Fixed-Point & Floating-Point Number Formats

CSC231—Assembly Language Week #14

Dominique Thiébaut
dthiebaut@smith.edu
Reference

http://cs.smith.edu/dftwiki/index.php/CSC231_An_Introduction_to_Fixed-_and_Floating-Point_Numbers
public static void main(String[] args) {

    int n = 10;
    int k = -20;

    float x = 1.50;
    double y = 6.02e23;

}
```java
public static void main(String[] args) {
    int n = 10;
    int k = -20;
    float x = 1.50;
    double y = 6.02e23;
}
```
public static void main(String[] args) {
    int n = 10;
    int k = -20;
    float x = 1.50;
    double y = 6.02e23;
}

Nasm knows what 1.5 is!
section .data

x dd 1.5

in memory, x is represented by

00111111 11000000 00000000 00000000
or 0x3FC00000

Nasm knows what 1.5 is!
• **Fixed-Point Format**

• **Floating-Point Format**
Fixed-Point Format

• Used in very few applications, but programmers know about it.

• Some micro controllers (e.g. Arduino Uno) do not have Floating Point Units (FPU), and must rely on libraries to perform Floating Point operations (VERY SLOW)

• Can be used when storage is at a premium (can use small quantity of bits to represent a real number)
Review Decimal System

\[ 123.45 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} \]
Can we do the same in binary?

- Let's do it with **unsigned numbers** first:

  \[1101.11 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}\]

  Binary Point
Can we do the same in binary?

• Let's do it with **unsigned numbers** first:

\[ 1101.11 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \]

\[ = 8 \quad + 4 \quad + 1 \quad + 0.5 \quad + 0.25 \]

\[ = 13.75 \]
• If we know where the binary point is, we do not need to "store" it anywhere. (Remember we used a bit to represent the +/- sign in 2’s complement.)

• A format where the binary/decimal point is fixed between 2 groups of bits is called a **fixed-point format**.
Definition

• A number format where the numbers are unsigned and where we have \(a\) integer bits (on the left of the decimal point) and \(b\) fractional bits (on the right of the decimal point) is referred to as a \(U(a,b)\) fixed-point format.

• Value of an \(N\)-bit binary number in \(U(a,b)\):

\[
x = \left(\frac{1}{2^b}\right) \sum_{n=0}^{N-1} 2^n x_n
\]
Exercise 1

\[ x = 1011\ 1111 = 0xBF \]

- What is the value represented by \( x \) in \( U(4,4) \)?
- What is the value represented by \( x \) in \( U(7,3) \)?
Exercise 2

• \( z = 00000001 \ 00000000 \)

• \( y = 00000010 \ 00000000 \)

• \( v = 00000010 \ 10000000 \)

• What values do \( z \), \( y \), and \( v \) represent in a \( U(8,8) \) format?
Exercise 3

• What is 12.25 in $U(4,4)$? In $U(8,8)$?
What about **Signed** Numbers?
Observation #1

• In an N-bit, **unsigned** integer format, the weight of the MSB is $2^{N-1}$
<table>
<thead>
<tr>
<th>nybble</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>+8</td>
</tr>
<tr>
<td>1001</td>
<td>+9</td>
</tr>
<tr>
<td>1010</td>
<td>+10</td>
</tr>
<tr>
<td>1011</td>
<td>+11</td>
</tr>
<tr>
<td>1100</td>
<td>+12</td>
</tr>
<tr>
<td>1101</td>
<td>+13</td>
</tr>
<tr>
<td>1110</td>
<td>+14</td>
</tr>
<tr>
<td>1111</td>
<td>+15</td>
</tr>
</tbody>
</table>

\[ N = 4 \]
\[ 2^{N-1} = 2^3 = 8 \]
Observation #2

- In an N-bit signed 2's complement, integer format, the weight of the MSB is $-2^{N-1}$
<table>
<thead>
<tr>
<th>nybble</th>
<th>2's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+0</td>
</tr>
<tr>
<td>0001</td>
<td>+1</td>
</tr>
<tr>
<td>0010</td>
<td>+2</td>
</tr>
<tr>
<td>0011</td>
<td>+3</td>
</tr>
<tr>
<td>0100</td>
<td>+4</td>
</tr>
<tr>
<td>0101</td>
<td>+5</td>
</tr>
<tr>
<td>0110</td>
<td>+6</td>
</tr>
<tr>
<td>0111</td>
<td>+7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

\[ N=4 \]
\[ -2^{N-1} = -2^3 = -8 \]
Fixed-Point Signed Format

- **Fixed-Point signed** format = sign bit + $a$ integer bits + $b$ fractional bits = $N$ bits = $A(a, b)$

