華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础实验

实验名称： 数据的表示

院　　系：计算机科学与技术

专业班级：CS2209

学　　号：U202210755

姓　　名：章宏宇

指导教师：李专

2024 年 3 月 12 日

一、实验目的与要求

⑴ 熟练掌握程序开发平台(VS2019) 的基本用法，包括程序的编译、链接和调试；

⑵ 熟悉地址的计算方法、地址的内存转换；

⑶ 熟悉数据的表示形式；

二、实验内容

任务 1　数据存放的压缩与解压编程

定义了 结构 student ，以及结构数组变量 old_s[N], new_s[N];　(N=5)

```
struct student {
    char   name[8];
    short   age;
    float   score;
    char   remark[200];   // 备注信息
};
```

编写程序，输入 N 个学生的信息到结构数组 old_s 中。将 old_s[N] 中的所有信息依次紧凑(压缩)存放到一个字符数组 message 中，然后从 message 解压缩到结构数组 new_s[N]中。打印压缩前(old_s)、解压后(new_s)的结果，以及压缩前、压缩后存放数据的长度。

要求：

（1） 输入的第 0 个人姓名(name)为自己的名字，分数为学号的最后两位；

（2） 编写指定接口的函数完成数据压缩

压缩函数有两个：　int　pack_student_bytebybyte(student* s, int sno, char *buf);

　　int　pack_student_whole(student* s, int sno, char *buf);

s 为待压缩数组的起始地址； sno 为压缩人数； buf 为压缩存储区的首地址；两个函数的返回均是调用函数压缩后的字节数。pack_student_bytebybyte 要求一个字节一个字节的向 buf 中写数据；pack_student_whole 要求对 short、float 字段都只能用一条语句整体写入，用 strcpy 实现串的写入。

（3） 使用指定方式调用压缩函数

old_s 数组的前 N1（N1=2）个记录压缩调用 pack_student_bytebybyte 完成；后 N2（N2==3）个记录压缩调用 pack_student_whole，两种压缩函数都只调用 1 次。

（4） 使用指定的函数完成数据的解压

解压函数的格式：int restore_student(char *buf, int len, student* s);

buf 为压缩区域存储区的首地址；len 为 buf 中存放数据的长度；s 为存放解压数据的结构数组的起始地址； 返回解压的人数。解压时不允许使用函数接口之外的信息（即不允许定义其他全局变量）

（5）仿照调试时看到的内存数据，以十六进制的形式，输出 message 的前 20 个字节的内容，并与调试时在内存窗口观察到的 message 的前 20 个字节比较是否一致。

（6）对于第 0 个学生的 score，根据浮点数的编码规则指出其个部分的编码，并与观察到

的内存表示比较，验证是否一致。

(7) 指出结构数组中个元素的存放规律，指出字符串数组、short 类型的数、float 型的数的存放规律。

任务 2 编写位运算程序

按照要求完成给定的功能，并**自动判断程序**的运行结果是否正确。（从逻辑电路与门、或门、非门等等角度，实现 CPU 的常见功能。所谓自动判断，即用简单的方式实现指定功能，并判断两个函数的输出是否相同。）

(1) int absVal(int x); 返回 x 的绝对值

仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 10 次

判断函数： int absVal_standard(int x) { return (x < 0) ? -x : x;}

(2) int negate(int x); 不使用负号，实现 -x

判断函数： int netgate_standard(int x) { return -x;}

(3) int bitAnd(int x, int y); 仅使用 ~ 和 |，实现 &

判断函数： int bitAnd_standard(int x, int y) { return x & y;}

(4) int bitOr(int x, int y); 仅使用 ~ 和 &，实现 |

(5) int bitXor(int x, int y); 仅使用 ~ 和 &，实现 ^

(6) int isTmax(int x); 判断 x 是否为最大的正整数（7FFFFFFF），

只能使用 !、 ~、 &、 ^、 |、 +

(7) int bitCount(int x); 统计 x 的二进制表示中 1 的个数

只能使用，!~&^|+<<>> ，运算次数不超过 40 次

(8) int bitMask(int highbit, int lowbit); 产生从 lowbit 到 highbit 全为 1，其他位为 0 的数。

例如 bitMask(5,3) = 0x38 ；要求只使用 !~&^|+<<>> ；运算次数不超过 16 次。

(9) int addOK(int x, int y); 当 x+y 会产生溢出时返回 1，否则返回 0

仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 20 次

(10) int byteSwap(int x, int n, int m); 将 x 的第 n 个字节与第 m 个字节交换，返回交换后的结果。 n、m 的取值在 0~3 之间。

