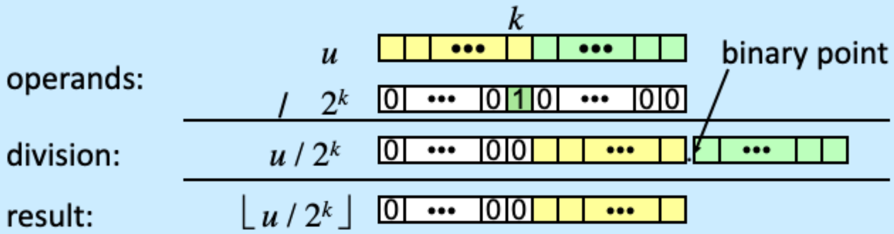
CS 33

Data Representation (Part 2)

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Unsigned Power-of-2 Divide with Shift

- **Quotient of unsigned by power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - uses logical shift

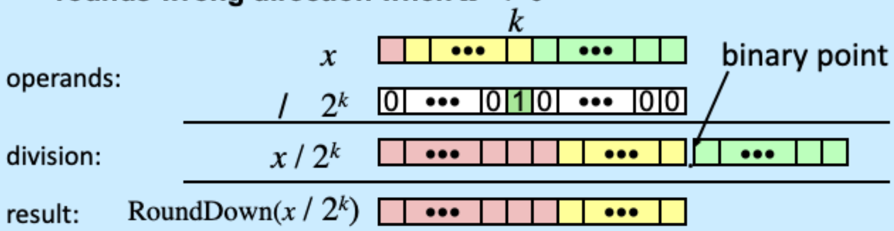


	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Supplied by CMU.

Signed Power-of-2 Divide with Shift

- **Quotient of signed by power of 2**
  - `x >> k` gives $\lfloor$ `x / 2`^{k} $\rfloor$
  - uses arithmetic shift
  - rounds wrong direction when `x < 0`



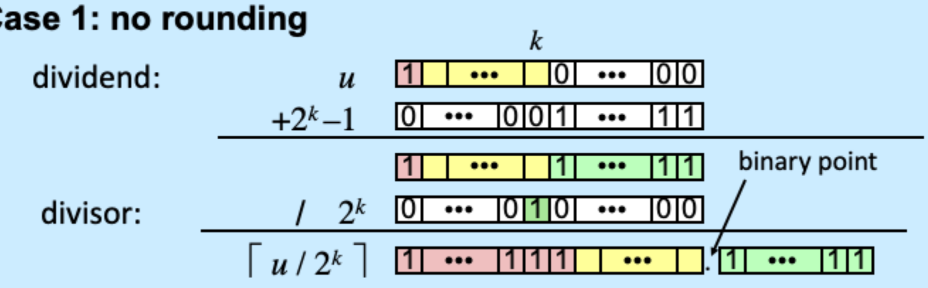
	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Supplied by CMU.

Correct Power-of-2 Divide

- **Quotient of negative number by power of 2**
  - **want $\lceil$ `x / 2`^{k} $\rceil$ (round toward 0)**
  - **compute as $\lfloor$ `(x+2`^{k}`-1) / 2`^{k} $\rfloor$**
    - » **in C: `(x + (1<<k)-1) >> k`**
    - » **biases dividend toward 0**

Case 1: no rounding

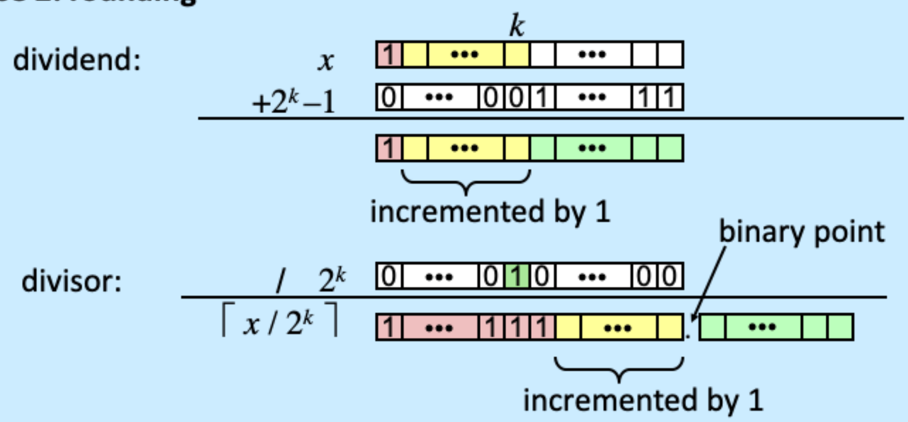


Biasing has no effect

Supplied by CMU.

