Declaring Floating Point Data

There are three ways to declare floating–point storage. These are

- **E** Single–precision floating point,
- **D** Double–precision floating point, and
- **L** Extended–precision floating point.

The lengths of these data types are as follows:

- **E** 32 bits or 4 bytes.
- **D** 64 bits or 8 bytes.
- **L** 128 bits or 16 bytes.

We shall focus on the first two data types (E and D). These evolved from the 48–bit floating point data type used on the IBM 7094, a predecessor of the IBM 360.

The creation of two related data types was a compromise necessary to drop the 48–bit data type.
Details of the Format

The floating–point number consists of a sign, characteristic (exponent), and a fraction. The bit allocations for each of the three formats is shown below.

<table>
<thead>
<tr>
<th>Format</th>
<th>Length</th>
<th>Sign bit</th>
<th>Characteristic</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>4 bytes</td>
<td>0</td>
<td>1 – 7</td>
<td>8 – 31</td>
</tr>
<tr>
<td>D</td>
<td>8 bytes</td>
<td>0</td>
<td>1 – 7</td>
<td>8 – 63</td>
</tr>
<tr>
<td>L</td>
<td>16 bytes</td>
<td>0</td>
<td>1 – 7</td>
<td>8 – 127</td>
</tr>
</tbody>
</table>

Recall that IBM numbers bits from left to right.

The sign bit, in bit position 0, applies to the fraction. The two values are:

0 the number is not a negative number; it is either positive or zero.
1 the number is a negative number.

Byte 0 (the most significant byte) or each representation holds both the sign for the fraction, and the characteristic field, which holds the exponent.
Details of the Exponent Field

Bits 1 – 7 of each format hold the characteristic field, which is defined to be the exponent in excess–64 notation; it is \((\text{exponent} + 64)\).

As a seven–bit unsigned integer can store values in the range 0 through 127, we have \(0 \leq (\text{exponent} + 64) \leq 127\), or

\[-64 \leq \text{exponent} \leq 63.\]

The leftmost byte of the format stores both the sign and exponent.

<table>
<thead>
<tr>
<th>Bits</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>Sign</td>
<td>Exponent in Excess–64 format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Examples**

Positive number, \(\text{Exponent} = -8\) \(E + 64 = 56 = 48 + 8 = \text{X’38’ = B’011 1000’}.

<table>
<thead>
<tr>
<th>Sign</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value stored in the leftmost byte is 0011 1000 or 38.
Normalized Formats

All floating point formats are of the form \((S, E, F)\) representing \((-1)^S \cdot B^E \cdot F\)

- \(S\) the sign bit, 1 for negative and 0 for non-negative.
- \(B\) the base of the number system; one of 2, 10, or 16.
- \(E\) the exponent.
- \(F\) the fraction.

The IBM 370 format uses base 16.

Each of the formats represents the numbers in normalized form.

For IBM 370 format, this implies that \(0.0625 < F \leq 1.0\). Note \((1/16) = 0.0625\).

In byte addressing, the layout of the E and D formats is as follows.

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>S</td>
<td>E</td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sign &amp; Exponent</td>
<td>Fraction 24 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>S</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sign &amp; Exponent</td>
<td></td>
<td>Fraction 56 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Converting Decimal to Hexadecimal

The first step in producing the IBM 370 floating point representation of a real number is to convert that number into hexadecimal format.

The process for conversion has two steps,
    one each for the integer and fractional part.

**Example:** Represent 123.90625 to hexadecimal.

Conversion of the integer part is achieved by repeated division with remainders.

\[
\begin{align*}
123 / 16 & = 7 \text{ with remainder } 11 \text{ X'B'} \\
7 / 16 & = 0 \text{ with remainder } 7 \text{ X'7'}. \\
\end{align*}
\]

Read bottom to top as X’7B’. Indeed \(123 = 7 \cdot 16 + 11 = 112 + 11\).

Conversion of the fractional part is achieved by repeated multiplication.

\[
\begin{align*}
0.90625 \cdot 16 & = 14.5 \quad \text{Remove the 14 (hexadecimal E)} \\
0.5 \cdot 16 & = 8.0 \quad \text{Remove the 8}. \\
\end{align*}
\]

The answer is read top to bottom as E8.

The answer is that 123.90625 in decimal is represented by \(X’7B.E8’\).
Converting Decimal to IBM 370 Floating Point Format

The decimal number is 123.90625.

