CS500

Lecture 4

Excess-x notation

Excess-x notation

The excess-4 notation signed decimal range in 3-bit pattern is [– 4 , + 3].

the **leftmost bit** is reseved for the **sign**
0 means -ve
1 means +ve

BIT PATTERN	UNSIGNED DECIMAL VALUE	EXCESS NOTATION DECIMAL VALUE
0 0 0	0	– 4
0 0 1	1	– 3
0 1 0	2	– 2
0 1 1	3	– 1
1 0 0	4	0
1 0 1	5	+ 1
1 1 0	6	+ 2
1 1 1	7	+ 3

Range from -4 to +3

[- (2^{n-1}), + $(2^{n-1}-1)$]

Excess-x notation

- For n-bit binary pattern in excess-x, $\mathbf{x = 2^{(n-1)}}$

Example:

Excess-4: (x=4, n=??)

$4=2^2$ --> $2^2=2^{n-1}$ --> $n-1=2$ --> $n=3$

Excess-8: (x=8, n=??)

$8=2^3$ --> $3=n-1$ --> $n=4$

Excess-16: (x=16, n=??)

$16=2^4$ --> $4=n-1$ --> $n=5$

n=6:

$x=2^5=32$

then excess-32 is used

n=5:

$x=2^4=16$

then excess-16 is used

Excess-x to Decimal

The following formula is used to convert a signed binary number represented in excess-x to it corresponding decimal value.

$(\text{Excess Decimal Value})_{10}$ =

Unsigned Decimal Value (Binary number) - x

Example :

Convert each of the following binary excess-x notations in 4-bit to its equivalent signed decimal values:

a) 1 1 0 1.

b) 0 1 0 0.

c) 0 0 0 0.

Solution

For 4 bits, excess 8 is used. $(2^{n-1}=2^3=8)$

excess decimal value = unsigned decimal Value (binary number) – 8

- (1101)=13 --> 13-8=+5
- (0100)= 4 -->4-8=-4
- (0000)= 0-->0-8=0

Decimal to Excess-x

first, find the range of the excess-x for n-bits $[(-2^{n-1}), (+2^{n-1}-1)]$.

second, check whether the decimal number is in the range of the excess-x or not.

third, if it is out of range, no solution exist

if it is in the range,

(Signed Binary Value)Excess-x = Decimal number + x

Example:

Find the excess-4 notations of the following signed decimal values:

a) +5

b) -3

Solution

For excess-4, n=3 bits ($4=2^2$ --> $2=n-1$ --> $n=3$)

Since range= $[-(2^{n-1}), +(2^{n-1}-1)]$

Then for excess-4: the allowed range =[-4, +3]

a) +5

+5 is NOT in the allowed range

Thus +5 is out of range

b) -3

-3 is in the allowed range

-3+4=+1=001

Signed Binary 2's Complement: Addition & Subtraction

The signed 2's complement notation is the most commonly used notation to implement arithmetic operations in a computer.

Signed 2's complement addition:

Take the 2's complement of both addends and perform a normal addition including their sign bits.

A carry out of the sign bit position is discarded.

Signed 2's complement subtraction:

Take the 2's complement of the subtrahend (including the sign bit) and add it to the 2's complement of the minuend including the sign bit.

A carry out of the sign bit position is discarded.

Signed Binary 2's Complement: Addition & Subtraction: Example

Using the signed 2's complement notation, calculate the following arithmetic operations in 4-bits:

a) $+2+4$

b) $-2+4$

c) $+2-4$

d) $-2-4$

Signed Binary 2's Complement: Addition & Subtraction: Example1

Solution: In 4-bits, range= $[-2^3, +2^3-1] = [-8, +7]$

(+2) in 2's complement is 0010

(-2) in 2's complement is 1110

(+4) in 2's complement is 0100

(-4) in 2's complement is 1100

a) $+2 + 4 = +6$

$$0010 + 0100 = 0110$$

b) $-2 + 4 = +2$

$$1110 + 0100 = \cancel{1}0010$$

c) $+2 - 4 = -2$

$$0010 + 1100 = 1110$$

d) $-2 - 4 = -6$

$$1110 + 1100 = \cancel{1}1010$$

Signed Binary 2's Complement: Addition & Subtraction: Example 2

Assuming 8-bit signed 2's complement binary numbers, calculate the following operation: (– 6) – (– 13).

Solution: in 8-bit signed 2's complement binary numbers, we have range=[-128, +127]:

(+ 6) = 00000110 , (– 6) = 11111010.

(+ 13) = 00001101 , (– 13) = 11110011.

