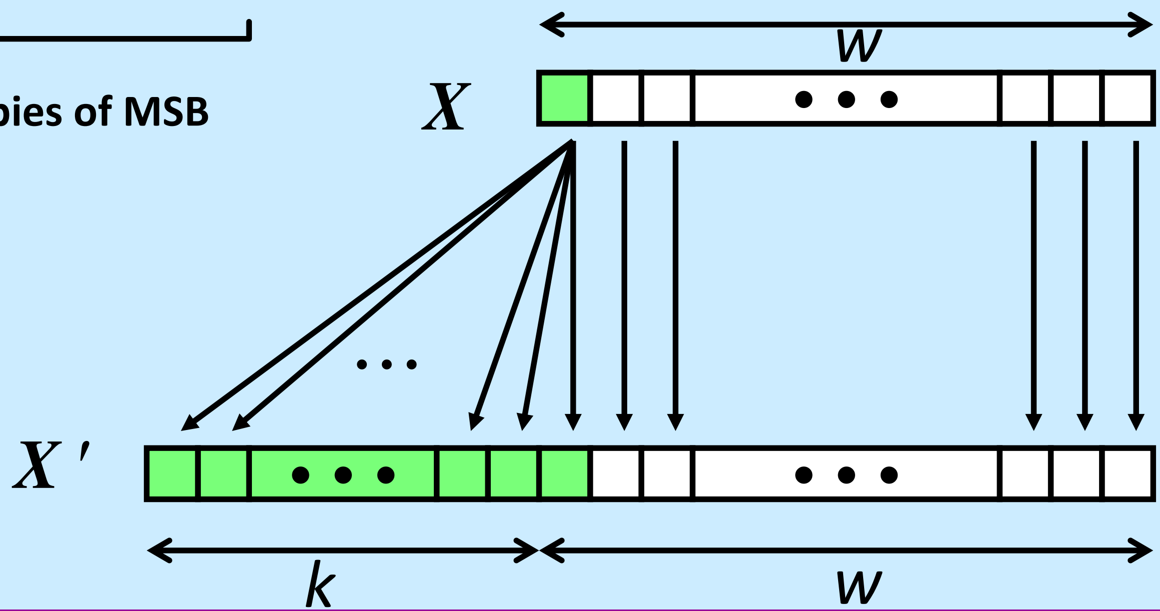
CS 33

Data Representation (Part 2)

Sign Extension

- **Task:**
  - given *w*-bit signed integer *x*
  - convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
  - C automatically performs sign extension

Does it Work?

$$val_w = -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

$$\begin{aligned} val_{w+1} &= -2^{w} + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

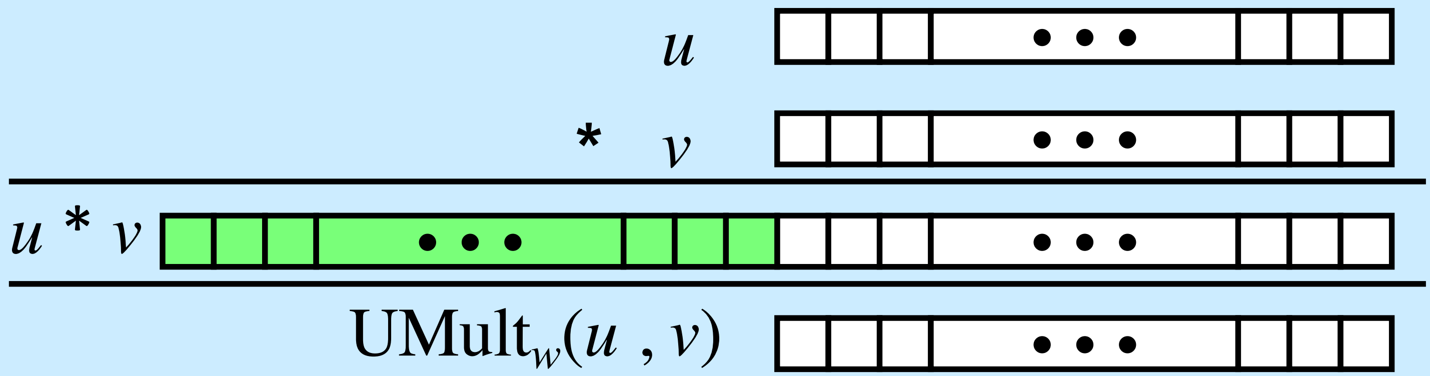
$$\begin{aligned} val_{w+2} &= -2^{w+1} + 2^{w} + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w} + 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \\ &= -2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i \end{aligned}$$

Unsigned Multiplication

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard multiplication function**
  - **ignores high order *w* bits**
- **Implements modular arithmetic**

