Developing Soft and Parallel Programming Skills Using Project-Based Learning

Justin Choi, Adam Henrie, Jean-Pierre Sacha, Titilayo Shonuyi, Shanza Siddiqi

Spring 2019
Group Name: Qubits

Planning and Scheduling

Team name: qubits

Name	Email	Task	Duration	Dependency	Due Date	Note
Justin Choi	jchoi34@student.gsu.edu	Report	3 hours	Task 3 and Task 4	3/28/19	Finish report a day before the due date
Adam Henrie	ahenrie1@student.gsu.edu	Video, support with parallel questions and arm programming	4 hours	Video	3/27/19	Please assist in answering questions and with arm programming as needed
JP Sacha	jsacha2@student.gsu.edu	Parallel programming and parallel questions	4 hours	None	3/27/19	Finish by Wednesday to run through the code with group
Titilayo Shonuyi	tshonuyi1@student.gsu.edu	Arm assembly programming	4 hours	None	3/27/19	Finish by Wednesday to run through code with group
Shanza Siddiqi	Ssiddiqi2@student.gsu.edu	Create To do/In Progress/Done columns and cards	3 hours	None	3/24/19	Create new cards on github and provide screenshots of the tasks

Parallel Programming Skills

Foundation

Race condition:

1. **What is race condition?**
   A race condition happens when two processes or threads are trying to update data from within a critical section/area of the .data section of a piece of code.

2. **Why race condition is difficult to reproduce and debug?**
   A race condition is difficult to reproduce and debug because there is no set amount of time that some calculations are required to complete their work. Because of this, in a multiprogramming scenario we can have many processes or threads altering communal pieces of code in the .data section at seemingly random points in time. This is difficult to track and debug.

3. **How can it be fixed? Provide an example from your Project_A3**
   This can be fixed by allowing each thread the ability to have their own memory location that does not conflict with other memory locations. Or we can implement Mutex or Semaphore locks.

4. **Summarize the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" (two pages) in your own words (one paragraph, no more than 150 words).**
   Parallel Programming patterns allow for the programmer to use pre-bottled and ready to use templates, in essence, to write their code. Following the common patterns seen in parallel computing allows us the ability to better code with parallelism in mind. We can use strategies in the pure algorithms that we write or we can use strategies that allow for the composition of the overall program to be in line with the patterns we observe. We can implement strategies that allow for concurrent data access as well as being able to use multiple processing cores to implement concurrency. We use two types of patternlets in parallelism. We use Process Thread control as a built in "methods". We can use coordination patterns as well which make use of a Message Passing Interface or OpenMP for multithreaded shared memory applications. Message passing allows for two processes to coordinate while we have one more coordination pattern which is mutual exclusion.

5. **In the section "Categorizing Patterns" in the "Introduction to Parallel Computing_3.pdf" <u>compare</u> the following.**
   O Collective synchronization (barrier) with Collective communication (reduction)

A barrier forms a hold point that allows processes and threads to catch up before executing the code after the barrier, hence synchronizing the processes.
A reduction shares the operating variable between threads, hence communicating the result of the process.
A reduction is also a form of synchronization that allows for multiple threads to provide a result for the same variable and can use a form of barrier. Both models manage processes to avoid race conditions and ensure that processes with dependency are performed sequentially.

O Master-worker with fork join

Fork-join runs parallel processes on multiple threads and joins the result
Master-worker divides a main process into smaller chunks delegated to worker processes.
Both models split a process into smaller sub-processes and "join" the results. It is possible that master-worker utilizes a form of fork-join to operate.

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

6. **Where can we find parallelism in programming?**
   We can find parallelism in the code that we are attempting to write in the Program, Data, and Resource viewpoint. Essentially, we can see that we can work towards programs that have non-conflicting statements that can execute concurrently. We can also find parallelism in repetitive tasks. Finally we can find it in how we implement parallelizable code on the hardware.

7. **What is dependency and what are its types (provide one example for each)?**
   Dependency happens when the input of one operation is the output of another or vice versa. There are three types of dependencies. True dependence, output dependence, and anti-dependence. True (flow) dependence happens when a second statement is dependent on the value of a first statement. Ex. D = 3: a = D. Output dependence happens when the output of the first statement determines the value of a second statement. Ex. A = g(y): a = b. Finally anti-dependence happens when a statement that has already tried to be executed can't fully execute because its value is dependent on the value of the next statement. D = c: c = 42.

8. **When a statement is dependent and when it is independent (Provide two examples)?**
   A statement is **dependent** when it is dependent on the value of another statement. Both of these statements cannot run independent of each other. A = f'(x) = 8x^2 + 2x + 2: b = a. Here we can see that be cannot run till is computed.

Independence occurs when two statement can be run concurrently with no input or binding behavior of another process. B = a; d = c. There is no relation between these two statements. They are independent and can run concurrently.

9. **When can two statements be executed in parallel?**
   Two statement can be executed in parallel if and only if both statement are independent of each other and have no dependencies.

10. **How can dependency be removed?**
    Dependency can be removed by adding or removing statements that cause dependencies between to statements in a piece of code.

