Lesson 6: Intro to Floating-Point Representation

Recall that for an m-bit fixed-point representation of a type I rational $x \in \mathbb{Q}_I$, we chose two nonnegative integer parameters $\ell, f \in \mathbb{Z}$ that determined

- the implicit location of the binary point when interpreting the value of the raw m-bit word
- the power in the denominator of our binary fraction (and thus the number of bits we shift the radix point left or right)

Example 6.1) Let $x = \frac{165}{16} = 10.3125$

$$\Rightarrow \quad x = 8 + 2 + \frac{1}{4} + \frac{1}{16}$$

$$\Rightarrow \quad x = 2^3 + 2^1 + \frac{1}{2^2} + \frac{1}{2^4}$$

Example 6.1) Let $x = \left(\frac{165}{16}\right) = \frac{165}{2^4} = 10.3125$

2^4: we know $x \in \mathbb{Q}_I$

$$\Rightarrow \quad x = \frac{128 + 32 + 4 + 1}{16}$$

$$\Rightarrow \quad x = \left(\frac{2^7 + 2^5 + 2^2 + 2^0}{2^4}\right)_{10} = \left(2^3 + 2^1 + 2^{-2} + 2^{-4}\right)_{10}$$

$$\Rightarrow \quad x = \left(\frac{10100101}{10000}\right)_2 = (1010.0101)_2$$

$$\Rightarrow \quad \boxed{x} = (1010.0101)_2$$

x: math object

Let's try to encode $x \in \mathbb{Q}_I$ exactly in fixed-point format

$$\Rightarrow \quad \boxed{\hat{x}} = ufixed_{2(4,4)}\left(\tfrac{165}{16}\right) = ufixed_{2(4,4)}(\underbrace{1010\,0101}_{\text{raw bit string}})$$

$\hat{x}$: programming object

$$= \underbrace{(1010.0101)_2}_{\text{interpreted value}}$$

the value of bit string used to store "something" related to x in a computer's memory

Example 6.1, continued...)

$$\hat{x}_1 = ufixed_{2(5,3)}\left(\frac{165}{16}\right) = ufixed_{2(5,3)}(1010\,0101)$$

$$= \underbrace{01010}_{\ell=5} . \underbrace{010}_{f=3} \boxed{1}$$

what do we do with this extra bit?

- ☐ Chop: 01010.010
- ☐ Round Down: 01010.010
- ☐ Round up: 01010.011

This forces a discussion of rules for rounding...

$$\approx 01010.010$$

this is the first time we've actually needed to approximate a number in this class. Prior to this lesson, all equality was exact... Welcome to numerical analysis :)

let's assume a naive approach and simply chop off all trailing bits that are unsupported by our data format

But notice if we set $\hat{x}_1 = 01010.010$ to be our approximation

then $$\hat{x}_1 = 8 + 2 + \frac{1}{4} = \frac{32 + 8 + 1}{4} = \frac{41}{4} = \frac{164}{16}$$

$\Rightarrow$ we have an absolute error in our approximation of

$$|x - \hat{x}_1| = \left|\frac{165}{16} - \frac{164}{16}\right| = \left|\frac{1}{16}\right| = \frac{1}{16}$$

Example 6.1, continued...)

Let's try a different method for dealing with the extra bit from our exact number that can't be stored in our $\text{ufixed}_{2(5,3)}\left(\frac{165}{16}\right)$ format.

$$\hat{x}_2 = \text{ufixed}_{2(5,3)}\left(\frac{165}{16}\right)$$

$$= \text{ufixed}_{2(5,3)}(10100101)$$

$$= \underbrace{01010}_{\ell=5}.\underbrace{010}_{f=3}\boxed{1}$$ this time, let's round up

$$\approx 01010.011$$

Now let's set $\hat{x}_2 = 01010.011$ to be our second approximation

$$\Rightarrow \quad \hat{x}_2 = 8 + 2 + \frac{1}{4} + \frac{1}{8} = \frac{64 + 16 + 2 + 1}{8} = \frac{83}{8} = \frac{166}{16} > x$$

we rounded up "

$\Rightarrow$ we have absolute error in our approximation of

$$|x - \hat{x}_2| = \left|\frac{165}{16} - \frac{166}{16}\right| = \left|\frac{-1}{16}\right| = \frac{1}{16}$$

Example 6.1, continued...)