例：byteSwap(0x12345678, 1, 3) = 0x56341278

byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD

仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>， 运算次数不超过 25 次

三、实验记录及问题回答

（1）任务 1 的算法思想、运行结果等记录

1. 对于 pack_student_bytebybyte 函数，考虑到需要对多个 student 类进行操作，所以拆分成对单个 student 类进行操作的函数 pack_student_bytebybyte_single 。

对于 pack_student_bytebybyte_single 函数，考虑到每个 student 中有四种类型，

故分别进行压缩。首先是 char 数组 name，先求出数组长度，然后逐一遍历加入 message 中，注意需要把终止符也加入。然后对于 short 类型的 age，因为 short 类型只占 2 个字节，故逐一加入 message 中。同理对于 float 类型的 score，已知 float 类型占 4 个字节，故把 buf 指针转成 float*类型，赋值为 s->score。最后对于 char 数组 remark，也是先求出数组长度再逐一加入 message。

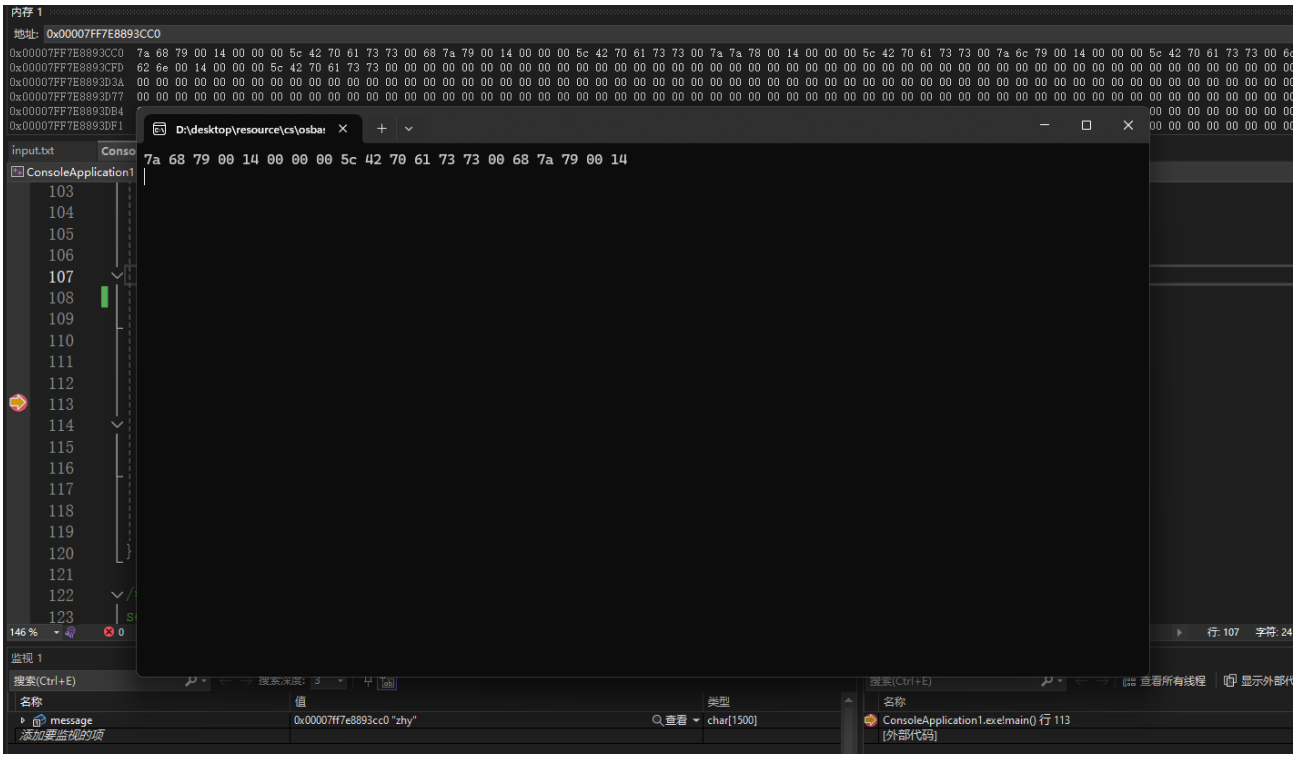
2. 对于 pack_student_whole 函数，同样拆分成单个 student 类的操作函数 pack_student_whole_single。

在 pack_student_whole_single 中，首先直接用 strcpy 函数将 name 复制到 message 中，因为 strcpy 把终止符一并复制，故不需要再做处理。然后对于 age 和 score，都采用把 char 指针强制转化成 short、float 类型指针再赋值。对于 remark 也采用跟 name 一样的做法。

3. 对于 restore_student 函数，同样拆分成单个 student 的操作函数 restore_student_single。

对于 name 和 remark，同样采用 strcpy 函数复制，无须考虑终止符。对于 age 和 score，也是将 buf 指针强制转换成对应类型赋值。大体上与解压函数一致。

