Chapter 13: Handling Floating Point Data

This chapter presents a brief discussion of floating point data and arithmetic. There are a few reasons to cover this topic, but none relate to the expectation that a programmer will actually use this data format in an assembly language program.

The two primary reasons to cover floating point data and arithmetic are simple:

1) So that we can make a plausible claim of a complete coverage of assembler language.
2) So that the student will more fully appreciate some of the complexities (handled by the run–time system of a HLL) of processing floating point data.

There is another reason for the study; this is one that is found in early texts on assembler language. It used to be the case that an understanding of floating point format would allow the programmer to write a High–Level–Language program that was considerably more efficient. With modern compilers, this is rarely the case.

The first thing to note in this chapter is the difference between floating point data and fixed point data. It is simple: in the first format the decimal point may be said to float in that it can assume any position in the data. In the fixed point data, the decimal point is fixed at a predefined position. The packed decimal format is a good example of a fixed–point format.

Consider the two packed decimal numbers defined as follows:
```
N1      DC P‘1234’  Stored as X‘01234C’
N2      DC P‘567’   Stored as X‘567C’
```

From the viewpoint of assembler language, each of these labels is simply a reference to data in packed decimal format. There is more to the definition than what is stated above. One must view each declaration in terms of a data type, which is a concept taken from a HLL. In particular, the data are defined completely only if the number of decimal places is specified.

So, we must ask if the numbers represented by the labels **N1** and **N2** are of compatible data type. Again, this cannot be determined from the simple definition. If both values are to be interpreted with the same number of decimal points, the types are compatible.

Suppose that **N1** represents the number 12.34.
Suppose that **N2** represents the number 5.67.
The two definitions can be viewed as belonging to a single data type.

Suppose, however that **N1** represents the number 1.234, while **N2** represents 5.67. Then the definitions belong to what might be called similar, but incompatible, data types. One of the two must be modified by a **SRP** instruction before they are added, or the results will be silly. One possible redefinition would be as follows:
```
N1A     DC P‘1234’  Stored as X‘01234C’ for 1.234
N2A     DC P‘5670’  Stored as X‘05670C’ for 5.670
```

In the above definition, each constant is defined with three implicit decimal places, and the two can be said to be of a common data type. We must emphasize that the idea of a data type is derived from the theory of high–level compiled languages and is somewhat foreign to assembler language. However, it is very helpful in understanding fixed point arithmetic.

The basic idea of floating point arithmetic is the explicit storage of a representation of the position of the decimal point in each data item. That way, the CPU can automatically adjust the two to the same basic representation, if that is necessary. As we shall see, this adjustment is required for addition and subtraction, but not multiplication and division.

Consider the above two numbers at labels **`N1`** and **`N2`**. Suppose that these are to be interpreted as 1.234 and 56.7 respectively. In floating–point notation, these would be represented as $1.234 \bullet 10^0$ and $5.67 \bullet 10^1$, respectively.

Addition might proceed by converting these to $1.234 \bullet 10^0$ and $56.7 \bullet 10^0$, then adding to get $57.934 \bullet 10^0$, which might be converted to either $5.7934 \bullet 10^1$ or $5.793 \bullet 10^1$. Multiplication will proceed without adjusting either value: $1.234 \bullet 10^0 \bullet 5.67 \bullet 10^1 = 6.99678 \bullet 10^1$, which might be rounded to $7.00 \bullet 10^1$, $6.997 \bullet 10^1$, or some other value.

As noted before, the floating point format has an advantage, which is simultaneously its disadvantage. The advantage is that the format handles the position of the decimal point explicitly. The disadvantage is that it might round and thereby lose precision.

At this point, the reader should review the material on floating point formats that is found in Chapter 4 (Data Representation) of this textbook. The topics for review should be:

1. Conversion between decimal format and hexadecimal representations,
2. Excess–64 representation of integers,
3. normalized and denormalized floating point numbers, and
4. the IBM Mainframe floating point formats.

For the readers convenience, the last topic will be summarized below.

The IBM Mainframe Floating–Point Formats

In this discussion, we shall adopt the bit numbering scheme used in the IBM documentation, with the leftmost (sign) bit being number 0. The IBM Mainframe supports three formats; those representations with more bits can be seen to afford more precision.

