Applied Programming

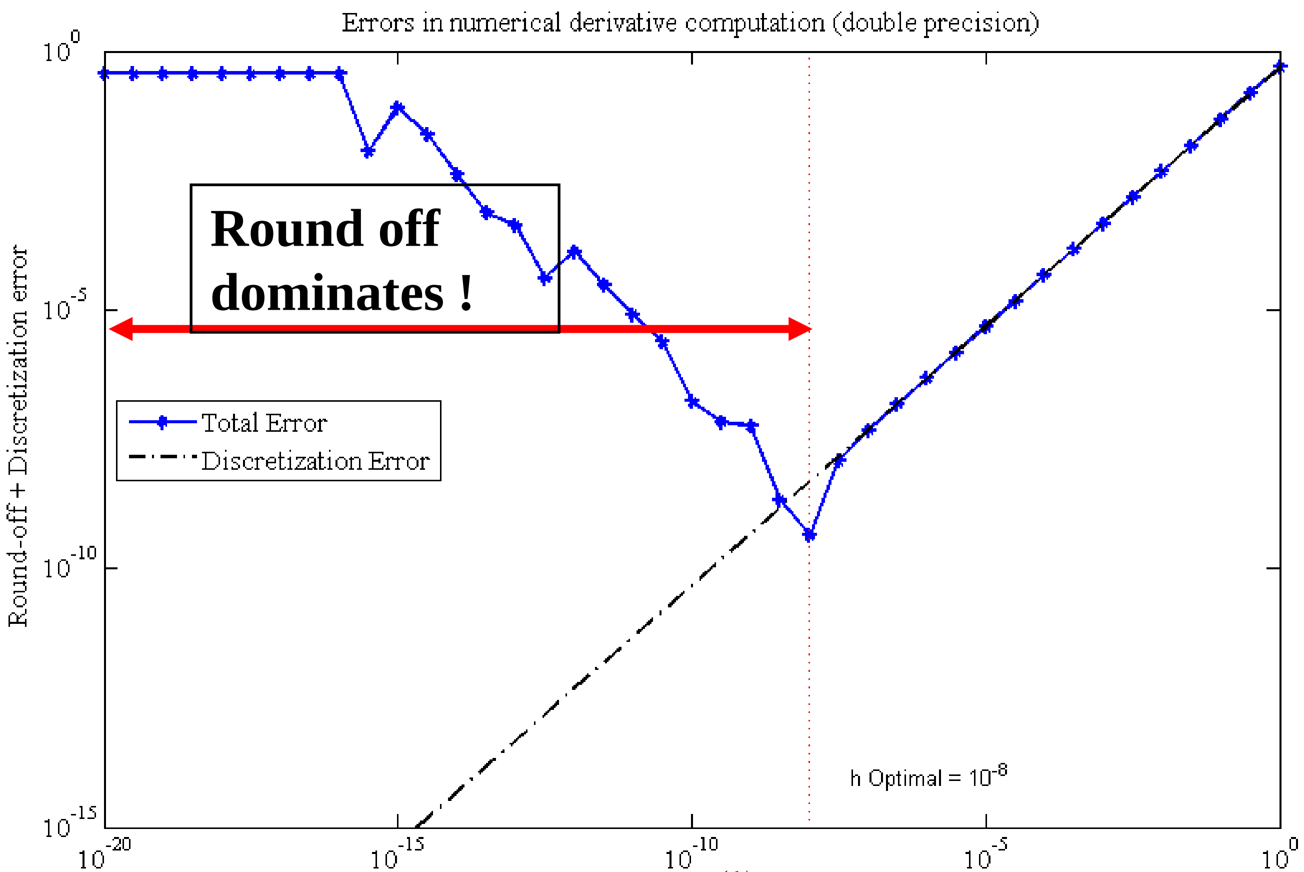
Representation of Numbers for Computing

Floating Point Systems

and the IEEE 754 Standard

Round-off Errors

- Impacts our calculations



$f(x)$=$sin(x)$, find $f^{(1)}(x_o)$ at x_o=1.2

Round-off in Computing

- For reliable numerical computations we need to ***keep round-off errors small***.
- The ***main source of round-off*** errors comes from the fact that :
    - Computers ***cannot represent*** all (real) ***numbers exactly.***
- Why we care about round-off :
    - ***Arithmetic*** with approximate numbers ***propagates round-off errors…***

… and we need to make sure they remain small

Computers, Numbers and C

- For computing we can distinguish between two types numbers: ***integers*** and ***reals***
  - ***Integers*** in C are provided by **`int`**
    - With additional qualifiers like **`short, long`**
  - ***Reals*** in C are provided by **`float`** and **`double`**

Most practical ***engineering problems*** involve ***computation with real numbers***

Integers and Reals
Main differences/similarities

Mathematics (ideal)

- There is an infinite number of integers and reals
- The range of integers and reals is infinite $(-\infty, \infty)$
- Integers are discrete
- Reals are continuous

Computers (practical)

- Can only represent a finite number of integers and reals
- The range of integers and reals is finite (-??,??)
- Integers are discrete
- Reals are discrete

Integers in C

- **`limits.h`** defines macros related to the representation of ***integers*** such as:
  - ***Range*** and ***storage space***

- Example: **`the_integers.c`**

Engineering Significance

- ***Integers*** can be ***represented exactly*** but …
- We can only represent a ***finite range*** of integers.

Consequence: ***Arithmetic with integers may cause overflow*** (reduce risk by ***scaling*** numbers properly)

Range of Integers

```
/******************************************************************************
 * The Integers in C
 * Author: Juan C. Cockburn
 * Reference: http://www.tutorialspoint.com/c_standard_library/limits_h.htm
 ******************************************************************************/
#include <stdio.h>
#include <limits.h>
int main() {
   printf("Integer numbers in Standard   \n");
   printf("------------------------------\n\n");
   printf("The number of bits in a byte %d\n\n", CHAR_BIT);
   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n\n", UCHAR_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR  INT = %d\n\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);
   printf("The maximum value of UNSIGNED SHORT INT = %d\n\n", USHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);
   printf("The maximum value of UNSIGNED INT = %d\n\n", UINT_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);
   printf("The maximum value of UNSIGNED LONG = %lu\n\n", ULONG_MAX);
   return(0);
}
```

Range of Integers

```
Range of integer types
-----------------------------------------------
The minimum value of CHAR          = -128
The minimum value of SIGNED CHAR   = -128
The maximum value of CHAR          =  127
The maximum value of SIGNED CHAR   = +127
The maximum value of UNSIGNED CHAR =  255

The minimum value of SHORT INT          = -32768
The maximum value of SHORT INT          = +32767
The maximum value of UNSIGNED SHORT INT =  65535

The minimum value of INT          = -2147483648
The maximum value of INT          = +2147483647
The maximum value of UNSIGNED INT =  4294967295

The minimum value of LONG INT      = -9223372036854775808
The maximum value of LONG INT      = +9223372036854775807
The maximum value of UNSIGNED LONG = 18446744073709551615
```

Reals in C

- `float.h` defines macros related to the representation of real number such as:
  - ***range***, ***storage space***, and ***machine epsilon***

