An Approach to Teaching Computer Arithmetic

E.M. Garzón, I. García, J.J. Fernández

Dept. Arquitectura de Computadores y Electrónica
Universidad de Almería
E-mail:{ester,inma,jose}@ace.ual.es

Resumo

In this work we present an initiative to support teaching computer representation of numbers (both integer and floating point) as well as arithmetic in undergraduate courses in computer science and engineering. Our approach is based upon a set of carefully designed practical exercises which highlights the main properties and computational issues of the representation. In conjunction to the exercises, an auxiliary computer-based environment constitutes a valuable support for students to learn and understand the concepts involved. For integer representation, we have focused on the standard format, the well known 2's complement. For floating point representation, we have made use of an intermediate format as an introduction to the IEEE 754 standard. Such an approach could be included in an introductory course related to either computer structure, discrete mathematics or numerical methods.

God made the integers, man made the rest.
L. Kronecker. German mathematician.

1 Introduction

Computational science is nowadays making it possible to solve grand-challenge problems thanks to greatly improved computational techniques and powerful computers. Numerical computing is the basis underlying computational science. Virtually, any technical, medical or scientific discipline relies heavily on numerical computing. Scientists and engineers make intensive use of numerical methods and powerful computers to solve complex problems, which may range from the modeling of the microstructure of the atom to building design [1].

Numerical computing is foremost based upon floating point computation. However, computer representation of integer numbers [2] is also involved since it is always implicit and it is essential for almost any task in which the use of a computer is present. The most important computer representation for integer numbers is the well known 2's complement, which constitutes the standard used in almost every modern computer. The main issues related to that format, which any computer scientist or engineer should be familiarized with, are the numeric ranges and how the arithmetic operations are carried out. Such issues are learned in first-level courses in computer science and engineering, usually making use of pencil-and-paper exercises as support.

Floating point computation [3] is undoubtedly of enormous importance in computational science. Computers have supported floating point computation since their earliest days, although using different representations developed by each computer manufacturer. However, in early 1980's, the fruitful cooperation between academic computer scientists and the most important hardware manufacturers allowed the establishment of a binary floating point standard, commonly known as the IEEE 754 Floating Point Representation [4]. In essence, the standard aimed at (1) making floating point arithmetic as accurate as possible, within the constraints of finite precision arithmetic; (2) producing consistent and sensible outcomes in exceptional situations (e.g., overflow, underflow, infinite, ...); (3) standardizing floating point operations across computer systems; and (4) providing the programmer with control over exception handling (e.g. division by zero). Nowadays, most computers offer this standard for floating point computation.

Due to the great importance of floating point computation, computer scientists and engineers should have an excellent knowledge of what a finite floating point representation and arithmetic involve and, in particular, of the ubiquitous IEEE 754 standard. First of all, from the point of view of numerical computing, computer scientists and engineers should be aware of extremely important issues such as, for instance, precision, ranges or algebraic properties related to any finite floating point representation. On the other hand, several aspects in the design of a computer system require a good knowledge on floating point. First, from the point of view of the computer architect, who has to deal with the design of instruction sets including floating point operations. Second, from the point of view of the compiler and programming language design, in the sense that the semantics of the language has to be defined precisely enough to prove statements about programs. Third, from the point of view of the operating system (as far as exception handling concerns), in the sense that trap handlers may be defined by the users/programmers to deal with the exceptions, according to the problem at hand.

The Joint IEEE Computer Society and ACM Task Force on Computing Curricula actually develops curricular guidelines for undergraduate programs in different computing disciplines [5]. Machine-level representation of data has always been a core unit within the Computer Architecture and Organization area in the introductory phase of the undergraduate curriculum, and tightly related to the Programming Languages and Computational Science areas.

In spite of its enormous importance, floating point representation still remains shrouded in mystery by the average computer science or engineering student, and only well understood by experts. Several initiatives have arisen in the nineties [6, 7, 8] and recently [10, 9] to make the floating point arithmetic accessible to the students. Some of them are focused directly on the IEEE floating point standard[7, 9, 10], explaining the representation itself and all the issues involved in it. Others [6, 8] make use of an intermediate floating point representation so as to illustrate the main concepts. The works of [6] and [10] also include exercises to clarify the concepts of precision and resolution involved in any finite floating point representation. Finally, [10] provides a set of computer programs to show clearly what the floating point computation using the IEEE standard involves.

