

COMP 3221

Microprocessors and Embedded Systems

Lectures 22 : Fractions

<http://www.cse.unsw.edu.au/~cs3221>

September, 2003

Saeid Nooshabadi

Saeid@unsw.edu.au

COMP3211 lec22-fraction.1

Saeid Nooshabadi

Review: Special Numbers

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

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

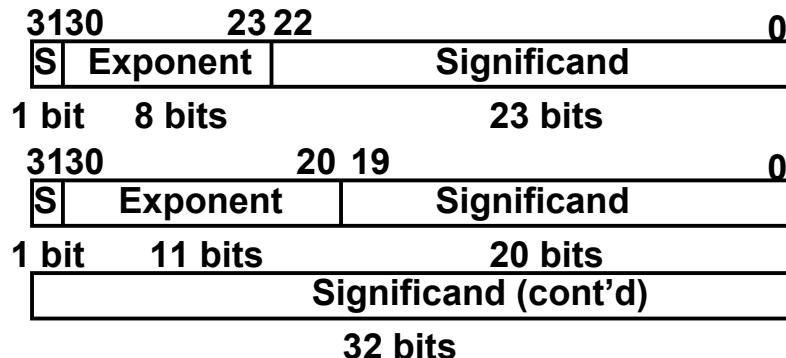
- Professor Kahan had clever ideas;
“Waste not, want not”

COMP3211 lec22-fraction.3

Saeid Nooshabadi

Review: Floating Point Representation

- Single Precision and Double Precision



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

COMP3211 lec22-fraction.2

Saeid Nooshabadi

Understanding the Ints/FLOATS (#1/2)

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



- Represents number (unsigned)
- $D_{31} \times 2^{31} + D_{30} \times 2^{30} + D_{29} \times 2^{29} + \dots + D_0 \times 2^0$

- In Float the Binary point is not fixed
(FLOATS!)

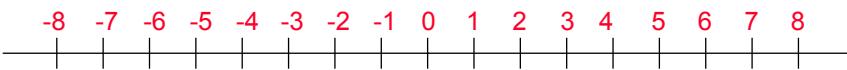
- 1.1000--- $\times 2^2 \rightarrow 00110.000---$
- 1.1000--- $\times 2^1 \rightarrow 0011.0000---$
- 1.1000--- $\times 2^0 \rightarrow 001.10000---$
- 1.1000--- $\times 2^{-1} \rightarrow 00.110000---$
- 1.1000--- $\times 2^{-2} \rightarrow 0.0110000---$

COMP3211 lec22-fraction.4 The Binary point is not fixed!

Saeid Nooshabadi

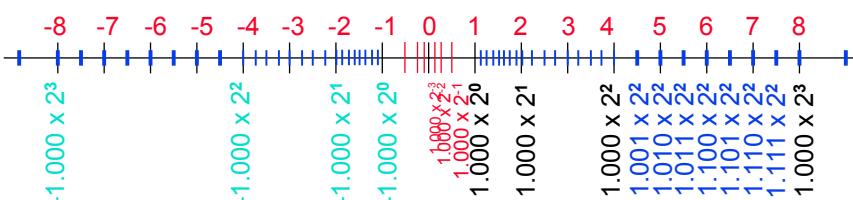
Understanding the Ints/Floats (#2/2)

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



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

- The separation changes exponentially

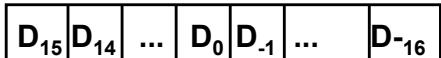


COMP3211 lec22-fraction.5

Saeid Nooshabadi

Representing Fraction

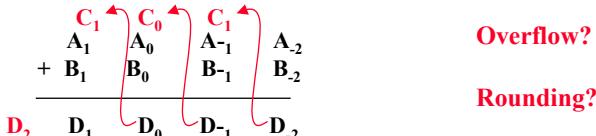
- Imagine the binary point in the middle



- Represents number

- $D_{15} \times 2^{15} + D_{14} \times 2^{14} + \dots + D_0 \times 2^0 + D_{-1} \times 2^{-1} + \dots + D_{-16} \times 2^{-16}$
- Numbers in the range: 0.0 to $(2^{16} - 1) \cdot (1 - 2^{-16})$
- 2^{32} fractional numbers with step size = 2^{-16}
- $2.5_{10} = 10.1_2 \Rightarrow 0000\ 0000\ 0000\ 0010\ 1000\ 0000\ 0000\ 0000$

- Same arithmetic mechanism for Fixed



The position of the binary point is maintained in software

COMP3211 lec22-fraction.7

Saeid Nooshabadi

Fractions with Equal Distribution

- How do we represent this?



