

**CS 201**

**Operations and Arithmetic**

**Floating point representation**

# Operations in C

## Have the data, what now?

- Boolean operations
- Logical operations
- Arithmetic operations

# Bit-level Boolean Operations in C

## Bit-level Operations $\&$ , $|$ , $\sim$ , $\wedge$ Available in C

- $\&$ 
  - Boolean AND,  $A\&B=1$  only when both  $A=1$  and  $B=1$
- $|$ 
  - Boolean OR,  $A|B=1$  when either  $A=1$  or  $B=1$
- $\sim$ 
  - Boolean NOT,  $\sim A=1$  when  $A=0$
- $\wedge$ 
  - Boolean Exclusive OR (XOR),  $A\wedge B=1$  when either  $A=1$  or  $B=1$ , but not both
- If you do not know what these are, see Figure 2.6 in B&O
- Apply to any “integral” data type
  - long, int, short, char
  - View arguments as bit vectors
  - Arguments applied bit-wise

# Examples

**0x69 & 0x55**

01101001

01010101

01000001 = 0x41

**0x69 | 0x55**

01101001

01010101

01111101 = 0x7D

**0x69 ^ 0x55**

01101001

01010101

00111100 = 0x3C

**~0x55**

01010101

10101010 = 0xAA

# Shift Operations

## Left Shift: $x \ll y$

- Shift bit-vector  $x$  left  $y$  positions
  - Throw away extra bits on left
  - Fill with 0's on right

## Right Shift: $x \gg y$

- Shift bit-vector  $x$  right  $y$  positions
  - Throw away extra bits on right
- Logical shift
  - Fill with 0's on left
- Arithmetic shift
  - Replicate most significant bit on left
  - Useful with two's complement integer representation
  - Perform division by 2 via shift

Argument $x$	01100010
$x \ll 3$	00010000
Log. $x \gg 2$	00011000
Arith. $x \gg 2$	00011000

Argument $x$	10100010
$x \ll 3$	00010000
Log. $x \gg 2$	00101000
Arith. $x \gg 2$	11101000

# Exercise

<b>x</b>	<b>x&lt;&lt;3</b>	<b>x&gt;&gt;2</b> <b>(Logical)</b>	<b>x&gt;&gt;2</b> <b>(Arithmetic)</b>
<b>0xf0</b>			
<b>0x0f</b>			
<b>0xcc</b>			
<b>0x55</b>			

# Exercise

<b>x</b>	<b>x&lt;&lt;3</b>	<b>x&gt;&gt;2</b> <b>(Logical)</b>	<b>x&gt;&gt;2</b> <b>(Arithmetic)</b>
<b>0xf0</b>	<b>0x80</b>	<b>0x3c</b>	<b>0xfc</b>
<b>0x0f</b>	<b>0x78</b>	<b>0x03</b>	<b>0x03</b>
<b>0xcc</b>	<b>0x60</b>	<b>0x33</b>	<b>0xf3</b>
<b>0x55</b>	<b>0xa8</b>	<b>0x15</b>	<b>0x15</b>

# Logic Operations in C

## Comparison operators

- `>`, `>=`, `<`, `<=`, `==`, `!=`
  - Always return 0 or 1

## Logical Operators

- `&&`, `||`, `!`
  - 0 is “False”, anything nonzero is “True”
  - Always return 0 or 1

## Examples (char data type)

- `!0x41 --> 0x00`
- `!0x00 --> 0x01`
- `!!0x41 --> 0x01`

## What are the values of:

- `0x69 || 0x55`
- `0x69 | 0x55`
- `p && *p` (avoids null pointer access)

# Logical operations in C

## Applying bit-wise and logical operators

**int x, y;**

**For any processor, independent of the size of an integer, write C expressions without any “=” signs that are true if:**

- x and y have any non-zero bits in common in their low order byte
- x has any 1 bits at higher positions than the low order 8 bits
- x has no 1 bits at higher positions than the low order 8 bits
- x is zero
- `x == y`

# Logical operations in C

## Applying bit-wise and logical operators

`int x, y;`

For any processor, independent of the size of an integer, write C expressions without any “=” signs that are true if:

- x and y have any non-zero bits in common in their low order byte  
 $0xff \& (x \& y)$
- x has any 1 bits at higher positions than the low order 8 bits  
 $\sim 0xff \& x$
- x has no 1 bits at higher positions than the low order 8 bits  
 $!(\sim 0xff \& x)$
- x is zero  
 $!x$
- `x == y`  
 $!(x \wedge y)$

# Arithmetic operations

## Signed/unsigned

- Addition and subtraction
- Multiplication
- Division

# Unsigned addition

**Suppose we have a computer with 4-bit words**

**What is the unsigned value of  $7 + 7$ ?**

- $0111 + 0111$

**What about  $9 + 9$ ?**

- $1001 + 1001$

**With  $w$  bits, unsigned addition is regular addition, modulo  $2^w$**

- Bits beyond  $w$  are discarded

# Unsigned addition

**With 32 bits, unsigned addition is modulo what?**

**What is the value of `0xc0000000 + 0x70004444` ?**

```
#include <stdio.h>
unsigned int sum(unsigned int a, unsigned int b)
{
    return a+b;
}
main () {
    unsigned int i=0xc0000000;
    unsigned int j=0x70004444;
    printf("%x\n",sum(i,j));
}
```

Output: 30004444

# Two's-Complement Addition

**Two's-complement numbers have a range of**

$$-2^{w-1} \leq x, y \leq 2^{w-1} - 1$$

**Their sum has the range**

$$-2^w \leq x + y \leq 2^w - 2$$

**When actual represented result is truncated, it is not modular as unsigned addition**

- However, the bit representation for signed and unsigned addition is the same

# Two's-Complement Addition

Since we are dealing with signed numbers, we can have negative overflow or positive overflow

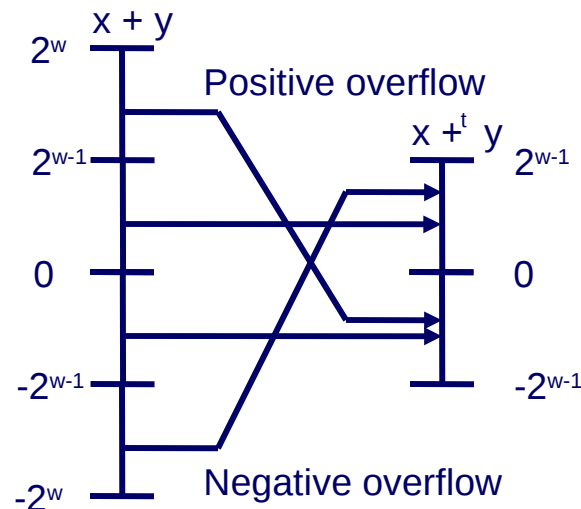
$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases}$$

Case 4

Case 3

Case 2

Case 1



# Example (w=4)

<b>x</b>	<b>y</b>	<b>x + y</b>	<b><math>x + \binom{t}{4}y</math></b>
<b>-8</b> <b>[1000]</b>	<b>-5</b> <b>[1011]</b>	<b>-13</b> <b>[10011]</b>	<b>3</b> <b>[0011]</b>
<b>-8</b> <b>[1000]</b>	<b>-8</b> <b>[1000]</b>	<b>-16</b> <b>[10000]</b>	<b>0</b> <b>[0000]</b>
<b>-8</b> <b>[1000]</b>	<b>5</b> <b>[0101]</b>	<b>-3</b> <b>[1101]</b>	<b>-3</b> <b>[1101]</b>
<b>2</b> <b>[0010]</b>	<b>5</b> <b>[0101]</b>	<b>7</b> <b>[0111]</b>	<b>7</b> <b>[0111]</b>
<b>5</b> <b>[0101]</b>	<b>5</b> <b>[0101]</b>	<b>10</b> <b>[1010]</b>	<b>-6</b> <b>[1010]</b>

Case 1

Case 1

Case 2

Case 3

Case 4

$$x + y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases} \begin{matrix} \text{(Case 4)} \\ \text{(Case 2/3)} \\ \text{(Case 1)} \end{matrix}$$

