第七届

全国大学生集成电路创新创业大赛

报告类型*：设计报告

参赛杯赛*：景嘉微杯

作品名称*：一种 ARGB 数据无损压缩/解压单元

队伍编号*：CICC1709

团队名称*：三三

一种 ARGB 数据无损压缩/解压单元

在 GPU、AI 等芯片设计领域，存储器访问往往是系统性能的瓶颈，提高存储器的访问效率对于提升芯片性能的意义重大，其中对颜色缓冲区数据（ARGB）的频繁读写对性能的影响很大；本课题从数据压缩的角度，通过减小访问颜色缓冲区的数据量来提高存储器的带宽和访问效率。本次设计旨在实现一种 ARGB 数据（二进制数据）的无损压缩/解压单元，用于 GPU 或其它存储器图形图像访问密集的系统中，利用无损数据压缩技术降低存储器带宽，提高访问效率。

本次设计报告包括以下几个部分：

1. 简介
2. 试验任务
3. 软件设计
4. 测试验证

1. 简介

数据压缩是一门通信原理和计算机科学都会涉及到的学科，在通信原理中，一般称为信源编码，在计算机科学里，一般称为数据压缩，两者本质上没有什么区别，在数学家看来，都是映射。在数据压缩的过程中，根据是否能够完整的还原原始信息，压缩又被分为有损压缩和无损压缩，有损，指的是压缩之后就无法完整还原原始信息，但是压缩率可以很高，主要应用于视频、话音等数据的压缩；无损压缩则用于文件等等必须完整还原信息的场合，Huffman、游程编码、JPEG-LS 等都是无损压缩，Huffman 和游程编码是一种通用压缩，常用于重复度较高的文本文件，JPEG－LS 则常用于压缩图片文件。本次设计则是参考了这三种算法。

2. 试验任务

本次项目的设计任务是实现一种 ARGB 数据（二进制数据）的无损压缩/解压单元，总体要求如下：

（1）支持线性块或二维块的 ARGB 数据压缩和解压；块大小支持 256Byte/512Byte/1KByte；

（2）输入：压缩：指定的图像 ARGB 数据；解压：压缩后数据；

（3）输出：压缩：压缩后的数据；解压：解压后 ARGB 数据；

（4）语言及标准: c99 without libs；

（5）输入压缩后数据执行解压后得到的结果与原始图像完全一致；

（6）能对指定大小的线性数据块/二维数据块执行压缩/解压

（7）能对 4Byte~1KByte 数据多次带压缩/解压读写；

3. 软件设计

在本次项目的软件设计中，整体设计流程如下：

第一步，对指定块数据进行 LOCO-I 预测。通过控制待压缩文件读取的位置，以实现指定块读取 ARGB 数据，并将其分通道存储进入结构体中。然后对 A、R、G、B 四个通道的数据分别进行 LOCO-I 预测，随后将预测的数据按照 A、R、G、B 分别写入预测差值文件。还原时按照文件写入的方式再读取，然后进行 LOCO-I 还原，并按照原文件的排列顺序写入还原文件。

第二步，对得到的差值文件进行游程压缩。由于自然图像的局域相似性，经过 LOCO-I 预测的处理后，差值数据中会出现较多的单字节及中等长度以上的 0 值像素，其他大小的像素值重复出现较少。因此在该游程编码中，仅实现对两个字节及以上的 0 值进行编码，解压时按照同样思想。

第三步，对游程压缩后的数据进行哈夫曼压缩。由于经由 LOCO-I 预测后的数据像素值会集中在 0 值附近，再经过对中等长度 0 值的游程编码后，数据大量集中在小像素值，然后对该数据进行哈夫曼压缩，得到最终压缩后的数据。

