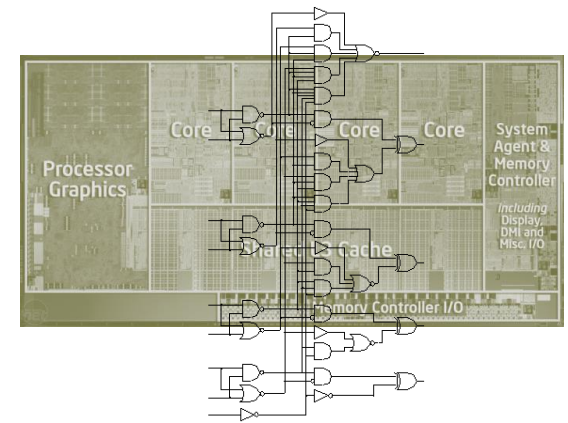




고려대학교
KOREA UNIVERSITY



COSE221, COMP211 Digital Logic Design

Lecture 9. Fixed and Floating Point Numbers

Prof. Taeweon Suh

Computer Science & Engineering

Korea University

What Would You Get?

```
#include <stdio.h>

int main()
{
    signed int sa = 7;
    signed int sb = -7;
    unsigned int ua = *((unsigned int *) &sa);
    unsigned int ub = *((unsigned int *) &sb);

    printf("sa = %d : ua = 0x%x\n", sa, ua);
    printf("sb = %d : ub = 0x%x\n", sb, ub);

    return 0;
}
```

What Would You Get?

```
#include <stdio.h>

int main()
{
    float f1 = -58.0;
    unsigned int u1 = *((unsigned int *) &f1);

    printf("f1 = %f\n", f1);
    printf("f1 = %3.20f\n", f1);
    printf("u1 = 0x%X\n", u1);

    return 0;
}
```

```
[suhtw@localhost temp]$ gcc float1.c -o float1
[suhtw@localhost temp]$ ./float1
f1 = -58.000000
f1 = -58.0000000000000000000000000000
u1 = 0xC2680000
```

What is this?

Korea Univ

[illegible]

4

What Would You Get?

```
#include <stdio.h>

int main()
{
    float f2 = -0.1;
    unsigned int u2 = *((unsigned int *) &f2);

    printf("f2 = %f\n", f2);
    printf("f2 = %3.20f\n", f2);
    printf("u2 = 0x%X\n", u2);

    return 0;
}
```

Why are these
different?

```
[suhtw@localhost temp]$ gcc float2.c -o float2
[suhtw@localhost temp]$ ./float2
f2 = -0.100000
f2 = -0.100000000149011611938
u2 = 0xBDCCCCD
```

And What is this?

What Would You Get?

```
#include <stdio.h>

int main()
{
    float f3 = 0.7;
    unsigned int u3 = *((unsigned int *) &f3);

    printf("f3 = %f\n", f3);
    printf("f3 = %3.20f\n", f3);
    printf("u3 = 0x%X\n", u3);

    return 0;
}
```

**Why are these
different?**

```
[suhtw@localhost temp]$ gcc float3.c -o float3
[suhtw@localhost temp]$ ./float3
f3 = 0.700000
f3 = 0.699999998807907104492
u3 = 0x3F333333
```

What is this?

0.07 != 0.07 ?

```
1 #include <stdio.h>
2
3 int main() {
4     double a, b, c, d;
5
6     a = 0.1;
7     b = 0.7;
8     d = 0.07;
9
10    c = a * b;
11
12    if (c == d) printf("%lf is equal to %lf\n", c, d);
13    else       printf("%lf is NOT equal to %lf\n", c, d);
14
15    return 0;
16 }
17 }
```

$a \times b \times c \neq b \times c \times a$?

```
1 #include <stdio.h>
2
3 int main()
4 {
5     float a, b, c;
6     float tmp1, tmp2;
7     float out1, out2;
8
9     a = 0.1;
10    b = 0.7;
11    c = 1.5;
12
13    tmp1 = a * b;
14    out1 = tmp1 * c;
15
16    tmp2 = b * c;
17    out2 = tmp2 * a;
18
19    if (out1 == out2) printf("out1 is equal to out2\n");
20    else               printf("out1 is Not equal to out2\n");
21
22    printf("out1 is %3.20f\n", out1);
23    printf("out2 is %3.20f\n", out2);
24
25    return 0;
26 }
```

We are going to answer to these questions!