- $N =$ number of bits = $1 + a + b$

- Format of an $N$-bit $A(a, b)$ number:

\[
x = (1/2^b) \left[ -2^{N-1} x_{N-1} + \sum_{0}^{N-2} 2^n x_n \right],
\]
Examples in A(7,8)

- 000000001 00000000 = 00000001 . 00000000 = ?
- 100000001 00000000 = 10000001 . 00000000 = ?
- 00000010 00000000 = 0000010 . 00000000 = ?
- 10000010 00000000 = 1000010 . 00000000 = ?
- 00000010 10000000 = 00000010 . 10000000 = ?
- 10000010 10000000 = 10000010 . 10000000 = ?
Examples in A(7,8)

- $00000001 \ 00000000 = 00000001 \cdot 00000000 = 1d$
- $10000001 \ 00000000 = 10000001 \cdot 00000000 = -128 + 1 = -127d$
- $00000010 \ 00000000 = 0000010 \cdot 00000000 = 2d$
- $10000010 \ 00000000 = 1000010 \cdot 00000000 = -128 + 2 = -126d$
- $00000010 \ 10000000 = 0000010 \cdot 10000000 = 2.5d$
- $10000010 \ 10000000 = 1000010 \cdot 10000000 = -128 + 2.5 = -125.5d$
Exercises

• What is -1 in $A(7,8)$?
• What is -1 in $A(3,4)$?
• What is 0 in $A(7,8)$?
• What is the smallest number one can represent in $A(7,8)$?
• The largest in $A(7,8)$?
Exercises

• What is the largest number representable in $U(a, b)$?

• What is the smallest number representable in $U(a, b)$?

• What is the largest positive number representable in $A(a, b)$?

• What is the smallest negative number representable in $A(a, b)$?
We Stopped Here Last Time...
• **Fixed-Point Format**
  
  • **Definitions**
    
    • Range
    
    • Precision
    
    • Accuracy
    
    • Resolution
  
  • **Floating-Point Format**
Range

- Range = difference between most positive and most negative numbers.

- **Unsigned Range:**
  The range of $U(a, b)$ is $0 \leq x \leq 2^a - 2^{-b}$

- **Signed Range:**
  The range of $A(a, b)$ is $-2^a \leq x \leq 2^a - 2^{-b}$
2 different definitions

- **Precision** = $b$, the number of fractional bits
  
  [https://en.wikibooks.org/wiki/Floating_Point/Fixed-Point_Numbers](https://en.wikibooks.org/wiki/Floating_Point/Fixed-Point_Numbers)

- **Precision** = $N$, the total number of bits
  
Resolution

• The **resolution** is the smallest non-zero magnitude representable.

• The **resolution** is the size of the intervals between numbers represented by the format

• Example: \( A(13, 2) \) has a resolution of 0.25.
A(13, 2) → sbbbb bbbbb bbbbb bb . bb

\[ 2^{-2} = 0.25 \]

\[ 2^{-1} = 0.5 \]
\[ A(13, 2) \rightarrow \text{sbbbb bbbbb bbbbb bb . bb} \]

Resolution

\[ 2^{-2} = 0.25 \]
\[ 2^{-1} = 0.5 \]
Accuracy

• The **accuracy** is the largest magnitude of the difference between a number and its representation.

• **Accuracy** = $\frac{1}{2}$ **Resolution**
A(13, 2) $\rightarrow$ sbbbb bbbbb bbbbb bb . bb

Real quantity we want to represent

$2^{-2} = 0.25$

$2^{-1} = 0.5$
$A(13, 2) \rightarrow \text{sbbb bbbb bbbb bb . bb}$

$2^{-2} = 0.25$

$2^{-1} = 0.5$

Real quantity we want to represent

Representation
A(13, 2) → sbbbb bbbbb bbbbb bb . bb

\[
\begin{align*}
2^{-2} &= 0.25 \\
2^{-1} &= 0.5
\end{align*}
\]

Error

-0.5  -0.25  0  0.25  0.5
A(13, 2) $\rightarrow$ sbbb bbbb bbbb bb . bb

Largest Error = Accuracy
Questions in search of answers…

- What is the accuracy of an U(7,8) number format?

- How good is U(7,8) at representing small numbers versus representing larger numbers? In other words, *is the format treating small numbers better than large numbers, or the opposite?*
Questions in search of answers…

U(7,8) resolution = $2^{-8} = 0.00390625$

accuracy = $2^{-9} = 0.001953125$

• What is the accuracy of an U(7,8) number format?