软件代码所定义的函数及结构体如下所示：

```
void predict_LOCO_I();            //LOCO-I 预测
void restore_LOCO_I();            //LOCO-I 还原
void LOCO_I_encode();             //LOCO-I 编码
void LOCO_I_decode();             //LOCO-I 解码
void code_length_compress();     //游程压缩
void code_length_decompress();  //游程解压
void select();                    //选择最小的两个节点以建立哈夫曼树
void CreatTree();                 //建立哈夫曼树
void HufCode();                   //生成哈夫曼编码
```

```
int huffman_compress();          //哈夫曼压缩
int huffman_extract();           //哈夫曼解压
int getFileSize();               //计算文件大小

//ARGB 数据分开存储
typedef struct{
    unsigned char color_A;
    unsigned char color_R;
    unsigned char color_G;
    unsigned char color_B;
}BLOCK_DATA;

// 统计字符频度的临时结点
typedef struct {
    unsigned char uch;          // 以 8bits 为单元的无符号字符
    unsigned long weight;       // 每类（以二进制编码区分）字符出现频度
} TmpNode;

// 哈夫曼树结点
typedef struct {
    unsigned char uch;              // 以 8bits 为单元的无符号字符
    unsigned long weight;           // 每类（以二进制编码区分）字符出现频度
    char *code;                     // 字符对应的哈夫曼编码（动态分配存储空间）
    int parent, lchild, rchild;     // 定义双亲和左右孩子
} HufNode, *HufTree;
```

下面分别介绍各个函数的原理及实现：

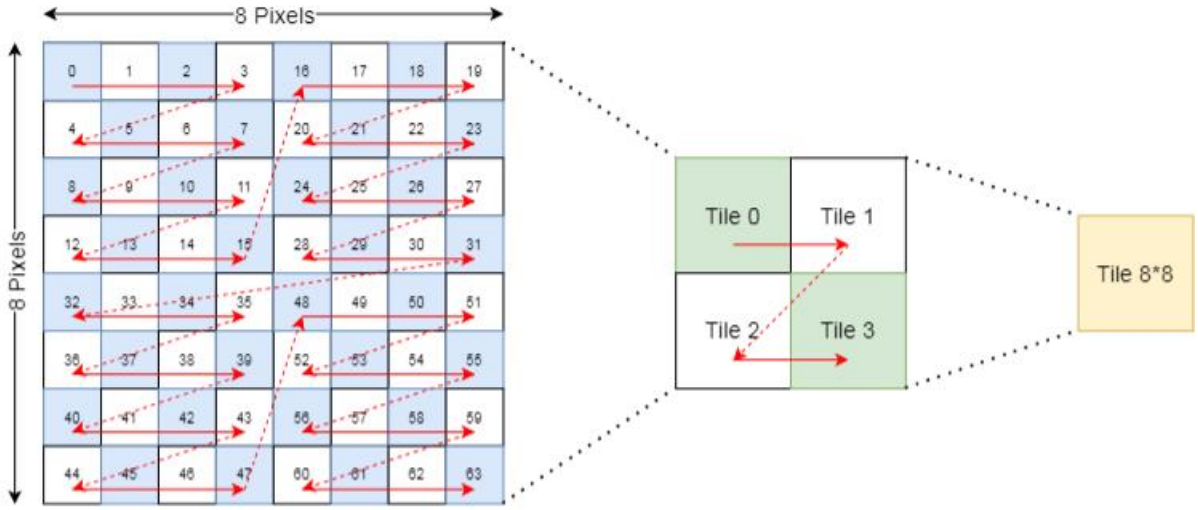
LOCO-I 预测函数：LOCO-I 预测过程是基于上文来数据来预测当前数据的一种算法，具体原理为，如果当前像素处于上边界，则预测数据 = 左数据，如果当前像素处于左边界，则预测数据 = 上一行数据，如果当前数据的左方及上方都存在数据，那么预测数据等于左数据上数据以及左上数据的中位值，然后通过实际值与预测值进行异或，得到差值。LOCO-I 还原函数的过程与此相同，但是通过差值与预测值进行异或从而还原实际值。

```
void predict_LOCO_I(unsigned char *data, int width, int height, unsigned char *difft){
    int i;
    unsigned char pred;
    for(i=0;i<width*height;i++){
        if(i == 0) pred = 0;
        //上边界：预测数据 = 左数据
        else if(i/width == 0) pred = data[i-1];
        //左边界：预测数据 = 上数据
        else if(i%width == 0) pred = data[i-width];
        else{
            if(data[i - (width+1)] >= __max(data[i-1], data[i-width])){
                pred = __min(data[i-1], data[i - width]);
            }else if(data[i - (width+1)] <= __min(data[i-1], data[i-width])){
                pred = __max(data[i-1], data[i-width]);
```

```
            }else{
                pred = data[i-1] + data[i-width] - data[i-(width+1)];
            }
        }
        difft[i] = data[i] ^ pred;
    }
}
```

