# 有符号数与无符号数的探讨

这个问题，要是简单的理解，是很容易的，不过要是考虑的深了，还真有些东西呢。
下面我就把这个东西尽量的扩展一点，深入一点和大家说说。

一、只有一个标准！

在 汇编语言层面，声明变量的时候，没有 signed 和 unsignde 之分，汇编器统统，将你输入的整数字面量当作有符号数处理成补码存入到计算机中，只有这一个标准！汇编器不会区分有符号还是无符号然后用两个标准来处理， 它统统当作有符号的！并且统统汇编成补码！也就是说，db -20 汇编后为：EC ，而 db 236 汇编后也为 EC 。这里有一个小问题，思考深入的朋友会发现，db 是分配一个字节，那么一个字节能表示的有符号整数范围是：-128 ~ + 127 ，那么 db 236 超过了这一范围，怎么可以？是的，+236 的补码的确超出了一个字节的表示范围，那么拿两个字节（当然更多的字节更好了）是可以装下的，应为：00 EC，也就是说 +236 的补码应该是 00 EC，一个字节装不下，但是，别忘了 “截断”这个概念，就是说最后汇编的结果被截断了，00 EC 是两个字节，被截断成 EC ，所以，这是个 “美丽的错误”，为什么这么说？因为，当你把 236 当作无符号数时，它汇编后的结果正好也是 EC ，这下皆大欢喜了，虽然汇编器只用一个标准来处理，但是借用了 “截断”这个美丽的错误后，得到的结果是符合两个标准的！也就是说，给你一个字节，你想输入 有符号的数，比如 -20 那么汇编后的结果是符合有符号数的；如果你输入 236 那么你肯定当作无符号数来处理了（因为236 不在一个字节能表示的有符号数的范围内啊），得到的结果是符合无符号数的。于是给大家一个错觉：汇编器有两套 标准，会区分有符号和无符号，然后分别汇编。其实，你们被骗了。:-）

二、存在两套指令！

第一点说明汇编器只用 一个方法把整数字面量汇编成真正的机器数。但并不是说计算机不区分有符号数和无符号数，相反，计算机对有符号和无符号数区分的十分清晰，因为计算机进行某 些同样功能的处理时有两套指令作为后备，这就是分别为有符号和无符号数准备的。但是，这里要强调一点，一个数到底是有符号数还是无符号数，计算机并不知 道，这是由你来决定的，当你认为你要处理的数是有符号的，那么你就用那一套处理有符号数的指令，当你认为你要处理的数是无符号的，那就用处理无符号数的那 一套指令。加减法只有一套指令，因为这一套指令同时适用于有符号和无符号。下面这些指令：mul div movzx … 是处理无符号数的，而这些：imul idiv

movsx … 是处理有符号的。
举例来说：
内存里有 一个字节 x 为：0x EC ，一个字节 y 为：0x 02 。当把 x，y 当作有符号数来看时，x = -20 ，y = +2 。当作无符号数看时，x = 236 ，y = 2 。下面进行加运算，用 add 指令，得到的结果为：0x EE ，那么这个 0x EE 当作有符号数就是：-18 ，无符号数就是 238 。所以，add 一个指令可以适用有符号和无符号两种情况。(呵呵，其实为什么要补码啊，就是为了这个呗，:-))
乘法运算就不行了，必须用两套指令，有符号的情况下用 imul 得到的结果是：0x FF D8 就是 -40 。无符号的情况下用 mul ，得到：0x 01 D8 就是 472 。(参看文后附录 2 例程)

三、可爱又可怕的 c 语言。

为 什么又扯到 c 了？因为大多数遇到有符号还是无符号问题的朋友，都是 c 里面的 signed 和 unsigned 声明引起的，那为什么开头是从汇编讲起呢？因为我们现在用的 c 编译器，无论 gcc 也好，vc6 的 cl 也好，都是将 c 语言代码编译成汇编语言代码，然后再用汇编器汇编成机器码的。搞清楚了汇编，就相当于从根本上明白了 c，而且，用机器的思维去考虑问题，必 须用汇编。(我一般遇到什么奇怪的 c 语言的问题都是把它编译成汇编来看。)

C 是可爱的，因为 c 符合 kiss 原则，对机器的抽象程度刚刚好，让我们即提高了思维层面（比汇编的机器层面人性化多了），又不至于离机器太远（像 c# ， java 之类就太远了）。当初 K&R 版的 c 就是高级一点的汇编……:-)

