

Floating Point Operations and SIMD Extensions

Advanced Topics Spring 2008

Prof. Robert van Engelen





Overview

- IEEE 754 Floating point
- IEEE 754 Exceptions
- FPU control and status registers
- Language and compiler issues with IEEE floating point
- Tricks
- Error analysis
- SIMD short vector extensions
- Programming with SIMD extensions
- GNU multi-precision library (GMP)



Floating Point

■ Definitions

- Notation: $s.d.dd\dots d \times r^e$
- Sign: s (+ or -)
- Significand: $d.dd\dots d$ with p digits (precision p)
- Radix: r (typically 2 or 10)
- Signed exponent: e where $e_{\min} \leq e \leq e_{\max}$

■ Represents a floating point number (a rational value)

$$\pm (d_0 + d_1 r^1 + d_2 r^2 + \dots + d_{p-1} r^{(p-1)}) r^e$$

where $0 \leq d_i < r$



IEEE 754 Floating Point

- The IEEE 754 standard species
 - Binary floating point format ($r = 2$)
 - Single, double, extended, and double extended precision
 - Representations for indefinite values (NaN) and infinity (INF)
 - Signed zero and denormalized numbers
 - Masked exceptions
 - Roundoff control
 - Specifies algorithms for arithmetic to ensure accuracy and bit-precise portability
- Note:
 - Programs that rely on IEEE 754 **may not** be bit-precise portable, because math function libraries are not identical across systems



IEEE 754 Floating Point Formats

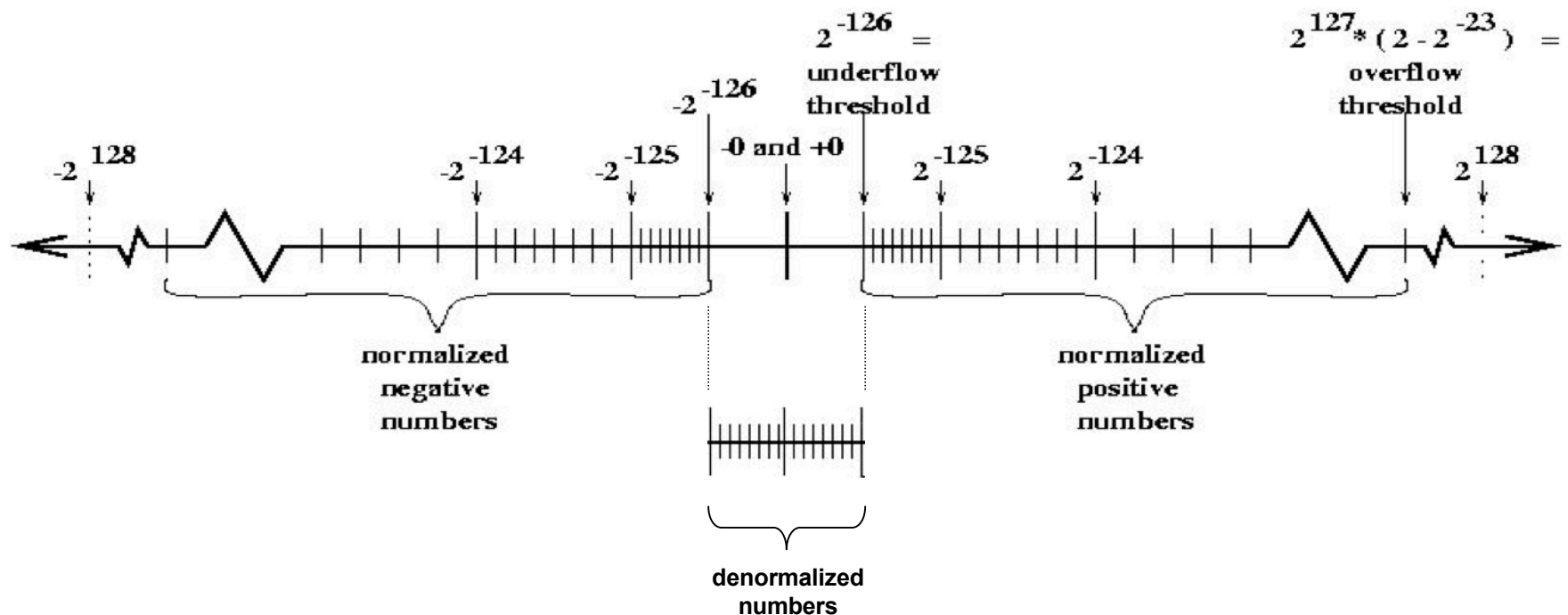
- Four formats:

