Chapter 17: Conversions for Floating–Point Formats

This chapter discusses conversion to and from the IBM floating point format. Specifically, the chapter covers the following four topics.

1. Conversion of data in 32–bit fullword to a an equivalent value in single–precision floating–point format.
2. Conversion of data in single–precision floating–point to a an equivalent value in 32–bit fullword format.
3. Conversion of data in the Packed Decimal format to an equivalent value in double–precision floating–point format. We shall discuss the problem of locating the decimal point, which is implicit in the Packed Decimal format.
4. Conversion of data in double–precision floating–point to an equivalent value in Packed Decimal format. This discussion may be a bit general, as the detailed assembler code is somewhat difficult to design.

The true purpose of this chapter is to focus the reader's attention on one of the many services provided by the RTS (Run–Time System) of a modern compiled high–level language. This is more of the text's focus on assembler language as a tool to understanding the workings of a modern computer as opposed to being a language in which the reader is likely to program.

The IBM Mainframe Floating–Point Formats

The first thing to do in this presentation is to give a short review of the IBM Mainframe format for floating–point numbers. We might note that the modern Series Z machines, such as the z/10 running z/OS, support three floating–point formats: binary floating–point (the IEEE standard), decimal floating–point, and hexadecimal floating–point. The older S/370 series supported only what is called "hexadecimal" format. This format is so named because the exponent is stored as a power of 16. This chapter will use only two of the standard floating–point formats for the S/370: single–precision (E) and double–precision (D).

Each floating point number in this standard is specified by three fields: the sign bit, the exponent, and the fraction. 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

In IBM terminology, the field used to store the representation of the exponent is called the **"characteristic field"**. 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. This field is prefixed by a sign bit, which is 1 for negative and 0 for non–negative. These two fields together will be represented by two hexadecimal digits in a one–byte field.

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 $-64 \le E \le 63$. 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.

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 $X \bullet 16^E$, after which we determine the sign and build the first two hex digits.

Example 1: Positive exponent and positive fraction.

The decimal number is 128.50. The format demands a representation in the form $X \bullet 16^E$, with $0.625 \le X < 1.0$. As $128 \le X < 256$, 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 2: 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 3: 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 $0.625 \le X < 1.0$.

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 4: 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 \bullet$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: 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`**.

The format of this “true zero” will be important when we consider conversions to and from the fullword format used for 32–bit integers. In particular, note that the bit field interpreted as a single–precision true zero will be interpreted as a 32–bit integer zero.

The structure of the formats facilitates conversion among them. For example, consider the positive decimal number 80.0, which in hexadecimal is **X'50'**. Conversion of this to floating–point format involves noting that 80 = 64 + 16 = 256•(0/2 + 1/4 + 0/8 + 1/16). Thus the exponent of 16 is 2 and the characteristic field stores **X'42'**. The fraction field for this number is **0101**, which is hexadecimal 5. The representation of the number in the two standard IBM floating–point formats chosen for this chapter's discussion is as follows.

Single precision (E) format **42 50 00 00**

Double precision (D) format **42 50 00 00 00 00 00 00**

Conversion from single precision to double precision format is quite easy. Just add 8 hexadecimal zeroes. Conversion from double precision to single precision is either easy or a bit trickier, depending on whether one truncates or attempts to round.

Convert the double precision value **42 50 00 00 11 10 00 00**

Simple truncation will yield **42 50 00 00**

A reasonable rounding will yield **42 50 00 01**

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.

To illustrate this idea consider the two data declarations.

```
EFLOAT    DS E     Declare a 32–bit single precision
DFLOAT    DS D     Declare a 64-bit double precision
```

Consider the following instructions that use floating–point register 0. Remember that this register holds 64 bits, which is enough for a double–precision (D) floating–point value.

```
LD   0,DFLOAT  Load the full 64-bit register from
               the double precision 64-bit value.

LE   0,EFLOAT  Load the leftmost 32 bits of the register
               from the single precision 32-bit value.
               The rightmost 32 bits of the register are
               not changed [R_15, page 43].

STD  0,DFLOAT  Store the 64 bits from the register into
               the 64-bit double precision target.

STE  0,EFLOAT  Store the leftmost 32 bits of the register
               into the 32-bit single precision target.
```

Another Look at Two's–Complement Integers

In order to develop the algorithms for converting between two's–complement integers and floating–point formats, we must examine the structure of positive integers from a slightly different viewpoint, one that is of little view in the "pure integer" world.

We shall focus on conversions for positive fullword integers. As we shall see, handling negative integers is a simple extension of the above. Handling halfword integers is even easier, as we shall use the LH (Load Halfword) instruction to load them into a register. All of our conversions from integer format to floating–point format will assume that the integer argument is found in a general–purpose register.

The fullword conversion code will begin with an instruction such as

```
        L  R9,FW     Load the fullword into register 9
```

The halfword conversion code will begin with an instruction such as

```
        LH R9,HW     Load the halfword into register 9,
                     extending the sign to make a fullword.
```

The handling of negative numbers is quite simple. We first declare a single–character (one byte) area called **THESIGN**, to hold a representation of the sign in a format that will assist the processing of the resulting floating point number.

For a negative number, **THESIGN** will be set to **X'80'**.
For a non–negative number, its value will be set to **X'00'**.

In the code, the location **THESIGN** would be declared as follows [R_17, page 41].

```
THESIGN   DS X1     One byte of storage
```

Here is a fragment of the code, assuming that the signed integer value is in R9. Note the use of the **MVI** instruction with the hexadecimal equivalent of a character [R_17, page 41].

```
          MVC  THESIGN,=X'00'       Initialize the sign field
          CH   R9,=H'0'             Look at the integer value
          BZ   DONE                 It is zero, nothing to do.
          BNL  NOTNEG               Is the value negative?

          MVC  THESIGN,=X'80'       Yes, it is negative.
          LCR  R9,R9                Get the absolute value

NOTNEG    Now process the positive number in R9.
```

For ease of illustration I shall discuss the structure of a signed 16–bit halfword. As seen above, we may assume that the halfword represents a positive integer.

