Using ISO Fixed Point Arithmetic for Neural Modelling

Theory and Practice

David Lester*

Manchester University

December 2013

* Research supported by EU-FET/EPSRC (BrainScales, Human Brain Project, BIMPC, BIMPA)

Overview

- Introduction to the ISO/IEC draft standard.

Overview

- Introduction to the ISO/IEC draft standard.
- How to write 'C' programs using fixed point arithmetic.

Overview

- Introduction to the ISO/IEC draft standard.
- How to write ‘C’ programs using fixed point arithmetic.
- Further Improvements.

Overview

- Introduction to the ISO/IEC draft standard.
- How to write 'C' programs using fixed point arithmetic.
- Further Improvements.
- Future Plans.

ISO Fixed Point Arithmetic

- By ISO Fixed Point Arithmetic we mean document:

  ISO/IEC TR 18037

  dated April 4, 2006, and produced under the auspices of ISO/IEC/ANSI.

ISO Fixed Point Arithmetic

- By ISO Fixed Point Arithmetic we mean document:

  ISO/IEC TR 18037

  dated April 4, 2006, and produced under the auspices of ISO/IEC/ANSI.
- Formal Title: "Programming languages - C - Extensions to support embedded processors", **ISO/IEC JTC1 SC22 WG14 N1169**

ISO Fixed Point Arithmetic

- By ISO Fixed Point Arithmetic we mean document:

  ISO/IEC TR 18037

  dated April 4, 2006, and produced under the auspices of ISO/IEC/ANSI.
- Formal Title: "Programming languages - C - Extensions to support embedded processors", **ISO/IEC JTC1 SC22 WG14 N1169**
- **Important**: "This document is an ISO/IEC draft Technical Report. It is not an ISO/IEC International Technical Report."

Fixed Point Data Types

- There are twelve *primary fixed-point types*. Six fractional types:

`unsigned short fract`	`signed short fract`
`unsigned fract`	`signed fract`
`unsigned long fract`	`signed long fract`

Fixed Point Data Types

- There are twelve *primary fixed-point types*. Six fractional types:

`unsigned short fract`	`signed short fract`
`unsigned fract`	`signed fract`
`unsigned long fract`	`signed long fract`

- And six `accum` types:

`unsigned short accum`	`signed short accum`
`unsigned accum`	`signed accum`
`unsigned long accum`	`signed long accum`

Fixed Point Data Types

- There are twelve *primary fixed-point types*. Six fractional types:

`unsigned short fract`	`signed short fract`
`unsigned fract`	`signed fract`
`unsigned long fract`	`signed long fract`

- And six `accum` types:

`unsigned short accum`	`signed short accum`
`unsigned accum`	`signed accum`
`unsigned long accum`	`signed long accum`

- For each of the primary types there is a corresponding (but different) *saturating fixed-point type*, *e.g.*

`unsigned long sat accum`

Fixed Point Data Type Minimal Formats

- We express the format of a fixed point number as

$$\langle \textit{optional sign bit} \rangle \langle \textit{integer part} \rangle . \langle \textit{fractional part} \rangle$$

Fixed Point Data Type Minimal Formats

- We express the format of a fixed point number as

$$\langle \textit{optional sign bit} \rangle \langle \textit{integer part} \rangle . \langle \textit{fractional part} \rangle$$

- The minimum sizes for the `fract` types:

`unsigned`	`short`	`fract`	`.7`	`signed`	`short`	`fract`	`s.7`
`unsigned`		`fract`	`.15`	`signed`		`fract`	`s.15`
`unsigned`	`long`	`fract`	`.23`	`signed`	`long`	`fract`	`s.23`

Fixed Point Data Type Minimal Formats

- We express the format of a fixed point number as

$$\langle \textit{optional sign bit} \rangle \langle \textit{integer part} \rangle . \langle \textit{fractional part} \rangle$$

- The minimum sizes for the `fract` types:

`unsigned short fract`	`.7`	`signed short fract`	`s.7`
`unsigned fract`	`.15`	`signed fract`	`s.15`
`unsigned long fract`	`.23`	`signed long fract`	`s.23`

- The minimum sizes for the `accum` types:

`unsigned short accum`	`4.7`	`signed short accum`	`s4.7`
`unsigned accum`	`4.15`	`signed accum`	`s4.15`
`unsigned long accum`	`4.23`	`signed long accum`	`s4.23`

Fixed Point Data Type gcc Formats

- The formats chosen by `gcc` for the `fract` types on `ARM`:

`unsigned`	`short`	`fract`	`.8`	`signed`	`short`	`fract`	`s.7`
`unsigned`		`fract`	`.16`	`signed`		`fract`	`s.15`
`unsigned`	`long`	`fract`	`.32`	`signed`	`long`	`fract`	`s.32`

Fixed Point Data Type gcc Formats

- The formats chosen by `gcc` for the `fract` types on `ARM`:

```
unsigned short fract .8      signed short fract s.7
unsigned       fract .16     signed       fract s.15
unsigned long  fract .32     signed long  fract s.32
```

- The formats chosen by `gcc` for the `accum` types on `ARM`:

```
unsigned short accum 8.8     signed short accum s8.7
unsigned       accum 16.16   signed       accum s16.15
unsigned long  accum 32.32   signed long  accum s32.31
```

Fixed Point Data Type gcc Formats

- The formats chosen by gcc for the fract types on ARM:

unsigned	short	fract	.8	signed	short	fract	s.7
unsigned		fract	.16	signed		fract	s.15
unsigned	long	fract	.32	signed	long	fract	s.32

- The formats chosen by gcc for the accum types on ARM:

unsigned	short	accum	8.8	signed	short	accum	s8.7
unsigned		accum	16.16	signed		accum	s16.15
unsigned	long	accum	32.32	signed	long	accum	s32.31

- There are two further non-ISO/IEC fract types:

unsigned	long	long	fract	.64
signed	long	long	fract	s.63

Arithmetic Operations (I)

- Consider the following program:

```
{
  fract x = 0.5;
  fract y;

  y = x + x;
  printf ("%k\n", y);
}
```

Arithmetic Operations (I)

- Consider the following program:

```
{
  fract x = 0.5;
  fract y;

  y = x + x;
  printf ("%k\n", y);
}
```

- What value is printed?

Arithmetic Operations (I)

- Consider the following program:

```
{
  fract x = 0.5;
  fract y;

  y = x + x;
  printf ("%k\n", y);
}
```

- What value is printed?
- Note constants and printing format control.

Arithmetic Operations (I)

- Consider the following program:

```
{
  fract x = 0.5;
  fract y;

  y = x + x;
  printf ("%k\n", y);
}
```

- What value is printed?
- Note constants and printing format control.
- `0.999984`.

Arithmetic Operations (II)

- Consider the following program:

```
{
  fract x = -1.0;
  fract y;

  y = x * x;
  printf ("%k\n", y);
}
```

Arithmetic Operations (II)

- Consider the following program:

```
{
  fract x = -1.0;
  fract y;

  y = x * x;
  printf ("%k\n", y);
}
```

- What value is printed?

Arithmetic Operations (II)

- Consider the following program:

```
{
  fract x = -1.0;
  fract y;

  y = x * x;
  printf ("%k\n", y);
}
```

- What value is printed?
- 0.999984 (or maybe 1.000000).

Arithmetic Operations (III)

- Consider the following program:

```
fract x = ...;
int y = (int)x;
```

Arithmetic Operations (III)

- Consider the following program:

```
fract x = ...;
int y = (int)x;
```

- What value is the value of `y`?

Arithmetic Operations (III)

- Consider the following program:

```
fract x = ...;
int y = (int)x;
```

- What value is the value of `y`?
- Probably what was intended was:

```
fract x = ...;
int y   = bitsr (x);
fract z = rbits (y);
```

Comparisons (I)

- Consider the following program:

```
#include <stdfix.h>

accum test1 (accum x, accum y)
{  return ((x < y)? x: y); }
```

Comparisons (I)

- Consider the following program:

```
#include <stdfix.h>

accum test1 (accum x, accum y)
{  return ((x < y)? x: y); }
```

- What arm code will be generated?

Comparisons (I)

- Disassembly of section .text:

```
00000000 <test1>:
   0:   e92d4038        push    {r3, r4, r5, lr}
   4:   e1a04000        mov     r4, r0
   8:   e1a05001        mov     r5, r1
   c:   e1a00001        mov     r0, r1
  10:   e1a01004        mov     r1, r4
  14:   ebfffffe        bl      0 <__gnu_cmpsa2>
  18:   e3500001        cmp     r0, #1
  1c:   c1a00004        movgt   r0, r4
  20:   d1a00005        movle   r0, r5
  24:   e8bd4038        pop     {r3, r4, r5, lr}
  28:   e12fff1e        bx      lr
```

Comparisons (II)

- As before, but using `bitsk` for comparison:

```
#include <stdfix.h>

accum test2 (accum x, accum y)
{  return ((bitsk(x) < bitsk(y))? x: y); }
```

