

CSE 31

Computer Organization

Lecture 18 – Floating Point Numbers (2)

Announcement

- ▶ Project #2
 - Start working on it during lab this week
 - Due Monday (4/29)
- ▶ HW #6 in CatCourses
 - Due Monday (4/22) at 11:59pm
- ▶ Reading assignment
 - Chapter 1.6, 6.1-6.3 of zyBooks
 - Make sure to do the Participation Activities
 - Due Monday (4/15) at 11:59pm
 - Chapter 6.4-6.7 of zyBooks
 - Make sure to do the Participation Activities
 - Due Friday (4/19) at 11:59pm

Announcement

- ▶ Midterm Exam 2
 - 4/24 (Wednesday, in lecture) Not 4/17 as scheduled
 - Lectures #8 - #18
 - HW #2 - #6
 - Practice exam in CatCourses
 - Closed book
 - 1 sheet of note (8.5" x 11")
 - MIPS reference sheet will be provided

Pseudocode

- How many of you write pseudocode before coding?
- Informal, English-like (or any written languages)
- Describes *how* an algorithm, a routine, a class, or a program *will work*
- Avoid **syntactic elements**

A Bad Example

```
increment resource number by 1  
allocate a dlg struct using malloc  
if malloc() returns NULL then return 1  
invoke OSsrc_init to initialize a resource for the OS  
*hRsrcPtr = resource number  
return 0
```

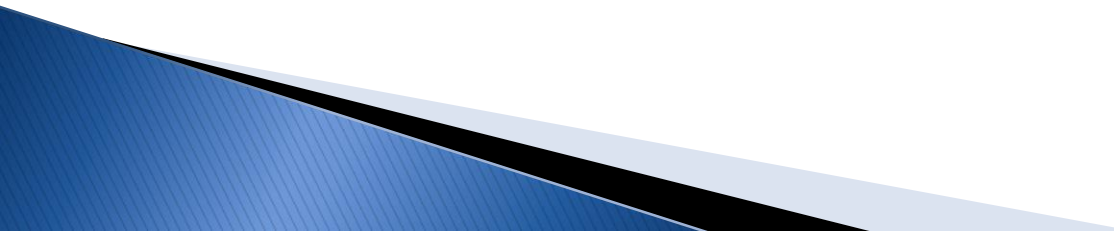
Why?

Better Pseudocode

```
Keep track of current number of resources in use
If another resource is available
    Allocate a dialog box structure
    If a dialog box structure could be allocated
        Note that one more resource is in use
        Initialize the resource
        Store the resource number at the location provided
        by the caller
    endif
endif
Return true if a new source was created; else return false
```

With structure

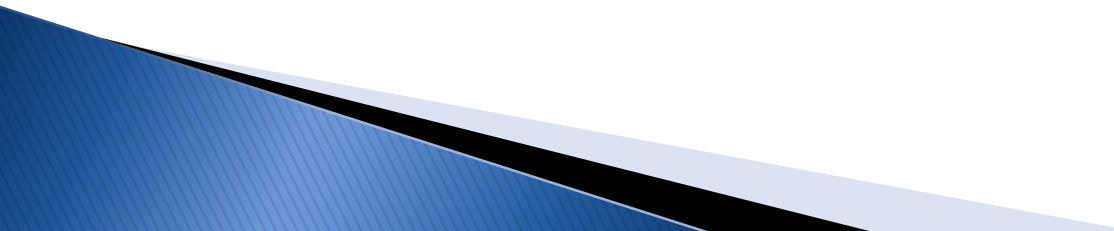
Pseudocode Programming Process (PPP)

