ECS404: Computer Systems and Networks 2016 Week 5

Signed Integers, Floating Point, Character Sets

This week

- Number representation
  - Brief recap on Unsigned
  - Signed
  - Floating Point
  - Rounding errors: Why you should be careful
- Character Sets
  - Common character sets and their structure.

Learning Objectives: Signed Integers

- Use of 2's complement to represent signed integers.
- Basic structure of 2's complement representation: which numbers can be represented, and the geometrical relationship between numbers and representations.
- Why we use it rather than sign and magnitude: how operations are implemented.
- How to encode and decode numbers into 2's complement: algorithms

Learning Objectives: Floating Point

- Floating point real numbers
- Scientific representation and its structure
- IEEE binary floating point numbers
- Rounding errors and problems with fixed width floating point systems.

Learning Objectives: Character Sets

- Text representation:
- ASCII, ISO-xxxx and Unicode-based character sets: numbers of bits used and basic facts about representations.

Number systems and recap

Three types of numbers

- Unsigned: we can think of these as positive integers (whole numbers)
- Signed: regular integers, can be positive or negative
- Floating point: real numbers, not necessarily whole numbers.

Three types of numbers

class	examples	representation	Java
unsigned	0,1,2,3..	unsigned binary	
signed	..-3,-2,-1,0,1,2,3..	2's complement	byte, short, int, long
floating point	-2.80,1.00,3.14	IEEE floating point	float, double

Java has 8 primitive datatypes

- byte: 8 bit signed 2’s complement
- short: 16 bit signed 2’s complement
- int: 32 bit signed 2’s complement
- long: 64 bit signed 2’s complement
- float: 32 bit IEEE floating point
- double: 64 bit IEEE floating point

C has a subtly different collection

- unsigned - unsigned integers
- int, long - signed integers
- float, double - floating point

All of these have fixed size (typically 32 or 64 bits), but can differ between implementations.

Examples: unsigned

- 32 bit: 4 8-byte blocks, numbers from 0 to 2^{32}-1
- Standard binary representation
- 00000000 00000000 00000000 00001001 = 9

Operations: unsigned

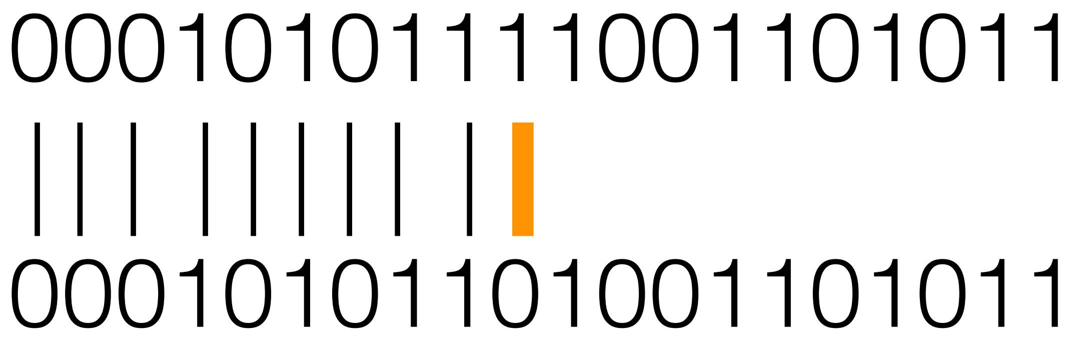
- Can use standard long addition to add (and subtraction to subtract, though we did not see that)
- Can use standard long multiplication to multiply… but binary version is easier because we only have to copy (shift) multiplicand.

Operations: unsigned

- Equality testing is easy: equal if and only if bit patterns the same
- Checking for < is easy: scan from left and first difference gives the order.

Operations: unsigned

- Checking for < is easy: scan from left and first difference gives the order.



```
00010101111001101011
||| ||||| ||
00010101101001101011
```

First difference shows top larger than bottom.

Signed Integers (2’s complement)

Negative numbers

- Of course we also need negative numbers.
- Let’s think about how we might do that.

Obvious answer

- Do what we do…
- Use one bit to give the sign, and the rest to give the magnitude.

Obvious answer

First bit is sign (1=-)

1000 0000 0000 0000 0000 0000 0001 0100

Remaining 31 bits are magnitude (20)

So this is -20

Problem 1

- We have two zeroes
- This makes some of our most common operations harder: testing for 0 and testing for equality

+0 = 0000 0000 0000 0000 0000 0000 0000 0000

-0 = 1000 0000 0000 0000 0000 0000 0000 0000

Problem 2

- The algorithm for adding two positive or two negative numbers is basically addition.
- The algorithm for adding a positive number to a negative one is basically subtraction.
- So our circuitry is more complicated, and probably slower.

Example

- Suppose we are using **8 bits**, and we use the first bit as a sign 0 is positive, and 1 is negative, say.
- 00001001 represents 9
- 10001001 represents -9
- 00000101 represents 5
- 10000101 represents -5

Example

- To compute 00001001 + 00000101 we add the last seven bits: 0001001 + 0000101
- Similarly to compute 10001001 + 10000101 we add the last seven bits.
- To compute 00001001 + 10000101 we subtract the last seven bits: 0001001 - 0000101 (and must check we get the sign right).
- To compute 10001001 + 00000101 we subtract the last seven bits in the other order: 0000101 - 0001001 (and again must check we get the sign right)… though we could use the above and just negate at the last minute.
- In any case, this is quite complicated.

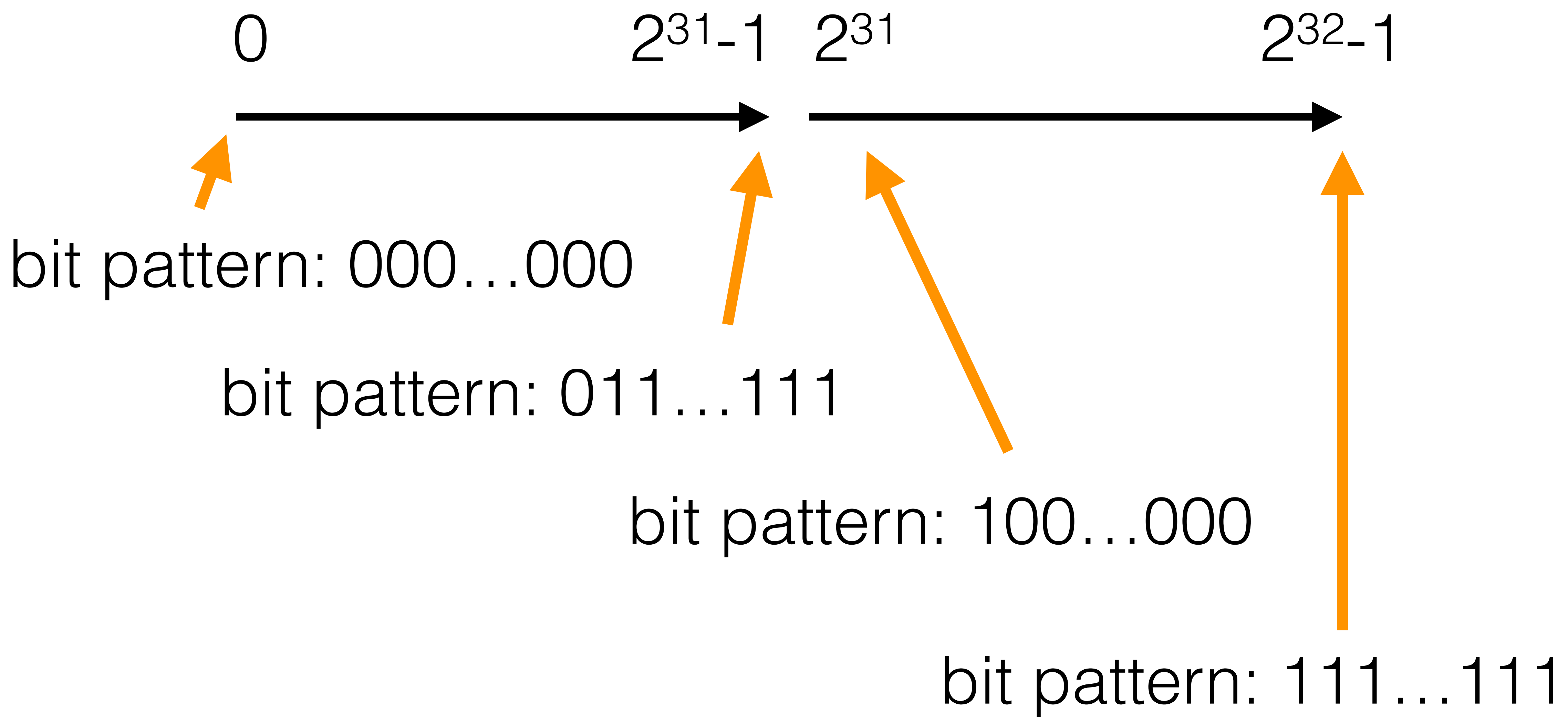
- there is a better way: 2’s complement

2's complement

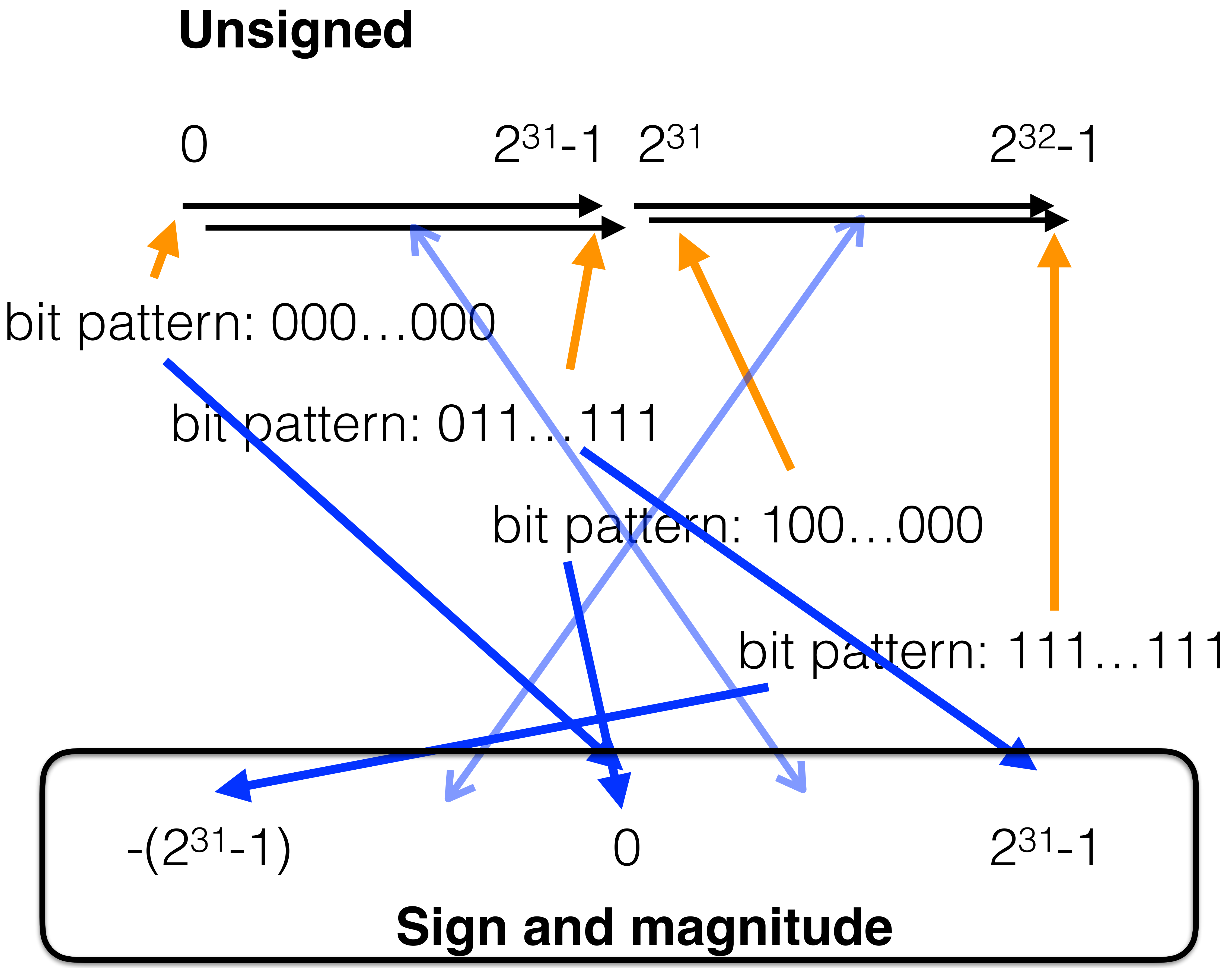
- 32 bit sign and magnitude uses unsigned $0 \ldots 2^{31}-1$ to represent signed $0 \ldots 2^{31}-1$, and the unsigned $2^{31} \ldots 2^{32}-1$ to represent signed $0 \ldots -(2^{31}-1)$ (decreasing)

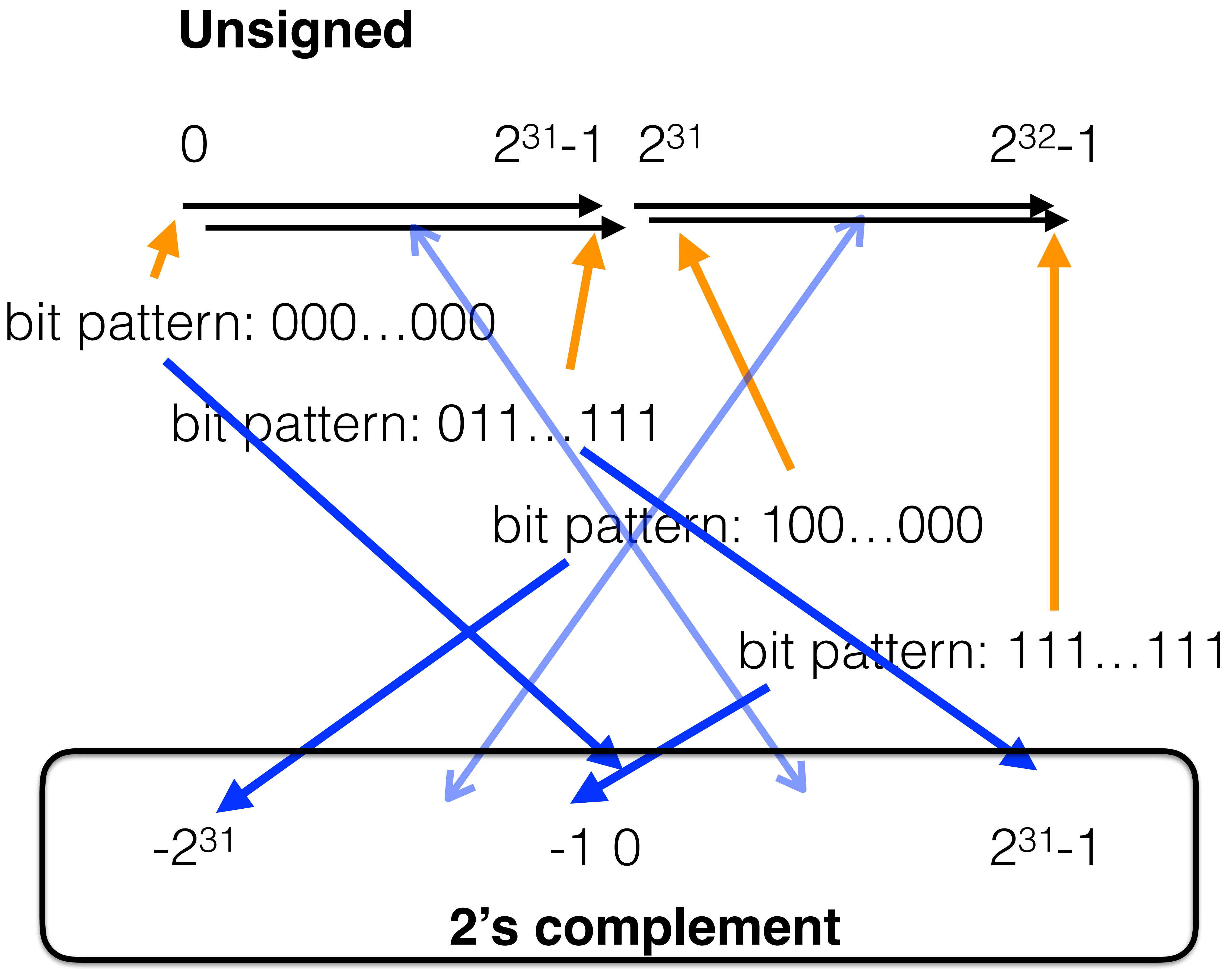
- 32 bit 2's complement uses unsigned $0 \ldots 2^{31}-1$ to represent signed $0 \ldots 2^{31}-1$, and the unsigned $2^{31} \ldots 2^{32}-1$ to represent signed $2^{31} .. -1$ (increasing)

