Developing Soft and Parallel Programming Skills Using Project Based Learning

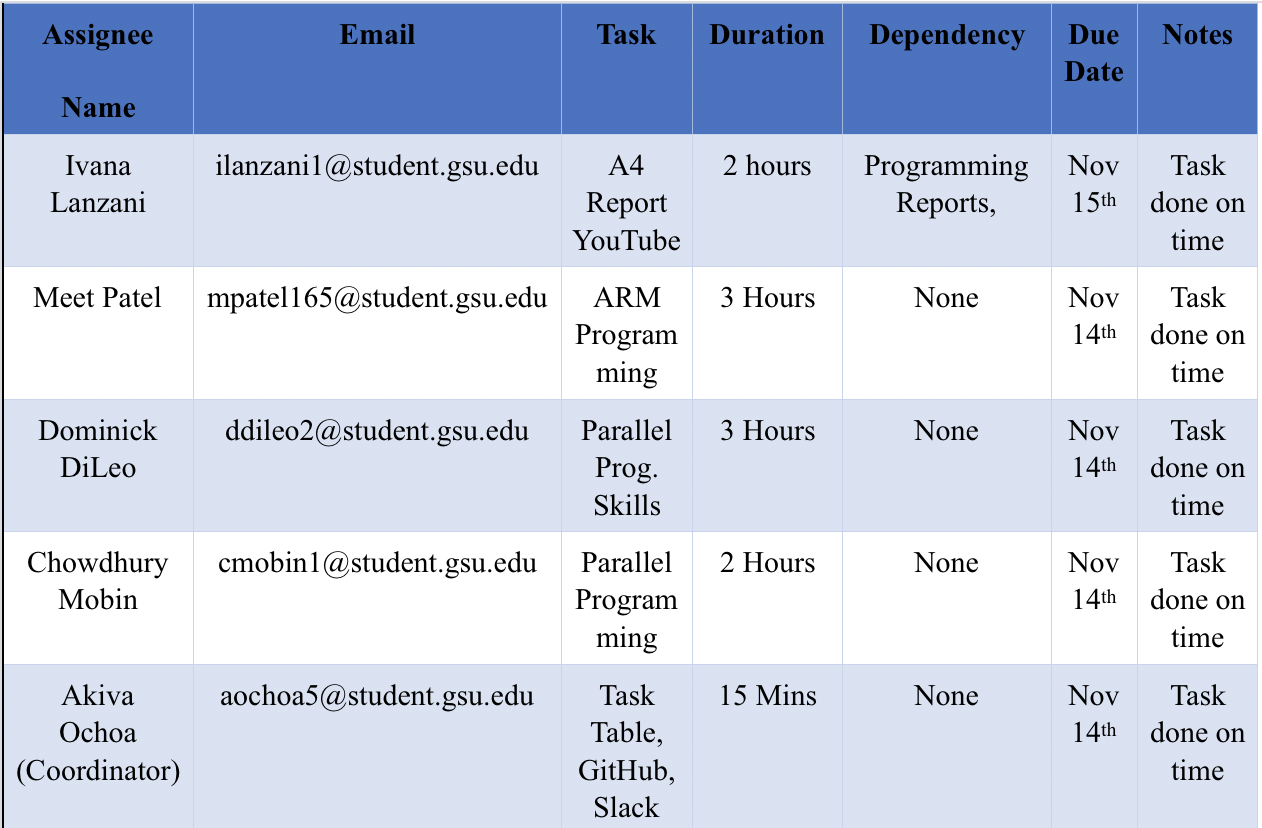
Fall 2019

Quinary

Chowdhury Mobin, Dominick DiLeo, Akiva Ochoa,

Meet Patel, Ivana Lanzani

Planning and Scheduling:



Parallel Programming Skills (a)

**Race condition:**

**What is race condition?**

The behavior of your program when the output is dependent on the sequence or timing of uncontrollable events.

**Why race condition is difficult to reproduce and debug?**

The end result is not able to be determined and depends on relative timing between the interfering threads, so it can change every time you run it.

**How can it be fixed? Provide an example from your Project\_A3 (see spmd2.c)**

It can be fixed by careful software design. Here is where an example occurs in Project A3

int parallelSum(int\* a, int n) {

int sum = 0;

int i;

#pragma omp parallel for reduction(+:sum)

for (i = 0; i < n; i++) {

sum += a[i];

}

return sum;

}

Without the reduction statement, the threads might try to write to sum at the same time, causing a data race or race condition. The solution is “careful software design”, that is the reduction statement. This statement gives each thread its own variable copy so they do not interfere with each other, then all get combined back into the original one after they are done.