Parameter	Format			
	Single	Single-Extended	Double	Double-Extended
p	24	≥ 32	53	≥ 64
$e_{\max}$	+127	$\geq +1023$	+1023	≥ 16383
$e_{\min}$	-126	≤ -1022	-1022	≤ 16382
Exponent width	8 bits	≥ 11 bits	11 bits	≥ 15 bits
Format width	32 bits	≥ 43 bits	64 bits	≥ 79 bits



IEEE 754 Floating Point

- Most significant bit of the significand d_0 not stored
- **Normalized numbers:** $\pm 1.dd\dots d 2^e$
- **Denormalized numbers:** $\pm 0.dd\dots d 2^{emin-1}$





IEEE 754 Floating Point

Overflow and Underflow

- Arithmetic operations can **overflow** or **underflow**
- Overflow: result value requires $e > e_{\max}$
 - Raise exception or return $\pm\text{infinity}$
 - Infinity (INF) represented by zero significand and $e = e_{\max} + 1$
- Underflow: result value requires $e < e_{\min}$
 - Raise exception or return denorm or return **signed** zero
 - Denorm represented by with $e = e_{\min} - 1$
- Why bother returning a denorm?

`if (a != b) then x = a / (a - b);`

- Why bother distinguishing +0 from -0?

`if (a > b) then x = log(a - b);`



IEEE 754 Floating Point NaN

- **Not-a-number** (NaN) represented by all 1 bits in exponent $e = e_{\max} + 1$ (e is **biased** by $+2^{\text{exp_width}-1}-1$)
- Sign and significand > 0 are irrelevant (but may carry info)
- Generated by indeterminate and other operations
 - $0/0$
 - $\text{sqrt}(-1)$
 - $\text{INF}-\text{INF}$, INF/INF , $0*\text{INF}$
- Two kinds of NaN
 - **Quiet**: propagates NaN through operations without raising exception
 - **Signaling**: raise an exception when touched
- Fortran initializes reals to NaN by default
 - Signaling NaN automatically detects uninitialized data