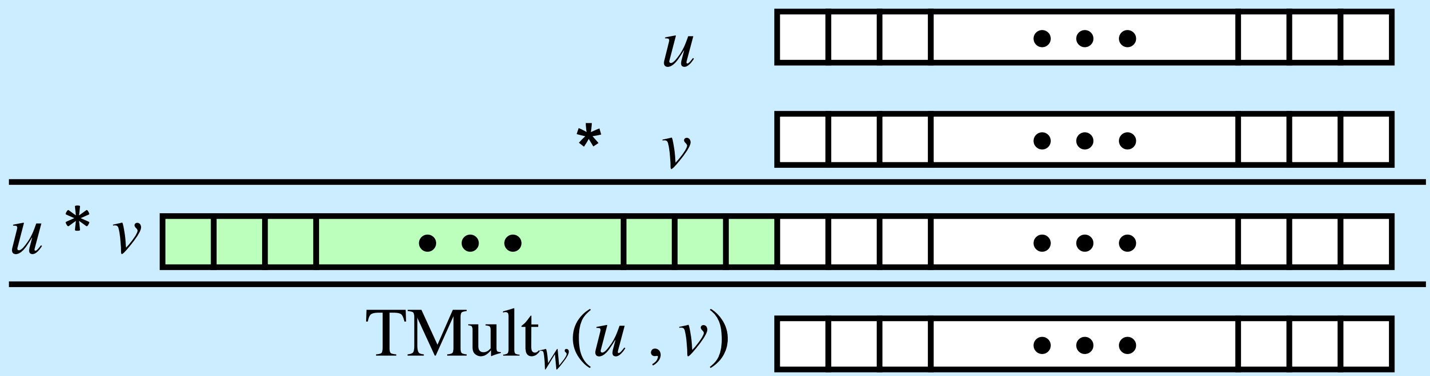
$\mathrm{UMult}_w(u\ ,\ v) \quad = \quad u \cdot v \bmod 2^w$

Signed Multiplication

Operands: *w* bits

True Product: 2**w* bits

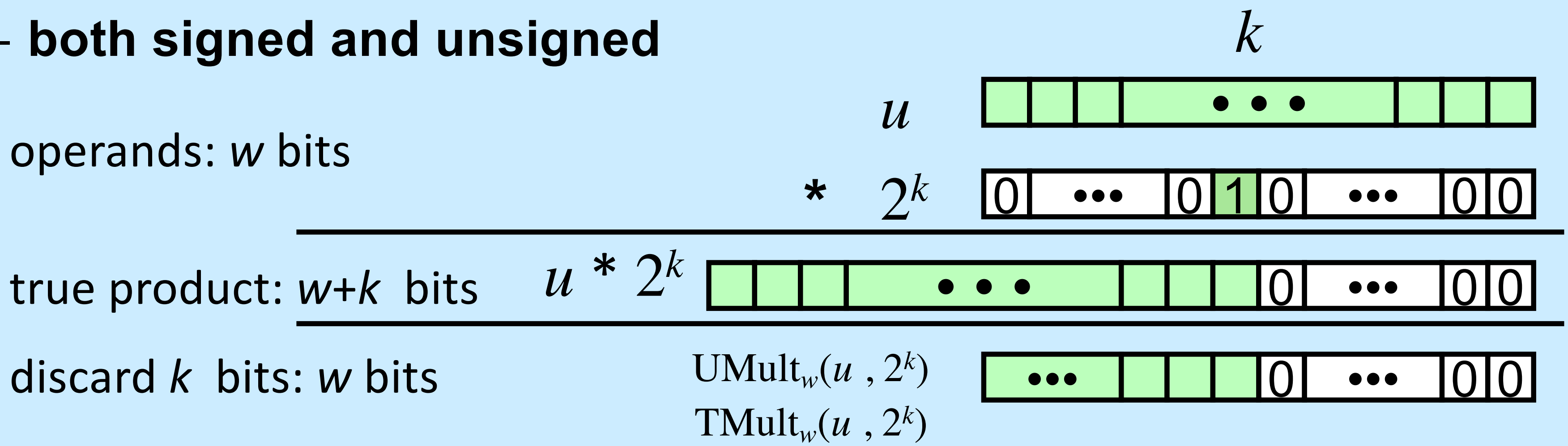
Discard *w* bits: *w* bits



- **Standard multiplication function**
  - **ignores high order *w* bits**
  - **some of which are different from those of unsigned multiplication**
  - **lower bits are the same**

Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` **gives** `u * 2`^{k}
  - **both signed and unsigned**



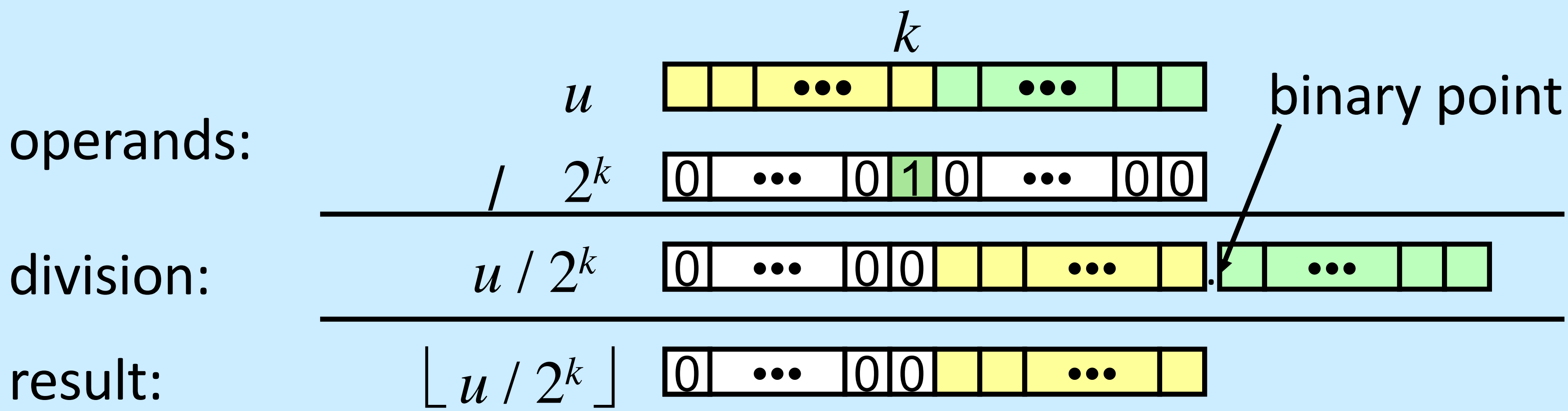
- **Examples**

```
u << 3 ==      u * 8
u << 5 - u << 3 == u * 24
```