**Summaries the Parallel Programming Patterns section in the**

**“Introduction\_to\_Parallel\_Computing\_4.pdf” (two pages) in your own words (one paragraph, no more than 150 words).**

There are two main categories developers use for writing parallel programs: Strategies and concurrent execution mechanisms. You should consider an algorithmic strategy that is concerned with making choices about what tasks can be done concurrently by multiple units. Based on your algorithmic strategy consider what implementation strategy to use; these are about the overall structure of the program and how the data being computed is structured. There are two main categories of concurrent execution mechanisms: Process/thread control patterns(how processing units are controlled at runtime), and coordination patterns(how multiple running tasks coordinate)

**In the section “Categorizing Patterns” in the**

**“Introduction\_to\_Parallel\_Computing\_4.” compare the following:**

**Collective synchronization (barrier) with Collective communication (reduction):**

Collective synchronization is when the threads must all be completed before any can move on to the next step. There is a barrier preventing them from continuing in this case. Collective communication is when all threads communicate in some way to come to a result. Reduction makes individual copies of the variable in the reduction clause and gives one to each thread. This process implies it makes a barrier as well. Before the variable in the reduction clause can be used again, the threads must all finish their individual actions on their private variable and be joined together again.

**Master-worker with fork join:**

Pragma forks threads and joins them back together. Master-worker is when during the forking, you make statements assigning a particular thread to do a task(the master), and the other threads do other tasks(The workers)

**Dependency: Using your own words and explanation, answer the following:**

**Where can we find parallelism in programming?**

We can find parallelism in between program statements and when statements can be done at the same time. Within loops or other program statements, there can be parallelism for the statements in them.

**What is dependency and what are its types (provide one example for each)?**

Dependency is where operations cannot execute without the execution of another operation. The types of dependencies are control, data, and system.

Control: An if loop like if(i < 0) {i++}. The execution of the statement i++ is control dependent on the condition that i is less than 0.

Data: An example would be the following two statements:

A = 1;

B = A;

Here B is dependent on the data of A

System: An example would be when there are multiple computers in a system at a company. There could be a task that requires the use of multiple computers, and thus there is system dependency between them.

**When a statement is dependent and when it is independent (Provide two examples)?**

Statements are independent when the order of the statements does not matter and dependent when the order does matter.

Dependent example:

DATE\_BORN = 1997

AGE = CURRENT\_DATE – DATE\_BORN

The AGE variable is dependent on DATE\_BORN being defined first. Without this statement, AGE could not be calculated.

Independent example:

AGE = 22

NAME = “NICK”

Here the AGE and NAME variables are completely unrelated. No matter the order the result will be the same.

**When can two statements be executed in parallel?**

Two statements can be executed in parallel if there are no dependencies between them.

**How can dependency be removed?**

You can remove dependency by modifying the program in ways like rearranging statements and eliminating statements.

**How do we compute dependency for the following two loops and what type/s of dependency?**



**a) b)**

You unroll the loop into separate iterations and show the dependencies between the iterations.

1. Each iteration is only affecting a unique index of the array a, and thus they have no dependency between each other. The only dependency here is that S1 will affect a different index based on what i is.
2. The two statements here are independent of each other, because no calculation in either are used in the other one. They could be reversed and the same result would occur. In addition to this, the statements are independent of the statements in the next iteration for the same reason as the explanation for a). Like with the previous loop, the only dependency is the statements on what i is each iteration.

Parallel Programming Skills (b)

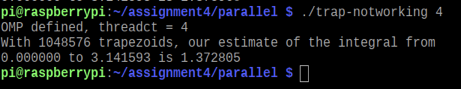
In the first part of the programming assignment, we are attempting to compute a Calculus value. Specifically, the “trapezoidal approximation of , using equal subdivisions.

First, I created a C file, named trap-notworking by typing the command “nano trap-notworking.c”. Then, I copy-pasted the code which was provided. After exiting from the nano editor and typing the command “gcc trap-notworking.c -o trap-notworking -fopenmp -lm”, it created an executable program “trap-notworking”.

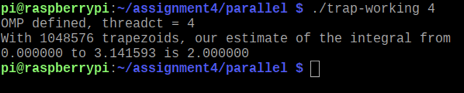
After this, I followed the same procedure for the second code, which was provided also. This time, I changed the name of the file to “trap-working.c” as it was instructed. Then copy-pasted the second code and created an executable program named “trap-working” for the C file by typing “gcc trap-working.c -o trap-working -fopenmp -lm”.

Finally, to run and compare the two programs, I typed “./trap-notworking 4” and “./trap-working 4”, one after another. As the Raspberry Pi already has a four-core processor, it is natural to try four threads.

Picture #1: trap-notworking



Picture #2: trap-working



Both gave us different results after execution. The difference between the two programs was on line 37 and line 38. The program “trap-notworking”.

