

**Correctly Rounded Floating-point Binary-to-Decimal  
and Decimal-to-Binary Conversion Routines in  
Standard ML**

By

**Prashanth Tilleti**

Advisor

Dr. Matthew Fluet

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology

Rochester, New York

December 2013

## Table of Contents

ABSTRACT .....	3
1. INTRODUCTION .....	4
1.1. NUMBER SYSTEMS .....	4
1.2. ROUNDING ERRORS .....	5
1.3. INPUT AND OUTPUT PROBLEMS .....	6
1.4. PROBLEM STATEMENT .....	8
2. BACKGROUND .....	10
2.1. IEEE STANDARD FOR FLOATING POINT NUMBERS .....	10
2.2. OUTPUT PROBLEM .....	12
2.3. INPUT PROBLEM .....	13
3. SOLUTION .....	15
3.1. INPUT PROBLEM .....	15
3.1.1. <i>AlgorithmM</i> .....	15
3.1.2. <i>AlgorithmR</i> .....	16
3.1.3. <i>Bellerophon</i> .....	16
3.2. OUTPUT PROBLEM .....	17
3.2.1. FP3 .....	17
3.2.2. IP2 .....	18
3.2.3. FPP2 .....	18
4. IMPLEMENTATION .....	19
5. RESULTS .....	20
6. CONCLUSION .....	21
7. FUTURE WORK .....	22
8. REFERENCES .....	23

## ABSTRACT

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a widely accepted and used representation for performing floating-point computations. This standard uses binary numbers for floating-point representation, whereas many applications use decimal numbers for this. The decimal-to-binary conversion problem takes a decimal floating-point representation and returns the best binary floating-point representation of that number. The binary-to-decimal conversion problem takes a binary floating-point representation and returns a decimal floating-point representation using the smallest number of digits that allow the decimal floating-point representation to be converted to the original binary floating-point representation.

MLton is an open-source, batch compiler for the Standard ML programming language. It makes use of the *gdt* library by David Gay for floating-point computations. While this is an excellent library, it generalizes the decimal-to-binary and binary-to-decimal conversion routines beyond what is required by the Standard ML Basis Library and induces an external dependency on the compiler. Native implementations of these conversion routines in Standard ML would obviate the dependency on the *gdt* library, while also being able to take advantage of Standard ML features in the implementation. The goal of this project is to develop a native implementation of the conversion routines in Standard ML.

## 1. INTRODUCTION

### 1.1. Number Systems

There are various number systems that can be used to represent non-integral numbers. Examples include floating-point, fixed-point, binary-coded decimal and logarithmic numbers systems. Floating-point and fixed-point number systems have been very popular, though floating-point is widely used in computer systems. Fixed-point system has a fixed number of digits before and after the radix point i.e. the number of digits and the position of the radix point are fixed. In floating-point system, the number of digits in the number is fixed, but not the position of the radix point (radix point ‘floats’). The main advantage of using floating-point system over fixed-point system is that a wider range of numbers can be represented using the same amount of storage.

For example, let us define a fixed-point numbering system having 4 digits, 3 before the radix point and 1 after the radix point. Let us also define a simplified floating-point numbering system having 4 digits (assume that we have only the significand part and no exponent). Consider the set of decimal (base 10) numbers. In fixed-point system, each digit can be anything in between 0 to 9 except the first digit, which clearly cannot be zero. Also, the number can be either positive or negative. This means that we can represent  $9 \times 10 \times 10 \times 10 \times 2 = 18000$  numbers ranging in  $[100, 1000)$ . In floating-point system, the radix point can be anywhere in the number, which means it has 5 possible positions. This means that we can represent  $9 \times 10 \times 10 \times 10 \times 2 \times 5 = 90000$  numbers. From this example, we can see that floating-point system can represent a wider range of numbers using the same amount of storage.

An actual floating point system has both significand and exponent parts allowing us to represent a wider range of numbers. In general any floating-point number can be represented using the following format –

$$f = \pm f_0.f_1f_2f_3 \dots f_{p-1} \times \beta^e, \quad 0 \leq f_i < \beta, f_0 \neq 0 \quad (1)$$

where  $p$  denotes the precision,  $\beta$  denotes the radix base and  $e$  denotes the exponent of the floating-point number  $f$ . The non-integral part of the number is called the significand ( $f_0.f_1f_2f_3 \dots f_{p-1}$ ). There are two more variables that have to be considered while representing numbers using floating-point system -  $e_{min}$  and  $e_{max}$ , i.e. the minimum and maximum allowable exponents, respectively. The value of this floating-point number can be represented as -

$$f = \sum_{i=-(p-1)}^0 f_i \times \beta^i \times \beta^e \quad (2)$$

Using the notation shown in (1), we cannot represent zero. In floating-point system, zero is considered a special case and is represented as  $1.0 \times \beta^{e_{min}-1}$ . IEEE standard for floating-point representation, a widely used and accepted format, defines how special cases like zero and infinity are handled. More details about this format are given in the next section.

