1. **Introduction / Purpose / Intent**

For this assignment I was tasked to write a program to emulate an arm computer, executing at least 7 arm instructions from a txt file. The program will have one array to represent the general ARM registers and a second, multi-dimensional, array that represents memory. The program will read content from the input file and store it into the memory array. Then it will execute the instructions it finds in the array one instruction at a time and print the contents of the registers after each instruction.

A list of requirements was provided in the introduction section of the assignment page on canvas. The program must implement the utilization of ADD, ADDI, LDUR, STUR, and B instructions along with at least one other arithmetic or logical instruction and at least one logical branch instruction. Instructions stored in memory will use the assembly aliases rather than the binary op code. Instruction arguments will be register aliases, such as X0 and X1, memory locations in the memory array, such as 100 or 104, or immediate values, such as #5 or #0. The LDUR and STUR instructions will only use indirect addressing, such as [X0, #4]. The program should use the specific address of 200 as the default address for the start instructions in memory. At least 6 registers must be available, to include X0, X1, X9, and X10. The program must implement a Program Counter. Branch instructions will use immediate or register addressing only. Addresses will use decimal numbering to ease calculations. Instructions in memory must be on word boundaries, and incremented by 4 after execution. Data and immediate values will be signed integers with no size restriction.

After the writing of the program is complete, I am tasked to additionally write a test instruction file to be used with this emulator that will be provided as an argument when running the program. Inside this file there must be present at least one example of each instruction implemented in the program, at least 10 instructions, and instructions to complete at least one loop. Provided to me, to get started, was a sample file that contained example data for how a instruction file would be sequenced. The program will output a listing showing the progress through the instruction file, displaying the program counter, the instruction at the program counter location, and the values of any registers used in that instruction.

1. **Process**

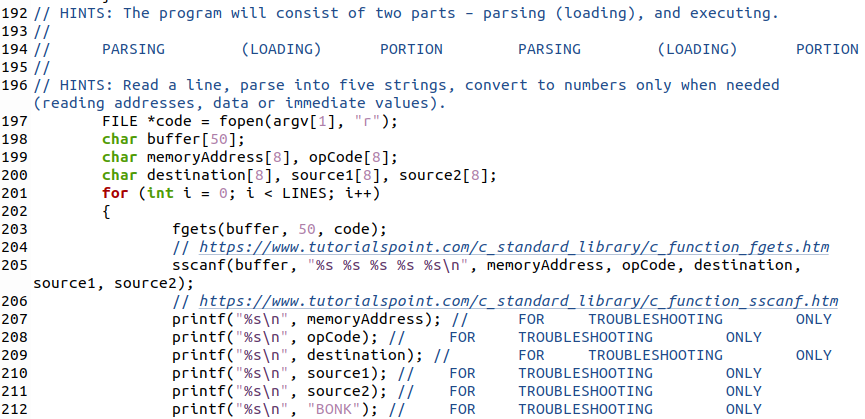
The first step in this lab was making sure I understood and was considering all that was being asked of me. To do this I copy and pasted every requirement from the assignment page on canvas as comments into a blank ARM.c file in my Ubuntu VM. Once this was complete, I constructed a very simple shell of a program, with a header comment at the top, the basic include statements I would require, space for global variables, space for functions, and the main() program. I then moved the comments to the areas I thought they would be applicable, for example the statement “*// HINTS: It will be useful to implement each supported instruction as a function.*” Was placed near the top in my “*// FUNCTION DECLARATIONS*” section. These comments outlined the basic tasks I would need to complete one at a time. I paid specific importance to the 5 HINTS comments, especially “*HINTS: The program will consist of two parts – parsing (loading), and executing.*” This line is what I used as my largest scope step to complete, first the parsing of the file, second the execution of the code. Because I cannot execute what isn’t there, I began with the code parsing.

