Programming Embedded Systems

Lecture 6
Real-valued data in embedded software

Wednesday Feb 2, 2011

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se

Lecture outline

- Floating-point arithmetic
- Fixed-point arithmetic
- Interval arithmetic

Real values

- Most control systems have to operate on real-valued data
  - Time, position, velocity, currents, ...
- Various kinds of computation
  - Signal processing
  - Solving (differential) equations
  - (Non)linear optimisation
  - ...

Real values (2)

- Safety-critical decisions can depend on accuracy of computations, e.g.
  - Correct computation of braking distances
  - Correct estimate of elevator position + speed
- It is therefore important to understand behaviour and pitfalls of arithmetic

Real values in computers

- Finiteness of memory requires to work with approximations
  - Rational numbers
  - Arbitrary-precision decimal/binary numbers
  - Floating-point numbers
  - Fixed-point numbers
  - Arbitrary-precision integers
  - Bounded integers

Real values in computers (2)

- Most computations involve **rounding**
  - Precise result of computation cannot be represented
  - Instead, the result will be some value close to the precise result (hopefully)
- **Is it possible to draw reliable conclusions from approximate results?**

Machine arithmetic in a nutshell

Idealised arithmetic

- Idealised domain of computation: $\mathbb{R}$
- Idealised operations:

Literals:	$0, 1, 2, 0.1, 0.288, \ldots$
Basic operations:	$+, -, \cdot, /$
Irrational op.:	$\sqrt{\ }$
Transcendental op.:	$\pi, \exp, \sin, \ldots$

  ... precise mathematical definitions
- Algorithms using those operations:
  - Equation solvers, optimisation, ...

Machine arithmetic

- Identify **domain** that can be represented on computers: $D \subseteq \mathbb{R}$
- Sometimes: further un-real values are added to domain D, such as $-0, \infty, -\infty$
- Define **rounding** operation:

$$round : \mathbb{R} \to D$$
$$\text{such that, for all } x \in \mathbb{R},\ |round(x) - x| \text{ is "small"}$$

- **Lift** operations from $\mathbb{R}$ to D ...

Machine arithmetic (2)

- Lift operations from $\mathbb{R}$ to D

for $x, y \in D$:

$$\begin{aligned} x \tilde{+} y &= round(x + y) \\ x \tilde{-} y &= round(x - y) \\ &\vdots \end{aligned}$$

Operation on D

Operation on $\mathbb{R}$

Machine arithmetic (3)

- **Conceptually**: machine operations are *idealised operations + rounding*
- Of course, **practical** implementation directly calculates on D

Floating-point arithmetic

Overview

- Most common way of representing reals on computers
- Normally used: IEEE 754-2008 floats
- Many processors support floats directly (but **most micro-controllers do not**)
- In many cases:
  Fast, robust, convenient method to work with reals

Domain of (Binary) Floats

Sign

Significand (or mantissa)

Exponent

$$D = \left\{ \begin{array}{l} (-1)^s \cdot m \cdot 2^e \mid \\ s \in \{0,1\},\ 0 \le m < 2^M,\ E^- \le e \le E^+ \end{array} \right\} \cup \{0^-, +\infty, -\infty, NaN\}$$

There are two zeroes

Infinities

Not a number

Float parameters

- Size of significand: M bit
- Size of exponent: E bit
  Defined range:

$$\begin{aligned} E^- &= 3 - 2^{E-1} - M \\ E^+ &= 2^{E-1} - M \end{aligned}$$

(two magical exponent values, signal NaN, etc.)

$$D = \left\{ \begin{array}{l} (-1)^s \cdot m \cdot 2^e \mid \\ s \in \{0,1\},\ 0 \le m < 2^M,\ E^- \le e \le E^+ \end{array} \right\} \cup \{-0, +\infty, -\infty, NaN\}$$

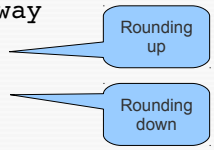
Binary encoding

- IEEE 754-2008 defines binary encoding in altogether $M + E$ bit (saving one bit)
- Typical float types:

	Significant size *M*	Exponent size *E*	Exponent range
binary16	11 bit	5 bit	-24 .. +5
binary32	24 bit	8 bit	-149 .. +104
binary64	53 bit	11 bit	-1074 .. +971
80-bit ext. prec.	65 bit	15 bit	-16446 .. +16319
binary128	113 bit	15 bit	-16494 .. +16271

Rounding operations

- Always round real number to next smaller/greater floating-point number
- 5 rounding modes:
  - `roundNearestTiesToEven`
  - `roundNearestTiesToAway`
  - `roundTowardPositive`
  - `roundTowardNegative`
  - `roundTowardZero`



Examples

- Of course, never compare for equality!

