Real numbers will be approximated by numbers rounded to a fixed number of decimal or binary digits. We will look at these for both our six-digit ±`EENMMM` representation and the double-precision floating-point representation (`double`), stored using 8 bytes or 64 bits.

Recall that four bits can be written as a hexadecimal digit, with the following substitutions:

```
0 0000   1 0001   2 0010   3 0011   4 0100   5 0101   6 0110   7 0111
8 1000   9 1001   a 1010   b 1011   c 1100   d 1101   e 1110   f 1111
```

For example, `0xece204` in binary is `1110 1100 1110 0010 0000 0100` or `111011001110001000000100`. As a `double` is stored using 8 bytes, it can be written using sixteen hexadecimal digits. For example, we will see why π is stored as either `+493142` or `0x400921fb54442d18`.

The sign of a number is stored using an explicit sign in our ±`EENMMM` representation, and as the first bit of `double`, where `0` represents a positive number and `1` represents a negative number.

1 Scientific notation and normal numbers

In the ±`EENMMM` representation, each real number is rounded to four decimal digits, while for `double`, it is fifty-three bits.

After the number is rounded, it is written in scientific notation, with one digit (decimal or binary) to the left of the radix point, which is called the *leading digit*. To ensure that there is a unique representation, we require that the leading digit is not zero. This is a normal form, and therefore numbers where the stored value has a non-zero leading digit are called normal numbers.

The digits after the radix point are called either the *fractional part* or the *mantissa*. Together, the leading digit and mantissa are called the *significand*, because these represent the significant digits. The significand is multiplied by 10^e and 2^e, respectively, for an appropriate integer exponent e. Thus, our numbers are in the format

$$n.m_1m_2m_3 \times 10^e \text{ and } 1.b_1b_2b_3b_4\cdots b_{52} \times 2^e.$$

Scientific notation requires that the digit n be non-zero, and the only non-zero bit is 1.

For the ±`EENMMM` representation, the leading digit and three digits of the mantissa are stored in the last four digits `NMMM`. Because $n.m_1m_2m_3$ is in scientific notation, the `N` cannot be `0`. For `double`, in scientific notation, we have $1.b_1b_2b_3b_4\cdots b_{52}$, so the last 52 bits store the mantissa. We do not store the leading bit `1`, because we assume it is there, and thus, we can store an extra bit of precision in the mantissa.

Remark

One of the most common mistakes is to forget that the leading bit `1` is not stored, but implied.

In the ±`EENMMM` representation, two decimal digits are used to store the exponent, while for `double`, it is eleven bits, stored immediately after the sign bit. This allows 100 and $2^11 = 2048$ possible exponents, respectively. For `double`, the sign bit and the exponent constitute 12 bits, and therefore may be read by looking at the first three hexadecimal digits.

Rather than storing the exponent in tens- and twos-complement, we will add a *bias* to the exponent, and then store that number. The bias will be, in both cases, the one subtracted from the maximum number of exponents divided by two, resulting in 49 and 1023 =`0x3ff` = `0b01111111111`, respectively.

Consequently, if the exponent is 0, the stored value will be `49` and `0x3ff`, respectively.

In both cases, the lowest exponent is reserved for de-normalized numbers (including zero) and the largest exponent is reserved for infinity and not-a-number (or the result of undefined operations).

Thus, the exponents `01` through `98` can store exponents 10^{-48} to 10^{49}, and the exponents `0x001` through `0xffe` = `0b11111111110` can store 2^{-1022} and 2^{1023}.

Therefore, the smallest and largest positive numbers in scientific notation we can store in each representation are 1.000×10^{-48} and 9.999×10^{49}, and 1.0×2^{-1022} and $1.fffffffffffff_{16} \times 2^{1023}$ or

$$1.1111111111111111111111111111111111111111111111111111_2 \times 2^{1023},$$

respectively, although it is easiest to think of these as numbers in the ranges $[10^{-48}, 10^{50})$ and $[2^{-1022}, 2^{1024})$.

Examples

1	`+491000`	`0x3ff0000000000000`
-1	`-491000`	`0xbff0000000000000`
2	`+492000`	`0x4000000000000000`
-0.5	`-485000`	`0xbfe0000000000000`

Table 1: Examples

The value 1970 would be stored as `+521970`, and as 1970 in binary is 11110110010_2, we have that it equals $1.1110110010_2 \times 2^{10}$. The mantissa is `111011001000000...0`, and converting each four bits to a hexadecimal character, we get `ec80000000000`. Adding a decimal 10 to `3ff` yields `409`, so 1970 is stored as `409ec80000000000`.