Another point to note here is that the numbers in fixed-point system are evenly spaced i.e. the difference between any two consecutive numbers is always the same, whereas the numbers in floating-point system are unevenly spaced. According to Allison, the difference between any two consecutive floating-point numbers in the range of  $[\beta^e, \beta^{e+1})$  has a constant value of  $\beta^{1-p+e}$ .

The floating-point number shown in (1) is said to be having a *normalized scientific notation*. Any floating-point representation having single non-zero digit before the radix point is considered as normalized representation. If the float has a single digit before the radix-point and if that digit is zero, then it is said to be in *denormalized* form.

## 1.2. Rounding Errors

Since the floating-point numbers have a fixed number of digits (called *precision*), rounding needs to be done when we intend to represent a number that has more digits. Any form of rounding always leads to a rounding error. For example, representing the value of 1.5342 using 4 digits of precision yields 1.534, resulting in a rounding error of 0.0002. There are three different types of metrics used for estimating these errors.

- *Absolute error*: It represents the exact error that results due to rounding. It is calculated as the absolute value of the difference between the actual number and the floating-point number. This can be formulated as –

$$absolute\ error = |float - original| \quad (3)$$

For example, the absolute error in representing 1.5342 using 4-digit precision is  $(1.534 - 1.5342) = 0.0002$ .

- *Relative error*: It represents the amount of error that occurred relative to the actual number. This can be formulated as –

$$relative\ error = \frac{|float - original|}{original} \quad (4)$$

For example, the relative error in representing 1.5342 using 4-digit precision is  $|1.534 - 1.5342|/1.5342 \approx 0.0001$ .

- *Ulp*s: It is the acronym for ‘Units in the last place’. It represents the number of units by which the two numbers differ in their last place. According to Goldberg, making use of the floating-point number defined in (1), calculation of ulps can be formulated as –

$$ulps = |f - (original/\beta^e)| \times \beta^{p-1} \quad (5)$$

For example, the value of ‘units in last place’ in representing 1.5342 is  $|1.534 - (1.5342/10^0)| \times 10^{4-1} = 0.2$ .

Relative error and ulps are the most widely used metrics for estimating the rounding errors. Muller et al. [8] have calculated the relative errors obtained in various cases and presented them in their paper. A lot of research has been done on the effectiveness and usage of ulps, which has been neatly summarized by Muller et al. in their paper.

When a floating-point operation is performed on two floating-point numbers, there is always a possibility for the rounding error to increase. For example, consider two numbers 1.5342 and 6.0438. These numbers when represented as floating-point numbers with 4-digit precision, have the values 1.534 and 6.044 respectively. The rounding error in representing these numbers is 0.2ulps each. When these numbers are subtracted from one another, we get 4.5096 (original numbers) and 4.510 (floating-point numbers). The rounding error associated with the difference is 0.4ulps. From this example, we can see that there is a chance for rounding error to increase when performing floating-point operations. According to Goldberg [3] and Clinger [2], floating-point multiplication and division are sufficiently accurate i.e. the rounding error is reasonable and within limits. But, floating-point addition and subtraction have a high chance of yielding larger rounding errors, especially if the numbers involved in the operation are separated by a large factor. Clinger’s algorithm works on the assumption that floating-point multiplication is sufficiently accurate.

According to Goldberg [3], this can be avoided or reduced using two methods. First method is to use guard digits i.e. use an extra bit of precision than required while doing the floating-point arithmetic. Second method is to perform the floating-point operation on the exact values and then round the result. This second method is called *exact rounding* and is most widely used. Goldberg has provided a few theorems in his paper, with proofs to explain these concepts.

### 1.3. Input and Output Problems

The representation of numbers in a computer is different from what humans use mostly. Computers usually use binary numbers for *internal representation* and decimal numbers for *external representation*. External decimal representation of numbers is helpful in two scenarios viz., human-machine communication and inter-machine communication. Based

on these two different representations, we can come up with two problem definitions – *input problem* (converting from external representation to internal representation) and *output problem* (converting from internal representation to external representation).

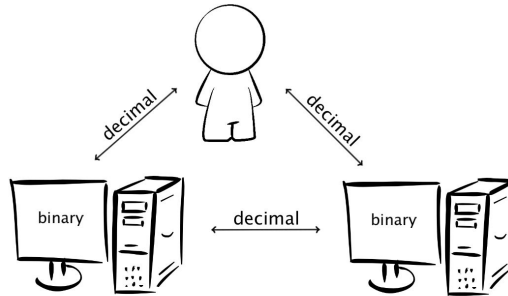


Figure 1: Numbers as seen by humans and machines

Internal representation follows IEEE notation standards for floating-point numbers. This means that the internal representation has a fixed number of digits. External representation is not governed by any such standards and therefore has two different formats that it can be expressed in – free format and fixed format. *Fixed-format* numbers have fixed number of digits. If the actual number has fewer digits than required, then zeroes are appended to it to achieve the required number of digits. Of course, the digits are appended in such a way that the value remains the same – trailing zeroes if the number has a fractional part or leading zeroes if the number has only integer part. The number of digits in *free-format* numbers is not fixed and is sufficient enough to represent the value accurately i.e. no number with fewer digits can gain this accuracy. From this discussion, it can be seen clearly that the input and output problems are not exactly same because the internal representation has fixed number of digits whereas the external representation may have variable number of digits. This is the reason why there are separate algorithms for each of these two problems (Steele and White – output problem, Clinger – input problem).