- Example: **`the_reals.c`**

Engineering Significance

- Only ***some reals*** can be represented ***exactly*** and ***most*** can only be represented ***approximately***
- Can only represent a ***finite range*** of reals.
- In addition to ***overflow*** reals can also ***underflow.***
  - there is a ***gap between 0 and the smallest possible number***

Reals in C

```c
#include <stdio.h>
#include <float.h>
int main() {
 printf("The maximum value of float = %.10e\n",   FLT_MAX);
 printf("The minimum value of float = %.10e\n\n", FLT_MIN);

 printf("The maximum value of DOUBLE = %.10e\n",   DBL_MAX);
 printf("The minimum value of DOUBLE = %.10e\n\n", DBL_MIN);

 printf("The maximum value of LONG DOUBLE = %.10e\n",   LDBL_MAX);
 printf("The minimum value of LONG DOUBLE = %.10e\n\n", LDBL_MIN);

 printf("The SINGLE PRECISION (float) machine epsilon is = %.10Le\n",  FLT_EPSILON);
 printf("The DOUBLE PRECISION (double) machine epsilon is = %.10Le\n", DBL_EPSILON);
 printf("The EXTENDED PRECISION (long double) machine epsilon is = %.10Le\n\n", LDBL_EPSILON);

 printf("IEEE-734 SINGLE PRECISION representation\n\m");
 printf("The number of binary digits (p) in a float is = % d\n", FLT_MANT_DIG);
 printf("The maximum value of the exponent (U) in a float is = % d\n",   FLT_MAX_10_EXP);
 printf("The minimum value of the exponent (L) in a float is = % d\n", FLT_MIN_10_EXP);
}
```

Reals in C

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38

The maximum value of DOUBLE = 1.7976931349e+308
The minimum value of DOUBLE = 2.2250738585e-308

The maximum value of LONG DOUBLE = 1.1897314954e+4932
The minimum value of LONG DOUBLE = 3.3621031431e-4932

The   SINGLE PRECISION (float)        machine epsilon is = 1.1920928955e-07
The   DOUBLE PRECISION (double)       machine epsilon is = 2.2204460493e-16
The EXTENDED PRECISION (long double) machine epsilon is = 1.0842021725e-19