Now let's try storing x in the $\text{ufixed}_{2(3,5)}\left(\frac{165}{16}\right)$ format and see what happens. To this end, we set

$$\hat{x}_3 = \text{ufixed}_{2(3,5)}\left(\frac{165}{16}\right)$$

$$= \text{ufixed}_{2(3,5)}(1010\ 0101)$$

$$= \boxed{1}\ \underbrace{010}_{e=3}.\ \underbrace{01010}_{f=5}$$

what do we do with this MSB?

$$\approx 010.01010$$

- Chop: 010.01010
- Round up: 110.01010

let's assume a naive approach again and simply chop off the MSB

$$\Rightarrow \hat{x}_3 = 2 + \frac{1}{4} + \frac{1}{16} = \frac{32+4+1}{16} = \frac{37}{16}$$

$\Rightarrow$ we have absolute error in our approximation of

$$|x - \hat{x}_3| = \left|\frac{165}{16} - \frac{37}{16}\right) = \left|\frac{128}{16}\right| = +8 \text{ units}$$

this makes sense because we chopped off $+8 = 2^3$ bit.

Example 6.1, continued...)

Of course, we might try to round up using the $ufixed_{2(3,5)}$ format to yield

$$\hat{X}_4 = ufixed_{2(3,5)}\left(\frac{165}{16}\right)$$

$$= ufixed_{2(3,5)}\,(1010\ 0101)$$

$$= \boxed{1}\ \underbrace{010}_{\ell=3}.\ \underbrace{01010}_{f=5}$$

$$\approx 110.01010$$

$$\Rightarrow \hat{X}_4 = 4 + 2 + \frac{1}{4} + \frac{1}{16} = \frac{64 + 32 + 4 + 1}{16} = \frac{101}{16}$$

$\Rightarrow$ we have absolute error in our approximation of

$$|X - \hat{X}_4| = \left|\frac{165}{16} - \frac{101}{16}\right| = \left|\frac{64}{16}\right| = +4$$

this error makes sense since we rounded the exact 4-bit integer value 1010 to an approximate 3-bit integer 110 and $1010 - 110 = 100$

Example 6.1, continued ...)

Notice that our choice of parameters $\ell, f \in \mathbb{Z}$ matter. In only one case can we encode the exact value of $X = \frac{165}{16}$. In all other formats, we must use an approximation to X.

Data type and rounding rule	Binary Value	Absolute Error	Type of error
☐ $\text{ufixed}_{2(4,4)}\left(\frac{165}{16}\right)$	$1010.0101 = \hat{x}$	$\lvert x - \hat{x} \rvert = 0$	NONE
☐ $\text{ufixed}_{2(5,3)}\left(\frac{165}{16}\right)$ (use chop)	$\hat{x}_1 = 01010.010$ ↑ error happens here	$\lvert x - \hat{x}_1 \rvert = \frac{9}{16}$	LSB error
☐ $\text{ufixed}_{2(5,3)}\left(\frac{165}{16}\right)$ (use round up)	$\hat{x}_2 = 01010.011$ ↑ error happens here	$\lvert x - \hat{x}_2 \rvert = \frac{5}{16}$	LSB error
☐ $\text{ufixed}_{2(3,5)}\left(\frac{165}{16}\right)$	$\hat{x}_3 = 010.0101$ ↑ error happens here	$\lvert x - \hat{x}_3 \rvert = 8$	MSB error
☐ $\text{ufixed}_{2(3,5)}\left(\frac{165}{16}\right)$	$\hat{x}_4 = 110.0101$	$\lvert x - \hat{x}_4 \rvert = 4$	MSB error

Example 6.1 ...)

- Given these observations, which parameter values are the most appropriate for this example?

- what if, for some reason, we could not choose $\ell = 4$ and $f = 4$? What would by your next favorite option? Why?

- What type of error is "preferrable": LSB errors or MSB errors? Why?

Example 6.2) Let $x = \left(\frac{9965}{64}\right)_{10}$

$$\Rightarrow x = 128 + 16 + 8 + 2 + 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} = (155.703125)_{10}$$

$$\Rightarrow x = 2^7 + 2^4 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-6}$$

$$\Rightarrow x = \frac{2^{13} + 2^{10} + 2^9 + 2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 2^0}{2^6}$$

$$\Rightarrow x = \left(\frac{10011011101101}{1000000}\right)_2$$

$$\Rightarrow x = (10011011.101101)_2$$

- Suppose we want to store this number in $m = 8$ bits. Do the same "optimal" choice of parameters $\ell = 4$ and $f = 4$ from example 6.1 yield an optimal data encoding strategy in this example?
- If not, what are more strategic values of $\ell, f \in \mathbb{Z}$ for this case if we assume we cannot extend our range (i.e. $m = 8$)?

