**Problem Statement**

Design and implementation of slow and fast division algorithm in computer architecture



### TRANING PROJECT REPORT

### By

### LUXMAN(210280464)

### RAJAN(210280494)

### RIYA(210280499)

**Introduction:-**

Restoring division algorithms are used in computer architecture to perform division operations. There are two common variations of restoring division algorithms: slow restoring and fast restoring. Both algorithms aim to divide a dividend by a divisor and produce the quotient and remainder.

Slow Restoring Division Algorithm:

The slow restoring division algorithm is a simple and straightforward method for division. It operates bit by bit, examining the dividend and divisor to determine the quotient and remainder. Here's a high-level overview of the algorithm:

Step 1: Initialize the quotient and remainder to zero.

Step 2: Compare the dividend (the number being divided) with the divisor (the number dividing the dividend).

Step 3: If the dividend is greater than or equal to the divisor, subtract the divisor from the dividend and set the least significant bit of the quotient to 1.

Step 4: If the dividend is less than the divisor, set the least significant bit of the quotient to 0.

Step 5: Shift the quotient and remainder one bit to the left.

Step 6: Shift the dividend one bit to the left and bring down the next bit.

Step 7: Repeat steps 2-6 until all bits of the dividend have been processed.

Step 8: The quotient obtained is the final result.

The slow restoring division algorithm is called "slow" because it performs multiple iterations to process each bit of the dividend. It can take a longer time to compute the division compared to other division algorithms.

Fast Restoring Division Algorithm:

The fast restoring division algorithm is an optimized version of the slow restoring algorithm. It reduces the number of iterations required to perform the division, resulting in faster computation. The key difference between the fast and slow restoring algorithms lies in steps 3 and 4.

In the fast restoring algorithm, after subtracting the divisor from the dividend in step 3, instead of immediately shifting the dividend and proceeding to the next bit, an additional adjustment is made. This adjustment involves adding the divisor back to the intermediate result obtained in step 3. The purpose of this adjustment is to expedite the convergence of the algorithm.

By performing the adjustment, the fast restoring algorithm aims to ensure that the divisor is approximately centered within the remaining bits of the dividend, increasing the chances of subsequent subtraction steps being successful. This reduces the number of iterations needed to complete the division.Overall, the fast restoring division algorithm improves the speed of the division operation compared to the slow restoring algorithm. However, it requires additional hardware resources to perform the adjustment in step 3, making it more complex to implement in hardware designs.

Both slow restoring and fast restoring division algorithms are fundamental techniques used in computer architectures to perform division operations efficiently. The choice between them depends on factors such as hardware constraints, performance requirements, and trade-offs between implementation complexity and speed.

## 

## Methodology for Fast Division Algorithm:

## *Step 1*: In this step , the corresponding value will be intilized to the register, A will contain value 0 , register M will contain Divisor , register Q will contain Dividend , and N is used to specify the number of bits in dividend.

## *Step 2*: in this step, we will check the sign bit of A.

## *Step 3*:  If this bit of register A is 1, then shift the value of AQ through left, and perform A = A + M. If this bit is 0, then shift the value of AQ into left and perform A = A - M. That means in case of 0, the 2's complement of M is added into register A, and the result is stored into A.

## *Step 4*:  Now, we will check the sign bit of A again.

## *Step 5:* If this bit of register A is 1, then Q[0] will become 0. If this bit is 0, then Q[0] will become 1. Here Q[0] indicates the least significant bit of Q.

## *Step 6 :* After that, the value of N will be decremented. Here N is used as a counter.

## *Step 7:*  If the value of N = 0, then we will go to the next step. Otherwise, we have to again go to step 2.

## *Step 8:* We will perform A = A + M if the sign bit of register A is 1.

## *Step 9:*  This is the last step. In this step, register A contains the remainder, and register Q contains the quotient.

**Flowchart ( non-restoring (fast )division algorithm)**

## 

N<-no. of bits

M<- Divisor

A<- 0

Q<-Dividend

Sign bit of A

Shift left: A, Q

A = A - M

Shift left: A, Q

A = A - M

Sign bit of A

Q[0]=0

Q[0]=1

N < = N- 1

N = N -1

If N = 0

Sign bit of A

A = A + M

Quotient is in register Q

And remainder is in register A

STOP

## Methodology for Restoring division algorithm

## Step 1: Initialize the dividend and divisor registers with the dividend and divisor, respectively. Also, initialize the quotient and remainder registers with zero.

## Step 2: Determine the sign of the quotient based on the signs of the dividend and divisor. Set the sign bit accordingly.

## Step 3: If the divisor is zero, handle the division by zero error.

## Step 4: Determine the initial guess for the quotient by dividing the most significant bits of the dividend and divisor. Place the initial guess in the quotient register.

