Direct Conversion to Posit without Loops or Branches

Statement of Problem

We wish to convert a nonzero real number *x* into the posit bit string *p* that most closely represents it, by posit rounding rules. Notice that 0 is not a possibility value for *x*, nor can *x* be of infinite magnitude. We assume *x* is not necessarily within the dynamic range covered by the posit, and we also assume *f* is more precise than the number of fraction bits available for the posit. The output posit bit string is in an environment set by {*nbits*, *es*}, where *nbits* is the total number of bits in the posit format, and *es* is the maximum number of exponent bits.

Furthermore, we wish to express the conversion without loops or branch conditions, just a straight-line formula that emits the right bits for *p.* We can treat *p* as a binary integer for purposes of setting its bit state.

The environment values *minpos* and *maxpos* come into play (the smallest and largest representable, so to make this a standalone explanation we define them here based on *es* and *nbits.* We won't need *useed*, except as a step to computing *minpos* and *maxpos*.

$$\mathbf{useed} = \mathbf{2}^{\mathbf{2}^{\mathbf{es}}};$$
$$\{\mathbf{minpos},\ \mathbf{maxpos}\} = \left\{\mathbf{useed}^{-\mathbf{nbits}+\mathbf{2}},\ \mathbf{useed}^{\mathbf{nbits}-\mathbf{2}}\right\};$$

Build up the pieces

The sign bit *s* is easy. The **`Boole`** function in Mathematica returns numeric value 1 if the argument is `True`, 0 if it is `False`:

```
s = Boole[x < 0];
```

The simplest way to take care of too-large or too-small magnitude argument is to use the minimum and maximum functions to bring the magnitude of the value into the range *minpos* ≤ *x* ≤ *maxpos*. Is that cheating? A minimum or maximum function (or even an absolute value function) looks a little like a conditional statement, but it really isn't. There are circuits that directly compute the minimum or maximum of two numbers, and the absolute value. There is no need for an "IF" statement construct.

```
y = Max[minpos, Min[maxpos, Abs[x]]];
```

Now the regime bits. First, we need to know if the regime bits r are 0s or 1s. Again, it is a simple Boolean function:

```
r = Boole[y ≥ 1];
```

Things start to get interesting in computing the regime run length, that is, how many r regime bits in a row there are, but doing it without a loop structure. The power-of-2 scaling e can be extracted directly if y is stored in floating-point format. Failing that, we assume it is possible to find the scaling with the floor function of the log base 2. While we're at it, we can find the fraction f as a value with $1 \leq f < 2$ by scaling y down by a factor of 2^e. The fraction is always between 1 and 2 for posits because there are no "subnormal" posits like there are with floats, so we subtract 1 from f here as step toward figuring out what the fraction bits are.

```
e = Floor[Log[2, y]];
f = y / 2^e - 1;
```

To find the number of run bits, use integer floored division, $q = \left\lfloor \frac{a}{b} \right\rfloor$. Since the denominator is a power of 2, there are fast ways to do this in hardware (or in C) simply by shifting. Take the absolute value, since we are looking for run length and not a bit shift. We also adjust for whether the number y is in the northeast quadrant ($r = 1$) or the southeast quadrant ($r = 0$). The run length is one bit longer if r is 1, so the simplest way to adjust directly is to add r.

```
run = Abs[Floor[e / 2^es]] + r;
```

In C or with hardware, it is pretty easy to create a function that produces *run* bits in a row that are r, followed by $\bar{r}$. In *Mathematica*, it looks clunky to do it with arithmetic (using the fact that $2^n - 1$ in binary is a string of n 1s in a row), but it gets the job done. In C at least, an arithmetic right shift that duplicates the leftmost bit, seems like the right way to do this. The **BitXor** function is one way to flip the r bit.

```
reg = BitOr[BitShiftLeft[r * (2^run - 1), 1], BitXor[1, r]];
```

It is possible that the run is as large as *nbits* – 1, leaving no bits for the exponent or fraction. Ignore that for now and compute the value of the exponent bits, *esval*:

```
esval = Mod[e, 2^es];
```

In other words, *esval* is the remainder of the integer division used to compute the run length. It is an integer, $0 \leq esval < 2^{es}$, hence representable as an unsigned integer with *es* bits.

