Project 2 Write Up - Sorting a List of Numbers with MARIE

Using the MARIE architecture and its corresponding assembly language, I was able to successfully build a Bubble Sort algorithm that organizes the elements within a list of size “n”, as specified by the user, so that each list item is rearranged in memory and displayed in the proper numerical order from least to greatest in value. My program operates using twelve variables and six functions. The twelve variables I used were “ONE”, “listSize”, “listPointer”, “listPointerValue”, “listPointer\_MIN”, “listPointer\_MAX”, “nextIndexAddressPeek”, “nextIndexValuePeek”, “greaterValueIndex”, “smallerValueIndex”, “sortLoopCounter1”, and “sortLoopCounter2”. “ONE” was used as a simple incrementer throughout the program, including an incrementation structure for when pointers needed to point towards the next element in memory. The “listSize” variable was designed to accept user input as for how large they want the list of their specified numbers to be. Easily, the most important variable of the entire program was “listPointer”, which keeps track of each of the values in the ‘list’ by storing the memory address of the list item needed at that point in time. This is useful when adding new items to the initial list, whenever an item’s address is updated by one, or when a “SWAP” between two items occurs. The “listPointerValue” element stores the numeric value being pointed to by “listPointer” and uses it for a numeric comparison with “nextIndexValuePeek” in the “COMPARE\_ELEMENTS” function. The “nextIndexAddressPeek” and “nextIndexValuePeek” elements operate the exact same way as “listPointer” and “listPointerValue” do, except that they instead refer to the next item in the list, as indicated in the “COMPARE\_ELEMENTS” function. The “listPointer\_MIN” and “listPointer\_MAX” elements store the first and last memory addresses, respectively, where the items of the list can be stored and, therefore, the range in which the “listPointer” can operate is defined between these two values. The “greaterValueIndex” and “smallerValueIndex” variables are used to indicate whether the value being pointed to by “listPointer” or “nextIndexAddressPeek” is greater in numeric value. These values are assigned in “COMPARE\_ELEMENTS” and used in “SWAP” so that the greater element is placed after the lesser one after each comparison occurs. Finally, “sortLoopCounter1” (set equal to ‘listSize - 1’) and “sortLoopCounter2” (set equal to ‘listSize’) are used in the “BUBBLESORT” function to increment through a makeshift for loop.

As previously stated, each of these variables were instrumental in the success of each of my six functions (“POINTER\_RESET”, “LIST\_APPEND”, “COMPARE\_ELEMENTS”, “SWAP”, “BUBBLESORT”, “LIST\_OUTPUT”) and within my overall program. “POINTER\_RESET” did exactly what it sounds like as it was responsible for resetting the memory address “listPointer” was looking at to be “listPointer\_MIN”. “LIST\_APPEND” handled assigning each of the newly inputted values for the list based on where the memory address “listPointer” was currently looking at. “COMPARE\_ELEMENTS” looked at the two elements being compared by “listPointer” and “nextAddressValuePeek” and determined which of them was greater. In addition, this function was only used by the “SWAP” function, which in itself was responsible for swapping the elements being looked at in “COMPARE\_ELEMENTS” to ensure that the former element holds a higher numeric value than the later does. The most important function was “BUBBLESORT”, which implemented the “SWAP” function (and therefore the “COMPARE\_ELEMENTS” function) so that these processes could be repeated for the number of elements present within the list through a for loop structure, as specified by the user.

The biggest problem that I encountered during this project was trying not to confuse myself on how I implemented my pointer variables and making sure to refer to them correctly. On multiple occasions I accidentally used the ‘LoadI’ command when I should have instead used the ‘Load’ command while trying to access the contents of the “listPointer” and “nextIndexAddressValue” variables. One specific instance of this occurred when I was trying to work on the “COMPARE\_ELEMENTS” function and was attempting to subtract two numeric values stored where I had two pointers looking. In short, I had forgotten that I had designed my pointers so that they would store the memory address that they were pointing to as the actual data that their memory addresses were supposed to point to. I somehow forgot that I had made this distinction and tried using ‘LoadI’ to access the addresses being pointed at, which instead called the actual memory addresses of the pointers themselves which, needless to say, created some substantial errors and triggered unintended effects, like accessing data from sections of my code I didn’t intend for. Eventually, I realized what my mistake was and fixed the problem, but it took some significant debugging to realize where I had gone astray.

As for my acknowledgements, the code and ideas that I implemented within this program are completely my own. In addition, the only help that I received to start building my design originated from both the textbook and my notes. I also did not collaborate with anyone else on this project. Finally, I am confident that my Bubble Sort program is able to function with any numerical user input as long as it doesn’t exceed the memory specifications or parameters that have been pre-set by default in the MARIE structure.