Unsigned



0
$2^{31}-1$
2^{31}
$2^{32}-1$
bit pattern: 000…000
bit pattern: 011…111
bit pattern: 100…000
bit pattern: 111…111


Unsigned
0
2^{31}-1
2^{31}
2^{32}-1
bit pattern: 000…000
bit pattern: 011…111
bit pattern: 100…000
bit pattern: 111…111
-(2^{31}-1)
0
2^{31}-1
Sign and magnitude


Unsigned
0
2^{31}-1
2^{31}
2^{32}-1
bit pattern: 000…000
bit pattern: 011…111
bit pattern: 100…000
bit pattern: 111…111
-2^{31}
-1 0
2^{31}-1
2's complement

2’s complement

- We have not duplicated any representations.
- (So tests for 0 and equality will be simple).
- We have not reversed any line segments.

2’s complement

- Positive signed 0 to $2^{31}-1$ are represented by themselves.

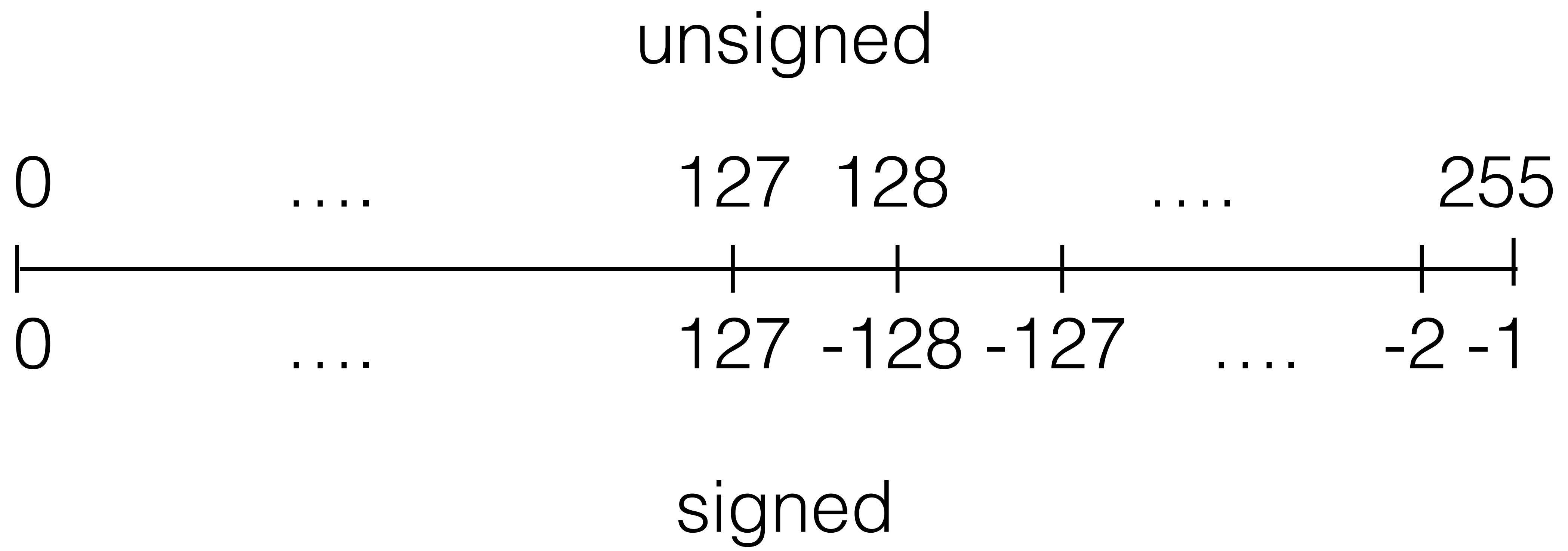
- Negative signed -2^{31} to -1 are represented by (themselves plus 2^{32}).

- So -3 is represented by $-3+2^{32} = 2^{32} -3$.

- Note that this is between 0 and 2^{32}.

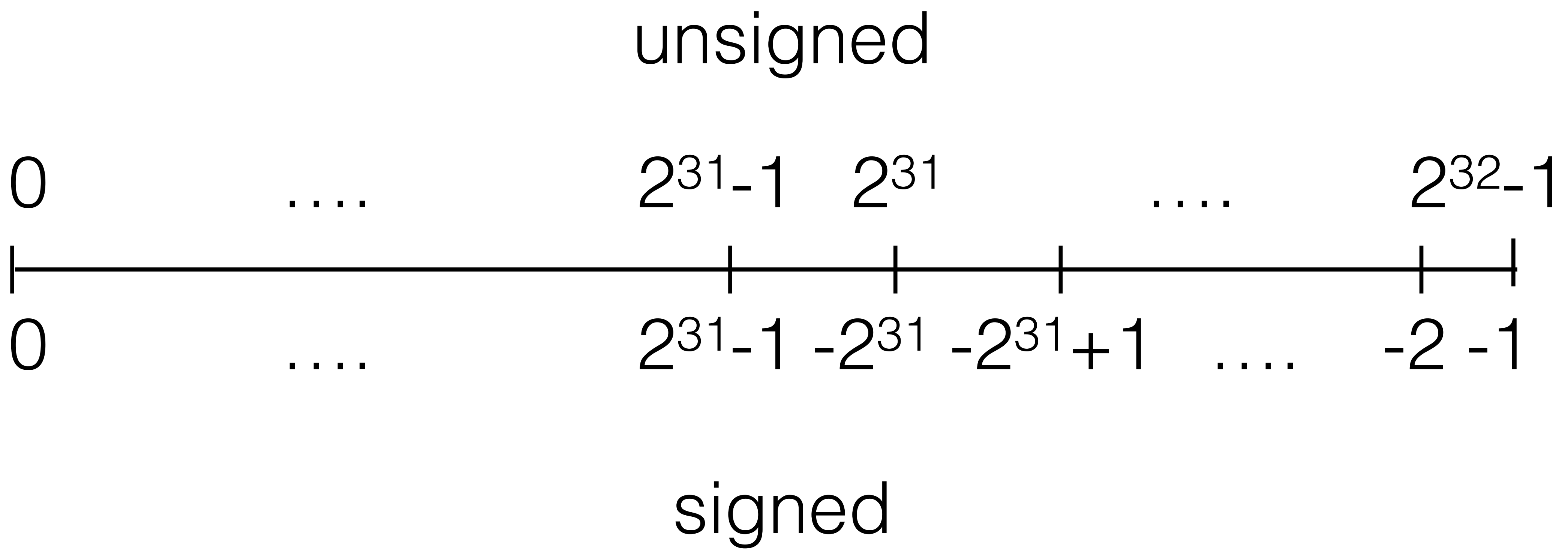
8 bit 2's complement

- For exercises we will use 8 bit 2's complement.
- 8 bit ($2^7 = 128$)
- 8 bit unsigned can represent numbers 0..255
- 8 bit 2's complement represents $-2^7=-128$ .. $127=2^7-1$



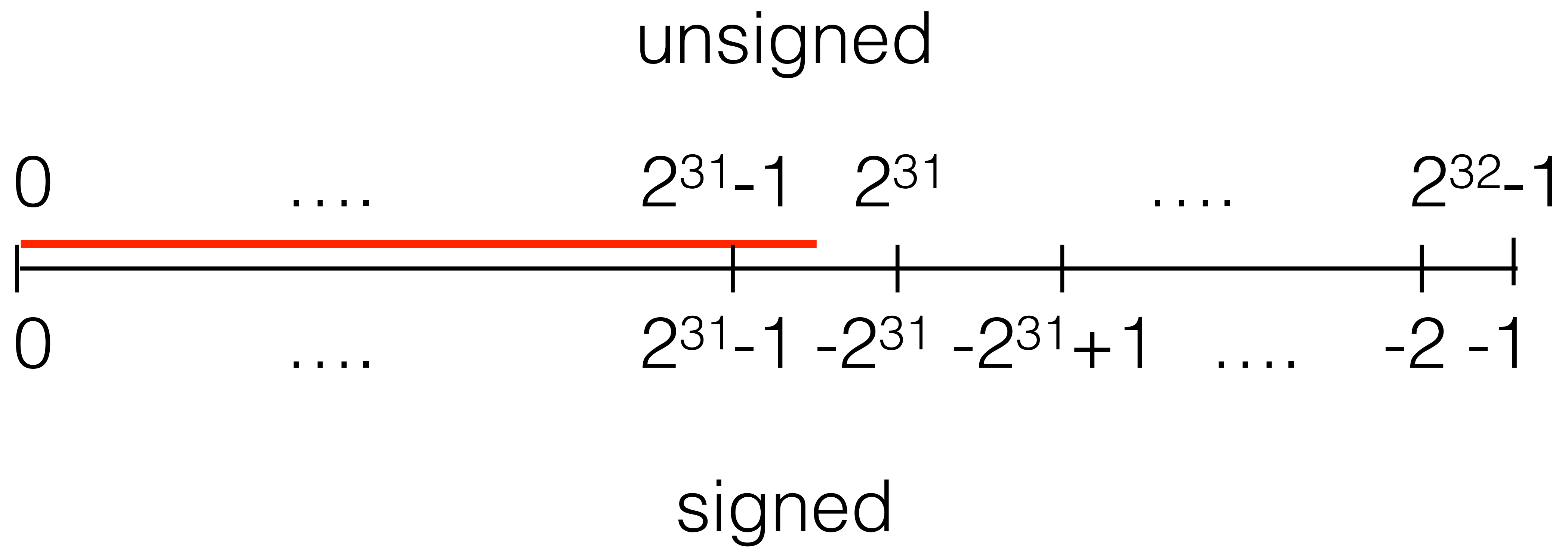
2's complement

- 32 bit
- 32 bit unsigned can represent numbers $0..2^{32}-1$
- 32 bit 2's complement represents $-2^{31} .. 2^{31}-1$



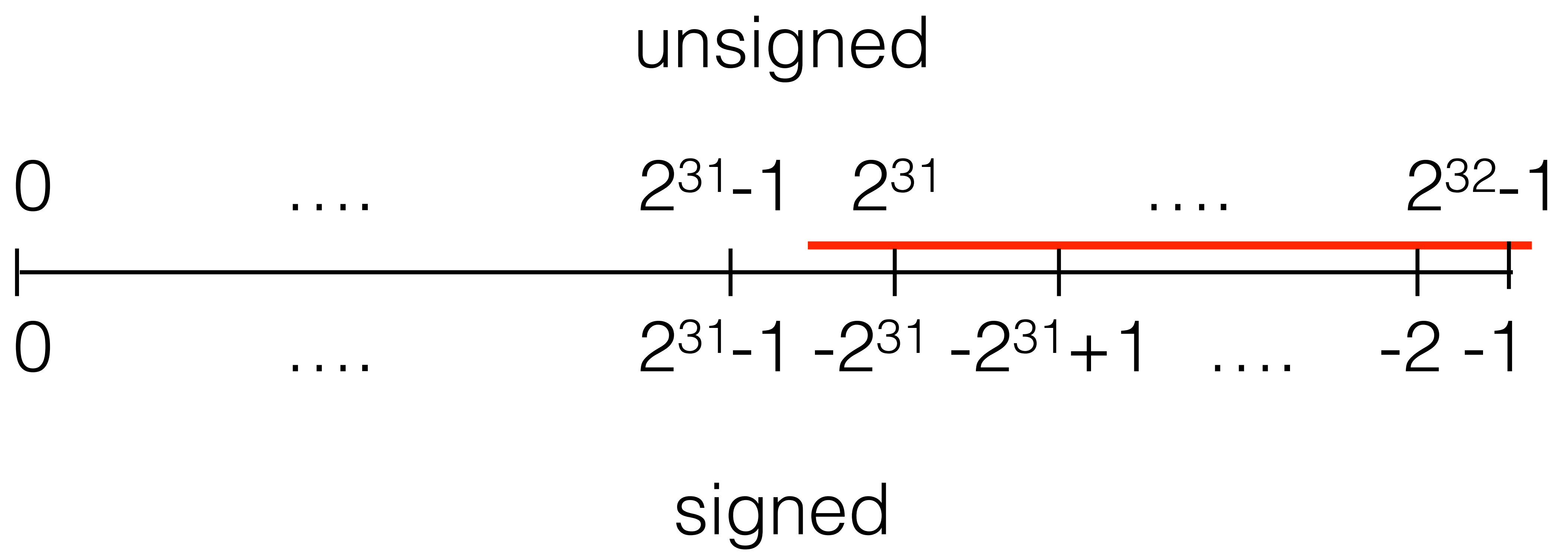
2's complement

- 32 bit
- On this bit signed = unsigned



2's complement

- 32 bit
- On this bit signed = unsigned - 2^{32}



2’s complement

- So it is always the case that if we take any 32-bit sequence, then the (unsigned value) - (signed value) is divisible by 2^{32}
- In other words the signed value = unsigned value mod 2^{32}

Operations mod n

Mathematical Property:

If a1 = a2 mod n and b1 = b2 mod n, then

- a1 + b1 = a2 + b2 mod n
- a1 - b1 = a2 - b2 mod n
- a1 * b1 = a2 * b2 mod n

Practical consequence

- With 2’s complement, we can use unsigned addition, subtraction, multiplication algorithms to implement signed addition, subtraction, multiplication.

