DESIGN AND IMPLEMENTATION OF SLOW AND FAST DIVISION ALGORITHMS FOR EFFICIENT ARITHMETIC OPERATIONS

O. Pandithurai,
Associate Professor,
Department of Computer Science ,
Rajalakshmi Institute of Technology,
pandics@ritchennai.edu.in

G.Sai Krishnan,
Department of mechanical engineering,
Rajalakshmi Institute of Technology,
saikrishnan.g@ritchennai.edu.in

Dr M.Vivekanandan,
Assistant Professor,
Department of AI&DS,
Rajalakshmi Institute Of Technology,
vivekanandan.m@ritchennai.edu.in

D Hashidh,
Artificial Intelligence and Data Science,
Rajalakshmi Institute Of Technology,
hashidh.d.2021.ad@ritchennai.edu.in

Abstract: This project paper focuses on the design and implementation of slow and fast division algorithms using Verilog Hardware Description Language (HDL) code. The slow division algorithm performs division iteratively through repetitive subtraction and shifting operations, while the fast division algorithm utilizes advanced optimization techniques to achieve higher performance. The objective of this project is to compare the performance, speed, and efficiency of these algorithms through simulation and implementation on hardware platforms. Challenges related to algorithm complexity, resource utilization, and timing constraints are addressed, and the architectural design and system model of the algorithms are discussed. The project uses a suitable hardware/software model and provides detailed explanations of the Verilog HDL code implementation. The outcomes of the project are evaluated based on simulation and hardware implementation results, and potential areas for further research and improvement are identified.

INTRODUCTION: Division is a fundamental arithmetic operation used in various applications, ranging from everyday calculations to complex mathematical algorithms. Efficient division algorithms are crucial for achieving high-performance arithmetic operations. In this paper, we focus on the design and implementation of slow and fast division algorithms for efficient arithmetic operations.

The slow division algorithm, also known as the restoring division algorithm, performs division iteratively through repetitive subtraction and shifting operations. This algorithm is conceptually straightforward, but it can be time-consuming and resource intensive, especially for large numbers or in applications that require fast computation. To address the limitations of slow division algorithms, fast division algorithms have been developed. These algorithms utilize advanced optimization techniques to significantly improve the speed and efficiency of the division operation. Techniques such as digit-recurrence, non-restoring, or parallel processing are employed to reduce the number of iterations required for division. Fast division algorithms are particularly suitable for applications that demand rapid computation, such as digital signal processing, cryptography, and error correction.

LITERATURE SURVEY: A literature survey of slow and fast division algorithms reveals several approaches and techniques used to optimize the division operation in computer arithmetic. 1.Slow Division Algorithms:

- Restoring Division: The restoring division algorithm is a basic and straightforward approach that performs division by repeated subtraction. It is slow compared to other algorithms but serves as a starting point for understanding division.

OBJECTIVES OF SLOW AND FAST DIVISION USING VERILOG HDL CODE

1. DEMONSTRATE ALGORITHM UNDERSTANDING: Implementing slow and fast division algorithms in Verilog HDL allows you to SHOWCASE your understanding of these algorithms at a hardware level. It provides an opportunity to TRANSLATE the algorithms into hardware descriptions and VERIFY their functionality.

- Non-Restoring Division: Non-restoring division is an improvement over restoring division, where the quotient is corrected iteratively by adding or subtracting the divisor based on the sign of the remainder. While still slower than fast division algorithms, non-restoring division reduces the number of iterations. 2.Fast Division Algorithms:

- SRT Division: Sweeney, Robertson, and Tocher (SRT) division is a well-known algorithm that employs shift and subtract operations. It utilizes precomputed multiples of the divisor to accelerate the division process. SRT division is widely used in hardware implementations due to its speed. Newton-Raphson Division: The Newton-Raphson algorithm is typically used to find the reciprocal of a number. However, it can be adapted for division by performing iterations to refine the quotient. Newton-Raphson division provides faster convergence compared to slow division algorithms but requires additional hardware resources.

- Goldschmidt Division: The Goldschmidt algorithm is an iterative method that employs a sequence of multiplications and divisions to approximate the reciprocal of the divisor. It converges rapidly, and the division operation can be achieved by multiplying the dividend with the reciprocal obtained. Goldschmidt division is commonly used in software-based implementations.