Its hexadecimal representation is 7B.E8.

Normalize this by moving the decimal point two places to the left. The number is now $16^2 \cdot 0.7BE8$.

The sign is 0, as the number is not negative.

The exponent is 2, $E + 64 = 66 = X’42’$. The leftmost byte is $X’42’$.

The fraction is 7BE8.

The left part of the floating point data is 427BE8.

In single precision, this would be represented in four bytes as 42 78 E8 00.
Floating–Point Registers

In addition to the 16 general–purpose registers, the IBM System/370 supports four floating–point registers. Later models support more.

Each floating–point register is a 64 bit register, suitable to hold the D data type.

These registers are numbered 0, 2, 4, and 6.

The even numbering was probably intended to reinforce the idea that each of these registers has twice the length of a general–purpose register.

It is the type of the instruction that determines that a register number refers to a floating–point register rather than a general–purpose register.

Consider the following code.

```
R4 EQU 4
AR R4, R4 ADD REGISTER, REFERING TO A GENERAL–PURPOSE INTEGER REGISTER.
ADR R4, R4 ADD DOUBLE PRECISION FLOAT REGISTER, REFERING TO THE FLOATING POINT REG.
```
Conversion Between Single and Double Precision

Recall that the structure of the two precisions is quite similar.

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Sign &amp; Exponent</td>
<td>24 bit fraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 hexadecimal digits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Sign &amp; Exponent</td>
<td>56 bit fraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 hexadecimal digits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To convert single-precision floating point to double-precision floating-point, just add eight hexadecimal zeroes at the end.

Thus single precision \( \text{42 78 E8 00} \) becomes double precision \( \text{42 78 E8 00 00 00 00 00} \).

Conversion from double-precision to single-precision might involve loss of accuracy. It might be done by rounding or truncation.

Thus, double precision \( \text{42 78 E8 04 A8 00 00 00} \) might become \( \text{42 78 E8 04} \) or it might become \( \text{42 78 E8 05} \).
Loading a Floating Point Register

When a floating-point register is loaded from a double-precision floating-point value, the process is simple: copy the 64-bit value into the 64-bit register.

When a floating-point register is loaded from a single-precision floating-point value, the process involves implicit conversion to double precision as shown above.

The 32-bit value represented by the single-precision float is loaded into the leftmost 32 bits of the floating point register and the right 32 bits are set to 0.

Consider the process of loading the decimal number 123.90625, represented as \texttt{42 78 E8 00} into a floating point register.

The value stored in the floating-point register is \texttt{42 78 E8 00 00 00 00 00}, which is the double-precision equivalent to the value loaded.
Range of the Standards

Given that the base of the exponent is 16, the range for these IBM formats is impressive.

The range is from somewhat less than $16^{-64}$ to a bit less than $16^{63}$.

Note that $16^{63} = (2^4)^{63} = 2^{252}$, and

$16^{-64} = (2^4)^{-64} = 2^{-256} = 1.0 / (2^{256})$

Recall that $\log_{10}(2) = 0.30103$.

Using this, we compute the maximum number storable at about

$(10^{0.30103})^{252} \approx 10^{75.86} \approx 9 \cdot 10^{75}$.

We may approximate the smallest positive number at $1.0 / (36 \cdot 10^{75})$ or about $3.0 \cdot 10^{-77}$.

The following non–negative real numbers can be represented in this standard:

$X = 0.0$ and

$3.0 \cdot 10^{-77} < X < 9 \cdot 10^{75}$. 
Precision of the Standards

The precision is dependent on the format used, depending on the number of bits used to represent the fraction.

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.

- \( 2^{24} = 16,777,216 \)
- \( 2^{56} \approx (10^{0.30103})^{56} = 10^{16.85} \approx 9\cdot10^{16} \)
- \( 2^{120} \approx (10^{0.30103})^{120} = 10^{36.12} \approx 1.2\cdot10^{36} \)

### Summary

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
<th>Positive Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>E</td>
<td>( 3.0\cdot10^{-77} &lt; X &lt; 9\cdot10^{75} )</td>
<td>7 digits</td>
</tr>
<tr>
<td>Double Precision</td>
<td>D</td>
<td>( 3.0\cdot10^{-77} &lt; X &lt; 9\cdot10^{75} )</td>
<td>16 digits</td>
</tr>
<tr>
<td>Extended Precision</td>
<td>L</td>
<td>( 3.0\cdot10^{-77} &lt; X &lt; 9\cdot10^{75} )</td>
<td>36 digits</td>
</tr>
</tbody>
</table>
Examples of Floating Point Format (Page 1)

Example 1: True 0
The number 0.0, called “true 0” by IBM, is stored as all zeroes.
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 $X \cdot 16^E$, with $0.625 \leq X < 1.0$.

