(CS 367) Floating Point

Prof. Ivan Avramovic

Assignments

- Reading for this class: 2.4.
- Reading on the horizon: 7.
- Project 1 is due at the end of the 3rd week.
  - Friday Sep. 10.
- Quiz is Tue-Thu.
  - 30 minutes online on Blackboard.
  - Quiz 1: Reviews C concepts from 262/222 (memory refs; arrays; bitwise ops).
  - Quiz 2: Integer representation and integer operations.

Fractional Values

- The result of a right shift is still an integer.
  - Dividing 3 by 2 should really give us 1.5. How do we write that?
  - How would we write fractional values in general using binary?

Fractional Values

- The result of a right shift is still an integer.
  - Dividing 3 by 2 should really give us 1.5. How do we write that?
  - How would we write fractional values in general using binary?

  - If right-shift is divide-by-2, what if we keep going?

$3_{10} = 11_2$
$3_{10}/2 = 11_2 >> 1 = 1.1_2$

Binary Point

- Decimal numbers use a *decimal point* as a frame of reference.
- Binary number use a *binary point* for the same reason.

$$123.45_{10} = \boxed{1\times10^{2} + 2\times10^{1} + 3\times10^{0}} + \boxed{4\times10^{-1} + 5\times10^{-2}.}$$

$$101.01_{2} = \boxed{1\times2^{2} + 0\times2^{1} + 1\times2^{0}} + \boxed{0\times2^{-1} + 1\times2^{-2}.}$$

Easy Conversion To a Fraction

How would we convert a number like 1010.0101_2 to a proper fraction?

Easy Conversion to a Fraction

How would we convert a number like 1010.0101_2 to a proper fraction?

$1010.0101_2 = (1010_2) + (0101_2)/2^4$.

$1010_2 = 10_{10}$.

$0101_2 = 5_{10}$.

$2^4 = 16_{10}$.

$1010.0101_2 = 10\frac{5}{16}$.

General formula:

$$x.y_2 = x + \frac{y}{2^n},$$

assuming that y is n digits long (including any leading zeros).

Conversion from a Fraction

How would we convert $3\frac{7}{32}$ to bits?

Conversion from a Fraction

How would we convert $3\,\frac{7}{32}$ to bits?

$3_{10} = 11_2$.
$7_{10} = 111_2$.
$32_{10} = 2^5$.
32 implies 5 bits, so $7/32 = .00111_2$.

$$3\,\frac{7}{32} = 11.00111_2.$$

In general:
$x\,\frac{y}{2^n} = binary(x) + (binary(y) \gg n)$,
assuming that we let the bit shift to take us past the binary point.

Notes About Converting from Fractions

Converting works well if the denominator is a power of 2.

- In class, assume that all problems will use power-of-2 denominators.
- What if it isn't?
- Example: 1/3 in decimal is 0.3333333333.....
- Example: 1/3 in binary is 0.01010101.....
- Some fractions will always produce infinite digit strings.

We can compute this using the standard long division algorithm, in binary.

Converting from a fraction $\frac{x}{y}$ that does not have a power of 2 denominator:

Find a nearby power of 2: $\frac{x}{y} = \frac{z}{2^n}$

The z will have a decimal part, so round it to the nearest integer.
Then, convert like we did before!

Example:
$$2\frac{3}{10} = 2\frac{4.8}{16} \approx 2\frac{5}{16} = 10.0101_2$$

Fractions on Computers

Mathematically, $(x + y) + z = x + (y + z)$.

- On a computer using floating points, this doesn't happen!
- Try: $x = 10^{20}, y = -10^{20}, z = 3.14$.
- The left size would evaluate 3.14, while the right would evaluate 0.

Scientific Notation

1GB = 1,073,741,824 bytes.

- If we say that 1GB = 1.1 billion bytes, we're basically right.
- If we add 1 byte to that, we would still probably say we had 1.1 billion bytes.
- Really we're saying we have about 1.1×10^9 bytes.
  - 1.1 has two digits of *precision*.
  - 10^9 has an *exponent* of 9.
  - The leading 1 digit is a number between 1-9.

