

ELEC2041

Microprocessors and Interfacing

Lectures 20: Floating Point Number Representation – II

<http://webct.edtec.unsw.edu.au/>

April, 2006

Saeid Nooshabadi

saeid@unsw.edu.au

ELEC2041 lec20-fp-II.1

Saeid Nooshabadi

Overview

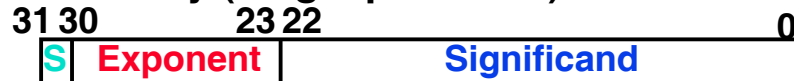
- IEEE – 754 Standard
 - Implied Hidden 1
 - Representation for 0
- Decimal to Floating Point conversion, and vice versa
- Big Idea: Type is not associated with data
- ARM floating point instructions, registers

ELEC2041 lec20-fp-II.2

Saeid Nooshabadi

Review: IEEE 754 Fl. Pt. Standard (#1/2)

- Summary (single precision):



1 bit 8 bits

23 bits

- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023

- Hidden 1 (implied)

ELEC2041 lec20-fp-II.3

Saeid Nooshabadi

Review: IEEE 754 Fl. Pt. Standard (#2/2)

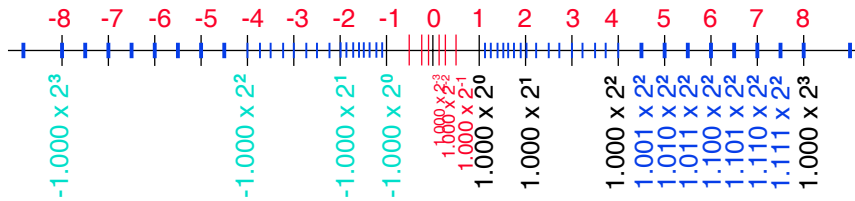
- Scientific notation in binary!
 $\pm 1.F \times 2^{\pm e}$
- Sign Magnitude for the fixed part
 $(-1)^S \times 1.F \times 2^{\pm e}$
- Hidden 1 of the significand
 $(-1)^S \times 1.\text{fffffffffffffffffffffff} \times 2^{\pm e}$
- Excess notation for the exponent
 $(-1)^S \times 1.\text{fffffffffffffffffffffff} \times 2^{\text{exp} - 127}$
- Ordering the fields so integer compare works on FP

ELEC2041 lec20-fp-II.4

Saeid Nooshabadi

Review: Floating Point Number Distribution

• Which numbers can be represented?



Using mantissa of 1.000 and positive exponents

and Sign bit

and negative exponents

in each of the intervals of exponentially increasing size can represent 2^s ($s=3$ here) numbers of uniform difference

But how do we represent 0?

Representing the Significand Fraction

◦ In normalized form, mantissa is either:
1.xxx xxxx xxxx xxxx xxx
or
0.000 0000 0000 0000 0000 000 (for Zero)

◦ Trick: If hardware automatically places 1 in front of binary point of normalized numbers, then get 1 more bit for the fraction, increasing accuracy “for free”

Hidden Bit 1.xxx xxxx xxxx xxxx xxx becomes
→ (1).xxx xxxx xxxx xxxx xxx

• Comparison OK; “subtracting” 1 from both

How differentiate from Zero in Trick Format?

◦ .0000 ... 000 => . 0000 ... 0000

◦ .0000 ... 000 => . 0000 ... 0000

◦ Solution: Reserve most negative (value 0) exponent to be only used for Zero; rest are normalized so prepend an implied 1

◦ Convention is $(-Max = -127 \rightarrow Exp=00000000$

in biased notation

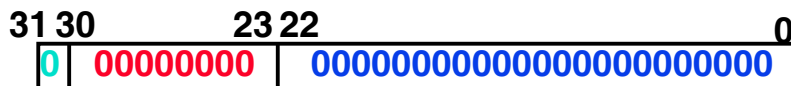


→ 0.00000

It's NOT 1.00000×2^{-127}



→ $1.00000 \times 2^{Exp - 127}$



1 bit 8 bits

23 bits

Understanding the Significand (#1/2)

◦ Method 1 (Fractions):

