digital numerical representations

markus fangerau @ mbi/dkfz

16.3.2011

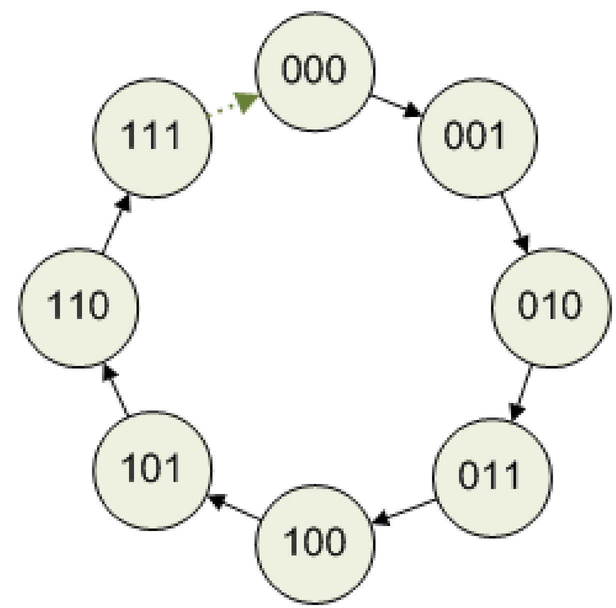
integer

- common sizes:
  - 8bit (char)
  - 16bit (short)
  - 32bit (int)
  - 64bit (int64_t)
- CPUs provide **modular arithmetic** operations to be performed on integers

Example: 3bit integer $\mathbb{Z}/8\mathbb{Z}$

bitwise	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
unsigned	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
signed	**0**	**1**	**2**	**3**	**-4**	**-3**	**-2**	**-1**

- In respect to **addition** forms an **abelian group** $(\mathbb{Z}/8\mathbb{Z}, +)$
- In respect to **multiplication** forms a **semigroup** $(\mathbb{Z}/8\mathbb{Z}, *)$
- It's **no finite field,** 8 (or any 2^n with n>1) is no prime



signed / unsigned comparison problems

– Example loop:

```
for ( unsigned int x=0; x<8; x++) ...;
```

– Now reverse iterating the loop as this won't work:

```
for ( unsigned int x=8-1; x>=0; x-- ) ...;
```

x is always >= 0, loop never ends

– Exactly reverse mirrored behavior while using unsigned types:

```
unsigned int x=8; while( x-- > 0 ) ...;
```

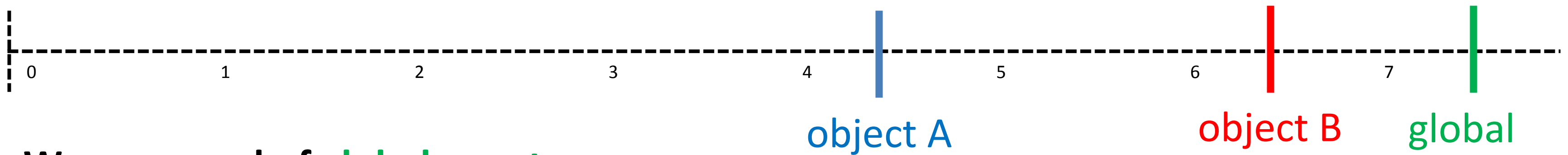
– Or just use signed types (like its enforced in Java):

```
for ( int x=8-1 ; x>=0 ; x-- ) ...;
```

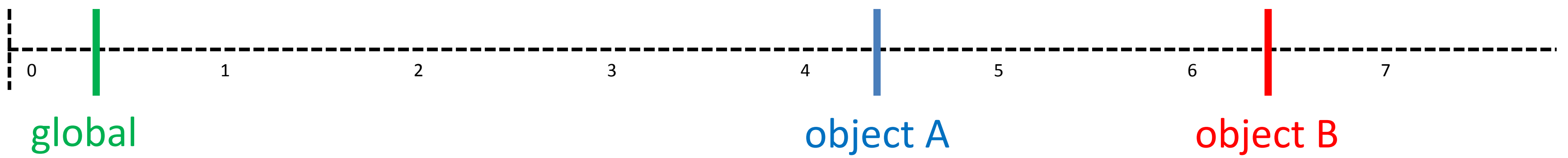
SO BE CAREFUL WHEN FIXING SIGNED/UNSIGNED WARNINGS