  - **most machines shift and add faster than multiply**
    - » **compiler generates this code automatically**

Unsigned Power-of-2 Divide with Shift

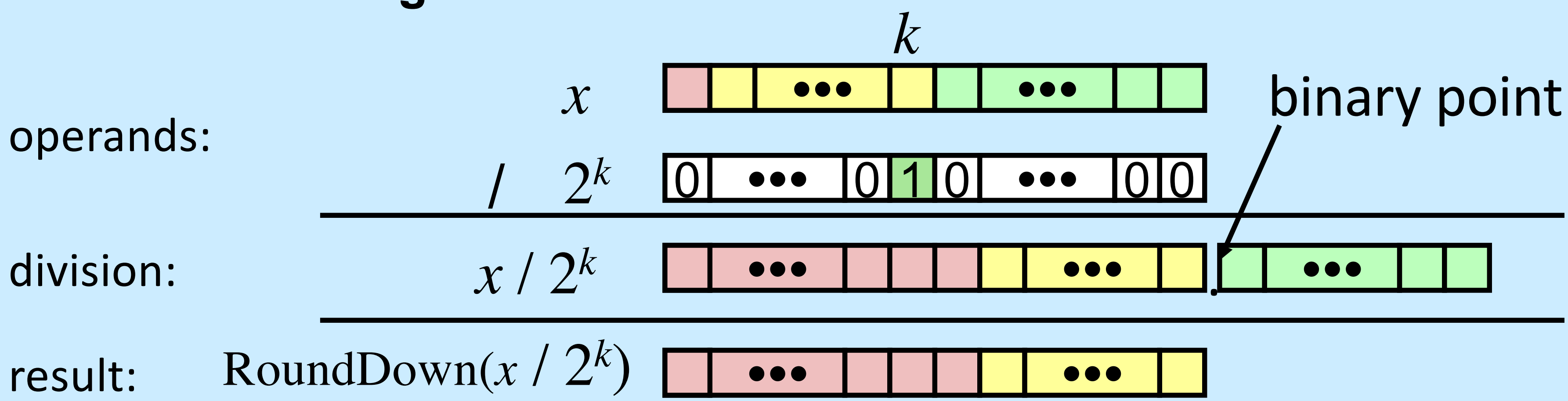
- **Quotient of unsigned by power of 2**
  - `u >> k` gives $\lfloor$ `u / ` $2^k \rfloor$
  - uses logical shift