- Accuracy is at a premium and not the range

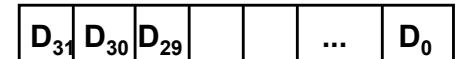
- We want to use all the bits for accuracy
- Situation in many DSP application: the small range and high accuracy.
- We used FIXed Point Fractions.

COMP3211 lec22-fraction.6

Saeid Nooshabadi

Understanding the Ints/Fixed/FLOATs

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



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

$$1. \boxed{D_{-1} | D_{-2} | D_{-3} | D_{-4} | D_{-5} | \dots | D_{-23}}$$

$$= 1 + D_{-1} \times 2^{-1} + D_{-2} \times 2^{-2} + D_{-3} \times 2^{-3} + D_{-4} \times 2^{-4} + D_{-5} \times 2^{-5} + \dots + D_{-23} \times 2^{-23}$$

- The exponent causes the binary point to float.

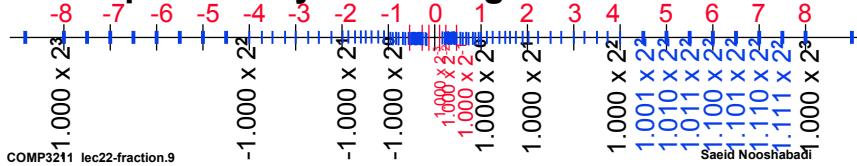
- Since calculations are limited to finite precision, must round result
 - few extra bits carried along in arithmetic
 - four rounding modes

COMP3211 lec22-fraction.8

Saeid Nooshabadi

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



What about Multiplication for Fractions

- ° Imagine the binary point on the left
 - | | | | | | | |
|-----------------|-----------------|-----|------------------|------------------|-----|------------------|
| D ₋₁ | D ₋₂ | ... | D ₋₁₆ | D ₋₁₇ | ... | D ₋₃₂ |
|-----------------|-----------------|-----|------------------|------------------|-----|------------------|
- ° ARM multiplication instruction won't work
 - `mul Rd, Rm, Rs ; Rd = Rm * Rs`
 - (Lower precision multiply instructions simply throws top 32 bits away).
 - Top 32 bits are more important. (eg. $0.11 * .10 = 0.1100 = 0.11$)

COMP3211 lec22-fraction.11

Saeid Nooshabadi

Recall: Multiplication Instructions

- ° ARM provides multiplication instruction
 - `mul Rd, Rm, Rs ; Rd = Rm * Rs`
 - (Lower precision multiply instructions simply throws top 32 bits away)

COMP3211 lec22-fraction.10

Saeid Nooshabadi

Multiply-Long for Fractions

- ° Instructions are
 - `MULL` which gives $RdHi, RdLo := Rm * Rs$
- ° Full 64 bit of the result now matter
 - Need to specify whether operands are signed or unsigned
- ° Syntax of new instructions are:
 - `umull RdLo, RdHi, Rm, Rs ; RdHi, RdLo := Rm * Rs`
 - `smull RdLo, RdHi, Rm, Rs ; RdHi, RdLo := Rm * Rs` (Signed)
 - Example: `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 $RdLo$ with some loss of accuracy

COMP3211 lec22-fraction.12

Saeid Nooshabadi

Fractions: Negative Powers of Two (#1/2)

- $1_2 = 2^0 = 1_{10}$
- $0.1_2 = 2^{-1} = 0.5_{10} = 1/2$
- $0.01_2 = 2^{-2} = 0.25_{10} = 1/4$
- $0.001_2 = 2^{-3} = 0.125_{10} = 1/8$
- $0.0001_2 = 2^{-4} = 0.0625_{10} = 1/16$
- $0.11_2 = 2^{-1} + 2^{-2} = 0.5_{10} + 0.25_{10} = 0.75_{10} = 1/2 + 1/4 = 3/4 = (1 - 1/4) = (1_2 - 0.1_2)$
- $0.101_2 = 2^{-1} + 2^{-3} = 0.5_{10} + 0.125_{10} = 1/2 + 1/8 = 0.625_{10}$
- $0.00110011001100 \cdots_2 = 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} + \cdots = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 + \cdots = 0.125_{10} + 0.0625_{10} + 0.03125 + 0.015625 + 0.0009765625 + 0.00048828125 + \cdots = 0.2_{10}$

