Developing Soft and Parallel Programming Skills Using Project-Based Learning

semester (Spring-2019)

Group name: Kenny and the Rest

Members name: Nish Patel, Abdiaziz Dakane, Soojie Chin, Brett Krokoff and Kehinde Adedara

Planning and scheduling

Name	Email	Task	Duration (Hours)	Dependency	Due Date	Note
Abdiaziz Dakane (Group coordinator)	Adakane1 @student.gsu.edu	3a: Parallel Programming Skill (Foundation)	2-4 Hours	Answering questions regarding foundation in parallel programming.	11/14/19	Please have task 3a done ASAP, so Nish has enough time to put the report together.
Brett Krokoff	Bkrokoff1 @student.gsu.edu	Task 6: Presentation video	2 – 3 Hours	Making sure everyone is there for the video as well as editing and uploading.	11/14/19	Please begin uploading video the night before or morning of 11/15 in case any technical difficulties arise.
Soojie Chin	Schin6 @student.gsu.edu	Team coordinator: organize meeting times and due dates	2-3Hours	(Group Co-Ordinator) Planning and scheduling.	11/14/19	In charge of Task 1 and 2. Assist team members if task cannot

						be completed in reasonable time.
Kehinde Adedara	Kadedara1 @student.gsu.edu	Task 3b & 4: Parallel Programmin g Basics & ARM Assembly Programmin g	2-3 hours	Carrying out experiment on raspberry pi	11/14/19	Please have tasks done ASAP, so Nish has enough time to complete the report.
Nish Patel	Npatel1161 @student.gsu.edu	Task 5: Report	2-3 hours	Reporting on observation and experimentations made on soft and Parallel programming using raspberry pi.	11/14/19	Please have report ready by 9pm on 11/15 if cannot be completed by 11/14.

Parallel Programming Skills

3A:

Race Condition:

- What is race condition? It is basically when a situation a software output is dependent on the timing of another and without that dependency or intended sequence it may result in an error.
- Why race condition is difficult to reproduce and debug? It is because they have a reputation of having different result on different runs which is called "nondeterministic" and as it says on the pdf, they may depend on another runs timing which maybe interrupted in the debug.
- How can they be fixed? It can be fixed by giving each keyword it's own word so each code can be executed by one thread. An example would be the reduction.c where we had to modify the commented line so it becomes private keyword for each thread and it can execute it.

Summaries Parallel Programming Patterns:

Parallel Programming is a pattern that is documented throughout your time and as new people see your code and how well it was implemented, they can make changed to it easily and as new programmers can learn it. It also talks about how there are two ways developers use this pattern by using Strategies and Concurrent Execution Mechanisms. Strategies is when you sacrifice on what tasks you can do and whether those tasks can be done parallel with other ones so it can be done quickly and to do this you have to use the right implementation strategy. As for Concurrent Execution Mechanisms, they fall into two parts first one process thread control patterns which is about how the process of parallel execution are controlled at a run time and second part being Coordinating patterns which set up how multiple concurrently running tasks on processing units coordinate to finish the parallel execution.

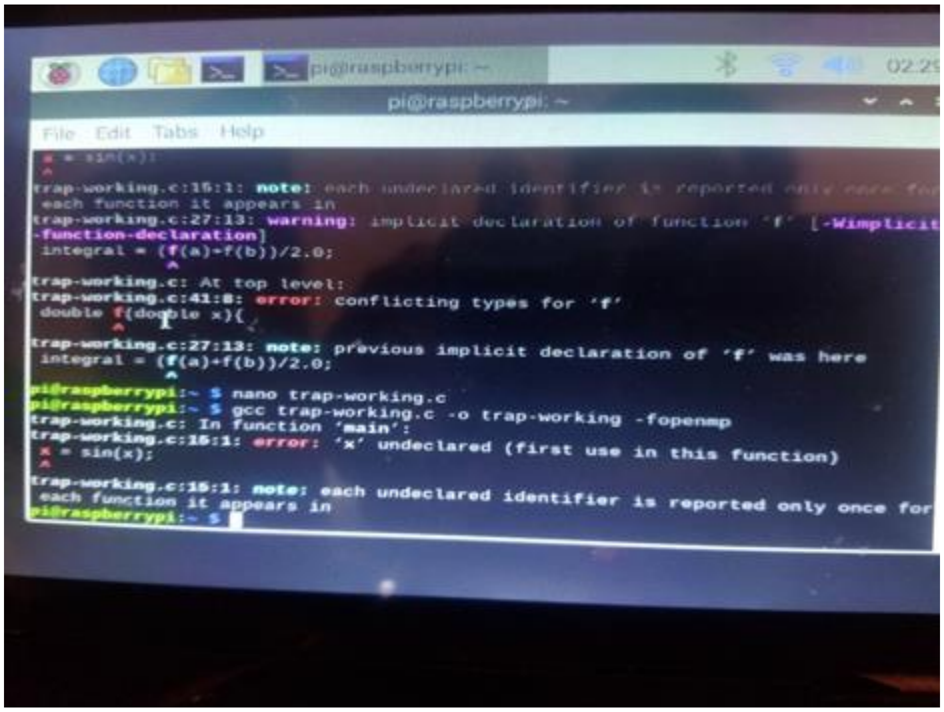
Categorizing Patterns:

- Collective Synchronization with a barrier are mostly used in all parallel programming and they usually do part of a code before executing it and that part could be used for another part of the code and Collective communication with reduction is
- Fork-join is used to execute all parallel processes and lightweight threads while master worker is used as a main processor that is divided into small parts and each with a small worker process and in the master worker while one code is being executed the rest of threads are executing a different block of code.

