerald A. Brown covers the Newton-Raphson algorithm in the context of computer architecture and Verilog implementa on.

To explore further, you can search for academic papers on pla orms like IEEE Xplore, ACM Digital Library, or Google Scholar using keywords such as "division algorithm," "computer architecture," "Verilog implementa on," and the specific algorithm names men oned above. Addi onally, textbooks on computer architecture and digital systems design can provide valuable insights into these algorithms and their Verilog implementa ons.

III. OBJECTIVES

The objec ve of both slow and fast division algorithms is to efficiently compute the quo ent and remainder when dividing one number (the dividend) by another number (the divisor). However, they differ in their approach and performance characteris cs:

1. Slow Division Algorithm:

- Objec ve: The slow division algorithm aims to provide a basic implementa on of division by repeatedly subtrac ng the divisor from the dividend un l the remainder is less than the divisor.
- Use cases: This algorithm is commonly used in basic so ware implementa ons or situa ons where performance is not a cri cal factor.
- Advantages: Simple to implement and understand.
- Disadvantages: Inefficient for large numbers, as it requires a large number of itera ons and subtrac on opera ons.

2. Fast Division Algorithm:

- Objec ve: The fast division algorithms aim to improve the efficiency of division opera ons by reducing the number of itera ons required.
- Use cases: Fast division algorithms are beneficial in performance-cri cal scenarios or applica ons that frequently perform division opera ons.
- Advantages: Improved efficiency and reduced computa onal complexity compared to slow division algorithms.
- Disadvantages: Fast division algorithms are o en more complex to implement and may require addi onal precomputa on steps.

The overall objec ve of both algorithms is to accurately compute the quo ent and remainder of a division opera on. However, the fast division algorithms seek to achieve this objec ve with improved performance by minimizing the number of opera ons or u lizing mathema cal proper es.

IV. OUTCOMES

The outcomes of implemen ng slow and fast division algorithms on a computer architecture can vary depending on various factors such as the specific algorithm used, the characteris cs of the computer architecture, and the input data. Here are some poten al outcomes:

1. Slow Division Algorithm:

- Simple implementa on: The slow division algorithm is rela vely straigh orward to implement on most computer architectures.
- Slower execu on: Due to the repe ve subtrac on

compared to fast division algorithms, especially for large numbers.

- Higher computa onal resource usage: The slow division algorithm may require more computa onal resources, such as CPU cycles and memory, due to the repeated itera ons.

2. Fast Division Algorithm:

- Improved performance: The fast division algorithm aims to reduce the number of itera ons and op mize the division opera on, leading to faster execu on mes compared to the slow division algorithm.
- Lower computa onal resource usage: Fast division algorithms o en u lize specialized techniques, precomputed tables, or hardware features to minimize computa onal resource usage, resul ng in more efficient division opera ons.
- Complexity trade-off: Fast division algorithms can be more complex to implement than slow division algorithms, requiring addi onal preprocessing steps or advanced mathema cal techniques.

Addi onal outcomes and considera ons:

- Division accuracy: Both slow and fast division algorithms should produce the correct quo ent and remainder according to the division opera on.
- Trade-offs: The choice of algorithm may involve trade-offs between execu on speed, implementa on complexity, and resource usage, depending on the specific requirements and constraints of the applica on.
- Hardware op miza on: Both algorithms can benefit from hardware op miza ons, such as parallelism, pipelining, or SIMD instruc ons, to further improve their performance on specific computer architectures.
- Benchmarking and profiling: It's important to evaluate the outcomes by benchmarking and profiling the implemented algorithms to measure their actual performance on the target architecture and op mize further if needed.

It's crucial to note that the specific outcomes will depend on the characteris cs of the computer architecture, the efficiency of the algorithm implementa on, and the specific input data being processed.

V.CHALLENGES

Implemen ng slow and fast division algorithms can present various challenges depending on the specific algorithm being used. However, here are some common challenges you might encounter:

1. Understanding the algorithm: Slow and fast division algorithms are typically more complex than straigh orward division methods. It is crucial to thoroughly understand the algorithm you're implemen ng before diving into the code. Take the me to study the algorithm and any associated mathema cal concepts.