Translating between unsigned and 2's complement

- There is more than one way.
- You ALWAYS have to distinguish between:
- signed - distinguish between positive and negative (>= 0 and <0)
- unsigned - distinguish between $< 2^{n-1}$ and $>= 2^{n-1}$

To get the 2’s complement bit pattern representing a signed integer

- We will use **8 bit** 2’s complement.
- Case 1: integer is positive
- Method: convert to unsigned binary. Pad with initial zeroes to correct length.
- Example: 20
- Convert to unsigned binary: 10100
- Pad with initial zeroes: 00010100

To get the 2’s complement bit pattern representing a signed integer

- We will use **8 bit** 2’s complement.
- Case 2: integer is negative
- Goal: if integer is x, then representation will be unsigned binary translation of 2^n+x
- Example: x=-20, then representation is 2^8+(-20) = 128 - 20

To get the 2's complement bit pattern representing a signed integer

- **8 bit** 2's complement, negative input: -20
- Representation of x is unsigned binary translation of 2^n+x
- Example: x=-20, then representation is $2^8+(-20)$ = 256 - 20
- There are many ways to calculate this.

To get the 2’s complement bit pattern representing a signed integer

- **8 bit** 2’s complement, negative input: -20
- Method 1: Do the subtraction 256-20 in decimal. Convert result to binary.
- 256 - 20 = 236
- $236_{10} = 11101100_2$
- Result: 1110 1100

To get the 2’s complement bit pattern representing a signed integer

- **8 bit** 2’s complement, negative input: -20
- Method 1: Do the subtraction 128-20 in decimal, then convert result to binary. Pad with zeroes.
- Problem: can leave you with a (very) large number to convert to binary.

To get the 2's complement bit pattern representing a signed integer

- **8 bit** 2's complement, negative input: -20
- Method 2: Convert 20 to binary. Do the subtraction 256-20 in binary.
- $20_{10} = 10100_2$
- Result: 1 0000 0000 - 1 0100

To get the 2's complement bit pattern representing a signed integer

- **8 bit** 2's complement, negative input: -20
- Method 2: Convert 20 to binary. Do the subtraction 256-20 in binary.

```
 1 0000 0000

      1 0100   -
 ___________

 ___________
```

To get the 2's complement bit pattern representing a signed integer

First two digits OK

```
1 0000 0000

     1 0100    -
    _______
         00
    _______
```

To get the 2's complement bit pattern representing a signed integer

Next needs borrow

```
  1 0000 0000
    1111 1100       Borrows
       1 0100   -
  ___________
         100
  ___________
```

To get the 2's complement bit pattern representing a signed integer

From then on, OK
Effectively flip the bit.

```
 1 0000 0000
   1111 1100      Borrows
      1 0100   -
   _________
   1110 1100
   _________
```

To get the 2's complement bit pattern representing a signed integer

```
  1 0000 0000
    1111 1100      Borrows
       1 0100   -
  ___________
    1110 1100
  ___________
```

Result is: 1110 1100

To get 2's complement representation of a signed integer

- If you look at the last algorithm, you see that you can write it as:
    - translate to binary
    - do something complicated with the bits (scan from right till first one, leave that unchanged, then flip all the bits to the left…
    - some of you may have been taught this

To get 2's complement representation of a signed integer

A better method:

To compute -n where n is in 2's complement binary:

1. **flip the bits of n (ie change 0's to 1's and vice versa)**

   Example: take 6 in 8-bit: 0000 0110 goes to 1111 1001

   (Note: if we add these two together we get 1111 1111, so in unsigned terms this is equivalent to computing 255 - n = 256 - 1 - n = 256 - n -1. In general if we are using k bits it is 2^k-n-1).

To get 2’s complement representation of a signed integer

To compute -n where n is in 2’s complement binary:

1. **flip the bits of n (ie change 0’s to 1’s and vice versa)**
2. **add 1**

Example: take 6 in 8-bit: 0000 0110 goes to

1. 1111 1001
2. 1111 1010

(Note: in stage 1 we computed 256 - n -1, so this second stage gives 256 -n. The last 8 bits of this are the 2’s complement representation of -n. In general if we are using k bits, we get 2^k-n).

To get 2’s complement representation of a signed integer

To compute -n where n is in 2’s complement binary:

1. **flip the bits of n (ie change 0’s to 1’s and vice versa)**

2. **add 1**

Note: both of these operations are basic binary operations that the cpu will have as standard instructions.

Translation to and from 2's complement

- There are other ways of doing this.
- You will NOT be penalised for using them PROVIDED you explain your method clearly.

Examples: signed

- 32 bit: 4 8-byte blocks, numbers from -2^{31} to $2^{31}-1$
- Standard 2's complement representation
- 00000000 00000000 00000000 00000101 = 5
- 11111111 11111111 11111111 11111011 = -5

Floating Point

Floating point

- Computers use the IEEE floating point standard to represent real numbers (real is in the mathematical sense of numbers that are not integers).
- This is based on standard scientific notation, but using a normalised form, and expressed in binary.
- Most common is 64 bit and we will use that in examples.

Scientific Notation

- Scientists write very large or very small numbers in **scientific notation.**
- Number is written as:
  - 2.9979×10^{8} (speed of light)
  - or 6.626×10^{-34} (Planck's constant)

Scientific Notation

Parts of the number:

- 2.9979×10^{8} (speed of light)

significand or mantissa

exponent

- or 6.626×10^{-34} (Planck's constant)

Scientific Notation

Parts of the number:

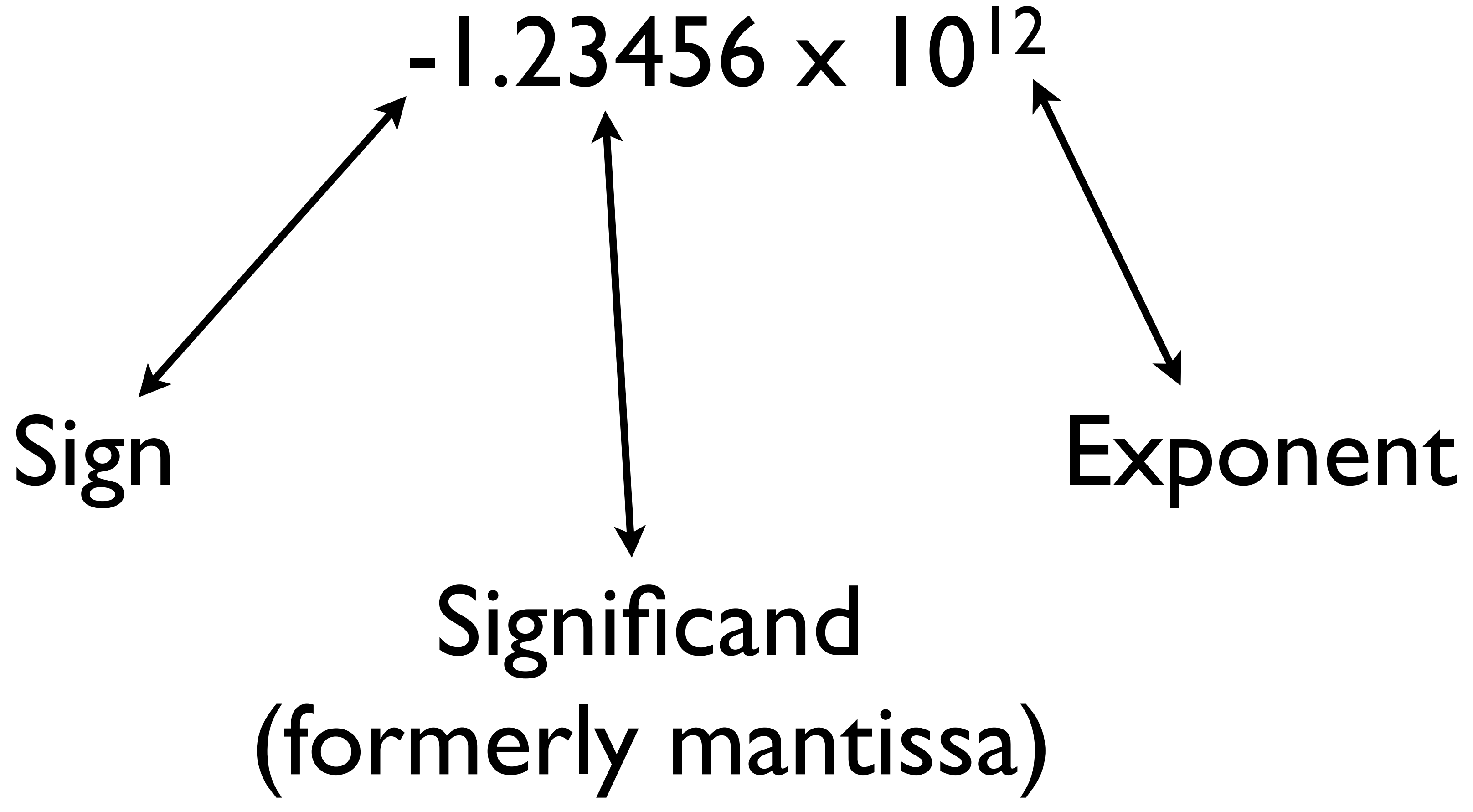
- 2.9979×10^{8} (speed of light)

significand is between 1 and 10

exponent is either positive or negative

- or 6.626×10^{-34} (Planck's constant)

Scientific notation



Multiplying two numbers in scientific notation

$(2.4 \times 10^{20}) * (1.5 \times 10^{-5})$

Multiply significands

Add exponents

$= 3.6 \times 10^{15}$

Adding two numbers in scientific notation

$(2.4 \times 10^{20}) + (1.2 \times 10^{19})$
$= 2.52 \times 10^{20}$

Translate to same exponent:
$1.2 \times 10^{19} = 0.12 \times 10^{20}$

Add significands:
$2.4 + 0.12 = 2.52$

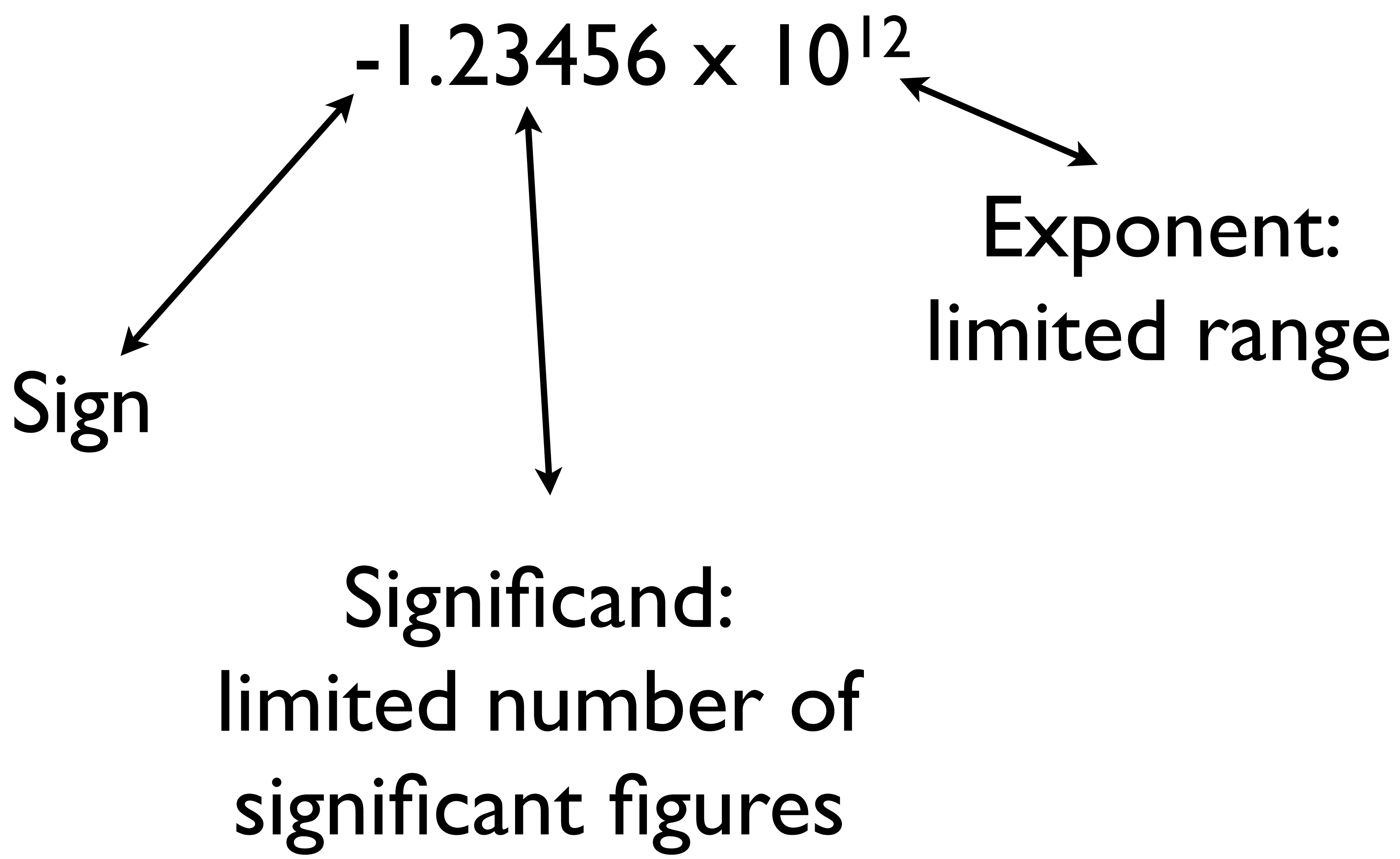
Computers

- Use a similar scheme to represent real numbers.
- It is called floating point.
- It uses a fixed size representation.
- But first we think about fixed size decimal scientific notation.