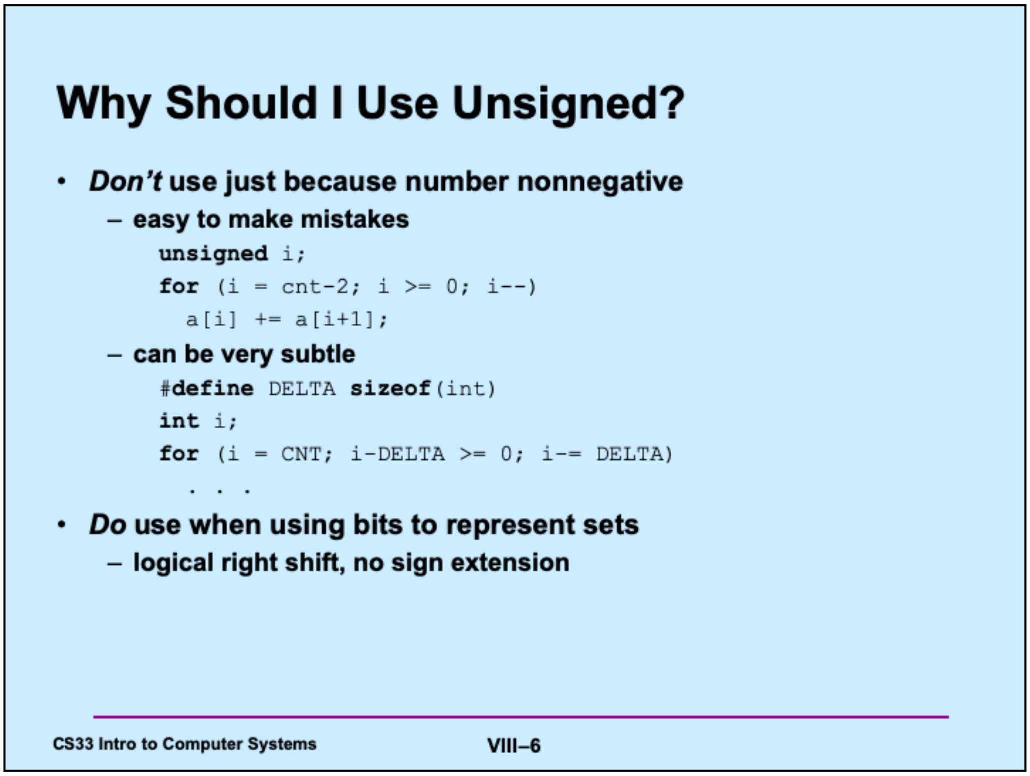
Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Supplied by CMU.



Supplied by CMU.

Note that "sizeof" returns an unsigned value. (Recall that, when mixing signed and unsigned items in an expression, the result will be unsigned.)

