ElectroBinary

Welcome to the world of Digital Electronics and VLSI

SATURDAY, OCTOBER 24, 2020

Floating Point Addition / Subtraction

Floating Point Representation in digital systems follows the IEEE-754 format. Let us see how to perform addition / subtraction on floating numbers represented in this format.

The operation that will be performed is addition. To perform subtraction, invert the sign bit of the number to be subtracted and send it to the floating point adder.

Algorithm (Verilog approach):

Obtain the two numbers to be added in IEEE-754 single precision format.
The number will be of the form: {sign (1-bit), exponent (8-bits), mantissa (23-bits)} = Total 32-bits
To automate this number conversion, you can use the h-schmidt converter.

Note: When we say mantissa, it does not consider the 'implicit 1' that is present before the decimal point. However for computations, we include the mantissa with the 'implicit 1', perform the required computations and then discard the implicit 1.

For the below algorithm, the word mantissa represents {1'b1, 23-bit mantissa} = 24-bits

1. Compare the exponents of the two numbers. Find the absolute difference (abs_diff) of the two exponents.
2. Now shift right the mantissa of the number with smaller exponent by abs_diff times so that now the exponents become equal. This is the exponent value of the sum.

CATEGORIES

assembly (7)
concepts (6)
design (2)
perl (3)
questions (3)
systemverilog (2)
tools (7)
tutorial (6)
verilog (20)

POPULAR POSTS

Booth Multiplier Verilog Code

Booth's Multiplication Algorithm is a commonly used algorithm for multiplication of two signed numbers. Let us see how to write a Verilo...

Synchronous FIFO Verilog Code

The First In First Out (FIFO) is a data arrangement structure in

3. Check the sign bit of the two numbers. If both are positive, add the two mantissas. If negative, then subtract the mantissas accordingly. If borrow has occurred during subtraction, take 2s complement of the mantissa.
4. Call the obtained mantissa as sum_mantissa. The sum will contain 25-bits to accommodate overflow.
5. Now the sum_mantissa needs to be normalized if needed. As we know, the sum_mantissa includes the implicit 1. This implicit 1 needs to be at bit location sum_mantissa[23] to comply with IEEE-754 format. If sum_mantissa[23] = 1, then no normalization is required.
6. For achieving this, search for the first 1 occurring in sum_mantissa traversing right from MSB. Shift sum_mantissa in such a way that this 1 occurs at location sum_mantissa[23]. According to the shifts, the exponent of sum must also be adjusted.
7. Now discard the implicit 1 and consider only the lower 23-bits of sum_mantissa. This will be the final value of sum_mantissa.
8. Thus we have calculated the sign, exponent and mantissa of the sum. Combining the three in IEEE-754 format, we will have obtained the final result.

Verilog Code Logic:

The same logic as elucidated above will be implemented in Verilog. A completely synthesizable FSM model is implemented with the following signals:

Inputs:

clk, rst, X, Y (2 float operands to be added/subtracted), start (This pulse indicates start of computation)

Outputs:

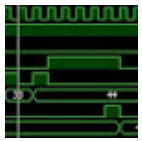
sum (Result) and valid (This pulse indicates the availability of final result)

1. In IDLE state, we wait for the start pulse. Upon its arrival, move to START state.
2. In START state, we follow the algorithm for performing Float Addition/Subtraction and obtain the sum_mantissa.
3. Final normalization step is performed in SHIFT_MANT state.

Verilog Code:

```
module Float_Add(clk,rst,start,X,Y,valid,sum);

input clk;
input rst;
```

which the data that enters first is the one that is removed first. Let us...



Stack or LIFO Verilog Code

The Last In First Out (LIFO) or Stack is a data arrangement structure in which the data that enters the last is the one that is removed fi...

Restoring Division Verilog Code

Restoring Division Algorithm is one of the division algorithms used for performing division in digital systems. Let us see how to write the...

Clock Divider Verilog Code

In this post, we will be implementing a Clock Divider circuit in Verilog. This is one of the most important circuits in VLSI as we ca...

```
input start;
input [31:0]X,Y;        //In IEEE754 format
output [31:0]sum;       //In IEEE754 format
output valid;

reg X_sign;
reg Y_sign;
reg sum_sign,next_sum_sign;
reg [7:0] X_exp;
reg [7:0] Y_exp;
reg [7:0] sum_exp,next_sum_exp;
reg [23:0] X_mant, Y_mant;
reg [23:0] X_mantissa, Y_mantissa; //24 bits = 23 + implicit 1
reg [24:0] sum_mantissa, next_sum_mantissa; //sum is 25 bits

reg [8:0] expsub,abs_diff;
reg [24:0] sum_mantissa_temp;
reg [1:0] next_state, pres_state;
reg valid, next_valid;

parameter IDLE = 2'b00;
parameter START = 2'b01;
parameter SHIFT_MANT = 2'b10;

assign sum = {sum_sign,sum_exp,sum_mantissa[22:0]};
assign add_carry = sum_mantissa[24] & !(X_sign ^ Y_sign);       //carry during add
assign sub_borrow = sum_mantissa_temp[24] & (X_sign ^ Y_sign); //borrow during sub

always @ (posedge clk or negedge rst)
begin
if(!rst)
begin
        valid        <= 1'b0;
        pres_state   <= 2'd0;
        sum_exp      <= 8'd0;
        sum_mantissa <= 25'd0;
        sum_sign     <= 1'b0;
end
else
begin
        valid        <= next_valid;
        pres_state   <= next_state;
        sum_exp      <= next_sum_exp;
```

ASIC Synthesis using Synopsys Design Compiler

► August (4)
► June (6)
► May (7)
► April (13)
► March (7)