Problems with floating-point arithmetic

Problem 1: performance

- Most micro-controllers don't provide a floating-point unit
  → All computations done in software
- E.g., on CORTEX M3, floating-point operations are more than 10 times slower than integer operations
- **Not an option if performance is important**

Problem 2: mathematical properties

- Floats don't quite behave like reals
  - No associativity: $(x + y) + z \neq x + (y + z)$
  - Density/precision varies
    → often strange for physical data
- Standard transformations of programs/expressions might affect result
- Can exhibit extremely unintuitive behaviour

Problem 3: rounding

- Rounding errors can add up and propagate (problematic when using **numerically unstable algorithms**)
- Usually, default rounding is not directed (round-to-nearest)
- Example (taken from Wikipedia):

```
function sumArray(array) is
    let theSum = 0
    for each element in array do
        let theSum = theSum + element
    end for
    return theSum
 end function
```


Problem 4: inconsistent semantics

- In practice, semantics depends on:
  - Processor (not all are IEEE compliant)
  - Compiler, compiler options
    (optimisations might affect result)
  - Whether data is stored in memory or in registers → register allocation
- For many processors:
  80bit floating-point registers;
  C/IEEE semantics only says 64bit

Problem 4: inconsistent semantics (2)

- Transcendental functions are not even standardised
- Altogether: it's a mess

- **Floats have to be used extremely carefully in safety-critical contexts**

Further reading

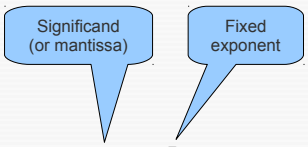
- Pitfalls of floats:
  http://arxiv.org/abs/cs/0701192
- IEEE 754-2008 standard
- More concise definition of floats:
  http://www.philipp.ruemmer.org/publications/smt-fpa.pdf

Fixed-point arithmetic

Overview

- Common alternative to floats in embedded systems
- *Intuitively:* store data as integers, with sufficiently small units
  E.g.
  floats in m ↔ integers in µm
- Performance close to integer arith.
- Uniform density, but smaller range
- Not directly supported in C
  → often has to be implemented by hand

Domain of Fixed-point Arithmetic



$$D_{\text{unsigned}} = \{m \cdot 2^{-P} \mid 0 \leq m < 2^M\}$$
$$D_{\text{signed}} = \{m \cdot 2^{-P} \mid -2^{M-1} \leq m < 2^{M-1}\}$$

- E.g, for $M = 3, P = 1$

$$D_{\text{unsigned}} = \{0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5\}$$

Rounding

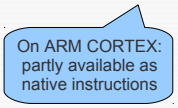
- Normally: rounding down

$$round(x) = \begin{cases} \max\{y \in D \mid y \leq x\} & \text{if } x \geq 0 \\ \text{undef} & \text{if } \neg\exists y \in D.\ y \leq x \end{cases}$$

- Extension to other rounding modes (e.g., rounding up) is possible

Implementation

- Normally:
  - Significand m is stored as integer variable (32bit, 64bit, signed/unsigned)
  - Exponent P is fixed upfront
- Operations:
  Integer operations + shifting



Operations: addition, subtraction

$$\begin{aligned} (m \cdot 2^{-P}) \tilde{+} (n \cdot 2^{-P}) &= (m+n) \cdot 2^{-P} \\ (m \cdot 2^{-P}) \tilde{-} (n \cdot 2^{-P}) &= (m-n) \cdot 2^{-P} \end{aligned}$$

- Simply add/subtract significands
- No rounding
- Over/underflows might occur

Operations: multiplication

$$(m \cdot 2^{-P}) \tilde{\cdot} (n \cdot 2^{-P}) = \big((m \cdot n) \gg P\big) \cdot 2^{-P}$$

- Multiply significands, shift result
- Shifting can cause rounding
- $\gg$: shift with sign-extension

$$x \gg y = \lfloor x \cdot 2^{-y} \rfloor$$

Operations: multiplication (2)

$$(m \cdot 2^{-P}) \tilde{\cdot} (n \cdot 2^{-P}) \quad = \quad \big((m \cdot n) \gg P\big) \cdot 2^{-P}$$

- E.g.: $M = 3, P = 1, \text{unsigned}$

$$\begin{aligned} 0.5 \tilde{\cdot} 2.0 &= 1.0 \\ (1 \cdot 2^{-1}) \tilde{\cdot} (4 \cdot 2^{-1}) &= (1 \cdot 4 \gg 1) \cdot 2^{-1} \end{aligned}$$

$$\begin{aligned} 0.5 \tilde{\cdot} 1.5 &= 0.5 \\ (1 \cdot 2^{-1}) \tilde{\cdot} (3 \cdot 2^{-1}) &= (1 \cdot 3 \gg 1) \cdot 2^{-1} \end{aligned}$$

Rounding!