LOCO-I 编码函数：该过程包括从原图像数据中按块取出数据、通过调用 LOCO-I 预测函数进行预测得到差值、将差值按一定次序排列存储。

首先介绍数据块的取出，出如下图所示，以 8*8 的原图像，4*4 的数据块为例，该图介绍了数据块的排列和读取方式。



除此之外，程序设计需要支持其他块大小的数据，比如 8*8，16*16 的大小，因此这一部分程序设计思想如下，利用 fseek 函数移动文件光标，首先移动到所需要读取块位置的开头，如要读取上图的 Tile0，则文件光标需要移动到待读取文件的 0 的位置，读取 tile1，则需移动到 4 的位置，读取 tile2 为 32，tile3 为 36。以下为实现代码。

```
//文件光标移动到所需块的开头
fseek(infile,ARGB*sizeof(unsigned char)*(width*block_height*(int)(n/w_div) + block_width*(n%w_div)),
SEEK_SET);
```

其中 ARGB 为 4，表示 4 个通道的数据，ARGB*sizeof(unsigned char)为一个像素所占的字节数，width 为图片的宽度，block_height 为数据块的宽度，w_div 的值为 width/block_width，表示图片可以分割的列数。因此 ARGB*sizeof(unsigned char)*width*block_height*(int)(n/w_div)即用来判断所需读取的块数在图片的第几行，而 ARGB*sizeof(unsigned char)*block_width*(n%w_div)则用来判断读取的数据在第几列，将两项加起来，便得到了所需读取的块开头位置。以下是实现代码。

```
//文件光标移动到所需块的开头
fseek(infile,ARGB*sizeof(unsigned char)*(width*block_height*(int)(n/w_div) + block_width*(n%w_div)),
SEEK_SET);
    //从源文件读取 ARGB 数据
    for(h=0;h<block_height;h++){
        for(i=0;i<block_width;i++){
            fread(&block_data[i+h*block_width].color_A, sizeof(unsigned char), 1, infile);
```

```
            fread(&block_data[i+h*block_width].color_R, sizeof(unsigned char), 1, infile);
            fread(&block_data[i+h*block_width].color_G, sizeof(unsigned char), 1, infile);
            fread(&block_data[i+h*block_width].color_B, sizeof(unsigned char), 1, infile);
        }
        //文件光标移动到下一行
        fseek(infile, ARGB*sizeof(unsigned char)*(width-block_width),SEEK_CUR);
```

然后将得到数数据分通道进行 LOCO-I 预测

```
    //预测差值
    for(i=0;i<ARGB;i++)
        predict_LOCO_I(data_temp[i], block_width, block_height, dif[i]);
```

最后将指定块差值数据存储到指定位置，排列按照全部的 A，全部的 R...G...B...存入差值文件，一共有 width*height*ARGB*sizeof(unsigned char)位数据，其中 ARGB 分别占用四等份

A 从 width*height*sizeof(unsigned char)*A 开始，存入 width*height*sizeof(unsigned char)个数据

R 从 width*height*sizeof(unsigned char)*R 开始

G 从 width*height*sizeof(unsigned char)*G 开始

B 从 width*height*sizeof(unsigned char)*B 开始

内部的 ARGB 各个小块又加上偏移 sizeof(unsigned char)*block_width*block_height*n

也就是说，指针定在 width*height*sizeof(unsigned char)*i + sizeof(unsigned char)*block_width*block_height*n

