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	Created video presentation submitted it to Youtube, Wrote Appendix	6	Youtube, Raspberry Pi
Austin Yuille	ayuille1@student.gsu.edu	Team Coordinator, Typed up parallel programming in report	6	Raspberry Pi
Micah Robins	mrobins1@student.gsu.edu	Created task sheet and wrote report	6	Google Docs
Matt Hayes	mhayes37@student.gsu.edu	Complete Task 3 Answers and organized the report	6	Google Docs
Nabeeha Ashfaq	nashfaq1@student.gsu.edu	Created and managed Github ProjectA3 with To-Do list	6	GitHub

Parallel Programming Skills

a) Foundation:

(1) Define the following:

- Task: a set of instructions that are performed by a processor

- Pipelining: a type of parallel processing. It divides a task into smaller steps so that multiple processors can do each step simultaneously.

- Shared Memory: when each processor running parallel tasks all have access to the same memory

- Communications: when parallel tasks exchange data

- Synchronization: a way to keep the timing of all the tasks running in parallel aligned. It usually involves making some tasks wait for another task to finish a specific process before they can move on.

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

Flynn's taxonomy classifies multiple processor setups in terms of two things: the instruction stream and the data stream. Since there can be either one stream or multiple streams for each, there are 4 possible classifications.

1. SISD (Single Instruction, Single Data). In SISD there is no parallel computing going on. During one clock cycle there is one instruction stream and one data stream used by the processor.
2. SIMD (Single Instruction, Multiple Data). All the processors work from the same instruction stream but can each work on a different data element simultaneously.
3. MISD (Multiple Instruction, Single Data). One data stream goes to multiple processors, and each processor works with the data independent of the others.
4. MIMD (Multiple Instruction, Multiple Data). The most common type of parallel computer. Each processor can work independently of the others because each one can have its own instruction stream and data stream.

(3) What are the Parallel Programming Models?

1. Shared memory without threads

2. Threads
3. Distributed memory / message passing
4. Data parallel
5. Hybrid
6. SPMD (Single Program Multiple Data)
7. MPMD (Multiple Program Multiple Data)

(4) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?

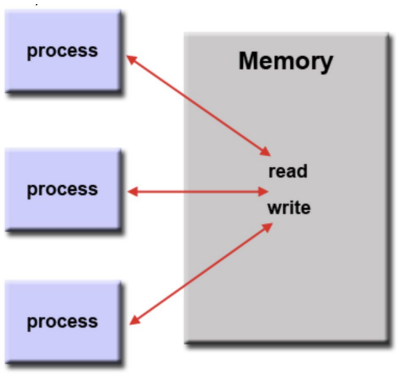
Shared memory machines are classified as either UMA (Uniform Memory Access) or NUMA (Non-Uniform Memory Access). In UMA, all the processors share a central chunk of memory. Therefore they have equal access to the memory and access times are equal for all processors. In NUMA, not all processors have equal access time to all of the memory. Since a NUMA system is usually made from linking together multiple symmetric multiprocessors (SMP), sometimes one processor will need to access memory on a different SMP. It is slower to access the memory on a different SMP. They all have direct access to each other's memory, but it is faster when a SMP is accessing its own memory rather than reaching across to a different one.

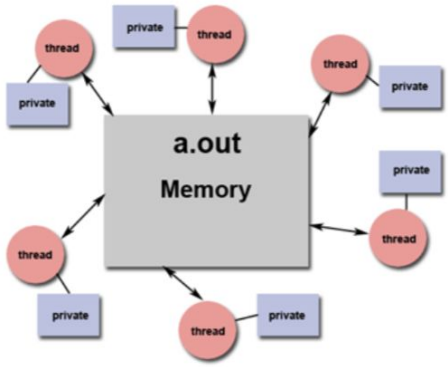
OpenMP uses shared memory via threads. It can be UMA or NUMA since both are shared memory architectures.

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

While both models share memory resources, in the thread model each task running concurrently (threads) will also have their own local data that they work with independent of the other threads. That is the main difference. In the shared memory model each task is utilizing all of the same memory that every other task is using. In the thread model there is local memory for each thread in addition to the shared memory. See below for illustrations:

<u>Shared Memory:</u> <u>Threads:</u>





(6) What is parallel programming?

Parallel programming is a technique to speed up the workflow of a computer. Rather than working on tasks one at a time, multiple tasks can be worked on simultaneously. This makes the run time for a program shorter and the program more efficient.

(7) What is system on chip (SoC)? Does Raspberry Pi use system on SoC?

SoC is when all of the components required for a computer (CPU, RAM, GPU, etc) are all contained on one component. Basically, putting the entire computer on one chip. The Raspberry Pi is an example of a SoC setup.

(8) Explain what the advantages are of having a System on Chip rather than separate CPU, GPU, and RAM components.

A SoC is smaller, consumes less power, and is cheaper than the traditional setup of having all of the components separate.

b) Parallel Programming Basics:

For this part of the assignment we were asked to copy a few programs that utilized parallel programming and examine the outcomes. For the first part of this we had to copy the program, "parallelLoopEqualChunks.c" (see Appendix B, 1). This code would take a certain number of iterations, we used the constant REPS, and divide it among however many threads were passed as an argument in the execution. Our REPS constant was set to sixteen for all of our executions. After copying the code, we were asked to execute the code by using the command "./pLoop 4" after compiling the code. This produced a result that used four threads with each thread producing four results that were the number of the thread and one of the numbers between the thread's number times four and the thread's number times four plus three. For example, thread zero produced the results that contained iteration zero through three (See Appendix B, 2). This showed us that all of the threads had an equal number of iterations, with each having performed four. Next we tried passing various numbers of threads that did not go into sixteen evenly, such as five and seven(See Appendix B,3 and B,4). We found that the threads would

even divide the iterations and give the extras to the first threads. For example, when we passed five as an argument thread zero performed four iterations and threads one through four performed three iterations(See Appendix B,3). This adds up to sixteen showing that all of the iterations were completed. When we used 7 threads, threads 0 and 1 performed 3 iterations while threads 2 through 6 performed 2(See Appendix B,4).

For the second part of the parallel programming section we were asked to compile and run two versions of the program "parallelLoopChunksOf1.c". Both versions of this program have the same task as the first program. They take an argument from the execution and use that many threads to complete a certain amount of iterations, which is still a constant of 16. For the first version, the program uses a pragma with the keywords static and schedule(See Appendix B, 5). These keywords cause the threads to each do one iteration normally before doing them simultaneously. This causes the result to be varied from the first program, because now the threads have done different iterations. They each did one in sequential order to begin with so thread zero did iteration zero, thread one did iteration one, and so on. Then when they began to do work simultaneously, they did every fourth iteration from their first one. For example, thread zero did iterations zero, four, eight, and twelve(See Appendix B,7). After seeing this, we commented the pragma code block and wrote a new one that doesn't use a schedule but accomplishes the same goal(See Appendix B, 6). This code block completes this task by having more complex for loop conditions. Each thread has its own for loop that starts from its id, and it is incremented by the number of threads being used. The result we received was the same, for example, if you look at thread zero it completed iterations zero, four, eight, and twelve as before(See Appendix B, 7). Looking at thread zero you can see how this works. The loop starts at the thread's id, zero in this case, then adds four until it reaches a number equal to or greater than sixteen, so zero, four, eight, twelve and then it stops at sixteen because it is equal to the number of iterations we wanted to complete. When using a number of threads that doesn't go into sixteen evenly it still works the same as before, so the threads still divide the work evenly and the first threads do the extra iterations. For example, when we used five threads, thread zero did four iterations while the other four threads did three(see Appendix B, 10). However, instead of doing every fourth iteration they did every fifth iteration.

For the third and final part of the parallel programming section we wrote a program, "reduction.c" that takes an array of size one million and adds all of the integers inside it and prints the sum. It does this using a sequential method and then a parallel method(See Appendix B, 11 and B, 12). We produced three results because we were asked to try this three times with certain pieces of code commented or uncommented(See Appendix B, 13). The first time we had all of line 47 commented, since the pragma was not actually used this was essentially the same as the sequential method so it yielded the correct result. The second time we uncommented part of line 47 leaving just the reduction piece commented, this gave us the wrong result as the parallel sum did not equal the sequential sum. We found that the issue in the code was that each thread would return a value for sum, but since they shared the variable and didn't add their results each thread would overwrite the previous value stored in sum. Finally we uncommented the reduction statement in line 47. This final execution produced the correct result while using a sequential method and a parallel method. The reduction statement contains an addition sign and the variable name sum. This causes each thread to take its result and add it to the sum value. Since each thread added different parts of the array when all of their results are summed together we get the correct result.

