CSC 236 Programming Assignment

RLC

Summary

The RLC assignment is worth 6% of your grade. For this assignment, you will be submitting a file named **rlc.ans**. This file is created by the grading system when you grade your program. The grade is based on correct answers, documentation and the number of executable instructions used and the number of instructions actually executed during testing.

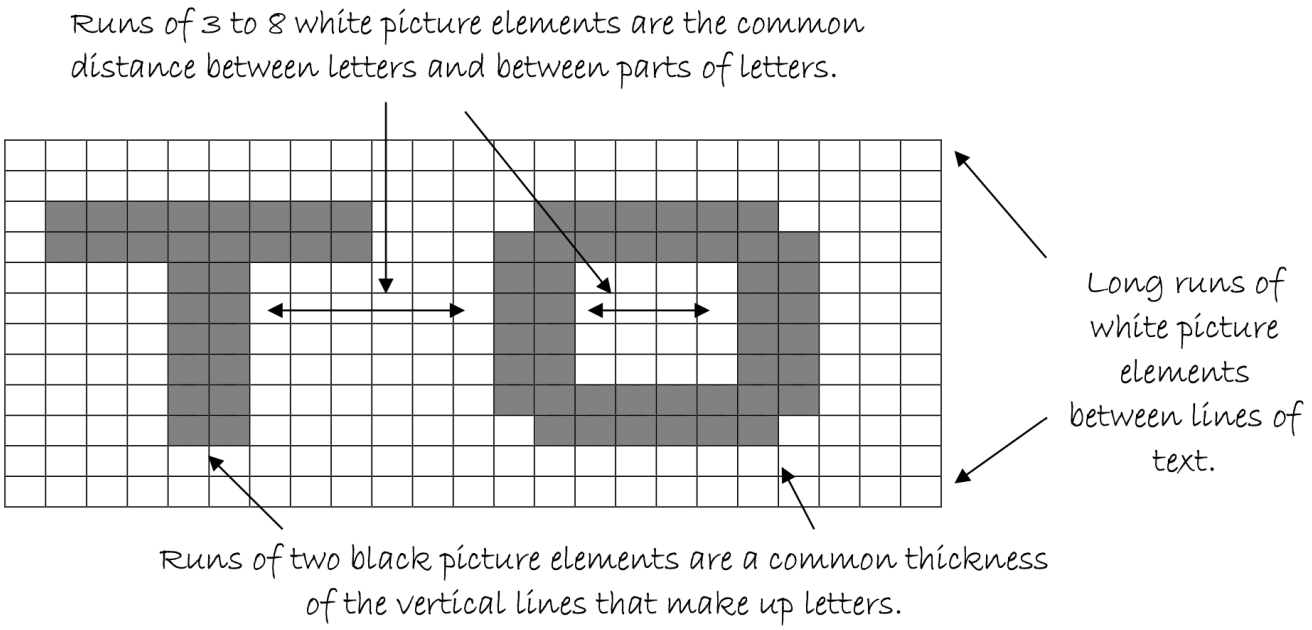
You can complete this assignment working in a team of two, if you'd like. If you work with a partner, create a single **rlc.ans** file. Both team members should submit a copy of this file to the RLC assignment in Moodle.

If you have questions about the assignment, feel free to use our EdStem forum, but be sure not to post any code from your solution in a public edstem post (even code that's not working). You can also stop by during office hours to get help from anyone on the teaching staff. Remember, if you need help, you need to make sure your code is documented, including block comments above sections of your code and line comments saying what you are trying to do with each instruction. We'll also expect you to have a design for the program that you're trying to follow.

Run-Length Coding

Lots of applications use black-and-white mages, facsimile machines, line drawings, some simple graphs, etc. In this kind of image, we can think of rows of the image as consisting of alternating runs of black pixels and white pixels, i.e., a sequence of consecutive white pixels, then a sequence of black pixels, then back to white, and so on. In an image like this, we may be able to save some storage space using Run-Length Coding (RLC). RLC is data compression technique where runs are stored as a single data value (e.g., a pixel color) and a count of the number of consecutive occurrences there are of that color.

The example image below shows how this works. Imagine that this is a tiny part of a larger image containing some text:



The CCITT (International Telegraph and Telephone Consultive Committee) Study Group XIV publishes a standard for one-dimensional RLC data compression. That standard is very comprehensive and would require more code than we can reasonably expect for a CSC236 assignment. Instead, you will implement a less complicated version. One significant simplification is that our picture elements will be bytes instead of bits. This will allow us to use the standard DOS character display to see the output. However, these simplifications will still maintain all the important concepts associated with one-dimensional RLC data compression.

