CSC231
Week 14 — Floating Point Numbers
Fall 2019
Floating Point Numbers
http://cs.smith.edu/dftwiki/index.php/CSC231_An_Introduction_to_Fixed-and_Floating-Point_Numbers

You can read Sections 1.1 and 1.2, and go directly to Section 3
IEEE Floating-Point Number Format
A bit of history…

• 1960s, 1970s: many different ways for computers to represent and process real numbers. Large variation in way real numbers were operated on.

• 1976: Intel starts design of first hardware floating-point co-processor for 8086. Wants to define a standard.

• 1977: Second meeting under umbrella of Institute for Electrical and Electronics Engineers (IEEE). Mostly microprocessor makers (IBM is observer).

• Intel first to put whole math library in a processor.
IBM PC Motherboard

## Intel Coprocessors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8087</td>
<td>1980</td>
<td>Numeric coprocessor for 8086 and 8088 processors.</td>
</tr>
<tr>
<td>80C187</td>
<td>19??</td>
<td>Math coprocessor for 80C186 embedded processors.</td>
</tr>
<tr>
<td>80287</td>
<td>1982</td>
<td>Math coprocessor for 80286 processors.</td>
</tr>
<tr>
<td>80387</td>
<td>1987</td>
<td>Math co-processor for 80386 processors.</td>
</tr>
<tr>
<td>80487</td>
<td>1991</td>
<td>Math co-processor for SX versions of 80486 processors.</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>2012</td>
<td>Multi-core co-processor for Xeon CPUs.</td>
</tr>
</tbody>
</table>
Integrated Coprocessor

(Early) Intel Pentium
Some Processors that do not contain FPUs

- Some ARM processors
- Arduino Uno
- Others
Some Processors that do not contain FPUs

Few people have heard of ARM Holdings, even though sales of devices containing its flavor of chips are projected to be 25 times that of Intel. The chips found in 99 percent of the world’s smartphones and tablets are ARM designs. About 4.3 billion people, 60 percent of the world’s population, touch a device carrying an ARM chip each day.

Ashlee Vance, Bloomberg, Feb 2014

With over 100 billion ARM processors produced as of 2017, ARM is the most widely used processor architecture produced in the largest quantity.

Wikipedia, ARM_architecture
How Much Slower is Library vs FPU operations?


Library-emulated FP operations = **10 to 100 times slower** than hardware FP operations executed by FPU
Floating Point Numbers Are Weird...
<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.45</td>
<td>101.11</td>
</tr>
</tbody>
</table>
Multiplying/Dividing by the Base

In Decimal

1234.56
1234.56 x 10 = 12345.6
12345.6 x 10 = 123456.0

1234.56
1234.56 / 10 = 123.456
123.456 / 10 = 12.3456
## Multiplying/Dividing by the Base

<table>
<thead>
<tr>
<th>In Decimal</th>
<th>In Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1234.56</strong></td>
<td><strong>101.11</strong></td>
</tr>
<tr>
<td>1234.56 ( \times 10 ) = 12345.6</td>
<td>101.11 ( \times 2 ) = 1011.1</td>
</tr>
<tr>
<td>12345.6 ( \times 10 ) = 123456.0</td>
<td>1011.1 ( \times 2 ) = 10111.0</td>
</tr>
<tr>
<td><strong>1234.56</strong></td>
<td><strong>101.11</strong></td>
</tr>
<tr>
<td>1234.56 ( \div 10 ) = 123.456</td>
<td>101.11 ( \div 2 ) = 10.111</td>
</tr>
<tr>
<td>123.456 ( \div 10 ) = 12.3456</td>
<td>10.111 ( \div 2 ) = 1.0111</td>
</tr>
</tbody>
</table>
# Multiplying/Dividing by the Base

<table>
<thead>
<tr>
<th>In Decimal</th>
<th>In Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.56</td>
<td>101.11</td>
</tr>
<tr>
<td>1234.56 x 10 = 12345.6</td>
<td>101.11 x 2 = 10111.0</td>
</tr>
<tr>
<td>12345.6 x 10 = 123456.0</td>
<td>1011.1 x 2 = 101111.0</td>
</tr>
<tr>
<td>1234.56</td>
<td>101.11</td>
</tr>
<tr>
<td>1234.56 / 10 = 123.456</td>
<td>101.11 / 2 = 10.111</td>
</tr>
<tr>
<td>123.456 / 10 = 12.3456</td>
<td>10.111 / 2 = 1.0111</td>
</tr>
</tbody>
</table>