In this work, we present our own initiative to support the teaching of the computer representation of numbers and arithmetic (both integer and floating point), intended to be included in a first level course for undergraduate computer science or engineering students. Our initiative combines (1) a set of key practical exercises for both the integer and the floating point cases, (2) the use of a supporting environment consisting of a set of auxiliary computer programs, and (3) the use of intermediate floating point representations as an introduction to the IEEE standard, with the aim of facilitating the illustration of all the computational issues involved in the 2's complement integer representation as well as in any floating point representation. Such an approach could be included in either a introductory course related to computer structure, discrete mathematics or numerical methods.

2 Teaching Computer Representation and Arithmetic of Integers

The most extended computer representation format for integer numbers is the well known 2's complement. The range of the 2's complement integer representation using strings of p bits is $[-2^{p-1}, 2^{p-1}-1]$. This is a representation specially well suited from the point of view of the computer hardware, since it does not require additional special hardware for integer subtraction. In order to illustrate all the issues related to this representation, a complete set of carefully designed practical exercises have been designed, supported by an auxiliary computer-based environment.

2.1 Representation of integer numbers.

The type of exercises in this category help the students to have experiences with the representation itself, the range of numbers that is covered, and different situations in which numbers do not fit ranges. In that sense, the exercises that are proposed include (1) conversions of integer numbers between decimal and 2's complement formats using different word lengths, and vice versa; (2) calculation of the minimum number of bits that are needed to represent given numbers; (3) determination of 2's complement representation ranges for different word lengths.

As an example of practical exercise, we ask students to do decimal-to-2's complement conversions of given numbers and then feed the results into the reverse conversion, using our support environment. As a result of such a "pipeline", only those numbers that fit the corresponding representation ranges will result in themselves. Students are then encouraged to think about the reasons underlying the different results. With this type of exercises, students experience, on their own, the limits of the representation ranges and acquire skill to realize and deal with such situations in real life.

2.2 Integer arithmetic.

Exercises in this category mainly include the computation of integer arithmetic operations (addition, subtraction, multiplication and division) using different word lengths. This type of exercises and the software that has been developed illustrate the algorithms used to carry out the integer arithmetic in the hardware units. In addition, the understanding of exceptional situations derived from overflow is specially facilitated. Auxiliary programs have been included in the support environment to show in detail the procedures of the integer addition and subtraction, and, specially, the Booth algorithm for integer multiplication as well as the restoring and non-restoring algorithms for integer division. Such procedures are shown by the software according to the notation used in [2]. For the particular case of multiplication, our environment also affords the chance to show the procedure in a pencil-and-paper format.

As an example of practical exercise related to integer arithmetic, we ask students to carry out different arithmetic operations with certain numbers and different word lengths, using the support environment. The numbers have been carefully chosen so that all possible situations occur, specially related to overflow. Figure 1 shows the output yielded by the support environment that describes the procedure of the integer multiplication of -5 and $+7$ using a word length of 4 bits through the Booth algorithm: The first column denotes the iteration of the algorithm; the second column represents the register that supposedly contains the multiplicand; the third column indicates the action to do in the corresponding step of the algorithm; finally, last column represents the double-sized register which initially contains the multiplier at its lower significant half, and into which the result of the multiplication is progressively computed and stored. The extra bit needed by the Booth algorithm is the right-most one in the last column.

MULTIPLICATION $(-5) \times (+7)$

MULTIPLICAND: $-5 \equiv 1011_2$
MULTIPLIER: $+7 \equiv 0111_2$

Iter	Multiplicand	Action	Product-Multiplier
0	1011	Initial Values	0000 0111 0
1	1011	Prod=Prod - Multiplicand	0101 0111 0
	1011	Right Shift	0010 1011 1
2	1011	No Operation	0010 1011 1
	1011	Right Shift	0001 0101 1
3	1011	No Operation	0001 0101 1
	1011	Right Shift	0000 1010 1
4	1011	Prod=Prod + Multiplicand	1011 1010 1
	1011	Right Shift	1101 1101 0
		Final Result: -35 ==> 11011101	

Figura 1: Integer multiplication by means of the Booth algorithm. The operands -5 and $+7$ are multiplied using a word length of 4 bits.

