在计算机的存储器中统一采用二进制数的方式进行数据存储。 而编程中则综合使用二进制、 八进制、 十进制与十六进制的数据表示方法， 程序编译后一般生成十六进制的可烧写文件， 而烧写到存储器后最终在存储单元中存放的还是二进制形式。而二进制又有真值， 原码，反码，补码，机器数，有符号数，无符号数，等诸多概念之分。 故下面主要就二进制数 （以整数为例， 后面提到的数据皆指整数）的存储与运算过程中涉及到的一些概念与规则进行梳理。 讲的主要是在计算机中的， 以 8 位单片机为例 ，后面以 32 位 ARM 单片机指令举例。

1 有符号数和无符号数。

数据首先分为有符号数和无符号数。

对于无符号数来说， 肯定指的 0 与正数，无负数之说，自然也无原码、反码、补码之说， 一般也不针对于无符号数讨论机器数、 真值等概念。 其存储方式与有符号数存储也无特别之处，具体的将在“存储单元中的数据”一节讲述。

有符号数，有正负与 0 三种，由于在计算机中无法表示负号，或者说用专门的符号表示负号很不方便，于是就采用对正负号进行数值编码的方式，用“ 0”代表非负数，“1”代表负数。根据不同的编码方式， 对有符号数一般可以有原码、反码、补码三种最常见的编码形式。

2 真值

真值就是所表示的数的大小，一般用 10 进制表征。

3 原码、反码、补码

具体概念我就不重复了，只重申下相关结论：

a.正数的原码、反码、补码都相同。

b.负数的反码为原码的按位取反（保持符号位不变） ，补码为反码加 1.

4 机器数

原码、反码、补码都是机器数的一种表示形式，或说都属于机器数。

5 存储单元中的数据（存储单元包括存储器中的存储单元和寄存器）

在计算机的存储器的存储单元中的数据均以补码形式存放的， 于是在计算机中的数据 表示有下面结论：

a 不使用原码与反码。但原码与反码可以作为计算真值的中间媒介。

b 存储单元中的数据以补码形式存在。

c 数据的存取与运算都以补码形式进行。

d 补码就是机器数，机器数就是补码。

解释：

掌握一个基本原则——简单，

存储单元是个很有原则的家伙，他只管存 01 序列，才不管该序列是表示指令编码还是数据呢，更不会管是有符号数还是无符号数，也不管是数据的原码、反码还是补码。 只是人们考虑到 0 只有在补码中的表示才是唯一的， 故规定数据以补码形式在计算机中存储。这就是 数据存在的简单原则 。

要知道，数据在计算机内部的存在形式以及转移、运算过程中，都是一串高低电平组合，如果要在数据转移、 运算过程中还要考虑补码、 原码、反码的转换，

将大大增加硬件复杂度， 而这种转换本身也是没有意义的。 所以如果你已知某存储单元中的内容是

1	0	0	0	0	0	0	1

如果要对其进行运算，不需进行任何转化，直接拿出来就行，例如对其加 1：

1000 0001 + 0000 0001 = 1000 0010

得到的结果是 1000 0010，要对其进行存储的话，也是不需进行任何转化，直接对号入座：

1	0	0	0	0	0	1	0

这就是 数据存取的简单原则 。

数据的运算，按常规的二进制运算规则进行，只需保证结果只取后 8 位。加满就进位，不够减就借位， 根据最高位的状态是不是负数以及是否发生了进位与借位。这就是 数据运算的简单原则 。

6 数据的运算以及对状态标志位 NZCV(以 ARM 为例)的影响。

就是普通的二进制运算规则。结果为 0 时， Z = 1; 结果最高位为 1 时 N = 1; 运算过程中最高位向前面的一位（可以认为该位就是 C）进了 1，则 C = 1; 运算过程中最高位从前面的一位（可以认为该位就是 C）借了 1，则 C = 0; 运算中最高位只有进入（ carryin）或进出（ carryout），或者只有借入（ borrowin ）或借出（borrowout ），则 V = 1。