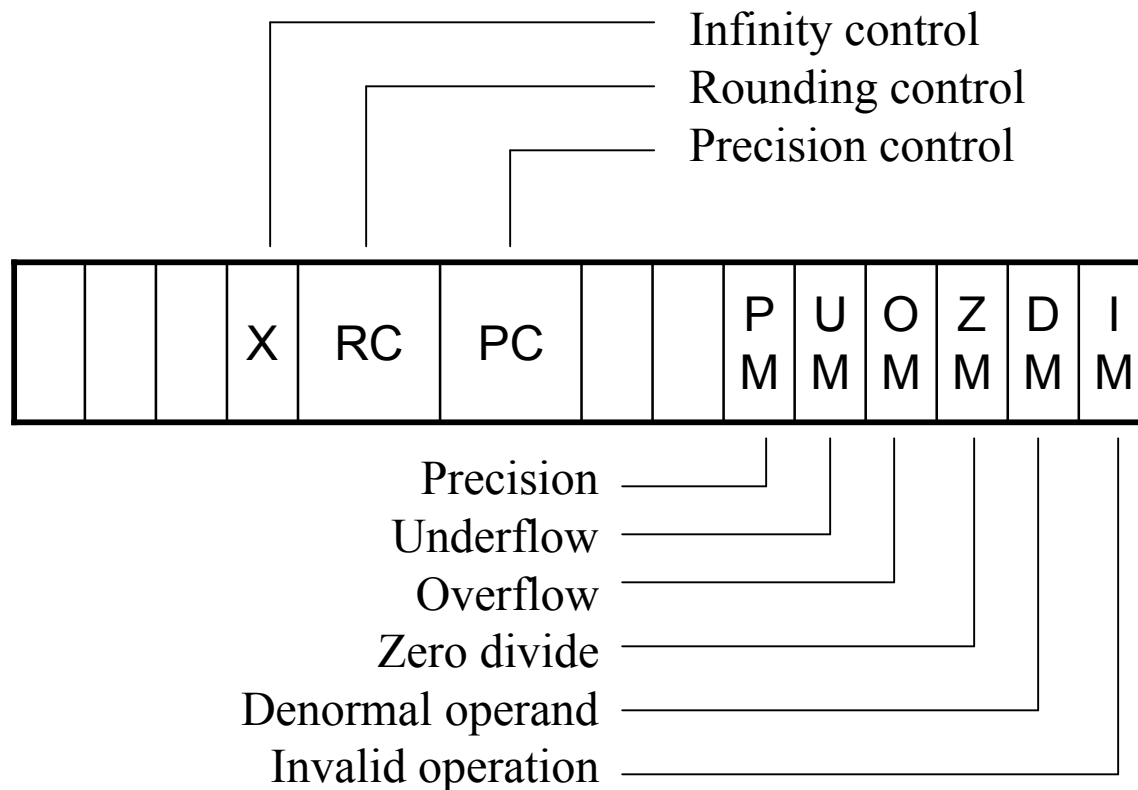
IEEE 754 Floating Point Exceptions

- Exceptions
 - ☐ **Invalid operation:** raised by a signaling NaN or illegal operation on infinity
 - ☐ **Divide by zero**
 - ☐ **Denormal operand:** indicates loss of precision
 - ☐ **Numeric overflow or underflow**
 - ☐ **Inexact result or precision:** can be safely ignored
- Exceptions can be **masked** using hardware control registers of an FPU
 - ☐ Masking means that quiet NaN and INF are returned and propagated



Intel x87 FPU FPCW

- Masking exceptions on the Intel x87 FPU using the FPCW control word



```
uint16_t setexc = ...;
uint16_t oldctrl, newctrl;
_asm {
    FSTCW oldctrl
    mov ax, oldctrl
    and ax, 0ffc0h
    or ax, setexc
    mov newctrl, ax
    FLDCW newctrl
}
```



Intel x87 FPU FPCW

- The Intel x87 FPU uses a pre-specified precision for all internal floating point operations
 - Extended double (80 bits) for Linux
 - Double (64 bits) for Windows
- Using **float** and **double** in C only affects storage, not the internal arithmetic precision
 - Changing the FPU precision can speedup div, rem, and sqrt

```
uint16_t prec = 0x0000; // 0x0000=sgl, 0x0200=dbl, 0x0300=ext
uint16_t oldctrl, newctrl;
_asm {
    FSTCW oldctrl
    mov ax, oldctrl
    and ax, 0fcffh
    or ax, prec
    mov newctrl, ax
    FLDCW newctrl
}
```



Language and Compiler Issues with IEEE Floating Point

- Associative rule does not hold: $(x + y) + z \neq x + (y + z)$
 - Take $x = 10^{30}$, $y = -10^{30}$, and $z = 1$
- Cannot replace division by multiplication: $x/10.0 \neq 0.1 * x$
 - 0.1 is not accurately represented
- Distributive rule does not hold: $x * y + x * z \neq x * (y + z)$
 - Take $y \approx -z$
- Negation is not subtraction: $-x \neq 0 - x$
 - Take $x = 0$ (recall zeros are signed)
- NaN is unordered, which affects comparisons
 - Any comparison to NaN returns false, thus when $x < \text{NaN}$ fails this does not imply $x \geq \text{NaN}$
 - Cannot sort array of floats with NaNs
 - $!(x < y)$ is not identical to $x \geq y$
 - $x == x$ is not true when $x = \text{NaN}$
- IEEE rounding modes may differ from language's rounding



Language and Compiler Issues with IEEE Floating Point (cont)

- Exceptions (e.g. signaling NaN) disallow expression optimization
 - These two instructions have no dependence and can be reordered:
 $x = y * z;$
 $a = b + c;$
 but each may trigger an exception and the reorder destroys relationship
- A change in rounding mode affects common sub-expressions
 - The expression $a * b$ is not common in this code:
 $x = a * b;$
 $\text{set_round_mode} = \text{UP};$
 $y = a * b;$
- Narrowing and widening type conversions
 - Use the type of the destination of the assignment
 - Obey type of operands, widen intermediate values when necessary, and then narrow final value to destination type
- IEEE ensures the following are valid for all values of x and y :
 $x + y = y + x$, $x + x = 2 * x$, $1.0 * x = x$, $0.5 * x = x / 2.0$