Number Systems

- So far we have studied the following **integer** number systems in computer
 - Unsigned numbers
 - Signed numbers
 - Sign/magnitude numbers
 - Two's complement numbers
- What about **rational numbers**?
 - For example, 2.5, -10.04, 0.75 etc
- Two common notations to represent rational numbers in computer
 - **Fixed-point numbers**
 - **Floating-point numbers**

Fixed-Point Numbers

- Fixed point notation has an **implied binary point** between the integer and fraction bits
 - The binary point is not a part of the representation but is implied
 - Example:
 - Fixed-point representation of 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- The number of integer and fraction bits must be agreed upon by those generating and those reading the number
 - There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number

Signed Fixed-Point Numbers

- Like signed integer, **signed fixed-point numbers** can be represented in two ways
 - Sign/magnitude notation
 - Two's complement notation
- Example:
 - -2.375 using 8 bits (4 bits each to represent integer and fractional parts)
 - $2.375 = 0010 . 0110$
 - Sign/magnitude notation: **1010 0110**
 - Two's complement notation:
 1. flip all the bits: 1101 1001
 2. add 1:
$$\begin{array}{r} 1101\ 1001 \\ + \qquad 1 \\ \hline 1101\ 1010 \end{array}$$
- Addition and subtraction works easily in computer with 2's complement notation like integer addition and subtraction

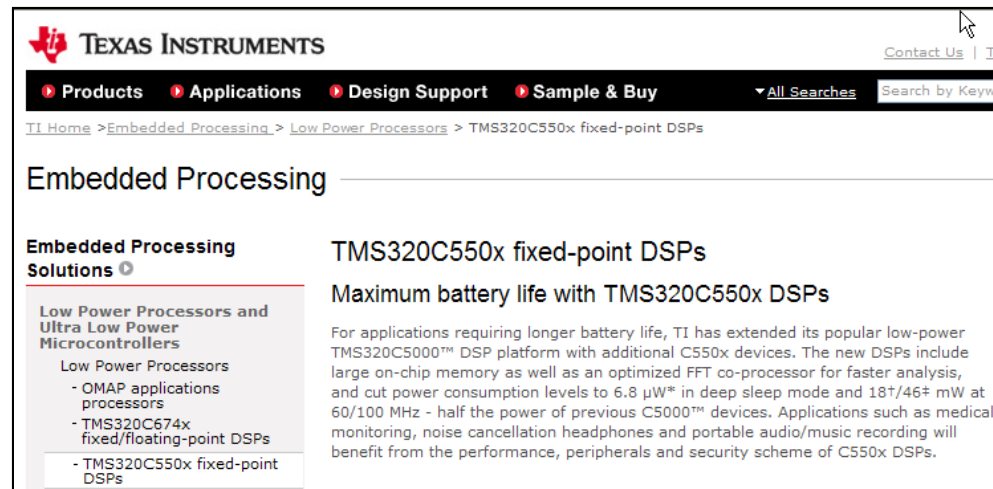
Example

- Suppose that we have 8 bits to represent a number
 - 4 bits for integer and 4 bits for fraction
- Compute $0.75 + (-0.625)$
 - $0.75 = 0000\ 1100$
 - $0.625 = 0000\ 1010$
 - -0.625 in 2's complement form: $1111\ 0110$

0.75	0000	1100
+ - 0.625	1111	0110
0.125	0000	0010

Fixed-Point Number Systems

- Fixed-point number systems have a limitation of having a constant number of integer and fractional bits
 - What are the largest and the smallest rational numbers you can represent with 32 bits, assuming 16 bits each for integer and fractional parts?
- Some low-end digital signal processors support fixed-point numbers
 - Example: TMS320C550x TI (Texas Instruments) DSPs: www.ti.com



Floating-Point Numbers

- Floating-point number systems circumvent the limitation of having a constant number of integer and fractional bits
 - They allow the representation of very large and very small numbers
- The binary point **floats** to the right of the most significant 1
 - Similar to decimal scientific notation
 - For example, write 273_{10} in scientific notation:
 - Move the decimal point to the right of the most significant digit and increase the exponent:
- In general, a number is written in scientific notation as:

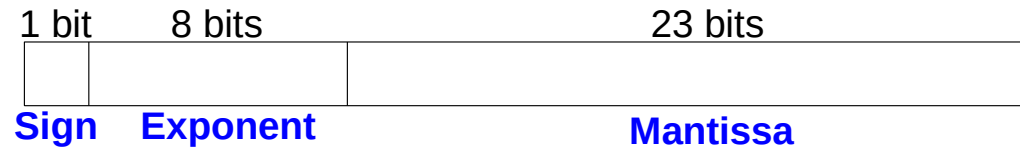
$$\pm M \times B^E$$

Where,

- M = mantissa
- B = base
- E = exponent
- In the example, M = 2.73, B = 10, and E = 2 (that is, $+2.73 \times 10^2$)

Floating-Point Numbers

- Floating-point number representation using 32 bits
 - 1 sign bit
 - 8 exponent bits
 - 23 bits for the mantissa.



