FPSim: A Floating-Point Arithmetic Demonstration Applet

Jeffrey Ward
Department of Computer Science
St. Cloud State University
waje9901@stcloudstate.edu

Abstract

An understanding of IEEE 754 standard conforming floating-point arithmetic is essential for any computer science student focusing on numerical computing. This topic, however, can often be difficult for students to understand from an abstract, conceptual point of view. FPSim presents instructors with a tool to visually demonstrate the operation of IEEE 754 floating-point arithmetic. This Java applet is designed to be a full and accurate implementation of the IEEE 754 specification. The applet demonstrates addition, subtraction, multiplication, and division of Single and Double floating-point numbers. Output shows the process of normalizing and rounding the number with guard bits and the sticky bit. It also shows the effects of the operation on the five floating-point exception flags.

In order to accurately implement the IEEE 754 standard, FPSim performs most arithmetic in software, with little reliance on the floating-point capabilities of either Java or the underlying hardware. The program stores numbers in binary as arrays of integers. Arithmetic functions then work with these arrays. Division is implemented by reciprocation through the use of Newton's Method. This paper will present the internal workings of FPSim.

Introduction

The ANSI/IEEE Std 754-1985 IEEE Standard for Binary Floating-Point Arithmetic defines the most widely used standard for binary floating-point arithmetic. Specifically, the standard defines how floating-point numbers are to be stored and rounded and how exceptional conditions are to be handled. Any student taking a course in numerical computing must begin with a solid comprehension of binary floating-point arithmetic as defined by this standard. Like anything else, a visual demonstration could give students a more thorough understanding of this topic. FPSim is designed to do just that. It is a Java applet which fully implements the IEEE 754 standard and provides a graphical display of the arithmetic process. What follows is a brief overview of the standard followed by a description of FPSim and its internal data structures and algorithms. For a more thorough overview of the IEEE 754 standard and floating-point arithmetic, the reader should see [3] and [4].

Binary Representation

Let β be the base of a floating-point number and p its precision. Then the number $\pm d_0.d_1d_2...d_{p-1} \times \beta^e$ represents the number

$$\pm(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + ... + d_{p-1}\beta^{-p-1})\beta^e,\ \ 0 \le d_i < \beta .$$

The part $d_0.d_1d_2...d_{p-1}$ is referred to as the significand (also known as the fraction or mantisa).

The IEEE 754 standard defines two floating-point number storage formats: Single and Double. A Single floating-point number consists of 32 bits and a Double floating-point number consists of 64 bits. Each format is separated into three fields as shown in Figure 1 and Figure 2.

Figure 1: The Single format bit fields.

Sign	Biased Exponent	Significand
1 bit	8 bits	23 bits ← Most significant bit Least significant bit →
31	30 23	22 0

Figure 2: The Double format bit fields.

Sign	Biased Exponent	Significand
1 bit	11 bits	52 bits ← Most significant bit Least significant bit →
63	62 52	51 0

The exponent is stored as an unsigned integer. From this integer a bias is subtracted to retrieve the correct, signed exponent. There are two numbers of interest associated with the exponent: e_{min} and e_{max} are the smallest and largest possible exponent values for a given format. Table 1 shows the details of the exponent for Single and Double format numbers.

Table 1: The exponent ranges for the standard IEEE 754 formats.

Format	***Exponent Length***	***Bias***	***Unbiased Range***	e_{min}	e_{max}
Single	8 bits	127	0 to 255	-126	127
Double	11 bits	1023	0 to 2047	-1022	1023

In addition to Single and Double types, the IEEE 754 standard allows for architecture-specific Extended formats. For example, the Intel IA-32 architecture includes 80-bit Double Extended floating-point numbers.

Normal and Subnormal Numbers

The most significant bit of the binary floating-point significand is always assumed to be the only digit to the left of the "binary point". Let $b_0.b_1b_2...b_{p-1}$ be the significand of a binary floating-point number of precision p. If b_0 is one, the number is said to be *normal*; if b_0 is zero, the number is *subnormal.* IEEE 754 treats normalized numbers as the common case and does not store the most significant bit. Subnormal numbers are required when a number's exponent would be less than e_{min} if normalized. In this case, the number is right-shifted until it has an exponent of $e_{min} - 1$ (see Table 2). This leads to a loss in precision and the *underflow* exception being thrown.

Table 2: Interpretation of binary floating-point numbers from their exponent.

