Number Systems

CS10003 PROGRAMMING AND DATA STRUCTURES

Number Representation

BINARY

HEXADECIMAL

DECIMAL

Topics to be Discussed

How are numeric data items actually stored in computer memory?

How much space (memory locations) is allocated for each type of data?

- int, float, char, double, etc.

How are characters and strings stored in memory?

- Already discussed.

Number System: The Basics

We are accustomed to using the so-called *decimal number system*.

- Ten digits :: 0,1,2,3,4,5,6,7,8,9
- Every digit position has a weight which is a power of 10.
- Base or radix is 10.

Example:

$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

$250.67 = 2 \times 10^2 + 5 \times 10^1 + 0 \times 10^0 + 6 \times 10^{-1} + 7 \times 10^{-2}$

Binary Number System

Two digits:

- 0 and 1.
- Every digit position has a weight which is a power of 2.
- *Base* or *radix* is 2.

Example:

$110 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$

$101.01 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$

Binary-to-Decimal Conversion

Each digit position of a binary number has a weight.

- Some power of 2.

A binary number:

$$B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0 \,.\, b_{-1}\, b_{-2} \ldots b_{-m}$$

Corresponding value in decimal:

$$D = \sum_{i=-m}^{n-1} b_i\, 2^i$$

Examples

1. 101011 $\Rightarrow$ $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 43$

   $(101011)_2 = (43)_{10}$

2. .0101 $\Rightarrow$ $0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = .3125$

   $(.0101)_2 = (.3125)_{10}$

3. 101.11 $\Rightarrow$ $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 5.75$

   $(101.11)_2 = (5.75)_{10}$

Decimal-to-Binary Conversion

Consider the integer and fractional parts separately.

For the integer part,

- Repeatedly divide the given number by 2, and go on accumulating the remainders, until the number becomes zero.
- Arrange the remainders *in reverse order*.

For the fractional part,

- Repeatedly multiply the given fraction by 2.
    - Accumulate the integer part (0 or 1).
    - If the integer part is 1, chop it off.
- Arrange the integer parts *in the order* they are obtained.

Example 1 :: 239

```
2      239
2      119     --- 1
2      59      --- 1
2      29      --- 1
2      14      --- 1
2      7       --- 0
2      3       --- 1
2      1       --- 1
2      0       --- 1
```

$(239)_{10} = (11101111)_2$

Example 2 :: 64

```
2    64
2    32       --- 0
2    16       --- 0
2    8        --- 0
2    4        --- 0
2    2        --- 0
2    1        --- 0
2    0        --- 1
```

$(64)_{10} = (1000000)_2$

Example 3 :: .634

```
.634   x   2    =    1.268
.268   x   2    =    0.536
.536   x   2    =    1.072
.072   x   2    =    0.144
.144   x   2    =    0.288
:
:
```

$$(.634)_{10} = (.10100\ldots)_2$$

Example 4 :: 37.0625

$(37)_{10} = (100101)_2$

$(.0625)_{10} = (.0001)_2$

$\therefore (37.0625)_{10} = (100101\,.\,0001)_2$

Hexadecimal Number System

A compact way of representing binary numbers.

16 different symbols (radix = 16).

0	$\Rightarrow$	0000	8	$\Rightarrow$	1000
1	$\Rightarrow$	0001	9	$\Rightarrow$	1001
2	$\Rightarrow$	0010	A	$\Rightarrow$	1010
3	$\Rightarrow$	0011	B	$\Rightarrow$	1011
4	$\Rightarrow$	0100	C	$\Rightarrow$	1100
5	$\Rightarrow$	0101	D	$\Rightarrow$	1101
6	$\Rightarrow$	0110	E	$\Rightarrow$	1110
7	$\Rightarrow$	0111	F	$\Rightarrow$	1111

Binary-to-Hexadecimal Conversion

For the integer part,

- Scan the binary number from *right to left*.
- Translate each group of four bits into the corresponding hexadecimal digit.
  - Add *leading* zeros if necessary.

For the fractional part,

- Scan the binary number from *left to right*.
- Translate each group of four bits into the corresponding hexadecimal digit.
  - Add *trailing* zeros if necessary.