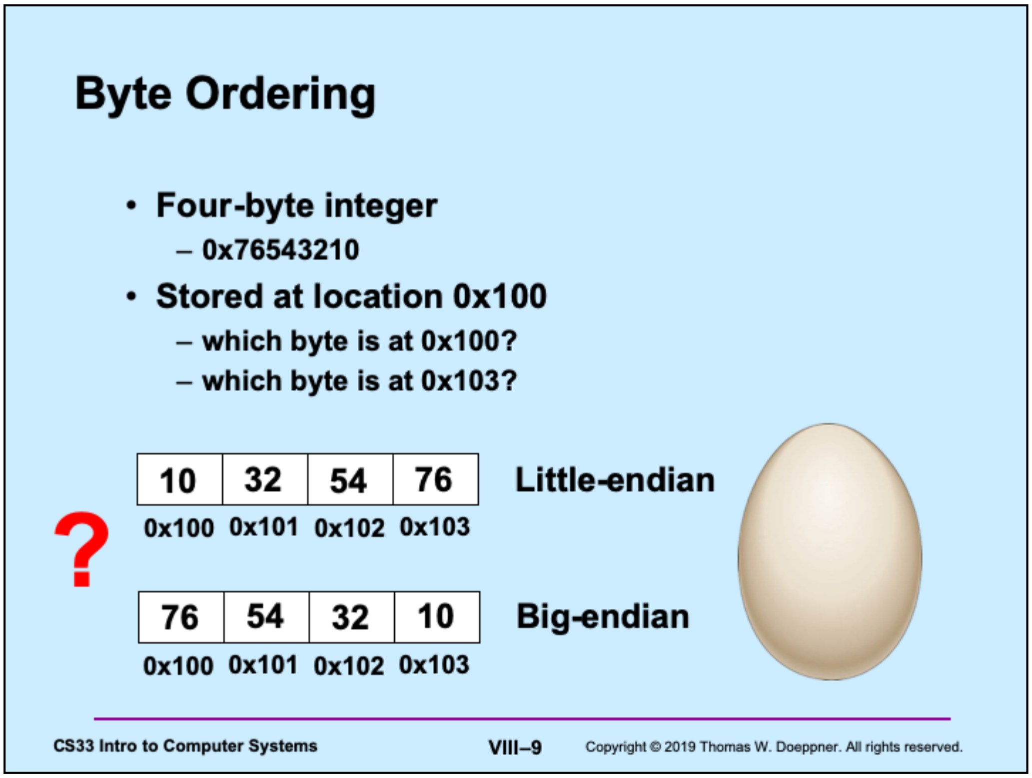
Combining Bytes

- **Data items of multiple sizes, usually powers of two**
  - one-byte, two-byte, four-byte, eight-byte integers
  - four-byte and eight-byte floating-point numbers
- **For example: four consecutive bytes interpreted as storing an integer (or a float)**
  - for best performance, address of lowest byte should be a multiple of the size of the item (four in this case)

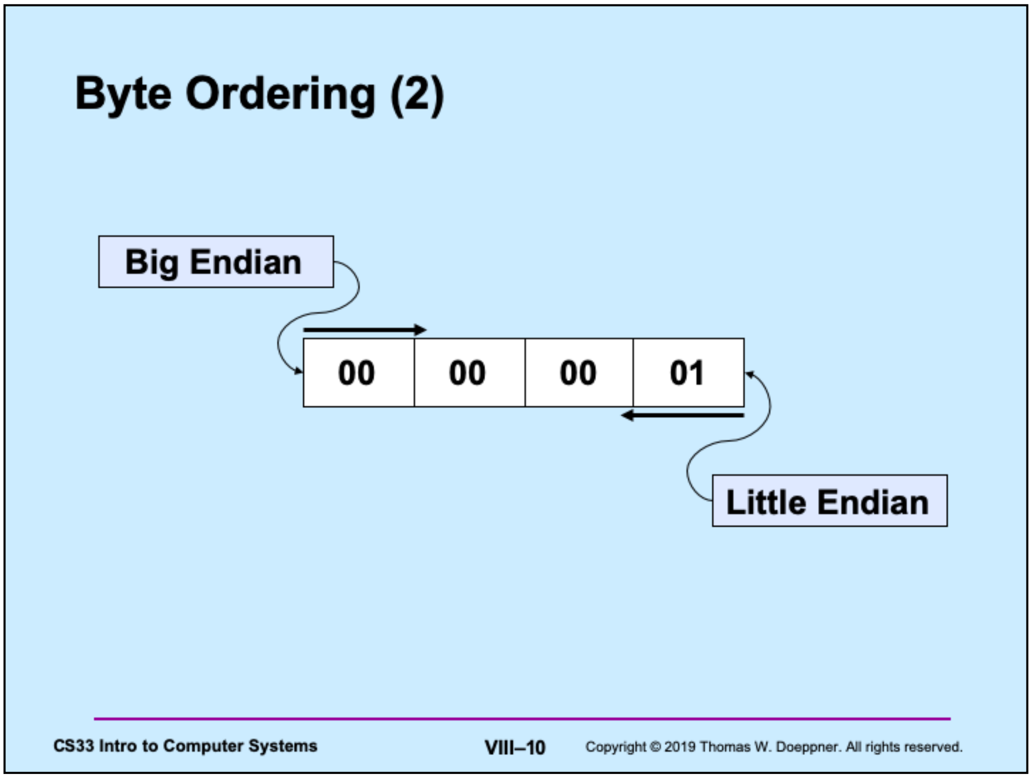
The reason for the performance issue has to do with how the memory subsystem works, somethings that will be explained in a few weeks.

