Developing Soft and Parallel Programming Skills Using Project-Based Learning

Spring 2019

Group Name: ATLAS-SQUAD

<u>Team Members:</u>

T'Avvion Jones *Team Coordinator*

Jason Poston

Sai Rampally

Zeak Sims

Shili Guan

Planning and Scheduling

Assignee Name	Email	Task	Duration (hrs)	Dependency	Due Date	Note
T'Avvion Jones **(Coordinator)**	tjones172@student.gsu.edu	Task 1, 2, 3b, 4 Task 5: Compile the report Task 6: Prepare for Video	4-5 hrs	Submit all assigned work properly	03/08/19	Review entire project and submit Excellent: (100%)
Sai Rampally	srampally1@student.gsu.edu	Task 3a: Q 5-6 Task 5: Submit task 3 response to A3 Google Doc/ GitHub Task 6: Prepare for Video	2 hrs	Upload Video to YouTube	03/06/19	All tasks must be finished and send to me by 3/6/19. Excellent: (100%)
Shili Guan	sguan2@student.gsu.edu	Task 3a: Q 1-2 Task 5: Submit task 3 response to A3 Google Doc/ GitHub Task 6: Prepare for Video	2 hrs	None	03/06/19	All tasks must be finished and send to me by 3/6/19. Excellent: (100%)
Zeak Sims	zsims2@student.gsu.edu	Task 3a: Q 3-4 Task 5: Submit task 3 response to A3 Google Doc/ GitHub Task 6: Prepare for Video	2 hrs	None	3/06/19	All tasks must be finished and send to me by 3/6/19. Excellent: (100%)
Jason Poston	jposton1@student.gsu.edu	Task 3a: Q 7-8 Task 5: Submit task 3 response to A3 Google Doc/ GitHub Task 6: Prepare for Video	2 hrs	None	3/06/19	All tasks must be finished and send to me by 3/6/19. Excellent: (100%)

Parallel Programming Skills

1) Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization (in your own words)

Task: in parallel program task is one or more set of instructions of the program or program alike that can run at same time by multiple processors.

Pipelining: is a type of parallel computing that splits a task into many steps that allow the steps to be performed by different processor units, connected in series, that output is the input of the next element.

Shared memory: we can describe this in two point of view which are from the hardware and programing. First, by the hardware definition is that all the processors are connect on bus based usually access to the common physical memory. From the programming point is that for all the parallel task which all have the address and access same as the logical memory no matter where the physical memory is located.

Communication: is referred to the event of data exchange happens in parallel tasks.

Synchronization: the coordination of parallel task in real time that very often associated with communication, synchronization often not be able to continue until two tasks reach the same or equivalent point. As this feature of the synchronization, it can cause a execution time of a parallel application's wall clock increase as tasks wait for one another.

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

They can be classified as two independent dimensions of instruction stream and data stream,
and for each of them can be only have of the single or multiple as: **Single Instruction Stream, Single Data Stream (SISD)** or **Single Instruction Stream, Multiple Data Stream (SIMD),** and **Multiple Instruction Stream, Single Data Stream (MISD)** or **Multiple Instruction Stream, Multiple Data Stream (MIMD).**

Single Instruction Stream, Single Data Stream (SISD): A serial(non-parallel) computer that during any one clock cycle it has only one instruction stream acted by on the CPU and only one data stream is being used as input. You can find this commonly in oldest computer such as: minicomputer, workstations and single processor or core PCs.

Single Instruction Stream, Multiple Data Stream (SIMD): One of the parallel computers. In this type of parallel computer, the same instruction can be executed at any given clock cycle by all processing units. Each of the unit can operate on a different data. Best suited for specialized problems characterized by a high degree of regularity. It has two varieties as

Processor Array and Vector Pipelines. It can be found on most the modern computer such as CPUs employ SIMD instructions and execution units.

Multiple Instruction Stream, Single Data Stream (MISD): One of the parallel computers. In this type of computer, each processing unit operates on the data independently through separate instruction streams. Multiple processing unites is fed in by a single data stream. Only rare of the actual example of this computer has ever existed.

Multiple Instruction Stream, Multiple Data Stream (MIMD): Another type of the parallel computers. A different instruction stream or data stream maybe be working by every processor. Many of them also include SIMD execution sub-components.

3.) What are the Parallel Programming Models

The parallel programming models are

- Shared Memory
- Threads
- Distributed Memory/ message passing
- Data Parallel
- Hybrid
- Single Program Multiple Data
- 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 model on a DISTRIBUTED memory machine:

Machine memory can be spread across physically across multiple machines, but can be virtually shared and become one giant memory.

DISTRIBUTED memory model on a SHARED memory machine:

Every task has an address to access the machine memory. Task can access memory globally, and multiple task can access the same memory.

Parallel Programming Models(shared memory model without threads):

Tasks share a commons address to memory space. This means the task are read and written one at a time. This is the simplest parallel programming model and allows each task to have equal access to shared memory. This can also present a disadvantage because it can get difficult to understand and manage data locality.

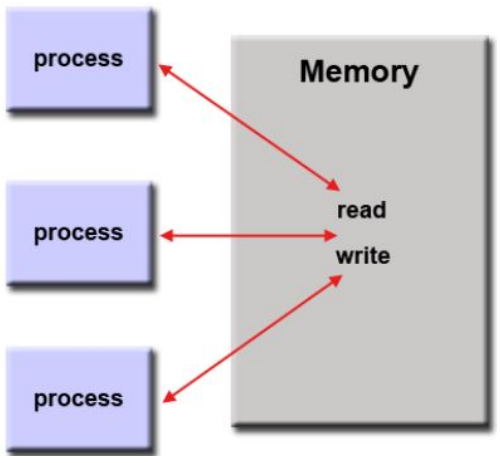
Parallel Programming Models(thread model):

This is a shared memory architecture that allows big task to be broken down into multiple smaller task that execute concurrently. It does this this by creating a number of threads, and each thread has their own local data, but has access to the entire resources. This architecture requires synchronization, so threads can communicate through global data.

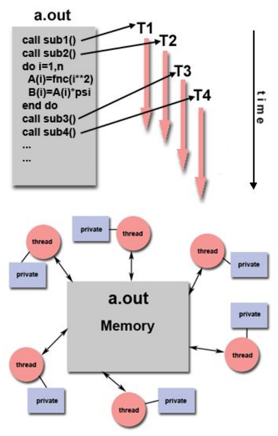
OpenMP is an industry standard that uses this architecture. It is used because it is portable? Multiplatform and provides ease of use for incremental parallelism.

<u>5.) Compare Shared Memory Model with Threads Model? (in your own words and show pictures)</u>

The shared memory model is the simplest parallel programming model. The processes and tasks share a common address space, so they can read and write to it. Various mechanisms like locks & semaphores are often used to control access to the shared memory, so they can prevent race conditions and deadlocks. One advantage of shared memory model is that all processes see and have equal access to the shared memory. One disadvantage of shared memory model is that in terms of performance, it often becomes more and more difficult to understand and manage data locality.