Operations: multiplication (3)

$$(m \cdot 2^{-P}) \tilde{\cdot} (n \cdot 2^{-P}) \quad = \quad \big((m \cdot n) \gg P\big) \cdot 2^{-P}$$

- *Problem*: with this naïve implementation, overflows can occur during computation even if the result can be represented

$$\begin{aligned} 1.5 \tilde{\cdot} 2.0 &= 3.0 \\ (3 \cdot 2^{-1}) \tilde{\cdot} (4 \cdot 2^{-1}) &= (3 \cdot 4 \gg 1) \cdot 2^{-1} \end{aligned}$$

Intermediate result 12 cannot be stored in 3bit

Operations: division

$$(m \cdot 2^{-P}) \tilde{/} (n \cdot 2^{-P}) \quad = \quad \big((m \ll P) \div n\big) \cdot 2^{-P}$$

- Shift numerator, then divide by denominator
- $\div$: integer division, rounding towards zero
- Same potential problem as with multiplication

Further operations: `pow`, `exp`, `sqrt`, `sin`, ...

- ... can be implemented efficiently using Newton iteration, shift-and-add, or CORDIC

Further reading

- Fixed-point arithmetic on ARM CORTEX: http://infocenter.arm.com/help/topic/com.arm.doc.dai0033a/

In practice ...

- Fixed-point operations are often implemented as macros

Interval methods

Motivation

- Is it possible to draw reliable conclusions from computations done with limited precision?

Motivation (2)

- ***Yes*** ... if we can control the rounding!
- E.g.

Safe if '+' rounds down

```
if (a + b >= 0) {
    // do something dangerous that
    // that only works if a+b >= 0
}
```


Interval methods

- Common in scientific computing, numerical mathematics; rich body of research
- Often an extremely efficient method to obtain estimates on the precise result of computations
- Are sound in the sense that no wrong answers are given, but can fail since error bounds might be too large to be useful

Idealised interval arithmetic

- Domain of real intervals:

$$\mathbb{IR} = \{\boldsymbol{x} = [\underline{x}, \overline{x}] \mid \underline{x} \in \mathbb{R} \cup \{-\infty\}, \overline{x} \in \mathbb{R} \cup \{+\infty\}, \underline{x} \leq \overline{x}\}$$

- Mathematical operations are lifted to intervals, e.g.:

$$\begin{aligned} \boldsymbol{x} + \boldsymbol{y} &= [\underline{x} + \underline{y},\ \overline{x} + \overline{y}] \\ \boldsymbol{x} - \boldsymbol{y} &= [\underline{x} - \overline{y},\ \overline{x} - \underline{y}] \end{aligned}$$

- Generally defined such that:

$$\boldsymbol{x}\ \mathrm{op}\ \boldsymbol{y}\ \supseteq\ \{x\ \mathrm{op}\ y \mid x \in \boldsymbol{x}, y \in \boldsymbol{y}\}$$

Machine interval arithmetic

- Intervals represented using two machine numbers (floating-point or fixed-point):

$$\mathbb{I}D = \{\boldsymbol{x} = [\underline{x}, \overline{x}] \mid \underline{x} \in D \cup \{-\infty\}, \overline{x} \in D \cup \{+\infty\}, \underline{x} \leq \overline{x}\}$$

- Operations are used *rounding outward:*

$$\boldsymbol{x} \mathbin{\tilde{+}} \boldsymbol{y} \quad = \quad [round_{\downarrow}(\underline{x} + \underline{y}),\ round_{\uparrow}(\overline{x} + \overline{y})]$$

- This ensures that the precise result is always within the computed machine interval

Example

- Fixed-point arithmetic with rounding:

$$\begin{aligned} 0.5 \mathbin{\tilde{\cdot}} 1.5 &= 0.5 \\ (1 \cdot 2^{-1}) \mathbin{\tilde{\cdot}} (3 \cdot 2^{-1}) &= (1 \cdot 3 \gg 1) \cdot 2^{-1} \end{aligned}$$

- In interval arithmetic:

$$\begin{aligned} [0.5, 0.5] \mathbin{\tilde{\cdot}} [1.5, 1.5] &= [round_{\downarrow}(0.75),\ round_{\uparrow}(0.75)] \\ &= [0.5, 1.0] \end{aligned}$$

Inputs are singleton intervals

Output bounds the precise result

Further reading

- Library on interval methods: http://www.cs.utep.edu/interval-comp/