3 Teaching Floating Point Representation and Arithmetic

Any floating point representation makes use of a exponential notation to represent real numbers, in which any number is decomposed into mantissa or significand and an exponent for a, normally, implicit base. For instance, a nonzero real number is represented by

$$\pm m \times B^E, \text{ with } 1 \leq m < B$$

where m denotes the mantissa, E the exponent, and B the implicit base.

Any computer-based representation involves a finite number of bits to represent the main fields of the floating point representation, mantissa and exponent. Consequently, rounding techniques have to be used for the real numerical value to fit the number of bits of the representation format. Therefore, floating point computation is by nature inexact. In the particular case of the IEEE 754 standard, the single precision of the standard involves 32 bits, one bit for the sign, 23 bits are used for the mantissa and the exponent is represented by 8 bits, using an implicit binary base. Double precision in the standard makes use of 64 bits, 52 for the mantissa and 11 for the exponent.

Finite floating point representation involves a considerable number of issues which any computer scientist or engineer should be aware of. The most important issues derived from the use of such a representation are:

- There exists an interesting trade-off in terms of precision and range of the format. The number of bits in the exponent and the mantissa defines the range and the precision, respectively.
- The gap between successive floating point numbers of the representation varies along the real numerical intervals. The gap is smaller as the magnitudes of the numbers themselves get smaller, and bigger as the numbers get bigger.
- There is a relatively great gap between zero and the nearest non-zero floating point number. However, the use of *denormalized* numbers allows underflow to be gradual, evenly filling such a gap. Denormalized numbers are denoted by a zero exponent field and an unnormalized mantissa, assuming that the implicit bit is 0. In this way, the value of a denormalized number in the single precision IEEE standard is given by:

$$\text{Real value} = (-1)^s \times (0.m) \times 2^{-126}$$

- The machine epsilon, ϵ_{mach} refers to the gap between 1.0 and the smallest floating point number greater than 1.0. It provides an idea of the accuracy of the floating point operations: floating point values are accurate to within a factor of about $1 + \epsilon_{\text{mach}}$ [10]. Except for denormalized numbers, neighboring floating point numbers differ by one bit in the last bit of the mantissa, so $\epsilon_{\text{mach}} = 2^{-n_m}$. For the single precision IEEE standard, $\epsilon_{\text{mach}} = 2^{-23} \simeq 10^{-7}$.
- The zero number is represented by a string of zero bits.
- Rounding, truncating and cancellation errors as well as error accumulation are heavily involved in the floating point arithmetic,
- Special quantities (Infinite, Not-a-Number) are designed to handle exceptional situations.
- Trap handlers for exception handling are under control of the user/programmer.
- Floating point representation introduces serious anomalies with respect to the conventional algebra, in the sense that some fundamental rules of arithmetic, such as the associative or distributive properties, are no longer guaranteed.

We have developed a relatively simple floating point representation format faithfully resembling all the important issues in the IEEE 754 floating point standard, with the aim of attenuating the relatively tedious aspect of teaching these kind of issues. This format is based on a user-defined word length so that the concepts related to precision and range is easily illustrated. The base format makes use

of 12 bits, 1 for sign, 5 for the exponent and 6 for the mantissa. Because the anomalies arising as a result of a finite floating point representation are amplified, the use of this base format allows to easily show them, and therefore, facilitating the understanding by the student. Table 1 summarizes the most important features of this representation. This shorter version of the IEEE standard 754 has a five-fold aim: (1) an easy identification of the relationship between the real number and its floating point representation as well as the effects of the truncation/rounding in the conversion; (2) a clear illustration of the concepts related to the range/precision trade-off; (3) the understanding of the algorithms underlying the floating point arithmetic operations; (4) a clear identification of which, when and why anomalies arise in finite precision floating point computation (such anomalies include the effects of rounding errors in arithmetic operations, the non-fulfillment of the properties of the conventional algebra, and the exceptional situations); (5) the word length may be small enough to allow calculations by-hand, if convenient.