The threads model can have a single heavy weight process which has multiple light weight execution paths. This model can be described as a subroutine. A thread can execute and process the tasks at the same time as other threads which process different tasks. In this model, each thread not only has local data, but also shares entire resources of the main program a.out. The threads in the threads model communicate with each other through global memory, but to do that it requires synchronization, constructs so that more than one thread is not updating the same address at any time.



```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```

6.) What is Parallel Programming?

Parallel Programming often defined as using multiple processors simultaneously to complete the execution of an application. It solves problems that don't fit on a single CPU and solves problems that can't be solved in a reasonable amount of time. This helps in performing large computations by dividing the workload between more than one processor, so that all of them work through the computation at the same time.

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

The system on chip is when the Central Processing Unit (CPU) is not divided into parts, such as Random Access Memory (RAM) or Graphics Processing Unit (GPU), USB controller, power management circuits and wireless radios. They are composed on one single component called System on Chip (SoC). The Raspberry PI does take advantage of utilizing a SoC, so basically the entire computer is on this one single chip.

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

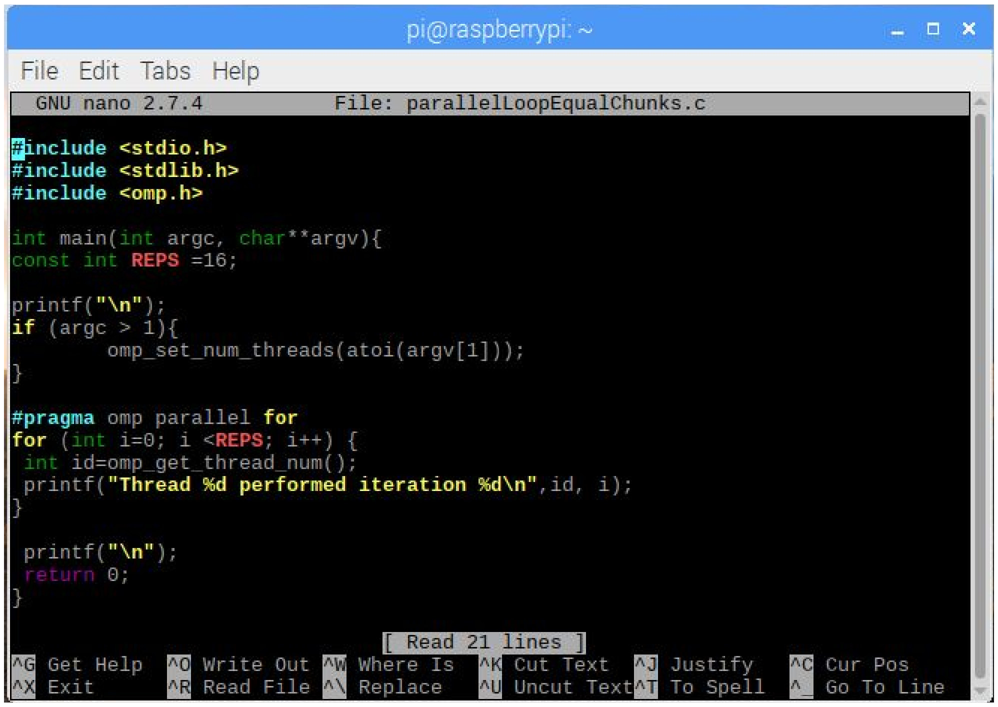
SoC's are becoming more and more popular, and one of the biggest advantage of using an SoC rather than the 'traditional' separate CPU, GPU and RAM components is its size. SoC's usually being no bigger than most CPU, and packing all the other components in along with it. Another advantage is its consumption of power. This becomes a big advantage in the mobile computing world, as battery life are big concerns. The cost of building SoC's is another advantage over traditional separate components, as you can cut down on wires, and physical chips making it cheaper to produce.

Parallel Programming Basics (Task 3b):

This week, we are observing patterns in data decomposition with the use of parallel loops. Essentially, we are learning the ways in which we can divide the amount of work for loops have to do onto multiple threads.

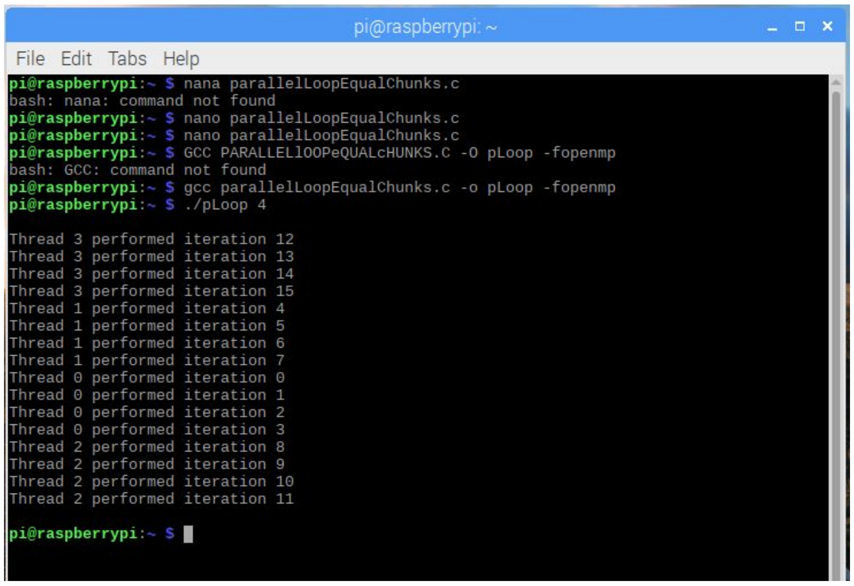
We start by looking at the following code:

At line 10 (#pragma omp parallel for), we initiate OpenMP's default parallel for loop that allows the for loop to be divided into equal parts and distributed to threads for execution of iterations. Each thread executes different iterations.



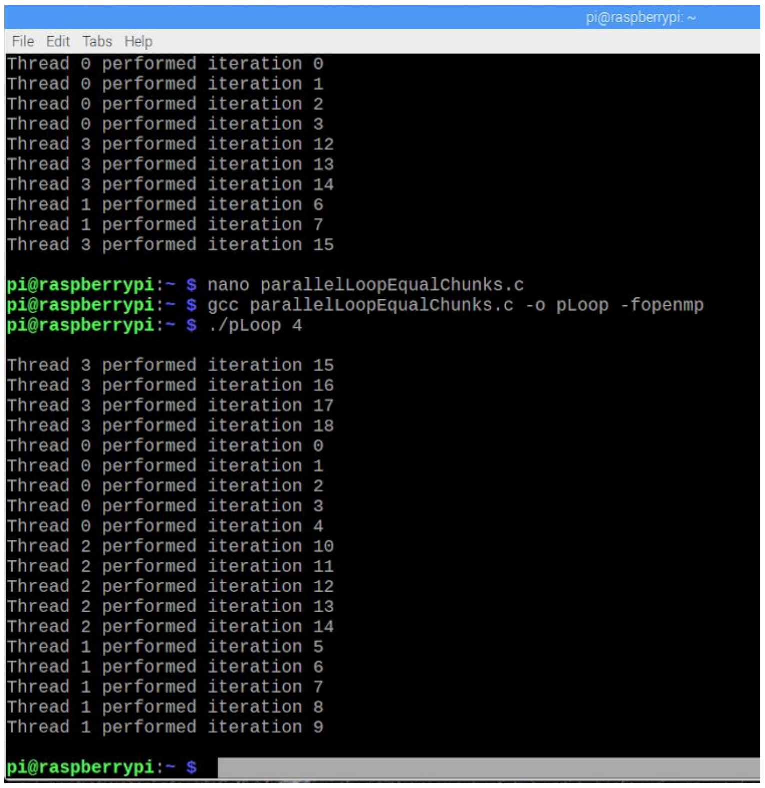
```
  GNU nano 2.7.4              File: parallelLoopEqualChunks.c

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

int main(int argc, char**argv){
const int REPS =16;

printf("\n");
if (argc > 1){
        omp_set_num_threads(atoi(argv[1]));
}

#pragma omp parallel for
for (int i=0; i <REPS; i++) {
 int id=omp_get_thread_num();
 printf("Thread %d performed iteration %d\n",id, i);
}

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

                               [ Read 21 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
```

After writing and saving the code we had to make an executable file called pLoop to run the code. The code that says "./pLoop 4" has the number 4 at the end which specifies how many threads to fork. In the case above, we have 16 different iterations that were evenly distributed onto 4 threads. The code below specifies which thread performed which iteration.



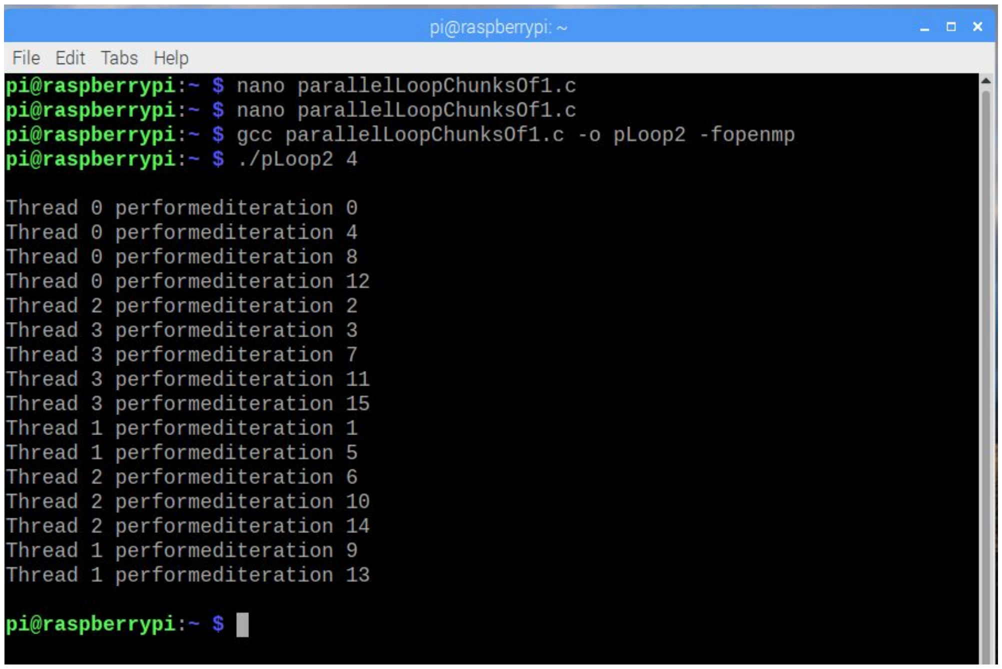
```
pi@raspberrypi:~ $ nana parallelLoopEqualChunks.c
bash: nana: command not found
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ GCC PARALLELlOOPeQUALcHUNKS.C -O pLoop -fopenmp
bash: GCC: command not found
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 3 performed iteration 15
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11

