Developing Soft and Parallel Programming Skills Using Project-Based Learning

Fall 2019

Group Name: The Snakes

Group Members: Austin Yuille, Nabeeha Ashfaq, Jose Diaz, Matthew Hayes, Micah Robins

Planning and Scheduling

Name	Email	Task	Duration (hours)	Dependency
Jose Diaz	jdiaz28@student.gsu.edu	Completed Task 3 questions and answers	6	Google docs and Intro to Parallel Computing Document
Austin Yuille	ayuille1@student.gsu.edu	Created and organized the Github repository	6	Github and Raspberry Pi
Micah Robins	mrobins1@student.gsu.edu	Team Coordinator and Wrote the ARM Assembly report	6	Google Docs and Raspberry Pi
Matt Hayes	mhayes37@student.gsu.edu	Wrote the Parallel Programming report	6	Google Docs and Raspberry Pi
Nabeeha Ashfaq	nashfaq1@student.gsu.edu	Created and directed video presentation	6	Youtube and video editor

Parallel Programming Skills

a) Foundation:

- Race condition:
  - What is a race condition?

        - A race condition occurs when uncontrollable events that must occur in sequence are the determining factor in achieving a set result. If these events are not completed in sequence then the output is not correct, however you are unable to control when the events take place and as such the events are “racing” to see who finishes first.
    - Why race condition is difficult to reproduce and debug?
        - Race conditions are difficult to debug because the times that they do present problems are completely random. When you go back to debug the processes, the timing of the threads could have changed to the point where there is no problem or they could not have and thus the problem is gone. As such it is better to avoid race conditions in general using good software design.
    - How can it be fixed? Provide an example from your Project_A3(spmd2.c)
        - Something such as the barrier program from Project_A3 could be used. The barrier prevented certain things from being executed until all threads had completed their given tasks. This could prevent threads from accessing values out of sequence.
- Summarize 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).
    - When writing parallel programs there a couple of patterns that developers have picked up on overtime. These patterns can be grouped into two categories. Strategies and concurrent execution mechanisms. There are generally two strategies to consider. Algorithmic strategies, or determining what can be done using parallel code, and implementation strategies, how to implement the code. Implementation strategies usually deal more with the program’s structure instead of data structures. Concurrent execution has two primary patterns, process/thread control and coordination. Process control determines how processors will be handle the process or thread. Coordination patterns deal with how multiple processors coordinate with each other. This usually falls into message passing, processes communicating across multiple processors, or mutual exclusion, threads communicating via a shared memory. There are two libraries that are used for the two different methods, MPI for message passing, and OpenMP for shared memory.
- In the section “Categorizing Patterns” in the “Introduction_to_Parallel_Computing_4.pdf” compare the following:
    - Collective synchronization (barrier) with Collective communication (reduction)
        - With collective synchronization, the barrier, the processes run in parallel but all reach a point where they all must be at the same state (execution wise) for all of them to continue. Meaning that in order for all of the process to finish, they all must complete an iteration together, and then move on to the next one. The barrier

        holds back the processes until they are all together and in sync. Once all the processes get to the barrier, then they can continue.
    - With collective communication, reduction, the threads are all modifying the same variables but at the end, the variable is being summed up correctly and not overwriting each other. This is why in trap-working we get the correct calculation for our integral.
  - Master-worker with fork join
    - In the master-worker we always assign the first thread, thread 0, the name and role of the master. Any thread that does not have the thread number 0 is given the role of a worker. The master will then take the problem and break it up so that the workers will execute it, once done the results of the computation go back to the master.
    - With fork join however the workload is split and all of the threads get to work, when the iteration is done they all join back together, return the results, and repeat until finished. Here there is no thread that has full control and each thread does work.
- Dependency: Using your own words and explanation, answer the following:
  - Where can we find parallelism in programming?
    - We can find parallelism in programming when we look at the task view and when looking at the program statements. We can also find parallelism when we find loops or blocks of code that are often repeated and can be improved upon. We can also look at how data is used and where it is stored. By looking at the resources of the program we can also find parallelism.
  - What is a dependency and what are its types (provide one example of each)?
    - A dependency is when an operation depends on the end of an earlier operation and as such can not run until the first one finishes. The second operation *depends* on the first one.
    - A statement can be either independent or dependent. If it is waiting to run due to another process then it is dependent, if it doesn't wait for another process then it is independent.
  - When a statement is dependent and when it is independent (Provide two examples)?
    - One example of an independent statement is when a program asks for input from the user and only needs one input. This statement does not rely on anything else and as such is independent
    - A dependent statement would be when an input for a program is needed but the input is only called once a function that calculates a new value is completed. When this function is done running, then the statement is called and as such it depends on the completing of the function before the statement is called.
  - When can two statements be executed in parallel?
    - Two statements can be executed in parallel if and only if there are no dependencies between the two statements

- How can dependency be removed?
    - We can remove a dependency by modifying the program. This can be done by rearranging statements and eliminating statements.
- How do we compute dependency for the following two loops and what type/s of dependency?
    - We can compute dependencies based on the IN and OUT of a statement.
    - IN is the set of variables that may be used in that statement.
    - Out is the set of variables that may be modified by the statement.
- for(i = 0; i < 100; i++){
    - S1: a[i] = i; }
    - In this loop there are no dependencies
- for(i = 0; i< 100; i++){
    - S1: a[i] - i;
    - S2: b[i] = 2 * i; }
    - In this loop, the first statement is dependent on the first one and as such the first must complete, then the second one completes, and then the next iteration can occur.

b) Parallel Programming Basics:

The parallel programming part of Task 3 involved us creating 4 different programs. Before we started creating any code, though, we started by making a new repository for this project in our github account and a new folder on our Raspberry Pi for our files and linked them together. That way we can more easily push our files to the github repository.

The first program, trap-notworking.c attempts to calculate an integral using the trapezoidal rule. We created it in nano and added the provided code from our instructions. You can see our code in Appendix B,1. After compiling and running this program, passing a value of 4 for the thread count, we were able to observe the results. The answer to the integral that this program is supposed to be calculating is 2. However, when we ran the trap-notworking program it gave us an answer of 1.533800. The output of this program can be seen in Appendix B,2. Our theory as to why the program is outputting the incorrect answer is that the threads are sharing variables and then overwriting these variables as they work in parallel.

The next step was to create a program that actually calculates the integral correctly. Fortunately, the correct code was provided to us. It was a small change to go from the incorrect trap-notworking.c to the correct trap-working.c. We copied the original code from trap-notworking.c and created the file trap-working.c from it. Then we changed line 38. In the original program line 38 read
“#pragma omp parallel for private(i) shared (a, n, h, integral)”
In the new trap-working.c program we changed lines 38 and 39 to read
“#pragma omp parallel for \
private(i) shared (a, n, h) reduction(+: integral)”

You can see our code in Appendix B,3. The new result is that the threads do not overwrite each other, but instead sum together properly and output a result of 2. Appendix B,4 contains a screen shot of our output from both trap-notworking.c (after running it again) and the corrected trap-working.c programs. Both instances were passed a value of 4 for the thread count. But only trap-working.c provided the correct answer of 2.

Next, the third program we were asked to create explores the use of the OpenMP barrier command. The provided code runs 2 print statements. One before the barrier pragma and another one after. With the "#pragma omp barrier" line commented out we noticed that the threads from the BEFORE print statement and the threads from the AFTER print statement were all intermingled, as can be seen in our output (Appendix B,5). In contrast, once the barrier pragma was active all of the BEFORE threads printed first, and the AFTER threads did not run until the BEFORE threads had finished. This output can be seen in Appendix B,6 and the code with the active barrier pragma can be found in Appendix B,7.

Finally, the fourth program we ran for the parallel programming exercise was named masterWorker.c. In the code we were provided with, there is an if…else statement where the first thread (with an ID of 0) is treated as the master within the if statement. In the else statement every other thread (that has an ID of any number aside from 0) is treated as the worker threads. However, the parallel pragma is commented out. As a result our output did not include any worker threads. Only the master thread printed because there is no parallelism happening. See Appendix B,8 for the output of the code with the pragma commented out.

After removing the comment for the parallel pragma, we saw that our output changed to include the worker threads as well. This makes sense considering that this pragma is what allows for parallel processing and the worker threads will never print unless multiple threads are created. You can see our output with the parallel pragma active in Appendix B,9 and our code is in Appendix B,10.

ARM Assembly Programming

Part 1:

The first part of Arm Assembly is compiling and analyzing a pre-given "if-then" branch program, named *fourth.s* (Appendix C,1). The point of the program is to establish .WORD memory values **x** and **y**, both respectively equating to **0**. The value of x is loaded into register **r1**. Using the **cmp r1, #0** statement, we change the value of the **y** memory to **1** if the value of **r1** is currently **0**. If the value of **r1** is **0**, we jump using **beq thenpart** to load the value of **1** in **r2** to the address of **y** in r3. If **x** does not equal **0**, we skip over **beq thenpart** and use **b endofif** to jump to the exit. The value of **y** remains **0** if we do not jump to **thenpart**. Since the value of **r1** is **0**, the final value of **y** is **1 (**Appendix C,2).

Part 2:

The second part of Arm Assembly is a modification of the first part. Rather can create two branching statements, it would be more efficient to use only one.

As has been established in Part 1, when a **cmp** statement returns false, it will skip over its accompanying jump (**b**) statement. We can replace **beq** (jump if equal) with **bne** (jump if not equal) and place only the statements of **thenpart** immediately after the **bne** jump condition statement. The second **b** jump is removed as it will no longer be needed. Modified code at (Appendix C,3). When set up, the program will jump to the exit when **x** does not equal **0**. If **x** does equal **0**, the program will skip over the **bne** jump statement into the statement changing **y** to **1**. (Appendix C,4) After changing **y**, the program will naturally proceed to the exit statement (Appendix C,5). Before exiting, the cpsr register reads **0x60000010**, indicating that the Zero-flag and Carry-flag have both been triggered (Appendix C,6).

