Sam Dinkelman

CIS 247

11/3/2018

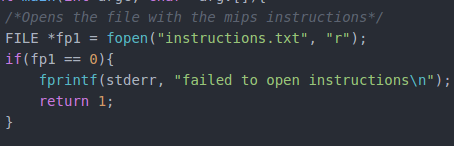
Lab report 4

Introduction:

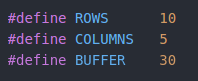
The purpose of this lab was to write an emulator for the MIPS assembly language. The emulator takes in a file of MIPS instructions and parses them out into a two-dimensional memory array. I also create a register array in order to correctly store values into registers within our program.

Process:

To begin this assignment I first had to plan my program in steps. I began by thinking of it as large main file where I read in the instructions using file pointers as I Ire shown in class. So my file pointer opens the instructions.txt file as a read only file and if it can't find the file it writes to stderr that it failed to open the file and exits the program. Here is a snippet of my code for this piece:



Next up I create a pointer to my memory array called memarray and made it a two dimensional array which is the size of the global variables I defined ROWS, set to ten for ten lines of instructions, and COLUMNS, set to five since that is the number of elements in each line that I will pass into the array. I also created another single dimension array called line which was of size BUFFER which was another global variable set to thirty. The size of BUFFER was determined by the fact that the longest line in the test file was under thirty characters so I considered thirty to be enough characters to grab in a single line.





After my two dimensional array was created I had to begin filling it with my file so I used the two standard library functions that I used in the last assignment, fgets and sscanf. I created a while loop that ran for as long as the index integer i was less than the BUFFER global variable and fgets was still getting lines. I passed the array line first to fgets which would be the first array that I feed my file into grabbing thirty characters at a time minus one which gives me all the characters without the newline character that fgets incorporates and stores that line into the array.



So here I have to feed the line array in my memarray using the format specifiers %[^,], which formats the input based on strings but delimits the commas and white space so that only the data at each location is fed into the memarray. Since the last column won't have a comma or space I leave that as a %s format specifier for string.



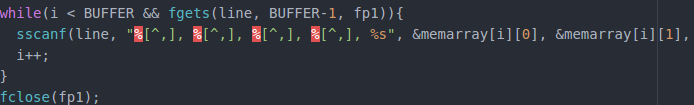
After this I feed the line array into the memarray at each address using the & and the index integer i so that I can go through the rows as I fill the columns zero through four.



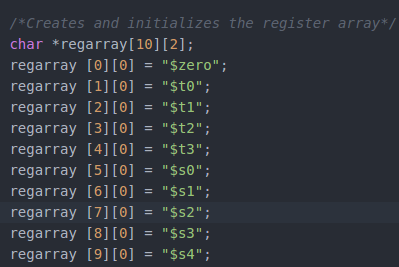
Next I increment my index integer i so that when my while loop runs it only grabs one line of thirty at a time and while it increments it stores the data at the correct row in memarray.



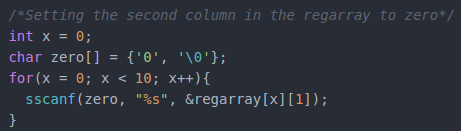
Here is this piece of code in full:



After this I create my next array which I decided would be easier to store and retrieve values and preserve the labels of the registers if I made it a two-dimensional array. So I make it ten by two in order to have the zeroth column be the labels of the register and the first column be the value stored in that register. Here I have all my registers:

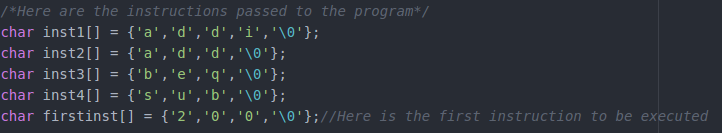


Next I had to initialize the second column of the array so I decided to do this through a for loop that would take a character array of zeros and fill the second column of the array with them. So I first created my array of zeros with a null terminating character at the end and wrote a for loop to sscanf my zero array into my regarray at the xth row and the first column.



Now I have both of my arrays created, storing instructions and registers and ready to begin reading and writing in order to execute the instructions. This was where I was stuck for the most amount of time trying different ways to pass instructions around and identify which instruction it was currently executing.

