

# **Developing Soft and Parallel Programming Skills Using Project-Based Learning**

Spring 2020

**Gitter-geeks**

Hyoungjun Lee

Michael Knight – Team Coordinator

Tek Acharya

Vedansi Parsana

Zoe Kosmicki

**Planning and Scheduling**

As Team Coordinator I was meet with challenges but overall with a great team my challenges weren't that difficult. We kept the same time as we had before to meet up to cover anything that needed to be covered and to record our video presentation. We used Slack a lot more this time around since we've gotten familiar with it. We came together and helped each other with any problem that we had with the assignment. It was challenging to assign tasks to everyone to be fair and not assign too much to any one person. Overall I'm proud of what we accomplished and I look forward with working more with my group.

**Hyoungjun Lee**

**Parallel Programing Skills:**

**Identifying the components on the raspberry PI B+**

HDMI port, USB port, Power port, SD card slot, Camera, display connecting slot, ethernet Controller, ethernet port, CPU/RAM

**How many cores does the Raspberry Pi's B+ CPU have**

it has quad-core, so 4 CPU

**List three main differences between X86 (CISC) and ARM Raspberry PI (RISC).Justify you answer and use your own words (do not copy and past)**

1. CISC has less register than RISC, also uses little endian when you store
2. CISC used for PC, or large capacity , because containing more feature instruction set, allow complex instructions to access into memory
3. RISC has simple instruction set, because main purpose is for small devices such as phone, tablets etc.

**What is the difference between sequential and parallel computation and identify the practical significance of each?**

Difference between sequential and parallel computation is sequential can handle only one instruction when parallel computation can handle multiple, by using parallel computation we can multi-tasking, practical significance of sequential is easy writing, and parallel is multi tasking

### **Identify the basic form of data and task parallelism in computational problems**

Data parallelism – same computation or one operation into multiple data items

Task parallelism – allow one task to one core, another to one core, multiple function or thread than data parallelism

### **Explain the differences between processes and threads**

when running the program, process does not share memory, thread is subset of process, threads share their common memory with the process, and same address

### **What is OpenMP and what is OpenMP pragmas?**

OpenMP – compiler which can do multithreading, makes user's task simpler, also reducing an errors

OpenMP pragmas – compiler that directive to programs into parallel computation, generate thread code

### **What applications benefit from multi-core(list four)?**

Benefits of using multi-core application are data server, CAD/CAM , multimedia application ,data server

### **Why Multicore? (why not single core, list four)**

Multicore can running multiple process, allow to operate multi tasking, execute faster than single core, in single core it could be heat up by speeding up , muticore manage this problem, process in short time , output faster

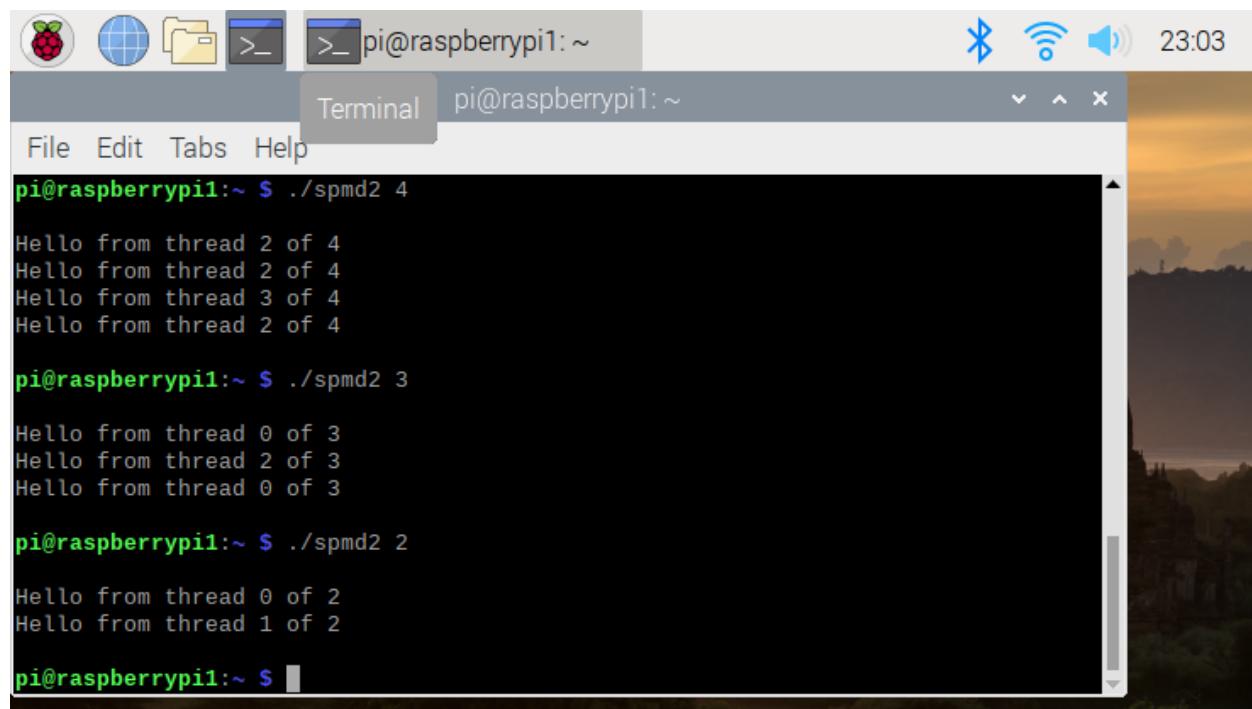
## B) Parallel Programming Basics

as following guide from top to bottom, I wrote first code and ran it, it did not run what I expected because id declared in main method, but there's more than 1 id in code

As I write code into PI and check the result, I can double check all the cores share same memory, cannot have multiple variables in same id. Have to declare other

So first part did not run correctly,

As shown in below

A screenshot of a Raspberry Pi desktop environment. At the top, there is a dock with icons for the home screen, network, file manager, and terminal. The terminal window is active and shows the command 'pi@raspberrypi1:~ \$ ./spmd2 4' followed by four lines of text: 'Hello from thread 2 of 4', 'Hello from thread 2 of 4', 'Hello from thread 3 of 4', and 'Hello from thread 2 of 4'. Below this, another command 'pi@raspberrypi1:~ \$ ./spmd2 3' is run, resulting in three lines of text: 'Hello from thread 0 of 3', 'Hello from thread 2 of 3', and 'Hello from thread 0 of 3'. Finally, 'pi@raspberrypi1:~ \$ ./spmd2 2' is run, producing two lines of text: 'Hello from thread 0 of 2' and 'Hello from thread 1 of 2'. The desktop background shows a sunset over a landscape.

```
pi@raspberrypi1:~ $ ./spmd2 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4

pi@raspberrypi1:~ $ ./spmd2 3
Hello from thread 0 of 3
Hello from thread 2 of 3
Hello from thread 0 of 3

pi@raspberrypi1:~ $ ./spmd2 2
Hello from thread 0 of 2
Hello from thread 1 of 2
```

Then change the id value inside, int id , I got correct answer what I expected

Second running result shown in below

The screenshot shows a terminal window titled 'pi@raspberrypi1: ~'. The window contains the following text output:

```
pi@raspberrypi1:~ $ ./spmd2 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

pi@raspberrypi1:~ $ ./spmd2 3
Hello from thread 0 of 3
Hello from thread 1 of 3
Hello from thread 2 of 3

pi@raspberrypi1:~ $ ./spmd2 2
Hello from thread 0 of 2
Hello from thread 1 of 2
```

The terminal window is part of a desktop environment, as evidenced by the window title bar and icons at the top. The desktop interface includes a file manager window showing files named '317\_736x4' and '320\_736x4'.

Now, working parallel

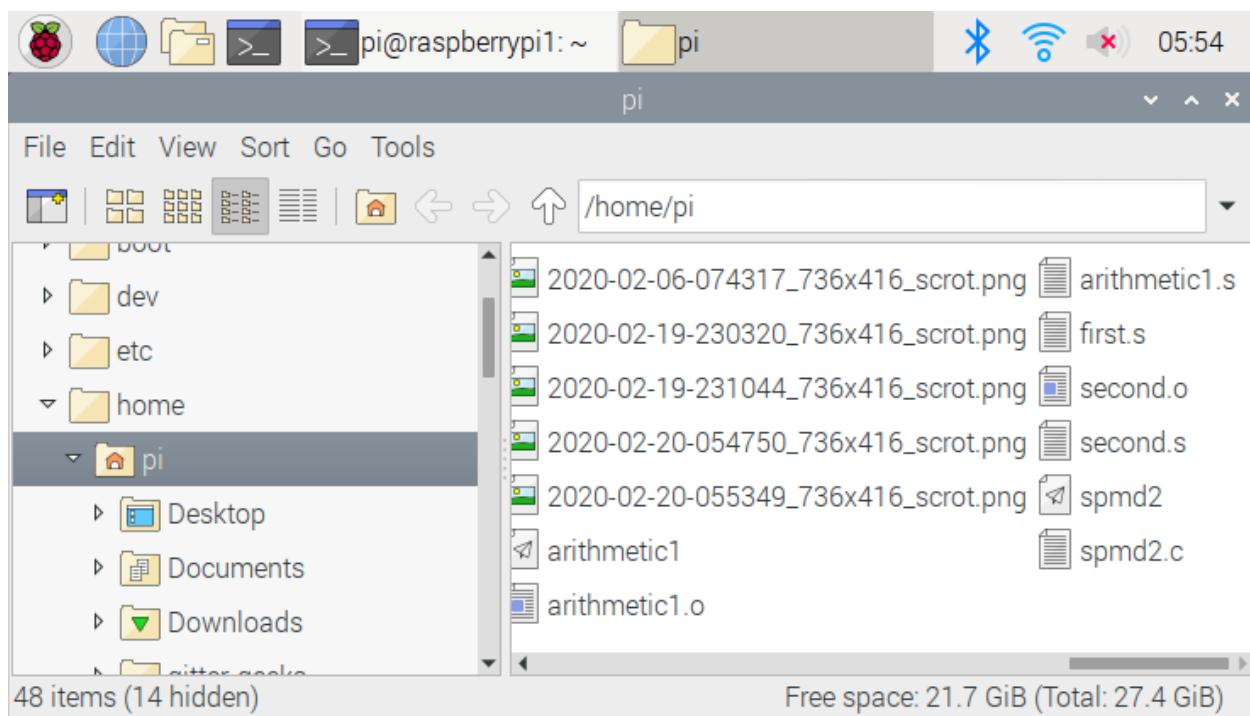
#### Task 4, ARM assembler in Raspberry PI

- a) Part1, Second Program

Here is program to represent  $c = a + b$ , written code is below

The screenshot shows a terminal window titled "pi@raspberrypi1: ~". The window contains the assembly code for a C program named "second.s". The code defines variables "a", "b", and "c" in the ".data" section and implements a simple addition routine in the ".text" section. The terminal interface includes a menu bar with File, Edit, Tabs, Help, and a toolbar with various keyboard shortcut icons.

```
@second program: c = a+b
.section.data
a:.word2 @32-bit variable a in memory
b:.word5 @32-bit variable b in memory
c:.word0 @32-bit variable c in memory
.section.text
.globl_start
_start:
    idr r1=a      @load the memory address of a into r1
    idr r1,[r1]    @load the value a into r1
    idr r2, =b     @load the memory address of b into r2
    idr r2, [r2]   @load the value b into r2
    add r1,r1,r2  @add r1 to r2 and store into r1
    idr r2, =c     @load the memory address of c into r2
```



I assembled file first, and create second.o file, and link this file to executable, but there was no output we can find, all the output should be inside registry that we can't see it but we can see by debugging

File Edit Tabs Help

```
second.s:14: Error: bad instruction `idr r2,=c'
second.s:18: Error: bad instruction `svc#0'
pi@raspberrypi1:~ $ nano second.s
pi@raspberrypi1:~ $ as -o second.o second.s
second.s: Assembler messages:
second.s:2: Error: unknown pseudo-op: `.section.data'
second.s:6: Error: unknown pseudo-op: `.section_text'
second.s:7: Error: unknown pseudo-op: `.global_start'
pi@raspberrypi1:~ $ nano second.s
pi@raspberrypi1:~ $ nano second.s
pi@raspberrypi1:~ $ as -o second.o second.s
pi@raspberrypi1:~ $ ld -o second second.o
pi@raspberrypi1:~ $ ./second
bash: ./: Is a directory
pi@raspberrypi1:~ $ ./second
Segmentation fault
pi@raspberrypi1:~ $
```

Free space: 21.7 GiB (Total: 27.4 GiB)

After I see the all the list inside second.o by debugging, to stop debugging, we set break point at b 15, in order to step by step , I used “stepi” command to execute program, as shown in the below

File Edit Tabs Help

```
pi@raspberrypi1:~ $ ld -o second second.o
pi@raspberrypi1:~ $ gdb second
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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 second...done.
(gdb) 
```

Free space: 21.7 GiB (Total: 27.4 GiB)

The screenshot shows a terminal window titled "pi@raspberrypi1: ~". The window contains the following text:

```
15         str r1,[r2]      @sotr r1 into memory c
16
17         mov r7,#1        @Program Termination:exit syscall
18         svc #0          @Program Termination:wake kernel
19         .end
(gdb) b 15
Breakpoint 1 at 0x1008c: file second.s, line 15.
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:15
15         str r1,[r2]      @sotr r1 into memory c
(gdb) stepi
17         mov r7,#1        @Program Termination:exit syscall
(gdb) x/3xw 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/3xw 0x1008c
0x1008c <_start+24>:    0xe5821000      0xe3a07001      0xef000000
(gdb) 
```

The right side of the terminal window shows a file list:

- 0-02-20-05
- 0-02-20-06
- arithmetic1
- arithmetic1.o
- arithmetic1.s
- .s
- second

At the bottom, it says "Free space: 21.7 GiB (Total: 27.4 GiB)".

This command `x/3xw 0x100990` was used to generate three words in hexadecimal: `0xe5821000`, `0xe3a07001`, `0xef000000`.