1. Design the routine
 2. Code the routine
 3. Check the code
 4. Clean up loose ends
 5. Repeat as needed
- 

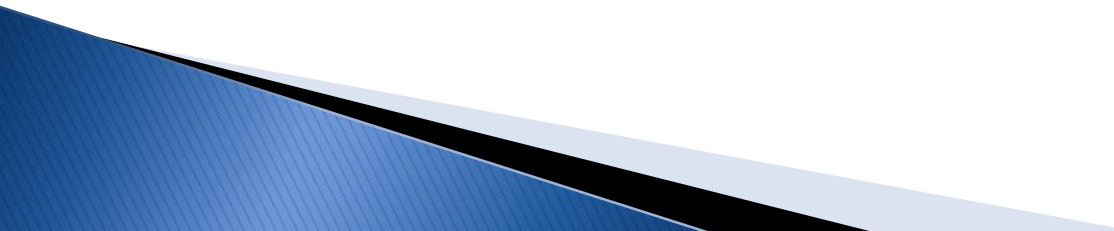
Design the Routine

- ▶ Informal Spec for `ReportErrorMessage()` :
 - `ReportErrorMessage()` **takes** an error code as an input argument and outputs an error message corresponding to the code. It's responsible for handling invalid codes.
 - If the program is operating interactively, `ReportErrorMessage()` **displays** the message to the user.
 - If it's operating in command-line mode, `ReportErrorMessage()` **logs** the message to a message file.
 - After outputting the message, `ReportErrorMessage()` **returns** a status value, indicating whether it succeeded or failed.

Design the Routine (cont'd)

- ▶ Check the prerequisites
 - does the routine fits well into the overall design
 - is the routine demanded?
 - ▶ Define the problem the routine will solve
 - Inputs/outputs
 - Pre/Post-condition
 - ▶ Name the routine
 - Pick a good descriptive name
 - ▶ Decide how to test the routine
- 

Design the Routine (cont'd)

- ▶ Research functionality available in the standard library
 - ▶ Think about error handling
 - ▶ Think about efficiency
 - ▶ Research algorithms and data types
 - ▶ Write the pseudocode
- 

Pseudocode

This routine outputs an error message based on an error code supplied by the calling routine. The way it outputs the message depends on the current processing state, which it retrieves on its own. It returns a value indicating success or failure

set the default status to "fail"

look up the message based on the error code

if the error code is valid

 if doing interactive processing, display the error
 message interactively and declare success

 if doing command line processing, log the error message
 to the command line and declare success

if the error code isn't valid, notify the user that an internal
error has been detected

return status information

Design the Routine (cont'd)

- ▶ Think about the data
 - internal variables, ...
- ▶ Check the pseudocode
- ▶ Try a few ideas in pseudocode, and keep the best (iterate)

Code the Routine

► Write the routine declaration

```
/* This routine outputs an error message based on an error code supplied by  
the calling routine. The way it outputs the message depends on the current  
processing state, which it retrieves on its own. It returns a value  
indicating success or failure */
```

```
Status ReportErrorMessage (ErrorCode errorToReport)
```

```
set the default status to "fail"
```

```
look up the message based on the error code
```

```
if the error code is valid
```

```
    if doing interactive processing, display the error  
    message interactively and declare success
```

```
    if doing command line processing, log the error message  
    to the command line and declare success
```

```
if the error code isn't valid, notify the user that an internal error has  
been detected
```

```
return status information
```

Turn the pseudocode into high-level comments

```
/* This routine outputs an error message based on an error code supplied by the
calling routine. The way it outputs the message depends on the current processing
state, which it retrieves on its own. It returns a value indicating success or
failure */
```

```
Status ReportErrorMessage (ErrorCode errorToReport) {

    // set the default status to "fail"
    // look up the message based on the error code

    // if the error code is valid
        // if doing interactive processing, display the error
        // message interactively and declare success

        // if doing command line processing, log the error
        // message to the command line and declare success

    // if the error code isn't valid, notify the user that an
    // internal error has been detected

    // return status information

}
```