Since the internal representation has a fixed number of digits, because of the IEEE notation being followed, any floating-point number having more digits is rounded to these fixed digits. This means that a lot of decimal numbers when converted to internal representation convert to the same floating-point number. For example, the decimal numbers 53.413, 53.41299819946, and 53.41298 translate to the floating-point number 01000010010101011010011001101001. From this we can say that, for every given number in internal representation there exist an infinite possible numbers in external representation i.e. a many-to-one mapping exists in between external and internal representations.

The conversion methods i.e. input and output problems are expected to follow certain requirements – internal identity requirement and external identity requirement. *Internal identity requirement* says that the conversion from internal representation to external representation and back should be an identity function. *External identity requirement* says that the conversion from external representation to internal representation and back should be an identity function. These requirements can be formulated as –

$$d2f(f2d(float)) = float \quad (6)$$

$$f2d(d2f(decimal)) = decimal \quad (7)$$

There are a few problems with these identity requirements, the first problem being that it is not always possible to have an exact representation of a number in all radices. The second problem concerns with choosing the decimal number for a given floating-point number, keeping in mind the many-to-one mapping discussed above.

The identity requirements are valid for finite integers, but they do not hold always for finite fractions. The reason is that a finite fraction in one radix may have an infinite representation in another radix. This generally happens for the fractions of the form  $1/p$  where  $p$  is a prime number that divides one of the radices and not the other. For example, 5 is a prime number that divides radix 10, but not radix 2. This means that the fraction  $1/5$  must have a finite representation in decimal form, but an infinite representation in the binary form. True to this deduction, the finite decimal fraction 0.2 when converted to binary gives the infinite repeating fraction  $0.\overline{0011}$ .

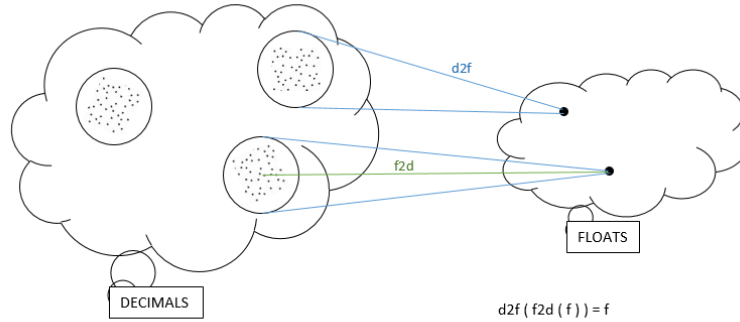


Figure 2: Mapping of decimal numbers to floating-point numbers

#### 1.4. Problem Statement

MLton [6] is an open-source, batch compiler for the Standard ML programming language. It makes use of the *gdtol* library by David Gay for floating-point computations. This library is implemented in C programming language. Even though this library is very helpful, it has a lot unnecessary generalization, which is not required by



Standard ML Basis Library [7]. Since this is an external library, there is always the problem of updating the MLton source whenever the *gdtoc* library has changes. Moreover, since this library is not a native implementation, whenever any floating-point computation is made, the compiler has to take care of interpreting the C code into SML. This introduces an external dependency on the compiler. Having a native implementation of these libraries would eliminate this dependency.

MLton has an in-built structure (*IntInf*) that can handle high precision integer arithmetic. This structure will be very helpful in developing the conversion routines. The goal of this project is to make use of the existing algorithms and MLton structures to develop a native implementation of these conversion routines.

The next sections in this document are organized in the following way. *Background* gives a brief overview of the existing algorithms that will be used in this project. This section also gives a summary of the IEEE standards followed for representing floating-point numbers in computers. *Method* discusses these algorithms in detail and also defines how the layout of this project. The implementation details and the results are presented in the latter sections, along with the conclusion and future work.

## 2. BACKGROUND

### 2.1. IEEE Standard for Floating-point numbers (IEEE 754)

Inter-machine communication becomes easier if there exist some well-defined standards for representing floating-point numbers. IEEE has defined a standard representation for floating-point numbers, which is followed by most of the modern day computers. Computers store numbers internally in binary format. The IEEE-754 specification defines two standards for floating-point representation – single precision and double precision formats. Single precision uses 32 bits for storing the floats and double precision uses 64 bits for doing so. The digits (or, bits) of the floating-point number are saved in a (*sign*, *exponent*, *significand*) format as shown in the figure below.