## Part2)

And here is program to represent Register = val2 +9 +val3 -val1

After write code inside arithmetic2.s, I assembled file and linked it in order to receive an executable file, but also, same as part 1, I could not see any output, because, all the things inside register, and I debug to see output (GDB), picture of codes are below

A screenshot of a terminal window on a Raspberry Pi. The title bar shows the session is running on pi@raspberrypi1: ~. The window contains assembly code for a program named arithmetic2.s. The code includes sections for .text and .start, and defines registers r1, r2, r3, and val1, val2, val3. It performs operations like loading values from memory into registers, adding them, and then saving the results back to memory. The file is marked as modified. The bottom of the window shows the nano editor's command palette.

```
.section .text
.globl _start
_start:
ldr r1, =val2
ldr r1, [r1]
ldr r2, =val3
ldr r2, [r2]
add r1, r1, r2
add r1, r1, #9
ldr r3, =val1
ldr r3, [r3]
sub r1, r1, r3
str r1, [r1]
Mov r7, #7
```

A second screenshot of a terminal window on a Raspberry Pi, showing the same assembly code as the first window. The title bar shows the session is running on pi@raspberrypi1: ~. The assembly code has been modified to include a svc #0 instruction at the end of the program. The file is still marked as modified. The bottom of the window shows the nano editor's command palette.

```
add r1, r1, r2
add r1, r1, #9
ldr r3, =val1
ldr r3, [r3]
sub r1, r1, r3
str r1, [r1]
Mov r7, #7
svc #0
.end
```

As I did in part 1, I set the break point to see how program work through, to stop debugging I set break point at “b 19” to make sure, I used “info register” to check value of register and compare to the result.

to execute step by step, I used “stepi” command to help this out. Pictures are below

```
pi@raspberrypi1:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi1:~ $ gdb arithmetic2
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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/>.

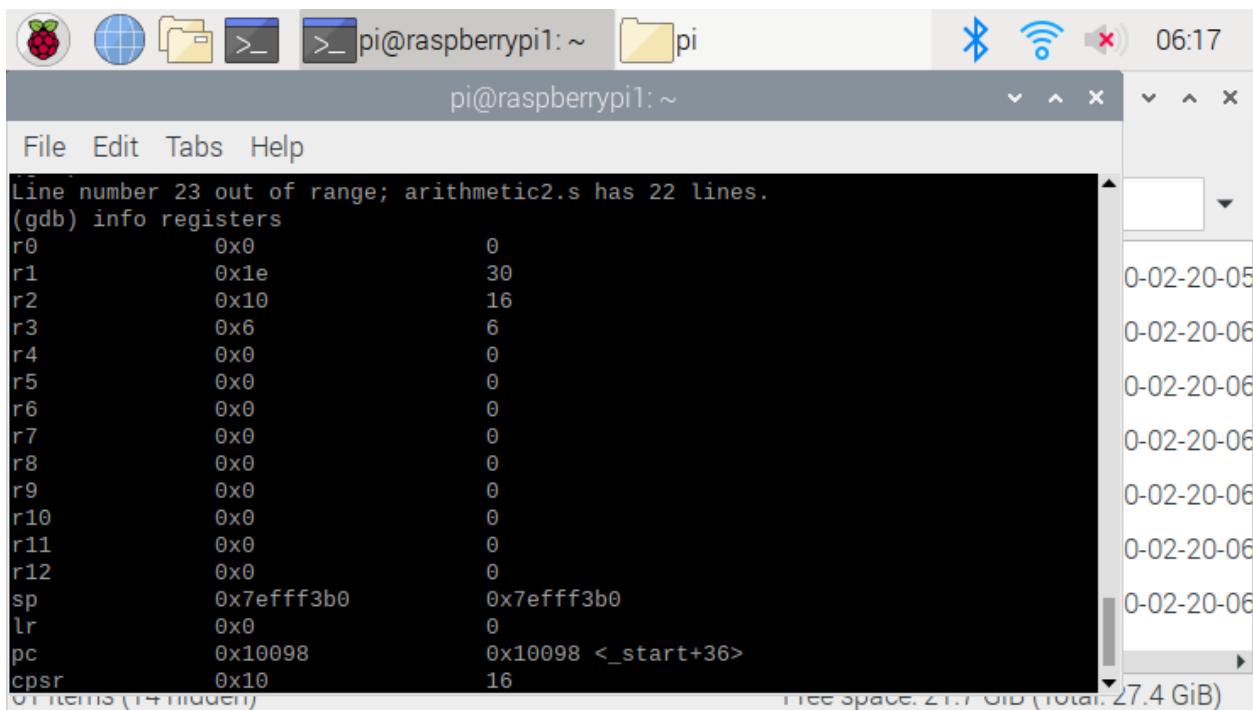
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from arithmetic2...done.
(gdb)
```

Free space: 21.7 GiB (Total: 27.4 GiB)

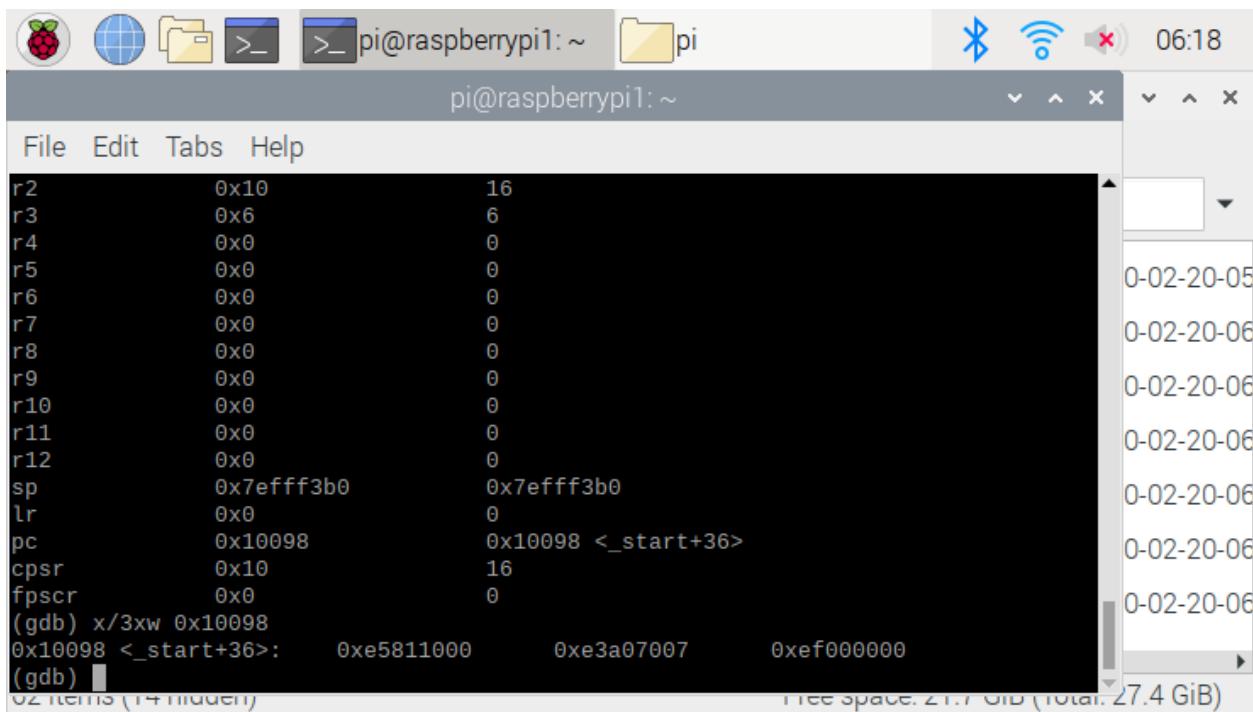
```
4      val1:.word 6
5      val2:.word 11
6      val3:.word 16
7      .section .text
8      .globl _start
9      _start:
10     ldr r1, =val2
(gdb)
11     ldr r1, [r1]
12     ldr r2, =val3
13     ldr r2, [r2]
14     add r1, r1, r2
15     add r1, r1, #9
16     ldr r3, =val1
17     ldr r3, [r3]
18     sub r1, r1, r3
19     str r1, [r1]
20     Mov r7, #7
(gdb)
```

Free space: 21.7 GiB (Total: 27.4 GiB)

This command `x/3xw 0x10098` was used to generate three words in hexadecimal: `0xe5811000`, `0xe3a07001`, `0xef000000` below picture



```
Line number 23 out of range; arithmetic2.s has 22 lines.
(gdb) info registers
r0          0x0          0
r1          0x1e         30
r2          0x10         16
r3          0x6           6
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          0x7efff3b0   0x7efff3b0
lr          0x0           0
pc          0x10098      0x10098 <_start+36>
cpsr        0x10          16
02 items (14 hidden)
Free space: 21.7 GiB (Total: 27.4 GiB)
```



```
r2          0x10         16
r3          0x6           6
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          0x7efff3b0   0x7efff3b0
lr          0x0           0
pc          0x10098      0x10098 <_start+36>
cpsr        0x10          16
fpSCR       0x0           0
(gdb) x/3xw 0x10098
0x10098 <_start+36>: 0xe5811000    0xe3a07007    0xef000000
(gdb)
02 items (14 hidden)
Free space: 21.7 GiB (Total: 27.4 GiB)
```

As the result, we can check that register 1 value is same as our equation, Register = val2, +9  
+val3 – val1

Which is 30, register 1 through 3, have same value which I declare value inside .data section.

## **Michael Knight**

### **Parallel Programming Skills:**

Part: A

1. Identifying the components on the raspberry Pi B+.

Ethernet port, 4 USB ports, HDMI port, micro USB port, Ethernet controller, CPU and RAM, Display

2. How many cores does the Raspberry Pi's CPU have?

Quadcore - 4

3. List three main differences between X86(CISC) and ARM Raspberry PI(RISC). Justify your answer and use your own words.

ARM uses less power, due to the fact that their cores don't require heatsinks to help reduce their heat.

ARM is primarily used for programming of smaller devices where X86 is used for larger computer programming.

RISC is a more register based programming than CISC uses more memory.

4. What is the difference between sequential and parallel computation and identify the practical significance of each?

Sequential computation is as the name suggests one task after another. Easier code.

Parallel computation helps reduce downtime while the CPU waits to receive information to carry on with the next task, it goes ahead and starts the next request so instead of waiting as long it starts the next step while it waits for the information.

5. Identify the basic form of data and task parallelism in computational problems.

Data Parallelism allows multiple data operations to be computed at once.

Task Parallelism refers to running multiple tasks at once.

6. Explain the differences between processes and threads.

A process is when a program runs independently, and threads are multiple processes that share information.

7. What is OpenMP and what are OpenMP pragmas?

OpenMP is an interface which supports parallel computation, and pragmas is a compiler that supports multithreaded code.

8. What applications benefit from multi-core(list four)?

## Multimedia applications, Web Servers, CAD/CAM, Compilers

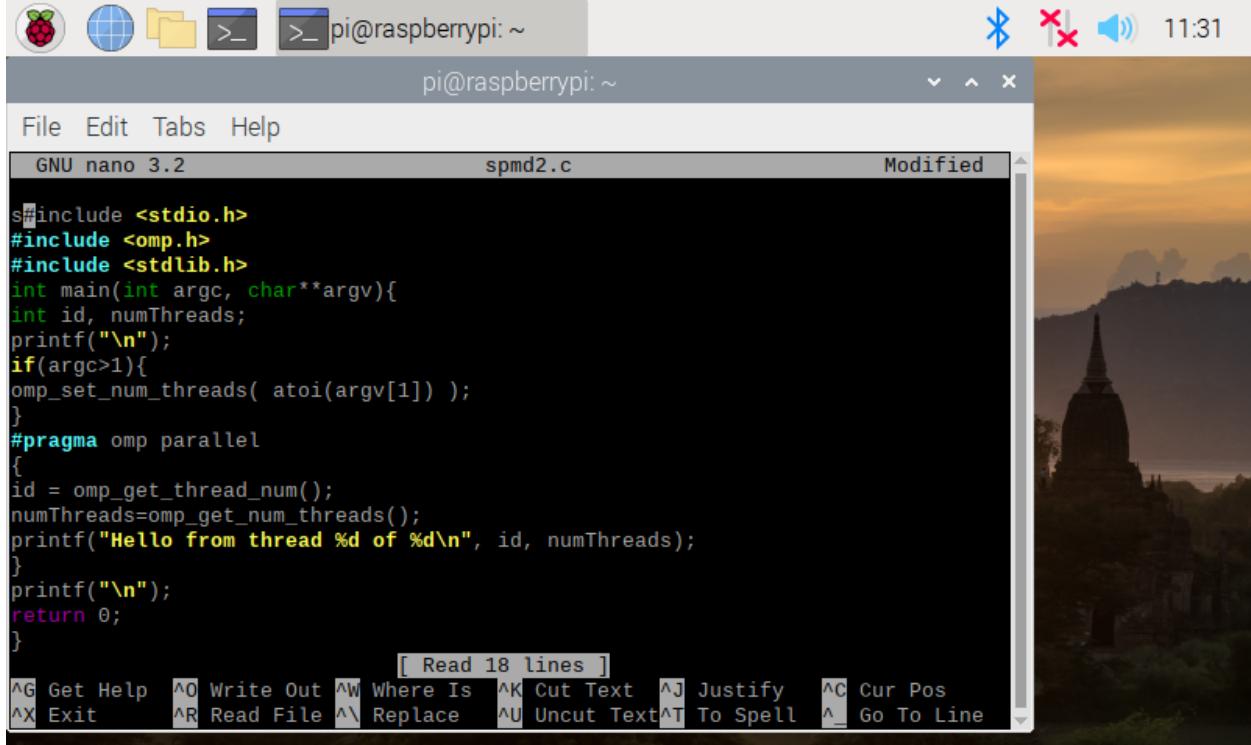
### 9. Why Multicore?(why not single core, list four)

Faster results while computing, allows for multitasking, more energy sufficient, more compact.

Part: B

### Parallel Programming Basics

First Using the code provided I ran this program.



The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window contains the command "spmd2.c" and the status "Modified". The terminal displays the following C code:

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char**argv){
    int id, numThreads;
    printf("\n");
    if(argc>1){
        omp_set_num_threads( atoi(argv[1]) );
    }
    #pragma omp parallel
    {
        id = omp_get_thread_num();
        numThreads=omp_get_num_threads();
        printf("Hello from thread %d of %d\n", id, numThreads);
    }
    printf("\n");
    return 0;
}
```

The terminal also shows a message "[ Read 18 lines ]" and a menu bar with various keyboard shortcuts for file operations like Get Help, Write Out, Where Is, Cut Text, Justify, Cur Pos, Exit, Read File, Replace, Uncut Text, To Spell, and Go To Line.

Which gave me this result.



```
pi@raspberrypi: ~
```

```
File Edit Tabs Help
```

```
return o;
^
spmd2.c:17:8: note: each undeclared identifier is reported only once for each function it appears in
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
spmd2.c: In function 'main':
spmd2.c:17:8: error: 'o' undeclared (first use in this function)
return o;
^
spmd2.c:17:8: note: each undeclared identifier is reported only once for each function it appears in
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $
```

I would have thought each tread 1 threw 4 would be displayed but instead 3 2 2 2 was the result.  
I also ran tests.