To round correctly, we need to convert out to at least $nbits + 2$ bits. That's *nbits* for the posit, one more bit of accuracy, and then the "sticky bit" after that, meaning "There are nonzero bits beyond $nbits + 1$". The number of bits generated so far is 1 sign bit, *run* regime bits, the complement of the regime bit to terminate the run, and *es* exponent bits. That's $2 + run + es$. The number of fraction bits still needed is *nf*, which is the maximum of zero or $nbits + 1$ minus the bits generated so far, $2 + run + es$. (Remember, having no fraction bits at all simply means the fraction is 1, because of the hidden bit.) The length of all the bits we need, temporarily, to hold all the generated fields, is 1 more than the maximum of $nbits + 1$ and $2 + run + es$, because we still need space for the sticky bit.

```
nf = Max[0, (nbits + 1) - (2 + run + es)];
len = 1 + Max[nbits + 1, 2 + run + es];
```

If you're a hardware designer, you probably need to know the maximum size of *len* so a design can allocate the right number of bits to hold the temporary value. That maximum is $nbits + es + 2$, and it reaches that maximum for the very largest and very smallest *x* values.

The fraction value *fv* is produced by scaling *f* by 2^{nf}, with the floor function again because it's truncated, and we compute the sticky bit *sb* by seeing if there was nothing to truncate; that is, scaling produces an exact integer. If it is exact, $sb = 0$. Otherwise, there are trailing bits.

```
fv = Floor[f * 2^nf];
sb = Boole[f * 2^nf > fv];
```

Construct *pt*, the temporarily too-long version of the posit bit string, by ORing together the regime, exponent, fraction, and sticky bit fields.

```
pt = BitOr[BitShiftLeft[reg, es + nf + 1],
   BitShiftLeft[esval, nf + 1], BitShiftLeft[fv, 1], sb];
```

The penultimate step is to round correctly. The round-up amount, *rup,* is **1** if we need to add 1 to the posit after clipping it back to a length of *nbits*; otherwise it is **0**. This involves three bits and is the traditional round-to-nearest, tie-goes-to-even method of rounding that is the default for IEEE 754 floats. Let *blast* be the last bit of the unrounded posit, *bafter* be the bit just after the last bit of the posit, and *bsticky* be the OR of all bits beyond *bafter*, here computed with an arithmetic "greater than zero" comparison but more properly computed as a multi-bit OR. The formula below uses $2^{len-nbits-1} - 1$ to create a mask of 1 bits beyond position *nbits*+1, which then finds if *any* of the trailing bits of the temporary value are nonzero. The `BitGet[a,n]` function finds the bit corresponding to the coefficient of 2^n in the integer *a*, so the second argument is the number of bit locations from the right end of the integer, not the left end.

```
blast = BitGet[pt, len - nbits];
bafter = BitGet[pt, len - nbits - 1];
bsticky = Boole[BitAnd[2^(len-nbits-1) - 1, pt] > 0];
```

A Karnaugh map can be used to find the three combinations of *blast*, *bafter*, and *bsticky* that result in a need to add 1 to the truncated posit. Perhaps the simplest way to describe it is that *rb* is 1 if both *blast* and *bafter* are 1, or if both *bafter* and *bsticky* are 1. That would be

```
rb = BitOr[BitAnd[blast, bafter], BitAnd[bafter, bsticky]];
```

The following is equivalent, and uses two logic operations instead of three:

```
rb = BitAnd[bafter, BitXor[blast, bsticky]];
```

Shift right to make it *nbits* long, and add the rounding bit. The result still a temporary value, so call it *ptt*.

```
ptt = BitShiftRight[pt, len - nbits] + rb;
```

Finally, apply the sign bit. In C or in hardware this is easy; just treat the bit string as a signed integer and take the 2's complement of the number if *s* is 1. Of course, that's an IF statement. The more direct computation is to create a mask of all *s* bits, *nbits* long, XOR that with *ptt*, and add *s*. This takes the 2's complement if *s* is 1, and leaves the number unaffected if *s* is 0. So finally we have our posit bit string, *p*.

```
p = BitXor[s * (2^nbits - 1), ptt] + s;
```

We can make it into a single Mathematica function for ease of testing. Clear all previous variable assignments, first.

```
Clear[x, s, y, r, e, f, run, reg, esval, nf, len,
 fv, sb, p, pt, ptt, blast, bafter, bsticky, rb, ptt, p]
```

