

# ELEC2041

## Microprocessors and Interfacing

### Lectures 21: Floating Point Number Representation – III

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

April 2006

Saeid Nooshabadi

saeid@unsw.edu.au

ELEC2041 lec21-fp-III.1

Saeid Nooshabadi

# Overview

- Special Floating Point Numbers: NaN, Denorms
- IEEE Rounding modes
- Floating Point fallacies, hacks
- Using floating point in C and ARM
- Multi Dimensional Array layouts

ELEC2041 lec21-fp-III.2

Saeid Nooshabadi

## Review: ARM FI. Pt. Architecture

- Floating Point Data: approximate representation of very large or very small numbers in 32-bits or 64-bits
- 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,  $2 \times 10^{-38}$ ...  $2 \times 10^{38}$ ): fcmps, fadds, fsubs, fmul, fdivs
  - Double Precision (64 bits,  $2 \times 10^{-308}$ ...  $2 \times 10^{308}$ ): fcmpd, fadd, fsubd, fmuld, fdivd

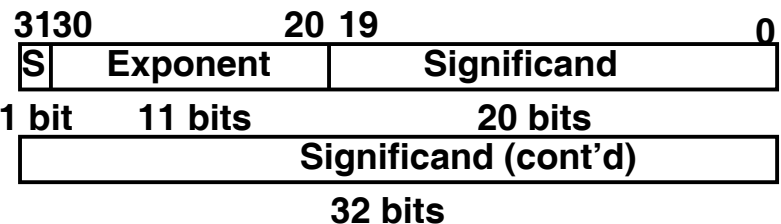
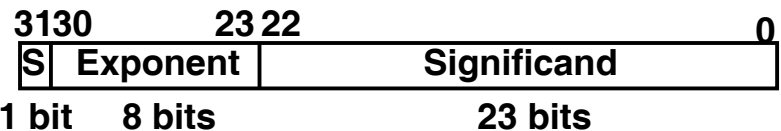
**Big Idea: Instructions determine meaning of data; nothing inherent inside the data**

ELEC2041 lec21-fp-III.3

Saeid Nooshabadi

## Review: Floating Point Representation

- Single Precision and Double Precision



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

ELEC2041 lec21-fp-III.4

Saeid Nooshabadi

## New ARM arithmetic instructions

- | Example        | Meaning               | Comments                 |
|----------------|-----------------------|--------------------------|
| fadds s0,s1,s2 | s0=s1+s2              | Fl. Pt. Add (single)     |
| fadd d0,d1,d2  | d0=d1+d2              | Fl. Pt. Add (double)     |
| fsubs s0,s1,s2 | s0=s1 - s2            | Fl. Pt. Sub (single)     |
| fsubd d0,d1,d2 | d0=d1 - d2            | Fl. Pt. Sub (double)     |
| fmuls s0,s1,s2 | s0=s1 × s2            | Fl. Pt. Mul (single)     |
| fmuld d0,d1,d2 | d0=d1 × d2            | Fl. Pt. Mul (double)     |
| fdivs s0,s1,s2 | s0=s1 ÷ s2            | Fl. Pt. Div (single)     |
| fdivd d0,d1,d2 | d0=d1 ÷ d2            | Fl. Pt. Div (double)     |
| fcmps s0,s1    | FCPSR flags = s0 - s1 | Fl. Pt. Compare (single) |
| fcmpd d0,d1    | FCPSR flags = d0 - d1 | Fl. Pt. Compare (double) |

Z = 1 if s0 = s1, (d0 = d1)

N = 1 if s0 < s1, (d0 < d1)

C = 1 if s0 = s1, (d0 = d1); s0 > s1, (d0 > d1), or unordered

V = 1 if unordered

Unordered? Next slide

## 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"

## Representation for Not a Number

- What do I get if I calculate  $\sqrt{-4.0}$  or  $0/0$ ?

- If infinity is not an error, these shouldn't be either.

- Called Not a Number (NaN)

- Exponent = 255, Significand nonzero

- Why is this useful?

- Hope NaNs help with debugging?

