1. **Introduction / Purpose / Intent**

For this assignment I was tasked to add additional functionality to the ARM emulator program that I had designed in Lab 4, ARM.c. The functionality required to add is the inclusion of stack operations. The objectives of this lab are listed as; to understand how the ARM stack function and understand function calls in ARM. Much of the requirements are similar to that of Lab 4, and indeed the start of this lab is with the submitted ARM.c file submitted for Lab 4, so I chose to focus on and note only the differing portions between the two Labs.

The program must have allocated stack space within the multidimensional **memory**[] array as well as be output to the terminal similarly to the program counter and executed instruction. The program will implement the SUBI, BL, and BR instructions, as these were not defined in Lab4, and branch instructions will use immediate or register addressing. At least 8 registers must be available, to include X0, X1, X9, X10, X28 and X30; X28 is the Stack Pointer and X30 is the Link Register. The “BR XZR” shall be used to terminate the program.

The stack must be at least 100 words deep, and care must be taken to ensure that the stack will not grow over the program/data area. The program is required to have a check for as well as a “stack overflow” error message. A high memory address are should be chosen to allow the stack to grow down. Stack memory locations should be on a double word boundary. The stack pointer shall be initialized to the top of the stack. The stack, when used, shall be output to the terminal to include any memory locations that currently store data.

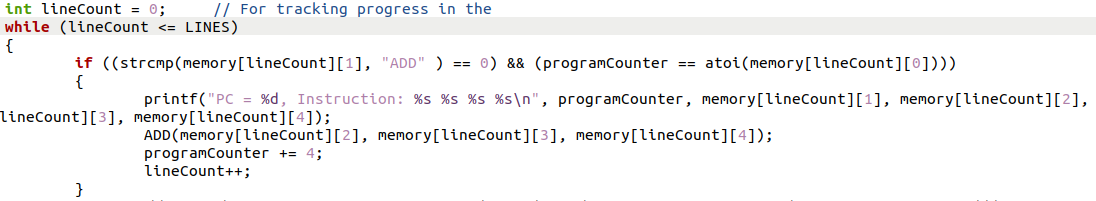
A portion of this lab is to create and provide a test file along with the program that demonstrates certain functionalities of the program. This file must include at least one example of each instruction that is implemented, at least 12 instructions, and must complete a single loop. It is also required to execute a function call and a push and pop to the stack in an effort to demonstrate comprehension. An additional file, not accessible to the student, will be used to grade the program’s functionality. This file will calculate the sum of a series of numbers from 1 to N using recursion. The file provided by the student is not required to contain this function, however it should test the functionality used.

1. **Process**

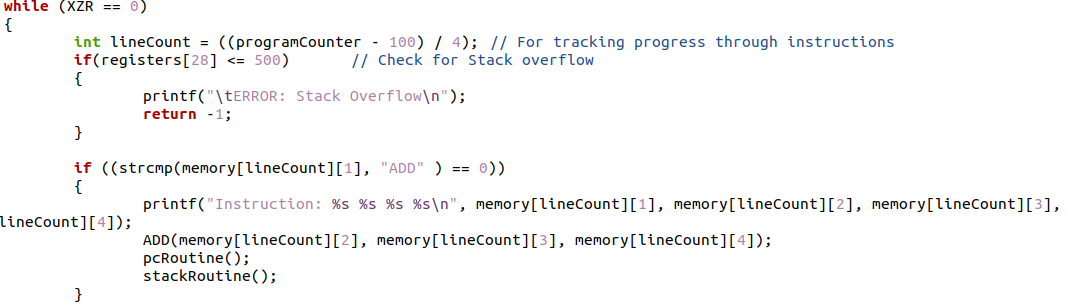
The first step in this lab was to comprehend what was required to be changed from the previous lab. For starters I knew that the functionality of my submitted Lab4 program was missing the ability to utilize loops, which poses a large threat to the completion of this lab. This was the largest task I would have to tackle. Additional pieces needed to be added or altered to the ARM.c code, to include; the adding of additional instructions, the adding of the stack, the adding of the stack output, the adding of the stack test, and the manipulation of the code.txt file. These all were my beginning pseudo-code tasks to complete, and I decided to tackle the loop issue held over from the previous lab first