Examples

1. $(\underline{1011}\ \underline{0100}\ \underline{0011})_2 = (B43)_{16}$
2. $(\underline{10}\ \underline{1010}\ \underline{0001})_2 = (2A1)_{16}$
3. $(.\underline{1000}\ \underline{010})_2 = (.84)_{16}$
4. $(\underline{101}\ .\ \underline{0101}\ \underline{111})_2 = (5.5E)_{16}$

Hexadecimal-to-Binary Conversion

Translate every hexadecimal digit into its 4-bit binary equivalent.

- **Discard leading and trailing zeros if desired.**

Examples:

$(3A5)_{16} = (0011\ 1010\ 0101)_2$

$(12.3D)_{16} = (0001\ 0010\ .\ 0011\ 1101)_2$

$(1.8)_{16} = (0001\ .\ 1000)_2$

Representation of Unsigned and Signed Integers

Unsigned Binary Numbers

An n-bit binary number

$B = b_{n-1}b_{n-2} \dots b_2b_1b_0$

- 2^n distinct combinations are possible, 0 to 2^n-1.

For example, for n = 3, there are 8 distinct combinations.

- 000, 001, 010, 011, 100, 101, 110, 111

Range of numbers that can be represented

$n = 8 \Rightarrow 0$ to 2^8-1 (255)

$n = 16 \Rightarrow 0$ to $2^{16}-1$ (65535)

$n = 32 \Rightarrow 0$ to $2^{32}-1$ (4294967295)

Signed Integer Representation

Many of the numerical data items that are used in a program are signed (positive or negative).

- Question:: How to represent sign?

Three possible approaches:

a) Sign-magnitude representation

b) One’s complement representation

c) Two’s complement representation

Sign-magnitude Representation

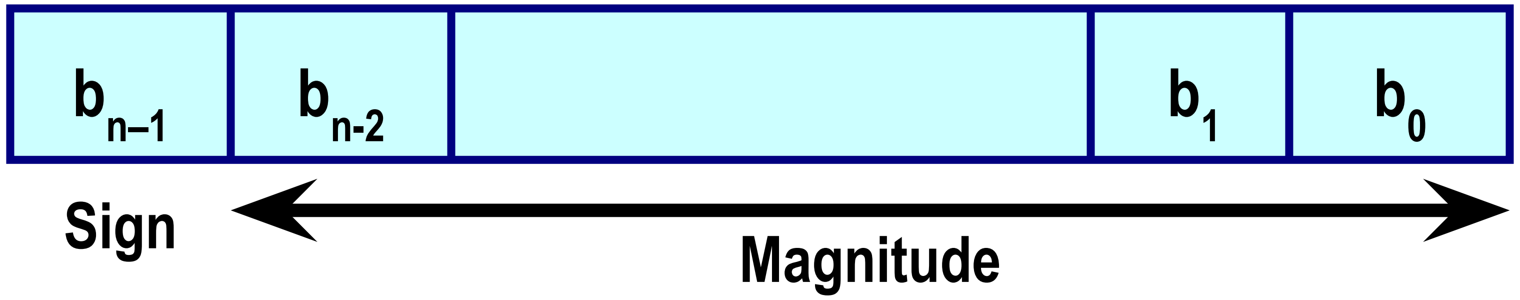
For an n-bit number representation

- The most significant bit (MSB) indicates sign

  0 $\Rightarrow$ positive

  1 $\Rightarrow$ negative

- The remaining n-1 bits represent magnitude.



Contd.

Range of numbers that can be represented:

Maximum :: $+ (2^{n-1} - 1)$

Minimum :: $- (2^{n-1} - 1)$

A problem:

Two different representations of zero.

$+0 \Rightarrow$ 0 000….0

$-0 \Rightarrow$ 1 000….0

One’s Complement Representation

Basic idea:

- Positive numbers are represented exactly as in sign-magnitude form.
- Negative numbers are represented in 1’s complement form.

How to compute the 1’s complement of a number?