Hex digit		0				1				2				3			
Power of 16	4				3				2				1				0
Power of 2	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}

In a signed halfword, the bits A_0 through A_{15} would represent the binary bits of the 16–bit integer. As we have specified that we have a positive integer, we know that $A_0 = 0$ and that at least one of the other bits is equal to 1.

The value of the halfword is $A_0 \bullet 2^{15} + A_1 \bullet 2^{14} + A_2 \bullet 2^{13} + A_3 \bullet 2^{12} + \ldots + A_{15} \bullet 2^0$.

Another way to write this would be as follows:

$2^{16}\bullet(A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{15}/2^{16})$, which can also be written as

$16^4\bullet(A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{15}/2^{16})$. This seems to be in a form that is ready for translation into the IBM floating–point representation. If one of A_1, A_2, or A_3 is nonzero, this will work. The exponent will be 4 and the fraction $A_0A_1A_2A_3\ldots A_{15}$.

Please note that the IBM Single–Precision Floating–Point format is a 32–bit format, so the above should not be taken literally. It is just an indicator of where we need to go.

The above method, as extended to 32 bits, might work if it were not for the issue of normalization. All IBM floating point standards require that the first two bytes (four hex digits) of the representation be of the following format.

Digit	0	1	2	3
Contents	Sign bit and 7–bit characteristic field holding the exponent.		High order bits of the fraction. At least one non-zero bit.	Next four bits of the fraction.

In other words, the value of hexadecimal digit 2 cannot be a zero. This is a requirement of the normalized representation, which calls for representing the floating–point value in the form $16^E \bullet F$, where $1/16 \le F < 1$.

In our 16–bit example, suppose that the four high order bits are all zero, but that at least one of the next four bits is not zero. What we have is of the following form.

Hex digit		0				1				2				3			
Power of 16	4				3				2				1				0
Power of 2	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
		0	0	0	0	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}

The value of the halfword is $A_0\bullet2^{11} + A_1\bullet2^{10} + A_2\bullet2^9 + A_3\bullet2^8 + \ldots + A_{11}\bullet2^0$.

Another way to write this would be as follows:

$2^{12}\bullet(A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{15}/2^{12})$, which can also be written as

$16^3\bullet(A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{15}/2^{12})$. This seems to be in a form that is ready for translation into the IBM floating–point representation. If one of A_1, A_2, or A_3 is nonzero, this will work. The exponent will be 3 and the fraction $A_0A_1A_2A_3\ldots A_{11}$.

Before continuing our discussion, let us reflect on a method to detect whether or not the four high–order bits in a register are all zero. For this, we need to turn to the logical AND, which was covered in the last pages of Chapter 12 of this textbook.

The type of instruction I choose to use is the type RX logical AND instruction, **N**.

```
         LR   R8,R9              COPY R9 INTO R8 SO THAT THE FOUR
*                                HIGH ORDER BITS CAN BE TESTED
*                                WITHOUT LOSING THE VALUE.
         N    R8,=X'F0000000'    MASK OUT THE 4 HIGH ORDER BITS
*                                THE MASK IS F0 00 00 00.
         BNZ HAVE1               FOUND A 1 BIT.
```

Note that the block above is that to be used for 32–bit integers. The logical AND will raise the zero condition flag only when all bits in R8 become 0 after the operation.

The Range of Exponents

The first thing to do is predict the largest exponent that can arise from converting a 32–bit fullword. The magnitude of such an integer cannot exceed 2^{31} = 2,147,483,648, which is the same as $2^{32} \bullet 1/2$. This value can be represented as $16^8 \bullet 1/2$, indicating that the largest exponent for a floating–point number converted from a fullword is 8.

The smallest positive integer is $1 = 16^1 \bullet 1/16$, indicating that the smallest exponent for a floating–point number converted from a fullword is 1. Recall that the characteristic field part of the floating–point representation contains the exponent stored in excess–64 format; the value stored is (Exponent + 64). The range of possible characteristics is from 72 down to 65.

In our example and in our code, we shall manipulate the characteristic directly.

A Fullword Example

The algorithm to be developed for fullwords will be inspired by the halfword example above. It will involve multiple shifts and logical operations to locate the most significant 1 bit and use the information obtained to generate the characteristic field and fraction. But first, let's do a computation "by hand". Our example is 32,685.

As a 16–bit integer, this can be represented as **`0111 1111 1010 1101`**. In 32 bits it would be **`0000 0000 0000 0000 0111 1111 1010 1101`**. To avoid this mess of ones and zeroes, this chapter will use hexadecimal notation; the value is **`0000 7FAD`**.

This algorithm functions by testing the leftmost hexadecimal digit in the value, which represents the four high–order bits in the representation. If the digit is zero, the value is shifted left by four bits, equivalent to shifting one hexadecimal digit. The **SLL** (Shift Left Logical) instruction will be used for this task, as it pads the right with zeroes.

1. Start Characteristic = 72, Value = **`0000 7FAD`**
   The most significant digit is 0, so shift left and reduce the characteristic by 1.
2. Characteristic = 71, Value = **`0007 FAD0`**
   The most significant digit is 0, so shift left and reduce the characteristic by 1.
3. Characteristic = 70, Value = **`007F AD00`**
   The most significant digit is 0, so shift left and reduce the characteristic by 1.
4. Characteristic = 69, Value = **`07FA D000`**
   The most significant digit is 0, so shift left and reduce the characteristic by 1.
5. Characteristic = 68, Value = **`7FAD 0000`**
   The most significant digit is not 0, so we have both our fraction and our characteristic with value 68 or **`X'44'`**. The fraction is X '7FAD00'.

The representation of this value in the 32–bit single–precision floating–point format is: **`44 7F AD 00`**. Just for fun, let's reverse engineer this value.