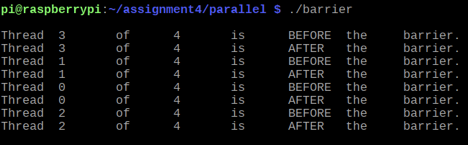
Both programs were created to find the definite integral of sin(x) from 0 to π. It is also stated that the answer should be 2.00. This is correct, but if only one thread is used to calculate the result, it gave the correct output while we edited out code again to run it on multiple threads; it gave us the wrong output. As we can see in picture #1, where “trap-notworking” program gave generated an output of 1.37, which is not the same if we re-run the program again. Every time we run it gives a different value but not the correct one.

On the other hand, “trap-working” gave us the correct value of 2.00. The line of code, “pragma omp parallel private (i) shared (a, n, h, integral), on program “trap-notwroking”, created a data race. Due to this, the “integral” is where the summation occurred, and it happened in the same memory space. On the second program “trap-working”, we added an extra directive, which is “reduction (+: integral)”. It avoided the data race. It gave each thread its own memory, and then it added it all up to the integral variable on the main thread. Even if we run the code with any number of threads, the output gave us 2.00 in picture #2 as a result, which was correct.

For the second part of the assignment, we used the barrier pattern. Which is used in parallel programming to ensure that before execution continues, all threads complete a parallel section of the code. When threads are generating computed data, this could be necessary.

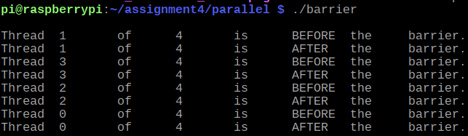
First, I copy-pasted the provided code on a new file named “barrier.c”. Then created an executable program for the file by typing the command, “gcc barrier.c -o barrier -fopenmp”. It generated an executable named “barrier”, then we run the program to see the output.

Picture #3: barrier



Now, we re-compile and re-run the code again, but this time without the commented pragma line “pragma omp barrier”.

Picture #4: barrier

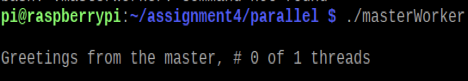


In this program, our main goal was to use the directive, “#pragma omp barrier”. When the directive was not used or commented out, the statements were printed in random order, which can be seen in picture #3. But when we used the directive or uncommented it, the statements printed were much more organized, which can be seen in picture #4. It started from thread 1, and each of the threads printed a before and after statement.

For the final part of this assignment, we will use the Master-Worker strategy. Here, one thread will be called the master, which execute one block of the code when it forks and the other threads called workers will execute different parts of the code when they fork.

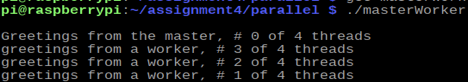
To demonstrate this method, we created a file named “masterWorker.c” and copy-pasted the code provided. Then compiled it by typing “gcc masterWorker.c -o masterWorker -fopenmp”. Which gave us an executable “masterWoeker” to run.

Picture #5: masterWorker



Now, we re-compile and re-run the “masterWorker.c” file, but this time we uncomment the pragma directive “#pragma omp parallel”.