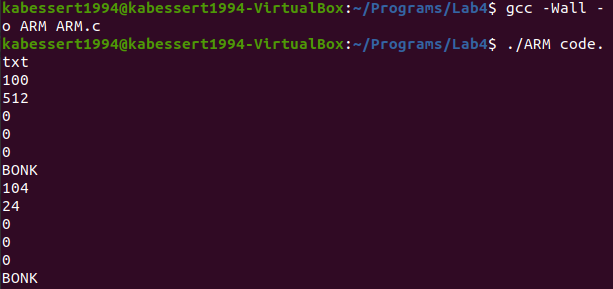
PARSING PORTION

An additional hint provided stated *“// HINTS: Read a line, parse into five strings, and convert to numbers only when needed (reading addresses, data or immediate values).*” This provided me the context for my next few lines of pseudo-code. First I would need to open up the provided code.txt file for work with my program, then I would read in line by line the input of that file. I utilized the function fopen() which took an argument of the test file “code.txt”, stored in **argv[1]** from the command line, and what mode I would like the file open for, which I chose “r” for read only. Now my program successfully opened the file upon execution.

I knew that I would then have to process the information in the file, which the above hint laid out for me. My next subtask would be to read in a line from the file, which I accomplished with an call to fgets() which reads data from a file until the new line character is read. I declared into existence a **buffer** string variable that I could use to store each line as it came through, and place a for loop around my call to fgets(). This way as long as there were lines left inside the file, fgets() would read the line and place it into **buffer**. Now that I had a loop and a string that only contained the contents of a single line from the file, I moved onto the next subtask.

I needed a method to parse the string I acquired through each loop iteration into the 5 separate fields. After communicating with my peers, the easiest method was to utilize the sscanf() function, that places string values into variables separated by whitespace. To utilize this function I would need places to store the data, so I declared into existence a string variable for each of the positions within the test file; **memoryAddress**, **opCode**, **destination**, **source1**, and **source2**. Even though not every instruction uses these parameters, the label tells me exactly where they would need to go in any instruction. The program now successfully separates all of the data from the buffer string and places them into variables that can be seen with a printf() statement. The program overwrites these variables for each line of the test file, and outputs them one at a time as expected. Here I use “BONK” to show the end of the loop to ensure the data is in its correct location.



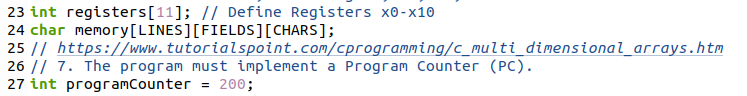


With this separation accomplished, I now needed to construct the multidimensional array that would store all of this data, as indicated by the instructions. To accomplish this, I defined 3 statements that would shape my array; **LINES**, **FIELDS**, and **CHARS**. **LINES** was set to 50 to account for possible lengths provided during testing later, while **FIELDS** and **CHARS** were set to 5 and 10 respectively, as no instruction passed would have more than 5 fields or contain more than 10 characters per field. I then declared, as a global variable, the **memory** array into existence, which used the 3 define statements as variables to construct its size. This array effectively replaces the local variables that were being used to store the results from sscanf().

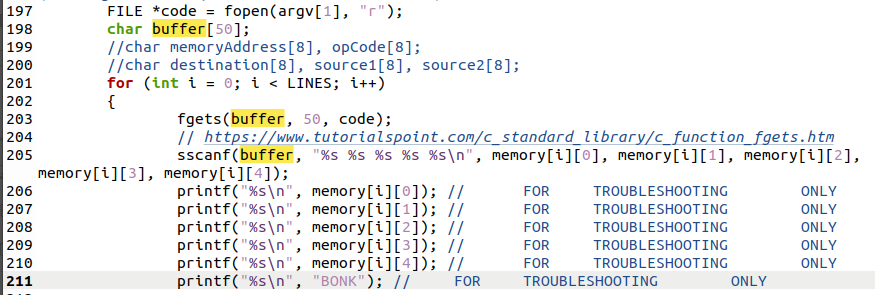


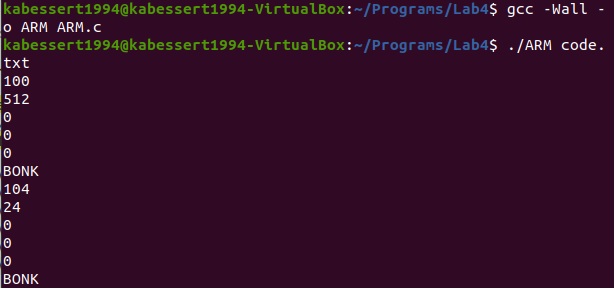


While defining this array I came across a few requirements that could easily be accomplished before returning to main to continue with the parsing of input data. One of these small tasks was the requirement to establish an array that represents the general ARM registers, so I declared as a global variable int **registers**[] with a size of 11 to accommodate the registers X0 – X10, as required. I also declared into existence, as another global variable, a **programCounter** variable and set it to 200. This was outlined as the beginning memory location for instructions, leaving 100 – 196 for data. With these tasks accomplished, I returned to main() to implement my multidimensional array.



