CSC 3210- Computer Organization and Programming
Assignment 2: Developing Soft and Parallel Programming Skillset Using Project- Based Learning
(Spring 2020)

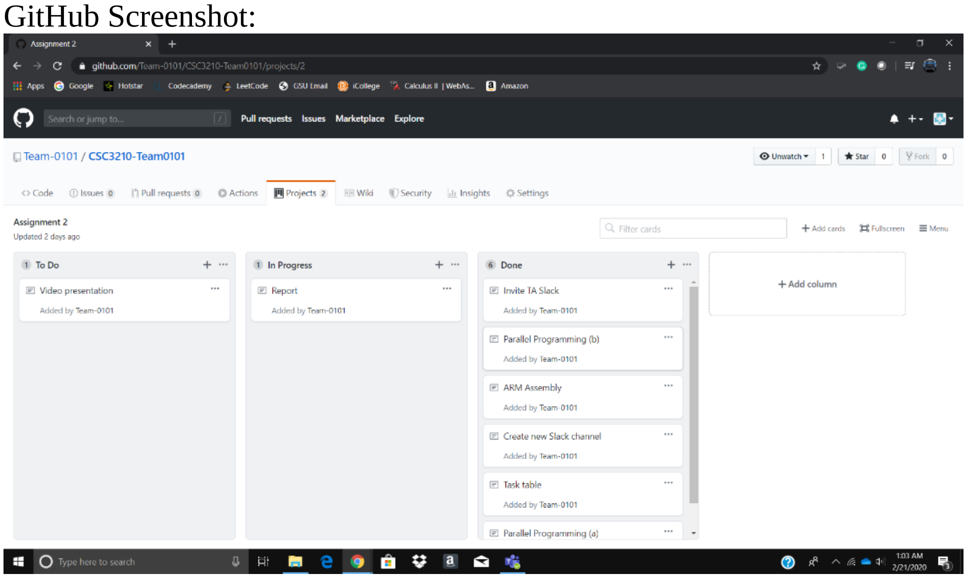
Team Name: Team 0101
Dhruv Parikh
Thu Vo
Kirby Liu
Humaira Ridi
Dhananjay Khazanchi

Task 1:

Name	**Email**	**Task**	**Duration**	**Dependency**	**Due Date**	**Note**
Dhruv Parikh (Team Coordinator)	Dparikh5@student.gsu.edu	Team Coordinator	2 hrs	Finalized report	2/21/20	
Thu Vo	Tvo55@student.gsu.edu	Parallel Programming	2 hrs	GitHub, Programming	2/21/20	Make sure the report's format is consistent and correct
Kirby Liu	Kliu8@student.gsu.edu	Arm Assembly Programming	2 hrs	Programming ,GitHub	2/21/20	
Humaira Ridi	Hridi1@student.gsu.edu	Video	2 hrs	Video Editing software, programming	2/21/20	Add music and diagrams
Dhananjay Khazanchi	Dkhazanchi1@student.gsu.edu	Report	3 hrs	Slack, GitHub, and the video	2/21/20	

Task 2:

GitHub Screenshot:



Task 3:

Dhruv Parikh

Parallel Programming Skills:

1. **Identifying the components on the raspberry PI B+**
    - Program memory (RAM), processor and graphics chip, CPU, GPU, Ethernet port, Power Connector, SD flash memory card.

2. **How many cores does the Raspberry Pi's B+ CPU have?**
    - Quad – Core Multicore CPU (4)

3. **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)**
    - The Intel X86 processor is a CISC (Complex Instruction Set Computing). X86 is faster and more powerful but also requires more energy to run. They are used in computers that can plug into the wall.
    - The ARM is a RISC (Reduced Instruction Set Computing). ARM is a simpler to use than X86. Because they have simple instruction, they can be executed more quickly than X86. The ARM is often used in smartphones and other devices that do not need constant power.