Example 6.2, continued ...)

The first observation we might make is that the economic decision to keep $m = 8$ has immediate mathematical consequences. Since we cannot store $x = \left(\frac{9965}{64}\right)_{10}$ exactly using only 8 bits and thus any 8-bit fixed-point data encoding will necessarily be approximate and inaccurate.

The first thing we should do after we come to this realization is cry. Then, after we recover our sensibilities, we might ask:

- even though any 8-bit fixed point data encoding of $x = \left(\frac{9965}{64}\right)_{10}$ will be inaccurate, what is the best we can do (i.e. which 8 bit encoding will produce an approximation w. the minimal error).

Example 6.2 continued...)

From our work in example 6.1 and using our hard earned intuition, we might guess that $\ell = 8$, $f = 0$ and the round-up scheme does the trick. Let's try this on for size:

$$\hat{X}_1 = \text{ufixed}_{2(8,0)}\left(\frac{9965}{64}\right)$$

$$= \text{ufixed}_{2(8,0)}\left(\underbrace{10\ 0110\ 1110\ 1101}_{\text{raw, uninterpreted 14-bit word}}\right)$$

$$= \underbrace{1001\ 1011}_{\ell = 8} . \underbrace{10111\ 01}_{f = 0}$$

we can't store 14-bits in an 8-bit word so this trailing part will be lost. But, we can round up:

$$\begin{array}{r} 1001\ 1011 \\ +\qquad\quad\ 1 \\ \hline 1001\ 1100 \end{array}$$

$$\approx 1001\ 1100$$

Example 6.2 continued ...)

If we set

$$\hat{x}_1 = 1001\ 1100.0$$ ← Implicit fractional part since $f = 0$

$$= 2^7 + 2^4 + 2^3 + 2^2$$

$$= 128 + 16 + 8 + 4$$

$$= 156 = \frac{9984}{64}$$

$\Rightarrow$ we have absolute error in our approximation of

$$|x - \hat{x}_1| = \left| \frac{9965}{64} - \frac{9984}{64} \right| = \left| \frac{-19}{64} \right|$$