Tabela 1: The Simpler Floating Point Representation

The most important features of the representation

Length of the representation:	12 bits:
	sign: 1 bit
	exponent: n_e =5 bits
	mantissa: n_m =6 bits
Exponent Bias:	$2^{n_e-1}-1 = 15 = 01111_2$
Range of the exponent:	$[-14, 15]$
Machine epsilon:	$\epsilon_{\text{mach}} = 2^{-n_m} = 2^{-6}$
Supported rounding modes:	*Truncation , Round to Nearest*
Denormalized numbers:	Fully supported
Supported exceptions:	*Overflow, Underflow, Invalid, Division by Zero*

The most important (positive) special values

Special value	Representation	Value
Largest normalized:	0 11110 111111	$+1.984375 \times 2^{15}$
Smallest normalized:	0 00001 000000	$+1.000000 \times 2^{-14}$
Largest denormal.:	0 00000 111111	$+0.984375 \times 2^{-14}$
Smallest denormal.:	0 00000 000001	$+0.015625 \times 2^{-14}$
+ Zero	0 00000 000000	+0.0
+ Infinite:	0 11111 000000	--
Not-a-Number (NaN):	0 11111 111111	--

In summary, the floating point format proposed here is specially well suited for illustrating all issues related to finite precision floating point representation, and in particular the IEEE standard 754. In that sense, a complete set of practical exercises have been thoroughly devised to highlight such aspects. Some of the exercises are based on the aforementioned format, but many others work on a general floating point format, by specifying different lengths for the exponent and mantissa fields. These practical exercises are accompanied by a computational environment that turns out to be an excellent auxiliary tool, from the pedagogical point of view, to illustrate such not-so-trivial issues.

The practical exercises that have been devised fall into different categories:

- The first category of exercises is intended to help students to learn the procedure to convert a real number to its floating point representation, analysing the effects of the rounding modes in terms of the relative error. In that sense, the exercises deal with the determination of the representation of different real numbers with the simpler floating point format (12 bits length), using different rounding techniques. Specially care has been taken to choose proper real values to give rise special situations in the conversions (for example, the smallest normalized floating point number, the biggest floating point number, the extreme values in the denormalized floating point range, etc).

  Figure 2 shows some examples of representations, as reported by the computer program in charge of the conversion. The first example was computed using truncation as the rounding mode, resulting in that the real number 1.999 is represented as 1.984375. The second, which turns out to be a denormalized number in this format, used round-to-nearest rounding mode. The third and fourth are examples of values bringing about exceptions: 0.0000001 is too small and 65500.0

```
Real Number:     1.999000  = 1.999000e+00 = 0.999500 x 2^1
Representation:  0 01111 111111
Float. P. Value: 1.984375  = 1.984375e+00 = 1.984375 x 2^0
-------------------------------------------------------------
Real Number:     0.000040  = 4.000000e-05 = 0.655360 x 2^-14
Representation:  0 00000 101010
Float. P. Value: 0.000040  = 4.005432e-05 = 0.656250 x 2^-14
                 !! Denormalized Number !!
-------------------------------------------------------------
Real Number:     0.0000001 = 1.000000e-07 = 0.838861 x 2^-23
Representation:  0 00000 000000
Float. P. Value: 0.0000000 = 0.000000e+00 = 0.000000 x 2^0
                  ! Exception: Underflow !
-------------------------------------------------------------
Real Number:     65500.000000 = 6.550000e+04 = 0.999451 x 2^16
Representation:  0 11111 000000
Float. P. Value: +Infinite
                  ! Exception: Overflow !
-------------------------------------------------------------
```

Figura 2: Representation of certain values according to our floating point format, using 5 and 6 bits for exponent and mantissa, respectively.

is to big to be represented in the format. The output of the program shows the input real number, the representation, and the value of the floating point representation.

- The following group of exercises aims at facilitating the understanding of the concepts of precision, range and the precision-range trade-off. Exercises in this group include the following aspects:

  – determination of the representation of different real numbers using different formats (i.e., varying bits for exponent and mantissa).
  – determination of the gap between floating point numbers in some intervals of different representation formats.
  – determination of the ranges covered for different floating point formats (including the subrange of denormalized numbers, and that of normalized numbers).
  – computation of the minimum number of bits dedicated for the exponent and mantissa fields to fulfill certain given constraints, for example, to be able to distinguish between the representations of two given real numbers very close to each other.

  As an example, Figure 3 summarizes the process to work out the minimum number of bits (in exponent and in mantissa fields) required to distinguish the real numbers 15.9 and 15.925 and, at the same time, to represent the real number 100000. Students start working from the simpler format, using $n_e = 5$, $n_m = 6$, and then they progressively increase the number of bits in mantissa until 15.9 and 15.925 get distinguishable representations. In that figure, it can be seen that $n_m = 8$ is the minimum number of bits required, which makes 15.9 and 15.925 get represented as 15.875 and 15.90625, respectively. Then, students try to get the representation of 100000 using the format just computed, obtaining an *Overflow* exception. They progressively increase the number of bits in the exponent field until 100000 gets representable. The final answer is that the minimum format to fulfill the requirements of the assignment is to use $n_e = 6$, $n_m = 8$.

