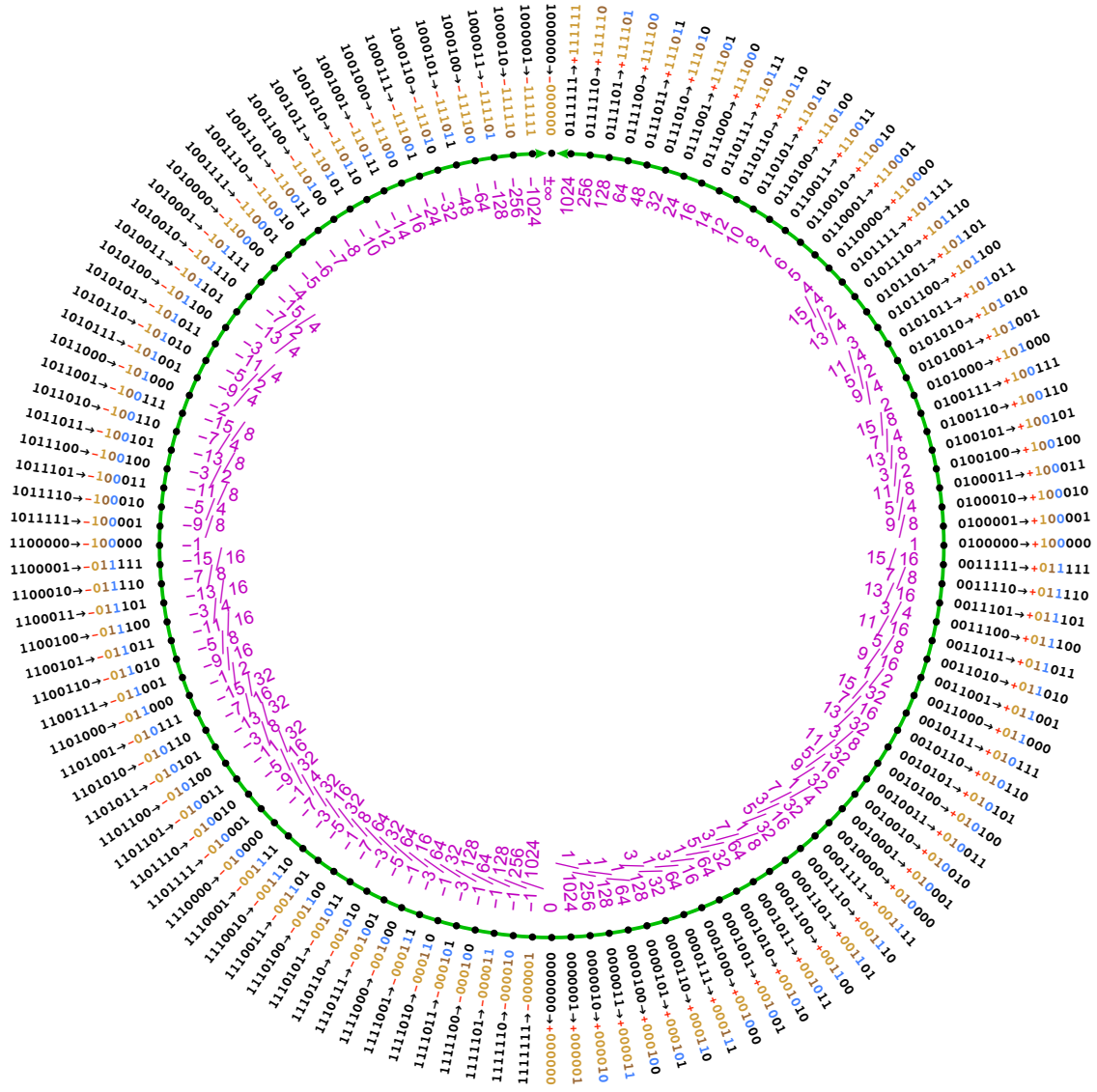
```
setpositenv[{7, 1}]; ringplot
```

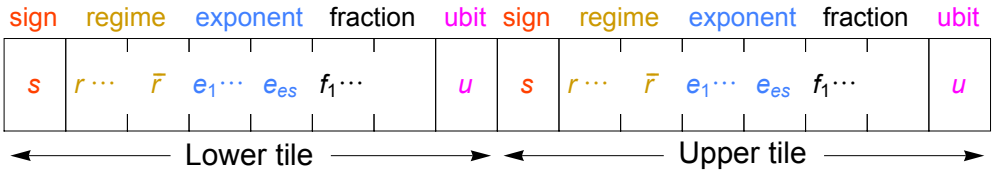


Perhaps this is a good place to point out how rounding is subtly different from what it is for floats. If the rounded bit is a fraction bit, it's *exactly* the same as for floats. Round to the nearest value, and if there's a tie, pick the number ending in a **0** bit. But suppose a calculation landed between 64 and 256; what is the midpoint where we decide if we round up or down? Notice that the rounded bit there is an *exponent* bit, so you are rounding to the *nearest exponent*. For the hardware, it's exactly the same algorithm as when rounding fraction bits. The midpoint between 64 and 256, if we had one more bit of precision, is 128. Numbers between 64 and 128 round to 64; numbers between 128 and 256 round to 256; the exact value 128 is a tie, and 64 is the posit ending in a **0** bit, so 128 rounds to 64.

The rounding rules are therefore consistent, except for results near the exception values 0 and ±∞. Never round toward those extremes. Use *–minpos*, *minpos*, *–maxpos*, or *maxpos* instead. At the very least, that preserves the correct *sign* of the answer.

3.6 Sneak Preview: Valid Arithmetic

One reason for carrying ringplots all the way to seven bits is that the one above forms an excellent basis for 16-bit valids. As a preview to the world of powerful, guess-free arithmetic, valids are pairs of posits that each have an uncertainty bit, or *ubit* appended to the fraction, forming a *tile*. Ubits are color-coded in magenta (RGB 1, 0, 1). The ubit is **0** for all the values shown above, and **1** for all the *open intervals between the values* shown above. A possible 16-bit valid format is a pair of 7-bit posits with ubits at the end of each one, like this:



(Again we use a general *es* value for the purpose of illustration. The *es* value for such a small number of bits would almost certainly be 0. In the ringplot above, *es* is 1.)