Floating Point Numbers

Computers express fractional values using a binary scientific notation.

- Example: $1.0011_2 \times 2^2 = 100.11_2 = 4\frac{3}{4}$.
- Note: in decimal, the leading digit is 1-9; in binary, the leading digit must be 1.

- The standard for *floating point* (FP) numbers is *IEEE Standard 754*.
- All computers use it – without a standard, everyone used their own variant.

IEEE Standard 754

- The standard defines 2 types: float and double.
  - The float is 4 bytes; the double is 8 bytes.
  - We typically should prefer the double in all cases unless size matters.

Newer versions of the standard have defined several additional sizes.

32-bit (single precision) float:

31	23-30	0-22
S	exp	frac

64-bit (double precision) double:

63	52-62	0-51
S	exp	frac

Binary Scientific Notation

- Let's pick any number x and write it as $x = (-1)^S M \times 2^E$.
  - S is the *sign bit*; the $(-1)^S$ shows if a number is positive or negative.
  - M is the *mantissa*; assume it's in the form 1.????.
  - E is the *exponent*, which can be positive, negative, or zero.

- Note: $101.1_2 = 101.1_2 \times 2^0 = 10.11_2 \times 2^1 = 1.011_2 \times 2^2$.

In general: If there are n digits
to the left of the binary point, then
the resulting $E = n - 1$ after shifting.

- We can write all numbers this way, with one(?) notable exception.

Floating Point Representation

Suppose $x = (-1)^S M \times 2^E$.

exp is computed from E.

7	4-6	0-3
S	exp	frac

This example is 8 bits. Our actual `float` type has more bits, but the format still looks similar to this.

S is 0 or 1 depending on whether x is positive or negative

frac is computed from M.

- Most of our numbers will be expressed in *normalized* form.
  - We'll talk about what it means and later, as well as what the exceptions are.

Floating Point Representation

Suppose $\mathrm{x} = (-1)^{s} M \times 2^{E}$.

7	4-6	0-3
S	exp	frac

- We know that M looks like 1.?????
  - For space purposes, ignore the leading 1.
  - Encode the digits after the point in $frac$.

The leading digit is always a 1, so it would be a waste of space to include it in the encoding.

Example:
If M = 1.101 and there are 4 bits of *frac*,
then *frac* = 1010.

Floating Point Representation

Suppose $\mathrm{x} = (-1)^s M \times 2^E$.

7	4-6	0-3
S	exp	frac

- How would we compute *exp* from the value E?
  - Our example FP type above uses a 3 bit exponent.
  - The bottom (000) and top (111) values of exp are reserved for special reasons.
  - The rest of exp is mapped linearly.
    - The smallest normalized E value corresponds to the smallest normalized exp (001).
    - The largest normalized E value corresponds to the largest normalized exp (110).

This same general rule is true no matter how many bits are in *exp*: the smallest and largest values are reserved, and the remaining values are mapped linearly from smallest to largest.

