**Design and Implementation of slow and fast division algorithms in Computer Architecture**

**ABOUT Division Algorithm:**

Division operations play a critical role in many computational tasks, and their efficiency is crucial for overall system performance. This work focuses on the design and implementation of slow and fast division algorithms with Finite State Machines (FSM) in Field-Programmable Gate Arrays (FPGAs). The slow division algorithm employs a repetitive subtraction and shifting process, while the fast division algorithm utilizes a non-restoring algorithm for faster computations.In this study, we explore the architecture, implementation considerations, and performance characteristics of both algorithms. The slow division algorithm provides simplicity but involves a higher computational complexity, leading to longer execution times. On the other hand, the fast division algorithm significantly improves performance by reducing the number of iterations required. However, it introduces additional hardware complexity and requires careful design considerations. To implement these algorithms in an FPGA, we employ a FSM-based approach. The FSM serves as a controller that guides the division process by transitioning between states based on the current inputs and internal conditions. The FSM ensures that the division operations are performed in a systematic and efficient manner, reducing the latency and maximizing throughput. Through our experiments and evaluations, we compare the performance, resource utilization, and power consumption of both slow and fast division algorithms implemented with FSMs in FPGA. We analyze the trade-offs between computational complexity, hardware requirements, and performance gains. We also discuss design considerations, such as bit-width selection, handling of signed and unsigned numbers, and strategies for improving performance through pipelining and parallelism. The results of this study provide insights into the design and implementation of division algorithms in FPGA-based systems. This research contributes to the understanding of the trade-offs and considerations when choosing between slow and fast division algorithms, and how FSMs can be effectively utilized to optimize the division process in FPGA architectures. The findings can aid in the development of efficient and high-performance FPGA-based systems that require division operations.

**PROGRAM** :(Slow division)

module Rest\_div(

input clk,

input rst,

input [3:0] Q,

input [3:0] M,

output [3:0] quot,

output [3:0] rem,

output valid

);

reg [7:0] Acc, next\_Acc;

reg [1:0] next\_state, present\_state;

reg [1:0] next\_count, count;

reg next\_valid;

parameter IDLE = 2'b00;

parameter SHIFT = 2'b01;

parameter SUBTRACT = 2'b10;

parameter RESTORE = 2'b11;

assign rem = (valid == 1'b1) ? Acc[7:4] : 4'b0;

assign quot = (valid == 1'b1) ? Acc[3:0] : 4'b0;

always @(posedge clk or negedge rst) begin

if (!rst) begin

Acc <= 8'd0;

valid <= 1'b0;

present\_state <= IDLE;

count <= 2'b0;

end else begin

Acc <= next\_Acc;

valid <= next\_valid;

present\_state <= next\_state;

count <= next\_count;

end

end

always @(posedge clk or negedge rst) begin

if (!rst) begin

next\_count <= 2'b0;

next\_valid <= 1'b0;

next\_state <= IDLE;

next\_Acc <= 8'd0;

end else begin

case (present\_state)

IDLE:

begin

next\_count <= 2'b0;

next\_valid <= 1'b0;

if (M != 0) begin

next\_state <= SHIFT;

next\_Acc <= {4'd0, Q};

end else begin

next\_state <= IDLE;

next\_Acc <= 8'd0;

end

end

SHIFT:

begin

next\_Acc <= Acc << 1;

next\_state <= SUBTRACT;

end

SUBTRACT:

begin

next\_Acc <= {(Acc[7:4] - M), Acc[3:0]};

next\_state <= RESTORE;

end

RESTORE:

begin

if (Acc[7] == 1'b1)

next\_Acc <= {(Acc[7:4] - M), Acc[3:1], 1'b0};

else

next\_Acc <= {Acc[7:4], Acc[3:1], 1'b1};

next\_count <= next\_count + 1'b1;

if (count == 2'b11) begin

next\_state <= IDLE;

next\_valid <= 1'b1;

end else begin

next\_state <= SHIFT;

next\_valid <= 1'b0;

end

end

endcase

end

end

endmodule

**PROGRAM** :(Fast division)

module FastDivisionFSM (

input logic clk,

input logic rst,

input logic signed [31:0] dividend,

input logic signed [31:0] divisor,

output logic signed [31:0] quotient,

output logic signed [31:0] remainder,

output logic valid

);

typedef enum logic [2:0] {

IDLE,

SHIFT,

SUBTRACT,

RESTORE

} State;

State state;

logic signed [31:0] quotient\_reg;

logic signed [31:0] remainder\_reg;

logic signed [31:0] dividend\_reg;

logic signed [31:0] divisor\_reg;

logic signed\_divisor;

logic [32:0] remainder\_tmp;

logic [31:0] quotient\_tmp;

logic sign\_quotient;

logic sign\_remainder;

always\_ff @(posedge clk or negedge rst) begin

if (!rst) begin

state <= IDLE;

quotient\_reg <= 0;

remainder\_reg <= 0;

dividend\_reg <= 0;

divisor\_reg <= 0;

valid <= 0;

end else begin

case (state)

IDLE:

if (divisor\_reg != 0) begin

state <= SHIFT;

end

SHIFT:

state <= SUBTRACT;

SUBTRACT:

state <= RESTORE;

