SCC131: Digital Systems

Topic 3: information coding 2

From decimal to binary…

- Decimal-oriented circuitry is too complex to implement in hardware
  - Can we find a better ***radix*** than 10?
- It's "easy" (according to Alan Turing!) to design circuitry with **two** states, **on** and **off**
  - Using a radix of 2 is called ***binary*** (or base 2)
  - Only two digits: 0 and 1

Most Significant Bit ... ***Least Significant Bit***

n x 128	n x 64	n x 32	n x 16	n x 8	n x 4	n x 2	n x 1
$n \times 2^7$	$n \times 2^6$	$n \times 2^5$	$n \times 2^4$	$n \times 2^3$	$n \times 2^2$	$n \times 2^1$	$n \times 2^0$

We call a binary digit a ***bit***, and 8 bits a ***byte***

What changes with binary?

- Everything works exactly as we saw earlier, except that we use multiples of 2 rather than multiples of 10
- Remember, each column can contain a number between *0* and *radix–1* (inclusive)
  - So, in binary each column can only contain either a 0 or a 1 (for decimal: 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9)

128	64	32	16	8	4	2	1
0	0	**1**	0	**1**	0	0	**1**

$$\mathbf{32 + 8 + 1 = 41}_{\text{decimal}}$$

Worked example of decimal to binary number conversion

- Convert $35_{decimal}$ to binary
  - Write out the powers of 2:
    … 64 32 16 8 4 2 1
  - Highlight the column with the largest number that is smaller than the number we are converting:
    64 **32** 16 8 4 2 1
  - Subtract this number from the number we are converting:
    35-32 = 3
  - Iterate the above two steps until the subtraction results in 0:
    64 **32** 16 8 4 **2** 1
    64 **32** 16 8 4 **2** **1**
  - Map each highlighted column to a 1, the rest to a 0:
    64 **32** 16 8 4 **2** **1** → 100011 (6-bit binary)
  - So $35_{decimal} = 100011_{binary}$

Worked example of decimal to binary number conversion

- Convert $189_{decimal}$ to binary
  - Write out the powers of 2:
    … 128 64 32 16 8 4 2 1
  - Highlight the column with the largest number that is smaller than the number we are converting:
    128 64 32 16 8 4 2 1
  - Subtract this number from the number we are converting:
    189 - 128 = 61
  - Iterate the above two steps until the subtraction results in 0:
    128 64 32 16 8 4 2 1
    128 64 **32** 16 8 4 2 1 (32: 61 – 32 = 29)
    128 64 **32** **16** 8 4 2 1 (16: 29 – 16 = 13)
    128 64 **32** **16** **8** 4 2 1 (8: 13 – 8 = 5)
    128 64 **32** **16** **8** **4** 2 1 (4: 5 – 4 = 1)
    128 64 **32** **16** **8** **4** 2 **1** (1: 1 – 1 = 0)

  - Map each highlighted column to a 1, the rest to a 0:
    128 64 **32** **16** **8** **4** 2 **1** → 10111101 (8-bit binary)
  - So $189_{decimal} = 10111101_{binary}$

Worked example of binary to decimal number conversion

- Convert 100011_{binary} to decimal
  - Write out the powers of two, as many as we have digits in our source number:
    32 16 8 4 2 1
  - Highlight the columns where there is a 1 in our source number:
    32 16 8 4 **2 1**
  - Add up the highlighted numbers:
    32+2+1 = 35
  - So $100011_{binary} = 35_{decimal}$

Worked examples of binary to decimal number conversion

- Convert 10111101_{binary} to decimal
  - Write out the powers of two, as many as we have digits in our source number:
    128 64 32 16 8 4 2 1
  - Highlight the columns where there is a 1 in our source number:
    128 64 **32 16 8 4** 2 **1**
  - Add up the highlighted numbers:
    128+32+16+8+4+1 = 189
  - So $10111101_{binary} = 189_{decimal}$

Worked example of binary addition

- Add 100011_{binary} to 101011_{binary}

```
100011
101011 +
----------
??????
```

- Start at rightmost end: $1 + 1 = 2_{decimal} = 10_{binary}$
  – So, "carry the 1"and put a 0 in the rightmost result column
- Now do next-right column: $1 + 1 + 1_{carry} = 3_{decimal} = 11_{binary}$
  – So, "carry the 1"and put a 1 in the next-right result column
- Now the next column: $0 + 0 + 1_{carry} = 1_{decimal} = 1_{binary}$
  - So, no carry, and put a 1 in the result column
- And so on…

Worked example: fixed point decimal to fixed point binary

Lets convert 11.375_{10} to binary.