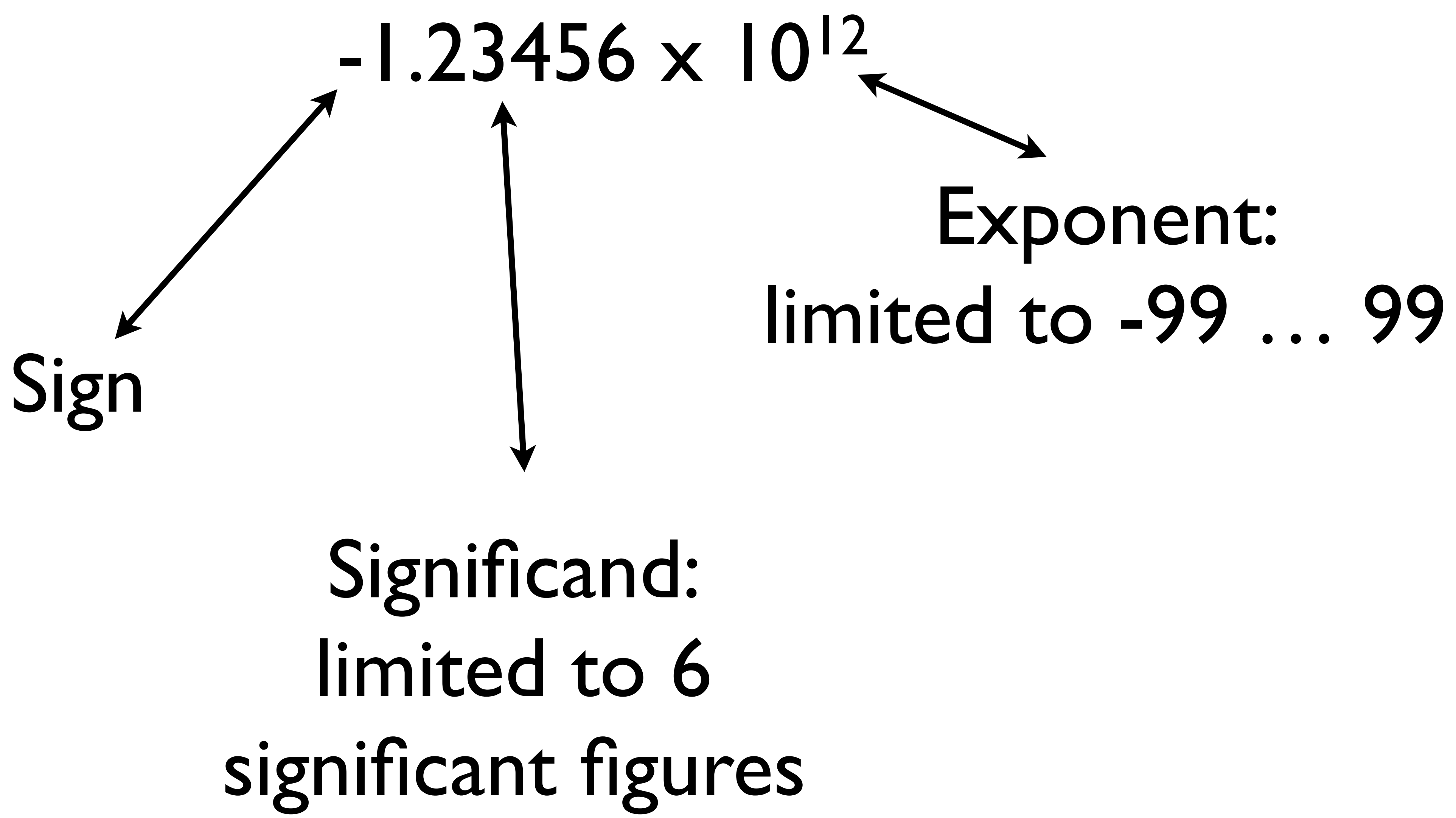
Fixed width decimal

- Use a similar scheme to represent real numbers.
- It is called floating point.
- It uses a fixed size representation (we refer to it as **fixed width**).
- But first we think about fixed width decimal scientific notation.

Fixed width decimal


-1.23456×10^{12}
Sign
Significand: limited number of significant figures
Exponent: limited range

Fixed width decimal: example


-1.23456×10^{12}
Sign
Significand: limited to 6 significant figures
Exponent: limited to -99 … 99

Problems with fixed width representations

1. You can't represent every number accurately.

- Some numbers are too big: 10^{102}
- Some numbers are just too small: 10^{-102}
- Some numbers have too many decimal places: 10/3 = 3.33333333….
- All of these numbers have to be approximated.
- This introduces **rounding errors.**

Rounding errors

- Given a number we **round** it to the nearest number we can represent.
    - 10^{-102} is too small to represent and would be rounded to 0.
    - 10/3 = 3.33333333…. would be rounded to 3.33333×10^0
    - 10^{102} would probably not be rounded (it is 100 times larger than the closest number we can represent). We would just say we had a number too big to represent.
    - The difference between the accurate number and the nearest representable number is the **rounding error**.

Problems with fixed width representations

2. You get rounding errors when you add or multiply two representable numbers together.

- $1x10^7 + 4x10^0 = 1.000004x10^7$ which gets rounded to $1.00000x10^7$

- $1.00001x10^0 * 1.1x10^0 = 1.100011x10^0$ which gets rounded to $1.10001x10^0$

Problems with fixed width representations

3. If you use two different mathematically equivalent methods to compute a value, you may get slightly different results.

- it is easy (and sobering) to run a test and find out for how many whole numbers N, N*(1/N) is *not* equal to 1.

Problems with fixed width floating-point

- Rounding: you can't represent numbers accurately
- Operations: even when two numbers are representable accurately, it is usually not the case that their sum/difference/product/quotient is. So operations of addition, subtraction, multiplication, division introduce further errors.
- Because of rounding, if you try to compute a number in two different ways, you will often get different results (technically this is statistically usually, but not always for simple examples).
- Therefore you should NEVER use equality testing on floating point reals.

Computers

- use standard forms of fixed with reals
- very similar to the fixed width decimal we just saw
- formalised by the IEEE.

IEEE binary floating point

- The way this should be implemented is set out in IEEE standards (IEEE=Institute of Electrical and Electronics Engineers).
- The relevant one is IEEE 754 as revised in 2008: IEEE 754-2008, on QMPlus.

Basic structure (IEEE p8)

— Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^e \times m$, where

- — s is 0 or 1.
- — e is any integer $emin \leq e \leq emax$..
- — m is a number represented by a digit string of the form $d_0 \bullet d_1 d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$ (therefore $0 \leq m < b$).

Table 3.2—Parameters defining basic format floating-point numbers

	Binary format (b=2)			**Decimal format** (b=10)	
parameter	**binary32**	**binary64**	**binary128**	**decimal64**	**decimal128**
p, digits	24	53	113	16	34
emax	+127	+1023	+16383	+384	+6144

emin shall be $1 - emax$ for all formats.

64-bit reals

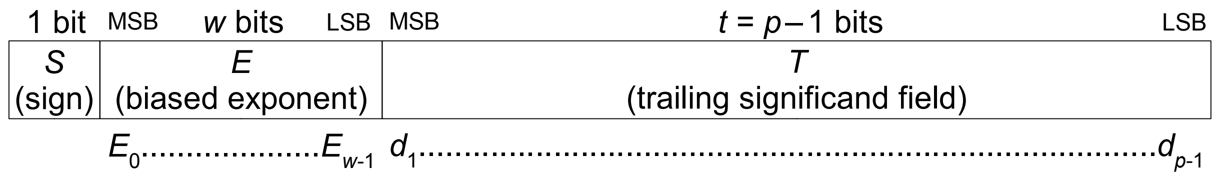


Figure 3.1—Binary interchange floating-point format

- 1 bit for the sign
- 11 bits for the exponent
- 53 bit precision
- This makes 65 bits.
- Lose one bit by using normalised form (if you know the leading bit is 1, you don’t need to record it).

64-bit reals: sign

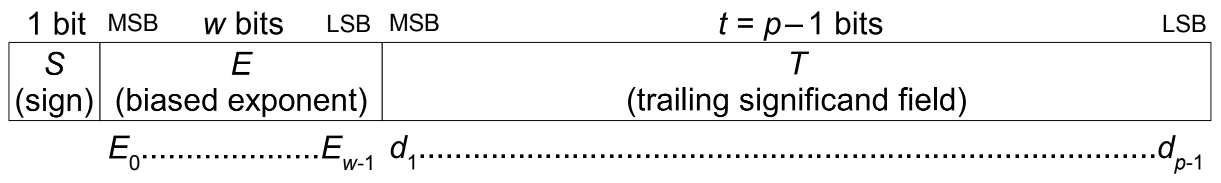


Figure 3.1—Binary interchange floating-point format

- 1 bit for the sign
- 0 is positive
- 1 is negative
- This is because $(-1)^0 = 1$, and $(-1)^1 = -1$

64-bit reals: exponent

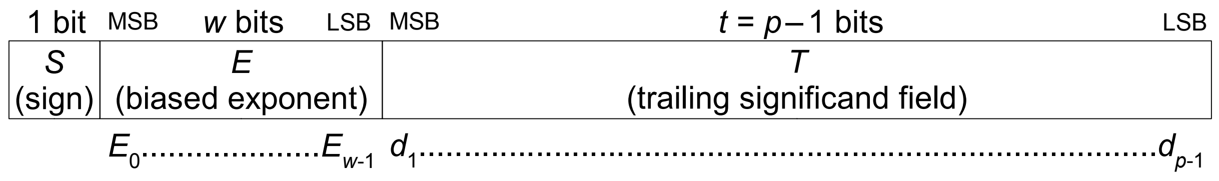


Figure 3.1—Binary interchange floating-point format

- 11 bits for the expon
- emax is 1023
- emin is 1 - emax = 1 - 1023 = -1022
- biased means “start at emin”
- 100 0000 0000 = 1024 represents 1
- 011 1111 1111 = 1023 represents 0
- 100 0000 0001 = 1025 represents 2

64-bit reals: exponent

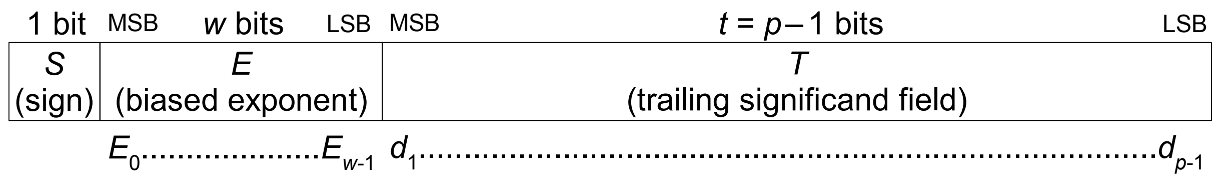

1 bit
MSB
w bits
LSB
MSB
$t = p-1$ bits
LSB
S
(sign)
E
(biased exponent)
T
(trailing significand field)
E_0...................E_{w-1}
d_1...d_{p-1}

Figure 3.1—Binary interchange floating-point format

- 11 bits for the exponent
- emax is 1023
- emin is 1 - emax = 1 - 1023 = -1022
- biased means "start at emin"
- 100 0000 0000 = 1024 represents 1
- 011 1111 1111 = 1023 represents 0
- 100 0000 0001 = 1025 represents 2
- So we get exponent by subtracting 1023 from the unsigned value.

64-bit reals: exponent

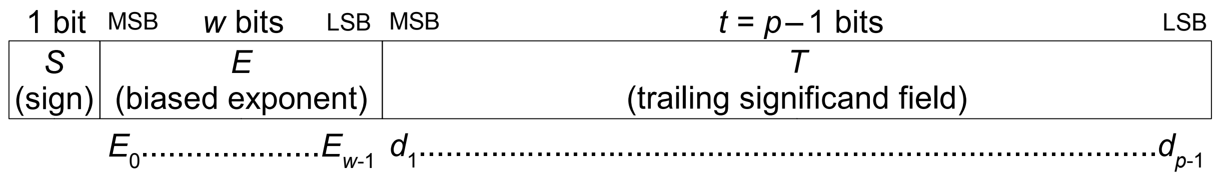

1 bit
MSB
w bits
LSB
MSB
t = p−1 bits
LSB
S
(sign)
E
(biased exponent)
T
(trailing significand field)
E0.....................Ew-1
d1.....................dp-1

Figure 3.1—Binary interchange floating-point format

- 11 bits for the exponent
- emax is 1023
- emin is 1 - emax = 1 - 1023 = -1022
- biased means "start at emin"
- We get exponent by subtracting 1023 from the unsigned value.
- This means 000 0000 0000 = 0 represents exponent 0 -1023 = -1023, which is out of range.
- And 111 1111 1111 = 2047 represents exponent 2047 -1023 = 1024, which is also out of range.
- That gives us some room to represent other things: e.g. 0, overflow, infinities, NaN's

64-bit reals: exponent

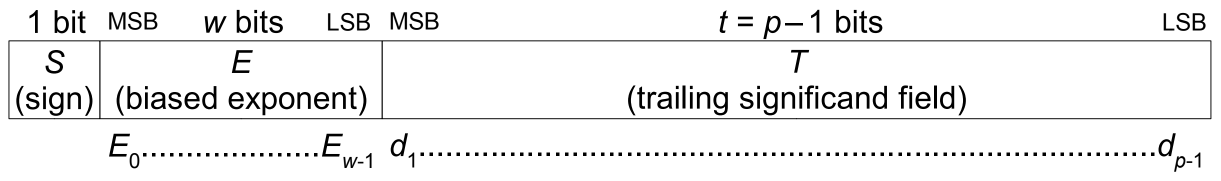

1 bit
MSB
w bits
LSB
MSB
t = p−1 bits
LSB
S
(sign)
E
(biased exponent)
T
(trailing significand field)
E0....................Ew-1
d1.....................dp-1

Figure 3.1—Binary interchange floating-point format

- 11 bits for the exponent
- emax is 1023
- emin is 1 - emax = 1 - 1023 = -1022
- biased means "start at emin"
- This means 000 0000 0000 = 0 represents exponent 0 -1023 = -1023, which is out of range.
- Note that in our experiment 0 was represented by 0000 0000 0000 0000 …. 0000
- This uses that out of range exponent.

Within each format, the following floating-point data shall be represented:

- Signed zero and non-zero floating-point numbers of the form $(-1)^s \times b^e \times m$, where
  - s is 0 or 1.
  - e is any integer $emin \leq e \leq emax$..
  - m is a number represented by a digit string of the form $d_0 \bullet d_1 d_2 \ldots d_{p-1}$ where d_i is an integer digit $0 \leq d_i < b$ (therefore $0 \leq m < b$).
- Two infinities, $+\infty$ and $-\infty$.
- Two NaNs, qNaN (quiet) and sNaN (signaling).

These are the only floating-point data represented.

64-bit reals: significand

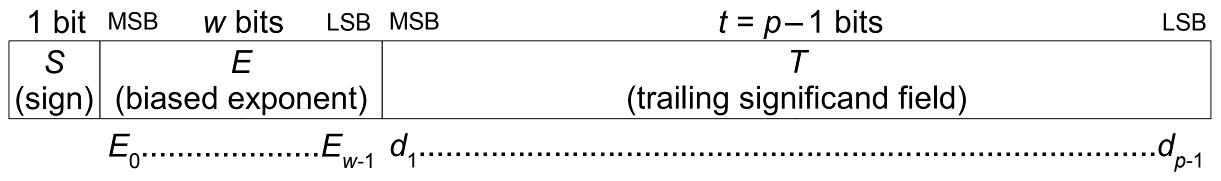


Figure 3.1—Binary interchange floating-point format

- 52 bits for the significand.
- Unless a binary number is 0, its most significant digit will be 1.
- This contrasts with decimal, where the most significant digit can be any of 1..9.
- This means that a normalised binary real in scientific notation will always look like:
- $+/- 1.d_1d_2d_3\ldots * 2^e$
- In IEEE floating point, the significand bits give us $d_1d_2d_3\ldots$

64-bit reals: significand

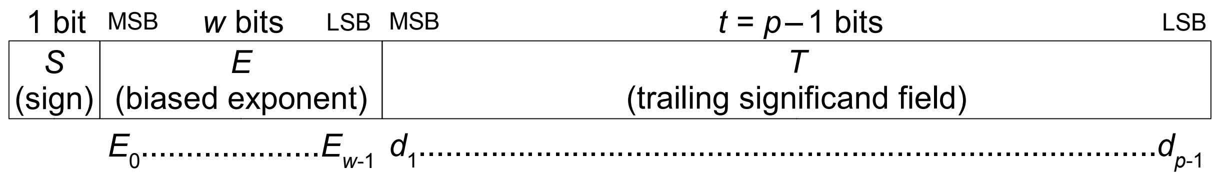


Figure 3.1—Binary interchange floating-point format

- 52 bits for the significand.
- This means that a normalised binary real in scientific notation will always look like:
- +/- $1.d_1d_2d_3\ldots * 2^e$
- In IEEE floating point, the significand bits give us $d_1d_2d_3\ldots$
- So 0000 0000 … 0000 means we have 1.000… as our significand (hence a power of 2: 1/4,1/2,1,2,4,…)
- Similarly 0100 0000 … 0000 means we have 1.01 times a power of 2, and 1.01 is 5/4 = 1.25 ($101_2 = 5_{10}$, and shift two places corresponds to division by $2^2 = 4$).