	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`

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`

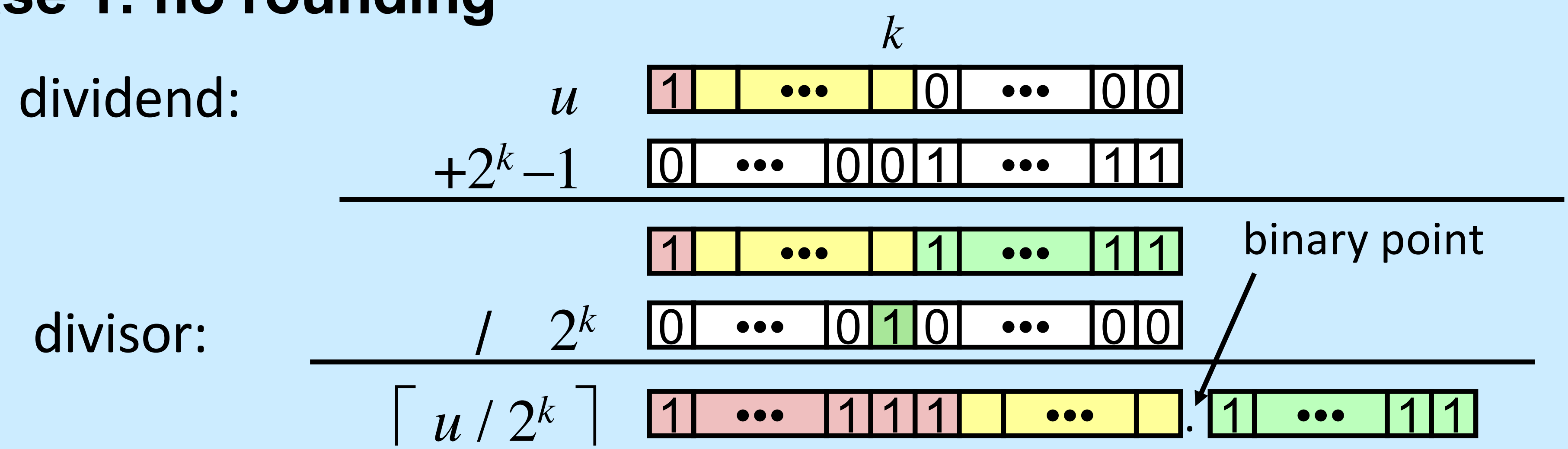


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

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

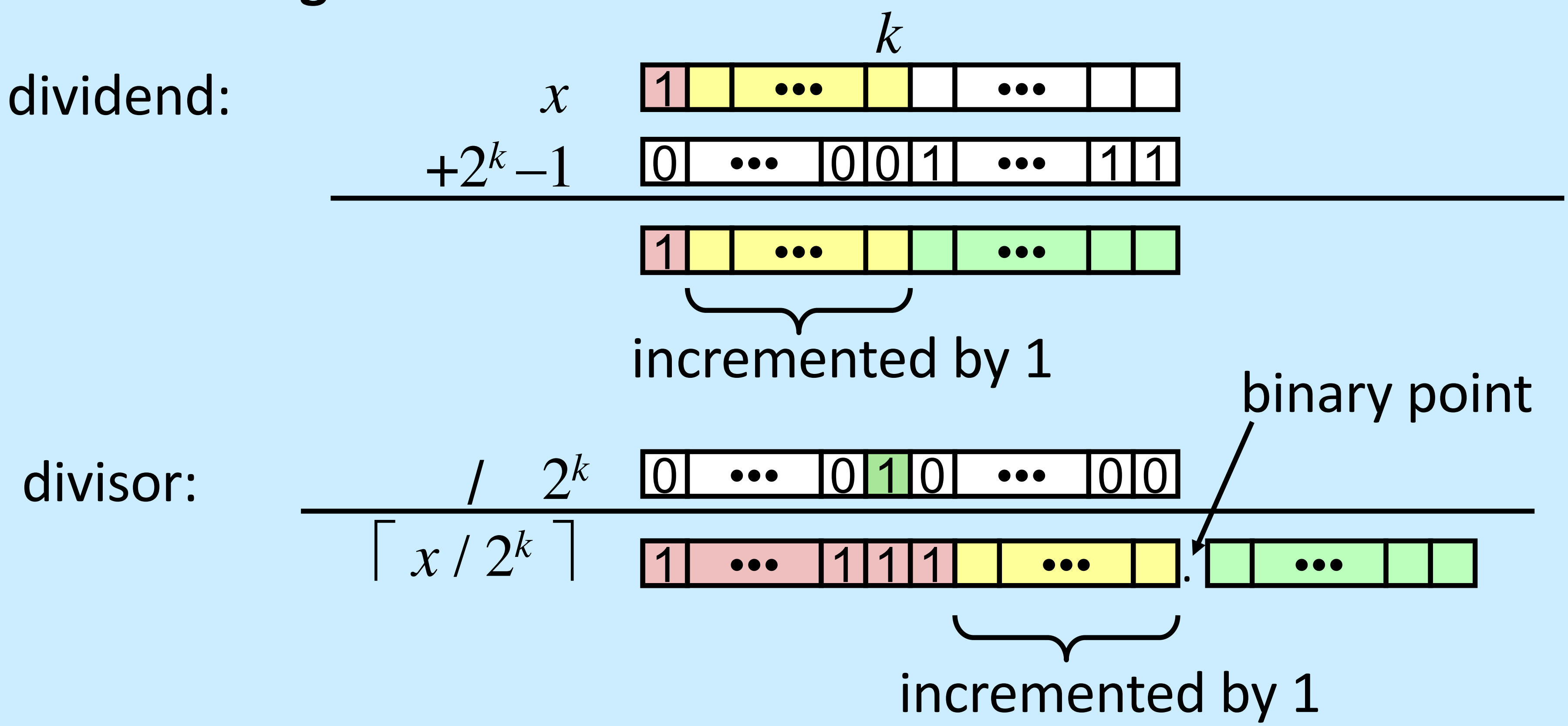
Case 1: no rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: rounding



Biasing adds 1 to final result

Why Should I Use Unsigned?

- *Don't* use just because number nonnegative
  - easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

- *Do* use when performing modular arithmetic
  - multiprecision arithmetic
- *Do* use when using bits to represent sets
  - logical right shift, no sign extension

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)

Byte Ordering

- **Four-byte integer**
  - 0x76543210
- **Stored at location 0x100**
  - which byte is at 0x100?
  - which byte is at 0x103?