pi@raspberrypi:~ $ 
```

We altered the number of iterations to see what would happen if we have a number that doesn't evenly divide into the 4 threads. We have 19 iterations and still 4 threads. It looks as though it evenly distributed what it could and stopped when there was no iterations left to be done. After multiple changes in the number of iterations, there seems to be no distinct pattern that show how it decides to split the threads.

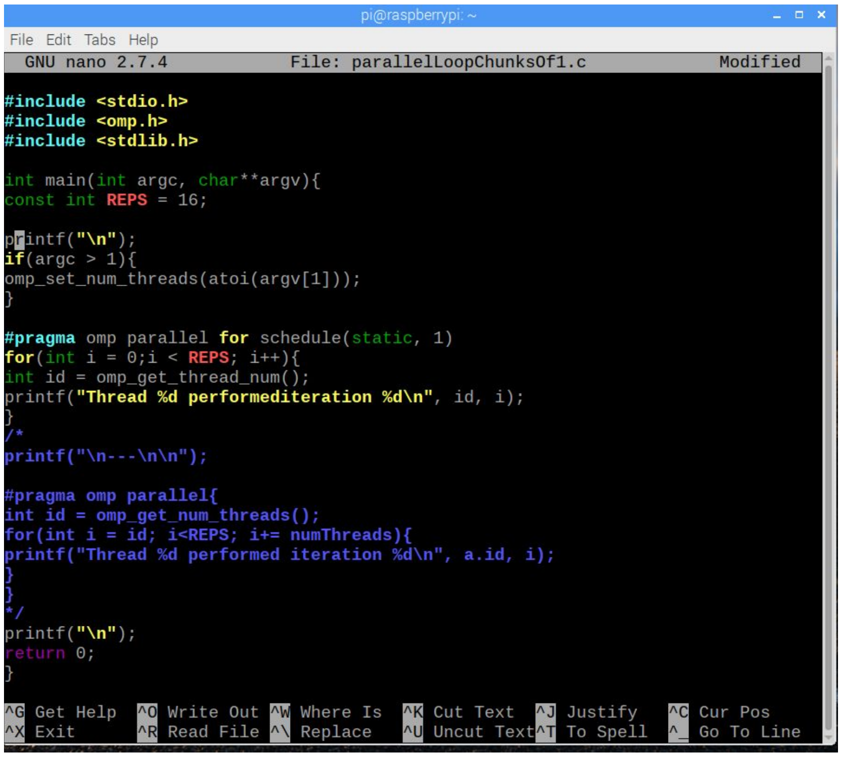


```
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 3 performed iteration 15

pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./pLoop 4

Thread 3 performed iteration 15
Thread 3 performed iteration 16
Thread 3 performed iteration 17
Thread 3 performed iteration 18
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 2 performed iteration 12
Thread 2 performed iteration 13
Thread 2 performed iteration 14
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 1 performed iteration 8
Thread 1 performed iteration 9

pi@raspberrypi:~ $ 
```

When running and compiling the second code below it outputs the threads and their iterations as before. The difference in this output and the preceding code's output is that here we can see that the iterations are in chronological order per thread and every iteration goes to 1 thread and looping through the threads in order. To better explain, iteration 0 goes to Thread 0, iteration 1, goes to thread 1, iteration 2> thread 2, iteration 3 > thread 3… then loops around the threads per iteration until there's no more.



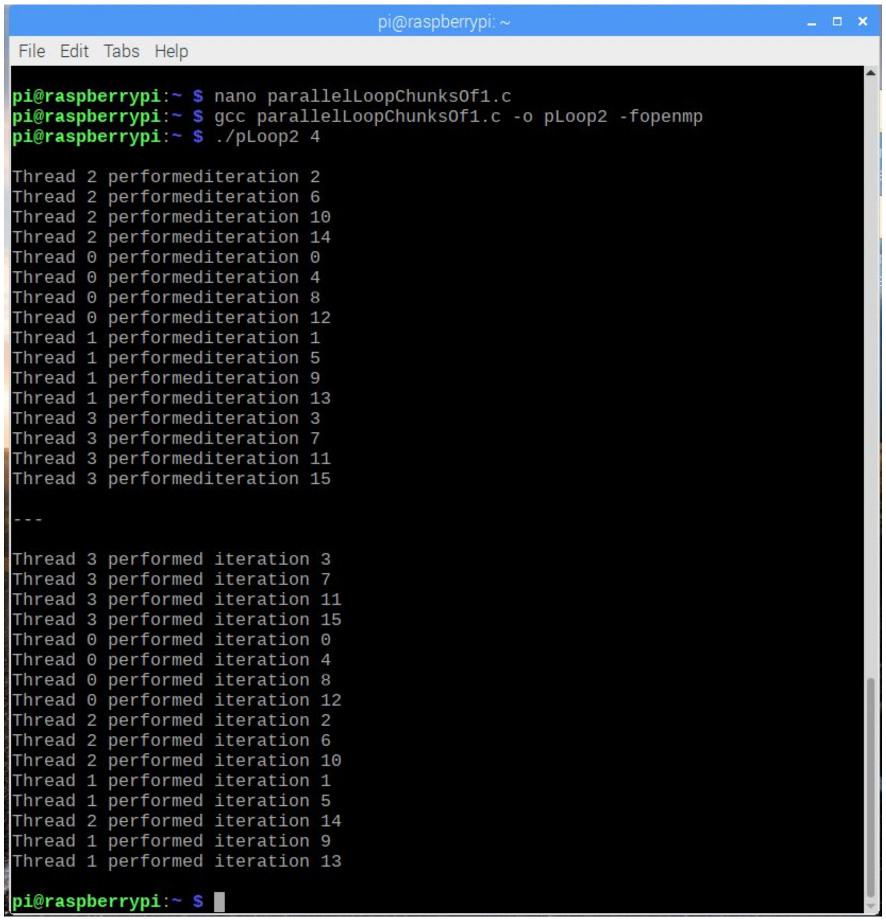
```
pi@raspberrypi:~ $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~ $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4

Thread 0 performediteration 0
Thread 0 performediteration 4
Thread 0 performediteration 8
Thread 0 performediteration 12
Thread 2 performediteration 2
Thread 3 performediteration 3
Thread 3 performediteration 7
Thread 3 performediteration 11
Thread 3 performediteration 15
Thread 1 performediteration 1
Thread 1 performediteration 5
Thread 2 performediteration 6
Thread 2 performediteration 10
Thread 2 performediteration 14
Thread 1 performediteration 9
Thread 1 performediteration 13

pi@raspberrypi:~ $
```



```
  GNU nano 2.7.4              File: parallelLoopChunksOf1.c              Modified

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

int main(int argc, char**argv){
const int REPS = 16;

printf("\n");
if(argc > 1){
omp_set_num_threads(atoi(argv[1]));
}

#pragma omp parallel for schedule(static, 1)
for(int i = 0;i < REPS; i++){
int id = omp_get_thread_num();
printf("Thread %d performediteration %d\n", id, i);
}
/*
printf("\n---\n\n");

#pragma omp parallel{
int id = omp_get_num_threads();
for(int i = id; i<REPS; i+= numThreads){
printf("Thread %d performed iteration %d\n", a.id, i);
}
}
*/
printf("\n");
return 0;
}