Part 3:

The third part gives the following expression to be converted to Assembly:

if X <= 3

X = X-1

else

X = X-2

X indicates a 32-bit integer memory variable with value 1. The converted program is to be named *ControlStructure1.s*

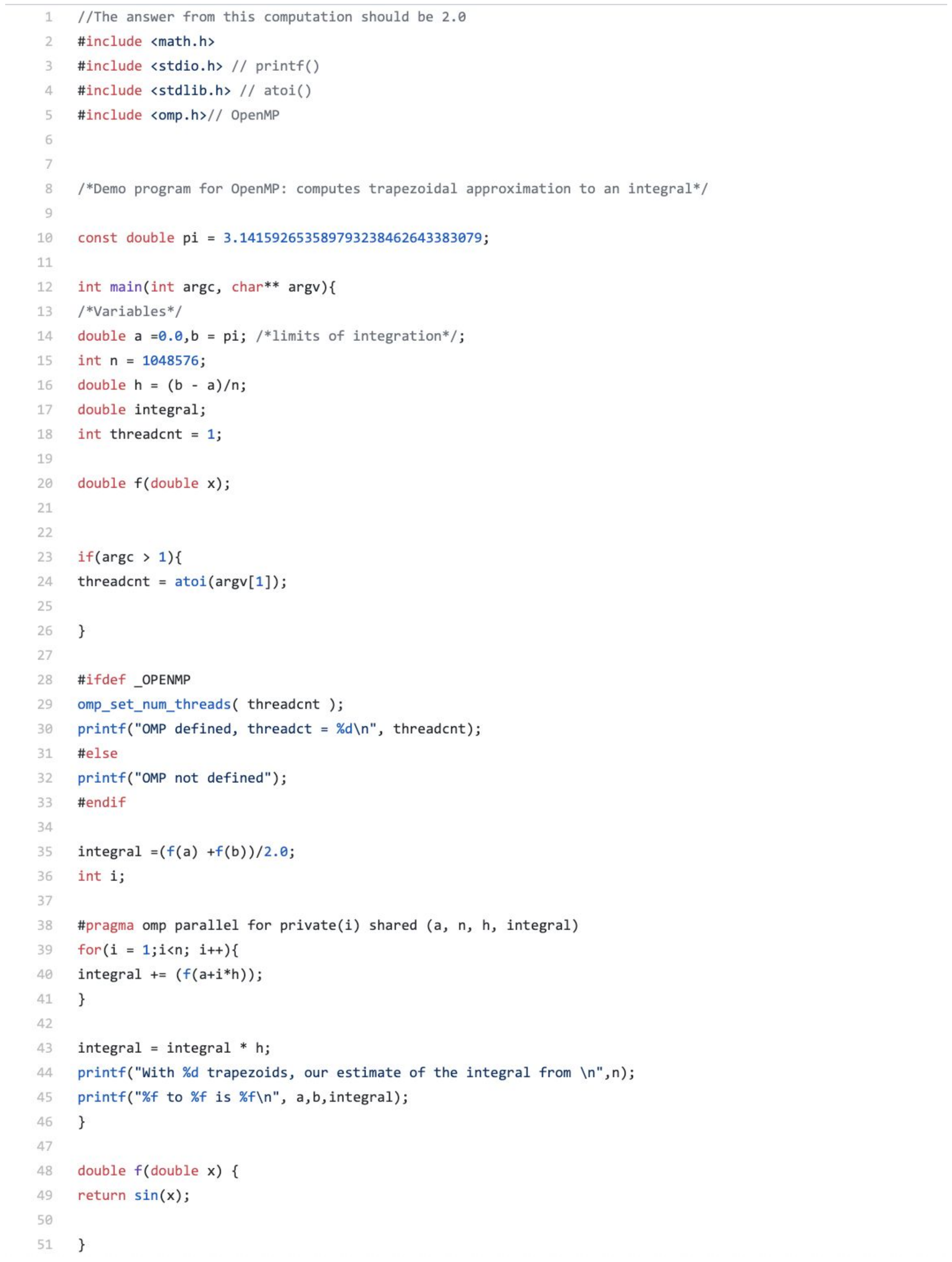
The resulting program creates a .WORD memory variable **x** with assigned value **1**. The address and value of **x** is loaded into register **r1**. Using **cmp r1, #3**, we compare the value of **x** (being **1**) to immediate **3**, following with conditional jump **bgt false** (Appendix C,7 & C,8). If the value of **r1** is greater than **3**, then the program will jump to **false**. **false** contains the statement **sub r1, r1, #2**; this will cause **r1** to subtract **2** from its value, then naturally proceed to the exit. If the value of **r1** is not greater than the value of **3**, the program will skip over the **cmp** and **bgt** jump statements to proceed to **sub r1, r1, #1** and **b next** (Appendix C,9). The two statements will first subtract **1** from the value of **r1**, then jump directly to **next**. **next** leads directly to the exit, so the program will completely skip the **false** statements. Final code seen at (Appendix C,6).By this program, if **x** is greater than **3**, it will subtract **2** from itself; if it is less than or equal to **3**, it will subtract **1** from itself. Because the original X memory variable is to be assigned value **1** in the beginning, variable **x** originally equals **1**, subtracts **1** from itself, and finally results in **0**. Cpsr register reads **0x80000010**, indicating the negative flag was triggered, but not the zero flag (Appendix C,10). The zero flag should be triggered, but there was a mistake with the code. Instead of using **sub r1, r1, #1**, **subs r1, r1, #1** should have been used to affect the cpsr register.

Appendix A: Links

Github Project: https://github.com/orgs/thesnakes-csc3210/projects/1
Github Repository: https://github.com/thesnakes-csc3210/AssignmentA4
Video Presentation: https://youtu.be/VIWLwID7HmA

Appendix B: Parallel Programming Task A4

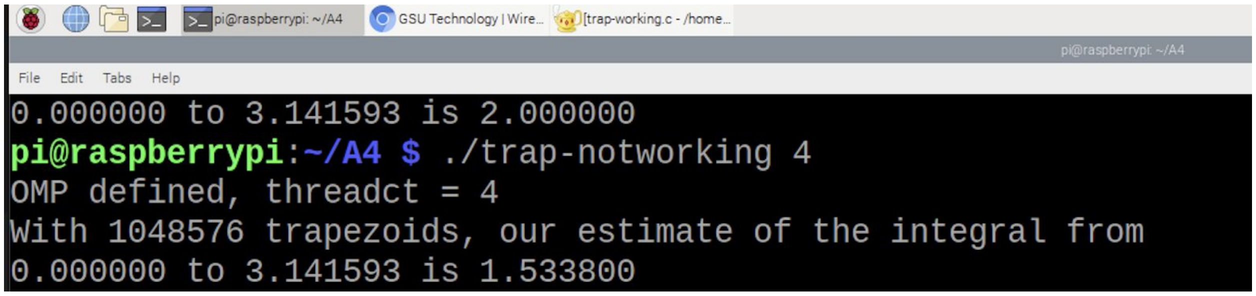
(B,1)



```c
//The answer from this computation should be 2.0
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h>// OpenMP


/*Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.141592653589793238462643383079;

int main(int argc, char** argv){
/*Variables*/
double a =0.0,b = pi; /*limits of integration*/;
int n = 1048576;
double h = (b - a)/n;
double integral;
int threadcnt = 1;

double f(double x);


if(argc > 1){
threadcnt = atoi(argv[1]);

}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral =(f(a) +f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared (a, n, h, integral)
for(i = 1;i<n; i++){
integral += (f(a+i*h));
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n",n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);

}
```

