Floating Point

Fractional Binary Number

representation:

- for $w = i + j + 1$ bits data b

$$\sum_{k=-j}^{i} b_k \times 2^k$$

for example:

- $5 + 3/4 = 23/4 = 101.11_2$
- $17/16 = 23/16 = 1.0111_2$

Limitations

- Can only exactly represent numbers of the form of $x/2^k$
- Just one setting of binary point within the w bits, which means that very small value or very large value cannot be represented

IEEE Floating Point Definition

IEEE Standard 754

Driven by numerical concerns:

- Nice standards for rounding, overflow, underflow
- But Hard to make fast in hardware
  - Numberical Analysts predominated over hw designers in defining standard

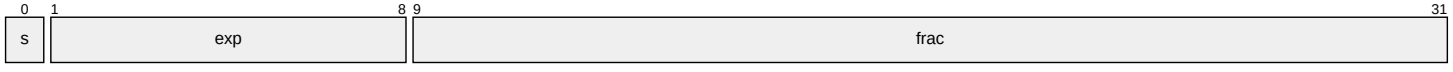
Representation

Form

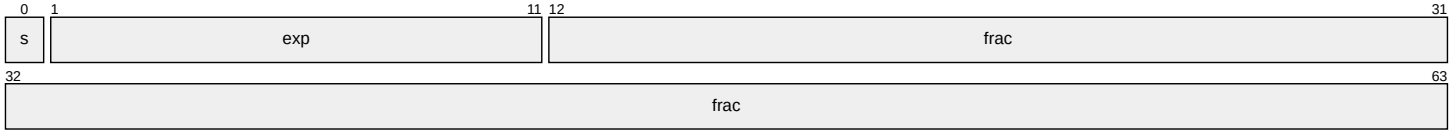
$$(-1)^s M 2^E$$

- s: sign bit
- M: mantissa fractional value in $[1.0, 2.0)$
- E: exponent

Encoding



Single Precision



Double Precision

There is three kinds of `float`: **normalized**, **denormalized**, **special**

normalized

$E = \exp - B$
$B = 2^{k-1} - 1$ where k is number of exp bits

- single: 127
- double: 1023

$M = 1.xxxx$
minumum when $\text{frac} = 0000\ldots \quad (M = 1.0)$
maximum when $\text{frac} \ = 1111\ldots \quad (M = 2.0 - \epsilon)$

denormalized

when `exp=000...0`
$\exp = 1 - Bias$
$M = 0.xxxx$

special

when `exp = 111...1`

- case `exp = 111...1, frac = 000...0`
  - repr ∞
  - operation that overflows
- case `exp = 111...1, frac = 111...1`
  - repr `NaN`
  - repr case when no numeric value can be determined
    - e.g., `sqrt(-1)`, `inf - inf`, `inf * 0`

```c
#include <stdio.h>

int main() {
  unsigned x_a = 0b0'11111111'00000000000000000000000;
  unsigned x_b = 0b0'11111111'00000000000000000000001;
  unsigned x_c = 0b0'01111111'00000000000000000000000;
  float a = *(float*)&x_a;
  float b = *(float*)&x_b;
  float c = *(float*)&x_c;
  double cx = c;
  printf("%08x: %f\n", x_a, a);
  printf("%08x: %f\n", x_b, b);
  printf("%08x: %f\n", x_c, c);
  printf("%016llx: %f\n", *(unsigned long long *)&cx, cx);
  return 0;
}
```

```
7f800000: inf
7f800001: nan
3f800000: 1.000000
3ff0000000000000: 1.000000
```

Properties

- FP0 is Same as Int0
- Can (almost) use unsigned int comparison

Arithmetic

$x + y = \text{Round}(x + y)$
$x \times y = \text{Round}(x \times y)$
Idea:

1. compute exact result
2. Make it fit into desired precision
    - overflow if too large
    - **round** to fit into frac

Rounding

- Twowards zero
- Round down
- Round up
- **Nearest Even***(default)

Nearest Even is default rounding mode
Any other kind rounding mode is hard to get without dropping into assembly, but C99 has support for rounding mode management.
This rounding mode is used because **reduced statistically bias**.
For binary fractional numbers:

- "Even" when least significant bit is 0
- "Half way" when bits to right of rounding position is $100\ldots_2$

so for example of rounding to neareast $1/4$:

Binary Value	Rounded	Action
`10.00011`	`10.00`	(<1/2)Down
`10.00110`	`10.01`	(>1/2)Up
`10.11100`	`11.00`	(=1/2)Up
`10.10100`	`10.10`	(=1/2)Down

`BBGRXXX`

- `G` : **G**uard bit: LSB of result
- `R` : **R**ound bit: first bit of removed
- `X` : **S**ticky bits: OR of remaining bits(001 = 1, 000 = 0)

Round up conditions

1. R = 1, S = 1 -> `>.5`
2. G = 1, R = 1, S = 0 -> Round to even

```c
#include <stdio.h>

int main() {
  unsigned long long
           tb = 0b0'10000010000'0000000000000000000010100000000000000000000000000000;
  unsigned xb =    0b0'10000001'01000000000000000000011;
  double t = *(double*)&tb;
  float x = t;
  for(int i=31; i>=0;i--) {
    if(i == 31 - 1) {
      printf("/");
    } else if (i == 31 - 1 - 8){
      printf("/");
    }
    printf("%d", !!((*(unsigned *)&x) & (1<<i)));
  }
  printf("\n");
  printf("%f", x);
}
```

```
0/10010000/00000000000000000000010
131072.031250
```

Float Quiz