^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
```

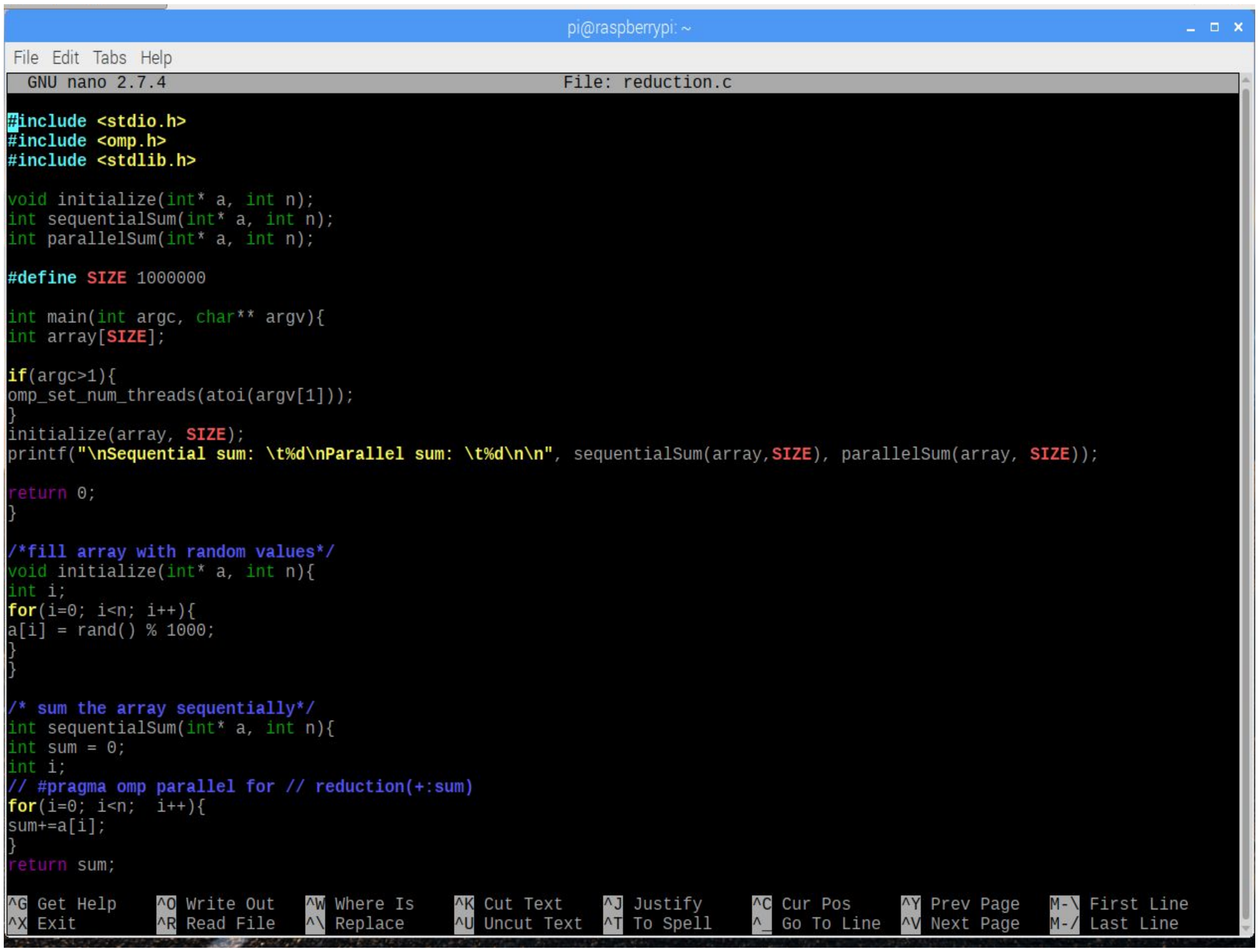
After uncommenting a section of the code we see the same exact code being output. In this section of the code we use static scheduling which schedules each thread to do a single iteration of the for loop in a pattern.

In the code we can change the static to dynamic for dynamic scheduling. Dynamic scheduling is used when we are unsure about the dependencies and how long they will take and so we manage them at compile time.



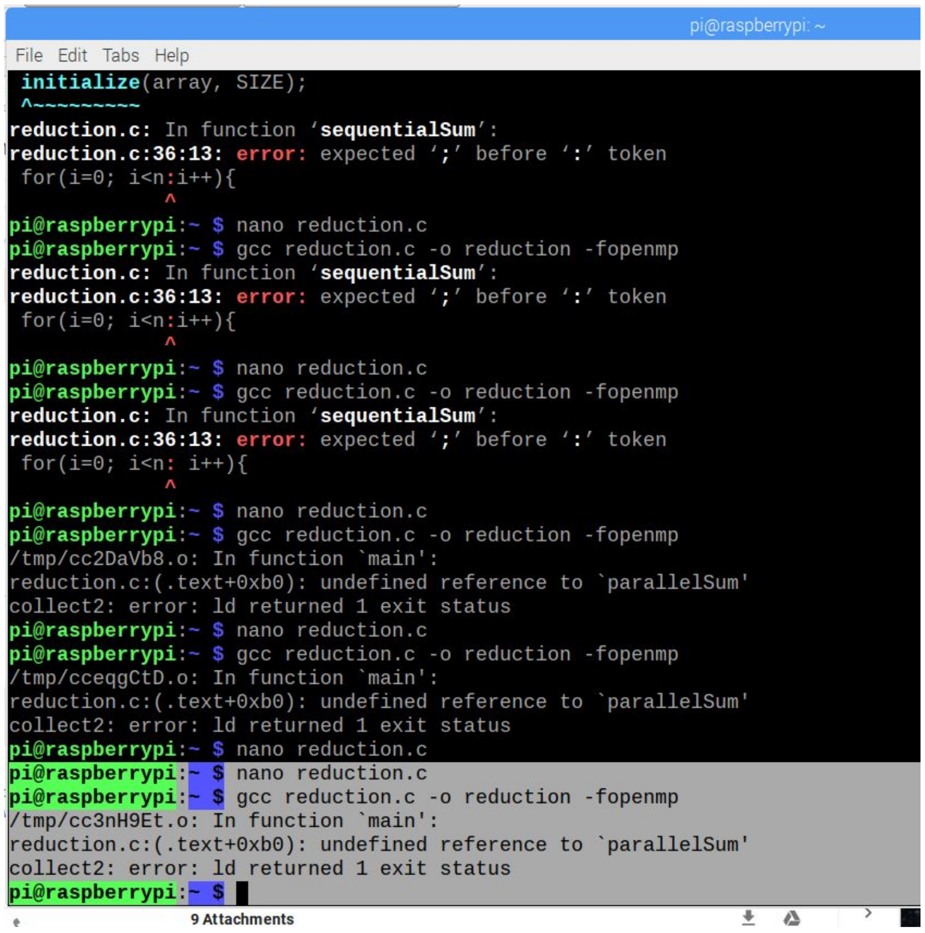
```
pi@raspberrypi:~ $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~ $ gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
pi@raspberrypi:~ $ ./pLoop2 4

Thread 2 performediteration 2
Thread 2 performediteration 6
Thread 2 performediteration 10
Thread 2 performediteration 14
Thread 0 performediteration 0
Thread 0 performediteration 4
Thread 0 performediteration 8
Thread 0 performediteration 12
Thread 1 performediteration 1
Thread 1 performediteration 5
Thread 1 performediteration 9
Thread 1 performediteration 13
Thread 3 performediteration 3
Thread 3 performediteration 7
Thread 3 performediteration 11
Thread 3 performediteration 15

---

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 2 performed iteration 14
Thread 1 performed iteration 9
Thread 1 performed iteration 13

pi@raspberrypi:~ $
```

Next, we have a program called Reduction. This code produces an array of random integers and and adds them up sequentially and with the use of multiple threads. To add up the integers in the array using multiple threads we use the reduction(+:sum) clause on the variable sum. The “+” means that the variable sum will be determined by adding the integers together.



```
  GNU nano 2.7.4                                        File: reduction.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;
}

/*fill array with random values*/
void initialize(int* a, int n){
int i;
for(i=0; i<n; i++){
a[i] = rand() % 1000;
}
}

/* sum the array sequentially*/
int sequentialSum(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;

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos     ^Y Prev Page   M-\ First Line
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell    ^_ Go To Line  ^V Next Page   M-/ Last Line
```

When saving the file and attempting to compile it it gives an unknown output and has a problem with the parallelSum reference. (Image of error below)

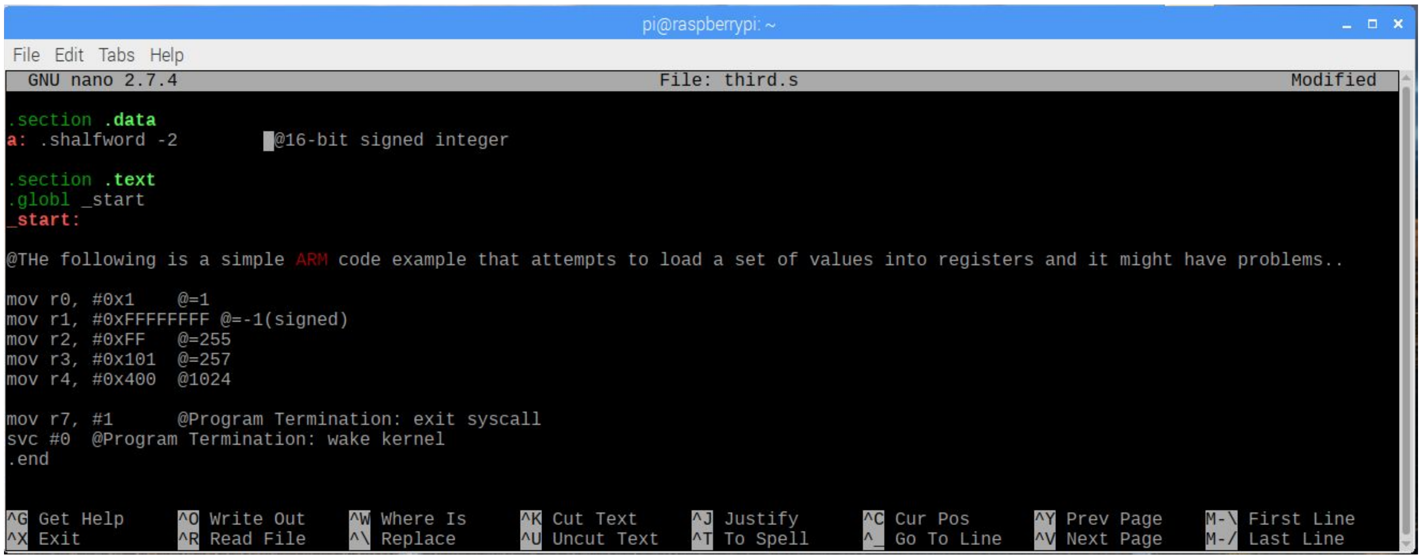


