CSC 252: Computer Organization
Fall 2021: Lecture 4

Binary Multiplication
Floating Point

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

Announcements

6 days left to complete assignment 1

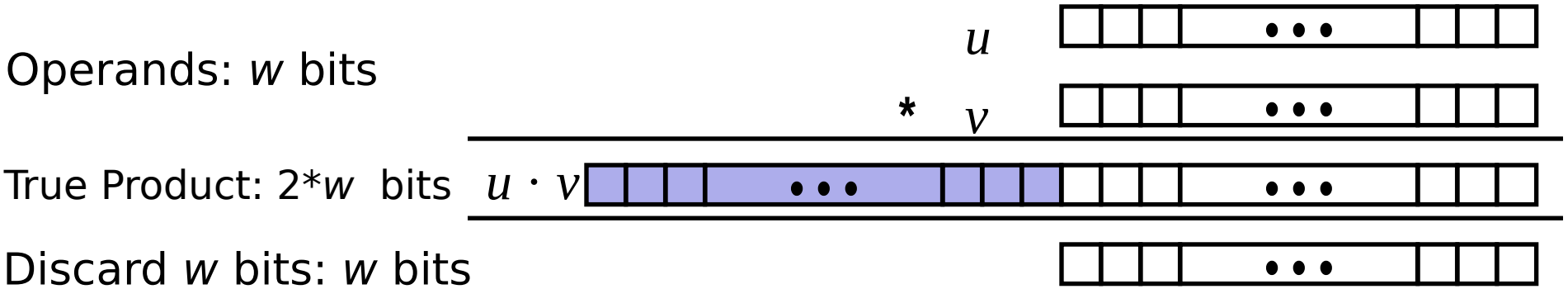
Office hours are shown on the website. Some are in person, some are on Zoom.

For office hours using Zoom, use the zoom link in blackboard sidebar

Today is the end of the data representation unit

Next class we begin assembly programming

Unsigned Multiplication in C



Standard Multiplication Function

Ignores high order w bits

Effectively Implements the following:

$\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

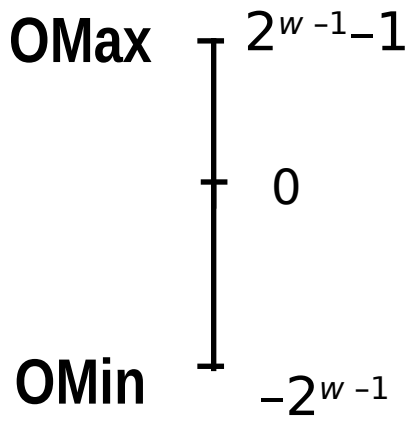
Multiplication

Goal: Computing Product of *w*-bit numbers *x*, *y*

Exact results can be bigger than *w* bits

Up to *2w* bits (both signed and unsigned)

Original Number (w bits)

