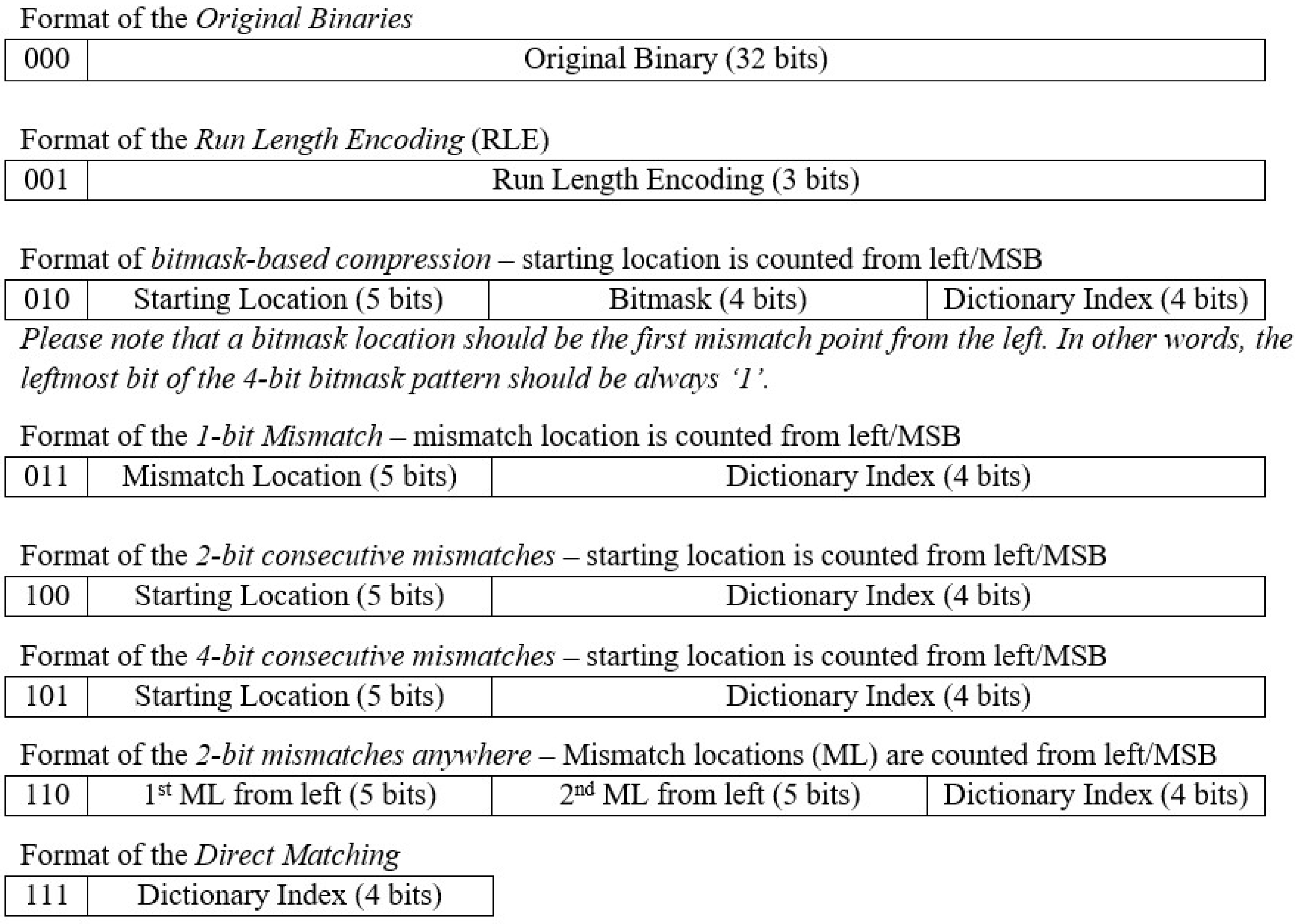
EN3240: Embedded Systems Engineering
Assignment 5 — Programming: Code Compression

May 13, 2021

This is an individual assignment!
Due Date: 6 June 2021 by 11.59 PM

Instructions

In this project, you need to **implement both code compression and decompression using C, C++ or Java**. Assume that the dictionary can have sixteen entries (index 4 bits) and the sixteen entries are selected based on frequency (the most frequent instruction should have index 0000). If two entries have the same frequency, priority is given to the one that appears first in the original program order. The original code consists of 32-bit binaries. You are allowed to use only eight possible formats for compression (as outlined below). Note that if one entry (32-bit binary) can be compressed in more than one way, choose the most beneficial one i.e., the one that provides the shortest compressed pattern. If two formats produce exactly the same compression, choose the one that appears earlier in the following listing (e.g., run-length encoding appears earlier than direct matching). If a 32-bit binary can be compressed using multiple dictionary entries by any specific compression format (e.g., bitmask-based compression), please use the dictionary entry with the smallest index value. Please count the starting location of a mismatch from the leftmost (MSB) bit of the pattern – the position of the leftmost bit is 00000.



