2 Representation of Numbers

2.0 Why binary?

Scales: **Binary**（二进制）, **Octal**（八进制）, **Decimal**（十进制）, **Hexadecimal**（十六进制）.

Given a number N, we can use R-scale to represent it, that will cost $\log_R N$ digits. The basic number of R-scale ($0, 1, \dots, R-1$) should be represent in different ways, so each digit should cost R space, then the total space cost D can be represented as:

$$D = R \log_R N = \ln N \frac{R}{\ln R}$$
$$\frac{\partial D}{\partial R} = \ln N \cdot \frac{\ln R - 1}{(\ln R)^2}$$

When $R = e = 2.71828...$, $\frac{\partial D}{\partial R} = 0$, which means D reaches its minimum.

Since $R \in \mathbb{N}$, we can get the optimal solution: $R = 2$ or $R = 3$.

Binary is more convenient for circuit design than ternary, so we choose binary to represent data in computer.

2.1 Encoding of Integer

$$m = X_0 X_1 X_2 \dots X_n$$

- Signed magnitude（原码）
  - X_0 represent the sign of the integer m (0: *non-negative*, 1: *non-positive*).
  - $X_1 X_2 \dots X_n$ is the binary form of $|m|$.
  - As a result, the number 0 has two codes $100...00$ or $000...00$.
  - Not convenient for calculation.
- One's complement（反码）
  - X_0 represent the sign of the integer m (0: *non-negative*, 1: *non-positive*).
  - $X_1 X_2 \dots X_n$ is the binary form of the number m if $X_0 = 0$;
  - $X_1 X_2 \dots X_n$ is the bitwise NOT of the binary form of the number $-m$ if $X_0 = 1$.
  - As a result, the number 0 has two codes $000...00$ or $111...11$.
  - Not convenient enough for calculation, either.
- Two's complement （补码）
  - X_0 represent the sign of the integer m (0: *non-negative*, 1: *non-positive*).
  - $X_1 X_2 \dots X_n$ is the binary form of the number m if $X_0 = 0$;
  - $X_1 X_2 \dots X_n$ is 1 plus the bitwise NOT of the binary form of the number $-m$ if $X_0 = 1$.
  - Actually, $X_1 X_2 \dots X_n$ is the binary form of the number $(2^n + m)$ if $X_0 = 1$, so $X_0 X_1 X_2 \dots X_n$ is actually $(2^{n+1} + m)$. That is,

    $$[m]_{TC} = [m]_2 \quad (0 \le m < 2^n)$$
    $$[m]_{TC} = [2^{n+1} + m]_2 \quad (-2^n \le m < 0)$$

    The two's complement has the same effect as *mod* 2^{n+1}.
  - As a result, the number 0 only has one code $000...00$.
  - How to write a negative number in two's complement?

- From the lowest digit of its absolute number's binary code, when we encounter `0` and the first `1`, we do not change their digits; then we change the digits after the first `1` to their opposite numbers (`0` to `1`, `1` to `0`). （从其绝对值的二进制编码的最低位开始，遇到的0和第一个1不变，之后的所有数取反。）
  - **CAN NOT** compare two numbers in two's complement directly.
  - $[X]_{TC} + [-X]_{TC} = 2^{n+1} = [0]_{TC}$
  - $[X+Y]_{TC} = [X]_{TC} + [Y]_{TC}$

    $[X-Y]_{TC} = [X]_{TC} + [-Y]_{TC}$

> [*Example*] (different encoding methods in computer)
>
> $$[-102]_{10} = [11100110]_S = [10011001]_{OC} = [10011010]_{TC}$$
>
> where S stands for *Signed magnitude*, OC stands for *One's complement* and TC stands for *Two's complement*.
>
> So -102 has the code of 10011010 in two's complement, as a result, it is stored as $[9A]_H = [9A]_{16}$ in computer.

> [*Example*] The encoding method of *int* in the computer is *two's complement*.
>
> $[100...00]_{TC} = -2147483648 = -2^{31}$, $[011...11]_{TC} = 2147483647$, so the range of *int* in computer is $[-2147483648, 2147483647]$.

2.2 Encoding of Unsigned Integer

$$m = X_0X_1X_2 \ldots X_n$$

The range is $[0, 2^{n+1} - 1]$, it is also same as *mod* 2^{n+1}.

The two's complement of a signed number can also be regard as a unsigned number, which means the code does not change! But the meaning of the code changes, a negative number becomes a quite big positive number.

