RLC (Run Length Coding)

<u>Summary of the RLC project. The RLC project is worth 5% of the course grade.</u>

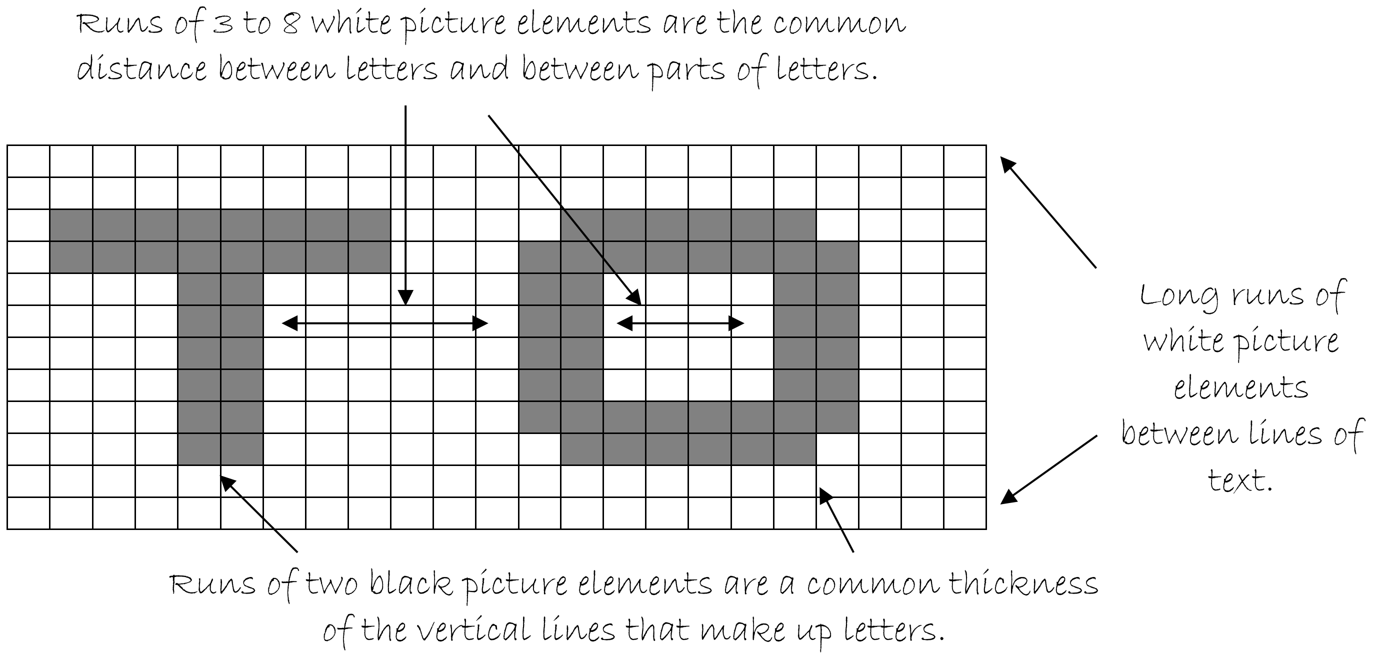
There is 1 submission:

- *rlc.ans* is worth 5% of the course grade. Grade based on correct answers, documentation, instructions written and executed.
- .See the calendar for the due date.
- If you need help from the staff:
  - submit your assembler source code to the ASK4HELP locker
  - submit the output file created when you ran the test using the *testcomp* procedure
  - send a note to the staff stating you have submitted code to ASK4HELP and explain the problem to be addressed
  - Assure your code is fully documented with header blocks and line comments. The staff may request to see your design.

<u>If working on a team … create a single rlc.ans file</u>
<u>that is submitted by both members of the team to their respective submit lockers.</u>

There are examples of data that consist of just runs of black and white picture elements: facsimile machine data; line drawings; simple graphics. Run-length coding (RLC) is a form of data compression for such data types in which runs of data are stored as a single data value (black or white) and a count specifying the run length.

