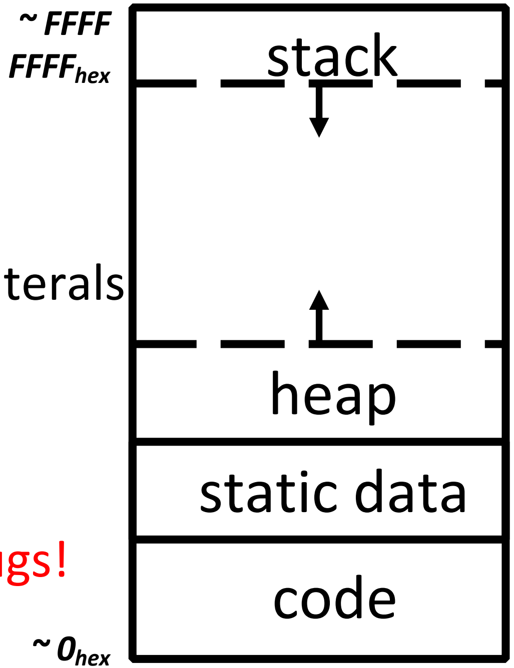
CS 61C Great Ideas in Computer Architecture

Floating Point

Instructor: Jenny Song

Review

- C Memory Layout
  - **Stack:** local variables (grows & shrinks in LIFO manner)
  - **Static Data:** global and string literals
  - **Code:** copy of machine code
  - **Heap:** dynamic storage using `malloc` and `free`
    The source of most memory bugs!
- Common Memory Problems



OS prevents accesses between stack and heap (via virtual memory)

Agenda

- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- Bonus: FP Conversion Practice

Number Representation Revisited

- Given 32 bits (a *word*), what can we represent so far?
  - Signed and Unsigned Integers
  - 4 Characters (ASCII)
  - Instructions & Addresses
- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^{23})
  - Very small numbers (e.g. 6.626×10^{-34})
  - Special numbers (e.g. ∞, NaN)

Floating Point

Reasoning about Fractions

Big Idea: Why can't we represent fractions? Because our bits all represent nonnegative powers of 2.

Example 6-bit representation: 0b xx xxxx

2^5 2^4 2^3 2^2 2^1 2^0

Example: $10\ 1010_{two} = 1\times2^5 + 1\times2^3 + 1\times2^1 = 42_{ten}$

- The lowest power of 2 is 2^0, so the ***smallest*** difference between any two numbers is $2^0 = 1$.

Example: $10\ 1011_{two} = 42+1 = 43_{ten}$

Representation of Fractions

Look at decimal(base 10) first:
Decimal "point" signifies boundary between integer and fraction parts.

Example 6-digit representation: $\texttt{xx.xxxx}$

10^1 10^0 10^{-1} 10^{-2} 10^{-3} 10^{-4}

Example: $25.2406_{ten} = 2x10^1 + 5x10^0 + 2x10^{-1} + 4x10^{-2} + 6x10^{-4}$

- Not much range: : 0 to 99.9999
- but lots of "precision": 6 significant figures, lowest power is 10^{-4}

Representation of Fractions

New Idea: Introduce a fixed "Binary Point" that signifies boundary between negative & nonnegative powers:

Example 6-bit representation: 0b xx.xxxx

2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

Example: $10.1010_{two} = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{ten}$

- The lowest power of 2 is 2^{-4}, so the ***smallest*** difference between any two numbers is $2^{-4} = 1/16$.