IEEE-734 SINGLE PRECISION representation
The number of binary digits (p) in a float is =  24
The maximum value of the exponent (U) in a float is =  38
The minimum value of the exponent (L) in a float is = -37
```

Reals as Floating Point Numbers

- There are ***many ways*** to represent real numbers .

- The ***most common*** way is given by the ***IEEE standard floating point number system***
  - We need to understand the ***essentials*** of this representation to ***avoid catastrophic errors*** in numerical computations

Essentials: Real Numbers

- Any ***real number*** can be represented, to any degree of accuracy, by an ***infinite sequence of digits***

- For example, a real number ***x*** can be represented as

$$
\begin{aligned}
x &= \pm d_0.d_1 d_2 \cdots \times \beta^e \\
&= \pm \left(d_0 \beta^0 + d_1 \beta^{-1} + d_2 \beta^{-2} + \cdots \right) \beta^e
\end{aligned}
$$

where

- d_i are the *digits* $(0 \leq d_i < \beta)$
- β is the *base* or radix
- e is the *exponent*
- $d_0.d_1 d_2 \ldots$ is the *mantissa* or *significand*
- $.d_1 d_2 \ldots$ is the *fraction*

Floating Point Representation

- ***In a computer*** real numbers are represented as ***floating point numbers*** , using only use a ***finite number of digits*** (e.g., bits in base 2)
- Example: a *real number x* is (approximately) represented as a *floating point number* $\mathrm{fl}(x)$ as

$$\begin{aligned}\mathrm{fl}(x) &= \pm d_0.d_1d_2\cdots d_{p-1}\times\beta^e\\ &= \pm\left(d_0\beta^0+d_1\beta^{-1}+d_2\beta^{-2}+\cdots+d_p\beta^{-(p-1)}\right)\beta^e\end{aligned}$$

where

- β is the *base* or radix
- p is the *precision*
- $e\in[L,U]$ is the *exponent* with
- $[L,U]$ is the *exponent range*

Floating Point Representation

Engineering Significance:

Only some real numbers can be represented exactly

Example:

$$
\begin{aligned}
2.125 &= 2^1 + 2^{-3} \\
&= (1 + \frac{0}{2} + \frac{0}{2^2} + \frac{1}{2^3})2^1 \\
&= 1.00100 \times 2^1 \\
3.1 &= 2^1 + 2^0 + 2^{-4} + 2^{-5} + \cdots \\
&\approx (1 + 2^{-1} + 2^{-5} + 2^{-6})2^1 \\
&\approx 1.100011 \times 2^1
\end{aligned}
$$

Q: We can now see where the ***round-off error*** comes from

IEEE Standard

- ***Up to the mid 80s*** it was up to the computer manufacturer to choose a floating point system
  - You ***never knew*** if your program will give the ***same answer*** in ***different computers !***

- The ***IEEE standard floating-point*** arithmetic standard was introduced in **1987 (IEEE 754)** and updated in 2008 (IEEE 754-2008)

Thanks to William Kahan (Turing Award winner 1989)
http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html

Floating Point Systems

- A floating-point system is defined by the parameters $\boldsymbol{\beta, p, L, U,}$ in the floating point representation

$$
\begin{aligned}
\mathrm{fl}(x) &= \pm d_0.d_1 d_2 \cdots d_{p-1} \times \beta^e \\
&= \pm \left(d_0 \beta^0 + d_1 \beta^{-1} + d_2 \beta^{-2} + \cdots + d_p \beta^{-(p-1)} \right) \beta^e
\end{aligned}
$$

where

- β is the *base* or radix
- p is the *precision*
- $e \in [L, U]$ is the *exponent* with
- $[L, U]$ is the *exponent range*

- A floating point system is ***described by four numbers*** $\boldsymbol{\beta, p, L, U,}$ and is denoted $\boldsymbol{F(\beta, p, L, U)}$

Example: IEEE single precision: $F(2, 24, -126, 127)$

IEEE standard "`float`"

- **Single Precision** (**`float`**): **32 bits** word

s	$e = $ 8-bit exponent	$f = $ 23-bit fraction

$$(\beta = 2, p = 23 + 1, L = -126, U = 127)$$

- s is the ***sign bit***.
- The ***exponent***, e , is represented by *8 bits* but only 7 bits are used to represent it value.
- The ***fraction***, ***f*** , (or ***significand***) is represented by *23 bits*

Comments:

- *A single precision IEEE standard floating point number requires 32 bits*
- *The IEEE standard defines how to perform arithmetic, rounding, etc.*

A Sample of Floating-Point Systems

Parameters for typical floating-point systems

system	β	p	L	U
IEEE Single Prec.	2	24	−126	127
IEEE Double Prec.	2	53	−1022	1023
Cray 2	2	48	−16383	16384
HP calculator	10	12	−499	499
IBM “mainframe”	16	6	−64	63

- Today most ***computers use β=2*** (binary)

Note: The **IEEE floating-point system** (**Standard IEEE 754**) is the most **widely used today.**

Normalization

- **The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.**
- **Not only do these synonymous representations waste space, but they can also cause confusion.**

- **Does value #1 equal #2?**

0 00110 10000000 =? 0 00111 01000000

0	0 0 1 1 0	1 0 0 0 0 0 0 0	0.1×2^6
0	0 0 1 1 1	0 1 0 0 0 0 0 0	0.01×2^7
0	0 1 0 0 0	0 0 1 0 0 0 0 0	0.001×2^8
0	0 1 0 0 1	0 0 0 1 0 0 0 0	0.0001×2^9

ALWAYS NORMALIZE!!!

Zero is a special case = exp=0

Normalization

- There are many possible representations of a number: 100 = 10.0e1 = 1.00e2 = 0.100e3 = 0.01e4 = ...

- A floating-point system is ***normalized*** if the ***leading digit $d_0 \neq 0$***
  - the only exception is the number zero !

- Normalization is used make the representation of each floating point number:
  - ***Unique***
  - ***Efficient*** (No digits wasted on leading zeros)

Normalized FP Numbers in Base 2

- In binary systems the leading bit d_0 (of normalized floating point numbers) ***does not have to be stored*** since it is always 1,

$$d_0 \neq 0 \Leftrightarrow d_0 = 1$$

This is why d_0 is often called the ***phantom bit***

Consequence: Any binary ***normalized floating point number*** has a representation of the form

$$\mathrm{fl}(x) = \pm 1.d_1 d_2 \cdots d_{p-1} \times 2^e$$