Single precision	32 bits	numbered 0 through 31,
Double precision	64 bits	numbered 0 through 63, and
Extended precision	128 bits	numbered 0 through 127.

As in the IEEE–754 standard, each floating point number in this standard is specified by three fields: the sign bit, the exponent, and the fraction. Unlike the IEEE–754 standard, the IBM standard allocates the same number of bits for the exponent of each of its formats. The bit numbers for each of the fields are shown below.

Format	Sign bit	Bits for exponent	Bits for fraction
Single precision	0	1 – 7	8 – 31
Double precision	0	1 – 7	8 – 63
Extended precision	0	1 – 7	8 – 127

Note that each of the three formats uses eight bits to represent the exponent, in what is called the **characteristic field**, and the sign bit. These two fields together will be represented by two hexadecimal digits in a one–byte field. The size of the fraction field does depend on the format.

Single precision	24 bits	6 hexadecimal digits,
Double precision	56 bits	14 hexadecimal digits, and
Extended precision	120 bits	30 hexadecimal digits.

The Characteristic Field

In IBM terminology, the field used to store the representation of the exponent is called the **"characteristic"**. This is a 7–bit field, used to store the exponent in excess–64 format; if the exponent is E, then the value (E + 64) is stored as an unsigned 7–bit number.

Recalling that the range for integers stored in 7–bit unsigned format is $0 \le N \le 127$, we have $0 \le (E + 64) \le 127$, or $-64 \le E \le 63$.

Range for the Standard

We now consider the range and precision associated with the IBM floating point formats. The reader should remember that the range is identical for all of the three formats; only the precision differs. The range is usually specified as that for positive numbers, from a very small positive number to a large positive number. There is an equivalent range for negative numbers. Recall that 0 is not a positive number, so that it is not included in either range.

Given that the base of the exponent is 16, the range for these IBM formats is impressive. It is from somewhat less than 16^{-64} to a bit less than 16^{63}. Note that $16^{63} = (2^4)^{63} = 2^{252}$, and $16^{-64} = (2^4)^{-64} = 2^{-256} = 1.0 / (2^{256})$ and recall that $\log_{10}(2) = 0.30103$. Using this, we compute the maximum number storable at about $(10^{0.30103})^{252} = 10^{75.86} \approx 9\bullet10^{75}$. We may approximate the smallest positive number at $1.0 / (36\bullet10^{75})$ or about $3.0\bullet10^{-77}$. In summary, the following real numbers can be represented in this standard: $X = 0.0$ and $3.0\bullet10^{-77} < X < 9\bullet10^{75}$.

One would not expect numbers outside of this range to appear in any realistic calculation.

Precision for the Standard

Unlike the range, which depends weakly on the format, the precision is very dependent on the format used. More specifically, the precision is a direct function of the number of bits used for the fraction. If the fraction uses F bits, the precision is 1 part in 2^F.

We can summarize the precision for each format as follows.

Single precision	F = 24	1 part in 2^{24}.
Double precision	F = 56	1 part in 2^{56}.
Extended precision	F = 120	1 part in 2^{120}.

The first power of 2 is easily computed; we use logarithms to approximate the others.

$$2^{24} \qquad = 16{,}777{,}216$$
$$2^{56} \approx (10^{0.30103})^{56} = 10^{16.85} \approx 9\bullet10^{16}.$$
$$2^{120} \approx (10^{0.30103})^{120} = 10^{36.12} \approx 1.2\bullet10^{36}.$$

The argument for precision is quite simple. Consider the single precision format, which is more precise than 1 part in 10,000,000 and less precise than 1 part in 100,000,000. In other words it is better than 1 part in 10^7, but not as good as 1 in 10^8; hence we say 7 digits.

Range and Precision

We now summarize the range and precision for the three IBM Mainframe formats.

Format	**Type**	**Positive Range**	**Precision**
Single Precision	E	$3.0\bullet10^{-77} < X < 9\bullet10^{75}$	7 digits
Double Precision	D	$3.0\bullet10^{-77} < X < 9\bullet10^{75}$	16 digits
Extended Precision	L	$3.0\bullet10^{-77} < X < 9\bullet10^{75}$	36 digits

Representation of Floating Point Numbers

As with the case of integers, we shall most commonly use hexadecimal notation to represent the values of floating–point numbers stored in the memory. From this point, we shall focus on the two more commonly used formats: Single Precision and Double Precision.

