Floating Point

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
- Practice problem

Introduction

- **A floating-point representation encodes rational numbers of the form $V = x \times 2^y$**

- **It is useful for performing computations involving**
  - very large numbers ($|V| \gg 0$)
  - numbers very close to 0 ($|V| \ll 1$)
  - and more generally as an approximation to real arithmetic

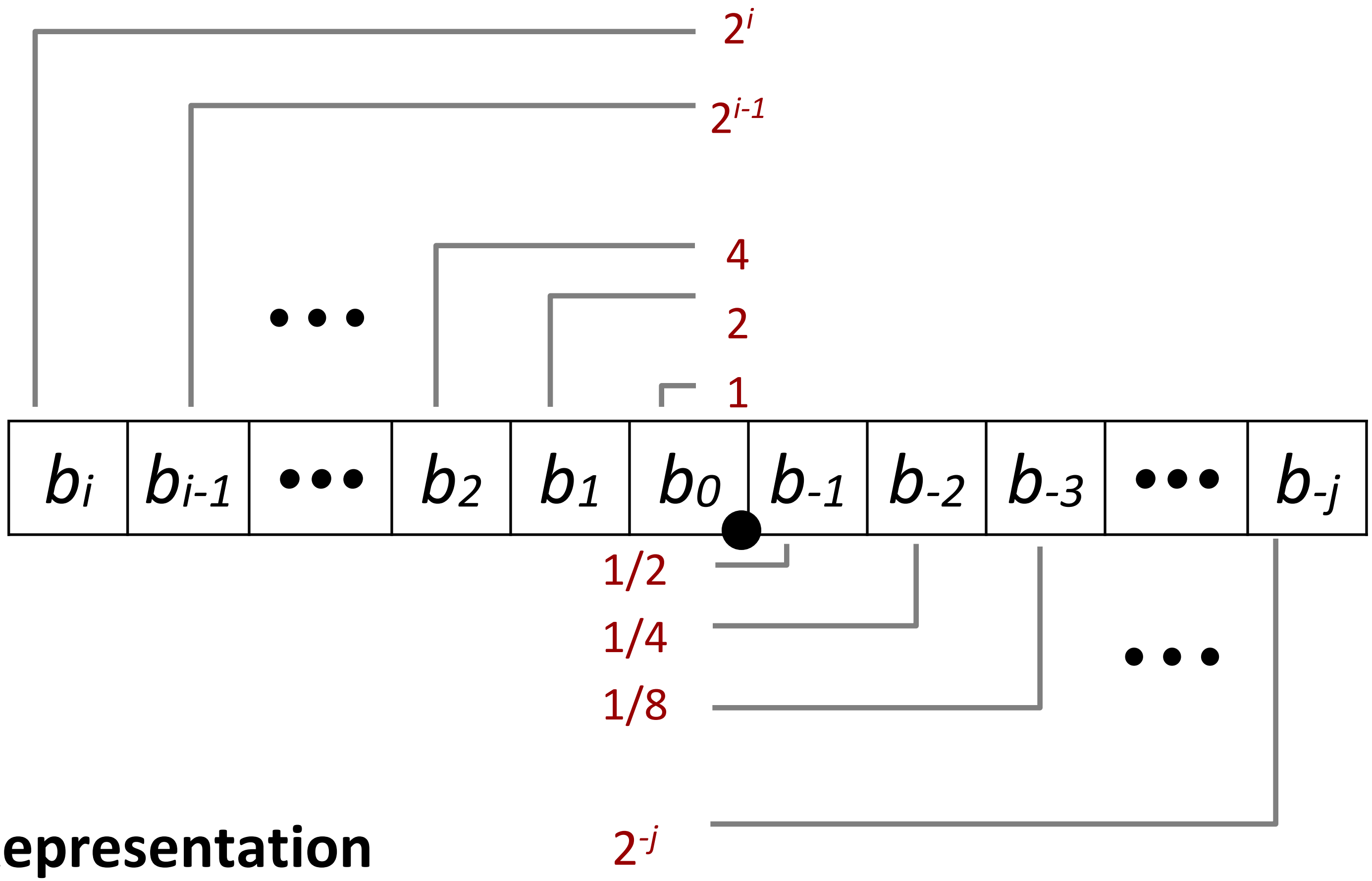
- **What issues come up when trying to devise a data representation for floating point numbers?**
  - It turns out these issues are more complicated than representing integers

Fractional binary numbers

- A first step in understanding floating-point numbers is to consider binary numbers having fractional values

- What is 1011.101_2?