- Caveat:
  - ***0 cannot be represented as a normalized number !***

  (the IEEE has provisions for this …)

IEEE Implied 1 Format

- Encoding
  - The real number is normalize to 1.yyyy x 2^{exp} format
  - The “1.” is stripped off and NOT STORED!

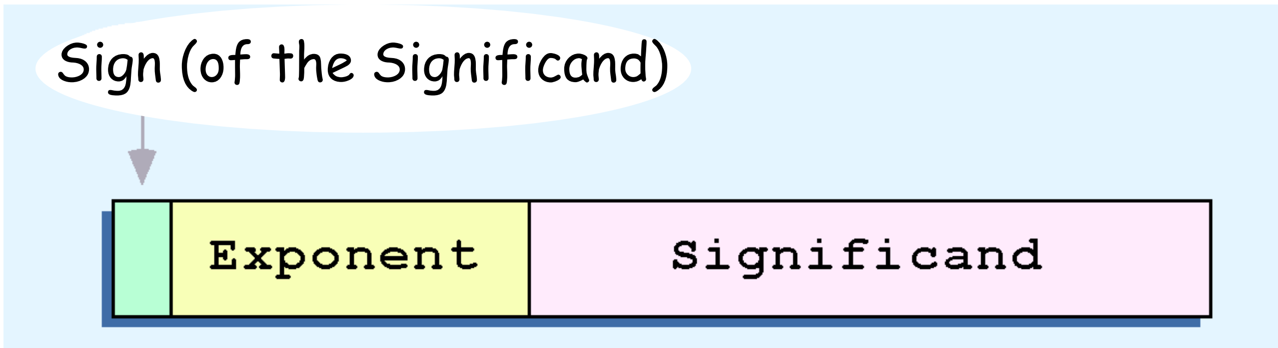
- Decoding
  - A “1.” is preppended
  - The resulting number is then decoded

Note: IEEE has TWO zero’s (+0.0 and -0.0)

- Never compare floating point numbers to zero!

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

Negative Exponents



- The Sign field is the sign of the significant NOT the exponent
  - We have no way to express 0.5 ($=2^{-1}$)!

- "Bias" addresses this issue

Bias

- A bias is a number that is approximately midway in the range of possible exponent values
  - Add constant to create a bias
  - Subtract a constant to determine the true value.

- In the IEEE Single precision case
  - We have a 8-bit exponent, we will use 2**7 = 127 for our bias
- IEEE Double precision uses 1023

Bias

- Exponent values less than 127 are negative
  - Representing fractional numbers
- Exponent values greater than 127 are positive
  - Representing larger numbers

BiasValue = realExponent + bias

realExponent = BiasValue – bias

Example: IEEE single precision

- IEEE single precision system

$$F(\beta, p, L, U) = F(2, 24, -126, 128)$$

- Example: **-0.453125 = (-0.453123**×**4)**2^{-2} **= - 1.8125**×2^{-2}

bias, phantom bit

$$-1 \times 2^{\overbrace{125 - 127}^{\text{bias}}} \times \overbrace{1}^{\text{phantom bit}}.8125 = -0.453125$$

- Storage in Memory (a `float` in C)

s	$e =$ 8-bit exponent	$f =$ 23-bit fraction

sign bit	exponent	significand
1	0 1 1 1 1 1 0 1	1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-1	125	1/2 + 1/4 + 1/16 = 0.8125

IEEE-754 Special Values

Value	Sign	Exponent	Fraction	Comments
+0	0	00000000	00000000…	**Positive 0** (zero) all exponent bits set to 0 all fraction bits set to 0
-0	1	00000000	00000000…	**Negative 0 - is this a problem?**
+ Inf	0	11111111	00000000…	Positive infinity all exponent bits set to 1 all fraction bits set to 0
- Inf	1	11111111	00000000…	Negative infinity
NaN	0/1	11111111	1000000… 01000000… Etc…	not a number all exponent bits set to 1 not all fraction bits are set to 0 sign bit is not used in **NaN**

Notice how an all “1” exponent is a special case, so it can’t be use in normal calculations.

http://www.ajdesigner.com/fl_ieee_754_word/ieee_32_bit_word.php

IEEE single precision (`float`)

- Single Precision: 32 bits word

s	$e = $ 8-bit exponent	$f = $ 23-bit fraction

$$(\beta = 2, p = 23 + 1, L = -126, U = 127)$$

- There are *8 bits (2^8-1=255)* for the exponent but ***only 7 bits are used*** to represent ***normalized numbers*** (where not all exponent bits can be 0)
- The actual ***exponent*** value is ***biased by -127*** (this explains why *L =-126* and *U=127*)
- The Numerical Value of a single precision floating point number represented as above is

$$\boxed{fl(x) = (-1)^s \times 2^{e-127} \times 1.f}$$

IEEE double precision (`double`)

- Double Precision: 64 bits word

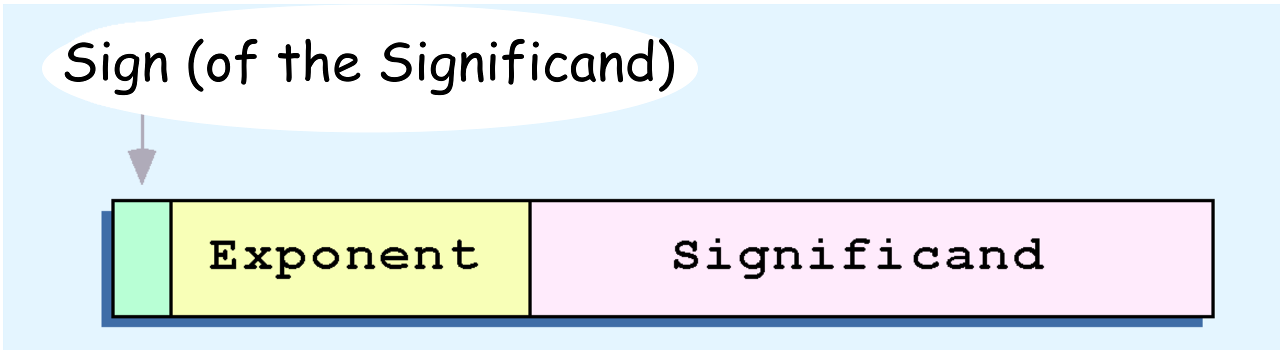
s	$e =$ 11-bit exponent	$f =$ 52-bit fraction

$$(\beta = 2, p = 52 + 1, L = -1022, U = 1023)$$

- There are *11 bits* for the exponent but ***only 10 bits are used*** to represent ***normalized numbers*** (where not all exponent bits can be 0)
- The actual ***exponent*** value is ***biased by -1023*** (this explains why *L=1022* and *U=1023*)
- The Numerical Value of a single precision floating point number represented as above is

$$\boxed{fl(x) = (-1)^s \times 2^{e-1023} \times 1.f}$$

Floating-Point Representation



- IEEE-754 single precision floating point standard uses an 8-bit exponent, a 23-bit significand and a bias of 127.

- The IEEE-754 double precision standard uses an 11-bit exponent, a 52-bit significand and a bias of 1023

Fraction Conversion Reminder

- Convert 0.8125 to fractional binary

0.8125		
2		
1.6250	extract the integer	1
0.6250		
2		
1.2500	extract the integer	1
0.2500		
2		
0.5000	extract the integer	0
2		
1.0000	extract the integer	1

1) Multiply fraction by 2
2) Record the integer value
3) Subtract the integer value
4) Continue

The integer values are the fractional coefficients

- Result 0.1101_2

Class Exercise

- Express 18.8125 in IEEE Single precision floating point.
  - Convert 18 = 10010
  - Convert .8125 = .1101
- 18.8125 = $10010.1101_2 \times 2^{0}$
  $= 1.00101101 \times 2^{4}$
  - Exponent = 4+127 = 131 = 10000011_2
  - Sign = 0 (positive)

- 18.8125 = 0 10000011 00101101000000000000000

Floating-Point Numbers

- A floating point system $\boldsymbol{F}(\beta, p, L, U)$ ***can only represent*** a *total of*

$$\#\boldsymbol{F} = 2(\beta - 1)\beta^{p-1}(U - L + 1) + 1 \quad \textit{different numbers}$$

(for binary base $\#\boldsymbol{F} = 2^{p}(U - (L-1)) + 1$)

Example: The FP system $\boldsymbol{F}(2, 3, -1, 1)$ can only represent

$$\#\boldsymbol{F} = 2^3(1 - (-1) + 1) + 1 = 25 \text{ numbers}$$

(we have used $\beta = 2$, $p = 3$, $L = -1$, and $U = 1$)

Note: Only some *real numbers* can be *represented exactly* as floating point numbers. These are called *machine numbers*; all other numbers can be only approximated.

Comparing Floating Point Numbers

- The main source of round-off error arises from numbers that are not representable
  - ***most real numbers are not representable!,*** e.g. 1/10

Consequence:

- Results may appear (to the uninitiated) non intuitive

