

Implementation of IEEE 32 Bit Single Precision Floating Point Addition and Subtraction

Khushbu Naik
EC-M.Tech UTU university
Bardoli-394601,
Gujarat,India
khushbu060192@gmail.com

Prof.Tarun Lad
EC-Department,UTU university
Bardoli-394601
Gujarat,India
tarun.lad@utu.ac.in

ABSTRACT

This paper presents a floating-point addition and subtraction algorithm and their pipeline design. Floating point unit have different operations which is hard to implement on FPGAs due to complexity of their algorithms. Many scientific applications require more accuracy in result. For that reason, we have explored implementation of addition and subtraction for IEEE single precision floating point numbers. We implemented trade-off between area and speed for accuracy. We have implemented adder and subtractor as a bit-parallel adder. The algorithms are designed in VHDL language and can be implemented on FPGA kit by use of Xilinx ISE compiler. Floating point adder and subtractor unit design using pipelining which provides high performance and increase the speed.it used for execute multiple instructions simultaneously. The language is used for coding is VHDL and tool is Xilinx ISE.

Keywords

Floating point arithmetic, algorithm, IEEE 754 format, simulation, VHDL.

1. INTRODUCTION

The main advantage of floating point representation compare to fixed point number is it supports wide range of values. The disadvantages of fixed numbers are it can't support of fractional numbers and limited dynamic range. On the other hand, many problems like text to speech converter require adder and subtractor with high accuracy of calculations [3,6].

Many of problems have a high degree of regularity that makes them good candidates for hardware implementations. Because of these reason need for 32-bit floating point adder and subtractor implemented in FPGAs arises. Floating point addition and subtraction are hard to implement on FPGAs due to complexity of algorithms [1].

The adder and subtractor built as a parallel structure and pipelining technique for increase through-put. In this paper we start the paper by briefly reviewing in first section IEEE standard format.in the next section, the adder and subtractor are described briefly algorithms. The floating point number representation shown in figure 1, main four components: the sign bit, the significant s,the base of exponent and the exponent [2].

+/-	e:exponent	S:significant
-----	------------	---------------

Figure.1: Representation of a Floating-Point Number in IEEE standard format[3]

The use of exponent biased format has no effect on cost and speed of adder and subtractor unit, the small number of bits used.it give facilitate zero detection, by use of small format, smaller and faster implementation can be built but we achieved less accuracy in calculations. For that in our implementation, accuracy is main objective for 32-bit operates was designed, the bit parallel adder give more chip area but have more speed[4].

2. IEEE standard floating point format

Table 1
Features of the ANSI/IEEE Standard
Floating-Point Representation

Features	Single
Word length, bits	32
Significant bits	23 + 1(hidden)
Significant range	[1,2-2 ²³]
Exponent bits	8
Exponent bias	127
Zero(± 0)	E+bias=0, $f = 0$
Denormal	E+bias=0, $f \neq 0$
Infinity($\pm \infty$)	E+bias=255, $f = 0$
Not-a-number(NAN)	E+bias=255, $f \neq 0$
Minimum	$2^{-126} \approx 1.2 * 10^{-38}$
Maximum	$\approx 2^{128} \approx 3.4 * 10^{38}$

A main aim to developing such a standard, floating point representation standard which make numerical programs predictable and completely portable. The standard specifies three types of formats for floating point number representation: basic (single precision), extended (double precision) and Quadruple precision [5].

2.1 floating point format for single precision

The MSB starts from left to right. The basic three components are the sign bit, the exponent and the significant (mantissa).

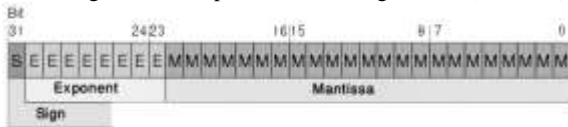


Figure 2: Single Precision Format for floating point numbers

Single precision floating point numbers are represented by 32 bits. The sign has 1 bit width, exponent has 8 bit width and the significant (mantissa) has 23 bit width (1 + 8 + 23). The range of single precision numbers that can be represented is $(-2^{-126-22}; 2 * 2^{127})$.the number in floating point single precision is composed of the below three fields[10]:

- (1) **Sign bit S:** the value of $s = 1$ indicates that the number is negative, and a $s = 0$ indicates a positive number.
- (2) **Biased exponent, $e = E + bias$:** This gives us an exponent range from $E_{min} = -126$ to $E_{max} = 127$.
- (3) **Significant:** The fractional part of the number.