The characteristic field is **`X'44'`**, indicating an exponent of 4. The value represented is $16^4 \bullet (7/16 + 15/16^2 + 10/16^3 + 13/16^4) = 7 \bullet 16^3 + 15 \bullet 16^2 + 10 \bullet 16 + 13 =$
$7 \bullet 4096 + 15 \bullet 256 + 160 + 13 = 28{,}672 + 3{,}840 + 173 = 32{,}685$.

Why Convert to Single–Precision Floating–Point?
The topic of this section is the conversion of 32–bit fullword integer values into equivalent floating–point values. One might wonder why we have selected the single–precision format as the target, in preference to the double–precision floating–point (D) format.

One reason for the choice is simplicity; it is easier to discuss single–precision format in this context. Another reason is precision. The single–precision floating–point format has a precision of seven digits. A fullword has at most 10 digits, with the most significant digit being restricted to 0, 1, or 2; one might stay that the format has only nine digits. In other words, for most cases, conversion to the single–precision format does not lose accuracy.

Two Unusual Cases
There are two cases in which the above "shift left to find the most significant 1 bit" strategy does not work. In each of these cases, one finds that one of the bits in the leftmost byte of the 32–bit integer is not zero. Recall that the format of the single–precision floating–point format may be expressed as **C C | F F F F F F**, where the first byte (denoted **C C**) holds the sign bit and the characteristic field.

Consider the case illustrated below, in which we assume that not all of A_0, A_1, A_2, and A_3 are zero. Since the number is positive, we do know that $A_0 = 0$, but that is not significant here.

Hex Digit		0				1				2	3	4	5	6	7
Power 16	8				7				6						
Power 2	32	31	30	29	28	27	26	25	24						
		A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8 through A_{31}					

The value of the fullword is $A_1 \bullet 2^{30} + A_2 \bullet 2^{29} + A_3 \bullet 2^{28} + A_4 \bullet 2^{27} + \ldots + A_{31} \bullet 2^0$.

Another way to write this would be as follows:
$2^{32} \bullet (A_1/4 + A_2/8 + A_3/16 + \ldots + A_{31}/2^{32})$, which can also be written as
$16^8 \bullet (A_1/4 + A_2/8 + A_3/16 + \ldots + A_{31}/2^{32})$. The single–precision floating–point format calls for an 8–bit field holding the sign and characteristic, followed by a 24–bit fraction, which here would be A_0 through A_{23}. The characteristic field would hold **X'48'**, indicating a positive number with an exponent field of 8.

The conversion for this case is to start with the 32–bit (8 hexadecimal digit) value.

Digit	0	1	2	3	4	5	6	7
	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}	A_{24}-A_{27}	A_{28}-A_{31}

This is logically right shifted by 8 bits (2 hexadecimal digits) to get the value:

Digit	0	1	2	3	4	5	6	7
	0000	0000	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}

The two hexadecimal digits for the characteristic field are then inserted to get the final value.

Digit	0	1	2	3	4	5	6	7
	4	**8**	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}

The Other Case

Consider now the second case. Again we assume that not all of A_0, A_1, A_2, and A_3 are zero.

Hex Digit		0				1				2	3	4	5	6	7
Power 16	8				7				6						
Power 2	32	31	30	29	28	27	26	25	24						
		0	0	0	0	A_0	A_1	A_2	A_3	A_4 through A_{27}					

The value of the fullword is $A_0 \bullet 2^{27} + A_1 \bullet 2^{26} + A_2 \bullet 2^{25} + A_3 \bullet 2^{24} + A_4 \bullet 2^{23} + \ldots + A_{27} \bullet 2^0$.

Another way to write this would be as follows:
$2^{28} \bullet (A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{27}/2^{28})$, which can also be written as
$16^7 \bullet (A_0/2 + A_1/4 + A_2/8 + A_3/16 + \ldots + A_{27}/2^{28})$. The characteristic field is **X'47'**.

Remember that the single–precision format calls for a 24–bit fraction field.

The conversion for this case is to start with the 32–bit (8 hexadecimal digit) value.

Digit	0	1	2	3	4	5	6	7
	0000	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}	A_{24}-A_{27}

This is logically right shifted by 8 bits (2 hexadecimal digits) to get the value:

Digit	0	1	2	3	4	5	6	7
	0000	0000	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}

The two hexadecimal digits for the characteristic field are then inserted to get the final value.

Digit	0	1	2	3	4	5	6	7
	4	**7**	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}

The "Left Shifter" Cases

In all other cases, the positive integer to be converted has the following format.

Digit	0	1	2	3	4	5	6	7
	0000	0000	A_0-A_3	A_4-A_7	A_8-A_{11}	A_{12}-A_{15}	A_{16}-A_{19}	A_{20}-A_{23}

The two hexadecimal digits that will be occupied by the characteristic field are already clear, so we do not require any right shifting to move the most significant part of the fraction to its proper location. We are only assured that at least one of bits A_0 through A_{23} is not zero. The procedure to follow is the test and shift left procedure sketched above for the halfword case.

The operation to be used in left shifting register R9 will be the SLL (Logical Left Shift), which will insert 4 binary zeroes on the right part of R9 every time it is left shifted by 4 bits. This usage is consistent with building a fraction with trailing zeroes; 0.4 = 0.400000.

At the end of the code to be developed, we shall store the integer contents of a the register R9 into a word declared to be in single–precision floating–point format. Note that both the fullword (F) format and single–precision (E) floating–point formats are 32–bit (four byte) formats, so that one value can be stored into another.

What we are doing here is manipulating each part of the formatted number as if it were an integer, and then creating a bit pattern that will bear interpretation as a floating–point value.