Unbiased Exponent	***Interpretation***
$e = e_{min} - 1$	$\pm\ 0.\ b_1b_2...b_{p-1} \times 2^{emin}$
$e_{min} \leq e \leq e_{max}$	$\pm\ 1.\ b_1b_2...b_{p-1} \times 2^{e}$
$e = e_{max} + 1$	$\pm$ Infinity if $b_1 = b_2 = ... = b_{p-1} = 0$, NaN otherwise

Special Values

IEEE 754 defines two special values: Infinity and Not a Number (NaN). NaN results from performing some invalid operation like those listed in Table 3 below. There is no single representation of NaN, but instead a whole set of possible representations. Any number with an exponent of e_{max} + 1 and at least one bit of the significand field set is NaN. There is only one distinction made between any possible bit patterns of a NaN. If the most significant bit of the significand field is set, the number is a quiet NaN; if that bit is clear, the number is a signaling NaN. A signaling NaN used in one of the invalid operations listed in Table 3 will trigger an invalid exception, but a quiet NaN will not. An invalid operation involving either a signaling or a quiet NaN as an input will output a quiet NaN.

Table 3: Invalid operations which produce NaN.

Operation	***NaN Produced By***
Addition/subtraction	$\infty + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0 / 0$, ∞ / ∞

Rounding

Regarding the issue of rounding, the IEEE 754 standard states that

> "...every operation...shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the [four rounding] modes" [5]

Of course producing a result that is correct to infinite precision would be impossible much of the time, so the use of guard bits and a sticky bit is employed for the purpose of correct rounding. A set of *n* guard bits will contain the last *n* bits right- shifted out of the result during an operation. Essentially, the number of significant bits is temporarily increased by *n*. The sticky bit will capture any set bits that are right- shifted out of the guard bits. It is referred to as "sticky" because once set, it remains set until the current operation is complete. FPSim uses two guard bits and a sticky bit to ensure correctly rounded results.

There are four different methods for rounding specified by IEEE 754. The defualt, *round to nearest*, will round to the nearest representable number. In the event that two representable numbers are equidistant from the result, the even number is taken. The three other modes are *round towards +Infinity*, *round towards -Infinity*, and *round towards zero*. Certain care must be taken when rounding ±Infinity with these last three modes – see Table 4.

Table 4: Results of rounding ±Infinity.

Round Mode	***Result for +Infinity***	***Result for - Infinity***
Round Towards Zero	$1.11...1\times 2^{emax}$	$-1.11...1\times 2^{emax}$
Round Towards +Infinity	+Infinity	$1.11...1\times 2^{emax}$
Round Towards - Infinity	$-1.11...1\times 2^{emax}$	- Infinity

Floating-Point Exceptions

The IEEE 754 standard defines five floating-point arithmetic exceptions.

1) Invalid – Occurs from any of the operations listed in Table 3.
2) Division by zero – Occurs when attempting to divide by zero.
3) Overflow – Indicates that the result of an operation was larger than the maximum possible value for the format being used.
4) Underflow – Occurs when the result of an operation is subnormal.
5) Inexact – Indicates that the rounded result of an operation is not exact – in other words, some precision may have been lost.

Design

One of the primary goals for FPSim was to create an implementation that would be as true as possible to the IEEE 754 standard. For this reason, the program stores all numbers and performs all floating-point arithmetic in software. There is little dependence on the underlying hardware or Java virtual machine floating-point functionality.

Internal Floating-point Number Storage

Within the program, the significand of each number is stored as an array of integers taking on the values zero or one. For the sake of simplicity in the arithmetic algorithms, the leading bit of the significand, which is implicit in the IEEE Single and Double formats, is included in this array. The array is exactly the size of the number of significant bits, with the most significant bit at index zero.

Several other fields also describe the number. The exponent is stored as an integer and is unbiased. The sign bit, as well, is an integer, being either zero to indicate positive or one to indicate negative. The current state of the number – whether it is normal, subnormal, zero, NaN, or infinity – is also stored.

Addition and Subtraction

Addition and subtraction within FPSim both begin in the same procedure. This procedure will decide whether to add or subtract the two operands based on the specified operation and the operand signs. This algorithm is outlined in Appendix A.1. Once this decision has been made, the operands are sent to either the addition or subtraction procedure. These procedures are outlined in Appendix A.2 and A.3.

The task of adding or subtracting the significands is performed by software implementations of binary adder and subtracter circuits. Early in the development of FPSim, subtraction and addition of negative numbers was implemented by obtaining the twos-complement of the operand's significand and then adding. This lead to an overly complicated process for subtracting, and eventually a separate binary subtracter circuit was implemented.

