Xi'an Jiaotong-Liverpool University
西交利物浦大学

Data Representations

Jianjun Chen (Jianjun.Chen@xjtlu.edu.cn)

Table of Contents

- Number systems
  - Base-2, Base-10, Base-16
- Binary real numbers
- Numerical encoding
  - Two’s complement
  - IEEE 754

What You Will Learn

- How to convert numbers from/to binary, decimal and hexadecimal.
- Why certain numbers can never be represented precisely in computers.
  - With this knowledge, how would you adjust the habit of writing programs?
- How are numbers stored inside our memory.

Part 1: Number Systems

Base-2, Base-10, and Base-16 number systems and the conversion method

Number Systems

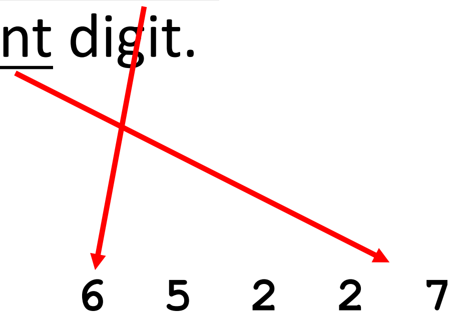
- Decimal, Binary and Hexadecimal are different number systems. A same value can be represented differently with them.

0xE	016	0b1110	14
Hexadecimal (Base-16)	Octal (Base-8)	Binary (Base-2)	Decimal (Base-10)

- Alternative writing style: $123_{10}, 1011_2, 2A_{16}$.
- Programmers commonly use hexadecimals to represent binary numbers because:
  - Short to write and align well with binary.
  - E.g. 0b1010110000000001 = 0xAC01 = 44033

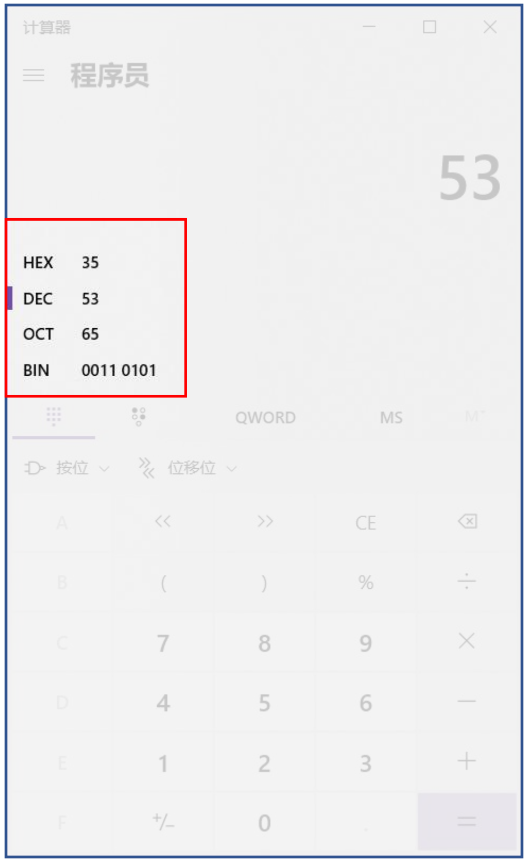
Conversion Method 1

1. Divide the number by BASE:
   - For example, if you want to convert a number into decimal, the divisor is 10.
2. Get the quotient, and the remainder.
3. If the quotient is 0, go to 4. Otherwise, apply step 1 to the quotient.
4. Covert reminders into the target number system.
5. Concatenate all converted reminders. The last reminder being the most significant digit and the first reminder being the least significant digit.



