OK Some Basic Arithmetic Tutorials

The plan is to do this one the white board.

## Base 10 integer to binary conversion

Let’s take some ‘random’ number and convert to binary

357

What’s a way to express 357 as the sum of products of 2

256 + 128 is too big, so it’s an expression of

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

So if we write this (actually write by hand right to left)

101100101

Another way to think of this is repeated divisions by 2

101001101

Wait, that isn’t right? Oh but it is, it’ just backwards

## Binary to hex conversion

e.g. 0001 0110 0101 (357)

Separate things into blobs of 4 binary digits (pad with 0’s from the left)

Then just look up the result from a binary to hex table

357 in base 10 is 165 (in base 16)

## binary to hexadecimal conversion table

## Integer Add

This should be easy

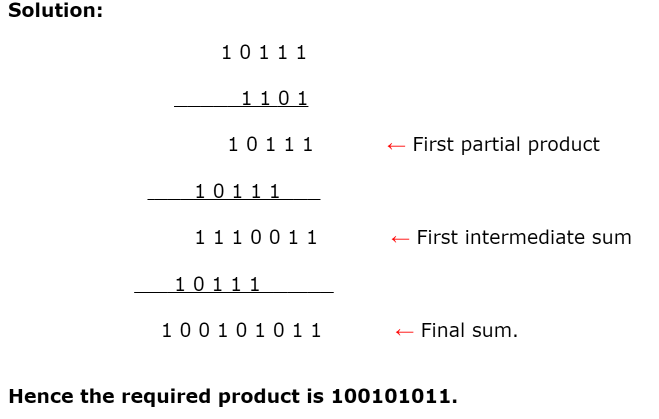
Add the individual bits, do the carry, worry about overflow if it’s on a computer (B+ F = 11+15=26)

## Integer Subtract

Same deal compare individual bits everyone is happy

On a computer you need a ‘sign bit’ by hand you just use a, well, sign.

## Integer Multiply



http://www.math-only-math.com/binary-multiplication.html

## Integer Divide

Recall base 10 long division

4 doesn’t go into 1 but it goes into 17 (4x4 =16), remainder 1

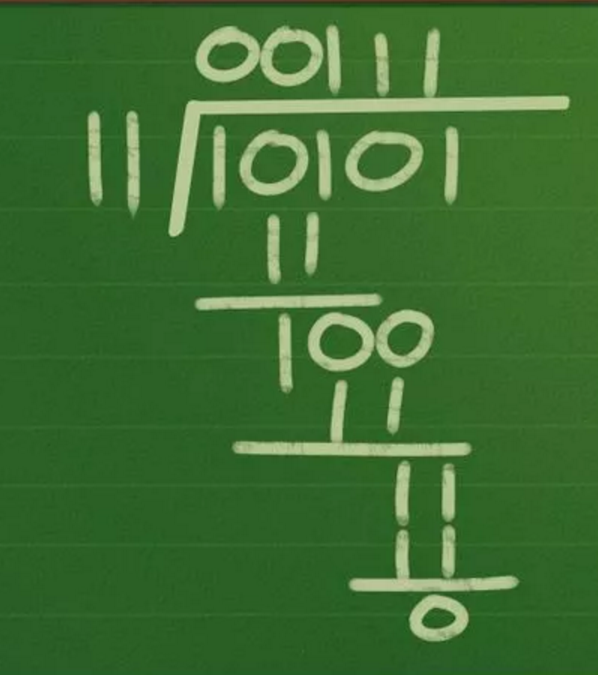
So we put in a 4. Keep the one, drop down the 2, four goes into 12 3 times, so 43\*4 = 172

(A better way to think of this is really 4\*40 = 160, then you need to worry about what’s left, but close enough for our purposes in being reliably systematic.)

For binary long division we sort of do the same thing

Lets do 10101/11 (base 2)

11 doesn’t go into 1, or 10, so you get 001, then we put the 11 underneath and have a 101-11 (=10), etc



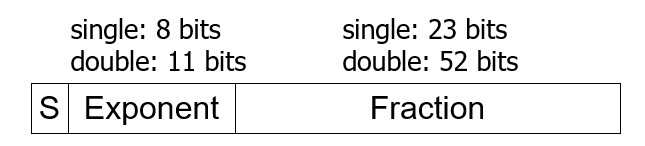
<http://www.wikihow.com/Divide-Binary-Numbers>

(apparently MS equation editor doesn’t have any easy way to do long division and I haven’t got the time or desire to fight with it unless a blind kid shows up in my class).

# Binary Decimal

Long summary – it’s possible to use binary (base 2) to represent decimal numbers, and to a limited extent we do this with floating point. 11011.101 is a perfectly valid binary number. But that isn’t really how we do stuff in practice. (Floating point is next). Binary decimal factors in, but we have the exponent part too.

