ESE 5320 Final Code Library
LZW.cpp:

```
#include "lzw.h"
//*****************************************************************************
**************************
#define CAPACITY 32768 // hash output is 15 bits, and we have 1 entry per bucket, so
capacity is 2^15
//#define CAPACITY 4096
// try  uncommenting the line above and commenting line 6 to make the hash table
smaller
// and see what happens to the number of entries in the assoc mem
// (make sure to also comment line 27 and uncomment line 28)

   unsigned int my_hash(unsigned long key)
{
   key &= 0xFFFFF; // make sure the key is only 20 bits

   unsigned int hashed = 0;

   for(int i = 0; i < 20; i++)
   {
       hashed += (key >> i)&0x01;
       hashed += hashed << 10;
       hashed ^= hashed >> 6;
   }
   hashed += hashed << 3;
   hashed ^= hashed >> 11;
   hashed += hashed << 15;
   return hashed & 0x7FFF;          // hash output is 15 bits
   //return hashed & 0xFFF;
}

void hash_lookup(unsigned long* hash_table, unsigned int key, bool* hit, unsigned int*
result)
{
   //std::cout << "hash_lookup():" << std::endl;
   key &= 0xFFFFF; // make sure key is only 20 bits

   unsigned long lookup = hash_table[my_hash(key)];

   // [valid][value][key]
   unsigned int stored_key = lookup&0xFFFFF;       // stored key is 20 bits
```

```
   unsigned int value = (lookup >> 20)&0xFFF;      // value is 12 bits
   unsigned int valid = (lookup >> (20 + 12))&0x1; // valid is 1 bit

   if(valid && (key == stored_key))
   {
       *hit = 1;
       *result = value;
       //std::cout << "\thit the hash" << std::endl;
       //std::cout << "\t(k,v,h) = " << key << " " << value << " " << my_hash(key) <<
std::endl;
   }
   else
   {
       *hit = 0;
       *result = 0;
       //std::cout << "\tmissed the hash" << std::endl;
   }
}

void hash_insert(unsigned long* hash_table, unsigned int key, unsigned int value,
bool* collision)
{
   //std::cout << "hash_insert():" << std::endl;
   key &= 0xFFFFF;   // make sure key is only 20 bits
   value &= 0xFFF;   // value is only 12 bits

   unsigned long lookup = hash_table[my_hash(key)];
   unsigned int valid = (lookup >> (20 + 12))&0x1;

   if(valid)
   {
       *collision = 1;
       //std::cout << "\tcollision in the hash" << std::endl;
   }
   else
   {
       hash_table[my_hash(key)] = (1UL << (20 + 12)) | (value << 20) | key;
       *collision = 0;
       //std::cout << "\tinserted into the hash table" << std::endl;
       //std::cout << "\t(k,v,h) = " << key << " " << value << " " << my_hash(key) <<
std::endl;
   }
```

```
}

// cast to struct and use ap types to pull out various feilds.

void assoc_insert(assoc_mem* mem,  unsigned int key, unsigned int value, bool*
collision)
{
   //std::cout << "assoc_insert():" << std::endl;
   key &= 0xFFFFF; // make sure key is only 20 bits
   value &= 0xFFF;   // value is only 12 bits

   if(mem->fill < 64)
   {
       mem->upper_key_mem[(key >> 18)%512] |= (1 << mem->fill);  // set the fill'th
bit to 1, while preserving everything else
       mem->middle_key_mem[(key >> 9)%512] |= (1 << mem->fill);  // set the fill'th
bit to 1, while preserving everything else
       mem->lower_key_mem[(key >> 0)%512] |= (1 << mem->fill);   // set the fill'th
bit to 1, while preserving everything else
       mem->value[mem->fill] = value;
       mem->fill++;
       *collision = 0;
       //std::cout << "\tinserted into the assoc mem" << std::endl;
       //std::cout << "\t(k,v) = " << key << " " << value << std::endl;
   }
   else
   {
       *collision = 1;
       //std::cout << "\tcollision in the assoc mem" << std::endl;
   }
}

void assoc_lookup(assoc_mem* mem, unsigned int key, bool* hit, unsigned int* result)
{
   //std::cout << "assoc_lookup():" << std::endl;
   key &= 0xFFFFF; // make sure key is only 20 bits

   unsigned int match_high = mem->upper_key_mem[(key >> 18)%512];
   unsigned int match_middle = mem->middle_key_mem[(key >> 9)%512];
   unsigned int match_low  = mem->lower_key_mem[(key >> 0)%512];

   unsigned int match = match_high & match_middle & match_low;
```

```
   unsigned int address = 0;
   for(; address < 64; address++)
   {
       if((match >> address) & 0x1)
       {
           break;
       }
   }
   if(address != 64)
   {
       *result = mem->value[address];
       *hit = 1;
       //std::cout << "\thit the assoc" << std::endl;
       //std::cout << "\t(k,v) = " << key << " " << *result << std::endl;
   }
   else
   {
       *hit = 0;
       //std::cout << "\tmissed the assoc" << std::endl;