- The following class of exercises is related to arithmetic, and is intended to help students to learn the procedures to carry out the floating point arithmetic operations. Exercises in this group are mainly focused on the computation of floating point additions, subtractions, multiplications and divisions using different pairs of operands, experiencing with the different rounding modes, and measuring the relative error of the results. The operands that are proposed are specially chosen so that different situations occur. Specially interesting is, for instance, the addition of pairs of operands that are enormously different in magnitude (for instance, our simpler format, using $n_e = 5$ and $n_m = 6$, makes $1000.0 + 2.5$ equal to 1000.0).

  Figure 4 is intended to show the output of the program in charge of illustrating the process of floating point addition. As can be seen, the program specifies in detail all the stages in the

Determination of the minimum number of bits in mantissa field to distinguish 15.9 and 15.925

	15.9	15.925
Floating Point Format	Representation	Representation
$n_e = 5$, $n_m = 6$	0 10010 111111	0 10010 111111
$n_e = 5$, $n_m = 7$	0 10010 1111110	0 10010 1111110
$n_e = 5$, $n_m = 8$	0 10010 11111100	0 10010 11111101
$n_e = 5$, $n_m = 8$	15.9→15.875	15.925 →15.90625

Determination of the minimum number of bits to distinguish 15.9 and 15.925 and represent 100000

	Real number: 100000	
Floating Point Format	Representation	Value
$n_e = 5$, $n_m = 8$	0 11111 00000000	+Infinite
$n_e = 6$, $n_m = 8$	0 101111 10000110	99840.0

Figura 3: Procedure to determine the minimum floating point format to fulfill the requirements that the real numbers 15.9 and 15.925 are distinguishable and that the real number 100000 is representable. Top: the minimum mantissa length to distinguish 15.9 and 15.925 is worked out to be $n_m = 8$. Bottom: $n_e = 6$ is the minimum exponent length required to represent 100000.

procedure of such an arithmetic operation: (1) conversion of operands to their representation; (2) alignment of mantissas; (3) addition of mantissas; and finally (4) the result of the normalization and rounding. Such a program is extremely useful for students to discern all the issues concerning the round-off problems in arithmetic operations. This operation was performed using our simpler floating point representation ($n_e = 5$, $n_m = 6$).

Floating Point Addition 1.0 + 0.999, using 2 guard bits.
Format using: 5 bits/Exp. 6 bits/Mantissa; 2 Guard bits.

```
1.- Representation of operands.
                              s   e      m      s   e      .m
                 Operand  1 ->  0 01111 000000 -> 0 01111   1.000000  = 1.000000
                 Operand  2 ->  0 01110 111111 -> 0 01110   1.111111  = 0.999000

2.- Alignment of operands.
                              s   e      .m      g
                 Operand  1 ->  0 01111   1.000000 00
                 Operand  2 ->  0 01111   0.111111 10

3.- Addition of mantissas.
                              s   e      .m      g
                 Operand  1 ->  0 01111   1.000000 00
                 Operand  2 ->  0 01111   0.111111 10
                 ------------------------------------
                 Addition   ->  0 01111   1.111111 10

4.- Normalization and Rounding of the result.

                 Result     ->  0 10000   1.000000     = 2.000000


Result of the Addition:  0 10000 000000
Decimal Value:  2.000000 = 2.000000e+00 = 1.000000 x 2^1
```

Figura 4: Procedure and result of the floating point addition of 1.0 and 0.999.

- This category deals with the algebraic anomalies that arise as a result of the finite nature of the floating point representation. It is imperative that students have insights into the machinations of floating point arithmetic on computers and, in particular, into those anomalies, to succeed in real life computation problems. So, exercises in this category include:
  - analysis of the cancellation error that arises in the subtraction of two operands extremely close to each other.

– practical exercises to show the effect of accumulation of rounding errors. For instance, multiplication of a given floating point number by a integer constant by means of accumulated sum of the former, using different rounding modes. Students have to do a graph showing the evolution of the relative error during the accumulated sum. Figure 5 shows the result of the multiplication 1.999×10 as shown by the auxiliary programs. Here, the intermediate results from the accumulated sum are shown. On the left, the sums are carried out using truncation. On the right, the round-to-nearest rounding mode is used. The last line shows the final result of the multiplication. Clearly, the superiority of the round-to-nearest rounding mode is evident. These results are rapidly generated using one of the programs of our environment. Students have to analyze the results, and generate graphs of the error evolution.