IEEE 754 Floating Point Manipulation Tricks

- Fast FP-to-integer conversion (rounds towards $-\infty$)

```
#define FLOAT_FTOI_MAGIC_NUM (float)(3<<21)
#define IT_FTOI_MAGIC_NUM (0x4ac00000)
inline int FastFloatToInt(float f)
{
    f += FLOAT_FTOI_MAGIC_NUM;
    return (*(int*)&f - IT_FTOI_MAGIC_NUM)>>1;
}
```

- Fast square root approximation with 5% error

```
inline float FastInvSqrt(float x)
{
    int t = *(int*)&x;
    t -= 0x3f800000;
    t >>= 1;
    t += 0x3f800000;
    return *(float*)&t;
}
```



Floating Point Error Analysis

■ Error analysis formula

- $fl(a \text{ op } b) = (a \text{ op } b) * (1 + \epsilon)$
- op is +, -, *, /
- $|\epsilon| \leq \text{machine eps} = 2^{\text{\#significant bits}}$ = relative error in each op
- Assumes no overflow, underflow, or divide by zero occurs
- Really a worst-case upper bound, no error cancellation

■ Example

- $$\begin{aligned} fl(x + y + z) &= fl(fl(x + y) + z) \\ &= ((x + y) * (1 + \epsilon) + z) * (1 + \epsilon) \\ &= x + 2\epsilon x + \epsilon^2 x + y + 2\epsilon y + \epsilon^2 y + z + \epsilon z \\ &\approx x * (1 + 2\epsilon) + y * (1 + 2\epsilon) + z * (1 + \epsilon) \end{aligned}$$



Numerical Stability

- **Numerical stability** is an algorithm design goal
- **Backward error analysis** is applied to determine if algorithm gives the exact result for slightly changed input values
- Extensive literature, not further discussed here...



Conditioning

- An algorithm is **well conditioned** (or **insensitive**) if relative change in input causes commensurate relative change in result

$$\begin{aligned} \text{Cond} &= | \text{relative change in solution} | / | \text{relative change in input} | \\ &= | (f(x+h) - f(x)) / f(x) | / | h/x | \end{aligned}$$

- Problem is **sensitive** or **ill-conditioned** if $\text{Cond} \gg 1$

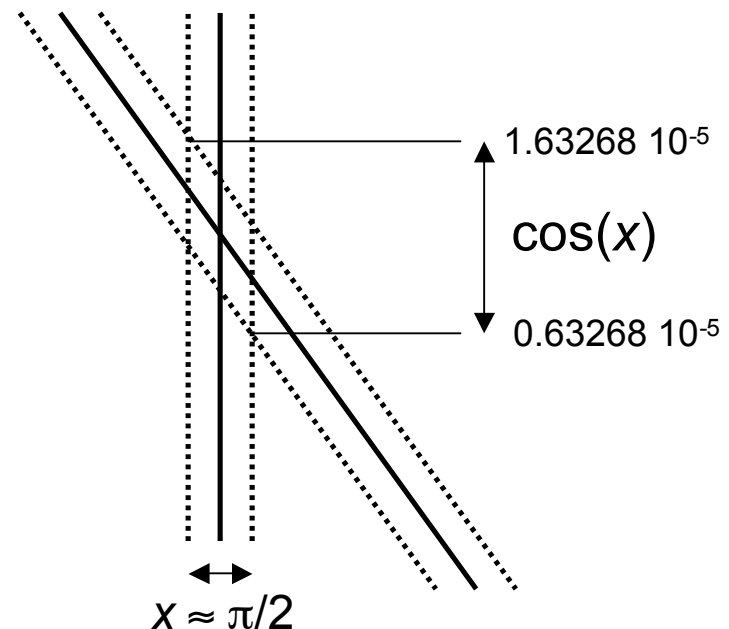
- Definitions

□ Absolute error	$= f(x+h) - f(x)$	$\approx h f'(x)$
□ Relative error	$= (f(x+h) - f(x)) / f(x)$	$\approx h f'(x) / f(x)$



An Ill-Conditioned Example

- Let $x = \pi/2$ and let h be a small perturbation to x
 - Absolute error = $\cos(x+h) - \cos(x) \approx -h \sin(x) \approx -h$
 - Relative error = $(\cos(x+h) - \cos(x)) / \cos(x) \approx -h \tan(x) \approx -\infty$
- Small change in x near $\pi/2$ causes relative large change in $\cos(x)$
 - $\cos(1.57078) = 1.63268 \cdot 10^{-5}$
 - $\cos(1.57079) = 0.63268 \cdot 10^{-5}$