Conversion Example 1

- Let’s look at an example: 53 -> 0b00110101



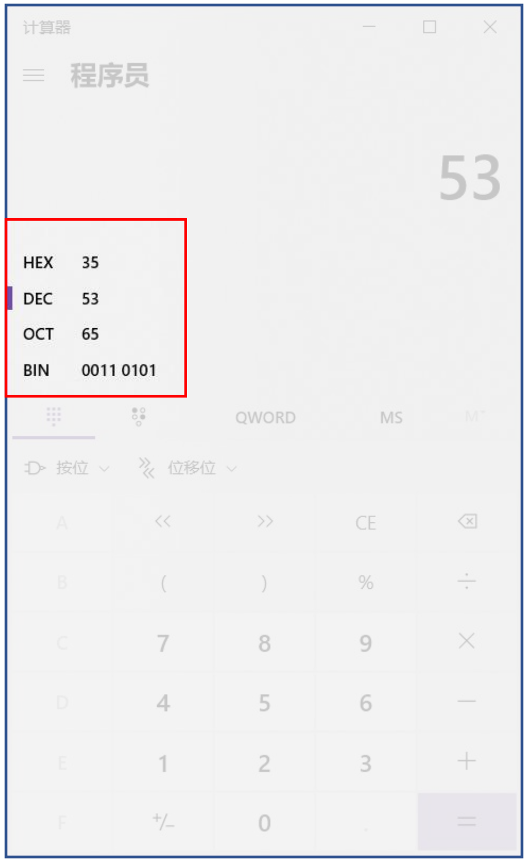
HEX 35
DEC 53
OCT 65
BIN 0011 0101

Conversion Example 2

- The opposite process (0b00110101 -> 53) is:
  - 10_{10} is 1010_2

Conversion Examples 3 & 4

- Another example: 53 -> 0x35
- And the opposite? 0x35 -> 53



```
HEX  35
DEC  53
OCT  65
BIN  0011 0101
```

Conversion Examples 3 & 4

- Another example: 53 -> 0x35
- And the opposite? 0x35 -> 53
  - Convert to binary first using the table on the right
  - Then use method 1
- Question:
  - 0x3 is 0b11
  - 0x5 is 0b101
  - Why 0x35 isn't 0b11101?

Hex	Bin	Dec
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7
0x8	1000	8
0x9	1001	9
0xA	1010	10
0xB	1011	11
0xC	1100	12
0xD	1101	13
0xE	1110	14
0xF	1111	15

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Conversion Method 2

- The process (0b00110101 -> 53) is simple:
  - 0b00110101 = 0b00000001 + 0b00000100 + 0b00010000 + 0b00100000

$$\begin{tikzcd}
2^0 & + & 2^2 & + & 2^4 & + & 2^5 \\
\text{0b00000001} \arrow[u] & & \text{0b00000100} \arrow[u] & & \text{0b00010000} \arrow[u] & & \text{0b00100000} \arrow[u]
\end{tikzcd}$$

Conversion Method 2

- The opposite way (53 -> 0b00110101):
  - $2^6 = 64$, too big.
  - $2^5 = 32$
  - $2^5 + 2^4 = 32 + 16 = 48$
  - …

Some Exercises

- Convert the following numbers to Binary:
  - 0x9A1C
  - 128
  - 7892
- Convert the following numbers to Decimal:
  - 0x11A3
  - 0b1010111
  - 0b1010101111111
- Please try both of the two methods you have learned. Which method is easier to follow for larger numbers? Smaller numbers?

Hex	Bin	Dec
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7
0x8	1000	8
0x9	1001	9
0xA	1010	10
0xB	1011	11
0xC	1100	12
0xD	1101	13
0xE	1110	14
0xF	1111	15

Xi'an Jiaotong-Liverpool University
西交利物浦大学

Discussion

- Why does method 1 work?
- Why are these the specific divisor used?
  - Converting to binary: 2
  - Converting to hexadecimal: 16
  - Converting to decimal: 10