```
if (0.1 + 0.2 == 0.3) /* false */
if (0.1 + 0.3 == 0.4) /* true */
```

- Never compare two floating point numbers with logical operators (what is **ZERO** ?)

```
(x == 0.0)           /* incorrect */

(fabs(x) < ZERO)      /* better    */

(y == z)             /* incorrect */
(fabs(y-z) < ZERO     /* better    */
```

http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Proof

```
int main()  /* compares.c */
{
   /* Create a union to "see"
   union {
      double f;
      long   i;
      } a,b,c,d,e,g;
    /* Initialize */
    a.f = 0.1l; b.f = 0.2l; c.f = 0.3l; d.f = 0.4l;
   e.f = a.f+b.f;
   printf("The value %0.2lf+%0.2lf gives %0.2lf 0x%16lx, does it equal %0.2lf 0x%16lx?\n",
                  a.f, b.f,    e.f, e.i,                c.f, c.i);
   g.f = a.f+c.f;
   printf("The value %0.2lf+%0.2lf gives %0.2lf 0x%16lx, does it equal %0.2lf 0x%16lx?\n",
                  a.f, c.f,    g.f, g.i,                d.f, d.i);
 if ( e.f == c.f) {printf("Equal %lf == %lf\n", e.f,c.f);}
   else          {printf("Not equal %lf == %lf\n", e.f, c.f);}
   return(0); }
```

```
The value 0.10+0.20 gives 0.30 0x3fd3333333333334,
                 does it equal 0.30 0x3fd3333333333333?
The value 0.10+0.30 gives 0.40 0x3fd999999999999a,
                 does it equal 0.40 0x3fd999999999999a?
Not equal 0.300000 == 0.300000
```

Notice: printf() lies, so do debuggers!

Catastrophic Cancellation

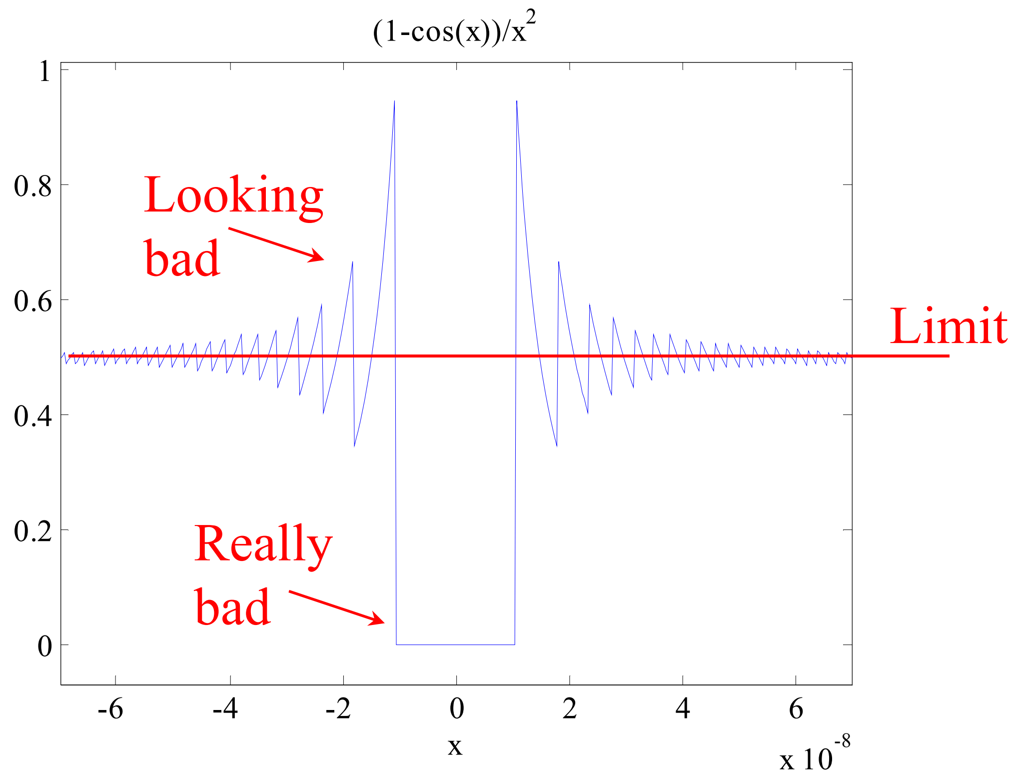
Occurs when we ***subtract*** two numbers that are **very close to each other**

- ***Example:*** Plot $f(x)$ for $x \in$ [-7E-8,7E+8]

$$f(x) = \frac{1-\cos(x)}{x^2}$$

- Using calculus

$$\lim_{x\to 0} f(x) = \left.\frac{\sin(x)}{2x}\right|_{x=0}$$
$$= \left.\frac{\cos(x)}{2}\right|_{x=0}$$
$$= \frac{1}{2}$$



" Floating point numbers are like piles of sand; every time you move them around, you lose a little sand and pick up a little dirt. "

— Brian Kernighan and P. J. Plauger

The Machine Precision

- ***Practical Definition***: The ***machine epsilon*** is the smallest (positive normalized) number ϵ_m such that $1.0 + \epsilon \neq 1.0$
- The following C code illustrates how to ***estimate the single precision machine epsilon*** (within a factor of two)

