Sam Dinkelman

11/11/2018

CIS 247

Lab report

Introduction:

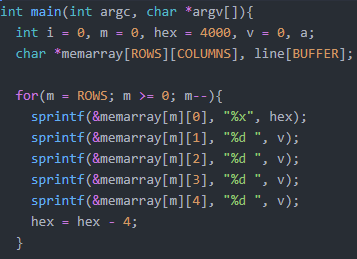
The intent of this Lab was to expand on our earlier project the MIPS Emulator. From the original lab we needed to have a fully functioning program that would read in instructions and perform operations on registers and immediate values. In this lab we were required to implement specific instructions such as, subu, sw, lw, li, add, addiu, jal, jr, bgtz. We were also instructed to implement a stack and push and pop registers to and from our stack.

Process:

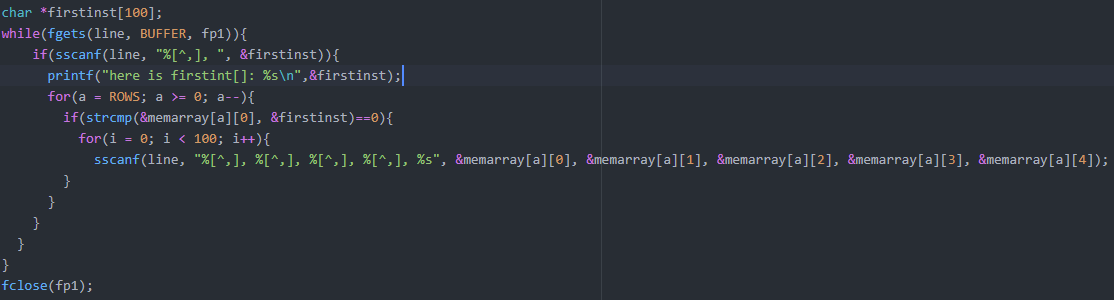
For this lab I began by building off of my previous lab program, which was working as intended but had lots of warnings when compiling. I didn’t fix most of these warnings since they were mostly pertaining to formatting and other trivial warnings.

Since I started with an already built program I won’t be covering my process for the first part of this lab I will begin with where the second lab starts.

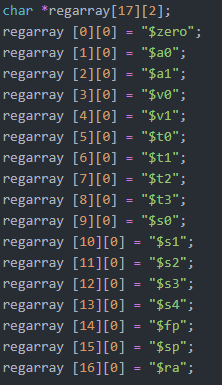
My first step of this lab was to get the Instructions from the file into the memory array at the correct locations for their respective memory addresses. To accomplish this I first created two new integers one for hex which was set to 4000 and one for v which was set at 0. I then changed the global constant of ROWS from 10 to 1000 to hold all of the memory we would need. Then I created and initialized the memarray array by fill it with hex values in the 0th column and zeros in all the other columns as seen below. I went from 1000 to 0 subtracting 4 from the hex value each time to fill it with the correct hex values for 1000 memory addresses.



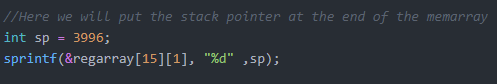
After this I change my while fgets loop from the first lab by first creating a new array of 100 in order to feed the instructions from the file into this array. Then I do my same fgets while loop and sscanf the lines of the file into that firstinst array which will hold the instructions from the file. I only parse out the 0th column of the instructions file because I want to fill this new array with the memory addresses and compare them to the addresses that we just put into my memarray array. So then I start a for loop with an if statement to go through the whole memory array and compare the hex addresses in the memory array to the memory addresses of the instructions. When we find matches we sscanf them into the memory array at the correct addresses. Here is my code for that bit:



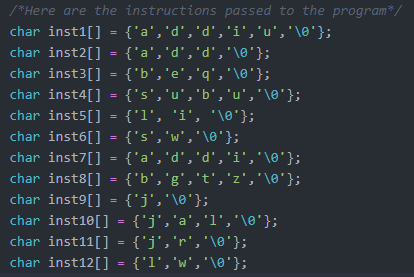
Next I expanded my register array, regarray, to include all of the registers we needed to add for this second part of the lab.



