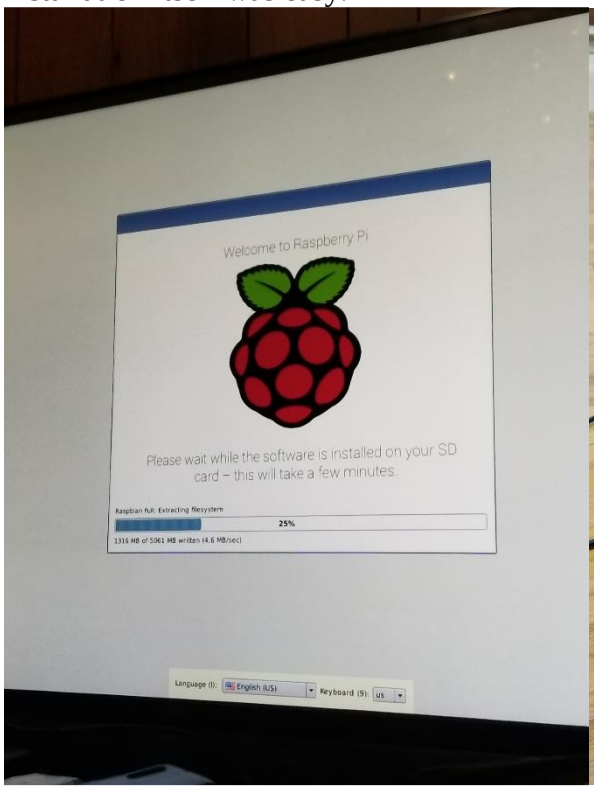
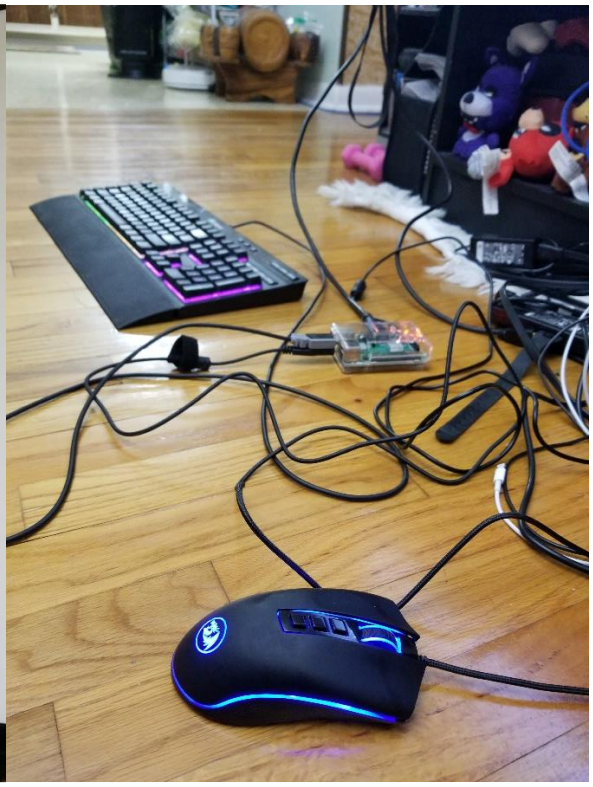
Task 5: Arm Assembly Programming

Preface/Setup:
After having received the Raspberry PI from our coordinator, I noticed that the Raspberry PI would power on, but not boot. This either meant that I had a faulty HDMI cable or there was no operating system on the micro sd card. I assumed the latter and downloaded NOOBS from the raspberry pi website and reformatted the micro sd card to FAT32 (I think by default it was formatted to FAT16 but was completely empty). NOOBS itself was around 2.1 gigabytes. After unzipping to micro sd card, I attempted to boot up the Raspberry PI again and this time it worked. Choosing Raspbian (FULL) as the OS, the installation took about 40 minutes, probably due to slow reading and writing capabilities of the micro sd card. Regardless, the installation itself was easy.





NOOBS was enough to get Raspbian installed. I didn't need to use Sleezy or any extra steps- just download the latest version of Noobs, extract to FAT32-formatted sd card, and mount it into the PI. Done.

My workstation was suboptimal, but got the job done.

