Lectures 22: Fractions

Review: Floating Point Representation

- Single Precision and Double Precision
  - Single Precision:
    - Significand: 23 bits
    - Exponent: 8 bits
    - Sign: 1 bit
    - Formula: 
      \[ (-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})} \]
  - Double Precision:
    - Significand: 20 bits
    - Exponent: 11 bits
    - Sign: 1 bit
    - Formula: 
      \[ (-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent} - \text{Bias})} \]

Review: Special Numbers

- What have we defined so far? (Single Precision)
  - Exponent: 0, Significand: 0, Object: 0 (Denorm)
  - Exponent: 0, Significand: nonzero, Object: Denorm
  - Exponent: 255, Significand: 0, Object: +/- infinity
  - Exponent: 255, Significand: nonzero, Object: NaN

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

Understanding the Ints/Floats (#1/2)

- Think of ints as having the binary point on the right
  \[ D_{31} D_{30} D_{29} \ldots D_0 . \]
  - Represents number (unsigned)
    - \( D_{31} x 2^{31} + D_{30} x 2^{30} + D_{29} x 2^{29} + \ldots + D_0 x 2^0 \)
- In Float the Binary point is not fixed (Floats!)
  - \( 1.1000\ldots \times 2^2 \rightarrow 00110.000\ldots \)
  - \( 1.1000\ldots \times 2^1 \rightarrow 0011.0000\ldots \)
  - \( 1.1000\ldots \times 2^0 \rightarrow 001.10000\ldots \)
  - \( 1.1000\ldots \times 2^{-1} \rightarrow 0.0110000\ldots \)
  - \( 1.1000\ldots \times 2^{-2} \rightarrow 0.00110000\ldots \)
  - The Binary point is not fixed!
Understanding the Ints/Floats (#2/2)

° The sequential Integer numbers are separated by a fixed values of 1

-8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8

° The sequential Floating numbers are not separated by a fixed value.

-8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8

Fractions with Equal Distribution

° How do we represent this?

-1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

° Accuracy is at a premium and not the range

• We want to use all the bits for accuracy
• Situation in many DSP application: the small range and high accuracy.
• We used FIXed Point Fractions.

Representing Fraction

° Imagine the binary point in the middle

\[ D_{15} D_{14} \ldots D_0 D_{-1} \ldots D_{-16} \]

° Represents number

• \( D_{15} \times 2^{15} + D_{14} \times 2^{14} + \ldots + D_0 \times 2^0 + D_{-1} \times 2^{-1} + \ldots + D_{-16} \times 2^{-16} \)
• Numbers in the range: 0.0 to \((2^{16} - 1) \times (1 - 2^{-16})\)
• 2\(^{32}\) fractional numbers with step size = 2\(^{-16}\)
• 2.5\(_{10}\) = 10.1\(_{2}\) => 0000 0000 0000 0010 1000 0000 0000 0000

° Same arithmetic mechanism for Fixed

\[ \begin{align*}
A_0 & \quad A_1 \\
A_{-1} & \quad A_{-2} \\
B_0 & \quad B_1 \\
B_{-1} & \quad B_{-2}
\end{align*} \]

Overflow? 

\[ D_1 D_0 D_{-1} D_{-2} \]

Rounding?

The position of the binary point is maintained in software

Understanding the Ints/Fixed/Floats

° Think of ints as having the binary point on the right

\[ \begin{align*}
D_3 & \quad D_2 \\
D_1 & \quad D_0 \\
D_{-1} & \quad D_{-2}
\end{align*} \]

° Think of the bits of the significand in Float as binary fixed-point value

\[ 1 \cdot D_1 D_2 D_3 D_4 D_5 \ldots D_{-23} \]

\[ = 1 + D_1 \times 2^{-1} + D_2 \times 2^{-2} + D_3 \times 2^{-3} + \ldots + D_{-23} \times 2^{-23} \]

° The exponent causes the binary point to float.