• In decimal: $0.340_{10} \Rightarrow 340_{10}/1000_{10}$
 $\Rightarrow 34_{10}/100_{10}$

• In binary: $0.110_2 \Rightarrow 110_2/1000_2 = 6_{10}/8_{10}$
 $\Rightarrow 11_2/100_2 = 3_{10}/4_{10}$

• Advantage: less purely numerical, more thought oriented; this method usually helps people understand the meaning of the significand better

Understanding the Significand (#2/2)

Method 2 (Place Values):

- Convert from scientific notation
- In decimal: $1.6732 = (1 \times 10^0) + (6 \times 10^{-1}) + (7 \times 10^{-2}) + (3 \times 10^{-3}) + (2 \times 10^{-4})$
- In binary: $1.1001 = (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
- Interpretation of value in each position extends beyond the decimal/binary point
- Advantage: good for quickly calculating significand value; use this method for translating FP numbers

Example: Converting Binary FP to Decimal

0 0110 1000 101 0101 0100 0011 0100 0011

- **Sign:** 0 => positive
- **Exponent:**
 - $0110\ 1000_{\text{two}} = 104_{\text{ten}}$
 - Bias adjustment: $104 - 127 = -23$
- **Significand:**
 - $1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + \dots$
 $= 1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-7} + 2^{-9} + 2^{-14} + 2^{-15} + 2^{-17} + 2^{-22} + 2^{-23}$
 $= 1.0 + 0.6661175$
- **Represents:** $1.6661175_{\text{ten}} \times 2^{-23} \sim 1.986 \times 10^{-7}$
 (about 2/10,000,000)

Continuing Example: Binary to ???

0011 0100 0101 0101 0100 0011 0100 0011

- **Convert 2's Comp. Binary to Integer:**
 $2^{29} + 2^{28} + 2^{26} + 2^{22} + 2^{20} + 2^{18} + 2^{16} + 2^{14} + 2^9 + 2^8 + 2^6 + 2^1 + 2^0$
 $= 878,003,011_{\text{ten}}$
- **Convert Binary to Instruction:**

0011	01000101	01010100	00110100	0011
3	1000101	5	4	835

`ldrbl r4, [r5], #-835`
- **Convert Binary to ASCII:**

0011 0100	0101 0101	0100 0011	0100 0011
4	U	C	C

Big Idea: Type not associated with Data

0011 0100 0101 0101 0100 0011 0100 0011

- **What does bit pattern mean:**
 - 1.986×10^{-7} ? 878,003,011? "4UCC"?
 - `ldrbl r4, [r5], #-835`
- **Data can be anything; operation of instruction that accesses operand determines its type!**
 - Side-effect of stored program concept: instructions stored as numbers
- **Power/danger of unrestricted addresses/pointers: use ASCII as Fl. Pt., instructions as data, integers as instructions, ...**
 (Leads to security holes in programs)

Converting Decimal to FP (#1/3)

- Simple Case: If denominator is an exponent of 2 (2, 4, 8, 16, etc.), then it's easy.
- Show IEEE 754 representation of -0.75
 - $-0.75 = -3/4$
 - $-11_{\text{two}}/100_{\text{two}} = -0.11_{\text{two}}/1.00_{\text{two}} = -0.11_{\text{two}}$
 - Normalized to $-1.1_{\text{two}} \times 2^{-1}$
 - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
Exponent-127 = -1 \rightarrow Exponent = 126
 - $(-1)^1 \times (1 + .100\ 0000 \dots 0000) \times 2^{(126-127)}$

1	0111 1110	100 0000 0000 0000 0000 0000
---	-----------	------------------------------

Converting Decimal to FP (#2/3)

- Not So Simple Case: If denominator is not an exponent of 2.
 - Then we can't represent number precisely, but that's why we have so many bits in significand: for precision
 - Once we have significand, normalizing a number to get the exponent is easy.
 - So how do we get the Significand of a never ending number?

Converting Decimal to FP (#3/3)

- Fact: All rational numbers have a repeating pattern when written out in decimal (eg $1/7 = 0.142857142857\dots$)
- Fact: This still applies in binary as well (eg $1/111 = 0.001001001\dots$)
- To finish conversion:
 - Write out binary number with repeating pattern.
 - Cut it off after correct number of bits (different for single vs double precision).
 - Derive Sign, Exponent and Significand fields.