Initially, I utilized the strcpy() function to place the contents of the local variables into the **memory** array one variable and line at a time within the for loop. As I was testing this method with printf() statements I realized that I had completely eliminated the need for the local variables with my declaration of memory, and instead placed it in the scanf() function to receive the values straight from buffer. This proved to work exactly the same as the prior rendition of the loop, and was implemented for the final version, removing the unused variables and testing lines. With this loop in place, the first overarching task was complete, and I moved on to the execution portion of the program.





EXECUTION PORTION

To begin in this portion of code, I referred to one of the hint comments provided by the assignment page on canvas; “// HINTS: A loop with a switch statement can iterate through the instructions.” At first I attempted to construct a switch statement nested in a while loop that would use the opcode names stored in **memory**[] as cases, but later could not get any of the cases to work and shifted my design as a sequence of “if” / “else if” statements nested within that same while loop. I utilized a single statement per instruction I would be executing. A while loop was chosen, instead of a for loop, to better control the indexing variable. This loop tested against the define statement **LINES** with a **lineCount** int variable to keep track of how many lines had been executed from memory, acting as an index of the **memory**[] array.

The biggest push to the form of the program was “// HINTS: It will be useful to implement each supported instruction as a function.” This is what drove me to place a commented out call to each of the required functions in each of the “if” statement prototype in main. With this completed, I moved up towards the top of the function and added function declarations for each of the required functions. I also settled on SUB and CBZ as my additional instructions to complete, as SUB would be very similar to ADD and CBZ would be similar to B. These declarations created a new overarching step, within the execution portion of code, and that was the definitions of these functions for actual functionality. I decided to leave these for later to focus on my “if/else” tree.

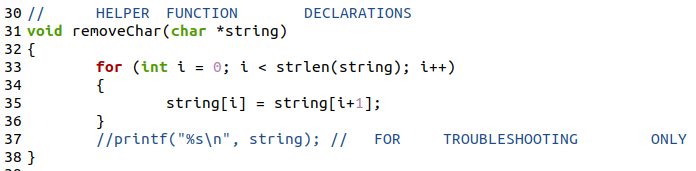
Each statement would test the strcmp() result of the operand value to the literal string it should match, very similarly to a switch case. Inside each of these statements I placed, alongside the commented out corresponding function call, a printf() statement that would display information of the instruction as required by the assignment page on canvas. Additionally I places an increment of the index variable **lineCount** and added 4 to the **programCounter** after the function call. At the very end of this I placed a single else statement that simply incremented the line count, as there are 3 values stored in memory, from the code.txt file, that are not instructions, and therefore would increment the lineCount but would not execute any function. I also placed a printf() statement to output “BONK” to error check as I proceeded. This statement was removed from the final portion of this program. This completed the basic design of main(), which allowed me to return to the function declaration portion of my program and begin the manipulation of the data stored in **memory**[].

FUNCTION PORTION

I decided to tackle this task as function pairs, focusing on the similar function directly after the completion of the first like it. I paired ADD/ADDI and SUB together, along with LDUR and STUR, and finally B and CBZ. The test file provided did not cover the scope of all of these functions, so I began with ADD and ADDI as they appeared most frequently in the provided code.txt example. Each function also would be passed a series of values, so prior to working on the definitions I simply added arguments to all of the declarations that correlated to their related ARM instruction; for example ADD() received the arguments **dest**, **src1**, and **src2**. I hopped quickly down to main() to update the commented out function calls with the corresponding **memory**[] values for later testing of completed functions.