Binary Real Numbers

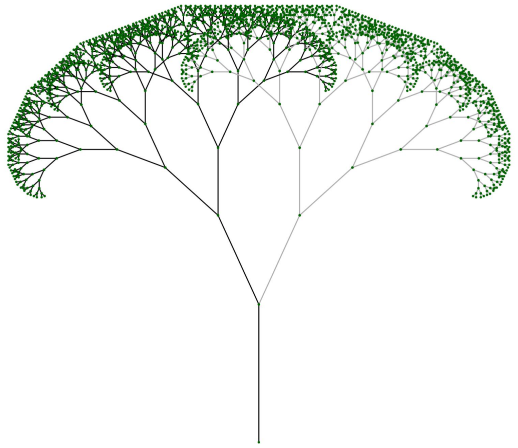
Converting between Base-2 and Base-10

Converting Real Numbers

- Converting between base-2 and base-10 real numbers is slightly more complicated.
- We had this example:

$$0b00110101 = 2^0 + 2^2 + 2^4 + 2^5 = 53$$

- But how about digits after the decimal point?
  - $101.11_2 \rightarrow\ ?_{10}$
  - $12.5_{10} \rightarrow\ ?_2$



- Let's look at a few examples:
  - $0.1_2 = 0.5_{10} = 2^{-1}$, because $0.1_2 + 0.1_2 = 1_2$
  - $0.01_2 = 0.25_{10} = 2^{-2}$, because $0.01_2 + 0.01_2 = 0.1_2$
  - $0.001_2 = 0.125_{10} = 2^{-3}$

Conversion Method 1

- For $101.11_2 \rightarrow\ ?_{10}$
  - $2^2 + 2^0 + 2^{-1} + 2^{-2} = 4_{10} + 1_{10} + 0.5_{10} + 0.25_{10} = 5.75_{10}$

- For $12.5_{10} \rightarrow\ ?_2$
  - $2^3 + 2^2 + 2^{-1}$
  - Just try from larger numbers to smaller numbers.
    - $2^4 = 16$, too big
    - $2^3 = 8 < 12.5$, perfect!
    - $2^3 + 2^2 = 12 < 12.5$, excellent!
    - …
  - Answer is 1100.1

Conversion Method 2

- $101.11_2 = \frac{10111_2}{100_2} = \frac{?_{10}}{?_{10}} = ?.??_{10}$

- $12.5_{10} = \frac{125_{10}}{10_{10}} = \frac{?_2}{?_2} = ?.??_2$

Question

- Can you represent following base-10 numbers with limited fraction digits?
  - 0.1 ?
  - 0.2 ?
  - 0.3 ?
  - 0.4 ?

- Convert 0.1_{10} into binary number with anyone of the methods mentioned earlier.

- Check you answer using this online tool: https://www.exploringbinary.com/floating-point-converter/