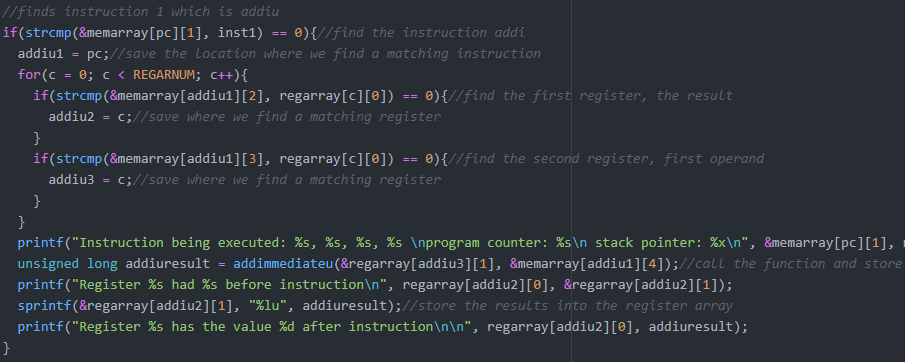
After this I began to work on my stack pointer but I didn’t get very far since I ran out of time to work on the lab. But here I create an integer sp and set it equal to the last hex address of the memory array that we fed in earlier. And I sprintf to the $sp register the value of that integer to set the stack pointer to the end of the memory array.



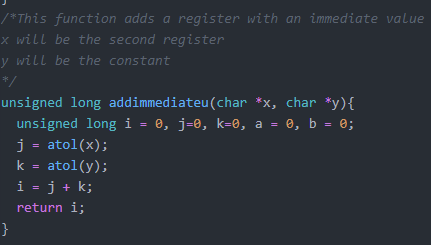
Here I have the same basic setup as the last lab where I create arrays of characters and then compare them to the instructions we find in the memory array. Here are the character arrays:



Then I converted some of my earlier string compares from the first lab to be some of the new instructions we had to support for this part of the lab. I started with changing addi to addiu. I go through the memory array and check for the instructions matching and then when I find them I perform a lookup for the registers and pass those to the function.



The function addiu is the same basic form as addi it just uses unsigned long integers to hold the values instead of regular integers. I also use atol instead of atoi in order to convert the strings to longs instead of ints.

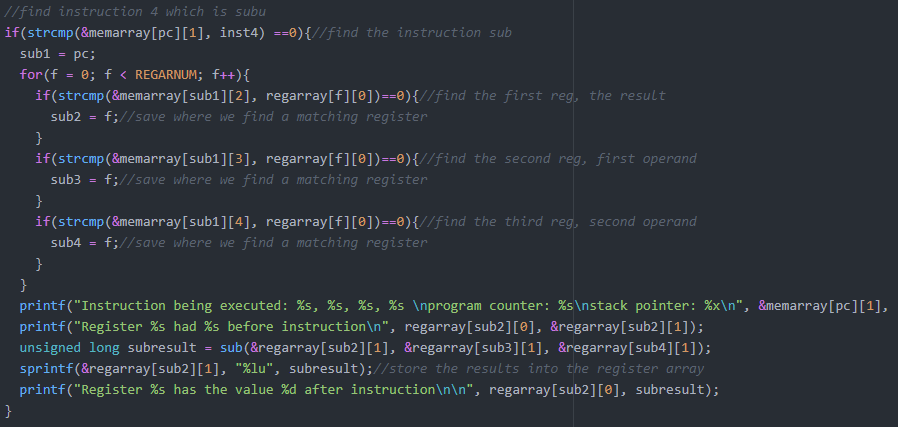


I then take the value returned by addiu and store it back into the register array using sprintf

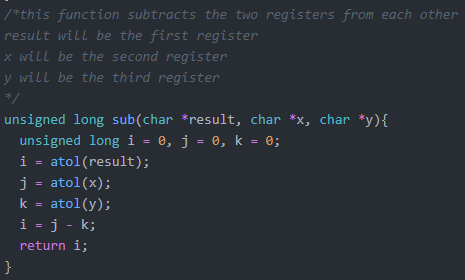


I had previously implemented the instruction add in the last lab so I wont cover it here since I covered it previously.

Next I converted my sub from the last lab into subu in this lab which again just like addiu uses unsigned long ints instead of just regular ints. I perform the same lookups and passes to the function.