As $128 \leq X < 256$, the number is converted to the form $X \cdot 16^2$.
Note that $128 = (1/2) \cdot 16^2 = (8/16) \cdot 16^2$, and $0.5 = (1/512) \cdot 16^2 = (8/4096) \cdot 16^2$.
Hence, the value is $128.50 = (8/16 + 0/256 + 8/4096) \cdot 16^2$; it is $16^2 \cdot 0x0.808$.

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sign</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>0000 0010</td>
</tr>
<tr>
<td>Hex value</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

The fractional part comprises six hexadecimal digits, the first three of which are 808. The number 128.50 is represented as 4280 8000.
Examples of Floating Point Format (Page 2)

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 $16^2 \cdot 0x0.808$.

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Sign</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>0 0 1 0</td>
</tr>
<tr>
<td>Hex value</td>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>

The number 128.50 is represented as \textbf{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 $16^0 \cdot 0.375 = 16^0 \cdot (6/16)$. This is in the required format $X \cdot 16^E$, 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sign</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Hex value</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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…..
Examples of Floating Point Format (Page 5)

Example 5: A Full Conversion (Continued)
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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sign</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>10 00 01 0</td>
</tr>
<tr>
<td>Hex value</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

The number 123.45 is represented in single precision as \( \text{427B 3333.} \)
The number is represented in double precision as \( \text{427B 3333 3333 3333.} \)

Example 6: One in “Reverse”
We are given the single precision representation of the number. It is \( \text{4110 0000.} \)
What is the value of the number stored? Begin by examination of the first two hex digits.

<table>
<thead>
<tr>
<th>Field</th>
<th>Sign</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>10 00 00 01</td>
</tr>
<tr>
<td>Hex value</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

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 \( X \cdot 16^1. \) The fraction field is \( 100 000, \) so the value is \( 16^1 \cdot (1/16) = 1.0. \)
More On DC (Define Constant)

The general format of the DC statement is as follows.

<table>
<thead>
<tr>
<th>Name</th>
<th>DC</th>
<th>dTLn ‘constant’</th>
</tr>
</thead>
</table>

The name is an optional entry, but required if the program is to refer to the field by name. The standard column positions apply here.

The declarative, DC, comes next in its standard position.

The entry “dTLn” is read as follows.

- **d** is the optional duplication factor. If not specified, it defaults to 1.
- **T** is the required type specification; usually either E and D.
  Note that the data actually stored at the location does not need to be of this type, but it is a good idea to restrict it to that type.
- **L** is an optional length of the data field in **bytes**.

The ‘constant’ entry is required and is used to specify a value.
If the length attribute is omitted, the length is specified implicitly by this entry.
Again, it is rarely desirable to specify a length for the E and D data types.
Examples of Floating–Point Declaratives

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** DC E’12.34’

Define a single precision value.

**FL3** DC E’−12.34’

The negative of the above value.

**DL2** DC D’0.0’

The constant 0.0, in double precision.

**DL3** DS D’0.0’

Just another storage allocation. Value is not initialized. ‘0.0’ is just a comment.
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.

The first two instructions are type RX and the last two are type RR.
Type RX Floating–Point Loads

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 \( \text{OP } R_1, D_2(X_2, B_2) \).

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>4</td>
<td>R1,D2(X2,B2)</td>
<td>OP</td>
<td>R_1</td>
<td>X_2</td>
<td>B_2</td>
</tr>
</tbody>
</table>

The first byte contains the 8-bit instruction code.

The second byte contains two 4-bit fields, each of which encodes a register number.
- \( R_1 \) is the floating point register to be loaded.
- \( X_2 \) is the general–purpose register used as an index register.

