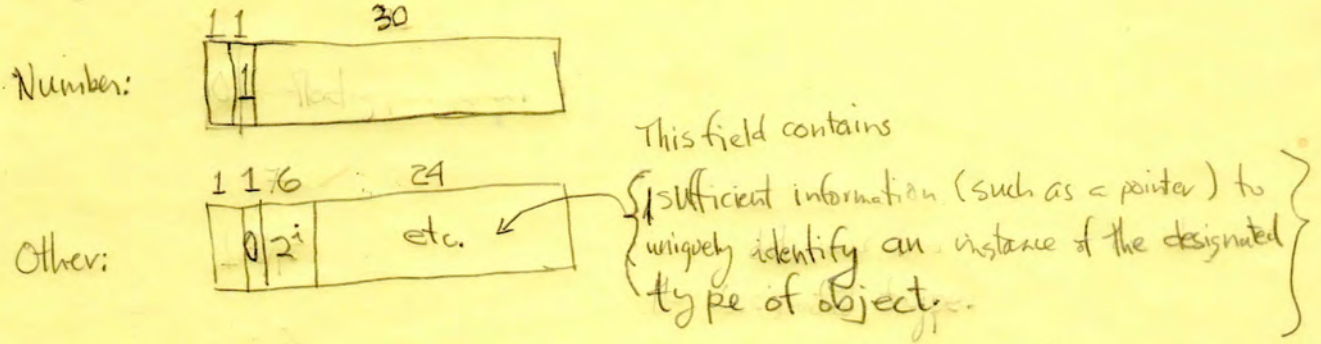
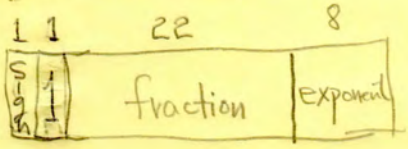
Meta APL Floating Point Arithmetic

Internal to the Meta4 APL system every object is represented by a 32-bit descriptor. There are two basic formats for descriptors, distinguished by the value of bit 1 of the descriptor:



The number of bits available to represent a (floating point) number is limited, so it was thought worthwhile to keep all numbers normalized*; indeed the leading one distinguishes floating numbers from all other values! Here then is the representation of a number:



If the number being represented is $X = s \times C \times 2^e$, then:

$$e = \text{exponent} - 128;$$

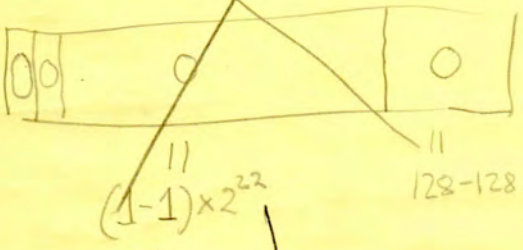
$$C = 1 + \frac{\text{fraction}}{2^{22}};$$

and $$s = \begin{cases} +1 & \text{if sign} = 0 \\ -1 & \text{if sign} = 1 \end{cases}.$$

---

* Obviously the number 0 is a special case; it will be discussed later.

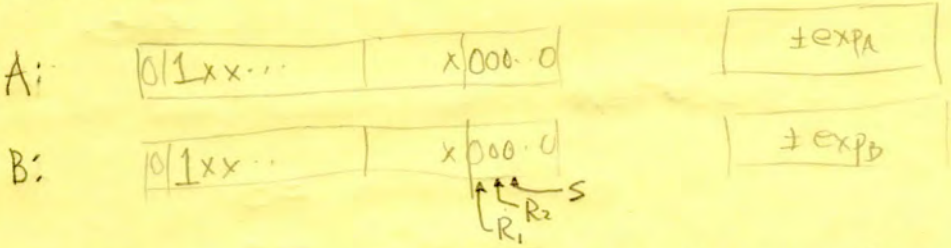
In this scheme, 0 is represented by $x = \pm 1 \times 2^{-128}$, because this is the one number whose reciprocal, 1×2^{128}, is not representable anyway. Zero is always represented with its "sign" = 0:



(Zero ≡ 

)

The operands of a floating point arithmetic operation are "unpacked" into a more convenient form. Basically this consists of ~~subtracting the bias from~~ moving the exponent ~~and moving it~~ to another register, and replacing the sign bit position with ~~the leading one of the fraction field.~~ a zero. This zero in bit position 0 acts as a positive sign to facilitate the internal arithmetic.