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

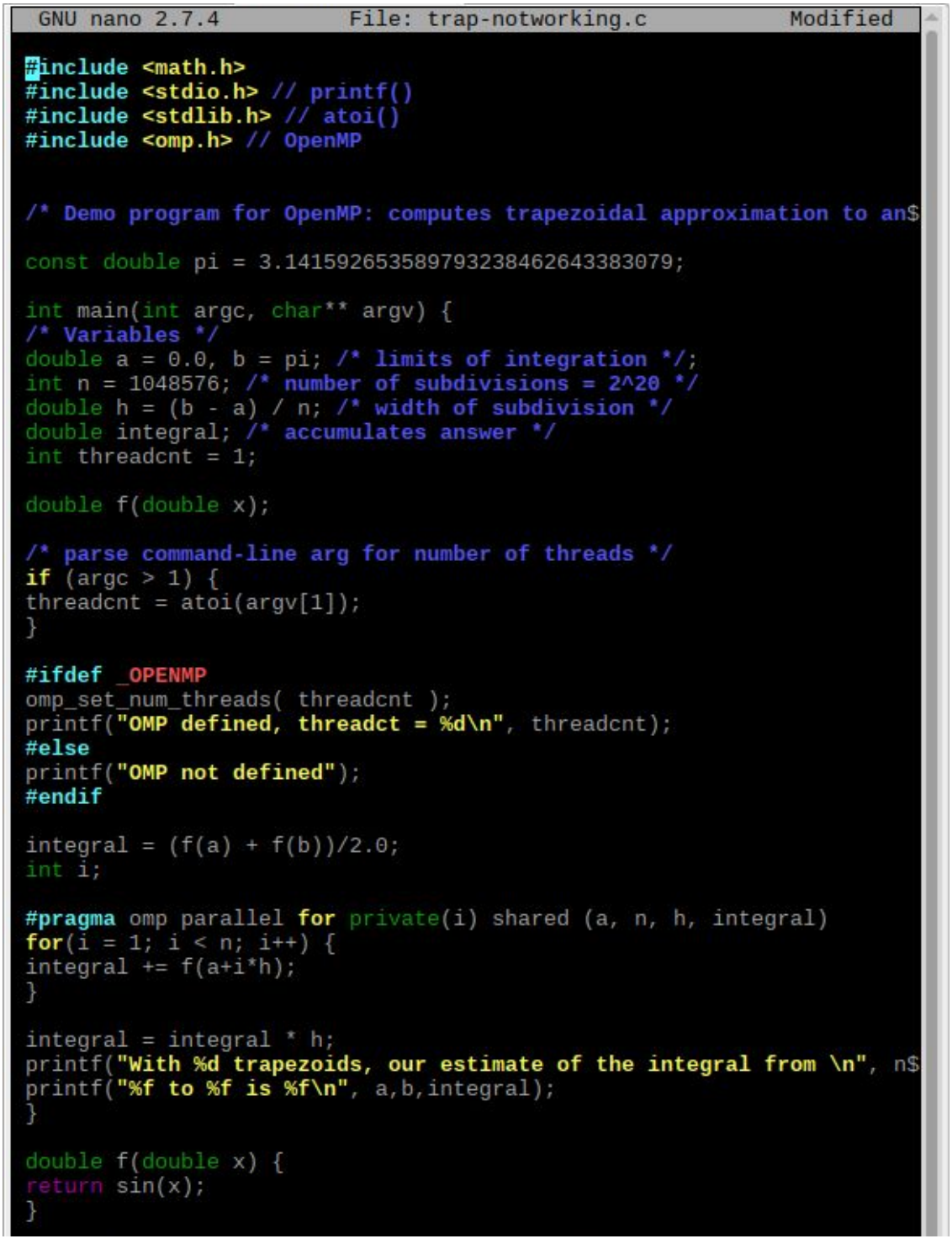
```
for (i=0; i<100; i++)
  S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
  S1: a[i] = i;
  S2: b[i] = 2*i;
}
```

Computing dependence occurs when one compares the In and out sets for statements. For both the first and the second loops we can see that neither of them has any dependencies between statements. What we do see is dependency on the main loop counter. However, statements that are in the second loop can be split and given in any amount deemed appropriate by the OS scheduler. Thus we can have 25 iteration of the incrementing value of “i” go to core 1, 25 to core 2, 25 to 3 and so on. Or any other combination thereof.

Parallel Programming Basics - Barrier and Master-worker

1. Code for trap-notworking.c.



```
 GNU nano 2.7.4          File: trap-notworking.c          Modified

#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


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

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
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);
}
```

2. Code for trap-working.c.

```
 GNU nano 2.7.4                File: trap-working.c                Modified

#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 = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a) / n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of threads */
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);
}
```

3. Here is the output for both codes.

```
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.358073
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
```

Trap-notworking produces 1.35… while trap-working produces 2, the correct result. As described in the assignment this is due to the accumulator variable 'integral'. Trap-working adds a reduction clause to the variable which solves the race condition.

4. Code for barrier.c. The barrier pragma is not yet un-commented.

```
 GNU nano 2.7.4                File: barrier.c                Modified

#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;
}
```

5. Output for barrier.c.

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

pi@raspberrypi:~ $ gcc barrier.c -o barrier -fopenmp
pi@raspberrypi:~ $ ./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 0 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.

pi@raspberrypi:~ $ ./barrier

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

Without the barrier pragma the threads perform both iterations. With the pragma each thread processes one iteration and waits for the rest of the threads before moving past the barrier for the second iteration.

