Floating-Point Computations in the Real World

November 2010

Intro

This presentation is about the behaviour of floating-point computations.

“How do I ...
write reasonably robust code ...
which performs floating-point computations...
without being a numerical scientist?”

Floating-point characteristics

- Able to handle both large and small numbers, with decent precision
- Fixed-size memory representation
- High computational performance .. usually
- Often have concepts of non-numerical quantities (NaN, inf)

Floating-point computational model

Mental model for single-precision FP values:

All calculations are done in scientific format
X = ±1 * A.BBBBB * 10^C

The 6 most significant digits of “A.BBB..” are kept, and C... oh, and the sign.

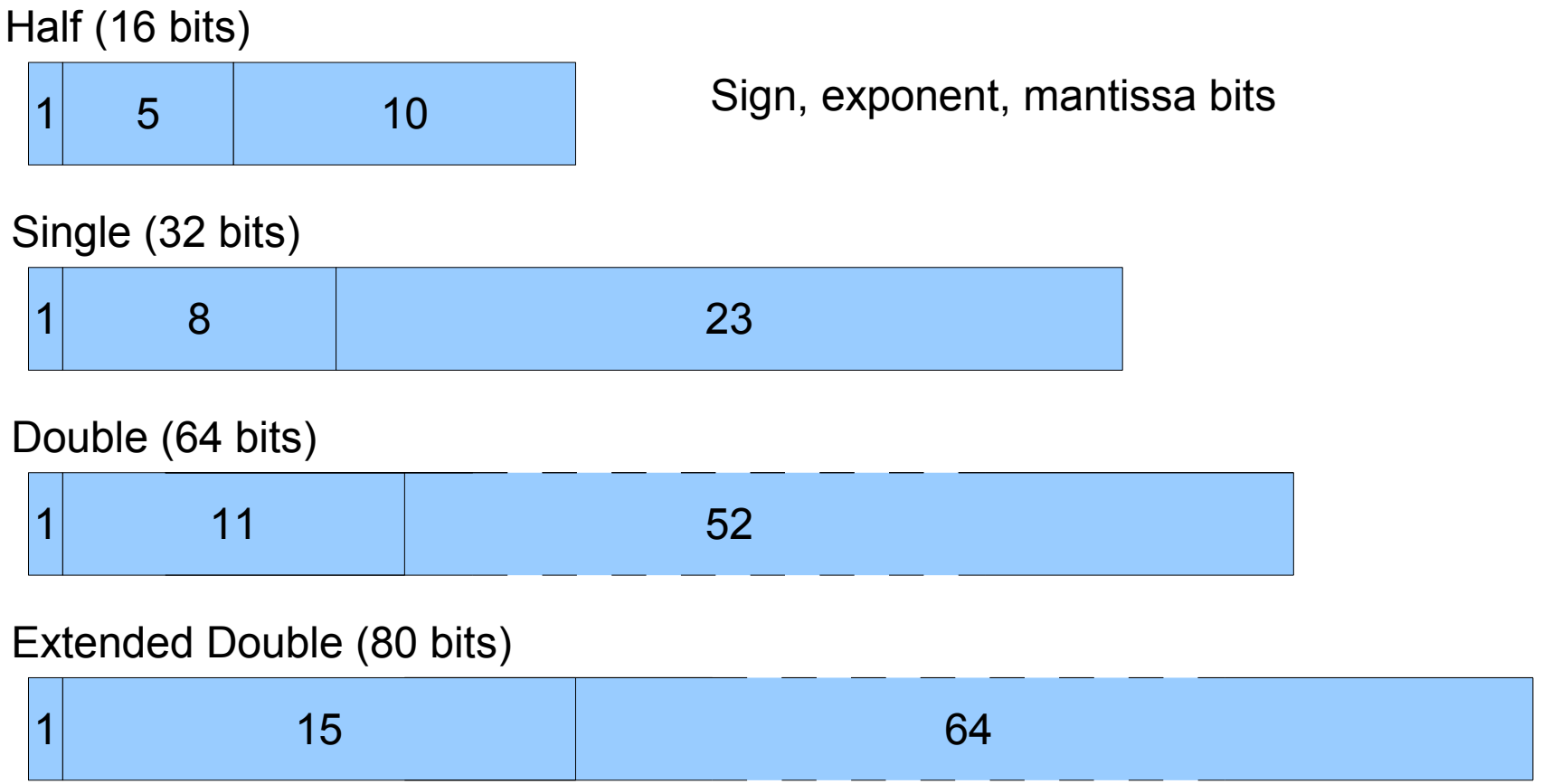
Floating-point encoding

Concept: represent the number in scientific notation (base 2), then encode it as 3 portions in binary