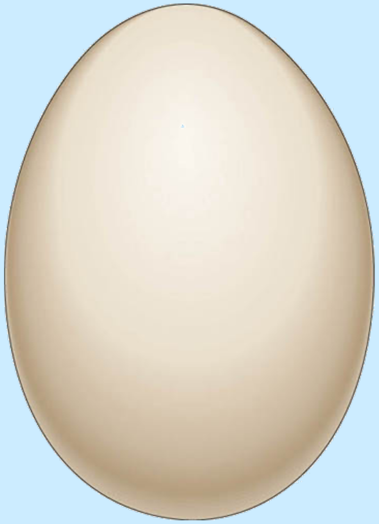
?

10	32	54	76
0x100	0x101	0x102	0x103

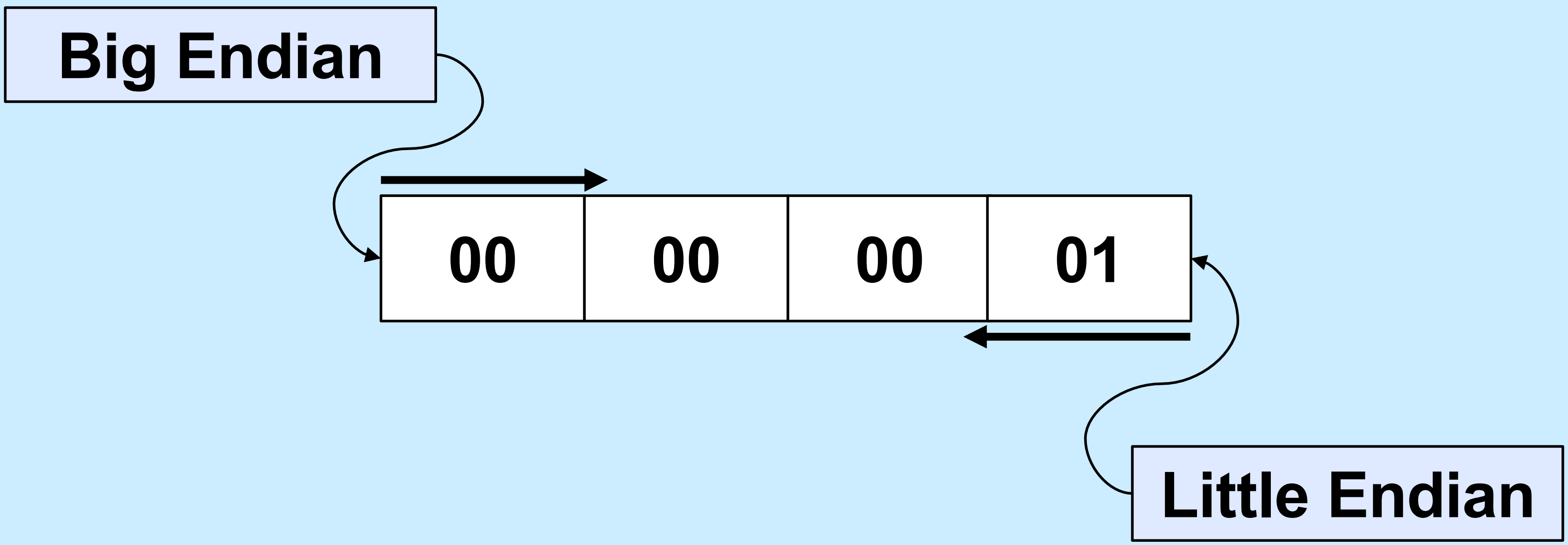
Little-endian

76	54	32	10
0x100	0x101	0x102	0x103

Big-endian



Byte Ordering (2)