- Binary point numbers that match the 6-bit format above range from 0 (00.0000_{two}) to 3.9375 (11.1111_{two})

Scientific Notation (Decimal)

significand exponent

$$2.625_{ten} \times 10^{23}$$

decimal point radix (base)

- *Normalized form*: exactly one digit (non-zero) to left of decimal point (the point "floats" to be in the standard position)
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

Scientific Notation (Binary)

significand

exponent

$$1.0101_{two} \times 2^{1}$$

binary point

radix (base)

- Computer arithmetic that supports this called floating point due to the "floating" of the binary point
  - Declare such variable in C as `float`

Translating To and From Scientific Notation

- Consider the number $1.011_{two} \times 2^4$
- To convert to ordinary number, shift the decimal to the right by 4
  - Result: $10110_{two} = 22_{ten}$
- For negative exponents, shift decimal to the left
  - $1.011_{two} \times 2^{-2} \Rightarrow 0.01011_{two} = 0.34375_{ten}$
- Go from ordinary number to scientific notation by shifting until in *normalized* form
  - $1101.001_{two} \Rightarrow 1.101001_{two} \times 2^3$

“Father” of Floating Point Standard

IEEE Standard 754 for Binary Floating-Point Arithmetic

1989
ACM Turing Award Winner!

Prof. Kahan
Prof. Emeritus
UC Berkeley

www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Goals for IEEE 754 Floating Point Standard

- Standard arithmetic for reals for all computers
  - Important because computer representation of real numbers is approximate. Want same results on all computers
- Keep as much precision as possible
- Help programmer with errors in real arithmetic
  - +∞, -∞, Not-A-Number (NaN), exponent overflow, exponent underflow, +/- zero
- Keep encoding that is somewhat compatible with two's complement
  - E.g., +0 in Fl. Pt. is 0 in two's complement
  - Make it possible to sort without needing to do floating-point comparisons

Floating Point Encoding: Single Precision

- Use normalized, Base 2 scientific notation:

  $+1.xxx...x_{two} \times 2^{yyy...y_{two}}$

- Split a 32-bit word into 3 fields:

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	**8 bits**	**23 bits**

  – **S** represents Sign (1 is negative, 0 positive)

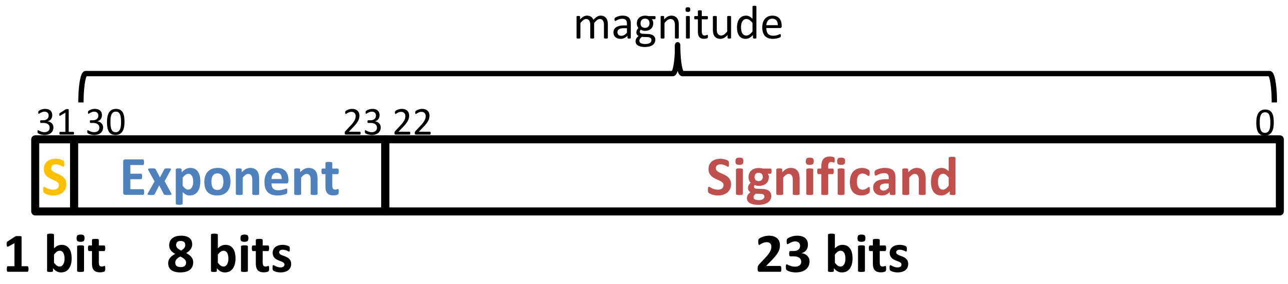
  – **Exponent** represents y's

  – **Significand** represents x's

  – Key Idea: More like Sign & Magnitude

The Exponent Field

- Why use biased notation for the exponent?
  - Remember that we want floating point numbers to look small when their actual value is small
    - We don't like how in 2's complement, -1 looks bigger than 0. Bias notation preserves the linearity of value
- Recall that only the first bit denotes sign
  - Thus, floating point resembles sign and magnitude



The Exponent Field

- Use biased notation but with *bias* of -127
  - Read exponent field as unsigned, add the bias (+ (-127) = -127) to get the actual exponent
  - Exponent Field: 0 (00000000_{two}) to 255 (11111111_{two})
  - Actual exponent: -127 (00000000_{two}) to 128 (11111111_{two})
- To encode in biased notation, subtract the bias (-(-127)=+127) then encode in unsigned:
  - If we had 2^1, exp = 1 => 128 => 10000000_{two}
  - 2^{127}: exp = 127 => 254 => 11111110_{two}

