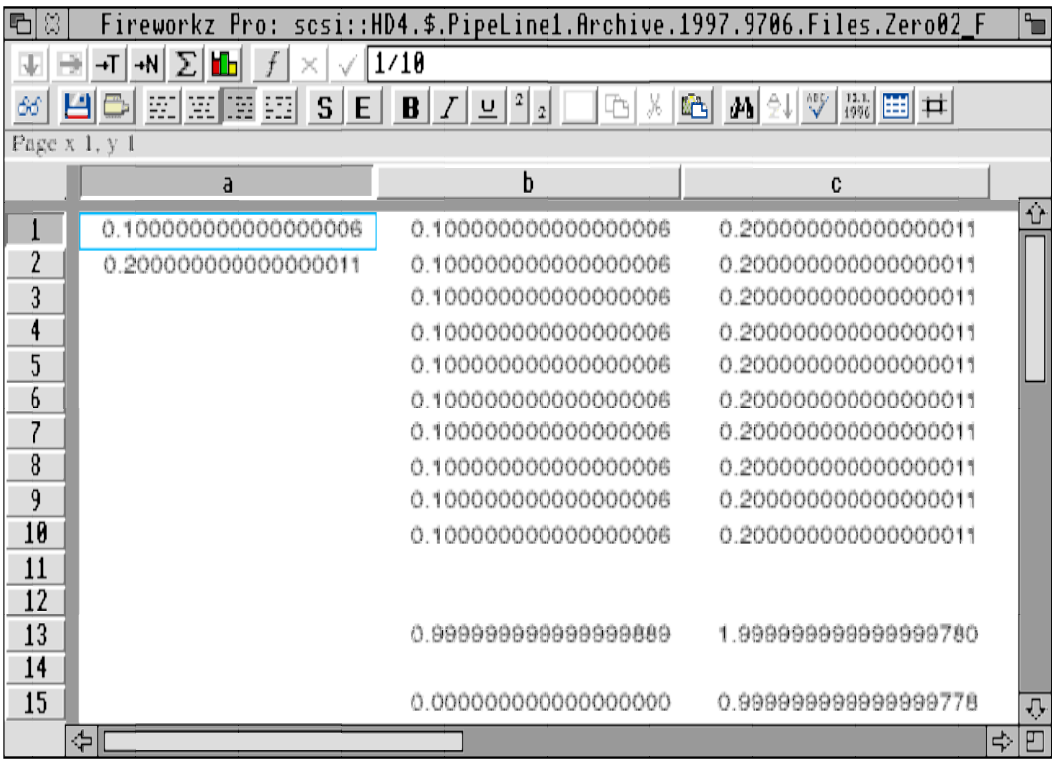
Gerald's Column
by Gerald Fitton

From the correspondence which I've received since last month's Archive was published it would seem that, for the first half, the pace of my article was about right but that I went a bit too quickly over the last third or quarter. To put it another way, you understood everything which I did in "integer arithmetic"; you understood that something which I called "floating point arithmetic" was causing all spreadsheets to return answers to simple sums such as 0.1 + 0.1 which were slightly in error. I have to confess that what I glossed over too quickly was what exactly went wrong, in what ways it went wrong and even when these errors were likely to appear.

I thank you for all your letters on the subject. I was fascinated by a letter from a reader who's son has had a great deal of trouble getting to grips with simple sums at school. My reader showed her son what she called "Gerald's Method" of adding up, subtracting, multiplication and division of numbers of any size without a calculator. I can take no credit for the method she has ascribed to me—place notation and integer arithmetic was discovered over 800 years ago. Anyway, the son is now able to do "experiments with numbers" to the extent that his maths teacher got him, the son, to explain to the remainder of the class this novel method, which didn't need a calculator, to do arithmetic on three digit numbers. The teacher has given the name the "Vertical Method" to what the mother described as "Gerald's Method"!

Introduction

Let me recapitulate by describing a screenshot similar to one which I used last month.