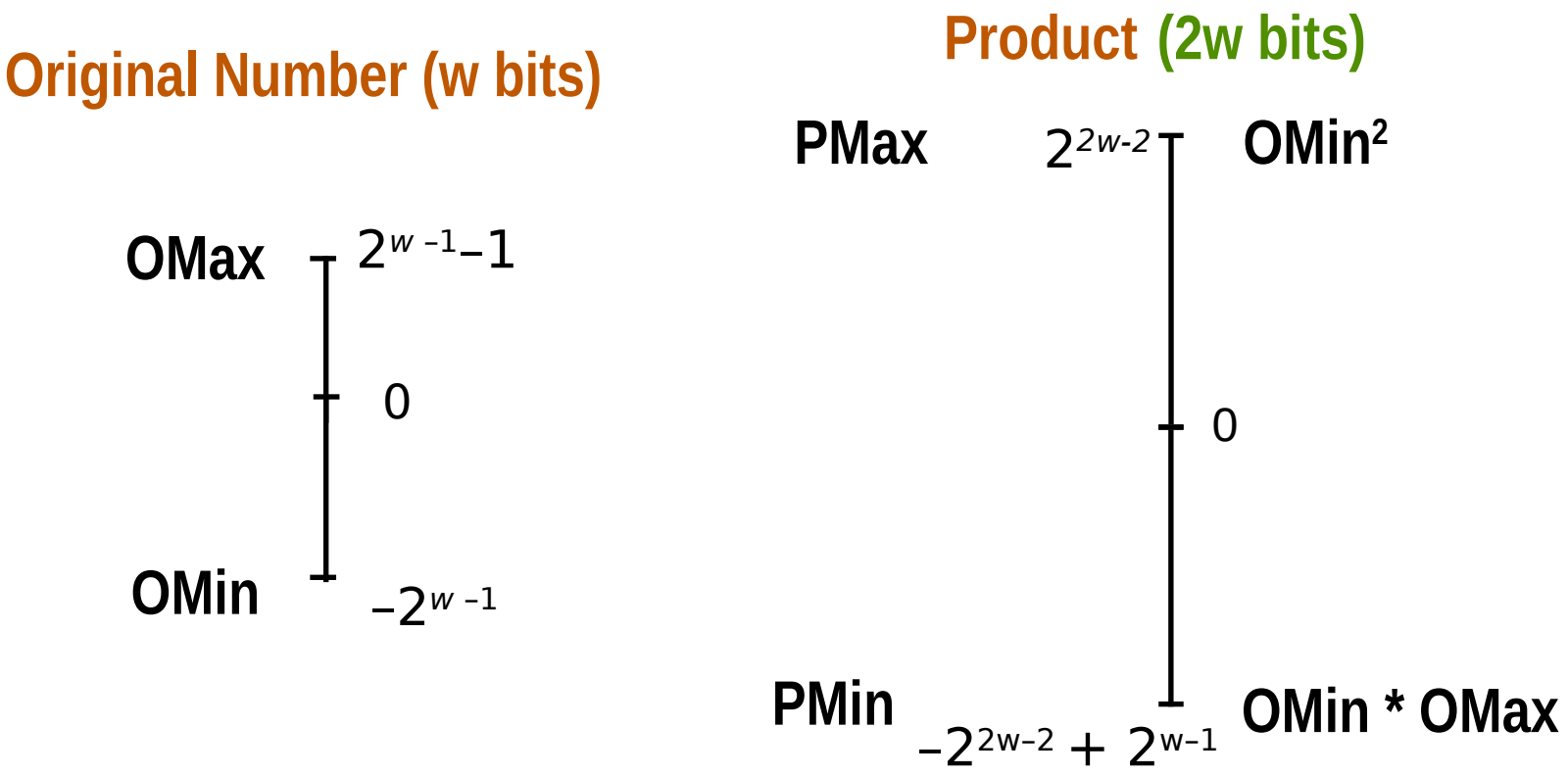


Multiplication

Goal: Computing Product of *w*-bit numbers *x*, *y*

Exact results can be bigger than *w* bits

Up to *2w* bits (both signed and unsigned)



Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^{1} + 0*2^{0} + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

What does 10.01_2 mean?