According to the signed 2's complement subtraction rule, we have:

(– 6) – (– 13) = (– 6) + (+ 13) =

$$= \begin{array}{r} 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0 \\ +\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 1 \\ \hline \boxtimes 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1 \end{array} = (+7)_{10}$$

Signed Binary 2's Complement: Addition & Subtraction: Overflow

Overflow Problem:

Assuming using 2's complement notation with 4-bit binary pattern, then the proper range for allowed signed binary numbers is [– 8 , + 7]. This means that the value + 9 is out of range and consequently has no pattern associated with it.

So when we calculate (+5) + (+4) in 4-bit 2's complement, the result in 4-bit 2's complement will not be (+9) **but it will be (-7) !!!** as follows:

+5		**Positive Number**		**0101 (+5) in 4-bit 2's complement**
+	+		+	
+4		**Positive Number**	**0100**	**(+4) in 4-bit 2's complement**
+9		**Negative Number!!**	**1001**	**(-7) in 4-bit 2's complement**
			0111	**(+7) in 4-bit 2's complement**

Signed Binary 2's Complement: Addition & Subtraction: overflow...

Overflow Problem:

Assuming using 2's complement notation with 4-bit binary pattern, then the proper range for allowed signed binary numbers is [– 8 , + 7]. This means that the value -9 is out of range and consequently has no pattern associated with it.

So when we calculate (-5) + (-4) in 4-bit 2's complement, the result in 4-bit 2's complement will not be (-9) **but it will be (+7) !!!** as follows:

-5	**Negative Number**	**1011 (-5) in 4-bit 2's complement**
+	**+**	**+**
-4	**Negative Number**	**1100 (-4) in 4-bit 2's complement**
-9	**Positive Number!!!**	**✗ 0111 (+7)in 4-bit 2's complement**

Signed Binary 2's Complement: Addition & Subtraction: Overflow

Overflow problem:

- This error is called **overflow** and it occurs when the result of an arithmetic operation lies outside the allowed range of the signed numeric data.
- Overflow in signed-2's complement notation occurs in the following two cases:
  a) **When adding two positive numbers, the result is negative.**
  b) **When adding two negative numbers , the result is positive.**
- To avoid the problem of overflow, the number of binary bits must be increased, to allow a quite wide range for representing signed numbers before the occurrence of overflow.

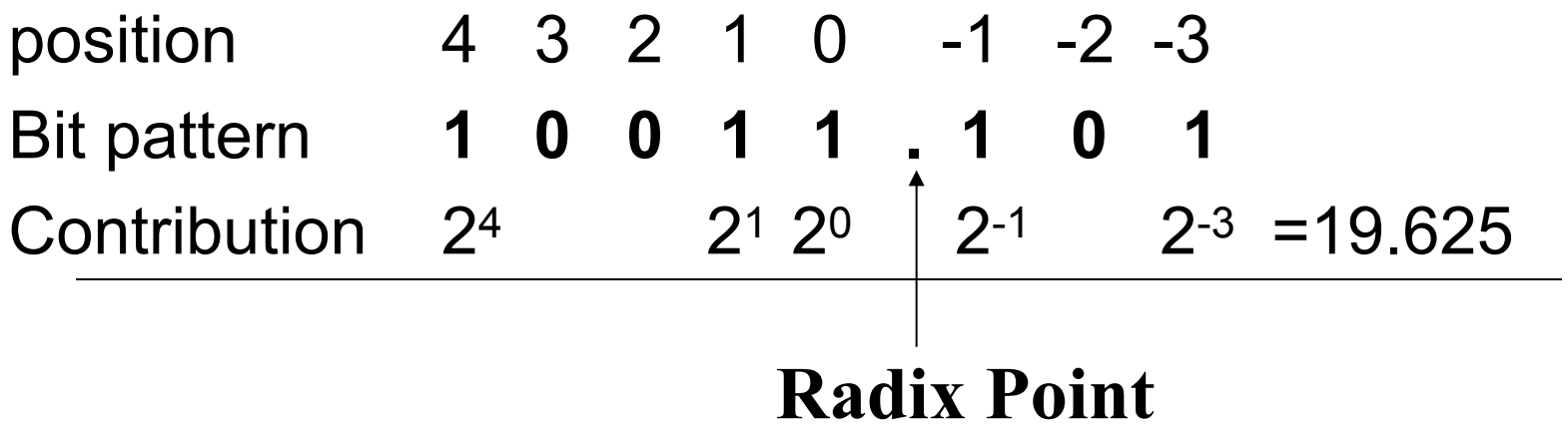
Signed Binary Fraction Number Representation

- Numbers used in scientific calculations are designed by a sign, by the magnitude of the number, and by the position of the radix point.

