计算机内部的数据表示

计算机可以处理多种不同类型的数据。这些数据在计算机内部可能有不同的表示方法。

数值

将一个定长二进制数从高位到低位列写如下：`00001111`。

其的逻辑最高位称为 MSB（most significant bit），最低位称为 LSB（least significant bit）。

整数

无符号数是容易处理的。怎么表示负数？最简单的方法是**符号-数值法**，直接引入一个符号位。符号位为 0 时数为正，否则为负。但这样很麻烦，而且 0 会有两个（正 0 负 0）。因此现在不用这种方式了。

二进制补码法用数的二进制补码来表示数。

- 它把最高位作为符号位，最高位为 1 时是负数，最高位为 0 时是非负数。
- 正数的补码是其自身。
- 负数 x 的补码等于其 -x 的反码再 + 1。例如 `-1` 在 8 位下，`1` 的反码是 `11111110`，再 + 1 就得到 `-1` 的补码 `11111111`。

> 补码的英文名为 Two's complement。还有另外一种表示符号的方式为 Ones' complement，即补码直接等于反码。这会导致 0 有两个表示：全 0 和全 1。

某种意义上这么规定是合理的——利用自然溢出。如 1 + (-1) 自然溢出后就是 0。

对于 32 位有符号数，其最高位为 1 时表示的数是自身的无符号版本 - 2^{32}。这可以通过：w 位有符号数的最高位的权重是 -2^{w-1}，而 w 位无符号数的最高位的权重是 2^{w-1}，来理解。

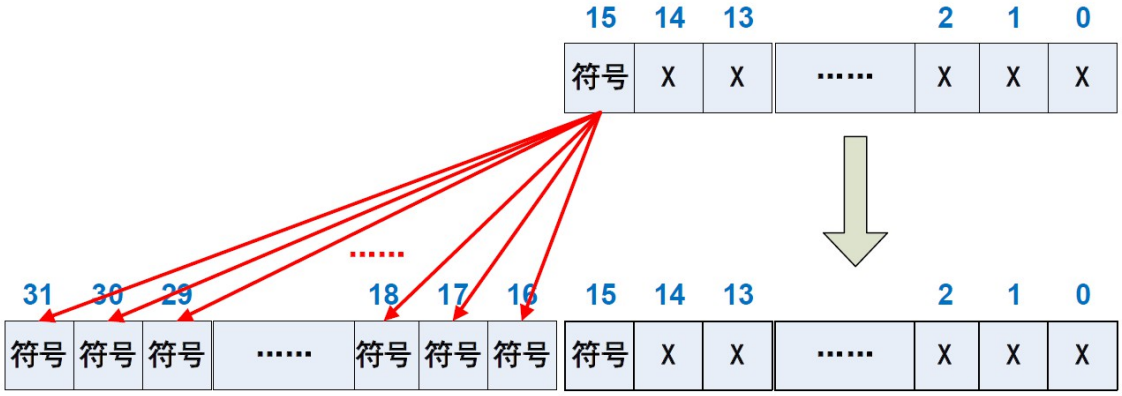
怎么计算溢出后的数呢？一般来说，溢出后的位表示就是原结果模 2^{32}。然后按照给定的解释方式解释这个结果即可。例如如果两个 32 位有符号正数 x 和 y 相加溢出了，那么最后得到的位表示按照 32 位有符号数解释就是 x + y - 2^{32}。

类型转换也是如此。强制类型转换并不会改变位表示，只是会改变位表示的解释方式。

引入了符号位需要注意：数之间比大小需要知道两点，**有无符号**和**位数**。例如 11001100 和 01001100，如果两个都是 8 位有符号数，那么前者小于后者。如果是 16 位数，那么前者大于后者。

符号扩展

如果要将一个位数小的数扩展为位数大的数，可能需要复制符号位以使得数代表的值不变。**有符号扩展（Sign Extension Shortcur）**会用 MSB（即符号位）填补高位，如图。