6. Code for master Worker.c.

```
 GNU nano 2.7.4                File: masterWorker.c                Modified

#include <stdio.h> //printf()
#include <stdlib.h> //atoi()
#include <omp.h> //OpenMP
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 ) {// thread with ID 0 is master
     printf("Greetings from the master, # %d of %d threads\n",id,numThreads);
   }    else {// threads with IDs > 0 are workers
     printf("Greetings from a worker, # %d of %d threads\n",id,numThreads);
       }
 }
 printf("\n");
 return 0;
}

pi@raspberrypi:~ $ ./masterWorker

Greetings from the master, # 0 of 1 threads

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

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

pi@raspberrypi:~ $ ./masterWorker

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

7. Output for Master worker.c. Without the parallel pragma only the 'master' thread, the thread with ID=0, is displayed since only one thread is utilized. With the parallel pragma each thread is displayed accordingly. Interestingly, the order of execution pattern appears to be 0-2-1-3.

8. Trace - Non-working:

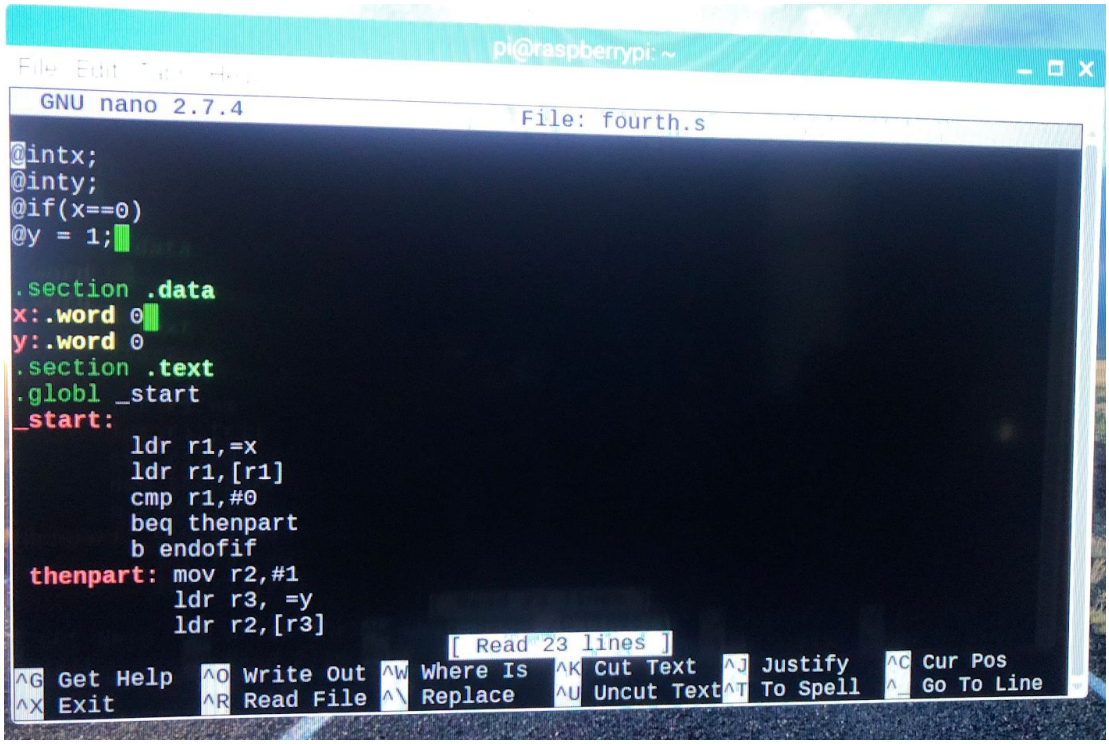
Step	ID	numThread
1	0	0

Working:

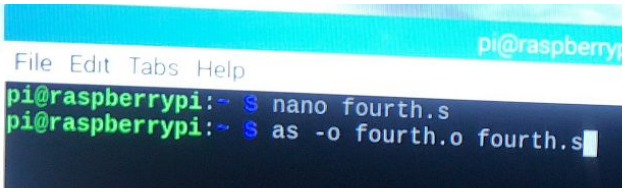
1	3	3
2	0	0
3	2	2
4	1	1

ARM Assembly Programming

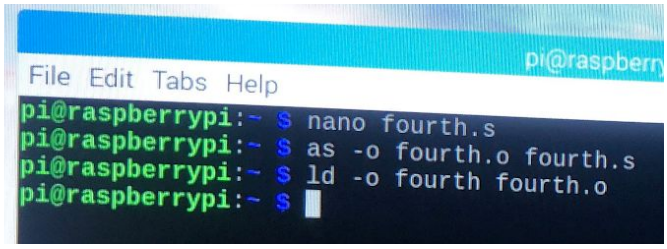
Typing the fourth.s code