Big Endian and Little Endian

- Remember: data is organised into 8-bit bytes.
- There is a choice about which order to store the bytes in.
- We are used to seeing high-order bytes first. This is Big Endian.
- Intel and others use Little Endian (low-order bytes first).
- This is used for both integers and reals.
- It can be a little confusing when looking at bit dumps.

Floating Point

- One theme of this course is that many of the techniques used in computing correspond very closely to things we use or do already in real life.
- The computer representation of reals corresponds to scientific notation.
- More specifically it corresponds to scientific notation to a fixed number of places, and with a fixed range of possible powers of 10.

Floating Point: additional material

Compare what we have just seen with

- Speed of light: 2.997930 * 10^{8} ms^{-1}
- Mass of hydrogen atom: 1.67 x 10^{-24} g
- Avogadro's constant: $6.02214179 \times 10^{23}$

Rounding 1

- If we take an arbitrary real number, then it probably isn’t one of our floating-point reals.
- Example: the closest we get to 1/3 is 3.33 e -1.
- This is out by 0.00033333....
- So we get errors when we put numbers into the system.
- These errors can build up as the program runs.

Rounding 1

- and we certainly can’t represent numbers like pi, e, or the square root of 2 exactly.

Rounding 1

- Moreover, how accurately we can represent a number depends on how big it is.

- Near 0 we can get within 10^{-9} of a number. Smallest non-zero number we can represent is $1.00*10^{9}$ if we only allow significands beginning with 1-9. It's $0.01*10^{-9} = 1.00*10^{-11}$ if we allow ones beginning with 0.

- That's 0.000000001 in the first case and 0.00000000001 in the second.

Rounding 1

- Up near 1,000,000 we have:
- 1,000,000 = 1.00*10^6
- so the next number we can represent is 1.01*10^6
- 1.01*10^6 = 1,010,000
- So the gap between numbers we can represent has jumped to 10,000.
- In other words, we cannot represent 1,005,000 accurately.

Class question

- What is the smallest whole number we can’t represent?
- Ans: 1001
- It would be $1.001*10^3$ which requires 4 digits in the exponent.

Rounding 2

- We basically use two algorithms, one for addition and one for subtraction.

- Let's concentrate on addition.

- ASSUME: significand is between 1.000… and 9.999…

- IE it has a single digit to the left of the decimal point, and that digit is not 0.

Addition

- To add $x*10^a$ and $y*10^{b:}$
- Take the greater of a and b (without loss of generality, suppose it's a).
- Shift y right by a-b, to get y' ($y*10^b = y'*10^a$)
- Add x and y' essentially as integers to get z
- Round z to the correct number of places to get z'
- If the result is less than 10, answer is $z'*10^a$
- If the result is bigger than 10, shift right by 1 to get z", answer is z" $* 10^{a+1}$.

Addition (Example)

- To add $x*10^a$ = $2.01*10^3$ and $y*10^b$ = $3.24*10^{2:}$
- Take the greater of a and b (a=3).
- Shift y right by a-b=1, to get y'=0.324 ($y*10^b$ = $y'*10^a$)
- Add x and y' essentially as integers (2010 and 0324) to get z=2.334
- Round z to the correct number of places to get z'=2.33
- If the result is less than 10, answer is $z'*10^a$ = $2.33*10^3$

Addition (Example)

- To add $x*10^a = 9.80 * 10^4$ and $y*10^b = 4.67 * 10^3$
- Take the greater of a and b (a=4).
- Shift y right by a-b=1, to get y'=0.467 ($y*10^b = y'*10^a$)
- Add x and y' essentially as integers to get z = 10.267
- Round z to the correct number of places to get z' = 10.3
- If the result is less than 10, answer is $z'*10^a$
- If the result is bigger than 10, shift right by 1 to get z"= 1.03, answer is $z'' * 10^{a+1} = 1.03 * 10^5$.

Rounding 2

- When we add, multiply or subtract two floating-point numbers we don't necessarily get another one.
- Example: 1.23e1 + 4.56e-1

  = 1.23e1 + 0.0456 e1

  = 1.2756e1

  - Now we have to decide which number to use as the value of the sum.: it could be either 1.27e1 or 1.28e1
  - We could decide to: round to +∞, round to -∞, round to nearest, round toward zero, round away from 0.
  - In fact, IEEE does not allow round away from 0, and there are two variants of round to nearest: round ties to even and round ties away from 0.
  - These errors can build up too.

Rounding 2

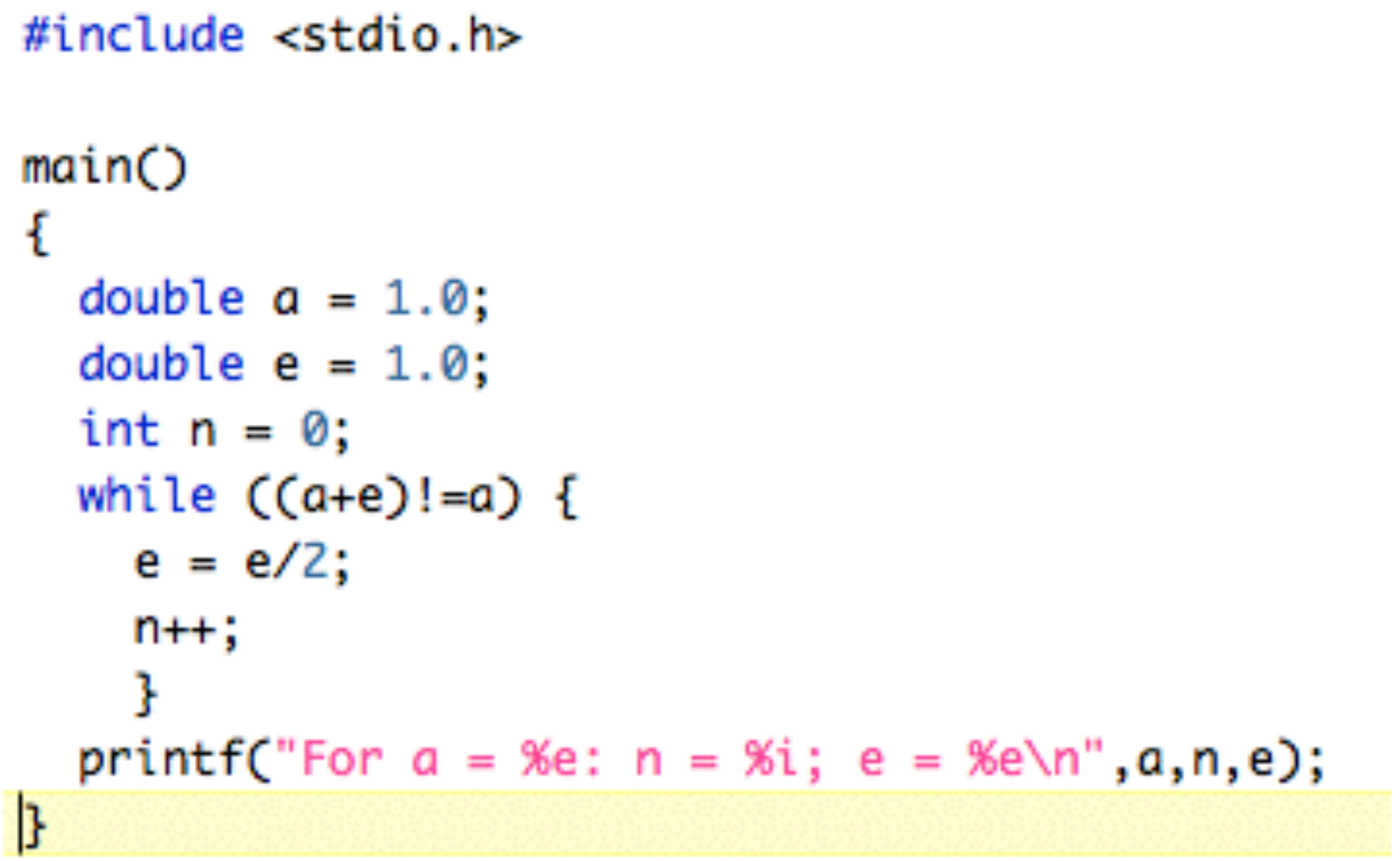
- Just to ram this point home: take a couple of extreme examples.
- 1.00 + 0.001 = 1.0 e0 + 1.0e-3 = 1.0 e0 !!!
- 1000 + 1 = 1.0e3 + 1.0e0 = 1.0 e3 = 1000 !!!
- This kind of thing really happens on your computer (not with these numbers), and we will be doing some experiments to see more detail.
- Notice that we are getting this problem because the numbers are apart by a factor of 10^3, which corresponds to the number of significant figures we have in the representation.

Experiment week4.6.c

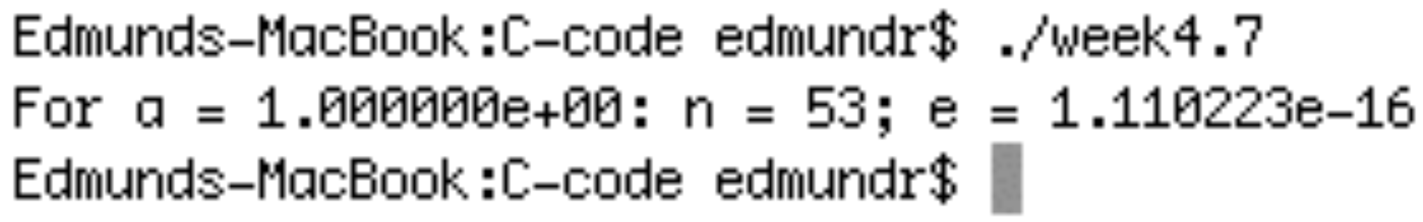
- Live experiment

Another test: week4.7.c

- Actually, we don’t need the constants to show this. We can find values.



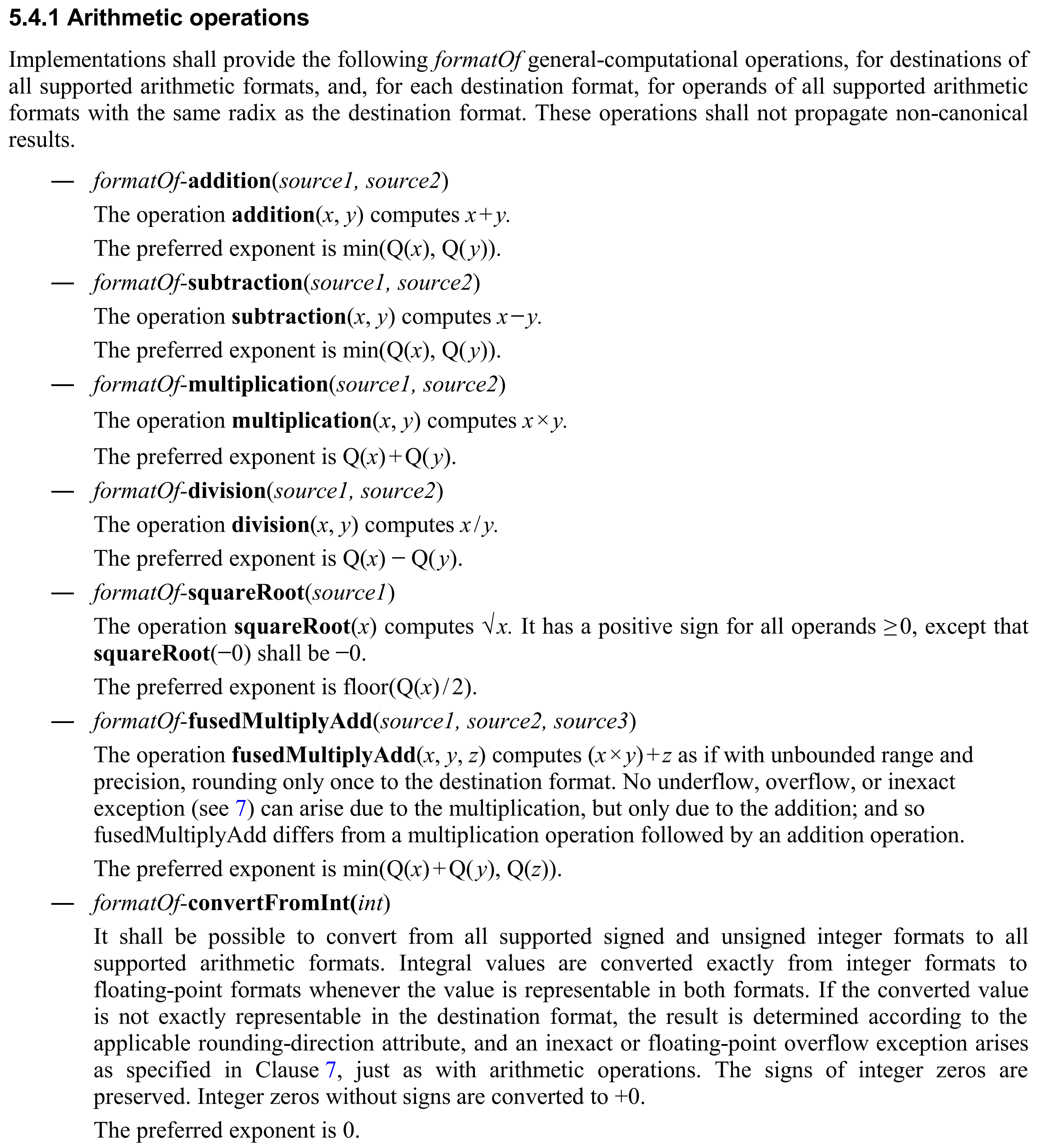
```c
#include <stdio.h>

main()
{
  double a = 1.0;
  double e = 1.0;
  int n = 0;
  while ((a+e)!=a) {
    e = e/2;
    n++;
    }
  printf("For a = %e: n = %i; e = %e\n",a,n,e);
}
```



```
Edmunds-MacBook:C-code edmundr$ ./week4.7
For a = 1.000000e+00: n = 53; e = 1.110223e-16
Edmunds-MacBook:C-code edmundr$
```

Operations

- The standard also specifies arithmetical operations...



5.4.1 Arithmetic operations

Implementations shall provide the following *formatOf* general-computational operations, for destinations of all supported arithmetic formats, and, for each destination format, for operands of all supported arithmetic formats with the same radix as the destination format. These operations shall not propagate non-canonical results.

— *formatOf*-**addition**(*source1, source2*)

The operation **addition**(x, y) computes $x+y$.

The preferred exponent is min(Q(x), Q(y)).

— *formatOf*-**subtraction**(*source1, source2*)

The operation **subtraction**(x, y) computes $x-y$.

The preferred exponent is min(Q(x), Q(y)).

— *formatOf*-**multiplication**(*source1, source2*)

The operation **multiplication**(x, y) computes $x \times y$.

The preferred exponent is Q(x)+Q(y).

— *formatOf*-**division**(*source1, source2*)

The operation **division**(x, y) computes x / y.

The preferred exponent is Q(x) − Q(y).