* ***Enabling Loop Functionality***

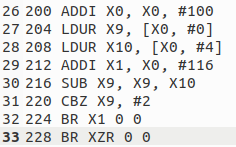
This process seemed to be the largest portion of the lab at the beginning, but later proved to be rather simple to accomplish. The first thing to alter from the old ARM.c code was its manner of progressing through the code file. The while loop that controlled the execution of each of the instructions through nested if else statements was using a **lineCount** variable comparison as its test, incrementing as instructions progressed in the file. This variable was tested against the declared **LINES** value and only increased, meaning that a loop was never possible.

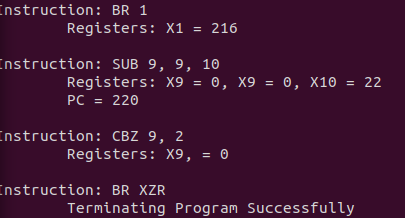


This test in the while statement was altered to an always true statement, so that the while loop would run until a return statement was encountered, and the lineCount variable was moved inside the loop, reinitializing through every iteration of the while loop. Additionally, instead of being incremented through one of the executions of an “if else” statement, **lineCount** is now arithmetically derived straight from the **programCounter** variable. This change to the handling of the code.txt file means that the additional test per if else statement nested within the while loop for executing instructions can be removed.



This new code was tested with a simple loop to ensure functionality, and it passed successfully, indicating that loops are now possible within the program.

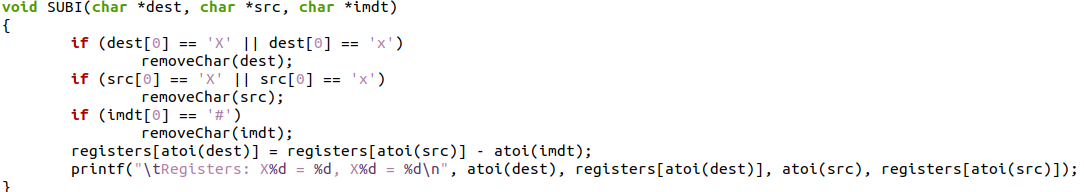


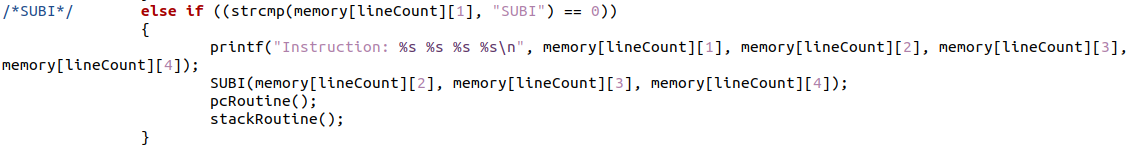


* ***Adding Instruction Functionality***

As seen in the above screenshots some functionality was required to be added to ensure that a loop was possible. At the point that I wished to execute a branch for testing I did not have a branch that I could utilize, as both of the branches for ARM.c used indirect addressing. I was required to put testing of the loop on hold to generate a branch that could utilize a memory address stored in a register, BR, and so I had to move forward and complete the required functions that would execute the additional SUBI, BL, and BR instructions.

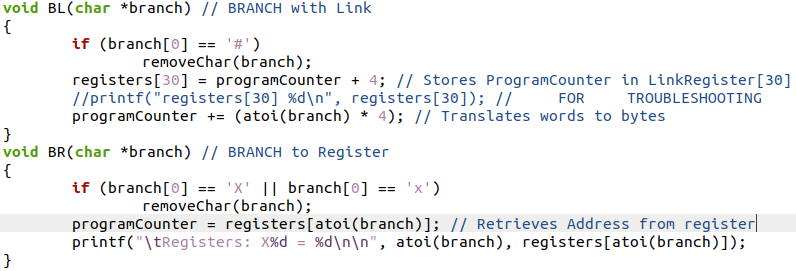
SUBI was easy and required little effort, as I simply copied the ADDI function and switched the sign used in its arithmetic. In main I also copied ADDI’s “if else” statement and tweaked to match the new instruction.