Further Experiment

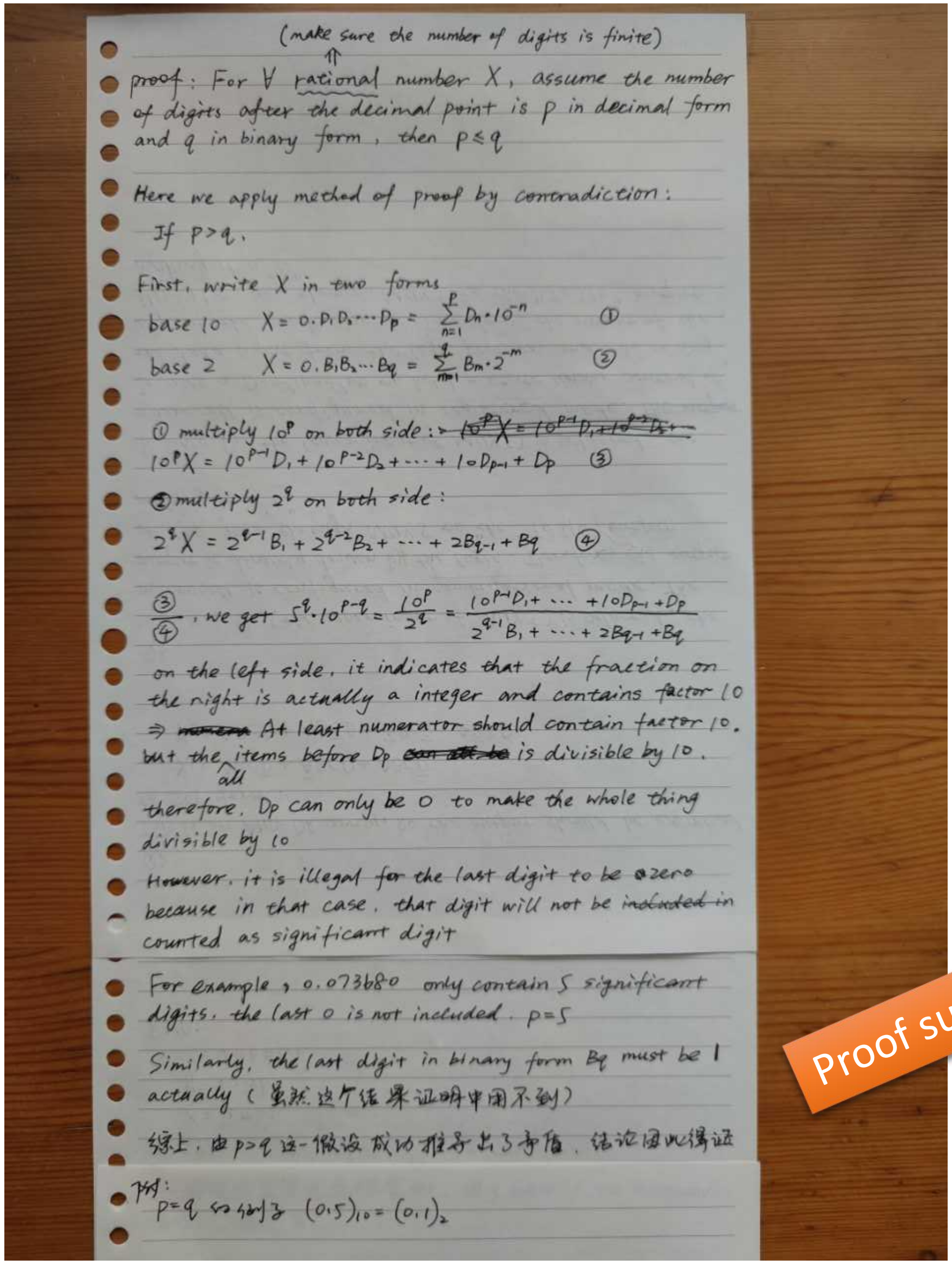
- Can you write a small C program to prove that 0.3 cannot be precisely represented by `float`?

An Unrelated Question

- For a same value X, assume that
  - The number of digits after the decimal point of its binary form is A.
  - The number of digits after the decimal point of its decimal form is B.

- Is A = B true? Or, which one of the following is true:
  1. $0.DDD_{10} \rightarrow 0.BBB_2$ (Same number of digits after the decimal point)
  2. $0.DDDDD_{10} \rightarrow 0.BBB_2$ (More digits for D)
  3. $0.DDD_{10} \rightarrow 0.BBBBB_2$ (Less digits for B)
  4. All of them may happen.

- D refers to the digits of decimal form
- B refers to the digits of binary form
- Assume that the number of Ds and Bs are always finite.
- Can you mathematically prove it?



(make sure the number of digits is finite)

proof: For $\forall$ rational number X, assume the number of digits after the decimal point is p in decimal form and q in binary form, then $p \leq q$

Here we apply method of proof by contradiction:

If $p > q$,

First, write X in two forms

base 10 $\quad X = 0.D_1D_2 \cdots D_p = \sum_{n=1}^{p} D_n \cdot 10^{-n}$ ①

base 2 $\quad X = 0.B_1B_2 \cdots B_q = \sum_{m=1}^{q} B_m \cdot 2^{-m}$ ②

① multiply 10^p on both side: ~~$10^p X = 10^{p-1}D_1 + 10^{p-2}D_2 +$~~

$10^p X = 10^{p-1}D_1 + 10^{p-2}D_2 + \cdots + 10 D_{p-1} + D_p$ ③

② multiply 2^q on both side:

$2^q X = 2^{q-1}B_1 + 2^{q-2}B_2 + \cdots + 2B_{q-1} + B_q$ ④

$\frac{③}{④}$, we get $5^q \cdot 10^{p-q} = \frac{10^p}{2^q} = \frac{10^{p-1}D_1 + \cdots + 10D_{p-1} + D_p}{2^{q-1}B_1 + \cdots + 2B_{q-1} + B_q}$

on the left side, it indicates that the fraction on the right is actually a integer and contains factor 10

$\Rightarrow$ ~~numera~~ At least numerator should contain factor 10.

but the all items before D_p ~~can all be~~ is divisible by 10.

therefore, D_p can only be 0 to make the whole thing divisible by 10

However, it is illegal for the last digit to be zero because in that case, that digit will not be ~~included in~~ counted as significant digit

For example, 0.073680 only contain 5 significant digits. the last 0 is not included. $p=5$

Similarly, the last digit in binary form B_q must be 1 actually (虽然这个结果证明中用不到)

综上，由$p>q$这一假设成功推导出了矛盾，结论因此得证

附：

$p=q$ 的例子 $(0.5)_{10} = (0.1)_2$

Proof submitted by Botao Yang in 2022 (not verified)

Part 2: Numerical Encodings

Integer and real numbers

Encoding

- You can map anything countable to numbers.
  - Emoji in Unicode: � 0x1F600, � 0x1F606, � 0x1F973
  - ASCII String: “ABC” 0x41, 0x42, 0x43; “321” 0x33, 0x32, 0x31
  - Colours: 🟪 0x352460, 🟩 0x1F6063

- But the encoding method must be shared to let others understand.
  - The same hex value 0x41 can be treated as ‘A’, or integer 65.

- The encoding method is affected by:
  - The target “language”: Emoji? String? Integers? Real numbers?
  - The limitations of the memory size.

These two will be covered

Memory Limitations

- In computer, data units usually use multiples of a byte as their sizes.
  - Integer can be 16 bits long (2 bytes) or 32 bits long (4 bytes).
  - Pointers can be 32 bits long or 64 bits long.
  - For other possibilities, check out this page.
  - In ARM, they are byte, half-word and word.
- Thus, the sizes of data units are limited.
  - How many values a 32-bit number can represent?
    - $2^{32} = 4294967296$ numbers
  - May need to allocate space for the sign bit (+/-)
- How should we arrange the space?
  - Integers: Two’s complement, one’s complement…
  - Real numbers: IEEE 754.

Address	Content
0xFFFFFFFF	0x8b
0xFFFFFFFE	0x11
.	.
.	.
.	.
.	.
0x00000001	0xAA
0x00000000	0xCD

Integer Encoding

- Positive signed integers and unsigned integers can use the standard base-2 system.

sign	b_6	b_5	b_4	b_3	b_2	b_1	b_0

$$b_7b_6b_5b_4b_3b_2b_1b_0$$
$$= b_7 \times 2^7 + b_6 \times 2^6 + b_5 \times 2^5 + \ldots b_0 \times 2^0$$

- But how about negative integers?
  - -0b00010001? Certainly not!
  - We need to sacrifice one bit to represent the sign.

- This leads to a simple solution to signed integer encoding, called **Sign and Magnitude**.
  - Not used in practice.

Sign and Magnitude

- Designate the high-order bit (MSB) as the “sign bit”
  - Sign = 0: positive numbers.
  - Sign = 1: negative numbers.
- The rest of the bits are magnitude (absolute value)
- Examples:
  - 0x00 = 0b00000000 = zero
  - 0x7F = 0b01111111 = 127
  - 0x81 = 0b10000001 = -1
  - 0x80 = 0b10000000 = negative zero ???!!!
- Problems:
  - Two zero values (equality checking problem)
  - Arithmetic unit design is more complicated. Requires arithmetic units for both + and -.
    - 4-3 != 4+(-3)

4	0100
- 3	- 0011
1	0001

4	0100
+ -3	+ 1011
1	1111

One’s Complement

- It will be good if subtraction units are removed and only use addition arithmetic units.
  - This is the idea of one’s complement representation.