Dependency:

- Where can we find parallelism in programming? We can find it in the program view. What programs can be executed at the same time or between program statements.
- What is dependency and what are its types? Dependency is when one thing depends on another thing to be completed and the two types are independent and dependent.
- When is a statement dependent and when is it independent? They are dependent when two statements outcome can be affected by their order. As for independent their outcome is not affected since two statement are equivalent to each other no matter the order.
- When can two statements executed in parallel? Only when two statements are not depending on each other.
- How can dependency be removed? You can remove them by making some changes to the program such as rearranging statements and eliminating statement.
- How do we compute the dependency for the following two loops and what type of dependency? You can compute the dependency of the following two for loops by comparing the INS and OUTS for both of them also you can check what both of them share and in this loop they both share a[i]. For the first loop the dependency is true dependence while the second one has a dependency of output dependency.

Task 3b:



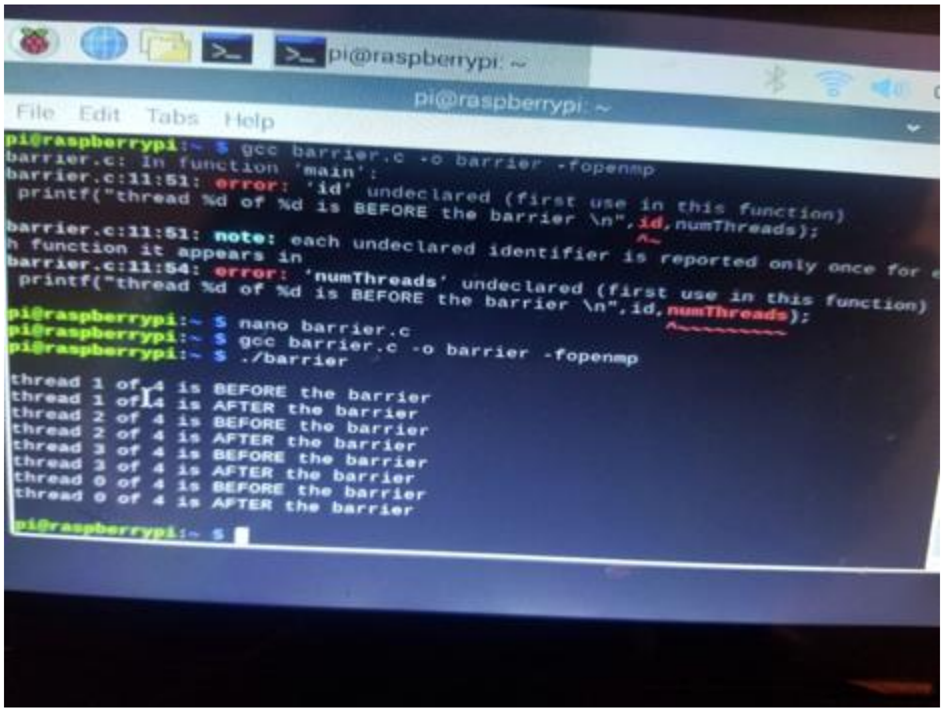
```
x = sin(x);
^
trap-working.c:15:1: note: each undeclared identifier is reported only once for
 each function it appears in
trap-working.c:27:13: warning: implicit declaration of function 'f' [-Wimplicit
-function-declaration]
 integral = (f(a)+f(b))/2.0;
             ^
trap-working.c: At top level:
trap-working.c:41:8: error: conflicting types for 'f'
 double f(double x){
        ^
trap-working.c:27:13: note: previous implicit declaration of 'f' was here
 integral = (f(a)+f(b))/2.0;
             ^
pi@raspberrypi:~ $ nano trap-working.c
pi@raspberrypi:~ $ gcc trap-working.c -o trap-working -fopenmp
trap-working.c: In function 'main':
trap-working.c:15:1: error: 'x' undeclared (first use in this function)
 x = sin(x);
 ^
trap-working.c:15:1: note: each undeclared identifier is reported only once for
 each function it appears in
pi@raspberrypi:~ $
```

Unfortunately, we couldn't get the program to work to satisfaction but because the Arm is not recognizing the math library so we had to declare sin of x originally in the main function. However, this complicated other issues in our programs. Objective, however, was to see how new statement added would thread out and execute.

- On this task, we will probably come together as a team to figure what is wrong with either the function or the library. Just unfortunate we cant figure it out sooner.