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

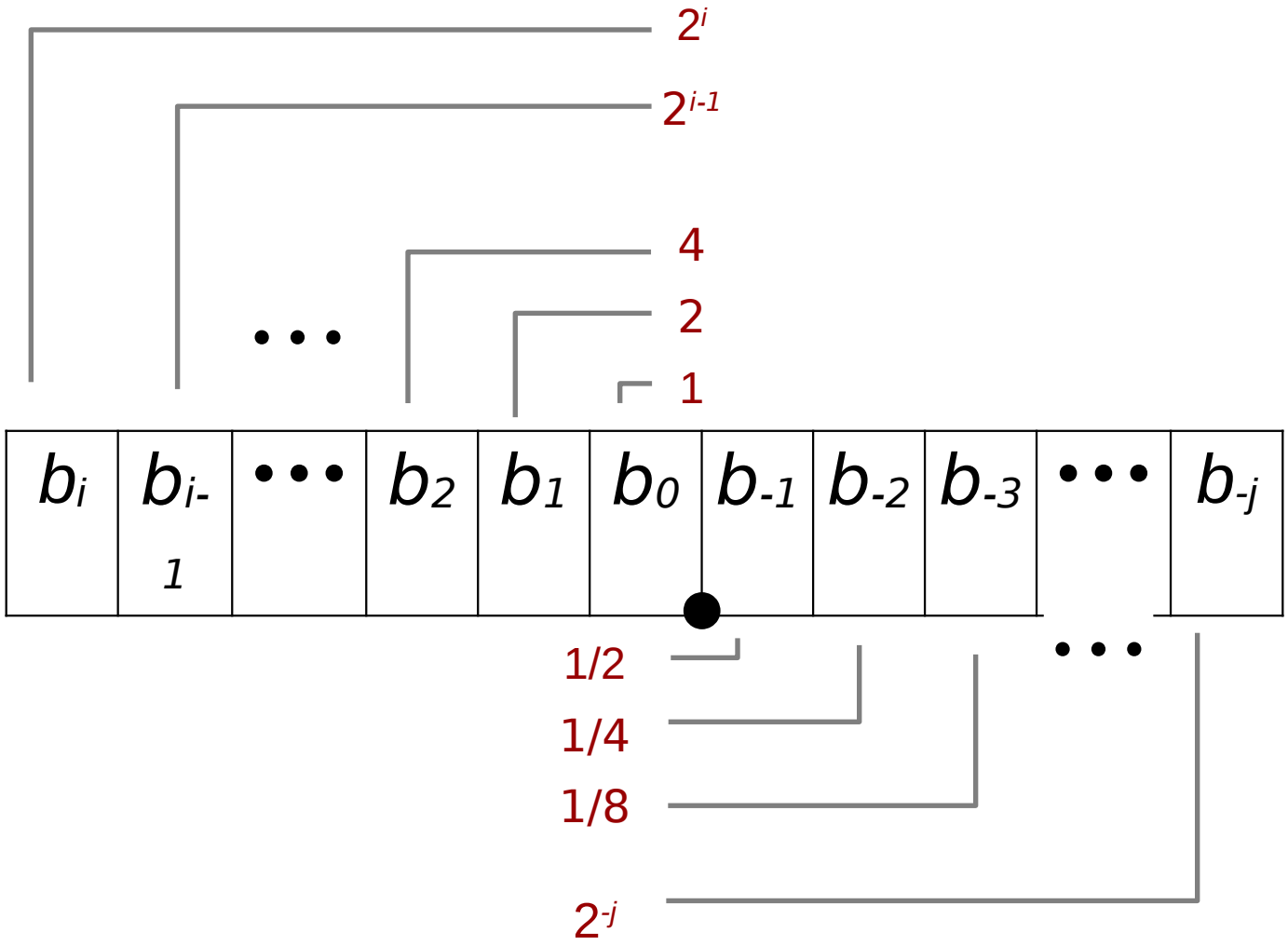
What does 10.01_2 mean?

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

$$= 2.25_{10}$$

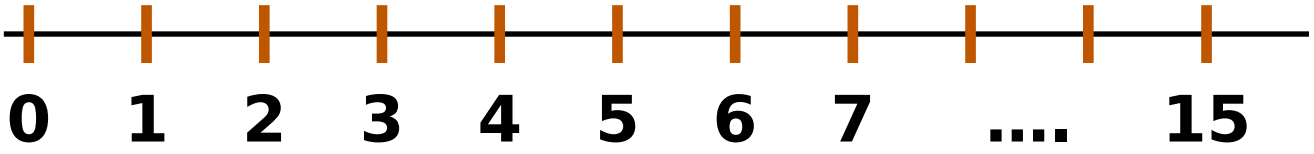
Fractional Binary Numbers



Fixed-Point Representation

Binary point stays fixed

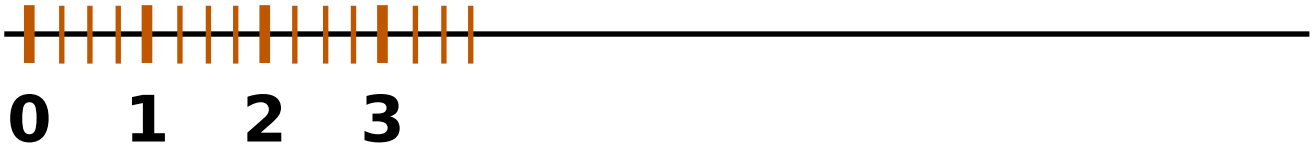
Fixed interval between representable numbers



Decimal	Binary
0	0000.
1	0001.
2	0010.
3	0011.
4	0100.
5	0101.
6	0110.
7	0111.
8	1000.
9	1001.
10	1010.
11	1011.
12	1100.
13	1101.
14	1110.
15	1111.