ARM Assembly Programming

Part 1:

For the first part of the ARM Assembly section, we were given code that would load various values into registers alongside declaring a signed halfword **a** to be loaded as well (See Appendix C, 1). The file was dubbed third.s as we started copying the given ARM Assembly code into nano. When trying to run the program, we immediately ran into a problem with ARM Assembly not recognizing .shalfword as a keyword(See Appendix C, 2). The error we recieved when trying to run the program was "unknown pseudo-op '.shalfword'". During analysis, we tried switching out the keyword .shalfword of variable **a** with other likely signed keyword names (sbyte, sword, shword, etc.) to no avail as we received the same error message(See Appendix C, 3). After more research, we concluded that signed keywords did not exist in ARM Assembly, so we would have to convert **a** to a signed halfword another way. Variable **a** was declared as an .hword initially (See Appendix C, 4). In order to define the value as a signed halfword, we used LDRSH instead of LDR to load **[r1]** into **r1** (See Appendix C, 5 and Appendix C, 6). If the signed recognition problem solved, all that was left was loading the other variables and running the program.

After all of the other registers were loaded, the instructions were run without a hitch, reflecting the expected results. When using commands to display the results, we had to be sure not to overlap differing formats and sizes. To display and verify r1 as -2, we used x/1xh 0x200a4 to display its 0xFFFe hex form and x/1dh 0x200a4 to show the -2 decimal form (See Appendix C, 7). We also used x/1sh 0x200a4 and found that it gave the result u"\xffe*unknown symbol*". Part of the result "fffe" seems to be correct, however, the rest of it seems weird. We believe that the s is interpreted as string so it produces the odd result.

Part 2:

The second part of ARM Assembly asked for an arithmetic calculation of registers: *Register = val2 +3 + val3 - val1*

The program would be written in nano, dubbed arithmetic3.s. Furthermore, the values **val1** and **val2** were to be declared as unsigned 8-bit integers (unsigned bytes), and **val3** was to be declared as a signed 8-bit integer (signed byte). The values of **val1**, **val2**, and **val3** were -60, 11, and 16 respectively, though **val1** displayed as 196 due to being an unsigned byte (See Appendix C, 9)

The values were declared in memory easily enough, using .byte for each variable (See Appendix C, 8). Using our knowledge of the previous part, we knew to load the two unsigned bytes with LDRB and load the signed byte with LDRSB (See Appendix C, 8 and C, 10). For the arithmetic equation, we added the immediate **#3** to **r1** (**val2**), added the resulting **r1** to **r2** (**val3**), then subtracted **r0** (**val1**) from the resulting **r2**, creating the final result in the **r2** register(See Appendix C, 9) . The **r2** register displays 0xFFFFFF5a, converted to decimal as -165, a negative result(See Appendix C, 11).

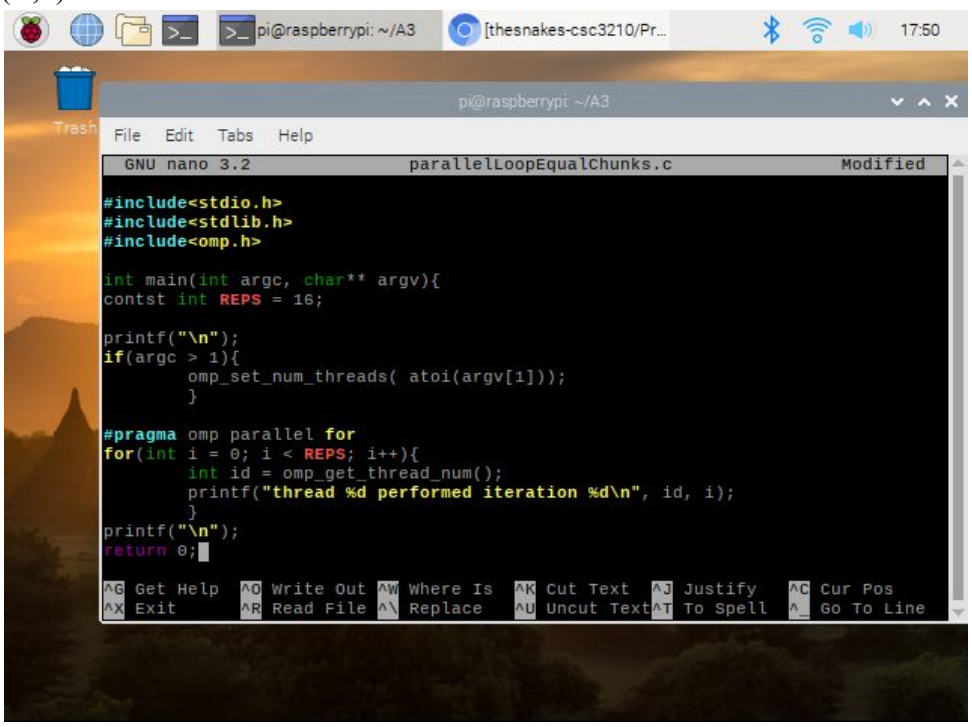
The results calculated as expected, but there was a glaring issue when we checked the registers: The cpsr flags did not trigger at all. Comparing ARM Assembly to x86 syntax, we assumed the flags would trigger after calculating the result of an arithmetic instruction. After seeing the results of this calculation as well as a few errant test cases, we realized that this was not necessarily the case(See Appendix C, 11). Further research yielded a solution: Adding an -s suffix to an arithmetic operand would grant it the privilege to affect **cpsr** flags (See Appendix C, 10). As simple as that, an -s suffix was added to all of the arithmetic operands, and after running the program, the flags triggered as expected (See Appendix C, 12). A quick calculation of the given equation would yield the -165 decimal. The **cpsr** register showed 0x080000010(See Appendix C, 13). Converting from hex, 10h would yield the usual 16th bit present in the cpsr. The 80h at the very beginning of the register would indicate that the negative flag present on bit 31 has been triggered. The results are finally complete.

Appendix A: Links

Github Project: https://github.com/orgs/thesnakes-csc3210/projects/1
Github Repository: https://github.com/thesnakes-csc3210/ProjectA3
Video Presentation: https://youtu.be/kU20uqccpdY

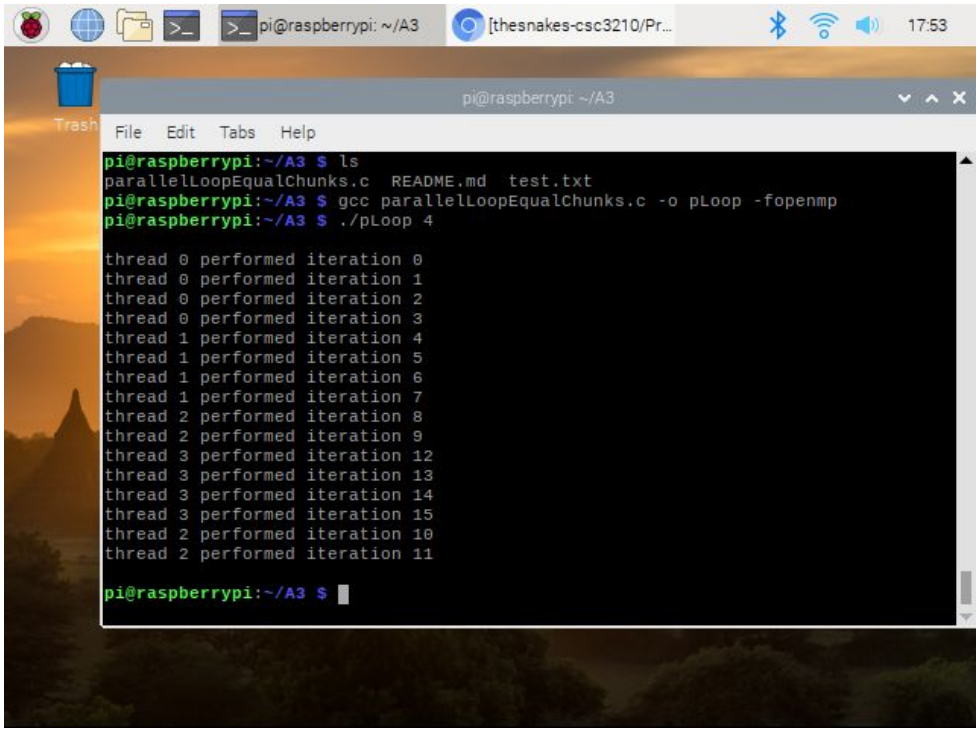
Appendix B: Parallel Programming Task A3

