**Structure of Computer Systems Project**

**Floating Point Arithmetic Logic Unit**

**Addition and Subtraction**

**Popa Adriana**

**Group 30433**

**Table of Contents**

1. Introduction
   1. Context
   2. Specifications
   3. Objectives
2. Bibliographic Study
   1. Representation in floating point single precision
   2. Special values
   3. Addition:
   4. Subtraction:
   5. Special Operations
3. Analysis
4. Design
5. Implementation
6. Testing and validation
7. Conclusions
8. Bibliography
9. **Introduction**

**1.1 Context**

The purpose of this project is to design, implement and test an arithmetical logic unit capable of performing the operations of addition and subtraction on 32 bits. The difference between a normal arithmetic logic unit implementation consists in performing the operations on floating point numbers with single precision.

**1.2 Specifications**

The device will be implemented and simulated in Active-HDL, the IDE provided by Aldec and then programmed in a basys3 board. The representation on the board will be in a readable format, although it will internally operate on IEEE Standard representation of floating-point numbers and will implement the operations of addition and subtraction.

**1.3 Objectives**

Design and implement a way to introduce 2 numbers, store them in the floating-point representation, select the operation to be performed (addition or subtraction) and display the result of the operations. I must transform the decimal number into its floating-point equivalent, then the device can perform the operation based on the user’s choice, and after all the steps of the operation is done, the result should be shown on the 7 segment displays.

**2. Bibliographic study**

**2.1 Representation in floating point single precision**

The format of IEEE single-precision floating-point standard representation requires 23 fraction bits **F**, 8 [exponent bits](https://www.sciencedirect.com/topics/engineering/exponent-bit) **E**, and 1 sign bit **S**, with a total of 32 bits for each word.

* **F** is the [**mantissa**](https://www.sciencedirect.com/topics/computer-science/mantissa) in 2’s complement positive binary fraction represented from bit 0 to bit 22. The mantissa is within the normalized range limits between +1 and +2.
* The sign bit **S** is employed to indicate the **sign** of the number, where when S = 1 the number is negative, and when S = 0 the number is positive.
* The **exponent** **E**is in excess 127 form. The value of 127 is the offset from the 8-bit exponent range from 0 to 255, so that E-127 will have a range from −127 to +128.

Table

Description automatically generated

To convert a base-10 real number into an IEEE 754 binary32 format using the following outline:

* Consider a real number with an integer and a fraction
* Convert and normalize the integer part into binary
* Convert the fraction part: multiply the fraction by 2, take the integer part and repeat with the new fraction by 2 until a fraction of zero is found or until the precision limit is reached which is 23 fraction digits for IEEE 754 binary32 format.
* Add the two results and adjust them to produce a proper final conversion

To convert from binary32 to decimal, we identify F, S and E and use the formula:



**2.2 Special values**

IEEE has reserved some values that can ambiguity.

* Zero: is a special value denoted with an exponent and mantissa of 0. They are both equal but represented differently:

Text

Description automatically generated with low confidence

* Infinity: +infinity and -infinity are denoted with an exponent of all ones and a mantissa of all zeroes. The sign bit distinguishes between negative infinity and positive infinity. Operations with infinite values are well defined in IEEE.

Text

Description automatically generated with low confidence

* Not A Number (NAN): is used to represent a value that is an error. This is represented when exponent field is all ones with a zero-sign bit or a mantissa that is not 1 followed by zeroes. This is a special value that might be used to denote a variable that doesn’t yet hold a value.

Text

Description automatically generated with medium confidence

* Denormalized: If the exponent is all zeroes, but the mantissa is not then then value is a denormalized number. This means this number does not have an assumed leading one before the binary point.

**2.3Addition:**

X3 = X1 + X2  
X3 = (M1 x 2E1) + (M2 x 2E2)

1) X1 and X2 can only be added if the exponents are the same i.e., E1=E2.  
 2) We assume that X1 has the larger absolute value of the 2 numbers. Absolute value of X1 should be greater than absolute value of X2, else swap the values such that Abs(X1) is greater than Abs(X2).   
 3) Initial value of the exponent should be the larger of the 2 numbers, since we know exponent of X1 will be bigger, hence Initial exponent result E3 = E1.  
 4) Calculate the exponent's difference i.e., Exp\_diff = (E1-E2).  
 5) Left shift the decimal point of mantissa (M2) by the exponent difference. Now the exponents of both X1 and X2 are same.  
 6) Compute the sum of the mantissas

7) Normalize the resultant mantissa (M3) if needed and the initial exponent result E3=E1 needs to be adjusted according to the normalization of mantissa.  
 8) If any of the operands is infinity or if (E3>Emax) , overflow has occurred ,the output should be set to infinity. If (E3 < Emin) then it's an underflow and the output should be set to zero.  
 9) Nan's are not supported.