Fixed-Point Representation

- Binary point stays fixed
  - Fixed interval between representable numbers
    - The interval in this example is 0.25_{10}



- Still need to remember the binary point, but just once for all numbers, which is implicit given the data type
- Usual arithmetic still works
  - No need to align (already aligned)

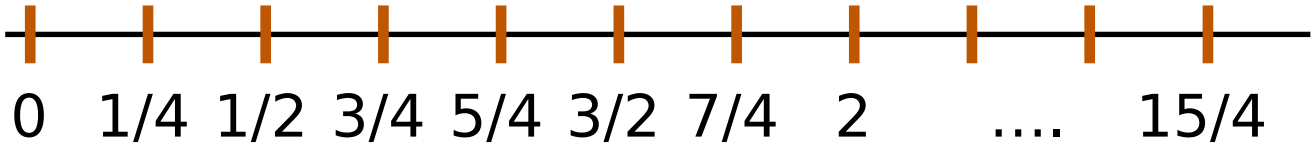
Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Limitations of Fixed-Point (*#1*)

Can exactly represent numbers only of the form $x/2^k$

Other rational numbers have repeating bit representations

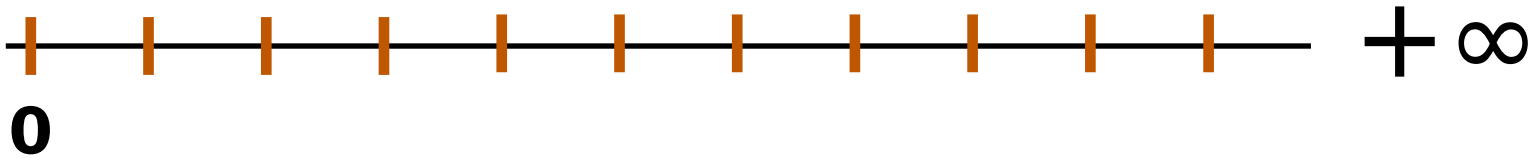
Decimal Value	**Binary Representation**
1/3	0.0101010101[01]…
1/5	0.001100110011[0011]…
1/10	0.0001100110011[0011]…



$b_3b_2.b_1b_0$

Limitations of Fixed-Point (*#2*)

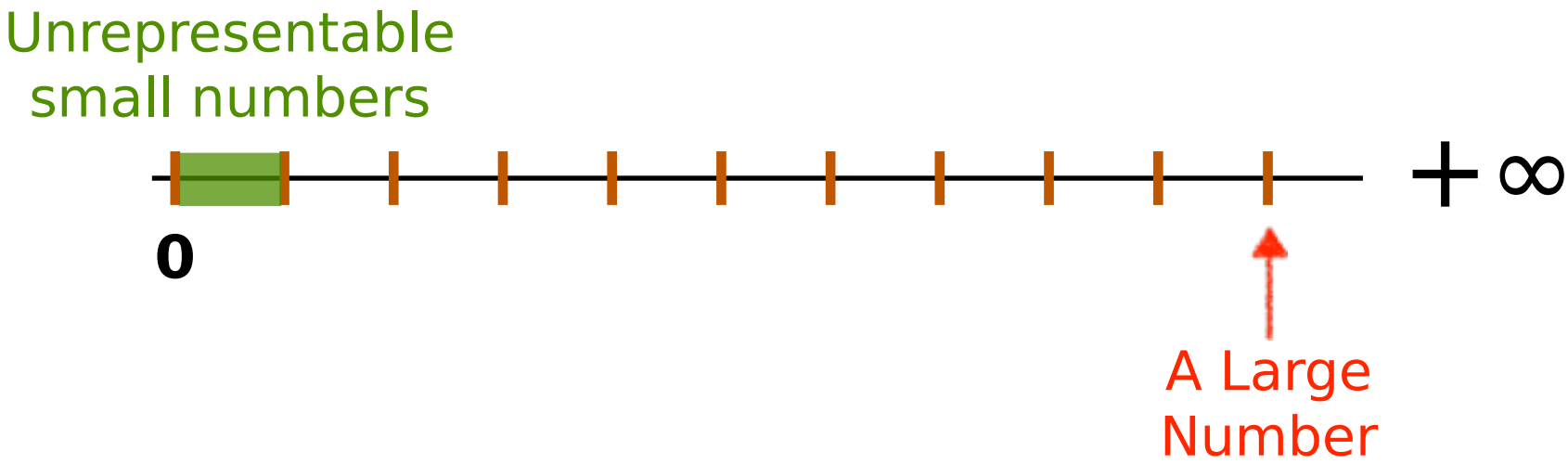
Can’t represent very small and very large numbers at the same time



Limitations of Fixed-Point (*#2*)