• How good is U(7,8) at representing small numbers versus representing larger numbers? In other words, is the format treating small numbers better than large numbers, or the opposite?
• Fixed-Point Format

• Floating-Point Format
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
## 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
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…
“0.1 decimal does not exist”

— D.T.
```java
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 = 8000000000f;

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

```
231b@aurora ~/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++)

\[ x = +/- 1.bbbbbbbb....bbb \times 2^{b\ldots b} \]

* 80 bits in assembly = 1 Tenbyte
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

+ 1.011001 \times 2^{100}

0 011001 100
10110.01

0 011001 100
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 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><strong>1234.56</strong></td>
<td><strong>101.11</strong></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>
## Multiplying/Dividing by the Base

### In Decimal

<table>
<thead>
<tr>
<th>Number</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234.56</td>
<td>$\times 10$</td>
<td>12345.6</td>
</tr>
<tr>
<td>12345.6</td>
<td>$\times 10$</td>
<td>123456.0</td>
</tr>
<tr>
<td>1234.56</td>
<td>$\div 10$</td>
<td>123.456</td>
</tr>
<tr>
<td>123.456</td>
<td>$\div 10$</td>
<td>12.3456</td>
</tr>
</tbody>
</table>

### In Binary

<table>
<thead>
<tr>
<th>Number</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>101.11</td>
<td>$\times 2$</td>
<td>1011.1</td>
</tr>
<tr>
<td>1011.1</td>
<td>$\times 2$</td>
<td>10111.0</td>
</tr>
<tr>
<td>101.11</td>
<td>$\div 2$</td>
<td>10.111</td>
</tr>
<tr>
<td>10.111</td>
<td>$\div 2$</td>
<td>1.0111</td>
</tr>
</tbody>
</table>

---

$=5.75d$  
$=11.50d$  
$=23.00d$  
$=5.75d$  
$=2.875d$  
$=1.4375d$
Multiplying/Dividing by the Base

In Decimal

\[
\begin{align*}
1234.56 & \times 10 = 12345.6 \\
1234.56 & \div 10 = 123.456 \\
1234.56 & \div 10 = 123.4560
\end{align*}
\]

In Binary

\[
\begin{align*}
101.11 & \times 2 = 1011.1 \\
101.11 & \div 2 = 10.111 \\
101.11 & \div 2 = 1.0111
\end{align*}
\]
Observations

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

- +/- is the sign. It is represented by a bit, equal to 0 if the number is positive, 1 if negative.
- the part 1.\overline{bbb} is called the mantissa
- the part \overline{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.
http://www.h-schmidt.net/FloatConverter/IEEE754.html
for ( double d = 0; d != 0.3; d += 0.1 )
    System.out.println( d );
Normalization (in decimal)

(normal = standard form)

\[ y = 123.456 \]

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

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

\[ y = 1.000100111 \times 2^3 \]
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!
implied bit!

\[ +1.000100111 \times 10^{11} \]

- **sign**: 0
- **mantissa**: 0001001110
- **exponent**: 11
IEEE Format

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 $00000011$?
How is the exponent coded?
<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>
<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>
\[ y = 1.000100111 \times 10^{11} \]

Ah! 3 represented by 130 = 128 + 2

\[ 1.0761719 \times 2^3 = 8.6093752 \]
Verification
8.6093752 in IEEE FP?

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

\[
\begin{array}{c|c|c}
1 & 1000 0011 & 111 1000 0000 0000 0000 0000 0000 \\
\end{array}
\]
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
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 );
### 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><strong>Special Case #1</strong></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><strong>Special Case #2</strong></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td></td>
</tr>
</tbody>
</table>
Zero

• Why is it special?

• 0.0 = 0 00000000 00000000000000000000000000000000
### 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>-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>
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!)
### Table: Signed-magnitude, 1-bit biased exponent

<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><strong>Special Case #1</strong></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>-1</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>-1</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>254</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>255</td>
<td><strong>Special Case #2</strong></td>
</tr>
</tbody>
</table>

- If mantissa is 0:  
  - number = 0.0
- If mantissa is !0:  
  - no hidden 1
Very Small Numbers

• Example: 0 00000000 00100000000000000000000

\[0\ 00000000\ 001000\ldots000\]
\[+ (2^{-126}) \times (0.001)\]
\[+ (2^{-126}) \times (0.125)\]
\[= 1.469 \times 10^{-39}\]
<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>.</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 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 ➔ +/- ∞

• if the mantissa is != 0 ➔ 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 00000000000000000000000000000000 = + ∞

• 1 11111111 00000000000000000000000000000000 = - ∞

• 0 11111111 10000010000000000000000000000000 = NaN
Operations that create NaNs (http://en.wikipedia.org/wiki/NaN):

