CSC 2400: Computer Systems

IEEE Floating Point Standard

Floating Point Numbers

- Decimal System: 11.625 analyzed as

10^1	10^0		10^{-1}	10^{-2}	10^{-3}
1	1	.	6	2	5

$11.625 = (1 \times 10) + 1 + (6 \times 10^{-1}) + (2 \times 10^{-2}) + (5 \times 10^{-3})$

- Binary System:

$$\begin{array}{cccccccc} 2^3 & 2^2 & 2^1 & 2^0 & . & 2^{-1} & 2^{-2} & 2^{-3} \\ 1 & 0 & 1 & 1 & . & 1 & 0 & 1_2 \end{array} = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) = 11.625_{10}$$

Floating Point Numbers

Table 1.5 Binary Weights for an 8-Bit Fraction

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

You try it:

1 0 0 1 0 . 0 1 0 0 1 =

How to Store FP Numbers?

- We have no way to store the point separating the whole part from the fractional part!
- Standard committees (IEEE) came up with a way to store floating point numbers

FP Normalization

- Every FP binary number (**except for zero**) can be normalized by choosing the exponent so that the radix point falls to the right of the leftmost 1 bit.

$37.25_{10} = 100101.01_2 = 1.0010101 \times \mathbf{2}^{5}$

$7.625_{10} = 111.101_2 = 1.11101 \times \mathbf{2}^{2}$

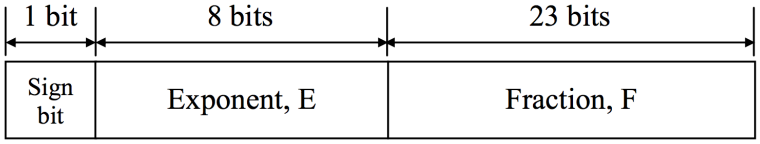
$0.3125_{10} = 0.0101_2 = 1.01 \times \mathbf{2}^{-2}$

fraction / mantissa / significand

exponent

IEEE Floating Point Standard (Single Precision, 32 bits)

- Sign-Magnitude: sign bit *S*, exponent *E* and fraction ***F***

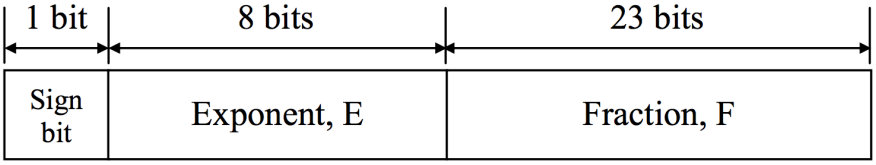


$$N = -1^{S} \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \; 1 \le \text{exponent} \le 254$$

- Special values:
  - E = 0, F = 0 represents 0.0
  - All bits in E equal to 1 (value 255), F= 0 represents *±infinity*
  - All bits in E equal to 1, *F* = 0 represents NaN (Not a Number), used for debugging, catching errors, and other special purposes

IEEE Floating Point Standard (32 bits)

- Sign-Magnitude: sign bit *S*, exponent *E* and fraction ***F***



- The binary exponent is not stored directly. Instead, *E* is the sum of the true exponent and 127. This *biased exponent* is always non-negative (seen as magnitude only).
- The fraction part assumes a normalized significand in the form 1.sssssssss (so we get the extra leading bit for free)

How would 15213.0 be stored?

- First, $15213_{10} = 11101101101101_2$
- Normalize to $1.1101101101101_2 \times 2^{13}$
- The true exponent is 13, so the biased E is

  E = 13 + 127 (Bias) = 140 = $\mathbf{10001100_2}$

- The fraction is

  F = $\underline{\mathbf{11011011011}}0000000000_2$

Floating Point Representation:								
Hex:	4	6	6	D	B	4	0	0
Binary:	0100	0110	0110	1101	1011	0100	0000	0000

How would 23.75 be stored?

- First, 23.75_{10} = _ _ _ _ _ . _ _ $_2$
- Normalize to _ . _ _ _ _ _ _ _ $_2$ x 2^4
- Biased exponent is

- Fraction is

IEEE Floating Point Representation:

How would -23.75 be stored?

- Just change the sign bit:

Hex:	4	6	6	D	B	4	0	0
Binary:	1100	0001	1011	1110	0000	0000	0000	0000

- Do not take the two's complement!

Exercise 1

- Find the IEEE representation of 40.0

Approximations: How would 0.1 be stored?

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

- First, 0.1_{10} = _ . _ $_2$
- Normalize to _ . _ $_2$ x 2^{-4}
- Biased exponent is
- Fraction is

IEEE Floating Point Representation:

Approximations: How would 0.1 be stored?

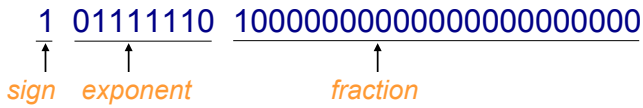
Table 1.5 Binary Weights for an 8-Bit Fraction

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

- In general, it is dangerous to think of floating point values as being "exact"
- Fractions will probably be approximate
  - If the fraction can be exactly expressed in binary, it might still be exact, like 1/2
  - But for example, 1/10 will be an approximate value

Reverse Your Steps:

- Single-precision IEEE floating point number:



  - Sign is 1 – number is negative
  - Exponent field is 01111110 = 126 (decimal)
  - Fraction is 100000000000… = 0.5 (decimal)

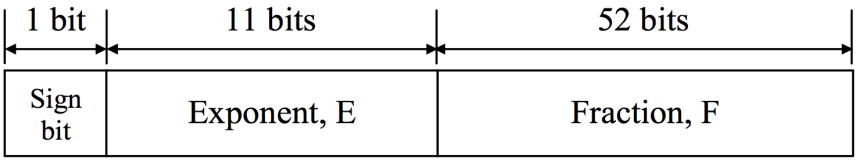
- Value = $-1.1_2 \times 2^{(126-127)} = -1.1_2 \times 2^{-1} = -0.11_2 = -0.75_{10}$

Exercise - Reverse your Steps

- Convert the following 32 bit number to its decimal floating point equivalent:

```
0    10000011  10011000..0
```

IEEE Floating Point Standard (Double Precision, 64 bits)



$$N = -1^{S} \times 1.\text{fraction} \times 2^{\text{exponent}-1023},\ 1 \le \text{exponent} \le 2046$$

- Exponent with all bits 1 (value 2047) is reserved to represent ±infinity (if fraction is 0) and NaN (if fraction is not 0)

Floating Point in C

- C Guarantees Two Levels
  - `float` single precision
  - `double` double precision
- Conversions
  - Casting between `int`, `float`, and `double` changes numeric values
  - `Double` or `float` to `int`
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range
  - `int` to `float`
    - Round according to rounding mode
  - `int` to `double`
    - Exact conversion, as long as int has ≤ 53 bit word size

`int` to `float` Rounding – Try It Out

```
int i = 0x1000003;
float f = (int)i;

printf("%x\n", i);
i = (int)f;
printf("%x\n", i);
```

Why is this important? Ariane 5!

- June 5, 1996
- Exploded 37 seconds after liftoff
- Cargo worth $500 million

- Why
  - Computed horizontal velocity as 64-bit floating point number
  - Converted to 16-bit integer
  - Worked OK for Ariane 4
  - Overflowed for Ariane 5
    - Used same software