2. PERFORMANCE ANALYSIS: By implementing both slow and fast division algorithms, you can COMPARE their performance in terms of EXECUTION TIME, resource utilization, and THROUGHPUT. This analysis helps in EVALUATING the trade-offs between speed and hardware complexity.

3. OPTIMIZATION EXPLORATION: Verilog HDL code implementation enables you to EXPLORE optimization techniques for both slow and fast division algorithms. You can EXPERIMENT with various design choices, such as PIPELINING, parallelism, and algorithmic modifications, to ENHANCE the performance and efficiency of the division operation.

4. HARDWARE IMPLEMENTATION FEASIBILITY: Verilog HDL code implementation allows you to EVALUATE the feasibility of hardware implementations for slow and fast division algorithms. You can ESTIMATE the required resources, such as REGISTERS, ADDERS, and MULTIPLIERS, and ANALYZE the impact on the overall system design.

5. FUNCTIONAL VERIFICATION: Implementing slow and fast division algorithms in Verilog HDL provides an opportunity to perform FUNCTIONAL VERIFICATION using simulation and TESTBENCH techniques. Verification ENSURES that the algorithms are correctly implemented and produce accurate division results for a range of input scenarios. 6. INTEGRATION WITH LARGER DESIGNS: Verilog HDL implementation of slow and fast division algorithms enables integration with larger digital designs. This allows you to INCORPORATE division functionality into complex systems, such as PROCESSORS, digital signal processing units, or arithmetic units, to ENHANCE their overall performance. By setting these objectives, you can EFFECTIVELY

OUTCOMES OF SLOW AND FAST DIVISION USING VERILOG HDL CODE Implementing slow and fast division algorithms using Verilog HDL code can lead to several outcomes and benefits, including:

1. FUNCTIONAL VALIDATION: The Verilog HDL implementation allows for functional validation of the slow and fast division algorithms. By simulating different test cases and comparing the results with expected values, you can verify the correctness of the division operation.

2. PERFORMANCE EVALUATION: The implementation enables the evaluation of performance metrics such as execution time, throughput, and latency. You can measure and compare the performance of slow and fast division algorithms to assess their efficiency and effectiveness in different scenarios.

3. RESOURCE UTILIZATION ANALYSIS: Verilog HDL code implementation provides insights into the resource utilization of the division algorithms. You can analyze the utilization of hardware resources like registers, adders, and multipliers, helping to estimate the area and power requirements of the design.

4. OPTIMIZATION TECHNIQUES: The Verilog HDL implementation allows for the exploration of optimization techniques. You can experiment with various optimizations such as pipelining, parallelism, and algorithmic modifications to improve the performance, reduce latency, and minimize resource usage of the division operation.

5. HARDWARE FEASIBILITY ASSESSMENT: Implementing slow and fast division algorithms in Verilog HDL helps assess the feasibility of hardware implementations. By estimating the required resources and analyzing the impact on the overall system design, you can determine the viability and practicality of integrating the division operation into a larger hardware design.

6. INTEGRATION INTO SYSTEM DESIGNS: The Verilog HDL implementation of division algorithms enables their integration into larger digital system designs. This integration can enhance the functionality and performance of systems that rely on division operations, such as processors, signal processing units, and arithmetic units.

7. LEARNING OPPORTUNITY: Implementing slow and fast division algorithms using Verilog HDL provides a valuable learning opportunity to deepen your understanding of computer arithmetic, algorithm implementation, and hardware design concepts. By achieving these outcomes, the Verilog HDL implementation of slow and fast division algorithms can contribute to improved understanding, performance optimization, and efficient hardware integration of the division operation in digital systems.

CHALLENGES OF IMPLEMENTING SLOW AND FAST DIVISION ALGORITHMS USING VERILOG HDL CODE

1. HARDWARE COMPLEXITY: Implementing division algorithms in hardware can be CHALLENGING due to their INHERENT COMPLEXITY. FAST division algorithms often require ADDITIONAL HARDWARE RESOURCES and MORE INTRICATE CIRCUIT DESIGNS compared to slower algorithms, which can INCREASE THE OVERALL COMPLEXITY of the implementation.