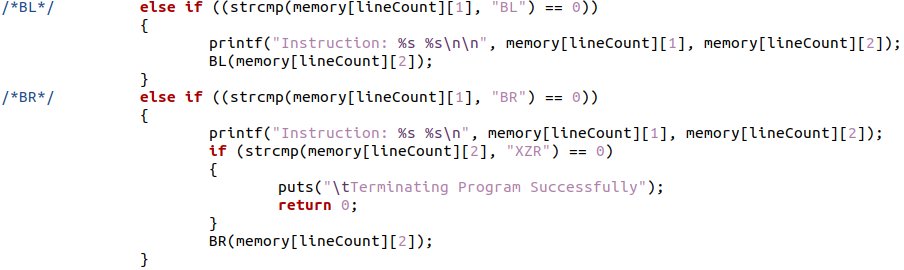






The following branches were not as simple to implement, however. Both BL and BR started life as copies of the existing Branch function and main() nested “if else” statement, and since BR was the function I wished to complete for testing of loop functionality I started with that one. BR would simply look inside a register, instead of using an offset as B did, take the value found inside that register and make the **programCounter** equal that value. With this pseudo-code generated the simple BR function that operates alongside with the new handling of the program counter inside of main(). The BL function was a Frankenstein derivation of BR and B, using the BR function as a base but adding the storage of the current **programCounter** into the link register, X30. BL utilizes offset values similarly to B, so that functionality was also changed from the BR code.





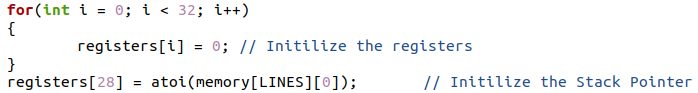
As seen in the above screenshot the functionality to exit the program upon a “BR XZR” was implemented inside of main() with a nested “if” statement. This statement simply looks for the branch to the XZR register and terminates operation. Otherwise if simply calls the BR function. After these were implemented I returned to testing loop functionality, which succeeded and allowed me to move forward to the next task.

* ***Adding Stack Functionality***

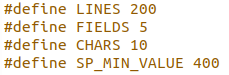
This portion proved to be the largest and most complex task to complete as the addition of stack functionality is very closely tied into the writing of instructions that utilize it. For this portion of the report I will go over what I have added to gain stack functionality and later explain how it was tested with the writing of a new code3.txt file.

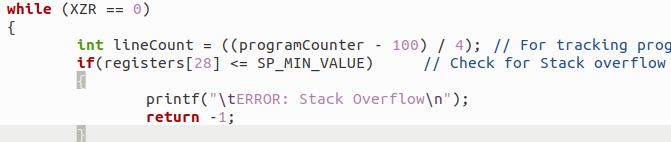
To begin I extended the range of my global variable declaration of registers to include all 32 registers in an arm processor. This extension enabled the X28 register that would become my stack pointer and the X30 register that would become my link register for branching with BL. Once the register existed I implemented an initialization of the X28 register to equate what the last line in the code.txt file would be. This accomplishes choosing a high memory address for the stack pointer, and allows it to grow down towards 0 as the stack grows.