Fill in the code below each comment

```
/* This routine outputs an error message based on an error code supplied by
the calling routine. The way it outputs the message depends on the current
processing state, which it retrieves on its own. It returns a value
indicating success or failure */
```

```
Status ReportErrorMessage (ErrorCode errorToReport) {

    // set the default status to "fail"
    Status errorMessageStatus = Status_Failure;

    // look up the message based on the error code
    Message errorMessage = LookupErrorMessage( errorToReport );

    // if the error code is valid
    if ( errorMessage.validCode() ) {
        ProcessingMethod errorProcessingMethod
            = CurrentProcessingMethod();

        // if doing interactive processing, display the error
        // message interactively and declare success
        if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
            DisplayInteractiveMsg( errorMessage.Text() );
        }
    }
}
```

Fill in the code below each comment

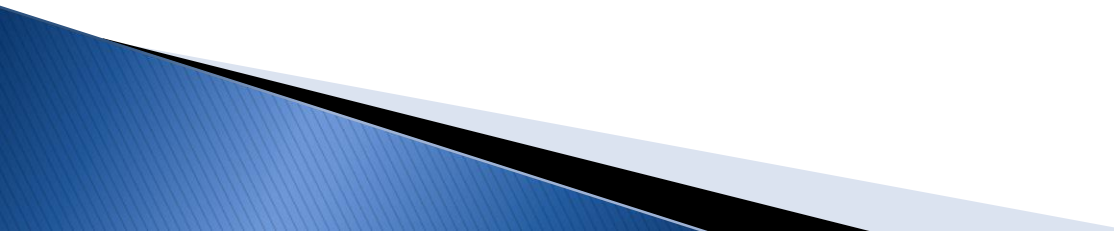
```
// if doing interactive processing, display the error
// message interactively and declare success
if ( errorProcessingMethod == ProcessingMethod_Interactive ) {
    DisplayInteractiveMsg( errorMessage.Text() );
    errorMessageStatus = Status_success;
}
// if doing command line processing, log the error
// message to the command line and declare success
else if (errorProcessingMethod == ProcessingMethod_CmdLine) {
    CommandLine messageLog;
    if ( messageLog.status() == CommandLineStatus_OK ) {
        messageLog.AddMsgQueue( errorMessage.Text() );
        messageLog.FlushMsgQueue();
        errorMessageStatus = Status_success;
    }
    else {
        // can't do anything since an error is being processed
    }
}
else {
    // can't do anything since an error is being processed
}
```


Fill in the code below each comment

```
// if the error code isn't valid, notify the user that an
// internal error has been detected
else {
    DisplayInteractiveMessage(
        "Invalid error code in ReportErrorMessage()" );
}

// return status information
return errorMessageStatus;
}
```

Pseudocode Programming Process (PPP)

1. Design the routine
 2. Code the routine
 3. Check the code
 - test, debugger
 4. Clean up loose ends
 - interface, variable names, layout, logic, documentation, remove unneeded comments, ...
 5. Repeat as needed
- 

Floating Point Review

- ▶ Floating Point lets us:
 - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
 - Store **approximate** values for very large and very small #s.
- ▶ **IEEE 754 Floating Point Standard** is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

single precision:



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
 - Double precision identical, except with exponent bias of 1023 (half, quad similar)

Example: Representing 1/3 in MIPS

▶ 1/3

$$= 0.33333..._{10}$$

$$= 0.25 + 0.0625 + 0.015625 + 0.00390625 + \dots$$

$$= 1/4 + 1/16 + 1/64 + 1/256 + \dots$$

$$= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + \dots$$

$$= 0.0101010101..._2 * 2^0$$

$$= \boxed{1}.0101010101..._2 * 2^{-2}$$

- Sign: 0
- Exponent = $-2 + 127 = 125 = 01111101$
- Significand = 0101010101...

0	0111 1101	0101 0101 0101 0101 0101 010
---	-----------	------------------------------

Floating Point Fallacy

▶ FP add associative?

- $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, and $z = 1.0$
- $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$
 $= (0.0) + 1.0 = \underline{1.0}$
- $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$
 $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$

▶ Therefore, Floating Point add is NOT associative!

