Floating Point

Problem Description

Computers use floating-point representation to store real numbers (such as 3.14159. . . , 1.06×1037, -2.5) as fixed-length sequences of bits. The way that these bits are interpreted is known as the computer's floating-point format. Choosing a format involves trading off storage needed per number against the precision with which numbers can be represented, so a computer may support more than one format. Most modern computers use the two basic formats defined in IEEE Standard 754-1985[1], which are known as IEEE single-precision and IEEE double-precision respectively.

However, other floating-point formats have been in use since at least the early 1950s. One important set of formats is that introduced in 1964 with IBM's System/360 family of mainframes and used in the mainframe world ever since. System/360 also supports single- and double-precision formats, but naturally these are incompatible with the much later IEEE standard.

Task

You have been contracted to help a client migrate their IT operations from an IBM mainframe to a new desktop-based system. They have a large amount of numeric data on the mainframe which needs to be converted into a format the desktop boxes will understand. Your task is to produce a robust, carefully tested program which will read a file of IBM System/360-format floating point numbers and write them to a new file in IEEE standard format. Extracts from the documents which define the formats are attached.

Your program should prompt for input and output filenames, and the precision (single or double) of each file. Note that the two precision specifications are independent, so you will have to consider all four combinations. You may need to talk to the client about how to handle exceptional cases. There are some little utility programs in Documents:Ass Data&Progs on the 326 server which might be helpful.

[1]The IEEE Standard 754-1985 has been superseded by IEEE Standard 754-2008, but for this exercise we will stick with the 1985 version.

Relates to Objectives

1.2, 1.4, 2.2, 2.5, 2.7, 2.8, 3.1, 3.5, 4.3, 4.4

(Group 2)

File No. S360-01
Form A22-6821-6

IBM Systems Reference Library

IBM System/360 Principles of Operation

This publication is the machine reference manual for the IBM System/360. It provides a direct, comprehensive description of the system structure; of the arithmetic, logical, branching, status switching, and input/output operations; and of the interruption system.

The reader is assumed to have a basic knowledge of data processing systems and to have read the *IBM System/360 System Summary*, Form A22-6810, which describes the system briefly and discusses the input/output devices available.

For information about the characteristics, functions, and features of a specific System/360 model, use the functional characteristics manual for that model in conjunction with the *IBM System/360 Principles of Operation*. Descriptions of specific input/output devices used with the System/360 appear in separate publications. Publications that relate to the IBM System/360 Model 20 are described in the *IBM System/360 Model 20 Bibliography*, Form A26-3565. Other IBM Systems Reference Library publications concerning the System/360 are identified and described in the *IBM System/360 Bibliography*, Form A22-6822.

Floating-Point Arithmetic

The floating-point instruction set is used to perform calculations on operands with a wide range of magnitude and yielding results scaled to preserve precision.

A floating-point number consists of a signed exponent and a signed fraction. The quantity expressed by this number is the product of the fraction and the number 16 raised to the power of the exponent. The exponent is expressed in excess 64 binary notation; the fraction is expressed as a hexadecimal number having a radix point to the left of the high-order digit.

To avoid unnecessary storing and loading operations for results and operands, four floating-point registers are provided. The floating-point instruction set provides for loading, adding, subtracting, comparing, multiplying, dividing, and storing, as well as the sign control of short or long operands. Short operands generally provide faster processing and require less storage than long operands. On the other hand, long operands provide greater preciseness in computation. Operations may be either register to register or storage to register. All floating-point instructions and registers are part of the floating-point feature.

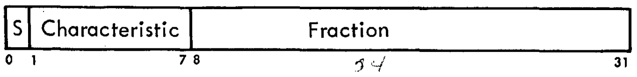
Maximum precision is preserved in addition, subtraction, multiplication, and division by producing normalized results. For addition and subtraction, instructions are also provided that generate unnormalized results. Normalized and unnormalized operands may be used in any floating-point operation.

The condition code is set as a result of all sign control, add, subtract, and compare operations.

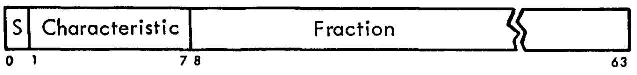
Data Format

Floating-point data occupy a fixed-length format, which may be either a fullword short format or a double-word long format. Both formats may be used in main storage and in the floating-point registers. The floating-point registers are numbered 0, 2, 4, and 6.

Short Floating-Point Number



Long Floating-Point Number



The first bit in either format is the sign bit (S). The subsequent seven bit positions are occupied by the characteristic. The fraction field may have either six or 14 hexadecimal digits.

The entire set of floating-point instructions is available for both short and long operands. When short-precision is specified, all operands and results are 32-bit floating-point words, and the rightmost 32 bits of the floating-point registers do not participate in the operations and remain unchanged. An exception is the product in MULTIPLY, which is a 64-bit word and occupies a full register. When long-precision is specified, all operands and results are 64-bit floating-point words.