The single precision format uses a 32–bit number, represented by 8 hexadecimal digits. The double precision format uses a 64–bit number, represented by 16 hexadecimal digits.

Due to the fact that the two formats use the same field length for the characteristic, conversion between the two is quite simple. To convert a single precision value to a double precision value, just add eight hexadecimal zeroes.

Consider the positive number 128.0.
As a single precision number, the value is stored as `4280 0000`.
As a double precision number, the value is stored as `4280 0000 0000 0000`.

Conversions from double precision to single precision format will involve some rounding. For example, consider the representation of the positive decimal number 123.45. In a few pages, we shall show that it is represented as follows.

As a double precision number, the value is stored as `427B 7333 3333 3333`.
As a single precision number, the value is stored as `427B 7333`.

The Sign Bit and Characteristic Field

We now discuss the first two hexadecimal digits in the representation of a floating–point number in these two IBM formats. In IBM nomenclature, the bits are allocated as follows.

Bit 0	the sign bit
Bits 1 – 7	the seven–bit number storing the characteristic.

Bit Number	0	1	2	3	4	5	6	7
Hex digit	0				1			
Use	Sign bit	Characteristic (Exponent + 64)						

Consider the four bits that comprise hexadecimal digit 0. The sign bit in the floating–point representation is the "8 bit" in that hexadecimal digit. This leads to a simple rule.

If the number is not negative, bit 0 is 0, and hex digit 0 is one of 0, 1, 2, 3, 4, 5, 6, or 7.
If the number is negative, bit 0 is 1, and hex digit 0 is one of 8, 9, A, B, C, D, E, or F.

Some Single Precision Examples

We now examine a number of examples, using the IBM single–precision floating–point format. The reader will note that the methods for conversion from decimal to hexadecimal formats are somewhat informal, and should check previous notes for a more formal method. Note that the first step in each conversion is to represent the **magnitude** of the number in the required form X•16^E, after which we determine the sign and build the first two hex digits.

Example 1: True 0

The number 0.0, called "true 0" by IBM, is stored as all zeroes [R_15, page 41].
In single precision it would be **`0000 0000`**.
In double precision it would be **`0000 0000 0000 0000`**.

Example 2: Positive exponent and positive fraction.
The decimal number is 128.50. The format demands a representation in the form $X\bullet 16^E$, with $0.625 \leq X < 1.0$. As $128 \leq X < 256$, the number is converted to the form $X\bullet 16^2$.
Note that $128 = (1/2)\bullet 16^2 = (8/16)\bullet 16^2$, and $0.5 = (1/512)\bullet 16^2 = (8/4096)\bullet 16^2$.
Hence, the value is $128.50 = (8/16 + 0/256 + 8/4096)\bullet 16^2$; it is $16^2\bullet$0x0.808.

The exponent value is 2, so the characteristic value is either 66 or 0x42 = 100 0010. The first two hexadecimal digits in the eight digit representation are formed as follows.

Field	Sign	Characteristic						
Value	0	1	0	0	0	0	1	0
Hex value	4				2			

The fractional part comprises six hexadecimal digits, the first three of which are 808.
The number 128.50 is represented as **`4280 8000`**.

Example 3: Positive exponent and negative fraction.
The decimal number is the negative number –128.50. At this point, we would normally convert the magnitude of the number to hexadecimal representation. This number has the same magnitude as the previous example, so we just copy the answer; it is $16^2\bullet$0x0.808.

We now build the first two hexadecimal digits, noting that the sign bit is 1.

Field	Sign	Characteristic						
Value	1	1	0	0	0	0	1	0
Hex value	C				2			

The number 128.50 is represented as **`C280 8000`**.
Note that we could have obtained this value just by adding 8 to the first hex digit.

Example 4: Negative exponent and positive fraction.
The decimal number is 0.375. As a fraction, this is 3/8 = 6/16. Put another way, it is $16^0\bullet 0.375 = 16^0\bullet(6/16)$. This is in the required format $X\bullet 16^E$, with $0.625 \leq X < 1.0$.

The exponent value is 0, so the characteristic value is either 64 or 0x40 = 100 0000. The first two hexadecimal digits in the eight digit representation are formed as follows.

