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, fmuls, fdivd

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

Review: Floating Point Representation
- Single Precision and Double Precision
  
  \[
  \begin{array}{ccc}
  31:0 & 23:22 & 0 \\
  S & \text{Exponent} & \text{Significand} \\
  1\text{ bit} & 8\text{ bits} & 23\text{ bits} \\
  \\
  31:0 & 20:19 & 0 \\
  S & \text{Exponent} & \text{Significand} \\
  1\text{ bit} & 11\text{ bits} & 20\text{ bits} \\
  \end{array}
  \]

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

<table>
<thead>
<tr>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadds $s0,s1,s2$</td>
<td>$s0=s1+s2$</td>
<td>Fl. Pt. Add (single)</td>
</tr>
<tr>
<td>fadd $d0,d1,d2$</td>
<td>$d0=d1+d2$</td>
<td>Fl. Pt. Add (double)</td>
</tr>
<tr>
<td>fsub $s0,s1,s2$</td>
<td>$s0=s1-s2$</td>
<td>Fl. Pt. Sub (single)</td>
</tr>
<tr>
<td>fsubd $d0,d1,d2$</td>
<td>$d0=d1-d2$</td>
<td>Fl. Pt. Sub (double)</td>
</tr>
<tr>
<td>fmuls $s0,s1,s2$</td>
<td>$s0=s1 \times s2$</td>
<td>Fl. Pt. Mul (single)</td>
</tr>
<tr>
<td>fmuld $d0,d1,d2$</td>
<td>$d0=d1 \times d2$</td>
<td>Fl. Pt. Mul (double)</td>
</tr>
<tr>
<td>fdivs $s0,s1,s2$</td>
<td>$s0=s1 \div s2$</td>
<td>Fl. Pt. Div (single)</td>
</tr>
<tr>
<td>fdivd $d0,d1,d2$</td>
<td>$d0=d1 \div d2$</td>
<td>Fl. Pt. Div (double)</td>
</tr>
<tr>
<td>fcmps $s0,s1$</td>
<td>FCPSR flags = $s0 - s1$</td>
<td>Fl. Pt. Compare (single)</td>
</tr>
<tr>
<td>fcmpd $d0,d1$</td>
<td>FCPSR flags = $d0 - d1$</td>
<td>Fl. Pt. Compare (double)</td>
</tr>
</tbody>
</table>

### 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: $\text{op}(\text{NaN},X) = \text{NaN}$
- OK if calculate but don’t use it
- $\text{cmp } s1, s2$ 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}) \rightarrow 0
  • Smallest representable positive num:
    - \( a = 1.0...2 \times 2^{-126} = 2^{-126} \)
  • Second smallest representable positive num:
    - \( b = 1.000...1 \times 2^{-126} = 2^{-126} + 2^{-149} \)
  • \( a - 0 = 2^{-126} \)
  • \( b - a = 2^{-149} \)

\( \Box \Box \) Gap! Gap!

0 \[ \infty \] b \[ \infty \] \[ \infty \] a

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: \(2^{-149} \leq X \leq 2^{-126} - 2^{-149}\)

\[ \Box \Box \] Gap!

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”

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:
  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
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”: $2.2001 \rightarrow 2.3$
  - $-2.3001 \rightarrow -2.3$
- Round towards -infinity
  - ALWAYS round “down”: $1.9999 \rightarrow 1.9$, $-1.9999 \rightarrow -2.0$
- Truncate
  - Just drop the last digits (round towards 0); $1.9999 \rightarrow 1.9$, $-1.9999 \rightarrow -1.9$
- 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
  - $2.55 \rightarrow 2.6$
  - $3.45 \rightarrow 3.4$
- 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 an integer by truncation (rounds to towards 0)
  - affected by rounding modes
  - $i = \text{(int)} (3.14159 \times f)$
  - $\text{fuitos (floating \rightarrow int)}$ In ARM to round to a selected mode (default nearest)
  - $\text{fuitozs (floating \rightarrow int)}$ In ARM to round towards zero
- (float) exp
  - converts integer to nearest floating point
  - $f = f + \text{(float)} i$
  - $\text{faitos (int \rightarrow floating)}$ In ARM
**Int, Fractions and rounding in C**

- **What do you get?**
  ```c
  { int x = 3/2;    int y = 2/3;
    printf("x: %d, y: %d", x, y); }
  ```
- **How about?**
  ```c
  int cela = ((fahr - 32) * 5)/ 9;
  int celb = (5 / 9) * (fahr - 32)
  float celt = (5.0 / 9.0) * (fahr - 32);
  ```

**Floating Point Fallacy**

- **FP Add, subtract associative: FALSE!**
  \[ X + (Y + z) = (X + y) + z \]
  \[ x = -1.5 \times 10^{38}, y = 1.5 \times 10^{38}, \text{ and } z = 1.0 \]
  \[ x + (y + z) = -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) = -1.5 \times 10^{38} + 1.5 \times 10^{38} = 0.0 \]
  \[ (x + y) + z = (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 = (0.0) + 1.0 = 1.0 \]
- **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 introduces when they optimised 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

IEEE 754 Floating Point Issues

- It is complex, involves lots of details
  - We just scratched the surface
- Check for gradual underflow and treating denomsrs 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!

Reading Material


Example: Matrix with Ft Pt, Multiply, Add?

\[
\begin{align*}
X & \rightarrow \text{Col 32} \\
\text{Row 32} & \downarrow \\
X & = \begin{pmatrix} \star \end{pmatrix} + \begin{pmatrix} Y \end{pmatrix} \ast \begin{pmatrix} Z \end{pmatrix} \\
\end{align*}
\]
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];
}
```

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

° Use flld/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 #)

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*2^5+j

Get byte address (8 bytes), load x[i][j]
```
ADD A4, A1, A4, LSL #3; A4 = a1 +a4*8
FLDD D0, [A4] ; D0 = x[i][j]
```

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*2^5+k
      add ip, a2, ip, lsl #3 ; ip = a2 +ip*8
      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
  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 \(yz\) to \(x\)
  
  ```asm
  fmacd d0,d1,d2 ; x[i][j] = x + y*z
  ```

- Increment \(k\); if end of inner loop, store \(x\)
  
  ```asm
  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 \(j\)
  
  ```asm
  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
  
  ```asm
  add v2,v2,#1 ; i = i + 1
  cmp v2,v1 ; if(i<32) goto L1
  blt L1
  ```

**ARM code for Return**

- Return
  
  ```asm
  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.