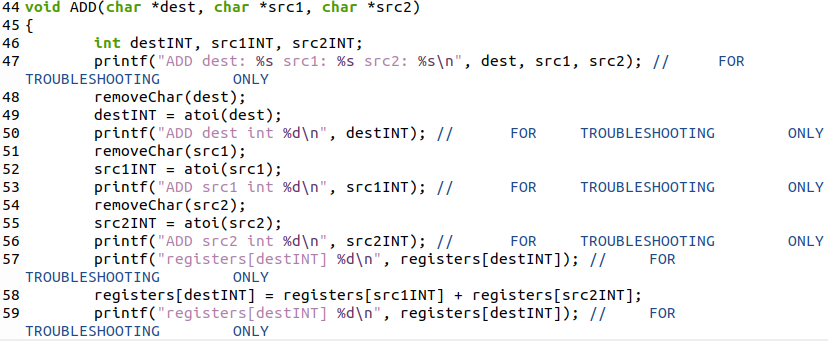
* *HELPER FUNCTION*

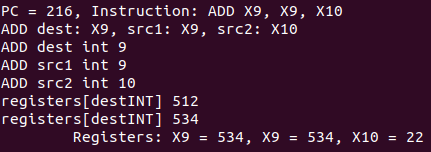
I realized as soon as I began working with ADD() that each of these arguments were still stored as strings. After deliberation with my fellow students, I decided that the creation of a helper function to remove the first character of each string was absolutely necessary, as the existing function atoi() was not only authorized but also ignored trailing non-integer values. This meant that as long as there was an integer as the first value passed to atoi(), it would successfully convert the string to the correct corresponding int. I declared he helper function removeChar(), which was a simple for loop that shifted each character up the array by a single value. This would take any string, such as a register X9, and replace the first value with the integer so that atoi() would be able to operate. With this helper function finished, I returned my focus to my first instruction function.



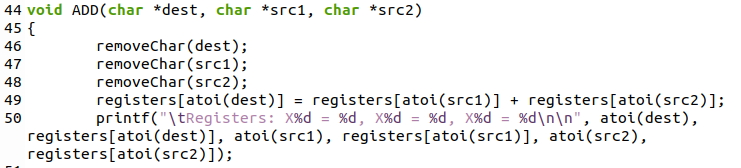
* *ADD, ADDI, SUB*

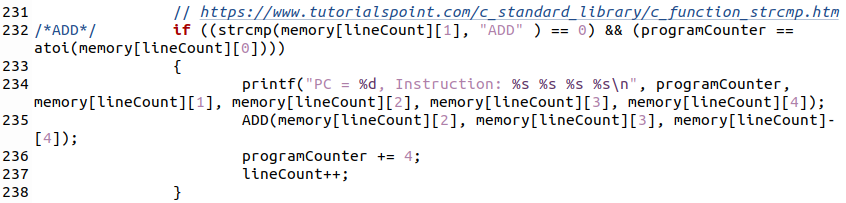
I first designed the function to utilize local int variables that would be used to store the return values of atoi() after the first non-integer character had been stripped. This was an efficient but cumbersome process, however it allowed me to place many testing statements throughout the function to track if the expected values were being passed and manipulated.





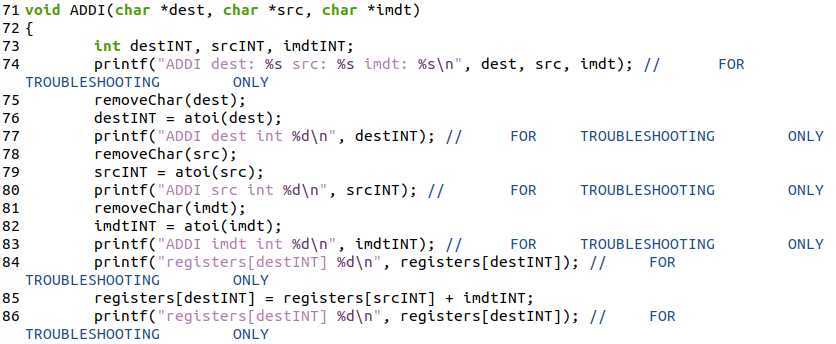
After the function tested successfully and I began commenting out the testing lines in preparation for the next function, I thought of a way that would eliminate the need for these integer variables, by using the return values of atoi() as the indexes of my **registers**[] array. This method was more efficient, however did not allow for as much tracking as previously stated. The original version of the function, utilizing the integer variables, was copied to the SUB and ADDI definitions for manipulation to ensure the proper passing and manipulation of data; however I did decide to utilize the more efficient code for the final product, which has been submitted alongside this report.