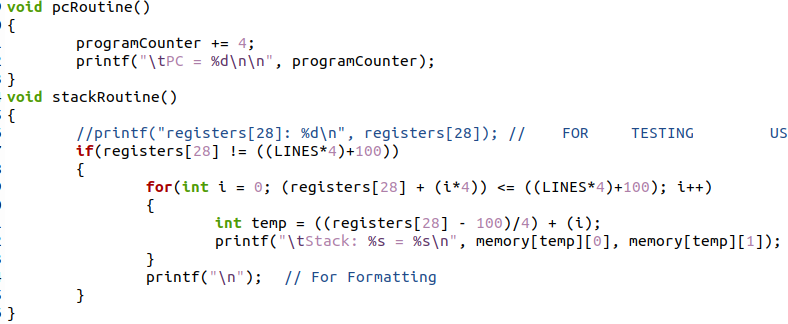


Because of the functionality of the stack, I would also need to ensure some test was being performed that checked if the stack had grown too large and may overwrite memory. The test I chose to implement was simple, and did not test the actual values held in any locations. Instead I chose to test the value of the stack pointer to ensure that it never crossed a specific value boundary. To do this I created a new “define” statement at the top of the program that would allow easy manipulation of this value. I set the statement to 400, which allowed me 50 lines of instruction assuming instruction starts at memory location 200. This will allow as much space as the program needs for instruction, and is easily manipulated if more is required.





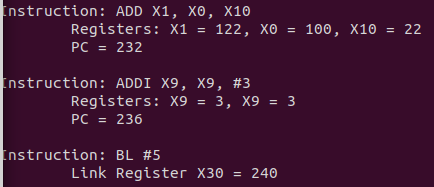
Finally, I knew that some form of routine would have to exist for the stack pointer, similarly to the routine that existed for the **programCounter** variable. The stack would need to be output to the terminal, and only the area of the stack that had been utilized. I crafted an “if” statement that checked the value stored in the stack pointer register, X28, against the value that it should be initialized to. If these were no equal, then I would know to output some information to the terminal. I crafted a “for” loop that would iterate through each position of the stack and output what was being held in that memory location. This was placed in a helper function, along with the **programCounter** routine and placed into main() inside each of the “if else” statements.



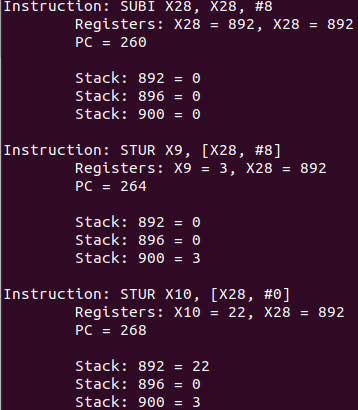
* ***Writing Instruction File***

This portion, as stated before, tied very closely into the testing of the stack pointer due to the requirement of instruction to manipulate the stack pointer. This took the longest time to accomplish and troubleshoot, and will be covered heavily in the testing portion of this report, but for now I will write only about the creation of the file and why it was chosen to be the way that it is. I began by increasing the size of the file, adding in additional space for the stack to exist and work within, all the way to memory location 200 which gave the stack plenty of space to operate in the recursive test that would be conducted during grading.

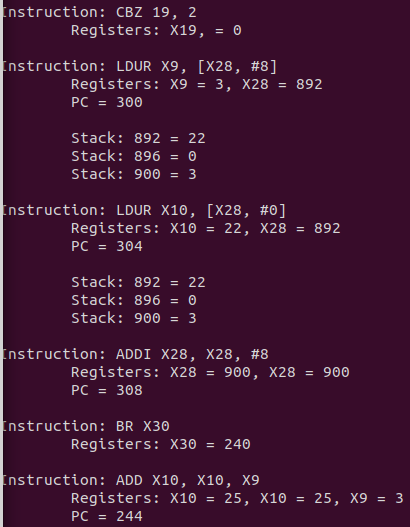
As demonstrated previously I had a working loop within the code3.txt file that utilized a number of instructions, which accomplished the requirement of demonstration a loop, so I decided to build on this test. I removed the BR XZR from line 228 and began to craft a function call using the BL function I had just earlier finished. This would test the functionality of BL() and accomplish one of the required tasks. Prior to this call I used the ADDI instruction to place some value into X9 for ease of tracking and testing. BL then was called, utilizing immediate addressing as instructed, stored the **programCounter** into the Link Register, and placed me at line 256 inside a function.