**2.4 Subtraction:**

X3 = X1 - X2  
X3 = (M1 x 2E1) - (M2 x 2E2)

1) X1 and X2 can only be added if the exponents are the same i.e., E1=E2.

2) We assume that X1 has the larger absolute value of the 2 numbers. Absolute value of X1 should be greater than absolute value of X2, else swap the values such that Abs(X1) is greater than Abs(X2).

3) Initial value of the exponent should be the larger of the 2 numbers, since we know exponent of X1 will be bigger, hence Initial exponent result E3 = E1.

4) Calculate the exponent's difference i.e., Exp\_diff = (E1-E2).

5) Left shift the decimal point of mantissa (M2) by the exponent difference. Now the exponents of both X1 and X2 are same.

6) Compute the difference of the mantissas

7) Normalize the resultant mantissa (M3) if needed and the initial exponent result E3=E1 needs to be adjusted according to the normalization of mantissa.

8) If any of the operands is infinity or if (E3>Emax), overflow has occurred, the output should be set to infinity. If (E3 < Emin) then it's an underflow and the output should be set to zero.

9) Nan's are not supported.

**2.5 Special Operations:**

Infinity + Infinity = Infinity

Infinity – -Infinity = Infinity

-Infinity - Infinity = -Infinity

-Infinity + -Infinity = Infinity

Nan == Nan => FALSE

**3.Analysis**

Floating point addition algorithm can be explained in two cases with example. Case I is when both the numbers are of same sign i.e., when both the numbers are either + or – means the MSB of both the numbers are either 1 or 0. Case II when both the numbers are of different sign i.e., when one number is + and other number is – means the MSB of one number is 1 and other is 0.

**Case I: -When both numbers are of same sign**

**Step 1:** Enter two numbers N1 and N2. E1, S1 and E1, S2 represent exponent and significand of N1 and N2.

**Step 2:** Is E1 or E2 =0‟. If yes set hidden bit of N1 or N2 is zero. If not, then check is E2 > E1 if yes swap N1 and N2 now contents of N2 in N1 and N1 in N2 and if E1 > E2 make contents of N1 and N2 same there is no need to swap.

**Step 3:** Calculate difference in exponents d=E1-E2. If d = „0‟ then there is no need of shifting the significand and if d is more than „0‟ say „y‟ then shift S2 to the right by an amount „y‟ and fill the left most bits by zero. Shifting is done with hidden bit.

**Step 4:** Amount of shifting i.e., „y‟ is added to exponent of N2 value. New exponent value of E2= previous E2 + „y‟. Now result is in normalize form because E1 = E2.

**Step 5:** Is N1 and N2 have different sign „no‟. In this case N1 and N2 have same sign.

**Step 6:** Add the significands of 24 bits each including hidden bit S=S1+S2.

**Step 7:** Is there is carry out in significand addition. If yes, then add „1‟ to the exponent value of either E1 or new E2 and shift the overall result of significand addition to the right by one by making MSB of S is „1‟ and dropping LSB of significand.

**Step 8:** If there is no carry out in step 6 then previous exponent is the real exponent.

**Step 9:** Sign of the result i.e., MSB = MSB of either N1 or N2.

**Step 10:** Assemble result into 32-bit format excluding 24th bit of significand i.e., hidden bit.

**Case II: - When both numbers are of different sign**

**Steps 1, 2, 3 & 4** are same as done in case I.

**Step 5:** Is N1 and N2 have different sign „Yes‟.

**Step 6:** Take 2‟s complement of S2 and then add it to S1 i.e., S=S1+2‟s complement of S2.

**Step 7:** Is there is carry out in significand addition. If yes, then discard the carry and shift the result to left until there is „1‟ in MSB also counts the amount of shifting say „z‟.

**Step 8:** Subtract „z‟ from exponent value either from E1 or E2. Now the original exponent is E1- „z‟. Also append the „z‟ amount of zeros at LSB.

**Step 9:** If there is no carry out in step 6 then MSB must be „1‟ and in this case simply replace „S‟ by 2‟s complement.