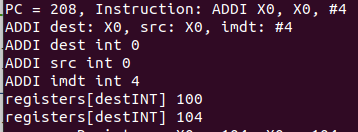




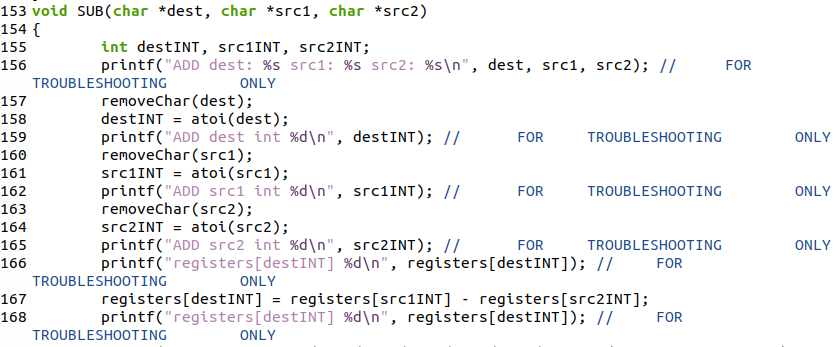


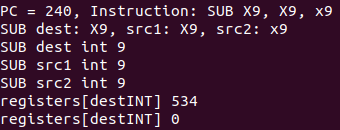
SUB and ADDI required very little tweaking to ensure successful implementation, with the biggest change being inside ADDI. Having an immediate value as one of the arguments, no register needed to be accessed, and so the return value of atoi() is what is operated with instead of it being the integer referring to some other value.



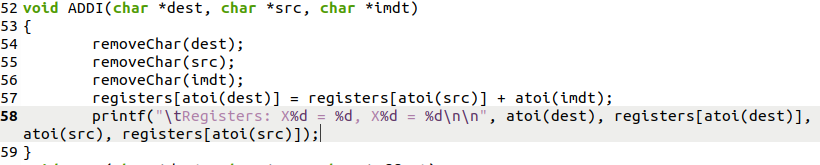


ADDI was present inside the code.txt example file, but SUB was not and I needed to construct a new file to test it. That process will be recorded later in this section. These functions were successfully implemented with their individual minor tweaks as can be seen in the above and below screenshots.

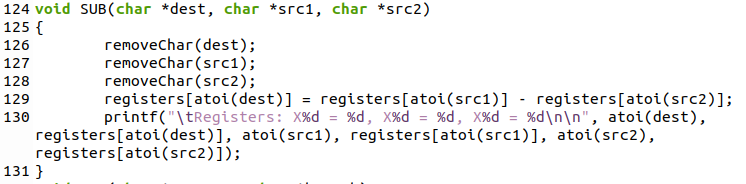




These function underwent the same slimming treatment as their parent function, and the bulky int variables and troubleshooting statements were cut out. Once these were confirmed still working, I moved onto the next pair of functions.









* *LDUR, STUR*

To begin work on these functions, I utilized the original code copied over from the ADD function as a base due to the acceptance of 3 arguments. Every function must undergo the same process of removing some non-integer characters, and the original clunky method used to test the implementations of these functions has proved very effective. The code borrowed from ADD was manipulated, adding an additional removeChar() call to the **src** due to the presence of a bracket before the register.

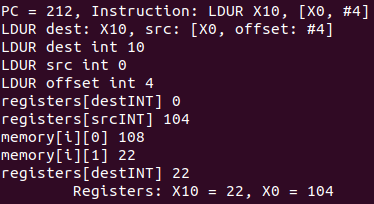
The logic with these functions, after the stripping of non-integer characters, would be to compare each of the integer values of the 100-level memory addresses to the value indicated by the LDUR or STUR instruction. For example, in the provided code.txt file, “*204 LDUR X9, [X0, #0]*” indicated to load the value stored in the address of register X0 plus an offset of 0. The value of X0 at this point in the code.txt file is 100. So the function would need to compare the sum of these to values to each memory location to find a match. Since the memory locations are stored as strings, these would need to be passed to atoi() for comparison purposes.

I was working first with LDUR, due to the fact that it appears in the provided code.txt file. the best way to index through the memory locations would be with a for loop, which would start at the bottom of the **memory**[] array and increment up. Inside this for loop I placed an if statement to compare the value contained in the first field, the opcode field, to the sum of the value contained in the **src** register and the offset value. Once a match was found by the if statement it would place that value into the corresponding destination register.