This screenshot is one of a Fireworkz file which you'll find as [Zero02_F] on this month's Archive disc. In last month's column I displayed a similar PipeDream file. Fireworkz can be forced to display one more (albeit incorrect) significant figure than PipeDream. Look in the formula line and you'll see that I've entered (1/10). This is displayed as the number:

0.100 000 000 000 000 006.

In slot a2 I have entered 2*a1; the number displayed is 0.100 000 000 000 000 011.

Is (1/10) > 0.1?

Based on the above I expect that you've concluded that the number in slot a1 is slightly bigger than (1/10), You probably believe that the number stored is about 0.000 000 000 000 000 006 larger than it ought to be.

The ten slots in the range b1 to b10 inclusive contain the formula a1. Slot b13 contains the formula – sum(b1b10) – which is the sum of ten lots of (1/10). Or it is the sum of numbers slightly larger than (1/10)? Well, in the screenshot the sum of ten lots of (1/10) is displayed as: 0.999 999 999 999 999 889, a number which is slightly less than 1.

If you have the monthly disc load the file [Zero02_F] and you'll get another surprise. If you haven't then take my word for it. The number displayed by the newly loaded file in slot b13 is exactly 1.000 000 000 000 000 000. But is that the number stored? Place the cursor in slot a1, click in the formula line where you'll fine the (1/10) and tap <Return>. The number in slot b13 reverts to the slightly smaller number!

The formula in slot b15 is int(b13), the integer part of b13. If the number in b13 is slightly larger than 1 then this formula will return 1; if the number is slightly less than 1 then the formula will return 0. It returns 0. This adds weight to the argument that the number stored in b13 is slightly less than 1. Ten lots of (1/10) add up to slightly less than 1 even though the individual 1/10ths are displayed as slightly larger than 1/10th.

I hope that I've explained the situation rather better this month than I did last month. To summarise:

(a) The number (1/10) can ***not*** be stored exactly in a spreadsheet slot.

(b) The number (1/10) is ***displayed*** as a number slightly larger than (1/10).

(c) The number (1/10) is ***stored*** as a value slightly smaller than (1/10).

Other spreadsheets

In both Eureka and Schema I have found it difficult to display (1/10) as a number other than 0.1. This is because the number of significant figures which can be displayed is less than that of PipeDream and Fireworkz. However, arithmetic calculations such as that of slots b13 and b15 of [Zero01_F] described above yield results which, to the limit of the accuracy of the display, are identical. The conclusion is that all the spreadsheets do their sums in the same way.

Other computers

To the limit of accuracy of the individual displays all the spreadsheets I've tried give results identical to those of slots b13 and b15 of [Zero01_F] which I've described above. The conclusion is that all computer spreadsheets do their sums in the same way.

The floating point emulator

There is an IEEE standard for floating point calculations which is common to all computers which conform to the standard. I do not have any floating point hardware but I have no doubt that all floating point maths chips conform to the IEEE standard, whether they are destined to be plugged into a windows machine or an Acorn.

Acorn's floating point emulator uses the RISC central processor to emulate commands which would be executed by a dedicated floating point maths chip if one were fitted. The Programmer's Reference Manual gives quite a bit of detail about the way numbers can be stored in floating point registers. There are four different storage formats. Let me quote what I think is the most relevant remark even though it applies to only one of the four formats: "A double precision number has a maximum (decimal) exponent of 340 and 17 (decimal) digits of significance."

A number such as 0.100 000 000 000 000 006 contains 18 decimal places. Only 17 of these are significant; the 18th, the number 6, is not significant. By experiment I have discovered that the Acorn spreadsheets use a maximum positive exponent of not 340 but 308; indeed the largest number you can enter into or return from PipeDream, Fireworkz, Schema and Eureka is just less than 2^1024. This implies that 10 binary bits are used to store the exponent. This largest number, 2^1024, is approximately 1.8+e308. The "+e308" means move the decimal point from between the 1 and 8 by 308 places to the right.