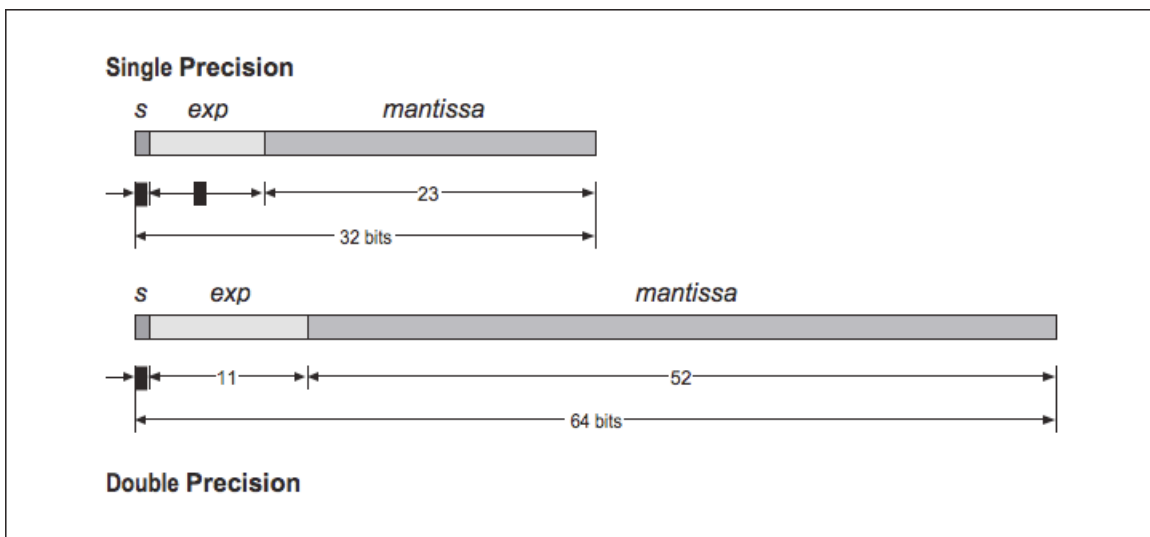


Figure 3: IEEE format for floating-point numbers

As seen in the figure, the first bit is given to the sign of the number – 0 for positive numbers and 1 for non-positive numbers. Exponent and significand (or, *mantissa*) occupy the remaining bits as shown in the figure.

Floating-point numbers follow normalized scientific notation i.e. the integer part of the significand is always a single-digit non-zero number. Since we are talking about binary floating-point numbers, normalized notation means that the integer part of the significand will be always 1. Based on this fact, IEEE format does not store this particular bit to save space. Only the fractional part of the significand is stored using this format. Though we use only 23 bits for the significand, the precision of the IEEE single float is 24 because of the extra hidden bit i.e. 1.

The exponent can also have a sign. IEEE format uses a biased exponent, eliminating the need to save the sign of the exponent and thus saving an extra bit of space. A biased exponent is the value that results after adding a *bias* value to the actual exponent. The

value of the bias is 127 for single precision and 1023 for double precision. The range of exponents that are representable using IEEE single precision format is from -126 to +127, which means the range of biased exponents is from 0 to 255, which require no more than 8 bits for storage.

The table below shows a list of the values of all the parameters for IEEE floating-point formats.

Parameter	Format	
	Single Precision	Double Precision
<b><math>p</math></b>	24	53
<b><math>e_{min}</math></b>	+127	+1023
<b><math>e_{max}</math></b>	-126	-1022
<b><math>bias</math></b>	127	1023
<b># of bits in sign</b>	1	1
<b># of bits in exponent</b>	8	11
<b># of bits in significand</b>	23	52
<b># of total bits</b>	32	64

Table 1: IEEE 754 format parameters

IEEE format also defines the notation to use for special numbers, which is summarized in the table below. Zero is considered a special case because it cannot be represented using the normalized scientific notation (discussed in previous section). Zeroes represented in this format have a sign, but care has been taken to make sure that the values of positive and negative zeroes are considered equal. Infinity is treated as a special case in order to handle overflow and underflow smoothly. *Overflow* occurs when we come across a number whose value is larger than the maximum value that can be represented using the floating-point format. *Underflow* occurs when we come across a number whose value is smaller than the minimum value that can be represented using the floating-point format. In case of overflow, IEEE gives it a value of positive infinity in order to make sure that the computation doesn't halt midway. Similarly, a value of negative infinity is given when an underflow occurs. Another special case that has been considered is NaN (Not a Number), which occurs when we come across numbers that are not valid (for example,  $\sqrt{x}$ , when  $x < 0$ ). A special representation has been specified to represent denormalized numbers in addition to the normalized scientific notation. Having a way to represent denormalized numbers helps in achieving *gradual underflow* rather than a *flush to zero*. (Add a reference here).

IEEE standard also describes how to handle exceptions like *divide by zero*, *invalid number*, etc. Goldberg has given a detailed explanation with examples, about each of the special cases and exceptions in his paper.

Special value	Exponent	Fraction
$\pm 0$	$e = e_{min} - 1$	$f = 0$
$0, f \times 2^{e_{min}}$	$e = e_{min} - 1$	$f \neq 0$
$1, f \times 2^e$	$e_{min} \leq e \leq e_{max}$	–
$\pm \infty$	$e = e_{max} + 1$	$f = 0$
NaN	$e = e_{max} + 1$	$f \neq 0$

Table 2: IEEE 754 special cases

Any floating-point computation leads to rounding. This occurs if the result of the computation is *inexact* i.e. has more digits than that can be representable. IEEE defines 5 different rounding modes – 2 are round to nearest and other 3 are direct roundings. A brief description of each of these rounding modes is given below. The table below shows examples of each rounding mode.

