PHYS-4007/5007: Computational Physics
Course Lecture Notes
Appendix F

Dr. Donald G. Luttermoser
East Tennessee State University

Version 7.0

Abstract

These class notes are designed for use of the instructor and students of the course PHYS-4007/5007: Computational Physics I taught by Dr. Donald Luttermoser at East Tennessee State University.

Appendix F: How Computers 'See' Numbers and Letters

A. Introduction.

1. Computers work with numbers (and any character for that matter) in **binary format**: Two **bits** (= Binary digITs), $0 \equiv$ *off* and $1 \equiv$ *on*.

a) There are 2^N integers that can be represented with N bits.

b) The sign of the integer is handled with the first bit ($0 \equiv$ positive number), which leaves $N-1$ bits to represent the value of the integer.

c) Therefore, N-bit integers will be in the range (absolute value-wise) of $[0 : 2^{N-1}]$.

i) Hence an 8-bit machine can handle integers in the range [–128 : 127].

ii) A 16-bit machine can handle integers in the range [–32,768 : 32,767].

iii) A 32-bit machine can handle integers in the range [–2,147,483,648 : 2,147,483,647].

iv) Finally, a 64-bit machine can handle integers in the range $[-9.223372... \times 10^{18} : 9.223372... \times 10^{18}]$. Note I am using 'real' notation here due to the large size of the number, on a computer, such a number would have no decimal point. Also note that in computer programming languages, numbers in scientific notation use the "E" ("D" for double

precision – see below) notation: [–9.223372E18 : 9.223372E18].

v) Calculating (or expressing) integers that are smaller (if negative) or larger (if positive) than the given chip size would result in an *underflow* (negative) or *overflow* of the computer register (and typically crash one's program) $\Longrightarrow$ the bigger the chip, the better it is for numerical work.

d) Since binary strings of numbers are not easy for people to work with, a **compiler** is used to translate the binary numbers of the machine to either *octal* (base 8, instead of base 2), *decimal* (base 10, what we normally are used to), or *hexadecimal* (base 16 numbers).

B. Bits and Bytes.

1. Numbers and character strings are stored on computers in **words**, where the *word length* is often expressed in **bytes**:

$$1 \text{ byte } (1 \text{ B}) \equiv 8 \text{ bits } (8 \text{ b}). \qquad \text{(F-1)}$$

a) Conventionally, storage size is measured in bytes (or kilobytes [KB], megabytes [MB], and gigabytes [GB]).

b) However, here "kilo" does not mean 1000, instead it is equal to

$$1 \text{ KB } (1 \text{ K}) = 2^{10} \text{ bytes } = 1024 \text{ bytes}. \qquad \text{(F-2)}$$

c) In the past, many machines measure their memory size in units of 1/2-kilobytes called **blocks**, or more precisely

$$512 \text{ B } = 2^{9} \text{ bytes } = 512 \text{ bytes}. \qquad \text{(F-3)}$$

d) One byte is the amount of memory required to store a single character. This adds up to a typical typed page requiring ~ 3 KB.

C. The Details of Data Types.

1. As discussed in §I of these notes, there are two basic types of data that are operated on and stored by computers $\Longrightarrow$ **integers** and **real numbers**. Depending on the programming language one is using, there are various types of "integers" and "reals." We will now outline the details of how computers store and operate on these data types.

2. **Integers** are stored as

$$I = (-1)^s \times \left(\sum_{n=1}^{N-1} \alpha_n 2^{n-1}\right), \tag{F-4}$$

where 's' is the sign bit ($0 \equiv$ positive, $1 \equiv$ negative) and α_n takes on either a '1' value (the bit is set) or '0' value (the bit is not set). For example, on an 8-bit machine ($N = 8$), we can write –57 as

$$\begin{aligned}&(\text{sign bit set to } 1) \times \left[(1 \times 2^0) + (0 \times 2^1) + (0 \times 2^2)\right.\\ &\quad \left.+(1 \times 2^3) + (1 \times 2^4) + (1 \times 2^5) + (0 \times 2^6))\right]\\ &\quad = (-)\ \ 1 + 0 + 0 + 8 + 16 + 32 + 0 = -57,\end{aligned}$$

or in binary format (where the bits are listed in the opposite-order of the summation above), that is, the least significant digit (2^0) is recorded on the far right side of the binary number (just as it is in decimal notation):

$$-57 = 1\ \ 011\ \ 1001,$$

where the first bit is the sign bit ($1 =$ negative).