Floating Point Representation

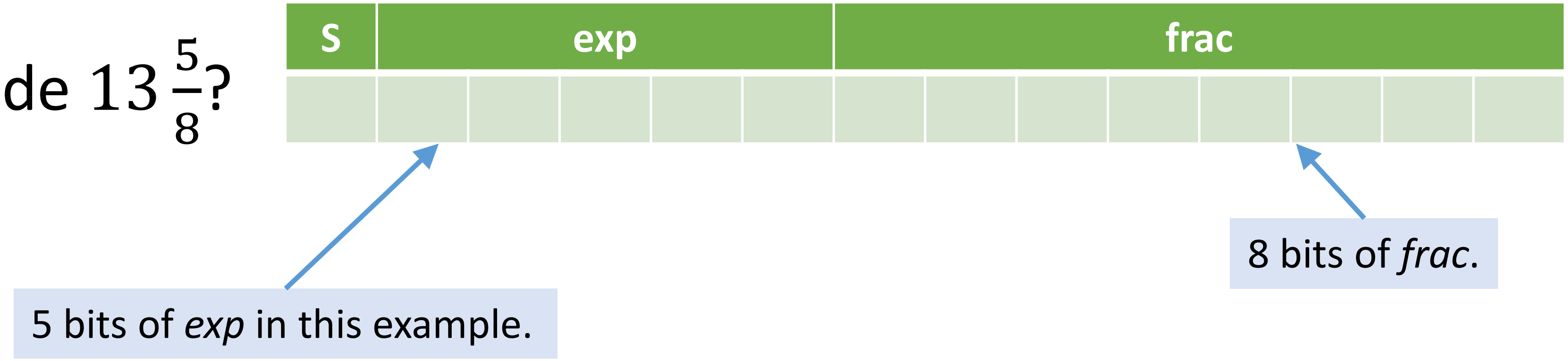
Suppose $\mathrm{x} = (-1)^s M \times 2^E$.

7	4-6	0-3
S	exp	frac

- What is exp? Use the following definitions:
  - Let e be the number of bits in exp (in this example, e = 3).
  - Let $bias = 2^{e-1} - 1$. In this example, $bias$ = 3.
  - Let exp = E + $bias$.
  - If $exp \leq 0$ or $exp \geq \sim 0$ then we have to treat this as a special case.

Floating Point Encoding

How would we encode $13\frac{5}{8}$?



Floating Point Encoding

S	exp					frac							

How would we encode $13\,\frac{5}{8}$?

- $13 = 1101_2$, $5 = 101_2$.
- $13\,\frac{5}{8} = 1101.101_2$.
- $1101.101_2 = 1.101101_2 \times 2^3$.
- $M = 1.101101_2$.
- $E = 3$.
- $S = 0$ (positive number).

Floating Point Encoding

S	exp					frac							
0	1	0	0	1	0	1	0	1	1	0	1	0	0

How would we encode $13\frac{5}{8}$?

- 13 = 1101_2, 5 = 101_2.
- $13\frac{5}{8} = 1101.101_2$.
- $1101.101_2 = 1.101101_2 \times 2^3 = (-1)^S \times M \times 2^E$.
- $M = 1.101101_2$.
- $E = 3$.
- $S = 0$ (positive number).
- $e = 5$ (bits).
- $bias = 2^{e-1}-1 = 15$.

$frac = 10110100_2$.

$exp = E + bias = 18 = 10010_2$.

Floating Point Encoding

How would we encode $-\frac{127}{256}$?

S	exp					frac							

Floating Point Encoding

How would we encode $-\frac{127}{256}$?

S	exp					Frac							
0	0	1	1	0	1	1	1	1	1	1	1	0	0

- $0 = 0_2$, $127 = 1111111_2$.
- $-\frac{127}{256} = -0.01111111_2$.
- $-0.01111111_2 = -1.111111_2 \times 2^{-2} = (-1)^S \times M \times 2^E$.
- $M = 1.111111_2$. $frac = 11111100_2$.
- $E = -2$. $exp = E + bias = 13 = 01101_2$.
- $S = 1$ (negative number).
- $e = 5$ (bits).
- $bias = 2^{e-1}-1 = 15$.

Floating Point Interpretation

Which number is this?

S	exp					frac							
0	1	0	1	0	0	0	0	1	0	0	0	0	0

Floating Point Interpretation

S	exp					frac							
0	1	0	1	0	0	0	0	1	0	0	0	0	0

Which number is this?

- S = 0.
- $frac$ = 0010000_2.
  - M = 1.001_2.
- exp = 10100_2 = 20 = $E + bias$.
  - e = 5 (bits).
  - $bias$ = 15.
  - $E = exp - bias$ = 5.
- The number is $1.001_2 \times 2^5$ = 100100_2. = 36.

Floating Point Interpretation

Which number is this?

s	exp					frac							
1	0	0	1	1	1	1	0	1	0	1	0	0	0

Floating Point Interpretation

S	exp					frac							
1	0	0	1	1	1	1	0	1	0	1	0	0	0

Which number is this?