- Why?
 - FP result approximates real result!
 - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Precision and Accuracy

Don't confuse these two terms!

Precision is a count of the number bits in a computer word used to represent a value.

Accuracy is a measure of the difference between the actual value of a number and its computer representation.

High precision permits high accuracy but doesn't guarantee it. It is possible to have high precision but low accuracy.

Example: `float pi = 3.14;`
pi will be represented using all 24 bits of the significant (highly precise), but is only an approximation (not accurate).

Representation for $\pm \infty$

- ▶ In FP, divide by 0 should produce $\pm \infty$, not overflow.
- ▶ Why?
 - OK to do further computations with ∞
 - E.g., $X/0 > Y$ may be a valid comparison
 - Ask math majors
- ▶ IEEE 754 represents $\pm \infty$
 - **Most positive** exponent reserved for ∞
 - Significands **all zeroes**

0	1111 1111	0000 0000 0000 0000 0000 000
---	-----------	------------------------------

Representation for 0

► Represent 0?

- exponent all zeroes
- significand all zeroes
- What about sign?
 - Both cases valid.

+0: 0 00000000 00000000000000000000000000000000

-0: 1 00000000 00000000000000000000000000000000

Special Numbers

- ▶ What have we defined so far?
- ▶ (Single Precision)

Exponent	Significand	Object
0	0	0
0	Nonzero	???
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	???

- ▶ “Waste not, want not”
 - We’ll talk about Exp=0,255 & Sig!=0 next

Representation for "Not a Number"

- ▶ What do you get if you calculate `sqrt(-4.0)` or `0/0`?
 - If ∞ is not an error, these shouldn't be either
 - Called **Not a Number (NaN)**
 - **Exponent = 255, Significand nonzero**
- ▶ Why is this useful?
 - Hope NaNs help with debugging
 - They contaminate: `op(NaN, X) = NaN`

Representation for Denorms (1/2)

- ▶ Problem: There's a gap among representable FP numbers around 0

- Smallest representable positive number:

$$a = 1.0..._2 * 2^{-126} = 2^{-126} \text{ (exp = 1, sig = 0)}$$

- Second smallest representable pos num:

$$b = 1.000.....1_2 * 2^{-126} \text{ (exp = 1, sig = 1)}$$

$$= (1 + 0.00...1_2) * 2^{-126}$$

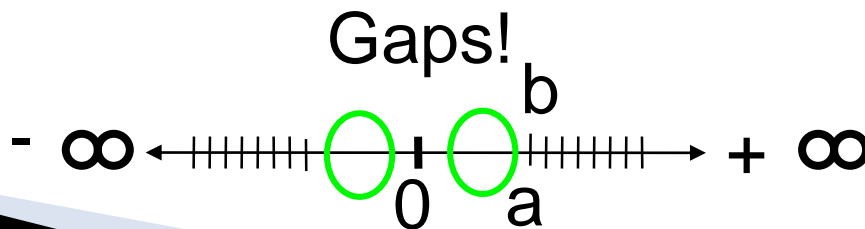
$$= (1 + 2^{-23}) * 2^{-126}$$

$$= 2^{-126} + 2^{-149}$$

$$a - 0 = 2^{-126}$$

$$b - a = 2^{-149}$$

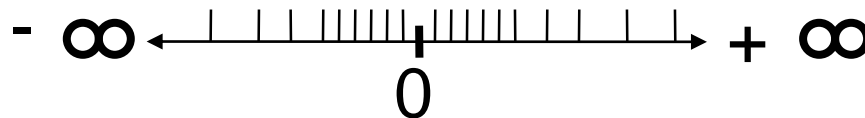
Normalization and implicit 1 is to blame!



