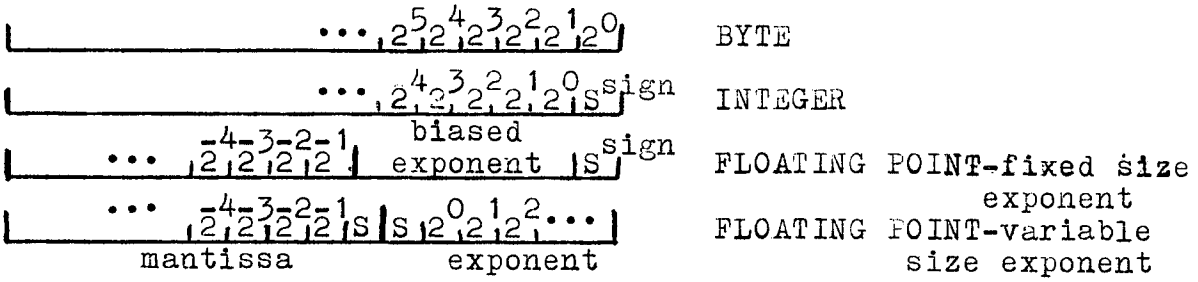
An Optimal Floating Point Format

There is an optimal format for signed numbers and floating point numbers. First, there is the matter of in what sense optimal. Primary memory has always been limited and expensive. Any technique which minimizes storage requirements is desirable. The most immeadiate method is to use only as many bits per number as required. Thus signed numbers should not be used when unsigned numbers will do, as the sign takes an extra bit. Likewise, floating point numbers should not be used when the location of the decimal point is known as the exponint takes extra bits. Also, only as much percision as is needed should be used. This requires variable percision unsigned numbers(bytes), signed numbers(integers), and floating point numbers.

The memory read and write operations occur more often in the critical path of a program than type conversions, hence it is optimum to minimize the effort in loading and storing variable precision numbers at the expense of the effort for type conversion. The least expensive loading method for variable precision is shift and mask. Extending the sign of this result would of course increase costs. The least expensive storing method is dependent on the storage device. However, the arithmetic organ can be assumed to be fixed length. Thus stores involue reducing the number of bits in a number and truncation is the least expensive method of doing this. Finally, if the very same load and store mechanisms can be used on all three of the data types, there can be no less expensive techniques. The intent is that the instruction specify the number of bits in the number for all loads and stores. The type need

not be specified.

There is a class of byte, integer, and floating point formats which have such properties. The byte is represented in ordinary binary radix. The integer is a byte with the sign bit next to the 2^0 bit. The floating point format is a bit reversed(flipped) byte for the mantissa followed by the exponent.



It can be seen that truncation causes no loss of information if the number will fit in that many bits. Further, zero extension causes no change in information. Thus, numbers are pulled out of memory and extended to the size of the arithmetic organ and truncated to fit back into memory. Conversion between the different types is not difficult. The novel feature being the requirement for a "flip circut".

There is the matter of whether type conversion should be explicit or implicit. Implicit type conversion would require specifying the type at load and store time and maintaining the type information with the number in the arithmetic organ.

James C. Brakefield
1235 Dartmouth
Madison, Wisconsin 53705