° Since calculations are limited to finite precision, must round result

• few extra bits carried along in arithmetic
• four rounding modes
Ints, Fixed-Point & Floating Point

° Ints represent $2^N$ equally spaced whole numbers
  • Fixed binary point at the right

° Moving binary point to the left can represent $2^N$ equally spaced fractions

° Exponent effectively shifts the binary point
  • Imagine infinite zeros to the right and left
  • Represent $2^M$ equally spaced values in each of $2^K$ exponentially increasing intervals

Recall: Multiplication Instructions

° ARM provides multiplication instruction
  • $\text{mul} \ Rd, \ Rm, \ Rs \ ; \ Rd = Rm \times Rs$
  • (Lower precision multiply instructions simply throws top 32 bits away)

What about Multiplication for Fractions

° Imagine the binary point on the left

° ARM multiplication instruction won’t work
  • $\text{mul} \ Rd, \ Rm, \ Rs \ ; \ Rd = Rm \times Rs$
  • (Lower precision multiply instructions simply throws top 32 bits away).
  • Top 32 bits are more important. (eg. $0.11 \times 0.10 = 0.1100 = 0.11$)

Multiply-Long for Fractions

° Instructions are
  • $\text{MULL}$ which gives $\text{RdHi}, \text{RdLo}:=Rm\times Rs$

° Full 64 bit of the result now matter
  • Need to specify whether operands are signed or unsigned

° Syntax of new instructions are:
  • $\text{umull} \ \text{RdLo}, \text{RdHi}, \text{Rm}, \text{Rs} \ ; \text{RdHi}, \text{RdLo}:=Rm\times Rs$
  • $\text{smull} \ \text{RdLo}, \text{RdHi}, \text{Rm}, \text{Rs} \ ; \text{RdHi}, \text{RdLo}:=Rm\times Rs$ (Signed)
  • Example: $\text{umull} \ r4, \ r5, \ r3, \ r2; \ r5:r4:=r3*\ r2$
  • Not generated by the general compiler. (Needs Hand coding).
  • DSP compilers generate them

° We can ignore the $\text{RdLo}$ with some loss of accuracy
Fractions: Negative Powers of Two (#1/2)

° \(1_2 = 2^0 = 1_{10}\)
° \(0.1_2 = 2^{-1} = 0.5_{10} = 1/2\)
° \(0.01_2 = 2^{-2} = 0.25_{10} = 1/4\)
° \(0.001_2 = 2^{-3} = 0.125_{10} = 1/8\)
° \(0.0001_2 = 2^{-4} = 0.0625_{10} = 1/16\)
° \(0.11_2 = 2^{-1} + 2^{-2} = 0.5_{10} + 0.25_{10} = 0.75_{10} = 1/2 + 1/4 = 3/4 = (1 - 1/4) = (1_2 - 0.1_2)\)
° \(0.101_2 = 2^{-1} + 2^{-3} = 0.5_{10} + 0.125_{10} = 1/2 + 1/8 = 0.625_{10}\)
° \(0.00110011001100 \ldots = 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} + \ldots = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 + \ldots = 0.125_{10} + 0.0625_{10} + 0.03125 + 0.015625 + 0.0009765625 + \ldots = 0.00048828125 + \ldots\)
° \(= 0.2_{10}\)

Add/Sub & Shift for Multiplication of Fractions

° Recall multiplication of integers via add/sub and shift:
  • Assume two integer variables \(f\) and \(g\)
  \[f = 3^g \\
  \text{/* } f = (2+1) \times g */ \text{ (in C)} \]
  \[\text{add } v1, v2, v2 \text{ lsl } #1 ; \ v1 = v2 + v2 \times 2 \text{ (in ARM)}\]
° What about: \(f = g \times 0.3\) (\(f\) and \(g\) are both integers)
  • Example: \(g=10 \Rightarrow f = 10 \times 0.3 = 3\)
  \<g=12 \Rightarrow f = 12 \times 0.3 = 3.6\)
  \(g=12 \Rightarrow f = 12 \times 0.3 = 3\)
° \(0.3_{10} = 0.001 \ldots \times 2^{-32}\)