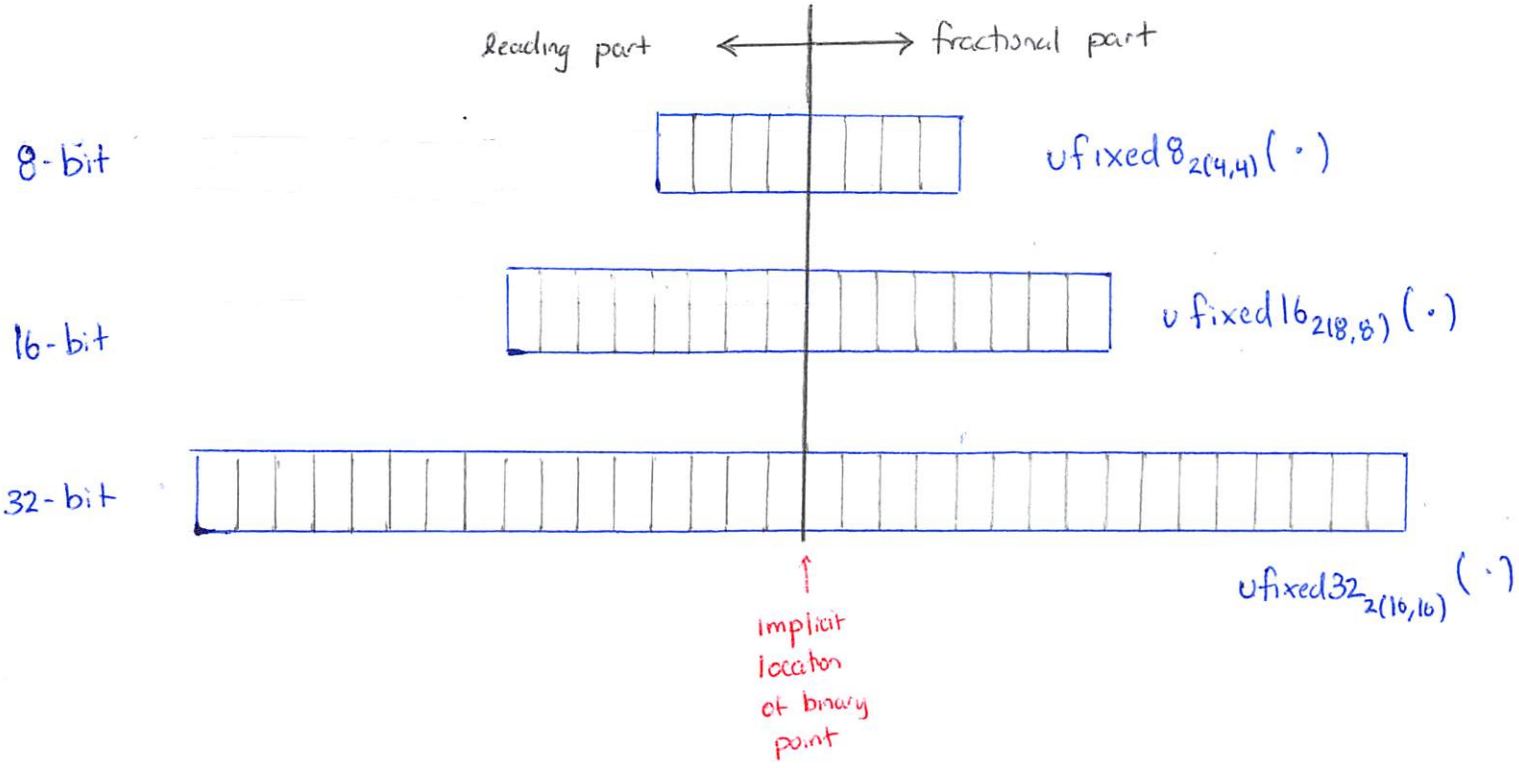
$$= \frac{+19}{64} = 16 + 2 + 1$$

$$= \frac{2^4 + 2^1 + 2^0}{2^6} = 2^{-2} + 2^{-5} + 2^{-6}$$

$$= 0.010011$$

Lesson 6, part 2: Intro to Floating-Point Continued

Based on our work in Lesson 6, part one we saw that for a chosen m-bit, unsigned fixed-point format with $m = 8, 16, 32,$ or 64, we might choose $\ell = \frac{m}{2} = f$ so that exactly half the bits are dedicated to the leading part and half the bits are dedicated to the fractional part of our number. Let's visualize this:



Let's analyze the range of each of these data formats.

$ufixed8_{2(4,4)}(\cdot)$: $x = ufixed8_{2(4,4)}(B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0) = \sum_{k=-4}^{3} b_k \cdot 2^k$

where $b_k = B_{k+4}$

lower-bound : 0000 0000 ← raw, uninterpreted 8-bit word

interpreted value

$\Rightarrow x = ufixed8_{2(4,4)}(0000\ 0000)$

$= (0000.0000)_2$

$= (0)_{10}$

upper-bound : 1111 1111 ← raw, uninterpreted 8-bit word

$\Rightarrow x = ufixed8_{2(4,4)}(1111\ 1111)$

$= (1111.1111)_2$

$= 1\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 1\cdot2^0 + 1\cdot2^{-1} + 1\cdot2^{-2} + 1\cdot2^{-3} + 1\cdot2^{-4}$

$= 8 + 4 + 2 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}$

$= \frac{128 + 64 + 32 + 16 + 8 + 4 + 2 + 1}{16} = \frac{255}{16}$

$\Rightarrow$ upper bound for $ufixed8_{2(4,4)}(1111\ 1111)$ is

$$\frac{255}{16} = \frac{256-1}{16} = \frac{2^8 - 2^0}{2^4} = 2^4 - \frac{1}{2^4} = 2^{\ell} - \frac{1}{2^f}$$

Exercise: Do the same analysis for

A. $ufixed16_{2(8,8)}$ (B)

B. $ufixed32_{2(16,16)}$ (B)

C. $ufixed64_{2(32,32)}$ (B)

Note: there are some really beautiful & powerful features of fixed point format including