HI READER

If you find any content issues or code bugs in this blog, add a comment or notify me using the contact form in the About page. Any other suggestions are also welcome!

```
        sum_mantissa <= next_sum_mantissa;
        sum_sign     <= next_sum_sign;
end
end

always @ (*)
begin
next_valid = 1'b0;
X_sign = X[31];
X_exp = X[30:23];
X_mant = {1'b1,X[22:0]};
Y_sign = Y[31];
Y_exp = Y[30:23];
Y_mant = {1'b1,Y[22:0]};
next_sum_sign = sum_sign;
next_sum_exp = 8'd0;
next_sum_mantissa = 25'd0;
next_state = IDLE;
sum_mantissa_temp = 25'd0;

case(pres_state)
IDLE:
begin
        next_valid = 1'b0;
        X_sign = X[31];
        X_exp = X[30:23];
        X_mant = {1'b1,X[22:0]};        //Implicit 1 added
        Y_sign = Y[31];
        Y_exp = Y[30:23];
        Y_mant = {1'b1,Y[22:0]};
        next_sum_sign = 1'b0;
        next_sum_exp = 8'd0;
        next_sum_mantissa = 25'd0;
        next_state = (start) ? START : pres_state;
end

START:
begin
    expsub = X_exp - Y_exp;
    abs_diff = expsub[8] ? !(expsub[7:0])+1'b1 : expsub[7:0];   //Absolute difference
    X_mantissa = expsub[8] ? X_mant >> abs_diff : X_mant;       //X mant shifts if expsub[
    Y_mantissa = expsub[8] ? Y_mant : Y_mant >> abs_diff;   //Y mant shifts if !expsub[8]
    next_sum_exp = expsub[8] ? Y_exp : X_exp;                   //Greater exp taken
```

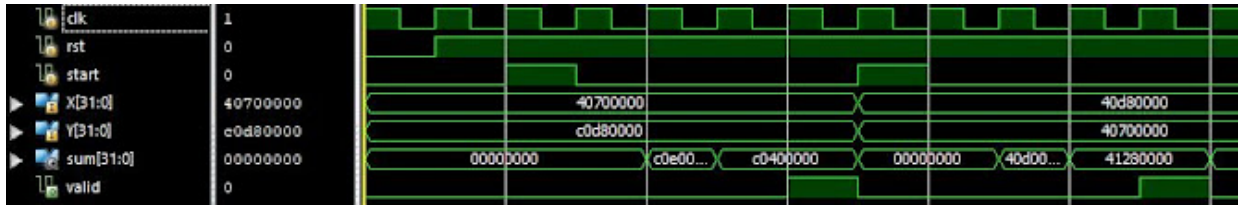
```
    sum_mantissa_temp = !(X_sign ^ Y_sign) ? X_mantissa + Y_mantissa :  //Add mantissas
                        (X_sign) ? Y_mantissa - X_mantissa :
                        (Y_sign) ? X_mantissa - Y_mantissa : sum_mantissa;
    next_sum_mantissa = (sub_borrow) ? ~(sum_mantissa_temp)+1'b1 : sum_mantissa_temp; //2s
    next_sum_sign = ((X_sign & Y_sign) || sub_borrow);
    next_valid = 1'b0;
    next_state = SHIFT_MANT;
end

SHIFT_MANT:                                     //State to shift Mantissa to make bit23 =
begin
    next_sum_exp = sum_mantissa[23] ? sum_exp : (add_carry)? sum_exp + 1'b1 : sum_exp - 1'
    next_sum_mantissa = sum_mantissa[23] ? sum_mantissa : (add_carry) ? sum_mantissa >> 1
    next_valid = sum_mantissa[23] ? 1'b1 : 1'b0;
    next_state = sum_mantissa[23] ? IDLE : pres_state;
end

endcase
end
endmodule
```

Testbench:

```
module Float_Add_tb;

reg clk,rst,start;
reg [31:0]X,Y;
wire [31:0]sum;
wire valid;

always #5 clk = ~clk;

Float_Add inst (clk,rst,start,X,Y,valid,sum);

initial
$monitor($time,"X=%d, Y=%d, sum=%d ",X,Y,sum);

initial
begin
//X=32'h40d80000; Y=32'hc0700000;       //6.75,-3.75
//X=32'hc0d80000; Y=32'h40700000;       //-6.75,3.75
```

```
X=32'h40700000; Y=32'hc0d80000;          //3.75,-6.75
//X=32'hc0700000; Y=32'h40d80000;        //-3.75,6.75
clk=1'b1; rst=1'b0; start=1'b0;
#10 rst = 1'b1;
#10 start = 1'b1;
#10 start = 1'b0;
@valid
#10 X=32'h40d80000; Y=32'h40700000;       //6.75,3.75
//#10 X=32'hc0d80000; Y=32'hc0700000;     //-6.75,-3.75
start = 1'b1;
#10 start = 1'b0;
end
endmodule
```

Simulation Result:



1. The waveform has been displayed in Hex notation. The first inputs given are 3.75 and -6.75. The output obtained is -3 in IEEE-754 format.
2. The second set of inputs given are 6.75 and 3.75. The output obtained is 10.5 in IEEE-754 format.
3. Verify the number in IEEE-754 format using the h-schmidt converter.

Conclusions:

This Floating Point Addition Algorithm implemented in this format can be implemented on FPGA devices.
Number of clock cycles taken to produce the output depends on how much shifting is to be done to normalize the mantissa of sum.
For a better understanding of Floating Point Addition, check this tutorial.

Posted by Admin at 7:52 PM

Labels: verilog

No comments:

Post a Comment

Enter Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)