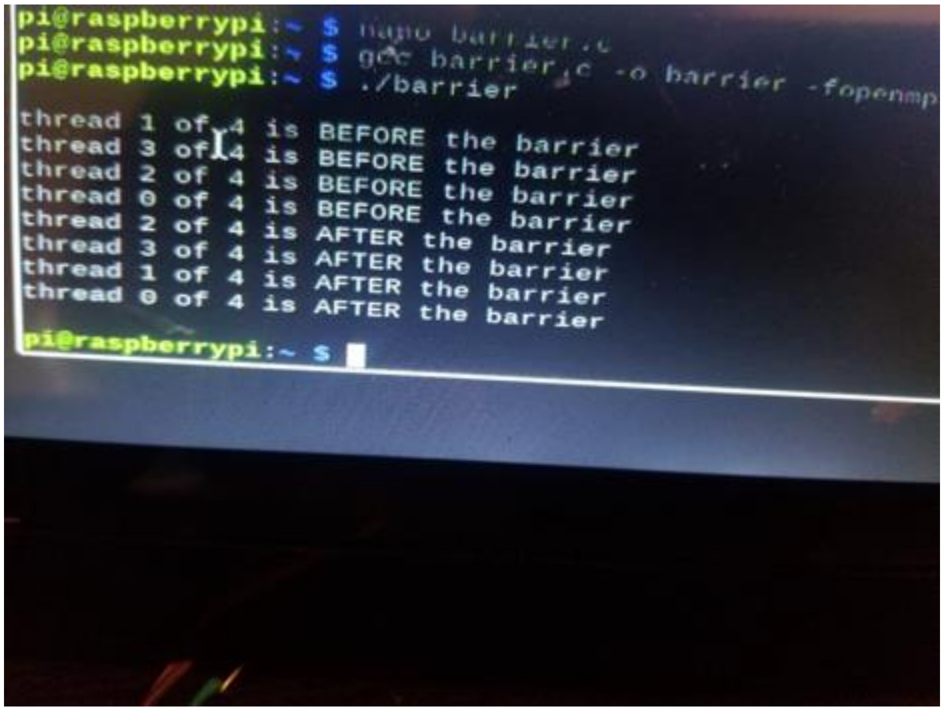
NANO BARRIER.C

Here is the output before the //pragma omp barrier



```
pi@raspberrypi:~ $ gcc barrier.c -o barrier -fopenmp
barrier.c: In function 'main':
barrier.c:11:51: error: 'id' undeclared (first use in this function)
 printf("thread %d of %d is BEFORE the barrier \n",id,numThreads);
                                                   ^~
barrier.c:11:51: note: each undeclared identifier is reported only once for
h function it appears in
barrier.c:11:54: error: 'numThreads' undeclared (first use in this function)
 printf("thread %d of %d is BEFORE the barrier \n",id,numThreads);
                                                      ^~~~~~~~~~
pi@raspberrypi:~ $ nano barrier.c
pi@raspberrypi:~ $ gcc barrier.c -o barrier -fopenmp
pi@raspberrypi:~ $ ./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
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
pi@raspberrypi:~ $
```

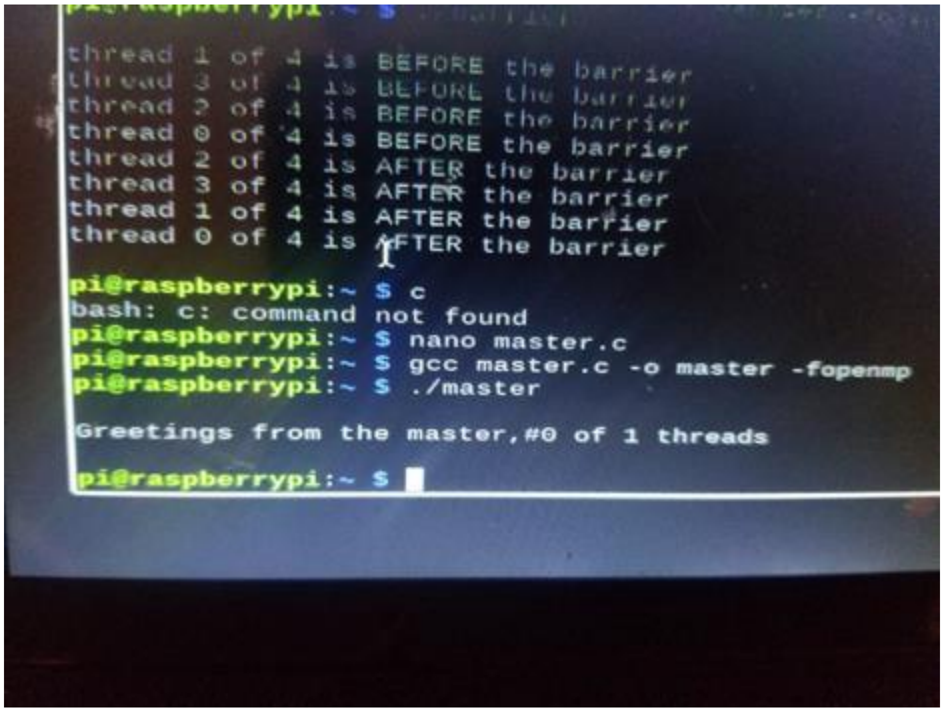
Here is the output after the new statement is input.



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

thread 1 of 4 is BEFORE the barrier
thread 3 of 4 is BEFORE the barrier
thread 2 of 4 is BEFORE the barrier
thread 0 of 4 is BEFORE the barrier
thread 2 of 4 is AFTER the barrier
thread 3 of 4 is AFTER the barrier
thread 1 of 4 is AFTER the barrier
thread 0 of 4 is AFTER the barrier