In order to do correct rounding one needs a very long accumulator or alternatively two rounding bits and a "sticky bit". These three bits, named respectively R_1, R_2, and S, are the three bits just to the right of the fraction. The rounding bits R_1 and R_2 behave like the other bits holding the fraction, but the sticky bit always holds the logical "OR" of all bits which have passed through it.

The analysis of floating point addition should be broken into two cases: addition of two quantities with the same sign, and addition of quantities with differeid signs, i.e. subtraction of magnitudes. There is some preparation common to the two cases, however.

Before the actual addition (or subtraction) can take place, the radix points of the operands must be aligned. This is accomplished by right-shifting the operand with the smaller exponent; the number of places to shift is given by the difference of the exponents. In this shift the sticky bit must be maintained correctly.

A zero operand should be detected and treated as a special case; due to the peculiar representation of zero, incorrect answers would result if it were actually used in arithmetic.

Addition of magnitudes

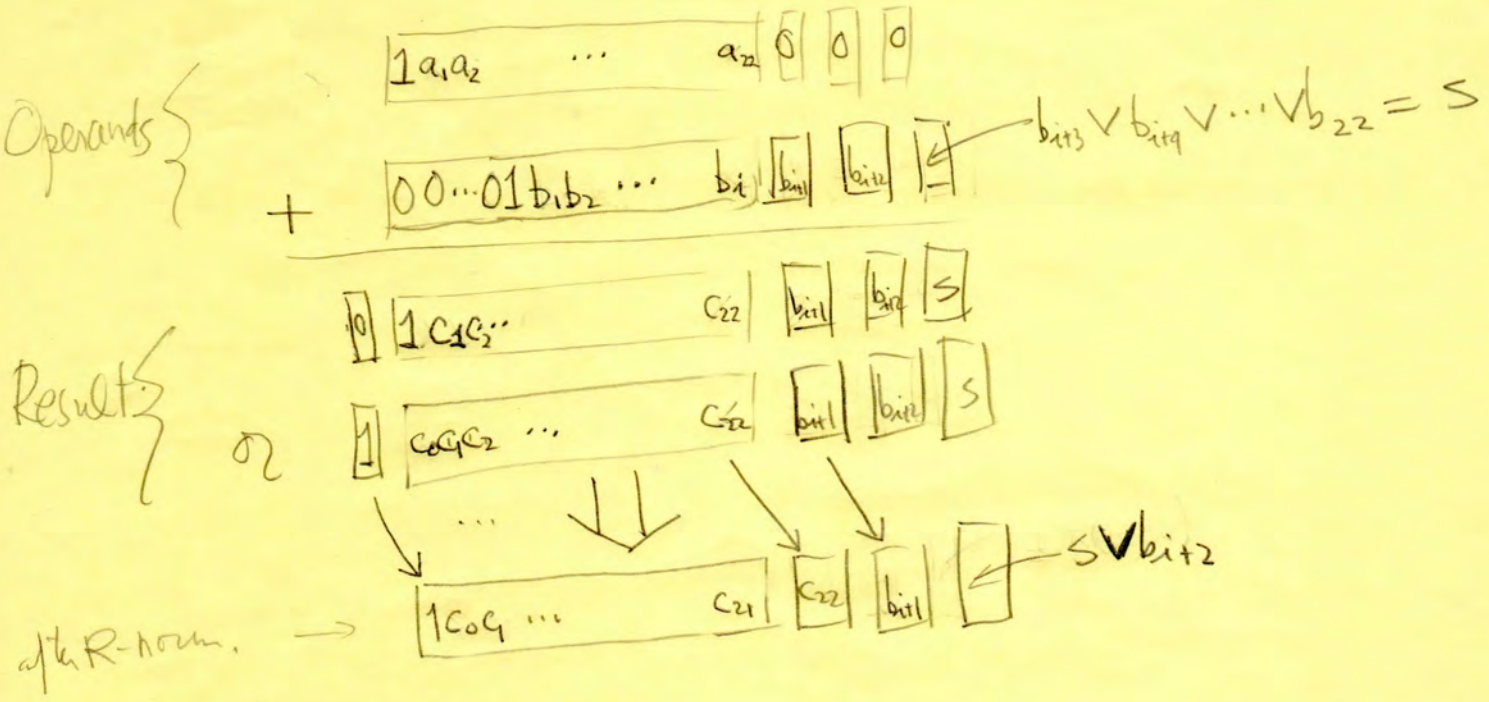
The two 26 bit quantities (23 fraction bits, R_1, R_2, and S) are added together. If a carry from the leftmost position occurs, the sum is right shifted* by 1 place; the exponent of the result, which is normally the max of the exponents of the operands, is increased by 1. Other than this possible single right shift, no normalization is necessary; the result can be rounded and repacked (see below).

