Computer Systems Architecture

http://cs.nott.ac.uk/~txa/g51csa/

Thorsten Altenkirch and Liyang Hu

School of Computer Science
University of Nottingham

Lecture 08: Real Numbers and IEEE 754 Arithmetic

Representing Real Numbers

- So far we can compute with a subset of numbers:
  Unsigned integers $[0, 2^{32}) \subset \mathbb{N}$
  Signed integers $[-2^{31}, 2^{31}) \subset \mathbb{Z}$
- What about numbers such as
  - $3.1415926\ldots$
  - $1/3 = 0.33333333\ldots$
  - 299792458
- How do we represent smaller quantities than 1?

The University of Nottingham

Shifting the Point

- Use *decimal point* to separate integer and fractional parts
  - Digits after the point denote $1/10^{\text{th}}$, $1/100^{\text{th}}$, ...
- Similarly, we can use the *binary point*

Bit	3^{rd}	2^{nd}	1^{st}	0^{th}	.	-1^{st}	-2^{nd}	-3^{rd}	-4^{th}
Weight	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}

  - Left denote integer weights; right for fractional weights
- What is 0.101_2 in decimal?
  - $0.101_2 = 2^{-1} + 2^{-3} = 0.5 + 0.125 = 0.625$
- What is 0.1_{10} in binary?
  - $0.1_{10} = 0.0001100110011\ldots_2$
  - Digits repeat forever – no exact finite representation!

The University of Nottingham

Fixed Point Arithmetic

- How do we represent non-integer numbers on computers?
- Use binary scaling: e.g. let one increment represent $1/16^{\text{th}}$

  Hence, $\mathtt{0000.0001}_2 = 1 \cong 1/16 = 0.0625$

  $\mathtt{0001.1000}_2 = 24 \cong 24/16 = 1.5$
- *Fixed* position for binary *point*
- Addition same as integers: $a/c + b/c = (a+b)/c$
- Multiplication mostly unchanged, but must *scale* after

$$\frac{a}{c} \times \frac{b}{c} = \frac{(a \times b)/c}{c}$$

  - e.g. $1.5 \times 2.5 = 3.75$

    $\mathtt{0001.1000}_2 \times \mathtt{0010.1000}_2 = \mathtt{0011.1100\,0000}_2$
  - Thankfully division by 2^e is fast!

The University of Nottingham

Overview of Fixed Point

- Integer arithmetic is simple and fast
  - Games often used fixed point for performance;
  - Digital Signal Processors still do, for accuracy
  - No need for a separate floating point coprocessor
- Limited range
  - A 32-bit Q24 word can only represent $[-2^7, 2^7 - 2^{-24}]$
  - Insufficient range for physical constants, e.g.

Speed of light	$c = 2.9979246 \times 10^9 \; m/s$
Planck's constant	$h = 6.6260693 \times 10^{-34} \; N{\cdot}m{\cdot}s$
- Exact representation only for (multiples of) powers of two
  - Cannot represent certain numbers exactly, e.g. 0.01 for financial calculations

Scientific Notation

- We use *scientific notation* for very large/small numbers
  - e.g. 2.998×10^9, 6.626×10^{-34}
- General form $\pm m \times b^e$ where
  - The *mantissa* contains a decimal point
  - The *exponent* is an integer $e \in \mathbb{Z}$
  - The *base* can be any positive integer $b \in \mathbb{N}^*$
- A number is *normalised* when $1 \leq m < b$
  - Normalise a number by adjusting the exponent e
  - 10×10^0 is not normalised; but 1.0×10^1 is
- In general, which number cannot be normalised?
  - Zero can never be normalised
  - *NaN* and $\pm\infty$ also considered *denormalised*

Floating Point Notation

- Floating point – scientific notation in base 2
- In the past, manufacturers had incompatible hardware
    - Different bit layouts, rounding, and representable values
    - Programs gave different results on different machines
- IEEE 754: Standard for Binary Floating-Point Arithmetic
    - IEEE – Institute of Electrical and Electronic Engineers
    - Exact definitions for arithmetic operations – *the same wrong answers, everywhere*
- IEEE Single (32-bit) and Double (64-bit) precision
    - Gives exact layout of bits, and defines basic arithmetic
- Implemented on coprocessor 1 of the MIPS architecture
- Part of Java language definition

The University of Nottingham

IEEE 754 Floating Point Format

- Computer representations are *finite*
- IEEE 754 is a *sign and magnitude* format
  - Here, magnitude consists of the exponent and mantissa

Single Precision (32 bits)

sign	exponent	mantissa
1 bit	← 8 bits →	← 23 bits →

Double Precision (64 bits)

sign	exponent	mantissa
1 bit	← 11 bits →	← 52 bits →

IEEE 754 Encoding (Single Precision)

- Sign: 0 for positive, 1 for negative
- Exponent: 8-bit *excess*-127, between $\texttt{01}_{16}$ and $\texttt{FE}_{16}$
  - $\texttt{00}_{16}$ and $\texttt{FF}_{16}$ are *reserved* – see later
- Mantissa: 23-bit binary fraction
  - No need to store leading bit – the *hidden bit*
  - Normalised *binary* numbers always begin with 1
  - c.f. normalised decimal numbers begin with 1...9
- Reading the fields as unsigned integers,

$$(-1)^s \times 1.m \times 2^{e-127}$$

- Special numbers contain $\texttt{00}_{16}$ or $\texttt{FF}_{16}$ in exponent field
  - $\pm\infty$, *NaN*, 0 and some very small values