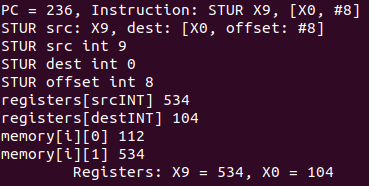
The logic for STUR was reversed, as I had an integer that needed to be stored from a register into memory. This was accomplished with the sprintf() function, placed inside the same loop system as is present in LDUR. The integer insidethe third argument is placed into the **memory**[] array upon a matching value in the first field of memory and the sum of the value held within the destination register plus the offset. In the instruction set I wrote for example, “*STUR X9, [X0, #8]*” is executed by looking at the value held inside register X0, which was 104, and adding the immediate 8 for a total of 112. Whatever was being held in the source register, X9, is going to be stored at that location in memory. This can be seen in the screenshots below.

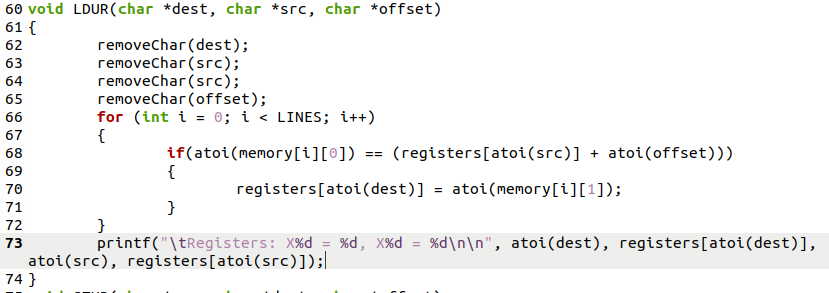
Both of the LDUR and STUR functions were originally written using base code that was copied from the ADD function, so each initially used integer variables to hold the returns of atoi(). They have been slimmed, just like the other functions, to be more efficient using atoi(). These second revisions, along with their outputs, can also be seen below.



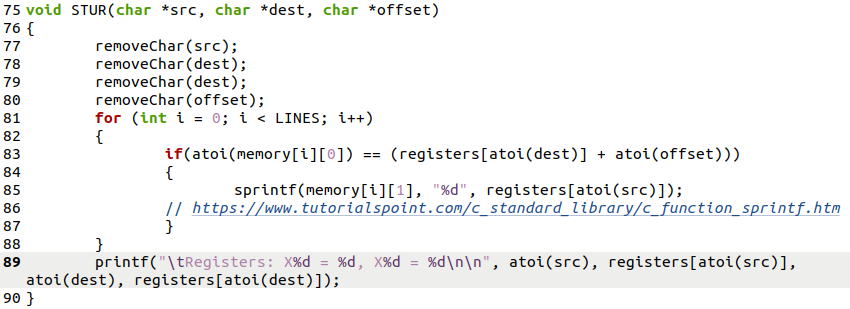








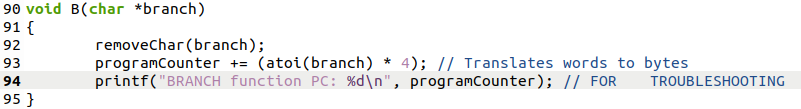


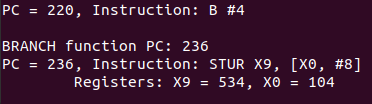


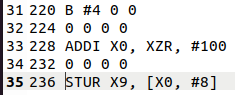


* *B, CBZ*

The last two functions were the most difficult to implement, as I had up until this point not been utilizing the **programCounter** for anything more than an additional **lineCount** variable, incrementing it with no real idea on how to implement it thus far. Inside the BRANCH function I decided to manipulate the **programCounter** directly by taking the input immediate, which indicates the number of instructions to jump over, and multiplying it by 4 and adding back to **programCounter**. This sets the program counter to the next instruction I wish to execute.



