CS 420 Exam Question

Gavin Gray

May 29, 2022

Q: *Assume that in order to add floating-point values to L_3, we decided to increase the size of basic values to 64 bits, and switch to NaN-tagging. Describe how L_3 values (both the existing ones as well as 64-bits floating-point values) would be represented in such a scheme. For simplicity, assume that primitives working on floating-point values are distinct from those working on integers (e.g. integer addition would be `@+` like now, but floating-point addition would be `@float-+`).*

I will be following the specification of the IEEE 754-2019 standard [1]. As seen in figure 1, the contents of a single double precision floating-point number has a 1-bit sign (S), an 11-bit biased exponent (E), and a 52-bit trailing significand (T) [1, §3.4]. A NaN value is indicated by setting all bits of E to one and can represent a *quiet NaN* or *signaling NaN*. For the purposes of NaN tagging in L_3 we will always assume quiet NaNs and thus the highest bit of T needs to be set to 1.

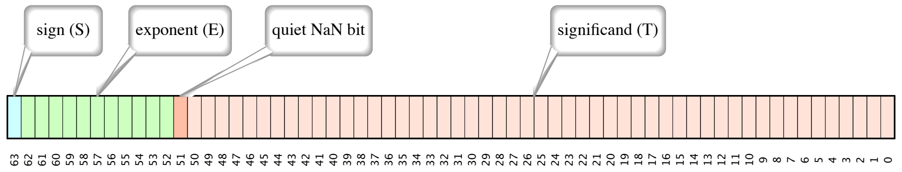

Figure 1: An example of the layout of a 64-bit floating-point.

In L_3 we need to represent 6 values: double precision floating-point numbers, integers, pointers, characters, booleans, and the unit value. We know that floats will take up the full 64 bit word and everything else needs to fit in the tagged NaN payload. For a tagged NaN we have 51 bits, which is more than enough to represent all other L_3 values.

Following the example of x86-64, we will use only the lower 48 bits of our 64 bit address leaving the 3 bits at indices [48:50] for us to encode the type stored in the actual payload. With these 3 bits we could encode our types as demonstrated in figure 2. Observe that a type tag for NaN is explicetly encoded because we also need to allow NaNs, as well as the other two *special values*: [+/-]∞. However, no explicit

Value	Type Encoding
NaN	000
Integer	001
Pointer	010
Character	011
Boolean True	100
Boolean False	101
Unit	110

Figure 2: Possible encoding of 3-bit type tags.

handling is needed for the infinities as they are represented by having all bits of T - including the quiet bit - set to 0.

For the sake of normality, I will assume 32-bit integers (instead of 31) even though, theoretically, they could have 48 bits. As before, checking for a word's type, or extracting the value, can be done with a simple bitmask. Due to the introduction of primitive floating-point operations, all other functionality will remain the same. As a final example, if you wanted to check if a given word is an integer, you would mask the most significant 16 bits with the mask `#b0111111111111001`, and to extract the integer value you can simply use the lower 32 bits of the word.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. 2019. `https://ieeexplore.ieee.org/servlet/opac?punumber=8766227`