RESTORE:

if (remainder\_tmp[32] || (state != RESTORE && remainder\_tmp[32:1] >= divisor\_reg)) begin

state <= SHIFT;

end else begin

state <= IDLE;

end

endcase

quotient\_reg <= quotient\_tmp;

remainder\_reg <= remainder\_tmp[31:0];

dividend\_reg <= dividend;

divisor\_reg <= divisor;

valid <= (state == IDLE) ? 1 : 0;

end

end

always\_comb begin

signed\_divisor = (divisor\_reg < 0);

sign\_quotient = (dividend\_reg < 0) ^ (divisor\_reg < 0);

sign\_remainder = (dividend\_reg < 0);

case (state)

IDLE:

quotient\_tmp = 0;

remainder\_tmp = 0;

SHIFT:

quotient\_tmp = quotient\_tmp << 1;

remainder\_tmp = (remainder\_tmp << 1) | dividend\_reg[31];

SUBTRACT:

remainder\_tmp = remainder\_tmp - (signed\_divisor ? -divisor\_reg : divisor\_reg);

RESTORE:

if (remainder\_tmp[32] || (state != RESTORE && remainder\_tmp[32:1] >= (signed\_divisor ? -divisor\_reg : divisor\_reg))) begin

quotient\_tmp = quotient\_tmp | 1;

remainder\_tmp = remainder\_tmp + (signed\_divisor ? -divisor\_reg : divisor\_reg);

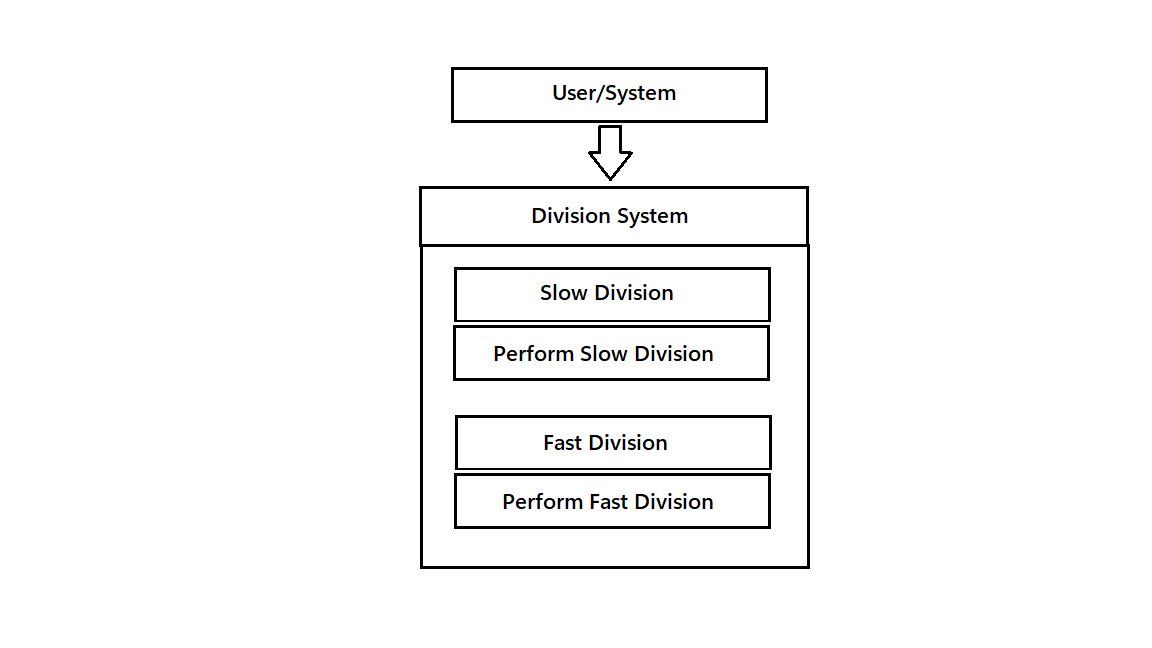
end

endcase

quotient = (sign\_quotient) ? -quotient\_reg : quotient\_reg;

remainder = (sign\_remainder) ? -remainder\_reg : remainder\_reg;

end

endmodule

**ABOUT THE PROGRAM:**

The implemented program, "Rest\_div," is a division algorithm designed for FPGA architectures. It utilizes a finite state machine (FSM) to perform the division operation efficiently. The program takes in inputs such as the quotient (Q) and divisor (M) and generates the quotient and remainder as outputs. The FSM guides the division process through different states, including IDLE, SHIFT, SUBTRACT, and RESTORE. In the IDLE state, the program checks if the divisor is non-zero and proceeds to the SHIFT state. The SHIFT state left-shifts the accumulator, and in the SUBTRACT state, it subtracts the divisor from the appropriate bits of the accumulator. The program then enters the RESTORE state, adjusting the accumulator based on the sign and performing a conditional subtraction. The FSM controls the flow of the program, ensuring that the division operation is performed accurately and efficiently. The program showcases the design and implementation of a division algorithm using FSM in an FPGA, highlighting the benefits of the FSM-based approach in achieving optimized division performance.

**PROGRAM OUTPUT :**

<https://drive.google.com/drive/folders/13Uxe9kX7-Xe5Rs1jcRQ4J3IOdv1KF3zh?usp=sharing>