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.
Type RR Floating–Point Loads

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>2</td>
<td>R1,R2</td>
<td>OP</td>
<td>R1 R2</td>
</tr>
</tbody>
</table>

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 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)**.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Operands</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>4</td>
<td>R1,D2(X2,B2)</td>
<td>OP</td>
<td>R₁</td>
<td>X₂</td>
<td>B₂</td>
</tr>
</tbody>
</table>

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.
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 MULTIPLE OF 4.
FL2 DC D‘45678.90’ A DOUBLE PRECISION FLOATING POINT CONSTANT. ADDRESS IS MULTIPLE OF 8
FL3 DS E JUST RESERVE AN ALIGNED FULLWORD
FL4 DS D RESERVE AN ALIGNED DOUBLE WORD.

The DS E and DC E declaratives align the address of the memory area as a multiple of 4.

The DS D and DC D declaratives align the address of the memory area as a multiple of 8.
Addition and Subtraction

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Opcode</th>
<th>Operand Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>Add single precision</td>
<td>7A</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>AD</td>
<td>Add double precision</td>
<td>6A</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>AER</td>
<td>Add register single precision</td>
<td>3A</td>
<td>R1,R2</td>
</tr>
<tr>
<td>ADR</td>
<td>Add register double precision</td>
<td>2A</td>
<td>R1,R2</td>
</tr>
<tr>
<td>SE</td>
<td>Subtract single precision</td>
<td>7B</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>SD</td>
<td>Subtract double precision</td>
<td>6B</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>SER</td>
<td>Subtract register single precision</td>
<td>3B</td>
<td>R1,R2</td>
</tr>
<tr>
<td>SDR</td>
<td>Subtract register double precision</td>
<td>2B</td>
<td>R1,R2</td>
</tr>
</tbody>
</table>

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.
Addition Example: Adjusting the Characteristic

Here is an example of adjusting the characteristic.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>29000</td>
</tr>
<tr>
<td>40</td>
<td>12000</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
41 & \quad 29000 = \quad 41 \quad 29000 \\
40 & \quad 12000 = \quad 41 \quad 01200 \\
\hline
    & \quad 41 \quad 2A200
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
<td>940000</td>
</tr>
<tr>
<td>41</td>
<td>760000</td>
</tr>
<tr>
<td>41</td>
<td>10A0000</td>
</tr>
</tbody>
</table>

Which becomes \(42 \ 10A000\).

Recall that in hexadecimal addition we have \(9 + 7 = 10\).
Multiplication

There are four distinct floating-point multiplication operations.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME</td>
<td>Multiply single precision</td>
<td>7C</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>MD</td>
<td>Multiply double precision</td>
<td>6C</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>MER</td>
<td>Multiply single (register)</td>
<td>3C</td>
<td>R1,R2</td>
</tr>
<tr>
<td>MDR</td>
<td>Multiply double (register)</td>
<td>2C</td>
<td>R1,R2</td>
</tr>
</tbody>
</table>

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.

Sample code segment

```
M1      LE    2, MULTCAN1       LOAD FIRST MULTIPLICAND
ME      2, MULTIPLIER         DO A MULTIPLICATION
LE    4, MULTCAN2            LOAD ANOTHER VALUE
MER    4, 2                   REG 4 GETS THE PRODUCT OF
                                 THE THREE VALUES.
```
Division

There are four distinct floating-point division operations.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Divide single precision</td>
<td>7D</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>DD</td>
<td>Divide double precision</td>
<td>6D</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>DER</td>
<td>Divide single (register)</td>
<td>3D</td>
<td>R1,R2</td>
</tr>
<tr>
<td>DDR</td>
<td>Divide double (register)</td>
<td>2D</td>
<td>R1,R2</td>
</tr>
</tbody>
</table>

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.

Sample code segment

D2     LD 2,DIVIDND1 LOAD FIRST DIVIDEND
DD 2,DIVSOR FIRST DIVISION
LD 4,DIVIDND2 LOAD SECOND DIVIDEND
DDR 4,2 ANOTHER DIVISION

DIVIDND2 / (DIVIDND1 / DIVSOR)
or (DIVIDND2 * DIVSOR) / DIVIDND1
Comparison

There are four distinct floating-point division comparison operations.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Action</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CE</td>
<td>Compare single precision</td>
<td>79</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>CD</td>
<td>Compare double precision</td>
<td>69</td>
<td>R1,D2(X2,B2)</td>
</tr>
<tr>
<td>CER</td>
<td>Compare single (register)</td>
<td>39</td>
<td>R1,R2</td>
</tr>
<tr>
<td>CDR</td>
<td>Compare double (register)</td>
<td>29</td>
<td>R1,R2</td>
</tr>
</tbody>
</table>

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.