The posit convertor program is simply the catenation of all the definitions given so far. It all fits in 20 lines of code, or 22 lines if you count the long lines that experienced wraparound. Notice that the code is single-assignment; no variable is assigned a value more than once. That wastes a bit of storage, but may assist in transcribing this function into circuit form.

```
p[x_] := Module[{s, y, r, e, f, run, reg, esval,
    nf, len, fv, sb, pt, blast, bafter, bsticky, rb, ptt, p},
  s = Boole[x < 0];
  y = Max[minpos, Min[maxpos, Abs[x]]];
  r = Boole[y ≥ 1];
  e = Floor[Log[2, y]];
  f = y / 2^e - 1;
  run = Abs[Floor[e / 2^es]] + r;
  reg = BitOr[BitShiftLeft[r * (2^run - 1), 1], BitXor[1, r]];
  esval = Mod[e, 2^es];
  nf = Max[0, (nbits + 1) - (2 + run + es)];
  len = 1 + Max[nbits + 1, 2 + run + es];
  fv = Floor[f * 2^nf];
  sb = Boole[f * 2^nf > fv];
  pt = BitOr[BitShiftLeft[reg, es + nf + 1],
    BitShiftLeft[esval, nf + 1], BitShiftLeft[fv, 1], sb];
  blast = BitGet[pt, len - nbits];
  bafter = BitGet[pt, len - nbits - 1];
  bsticky = Boole[BitAnd[2^(len-nbits-1) - 1, pt] > 0];
  rb = BitOr[BitAnd[blast, bafter], BitAnd[bafter, bsticky]];
  ptt = BitShiftRight[pt, len - nbits] + rb;
  BitXor[s * (2^nbits - 1), ptt] + s]
```

Tests and Examples

As an example of a test, try *nbits* = 8 and *es* = 3. That may seem like a crazy *es* value for such a small posit and it's not the recommended standard *es* = 0 for 8-bit posits, but it makes for a good test set. (It's possible that a customized 64-bit app might want to start with {8, 3} posits and then grow the accuracy to {64, 3} simply by appending bits to the right of each posit as an algorithm converges.)

Here are tables of the values of every {8, 3} posit other than 1 and ±∞.

```
setpositenv[{8, 3}]
Table[{p, p2x[p], colorcodep[p]}, {p, 1, npat / 2 - 1}] // TableForm
Table[{p, p2x[p], colorcodep[p]}, {p, npat / 2 + 1, npat - 1}] // TableForm
```

1	$\frac{1}{281\,474\,976\,710\,656}$	00000001→+0000001
2	$\frac{1}{1\,099\,511\,627\,776}$	00000010→+0000010
3	$\frac{1}{68\,719\,476\,736}$	00000011→+0000011
4	$\frac{1}{4\,294\,967\,296}$	00000100→+0000100
5	$\frac{1}{1\,073\,741\,824}$	00000101→+0000101
6	$\frac{1}{268\,435\,456}$	00000110→+0000110
7	$\frac{1}{67\,108\,864}$	00000111→+0000111

8	$\frac{1}{16777216}$	00001000→+0001000
9	$\frac{1}{8388608}$	00001001→+0001001
10	$\frac{1}{4194304}$	00001010→+0001010
11	$\frac{1}{2097152}$	00001011→+0001011
12	$\frac{1}{1048576}$	00001100→+0001100
13	$\frac{1}{524288}$	00001101→+0001101
14	$\frac{1}{262144}$	00001110→+0001110
15	$\frac{1}{131072}$	00001111→+0001111
16	$\frac{1}{65536}$	00010000→+0010000
17	$\frac{3}{131072}$	00010001→+0010001
18	$\frac{1}{32768}$	00010010→+0010010
19	$\frac{3}{65536}$	00010011→+0010011
20	$\frac{1}{16384}$	00010100→+0010100
21	$\frac{3}{32768}$	00010101→+0010101
22	$\frac{1}{8192}$	00010110→+0010110
23	$\frac{3}{16384}$	00010111→+0010111
24	$\frac{1}{4096}$	00011000→+0011000
25	$\frac{3}{8192}$	00011001→+0011001
26	$\frac{1}{2048}$	00011010→+0011010
27	$\frac{3}{4096}$	00011011→+0011011
28	$\frac{1}{1024}$	00011100→+0011100
29	$\frac{3}{2048}$	00011101→+0011101
30	$\frac{1}{512}$	00011110→+0011110
31	$\frac{3}{1024}$	00011111→+0011111
32	$\frac{1}{256}$	00100000→+0100000
33	$\frac{5}{1024}$	00100001→+0100001
34	$\frac{3}{512}$	00100010→+0100010
35	$\frac{7}{1024}$	00100011→+0100011
36	$\frac{1}{128}$	00100100→+0100100
37	$\frac{5}{512}$	00100101→+0100101
38	$\frac{3}{256}$	00100110→+0100110
39	$\frac{7}{512}$	00100111→+0100111
40	$\frac{1}{64}$	00101000→+0101000
41	$\frac{5}{256}$	00101001→+0101001