On the other hand, `0x406ece0000000000` has an exponent seven higher than `0x3ff`, so the power is 2^7. The mantissa is 111011001110, so the significand is 1.111011001110_2, so the number is $1.111011001110_2 \times 2^7$ or 11110110.01110_2. The integer component is 246, and the fractional part is $\frac{1}{4} + \frac{1}{8} + \frac{1}{16} = \frac{7}{16}$ or 0.4375. Thus, `0x406ece0000000000` stores the value 246.4375.

Finally, $\pi \approx 3.14159265358979323846$, and rounded to four digits is 3.142×10^0, so this is stored as `+493142`. If we write π in binary to 80 bits after the radix point, we have

$$\pi \approx 11.00100100001111110110101010001000100001011010001100001000110100110001001100011010_2,$$

and rounded to 53 bits, we have $11.001001000011111101101010100010001000010110100011000_2$ or

$$1.1001001000011111101101010100010001000010110100011000_2 \times 2^1.$$

Thus, the exponent is `0x3ff` plus 1, or `0x400` and ignoring the leading 1, we write each subsequent group of four bits as a single hexadecimal character: `921fb54442d18`, so the representation is `400921fb54442d18`.

2 Infinity and not-a-number

Any number larger than 10^{50} and 2^{1024}, respectively, is stored as $+\infty$, which is represented as `+990000` and `7ff0000000000000`, respectively. Similarly, $-\infty$ is represented with `-990000` and `fff0000000000000`, respectively, where for `double`, the leading bit is now `1`.

These are used to represent any real number too large in magnitude to store in the corresponding representation.

When a computation is *undefined* (for example, $\frac{0}{0}$, $0 \cdot \infty$ or $\infty - \infty$), we represent this with a value called not-a-number or `NaN`, and it has the first digit or bit of the mantissa set to 1 when the exponent is the largest possible, so `+991000` and `7ff8000000000000`, respectively. Remember that `0x8` is `0b1000`.

3 Zero and subnormal numbers

Note, *subnormal* numbers used to be referred to as *denormalized* numbers.

If the exponent is `00` or `0x000`, respectively, then we will allow the leading digit to be zero, so the number will not be recorded using scientific notation. In each case, the multiplier is 10^{-49} and 2^{-1022}, respectively. Thus, the smallest non-zero number that can be represented is $0.001 \times 10^{-49} = 10^{-53}$ and

$$0.0000000000000000000000000000000000000000000000000001 \times 2^{-1022} = 2^{-52} \cdot 2^{-1022} = 2^{-1074}.$$

Of course, if all digits in a subnormal number are zero, then this equals 0.0×2^{-1022} which equals zero.

One interesting point is that we have signed zeros, because ever real number greater than 5×10^{-54} rounds up to be represented as `+000001`, while all real numbers in the interval $[0, 10^{-54}]$ round down to zero and are thus represented by `+000000`. Similarly, `-000000` represents all real numbers in the interval $[-10^{-54}, 0]$. Similarly, while all real numbers greater than 2^{-1075} round up to 2^{-1074}, and are thus represented as `0x0000000000000001`, all real numbers in the interval $[0, 2^{-1075}]$ would round down to zero and thus be represented by `0x0000000000000000`. Similarly, all real numbers in the interval $[-2^{-1075}, 0]$ would be represented by -0 or `0x8000000000000000`.

4 Changing sign

To take the absolute value of a `double`, make the first bit a `0`. If the format is using hexadecimal digits, the number is negative if the first digit is `8` through `f`, and to make such a number positive, subtract `8` from that first digit.

To multiply a `double` by -1, flip the first bit, or calculate the exclusive-or of the representation and `0b100000...0`. If the format is using hexadecimal digits, if the number is less than eight, add `8` to make it negative, otherwise subtract `8` to make it positive.

