DEVELOPING SOFT AND PARALLEL PROGRAMMING SKILLS USING PROJECT BASED LEARNING

CSC-3210 SPRING 2019 (M/W-CLASS)

INSTRUCTOR: DR. AWAD MUSSA

DATE: 03/08/2019

SUBMITTED BY **TEAM HACKS**

DAVID CARR
JEFFERY KONG
MOHAMED ALI
SABINA MAHARJAN
VINCENT HU

PLANNING AND SCHEDULING:

Assignee Name	Email	Task	Duration (hours)	Dependency	Due date
David Carr	Dcarr15@student.gsu.edu	Task 1	1hr		3/2/19
Jeffery Kong	Jkong8@student.gsu.edu	Programmer	3h	Must have PI	3/4/19
Mohamed Ali	Mali41@student.gsu.edu	Group coordinator	2h	Read and answer questions	3/3/19
Sabina Maharjan	Smaharjan1@student.gsu.edu	Report	2h	Slack, GitHub	3/3/19
Vincent Hu	Vhu2@student.gsu.edu	Programmer	3h	Must have PI	3/4/19
Everyone		Record Video			

PARALLEL PROGRAMMING SKILLS:

Part A: Foundation

- Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words

- A task is a set of instructions that are executed by the processor and parallel program consists of many tasks.
- Pipelining is allowing multiple instructions to be processed at the same time.
- Shared Memory is a form of memory where all processors have direct access to a shared physical memory and parallel tasks can directly access that same logical memory.
- Communications is a form of data exchange between parallel tasks.
- Synchronization is a point where a task may not proceed further until another task reaches that same point.

- Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them

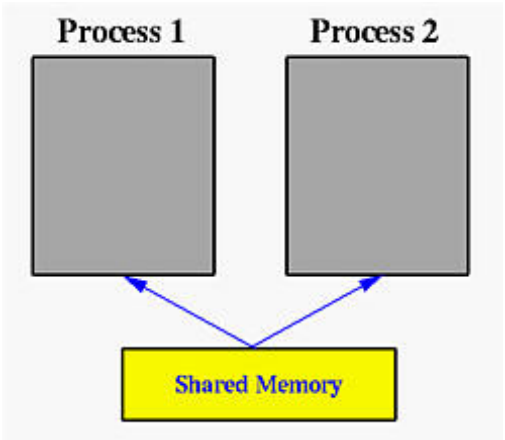
- Single Instruction, Single Data (SISD): executes exactly one instruction stream at a time

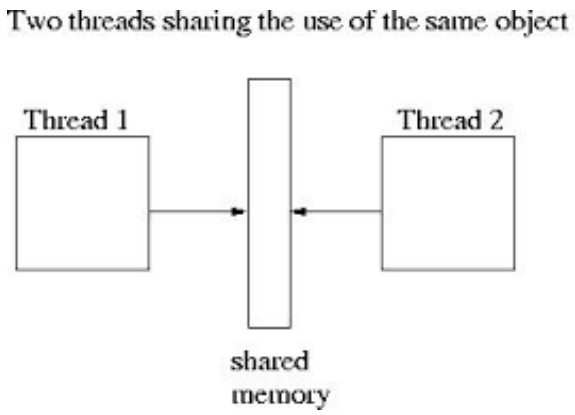
- Single Instruction, Multiple Data (SIMD): is an instruction set architecture (ISA) that have a single CU and more than one PU, and it executes a single instruction stream over PU handled through the CU.
- Multiple Instruction, Single Data (MISD): is an ISA for parallel computing where multiple functional units execute different operations on the same data set.
- Multiple Instruction, Multiple Data (MIMD): is an ISA for parallel computing that is ideal for computers with multiprocessors and each processor may be executing a different instruction stream.

- What are the Parallel Programming Models?
- Shared Memory, Threads, Distributed Memory, Data Parallel, Hybrid, Single Program Multiple Data (SPMD), Multiple Program Multiple Data (MPMD)

- List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?
- General Characteristics: multiple processors that can run separately but share the same memory and all the processors have access to the memory as global address space.
- Uniform Memory Access (UMA): identical processors that have equal access times to memory and upon update on location in shared memory, all other processors know about the update.
- Non-Uniform Memory Access (NUMA): SMPs are linked, and SMP can directly access the memory of another SMP.
- Uniform Memory Access uses OpenMP because OpenMP targets shared-memory architectures.

- Compare Shared Memory Model with Threads Model? (in your own words and show pictures)





- Shared Memory is the simplest parallel programming model, and tasks share a common address space, without threads Model it's hard to manage data locality.

- What is Parallel Programming? (in your own words)
- Parallel Programming is using multiple computing resources to solve problem concurrently.

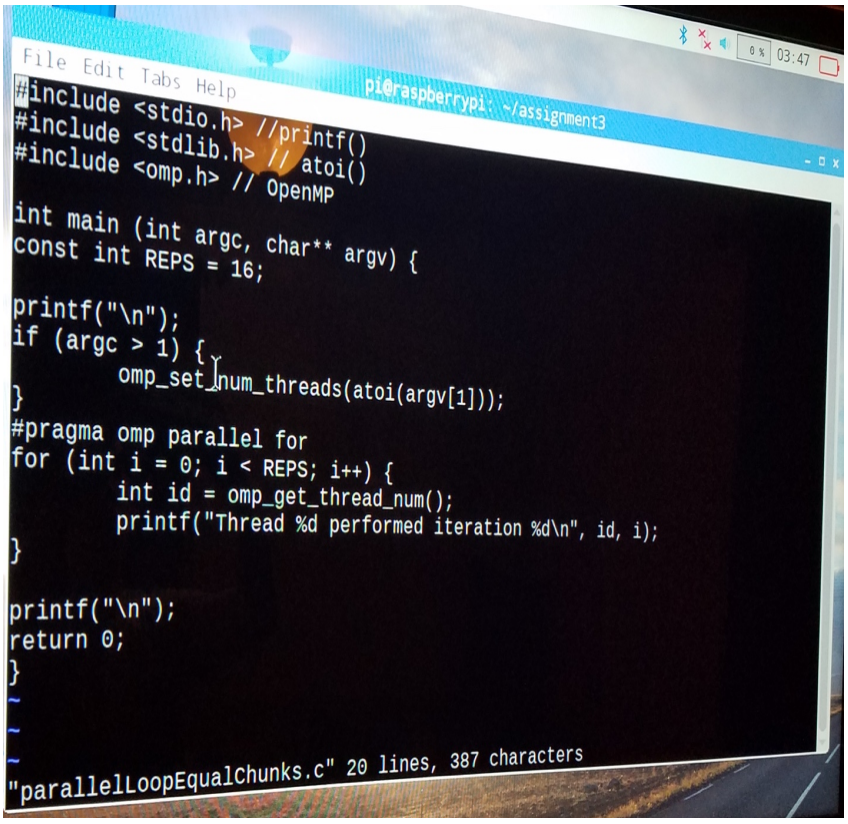
- What is system on chip (SoC)? Does Raspberry PI use system on SoC?
- SoC integrates all the components into a single chip.
- Yes, Raspberry PI uses system on SoC.

- Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components
- System on a chip consumes less power and costs less. Also, System on a chip is small and lightweight compared to having separate CPU and RAM.

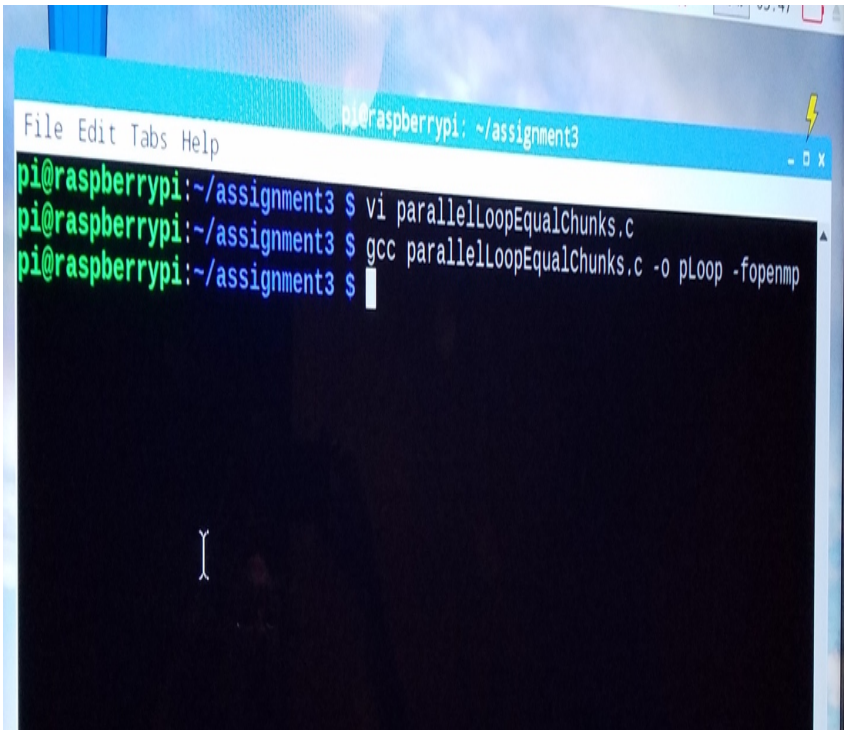
Part B: Parallel Programming Basics

After reading through the Parallel Programming Task assignment, we started to copy the example code onto our Raspberry Pi.

