

## IEEE Floating Point Arithmetic

### Topics:

- IEEE Floating-Point Representation
  - Floating-Point Addition
  - Other Floating-Point Arithmetic
  - Pipelined Floating-Point Arithmetic
- 
- 

## An Overview of the IEEE Floating-Point Standard 754 (1985)

---

---

### Basic Concepts for Floating-Point Numbers

Motivation:

- Need to represent very small fractions  
e.g., Planck's  $6.6254 * 10^{-27}$
- Need to represent very large numbers  
e.g., Avo's  $6.0247 * 10^{23}$

Format:

- sign
- exponent
- mantissa (significand)

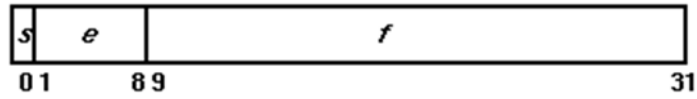
Normalization:

- eliminates leading zeros of mantissa
  - adjusts exponent accordingly
- 

### Floating-Point Representation

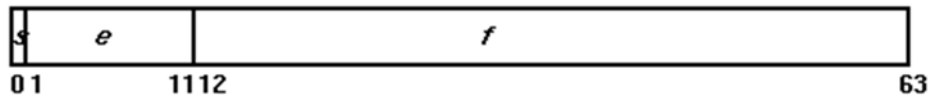
Single-Precision Numbers (*32 bits*)

- 1 bit for sign  
*s (bit 0)*
- 23 bits for magnitude  
*f (bits 9-31)*
- 8 bits for biased exponent  
*e (bits 1-8)*



Double-Precision Numbers (64 bits)

- 1 bit for sign  
 $s$  (bit 0)
- 52 bits for magnitude  
 $f$  (bits 12-63)
- 11 bits for biased exponent  
 $e$  (bits 1-11)




---

## Single-Precision Numbers

Sign ( $s$ )

- FP number is + , if  $s$  is zero
- FP number is - , if  $s$  is one

Fraction ( $f$ ):

- 23-bit binary representation is  $0.f_2$
- True magnitude value =  $0.f_2 + 1.0_2$
- Called a **normalized** representation
- In other words, the magnitude of the value  $1.75_{10}$  would be represented as **11000000000000000000000**,  
not as **11100000000000000000000**,  
because the first **1.0** is assumed to be there

**Biased Exponent ( $e$ )** (*Bias is 127*)

where  $e = p + 127$

such that  $p$  is the true exponent

and  $e$  is the representation of the exponent to the base 2

---

## Example: Single-Precision Numbers

Example

0    0000 1110    1010 0000 0000 0000 0000 000  
s                    e                                    f

Magnitude =  $1.1010\ 0000\ 0000\ 0000\ 0000\ 0000_2$   
or  $1.625$

Exponent =  $-0111\ 0001_2$   
because  $p = e - 127$

$= ((0000\ 1110) - (0111\ 1111))_2$   
(since  $(0111\ 1111)_2 = 127$ )

So  $p = -(0111\ 0001)_2$   
 $= -(64 + 32 + 16 + 1)$   
 $= -113$

Thus, the number represented in binary is

$$(1.1010\ 0000\ 0000\ 0000\ 0000\ 000\ x\ 2^{-0111\ 0001})_2$$

$$\text{or } 1.625 \times 2^{-113}$$

### Exercise: Single-Precision Numbers

Suppose

- $s = 0$  ( $\Rightarrow$  positive)
- $f = 0110\ 0000\ 0000\ 0000\ 0000\ 0000$  ( $= 0$ )
- $e = 1000\ 1111$  ( $= 143$ )

What are the values of the magnitude and exponent?

What is the binary number represented?

What is the decimal number represented?

*Answers:  $1.375 \times 2^{16}$*

### Special Single-Precision Values

When  $e = 0000\ 0000_2$

or  $e = 1111\ 1111_2$ ,

the number is considered to be a special value

+0 as an IEEE single-precision floating-point number

$$= (0\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ 000)_2$$

-0 as an IEEE single-precision floating-point number

$$= (1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ 000)_2$$

+infinity as an IEEE single-precision floating-point number

$$= (0\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ 000)_2$$

-infinity as an IEEE single-precision floating-point number

$$= (1\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\ 000)_2$$

### Range of Single-Precision Numbers

Because of the special cases above,

$$0 < e < 255$$

Thus,  $-127 < p < +128$

$$\text{or } -126 \leq p \leq +127$$

The smallest possible single precision positive number:

$$\begin{aligned} & 1.0000\ 0000\ 0000\ 0000\ 0000\ 000\ x\ 2^{0000\ 0001} \text{ (biased exponent)} \\ = & 1.0000\ 0000\ 0000\ 0000\ 0000\ 000\ x\ 2^{-0111\ 1110} \\ & \text{or } 1.0 \times 2^{-126} \end{aligned}$$

or approximately  $1.2 \times 10^{-38}$

The largest possible single precision positive number:

$$\begin{aligned} & 1.1111\ 1111\ 1111\ 1111\ 1111\ 111\ x\ 2^{1111\ 1110} \text{ (biased exponent)} \\ = & 1.1111\ 1111\ 1111\ 1111\ 1111\ 111\ x\ 2^{0111\ 1111} \\ & \text{or } (2 - 2^{-23}) \times 2^{+127} \\ & \text{or approximately } 3.4 \times 10^{+38} \end{aligned}$$