## Step 5: Subtract the product of the divisor and quotient from the dividend register and store the result in the remainder register.

## Step 6: Check the sign of the remainder:

## Step 7: If the remainder is negative, add the divisor to the remainder and shift the quotient register left by 1 bit, setting the least significant bit to zero.

## If the remainder is non-negative, shift the quotient register left by 1 bit and set the least significant bit to one.

## Step 8: Repeat steps 5 and 6 until the number of iterations equals the number of bits in the dividend.

## FLOWCHART( restoring (slow )division algorithm)

N<-no. of bits

M<- Divisor

A<- 0

Q<-Dividend

Yes

Yes

No

N= 0

N < = N- 1

Q˳ < = 0

A < = A + M

restore

Q˳ < = 1

A < 0?

Shift left: A, Q

A = A - M

## FINITE-STATE MACHINE(FSM)

## A state machine, also known as a finite-state machine (FSM), is a mathematical model used to represent and control the behavior of a system or process. It is an abstract machine that can exist in a finite number of states at any given time and can transition from one state to another based on a defined set of rules or events.

## The basic components of a state machine include:

## States: These are the distinct conditions or modes that the system can be in. Each state represents a specific behavior or configuration of the system.

## Transitions: Transitions define the rules or events that cause the system to move from one state to another. They represent a change in the behavior or configuration of the system.

## Events: Events are the inputs or triggers that initiate a state transition. They can be external stimuli, internal conditions, or time-based occurrences.

## Actions: Actions are the activities or operations that are performed when a state transition occurs. They represent the behavior or tasks associated with a particular state.

## The behavior of a state machine can be visualized using a state diagram or state chart, which shows the states, transitions, and events in a graphical representation. State machines are widely used in various domains, including software engineering, control systems, protocol design, and artificial intelligence, to model and control the flow of operations and decision-making processes.

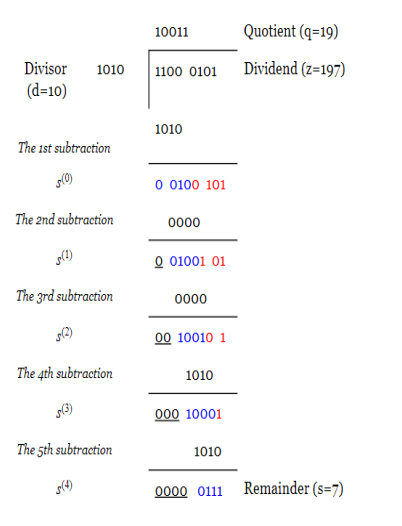
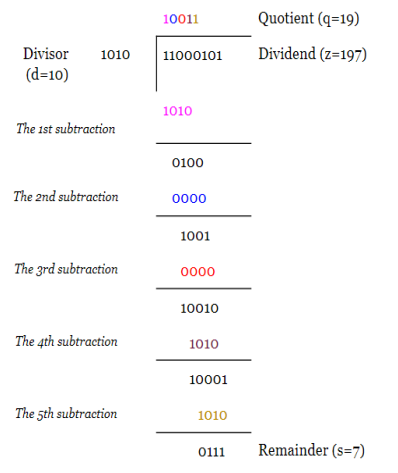
## RTL viewer:-

## 

## State diagram

## 

## How can solve the division :-BINARY DIVISION METHOD :-To begin, contemplate associate example of dividing 11000101 by 1010. Just as in decimal division, we are able to compare the four most vital bits of the dividend (i.e., 1100) with the divisior to seek out the primary digit of the quotient. We tend to are operating with binary numbers, that the digits of the quotient cn be either zero or one. Since 1100 is larger than 1010, the primary digit of the quotient are going to be one. The obtained digit should be increased by the divisior and also the result must be subtracted from the dividend. Hence, we have Now, we should write the next bit of the dividend (shown in red) to the right of the difference and continue the procedure just as we do in a decimal division. Hence, we obtain The below example shows the decimal equivalent of the parameters as well as the letter used to represent them. We can verify the calculations by evaluating ‘z = q × d + s’and that s<d. First, the divisior is subtracted from the four most significant bits of the dividend the default of this subtraction i.e., 1010 is shown in blueTo get a better insight into the implementation of the division algorithm, we rewrite the above example as We ca replace the four MSBs of the dividend with 0010 and obtain s(0)=00100101. The four LSBs of s(0), which are the same as the four LSBs of the dividend, are Shown in red, note that we no longer longer need the original dividend and we can replace it with s(0). From an implementation point of view, this means that we can use the register which was originally storing the value of the dividend to store s(0).For the second subtraction, the divisor is shifted to the right by one bit. After subtraction, we obtain (1)=00100101. Again, the bits obtained from subtraction are shown in blue and the unaltered bits of s(0) are shown in red. We can now update the dividend register with s(1). This procedure goes on until the final subtraction in which the LSB of the shifted divisor is aligned with the LSB of the dividend. After this final subtraction the dividend. After this final subtraction, the remainder will be less than the divisor. Note that, as we proceed with the algorithms, the high order bits of the s(.) terms become zero. This suggests that some bit positions of the dividend register will be no longer required. In the next section, we’ll see which bit positions are redundant. In the above example, the bit positions that can be discarded are underscored.