```
/* Estimate, within a factor of two, of the single *
 * precision machine epsilon        machine_eps.c  */
#include <stdio.h>
int main( int argc, char **argv ) {
 float eps;
 eps = 1.0f;         /* start binary search from eps=1.
 do {
   printf("%10.8g\t%.20f\n",eps,(1.0f + eps));
   eps /= 2.0f; /* divide eps by 2 */
 } while ( (float)(1.0f + (eps/2.0f)) != 1.0f );
/* stop when 1 + eps != 1 */
 printf("\n");
 printf("Calculated Machine Epsilon: %2.6g\n", eps );
 return 0;
} /* exact value given by FLT_EPSILON in float.h */
```

```
current Epsilon, 1 + current Epsilon
         1      2.00000000000000000000
       0.5      1.50000000000000000000
      0.25      1.25000000000000000000
…
9.5367432E-07   1.00000095367431640625
4.7683716E-07   1.00000047683715820312
2.3841858E-07   1.00000023841857910156
```

Calculated Machine epsilon: 1.19209E-07

Significance of Machine Epsilon

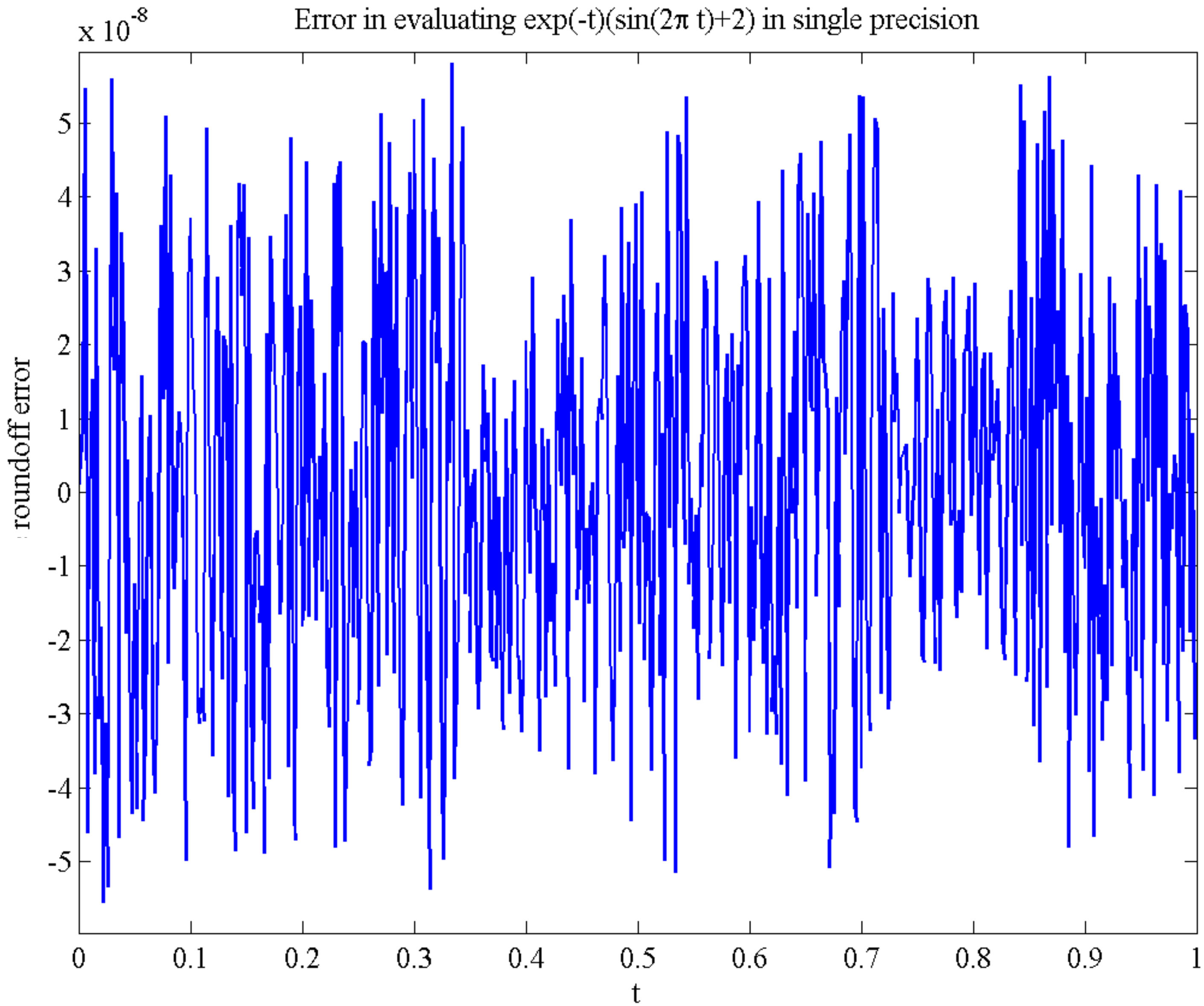
- Let $\mathrm{fl}(x) = 1.d_1 \ldots d_{p-1} \times \beta^e$ be the IEEE floating point representation of a real number ***x*** with precision *p*
- The ***machine epsilon*** quantifies the magnitude of the ***largest relative error*** incurred in computing with floating point numbers
- The *maximum relative error* in representing a real number *x* using floating point fl*(x)* is *bounded by*

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \epsilon_m$$

- The *maximum absolute error* in representing a real number *x* using floating point fl*(x)* is *bounded by*

$$|\mathrm{fl}(x) - x| \leq \epsilon_m \beta^e$$

Almost Random Nature of Round-off



Epilogue: The Machine Epsilon

- The ***machine epsilon*** ϵ_m ***(eps)*** is used to characterize the ***accuracy of floating-point arithmetic.*** It is also called the ***rounding unit*** or ***machine accuracy*** (denoted by η in the textbook)

- For a floating point system of *base* β *and p digits of precision* the machine epsilon can be computed exactly as

$$\boxed{\epsilon_m = \beta^{-p} = 1/\beta^p}$$

(when using the standard rounding to nearest)

Engineering Significance:

*The **accuracy** of floating point computations increases with the **number of digits of precision***

Summary: Floating Point Numbers

- The ***IEEE-754 floating-point*** system is the most commonly used in numerical computing.
- It has the following characteristics:
  - **Single precision** (C **`float`**) [32 bits]
    - machine epsilon (eps): $\mathbf{2^{-23}} = 1.192\mathrm{E}{-7} \approx \mathbf{10^{-7}}$
    - decimal digits of precision: **7**
  - **Double precision** (C **`double`**) [64 bits]
    - machine epsilon (eps): $\mathbf{2^{-53}} = 1.110\mathrm{E}{-16} \approx \mathbf{10^{-16}}$
    - decimal digits of precision: **16**
- To avoid large round-off errors it is important to understand the floating point system used in your computing environment and how this error propagates.

Applied Programming

Computing with Numbers of

Finite Word length

Fixed-Point Representations

(Q-format)

Embedded Systems

- Computers *use a* *finite* *number of bits* to represent *infinitely* *many* numbers.
- **Floating Point Numbers** is a way to represent real numbers
  - the binary point is variable (e.g., "floating") as it depends on the value of the **exponent**.
- ***High-end processors*** perform floating-point arithmetic in dedicated hardware (**FPUs**)

Most ***embedded systems do not have FPUs !***

Arithmetic in Embedded Systems

- In processors that do not have ***FPUs*** we have the following alternatives:

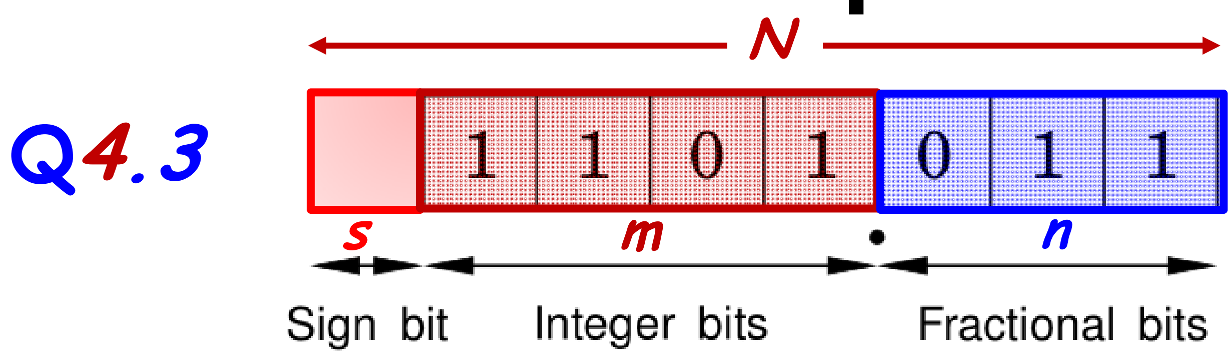
  - **Software emulation of floating-point arithmetic**
    - Large code size (memory footprint)
    - ***Sloooow*** execution

  - **Fixed-point arithmetic**
    - Uses hardware integer arithmetic
    - Compact code size (memory footprint)
    - ***Fast*** execution

Efficiency

- Floating point computations are significantly slower than fixed point and use more memory
  - More memory means more memory access
- ***High performance*** software is often implemented using ***fixed point*** arithmetic;
  - **Example: video codecs.**

Fixed-Point Representation



- A real number ***x*** can be represented by an ***N***-bit integer with ***n*** implicit fractional bits, where
  - ***N*** is the word length ($N=m+n+1$)
  - ***m*** is the number of ***integer*** bits
  - ***n*** is the number of ***fractional*** bits
  - ***s*** is the sign bit

Q*m*.*n* representation called ***the Q-format***

Some Q-Format Applications

- VisSim—simulation and system design software
  - http://www.vissim.com/
- GnuCash—open-source accounting software
  - http://www.gnucash.org/
- Tremor— Ogg Vorbis **decoder**
  - http://en.wikipedia.org/wiki/Tremor_%28software%29
- Sony's original PlayStation **3-D graphics engine**
  - http://us.playstation.com/ps3/
- Nintendo DS (2-D and 3-D) **game system**
  - http://www.nintendo.com/3ds

Source: "Fixed-point arithmetic," Wikipedia, http://en.wikipedia.org/wiki/Fixed-point_arithmetic, Oct. 5, 2006.

Q-format Range and Resolution

- **Q*m*.*n*** denotes a ***fixed point number*** such that:
  - ***n*** bits represent the ***fractional part***
  - ***m*** bits represent the ***unsigned integer part***, (excluding the MSB, *e.g.*, "sign bit")

http://www.exploringbinary.com/twos-complement-converter

Characteristics of a **Q*m*.*n*** fixed point number:

- Requires $N=m+n+1$ bits (N usually **8,16,23,64, …**)
- Its ***range*** is $[-2^m, 2^m-1]$
- Its ***resolution*** (or ***quantization step***) is $Q=1/2^n$ (distance between two consecutive numbers)

Example

Find the Qm.n-format for an application that uses numbers in the range 0 to 100 with resolution 0.01

- Unsigned integer
- Maximum number $M = 100$ (absolute value)

- Determine ***m*** *(e.g. what power of 2)*

$$M \leq 2^m - 1 \Rightarrow m = \lceil \log_2(M+1) \rceil = 7$$

- Determine ***n*** *(e.g. what power of 2)*

$$r \leq 2^{-n} \Rightarrow n = \lceil -\log_2(r) \rceil = 7$$

- Conclusion: Need ***Q7.7 (e.g., N=16*** ***(~=7+7+1))***

The Q-format

- In practice *N* **(**the **word length)** is chosen equal to the ***size of primitive integer types***,
  - *e.g.*, **8, 16, 32, 64** bits.

- When the size of the word length is clear from the context the ***m*** in **Q*m*.*n*** is dropped we just use **Q*n***
  - If *N*=8, Q3.4 becomes Q4
  - If *N*=16, Q7.8 becomes Q8
  - If *N*=32, Q15.16 becomes Q16

Ambiguous Specifications

- ***Qn* format (*N* implicit)**
  - Assume data types: **integer** (char, short, long, etc.)

  - Interpretation: divide by $\mathbf{2}^{n}$
    - Q*6*
    - Q*15*

    $\Big\}$ Ambiguous without specification of word length *N*
  - Choice of word length (*N-bits*)
    - Based on desired range and resolution (or granularity)

- **Full Q*m*.*n*** is always clear

Q-format: Example

- Write 12.25 as a Q4.3 number (N=4+3+1=8)

$$12 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$$
$$0.25 = 0 \times 2^{-1} + 1 \times 2^{-2}$$

Ans: 01100010 (as a signed 8-bit integer it is 98)

(Note that the fixed point is implicit by the representation)

- Write -12.25 as a Q4.3 number
  - Find 2's complement of above

$$\begin{array}{l} 01100010 \\ 10011101 \text{(flip bits)} \\ \quad\quad +1 \text{(add 1)} \\ 10011110 \end{array}$$

Ans: 10011110

(as a signed 8-bit integer it is -98)

Real to Fixed point

- From ***real*** number ***r*** to a Q*m*.*n* ***fixed-point*** ***F***

  $F = \text{round}(\mathbf{r} * 2\text{^}n)$ (round to the nearest integer)

Example: *Represent* ***x=13.4*** *using Q4.3 format*

$$F = \text{round}(\mathbf{13.4} * 2\text{^}\mathbf{3})$$

8 bits

$$F = \text{round}(107.2) = 107 = 01101011$$

Key Observation: $X=107=01101011_2$ is an integer that when interpreted as a *Q4.3* number becomes

$$01101.011_2 = 13.375$$

* Since the resolution is $2^{-3}=0.125$ we ***cannot represent exactly 13.4*** (it is not representable)

Fixed Point to Real

- From *Qm.n* ***fixed-point F to real*** number ***r***

  $R = F * 2$^-n or `(float)X / (1<<n)`

Example: *Convert the Q4.3 format number $X=107=01101011_2$ to real*

$$x = 107 \cdot 2^{-3}$$
$$= 107/8 = 13.375$$

Key Observation: We have used "floating point" division (as opposed to integer)

- Should be: (float)107/(float) 8.0

Floating-Point to Fixed-Point code

- Convert:
  - C code example, (ansi C89)