4. **What is the difference between sequential and parallel computation and identify the practical significance of each?**
    - Software for serial computing is written as:
        i. A problem is broken into a discrete series of instructions
        ii. Instructions are executed sequentially one after another
        iii. Executed on a single processor
        iv. Only one instruction may execute at any moment in time
    - Parallel programming is the simultaneous use of multiple compute resources to solve a computation problem:
        i. A problem is broken into discrete parts that can be solved concurrently.
        ii. Each part is further broken down to a series of instructions
        iii. Instructions from each part execute simultaneously on different processors
        iv. An overall control mechanism is employed.

5. **Identify the basic form of data and task parallelism in computational problems.**
    - Data parallelism refers to a board category of parallelism in which the same computation is applied to multiple data item, so the amount of available parallelism is proportional to the input size, leading to tremendous amounts of potential parallelism.
    - The broad classification of task parallelism applies to solutions where parallelism is organized around the function to be performed rather than around the data.

6. **Explain the differences between processes and threads.**
    - A process is the abstraction of a running program. Processes do not share memory with each other.

- A thread is a lightweight process that allows a single executable/ process to be decomposed to smaller, independent parts. All threads share the common memory of the process they belong to.

7. **What is OpenMP and what is OpenMP pragmas?**
   - OpenMP is a set of complier directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared – memory environment. OpenMP pragmas are complier directives that enable the complier to generate threaded code.

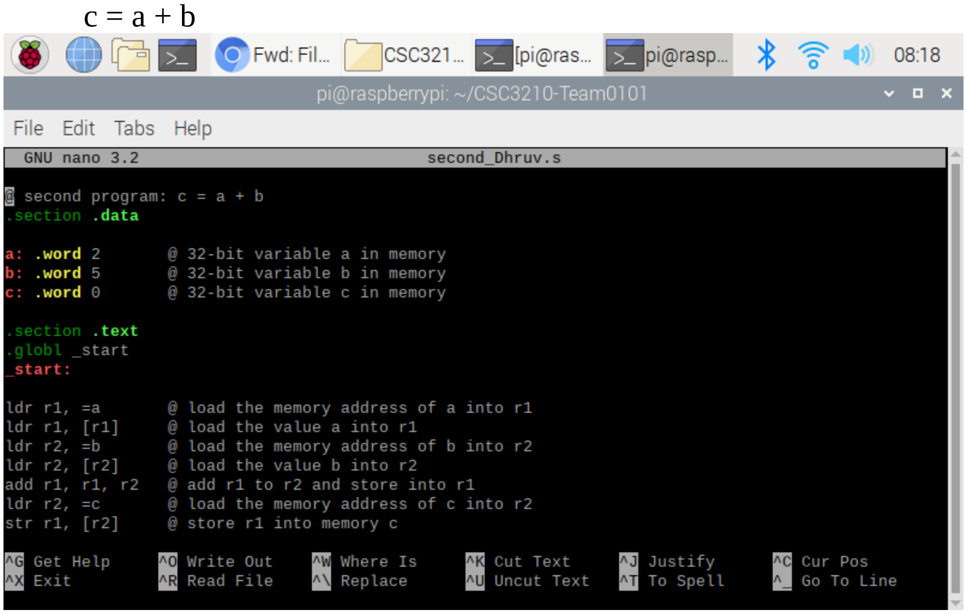
8. **What applications benefit from multi-core (list four)?**
   - Multimedia applications (Adobe Photoshop, Adobe Premier)
   - Compliers
   - Databases servers
   - Web servers

9. **Why Multicore? (why not single core, list four)**
   - Multicore processor means that you can execute multiple processes at the same time.
   - The single core processors have many faults like heat problems, speed of light problems, difficult design and verification, large design teams necessary and server farms need expensive air- conditioning.
   - Multicore increases throughput of system and run on lower frequency as compared to the single processing unit, which reduces the power dissipation or temperature.
   - Because most of the application need multicore processing rather that single core.