(B,1)



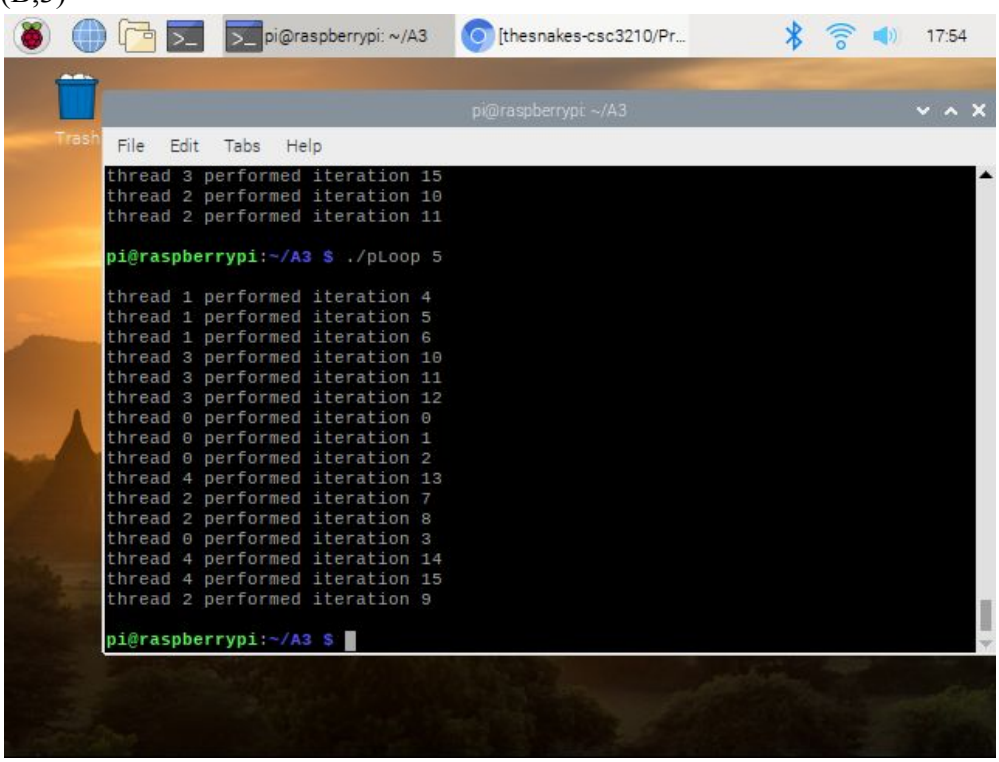
Code snippet of parallelLoopEqualChunks.c

(B,2)



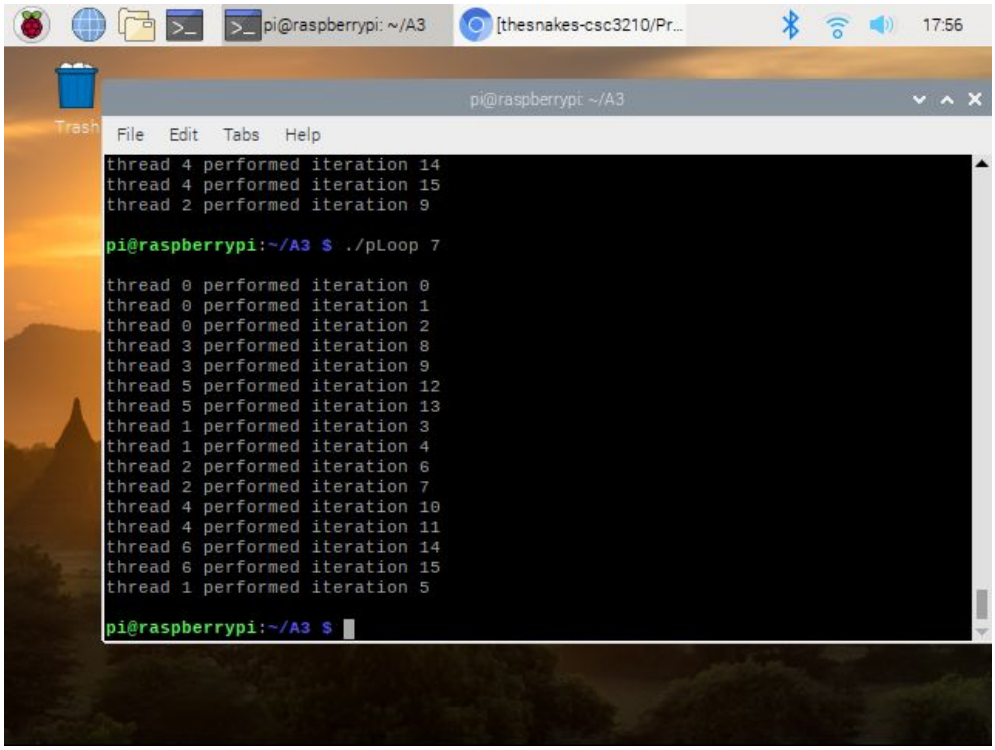
Result of parallelLoopEqualChunks.c when using 4 threads.

(B,3)



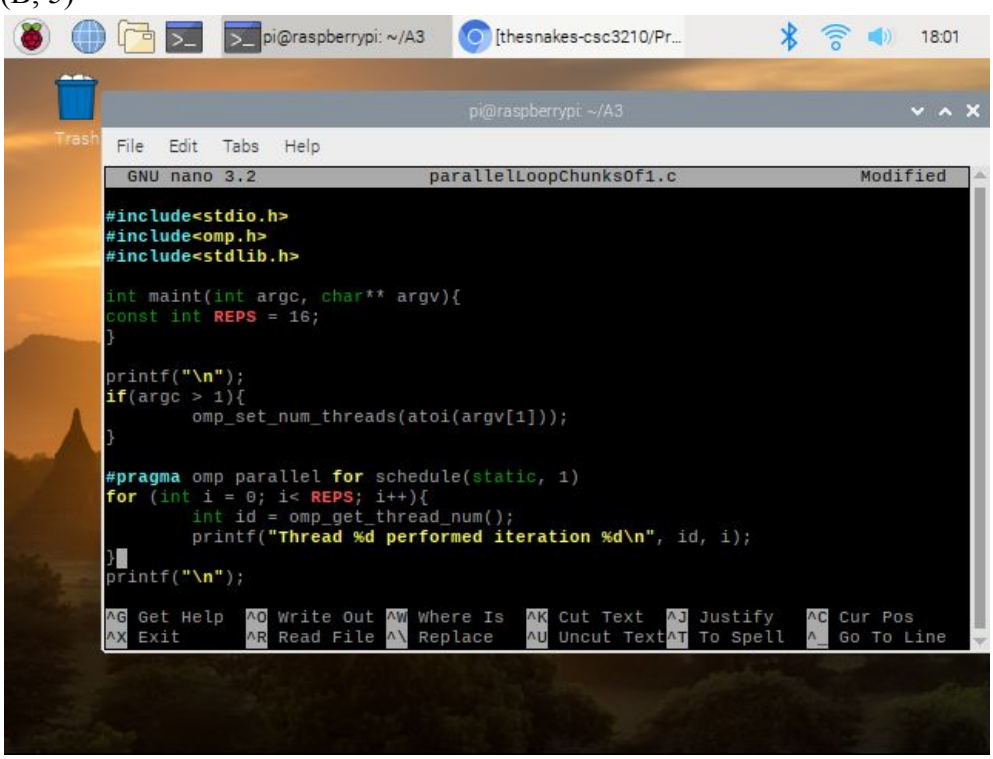
Result of parallelLoopEqualChunks.c when using 5 threads.