A. fast implementation for arithmetic in binary using same circuitry as for integer arithmetic

B. Easily defined signed data classes fixed8, fixed16, etc.-- based on twos complement representation

C. Relatively

We saw in class that the fixed-point format has some significant draw backs though. Two of these include:

A. wasted bits: if we encoded the number $x = 4 + \frac{1}{4}$ in $ufixed16_{2(8,8)}(\cdot)$, we would have the following bit string

0000 0100.0100 0000

look at all these "wasted" bits

B. relative limited range: the range

$$0 \leq x \leq 2^{\ell} - \frac{1}{2^{f}}$$

is decent but falls far short of being versitile enough to encode the entire span of numbers we might want to use in scientific computing applications...

These draw backs are significant enough that they warrent deep thought abaut possible alternative methods to encode $x \in \mathbb{Q}_I$. One very powerful, well-designed, and famous solution to this problem is known as IEEE Floating-Point format. In our work, we will focus heavily on four floating-point formats for signed $x \in \mathbb{Q}_I$ and lightly sample a fifth format:

major focus

- binary 8(·) ← quarter precision "IEEE" floating point
- binary 16(·) ← half precision IEEE floating point
- binary 32(·) ← single precision IEEE floating point
- binary 64(·) ← dobble precision IEEE floating point

light sample

- binary 128(·) ← extended precision IEEE floating point

Foundational Idea for Floating Point Formats

The major inspiration for floating point format is not complicated. In fact, you might be familiar with the idea already and know it as either

- scientific notation or
- exponential notation

These tools allow us to save time by representing numbers optimally

Example 6.3) Suppose we wanted to represent the number

$$X = (0.0009765625)_{10}$$

(leading zeros)

$$= \left(\frac{1}{2^{10}}\right)_{10}$$

$$= (0.0000000001)_2$$

(leading zeros)

Notice how annoying it is to have to write out all the leading zeros.

Example 6.3, continued ...)

- Moreover, these leading zeros don't add any significant "value" to our representation outside of telling us where the radix point should be.

- Based on a clever use of our knowledge of arithmetic and algebra, we can capture the same information much more efficiently by allowing the radix point to float:

$$x = (0.0009765625)_{10}$$

$$= 0000\ 0000\ 0000\ 0000.0009\ 7655\ 2500\ 0000$$

↖ $\text{ufixed32}_{10(16,16)}(\cdot)$ format required to capture exactly

$$= 9.765625 \times 10^{-4}$$

Note: to transform

$$0.0009765625 = 9.765625 \times 10^{-4}$$

take radix point and float it down the digit stream four places to the right

Example 6.3, continued

The math goes like this

$$x = (0.0009765625)_{10}$$

$$= (0.00097\ 65625) \cdot 1$$

$$= (0.00097\ 65625) \cdot 10^{0}$$

$$= (0.00097\ 65625) \cdot 10^{4-4}$$

$$= (0.00097\ 65625) \cdot 10^{4} \cdot 10^{-4}$$

$$= (0.0009765625 \cdot 10^{4}) \cdot 10^{-4}$$

float to the right by four spaces

properly account for shift in exponent notation

$$= 9.765625 \times 10^{-4}$$ ← scientific notation (in decimal)

This down side to fixed point lead to a very tedius & focused development of an alternate much more versitile set of data classes called floating-points

binary 8 ← quarter precision

binary 16 ← half precision

binary 32 ← single precision

binary 64 ← double precision

binary 128 ← extended precision

You already have all the genius you need to understand the main idea behind these formats

$$X = \frac{1}{2^{10}} = 0.000\underbrace{9765625}_{\text{right shift by four places}} \cdot 1$$

$$= 0.0009765625 \cdot 10^{0}$$

$$= 0.0009765625 \cdot 10^{4-4}$$

$$= (0.0009765625) \cdot 10^{4} \cdot 10^{-4}$$

$$= (0.0009765625 \cdot 10^{4}) \times 10^{-4}$$

Example 6.3, continued ...)