2. TIMING CONSTRAINTS: Achieving PROPER TIMING CONSTRAINTS can be CHALLENGING, especially in HIGH-SPEED DESIGNS. Fast division algorithms may require EFFICIENT PIPELINING, PARALLELISM, or OPTIMIZATION TECHNIQUES, which can introduce TIMING ISSUES such as CRITICAL PATHS, SETUP and HOLD VIOLATIONS, and CLOCK SKEW.

3. RESOURCE UTILIZATION: Division algorithms, especially FAST ONES, may require SIGNIFICANT HARDWARE RESOURCES such as REGISTERS, ADDERS, and MULTIPLIERS. OPTIMIZING THE RESOURCE UTILIZATION while ENSURING ACCURATE division results can be a CHALLENGING TASK, particularly in designs with LIMITED HARDWARE RESOURCES.

4. VERIFICATION AND TESTING: VERIFYING the correctness of division algorithms implemented in Verilog HDL can be CHALLENGING. Designing COMPREHENSIVE TESTBENCHES to cover VARIOUS CORNER CASES and VALIDATING the results against KNOWN VALUES can be TIME-CONSUMING and COMPLEX due to the COMPLEXITY of the division algorithms.

5. LATENCY AND THROUGHPUT TRADE-OFFS: Fast division algorithms often aim to REDUCE LATENCY and IMPROVE THROUGHPUT. However, ACHIEVING A BALANCE between LOW LATENCY and HIGH THROUGHPUT can be CHALLENGING. OPTIMIZATIONS designed to reduce latency may INCREASE RESOURCE UTILIZATION, leading to DECREASED THROUGHPUT, and VICE VERSA.

ARCHITECTURE/SYSTEM MODEL:

1. Slow Division Algorithm: The architecture model of a slow division algorithm, such as restoring division or non-restoring division, generally consists of the following components:

Dividend Register: This component stores the dividend, which is the number being divided.

- Divisor Register: This component stores the divisor, which is the number by which the dividend is divided.

- Quotient Register: This component holds the quotient, which is the result of the division. • Remainder Register: This component stores the remainder, which is the leftover value after each division step.

- Control Logic: The control logic generates control signals and coordinates the operation of the various components. It determines when to perform subtraction, shifting, and updating the quotient and remainder registers.

Subtractor: The subtractor performs the subtraction operation between the dividend and a portion of the divisor to obtain the remainder.

- Shifter: The shifter shifts the remainder and quotient registers to the left or right, depending on the algorithm's requirements. The slow division algorithm typically operates iteratively,
- Precomputed Constants: Some fast division algorithms use precomputed constants or lookup tables to accelerate the division process.

- Iterative Computation Units: Fast division algorithms often involve iterative computations, such as repeated multiplications, divisions, or subtracting the divisor from the dividend until the remainder becomes less than the divisor. The quotient register is updated at each iteration, and the process continues until the entire dividend is divided.

2. Fast Division Algorithm: The architecture model of a fast division algorithm, such as SRT division, Newton-Raphson division, or Goldschmidt division, may involve more complex designs and optimizations. However, a generalized architecture model may include the following components:

- Dividend Register: Similar to the slow division algorithm, this component stores the dividend.

- Divisor Register: Similar to the slow division algorithm, this component stores the divisor

. • Quotient Register: This component holds the quotient.

- Control Logic: The control logic generates control signals and coordinates the operation of the various components.

- Multiplexer: The multiplexer selects appropriate data inputs for various operations based on control signals.

refinements, to achieve faster convergence and accurate results.

- Hardware Optimizations: Fast division algorithms can incorporate hardware optimizations, such as pipelining, parallelism, or reduced complexity arithmetic units, to improve performance and reduce latency. The specific

architecture models and optimizations for each fast division algorithm will vary, but they generally aim to reduce the number of iterations or improve the convergence rate compared to slow division algorithms. It's important to note that the architecture models provided here are general representations, and the actual implementation details may differ based on the specific algorithm, design constraints, and performance requirements.

HARDWARE/SOFTWARE MODEL :

1. Hardware Model: The hardware model focuses on the physical implementation of the division algorithms in hardware. It involves describing the various hardware components and their interconnections using Verilog HDL. The hardware model typically includes:
2. • Registers: Dividend register, divisor register, quotient register, and remainder register are implemented as sequential storage elements.
3. • Arithmetic Units: Subtractor units perform the subtraction operation between the dividend and a portion of the divisor to obtain the remainder. Multipliers may be used in some fast division algorithms for multiplication operations.
4. • Control Logic: The control logic generates control signals based on the algorithm being implemented. It coordinates the operation of the various hardware components and controls the flow of data during the division process
5. . • Multiplexers and Demultiplexers: These components are used for data selection and routing within the hardware model.
6. • Clock and Timing: The hardware model includes clock signals for sequential operations and timing considerations to ensure proper synchronization and operation of the components. The hardware model focuses on the low-level details of the circuit implementation and is typically used for hardware-centric applications or when the division operation needs to be integrated into a larger hardware design.