3. Integers can come in a variety of flavors in various programming languages:

a) **Logical**: 1-bit word, values = [.FALSE. (=0) : .TRUE. (=1)], maximum value = 1.

b) **Short Integer**: 8-bit or one-byte word length, range = [–128 : 127], maximum value = $2^7 - 1$, number of digits = 3.

c) **Character**: Like a short integer typically containing the ASCII code of the character, however, the programmer in a higher-level language (like Fortran or IDL) uses the **string** notation (characters surrounded by single- and/or double-quotation marks) and the compiler changes these characters to the short integer ASCII code, then to binary, to which the machine then operates upon. See §I.E for useful information dealing with the ASCII character set.

d) **Integer**: 16-bit or two-byte word length, range = [–32,768 : 32,767], maximum value = $2^{15} - 1$, number of digits = 5.

e) **Long Integer** (sometimes just called 'Long' or 'Double-Precision Integer'): 32-bit or four-byte word length, range = [–2,147,483,648 : 2,147,483,647], maximum value = $2^{31} - 1$, number of digits = 10.

f) **Double-Long Integer** (sometimes just called 'Double Long' or 'Quad-Precision Integer'): 64-bit or eight-byte word length, range = [–9.22E18 : 9.22E18], maximum value = $2^{63} - 1$, number of digits = 20. Note that these types of integers are only found on 64-bit architectures, and even then, only found in some programming languages (*e.g.*, Intel Fortran which is based on the old DEC Fortran compilers).

4. Unlike integers, **real** numbers are indicated with a decimal points located in the number.

5. The computer can handle real numbers in two different ways: *Fixed-point* (not to be confused with 'fixed' or 'integer' data types) and *floating-point* notation.

a) In fixed-point real notation, the number x is represented as

$$x_{\rm fix} = \text{sign} \times (\alpha_n 2^n + \alpha_{n-1} 2^{n-1} + \cdots + \alpha_0 2^0 + \cdots + \alpha_{-m} 2^{-m}) \ , \qquad \text{(F-5)}$$

(note that the textbook uses $I_{\rm fix}$ for this type of number).

b) The first bit is used to store the sign of the number and the remaining $N-1$ bits are used to store the α_i values such that $n+m = N-2$, where N is the total number of bits used to represent a number. The particular values for N, m, and n are machine dependent. For instance, the number 9.875 could be represented in real fixed-point binary as

$$\begin{aligned} 0 \ \ 1001111 \Longrightarrow (\text{sign bit}) \times \big[&(1\times 2^3) + (0\times 2^2) \\ &+(0\times 2^1) + (1\times 2^0) + (1\times 2^{-1}) + (1\times 2^{-2}) \\ &+(1\times 2^{-3})\big] = 8 + 0 + 0 + 1 + 0.5 + 0.25 + 0.125 \\ &= 9.875 \end{aligned}$$

on an 8-bit machine.

c) All fixed-point real numbers have the same absolute error of 2^{-m-1} [the term left off the right-hand side of Eq. (I-5)].

d) The correspondingly disadvantage is that *small* numbers (those which the first string of α values are zeros) have large *relative* errors.

e) Relative errors tend to be more important than absolute errors (we cover errors in more detail in §IV of these notes), fixed-point real numbers are used mainly in special applications (like business), but typically not used in science and engineering.