- S = 1.
- $frac$ = 1010100_2.
  - M = 1.10101_2.
- exp = 111_2 = 7 = $E + bias$.
  - e = 5 (bits).
  - $bias$ = 15.
  - $E = exp - bias$ = -8.
- The number is $-1.10101_2 \times 2^{-8}$ = -0.0000000110101_2. = 53/ 2^{13}.

Other floating point standards?

IEEE 754 defines float and double.

- Since the original version of the standard, it has been revised.
- Now includes 2-byte, float, double, 16-byte, and 32-byte types.
- Includes types which are specialized for decimal (rather than binary).

Interesting new takes on floating points.

- Posits (also known as unums), circa 2017.

Formats optimized for neural nets.

- High precision is not needed, small size is.
- Google's bfloat16, circa 2018.
- Facebook's ELMA-based floating point, circa 2018.

Smallest and Largest Normalized Values?

What is the smallest and largest normalized floating point magnitude?

- We know that the normalized *exp* goes from 00…001 to 11….110.
- That’s the same as 1 to $(2^{e-1}-1)\times 2$.

S	exp					frac							

Smallest and Largest Normalized Values?

What is the smallest and largest normalized floating point magnitude?

- We know that the normalized *exp* goes from 00…001 to 11….110.
- That's the same as 1 to $(2^{e\text{-}1}\text{-}1)\times 2$.
- bias = $2^{e\text{-}1}\text{-}1$.

E = *exp - bias*

Largest possible M is 1.11…11.

- Largest:
  - $E = 2(2^{e\text{-}1}\text{-}1)\text{-}(2^{e\text{-}1}\text{-}1) = 2^{e\text{-}1}\text{-}1,\qquad M = 2 - 2^{-f}.$
- Smallest:
  - $E = 1 - (2^{e\text{-}1} - 1) = 2 - 2^{e\text{-}1},\qquad M = 1.$

Smallest possible M is 1.00…00.

S	exp					frac								

Outside of Normalized Range

What happens if the numbers go above the max?

- If numbers go above, we still allow *infinity* and *not-a-number* (NaN).

What happens if the numbers go below the min?

- If they go below the min, we switch to *denormalized* values.
  - Denormalized values are scaled fixed point numbers.
  - Denormalized values sacrifice precision for increased range.
  - Denormalized values pick up where normalized values leave off.

Example:
if the smallest normalized is 0.010000_2,
then the largest denormalized is 0.001111_2.

Denormalized Values

In a denormalized encoding:

- exp = 00…000 = 0, no matter what.
- $E = 1 - bias = 1 - (2^{e\text{-}1} - 1) = 2 - 2^{e\text{-}1}$.
- $M = 0.frac = frac\ /\ 2^{f}$.

Same as the min normalized exponent, i.e. it stops going lower.

0.*frac* instead of 1.*frac* for denormalized values.

s	exp					frac							
0	0	0	0	0	0	0	0	0	1	0	0	0	0

Always 0's for denormalized values.

Denormalized Values

In a denormalized encoding:

- exp = 00…000 = 0, no matter what.
- $E = 1 - bias = 1 - (2^{e\text{-}1} - 1) = 2 - 2^{e\text{-}1}$, same as the min normalized exponent.
- $M = 0.frac = frac\ /\ 2^f$.

S	exp					frac							
0	0	0	0	0	0	0	0	0	1	0	0	0	0

- Example: $E = 2 - 2^{5\text{-}1} = \text{-}14$, $M = 16/2^8 = 1/16 = 0.0001_2$, $S = 0$.
  - Value = $+0.0001_2 \times 2^{-14} = 1 \times 2^{-18}$.

Floating Point Ranges, so far

We can use normalized floating point encodings for most numbers.

- Includes both very large and very small numbers.
- Includes positives and negatives.
- Does not include zero.

Denormalized encodings work for smaller than normalized values.

- Includes positive and negative values.
- A value should be denormalized if its *calculated exp is zero or below*.

