Exploring Binary

Hexadecimal Floating-Point Constants

By Rick Regan October 4th, 2010

Hexadecimal floating-point constants, also known as *hexadecimal floating-point literals*, are an alternative way to represent floating-point numbers in a computer program. A hexadecimal floating-point constant is shorthand for binary scientific notation, which is an abstract — yet direct — representation of a binary floating-point number. As such, hexadecimal floating-point constants have exact representations in binary floating-point, unlike decimal floating-point constants, which in general do not.

Hexadecimal floating-point constants are useful for two reasons: they bypass decimal to floating-point conversions, which are sometimes done incorrectly, and they bypass floating-point to decimal conversions which, even if done correctly, are often limited to a fixed number of decimal digits. In short, their advantage is that they allow for direct control of floating-point variables, letting you read and write their exact contents.

In this article, I'll show you what hexadecimal floating-point constants look like, and how to use them in C.

Anatomy of a Hexadecimal Floating-Point Constant

0x1.999999999999ap-4 is an example of a normalized, double-precision hexadecimal floating-point constant; it represents the double-precision floating-point number nearest to the decimal number 0.1. The constant is made up of four parts:

- The prefix '0x', which shows it's a hexadecimal constant.
- A one hex digit integer part '1', which represents the leading 1 bit of a normalized binary fraction.
- A thirteen hex digit fractional part '.999999999999a', which represents the remaining 52 significant bits of the normalized binary fraction.
- The suffix 'p-4', which represents the power of two, written in *decimal*: 2^{-4}.

If you replace each hexadecimal digit with its binary equivalent, it translates to binary scientific notation as

1.1001100110011001100110011001100110011001100110011001101 x 2^{-4}.

Single-Precision Hexadecimal Floating-Point Constants

Hexadecimal floating-point constants can represent single-precision floating-point values as well; for example, 0x1.99999ap-4 is the single-precision constant representing 0.1. Single-precision values don't map as neatly to hexadecimal constants as double-precision values do; single-precision is 24 bits, but a normalized hexadecimal constant shows 25 bits. This is not a problem, however; the last hex digit will always have a binary equivalent ending in 0.

(For further details on the syntax of hexadecimal floating-point constants, see pages 57-58 of the C99 specification.)

Examples

Here's a gcc C program I wrote to demonstrate usage of hexadecimal floating-point constants:

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
```

```
int main (void)
{
 double d;

 d = 2;
 printf("Ex 1: 2 in hex: %a\n\n",d);

 d = 256;
 printf("Ex 2: 2^8 in hex: %a\n\n",d);

 d = 0.015625; //= 2^-6
 printf("Ex 3: 2^-6 in hex: %a\n\n",d);

 d = 0.857421875;
 printf("Ex 4: 0.857421875 in hex: %a\n\n",d);

 d = DBL_MAX;
 printf("Ex 5: DBL_MAX in hex: %a\n\n",d);

 d = DBL_MIN; //Smallest double (normalized)
 printf("Ex 6: DBL_MIN in hex: %a\n\n",d);

 d = 0x1p-1074; //Smallest double (unnormalized)
 printf("Ex 7: 0x1p-1074 in hex: %a\n\n",d);

 d = 3.1415926;
 printf("Ex 8: 3.1415926 in upper case hex: %A\n\n",d);

 d = 0.1;
 printf("Ex 9: 0.1 in hex: %a\n\n",d);

 d = 0x3.3333333333334p-5;
 printf("Ex 10: 0x3.3333333333334p-5 in hex: %a\n\n",d);

 d = 0xcc.ccccccccccdp-11;
 printf("Ex 11: 0xcc.ccccccccccdp-11 in hex: %a\n\n",d);

 d = strtod("0x1.999999999999ap-4",NULL);
 printf("Ex 12: strtod 0x1.999999999999ap-4 in decimal: %0.1f\n",d);
}
```

Here is the output of the program:

```
Ex 1: 2 in hex: 0x1p+1

Ex 2: 2^8 in hex: 0x1p+8

Ex 3: 2^-6 in hex: 0x1p-6

Ex 4: 0.857421875 in hex: 0x1.b7p-1

Ex 5: DBL_MAX in hex: 0x1.fffffffffffffp+1023

Ex 6: DBL_MIN in hex: 0x1p-1022

Ex 7: 0x1p-1074 in hex: 0x0.0000000000001p-1022
Ex 8: 3.1415926 in upper case hex: 0X1.921FB4D12D84AP+1