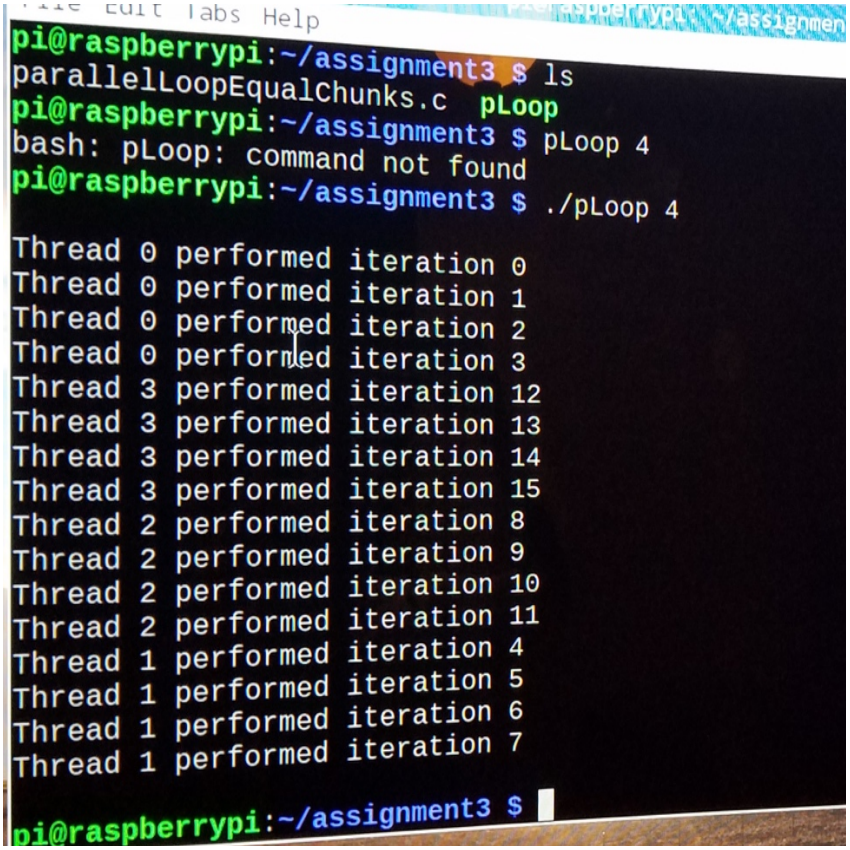
Our understanding of the code is that the line: *#pragma omp parallel for*, is different from the previous project's code: *#pragma omp parallel* because it specifically tells the processor to run the for loop in parallel, not just all of the code in the block, so the for loop is split up among the 4 processors of the Raspberry Pi.



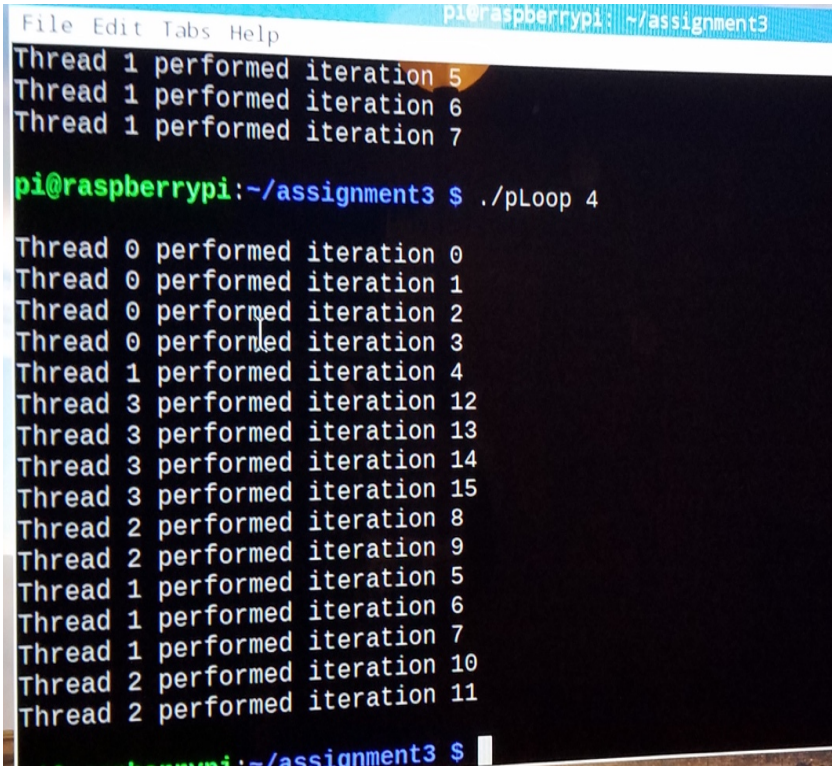
After copying the example code, we had to compile and link it using the gcc command. This command took the source code: parallelLoopsEqualChunks.c and created an executable called pLoop. In order to run the program, We had to call ./pLoop with the optional argument of 4.



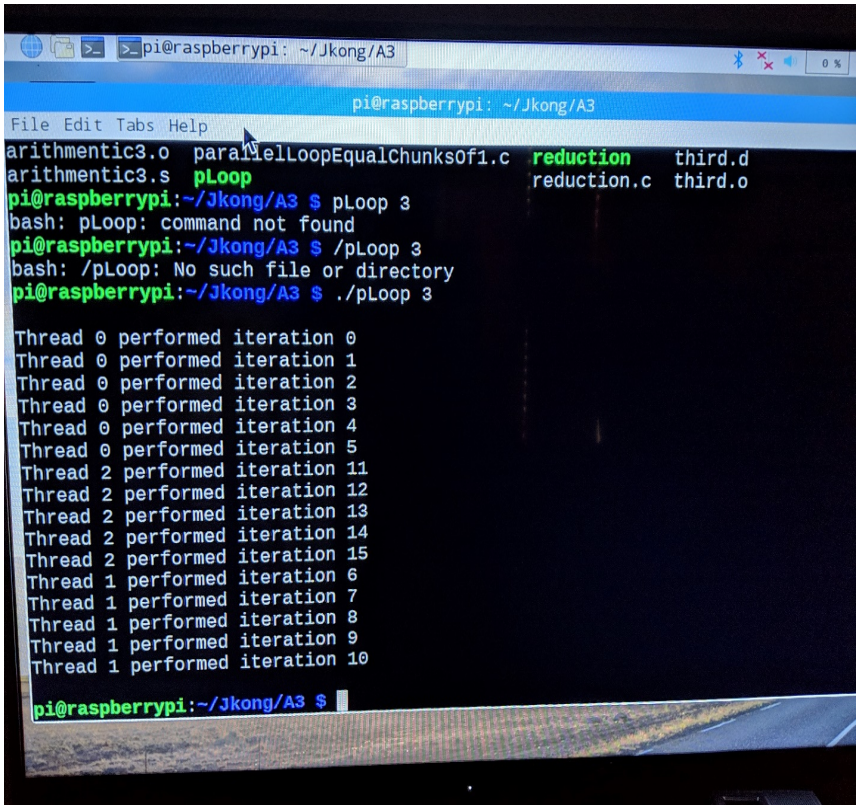
At first, we tried to run the command using pLoop 4, but because the ./ path is not in our shell's current $PATH variable, the command was not found. We could have added the ./ path to the $PATH variable but it was easier to simple add the directory to the command name. The output split the 16 loops among the 4 processors and ran them in parallel. So, each thread performed 4 iterations.



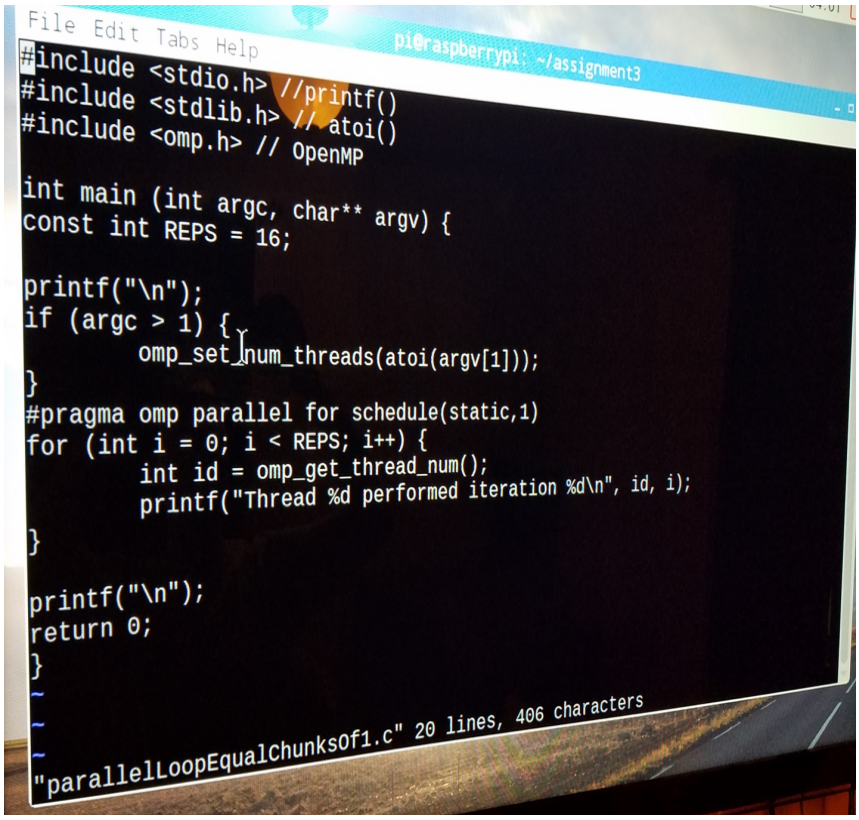
To double check, command prompt was checked again for the output. It is worth noting that the order that the iterations completed was random.