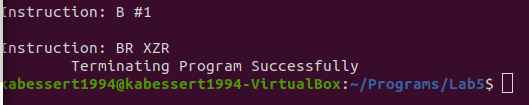
Some things required to follow this call, as I am now inside a function with my old program counter stored. The data I had stored previously in my temporary registers needs to be stored in order to use those registers for the execution of instructions inside this function. I utilize the SUBI instruction to move the stack pointer down enough to store both of my temporary registers. Then the STUR command was called twice storing each of my temporary registers onto the stack, causing the stack routine to output to the terminal and indicating successful storage and implementation of the stack pointer.



Following this, I crafted another simple loop, using the temporary registers to hold the data, which would demonstrate again the functionality of a loop within a function. Once the loop was again complete, I utilized the LDUR instruction to place my original temporary register values from the stack back into their registers before enacting the BR instruction to the link register to return to where the function call had been made. Since the stack was now not in use, as the function call was coming to an end, I utilized the ADDI instruction to place the stack pointer back to where it originally pointer, ending the utilization of the stack. After the branch to the value held in the link register, I added a final line utilizing the ADD instruction to add the temporary registers together and demonstrate that they again held the values they had prior to the function call, which successfully executed.



Following this, I had yet to utilize the B instruction, so I utilized it to point to the BR XZR instruction that would end the program successfully. With this completed, the program and the code3.txt file had both been successfully completed and tested working. All conditions of the assignment were met, and all functionality required of the program was demonstrated. This concluded the writing of the program for this lab.

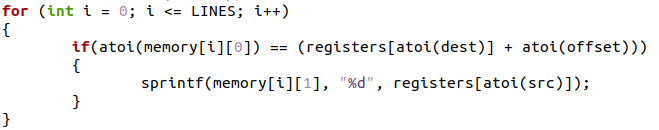


1. **Testing**

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 ARM2.c code that was submitted alongside this report.



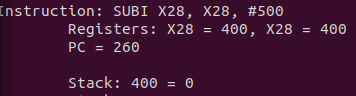
During the writing/editing of the ARM2.c program, a large portion of the time spent was spent in troubleshooting and tracking down errors as previously mentioned. One of which errors was the inability to store data at the largest stack location, at the time 900. Data would be stored perfectly fine anywhere else in the stack up to the barrier that was placed for stack overflow errors, which tested successfully. This lead me down a multiple hour examination of the code, trying to follow line by line in both the ARM2.c file and the code2.txt file to ensure the STUR() function operated correctly with the stack pointer register and the stackRoutine() function successfully output the data of all memory locations. All indications pointed to successful storage of the X9 register into the 900 memory location, until I stumbled across the smallest error within the STUR and LDUR functions. The test condition within the for loop inside the functions only tested “i < lines”, so for the 900th memory location to be loaded or stored would require “i <= lines”.

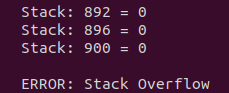


This small error was so difficult to find due to the massive scope of this project, and when I asked for help from my peers it was difficult concisely convey how the program operated and what was going wrong. After the error was changed and the equals sign was added to both of the tests the program continued as expected and further testing could be conducted.

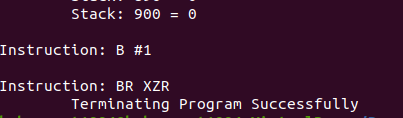
As mentioned a test was conducted to test the barrier set for the stack overflow error. This was done by manipulating the SUBI instruction on line 256 of the code3.txt file to make enough room on the stack to break that boundary. The stack, initialized to 900 with the minimum value defined to 400, was subtracted 500 placing the stack exactly at the barrier. This resulted in a stack overflow error as expected. The SUBI is executed, but the program terminates erroneously before any values can be stored in the stack from additional exectuitons of instructions.







Additionally, the SUBI was again altered to only subtract 496 from the stack pointer, placing it just before the barrier of a stack overflow error was reached, and the program executed correctly. This demonstrates successful implementation of the test for a stack overflow, although this test is primitive and dependent on human alteration of the definition at the top of the file.