(B,4)



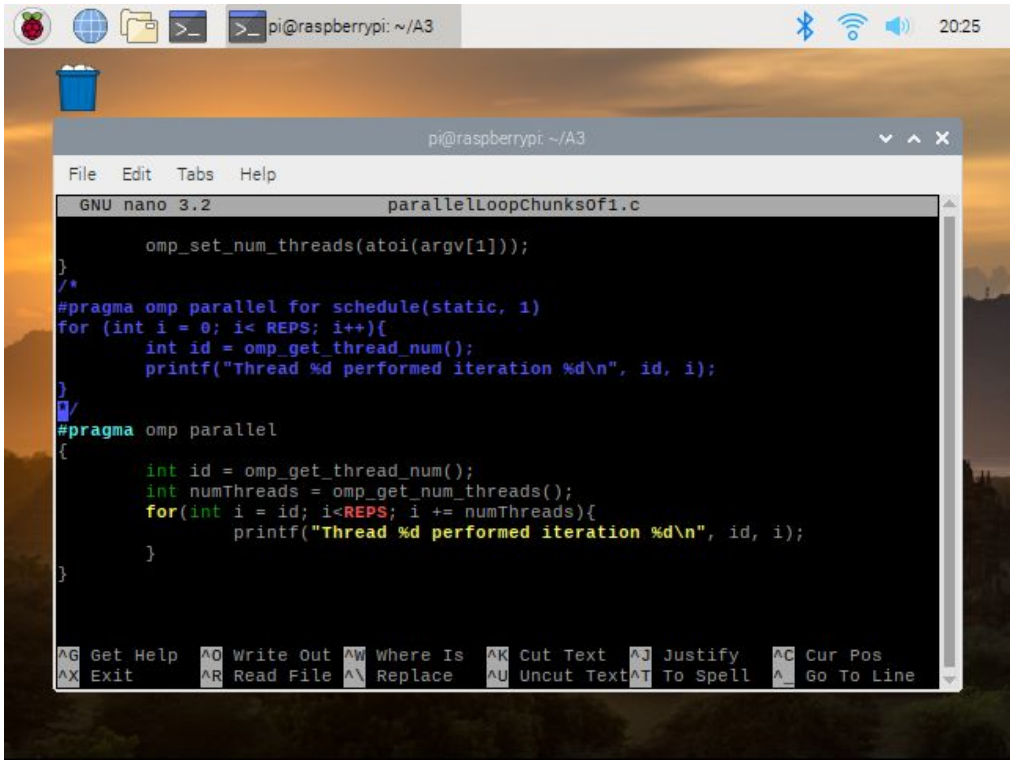
Result of parallelLoopEqualChunks.c when using 7 threads

(B, 5)



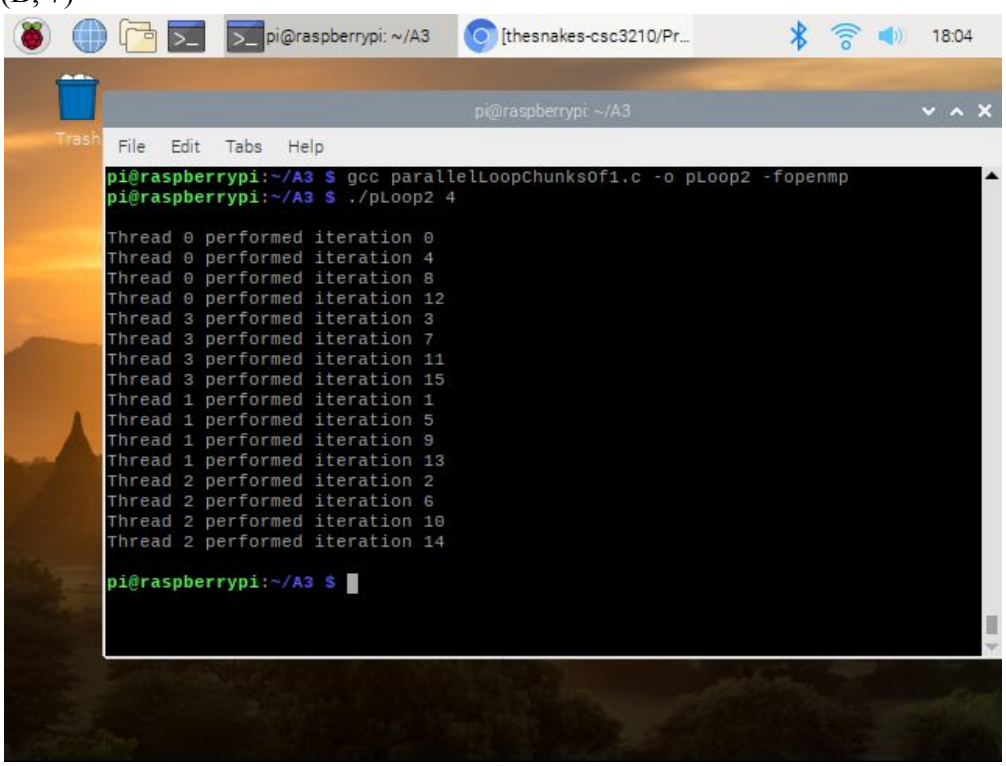
Code snippet of first version of parallelLoopChunksOf1.c.

(B, 6)



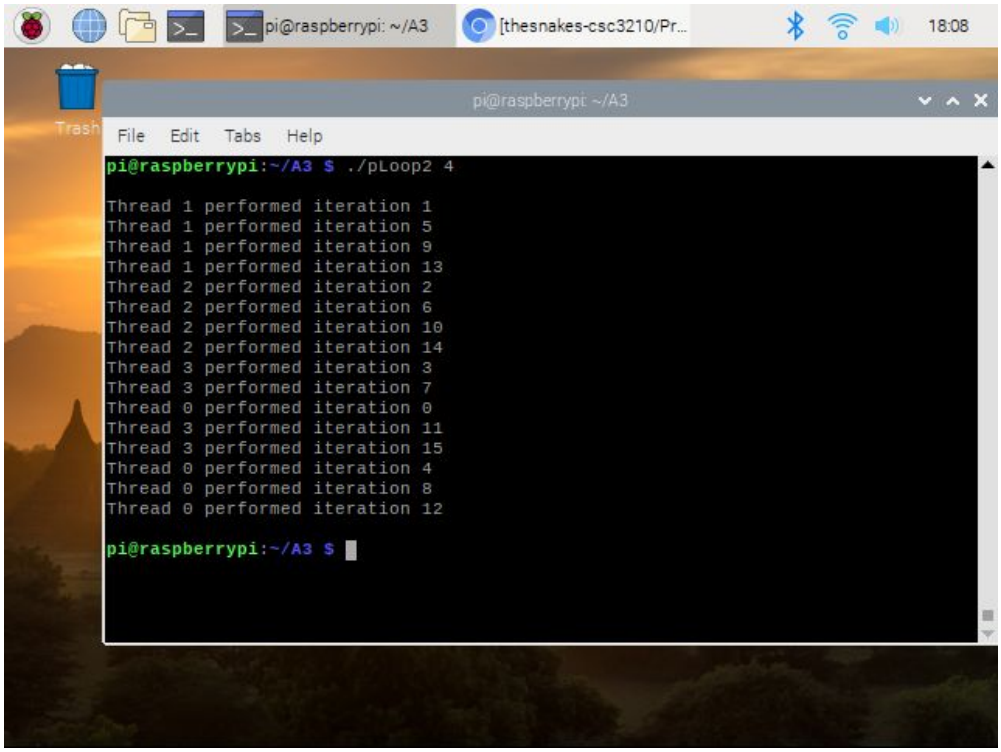
Code snippet of second version of parallelLoopChunksOf1.c.

(B, 7)



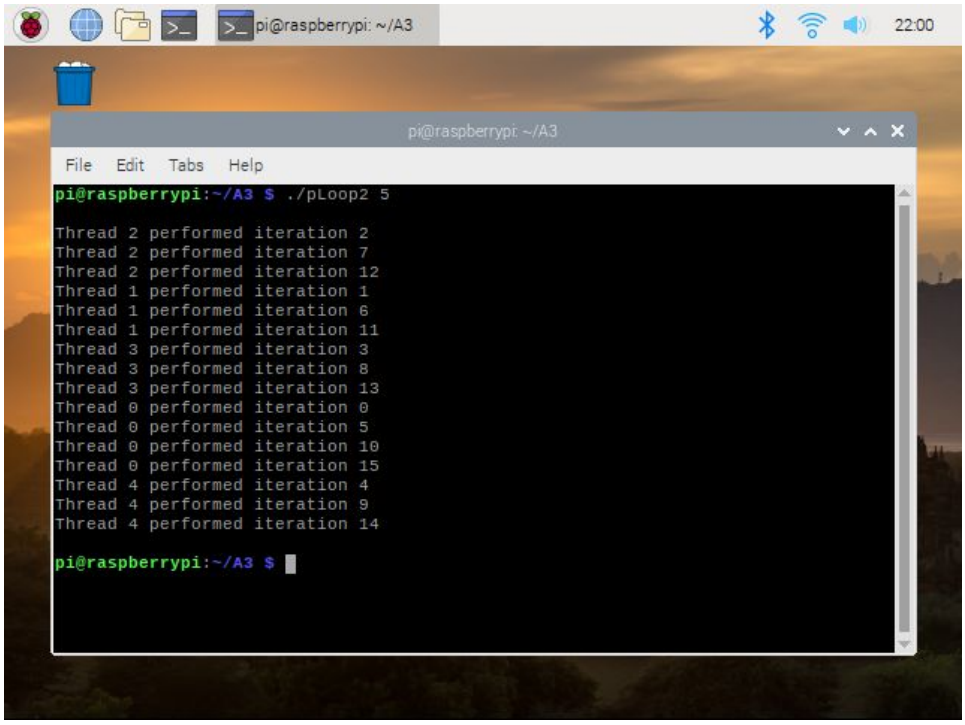
Result of first version of parallelLoopChunksOf1.c.