So I decided to create character arrays for each of the instructions I would include in my MIPS file and compare these strings to the ones found in the file. Below are the instruction strings I created and that are included in the MIPS file I wrote. The last one first instruction is the address where I will begin executing instructions in my code.



After this I begin heading into the biggest part of my program which consists of many for loops with if statements that compare strings to each other in order to get the correct instruction and registers and pass them to the functions which do the math.

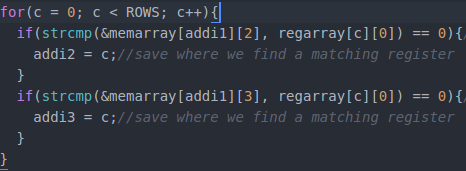
Below is the first for and if statement the for loop simply goes through every row, since I decided to have ten rows and ten registers it worked out very Ill for the rest of the for loops. This first if statement compares the first column of the memory array memarray to the string 200 which is the memory address of the first instruction to be executed.



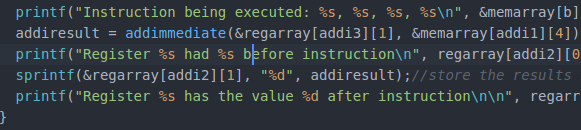
Here I begin my for loop that will run through my whole program and act as a program counter using the index integer b. After this for loop I head into an if statement that compares the string in the second column of the memory array to the string stored in the character array inst1 which is the instruction addi. Then if this evaluates to zero which is true then it will jump down and save the location of the program counter where I find addi.



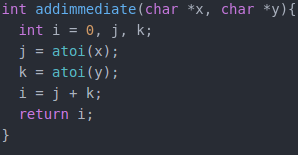
After this I head into another for loop that iterates through the rows of the program again to find the registers stored at this line and store their locations. I do this through more string compares which compare the strings stored in the whole of the register array in the first column which should be the label of each register to the third column of the memory array. When it finds a match I save the location in another integer, in this case addi2. I do this again for the fourth column of the memory array. I don’t have to perform a lookup on the fifth column of the memory array since it will be a constant in the case of the addi.



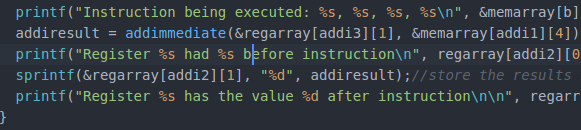
After all these lookups are complete I move on to giving the functions the proper elements of the array using the lookups I just performed. So I pass the addimmediate function the register array at addi3 which is the lookup for the line of that register and I pass it the second column which should be initialized to zero so I can directly store the values I calculate into it. Then I pass it the memarray at the addi1 which is the line I are on and the fifth column which will be the constant number since this is an addi. I don’t really have to pass it the third column, which would be the regarray at addi2, since I don't perform any calculations on that register. I also store this function call into an integer addiresult so that the value it returns is stored.



Now in my addimmediate function I kept it very simple. It takes two pointers in which I passed back in main which should be the string of the register, x, and the immediate value, y. Then create three integers, I take these three integers and set them equal to the value of the strings after they have been turned into integers using the standard library function atoi. Then I take those two values j and k and add them together, store them in i and return i.



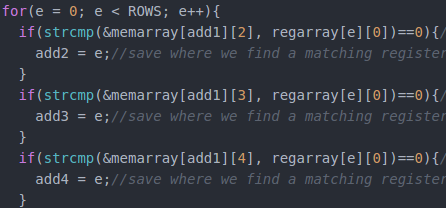
Back down in main I store the result of this function into addiresult and then pass it off to another library function called sprintf which accomplishes two things. First it converts the integers back into strings and second it stores them back into the second column of the register array at the correct line for the register that will hold the result.



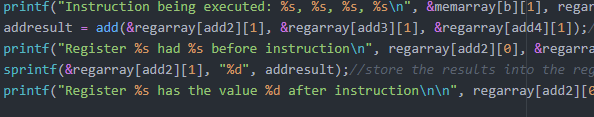
Okay so moving into my next function I have the same basic set up which starts with an if statement that compares the strings stored in the second column of memarray to the second string of instructions. Then I store that location in add1.



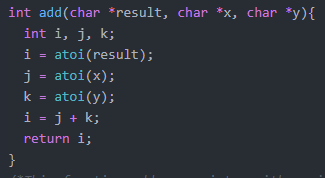
Then I head into another for loop full of if statements that have string compares in them. So here I are looking through each element at this line that I have grabbed and comparing it to the entire first column of labelled registers in the regarray.