Multiplication 1.999×10

	Truncation		Round to Nearest	
Iter	Representation	Value	Representation	Value
1	-> 0 01111 111111	-> 1.99900	0 01111 111111	-> 1.99900
2	-> 0 10000 111111	-> 3.96875	0 10000 111111	-> 3.96875
3	-> 0 10001 011111	-> 5.93750	0 10001 011111	-> 5.93750
4	-> 0 10001 111110	-> 7.87500	0 10001 111111	-> 7.93750
5	-> 0 10010 001110	-> 9.75000	0 10010 001111	-> 9.87500
6	-> 0 10010 011101	-> 11.62500	0 10010 011111	-> 11.87500
7	-> 0 10010 101100	-> 13.50000	0 10010 101111	-> 13.87500
8	-> 0 10010 111011	-> 15.37500	0 10010 111111	-> 15.87500
9	-> 0 10011 000101	-> 17.25000	0 10011 000111	-> 17.75000
10	-> 0 10011 001100	-> 19.00000	0 10011 001111	-> 19.75000

Figura 5: Multiplication 1.999×10 by means of an accumulated sum. The simpler floating point representation has been used (5/6 bits for exponent/mantissa).

– practical exercises to show that some fundamental rules of the conventional algebra are not fulfilled in floating point computation:
  * floating point addition is not associative.
  * floating point multiplication is not associative.
  * floating point multiplication does not necessarily distribute over addition
  * ordering of operations is significant.
  * the *cancellation* property is not always valid, i.e., there exist positive floating point numbers A, B, C such that $A + B = A + C$ and $B \neq C$.
  * multiplication of a floating point number by its inverse is not always equal to 1.
  * it is almost always wrong to ask whether two floating point numbers are equal.

One of the most interesting examples of algebraic anomalies is related to the computation of Harmonic series:

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n.$$

We use such a series to show how the ordering of operations may be extremely significant. We propose to compute the series, first, literally as in the formula and, second, in reverse order. The results help students to learn that summing the smallest values first, progressively increasing in magnitude, yields more accurate final results. Such an ordering avoids loosing the lowest precision bits of the smallest quantities in summing with values very different in magnitude. Figure 6 shows the results generated by our software package. This exercise aims at highlighting the fact that the ordering of floating point operations may be significant. On the left, the results for an ordering

according to the original formula. On the right, the results for the reverse ordering. The results include the current term that is to be added in the series, and the intermediate value of the accumulated sum.

On the other hand, such a series is mathematically proven to diverge. However, in floating point representations, the series converge due to the round-off errors. We have also designed practical exercises on that.

Computation of Harmonic series

	$\sum_{n=1}^{10} \frac{1}{n}$		$\sum_{n=10}^{1} \frac{1}{n}$	
Index	Number	Sum 1..N	Number	Sum N..1
1	1/1 = 1.00000	1.000000	1/10 = 0.10000	0.100000
2	1/2 = 0.50000	1.500000	1/9 = 0.11111	0.210938
3	1/3 = 0.33333	1.828125	1/8 = 0.12500	0.335938
4	1/4 = 0.25000	2.062500	1/7 = 0.14286	0.476562
5	1/5 = 0.20000	2.250000	1/6 = 0.16667	0.640625
6	1/6 = 0.16667	2.406250	1/5 = 0.20000	0.843750
7	1/7 = 0.14286	2.562500	1/4 = 0.25000	1.093750
8	1/8 = 0.12500	2.687500	1/3 = 0.33333	1.421875
9	1/9 = 0.11111	2.812500	1/2 = 0.50000	1.921875
10	1/10 = 0.10000	2.906250	1/1 = 1.00000	2.937500

Figura 6: Computation of the Harmonic series with $N = 10$ using our simpler floating point format (5 bits exp., 6 bits mantissa), and using different orderings.