(B, 8)



Result of second version of parallelLoopChunksOf1.c when using four threads.

(B, 9)



Result of second version of parallelLoopChunksOf1.c when using five threads.

(B, 10)

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

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
        int array[SIZE];

if (argc > 1) {
        omp_set_num_threads(atoi(argv[1]));
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE));

return 0;
}

void initialize(int* a, int n){
int i;
for(i = 0; i < n; i++){
        a[i] = rand() % 1000;
        }
}


int sequentialSum(int* a, int n){
int sum = 0;
```

Code snippet of final version of reduction.c
(B,11)

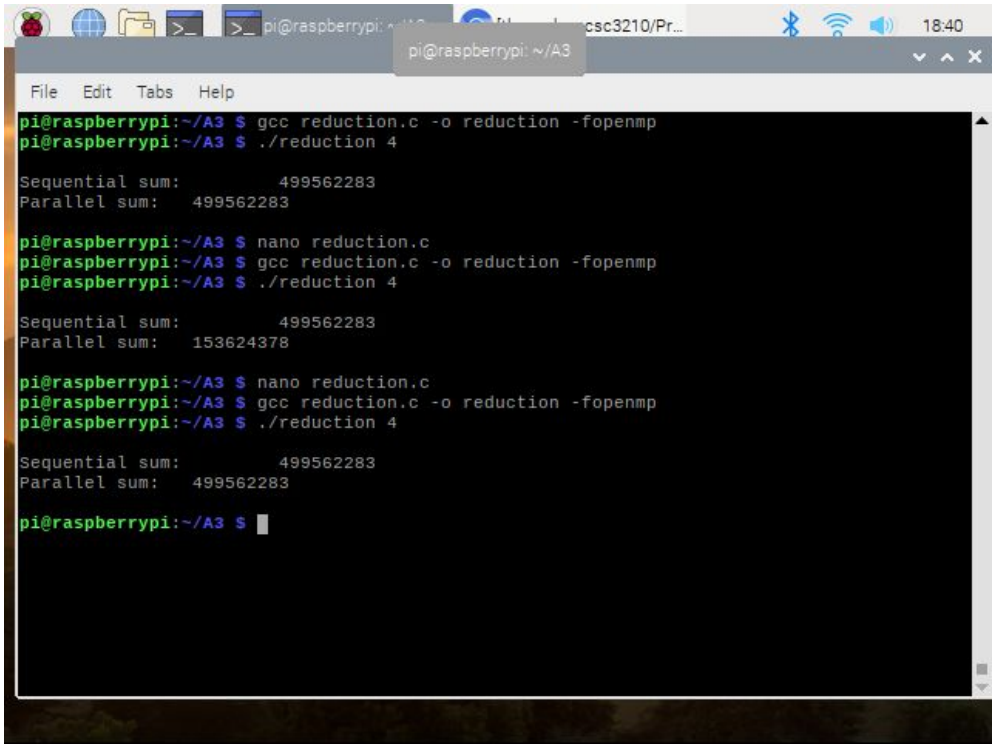
```c
int i;
for(i = 0; i < n; i++){
        sum += a[i];
        }
return sum;
}