> [*Example*] **Warning** In C compiler, if the two operands are a signed number and an unsigned number, the signed number will be implicitly transformed to unsigned number.
>
> ```
> unsigned int length = 0;
> for (int i = 0; i <= length - 1; ++ i)
>     // do something ...
> ```
>
> In the program above, `length` is an unsigned number while `i` and `1` are a signed number, the compiler will automatically transform `1` to unsigned number, and calculate `length - 1` which is `0-1` in unsigned number, so the result will be $2^{32} - 1$, which is the maximum number of *unsigned int*. The comparison between `i` and `length - 1` will also be treated as unsigned number comparison, so the loop will never end because all the *unsigned int* number is not greater than $2^{32} - 1$.

How to use *unsigned* in programming?

- Use *unsigned int* to represent set (subset).
- Use *unsigned int* as a modulo system.

2.3 Bitwise Operators

Operator `&`, `|`, `^`, `~`, `<<`, `>>` in C language are bitwise operator. They can fully use the feature of the binary number. Here are some useful methods to use the bitwise operator:

- Masking（掩码）：Use operator `&` to extract some certain digit of an number:

  > [*Example*] `0x8C & 0x0F = 0x0C` extract the digit `C`; `0x238C & 0x0FF0 = 0x0380` extract the digits `38`.

- Set or check in certain digits: Use operator `&` or `|` to set 1 or 0 in some certain digit, or check if certain digit is 0 or 1.

  > [*Example*] `0x0C | 0xF0 = 0xFC`; `0x238C & 0x0FFF = 0x038C`;
  >
  > Check if the last binary digit of `x` is 1: `x & 1`.

- Represent a set: The i-th binary digit represent whether p_j is in the set. The `&` operator can be used to get the *intersection* of two sets; the `|` operator can be used to get the *union* of two set; the `^` operator can be used to get the *symmetric difference* of two sets.

  > [*Example*] $\{1, 2, 4, 6\}$ can be represent as `01010110`, $\{1, 3, 5\}$ can be represent as `00101010`; `01010110 & 00101010 = 01111110` represent $\{1, 2, 3, 4, 5, 6\} = \{1, 2, 4, 6\} \cup \{1, 3, 5\}$.

- Digit-extension（位扩展）: the C programming language will do it automatically, after the extension, the number **WILL NOT** change.
  - 0-extension（0扩展）: the transformation of *unsigned* numbers, all the extension digit will be filled with 0.
  - signed-extension（带符号扩展）: the transformation of *signed* numbers, all the extension digit will be filled with the sign digit of the original number.

  > [*Example*]
  >
  > `(unsigned short)111...11 = (int)000..00111...11`
  >
  > `(short)011...11 = (int)000...00011...11`
  >
  > `(short)111...11 = (int)111...11111...11`

- Digit-truncation（位截断）: the C programming language will do it automatically. Force to truncate, so the meanings may be different.

  > [*Example*]
  >
  > ```
  > int i = 32768;
  > short j = (int) i;
  > int k = (short) j;
  > ```
  >
  > Both `i` and `j` have code of `0x8000` in the two's complement, which represent the number of `-32768` in *signed short*. When cast to *int* again, the number is still `-32768` according to the digit-extension rules, that is, `k` have a code of `0xFFFF8000` in the two's complement, which is different from original number `32768`.

- Shift-truncation（移位）: the C programming language has the operator `<<` and `>>`.
  - Left-shift（左移）: throw away the high digits, and filled the low digits with 0. Usually, `x << 1` has the same effect as `x * 2`. Left-shift may cause overflow and get the wrong result.

- Right-shift（右移）: throw away the low digits, and filled the high digits with 0 (logic right-shift（逻辑右移）); or throw away the low digits, and filled the high digits with the sign digit of original number (arithmetic right-shift（算术右移）). Just like digit-extension, compiler will automatically choose one of the methods. Similarly, `x >> 1` has the same effect as `x / 2` usually.

  > [*Example*]

  > `x >> y` get the result $\lfloor x/2^y \rfloor$, so if `x` is negative, we may get unexpected result (because the result is not zero-correction（向零取整）), that is, the result is different from `x / (1 << y)`.
  >
  > Let's say `y = 2`, when `x` is negative, the result of `x >> 2` may be different from `x / 4`.

- **Warning**: In the shift operator `x << y` or `x >> y`, if `y` is greater than the digit-length of `x`, then the C compiler / MIPS will automatically do the modulo operation in `y`.

  > [*Example*] If the `x` has 32 digits, the result will be `x >> (y%32)`.

2.4 Logic Operators

Logic Operators `&&`, `||`, `!` only get the result *true* (not 0) or *false* (0).

> [*Example*] Short-circuit evaluation in logic operator: `true || p == 1`, we don't need to check if `p==1`, we can get the result is `true`.

The Comparison between Logic Operators and Bitwise Operators:

- Logic Operators: only has *true* or *false*, don't care about the actual numbers.
- Bitwise Operators: are operations between actual numbers.

2.5 The +/- Operators

Suppose the digit numbers of the operands are w.