# Unsigned Multiplication

**For unsigned numbers:  $0 \leq x, y \leq 2^{w-1} - 1$**

- Thus,  $x$  and  $y$  are  $w$ -bit numbers

**The product  $x*y$ :  $0 \leq x * y \leq (2^{w-1} - 1)^2$**

- Thus, product can require  $2w$  bits

**Only the low  $w$  bits are used**

- The high order bits may overflow

**This makes unsigned multiplication modular**

$$x *_{w}^u y = (x * y) \bmod 2^w$$

# Two's-Complement Multiplication

## Same problem as unsigned

- The result of multiplying two  $w$ -bit numbers could be as large as  $2w$  bits

## The bit-level representation for two's-complement and unsigned is identical

- This simplifies the integer multiplier

## As before, the interpretation of this value is based on signed vs. unsigned

# Multiplication by Powers of Two

**What happens if you shift a binary number left one bit?**

**What if you shift it left N bits?**

$$00001000_2 \ll 2 = 00100000_2$$

$$(8_{10}) \ll 2 = (32_{10})$$

$$11111000_2 \ll 2 = 11100000_2$$

$$(-8_{10}) \ll 2 = (-32_{10})$$

## Examples

$$u \ll 3 \quad == \quad u * 8$$

$$(u \ll 5) - (u \ll 3) == \quad u * 24$$

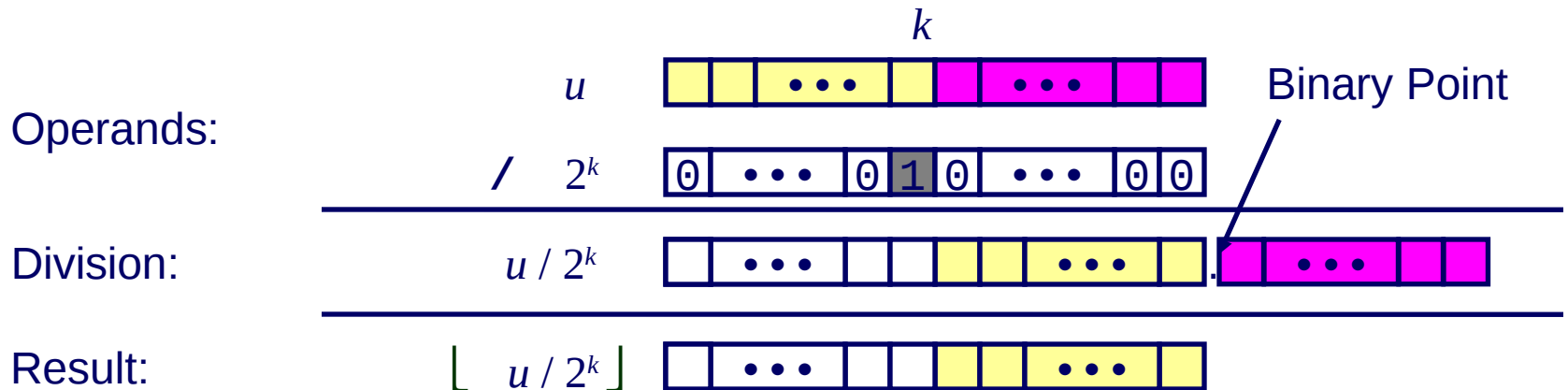
- Most machines shift and add faster than multiply
  - Compiler may generate this code automatically