- *Round to nearest – even*: rounds to the nearest value; in case of a tie, rounded to the nearest value that has an even digit in the least significant position. This is the most widely used rounding mode, especially in floating-point representation.
- *Round to nearest – away from zero*: rounds to the nearest value; in case of a tie, rounds to the nearest value away from zero.
- *Round towards zero*: equivalent to truncation.
- *Round towards positive infinity*: equivalent to ceil function in many languages.
- *Round towards negative infinity*: equivalent to floor function in many languages.

Rounding mode	+8.5	+9.5	-8.5	-9.5
to nearest – even	+8.0	+10.0	-8.0	-10.0
to nearest – away from zero	+9.0	+10.0	-9.0	-10.0
to zero	+8.0	+9.0	-8.0	-9.0
to positive infinity	+9.0	+10.0	-8.0	-9.0
to negative infinity	+8.0	+9.0	-9.0	-10.0

Table 3: IEEE Rounding Modes

## 2.2. Output problem [1]

Output problem deals with the conversion of a number from internal representation to external representation; specifically radix 2 number to radix 10 number. There is no prime that divides 2 and does not divide 10. This means that every number in internal representation can be converted into an exact number in external representation. Also, because of the many-to-one mapping between floating-point numbers and decimal numbers, the decimal representation of a given float can have infinite possibilities. But all of these possible decimals when converted back to internal representation give back the

same float. From this we can say that internal identity requirement always holds for the output problem. Following a similar argument, we can say that the external identity requirement cannot be strictly satisfied. So instead, we should be able to guarantee an *external consistency requirement* i.e. the output problem prints the same decimal every time. This way, we can ensure that the external identity requirement is conditionally valid.

Now, the problem that remains is to decide which decimal to print after converting a number from internal to external representation; in other words how many digits have to be printed out. Steele and White suggest printing enough digits to preserve the information contained in the binary number, which clearly depends on the precision and value of the float. In other words, this means that we chose the decimal that is closest in value to the given floating-point number and also is as short as possible. In case there are more than one numbers that satisfy the required criteria i.e. if there is a tie, it happens only due to the difference in the last digit. Steele and White have taken care of this in their algorithm described in the next section.

Steele and White have given algorithms for accurate conversion of floating-point numbers to decimal representation. They have achieved this in four steps. Firstly, they gave an algorithm to print fixed format fraction output. Secondly, they modify this algorithm to get another algorithm for printing integer output. Thirdly, they combine these two algorithms and give an algorithm that prints out free format output. Finally, they propose the Dragon4 algorithm, which is generalized to print out either fixed format or free format output based on the requirement. Note that all the output algorithms print out one digit at a time rather than printing the whole number at once. This project only deals with the first 3 algorithms (FP3, IP2, FPP2) and not Dragon4.

### **2.3. Input Problem [2]**

As mentioned earlier, input problem can be defined as the method to convert a decimal number to its corresponding floating-point representation. The work by Clinger has been a major contribution to this particular problem. He has given an algorithm that is very efficient in finding the best binary approximation to a decimal number. The term ‘approximation’ is used here because there is some rounding involved in this conversion process, in most of the cases. The rounding occurs due to the fact that the decimal representation of a number need not have fixed number of digits like the floating-point representation. IEEE standards specify that the error that occurs due to this rounding should not exceed 0.97ulps, but according to Clinger, finding the best approximation would yield a rounding error no more than 0.5ulps.

This algorithm works on the basic premise that the floating-point operations of multiplication and division do not induce a significant rounding error. Clinger has proved

this fact for floating-point multiplication operation, in his paper. His algorithm uses extra bits of precision than required by the floating-point format to compute the binary approximation to a given decimal. This results in a floating-point number that has more accuracy than required. The extra bits in the least significant portion of the number are checked to find the error that would be caused after rounding it to lower precision. Based on this error, the algorithm proceeds on to either round off the approximation or to extend the precision more. Clinger specifically deals with positive floating-point numbers that use round to nearest (tie to even) rounding mode.

The solution to the input problem is given as a 3-step process. Firstly, a simple algorithm called *AlgorithmM* is provided that uses infinite precision integer arithmetic to compute the binary approximation of a decimal number. This algorithm uses too much high precision that is not required all the time. Clinger suggests that this algorithm be used only for computing the binary values of numbers that are not representable by the floating-point system i.e. in the case of overflow or underflow. Secondly, an iterative algorithm called *AlgorithmR* is provided that takes a starting approximation and tries to converge this number to the actual floating-point number and gives us the best approximation that it can find. The process of convergence depends on the initial approximation given and hence this algorithm might be very slow if the initial float is not too close to the required float. Clinger suggests using this algorithm only when the starting approximation is good. Lastly, a non-iterative algorithm called *Bellerophon* is provided which uses extra bits of precision than required to do floating-point computations in calculating the best approximation. This algorithm does various checks and estimates the amount of rounding error that would result in rounding the high precision float to the required precision. If this estimate falls within defined bounds, then the rounding is done, else the high precision float is sent to *AlgorithmR* as a starting approximation. Clinger argues that *AlgorithmR* would require only one pass to complete its part, thus making *Bellerophon* an efficient algorithm. A detailed explanation of each of these algorithms is given in the next section.