5 Comparing two representations

The beauty of both representations here is that if two numbers are positive, we can tell which is largest simply by looking at the two numbers as if they were positive integers. For example, given `+796947`, `+533271`, `+332311`, `+976241`, `+847034`, `+599938`, `+827005`, we can sort these simply by placing them in order as if they were integers: `+332311`, `+533271`, `+599938`, `+796947`, `+847034`, `+827005`, and `+976241`. Similarly,

```
0x3fe97759468dee08   0x3f98dbe806f65de0   0x3fb81d838252f0a8   0x3fee331238629281
0x3fe25841e8a9614a   0x3feeeda367d96252   0x3fed29923ed54883   0x3fd967402bd818c4
```

are all positive, and looking at the exponents, they are all between 2^{-6} and 1 because the smallest exponent is `0x3f9` which is six less than `0x3ff`. To sort them, we just sort them as if they were hexadecimal integers:

```
0x3f98dbe806f65de0   0x3fb81d838252f0a8   0x3fd967402bd818c4   0x3fe25841e8a9614a
0x3fe97759468dee08   0x3fed29923ed54883   0x3fee331238629281   0x3feeeda367d96252
```

The first three are sorted because the first two digits are equal and `9` $<$ `b` $<$ `d` $<$ `e`, the next three are sorted because the first three digits are equal and `2` $<$ `9` $<$ `d` $<$ `e`, and finally the last two are sorted because the first four digits are equal and `3` $<$ `e`.

This works because the exponents determine the magnitude, and by using a bias, the exponents are stored in such a way that smaller most values in the most digits imply smaller values. If two exponents are the same, then the mantissas determine which of the numbers is smaller.

For example, all `+49NMMM` (1.000 up to 9.999) are less than `+501000` (= 10), and they are also all greater than `489999` (= 0.9999).

6 Arithmetic

Here, we will give some basic examples of floating-point arithmetic using both our ±`EENMMM` and `double` representations.

6.1 Adding two numbers with the same sign with the same exponent

When adding two numbers with the same exponent, we simply add the leading digit and mantissa as per normal. The answer must be written in scientific notation to the required number of digits. For example, adding `+358559` and `+354716`, we have

$$\begin{array}{rrl} & 8.559 & \times 10^{35-49} \\ + & 4.716 & \times 10^{35-49} \\ \hline & 13.275 & \times 10^{35-49} \end{array}$$

and rounded to four digits (raising the odd 7 because all subsequent digits are $500\cdots$) and shifting the decimal point, we get $1.328 \times 10^{(35+1)-49}$, so we store `+361328`. You'll notice we didn't even have to calculate the actual exponent of the solution, which would be 1.328×10^{-13}.

As another example, when adding `0x38e5f0...0` and `0x38ea20...0`, we have

$$\begin{array}{rrl} & 1.01011111 & \times 2^{\texttt{0x38e}-\texttt{0x3ff}} \\ + & 1.10100010 & \times 2^{\texttt{0x38e}-\texttt{0x3ff}} \\ \hline & 11.00000001 & \times 2^{\texttt{0x38e}-\texttt{0x3ff}} \end{array}$$

and given that this is significantly below 53 bits, we don't need to round, but to write it in scientific notation, we must shift the radix point to get $1.100000001 \times 2^{(\texttt{0x38e}+1)-\texttt{0x3ff}}$. Recalling that we do not store the leading 1, we break the mantissa into groups of four bits starting from the radix point, `1000 0000 1000 ...` to see that the result is now `0x38f8080...0`.

Matlab

You can test this in Matlabas follows:

```
format long
a = hex2num( '38e5f' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = 1.320315254843593e-34
```

```
b = hex2num( '38ea2' )

    b = 1.572341243659892e-34


c = a + b

    c = 2.892656498503485e-34


format hex
c

    c = 38f8080000000000
```

Test yourself by adding `0x5e3710...0` and `0x5e3ed0...0`, and check your answer in MATLAB.

6.2 Adding two numbers with the same sign with nearby exponents

When adding two numbers with different but nearby exponents, we must first get the same multiplier before we add. Again, the answer must be written in scientific notation to the required number of digits. For example, adding `+579453` and `+593148`, we have

$$\begin{array}{rrl} & 9.453 & \times 10^{57-49} \\ + & 3.148 & \times 10^{59-49} \\ \hline \end{array}$$

To get the same exponent, we add two to the first exponent, meaning we must shift the decimal point two to the left (for example, $3.5 \times 10^3 = 0.035 \times 10^5$):

$$\begin{array}{rrl} & 0.09453 & \times 10^{59-49} \\ + & 3.148 & \times 10^{59-49} \\ \hline & 3.24253 & \times 10^{59-49} \end{array}$$

and rounded to four digits (raising the 2 because the next digit is 5 but not $500\cdots$), we store `+593243`. You'll notice we didn't even have to calculate the actual exponent of the solution, which would be 3.243×10^{10}.