**Step 10:** Sign of the result i.e., MSB = Sign of the larger number either MSB of N1or it can be MSB of N2.

**Step 11:** Assemble result into 32-bit format excluding 24th bit of significand i.e., hidden bit.

Diagram

Description automatically generated

Diagram

Description automatically generated

**4.Design**

Diagram

Description automatically generated The black box view and block diagram of the single precision floating point Adder is shown in figures. The input operands are separated into their sign, mantissa and exponent components. This module has inputs opa and opb of 32-bit width. Before the operation Special cases are treated separately without Adder and some other resources. In addition to normal and subnormal numbers, infinity, NaN and zero are represented in IEEE 754 standard. Some possible combinations have a direct result, for example, if a zero and a normal number are introduced the output will be the normal number directly.

The **Block diagram** has two branches:

¬ The special cases one is quite simple because only the combination of the different exceptions are taken into account.

**Text, table

Description automatically generated**

Diagram, schematic

Description automatically generated¬ The second one that includes the main operation of the adder. The different operations that should be done are divided in three big blocks: preadder, adder and normalizing block.

The inputs to the n\_case block are A and B, and outputs are vector S and enable. Vector S contains the result when there is a special case as shown in table otherwise undefined. Finally, enable signal enables or disables the adder block if it is needed or not. If any normal or subnormal combination Occurs the enable signal is high, otherwise low.

**Pre-Adder block**

The first subblock is the Pre-Adder block. Various functions of the block are:

1. Distinguishing between normal, subnormal or mixed (normal-subnormal combination) numbers.

2. Treating the numbers in order to be added (or subtracted) in the adder block. · Setting the Output’s exponent · Shifting the mantissa · Standardizing the subnormal number in mixed numbers case to be treated as a normal case.

1. Normal numbers case

The procedure is as follows:

1. Making a comparison between both A and B numbers and obtaining the largest number

2. Obtaining the output exponent (the largest one)

3. Shifting the smallest mantissa to equal both exponents.

1. Subnormal numbers case

The operation using subnormal numbers is the easiest one. It is designed in just one block and the procedure is as follows:

1. Obtaining the two sign bits and both mantissas.

2. Making a comparison between both A and B numbers in order to acquire the largest number.

3. Fixing the result exponent in zero.

1. Mixed Numbers case

When there is a mixed combination of numbers (one subnormal and other normal) the subnormal one must have a special treatment in order to be added or subtracted to the normal one. The architecture of mixed numbers case is shown in figure. The comp block identifies the subnormal number and fixes it in NumberB\_aux. Then the number of zeros on the beginning of the subnormal number are counted for shifting the vector and new exponent is calculated.

Diagram

Description automatically generated

**Adder Block**

Diagram, engineering drawing

Description automatically generated Adder is the easiest part of the blocks. This block only implements the operation (addition or subtraction). It can be said the adder block is the ALU (Arithmetic Logic Unit) of the project because it oversees the arithmetic operations.

Two functions are implemented in this part of the code:

1. Obtaining the output’s sign

2. Implementing the desired operation.

A Carry Look Ahead structure has been implemented. This structure allows a faster addition than other structures. It improves by reducing the time required to determine carry bits.

This is achieved by calculating the carry bits before the sum which reduces the wait time to calculate the result of the large value bits. The code has been designed implementing a 1-bit CLA structure and generating the other components up to 28 (the number of bits of the adder) by the function generate.

**Standardizing Block**

Finally, the Standardizing Block takes the result of the addition/subtraction and gives it an IEEE 754 format.

The procedure is as follows:

1. Shifting the mantissa to standardize the result

2. Calculating the new exponent according with the addition/subtraction overflow (carry out bit) and the displacement of the mantissa.

**Round block**

Round block provides more accuracy to the design. Four bits at the end of the vector had been added in the Pre-Adder block. Now it is time to use these bits in order to round the result. The process to round is chosen arbitrarily: if the round bits are greater than the value “1000” the value of the mantissa will be incremented by one. Otherwise, the value keeps the same value. Apart from that a vector block is used to group the final sign, exponent and mantissa of the result. The final Result is Selected between the Special case output and operated output based on inputs.

**Diagram**

**Diagram

Description automatically generated**

**ALU** – The adder uses to ALU (small and big) one for the mantissa bits and the other for the exponent bits. The ALU is responsible for finding the bigger number such that the normalization can take place easily.

**Shifters** – The shifters are used to perform the normalization. Depending on the shift value generated by the Control unit, the Mantissa/Exponents are shifted accordingly.