Here is the code developed as a result of the discussions so far. The assumption is that the integer value to be converted is found in general–purpose register R9 and that the result is to be deposited in an area of memory declared as **EFLOAT DS E**.

```
          MVI  THESIGN,X'00'    Initialize the sign field
          CH   R9,=H'0'         Look at the integer value
          BZ   DONE             It is zero, nothing to do.
          BNL  NOTNEG           Is the value negative?

          MVI  THESIGN,X'80'    Yes, it is negative.
          LCR  R9,R9            Get the absolute value
NOTNEG    LR   R8,R9            Get a copy into R8
*         At this point, R9 will contain the value as it is
*         being transformed from 32–bit integer format into
*         Single-Precision Floating-Point format.
*         R8 is a work register used to test values.

          N    R8,=X'F0000000'  Is the high-order hex digit 0
          BZ   D0IS0            Yes, not this special case
          SRL  R9,8             Clear out the characteristic
          O    R9,=X'48000000'  Set the exponent
          B    SETSIGN          Set the sign

D0IS0     LR   R8,R9            Get the value back into R8
          N    R8,=X'FF000000'  Is the next digit 0
          BZ   D1IS0            Yes, not this special case
          SRL  R9,4             Clear out the characteristic
          O    R9,=X'47000000'  Set the exponent
          B    SETSIGN          Set the sign

*
*         Here we make sure that the two high-order digits in
*         R9 are zero, and test that we have a positive value.
*
D1IS0     N    R9,=X'00FFFFFF'  Sanity check: is any bit = 1
          BZ   SETSIGN          NO, the result is 0

LFTSHFT   EQU  *
*         Here we do the left shifting of the number to
*         generate the proper characteristic field and
*         normalized fraction.

SETSIGN   SR   R8,R8            Set R8 to zero
          IC   R8,THESIGN       Get the sign byte into R8
          CH   R8,=H'0'         Is it zero (non-negative)?
          BZ   DONE             Yes, value is not negative
          O    R9,=X'80000000'  Set the sign bit

DONE      ST   R9,EFLOAT        Store the result into the
*                               32-bit field EFLOAT
```

Left Shifting

Having disposed of the two cases that are unusual due to the bit structure of the single–precision floating–point format, let us consider the "more general" case. In this case, the first two hexadecimal digits of the integer value are 0 and one (or more) of the other six is non–zero. From a magnitude consideration, this covers numbers with integer values at least 1 and less than 2^{24} = 16,777,216. Many commonly used integer values easily fit within this range.

The preconditions for this section of the code are simple. Register R9 contains the absolute value of the integer, with the sign having been tested and recorded for use later. As noted above, this magnitude is not greater than 16,777,215, which in hexadecimal is **X'FFFFFF'**.

```
LFTSHFT
ISD3A0   LR   R8,R9              Copy value into R8
         N    R8,=X'00F00000'    Is the third digit nonzero?
         BNZ  SETVAL             Yes, it is non-zero.
         SHL  R9,4               No, it is not.  Shift left by 4
*                                bits to examine another digit
         B    ISD3A0             Try again
SETVAL   EQU *
```

There is quite a bit missing from the above loop. What we need to do is begin with the characteristic field as **X'46'**, or decimal 70, representing an exponent of +6. As each test reveals digit 3 to be zero, we need to count down the exponent and shift left. The lowest admissible exponent is 1, represented in the characteristic field as 65 or **X'41'**.

The construct appropriate for this is **BXH**, which will branch on a value higher than **X'41'**. Recall the format of the source code for this instruction.

```
BXH  Register,Register_Pair,Target_Address
```

Where **Register** denotes a register containing a count; here the characteristic field.

Register_Pair contains the even register of an even–odd pair.
The even register contains a value to be used in incrementing the count.
The odd register contains a value to be used as a limit.

Target_Address contains the branch target.

The design here is to start the characteristic field at **X'46'**, representing an exponent of +6. For each time the digit is found to be zero, we shift left by 4 bits (one hexadecimal digit), and decrement the exponent. This continues until the characteristic field is **X'41'**, indicating the smallest characteristic field for a positive non–zero integer.

With R8 and R9 in use, this design calls for the following.
R6 and R7 are selected as the even–odd register pair.
R5 will be used to hold the value of the characteristic field.

Here is an example of the shift strategy. Let R9 contain **X'0000 2BAD'**. (R5) = **X'46'**.
Is digit 3 a 0? Yes. Shift left and decrement (R5). **X'0002 BAD0'**. (R5) = **X'45'**.
Is digit 3 a 0? Yes. Shift left and decrement (R5). **X'002B AD00'**. (R5) = **X'44'**.
Is digit 3 a 0? No, it is not. Keep (R9) = **X'002B AD00'** and (R5) = **X'44'**.

The answer is that the floating–point representation is **X'442B AD00'**.

Here is the code.

```
          MVI  THESIGN,X'00'    Initialize the sign field
          CH   R9,=H'0'         Look at the integer value
          BZ   DONE             It is zero, nothing to do.
          BNL  NOTNEG           Is the value negative?

          MVI  THESIGN,X'80'    Yes, it is negative.
          LCR  R9,R9            Get the absolute value
NOTNEG    LR   R8,R9            Get a copy into R8
*         At this point, R9 will contain the value as it is
*         being transformed from 32–bit integer format into
*         Single-Precision Floating-Point format.
          N    R8,=X'F0000000'  Is the high-order hex digit 0
          BZ   D0IS0            Yes, not this special case
          SRL  R9,8             Clear out the characteristic
          O    R9,=X'48000000'  Set the exponent
          B    SETSIGN          Set the sign

D0IS0     LR   R8,R9            Get the value back into R8
          N    R8,=X'FF000000'  Is the next digit 0
          BZ   D1IS0            Yes, not this special case
          SRL  R9,4             Clear out the characteristic
          O    R9,=X'47000000'  Set the exponent
          B    SETSIGN          Set the sign

D1IS0     N    R9,=X'00FFFFFF'  Sanity check: is any bit = 1
          BZ   SETSIGN          NO, the result is 0

LFTSHFT   LH   R5,=H'70'        Start value for characteristic
          LH   R6,=H'-1'        Increment value.
          LH   R7,=H'65'        Limit value

ISD3A0    LR   R8,R9            Copy value into R8
          N    R8,=X'00F00000'  Is the third digit nonzero?
          BNZ  SETVAL           Yes, it is non-zero.
          SHL  R9,4             No, it is not.  Shift left by 4
          BXH  R5,R6,ISD3A0     Try again

SETVAL    SHL  R5,24            Move characteristic into place
          OR   R9,R5            Create the number

SETSIGN   SR   R8,R8            Set R8 to zero
          IC   R8,THESIGN       Get the sign byte into R8
          CH   R8,=H'0'         Is it zero (non-negative)?
          BZ   DONE             Yes, value is not negative
          O    R9,=X'80000000'  Set the sign bit

DONE      ST   R9,EFLOAT        Store the result into the
*                               32-bit field EFLOAT
```

Conversion of Single–Precision Floating–Point to Integer

Here again, the first step is to detect and note the sign of the number. We shall focus on converting positive values, with the sign added at the end of the conversion process. In this case the process of "adding the sign" will be that of taking the two's complement.