Fractional binary numbers



- **Representation**
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Fractional binary numbers

- **A first step in understanding floating-point numbers is to consider binary numbers having fractional values**

- **What is 1011.101_2?**

- **The binary representation of 11.625**

$$1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-3} =$$

$$8 + 2 + 1 + 1/2 + 1/8 =$$

$$11.625$$

Fractional binary numbers: Examples

- **Value** **Representation**

Value	Representation
5.75 = 5 3/4	`101.11`$_2$
2.875 = 2 7/8	`10.111`$_2$
1.4375 = 1 7/16	`1.0111`$_2$

- **Observations**
  - Divide by 2 by shifting binary point left (unsigned)
  - Multiply by 2 by shifting binary point right
  - Numbers of form $0.111111\ldots_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable numbers

- **Limitation #1**
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations

Value	Representation
1/3	`0.0101010101[01]`…$_2$
1/5	`0.001100110011[0011]`…$_2$
1/10	`0.0001100110011[0011]`…$_2$

- **Limitation #2**
  - Just one setting of binary point within the *w* bits
    - Not efficient for representing very large or very small numbers

Today: Floating Point

- Background: Fractional binary numbers
- **IEEE floating point standard: Definition**
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
- Practice problem

IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware designers in defining standard

Floating point representation

- **Numerical form:**

$$(-1)^s \, M \;\; 2^E$$

  - **Sign bit *s*** determines whether number is negative (*s* = 1) or positive (*s* = 0)
  - **Significand *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two

- **Encoding**
  - MSB **s** is sign bit ***s***
  - **exp** field encodes ***E*** (but is not equal to E)
  - **frac** field encodes ***M*** (but is not equal to M)

s	exp	frac

Precision options

- **Single precision: 32 bits (float type in C)**

s	exp	frac
1	8-bits	23-bits

- **Double precision: 64 bits (double type in C)**

s	exp	frac
1	11-bits	52-bits

- **Extended precision: 80 bits (Intel only)**

s	exp	frac
1	15-bits	63 or 64-bits

Normalized values

$v = (-1)^s M \, 2^E$

- **The common case, when *exp* ≠ 000…0 and *exp* ≠ 111…1**

- **Exponent coded as a *biased* value: *E* = *Exp* – *Bias***
  - *Exp*: unsigned value of *exp* field
  - *Bias* = 2^{k-1} - 1, where *k* is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- **Significand coded with implied leading 1: *M* =** $1.\text{xxx…x}_2$
  - xxx…x: bits of *frac* field
  - Minimum when *frac = 000…0* (M = 1.0)
  - Maximum when *frac = 111…1* (M = 2.0 – ε)
  - Get extra leading bit for "free"

Normalized encoding example

$v = (-1)^s\, M\, 2^E$

$E = Exp - Bias$

- **Value:** `float f = 15213.0;`
  - $15213_{10} = 11101101101101_2$

    $= 1.1101101101101_2 \times 2^{13}$

- **Significand**

  $M = 1.\underline{1101101101101}_2$

  $\mathtt{frac} = \underline{1101101101101}0000000000_2$ (23 bits)

- **Exponent**

  $E = 13$

  $Bias = 127$

  $Exp = 140 = 10001100_2$ (8 bits)

- **Result:**

s	exp	frac
0	10001100	11011011011010000000000

Denormalized values

$v = (-1)^s M\, 2^E$
$E = 1 - Bias$

- **Condition: `exp` = 000…0**
- **Exponent value: $E = 1 - Bias$ (instead of $E = 0 - Bias$)**
- **Significand coded with implied leading 0: $M = 0.xxx\ldots x_2$**
  - **`xxx`…`x`**: bits of **`frac`**
- **Case: `exp` = 000…0, `frac` = 000…0**
  - Represents zero value
  - Note distinct values: +0 and –0 (why?)
- **Case: `exp` = 000…0, `frac` ≠ 000…0**
  - Numbers closest to 0.0
  - Equispaced

Denormalized values

$v = (-1)^s \, M \, 2^E$
$E = 1 - Bias$

- **Condition: *`exp`* = `111…1`**

- **Case: *`exp`* = `111…1`, *`frac`* = `000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: *`exp`* = `111…1`, *`frac`* ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating point encodings