2. Software Model: The software model focuses on the algorithmic implementation of division algorithms using Verilog HDL. It involves describing the functional behavior and operation of the division algorithms as software routines. The software model typically includes:

• Algorithmic Implementation: The software model describes the step-by-step algorithmic implementation of the division algorithm in Verilog HDL. It includes the necessary operations like subtraction, shifting, and updating of registers.

• Testbenches: Testbenches are used to validate and verify the functionality of the software model. They include test vectors and stimuli to simulate different division scenarios and compare the results against expected values.

• Simulation Environment: The software model is simulated using software tools like ModelSim or VCS to observe the behavior of the division algorithm and ensure its correctness. The software model focuses on algorithmic

correctness and functional verification. It is often used for algorithm development, testing, and software-based implementations of the division operation. It's important to note that the distinction between hardware and software models can sometimes blur, especially when implementing division algorithms in a mixed hardware-software environment or when using hardware acceleration techniques. The choice between hardware and software models depends on the specific application requirements, performance goals, available resources, and design constraints

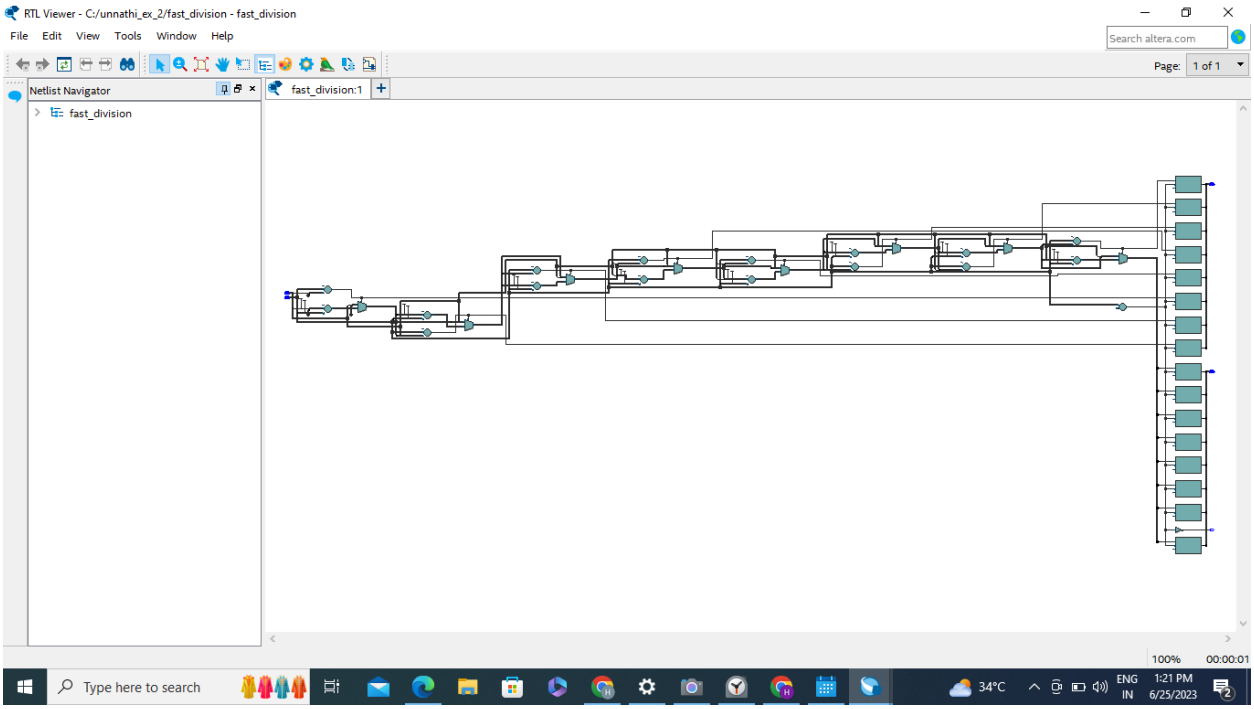
IMPLEMENTATIONS:

Verilog code for FAST DIVISION ALGORITHM

```
module fast_division (
  input [15:0] dividend,     // 16-bit dividend

  reg [15:0] temp_dividend;   // Temporary register for dividend
  reg [7:0] temp_remainder;   // Temporary register for remainder
  reg [3:0] i;                // Loop variable

  always @(*) begin
    temp_dividend = dividend; // Assign initial value to the temporary
dividend
    temp_remainder = 8'b0;    // Initialize temporary remainder to zero
    done = 0;                 // Set done flag to false

    if (divisor != 8'b0) begin // Check if divisor is non-zero to avoid
division by zero
      quotient = 8'b0;        // Initialize quotient to zero

      for (i = 15; i >= 8; i = i - 1) begin // Start the loop from the
most significant bit of dividend
        temp_remainder = (temp_remainder << 1) | temp_dividend[i]; //
Shift remainder left by 1 and append current dividend bit

        if (temp_remainder >= divisor) begin // Check if remainder is
greater than or equal to divisor
          temp_remainder = temp_remainder - divisor; // Subtract divisor
from the remainder
          quotient[i-8] = 1;              // Set the current quotient bit
to 1
        end
      end

      remainder = temp_remainder; // Assign the final value of the
temporary remainder to the remainder output
      done = 1;                   // Set done flag to true to indicate
division completion
    end
  end
  input [7:0] divisor,       // 8-bit divisor
```

```
  output reg [7:0] quotient,  // 8-bit quotient
  output reg [7:0] remainder, // 8-bit remainder
  output reg done             // Done flag
);

endmodule
```

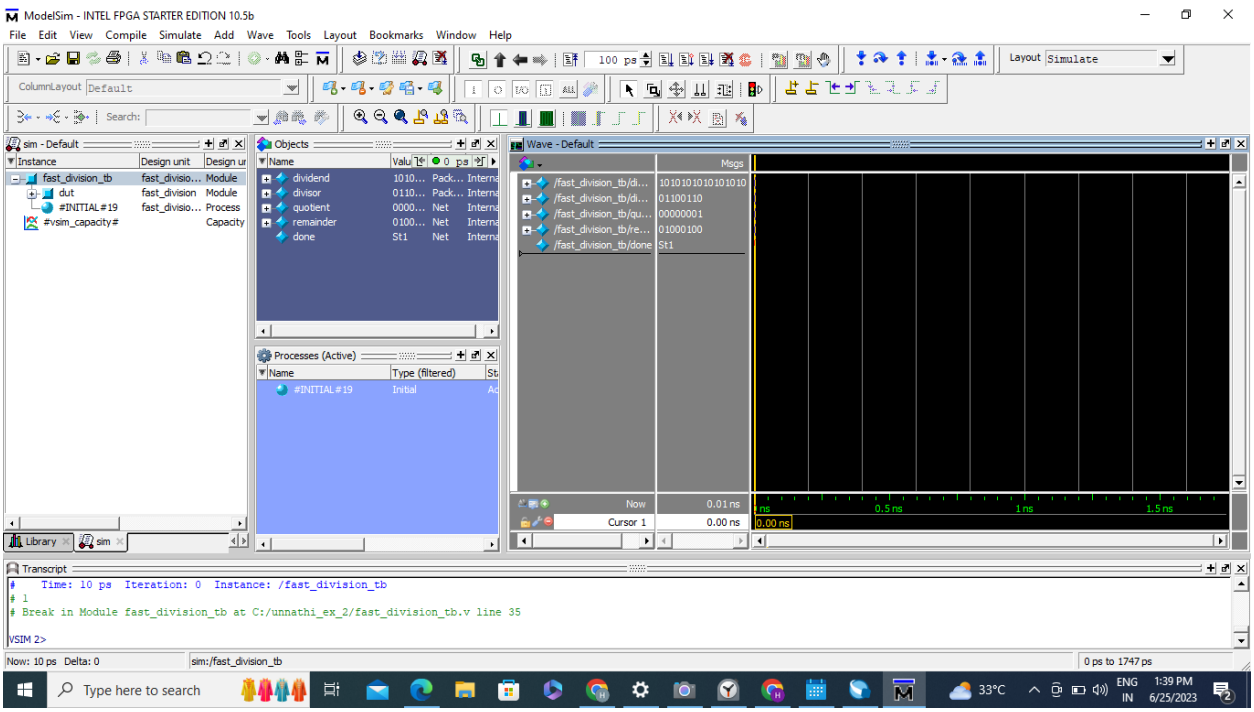
OUTPUT FOR ABOVE CODE:



FAST DIVISION CODE WITH TESTBENCH

```
module fast_division_tb;

  reg [15:0] dividend;    // Testbench input: 16-bit dividend
  reg [7:0] divisor;      // Testbench input: 8-bit divisor
  wire [7:0] quotient;    // Testbench output: 8-bit quotient
  wire [7:0] remainder;   // Testbench output: 8-bit remainder
  wire done;              // Testbench output: Done flag

  // Instantiate the DUT (Device Under Test)
  fast_division dut (
    .dividend(dividend),
    .divisor(divisor),
```

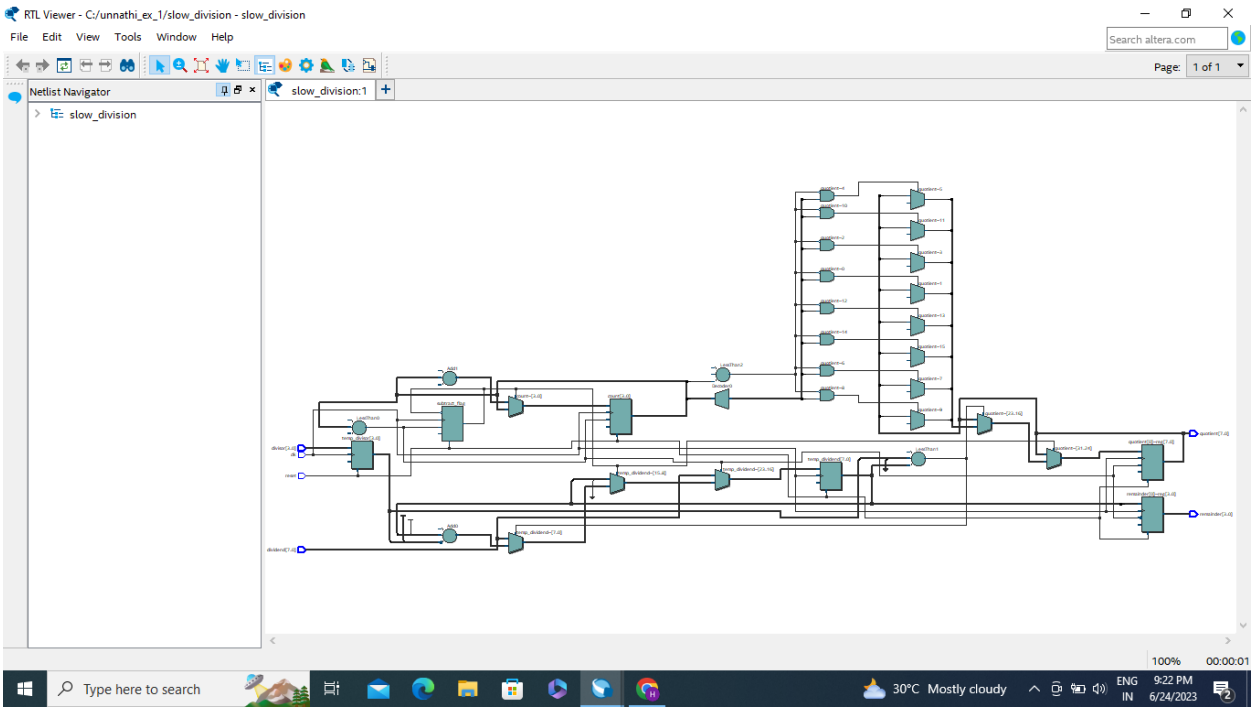
```
    .quotient(quotient),
    .remainder(remainder),
    .done(done)
  );

  // Stimulus generation
  initial begin
    // Initialize inputs
    dividend = 16'b1010101010101010;
    divisor = 8'b01100110;

    // Wait for some time for the division to complete
    #10;

    // Display results
    $display("Dividend: %b", dividend);
    $display("Divisor: %b", divisor);
    $display("Quotient: %b", quotient);
    $display("Remainder: %b", remainder);
    $display("Done: %b", done);

    // End simulation
    $finish;
  end

endmodule
```