Each tile can be any value shown in the above ring of 7-bit posits, followed by a **0** ubit or a **1** ubit. Conventional interval arithmetic is in the form of closed intervals [*a*, *b*] where *a* and *b* are floats, and $a \le b$. In the case of valids *The endpoints need not be ordered*. The lower tile says where to start on the ring; the interval represented includes all tiles counterclockwise from the lower tile until you reach the upper tile, which is also included in the set; it wraps around the circle, crossing ±∞ if necessary! This allows representation of closed, open, and half-open intervals, and because of the use of the projective reals, contiguous intervals **remain contiguous** under addition, subtraction, multiplication, **and division**. For example, the half-open interval (–1/4, 1/16] in the {8, 1} environment of the above ringplot is represented by the pair of tiles

11100011→**–00111011**, representing the tile (–1/4, –7/32), and
00100000→**–00100000**, representing the tile 1/16.

That's a compact way to say the set of all tiles between the two points:

$$(-1/4, -7/32),\ -7/32,\ (-7/32, -3/16),\ -3/16,\ \ldots,\ 3/64,\ (3/64, 1/16),\ 1/16.$$

To reiterate: *posits* are for where floats are good enough, and the algorithms have been shown reliable enough to satisfy user requirements. In contrast, *valids* are for where you need a *provable bound on the answer*. Or when you are still developing an algorithm and debugging its numerical behavior. Or where you want to describe *sets* of real numbers and not just point values. The algorithms for valids are often quite different from the ones for floats, and vice versa. Valids can express a rich source of exception conditions that are useful for debugging, such as the empty set, the entire set of real numbers, or trigger a halt when the interval has gotten too wide (as set by the user) to be useful. Details about the use of valids are discussed elsewhere, but for now the interested reader is referred to *The End of Error: Unum Arithmetic* and its discussion of "ubounds."