Word Size

- (Mostly) obsolete term
  - old computers had items of one size: the word size
- Now used to express the number of bits necessary to hold an address
  - 16 bits (really old computers)
  - 32 bits (old computers)
  - 64 bits (most current computers)



Read “Gulliver’s Travels” for an explanation of the egg.



Here we have a four-byte integer one. In the big-endian representation, the address of the integer is the address of the byte containing its most-significant bits (the big end), while in the little-endian representation, the address of the integer is the address of the byte containing its least-significant bits (the little end). Suppose we pass a pointer to this integer to some procedure. However, in a type-mismatch, the procedure assumes that what is passed it is a two-byte integer. On a big-endian system, it would think it was passed a zero, but on a little-endian system, it would think it was passed a one.

This is not an argument in favor of either approach, but simply an observation that behaviors could be different.

Quiz 1

```
int main() {
  long x=1;
  func((int *)&x);
  return 0;
}

void func(int *arg) {
  printf("%d\n", *arg);
}
```

What value is printed on a big-endian 64-bit computer?

a) 0
b) 1
c) 2^{32}
d) 2^{32}-1

Which Byte Ordering Do We Use?

```
int main() {
    unsigned int x = 0x03020100;
    unsigned char *xarray = (unsigned char *)&x;
    for (int i=0; i<4; i++) {
            printf("%02x", xarray[i]);
    }
    printf("\n");
    return 0;
}
```

CS33 Intro to Computer Systems VIII–12

This code prints out the value of x, one byte at a time, starting with the byte at the lowest address (little end). On x86-based computers, it will print:

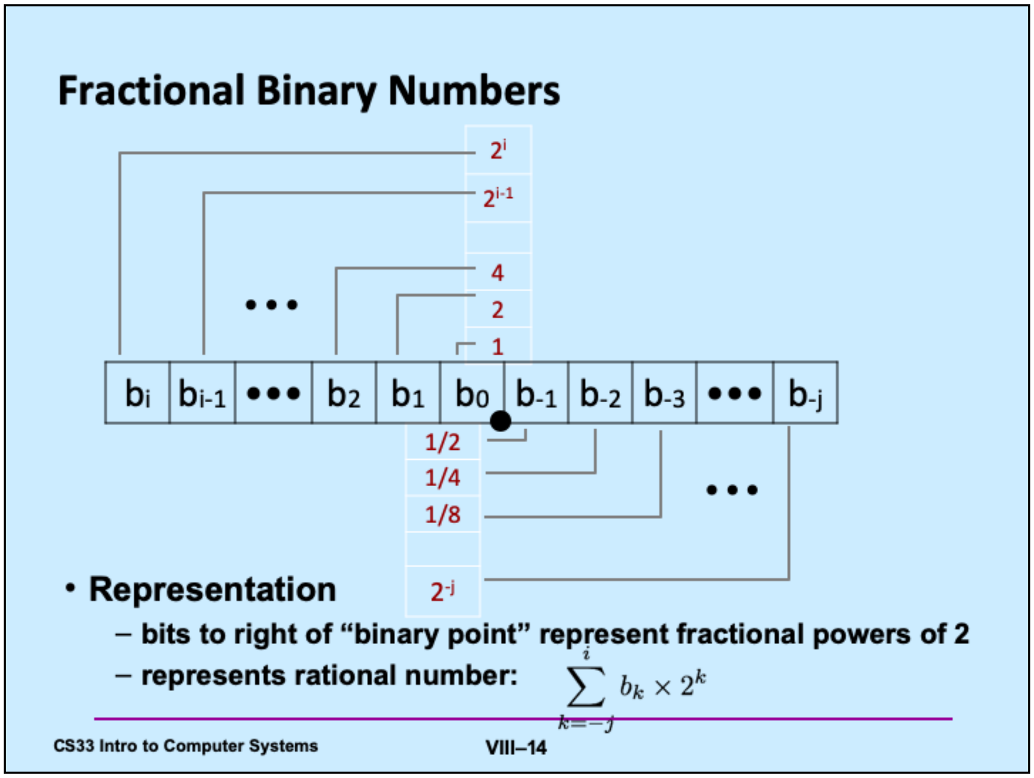
00010203

which means that the address of an int is the address of the byte containing its least significant digits (little endian).

Fractional binary numbers

- What is 1011.101_2?

Supplied by CMU.



Supplied by CMU.