Example: unsigned modification time stamp counters

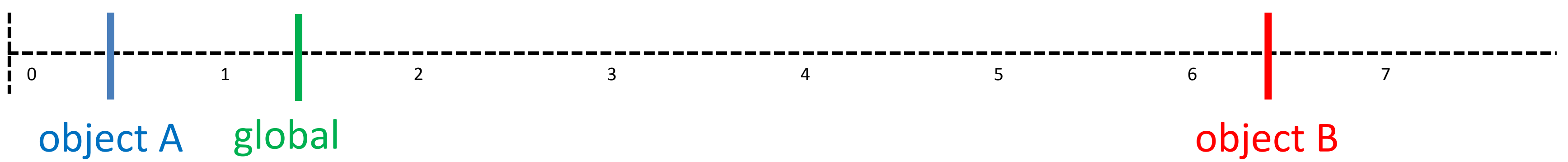
object B depends on object A:



Wrap-around of global counter:



object A gets updated:



object B does not recognize a modified object A and wont update

Example: unsigned modification time stamp counters

- After the counter wrap-around, all modification times are invalid and following does not work always:

```
If( objectA.time > objectB.time ) objectB.update();
```

when using an unsigned comparison your program may very likely show unexpected behavior after a specific time

- Following works by using modular arithmetic and computing the relative signed time difference:

```
If( int(objectA.time-objectB.time) > 0 )
   objectB.update();
```

using a signed comparison ignores the counter wrap-around

floating-point

Many numbers in decimal (base 10) format can not be exactly expressed as binary (base 2) floats:

Base 10 (decimal)		Base 2 (binary)	
1.0	$1.0 * 10^0$	**1.0**	$1.0 * 2^0$
2.0	$2.0 * 10^0$	**10.0**	$1.0 * 2^1$
0.5	$5.0 * 10^{-1}$	**0.1**	$1.0 * 2^{-1}$
0.0625	$6.25 * 10^{-2}$	**0.0001**	$1.0 * 2^{-4}$
123.0	$1.23 * 10^2$	**1111011.0**	$1.111011 * 2^6$
0.1	$1.0 * 10^{-1}$	**0.0011001100…**	$1.10011.. * 2^{-3}$
fixed-point	***floating-point***	***fixed-point***	***floating-point***

floating-point (IEEE-754)

32 (float) or 64 (double) bits

Sign	**Exponent+Bias**	**Mantissa**
1bit	**8 or 11bits**	**23 or 52 bits**

Bias is 127 for float and 1023 for double

represents: $\pm 1, Mantissa \; * \; 2^{Exponent}$

Can be interpreted as a logarithmic-like transferfunction

Example for a float:

1	**0 1 1 1 1 1 1 1**	**0 0 1 0 1 0 …**

$$\mathbf{-1,001010 \ldots * \; 2^{0}}$$

floating-point ranges & special values

	Denormalized	Normalized	Approximate Decimal
float	$\pm\ 2^{-149}$ to $(1\text{-}2^{-23})\times 2^{-126}$	$\pm\ 2^{-126}$ to $(2\text{-}2^{-23})\times 2^{127}$	$\pm$ ~$10^{-44.85}$ to ~$10^{38.53}$
double	$\pm\ 2^{-1074}$ to $(1\text{-}2^{-52})\times 2^{-1022}$	$\pm\ 2^{-1022}$ to $(2\text{-}2^{-52})\times 2^{1023}$	$\pm$ ~$10^{-323.3}$ to ~$10^{308.3}$

Denormalized:
if the exponent is all bits 0 then the value is a *denormalized* number (no implicit leading 1)

Infinite:
If the exponent is all bits 1 and the mantissa is all 0,
then the value represents +Infinity or –Infinity

NaN (quiet and signaling Not-a-Number's):
If the exponent is all bits 1 and the mantissa is non-zero,
Quiet NaNs propagate through computations
Signaling NaNs throw exception on use (i.e. for catching uninitialized variables access)

some words about precision

- multiplication: almost hassle-free
  - Especially multiplication with powers of two retain full precision
  - Exponents get added, Mantissas get multiplied
- addition: little more difficult
  - With too much differing exponents between summands, one of the mantissa may be partly or even completely ignored.

Get a “feeling” for how much of the mantissa is already used up and how much information you loose.

some about runtime

	integer	floating point
addition	fast	slow
subtraction	fast	slow
multiplication	slow	fast
division	very slow	very slow

- Prefer the term **a*b+c** instead of the term **(a+b)*c** , because it can be often the compiled into a single multiply-add-instruction which is twice as fast.

- Avoid casts between floating-point and integer (results in register transfers)

Reducing memory requirements

- nVidias and Industrial Light & Magic's 16bit floating-point (half)
- Shared exponent on vector float types (i.e. hdr RGBE photo images)
- Fixed-point representations omitting the exponent
- Applying transferfunctions before en/decoding as an integer (i.e. sRGB)

Danke