```
 initialize(array, SIZE);
 ^~~~~~~~~~
reduction.c: In function ‘sequentialSum’:
reduction.c:36:13: error: expected ‘;’ before ‘:’ token
 for(i=0; i<n:i++){
             ^
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
reduction.c: In function ‘sequentialSum’:
reduction.c:36:13: error: expected ‘;’ before ‘:’ token
 for(i=0; i<n:i++){
             ^
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
reduction.c: In function ‘sequentialSum’:
reduction.c:36:13: error: expected ‘;’ before ‘:’ token
 for(i=0; i<n: i++){
             ^
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
/tmp/cc2DaVb8.o: In function `main':
reduction.c:(.text+0xb0): undefined reference to `parallelSum'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
/tmp/cceqgCtD.o: In function `main':
reduction.c:(.text+0xb0): undefined reference to `parallelSum'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ nano reduction.c
pi@raspberrypi:~ $ gcc reduction.c -o reduction -fopenmp
/tmp/cc3nH9Et.o: In function `main':
reduction.c:(.text+0xb0): undefined reference to `parallelSum'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~ $ 
```

Being that we’re having trouble here with the ParallelSum reference, we’re unable to see the actual output. We can conclude that in order to add integers with the use of multiple threads, we must use the reduction clause to help the loops communicate it with each other to get the proper sum.

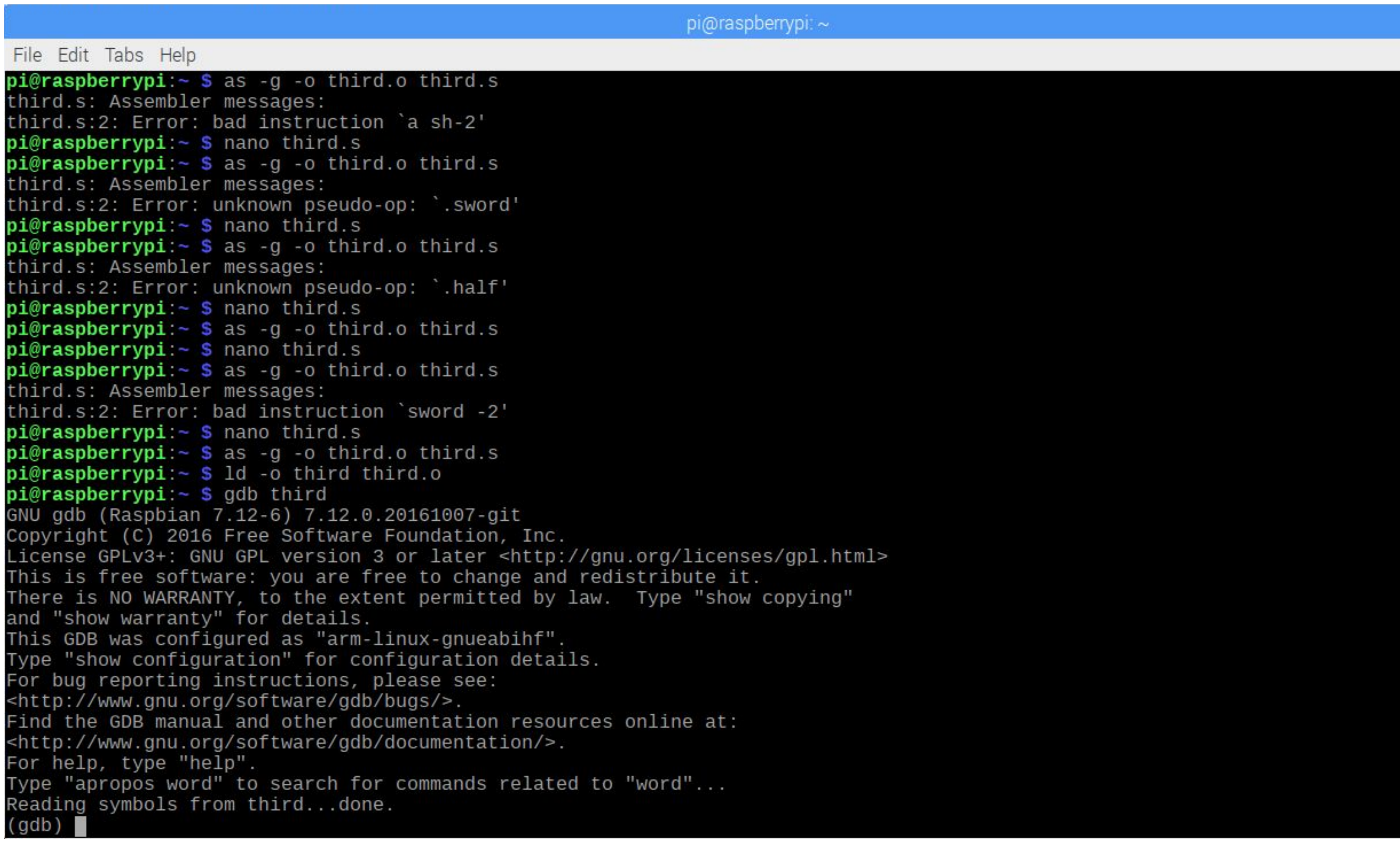
ARM Assembly Programming

We first start by typing in the following program:



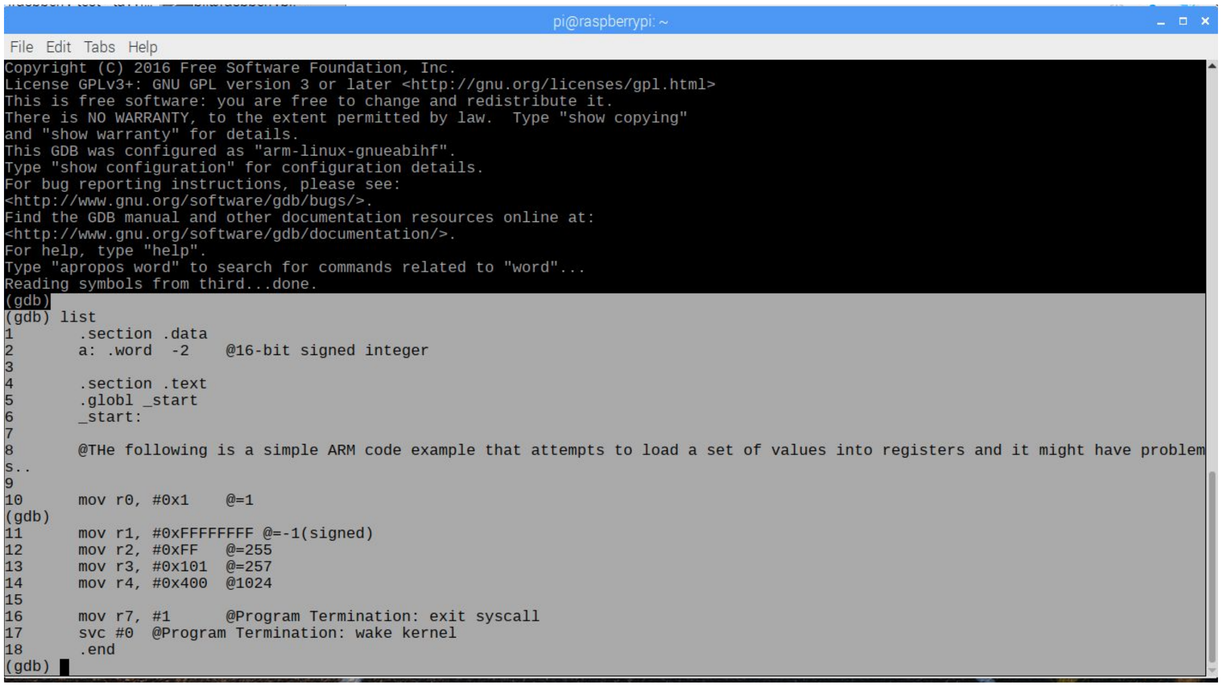
```
  GNU nano 2.7.4                                       File: third.s                                          Modified