The process will begin with the floating–point value stored in location EFLOAT, declared as:

```
EFLOAT    DS  E      A 32–bit (4 byte) storage allocation.
```

Note however that, throughout this conversion, **EFLOAT** will be treated as if it were a 32–bit fullword integer. Again, we are processing this bit by bit, and hexadecimal digit by hex digit. We are not really interested in its value when considered as a floating point number.

The first step is to access this four–byte location using an integer instruction.

```
          L  R9,EFLOAT
```

Now, we use a characteristic common to both integer and floating–point arithmetic. If bit 0 (the leftmost bit) of the 32–bit representation is 1, the number is negative. Otherwise, the number is non–negative, and might be zero. We immediately test for this.

Recall two things when examining the code below.

1. The value **X'00000000'**, viewed as a single–precision floating–point value is what IBM calls a "true zero". It converts to integer zero.
2. The value **X'7FFFFFFF'** represents a 32–bit number in which all bits, save bit 0 (the sign bit) are 1. We use this to mask out the sign bit and keep in R9 a value that would be interpreted as the absolute value of the floating–point number.

```
          MVI  THESIGN,X'00'    Initialize the sign field
          L  R9,EFLOAT          Load the floating-point value
          CH   R9,=H'0'         and examine the sign bit.
          BZ   DONE             The value is zero, nothing to do.
          BNL  NOTNEG           Is the value negative?
          MVI  THESIGN,X'80'    Yes, it is negative.
          N    R9,=X'7FFFFFFF'  Zero out the sign bit.
```

The next section of code reflects the fact that, if the fraction part of the representation is zero, then the value represented is 0 without regard to the characteristic field.

```
NOTNEG    LR   R8,R9            Copy the value into R8
          N    R8,=X'00FFFFFF'  Examine the fraction.  Is it 0?
          BNZ  FRNZ             No, keep on working
          SR   R9,R9            Yes, the value is zero.  So set
          B    DONE             the result as 0 and exit.
FRNZ      EQU  *                Keep on processing.
```

We now check the range of the characteristic field to determine if the exponent is consistent with conversion to an fullword integer. If the characteristic is less than 65 (**X'41'**), the value is less than 1 and will be converted to a 0. If the characteristic is greater than 72 (**X'48'**), the magnitude is too large to be represented as a fullword. What the code should do in this situation is up to the designer; here we set the integer to the maximum value. This is probably a poor design choice, but for now it is as good as any. Here is this code.

```
         LR   R8,R9            Copy the value into R8
         N    R8,=X'FF000000'  Isolate the characteristic field
         SRL  R8,24            Shift to least significant byte
         CH   R8,=H'64'        Is exponent big enough?
         BH   OVER1            Yes, number is not < 1.
         SR   R9,R9            No, set result to zero
         B    DONE             and be done with it.
OVER1    CH   R8,=H'72'        Is the exponent too big?
         BNH  CANDO            NO, it is fine.
         L    R9,=X'7FFFFFFF'  Biggest positive number
         B    SETVAL           Go adjust the sign.
CANDO    EQU *                 Value can be converted.
```

Here is the code as it exists at this point.

```
         MVI  THESIGN,X'00'    Initialize the sign field
         L    R9,EFLOAT        Load the floating-point value
         CH   R9,=H'0'         and examine the sign bit.
         BZ   DONE             The value is zero, nothing to do.
         BNL  NOTNEG           Is the value negative?
         MVI  THESIGN,X'80'    Yes, it is negative.
         N    R9,=X'7FFFFFFF'  Zero out the sign bit.

NOTNEG   LR   R8,R9            Copy the value into R8
         N    R8,=X'00FFFFFF'  Examine the fraction.  Is it 0?
         BNZ  FRNZ             No, keep on working
         SR   R9,R9            Yes, the value is zero.  So set
         B    DONE             the result as 0 and exit.

FRNZ     LR   R8,R9            Copy the value into R8
         N    R8,=X'FF000000'  Isolate the characteristic field
         SRL  R8,24            Shift to least significant byte
         CH   R8,=H'64'        Is exponent big enough?
         BH   OVER1            Yes, number is not < 1.
         SR   R9,R9            No, set result to zero
         B    DONE             and be done with it.

OVER1    CH   R8,=H'72'        Is the exponent too big?
         BNH  CANDO            NO, it is fine.
         L    R9,=X'7FFFFFFF'  This is the biggest positive number
         B    SETVAL           Go adjust the sign.
CANDO    EQU *                 Value can be converted.

*        Here is the code for processing the values that
*        can be converted into a fullword 32-bit integer.

SETVAL   SR   R8,R8            Set R8 to 0.
         IC   R8,THESIGN       Load the sign value
         CH   R8,=H'0'         Is the sign bit set?
         BZ   ISPOS            No, we are OK
         LCR  R9,R9            Negate the absolute value
DONE     EQU  *                We are done here.
```

Now we discuss the code for converting those floating point values that can be converted into positive fullword values. This code will use **SLDL** (Shift Left Double Logical).

We begin with another test to account for a case that cannot be converted. When the characteristic field is **X'48'** (decimal 72, representing an exponent of 8) and the most significant bit in the fraction is a 1, the absolute value will be not less than 2^{31}, which cannot be represented as a fullword integer. In other words, we are looking for this value.