pi@raspberrypi:~ $
```

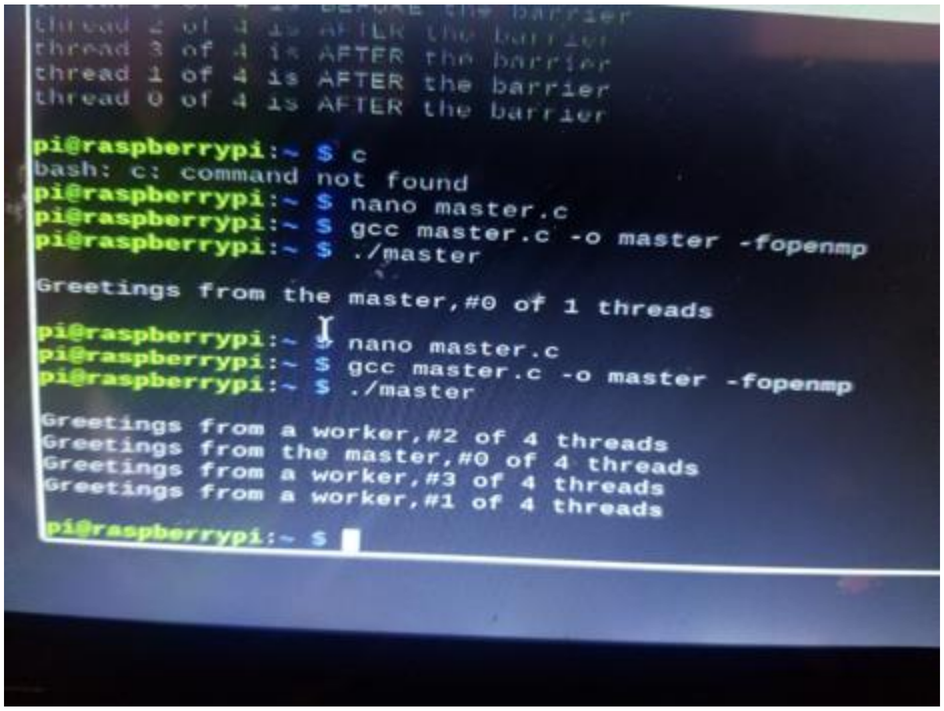
Task 3:

Master without the pragma omp parallel line



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

pi@raspberrypi:~ $ c
bash: c: command not found
pi@raspberrypi:~ $ nano master.c
pi@raspberrypi:~ $ gcc master.c -o master -fopenmp
pi@raspberrypi:~ $ ./master

Greetings from the master,#0 of 1 threads

pi@raspberrypi:~ $
```

With the pragma omp parallel



```
thread 2 of 4 is AFTER the barrier
thread 3 of 4 is AFTER the barrier
thread 1 of 4 is AFTER the barrier
thread 0 of 4 is AFTER the barrier

pi@raspberrypi:~ $ c
bash: c: command not found
pi@raspberrypi:~ $ nano master.c
pi@raspberrypi:~ $ gcc master.c -o master -fopenmp
pi@raspberrypi:~ $ ./master

Greetings from the master,#0 of 1 threads

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

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

pi@raspberrypi:~ $
```

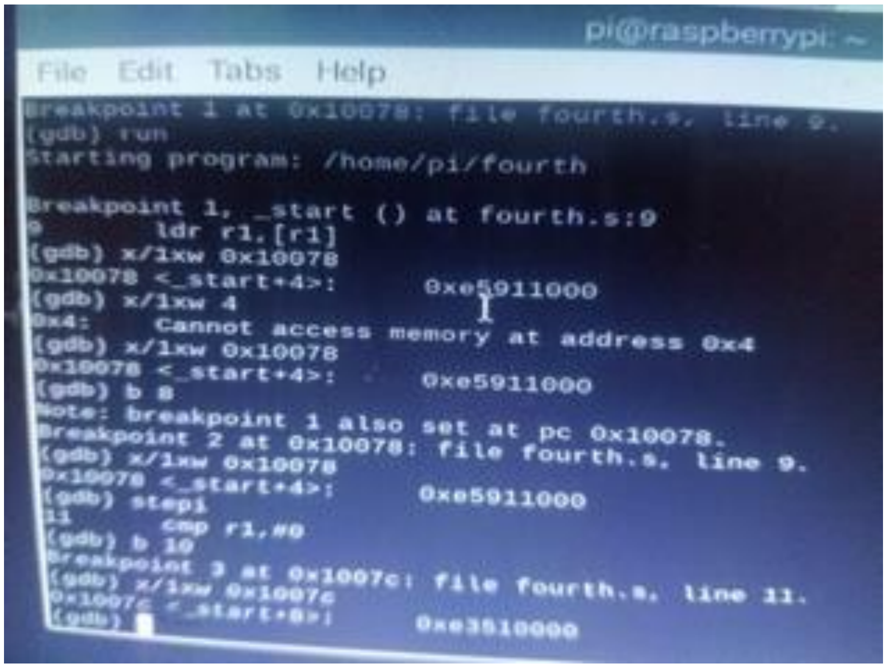
ARM Assembly Programming

Conclusion:

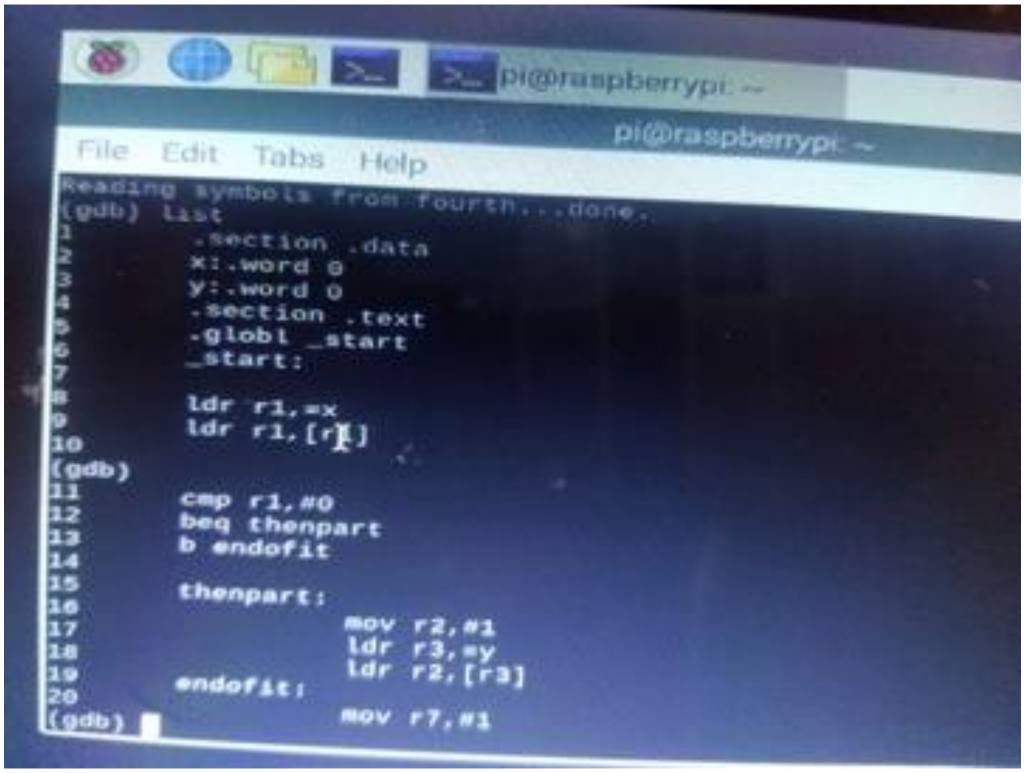
Based on the observation that we make; we can understand how the ARM board executes complex programming task. The barrier pattern ensures that all threads are read in a parallel format which is highly efficiency as the time complexity is better. The C programs are not the focus in this section, but rather, how to synchronize a program to execute using the barrier pattern. That, way the raspberry pi that split its task into the four cores and execute parallel.