\[
\begin{align*}
  f &= g \times 2^{-2} + g \times 2^{-5} + g \times 2^{-6} + g \times 2^{-10} + g \times 2^{-11} + g \times 2^{-14} + \ldots + g \times 2^{-32} + \ldots \\
  \text{sub } v1, v2, v2 \text{ lsr } #2 ; v1 &= g \times (1 - 1/4) = g \times 3/4 \text{ (0.11)} \\
  \text{add } v1, v1, v1 \text{ lsr } #4 ; g \times 0.11001100 \\
  \text{add } v1, v1, v1 \text{ lsr } #8 ; g \times 0.1100110011001100 \\
  \text{add } v1, v1, v1 \text{ lsr } #16; g \times 0.11001100110011001100110011001100 \\
  \text{mov } v1, v1 \text{ lsr } #4; g \times 0.000011001100110011001100110011001100 \\
  \text{add } v1, v1, v2 \text{ lsr } #2 ; g \times 0.010011001100110011001100110011001100 \\
\end{align*}
\]

Loss of Accuracy in Multiplication with Fraction

° But bits drop of from the right side as \(g\) shifts right
  • Loosing the shifted bits could produce wrong result (loss of accuracy)
  • In reality we would have like to keep the shifted bits and include them in the additions (64 bit addition).
  \[\begin{array}{cccccccccccc}
D_{33} & D_{32} & D_{29} & \ldots & D_0 & D_{-1} & D_{-2} & D_{-16} & D_{-17} & \ldots & D_{-32} \\
\end{array}\]

Approximation

How to get the Carrys in successive additions?
Not always easy, needs a lot of house keeping in software.
Think about it!
**Loss of Accuracy Example in Decimal**

Considering the Shifted Digits Example:

\[
\begin{align*}
123999 \times 0.1111 &= 123999 \times (0.1 + 0.01 + 0.001) \\
&= 1239.99 + 1239.99 + 123.999 \\
&= 13763.889 \\
\end{align*}
\]

Not Considering the Shifted Digits Example:

\[
\begin{align*}
123999 \times 0.1111 &= 123999 \times (0.1 + 0.01 + 0.001) \\
&= 1239.9 + 1239.9 + 123.999 \\
&= 13761 \\
\end{align*}
\]

**Off by 2**

**Recall: Division**

- No Division Instruction in ARM

- Division has two be done in software through a sequence of shift/subtract/add instruction.

  - General A/B implementation (See Experiment 3)

  - For B in A/B a constant value (eg 10) simpler technique via Shift, Add and Subtract is available (Will discuss it Now)

**Division by a Constant**

\[
A/B = A \times (1/B)
\]

- The lines marked with a '#' are the special cases $2^n$, which are easily dealt with just by simple shifting to right by $n$ bits.

- The lines marked with an '*' have a simple repeating pattern.

- The lines marked with a '$$' have more complex repeating pattern.

- Division can be performed by successive right shifts & additions and /subtractions

**Division by a Constant Regular Patterns**

Regular patterns are for $B=2^n+2^m$ or $B=2^n-2^m$ (for $n>m$):

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>(2^n+2^m)</th>
<th>n</th>
<th>m</th>
<th>(2^n-2^m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>17</td>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>18</td>
<td>4</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>20</td>
<td>4</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>24</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>33</td>
<td>5</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>34</td>
<td>5</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>36</td>
<td>5</td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>40</td>
<td>5</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>48</td>
<td>5</td>
<td>0</td>
<td>31</td>
</tr>
</tbody>
</table>
Division by a Constant Example (by 10)