## Denormalized Numbers

Suppose  $N < 2^{-126}$

- May be set to zero
- May be considered **denormalized**

For denormalized single-precision values,  
the magnitude is in the form  $0.f$

The smallest possible single precision denormalized number:

$2^{-23} \times 2^{-126}$  which is represented as

0 0000 0001 0000 0000 0000 0000 0000 001

This is called **graceful underflow**  
or **gradual underflow**

---

## Numerical Precision

Consider

$x_1 = 1.0000\ 0000\ 0000\ 0000\ 0000\ 000\ 0 \times 2^{-3}$

and

$x_2 = 1.0000\ 0000\ 0000\ 0000\ 0000\ 001\ 0 \times 2^{-3}$

**No value** between these two numbers

can be represented as a single-precision FP number **exactly**

Any value between these two numbers

must be rounded up to  $x_1$  or down to  $x_2$

$x_1$  and  $x_2$  are called **machine numbers**,

because they can be represented **exactly** in the machine

Values between  $x_1$  and  $x_2$  are **not** machine numbers

**Machine epsilon**: distance between  $x_1$  and  $x_2$

---

## Double-Precision Numbers

These are basically the same as single-precision numbers,

but with more bits representing the exponent and the magnitude

Sign ( $s$ )

- Value of number is + , if  $s$  is zero
- Value of number is - , if  $s$  is one

Fraction ( $f$ ):

- 52-bit binary representation is  $0.f_2$
- True magnitude value =  $0.f_2 + 1.0_2$
- Called a **normalized** representation

**Biased Exponent ( $e$ )** (*Bias is 1023*)

where  $e = p + 1023$

such that  $p$  is the true exponent

---

## Range of Double-Precision Numbers

**Exercise**: Provide this range of values

---

## Infinity, NaN, and zero

Since the IEEE standard permits the representation of values like infinity, it must also be able to perform operations on such values

### Infinity

treated like a large number, usually

Examples: Given a valid normalized floating-point number  $Z$ :

- $+infinity + Z \rightarrow +infinity$
  - $+infinity \times Z \rightarrow +infinity$
  - $Z / +infinity \rightarrow +0$
  - $+infinity / Z \rightarrow +infinity$
- $+infinity$  and negative  $Z \Rightarrow$  similar results

### NaN (Not A Number)

used when arithmetic operations produce an indeterminate value

Examples:

- $0 / 0 \rightarrow NaN$
  - $NaN + Z \rightarrow NaN$
  - $+infinity - (+infinity) \rightarrow NaN$
- 0**: a mechanism for recognizing that a number is zero to machine precision

---

## Addition of Floating-Point Numbers

The steps (or stages) of a floating-point addition:

1. The exponents of the two floating-point numbers to be added are compared to find the number with the smallest magnitude
2. The significand of the number with the smaller magnitude is shifted so that the exponents of the two numbers agree
3. The significands are added
4. The result of the addition is normalized
5. Checks are made to see if any floating-point exceptions occurred during the addition, such as overflow
6. Rounding occurs

---

## Floating-Point Addition Example

Example:  $s = x + y$

- numbers to be added are  $x = 1234.00$  and  $y = -567.8$
- these are represented in decimal notation with a mantissa (significand) of four digits
- six stages (A - F) are required to complete the addition

Step	A	B	C	D	E	F
$x$	0.1234E4	0.12340E4				
$y$	-0.05678E3	-0.05678E4				
$s$			0.066620E4	0.6662E3	0.6662E3	0.6662E3

(For this example, we are throwing out biased exponents and the assumed 1.0 before the magnitude. Also all numbers are in the decimal number system and no complements are used.)

---

### Time for Floating-Point Addition

Consider a set of floating-point additions

sequentially following one another

(as in adding the elements of two arrays)

Assume that each stage of the addition takes  $t$  time units

Time:	$t$	$2t$	$3t$	$4t$	$5t$	$6t$	$7t$	$8t$
Step								
A	$x_1 + y_1$						$x_2 + y_2$	
B		$x_1 + y_1$						$x_2 + y_2$
C			$x_1 + y_1$					
D				$x_1 + y_1$				
E					$x_1 + y_1$			
F						$x_1 + y_1$		

Each floating-point addition takes  $6t$  time units

---

### Pipelined Floating-Point Addition

With the proper architectural design,

the floating-point addition stages can be overlapped

Time:	$t$	$2t$	$3t$	$4t$	$5t$	$6t$	$7t$	$8t$
Step								
A	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$	$x_8 + y_8$
B		$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	$x_7 + y_7$
C			$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$
D				$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$
E					$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$
F						$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$