- Normalized

s	≠ 0 & ≠ 255	frac

- Denormalized

s	00000000	frac

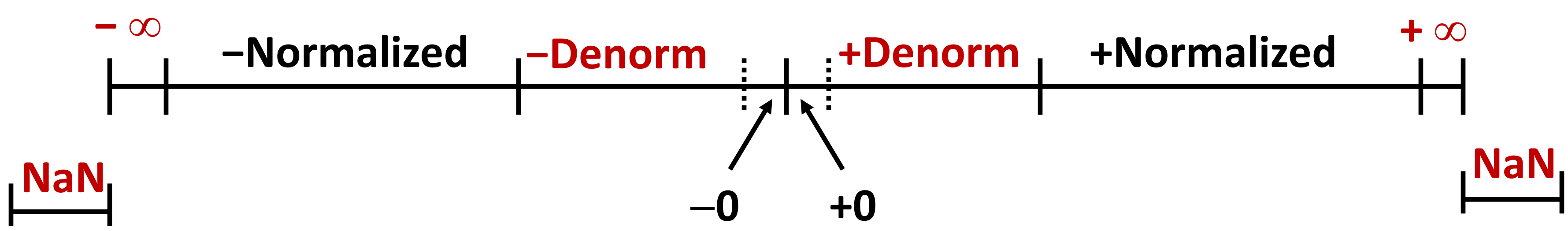
- Infinity

s	11111111	00000000000000000000000

- NaN

s	11111111	≠ 0

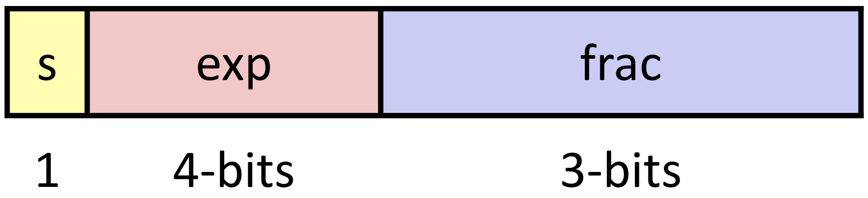
Visualization: Floating point encodings



Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- **Example and properties**
- Rounding, addition, multiplication
- Floating point in C
- Summary
- Practice problem

Tiny floating point example



- **8-bit floating point representation**
  - The sign bit is in the most significant bit
  - The next four bits are the exponent, with a bias of $2^4 - 1 = 7$
  - The last three bits are the **`frac`**

- **Same general form as IEEE Format**
  - Normalized, denormalized
  - Representation of 0, NaN, infinity

Dynamic range (positive only)

$v = (-1)^s M\, 2^E$

n: E = Exp – Bias

d: E = 1 – Bias

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denormalized
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest normalized
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest normalized
	0	1111	000	n/a	inf	

Interesting numbers

{single,double}

Description	*exp*	*frac*	*Numeric Value*
Zero	**00…00**	**00…00**	**0.0**
Smallest positive denorm.	**00…00**	**00…01**	$2^{-\{23,52\}} \times 2^{-\{126,1022\}}$
Single ≈ 1.4×10^{-45}			
Double ≈ 4.9×10^{-324}			
Largest denormalized	**00…00**	**11…11**	$(1.0 - \varepsilon) \times 2^{-\{126,1022\}}$
Single ≈ 1.18×10^{-38}			
Double ≈ 2.2×10^{-308}			
Smallest positive normalized	**00…01**	**00…00**	$1.0 \times 2^{-\{126,1022\}}$
Just larger than largest denormalized			
One	**01…11**	**00…00**	**1.0**
Largest Normalized	**11…10**	**11…11**	$(2.0 - \varepsilon) \times 2^{\{127,1023\}}$
Single ≈ 3.4×10^{38}			
Double ≈ 1.8×10^{308}			

Special properties of the IEEE encoding

- **Floating-point zero same as integer zero**
  - All bits are equal to 0

- **Can (almost) use unsigned integer comparison**
  - Must first compare sign bits
  - Must consider −0 = 0
  - NaNs problematic
    - Will be greater than any other values, including infinity
    - What should comparison yield?
  - Otherwise OK
    - Denormalized vs. normalized
    - Normalized vs. infinity

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- **Rounding, addition, multiplication**
- Floating point in C
- Summary

Floating point operations: Basic idea

- $\mathbf{x} +_f \mathbf{y} = \mathrm{Round}(\mathbf{x} + \mathbf{y})$

- $\mathbf{x} \times_f \mathbf{y} = \mathrm{Round}(\mathbf{x} \times \mathbf{y})$

- **Basic idea**
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into ***frac***

Rounding

- **For a value *x*, find the "closest" matching value *x'* that can be represented in the desired floating-point format**
  - One key problem is to define the direction to round a value that is halfway between two possibilities

- **IEEE floating-point rounding modes**

	1.40	**1.60**	**1.50**	**2.50**	**–1.50**
Towards zero	1	1	1	2	–1
Round down (–∞)	1	1	1	2	–2
Round up (+∞)	2	2	2	3	–1
Round to nearest, ties to even (default)	1	2	2	2	–2

Closer look at round-to-even

- **Default rounding mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of set of positive numbers will consistently be over- or under-estimated

- **Applying to other decimal places / bit positions**
  - When exactly halfway between two possible values
    - Round so that least significant digit is even
  - E.g., round to nearest hundredth

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up)
7.8850000	7.88	(Half way—round down)

Rounding binary numbers

- **Binary fractional numbers**
  - "Even" when least significant bit is **0**
  - "Half way" when bits to right of rounding position = $\mathbf{100}\ldots_2$