4. 以十六进制的形式，输出 message 的前 20 个字节的内容，与调试时内存窗口的前 20 个字节比较，如下图：

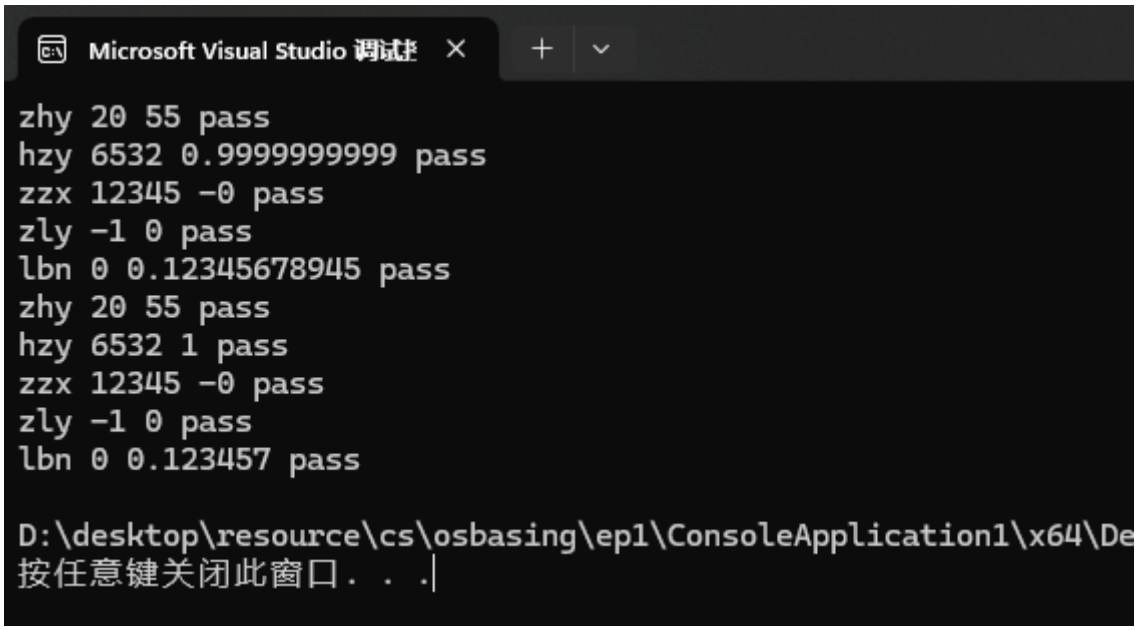


由图可见，完全一致。

5. 对于第 0 个学生的 score：“55”，根据 IEEE754 标准，可知正数符号位为 0，55 的二进制表示为 110111，故最高位为 2^5，故阶码为 5 加上偏移量 127 为(10000100)，尾码为去掉最高位 1 的其余位，末尾补 0：10111000000000000000000，最终用十六进制表示为 0x425c0000，与内存窗口观察到的一致。

6.结构数组中各个元素的存放按照声明顺序依次存放。如本类中 student 中有四个变量：name、age、score、remark，所以在内存中是也是按照这个顺序存放。需注意在本机上各变量是小端存放，即先存 LSB 再是 MSB，如 score 为 55，十六进制为 0x425c0000，在内存中是 00 00 5c 42。short 类型也一样。字符串数组则每位只有一个字节，不受小端法和大端法影响。

