

# ELEC2041

## Microprocessors and Interfacing

### Lectures 22: Fractions

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

May 2006

Saeid Nooshabadi

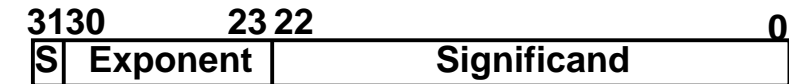
saeid@unsw.edu.au

ELEC2041 lec22-fraction.1

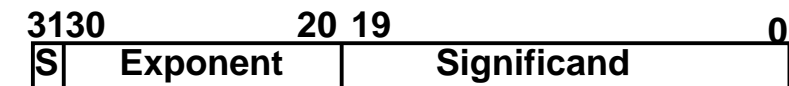
Saeid Nooshabadi

## Review: Floating Point Representation

### Single Precision and Double Precision



1 bit 8 bits 23 bits



1 bit 11 bits 20 bits



32 bits

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

ELEC2041 lec22-fraction.2

Saeid Nooshabadi

## Review: 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>

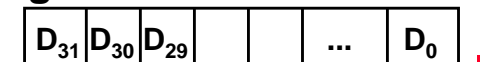
Professor Kahan had clever ideas;  
"Waste not, want not"

ELEC2041 lec22-fraction.3

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!)**

$$1.1000\text{---} \times 2^2 \rightarrow 00110.000\text{---}$$

$$1.1000\text{---} \times 2^1 \rightarrow 0011.0000\text{---}$$

$$1.1000\text{---} \times 2^0 \rightarrow 001.10000\text{---}$$

$$1.1000\text{---} \times 2^{-1} \rightarrow 00.110000\text{---}$$

$$1.1000\text{---} \times 2^{-2} \rightarrow 0.0110000\text{---}$$

The Binary point is not fixed!

ELEC2041 lec22-fraction.4

Saeid Nooshabadi

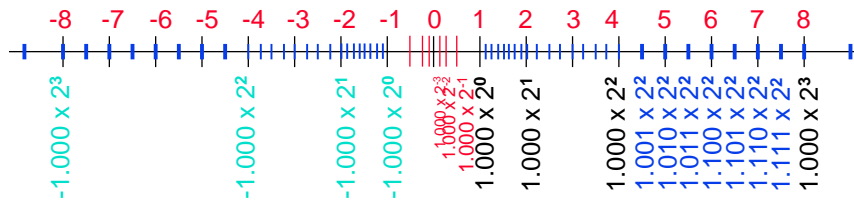
## Understanding the Ints/Floats (#2/2)

◦ 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



## Fractions with Equal Distribution

◦ How do we represent this?

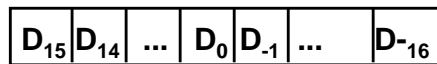


◦ Accuracy is at a premium and not the range

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

## Representing Fraction

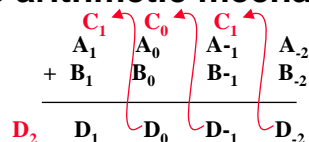
◦ Imagine the binary point in the middle



◦ 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



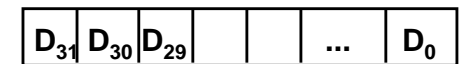
Overflow?

Rounding?

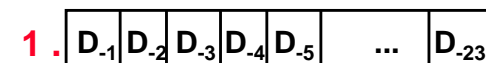
The position of the binary point is maintained in software

## Understanding the Ints/Fixed/Floats

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



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



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

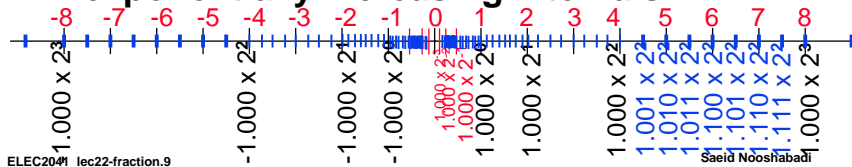
◦ The exponent causes the binary point to float.