- The following slides show **three** versions of floating-point representation with 228_{10} using a 32-bit
 - The final version is called the **IEEE 754 floating-point standard**

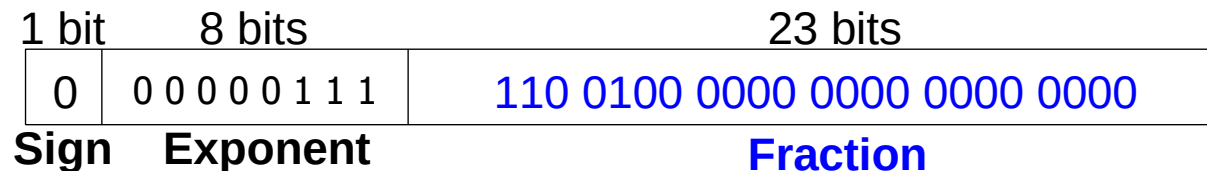
Floating-Point Representation #1

- First, convert the decimal number to binary
 - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Next, fill in each field in the 32-bit:
 - The sign bit (1 bit) is positive, so 0
 - The exponent (8 bits) is 7 (111)
 - The mantissa (23 bits) is 1.11001

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	Exponent	Mantissa

Floating-Point Representation #2

- You may have noticed that the first bit of the mantissa is always 1, since the binary point floats to the right of the most significant 1
 - Example: $228_{10} = 11100100_2 = \mathbf{1}.11001 \times 2^7$
- Thus, storing the most significant 1 (also called the implicit leading 1) is redundant information
- We can store just the **fraction parts** in the 23-bit field
 - Now, the leading 1 is implied



Floating-Point Representation #3

- The exponent needs to represent both positive and negative
- The final change is to use a **biased exponent**
 - The IEEE 754 standard uses a bias of 127
 - Biased exponent = bias + exponent
 - For example, an exponent of 7 is stored as $127 + 7 = 134 = 10000110_2$
- Thus , 228_{10} using the IEEE 754 32-bit floating-point standard is

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	Biased Exponent	Fraction

Example

- Represent -58_{10} using the IEEE 754 floating-point standard
 - First, convert the decimal number to binary
 - $58_{10} = 111010_2 = 1.1101 \times 2^5$
 - Next, fill in each field in the 32-bit number
 - The sign bit is negative (1)
 - The 8 exponent bits are $(127 + 5) = 132 = 10000100_{(2)}$
 - The remaining 23 bits are the fraction bits: $11010000..000_{(2)}$

1 bit	8 bits	23 bits
1	10000100	110 1000 0000 0000 0000 0000
Sign	Exponent	Fraction

- It is 0xC2680000 in the hexadecimal form
 - Check this out with the result of the sample program in the slide# 3

Floating-Point Numbers: Special Cases

- The IEEE 754 standard includes special cases for numbers that are difficult to represent, such as 0 because it lacks an implicit leading 1

Number	Sign	Exponent	Fraction
0	X	00000000	000000000000000000000000
∞	0	11111111	000000000000000000000000
$-\infty$	1	11111111	000000000000000000000000
NaN	X	11111111	non-zero

NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or $\log(-5)$

Floating-Point Number Precision

- The IEEE 754 standard also defines 64-bit double-precision that provides greater precision and greater range
 - **Single-Precision** (use the `float` declaration in C language)
 - 32-bit notation
 - 1 sign bit, 8 exponent bits, 23 fraction bits
 - bias = 127
 - It spans a range from $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$
 - **Double-Precision** (use the `double` declaration in C language)
 - 64-bit notation
 - 1 sign bit, 11 exponent bits, 52 fraction bits
 - bias = 1023
 - It spans a range from $\pm 2.22507385850720 \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$
- Most general purpose processors (including Intel and AMD processors) provide hardware support for double-precision floating-point numbers and operations