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;
}
```

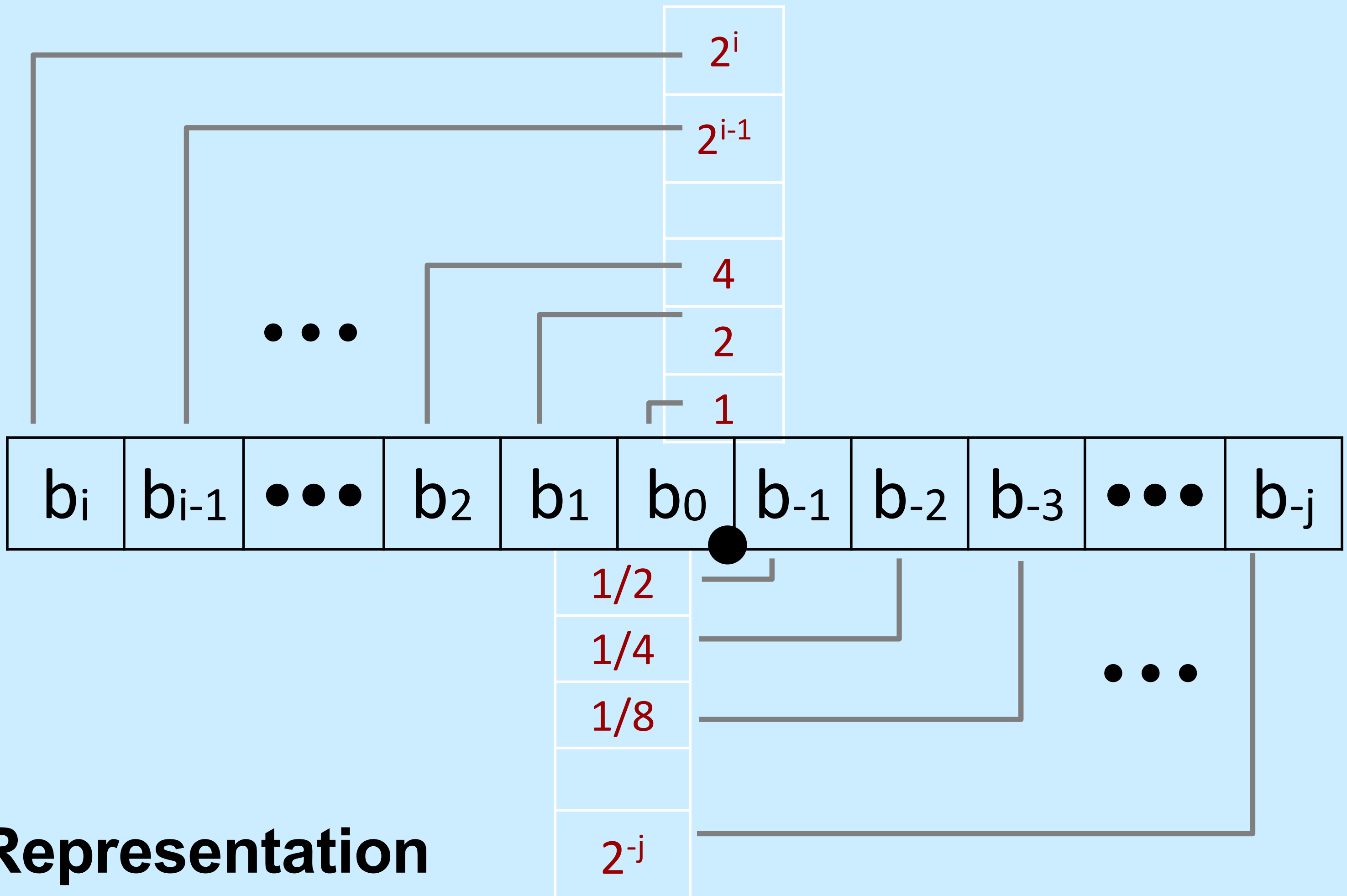
Possible results:

```
00010203
03020100
```

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



- **Representation**
  - **bits to right of "binary point" represent fractional powers of 2**
  - **represents rational number:** $\sum_{k=-j}^{i} b_k \times 2^k$

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 — $0.0101010101[01]\ldots_2$
    - » 1/5 — $0.001100110011[0011]\ldots_2$
    - » 1/10 — $0.0001100110011[0011]\ldots_2$

- **Limitation #2**
  - just one setting of decimal point within the *w* bits
    - » limited range of numbers (very small values? very large?)

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

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)

s	exp	frac

Precision options

- **Single precision: 32 bits**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	64-bits

“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”

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`

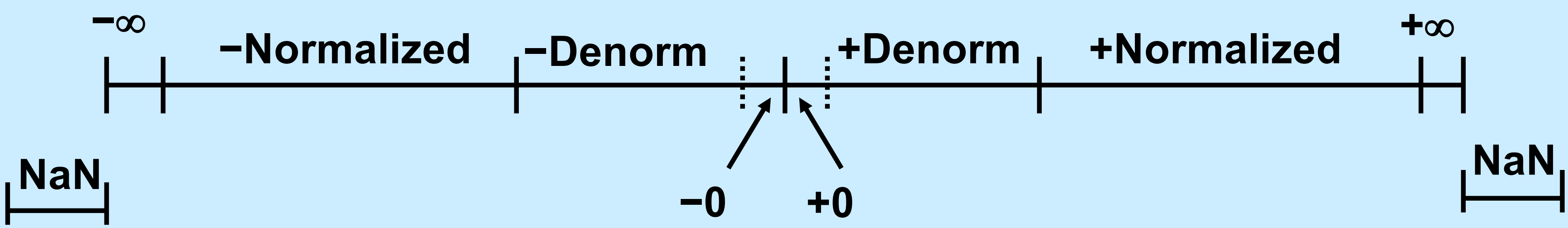
Denormalized Values

- 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

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

Visualization: Floating-Point Encodings



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`

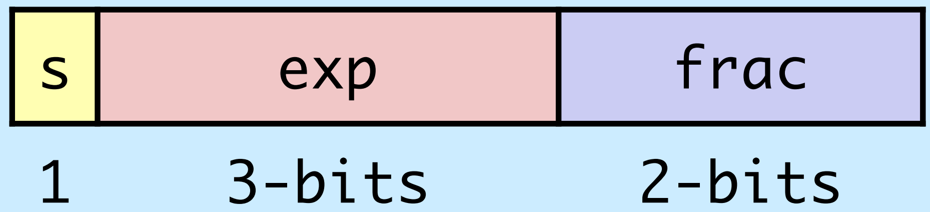
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

Dynamic Range (Positive Only)

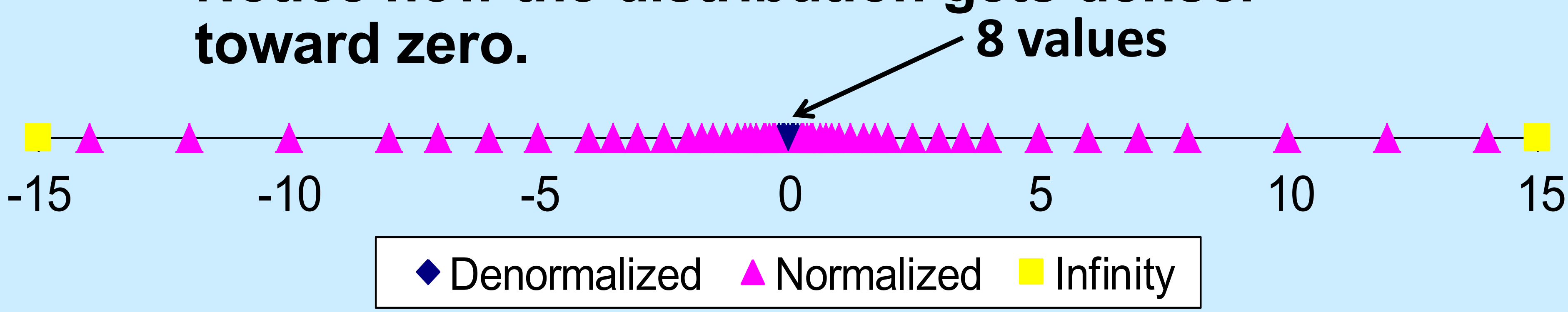
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-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
	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	

Distribution of Values

- **6-bit IEEE-like format**
  - **e = 3 exponent bits**
  - **f = 2 fraction bits**
  - **bias is 2^{3-1}-1 = 3**

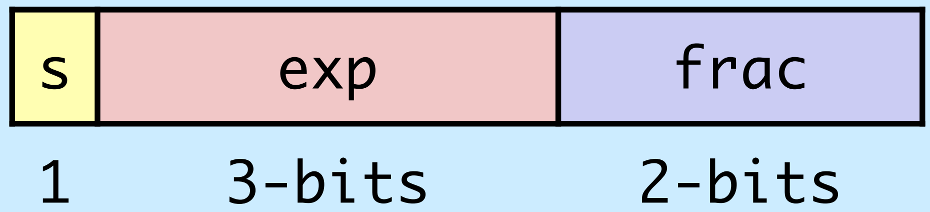


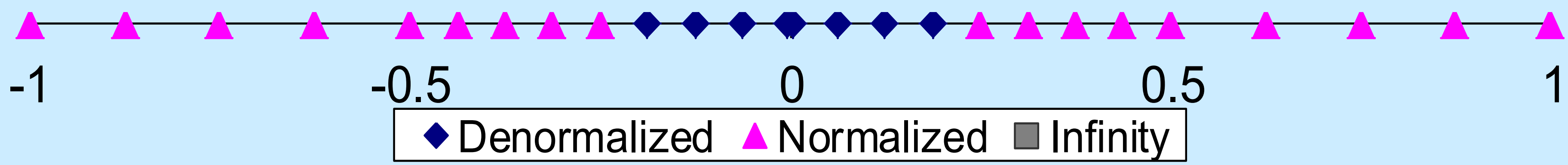
- **Notice how the distribution gets denser toward zero.**



Distribution of Values (close-up view)

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





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?

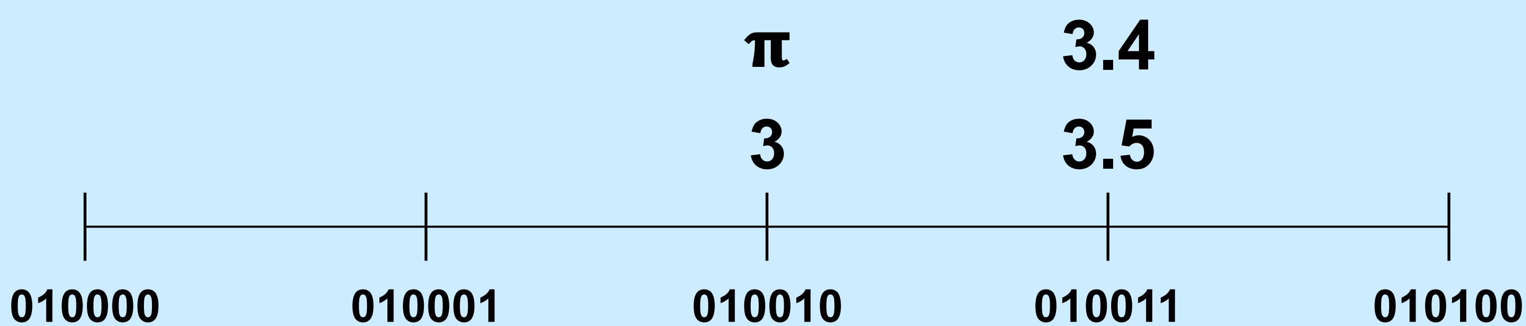
a) 12
b) 1.5
c) .5
d) none of the above

Mapping Real Numbers to Float

- The real number 3 is represented as
  0 100 10
- The real number 3.5 is represented as
  0 100 11
- How is the real number 3.4 represented?

  0 100 11
- How is the real number π represented?

  0 100 10

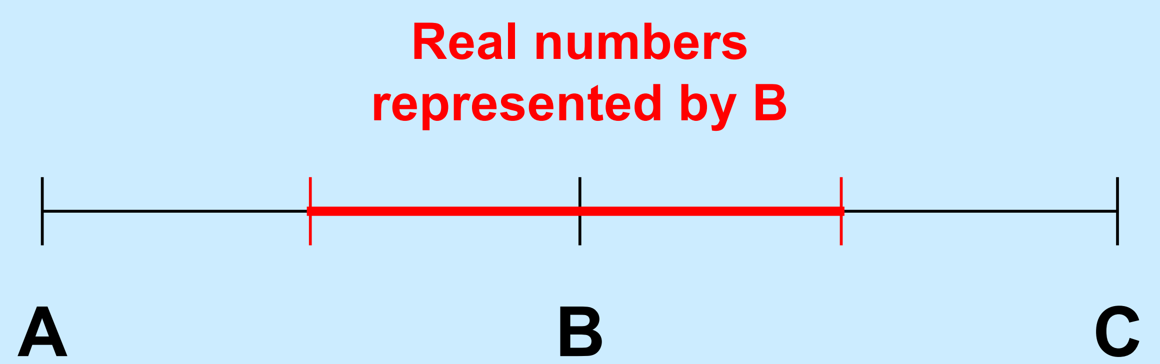


Mapping Real Numbers to Float

- **If R is a real number, it's mapped to the floating-point number whose value is closest to R**