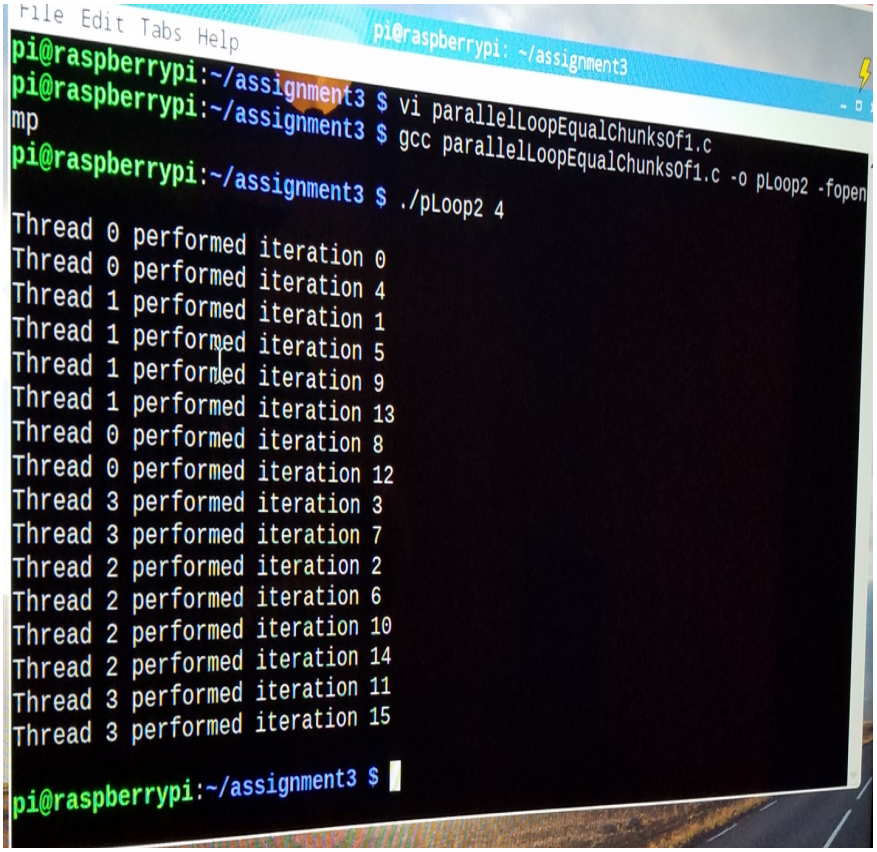
When the program is run with a number that is not evenly divisible by the number of loops, the processor tries to spread the tasks out as best it can. It will divide up the loops by the number of threads and the remainder is added to thread 0.



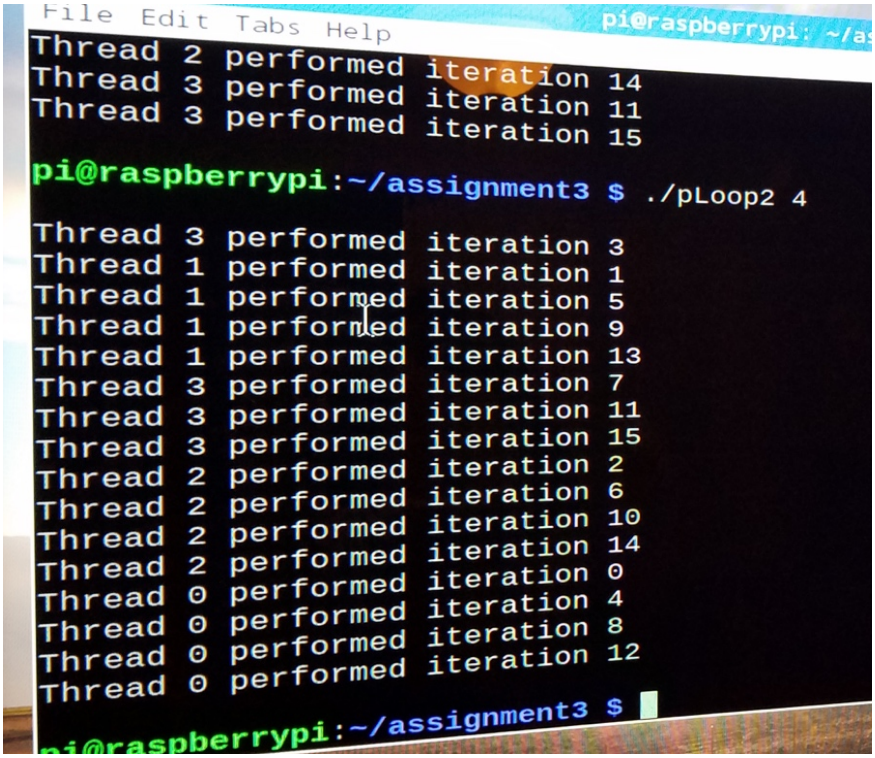
Following the example of parallelism is assigning the threads as each completes its task, so the assignment is done statically. Each thread still has equal amounts of work, just not consecutive iterations.



After copying the code, we compiled it and ran it, using the same techniques as an example 1 above. The output was relatively similar between the two examples, even though the second example used a different method of assigning threads.

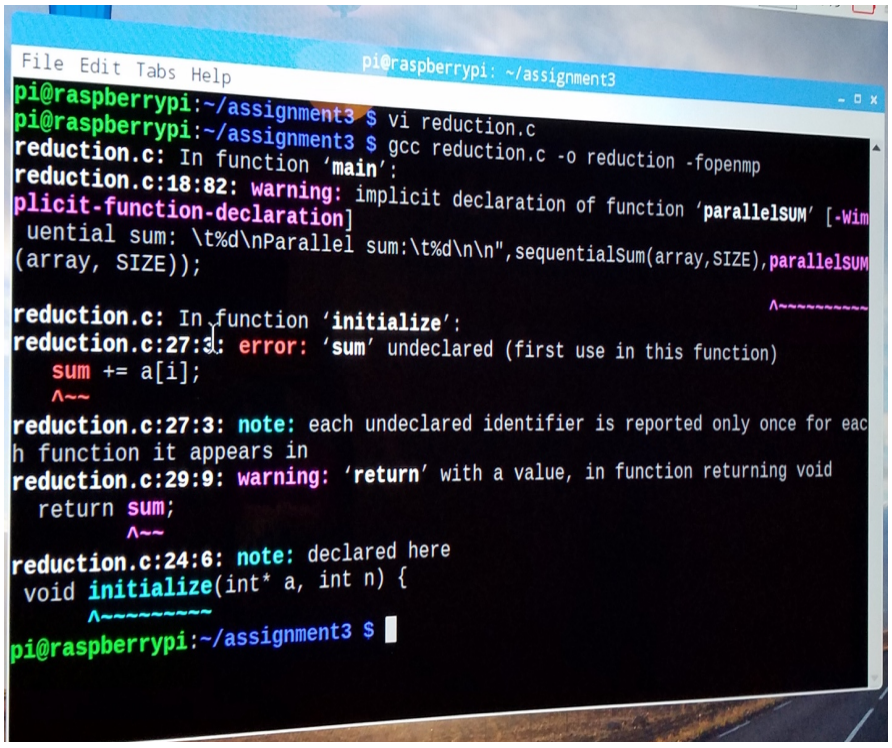


We ran the code again, just to ensure that the code was functioning as expected.

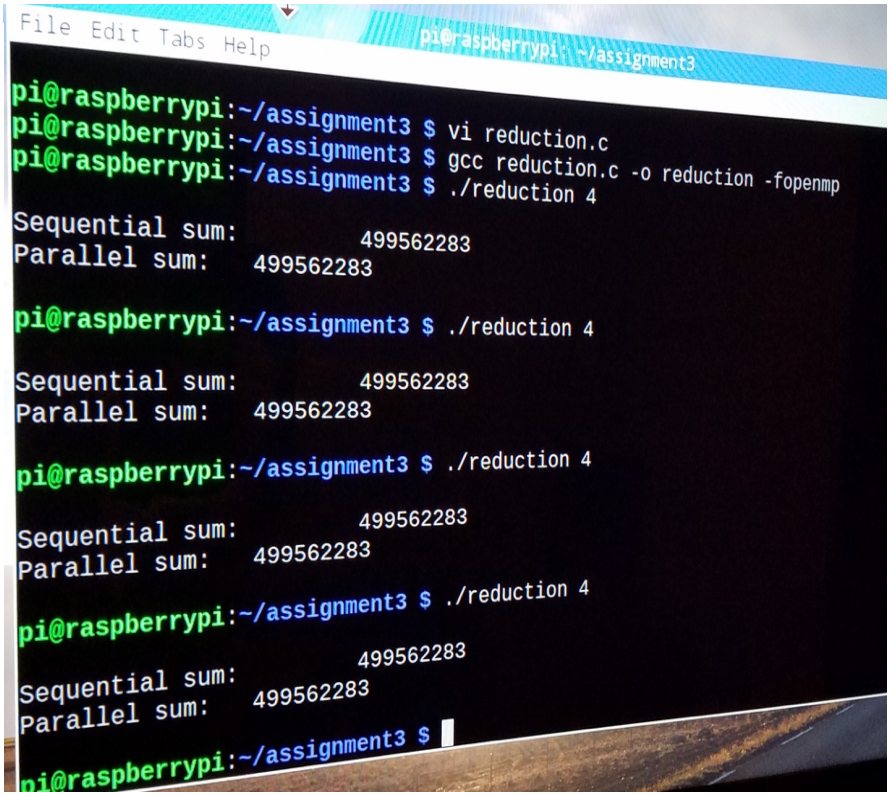


The code in example 4.1 Loops with Dependencies was long and difficult to copy, so we had a few errors for the first time we tried to compile the code.