- The last category of exercises comprises those related to exceptions. Since the IEEE standard 754 allows exception handling to be under control of the user/programmer, it is extremely important to familiarize students with the situations that produce exceptions, how to manage them, and the floating point values resulting from exceptions. The exercises in this category are mainly intended to come exceptions into manifest:
  - practical exercises to show that some arithmetic operations with, in principle, normal numbers may result in exceptions because the operation yields a result not representable in the format. For instance, the multiplication of extremely small floating point numbers may give rise to an *Underflow* exception, or the addition/multiplication of two big numbers may result in an *Overflow* exception, etc. Two concrete examples are that $0.0009 * 0.001$ produces an *Underflow* exception and $1875 * 35$ brings about an *Overflow* one, if our simpler format (with $n_e = 5$ and $n_m = 6$) is used.
  - computation of $A + \infty$, $A * \infty$, $A/0$, A/∞, etc, given a floating point number A.
  - computation of $\infty + 0$, $\infty + \infty$, $\infty - \infty$, $\infty * 0$, $\infty * (-\infty)$, $0/0$, ∞/∞, $\infty/0$, etc.

To conclude this section, we should mention that all the practical exercises that we have devised are supported by the computer-based environment that will be described in the following section. The programs in the environment are intended to provide answers and their justifications to all the questions and exercises formulated in the assignments.

4 Conclusions

In this article we have described an approach to teaching computer representation of numbers and arithmetic. This initiative mainly consists of a complete set of thoroughly designed practical exercises conceived to emphasize all the important issues in finite length computer arithmetic. A computational environment that turns out to be a very valuable support tool for students to practice is also provided. Our experience after several years of lecturing and teaching computer arithmetic in an introductory course on computer organization in undergraduate computer science curricula allows us to claim the success of this initiative.

As far as the integer representation is concerned, we have experienced that the use of carefully designed practical exercises allows students to learn easily how the 2's complement represents the integer numbers, and all the issues regarding range and word lengths. The algorithms and the computer hardware involved in the integer arithmetic are also facilitated. In conjunction to the practical exercises, the computer-based environment devised in this work provides students with a valuable support for their learning.

From the pedagogical point of view, we have experienced that teaching floating point representation based upon a simple format which helps us illustrate all the issues in any floating point representation system is successful. We use this simple format as an introduction to the IEEE 754 floating point standard. Students feel more comfortable to deal with the IEEE standard once they are familiarized with the simple format and all the computational issues involved. The use of a simpler format allows all the singularities of floating point representation to be highlighted and amplified, and consequently, students are more impressed by those effects.

In ultimate instance, this approach facilitates lecturers to achieve the goal as a academic staff of making students aware of the computational issues involved in finite length number representation and arithmetic. This approach may be valuable for introductory undergraduate courses related to computer organization, programming, discrete mathematics and numerical methods in computer science or engineerings.

Acknowledgments

This work has been partially supported by the Spanish CICYT through grants TIC99-0361 and TIC2002-00228.

Referências

[1] C.W. Ueberhuber *Numerical Computation. Methods, Software, and Analysis*, Vols. 1&2. Springer-Verlag, 1997.

[2] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface* Morgan Kaufmann Pub., 1998.

[3] D.E. Knuth. *The Art of Computer Programming*, 3rd ed, volume 2, Seminumerical Algorithms. Addison-Wesley, 1998.

[4] W. Kahan. IEEE Standard 754 for Binary Floating-Point Arithmetic. WWW document, 1996. http://www.cs.berkeley.edu/ wkahan/ieee754status/ieee754.ps

[5] Curriculum 2001 Joint IEEE Computer Society/ACM Task Force. "Year 2001 Model Curricula for Computing ((CC-2001)," Final report, December 15, 2001.

[6] T.J. Scott. "Mathematics and computer science at odds over real numbers," *ACM SIGCSE Bulletin*, 23(1):130–139, 1991.

[7] D. Goldberg. "What every computer scientist should know about floating-point arithmetic," *ACM Comp. Surveys*, 23:5–48, 1991.

[8] C.W. Steidley. "Floating point arithmetic basic exercises in mathematical reasoning for computer science majors," *Computers in Education Journal*, 2(4):1–6, 1992.

[9] W. Kahan. Ruminations on the design of floating-point arithmetic. WWW document, 2000. http://www.cs.nyu.edu/cs/faculty/overton/book/docs/)

[10] M.L. Overton. *Numerical Computing and the IEEE Floating Point Standard.* SIAM, 2001.

[11] S. Guelich, S. Gundavaram, G. Birznieks. *CGI Programming on the World Wide Web.* O'Reilly, 2000.