=5.75\_d
=11.50\_d
=23.00\_d
=5.75\_d
=2.875\_d
=1.4375\_d
**Multiplying/Dividing by the Base**

101.11 x 10 = 1011.1

<table>
<thead>
<tr>
<th>In Decimal</th>
<th>In Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.56</td>
<td>101.11</td>
</tr>
<tr>
<td>1234.56 x 10 = 12345.6</td>
<td>101.11 x 2 = 1011.1</td>
</tr>
<tr>
<td>12345.6 x 10 = 123456.0</td>
<td>1011.1 x 2 = 10111.0</td>
</tr>
<tr>
<td>1234.56 / 10 = 123.456</td>
<td>101.11 / 2 = 10.111</td>
</tr>
<tr>
<td>123.456 / 10 = 12.3456</td>
<td>10.111 / 2 = 1.0111</td>
</tr>
</tbody>
</table>

10.111 / 2 = 1.0111

= 5.75d
= 11.50d
= 23.00d
= 5.75d
= 2.875d
= 1.4375d
“0.1 decimal does not exist”

— D.T.
import java.util.*;

public class SomeFloats {
    public static void main(String args[]) {
        float x = 6.02E23f,
             y = -0.000001f,
             z = 1.23456789E-19f,
             t = -1.0f,
             u = 80000000000f;

        System.out.println("nx = " + x
                           + "ny = " + y
                           + "nz = " + z
                           + "nt = " + t
                           + "nu = " + u);
    }
}

```java
import java.util.*;

public class SomeFloats {
    public static void main(String args[]) {
        float x = 6.02E23f,
             y = -0.0000001f,
             z = 1.23456789E-19f,
             t = -1.0f,
             u = 8000000000f;

        System.out.println("nx = " + x
                           + "ny = " + y
                           + "nz = " + z
                           + "nt = " + t
                           + "nu = " + u);
    }
}
```

```
231a@marax ~/handout $ java SomeFloats

x = 6.02E23
y = -1.0E-6
z = 1.2345678E-19
t = -1.0
u = 8.0E9
```
1.230

= 12.30 \times 10^{-1}

= 123.0 \times 10^{-2}

= 0.123 \times 10^{1}
IEEE Format

- 32 bits, single precision (floats in Java)
- 64 bits, double precision (doubles in Java)
- 80 bits*, extended precision (C, C++)

\[ x = +/- 1.bbbbbbb...bbb \times 2^{bbb...bb} \]

* 80 bits in assembly = 1 Tenbyte
10110.01
10110.01

1.011001 \times 2^4
10110.01

1.011001 \times 2^4

1.011001 \times 2^{100}
10110.01

1.011001 \times 2^4

+ 1.011001 \times 2^{100}
\[ 10110.01 \]
\[ 1.011001 \times 2^4 \]
\[ + \quad 1.011001 \times 2^{100} \]

0 011001 100
Observations

\[ x = +/- 1.bbbbbbb....bbb \times 2^{bbb...bb} \]

• +/- is the sign. It is represented by a bit, equal to 0 if the number is positive, 1 if negative.

• the part 1.bbbbbbb....bbb is called the **mantissa**

• the part bbb...bb is called the **exponent**

• 2 is the **base** for the exponent (could be different!)

• the number is **normalized** so that its binary point is moved to the right of the leading 1.

• because the leading bit will always be 1, we don't need to store it. This bit will be an **implied bit, or hidden bit**.
http://www.h-schmidt.net/FloatConverter/IEEE754.html
Normalization (in decimal)
(normal = standard form)

\[ y = 123.456 \]

\[ y = 1.23456 \times 10^2 \]
Normalization
(in binary)

\[ y = 1000.100111 \quad (8.609375d) \]

\[ y = 1.000100111 \times 2^3 \]
Normalization (in binary)

\[ y = 1000.100111 \]

\[ y = 1.000100111 \times 2^3 \]
Normalization (in binary)

\[ y = 1000.100111 \]

\[ y = 1.000100111 \times 2^3 \]

\[ y = 1.000100111 \times 10^{11} \]
$+1.000100111 \times 10^{11}$

- **sign**: 0
- **mantissa**: 1000100111
- **exponent**: 11
But, remember, all* numbers have a leading 1, so, we can pack the bits even more efficiently!