无符号扩展（或者称为**零扩展，Zero Extension**）则用 0 填充。

整数运算

一般来说无符号数的运算是比较平凡的，因此这里主要讨论有符号数的运算。而运算本身也是不困难的，我们着重讨论两个问题。

- 溢出如何处理。
- 运算如何加速。

什么情况下会发生溢出？对于加减法来说，如果加数异号、减数同号，那么必然不会溢出。否则就可能溢出。

在 C 标准中，有符号数的溢出是 undefined behavior。有可能在溢出的时候要报出异常，有可能不要。而有的程序语言标准规定在溢出时必须报出异常。为了应对两种情况，MIPS 使用了两套指令。

- 不带 `u` 的，如 `add`，`sub`，结果溢出时产生异常。
- 带 `u` 的，如 `addu`，`subu`，结果溢出时不产生异常。

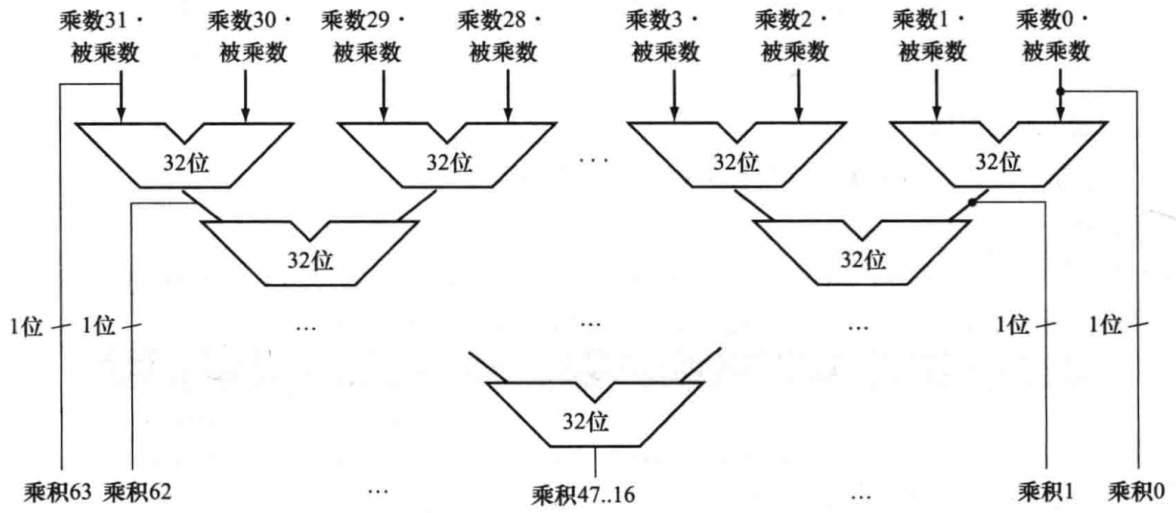
> 因此带 `u` 不意味着运算数是无符号的。例如 `addiu`，其立即数仍需要做符号扩展再参与运算。

溢出产生异常时，产生溢出的指令地址保存在 EPC（Exception Program Counter，异常程序计数器）中，而后计算机会跳到一个预先设定好的地址去执行相应的异常处理程序（Exception Handler）。`mfc0` 指令可将 EPC 存入一个通用寄存器。有时处理完异常可以利用这条指令跳转回原位置。

> 会发现一个问题：虽然异常时寄存器的值会被保存，但如果要返回原位置，恢复后会有一个通用寄存器的值被改变（用作 `jr`）。这可能会带来一些问题。MIPS 设定 k0，k1 两个寄存器在异常时不会恢复，因此可以将 EPC 中的值放入其中一个。