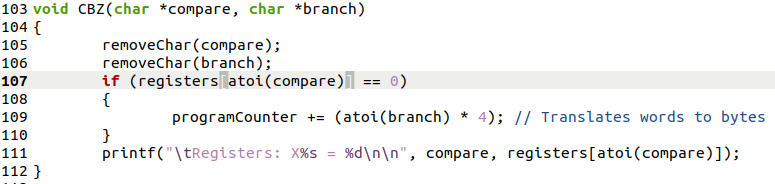


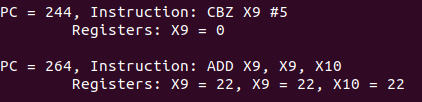


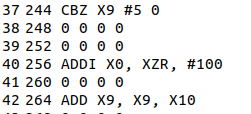
This was partially accomplished by adding an additional clause to each of my if statements in main(), not only testing if the opcode matched one of the expected opcode aliases, but also now testing that the **programCounter** matches for that given instruction as well. This ensures that only instructions that have matching opcodes and memory locations are successfully executed, as seen in the screenshots above the instruction at 228 is not executed because the **programCounter** does not match the memory location even though ADDI is a valid opcode. These were added to each of the if/else statements in main as can be seen below.



Finally for adapting this for the CBZ function required only the addition of an if statement around the code for BRANCH, which tested the value of the condition to compare against 0. If the value in the compare is not zero, then the instruction does not execute. And additional printf() statement was added for this function to meet the requirements of outputting registers that are utilized in functions. Since Branch only used an immediate value and already output the **programCounter**, no further statement was required.







Similarly to BRANCH, CBZ skips the execution of the ADDI instruction at memory location 256 because the if statement in main is checking for matching opcode and **programCounter** values. Within main(), neither branch increments the **programCounter** after execution, as this is calculated into the function call itself. With all of the functions written, this concludes this subtask which at this point in the project had been completed, but I need to go over for completion.

INSTRUCTION PORTION

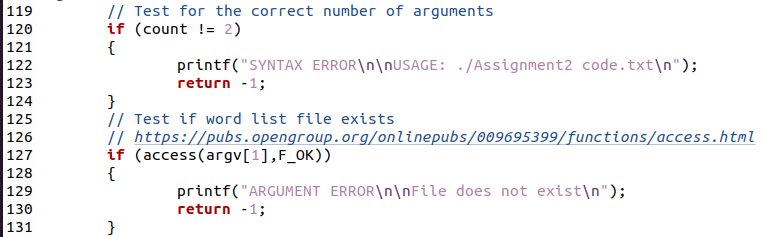
Because the provided code.txt had such limited testing capabilities, I quickly had to generate a new file that had the capability to test all of my functions and the program’s function in its entirety. To begin I created a space for every word location in memory up to 300. This is so that once the file was read into memory at the beginning of main() the correct size array would be made to allow for additional storage. The meat of the file was added after the 216 instruction to test the branches I had created. Luckily attempting to test the CBZ branch required use of the SUB function as well to create a zero condition. The final thing to test, that was not included with the example code.txt was the STUR function. I used is as a BRANCH target to demonstrate its functionality. 2 ADDI statements were added between both branches and the branch targets for testing of the **programCounter** implementation, which was a great success! By providing this file as an argument, the program runs successfully as intended to display the range of functionality in an orderly manner.

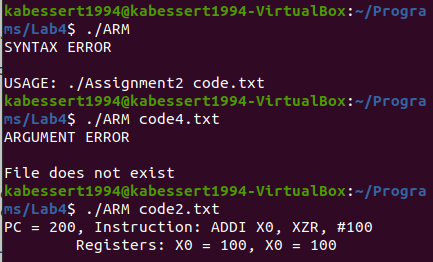
1. **Testing**

All throughout the coding process many “printf();” statements were used to output variables in various locations, such as in functions, loops, before and after assignment statements, and within conditional statements to ensure proper use of variables, expected behavior of the program, and manipulation of data. These statements used for troubleshooting have been shown in the screenshots provided up to this point of the report, but have been removed from the final revision of the submitted ARM.c code that was submitted alongside this report.



Since I knew a file would be passed to main from the command line, I began by conducting a simple test at the beginning of main to first, ensure the file was present as an argument, and second to test if the file specified existed in the directory the program was located. This was then tested successfully as can be seen below.





A test was conducted with a file holding no data to ensure proper execution, but because of the way the file is parsed and loaded into memory nothing happens. The blank file is successfully found, opened, read, parsed, placed into memory, checked, and the program exits successfully. As nothing is present in the file, no action is taken by the program.

Next a test was conducted of a file that utilizes instructions at earlier memory locations than 200. This invalid organization for the instruction file does not cause problems to the program, as only instructions after memory location 200 are executed. This is because the **programCounter** starts at 200. An instruction file containing a branch to change the value of the **programCounter** was utilized to attempt to access the data in the earlier memory locations, however this was unsuccessful as the program only continues ‘down’ through memory, not in circles.