Digit	0	1	2	3 – 7
	4	**8**	1 A_1 A_2 A_3	A_4-A_{23}

We use the **SLDL** instruction on the register pair (8, 9). At first, we shall clear R8 and shift the two high–order hexadecimal digits of R9 into it (**SLDL** by 8 bits). If this value is not **X'48'**, we proceed with the conversion. Otherwise one more **SLDL** will tell the tale.

```
CANDO    SR   R8,R8           Set R8 to zero
         SLDL R8,8            Shift two high-order digits into R8
         CH   R8,=H'72'       Is the exponent an 8?
         BL   DOIT            Yes, we can continue
*
*        At this point, the most significant fraction bit occupies
*        the sign bit in R9.  Check to see if R9 is negative.
*
         CH   R9,=H'0'        Is the sign bit set?
         BP   DOIT            No, the high-order fraction bit is 0
         L    R9,=X'7FFFFFFF'  Set to the biggest positive integer
         B    SETVAL           Go adjust the sign.
```

At this point, the register values are as follows:

1. R8 contains the characteristic value, equal to (Exponent + 64).
2. R9 contains the fraction in which the most significant hex digit is not 0.
   This is a result of the fact that the number was stored in a normalized format.
3. The low order eight bits (two hexadecimal digits) of R9 are all zero.
   This is due to the execution of the logical left shift **SLDL**.

The final processing of R9 to produce an integer that contains the absolute value of the desired result is to shift it right by a count related to the exponent. This will shift some 1 bits off the right, thus truncating the value. The requirements are as follows.

Characteristic	Exponent	Shift right by
72	8	0 bits
71	7	4 bits (1 hexadecimal digit)
70	6	8 bits (2 hexadecimal digits)
69	5	12 bits (3 hexadecimal digits)
68	4	16 bits (4 hexadecimal digits)
67	3	20 bits (5 hexadecimal digits)
66	2	24 bits (6 hexadecimal digits)
65	1	28 bits (7 hexadecimal digits)

The formula for the shift count is seen to be (72 – Characteristic)•4.

Here is the code to do that computation and produce the absolute value of the integer. Recall that the source code format of the logical right shift can be in the following form. **SRL R1,D2(B2)** in which the shift amount is determined by adding the value in **D2** to the contents of the register indicated by **B2**. This is base/displacement form used to compute a number and not an address.

```
DOIT     SH   R8,=H'72'     Produce (Characteristic – 72)
         LCR  R8,R8         Produce (72 – Characteristic)
         SLL  R8,2          Multiply by 4
         SRL  R9,0(R8)      Shift R9 by the amount in R8
```

Let's try a few examples to see if this works.

1. A large positive number. In hexadecimal, it is represented as **46 7F 03 00**.

Note immediately that the characteristic field is **X'46'**, so that the exponent is 6. We first compute the value directly; it is $16^6 \bullet (7/16 + 15/256 + 0/16^3 + 3/16^4) =$
$7 \bullet 16^5 + 15 \bullet 16^4 + 3 \bullet 16^2 =$
$7 \bullet 1048576 + 15 \bullet 65536 + 3 \bullet 256 =$
$7340032 + 983040 + 768 = 8{,}323{,}840.$

We now apply our algorithm. At the time that the characteristic is tested, R8 contains decimal 70 and R9 contains **7F 03 00 00**. The algorithm calls for a logical right shift of R9 by 8 bits or 2 hexadecimal digits. The new value is **X'007F0300'**, which represents the value $0 \bullet 16^7 + 0 \bullet 16^6 + 7 \bullet 16^5 + 15 \bullet 16^4 + 0 \bullet 16^3 + 3 \bullet 16^2 + 0 \bullet 16 + 0$, the value above.

2. A much smaller number related to the above. It is represented as **42 7F 03 00.** I have elected to keep the same fraction to simplify my work in doing the calculations. We first compute the value directly; it is $16^2 \bullet (7/16 + 15/256 + 0/16^3 + 3/16^4) =$
$7 \bullet 16 + 15 + 3/256 =$
$112 + 15 + 0.01171875 = 127.01171875.$

We now apply our algorithm. At the time that the characteristic is tested, R8 contains decimal 66 and R9 contains **7F 03 00 00**. The algorithm calls for a logical right shift of R9 by 24 bits or 6 hexadecimal digits. The new value is **X'0000007F'**, which represents the value $0 \bullet 16^7 + 0 \bullet 16^6 + 0 \bullet 16^5 + 0 \bullet 16^4 + 0 \bullet 16^3 + 0 \bullet 16^2 + 7 \bullet 16 + 15$, which is 127. Note that this integer conversion has simply dropped the fractional part. Writing code to round off is not very tricky; your author just elects not to do it.

Here is a sketch of one approach to the round–off question. This will be based on the use of the shift right logical double instruction **SRDL**. The value to be converted must be placed in the even register of an even–odd register pair and the odd register cleared. To illustrate, suppose that it is R8 that contains the value, represented as **43 7F 03 00**.

1. Clear R9. The register pair contains **43 7F 03 00 | 00 00 00 00**.
2. Shift right double by 5 digits to get **00 00 04 37 | F0 03 00 00**
3. The bit pattern in R8 represents the integer $4 \bullet 256 + 3 \bullet 16 + 7 = 1079$
   The bit pattern in R9 represents a fraction $(15/16 + 3/16^4)$, bigger than 0.50.
   This is noted by detecting the sign bit in R9.
4. Round off the value to 1080.

Packed Decimal Format to Floating–Point

Here we face a problem with no precise solution unless we give some additional input. Recall that a number in the packed decimal format is represented by a sequence of hexadecimal digits (all having decimal values in the range 0 through 9) followed by a single hexadecimal digit in the range **X'A'** through **X'F'**. Such a number may have from 1 through 31 decimal digits, with the trailing sign digit being necessary.

The routine for this conversion will be based on selecting individual digits from the packed format, one at a time. For this purpose, the digits will be numbered left to right in a manner reminiscent of the bit numbering used in a register. Consider the following.