Part 1:
Raspbian is a pretty smooth operating system. Having used resource-heavy, feature-rich desktop OSes for a very long time (My personal computer runs Windows 10 Enterprise LTSC 2019 and a Debian virtual machine), It was really interesting to see how bare-bones Raspbian was. From the first log-in, there really isn't much to set-up besides the password and Wi-Fi, whereas Windows requires a plenitude of settings-tweaking and permission-granting. This operating system really feels like it was designed with minimalism and power-saving in mind.

Now with that out of the way, working with terminal was pretty straight-forward (I have created BASH scripts before so this terminal wasn't exactly foreign--ls, cd, mv, cp, grep, sed, awk, and the compilers work fine. I am completely new to Assembly however, so I had to run the following code without too much knowledge of what the headers meant:

```
@first program
.section .data
.section .text
.global _start
_start:
        mov r1,#5
        sub r1,r1,#1
        add r1,r1,#4

        mov r7,#1
        svc #0
.end
```

The function of the header of this code is still unknown but the assembly instructions under the _start: line are parse-able:

mov r1,#5 basically assigns the value 5 to register r1.
The syntax would be *mov destination,source.*
sub r1,r1,#1 subtracts 1 from the register r1 and stores it into r1. The syntax would be *sub destination,operand1,operand2,* where operand1 minus operand2 is stored into the destination.

add r1,r1,#4 is the same thing, except instead of subtract 1 from r1, 4 is added to r1 and stored into r1 as well.

mov r7,#1 is just like mov r1,#5, this time using the immediate operand 1 and assigning it to the register r7.

svc #0 -- this one was new so I had to perform a quick search on what the svc function does. It apparently stands for (Supervisor Calls), and is usually followed with a number called the SVC number (in this case that would be zero). It's mainly used to "interrupt"the operating

system to make a request. For all intents and purposes, this seems to act like a termination/break point or pause for our debugging program so that we can see what happens in the previous lines.

I wrote the code using the nano command, but I'm sure other ways of writing it were possible (like the cat command or vi editor). Then I compiled and linked the program as requested. Then I was asked to run the following command:

./first

Normally this should execute some type of compiled program (I used this command a lot when writing in C to execute the program in terminal), and it didn't cause any errors, but it also didn't yield any output.

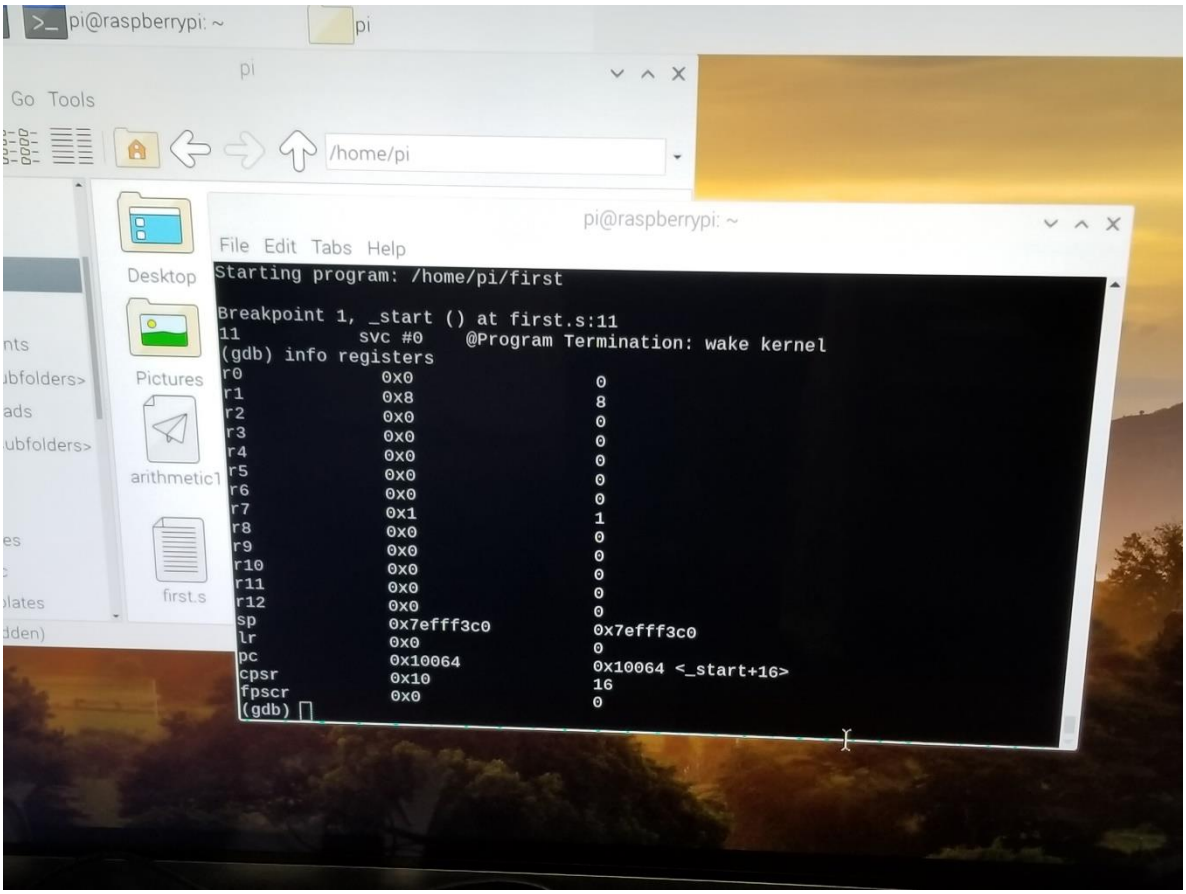
Did you see any output, why or why not?
No. The best explanation would be that none of the assembly instructions explicitly say to print what they are doing to the registers.
For example, "*mov r1,#5*" does modify the value inside of r1 but it doesn't invoke a command to print anything like "Hello human, I did what you asked me to" to the terminal.