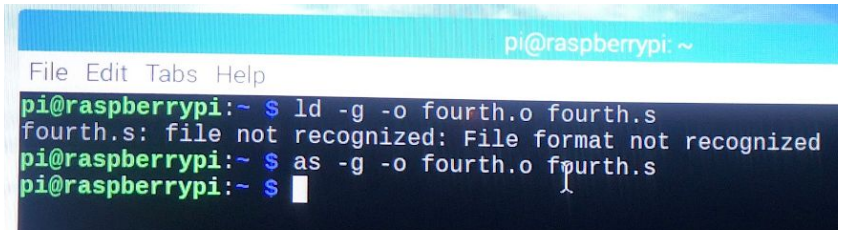
Assembled fourth.s file



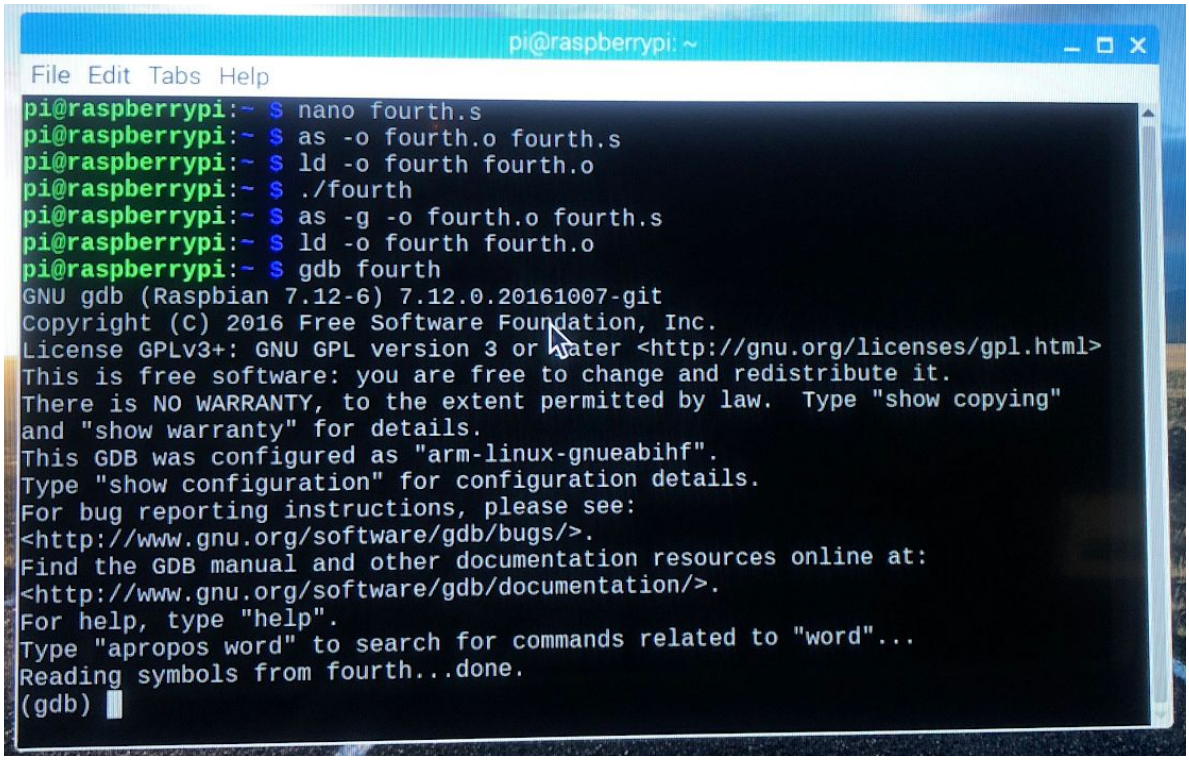
Link file



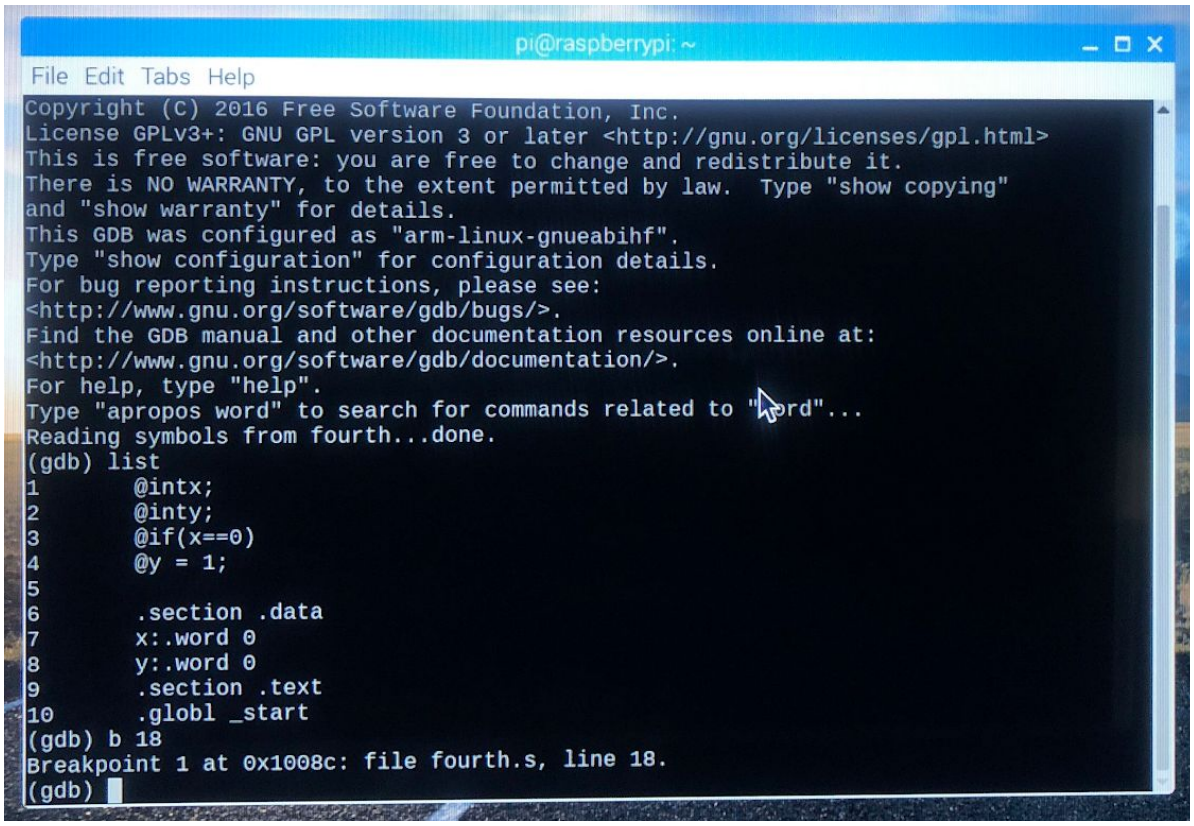
Preserving fourth.s file for debugging



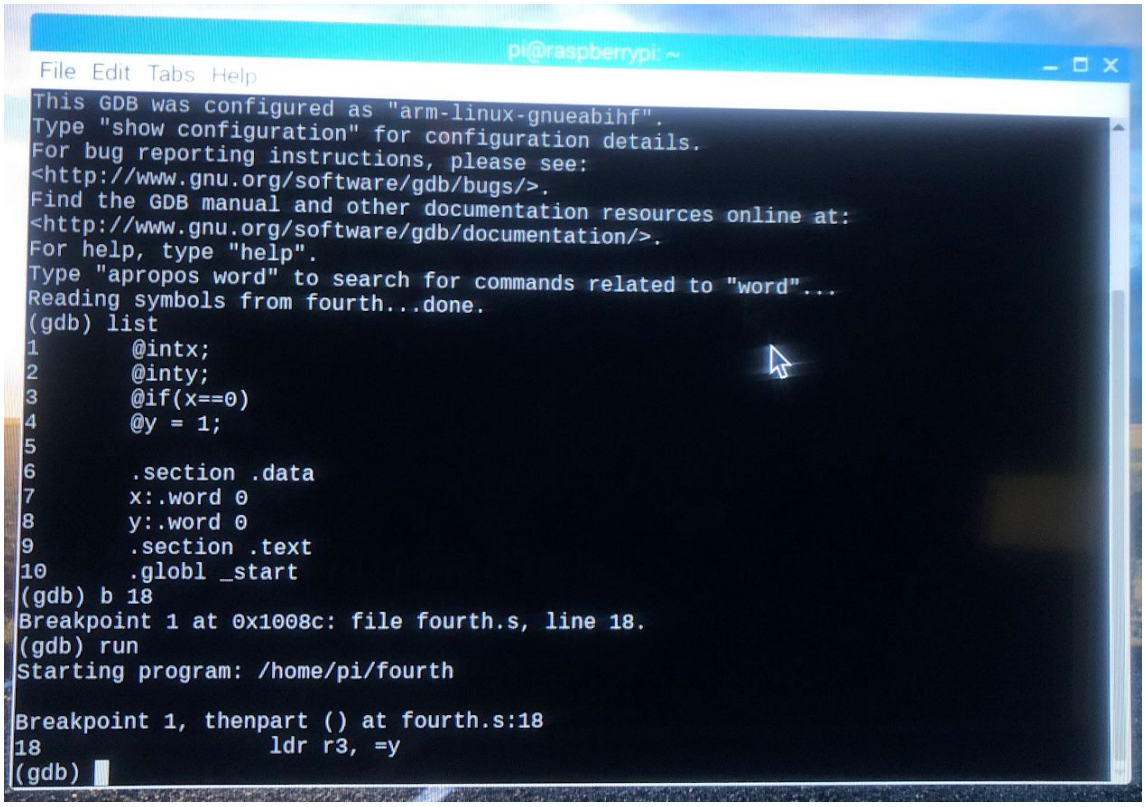
Displayed pi license and warranty with (gdb)



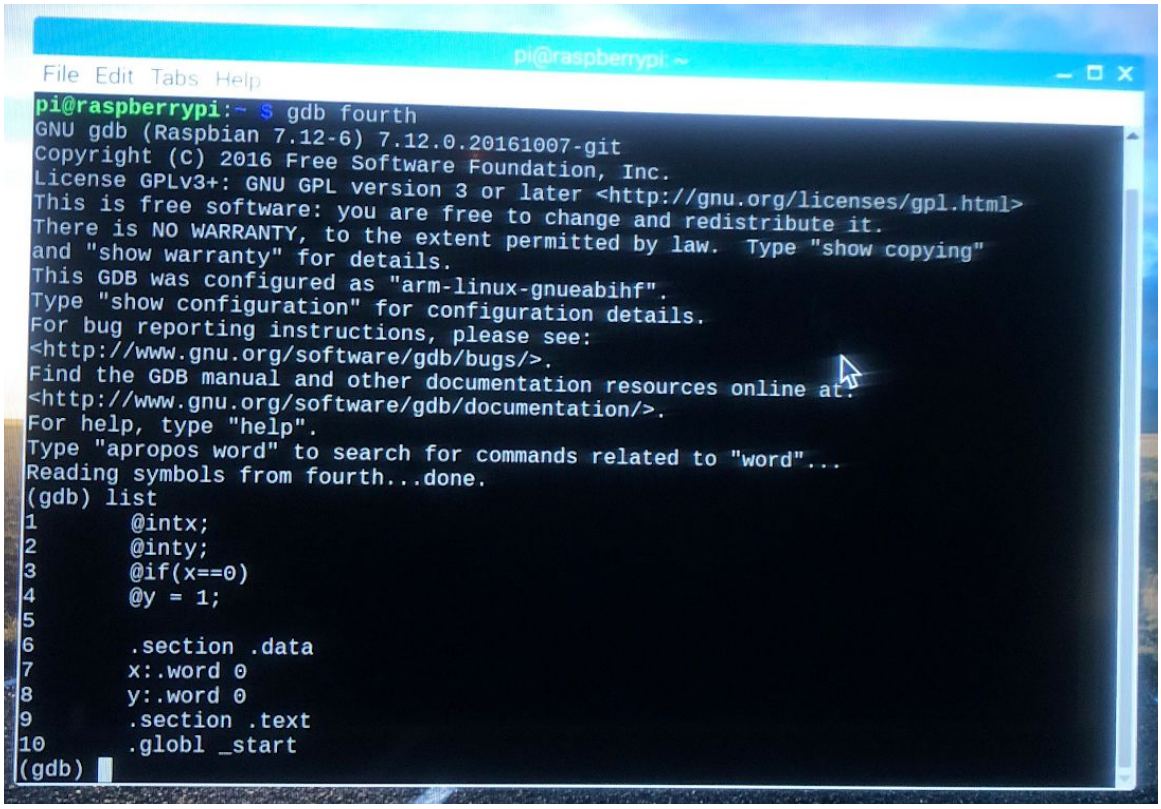
Setting break point in line 18



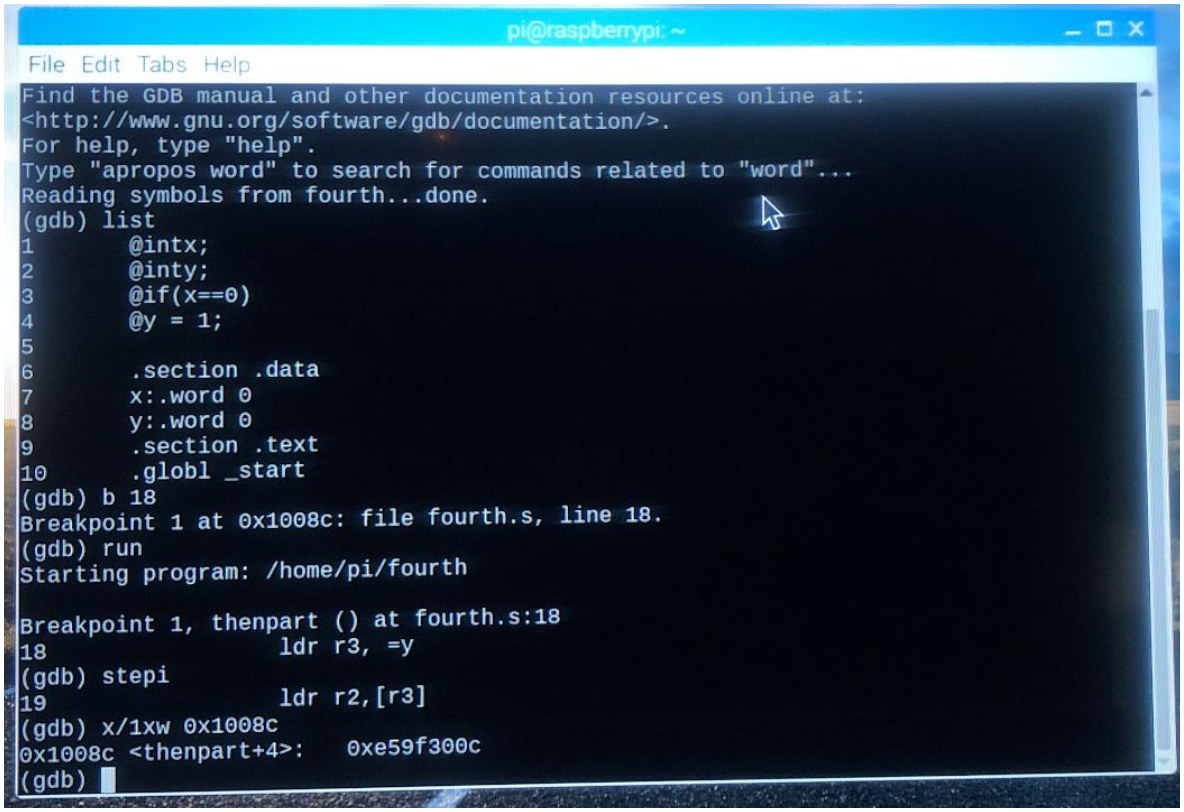
Run the program at break point



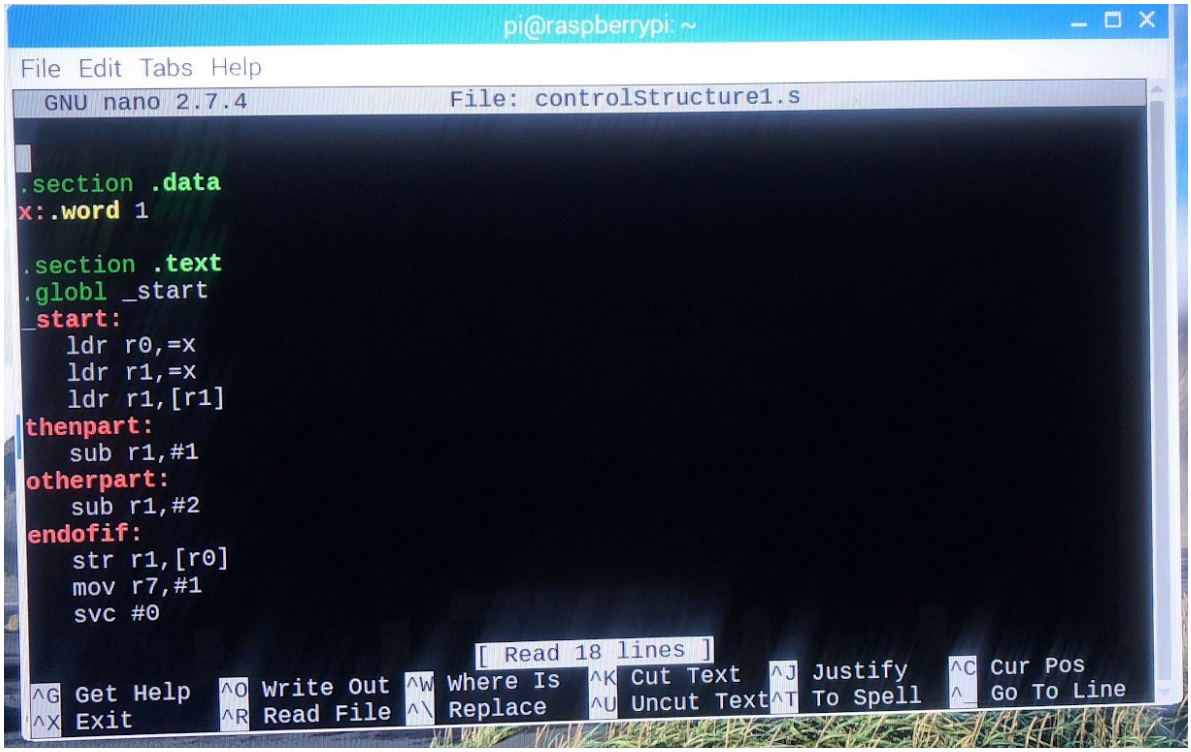