<u>Arm Assembler in Raspberry Pi:</u>

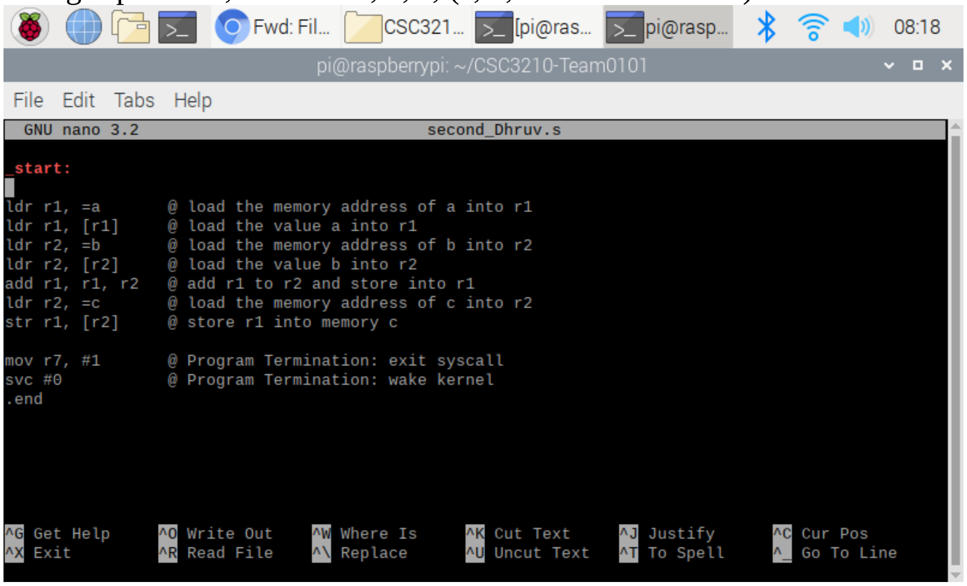
a. **<u>Part 1: Second Program</u>**

c = a + b

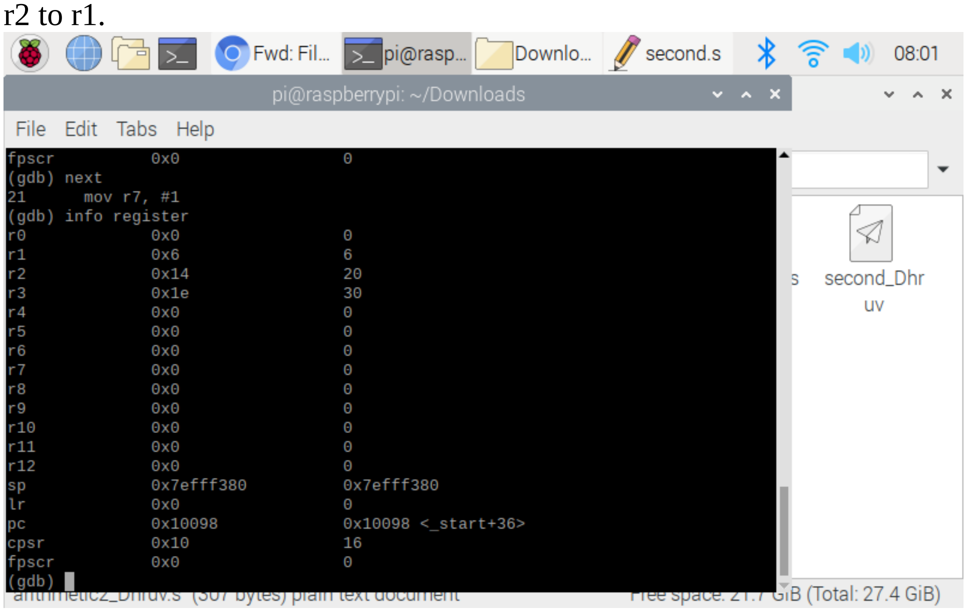


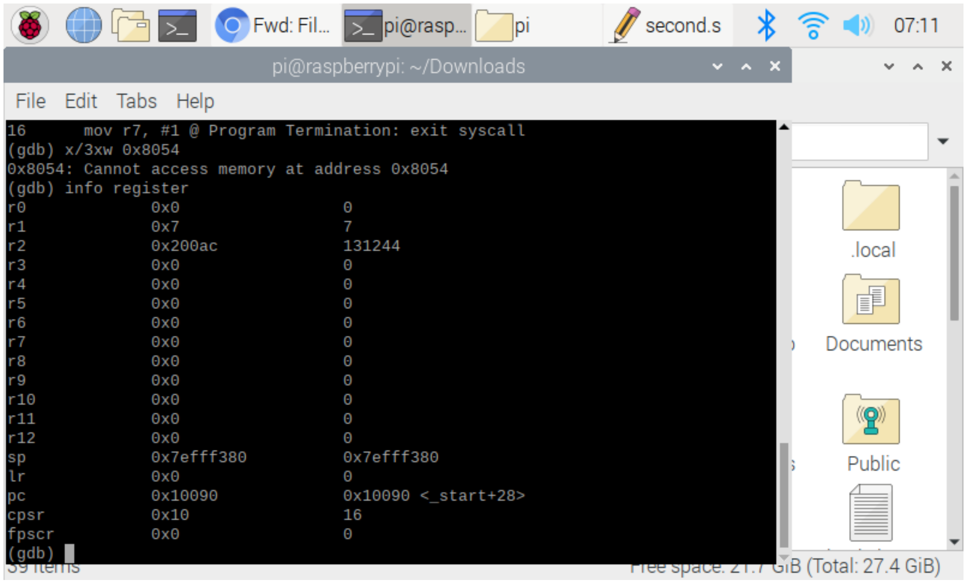

```
@ second program: c = a + b
.section .data

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 b 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]    @ store r1 into memory c
```

In this program we store the variable in the .data section of our code. The. word is a 32-bit storage space. So, we store a, b, c, (a, b, c is the addresses) with the value of 2, 5, 0 respectively.



In this snippet we store the values in the register. Ldr means to load. We are loading the value of a, b, and c to r1, r2 and r3 respectively. Str means to store. On line 16 we are storing the value of r2 to r1.