Code snippet of trap-notworking.c

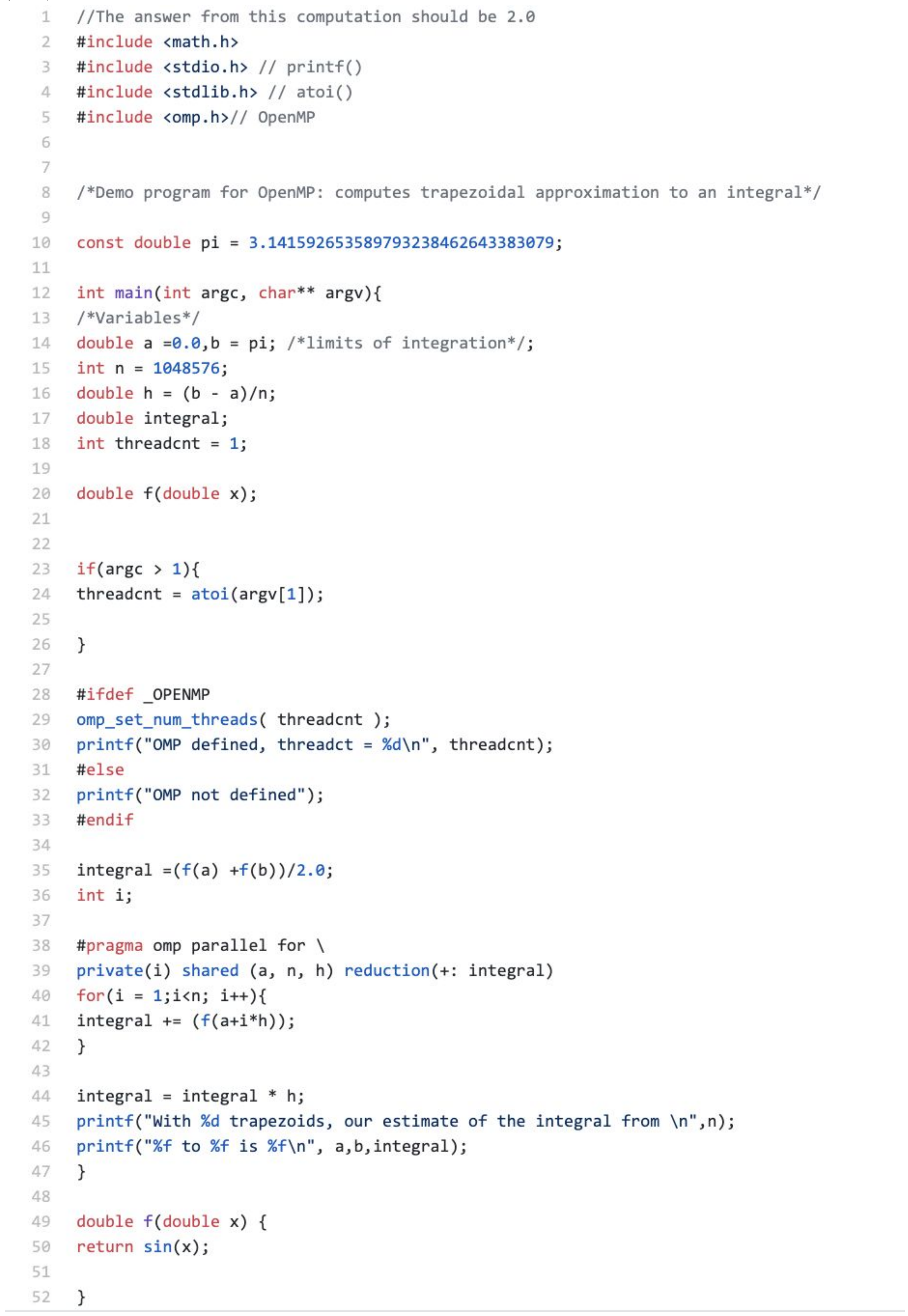
(B,2)



```
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~/A4 $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.533800
```

Result of running the trap-notworking.c program

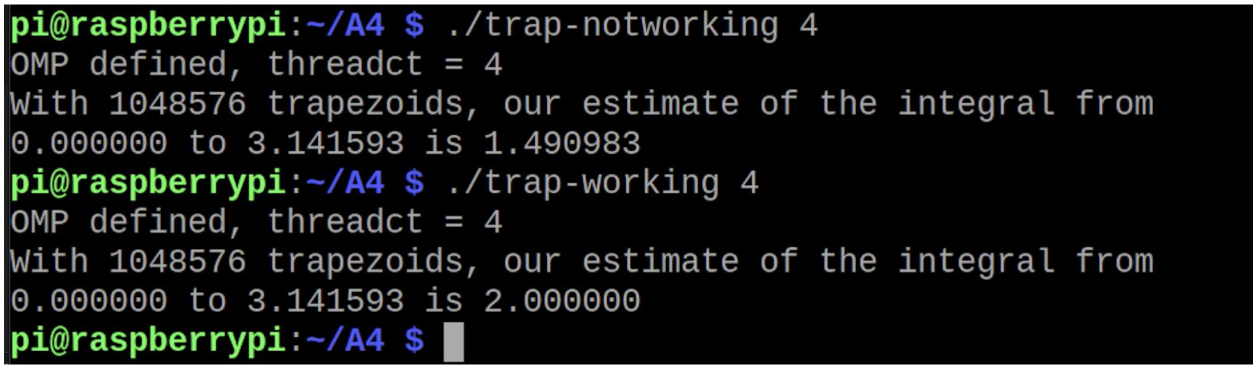
(B,3)



```
//The answer from this computation should be 2.0
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h>// OpenMP


/*Demo program for OpenMP: computes trapezoidal approximation to an integral*/

const double pi = 3.141592653589793238462643383079;

int main(int argc, char** argv){
/*Variables*/
double a =0.0,b = pi; /*limits of integration*/;
int n = 1048576;
double h = (b - a)/n;
double integral;
int threadcnt = 1;

double f(double x);


if(argc > 1){
threadcnt = atoi(argv[1]);

}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadct = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral =(f(a) +f(b))/2.0;
int i;

#pragma omp parallel for \
private(i) shared (a, n, h) reduction(+: integral)
for(i = 1;i<n; i++){
integral += (f(a+i*h));
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n",n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);

}
```

Code snippet of trap-working.c

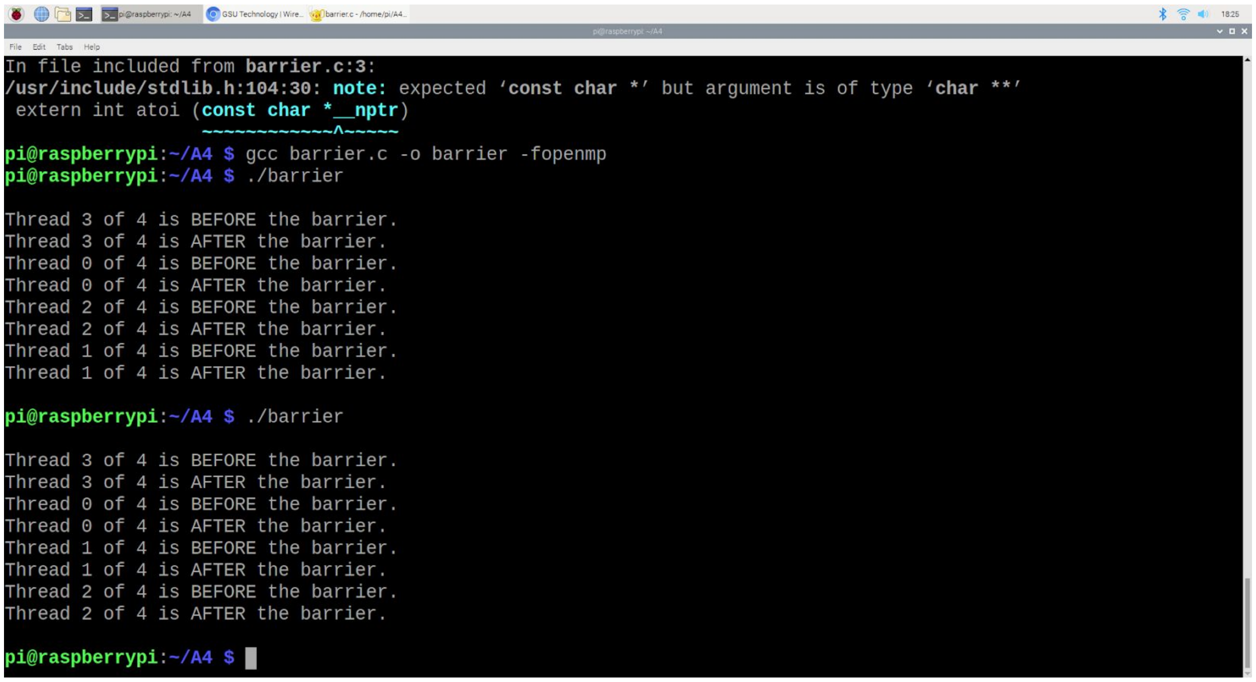
(B,4)



```
pi@raspberrypi:~/A4 $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.490983
pi@raspberrypi:~/A4 $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~/A4 $ 
```

Results from running both the trap-notworking.c and trap-working.c programs

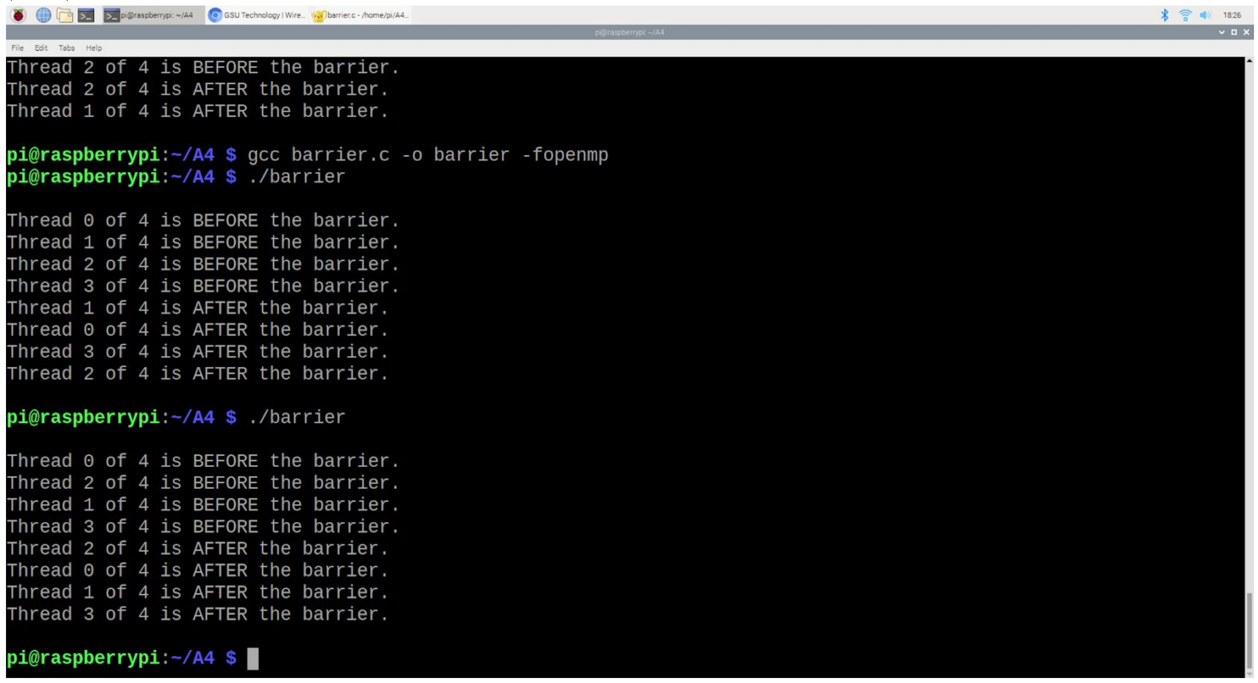
(B, 5)



```
In file included from barrier.c:3:
/usr/include/stdlib.h:104:30: note: expected ‘const char *’ but argument is of type ‘char **’
 extern int atoi (const char *__nptr)
                  ~~~~~~~~~~~~^~~~~~
pi@raspberrypi:~/A4 $ gcc barrier.c -o barrier -fopenmp
pi@raspberrypi:~/A4 $ ./barrier

Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.

pi@raspberrypi:~/A4 $ ./barrier

Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.

pi@raspberrypi:~/A4 $ 
```