```
pi@raspberrypi: ~
```

```
File Edit Tabs Help
```

```
action it appears in
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ ./spmd2 4

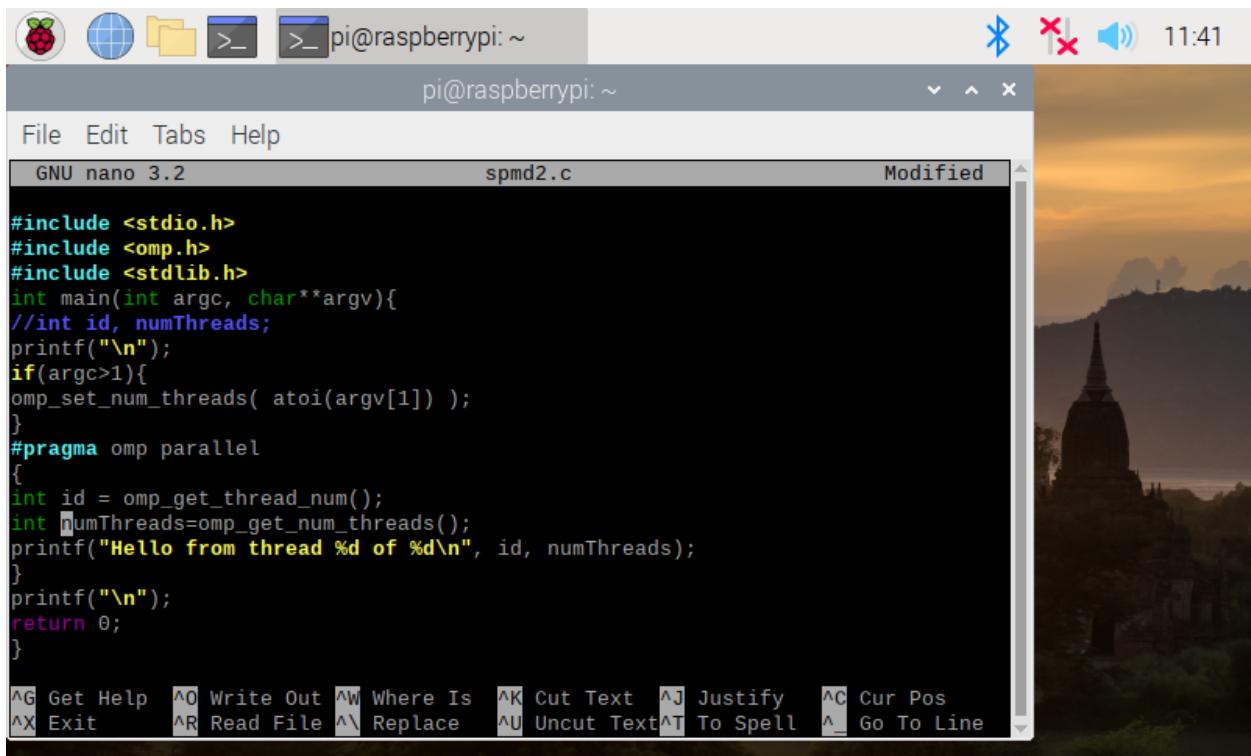
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ ./spmd2 8

Hello from thread 1 of 8
Hello from thread 6 of 8
Hello from thread 7 of 8
Hello from thread 3 of 8
Hello from thread 2 of 8
Hello from thread 0 of 8
Hello from thread 4 of 8
Hello from thread 5 of 8

pi@raspberrypi:~ $
```

I then made the corrections to the code

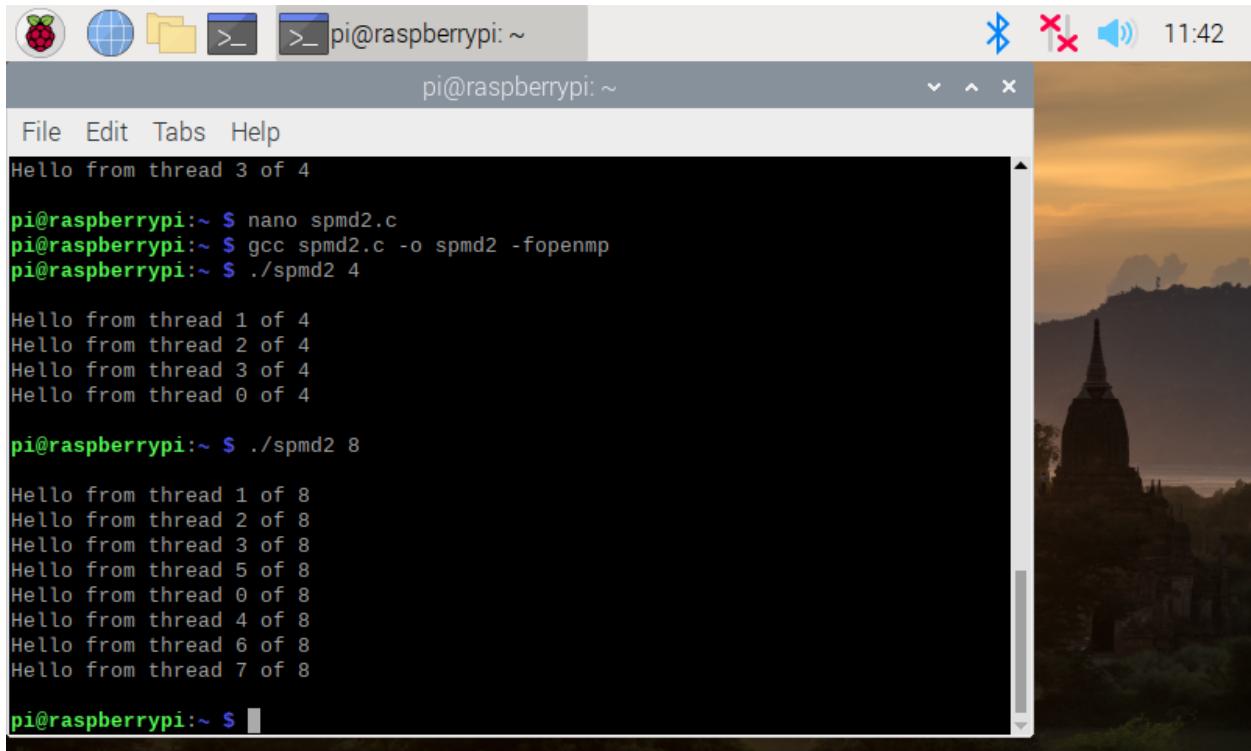


The screenshot shows a terminal window titled "pi@raspberrypi: ~". Inside the window, the nano text editor is open with the file "spmd2.c". The code in the editor is:

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc, char**argv){
//int id, numThreads;
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("Hello from thread %d of %d\n", id, numThreads);
}
printf("\n");
return 0;
}
```

At the bottom of the terminal window, there is a menu bar with options like File, Edit, Tabs, Help, and a series of keyboard shortcuts for various functions.

And ran the code.



The screenshot shows a terminal window titled "pi@raspberrypi: ~". The user has run the command "nano spmd2.c" to edit the file, then "gcc spmd2.c -o spmd2 -fopenmp" to compile it, and finally "./spmd2 4" to run it with 4 threads. The output shows four separate lines of text, each starting with "Hello from thread" followed by a number between 0 and 3. After this, the user runs "./spmd2 8" to run it with 8 threads, resulting in 8 separate lines of text, each starting with "Hello from thread" followed by a number between 0 and 7.

```
Hello from thread 3 of 4

pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 0 of 4

pi@raspberrypi:~ $ ./spmd2 8

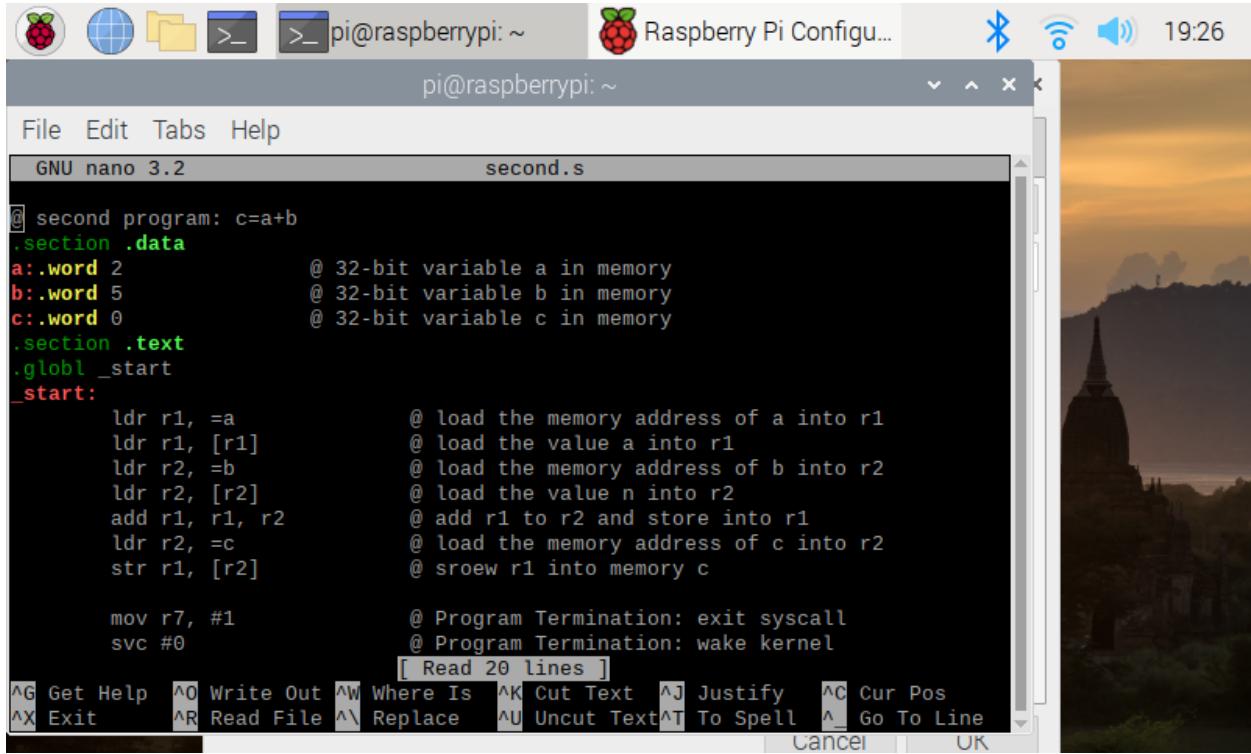
Hello from thread 1 of 8
Hello from thread 2 of 8
Hello from thread 3 of 8
Hello from thread 5 of 8
Hello from thread 0 of 8
Hello from thread 4 of 8
Hello from thread 6 of 8
Hello from thread 7 of 8

pi@raspberrypi:~ $
```

## Arm Assembly Programming A2:

PartA:

First I coded the sample code as instructed:



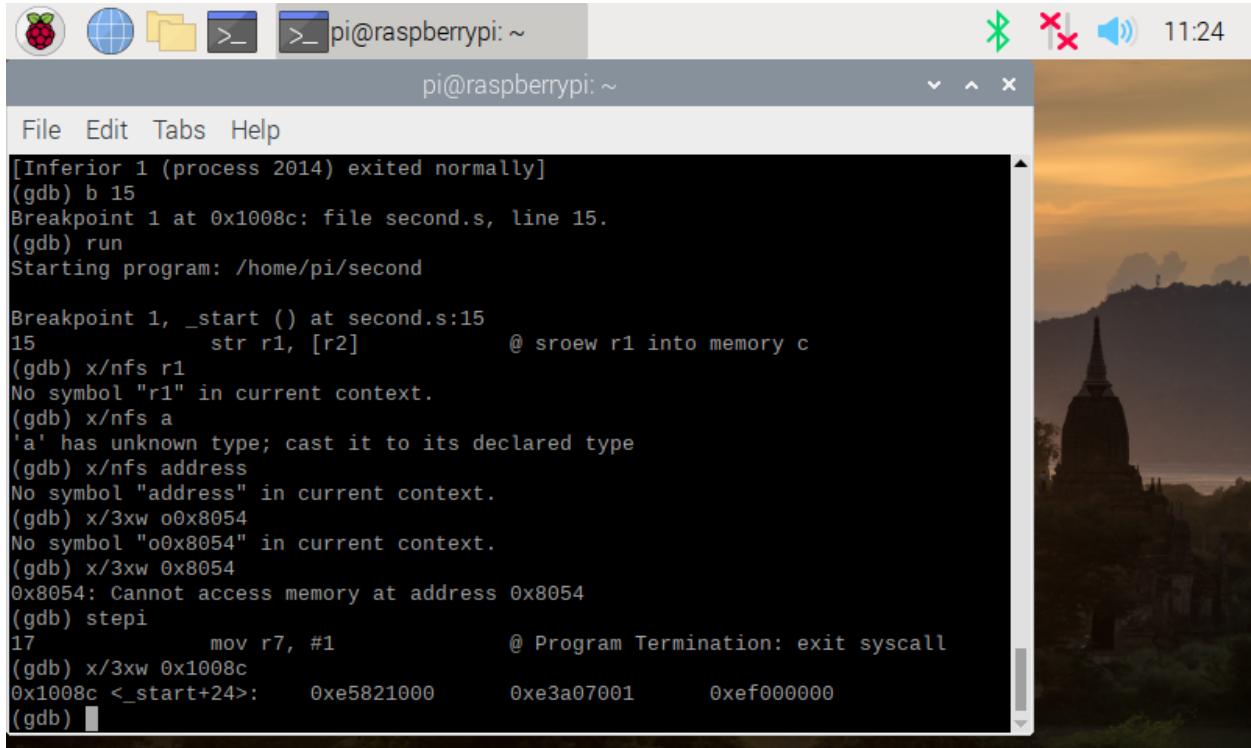
A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows the assembly code for a program named "second.s". The code defines variables "a", "b", and "c" in the ".data" section, and implements a simple addition routine in the ".text" section. The assembly instructions use RISC-like syntax with registers r1, r2, and r3. The terminal includes a menu bar with File, Edit, Tabs, Help, and a bottom row of keyboard shortcuts.

```
GNU nano 3.2 second.s
@ second program: c=a+b
.section .data
a:.word 2          @ 32-bit variable a in memory
b:.word 5          @ 32-bit variable b in memory
c:.word 0          @ 32-bit variable c in memory
.section .text
.globl _start
_start:
    ldr r1, =a      @ load the memory address of a into r1
    ldr r1, [r1]     @ load the value a into r1
    ldr r2, =b      @ load the memory address of b into r2
    ldr r2, [r2]     @ load the value n into r2
    add r1, r1, r2  @ add r1 to r2 and store into r1
    ldr r2, =c      @ load the memory address of c into r2
    str r1, [r2]    @ stroew r1 into memory c

    mov r7, #1      @ Program Termination: exit syscall
    svc #0          @ Program Termination: wake kernel
[ Read 20 lines ]
^G Get Help ^A 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 assembling, linking, and running I saw no output because there is no output to display, its only storing and playing with registers.

Then ran debug on the program with a break point at 15 aswell as checking the register



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows a GDB session. It starts by exiting an inferior process, then sets a breakpoint at line 15 of "second.s". It runs the program and reaches the breakpoint. The user then checks the register "r1" which is undefined, and the memory location at address 0x8054 which also cannot be accessed. The assembly code for the program is shown at the bottom of the screen.

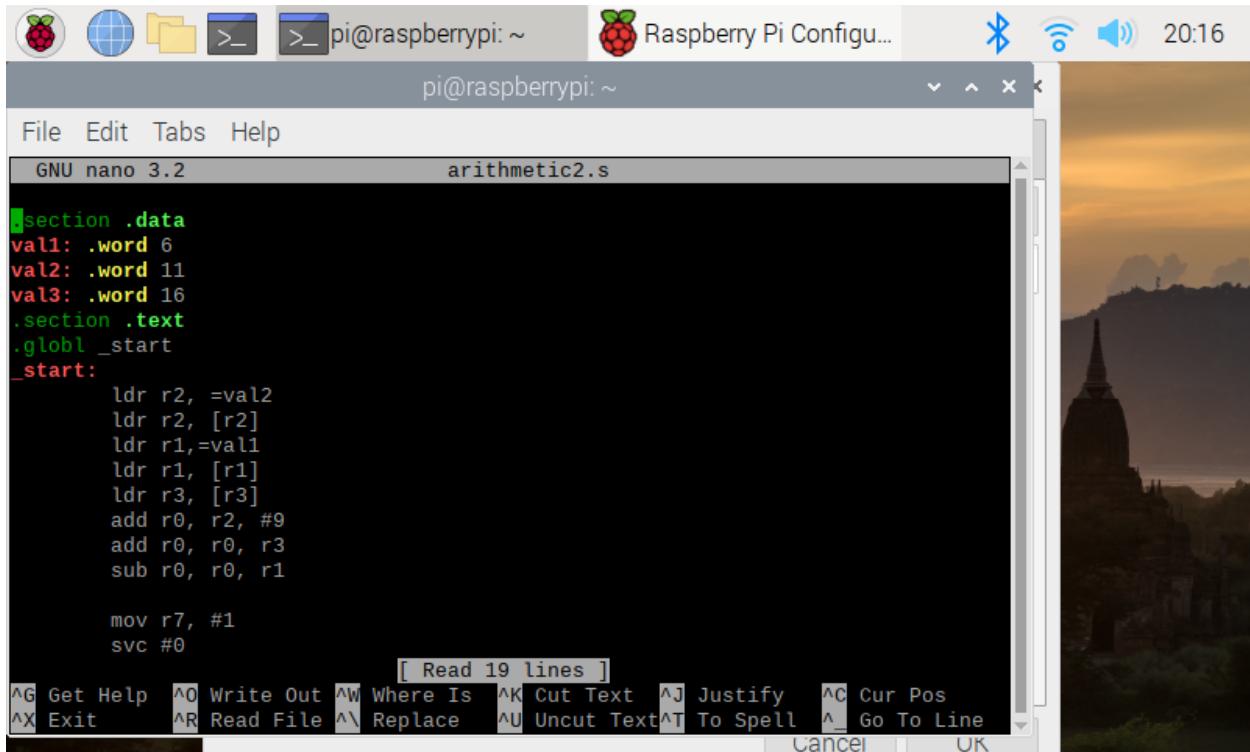
```
[Inferior 1 (process 2014) exited normally]
(gdb) b 15
Breakpoint 1 at 0x1008c: file second.s, line 15.
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:15
15          str r1, [r2]      @ stroew r1 into memory c
(gdb) x/nfs r1
No symbol "r1" in current context.
(gdb) x/nfs a
'a' has unknown type; cast it to its declared type
(gdb) x/nfs address
No symbol "address" in current context.
(gdb) x/3xw o0x8054
No symbol "o0x8054" in current context.
(gdb) x/3xw 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) stepi
17          mov r7, #1      @ Program Termination: exit syscall
(gdb) x/3xw 0x1008c
0x1008c <_start+24>: 0xe5821000 0xe3a07001 0xef000000
(gdb)
```

Gave me the results 0x25821000 0xe3a07001 0xef000000

Part2:

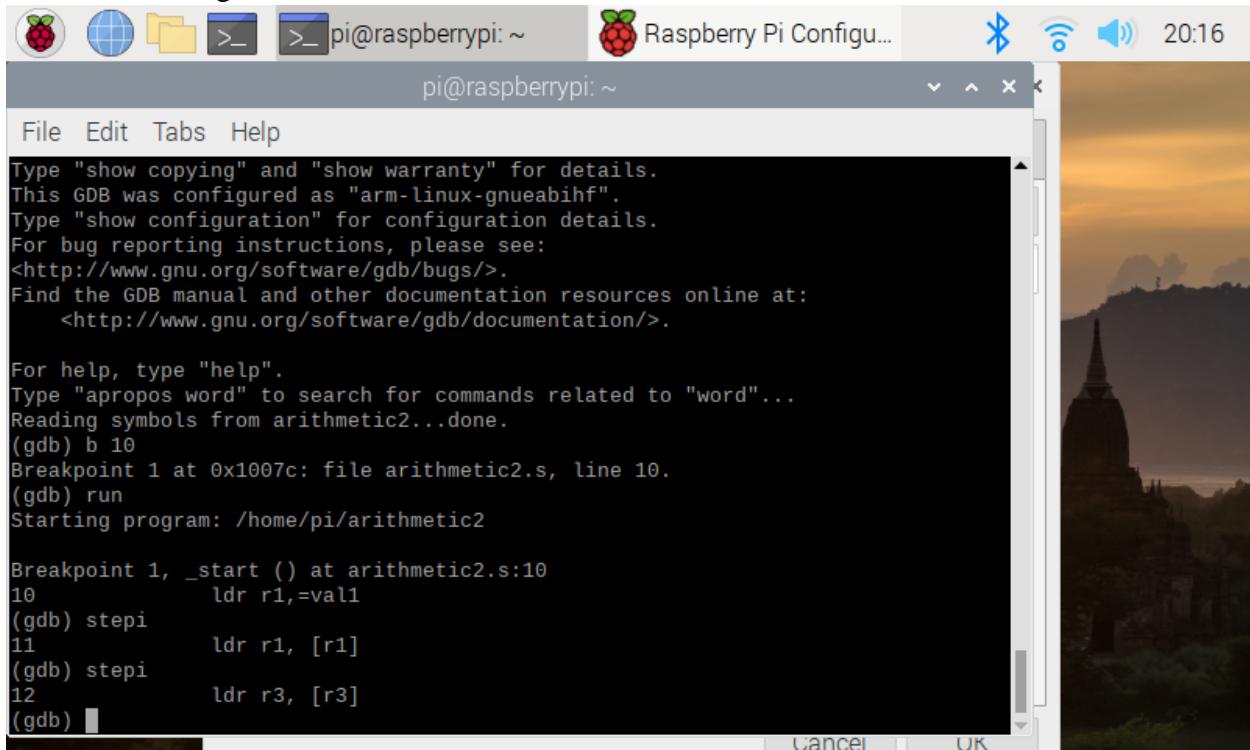
Coded what was instructed Arithmetic2:



```
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2          arithmetic2.s
section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
    ldr r2, =val2
    ldr r2, [r2]
    ldr r1,=val1
    ldr r1, [r1]
    ldr r3, [r3]
    add r0, r2, #9
    add r0, r0, r3
    sub r0, r0, r1

    mov r7, #1
    svc #0
[ Read 19 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
Cancel   OK
```

Then I ran Debug on it as instructed:



```
pi@raspberrypi: ~
File Edit Tabs Help
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 arithmetic2...done.
(gdb) b 10
Breakpoint 1 at 0x1007c: file arithmetic2.s, line 10.
(gdb) run
Starting program: /home/pi/arithmetic2

Breakpoint 1, _start () at arithmetic2.s:10
10          ldr r1,=val1
(gdb) stepi
11          ldr r1, [r1]
(gdb) stepi
12          ldr r3, [r3]
(gdb) 
```

## Tek Acharya

### Parallel Programming Skills:

- Identify the components on the Raspberry PI B+ (5)

CPU/RAM, Power socket, HDMI socket, ethernet controller, USB sockets, ethernet socket, camera connecting pin, display connecting pin, SD card slot, transistors, capacitors, diodes, serial buses, audio connector and many more.

- How many cores does the raspberry pi's B+ CPU have? (5)

The raspberry pi B+ version is a quad-core; so, four cores.

- List three main differences between x86(CISC) and ARM Raspberry PI(RISC). Justify your answer and use your own words (do not copy-paste) (8)

x86 processor has relatively fewer registers as compared to ARM. This is because x86 has a complex and ARM has reduced instruction sets.

x86 is used in PCs, large workstations and supercomputers whereas ARM is used for smaller purpose devices such as phones, routers, wearables, and tablets. Since ARM uses reduced instructions, it became easy to use them into smaller devices with fewer transistors and integrated circuitry.

ARM processor uses a register-register or load/store memory model but x86 uses the register-memory memory model. This is two separate ways of computing while in a clock cycle of instruction. ARM needs both operands in its register whereas for intel it will compute by loading one into its register and others can be done directly from the memory.

- What is the difference between sequential and parallel computation and identify the practical significance of each? (6)

The main difference between sequential and parallel computation is that in sequential only one instruction is handled by a processor at a time whereas in parallel computation multiple instructions are handled at a time using multiple cores/processors. Multi-tasking has been made easier due to this parallel computation technology. Editing a video while playing it is possible while using parallel computing but since only one instruction is processed at a given time both tasks cannot be possible in sequential computing.

- Identify the basic form of data and task parallelism in a computational problem. (5)

#### Data Parallelism:

One program → multiple data; single operation using multiple data while running a parallel computation is the base form of this problem-solving scheme. The data parallelism allows programmers to write scalable programs though the size of input and output.

#### Task/thread Parallelism:

The specification is given to multiple functions or threads rather than to data parallelism. In this system of parallelism, however, the computation is difficult to scale as it is hard to ensure that all works are done properly that contributed to the result.

- Explain the differences between process and threads. (6)

An executing instance of a thread is called a process.

The process is sure to execute but a thread may not always execute.

Thread is a subset of a process.

Usually, a process has only one thread of control – one set of machine instructions executing at a time.

All the threads running within a process share the same address space.

Usually, a process has only one thread of control – one set of machine instructions executing at a time.

- What is OpenMP and what are OpenMP pragmas? (3)

An OpenMP is a compiler with an implicit multithreading model reducing the error to a programmer. OpenMP pragmas in code mark through special directives that marks the programs into parallel computation models. Here each slave threads run independently completing its tasks and recombine later into its master while completing their tasks.

- What applications benefits from multi-core (List four)? (4)

Database server machines

Web servers

Scientific application

CAD/CAM

Multimedia applications

Compilers

➤ Why multi-core? (Why not single-core, list four?) (4)

Multi-core allows us to;

Process output faster

The need for multi-threaded tasks such as multi-tasking at a time in a computer

The computer does not get hot very often as all programs get executed faster

Low probability of signal degradation as it is processed in a short time.

The need for more parallelism in computing.

## Parallel Programming Basics:

The screenshot shows a desktop environment on a Raspberry Pi. In the top-left window, a terminal session is running the nano editor on a file named spmd2.c. The code in the file is as follows:

```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
int main(int argc, char**argv){
//int id, numThreads;
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("Hello from thread %d of %d\n", id, numThreads);
}
printf("\n");
return 0;
}
```

The bottom-left terminal window shows the compilation and execution of the program:

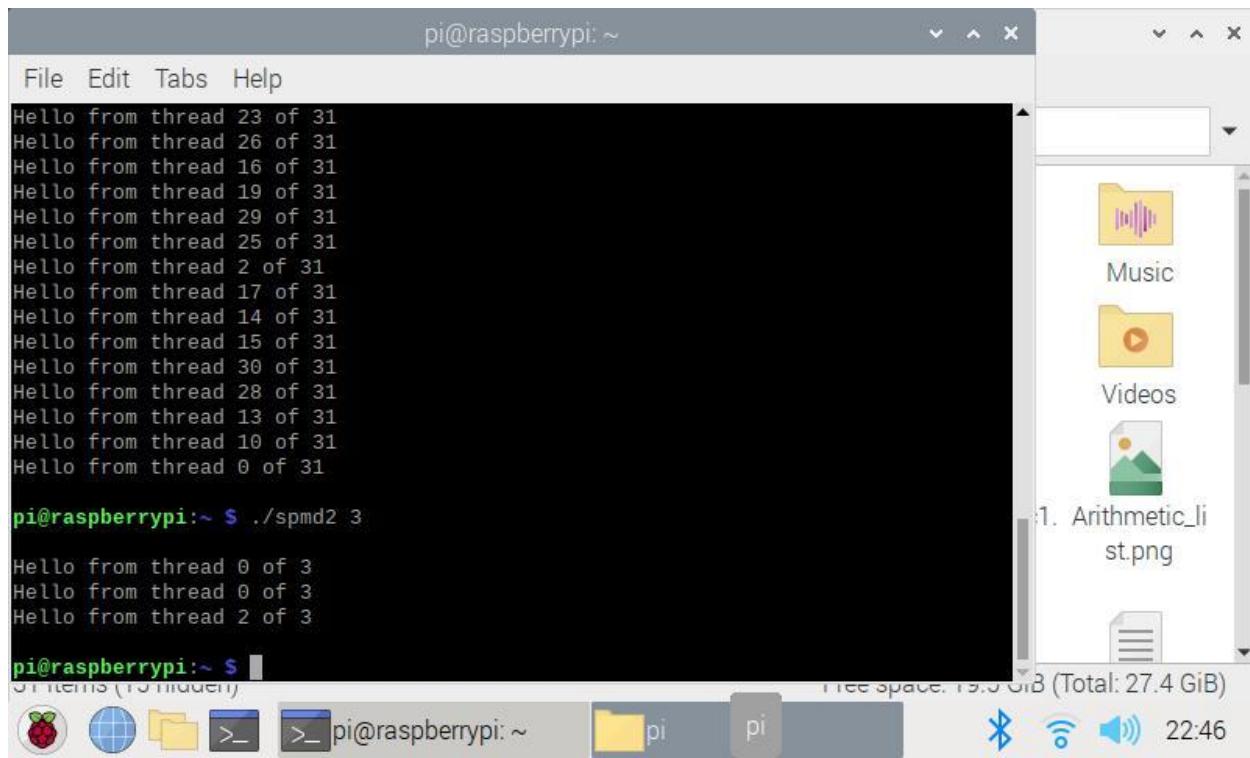
```
pi@raspberrypi:~ $ nano
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
spmd2.c: In function 'main':
spmd2.c:12:4: warning: implicit declaration of function 'omp_get_thread_num_thre
ads'; did you mean 'omp_get_num_threads'? [-Wimplicit-function-declaration]
id=omp_get_thread_num_threads();
^~~~~
omp_get_num_threads
/usr/bin/ld: /tmp/ccSal0xT.o: in function `main._omp_fn.0':
spmd2.c:(.text+0xac): undefined reference to 'omp_get_thread_num_threads'
collect2: error: ld returned 1 exit status
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

The right side of the screen shows a file manager window with a sidebar containing 'Music', 'Videos', and a file named '1. Arithmetic\_li st.png'. The main area of the file manager shows a list of files.

Compiled and ran the spmd2.c file in nano and tried pragma with 4 threads in parallel.

Observed that only three from id 2 and one from id 3 has been involved out of four cores.

This is because, the parallel programming hasn't been started that started with curly braces from where the pragma has started. Even though the code was there, it was incorrect-the variable declaration wasn't done.



```
pi@raspberrypi:~
```

File Edit Tabs Help

```
Hello from thread 23 of 31
Hello from thread 26 of 31
Hello from thread 16 of 31
Hello from thread 19 of 31
Hello from thread 29 of 31
Hello from thread 25 of 31
Hello from thread 2 of 31
Hello from thread 17 of 31
Hello from thread 14 of 31
Hello from thread 15 of 31
Hello from thread 30 of 31
Hello from thread 28 of 31
Hello from thread 13 of 31
Hello from thread 10 of 31
Hello from thread 0 of 31
```

```
pi@raspberrypi:~ $ ./spmd2 3
```

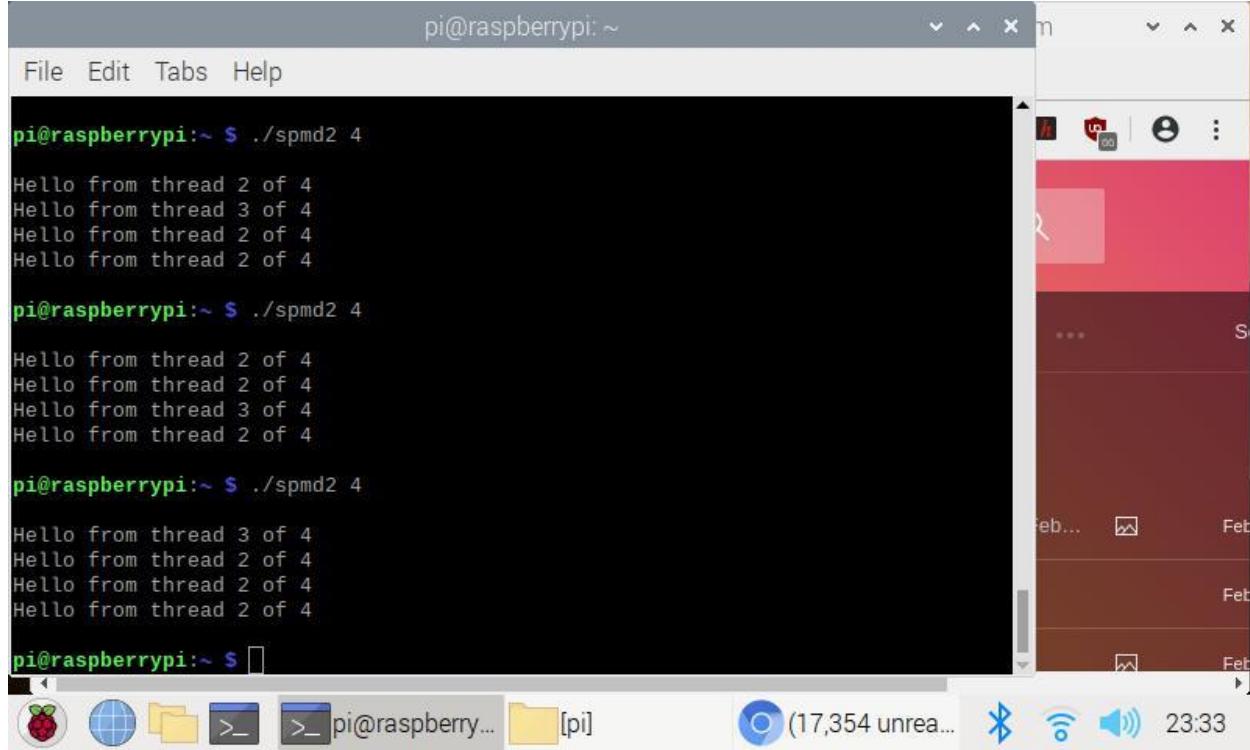
```
Hello from thread 0 of 3
Hello from thread 0 of 3
Hello from thread 2 of 3
```

```
pi@raspberrypi:~ $
```

Free space: 19.5 GiB (Total: 27.4 GiB)

Tried running with different thread counts:

Noticed that not all cores have been used.



```
pi@raspberrypi:~
```

File Edit Tabs Help

```
pi@raspberrypi:~ $ ./spmd2 4
```

```
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
```

```
pi@raspberrypi:~ $ ./spmd2 4
```

```
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
```

```
pi@raspberrypi:~ $ ./spmd2 4
```

```
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
```

```
pi@raspberrypi:~ $
```

(17,354 unread...)