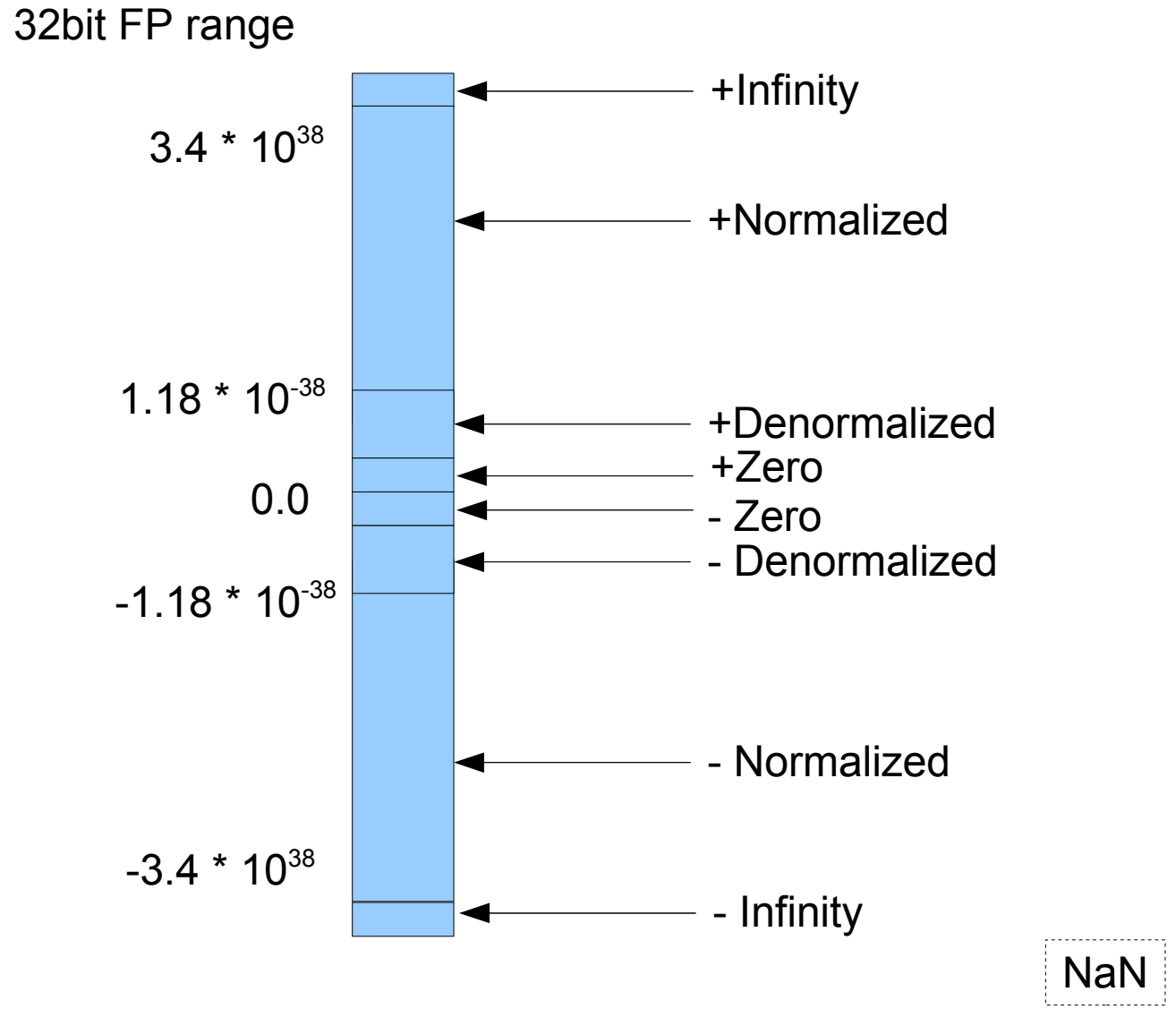
924.75 = +1 * 1.80615234375 * 2^9

Sign Mantissa Exponent

IEEE 754 floating-point formats



IEEE 754 number line



Let's encode some FP numbers

Type	Value	Binary representation
+Zero	0.0	0 | 00000000 | 00000000000000000000000
-Zero	-0.0	0 | 00000000 | 00000000000000000000000
Denormal	0.00000000000000000000000000000000000001	
	->	0 | 00000000 | 11011001110001111101110
Normalized	3,25	0 | 10000000 | 10100000000000000000000
+Inf		0 | 11111111 | 00000000000000000000000
-Inf		1 | 11111111 | 00000000000000000000000
NaN		? | 11111111 | XXXXXXXXXXXXXXXXXXXXXXX

Not all numbers encode exact

Example in base-10:

$$1_{10}/7_{10} = 0.166666666..._{10} \approx 0.166667_{10}$$

Example in base-2:

$$1_{10}/5_{10} = 1_{2}/101_{2} = 0.0110011001..._{2} \approx 0.0110011_{2}$$

Which numbers encode exact?

1.0 .. yes
12345.0 ... yes

1000000000.0 .. no

0.5 .. yes
0.25 ... yes

0.3 ... nope

Not all FP implementations are created equal

	Zero	Denormals	Normals	Inf	Nan
680x0 FPU	Y	Y	Y	Y	Y
X86 FPU	Y	Slow	Y	Y	Y
X86 SSE	Y	N	Y	Y	Y
PPC (G3 mac)	Y	Slow	Y	Y	Y
PPC (G4 mac)	Y	Y	Y	Y	Y
PS2 VU	Y	N	Y	N	N
PS3 SPE	Y	N	Y	N	N

“Slow” means 10x-1000x slower than expected

Safe subset: Zero & normals

Relax

We'll continue in 5 minutes.

(Questions?)

Let's count with 32bit integer

<run example 1>

Let's count with 32bit FP

<run example 2>

What will happen?

Let's count with 32bit FP

Stops at 16777216. why?

Sometimes, A + b = A

Beware adding small & large numbers.

Let's accumulate time, FP

<run example 3>

What happens?

Let's accumulate time, FP

steplength goes up & down as precision goes down
important: difficult to predict **how** steplength changes

Let's accumulate time, integer

<run version 4>

Let's accumulate time, integer

steplength constant

but we get quantization

conclusion