Multiplication

The multiplication algorithm used in FPSim is outlined in Appendix A.4. Multiplication of the significands is performed by a software binary multiplier circuit. An array of size $2n$, where n is the number of bits in the significand, is used for the algorithm. When the multiplication is finished, the result is taken to be the first n bits in the array. The next two bits comprise the guard bits; any set bit after those two will cause the sticky bit to be set.

Division

Division in FPSim is by far the most complex of the four arithmetic operations. Division is performed by reciprocation, where the reciprocal for the divisor is found and then multiplied by the dividend. FPSim uses Newton's Method to find this reciprocal. We begin with the equation

$$f(r)=\frac{1}{r}-d,$$

where d is the divisor. The root of this function corresponds to the reciprocal of the divisor. Note that for the derivative, we have

$$f'(r)=-\frac{1}{r^2}.$$

If A_0 is an initial approximation for the reciprocal of the divisor, than at each iteration of Newton's Method, we will compute

$$\begin{aligned} A_n &= A_{n-1} + \frac{f(A_{n-1})}{f'(A_{n-1})} \\ &= A_{n-1} + \frac{1/r - d}{1/r^2} \\ &= A_{n-1} + r - dr^2 \quad for\ n \geq 1. \end{aligned}$$

Since A_{n-1} is an approximation of r, we replace r with A_{n-1} and obtain

$$\begin{aligned} A_n &= 2A_{n-1} - dA_{n-1}^2 \\ &= A_{n-1}(2 - dA_{n-1}) \quad for\ n \geq 1. \end{aligned}$$

The greatest difficulty in this approach to division is finding a good initial approximation for the reciprocal. FPSim uses a formula described in [2] to make this approximation. Let Y be the divisor, n be the number of significant bits in Y and m be an integer such that $0 < m < n$. Let $p = 1.y_1y_2...y_m$ and $q = 0.00...0y_{m+1}y_{m+2}...y_n$. Then the initial approximation A_0 is given by

$$A_0 = \frac{q(q - 2^{-m})}{p^3} \quad .$$

The computation of A_0 is the one and only part of FPSim that uses Java's floating-point functions. A hardware implementation would likely use a lookup table with a number of values of this formula already computed. This is not as feasible in FPSim, where a floating-point number requires much more than the 4 to 16 bytes needed in hardware. Even a relatively small lookup table would consume a very large amount of memory. Neither could the approximation be computed on demand in software, since it requires a division – the very operation that we are ultimately trying to perform. So the approximation is computed by hardware and then converted to FPSim's internal floating-point format.

Testing

A program called TestFloat [1] is being used to verify the accuracy of FPSim. TestFloat compares the results of a computation from a software-based floating-point arithmetic engine, called SoftFloat, to that of the target implementation. It reports any discrepancies between the two in the resulting number or exception flags. TestFloat was designed for testing hardware, so some modifications were necessary. Each line of C code in TestFloat that would normally compute the given operation was replaced with a function to execute a simple Java program. This program contained all of FPSim's arithmetic code. The operands and results were then passed back and forth between TestFloat and Java.

At the time of this paper's writing, testing is still in progress. At present, addition is largely error-free, with only a few minor corrections that need to be made. Multiplication is about 50% error-free, and division is almost completely untested.

References

1) Hauser, J. *TestFloat*. Retrieved February 21, 2004, from http://www.jhauser.us/arithmetic/TestFloat.html
2) Hennessy, J. L, & Patterson, D. A. (2003). *Computer Architecture: A Quantitative Approach*. 3rd Edition. San Francisco: Morgan Kaufmann.
3) Ito, M., Takagi, N., & Yajima, S. (1995). Efficient Initial Approximation and Fast Converging Methods for Division and Square Root. *IEEE 12th Symposium on Computer Arithmetic*. 2 – 9.
4) Overton, M. L. (2001). *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM.
5) Sun Microsystems. (1996). *Numerical Computing Guide*.
6) The Institute for Electrical and Electronics Engineers, Inc. (1985). *IEEE Standard for Binary Floating-Point Arithmetic*.