Tried with thread 4 multiple times. Noticed that only two cores have been used as before.

```
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 0 of 4

pi@raspberrypi:~ $
```

Once we fixed the code errors per instruction, we can see each thread coming from one core parallelly. This is because the pragma started the parallel computation as we declared the variable type within the curly braces of pragma.

```
pi@raspberrypi:~ $ .spmd2 1
bash: .spmd2: command not found
pi@raspberrypi:~ $ .spmd2.c 1
bash: .spmd2.c: command not found
pi@raspberrypi:~ $ ./spmd2 2

Hello from thread 0 of 2
Hello from thread 1 of 2

pi@raspberrypi:~ $ ./spmd2 3

Hello from thread 2 of 3
Hello from thread 1 of 3
Hello from thread 0 of 3

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 0 of 4

pi@raspberrypi:~ $
```

By now we can see the all cores working parallelly.

## Section 3: ARM Assembly Programming

### Part 1: Second Program

The screenshot shows a terminal window titled "pi@raspberrypi: ~". Inside the terminal, the nano editor is open with a file named "second.s". The code in the editor is:

```
@second program: c = a + b
.section .data
a: .word 2          @32-bit variable a in memory
b: .word 5          @32-bit variable b in memory
c: .word 0          @32-bit variable c in memory
.section _text
.global _start
_start:
    ldr r1, =a      @load the memory address of a into r1
    ldr r1, [r1]    @load the value of a into r1
    ldr r2, =b      @load the memory address of b into r2
    ldr r2, [r2]    @load the value of b into r2
    add r1,r1, r2  @add r1 to r2 and store the value into r1
    ldr r2, =c      @load the memory address of c into r2
    str r1, [r2]    @store r1 into memory c

    mov r7, #1      @program termination: exit syscall
    svc #0          @program termination: wake kernel
[ Read 19 lines ]
```

The terminal window also shows various keyboard shortcuts at the bottom, including "Get Help", "Write Out", "Where Is", "Cut Text", "Justify", "Cur Pos", "Exit", "Read File", "Replace", "Uncut Text", "To Spell", and "Go To Line". The status bar at the bottom right shows "(17,618 unrea...", a signal icon, a battery icon, and the time "15:34".

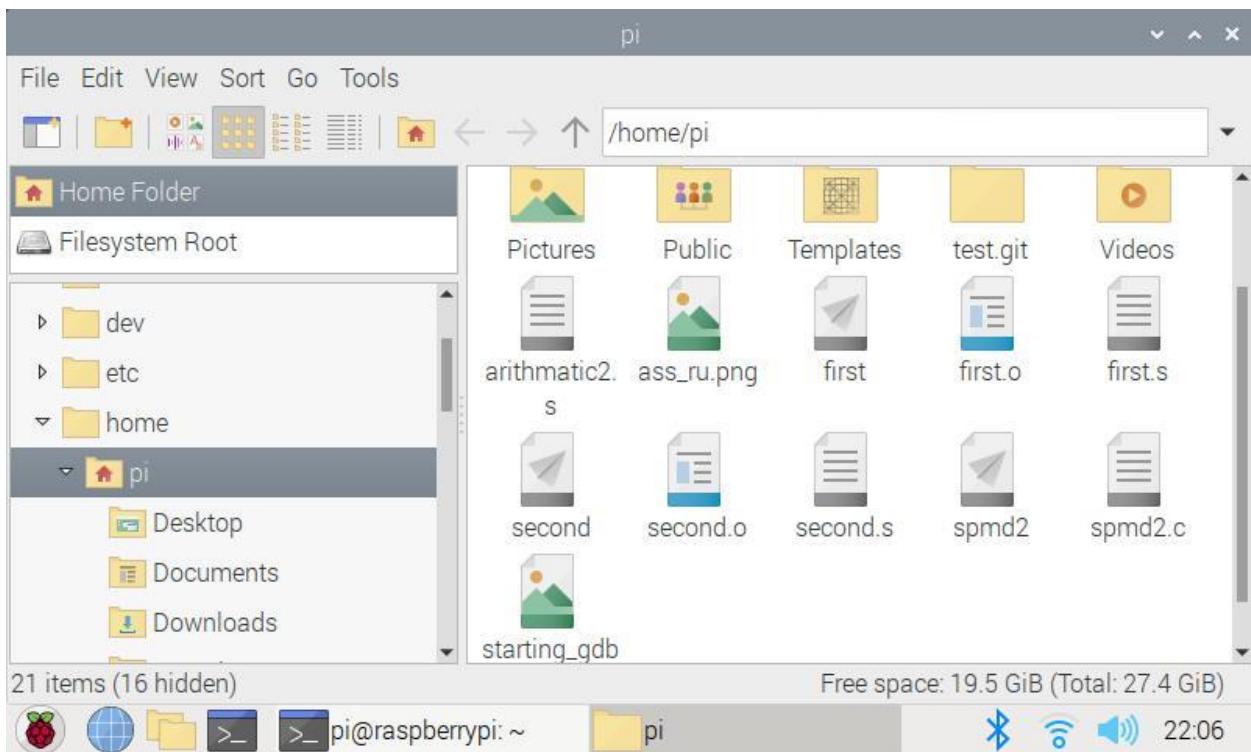
Second.s compiled using nano editor

The screenshot shows a terminal window titled "pi@raspberrypi: ~". The user runs the command "nano second.s" to edit the assembly code. They then run "as -o second.o second.s" to assemble it. When they try to run "./second", they get a segmentation fault. The terminal output is:

```
second.s: Assembler messages:
second.s:18: Error: bad instruction `svc#0'
pi@raspberrypi:~ $ nano second.s
pi@raspberrypi:~ $ as -o second.o second.s
second.s: Assembler messages:
second.s:18: Error: bad instruction `svc#0'
pi@raspberrypi:~ $ nano second.s
pi@raspberrypi:~ $ as -o second.o second.s
second.s: Assembler messages:
second.s:18: Error: bad instruction `svc#0'
pi@raspberrypi:~ $ nano second.s
pi@raspberrypi:~ $ as -o second.o second.s
pi@raspberrypi:~ $ 
pi@raspberrypi:~ $ id -o second second.s
id: invalid option -- 'o'
Try 'id --help' for more information.
pi@raspberrypi:~ $ ld -o second second.s
second.s: file not recognized: file format not recognized
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ ./second
bash: ./second: command not found
pi@raspberrypi:~ $ ./second
Segmentation fault
```

The status bar at the bottom right shows "(17,618 unrea...", a signal icon, a battery icon, and the time "22:04".

Assembled, linked, and ran the second.s file.



Created executable file

No output is seen as the output are not visible into the console but are inside the register of the microprocessor.

```
pi@raspberrypi: ~
File Edit Tabs Help
bash: .second: command not found
pi@raspberrypi:~ $ ./second
Segmentation fault
pi@raspberrypi:~ $ as -g -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ gdb second
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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 second...done.
(gdb) 
```



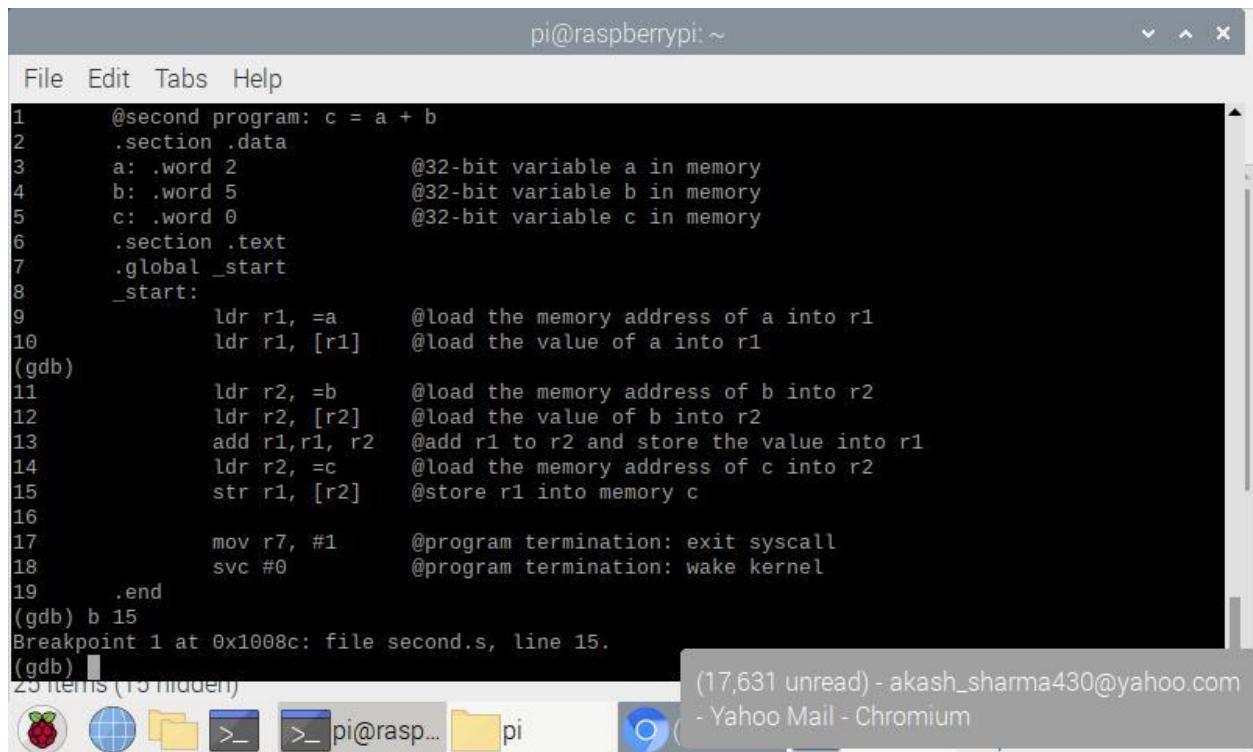
Started debugging using gdb

```
pi@raspberrypi: ~
File Edit Tabs Help
Reading symbols from second...done.
(gdb) list
1      @second program: c = a + b
2      .section .data
3      a: .word 2          @32-bit variable a in memory
4      b: .word 5          @32-bit variable b in memory
5      c: .word 0          @32-bit variable c in memory
6      .section _text
7      .global _start
8      _start:
9          ldr r1, =a      @load the memory address of a into r1
10         ldr r1, [r1]    @load the value of a into r1
(gdb)
11         ldr r2, =b      @load the memory address of b into r2
12         ldr r2, [r2]    @load the value of b into r2
13         add r1,r1, r2   @add r1 to r2 and store the value into r1
14         ldr r2, =c      @load the memory address of c into r2
15         str r1, [r2]    @store r1 into memory c
16
17         mov r7, #1      @program termination: exit syscall
18         svc #0          @program termination: wake kernel
19     .end
(gdb) 
```



Using line number for accurate debugging.

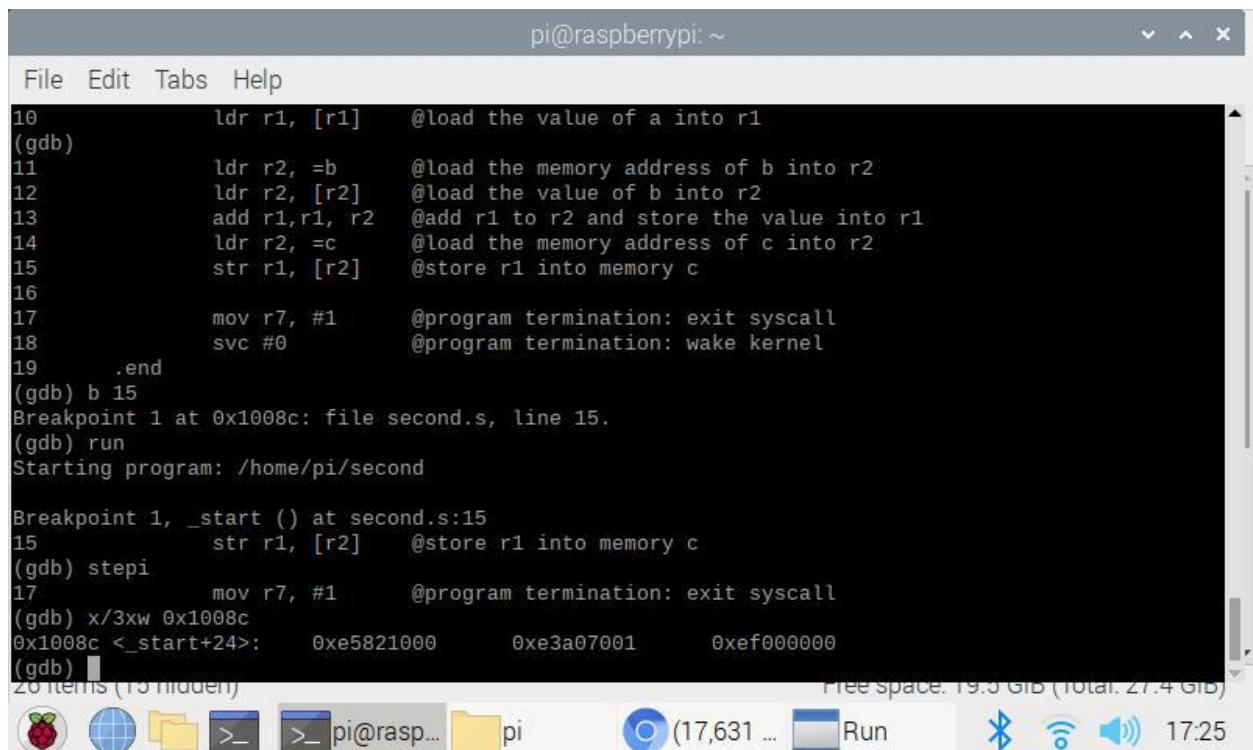
```
pi@raspberrypi: ~
File Edit Tabs Help
1      @second program: c = a + b
2      .section .data
3      a: .word 2          @32-bit variable a in memory
4      b: .word 5          @32-bit variable b in memory
5      c: .word 0          @32-bit variable c in memory
6      .section .text
7      .global _start
8      _start:
9          ldr r1, =a      @load the memory address of a into r1
10         ldr r1, [r1]    @load the value of a into r1
(gdb)
11         ldr r2, =b      @load the memory address of b into r2
12         ldr r2, [r2]    @load the value of b into r2
13         add r1,r1, r2   @add r1 to r2 and store the value into r1
14         ldr r2, =c      @load the memory address of c into r2
15         str r1, [r2]    @store r1 into memory c
16
17         mov r7, #1      @program termination: exit syscall
18         svc #0          @program termination: wake kernel
19     .end
(gdb) b 15
Breakpoint 1 at 0x1008c: file second.s, line 15.
(gdb) 
20 items (13 hidden)
```



Breakpoint set to line 15 using the command as shown above

```
pi@raspberrypi: ~
File Edit Tabs Help
10        ldr r1, [r1]    @load the value of a into r1
(gdb)
11        ldr r2, =b      @load the memory address of b into r2
12        ldr r2, [r2]    @load the value of b into r2
13        add r1,r1, r2   @add r1 to r2 and store the value into r1
14        ldr r2, =c      @load the memory address of c into r2
15        str r1, [r2]    @store r1 into memory c
16
17        mov r7, #1      @program termination: exit syscall
18        svc #0          @program termination: wake kernel
19     .end
(gdb) b 15
Breakpoint 1 at 0x1008c: file second.s, line 15.
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:15
15          str r1, [r2]    @store r1 into memory c
(gdb) stepi
17          mov r7, #1      @program termination: exit syscall
(gdb) x/3xw 0x1008c
0x1008c <_start+24>: 0xe5821000 0xe3a07001 0xef000000
(gdb) 
20 items (13 hidden)
```



Using command x/3xw 0x1008c supplied, a combination of three words used to generate in hexadecimal. These words are seen in the above slide.