- Complement every bit of the number (1→0 and 0→1).
- MSB will indicate the sign of the number.

  $0 \Rightarrow$ positive

  $1 \Rightarrow$ negative

Example :: n = 4

0000	⇒	+0	1000	⇒	-7
0001	⇒	+1	1001	⇒	-6
0010	⇒	+2	1010	⇒	-5
0011	⇒	+3	1011	⇒	-4
0100	⇒	+4	1100	⇒	-3
0101	⇒	+5	1101	⇒	-2
0110	⇒	+6	1110	⇒	-1
0111	⇒	+7	1111	⇒	-0

To find the representation of, say, –4, first note that

+4 = 0100

−4 = 1's complement of 0100 = 1011

Contd.

Range of numbers that can be represented:

Maximum :: $+ (2^{n-1} - 1)$

Minimum :: $- (2^{n-1} - 1)$

A problem:

Two different representations of zero.

$+0 \Rightarrow$ 0 000…0

$-0 \Rightarrow$ 1 111…1

Advantage of 1's complement representation

- Subtraction can be done using addition.
- Leads to substantial saving in circuitry.

Two's Complement Representation

Basic idea:

- **Positive numbers are represented exactly as in sign-magnitude form.**
- **Negative numbers are represented in 2's complement form.**

How to compute the 2's complement of a number?

- **Complement every bit of the number (1⇒0 and 0⇒1), and then *add one* to the resulting number.**
- **MSB will indicate the sign of the number.**

  0 ⇒ positive

  1 ⇒ negative

Example :: n = 4

0000	⇒	+0	1000	⇒	-8
0001	⇒	+1	1001	⇒	-7
0010	⇒	+2	1010	⇒	-6
0011	⇒	+3	1011	⇒	-5
0100	⇒	+4	1100	⇒	-4
0101	⇒	+5	1101	⇒	-3
0110	⇒	+6	1110	⇒	-2
0111	⇒	+7	1111	⇒	-1

To find the representation of, say, –4, first note that

+4 = 0100

−4 = 2’s complement of 0100 = 1011+1 = 1100

Contd.

Range of numbers that can be represented:

Maximum :: $+ (2^{n-1} - 1)$

Minimum :: $- 2^{n-1}$

Advantage:

- *Unique representation of zero*.
- Subtraction can be done using addition.
- Leads to substantial saving in circuitry.

Almost all computers today use the 2's complement representation for storing negative numbers.

Contd.

In C, typically:

- char
  - 8 bits $\Rightarrow$ $+(2^7 - 1)$ to -2^7
- short int
  - 16 bits $\Rightarrow$ $+(2^{15}-1)$ to -2^{15}
- int
  - 32 bits $\Rightarrow$ $+(2^{31}-1)$ to -2^{31}
- long int
  - 64 bits $\Rightarrow$ $+(2^{63}-1)$ to -2^{63}

Binary operations

Addition / Subtraction using addition

Binary addition

Rules for adding two bits

0 + 0 is 0

0 + 1 is 1

1 + 0 is 1

1 + 1 is 10, that is, 0 with carry of 1

Rules for adding three bits

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Subtraction Using Addition :: 1’s Complement

How to compute A – B ?

- Compute the 1’s complement of B (say, B_1).
- Compute $R = A + B_1$
- If the carry obtained after addition is ‘1’
    - Add the carry back to R (called *end-around carry*).
    - That is, R = R + 1.
    - The result is a positive number.

  Else
    - The result is negative, and is in 1’s complement form.

Example 1 :: 6 – 2

1's complement of 2 = 1101

```
  6 ::  0110
 –2 ::  1101
      ------
      1 0011
           1
        0100   ⇒  +4
      ------
```

A

B_1

R

End-around carry

Assume 4-bit representations.

Since there is a carry, it is added back to the result.

The result is positive.

Example 2 :: 3 – 5

1's complement of 5 = 1010

```
 3  ::  0011
–5  ::  1010
       ------
         1101
```

⇩

–2

A

B_1

R

Assume 4-bit representations.

Since there is no carry, the result is negative.

1101 is the 1's complement of 0010, that is, it represents –2.