Hairy Example (#1/2)

- How to represent $1/3$ in IEEE 754?
- $1/3$
 - = $0.33333\dots_{10}$
 - = $0.25 + 0.0625 + 0.015625 + 0.00390625 + 0.0009765625 + \dots$
 - = $1/4 + 1/16 + 1/64 + 1/256 + 1/1024 + \dots$
 - = $2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10} + \dots$
 - = $0.0101010101\dots_2 \times 2^0$
 - = $1.0101010101\dots_2 \times 2^{-2}$ (Normalized)

Hairy Example (#2/2)

- $1/3 = 1.0101010101..._2 * 2^{-2}$
- Sign: 0
- Exponent = $-2 + 127 = 125_{10} = 01111101_2$
- Significand = 01010101...

0 0111 1101 0101 0101 0101 0101 0101 010

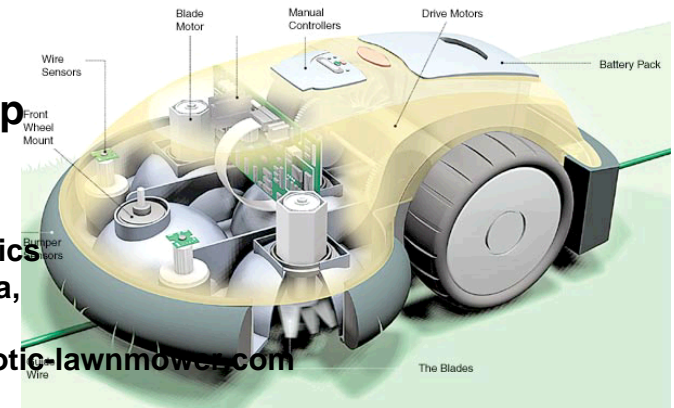
What's this stuff good for? Mow Lawn?

- Robot lawn mower: "Robomow RL-800"
- Surround lawn, trees with perimeter wire
- Senses tall grass to spin blades faster: up to 5800 RPM

- Slows if senses object, stop if bumps

◦ US\$700

Friendly Robotics of Even Yehuda, Israel,
<http://www.robotic-lawnmower.com>



Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow.
- Why?
 - OK to do further computations with infinity
 e.g., $\arccos(x) = 2 * \arctan[(1-x)/(1+x)]$ for $x \rightarrow -1$ gives $2 * \arctan[2/0] = 2 * \arctan(\infty) = \pi$: (a valid Operation)
 or $1/(1/x)$ for $x \rightarrow 0$ gives $1/\infty = 0$: (a valid Operation)
 or, $x/0 > Y$ may be a valid comparison (Ask math prof.)
- IEEE 754 represents +/- infinity
 - Most positive exponent reserved for infinity: Exp = 255
 - Significands all zeroes: Significand = 0000...

Two Representation for 0!

- Represent 0?
 - exponent all zeroes
 - significand all zeroes too
 - What about sign?
 - +0: 0 00000000 000000000000000000000000
 - -0: 1 00000000 000000000000000000000000
- Why two zeroes?
 - Helps in some limit comparisons
 - Ask math prof.

Special Numbers

◦ What have we defined so far? (Single Precision)

Exponent	Significand	Object
0	0	0
0	<u>nonzero</u>	???
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	<u>nonzero</u>	???

◦ Professor Kahan had clever ideas; “Waste not, want not”

- We will talk about Exp=**0,255** & Significand **!=0** later

Recall: arithmetic in scientific notation

Addition:

- $3.2 \times 10^4 + 2.3 \times 10^3 \rightarrow$ common exp
- $= 3.2 \times 10^4 + 0.23 \times 10^4 \rightarrow$ add
- $= 3.43 \times 10^4 \rightarrow$ normalize and round
- $\sim 3.4 \times 10^4$

Multiplication:

- $3.2 \times 10^4 \times 2.3 \times 10^5$
- $= 3.2 \times 2.3 \times 10^9 = 7.36 \times 10^9 \sim 7.4 \times 10^9$