As another example, when adding `0x5a6e70...0` and `0x5a2830...0`, we have

$$\begin{array}{rrl} & 1.11100111 & \times 2^{\texttt{0x5a6}-\texttt{0x3ff}} \\ + & 1.10000011 & \times 2^{\texttt{0x5a2}-\texttt{0x3ff}} \\ \hline \end{array}$$

We observe that the exponents differ by 4, and so to get a common multiplier, we add 4 to the second exponent and move the radix point to the left by four:

$$\begin{array}{rrl} & 1.11100111 & \times 2^{\texttt{0x5a6}-\texttt{0x3ff}} \\ + & 0.000110000011 & \times 2^{\texttt{0x5a2}+4-\texttt{0x3ff}} \\ \hline & 1.111111110011 & \times 2^{\texttt{0x5a6}-\texttt{0x3ff}} \end{array}$$

and given that this is significantly below 53 bits, we don't need to round. Recalling that we do not store the leading 1, we break the mantissa into groups of four bits starting from the radix point, `1111 1111 0011` to see that the result is now `0x5a6ff30...0`.

Matlab

You can test this in MATLABas follows:

```
format long
a = hex2num( '5a6e7' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = 4.120758487050894e+127


b = hex2num( '5a283' )

    b = 2.046629279374610e+126


c = a + b

    c = 4.325421414988355e+127


format hex
c

    c = 5a6ff30000000000
```

Test yourself by adding `0x5e3710...0` and `0x5e3ed0...0`, and check your answer in MATLAB.

6.3 Adding two numbers with the significantly different exponents

When adding two numbers with significantly exponents, we get into the situation where $x + y = x$ even though y is not zero when $|x| \gg |y|$ (much greater). For our $\pm$`EENMMM` representation, the numbers are sufficiently different if the exponents differ by 5 or more. For example, adding `+873215` and `+793104`, we see that the difference in the exponents is $8 \geq 5$, so the result is the larger of the two numbers in absolute value, which is the first: `+873215`.

For `double`, this difference in exponent is 54, so for example, when adding `0x2482040...0` and `0xb5fece0...0`, we first note the second number is negative, so its absolute value is `0x35fece0...0`. We next note the difference in exponents is greater than `0x100`, and $16^2 > 10^2 \geq 54$, so the result must be the larger of the two original numbers in absolute value: `0xb5fece0...0`.

MATLAB

You can test this in MATLABas follows:

```
format long
a = hex2num( '248204' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = 7.931608219673450e-133


b = hex2num( 'b5fece' )

    b = -1.317338906486691e-48


c = a + b

    c = -1.317338906486691e-48


format hex
c

    c = b5fece0000000000
```

Test yourself by adding `0xaf1f320...0` and `0xe2131e0...0`, and check your answer in MATLAB.

Remark

You would never be tested on an example where the difference between the two exponents is close to 54: either the difference will be small (no more than 9) or obviously larger than 54.

6.4 Adding a larger positive number onto a smaller negative number

As $x + (-y) = x - y$, adding a smaller negative number onto a larger positive number can be calculated the same way that you learned subtraction in elementary school. For example, adding `+237532` and `-219548`, we have

$$\begin{array}{rl} & 7.532 \quad \times 10^{23-49} \\ - & 9.548 \quad \times 10^{21-49} \\ \hline \end{array}$$

To get the same exponent, we add two to the second exponent, meaning we must shift the decimal point two to the left:

$$\begin{array}{rl} & 7.532 \quad \times 10^{23-49} \\ - & 0.09548 \quad \times 10^{23-49} \\ \hline & 7.43652 \quad \times 10^{23-49} \end{array}$$

and rounded to four digits (raising the 6 because the next digit is 5 but not $500\cdots$), we store `+237437`. You'll notice we didn't even have to calculate the actual exponent of the solution, which would be 7.437×10^{-12}.

As another example, when adding `0x2e7530...0` and `0xae4f1...0`, we note the second number is negative, so its absolute value can be found by subtracting 8 from the first hexadecimal digit to get `0x2e4f1...0`, so we are calculating:

$$\begin{array}{rl} & 1.01010011 \quad \times 2^{\texttt{0x2e7}-\texttt{0x3ff}} \\ - & 1.11110001 \quad \times 2^{\texttt{0x2e4}-\texttt{0x3ff}} \\ \hline \end{array}$$

We observe that the exponents differ by 3, and so to get a common multiplier, we add 3 to the second exponent and move the radix point to the left by three:

$$\begin{array}{rl} & 1.01010011 \quad \times 2^{\texttt{0x2e7}-\texttt{0x3ff}} \\ - & 0.00111110001 \quad \times 2^{\texttt{0x2e4}+3-\texttt{0x3ff}} \\ \hline & 1.00010100111 \quad \times 2^{\texttt{0x2e7}-\texttt{0x3ff}} \end{array}$$

Recalling that we do not store the leading 1, we break the mantissa into groups of four bits starting from the radix point, `0001 0100 1110` to see that the result is now `0x2e714e0...0`.