```
#define n 3          /* type Q4.3         */
typedef _int8 fixn  /* wordlength 8 bits */
typedef float real  /* single precision  */
real x;
fixn X;
X = (fixn) ( x * (real) (1 << n) );
```

  - C macro example

```
#define FLOAT_TO_FIX(x,n) \
    ( (fixn) (x * (real) (1 << n)) )
/* usage */
X = FLOAT_TO_FIX(x,n)
```

Fixed-Point to Floating-Point code

- Convert:
  - C code example

```
#define n 3
typedef int fixn /* need to know word length of int*/
typedef float real
real x;
fixn X;
x = (real) X / (real) (1 << n));
```

  - C macro example

```
#define FIX_TO_FLOAT(X,n) ((real) (X) / (real) (1 << n))


x = FIX_TO_FLOAT(X,n)
```

Q-numbers are Scaled Integers

- ***Fixed-point numbers*** are ***integers scaled by 2-n***
  - For the computer they are *"just integers"*
  - We are the only ones that are aware of the Q format

- What it the impact on CPU calculations?
  - Addition/subtraction
  - Multiplication
  - Division

Q-Format Addition/Subtraction

- Two fixed-point numbers ***X,Y*** in the **same** ***Qm.n*** **format** can be added or subtracted directly
  - because for the computer ***X,Y*** are just integers

$$x, y \in \mathbb{R}, \quad X, Y \in Qm.n$$
$$z = x + y \Rightarrow Z = X \pm Y$$

Warning

The result ***Z*** will have the ***same number of fractional bits*** but the ***integer part*** may require ***m+1*** bits (e.g., ***may overflow***)

$$Z \in Q(m+1).n$$

Fixed-Point Addition w/o Overflow
N=16; n=6

Add 64.125 and -.75 using fixed point Q9.6 numbers

(don’t forget to represent negative numbers using 2’s complement)

```
 Binary               Integer                Q9.6
 1000000.001          0001000000001000       0001000000001000
+       -.110      +(-0000000000110000)     +1111111111010000
-------------     --------------------     -----------------
  111111.011                                 0000111111011000

Decimal 64.125
      +   -.75
     ---------
        63.375
```

Example Addition with Overflow

- Add two numbers in *Q4.3* format (*N*=*4*+*3*+*1*=*8*)

$$x = 12.25 \xrightarrow{Q4.3} X = 98 = 1100010_2$$

$$y = 14.75 \xrightarrow{Q4.3} Y = 118 = 1110110_2$$

$$\begin{array}{c|c|c|c|c|c|c|c|c}
 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & \bullet & & & \\
+ & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
 & & & & & \bullet & & & \\
\hline
= & \boxed{1} & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
 & & & & & \bullet & & &
\end{array}$$

Example Addition with Overflow

	0	1	1	0	0	0	1	0
+	0	1	1	1	0	1	1	0
=	$\boxed{1}$	1	0	1	1	0	0	0

- Their sum is ***out of range (overflow)***

$$Z = X + Y = 216 = 11011000_2$$

- The ***result*** will be ***interpreted as***

$$216 - 2^8 = -40 \Rightarrow z = -5.0$$

Q-Format Addition/Subtraction

- Add/Subtract two ***Qm.n*** numbers
  - ignoring overflow

$$S = A \pm B$$

  - C code example

```
typedef int fixn
fixn  A, B, S;
S = ADD_FIX(A,B);
S = SUB_FIX(A,B);
```

  - C macros example