Although I have only circumstantial evidence to support my next two remarks I think that they are the basis of a good working hypothesis. My first remark is that the reason why all Acorn spreadsheets display not only the same level of inaccuracy but also return the same inaccurate number in slots such as b13 (etc) is that they all use the same floating point emulator or hardware. My second remark is that the reason why all windows spreadsheets exhibit the same features as the Acorn spreadsheets is because the floating point arithmetic is executed with hardware or software which conforms to the same IEEE standard as does the Acorn floating point software/hardware.

Storing (1/10) in binary

It is the fact that (1/10), along with most fractions, can not be stored exactly in floating point binary which causes the error to arise. Let me explain how in more detail than I did last month.

Just as you can work out the decimal value of (1/17) by long division using the method I demonstrated last month, so you can use long division in binary to work out the binary representation of (1/10).

First a few simpler sums.

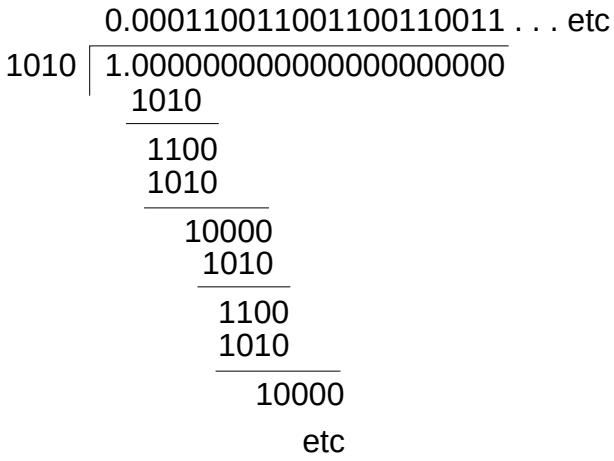
In two digit binary 01 + 01 = 10. One and one make zero and one to carry.

In two digit binary 10 − 01 = 01. One from zero won't go. Borrow one from the next significant column and get the result shown.

Once you have grasped the principles of place notation in binary then you will find long multiplication is as straight forward as addition.

The graphic below is a draw file showing the workings of 1 divided by decimal 10 (binary 1010) using long division in binary. I don't expect you to actually do the sum in binary; just accept my word for it. The result is 0.00011 0011 0011 0011 . . . etc, a series which repeats every four digits but which goes on for ever.

An exact value for (1/10) can not be stored in binary because the series never terminates.

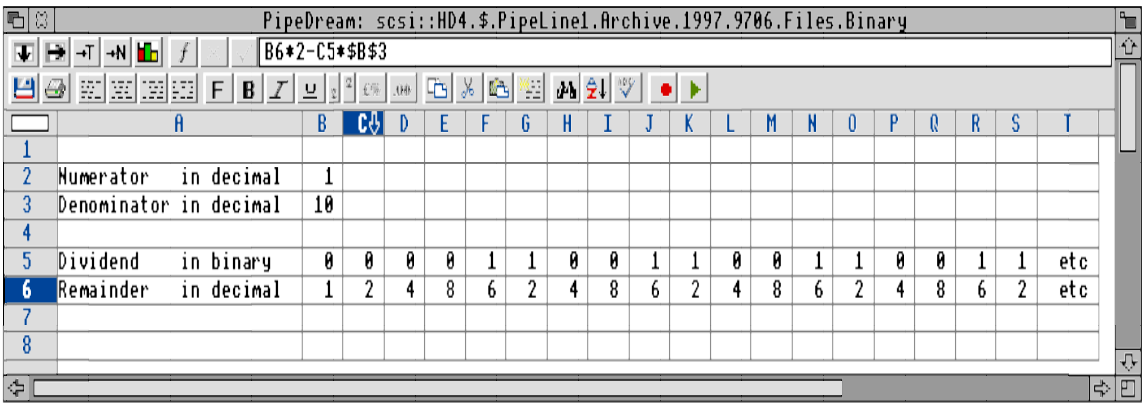


Decimal to binary

Although you may be interested in teaching yourself how to do long division in binary I suggest that you might find it easier to use a spreadsheet to do the sums.

The PipeDream spreadsheet shown in the screenshot (I hope that Paul can make it legible) has been used to divide 1 by 10 and return the answer in binary.