42	$\frac{3}{128}$	00101010→+0101010
43	$\frac{7}{256}$	00101011→+0101011
44	$\frac{1}{32}$	00101100→+0101100
45	$\frac{5}{128}$	00101101→+0101101
46	$\frac{3}{64}$	00101110→+0101110
47	$\frac{7}{128}$	00101111→+0101111
48	$\frac{1}{16}$	00110000→+0110000
49	$\frac{5}{64}$	00110001→+0110001
50	$\frac{3}{32}$	00110010→+0110010
51	$\frac{7}{64}$	00110011→+0110011
52	$\frac{1}{8}$	00110100→+0110100
53	$\frac{5}{32}$	00110101→+0110101
54	$\frac{3}{16}$	00110110→+0110110
55	$\frac{7}{32}$	00110111→+0110111
56	$\frac{1}{4}$	00111000→+0111000
57	$\frac{5}{16}$	00111001→+0111001
58	$\frac{3}{8}$	00111010→+0111010
59	$\frac{7}{16}$	00111011→+0111011
60	$\frac{1}{2}$	00111100→+0111100
61	$\frac{5}{8}$	00111101→+0111101
62	$\frac{3}{4}$	00111110→+0111110
63	$\frac{7}{8}$	00111111→+0111111
64	1	01000000→+1000000
65	$\frac{5}{4}$	01000001→+1000001
66	$\frac{3}{2}$	01000010→+1000010
67	$\frac{7}{4}$	01000011→+1000011
68	2	01000100→+1000100
69	$\frac{5}{2}$	01000101→+1000101
70	3	01000110→+1000110
71	$\frac{7}{2}$	01000111→+1000111
72	4	01001000→+1001000
73	5	01001001→+1001001
74	6	01001010→+1001010
75	7	01001011→+1001011
76	8	01001100→+1001100
77	10	01001101→+1001101
78	12	01001110→+1001110
79	14	01001111→+1001111
80	16	01010000→+1010000
81	20	01010001→+1010001

82	24	01010010→+1010010
83	28	01010011→+1010011
84	32	01010100→+1010100
85	40	01010101→+1010101
86	48	01010110→+1010110
87	56	01010111→+1010111
88	64	01011000→+1011000
89	80	01011001→+1011001
90	96	01011010→+1011010
91	112	01011011→+1011011
92	128	01011100→+1011100
93	160	01011101→+1011101
94	192	01011110→+1011110
95	224	01011111→+1011111
96	256	01100000→+1100000
97	384	01100001→+1100001
98	512	01100010→+1100010
99	768	01100011→+1100011
100	1024	01100100→+1100100
101	1536	01100101→+1100101
102	2048	01100110→+1100110
103	3072	01100111→+1100111
104	4096	01101000→+1101000
105	6144	01101001→+1101001
106	8192	01101010→+1101010
107	12 288	01101011→+1101011
108	16 384	01101100→+1101100
109	24 576	01101101→+1101101
110	32 768	01101110→+1101110
111	49 152	01101111→+1101111
112	65 536	01110000→+1110000
113	131 072	01110001→+1110001
114	262 144	01110010→+1110010
115	524 288	01110011→+1110011
116	1 048 576	01110100→+1110100
117	2 097 152	01110101→+1110101
118	4 194 304	01110110→+1110110
119	8 388 608	01110111→+1110111
120	16 777 216	01111000→+1111000
121	67 108 864	01111001→+1111001
122	268 435 456	01111010→+1111010
123	1 073 741 824	01111011→+1111011
124	4 294 967 296	01111100→+1111100
125	68 719 476 736	01111101→+1111101
126	1 099 511 627 776	01111110→+1111110
127	281 474 976 710 656	01111111→+1111111

129	−281 474 976 710 656	10000001→−1111111
130	−1 099 511 627 776	10000010→−1111110
131	−68 719 476 736	10000011→−1111101
132	−4 294 967 296	10000100→−1111100
133	−1 073 741 824	10000101→−1111011
134	−268 435 456	10000110→−1111010
135	−67 108 864	10000111→−1111001
136	−16 777 216	10001000→−1111000