Part 1:

By detecting the address values when we set the break, we can check the where in the memory the value is stored and how many offsets there are from the first line of code.

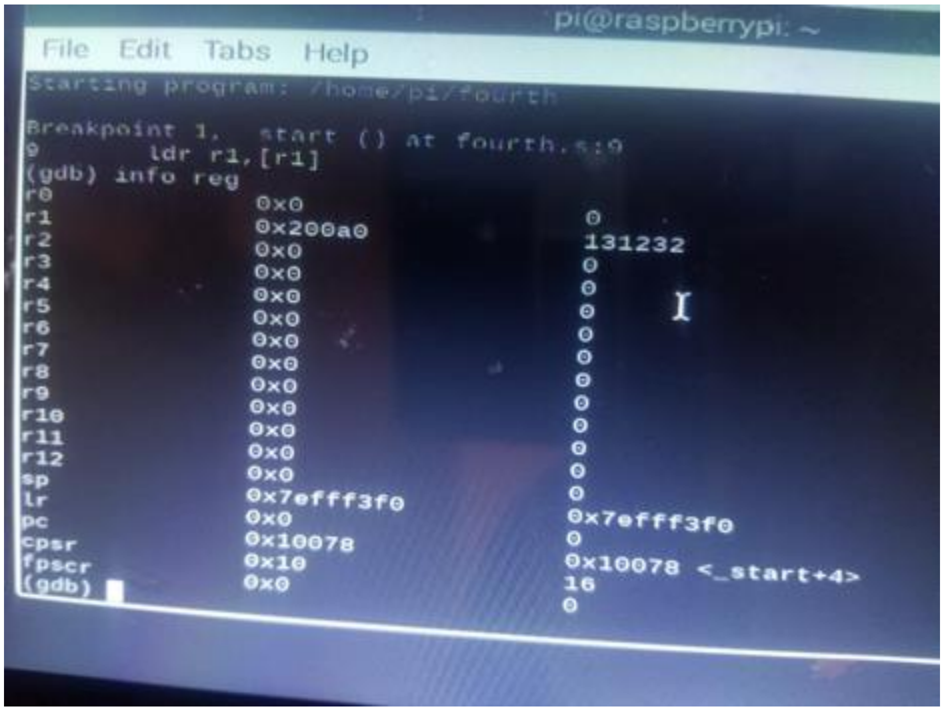


```
Breakpoint 1 at 0x10078: file fourth.s, line 9.
(gdb) run
Starting program: /home/pi/fourth

Breakpoint 1, _start () at fourth.s:9
9           ldr r1,[r1]
(gdb) x/1xw 0x10078
0x10078 <_start+4>:      0xe5911000
(gdb) x/1xw 4
0x4:    Cannot access memory at address 0x4
(gdb) x/1xw 0x10078
0x10078 <_start+4>:      0xe5911000
(gdb) b 8
Note: breakpoint 1 also set at pc 0x10078.
Breakpoint 2 at 0x10078: file fourth.s, line 9.
(gdb) x/1xw 0x10078
0x10078 <_start+4>:      0xe5911000
(gdb) stepi
11          cmp r1,#0
(gdb) b 10
Breakpoint 3 at 0x1007c: file fourth.s, line 11.
(gdb) x/1xw 0x1007c
0x1007c <_start+8>:      0xe3510000
(gdb)
```