On the monthly disc this spreadsheet is called [Binary]. The slot C5 contains the formula int(B6*2/$B$3). Slot C6 contains (B6*2−C5*$B$3). Slots C5 and C6 can be replicated as far to the right as you wish. B5 contains int(B2/$B$3) and B6 contains ($B$2−B5*$B$3).



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1																				
2	Numerator in decimal	1																		
3	Denominator in decimal	10																		
4																				
5	Dividend in binary	0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	etc
6	Remainder in decimal	1	2	4	8	6	2	4	8	6	2	4	8	6	2	4	8	6	2	etc
7																				
8																				

You can enter any pair of numbers as numerator and denominator into slots B2 and B3 of this spreadsheet provided that they make a proper fraction (B2 must be smaller than B3). I have entered 1 as the numerator and 10 as the denominator. The binary result is displayed in row 5 as 0.00011 0011 0011 etc.

The value of (1/3) returned by this spreadsheet is 0.01 01 01 01 etc.

The value of (1/7) returned by this spreadsheet is 0.001 001 001 001 etc.

The value of (1/9) is 0.000111 000111 000111 etc.

Numbers such as (1/16) terminate; (1/16) returns 0.0001 00000000 etc.

Making an analogy with the place notation of decimal with its Units, Tens, Hundreds and Thousands columns these results can be interpreted as the first column after the point is the ‘half’ column, the second is the ‘quarter’ column, the next the ‘eighth’ column, etc. You will see that the number (1/16) returns a 1 in the ‘sixteenth’ column and zeros elsewhere.

The binary representation of the number (1/3), 0.01 01 01 01 01 etc, shows that it is the sum of the infinite series of fractions:

(1/4) + (1/16) + (1/64) + (1/256) + etc

where each denominator is four times the previous denominator.

Approximations

Those numbers for which the binary series does not terminate can not be stored with complete accuracy in a computer which stores numbers in floating point binary. I am not sure exactly which of the four available floating point formats are used by Acorn spreadsheets but, by looking at the accuracy and precision I can have a reasonable guess.

The four formats are:

(a) 32 bit single precision to the IEEE single precision standard.

(b) 64 bit double precision to the IEEE double precision standard.

(c) 80 bit double extended precision which I think is not one of the IEEE standards.

(d) 80 bit packed binary coded decimal which I think is not one of the IEEE standards.

Let us ignore (a), single precision. I don't know of any spreadsheet which uses it.

Double precision, (b), uses 52 bits for the mantissa and 11 bits for the exponent. Now 2^52 will not give 17 significant decimal digits so I think that Acorn spreadsheets don't use this format except when porting to windows type spreadsheets.

Acorn's double extended precision, (c), uses 64 bits for the mantissa and 15 bits for the exponent. This format will give more than 17 significant decimal digits but it is the format recommended by Acorn for all internal floating point calculations.

It is the packed binary coded decimal format, (d), which gives 17 significant decimal digits.

Encoding numbers

When you have finished doing your sums in double extended precision, the (c) format, you can use a simple Acorn floating point instruction to store the number in packed binary coded decimal, the (d) format. You can also convert this (d) format into the (c) format just as easily.

It is much easier to convert a (text) string of digits into the (d) format than to any other format. It is my guess that the writers of the spreadsheet packages accept the string of digits which is in the formula line, convert this text string to packed binary coded decimal and then leave the floating point emulator or hardware to convert this to the (c) format for doing sums.

When returning numbers to be displayed in the spreadsheet it is much easier to use the floating point emulator to convert then stored number from the (c) format to the (d) format and then convert from the (d) format to the text string which is displayed in the spreadsheet slot.

I have more evidence for this theory than the 17 significant digits to which the spreadsheets appear to work. It was in 1988 that I decided that I needed to do some sums to a greater precision than BASIC would allow. I decided to use the BASIC assembler rather than the C language and that meant that I had to write an extension to Acorn's BASIC assembler which would allow me to compile the floating point emulator mnemonics. I did this. To my surprise, although I designed my compiler to work under the original Arthur operating system, RISC OS 1, when I tried it just now it still works!