Result of the barrier.c program not utilizing the barrier pragma

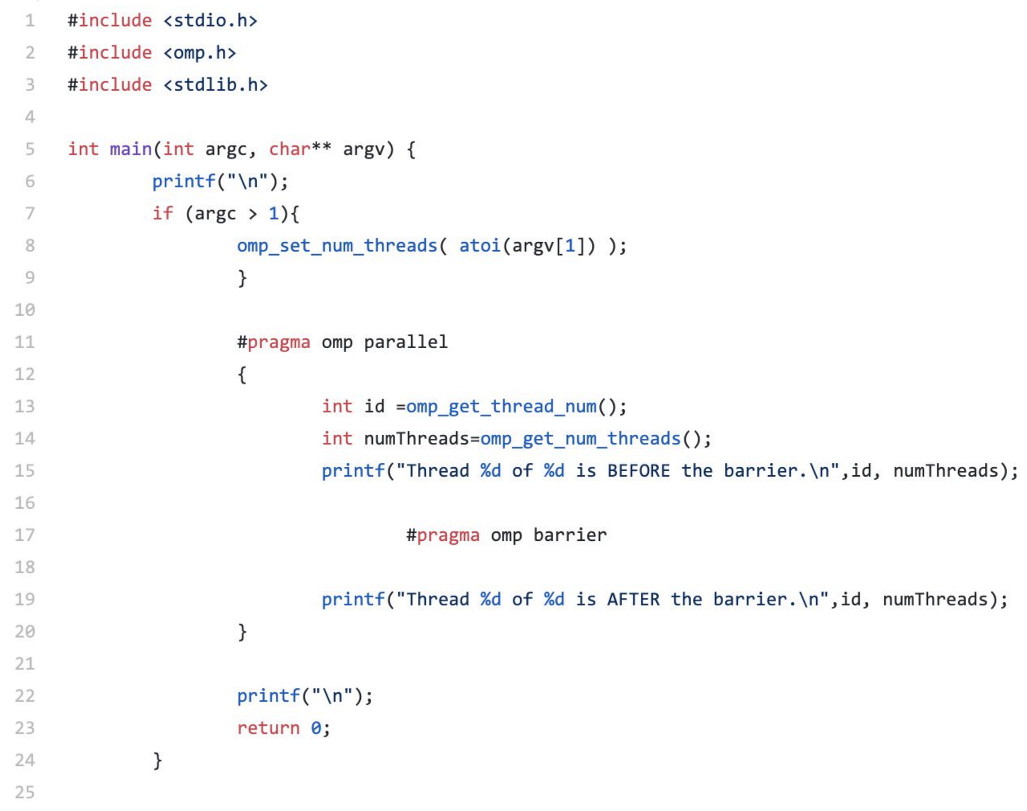
(B, 6)



```
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.

pi@raspberrypi:~/A4 $ gcc barrier.c -o barrier -fopenmp
pi@raspberrypi:~/A4 $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.

pi@raspberrypi:~/A4 $ ./barrier

Thread 0 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 3 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.

pi@raspberrypi:~/A4 $ 
```

Result of the barrier.c program which includes the barrier pragma

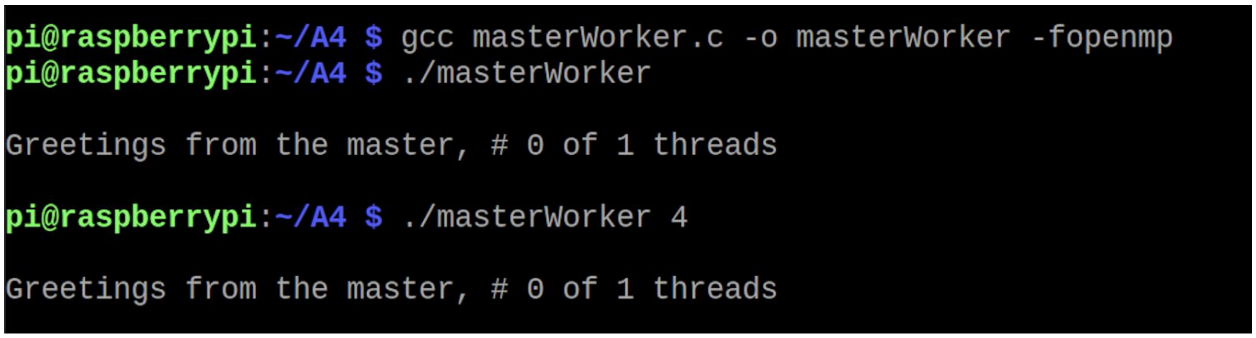
(B, 7)



```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
        printf("\n");
        if (argc > 1){
                omp_set_num_threads( atoi(argv[1]) );
                }

                #pragma omp parallel
                {
                        int id =omp_get_thread_num();
                        int numThreads=omp_get_num_threads();
                        printf("Thread %d of %d is BEFORE the barrier.\n",id, numThreads);

                                #pragma omp barrier

                        printf("Thread %d of %d is AFTER the barrier.\n",id, numThreads);
                }

                printf("\n");
                return 0;
        }

```

Code snippet of barrier.c

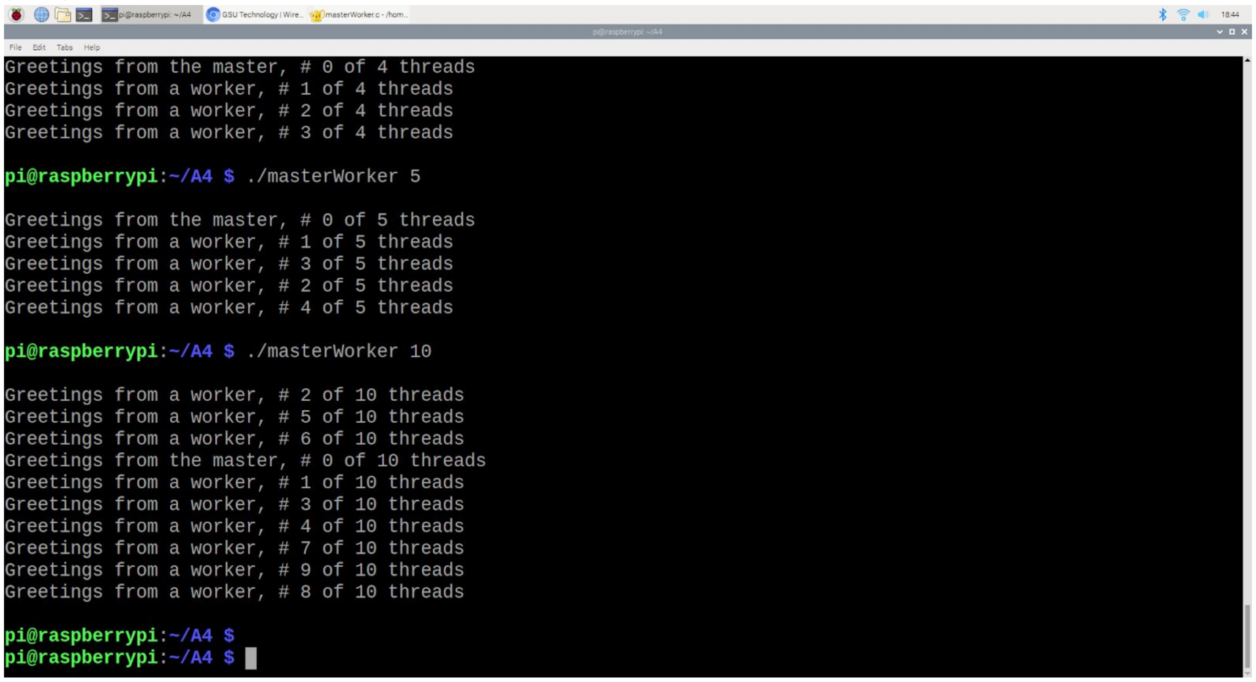
(B, 8)



```
pi@raspberrypi:~/A4 $ gcc masterWorker.c -o masterWorker -fopenmp
pi@raspberrypi:~/A4 $ ./masterWorker

Greetings from the master, # 0 of 1 threads

pi@raspberrypi:~/A4 $ ./masterWorker 4

Greetings from the master, # 0 of 1 threads
```