So the only way we can really see what is going on in the program is to run it step by step in debugging mode, which was the what the
next steps were.

The debugger of choice was GDB (GNU Debugger) and it has some very useful features (in particular, setting break points where the program should pause, and also listing the current values of the registers). Using it for the first time, a dialogue popped up about the GNU license, maybe as a sign of precaution. typing in the "list" command was useful, but displaying only the first 10 lines and every 10 lines afterwards was not the most efficient way of doing things. I added two operands after. making it "list 1,15" as in showing the first 15 lines.

The break point allows us to "freeze" the program in the middle of execution. We want to see the result of our mov, sub and add instructions so the break point was set after them on the 11th line using "*(gdb) b 11*."

Now I could safely run the program up until line 11 using the "run" command, and during this time we can assess the final values of the registers using the "info registers" command.



As seen in this snippet of the output, the value stored in r1 was 8, and r7 was 1. This makes sense because we first set the value of r1 to 5, subtract one to make it 4, and then add 4 to make it 8. Then we simply assigned the immediate value 1 to r7. What surprises me, though, is unlike in Visual Studio, the rest of the registers are assigned the value 0 as the default. In Visual Studio the values stored in the registers **are initialized as random hexadecimals**. This brings up a question that I hope to one day answer--why do some debuggers assign random values to registers, and why do some debuggers assign the value zero?

Part 2:
Our next task was to implement A = (A+B)-(C*D) in ARM Assembly, using registers as variable names.
To make things more assembly-friendly, I first assigned A, B, C, and D registers and rewrote the problem.

r1 = (r1+r2)-(r3*r4)
where r1 = A, r2 = B, r3 = C, r4 = D.

Looking at the right side of the equation, the task requires two sub problems to be resolved first, r1+r2 and r3*r4. Then, taking the resultant of the first sub problem and subtracting it with the resultant of the second would give us our final value of r1.

r1+r2 in assembly is as simple as
add r1,r1,r2 -- I prefer using three operands to be absolutely sure that the destination is r1, and that we are using r1 and r2 as the arithmetic inputs.