11 . 375

2^3=8	**2^2=4**	**2^1=2**	**2^0=1**

$$11 = 8 + 0 + 2 + 1$$

1 0 1 1

1/2 = 0.5	**1/4= 0.25**	**1/8 = 0.125**	**1/16= 0.0625**

- Can we remove 0.5 from 0.375? No => **0**
- Can we remove 0.25 from 0.375? Yes we get 0.125 => **1**
- Can we remove 0.125 from 0.125? Yes => **1**

And we are left with 0.125-0.125 = **0**

0 1 1 0

Final answer :1011.0110_2 .

Let's try some more radices...

Binary *(base 2)*

n x 128	n x 64	n x 32	n x 16	n x 8	n x 4	n x 2	n x 1
n x 2^7	n x 2^6	n x 2^5	n x 2^4	n x 2^3	n x 2^2	n x 2^1	n x 2^0

Octal *(base 8)*

n x 32768	n x 4096	n x 512	n x 64	n x 8	n x 1
n x 8^5	n x 8^4	n x 8^3	n x 8^2	n x 8^1	n x 8^0
$= n \times 2^{15}$	$= n \times 2^{12}$	$= n \times 2^{9}$	$= n \times 2^{6}$	$= n \times 2^{3}$	$= n \times 2^{0}$

Hexadecimal *(base 16)*

n x 65536	n x 4096	n x 256	n x 16	n x 1
n x 16^4	n x 16^3	n x 16^2	n x 16^1	n x 16^0
$= n \times 2^{16}$	$= n \times 2^{12}$	$= n \times 2^{8}$	$= n \times 2^{4}$	$= n \times 2^{0}$

Notice the nice, regular, binary (2^n) multipliers for octal and hex

Decimal to octal (via binary)

$$\mathbf{41_{decimal} = 32_{decimal} + 8_{decimal} + 1_{decimal}}$$

Groups of 3 bits

256	128	64	32	16	8	4	2	1
0	0	0	**1**	0	**1**	0	0	**1**

4	2	1	4	2	1	4	2	1
0	0	0	1	0	1	0	0	1

0	**5**	**1**
(0 x 4) + (0 x 2) + (0 x 1)	(1 x 4) + (0 x 2) + (1 x 1)	(0 x 4) + (0 x 2) + (1 x 1)

$$\mathbf{= 051_{octal}}$$

Decimal to hexadecimal (via binary)

$$\mathbf{41_{decimal} = 32_{decimal} + 8_{decimal} + 1_{decimal}}$$

Groups of 4 bits

128	*64*	*32*	*16*	*8*	*4*	*2*	*1*
0	0	**1**	0	**1**	0	0	**1**

8	*4*	*2*	*1*	*8*	*4*	*2*	*1*
0	0	1	0	1	0	0	1

2	**9**
(0 x 8) + (0 x 4) + (1 x 2) + (0 x 1)	(1 x 8) + (0 x 4) + (0 x 2) + (1 x 1)

$$\mathbf{= 29_{hex}}$$

Hexadecimal digits beyond 0..9

- We can hold numbers bigger than 9 in four binary digits (bits)
  - So we need more digit symbols for hex!

0 .. 9 as decimal, then…

Radix - 1

$10_{decimal} \rightarrow A_{hex}$
$11_{decimal} \rightarrow B_{hex}$
$12_{decimal} \rightarrow C_{hex}$
$13_{decimal} \rightarrow D_{hex}$
$14_{decimal} \rightarrow E_{hex}$
$15_{decimal} \rightarrow F_{hex}$

8's	4's	2's	1's
1	0	1	1

= B

$(1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1) = 11_{decimal}$

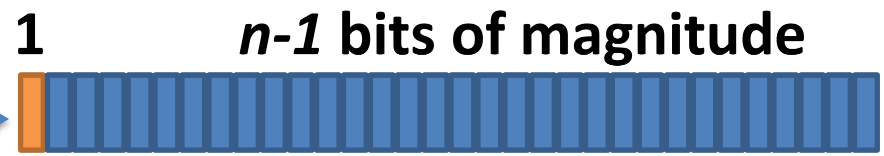
We could have chosen any symbols for the extra digits, but A to F are "convenient"

Why octal and hexadecimal?