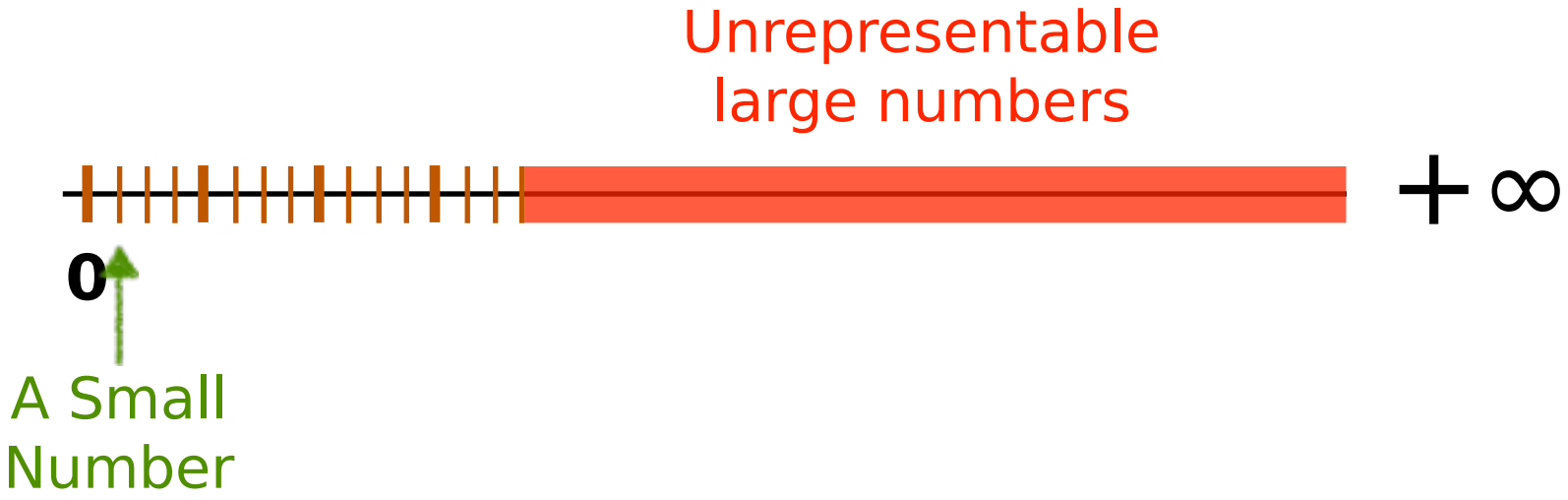
Can’t represent very small and very large numbers at the same time

To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers



Limitations of Fixed-Point (*#2*)

- Can’t represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers
  - To represent very small numbers, the (fixed) interval needs to be small, making it hard to represent large numbers



Primer: (Normalized) Scientific Notation

In decimal: $M \times 10^E$

Decimal Value	Scientific Notation
2	2×10^{0}
-4,321.768	-4.321768×10^{3}
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

In decimal: $M \times 10^E$

E is an integer

Normalized form: $1 <= |M| < 10$

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

In decimal: $M \times 10^E$

E is an integer

Normalized form: $1 <= |M| < 10$

$$M \times 10^E$$

Significand Base Exponent

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

In binary: $(-1)^s \; M \; 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0 b_1 b_2 b_3 \ldots$

Binary Value	**Scientific Notation**
1110110110110	$(-1)^0 \; 1.110110110110 \times 2^{12}$
-101.11	$(-1)^1 \; 1.0111 \times 2^2$
0.00101	$(-1)^0 \; 1.01 \times 2^{-3}$

Primer: (Normalized) Scientific Notation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Sign

$$(-1)^s\ M \times 2^E$$

Significand Base Exponent

Binary Value	**Scientific Notation**
1110110110110	$(-1)^0\ 1.110110110110 \times 2^{12}$
-101.11	$(-1)^1\ 1.0111 \times 2^2$
0.00101	$(-1)^0\ 1.01 \times 2^{-3}$

Primer: (Normalized) Scientific Notation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Fraction

Sign

$$(-1)^s\ M \times 2^E$$

Significand Base Exponent

Binary Value	Scientific Notation
1110110110110	$(-1)^0\ 1.110110110110 \times 2^{12}$
-101.11	$(-1)^1\ 1.0111 \times 2^2$
0.00101	$(-1)^0\ 1.01 \times 2^{-3}$

Primer: (Normalized) Scientific Notation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Fraction

Sign

$(-1)^s\ M \times 2^E$

Significand Base Exponent

If I tell you that there is a number where:

Fraction = 0101

s = 1

E = 10

You could reconstruct the number as $(-1)^1 1.0101 \times 2^{10}$

Primer: Floating Point Representation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Fraction