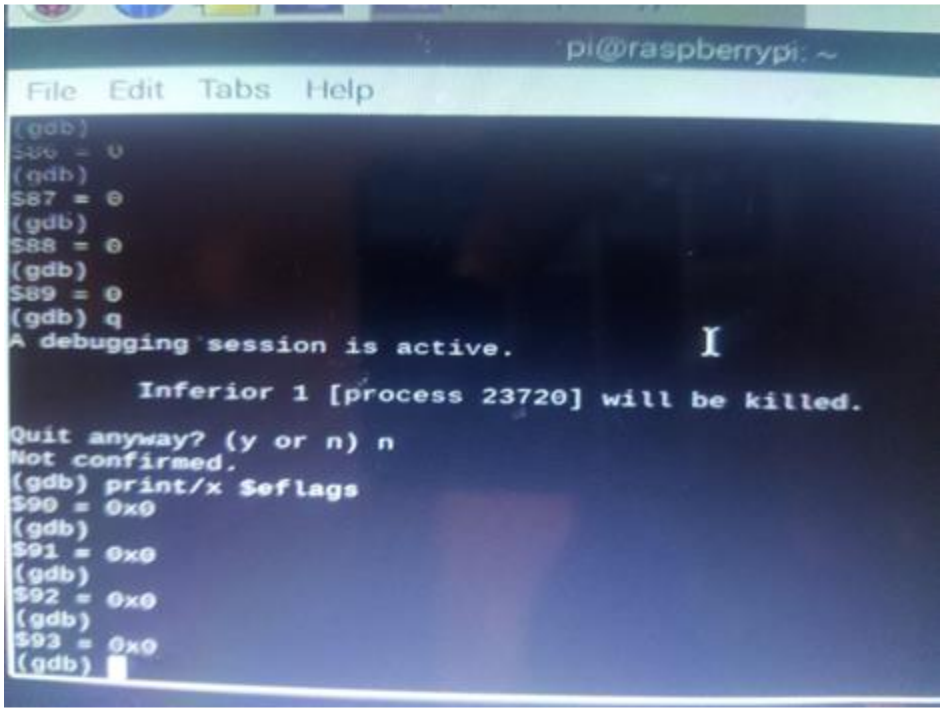


```
Reading symbols from fourth...done.
(gdb) list
1           .section .data
2           x:.word 0
3           y:.word 0
4           .section .text
5           .globl _start
6           _start:
7
8           ldr r1,=x
9           ldr r1,[r1]
10
(gdb)
11          cmp r1,#0
12          beq thenpart
13          b endofit
14
15          thenpart:
16
17                      mov r2,#1
18                      ldr r3,=y
19                      ldr r2,[r3]
20          endofit:
(gdb)                   mov r7,#1
```

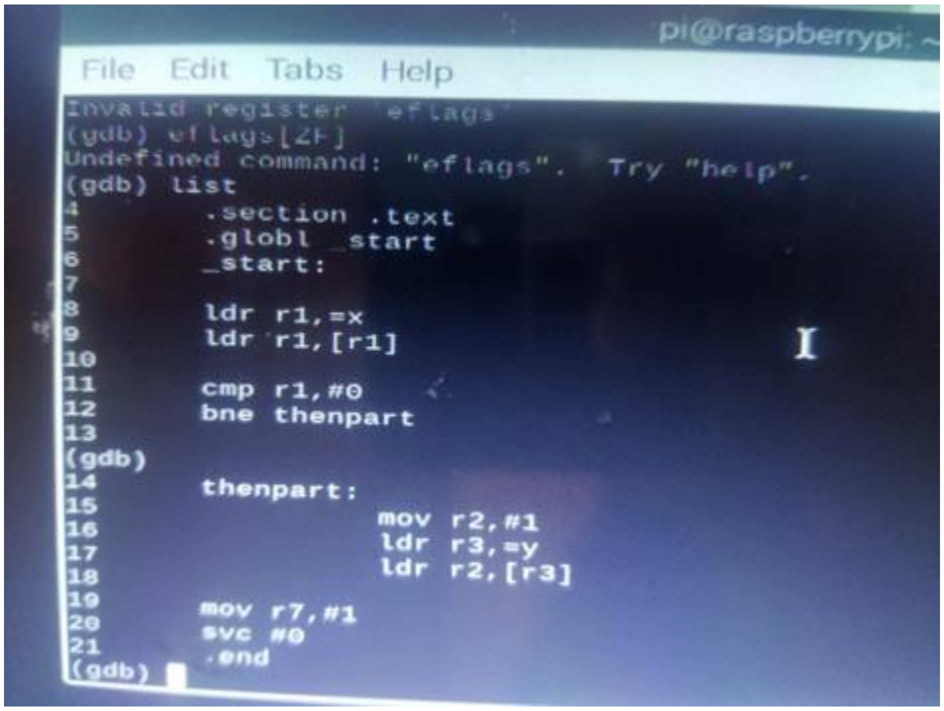
Part 2:



```
Starting program: /home/pi/fourth

Breakpoint 1, _start () at fourth.s:9
9           ldr r1,[r1]
(gdb) info reg
r0             0x0          0
r1             0x200a0      131232
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             0x7efff3f0   0x7efff3f0
lr             0x0          0
pc             0x10078      0x10078 <_start+4>
cpsr           0x10         16
fpscr          0x0          0
(gdb) 
```