Subtraction Using Addition :: 2’s Complement

How to compute A – B ?

- Compute the 2’s complement of B (say, B_2).
- Compute $R = A + B_2$
- If the carry obtained after addition is ‘1’
    - Ignore the carry.
    - The result is a positive number.

  Else
    - The result is negative, and is in 2’s complement form.

Example 1 :: 6 – 2

2's complement of 2 = 1101 + 1 = 1110

```
 6 ::  0110
–2 ::  1110
     ------
      1 0100
```

$\Downarrow$ +4

Ignore carry

A
B_2
R

Assume 4-bit representations.

Presence of carry indicates that the result is positive.

No need to add the end-around carry like in 1's complement.

Example 2 :: 3 – 5

2’s complement of 5 = 1010 + 1 = 1011

```
 3  ::  0011
–5  ::  1011
      ------
         1110
```

-2

A

B_2

R

Assume 4-bit representations.

Since there is no carry, the result is negative.

1110 is the 2’s complement of 0010, that is, it represents –2.

2’s complement arithmetic: More Examples

- Example 1: 18 – 11 = ?
- 18 is represented as 00010010
- 11 is represented as 00001011
  - 1’s complement of 11 is 11110100
  - 2’s complement of 11 is 11110101
- Add 18 to 2’s complement of 11

```
   00010010
+  11110101
----------------
   00000111 (with a carry of 1
             which is ignored)
```

00000111 is 7

2’s complement arithmetic: More Examples

- Example 2: 7 – 9 = ?
- 7 is represented as 00000111
- 9 is represented as 00001001
  - 1’s complement of 9 is 11110110
  - 2’s complement of 9 is 11110111
- Add 7 to 2’s complement of 9

```
   00000111
+  11110111
----------------
   11111110 (with a carry of 0
                which is ignored)
```

11111110 is –2

Overflow and Underflow

Adding two +ve (-ve) numbers should not produce a –ve (+ve) number.
If it does, overflow (underflow) occurs

Another equivalent condition :
carry in and carry out from Most Significant Bit (MSB) differ.

```
(64)  01000000
( 4)  00000100
      --------------
 (68) 01000100
```

```
(64)  01000000
(96)  01100000
      --------------
(-96) 10100000
```

```
carry (out)(in)
        0    0
```

```
carry (out)(in)
        0    1
```

differ:

overflow

Floating-point number representation

The IEEE 754 Format

Fixed Point Representation

- Consists of a whole or integral part and a fractional part
- The two parts are separated by a binary point
- For k whole digits and l fractional digits, the value obtained is:

$$x = \sum_{i=-l}^{k-1} x_i \, 2^i = (x_{k-1} x_{k-2} \ldots x_0 x_{-1} x_2 \ \ldots x_{-l})_2$$

- In a $(k + l)$-bit representation,
  numbers from 0 to $(2^k - 2^{-l})$ can be represented
- Hence, k decided the range and l decides the precision
- As $(k + l)$ is constant, we have a tradeoff.

Limitations of using Fixed Point Representation

- Fixed point representations are hence not good for applications dealing with very large (needing a larger range), and extremely small numbers (and hence need precision) at the same time
- Consider the $(8+8)$-bit fixed point numbers
  - $x = (0000\,0000\,.\,0000\,1001)_2$ → small number
  - $y = (1001\,0000\,.\,0000\,0000)_2$ → large number
- Points to note:
  - The relative representation error due to truncation or rounding of digits beyond the 8th position is significant for x, but it is less severe for y
  - On the other hand, neither y^2 , nor $\frac{y}{x}$ is representable in this format

Floating point numbers address this issue, and is made of fixed point signed-magnitude number and an accompanying scale factor.