SIMD Short Vector Extensions

- The use of **SIMD short vector extensions** can result in large performance gains
 - Instruction set extensions execute fast
 - New wide registers to hold short vectors of ints, floats, doubles
 - Parallel operations on short vectors
 - Typical vector length is 128 bit, which holds 4 floats, 2 doubles, or 1 to 16 ints (128 bit to 8 bit)
- Technologies:
 - MMX and SSE (Intel)
 - 3DNow! (AMD)
 - AltiVec (PowerPC)
 - PA-RISC MAX (HP)



Intel SSE Programming

- Programming languages such as C, C++, and Fortran do not natively support SIMD instructions
- The Intel compiler supports four methods to use SSE:
 - **Assembly**: direct control, but hard to use and processor-specific
 - **Intrinsics**: similar to assembly instructions with operands that are C expressions, but may be processor-specific
 - **C++ class libraries**: easy to use and portable, but limited support for instructions and gives lower performance
 - **Automatic vectorization**: no source code changes needed, new instruction sets automatically used, but compiler may fail to vectorize code



C++ Class Libraries for SSE

- Integer class types ***Ibvecn***

I8vec8	(8 8bit)	I8vec16	(16 8bit)
I16vec4	(4 16bit)	I16vec8	(8 16bit)
I32vec2	(2 32bit)	I32vec4	(4 32bit)
I64vec1	(1 64bit)	I64vec2	(2 64bit)
		I128vec1	(1 128bit)

Note: use 's' or 'u' after 'I' for packed signed or unsigned integers, e.g. ***Is32vec4***

- Floating point class types ***Fbvecn***

F32vec4	(4 32bit)	F64vec2	(2 64bit)
----------------	-----------	----------------	-----------



C++ Class Library Example

```
#include <dvec.h> // SSE2
...
// array of ints, 16-byte aligned
__declspec(align(16)) int array[len];
...
Is32vec4 *array4 = (Is32vec4*)array;
for (int i = 0; i < len/4; i++)
    array4[i] = array4[i] + 1; // increments 4 ints
```



SIMD Instruction Intrinsics

- Use `#include <emmintrin.h>` (SSE2) or `<pmmintrin.h>` (SSE3)

- Data types:

<code>__m64</code>	MM register
<code>__m128</code>	packed single precision (XMM register)
<code>__m128d</code>	packed double precision (XMM register)
<code>__m128i</code>	packed integer (XMM register)

- Intrinsics operate on these types and have the format:

`_mm_op_suffix(...)`

where `op` is an operation and `suffix` denotes the packed (**p**), extended packed (**ep**), or scalar (**s**) data operated on followed by the operation type:

<code>s</code>	single precision
<code>d</code>	double precision
<code>i#</code>	integer of # bits (8, 16, 32, 64, 128)
<code>u#</code>	unsigned integer of # bits (8, 16, 32, 64, 128)



SIMD Instruction Intrinsics

Examples

- Load two doubles in a vector:
`double a[2] = {1.0, 2.0};`
`__m128d x = _mm_load_pd(a);`
- Add two vectors containing two doubles:
`__m128d a, b;`
`__m128d x = _mm_add_pd(a, b);`
- Multiply two vectors containing four floats:
`__m128 a, b;`
`__m128 x = _mm_mul_ps(a, b);`
- Add two vectors of 8 16-bit signed ints using saturating arithmetic
`__m128i a, b;`
`__m128i x = _mm_adds_epi16(a, b);`
- Compare two vectors of 16 8-bit signed integers
`__m128i a, b;`
`__m128i x = _mm_cmpgt_epi8(a, b);`
- Note: rounding modes and exception handling are set by masking the MXCSR register



Intrinsics Example

```
#include <emmintrin.h> // SSE2
...
// array of ints, 16-byte aligned
__declspec(align(16)) int array[len];
...
__m128i ones4 = _mm_set1_epi32(1);
__m128i *array4 = (__m128i*)array;
for (int i = 0; i < len/4; i++)
    array4[i] = _mm_add_epi32(array4[i], ones4);
```