Exponent overflow is possible, when adding the two coefficients or when rounding the sum, is possible; if it occurs, the (biased) exponent will equal 400B.

---

* Correctly updating the sticky bit, as usual.

Illustration: Magnitude Add



Subtraction of Magnitudes

The smaller magnitude should be subtracted from the larger (round and sticky bits included). If one of the operands was right shifted to align radix points, then it must have the smaller magnitude; if both exponents were identical we must compare magnitudes:

(i) if they are equal, result is zero -- quit;

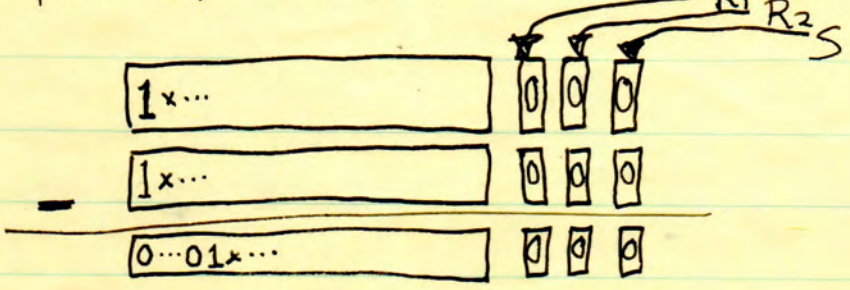
(ii) otherwise, go ahead and subtract the smaller from the larger.

To see that the two round bits are sufficient, consider the four following cases:

Magnitude

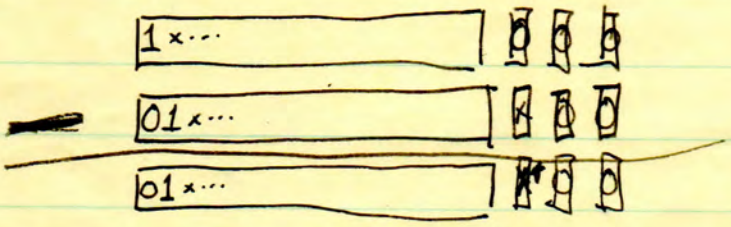
Floating Point Subtraction

Case 1: The operands require no radix point alignment.



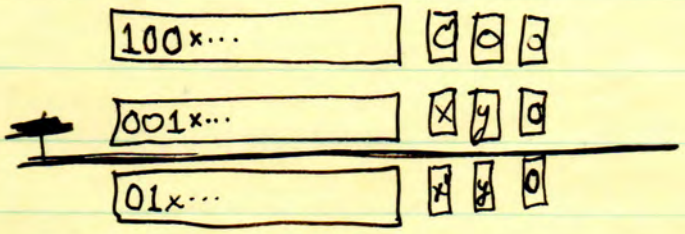
The result will require one or more places of left-shift to be normalized; zeros should be supplied from the right. (This is the only one of the four cases in which a zero result is possible.)