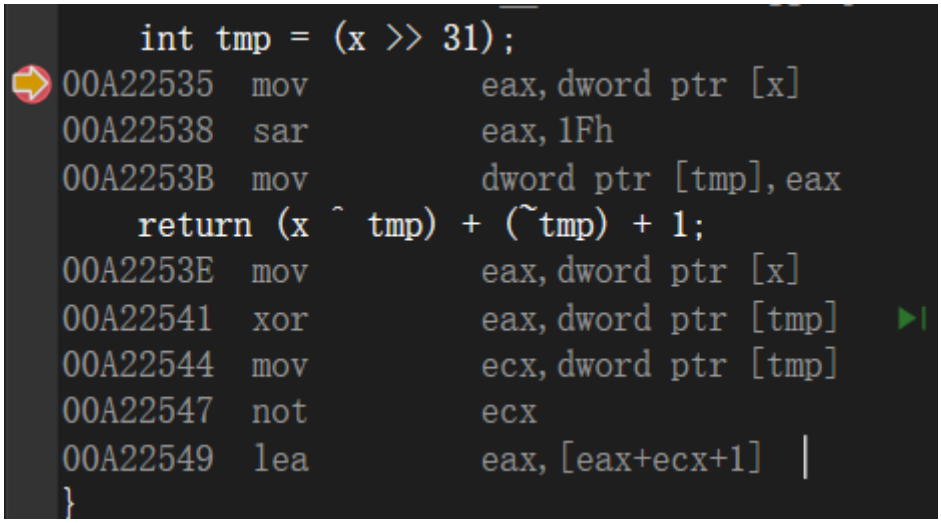
7.为了验证算法的正确性和实现的准确性，本机简单地测试了几组数据，以下是截图。



可以看到，压缩、解压后完全一致，故证明正确实现了所需功能。

（2）任务 2 的算法思想、运行结果等记录

1.对于 absVal 函数，因为已知取负可以表示成取补操作，即各位取反加一。又知道取反操作实际可以是异或全一，即补码表示下的-1。又知道负数算术右移后会补符号位，所以可以通过算术右移 31 位来得到-1。“加一”的 1 同理。反汇编结果如下：



实际运算次数为 8 次，小于任务要求 10 次，完美！

同时，也手写了测试函数 test_absVal，测试了 10 组随机数，结果均正确：

```
x: -74 absVal: 74 absVal_standard: 74
x: -69 absVal: 69 absVal_standard: 69
x: -57 absVal: 57 absVal_standard: 57
x: -19 absVal: 19 absVal_standard: 19
x: -83 absVal: 83 absVal_standard: 83
x: -49 absVal: 49 absVal_standard: 49
x: -39 absVal: 39 absVal_standard: 39
x: -84 absVal: 84 absVal_standard: 84
x: 34 absVal: 34 absVal_standard: 34
x: -57 absVal: 57 absVal_standard: 57
```

2.对于 negate 函数，已知负号在模意义下相当于取补操作，即各位取反后加一，所以可得 -x = ~x + 1;

3.对于 bitAnd 函数，采用逻辑式推导加上摩尔定理。(x&y) = ~(~(x&y)) = ~((~x)|(~y));

4.对于 bitOr 函数，跟 bitAnd 函数同理。(x|y) = ~(~(x|y)) = ~((~x)&(~y));

5.对于 bitXor 函数，根据实际意义可得：(x^y) = (x&(~y)) | ((~x)&y) = ~((~(x&(~y)) & (~((~x)&y))));

6.对于 isTmax 函数，考虑到补码的非对称性，我们可以判断 0x8fffffff 通过判断取负后是否等于自身。即判断 (~(x+1)) + 1 =? x + 1。等价于判断 (~(x+1)) =? x。

7.对于 bitCount 函数，我们注意到（实际上我试了 2h+才发现这个方法）若我们能够实现将相邻 n 位捆绑，预处理出 n 位对应的 1 的个数并存在该 n 位中，我们就可以用分治的思想运用加法运算合并相邻组，并在 log 级别的运算次数内完成统计。所以难点在于如何预处理。

我上机时首先想到充分运用位运算的三种运算符& | ^。我们处理出组内&的结果 x&，组内|的结果 x|，组内^的结果 x^，那么最后的结果就是 (x^) + 2 * (x&) + ((x^)^(x|))*2。对于*2 运算可以直接在处理时就左移一位，这样加法时自然带 2 的权。

容易发现实在是复杂，实测预处理就占了 30+的运算次数。同时注意到 3 位一组实在是不优美，32 无法整除 3 导致还要单独处理剩余 2 位。同时 3 位相比于 2 位在分治的运算次数上并无优势，顶多是 2/3，反而还复杂许多，实在不值。

那 4 位呢？我的智商不支持我推导出优美的 4 位表达式，在尝试了一晚上后无功而返。

故我回宿舍后想到不如直接 2 位。发现这表达式实在优雅，直接 (x&) + (x|)即可，还不用*2，实际表现也不错，截图如下：

```
    x = ((x & (x >> 1) ) & 0x55555555) + ((x | (x>>1)) & 0x55555555);
003C27D1  mov         eax,dword ptr [x]
003C27D4  sar         eax,1
003C27D6  and         eax,dword ptr [x]
003C27D9  and         eax,55555555h
003C27DE  mov         ecx,dword ptr [x]
003C27E1  sar         ecx,1
003C27E3  or          ecx,dword ptr [x]
003C27E6  and         ecx,55555555h
003C27EC  add         eax,ecx
003C27EE  mov         dword ptr [x],eax
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
003C27F1  mov         eax,dword ptr [x]
003C27F4  and         eax,33333333h
003C27F9  mov         ecx,dword ptr [x]
003C27FC  sar         ecx,2
003C27FF  and         ecx,33333333h
003C2805  add         eax,ecx
003C2807  mov         dword ptr [x],eax
    x = (x & 0x0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f);
003C280A  mov         eax,dword ptr [x]
003C280D  and         eax,0F0F0F0Fh
003C2812  mov         ecx,dword ptr [x]
003C2815  sar         ecx,4
003C2818  and         ecx,0F0F0F0Fh
003C281E  add         eax,ecx
003C2820  mov         dword ptr [x],eax
    x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
```

```
    x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
003C2823  mov         eax,dword ptr [x]
003C2826  and         eax,0FF00FFh
003C282B  mov         ecx,dword ptr [x]
003C282E  sar         ecx,8
003C2831  and         ecx,0FF00FFh
003C2837  add         eax,ecx
003C2839  mov         dword ptr [x],eax
    x += (x >> 16) & 0x0000ffff;
003C283C  mov         eax,dword ptr [x]
003C283F  sar         eax,10h
003C2842  and         eax,0FFFFh
003C2847  add         eax,dword ptr [x]
003C284A  mov         dword ptr [x],eax
    return x&0x0000ffff;
003C284D  mov         eax,dword ptr [x]
003C2850  and         eax,0FFFFh

}
```