2. Handling edge cases: Division algorithms need to handle various edge cases, such as dividing by zero, dividing zero by a non-zero number, or dealing with extremely large or small dividend and divisor values. Ensure your implementa on can handle these scenarios gracefully and return appropriate results or error messages.

3. Performance op miza on: The goal of fast division algorithms is to improve computa onal efficiency. Achieving this requires careful op miza on of the code. U lize efficient data structures, minimize unnecessary calcula ons, and consider algorithmic improvements like precomputa on or memoiza on when applicable. Benchmark and profile your code to iden fy poten al bo lenecks and op mize them.

4. Precision and accuracy: Division algorithms can introduce rounding errors or inaccuracies due to finite precision arithme c in computers. Ensure your implementa on maintains the desired level of precision and accuracy. Pay a en on to issues like floa ng-point arithme c, rounding modes, and error propaga on.

5. Tes ng and valida on: Test your division algorithm against a variety of test cases, including both

common scenarios and corner cases. Compare the results of your implementa on with trusted references or known correct implementa ons. This

process will help iden fy and rec fy any discrepancies or errors in your code.

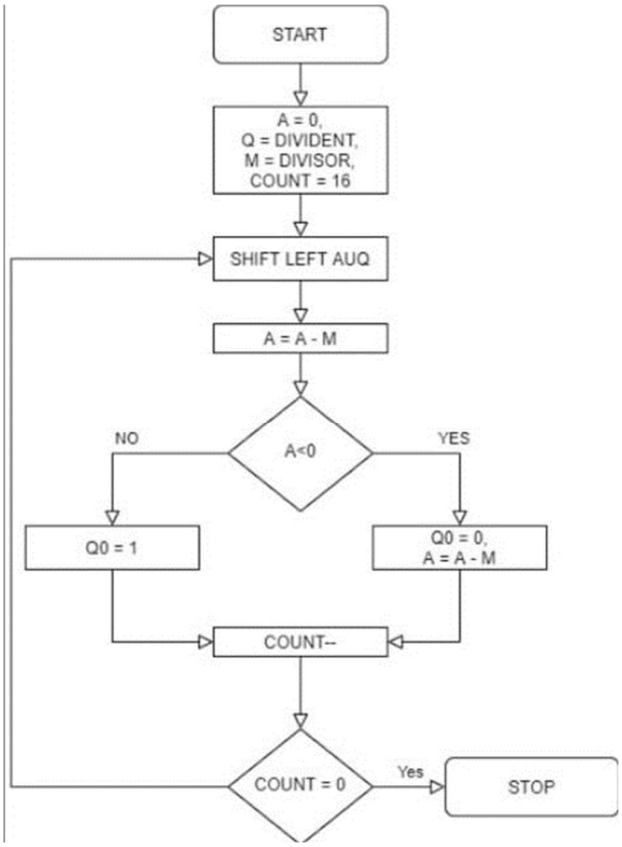
6. Handling nega ve numbers and remainders: Division algorithms should handle nega ve dividends and divisors correctly, adhering to the desired conven ons for quo ent signs and remainders. Consider the specific rules you need to follow and ensure your implementa on abides by them.

7. Code maintainability and readability: Division algorithms can be complex, involving mul ple steps and mathema cal opera ons. Focus on wri ng clean, modular, and well-

documented code to enhance maintainability and readability. Use meaningful variable names, logical func ons, and comments to make your code more understandable and easier to debug or modify in the future.

Remember, the specific challenges you encounter will depend on the algorithm you choose to implement. Research and familiarize yourself with the specific algorithm's intricacies and the poten al obstacles associated with it.

VI. ARCHITECTURE



Architecture flow of CNN

[The implementation of slow and fast division algorithms on a computer architecture involves utilizing the available hardware resources efficiently. While the exact details may vary depending on the specific algorithm and hardware architecture, here are some key aspects to consider:

1. rely on the arithmetic units of a computer architecture, primarily the integer and floating-point units. Understanding the capabilities and limitations of these units is essential for efficient implementation.

2. Instruction set: Familiarize yourself with the instruction set architecture (ISA) of the target hardware. Different ISAs offer various instructions and addressing modes that can affect the implementation approach. Utilize the available instructions effectively to optimize the division algorithm.