Representable Numbers

- **Limitation #1**
  - can exactly represent only numbers of the form $n/2^k$
    - » other rational numbers have repeating bit representations
  - value — representation
    - » 1/3 — `0.0101010101[01]…2`
    - » 1/5 — `0.001100110011[0011]…2`
    - » 1/10 — `0.0001100110011[0011]…2`
- **Limitation #2**
  - just one setting of decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

Supplied by CMU.

IEEE Floating Point

- IEEE Standard 754
  - established in 1985 as uniform standard for floating point arithmetic
    - before that, many idiosyncratic formats
  - supported on all major CPUs

- Driven by numerical concerns
  - nice standards for rounding, overflow, underflow
  - hard to make fast in hardware
    - numerical analysts predominated over hardware designers in defining standard

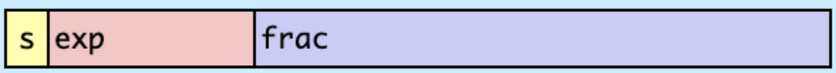
Supplied by CMU.

Floating-Point Representation

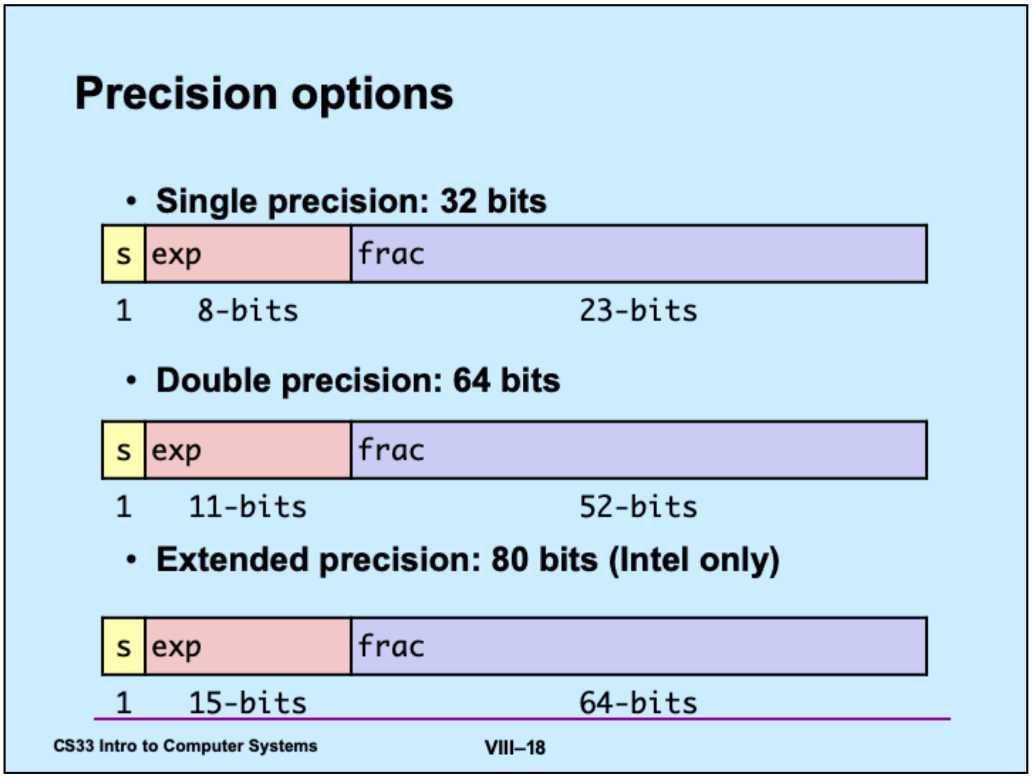
- **Numerical Form:**

$$(-1)^s \, M \; 2^E$$

  - sign bit **s** determines whether number is negative or positive
  - significand **M** normally a fractional value in range [1.0,2.0)
  - exponent **E** weights value by power of two
- **Encoding**
  - MSB `s` is sign bit **s**
  - `exp` field encodes **E** (but is not equal to E)
  - `frac` field encodes **M** (but is not equal to M)



Supplied by CMU.



Supplied by CMU.

On x86 hardware, all floating-point arithmetic is done with 80 bits, then reduced to either 32 or 64 as required.