Normalization

Write a positive non-zero number as

$$1.b_1b_2b_3\ldots b_k \times 2^E = (1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + \ldots + b_k \times 2^{-k}) \times 2^E$$

Examples

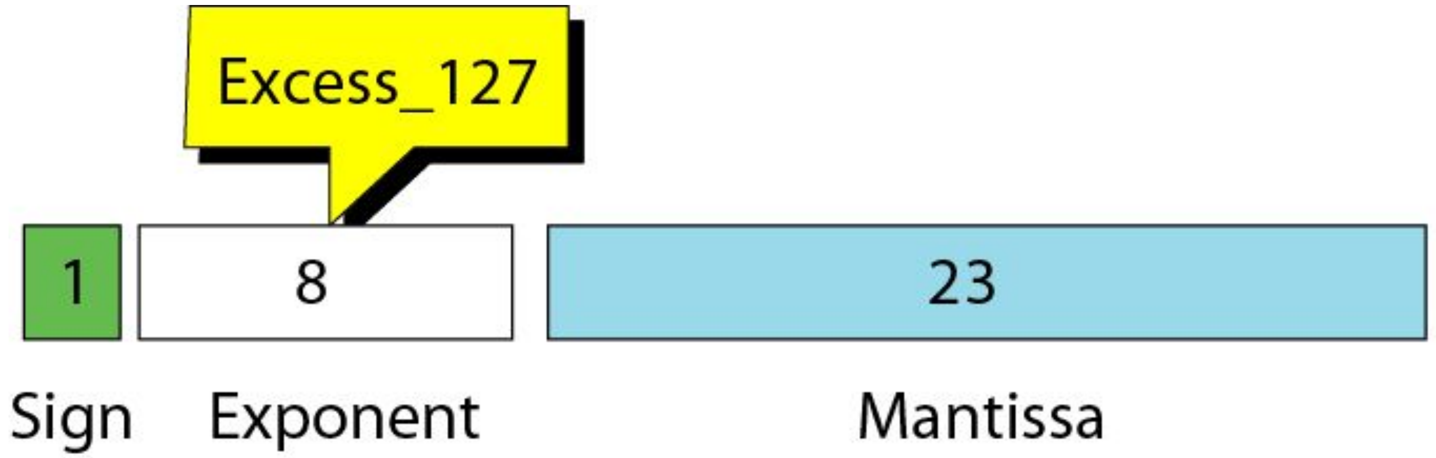
Original Number	***Move***	***Normalized Representation***
+1010001.1101	$\leftarrow$ 6	$+ 1.0100011101 \times 2^6$
–111.000011	$\leftarrow$ 2	$- 1.11000011 \times 2^2$
+0.00000111001	6 $\rightarrow$	$+ 1.11001 \times 2^{-6}$
–0.001110011	3 $\rightarrow$	$- 1.110011 \times 2^{-3}$

