IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)
Volume 5, Issue 5, Ver. I (Sep - Oct. 2015), PP 54-59
e-ISSN: 2319 – 4200, p-ISSN No. : 2319 – 4197
www.iosrjournals.org

Design and Implementation of Floating Point ALU with Parity Generator Using Verilog HDL

Mohammad Ziaullah[1], Abdul Munaff[2]
[1](Asst. Professor, Electronics and Communication Engineering, S.I.E.T Vijayapur, Karnataka, India)
[2](Asst. Professor, Electrical and Electronics Engineering, S.I.E.T Vijayapur, Karnataka, India)

Abstract: *A floating-point unit (FPU) colloquially is a math coprocessor, which is a part of a computer system specially designed to carry out operations on floating point numbers. Typical operations that are handled by FPU are addition, subtraction, multiplication and division. The functions performed are handling of Floating Point data, converting data to IEEE754 format, perform any one of the following arithmetic operations like addition, subtraction, multiplication, division . All the above algorithms have been evaluated under Modelsim environment. All the functions are built by possible efficient algorithms with several changes incorporated at our end as far as the scope permitted. Consequently all of the unit functions are unique in certain aspects and given the right environment these functions will tend to show comparable efficiency and speed ,and if pipelined then higher throughput.*

Keywords: *Floating, Algorithm, coprocessor, pipelined, throughput.*

I. Introduction

In computing, an arithmetic logic unit (ALU) is a digital circuit that performs arithmetic and logical operations. The ALU is a fundamental building block of the central processing unit (CPU) of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers. The processors found inside modern CPUs and graphics processing units (GPUs) accommodate very powerful and very complex ALUs; a single component may contain a number of ALUs. ALU is a necessity for a computer because it is guaranteed that a computer will have to compute basic mathematical operations, including addition, subtraction, multiplication, and division.

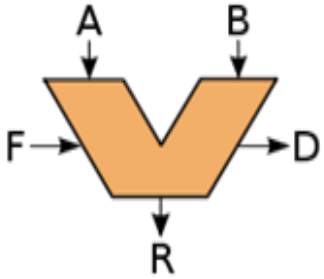

Fig1:General ALU

II. Floating Point ALU

2.1 Floating Point Unit

When a CPU executes a program that is calling for a floating-point (FP) operation, there are three ways by which it can carry out the operation. Firstly, it may call a floating-point unit emulator, which is a floating-point library, using a series of simple fixed-point arithmetic operations which can run on the integer ALU. These emulators can save the added hardware cost of a FPU but are significantly slow. Secondly, it may use an add-on FPUs that are entirely separate from the CPU, and are typically sold as an optional add-ons which are purchased only when they are needed to speed up math-intensive operations. Else it may use integrated FPU present in the system . The FPU designed by us is a single precision IEEE754 compliant integrated unit. It can handle not only basic floating point operations like addition, subtraction, multiplication and division but can also handle operations like shifting, logical operations.

2.2 Floating point ALU Model

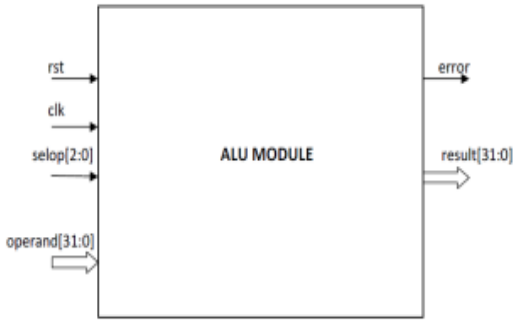

Fig2: Top model

Floating point ALUs are used for high precision computing. This ALU uses 32 bit numbers, which is the common computer word length. The numbers are represented in IEEE 754 standard. This standard is widely used in floating point arithmetic. The ALU can perform the arithmetic operation: addition, subtraction, multiplication and division. It can also perform some logical operations such as logical NOT, logical NAND and shift right. To support RISC design the arithmetic operations are done in pipelined fashion. There are two pipelined units in this ALU: the add/subtract pipeline and mult/div pipeline. The addition and subtraction operations are done in the add/subtract pipeline and the multiplication and division operations are done in the mult/div pipeline

III. IEEE 754 Standards

IEEE754 standard is a technical standard established by IEEE and the most widely used standard for floating-point computation, followed by many hardware (CPU and FPU) and software implementations [3]. Single-precision floating-point format is a computer number format that occupies 32 bits in a computer memory and represents a widedynamic range of values by using a floating point. In IEEE 754-2008, the 32-bit with base 2 format is officially referred to as single precision or binary32. It was called single in IEEE 754-1985. The IEEE 754 standard specifies a single precision number as having sign bit which is of 1 bit length, an exponent of width 8 bits and a significant precision of 24 bits out of which 23 bits are explicitly stored and 1 bit is implicit 1.