Unsigned Addition Operation

Unsigned addition operation is same as addition operation under modulo 2^w, that is,

$$UAdd_w(u, v) = (u + v)\ mod\ 2^w$$

Signed Addition Operation (TC)

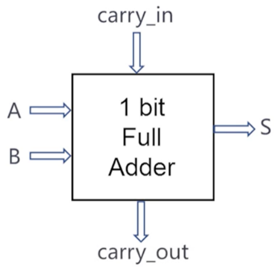
According to the addition formula in two's complement, we still have:

$$TAdd_w(u, v) = (u + v)\ mod\ 2^w$$

> That means, in the following program, `sum1` is equal to `sum2`.
>
> ```c
> int u, v;
> int sum1 = (int)((unsigned int)u + (unsigned int v);
> int sum2 = u + v;
> ```
>
> **Explanation** *The numbers are the same, only the ways we look at the numbers change.*

Serial Carry Adder

So the *addition operation* can be implemented with many **1-bit Full Adder**s（一位全加器）.



where, *A* and *B* are operands, *carry_in* is the carry of lower digits, *carry_out* is the carry of higher digits, *S* is the result of this digit.

We can have:

```
S = A ^ B ^ carry_in
carry_out = (A & B) | (A & carry_in) | (B & carry_in)
```

With many *1-bit full adders* connected together, we get a **serial carry adder** (simple but slow).

Unsigned Subtraction Operation

Unsigned subtraction operation is same as substract operation under modulo 2^w, that is,

$$USub_w(u,v) = u - v = u - v + 2^w = u + (2^w - v) = u + \bar{v} + 1$$

Signed Subtraction Operation (TC)

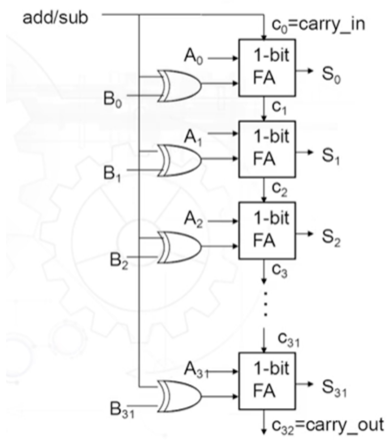
According to the addition formula in two's complement, $[A-B]_{TC} = [A+(-B)]_{TC} = [A]_{TC} + [-B]_{TC}$. And we know that $[B]_{TC} + [-B]_{TC} = 0$, so we have $[-B]_{TC} = 0 - [B]_{TC} = \overline{[B]_{TC}} + 1$.

$$TSub_w(u,v) = u - v = u + \bar{v} + 1$$

Arithmetic/Logic Unit

Thus, we can design a *serial carry adder-subtracter* （串行加减法器） in the following structure.

When doing addition operation, the signal `add/sub` is `0`; when doing subtraction operation, the signal is `1`.



We use a *xor-gate* to implement the bitwise NOT operation in the subtraction.

This unit can also do other logic/arithmetic things, so we call it **ALU** (**Arithmetic/Logic Unit**).

> **Carry Lookahead Adder**
>
> We summarize the carry signals, then we get:

```
c1 = (x0 & c0) | (y0 & c0) | (x0 & y0);
// c2 = (x1 & c1) | (y1 & c1) | (x1 & y1);
c2 = (x1 & x0 & y0) | (x1 & x0 & c0) | (x1 & y0 & c0) |
   (y1 & x0 & y0) | (y1 & y0 & c0) | (y1 & x0 & c0) | (x1 & y1);
// ...
```

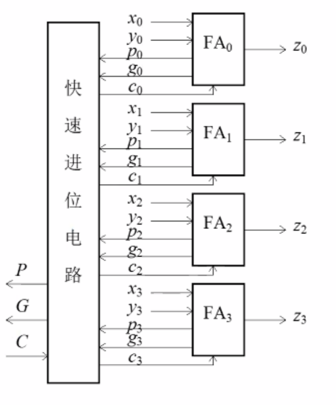
We can use some notations to make the formulas simpler.

$$g_i \triangleq x_i \text{ and } y_i$$
$$p_i \triangleq x_i \text{ or } y_i$$

then we have:

```
c1 = g0 | (p0 & c0);
c2 = g1 | (p1 & g0) | (p1 & p0 & c0);
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c0);
c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
   (p3 & p2 & p1 & p0 & c0);
```

With the formulas above, we can calculate carry of 4 digits in one special unit, which speed up the process of calculation. We call it **Carry Lookahead Adder** (**CLA**).