C 又是可怕的，因为它把机器层面的所有的东西都反应了出来，像这个有没有符号的问题就是一例（java 就不存在这个问题，因为它被设计成所有的整数都是有符号的)。为了说明它的可怕特举一例：

```
#include<stdio.h>
#include<string.h>

int main()
{
  int x = 2;
  char*str= "abcd";
  int y = (x - strlen(str))/2;

  printf("%d\n",y);
}
```

结果应该是 -1 但是却得到：2147483647 。为什么?因为 strlen 的返回值，类型是 size_t，也就是 unsigned int ，与 int 混合计算时有符号类型被自动转换成了无符号类型，结果自然出乎意料。。。
观察编译后的代码，除法指令为 div ，意味无符号除法。
解决办法就是强制转换，变成 int y = (int)(x - strlen(str))/2; 强制向有符号方向转换（编译器默认正好相反），这样一来，除法指令编译成 idiv 了。
我 们知道，就是同样状态的两个内存单位，用有符号处理指令 im ul ，idiv 等得到的结果，与用 无符号处理指令 m ul，div 等得到的结果，是截然不同的！所以牵扯到有符号无符号计算的问题，特别是存在讨厌的自动转换时，要倍加小心！（这里自动转换 时，无论 gcc 还是 cl都不提示!!!）

为了避免这些错误，建议，凡是在运算的时候，确保你的变量都是 signed 的。

四、c 的做法。

对 于有符号和无符号的处理上，c 语言层面做的更 “人性化”一些。比如在声明变量的时候，c 有 signed 和 unsigned 前缀来区别，而汇编呢，没有任何区别，把握全在你自己，比如：你想在一个字节中输入一个有符号数，那么这个数就别超过 -128 ~ +127 ，想输入无符号数，要保证数值在 0~255 之间。如果你输入了 236 ，你还要说你输入的是有符号数，那么你肯定错了，因为有符号数 236 至少要两个字节来存放（为 00 EC），不要小看了那一个字节的 00，在有符号乘法下，两个字节的 00 EC 与 一个字节的 EC，在与同样一个数相乘时，得到的结果是截然不同的!!!

我们来看下具体的列子（用 vc6 的 cl编译器生成）：

C 语言 编译后生产的汇编语言

```
……
char x;
unsigned char y;
int z;

x = 3;
y = 236;

z = x*y;
…… ……
```

```
_x$ = -4
_y$ = -8
_z$ = -12
……
mov BYTE PTR _x$[ebp], 3
mov BYTE PTR _y$[ebp], 236

movsx eax, BYTE PTR _x$[ebp]
mov ecx, DWORD PTR _y$[ebp]
and ecx, 255

imul eax, ecx
mov DWORD PTR _z$[ebp], eax
……
```

我 们看到，在赋值的时候（绿色部分），汇编后与本文第一条论述相同，是否有符号把握全在自己，c 比汇编做的更好这一点没有得到体现，这也可以理解，因为 c 最 终要被编译成汇编，汇编没有在变量声明时区分有无符号这一功能，自然，c 也没有办法。但既然 c 提供了 signed 和 unsigned 声明，汇编后，肯定有 代码体现这一点，表格里的红色部分就是。对有符号数 x 他进行了符号扩展，对无符号 y 进行了零扩展。这里为了举例的方便，进行了有符号数和无符号数的混合运 算，实际编程中要避免这种情况。

（完）

附录：

1．计算机对有符号整数的表示只 采取一套编码方式，不存在正数用原码，负数用补码这用两套编码之说，大多数计算机内部的有符号整数都是用补码，就是说无论正负，这个计算机内部只用补码来 编码！！！只不过正数和 0 的补码跟他原码在形式上相同，负数的补码在形式上与其绝对值的原码取反加一相同。

2．两套乘法指令结果例程：

;; 程序存储为 x.s

```
extern printf
```

```
global main

section .data
  str1: db "%x",0x0d,0x0a,0
  n: db 0x02
section .text
main:
  xor eax,eax
  mov al, 0xec
  mulbyte [n];有符号乘法指令为:imul

  push eax
  push str1
  call printf

  add esp,byte 4
  ret
```

编译步骤:
1. nasm -felf x.s
2. gcc x.o

ubuntu7.04 下用 nasm 和 gcc 编译通过。结果符合文章所述