Comparisons (II)

- Now we get:

```
00000000 <test2>:
   0:   e92d4070        push    {r4, r5, r6, lr}
   4:   e1a04001        mov     r4, r1
   8:   e1a05000        mov     r5, r0
   c:   ebfffffe        bl      0 <bitsk>
  10:   e1a06000        mov     r6, r0
  14:   e1a00004        mov     r0, r4
  18:   ebfffffe        bl      0 <bitsk>
  1c:   e1560000        cmp     r6, r0
  20:   a1a00004        movge   r0, r4
  24:   b1a00005        movlt   r0, r5
  28:   e8bd4070        pop     {r4, r5, r6, lr}
  2c:   e12fff1e        bx      lr
```

Comparisons (III)

- Now including my extra header file:

```
#include <stdfix.h>
#include <stdfix-full-iso.h>

accum test3 (accum x, accum y)
{  return ((bitsk(x) < bitsk(y))? x: y); }
```

Comparisons (III)

- Now including my extra header file:

```
#include <stdfix.h>
#include <stdfix-full-iso.h>

accum test3 (accum x, accum y)
{  return ((bitsk(x) < bitsk(y))? x: y); }
```

- Now we get:

```
00000000 <test3>:
   0:    e1510000        cmp     r1, r0
   4:    b1a00001        movlt   r0, r1
   8:    a1a00000        movge   r0, r0
   c:    e12fff1e        bx      lr
```

Getting a compiler with fixed point support

- Not available with `armcc`.

Getting a compiler with fixed point support

- Not available with `armcc`.
- Can be compiled into `gcc`.

Getting a compiler with fixed point support

- Not available with `armcc`.
- Can be compiled into `gcc`.
- `gcc-4.7.5` available from Code Sourcery (Mentor Graphics).

Getting a compiler with fixed point support

- Not available with `armcc`.
- Can be compiled into `gcc`.
- `gcc-4.7.5` available from Code Sourcery (Mentor Graphics).
- Or roll your own:

Getting a compiler with fixed point support

- Not available with `armcc`.
- Can be compiled into `gcc`.
- `gcc-4.7.5` available from Code Sourcery (Mentor Graphics).
- Or roll your own:
- **Note:** this will not work unless you are compiling for "bare-metal". Don't ask why.

Getting a compiler with fixed point support (II)

```
Target: arm-none-eabi Configured with: ../gcc-4.8.2/configure
 --prefix=/home/dave/Project/x-tools
 --target=arm-none-eabi --disable-shared --disable-nls
 --disable-threads --disable-libssp --with-gcc
 --disable-libstdcxx-pch  --disable-libmudflap
 --disable-libgomp --disable-libquadmath --with-gnu-ld
 --disable-libstdc++-v3 -v --enable-languages=c
 --with-arch=armv5te --with-cpu=arm968e-s
 --with-mode=arm --disable-bootstrap
 --enable-interwork --disable-multilib --with-gnu-as
 --enable-fixed-point  --disable-decimal-float
 --without-long-double-128 --with-dwarf2
 --with-pkgversion='SpiNNaker ArmBall V0.2 (12th December 2013
Thread model: single
gcc version 4.8.2 (GCC 4.8.2 for arm-ball)
```

Next steps

- Provide fixed point support for non-bare metal platforms.

Next steps

- Provide fixed point support for non-bare metal platforms.
- Improve code quality emitted by gcc for fixed point arithmetic.

Next steps

- Provide fixed point support for non-bare metal platforms.
- Improve code quality emitted by gcc for fixed point arithmetic.
- Provide transcendental support.

Next steps: Progress

Platforms Not yet attempted.

Next steps: Progress

Platforms Not yet attempted.

Code Quality As has been seen, an improved library has already been started.

Next steps: Progress

Platforms Not yet attempted.

Code Quality As has been seen, an improved library has already been started.

Transcendentals Kahan's `reciprt` has been implemented.

Why are we still interested in Fix Point?

- Energy efficiency. Two to six times more energy efficient than floating point hardware.

Why are we still interested in Fix Point?

- Energy efficiency. Two to six times more energy efficient than floating point hardware.
- If weights in synaptic matrices are held as 16-bit integer quantities, we can use ARM Neon SIMD units as 8×16 lanes.

Why are we still interested in Fix Point?

- Energy efficiency. Two to six times more energy efficient than floating point hardware.
- If weights in synaptic matrices are held as 16-bit integer quantities, we can use ARM Neon SIMD units as 8×16 lanes.
- But for neural processing we will provide floating point hardware. Neon permits us vectorize with four lanes.