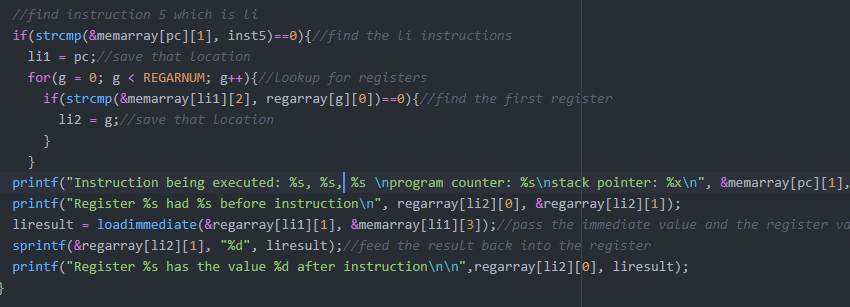
In my subu function it is basically the same as before just with computations performed on unsigned longs instead of ints



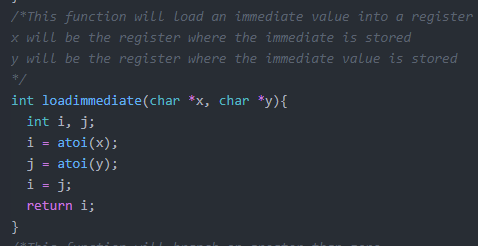
When I return the value I again sprintf it back into the register array.



After this I completed my load immediate function which was very simple. I do the same lookups for the instruction and for the register since there is only one it was very quick. Then I pass that register and the immediate value off to a function.



The function takes in the immediate value and the register value and simple converts them to integers and sets the register equal to the value of the immediate

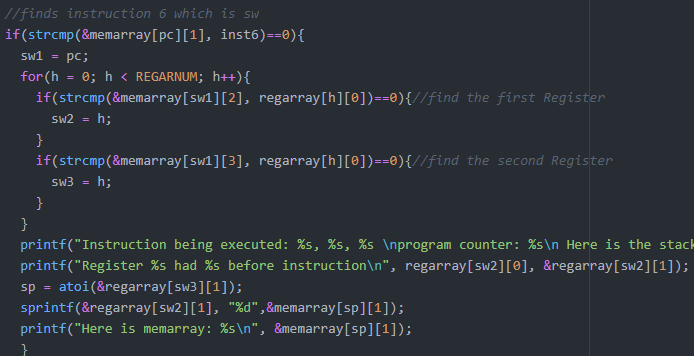


Then I take them and sprintf them back into the register array at the correct register location.



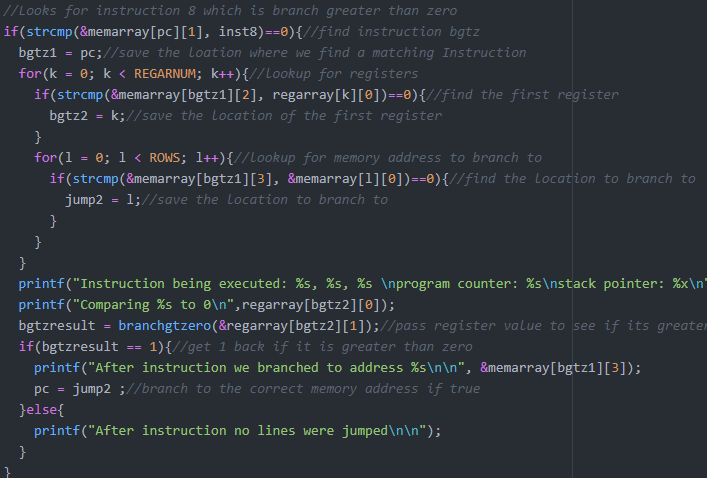
After this I began to work on my Store Word and Load Word. I didn’t finish these functions however and thus I cannot push and pop my stack and couldn’t fully implement my stack and get it working. But I will show my process and explain what I would have done if I was able to complete it.

So in my sw I setup the same basic lookups for the instructions and the registers saving the locations where I find them. Next I take the stack pointer and set it equal to the integer value of the location where we found the third register which is the value we will be storing into memory. Then we simply feed that into the location in memory where the stack pointer is located.