Case 2: The operands require exactly 1 place radix point alignment shift.



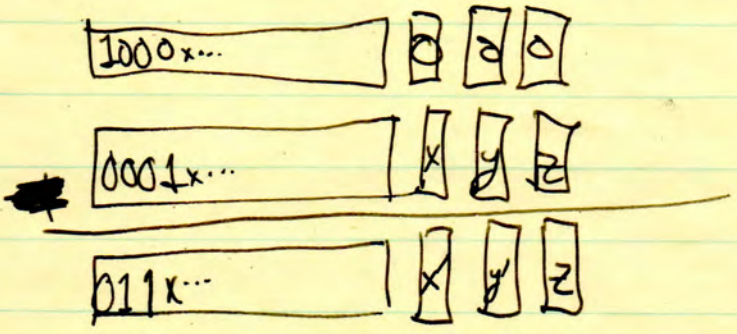
The result will require zero or more places of normalization; R_1 should participate in this shift (followed by zeros).

Case 3: Radix point alignment requires two place shift.



Normalization by zero or 1 place is required; in the latter case R_1 shifts into the fraction and R_2 shifts into R_1.

Case 4: Radix point alignment requires a shift of 3 or more places.



Normalization by zero or one place is required; in the latter case $R_1 \Rightarrow$ fraction low order bit, $R_2 \Rightarrow R_1$.

~~Case N: Radix point alignment requires at least (N-1) shifts, where N is width of fraction~~

Rounding

After a magnitude add or subtract and subsequent normalization have been performed, the result must be rounded to the correct word size. Rounding proceeds as follows:

R_1	R_2 VS	Action
0	0	Round down, or truncate
0	1	Round down, or truncate
1	0	Round to nearest even (to prevent "drift")
1	1	Round up

Note that two round bits are really only needed for magnitude subtractions, but the treatment described above gives correct results for magnitude additions as well.

Rounding up may cause "fraction overflow"; in this case add 1 to the exponent of the result, and set the fraction to the value $100\cdots0$.

After rounding, check for possible exponent underflow, then "pack" the result. (Exponent underflow results from normalizing certain results of magnitude subtraction; it shouldn't be checked for until after rounding, however.)

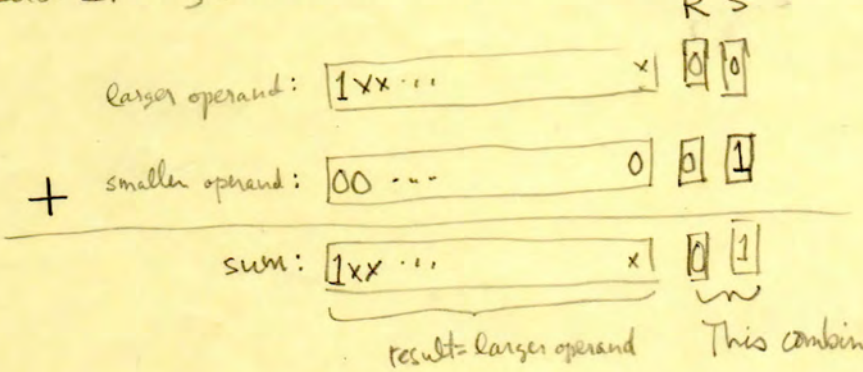
A June 72

Note on Floating-Point Addition/Subtraction Concerning "Neglisibly Small Operands"