The Exponent Field

∞, NaN

Getting closer to zero

Zero

Decimal Exponent	Biased Notation	Decimal Value of Biased Notation
For infinities	11111111	255
127	11111110	254
...	...	...
2	10000001	129
1	10000000	128
0	01111111	127
-1	01111110	126
-2	01111101	125
...	...	...
-126	00000001	1
For Denorms	00000000	0

The Significand Field

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$$(-1)^{S} \times \boxed{(1\,.\,\text{Significand})} \times 2^{(\text{Exponent}-127)}$$

- What does this mean?
  - Think of it as: (1 + Value of Significand)
  - Since the Significand represents all the negative powers of 2, its total value is always < 1
  - Example: $1.0101_{two} = 1 + 2^{-2} + 2^{-4} = 1.3125$

Floating Point Encoding

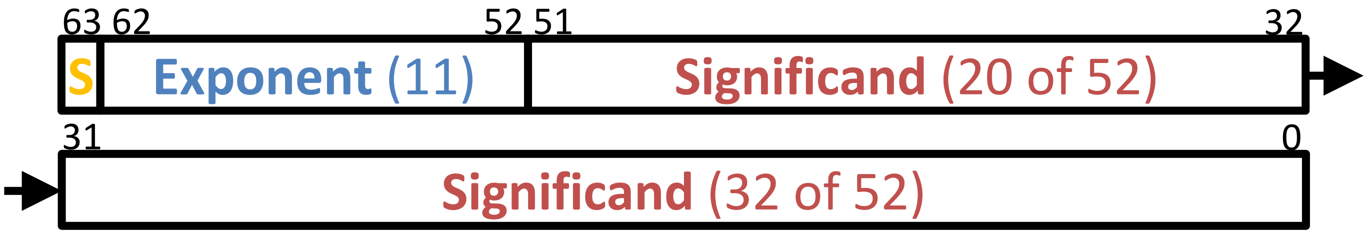
31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$$(-1)^{S} \times (1\ .\ \text{Significand}) \times 2^{(\text{Exponent}-127)}$$

- Note the implicit 1 in front of the Significand
  - Ex: $0011\ 1111\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000_{two}$
    - $(-1)^{0} \times (1\ .\ 1_{two}) \times 2^{(127-127)} = (1\ .\ 1_{two}) \times 2^{(0)}$
    - $1.1_{two} = 1*2^{0}+1*2^{-1}=1.5_{ten}$, NOT $0.1_{two} = 0.5_{ten}$
  - Gives us an extra bit of precision

Double Precision FP Encoding

- Next multiple of word size (64 bits)



- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Exponent bias is 2^{10}-1 = 1023
  - Primary advantage is greater precision due to larger Significand

Agenda

- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- Bonus: FP Conversion Practice

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	?	?
0	?	?
1-254	anything	± fl. pt
255	?	?
255	?	?

Representing Zero

- But wait… what happened to zero?
  - Using standard encoding 0x00000000 is $1.0 \times 2^{-127} \neq 0$
    - All because of that dang implicit 1
  - *Special case:* Exp and Significand all zeros = 0
  - Two zeros! But at least 0x00000000 = 0 like integers

	31	30 … 23	22 … 0
+0	0	00000000	00000000000000000000000
	sign	exponent	significand

	31	30 … 23	22 … 0
-0	1	00000000	00000000000000000000000
	sign	exponent	significand

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	?	?
1-254	anything	± fl. pt
255	?	?
255	?	?

Representing ± ∞

- Division by zero
  - infinity is a number!
  - okay to do further comparison eg. x/0 > y
- Representation
  - Max **exponent** = 255
  - all zero **significand**

	31	30 … 23	22 … 0
+ ∞	0	11111111	00000000000000000000000
	sign	exponent	significand

	31	30 … 23	22 … 0
- ∞	1	11111111	00000000000000000000000
	sign	exponent	significand

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	?
1-254	anything	± fl. pt
255	0	± ∞
255	non-zero	?

