ECSE 324 - Lab #1: Introduction to ARM Programming

Alex Hale (260672475) and Tre Mansdoerfer (260680162)

October 6, 2017

Section 1: Largest Integer Program

The initial assembly program served as a means to introduce us to the language. There was no coding in this section; the code was provided. This program taught us how to bring data into our program with the load instruction, perform iterations and conditional statements using branch instructions, and how to structure assembly code files. Learning the structure of assembly files was more important than learning the instructions themselves, because while it is quite easy to look up ARM instructions in the reference manual, it would have taken some time to determine how an ARM assembly program should be structured without this complete example.

The loop portion of the program demonstrated the design of a loop counter. Following the counter, the comparing and moving instructions taught us how to operate on stored values in the registers. At the end of the program, we learned how to store values to memory and how to end the program at a given line by entering an infinite loop.

There were no difficulties with running this program and the code is, to our knowledge, efficient.

Section 2: Standard Deviation Program

When creating the standard deviation program, we reused as much code as possible from the Part 1 program. Specifically, we used the code from Part 1 to find the largest integer. We also added an extra storage location for the smallest integer, and we included an extra branch statement within the loop to find the smallest integer. We stored the max and min result to predefined addresses in memory. We calculated the difference between max and min, then divided the result by four using a logical shift of two places right. The logical shift was deemed as an appropriate choice (as opposed to the arithmetic shift), because the result of *max - min* will always be positive. Finally, the result of the standard deviation calculation was stored in memory at a predetermined location.

The challenges faced when writing this program arose from our inexperience with writing assembly programs, as opposed to difficulty determining the appropriate logic. We had to look up the syntax for most of the instructions, particularly the order in which to place the registers in the STR, LDR, and CMP instructions.

This program could be improved by performing the proper calculation of standard deviation instead of an approximation.

Section 3: Centre Program

The centre program calculates the average of a list of numbers, then subtracts that average from each of the numbers such that the average of the list becomes zero. To achieve this, we iterated through the list, summing the numbers to achieve a total. Knowing the length

of the list would be a power of two, we used another loop to determine that power. We then used a logical shift right to divide by the appropriate power, giving us the average. Finally, we looped back through the list of numbers, subtracting the average from each number and storing it back to its original memory location.

Our approach to this program was to begin from scratch and use the previous two programs as a guide for this program's structure. We reused the loop structure from the standard deviation program, with modifications to add the numbers to a total instead of finding the max and min. We also reused the concept of a logical shift right to divide by a power of two.

This approach caused more problems than the previous program, since we were still learning the basics of the structure of an assembly program. For example, we had many conflicts between which registers stored memory addresses and which registers stored data. We were confused about which registers needed to have square brackets around them in which instructions. We also hadn't understood the concept of the infinite END loop as the last instruction - it took us a few debugging cycles to realize that we had to create an infinite loop at END.

This program could be improved by finding a more efficient way to find the power of two of the length of the list instead of counting upward from zero. If the list were large, this would take a lot of time.

Section 4: Sort Program

The sort program uses the bubble sort algorithm to sort a list in increasing order, in-place. After the initial setup of counters and memory/register locations, the program enters the outer loop, which counts the number of times through which the list has been iterated. So long as the counter has not reached zero, the first two values in the list are selected, and the inner loop is entered. Inside the inner loop, each adjacent pair of values is compared. If the later value is less than the sooner value, they are swapped. After the inner loop has iterated through the whole list, the program returns to the outer loop. The process is repeated the same number of times as the length of the list, after which the list logically *must* be sorted.

Our approach to this program was to use the memory address and value setup from the previous programs, but to restructure the loops entirely. The nested loops allowed us to easily iterate through all the values in the list.

We took more time to write this program than the previous programs, because we were struggling to understand how we could use a flag to stop iterating once the list was sorted. Instead, we decided to iterate enough times that the list *must* be sorted, which was less efficient but simpler.

This program could be improved by using a flag to identify when no swaps have been made on a given pass, meaning the list is sorted.