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

- few extra bits carried along in arithmetic
- four rounding modes

## 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  $2^N$  equally spaced values in each of  $2^E$  exponentially increasing intervals



ELEC2041 lec22-fraction.9

Saeid Nooshabadi

## Recall: Multiplication Instructions

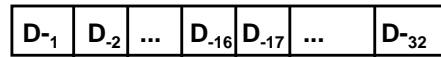
- ARM provides multiplication instruction
  - `mul Rd, Rm, Rs ; Rd = Rm * Rs`
  - (Lower precision multiply instructions simply throws top 32 bits away)

ELEC2041 lec22-fraction.10

Saeid Nooshabadi

## What about Multiplication for Fractions

- Imagine the binary point on the left



- 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.
- 2 bit example:  $0.11 * .10 = 0.0110 \approx 0.01$

We want to keep 01 and not 10

ELEC2041 lec22-fraction.11

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: `smull r4, r5, r3, r2; r5:r4:=r3*r2`
  - May not be generated by the general compilers. (May Need Hand coding).
  - DSP compilers generate them

- We can ignore the `RdLo` with some loss of accuracy

ELEC2041 lec22-fraction.12

Saeid Nooshabadi

## Multiplication for Int/Fractions in C

```
int mul_int(int a, int b)
{
    return (c * d);
}
// returns r0 after (mul r0, r1, r2). Upper part
// lost
```

Assume 16 bit integer  
and 16 fraction parts



```
int mul_fraction(int a, int b)
{
    return (int) (((long long) c * (long long) d) >> 32);
}
// returns r0 after (smull r3, r4, r1, r2) and mov r0,
// r4. R3 holds lower part, and R4 higher part
```



ELEC2041 lec22-fraction.13

Saeid Nooshabadi

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

- $1_2 = 2^0 = 1_{10}$
- $0.1_2 = 2^{-1} = 0.5_{10} = 1/2$
- $0.01_2 = 2^{-2} = 0.25_{10} = 1/4$
- $0.001_2 = 2^{-3} = 0.125_{10} = 1/8$
- $0.0001_2 = 2^{-4} = 0.0625_{10} = 1/16$
- $0.11_2 = 2^{-1} + 2^{-2} = 0.5_{10} + 0.25_{10} = 0.75_{10} = 1/2 + 1/4 = 3/4 = (1 - 1/4) = (1.0_2 - 0.01_2)$
- $0.101_2 = 2^{-1} + 2^{-3} = 0.5_{10} + 0.125_{10} = 1/2 + 1/8 = 0.625_{10}$
- $0.00110011001100 \dots_2 = 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + 2^{-15} + 2^{-16} + \dots = 1/8 + 1/16 + 1/128 + 1/256 + 1/2048 + 1/4096 + \dots = 0.125_{10} + 0.0625_{10} + 0.03125 + 0.015625 + 0.0009765625 + 0.00048828125 + \dots = 0.2_{10}$  (No Exact representation for 0.2 for finite precision)

ELEC2041 lec22-fraction.14

Saeid Nooshabadi

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

- $0.2_{10} = 0.00110011001100 \dots_2 \rightarrow$
- $0.1_{10} = 0.2_{10}/2 = 0.000110011001100 \dots_2$
- $0.3_{10} = 0.2_{10} + 0.1_{10} = 0.0011001100110011 \dots_2 + 0.0001100110011001 \dots_2 = 0.0100110011001100 \dots_2$

ELEC2041 lec22-fraction.15

Saeid Nooshabadi

## Toys & Tools

YOU NEVER KNEW YOU NEEDED IT

TVs that light up the room—even when they're off

- Philips Electronics NV this month began shipping flat-screen televisions that are also room lights. Philips's Ambilight technology projects background light from the rear of the television onto the wall, creating a halo around the television, which softly lights the room. The viewer can adjust the color choice and brightness via remote control (whether the television itself is on or off). Or the system can be set to an automatic mode, in which the lighting is continuously adjusted in relation to the image on the screen and to the overall brightness of the room, determined by built-in sensors.



Available in  
50-inch  
plasma  
FlatTV  
\$US10 000

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 ASCII.
- Repeat division with new quotient until it is zero

◦ **Example:** 10011001100110011001100110011001/10 =  
1111010111000010100011110101 (257698037) and  
Remainder of 111 (7) So:

◦ 1001100110011001100110011001 (2576980377)	
◦ 1111010111000010100011110101 (257698037)	7
◦ 1100010010011011101001011 (25769803)	7
◦ 1100010010011011101001011 (2576980)	3
	⋮
◦ 0	2

## 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++;  
}
```

## Uns. Int to Decimal ASCII Converter in ARM

```
utoa:  
; function entry: On entry a1 has the address of memory  
; to store the ASCII string and a2 contains the integer  
; to convert  
  
stmfd sp!, {v1, v2, lr}; save v1, v2 and ret. address  
mov v1, a1 ; preserve arg a1 over following func. calls  
mov a1, a2  
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...  
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
```

## “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  $2^N$  equally spaced values in each of  $2^E$  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).