After studying the problem of data entry and retrieval I came to the conclusion that the simplest thing to do would be to convert strings of digits into the (d) format and then leave it to the floating point emulator to convert to the (c) format. I don't think my line of reasoning is brilliant. I think that it is so obvious that anyone faced with the same problem, for example the designers of Acorn spreadsheet packages, would do the same thing.

Floating point arithmetic

I have been asked to give some examples, similar to my integer arithmetic examples, showing how floating point arithmetic works.

I'll demonstrate the method with a couple of floating point sums in decimal. I shall assume that we have a floating point system which holds only four significant decimal digits so that I don't use up too much space with meaningless digits.

The first sum we're going to do is to add the two decimal numbers 1234 and 123.4. In decimal floating point these numbers will be stored as a decimal exponent and a decimal mantissa. The mantissa of both these numbers is 1234. The exponent of the first number is 3 and that of the second number 2. If you like you can consider the first number as 1.234 * 10 ^ 3; start with 1.234 and move the decimal point three places to the right to get the number 1234.

Integer	Floating
1234	1234
123.4	0123
1357.4	1357

The first of the two simple draw files above shows the calculation as you might do it using integer arithmetic. The second shows how the calculation might look using four digit floating point arithmetic.

The way in which you do the calculation in floating point arithmetic is as follows. First identify that the first number has the larger of the two exponents namely 3. Take the 2, the exponent of the second number, away from the 3 of the first number to get 1. Shift the mantissa of the second number this 1 place to the right so that the digit 4 in the mantissa of the second number gets shifted into oblivion. The two mantissa are now 1234 and 0123. Add these two numbers using integer arithmetic to get 1357. This is the mantissa of the answer. The exponent of the answer is 3, the larger of the original exponents.

The second sum is shown in the draw files below:

Integer	Floating
12.34	1234
0.1234	0012
12.4634	1246

The first number is 12.34 and the second 0.1234. The exponents are + 1 and − 1 respectively. The difference between the exponents is 2 so shift the mantissa of the second number two places to the right. The 3 and 4 disappear leaving the mantissa as 0012. Integer arithmetic applied to the two mantissa yields 1246 as the mantissa of the answer. The exponent of the answer is that of the larger number, 2. In four digit decimal, the

answer is 12.46.

Multiplication is executed as multiple addition with appropriate shifts of the mantissa of the multiplier. I am not going to give examples of this nor of division.

I would like to add that when it comes to doing sums in binary with computer storage of the numbers being of fixed length floating point arithmetic is much more economical of processing power than is integer arithmetic. Of course that's why computers use floating point arithmetic and not integer arithmetic.

I have cheated

But not too much. When any floating point operation is executed you have a choice of four methods of rounding with 'round to nearest' being the default. I think I made a mistake last month by suggesting that numbers were truncated (chopped off). Since then I have done some experiments using my BASIC floating point compiler and looking at the contents of the floating point registers in the four formats (a), (b), (c) and (d).

I have concluded that it is most likely that in PipeDream and Fireworkz the number stored for calculation purposes is held in double extended precision format, (c) whilst the number displayed is derived from the packed binary coded decimal version of the number, (d). It is the 'round to nearest' feature which causes the number to be different in the two formats.

In (c) format the number (1/10) is slightly smaller than (1/10) whereas in (d) format it is slightly larger. Although I haven't proved to you that the conversion from (c) to (d) and from (d) to (c) does change the number slightly I have a lot of experimental evidence that this is the case.

I believe that it is this 'round to nearest', used by all Acorn spreadsheets, which causes the number (1/10) when displayed to be larger than the number stored.

Finally

My address is that of Abacus Training (see back inside cover). If you have any comments or contributions to this discussion then please write to me.

As I write I am still not on e-mail but, thanks to Paul's persuasiveness, I shall be shortly. Bear with me for another month or so and send me discs and pieces of paper by (what I believe is called) 'snail mail'.