— *formatOf*-**squareRoot**(*source1*)

The operation **squareRoot**(x) computes $\sqrt{x}$. It has a positive sign for all operands ≥ 0, except that **squareRoot**(−0) shall be −0.

The preferred exponent is floor(Q(x)/2).

— *formatOf*-**fusedMultiplyAdd**(*source1, source2, source3*)

The operation **fusedMultiplyAdd**(x, y, z) computes $(x \times y)+z$ as if with unbounded range and precision, rounding only once to the destination format. No underflow, overflow, or inexact exception (see 7) can arise due to the multiplication, but only due to the addition; and so fusedMultiplyAdd differs from a multiplication operation followed by an addition operation.

The preferred exponent is min(Q(x)+Q(y), Q(z)).

— *formatOf*-**convertFromInt**(*int*)

It shall be possible to convert from all supported signed and unsigned integer formats to all supported arithmetic formats. Integral values are converted exactly from integer formats to floating-point formats whenever the value is representable in both formats. If the converted value is not exactly representable in the destination format, the result is determined according to the applicable rounding-direction attribute, and an inexact or floating-point overflow exception arises as specified in Clause 7, just as with arithmetic operations. The signs of integer zeros are preserved. Integer zeros without signs are converted to +0.

The preferred exponent is 0.

More on rounding

- The inexactness in floating-point means that you need to be careful how you do things.
- There are some basic rules:
  - never base a test on whether two floating-point numbers are equal (see experiment)
  - never simply use raw values (almost always better to work with a base and offset, x+e).
- For more (much more) see "What every computer scientist should know about floating-point arithmetic"

Reading

- What every computer scientist should know about floating-point arithmetic.
- IEEE 754-2008

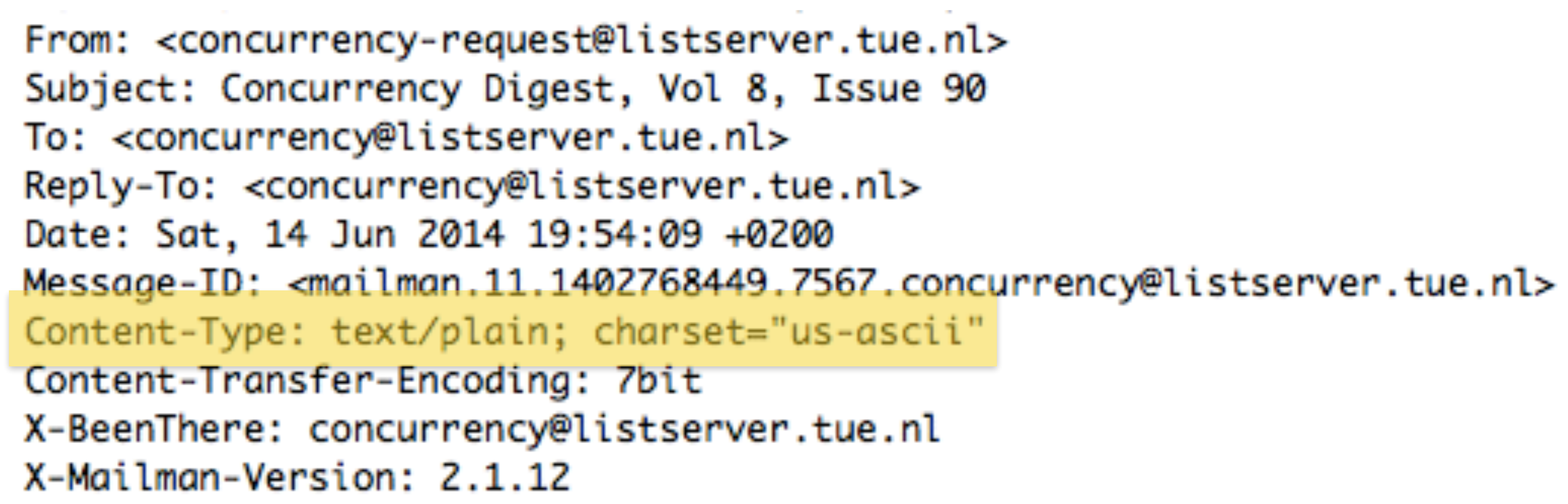
Floating Point: end of additional material

Character Sets

Text: the simplest example

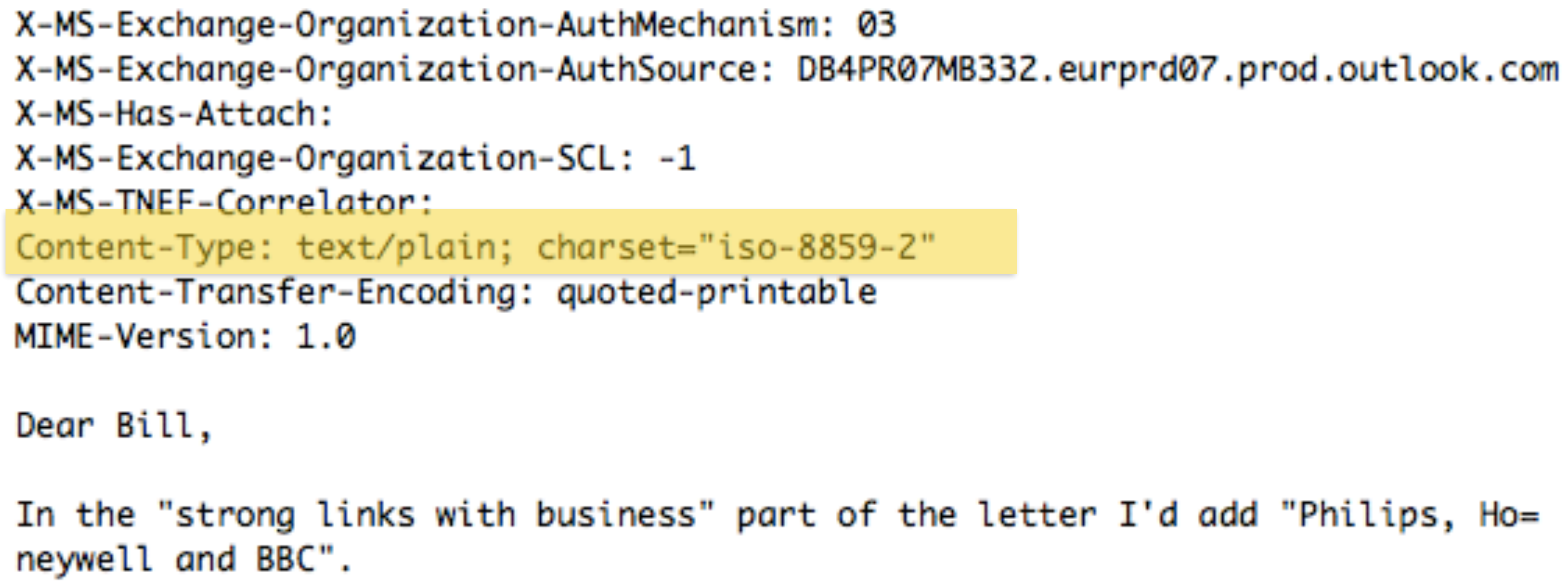
- Computers need to deal with text.
- Text is made up of individual characters.
- Each character is represented as a number.
- Exactly how it is represented as a number depends on the encoding and the “character set” being used.

From a mail message header



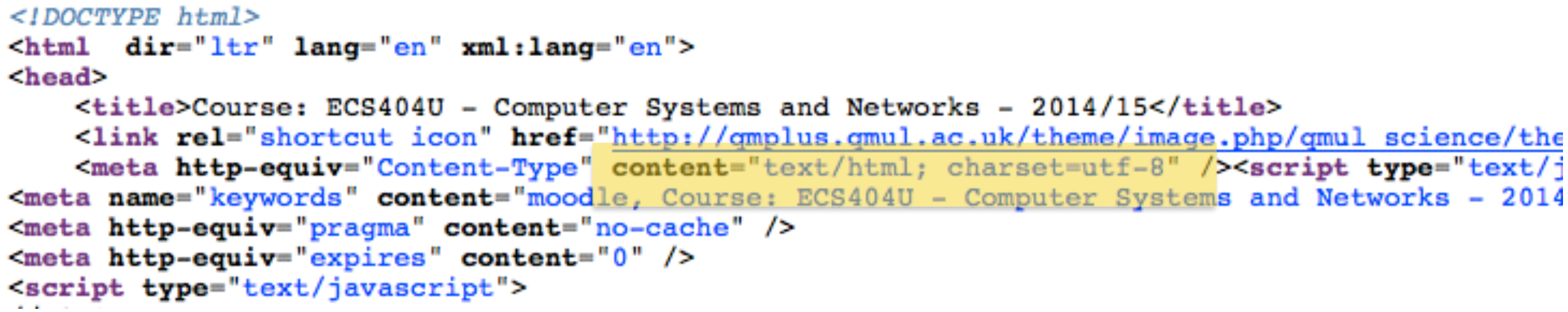
```
From: <concurrency-request@listserver.tue.nl>
Subject: Concurrency Digest, Vol 8, Issue 90
To: <concurrency@listserver.tue.nl>
Reply-To: <concurrency@listserver.tue.nl>
Date: Sat, 14 Jun 2014 19:54:09 +0200
Message-ID: <mailman.11.1402768449.7567.concurrency@listserver.tue.nl>
Content-Type: text/plain; charset="us-ascii"
Content-Transfer-Encoding: 7bit
X-BeenThere: concurrency@listserver.tue.nl
X-Mailman-Version: 2.1.12
```

Another



```
X-MS-Exchange-Organization-AuthMechanism: 03
X-MS-Exchange-Organization-AuthSource: DB4PR07MB332.eurprd07.prod.outlook.com
X-MS-Has-Attach:
X-MS-Exchange-Organization-SCL: -1
X-MS-TNEF-Correlator:
Content-Type: text/plain; charset="iso-8859-2"
Content-Transfer-Encoding: quoted-printable
MIME-Version: 1.0

Dear Bill,

In the "strong links with business" part of the letter I'd add "Philips, Ho=
neywell and BBC".
```

QMPlus Web Page



```
<!DOCTYPE html>
<html  dir="ltr" lang="en" xml:lang="en">
<head>
    <title>Course: ECS404U - Computer Systems and Networks - 2014/15</title>
    <link rel="shortcut icon" href="http://qmplus.qmul.ac.uk/theme/image.php/qmul_science/the
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" /><script type="text/j
<meta name="keywords" content="moodle, Course: ECS404U - Computer Systems and Networks - 2014
<meta http-equiv="pragma" content="no-cache" />
<meta http-equiv="expires" content="0" />
<script type="text/javascript">
```

Characters: ASCII

- The oldest and still the most famous representation is ASCII (American Standard Code for Information Interchange).
- It uses 7 bits, and so numbers 0…2^7-1 =127
- It is based on the idea of an old-style line-printer.

Regular ASCII Chart (character codes 0 – 127)

000*d*	00*h*	⸜	(nul)	016*d*	10*h*	►	(dle)	032*d*	20*h*	␣	048*d*	30*h*	0	064*d*	40*h*	@	080*d*	50*h*	P	096*d*	60*h*	‘	112*d*	70*h*	p
001*d*	01*h*	☺	(soh)	017*d*	11*h*	◄	(dc1)	033*d*	21*h*	!	049*d*	31*h*	1	065*d*	41*h*	A	081*d*	51*h*	Q	097*d*	61*h*	a	113*d*	71*h*	q
002*d*	02*h*	●	(stx)	018*d*	12*h*	↕	(dc2)	034*d*	22*h*	"	050*d*	32*h*	2	066*d*	42*h*	B	082*d*	52*h*	R	098*d*	62*h*	b	114*d*	72*h*	r
003*d*	03*h*	♥	(etx)	019*d*	13*h*	‼	(dc3)	035*d*	23*h*	#	051*d*	33*h*	3	067*d*	43*h*	C	083*d*	53*h*	S	099*d*	63*h*	c	115*d*	73*h*	s
004*d*	04*h*	♦	(eot)	020*d*	14*h*	¶	(dc4)	036*d*	24*h*	$	052*d*	34*h*	4	068*d*	44*h*	D	084*d*	54*h*	T	100*d*	64*h*	d	116*d*	74*h*	t
005*d*	05*h*	♣	(enq)	021*d*	15*h*	§	(nak)	037*d*	25*h*	%	053*d*	35*h*	5	069*d*	45*h*	E	085*d*	55*h*	U	101*d*	65*h*	e	117*d*	75*h*	u
006*d*	06*h*	♠	(ack)	022*d*	16*h*	▬	(syn)	038*d*	26*h*	&	054*d*	36*h*	6	070*d*	46*h*	F	086*d*	56*h*	V	102*d*	66*h*	f	118*d*	76*h*	v
007*d*	07*h*	•	(bel)	023*d*	17*h*	↨	(etb)	039*d*	27*h*	'	055*d*	37*h*	7	071*d*	47*h*	G	087*d*	57*h*	W	103*d*	67*h*	g	119*d*	77*h*	w
008*d*	08*h*	◘	(bs)	024*d*	18*h*	↑	(can)	040*d*	28*h*	(	056*d*	38*h*	8	072*d*	48*h*	H	088*d*	58*h*	X	104*d*	68*h*	h	120*d*	78*h*	x
009*d*	09*h*		(tab)	025*d*	19*h*	↓	(em)	041*d*	29*h*	)	057*d*	39*h*	9	073*d*	49*h*	I	089*d*	59*h*	Y	105*d*	69*h*	i	121*d*	79*h*	y
010*d*	0A*h*	◙	(lf)	026*d*	1A*h*		(eof)	042*d*	2A*h*	*	058*d*	3A*h*	:	074*d*	4A*h*	J	090*d*	5A*h*	Z	106*d*	6A*h*	j	122*d*	7A*h*	z
011*d*	0B*h*	♂	(vt)	027*d*	1B*h*	←	(esc)	043*d*	2B*h*	+	059*d*	3B*h*	;	075*d*	4B*h*	K	091*d*	5B*h*	[	107*d*	6B*h*	k	123*d*	7B*h*	{
012*d*	0C*h*		(np)	028*d*	1C*h*	∟	(fs)	044*d*	2C*h*	,	060*d*	3C*h*	<	076*d*	4C*h*	L	092*d*	5C*h*	\	108*d*	6C*h*	l	124*d*	7C*h*	|
013*d*	0D*h*	♪	(cr)	029*d*	1D*h*	↔	(gs)	045*d*	2D*h*	-	061*d*	3D*h*	=	077*d*	4D*h*	M	093*d*	5D*h*	]	109*d*	6D*h*	m	125*d*	7D*h*	}
014*d*	0E*h*	♫	(so)	030*d*	1E*h*	▲	(rs)	046*d*	2E*h*	.	062*d*	3E*h*	>	078*d*	4E*h*	N	094*d*	5E*h*	^	110*d*	6E*h*	n	126*d*	7E*h*	~
015*d*	0F*h*	☼	(si)	031*d*	1F*h*	▼	(us)	047*d*	2F*h*	/	063*d*	3F*h*	?	079*d*	4F*h*	O	095*d*	5F*h*	_	111*d*	6F*h*	o	127*d*	7F*h*	⌂

Some things to notice

- 7 bits: abcdefg
- A-Z occupy: 65-90: 1000001-1011010
- a-z occupy: 97-122: 1100001-1111010
- they differ by exactly 32 (hence in one bit), and capitals start at binary 100000.
- 0-9 occupies: 48-57 (binary 0110000 - 0111001)

Some things to notice

- There's a whole bunch of strange stuff, eg carriage return (moves the print head back to the start of the line) and line-feed (moves the paper up a line).
- There are no pound signs (only dollar=36), and no funny accents.