即 sizeof(unsigned char)*（width*height*i+block_width*block_height*n）

因此实现代码如下：

```
fseek(outdif,sizeof(unsigned char)*(width*height*A+block_width*block_height*n), SEEK_SET);
fwrite(dif[A], sizeof(unsigned char), block_width*block_height, outdif);
fseek(outdif,sizeof(unsigned char)*(width*height*R+block_width*block_height*n), SEEK_SET);
fwrite(dif[R], sizeof(unsigned char), block_width*block_height, outdif);
fseek(outdif,sizeof(unsigned char)*(width*height*G+block_width*block_height*n), SEEK_SET);
fwrite(dif[G], sizeof(unsigned char), block_width*block_height, outdif);
fseek(outdif,sizeof(unsigned char)*(width*height*B+block_width*block_height*n), SEEK_SET);
fwrite(dif[B], sizeof(unsigned char), block_width*block_height, outdif);
```

LOCO-I 解码函数则与此过程相反，但思想相同，解码时按照差值文件的数据排列方式确定数据块的开头位置，并通过 LOCO-I 还原函数，最后按照同编码时读取原图像那样写入原图像，则完成解码。

游程压缩函数：该函数思想在于将中长字节的 0 值数据进行游程压缩，实现过程如下，首先读取一字节数据判断是否为 0，若为 0 则再读取一字节数据，若为 0，则计数加 1，表示两个字节的 0 值，知道读取的下以为数据不为 0 时，输出两个字节的 0，再输出一字节大小的计数值，若计数超出范围则立即输出，之后重新计数。实现代码如下：

```
while(1){
        fread(&current, sizeof(unsigned char), 1, input_file);
        if(feof(input_file))break;
        if (current==0){
            fread(&current, sizeof(unsigned char), 1, input_file);
            if(feof(input_file)){
                flag = 1;
                break;
```

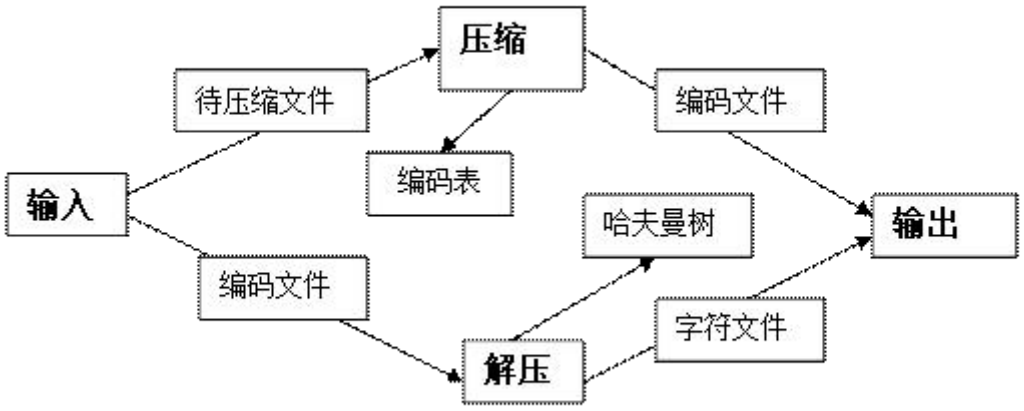
```
            }
            if(current == 0){
                count++;
                if (count == 255) {
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);//先写两个 0，再写重复次数
                    fwrite(&count, sizeof(unsigned char), 1, output_file);
                    count = 0;
                }
            }else{
                if(count > 0){
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);//先写两个 0，再写重复次数
                    fwrite(&count, sizeof(unsigned char), 1, output_file);
                    count = 0;
                }
                fwrite(&zero, sizeof(unsigned char), 1, output_file);
                fwrite(&current, sizeof(unsigned char), 1, output_file);
            }
        }else{
            if (count > 0) {
                fwrite(&zero, sizeof(unsigned char), 1, output_file);
                fwrite(&zero, sizeof(unsigned char), 1, output_file);//先写两个 0，再写重复次数
                fwrite(&count, sizeof(unsigned char), 1, output_file);
                count = 0;
            }
            fwrite(&current, sizeof(unsigned char), 1, output_file);
        }
    }
    if(count > 0){//如果文件末尾一直是 0，但不够 255 个，则最后输出重复个数
        fwrite(&zero, sizeof(unsigned char), 1, output_file);
        fwrite(&zero, sizeof(unsigned char), 1, output_file);//先写两个 0，再写重复次数
        fwrite(&count, sizeof(unsigned char), 1, output_file);
        count = 0;
    }
    if(flag == 1){
        fwrite(&zero, sizeof(unsigned char), 1, output_file);
        flag =0;
    }
```