   }
}
//*****************************************************************************
**************************
void insert(unsigned long* hash_table, assoc_mem* mem, unsigned int key, unsigned int
value, bool* collision)
{
   hash_insert(hash_table, key, value, collision);
   if(*collision)
   {
       assoc_insert(mem, key, value, collision);
   }
}

void lookup(unsigned long* hash_table, assoc_mem* mem, unsigned int key, bool* hit,
unsigned int* result)
{
   hash_lookup(hash_table, key, hit, result);
   if(!*hit)
   {
       assoc_lookup(mem, key, hit, result);
   }
```

```
}

static void write_encoded_file(uint16_t* out_code, uint32_t out_len, uint32_t
&header){
   int total_bits = out_len * 12;
   int total_bytes = static_cast<int>(std::ceil(total_bits / 8.0));
   header = static_cast<uint32_t>(total_bytes & 0xFFFFFFFF) << 1;
   unsigned char* file_buffer = (unsigned char*)malloc(sizeof(unsigned char) *
(total_bytes + 4));

   int i = 0, j = 0;
   // file_buffer[j++] = static_cast<unsigned char>(header >> 24);
   // file_buffer[j++] = static_cast<unsigned char>((header >> 16) & 0xFF);
   // file_buffer[j++] = static_cast<unsigned char>((header >> 8) & 0xFF);
   // file_buffer[j++] = static_cast<unsigned char>(header & 0xFF);
   file_buffer[j++] = static_cast<unsigned char>(header & 0xFF);
   file_buffer[j++] = static_cast<unsigned char>((header >> 8) & 0xFF);
   file_buffer[j++] = static_cast<unsigned char>((header >> 16) & 0xFF);
   file_buffer[j++] = static_cast<unsigned char>(header >> 24);
   for(i = 0; i + 1 < out_len; i += 2){
       file_buffer[j++] = static_cast<unsigned char>(out_code[i] >> 4);
       file_buffer[j++] = static_cast<unsigned char>(((out_code[i] << 4) & 0xF0) |
((out_code[i + 1] >> 8) & 0x0F));
       file_buffer[j++] = static_cast<unsigned char>(out_code[i + 1] & 0xFF);
   }
   if(i != out_len){
       file_buffer[j++] = static_cast<unsigned char>(out_code[i] >> 4);
       file_buffer[j++] = static_cast<unsigned char>((out_code[i] << 4) & 0xF0);
   }

   std::ofstream outfile("encoded_data.bin", std::ios::binary);
   if (!outfile.is_open()) {
       std::cerr << "Could not open the file for writing.\n";
       return;
   }

   // Write the data to the file
   outfile.write(reinterpret_cast<const char*>(file_buffer), total_bytes + 4);

   // Check for write errors
   if (!outfile.good()) {
       std::cerr << "Error occurred while writing to the file.\n";
```

```
    }

    // Close the file
    outfile.close();
}

//*************************************************************************************
**************************
void hardware_encoding(unsigned char* s1, int length, uint16_t* out_code, uint32_t
&header, int &out_len)
{
    // create hash table and assoc mem
    unsigned long hash_table[CAPACITY];
    assoc_mem my_assoc_mem;

    // make sure the memories are clear
    for(int i = 0; i < CAPACITY; i++)
    {
        hash_table[i] = 0;
    }
    my_assoc_mem.fill = 0;
    for(int i = 0; i < 512; i++)
    {
        my_assoc_mem.upper_key_mem[i] = 0;
        my_assoc_mem.lower_key_mem[i] = 0;
    }

    // init the memories with the first 256 codes
    for(unsigned long i = 0; i < 256; i++)
    {
        bool collision = 0;
        unsigned int key = (i << 8) + 0UL; // lower 8 bits are the next char, the upper
bits are the prefix code
        insert(hash_table, &my_assoc_mem, key, i, &collision);
    }
    int next_code = 256;


    int prefix_code = s1[0];
    unsigned int code = 0;
    char next_char = 0;
```

```
   int i = 0, j = 0;
   while(i < length)
   {
       if(i + 1 == length)
       {
           std::cout << prefix_code;
           std::cout << "\n";
           // i++;
           break;
       }
       next_char = s1[i + 1];

       bool hit = 0;
       //std::cout << "prefix_code " << prefix_code << " next_char " << next_char <<
std::endl;
       lookup(hash_table, &my_assoc_mem, (prefix_code << 8) + next_char, &hit, &code);
       if(!hit)
       {
           std::cout << prefix_code;
           out_code[j++] = prefix_code;
           // out_code[i]=prefix_code;
           std::cout << "\n";

           bool collision = 0;
           insert(hash_table, &my_assoc_mem, (prefix_code << 8) + next_char,
next_code, &collision);
           if(collision)
           {
               std::cout << "ERROR: FAILED TO INSERT! NO MORE ROOM IN ASSOC MEM!" <<
std::endl;
               return;
           }
           next_code += 1;

