applied informatics

Cross-Platform Issues With Floating-Point Arithmetics in C++

ACCU Conference 2006

Günter Obiltschnig
guenter.obiltschnig@appinf.com

ENGINEERING THE FUTURE

Floating-Point Arithmetic

- is hard...
(the classic paper by Goldberg - What Every Computer Scientist Should Know About Floating-Point Arithmetic has 94 pages)
- is even harder if cross-platform issues come into play

Floating-Point Types in C++

- Three distinct floating-point types
  - float
  - double
  - long double
- No binary representation is specified

IEEE 754-1985

- De-facto standard for float and (long) double
- Usually implemented in hardware
- Specifies:
  - binary representation
  - semantics for floating-point operations

History of IEEE 754

- 1960's and 1970's: each line of computers supported its own format, range and precision
- 1976: Intel started work on a coprocessor
- 1977: IEEE p754 was formed
- Draft by Kahan/Coonen/Stone (based on Kahan's work for Intel)
- 1984: implementations by Intel, AMD, Apple, Motorola, IBM, etc.

long double

- Not fully supported everywhere
- Corresponding IEEE 754 type: double extended/ quadruple
- May be same as double (64 bit), 80 bit or 128 bit
- Often no hardware support – performance
- Not portable

Binary Representation

$$value = sign \cdot mantissa \cdot 10_2^{exponent}$$

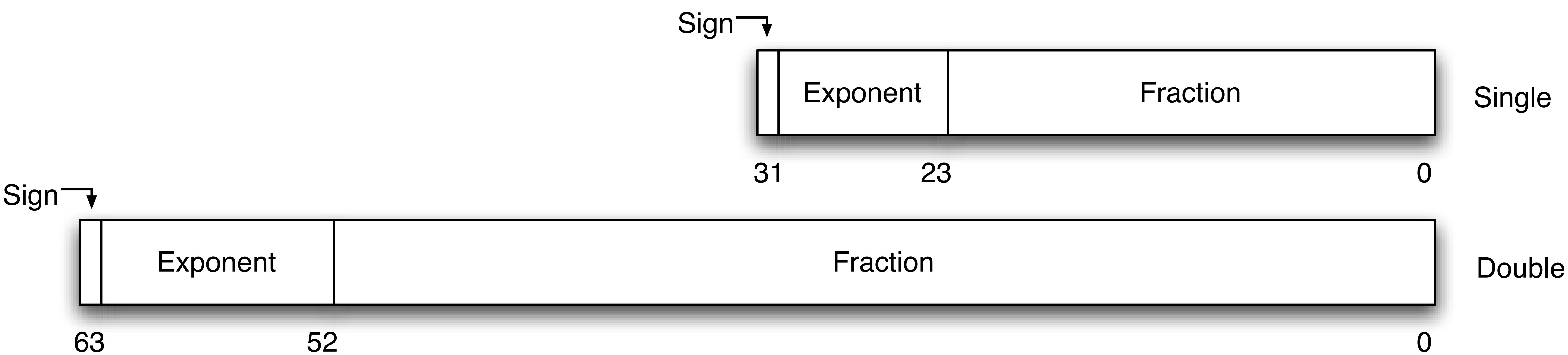
C++ Type	Precision	Size in bits	Mantissa bits	Exponent bits
float	single	32	24	8
double	double	64	53	11
long double	double ext. (quadruple)	128 80	113 65	15

sign bit not shown

Mantissa vs. Fraction

- IEEE 754 floating-point numbers are normalized (exception: denormal numbers)
- The radix point is after the first non-zero digit
- The most significant bit of the mantissa is 1
- No need to store the MSB
- Fraction: Mantissa excluding MSB

Binary Representation



Example:

$1.11101000010111111000111*10^{-01111110}$ $(2.24*10^{-38})$

Sign	Exponent	Fraction
0	000 0000 1	111 0100 0010 1111 1100 0111

Exponent Bias

Example:
$1.11101000010111111000111 * 10^{-01111110}$ $(2.24 * 10^{-38})$

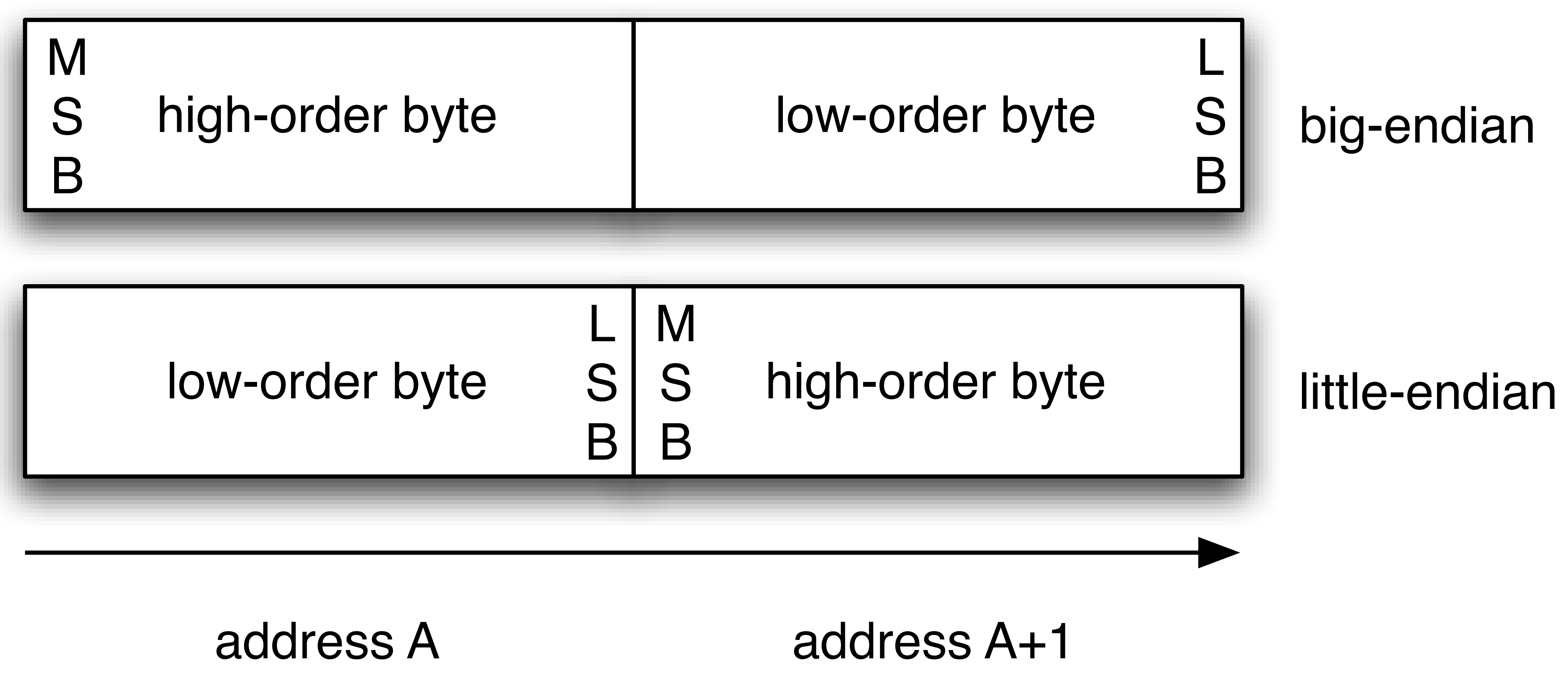
Sign	Exponent	Fraction
0	000 0000 1	111 0100 0010 1111 1100 0111

- –01111110 binary is -126 decimal
- exponent stored as 00000001
- bias of 127 (1023 for double) is added
- exponent is always positive