## Base 10 Decimal to Floating point representation





(See lecture 3 slide 20 for Screen reader compatible version)

* Normalize significand: 1.0 ≤ |significand| < 2.0
  + Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  + Significand is Fraction with the “1.” restored
* Exponent: excess representation: actual exponent + Bias
  + Ensures exponent is unsigned

Single: Bias = 127; Double: Bias = 1023 (slides say 1203, that’s a typo)

(note on terminology – the “fraction” part is sometimes called mantissa or significand)

Also note, I have no reason to ask you to do conversions to double precision, it’s the same steps, just more precision.

Example convert -5.75 to binary decimal first then to floating point

-5 is -101.

0.75 is 0.5 + 0.25 = 0.11

1. So -101.11
2. Shift the decimal to the leftmost 1 (the point floated)
3. -1.01110 \* 2 shifts
4. Exponent = bias + #shifts = 127 + 2 = 129 (base 10), 128 in binary is 1000 0000 in base 2, so add 1
5. Put them together using the format above/earlier (1 1000 0001 01110)
6. Note the 01110 is the 1.01110 but you ignore the first digit (the 1 is implied in the standard)
7. Pad with 0’s to get to 32 bits (exponent and fraction part could have 0s theoretically)
8. 1 1000 0001 01110 0000 0000 0000 0000 0000 0000 000

Note lets say you just wanted to convert 0.75 (0.110) , so then your shift would be -1, so your exponent would be 127 – 1 = 126. Or say 0.125 (base 10) would be 0.001 in binary, so then you shift right 3, exponent is then 127 -3 = 124

Converting Back the other way is sort of annoying. You need to extract the relevant digits

0 10000010 00100000000000000000000

Sign bit tells us it’s positive

Exponent was 127 + shift, so we recover the number from the 8 exponent bits, subtract 127. (130 – 127)

That last horrible blob 00100000000000000000000 is of the form 1.fffffffffffffff

So 1.001 (binary) exponent 3

1001 Just convert to base 10. (9).

**Rounding:** There is a weird representation convention on the ‘fraction’ part let’s say it was 4 bits long not 23 and had the following value

1001. No problem. But if the correct representation was 10011 but the last one was dropped the convention is to round up the last bit – to 1010. Exactly this happens with 23 bits… just with more digits. The least significant bit rounds up if there is a dropped 1 bit.

## Floating point addition

Let’s say I want to add (base 10 ) 1.23456x10-5 to 7.89. Get everything in the same exponent

0.0000123456 + 7.89 or 1.23456x10-5 + 789000 x 10-5

Binary is pretty much the same deal

Let’s add (these aren’t the same numbers as above)  
(Format: sign bit, 8 bits for exponent, 23 bits for fraction)

A= 0 1101 0111 111 0011 1010 0000 1100 0011

B= 0 1101 0001 000 1110 0101 1111 0001 1100

Exponent of A = 215

Exponent of B = 209

Normalize B up to 215 (shift B right by six places)

Fraction of b is 1.000 1110 0101 1111 0001 1100

Shift right 6 total places (5 0’s and then the leftmost digit)

0.000 0010 0011 1001 0111 1100 ~~011100~~

The last bits are just lost

Add the Fraction/Significand parts

A (Fraction Part) = 1.111 0011 1010 0000 1100 0011

B (Fraction Part) = 0.000 0010 0011 1001 0111 1100

So the representation,

Sign bit is positive still so that’s 0

Exponent is 215 (represented in binary)

<https://www.youtube.com/watch?v=IiHK18n0pm4>

Has (a 27 minute total) this tutorial with two sample problems.

## Floating point subtraction

Like addition, only with subtraction. (no really, it’s the same thing).

## Floating point multiplication

For the sake of actually getting anything done let’s make a (simple ish) floating point representation – an 8 bit float. 1 sign bit, 4 exponent bits, and 3 fraction bits, with a bias of 7  
**Side Note**: There are other minifloats with 3 bit exponents and 4 bit fraction, it all works the same way, you just have a bias of 3 in that case.

X = 0 1001 010

Y = 0 0111 110

So we multiply them

If we convert to scientific notation (note that I’ve dealt with the bias here).

X is 1.01 x22 Y is 1.110 x20

So the exponent part… add them. 2+0 = 2

Multiply the mantissas (the fraction parts) as usual: 1.01\* 1.110 = 10.0011

Normalize 1.00011 x 23

Sign bit is 0 (technically we add the sign bits mod 2, AKA XOR the sign bit)

Convert back to floating point – well, we can’t represent all of those bits. So it’s just a fraction of 000

An exponent of 3+7 = 10

The bias in floating point actually makes things more of a nuisance, since you need to add the exponents, that either requires converting from the bias format back to a (signed) integer and adding, and then converting back to a bias or having a special hardware unit for handling bias additions

## Floating point Division

Same as multiplication except subtract exponents and divide not multiply

<https://www.youtube.com/watch?v=fi8A4zz1d-s>

is a longwinded example if you think you need it

short summary

for X/Y = (Xf/Yf)\*2(Xe-Ye)

normalize exponent.