           prefix_code = next_char;
       }
       else
       {
           prefix_code = code;
       }
       i += 1;
   }
```

```
   out_len = j;
   write_encoded_file(out_code, out_len, header);

   // header = static_cast<uint32_t>(out_len) << 1;

   std::cout << std::endl << "assoc mem entry count: " << my_assoc_mem.fill <<
std::endl;

   // std::ofstream outfile("encoded_data.bin", std::ios::binary);
   // if (!outfile) {
   //     std::cerr << "Could not open the file for writing." << std::endl;
   //     return;
   // }
   // outfile.write(reinterpret_cast<const char*>(&header), sizeof(header));
   // for (int i = 0; i < out_len; ++i) {
   //     outfile.write(reinterpret_cast<const char*>(&out_code[i]),
sizeof(uint16_t));
   // }
   // outfile.close();
}
//*****************************************************************************
**************************
std::vector<int> encoding(std::string s1)
{
   std::cout << "Encoding\n";
   std::unordered_map<std::string, int> table;
   for (int i = 0; i <= 255; i++) {
       std::string ch = "";
       ch += char(i);
       table[ch] = i;
   }
   std::string p = "", c = "";
   p += s1[0];
   int code = 256;
   std::vector<int> output_code;
   std::cout << "String\tOutput_Code\tAddition\n";
   for (int i = 0; i < s1.length(); i++) {
       if (i != s1.length() - 1)
           c += s1[i + 1];
       if (table.find(p + c) != table.end()) {
           p = p + c;
       }
```

```
        else {
            std::cout << p << "\t" << table[p] << "\t\t"
                 << p + c << "\t" << code << std::endl;
            output_code.push_back(table[p]);
            table[p + c] = code;
            code++;
            p = c;
        }
        c = "";
    }
    std::cout << p << "\t" << table[p] << std::endl;
    output_code.push_back(table[p]);
    return output_code;
}

void decoding(std::vector<int> op)
{
    std::cout << "\nDecoding\n";
    std::unordered_map<int, std::string> table;
    for (int i = 0; i <= 255; i++) {
        std::string ch = "";
        ch += char(i);
        table[i] = ch;
    }
    int old = op[0], n;
    std::string s = table[old];
    std::string c = "";
    c += s[0];
    std::cout << s;
    int count = 256;
    for (int i = 0; i < op.size() - 1; i++) {
        n = op[i + 1];
        if (table.find(n) == table.end()) {
            s = table[old];
            s = s + c;
        }
        else {
            s = table[n];
        }
        std::cout << s;
        c = "";
        c += s[0];
```

```
        table[count] = table[old] + c;
        count++;
        old = n;
    }
}
//*****************************************************************************
**************************
int main()
{

    std::string s = "WYS*WYGWYS*WYSWYSG";
    // std::cout << "Our message is: " << s << std::endl << std::endl;
    // std::cout << "Running the software compression we get: " << std::endl;
    std::vector<int> output_code = encoding(s);
    // std::cout << "The compressed output stream is: ";
    for (int i = 0; i < output_code.size(); i++) {
        std::cout << output_code[i] << " ";
    }
    std::cout << std::endl << std::endl;

    std::cout << "Running the hardware version we get " << std::endl;
    std::cout << "The compressed output stream is: " << std::endl;
    unsigned char s1[] = "WYS*WYGWYS*WYSWYSG";
    uint16_t out_code[20];
    uint32_t header;
    int out_len;
    hardware_encoding(s1,20,out_code, header, out_len);
    std::cout << "The compressed output stream is: " << std::endl;
    for (int i = 0; i < out_len; ++i) {
        std::cout << "Pointer " << i << ": " << out_code[i]
                  << ", Value: " << (out_code[i]) << std::endl;
    }
    return 0;
}
```

encoder.cpp:

```
#include "encoder.h"
#include "cdc.h"
#include "sha.h"
#include "lzw.h"
#include "utils.h"
#include <stdio.h>
#include <stdint.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include "server.h"
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unordered_map>
#include <vector>
#include "stopwatch.h"

#define NUM_PACKETS 8
#define pipe_depth 4
#define DONE_BIT_L (1 << 7)
#define DONE_BIT_H (1 << 15)

int offset = 0;
unsigned char* file;

void handle_input(int argc, char* argv[], int* blocksize) {
   int x;
   extern char *optarg;

   while ((x = getopt(argc, argv, ":b:")) != -1) {
       switch (x) {
       case 'b':
           *blocksize = atoi(optarg);
           printf("blocksize is set to %d optarg\n", *blocksize);
           break;
       case ':':