Of course, we can do the same type of analyss in binary

$$X = \frac{1}{2^{10}} = (0.0000000001)_2$$

$$= (0.0000000001) \cdot \underbrace{2^0}_{=1}$$

$$= (0.0000000001) \times \underbrace{2^{10} \cdot 2^{-10}}_{=2^0}$$

$$= (0.0000000001 \cdot 2^{10}) \times 2^{-10}$$

$$= 1.0 \times 2^{-10}$$ ← scientific notation (in binary)

Example 6.4) Consider the following $x \in \mathbb{B}_I$

$$x = 2^{-4} + 2^{-7} + 2^{-8}$$

$$= (0.00010011)_2$$

If we represented this in fixed point, we would need 8 = f bits for the fractional part to capture this value exactly. However, let's transform into scientific (floating point) format as we did in Example 6.3:

$$x = (0.00010011) \cdot 2^{4} \cdot 2^{-4}$$

$= 2^0 = 1$

want right shift by four bits

$$= 1.0011 \times 2^{-4}$$

-4 ← exponent value

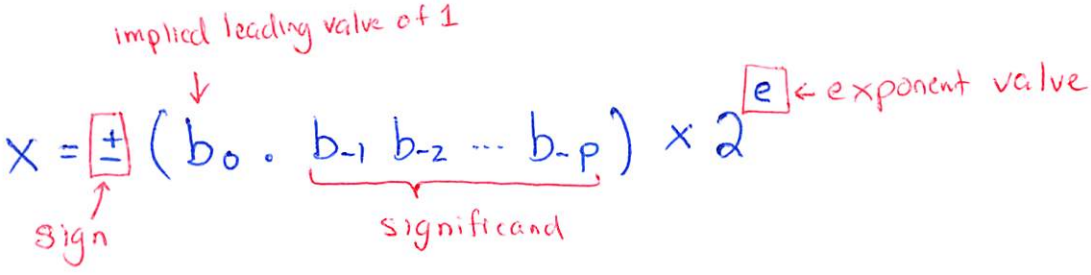
leading part = 1 (normalized)

new fractional part

binary is always base = 2

Normalized floating-point numbers

Mathematically, we can denote the general format of a normalized floating point number as



where $b_0 = 1$ when we have normalized numbers

$$\Rightarrow \quad x = \pm \left(1 + \underbrace{0.b_{-1} b_{-2} \cdots b_{-f}}_{\text{significand}}\right) \times 2^{e}$$

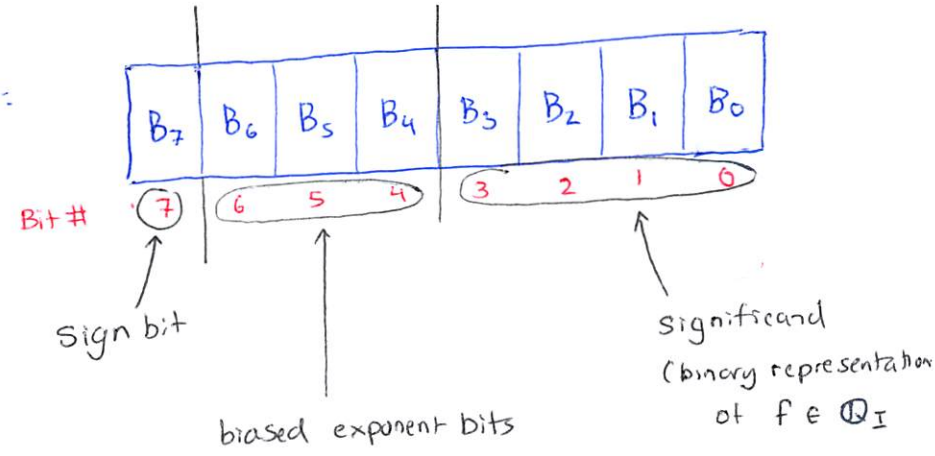
$$= \pm (1 + f) \times 2^{e} \qquad \text{w/} \quad f = 0.b_{-1} b_{-2} \cdots b_{-p}$$

Let's explore how we might store quarter precision floating-point numbers using 8-bits.