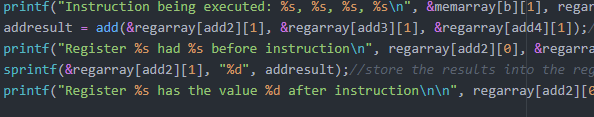
After this we pass the register array to the function at the saved locations we found, passing it the second column which is full of zeros or any previously calculated values.



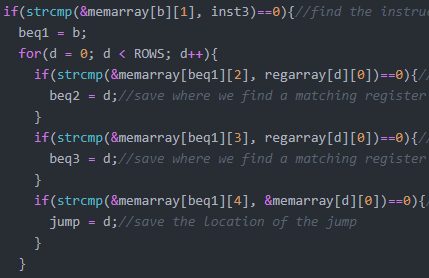
Up in my add function I get passed three character pointers from main for the result register, the first operand register and the second operand register. Now we declare three integers i, j, and k and set them equal to the result of converting these three characters to integers using the atoi standard library function. After that we add our two operands j and k to each other and store the results in i and return it.



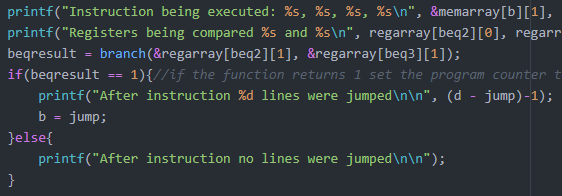
Back down in main we use the sprintf again to feed the returned value of this function back into the register array at the second column where we found the results register.



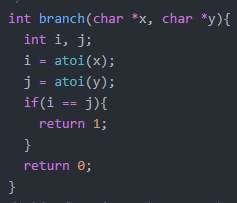
Moving right along we have another similar block of code where we compare the memarray to the third instruction which is branch on equal, when it finds this it will save that location for later lookups. Then we move into a for loop which goes through the rows of registers and checks to see if any on this line are equal to the strings stored in the memory array and if they are store that location for passing to the function. Since we get a memory address in the fourth column of a branch instruction my fourth lookup compares this to the first column of the memory array to see where this memory address is located in the instructions and saves that location.



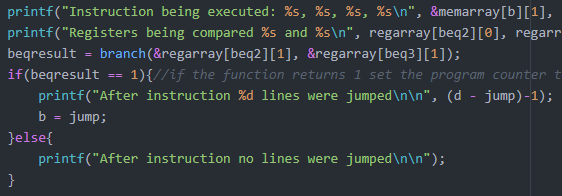
Now I pass my branch function the first and second registers that I found so that it can compare them and store the result of that comparison into beqresult.



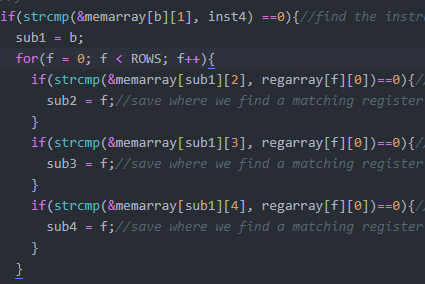
Up in my branch function I get the strings and again convert them into integers in order to compare them. So I use ints i and j again and store the results of using atoi on x and y into them. Then below I have an if statement that says if i is equal to j return 1 and if this does not evaluate to true this function will return 0.



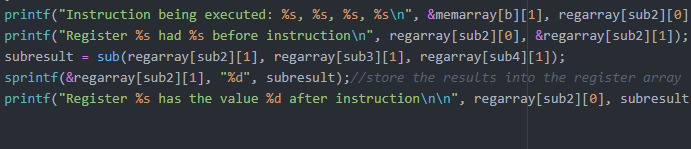
Now back in main I have the code which then takes this result and since we don't have anything to store it into in the regarray I instead say if the result was 1 then make our program counter b go to wherever jump is was calculated and stored earlier.



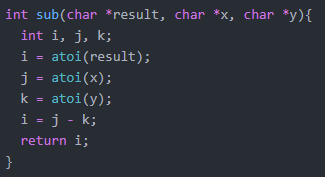
Now for my last function I have the same logic once again where I take the string that stores the instruction this time subtract and compare it to the strings in the second column of memory in order to store that location. Then I again go through comparing each element of the memarray to the regarray to lookup the matches and store those locations.