Field	Sign	Characteristic						
Value	0	1	0	0	0	0	0	0
Hex value	4				0			

The fractional part comprises six hexadecimal digits, the first of which is a 6.
The number 0.375 is represented in single precision as **`4060 0000`**.
The number 0.375 is represented in double precision as **`4060 0000 0000 0000`**.

Example 5: A Full Conversion
The number to be converted is 123.45. As we have hinted, this is a non–terminator.

Convert the integer part.
`123 / 16 = 7` with remainder `11` this is hexadecimal digit B.
`  7 / 16 = 0` with remainder `7` this is hexadecimal digit 7.
Reading bottom to top, the integer part converts as 0x7B.

Convert the fractional part.
`0.45 • 16 = 7.20` Extract the 7,
`0.20 • 16 = 3.20` Extract the 3,
`0.20 • 16 = 3.20` Extract the 3,
`0.20 • 16 = 3.20` Extract the 3, and so on.
In the standard format, this number is 16^{2}•0x0.7B33333333…...

The exponent value is 2, so the characteristic value is either 66 or 0x42 = 100 0010. The first two hexadecimal digits in the eight digit representation are formed as follows.

Field	Sign	Characteristic						
Value	0	1	0	0	0	0	1	0
Hex value	4				2			

The number 123.45 is represented in single precision as **`427B 3333`**.
The number 0.375 is represented in double precision as **`427B 3333 3333 3333.`**

Example 5: One in "Reverse"
We are given the single precision representation of the number. It is `4110 0000`.
What is the value of the number stored? We begin by examination of the first two hex digits.

Field	Sign	Characteristic						
Value	0	1	0	0	0	0	0	1
Hex value	4				1			

The sign bit is 0, so the number is positive. The characteristic is 0x41, so the exponent is 1 and the value may be represented by X•16^{1}. The fraction field is `100 000`, so the value is 16^{1}•(1/16) = 1.0.

Example 6: Another in "Reverse"
We are given the single precision representation of the number. It is `BEC8 0000`.
What is the value of the number stored? We begin by examination of the first two hex digits.

Field	Sign	Characteristic						
Value	1	0	1	1	1	1	1	0
Hex value	B				E			

The characteristic has value 0x3E or decimal 3•16 + 14 = 62. The exponent has value 62 – 64 = –2. The number is then 16^{-2}•0x0.C8 = 16^{-2}•(12/16 + 8/256), which can be converted to decimal in any number of ways. I prefer the following conversion.
16^{-2}•(12/16 + 8/256) = 16^{-2}•(3/4 + 1/32) = 16^{-2}•(24/32 + 1/32) = 16^{-2}•(25/32)
= 25 / (32•256) = 25 / 8192 ≈ 3.0517578•10^{-3}.
The answer is approximately the negative number **–3.0517578•10^{-3}**.

Why Excess–64 Notation for the Exponent?

We have introduced two methods to be used for storing signed integers: two's–complement notation and excess–64 notation. One might well ask why two's-complement notation is not used to store the exponent in the characteristic field.

The answer for integer notation is simple. Consider some of examples.

 `128.50` is represented as **`4280 8000`**. Viewed as a 32–bit integer, this is positive.
`-128.50` is represented as **`C280 8000`**. Viewed as a 32–bit integer, this is negative.
  `1.00` is represented as **`4110 0000.`** Viewed as a 32–bit integer, this is positive.
$-3.05 \bullet 10^{-3}$ is represented as **`BEC8 0000.`** Viewed as a 32–bit integer, this is negative

It turns out that the excess–64 notation allows the use of the integer compare unit to compare floating point numbers. Consider two floating point numbers X and Y. Pretend that they are integers and compare their bit patterns as integer bit patterns. It viewed as an integer, X is less than Y, then the floating point number X is less than the floating point Y. Note that we are not converting the numbers to integer form, just looking at the bit patterns and pretending that they are integers. For example, the above examples would yield the following order.

`4280 8000` for `128.50`. This is the largest.
`4110 0000` for `1.00`.
`BEC8 0000` for $-3.05 \bullet 10^{-3}$.
`C280 8000` for `–128.50`. This is the most smallest (most negative).

Examples of Floating–Point Declaratives

Floating point storage and constants are defined with the standard DS and DC declaratives. There are three possible formats: E (Single Precision), D (Double Precision) and L (Extended Precision). Standard programs use the E (Single Precision) format, with occasional use of the D (Double Precision) format. The L format is probably unusual.