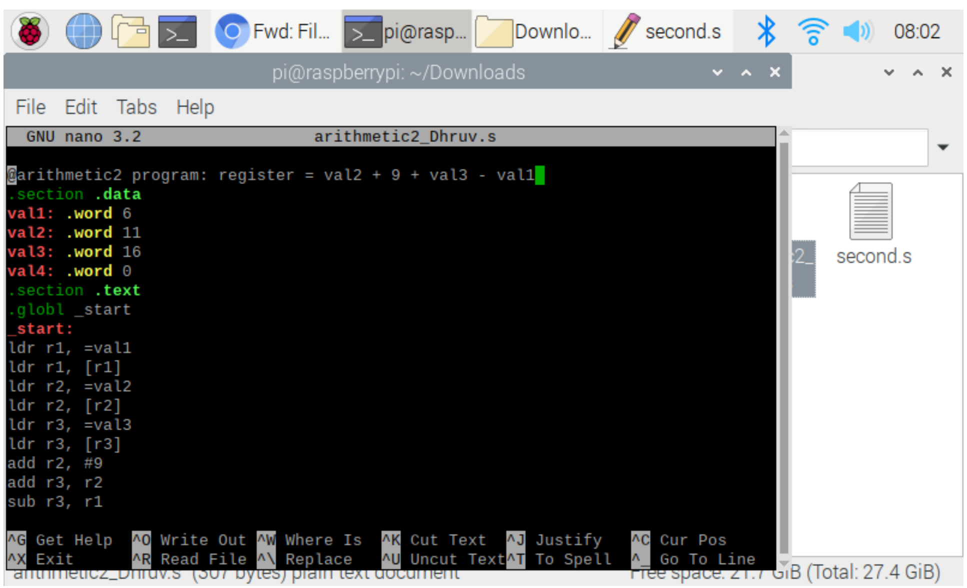




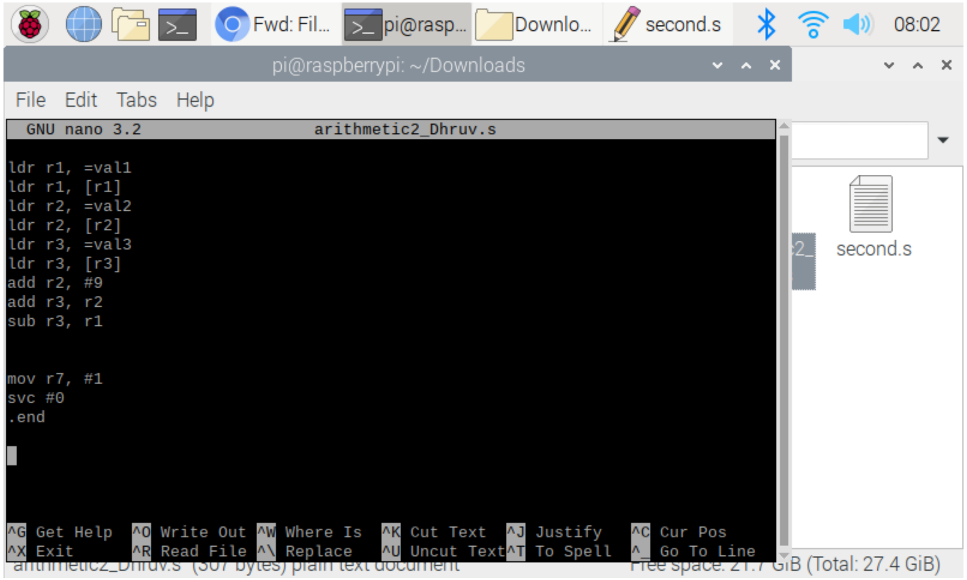
As we can see that in register 1 (r1) the value is 7 after writing the command.