This is called **pipelined** floating-point addition

Once the pipeline is full

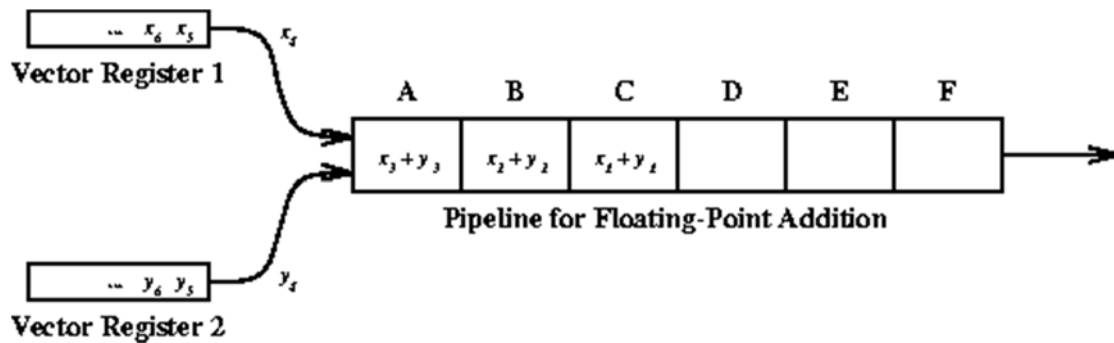
and has produced the first result in  $6t$  time units,

it only takes  $t$  time units to produce each succeeding sum

This is the idea behind vector computers

(Cray-1, Cray-2, Cyber 205)

where a vector can be viewed as a one-dimensional array of floating-point numbers



For an animated version of the figure above,

click [here](#)

For more information on vector computers,

click [here](#)

---

## Main Features of the IEEE Floating-Point Standard 754

**NaN:** Not a Number

- allows computation to continue past exceptions
- if the argument of an expression is **NaN**, then so is the result
- $\Rightarrow$  checking for exceptions and related measures can be done within the computation without having to involve the OS
- Example: suppose  $x = \text{SQRT}(-1)$ 
  - $\text{SQRT}(-1) = \text{NaN}$
  - So  $1 + x$  is NaN
  - and  $1 - x$  is NaN

**Infinity Arithmetic:**

- Division by zero:
  - $1 / 0$  is not NaN
  - $1 / 0$  is infinity
- Division by infinity:
  - $1 / \text{infinity}$  is not NaN
  - $1 / \text{infinity}$  is not infinity
  - $1 / \text{infinity}$  is zero
- All the standard rules for handling infinity apply
- Example: consider  $\arccos x = 2 \arctan (\text{SQRT}((1-x)/(1+x)))$ 
  - $\arctan x$  (as  $x \rightarrow \text{infinity}$ ) =  $\pi/2$
  - So  $\arctan(-1)$ 
    - $= 2 \arctan (\text{SQRT}(2/0))$
    - $= 2 \arctan (\text{SQRT}(\text{infinity}))$
    - $= 2(\pi/2)$
    - $= \pi$

**Rounding Rules:** 4 rounding modes

1. **(Default)** Round to nearest value

-- round to even LSD on ties

2. Round toward zero
3. Round toward + infinity
4. Round toward - infinity

### Representation Issues:

- Mantissas must be unpacked before used  
ie., the implicit bit must be made explicit
- Exponent field range (single-precision):  $1 < e < 254$
- Representation as  $a = s * 1.f * 2^{e-127}$

where

- $1.f$ : significand (mantissa or integer part)
- $f$ : fraction
- $e$ : exponent (field or characteristic)
- $s$ : sign
- Special cases:

$e = 0$	$f = 0$	0 (zero), by convention
$e = 0$	$f \neq 0$	denormalized (subnormal) value
$e = 255$	$f = 0$	+ or = infinity
$e = 255$	$f \neq 0$	NaN

- Example: assume decimal representation with 4 digits  
and let  $x = 1.234 * 10^{E_{min}}$   
(where  $E_{min}$  is the smallest possible exponent)

Then

- $x / 10 = 0.123$
- $x / 100 = 0.012$
- $x / 1000 = 0.001$
- $x / 10000 = 0.000$

This is called **gradual underflow**

Alternative: **flush to zero**

- Note: Denormal values are **not** normalized

Advantage:  $x \neq y \implies x - y \neq 0$

This might not be the case with **flush to zero**

Extra digits (or bits): used for rounding

- **guard digit**: first to the right of mantissa  
(needed for rounding addition results)
- **round digit**: second to the right of mantissa  
(needed for rounding multiplication results)
- **sticky bit**: third to the right of mantissa  
(needed in case of ties, 0.50..00 vs. 0.50..01)

Rounding Mode	Sum $\geq 0$	Sum $< 0$
Toward - Infinity		+1 if (r or s)



Toward + Infinity	+1 if (r or s)	
Toward Zero (truncate)		
Toward Nearest Even	+1 if ( (r and LSB) or (r and s) )	+1 if ( (r and LSB) or (r and s) )

---

Based on notes provided by Carolyn J. C. Schauble, Colorado State University