2.6 Overflow

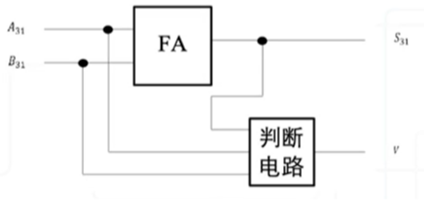
Overflow: the result of operation is out of the range of number.

Two number with the opposite sign in addition operation or two number with the same sign in subtraction operation **CAN NOT** cause *overflow*. Only two number with the same sign in addition operation or two number with the opposite sign in subtraction operation **MAY** cause *overflow*.

Operation	Operand A	Operand B	Overflow Result
C = A + B	$A \geq 0$	$B \geq 0$	$C < 0$
C = A + B	$A < 0$	$B < 0$	$C \geq 0$
C = A - B	$A \geq 0$	$B < 0$	$C < 0$
C = A - B	$A < 0$	$B \geq 0$	$C \geq 0$

How to check if overflow happens?

- Check the sign digit: suppose the highest digit number is 31 ($n = 32$).




```
Overflow_Sign = (A31 & B31 & (~ S31)) | ((~ A31) & (~ B31) & S31);
Overflow_Sign = (A(n-1) & B(n-1) & ~ (S(n-1))) | (~ (A(n-1)) & ~ (B(n-1)) & C(n-1));
```

- Check the carry of the highest digit c_{n-1} and the carry of the second-highest digit c_n.
  - If $c_{n-1} = c_n$, then **NO** overflow.
  - If $c_{n-1} \neq c_n$, then **OVERFLOW**.

  c_{n-1} is the carry_in of $(n-1)$ digit and c_n is the carry_out of $(n-1)$ digit.

  P.S: We count the digit from 0 to $(n-1)$.

  So we can add a *xor-gate* between carry_in and carry_out of the digit $(n-1)$, that is,

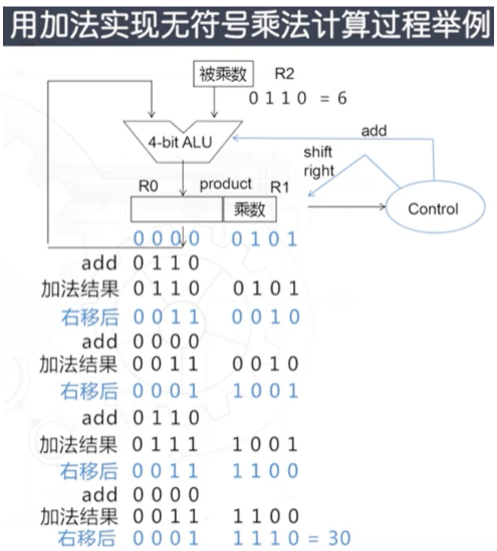
```
Overflow_Sign = carry_in(n-1) ^ carry_out(n-1);
```

- Double sign-digit. Extend the sign digit to 2 digits, and use `00` to represent positive and `11` to represent negative. Then if the result's sign digits are `01` or `10`, overflow happens.

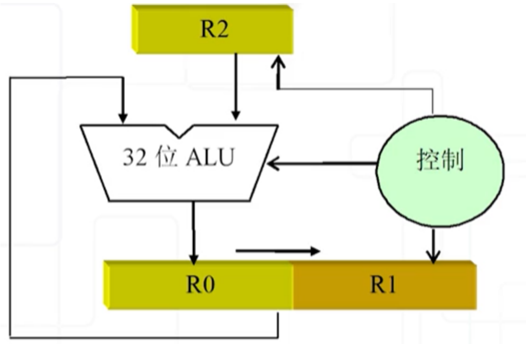
2.7 The * Operator

Multiplication Operation with Signed magnitude

When consider only one digit, we can use an *and-gate* to get the answer simply, so we can use *ALU* and *and-gate* to implement the multiplication operation.

> We can also design a more specific circuit to finish the task.
>
> - If the last digit of R1 is 0, do nothing (R0 plus 0)
> - If the last digit of R1 is 1, then set R0 to R0 plus R2.
>
> After addition operation end, do the *right-shift* to R0 and R1.
>
> **NOTE**: when doing *right-shift*, treat R0 and R1 as a total!
>
> 
>

With the method above, we can design a circuit to do 32-bit multiplication operation of signed magnitude.



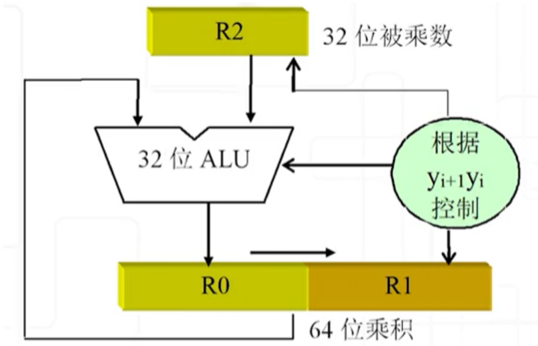
NOTE: the sign digit is **NOT** involved in the calculation above!

