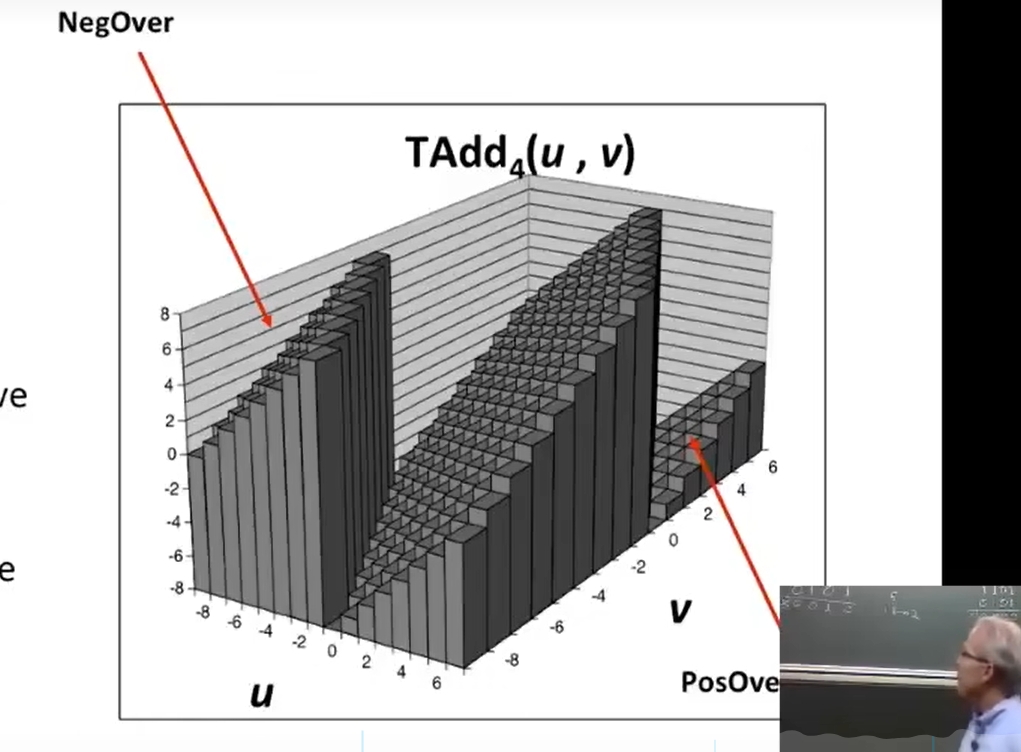


在底层中，程序溢出情况如图



此图绝美，完美的阐述了溢出的情况

总和太小，它会变正。总和太大，他会变负，物极必反了属于是

程序底层对其的截断是为了不一遍遍的扩展字长，这样会导致内存的快速消失，故这里使用截断来处理

对于无符号数，其的溢出是可以用取模来计算的，但也就只有无符号数可以了

5:0101

5\*5=25：0001 1001

对25截断为后四位，结果为9 而9=25%2^4

这是无符号数的情况，接下来补码的情况将更加充满魅力

例如，对于0001 1001截断后的11001进行补码的解释，它的结果会是-7

5的原码是10101，有5位，故这里截断也应该有5位

“当你希望做乘法时，编译器可能解释为移位，”

位移操作是可以视作乘以一个2的幂数次的，而我们有可以通过lea等汇编指令来实现其他次数的叠加，这里的具体操作取决于编译器的优化，而这样优化的原因通常是因为乘法的逻辑电路通常更加耗时

位移操作只需要一个时钟周期，而在现在的计算机中，乘法操作通常需要3个时钟周期的时间

右移除以2的幂数次基本与左移的逻辑同理

但是当右移数为负数时会存在一些问题

就比如对于-6进行右移操作是-6:1010

对它进行右移一位除以2^1得1101~-3，此时正好满足

再对其进行右移呢 1110~-2，此时就不符合逻辑了，但是这其实是一种舍入逻辑，

即向无穷小舍入

而要想改变这种舍入，需要添加一个**偏移量**

这里的偏移量应该为1，比如当-3再右移时，其会先对1101+1得到1110

此时再右移得到1111，此时就是-1的补码，此时舍入方式就发生了改变

unsigned i;

for(i=cnt-1;i<cnt;i--)

a[i]+=a[i+1];

在c中size\_t一般是有unsigned long转换来的

size\_t i;

for(i=cnt-1;i<cnt;i--)

a[i]+=a[i+1];

大端序和小端序在这里直接给出

\*\*大端序（Big-Endian）\*\*和\*\*小端序（Little-Endian）\*\*是两种数据在内存中存储的顺序方式。这两种字节序在处理多字节数据时决定了字节的排列顺序。

### 大端序（Big-Endian）

在大端序中，数据的高位字节存储在内存的低地址处，而低位字节存储在高地址处。这意味着，当你查看一个多字节的整数时，它的字节顺序从高到低是依次存储的。例如，对于 0x12345678 这样的32位整数：

```

地址 内容

0x00 0x12 (高位字节)

0x01 0x34

0x02 0x56

0x03 0x78 (低位字节)

```

### 小端序（Little-Endian）

在小端序中，数据的低位字节存储在内存的低地址处，而高位字节存储在高地址处。因此，对于同样的32位整数 0x12345678，它的字节顺序在内存中是：

```

地址 内容

0x00 0x78 (低位字节)

0x01 0x56

0x02 0x34

0x03 0x12 (高位字节)

```

### 总结

- \*\*大端序\*\*：高位字节在低地址，低位字节在高地址。

- \*\*小端序\*\*：低位字节在低地址，高位字节在高地址。

### 实际应用

1. \*\*网络协议\*\*：大端序通常被用于网络协议中，许多网络协议（如TCP/IP）采用大端序，以确保不同机器之间的数据一致性。

2. \*\*处理器架构\*\*：

- \*\*小端序\*\*：x86和x86-64架构使用小端序。

- \*\*大端序\*\*：一些RISC架构，如SPARC和PowerPC可以选择大端或小端序。

### 字节序转换

在某些情况下，可能需要在不同字节序之间转换数据，以确保在不同的系统或协议中正确处理。大多数编程语言和库提供了相关的函数或工具来处理这种转换。

**也就是说一个01串数字，大端序时从左往右读，而小端序相反**