IEEE 754 Special Values

- Largest and smallest *normalised* 32-bit number?
  - Largest: $1.1111\ldots_2 \times 2^{127} \approx 3.403 \times 10^{38}$
  - Smallest: $1.000\ldots_2 \times 2^{-126} \approx 1.175 \times 10^{-38}$
- We can still represent some values less than 2^{-126}
  - *Denormalised* numbers have 00_{16} in the exponent field
  - The *hidden bit* no longer read as 1
  - Conveniently, $0000\,0000_{16}$ represents (positive) zero

IEEE 754 Single Precision Summary

Exponent	Mantissa	Value	Description
00_{16}	$= 0$	0	Zero
00_{16}	$\neq 0$	$\pm 0.m \times 2^{-126}$	Denormalised
01_{16} to FE_{16}		$\pm 1.m \times 2^{e-127}$	Normalised
FF_{16}	$= 0$	$\pm\infty$	Infinities
FF_{16}	$\neq 0$	*NaN*	Not a Number

Overflow, Underflow, Infinities and NaN

- Underflow: result $<$ smallest normalised number
- Overflow: result $>$ largest representable number
- Why do we want infinities and *NaN*?
  - Alternative is to give a wrong value, or raise an exception
  - Overflow gives $\pm\infty$ (underflow gives denormalised)
- In long computations, only a few results may be wrong
  - Raising exceptions would abort entire process
  - Giving a misleading result is just plain dangerous
- Infinities and *NaN* propagate through calculations, e.g.
  - $1 + (1 \div 0) = 1 + \infty = \infty$
  - $(0 \div 0) + 1 = NaN + 1 = NaN$

The University of Nottingham

Examples

- Convert the following to 32-bit IEEE 754 format
  - $1.0_{10} = 1.0_2 \times 2^0 = \boxed{0}\boxed{0111\,1111}\boxed{00000\ldots}$
  - $1.5_{10} = 1.1_2 \times 2^0 = \boxed{0}\boxed{0111\,1111}\boxed{10000\ldots}$
  - $100_{10} = 1.1001_2 \times 2^6 = \boxed{0}\boxed{1000\,0101}\boxed{10010\ldots}$
  - $0.1_{10} \approx 1.10011_2 \times 2^{-4} = \boxed{0}\boxed{0111\,1011}\boxed{100110\ldots}$
- Check your answers using http://www.h-schmidt.net/FloatApplet/IEEE754.html
- Convert to a hypothetical 12-bit format, 4 bits excess-7 exponent, 7 bits mantissa:
  - $3.1416_{10} \approx 1.1001001_2 \times 2^1 = \boxed{0}\boxed{1000}\boxed{1001001}$

The University of Nottingham

Floating Point Addition

- Suppose $f_0 = m_0 \times 2^{e_0}$, $f_1 = m_1 \times 2^{e_1}$ and $e_0 \geq e_1$
  - Then $f_0 + f_1 = (m_0 + m_1 \times 2^{e_1 - e_0}) \times 2^{e_0}$

1. Shift the smaller number right until exponents match
2. Add/subtract the mantissas, depending on sign
3. Normalise the sum by adjusting exponent
4. Check for overflow
5. Round to available bits
6. Result may need further normalisation; if so, goto step 3

Floating Point Multiplication

- Suppose $f_0 = m_0 \times 2^{e_0}$ and $f_1 = m_1 \times 2^{e_1}$
  - Then $f_0 \times f_1 = m_0 \times m_1 \times 2^{e_0+e_1}$

1. Add the exponents (be careful, excess-n encoding!)
2. Multiply the mantissas, setting the sign of the product
3. Normalise the product by adjusting exponent
4. Check for overflow
5. Round to available bits
6. Result may need further normalisation; if so, goto step 3

IEEE 754 Rounding

- Hardware needs two extra bits (round, guard) for rounding
- IEEE 754 defines four rounding modes

  Round Up Always toward $+\infty$

  Round Down Always toward $-\infty$

  Towards Zero Round down if positive, up if negative

  Round to Even Rounds to nearest even value: in a tie, pick the closest ‘even’ number: e.g. 1.5 rounds to 2.0, but 4.5 rounds to 4.0

- MIPS and Java uses *round to even* by default

Exercise: Rounding

- Round off the last two digits from the following
  - Interpret the numbers as 6-bit sign and magnitude

Number	To $+\infty$	To $-\infty$	To Zero	To Even
+0001.01	+0010	+0001	+0001	+0001
−0001.11	−0001	−0010	−0001	−0010
+0101.10	+0110	+0101	+0101	+0110
+0100.10	+0101	+0100	+0100	+0100
−0011.10	−0011	−0100	−0011	−0100

- Give 2.2 to two bits after the binary point: 10.01_2
- Round 1.375 and 1.125 to two places: 1.10_2 and 1.00_2

The University of Nottingham