Multiplication Operation with Two's Complement: **Booth Algorithm**

With the method above, we use the last two digit of R1 to control the operation:

- If the last two digit of R1 $y_{i+1}y_i$ is 00 or 11, do nothing.
- If the last two digit of R1 $y_{i+1}y_i$ is 01, then set R0 to R0 plus R2.
- If the last two digit of R1 $y_{i+1}y_i$ is 10, then set R0 to R0 minus R2.

We can use the same structure above to complete the multiplication operation.



> **Correctness Proof**
>
> Suppose $[x]_{TC} = x_n x_{n-1} \dots x_1 x_0$, $[y]_{TC} = y_n y_{n-1} \dots y_1 y_0$, then
>
> $$result = (0 - y_0)x \times 2^0 + (y_0 - y_1)x \times 2^1 + (y_1 - y_2)x \times 2^2 + \dots + (y_{30} - y_{31})x \times 2^{31}$$
>
> So,
>
> $$result = x(-y_{31} \times 2^{31} + y_{30} \times 2^{30} + y_{29} \times 2^{29} + \dots + y_0 \times 2^0)$$
>
> y_{31} is the sign digit of y, so whether y is positive or negative, we have:
>
> $$y = (-y_{31} \times 2^{31} + y_{30} \times 2^{30} + y_{29} \times 2^{29} + \dots + y_0 \times 2^0)$$
>
> As a result, $result = xy$, the algorithm is correct!

Suppose the digit numbers of the operands are w.

Unsigned Multiplication Operation

Unsigned multiplication operation is same as multiplication operation under modulo 2^w, that is,

$$UMult_w(u, v) = (u \cdot v) \ mod \ 2^w$$

Signed Multiplication Operation (TC)

We can find that the formula still works:

$$Tmult_w(u,v) = (u \cdot v)\ mod\ 2^w$$

Compiler Optimization in Multiplication Operation

Use `<<` or `>>` and `+` or `-` to optimize:

> [*Example*] The following code can be optimized.
>
> ```
> long mul12(long x) {
>     return x * 12;
> }
> ```
>
> After compiling with optimization, we can get:
>
> ```
> leaq (%rax, %rax, 2), %rax
> salq $2, %rax
> ```
>
> which means:
>
> ```
> t = x + x * 2;
> return (t << 2);
> ```

> [*Example*] When the compiler optimize division operation, a correct term may be added.
>
> We have mentioned before that the result of `x >> y` and `x / (1 << y)` may be different. The compiler will automatically optimize the code `x / (1 << y)` as `(x + (1 << y) - 1) >> y`, that is, $\lfloor (x + (2^y - 1))/2^y \rfloor$, then the two results are the same.

Overflow

The multiplication operation usually **DO NOT** have an overflow-check feature, and the compiler **MAY NOT** check the overflow problem of multiplying.

2.8 Encoding of Floating Number

Floating Number

S	E	M

- Mantissa（尾数, M）: Its integer part is 0, and the highest digit of the decimal part is 1, so the number is basically $0.1???$. However, in **IEEE 754**, the mantissa number is represented as $1.???$. (*normalization*)
- Base（基数, R）: In computer $R = 2$ because data is in binary.
- Exponent（指数, E）: an integer.
- Sign（符号, S）: 0 for positve and 1 for negative.

$$N = (-1)^S \times M \times R^E$$

The *normalization*（规格化） operation makes the representation unique and maximize the significant digits that can be represented in M.

> [*Example*] How to get the code of the floating number?
>
> Example: 13.125 in decimal.

> - Determine the sign S of the number, $S = 1$.
> - Convert the number to binary: $13.125 \rightarrow [1101.001]_2$;
> - Convert the number to the pure decima: $[1101.001]_2 = 0.1101001 \times 2^{[100]_2}$;
> - Normalization. $M = 0.1101001, E = 100$

The range and precision of floating number

- Exponent part has more digits, the range is larger;
- Mantissa part has more digits, the number is more precise;
- If the *word length of machine* is fixed, the larger the range is, the less precise the number is; the more precise the number is, the smaller the range is.
- Floating number can represent limited numbers and limited range.

> [*Example*] Floating number is inaccurate, so don't use `==` when comparing two float numbers, instead, use the following codes.
>
> ```
> float a, b;
> if (abs(a - b) < epsilon) ...
> ```

2.9 IEEE754 Floating Number

IEEE754 Standard

- Single precision: 32 bits (1 + 8 + 23);
  - *float* in C programming language;
  - used when exact precision is less important (e.g. 3D games)
- Double precision: 64 bits (1 + 11 + 52);
  - *double* in C programming language;
  - used for scientific computations.
- Extended precision: 80 bits (1 + 15 + 63/64) (*Intel*).