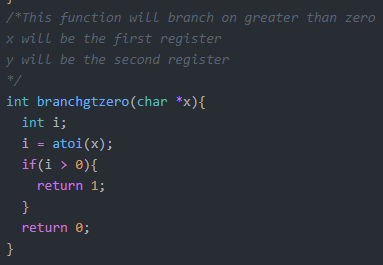


Unfortunately I was unable to get this working with my stack pointer in time which means that I cannot use sw or lw and I cannot push and pop to and from my stack.

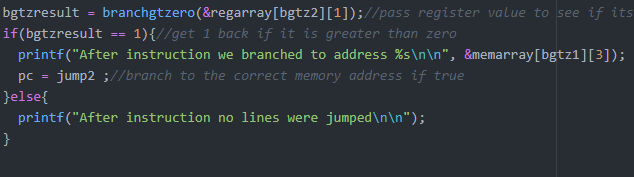
Next I worked on the bgtz instruction which was similar to the beq instruction in the first lab. I start with all the similar lookups but instead of looking for a second register I find the location of the jump we are going to perform. I then pass the value of the first register to the function bgtz.



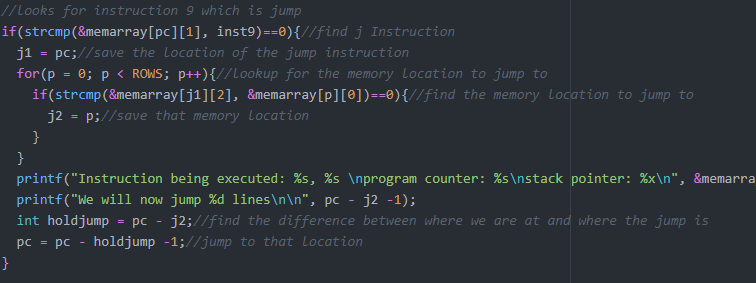
In my bgtz function I simply pass one value for the register into it and turn it into an integer. Then I do an if statement that says that if that value is greater than zero return 1 and if not then return 0.



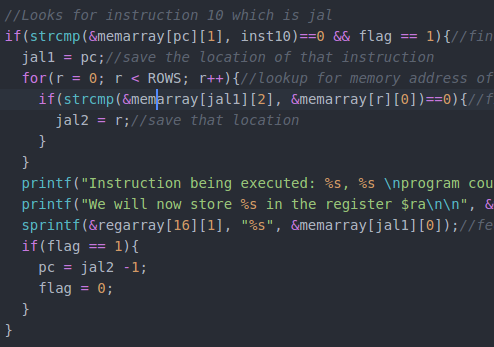
Back down in main I take that value and state that if it returns 1 which means the value in the register is greater than zero then set the program counter equal to the location of the jump that we calculated earlier. This works as long as the branch is above the jump instruction since whenever branch above the jump instruction it will get stuck in an infinite loop.



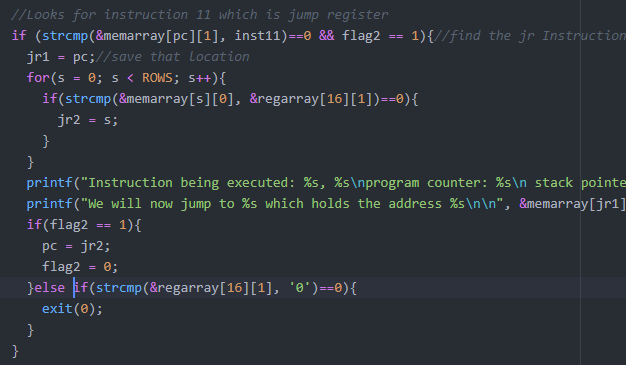
Next up is my jump instruction which is very short but works. I do the same lookups for the instructions but instead of looking for any registers I look for the location to jump to by comparing the jump location to the memory addresses. I then save the location and below I create the integer holdjump to be equal to the program counter minus the jump value. Then I take the program counter and set it equal to the program counter minus the difference between where the program counter is set and the value of the jump and subtract one to even it out.