Memory Alignment

- Memory operands must be **aligned** for maximum performance
 - 8-byte aligned for MMX
 - 16-byte aligned for SSE
 - Use `_declspec(align(8))` and `_declspec(align(16))`
- Aligned memory load/store operations segfault on unaligned memory operands
 - `__m128d x = _mm_load_pd(aligned_address);`
- Unaligned memory load/store operations are safe to use but incur high cost
 - `__m128d x = _mm_loadu_pd(unaligned_address);`
- Use `_mm_malloc(len, 16)` for dynamic allocation



Data Layout

- Application's data layout may need to be reconsidered to use SIMD instructions effectively
- Vector operations require consecutively stored operands in memory
 - Cannot vectorize row-wise with row-major matrix layout
 - Cannot vectorize column-wise with column-major matrix layout
- Aligned structs may have members that are unaligned
 - ```
struct node {
 int x[7];
 int dummy; // padding to make a[] aligned
 float a[4];
}
```



# **GMP:**

## **GNU Multi-Precision Library**

- GMP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers
- GMP aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types
- Used by many projects, including computer algebra systems
- Programming language bindings: C, C++, Fortran, Java, Prolog, Lisp, ML, Perl, ...
- License: LGPL



# GMP Usage

- Introduces three types (C language binding):

|                    |              |
|--------------------|--------------|
| <code>mpz_t</code> | bigint       |
| <code>mpq_t</code> | big rational |
| <code>mpf_t</code> | bignum       |

- Use (similar for mpq and mpf):

```
#include <gmp.h>
```

```
mpz_t n;
```

```
mpz_init(n);
```

```
mpz_init2(n, 123);
```

```
mpz_init_set_str(n, "6", 10);
```

```
...
```

```
mpz_clear(n);
```

Use one of these to initialize.

Note: `mpf_init2` sets precision

↑  
*base*

- Link with `-lgmp`



# GMP

- Dynamic memory allocation
  - Efficient implementation limits the need for frequent resizing
  - Configurable
- 150 integer operations on unlimited length bigint
  - Arithmetic
  - Comparison
  - Logic and bit-wise operations
  - Number theoretic functions
  - Random numbers
- 60 floating point operations on high-precision bignum
  - Arithmetic
  - Comparison



# GMP C Example

```
void myfunction(mpz_t result, mpz_t param, unsigned long n)
{
 unsigned long i;

 mpz_mul_ui(result, param, n);
 for (i = 1; i < n; i++)
 mpz_add_ui(result, result, i*7);
}

int main(void)
{
 mpz_t r, n;
 mpz_init(r);
 mpz_init_set_str(n, "123456", 0);

 myfunction(r, n, 20L);
 mpz_out_str(stdout, 10, r); printf("\n");

 return 0;
}
```



# GMP C++ Bindings

- Defines three classes:

|                        |                   |
|------------------------|-------------------|
| <code>mpz_class</code> | for bigint        |
| <code>mpq_class</code> | for big rationals |
| <code>mpf_class</code> | for bignum        |

- Most GMP functions have C++ wrappers, but not all

- ☐ Root of 0.2 in 1000 bit precision:

```
mpf_class x(0.2, 1000), y(sqrt(x));
```

- ☐ GCD of two bigints:

```
mpz_class a, b, c;
```

```
...
```

```
mpz_gcd(a.get_mpz_t(), b.get_mpz_t(), c.get_mpz_t());
```

- Use `#include <gmpxx.h>` and link `-lgmpxx -lgmp`



# GMP C++ Example

```
#include <gmpxx.h>

mpz_class a, b, c; // integers

a = 1234;
b = "-5678";
c = a+b;
cout << "sum is " << c << "\n";
cout << "absolute value is " << abs(c) << "\n";
```

Expression like  $a=b+c$  results in a single call to the corresponding `mpz_add`, without using a temporary for the  $b+c$  part.

The classes can be freely intermixed in float, double, int/long, expressions.



## Further Reading

- “*What Every Computer Scientist Should Know About Floating Point Arithmetic*” by D. Goldberg, Computing Surveys, 1991  
[http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html)
- Chapters 11 and 12 of “*The Software Optimization Cookbook*” 2nd ed by R. Gerber, A. Bik, K. Smith, and X. Tan, Intel Press.
- Intel Compiler intrinsics reference:  
[http://download.intel.com/support/performance/c/linux/v9/intref\\_cls.pdf](http://download.intel.com/support/performance/c/linux/v9/intref_cls.pdf)
- GNU GMP: <http://gmplib.org>