Ex 9: 0.1 in hex: 0x1.999999999999ap-4

Ex 10: 0x3.3333333333334p-5 in hex: 0x1.999999999999ap-4

Ex 11: 0xcc.cccccccccccdp-11 in hex: 0x1.999999999999ap-4

Ex 12: strtod 0x1.999999999999ap-4 in decimal: 0.1
```

- Examples 1, 2, and 3 show how powers of two display in hexadecimal. They are written as the numeral '1' followed directly by the power of two exponent — there's no radix point or fraction.
- Example 4 shows how a dyadic fraction displays in hexadecimal. A dyadic fraction has a terminating binary expansion, so in hexadecimal it may be shorter than 14 hex digits. 0.857421875 = 0.110110111 = 1.10110111 x 2^{-1}, which translates to the hexadecimal constant 0x1.b7p-1. (Negative powers of two are dyadic fractions, which is why they have a short hexadecimal representation.)
- Examples 5, 6, and 7 display the limits of a double-precision variable in hexadecimal:
  - DBL_MAX is the largest positive number, and consists of a binary fraction with all 1 bits and the maximum exponent of 2^{1023}.
  - DBL_MIN is the smallest positive normalized number, which is simply 2^{-1022}.
  - 2^{-1074} is the smallest positive number a double can represent — it is a subnormal number. It is displayed in hexadecimal as *unnormalized* to match its internal structure.
- Example 8 uses the "%A" specifier to print the alphabetic hexadecimal numerals in capital letters (it also prints the exponent letter — P — as a capital).
- Examples 9, 10 and 11 show three ways to represent the decimal fraction 0.1: 0x1.999999999999ap-4, 0x3.3333333333334p-5, and 0xcc.cccccccccccdp-11. The latter two constants, although appearing in the source code as unnormalized, print as normalized. (It's not clear *why* you'd want to enter unnormalized constants, but it's allowed.)
- Example 12 shows that, not only can hexadecimal constants be converted from floating-point literals, but they can be converted from strings — in this case using the strtod() function.

Bypassing Decimal to Floating-Point Conversion

One use for hexadecimal floating-point constants is to bypass decimal to floating-point conversion. Some conversion routines don't always round decimal numbers correctly, whether it be to nearest or to some other direction. This program shows how to bypass two examples of incorrect decimal to floating-point conversions done by gcc/glibc:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
 double gcc_dec, gcc_hex, glibc_dec, glibc_hex;

 gcc_dec = 0.500000000000000166533453693773481063544750213623046875;
 gcc_hex = 0x1.0000000000002p-1;
 printf("gcc (from decimal)   = %a\n",gcc_dec);
 printf("gcc (from hex)       = %a\n",gcc_hex);

 glibc_dec = strtod("62.5364939768271845828",NULL);
 glibc_hex = strtod("0x1.f44abd5aa7ca4p+5",NULL);
 printf("glibc (from decimal) = %a\n",glibc_dec);
 printf("glibc (from hex)     = %a\n",glibc_hex);
}
```

Here is the output of the program:

```
gcc (from decimal)   = 0x1.0000000000001p-1
gcc (from hex)       = 0x1.0000000000002p-1
glibc (from decimal) = 0x1.f44abd5aa7ca3p+5
glibc (from hex)     = 0x1.f44abd5aa7ca4p+5
```

The correctly rounded conversion of gcc_dec is 0x1.0000000000002p-1, but gcc gives 0x1.0000000000001p-1. The correctly rounded conversion of glibc_dec is 0x1.f44abd5aa7ca4p+5, but glibc gives 0x1.f44abd5aa7ca3p+5. These incorrect conversions are avoided by assigning the correctly rounded values directly, using hexadecimal floating-point constants.

To get the correctly rounded hexadecimal constant to enter into your program, you either compute it by hand, with the help of an arbitrary-precision decimal/binary converter, or you take the hexadecimal output from a language known to give correct results (like Java or Python).

Bypassing Floating-Point to Decimal Conversion

Another use for hexadecimal floating-point constants is to bypass floating-point to decimal conversion. There are two reasons to do this:

- To save the exact value of a floating-point variable, in text, for future use.
  The traditional way to do this is to convert the floating-point value to a decimal string that, theoretically at least, is guaranteed to "round-trip." For a double, that means printing to 17 significant decimal digits; for a float, that means printing to 9 significant decimal digits. This guarantee is purely mathematical; it's possible a conversion to decimal could be done incorrectly (although I haven't yet found a conversion that fails to round-trip when it meets the minimum digit requirement). Nonetheless, to ensure that a floating-point value is preserved across an intermediate text representation, save it as a hexadecimal constant.

- To view the exact value of a floating-point variable.
  Many programming languages limit the number of decimal digits you can print, preventing you from seeing the exact contents of a floating-point variable. I've written four C functions to address this: fp2bin(), print_double_binsci(), print_raw_double_binary(), and print_raw_double_hex(). Printing a floating-point variable as a hexadecimal constant is yet another way to display its exact value.

Hexadecimal Floating-Point Constants in Other Languages

Java supports hexadecimal floating-point constants much like C, with some slight difference in format. For example, it doesn't print the '+' sign for positive exponents, and it prints powers of two with a radix point followed by a zero (for example, 2^8 prints as 0x1.0p8).

