**LC-3 PSEUDO-OPS**

Some notes on the use of pseudo-ops in assembly language programming

The LC-3 assembly language has a number of "pseuo-operations" which are directions to the assembler to set up the conditions needed to run the actual program – kind of like the "pre-processor directives" of the C++ compiler like #include and so on.   
They are ***not*** Assembly Language instructions for the LC-3 itself.

In the LC-3 assembler, pseudo-ops are distinguished by an initial '.' – e.g .ORIG and .FILL

To understand the details of these pseudo-ops, we first need to review how the assembler works, and how the processor "runs" a program:

* The assembler translates each Assembly Language (AL) instruction into its Machine Language (ML) equivalent *(in the LC-3, that means a single 16-bit word)*. This a strictly one-to-one translation – each AL instruction is a single 16-bit ML instruction.  
  So a program is actually just a long list of ML instructions, usually followed by a block of data values.
* The **.ORIG** pseudo-op directs the assembler to mark the ML instructions as starting at the specified address *(we generally use x3000 - most memory below that is taken up by the BIOS)*.  
  Then, when the program is "run", an operating system routine called the "loader" loads the stored ML instructions into memory starting at that address.   
  Note that .ORIG does ***not*** result in any ML instruction itself - it is simply a directive to the assembler and loader programs.
* The final step is to copy the starting address of the ML code (x3000 in this example) into the microprocessor's PC (Program Counter).  
  At that point, the microprocessor fetches the instruction at that address, loads it into the Instruction Register (IR), and executes it. The PC is immediately incremented to point to the next instruction, and the process keeps on going until the HALT bios subroutine is executed.

**Now let's look at pseudo-ops that write data to memory locations:**

* **label1 .FILL #10**  
  will simply store the value #10 into the location we have called "label1"   
  Note that the symbol **#** indicates that the following number is a decimal integer; the symbol **x** indicates a hexadecimal integer; the symbol **b** indicates a binary integer (two's complement); a character enclosed in single or double quotes will generate the ASCII code for that character *(remember that in the LC-3, a character occupies a whole 16-bit word: the ASCII code is stored in the lower byte, and the upper byte is set to 0)*.
* **label2 .BLKW #5**  
  will reserve the 5 memory locations starting at label2 - i.e. the assembler will set aside those locations for use by the program when it runs.   
  Note that these memory locations are ***not*** guranteed to be set to 0 – when the program is run they may have random values stored in them until the program actually writes its own values there.  
  A very common use of the "block word" pseudo-op is to reserve memory space for an array.
* **label3 .STRINGZ "whatever"**  
  will write a null-terminated string of ascii characters to memory, starting at the address label3.  
  In other words, the characters 'w' 'h' 'a' 't' 'e' 'v' 'e' 'r', are written one ascii code per memory address, to the 8 memory locations starting at label3; this string is terminated by a null *('\0', i.e. the ASCII code b0000 0000)* written to the 9th address.

Now, what happens if one of these pseudo-ops is the first line in the code (or if it gets mixed up with the code block at any point)?

Whatever number or value you have stored there will get loaded into the Instruction Register, exactly as though it were a real ML instruction! The control unit will try to interpret it as an instruction & end up doing something completely unexpected.

In other words, any pseudo-op that writes stuff into memory ***has to be kept separate from the code itself*** - remember anything you store in memory with these pseudo-ops is DATA, not INSTRUCTIONS, and the two should not be mixed.

The simplest & most effective way of keeping them separate is to place a **HALT (TRAP x25)** instruction at the end of your code, and place all data after this.

The HALT instruction explicitly terminates processing *(actually by running a small routine that outputs a "farewell" message and then stops the processor's clock)*, so anything you place after it can never be "accidentally" accessed by the PC.

Another time you need the HALT instruction is to separate subroutines in your code.

Finally, **.END**, unlike HALT, is a pseudo-op, not an instruction.

It simply tells the assembler to ignore anything that appears after it in the source code file: do not confuse it with HALT, which translates into a specific ML instruction (invoking the halt subroutine), as described.