- Negative numbers are obtained by flipping the bits of its signed version.
  - 0001 -> 1110

- Why?
  - 0001 + 1110 = 1111 = zero!

-0	1111	0000	+0
-1	1110	0001	+1
-2	1101	0010	+2
-3	1100	0011	+3
-4	1011	0100	+4
-5	1010	0101	+5
-6	1001	0110	+6
-7	1000	0111	+7

One’s Complement: Issues

- Try to add any two numbers from the right table:
  - 3 + 4
  - -2 + 3
  - -1 + (-5)
- What problem is encountered?
- How to solve this problem?

-0	1111	0000	+0
-1	1110	0001	+1
-2	1101	0010	+2
-3	1100	0011	+3
-4	1011	0100	+4
-5	1010	0101	+5
-6	1001	0110	+6
-7	1000	0111	+7

One’s Complement: Issues

- Any addition operation that involves negative numbers are one bit off:
  - 3 + 4 = 7, correct!
  - -2 + 3 = 0, incorrect!
  - -1 + (-5) = -7, incorrect!
- How to solve this problem?
  - When negative addition is involved, increase the result by 1.

-0	1111	0000	+0
-1	1110	0001	+1
-2	1101	0010	+2
-3	1100	0011	+3
-4	1011	0100	+4
-5	1010	0101	+5
-6	1001	0110	+6
-7	1000	0111	+7

Xi'an Jiaotong-Liverpool University 西交利物浦大学

Two’s Complement

Two’s complement solves this issue at the root:

- Instead of adding 1 when addition with a negative number is involved.
- 1 will be added when negative numbers are encoded

- **Two’s complement for negative numbers**: Flip the bits of a positive number, then add 1.

-1	1111	0000	+0
-2	1110	0001	+1
-3	1101	0010	+2
-4	1100	0011	+3
-5	1011	0100	+4
-6	1010	0101	+5
-7	1001	0110	+6
-8	1000	0111	+7

Two's Complement: Advantages

- Roughly same number of positive and negative numbers
- Positive number encodings match unsigned
- Only one zero representation
- Negation is simple:

```
~x + 1 == -x
```

- Number addition/subtraction in two's complement is easy
  - Positive + positive: Just like base-2 number addition.
  - Positive + negative: Same as positive + positive
  - Positive - positive: Just change to positive + negative then do the addition.
  - …
- Only addition arithmetic unit is needed!
  - Still needs overflow detection though ☺

Two’s Complement: Addition Examples

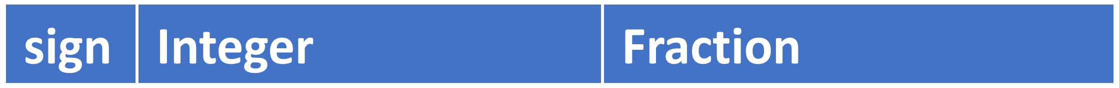
- For simplicity, we assume a 4 bit memory unit limitation. Please use addition to calculate the results of the following numbers and their absolute values.
  - 2 + 5 = ?
  - 5 + 7 = ?
  - 5 - 8 = ?
  - -2 + (-6) = ?
  - -5 + (-6) = ?
- How overflow or underflow are reflected in these bits?

