Exploring Binary

Displaying IEEE Doubles in Binary Scientific Notation

By Rick Regan July 1st, 2010

An IEEE double-precision floating-point number, or double, is a 64-bit encoding of a rational number. Internally, the 64 bits are broken into three fields: a 1-bit sign field, which represents positive or negative; an 11-bit exponent field, which represents a power of two; and a 52-bit fraction field, which represents the significant bits of the number. These three fields — together with an implicit leading 1 bit — represent a number in binary scientific notation, with 1 to 53 bits of precision.

For example, consider the decimal number 33.75. It converts to a double with a sign field of 0, an exponent field of 10000000100, and a fraction field of 0000111000000000000000000000000000000000000000000000. The 0 in the sign field means it's a positive number (1 would mean it's negative). The value of 10000000100 in the exponent field, which equals 1028 in decimal, means the exponent of the power of two is 5 (the exponent field value is offset, or biased, by 1023). The fraction field, when prefixed with an implicit leading 1, represents the binary fraction 1.0000111. Written in normalized binary scientific notation — following the convention that the fraction is written in binary and the power of two is written in decimal — 33.75 equals 1.0000111 x 2^5.

In this article, I'll show you the C function I wrote to display a double in normalized binary scientific notation. This function is useful, for example, when verifying that decimal to floating-point conversions are correctly rounded.

Subnormal Numbers

In double-precision floating-point, most numbers are represented in normalized form, with an implicit 1 bit giving 53 bits of precision. However, very small numbers — the so-called subnormal numbers — are represented in unnormalized form, with no implicit leading 1 bit and zero to 51 leading zeros of fraction field. These numbers are encoded with an exponent field of zero, with their true exponent equal to -1022 minus the location of the first 1 bit in their fraction field. This means that subnormal numbers are scaled by powers of two in the range 2^{-1074} through 2^{-1023}, with accompanying precision of one to 52 bits.

Although subnormal numbers are encoded as unnormalized, they can still be written as normalized. For example, the decimal number 1e-310 converts to a subnormal double with a sign field of 0, an exponent field of 00000000000, and a fraction field of 0000000100100110100010001011011100001110011000101011. This can be printed as 1.00100110100010001011011100001110011000101011 x 2^{-1030} — which is what my C function does.

The Code

I wrote a function called **print_double_binsci()** that prints double-precision floating-point numbers in normalized binary scientific notation. It is based on a call to my function parse_double(), which isolates the three fields of a double.

I declared and defined this function in files I named binsci.h and binsci.c, respectively.

binsci.h

```
/***********************************************************/
/* binsci.h: Function to print an IEEE double-precision    */
/*           floating-point number in normalized binary    */
/*           scientific notation                           */
/*                                                         */
/* Rick Regan (https://www.exploringbinary.com)            */
/*                                                         */
/* Version 2 (support subnormals)                          */
/***********************************************************/
void print_double_binsci(double d);
```

binsci.c

```
/***********************************************************/
/* binsci.c: Function to print an IEEE double-precision    */
/*           floating-point number in normalized binary    */
/*           scientific notation                           */
/*                                                         */
/* Rick Regan (https://www.exploringbinary.com)            */
/*                                                         */
/* Version 2 (support subnormals)                          */
/***********************************************************/
#include <stdio.h>
#include "rawdouble.h"
#include "binsci.h"
```

```
void print_double_binsci(double d)
{
 unsigned char sign_field;
 unsigned short exponent_field;
 short exponent;
 unsigned long long fraction_field, significand;
 int i, start = 0, end = 52;

 //Isolate the three fields of the double
 parse_double(d,&sign_field,&exponent_field,&fraction_field);

 //Print a minus sign, if necessary
 if (sign_field == 1)
   printf("-");

 if (exponent_field == 0 && fraction_field == 0)
   printf("0\n"); //Number is zero
 else
  {
   if (exponent_field == 0 && fraction_field != 0)
     {//Subnormal number
      significand = fraction_field; //No implicit 1 bit
      exponent = -1022; //Exponents decrease from here
      while (((significand >> (52-start)) & 1) == 0)
        {
         exponent--;
         start++;
        }
     }
   else
     {//Normalized number (ignoring INFs, NANs)
      significand = fraction_field | (1ULL << 52); //Implicit 1 bit
      exponent = exponent_field - 1023; //Subtract bias
     }

   //Suppress trailing 0s
   while (((significand >> (52-end)) & 1) == 0)
     end--;

   //Print the significant bits
   for (i=start; i<=end; i++)
    {
     if (i == start+1)
       printf(".");
     if (((significand >> (52-i)) & 1) == 1)
       printf("1");
     else
       printf("0");
    }

   if (start == end) //Special case: 1 bit (a power of two)
     printf(".0");

   //Print the exponent
   printf(" x 2^%d\n",exponent);
  }
}
```

Notes

- Numbers that are not raised to a power are printed with the suffix “x 2^0”.
- Not-a-number (NaN) and infinity values are not handled.