Result from the masterWorker.c program with the parallel pragma commented out

(B, 9)



```
Greetings from the master, # 0 of 4 threads
Greetings from a worker, # 1 of 4 threads
Greetings from a worker, # 2 of 4 threads
Greetings from a worker, # 3 of 4 threads

pi@raspberrypi:~/A4 $ ./masterWorker 5

Greetings from the master, # 0 of 5 threads
Greetings from a worker, # 1 of 5 threads
Greetings from a worker, # 3 of 5 threads
Greetings from a worker, # 2 of 5 threads
Greetings from a worker, # 4 of 5 threads

pi@raspberrypi:~/A4 $ ./masterWorker 10

Greetings from a worker, # 2 of 10 threads
Greetings from a worker, # 5 of 10 threads
Greetings from a worker, # 6 of 10 threads
Greetings from the master, # 0 of 10 threads
Greetings from a worker, # 1 of 10 threads
Greetings from a worker, # 3 of 10 threads
Greetings from a worker, # 4 of 10 threads
Greetings from a worker, # 7 of 10 threads
Greetings from a worker, # 9 of 10 threads
Greetings from a worker, # 8 of 10 threads

pi@raspberrypi:~/A4 $
pi@raspberrypi:~/A4 $
```

Result from the masterWorker.c program using the parallel pragma properly

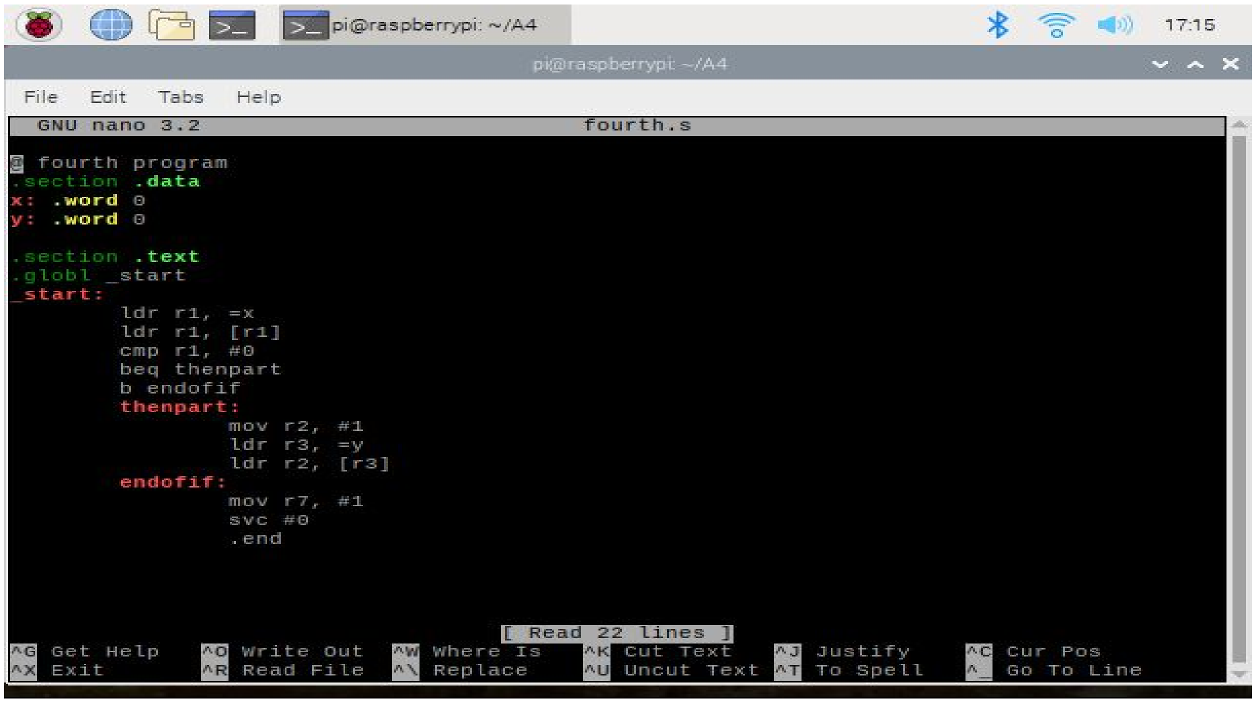
(B, 10)

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (int argc, char** argv) {
printf("\n");
if (argc>1) {
omp_set_num_threads(atoi(argv[1]));
}

        #pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();

if (id==0) {
printf("Greetings from the master, # %d of %d threads \n", id, numThreads);
}else{
printf("Greetings from a worker, # %d of %d threads\n", id, numThreads);

}
}

printf("\n");

return 0;
}

```

Code snippet from the program masterWorker.c

Appendix C: ARM Assembly Programming A4

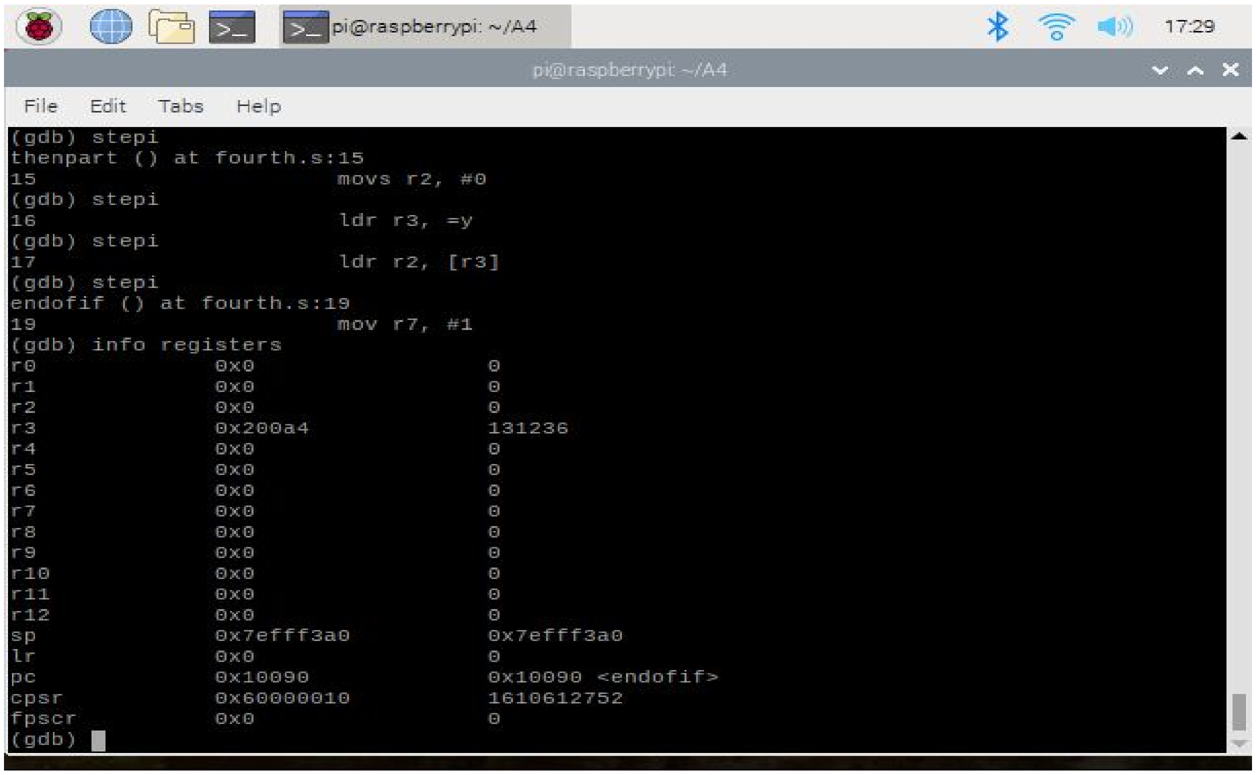
(C, 1)



```
  GNU nano 3.2                                        fourth.s

@ fourth program
.section .data
x: .word 0
y: .word 0

.section .text
.globl _start
_start:
        ldr r1, =x
        ldr r1, [r1]
        cmp r1, #0
        beq thenpart
        b endofif
        thenpart:
                mov r2, #1
                ldr r3, =y
                ldr r2, [r3]
        endofif:
                mov r7, #1
                svc #0
                .end

                                                    [ Read 22 lines ]
^G Get Help     ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
^X Exit         ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^_ Go To Line
```

Part 1 code

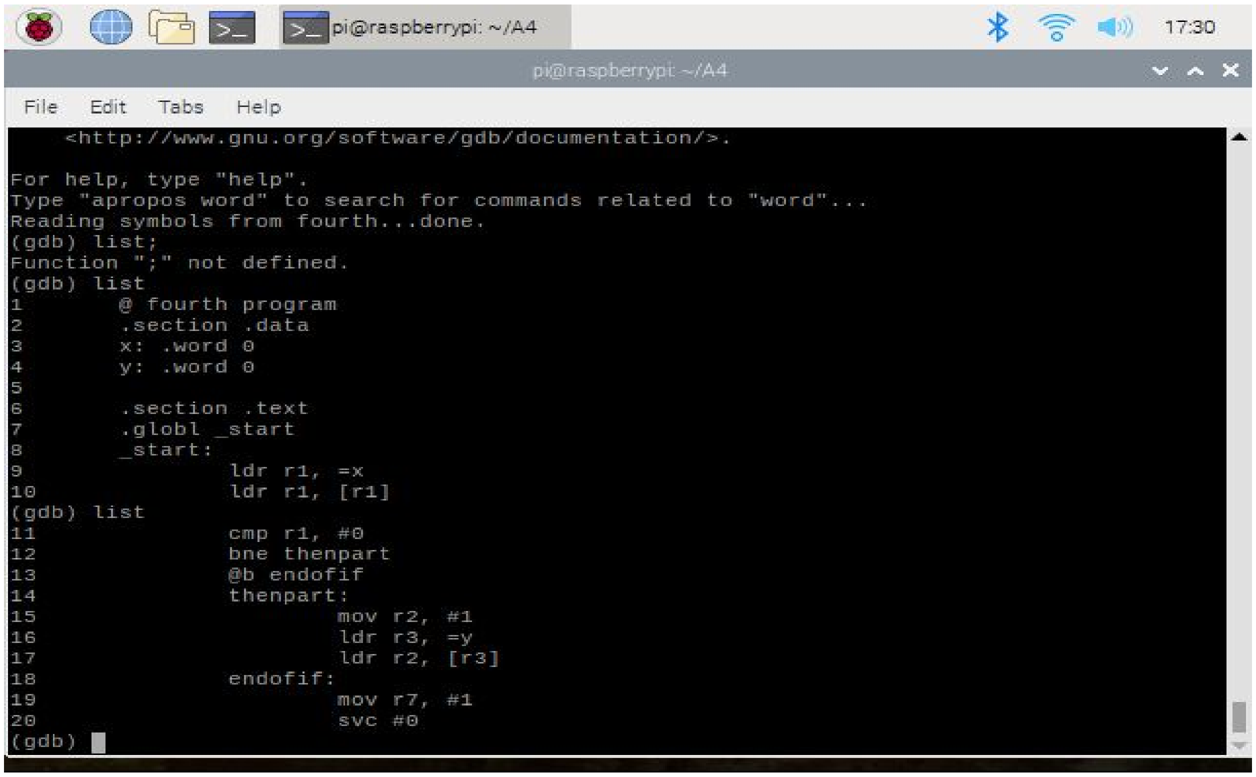
(C, 2)