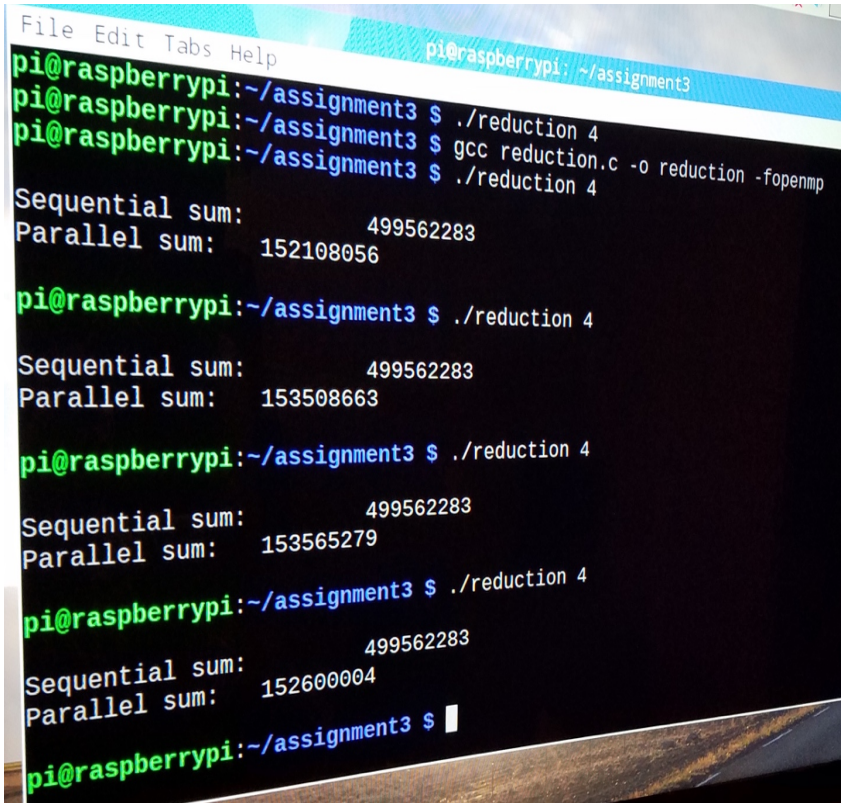
After going back and fixing the errors, we were able to get the code to compile correctly.



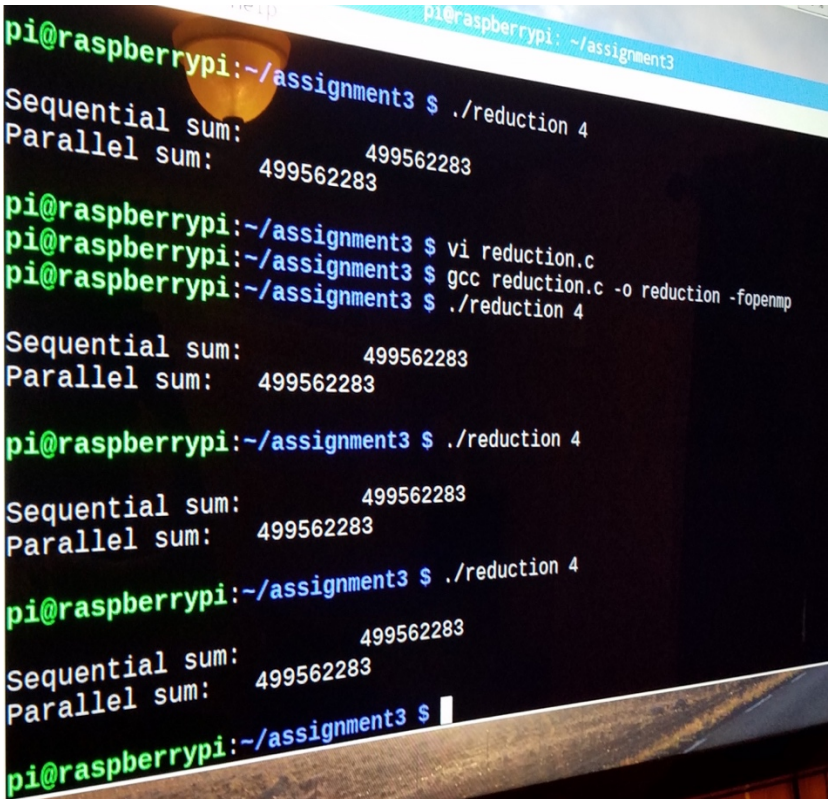
In this first example, the *// #pragma omp parallel for // reduction (+:sum)* code is commented out, so the code is not actually running in parallel. When it is not running in parallel, the sums match.



In this second example, the *#pragma omp parallel for // reduction(+:sum)* code is partially commented. The for loop is still being run in parallel, but the reduction for the sum function is not being called. When a loop with dependencies is being run in parallel but the dependency is not address, the results for the function can be unexpected or incorrect. Notice that the Sequential sum and parallel sum do not match.



In this final example, the code is running in parallel with reduction on the sum function. With the reduction function forcing the threads to run each sum independently and then summing up the final result at the end.

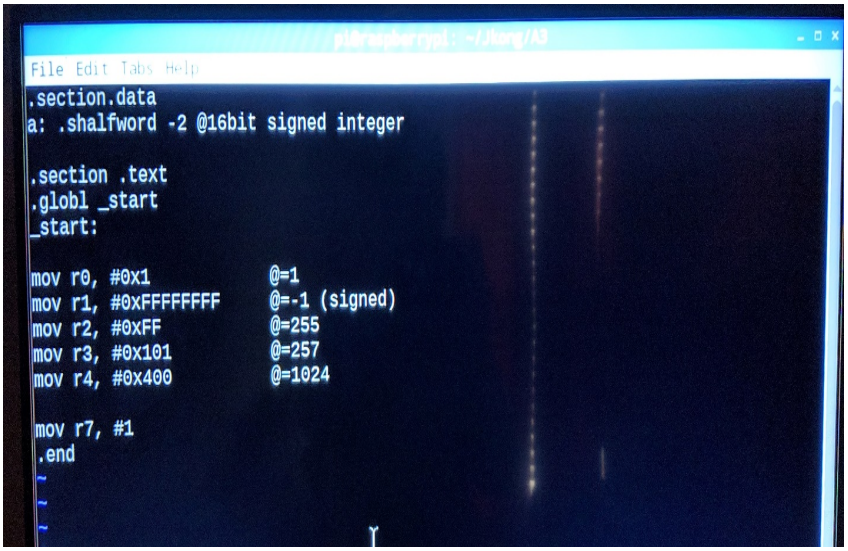


With this example, the dependencies could be split up between the threads and totaled at the end of each thread. there are probably some situations where the dependency is not so easily split and recombined, but we will possibly learn about these problems in a future project.

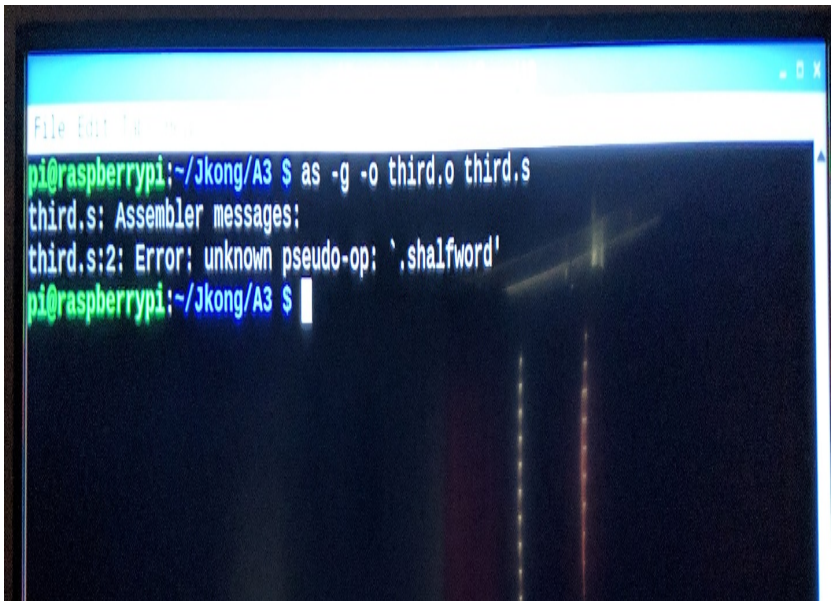
ARM Assembly Programming:

Part 1: Third Program

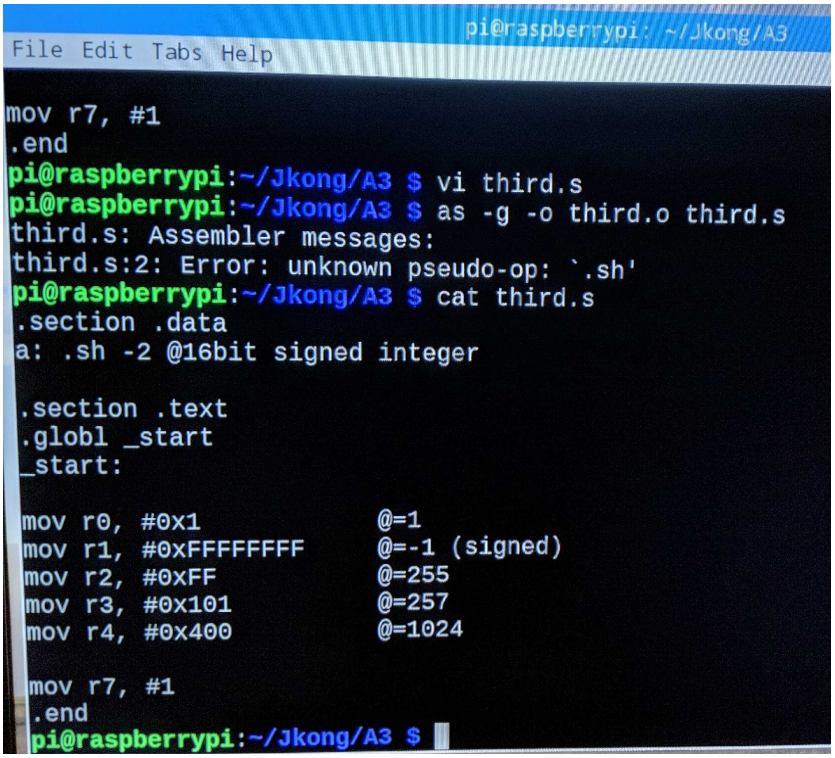
Here is how the specified "third.s" file was created using the vi editor.



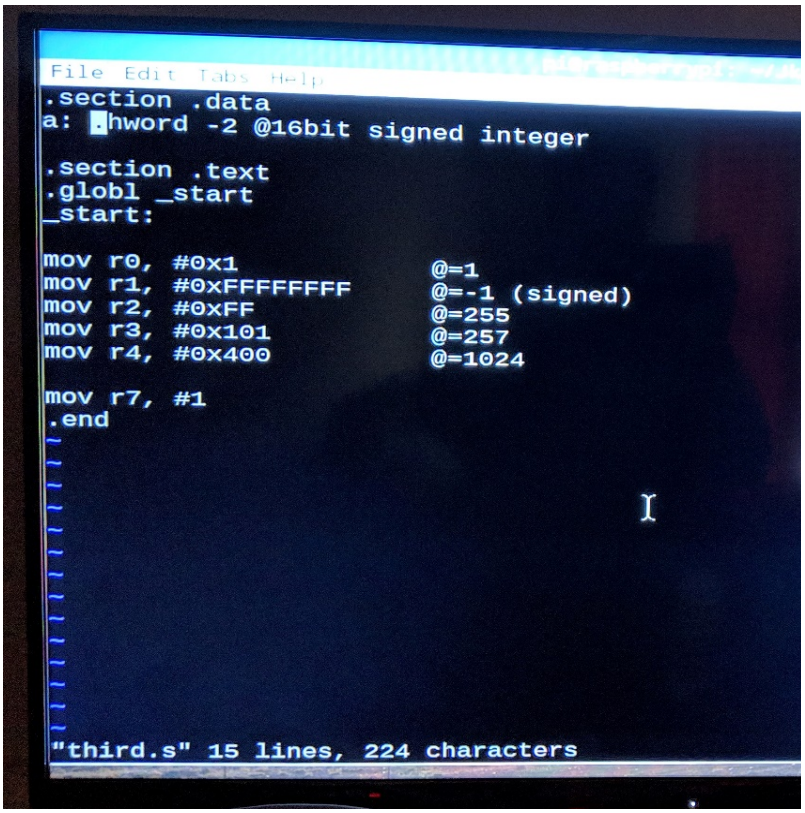
After creating the file and assembling it, we ran into an error which states: “unknown pseudo-op: ‘.shalfword’. We then went back into vi to attempt to fix it.



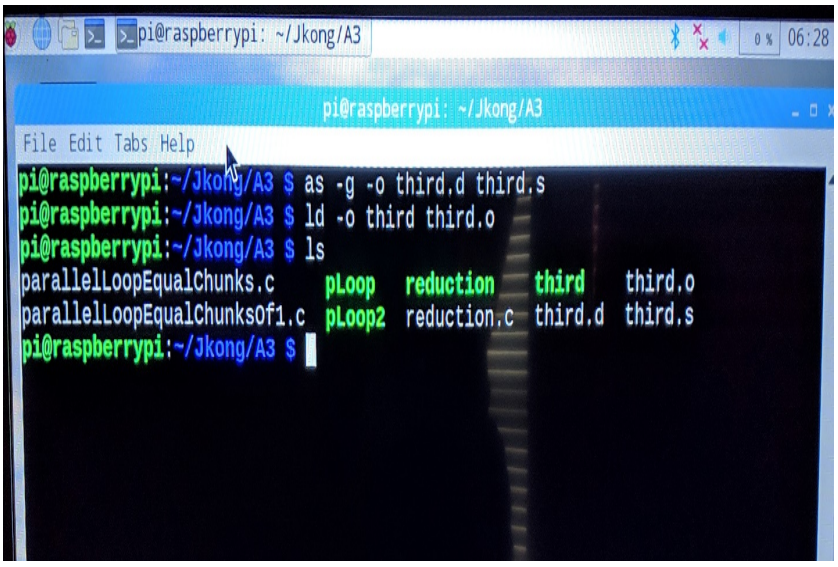
We tried to make variable “a” into a 16 bit signed integer was to change the .shalfword to .sh or to .shword but none of those work. Here is the output of showing that the assembler did not recognize .sh for a signed 16 bit integer.



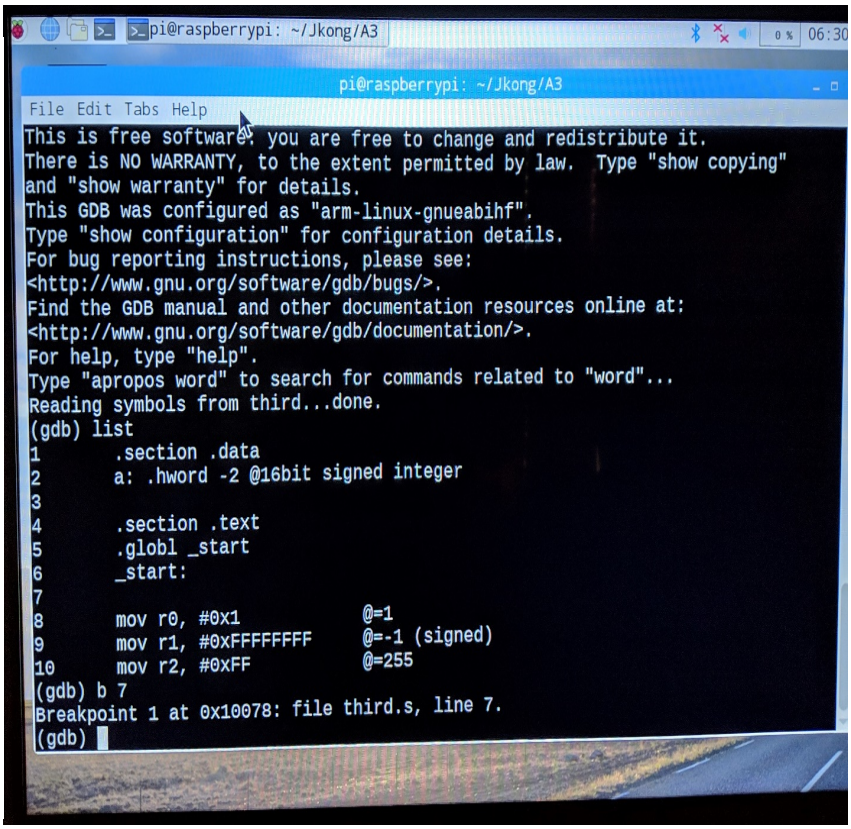
After looking through online resources to solve this error, we came to the conclusion that to declare a variable as 16-bit, we have to use ".hword" instead of ".shalfword". When we load the variable into a register, we can then use "ldrsh" to load the variable as signed.



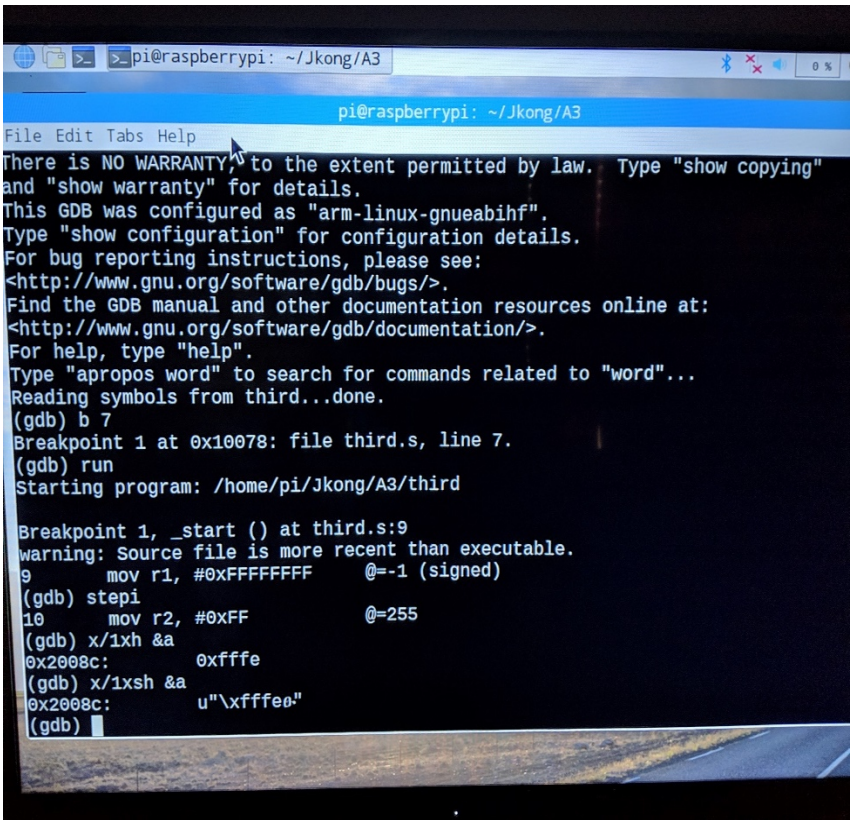
After resolving the error, we were able to assemble the objective file and link it for the executable. We also used the "ls" command to make sure it was there. The "third.d" was a type for when we tried to assemble the file the first time.