Format of the *Original Binaries*

000	Original Binary (32 bits)

Format of the *Run Length Encoding* (RLE)

001	Run Length Encoding (3 bits)

Format of *bitmask-based compression* – starting location is counted from left/MSB

010	Starting Location (5 bits)	Bitmask (4 bits)	Dictionary Index (4 bits)

Please note that a bitmask location should be the first mismatch point from the left. In other words, the leftmost bit of the 4-bit bitmask pattern should be always '1'.

Format of the *1-bit Mismatch* – mismatch location is counted from left/MSB

011	Mismatch Location (5 bits)	Dictionary Index (4 bits)

Format of the *2-bit consecutive mismatches* – starting location is counted from left/MSB

100	Starting Location (5 bits)	Dictionary Index (4 bits)

Format of the *4-bit consecutive mismatches* – starting location is counted from left/MSB

101	Starting Location (5 bits)	Dictionary Index (4 bits)

Format of the *2-bit mismatches anywhere* – Mismatch locations (ML) are counted from left/MSB

110	1st ML from left (5 bits)	2nd ML from left (5 bits)	Dictionary Index (4 bits)

Format of the *Direct Matching*

111	Dictionary Index (4 bits)

Figure 1: Compression formats.

Run-Length Encoding (RLE) can be used when there is consecutive repetition of the same instruction. The first instruction of the repeated sequence will be compressed (or kept uncompressed if it is not part of the dictionary) as usual. The remaining ones will be compressed using RLE format shown above. The three bits in the RLE indicates the number of occurrences (000, 001, 010, 011, 100, 101, 110 and 111 imply 1, 2, 3, 4, 5, 6, 7 and 8 occurrences, respectively), excluding the first one. A single application of RLE can encode up to 8 instructions. In other words, up to 9 repetitions can be covered by RLE (first one non-RLE compression followed by up to 8 RLE compression). If you have more than 9 repetitions, you can apply RLE multiple times as long as it is profitable compared to other available options. While multiple combinations are possible, please cover 9 repetitions followed by the remaining ones. For example, if you have 21 repetitions, you should apply RLE three times as $< 1 + 8 > + < 1 + 8 > + < 1 + 2 >$ (instead of $< 1 + 6 > + < 1 + 6 > + < 1 + 6 >$, or other possible compositions).

You need to show a working prototype that will take any 32-bit binary (0/1 text) file and compress it to produce a output file that shows compressed patterns arranged in a sequential manner (32-bit in each line, last line padded with 0's, if needed), a separation marker "xxxx", followed by sixteen dictionary entries. Your program should also be able to accept a compressed file (in the above format) and decompress to generate the decompressed (original) patterns. Please see the sample files posted in the webpage.

Command Line and Input/Output Formats:

The simulator should be executed with the following command line. Please use parameters "1" and "2" to indicate compression, and decompression, respectively.
./SIM 1 (or **java SIM 1**) for compression
./SIM 2 (or **java SIM 2**) for decompression

Please hardcode the input and output files as follows:

1. Input file for your compression function: **original.txt**
2. Output produced by your compression function: **cout.txt**
3. Input file for your decompression function: **compressed.txt**
4. Output produced by your decompression function: **dout.txt**

Submission Policy:

Please follow the submission policy outlined below. There will be up to 10% score penalty based on the nature of submission policy violations.

1. Please submit only one source file that includes both compression and decompression. Please add ".txt" at the end of your filename. Your file name must be SIM (e.g., SIM.c.txt or SIM.cpp.txt or SIM.java.txt).

2. Please test your submission. These are the exact steps that we will follow to grade your submission.

   - Download your submission from Moodle (ensures your upload was successful)
   - Remove ".txt" extension (e.g., SIM.c.txt should be renamed to SIM.c)
   - Compile to produce an executable named SIM.
     - gcc SIM.c –o SIM or
     - javac SIM.java or
     - g++ SIM.cpp –o SIM or
     - g++ -std=c++0x SIM.cpp -o SIM
   - Please do not print anything on screen.
   - Assume hardcoded input/output files as outlined in the project description.
   - Compress the input file (original.txt) and check with the expected output (compressed.txt)
     - ./SIM 1 (or java SIM 1)
     - diff –w –B cout.txt compressed.txt
   - Decompress the input file (compressed.txt) and check with the expected output (original.txt)
     - ./SIM 2 (or java SIM 2)
     - diff –w –B dout.txt original.txt

3. If you do not follow these instructions, there can be cases where output format is different, filename is different, command line arguments are different, or Moodle submission is missing. All of these lead to un-necessary waste of time for TA, instructor and students. **Please use the exactly same commands as outlined above to avoid 10% score penalty.**

4. You are not allowed to take or give any help in completing this project.

Grading Policy

The Moodle page has the sample input and output files. Correct handling of the sample input will be used to determine 60% of credit awarded. The remaining 40% will be determined from other input test cases that you will not have access prior to grading. The other test cases can have different types and number of 32-bit binaries (0/1 text). It is recommended that you construct your own sample input files with which to further test your compression and decompression functions. You can assume that we will use less than 1024 32-bit binary (0/1 text) patterns in the new test file. Please note that the new test case will NOT test any exceptional scenarios that are not described in this document.