Normalized numbers in Single Precision Format

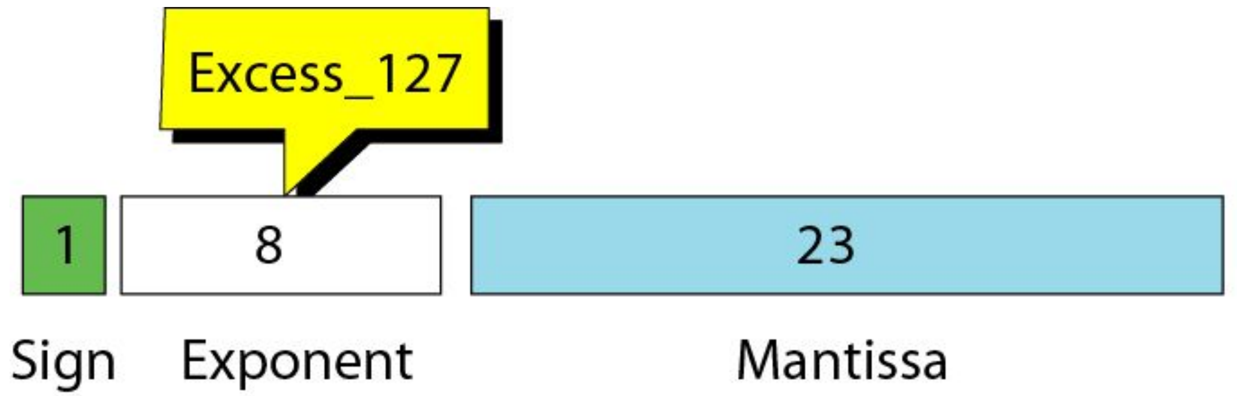
The normalized numbers are

$$(-1)^{s}\ 1.f \times 2^{E-127}.$$

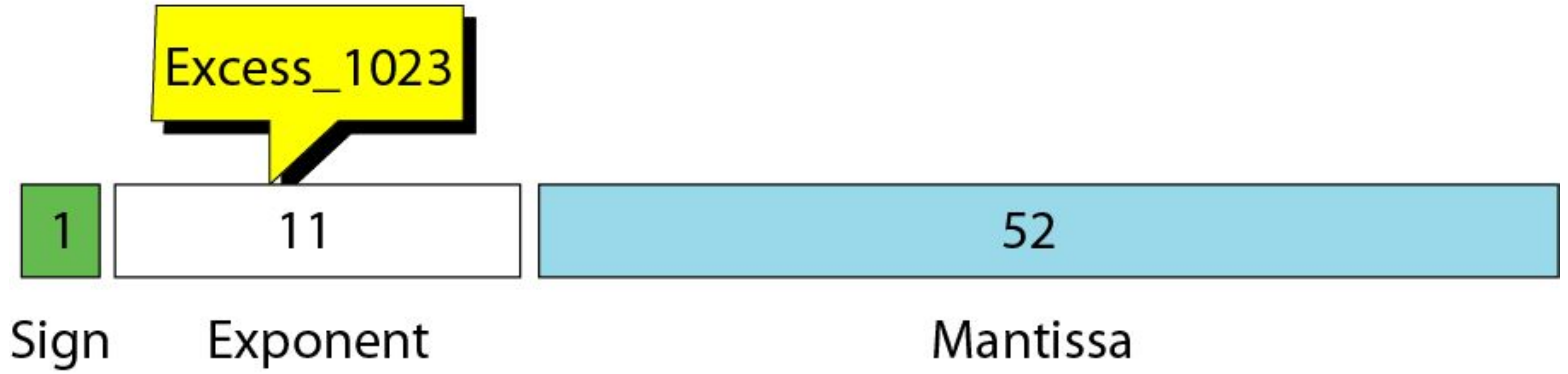
Here, s is the sign bit, *f* is the mantissa (fractional part), and E is the exponent (plus 127). The 1 before the binary point is not stored.



IEEE standards for floating-point representation



a. Single Precision



b. Double Precision

Example

Show the representation of the normalized number + 1.01000111001 x 2^6.

Solution

The sign is positive. The Excess_127 representation of the exponent is 133. You add extra 0s on the right to make it 23 bits. The number in memory is stored as:

0 10000101 01000111001000000000000

Example of floating-point representation

Number	***Sign***	***Exponent***	***Mantissa***
-1.11000011×2^{2}	1	10000001	11000011000000000000000
$+1.11001 \times 2^{-6}$	0	01111001	11001000000000000000000
-1.110011×2^{-3}	1	01111100	11001100000000000000000

Example

Interpret the following 32-point floating-point number

1 01111100 11001100000000000000000

Solution

The sign is negative.
The exponent is 124 – 127 = –3
The number is

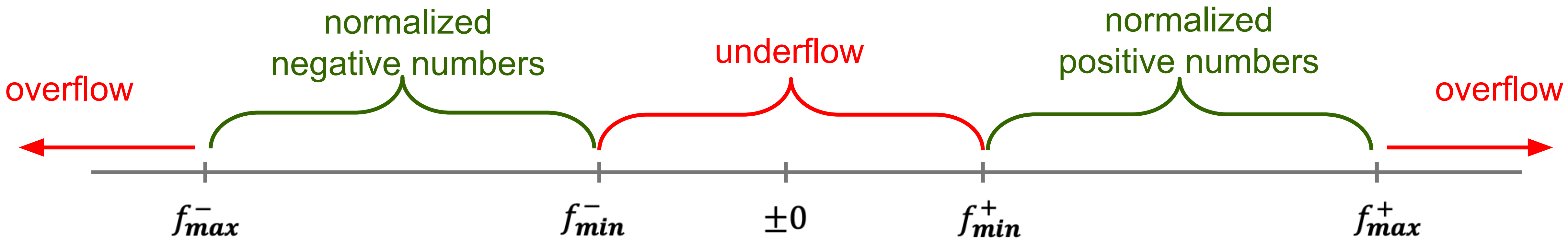
$-1.110011 \times 2^{-3} = -(1 + (1/2) + (1/2)^2 + (1/2)^5 + (1/2)^6) \times 2^{-3}$