           printf("-%c without parameter\n", optopt);
           break;
       }
   }
}

int main(int argc, char* argv[]) {
   std::cout << "11:05am" << std::endl;
```

```
std::cout << argv[1] << std::endl;
stopwatch ethernet_timer;
unsigned char* input[NUM_PACKETS];
int writer = 0;
int done = 0;
int length = 0;
int count = 0;
ESE532_Server server;

// default is 2k
int blocksize = BLOCKSIZE;

// set blocksize if decalred through command line
handle_input(argc, argv, &blocksize);

file = (unsigned char*) malloc(sizeof(unsigned char) * 70000000);
if (file == NULL) {
    printf("help\n");
}

for (int i = 0; i < NUM_PACKETS; i++) {
    input[i] = (unsigned char*) malloc(
            sizeof(unsigned char) * (NUM_ELEMENTS + HEADER));
    if (input[i] == NULL) {
        std::cout << "aborting " << std::endl;
        return 1;
    }
}

server.setup_server(blocksize);

writer = pipe_depth;
server.get_packet(input[writer]);
count++;

// get packet
unsigned char* buffer = input[writer];

// decode
done = buffer[1] & DONE_BIT_L;
length = buffer[0] | (buffer[1] << 8);
length &= ~DONE_BIT_H;
```

```
// printing takes time so be weary of transfer rate
//printf("length: %d offset %d\n",length,offset);

// we are just memcpy'ing here, but you should call your
// top function here.
memcpy(&file[offset], &buffer[HEADER], length);

offset += length;
writer++;

std::unordered_map<std::string, int> chunks_map;
int sum_raw_length = 0, sum_lzw_cmprs_len = 0;

//last message
while (!done) {
    // reset ring buffer
    if (writer == NUM_PACKETS) {
        writer = 0;
    }

    ethernet_timer.start();
    server.get_packet(input[writer]);
    ethernet_timer.stop();

    count++;

    // get packet
    unsigned char* buffer = input[writer];

    // decode
    done = buffer[1] & DONE_BIT_L;
    length = buffer[0] | (buffer[1] << 8);
    length &= ~DONE_BIT_H;
    //printf("length: %d offset %d\n",length,offset);
    memcpy(&file[offset], &buffer[HEADER], length);

    offset += length;
    writer++;

    // encode the obtained information in buffer

    // initialize the vector to store the obtained chunks
```

```
        std::vector<std::string> chunks;
        // get the chunked result
        cdc(buffer, chunks, NUM_ELEMENTS + HEADER);

        //calculate hash value and chunk id for each chunk
        //add those key-value pairs to chunks map
        //Question: do we need to consider the situation that different chunks share
the same hash value calculated by SHA at this point
        for(std::vector<std::string>::size_type i = 0; i < chunks.size(); i++){
        // for(std::vector<std::string>::size_type i = 0; i < 10; i++){
            hash_part hash_value;
            sha(chunks[i], hash_value);
            std::string hash_hex_string = toHexString(hash_value);

            if(chunks_map.find(hash_hex_string) == chunks_map.end()){
                chunks_map.insert({hash_hex_string, i});
                unsigned char* chunk_content = (unsigned char*)malloc(sizeof(unsigned
char) * (chunks[i].length() + 1));
                convert_string_char(chunks[i], chunk_content);
                uint32_t header;
                uint16_t* out_code = (uint16_t*)malloc(sizeof(uint16_t) *
chunks[i].length() + 32);
                int out_len;
                hardware_encoding(chunk_content, chunks[i].length(), out_code, header,
out_len, argv[1]);
                sum_raw_length += chunks[i].length();
                sum_lzw_cmprs_len += out_len;
                free(out_code);
                free(chunk_content);
            }

            else{

            }
        }
    }
    float lzw_compress_ratio = sum_raw_length / sum_lzw_cmprs_len;
    std::cout << "LZW compress ratio: " << lzw_compress_ratio << std::endl;


    // write file to root and you can use diff tool on board
    FILE *outfd = fopen("output_cpu.bin", "wb");
```

```
    int bytes_written = fwrite(&file[0], 1, offset, outfd);
    printf("write file with %d\n", bytes_written);
    fclose(outfd);

    for (int i = 0; i < NUM_PACKETS; i++) {
        free(input[i]);
    }

    free(file);
    std::cout << "--------------- Key Throughputs ---------------" << std::endl;
    float ethernet_latency = ethernet_timer.latency() / 1000.0;
    float input_throughput = (bytes_written * 8 / 1000000.0) / ethernet_latency; //
Mb/s
    std::cout << "Input Throughput to Encoder: " << input_throughput << " Mb/s."
            << " (Latency: " << ethernet_latency << "s)." << std::endl;

    return 0;
}
```