.section .data
a: .shalfword -2        @16-bit signed integer

.section .text
.globl _start
_start:

@THe following is a simple ARM code example that attempts to load a set of values into registers and it might have problems..

mov r0, #0x1    @=1
mov r1, #0xFFFFFFFF @=-1(signed)
mov r2, #0xFF   @=255
mov r3, #0x101  @=257
mov r4, #0x400  @1024

mov r7, #1      @Program Termination: exit syscall
svc #0  @Program Termination: wake kernel
.end

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos     ^Y Prev Page   M-\ First Line
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Spell    ^_ Go To Line  ^V Next Page   M-/ Last Line
```

After saving the file, we need to assemble it. Being that there is no output we use the GDB, which is a GNU Debugger. The GDB will be used to check the values via the registers and memory. When assembling the program, we add a flag, "-g", to help the debugger link the machine code to the source code line by line. When doing so we get an error about the pseudo ".shalfword" which was for the 16-bit signed integer. We get the error because this isn't the correct pseudo for a 16-bit signed integer. The correct way to declare the integer is to use ".word". We use ".word" regardless of whether the value is signed or not because ARM uses two's complement to convert negative numbers. After, we write the linker command and then launch the GDB. When launching the debugger it then displays the following lines:



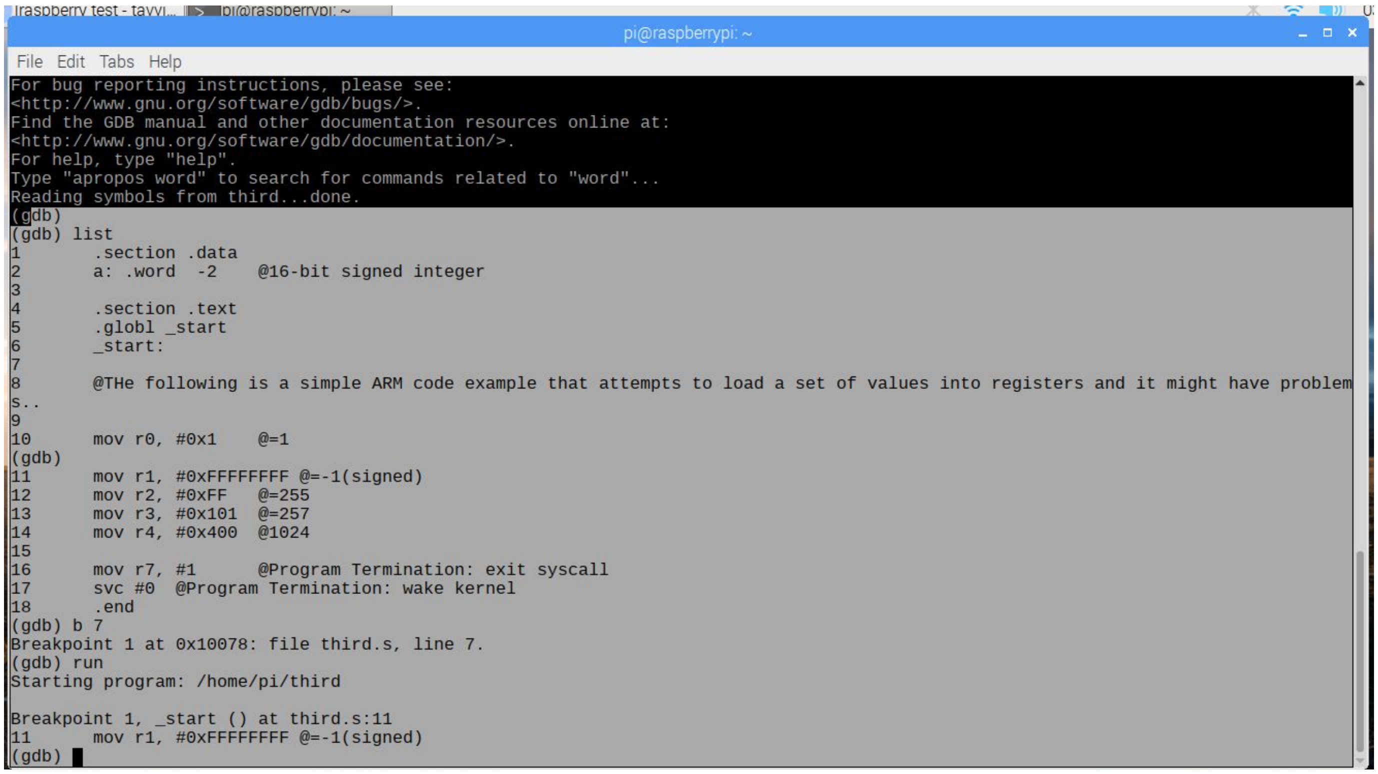
```
pi@raspberrypi:~ $ as -g -o third.o third.s
third.s: Assembler messages:
third.s:2: Error: bad instruction `a sh-2'
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
third.s: Assembler messages:
third.s:2: Error: unknown pseudo-op: `.sword'
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
third.s: Assembler messages:
third.s:2: Error: unknown pseudo-op: `.half'
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
third.s: Assembler messages:
third.s:2: Error: bad instruction `sword -2'
pi@raspberrypi:~ $ nano third.s
pi@raspberrypi:~ $ as -g -o third.o third.s
pi@raspberrypi:~ $ ld -o third third.o
pi@raspberrypi:~ $ gdb third
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from third...done.
(gdb) 
```

Next, we list the source code by typing (gdb) list. This will display 10 lines of source code.



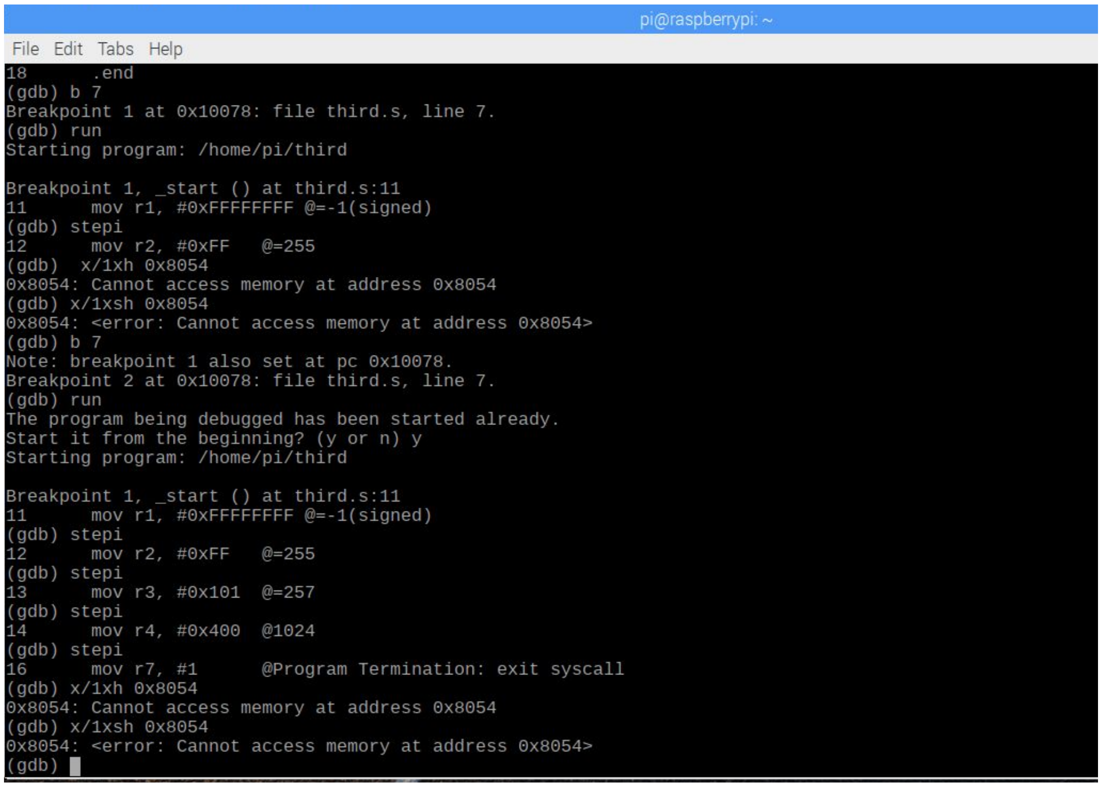
```
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from third...done.
(gdb)
(gdb) list
1       .section .data
2       a: .word  -2    @16-bit signed integer
3
4       .section .text
5       .globl _start
6       _start:
7
8       @THe following is a simple ARM code example that attempts to load a set of values into registers and it might have problem
s..
9
10      mov r0, #0x1    @=1
(gdb)
11      mov r1, #0xFFFFFFFF @=-1(signed)
12      mov r2, #0xFF   @=255
13      mov r3, #0x101  @=257
14      mov r4, #0x400  @1024
15
16      mov r7, #1      @Program Termination: exit syscall
17      svc #0  @Program Termination: wake kernel
18      .end
(gdb) 
```

Next we create a breakpoint by typing “(gdb) b 7”. This means that we are creating a breakpoint at line 7. Afterwards we type “(gdb) run” to run the program from the given breakpoint. This then displays the line of code in which follows the breakpoint.



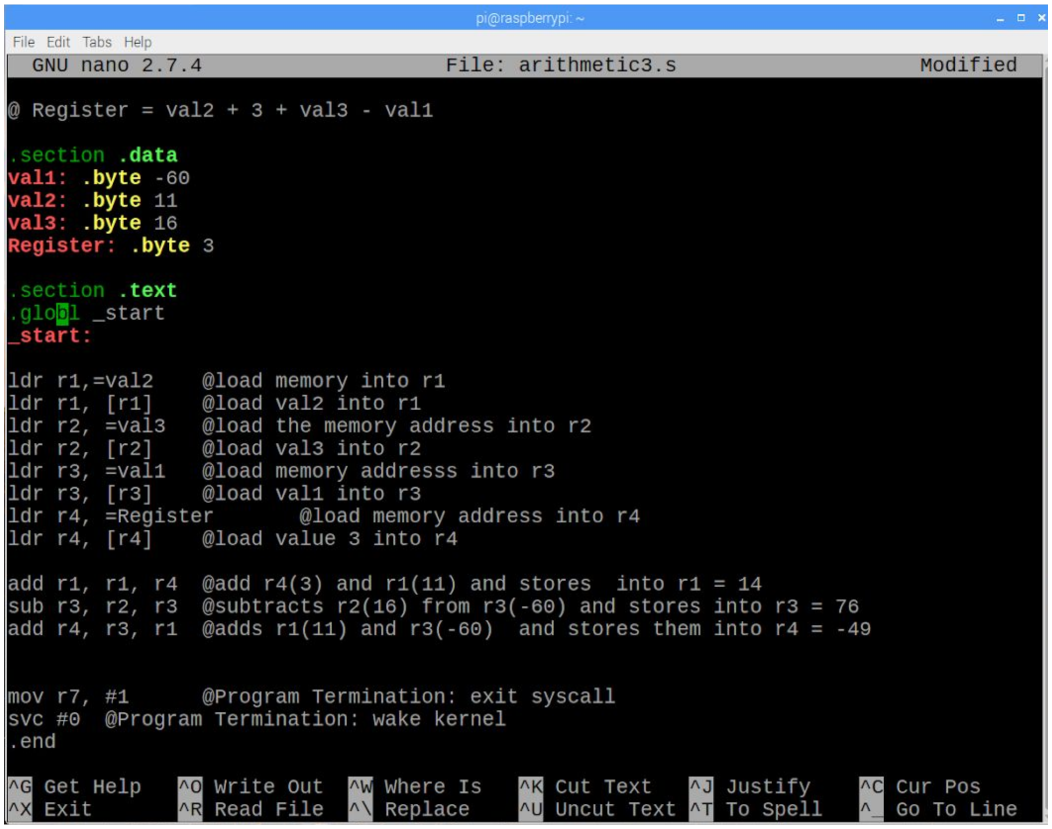
```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from third...done.
(gdb)
(gdb) list
1       .section .data
2       a: .word  -2    @16-bit signed integer
3
4       .section .text
5       .globl _start
6       _start:
7
8       @THe following is a simple ARM code example that attempts to load a set of values into registers and it might have problem
s..
9
10      mov r0, #0x1    @=1
(gdb)
11      mov r1, #0xFFFFFFFF @=-1(signed)
12      mov r2, #0xFF   @=255
13      mov r3, #0x101  @=257
14      mov r4, #0x400  @1024
15
16      mov r7, #1      @Program Termination: exit syscall
17      svc #0  @Program Termination: wake kernel
18      .end
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 7.
(gdb) run
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:11
11      mov r1, #0xFFFFFFFF @=-1(signed)
(gdb)
```

We use “(gdb) stepi” to step to the next line one by one. In order to examine the memory we must type “x” followed by the options that include the number of items, it’s format, and size, as well as the starting address. First, we will try to type in “(gdb) x/1xh (enter any memory address used)”. This means that we will be accessing the memory given a number of items in the format of hexadecimal and is of size halfword or signed halfword.



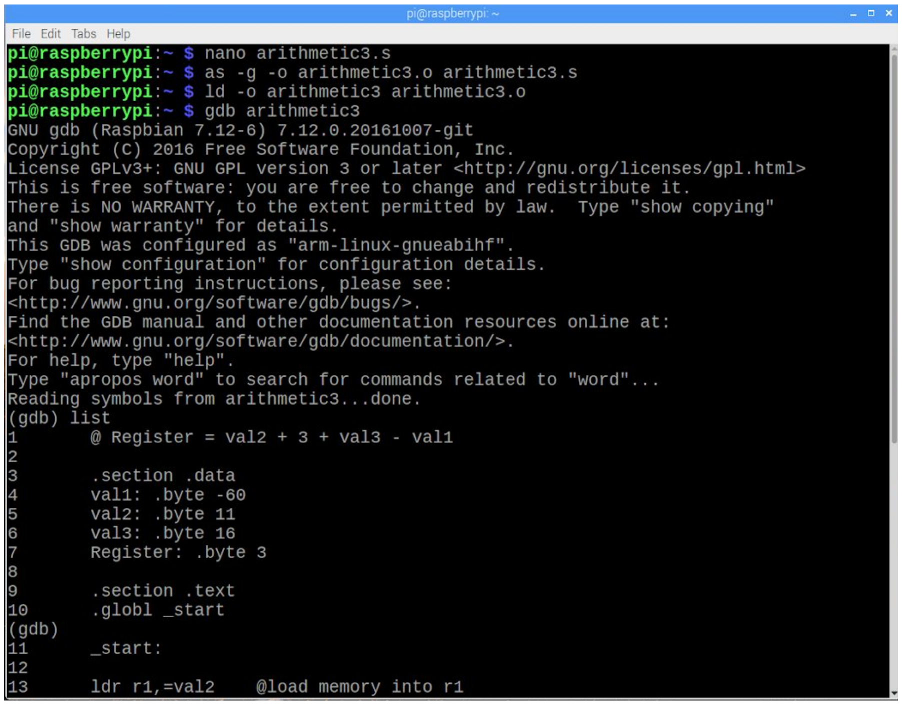
```
18      .end
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 7.
(gdb) run
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:11
11      mov r1, #0xFFFFFFFF @=-1(signed)
(gdb) stepi
12      mov r2, #0xFF   @=255
(gdb)  x/1xh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1xsh 0x8054
0x8054: <error: Cannot access memory at address 0x8054>
(gdb) b 7
Note: breakpoint 1 also set at pc 0x10078.
Breakpoint 2 at 0x10078: file third.s, line 7.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pi/third