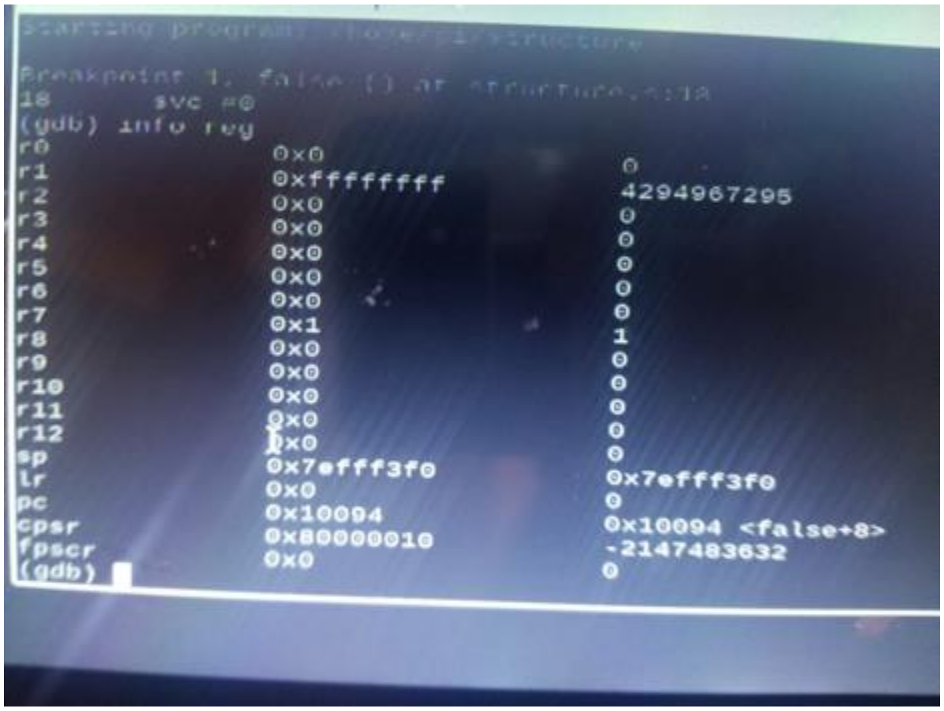
```
(gdb)
$86 = 0
(gdb)
$87 = 0
(gdb)
$88 = 0
(gdb)
$89 = 0
(gdb) q
A debugging session is active.

        Inferior 1 [process 23720] will be killed.

Quit anyway? (y or n) n
Not confirmed.
(gdb) print/x $eflags
$90 = 0x0
(gdb)
$91 = 0x0
(gdb)
$92 = 0x0
(gdb)
$93 = 0x0
(gdb) 
```



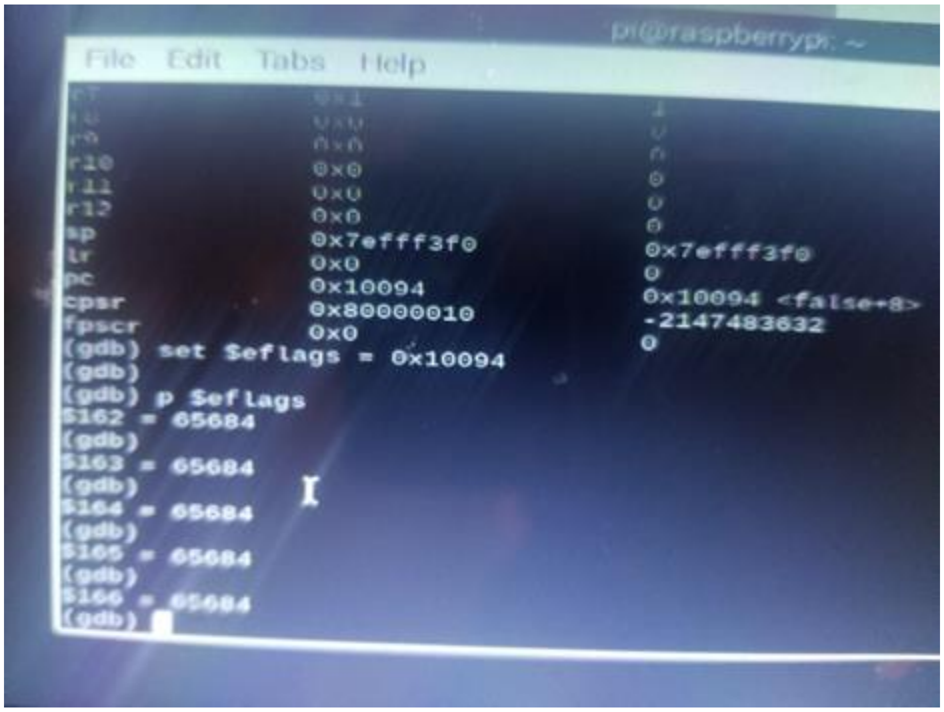
```
Invalid register `eflags`
(gdb) eflags[ZF]
Undefined command: "eflags".  Try "help".
(gdb) list
        .section .text
        .globl _start
        _start:

        ldr r1,=x
        ldr r1,[r1]

        cmp r1,#0
        bne thenpart

(gdb)
        thenpart:
                        mov r2,#1
                        ldr r3,=y
                        ldr r2,[r3]

        mov r7,#1
        svc #0
        .end
(gdb)
```

Part 3:

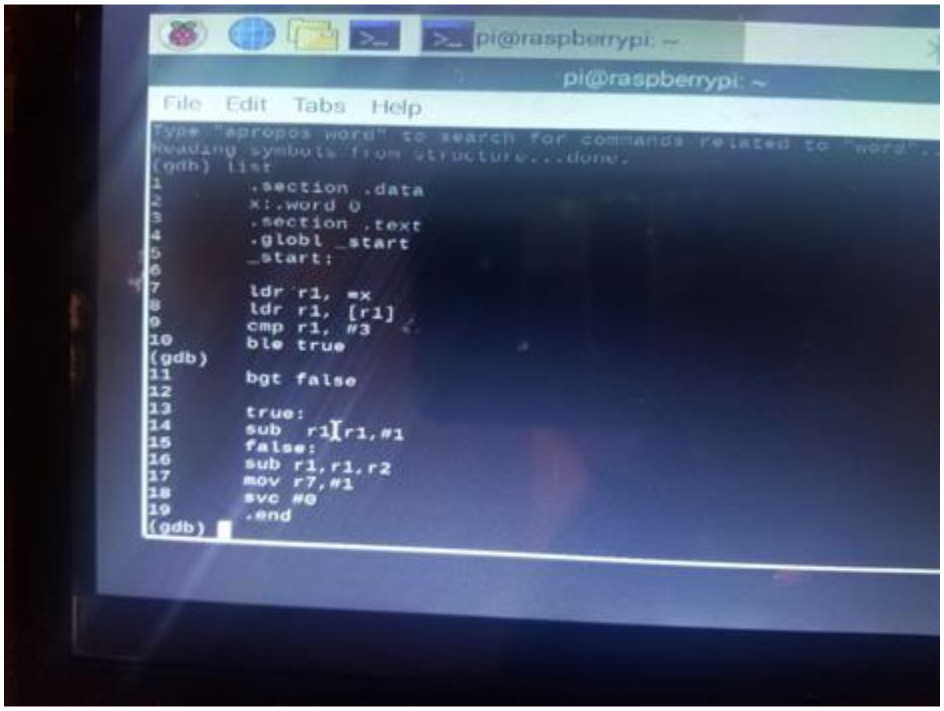
Like the X86 programming, the ARM instruction lets us specify where to jump by using a Boolean logic. And we can apply it to our program, especially when we want to compare two numbers.