Ranges

	Normal	Denormal	Approx. Decimal
float	$\pm\ 2^{-149}$ to $(1\text{-}2^{-23})*2^{-126}$	$\pm\ 2^{-126}$ to $(2\text{-}2^{-23})*2^{127}$	$\pm\ \sim 10^{-44.85}$ to $\sim 10^{38.53}$
double	$\pm\ 2^{-1074}$ to $(1\text{-}2^{-52})*2^{-1022}$	$\pm\ 2^{-1022}$ to $(2\text{-}2^{-52})*2^{1023}$	$\pm\ \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Byte Order


MSB
high-order byte
low-order byte
LSB
big-endian
low-order byte
LSB
MSB
high-order byte
little-endian
address A
address A+1

Byte Order

$\pi = 3.1415926...$ as float:

Sign	Exponent	Fraction
0	100 0000 0	100 1001 0000 1111 1101 1011

byte offset	0	1	2	3
big endian	40	49	0f	db
little endian	db	0f	49	40

Storage vs. Native Format

- Intel (IA32) hardware supports only 80-bit double extended format
- Conversion is required before and after arithmetic operation
- Rounding errors
- Loss of precision if intermediates are stored in memory

IEEE 754 Special Features

- NaN
- Infinity
- Signed Zero
- Gradual Underflow and Denormal Numbers
- Rounding Modes, Exceptions, Flags

NaN - Not a Number

- Result of $\sqrt{n}$ where $n < 0$ and 0/0
- NaN bit pattern does not yield valid number
- Exponent is all ones
- Fraction is non-zero
- QNaN (Quiet NaN): fraction MSB set: indeterminate operation
- SNaN (Signalling NaN): fraction MSB cleared: invalid operation
- NaN is signed

NaN Operations

- The result of every arithmetic operation involving at least one NaN is again a NaN
- The comparison of a NaN with another value, including another NaN, always yields false.

NaN Issues

- some implementations do not distinguish between SNaN and QNaN
- some implementations always generate negative NaNs, regardless of the operands signs
- bit pattern for the NaN fraction is not exactly specified (except non-zero-ness and MSB)
- implementations are free to use fraction bits for whatever they like

Checking for NaN

- C99 library: isnan()
- non-standard functions: _isnan()
- not portable

Checking for NaN

- std::numeric_limits<>
  - quiet_NaN(), has_quiet_NaN()
  - signalling_NaN(), has_signalling_NaN()
- no comparison with xxx_NaN()
- portable workaround:
  (x != x) == true iff x is NaN

Infinity

- Result of divide by zero (except 0/0)
- Infinity bit pattern does not yield valid number
- Exponent is all ones
- Fraction is zero
- Infinity is signed

Infinity Operations

Operation	Result
x / ±Infinity	0
±Infinity * ±Infinity	±Infinity
x / 0 (for x != 0)	±Infinity
±0 / ±0	NaN
Infinity + Infinity	Infinity
Infinity - Infinity	NaN
±Infinity / ±Infinity	NaN
±Infinity * 0	NaN

Testing for Infinity

- C99: isinf()
- Non-standard: _infinity(), fpclass()
- compare with std::numeric_limits<>::infinity()

Signed Zero

- Not directly representable (due to normalization)
- Exponent and fraction are all zero
- Sign bit can be 1 or 0
- +0 and -0 are distinct numbers,
  but compare as equal

Gradual Underflow

- What if the result of a computation lies between zero and the smallest normalized number?
- Treat it as zero?
  x - y == 0 may be true for x != y
  You are in trouble if the result is used later in the computation.
- Or sacrifice precision?
  Gradual Underflow → Denormal Numbers

Denormal Numbers

- Zero exponent
- Mantissa is not normalized
- MSB of mantissa is zero
- The smaller the denormal number, the lower its precision

Denormal Numbers

$$
\begin{array}{r}
8.97*10^{-38} \\
\underline{-\ 8.95*10^{-38}} \\
2.0*10^{-40}
\end{array}
\quad (1.99999*10^{-40})
$$