Quarter-precision floats $m = 8$-bits & normalized

Let $x = \text{binary8}(B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0)$

Raw, uninterpreted 8-bit word:



Let x be the interpreted value resulting from 8-bit word

sign bit: $B_7 = \begin{cases} 0 & \text{if } x \geq 0 \\ 1 & \text{if } x < 0 \end{cases}$

biased exponent field: $B_6 B_5 B_4 \xrightarrow{\text{mapped to}}$

Example 6.5)

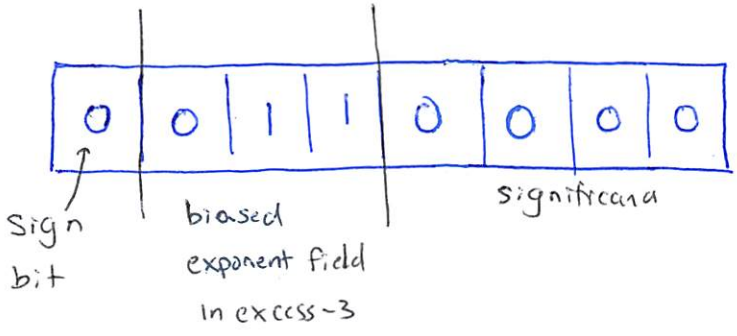
$$X = 1 = 2^0$$

$$= 1.0 \times 2^0$$

$$= +1.\underbrace{0000}_{\text{significant}} \times 2^0$$

$$\Rightarrow \begin{cases} B_7 = 0 \quad \text{since} \quad X \geq 0 \\ f = 0 \leftrightarrow B_3 B_2 B_1 B_0 = 0000 \\ e = (0)_0 \leftrightarrow 011 = B_6 B_5 B_4 \end{cases}$$

$X = (1)_{10} = (1)_2$ $\xrightarrow{\text{stored}}$



$$X = (2)_{10}$$

$$= (10)_2$$

left shift

$$= (10 \cdot 2^{-1}) \cdot 2^{1}$$

sign bit

$$= \boxed{+}\ 1.0000 \times 2^{\boxed{1}}$$

exponent value

significant

Implied normalized leading bit

$$\Rightarrow \quad B_7 = 0 \quad \text{is} \quad X \geq 0$$

$$f = 0 \Rightarrow B_3 B_2 B_1 B_0 = 0000$$

$$e = 1 \Rightarrow B_6 B_5 B_4 = 100$$

Note: unsigned math

$$e = 100 - 011 = 4 - 3 = 1$$

$$u - K = e$$

K-bais

Unsigned integer value of exponent field

$X = (2)_{10} = (10)_2$ Stored →

0	1	0	0	0	0	0	0

$$X = (11)_{10}$$

$$= 8 + 2 + 1$$

$$= 2^3 + 2^1 + 2^0$$

$$= (1011)_2$$

$$= 1.011 \times 2^3$$

$$= \boxed{+}\ 1.0110 \times 2^{\boxed{3}}$$

(implied leading value; sign; significant; ← exponent)

$$\Rightarrow \quad B_7 = 0 \qquad \text{since} \quad X \geq 0$$

$$f = \qquad \Rightarrow \quad B_3 B_2 B_1 B_0 = 0110$$

$$e = 3$$

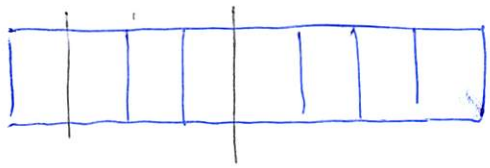
Note: unsigned math

$$e = 110 - 011 = 6 - 3 = 3 \checkmark$$

$$e = u - K \quad \text{where} \quad u = \text{uint3}(B_6 B_5 B_4)$$

$$\Rightarrow u = e + K = (e + 3)_{10}$$

$$= (e + 011)_2$$

$x = (15)_{10}$ ⟶ Stored as 

$= 8 + 4 + 2 + 1$

$= (1111)_2$

$= 1.111 \times 2^3$

Implied leading bit ↓

$= \boxed{+}\ 1.1110 \times 2^{\boxed{3}}$ exponent value

sign bit — significant

$e = 3 = \text{uint3}(B_6 B_5 B_4) - 3$

$\Rightarrow e = 011 = u - 011$

$= u - k$

$\Rightarrow u = e + k$

$= 011 + 011$

$= (110)_2$

$= 6$

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 011 \\ +011 \\ \hline 110 \end{array}$$

sometimes called offset binary

unsigned dec	exponent code = u raw, uninterpreted value for 3-bit exponent field $B_6 B_5 B_4$	exponent value 3-bit, excess-3 interpreted, biased value of e (in decimal)
0	000	-3
1	001	-2
2	010	-1
3	011	0
4	100	1
5	101	2
6	110	3
7	111	4

$$e = u - K$$

$$= uint3(B_6 B_5 B_4) - 3$$

since $K = 3$

(binary 8 has an excess-3 offset)