实测只用了 38 次运算次数，小于任务要求的 40 次，完美！

同时，也手写了测试函数 test_bitCount 和采用 _popcount 函数的 bitCount_standard，测试了 10 组随机数，结果均正确：

```
x: 275471792 bitCount: 14 bitCount_standard: 14
x: 1137527756 bitCount: 16 bitCount_standard: 16
x: -1063330320 bitCount: 17 bitCount_standard: 17
x: -242371853 bitCount: 19 bitCount_standard: 19
x: -409230115 bitCount: 20 bitCount_standard: 20
x: -1174019408 bitCount: 14 bitCount_standard: 14
x: 928681775 bitCount: 19 bitCount_standard: 19
x: -655808320 bitCount: 13 bitCount_standard: 13
x: 1660222336 bitCount: 16 bitCount_standard: 16
x: -888353895 bitCount: 16 bitCount_standard: 16
```

8.对于 bitMask 函数，容易想到用做差的方法来构造连续的 1。所以我们可以用 2^(highbit+1) - 2^(lowbit) 即可，因为不能用减号，所以采用加上对应的负数即可。截

图如下：

```
    return (1 << (highbit + 1)) +  (~((1 << lowbit))) + 1;
00AD29D1  mov         ecx,dword ptr [highbit]
00AD29D4  add         ecx,1
00AD29D7  mov         eax,1
00AD29DC  shl         eax,cl
00AD29DE  mov         edx,1
00AD29E3  mov         ecx,dword ptr [lowbit]
00AD29E6  shl         edx,cl
00AD29E8  not         edx
00AD29EA  lea         eax,[eax+edx+1]
}
```

如图，实测只用了 9 次运算次数，远远小于任务要求的 16 次，完美！

9. 对于 addOK 函数，因为上课时讲到加法器时着重强调了如何判断溢出，所以容易想到只需判断 x，y，x+y 的符号即可。即若 x 和 y 符号相同，却和 x+y 符号不同则代表溢出。所以可得 (((~(x^y))&((x+y)^x))>>31)&1，看起来很复杂，截图如下：

```
    return (((~(x ^ y)) & (x ^ (x + y))) >> 31) & 1;
00E32621  mov         eax,dword ptr [x]
00E32624  xor         eax,dword ptr [y]
00E32627  not         eax
00E32629  mov         ecx,dword ptr [x]
00E3262C  add         ecx,dword ptr [y]
00E3262F  xor         ecx,dword ptr [x]
00E32632  and         eax,ecx
00E32634  sar         eax,1Fh
00E32637  and         eax,1
}
```

表达式看起来很复杂，但在 x86 环境下十分精简，只有 9 次运算次数，远小于任务要求的 20 次。就算是在 x64 环境下，也只需 16 次。完美！

10. 对于 byteSwap 函数，我们知道一个定理 x^a^a = x，即一个数异或上自身为 0。所以我们可以用^操作来交换。即把第 n 个字节和第 m 个字节先异或起来，然后再在 x 上进行操作，即可实现交换。

值得一提的是，由于我一开始不知道 x64 和 x86 的区别之大，所以死磕 x64。但无论怎么压都要 29 次。这时我灵机一动，于是对于 x ^ (tmp << n) ^ (tmp << m) 这行进行了微调，把 x 扔到了最后面。这样就直接减少了 2 次运算次数。

这是因为编译器很死板，对于 x ^ y 这样的操作，它只能把 x 放在 eax 寄存器中。即死板地按照表达式的顺序，即使实际上左右是可调换的。那么因为最高优先级是括号，所以当括号内运算完后，结果自然保存在 eax 中。这时若是下一个表达式能让括号结果在运算符左侧，则自然会减少 mov 指令，自然更优了。