Single-precision Floating Number (S: 1bit, E: 8 bits, M: 23 bits).

Sign	Exponent	Mantissa	Representation
0/1	255	1...... (not zero)	NaN: Not a Number
0/1	255	0...... (not zero)	sNaN: signal NaN
0	255	0	$+\infty$
1	255	0	$-\infty$
0/1	1~254	M	*normal*: $(-1)^S \times (1.M) \times 2^{E-127}$
0/1	0	M (not zero)	*subnormal*: $(-1)^S \times (0.M) \times 2^{E-126}$
0/1	0	0	$+0/-0$

Normal number (IEEE754, Single-precision): $(-1)^S \times (1.M) \times 2^{E-127}, E \in [1, 254]$.

- Exponent: $e = E - 127 \in [-126, +127]$.
- Mantissa: there is a invisible default 1 in the front of the mantissa (because of *normalization*), that is, $1.M$ is the *normal* number（规格化数）.

> [*Example*] 15213 in decimal.

> - positive, $S = 0$;
> - $15213 = [11101101101101]_2 = [1.1101101101101]_2 \times 2^{13}$;
> - $M = [11011011011010000000000]_2$ (extra zeros);
> - $E = 13 + 127 = 140 = [10001100]_2$;
> - The **IEEE754** code of 15213.0 is
>
> $$0\ 10001100\ 11011011011010000000000$$
>
> [*Example*] 1 10000001 01000000000000000000000 in **IEEE754** code.
>
> - $S = 1$, negative;
> - $E = [10000001]_2 = 129$
> - $[1.M]_2 = [1.01]_2 = 1.25$
> - $f = (-1)^S \times (1.M) \times 2^{E-127} = -5.0$.

The range of IEEE754 Normal Floating Number

- Single precision:
  - $E_{min} = 1, M = 0, f = 1.0 \times 2^{1-127} = 2^{-126}$.
  - $E_{max} = 254, M = 111\dots11, f = 1.111\dots11 \times 2^{254-127} = 2^{127} \times (2 - 2^{-23})$;
- Double precision
  - $E_{min} = 1, M = 0, f = 1.0 \times 2^{1-1023} = 2^{-1022}$;
  - $E_{max} = 2046, M = 111\dots11, f = 1.111\dots11 \times 2^{2046-1023} = 2^{1023} \times (2 - 2^{-52})$;

Subnormal Number（非规格化数）$(-1)^S \times (0.M) \times 2^{-126}$.

The positive subnormal number is smaller than the smallest positive normal number.

Infinity (when $E = 111\dots11, M = 000\dots00$)

Usually overflow result.

> [*Example*] Overflow result.
>
> $$1.0/0.0 = -1.0/-0.0 = +\infty, \quad 1.0/-0.0 = -1.0/0.0 = -\infty$$

Not a Number (when $E = 111\dots1, M = 000\dots00$)

Usually when the number cannot be determined.

> [*Example*] NaN result.
>
> $$\sqrt{-1}, \quad \infty - \infty, \quad \infty \times 0$$

Features of IEEE754 Floating Number

- There is a invisible default 1 in normal number, thus the range of mantissa is larger;
- Provide subnormal number, NaN, ∞ and more complex and various representations;
- Floating 0 has the same code with Integer 0;
- Almost can use the Unsigned Integer Comparer:
  - Normal number v.s. Subnormal number;
  - Normal number v.s. Infinity;
  - Submornal number v.s. Infinity;
  - **Except**:
    - The sign digit should be compared seperately;
    - $-0 = 0$;

- NaN is a special case (because its code is bigger than any other number).

2.10 The Operation of Floating Number

The result of floating operation is not precise.

$$x +_f y = Round(x + y)$$
$$x \times_f y = Round(x \times y)$$

We represent the calculation result as the standard format, if E is too big then it will cause an overflow; if the length of result's mantissa is bigger than the length of M, the mantissa should be rounded（舍入）.