Breakpoint 1, _start () at third.s:11
11      mov r1, #0xFFFFFFFF @=-1(signed)
(gdb) stepi
12      mov r2, #0xFF   @=255
(gdb) stepi
13      mov r3, #0x101  @=257
(gdb) stepi
14      mov r4, #0x400  @1024
(gdb) stepi
16      mov r7, #1      @Program Termination: exit syscall
(gdb) x/1xh 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/1xsh 0x8054
0x8054: <error: Cannot access memory at address 0x8054>
(gdb) 
```

Part 2.
For the Second Part to the ARM Assembly Programming, we input the following arithmetic program:



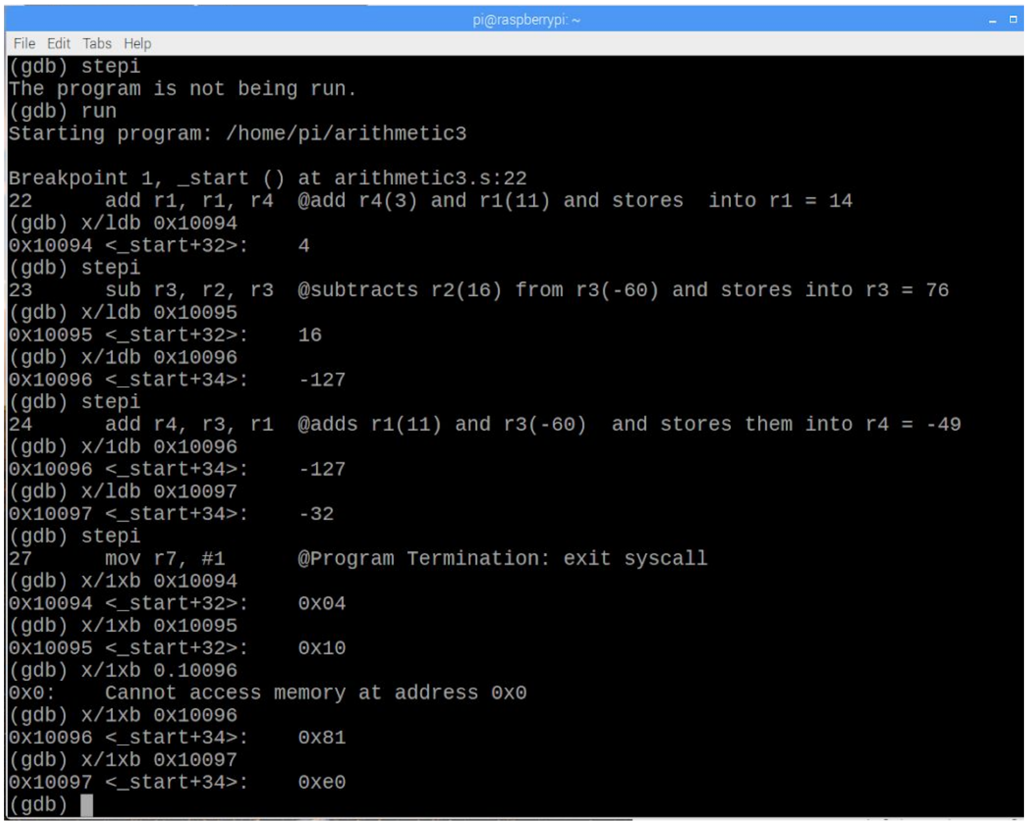
```
 GNU nano 2.7.4                  File: arithmetic3.s                   Modified