Due to the successful testing of the code3.txt file, it is my belief that an appropriate amount of testing has been conducted to ensure that each of the instruction functions operates as intended. This is due to the difficult challenge of writing and revising not 1 but 2 files simultaneously. Due to not having a control file to work from, I was required to troubleshoot both the ARM2.c program and code3.txt files extensively when problems would arise. This includes a situation where for an hour I continuously had a segmentation fault that would erroneously end the execution of the program. Such faults could not be isolated to either of the two files, and both were required to be verified up to the point that the error had occurred for fear of a design error. The amount of time spent on this task is unknown, but substantial. Smaller code.txt files were written in an effort to isolate such issues to just the ARM2.c file, which lead to the eventual successful completion of this lab. These files tested specific functions only in very simple ways, proving to me that the program itself either held an error, or worked as intended and the error existed inside the code3.txt file that was submitted with this report.

1. **Results**

The results of this lab are a successful generation and execution of the code3.txt file by the ARM2.c program to the extent outlined in the lab page on canvas. The additional functionality was successfully implemented to the ARM.c program to include stack operations. The objectives listed in the lab page, and in the intent section of this report, have been completed through the requirement of comprehension to troubleshoot and implement the functions for executing the required branch instructions. Much time was required and spent analyzing the slides from previous lectures on the topic, as well as the textbooks assigned for readings, prior to the working implementation of the stack pointer and BL function.

The program has allocated stack space within the multidimensional **memory**[] array, initializing the stack pointer to the highest memory location available in the code3.txt file. This allows for the stack to downward towards 0 as it needs to store more and more data from the temporary registers in recursive function calls. The stack is also output, when utilized, similarly to the **programCounter** and instruction registers. The program successfully implements the SUBI, BL, and BR instructions, which were previously undefined from Lab4, as functions and have been demonstrated and tested to prove functionality. These branch instructions utilize immediate or register addressing as instructed. At least 8 registers are available to encompass all of the required registers, including the X28 stack pointer and X30 Link register. A “BR XZR” condition for termination of the program has been included and proven successfully functioning.

As instructed the stack is at least 100 words deep and care had been taken to ensure that the stack cannot and will not grow over the program/data area. This has been implemented in a way that can be altered by the grading individual if their code exceeds memory location 400. The program checks that the stack pointer does not exceed this limit, defined at the top of the file, and in the event that it does issues a “stack overflow” error message before terminating the program erroneously. The highest memory address has been chosen to allow for the requirement of 100 words to exist. The stack pointer is initialized to the top of the stack and, when used, is output to the terminal to include any memory locations that currently store data.

The test file that is required to be generated alongside this lab also conforms to all requirements expected upon submission. The file includes at least one example of each instruction implemented and is longer than 12 instructions long. Loop functionality has been added to this revision of ARM2.c, which differs from the prior program submission, and the demonstration of this has been added to the code3.txt file twice. A function call, a push, and a pop to the stack have been included in the file utilizing the BL, STUR, and LDUR functions, demonstrating comprehension of stack operations. The file provided does not contain recursion, which is not required, however the program is able to handle recursion as required,

1. **Conclusions**

Based on the results and intent of this assignment I conclude that I have successfully completed Lab5 to the best of my abilities. In implementing the functions and stack functionality required by the lab, I have gained a working knowledge of how ARM stack operations are conducted. I believe, as stated in the results portion of this report, that I have successfully accomplished the 2 major objectives of the lab.

The largest lesson learned from this lab was the process in which it took multiple hours to determine what was causing my inability to write to memory location 900 on the stack. A way suggested to me to reverse engineer this problem would be to write, in as small of English as possible similarly to pseudo-code, what the issue I am having is, then to walk away from the problem for an amount of time. This would allow a more objective look at the problem the next time I would approach the program and may have led to finding the issue easier than the process I had taken. I started troubleshooting thinking that the issue was going to be on a large scale, such as my design of execution was flawed, where the issue lay simply in a for loop test. I am lucky that I did not break any functionality of my program through that troubleshooting process.

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.

Logan Lipke assisted during the planning of implementation of the stack 29 Oct 2020