该游程压缩对应的解压函数与压缩函数相对应，具体过程如下，如果连续读取到两个字节的 0 值，则下一字节数据表示两个字节 0 值的重复个数，根据重复次数在解压文件中写入 0 值，若没有检测到 0 值则正常输出。实现代码如下：

```
while (1) {
        fread(&current, sizeof(current), 1, input_file);
        if(feof(input_file))break;
```

```
        if(current == 0){
            fread(&current, sizeof(current), 1, input_file);
            if(feof(input_file)){
                fwrite(&zero, sizeof(unsigned char), 1, output_file);
                break;
            }
            if(current == 0){
                fread(&count, sizeof(current), 1, input_file);
                for(i=0;i<count;i++){
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);
                    fwrite(&zero, sizeof(unsigned char), 1, output_file);
                }
            }else{
                fwrite(&zero, sizeof(unsigned char), 1, output_file);
                fwrite(&current, sizeof(unsigned char), 1, output_file);
            }
        }else{
            fwrite(&current, sizeof(unsigned char), 1, output_file);
        }
    }
```

Huffman 压缩函数：为了建立哈夫曼树，首先需要扫描源文件，统计每类字符出现的频度，然后根据字符频度建立哈夫曼树，接着根据哈夫曼树生成哈夫曼编码。再次扫描文件，每次读取一字节的数据，根据“字符—编码”表，匹配编码，并将编码存入压缩文件，同时存入编码表。解压时，首先读取编码表，然后读取编码匹配编码表找到对应字符，之后存入文件，完成解压。该过程的 UML 协同图如下：



为了建立 Huffman 树，首先需要统计各类字符的频度：首先定义静态数组，用数组的下标匹配字符，不需扫描数组就可以找到每类字符的位置，达到随机存储的目的，效率有很大的提高。当然，不一定每类字符都出现，因此在统计完后，需要排序，将字符频度为零的结点剔除。实现代码如下：

```
    TmpNode *tmp_nodes =(TmpNode *)malloc(256*sizeof(TmpNode));  // 动态分配 256 个结点
    for(i = 0; i < 256; ++i)
    {
```

```
        tmp_nodes[i].weight = 0;
        tmp_nodes[i].uch = (unsigned char)i;          // 数组的 256 个下标与 256 种字符对应
    }
    infile = fopen(ifname, "rb");
    if (infile == NULL)
        return -1;
    fread((char *)&char_temp, sizeof(unsigned char), 1, infile);          // 从输入文件数据流读入一个字符暂存
    while(!feof(infile))     //判断文件是否结束
    {
        ++tmp_nodes[char_temp].weight;        // 统计下标对应字符的权重，利用随机访问快速统计字符频度
        ++file_len;
        fread((char *)&char_temp, sizeof(unsigned char), 1, infile);          // 读入一个字符
    }
    fclose(infile);
    // 排序，将频度为零的放最后，剔除
    for(i = 0; i < 256-1; ++i)
        for(j = i+1; j < 256; ++j)            //冒泡排序
            if(tmp_nodes[i].weight < tmp_nodes[j].weight)
            {
                node_temp = tmp_nodes[i];
                tmp_nodes[i] = tmp_nodes[j];
                tmp_nodes[j] = node_temp;
            }
```