Sign bit determines the sign of the number where 0 denotes a positive number and 1 denotes a negative number. It is the sign of the mantissa as well. Exponent is an 8 bit signed integer from -128 to 127 (2's Complement) or can be an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 single precision definition. In this case an exponent with value 127 represents actual zero. The true mantissa includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the mantissa appear in the memory format but the total precision is 24 bits. IEEE754 also defines certain formats which are a set of representation of numerical values and symbols. It may also include how the sets are encoded.

3.1 Conversion Of The Binary Integer To Its Ieee754 Format

As our FPU is IEEE754 compliant, the next step is to convert the input (here the effective operand into the IEEE specified format. IEEE754 single precision can be encoded into 32 bits using 1 bit for the sign bit (the most significant i.e. 31st bit), next eight bits are used for the exponent part and finally rest 23 bits are used for the mantissa part.

For example:

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF

31 30 23 22 0

However, it uses an implicit bit, so the significant part becomes 24 bits, even though it usually is encoded using 23 bits.

This conversion can be done using the below algorithm of above example

Step1: Sign bit of the binary number becomes the sign bit (31st bit) of the IEEE equivalent.

Step 2: 30th bit to 8th bit of the binary number becomes the mantissa part of the IEEE equivalent.

Step 3: The exponent part is calculated by subtracting the position of the 1st one obtained in the algorithm described in section 2.2.1.

Step 4: A bias of 127 is added to the above exponent value.

3.2 Pre-Normalization Of The Operands

Pre-normalization is the process of equalizing the exponents of the operands and accordingly adjusting the entire IEEE754 expression of the inputs to produce correct results maintaining the IEEE754 standard throughout all calculation steps inclusive of the intermediate calculations and their outputs.

This conversion can be done using the below algorithm

Step 1: Insert the implicit 1 in the mantissa part of each of the operands.

Step 2: Find positive difference between the exponents of the operands

Step 3: Set the lower operand.s exponent same as that of the operand with higher exponent.

Step 4: Right shift mantissa of the lower operand by steps equal to difference calculated.

3.3 Performing The Selected Operation

After completion of the preliminary steps the next step is to perform the actual operation. The choice of operation is taken as input via a 4 bit wire oper. Following is the table 2.1 that describes the functions and their corresponding operation code.

fpu_op	Operation
000	Add
001	Subtract
010	Multiply
011	Divide
100	Shifting

3.4 Module Add

Addition is a mathematical operation which represents combining a collection of objects together to form larger collection. The process of developing an efficient addition module in our FPU was an iterative process and with gradual improvement at each attempt.

3.4.1 Add Using The ¡°+¡± Operator

The initial attempt was to add using the simple in-built ¡°+¡± operator available in Verilog library. It used a 23 bit register sum and a 1 bit register Co (for carry). The algorithm for the addition can be described below

Step 1: Check if oper = 4.b000

Step 2: {Co,Sum} = Temp_op1_ieee[22:0] + Temp_op2_ieee[22:0]

Step 3: If carry is 1, then. Resultant_exponent = Larger_exponent + 1;

Else if carry is 0, then do

Resultant_exponent = Larger_exponent . (21-difference) (difference as in sec.2.2.3)

Step 4: Check for overflow and underflow-

.If for any of the operands (sign(operand with greater exponent)==0 & (exp_greater + 1 > 255)) then, Set the overflow flag to 1.

Else if (sign(operand with lesser exponent==0) & (exp_lesser<0)), then set the underflow flag to 1

Step 5: Aggregate the result as concatenation of {Sign_bit,Resultant_exponent,Sum}

3.5 Subtract Module

Subtraction is an operation which is treated as inverse of addition operation. The process of developing an efficient SUB module followed the iterative development of the ADD module.

3.5.1 Sub Using The ¡°-¡± Operator

The initial attempt was to subtract using the simple in-built ¡°-¡± operator available in Verilog library. It used a 23 bit register diff and a 1 bit register borrow (for borrow). The algorithm for the subtraction module can be described.