- There are two ways of specifying the position of the radix point which are:

  1. Fixed-point representation (which we have studied up till now) .
  2. Floating-point representation. (IEEE 32-bit)

Fixed-Point Representation



Limitation of Fixed-Point Representation:

To represent very large or very small numbers we **need a very long sequences of bits**,

this is because we have to give bits **to both the integer part and the fraction part.**

Floating-Point Representation

In Floating-point representation, there are two ways for positioning the radix point:

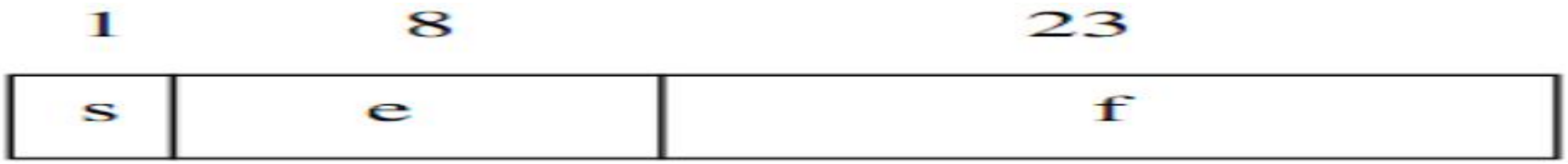
- 1- Putting the radix point at the **extreme left** of the number.
- 2-Putting the radix point at the **extreme right** of the number.

According to the first way, any number can be expressed as a fraction (mantissa) and a positive exponent. (e.g., **255.489 = 0.255489 * 10^{+3}**).

According to the second way, any number can be expressed as an integer and a negative exponent. (e.g., **255.489 = 255489.0 * 10^{-3}**).

IEEE-32Bits Single Precision Method

In this method, the single precision floating-point number is expressed using 32-bits



sign

1 bit

0: +ve

1: -ve

Exponent part

8-bits in excess-127 notation

e.g., +4

+4+127=131 $(10000011)_2$

Fraction part

23-bits in normalized form (1.f)

1 is hidden and appears only during conversion to decimal

IEEE-32Bits Single Precision Method...

Conversion from decimal number to IEEE-32bits binary floating-point number:

1. Take the integer part of the number and generate its binary equivalent.
2. Take the fractional part and generate its binary equivalent.
3. Put the two parts together with radix point in the middle between them.
4. Normalise the whole binary number to the form (**1.f**) by moving the radix point whether to the left or to the right.
5. Calculate the decimal exponent as the number of radix point movements.
6. Represent the decimal exponent into binary in excess-127 by adding it to 127, and then convert the addition result as unsigned integer to binary.
7. Fill in the sign bit with 0 if the decimal number is positive; otherwise with 1.
8. Fill in exponent part with the binary exponent in excess-127.
9. Fill in the fraction (mantissa) part with (**.f**) normalized binary bits.

IEEE-32Bits Single Precision Method: Example

Code the decimal **+.001275** as IEEE-32bits binary floating-point pattern.

Solution:

1- Sign +(.001275) is a positive number then the sign bit is 0.

2- Convert decimal number to binary

+ (.001275) = +(.000000000101)

3- Put the resulted binary number in the normalized form $s\ (1.f)\ *2^e$

+ (1.01) X 2^{-10}

4- The fraction part 'f' is (01)

5- The exponent part 'e' is (-10) in excess-127

-10+127=117= 01110101

1-bit Sign of Fraction (s)	8-bits Exponent (e)	23-bit Fraction (f)
0	01110101	01000000000000000000000

The binary pattern of floating-point format of (+.001275)10 is

$(0\ 011\ 1010\ 1\ 010\ 0000\ 0000\ 0000\ 0000\ 0000)_2 = (3AA00000)_{16}$

2 X	.001275	
	.00255	0
	.0051	0
	.0102	0
	.0204	0
	.0408	0
	.0816	0
	.1632	0
	.3264	0
	.6528	0
	.3056	1
	.6112	0
	.2224	1

IEEE-32Bits Single Precision Method: Example 2

Code the decimal **-6.75** as IEEE-32bits binary floating-point pattern**.**

Solution:

-(6.75) is a negative number then the sign bit is 1.

$-(6.75) = -(110.11) = -1.1011 * 2^{+2}$

The decimal exponent is +2, then its binary in excess-127 is:
+2 + 127=129 = 10000001.