如对下面两个存储单元中的数据相加：

1	1	0	0	0	1	1	1
1	1	1	1	1	1	0	0

什么都不用管，直接取出来相加，结果为：

1

1	1	0	0	0	0	1	1

很容易得到 Z = 0; N = 1; C = 1; V = 0;

7 从指令到存储单元

可能有些同学对从程序中的数据到存储单元（包括寄存器）中的数据怎样对应还有疑问。以 ARM 指令为例，如：

MOV R0, #0XFFFFFFFF;

简单，直接转化成二进制，即 32 个 1，存到存储单元中。

MOV R0, #0B1001 1100 0011…0001;

简单，不用做任何转化，直接存到存储单元中。

MOV R0, #35;或 MOV R0, # -35;

简单，直接转化为二进制补码，存到存储单元中。

其实，这些转化都是编译器在编译的过程中进行的，也就是“纯软件”操作。一旦编译完成， 把数据存到存储单元中， 后面的就依据前面的提到的 “三个简单原则”进行了。

8 从存储单元到指令

那么把存储单元中的数，或运算后的数取出来，其真值应该是多少呢？也简单，直接按补码的真值计算方法计算即可。 有没有可能是无符号数呢？即使你用unsigned in a;定义了一个无符号数，也可以认为是按补码形式存放的，因为如果你这样赋值：

a = -1;

如果你使用的编译器不报错的话，他也自动给你存的是 -1 的补码形式。
如果你这样赋值：

a = 65536*65536 - 1;

（即二进制的 32 个 1）

在存储单元中存的是 32 个 1；而给 int b = 65536*65536 - 1;赋值，对应存储单元中依然存的是 32 个 1.

内容都一样，又怎么区别那个是无符号数那个是有符号数呢。其实在 C语言中就看你把它给什么类型的变量， 根据变量类型的不同， 用不同翻译方法求取出的数据的真值即可。 而你怎么翻译， 都跟计算机没关， 他内部处理数据仍然只是按他自己的那几条规则。

9 ARM 中的条件指令

如果运算时不进行有符号数、 无符号数间的转化， 怎么确定 CC(无符号数小于)，LT(带符号小于) 等条件呢？其实二者并矛盾，因为 CC的判断依据是标志位 C = 0; 而 LT的判断依据是标志位 N≠V。其他的也都是根据状态标志位来判断的。而前面我们已经讨论了，确定条件标志位 NZCV的状态时，只需按“数据运算的简单原则”进行，而不必管什么码啊，什么符号的。

当然也可以直接按字面意思来判断， 即判断 LT时，把后面的操作数都当做带符号数的补码，计算出对应的真值，就可进行判断是否“小于” ；判断 CC时，把后面的操作数都当做无符号数， 即直接用二进制到十进制的转化规则 （就是指数法嘛）来计算真值，然后就可以确定是否“小于”了。通过对不同情况的考察会发现，两种方法是等效的。

10 ARM 中字节到半字到字的扩展

在位数的扩展中，就要考虑有符号数和无符号数的区别了。因为对于一个字节数 1000 0001，扩展到一个字，即 32 位，扩展的原则是保证前后真值不变。按有符号数扩展要在前面补 24 个 1，按无符号扩展，则要补 24 个 0.当然如果对于正数进行扩展，一律在前面补若干个 0 即可。

在这里对无符号数的特别关照， 并不与前面的对无符号数的 “忽略”相矛盾，因为把他当做有符号数还是无符号数，全凭用户（或程序员）决定，与计算机内部处理的“简单原则”无关。如何决定？ ARM 提供了专门的指令，如 LDRB,LDRSB,STRH,STRSH,可见要想让计算机做额外的工作就必须付出额外的代价（指令的增多）。

而且可以说，只要没有标志性的无符号数处理指令，计算机的统一处理数据格式就是补码，或更根本的，就是高低电平序列。

11 总结

站在计算机的角度看待数据，以最简单的规则办事，就会看到有序的工作流

程了。至于“直观”，就要用户付出一些精力来达到了。