### 3. SOLUTION

This project aims at developing native implementation of the conversion routines in Standard ML programming language. The algorithms presented by Clinger, Steele and White will be implementing in SML using MLton compiler. The implementation of these algorithms will be compared with the existing implementations that use David Gay's libraries. A detailed explanation of the algorithms is given below.

#### 3.1. Input Problem

##### 3.1.1. AlgorithmM

This algorithm uses integer arithmetic having unlimited precision. It takes exact decimal integers  $f$  and  $e$  as inputs, with  $f$  being non-negative and returns the floating-point number that is closest in value to  $f \times 10^e$ . The figure below summarizes the algorithm.

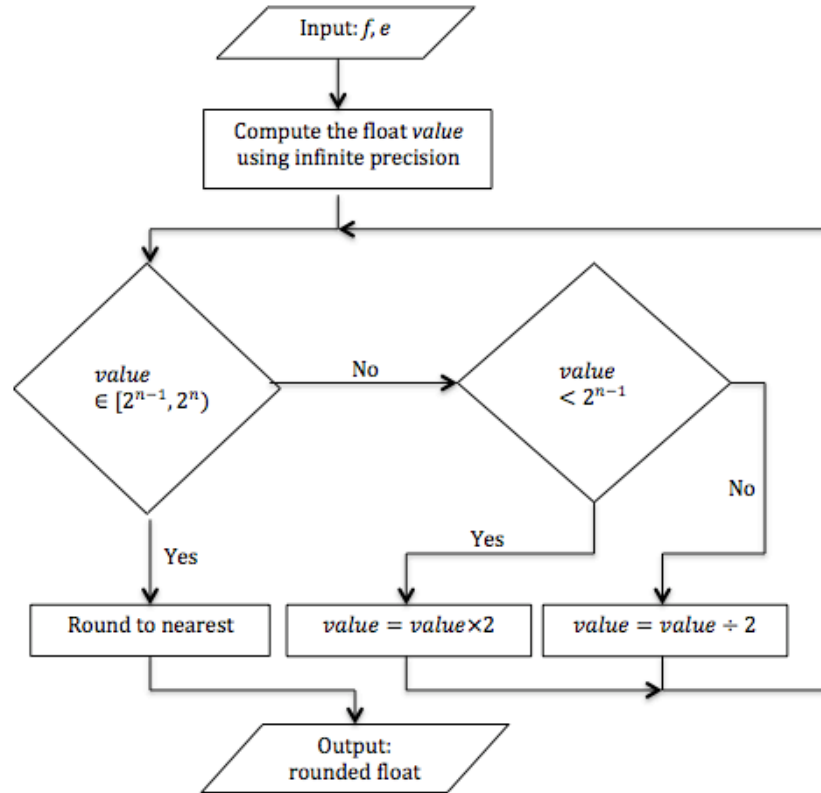


Figure 4: Flowchart representing AlgorithmM

In the above figure,  $n$  is the precision required to represent the floating-point numbers. The float value of the given decimal is computed using infinite precision and if this value falls in the range  $[2^{n-1}, 2^n)$ , then an IEEE float is constructed using this value as significand and the number of loops as the exponent. If the value is less than  $2^{n-1}$ , then it is multiplied by 2 until it falls in the range. If the value is larger than  $2^n$ , then it is divided

by 2 until it falls in the range. This algorithm uses a lot of high-precision integer arithmetic and hence is not suggestible.

### 3.1.2. *AlgorithmR*

This algorithm takes a starting floating-point approximation and tries to converge it to the actual float-value to obtain the best approximation for the given decimal number. The algorithm defines error bounds and checks using high precision arithmetic if the starting approximation lies within the defined bounds. If not, it takes the next or previous floating-point number and does a comparison until the approximation falls within the limits. The working of this algorithm is explained below.

Let us consider  $f$  and  $e$  to be the significand and exponent of the input decimal number  $d$  whose value is  $f \times 10^e$ . Let  $z = m \times 2^q$  denote the starting approximation and  $(m + \varepsilon) \times 2^q$  denote the actual float value of  $d$  i.e.  $d = f \times 10^e = (m + \varepsilon) \times 2^q$ . The distance between the actual float and the starting approximation is defined as  $\varepsilon$ . AlgorithmR tries to get  $z$  as close as possible to  $(m + \varepsilon) \times 2^q$  i.e. it tries to reduce the value of  $\varepsilon$  as much as possible to get the best approximation to  $d$ . High precision integer arithmetic is used to check the difference between these two numbers and if  $\varepsilon$  is found to be large, then  $z$  is assigned the next or previous representable float, until  $\varepsilon$  reaches an acceptable value.

This is achieved by computing positive integers  $x$  and  $y$  such that -

$$\frac{x}{y} = \frac{f \times 10^e}{m \times 2^q} \quad (8)$$

Substituting  $(m + \varepsilon) \times 2^q$  in place of  $f \times 10^e$ , we can compute the value of  $\varepsilon$  as -

$$\varepsilon = \frac{m(x-y)}{y} \quad (9)$$

The value of  $|\varepsilon|$  is then compared to  $1/2$  to decide the closeness of the number. The comparison with  $1/2$  comes from the fact mentioned in the previous section, that the rounding error cannot exceed 0.5ulps. Another important point to note here is that the algorithm makes sure that no actual division occurs in computing the value of  $\varepsilon$ .