建立哈夫曼树的过程如下：哈夫曼树为二叉树，树结点含有权重（在这里为字符频度，同时也要把频度相关联的字符保存在结点中）、左右孩子、双亲等信息。考虑到建立哈夫曼树所需结点会比较多，也比较大，如果静态分配，会浪费很大空间，因此在这里使用动态分配的方法，并且，为了利用数组的随机访问特性，也将所需的所有树节点一次性动态分配，保证其内存的连续性。另外，结点中存储编码的域，由于长度不定，也动态分配内存。建立哈夫曼树的函数代码如下：

```
void CreateTree(HufNode *huf_tree, unsigned int char_kinds, unsigned int node_num)
{
    unsigned int i;
    int s1, s2;
    for(i = char_kinds; i < node_num; ++i)
    {
        select(huf_tree, i, &s1, &s2);        // 选择最小的两个结点
        huf_tree[s1].parent = huf_tree[s2].parent = i;
        huf_tree[i].lchild = s1;
        huf_tree[i].rchild = s2;
        huf_tree[i].weight = huf_tree[s1].weight + huf_tree[s2].weight;
    }
}
```

接下来生成哈夫曼编码，每类字符对应一串编码，故从叶子结点（字符所在结点）由下往上生成每类字符对应的编码，左‘0’，右‘1’。为了得到正向的编码，设置一个编码缓存数组，从后往前保存，然后从前往后拷贝到叶子结点对应编码域中，根据得到的编码长度为编码域分配空间。对于缓存数组的大小，由于

字符种类最多为 256 种，构建的哈夫曼树最多有 256 个叶子结点，树的深度最大为 255，故编码最长为 255，所以分配 256 个空间，最后一位用于保存结束标志。实现函数如下

```
void HufCode(HufNode *huf_tree, unsigned char_kinds)
{
    unsigned int i;
    int cur, next, index;
    char *code_tmp = (char *)malloc(256*sizeof(char));        // 暂存编码，编码长度不超多 255
    code_tmp[256-1] = '\0';

    for(i = 0; i < char_kinds; ++i)
    {
        index = 256-1;  // 编码临时空间索引初始化
        // 从叶子向根反向遍历求编码
        for(cur = i, next = huf_tree[i].parent; next != 0;
            cur = next, next = huf_tree[next].parent)
            if(huf_tree[next].lchild == cur)
                code_tmp[--index] = '0';     // 左‘0’
            else
                code_tmp[--index] = '1';     // 右‘1’
        huf_tree[i].code = (char *)malloc((256-index)*sizeof(char));              // 为第 i 个字符编码动态分配存储空间
        strcpy(huf_tree[i].code, &code_tmp[index]);       // 正向保存编码到树结点相应域中
    }
    free(code_tmp);       // 释放编码临时空间
}
```

上面协定以 8 位的字符为单元编码，这里压缩当然也以 8 位为处理单元。首先将字符及种类和编码（编码表）存储于压缩文件中，供解压时使用。然后以二进制打开源文件，每次读取一个 8 位的无符号字符，循环扫描匹配存储于哈夫曼树节点中的编码信息。由于编码长度不定，故需要一个编码缓存，待编码满足 8 位时才写入，文件结束时缓存中可能不足 8 位，在后面补 0，凑足 8 位写入，并将编码的长度随后存入文件。在哈夫曼树节点中，编码的每一位都是以字符形式保存的，占用空间很大，不可以直接写入压缩文件，故需要转为二进制形式写入；

以下是压缩文件的存储结构：字符种类用来判断读取的字符、频度序偶的个数，同时用来计算哈夫曼结点的个数；文件长度用来控制解码生成的字符个数，即判断解码结束。

字符种类	字符、频度序偶.....	文件长度	哈夫曼编码

Huffman 解压函数：执行解压时，以二进制方式打开压缩文件，首先将文件前端的字符种类数读取出来，据此动态分配足够空间，然后将随后的字符—编码表读取处理保存到动态分配的结点中，然后以 8 位为处理单元，依次读取随后的编码匹配对应的字符，这里对比编码依然用在文件压缩中所用的方法，就是用 C 语言的位操作，同 0x80 与操作，判断 8bits 字符的最高位是否为‘1’，对比一位后，左移一位，将最高位移除，次高位移到最高位，依次对比。这次是从编码到字符反向匹配，与压缩时有一点不同，需要用读取的编码逐位与编码表中的编码进行对比，对比一位后，增加一位再对比，而且每次对比都是一个循环（与每个字符的编码对比），效率很低。