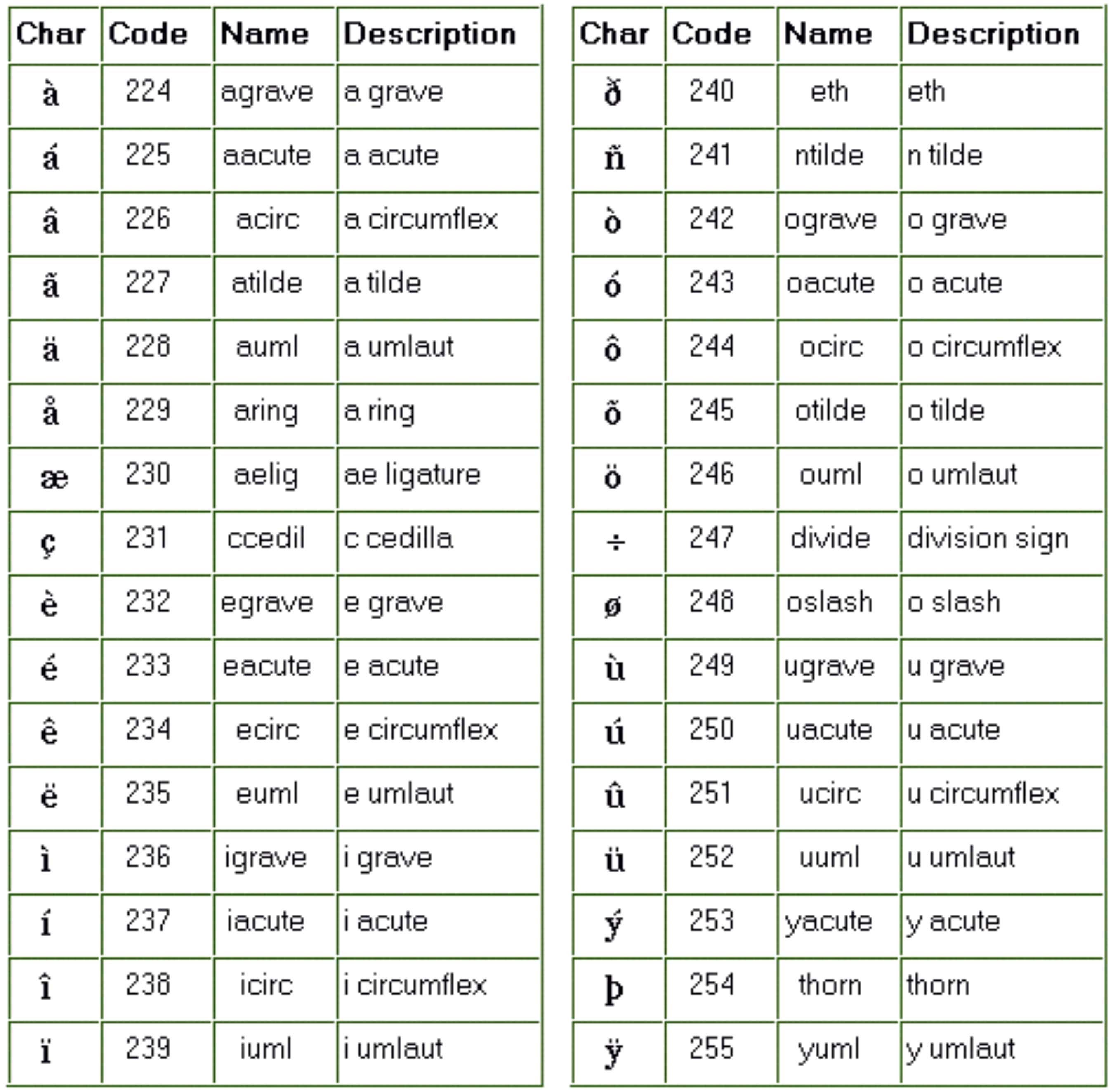
More modern character sets

- Computers work in units of 2^n bits. So more modern character sets would have 8 bits not 7, and 256 characters (0..255) not 128 (0..127).
- Most modern character sets are standardised by ISO (International Standards Organisation).

ISO-8859-1

- Extension of ASCII (to 8 bits)
- (Was) default character set in many browsers (now UTF-8)
- One of a number of ISO character sets designed to allow 8-bit encoding of other national alphabets.
- This one is Latin, and includes most European symbols (eg accents).

ISO-8859-1 (Latin)



Char	Code	Name	Description
à	224	agrave	a grave
á	225	aacute	a acute
â	226	acirc	a circumflex
ã	227	atilde	a tilde
ä	228	auml	a umlaut
å	229	aring	a ring
æ	230	aelig	ae ligature
ç	231	ccedil	c cedilla
è	232	egrave	e grave
é	233	eacute	e acute
ê	234	ecirc	e circumflex
ë	235	euml	e umlaut
ì	236	igrave	i grave
í	237	iacute	i acute
î	238	icirc	i circumflex
ï	239	iuml	i umlaut
ð	240	eth	eth
ñ	241	ntilde	n tilde
ò	242	ograve	o grave
ó	243	oacute	o acute
ô	244	ocirc	o circumflex
õ	245	otilde	o tilde
ö	246	ouml	o umlaut
÷	247	divide	division sign
ø	248	oslash	o slash
ù	249	ugrave	u grave
ú	250	uacute	u acute
û	251	ucirc	u circumflex
ü	252	uuml	u umlaut
ý	253	yacute	y acute
þ	254	thorn	thorn
ÿ	255	yuml	y umlaut

Char	Code	Name	Description
	32	-	Normal space
!	33	-	Exclamation
"	34	quot	Double quote
#	35	-	Hash or pound
$	36	-	Dollar
%	37	-	Percent
&	38	-	Ampersand
'	39	-	Apostrophe
(	40	-	Open bracket
)	41	-	Close bracket
*	42	-	Asterisk
+	43	-	Plus sign
,	44	-	Comma
-	45	-	Minus sign
.	46	-	Period
/	47	-	Forward slash

Char	Code	Name	Description
0	48	-	Digit 0
1	49	-	Digit 1
2	50	-	Digit 2
3	51	-	Digit 3
4	52	-	Digit 4
5	53	-	Digit 5
6	54	-	Digit 6
7	55	-	Digit 7
8	56	-	Digit 8
9	57	-	Digit 9
:	58	-	Colon
;	59	-	Semicolon
<	60	lt	Less than
=	61	-	Equals
>	62	gt	Greater than
?	63	-	Question mark

Char	Code	Name	Description
@	64	-	At sign
A	65	-	A
B	66	-	B
C	67	-	C
D	68	-	D
E	69	-	E
F	70	-	F
G	71	-	G
H	72	-	H
I	73	-	I
J	74	-	J
K	75	-	K
L	76	-	L
M	77	-	M
N	78	-	N
O	79	-	O

Char	Code	Name	Description
P	80	-	P
Q	81	-	Q
R	82	-	R
S	83	-	S
T	84	-	T
U	85	-	U
V	86	-	V
W	87	-	W
X	88	-	X
Y	89	-	Y
Z	90	-	Z
[	91	-	Open square bracket
\	92	-	Backslash
]	93	-	Close square bracket
^	94	-	Pointer
_	95	-	Underscore

Char	Code	Name	Description
`	96	-	Grave accent
a	97	-	a
b	98	-	b
c	99	-	c
d	100	-	d
e	101	-	e
f	102	-	f
g	103	-	g
h	104	-	h
i	105	-	i
j	106	-	j
k	107	-	k
l	108	-	l
m	109	-	m
n	110	-	n
o	111	-	o

Char	Code	Name	Description
p	112	-	p
q	113	-	q
r	114	-	r
s	115	-	s
t	116	-	t
u	117	-	u
v	118	-	v
w	119	-	w
x	120	-	x
y	121	-	y
z	122	-	z
{	123	-	Left brace
|	124	-	Vertical bar
}	125	-	Right brace
~	126	-	Tilde
✕	127	-	(Unused)

Char	Code	Name	Description
	160	nbsp	Non-breaking space
¡	161	iexcl	Inverted exclamation
¢	162	cent	Cent sign
£	163	pound	Pound sign
¤	164	curren	Currency sign
¥	165	yen	Yen sign
¦	166	brvbar	Broken bar
§	167	sect	Section sign
¨	168	uml	Umlaut or diaeresis
©	169	copy	Copyright sign
ª	170	ordf	Feminine ordinal
«	171	laquo	Left angle quotes
¬	172	not	Logical not sign
	173	shy	Soft hyphen
®	174	reg	Registered trademark
¯	175	macr	Spacing macron

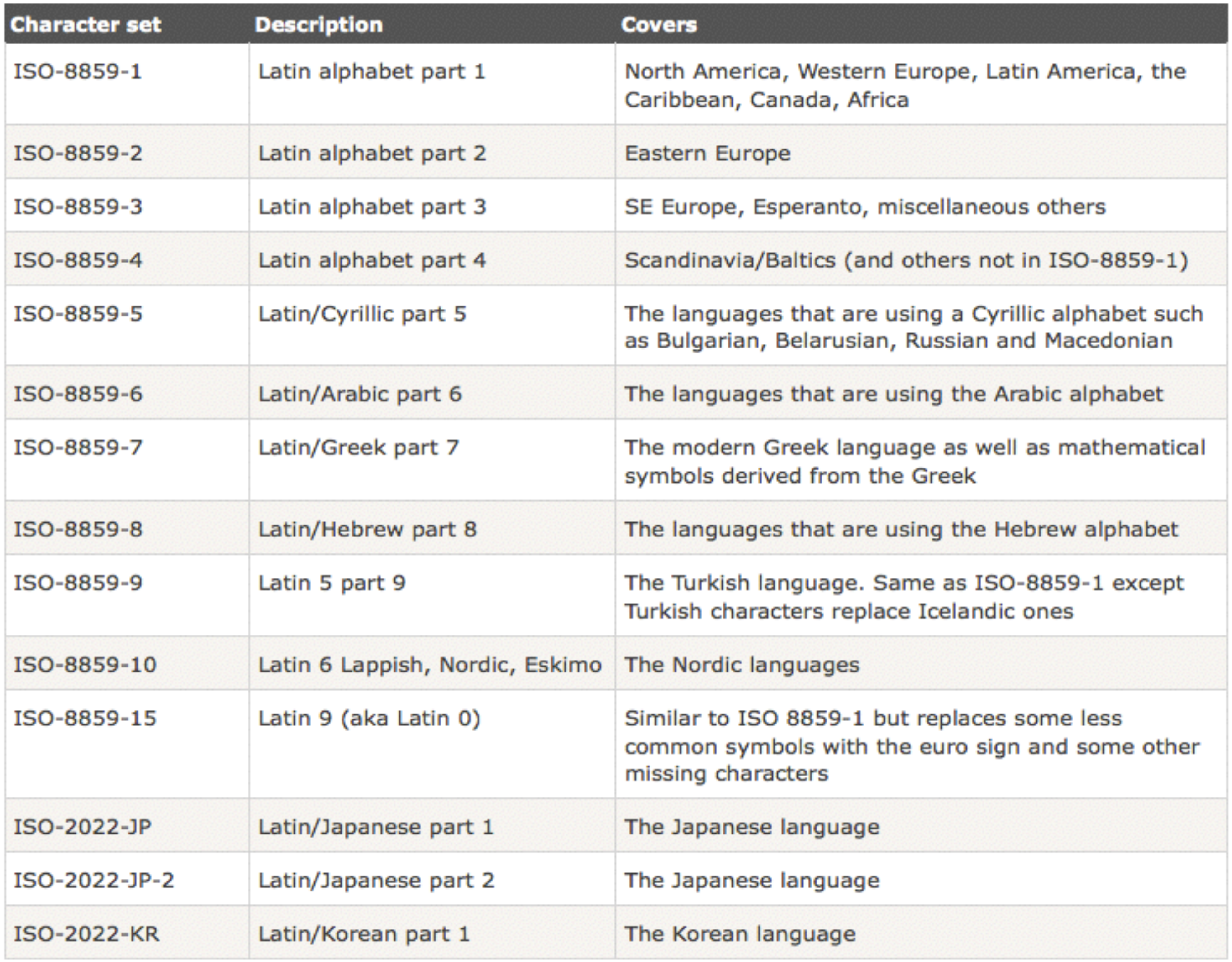
Char	Code	Name	Description
°	176	deg	Degree sign
±	177	plusmn	Plus-minus sign
²	178	sup2	Superscript 2
³	179	sup3	Superscript 3
´	180	acute	Spacing acute
µ	181	micro	Micro sign
¶	182	para	Paragraph sign
·	183	middot	Middle dot
¸	184	cedil	Spacing cedilla
¹	185	sup1	Superscript 1
º	186	ordm	Masculine ordinal
»	187	raquo	Right angle quotes
¼	188	frac14	One quarter
½	189	frac12	One half
¾	190	frac34	Three quarters
¿	191	iquest	Inverted question mark

Char	Code	Name	Description
À	192	Agrave	A grave
Á	193	Aacute	A acute
Â	194	Acirc	A circumflex
Ã	195	Atilde	A tilde
Ä	196	Auml	A umlaut
Å	197	Aring	A ring
Æ	198	AElig	AE ligature
Ç	199	Ccedil	C cedilla
È	200	Egrave	E grave
É	201	Eacute	E acute
Ê	202	Ecirc	E circumflex
Ë	203	Euml	E umlaut
Ì	204	Igrave	I grave
Í	205	Iacute	I acute
Î	206	Icirc	I circumflex
Ï	207	Iuml	I umlaut

Char	Code	Name	Description
Ð	208	ETH	ETH
Ñ	209	Ntilde	N tilde
Ò	210	Ograve	O grave
Ó	211	Oacute	O acute
Ô	212	Ocirc	O circumflex
Õ	213	Otilde	O tilde
Ö	214	Ouml	O umlaut
×	215	times	Multiplication sign
Ø	216	Oslash	O slash
Ù	217	Ugrave	U grave
Ú	218	Uacute	U acute
Û	219	Ucirc	U circumflex
Ü	220	Uuml	U umlaut
Ý	221	Yacute	Y acute
Þ	222	THORN	THORN
ß	223	szlig	sharp s

Char	Code	Name	Description
à	224	agrave	a grave
á	225	aacute	a acute
â	226	acirc	a circumflex
ã	227	atilde	a tilde
ä	228	auml	a umlaut
å	229	aring	a ring
æ	230	aelig	ae ligature
ç	231	ccedil	c cedilla
è	232	egrave	e grave
é	233	eacute	e acute
ê	234	ecirc	e circumflex
ë	235	euml	e umlaut
ì	236	igrave	i grave
í	237	iacute	i acute
î	238	icirc	i circumflex
ï	239	iuml	i umlaut