- You may be wondering what is the *point* of octal and hex!
- Octal and hex, especially hex, are used very widely as a human-friendly way of dealing with binary bit patterns.
- You'll soon get to thinking of unwieldy things like: **1111111111111111(base 2)**, as more straightforward things like: **FFFF(base 16)**
  - (This is a 16-bit pattern; recall that one hex digit corresponds to a group of 4 bits; so 4 hex digits)

Negative numbers in binary

- Approaches seen so far:
  1. Sign and magnitude



  2. Excess *n*
     - Same approach as seen previously: we code as number + excess

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0

     - E.g. excess 16 for 5 columns (base 2)

       [for b columns, excess $2^b/2 = 2^{b-1}$ works well (as seen previously) so for b=5 columns, use excess 2^4 = excess 16]

Arithmetic in binary

Dec	*Binary*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

- Everything seems to work as expected...

2_{dec}	**0**	**1**	**0**	
+ 1_{dec}	**0**	**0**	**1**	
	0	**1**	**1**	**= 3**

2_{dec}	**0**	**1**	**0**	
+ 2_{dec}	**0**	**1**	**0**	
	1	**0**	**0**	**= 4**

Carry

…But we have to be wary of our old friend *overflow*

	6	**1**	**1**	**0**	
+					
	2	**0**	**1**	**0**	
	1	**0**	**0**	**0**	**= 0**

Carry

Arithmetic Overflow

… ***And*** here’s a problem with excess *n*

- We already saw a couple of problems with sign and magnitude
    - We sacrifice a digit column for the sign indicator
    - We have two representations for zero
- Well, here’s more bad news: there’s a problem with excess *n* when it comes to arithmetic

$$x - y \quad \neq \quad x + (-y) \qquad !!??$$

The problem with excess *n, contd.*

- Let's see if **(0 + -1)** is as expected (in excess 4)

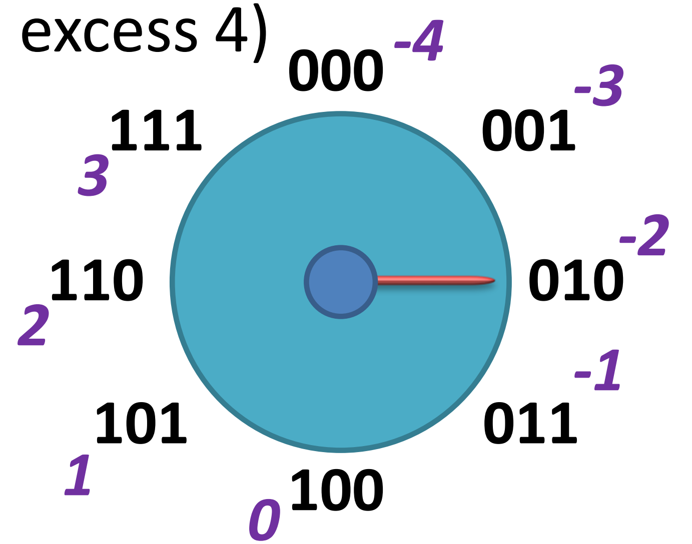
$=\quad 0_{decimal} + -1_{decimal}$

$\quad = 100_{excess4} + 011_{excess4}$

$\quad = 111_{excess4}$ *[no overflow…]*

$\quad = 3_{decimal}$

=> 0 + -1 = 3 !? ***X*** *wrong!*



"to code *m* using excess 4 we store *m* + 4"

- This is very unfortunate: if we can't treat ***x – y*** as ***x + -y*** *we will need extra hardware (i.e. hardware for subtraction as well as for addition)!*