但还是不如直接用 x86，直接少了一堆没必要的 mov 指令。

```
    n *= 8, m *= 8;
00B62C35  mov         eax,dword ptr [n]
00B62C38  shl         eax,3
00B62C3B  mov         dword ptr [n],eax
00B62C3E  mov         ecx,dword ptr [m]
00B62C41  shl         ecx,3
00B62C44  mov         dword ptr [m],ecx
    int tmp = ((x >> n) ^ (x >> m)) & 0xff;
00B62C47  mov         eax,dword ptr [x]
00B62C4A  mov         ecx,dword ptr [n]
00B62C4D  sar         eax,cl
00B62C4F  mov         edx,dword ptr [x]
00B62C52  mov         ecx,dword ptr [m]
00B62C55  sar         edx,cl
00B62C57  xor         eax,edx
00B62C59  and         eax,0FFh
00B62C5E  mov         dword ptr [tmp],eax
    return (tmp << n) ^ ((tmp << m) ^ x);
00B62C61  mov         eax,dword ptr [tmp]
00B62C64  mov         ecx,dword ptr [n]
00B62C67  shl         eax,cl
00B62C69  mov         edx,dword ptr [tmp]
00B62C6C  mov         ecx,dword ptr [m]
00B62C6F  shl         edx,cl
00B62C71  xor         edx,dword ptr [x]
00B62C74  xor         eax,edx
}
```

实测只需 23 次运算次数，小于任务要求的 25 次运算次数，完美！

测试了任务书中给的两个样例，均正确输出：

```
input = 12345678, output = 56341278
input = deadbeef, output = deefbead
```

四、体会

1. 在实验之前，说实话我并没有任何准备也没打算准备，甚至睡眼惺忪地来机房。因为我上课尚且认真听了，不说全部，也消化了一半左右。但这实验真是越做越清醒，极富挑战。就拿实验 2 来说。一开始的几道题都是小菜一碟，砍瓜切菜几分钟一道。但直到第七题的 bitCount，苦思冥想死活想不出来。就像我在实验记录中所说，前一两个小时，我都毫无思路，一度想出了压三位的方法，但最后实现的运算次数却远超出限制。周围的同学大多放弃了，选择看参考程序。但坚持还是有结果的，虽然我选择了错误的位数，但也启发了我思考如何整合结果，这自然而然地导向了分治合并。领悟了分治的做法后，自然而然地，我就想到压 3 位远远不如压 2 位的收益大，故最终实现了正确解法。

书山有路勤为径，学海无涯苦作舟。冥思苦想最终灵机一动，醍醐灌顶酣畅淋漓。我享受思考的过程，更自豪于思考的结果。

2.但在这里还是得提几点建议，首先题目限制含糊，没有给出常数的使用范围，全凭感觉，导致对于需要使用常数的地方不敢使用，如 bitCount 函数的分治过程中使用的常数。其次对于实验环境的选择并没有及时限制，导致在 x86 和 x64 环境下的运算次数差距过大，浪费了大把时间去卡无用的优化。其它方面还是很优秀的，老师也很耐心地解答疑问，点赞！