-1	1111	0000	+0
-2	1110	0001	+1
-3	1101	0010	+2
-4	1100	0011	+3
-5	1011	0100	+4
-6	1010	0101	+5
-7	1001	0110	+6
-8	1000	0111	+7

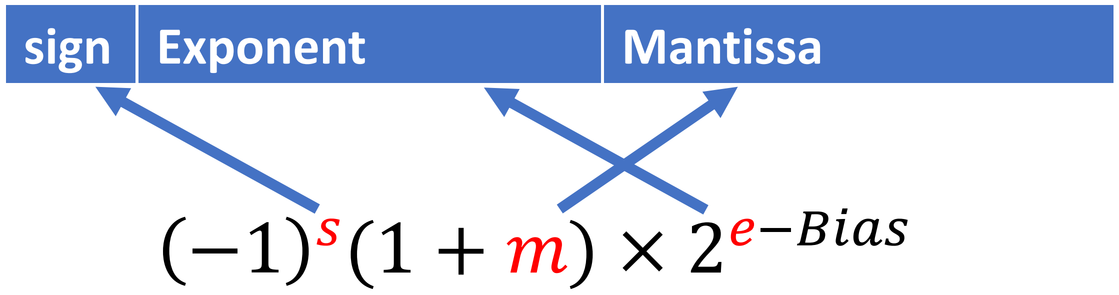
IEEE 754 Floating Point

There are two ways to encode real numbers.

- Fixed point: This representation has fixed number of bits for integer part and for fractional part.
  - +0000.0001
  - -9999.9999
  - `int` and `long` are special forms of fixed point numbers without any fraction bits.
  - Higher arithmetic performance but limited range of values.



- Floating point: The "binary point" floats in this representation form. It is the "binary form of scientific notation"
  - -1.00101×2^5
  - 1.01111×2^{12}
  - Widely used.



$$(-1)^s(1+m) \times 2^{e-Bias}$$

IEEE 754

- About IEEE 754:
  - A standard introduced in 1985 to make numerically-sensitive programs portable.
  - Specifies two things: representation scheme and result of floating point operations
- Representation scheme (all in binary):
  - Sign bit: 0 is positive,; 1 is negative
  - Mantissa: the fractional part of the number in normalised form.
  - Exponent: weights the value by a (possibly negative) power of 2.
- Precision levels:
  - Half Precision (16 bit): 1 sign bit, 5 bit exponent, and 10 bit mantissa.
  - Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
  - Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa
  - Quadruple Precision (128 bit): 1 sign bit, 15 bit exponent, and 112 bit mantissa

Exponent

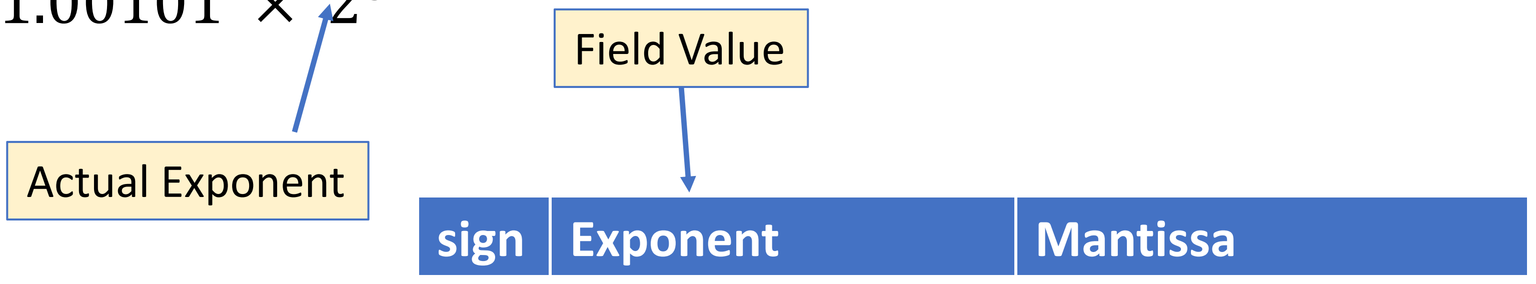
- Assume that we are going to use single precision (32 bit).
  - thus 8 bits for the exponent.
- Then $bias = 2^{8-1} - 1$ = 127.