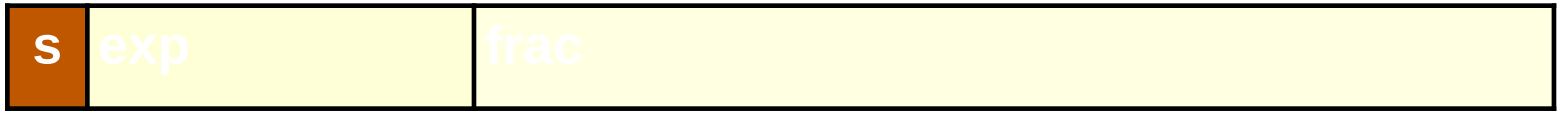
Encoding

MSB s is sign bit s

Sign

$$(-1)^s\ M \times 2^E$$

Significand Base Exponent



Primer: Floating Point Representation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Fraction

Sign

$$(-1)^s\ M \times 2^E$$

Significand Base Exponent

Encoding

MSB *s* is sign bit s

exp field encodes Exponent (but not exactly the same, more later)

Primer: Floating Point Representation

In binary: $(-1)^s\ M\ 2^E$

Normalized form:

$1 <= M < 2$

$M = 1.b_0b_1b_2b_3\ldots$

Fraction

Sign

$$(-1)^s\ M \times 2^E$$

Significand Base Exponent

Encoding

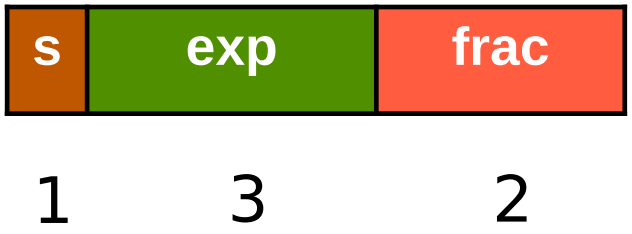
MSB *s* is sign bit s

exp field encodes Exponent (but not exactly the same, more later)

frac field encodes Fraction (but not exactly the same, more later)

6-bit Floating Point Example

$v = (-1)^s M 2^E$



E	exp
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

3 bits can represent exponents from **0 to 7**

E	exp
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
- bias is always $2^{k-1} - 1$, where k is number of exponent bits

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
- bias is always $2^{k-1} - 1$, where k is number of exponent bits

Example when we use 3 bits for exp (i.e., $k = 3$):

- bias = 3

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
- bias is always $2^{k-1} - 1$, where k is number of exponent bits

Example when we use 3 bits for exp (i.e., $k = 3$):

- bias = 3
- If *E* = -2, *exp* is 1 (001_2)

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
- bias is always $2^{k-1} - 1$, where k is number of exponent bits

Example when we use 3 bits for exp (i.e., $k = 3$):

- bias = 3
- If *E* = -2, *exp* is 1 (001_2)
- Reserve 000 and 111 for other purposes (more on this later)

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$

1 3 2

exp has 3 bits, interpreted as an unsigned value

- 3 bits can represent exponents from **0 to 7**
- How about negative exponent?
- Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
- bias is always $2^{k-1} - 1$, where k is number of exponent bits

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

Example when we use 3 bits for exp (i.e., $k = 3$):

- bias = 3
- If $E = -2$, *exp* is 1 (001_2)
- Reserve 000 and 111 for other purposes (more on this later)
- We can now represent exponents from **-2 (exp 001) to 3 (exp 110)**

6-bit Floating Point Example

$v = (-1)^s M 2^E$

frac has 2 bits, append them after “1.” to form M

frac = 10 implies M = 1.10

Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

6-bit Floating Point Example

$v = (-1)^s M 2^E$

frac has 2 bits, append them after “1.” to form M

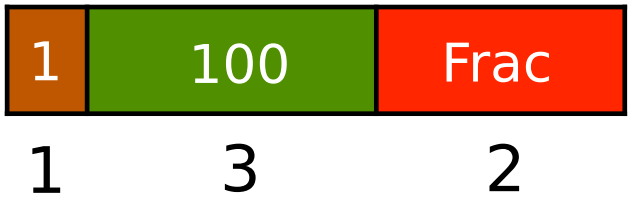
frac = 10 implies M = 1.10

Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

6-bit Floating Point Example

$v = (-1)^s M 2^E$



frac has 2 bits, append them after “1.” to form

frac = 10 implies M = 1.10

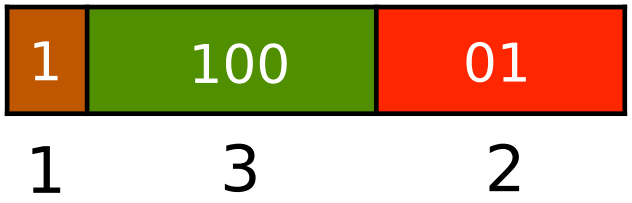
Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