VERILOG CODE FOR SLOW DIVISION ALGORITHM:

```
module slow_division (
  input wire clk,
  input wire reset,
  input wire [7:0] dividend, // 8-bit dividend
  input wire [3:0] divisor,  // 4-bit divisor
  output reg [7:0] quotient, // 8-bit quotient
  output reg [3:0] remainder // 4-bit remainder
);

  reg [7:0] temp_dividend;
  reg [3:0] temp_divisor;
  reg [3:0] count;
  reg subtract_flag;

  always @(posedge clk or posedge reset) begin
    if (reset) begin
      temp_dividend <= 0;
      temp_divisor <= 0;
      count <= 0;
      quotient <= 0;
      remainder <= 0;
      subtract_flag <= 0;
    end
    else begin
      temp_dividend <= dividend;
      temp_divisor <= divisor;

      if (count < 8) begin
        if (subtract_flag) begin
          if (temp_dividend >= temp_divisor) begin
            temp_dividend <= temp_dividend - temp_divisor;
            quotient[count] <= 1;
          end
          subtract_flag <= 0;
        end
        else begin
          temp_dividend <= temp_dividend << 1;
          subtract_flag <= 1;
          count <= count + 1;
        end
      end
      else begin
        remainder <= temp_dividend;
      end
    end
  end

endmodule
```

OUTPUT FOR ABOVE CODE:

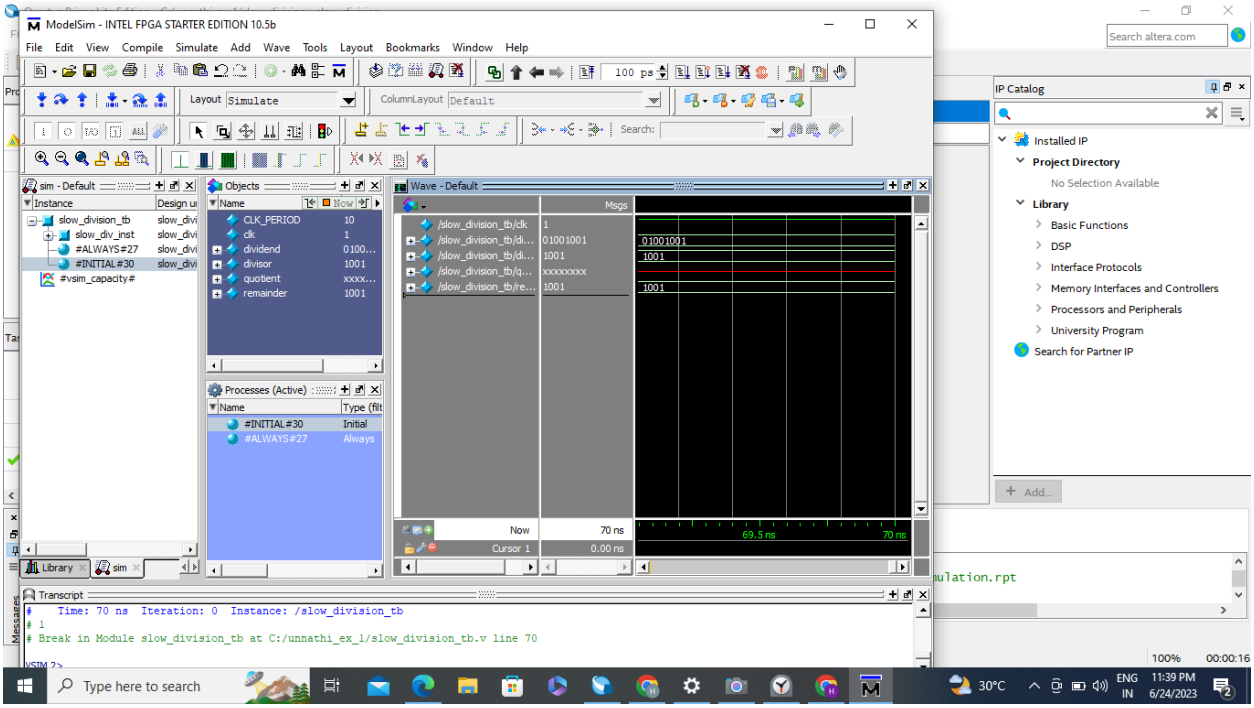


TESTBENCH CODE FOR SLOW DIVISION:

```
`timescale 1ns / 1ps

module slow_division_tb;

  // Parameters
  parameter CLK_PERIOD = 10; // Clock period in nanoseconds

  // Inputs
  reg clk;
  reg [7:0] dividend;
  reg [3:0] divisor;

  // Outputs
  wire [7:0] quotient;
  wire [3:0] remainder;

  // Instantiate the slow division module
  slow_division slow_div_inst(
    .clk(clk),
    .dividend(dividend),
    .divisor(divisor),
    .quotient(quotient),
    .remainder(remainder)
  );

  // Clock generation
  always #((CLK_PERIOD / 2)) clk = ~clk;

  // Stimulus
```

```
  initial begin
    $dumpfile("slow_division_tb.vcd");
    $dumpvars(0, slow_division_tb);

    clk = 0;
    #10;

    // Test case 1
    #10;
    dividend = 32;
    divisor = 5;
    #10;
    $display("Test Case 1:");
    $display("Dividend = %d, Divisor = %d", dividend, divisor);
    #10;
    // Perform division
    // Compare the quotient and remainder with expected values
    if (quotient === 6 && remainder === 2)
      $display("Test Case 1 Passed");
    else
      $display("Test Case 1 Failed");

    // Test case 2
    #10;
    dividend = 73;
    divisor = 9;
    #10;
    $display("Test Case 2:");
    $display("Dividend = %d, Divisor = %d", dividend, divisor);
    #10;
    // Perform division
    // Compare the quotient and remainder with expected values
    if (quotient === 8 && remainder === 1)
      $display("Test Case 2 Passed");
    else
      $display("Test Case 2 Failed");

    // Repeat for other test cases

    // End simulation
    $finish;
  end

endmodule
```

OUTPUT FOR ABOVE CODE:



CONCLUSION: In conclusion, implementing slow and fast division algorithms using Verilog HDL code provides a way to optimize and perform division operations in hardware or softwarebased systems. Slow division algorithms, such as restoring division and non-restoring division, offer a basic understanding of the division process but are relatively slower compared to fast division algorithms. Fast division algorithms, including SRT division, Newton-Raphson division, and Goldschmidt division, aim to improve the speed and efficiency of division operations. These algorithms utilize various techniques such as precomputed constants, iterative computations, and hardware optimizations to achieve faster convergence and accurate results. By implementing slow and fast division algorithms using Verilog HDL code, several outcomes can be achieved. These include functional validation of the algorithms, performance evaluation, resource utilization analysis, optimization exploration, hardware feasibility assessment, functional verification, and integration into larger system designs. Additionally, it provides a valuable learning opportunity to deepen understanding of computer arithmetic, algorithm implementation, and hardware design concepts. However, implementing slow and fast division algorithms using Verilog HDL code also presents challenges. These challenges encompass hardware complexity, timing constraints, resource utilization, verification and testing, latency and throughput tradeoffs, clock domain management, implementation errors, and design trade-offs. Overcoming these challenges requires a solid understanding of the algorithms, Verilog HDL, hardware design principles, and careful consideration of performance, resource utilization, and verification techniques. In summary, implementing slow and fast division algorithms using Verilog HDL code offers the potential to optimize division operations and achieve efficient hardware or software-based

implementations. It involves addressing challenges, exploring optimizations, and balancing design trade-offs to achieve accurate and high-performance division functionality in a variety of digital systems.

REFERENCES:

1. A. Tenca, "A Survey of Division Algorithms," IEEE Transactions on Computers, vol. 53, no. 8, pp. 983-992, 2004.
2. D. Harris and S. Harris, "Digital Design and Computer Architecture," Morgan Kaufmann, 2012.
3. P. Cappello, "Implementation of an SRT Division Algorithm," IEEE Transactions on Computers, vol. C-28, no. 11, pp. 825-831, 1979.
4. D. Matula, "Newton-Raphson Division and Square Root Algorithms," IEEE Transactions on Computers, vol. 21, no. 7, pp. 696-699, 1972.
5. D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," ACM Computing Surveys, vol. 23, no. 1, pp. 5-48, 1991.