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;
}


```

Code snippet of final version of reduction.c (continued).

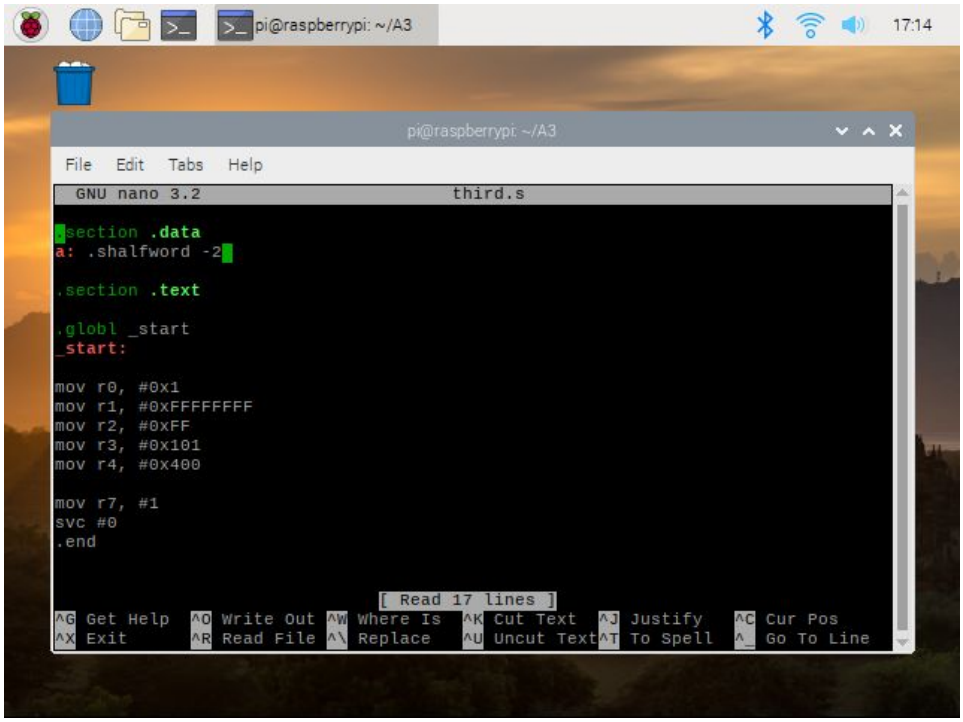
(B, 12)



Results of all versions of reduction.c

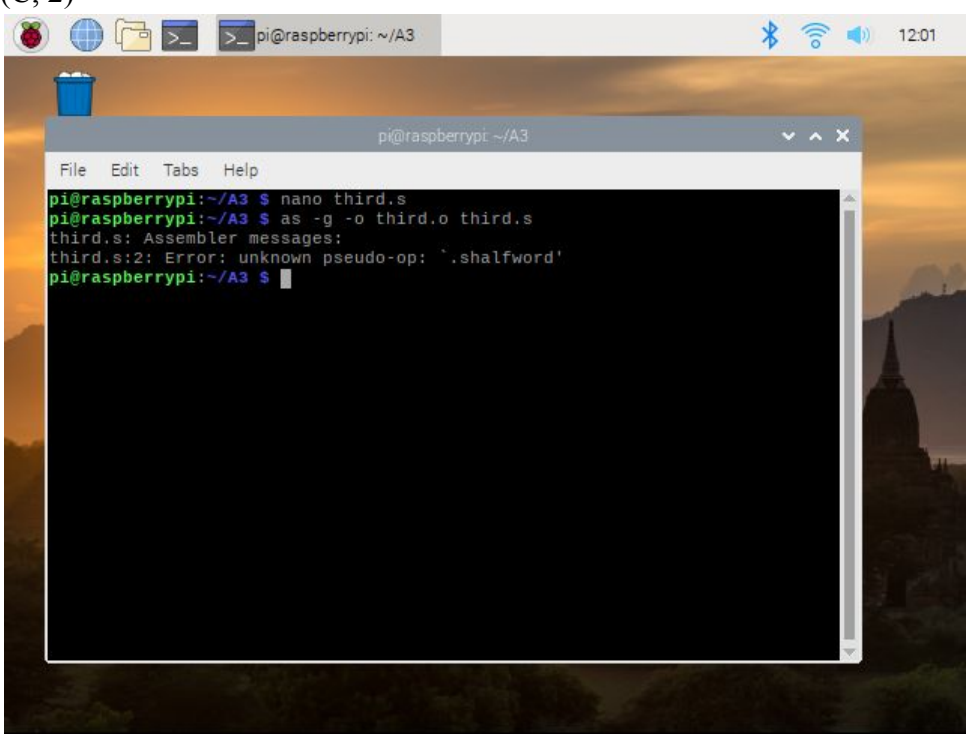
<u>Appendix C: ARM Assembly Programming A3</u>

(C, 1)



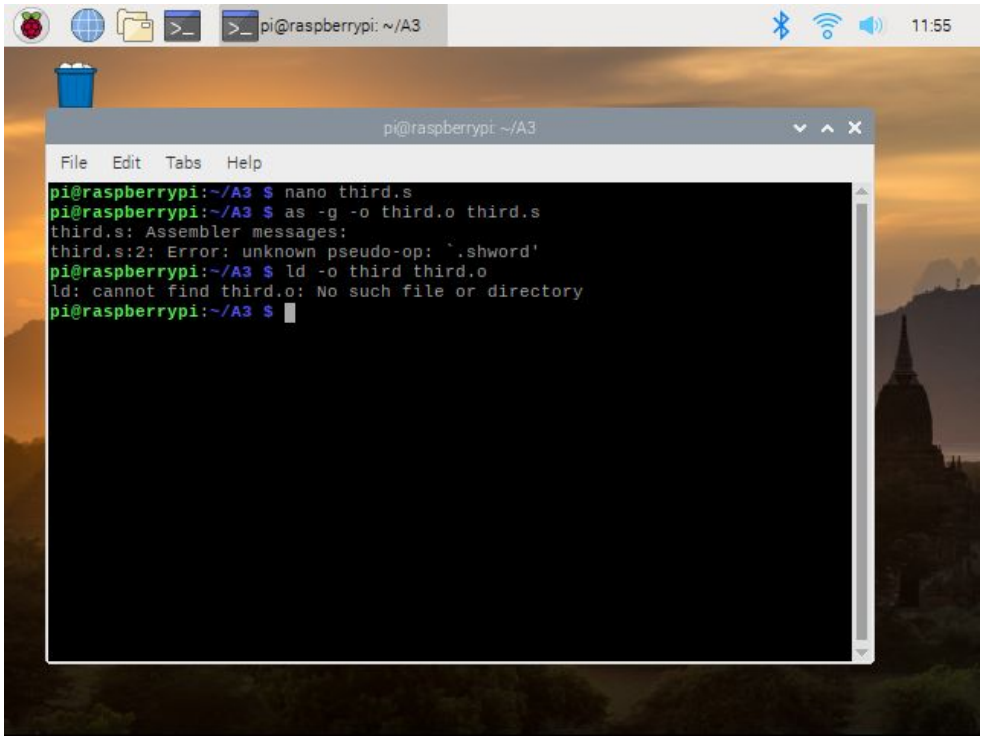
Given Code for Part 1 of ARM Assembly

(C, 2)



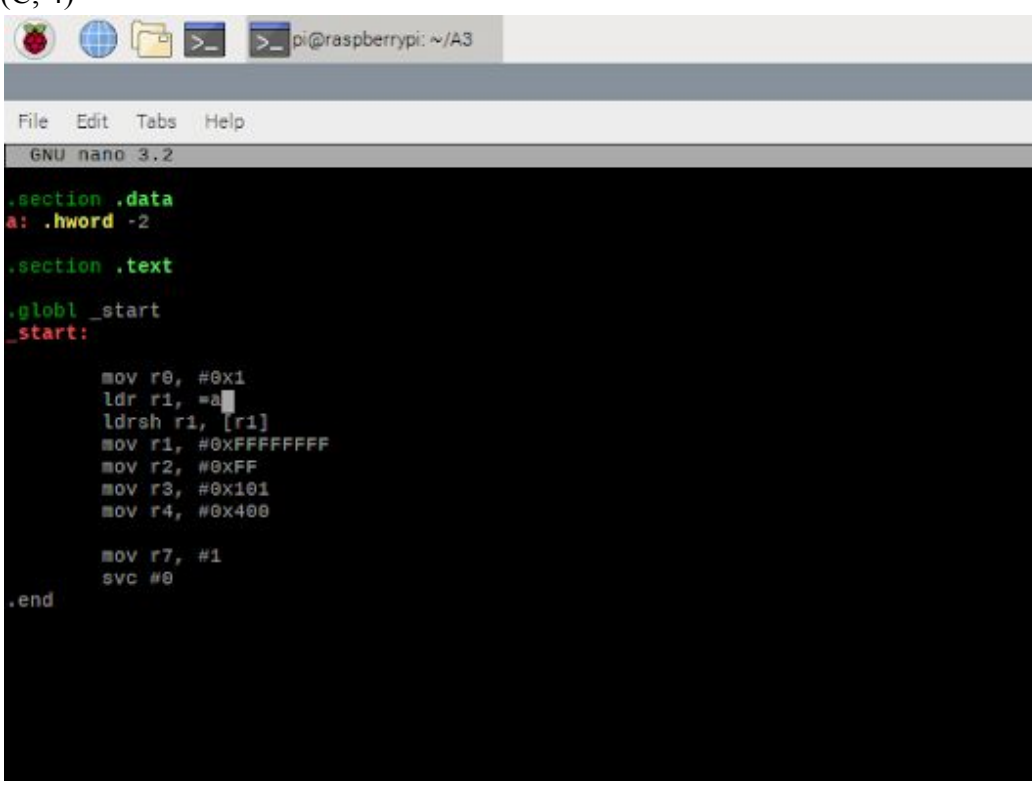
Error message when trying to use .shalfword as a data type.

(C, 3)



The error still occurred when using .shword.

(C, 4)



Memory **a** declared as an unsigned .hword -2 before being loaded as signed

(C, 5)

```
pi@raspberrypi: ~/A3                                   15:30
pi@raspberrypi: ~/A3
File  Edit  Tabs  Help
Reading symbols from third...done.
(gdb) list
1       .section .data
2       a: .hword -2
3
4       .section .text
5
6       .globl _start
7       _start:
8
9               mov r0, #0x1
10              ldr r1, =a
(gdb) list
11              ldr r1, [r1]
12              ldr r1, =a
13              ldrsh r1, [r1]
14              mov r1, #0xFFFFFFFF
15              mov r2, #0xFF
16              mov r3, #0x101
17              mov r4, #0x400
18
19              mov r7, #1
20              svc #0
(gdb) b 9
Breakpoint 1 at 0x10078: file third.s, line 10.
(gdb) run
Starting program: /home/pi/A3/third

Breakpoint 1, _start () at third.s:10
10              ldr r1, =a
(gdb)
```

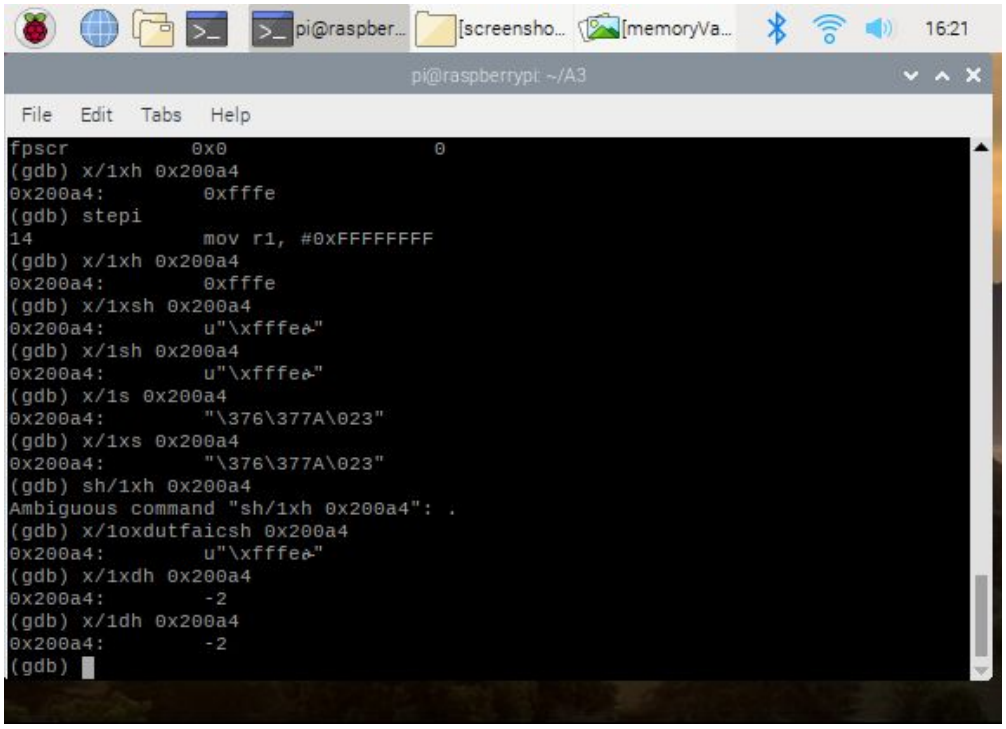
Loaded **[r1]** into **r1** as a signed .hword via LDRSH

(C, 6)