Although final results in short-precision have six fraction digits, intermediate results in ADD NORMALIZED, SUBTRACT NORMALIZED, ADD UNNORMALIZED, SUBTRACT UNNORMALIZED, and COMPARE may extend to seven fraction digits. The low-order digit of a seven-digit fraction is called the *guard digit* and serves to increase the precision of the final result. Intermediate results in long-precision do not exceed 14 fraction digits.

Number Representation

The fraction of a floating-point number is expressed in hexadecimal digits. The radix point of the fraction is assumed to be immediately to the left of the high-order fraction digit. To provide the proper magnitude for the floating-point number, the fraction is considered to be multiplied by a power of 16. The characteristic portion, bits 1-7 of both floating-point formats, indicates this power. The bits within the characteristic field can represent numbers from 0 through 127. To accommodate large and small magnitudes, the characteristic is formed by adding 64 to the actual exponent. The range of the exponent is thus –64 through +63. This technique produces a characteristic in excess 64 notation.

Both positive and negative quantities have a true fraction, the difference in sign being indicated by the sign bit. The number is positive or negative accordingly as the sign bit is zero or one.

The range covered by the magnitude (M) of a normalized floating-point number is
in short precision $16^{-65} \leq M \leq (1 - 16^{-6}) \cdot 16^{63}$, and
in long precision $16^{-65} \leq M \leq (1 - 16^{-14}) \cdot 16^{63}$,
or approximately $5.4 \cdot 10^{-79} \leq M \leq 7.2 \cdot 10^{75}$
in either precision.

A number with zero characteristic, zero fraction, and plus sign is called a true zero. A true zero may arise as the result of an arithmetic operation because of the particular magnitude of the operands. A result is forced to be true zero when an exponent underflow occurs or when a result fraction is zero and no program interruption due to significance exception is taken. When the program interruption is taken, the true zero is not forced, and the characteristic of the result remains unchanged. Whenever a result has a zero fraction, the exponent overflow and underflow exceptions do not cause a program interruption. When a divisor has a zero fraction, division is omitted, a floating-point divide exception exists, and a program interruption occurs. Otherwise, zero fractions and zero characteristics participate as normal numbers in all arithmetic operations.

The sign of a sum, difference, product, or quotient with zero fraction is positive. The sign of a zero fraction resulting from other operations is established by the rules of algebra from the operand signs.

Normalization

A quantity can be represented with the greatest precision by a floating-point number of given fraction length when that number is normalized. A normalized floating-point number has a nonzero high-order hexadecimal fraction digit. If one or more high-order fraction digits are zero, the number is said to be unnormalized. The process of normalization consists of shifting the fraction left until the high-order hexadecimal digit is nonzero and reducing the characteristic by the number of hexadecimal digits shifted. A zero fraction can not be normalized, and its associated characteristic therefore remains unchanged when normalization is called for.

Normalization usually takes place when the intermediate arithmetic result is changed to the final result. This function is called *postnormalization.* In performing multiplication and division, the operands are normalized prior to the arithmetic process. This function is called *prenormalization.*

Floating-point operations may be performed with or without normalization. Most operations are performed in only one of these two ways. Addition and subtraction may be specified either way.

When an operation is performed without normalization, high-order zeros in the result fraction are not eliminated. The result may or may not be normalized, depending upon the original operands.

In both normalized and unnormalized operations, the initial operands need not be in normalized form. Also, intermediate fraction results are shifted right when an overflow occurs, and the intermediate fraction result is truncated to the final result length after the shifting, if any.

Programming Note

Since normalization applies to hexadecimal digits, the three high-order bits of a normalized number may be zero.

Condition Code

The results of floating-point sign-control, add, subtract, and compare operations are used to set the condition code. Multiplication, division, loading, and storing leave the code unchanged. The condition code can be used for decision-making by subsequent branch-on-condition instructions.

The condition code can be set to reflect two types of results for floating-point arithmetic. For most operations, the states 0, 1, or 2 indicate the content of the result register is zero, less than zero, or greater than zero. A zero result is indicated whenever the result fraction is zero, including a forced zero. State 3 is used when the exponent of the result overflows.

For comparison, the states 0, 1, or 2 indicate that the first operand is equal, low, or high.

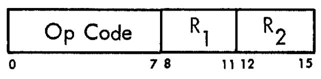
CONDITION CODE SETTING FOR FLOATING-POINT ARITHMETIC

	0	1	2	3
Add Normalized S/L	zero	$<$ zero	$>$ zero	overflow
Add Unnormalized S/L	zero	$<$ zero	$>$ zero	overflow
Compare S/L	equal	low	high	--
Load and Test S/L	zero	$<$ zero	$>$ zero	--
Load Complement S/L	zero	$<$ zero	$>$ zero	--
Load Negative S/L	zero	$<$ zero	--	--
Load Positive S/L	zero	--	$>$ zero	--
Subtract Normalized S/L	zero	$<$ zero	$>$ zero	overflow
Subtract Unnormalized S/L	zero	$<$ zero	$>$ zero	overflow

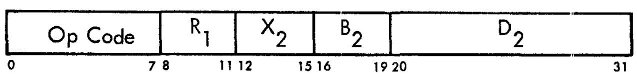
Instruction Format

Floating-point instructions use the following two formats:

RR Format



RX Format



In these formats, R_1 designates the address of a floating-point register. The contents of this register will be

3. Formats

This standard defines four floating-point formats in two groups, basic and extended, each having two widths, single and double. The standard levels of implementation are distinguished by the combinations of formats supported.