因此在这里可以设计为可以将哈夫曼树保存到文件中，解码时，从树根到叶子对比编码，只要一次遍历就可以找到编码对应的存于叶子结点中的字符，极大提高了效率。然而，树结点中有字符、编码、左右孩子、双亲，而且孩子和双亲还必须是整型的（树节点最多为 256*2-1=511 个），占用空间很大，会导致

压缩文件变大。

进一步考虑，可以仅存储字符及对应频度（频度为 unsigned long，一般情况下与 int 占用空间一样，同为 4 个字节），解码时读取数据重建哈夫曼树，这样就解决了空间问题。

虽然重建哈夫曼树（双重循环，每个循环的次数最大为 511）也要花费一定的时间，但是相对上面的与编码表匹配（每位编码都要循环匹配所有字符（最多为 256 种）一次，而总的编码位数一般很大，且随着文件变大而增长）所花费的时间更少。

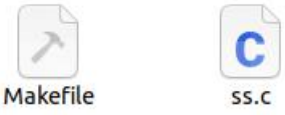
最后实现的解压函数如下：

```
int huffman_extract(char *ifname, char *ofname){
    unsigned int i;
    unsigned long file_len;
    unsigned long writen_len = 0;          // 控制文件写入长度
    FILE *infile, *outfile;
    unsigned int char_kinds;           // 存储字符种类
    unsigned int node_num;
    HufTree huf_tree;
    unsigned char code_temp;           // 暂存 8bits 编码
    unsigned int root;        // 保存根节点索引，供匹配编码使用
    infile = fopen(ifname, "rb");          // 以二进制方式打开压缩文件
    // 判断输入文件是否存在
    if (infile == NULL)
        return -1;
    // 读取压缩文件前端的字符及对应编码，用于重建哈夫曼树
    fread((char *)&char_kinds, sizeof(unsigned int), 1, infile);       // 读取字符种类数
    if (char_kinds == 1){
        fread((char *)&code_temp, sizeof(unsigned char), 1, infile);       // 读取唯一的字符
        fread((char *)&file_len, sizeof(unsigned long), 1, infile);       // 读取文件长度
        outfile = fopen(ofname, "wb");                        // 打开压缩后将生成的文件
        while (file_len--)
            fwrite((char *)&code_temp, sizeof(unsigned char), 1, outfile);
        fclose(infile);
        fclose(outfile);
    }
    else{
        node_num = 2 * char_kinds - 1;        // 根据字符种类数，计算建立哈夫曼树所需结点数
        huf_tree = (HufNode *)malloc(node_num*sizeof(HufNode));       // 动态分配哈夫曼树结点空间
        // 读取字符及对应权重，存入哈夫曼树节点
        for(i = 0; i < char_kinds; ++i){
            fread((char *)&huf_tree[i].uch, sizeof(unsigned char), 1, infile);        // 读入字符
            fread((char *)&huf_tree[i].weight, sizeof(unsigned long), 1, infile);     // 读入字符对应权重
            huf_tree[i].parent = 0;
        }
        for(; i < node_num; ++i)
            huf_tree[i].parent = 0;
        CreateTree(huf_tree, char_kinds, node_num);       // 重建哈夫曼树（与压缩时的一致）
        // 读完字符和权重信息，紧接着读取文件长度和编码，进行解码
        fread((char *)&file_len, sizeof(unsigned long), 1, infile); // 读入文件长度
```