Step 1: Check if oper = 4.b0001

Step 2: {borrow.diff} = Temp_op1_ieee[22:0] - Temp_op2_ieee[22:0]

Step 3: Resultant_exponent = Larger_exponent + (21-difference)

Step 4: Check for overflow and underflow-

If for any operand (sign(operand with greater exponent)==1 AND (exp_greater + 1 < 0))

Set the overflow flag to 1
If for any operand (sign(operand with exponent)==1'b1 AND (exp_lesser>8'd255))
Set the underflow flag to 1
Step 5: Aggregate the result as concatenation of {Sign_bit,Resultant_exponent,diff}

3.6 Multiplication Module

The process of developing an efficient multiplication module was iterative and with gradual improvement at each attempt. The product of two n-digit operands can be accommodated in 2n-digit operand.

3.6.1multiplication Using ¡°*¡± Operator

It used a 47 bit register to store the product. The algorithm is explained
Step 1: Check if oper = 4.b0010
Step 2: product = Temp_op1_ieee[22:0] * Temp_op2_ieee[22:0]
Step 3: Resultant_exponent = op1_ieee[30:23] + op2_ieee[30:23] - 127
Step 4: If for product (Resultant_exponent >255), then do,
. Set the overflow flag to 1
Step 5: Sign_bit = op1_ieee[31] ^op2_ieee[31]
Step 6: Aggregate the result as concatenation of { Sign_bit, Resultant_exponent, product }

3.7 Module Division

Division is regarded as the most complex and time-consuming of the four basic arithmetic operations. Given two inputs, a dividend and a divisor, division operation has two components as its result, quotient and a remainder.

3.7.1 Division Using ./. Operator

The initial attempt was to divide two numbers using the simple in-built ¡°/¡± operator available in Verilog library. It used a 32 bit result_div_ieee register to store the quotient and register remainder to store the remainder of the division operation.
Step 1: Check if the oper = 4 bit 0100
Step 2: result_div_ieee = temp_op1_ieee[22:0] / temp_op2_ieee[22:0]
Step 3: If op2_ieee[30:0] is all 0
. Set div_bby_zero flag to 1
Step 4: Aggregate the result as concatenation of {Sign_bit, Resultant_exponent, result_div_ieee}

IV. Block Diagram

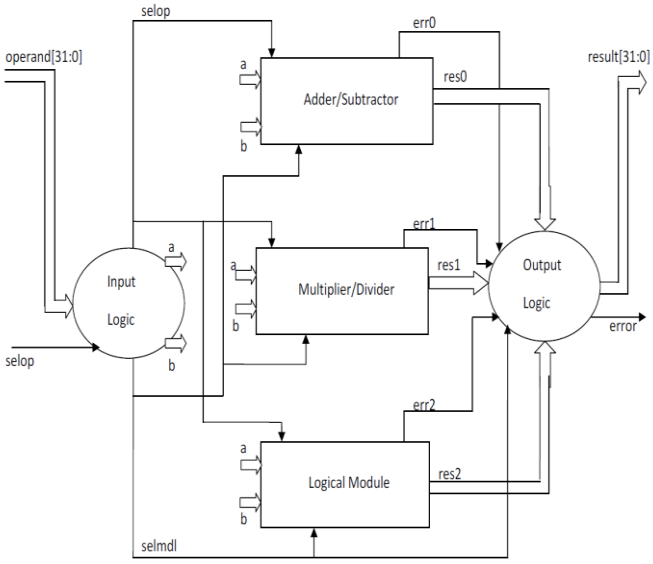

Fig.3 Block Diagram of ALU

In the ALU top module there are in total 70 I/O pins. It include 32 pins for input operand , 32 pins for output result, 3 pins for selop, 1 pin for rst,1 pin for clk and 1 output pin for error. For actual arithmetic

calculations we need two operands but in a microprocessor there is only one databus, and we have to multiplex this databus. If selop is 000, it is considered as first operand and if selop have any other values, then the operand that present at that time is considered as the second operand.

Selop	Operation
000	Select operand 'a'
001	Add operand 'a' and 'b'
010	Subtract operands 'a' and 'b'
011	Multiply operands 'a' and 'b'
100	Divide operands 'a' and 'b'
101	Logically NOT operand 'a'
110	Logically NAND operand 'a' and 'b'
111	Shift right the operand 'a'

4.2 Block diagram description

The floating point ALU is internally divided in to three logical modules: the adder/subtracter module, the multiplier/divider module and the logical module. All the input parameters are given to the input logic and also the output is taken from the output logic. The input parameters to the ALU include operand, selop, clk and reset. The output from the ALU includes 32 bit result and error signal. Here selop is used to selet an operation. In other words the selop is equivalent to the instruction set of a microprocessor. The selop is three bit wide, and so there are totally seven different operations are possible. All these operations are given in the table.

4.3 Input Logic

The input logic receives two 32 bit operands one by one and store it in registers a and b. According to the selop signal, the input logic produces the semdl signal to activate any one of the logical module.