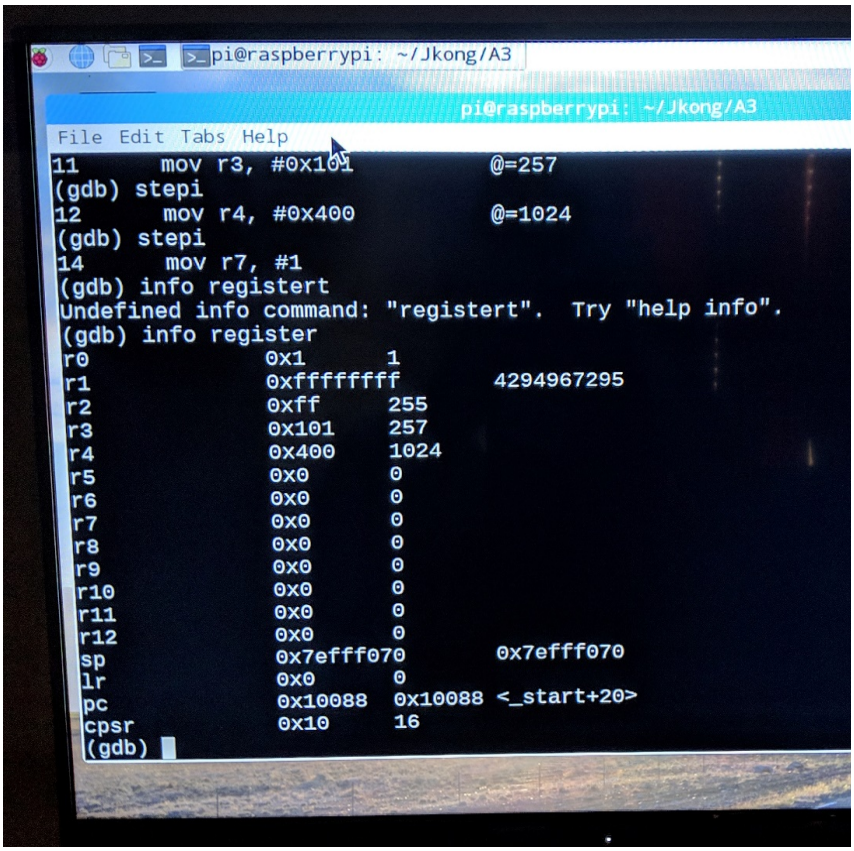
Here is us executing “third” in GDB and setting a break point at line 7.



After setting the breakpoint, we ran the program and used the “x” command to show what the value of the variable “a” is by using “&a” in the command line. When using “x/1xh”, the gdb outputted 0xfffe, which is the 2’s complement representation of -2. However, when using “x/1xsh”, the gdb outputted the same thing along with some other symbols.

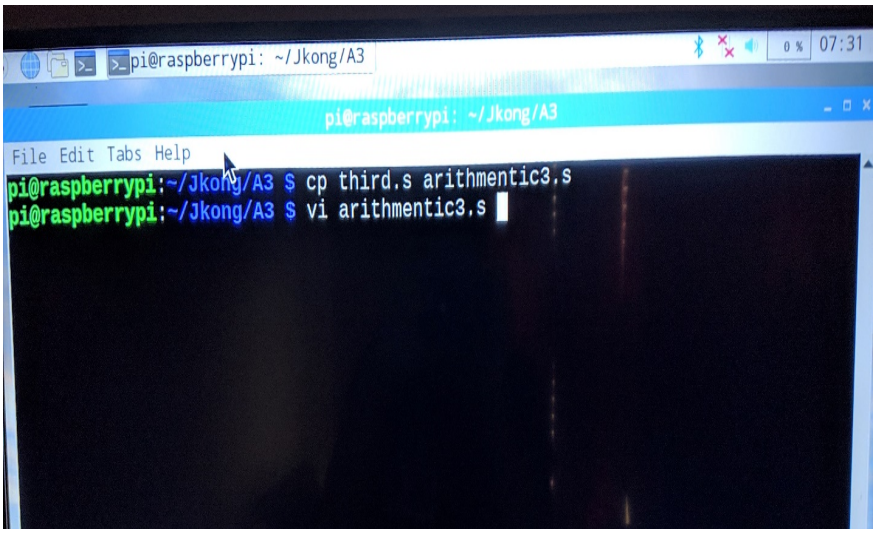


Finally, we used the “stepi” command to run through the program and viewed the registers to make sure everything was moved correctly. For register “R1”, the value is 0xffffffff which makes sense because that’s the 2’s complement representation of -1.

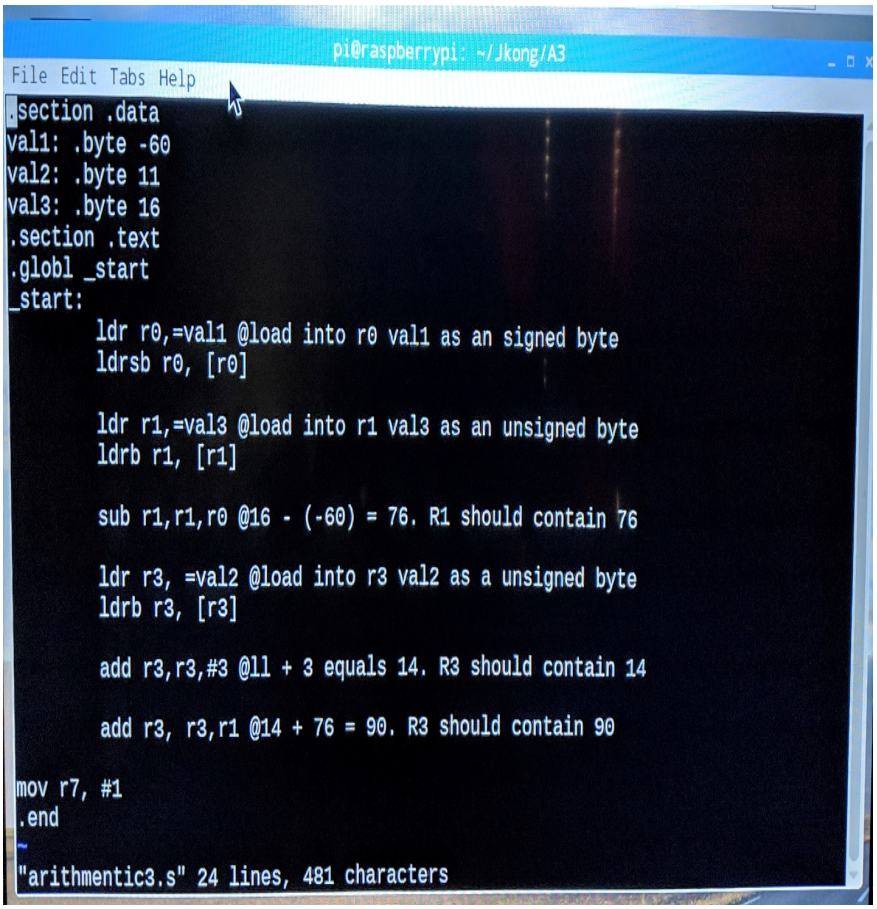


Part 2: arithmetic.3

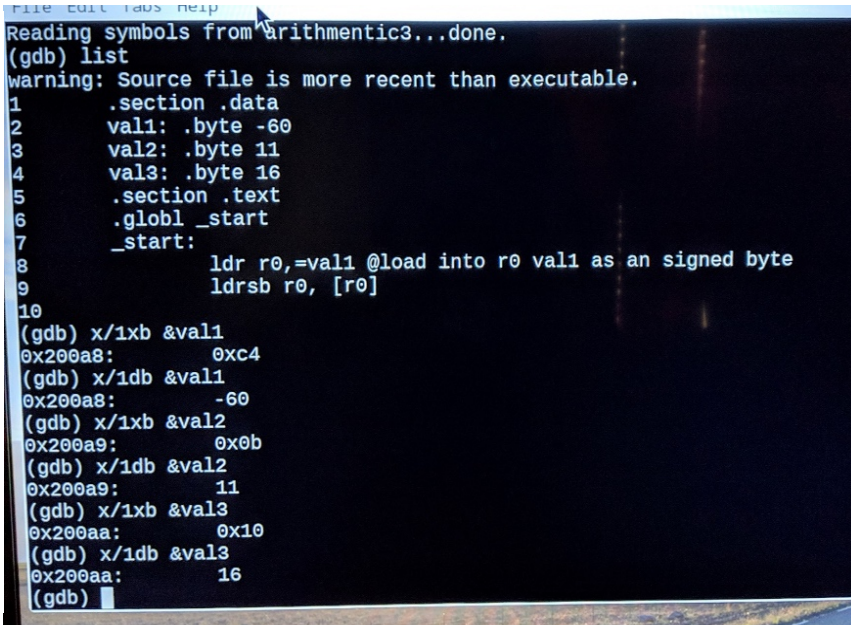
To save us the trouble of writing some of the assembly code again, we copied “third.s” from the previous part and used it as a template for writing “arithmetic3.s”. We realized there’s a typo in the picture. Unfortunately, that is the name of my program now.