3.1 Sets of Values

This section concerns only the numerical values representable within a format, not the encodings. The only values representable in a chosen format are those specified by way of the following three integer parameters:

p	= the number of significant bits (precision)
$E_{\max}$	= the maximum exponent
$E_{\min}$	= the minimum exponent

Each format's parameters are given in Table 1. Within each format only the following entities shall be provided:

Numbers of the form $(-1)^s 2^E(b_0 \cdot b_1 b_2 \ldots b_{p-1})$

where

s	= 0 or 1
E	= any integer between $E_{\min}$ and $E_{\max}$, inclusive
b_i	= 0 or 1

Two infinities, $+\infty$ and $-\infty$

At least one signaling NaN

At least one quiet NaN

The foregoing description enumerates some values redundantly, for example, $2^0(1 \cdot 0) = 2^1\ (0 \cdot 1) = 2^2(0 \cdot 01) = \ldots$. However, the encodings of such nonzero values may be redundant only in extended formats (3.3). The nonzero values of the form $\pm 2^{E_{\min}}(0 \cdot b_1 b_2 \ldots b_{p-1})$ are called denormalized. Reserved exponents may be used to encode NaNs, $\pm\infty$, ± 0, and denormalized numbers. For any variable that has the value zero, the sign bit s provides an extra bit of information. Although all formats have distinct representations for +0 and −0, the signs are significant in some circumstances, such as division by zero, and not in others. In this standard, 0 and ∞ are written without a sign when the sign is not important.

Table 1— Summary of Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
$E_{\max}$	+127	≥ +1023	+1023	≥ +16383
$E_{\min}$	−126	≤ −1022	−1022	≤ −16382
Exponent *bias*	+127	unspecified	+1023	unspecified
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

3.2 Basic Formats

Numbers in the single and double formats are composed of the following three fields:

1) **1-bit sign s**
2) **Biased exponent $e = E+bias$**
3) **Fraction $f = \cdot\, b_1 b_2 \ldots b_{p-1}$**

The range of the unbiased exponent E shall include every integer between two values $E_{\min}$ and $E_{\max}$, inclusive, and also two other reserved values $E_{\min}-1$ to encode ±0 and denormalized numbers, and $E_{\max}+1$ to encode ±∞ and NaNs. The foregoing parameters are given in Table 1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

3.2.1 Single

A 32-bit single format number X is divided as shown in Fig 1. The value v of X is inferred from its constituent fields thus

1) If $e = 255$ and $f \neq 0$, then v is NaN regardless of s
2) If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$
3) If $0 < e < 255$, then $v = (-1)^s 2^{e-127}(1 \cdot f)$
4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-126}(0 \cdot f)$ (denormalized numbers)
5) If $e = 0$ and f = 0, then $v = (-1)^s 0$ (zero)

3.2.2 Double

A 64-bit double format number X is divided as shown in Fig 2. The value v of X is inferred from its constituent fields thus

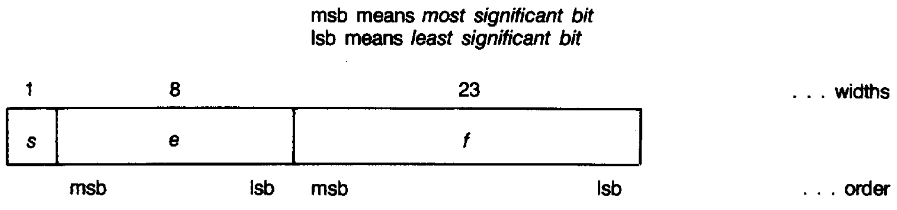

msb means *most significant bit*
lsb means *least significant bit*
1
8
23
. . . widths
s
e
f
msb
lsb
msb
lsb
. . . order

Figure 1— Single Format

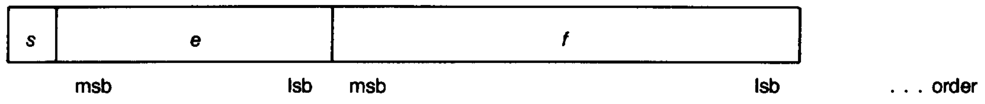


Figure 2— Double Format

1) If $e = 2047$ and $f \neq 0$, then v is NaN regardless of s
2) If $e = 2047$ and $f = 0$, then $v = (-1)^s \infty$
3) If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023}(1 \cdot f)$
4) If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{-1022}(0 \cdot f)$ (denormalized numbers)
5) If $e = 0$ and $f = 0$, then $v = (-1)^s\, 0$ (zero)