# Dividing by Powers of Two

For unsigned numbers, performed via logical right shifts

Quotient of unsigned division by power of 2

- $u \gg k$  gives  $\lfloor u / 2^k \rfloor$



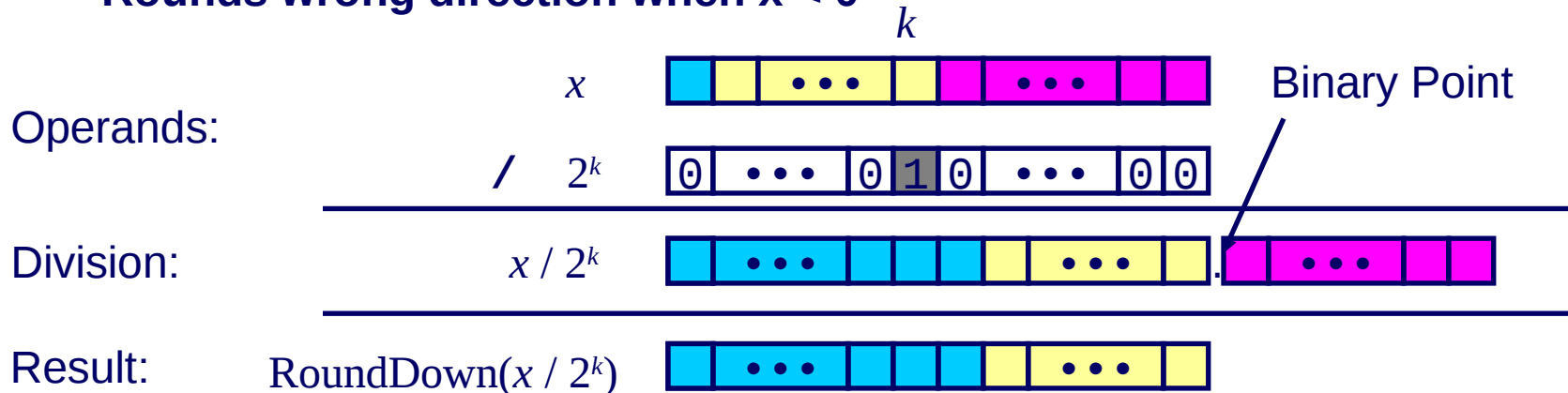
	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

# Dividing by Powers of Two

For signed numbers, performed via arithmetic right shifts

Quotient of signed division by power of 2

- $x \gg k$  gives  $\lfloor x / 2^k \rfloor$
- Rounds wrong direction when  $x < 0$



	Division	Computed	Hex	Binary
$y$	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	<b>1</b> 1100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	<b>1111</b> 1100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	<b>11111111</b> 11000100

# Floating point representation and operations

# Floating Point

## Integer data type

- 32-bit unsigned integers limited to whole numbers from 0 to just over 4 billion
- What about small numbers and fractions (e.g.  $1/2$  or  $\pi$ )?
- What about large numbers (e.g. national debt, bank bailout bill)?

## Requires a different interpretation of the bits!

- New data types in C
  - `float` (32-bit IEEE floating point format)
  - `double` (64-bit IEEE floating point format)
- 32-bit `int` and `float` both represent  $2^{32}$  distinct values!
  - Trade-off range and precision
  - e.g. to support large numbers ( $> 2^{32}$ ) and fractions, `float` can not represent every integer between 0 and  $2^{32}$ !

# Floating Point overview

**Problem: how can we represent very large or very small numbers with a compact representation?**

- Current way with int
  - $5 \cdot 2^{100}$  as 1010000....000000000000? (103 bits)
  - Not very compact, but can represent all integers in between
- Another
  - $5 \cdot 2^{100}$  as 101 01100100 (i.e.  $x=101$  and  $y=01100100$ )? (11 bits)
  - Compact, but does not represent all integers in between

**Basis for IEEE Standard 754, “IEEE Floating Point”**

- Supported in most modern CPUs via floating-point unit
- Encodes rational numbers in the form  $(x * 2^y)$ 
  - Large numbers have positive  $y$  exponent
  - Small numbers have negative  $y$  exponent
  - Rounding can lead to errors

# IEEE Floating-Point

**Specifically, IEEE FP represents numbers in the form**

- $V = (-1)^s * M * 2^E$

## **Three fields**

- **s** is sign bit: 1 == negative, 0 == positive
- **M** is the *significand*, a fractional number
- **E** is the, possibly negative, exponent

# Fractional Binary Numbers

**In Base 10, a decimal point for representing non-integer values**

- **125.35** is  $1*10^2+2*10^1+5*10^0+3*10^{-1}+5*10^{-2}$

**In Base 2, a binary point**

- $b_n b_{n-1} \dots b_1 b_0 . b_{-1} b_{-2} \dots b_{-m}$
- $b = \sum 2^i * b_i, \quad i = -m \dots n$
- Example: **101.11**<sub>2</sub> is
  - $1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2}$
  - $4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$