*really?
+ $1.000100111 \times 10^{11}$

- **Sign**: 0
- **Mantissa**: 0001001110
- **Exponent**: 11

**implied bit!**
IEEE Format

31 30 23 22

S Exponent (8) Mantissa (23)

24 bits stored in 23 bits!
\[ y = 1000.100111 \]
\[ y = 1.000100111 \times 2^3 \]
\[ y = 1.000100111 \times 10^{11} \]
\[ y = 1.000100111 \times 10^{11} \]

Why not 00000011 ?
101.110 $\rightarrow$ $1.01110 \times 2^2 = 1.01110 \times 10^{10}$
How is the exponent coded?
### Table: Exponent Bias

<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>-1</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
y = +1.000100111 \times 10^{11}

1.0761719 \times 2^3
= 8.6093752
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
<tr>
<td>real exponent</td>
<td>stored exponent</td>
<td>Comments</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>

130 = 1000 0010

dec bin
\[ y = 1.000100111 \times 10^{11} \]

Real exponent is 3
So we store \( 3 + 127 = 130 \)
\( = 10000010 \) in binary

\[ 1.0761719 \times 2^3 \]
\[ = 8.6093752 \]
Verification

8.6093752 in IEEE FP?

http://www.h-schmidt.net/FloatConverter/IEEE754.html
What About Nasm?

\[
x \text{ dd } 8.6093752
\]

RAM

<table>
<thead>
<tr>
<th>(0x00)</th>
<th>(0x09)</th>
<th>(0x41)</th>
</tr>
</thead>
</table>

\[x\]
Exercises

• How is 1.0 coded as a 32-bit floating point number?
• What about 0.5?
• 1.5?
• -1.5?
• what floating-point value is stored in the 32-bit number below?

1 | 1000 0011 | 111 1000 0000 0000 0000 0000
What about 0.1?
0.1 decimal, in 32-bit precision, IEEE Format:

0 01111011 10011001100110011001101
0.1 decimal, in 32-bit precision, IEEE Format:

0 01111011 10011001100110011001101

Value in double-precision: 0.10000000149011612
NEVER
NEVER
NEVER
COMPARE FLOATS
OR DOUBLES FOR
EQUALITY!
N-E-V-E-R!
for ( double d = 0; d != 0.3; d += 0.1 )
System.out.println( d );
for ( double d = 0; d != 0.3; d += 0.1 )
    System.out.println( d );
<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>-127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>
Zero

- Why is it special?

- \( 0.0 = 0 \ 00000000 \ 000000000000000000000000000000000 \)
### Mantissa Case 0

If the mantissa is 0:

- The number is 0.0

### Table

<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Very Small Numbers

• Smallest numbers have stored exponent of 0.

• In this case, the implied 1 is omitted, and the exponent is -126 (not -127!)
if mantissa is 0: number = 0.0
if mantissa is !0: no hidden 1

<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>
Very Small Numbers

- Example: 0 00000000 00100000000000000000000

\[ \begin{align*} 
0 & \quad 00000000 \quad \text{001000...000} \\
+ & \quad (2^{-126}) \times (0.001) \\
+ & \quad (2^{-126}) \times (0.125) \\
\end{align*} \]

\[ = 1.469 \times 10^{-39} \]
### Special Cases

<table>
<thead>
<tr>
<th>real exponent</th>
<th>stored exponent</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126</td>
<td>0</td>
<td>Special Case #1</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-125</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-124</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-123</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td>Special Case #2</td>
</tr>
</tbody>
</table>

D. Thiebaut, Computer Science, Smith College
Very large exponents

- stored exponent $= 1111\ 1111$

- if the mantissa is $= 0 \Rightarrow +/- \infty$
Very large exponents

- stored exponent = 1111 1111
- if the mantissa is = 0 \rightarrow +/- \infty
Very large exponents

- stored exponent = 1111 1111

- if the mantissa is = 0 \(\rightarrow\) \(+/-\ \infty\)

- if the mantissa is \(!= 0\) \(\rightarrow\) NaN
Very large exponents

- stored exponent = 1111 1111

- if the mantissa is = 0 ➞ +/- ∞

- if the mantissa is != 0 ➞ NaN = Not-a-Number
Very large exponents

- stored exponent = 1111 1111
- if the mantissa is = 0 ==> +/- \( \infty \)
- if the mantissa is != 0 ==> NaN
NaN is sticky!
• $0 \ 11111111 \ 000000000000000000000000 = + \infty$

• $1 \ 11111111 \ 000000000000000000000000 = - \infty$

• $0 \ 11111111 \ 100000100000000000000000 = \text{NaN}$

- The **divisions** $0/0$ and $\pm\infty/\pm\infty$

- The **multiplications** $0\times\pm\infty$ and $\pm\infty\times0$

- The **additions** $\infty + (\infty)$, $(\infty) + \infty$ and equivalent subtractions

- The **square root** of a negative number.