In the example below, we have two letters and you can see many of runs of picture elements.


Runs of 3 to 8 white picture elements are the common distance between letters and between parts of letters.
Long runs of white picture elements between lines of text.
Runs of two black picture elements are a common thickness of the vertical lines that make up letters.

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 is warranted 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 concepts associated with one-dimensional RLC data compression.

You will write a *serially reusable* subroutine named *_rlc* to decompress data that has been compressed by my compression software. Your subroutine will be linked with a C main driver program. It will need to observe all the conventions for programming with mixed languages (C and assembler) as given in the class notes.

The RLC subroutine

Your source file must be named *rlc.asm*   Your assembler routine must be named *_rlc*

Input to RLC
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 RLC is to decompress the data to recreate the original character array.

The C calling sequence to rlc is *rlc (inlist, outlist);*

This is rlc.m which is the model for your code.
It will be unpacked with the other grading files.
It has the code to load si and di with the pointers to the input data and output buffer.
Do not modify the code or directives; just add your code to decompress the data.

```
         .code
;---------------------------------------
; Save registers ... 'C' requires (bp, si,di)
; Access the input and output lists
;---------------------------------------
_rlc:                                  ;
      push bp         ;save 'C' registers
      mov  bp,sp      ;set bp to point to stack
      push si         ;save 'C' registers
      push di         ;save 'C' registers
      mov  si,[bp+4]  ;si pts to input compressed data
      mov  di,[bp+6]  ;di pts to empty output buffer
;---------------------------------------

      Your code goes here

;---------------------------------------
; Restore registers and return
;---------------------------------------
exit:                                  ;
      pop  di         ;restore 'C' registers
      pop  si         ;restore 'C' registers
      pop  bp         ;restore 'C' registers
      ret             ;return
```

RLC processing

- 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.

Output from RLC

- 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. They are packed two run length codes per byte.

RLC – 4 bits	**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 to the end of the 80 byte line.

A compressed byte of data that is all zero signifies the end of the input stream.

Feature - 8 bits	**Meaning**
0000 0000	You have reached the end of the compressed input data. Return to the caller.

Rules for the decompression algorithm

- 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.
- Each output *p*icture *el*ement (pel) is one byte. White picture elements are 20h and black picture elements are DBh.
- The length of a line is 80 bytes ... or 80 picture elements.
- The only items you put in the output buffer are 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 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.

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
```

Which produce this image ....

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.

We provide, 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 straight forward. 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.

- Go to the class homepage. Click on *Lockers With Program Grading System Files*. Click on the RLC assignment.
- Download unpack.exe file to \P23X\RLC directory. Execute unpack.exe in DOSBox to create the testing/grading files.

One of the files is a model for your subroutine.

- For MASM it is named rlc.m. This file illustrates how a main program and subroutine are linked together.
- Rename rlc.m to be 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 using these commands.

`ml  /c  /Zi  /Fl  rlc.asm`	`link  /CO  rlcdrvr.obj  rlc.obj`

Step 3. Test and debug your solution.

I provide a C driver for your subroutine that:

- has compressed strings of black and white 1 byte picture element runs
- calls your code to decompress the data runs back into a character array
- displays and verifies the decompressed data created by your code

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

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*, which you have been testing.

Type the following DOS command: ***gradrlc***

This will run the grading program against your code.

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

The 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. Therefore, efficiency and documentation grading only occur after your code passes the functional tests.

Step 5. Submit your assignment

Electronically submit the file ***rlc.ans*** created by the grading system.

It is your responsibility to assure:

- It contains two concatenated files. First the *results* file and then your source file *rlc.asm*.
- It contains the line: ++ Grade ++ nnn = Total grade generated by the Grading System.

Incorrect electronic submissions result in your program not being graded and considered late.

Do not edit or modify any of the grading system files. This is especially true for the files named *results* and *rlc.ans.* Any modification of grading system files is considered cheating.

```
//************************************************************************
// 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
```