137	−8 388 608	10001001→−1110111
138	−4 194 304	10001010→−1110110
139	−2 097 152	10001011→−1110101
140	−1 048 576	10001100→−1110100
141	−524 288	10001101→−1110011
142	−262 144	10001110→−1110010
143	−131 072	10001111→−1110001
144	−65 536	10010000→−1110000
145	−49 152	10010001→−1101111
146	−32 768	10010010→−1101110
147	−24 576	10010011→−1101101
148	−16 384	10010100→−1101100
149	−12 288	10010101→−1101011
150	−8192	10010110→−1101010
151	−6144	10010111→−1101001
152	−4096	10011000→−1101000
153	−3072	10011001→−1100111
154	−2048	10011010→−1100110
155	−1536	10011011→−1100101
156	−1024	10011100→−1100100
157	−768	10011101→−1100011
158	−512	10011110→−1100010
159	−384	10011111→−1100001
160	−256	10100000→−1100000
161	−224	10100001→−1011111
162	−192	10100010→−1011110
163	−160	10100011→−1011101
164	−128	10100100→−1011100
165	−112	10100101→−1011011
166	−96	10100110→−1011010
167	−80	10100111→−1011001
168	−64	10101000→−1011000
169	−56	10101001→−1010111
170	−48	10101010→−1010110
171	−40	10101011→−1010101
172	−32	10101100→−1010100
173	−28	10101101→−1010011
174	−24	10101110→−1010010
175	−20	10101111→−1010001
176	−16	10110000→−1010000
177	−14	10110001→−1001111
178	−12	10110010→−1001110
179	−10	10110011→−1001101
180	−8	10110100→−1001100
181	−7	10110101→−1001011
182	−6	10110110→−1001010
183	−5	10110111→−1001001
184	−4	10111000→−1001000
185	$-\frac{7}{2}$	10111001→−1000111
186	−3	10111010→−1000110
187	$-\frac{5}{2}$	10111011→−1000101
188	−2	10111100→−1000100
189	$-\frac{7}{4}$	10111101→−1000011

190	$-\frac{3}{2}$	10111110→-1000010
191	$-\frac{5}{4}$	10111111→-1000001
192	-1	11000000→-1000000
193	$-\frac{7}{8}$	11000001→-0111111
194	$-\frac{3}{4}$	11000010→-0111110
195	$-\frac{5}{8}$	11000011→-0111101
196	$-\frac{1}{2}$	11000100→-0111100
197	$-\frac{7}{16}$	11000101→-0111011
198	$-\frac{3}{8}$	11000110→-0111010
199	$-\frac{5}{16}$	11000111→-0111001
200	$-\frac{1}{4}$	11001000→-0111000
201	$-\frac{7}{32}$	11001001→-0110111
202	$-\frac{3}{16}$	11001010→-0110110
203	$-\frac{5}{32}$	11001011→-0110101
204	$-\frac{1}{8}$	11001100→-0110100
205	$-\frac{7}{64}$	11001101→-0110011
206	$-\frac{3}{32}$	11001110→-0110010
207	$-\frac{5}{64}$	11001111→-0110001
208	$-\frac{1}{16}$	11010000→-0110000
209	$-\frac{7}{128}$	11010001→-0101111
210	$-\frac{3}{64}$	11010010→-0101110
211	$-\frac{5}{128}$	11010011→-0101101
212	$-\frac{1}{32}$	11010100→-0101100
213	$-\frac{7}{256}$	11010101→-0101011
214	$-\frac{3}{128}$	11010110→-0101010
215	$-\frac{5}{256}$	11010111→-0101001
216	$-\frac{1}{64}$	11011000→-0101000
217	$-\frac{7}{512}$	11011001→-0100111
218	$-\frac{3}{256}$	11011010→-0100110
219	$-\frac{5}{512}$	11011011→-0100101
220	$-\frac{1}{128}$	11011100→-0100100
221	$-\frac{7}{1024}$	11011101→-0100011
222	$-\frac{3}{512}$	11011110→-0100010
223	$-\frac{5}{1024}$	11011111→-0100001
224	$-\frac{1}{256}$	11100000→-0100000