**Accuracy is a problem**

- Numbers such as 1/5 or 1/3 must be approximated
  - This is true also with decimal

# IEEE Floating Point Encoding



- **s** is sign bit
- **exp** field encodes  $E$
- **frac** field encodes  $M$
- **Sizes**
  - Single precision: 8 exp bits, 23 frac bits (32 bits total)
    - » C type `float`
  - Double precision: 11 exp bits, 52 frac bits (64 bits total)
    - » C type `double`
  - Extended precision: 15 exp bits, 63 frac bits
    - » Only found in Intel-compatible machines
    - » Stored in 80 bits (1 bit wasted)

# IEEE Floating-Point

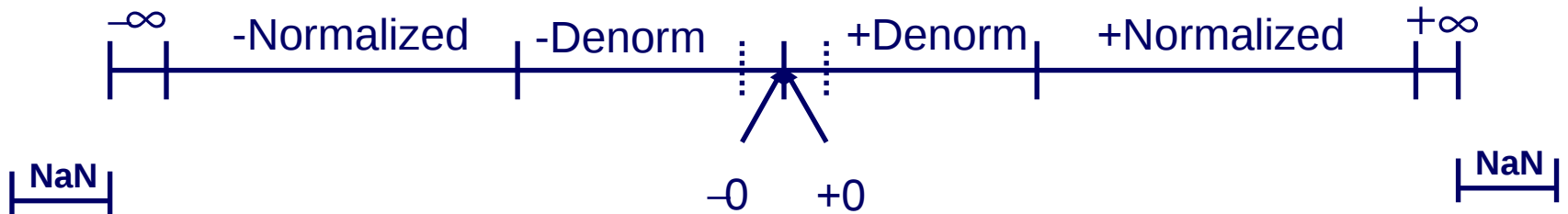
Depending on the exp value, the bits are interpreted differently

- Normalized: exp is neither all 0's nor all 1's
  - E is  $(\text{exp} - \text{Bias})$ 
    - » E is in biased form:
      - Bias=127 for single precision
      - Bias=1023 for double precision
    - » Allows for negative exponents
  - M is  $1 + \text{frac}$
- Denormalized (numbers close to 0): exp is all 0's
  - E is  $1 - \text{Bias}$ 
    - » Not set to  $-\text{Bias}$  in order to ensure smooth transition from Normalized
  - M is  $\text{frac}$ 
    - » Can represent 0 exactly
    - » IEEE FP handles +0 and -0 differently
- Special values: exp is all 1's
  - If  $\text{frac} == 0$ , then we have  $\pm\infty$ , e.g., divide by 0
  - If  $\text{frac} != 0$ , we have NaN (Not a Number), e.g.,  $\text{sqrt}(-1)$

# Encodings form a continuum

## Why two regions?

- Allows 0 to be represented
- Allows for smooth transition near 0
- Encoding allows magnitude comparison to be done via integer unit



# Normalized Encoding Example

## Using 32-bit float

### Value

- `float f = 15213.0; /* exp=8 bits, frac=23 bits */`
- $15213_{10} = 11101101101101_2$   
=  $1.1101101101101_2 \times 2^{13}$  (normalized form)