The fraction is (.1011)

1-bit Sign of Fraction (s)	8-bits Exponent (e)	23-bit Fraction (f)
1	10000001	10110000000000000000

The binary pattern of floating-point format of (-6.75)10 is
$\mathbf{(1\ 10000001\ 10110000000000000000)_2 = (C0D80000)_{16}}$

Exercise

- Code the decimal +.002595 as IEEE-32bits binary floating-point pattern.

- Code the decimal value +47.763 as a binary pattern of floating-point format.

IEEE-32Bits Single Precision Method...

Conversion from IEEE-32bits binary floating-point number to decimal number:

1. Split the binary pattern from the left into 1-bit sign, 8-bits exponent and 23-bits fraction.
2. If the sign bit is 0 then the decimal number sign is positive; otherwise its negative.
3. For the binary exponent part, convert it to unsigned decimal number and then subtract 127 from it to get the decimal exponent.
4. Restore the binary mantissa (**.f**) in its normalized form (**1.f**)
5. Move the radix point whether to the left or to the right based on the decimal exponent sign.
6. Calculate the decimal equivalent of the integer part.
7. Calculate the decimal equivalent of the fraction part.

Example 1: Decode the hexadecimal pattern $\mathbf{(C3D1E000)_{16}}$ as IEEE-32 bits binary floating-point pattern to its equivalent decimal value.

Solution:

1- Convert the number to binary

$(C3D1E000)_{16}$ = 1100 0011 1101 0001 1110 0000 0000 0000

2- Put the resulted binary number in the IEEE format

1-bit Sign of Fraction (s)	8-bits Exponent (e)	23-bit Fraction (f)
1 (-ve)	10000111	10100011110000000000000

3- Convert exponent part to decimal (125-127=-2)

e = (10000111) in excess-127. Then e= 135 – 127 = +8

4- Restore the (1.f) normal form

1.1010001111

5- Substitute s (1.f) * 2^e

- (1.10100011110000000000000) * 2^{+8}

6- Move the radix point

- (110100011.10000000000000) = - (110100011.11)

7- Convert to binary

- 419.75

IEEE-32Bits Single Precision Method: Example 2

Example: Decode the hexadeciaml pattern $\mathbf{(41200000)_{16}}$ as IEEE-32bits binary floating-point pattern to its equivalent decimal value.

Solution:

$(41200000)_{16}$ = 0100 0001 0010 0000 0000 0000 0000

1-bit Sign of Fraction (s)	8-bits Exponent (e)	23-bit Fraction (f)
0 (+ve)	10000010	01000000000000000000000

Sign bit = 0, hence, it is positive number.

Binary exponent in excess-127= 10000010 = 130 –127 = +3.

The fraction (**.f**) is 01000000000000000000000

It is normalised, hence the true mantissa (**1.f**) is +1.01

The whole binary number is $+1.01 \times 2^{+3} = +1010.0$

Finally, the decimal number is **+10**

IEEE-32Bits Single Precision Method...

Possible Representations of Exponent

Binary	Unsigned Integer	Excess-127
00000000	0	-127 {reserved}
00000001	1	-126
00000010	2	-125
01111110	126	-1
01111111	127	0
10000000	128	1
10000001	129	2
11111110	254	127
11111111	255	128 {reserved}

IEEE-32Bits Single Precision Method: special cases

The maximum value (11111111) and the minimum value (00000000) of the e-field are reserved for the following exceptional conditions:

1. $e = 255$ and $f = 0$

   In this case, the number represents plus or minus infinity.

2. $e = 255$ and $f \neq 0$

   In this case, the representation is considered to be "not a number", i.e. NaN, regardless of the sign value. NaNs are used to signify invalid operations such as multiplication of zero with infinity.

3. $e = 0$ and $f = 0$

   In this case, the number denotes plus or minus zero.

4. $e = 0$ and $f \neq 0$

   In this case, the number has a magnitude which is less than the allowed minimum value.

IEEE-32Bits Single Precision Method: Reserved special Cases

Reserved Special Cases:

Positive Zero	0 00000000 00000000000000000000000
Negative Zero	1 00000000 00000000000000000000000
Positive Infinity	0 11111111 00000000000000000000000
Negative Infinity	1 11111111 00000000000000000000000
Positive Underflow	0 00000000 xxxxxxxxxxxxxxxxxxxxxxx
Negative Underflow	1 00000000 xxxxxxxxxxxxxxxxxxxxxxx
Not A Number (NaN)	x 11111111 xxxxxxxxxxxxxxxxxxxxxxx

Excercise

- Decode the hexadecimal pattern $(BEA1E000)_{16}$ as IEEE-32bits binary floating-point pattern to its equivalent decimal value.

- Decode the pattern 1 10000001 01001000000000000000000 into its equivalent decimal value.

Thank You