Representing NaN

- 0/0, sqrt(-4), ∞–∞ ?
  - Useful for debugging
  - Op(NaN, some number) = NaN
- Representation
  - Max **exponent** = 255
  - non-zero **significand**

	31	30 … 23	22 … 0
+ NaN	0	**11111111**	**Non-zero**
	sign	**exponent**	**significand**

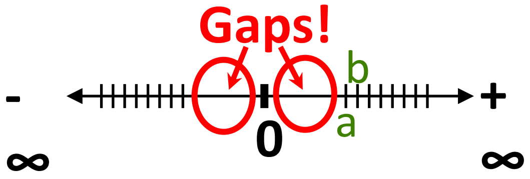
	31	30 … 23	22 … 0
- NaN	1	**11111111**	**Non-zero**
	sign	**exponent**	**significand**

Floating Point Numbers Summary

Exponent	Significand	Meaning
0	0	± 0
0	non-zero	?
1-254	anything	± Norm fl. pt
255	0	± ∞
255	non-zero	NaN

Representing Very Small Numbers

- What are the normal numbers closest to 0? (here, normal means the exponent is **nonzero**)
  - a = $1.0...00_{two} \times 2^{1-127} = (1+0) \times 2^{-126} = 2^{-126}$
  - b = $1.0...01_{two} \times 2^{1-127} = (1+2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149}$

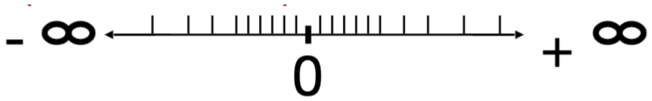


  - The gap between 0 and a is 2^{-126}
  - The gap between a and b is 2^{-149}
  - We want to represent numbers between 0 and a
    - How? The implicit 1 forces the 2^{-126} term to stay :(
    - Solution: Take out the implicit 1!
  - *Special case:* Exp = 0, Significand ≠ 0 are denorm numbers

Denorm Numbers

- Short for "denormalized numbers"
  - No leading 1
  - Careful! Implicit exponent = -126 when Exp = 0x00 (intuitive reason: the "binary point" moves one more bit to the left of the leading bit)
- Now what do the gaps look like?
  - Smallest denorm: $\pm\ 0.0\ldots01_{two} \times 2^{-126} = \pm\ 2^{-149}$
  - Largest denorm: $\pm\ 0.1\ldots1_{two} \times 2^{-126} = \pm\ (2^{-126} - 2^{-149})$
  - Smallest norm: $\pm\ 1.0\ldots0_{two} \times 2^{-126} = \pm\ 2^{-126}$

So much closer to 0



No uneven gap! Increments by 2^{-149}

Floating Point Numbers Summary

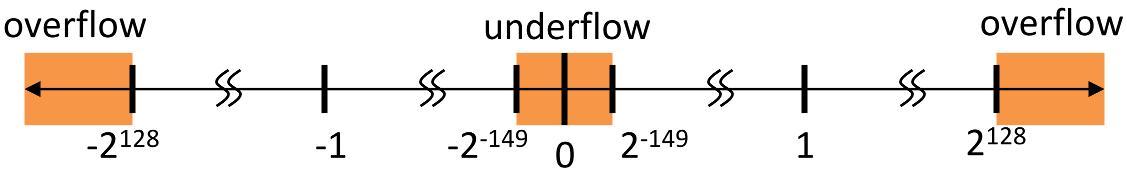
Exponent	Significand	Meaning
0	0	± 0
0	non-zero	± Denorm fl pt.
1-254	anything	± Norm fl. pt
255	0	± ∞
255	non-zero	NaN

Agenda

- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- Bonus: FP Conversion Practice

Floating Point Limitations (1/2)