### 3.1.3. *Bellerophon*

This algorithm uses extra bits of precision to perform floating-point multiplication in order to make sure that the product when rounded to  $n$ -bits gives the best approximation possible (or the next best) to the input decimal number. The working of this algorithm is explained below.

Let us consider  $f$  and  $e$  to be the significand and exponent of the input decimal number  $d$  whose value is  $f \times 10^e$ . Let  $n$  denote the precision required for representing a floating-



point number (24 for single precision and 53 for double precision) and  $p$  denote the extended precision that is needed for this algorithm ( $p \geq n + 4$ ). If  $f$  and  $10^e$  require only  $n$ -bits to be represented in floating-point format, then the product of those two gives the best floating-point approximation to  $d$ . If the exponent  $e$  is negative, and if  $f$  and  $10^{-e}$  require  $n$ -bits to be represented in floating-point format, then the quotient of these two gives the best approximation needed. Note that the product and quotient here are the results of  $n$ -bit multiplication and  $n$ -bit division respectively. These two floating-point operations are assumed to be sufficiently accurate.

If  $f$  and  $10^e$  (or,  $10^{-e}$  when  $e < 0$ ) are not representable using  $n$ -bits, then their floating-point values are computed using  $p$ -bits of precision. The product (or, quotient) of these two resulting floats is computed by performing a  $p$ -bit floating-point multiplication. This product may have some rounding error due to the rounding done for the  $p$ -bit conversion of  $f$  and  $10^e$ . Let us denote this error by  $\varepsilon$ . The computed product can be formulated as –

$$d = f \times 10^e = (z + \varepsilon) \times 2^q \quad (10)$$

where  $z$  and  $q$  represent the significand and exponent of the floating-point number. If  $z$  lies in  $\varepsilon$ -neighborhood of the midpoint of any two  $n$ -bit floating-point numbers, then it is rounded to  $n$ -bits precision to get the best  $n$ -bit approximation to  $d$ . If not, then  $f, e, z$  are passed to AlgorithmR (*failover* case). Since  $z$  is still the closest approximation to  $d$  (though, not the best), AlgorithmR should converge very quickly by using it as the starting approximation. Clinger argues that AlgorithmR requires only one pass to converge in case of a failover.

Instead of computing the value of  $10^e$ , Clinger suggests maintaining a lookup table of powers of 10 i.e. the floating-point values of all the allowable powers of 10 using  $p$ -bit precision (allowable means all those that can be represented using the floating-point number system). Since the range of allowable exponents is very large and requires lot of storage, he suggests that only the smaller exponents be saved using their exact values in a table. He also suggests maintaining a second table for higher exponents, but only those exponents that are multiples of the maximum exponent from the first table. For example, if the first table saves the floating-point values of the numbers from  $10^0$  to  $10^9$ , then the second table should save the floating-point values for  $10^{18}, 10^{27}, 10^{36}, \dots$  so on. Any intermediate powers can be computed as the product of powers from both the tables.

## 3.2. Output Problem

### 3.2.1. Finite-Precision Fixed-Point Fraction Printout (FP3)

FP3 takes a floating-point fraction as input and returns the corresponding decimal fraction as output. As discussed in previous section, the main goal of any output problem

would be to figure out how when to stop printing the digits. This algorithm defines some error bounds and then prints out the decimal digits until the error limits are not exceeded.

This algorithm works on the concept that the maximum possible error for a fraction of limited precision is equal to 0.5ulps. Since we are dealing with binary numbers, the units in the last place can be measured as being equal to  $2^{-n}$ , where  $n$  denotes the precision of the floating-point representation used for the binary number. This means that maximum possible error is  $0.5 \times 2^{-n} = 2^{-n-1}$ . This algorithm initializes a variable to this value and checks that the remaining fraction (after printing out a digit) is still within the error. The printing stops when the conditions are not met. Rounding is done on the last digit.

### *3.2.2. Indefinite Precision Integer Printout (IP2)*

IP2 uses a lot of high precision integer arithmetic. It basically works on the same concept as FP3 – prints the digits as long as the error conditions are not met. The only difference is that in FP3, since we were dealing with fractions, the residual fraction was always multiplied by a factor of 10 to get the next digit, whereas in IP2, since we are dealing with integers we divide by a factor of 10 to get the digits.

### *3.2.3. Fixed-Precision Positive Floating-Point Printout (FPP2)*

This algorithm is just a combination of FP3 and IP2. It uses FP3 for printing out digits from the fractional portion and IP2 for printing out digits from the integer portion of the given number.

## 4. IMPLEMENTATION

Each of the algorithms explained in the previous section are implemented using Standard ML. The implementation of the binary-to-decimal problem required a data structure that can handle rational numbers. Standard ML Basis Library has structures for integers and real numbers, but not rational numbers. Prof. Fluet helped me in defining a new structure for rational numbers that was very helpful in implementing the FPP2 algorithm. This project has been implemented only for single precision floating point numbers using round to nearest (tie to even) rounding mode. The algorithms are implemented as they are, without any changes.