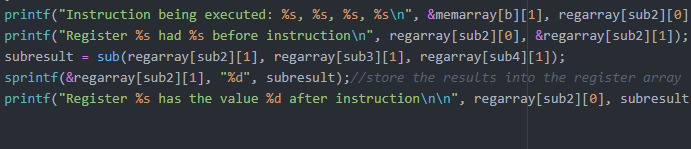
Now I take the register array and pass it to the subtract function at the second column of the line which holds the correct register in order to perform calculations on it.



Looking up at my sub function I again take it three character pointers like in add and make int i, j, k and set them equal to the result of running the atoi command on the parameters passed. Then I take those integers and subtract j from k and store that result into i which I then return.



Back in main I store the returned result into subresult and take that and pass it into the second column of regarray where the correct result register is located.

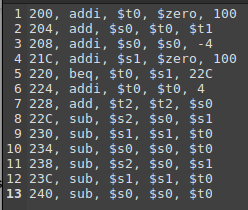


This completes my MIPS emulator which runs the four functions addi, add, sub and beq.

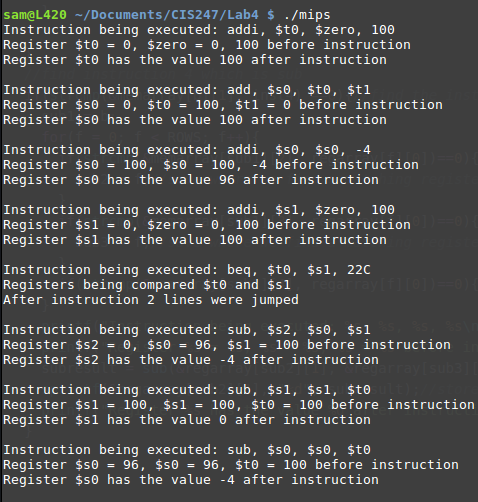
Testing:

For testing my code the first thing I wanted to try running against it was a MIPS instruction file that had more than ten lines of instructions. Since I defined all of my many for loops as being a variable less than ROWS which I defined globally as 10 I think this will probably break my program.

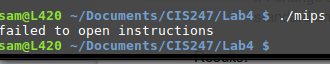
So I ran it against this file



And the result was that it just stopped after ten lines of code and didn’t run anything past that, which isn’t exactly breaking but doesn’t function properly. Here are those results:

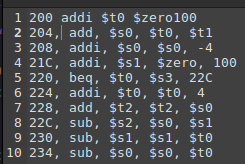


If I change the name of the instructions file that I run it against it is supposed to write to standard error that it couldn’t open the file, next I will test this



This does work as I intended it to work.

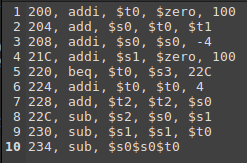
Next I wanted to test if I took out spaces and commas from my first line of code what it would do. So I ran it against this file:



And here are my results:

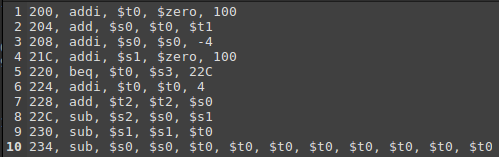


So when I ran it against this file it wouldn’t even run or execute any instructions. Now I’m going to try doing that on the last line and see if it will at least execute up until the point where it can’t recognized input. Running it against this file

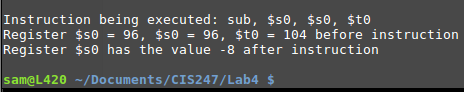


Makes it so that every instruction is executed except for that last one and I’m guessing if I put it in the middle the program would just halt on any unexpected input being read into my memarray.