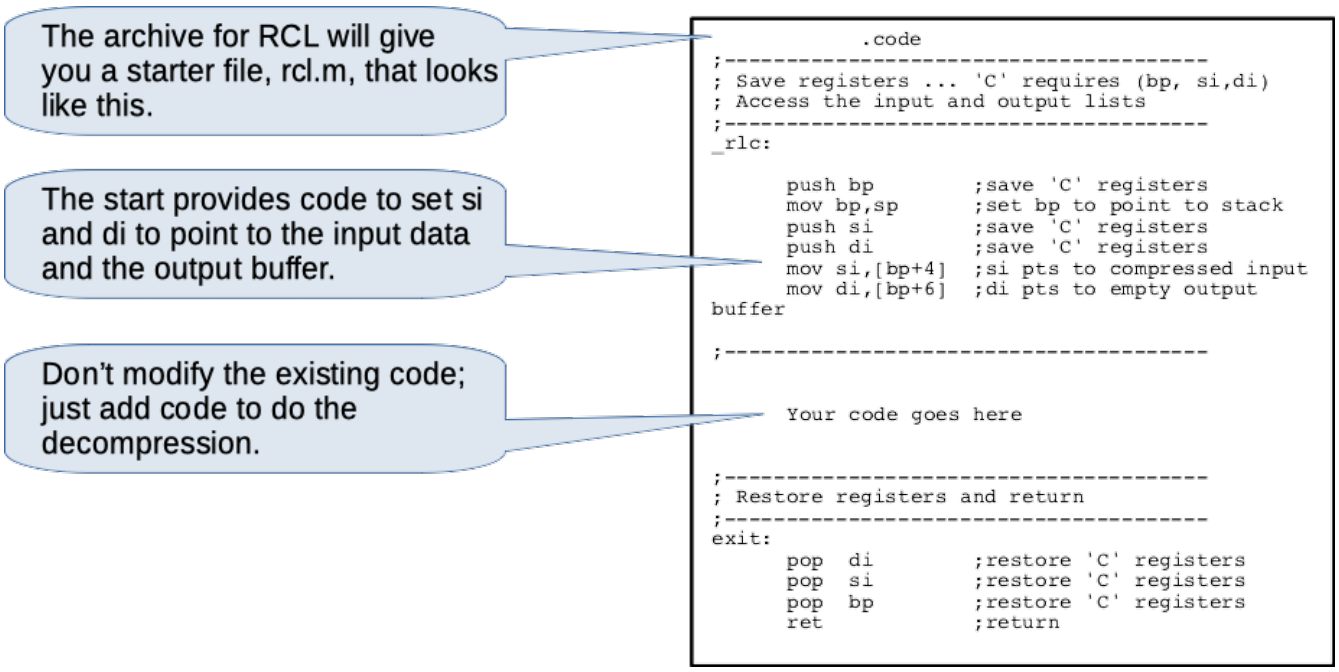
You will write a serially reusable subroutine named _rlc to decompress data that has been compressed into an RLC representation. Your subroutine will be linked with a C main driver program. It will need to follow all the calling conventions for programming a subroutine that can be called from C. Check the class notes for details about what is required for this.

The RLC subroutine

Your source file must be named **rlc.asm**, and your assembly language subroutine must be named _rlc (following the C naming conventions by putting an underscore in front).

The input parameters passed to rlc are two pointers:

- A pointer to the compressed data which consists of strings of runs and features described below.
- A pointer to a buffer, into which your subroutine will store decompressed data. In this buffer, you will the original character array based on its compressed representaiton.



To decompress a line of the RLC-compressed image, you will need to:

- Scan the input compressed data.
- Recognize the bit pattern for a feature or run length code.
- Place the corresponding decompressed run of picture elements into the output buffer.
- Continue until the end of data feature is located.

For the output:

- The decompressed data is placed into the output buffer.
- The C registers (bp, si, di, ss, ds) must restored to the original value they had upon input to RLC.

Compression algorithm

Compressed information is encoded as a bit pattern that represents either a feature or a *run_length_code* (*rlc*). These terms come from the CCITT standard which supports many more variations of features and rlcs.

Run length codes are each 4 bits. Using 4 bits each lets us pack two run length codes per byte.