有的处理器使用**饱和运算（Saturating）**处理溢出。当结果比最大的数大（比最小的数小）时，结果变为最大的数（最小的数）。

运算加速一般可以使用类似于流水线的技术。一个可流水线化的快速乘法器如图所示。



浮点数

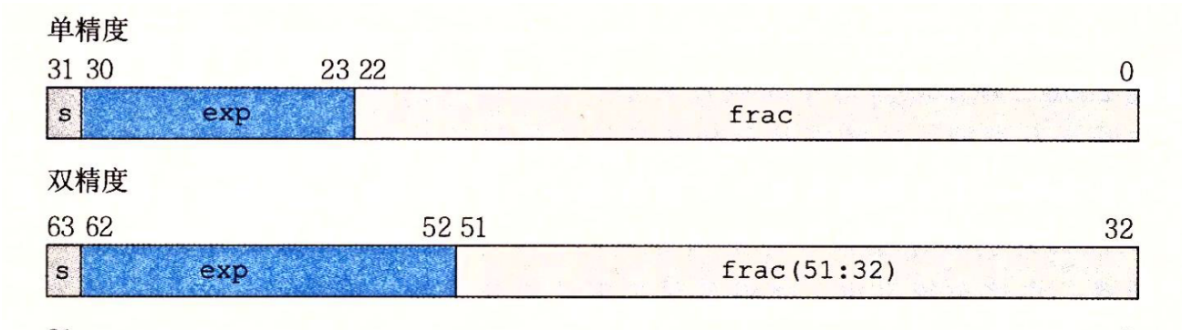
一般使用 IEEE 754 标准，即 IEEE 浮点数标准来表示一个浮点数。

浮点数表示分成三个部分：

1. 符号位。一般记为 s。和整数一样，符号位为 1 表示负数，否则为非负数。
2. **尾数（Significand）**。一般记为 M。
3. **阶码（Exponent）**。一般记为 E。

这三个部分组合成的浮点数就是 $V = (-1)^S \times M \times 2^E$。

IEEE 浮点数标准为这三个部分分配了三个位域，如图所示。



解释：

1. 单精度浮点数为 32 位，其最高位为符号位，紧跟着是 8 位 exp，和阶码有关，最后是 23 位 frac，和尾数有关。
2. 双精度浮点数为 64 位，其最高位为符号位，紧跟着是 11 位 exp，和阶码有关，最后是 52 位 frac，和尾数有关。

上述表示和浮点数值的联系：

1. M = 1.frac，即 M 的二进制表示中，小数点前有一个 1（可以被称为 implied leading 1，因为其没有显式被写出），小数点后的部分就是 frac。
2. E = exp - B，B 为**偏差（Bias）**，其值为 exp 可以表示的最大值除以 2（向下取整）。对单精度，B = 127。对双精度，B = 1023。

对于**规格化的（Normalized）**浮点数，exp 不能全为 0 或者全为 1，这两种情况被保留。

- exp 全为 0：表示**非规格化的（Denormalized）**浮点数，这时 E = 1 - B，M = frac，没有开头的 1。
    - 它给出了 0 的表示：exp 和 frac 全为 0 就是 0。显然，这导致有两种 0：+0 和 -0。

    - 它可以用来表示非常接近于 0 的数。
- exp 全为 1：表示无穷大或者 NaN（Not a Number）。
    - frac 全为 0 时：s = 0 表示正无穷，s = 1 表示负无穷。
    - 否则表示 NaN。

将一些重要的*非负*浮点数以表格方式列出，如下。

描　述	exp	frac	单精度		双精度	
			值	十进制	值	十进制
0	00 ··· 00	0 ··· 00	0	0.0	0	0.0
最小非规格化数	00 ··· 00	0 ··· 01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非规格化数	00 ··· 00	1 ··· 11	$(1-\varepsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\varepsilon) \times 2^{-1022}$	2.2×10^{-308}
最小规格化数	00 ··· 01	0 ··· 00	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
1	01 ··· 11	0 ··· 00	1×2^{0}	1.0	1×2^{0}	1.0
最大规格化数	11 ··· 10	1 ··· 11	$(2-\varepsilon) \times 2^{127}$	3.4×10^{38}	$(2-\varepsilon) \times 2^{1023}$	1.8×10^{308}