- They contaminate:  $\text{op}(\text{NaN}, X) = \text{NaN}$

- OK if calculate but don't use it

- cmp s1, s2 produces unordered results if either is an NaN**

## Special Numbers (cont'd)

- 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	nonzero	NaN

## Representation for Denorms (#1/2)

◦ Problem: There's a gap among representable FP numbers around 0

- **Significand = 0, Exp = 0 ( $2^{-127}$ )  $\rightarrow$  0**
- Smallest representable positive num:
  - $a = 1.0..._2 * 2^{-126} = 2^{-126}$
- Second smallest representable positive num:
  - $b = 1.000.....1_2 * 2^{-126} = 2^{-126} + 2^{-149}$
- $a - 0 = 2^{-126}$
- $b - a = 2^{-149}$



ELEC2041 lec21-fp-III.9

Saeid Nooshabadi

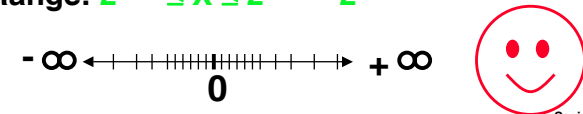
## Representation for Denorms (#2/2)

◦ Solution:

- We still haven't used Exponent = 0, Significand nonzero
- Denormalized number: no leading 1
- Smallest representable pos num:
  - $a = 2^{-149}$
- Second smallest representable pos num:
  - $b = 2^{-148}$

Meaning:  $(-1)^S \times (0 + \text{Significand}) \times 2^{(-126)}$

Range:  $2^{-149} \leq X \leq 2^{-126} - 2^{-149}$



ELEC2041 lec21-fp-III.10

Saeid Nooshabadi

## Special Numbers

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

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

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

ELEC2041 lec21-fp-III.11

Saeid Nooshabadi

## Clever Idea and Hardware Implementation

◦ 0 nonzero Denorm

- Very Clever Idea by Prof Kahan
- **BUT** such corner cases make the hardware design very complex
- Good idea but hard practice!
- Even software emulation not easy.
- In Ref. Ft. Pt. Emulator: **A student mini project on:**  
<http://dsl.ee.unsw.edu.au/unsw/projects/armvfp/README.html>
- 25% - 30% of the code is to get the operations on denorms right
- In most hardware implementations denorms are flushed to zero, or implemented in software via exceptions

ELEC2041 lec21-fp-III.12

Saeid Nooshabadi

## Rounding

- When we perform math on real numbers, we have to worry about rounding
- The actual hardware for Floating Point Representation carries two extra bits of precision, and then round to get the proper value
- Rounding also occurs when converting a double to a single precision value, or converting a floating point number to an integer

## IEEE Rounding Modes

- Round towards +infinity
  - ALWAYS round “up”: 2.2001 → 2.3
  - -2.3001 → -2.3
- Round towards -infinity
  - ALWAYS round “down”: 1.9999 → 1.9,
  - -1.9999 → -2.0
- Truncate
  - Just drop the last digits (round towards 0);
  - 1.9999 → 1.9, -1.9999 → -1.9
- Round to (nearest) even
  - Normal rounding, almost

## Round to Even

- Round like you learned in high school
- Except if the value is right on the borderline, in which case we round to the nearest EVEN number
  - 2.55 → 2.6
  - 3.45 → 3.4
- Insures fairness on calculation
  - This way, half the time we round up on tie, the other half time we round down
  - Ask statistics Prof.
- This is the default rounding mode

## Casting floats to ints and vice versa in C

- `(int) exp`
  - In C float to int type casting coerces and converts it to the an integer by truncation (**rounds to towards 0**)
  - affected by rounding modes
  - `i = (int) (3.14159 * f);`
  - **`fsitos` (floating → int) In ARM to round to a selected mode (default nearest)**
  - **`fsitozs` (floating → int) In ARM to round towards zero**
- `(float) exp`
  - converts integer to nearest floating point
  - `f = f + (float) i;`
  - **`fsitos` (int → floating) In ARM**

## int → float → int

```
if (i == (int)((float) i)) {  
    printf("true");  
}
```

- Will not always work
- Large values of integers don't have exact floating point representations
- Similarly, we may round to the wrong value

## float → int → float

```
if (f == (float)((int) f)) {  
    printf("true");  
}
```

- Will not always work
- Small values of floating point don't have good integer representations
- Also rounding errors

## Ints, Fractions and rounding in C

- What do you get?

```
{ int x = 3/2;    int y = 2/3;  
  printf("x: %d, y: %d", x, y); }
```

