



Xi'an Jiaotong-Liverpool University

西交利物浦大學

# Data Representations

# Table of Contents

- Number systems
  - Base-2, Base-10, Base-16
- Numerical encoding
  - Two's complement
  - IEEE 754

# What You Will Learn

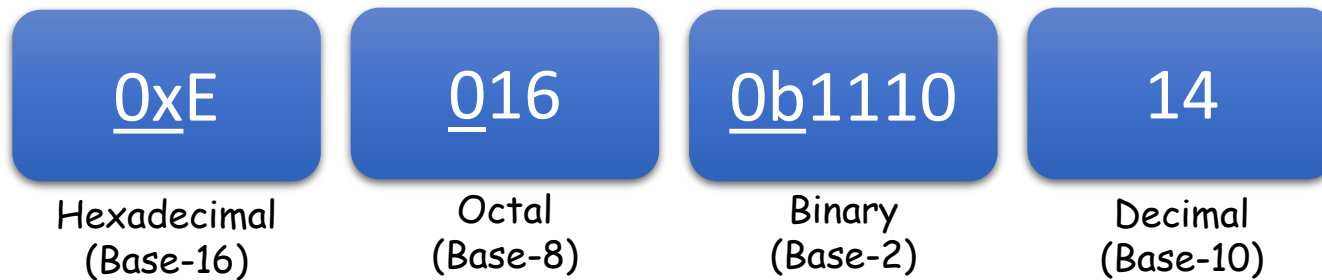
- How to convert numbers from/to binary, decimal and hexadecimal.
- Why certain numbers can never be represented precisely in computers.
  - With this knowledge, how would you adjust the habit of writing programs?
- How are numbers stored inside our memory.

# 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.



- 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.  $0b101011000000001 = 0xAC01 = 44033$

# Number Representation

- A number is represented by a string of digits where the position of each digit is associated with a weight

Digit	$A_n$	$A_0$	•	$A_{-1}$	$A_{-2}$
Weight	$B^n$	$B^0$	•	$B^{-1}$	$B^{-2}$

$A_n$  = digit

$B$  = base

$B^n$  = weight

- Example: 0b1110.001

Digit	1	1	1	0	•	0	0	1
Weight	8	4	2	1	•	0.5	0.25	0.125

# Any Base to Decimal

- Conversion from any base to decimal:

$$N = A_n B^n + A_{n-1} B^{n-1} + \dots + A_1 B^1 + A_0 B^0 + A_{-1} B^{-1} + \dots$$

## Examples:

- $127 = 1 \times 10^2 + 2 \times 10^1 + 7 \times 10^0 = \mathbf{127}$
- $78.45 = 7 \times 10^1 + 8 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} = \mathbf{78.45}$
- $0b1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = \mathbf{11}$
- $0b1001.01 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = \mathbf{9.25}$
- $0x26A = 2 \times 16^2 + 6 \times 16^1 + 10 \times 16^0 = \mathbf{618}$
- $0x12.1 = 1 \times 16^1 + 2 \times 16^0 + 1 \times 16^{-1} = \mathbf{18.0625}$

# Decimal to Any Base (Whole Numbers)

1. Divide the number by target BASE
2. Get the quotient, and the remainder.
3. If the quotient is 0, go to 4. Otherwise, apply step 1 to the quotient.
4. Convert 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.