- What if result `x` is too large? (`abs(x)` > 2^{128})
  - Overflow: Exponent is larger than can be represented
- What if result `x` too small? (0 < `abs(x)` < 2^{-149})
  - Underflow: Negative exponent is larger than can be represented

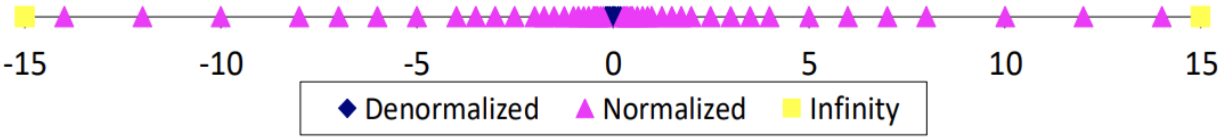


- What if result runs off the end of the Significand?
  - Rounding occurs and can lead to unexpected results
  - FP has different *rounding modes*

Floating Point Gaps

- Does adding 0x00000001 always add the same value to the floating point number?
- NO—it's value depends on the exponent field
- ex: $1.0_{two} \times 2^2 = 4$ $\quad$ +2 $\quad$ $1.0_{two} \times 2^3 = 8$ $\quad$ +4
  $1.1_{two} \times 2^2 = 6$ $\quad\quad\quad$ $1.1_{two} \times 2^3 = 12$
- Thus floating points are quite different from the number representations you've learned so far

❖ Distribution of values is denser toward zero



Floating Point Limitations (2/2)

- FP addition is NOT associative!
  - You can find Big and Small numbers such that: Small + Big + Small ≠ Small + Small + Big
  - This is due to *rounding* errors: FP *approximates* results because it only has 23 bits for Significand
- Despite being seemingly "more accurate," FP cannot represent all integers
  - e.g. $2^{24} + 1 = 16777216$ (fp) 16777217 (actual)

Question:

Let FP(1,2) = # of floats between 1 and 2
Let FP(2,3) = # of floats between 2 and 3

Which of the following statements is true?

Hint: Try representing the numbers in FP

(A)	**FP(1,2) > FP(2,3)**
(B)	**FP(1,2) = FP(2,3)**
(C)	**FP(1,2) < FP(2,3)**
(D)	**It depends**

Question:

Let FP(1,2) = # of floats between 1 and 2
Let FP(2,3) = # of floats between 2 and 3

Which of the following statements is true?

Hint: Try representing the numbers in FP

(A) FP(1,2) > FP(2,3)
(B) FP(1,2) = FP(2,3)
(C) FP(1,2) < FP(2,3)
(D) It depends

$1 = 1.0 \times 2^0$
$2 = 1.0 \times 2^1$
$3 = 1.1 \times 2^1$

$FP(1,2) \approx 2^{23}$, $FP(2,3) \approx 2^{22}$

Question: Suppose we have the following floats in C:

Big = 2^{60}, Tiny = 2^{-15}, BigNeg = -Big

What will the following conditionals evaluate to?

1) (Big * Tiny) * BigNeg == (Big * BigNeg) * Tiny
2) (Big + Tiny) + BigNeg == (Big + BigNeg) + Tiny

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

In general:

(1) is TRUE as long as Big * BigNeg doesn't overflow.

(2) evaluates to 0 != Tiny, which is FALSE as long as Tiny is at least 2^{24} times smaller than Big.