The classes of floating point single precision numbers are as follows:

- **Normalized numbers:** the bias is $2^{8-1} - 1 = 127$; the range of the exponent is $[-126:127]$, while its binary value is in the range $[1:254]$
- **Denormalized numbers:** the exponent is -126 , while its binary value is 0 (as is the case for denormalized numbers)
- **Infinites & NaN:** these special representation have a binary value of $2^8 - 1 = 256 - 1 = 255$ for the exponent (all ones)

3. Floating point adder and subtractor

3.1 addition and subtraction: overview

Floating-point numbers are coded as sign bit, reversing the sign-bit inverts the sign. The same operator performs addition or subtraction according to the two operand's signs. Addition/subtraction of floating-point numbers $S = A \pm B$ and it is more hard to implement than multiplication and division of floating point numbers.

Following steps are used for addition and subtraction for floating point numbers.

- **significands alignment** if A and B exponents are different to each other.
- **Addition/subtraction** of the aligned significands,
- **Renormalization** of the significand sum S if not already normalized,
- **Rounding** of sum S.

3.2 addition and subtraction algorithms

Here, we use two operands i.e. N_1 and N_2 for that the algorithms shown in fig.3 to compute their successive addition or subtraction. For that e_1 and e_2 are the exponent and s_1 and s_2 are the significands of the numbers [7,8].

The other detailed description of algorithms is given below:

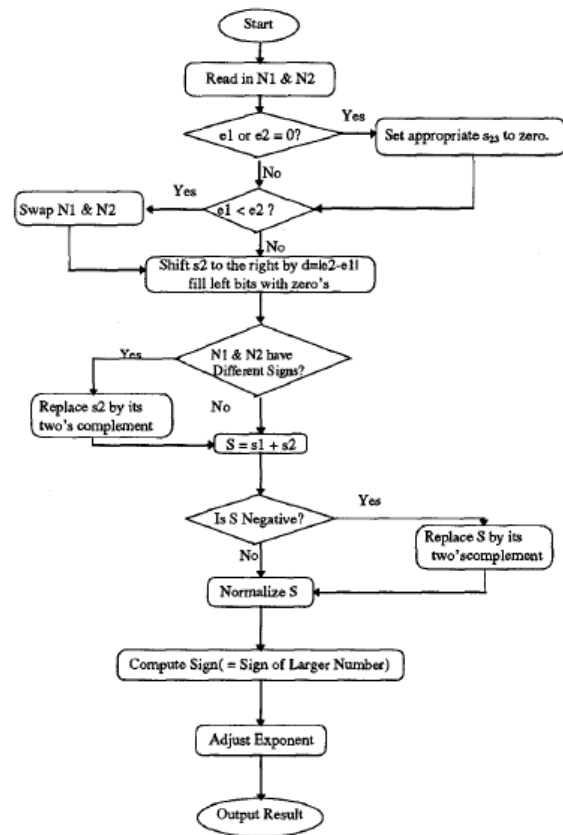


Figure 3: Floating Point Addition Algorithm

1. First the 24th hidden bit explicit. If $e_1 = 0$ or $N_1 = 0$ and make it directly 0, otherwise $e_1 \neq 0$ or $N_1 \neq 0$ then make it '1' at that stage 33 bits are necessary to store the number, in that 8 bits for the exponent e , 24 bits for the significand s and 1 bit for the sign.

2. Compare the exponent e_1 and e_2 . if the value of exponent of e_2 is larger than e_1 i.e. $e_2 > e_1$. if this condition is satisfied then swap both N_1 and N_2 . this swapping is occurred then $s_1(e_1)$ will referring to the $s_2(e_2)$ and $s_2(e_2)$ will referring as $s_1(e_1)$. also, find the difference between exponent values $d = (|e_2 - e_1|)$.