### Significand

- $M = 1.1101101101101_2$
- $\text{frac} = \underline{1101101101101000000000}_2$

### Exponent

- $E = 13$
- $\text{Bias} = 127$
- $\text{Exp} = 140 = 10001100_2$

### Floating Point Representation :

Hex:        4        6        6        D        B        4        0        0

Binary: 0100 0110 0110 1101 1011 0100 0000 0000

140:        100 0110 0

15213:                1110 1101 1011 01

# Denormalized Encoding Example

Using 32-bit float

Value

- `float f = 7.347e-39; /* 7.347*10-39 */`

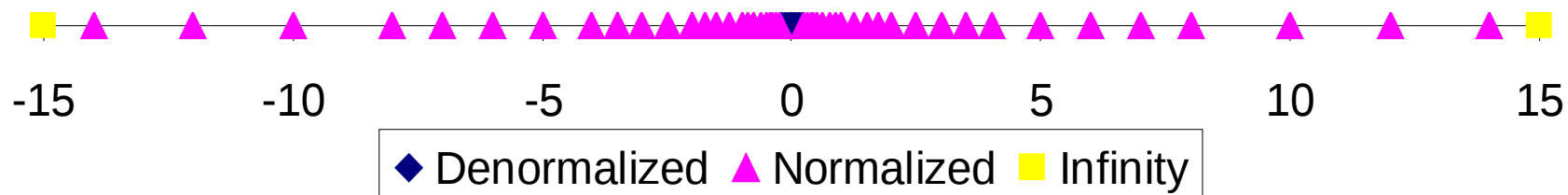
[http://thefengs.com/wuchang/work/courses/cs200/class/05/denormalized\\_float.c](http://thefengs.com/wuchang/work/courses/cs200/class/05/denormalized_float.c)

# Distribution of Values

## 7-bit IEEE-like format

- $e = 4$  exponent bits
- $f = 3$  fraction bits
- Bias is 7

**Notice how the distribution gets denser toward zero.**



# 7-bit IEEE FP format (Bias=7)

	s	exp	frac	E	Value		
Denormalized numbers	0	0000	000	-6	0		
	0	0000	001	-6	$1/8 * 1/64 = 1/512$	← closest to zero	
	0	0000	010	-6	$2/8 * 1/64 = 2/512$		
	...						
	0	0000	110	-6	$6/8 * 1/64 = 6/512$		
	0	0000	111	-6	$7/8 * 1/64 = 7/512$	← largest denorm	
	<hr/>						
	0	0001	000	-6	$8/8 * 1/64 = 8/512$	← smallest norm	
	0	0001	001	-6	$9/8 * 1/64 = 9/512$		
	...						
Normalized numbers	0	0110	110	-1	$14/8 * 1/2 = 14/16$		
	0	0110	111	-1	$15/8 * 1/2 = 15/16$	← closest to 1 below	
	0	0111	000	0	$8/8 * 1 = 1$		
	0	0111	001	0	$9/8 * 1 = 9/8$	← closest to 1 above	
	0	0111	010	0	$10/8 * 1 = 10/8$		
	...						
	0	1110	110	7	$14/8 * 128 = 224$		
	0	1110	111	7	$15/8 * 128 = 240$	← largest norm	
	<hr/>						
	0	1111	000	n/a	inf		

# Floating Point Operations

## FP addition is

- Commutative:  $x + y = y + x$
- NOT associative:  $(x + y) + z \neq x + (y + z)$ 
  - $(3.14 + 1e10) - 1e10 = 0.0$ , due to rounding
  - $3.14 + (1e10 - 1e10) = 3.14$
- Very important for scientific and compiler programmers

## FP multiplication

- Is not associative
- Does not distribute over addition
  - $1e20 * (1e20 - 1e20) = 0.0$
  - $1e20 * 1e20 - 1e20 * 1e20 = \text{NaN}$
- Again, very important for scientific and compiler programmers

# Floating Point in C

## C supports two levels

`float`      single precision (32-bit)

`double`     double precision (64-bit)

## Conversions

- Casting between `int`, `float`, and `double` changes numeric values
- `double` or `float` to `int`
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range
    - » Generally saturates to TMin or TMax
- `int` to `double`
  - Exact conversion, as long as `int` has  $\leq 53$  bit word size
- `int` to `float`
  - Will round according to rounding mode

# Floating Point Puzzles

- For each of the following C expressions, either:
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = ...;  
float f = ...;  
double d = ...;
```

Assume neither  
d nor f is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (float) d`
- `f == -(-f);`
- `2/3 == 2/3.0`
- `d < 0.0                   ⇒       ((d*2) < 0.0)`
- `d > f   ⇒ -f > -d`
- `d * d >= 0.0`
- `(d+f)-d == f`

# Answers to Floating Point Puzzles

```
int x = ...;
float f = ...;
double d = ...;
```

Assume neither  
d nor f is NAN

- `x == (int)(float) x` No: 24 bit significand
- `x == (int)(double) x` Yes: 53 bit significand
- `f == (float)(double) f` Yes: increases precision
- `d == (float) d` No: loses precision
- `f == -(-f);` Yes: Just change sign bit
- `2/3 == 2/3.0` No: `2/3 == 0`
- `d < 0.0 ⇒ ((d*2) < 0.0)` Yes (Note use of  $-\infty$ )
- `d > f ⇒ -f > -d` Yes!
- `d * d >= 0.0` Yes! (Note use of  $+\infty$ )
- `(d+f) - d == f` No: Not associative

# Wait a minute...

