

# How Computers Handle Numbers

*Some of the Sordid Details*

Nick Maclaren

July 2009

# This Is Now “Later”

Explanations of **a few** of the warnings

Please ask for more detail if interested

**WARNING:** This May Cause Nightmares

Anyone already confused should leave now  
Yes, I **AM** serious about that!

# More on Shifts

Hardware and languages mess these up

I know the history/excuses, from 40 years back

[ Gate count on discrete-logic computers ]

%deity alone knows why no improvement

Shifts often unsigned only in hardware

Involving the sign bit can have weird effects

Usually, only some bits of the count used

Typically, the bottom 5/6/8 bits of count

Why? The ICL 1904 (c. 1965) did it properly

# What Languages Do

Shifts  $\geq$  bits in word usually undefined  
Java defined, but uses only 5/6 bits of shift

Usually undefined if signed shifts overflow  
I.e. left shift a one into or out of sign bit  
Right shifts on negatives usually unspecified

As mentioned earlier, Python gets these right  
Why don't other languages do the same?

# More on Signed/Unsigned

Exactly **what** does the language specify?

- May vary with compiler versions, options, etc.

Main “**gotcha**” is with implicit conversions

- Rules often depend on language **context**

In **C**: preprocessor/constant/initializer/other

Often **undefined**  $\Rightarrow$  behaviour **unpredictable**

**C** result depends on **order** of type changes

Not just **signed/unsigned/float** but **size** of number

# C/C++ Signed/Unsigned (1)

C/C++ can be insane even when defined

Assume 32-bit ints and 64-bit longs:

- (long)-0x7F000000 = -2130706432
- (long)-0x80000000 = 2147483648

char x = 'a'; x == 'a' may be false

char x = 'a'; islower(x) may be undefined

## C/C++ Signed/Unsigned (2)

```
extern void fred(long,long);  
int a = 1, b = -1;  
unsigned int c = 1, d = -1;  
long A = 1, B = -1;  
unsigned long C = 1, D = -1;
```

fred(a\*d,b\*c)  $\Rightarrow$  fred(-1,-1)

fred(A\*D,B\*C)  $\Rightarrow$  fred(-1,-1)

fred(a\*D,b\*C)  $\Rightarrow$  fred(-1,-1)

fred(A\*d,B\*c)  $\Rightarrow$  fred(-1,-1) . . .

. . . **OR**  $\Rightarrow$  fred(4294967295,-1)

# More on IEEE Signed NaNs

Consider  $X = 0.0/0.0 = \text{NaN}$

Fortran `SIGN(1.0,X)` and C `copysign(1.0,X)`

Both must be either  $-1.0$  or  $+1.0$ , unpredictably

And, for both,  $\text{SIGN}(1.0,X) = -\text{SIGN}(1.0,-X)$

But  $\text{SIGN}(1.0,0.0+X)$ ,  $\text{SIGN}(1.0,1.0*X)$  etc.?

They must be either  $-1.0$  or  $+1.0$ , unpredictably

That is a useful specification? Get real

# IEEE and Decimal (1)

Two encodings (a committee compromise)

One encodes 3 decimal digits into 10 bits

The other uses binary, but bounded by  $10^N$

The standard says 'bad' values are valid

Random rubbish will give defined nonsense

Exact half rounding is language-defined

It does have a recommended default

# IEEE and Decimal (2)

Infinites and NaNs are similar to binary  
No denormalised numbers but there are cohorts

Probably separate decimal and binary types  
Probably only IBM will push it much

Unformatted I/O may well become much trickier  
There MAY be compiler options to convert  
That is possible in Fortran but not C etc.

# Decimal Cohorts

Just like **IBM 370 unnormalised** numbers

E.g.  $1.23 = 0.123 \times 10^1 = 0.00123 \times 10^3$

Cohort members are used in arcane ways

I haven't bothered to study this area in detail

May cause strange output (e.g. **0.00123e3**)

Decimal **might** do **just** what you want

And pigs might fly, but it's not likely

# IEEE 754 Rounding Modes

**DON'T GO THERE**

The reasons are too complicated to go into  
Yes, even in these 'extra' slides – sorry

Nor primarily **IEEE 754**'s fault  
Please ask if you want to know them

# Exception Handling Design

**Clean** model 1 (trad./**LIA-1**) -- trap on failure  
Now generally rejected on dogmatic grounds

**Clean** model 2 (**IEEE 754**) -- use error values  
OK, when done **properly** -- but it **isn't**

**Ghastly** model 3 (**Java/C99/C++**) -- define result  
Changes **numeric** error to **logical** error

**Ghastly** model 4 (very common) -- undefined  
If you make a mistake, that's **your** problem

# Fortran and IEEE Exceptions (1)

This is available only in **Fortran 2003**

It defines some **IEEE 754** exception handling

Actually pretty well, considering the constraints

- But an implementation need not support it

I don't know many implementations yet, either

I expect to retire before seeing it much used

I have no idea how useful or reliable it will be

## Fortran and IEEE Exceptions (2)

Flags are associated with the **call tree**

They are **saved and cleared** on procedure **entry**

And **merged back** on procedure return

Flags never get **unset** except by programmer

**Intrinsics** and **I/O** never set flags unnecessarily

But any **serious** exception flag shows an error

I.e. **Divide-by-zero**, **overflow** and **invalid**

# Fortran and IEEE Exceptions (3)

- Big question – are they set reliably?

There is one **explicit** exception

```
IF (X/Y > Z) PRINT *, 'Oh'
```

And the **general** requirement is a little vague

- **Please** tell me if you investigate!