Char	Code	Name	Description
ð	240	eth	eth
ñ	241	ntilde	n tilde
ò	242	ograve	o grave
ó	243	oacute	o acute
ô	244	ocirc	o circumflex
õ	245	otilde	o tilde
ö	246	ouml	o umlaut
÷	247	divide	division sign
ø	248	oslash	o slash
ù	249	ugrave	u grave
ú	250	uacute	u acute
û	251	ucirc	u circumflex
ü	252	uuml	u umlaut
ý	253	yacute	y acute
þ	254	thorn	thorn
ÿ	255	yuml	y umlaut



Character set	Description	Covers
ISO-8859-1	Latin alphabet part 1	North America, Western Europe, Latin America, the Caribbean, Canada, Africa
ISO-8859-2	Latin alphabet part 2	Eastern Europe
ISO-8859-3	Latin alphabet part 3	SE Europe, Esperanto, miscellaneous others
ISO-8859-4	Latin alphabet part 4	Scandinavia/Baltics (and others not in ISO-8859-1)
ISO-8859-5	Latin/Cyrillic part 5	The languages that are using a Cyrillic alphabet such as Bulgarian, Belarusian, Russian and Macedonian
ISO-8859-6	Latin/Arabic part 6	The languages that are using the Arabic alphabet
ISO-8859-7	Latin/Greek part 7	The modern Greek language as well as mathematical symbols derived from the Greek
ISO-8859-8	Latin/Hebrew part 8	The languages that are using the Hebrew alphabet
ISO-8859-9	Latin 5 part 9	The Turkish language. Same as ISO-8859-1 except Turkish characters replace Icelandic ones
ISO-8859-10	Latin 6 Lappish, Nordic, Eskimo	The Nordic languages
ISO-8859-15	Latin 9 (aka Latin 0)	Similar to ISO 8859-1 but replaces some less common symbols with the euro sign and some other missing characters
ISO-2022-JP	Latin/Japanese part 1	The Japanese language
ISO-2022-JP-2	Latin/Japanese part 2	The Japanese language
ISO-2022-KR	Latin/Korean part 1	The Korean language

From w3schools.com

But

- 256 characters is not enough...
- if you need to cover more than one language
- if you're a mathematician
- if you're Chinese

And so we have unicode

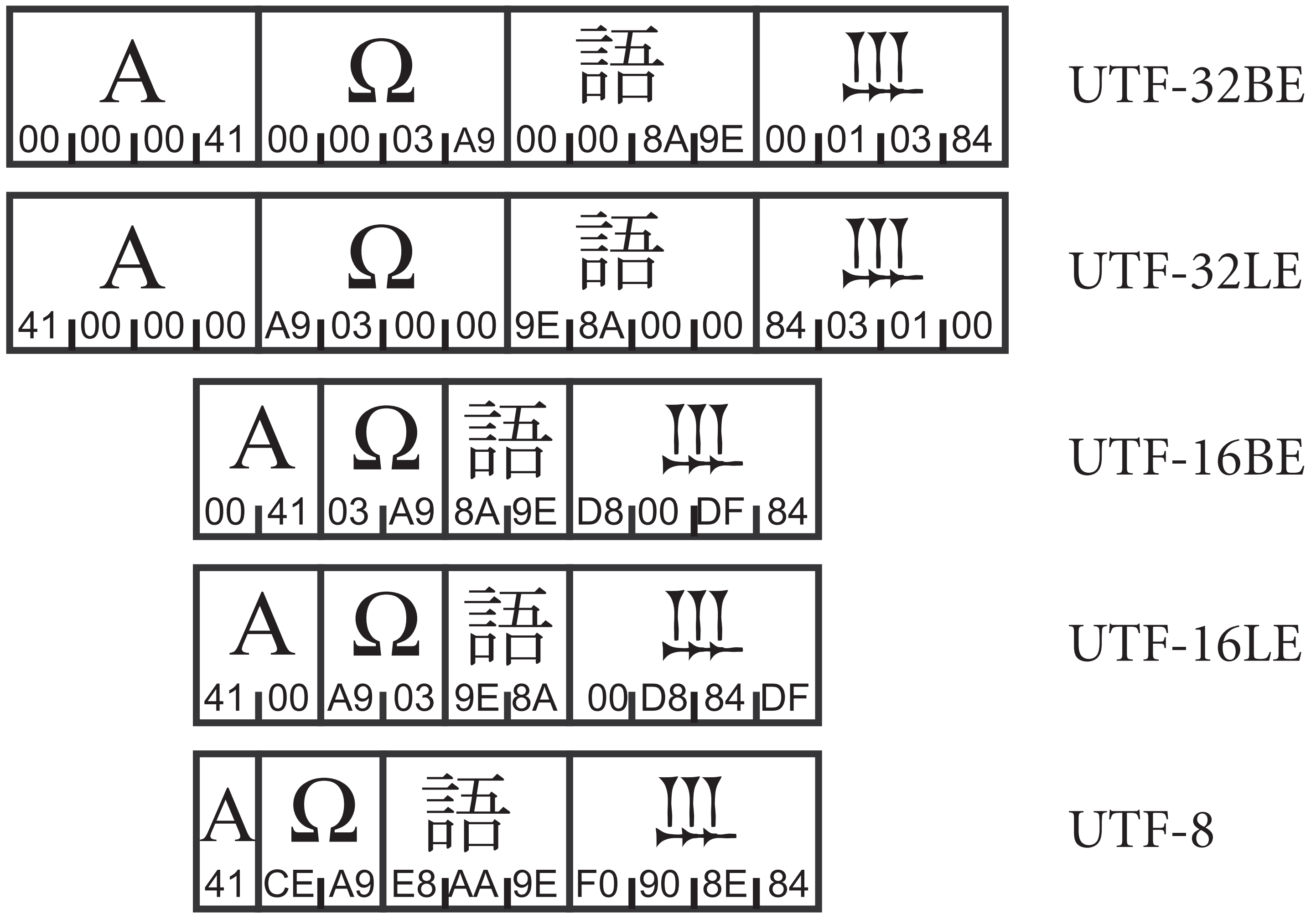
- Character set used in Java
- First 128 characters are ASCII
- First 256 are ISO-8859-1
- Total number of characters available is: 1,114,112 = 21 (just over 20) bits
- Lots of segments correspond to particular languages
- See http://www.unicode.org/

Unicode

- Distinguishes between the number of the character (the code point) and the way it is represented.
- So we have UTF-32 (represents character in 32 bits = 4 bytes)
- But also UTF-16 and UTF-8 (16 and 8 bits).
- UTF-8 is now standard on web applications.
- UTF-32 and UTF-16 have "big endian" and "little endian" variants, referring to the order of the bytes.

Unicode encodings

Figure 2-12. Unicode Encoding Schemes



A Ω 語 𐎄
00 00 00 41 00 00 03 A9 00 00 8A 9E 00 01 03 84
UTF-32BE
41 00 00 00 A9 03 00 00 9E 8A 00 00 84 03 01 00
UTF-32LE
00 41 03 A9 8A 9E D8 00 DF 84
UTF-16BE
41 00 A9 03 9E 8A 00 D8 84 DF
UTF-16LE
41 CE A9 E8 AA 9E F0 90 8E 84
UTF-8

UTF-8

Preferred Usage. UTF-8 is typically the preferred encoding form for HTML and similar protocols, particularly for the Internet. The ASCII transparency helps migration. UTF-8 also has the advantage that it is already inherently byte-serialized, as for most existing 8-bit character sets; strings of UTF-8 work easily with C or other programming languages, and many existing APIs that work for typical Asian multibyte character sets adapt to UTF-8 as well with little or no change required.

From Unicode Standard v 6.2

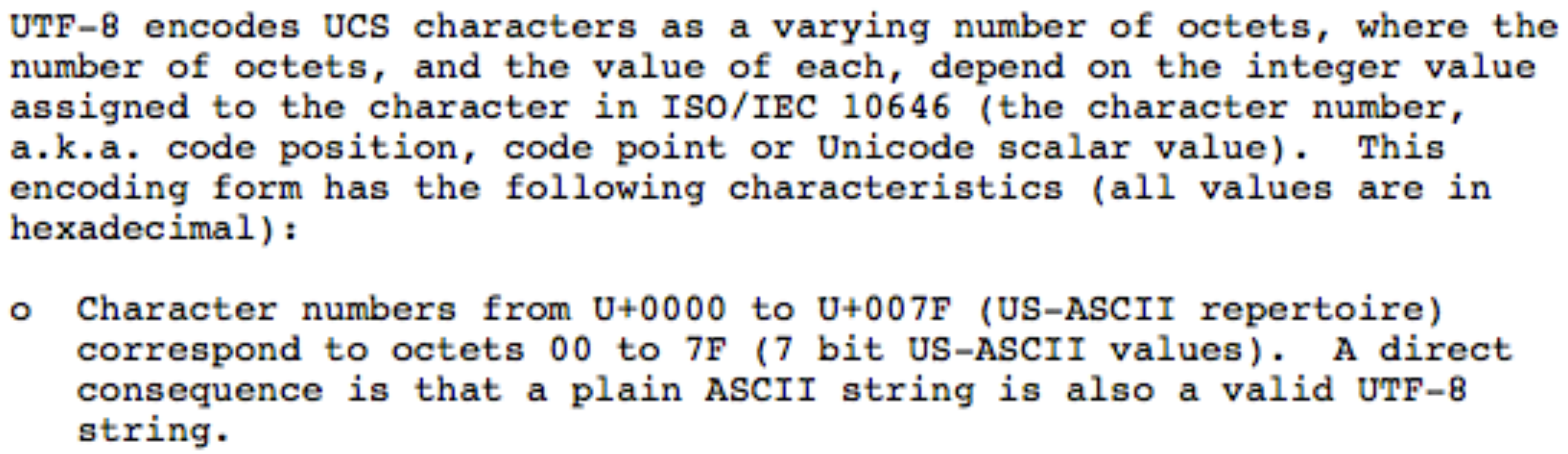
Translation

- We want you to use UTF-8 for internet and web-based protocols
- here are some reasons

UTF-8

- But UTF-8 uses different numbers of bytes to encode individual characters.
- It therefore has to be cleverly and carefully designed so that it is not ambiguous, and so that as much existing software continues to work with it as possible.
- The designer (Ken Thompson) was one of the lead designers of Unix.

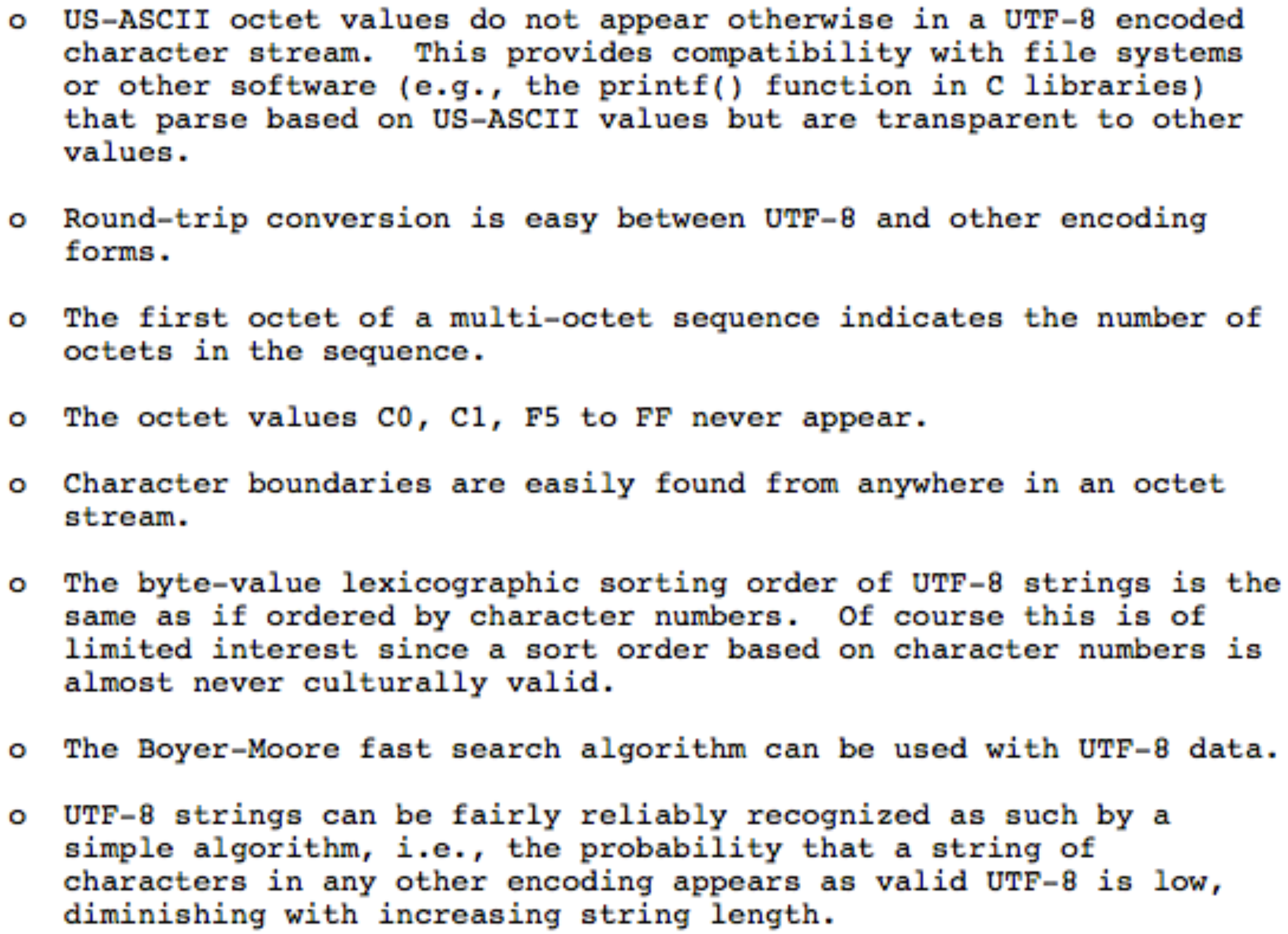
From the rfc for UTF-8



UTF-8 encodes UCS characters as a varying number of octets, where the number of octets, and the value of each, depend on the integer value assigned to the character in ISO/IEC 10646 (the character number, a.k.a. code position, code point or Unicode scalar value). This encoding form has the following characteristics (all values are in hexadecimal):

o Character numbers from U+0000 to U+007F (US-ASCII repertoire) correspond to octets 00 to 7F (7 bit US-ASCII values). A direct consequence is that a plain ASCII string is also a valid UTF-8 string.

From the rfc for UTF-8



o US-ASCII octet values do not appear otherwise in a UTF-8 encoded character stream. This provides compatibility with file systems or other software (e.g., the printf() function in C libraries) that parse based on US-ASCII values but are transparent to other values.

o Round-trip conversion is easy between UTF-8 and other encoding forms.

o The first octet of a multi-octet sequence indicates the number of octets in the sequence.

o The octet values C0, C1, F5 to FF never appear.

o Character boundaries are easily found from anywhere in an octet stream.

o The byte-value lexicographic sorting order of UTF-8 strings is the same as if ordered by character numbers. Of course this is of limited interest since a sort order based on character numbers is almost never culturally valid.

o The Boyer-Moore fast search algorithm can be used with UTF-8 data.

o UTF-8 strings can be fairly reliably recognized as such by a simple algorithm, i.e., the probability that a string of characters in any other encoding appears as valid UTF-8 is low, diminishing with increasing string length.