- The **logarithm** of a negative number

- The **inverse sine or cosine** of a number that is less than $-1$ or greater than $+1$
Generating NaNs

```java
import java.util.*;
import static java.lang.Double.NaN;
import static java.lang.Double.POSITIVE_INFINITY;
import static java.lang.Double.NEGATIVE_INFINITY;

public class GenerateNaN {
    public static void main(String args[]) {
        double[] allNaNs = {
            0D / 0D,
            POSITIVE_INFINITY / POSITIVE_INFINITY,
            POSITIVE_INFINITY / NEGATIVE_INFINITY,
            NEGATIVE_INFINITY / POSITIVE_INFINITY,
            NEGATIVE_INFINITY / NEGATIVE_INFINITY,
            0 * POSITIVE_INFINITY,
            0 * NEGATIVE_INFINITY,
            Math.pow(1, POSITIVE_INFINITY),
            POSITIVE_INFINITY + NEGATIVE_INFINITY,
            NEGATIVE_INFINITY + POSITIVE_INFINITY,
            POSITIVE_INFINITY - POSITIVE_INFINITY,
            NEGATIVE_INFINITY - NEGATIVE_INFINITY,
            Math.sqrt(-1),
            Math.log(-1),
            Math.asin(-2),
            Math.acos(+2),
        };
        System.out.println(Arrays.toString(allNaNs));
        System.out.println(NaN == NaN);
        System.out.println(Double.isNaN(NaN));
    }
}
```

[NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN] false true
### Range of Floating-Point Numbers

<table>
<thead>
<tr>
<th></th>
<th>Denormalized</th>
<th>Normalized</th>
<th>Approximate Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>$\pm 2^{-149} \cdot (1-2^{-23}) \times 2^{-126}$</td>
<td>$\pm 2^{-126} \cdot (2-2^{-23}) \times 2^{127}$</td>
<td>$\pm \approx 10^{-44.85}$ to $\approx 10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>$\pm 2^{-1074} \cdot (1-2^{-52}) \times 2^{-1022}$</td>
<td>$\pm 2^{-1022} \cdot (2-2^{-52}) \times 2^{1023}$</td>
<td>$\pm \approx 10^{-323.3}$ to $\approx 10^{308.3}$</td>
</tr>
</tbody>
</table>
## Range of Floating-Point Numbers

Remember that!

<table>
<thead>
<tr>
<th></th>
<th>Denormalized</th>
<th>Normalized</th>
<th>Approximate Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>± $2^{-149}$ to $(1-2^{-23})\times2^{-126}$</td>
<td>± $2^{-126}$ to $(2-2^{-23})\times2^{127}$</td>
<td>± $\sim10^{-44.85}$ to $\sim10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>± $2^{-1074}$ to $(1-2^{-52})\times2^{-1022}$</td>
<td>± $2^{-1022}$ to $(2-2^{-52})\times2^{1023}$</td>
<td>± $\sim10^{-323.3}$ to $\sim10^{308.3}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th><strong>Binary</strong></th>
<th><strong>Decimal</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Precision</strong></td>
<td>$± (2-2^{-23}) \times 2^{127}$</td>
<td>$\sim ± 10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>$± (2-2^{-52}) \times 2^{1023}$</td>
<td>$\sim ± 10^{308.25}$</td>
</tr>
</tbody>
</table>
Resolution of a Floating-Point Format

Check out table here: http://tinyurl.com/FPResol
Resolution:
Another way to look at it

http://jasss.soc.surrey.ac.uk/9/4/4.html
Rosetta Landing on Comet

- 10-year trajectory
Why not using 2’s Complement for the Exponent?

0.000000005 = 0 01100110 1010110101111111110010101
1 = 0 01111111 0000000000000000000000000000000
65536.25 = 0 10001111 00000000000000000000001000000
65536.5 = 0 10001111 00000000000000000000001000000
Does this converter support NaN, and \( \infty \)?

Are there several different representations of \(+\infty\)?

What is the largest float representable with the 32-bit format?