3. Register usage: Efficient utilization of registers is crucial for performance. Division algorithms often involve multiple intermediate values and require

temporary storage. Minimize memory accesses by utilizing registers effectively to store intermediate results, operands, and loop counters.

4. Pipelining and parallelism: Exploiting pipelining and parallelism can significantly enhance the performance of division algorithms. Break down the algorithm into stages and overlapping operations to take advantage of pipelined execution. Utilize multiple hardware execution units or instructionlevel parallelism (ILP) techniques to perform division-related calculations concurrently.

5. Data dependencies and hazards: Division algorithms can have data dependencies and hazards that impact performance. Analyze the dependencies between instructions and schedule them in a way that minimizes stalls and pipeline hazards. Techniques such as instruction reordering, loop unrolling, and software pipelining can be employed to mitigate these issues.

6. Memory hierarchy: Consider the memory hierarchy of the computer architecture, including caches and main memory. Optimize memory accesses by minimizing cache misses and utilizing prefetching techniques. Align data structures and memory accesses to maximize cache utilization.

7. Vectorization: If the hardware supports vector instructions (e.g., SIMD instructions), consider vectorizing the division algorithm. Vectorization allows simultaneous execution of multiple data elements, which can lead to significant speedup. Utilize vector instructions to perform parallel computations on multiple operands simultaneously.

8. Compiler optimizations: Modern compilers can automatically apply various optimizations to improve the performance of division algorithms. Enable optimization flags and explore compilerspecific directives to guide the optimization process. Analyze the generated assembly code to ensure the compiler is effectively utilizing the hardware resources.

9. Profiling and benchmarking: Profile the division algorithm implementation to identify performance bottlenecks. Use performance analysis tools to measure and understand the runtime behavior of the code. Benchmark the implementation against relevant test cases and compare the results to assess its efficiency.

Remember, the implementation details will vary based on the specific algorithm and hardware architecture. It is essential to understand the target architecture's features, limitations, and optimization techniques to develop an efficient implementation of slow and fast division algorithms.

VI. METHODOLOGY

Division can be wri en as the product of the dividend and the reciprocal of the divisor, or Q = a=b = a (1=b); (14) where Q is the quo ent, a is the dividend, and b is the divisor. In this case, the challenge becomes how to ciently compute the reciprocal of the divisor. In the Newton-Raphson algorithm, a priming func on is chosen which has a root at the reciprocal [10].

In general, there are many root targets that could be used, including 1 b , 1 b2 , a b , and 1 1 b . The choice of which root target to use is arbitrary. The selec on is made based on convenience of the itera ve form, its convergence rate, its lack of divisions, and the overhead involved when using a target root other than the true quo ent. The most widely used target root is the divisor reciprocal 1 b , which is the root of the priming func on: f (X) = 1=X b = 0: (15) To appear in IEEE Transac ons on Computers 1997 The well-known quadra cally converging Newton-Raphson equa on is given by: xi+1 = xi f (xi) f 0(xi) (16) The Newton-Raphson equa on of (16) is then applied to (15), and this itera on is then used to nd an approxima on to the reciprocal: Xi+1 = Xi f (Xi) f 0(Xi) = Xi + (1=Xi b) (1=X2 i) = Xi (2 b Xi) (17) The corresponding error term is given by i+1 = 2 i (b) and thus the error in the reciprocal decreases quadra cally a er each itera on. As can be seen from (17), each itera on involves two mul plica ons and a subtrac on. The subtrac on is equivalent to forming the two's complement and is commonly replaced by it.

Thus, two dependent mul plica ons and one two's complement opera on are performed each itera on. By using a more accurate star ng approxima on, the total number of itera ons required can be reduced. To achieve 53 bits of precision for the nal reciprocal star ng with only 1 bit, the algorithm will require 6 itera ons: 1 ! 2 ! 4 ! 8 ! 16 ! 32 ! 53 By using a more accurate

[1] SPEC benchmark suite release 2/92.

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

[3] S. F. Oberman and M star ng approxima on, for example 8 bits, the latency can be reduced to 3 itera ons. By using at least 14 bits, the latency could be further reduced to only 2 itera ons.