List lines of code



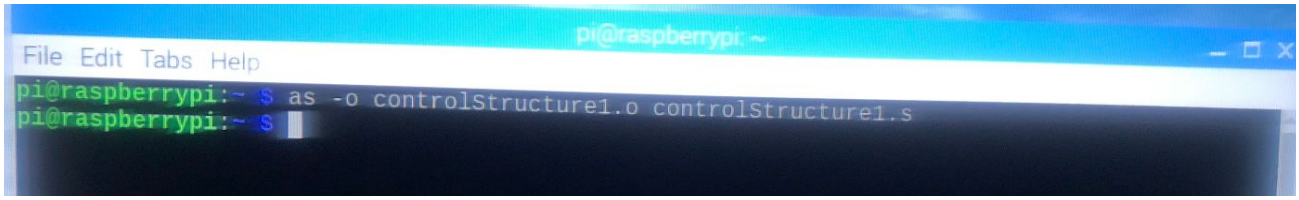
Displayed the value of x/1xw 0x1008c

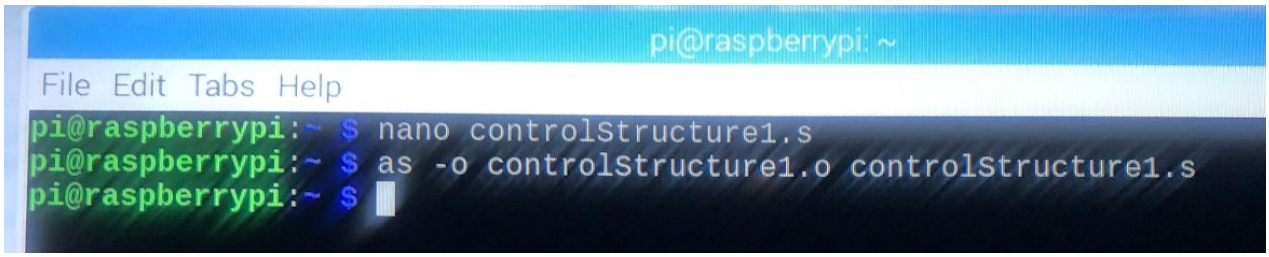


ControlStructure1.s Typing code

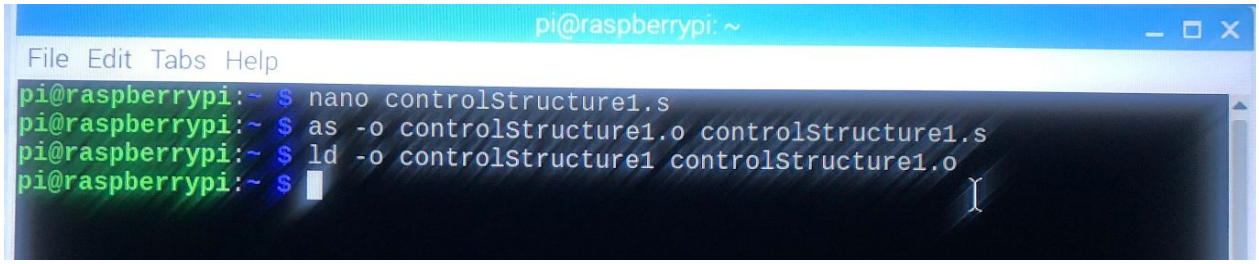


Assemble file

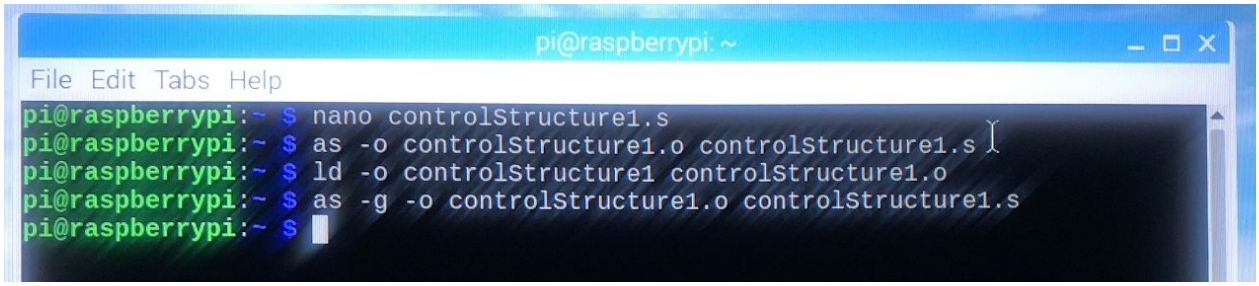




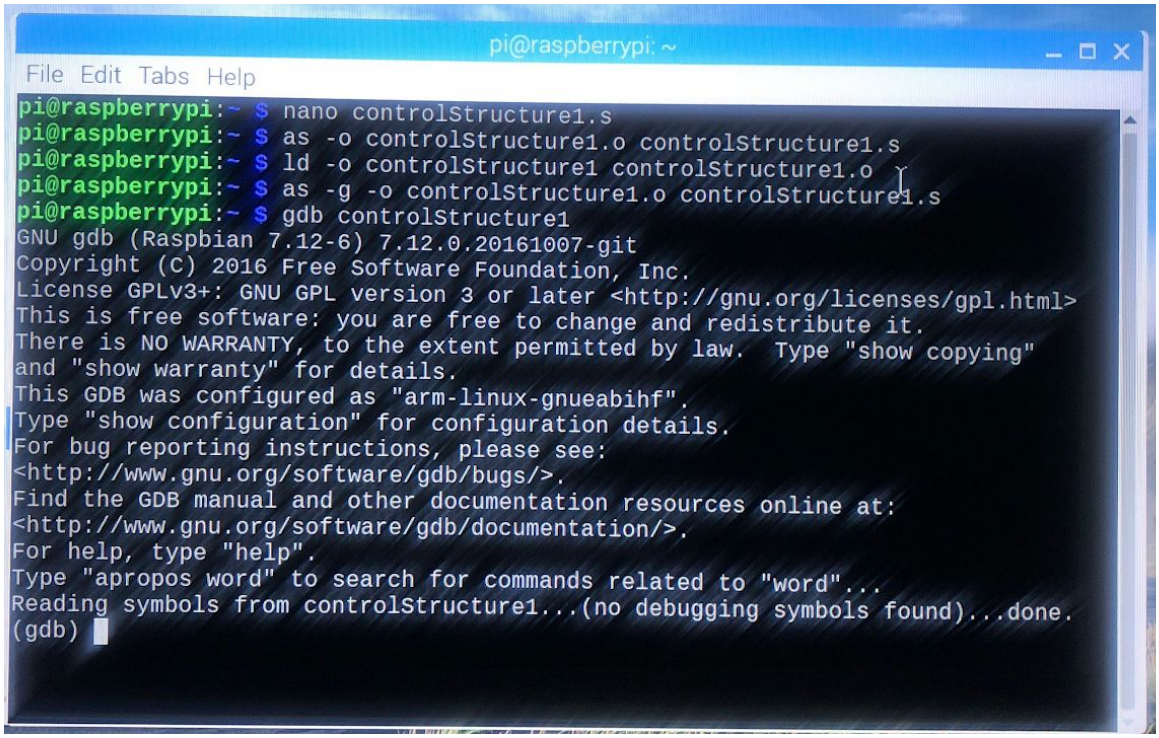
Link file



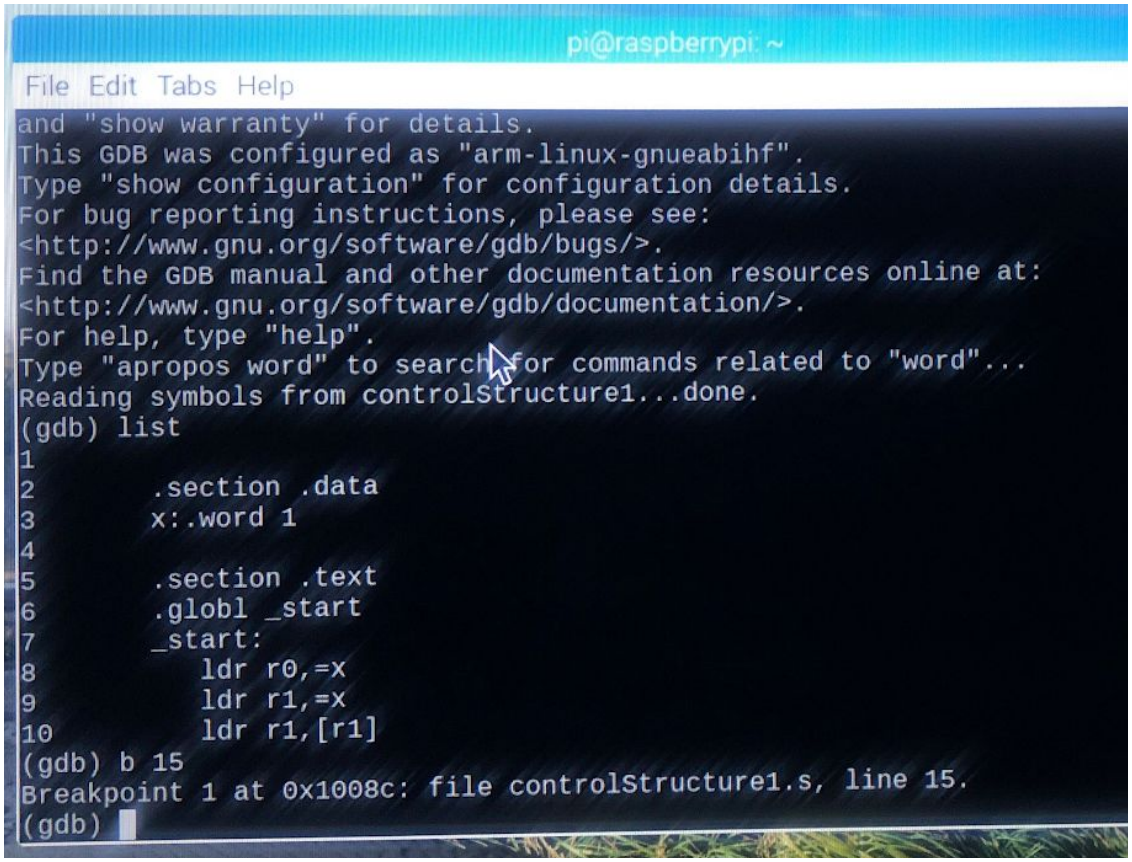
Preserving controlStructure1.s file for debugging