Digit number	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**
Value	**3**	**1**	**4**	**1**	**5**	**9**	**2**	**6**	**5**	**3**	**6**

The reason that this has no precise solution is that the decimal place is not explicitly specified in the packed format representation of the value. Since we recognize this value as an approximate representation of the value π, we know the number is 3.1415926536. There are ten digits to the right of the decimal point and just one before it. In our representation we shall specify the value as a pair: (packed value, digit position).

The value here would be (**31415926536C**, **10**), indicating that we produce the floating point value corresponding to the integer value **31415926536** and then divide that by 10^{10}. At this point, we do not care that the integer just given cannot be represented as either a halfword or fullword by the S/370; this is just a conceptual calculation. All of our arithmetic here will be done in double–precision floating–point format.

Design of the Algorithm: Preconditions and Subprograms Used

The algorithm and its implementation in assembly language are designed assuming that:

1. The packed decimal value is found at location **PACKNUM**, and is validly formatted. In particular, the number has no more than 31 decimal digits and has a sign digit.
2. The sign digit is either **X'B'** or **X'D'** for a negative number or one of **X'A'**, **X'C'**, **X'E'**, or **X'F'** for a non–negative number.
3. The digits in the number will be indexed by a value from 0 possibly through 31.
4. One subroutine and one array will be used to help the computation. The subroutine accepts the digit index in R8 and returns the hexadecimal value of the digit in R9. The array is an array of double–precision floating–point values in which the Kth entry, at offset K•4, contains the equivalent of float(K).
5. The digits are scanned left to right until a sign digit is found. The numeric digits are processed very much as was done for the EBCDIC to integer direct conversion, except that floating–point arithmetic is used.
6. The result will be found in floating–point register 0.
7. As a precaution, the loop will be controlled by a **BXLE** instruction, to guarantee that no more than 32 hexadecimal digits are processed.
8. Each addressable byte contains two hexadecimal digits, each of which must be retrieved individually to compute the value of the Packed Decimal number.

Here is the subprogram used to return the hexadecimal value of a digit. The basic processing is first to convert the digit offset into a byte offset and then to move the digit into position.

The digit position is converted to a byte offset by division by 2. Consider the example:

```
PI       DC  P '31415926536'
```

What is stored is shown in the following table.

Address	PI	PI+1	PI+2	PI+3	PI+4	PI+5
Value	31	41	59	26	53	6C

The goal of this routine is to get the hexadecimal value of the digit into the low–order four bits of the general–purpose register 9 and have all other hexadecimal digits equal to 0.

Consider the processing of the digit at offset **2**. This is the digit **4**.

1. The digit index in R8 is forced to be in the range [0,31].
2. The index is copied into R7 and converted to a byte offset. This offset is 1.
3. The byte with value **X'41'** is loaded into R9.
4. The digit index is tested for being odd. It is not, so there is a logical right shift to place the first digit into the least significant place. R9 now has **X'4'**.
5. The seven high–order hex digits in R9 are masked out, returning the value **X'4'**.

Consider the processing of the digit at offset **3**. This is the digit **1**.

1. The digit index in R8 is forced to be in the range [0,31].
2. The index is copied into R7 and converted to a byte offset. This offset is 1.
3. The byte with value **X'41'** is loaded into R9.
4. The digit index is tested for being odd. It is odd, so there is a logical right so no right shifting is required. R9 now has **X'41'**.
5. The seven high–order hex digits in R9 are masked out, returning the value **X'1'**.

Here is the complete subroutine to get the digit. It is called "**GETDIGIT**".

```
*     Index of digit is in register R8. Value of R8 is preserved
*     The value of the digit is returned in R9.
*     R7 is used but not saved.  R4 contains the return address.
*
GETDIGIT  N   R8,=X'0000001F'  X'1F'=0001 1111; GET 5 LOW ORDER BITS
          LR  R7,R8            COPY VALUE OF DIGIT INDEX INTO R7
          SRL R7,1             CONVERT INTO BYTE OFFSET
          SR  R9,R9            SET R9 TO ZERO
          IC  R9,PACKNUM(R7)   GET THE BYTE INTO R7
*
          LR  R7,R8            GET THE DIGIT INDEX BACK INTO R7
          N   R7,=X'00000001'  MASK OUT THE UNIT BIT IN THE INDEX
          BNZ  ISODD           IF UNIT BIT IS NOT 0, INDEX IS ODD
          SRL R9,4             SHIFT TO GET DIGIT INTO POSITION
ISODD     N   R9,=X'0000000F'  ISOLATE THAT DIGIT
*
*     R9 NOW CONTAINS THE NUMERIC VALUE OF THE DIGIT.
*     IF VALUE > 9, THE DIGIT IS THE SIGN DIGIT.
          BR  R4               R4 contains the return address.
```

The array in question is just a table holding the double–precision floating–point values equivalent to the first ten non–negative integers (0 through 9), as well as the value 10.0 (which will be used for multiplication in forming the number). This is just the easiest way to convert a single digit positive integer into its equivalent floating–point value.

```
*     CONVERT SINGLE DIGIT INTEGER TO FLOATING POINT.
*
*     USE: PLACE VALUE INTO INDEX REGISTER, THEN SLL 2
*          LD 2,FPVALS(Index Register)

FPVALS   DC  D'0.0'
FPV1     DC  D'1.0'
FPV2     DC  D'2.0'
FPV3     DC  D'3.0'
FPV4     DC  D'4.0'
FPV5     DC  D'5.0'
FPV6     DC  D'6.0'
FPV7     DC  D'7.0'
FPV8     DC  D'8.0'
FPV9     DC  D'9.0'
FPV10    DC  D'10.0'
```