$= 1.796875 \times 2^{-3} = 0.224609375.$

Range of normalized numbers

- $f_{max}^{+} = (1.111 \ldots 1) \times 2^{254-127}$
  - $E = 0$ is reserved for zero (with $f = 0$) and denormalized numbers (with $f \neq 0$).
  - $E = 255$ is reserved for $\pm\infty$ (with $f = 0$) and for NaN (Not a Number) (with $f \neq 0$).

- Thus, $f_{max}^{+} = \left(2 - 2^{-23}\right) \times 2^{127} = \left(1 - 2^{-24}\right) \times 2^{128}$
- Similarly, $f_{min}^{+} = (1.0) \times 2^{1-127} = 2^{-126}$



- The exponent bias and significand range were selected so that the reciprocal of all normalized numbers can be represented without overflow. (in particular f_{min}^{+}).

Denormalized numbers

- These numbers correspond to the 8-bit exponent E = 0
- If M denotes the 23-bit mantissa, then the number is to be interpreted as:

$$(-1)^S \times 0.M \times 2^{-126} = M \times 2^{-149}$$

- The largest positive denormalized number is 11111111111111111111111 x $2^{-149} = (2^{23} - 1) \times 2^{-149} = 2^{-126} - 2^{-149}$. This is slightly smaller than the smallest normalized number.
- For each decrement of M by 1, the value of the denormalized number reduces by 2^{-149}. The smallest positive denormalized number is 2^{-149} (corresponding to M = 00000000000000000000001).
- When all bits of M are zero, we get the representation of +0 as a string of 32 zero bits.
- –0 is represented as 1 followed by 31 zero bits.
- This process of going from 2^{-126} to 0 is called gradual underflow.

Special numbers

These numbers correspond to the 8-bit exponent E = 255 (all 1 bits).

0 11111111 00000000000000000000000	+Inf
1 11111111 00000000000000000000000	–Inf
0 11111111 Any non-zero value	NaN
1 11111111 Any non-zero value	NaN

Inf means Infinity.

NaN means Not a Number.

A program to view the floating-point representation

```c
#include <stdio.h>

void prn32 ( unsigned a )
{
   int i;

   for (i=31; i>=0; --i) {
       printf("%d", (a & (1U << i)) ? 1 : 0 );
       if ((i == 31) || (i == 23)) printf(" ");
   }
   printf("\n");
}
```

```c
int main ()
{
   float x = -123.45;
   unsigned *p;

   p = (unsigned *)&x;
   prn32(*p);

   return 0;
}
```

Output

```
1 10000101 11101101110011001100110
```

Check for correctness

- 123 = 64 + 32 + 16 + 8 + 2 + 1 = $2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0$ = 1111011

- 0.45 x 2 = 0.90, 0.90 x 2 = 1.80, 0.80 x 2 = 1.60, 0.60 x 2 = 1.20, 0.20 x 2 = 0.40, 0.40 x 2 = 0.80, …

- 0.45 = $0.011100\underline{1100}$

- $$\begin{aligned} 123.45 &= 1111011.011100\underline{1100} \approx 1111011.01110011001100110 \\ &= 1.11101101110011001100110 \times 2^{6} \\ &= 1.11101101110011001100110 \times 2^{133-127} \\ &= 1.11101101110011001100110 \times 2^{(128+4+1)-127} \\ &= 1.11101101110011001100110 \times 2^{10000101-127} \end{aligned}$$

- What we should have: `1 10000101 11101101110011001100110`
- What the program gives: `1 10000101 11101101110011001100110`