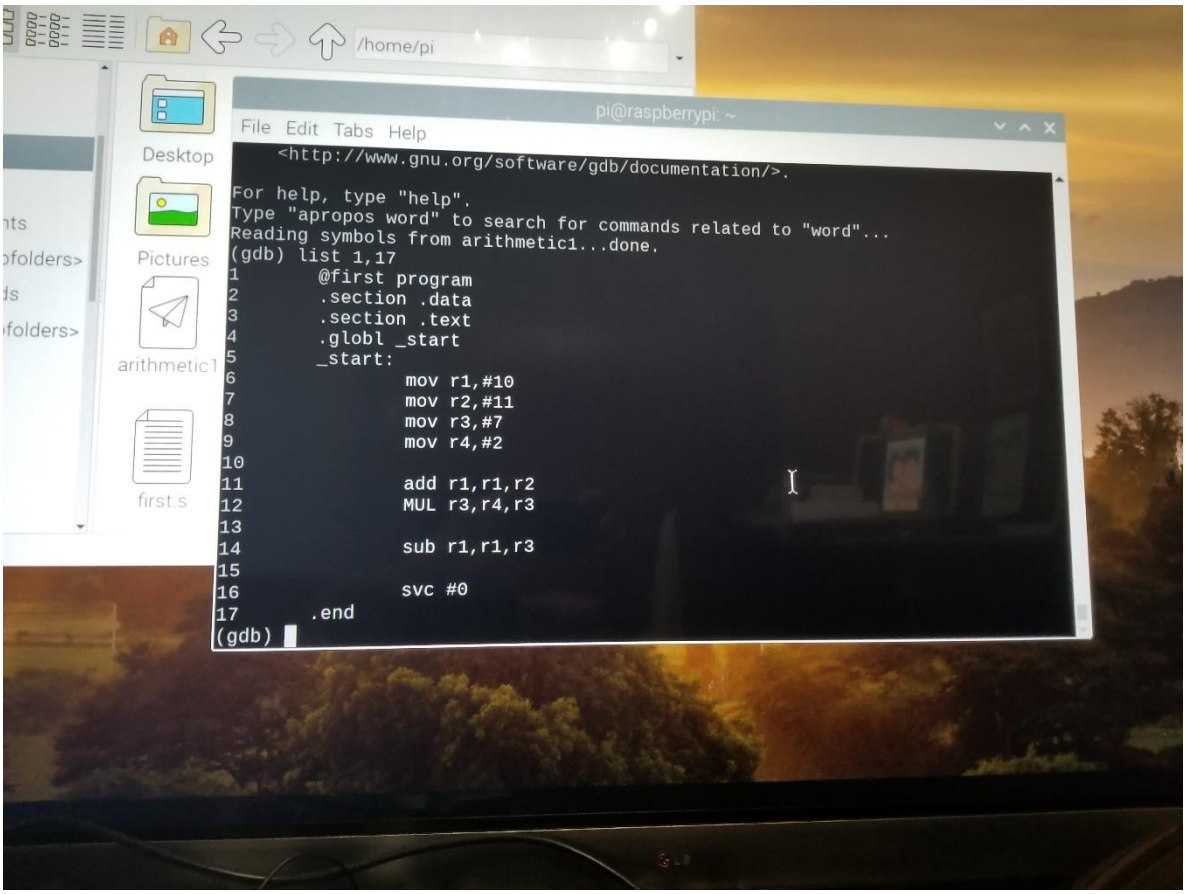
r3*r4 is the same format, but I used the "MUL" instruction to multiply, and the operands are destination,multiplicative1,multiplicative2.
MUL r3,r4,r3 -- *MUL* is used here instead of *IMUL* because we are given the information later that we are multiplying in the positive domain, so unsigned integers will work fine.
MUL doesn't like operand1 being the destination, so I just swapped the multiplicatives.

The next problem I had to think about was what the order of the operands would be: *sub destination,operand1,operand2* performs the operation operand1 minus operand2 and stores it into destination, so it goes like this:
sub r1,r1,r3 -- This makes sense because r1 and r3 are the resultants of the sub problems, and r1 is the destination because in the original substitution, A = r1.
However, using the subtraction instruction now requires us to look for the sign flag to be thrown in case r3 is greater than r1. In other words, there is a risk of r1 being a negative number.

At this point the logic of the arithmetic is implemented but the values themselves are not initialized. According to the task, we had to assign 10 to A, 11 to B, 7 to C, and 2 to D. All we need to do is use the mov instruction at the beginning of our assembly program with immediates:

mov r1,#10
mov r2,#11
mov r3,#7
mov r4,#2

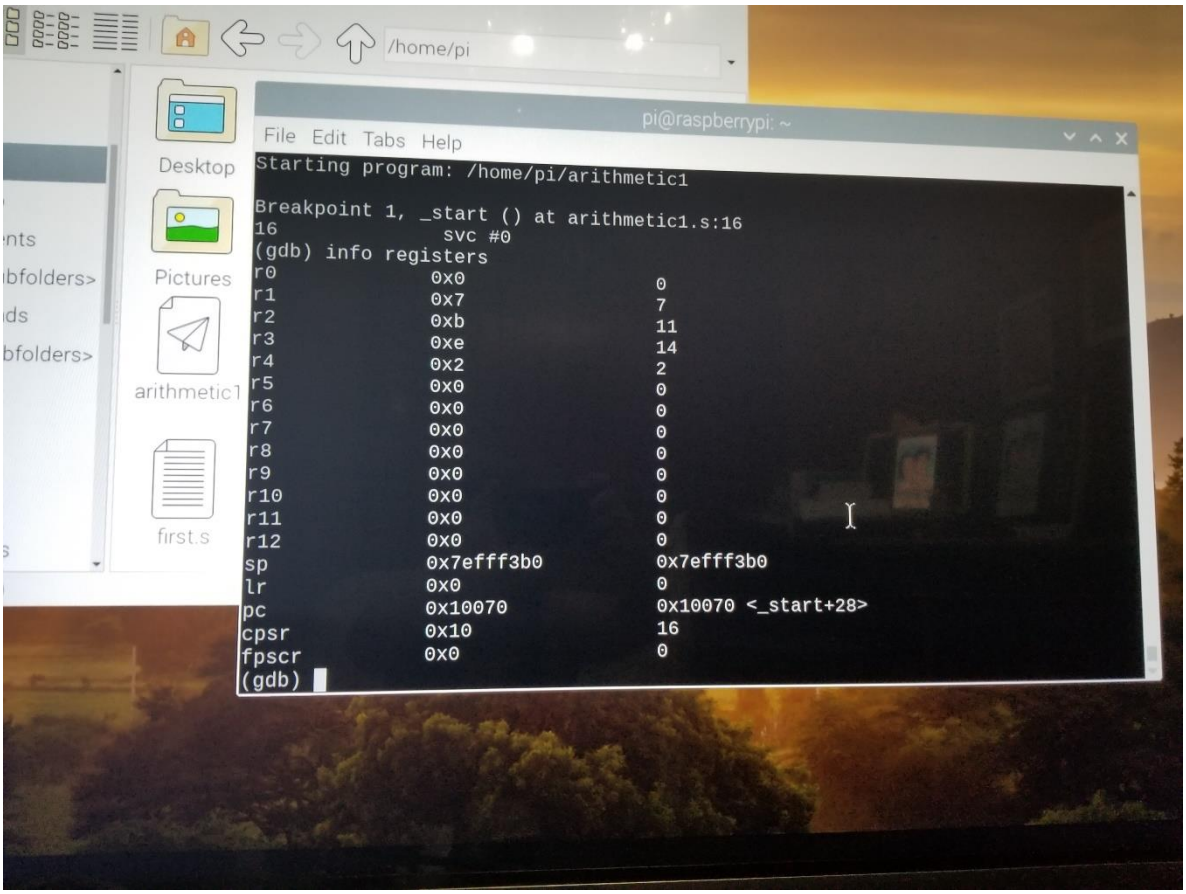
Thankfully, due to these values, our initial worry about having to keep up with the sign flag is gone, because if we compare the resultant of the two subproblems:
(10+11) and (7*2) = 21 and 14
We see that the former is greater than the latter, so the resulting value of r1 is positive, we don't have to worry about the sign flag.



r1 = 21 - 14 = 7. **Our predicted value of r1 is 7**. For safe measures, I also added the svc instruction to the end, to set the break point there. **r2 should remain 11** because we did not perform any arithmetic on it**. r3 should have stored the value of 14** because it is the resultant of 7*2 (r3*r4). **r4 should be 2** because we also didn't perform any arithmetic on it.

The next few steps are essentially the same commands as before--assembling the code, linking it make the executable, and then debugging it using GDB. However, I forgot to mention that used -g because apparently it makes debugging more consistent.

as -o -g arithmetic1.o arithmetic1.s (create an object file called "arithmetic1.o from arithmetic1.s")
ld -o arithmetic1 arithmetic1.o (create an executable called "arithmetic1" from "arithmetic1.o")
gdb arithmetic1 (opens up GNU Debugger so we can see what happens to the registers during execution)
(gdb) list 1,17 (shows us the first 17 lines of my code)
(gdb) b 17 (adds a break point to the last line)
(gdb) run (runs the executable up until the break point)
(gdb) info registers (shows the current values of the registers at the break point)



The resulting screenshot indicates that the implementation followed through successfully. The values of r1 thru r4 are as expected, which is a good sign that the instructions worked.

Working with the Raspberry PI has been quite a fun task, but there is still so much more to learn. It's a shame that our group only has one so this little computer will be separating from me for the next assignment if things go differently. But this experience has definitely given me a deeper love for Computer Science.

- Marjon Santiago