b. <u>Part 2:</u>

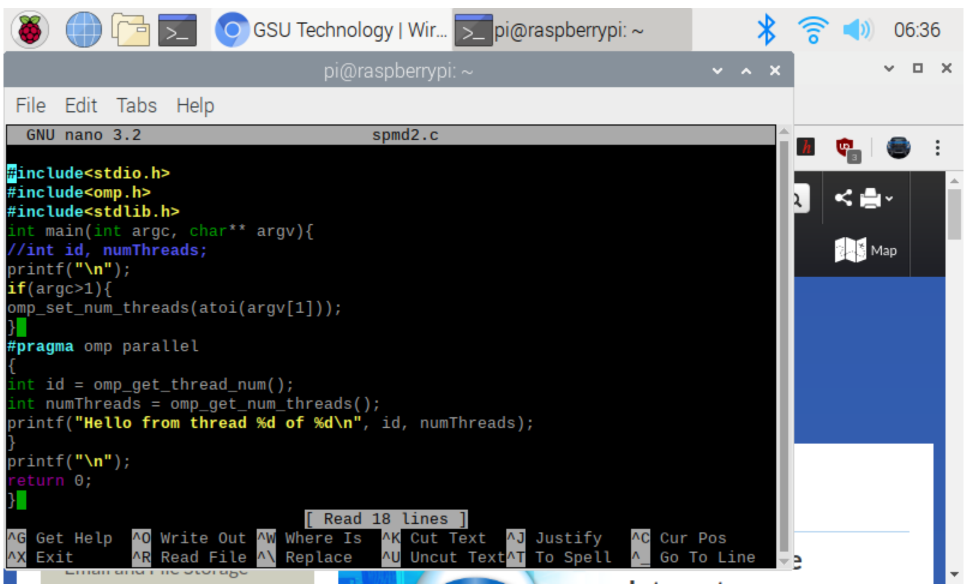
Register = val2 + 9 + val3 - val1



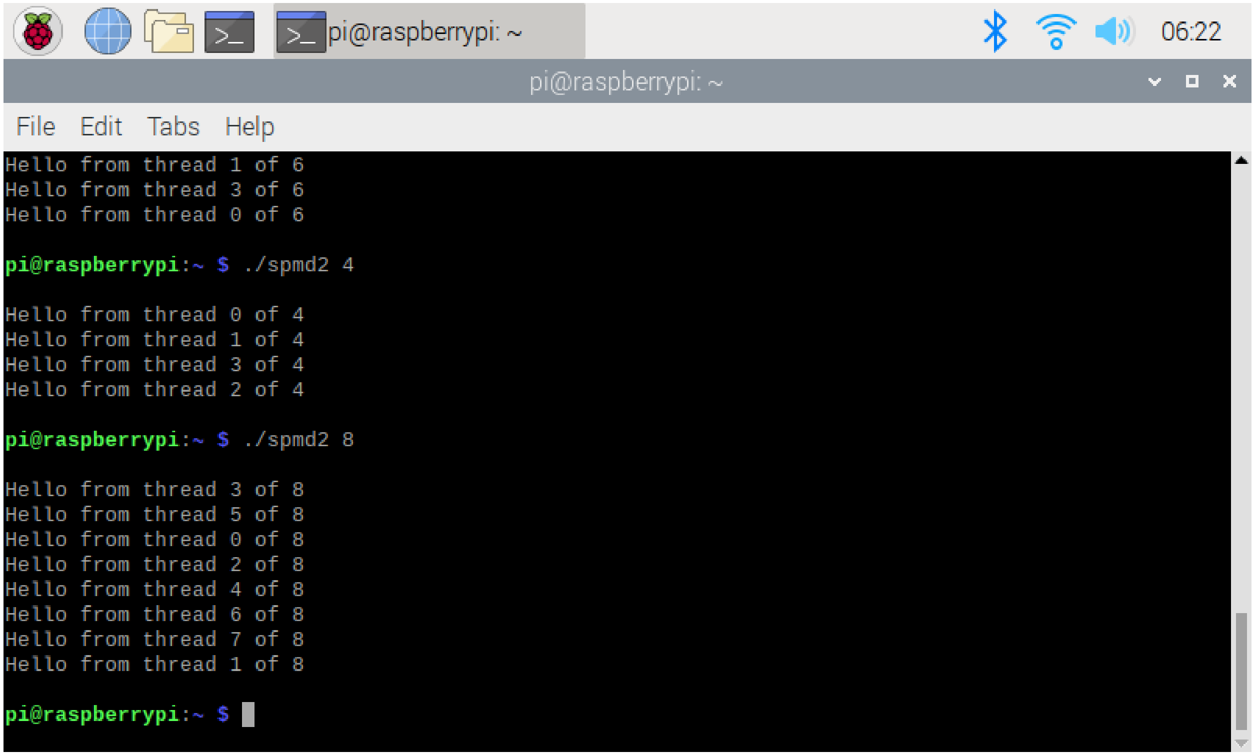
This time we store our variable as val1, val2 and val3, in .word (16-bits). An also have places my break points from ldr r1, = val1 to svc #0.



<u>Parallel Programming:</u>



In this program we see that how initializing variable outside the curly brackets make a difference in how the program uses single core or multi core. In the above screenshot we have the updated version of the code that uses all four cores of the processor. So, whenever we write the code./spmd2 (number) it's going to print different number every time.



Kirby Liu

Parallel Programming Skills: Foundations

1. **Identifying the components on the raspberry PI B+?**
   - 2x USB
   - Ethernet
   - Power
   - Ethernet Controller
   - Camera
   - HDMI
   - Power
   - Display
   - CPU/RAM

2. **How many cores does the Raspberry Pi's B+ CPU have?**
   - 4 cores

3. **List three main differences between X86 (CISC) and ARM Raspberry PI (RISC).**
   - CISC has more operations, addressing modes, but less registers than ARM
   - RISC instructions operate only on register, and therefore must use a Load/Store memory model for memory access
   - CISC has instructions that can access memory

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

- Sequential- Problems are broken down into instructions that are executed sequentially one after another, which are then executed on a single processor
- Parallel- Problems are broken into parts that can be given into instructions that can be executed simultaneously with different processors

5. **Identify the basic form of data and task parallelism in computational problems.**
    - Data Parallelism- Same computational instructions is applied to multiple data items
    - Task Parallelism- Multiple tasks is applied to the same data set divided up among different processes

6. **Explain the differences between processes and threads.**
    - Process do not share memory and is an abstraction of a running program
    - Thread are process that allow decomposed into smaller/independent parts and share memory of the process

7. **What is OpenMP and what is OpenMP pragmas?**
    - An API that supports multi-platform shared memory multiprocessing programming
    - Openmp pragmas are used to mark sections of code that is to be executed in parallel