- **Examples**
  - Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

Floating point multiplication

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ \times\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- **Exact Result:** $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*: $s1 \wedge s2$
  - Significand *M*: $M1 \times M2$
  - Exponent *E*: $E1 + E2$

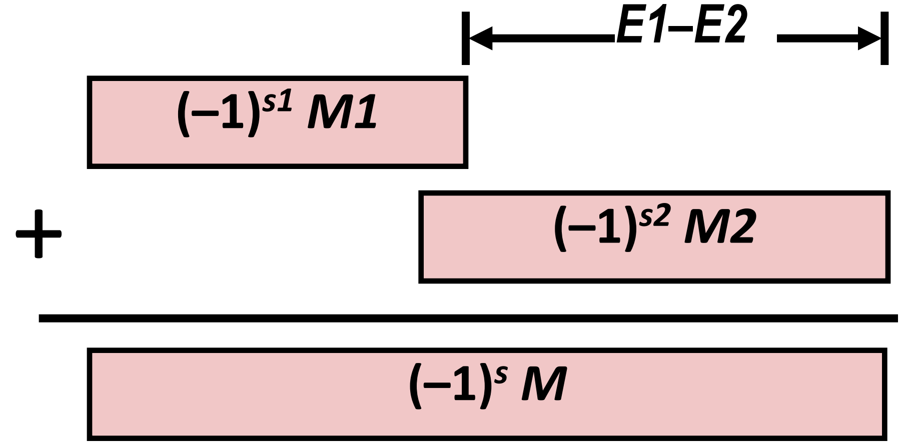
- **Fixing**
  - If $M \geq 2$, normalize - shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit `frac` precision

- **Implementation**
  - Biggest inconvenience is multiplying significands

Floating point addition

- $(-1)^{s1}\, M1\;\; 2^{E1} \;+\; (-1)^{s2}\, M2\;\; 2^{E2}$
  - Assume $E1 > E2$

Get binary points lined up



- **Exact Result:** $(-1)^{s}\, M\;\; 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*

- **Fixing**
  - If $M \geq 2$, normalize - shift *M* right, increment *E*
  - if $M < 1$, normalize - shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit `frac` precision

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- **Floating point in C**
- Summary

Floating Point in C

- **C guarantees two levels**
  - **`float`** single precision
  - **`double`** double precision
- **Conversions/casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`/`float` → `int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int` → `double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int` → `float`**
    - Will round according to rounding mode

Floating Point in C: Exercise

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

- ❑ `x == (int)(float) x`
- ❑ `x == (int)(double) x`
- ❑ `f == (float)(double) f`
- ❑ `d == (double)(float) d`
- ❑ `f == -(-f);`
- ❑ `2/3 == 2/3.0`
- ❑ `d < 0.0` ⇒ `((d*2) < 0.0)`
- ❑ `d > f` ⇒ `-f > -d`
- ❑ `d * d >= 0.0`
- ❑ `(d+f)-d == f`

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- **Summary**
- Practice problem

Summary

- **IEEE Floating Point has clear mathematical properties**
- **Represents numbers of form $(-1)^s\ M\ 2^E$**
- **One can reason about operations independent of implementation**
  - As if computed with perfect precision and then rounded
- **Not the same as real arithmetic**
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary
- **Practice problem**

Practice problem

s	exp	frac
1	4-bits	3-bits

- **Convert 8-bit unsigned numbers to tiny floating point format**

Value	*Binary*
128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Practice problem

s	exp	frac
1	4-bits	3-bits

- **Steps:**

1. **Normalize to have leading 1**
2. **Round to fit within fraction**
3. **Postnormalize to deal with effects of rounding**

Step 1: Normalize

s	exp	frac
1	4-bits	3-bits

- **Requirement**
  - Set binary point so that numbers of form 1.xxxxx
  - Adjust all to have leading one
    - Decrement exponent as shift left

Value	*Binary*	*Fraction*	*Exponent*
`128`	`10000000`	`1.0000000`	`7`
`15`	`00001101`	`1.1010000`	`3`
`17`	`00010001`	`1.0001000`	`4`
`19`	`00010011`	`1.0011000`	`4`
`138`	`10001010`	`1.0001010`	`7`
`63`	`00111111`	`1.1111100`	`5`

Step 2: Rounding

1.BBGRXXX

Guard bit: LSB of result

Round bit: 1st bit removed

Sticky bit:
OR of remaining bits

- **Round up conditions**
  - Round = 1, Sticky = 1 ➙ > 0.5
  - Guard = 1, Round = 1, Sticky = 0 ➙ Round to even

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Step 3: Postnormalize

- **Issue**
  - Rounding may have caused overflow
  - Handle by shifting right once & incrementing exponent

Value	*Rounded*	*Exp*	*Adjusted*	*Result*
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64