How would we represent zero?

- Hint: will denormalized encodings help us at all?

Special Values

Zero is expressed using denormalized values.

- In fact, all-zeros represents the number 0 already.

Due to the sign bit, -0 exists – but it is numerically equal to 0.

If we go above the normalized (exp = all 1's), we get special values.

- Special values are either infinity or NaN.
- If $frac$ = all 0's, the number is infinity.
- For any other $frac$ value, the number is NaN.

$+\infty$ or $-\infty$, depending on the value of S.

Addition

Suppose we want to add $(-1)^{S_1} M_1 \times 2^{E_1} + (-1)^{S_2} M_2 \times 2^{E_2}$.

Example:
$1.10000 \times 2^1 + 1.01100 \times 2^4$

Addition

Suppose we want to add $(-1)^{S_1} M_1 \times 2^{E_1} + (-1)^{S_2} M_2 \times 2^{E_2}$.

- First, make the exponents match.
  - Pick the larger exponent.
    - If $E_1 > E_2$, use E_1 as the reference.
    - If $E_1 < E_2$, use E_2 as the reference.
    - If $E_1 = E_2$, we're done with this part!
    - For the sake of example, let's assume that E_1 is larger.
  - Right shift the M with the smaller E to get the E's to match.
    - For example, if $E_1 > E_2$, then shift M_2 to get M_2 >> (E_1-E_2).
    - Note that $(-1)^{S_2}(M_2 \gg (E_1 - E_2)) \times 2^{E_1} = (-1)^{S_2} M_2 \times 2^{E_2}$.

Example:
$1.10000 \times 2^1 + 1.01100 \times 2^4$

Since $1 < 4$ (alternately, $2^1 < 2^4$),
E_2 is our reference:
$1.10000 \times 2^1 = 0.00110 \times 2^4$

Our sum is thus equivalent to:
$0.00110 \times 2^4 + 1.01100 \times 2^4$

Addition

Suppose we want to add $(-1)^{S_1} M_1 \times 2^{E_1} + (-1)^{S_2} M_2 \times 2^{E_2}$.

- Second, add the two numbers since the exponents are now the same.
  - After adding, shift to the right if necessary to get a normalized value.
  - Re-encode the result as a new floating point value.

Note: if one of the numbers were negative, we would subtract instead of add. If both were negative, we would still add, but the result would be negative.

Example:
$1.10000 \times 2^1 + 1.01100 \times 2^4$
$= 0.00110 \times 2^4 + 1.01100 \times 2^4$
$= (0.00110 + 1.01100) \times 2^4$

$$
\begin{array}{rl}
0.00110 & \times 2^4 \\
+\ 1.01101 & \times 2^4 \\
\hline
1.10011 & \times 2^4
\end{array}
$$

Multiplication

Suppose we want to multiply $(-1)^{S_1} M_1 \times 2^{E_1} \times (-1)^{S_2} M_2 \times 2^{E_2}$.

Example:
$-1.10000 \times 2^1 \times 1.01100 \times 2^4$

Multiplication

Suppose we want to multiply $(-1)^{S_1} M_1 \times 2^{E_1} \times (-1)^{S_2} M_2 \times 2^{E_2}$.

$$= (-1)^{S_1}(-1)^{S_2} M_1 \times M_2 \times 2^{E_1} 2^{E_2}$$

$$= (-1)^{S_1+S_2}(M_1 \times M_2) \times 2^{E_1+E_2}$$

$$= (-1)^{S} M \times 2^{E}$$

$E = E_1 + E_2$

$S = S_1 \wedge S_2$

$M = M_1 * M_2$

Example:

$-\,1.10000 \times 2^1 \times 1.01100 \times 2^4$

$= (-1)^1(1.10000 \times 1.01100) \times 2^{1+4}$

$= -\,10.0001 \times 2^5$

$= -\,1.00001 \times 2^6$

Shift the new value to get a normalized result, if necessary.

Rounding

Our arithmetic may lead us to create a number which has too many bits to fit into our data type.