Picture #6: masterWorker

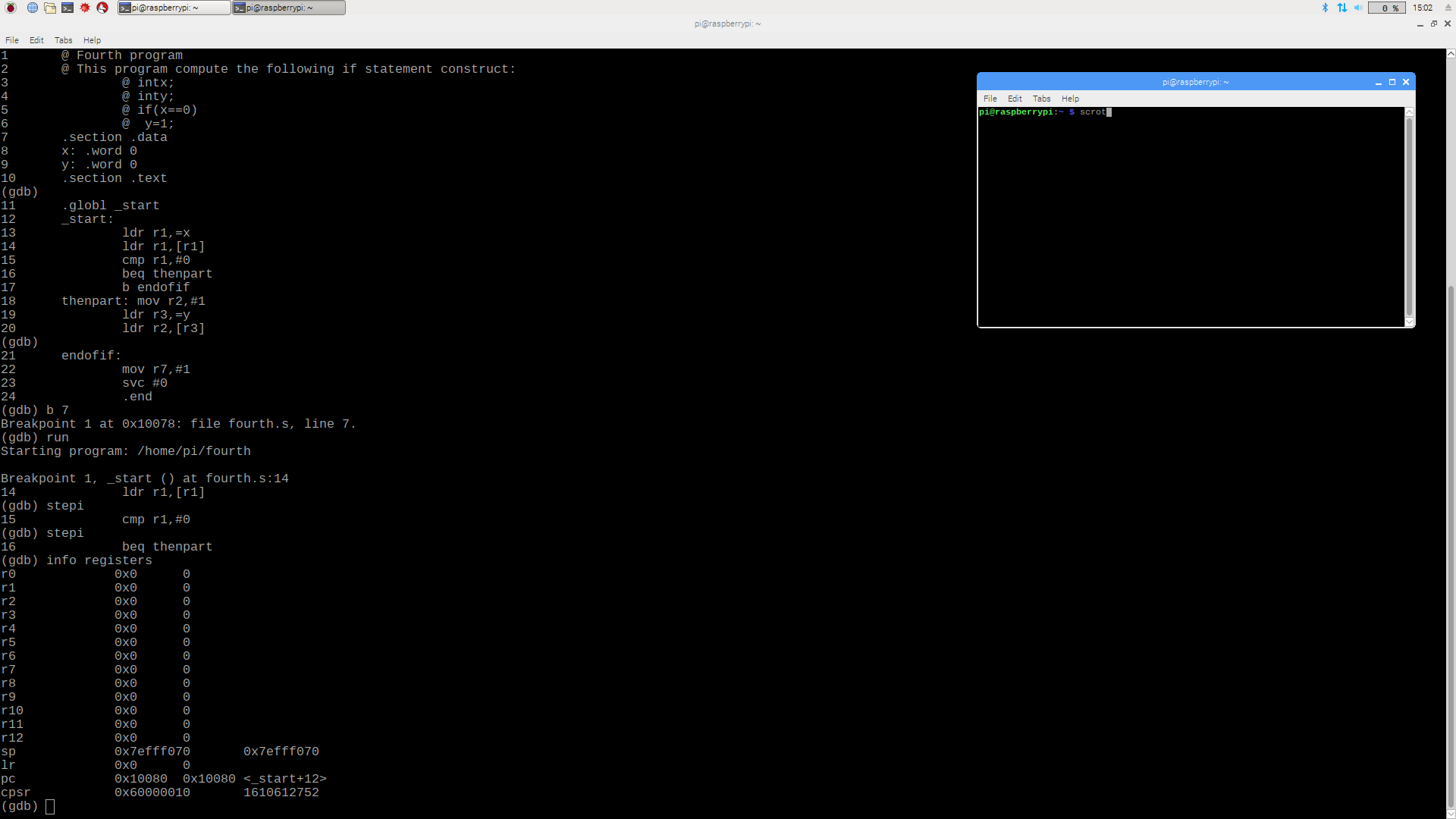


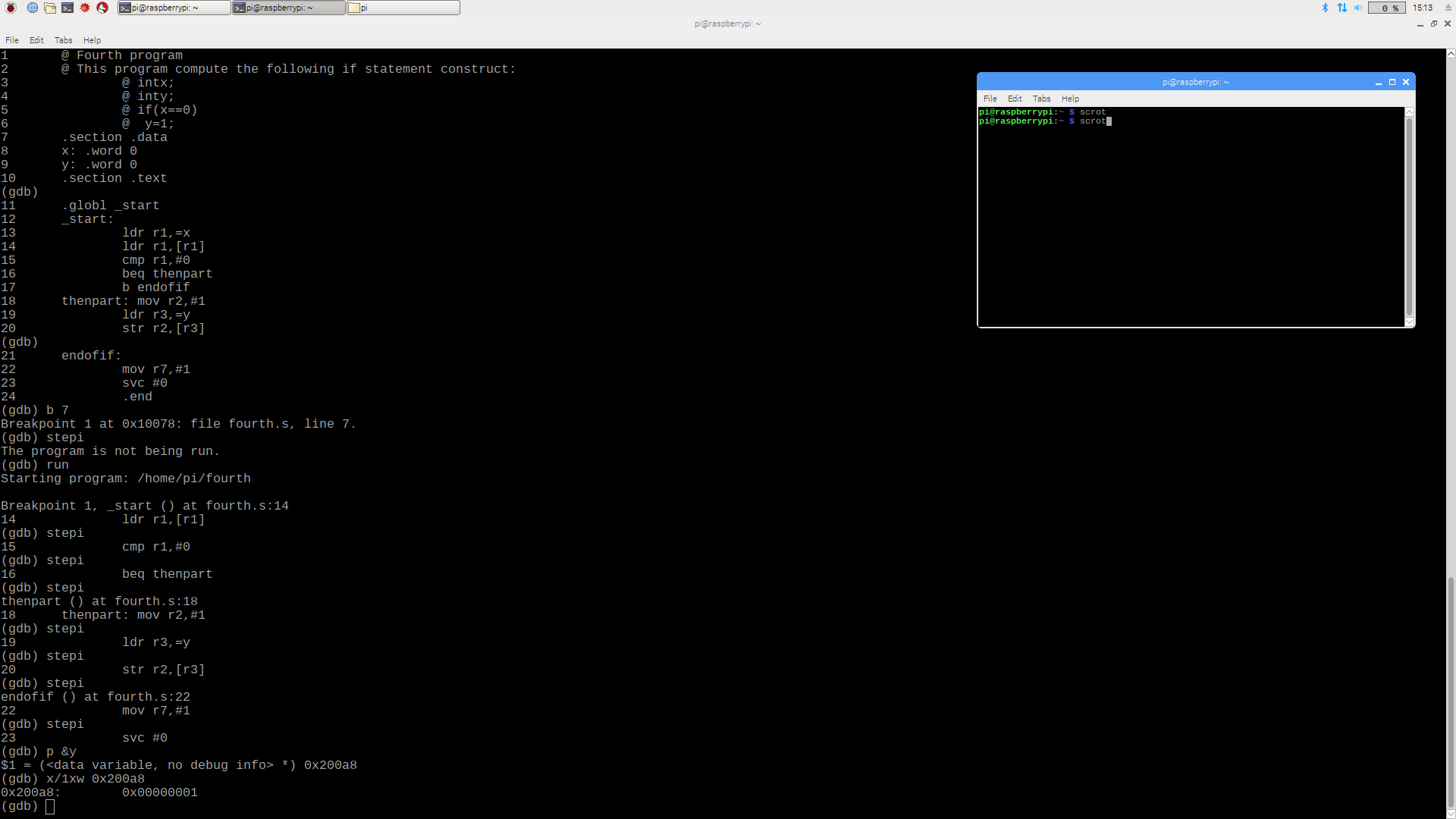
The last part of the assignment was to demonstrate how the master and worker strategy works on parallel programming. Here the master thread runs a different section of code, and workers run the other section. When we commented out the directive, “pragma omp parallel” it only printed the master thread as it was shown on picture #5. Editing the code again and using the pragma directive this time the program gave us a different output. The master creates worker and sends initial values to workers. Pragma limits the execution of a block to a single thread. The master thread would remain the same if the “id == 0”. It would print, “Greetings from the master, # 0 of 1 threads”. The rest would print similar statements, but the word “master” would be replaced by “worker”, as shown in picture#6.