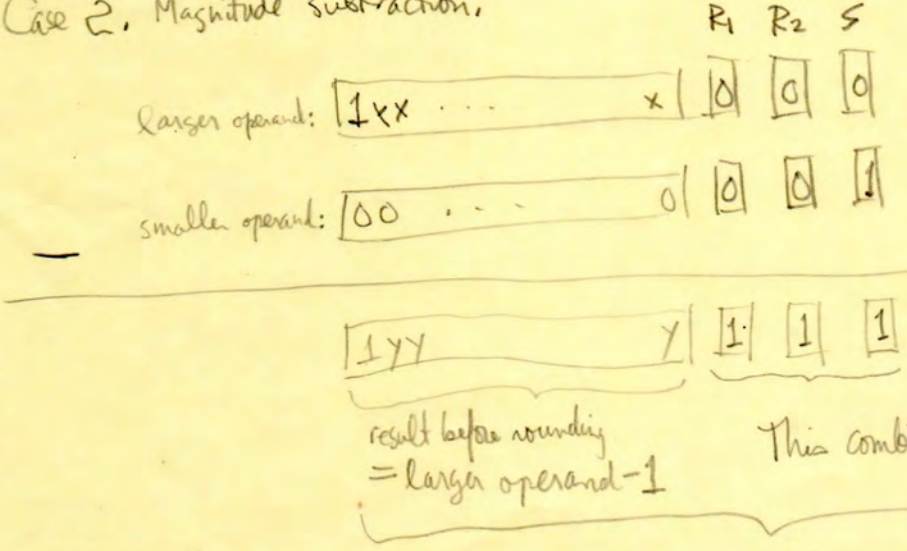
Before addition or subtraction takes place the radix points of the operands must be aligned. It may be that alignment will result in one operand being shifted 'clear off the end"; in this case we can show that the smaller operand can be treated as zero -- i.e. the result of the addition/subtraction is simply the larger operand.

Suppose the width of a fraction is W bits, and k round bits are carried through the calculation (k=1 for addition, 2 for subtraction). Then an operand whose exponent is at least w+k smaller than the other operand may be treated as zero.

Case 1. Magnitude addition.



Case 2. Magnitude subtraction.



Meta APL

"Funnies Due to the choice of representation for 0.0"

1. An operand of 0.0 should be detected and treated specially;

$X+0 = 0+X = X$, all X (including $X=0$)

$X \cdot 0 = 0 \cdot X = 0$, all X (including $X=0$)

$X \div 0$ is an error, for $X \neq 0$;

$0 \div 0 = 1$;

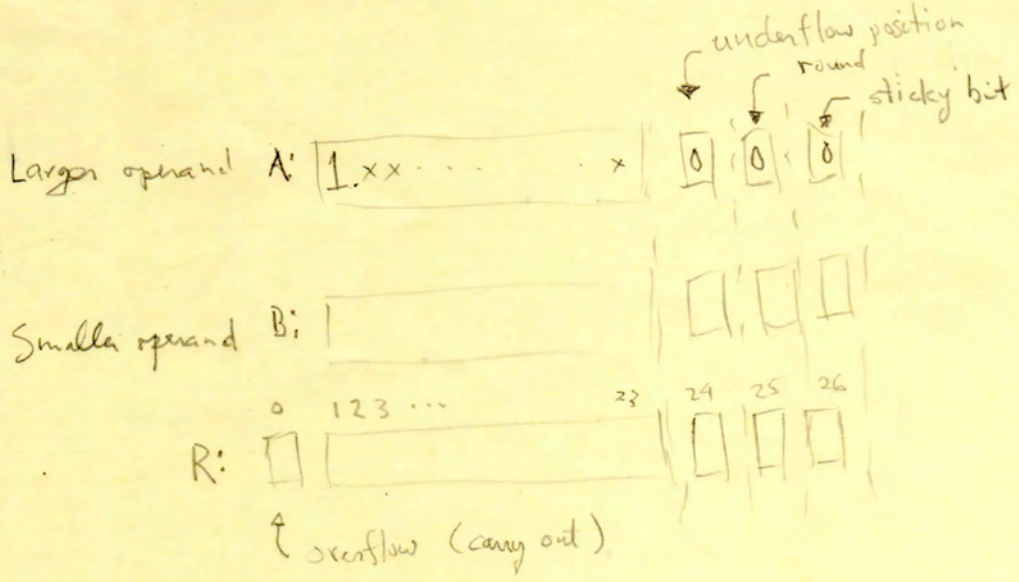
$0 \div X = 0$, all $X \neq 0$

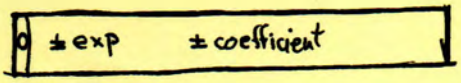
[Note: $X - Y = X + (-Y)$]