Examples of Usage

I wrote a program, called binsciTest.c, that shows some example calls to print_double_binsci():

```
/***********************************************************/
/* binsciTest.c: Program to test printing of IEEE double   */
/*               precision floating-point numbers in       */
/*               binary scientific notation                */
/*                                                         */
/* Rick Regan (https://www.exploringbinary.com)            */
/*                                                         */
/* Version 2 (print subnormals)                            */
/***********************************************************/
#include <stdio.h>
#include "binsci.h"

int main (void)
{
 printf("33.75 =\n");
 print_double_binsci(33.75);
 printf("\n");

 printf("0.1 =\n");
 print_double_binsci(0.1);
 printf("\n");

 printf("-0.6 =\n");
 print_double_binsci(-0.6);
 printf("\n");

 printf("3.518437208883201171875e13 =\n");
 print_double_binsci(3.518437208883201171875e13);
 printf("\n");

 printf("9214843084008499.0 =\n");
 print_double_binsci(9214843084008499.0);
 printf("\n");

 printf("30078505129381147446200.0 =\n");
 print_double_binsci(30078505129381147446200.0);
 printf("\n");

 printf("1777820000000000000001.0 =\n");
 print_double_binsci(1777820000000000000001.0);
 printf("\n");
```

```
 printf("0.3932922657273 =\n");
 print_double_binsci(0.3932922657273);
 printf("\n");

 printf("4.9406564584124654e-324 =\n");
 print_double_binsci(4.9406564584124654e-324);
 printf("\n");

 printf("1.2e-321 =\n");
 print_double_binsci(1.2e-321);
 printf("\n");

 printf("2.2250738585072011e-308 =\n");
 print_double_binsci(2.2250738585072011e-308);

 return (0);
}
```

(Some of these examples were taken from my articles Incorrectly Rounded Conversions in Visual C++ and Incorrectly Rounded Conversions in GCC and GLIBC.)

I compiled and ran it on both Windows and Linux:

- On Windows, I built a project in Visual C++ with files binsci.c, binsci.h, binsciTest.c, rawdouble.c, and rawdouble.h, and compiled and ran it in there.
- On Linux, I compiled with "gcc binsciTest.c binsci.c rawdouble.c -o binsciTest" and then ran it with "./binsciTest".

Output

This is the *Windows* output (the Linux output is a little different; Visual C++ and gcc differ in some of their decimal to floating-point conversions):

```
33.75 =
1.0000111 x 2^5

0.1 =
1.100110011001100110011001100110011001100110011001101 x 2^-4

-0.6 =
-1.0011001100110011001100110011001100110011001100110011 x 2^-1

3.5184372088832011171875e13 =
1.00000000000000000000000000000000000000000000000001 x 2^45

9214843084008499.0 =
1.0000010111100110110011101100010101110111011000011001 x 2^53
```

```
30078505129381147446200.0 =
1.10010111101000111100011100100111000110110000001 x 2^74

1777820000000000000001.0 =
1.100000011000000001101010110111010110100101110001111 x 2^70

0.3932922657273 =
1.1001001010111011001101010010110001000110001000111001 x 2^-2

4.9406564584124654e-324 =
1.0 x 2^-1074

1.2e-321 =
1.1110011 x 2^-1067

2.2250738585072011e-308 =
1.111111111111111111111111111111111111111111111111111 x 2^-1023
```

Except for 33.75, which is exact, all the other examples are 53 significant bit *approximations* to the decimal numbers they stand in for (type '0.1' into my decimal to binary converter and see for yourself).

Related

- Hexadecimal Floating-Point Constants
- Converting Floating-Point Numbers to Binary Strings in C
- What Powers of Two Look Like Inside a Computer

Get articles by RSS (What Is RSS?)

Get articles by e-mail

Numbers in computers / C, Code, Convert to binary, Decimals, Floating-point

2 comments

1. **Rick Regan**

   January 30, 2011 at 12:32 pm

   1/30/11: Enhanced code to print subnormal numbers (and revised article accordingly).

2. Pingback: Standard notation needed for binary numbers as exponents | Physics Forums - The Fusion of Science and Community

Comments are closed.