五、源码

实验 1、2、3 的源程序（单倍行距，5 号宋体字）

（1）

```
#include<iostream>
#include<string>
#include<string.h>
#define _CRT_SECURE_NO_WARNINGS
using namespace std;
#define N 5
struct student {
    char name[8];
    short age;//2
    float score;//4
    char remark[200];
}old_s[N], new_s[N];

char message[1500];
int m_id;
char* pack_student_bytebybyte_single(student* s, char* buf) {
    char* begin = buf, * end = buf;
    char* it = (char*)&(s);
    for (int i = 0; i < strlen(s->name) + 1; i++) {
        *end = s->name[i];
        ++end;
    }
    it = (char*)&(s->age);
    for (int i = 1; i <= 2; ++i) {
        *end = *it;
        ++end;
        ++it;
    }
    it = (char*)&(s->score);
    for (int i = 1; i <= 4; ++i) {
        *end = *it;
        ++end;
        ++it;
    }
    for (int i = 0; i < strlen(s->remark) + 1; ++i) {
```

```
        *end = s->remark[i];
        ++end;
    }
    return end;
}
int pack_student_bytebybyte(student* s, int sno, char* buf) {
    char* begin = buf;
    for (int i = 1; i <= sno; ++i) {
        buf = pack_student_bytebybyte_single(s, buf);
        s++;
    }
    return (int)(buf - begin);
}

char* pack_student_whole_single(student* s, char* buf) {
    char* end = buf;
    strcpy(buf, s->name);
    buf += strlen(s->name) + 1;
    short* tmp_s = (short*)(buf);
    *tmp_s = s->age;
    buf += 2;
    float* tmp_f = (float*)(buf);
    *tmp_f = s->score;
    buf += 4;
    strcpy(buf, s->remark);
    buf += strlen(s->remark) + 1;
    return buf;
}
int pack_student_whole(student* s, int sno, char* buf) {
    char* begin = buf;
    for (int i = 1; i <= sno; ++i) {
        buf = pack_student_whole_single(s, buf);
        s++;
    }
    return (int)(buf - begin);
}

char* restore_student_single(char* buf, student* s) {
    char* tmp = buf;
    strcpy(s->name, buf);
    buf += strlen(buf) + 1;
    s->age = *((short*)buf);
    buf += 2;
    s->score = *((float*)buf);
    buf += 4;
```

```
    strcpy(s->remark, buf);
    buf += strlen(buf) + 1;
    return buf;
}
int restore_student(char* buf, int len, student* s) {
    char* begin = buf;
    int cnt = 0;
    while (buf - begin <= len) {
        buf = restore_student_single(buf, s);
        s++;
        ++cnt;
    }
    return cnt;
}
int main() {
    freopen("input.txt", "r", stdin);
    for (int i = 0; i < 5; ++i) {
        cin >> old_s[i].name >> old_s[i].age >> old_s[i].score >> old_s[i].remark;
    }
    char* buf = message;
    student* s = old_s;
    buf += pack_student_bytebybyte(s, 2, buf);
    s += 2;
    buf += pack_student_whole(s, 3, buf);
    int len = buf - message;

    for (int i = 0; i < 20; ++i) {
        printf("%x ", message[i]);
    }
    puts("");


    int num = restore_student(message, len, new_s);
    for (int i = 0; i < 5; ++i) {
        cout << new_s[i].name << " " << new_s[i].age << " " << new_s[i].score << " " <<
new_s[i].remark << endl;
    }

    return 0;

}

/*
score = 55 = 001.1 0111
= 0  2^(5+127)  10111
```

```
= 0 10000100 10111000000000000000000
= 0x425c0000

score = 49 = 110001
= 0 2^(5+127) 10001
= 0 10000100 10001000000

struct : 顺序

short:low-end
float:low-end

*/
```

（2）

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <ctime>
#include <bit>

using namespace std;
inline int ran(int n) {
    return 1ll * rand() * rand() % n + 1;
}
int absVal(int x) {
    int tmp = (x >> 31);
    return (x ^ tmp) + (~tmp) + 1;
}
int absVal_standard(int x) {
    return (x < 0) ? -x : x;
}
void test_absVal() {
    int n = 10,lim = 100;
    while (n--) {
        int x = ran(2 * lim) - lim;
        if (absVal(x) != absVal_standard(x)) {
            cout << x << " " << absVal(x) << " " << absVal_standard(x) << endl;
        }
    }
}

int negate(int x) {
    return (~x) + 1;
```

```
}
int negate_standard(int x) {
    return -x;
}
void test_negate() {
    int n = 10, lim = 100;
    while (n--) {
        int x = ran(2 * lim) - lim;
        if (negate(x) != negate_standard(x)) {
            cout << x << " " << negate(x) << " " << negate_standard(x) << endl;
        }
    }
}

int bitAnd(int x, int y) {
    return ~((~x) | (~y));
}
int bitAnd_standard(int x, int y) {
    return x & y;
}
void test_bitAnd() {
    int n = 10, lim = 100;
    while (n--) {
        int x = ran(2 * lim) - lim, y = ran(2 * lim) - lim;
        if (bitAnd(x,y) != bitAnd_standard(x,y)) {
            cout << x << " " << y << " " << bitAnd(x, y) << " " << bitAnd_standard(x, y) <<
endl;
        }
    }
}

int bitOr(int x, int y) {
    return ~((~x) & (~y));
}
int bitOr_standard(int x, int y) {
    return x | y;
}
void test_bitOr() {
    int n = 10, lim = 100;
    while (n--) {
        int x = ran(2 * lim) - lim, y = ran(2 * lim) - lim;
        if (bitOr(x, y) != bitOr_standard(x, y)) {
            cout << x << " " << y << " " << bitOr(x, y) << " " << bitOr_standard(x, y) <<
endl;
        }
```