4-bit rlc	**Meaning**
0000	Build a run of length 0 of the current color.
0001	Build a run of length 1 of the current color.
0010	Build a run of length 2 of the current color.
0011	Build a run of length 3 of the current color.
0100	Build a run of length 4 of the current color.

0101	Build a run of length 5 of the current color.
0110	Build a run of length 6 of the current color.
0111	Build a run of length 7 of the current color.
1000	Build a run of length 8 of the current color.
1001	Build a run of length 9 of the current color.
1010	Build a run of length 10 of the current color.
1011	Build a run of length 11 of the current color.
1100	Build a run of length 12 of the current color.
1101	Build a run of length 13 of the current color.
1110	Build a run of length 14 of the current color.
1111	Build a run of the current color that goes the rest of the way to the end of the current 80-byte line.

In the compressed data, if a byte has a value of zero (i.e., both 4-bit codes are zero), that indicates the end of the data stream; no codes follow.

Byte value (8 bits)	**Meaning**
0000 0000	You have reached the end of the compressed input data. Return to the caller.

For the decompression algorithm

- The length of a line is 80 bytes ... or 80 picture elements.
- Each output picture element (pel) is one byte. White picture elements should be set to 20h and black picture elements should be set to Dbh. The only items you put in the output buffer should be white pels (20h) or black pels (Dbh).
- The first run on a line is assumed to be the color white.
- Runs alternate: white, black, white, black, etc. until the end of a line is reached. The last run on the line can be either white or black.
- The compressed data passed to RLC will be valid. No error checking is needed.
  - There will never be a run that takes you past the end of a line (crosses the end of a line boundary).
  - The decompression buffer will be large enough to hold all decompressed data created by RLC.
- The last run on every line will always use the RLC = 1111 When you encounter this RLC you should fill all the remaining pels on the line with the current pel color. Remember that a line is 80 bytes.
- The decompression code always processes an integral number of lines. It will not know exactly how many lines it will decompress. It decompresses lines of data until it encounters the 00h feature code in the compressed input data buffer.
- The initial value of the contents of the output buffer into which RLC decompresses the data is unknown. You should think of this as uninitialized memory.

Observations about compression

- The vast majority of the compression comes from the long runs that go to the end of a line.
- Why do we need a run length code of zero? There are two reasons.
  1. The first run on a line is always assumed to be white. If it turns out that the first run on the line is black then the sender needs to send a white run of zero.
  2. If a run of picture elements is longer than 14 (for example a run of 20 black picture elements) then the sender needs to send a run of 14 black picture elements, then 0 white picture elements, then 6 black picture elements.

Sample data for a one dimensional run

This is what the data would like for just the letter T. In hex it consists of these 26 runs:

```
F0 6F 06 F2 2F 22 F2 2F 22 F2 2F 22 FF
```

Under decompression, this should produce the following:

F = white run to end of line	
0 = white run of zero 6 = black run of 6 F = white run to eol	■■■■■■
0 = white run of zero 6 = black run of 6 F = white run to eol	■■■■■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
2 = white run of 2 2 = black run of 2 F = white run to eol	■■
F = white run to end of line	

Steps To Complete This Program

Step 1. Create a design

This assignment provides, for your optional use, one version of the logic for RLC. It is not optimized, but does function correctly. Thus getting the assembler code to work should be straightforward. The challenge will be achieving an efficient solution.

You may use any 8086 instruction in your solution.

- To earn any efficiency points for instructions executed you will need to use the lodsb and rep stosb string instructions to load input compressed data and store output decompressed pel values. To use the string instructions you must correctly initialize certain registers and the Direction Flag (see Class Notes chapter 16A).

- There are other 8086 instructions that may help with achieving an efficient solution. Check the Class Notes chapter 6.

A safe approach would be to implement the logic provided and when that works modify it to improve performance.

Step 2. Code your solution

Retrieve the testing and grading files. All files are packed together in one self-extracting file named **unpack.exe**. You can download a zipfile containing unpack.exe from the course Moodle page.

Put the unpack.exe file in the RLC subdirectory of your shared do DOSBox directory. Start up DOSBox, mount your e drive as usual, run the dbset command then switch to the RLC subdirectory. If you execute unpack.exe, it should unpack all the grading and testing files you need.

One of the files is a model for your subroutine. It's named **rlc.m**. This file illustrates how a main program and subroutine are linked together. Rename **rlc.m** to **rlc.asm** and then add your code to that file.