Introducing 2's complement

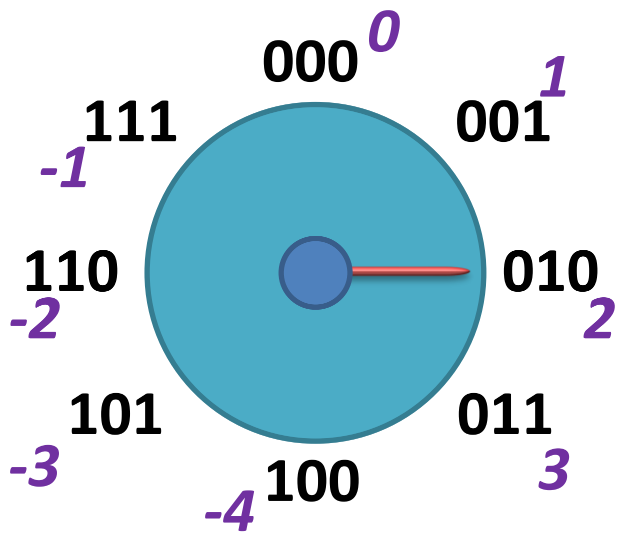
- Two's complement is achieved by (for negative number):

Step 1: starting with the equivalent positive number.

Step 2: inverting (or flipping) all bits – changing every 0 to 1, and every 1 to 0;

Step 3: adding 1 to the entire inverted number, ignoring any overflow.

Calculate 2's complement



Negative Number	*Corresponding Positive binary*	*Invert bits*	*Add 1*
-1	001	110	111
-2	010	101	110
-3	011	100	101
-4	100	011	100

2's complement

A positive number's 2's complement is itself

2’s complement *contd.*

- So, let’s try...