```
#define ADD_FIX(X,Y) ( (X) + (Y) )
#define SUB_FIX(X,Y) ( (X) - (Y) )
```

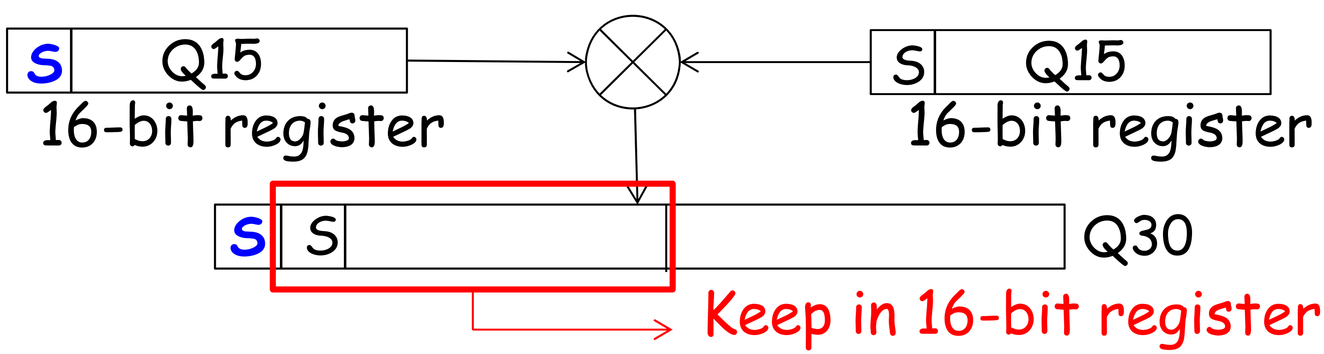
Q-Format Multiplication/Division

- Product of $Qm_1.n_1$ and $Qm_2.n_2$ formats results in $Q(m_1+m_2).(n_1+n_2)$ format

Example:

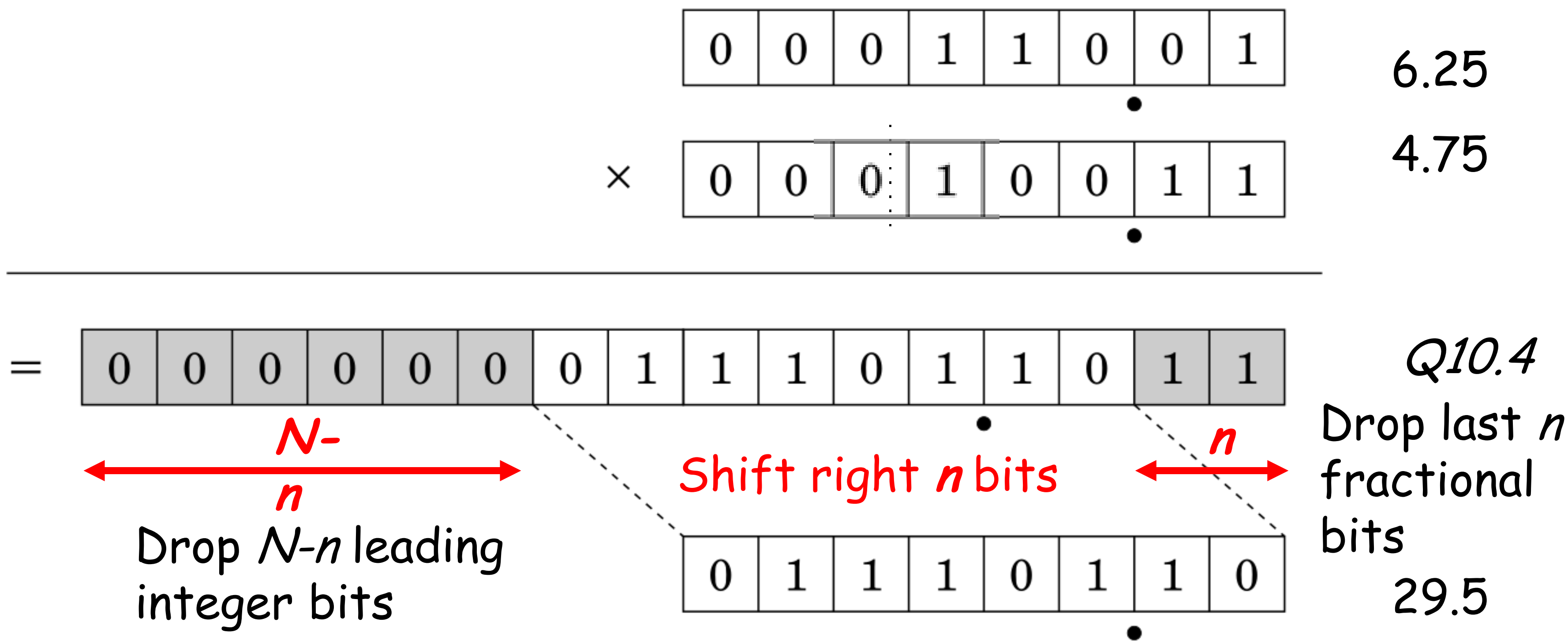
```
     1.11  (Q1.2 format)
x    1.11  (Q1.2 format)
-----------
      111
     111
    111
-----------
  11.0001  (Q2.4 format)
```

- There is a sign bit and an **added MSB-sign** extension bit.



Multiplication w/o Overflow

- Multiplication of two *Q5.2* numbers (N=8,n=2)



Q-Format Multiplication

- If the there is enough precision
- Multiply two ***Qm.n*** numbers: $C = A\ B$
  - C code

```
typedef int fixn
fixn  A, B, C;
C = (A * B) >> n;
```

  - C macro

```
#define MUL_FIX(A,B,n) ((A) * (B) >> n)
```

Q-Format Multiplication

If there is NOT enough precision

Problem: ***Qn*** word size (*N*) is largest machine word

- Multiplication may result in *overflow*
- Perform *half-n shift* of each factor first
  - Less accurate than first version (if useable)

Solution: C macro

```
#define Qn_MULTIPLY(A,B) ((A>>(n/2))*(B >>(n-(n/2))))
```

Q-Format Division

- Divide two ***Qn*** numbers: $C = A \div B$
  - C code

```
typedef int fixn
typedef long int fix2n
fixn  A, B, C;
C = (A << n) / B;
```

  - C macro

```
#define DIV_FIX(A,B,n) (( (A) << n)/(B) )
```

Problem:

Dividend must be ***shifted left n*** bits ***before division***

Solution: Typecast ***intermediate result to larger word***

C macro example (`fix2n` twice as large as `fixn`)

```
#define Qn_DIVIDE(A,B,n) ((fixn) (((fix2n)(A)<<n)/(B) ))
```

Fixed-Point Multiplication Example
N=16; q=6

Multiply -.75 by 64.125 using fixed-point Q9.6 numbers

Binary		Q6 (N16)
-.11	(-0000000000110000)	1111111111010000
x 1000000.001	x 0001000000001000	x0001000000001000
-------------	--------------------	------------------

Sign-extend to 2*N* bits

1111111111111111111111010000
1111111111111010000

Q19.12 (N32) ~~1111111111~~$\boxed{\texttt{1111001111111010}}$~~000000~~

Drop *N*-*n* (10) bits on left

Drop *n* (6) bits on right

Decimal

-48.09375 ← -(0000110000000110) ← (1111001111111010)

Fixed-Point Division Example
N=8; Q=5.2

Divide 6.25 by 4.75 using fixed-point Q5.2 numbers

```
  Binary
  6.25 =00011001 →  0001100100 (preshift n =2)
÷ 4.75 =00010011
------
  1.32             0001100100/00010011 = 00000101

                   00001.01  = 1.25 ~= 1.32
```

Q*m.n* Format Highlights

- Represents real numbers as ***fixed-point numbers***
- Uses ***hardware integer arithmetic***
- Resolution: 2^{-n}
  - *In practice the desired resolution is "known" (part of the design specs.), it determines **n***
- Range: $[-2^m, 2^m - -1]$, $m=N-n-1$
  - *In practice one needs to estimate the range carefully (signal scaling). This (and the machine) determines the word-length **N***

Resolution and Range fundamental to determine a suitable ***Qm.n*** format for a given application

Q*m*.*n* Summary

- Addition/subtraction just works
- Multiplication
  - Cast variables to the next larger size
  - Shift answer >> *n*
  - Recast to normal size
- Division
  - Cast dividend to the next larger size
  - Shift by dividend << *n* before calculation

Exercise 1

- The following is an IEEE single precision number (8, 23, 127): 0x41968000
  - What is the decimal value?

Convert to binary, separate the fields:

0 10000011 00101101000000000000000

Sign = 0 (positive)

Exp = 10000011 -> 131-127 = 4 (real exp)

Sig = 1.00101101 (put the implied 1 back)

value = $1.00101101 \times 2^4 = 10010.1101 \times 2^0$

value = 18.8125

Exercise 2

Find the Qm.n-format for an application that uses numbers in the range 10,002 to -1,022 with a resolution 0.1

- Determine ***m*** *(e.g. what power of 2)*

  $10002 <= 2^m - 1 \Rightarrow 2^{**}14 = 16{,}384$ so $m = 14$

- Determine ***n*** *(e.g. what power of 2)*

  $0.1 <= 2^{-n} \Rightarrow 0.0625 = 2^{**}(-4)$ so $n = 4$

- Need ***Q14.4 e.g., N=19 (use a 32 integer)***

Exercise 3

Using Q3.4 numbers, divide 3.6 by 2.

- 2**4 = 16, so 3.6*16 = 57.6 => 00111001 (57 dec)
  and 2 => 00100000 (32 dec)

- Divide (preshift n)
  001110010000/00100000 = 00011100 (28 dec) =>1.75

Note: 00111001 >> 1 (shift division) or
00111001/10 (binary division) = 00011100 => 1.75

Gave the right answer because we are really treating 2 as a Qn 2.0 number.