225	$-\frac{3}{1024}$	11100001→-0011111
226	$-\frac{1}{512}$	11100010→-0011110
227	$-\frac{3}{2048}$	11100011→-0011101
228	$-\frac{1}{1024}$	11100100→-0011100
229	$-\frac{3}{4096}$	11100101→-0011011
230	$-\frac{1}{2048}$	11100110→-0011010
231	$-\frac{3}{8192}$	11100111→-0011001
232	$-\frac{1}{4096}$	11101000→-0011000
233	$-\frac{3}{16\,384}$	11101001→-0010111
234	$-\frac{1}{8192}$	11101010→-0010110
235	$-\frac{3}{32\,768}$	11101011→-0010101
236	$-\frac{1}{16\,384}$	11101100→-0010100
237	$-\frac{3}{65\,536}$	11101101→-0010011
238	$-\frac{1}{32\,768}$	11101110→-0010010
239	$-\frac{3}{131\,072}$	11101111→-0010001
240	$-\frac{1}{65\,536}$	11110000→-0010000
241	$-\frac{1}{131\,072}$	11110001→-0001111
242	$-\frac{1}{262\,144}$	11110010→-0001110
243	$-\frac{1}{524\,288}$	11110011→-0001101
244	$-\frac{1}{1\,048\,576}$	11110100→-0001100
245	$-\frac{1}{2\,097\,152}$	11110101→-0001011
246	$-\frac{1}{4\,194\,304}$	11110110→-0001010
247	$-\frac{1}{8\,388\,608}$	11110111→-0001001
248	$-\frac{1}{16\,777\,216}$	11111000→-0001000
249	$-\frac{1}{67\,108\,864}$	11111001→-0000111
250	$-\frac{1}{268\,435\,456}$	11111010→-0000110
251	$-\frac{1}{1\,073\,741\,824}$	11111011→-0000101
252	$-\frac{1}{4\,294\,967\,296}$	11111100→-0000100
253	$-\frac{1}{68\,719\,476\,736}$	11111101→-0000011
254	$-\frac{1}{1\,099\,511\,627\,776}$	11111110→-0000010
255	$-\frac{1}{281\,474\,976\,710\,656}$	11111111→-0000001

As a first “sanity check,” see if it converts exact posits, expressed as real values, into the correct posit viewed as an unsigned integer. The second and third columns below should match for every row of output, and they do.

```
Table[{p2x[i], i, p[p2x[i]]}, {i, 1, npat / 2 - 1}] // TableForm
Table[{p2x[i], i, p[p2x[i]]}, {i, npat / 2 + 1, npat - 1}] // TableForm
```

$\frac{1}{281\,474\,976\,710\,656}$	1	1
$\frac{1}{1\,099\,511\,627\,776}$	2	2
$\frac{1}{68\,719\,476\,736}$	3	3
$\frac{1}{4\,294\,967\,296}$	4	4
$\frac{1}{1\,073\,741\,824}$	5	5
$\frac{1}{268\,435\,456}$	6	6
$\frac{1}{67\,108\,864}$	7	7
$\frac{1}{16\,777\,216}$	8	8
$\frac{1}{8\,388\,608}$	9	9
$\frac{1}{4\,194\,304}$	10	10
$\frac{1}{2\,097\,152}$	11	11
$\frac{1}{1\,048\,576}$	12	12
$\frac{1}{524\,288}$	13	13
$\frac{1}{262\,144}$	14	14
$\frac{1}{131\,072}$	15	15
$\frac{1}{65\,536}$	16	16
$\frac{3}{131\,072}$	17	17
$\frac{1}{32\,768}$	18	18
$\frac{3}{65\,536}$	19	19
$\frac{1}{16\,384}$	20	20
$\frac{3}{32\,768}$	21	21
$\frac{1}{8192}$	22	22
$\frac{3}{16\,384}$	23	23
$\frac{1}{4096}$	24	24
$\frac{3}{8192}$	25	25
$\frac{1}{2048}$	26	26
$\frac{3}{4096}$	27	27
$\frac{1}{1024}$	28	28
$\frac{3}{2048}$	29	29
$\frac{1}{512}$	30	30
$\frac{3}{1024}$	31	31

$\frac{1}{256}$	32	32
$\frac{5}{1024}$	33	33
$\frac{3}{512}$	34	34
$\frac{7}{1024}$	35	35
$\frac{1}{128}$	36	36
$\frac{5}{512}$	37	37
$\frac{3}{256}$	38	38
$\frac{7}{512}$	39	39
$\frac{1}{64}$	40	40
$\frac{5}{256}$	41	41
$\frac{3}{128}$	42	42
$\frac{7}{256}$	43	43
$\frac{1}{32}$	44	44
$\frac{5}{128}$	45	45
$\frac{3}{64}$	46	46
$\frac{7}{128}$	47	47
$\frac{1}{16}$	48	48
$\frac{5}{64}$	49	49
$\frac{3}{32}$	50	50
$\frac{7}{64}$	51	51
$\frac{1}{8}$	52	52
$\frac{5}{32}$	53	53
$\frac{3}{16}$	54	54
$\frac{7}{32}$	55	55
$\frac{1}{4}$	56	56
$\frac{5}{16}$	57	57
$\frac{3}{8}$	58	58
$\frac{7}{16}$	59	59
$\frac{1}{2}$	60	60
$\frac{5}{8}$	61	61
$\frac{3}{4}$	62	62
$\frac{7}{8}$	63	63
1	64	64
$\frac{5}{4}$	65	65
$\frac{3}{2}$	66	66