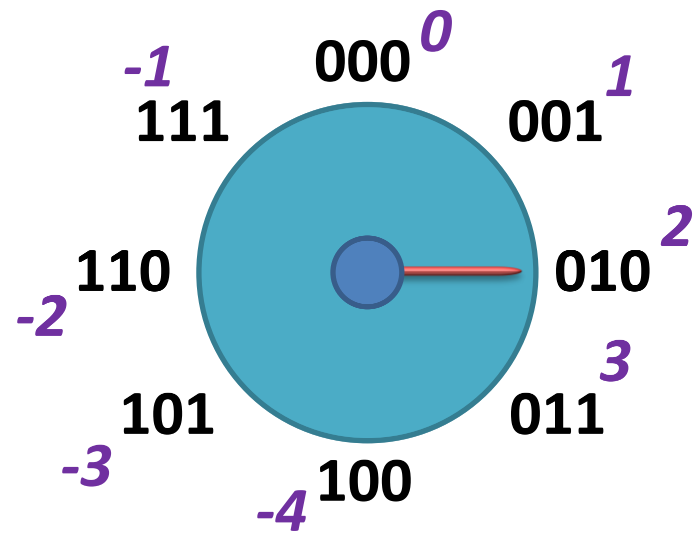
	0	0	0	0	
+	-1	1	1	1	
		1	1	1	= -1

+	3	0	1	1	
	-4	1	0	0	
		1	1	1	= -1

Ignoring the carry here

	2	0	1	0	
+	-2	1	1	0	
	1	0	0	0	= 0

Carry



2’s complement

So, ***x* + -*y* seems to be same as *x* – *y***

...so, adder hardware can handle both addition and subtraction

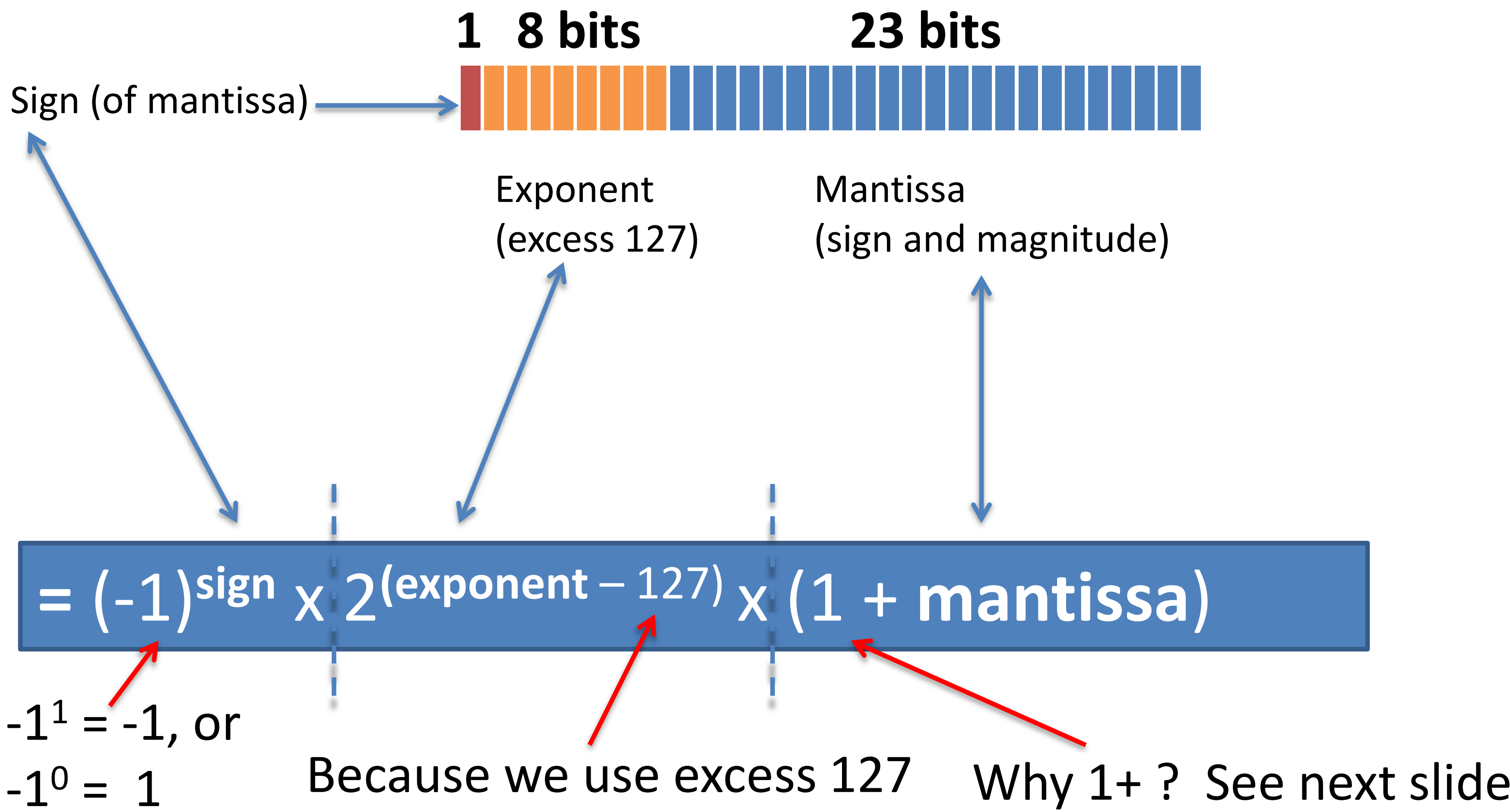
IEEE 754 Floating Point

- This is the standard floating point representation that is used by almost all modern computers
- The mantissa is coded using **sign and magnitude**
  - Although we don't store the most significant bit (*see later*)
- The exponent is coded in **excess n**
  - For an exponent held in *b* bits, n = (2^{b-1}) **− 1**

    Different from what we've used so far
  - So, use excess 127 for an 8-bit exponent...

    [$(2^{8-1}) - 1 \quad = \quad 2^7 - 1 \quad = \quad 127$]
- There are multiple formats available for different "precisions": half, **single**, double, quadruple

The “single precision” format (32 bits)



Normalising the mantissa: a space saving optimisation (saves 1 bit)

- IEEE 754 always *normalises* the mantissa to ***1.xxx***... and just stores the fractional part
  - Leaves the "1." part as implicit: there's actually no need to store it explicitly!
- (It is *always possible* to normalise such that the most significant [first] bit is a 1)

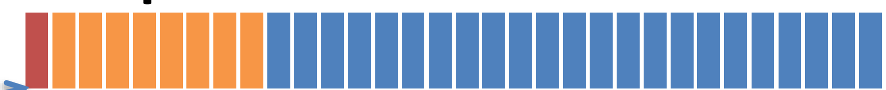
Assumed bit, not explicitly represented

$$= (-1)^{\mathbf{sign}} \times 2^{(\mathbf{exponent} - 127)} \times (1 + \mathbf{mantissa})$$