- The **divisions** $0/0$ and $\pm \infty/\pm \infty$
- The **multiplications** $0 \times \pm \infty$ and $\pm \infty \times 0$
- 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 $+1$
public class GenerateNaN {
    public static void main(String args[]) {
        double[] allNaNs = {
            0D / 0D,  // NaN
            POSITIVE_INFINITY / POSITIVE_INFINITY,  // NaN
            POSITIVE_INFINITY / NEGATIVE_INFINITY,  // NaN
            NEGATIVE_INFINITY / POSITIVE_INFINITY,  // NaN
            NEGATIVE_INFINITY / NEGATIVE_INFINITY,  // NaN
            0 * POSITIVE_INFINITY,  // NaN
            0 * NEGATIVE_INFINITY,  // NaN
            Math.pow(1, POSITIVE_INFINITY),  // NaN
            POSITIVE_INFINITY + NEGATIVE_INFINITY,  // NaN
            NEGATIVE_INFINITY + POSITIVE_INFINITY,  // NaN
            POSITIVE_INFINITY - POSITIVE_INFINITY,  // NaN
            NEGATIVE_INFINITY - NEGATIVE_INFINITY,  // NaN
            Math.sqrt(-1),  // NaN
            Math.log(-1),  // NaN
            Math.asin(-2),  // NaN
            Math.acos(+2)  // NaN
        };
        System.out.println(Arrays.toString(allNaNs));
        System.out.println(Double.isNaN(NaN));
        System.out.println(NaN == NaN);
    }
}
## 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>± $2^{-149}$ to $(1-2^{-23}) \times 2^{-126}$</td>
<td>± $2^{-126}$ to $(2-2^{-23}) \times 2^{127}$</td>
<td>± $\sim 10^{-44.85}$ to $\sim 10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>± $2^{-1074}$ to $(1-2^{-52}) \times 2^{-1022}$</td>
<td>± $2^{-1022}$ to $(2-2^{-52}) \times 2^{1023}$</td>
<td>± $\sim 10^{-323.3}$ to $\sim 10^{308.3}$</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 ($1-2^{-23}) \times 2^{-126}$</td>
<td>± $2^{-126}$ to ($2-2^{-23}) \times 2^{127}$</td>
<td>± $\sim10^{-44.85}$ to $\sim10^{38.53}$</td>
</tr>
<tr>
<td><strong>Double Precision</strong></td>
<td>± $2^{-1074}$ to ($1-2^{-52}) \times 2^{-1022}$</td>
<td>± $2^{-1022}$ to ($2-2^{-52}) \times 2^{1023}$</td>
<td>± $\sim10^{-323.3}$ to $\sim10^{308.3}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Binary</th>
<th>Decimal</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
What does it have to do with Art?
We Stopped Here
Last Time...
- Rosetta Landing on Comet
- 10-year trajectory
Why not using 2’s Complement for the Exponent?

0.000000005 = 0 01100110 1010110101111111110010101
1 = 0 01111111 00000000000000000000000
65536.5 = 0 10001111 000000000000000000001000000
65536.25 = 0 10001111 00000000000000000000100000
Does this converter support NaN, and ∞?

Are there several different representations of +∞?

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?
• $fp_1 = s_1 \ m_1 \ e_1$
  $fp_2 = s_2 \ m_2 \ e_2$
  $fp_1 + fp_2 = ?$

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

• assume $fp_1$ has largest exponent $e_1$: make $e_2$ **equal** to $e_1$ and **shift decimal point** in $m_2 \rightarrow m_2'$

• compute **sum** $m_1 + m_2'$

• **truncate & round** result

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

\[
\begin{align*}
1.110000000 \times 2^8 \\
+ 0.001111000 \times 2^8 \\
\hline
1.111111000 \times 2^8 \\
\downarrow 1.1111100 \times 2^8 \\
\hline
= 10.000 \times 2^8 \\
= 1.000 \times 2^9
\end{align*}
\]

- **after expansion**
- **locates largest number**
- **shifts mantissa of smaller**
- **computes sum**
- **rounds & truncates**
- **normalizes**
How do we multiply 2 FP numbers?
• \( fp_1 = s_1 \ m_1 \ e_1 \)
  \( fp_2 = s_2 \ m_2 \ e_2 \)
  \( fp_1 \times fp_2 = ? \)

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

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

• compute product of \( m_1 \times m_2 \)

• compute **sum** \( e_1 + e_2 \)

• **truncate** & **round** \( m_1 \times m_2 \)

• **adjust** \( e_1 + e_2 \) 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

Cannot do FP computation
Operation: \( (7+10)/9 \)
Operation: \((7 + 10) / 9\)

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

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

- \(\text{fpush} \ 7\)
- \(\text{fpush} \ 10\)
- \(\text{fadd}\)
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]
```