How to represent real numbers

- In decimal scientific notation: 5.280×10^3
  - sign
  - fraction
  - base (i.e., 10) to some power
- Most of the time, usual representation 1 digit at left of decimal point
  - Example: $- 0.1234 \times 10^6$
- A number is *normalized* if the leading digit is not 0
  - Example: -1.234×10^5

Real numbers representation inside computer

- Use a representation akin to scientific notation

  sign x *mantissa* x *base* ^{exponent}

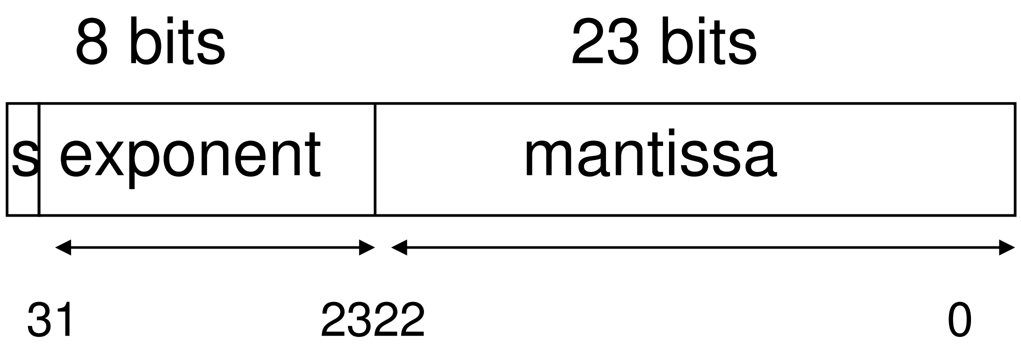
- Many variations in choice of representation for
  - mantissa (could be 2's complement, sign and magnitude etc.)
  - base (could be 2, 8, 16 etc.)
  - exponent (cf. mantissa)
- Arithmetic support for real numbers is called *floating-point* arithmetic

Floating-point representation: IEEE Standard

- Basic choices
  - A single precision number must fit into 1 word (4 bytes, 32 bits)
  - A double precision number must fit into 2 words
  - The base for the exponent is 2
  - There should be approximately as many positive and negative exponents
- Additional criteria
  - The mantissa will be represented in sign and magnitude form
  - Numbers will be normalized

Example: MIPS representation of IEEE Standard

- A number is represented as : $(-1)^{S} \cdot F \cdot 2^{E}$
- In single precision the representation is:



MIPS representation (continued)

- Bit 31 sign bit for mantissa (0 pos, 1 neg)
- Exponent 8 bits ("biased" exponent, see next slide)
- mantissa 23 bits : always a *fraction* with an implied binary point at left of bit 22
- Number is normalized (see implication next slides)
- 0 is represented by all zero's.
- Note that having the most significant bit as sign bit makes it easier to test for positive and negative

Biased exponent

- The “middle” exp. (01111111) will represent exponent 0, i.e. 127
- All exps starting with a “1” will be positive exponents .
  - Example: 10000001 is exponent 2 (10000001 -01111111)
- All exps starting with a “0” will be negative exponents
  - Example 01111110 is exponent -1 (01111110 - 01111111)
- The largest positive exponent will be 11111111, about 10^{38}
- The smallest negative exponent is about 10^{-38}

Normalization

- Since numbers must be normalized, there is an implicit "one" at the left of the binary point
- No need to put it in (improves precision by 1 bit)
- But need to reinstate it when performing operations.
- In summary, in MIPS a floating-point number has the value:

$$(-1)^{S} \cdot (1 + \text{mantissa}) \cdot 2^{(\text{exponent} - 127)}$$

Double precision

- Takes 2 words (64 bits)
- Exponent 11 bits (instead of 8)
- Mantissa 52 bits (instead of 23)
- Still biased exponent and normalized numbers
- Still 0 is represented by all zeros
- We can still have *overflow* (the exponent cannot handle super big numbers) and *underflow* (the exponent cannot handle super small numbers)

Floating-Point Addition

- Quite “complex” (more complex than multiplication)
- Need to know which of the addends is larger (compare exponents)
- Need to shift “smaller” mantissa
- Need to know if mantissas have to be added or subtracted (since it’s a sign/magnitude representation)
- Need to normalize the result
- Correct round-off procedures are not simple (not covered in detail here)

One of the 4 round-off modes

- Round to nearest even
  - Example 1: in base 10. Assume 2 digit accuracy.

    $$3.1 * 10^0 + 4.6 * 10^{-2} = 3.146 * 10^0$$

    clearly should be rounded to $3.1 * 10^0$
  - Example 2:

    $$3.1 * 10^0 + 5.0 * 10^{-2} = 3.15 * 10^0$$

    By convention, round-off to nearest “even” number $3.2 * 10^0$
- Other round-off modes: towards 0, $+\infty$, $-\infty$

F-P add (details for round-off omitted)

1. Compare exponents . If e1 < e2, swap the 2 operands such that
   $d = e1 - e2 >= 0$. Tentatively set exponent of result to e1.
2. Insert 1's at left of mantissas. If the signs of operands differ, replace 2nd mantissa by its 2's complement.
3. Shift 2nd mantissa d bits to the right (this is an arithmetic shift, i.e., insert either 1's or 0's depending on the sign of the second operand)
4. Add the (shifted) mantissas. (There is one case where the result could be negative and you have to take the 2's complement; this can happen only when $d = 0$ and the signs of the operands are different.)

F-P Add (continued)

5. Normalize (if there was a carry-out in step 4, shift right once; else shift left until the first “1” appears on msb)
6. Modify exponent to reflect the number of bits shifted in previous step

Example

- Add decimal: 0.375 + 0.75
- $3/2^3 + 3/2^2 = 0.011 + 0.11 = 1.1 \times 2^{-2} + 1.1 \times 2^{-1}$
- Now add:
- Align fractions: $0.11 \times 2^{-1} + 1.1 \times 2^{-1}$
- Add fractions: 10.01×2^{-1}
- Normalize: $10.01 \times 2^{-1} = 1.001 \times 2^0$
- Round: Not needed
- $1.001 \times 2^0 = 1 + 1/2^3 = 1 + 1/8 = 1.125$ decimal
- In IEEE single precision 0.75 i.e. 0.11 is

  0 0111 1110 100 0000 0000 0000 0000 0000 why?

Using pipelining

- Stage 1
  – Exponent compare
- Stage 2
  – Shift and Add
- Stage 3
  – Round-off , normalize and fix exponent
- Most of the time, done in 2 stages.

Floating-point multiplication

- Conceptually easier

1. Add exponents (careful, subtract one “bias”)
2. Multiply mantissas (don’t have to worry about signs)
3. Normalize and round-off and get the correct sign

Pipelining

- Use tree of “carry-save adders” (cf. CSE 370) Can cut-it off in several stages depending on hardware available
- Have a “regular” adder in the last stage.

Special Values

- Allow computation to continue in face of exceptional conditions
  - For example: divide by 0, overflow, underflow
- Special value: NaN (Not a Number; e.g., sqrt(-1))
  - Operations such as 1 + NaN yield NaN
- Special values: $+\infty$ and $-\infty$ (e.g, 1/0 is $+\infty$)
- Can also use “denormal” numbers for underflow and overflow allowing a wider range of values