Here are some examples of floating–point declaratives.

```
FL1  DS E          This defines a 4–byte storage area, aligned
                   on a fullword boundary.  Presumably, it
                   will store Single Precision Data.
DL1  DS D          An 8-byte storage area, aligned on a double
                   word boundary.  It could store data in
                   Double Precision format.
FL2  DS E'12.34'   Define a single precision value.
FL3  DS E'-12.34'  The negative of the above value.
DL2  DS D'0.0'     The constant 0.0, in double precision.
```

The reader may recall that we have used statements, such as the **`DL1`** declarative, to reserve an 8–byte storage area aligned on a double–word boundary, for use in the **`CVB`** and **`CVD`** instructions associated with packed decimal arithmetic. This emphasizes the fact that the declaratives do not really determine a data type, but just set aside storage. In assembler language, it is the instructions that determine the data type.

Before proceeding, we must give a disclaimer. There are a few features of floating–point arithmetic, such as the exponent modifier and scale modifier, that will not be covered in this discussion. This discussion will also be limited to normalized floating–point numbers and totally ignore the unnormalized instructions which handle data not in normalized format.

The Floating–Point Registers
In addition to the sixteen general–purpose registers (used for binary integer arithmetic), the S/360 architecture provides four registers dedicated for floating–point arithmetic. These registers are numbered 0, 2, 4, and 6(*). Each is a 64–bit register. It is possible that the use of even numbers to denote these registers is to emphasize that they are not 32–bit registers.

The use of the registers by the floating–point operations depends on the precision.
Single precision formats use the leftmost 32 bits of a floating–point register.
Double precision formats use all 64 bits of the register.
Extended precision formats use two adjacent registers for a 128 bit number.

In order to understand why it is the leftmost 32 bits of the register that are used for the 32–bit single precision floating–point format, we must consider what is involved in extending the single precision format to a double precision format.

Consider the single–precision constant 123.45, which would be represented by the 32–bit (8 hexadecimal digit) constant **427B 3333**. Were this to be extended to double precision, it would be stored as the 64–bit constant **427B 3333 0000 0000**. In other words, each floating point register stores a value as if it were a double–precision value; the single–precision values being stored as values with limited precision.

As we saw above, a conversion of 123.45 directly to the double–precision floating–point format would yield the value **427B 3333 3333 3333**, rather than the truncated value **427B 3333 0000 0000** seen above. Put another way, single–precision values stored as double precision are not as precise as true double–precision constants.

To make this point completely obvious, suppose that a 64–bit floating–point register contains the following value, expressed as 16 hexadecimal digits.

Byte	0	1	2	3	4	5	6	7
Value	42	7B	33	33	33	33	33	33

An E format (Single Precision) floating–point reference to this register would access only the leftmost four bytes and use the value **427B 3333**. A D format (Double Precision) floating–point reference would access all eight bytes and use the value **427B 3333 3333 3333**. Each reference should correspond to the decimal value 123.45, just with different precision.

As a side note, some early versions of FORTRAN might print the above number at something like 123.449999, due to the fact that 123.45 cannot be represented exactly as a floating–point number. The FORMAT statement provided by the FORTRAN run–time system was modified to round this value to 123.45.

* Modern System/z architecture supports 16 floating–point registers, numbered 0 – 15. The additional registers (1, 3, 5, and 7 – 15) are useable if the AFPR option has been selected in the ACONTROL instruction for the code. [R-17, page 100]

The Floating–Point Instructions
The floating–point instruction set is less extensive than the binary integer instruction set, but bears some similarities. The instructions are in two formats: RR (Register to Register) and RX (Register and Indexed Storage). Obviously the register reference in these instructions are the floating–point registers. The mnemonics for the instructions are structured into three parts: a one or two character code for the operation, followed by a character indicating the precision, then either a blank for type RX or an **'R'** for type RR.

The Load Instructions
The load instructions load a 64–bit floating point register from either storage or another floating–point register. The valid register numbers are 0, 2, 4, or 6.

```
LE R1,D2(X2,B2)      Load R1 single precision from memory
                     Operand 2 is an aligned fullword;
                     its address is a multiple of 4.
LD R1,D2(X2,B2)      Load R1 double precision from memory
                     Operand 2 is an aligned double word;
                     its address is a multiple of 8.
LER R1,R2            Load the leftmost 32 bits of R1
                     from the leftmost 32 bits of R2.
LDR R1,R2            Load the 64-bit register R1 from
                     the 64-bit register R2.
```

Neither **LE** or **LER** change the rightmost 32 bits of the target floating–point register.