COMP3211 lec22-fraction.13

Saeid Nooshabadi

Fractions: Negative Powers of Two (#2/2)

- $0.2_{10} = 0.00110011001100 \cdots_2 \rightarrow$
- $0.1_{10} = 0.2_{10}/2 = 0.000110011001100 \cdots_2$
- $0.3_{10} = 0.2_{10} + 0.1_{10} = 0.0011001100110011 \cdots_2 + 0.0001100110011001 \cdots_2 = 0.0100110011001100 \cdots_2$

COMP3211 lec22-fraction.14

Saeid Nooshabadi

Add/Sub & Shift for Multiplication of Fractions

- Recall multiplication of integers via add/sub and shift:

- Assume two integer variables f and g
 $f = 3*g$ /* $f = (2+1) * g$ */ (in C)
 $\text{add } v1, v2, v2 \text{ lsl } \#1 ; v1 = v2 + v2 * 2$ (in ARM)

- What about: $f = g * 0.3$ (f and g are both integers)

- Example: $g=10 \rightarrow f = 10 * 0.3 = 3$
 $g=12 \rightarrow f = 12 * 0.3 = 3$
 $g=12 \rightarrow f = 12 * 0.3 = 3$
- $0.3_{10} = 0.0100110011001100110011001100 \cdots_2$
 $\approx 0.0100110011001100110011001100_2$
32 bits

$f = g * 2^{-2} + g * 2^{-5} + g * 2^{-6} + g * 2^{-9} + g * 2^{-10} + g * 2^{-13} + g * 2^{-14} + \cdots + g * 2^{-29} + \dots$

```

sub v1, v2, v2 lsr #2 ;v1 =g*(1-1/4)=g*3/4 (0.11)
add v1, v1, v1 lsr #4 ;g*0.11001100
add v1, v1, v1 lsr #8 ;g*0.110011001100
add v1, v1, v1 lsr #16;g*0.110011001100110011001100
mov v1,      v1 lsr #4 ;g*0.0000110011001100110011001100
add v1, v1, v2 lsr #2 ;g*0.01001100110011001100110011001100

```

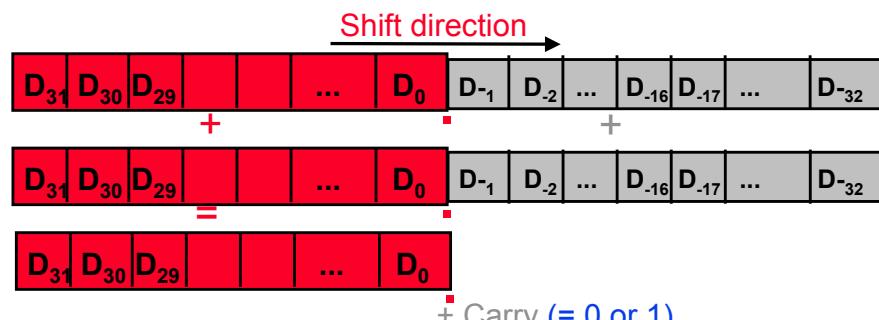
COMP3211 lec22-fraction.15

Saeid Nooshabadi

Approximation

Loss of Accuracy in Multiplication with Fraction

- But bits drop off 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).



+ Carry (= 0 or 1)
How to get the Carries in successive additions?
Not always easy, needs a lot of house keeping in software.
Think about it!