Field value = Actual Exponent + bias

Actual Exponent = Field value - bias

	Binary	Exp Field value	Actual Exp
	$0000\ 0001_2$	1	-126
	$0000\ 0010_2$	2	-125
127 slots	.	.	.
	.	.	.
	$0111\ 1111_2$	127	0
	$1000\ 0000_2$	128	1
	$1000\ 0001_2$	129	2
127 slots	.	.	.
	.	.	.
	$1111\ 1110_2$	254	127

$$-1.00101 \times 2^5$$



IEEE 754: Special Values

- All the exponent bits 0 with all mantissa bits 0 represents 0. If sign bit is 0, then +0, else -0.
- All the exponent bits 1 with all mantissa bits 0 represents infinity. If sign bit is 0, then +∞, else -∞.
- All the exponent bits 0 and mantissa bits non-zero represents denormalized number.
- All the exponent bits 1 and mantissa bits non-zero represents error.

Practice on the IEEE 754 standard

$0b\,1100.1 \Rightarrow 1.1001 \times 2^{3} : E$

M

$0.5 \times 2 = 1$

$2 \,|\, 12.5$

$2 \,|\, 6 - 0$

$2 \,|\, 3 - 0$

$2 \,|\, 1 - 1$

$0 - 1$

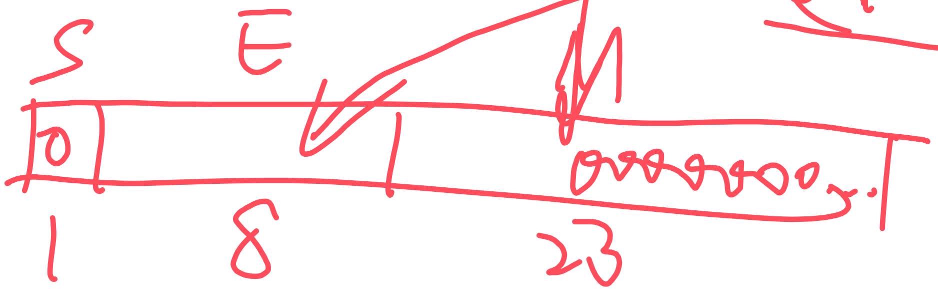
- Convert the following numbers into IEEE 754 single precision representation:
  - $12.5_{10} = 1100.1_2$
  - 0.625_{10}

Field Value.

$3 + 127 = 130$

- Verify your answer using the following tool: https://www.h-schmidt.net/FloatConverter/IEEE754.html.

$\Rightarrow 0\,10000010$



Xi'an Jiaotong-Liverpool University 西交利物浦大学

IEEE 754: Precision Issues

- The exponent field of IEEE 754 representation affects the precision of numbers.
- The code below illustrates the gaps between floating numbers:

```
double num = 1000000000000000.0;
double addition = 0.0;
for (int i = 0; i < 30; i++) {
    num += 1.05;
    addition += 1.05;
    System.out.println("loop " + i + " num: " + num + " addition: " +
addition);
}
```

```
• loop 0 num: 1.000000000000001E15 addition: 1.05
• loop 1 num: 1.000000000000002E15 addition: 2.1
• loop 2 num: 1.000000000000003E15 addition:
  3.1500000000000004
```

A Note about C/C++ and Assembly

- At hardware level, the CPU **does not care about the data type** of a certain piece of memory. It only carries out operations based on the instructions sent to it. The instructions are generated by compilers, based on the data type of variables and their operations.

```
int f(int a)
{
    return a > 0;
}

int f2(unsigned int a)
{
    return a > 0;
}
```

```
str      r0, [fp, #-8]
ldr      r3, [fp, #-8]
cmp      r3, #0
movgt    r3, #1
movle    r3, #0
and      r3, r3, #255
```

```
str      r0, [fp, #-8]
ldr      r3, [fp, #-8]
cmp      r3, #0
movne    r3, #1
moveq    r3, #0
and      r3, r3, #255
```