The opcodes for the two type RR instructions are as follows:

LER	**X'38'**	LDR	**X'28'**

The object code format for these type RR instructions follows the standard. Each is a two–byte instruction of the form **OP R1,R2**.

Type	Bytes	Operands		
RR	2	R1,R2	OP	R_1 R_2

The first byte contains the 8–bit instruction code.
The second byte contains two 4–bit fields, each of which encodes a register number.
This instruction format is used to process data between registers.

The opcodes for the two type RX instructions are as follows:

LE	**X'78'**	LD	**X'68'**

Each is a four–byte instruction of the form **OP R1,D2(X2,B2)**.

Type	Bytes	Operands	1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code. The second byte contains two 4–bit fields, each of which encodes a register number. The third and fourth bytes contain an address in the standard base/displacement with index register format. The load instructions do not set any condition code.

The Store Instructions

There are two store instructions for storing either the leftmost 32 bits or all 64 bits of the 64 bit floating–point registers. Again, the valid register numbers are 0, 2, 4, or 6.

STE R1,D2(X2,B2)	**Store the 32 leftmost bits of register R1 as a single precision result into the aligned fullword address.**
STD R1,D2(X2,B2)	**Store the 64 bits of register R1 as a double precision result into the aligned double word address.**

The opcodes for these two instructions are as follows:
STE **X'70'** STD **X'60'.**

Each is a four–byte instruction of the form **OP R1,D2(X2,B2)**.

Type	Bytes	Operands	1	2	3	4
RX	4	R1,D2(X2,B2)	OP	R_1 X_2	B_2 D_2	D_2D_2

The first byte contains the 8–bit instruction code. The second byte contains two 4–bit fields, each of which encodes a register number. The third and fourth bytes contain an address in the standard base/displacement with index register format.

The rules for forming the address of operand 2 in each of the above instructions follow the standard for type RX instructions. Again, the only new feature is that the address of the operand must be properly aligned. In practice that means using the proper declarative for specifying the storage.

Single precision	Use **DS E** or **DC E**, to insure that the address is a multiple of 4.
Double precision	Use **DS D** or **DC D**, to insure that the address is a multiple of 8.

Sample Code

```
LOAD1   LE 0,FL1        LOAD FP REG 0 FROM ADDRESS FL1
LOAD2   LD 2,FL2        LOAD DOUBLE PRECISION
LOAD3   LER 4,0         COPY SINGLE PRECISION INTO FP REG 4
LOAD4   LDR 6,2         COPY DOUBLE PRECISION INTO FP REG 6
STORE1  STE 6,FL3       STORE THE SINGLE PRECISION INTO FL3
STORE2  STD 6,FL4       STORE DOUBLE PRECISION INTO FL4

FL1     DC  E'123.45'   A SINGLE PRECISION FLOATING POINT
                        CONSTANT.  ADDRESS IS A MULTIPLE OF 4.
FL2     DC  D'45678.90' A DOUBLE PRECISION FLOATING POINT
                        CONSTANT.  ADDRESS IS A MULTIPLE OF 8.
FL3     DS  E           JUST RESERVE AN ALIGNED FULLWORD
FL4     DS  D           RESERVE AN ALIGNED DOUBLE WORD.
```

Note that the contents of register 6 are first stored as a single precision result, then as a double precision result.

Addition and Subtraction

There are four distinct addition instructions and four distinct subtraction instructions for normalized floating–point numbers. These instructions are as follows:

Mnemonic	Operation	Opcode	Operand Format
AE	Add single precision	7A	R1,D2(X2,B2)
AD	Add double precision	6A	R1,D2(X2,B2)
AER	Add register single precision	3A	R1,R2
ADR	Add register double precision	2A	R1,R2
SE	Subtract single precision	7B	R1,D2(X2,B2)
SD	Subtract double precision	6B	R1,D2(X2,B2)
SER	Subtract register single precision	3B	R1,R2
SDR	Subtract register double precision	2B	R1,R2

Subtraction functions by changing the sign of the second operand and then performing an addition. The first step in each is ensuring that the characteristics of both operands are equal. If unequal, the field with the smaller characteristic is adjusted by shifting the fraction to the right and incrementing the characteristic by 1 until the characteristics are equal.