The implementation of the Bellerophon algorithm takes the significand and exponent of the decimal number as input. It computes the floating-point approximation of that decimal number. A string representation of the decimal value of this approximation is returned back by the function. The implementation of FPP2 algorithm takes the list of digits in the binary float and the exponent as input and returns back the list of digits that make up the decimal output and the exponent.

These algorithms were used later in developing the following two functions –

- *real32\_fromString: string  $\rightarrow$  Real32.real option*
- *real32\_toString: Real32.real  $\rightarrow$  string*

The *fromString* function takes a string representation of a decimal number as input and extracts the significand and exponent parts out of it. It then passes these values to the Bellerophon algorithm to get back a string. This string is converted to a real number and is returned by the function.

The *toString* function takes a real number as input and gets the floating-point representation of this number using the helper functions. The significand and exponent of the float obtained from here are passed to the FPP2 algorithm which returns back the decimal representation. A string representation of this number is constructed and is returned by the function.

The equivalent functions that exist in the MLton Real32 structure are –

- *Real32.fromString:*
- *Real32.fmt StringCvt.EXACT*

Various test cases are generated and are passed as parameters to these functions. The results obtained from the in-built functions are compared with those obtained from the current implementation.

## 5. RESULTS

The results obtained from the implementation of *toString* and *fromString* functions for various test cases are shown in the tables below. These results are compared with the existing implementations as mentioned in the previous section. Only a few of the test cases are shown here. All the test cases that have been used to test the implementation will be listed along with the code submission.

Input String	Output	
	<code>real32_fromString</code>	<code>Real32.fromString</code>
53.413	53.413	53.4129981995
0.12345678912	0.12345678848	0.123456791043
12345E~4	1.2345	1.23450005054
10000000000	1E10	1E10
434.08624	434.0862121582031	434.086242676

Table 4: decimal-to-binary results

Input String	Output	
	<code>real32_toString</code>	<code>Real32.fmt StringCvt.EXACT</code>
434.08624	434.08625	0.43408624E3
3.124639	3.124639	0.3124639E1
200	200.0000052	0.2E3
13.123457	13.12345678	0.13123457E2
785.8765	785.876540321	0.7858765E3
12.235702	12.2357012528	0.12235702E2

Table 5: binary-to-decimal results

As seen from the tables, there are some noticeable differences in the results obtained from both the implementations. This can be attributed majorly to the fact that David Gay has made some enhancements [5] to the original algorithms in order to achieve better performance. This project is a direct implementation of those algorithms without any changes.

## 6. CONCLUSION

The goal of this project was to implement the correctly rounded routines for binary-to-decimal and decimal-to-binary floating-point conversions in Standard ML using MLton compiler. These SML routines are a native implementation of the floating-point conversions that can replace existing *gdtol* libraries by David Gay that were written in C programming language. To achieve this, the algorithms given by Clinger and Steele and White are used. These algorithms have been successfully implemented in SML. The results from this implementation look similar to what has been expected.

The implementation of the algorithms has been successfully accomplished. Also the formatting required for the input and output has been achieved using the *toString* and *fromString* functions (only for single precision floating-point numbers). The major limitation to this project is its performance, which could have been improved. Moreover, the initial plan of implementing all the rounding modes has not been accomplished

## 7. FUTURE WORK

A list of things that have not been accomplished and can be worked upon are mentioned below –

- There are five different rounding modes defined by the IEEE standard for floating-point numbers. This project has been implemented using only the round to nearest (tie to even) rounding mode. Other rounding modes can be added to the implementation. Also, the rounding mode can be used as a parameter to the functions.
- The usage of *strings* in formatting the input and output is one of the major causes for the slow performance. String manipulations always have a cost. The usage of strings has to be eliminated wherever possible in order to achieve better performance.
- The enhancements suggested by David Gay can be implemented.
- Clinger suggested the usage of lookup tables in order to speed up the Bellerophon algorithm. This project uses direct computation of powers of 10 instead of the lookup table. Lookup tables can be implemented in order to get faster output.
- This project uses only single precision floating-point numbers. It can be extended to double precision floating-point numbers.
- Dragon4 algorithm can be implemented to have a more generalized output format.

## 8. REFERENCES

- [1] Guy L. Steele Jr., and Jon L White. *How to Print Floating-Point Numbers Accurately*. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, New York, June 1990).
- [2] William D. Clinger. *How to Read Floating Point Numbers Accurately*. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (White Plains, New York, June 1990).
- [3] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. ACM Computing Surveys, Vol 23, No 1, March 1991.
- [4] Chuck Allison. *Where did all my decimals go?* Proceedings of the Consortium of Computing Science in Colleges, Rocky Mountain Conference (February, 2006)
- [5] David M. Gay. *Correctly Rounded Binary-Decimal and Decimal-Binary Conversions*.
- [6] <http://mlton.org/>
- [7] <http://www.standardml.org/Basis/manpages.html>
- [8] Jean-Michel Muller, et al. *Handbook of Floating-Point Arithmetic*.