6 5 2 2 7

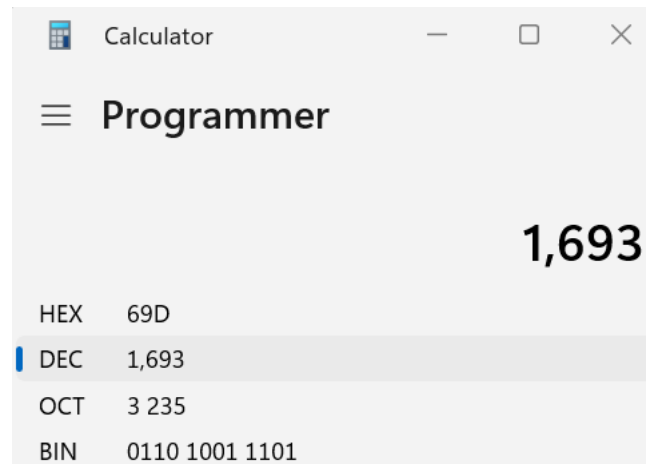
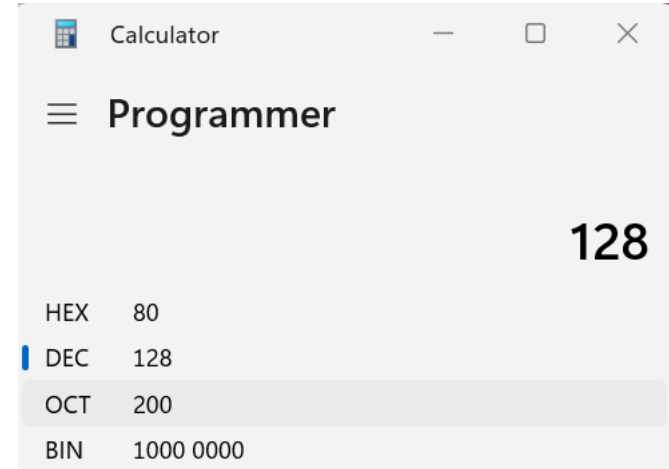
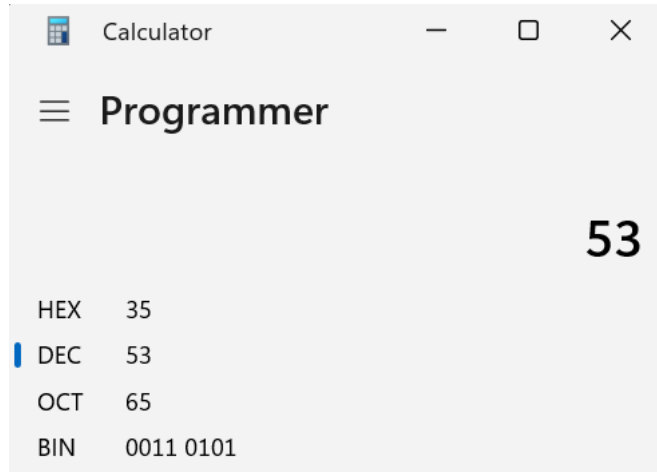




# Examples

- Convert 53 to binary
- Convert 128 to octal
- Convert 1693 to hex

# Answers



# Binary, Octal, Hexadecimal

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

- Conversion between  $2^k$  bases has interesting property
- All numbers within these systems can be uniquely represented by  $k$  binary bits
- Octal numbers can be represented by three bits
- Hex numbers can be represented by four bits

$$11010110_2 = \overbrace{011} \overbrace{010} \overbrace{110}_2 = 326_8$$

$$= \overbrace{110} \overbrace{101} \overbrace{110}_2 = D6_{16}$$

$$11010010.10110_2 = \overbrace{011} \overbrace{010} \overbrace{001} \overbrace{0.101} \overbrace{100}_2 = 322.54_8$$

$$= \overbrace{110} \overbrace{100} \overbrace{10.101} \overbrace{1}_2 = D2.B_{16}$$

$$273_8 = \overbrace{010} \overbrace{111} \overbrace{011}_2$$

$$= \overbrace{101} \overbrace{110} \overbrace{11}_2$$

$$= BB_{16}$$

# Decimal to Any Base (Fractions)

1. Multiply the number by target BASE
2. Get the fractional product, and the carry.
3. If the fractional product is 0 or the desired number of decimal places is reached, go to 4. Otherwise, apply step 1 to the fractional product.
4. Convert carries into the target number system.
5. Concatenate all converted reminders. The last carry being the least significant digit and the first carry being the most significant digit.

# Examples

- Convert 5.625 to binary
- Convert 17.265625 to octal
- Convert 1693.0628 to hex

# Answers

Binary (base 2)	101.101
Octal (base 8)	5.5
<b>Decimal (base 10)</b>	5.625
Hexadecimal (base 16)	5.A



Binary (base 2)	1 0001.0100 01
Octal (base 8)	21.21
<b>Decimal (base 10)</b>	17.265625
Hexadecimal (base 16)	11.44

Binary (base 2)	110 1001 1101.0001 0000 0001 0011 1010 1001 0010 1010 0011 0000 0101 0101 0011 0010 0110 0001 1
Octal (base 8)	3235.0401 1651 1243 0125 1446 1
<b>Decimal (base 10)</b>	1693.0628
Hexadecimal (base 16)	69D.1013 A92A 3055 326

# 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 multiplies 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
0xFFFFFFF0	0x11
.	.
.	.
.	.
.	.
0x00000001	0xAA
0x00000000	0xCD

# Integer Encoding

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

sign	$b_6$	$b_5$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$
------	-------	-------	-------	-------	-------	-------	-------

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

- 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:

- $0x00 = 0b00000000 = \text{zero}$
- $0x7F = 0b01111111 = 127$
- $0x81 = 0b10000001 = -1$
- $0x80 = 0b10000000 = \text{negative zero} ???!!!$

4	0100
- 3	- 0011
<hr/>	<hr/>
1	0001

- Problems:

- Two zero values (equality checking problem)
- Arithmetic unit design is more complicated.  
Requires arithmetic units for both + and -.
  - $4-3 \neq 4+(-3)$

4	0100
+ -3	+ 1011
<hr/>	<hr/>
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?

- $0001 + 1110 = 1111 = \text{zero!}$

-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

# 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:

$$\sim x + 1 == -x$$

- 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.