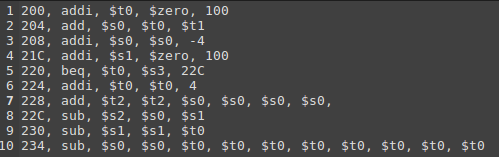
Since I have that BUFFER of 30 that reads in each line of 30 characters I want to try to give it a line with characters over 30 and see how it handles it. For this I will run it against this file



My results for this were actually pretty surprising, I expected it to possibly keep trying to subtract $t0 over and over or just crash and exit the program but it looks like my buffer just scrubbed off the extra code and ran it anyways as normal. Here is the last instruction being executed



Trying this same thing in the middle of my MIPS instructions did give me a strange error though. I ran it against this file



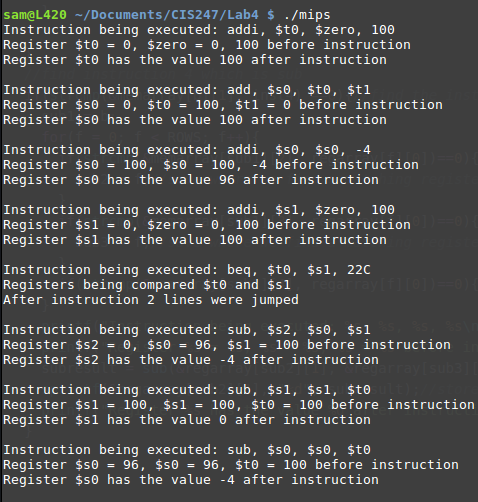
For some reason it mistook $s0 in the add on line 7 as a $t1 and returned this result:



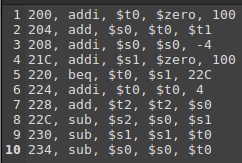
This is very strange and I’m not sure why it would do this exactly but it seems to be a pretty substantial error.

Results:

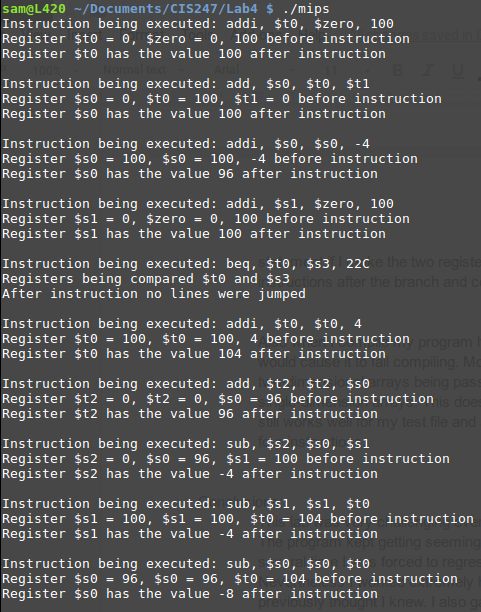
Below are the result of running my code against a file I generated for it to run:



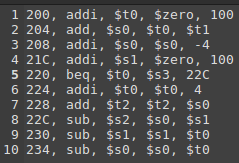
And here is the file that I ran against it:



As you can see from my print statements my code performs the functions as specified in MIPS, it acts exactly as a MIPS program would run. In my print statements it only prints for 8 instructions this is because of the branch on line five which jumps two lines of code and then begins executing instructions again. Since I have a branch on equal statement if I make the two registers that I am comparing not equal it will run the instructions after the branch and continue on with the program. As demonstrated here:



And heres the file I ran it against:



Also when I compile my program has a lot of warnings and notes but no errors that would cause it to fail compiling. Most of these warning relate to format errors or two-dimensional arrays being passed to standard library functions that are expecting single-dimension arrays. This doesn’t stop my code from running and compiling and it still works well for my test file and any file under ten instructions that use only these four instructions.

Conclusions:

This lab was very challenging even with the two weeks we were given to complete it. The program kept getting seemingly larger and larger as I progressed through it, and several time I was forced to regress in order to complete certain functions. Nevertheless I worked extremely hard on this lab and learned more about MIPS than I previously thought I knew. I also gained a broader knowledge of how to work with pointers and pass between functions properly, as well as reading in files and parsing them into arrays. This lab also taught me how to properly use two-dimensional arrays which proved challenging but useful and I now see how implementing them could solve lots of problems. Overall I found this lab to be a sufficient challenge to our level of knowledge at this point in the course.

References and Acknowledgements:

Over the course of this assignment I worked with a lot of other student in order to properly understand the assignment and work through my ideas and assist others in formulating theirs. I also consulted stack overflow several times in order to get things like delimiting the commas and white space and also to get my two dimensional array working. However all of this code was my own work.