What is the smallest normalized float (i.e. a float which has an implied leading 1. bit)?
How do we add 2 FP numbers?
• \( \text{fp1} = s_1 \ m_1 \ e_1 \)
  \( \text{fp2} = s_2 \ m_2 \ e_2 \)
  \( \text{fp1} + \text{fp2} = ? \)

• \textbf{denormalize} both numbers (restore hidden 1)

• assume \text{fp1} has largest exponent \(e_1\): make \(e_2\) \textbf{equal} to \(e_1\) and \textbf{shift decimal point} in \(m_2\) \(\rightarrow\) \(m_2'\)

• compute \textbf{sum} \(m_1 + m_2'\)

• \textbf{truncate} & \textbf{round} result

• \textbf{renormalize} result (after checking for special cases)
\[ 1.111 \times 2^5 + 1.110 \times 2^8 \]

After expansion:

\[ 1.110000000 \times 2^8 + 0.00111100 \times 2^8 \]

Locate largest number:

Shelf mantissa of smaller:

Compute sum:

Round & truncate:

Normalize:

\[ = 1.11111100 \times 2^8 \]

\[ = 1.11111100 \times 2^8 \]

\[ = 1.11111100 \times 2^8 \]

\[ = 10.000 \times 2^8 \]

\[ = 1.000 \times 2^9 \]
How do we multiply 2 FP numbers?
• \( fp1 = s1 \ m1 \ e1 \)
  \( fp2 = s2 \ m2 \ e2 \)
  \( fp1 \times fp2 = ? \)

• Test for multiplication by special numbers (0, NaN, \( \infty \))

• **denormalize** both numbers (restore hidden 1)

• compute product of \( m1 \times m2 \)

• compute **sum** \( e1 + e2 \)

• **truncate** & **round** \( m1 \times m2 \)

• **adjust** \( e1 + e2 \) and **normalize**.
How do we compare two FP numbers?
Check sign bits first, the compare as unsigned integers! No unpacking necessary!
Programming FP Operations in Assembly…
Pentium

EAX

EBX

ECX

EDX

ALU
Pentium

Cannot do FP computation
FLOATING POINT UNIT

SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
Operation: \((7+10)/9\)
Operation: \((7+10)/9\)

\[\text{fpush } 7\]
Operation: \((7+10)/9\)

- `fpush 7`
- `fpush 10`
Operation: \((7+10)/9\)

fpush 7
fpush 10
fadd
Operation: \((7+10)/9\)

- fpush 7
- fpush 10
- fadd
Operation: \((7+10)/9\)

- fpush 7
- fpush 10
- fadd
- fpush 9
Operation: \((7+10)/9\)

```
fpush 7
fpush 10
fadd
fpush 9
fdiv
```
Operation: \((7+10)/9\)

fpush 7
fpush 10
fadd
fpush 9
fdiv
The Pentium computes FP expressions using RPN!
The Pentium computes FP expressions using RPN!

Reverse Polish Notation
Nasm Example: $z = x + y$

```assembly
SECTION .data

x       dd      1.5
y       dd      2.5
z       dd      0

; compute $z = x + y$
SECTION .text

fld     dword [x]
fld     dword [y]
fadd
fstp    dword [z]
```
#include "stdio.h"

int main() {
    float z = 1.2345e10;
    printf("z = %e\n\n", z);
    return 0;
}
Printing floats in C

```c
#include "stdio.h"

int main() {
    float z = 1.2345e10;
    printf( "z = %e\n\n", z );
    return 0;
}

gcc -m32 -o printFloat printFloat.c
./printFloat
z = 1.234500e+10
```
Printing floats in Assembly?

asm
program

call printf

C stdio.h library (printf)

object file

nasm

gcc

executable
extern printf ; the C function to be called
SECTION .data ; Data section
msg db "sum = %e",0x0a,0x00
x dd 1.5
y dd 2.5
z dd 0
temp dq 0
SECTION .text ; Code section.
global main ; "C" main program
main:
    fld dword [x] ; need to convert 32-bit to 64-bit
    fld dword [y]
    fadd
    fstp dword [z] ; store sum in z
    fld dword [z] ; transform z to 64-bit by pushing in stack
    fstp qword [temp] ; and popping it back as 64-bit quadword
    push dword [temp+4] ; push temp as 2 32-bit words
    push dword [temp]
    push dword msg ; address of format string
    call printf ; Call C function
    add esp, 12 ; pop stack 3*4 bytes
    mov eax, 1 ; exit code, 0=normal
    mov ebx, 0
    int 0x80 ;
dthiebaut@hadoop:~/temp$ nasm -f elf addFloats.asm

dthiebaut@hadoop:~/temp$ gcc -m32 -o addFloats addFloats.o

dthiebaut@hadoop:~/temp$ ./addFloats

sum = 4.000000e+00

dthiebaut@hadoop:~/temp$
More code examples here:

http://cs.smith.edu/dftwiki/index.php/
CSC231_An_Introduction_to_Fixed-__and_Floating-Point_Numbers#Assembly_Language_Programs