**CODE:-**

//FAST DIVISION ALGORITHM

module slow\_division\_tb;

reg clk;

reg rst;

reg signed [15:0] dividend;

reg signed [15:0] divisor;

wire signed [15:0] quotient;

wire signed [15:0] remainder;

slow\_division dut (

.clk(clk),

.rst(rst),

.dividend(dividend),

.divisor(divisor),

.quotient(quotient),

.remainder(remainder)

);

always #5 clk = ~clk;

initial begin

clk = 0;

rst = 1;

dividend = 32; // Example dividend value

divisor = 5; // Example divisor value

#10 rst = 0; // Deassert reset

#100 $finish; // Finish simulation after some time

end

always @(posedge clk) begin

$display("Dividend: %d, Divisor: %d, Quotient: %d, Remainder: %d",

dividend, divisor, quotient, remainder);

end

endmodule

//TESTBENCH FOR FSAT DIVISION ALGORITHM

module fast\_division\_tb;

reg clk;

reg rst;

reg signed [15:0] dividend;

reg signed [15:0] divisor;

wire signed [15:0] quotient;

wire signed [15:0] remainder;

fast\_division dut (

.clk(clk),

.rst(rst),

.dividend(dividend),

.divisor(divisor),

.quotient(quotient),

.remainder(remainder)

);

always #5 clk = ~clk;

initial begin

clk = 0;

rst = 1;

dividend = 32; // Example dividend value

divisor = 5; // Example divisor value

#10 rst = 0; // Deassert reset

#100 $finish; // Finish simulation after some time

end

always @(posedge clk) begin

$display("Dividend: %d, Divisor: %d, Quotient: %d, Remainder: %d",

dividend, divisor, quotient, remainder);

end

endmodule

//SLOW DIVISION ALGORITHM

module slow\_division (

input clk,

input rst,

input signed [15:0] dividend,

input signed [15:0] divisor,

output signed [15:0] quotient,

output signed [15:0] remainder

);

reg signed [15:0] quotient\_reg;

reg signed [15:0] remainder\_reg;

reg [3:0] count;

always @(posedge clk or posedge rst) begin

if (rst) begin

quotient\_reg <= 0;

remainder\_reg <= 0;

count <= 0;

end else begin

if (count < 16) begin

remainder\_reg <= remainder\_reg << 1;

remainder\_reg[0] <= dividend[15];

if (remainder\_reg >= 0) begin

remainder\_reg <= remainder\_reg - divisor;

quotient\_reg[count] <= 1;

end else begin

quotient\_reg[count] <= 0;

end

count <= count + 1;

end

end

end

assign quotient = quotient\_reg;

assign remainder = remainder\_reg;

endmodule

//TESTBENCH FOR SLOW DIVISION ALGORITHM

module slow\_division\_tb;

reg clk;

reg rst;

reg signed [15:0] dividend;

reg signed [15:0] divisor;

wire signed [15:0] quotient;

wire signed [15:0] remainder;

slow\_division dut (

.clk(clk),

.rst(rst),

.dividend(dividend),

.divisor(divisor),

.quotient(quotient),

.remainder(remainder)

);

always #5 clk = ~clk;

initial begin

clk = 0;

rst = 1;

dividend = 32; // Example dividend value

divisor = 5; // Example divisor value

#10 rst = 0; // Deassert reset

#100 $finish; // Finish simulation after some time

end

always @(posedge clk) begin

$display("Dividend: %d, Divisor: %d, Quotient: %d, Remainder: %d",

dividend, divisor, quotient, remainder);

end

endmodule

**Conclusion:-**

In this paper we report on a novel 16 bit dividend by 16 bit divisor architecture based on the formulas of Mathematics. In this work, division operation was carried out through addition and small digit multiplication that eventually reduces the iteration, owing the achievement of high speed. The pipelined designs of user-defined binary division. The user-defined binary complex division using modified algorithms was done with a co-simulation techniques with Modalism and Simulink. The whole pipelined architectures was synthesized using on FPGA. The fully pipelined implementation tremendously raises the system’s throughput. Hence our designs could be applied to those field with high performance requirement. The design has been able to achieve best power inflation over the past work in the same field.