ARM Assembly Programming

**Part 1:**

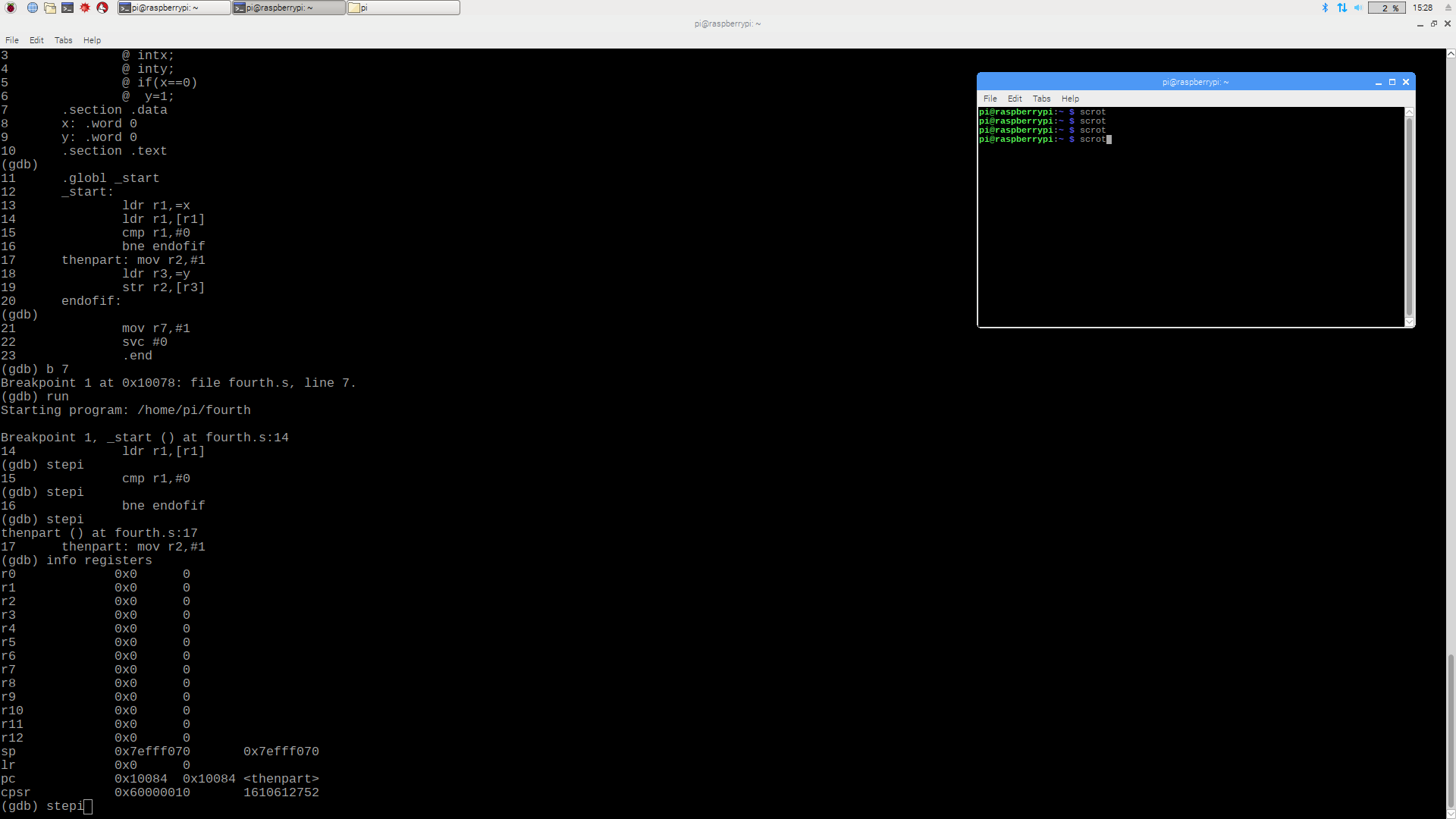
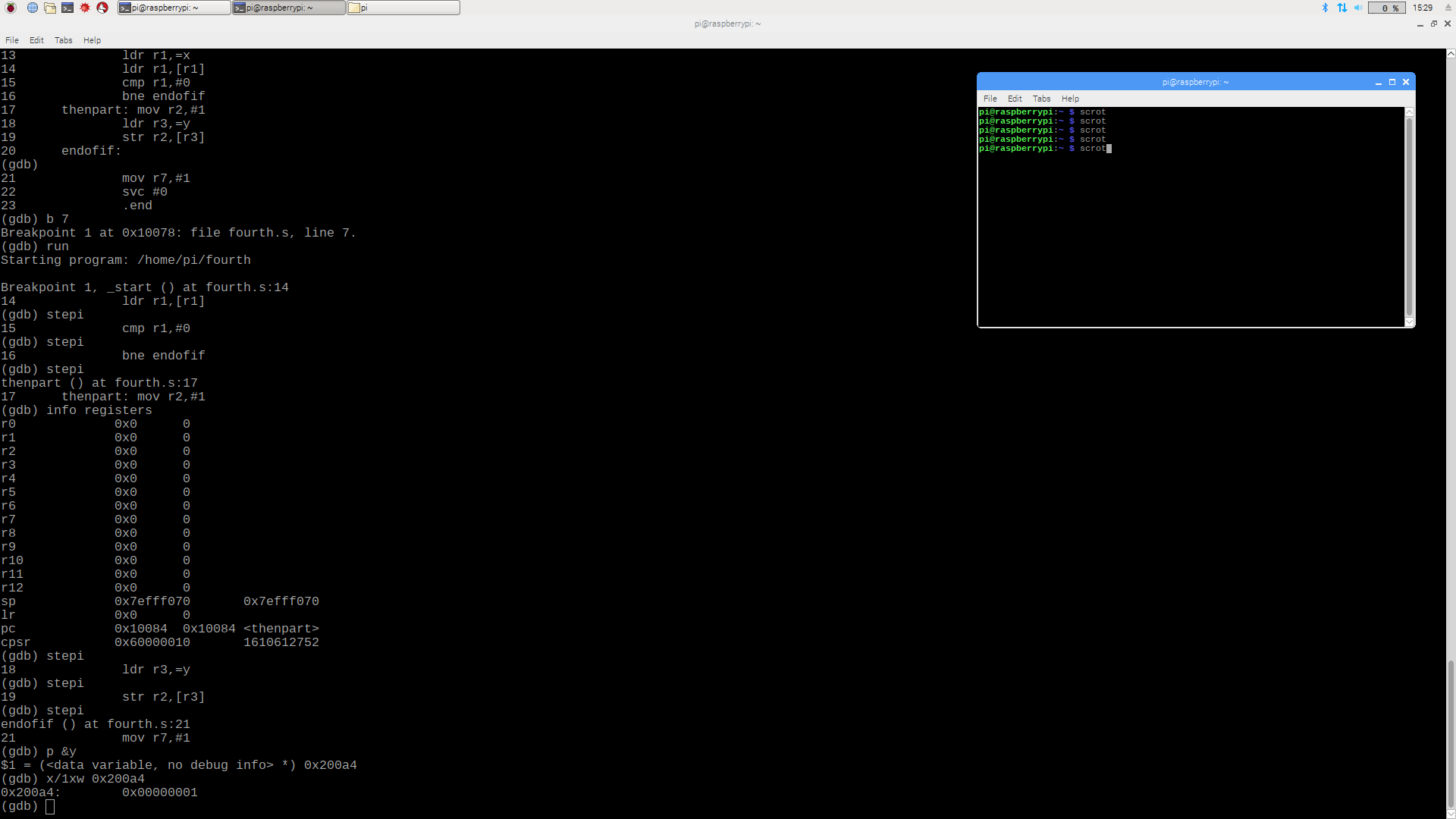
For part 1 we simply typed out all the code supplied in the instructions. We went through the steps given next in order to assemble and view with gdb. We used ‘p & a’ to find the address of the variable y = 1, then we used the x/1xw. First, we put the x and y word value 0, then we started loading into the registers to find if y = 1. It started from loading x value in r1, then we stored that value in r1, and we compared r1 with 0. If the comparison was true, then it would go to thenpart, otherwise if it was false, it would go to endofif, such that the value was true so the Z flag had the value 1 because you could see the register below 0x60000010 (6 in decimal is 0110). Before continuing with the code, we put a breakpoint on line 7 and ran the entire program by stepi. There was a mistake in the assignment on line 20, it is supposed to have str instead of ldr, because we cannot store two different registers. Hence, we proved the value y =1.