4.4 Adder/Sbtracter

The the adder/subtracter module performs the addition and subtraction operations according to the selop signal. Internally it is a four level pipelined add/sub unit.

4.5 Multiplier/Divider

The the multiplier/divider module performs the multiplication and division operations according to the selop signal. Internally it is a four level pipelined add/sub unit.

4.6 Logical Module

The logical module can perform logical NOT, logical NAND and logical right shift operations. These operations are performed according to the selop signal.

4.7 Output Logic

The output logic accepts the three results res0, res1 and res2 respectively from adder/subtracter, multiplier/divider and logical module. It directs any one of them to the final result according to the selop signal.

V. Simulation Results

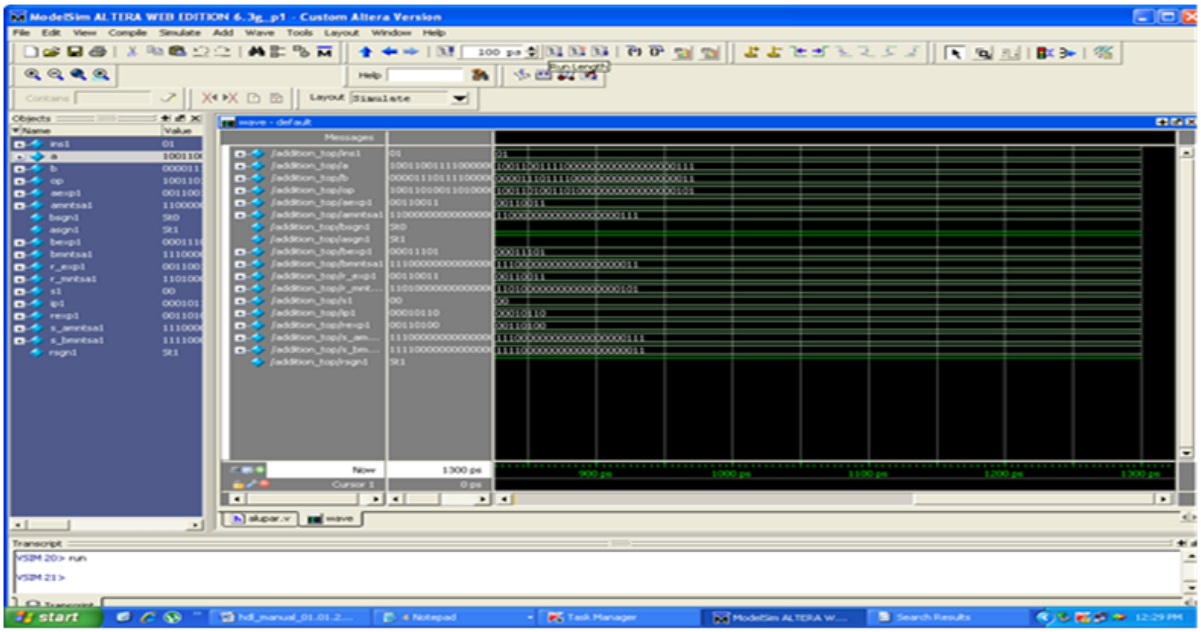

Fig 4:Simulation Result waveform

Synthesis Results

After the simulation of the code was successful, we proceeded for the synthesis analysis. The simulation results gave a detailed description of the inputs used showing separately sign bit, exponent bits and mantissa of inputs and outputs of the operation.

VI. Conclusion

Hence, Floating Point ALU is successfully designed and implemented by using verilog description language and simulated using modelsim6.3g version. The addition of two floating point numbers are determined by showing separately sign,exponent,mantissa.

References

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[2] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96), pp. 107–116, 1996.

[3] A. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp. 897-900.

[4] "DesignChecker User Guide", HDL Designer Series 2010.2a, Mentor Graphics, 2010

[5] "Precision® Synthesis User's Manual", Precision RTL plus 2010a update 2, Mentor Graphics, 2010.

[6] Patterson, D. & Hennessy, J. (2005), Computer Organization and Design: The Hardware/software Interface , Morgan Kaufmann

[7]] John G. Proakis and Dimitris G. Manolakis (1996), "Digital Signal Processing: Principles,. Algorithms and Applications", Third Edition.

[8] R.V.K Pillai, D. Al-Khalili, and A.J. Al-Khalili. A Low Power Approach to Floating Point Adder Design. In Proceedings of ICCD '97., Computer Design: VLSI in Computers and Processors, pages 178{185, Austin, TX, October 1997.ICCD.

[9] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago, "Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM", 2005, University of Southern California/Information Sciences Institute