```
pi@raspberrypi: ~/A3                                   15:32
pi@raspberrypi: ~/A3
File  Edit  Tabs  Help
(gdb) stepi
11              ldr r1, [r1]
(gdb) stepi
12              ldr r1, =a
(gdb) info registers
r0             0x1                 1
r1             0x1341fffe          323092478
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           0x10                16
fpscr          0x0                 0
(gdb) stepi
13              ldrsh r1, [r1]
(gdb) stepi
14              mov r1, #0xFFFFFFFF
(gdb) info registers
r0             0x1                 1
r1             0xfffffffe          4294967294
r2             0x0                 0
```

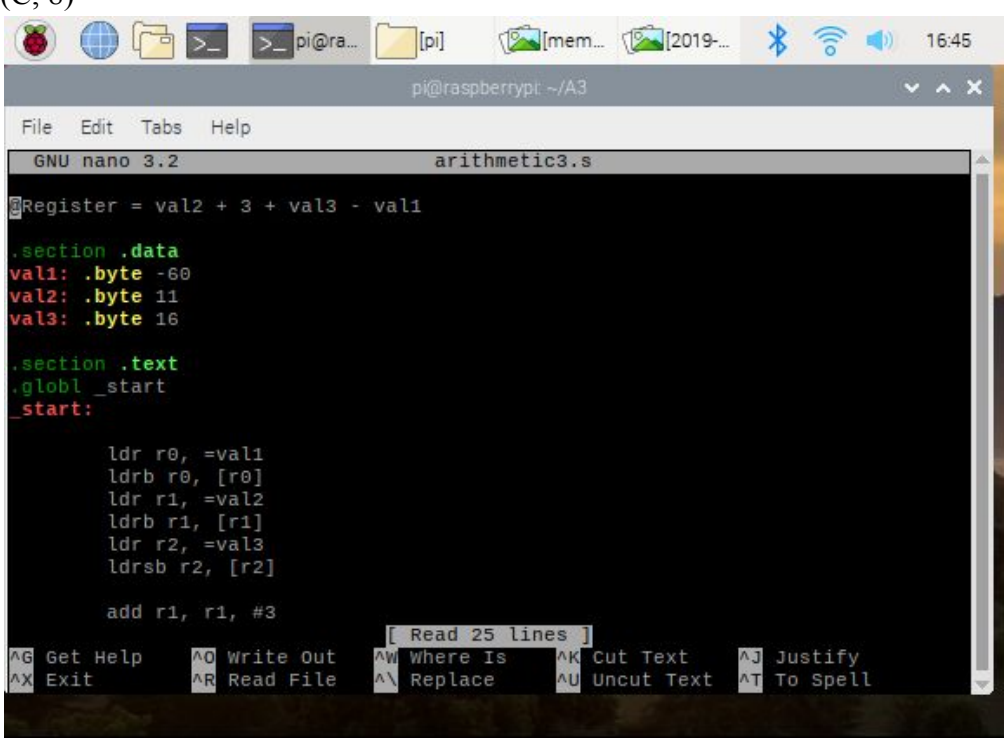
Comparison of **r1** LDR vs LDRSH, using LDRSH to display -2 in hex correctly

(C, 7)



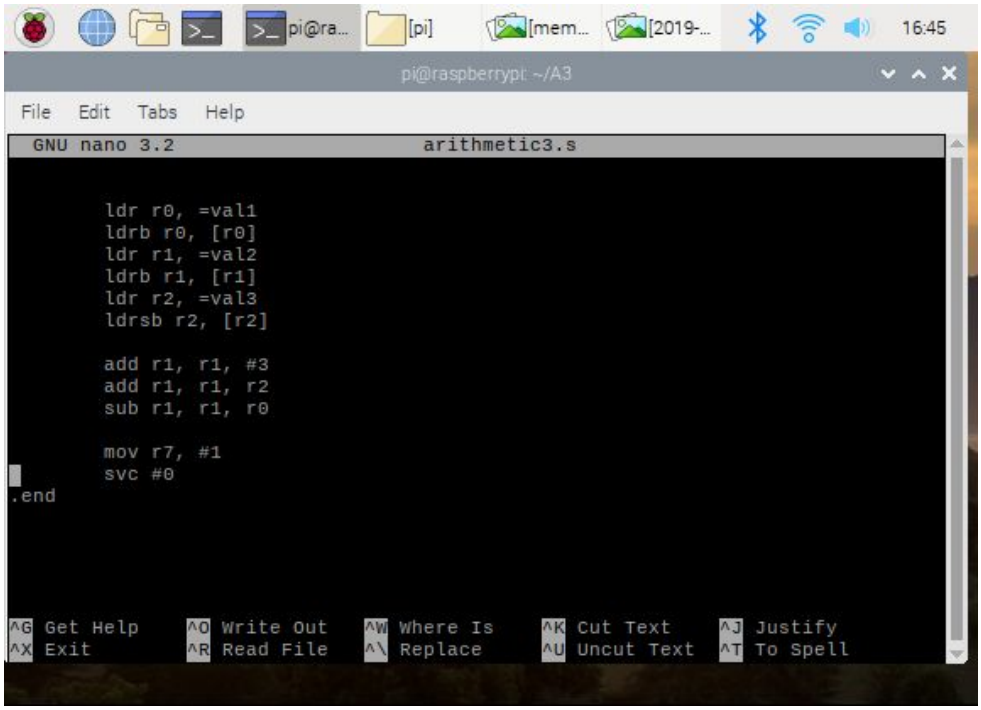