3. Shift significand s_2 shift right by difference value $d = (|e_2 - e_1|)$. fill the left part bits with simple zeros.

4. If the operands N_1 and N_2 have different signs bit, replace significand s_2 by its 2's complement.

5. Compute the final sum i.e. s , by simply add s_1 and s_2 .

6. The final significand of sum s is negative then sum s is replaced by its 2's complement. The sum s is negative then the below conditions is true:

- (1) The operands have different signs.
- (2) the msb of significand sum of s is '1'.
- (3) no carry out in step 5.

7. Normalization steps for significand sum,

- (1) If operands have same sign and there was carry-out in step 5 then the significands sum s shift right by one, drop the lsb bit and fill up MSB with 1.
- (2) Else, shift sum of significands s left up to there is a '1' in msb. the number of left shifts must be stored.

(3) If sum s was shifted by left more than 24 times, then result should be directly zero.

8. now, the sign of results output is determined by make the output sign by the larger number of operand sign i.e. N_1 and N_2 . this sign bit if sign is positive then it will be '0' and negative then it will be '1' is replaced as a msb of s with this sign bit.

9. The result of exponent is adjusted by added amount determined in step (7). If it was determined in step 7 part 3 that is $s=0$ then set directly exponent should be zero.

10. Convert all number into 32 bit standard format.

3.2.1 Special condition

Some special conditions are checked before processing for algorithms. If any condition is satisfied then we have no need to calculate the result by normal procedure. Results are directly calculated. So all the operations are bypassed, when any such condition is satisfied.

- 1) If operands $N_1=0$ and $N_2=0$ then result will be directly zero.
- 2) If $N_1=N_2$ and sign of $N_1 \neq$ sign of N_2 then result will be again zero.
- 3) If $N_1=0$ and $N_2 \neq 0$ then result will be equal to N_2 .
- 4) If $N_2=0$ and $N_1 \neq 0$ then result will be equal to N_1 .
- 5) If $d=|e_1 - e_2| > 24$ then result will be equal to larger of N_1 and N_2 .

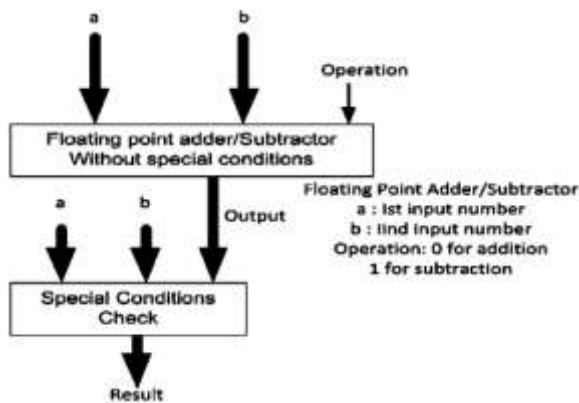


Figure 4: Architecture of Adder/Subtractor without Special Conditions

3.3 Adder implementation

The hardware implementation of the floating point adder and subtractor are outlined in figure 4. this outlined reflects the algorithms explained in section 3.2. Two points worth nothing are the hidden bit extraction and the re-assembly of the result into the 32-bit floating point format.

The implicit bit for each of the N_1 and N_2 operands must be explicit. Many of the times this will be a '1', then possibility of the number to become zero is neglected. From the IEEE standard format, the biased exponent and significand field are zero '0', then the also number representation is zero. so, in order to extract the correct bit, two 8 bit binary input is inserted in OR gates are used. If all bits of the exponent are zero '0', then directly number directly say zero '0' and there 24th bit will be zero or '1' is inserted.

Once get the result of the simple addition of two operand N_1 and N_2 is obtained, it will must be converted into the 32-bit standard floating point format. Now, that addition output feed into normalization unit and the results are shifted left until to the msb bit set to '1' at position 24th bit. After that we can say the result is normalized and the 24th bit directly replace by the sign of result. Also, the exponent of N_1 and N_2 is selected as the result of exponent must be adjusted by reflecting shifting that took place, although shifting amount of result is stored and it directly added exponent of N_1 i.e. e_1 to get the correct exponent result. Now at that stage, the final output or result is available in 32 bit standard IEEE format. it can be passed to the next operator and stored in memory.