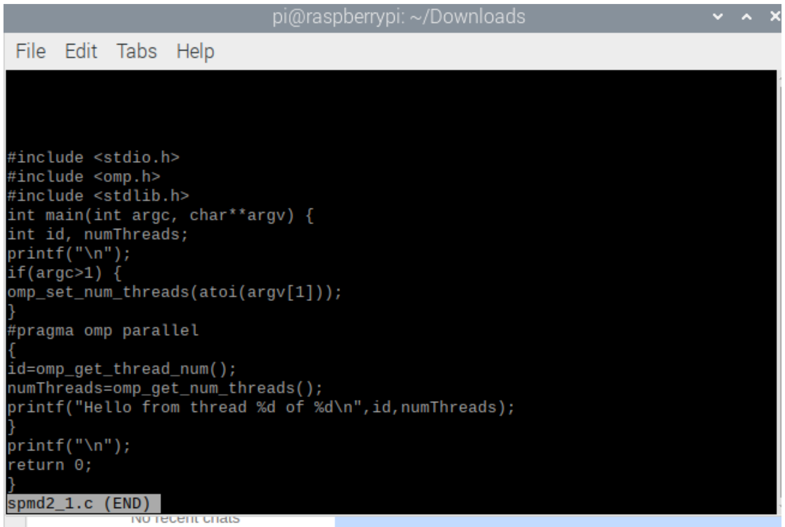
8. **What applications benefit from multi-core (list four)?**
    - Applications that are heavily multiple/parallel instruction reliant:
        - **i.** Database Servers
        - **ii.** Web Servers
        - **iii.** Multimedia Applications
        - **iv.** Scientific Application (CAD/CAM)

9. **Why Multicore? (why not single core, list four)**
    - Has the ability to process multiple applications at the same time
    - Requires less power due to the ability to turn off some of the cores
    - It is more efficient to have more cores with less computing power working together than to build a single core with more computing power working alone
    - More computing applications are built with multicore processing in mind than via single core

Kirby Liu

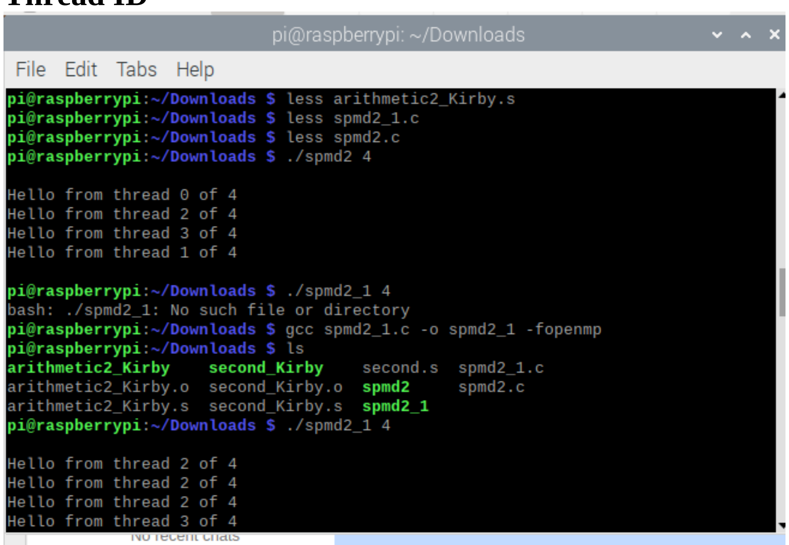
Parallel Programming Basics:

Initial Iteration of the Code



This was the first iteration of the code that was inputted.

Thread ID



As we see from running the code, there was a lot of duplications from the when the program was executed.

Code Fix

```
#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;
}
spmd2.c (END)
```

After the pragma omp parallel line which is where the parrallel process begins to form, the thread begins to diverge but does so in parallel so that the thread do not coincide.