One of the files is the driver program for your subroutine.

- It is named rlcdrvr.obj and will be used for testing and grading your subroutine.
- Assemble and link your subroutines with the test driver. You should be able to compile your edited source file by running:

```
ml /c /Zi /Fl rlc.asm
```

Then, you should be able to link your object file with the rcl driver program using the following command

```
link /CO rlcdrvr.obj rlc.obj
```

Step 3. Test and debug your solution

The provided test driver is a C program. It does a few things to exercise your subroutine:

- It has compressed representations fo black and white runs of pixels.
- It calls your code to decompress the data runs back into a character array
- It displays and verifies the decompressed data created by your code

The driver has three built-in test cases. To run a test, type the following, where *n* is 1, 2 or 3.

```
testrlc n
```

After your code works for these three tests then it is ready to grade.

If you wish to run your code under the CodeView debugger, there are separate instructions on the class web site for using a simplified test driver program ... **testdrvr.asm**.

Step 4. Grading

The grading program will use the same executable file, **rlcdrvr.exe**, that you used when testing yoru subroutine. Type the following DOS command. This will run the grading program against your code:

```
gradrlc
```

Once RLC is working correctly, you may make additional grading runs to improve efficiency. If you execute the following command, then all subsequent grading runs will be marked as only being done to improve efficiency:

```
testrlc mark
```

Your final grade will be based on:

- 40 points for getting the correct answers
- 20 points for the number of executable instructions written to correctly complete RLC.
- 20 points for the number of instructions executed in the run that had the correct answers.
- 20 points for documentation of a program that functions correctly.

Efficiency and documentation are only a concern after the code works correctly, so efficiency and documentation grading will only happen once your code passes the functional tests.

Step 5. Submit your assignment

The grading program should create a file with the following name. Electronically submit a copy of this file to the Moodle assignment for the RCL program:

rlc.ans

The **rcl.ans** file contains two concatenated files. First the results file and then your source file **rlc.asm**.

Before you submit, you need to make sure that your **rcl.ans** file contains the line:
++ Grade ++ nnn = Total grade generated by the Grading System.

Incorrect electronic submissions result in your program not being graded. Be sure not to edit or modify any of the grading system files. This is especially important for the files named **results** and **rlc.ans**. Any modification of grading system files will look like an academic integrity violation.

```
//*********************************************************************
// This is a version of the RLC subroutine written in C.
// It is not optimized, but it does function correctly.
//*********************************************************************
// Input are pointers to the compressed data and uncompressed output buffer
// *comp points to the compressed data *dcomp points to output buffer
int rlc (unsigned char *comp, unsigned char *dcomp)
   {
   unsigned char wh=32, bl=219;                  // define white and black
   // run   = each input byte holds two runs
   //         run[0] is the length of the first run
   //         run[1] is the length of the second run
   // code  = current compressed input byte
   // cur   = current color to output
   unsigned char run[2], code, cur;             // define variables

   unsigned pels_left, len, i;                  // pels_left = pels left in the current line
                                                // len = number of pels in the current run

   pels_left = 80;                              // there are 80 pels on a line
   cur       = wh;                              // the first output color is white

   while( 1 )                                   // process all input runs
   {
      code = *comp;                             // code gets the current input byte
      comp++;                                   // advance the input pointer
      if (code == 0) return(0);                 // a byte of 00h signals end of data

      // each input byte holds two run lengths
      run[0] = (unsigned char)(code >> 4);      // run[0] = left 4 bits of input
      run[1] = (unsigned char)(code & 0x0f);    // run[1] = right 4 bits of input

      for (i=0;i<2;i++)                         // loop for each of the 2 runs
         {                                      // process a run

         if (pels_left == 0)                    // if at the end of a line then
            {                                   // start a new line
            pels_left = 80;                     // 80 pels to fill on that new line
            cur       = wh;                     // first color is white
            }

         if (run[i] == 15) len = pels_left;     // a run of 15 means go to end of line
         else              len = run[i];        // else we have a length 0-14

         while (len > 0)                        // output the run
            {                                   //
            *dcomp = cur;                       // output a pel of the current color
            dcomp++;                            // advance the output pointer
            len--;                              // reduce # of pels to still to output
            pels_left--;                        // reduce # of pels avail on line
            }

         if (cur == wh) cur = bl; else cur = wh;// swap the output pel color
         } // end loop for 2 runs
      }    // end loop to process all input runs
   }       // end the subroutine
```