- How about?

```
int cela = ((fahr - 32) * 5) / 9;  
int celb = (5 / 9) * (fahr - 32)  
float celc = (5.0 / 9.0) * (fahr - 32);
```

```
fahr = 60 =>  
cela: 15,  
celb: 0,  
celc: 15.55556
```

## Floating Point Fallacy

- FP Add, subtract associative: FALSE!

$$X + (Y + z) = (X + y) + Z$$

•  $x = -1.5 \times 10^{38}$ ,  $y = 1.5 \times 10^{38}$ , and  $z = 1.0$

•  $x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0)$   
 $= -1.5 \times 10^{38} + (1.5 \times 10^{38}) = \underline{0.0}$

•  $(x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0$   
 $= (0.0) + 1.0 = \underline{1.0}$

- Therefore, Floating Point add, subtract are not associative!

• Why? FP result approximates real result!

• In this example:  $1.5 \times 10^{38}$  is so much larger than 1.0 that  $1.5 \times 10^{38} + 1.0$  in floating point representation is still  $1.5 \times 10^{38}$

## Floating Point In the News!

- **July 1994: Intel discovers bug in Pentium**
  - Occasionally affects bits 12-52 of D.P. divide
  - The bug was introduced when they optimized divide unit to run much faster. They ignored some rare corner cases
- **Sept: Math Prof. discovers, puts on WWW**
- **Nov: Front page trade paper, then NY Times**
  - Intel: "several dozen people that this would affect. So far, we've only heard from one."
  - Intel claims customers see 1 error/27000 years for random set of Ft. Pt. Inputs.
  - Does not explain why anybody wants to use Ff. Pt. No. in random
  - IBM claims 1 error/month, stops shipping
- **Dec: Intel apologizes, replace chips \$300M**

ELEC2041 lec21-fp-III.21

Saeid Nooshabadi

## IEEE 754 Floating Point Issues

- **It is complex, involves lots of details**
  - We just scratched the surface
- **Check for gradual underflow and treating denoms makes it much harder**
- **Beyond Prof. Kahan very few really understand it!**
- **It was finally approved as IEEE 754 after 10 years of controversy in 1983**
  - Denorm was the most controversial aspect
  - The visitors to the US were advised of 3 most interesting places to visit: Las Vegas, Great Canyon and **IEEE committee rooms!**

ELEC2041 lec21-fp-III.22

Saeid Nooshabadi

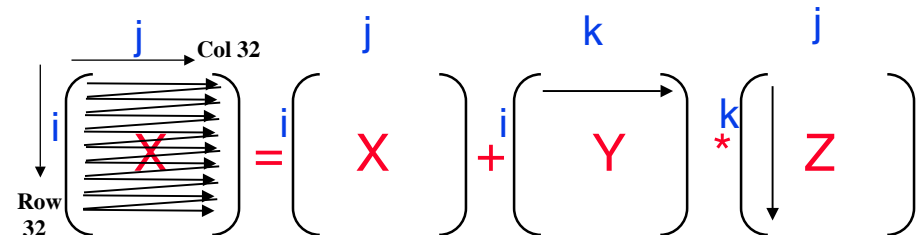
## Reading Material

- **ARM Architecture Reference Manual 2<sup>nd</sup> Ed, Addison-Wesley, 2001, ISBN: 0-201-73719-1, Part C, Vector Floating Point Architecture, chapters C1 – C5**
- **Ft. Pt. Emulator: A student mini project on:**  
<http://dsl.ee.unsw.edu.au/unsw/projects/armvfp/README.html>
- **Steve Furber: ARM System On-Chip; 2nd Ed, Addison-Wesley, 2000, ISBN: 0-201-67519-6. chapter 6 (NOT up to date)**

ELEC2041 lec21-fp-III.23

Saeid Nooshabadi

## Example: Matrix with FI Pt, Multiply, Add?



ELEC2041 lec21-fp-III.24

Saeid Nooshabadi

## Example: Matrix with FI Pt, Multiply, Add in C

```
void mm(double x[][32],double
y[][32], double z[][32]){
int i, j, k;

for (i=0; i<32; i=i+1)
for (j=0; j<32; j=j+1)
for (k=0; k<32; k=k+1)
x[i][j] = x[i][j] +
y[i][k] * z[k][j];
}