"Normalized" Values

- When: exp ≠ 000…0 and exp ≠ 111…1
- Exponent coded as biased value: E = Exp – Bias
  - exp: unsigned value exp
  - bias = 2^{k-1} - 1, where k is number of exponent bits
    - » single precision: 127 (Exp: 1…254, E: -126…127)
    - » double precision: 1023 (Exp: 1…2046, E: -1022…1023)
- Significand coded with implied leading 1: M = $1.xxx…x_2$
  - xxx…x: bits of frac
  - minimum when frac=000…0 (M = 1.0)
  - maximum when frac=111…1 (M = 2.0 – ε)
  - get extra leading bit for "free"

Supplied by CMU.

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**
  - *M* = `1.`$\underline{\texttt{1101101101101}}_2$
  - `frac` = $\underline{\texttt{1101101101101}}\texttt{0000000000}_2$

- **Exponent**
  - *E* = 13
  - *bias* = 127
  - *exp* = 140 = $\texttt{10001100}_2$

- **Result:**

`s`	`exp`	`frac`
0	10001100	11011011011010000000000

Supplied by CMU.

Denormalized Values

- Condition: exp = `000…0`
- Exponent value: E = –Bias + 1 (instead of E = 0 – Bias)
- Significand coded with implied leading 0:
  $M = 0.xxx\ldots x_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - » represents zero value
    - » note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - » numbers closest to 0.0
    - » equispaced

Supplied by CMU.

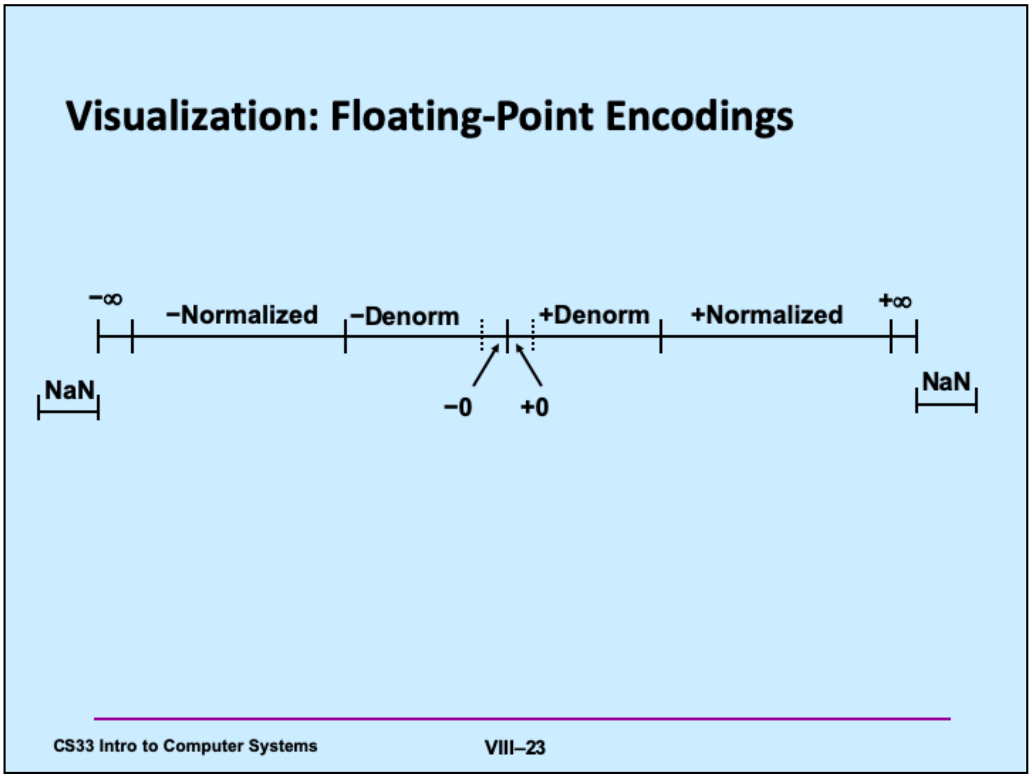
Special Values

- Condition: `exp` = 111…1

- Case: `exp` = 111…1, `frac` = 000…0
  - represents value ∞ (infinity)
  - operation that overflows
  - both positive and negative
  - e.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- Case: `exp` = 111…1, `frac` ≠ 000…0
  - not-a-number (NaN)
  - represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0

Supplied by CMU.



Supplied by CMU.