## Part II

The algorithm used in this project for assembly program is

Register = Val2 + 9 + Val3 – Val1 and the contents of the memory as given below in the data segment of the code

I wrote an assembly code and compiled it as shown in the picture below.

```
File Edit Tabs Help pi@raspberrypi: ~
GNU nano 3.2 arithmetic21.s
@ arithmetic2 program
@Register = Val2 + 9 + Val3 - Val1
@Assume Val1, Val2, Val3 are 32-bit integer memory variables
@Val1 = 6, Val2 = 11, Val3 = 16
.section .data
Val1: .word 6
Val2: .word 11
Val3: .word 16
.section .text
.global _start
_start:
    ldr r1, =Val2          @load the memory address of Val2 into r1
    ldr r1, [r1]            @load the value of Val2 into r1
    ldr r2, =Val3          @load the memory address of Val3 into r2
    ldr r2, [r2]            @load the value of Val3 into r2
    add r1, r1, r2          @add r2 and r1 & store the result in r1
    add r1, #9              @add immediate 9 to r1 and store it into r1
    ldr r3, =Val1          @load the memory address of Val1 into r3
    ldr r3, [r3]            @load the value of r3 into r3
    sub r1, r1, r3          @subtract the value of r3 from r1 and store it into r1
    str r1, [r1]            @store the value of r1 back into r1
    mov r7, #1              @Program termination: exit syscall
    svc #0                 @Program termination: wake kernel
end

[ Unbound key ]
G Get Help   W Write Out   W Where Is   K Cut Text   J Justify   C Cur Pos
X Exit      R Read File   R Replace   U Uncut Text   T To Spell   A Go To Line
Free Space: 19.3 GiB (Total: 27.4 GiB)
22 items (10 hidden)  pi@raspberrypi... pi  18:33
[ Unbound key ]
G Get Help   W Write Out   W Where Is   K Cut Text   J Justify   C Cur Pos
X Exit      R Read File   R Replace   U Uncut Text   T To Spell   A Go To Line
Free Space: 19.3 GiB (Total: 27.4 GiB)
22 items (10 hidden)  pi@raspberrypi... pi  18:33
```

The executables have been created. Then I assembled, linked and run using the command line.

```
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./ arithmetic2
bash: ./: Is a directory
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ gdb arithmetic2
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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"...
```

I then used gdb debugger to see the inside of the registers and memory

```
(gdb) list
1      @ arithmetic program : val2 + 9 + val3 - val1
2
3      .section .data
4      val1:.word 6
5      val2:.word 11
6      val3:.word 16
7      .section .text
8      .globl _start
9      _start:
10     ldr r1, =val2
(gdb)
11     ldr r1, [r1]
12     ldr r2, =val3
13     ldr r2, [r2]
14     add r1, r1, r2
15     add r1, r1, #9
16     ldr r3, =val1
17     ldr r3, [r3]
18     Sub r1, r1, r3
19     str r1, [r1]
20     Mov r7, #1
(gdb)
```

Used list command for better debugging and studied inside the registers

```
Breakpoint 1, _start () at arithmetic2.s:19
19      str r1, [r1]
(gdb) info registers
r0          0x0          0
r1          0x1e         30
r2          0x10         16
r3          0x6           6
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          0x7efff3b0   0x7efff3b0
lr          0x0           0
pc          0x10098       0x10098 <_start+36>
cpsr        0x10         16
fpscr       0x0           0
(gdb) █
```

Then I set the breakpoint at line 19 as shown above and used *info register* to view into the registers. I also used *stepi* command to get into the next step while studying the memory and registers value changing stepwise.

```
r3          0x6          6
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          0x7efff3b0    0x7efff3b0
lr          0x0          0
pc          0x10098        0x10098 <_start+36>
cpsr        0x10          16
fpscr       0x0          0
(gdb) stepi

Program received signal SIGSEGV, Segmentation fault.
_start () at arithmetic2.s:19
19      str r1, [r1]
(gdb) x/3xw 0x10098
0x10098 <_start+36>:  0xe5811000  0xe3a07001  0xef000000
(gdb) █
```

The command `x/3xw 0x10098` as shown above has been used to view three words in hexadecimal from the code.

Those three words are shown in the above screenshot at the last line.

I then manually checked the arithmetic into my calculator and found that 30 corresponds to the final value of the register. Also each values of the assigned variable corresponded to the memory of the CPU.

**Vedansi Parsana**

**Parallel Programming Skills**

**A) Foundation**

**The components on the raspberry PI B+ are:**

Two USB ports, 2 power ports, a camera, HDMI port, ethernet Controller, ethernet port, CPU/RAM, and a display port.

**Raspberry PI's B+ CPU have:**

The Raspberry PI's B+ CPU has 4 cores.

**The three main differences between INTEL x86 (CISC) and ARM Raspberry PI (RISC) are:**

- 1) The CISC has a larger and more feature rich instruction set, which allows several complex instructions to access memory with more operations.
  - 2) CISC uses little-endian format for storage.
  - 3) CISC has more addressing modes but less registers than RISC.
- 1) RISC has a simplified instruction set (Which helps to execute the program quickly.), with more general-purpose registers than CISC.
  - 2) RISC's instructions only operate on registers and uses Load and Store for memory access.
  - 3) RISC is stored with BI-endian (it can switch between small and big endian).

**The difference between sequential and parallel computation and its practical significance of each are:**

As we know software is written for serial computation. The problem is categorized into discrete series of instructions and executed on a single processor and it allows only one instruction to execute at a time. The practical significance of sequential computing is the ease of writing instructions.

Parallel computing is the instantaneous use of multiple compute resources to solve a computational problem. The problem is categorized into series of instructions that can be executed parallelly in different processors. The practical significance of parallel computing is the speed at which instructions are executed.

**The basic form of data and task parallelism in computational problems are:**

Data parallelism refers to a broad category of parallelism, in which same computation (tasks) is applied to multiple data items in different cores.

Task Parallelism is used where parallelism is organized around the functions to be performed rather than performing on the data around it. In simple words, it allows to compute one task on one core and another task on another core.

**The differences between processes and threads are:**

Processes means when the program runs, they do not share memory with each other, and they operate one per core. Whereas threads are easy processes that allow executables to be decomposed into independent parts and they share a common memory with the process to which they belong.

**OpenMP and OpenMP pragmas means:**

OpenMP is an application programming interface which uses an implied multi-platform model in which the library handles thread creation and management and makes the programmer's task simpler.

OpenMP pragmas is the compilers directive that enables the compiler to generate threaded code.

**Applications that benefit from multi-core are:**

Applications that benefit from multicore usage include are not limited to database servers, scientific applications (CAD/CAM), multimedia application, and web servers.

## Multicore is used because:

Multi cores can operate running multiple processes at the time while single core can only operate one process at the time, which reduces the time to complete the task and increases the output of the system. Speeding up the clock frequencies in single core is difficult due to heat problems, maximum speed of light, and costs. Lastly, many new applications are multithreaded to utilize multicores.

## B) Parallel Programming Basics

→ This is a C program to get familiar with special kind of editions in OpenMP, which allow to run the parts of a given program on multiple threads of a multicore machine, which is shown in the following program.

→ When I first ran this example did not run as expected it showed that the first iteration of the code was that int id was declared in main method. The id appeared more than once in the output or when program was executed.

→ As all the cores share the same memory, it cannot have multiple variables with the same id.

→ I then referred to the given instructions to solve the program. After it was fixed the difficult lines with full variable declarations were done and it executed successfully as expected.

## Parallel Programming Skills

The program when ran for the first time.

pi@raspberrypi:~ \$ nano spmd2.c  
pi@raspberrypi:~ \$ gcc spmd2.c -o spmd2 -fopenmp  
pi@raspberrypi:~ \$ ./spmd2 4  
Hello from thread 2 of 4  
Hello from thread 2 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4  
  
pi@raspberrypi:~ \$ ./spmd2 3  
Hello from thread 0 of 3  
Hello from thread 0 of 3  
Hello from thread 2 of 3  
  
pi@raspberrypi:~ \$ ./spmd2 2  
bash: ./spmd: No such file or directory  
pi@raspberrypi:~ \$ ./spmd2 2  
Hello from thread 0 of 2  
Hello from thread 1 of 2  
  
pi@raspberrypi:~ \$ [ ]

pi@raspberrypi:~ \$ gcc spmd2.c -o spmd2 -fopenmp  
pi@raspberrypi:~ \$ ./spmd2 4  
Hello from thread 0 of 4  
Hello from thread 2 of 4  
Hello from thread 2 of 4  
Hello from thread 3 of 4  
  
pi@raspberrypi:~ \$ ./spmd2 3  
Hello from thread 0 of 3  
Hello from thread 0 of 3  
Hello from thread 2 of 3  
  
pi@raspberrypi:~ \$ ./spmd2 2  
bash: ./spmd: No such file or directory  
pi@raspberrypi:~ \$ ./spmd2 2  
Hello from thread 0 of 2  
Hello from thread 1 of 2  
  
pi@raspberrypi:~ \$ ./spmd2 1  
Hello from thread 0 of 1

After fixing the problem:

```
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
pi@raspberrypi:~ $ ./spmd2 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 0 of 4

pi@raspberrypi:~ $ ./spmd2 3
Hello from thread 0 of 3
Hello from thread 2 of 3
Hello from thread 1 of 3

pi@raspberrypi:~ $ ./spmd2 2
Hello from thread 0 of 2
Hello from thread 1 of 2

pi@raspberrypi:~ $ ./spmd2 1
Hello from thread 0 of 1
```

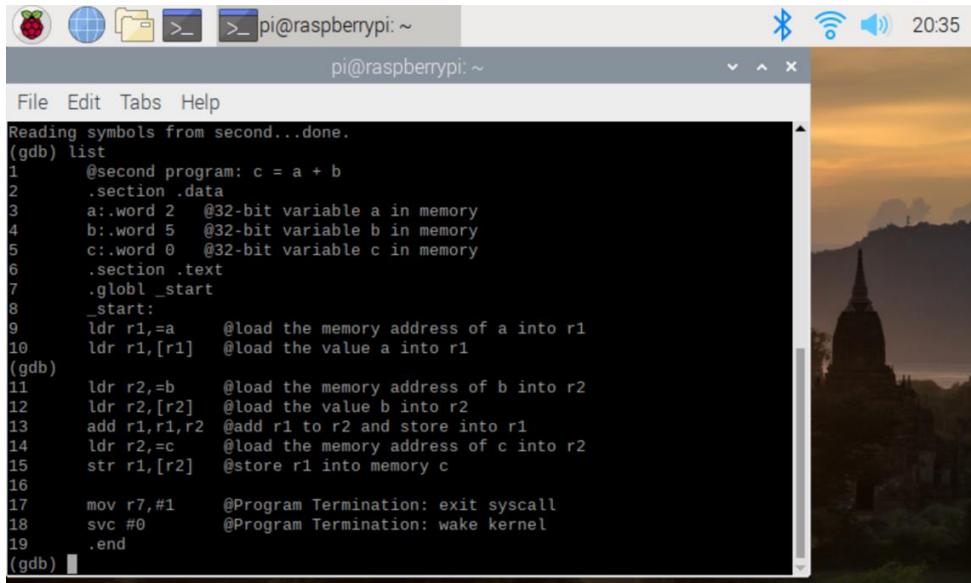
## Section 3: ARM Assembly Programming

### Part 1: Second Program

First, I composed the file, then I assembled and linked it in order to receive an executable file. However, there was no output. The entire program takes place in the registry, so we did not expect to receive any output. Next, I practiced utilizing the GDB (GNU DeBugger),

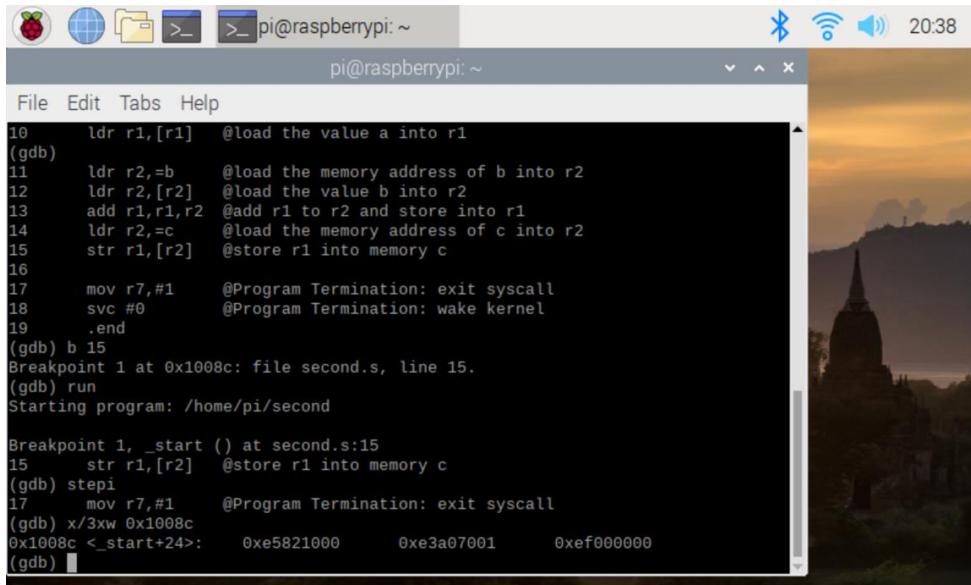
```
pi@raspberrypi:~ $ nano second.s
pi@raspberrypi:~ $ as -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ ./second
pi@raspberrypi:~ $ as -g -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ gdb second
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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 second...done.
```



```
pi@raspberrypi: ~
File Edit Tabs Help
Reading symbols from second...done.
(gdb) list
1     @second program: c = a + b
2     .section .data
3     a:.word 2    @32-bit variable a in memory
4     b:.word 5    @32-bit variable b in memory
5     c:.word 0    @32-bit variable c in memory
6     .section .text
7     .globl _start
8     _start:
9     ldr r1,a      @load the memory address of a into r1
10    ldr r1,[r1]   @load the value a into r1
(gdb)
11    ldr r2,b      @load the memory address of b into r2
12    ldr r2,[r2]   @load the value b into r2
13    add r1,r1,r2  @add r1 to r2 and store into r1
14    ldr r2,c      @load the memory address of c into r2
15    str r1,[r2]   @store r1 into memory c
16
17    mov r7,#1    @Program Termination: exit syscall
18    svc #0       @Program Termination: wake kernel
19    .end
(gdb)
```