Decimal	Internal	Interpret as (binary)
$8.97*10^{-38}$	0 00000011 11101000010111111000111	$1.11101000010111111000111*10^{11}$
$8.95*10^{-38}$	0 00000011 11100111010010001100110	$1.11100111010010001100110*10^{11}$
$1.99999*10^{-40}$	0 00000000 00000100010110110000100	$0.00000100010110110000100*10^{1}$

Rounding Modes

- The result of an operation cannot be exactly represented
  - Rounding towards nearest
  - Rounding towards zero
  - Rounding towards $+\infty$
  - Rounding towards $-\infty$
- C++: No standard API
- C99: Floating-Point Environment

Exceptions and Flags

- What to do if an operation fails?
  - overflow
  - underflow
  - divide by zero
  - invalid operation
  - inexact result

Exceptions and Flags

- Deliver special result (NaN, infinity)
- Set status flags
- Interrupt/Trap (SIGFPE)
- C++: No standard API
- C99: Floating-Point Environment

Portability Issues

- The implementation of transcendental functions sin(), cos(), exp(), etc.
- Conversion from decimal to binary and back
- Conversion to string (especially NaN and Infinity)
- Conversion to integer (NaN and Infinity)
- Support for some IEEE 754 features varies or may incur performance penalties
- $z = pow(x, y)$ is only portable for integral $y > 0$
  Workaround: $z = exp(y * log(x))$

API Support

- IEEE 754 does not specify API
- C99 has Floating-Point Environment <fenv.h>
- Most platforms have proprietary APIs

API Comparison

Rounding Modes

	C99	Windows	Solaris
downward	FE_DOWNWARD	RC_DOWN	FP_RM
upward	FE_UPWARD	RC_UP	FP_RP
to nearest	FE_TONEAREST	RC_NEAR	FP_RN
toward zero	FE_TOWARDZERO	RC_CHOP	FP_RZ

API Comparison

Flags

	C99	Windows	Solaris
divide by zero	FE_DIVBYZERO	SW_ZERODIVIDE	FP_X_DZ
inexact	FE_INEXACT	SW_INEXACT	FP_X_IMP
overflow	FE_OVERFLOW	SW_OVERFLOW	FP_X_OFL
underflow	FE_UNDERFLOW	SW_UNDERFLOW	FP_X_UFL
invalid	FE_INVALID	SW_INVALID	FP_X_INV

API Comparison

Operations

	C99	Windows	Solaris
clear flags	feclearexcept()	_clearfp()	fpsetsticky()
test flag	fetestexcept()	_statusfp()	fpgetsticky()
set rounding mode	fesetround()	_controlfp()	fpsetround()
get rounding mode	fegetround()	_controlfp()	fpgetround()
test for infinity	isinf()	_finite()	fpclass()
test for NaN	isnan()	_isnan()	isnanf() isnan()
copy sign	copysignf() copysign()	_copysign()	copysign()
save environment	fegetenv()	_controlfp()	fpsetround() fpsetmask()
restore env.	fesetenv()	_controlfp()	fpgetround() fpgetmask()

The FPEnvironment Class

```
class FPEnvironment
{
public:
   enum RoundingMode
   {
      FP_ROUND_DOWNWARD
      FP_ROUND_UPWARD
      FP_ROUND_TONE
      FP_ROUND_T
   };
   enum
   {

   oid clearFlags();
   tic bool isFlag(Flag flag);
   static void setRoundingMode(RoundingMode mode);
   static RoundingMode getRoundingMode();
   static bool isInfinite(float value);
   static bool isInfinite(double value);
   static bool isInfinite(long double value);
   static bool isNaN(float value);
   static bool isNaN(double value);
   static bool isNaN(long double value);
};
```

find the implementation in the
C++ Portable Components
http://poco.sf.net

Conclusions

- ordinary float and double values can be exchanged in binary form between systems that implement IEEE 754, long double values cannot
- NaN and denormals are problematic
- byte order must be taken into account
- the same code may produce slightly different results on different systems
- you have to implement your own portable API for working with some IEEE 754 features

Q & A