B = 1/1010 = 0.00011001100110011001100110011001---2

Assume A → a1 and A (1/B) → a1

sub a1, a1, al lsr #2 ; al = A*(1-1/4) = A*3/4 (0.11)
add al, al, al lsr #4 ; A*0.11001100
add al, al, al lsr #8 ; A*0.1100110011001100
add al, al, al lsr #16; A*0.11001100110011001100110011001100
mov a1, al lsr #3 ; A*0.00011001100110011001100110011001100

° But what about bits drop of from the right side as a shifts right?
° This could cause the answer to be less by 1
° This can be corrected!
° Since correct divide by 10 would rounds down (eg 98/10=9), the remainder (8) can be calculated by:
  A - (A/10)*10 = 0..9
° If bit drop offs from the right cause (A/10) to be less by 1 then
  A - (A/10)*10 = 10..19. So add 1 to computed (A/10)

Division by a Constant 10 Function

B = 1/1010 = 0.00011001100110011001100110011001---2

Assume A → a1 and A (1/B) → a1

Div10:
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required

sub a2, a1, #10 ; keep (A-10) for later
sub a1, a1, al lsr #2 ; al = A*(1-1/4) = A*3/4 (0.11)
add al, al, al lsr #4 ; A*0.11001100
add al, al, al lsr #8 ; A*0.1100110011001100
add al, al, al lsr #16; A*0.1100110011001100110011001
mov al, al lsr #3 ; A*0.00011001100110011001100110011001100
add a3, a1, a1 lsl #2 ; (A/10)*5
subs a2, a2, a3, lsl #1 ; calc (A-10) - (A/10)*10, <0 or 0>?
addpl a1, a1, #1 ; fix-up quotient
addmi a2, a2, #10 ; fix-up remainder (-10..-1)+10 → (0..9)
mov pc, lr

Uns. Int to Decimal ASCII Converter via div10

° Aim: To convert an unsigned integer to Decimal ASCII
° Example: 10011001100110011001100110011001/10 = "2576980377"
° Algorithm:
  • Divide it by 10, yielding a quotient and a remainder. The remainder (in the range 0-9) is the last digit (right most) of the decimal. Convert remainder to to ASCII.
  • Repeat division with new quotient until it is zero
° Example: 10011001100110011001100110011001/10 = 1111010111000010100011110101 (257698037) and Remainder of 111 (7) So:
  ° 10011001100110011001100110011001 (2576980377)
  ° 1111010111000010100011110101 (257698037) 7
  ° 1100010010010111011101001011 (25769803) 7
  ° 1100010010010111011101001011 (2576980)
  ° 0

Uns. Int to Decimal ASCII Converter Function

utoa:
; function entry: On entry al has the address of memory ; to store the ASCII string and a1 contains the integer ; to convert
stmfd sp!, {v1, v2, lr}; save some v1, v2 and ret address
mov v1, a1 ; preserve arg a1 over following func. call
bl div10 ; a1 = a1 / 10, a2 = a2 % 10
mov v2, a2 ; move remainder to v2
cmp a1, #0 ; quotient non-zero?
movne a2, a1 ; quotient to a2...
cmp a1, #0 ; quotient non-zero?
mov v1, a1 ; buffer pointer unconditionally to a1
bne utoa ; conditional recursive call to utoa
mov v2, #0 ; convert to ascii (final digit first)
strb v2, [al], #1 ; store digit at end of buffer
ldmf sp!, {v1, v2, pc} ; function exit-restore and return
Uns. Int to Decimal ASCII Converter in C

```c
void utoa (char* Buf, int n) {
    if (n/10) utoa(Buf, n/10);
    *Buf=n%10 +'0';
    Buf++;
}
```

“And in Conclusion..”

- ints represent $2^N$ equally spaced whole numbers. fixed binary point at the right
- Moving binary point to the left can represent $2^N$ equally spaced fractions
- Exponent represent $2^M$ equally spaced values in each of $2^K$ exponentially increasing intervals
- Division by a constant via shift rights and adds/subs.
  - Beware of errors due to loss shifted bits from the right (lack of 64 bit addition).