Then I set a breakpoint to see how the program completing took place. To stop debugging I get a breakpoint at “b 15” as shown in the below image. With the help of run I executed the program. To execute the program step by step I used “stepi” command as shown in the image below.



```
pi@raspberrypi: ~
File Edit Tabs Help
10    ldr r1,[r1]   @load the value a into r1
(gdb)
11    ldr r2,b      @load the memory address of b into r2
12    ldr r2,[r2]   @load the value b into r2
13    add r1,r1,r2  @add r1 to r2 and store into r1
14    ldr r2,c      @load the memory address of c into r2
15    str r1,[r2]   @store r1 into memory c
16
17    mov r7,#1    @Program Termination: exit syscall
18    svc #0       @Program Termination: wake kernel
19    .end
(gdb) b 15
Breakpoint 1 at 0x1008c: file second.s, line 15.
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:15
15    str r1,[r2]   @store r1 into memory c
(gdb) stepi
17    mov r7,#1    @Program Termination: exit syscall
(gdb) x/3xw 0x1008c
0x1008c <_start+24>: 0xe5821000      0xe3a07001      0xef000000
(gdb)
```

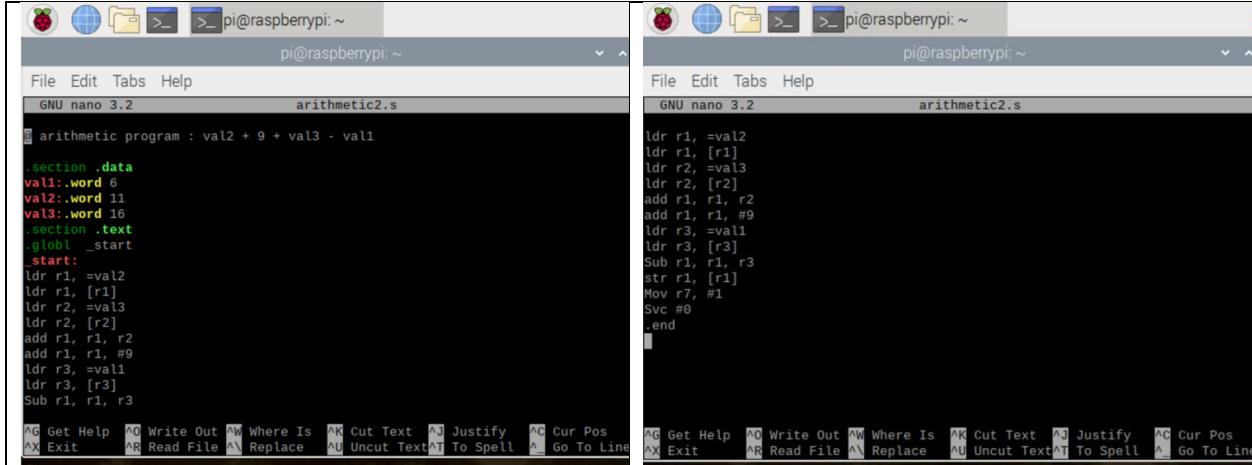
This command `x/3xw 0x100990` was used to generate three words in hexadecimal: `0xe5821000`, `0xe3a07001`, `0xef000000`.

## Part 2: Calculate the Expression

I had to write the program for the following: Register = val2 + 9 + val3 – val1, where val1=6, val2=11, and val3=16.

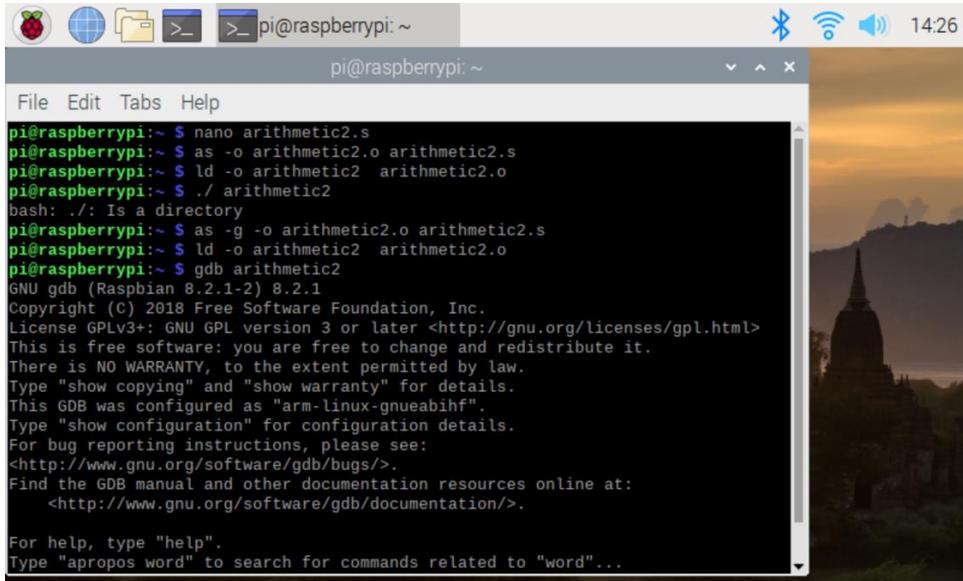
First, I composed the file,

then I assembled and linked it in order to receive an executable file. However, there was no output. The entire program takes place in the registry, so I did not expect to receive any output. Next, I practiced utilizing the GDB (GNU DeBugger) as I did in the above example.



The image shows two side-by-side terminal windows on a Raspberry Pi. Both windows have the title "pi@raspberrypi: ~". The left window displays the assembly code for "arithmetic2.s" using the "nano" editor. The right window also displays the same assembly code using the "nano" editor. The assembly code is as follows:

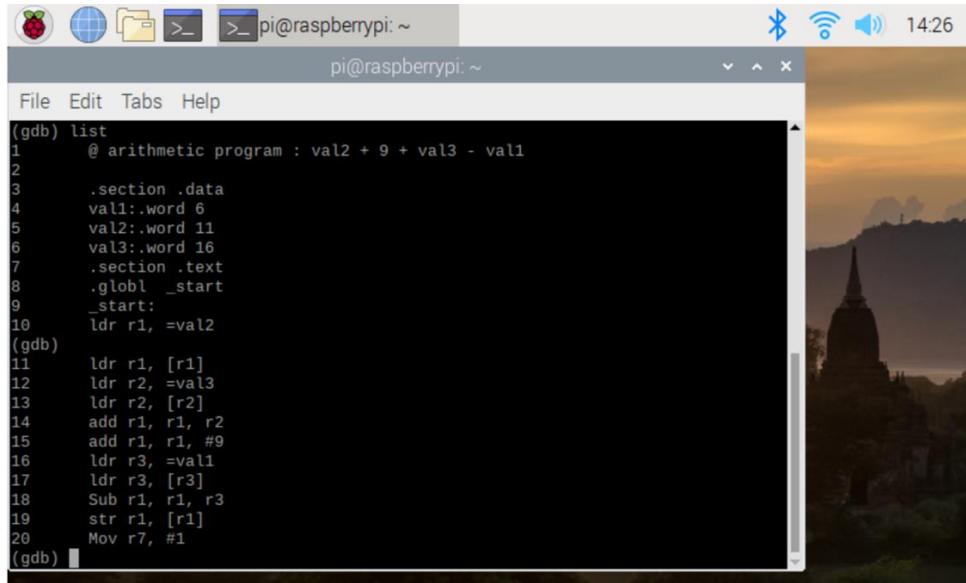
```
ldr r1, =val2
ldr r1, [r1]
ldr r2, =val3
ldr r2, [r2]
add r1, r1, r2
add r1, r1, #9
ldr r3, =val1
ldr r3, [r3]
Sub r1, r1, r3
str r1, [r1]
Mov r7, #1
Svc #0
.end
```



The image shows a single terminal window titled "pi@raspberrypi: ~". The window displays the command-line session used to compile and link the assembly code. The session starts with the user navigating to the directory containing "arithmetic2.s", then running "as -o arithmetic2.o arithmetic2.s", followed by "ld -o arithmetic2 arithmetic2.o", and finally executing the program with "./arithmetic2". The terminal then switches to the GNU GDB debugger, showing its standard welcome message and configuration details. The background of the terminal window shows a scenic sunset over a mountain range.

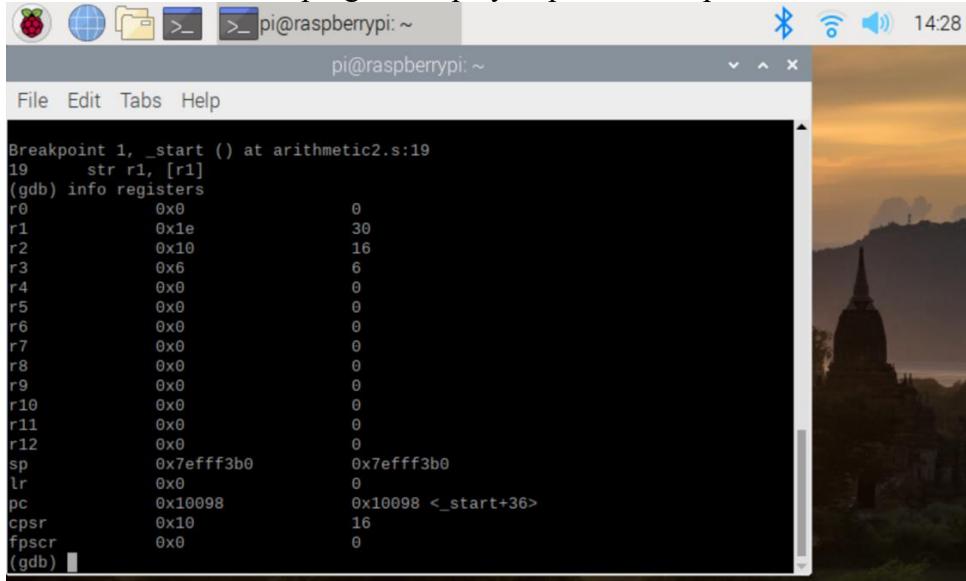
```
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
bash: ./: Is a directory
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ gdb arithmetic2
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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"...
```



```
(gdb) list
1     @ arithmetic program : val2 + 9 + val3 - val1
2
3     .section .data
4     val1:.word 6
5     val2:.word 11
6     val3:.word 16
7     .section .text
8     .globl _start
9     _start:
10    ldr r1, =val2
(gdb)
11    ldr r1, [r1]
12    ldr r2, =val3
13    ldr r2, [r2]
14    add r1, r1, r2
15    add r1, r1, #9
16    ldr r3, =val1
17    ldr r3, [r3]
18    Sub r1, r1, r3
19    str r1, [r1]
20    Mov r7, #1
(gdb)
```

As I did in the earlier example, I then set a breakpoint to see how the program completed took place. To stop debugging I get a breakpoint at “b 19” as shown in the below image. With the help of run I executed the program. I also used the command “info registers” to make sure my code is correct. To execute the program step by step I used “stepi” command as shown in the image below.



```
Breakpoint 1, _start () at arithmetic2.s:19
19    str r1, [r1]
(gdb) info registers
r0      0x0          0
r1      0x1e         30
r2      0x10         16
r3      0x6           6
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      0x7efff3b0   0x7efff3b0
lr      0x0           0
pc      0x10098       0x10098 <_start+36>
cpsr    0x10          16
fpscr  0x0           0
(gdb)
```

This command `x/3xw 0x10098` was used to generate three words in hexadecimal: `0xe5811000, 0xe3a07001, 0xef000000`.

The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window displays assembly code and register values. The assembly code includes instructions like "str r1, [r1]" and "x/3wx 0x10098". Register values show r3=6, r4=0, r5=0, r6=0, r7=0, r8=0, r9=0, r10=0, r11=0, r12=0, sp=0x7efff3b0, lr=0x0, pc=0x10098 (<\_start+36>), cpsr=16, fpcsr=0x0, and (gdb) stepi. A message indicates a SIGSEGV segmentation fault at address 0x10098. The background of the terminal window shows a scenic sunset over a pagoda.

```

r3      0x6          6
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      0x7efff3b0  0x7efff3b0
lr      0x0          0
pc      0x10098     0x10098 <_start+36>
cpsr    0x10         16
fpcsr   0x0          0
(gdb) stepi

Program received signal SIGSEGV, Segmentation fault.
_start () at arithmetic2.s:19
19      str r1, [r1]
(gdb) x/3wx 0x10098
0x10098 <_start+36>: 0xe5811000 0xe3a07001 0xef000000
(gdb) █

```

→ To make sure the code and the answer for the equation is correct I compared the values in the register and all of them were correct.

→ Register 1 has the assigned value because the answer for the equation: val2 + 9 + val3 – val1 is 30. Also, Registers 1 to 3 has the values 30, 16, and 6 which were the values assigned to them.

## Zoe Kosmicki

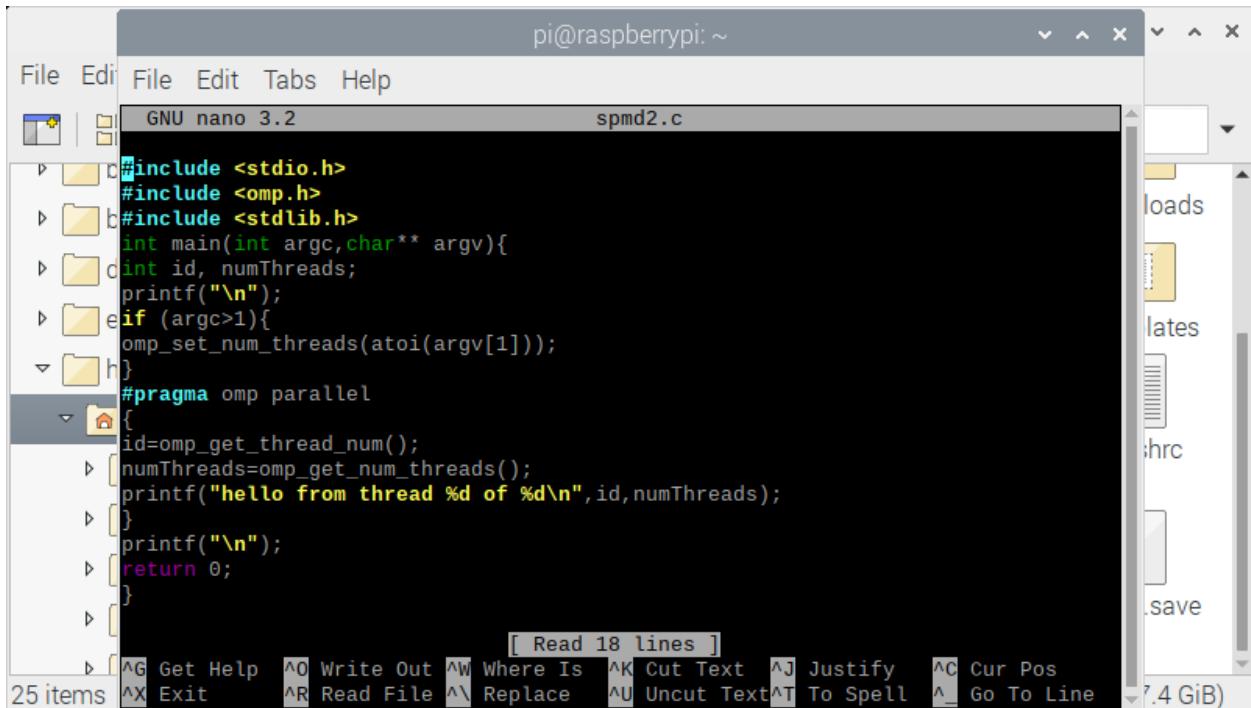
### Parallel Programming Skills

- 1) The raspberryPI B+ has a quadcore CPU that includes RAM, a microSD chip reader, and USB ports.
- 2) It has four cores.
- 3) ARM has a simplified ISA, meaning it can sometimes be faster than x86 processors. ARM also has different modes for its instructions, while x86 processors only have one. X86 processors also use little endian, while current ARM processor can switch between little and big endian.
- 4) Parallel computing is different from sequential computing because it can execute many different tasks at once, whereas sequential can only do one task at a time. As such, parallel computing can be faster, but there needs to be no dependencies between the various threads that are running, or else sequential computing must be done.
- 5) The basic form of data parallelism and task parallelism is that they're both types of parallel computing, but data parallelism focuses on finishing one task on lots of subsets

of data, while task parallelism is oriented around doing different tasks on various sets of data.