If you missed what happened with the subtraction, recall that with subtraction, if the digit being subtracted is larger than the digit being subtracted from, then you must *borrow* from the digit to the left. If that is zero, you must first borrow from the next digit, and so on. For binary, however, when you borrow, the 10 is actually 2, so $2 - 1 = 1$, and when you borrow from 10, you get 1, again, because it is base 2.

MATLAB

You can test this in MATLABas follows:

```
format long
a = hex2num( '2e753' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = 4.120758487050894e+127


b = hex2num( 'ae4f1' )

    b = 2.046629279374610e+126


c = a + b

    c = 4.325421414988355e+127


format hex
c

    c = 5a6ff30000000000
```

Test yourself by adding `0x5e3710...0` and `0xde2ed0...0`, and check your answer in MATLAB.

6.5 Adding a smaller positive number onto a larger negative number

As $x + (-y) = x - y$, adding a larger negative number onto a smaller positive number can be calculated the same way that you learned subtraction in elementary school. However, if you need to calculate $x - y$ when $y > x$, you used the rule that $-(x - y) = y - x$, so you calculated $y - x$ and negated the result. For example, adding `+842354` and `-879326`, we have

$$\begin{array}{rrl} & 2.354 & \times 10^{84-49} \\ - & 9.326 & \times 10^{87-49} \\ \hline \end{array}$$

To get the same exponent, we add three to the first exponent, meaning we must shift the decimal point three to the left:

$$\begin{array}{rrl} & 0.002354 & \times 10^{87-49} \\ - & 9.326 & \times 10^{87-49} \\ \hline \end{array}$$

The first number is smaller in absolute value, so instead, we will calculate:

$$\begin{array}{rrl} & 9.326 & \times 10^{87-49} \\ - & 0.002354 & \times 10^{87-49} \\ \hline & 9.323646 & \times 10^{87-49} \end{array}$$

and rounded to four digits (raising the 3 because the next digit is a 6), we store `-879324`. You'll notice we didn't even have to calculate the actual exponent of the solution, which would be 9.324×10^{38}.

As another example, when adding `0x953090...0` and `0x14fe8...0`, we note the first number is negative, so its absolute value can be found by subtracting 8 from the first hexadecimal digit to get `0x153090...0`, so we subtracting the first number from the second:

$$\begin{array}{rrl} & 1.11101000 & \times 2^{\texttt{0x14f}-\texttt{0x3ff}} \\ - & 1.00001001 & \times 2^{\texttt{0x153}-\texttt{0x3ff}} \\ \hline \end{array}$$

We observe that the exponents differ by 4 (`0x14f + 1 = 0x150`, `0x14f + 2 = 0x151`, etc.), and so to get a common multiplier, we add 4 to the first exponent and move the radix point to the left by four:

$$\begin{array}{rrl} & 0.000111101000 & \times 2^{\texttt{0x14f}+4-\texttt{0x3ff}} \\ - & 1.00001001 & \times 2^{\texttt{0x153}-\texttt{0x3ff}} \\ \hline \end{array}$$

We see now that we are subtracting a larger number from a smaller number, so instead of calculating $x - y$, we will calculate $y - x$ and then negate the result:

$$\begin{array}{rrl} & 1.00001001 & \times 2^{\texttt{0x153}-\texttt{0x3ff}} \\ - & 0.000111101000 & \times 2^{\texttt{0x14f}+4-\texttt{0x3ff}} \\ \hline & 0.111010101 & \times 2^{\texttt{0x153}-\texttt{0x3ff}} \end{array}$$

First we must write this in scientific notation, so we must move the radix point one to the right, so we subtract 1 from the exponent to get $1.11010101 \times 2^{(\texttt{0x153}-1)-\texttt{0x3ff}}$. Now, the exponent is `0x152`, and recalling we do not store the leading 1, we break the mantissa into groups of four bits starting from the radix point, `1101 0101` to see that the result is now `0x152d5...0`, but we must negate this, so we add 8 to the first digit to get `0x952d5...0`.