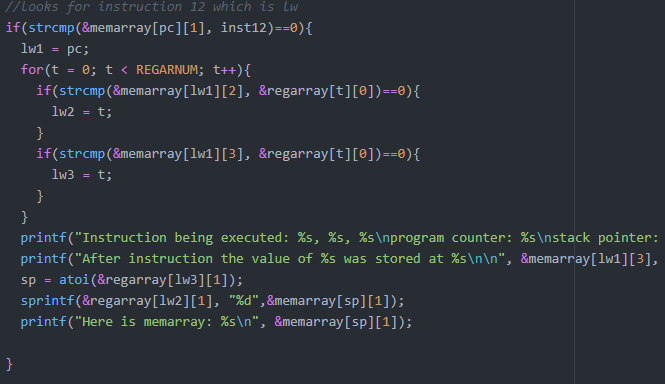
Now we have jump and link which will jump to a location and save the next instruction after it to the register $ra in order to jump back to the instruction. For this I do the same lookups as before where I find the instruction and the location to jump to. Then underneath I feed the location of the next instruction into the $ra register. Before I start my program counter I have an integer called flag that I set to one. This instruction will only happen once since I always seemed to get stuck in infinite loops if I didn’t set the flag. So below my lookups I have an if statement that says that if the flag is one then set the program counter equal to the location of the jump and then set the flag to zero so that this instruction can only run once.



After this I have my jr instruction which just like jal can only be run once since the second flag which I set to one at the top of my program counter has to be one to run and after it is done it will set it to zero. So I do the similar lookups for instructions and then I lookup the address that is saved in $ra register and compare it the memory addresses and save the location. Then below I have the if statement that says if the flag is equal to one then set the program counter to the location of the jump and then set it to zero. Again this will only work once in instruction I couldn’t figure out a way to get it to jal and jr multiple times. I also have an else if statement that says if the $ra register holds the value 0 then exit the program. This works but it doesn’t work gracefully and I couldn’t get it to simply exit without getting a segmentation fault, but it does stop the program.



Lastly I have my Load Word instruction which again doesn’t work which unfortunately means I can’t push and pop my stack or anything. This function would have been basically the same as the Store Word function just with the order of the sprintf switched around so that it would put the values from the stack into a register.

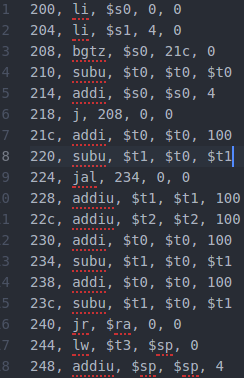


That concludes my process for the second portion of this lab.

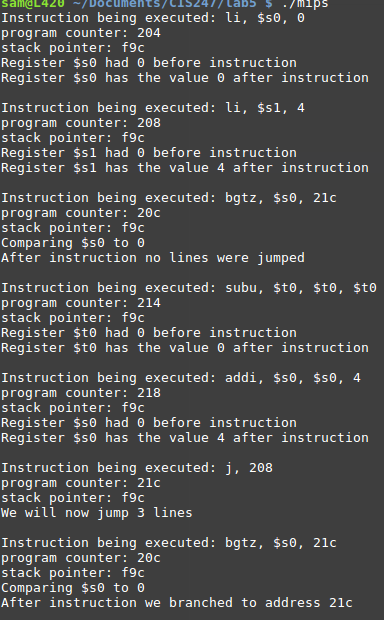
Results:

My results for this lab are varied, I implemented most of the functions but couldn’t get my stack and sw and lw working properly. As such I have included one lw instruction and addiu to the stack pointer in my instructions file just for testing purposes. But it doesn’t behave as it should. It should also be noted that my code does compile but it contains a huge amount of warnings that should be ignored by using the -w option in the gcc compiler.

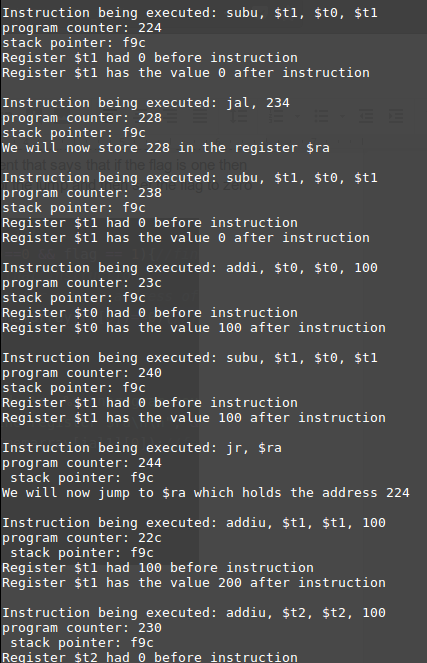
Here below I have the output of my code from running the instructions below:



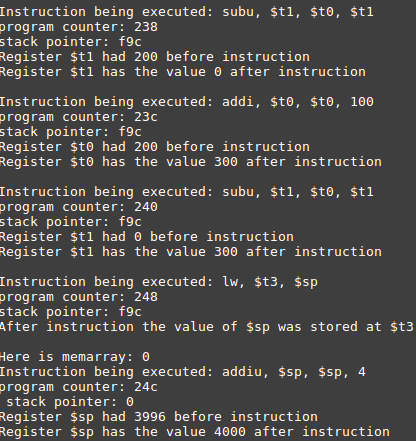
So here we can see that I have my load immediate function working and I load 0 into the register $s0 and 4 into the register $s1. My program counter points to the memory address of the next instruction to be executed. And my stack pointer is pointing to the last address in my stack. Further down we can see that my bgtz instruction checks if $s0 is greater than 0 which it is not so it does not branch. Then my subu was just a space filler because next I do an addi to add 4 to $s0 then jump back to the bgtz instruction which compares $s0 to 0 again and finds that it is greater than 0 so it jumps to address 21c which is just after my jump instruction.



Down here we can see that my branch actually didn’t work and jumped the next instruction which should have been addi but I couldn’t get this to work. Next I do my jal instruction which jumps to instruction 234 which is a subu instruction. Then we run through a few more instructions which are place holders to get to the jr instruction which jumps to the address of 224 which is immediately after the jal instruction.



Next have some subu instructions which work as expected since some of my previous ones weren’t working on negative values which make sense since it only wants to operate on unsigned numbers. Then we go down to the load word function which doesn’t work as intended.

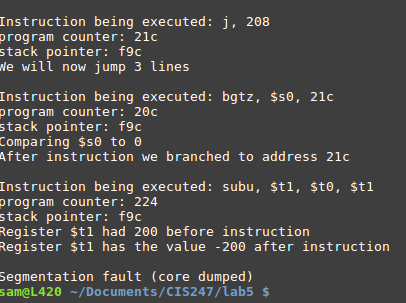


Testing:

Testing my program I had a lot of warnings and notes when I compiled it which I ignored and it works decently well if I just ignore all of the formatting error and such but I would have liked to have fixed those. My load word and store word functions don’t work properly as well as my stack pointer which doesn’t do much. Also my subu doesn’t like values that go negative and will return positive numbers if it thinks it’s going to go negative as you can see in the examples above. Another large error in my program is that my branch will branch one instruction past where it is supposed to go, I couldn’t get this error figured out since I just discovered it as I was writing this report.

Since there are so many errors in my program it is hard to test for bugs since there will be so many. One large bug is that I don’t have any bounds checking on my stack so it could grow over my instructions however it doesn’t even grow so theres not really a chance of that.

Here I found that when I got a negative value from subu it would segmentation fault my program and it would crash. This is a huge error and a way I could fix this would be to have bounds checking on my subu function and throw an error if it goes below zero since unsigned will only want to do operations on non-negative numbers.



Also my exit on jr = 0 doesn't seem to work and simply jumps us back to instruction 0 which will run through all of my instructions again. I also found that when I would have a branch to an instruction that wasn’t there it would just go to another instruction past jump and I have no way of explaining that behavior. Also if I had an empty line it would give me a segmentation fault and close the program as well. Overall my code was very very buggy and this made testing it both easy and difficult.

Conclusion:

Based on the results and the intent of this assignment I don’t believe my code was up to par with the goals of this assignment. Since this code carried over from the first part of this lab I had to fix many many bugs in it and had to also stick to the systems I had set up in the first lab to complete this one. This worked for a while but I kept hitting walls where my choices to perform a function in the first lab wouldn’t work for this second part. I think this was a good lab overall however since we had to learn how to work with code that was already written by ourselves and try to expand and build on it. I feel like it mirrored the real world very well since we will often be given code and told to improve it rather than build from scratch.

References and acknowledgements:

In this lab I worked very closely with a few classmates including John Traner and Scarlett Anderson. We spent close to thirty hours on the first part of this lab and probably around the same on the second part. We all bounced ideas off of each other and helped each other when we were stuck. I would like to acknowledge them here for their help and support during these labs.