Don't use floats if you can do it with integers instead

accumulated FP calculations accumulate error

Good strategy: accumulate in integer, then convert to FP

if possible, don't accumulate time (just be happy with the current delta)

Relax.

We'll continue in 5 minutes.

(Questions?)

Let's do some random maths

<compute x * (1/x) for some x>

Sometimes, x * (1/x) != x

The calculation result is not random. You can say in advance whether you will get exact 1.0 back.

But it's a lot of work. Usually too much work.

=> pretend that the calculation is “inexact”.

But wait, there's more

a + b * c = ?

Varies with CPU: IEEE754 compliance, # of guard bits, native computation format, mul+add or MAC, etc

Varies with compiler

=> pretend it is “inexact”

Moving calculations between CPUs

* converting a run-time calculation into compile-time

* converting a run-time calculation into data-build-time

* moving a calculating from a pixel/vertex shader to the C++ code

=> don't expect the same result
=> don't binary diff datafiles generated on different machines

Cross-compilation can also be dangerous

```
float divideByThree(float x)
{
    return f * (1.f / 3.f);
}
```

Build on x86, run on 360.

Who computes “1.0f / 3.0f”? The x86 or the 360?

=> pretend it is “inexact”

My approach

Assume that every FP operation (including assignments) carries an error of about 1%

Don't use ==, use < or > tests

Expect 0.0 and integers to be represented precisely, other values not (again, 1% error)

Use clamp operations to avoid out-of-bounds array accesses

That's about it

Questions?

References

What Every Computer Scientist Should Know About Floating-Point Arithmetic
David Goldberg

Denormal numbers in floating point signal processing applications
Laurent de Soras
http://ldesoras.free.fr/doc/articles/denormal-en.pdf