Summary

- Floating point approximates real numbers:
  - Largest magnitude: $2^{128} - 2^{104}$ (Exp = 0xFE)
    Smallest magnitude: 2^{-149} (denorm)
  - Also has encodings for 0, ±∞, NaN
- Floating point has some limitations:
  - Overflow, underflow, rounding
  - Gaps and distribution of values
  - Associativity

Agenda

- Floating Point
- Floating Point Special Cases
- Floating Point Limitations
- Bonus: FP Conversion Practice

Example: Convert FP to Decimal

0	**0110 1000**	**101 0101 0100 0011 0100 0010**

- Sign: 0 means positive
- Exponent:
  - $0110\ 1000_{two} = 104_{ten}$
  - Bias adjustment: 104 - 127 = -23
- Significand:

  1 . 10101010100001101000010

  $= 1 + 1x2^{-1} + 0x2^{-2} + 1x2^{-3} + 0x2^{-4} + 1x2^{-5} + \ldots$

  $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22}$

  $= 1.0 + 0.666115$
- Represents: $1.666115_{ten} \times 2^{-23} \approx 1.986 * 10^{-7}$

Example: Scientific Notation to FP

-2.340625×10^{1}

1. Denormalize: -23.40625
2. Convert integer part:

   $23 = 16 + 4 + 2 + 1 = 10111_{two}$

3. Convert fractional part:

   $.40625 = .25 + .125 + .03125 = 2^{-2}+2^{-3}+2^{-5} = 0.01101_{two}$

4. Put parts together and normalize:

   $10111.01101 = 1.011101101 \times 2^{4}$

5. Convert exponent: $4 + 127 = 10000011_{two}$
6. | 1 | 1000 0011 | 011 1011 0100 0000 0000 0000 |
   |---|---|---|

n	2^n
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625

Bonus

The Following slides include detailed steps of Floating Point conversions, which will be helpful for homework assignment. Please read on your own.

Converting From Hex and Decimal

Convert 0x40600000 to decimal

1 bit for sign, 8 bits for exponent, 23 bits for significand, bias of -127

Step 1: Convert to Binary

0x40600000 = 0100 0000 0110 0000 0000 0000 0000 0000

Step 2: Split Bits Up

0100 0000 0110 0000 0000 0000 0000 0000

Sign	Exponent	Significand
0	100 0000 0	110 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits

Step 3: Check If Norm/Denorm

0100 0000 0110 0000 0000 0000 0000 0000

Sign	Exponent	Significand
0	100 0000 0	110 0000 0000 0000 0000 0000
1 bit	8 bits	23 bits

Exponent is not 00000000, so normalized!

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = 127, 1.significand = 1.11 Ignore trailing 0's

NOTE: In the context of this formula,
Bias = 127

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = 127, 1.significand = 1.11 Ignore trailing 0's

$(-1)^0 * 2^{128 - 127} * 1.11_2 = 2 * 1.11_2$

Step 4: Evaluate

$$(-1)^{Sign} * 2^{Exp-Bias} * 1.\mathbf{significand}_2$$

Plug into normalized formula

0 10000000 11000000000000000000000

Sign = 0, Exp = 128, Bias = -127, 1.significand = 1.11 Ignore trailing 0's

$(-1)^0 * 2^{128 - 127} * 1.11_2 = 2^1 * 1.11_2$

$= 11.1_2$ exponent is 1 = shifting decimal right by 1

$= 2^1 + 2^0 + 2^{-1}$

$= 3.5$

Converting From Decimal to Binary

Convert -5.625 to binary

1 bit for sign, 8 bits for exponent, 23 bits for significand, bias of -127

Step 1: Convert Left Side of Decimal

-5.625

Ignore sign for now (just make sign bit a 1 at the end)

$$5 = 2^2 + 2^0$$

$$= 101_2$$

Step 2: Convert Right Side of Decimal

$.625 = .5 + .125$

$= 2^{-1} + 2^{-3}$

$= .101_2$

Step 3: Combine Both Results and Normalize

$5.625 = 5 + .625$

$= 101_2 + .101_2$

$= 101.101_2$

$101.101_2 = 1.01101_2 * 2^2$

Decimal moved 2 places to the left

Step 4: Convert to Binary

$$-1.01101_2 * 2^2$$

sign	exponent	significand
1	10000001	01101
(negative)	(2 + 127 for bias)	(ignore implicit 1)

Combine to get: -5.625 = 11000000101101000000000000000000