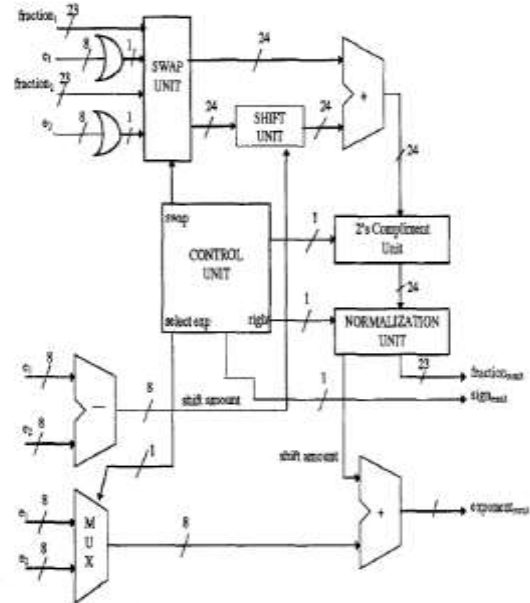


Figure 5: Floating Point Adder and subtractor Circuit

4. Results

4.1. Summary of Synthesis Report

4.1.1 HDL synthesis report

```

=====
HDL Synthesis Report
Macro Statistics
# Adders/Subtractors : 8
23-bit adder : 1
25-bit subtractor : 1
28-bit adder : 1
28-bit subtractor : 1
8-bit adder : 1
8-bit subtractor : 1
9-bit subtractor : 2
# Registers : 3
32-bit register : 3
# Comparators : 2
8-bit comparator equal : 1
8-bit comparator greater : 1
# Xors : 2
1-bit xor2 : 2
=====
    
```

4.1.2 Device utilization summary

Device utilization summary:

 Selected Device : 3s100evq100-5
 Number of Slices: 438 out of 960 45%
 Number of Slice Flip Flops: 96 out of 1920 5%
 Number of 4 input LUTs: 838 out of 1920 43%
 Number of IOs: 99
 Number of bonded IOBs: 99 out of 66 150% (*)
 Number of GCLKs: 1 out of 24 4%

4.1.3 Timing summary

Timing Summary:

 Speed Grade: -5
 Minimum period: 28.743ns (Maximum Frequency: 34.791MHz)
 Minimum input arrival time before clock: 23.158ns
 Maximum output required time after clock: 4.040ns
 Maximum combinational path delay: No path found

4.2 SIMULATION RESULTS

4.2.1 Addition

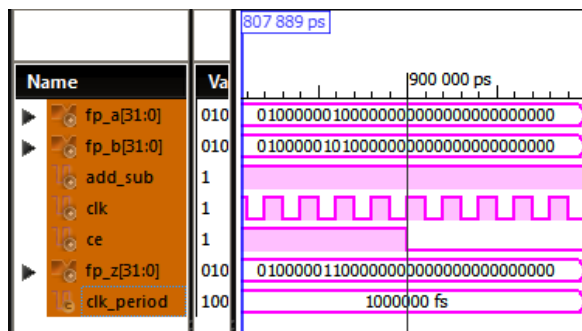


Figure 6: Waveforms generated while performing Addition.

Figure 6 shows the waveforms generated using model sim while performing addition. The detailed description of the given inputs and the output generated is given further.

4.2.1.1 Input operands

$$fp_a = 4 = (100)_2 = 1.00 * 2^2.$$

So, sign_{fp_a} = 0 (value is positive)

$$\text{Exponent}_{fp_a} = 2+127 = 129 = 1000\ 0001.$$

$$\text{Fraction}_{fp_a} = 000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_a (in base 2) = 0 1000 0001 000 0000 0000 0000 0000 0000.

$$fp_b = 12 = (1100)_2 = 1.10 * 2^3.$$

So, sign_{fp_b} = 0 (value is positive).

$$\text{Exponent}_{fp_b} = 3+127 = 130 = 1000\ 0010.$$

$$\text{Fraction}_{fp_b} = 100\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_b (in base 2) = 0 1000 0010 100 0000 0000 0000 0000 0000.

4.2.1.2 Output

$$\text{Output (in base 2)} = 16 = (10000)_2 = 1.0000 * 2^4$$

Sign_{fp_z} = 0 (value is positive).

$$\text{Exponent}_{fp_z} = 4+127 = 131 = 1000\ 0011.$$

$$\text{Fraction}_{fp_z} = 000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_z (in base 2) = 0 1000 0011 000 0000 0000 0000 0000 0000.