2. A result of 0,0 should be detected and put in the special representation: $|X-1Y|$

~~3. A result of 1×2^{-128}, which is the number chosen to represent 0, should cause an exponent underflow;~~

~~$\pm(X+Y)$, $X \div Y$, $X \times Y$ may yield such a result~~


underflow position
round
sticky bit
Larger operand A:
1.xx . . . x
0
0
0
Smaller operand B:
R:
0
1 2 3 . . .
23
24
25
26
overflow (carry out)



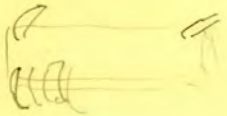
Floating point predisplacement shift is usually $\ll 8$.

Thirty-two bit format is probably big enough for floating point applications. But for integer applications, at least 32 bits of coefficient should be provided.

"Solutions"

1. several different representations (a la APL\360)
2. go to single, long floating-point format – say 64 bits.
3. Add a switchable "integer overflow trap" mode, and represent integers as unnormalized fl. point numbers with neutral exponent.
4. Force the user to test for non-integer results of "integer arithmetic" – again represent integers as unnormalized numbers with neutral exponent.

Winograd & Rabin on fast multiplication.

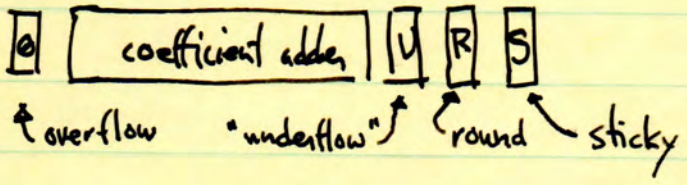


Appendix 4: Notes on floating-point (and integer?) arithmetic

1. Algorithms guarantee the result of f.p. +, −, ×, ÷ is the (ideal) correct result, rounded to the nearest representable number.

2. If the unrounded result lies exactly half way between two representable numbers, then the one ending in "0" is chosen.

3. It might seem that a very long register is needed to implement these algorithms, but actually only a few bits are needed in addition to an adder for the coefficients:



4. Since coefficients are so short (23 bits), a special form for integers subject to exact integer operations is provided.

Appendix 4:

The algorithms for floating-point sum, difference, product, and quotient were designed to make optimal use of the rather small (32 bit) word size. [illegible]. By using only normalized representations (see section 2.1 for a description of the floating-point number format), a large set of bit patterns becomes available to represent other types of values. ~~In addition, every normalized floating-point number has a representable reciprocal.~~ A unique representation for every floating-point number is also assured by disallowing unnormalized numbers.

Appendix 4:

Computer floating-point arithmetic is by its very nature approximate since only a few significant figures (and a scalar factor) are used to represent a number. It is nevertheless important that algorithms for the basic operations on these numbers (addition, subtraction, etc.) provide the best possible result: that floating-point number nearest the ideal, mathematical result (which in general is not a representable floating point number). When correctly rounded floating-point arithmetic is provided, the task of programming efficient numerical algorithms is made far easier.

One might imagine that implementing correctly rounded arithmetic requires a very long adder, but a technique (made known to the author by Professor William Kahan) exists for a cheap implementation. An adder the length of the coefficient of a floating-point number and a few extra bits are all the facilities that are required. Let us consider addition, the basic operation.

Appendix 4: Notes on the Arithmetic Algorithms

Since the word size on the APL processor is limited (32 bits), some programs may be hindered by the lack of precision. Several features of the processor should help this situation somewhat. First of all, a special representation for integers, and operators performing exact arithmetic on these integers, is provided. The integer representation carries no more precision than the coefficient of a floating-point number (so converting from integer to floating-point can never overflow). The only reason for integer arithmetic is to provide a warning (in the form of a trap) when a result overflows the maximum integer magnitude.

The other feature of the arithmetic is that the "best possible" use of the limited word size is made by the floating point arithmetic. That is, a floating-point result is always correctly rounded and accurate to the last place.

Formally