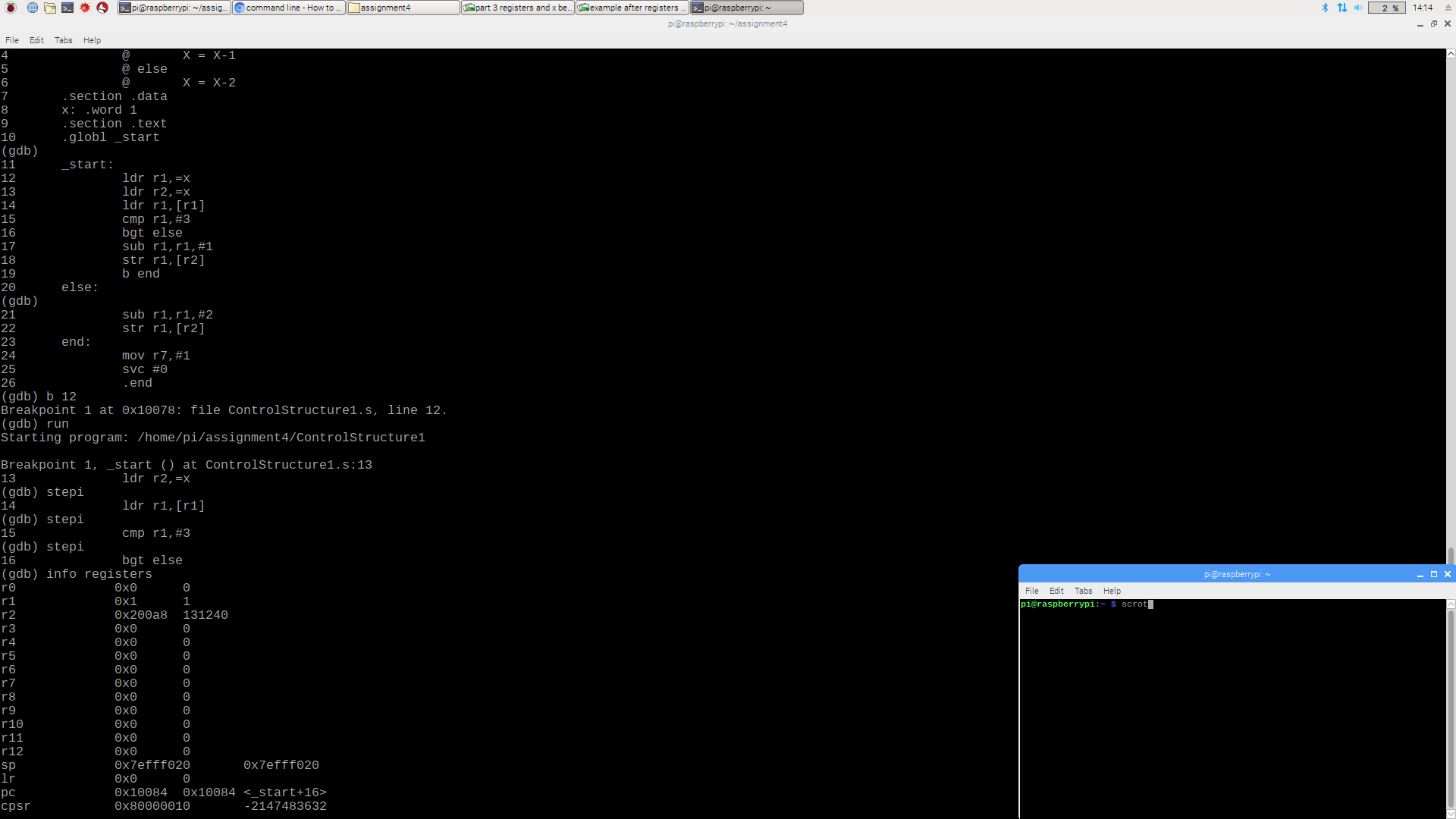


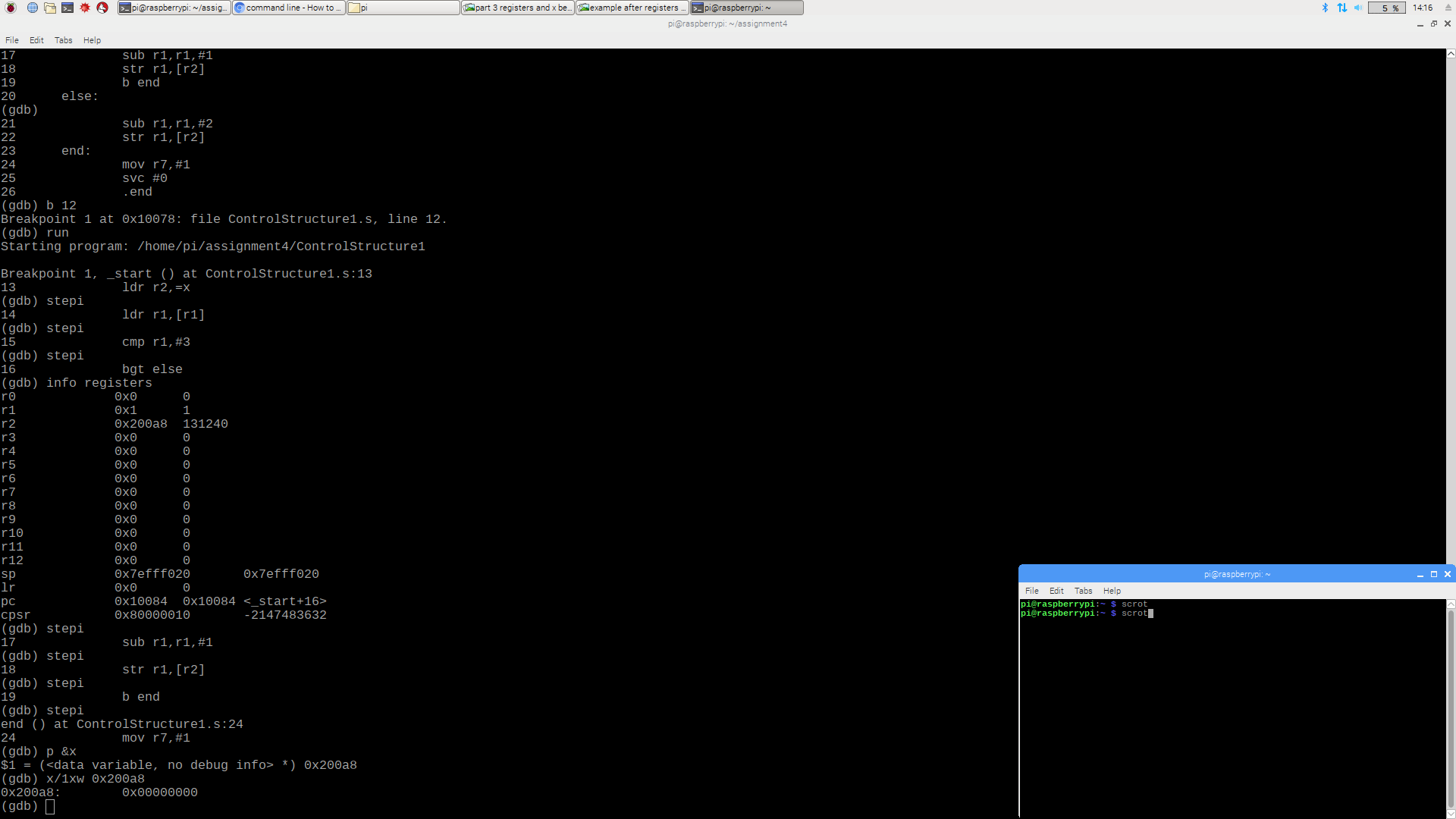
**Part 2:**

For part 2 we removed beq, then parted and converted line 16 from b endofif to bne endofif because if branch on is not equal (z = 0) then it goes to endofif part but eventually z value is 1, exact same as part 1. Also y value is 1.

**Part 3:** 

For part 3 we changed the entire program and name it ControlStructure1. First, we put the comment that if X less then or equal to 3, X is equal to x – 1 else X is equal to x – 2. We take two registers r1 and r2 to load then store r1 value in r1 and compare with =3. We use bgt means branch on greater than. If the value is greater than 3 it subtracts 1 from r1 and str r2 value in r1 but in this case the value of r1 is not greater than 3 so it goes to else statement and subtract r1 value from 2 and store the r2 value in r1. As you can see the registers cpsr value 0x80000010, so z flag value is 1 because 8 is 1111. After running the program by stepi, we put gdb p &x to find the data variable which is 0x200a8 and then gdb x/1xw 0x200a8 to find the x value which is 0.





Appendix

Slack: https://app.slack.com/client/TN3HREW9X/CN3HRGB33

GitHub: <https://github.com/Quinary-GSU>

Youtube Video: <https://www.youtube.com/watch?v=Jb-isV8DCmg&feature=youtu.be>