4.2.2 Subtraction

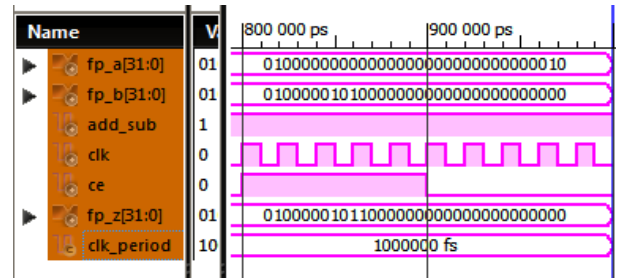


Figure 7: Waveforms generated while performing Subtraction.

Figure 7 shows the waveforms generated using model sim while performing subtraction. The detailed description of the given inputs and the output generated is given further.

4.2.2.1 Input operands

$$fp_a = -4 = (100)_2 = 1.00 * 2^2.$$

So, sign_{fp_a} = 1 (value is negative)

$$\text{Exponent}_{fp_a} = 2+127 = 129 = 1000\ 0001.$$

$$\text{Fraction}_{fp_a} = 000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_a (in base 2) = 1 1000 0001 000 0000 0000 0000 0000 0000.

$$fp_b = 12 = (1100)_2 = 1.10 * 2^3.$$

So, sign_{fp_b} = 0 (value is positive).

$$\text{Exponent}_{fp_b} = 3+127 = 130 = 1000\ 0010.$$

$$\text{Fraction}_{fp_b} = 100\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_b (in base 2) = 0 1000 0010 100 0000 0000 0000 0000 0000.

4.2.2.2 Output

$$\text{Output (in base 2)} = 8 = (1000)_2 = 1.000 * 2^3$$

Sign_{fp_z} = 0 (value is positive).

$$\text{Exponent}_{fp_z} = 3+127 = 130 = 1000\ 0010.$$

$$\text{Fraction}_{fp_z} = 000\ 0000\ 0000\ 0000\ 0000\ 0000.$$

Operand fp_z (in base 2) = 0 1000 0010 000 0000 0000 0000 0000 0000.

5 Conclusions

IEEE single precision floating point arithmetic is implemented on Spartan 3E using XILINX ISE. The architectures have been chosen according to the needs. For some operations where speed is critical, combinational architecture has been chosen and where area is critical sequential architecture has been implemented. We get delay 28.743ns and maximum frequency is 34.791MHz.

6. REFERENCES

- [1] R. V. K. Pillai, D. A1 - IShalili and A. J. A1 - Khalili, "A Low Power Approach to Floating Point Adder Design", in Proceedings of the 1997 International Conference on Computer Design, pp. 178-185.
- [2] L. A. Tawalbeh, "Radix-4 ASIC Design of a Scalable Montgomery Modular Multiplier using Encoding Techniques," M.S. Thesis, Oregon State University, USA, October 2002.
- [3] Dhiraj Sangwan & Mahesh K. Yadav, "Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic", in International Journal of Electronics Engineering, 2(1), 2010, pp. 197-203.
- [4] Loucas Louca, Todd A. Cook, William H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", In IEEE International Conference on Electronics, Circuits and Systems, 1996.
- [5] IEEE computer society: IEEE Standard 754 For binary floating-point arithmetic, 1985.
- [6] R. V. K. Pillai, D. A1 - IShalili and A. J. A1 - Khalili, "A Low Power Approach to Floating Point Adder Design", in Proceedings of the 1997 International Conference on Computer Design, pp. 178-185.
- [7] Ali Malik, Seok-Bum Ko, "A Study On The Floating Point Adder In FPGAS", IEEE CCECE/CCGEI, Ottawa, May 2006.
- [8] W. Kahan "IEEE Standard 754 for Binary Floating-Point Arithmetic," 1997.
- [9] Wakerly, John F., "Digital Design – Principles and Practices", Tata McGraw Hill Series.
- [10] M.P. Farmwald, "On the Design of High-Performance Digital Arithmetic Units," PhD thesis, Stanford Univ., Aug. 1981.
- [11] N.T. Quach and M.J. Flynn, "An Improved Floating Point Addition Algorithm," Technical Report CSL-TR-90-442, Stanford Univ., June 1990. (Available at <http://umunhum.stanford.edu/main.html>).