COMP3211 lec22-fraction.16

Saeid Nooshabadi

Loss of Accuracy Example in Decimal

- ### ◦ Considering the Shifted Digits Example:

$$\begin{array}{r}
 123999 \times 0.1111 = 123999 \times (0.1 + 0.01 + 0.001) \\
 = 12399.9 + \\
 1239.99 \\
 \hline
 123.999 \\
 \hline
 \textcolor{red}{13763.889} \quad \rightarrow 13763
 \end{array}$$

- ## **Not Considering the Shifted Digits Example:**

$$\begin{array}{r}
 123999 \times 0.1111 = 123999 \times (0.1 + 0.01 + 0.001) \\
 = 12399. \quad + \\
 \quad 1239. \\
 \hline
 \quad 123. \\
 \hline
 13761
 \end{array}
 \rightarrow 13761$$

- ## ° Off by 2

COMP3211 lec22-fraction 17

Saeid Nooshabadi

Division by a Constant

- $$\circ \quad A/B = A (1/B)$$

- 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 a '*' have a simple repeating pattern.

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

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

Recall: Division

- ## ◦ No Division Instruction in ARM

- Division has to 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**)

COMP3211 lec22-fraction 18

Saeid Nooshaba

Division by a Constant Regular Patterns

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

n	m	$(2^n + 2^m)$	n	m	$(2^n - 2^m)$
1	0	3	1	0	1
2	0	5	2	1	2
2	1	6	2	0	3
3	0	9	3	2	4
3	1	10	3	1	6
3	2	12	3	0	7
4	0	17	4	3	8
4	1	18	4	2	12
4	2	20	4	1	14
4	3	24	4	0	15
5	0	33	5	4	16
5	1	34	5	3	24
5	2	36	5	2	28
5	3	40	5	1	30
5	4	48	5	0	31

COMP3211 lec22-fraction.2

Saeid Nooshaba

Division by a Constant Example (by 10)

$B = 1/10_{10} = 0.0001100110011001100110011001\dots_2$

Assume A → a1 and A (1/B) → a1

```
sub a1, a1, a1 lsr #2 ;a1 = A*(1-1/4)= A*3/4 (0.11)
add a1, a1, a1 lsr #4 ;A*0.11001100
add a1, a1, a1 lsr #8 ;A*0.1100110011001100
add a1, a1, a1 lsr #16;A*0.110011001100110011001100
mov a1, a1 lsr #3 ;A*0.0001100110011001100110011001100
```

- But what about bits drop off 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 $(A/10)$

COMP3211 lec22-fraction.21

Saeid Nooshabadi

Uns. Int to Decimal ASCII Converter via div10

- Aim: To convert an unsigned integer to Decimal ASCII
- Example: 10011001100110011001100110011001 → "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: $1001100110011001100110011001/10 = 1111010111000010100011110101$ (257698037) and Remainder of 111 (7) So:
 - 1001100110011001100110011001 (2576980377)
 - 1111010111000010100011110101 (257698037) 7
 - 110001001001101101001011 (25769803) 7
 - 110001001001101101001011 (2576980) 3
 - ...
 - 0

COMP3211 lec22-fraction.23

Saeid Nooshabadi

Division by a Constant 10 Function

$B = 1/10_{10} = 0.000110011001100110011001\dots_2$

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, a1 lsr #2 ;a1 = A*(1-1/4)= A*3/4 (0.11)
add a1, a1, a1 lsr #4 ;A*0.11001100
add a1, a1, a1 lsr #8 ;A*0.1100110011001100
add a1, a1, a1 lsr #16;A*0.110011001100110011001100
mov a1, a1 lsr #3 ;A*0.0001100110011001100110011001100
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 Function

```
utoa:
; function entry: On entry a1 has the address of memory
; to store the ASCII string and a1 contains the integer
; to convert
stmfdf sp!, {v1, v2, lr};save some v1, v2 and ret. address
mov v1, a1 ; preserve arg a1 over following func. calls
mov a1, a2
b1 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...
mov a1, v1 ; buffer pointer unconditionally to a1
blne utoa ; conditional recursive call to utoa
add v2, v2, #'0' ; convert to ascii (final digit
; first)
strb v2, [a1], #1 ; store digit at end of buffer
ldmf sp!, {v1, v2, pc} ; function exit-restore and
; return
```

COMP3211 lec22-fraction.24

Saeid Nooshabadi

Uns. Int to Decimal ASCII Converter in C

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

COMP3211 lec22-fraction.25

Saeid Nooshabadi

“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).

COMP3211 lec22-fraction.26

Saeid Nooshabadi