- We have to find a way to drop extra bits.
- Simplest solution:
  - Drop the excess bits/floor (e.g. $101.1011_2 \rightarrow 101_2$).
  - Ceiling (e.g. $101.1011_2 \rightarrow 110_2$).

We must decide ahead of time how many bits of precision we want to keep.

For example, if our goal is to truncate $\boxed{1.01}101$ at $1/4^{ths}$ place, we would get the result 1.01.

Rounding

Our arithmetic may lead us to create a number which has too many bits to fit into our data type.

- In practice, we will actually round the number:
  - Decide on the precision we want.
  - Find the largest number ≤ our actual number that fits into our precision.
  - Find the smallest number ≥ our actual number that fits into our precision.
  - Pick which of the two numbers is closer to our actual number.
    - Check whether the "left over" bits are greater than or less than the halfway point, 0.1.

Note that $0.1_2 = ½$.

Example:
1.01101011, rounded to the $1/8^{\text{ths}}$.

1.01101011

1.100

1.011

Are the remaining bits, 0.01011, larger or smaller than 0.1?
0.01011 < 0.1, so round down.

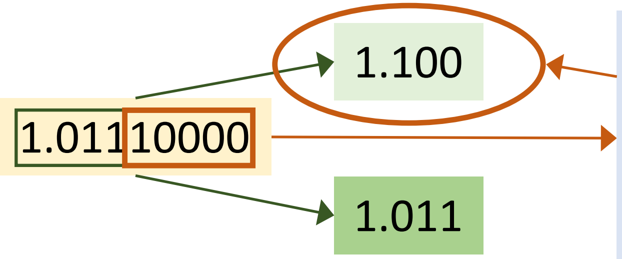
- What if it is exactly in the middle?

Rounding

Our arithmetic may lead us to create a number which has too many bits to fit into our data type.

- What if our actual number is exactly in between our two rounding options?
  - Use a semi-arbitrary scheme for the case when the “left over” bits = 0.1.
  - One of the two options will always end on an even bit – pick that one!
    - Statistically, we will go up half the time, and down half the time.

Example:
1.01110000, rounded to the 1/8ths.



Are the remaining bits, 0.10000, larger or smaller than 0.1?
0.10000 = 0.1, so pick the option which ends on an even bit.

Floating Point Review: Rounding

What do we get when we round to the nearest 1/4th?

- $101.01010 \approx$
- $110.11100 \approx$
- $1010.001 \approx$
- $1011.10110001 \approx$

Floating Point Review: Rounding

What do we get when we round to the nearest 1/4th?

- $101.01010 \approx 101.01$ (because $.010 < ½$, so we round down)
- $110.11100 \approx 111.00$ (because $.100 = ½$, so we round up to the nearest even)
- $1010.001 \approx 1010.00$ (because $.1 = ½$, so we round down to the nearest even)
- $1011.10110001 \approx 1011.11$ (because $.110001 > ½$, so we round up)

Floating Point Review: Addition

What is the result of the following sums?

- $1.0110*2^8 + 1.1001*2^7 =$

- $1.1100*2^{-1} + 1.0010*2^2 =$

Floating Point Review: Addition

What is the result of the following sums?

- $1.0110*2^8 + 1.1001*2^7 = 1.0110*2^8 + 0.1100*2^8$
  $= 10.0010*2^8$
  $= 1.0001*2^9$

- $1.1100*2^{-1} + 1.0010*2^2 = 0.0100*2^2 + 1.0010*2^2$
  $= 1.0110*2^2$

Floating Point Review: Multiplication

- What is the result of the following product?
  - $(-1.011 * 2^{8}) * (1.000 * 2^{-3}) =$

Floating Point Review: Multiplication

- What is the result of the following product?

  - $(-1.011 * 2^8) * (1.000 * 2^{-3}) = -(1.011 * 1.000) * (2^8 * 2^{-3})$

    $= -1.011 * 2^{8-3}$

    $= -1.011 * 2^5$