Week 14
Fixed & Floating Point Formats

CSC231—Fall 2017
Week #14

Dominique Thiébaut
dthiebaut@smith.edu
• In C, strings are terminated by a byte containing 0 decimal, or 0000 0000 binary. In C, we express this quantity as '\0'.

• In assembly, 0 as a byte is expressed as 0

• '\0' in C = 0000 0000 = 0

• '0' in assembly = 0011 0000 = 0x30

Cmsg       db    "hello", 0
            cmp   al, 0
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;

}
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
Outline

• 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)
- Fixed-Point can be used when storage is at a premium (can use small quantity of bits to represent a real number)
Review Decimal Real Numbers

123.45 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2}

Decimal Point
Can we do the same in binary?

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

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

**Binary Point**
Can we do the same in binary?

- Let's do it with \textbf{unsigned numbers} first:

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

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

\[= 13.75\]
• 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**.
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

What values do $z$, $y$, and $v$ represent in a $U(8,8)$ format?

- $z = 00000001 \ 00000000$
- $y = 00000010 \ 00000000$
- $v = 00000010 \ 10000000$
Exercise 3

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

• In an N-bit, **unsigned** integer format, the weight of the MSB is $2^{N-1}$
### nybble vs Unsigned

<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:

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

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

- 00000001 00000000 = 00000001 . 00000000 = 1d
- 10000001 00000000 = 10000001 . 00000000 = -128 + 1 = -127d
- 00000010 00000000 = 0000010 . 00000000 = 2d
- 10000010 00000000 = 1000010 . 00000000 = -128 + 2 = -126d
- 00000010 10000000 = 00000010 . 10000000 = 2.5d
- 10000010 10000000 = 1000010 . 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 -1 in $A(7,8)$?
  11111111 00000000

• What is -1 in $A(3,4)$?
  1111 0000

• What is 0 in $A(7,8)$?
  00000000 00000000

• What is the smallest number one can represent in $A(7,8)$?
  10000000 00000000

• The largest in $A(7,8)$?
  01111111 11111111
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 $A(a, b)$?
Exercises

• What is the largest number representable in $U(a, b)$?
  \[1111\ldots1\  111\ldots1 = 2^a - 2^{-b}\]

• What is the smallest number representable in $U(a, b)$?
  \[0000\ldots0\  000\ldots01 = 2^{-b}\]

• What is the largest positive number representable in $A(a, b)$?
  \[0111\ldots11\  111\ldots11 = 2^{a-1} - 2^b\]

• What is the smallest negative number representable in $A(a, b)$?
  \[1000\ldots00\  000\ldots000 = 2^{a-1}\]
• Fixed-Point Format
  • Definitions
    • Range
    • Precision
    • Accuracy
    • Resolution
• Floating-Point Format
• 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 \( -2^a \leq x \leq 2^a - 2^{-b} \)
Precision

• **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) $\rightarrow$ sbbbb bbbbb bbbbb bb . bb

$2^{-2} = 0.25$

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

Real quantity we want to represent

$2^{-2} = 0.25$

$2^{-1} = 0.5$
A(13, 2) —> sbbb bbbb bbbb bb . bb

$2^{-2} = 0.25$

$2^{-1} = 0.5$

Real quantity we want to represent
A(13, 2) → sbb...bb . bb
\[ A(13, 2) \rightarrow \text{sbbb bbbb bbbb bb . bb} \]

Largest Error = \text{Accuracy}

\[ 2^{-2} = 0.25 \]
\[ 2^{-1} = 0.5 \]
We stopped here last time…
A look at Homework 8
section .data
s1 db  "Hello world! 123 <=#$", 10, 0
s1Len equ $-s1

section .text
global _start
extern _printString, _println

_start:
mov ebx, s1 ; ebx points to start of s1
for:
cmp byte[ebx], 0 ; char at ebx == 0?
je done ; if so, we're done
cmp byte[ebx], 'a' ; char at ebx lower than 'a'
jb next ; if so, not interested
cmp byte[ebx], 'z' ; char at ebx higher than 'z'
ja next ; if so, not interested
add byte[ebx], -'a'+'A' ; else, it's a lowercase, make it
next:
inc ebx ; point to next char
jmp for ; loop back
done:
mov ecx, s1 ; we're done, print the string
mov edx, s1Len-1
call _printString
call _println

getcopy hw8_f1.asm
;;; f1: takes string address passed in stack at ebp+8
;;; and transforms it into its uppercase equivalent.
;;; f1 assumes string is terminated by 0

f1:
    push    ebp
    mov     ebp, esp

    push    ebx                ; save ebx

    mov     ebx, s1
    mov     ebx, dword[ebp+8]