Each of these operations sets the proper condition code for conditional branching.

Recall that the standard floating point format is as follows:

Leftmost 8 bits		Other bits
Sign bit	7–bit characteristic	The fraction

Here is an example of adjusting the characteristic.

```
Characteristic   Fraction
    41            29000   =    41   29000
    40            12000   =    41   01200
                               ----------
                               41   2A200
```

Suppose that the fraction overflows. If that happens, the fraction is shifted right by one hexadecimal digit and the characteristic is incremented by 1. This last operation is called normalization, in that it returns the result to the expected normal form.

```
Characteristic   Fraction
    41           940000
    41           760000
    41          10A0000    which becomes  42 010A000.
```

Normalized addition and subtraction perform post–normalization; that is, they normalize the result after the operation. This is seen in the example above. Precision is maintained by use of a guard digit, which saves the last digit shifted during normalization prior to addition or subtraction. This digit may be restored during post–normalization. Here is an example.

```
Characteristic   Fraction
    42           0B2584   =    42   0B2584
    40           114256   =    42   001142(5)   5 is the guard digit.
                               --------------
                               42   0B36C6(5)
Normalize the result           41   B36C65.
```

Here are some examples of the operation, which appear exactly as expected.

```
*        SINGLE PRECISION FLOATING POINT ADDITION
*
ADDSNG   LE  2,FL1     LOAD THE REGISTER
         AE  2,FL2     ADD SINGLE PRECISION
         LE  4,FL3     LOAD ANOTHER REGISTER
         AER 4,2       REGISTER 4 IS CHANGED
*
*        SINGLE PRECISION FLOATING POINT SUBTRACTION
*
SUBSNG   LE  2,FL1     LOAD THE REGISTER
         SE  2,FL2     SUBTRACT FROM THE VALUE IN REG 2
         LE  4,FL3     LOAD ANOTHER REGISTER
         SER 4,2       SUBTRACT REG 2 FROM REG 4,
                       CHANGING THE VALUE IN REGISTER 4
*
*        DOUBLE PRECISION FLOATING POINT ADDITION
*
ADDDBL   LD  2,FL1     LOAD THE REGISTER
         AD  2,FL2     ADD DOUBLE PRECISION
         LD  4,FL3     LOAD ANOTHER REGISTER
         ADR 4,2       REGISTER 4 IS CHANGED
*
*        DOUBLE PRECISION FLOATING POINT SUBTRACTION
*
SUBBDL   LD  2,FL1     LOAD THE REGISTER
         SD  2,FL2     SUBTRACT FROM THE VALUE
         LD  4,FL3     LOAD ANOTHER REGISTER
         SDR 4,2       REGISTER 4 IS CHANGED
*
*        A FEW MIXED OPERATIONS
*
MIXED    LD  2,FL1     DOUBLE PRECISION LOAD
         AE  2,FL2     SINGLE PRECISION ADDITION
         SE  2,FL3     SINGLE PRECISION SUBTRACTION
         STD 2,FL1     STORE AS DOUBLE PRECISION, THOUGH THE
                       RESULT IS NOT QUITE THAT PRECISE
*
FL1      DC D ‘123.4’
FL2      DC D ‘10.0’
FL3      DC D ‘150000.0’
```

Consider the “mixed precision” operations in which both single precision and double precision floating point numbers are used. The result is certainly less precise than a true double precision result, though possibly a bit more precise than a single precision one. This last claim of precision greater than single would be difficult to justify.

Multiplication
There are four distinct floating–point multiplication operations.

Mnemonic	Action	Opcode	Operands
ME	Multiply single precision	7C	R1,D2(X2,B2)
MD	Multiply double precision	6C	R1,D2(X2,B2)
MER	Multiply single (register)	3C	R1,R2
MDR	Multiply double (register)	2C	R1,R2

In each, the first operand specifies a register that stores the multiplicand and, after the multiplication, stores the product.

The operation normalizes the product, which in all cases is a double–precision result.

Division
There are four distinct floating–point division operations.