```
int x = ...;
float f = ...;
double d = ...;
```

## Recall

- `x == (int)(float) x` No: 24 bit significand

**Compiled with gcc -O2, this is true!**

**Example with `x = 2147483647`.**

## What's going on?

- See B&O 2.4.6
- IA32 uses 80-bit floating point registers
- Optimized code does not return intermediate results into memory
  - Keeps case in 80-bit register
- Non-optimized code returns results into memory
  - 32 bits for intermediate float

[http://thefengs.com/wuchang/work/courses/cs200/class/05/cast\\_ia32\\_noround.c](http://thefengs.com/wuchang/work/courses/cs200/class/05/cast_ia32_noround.c)

# Practice problem 2.47

**Consider a 5-bit IEEE floating point representation**

- 1 sign bit, 2 exponent bits, 2 fraction bits
- Bias = 1 ( $2^{(2-1)} - 1$ )

**Fill in the following table**

<b>Bits</b>	<b>exp</b>	<b>E</b>	<b>frac</b>	<b>M</b>	<b>V</b>
<b>0 00 00</b>					
<b>0 00 11</b>					
<b>0 01 00</b>					
<b>0 01 10</b>					
<b>0 10 11</b>					

# Practice problem 2.47

**Consider a 5-bit IEEE floating point representation**

- 1 sign bit, 2 exponent bits, 2 fraction bits
- Bias = 1 ( $2^{(2-1)} - 1$ )

**Fill in the following table**

<b>Bits</b>	<b>exp</b>	<b>E</b>	<b>frac</b>	<b>M</b>	<b>V</b>
<b>0 00 00</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>0 00 11</b>	<b>0</b>	<b>0</b>	<b><math>\frac{3}{4}</math></b>	<b><math>\frac{3}{4}</math></b>	<b><math>\frac{3}{4}</math></b>
<b>0 01 00</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>
<b>0 01 10</b>	<b>1</b>	<b>0</b>	<b><math>\frac{1}{2}</math></b>	<b><math>1 \frac{1}{2}</math></b>	<b><math>1 \frac{1}{2}</math></b>
<b>0 10 11</b>	<b>2</b>	<b>1</b>	<b><math>\frac{3}{4}</math></b>	<b><math>1 \frac{3}{4}</math></b>	<b><math>3 \frac{1}{2}</math></b>

# Practice problem 2.49

**For a floating point format with a  $k$ -bit exponent and an  $n$ -bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an  $n+1$  bit fraction to be exact)**

# Practice problem 2.49

**For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an n+1 bit fraction to be exact)**

- What is the smallest n+1 bit integer?
  - $2^{(n+1)}$ 
    - » Can this be represented exactly?
    - » Yes.  $s=0$ ,  $\text{exp}=\text{Bias}+n+1$ ,  $\text{frac}=0$
    - »  $E=n+1$ ,  $M=1$ ,  $V=2^{(n+1)}$
- What is the next largest n+1 bit integer?
  - $2^{(n+1)} + 1$ 
    - » Can this be represented exactly?
    - » No. Need an extra bit in the fraction.

# Extra slides

# Multiplication example (3-bit numbers)

Mode	x	y	x*y	trunc(x*y)
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
Two's-comp	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
Two's-comp	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
Two's-comp	3 [011]	3 [011]	9 [001001]	1 [001]

# Questions

- Suppose  $x = 8 * 2^{100}$ ,  $y = 1.23 * 2^{-100}$
- What kind of thing is each of the following expressions?

```
int a = ...;  
float f = ...;  
double d = ...;
```

- `f = x ** 2;`
- `d = x ** 2;`
- `a = x;`
- `d = y ** 2;`
- `a = x/y;`
- `a = y/x`

reminder: `infinity.c`