```

**Why pass in # of cols?**

- Starting addresses are parameters in a1, a2, and a3. Integer variables are in v2, v3, v4. Arrays 32 x 32

- Use fldd/fstd (load/store 64 bits)

ELEC2041 lec21-fp-III.25

Saeid Nooshabadi

## Multidimensional Array Addressing

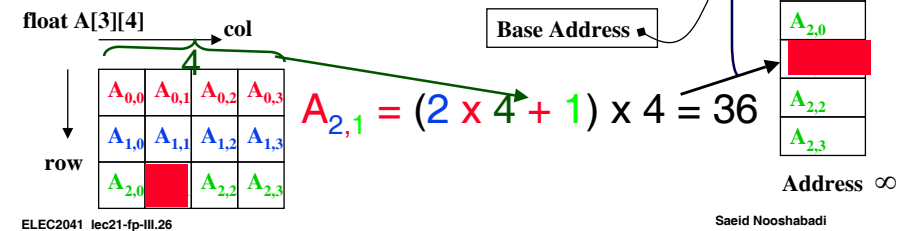
- C stores multidimensional arrays in row-major order

- elements of a row are consecutive in memory (Next element in row)

- FORTRAN uses column-major order (Next element in col)

- What is the address of A[x][y]? (x = row # & y = col #)

**Why pass in # of cols?**



ELEC2041 lec21-fp-III.26

Saeid Nooshabadi

## ARM code for first piece: initialize, x[][]

- Initialize Loop Variables

```
mm: ...
stmfd sp!, {v1-v4}
mov v1, #32 ; v1 = 32
mov v2, #0 ; i = 0; 1st loop
L1: mov v3, #0 ; j = 0; reset 2nd
L2: mov v4, #0 ; k = 0; reset 3rd

```

- To fetch x[i][j], skip i rows (i\*32), add j

```
add a4,v3,v2, lsl #5 ;a4 = i*25+j
```

- Get byte address (8 bytes), load x[i][j]

```
add a4,a1,a4, lsl #3;a4 = a1 +a4*8
; (i,j byte addr.)
fldd d0, [a4] ; d0 = x[i][j]
```

ELEC2041 lec21-fp-III.27

Saeid Nooshabadi

## ARM code for second piece: z[], y[][]

- Like before, but load y[i][k] into d1

```
L3: add ip,v4,v2, lsl #5 ;ip = i*25+k
add ip,a2,ip, lsl #3 ;ip = a2 +ip*8
; (i,k byte addr.)
fldd d1, [ip] ; d1 = y[i][k]
```

- Like before, but load z[k][j] into d2

```
add ip,v3,v4, lsl #5 ;ip = k*25+j
add ip,a3,ip, lsl #3 ;ip = a3 +ip*8
; (k,j byte addr.)
fldd d2, [ip] ; d2 = z[k][j]
```

- Summary: d0:x[i][j], d1:y[i][k], d2:z[k][j]

ELEC2041 lec21-fp-III.28

Saeid Nooshabadi

## ARM code for last piece: add/mul, loops

- Add  $y*z$  to  $x$   
`fmacd d0,d1,d2 ; x[][] = x + y*z`
- Increment  $k$ ; if end of inner loop, store  $x$   
`add v4,v4,#1 ; k = k + 1`  
`cmp v4,v1 ; if(k<32) goto L3`  
`blt L3`  
`fstd d0,[a4] ; x[i][j] = d0`
- Increment  $j$ ; middle loop if not end of  $j$   
`add v3,v3,#1 ; j = j + 1`  
`cmp v3,v1 ; if(j<32) goto L2`  
`blt L2`
- Increment  $i$ ; if end of outer loop, return  
`add v2,v2,#1 ; i = i + 1`  
`cmp v2,v1 ; if(i<32) goto L1`  
`blt L1`

## ARM code for Return

- Return  
`ldmfd sp!, {v1-v4}`  
`mov pc, lr`

## “And in Conclusion..”

- Exponent = 255, Significand nonzero  
Represents NaN
- Finite precision means we have to cope with round off error (arithmetic with inexact values) and truncation error (large values overwhelming small ones).
- In NaN representation of Ft. Pt. Exponent = 255 and Significand  $\neq 0$
- In Denorm representation of Ft. Pt. Exponent = 0 and Significand  $\neq 0$
- In Denorm representation of Ft. Pt. numbers there no hidden 1.