Double Precision Example

- Represent -58_{10} using the IEEE 754 double precision
 - First, convert the decimal number to binary
 - $58_{10} = 111010_2 = 1.1101 \times 2^5$
 - Next, fill in each field in the 64-bit number
 - The sign bit is negative (1)
 - The 11 exponent bits are $(1023 + 5) = 1028 = 10000000100_{(2)}$
 - The remaining 52 bits are the fraction bits: $11010000..000_{(2)}$
 - It is 0xC04D0000_00000000 in the hexadecimal form
 - Check this out with the result of the sample program in the slide# 4

Represent 0.7

- Represent 0.7 in IEEE 754 single precision form
 - $\frac{1}{2} = 0.5 = 0.1_{(2)}$ // $0.7 - 0.5 = 0.2$
 - $\frac{1}{8} = 0.125 = 0.001_{(2)}$ // $0.2 - 0.125 = 0.075$
 - $\frac{1}{16} = 0.0625 = 0.0001_{(2)}$ // $0.075 - 0.0625 = 0.0125$
 - $\frac{1}{128} = 0.0078125 = 0.0000001_{(2)}$ // $0.0125 - 0.0078125 = 0.0046875$
 - $\frac{1}{256} = 0.00390625 = 0.00000001_{(2)}$ // $0.0046875 - 0.00390625 = 0.00078125$
 -
 - Thus, $0.7 = 0.10110011..._{(2)} = 1.0110011..._{(2)} \times 2^{-1}$
 - In IEEE754 single precision, $0.7 = 0x3F333333$
 - Check it out with the slide#6
- IEEE754 floating-point standard can't represent some (I would say, most of) real numbers exactly

Binary Coded Decimal (BCD)

- Since floating-point number systems can't represent some numbers exactly such as 0.1 and 0.7, some application (calculators) use BCD (Binary coded decimal)
 - BCD numbers encode each decimal digit using 4 bits with a range of 0 to 9

Decimal	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD fixed-point notation examples

1.7 = 0001 . 0111

4.9 = 0100 . 1001

- BCD is very common in electronic systems where a numeric value is to be displayed, especially, in systems consisting solely of digital logic (not containing a microprocessor) - Wiki

Backup Slides

Floating-Point Numbers: Rounding

- Arithmetic results that fall outside of the available precision must round to a neighboring number
- Rounding modes
 - Round down
 - Round up
 - Round toward zero
 - Round to nearest
- Example
 - Round 1.100101 (1.578125) so that it uses only 3 fraction bits
 - Round down: 1.100
 - Round up: 1.101
 - Round toward zero: 1.100
 - Round to nearest: 1.101
 - 1.625 is closer to 1.578125 than 1.5 is

Floating-Point Addition with the Same Sign

- Addition with floating-point numbers is not as simple as addition with 2's complement numbers
- The steps for adding floating-point numbers with the same sign are as follows
 1. Extract exponent and fraction bits
 2. Prepend leading 1 to form mantissa
 3. Compare exponents
 4. Shift smaller mantissa if necessary
 5. Add mantissas
 6. Normalize mantissa and adjust exponent if necessary
 7. Round result
 8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition Example

Add the following floating-point numbers:

$$1.5 + 3.25$$

$$1.5_{(10)} = 1.1_{(2)} \times 2^0$$

$$3.25_{(10)} = 11.01_{(2)} = 1.101_{(2)} \times 2^1$$

$$1.1_{(10)} = 0x3FC00000 \text{ in IEEE 754 single precision}$$

$$3.25_{(10)} = 0x40500000 \text{ in IEEE 754 single precision}$$

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	Exponent	Fraction

For first number (N1): $S = 0, E = 127, F = .1$

For second number (N2): $S = 0, E = 128, F = .101$

2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

Floating-Point Addition Example

3. Compare exponents

$127 - 128 = -1$, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary

shift N1's mantissa: $1.1 \gg 1 = 0.11$ ($\times 2^1$)

5. Add mantissas

$$\begin{array}{r} 0.11 \times 2^1 \\ + 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

Floating-Point Addition Example

6. Normalize mantissa and adjust exponent if necessary

$$10.011 \times 2^1 = 1.0011 \times 2^2$$

7. Round result

No need (fits in 23 bits)

8. Assemble exponent and fraction back into floating-point format

$$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100..$$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	Exponent	Fraction

$$4.75_{(10)} = \text{0x40980000} \text{ in the hexadecimal form}$$