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

Review: ARM Fl. 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, 2x10^{-38} ... 2x10^{38}):
    - fcmps, fadds, fsubs, fmuls, fdivs
  - Double Precision (64 bits, 2x10^{-308}...2x10^{308}):
    - fcmpd, fadd, fsubd, fmul, fdivd
- Big Idea: Instructions determine meaning of data; nothing inherent inside the data

Review: Floating Point Representation

- Single Precision and Double Precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>...</th>
<th>22</th>
<th>21</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>...</th>
<th>19</th>
<th>18</th>
<th>Exponent</th>
<th>Significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 bit</th>
<th>11 bits</th>
<th>20 bits</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>...</th>
<th>0</th>
<th>Significand (cont’d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 bit</th>
<th>31 bits</th>
</tr>
</thead>
</table>

\((-1)^S \times (1+\text{Significand}) \times 2^{(\text{Exponent-Bias})}\)
New ARM arithmetic instructions

- **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 \( s0 = s1 \times s2 \)  Fl. Pt. Mul (single)
- **fmul d0,d1,d2 \( d0 = d1 \times d2 \)  Fl. Pt. Mul (double)
- **fdivs** s0,s1,s2 \( s0 = s1 \div s2 \)  Fl. Pt. Div (single)
- **fdivd** d0,d1,d2 \( d0 = d1 \div d2 \)  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?  See later

Special Numbers

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

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>???</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>???</td>
</tr>
</tbody>
</table>

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: \( \text{op(NaN,X)} = \text{NaN} \)
  - OK if calculate but don’t use it
  - Ask math Prof
  - \( \text{cmp s}1, s2 \) produces unordered results if either is an NaN

Special Numbers (cont’d)

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

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>???</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Representation for Denorms (#1/2)

Problem: There's a gap among representable FP numbers around 0
- Significand = 0, Exp = 0 ($2^{-127}$) $\not\rightarrow$ 0
- Smallest representable positive num:
  - $a = 1.0..._2 \times 2^{-126} = 2^{-126}$
- Second smallest representable positive num:
  - $b = 1.000..._2 \times 2^{-126} = 2^{-126} + 2^{-149}$
- $a - 0 = 2^{-126}$
- $b - a = 2^{-149}$

Gap! Gap!

$-\infty \rightarrow 0 \rightarrow +\infty$

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

Special Numbers

What have we defined so far? (Single Precision)

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denorm</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- infinity</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

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

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 digitss (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

- **(int) exp**
  - Coerces and converts it to the nearest integer (truncates)
  - affected by rounding modes
    - i = (int) (3.14159 * f);
    - fuitos (floating → int) In ARM

- **(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 \rightarrow \text{int} \rightarrow \text{float}

\text{if } (f == \text{(float)}((\text{int}) f)) \{ \
\quad \text{printf("true");} \\
\}

\circ \text{Will not always work}

\circ \text{Small values of floating point don’t have good integer representations}

\circ \text{Also rounding errors}

\text{Ints, Fractions and rounding in C}

\circ \text{What do you get?}

\begin{align*}
\{ \text{int } x = 3/2; & \quad \text{int } y = 2/3; \\
& \quad \text{printf("x: \%d, y: \%d", x, y); } \\
\}
\end{align*}

\circ \text{How about?}

\begin{align*}
\text{int } cela &= (\text{fahr - 32})/9; \\
\text{int } celb &= (5/9) \ast (\text{fahr - 32}) \\
\text{float } cel &= (5.0/9.0) \ast (\text{fahr - 32}); \\
\text{fahr} = 60 \Rightarrow cela: 15 \text{ celb: 0 cel: } 15.55556
\end{align*}

\text{Floating Point Fallacy}

\circ \text{FP Add, subtract associative: FALSE!}

\begin{align*}
\circ & \quad x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, \text{ and } z = 1.0 \\
\circ & \quad x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\
& \quad = -1.5 \times 10^{38} + (1.5 \times 10^{38}) = 0.0 \\
\circ & \quad (x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\
& \quad = (0.0) + 1.0 = 1.0
\end{align*}

\text{Therefore, Floating Point add, subtract are not associative!}

\circ \text{Why? FP result approximates real result!}

\circ \text{This example: } 1.5 \times 10^{38} \text{ is so much larger than 1.0 that } 1.5 \times 10^{38} + 1.0 \text{ in floating point representation is still } 1.5 \times 10^{38}

\text{Floating Point Fallacy: Accuracy optional?}

\circ \text{July 1994: Intel discovers bug in Pentium}

\circ \quad \circ \quad \text{Occasionally affects bits 12-52 of D.P. divide}

\circ \text{Sept: Math Prof. discovers, put on WWW}

\circ \text{Nov: Front page trade paper, then NY Times}

\circ \quad \circ \quad \text{Intel: “several dozen people that this would affect. So far, we’ve only heard from one.”}

\circ \quad \circ \quad \text{Intel claims customers see 1 error/27000 years}

\circ \quad \circ \quad \text{IBM claims 1 error/month, stops shipping}

\circ \text{Dec: Intel apologizes, replace chips $300M}
**Reading Material**


---

**Example: Matrix with Fl Pt, Multiply, Add?**

\[
\begin{bmatrix}
X
\end{bmatrix}
= \begin{bmatrix}
i
\end{bmatrix} X + \begin{bmatrix}
j
\end{bmatrix} Y \begin{bmatrix}
k
\end{bmatrix} Z
\]

Example: Matrix with Fl Pt, Multiply, Add in C

```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)

---

**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?

![Address Calculation Diagram](diagram.png)

\( A_{2,1} = 2 \times 4 + 1 = 9 \)

Address \( \infty \)

Base Address \( \star \)
ARM code for first piece: initialize, x[i][j]

° 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*2^5+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]

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

ARM code for second piece: z[k][j], y[i][k]

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

L3: add ip,v4,v2, lsl #5 ; ip = i*2^5+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*2^5+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]

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 L3

° 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 ≠ 0

- In Denorm representation of Ft. Pt.
  Exponent = 0 and Significand ≠ 0

- In Denorm representation of Ft. Pt.
  numbers there no hidden 1.