Here is the code for the conversion routine.

```
PACKTOFP LD   0,FPVALS      GET THE 0 ENTRY.  CLEAR FP REG 0

*        NOW SET UP FOR THE BXLE INSTRUCTION
*
         SR   R8,R8         SET COUNT TO 0.  THIS IS THE DIGIT INDEX
         LH   R10,=H'1'     LOOP INCREMENT IS 1, MOVE LEFT TO RIGHT
         LH   R11,=H'31'    LIMIT ON THE DIGIT INDEX

CONVERT  BAL  R4,GETDIGIT     GET THE DIGIT AT INDEX = R8
         CH   R9,=H'10'       DIGIT VALUE RETURNED IN R9
         BNL  DONE            WE HAVE THE SIGN DIGIT
         MD   0,FPV10         MULTIPLY CURRENT VALUE BY 10.0
         SLL  R9,2            MULTIPLY DIGIT VALUE BY 4
         LD   2,FPVALS(R9)    GET FP VALUE OF THE DIGIT
         ADR  0,2             ADD TO ACCUMULATING RESULT
         BXLE R8,R10,CONVERT  GO GET ANOTHER DIGIT

DONE     LH   R8,DECPLACE     GET THE DECIMAL PLACE INDICATOR
         CH   R8,=H'0'        IS IT POSITIVE
         BNH  SETSIGN         NO, JUST SET THE SIGN
ADJUST   DD   0,FPV10         DIVIDE TO ADJUST TO DECIMAL POINT
         BCT  R8,ADJUST       KEEP DIVIDING UNTIL VALUE IS RIGHT

SETSIGN  CH   R9,=H'11'       R9 HAS SIGN DIGIT.  IS IT X'B'?
         BE   ISNEG           YES, THE VALUE IS NEGATIVE
         CH   R9,=H'13'       R9 HAS SIGN DIGIT.  IS IT X'D'?
         BNE  FINISH          NO.  THE VALUE IS POSITIVE
ISNEG    LDR  2,0             COPY VALUE TO FP REGISTER 2
         SDR  0,0             SET FP REGISTER TO 0
         SDR  0,2             NOW FP 0 HAS BEEN NEGATED
FINISH   BR   R3              ALL DONE.
```

Floating–Point to Packed Decimal Conversion

When considering this conversion, one is presented with a number of technical difficulties in handling the floating–point as well as issues due to the fundamental incompatibility of the two formats under consideration. All are difficult, but none are insurmountable.

Here are some of the issues that must be addressed.

1. How to produce repeatedly the most significant decimal digit from a floating–point representation, and when to stop producing these digits. The floating–point formats" call for single–precision to have 7 digits significant, and double–precision to have 15 digits significant. Do we stop at these counts?

   Consider the number $1.2345 \bullet 10^7$, which of course will be represented in the IBM formats using hexadecimal notation. We want to extract the exponent as 7, and then extract the digits 1, 2, 3, 4, and 5 in that order. The answer might be **`012345000C`**.

2. As a related issue, how at any time to determine the largest power of ten that is not larger than the absolute value of the floating–point number being converted. The simple way to do this appears to be quite verbose and tedious.

3. The position of the decimal point in any floating–point representation is important and almost explicitly specified in the representation. The position of the decimal point for the Packed Decimal format is assumed in the code and not stored in the format. We got around this for the Packed Decimal to Floating–Point conversion by by specifying the position of the decimal, but this not a part of the standard.

One obvious way to convert from floating–point to packed decimal is first to convert the value to a fullword integer and then to convert that value (using CVD) to Packed Format. This will work for floating–point values that represent integers within the proper range, but any fractional digits will be lost.

This issue of converting from Floating–Point to Packed Decimal is closely related to the problem of providing a print representation for floating–point values. As the author of your textbook, I intend to continue investigating these two conversion issues.

Any solutions found that are suitable for publishing in a textbook will appear in the next revision.

Here is the complete code for the floating–point to integer conversion.

```
         MVI  THESIGN,X'00'    Initialize the sign field
         L    R9,EFLOAT        Load the floating-point value
         CH   R9,=H'0'         and examine the sign bit.
         BZ   DONE             The value is zero, nothing to do.
         BNL  NOTNEG           Is the value negative?
         MVI  THESIGN,X'80'    Yes, it is negative.
         N    R9,=X'7FFFFFFF'  Zero out the sign bit.

NOTNEG   LR   R8,R9            Copy the value into R8
         N    R8,=X'00FFFFFF'  Examine the fraction.  Is it 0?
         BNZ  FRNZ             No, keep on working
         SR   R9,R9            Yes, the value is zero.  So set
         B    DONE             the result as 0 and exit.

FRNZ     LR   R8,R9            Copy the value into R8
         N    R8,=X'FF000000'  Isolate the characteristic field
         SRL  R8,24            Shift to least significant byte
         CH   R8,=H'64'        Is exponent big enough?
         BH   OVER1            Yes, number is not < 1.
         SR   R9,R9            No, set result to zero
         B    DONE             and be done with it.

OVER1    CH   R8,=H'72'        Is the exponent too big?
         BNH  CANDO            NO, it is fine.
         L    R9,=X'7FFFFFFF'  This is the biggest positive number
         B    SETVAL           Go adjust the sign.

CANDO    SR   R8,R8           Set R8 to zero
         SLDL R8,8            Shift two high-order digits into R8
         CH   R8,=H'72'       Is the exponent an 8?
         BL   DOIT            Yes, we can continue
         CH   R9,=H'0'        Is the sign bit set?
         BP   DOIT            No, the high-order fraction bit is 0
         L    R9,=X'7FFFFFFF'  Set to the biggest positive integer
         B    SETVAL           Go adjust the sign.

DOIT    SH   R8,=H'72'        Produce (Characteristic – 72)
        LCR  R8,R8            Produce (72 – Characteristic)
        SLL  R8,2             Multiply by 4
        SRL  R9,0(R8)         Shift R9 by the amount in R8

SETVAL   SR   R8,R8           Set R8 to 0.
         IC   R8,THESIGN      Load the sign value
         CH   R8,=H'0'        Is the sign bit set?
         BZ   ISPOS           No, we are OK
         LCR  R9,R9           Negate the absolute value

DONE     EQU  *               We are done here.
```