```
(gdb) stepi
thenpart () at fourth.s:15
15                      movs r2, #0
(gdb) stepi
16                      ldr r3, =y
(gdb) stepi
17                      ldr r2, [r3]
(gdb) stepi
endofif () at fourth.s:19
19                      mov r7, #1
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x0                 0
r3             0x200a4             131236
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3a0          0x7efff3a0
lr             0x0                 0
pc             0x10090             0x10090 <endofif>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
(gdb)
```

Part 1 code result. R3

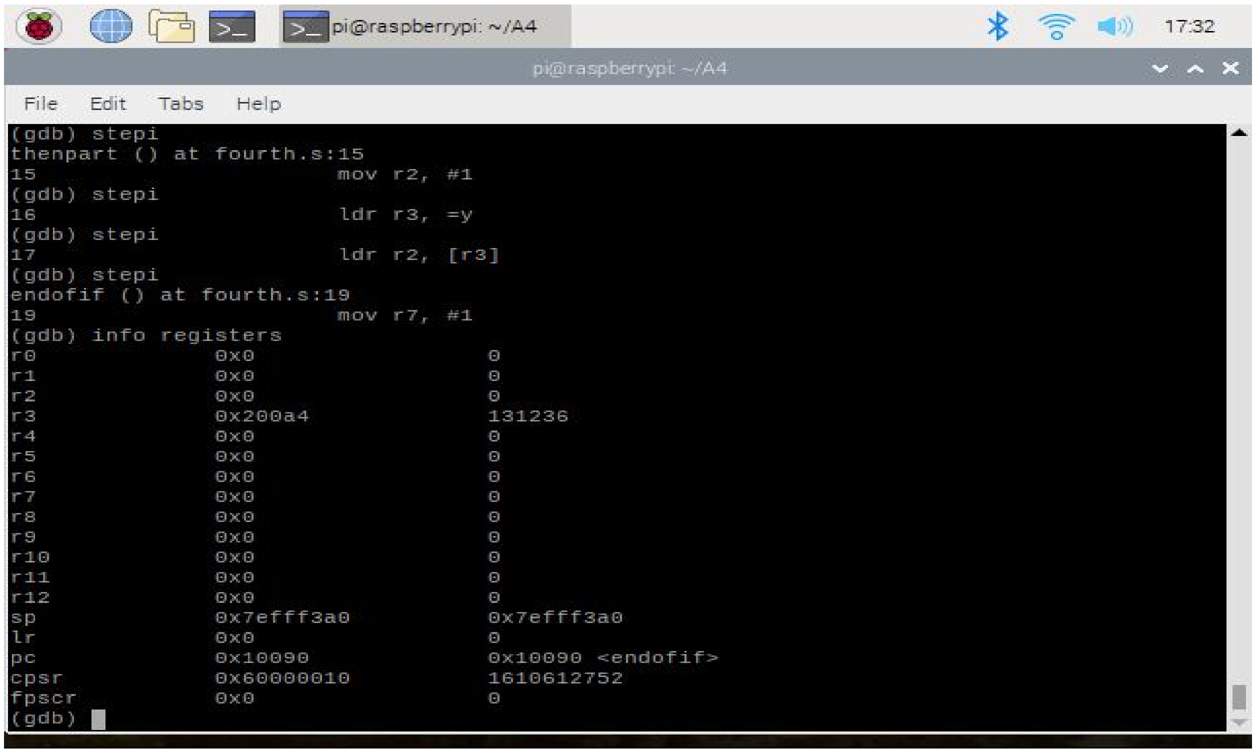
(C, 3)



```
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from fourth...done.
(gdb) list;
Function ";" not defined.
(gdb) list
1	@ fourth program
2	.section .data
3	x: .word 0
4	y: .word 0
5
6	.section .text
7	.globl _start
8	_start:
9		ldr r1, =x
10		ldr r1, [r1]
(gdb) list
11		cmp r1, #0
12		bne thenpart
13		@b endofif
14		thenpart:
15			mov r2, #1
16			ldr r3, =y
17			ldr r2, [r3]
18		endofif:
19			mov r7, #1
20			svc #0
(gdb) 
```

Modified Part 1 code. Commented out **b endofif**. Using bne instead of beq.

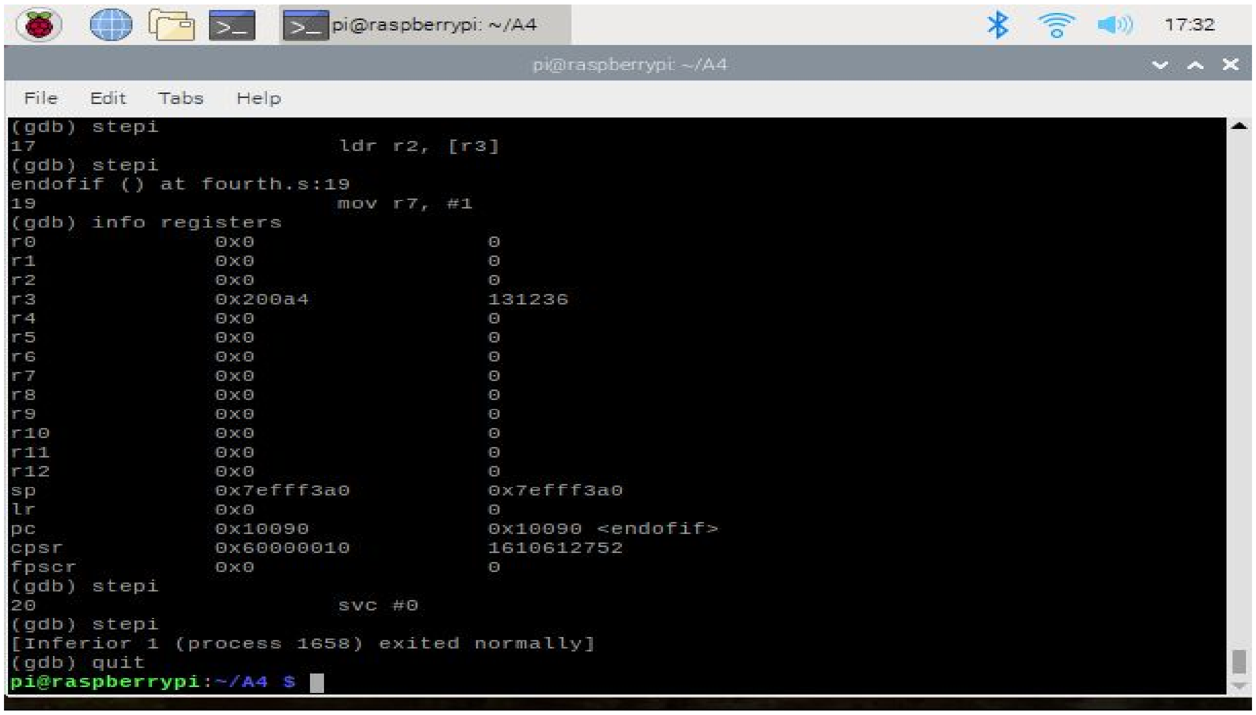
(C, 4)