The Addition Operation

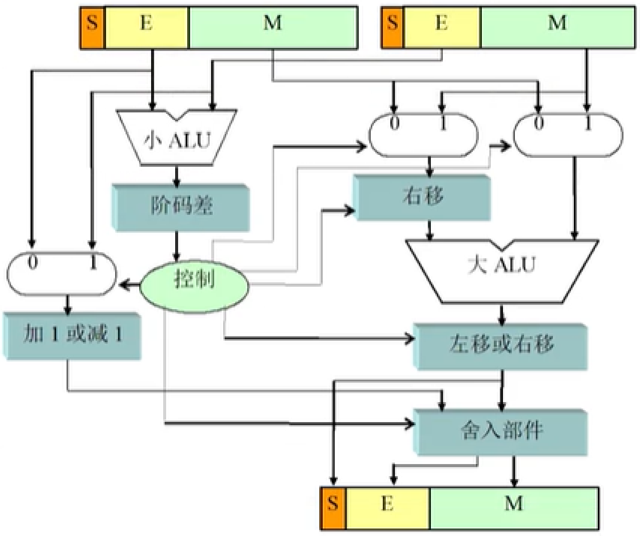
$(-1)^{S_1} \cdot M_1 \cdot 2^{E_1} + (-1)^{S_2} \cdot M_2 \cdot 2^{E_2}$, supposing $E_1 > E_2$ and the result is $(-1)^S \cdot M \cdot 2^E$.

- Align the exponent（对阶）(small aligned to big);
- Add/Subtract the mantissa（尾数加减）；
- normalize（规格化）(left-normalize（左规）, right-normalize（右规）);
- Round-up（舍入）；
- Check overflow（检查溢出）.

> [*Example*] Suppose that (sign: 1 bit, exponent: 5 bits, mantissa: 10bits) *IEEE754-like* code.
>
> $$A = 2.6125 \times 10^1, B = 4.150390625 \times 10^{-1}$$
>
> Calculate $A + B$.
>
> - $A = 2.6125 \times 10^1 = [1.1010001000]_2 \times 2^4$
>
> $B = 4.1503960625 \times 10^{-1} = [1.1010100111]_2 \times 2^{-2}$;
>
> - Align: $B = 0.000001101010\ 0111 \times 2^4$ (left-shift of the decimal point);
> - Add: $1.1010001000 + 0.000001101010\ 0111 = 1.1010100010\ 10\ 0111$;
> - Normalize: $1.1010100010\ 10\ 0111$ is a normal mantissa;
> - Round-up: $1.1010100010\ 10\ 0111 \rightarrow 1.1010100011$ (10 bits);
> - Check overflow: $E = 4$, ok.
> - Output: $[1.1010100011]_2 \times 2^4 = 26.546875$.

> [*Example*] Round-up problems.
>
> ```
> float x, y, z;    // IEEE754 single precision floating number
> x = -1.5e38, y = 1.5e38, z = 1.0;
> if ((x + y) + z != x + (y + z))
>     cout << "Unexpected!";
> ```
>
> The result will be 'Unexpected!'.
>
> $(x + y) + z = 1.0$
>
> $x + (y + z) = 0.0$ (the alignment will lost the mantissa of number z)
>
> So, in IEEE 754 single precision floating number standard, if $|\Delta E| > 24$ then the result is the bigger one (assume the numbers are positive). That means, we don't need to add these two numbers. $24 = 23 + 1$; 23 is the length of the mantissa, 1 is the **invisible default digit** 1.

The circuits of floating adder



The Multiplication Operation

$(-1)^{S_1} \cdot M_1 \cdot 2^{E_1} \times (-1)^{S_2} \cdot M_2 \cdot 2^{E_2}$, supposing the result is $(-1)^S \cdot M \cdot 2^E$.

- Add the exponents（阶码加）；
- Multiply the mantissas（尾数乘）；
- Normalize the mantissa of the result（规格化）：

  The process is just like the code below, where M is the original result's mantissa and E is the result's exponent.

  ```
  while (M >= 2) {
      M >>= 1;
      E ++;
  }
  ```

- Round-up（舍入）(potentially re-normalize)
- Check overflow（检查溢出）.

Round-ups

Methods	1.4	1.6	1.5	2.5	-1.5
Round to 0 (cut-off)	1	1	1	2	-1
Ceiling Round	2	2	2	3	-1
Floor Round	1	1	1	2	-2
Nearest Round	1	2	2	2	-2

Nearst Round (Round-to-even)（最近舍入）：

- The mantissa is not 0.5, same as the normal round-up（四舍五入）；
- The mantissa is 0.5, take the nearest even number;

  > [*Example*] In the nearest round,
  >
  > $$Round(0.5) = 0, Round(1.5) = 2, Round(-0.5) = 0, Round(-1.5) = -2$$

- **In binary**, same:

> [*Example*] Round-to-even in binary (two digits after the point)
>
> - $[10.00011]_2 \to [10.00]_2$ ($011 < 100$, down);
> - $[10.00110]_2 \to [10.01]_2$ ($110 > 100$, up);
> - $[10.11100]_2 \to [11.00]_2$ ($100 = 100$, round-to-even);
> - $[10.10100]_2 \to [10.10]_2$ ($100 = 100$, round-to-even).
>
> **Tips**: In the round-to-even case (case 3, 4), the last digit after rounding must be even (0).

- Reduce the error of the floating operation. (50%-up, 50%-down).

