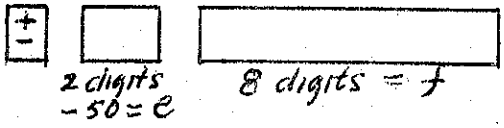
We examine three machines to see how they represent numbers and do arithmetic:

THE IBM 650 number representation is as follows: there is a sign bit containing either a + or a - followed by a string of ten decimal digits.



Such a string can represent either an instruction or a number, and its decoding depends upon its intended use. If a floating point number, it is decoded as follows: from the two leading digits subtract 50 to obtain a number called e. 10^e is prefaced by the sign and followed by a point and the last 8 digits labeled f. The number is interpreted as $\pm 10^e x.f$, where $0 \leq .f \leq .99999999$

Example: +38 64213907 is interpreted as $= +10^{38-50} \times .64213907$

The following technicalities arise: the representation of zero is not unique, since any number whose last 8 digits are zeros will be zero. ∴ define zero to have all 10 digits = 0. Then e = -50. But still you have the problem of having both a +0 and a -0. This occurs in all machines which have the property that if x is represented then so is -x. Similarly, to insure uniqueness of non-zero numbers, the first digit of the last 8 is required to lie between land 9 inclusive. For such numbers, called normalized numbers, we have $.10000000 \leq .f \leq .99999999$.

Let us now look at some examples of how arithmetic is done on the IBM 650. Suppose we want to add two numbers, say

$$\begin{array}{lll} +72\ 6413 & = & +10^{22} \times .6413 \\ +70\ 4096 & = & +10^{20} \times .4096 \end{array}$$

where we use four digits for brevity. We shift the decimal point to equalize the exponents and add. Any digits in the machine which fall out of a register are lost.

+72 6413	$+10^{22} \times .6413$	the error here is not too bad
+72 00409$\not{0}\not{0}$	$+10^{22} \times .004096$	
+72 6453	$+10^{22} \times .645396$	

(2)

Now suppose we try the subtraction

+51 1000

$-$ +50 9999

Rewriting and subtracting we have

+51 1000	1.000
-51 0999$\not{0}$	$-$ .9999
+51 0001	.0001

Finally the answer is normalized to +48 1000, which corresponds to an answer of 10^{-3} whereas we can see from the above calculation on the right that the answer is 10^{-4}. We are off by an order of magnitude! ∴ the $\not{4}$ rule that the fortran statement $X = A \bar{+} B$ is represented in the machine by $x = (a \bar{+} b)(1+\varepsilon)$ is not valid unless we allow values of ε up to 9, whereas for 4 digit arithmetic we would like and expect $|\varepsilon| \leq 10^{-3}$. Thus the IBM 650 is not the kind of machine we like to do arithmetic on.
How many floating point numbers on the 650? $2 \times 10^2 \times 10^8 (1 - \frac{1}{10}) + 1$

Numbers on the IBM 360: Again we have a sign bit followed this time by 7 bits labeled c (characteristic) and 6 (or 14 or 28) hexadecimal digits labeled f. [±] [7 BITS] (c) [6 HEX DIGITS] (f)

To decode the number, write the sign followed by $16^e = 16^{c-64_{10}}$, a hexadecimal point, and the digits f. The number is then interpreted as $\pm 16^e \times .f$

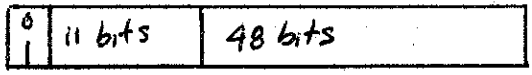
Why subtract something from the characteristic to get e as we have done? Not only to allow negative exponents, but also to make possible comparisons of normalized numbers without decoding them. On the 360 for normalized numbers $.100... \leq f \leq .FFF...$ where F is the hex digit for 15. The problem of -0 occurs infrequently on the 360, and the rule $X=A-B \longrightarrow x=(a-b)(i+5)$ is valid on this machine. How many floating point numbers are possible on the 360? The number is $2x2^7x16^6(1-\frac{1}{16})+1$ for the 6 hex digit numbers. The first 2 is due to sign; there are 2^7 possible characteristics; there are 16^6 possible arrangements for f but $\frac{1}{16}$ of them have leading zeros; finally we add 1 for zero.

CDC 6000 & 7000 SERIES: 1's complement binary integers. We are given 60 binary bits the first of which determines the sign of the integer. If the leading digit is a 0,

1) write + SIGN
2) follow the ~~#~~ by the magnitude of the remaining bits

If the leading digit is 1, write '-', complement the whole word, and go to (2) above.

For floating point numbers the first 11 bits after the sign bit serves as the characteristic:



Decoding proceeds as follows: 1) If the sign bit is 0 write '+'; otherwise write '-' and complement the whole word. (2) Complement the first bit of the characteristic; call the 11 bit result, interpreted as a 1's complement binary integer, e. (3) If e=-0 write '~~0~~' meaning indefinite and stop. If $e = 2^{10}-1$, write ∞ and stop. (4) Interpret the last 48 bits as a binary integer, not a 1's complement binary integer, $I \geq 0$. $\therefore$ we have $\pm 2^e \times I$.

Exdeptions: If $e = \pm(2^{10}-1)$, then the value is treated like zero for multiply/divide operations. The normal zero has $e = -(2^{10}-1)$ and $I=0$. Normalized non-zero numbers have $2^{47} \leq I \leq 2^{48}-1$ to guarantee uniqueness of representation.

How many numbers? for add/subtract/~~[illegible]~~ we have $2x(2^{11}-2)x2^{47}+1$
for mult/div we have $2x(2^{11}-3)x2^{47}+1$

As for the IBM 650, the rule of correspondence between the fortran statement X=A-B and the internal representation $x = (a-b)(1+\varepsilon)$ is invalid, but this time for a different reason. Consider the example we had before in 4 digit arithmetic. After the right shift to equalize the exponents, and the subtraction, the registers appear as follows:

A	1000	000 ← these zeros aren't really there in machine	
B	0999	9	
X	0000	1	

The subtraction is performed properly to obtain the contents of double register X. Notice that the register B has been extended to accomodate the shifted digits, and the subtraction has been performed as if the number in register A were followed by zeros. Unfortunately when we now ask for the value of X, the right half of double register X is thrown away, and the contents of the left half are normalized, in this case to zero. The new compiler which will be available for us will perform the normalization on the whole double register X, thus preserving in many cases important digits.

Why does the old compiler on the CDC 6400 give 1.000 - .9999 = 0 ? The machine was designed with the following ideas in mind. Suppose you write the fortran statement A = B-C. Since the values b and c stored in cells B and C are not normally

known precisely, we should not complain if they are altered by less than a unit in their last places, so we cannot complain if the computed value in A turns out to be $a = b(1+\beta) - c(1+\gamma)$ for some very tiny relative rounding errors β and γ. For example

if b=1.000 take $\beta = -.0001$
c=.9999 take $\gamma = 0$

Then a = 1.000(1-.0001) - .9999(1) = 0 exactly! Thus we can say "1.000 - .9999 = 0 to 4 significant figures." If we adopt the foregoing view, which seems to be more appropriate for used car salesmen and North Beach touts than for scientists, we deserve to get zero for an answer.