RESULT:

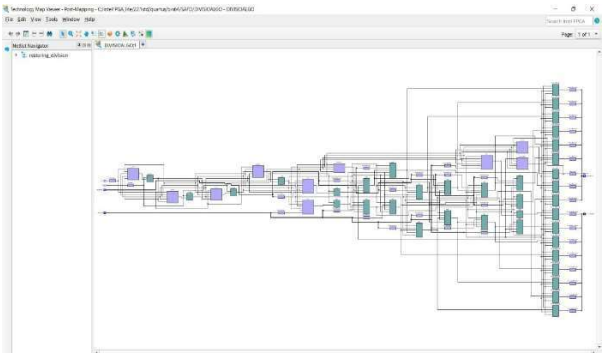


Fig:1

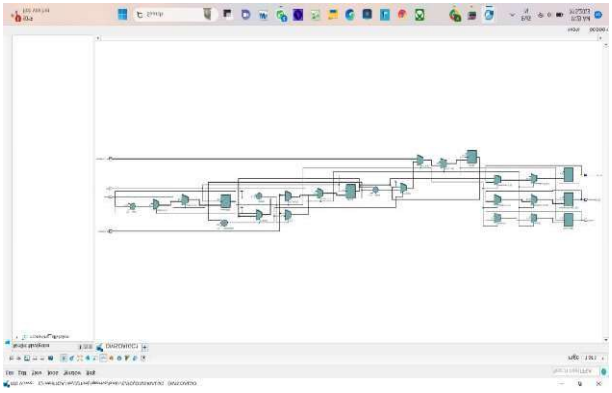


Fig:2

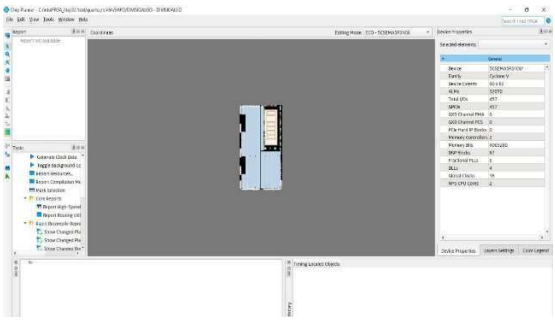


Fig 3

VII. CONCLUSION

In conclusion, the implementation of both slow and fast division algorithms in computer architecture offers tradeoffs in terms of speed and resource utilization.

The slow division algorithm, such as the non-restoring division or the restoring division, is a straightforward approach that performs division by repeated subtraction or trial-and-error quotient selection. While this algorithm is relatively simple to implement, it is slower compared to other methods. It requires a larger number of clock cycles and imposes a higher computational burden on the system. However, slow division algorithms may be useful in situations where hardware resources are limited, or when precision is not a critical factor.

On the other hand, fast division algorithms, such as the SRT division or the Newton-Raphson division, aim to reduce the division time by employing more complex techniques. These algorithms exploit mathematical properties to accelerate the division process. They typically involve iterative calculations and require additional hardware resources, such as multipliers or shifters. Fast division algorithms offer significant speed improvements over their slower counterparts but come at the cost of increased complexity and resource utilization.

The choice between slow and fast division algorithms depends on the specific requirements of the system. If computational speed is a top priority, and hardware resources are sufficient, a fast division algorithm is preferable. However, if simplicity and resource conservation are more important, a slow division algorithm may be a suitable choice.

Ultimately, the selection of the division algorithm in computer architecture should be based on a careful evaluation of the system's performance requirements, available hardware resources, and desired trade-offs between speed and complexity.

VIII. 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 oa ng-point opera ons," To appear in IEEE Trans. Computers, 1997.

[4] C. V. Freiman, \Sta s cal 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 mul plica on and division for automa c binary computers," Quart. J. Mech. Appl. Math., vol. 11, pt. 3, pp. 364{384, 1958.

[7] D. E. Atkins, \Higher-radix division using es mates of the divisor and par al remainders," IEEE Trans. Computers, vol. C-17, no. 10, Oct. 1968.

[8] K. G. Tan, \The theory and implementa on of highradix division," in Proc. 4th IEEE Symp. Computer Arithme c, pp. 154{163, June 1978.

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

[10] M. Flynn, \On division by func onal itera on," IEEE Trans. Computers, vol. C-19, no. 8, Aug. 1970.