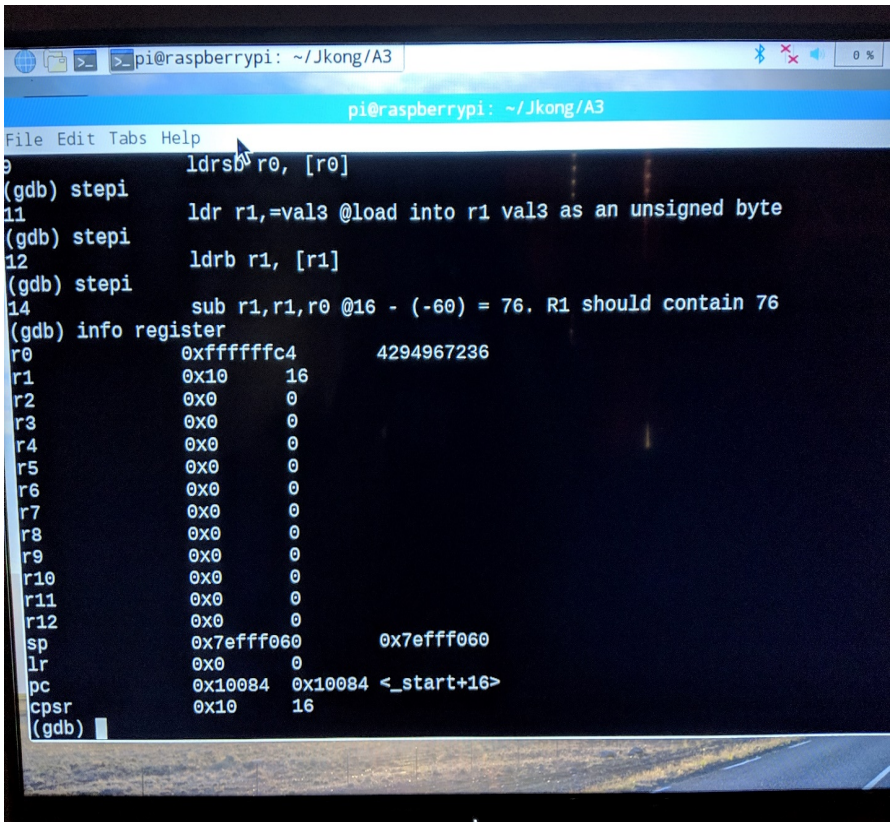
Here is the finalized version of my "arithmetic3.s" file. We had trouble trying to load the variables into the registers simply because we did not remember, so we looked back at the arithmetic file from assignment A2. We declared all three variables as a byte and assigned them their variables. Qw solved this expression from right to left. First, we assigned R0 val1 and loaded it as a signed byte. Then, we loaded R1 with val3 as an unsigned byte. We subtracted R1 from R0 and put the result in R1. We then loaded R3 with val2 as an unsigned byte and added 3 to it. All that's left is to add R3 and R1 and the final answer is in R3.



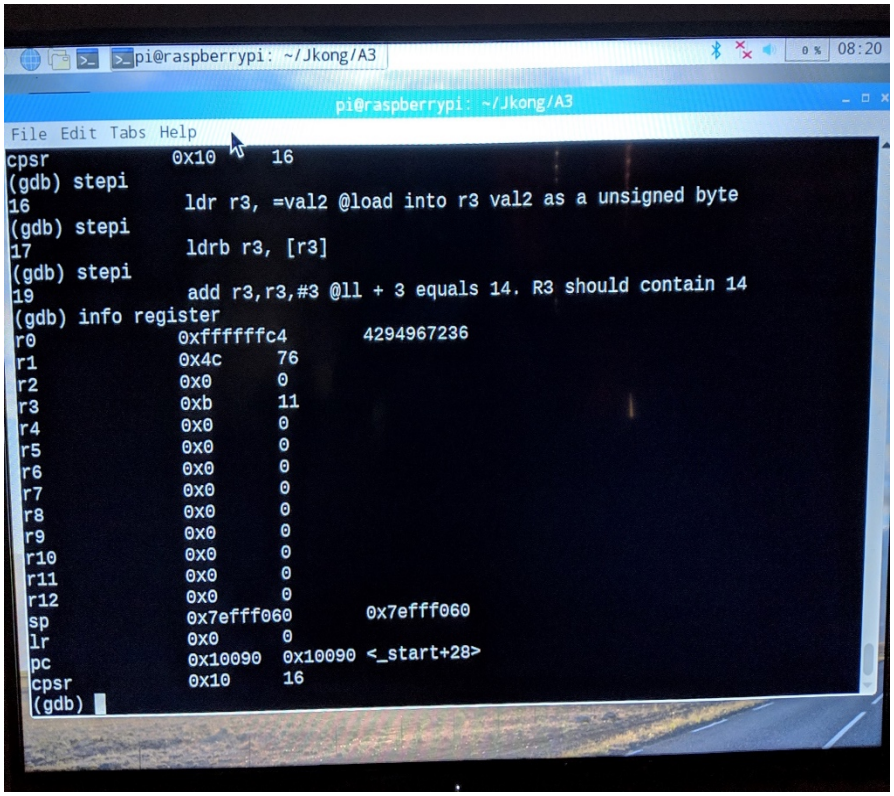
I've assembled and linked the file and opened it in gdb. Upon entering gdb we used "list" to show the variables. Using the "x" command, we have shown the hexadecimal contents for each variable. We've also shown it as a decimal to fully confirm it because we did not want to do hexadecimal to decimal calculations.



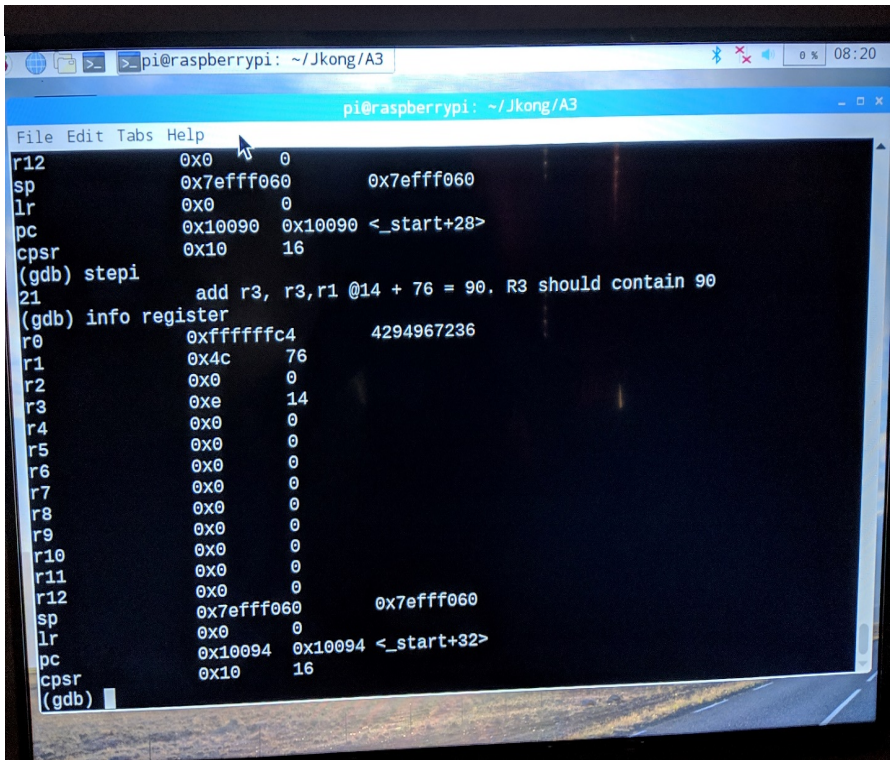
We set a breakpoint before running the program and used "info register" to show the contents of the register as the program progresses. Here, it can be seen that var1 and var3 being loaded into R0 and R1 respectively.



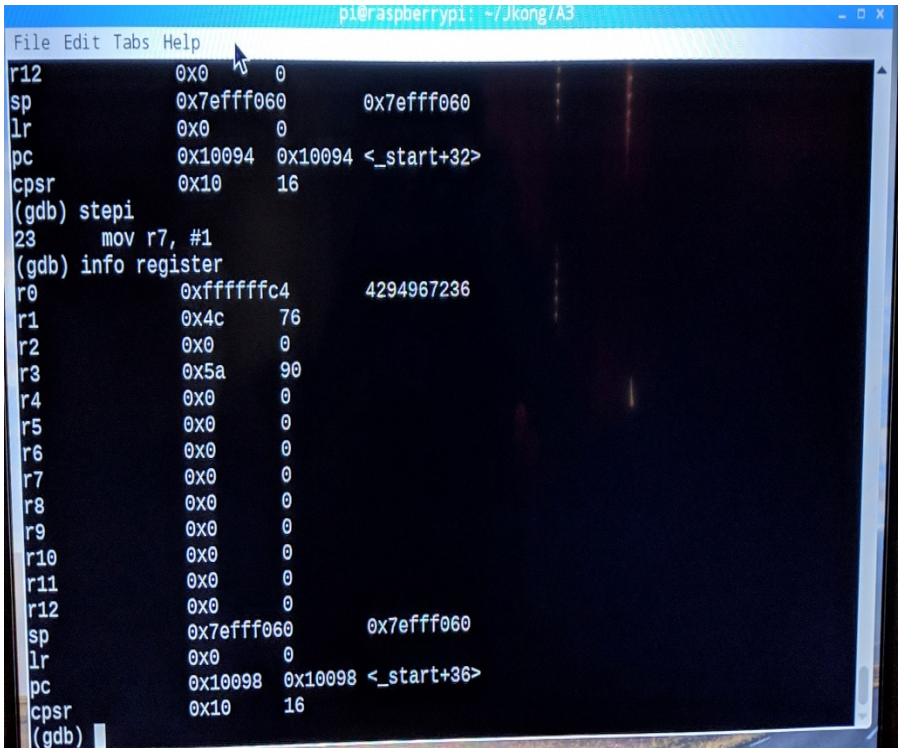
Now, it can be seen that the results of R0 being subtracted from R1. The result is 76. Also, var2 has successfully been loaded into register R3.



