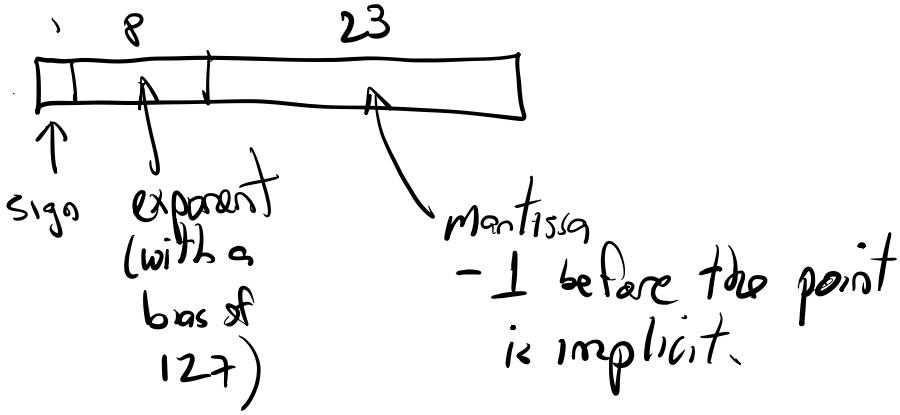
32-bit IEEE Floating Pt



In C, how do we extract the bits of a fp number?

- use masks & shifts
  - needs to be an integer variable, preferably unsigned.
- So copy the bits from the fp number to an unsigned int.

```
float f = 29.314;
unsigned int x = (unsigned int) f;
```

✗ C won't work!
- converts 29.314 to 29
  - totally different bit pattern.

Instead, create an unsigned int pointer to point to f, then dereference the pointer:

```
float f = 29.314;
unsigned int x;
unsigned int *p = (unsigned int *) &f;
x = *p;
```

OR, even more concisely:

```
float f = 29.314;
unsigned int x = *((unsigned int *) &f);
```

dereference.

How do we extract the sign bit, exponent bits, and mantissa bits?

Sign:

```
unsigned int sign = (x >> 31) & 1;
```

(x >> 31): moves sign bit to the rightmost position.

1: mask

Exponent bits:
- 8 bits starting at bit 23.

```
#define MASK8 0xff
#define EXPSHIFT 23

unsigned int exp = (x >> EXPSHIFT)
                   & MASK8;
```

- this result will be the actual exponent plus 127

Mantissa Bits
- rightmost 23 bits (no shift needed)

```
#define MASK23 0x7fffff
```

3 1's, 20 1's

or

```
#define MASK23 ((1 << 23) - 1)
```

outer parens are very important.
- see next page

```
#define W 3+4
int x = W; //int x = 3+4;
int y = W*2; //int y = 3+4*2;
```

- y gets 11, not 14

Should have written

```
#define W (3+4)
int y = W*2; // int y = (3+4)*2;
             // = 14.
```

```
unsigned int mant = X & MASK23;
```

- contain only the digits (bits) after the point.
  - not the implicit 1 before the point.

How do we insert the 1 before the point into the mantissa bits?

- the 23 bits already there are at positions 0 through 22

- need to insert a 1 at bit position 23.

```
mant = mant | (1 << 23);
```

OR

```
Mant |= (1 << 23);
```

How do we add floating point numbers?

- in scientific notation.

Decimal: $23.314 + 6975.2$

$= 2.3314 \times 10^1 + 6.9752 \times 10^3$

Need to make the exponents the same, then add the mantissas.

- shift the number w/ the smaller exponent to the right, each time increasing the exponent:

$2.3314 \times 10^1 = 0.23314 \times 10^2$

$= 0.023314 \times 10^3$

$$\begin{array}{r} \text{Result} = 0.023314 \times 10^3 \\ +\ 6.975200 \times 10^3 \\ \hline 6.998514 \times 10^3 \end{array}$$

- the result of the addition may have two non-zero digits before the decimal point. If so, renormalize by shifting the mantissa of the result to the right by one, and add one to the exponent.

Example: If the mantissa of the result is 32.64×10^8, normalize it to 3.264×10^9.

The algorithm for binary floating point addition is identical!

- denormalize the smaller number so they have same exponent
- add the mantissas
- renormalize the result.

Note: If one of the numbers is negative, subtract the smaller number from the larger number and the sign of the result is the sign of the larger number. If the numbers are both negative, add them together and the sign of the result is negative.

How to multiply floating point numbers?

$$(a \times 10^b) \times (c \times 10^d)$$

$$= (a \times c) \times \underbrace{(10^b \times 10^d)}_{10^{b+d}}$$

$$= \underbrace{(a \times c)}_{\text{product of mantissa}} \times 10^{\overbrace{b+d}^{\text{sum of the exponents}}}$$

So:

- mantissa of the result is the product of the mantissa's of the operands
- exponent of the result is the sum of the exponents of the operands
- sign of the result is just the XOR of the signs of the operands.

In binary, multiplication works exactly the same.

- You'll be doing this in Assignment 2.

Important:

- the stored exponent has a bias (127).
  - you can remove the biases from the exponents before adding, and then apply the bias before storing the result.
    - but you don't need to!

- actual result = (expA − 127) + (expB − 127)
- stored result = actual result + 127
  $= (expA - 127) + (expB - 127) + 127$
  $= expA + expB - 127$

So, add the stored exponents together and subtract one bias (127).

- Insert the leading 1 into each mantissa before multiplying them (integer multiplication).
- You may need to renormalise, due to the mantissa of the result having two non-zero bits before the point.