Appendix A

A.1 – Addition and Subtraction

```
Input: Floating- point numbers OPA and OPB
Output: Floating- point number RESULT.

If (OPA.exp < OPB.exp)
    Shift significand of OPA right until the exponent of OPA matches
    that of OPB;
Else If (OPA.exp > OPB.exp)
    Shift significand of OPB right until the exponent of OPB matches
    that of OPA;
If (OPA == NaN or OPB == NaN)
    RESULT = NaN;
    If (the NaN operand is a signaling NaN)
        Throw an invalid exception;
    Return;
If (the specified operation is addition)
    If (OPA.sign == OPB.sign)
        RESULT = OPA + OPB;
    Else
        If (OPA > OPB)
            RESULT = OPA – OPB;
        Else If (OPA < OPB)
            RESULT = OPB – OPA;
        Else
            RESULT = Zero;
Else If (the specified operation is subtraction)
    If (OPA.sign == OPB.sign)
        RESULT = OPA – OPB;
    Else
        RESULT = OPA + OPB;
```

A.2 – Addition

```
Input: Floating- point numbers OPA and OPB
Output: Floating- point number RESULT.

If (OPA == Infinity or OPB == Infinity)
    RESULT = Infinity;
    Return;
RESULT.significand = OPA.signficand + OPB.significand;
If (the addition operation resulted in a carry out)
    Shift RESULT's significand right by one and set the most significant
```

```
        bit to one;
Normalize RESULT;
Round RESULT;
```

A.3 – Subtraction

```
Input: Floating-point numbers OPA and OPB
Output: Floating-point number RESULT.

If (OPA == Infinity and OPB == Infinity)
        RESULT = NaN;
        Throw invalid exception;
        Return;
If (OPA == Infinity or OPB == Infinity)
        RESULT = Infinity;
        Return;
RESULT.significand = OPA.signficand – OPB.significand;
Normalize RESULT;
Round RESULT;
```

A.4 – Multiplication

```
Input: Floating-point numbers OPA and OPB
Output: Floating-point number RESULT.

If (OPA == NaN or OPB == NaN)
        RESULT = NaN;
        If (the NaN operand is a signaling NaN)
                Throw an invalid exception;
        Return;
If (one operand is Infinity and the other is Zero)
        RESULT = NaN;
        Throw invalid exception;
        Return;
If (OPA == Infinity or OPB == Infinity)
        RESULT = Infinity;
        Return;
If (OPA.sign == OPB.sign)
        RESULT.sign = Positive;
Else
        RESULT.sign = Negative;
RESULT.exponent = OPA.exponent + OPB.exponent;
RESULT.significand = OPA.significand * OPB.significand;
Normalize RESULT;
```

```
Round RESULT;
```

A.5 – Division

```
Input: Floating-point numbers OPA and OPB
Output: Floating-point number RESULT.

If (OPA == NaN or OPB == NaN)
    RESULT = NaN;
    If (the NaN operand is a signaling NaN)
        Throw an invalid exception;
    Return;
If (OPA == Zero and OPB == Zero)
    RESULT = NaN;
    Throw an invalid exception;
    Return;
If (OPB == Zero)
    RESULT = Infinity;
    Throw divide by zero exception;
    Return;
If (OPA.sign == OPB.sign)
    RESULT.sign = Positive;
Else
    RESULT.sign = Negative;
Compute OPB_INVERSE, the reciprocal of OPB;
RESULT = OPA * OPB_INVERSE;
Normalize RESULT;
Round RESULT;
```

A.6 – Normalization

```
Input: Floating-point number NUMBER

If (NUMBER.exponent < $e_{min}$)
    Right-shift NUMBER until NUMBER.exponent == $e_{min}$;
    Return;
If ($e_{min}$ < NUMBER.exponent ≤ $e_{max}$)
    Left-shift NUMBER until the most significant bit is one or until
    NUMBER.exponent == $e_{min}$;
    Return;
If (NUMBER.exponent > $e_{max}$)
    NUMBER = Infinity;
    Throw overflow and inexact exceptions;
    Return;
```

A.7 – Rounding

Input: Floating-point number NUMBER; guard/sticky bits from the last operation stored in the three-element array GUARD

```
If (no guard/sticky bits are set)
    Return;

If (Rounding Mode == Round Towards -Infinity)
    If (NUMBER.sign == Positive)
        Truncate NUMBER.significand;
    Else
        Increment NUMBER.significand;
Else If (Rounding Mode == Round Towards +Infinity)
    If (NUMBER.sign == Positive)
        Increment NUMBER.significand;
    Else
        Truncate NUMBER.significand;
Else If (Rounding Mode == Round Towards Zero)
    Truncate NUMBER.significand;
Else If (Rounding Mode == Round to Nearest)
    GUARDVALUE = GUARD[0] * 4 + GUARD[1] * 2 + GUARD[2];
    If (GUARDVALUE < 4)
        Truncate NUMBER.signficand;
    Else If (GUARDVALUE > 4)
        Increment NUMBER.significand;
    Else
        If (least significant bit of NUMBER.significand == 1)
            Increment NUMBER.significand;
        Else
            Truncate NUMBER.significand;
```