# C99 and IEEE Exceptions

CENSORED

Reason: good taste and public decency

Ask me for the sordid details if you need it

- You cannot imagine the “gotchas”

Don't trust anything that implies it is useful

There is a bit on it later, actually . . .

# C++ (Forthcoming Standard)

C++ is schizophrenic about C

Is it a separate language?

Is it a language extension?

No, it's %deity alone knows what

C++ 2003 inherits most of C99, not C90

I failed to get the inconsistencies fixed

C++ relies on C for its arithmetic etc.

- So that area will be broken in the same way

# Exception Implementation

Modern FP hardware/software is very sick

C99 IEEE 754 requires flag-and-continue  
Permits trapping to interrupt routine

But hardware interrupts are totally privileged  
Fixups by kernel/library/compiler handshakes  
Unlike in 1970s, not documented in architecture

Option/configuration-dependent bugs are legion  
Can even crash systems from applications

# Complex Number Exceptions (1)

Not an easy problem, made worse by misdesign  
**Complex** and **real** fundamentally incompatible

The **real line** is closed by **two** infinities  
One at each end, obviously – i.e. like **IEEE 754**

The **complex plane** is closed by **one** infinity  
A sort of enclosing circle, but a **single** point

**Cartesian** representation is all wrong for that

# Complex Number Exceptions (2)

IEEE 754 shows the problem very clearly  
Consider division as an example

$$(A,B)/(C,D) = (A*C+B*D, B*C-A*D)/(C*C+D*D)$$

Blows up in almost any arithmetic when:

$$\text{abs}(C,D) > \sqrt{\text{maxreal}}$$

So we need something a bit fancier

# Complex Number Exceptions (3)

A better (but not perfect) approach is:

if  $\text{abs}(C) > \text{abs}(D)$  :

$$r = D/C;$$

$$(A,B)/(C,D) = (A+B*r, B-A*r)/(C+D*r)$$

else :

$$r = C/D;$$

$$(A,B)/(C,D) = (A*r+B, B*r-A)/(C*r+D)$$

That gets it right, except near infinity

# Complex Number Exceptions (4)

$X = 1.0e308$ ,  $N = 0.0, 0.1, 0.2, 0.3, \dots$   
Calculate  $(X,X)/(X,N*X)$

$N=0.0\dots0.7 \Rightarrow (1.0,1.0) \dots (1.14,0.20)$

$N=0.8 \Rightarrow (+infinity,0.12) \quad \leftarrow\leftarrow\leftarrow\leftarrow\leftarrow$

$N=0.9\dots1.2 \Rightarrow (NaN,0.0)$

$N=1.3\dots1.7 \Rightarrow (0.0,0.0) \quad \leftarrow\leftarrow\leftarrow\leftarrow\leftarrow$

$N=1.8\dots \Rightarrow (NaN,NaN)$

C99 Annex G example code is even worse

# General C99 Nightmares

Wording is ambiguous and inconsistent

Footnotes/optional wording overrides main text

No agreement even on the intent in SC22WG14

Perhaps 1–2 full implementations after 12 years

Developers/customers still often specify C90

long was longest integer type — now isn't

Breaks most portable C90 code, subtly

Implications and details rarely understood

# C99/IEEE Nightmares (1)

`<math.h>` may set either `errno` or IEEE flags  
All but `<math.h>/<fenv.h>` may set spuriously  
Error values may be anything — 0.0, 42.0, NaN  
Only implementation-defined value, anyway  
Makes portable error detection a nightmare

Mode setting is disaster — but you don't want it  
Can't even call standard library or return  
Don't even think about `setjmp/longjmp/signal`

## C99/IEEE Nightmares (2)

IEEE 754 only if `__STDC_IEC_559__` is set

Nobody knows what 'IEEE' features do if not

Or what `FP_CONTRACT ON` means if it is set

Or if `CX_LIMITED_RANGE ON` means anything

Flags also need pragma `FENV_ACCESS ON`

Totally incompatible with optimisation

Flags corrupted by library, just like `errno`

Compilers will probably just get them wrong

# C99/IEEE Nightmares (3)

Many **REQUIRED** ways to lose **NaN** values  
Contradicts **IEEE 754**'s stated intent (6.2)

$\text{fmax}(1.23, \text{NaN}) = 1.23$

$\text{atan}(\pm 0.0, \pm 0.0)$  returns  $-\pi, -0.0, +0.0$  or  $+\pi$

$\text{pow}(-1, \pm \text{infinity})$  returns  $1.0$

Simply comparing values is ambiguous

And so on, ad nauseam

The sign of **NaNs** is meaningful (e.g. **copysign**)

But they don't actually contain any meaning!

# C99/IEEE Annex G (1)

Complex **infinity**/**NaN** **totally** broken

$A = (1.0, 0.0) / 0.0 = (+\text{inf}, \text{NaN}) = \text{infinity}$

$A, A+A, A*A$  **must** be **infinities**

$A*(+\text{inf}) + A*(I*\text{NaN})$  **must** be a **NaN**

$A+(-1.0, 0.0)*A$  **must** be **either infinity OR NaN**

`double d = (INFINITY+I*0.0)*(1.0+I*0.0)`

`double e = (NaN+I*NaN)*(NaN+I*NaN)`

... `d, e` are **undefined** (may be **anything**)

## C99/IEEE Annex G (2)

Complex arithmetic may set flags **spuriously**  
<complex.h> need not set & may corrupt **errno**

**CX\_LIMITED\_RANGE OFF** may be **very** slow  
Is **misimplemented** under some systems

Math. functions **defined** to lose **NaNs/infs**  
Depending on **explicitly undefined** behaviour

Example code for **division** already mentioned