This is just R3 + 3. The result is 14.



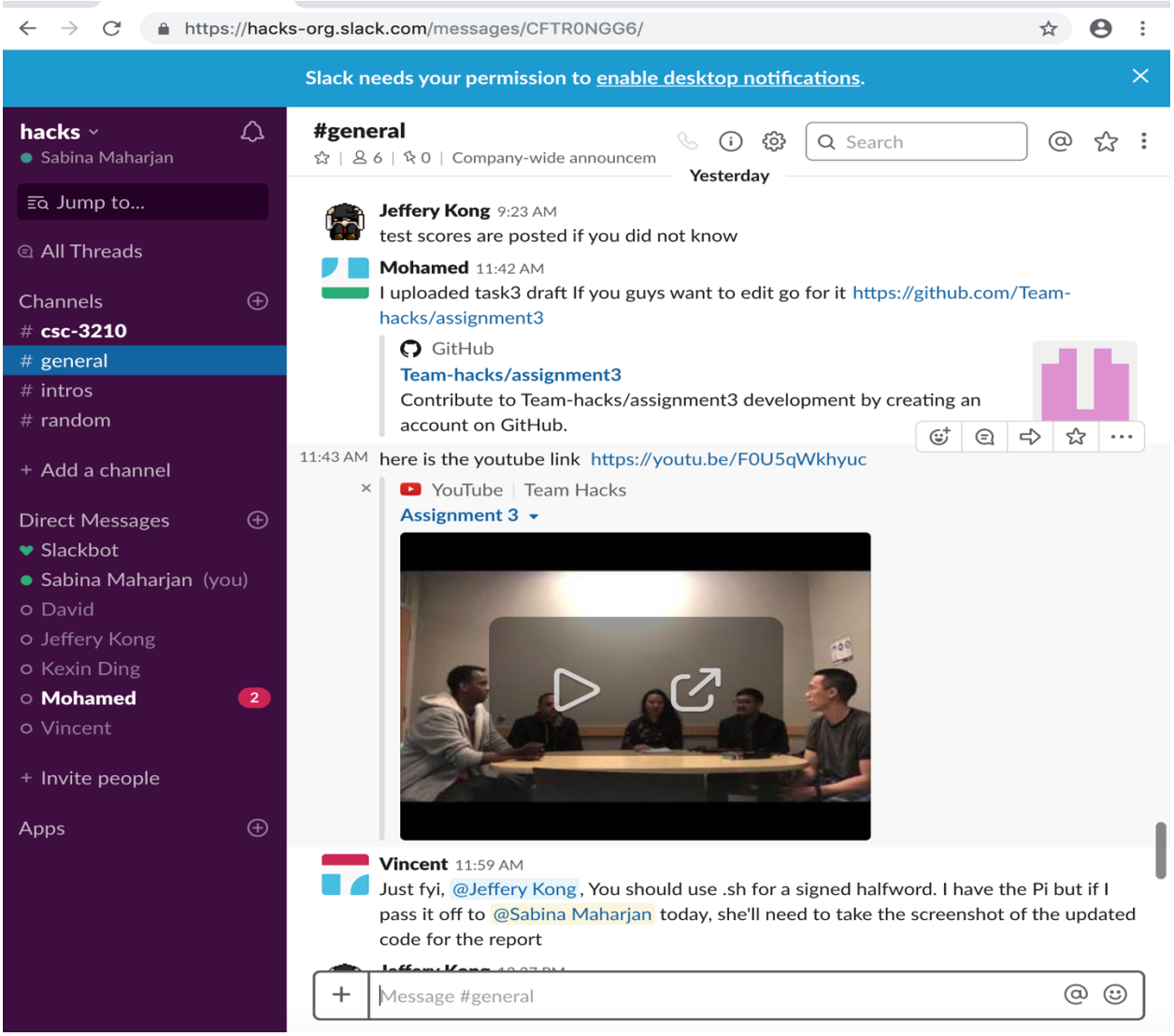
Finally, just add R1 to R3 and the final result is 90. It can be seen that the CPSR register below with contents 0x10 which is 16 in decimal. This means that the contents of CPSR is as follows:
0000.0000.0001.0000



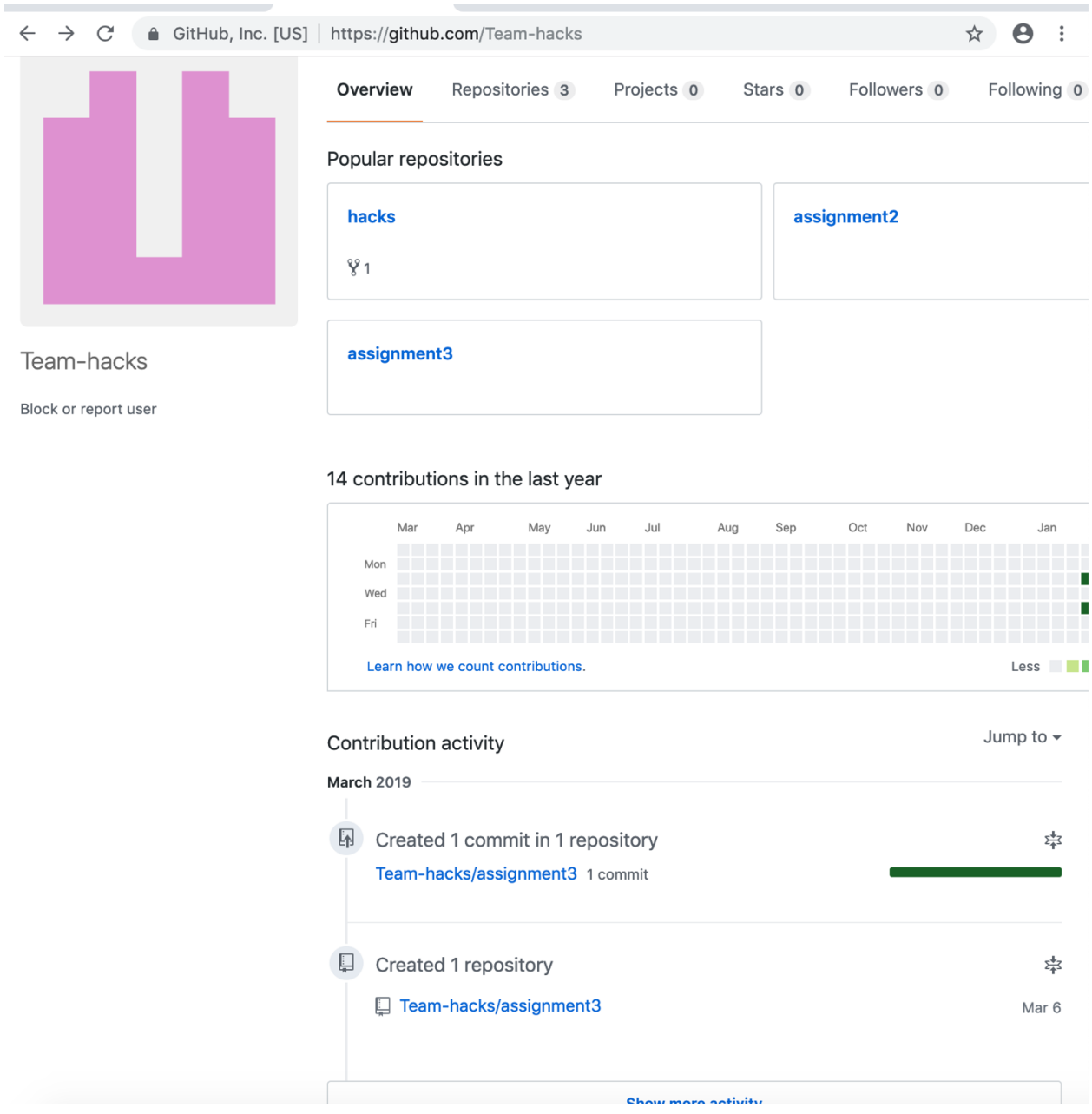
In the CSPR register, the Sign (Negative) Flag is the 32nd bit and it is not set at the end of the program because the answer is positive.

APPENDIX:

Slack
https://hacks-org.slack.com/messages/CFTR0NGG6/team/UFSJLJGSH/



GitHub
https://github.com/Team-hacks/assignment3/blob/master/assignment3.docx



You-Tube:
https://www.youtube.com/watch?v=F0U5qWkhyuc&feature=youtu.be