舍入规则

显然有限的位宽不可能表示无限精度的数。因此，对于实数 x，需要找到一个和它尽可能接近的数 y，且 y 可以用这种浮点格式无误差地表示。

IEEE 浮点格式定义了四种不同的舍入方式。

- 向零舍入。即保证 |y| <= |x| 且舍入误差最小的舍入。
- 向下舍入。即 y = floor(x)。
- 向上舍入。即 y = ceil(x)。
- **向偶数舍入（round-to-even）**。也称为**向最接近的值舍入（round-to-nearest）**。它是用于寻找近似表示的**默认方式**。
    - 如果 x 向下舍入比向上舍入产生误差更少，那么向下舍入。
    - 如果 x 向上舍入比向下舍入产生误差更少，那么向上舍入。
    - 否则，向上或者向下舍入，使得**需要保留的最低有效位为偶数**。特别地，对于二进制小数而言，“偶数”就是 0。

举例如下。

方　式	1.40	1.60	1.50	2.50	−1.50
向偶数舍入	1	2	2	2	−2
向零舍入	1	1	1	2	−1
向下舍入	1	1	1	2	−2
向上舍入	2	2	2	3	−1

向偶数舍入的一个优点在于对于大量的随机数据，向偶数舍入使得这些数据精确的平均值和全部做舍入之后的平均值基本相同。

浮点运算

IEEE 标准中，对 x 和 y 两个浮点数做某个**精确的**运算 Opr，得到的结果是 Opr x y 舍入后的值。

还有一些额外的规则。

- `1/-0` 是负无穷，`1/+0` 是正无穷。
- 正无穷减负无穷等于 NaN。
- 大多数运算中 NaN 参与运算则结果为 NaN。

由于精度的原因，很多时候浮点运算顺序会影响结果。例如单精度下 `3.14+(1e10-1e10)` 和 `(3.14+1e10)-1e10`。两者在数学上相同，但由于前面的运算原则，`3.14+1e10` 舍入后还是 `1e10`，从而前者得到 `3.14`，后者得到 `0`。

由此可以推知，这意味着**浮点加法的结合律、浮点乘法的结合律和分配律并不保持。**

但很多其他性质是保持的，例如交换律，还有单调性。单调性如 a >= b，则对任意 x 有 x + a >= x + b。这是因为浮点运算不会出现整数那样取模导致大小关系改变的问题。

C/C++ 中的浮点数

使用 C 中的 union 类型以及位段特性能很方便地让我们看到浮点数中每一个位域的情况。下面这段示例程序就打印了 6.25 以单精度浮点数存储时的二进制表示。

```c
#include <stdio.h>
typedef union{
    float a;
    struct {
        // go from LSB to MSB
        unsigned int frac : 23;
        unsigned int exp : 8;
        unsigned int signbit : 1;
    } bit_fields;
} my_float;
void printb(unsigned int x, int len){
    int i, j;
    for (i = 0, j = (1 << len >> 1); i < len; ++i, j >>= 1)
        printf("%d", (j & x) ? 1: 0);
}
int main(){
    my_float f;
    f.a = 6.25;
    printb(f.bit_fields.signbit, 1);
    printb(f.bit_fields.exp, 8);
    printb(f.bit_fields.frac, 23);
    return 0;
}
```

类似的可以对 `double` 进行处理。

注意，并不是任何时候，浮点数都是符合 IEEE 754 标准的。C 标准中并不强制规定使用 IEEE 754 标准。在 C++ 中，标准库提供了一个函数 `std::numeric_limits<T>::is_iec559()` 用于测试当前使用的类型 `T` 是否符合 IEEE 754 标准。有意义的 `T` 包括 `float`，`double`，`long double` 等。

有可能有的程序语言会在运算浮点数时，对浮点数采用某种不同的中间表示。

整数和浮点数转换

一般而言（下面假定 `int` 为 32 位）：

- `int` -> `float`：不会溢出，但可能发生舍入。
- `double` -> `float`：可能溢出成无穷，还可能发生舍入。
- `int` / `float` -> `double`：保留相应精度。
- `float` / `double` -> `int`：向零舍入。也有可能溢出，但对溢出的处理是 undefined behavior。

致谢

本文的主要内容参考自上海交通大学《计算机组成》（课程代号：EI332）一课的课程材料，以及《Computer Organization and Design: Hardware/Software Interface (5th edition)》、《深入理解计算机系统（第 3 版）》。

本文的图片都来自《深入理解计算机系统（第 3 版）》。