@ Register = val2 + 3 + val3 - val1

.section .data
val1: .byte -60
val2: .byte 11
val3: .byte 16
Register: .byte 3

.section .text
.globl _start
_start:

ldr r1,=val2    @load memory into r1
ldr r1, [r1]    @load val2 into r1
ldr r2, =val3   @load the memory address into r2
ldr r2, [r2]    @load val3 into r2
ldr r3, =val1   @load memory address into r3
ldr r3, [r3]    @load val1 into r3
ldr r4, =Register       @load memory address into r4
ldr r4, [r4]    @load value 3 into r4

add r1, r1, r4  @add r4(3) and r1(11) and stores  into r1 = 14
sub r3, r2, r3  @subtracts r2(16) from r3(-60) and stores into r3 = 76
add r4, r3, r1  @adds r1(11) and r3(-60)  and stores them into r4 = -49


mov r7, #1      @Program Termination: exit syscall
svc #0  @Program Termination: wake kernel
.end

^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
```

We then used the debugger to verify the results in the memory and register. We debugged by stepping through each line using “stepi”. After stepping through the code we began to look at the memory based off of their address.



```
pi@raspberrypi:~ $ nano arithmetic3.s
pi@raspberrypi:~ $ as -g -o arithmetic3.o arithmetic3.s
pi@raspberrypi:~ $ ld -o arithmetic3 arithmetic3.o
pi@raspberrypi:~ $ gdb arithmetic3
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from arithmetic3...done.
(gdb) list
1       @ Register = val2 + 3 + val3 - val1
2
3       .section .data
4       val1: .byte -60
5       val2: .byte 11
6       val3: .byte 16
7       Register: .byte 3
8
9       .section .text
10      .globl _start
(gdb)
11      _start:
12
13      ldr r1,=val2    @load memory into r1
```

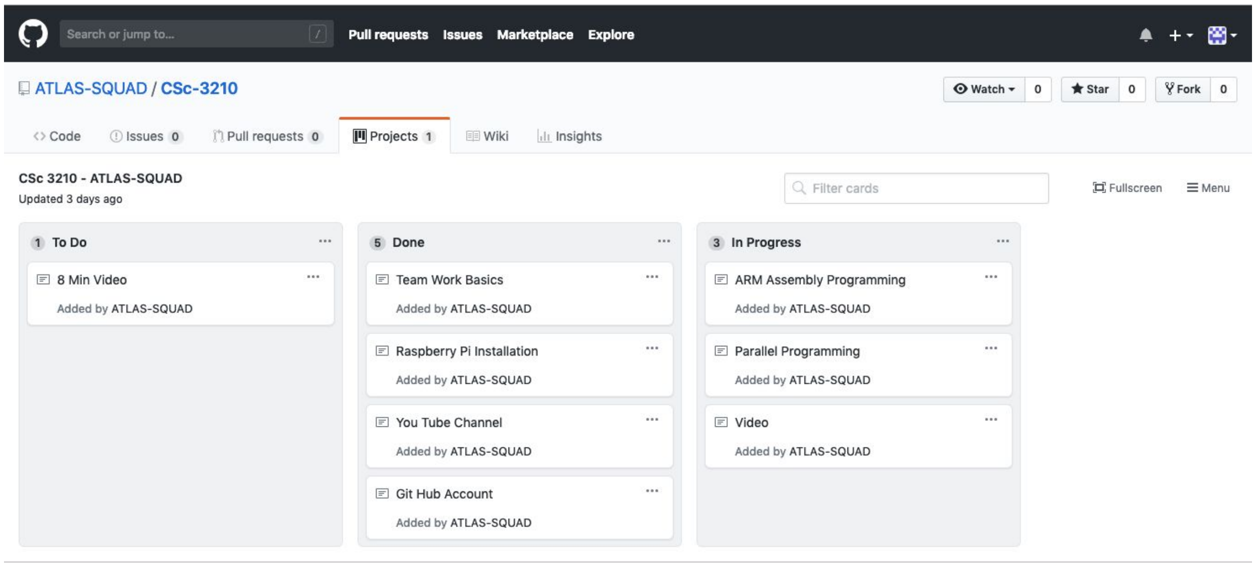
At the bottom of the image below we look at the memory in hexadecimal and if we convert the number to decimal we can determine the types of flames they raise. For example,when looking at memory 0x10096 it gives us 0x81. When converting 81h to binary, we get 01000001. The most significant number is 0 which means it is a positive number, therefore, the signed flag is 0. The zero flag is 0 because the result is not 0.



```
(gdb) stepi
The program is not being run.
(gdb) run
Starting program: /home/pi/arithmetic3

Breakpoint 1, _start () at arithmetic3.s:22
22        add r1, r1, r4  @add r4(3) and r1(11) and stores  into r1 = 14
(gdb) x/ldb 0x10094
0x10094 <_start+32>:    4
(gdb) stepi
23        sub r3, r2, r3  @subtracts r2(16) from r3(-60) and stores into r3 = 76
(gdb) x/ldb 0x10095
0x10095 <_start+32>:    16
(gdb) x/1db 0x10096
0x10096 <_start+34>:    -127
(gdb) stepi
24        add r4, r3, r1  @adds r1(11) and r3(-60)  and stores them into r4 = -49
(gdb) x/1db 0x10096
0x10096 <_start+34>:    -127
(gdb) x/ldb 0x10097
0x10097 <_start+34>:    -32
(gdb) stepi
27        mov r7, #1      @Program Termination: exit syscall
(gdb) x/1xb 0x10094
0x10094 <_start+32>:    0x04
(gdb) x/1xb 0x10095
0x10095 <_start+32>:    0x10
(gdb) x/1xb 0.10096
0x0:    Cannot access memory at address 0x0
(gdb) x/1xb 0x10096
0x10096 <_start+34>:    0x81
(gdb) x/1xb 0x10097
0x10097 <_start+34>:    0xe0
(gdb)
```

Appendix

Youtube Channel: https://www.youtube.com/watch?v=ITSZtSDwj0Y
Slack: https://atlas-squad.slack.com/messages/CFR6D9LHJ/
GitHub: https://github.com/ATLAS-SQUAD



CSc 3210 - ATLAS-SQUAD
Updated 3 days ago

1 To Do
- 8 Min Video — Added by ATLAS-SQUAD

5 Done
- Team Work Basics — Added by ATLAS-SQUAD
- Raspberry Pi Installation — Added by ATLAS-SQUAD
- You Tube Channel — Added by ATLAS-SQUAD
- Git Hub Account — Added by ATLAS-SQUAD

3 In Progress
- ARM Assembly Programming — Added by ATLAS-SQUAD
- Parallel Programming — Added by ATLAS-SQUAD
- Video — Added by ATLAS-SQUAD