```
    }
}


int bitXor(int x, int y) {
    return ~( (~(x & (~y) ) ) & (~( (~x) & y) ) );
}
int bitXor_standard(int x, int y) {
    return x ^ y;
}
void test_bitXor() {
    int n = 10, lim = 100;
    while (n--) {
        int x = ran(2 * lim) - lim, y = ran(2 * lim) - lim;
        if (bitXor(x, y) != bitXor_standard(x, y)) {
            cout << x << " " << y << " " << bitXor(x, y) << " " << bitXor_standard(x, y) <<
endl;
        }
    }
}


int isTmax(int x) {
    return (~(x + 1)) == x;
}

int bitCount(int x) {
//  register int cnt = 0;
//  while (x) {
//      ++cnt;
//      x &= (x - 1);
//  }
//  ^ 000 010 001 010: 1,3 -> 1   0,2 -> 0
//  & : 3->1
//  | : 0->0
//  ^ + 2*& + (^)^(|)*2
//   000: 000
//   001: 010 001 100
//   010: 011 101 110
//   011: 111
//
//

/*
    const int mod3=1227133513;
```

```
    int x1 = x >> 1, x2 = x >> 2;
    int x_xor = x ^ x1 ^ x2, x_and = x & x1 & x2, x_or = x | x1 | x2; 不行，除了关键位的其余位不好清
    x_xor&=mod3, x_and
    x_and <<= 1;
    x_or = (x_xor ^ x_or) << 1;
    x_xor += x_and + x_or;


*/
//  0000 0001 0010 0100:
//  ^: 1,3->1 2,4->0
//  |: 0->0
//  &: 4->1S
//
//  00 01 10 11
//  00 - 0
//  01 - 0
//  10 - 1
//  11 - 1
//
//  00 11
//  00001111
//  0000000011111111
//  00000000 00000000 11111111 11111111
//  00 01 10 11
//  |  1 2 -> 1
//  ^  1 -> 1
//  &  2 -> 1
//  0x0101
    x = ((x & (x >> 1) ) & 0x55555555) + ((x | (x>>1)) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f);
    x = (x & 0x00ff00ff) + ((x >> 8) & 0x00ff00ff);
    x += (x >> 16) & 0x0000ffff;
    return x&0x0000ffff;

}
int bitCount_standard(int x) {
    unsigned int y = x;
    return  __popcnt(y);
}
void test_bitCount() {
    int n = 10, lim = 100000000;
    while (n--) {
```

```
        int x = ran(2 * lim) * ran(2 * lim) * ran(2 * lim);
        if (bitCount(x) != bitCount_standard(x)) {
            cout << x << " " << bitCount(x) << " " << bitCount_standard(x) << endl;
        }
    }

}

int bitMask(int highbit, int lowbit) {

    return (1 << (highbit + 1)) +  (~((1 << lowbit))) + 1;
}
int bitMask_standard(int highbit, int lowbit) {
    int x=0;
    for (int i = lowbit; i <= highbit; ++i) {
        x += 1 << i;
    }
    return x;
}
void test_bitMask() {
    int n = 10, lim = 32;
    while (n--) {
        int h = ran(lim) - 1, l = ran(lim) - 1;
        if (h < l)swap(h, l);
        if (bitMask(h, l) != bitMask_standard(h, l)) {
            cout << h << " " << l << " " << bitMask(h, l) << " " << bitMask_standard(h, l)
<< endl;
        }
    }
}

int addOK(int x, int y) {

    return (((~(x ^ y)) & (x ^ (x + y))) >> 31) & 1;
}
int addOK_standard(int x, int y) {
    return (x > 0 && y > 0 && x + y < 0) || (x < 0 && y < 0 && x + y>0);
}
void test_addOK() {
    int n = 1000000, lim = 1e9;
    while (n--) {
        int x = 1ll * ran(lim) * ran(lim) * ran(lim), y = 1ll * ran(lim) * ran(lim) *
ran(lim);
        if (addOK(x, y) != addOK_standard(x, y)) {
            cout << x << " " << y << " " << addOK(x, y) << " " << addOK_standard(x, y) <<
```

```
endl;
        }
    }
}
int byteSwap(int x, int n, int m) {
    n *= 8, m *= 8;
    int tmp = ((x >> n) ^ (x >> m)) & 0xff;
    return (tmp << n) ^ ((tmp << m) ^ x);
}
// 改变运算顺序可以优化指令数
// 用 x86 不用 x64
int main() {
    srand((unsigned(time(0))));
//  test_absVal();
//  test_negate();
//  test_bitAnd();
//  test_bitOr();
//  test_bitXor();
//  cout << isTmax(0x7fffffff) << endl;
//  if (isTmax((1ll << 31) - 1))cout << "Yes" << endl;
//  else cout << "No" << endl;
//  test_bitCount();
//  bitCount_second(-1);
//  bitCount(-1);
//  test_bitMask();
//  test_addOK();
//  cout << addOK(1, -3) << endl;
//  printf("%8x\n", byteSwap(0x12345678, 1, 3));
    printf("%8x\n", byteSwap(0xdeadbeef, 0, 2));
    return 0;
}
```