$\frac{7}{4}$	67	67
2	68	68
$\frac{5}{2}$	69	69
3	70	70
$\frac{7}{2}$	71	71
4	72	72
5	73	73
6	74	74
7	75	75
8	76	76
10	77	77
12	78	78
14	79	79
16	80	80
20	81	81
24	82	82
28	83	83
32	84	84
40	85	85
48	86	86
56	87	87
64	88	88
80	89	89
96	90	90
112	91	91
128	92	92
160	93	93
192	94	94
224	95	95
256	96	96
384	97	97
512	98	98
768	99	99
1024	100	100
1536	101	101
2048	102	102
3072	103	103
4096	104	104
6144	105	105
8192	106	106
12 288	107	107
16 384	108	108
24 576	109	109
32 768	110	110
49 152	111	111
65 536	112	112
131 072	113	113
262 144	114	114
524 288	115	115
1 048 576	116	116
2 097 152	117	117
4 194 304	118	118
8 388 608	119	119

16 777 216	120	120
67 108 864	121	121
268 435 456	122	122
1 073 741 824	123	123
4 294 967 296	124	124
68 719 476 736	125	125
1 099 511 627 776	126	126
281 474 976 710 656	127	127

−281 474 976 710 656	129	129
−1 099 511 627 776	130	130
−68 719 476 736	131	131
−4 294 967 296	132	132
−1 073 741 824	133	133
−268 435 456	134	134
−67 108 864	135	135
−16 777 216	136	136
−8 388 608	137	137
−4 194 304	138	138
−2 097 152	139	139
−1 048 576	140	140
−524 288	141	141
−262 144	142	142
−131 072	143	143
−65 536	144	144
−49 152	145	145
−32 768	146	146
−24 576	147	147
−16 384	148	148
−12 288	149	149
−8192	150	150
−6144	151	151
−4096	152	152
−3072	153	153
−2048	154	154
−1536	155	155
−1024	156	156
−768	157	157
−512	158	158
−384	159	159
−256	160	160
−224	161	161
−192	162	162
−160	163	163
−128	164	164
−112	165	165
−96	166	166
−80	167	167
−64	168	168
−56	169	169
−48	170	170
−40	171	171
−32	172	172
−28	173	173
−24	174	174

-20	175	175
-16	176	176
-14	177	177
-12	178	178
-10	179	179
-8	180	180
-7	181	181
-6	182	182
-5	183	183
-4	184	184
$-\frac{7}{2}$	185	185
-3	186	186
$-\frac{5}{2}$	187	187
-2	188	188
$-\frac{7}{4}$	189	189
$-\frac{3}{2}$	190	190
$-\frac{5}{4}$	191	191
-1	192	192
$-\frac{7}{8}$	193	193
$-\frac{3}{4}$	194	194
$-\frac{5}{8}$	195	195
$-\frac{1}{2}$	196	196
$-\frac{7}{16}$	197	197
$-\frac{3}{8}$	198	198
$-\frac{5}{16}$	199	199
$-\frac{1}{4}$	200	200
$-\frac{7}{32}$	201	201
$-\frac{3}{16}$	202	202
$-\frac{5}{32}$	203	203
$-\frac{1}{8}$	204	204
$-\frac{7}{64}$	205	205
$-\frac{3}{32}$	206	206
$-\frac{5}{64}$	207	207
$-\frac{1}{16}$	208	208
$-\frac{7}{128}$	209	209
$-\frac{3}{64}$	210	210
$-\frac{5}{128}$	211	211
$-\frac{1}{32}$	212	212
$-\frac{7}{256}$	213	213
$-\frac{3}{128}$	214	214