Representation for Denorm (2/2)

► Solution:

- We still haven't used Exponent=0, Significand nonzero
- Denormalized number: no (implied) leading 1, implicit exponent = -126 ($0 - 127 + 1$)
 - $(-1)^S \times (\text{Significand}) \times 2^{(-126)}$
- Smallest representable pos num:
 - $a = 2^{-149}$ (sig = 1)
- Second smallest representable pos num:
 - $b = 2^{-148}$ (sig = 2)



Special Numbers Summary

- ▶ Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

Rounding

- ▶ When we perform math on real numbers, we have to worry about rounding to fit the result in the significant field.
- ▶ The FP hardware carries two extra bits of precision, and then round to get the proper value
- ▶ Rounding also occurs when converting:
 - double to a single precision value, or floating point number to an integer

IEEE FP Rounding Modes

- ▶ Halfway between two floating point values (rounding bits read 10)? Choose from the following:
 - Round towards $+\infty$
 - Round “up”: $1.01 \underline{10} \rightarrow 1.10$, $-1.01 \underline{10} \rightarrow -1.01$
 - Round towards $-\infty$
 - Round “down”: $1.01 \underline{10} \rightarrow 1.01$, $-1.01 \underline{10} \rightarrow -1.10$
- ▶ Truncate
 - Just drop the extra bits (round towards 0)
- ▶ **Unbiased (default mode)**. Round to nearest EVEN number
 - Half the time we round up on tie, the other half time we round down. Tends to balance out inaccuracies.
 - In binary, even means least significant bit is 0.
- ▶ Otherwise, not halfway (00, 01, 11)! Just round to the nearest float.

Casting floats to ints and vice versa

(int) floating_point_expression

Coerces and converts it to the nearest integer

(C uses truncation)

```
i = (int) (3.14159 * f);
```

(float) integer_expression

Converts integer to nearest floating point

```
f = f + (float) i;
```


int → float → int

```
if (i == (int) ((float) i)) {  
    printf("true");  
}
```

- ▶ Will not always print “true”
- ▶ Most large values of integers don’t have exact floating point representations!
- ▶ What about double?

float → int → float

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- ▶ **Will not** always print “true”
- ▶ Small floating point numbers (<1) don't have integer representations
- ▶ For other numbers, rounding errors

Quiz 1:

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:
$$(x+y)+z = x+(y+z)$$

ABC
1: FFF
2: FFT
3: FTF
4: FTT
5: TFF

Quiz 1:

1. Converting float -> int -> float produces same float number
2. Converting int -> float -> int produces same int number
3. FP add is associative:
$$(x+y) + z = x + (y+z)$$

1. 3.14 -> 3 -> 3

2. 32 bits for signed int,
but 24 for FP mantissa?

3. x = biggest pos #,
y = -x, z = 1 (x != inf)

ABC

1: FFF

2: FFT

3: FTF

4: FTT

5: TFF

Quiz 2:

- ▶ Let $f(1, 2)$ = # of floats between 1 and 2
- ▶ Let $f(2, 3)$ = # of floats between 2 and 3

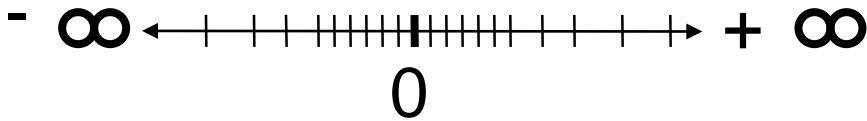
1: $f(1,2) < f(2,3)$

2: $f(1,2) = f(2,3)$

3: $f(1,2) > f(2,3)$

Quiz 2:

- ▶ Let $f(1, 2) = \#$ of floats between 1 and 2
- ▶ Let $f(2, 3) = \#$ of floats between 2 and 3



1: $f(1,2) < f(2,3)$

$$2: f(1,2) = f(2,3)$$

3: $f(1,2) > f(2,3)$

Summary

- ▶ Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	Nonzero	Denorm
1-254	Anything	+/- FP #
255	0	+/- ∞
255	Nonzero	NaN

- ▶ 4 Rounding modes (default: unbiased)
- ▶ MIPS FI ops complicated, expensive