    .for:   cmp      byte[ebx], 0   ; char at ebx == 0?
            je       .done        ; if so, we're done
            cmp      byte[ebx], 'a'
            jb       .next         ; if so, not interested
            cmp      byte[ebx], 'z'
            ja       .next         ; if so, not interested
            add      byte[ebx], '-a'+'A'     ; else, it's a lowercase, make it
            inc      ebx            ; uppercase
            jmp      .for          ; point to next char
            .next:
    .done:   pop       ebx
    pop       ebp
    ret       4

getcopy hw8_f1b.asm
;;; hw8 f3 solution
;;; D. Thiebaut

section .data
array dd 3, 5, 0, 1, 2, 10, 100, 4, 1
arrayLen equ ($-array)/4 ; figure out the /4 part.

section .text
global _start
extern _printInt
extern _println

_start:
  mov ebx, array
  mov ecx, arrayLen
  mov eax, 0 ; set counter of even numbers
               ; to 0

  for:     mov edx,dword[ebx] ; get int at ebx in edx
           and edx, 1 ; test last bit of edx for parity
           jnz next ; if 1, then odd, skip increment
           inc eax ; if 0, then even, increment counter
    next:    add ebx, 4 ; ebx points to next int
               ; keep looping...
    loop     for
  call _printInt ; print # of even ints found
  call _println

-UU-:**:--F1 hw8_f3.asm Top L14 (Assembler)
f3: gets array and array length in stack. 
counts the number of even ints in array and returns 
it in eax.

```
push   ebp
mov    ebp, esp ;set up stack frame

push   ebx ;save regs used (but not eax)
push   ecx
push   edx

mov    ebx, dword[ebp+12]
mov    ecx, dword[ebp+8]
mov    eax, 0 ;set counter of even numbers ; to 0

.for:   mov    edx,dword[ebx] ;get int at ebx in edx
        and    edx, 1 ;test last bit of edx for parity
        jnz    .next ;if 1, then odd, skip increment
        inc    eax ;if 0, then even, increment counter

.next:  add    ebx, 4 ;ebx points to next int
        loop   .for ;keep looping...

pop     edx ;restore regs saved
pop     ecx
pop     edx
pop     ebp ;restore old stack frame
ret     2*4 ;return and pop 2 dwords
```
Documentation is IMPORTANT!
A word about Hw7a
• Fixed-Point Format

• Floating-Point Format
Exercise

- 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>

Pentium
March 1993
Integrated Coprocessor
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

- Some ARM processors
- Arduino Uno
- Others
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 = 80000000000f;

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

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

* 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

\begin{array}{c}
\begin{array}{c}
+ \\
1.011001 \times 2^{100}
\end{array}
\end{array}

\begin{array}{c}
\begin{array}{c}
0 \\
011001 \\
100
\end{array}
\end{array}
Multiplying/Dividing by the Base

\[
\begin{align*}
1234.56 \times 10 &= 12345.6 \\
1234.56 \times 10 &= 123456.0
\end{align*}
\]

\[
\begin{align*}
1234.56 \div 10 &= 123.456 \\
123.456 \div 10 &= 12.3456
\end{align*}
\]
## 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>$/ 10$</td>
<td>123.456</td>
</tr>
<tr>
<td>123.456</td>
<td>$/ 10$</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>$/ 2$</td>
<td>10.111</td>
</tr>
<tr>
<td>10.111</td>
<td>$/ 2$</td>
<td>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 = 10111.1</td>
</tr>
<tr>
<td>12345.6 x 10 = 123456.0</td>
<td>=1011.1 x 2 = 101110.0</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<sub>d</sub>

=11.50<sub>d</sub>

=23.00<sub>d</sub>

=5.75<sub>d</sub>

=2.875<sub>d</sub>

=1.4375<sub>d</sub>
### 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 = 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>

=5.75\text{d} 
=11.50\text{d} 
=23.00\text{d} 
=5.75\text{d} 
=2.875\text{d} 
=1.4375\text{d}
Observations

\[ x = +/- \ 1.\underbrace{b\ b\ b\ b\ b\ b}_{\text{mantissa}} \times 2^{\underbrace{b\ b\ b\ \ldots\ b}_{\text{exponent}}} \]

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

- the part \(1.\underbrace{b\ b\ b\ b\ b\ b}_{\text{mantissa}}\) is called the **mantissa**

- the part \(\underbrace{b\ b\ b\ \ldots\ b}_{\text{exponent}}\) 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**
### IEEE 754 Converter

This page allows you to convert between the decimal representation of numbers (like "1.02") and the binary format used by all modern CPUs (IEEE 754 floating point). The conversion is limited to single precision numbers (32 Bit). The purpose of this webpage is to help you understand floating point numbers.

**IEEE 754 Converter (JavaScript), V0.12**

Note: This JavaScript-based version is still under development, please report errors [here](http://www.h-schmidt.net/FloatConverter/IEEE754.html).

<table>
<thead>
<tr>
<th>Value</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>2⁻⁴</td>
<td>1.60000023841858</td>
</tr>
<tr>
<td></td>
<td>123</td>
<td>5033165</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 0 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1</td>
</tr>
</tbody>
</table>

- **Decimal Representation**: 0.1
- **Binary Representation**: 00111101110011001100110011001101
- **Hexadecimal Representation**: 0x3dcccccd
- **After casting to double precision**: 0.10000000149011612