Mnemonic	Action	Opcode	Operands
DE	Divide single precision	7D	R1,D2(X2,B2)
DD	Divide double precision	6D	R1,D2(X2,B2)
DER	Divide single (register)	3D	R1,R2
DDR	Divide double (register)	2D	R1,R2

In each, the first operand specifies a register that stores the dividend and, after the division, stores the quotient. There is no remainder.

The operation normalizes the quotient, which in all cases is a double–precision result.

A divisor containing a zero fraction causes a program interrupt. Recall that the two parts of any number in IBM floating point format are the characteristic and the fraction. The characteristic is held in the leftmost byte of the format and represents the sign and exponent. The fraction is held in the rest of the format: for single precision this is the 24 rightmost bits, and for double precision this is the 56 rightmost bits.

Comparison
There are four distinct floating–point division comparison operations.

Mnemonic	Action	Opcode	Operands
CE	Compare single precision	79	R1,D2(X2,B2)
CD	Compare double precision	69	R1,D2(X2,B2)
CER	Compare single (register)	39	R1,R2
CDR	Compare double (register)	29	R1,R2

In each, the comparison sets the condition codes as would be expected for comparisons in the other formats. Each operation proceeds as a modified subtraction. The characteristic fields of the two operands are checked, and the smaller exponent is incremented while right shifting its fraction (denormalizing the number, but preserving its value) before the comparison.

If both operands have fractions that are zero (all bits in the field are 0), the result is declared to be equal without consideration of either the exponent or the sign.

The single precision operations compare only the leftmost 32 bits in each value.

Conversions To and From Floating Point

We now briefly discuss conversion of data between the three main forms that we have discussed at this time: two's–complement integer, packed decimal, and floating point. We begin by considerations related to conversion to and from integer data.

Recall that all floating point formats store data in the form $16^E \bullet F$, where F is a fraction not less than 1/16 = 0.625 and less than 1.0. Of particular interest here are the values $16^1 = 16$, and $16^8 = (2^4)^8 = 2^{32} = 4{,}294{,}967{,}296$.

Recall that a 32–bit two's–complement format can represent integers in a specific range with the most negative value being –2,147,483,648 and the most positive being 2,147,483,647. In terms of powers of 2, this range is from $-(2^{31})$ to $(2^{31} - 1)$. We see that $2^{31} = 16^8 \bullet 1/2$.

Consider non–zero integers. The range consistent with the fullword format is as follows.
The smallest magnitude is $1 = 16^1 \bullet (1/16)$; the exponent is 1, stored as 65 = **`X'41'`**.
The largest magnitude is $2^{31} = 16^8 \bullet 1/2$; the exponent is 8, stored as 72 = **`X'48'`**.

Immediately, we see that positive floating point numbers with characteristic fields at least **`X'41'`** but not greater than **`X'48'`** can be converted to integer form. If the characteristic field is less than **`X'41'`**, the floating–point value will be come an integer 0. If the characteristic field is greater than **`X'48'`**, the value is too large to be represented.

The primary difficulty with conversions between packed decimal format and floating–point format is the fact that the position of the decimal point in packed format is not specified. This difficulty presents itself in different forms, depending on the exact operation.

In translation from packed decimal to floating–point, the position of the decimal point must be explicitly specified in order to correctly set the exponent.

In translation from floating–point to packed decimal, the computations can be done almost exactly, but there is no place to store a value representing the position of the decimal point.

We shall explore these conversions again after we have developed a few more tools that can be used in scanning an array of digits or an array of bytes.

Input and Output of Floating Point Values

At this point in the chapter, the reader should be aware of a glaring omission. There has been no discussion of conversion from EBCDIC format to any of the floating–point formats or conversions from those formats back to EBCDIC. In other words, there has been no mention of methods to write assembly language programs either to read floating point data or to print those data in some readable format.

The plain fact is that floating–point I/O is not mentioned in any standard source, either the textbooks [R_02, R_05, R_07, or R_18] or in any of the IBM manuals for the S/370 and successor systems [R_15, R_16, R_17, R_19, R_20, R_21, R_22, or R_23]. A request to IBM for any information yielded only a copy of the z/Series Principles of Operation [R_16].

It is likely the case that nobody writes complete floating–point oriented programs in IBM Assembler Language and has not for quite a few years. If your author finds any additional information, it will be shared in the next revision of this textbook.