$v = (-1)^s M 2^E$



frac has 2 bits, append them after “1.” to form

frac = 10 implies M = 1.10

Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

“Normalized” Values

$v = (-1)^s M 2^E$

- When: exp ≠ 000…0 and exp ≠ 111…1

- Exponent coded as a biased value: E = Exp – Bias
  - Exp: unsigned value of exp field
  - 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 field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for “free”

Denormalized Values

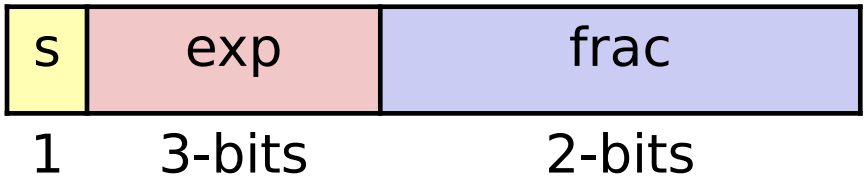
$v = (-1)^s M 2^E$
$E = 1 - Bias$

- Condition: exp = 000…0

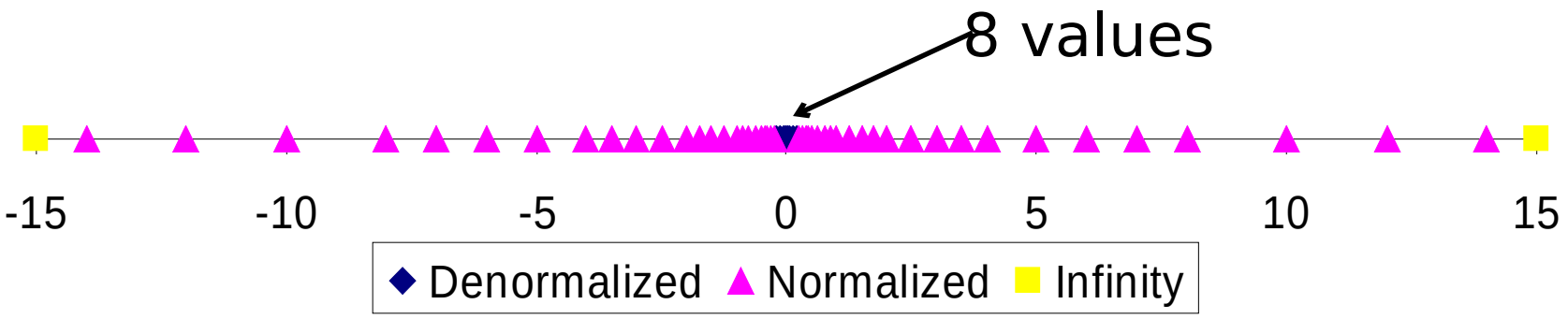
- Exponent value: E = 1 – Bias (instead of E = 0 – Bias)
- Significand coded with implied leading 0: M = 0.xxx…x_2
  - xxx…x: bits of frac
- Cases
  - exp = **000…0**, frac = **000…0**
    - Represents zero value
    - Note distinct values: +0 and –0 (why?)
  - exp = **000…0**, frac ≠ **000…0**
    - Numbers closest to 0.0
    - Equispaced

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.



Special Values

$v = (-1)^s M 2^E$

E	exp	E	exp
-2	000	1	100
-2	001	2	101
-1	010	3	110
0	011	4	111

There are many special values in scientific computing

- +/- ∞, Not-a-Numbers (NaNs) (e.g., 0 / 0, 0 / ∞, ∞ / ∞, sqrt(–1), ∞ - ∞, ∞ x 0, etc.)

exp = 111 is reserved to represent these numbers

exp = 111, frac = 00

- +/- ∞ (depending on the *s* bit). Overflow results.
- Arithmetic on ∞ is exact: 1.0/0.0 = −1.0/−0.0 = + ∞, 1.0/−0.0 = -∞

exp = 111, frac != 00

- Represent NaNs

IEEE 754 Floating Point Standard

Single precision: 32 bits

Double precision: 64 bits

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs (and even GPUs and other processors)