Convert decimal → IEEE 754

-0.75

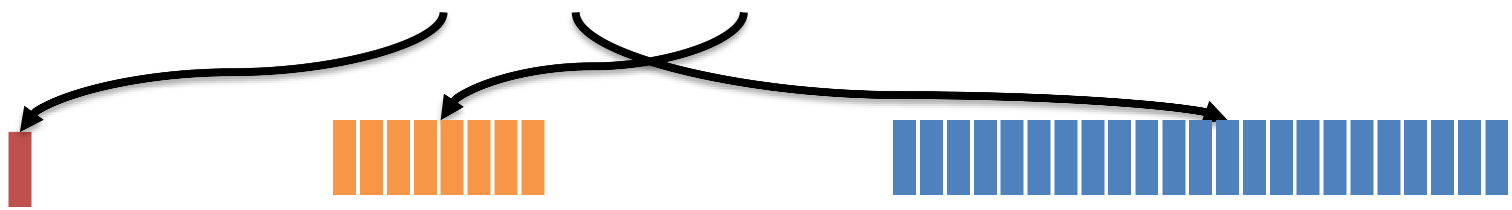
1 **8 bits of exponent** **23 bits of mantissa**



Sign of mantissa

decimal → binary (i.e. ½ + ¼) → normalised

$$-0.75 = -0.11 = -1.1 \times 2^{-1}$$



1

01111110

(-1 + 127 =) 126

1+ .1000000,00000000,00000000

10111111, 01000000, 00000000, 00000000

(Just for fun, let's look at that result from a different perspective...)

8 + 2 + 1 = 11 → <u>B</u>

8 + 4 + 2 + 1 = 15 → <u>F</u>

= <u>4</u>

= <u>0</u> = <u>0</u> = <u>0</u> = <u>0</u> = <u>0</u>

1011, 1111, 0100, 0000, 0000, 0000, 0000, 0000

10111111, 01000000, 00000000, 00000000

So, in hexadecimal...

Result is: BF40 0000$_{16}$

Convert IEEE 754 → decimal

01000010, 10101010, 00000000, 00000000

Assume 1.xxx

+ve

10000101

128 + 4 + 1 = 133

133 – 127 = 6

1 + .0101010, 00000000, 00000000

$1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64}$

= 1 + 0.25 + 0.0625 + 0.015625

= 1.328125

1 **8 bits of exponent** **23 bits of mantissa**

$= (-1)^{\text{sign}} \times 2^{(\text{exponent} - 127)} \times (1 + \text{mantissa})$

$+1.328125 \times 2^6$

$= +1.328125 \times 64$

$= \mathbf{85}_{\text{decimal}}$

“Special” values in IEEE 754

- In practice, some bit patterns never appear...
  - So we can use them to indicate *special cases*, such as:

Meaning	**Exponent**	**Mantissa**
Zero	0	0
Infinity (∞)	2^8-1 (all 8 bits set)	0
Not a Number (**NaN**)	2^8-1 (all 8 bits set)	Non zero

(“NaN” is used to indicate results for which there is no valid outcome, such as division by zero)

Equality testing with floating point numbers

- With numbers coded as floating point, we must <u>never</u> write code like this :

```
if (x == y) {…}
```

or, `if (0.1 + 0.2 == 0.3) {…}`

- *Why not*?
  - A possible **rounding error** resulting from lack of precision means that *we can never be confident that two numbers that "should be" equal are actually coded as equal…*
- Instead, we should take a cautious approach:

```
if (abs(x-y) < error_tolerance) {…}
```

Don’t believe me?

- 0.1 + 0.2 = 0.3 ??

```
$ more check.c

#include <stdio.h>

main ( ) {
	if ( ( 0.1 + 0.2 ) == 0.3)
			printf ("ok");
	else	printf ("oops!");
}

$ cc check.c -o check
$
$ ./check
oops!
```

Maybe Java is cleverer?

- 0.1 + 0.2 = 0.3 ??

```
C:\> more check.java
class check {
        public static void main ( String[ ] args ) {
                if ( ( 0.1 + 0.2 ) == 0.3)
                            System.out.println ("ok");
                else    System.out.println ("oops");
        }
}
C:\> javac check.java

C:\> java check
oops
```

Summary

- When switching consideration from decimal to non-decimal number systems, it's "just" a matter of working with a different radix
- Octal and hex are useful mental tools for people
- We have discovered a problem with addition/subtraction in excess *n*: can't treat subtraction as addition of a negative
  - Two's complement is the fix for this
- We understand the IEEE 754 standard for representing floating point numbers
  - and, incidentally, we have seen in this a direct application of the excess *n* and sign-and-magnitude coding techniques
- Care is needed in using floating point numbers