```
(gdb) stepi
thenpart () at fourth.s:15
15			mov r2, #1
(gdb) stepi
16			ldr r3, =y
(gdb) stepi
17			ldr r2, [r3]
(gdb) stepi
endofif () at fourth.s:19
19			mov r7, #1
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x0                 0
r3             0x200a4             131236
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3a0          0x7efff3a0
lr             0x0                 0
pc             0x10090             0x10090 <endofif>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
(gdb) 
```

Skip over **thenpart**, to transfer the value of **1** to the **y** address.

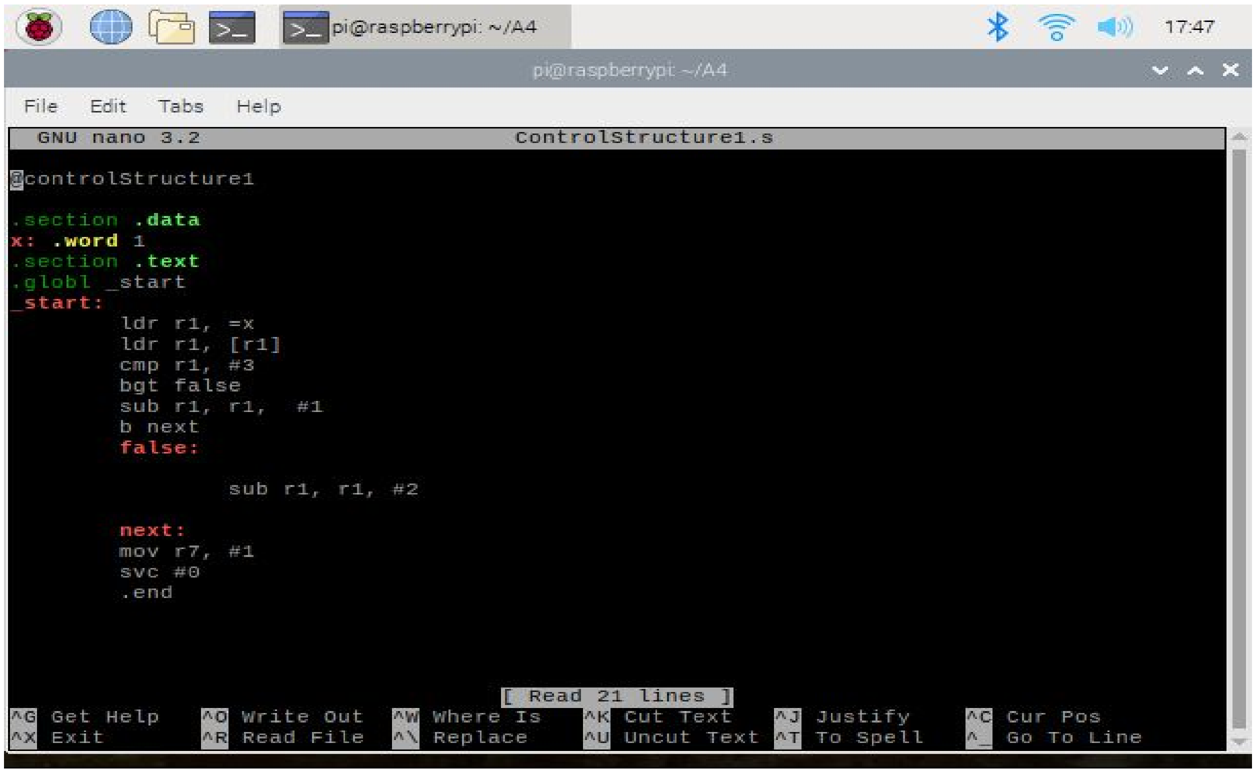
(C, 5)



```
(gdb) stepi
17                      ldr r2, [r3]
(gdb) stepi
endofif () at fourth.s:19
19                      mov r7, #1
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x0                 0
r3             0x200a4             131236
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3a0          0x7efff3a0
lr             0x0                 0
pc             0x10090             0x10090 <endofif>
cpsr           0x60000010          1610612752
fpscr          0x0                 0
(gdb) stepi
20                      svc #0
(gdb) stepi
[Inferior 1 (process 1658) exited normally]
(gdb) quit
pi@raspberrypi:~/A4 $
```

Before exiting, cpsr register reads 0x60000010. The Zero Flag and the Carry Flag are triggered.

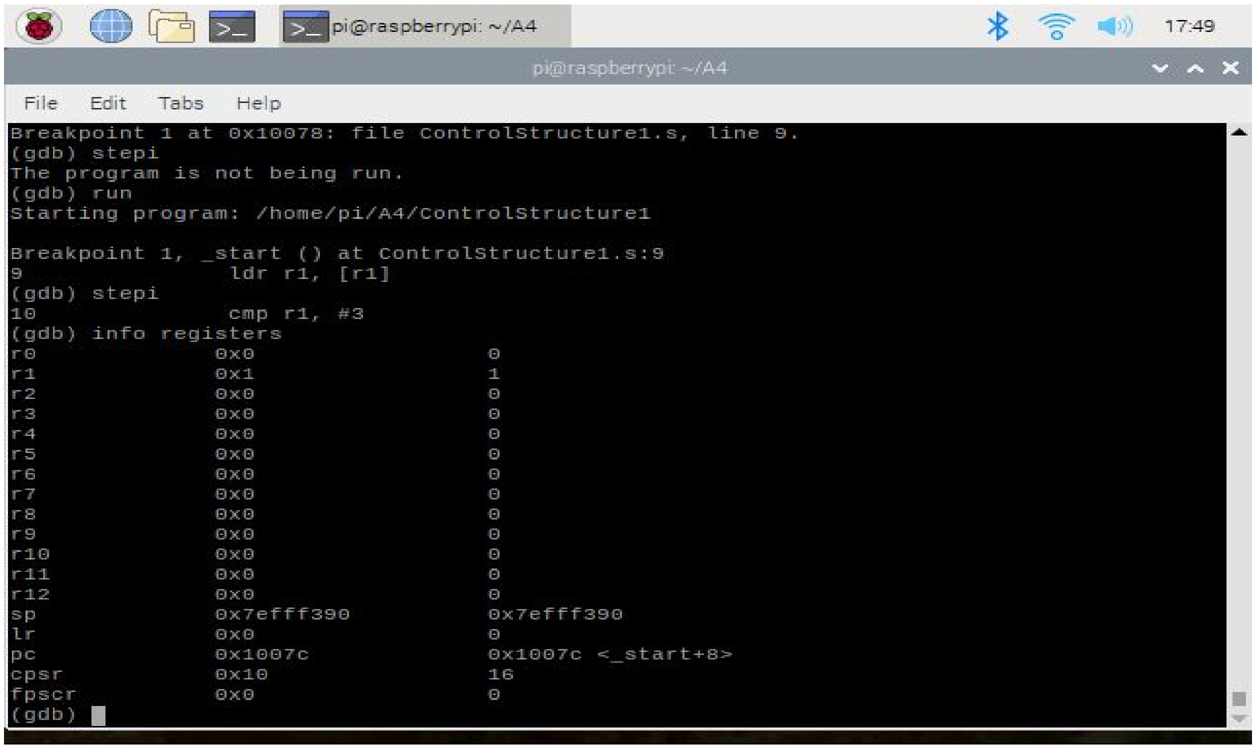
(C, 6)



```
@controlStructure1

.section .data
x: .word 1
.section .text
.globl _start
_start:
        ldr r1, =x
        ldr r1, [r1]
        cmp r1, #3
        bgt false
        sub r1, r1,  #1
        b next
        false:

                sub r1, r1, #2


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

Code for Part 3

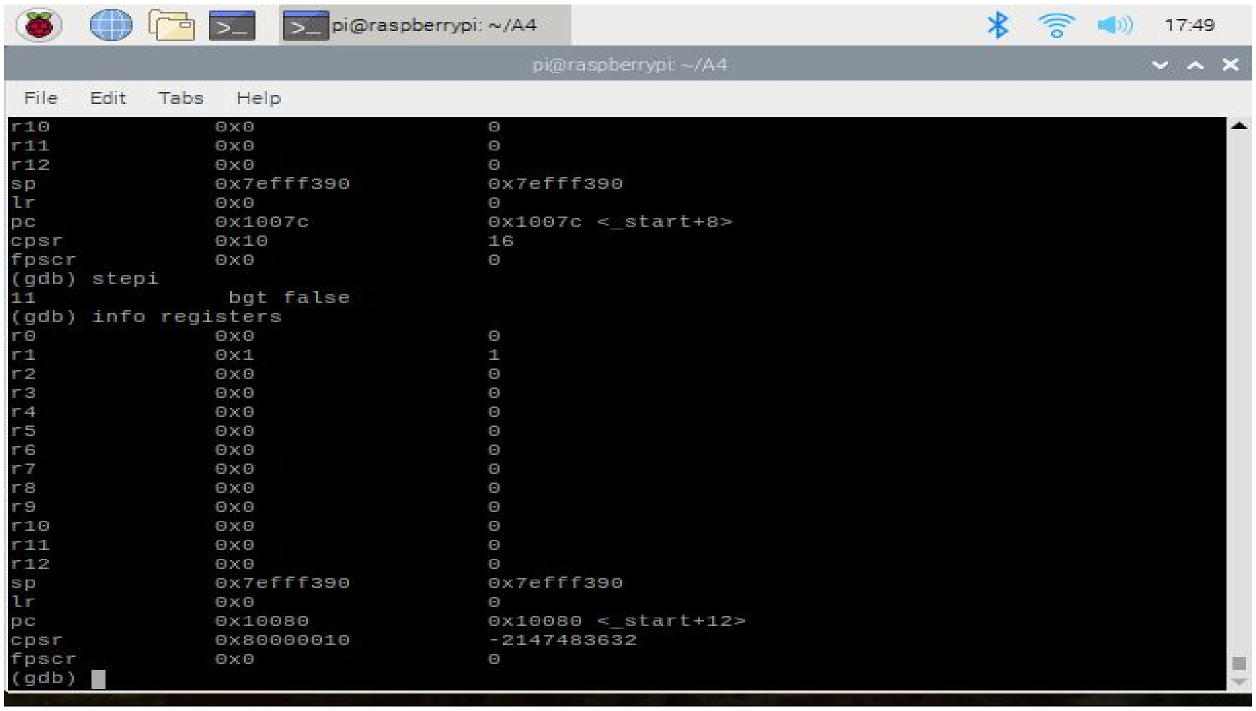
(C, 7)



```
Breakpoint 1 at 0x10078: file ControlStructure1.s, line 9.
(gdb) stepi
The program is not being run.
(gdb) run
Starting program: /home/pi/A4/ControlStructure1

Breakpoint 1, _start () at ControlStructure1.s:9
9               ldr r1, [r1]
(gdb) stepi
10              cmp r1, #3
(gdb) info registers
r0             0x0                 0
r1             0x1                 1
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff390          0x7efff390
lr             0x0                 0
pc             0x1007c             0x1007c <_start+8>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

stepi to load value **1** from **x** to **r1**, then compare value to **3**.

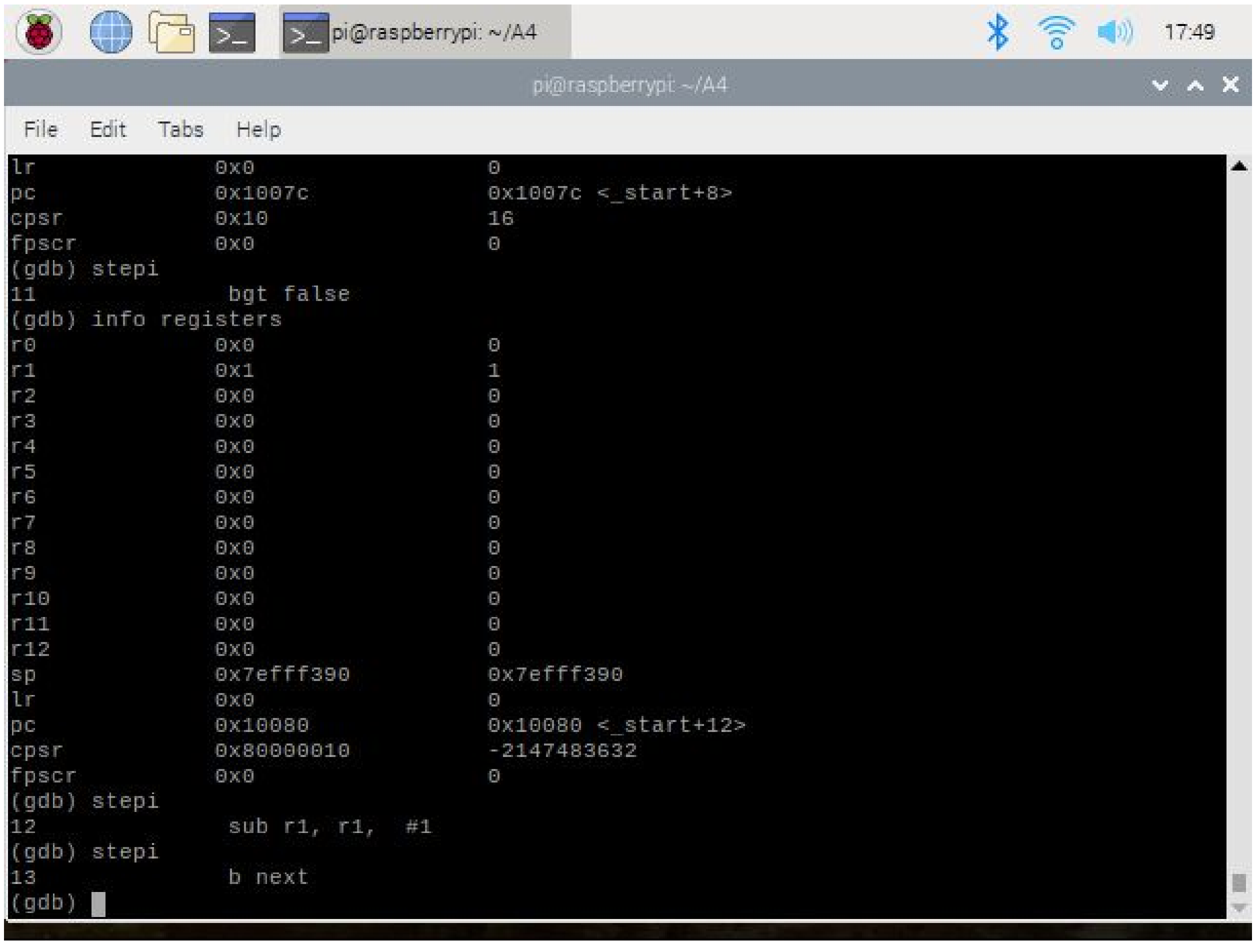
(C, 8)



```
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff390          0x7efff390
lr             0x0                 0
pc             0x1007c             0x1007c <_start+8>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
11              bgt false
(gdb) info registers
r0             0x0                 0
r1             0x1                 1
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff390          0x7efff390
lr             0x0                 0
pc             0x10080             0x10080 <_start+12>
cpsr           0x80000010          -2147483632
fpscr          0x0                 0
(gdb)
```

stepi to **bgt false**

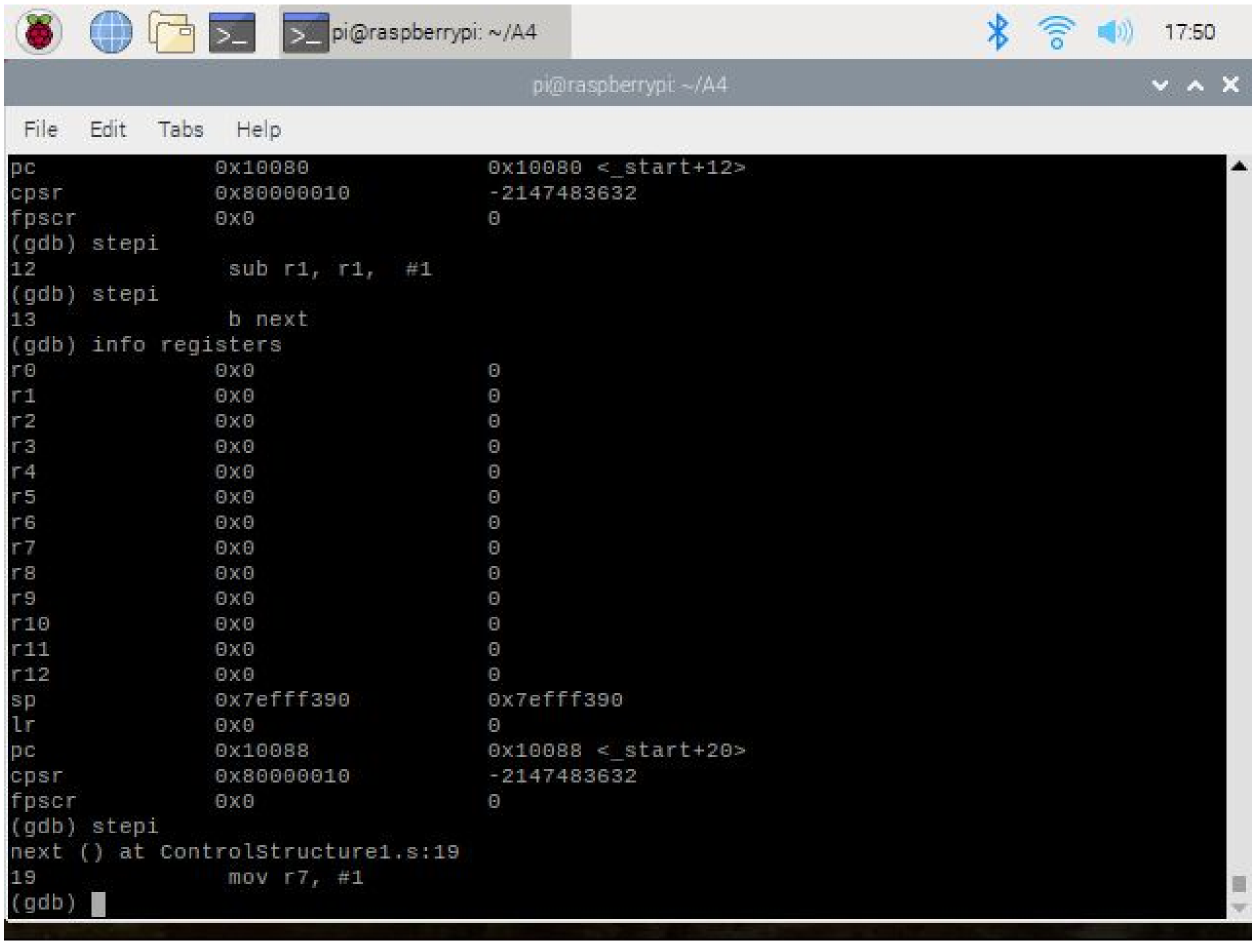
(C, 9)



```
lr             0x0                 0
pc             0x1007c             0x1007c <_start+8>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
11              bgt false
(gdb) info registers
r0             0x0                 0
r1             0x1                 1
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff390          0x7efff390
lr             0x0                 0
pc             0x10080             0x10080 <_start+12>
cpsr           0x80000010          -2147483632
fpscr          0x0                 0
(gdb) stepi
12              sub r1, r1,  #1
(gdb) stepi
13              b next
(gdb) 
```

Skipping jump to **false**, due to **r1** equating to **1**, not greater than **3**.

(C, 10)


pi@raspberrypi: ~/A4

File Edit Tabs Help

```
pc             0x10080             0x10080 <_start+12>
cpsr           0x80000010          -2147483632
fpscr          0x0                 0
(gdb) stepi
12              sub r1, r1,  #1
(gdb) stepi
13              b next
(gdb) info registers
r0             0x0                 0
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff390          0x7efff390
lr             0x0                 0
pc             0x10088             0x10088 <_start+20>
cpsr           0x80000010          -2147483632
fpscr          0x0                 0
(gdb) stepi
next () at ControlStructure1.s:19
19              mov r7, #1
(gdb)
```

r1 equates to 0 after subtraction. Program exits. cpsr register reads 0x80000010, indicating Negative flag was triggered, but not the Zero flag. The zero flag should trigger if **subs** was used instead of **sub** for **sub r1, r1, #1**