- $2 + 5 = ?$
- $5 + 7 = ?$
- $5 - 8 = ?$
- $-2 + (-6) = ?$
- $-5 + (-6) = ?$

-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

- How overflow or underflow are reflected in these bits?

# 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”

- $-1.00101 \times 2^5$

- $1.01111 \times 2^{12}$

- 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

$$-1.00101 \times 2^5$$

Actual Exponent

Field Value



	Binary	Exp Field value	Actual Exp
127 slots	0000 0001 <sub>2</sub>	1	-126
	0000 0010 <sub>2</sub>	2	-125
	.	.	.
	.	.	.
	0111 1111 <sub>2</sub>	127	0
127 slots	1000 0000 <sub>2</sub>	128	1
	1000 0001 <sub>2</sub>	129	2
	.	.	.
	.	.	.
	1111 1110 <sub>2</sub>	254	127

# 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  $+\infty$ , else  $-\infty$ .
- 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

- Convert the following numbers into IEEE 754 single precision representation:
  - $12.5_{10} = 1100.1_2$
  - $0.625_{10}$
- Verify your answer using the following tool: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>.

# 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);
}
```

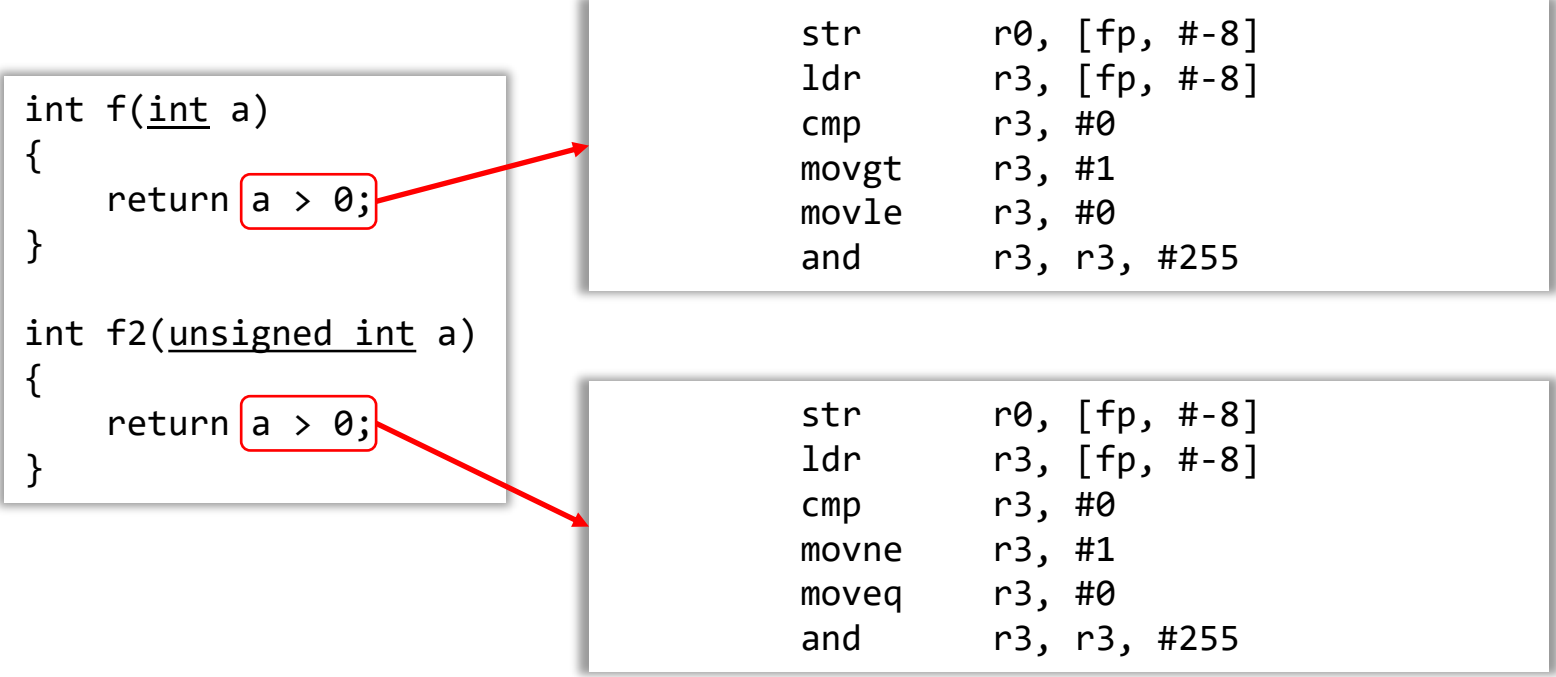
- loop 0 num: 1.000000000000000001E15 addition: 1.05
- loop 1 num: 1.000000000000000002E15 addition: 2.1
- loop 2 num: 1.000000000000000003E15 addition:  
3.150000000000000004

# 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
```