IEEE754 Assumes there is a guard bit（保护位, *G*）, a round bit（舍入位, *R*） and a sticky bit（粘滞位, *S*）in floating number operations, in order to maintain precision.

- *Guard bit*: the 1st bit removed;
- *Round bit*: the 2nd bit removed;
- *Sticky bit*: OR of remaining bits.

Why do we need 3 extra bits?

- For rounding, but 2 extra bits seems to be enough?
- For operation! In subtraction, there may be zero before the floating point, thus we need to *left-shift* to normalize, the guard bit now is within precision! So we need guard bit.

> [*Example*] Let's look at the difference of using 2 extra bits and 3 extra bits when doing subtraction.
>
> Assume $x = 1.000 \times 2^5, y = 1.001 \times 2^1$ and we want to compute $x - y$.
>
> The exact process should be:
>
> ```
>   1.000 0000  x 2^5
> - 0.000 1001  x 2^5
> ------------------------------------------------------------
>   0.111 0111  x 2^5   Need to shift left to normalize
>   1.110 111   x 2^4   Round up, since more than half unit of the last place
>   1.111       x 2^4
> ```
>
> If we use only 2 extra bits, we get
>
> ```
>   1.000 00  x 2^5
> - 0.000 11  x 2^5   Round is 1, Sticky is 1
> ------------------------------------------------------------
>   0.111 01  x 2^5   Need to shift left to normalize, must use Round bit
>   1.110 1   x 2^4   Can't round using Sticky, since can't tell if >/=/< 1/2 ULP
> ```
>
> which is not correct.
>
> But if we use 3 extra bits, we get the correct answer
>
> ```
>   1.000 000  x 2^5
> - 0.000 101  x 2^5   Guard is 1, Round 0, and Sticky is 1
> ------------------------------------------------------------
>   0.111 011  x 2^5   Need to shift left to normalize, using Guard bit
>   1.110 11   x 2^4   Round up, since more than half unit of the last place
>   1.111      x 2^4   Result is correctly rounded
> ```

The Latency of Floating Number Operations:

- Add/Sub/Multiply: slower than integer operations.
- Divide: faster than integer operations.

Casting/Conversions between floating number and integer

- *double*/*float* to *int*
  - Truncates fractional part;

- Like rounding toward zero;
- Not defined when out of range or NaN: generally sets to the minimum.

- *int* to *double*: *Exact conversion*, as long as the size of *int* doesn't exceed 53 bits.
- *int* to *float*: will round according to rounding mode.

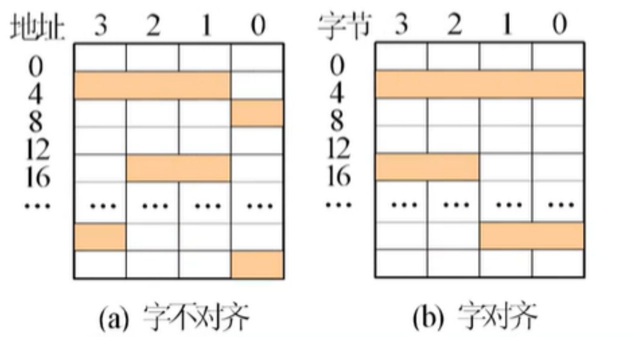
2.11 Storage Format of Data in Memory

Byte Order (Endianness)（字节顺序）

- The byte order in the memory that the data (*more than 1 byte*) is stored.
- Big Endian（大数端）: The lowest byte stores in the biggest address.
- Little Endian（小数端）: The lowest byte stores in the smallest address.

> [*Example*] The byte order of the data `0x000F4240`
>
> - Big Endian: `00` `0F` `42` `40` (address: small → big, normal reading order)
> - Little Endian: `40` `42` `0F` `00` (address: small → big)

Alignment（对齐方式）



- *Unaligned* data storage:
  - Save the storage space;
  - **Low access speed** (the data may be cut off from the middle, need to access the storage twice to get the full data);
  - The interface is complexer.
- *Aligned* data storage:
  - May cause the waste of storage space;
  - High access speed;
  - The interface is simpler.
- **Principles**:
  - char: one-byte aligned;
  - short: two-bytes aligned;
  - int/float: four-bytes aligned;
  - double: eight-bytes aligned.
  - *(long double: ten bytes.)*
  - *Tips*: x-bytes aligned means data's address in the memory must be the multiple of x.

> [*Example*] Change the definition of the data structure to save space.

```
struct loose {
    short s;     // 16 bits
    int i;       // 32 bits
    char c;      // 8 bits
    double p;    // 64 bits
}
```


```
struct tight {
    double p;    // 64 bits
    int i;       // 32 bits
    short s;     // 16 bits
    char c;      // 8 bits
}
```