6. In scientific work, the programming language compilers (like Fortran and IDL) use floating-point numbers for reals (as such, reals are often referred to as 'floats' in some languages).

a) In floating-point notation, the number x is stored as a sign, a mantissa, and an exponential field (*expfld* in Eq. I-6).

b) The number is reconstituted as

$$x_{\rm float} = (-1)^s \times \text{ mantissa } \times 2^{\,(\text{expfld} - \text{bias})}\,, \qquad \text{(F-6)}$$

(note that the book uses a slightly different form for this equation). The mantissa contains the significant figures of the number, s is the sign bit (still the first bit of the binary number), and the actual exponent of the number has a *bias* added to it.

• Since we have a sign bit, the mantissa will always be positive.

• The bias guarantees that the number stored as the exponent field is always positive (of course the actual exponent can be negative).

• The use of the bias is rather indirect. For example, a single-precision 32-bit word may use 8-bits for the exponent in Eq. (I-6), and represent it as an integer. This 8-bit integer "exponent" has a range of [0:255]. Numbers with actual negative exponents are represented by a bias equal to 127, a fixed number for a given machine. Consequently, the exponent has the range [-127:128] even

though the value stored in the exponent in Eq. (I-6) is a positive number.

• Of the remaining bits, one is use for the sign and the other 23 for the mantissa, where

$$\begin{aligned} \text{mantissa} &= (m_1 \times 2^{-1}) + (m_2 \times 2^{-2}) + \\ &\quad \cdots + (m_{23} \times 2^{-23}) \ , \end{aligned} \tag{F-7}$$

with the m_i stored liked the α_i in Eq. (I-5).

• As an example, the number 0.5 is stored as

$$0 \quad 0111 \ \ 1111 \quad 1000 \ \ 0000 \ \ 0000 \ \ 0000 \ \ 0000 \ \ 000$$

on a 32-bit machine, where the bias is $0111 \ \ 1111_2 = 127_{10}$.

c) Typically, the largest possible floating-point number for a 32-bit machine is

$$0 \quad 1111 \ \ 1111 \quad 1111 \ \ 1111 \ \ 1111 \ \ 1111 \ \ 1111 \ \ 111 \ ,$$

which has the value 1 for all its bits (except the sign) and adds up to $2^{128} = 3.4 \times 10^{38}$. Whereas the smallest possible floating-point number for a 32-bit machine is

$$0 \quad 0000 \ \ 0000 \quad 1000 \ \ 0000 \ \ 0000 \ \ 0000 \ \ 0000 \ \ 000 \ ,$$

which has the value 0 for almost all the bits and adds up to $2^{-128} = 2.9 \times 10^{-39}$. As built in by the use of the bias, the smallest number possible to store is the inverse of the largest.

7. Just as was the case for integers, reals come in a variety of "flavors" in various programming languages. For example:

a) **Single-Precision Real** (sometimes called *Floats* or *Real∗4* numbers): 32-bit = 4-byte word-size numbers, 6-7 decimal places of precision (1 part in 2^{23}), and a range = $[1.17549435 \times 10^{-38} : 3.40282347 \times 10^{38}]$.

b) **Double-Precision Real** (sometimes called just *Double Precision* or *Real∗8*): 64-bit (2 32-bit words → 11 bits used for the exponent, 1 for the sign, and 52 bits for the mantissa) = 8-byte word-size numbers, about 16 decimal places of precision (1 part in 2^{52}), and a range = $[2.2250738585072014\times10^{-308} : 1.7976931348623157\times10^{308}]$.

c) **Quad-Precision Real** (*Real∗16*): 128-bit (4 32-bit words → 15 bits used for the exponent, 1 for the sign, and 112 bits for the mantissa) = 16-byte word-size numbers, about 36 decimal places of precision (1 part in 2^{112}), and a range = $[\sim 3.362 \times 10^{-4932} : \sim 1.189 \times 10^{4932}]$.

d) **Complex Numbers**: Some programming languages (in particular, Fortran and IDL) have complex data types. A complex number has the form

$$z = x + iy \ , \tag{F-8}$$

where $i \equiv \sqrt{-1}$ and x corresponds to the real part of the number and y to the imaginary part. These numbers are stored as two-element arrays (real, imaginary) of real numbers.