This is counter to the requirement of the operation and display of a loop within the program. An instruction file containing an attempted while loop was tested, but as the program only continues ‘down’ through memory no other instructions are executed after the branch sets the **programCounter** to a ‘higher’ value. This oversight is correctible, but due to time limitations and the amount of work that would have to go into redesigning the current version of the code I have chosen to forego this endeavor. It may be possible that I will implement it by the next half of the ARM.c lab.

Following on from this, utilization of the XZR register is supported, but not intentionally. It was not expressly stated that functionality needed to exist for this register, however it was present in the provided code.txt example. Testing of the original code.txt and my generated code2.txt file successfully executed due to the interaction between atoi() and a string of chars. Since no integer exists in the XZR string, stripped to ZR by removeChar(), the function returns a zero. This was not an intentional design choice, but a coincidence between provided example and function operation.



1. **Results**

The results of this lab are majority successful in completing the intent of the assignment. The code2.txt file I generated supports the statement that the program operates properly as an arm emulator within the given constraints of the assignment. As stated the functionality of loops is not present, however all other requirements for completion have been achieved. 7 arm instructions are successfully executed from a text file, and the program has 2 arrays present to represent registers and memory. The program does read content from an input file and store this data into the multidimensional memory array. It executes the instructions it finds in the array one instruction at a time and prints the contents of the registers used after each instruction.

The 5 provided arm instructions ADD, ADDI, LDUR, STUR, and B are present alongside the additional arithmetic instruction, SUB, and the additional logical branch instruction, CBZ. Instructions stored in memory use the assembly alias for opcode rather than their binary equivalent. Instruction arguments are register aliases, locations in the memory array, or immediate values. The LDUR and STUR instructions only use indirect addressing. The program does utilize a program counter and uses the specific address of 200 as the default address for the start instructions in memory. At least 6 registers are available including X0, X1, X9, and X10. Branch instructions use immediate or register addressing only. Addresses use decimal numbering for ease of calculations. Instructions in memory are on word boundaries, and the program counter is incremented by 4 after execution of an instruction. Data and immediate values are signed integers.

Finally, the test file written for use of this emulator does include one example of each instruction implemented in the program and at least 10 instructions. As stated it fails to complete at least one loop. This is not because it is not included within the code2.txt file, however that is true, but the code itself was not designed in such a way to handle an operation. The program outputs a listing showing the progress through the instructions and properly displays a program counter, the instruction at the program counter location, and the values of any registers used in that instruction.

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have successfully completed the lab to the best of my abilities. This lab took a significant amount of time and effort to merely understand what was being asked of me to perform as a programmer. However, like other labs before this, the key to success was breaking up the larger tasks into smaller accomplishable steps. The HINTS section of this lab was instrumental at directing me how to begin with each of the few major tasks of this lab. This lab also required much deliberation among myself and other students. Without my peers I would be unable to check my understanding of the ARM logic, and how to craft C solutions to match the expected operation of those instructions. The goal of the lab seems to have been to generate a working relationship between the programmer and the arm language, reinforcing the understanding of operations.

1. **References / Acknowledgements**

C Programming Language, B. W. Kernighan & D. M. Ritchie, 2nd Edition, Prentice Hall, 1988.

C Programming: A Modern Approach, K.N. King, Norton, 2008.

[Tutorials Point: strcpy](https://www.tutorialspoint.com/c_standard_library/c_function_strcpy.htm)

[Tutorials Point: atoi](https://www.tutorialspoint.com/c_standard_library/c_function_atoi.htm)

[Tutorials Point: Multidimensional Arrays](https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm)

[Tutorials Point: sprintf](https://www.tutorialspoint.com/c_standard_library/c_function_sprintf.htm)

[Tutorials Point: access](https://pubs.opengroup.org/onlinepubs/009695399/functions/access.html)

[Tutorials Point: fgets](https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm)

[Tutorials Point: strcmp](https://www.tutorialspoint.com/c_standard_library/c_function_strcmp.htm)

Logan Lipke & Calvin Hewitt for general understanding of the requirements of the lab, logic checks for function definitions, logic checks for program functionality, logic checks for arm instruction operations, and troubleshooting 23-24 Oct 2020