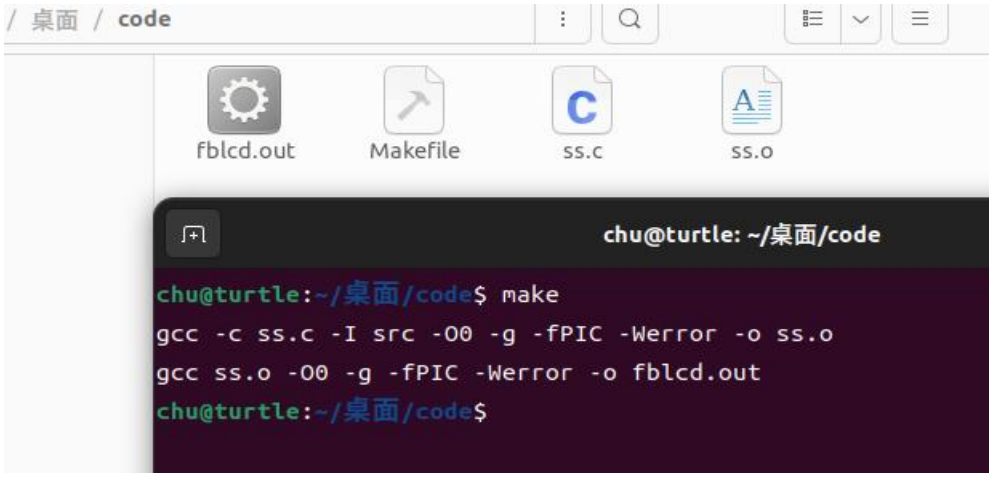
```
        outfile = fopen(ofname, "wb");        // 打开压缩后将生成的文件
        root = node_num-1;
        while(1){
            fread((char *)&code_temp, sizeof(unsigned char), 1, infile);          // 读取一个字符长度的编码
            for(i=0;i<8;++i){          // 由根向下直至叶节点正向匹配编码对应字符
                if(code_temp & 128)
                    root = huf_tree[root].rchild;
                else
                    root = huf_tree[root].lchild;
                if(root < char_kinds){
                    fwrite((char *)&huf_tree[root].uch, sizeof(unsigned char), 1, outfile);
                    ++writen_len;
                    if (writen_len == file_len) break;        // 控制文件长度，跳出内层循环
                    root = node_num-1;          // 复位为根索引，匹配下一个字符
                }
                code_temp <<= 1;          // 将编码缓存的下一位移到最高位，供匹配
            }
            if (writen_len == file_len) break;        // 控制文件长度，跳出外层循环
        }
        fclose(infile);
        fclose(outfile);
        free(huf_tree);
    }
}
```

4. 测试验证

该软件程序包含文件如下：



其中 ss.c 文件为源文件，Makefile 为 make 文件。打开终端输入 make，生成 fblcd.out 为输出可执行文件。



输入 ./fblcd.out --version 查看版本及团队信息

```
chu@turtle:~/桌面/code$ ./fblcd.out --version
USAGE: ./fblcd.out [--version] [-{en,de,cp} infile outfile]
made by: ChuJiangheng
Registration name: sansan
Registration number: CICC1709
version: 1.0.0

Execution time:0.0s

chu@turtle:~/桌面/code$
```

接下来以测试用例中的 sample01.bmp 图片示例，该文件大小为 8110202 字节。

输入 ./fblcd.out -en sample01.bmp sample01.jlcd 进行压缩：

```
chu@turtle:~/桌面/code$ ./fblcd.out -en sample01.bmp sample01.jlcd
Compressing...

Execution time:5.0s
compressibility:2.69%

chu@turtle:~/桌面/code$
```

可以看到计算所得压缩率为 2.69%，压缩文件 sample01.jlcd 大小为 218539 字节

输入./fblcd.out -de sample01.jlcd sample01.jlcd.bmp 进行解压：

```
chu@turtle:~/桌面/code$ ./fblcd.out -de sample01.jlcd sample01.jlcd.bmp
Decompressing...

Execution time:5.0s

chu@turtle:~/桌面/code$
```

可以得到 sample01.jlcd.bmp 大小为 8110202 字节，同 sample01.bmp 文件大小完全一致。

最后输入./fblcd.out -cp sample01.bmp sample01.jlcd.bmp 对两者进行比较：

```
chu@turtle:~/桌面/code$ ./fblcd.out -cp sample01.bmp sample01.jlcd.bmp
the two files are same

Execution time:1.0s

chu@turtle:~/桌面/code$
```

可以看到两个文件完全一致，解压过程无损。