**Control Unit** – The Control Unit uses inputs at various stages and generates the expected control values for shifters/ALUs.

**5. Implementation**

The implementation was done by dividing the problem into components the as follows:

* 1. **Cases:**

This component gets two numbers A and B. It checks whether one of them or both are 0, Infinity or NaN the Result will be directly calculated, and the enable will be set to 0, otherwise the enable is 1.

5.2 **Pre-Adder:**

The Pre-Adder uses 2 other components: CompareExponentBlock and ShiftRight. Its purpose is to prepare the numbers for add/subtract operation by calculating the maximum exponent, shifting the mantissa of the other, and getting the signs of each number. The mantissas will be extended on 28 bits for a more precise calculation.

5.2.1 **CompareExponentBlock:**

The role of this component is to break the numbers into sign, exponent and mantissa and return the signs, the maximum exponent, the difference of the exponents and the mantissas.

5.2.2 **ShiftRight:**

Its purpose is to shift the mantissa of the numbers with the small exponent by the difference of the exponents.

5.3 **Adder Block:**

The Adder is using multiple Carry Look Ahead Adders to compute the sum/difference of the mantissas, computes the final sign and returns the carry out of the operation.

Graphical user interface, text, application, email

Description automatically generated

5.4 **Standardizing Block:**

It uses the ShiftLeft and RoundBlock to compute the calculated carryout, sign, exponent and mantissa into a number according to the IEEE 754 Standard.

5.4.1 **ShiftLeft:**

Shift the mantissa of the result by the number of zeroes it has in front of the first 1 it contains.

5.4.2 **RoundBlock:**

Based on the last 4 bits of the mantissa it rounds the 5th bit, restoring the mantissa to its original length, 23 bits.

5.5 **FPAdder:**

Is the top component of the project. It maps the other components and then, based on the enable of the Cases components it returns the final result of the operation.

Additionally, for displaying the numbers on the Basys 3, I implemented a **ROM Memory** to store the numbers and choose them directly, since the board doesn’t have the possibility to read 32 bits. And also, a **Seven Segment Display** to display the first 16 bits of the result.

**6.Testing and validation**

To prove the correctness of my project I used the following examples:

* Special cases:

0 + 2.75 = 2.75

0 = 0b00000000000000000000000000000000

2.75 = 0b01000000001100000000000000000000 = 0x40300000Graphical user interface

Description automatically generated with medium confidence

Infinity – 2.75 = Infinity

A screenshot of a computer

Description automatically generated with medium confidence

Nan + 2.75 = NanA picture containing diagram

Description automatically generated

* Addition:

3.25 + 2.75 = 6

3.25 = 0x40500000

6 = 0x40C00000

A picture containing graphical user interface

Description automatically generated

3.25 + 3.5 = 6.75

3.5 = 40600000

6.75 = 0x40D80000

Graphical user interface

Description automatically generated with medium confidence

* Subtraction:

3.5 – 3.5 = 0

Graphical user interface

Description automatically generated

3.5 – 3.25 = 0.25

0.25 = 0x3E800000

Graphical user interface

Description automatically generated

**7.Conclusions**

The goal of this project was to design, implement and test 2 operations: addition and subtraction on the standard IEEE format on 32 bits. The difference from the usual implementations is that this unit will perform them on floating point numbers with single precision.

This project was a challenge due to the differences between normal operations and operations in floating point because I had to understand the floating-point representation very well to design and implement the operations. I enjoyed working on this project because it was an opportunity to test my logic and the knowledge of the VHDL language and computer systems. Finally, from all the papers I’ve read and the examples I’ve seen I’ve come up with my own implementation for the floating-point addition and subtraction, managing to obtain correct solutions to my operations.

**8.Bibliography**

1. <https://en.wikipedia.org/wiki/Single-precision_floating-point_format#Converting_from_single-precision_binary_to_decimal>
2. <https://www.sciencedirect.com/topics/computer-science/single-precision-format>
3. <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>
4. <https://en.wikipedia.org/wiki/Floating-point_arithmetic#Addition_and_subtraction>
5. <https://www.ijert.org/design-of-32-bit-floating-point-addition-and-subtraction-units-based-on-ieee-754-standard>
6. <https://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html>
7. <https://www.ijert.org/research/design-of-high-performance-ieee-754-single-precision-32-bit-floating-point-adder-using-vhdl-IJERTV2IS70837.pdf>