Python supports hexadecimal floating-point constants with float.fromhex() and float.hex(): float.fromhex() creates a floating-point value from a hexadecimal constant, and float.hex() creates a hexadecimal constant from a floating-point value.

Visual C++ supports only *printing* of hexadecimal floating-point constants, and even then, in a restricted way: it will round its output, not displaying all 13 hex digits of the binary fraction. I've found it's easy enough to get around this — just use a format specifier of "%.13a" instead of "%a".

Numbers in computers / Convert to binary, Convert to decimal, Convert to hexadecimal, Decimals, Exponents, Floating-point

11 comments

1. asdf

February 9, 2011 at 2:07 pm

Do you know where a derivation of "For a double, that means printing to 17 significant decimal digits; for a float, that means printing to 9 significant decimal digits. This guarantee is purely mathematical" can be found?

2. Rick Regan

February 9, 2011 at 2:29 pm

@asdf,

This is something I plan to write about someday, but in the meantime…

David W. Matula did the original work in this area (his papers are 40+ years old, and are not all available free on the Web). Also, Goldberg's "What Every Computer Scientist Should Know About Floating-Point Arithmetic" discusses this, as does W. Kahan's "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic."

Good luck.

Update 5/12/15: I finally wrote about this: see my article "Number of Digits Required For Round-Trip Conversions".

3. Michael Pohoreski

April 6, 2011 at 11:58 am

@asdf

> double, that means printing to 17 significant decimal digits;
> float, that means printing to 9 significant decimal digits.

To calculate how many decimal digits are needed to display the full significant bits, you do the calculation:

Displayed Digits = ceiling(num mantissa bits * log(2))+1

The +1 is a safety guard digit.

Given a float has 23 mantissa bits, plus an implicit 1, thus 24 bits * log(2) = 7.24 ~= 8+1 = 9 decimal digits needed.
Similarly, a double has 52 mantissa bits, plus an implicit 1, thus 53 * log(2) = 15.95 ~= 16+1 = 17 decimal digits needed.

Another way to think of it, is given an n-bit integer how many decimal digits are needed to represent it? (Application: Maybe we want to allocate an buffer for it.) We can treat the mantissa as an unsigned integer.

Displayed Digits=Log(2^num_mantissa_bits)

Thus for floats Log(2^24) = Log(16777216) = 7.224719896
Thus for doubles Log(2^54) = Log(9007199254740990) = 15.95458977

Again round up, add 1 digit.

IMHO, as a computer scientist, you should be able to derive these formulas on your own, if you correctly understand how (floating-point) numbers are represented.

It would be best to start with integers and work out these 2 problems:

1. Given a d-digit decimal number, how many n bits are needed to represent it?
2. Given n bits, what is the maximum number of d decimal digits printed?

Cheers

References:
* http://en.wikipedia.org/wiki/IEEE_754-2008#Basic_formats
* How to Print Floating Point Numbers Accurately, Guy L. Steele & Jon L. White (binary to decimal)

4. **Rick Regan**

   April 11, 2011 at 9:22 am

   @Michael,

   Your formulas are correct, but the derivation is more complicated than you show. It is not as simple as treating the mantissa as an integer and counting the number of unique values. It's the relative spacing between binary numbers and decimal numbers in a given exponent range that matters. You'll find this approach detailed in papers by Matula, I. Goldberg, D. Goldberg, and Kahan
   (for example, see Kahan's derivation). Also, on p. 377 in the paper you cite, "How to Print Floating Point Numbers Accurately," they talk about spacing as well (they use the term "graininess").

   (BTW, your line

   "Thus for doubles Log(2^54) = Log(9007199254740990) = 15.95458977"

   should read

   "Thus for doubles Log(2^53) = Log(9007199254740992) = 15.95458977")

5. **hello**

   February 8, 2012 at 11:34 pm

   What algorithm javascript uses to round double? double is quickly obtained at http://www.merlyn.demon.co.uk/js-exact.htm

6. **Rick Regan**

   February 9, 2012 at 8:34 am

   @hello,

Take a look at this article and see if that helps: https://www.exploringbinary.com/inconsistent-rounding-of-printed-floating-point-numbers/ .

7. **hello**

   February 15, 2012 at 7:33 pm

   Not really but that gave me new insight.

8. Pingback: Find and replace floating HEX points with its normal equivalent value in bash

9. **Kiry**

   August 6, 2016 at 7:51 am

   Please give the answer for this question
   Ox41810000 what is representation value for this?
   & decimal value? Is Ox41810000 floating no ?or not

10. **Rick Regan**

    August 6, 2016 at 10:00 am

    @Kiry,

    That’s not a hexadecimal floating-point constant; it looks like a hexadecimal representation of an integer.

11. Pingback: Is floating point math broken? – Acrosoft Solutions

Comments are closed.