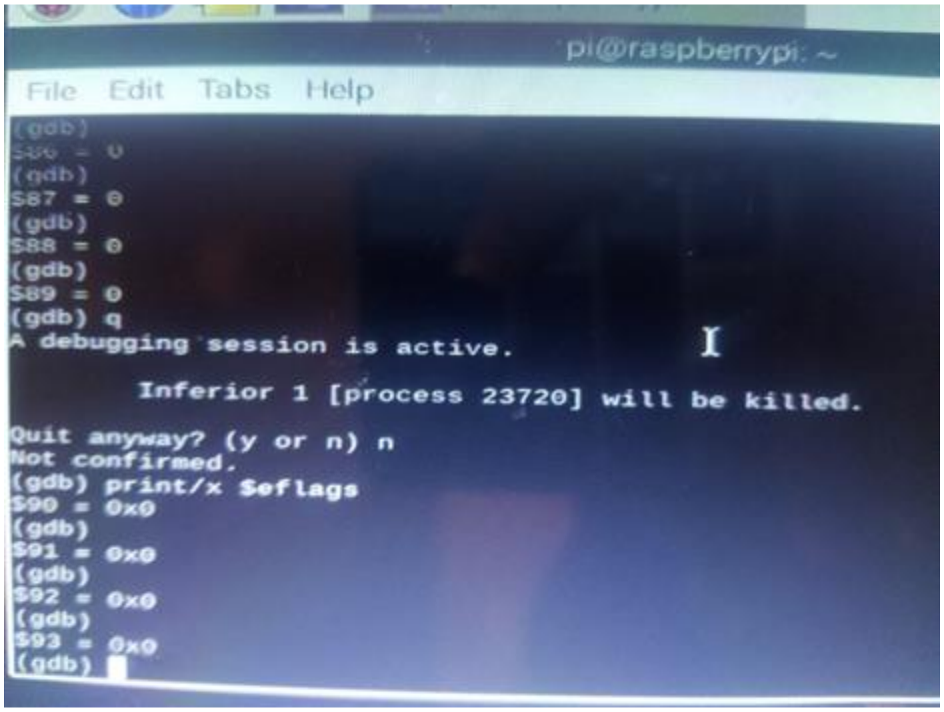
```
Starting program: /home/pi/structure

Breakpoint 1, false () at structure.s:18
18          svc #0
(gdb) info reg
r0             0x0                  0
r1             0xffffffff           4294967295
r2             0x0                  0
r3             0x0                  0
r4             0x0                  0
r5             0x0                  0
r6             0x0                  0
r7             0x1                  1
r8             0x0                  0
r9             0x0                  0
r10            0x0                  0
r11            0x0                  0
r12            0x0                  0
sp             0x7efff3f0           0x7efff3f0
lr             0x0                  0
pc             0x10094              0x10094 <false+8>
cpsr           0x80000010           -2147483632
fpscr          0x0                  0
(gdb)
```



```
r7             0x1                  1
r8             0x0                  0
r9             0x0                  0
r10            0x0                  0
r11            0x0                  0
r12            0x0                  0
sp             0x7efff3f0           0x7efff3f0
lr             0x0                  0
pc             0x10094              0x10094 <false+8>
cpsr           0x80000010           -2147483632
fpscr          0x0                  0
(gdb) set $eflags = 0x10094
(gdb)
(gdb) p $eflags
$162 = 65684
(gdb)
$163 = 65684
(gdb)
$164 = 65684
(gdb)
$165 = 65684
(gdb)
$166 = 65684
(gdb)
```



```
Type "apropos word" to search for commands related to "word"...
Reading symbols from structure...done.
(gdb) list
        .section .data
        x:.word 0
        .section .text
        .globl _start
        _start:

        ldr r1, =x
        ldr r1, [r1]
        cmp r1, #3
        ble true
(gdb)
        bgt false

        true:
        sub  r1,r1,#1
        false:
        sub r1,r1,r2
        mov r7,#1
        svc #0
        .end
(gdb)
```



```
(gdb)
$86 = 0
(gdb)
$87 = 0
(gdb)
$88 = 0
(gdb)
$89 = 0
(gdb) q
A debugging session is active.

        Inferior 1 [process 23720] will be killed.

Quit anyway? (y or n) n
Not confirmed.
(gdb) print/x $eflags
$90 = 0x0
(gdb)
$91 = 0x0
(gdb)
$92 = 0x0
(gdb)
$93 = 0x0
(gdb)
```

Appendix

Slack link:https://app.slack.com/client/TNAGW4TJ9/CNCCR06S3

YouTube link: https://youtu.be/DqcOHshowzk