Matlab

You can test this in MATLABas follows:

```
format long
a = hex2num( '95309' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = -1.289700518574636e-206
```

```
b = hex2num( '14fe8' )

    b = 1.484372294963261e-207


c = a + b

    c = -1.141263289078310e-206


format hex
c

    c = 952d500000000000
```

Test yourself by adding `0x313a30...0` and `0xb15210...0`, and check your answer in MATLAB. Also, try adding these two pairs of numbers: `0x3ff320...0` and `0xbff4a0...0`; and `0x3fff90...0` and `0xbff7d0...0`.

Remark

To create your own examples for addition, choose two random integers with values between 128 and 256. Then, multiply the first by 2^n where n is any value between -500 and 500, and then multiply the second by 2^{n+k} where k is a value between -5 and 5.

```
format hex
a = 123;      % The ';' suppresses the output
b = 214;
a = a*2^-57;
b = b*2^-54;
a

    a = 3ccec00000000000


-a

    ans = bccec00000000000


b

    b = 3d0ac00000000000


-b

    ans = bd0ac00000000000


a + b

    ans = 3d0cac0000000000


a + (-b)

    ans = bd08d40000000000


(-a) + (-b)    % = -(a + b)

    ans = bd0cac0000000000


(-a) + b      % = -(a + (-b))

    ans = 3d08d40000000000
```

6.6 Multiplying two numbers with small exponents

First, we check the signs of both x and y, and if they are both either positive or both negative, the result will be positive; otherwise, the result will be negative.

Next, we will calculate $|x||y|$, where we will multiply the two significands and add the exponents, perhaps shifting the exponent to get the result in scientific notation.

For example, multiplying `+523240` and `-487610`, first, only one is negative, so the product will be negative. Next, we are calculating $(3.240 \times 10^3) \times (7.610 \times 10^{-1})$, so the result will be multiplied by $10^{3+(-1)} = 10^2$. Next, performing the multiplication, we have

$$\begin{array}{r} 3.240 \\ \times 7.610 \\ \hline 32400 \\ 1944000 \\ +22680000 \\ \hline 24656400 \end{array}$$

and counting the digits after the decimal point, we get that the answer is 24.656400×10^2. We round the answer to four digits, and to get the result into scientific notation, we move the radix point one the left, and therefore add one to the exponent: 2.466×10^3, so our answer is `-522466`.

Remark

Any question asking you to multiply two `double`s will have exponents that are close to 0, so between -10 and 10, so you may see exponents between `3f5` and `409` or, if they are negative numbers, between `bf5` and `c09`.

As another example, when multiply `0xc0390...0` and `0xbfdb0...0`, we note both numbers are negative, so the result will be positive. Next, we note that the two exponents are 2^4 and 2^{-2}, so we will multiply the product of the significands by $2^{4+(-2)} = 2^2$. Next, we multiply the significands:

$$\begin{array}{r} 1.1001 \\ \times 1.1011 \\ \hline 11001 \\ 110010 \\ 11001000 \\ +110010000 \\ \hline 1010100011 \end{array}$$

Counting the digits to the right of the radix point, we get the result is 10.10100011×10^2 and to convert this into scientific notation, we move the radix point one the left and add one to the exponent to get 1.010100011×10^3. The exponent is `0x3ff + 3 = 0x402`. Recalling we do not store the leading 1, we break the mantissa into groups of four bits starting from the radix point, `0101 0001 1000` to see that the result is now `0x4025180...0`.

Matlab

You can test this in MATLABas follows:

```
format long
a = hex2num( 'c039' )  % Matlab assumes all remaining hexadecimal digits are 0

    a = -25


b = hex2num( 'bfdb' )
```

```
    b = -0.421875


c = a*b

    c = 10.546875


format hex
c

    c = 4025180000000000
```

Test yourself by adding `0xbfe280...0` and `0xbfd340...0`, and check your answer in MATLAB.

Remark

To create your own examples for multiplication, choose two random integers with values between 16 and 31. Then, multiply each by a power of two where the power falls between -14 and 4.

```
format hex
a = 23;      % The ';' suppresses the output
b = 19;
a = a*2^3;
b = b*2^-7;
a

    a = 4067000000000000


-a

    ans = c067000000000000


b

    b = 3fc3000000000000


-b

    ans = bfc3000000000000


a*b   % = (-a)*(-b)

    ans = 403b500000000000


a*(-b)   % = (-a)*b = -(a*b)

    ans = c03b500000000000
```

7 Acknowledgements

Aviral Gupta, who noted the exponent of 1970 was not correctly calculated for the hexadecimal representation.