$-\frac{5}{256}$	215	215
$-\frac{1}{64}$	216	216
$-\frac{7}{512}$	217	217
$-\frac{3}{256}$	218	218
$-\frac{5}{512}$	219	219
$-\frac{1}{128}$	220	220
$-\frac{7}{1024}$	221	221
$-\frac{3}{512}$	222	222
$-\frac{5}{1024}$	223	223
$-\frac{1}{256}$	224	224
$-\frac{3}{1024}$	225	225
$-\frac{1}{512}$	226	226
$-\frac{3}{2048}$	227	227
$-\frac{1}{1024}$	228	228
$-\frac{3}{4096}$	229	229
$-\frac{1}{2048}$	230	230
$-\frac{3}{8192}$	231	231
$-\frac{1}{4096}$	232	232
$-\frac{3}{16384}$	233	233
$-\frac{1}{8192}$	234	234
$-\frac{3}{32768}$	235	235
$-\frac{1}{16384}$	236	236
$-\frac{3}{65536}$	237	237
$-\frac{1}{32768}$	238	238
$-\frac{3}{131072}$	239	239
$-\frac{1}{65536}$	240	240
$-\frac{1}{131072}$	241	241
$-\frac{1}{262144}$	242	242
$-\frac{1}{524288}$	243	243
$-\frac{1}{1048576}$	244	244
$-\frac{1}{2097152}$	245	245
$-\frac{1}{4194304}$	246	246
$-\frac{1}{8388608}$	247	247
$-\frac{1}{16777216}$	248	248

$-\frac{1}{67\,108\,864}$	249	249
$-\frac{1}{268\,435\,456}$	250	250
$-\frac{1}{1\,073\,741\,824}$	251	251
$-\frac{1}{4\,294\,967\,296}$	252	252
$-\frac{1}{68\,719\,476\,736}$	253	253
$-\frac{1}{1\,099\,511\,627\,776}$	254	254
$-\frac{1}{281\,474\,976\,710\,656}$	255	255

While not shown here, tests have also been done to show the routine can correctly handle real input values that are not exact posits. Here’s an example, and remember that the fraction precision is very low here:

```
p[π]
N[p2x[%], 4]
```

70

3.000

The geometrically-rounded values do just fine as well. For instance:

```
Table[{maxpos / 2^i, IntegerString[p[maxpos / 2^i], 2, nbits], p2x[p[maxpos / 2^i]]},
  {i, 0, 2^(es+1)}] // TableForm
```

281 474 976 710 656	01111111	281 474 976 710 656
140 737 488 355 328	01111111	281 474 976 710 656
70 368 744 177 664	01111111	281 474 976 710 656
35 184 372 088 832	01111111	281 474 976 710 656
17 592 186 044 416	01111110	1 099 511 627 776
8 796 093 022 208	01111110	1 099 511 627 776
4 398 046 511 104	01111110	1 099 511 627 776
2 199 023 255 552	01111110	1 099 511 627 776
1 099 511 627 776	01111110	1 099 511 627 776
549 755 813 888	01111110	1 099 511 627 776
274 877 906 944	01111110	1 099 511 627 776
137 438 953 472	01111101	68 719 476 736
68 719 476 736	01111101	68 719 476 736
34 359 738 368	01111101	68 719 476 736
17 179 869 184	01111100	4 294 967 296
8 589 934 592	01111100	4 294 967 296
4 294 967 296	01111100	4 294 967 296

Forced to choose a power of 2 that is nearest in the sense of ratio, not distance, all the numbers rounded to the nearest ratio, or in the case of a tie, chose the posit with an even bit string. This is why the rounding algorithm is the same no matter where you are on the positive or negative part of the real numbers.

One last demonstration… conversion of $\sqrt{2}$ in a 64-bit posit environment ($es = 3$). It is interesting to compare the posit approximation that with the approximation of $\sqrt{2}$ for a 64-bit float, where 11 bits must be spent to store the power-of-2 scaling factor. With posits, we only need 5 bits to express the power-of-2 scaling factor.

```
setpositenv[{64, 3}]
setfloatenv[{64, 11}]
positroot2 = p[√2];
floatroot2 = x2f[√2];
```

Here are the numerical approximations for 64-bit floats, 64-bit posits, and the correct value rounded to 19 decimal places:

```
N[f2x[floatroot2], 19]
N[p2x[positroot2], 19]
N[√2, 19]
```

```
1.414213562373095145

1.414213562373095048

1.414213562373095049
```

Similar to the results obtained by LLNL, posits are a couple of orders of magnitude more accurate than floats of the same size.

```
N[(√2 - f2x[floatroot2]) / (√2 - p2x[positroot2]), 20]
```

```
-204.99728890830132454
```