Used x/1xh 0x200a4 command to display hex 0xFFFE and x/1dh 0x200a4 to display decimal -2.

(C, 8)



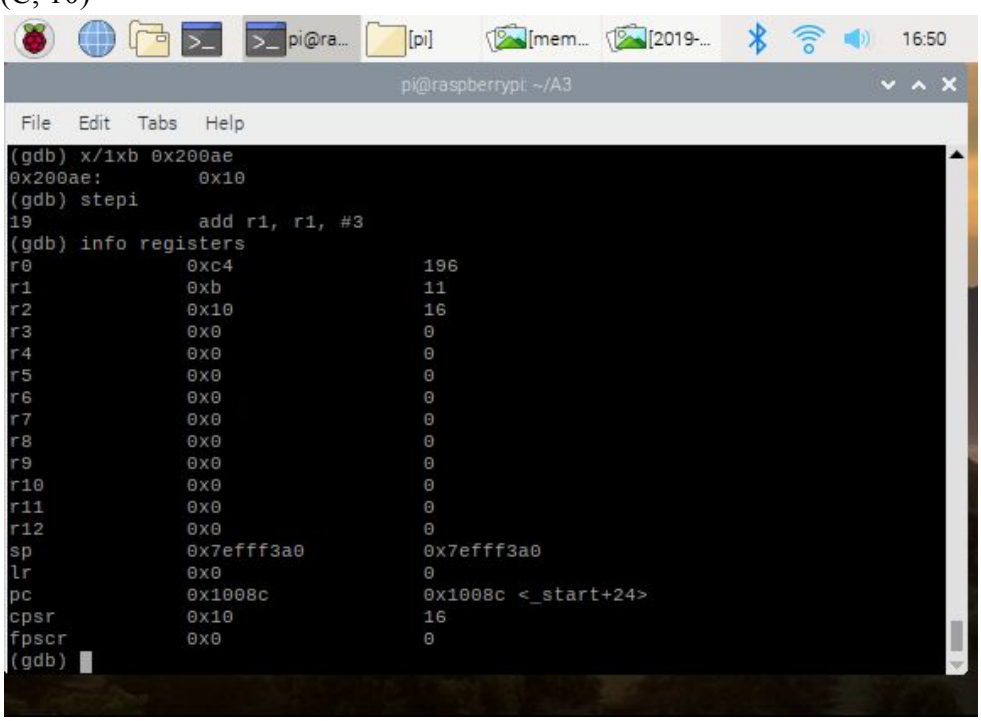
Individually loaded the given values into .data as .bytes. Loaded values **val1** and **val2** into respective registers with LDRB, and **val3** with LDRSB.

(C, 9)



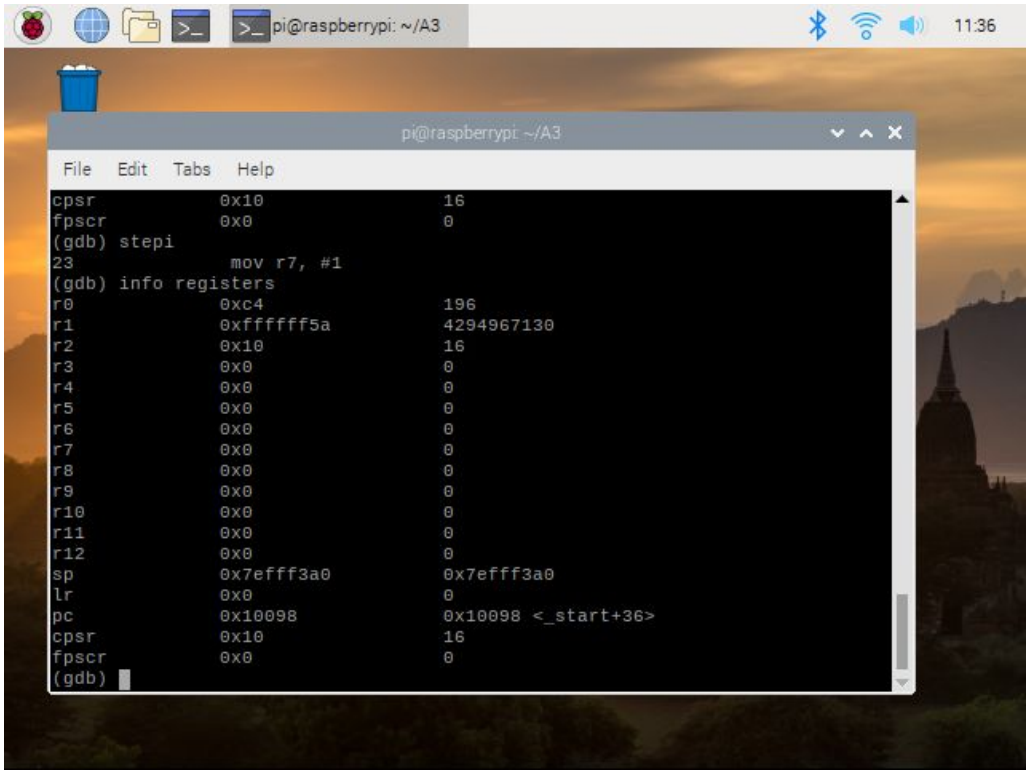
Arithmetic instructions to calculate the given equation: **Register = val2 +3 + val3 - val1**

(C, 10)



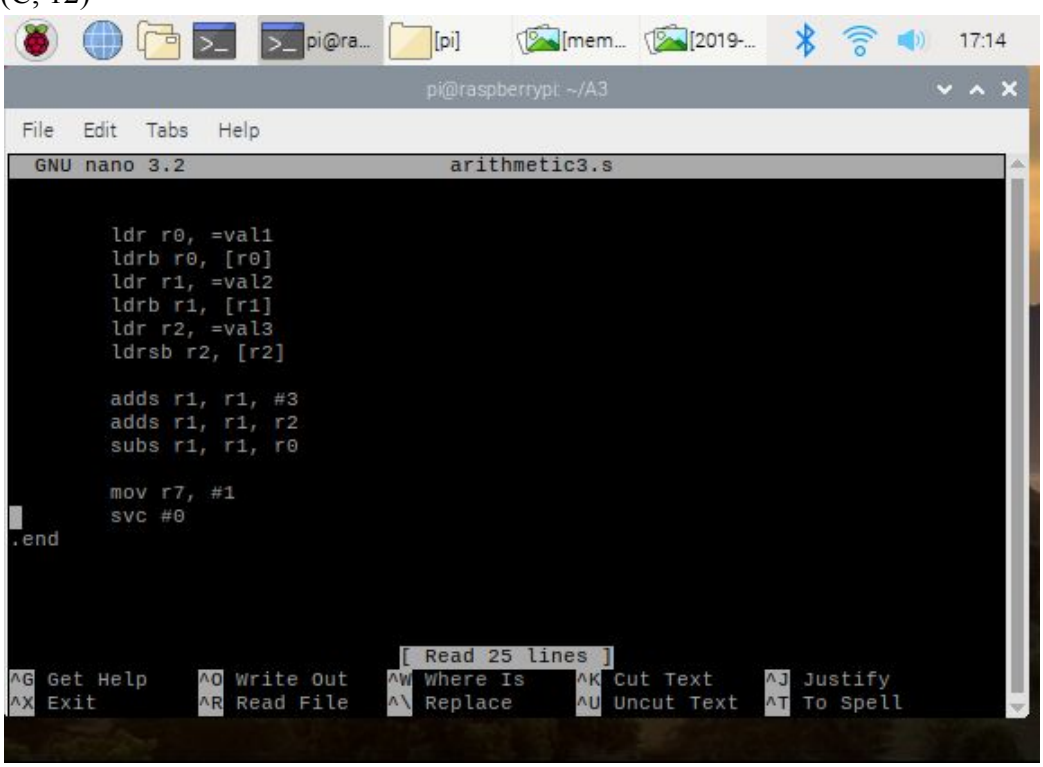
Registers for **val1**, **val2**, and **val3** before arithmetic calculations.

(C, 11)

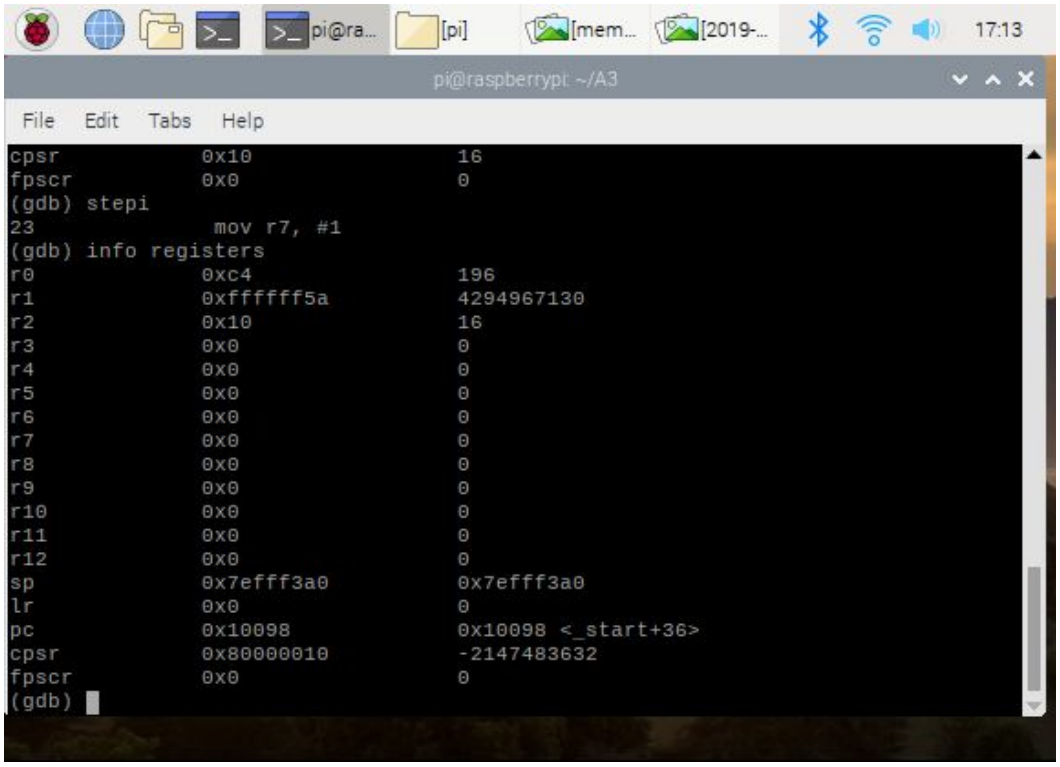


Results of registers after arithmetic instructions, but un-updated cpsr.

(C, 12)



Supplemented the -s suffix to each arithmetic instruction in order to grant flag privilege.
(C, 13)



Arithmetic result in **r1**. The cpsr flag triggers its 31st bit, the negative flag.