Basic Fl. Pt. Addition Algorithm

- Much more difficult than with integers
 - For addition (or subtraction) of X to Y (X<Y):
 - (1) Compute $D = \text{Exp}_Y - \text{Exp}_X$ (align binary point)
 - (2) Right shift $(1+\text{Sig}_X)$ D bits
 $\rightarrow ((1+\text{Sig}_X) \gg D) * 2^{(\text{Exp}_X - \text{Exp}_Y)}$
 - (3) Compute $((1+\text{Sig}_X) \gg D) * 2^{(\text{Exp}_X - \text{Exp}_Y)} + (1+\text{Sig}_Y)$
- Normalize if necessary; continue until MS bit is 1
- (4) Too small (e.g., 0.001xx...) left shift result, decrement result exponent
 - (4') Too big (e.g., 101.1xx...) right shift result, increment result exponent
 - (5) If result significand is 0, set exponent to 0

FP Addition/Subtraction Problems

- Problems in implementing FP add/sub:
 - If signs differ for add (or same for sub), what will be the sign of the result?
- Question: How do we integrate this into the integer arithmetic unit?
- Answer: We don't!
- We need a separate Ft. Pt. Hardware (co-processor) (Why?)

ARM's Floating Point Architecture (#1/4)

- **Separate floating point instructions:**
 - **Single Precision:**
fcmps, fadds, fsubs, fmuls, fdivs
 - **Double Precision:**
fcmpd, fadd, fsubd, fmuld, fdivd
- **These instructions are far more complicated than their integer counterparts, so they can take much longer.**
- **So, What does complication mean for our hardware?**

ARM's Floating Point Architecture (#2/4)

- **Observations:**
 - **It's inefficient to have different instructions take vastly differing amounts of time.**
 - **Generally, a particular piece of data will not change from FP to int, or vice versa, within a program. So only one type of instruction will be used on it.**
 - **Some programs do no floating point calculations**
 - **It takes lots of hardware relative to integers to do Floating Point fast**

ARM Floating Point Architecture (#3/4)

- **ARM Solution: Make completely separate Co-processors that handles only IEEE-754 FP.**
- **Coprocessors 10 (for SP) & 11 (for DP):**
 - **Actually a Single hardware used differently for Single Precision and Double Precision**
 - **contains 32 32-bit registers: s0 - s31**
 - **Arithmetic instructions use this register set**
 - **separate load and store: flds and fsts**
("Float load Single Coprocessor 10", "Float Store ...")
 - **Double Precision: even/odd pair overlap one DP**
FP number: s0/s1 = d0, s2/s3 = d1, ..., s30/s31 = d15
 - **separate double load and store: fldd and fstd**
("Float load Double Coprocessor 11", "Float Store ...")

ARM Floating Point Architecture (#4/4)

- **ARM Solution:**
 - **Processor: handles all the normal stuff**
 - **Coprocessors 10 & 11: handles FP and only FP;**
 - **more coprocessors?... Yes, later**
 - **Today, cheap chips may leave out FP HW**
(Example: Chip on Lab's DSLMU Board)
 - **Use Ft. Pt. Emulator:**
 - **Ref: Ft. Pt. Emulator: A student mini project on:**
<http://dsl.ee.unsw.edu.au/unsw/projects/armvfp/README.html>
- **Instructions to move data between main processor and coprocessors:**
 - **fmsr (Sn = Rd), fmrs (Rd = Sn), etc.**
- **Check ARM instruction reference manual on CD-ROM for many, many more FP operations.**

“In Conclusion...”

- **Floating Point numbers *approximate* values that we want to use.**
- **IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers**
- **New ARM registers(*s0-s31*), instruct.:**
 - **Single Precision (32 bits, 2×10^{-38} ... 2×10^{38}):**
`fcmps, fadds, fsubs, fmul, fdivs`
 - **Double Precision (64 bits, 2×10^{-308} ... 2×10^{308}):**
`fcmpd, fadd, fsubd, fmuld, fdivd`
- **Type is not associated with data, bits have no meaning unless given in context**