- **What if it's midway between two values?**
  - **rounding rules coming up soon!**

Floats are Sets of Values

- If A, B, and C are successive floating-point numbers
  - e.g., 010001, 010010, and 010011
- B represents all real numbers from midway between A and B through midway between B and C



Significance

- **Normalized numbers**
  - for a particular exponent value E and an S-bit significand, the range from 2^E up to 2^{E+1} is divided into 2^S equi-spaced floating-point values
    - » thus each floating-point value represents $1/2^S$ of the range of values with that exponent
    - » all bits of the signifcand are important
    - » we say that there are S significant bits – for reasonably large S, each floating-point value covers a rather small part of the range
      - high accuracy
      - for S=23 (32-bit float), accurate to one in 2^{23} (.0000119% accuracy)

Significance

- **Unnormalized numbers**
  - high-order zero bits of the significand aren't important
  - in 8-bit floating point, 0 0000 001 represents 2^{-9}
    - » it is the only value with that exponent: 1 significant bit (either 2^{-9} or 0)
  - 0 0000 010 represents 2^{-8}
    0 0000 011 represents $1.5*2^{-8}$
    - » only two values with exponent -8: 2 significant bits (encoding those two values, as well as 2^{-9} and 0)
  - fewer significant bits mean less accuracy
  - 0 0000 01 represents a range of values from $.5*2^{-9}$ to $1.5*2^{-9}$
  - 50% accuracy

Floating-Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$

- $x \times_f y = \text{Round}(x \times y)$

- Basic idea
  - first compute exact result
  - make it fit into desired precision
    - » possibly overflow if exponent too large
    - » possibly round to fit into `frac`

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

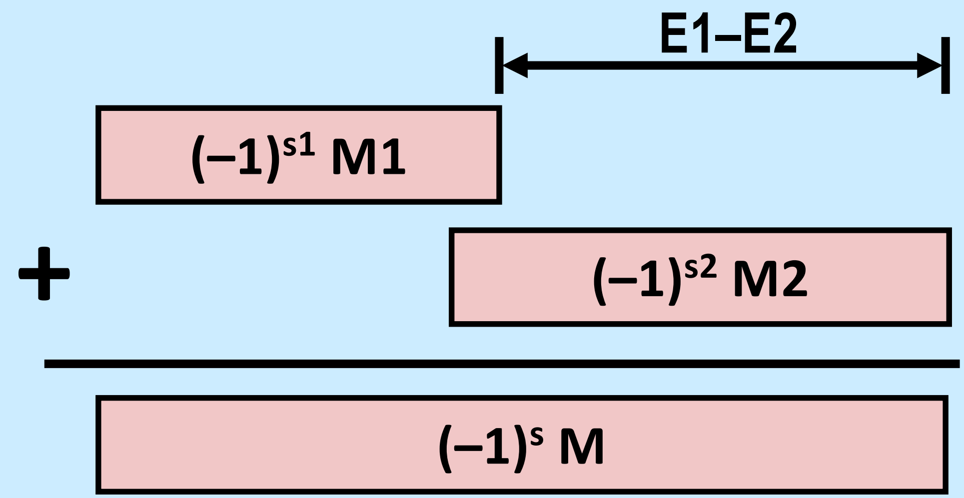
Floating-Point Multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \ x\ \ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **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

Floating-Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1} \quad + \quad (-1)^{s2}\ M2\ \ 2^{E2}$
  - **–assume E1 > E2**



- **Exact result:** $(-1)^{s}\ M\ \ 2^{E}$
  - –sign s, significand M:
    - » result of signed align & add
  - –exponent E: E1

- **Fixing**
  - –if M ≥ 2, shift M right, increment E
  - –if M < 1, shift M left k positions, decrement E by k
  - –overflow if E out of range
  - –round M to fit `frac` precision