- 6) Threads are a type of process that can be broken down into smaller steps, while processes are the entire program to be executed. A process can be made of many threads, but a thread cannot contain processes.
- 7) OpenMP is an interface introduced in the 90's that supports parallel processing and computing. OpenMP pragmas are compiler directives that allow for multithreaded code to be run and compiled.
- 8) Web servers, compilers, multimedia applications, web browsers.
- 9) Multicore is generally faster than single, lets the user do multiple activities at once, multicore CPUs can be more energy efficient than single core, and it takes up less space on a circuit board since the cores are packaged into one CPU.

### Parallel Programming Task



```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc,char** argv){
    int id, numThreads;
    printf("\n");
    if (argc>1){
        omp_set_num_threads(atoi(argv[1]));
    }
    #pragma omp parallel
    {
        id=omp_get_thread_num();
        numThreads=omp_get_num_threads();
        printf("hello from thread %d of %d\n",id,numThreads);
    }
    return 0;
}
```

I began by typing out the program as listed in the instructions, checking over it to be sure I didn't mistype anything.

```

root@raspberrypi:~# nano spmd2.c
root@raspberrypi:~# gcc spmd2.c -o spmd2 -fopenmp
root@raspberrypi:~# ./spmd2 4

hello from thread 2 of 4
hello from thread 2 of 4
hello from thread 2 of 4
hello from thread 3 of 4

```

26 items root@raspberrypi:~#

Next, I ran the program using the dot slash command and tested the output with 4 threads. I tested again with 10 and 2 threads. I noticed that not all threads were being used, with some being used more than once and others not being used at all.

```

root@raspberrypi:~# ./spmd2 10

hello from thread 1 of 10
hello from thread 7 of 10
hello from thread 3 of 10
hello from thread 4 of 10
hello from thread 0 of 10
hello from thread 6 of 10
hello from thread 8 of 10
hello from thread 5 of 10
hello from thread 7 of 10
hello from thread 2 of 10

27 items root@raspberrypi:~#
root@raspberrypi:~# ./spmd2 2

hello from thread 0 of 2
hello from thread 1 of 2

28 items root@raspberrypi:~#

```

Next, I altered the code to the specifications in the instructions. I decided to rename this altered version spmd3.c, so I wouldn't confuse it with the prior program.

```

pi@raspberrypi: ~
File Edit File Edit Tabs Help
GNU nano 3.2
spmd2.c
Modified

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int main(int argc,char** argv){
//int id, numThreads;
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("hello from thread %d of %d\n",id,numThreads);
}
return 0;
}

```

I linked the file and ran it, and then it began using each thread once and only once. I again tested it with the inputs 4, 10, and 2, just to be sure my outputs were consistent across all initial inputs.

```
root@raspberrypi:~# gcc spmd3.c -o spmd3 -fopenmp
root@raspberrypi:~# ./spmd3 4
hello from thread 0 of 4
hello from thread 1 of 4
hello from thread 3 of 4
hello from thread 2 of 4
root@raspberrypi:~#
2020-02-21 08:00:00.800000 scrot... 7.4 GiB
root@raspberrypi:~# ./spmd3 10
hello from thread 5 of 10
hello from thread 8 of 10
hello from thread 2 of 10
hello from thread 9 of 10
hello from thread 6 of 10
hello from thread 0 of 10
hello from thread 4 of 10
hello from thread 1 of 10
hello from thread 7 of 10
hello from thread 3 of 10
root@raspberrypi:~#
28 items recorded 02-21 08:00:00.800000 scrot... 7.4 GiB
```

```
root@raspberrypi:~# ./spmd3 2
hello from thread 0 of 2
hello from thread 1 of 2
root@raspberrypi:~#
202-21 08:00:00 scrot... 7.4 GiB
```

They were all consistent, confirming the changes fixed the initial problem.

## ARM Assembly Programming Task

Using second.s as a template, I wrote up a version of arithmetic2 that loaded the memory address and then values of var1, var2, and var3 into r1, r2, and r3 respectively. Then I used r0 to store all the arithmetic and finished values as they were being done.

The screenshot shows a terminal window titled "pi" running on a Raspberry Pi. The file path is "/home/pi". The terminal displays the assembly code for "arithmetic2.s", which includes sections for .data and .text, global symbols for val1, val2, and val3, and a main routine (\_start) that performs arithmetic operations on these values. Below the code, the terminal shows the command-line steps to compile ("as"), link ("ld"), and run ("./") the program. The output of the program is shown as well.

```
pi@raspberrypi: ~
File Edit Tabs Help
GNU nano 3.2
arithmetic2.s
@reg = val2+9+val3-val1
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
    ldr r2,=val2
    ldr r2,[r2]
    ldr r1,=val1
    ldr r1,[r1]
    ldr r3,=val3
    ldr r3,[r3]
    add r0, r2, #9
    add r0, r0, r3
    sub r0, r0, r1

    mov r7,#1
[Wrote 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
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
pi@raspberrypi:~ $
```

After linking arithmetic2, I ran it with the dot slash instruction, and nothing happened because no output is specified. Next, I ran arithmetic2 with the debugger, adding breakpoints at lines 11 and 14.

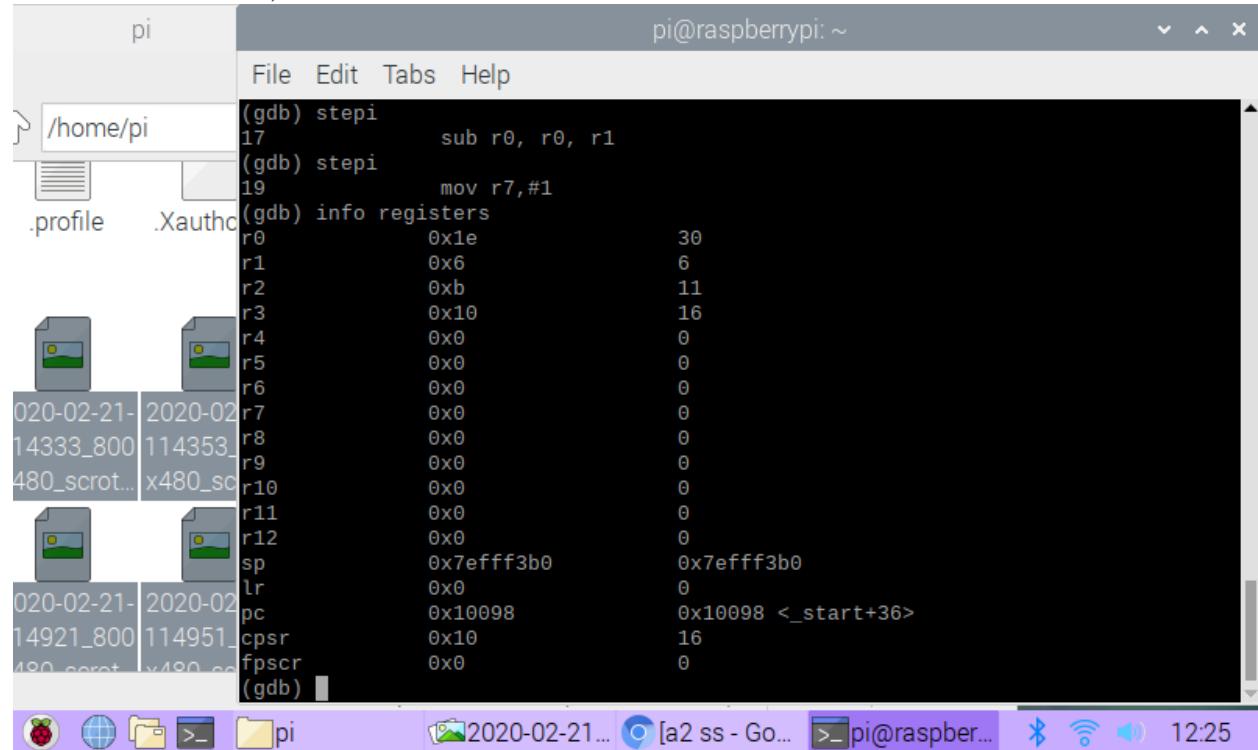
The screenshot shows a terminal window titled "pi" running on a Raspberry Pi. The file path is "/home/pi". The terminal shows the assembly code for "arithmetic2.s" and the interaction with the GDB debugger. The user lists the assembly code, sets breakpoints at lines 14 and 11, runs the program, and then inspects memory at address 0x1007c. The output shows the program starting, hitting both breakpoints, and then continuing to execute the code.

```
(gdb) list
11      ldr r1,=val1
12      ldr r1,[r1]
13      ldr r3,=val3
14      ldr r3,[r3]
15      add r0, r2, #9
16      add r0, r0, r3
17      sub r0, r0, r1
18
19      mov r7,#1
20      svc #0
(gdb) b 14
Breakpoint 1 at 0x10088: file arithmetic2.s, line 14.
(gdb) b 11
Breakpoint 2 at 0x1007c: file arithmetic2.s, line 11.
(gdb) run
Starting program: /home/pi/arithmetic2

Breakpoint 2, _start () at arithmetic2.s:11
11      ldr r1,=val1
(gdb) x/3xw 0x1007c
0x1007c <_start+8>: 0xe59f1020      0xe59f11000     0xe59f301c
(gdb)
```

I checked the memory for three hexadecimal words at the address specified at line 11, and above is what I got. After checking the memory, I used the stepi command to walk through lines 12 to 19, and checked the registers to make sure the program executed correctly.

$11 + 9 + 16 - 6 = 30$ , so this is correct.



The screenshot shows a terminal window titled "pi" running on a Raspberry Pi. The window displays a GDB session with the following commands and output:

```
(gdb) stepi
17      sub r0, r0, r1
(gdb) stepi
19      mov r7,#1
(gdb) info registers
r0      0x1e          30
r1      0x6           6
r2      0xb           11
r3      0x10          16
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      0x7efff3b0   0x7efff3b0
lr      0x0           0
pc      0x10098       0x10098 <_start+36>
cpsr    0x10          16
fpscr   0x0           0
(gdb)
```

The terminal window is part of a desktop environment, with a file browser window visible in the background showing files like ".profile" and ".Xauthrc". The taskbar at the bottom includes icons for the terminal, file browser, camera, and system status.

## Appendix

### LINKS

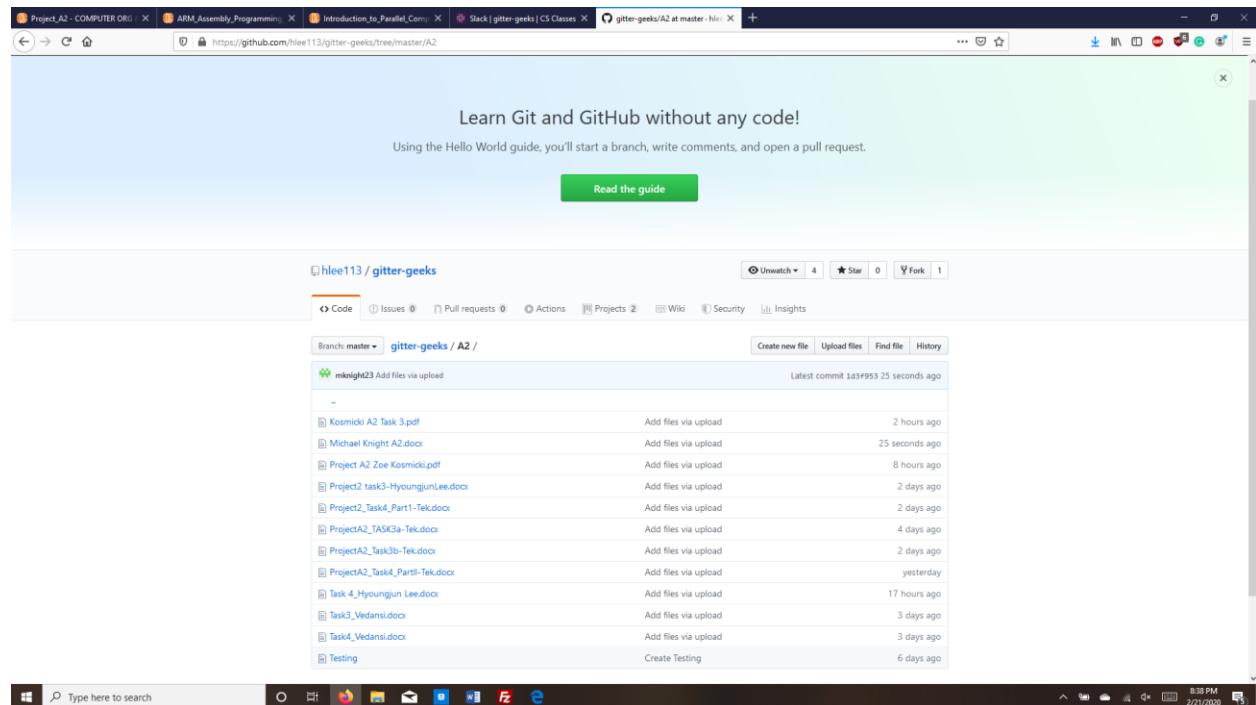
GitHub: <https://github.com/hlee113/gitter-geeks/tree/master/A2>

YouTube: <https://www.youtube.com/watch?v=w85mGA-7ugs&feature=youtu.be>

Slack: <https://app.slack.com/client/TN46XP41L/GSUL94RJ5>

### SCREENSHOTS:

#### GitHub:



#### Planning and Scheduling Timetable:

Names	Email	Task	Dependencies	Duration	Due Date	Notes
Hyoungjun Lee	hlee113@student.gsu.edu	Programming with Raspberry pi and Updating Github with new tasks	None	3 hours	2/19/20	100%
Michael Knight	mknight23@student.gsu.edu	Programming with Raspberry pi and Assigning tasks and Writing the final report and turning it in	Needs Everyone's submissions to combine into a report for submitting.	5 hours	2/21/20	I am very proud of my group and what we accomplished. Everything ran smoothly because everyone was on time or early with their submission.
Tek Acharya	tacharya2@student.gsu.edu	Programming with Raspberry pi and uploading video to youtube	None	3 hours	2/19/20	100%
Vedansi Parsana	vparsana1@student.gsu.edu	Programming with Raspberry pi	None	3 hours	2/19/20	100%
Zoe Kosmicki (coordinator)	zcosmicki1@student.gsu.edu	Programming with Raspberry pi and Video Recording/Editing	None	4 hours	2/20/20	100%