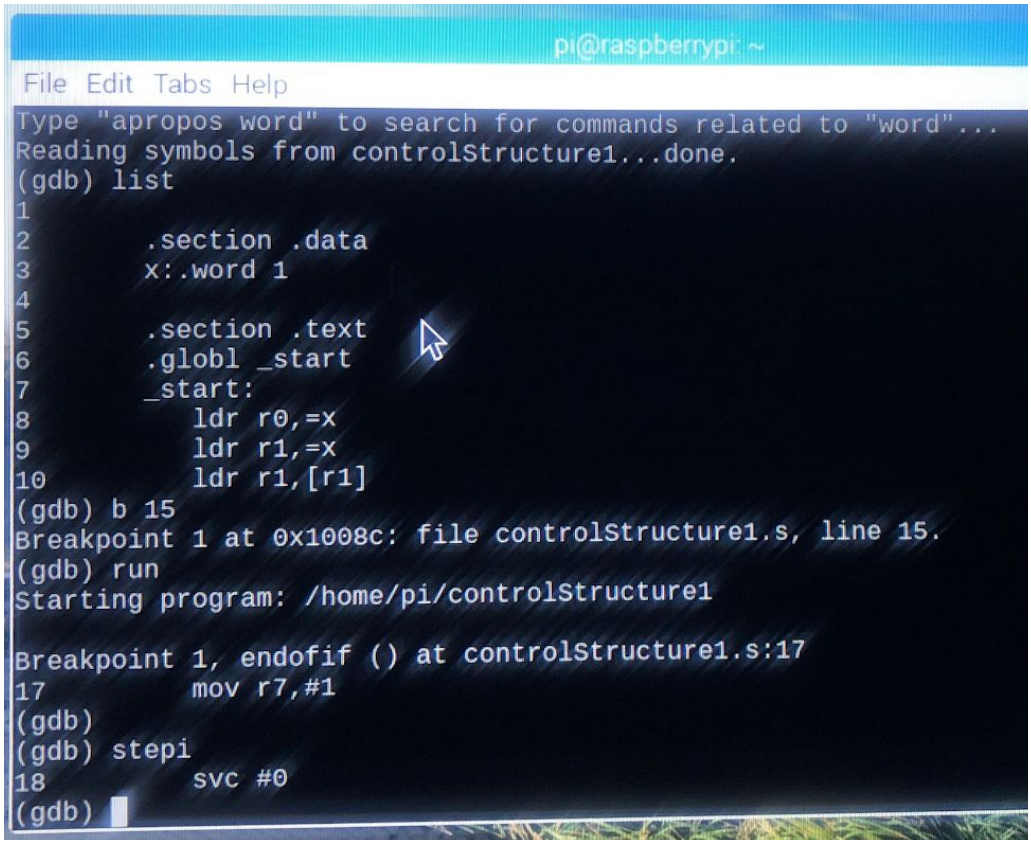
Displayed pi license and warranty with (gdb)



List and Set break point in line 15

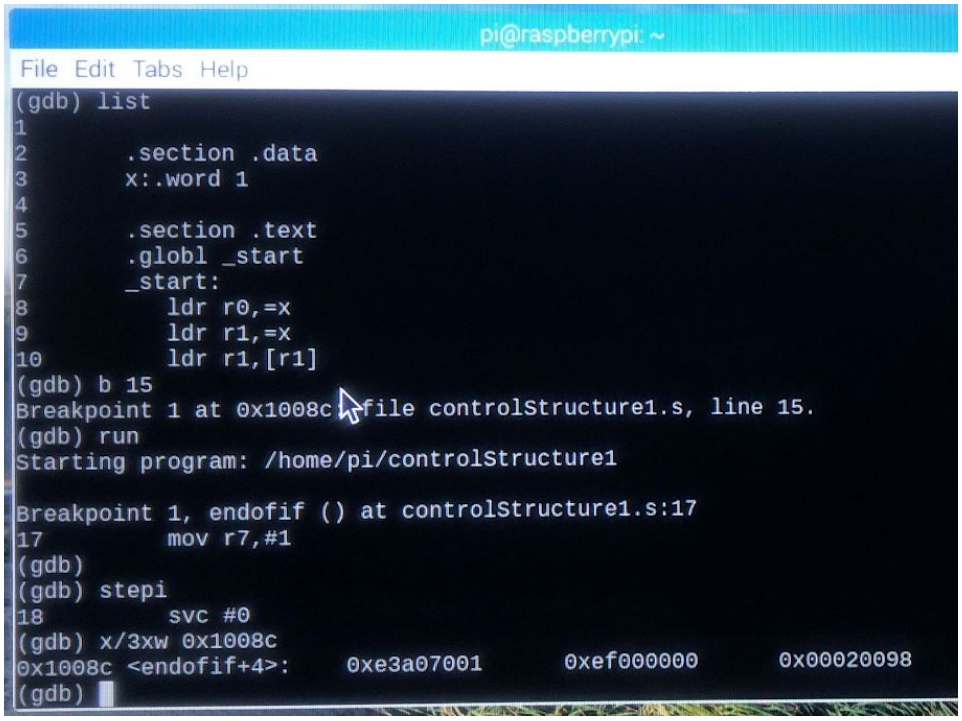


gdb stepi



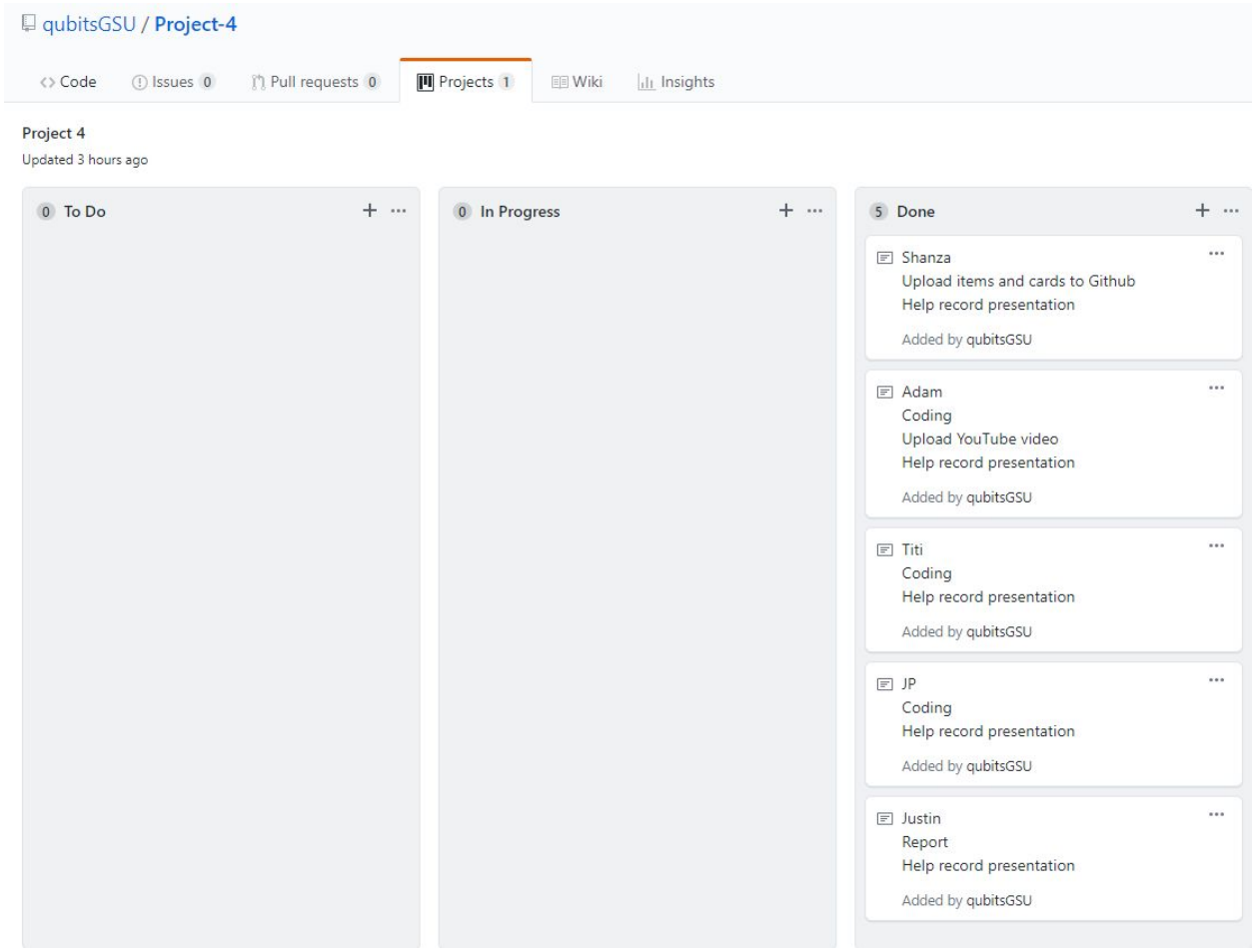
The break point set in line 15 is <endofif + 4>

gdb x/3xw 0x1008c



Appendix:

GitHub



Important Links:

- **Slack:** computerorganizespr19.slack.com
- **GitHub:** https://github.com/qubitsGSU
- **YouTube:** https://youtu.be/bysjFzIN5dI