Tiny Floating-Point Example

s	exp	frac
1	4-bits	3-bits

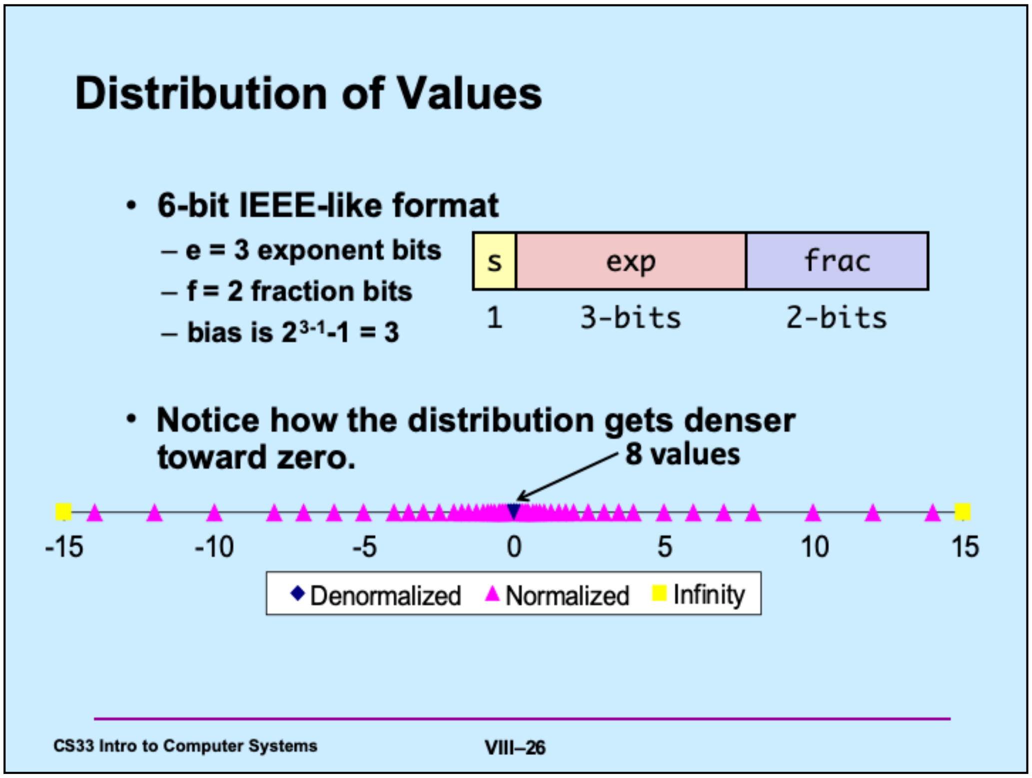
- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the `frac`
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Supplied by CMU.

Dynamic Range (Positive Only)

	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

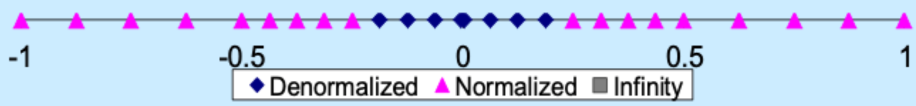
Supplied by CMU.



Supplied by CMU.

Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3



Supplied by CMU.

Quiz 2

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - bias is 3

s	exp	frac
1	3-bits	2-bits

What number is represented by 0 011 10?

a) 12
b) 1.5
c) .5
d) none of the above

Floating-Point Operations: Basic Idea

- `x` $+_f$ `y = Round(x + y)`
- `x` $\times_f$ `y = Round(x` $\times$ `y)`
- Basic idea
  - first **compute exact result**
  - make it fit into desired precision
    - » possibly overflow if exponent too large
    - » possibly **round to fit into** `frac`

Supplied by CMU.

Rounding

- **Rounding modes (illustrated with $ rounding)**

	$1.40	$1.60	$1.50	$2.50	–$1.50
towards zero	$1	$1	$1	$2	–$1
round down (−∞)	$1	$1	$1	$2	–$2
round up (+∞)	$2	$2	$2	$3	–$1
nearest integer	$1	$2	?	?	?
nearest even (default)	$1	$2	$2	$2	–$2

Supplied by CMU.

Creating a Floating Point Number

s	exp	frac
1	4-bits	3-bits

- **Steps**
  - normalize to have leading 1
  - round to fit within fraction
  - postnormalize to deal with effects of rounding

- **Case study**
  - convert 8-bit unsigned numbers to tiny floating-point format

example numbers

128	10000000
13	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Supplied by CMU.

Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - set binary point so that numbers of form 1.xxxxx
  - adjust all to have leading one
    - » decrement exponent as shift left

Value	Binary	Fraction	Exponent
128	10000000	1.0000000	7
13	00001101	1.1010000	3
17	00010001	1.0001000	4
19	00010011	1.0011000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Supplied by CMU.

Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit: OR of remaining bits

- **Round-up conditions**
  - round = 1, sticky = 1 ⇒ > 0.5
  - guard = 1, round = 1, sticky = 0 ⇒ round up to even

Value	Fraction	GRS	Incr?	Rounded
128	1.0000000	000	N	1.000
13	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Supplied by CMU.

Postnormalize

- **Issue**
  - rounding may have caused overflow
  - handle by shifting right once & incrementing exponent

Value	Rounded	Exp	Adjusted	Result
128	1.000	7		128
13	1.101	3		13
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	$1.000*2^6$	64

Supplied by CMU.

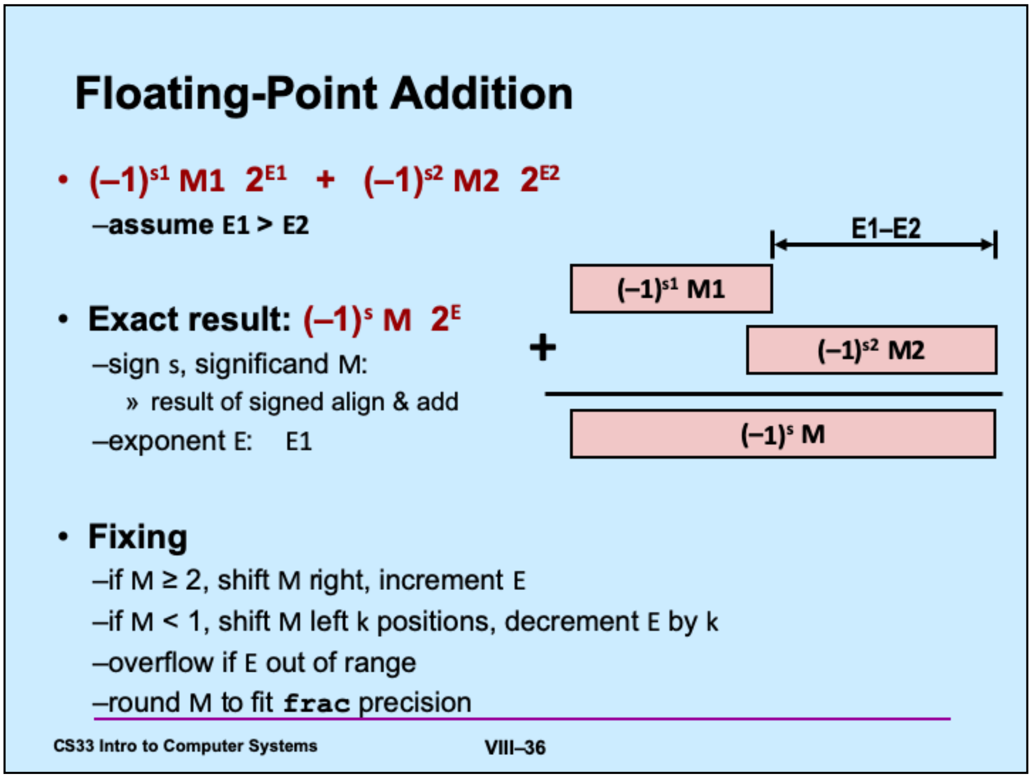
Floating-Point Multiplication

- $(-1)^{s1}$ M1 2^{E1} x $(-1)^{s2}$ M2 2^{E2}
- **Exact result:** $(-1)^{s}$ M 2^{E}
  - sign s: s1 ^ s2
  - significand M: M1 x M2
  - exponent E: E1 + E2

- **Fixing**
  - if M ≥ 2, shift M right, increment E
  - if E out of range, overflow (or underflow)
  - round M to fit `frac` precision
- **Implementation**
  - biggest chore is multiplying significands

Supplied by CMU.

Note that to compute E, one must first convert exp_1 and exp_2 to E_1 and E_2, then add them them together and check for underflow or overflow (corresponding to -∞ and +∞), and then convert to exp.



Supplied by CMU.

Note that, by default, overflow results in either +∞ or -∞.