IEEE Floating Point

- IEEE Standard 754
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs (and even GPUs and other processors)
- 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

IEEE 754 Single Precision (32-bit)

$v = (-1)^s \boldsymbol{M}\, 2^{\boldsymbol{E}}$

$bias = 2^{(8-1)}-1 = 127$

$$15213_{10} = 11101101101101_2$$
$$= (-1)^0\, 1.11011011011101_2 \times 2^{13}$$

$$exp = E + bias = 140_{10}$$

Floating Point Computations

The problem: Computing on floating point numbers might produce a result that can’t be precisely represented

Basic idea

- We perform the operation & produce the infinitely precise result
- Make it fit into desired precision
  - Possibly overflow if exponent too large
  - Possibly round to fit into frac

Rounding Modes (Decimal)

Common ones:

- Towards zero (chop)
- Round down (-∞)
- Round up (+∞)

Nearest Even: Round to nearest; if equally near, then to the one having an even least significant digit (bit)

Rounding Mode	**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 even (default)	1	2	2	2	-2

Rounding Modes (Binary Example)

Nearest Even; if equally near, then to the one having an even least significant digit (bit)

Assuming 3 bits for *frac*

Precise Value	**Rounded Value**	**Notes**
1.000011	1.000	1.000 is the nearest (down)
1.000110	1.001	1.001 is the nearest (up)
1.000100	1.000	1.000 is the nearest even (down)
1.001100	1.010	1.010 is the nearest even (up)

Floating Point Addition

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

$1.000 \times 2^{-1} + 11.10 \times 2^{-3}$

Floating Point Addition

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

$1.000 \times 2^{-1} + 11.10 \times 2^{-3}$

align $1.000 \times 2^{-1} + 0.111 \times 2^{-1}$

Floating Point Addition

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

$1.000 \times 2^{-1} + 11.10 \times 2^{-3}$

align $1.000 \times 2^{-1} + 0.111 \times 2^{-1}$

add 1.111×2^{-1}

Floating Point Addition

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

Exact Result: $(-1)^{s}\ M\ \ 2^{E}$

- Sign *s*, significand *M*:
  - Result of signed align & add
- Exponent E: E1
  - Assume E1 > E2

$1.000 \times 2^{-1} + 11.10 \times 2^{-3}$

align $1.000 \times 2^{-1} + 0.111 \times 2^{-1}$

add 1.111×2^{-1}

Floating Point Addition

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

Exact Result: $(-1)^{s}\ M\ \ 2^{E}$

- Sign *s*, significand *M*:
  - Result of signed align & add
- Exponent E: E1
  - Assume E1 > E2

$1.000 \times 2^{-1} + 11.10 \times 2^{-3}$

align $1.000 \times 2^{-1} + 0.111 \times 2^{-1}$

add 1.111×2^{-1}

Fixing

- If $M \geq 2$, 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

Mathematical Properties of FP Add

Commutative? ***Yes***

Associative? ***No***

Overflow and inexactness of rounding

`(3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14`

0 is additive identity? ***Yes***

Every element has additive inverse (negation)? ***Almost***

Except for infinities & NaNs

Monotonicity: $a \geq b \Rightarrow a+c \geq b+c$? ***Almost***

Except for infinities & NaNs

Floating Point Multiplication

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

Floating Point Multiplication

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

Exact Result: $(-1)^{s}\ M\ \ 2^{E}$

- Sign *s*: $s1\ \hat{}\ s2$
- Significand *M*: $M1 \times M2$
- Exponent *E*: $E1 + E2$

Floating Point Multiplication

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

Exact Result: $(-1)^{s}\ M\ \ 2^{E}$

- Sign *s*: $s1\ \hat{}\ s2$
- Significand *M*: $M1 \times M2$
- Exponent *E*: $E1 + E2$

Fixing

- If M ≥ 2, shift M right, increment E
- If E out of range, overflow
- Round M to fit frac precision

Floating Point Multiplication

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

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
- Round M to fit frac precision

Implementation

- Biggest chore is multiplying significands

Mathematical Properties of FP Mult

Multiplication Commutative? ***Yes***

Multiplication is Associative? ***No***

Possibility of overflow, inexactness of rounding

Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`

1 is multiplicative identity? ***Yes***

Multiplication distributes over addition? ***No***

Possibility of overflow, inexactness of rounding

`1e20*(1e20-1e20)= 0.0,  1e20*1e20 – 1e20*1e20 = NaN`

Monotonicity: $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***

Except for infinities & NaNs