Execution of the Code after Fix

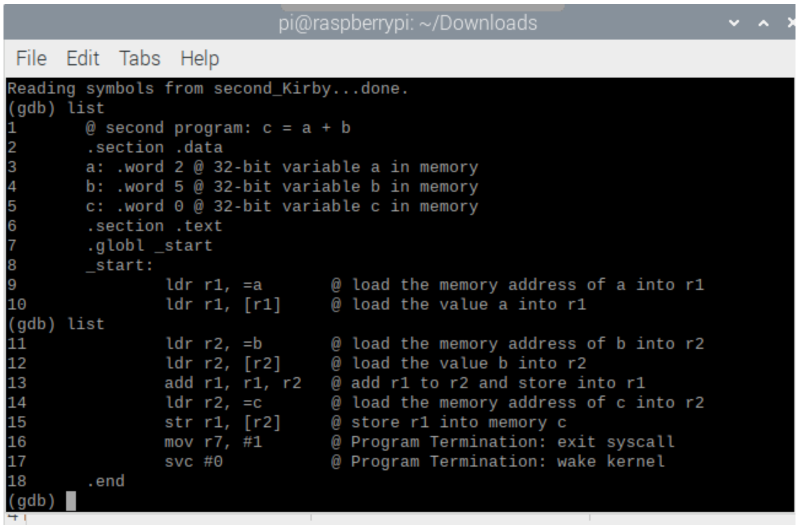
```
pi@raspberrypi:~/Downloads $ ./spmd2 4

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

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

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

pi@raspberrypi:~/Downloads $ ./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@raspberrypi:~/Downloads $
```

We see where the threads run without duplication and is fixed.

Kirby Liu

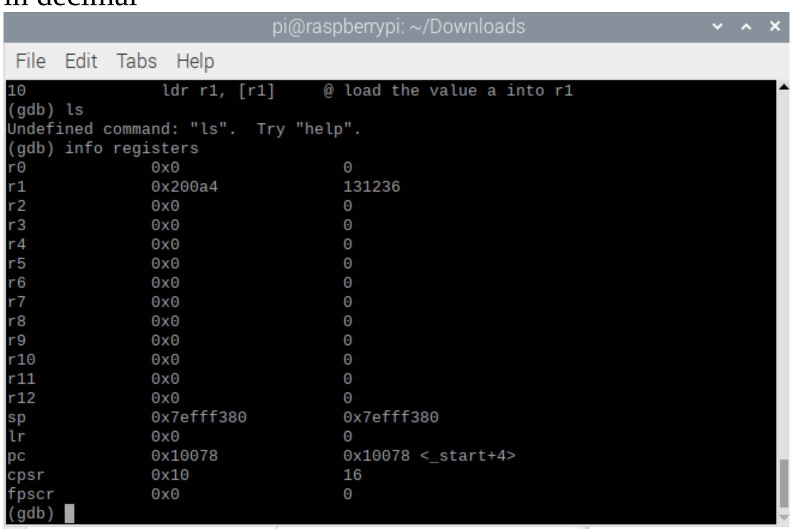
ARM Assembly Programming:

Second_Kirby.s code



```
pi@raspberrypi: ~/Downloads
File  Edit  Tabs  Help
Reading symbols from second_Kirby...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) list
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              mov r7, #1      @ Program Termination: exit syscall
17              svc #0          @ Program Termination: wake kernel
18      .end
(gdb)
```

Running a gdb on second_Kirby, I inserted a break at line “ldr r1, =a” @ load the memory address of a into r1. The memory address is stored in the register r1 at (in hex) 200a4 or 131236 in decimal



```
pi@raspberrypi: ~/Downloads
File  Edit  Tabs  Help
10              ldr r1, [r1]    @ load the value a into r1
(gdb) ls
Undefined command: "ls".  Try "help".
(gdb) info registers
r0             0x0                 0
r1             0x200a4             131236
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff380          0x7efff380
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

After loading the value of variable a and b into registers r1 and r2, we see that the memory address begins at 200a4. At the last line before the program termination, exit syscall I ran x/3dw 0x200a4 (I wanted the values of the 3 x 2 byte, words, starting at 0x200a4 in decimal form), which returned values 2, 5, and 7.

```
16              mov r7, #1       @ Program Termination: exit syscall
(gdb) info registers
r0             0x0                 0
r1             0x7                 7
r2             0x200ac             131244
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff380          0x7efff380
lr             0x0                 0
pc             0x10090             0x10090 <_start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3dw 0x200a4
0x200a4:        2       5       7
(gdb)
```

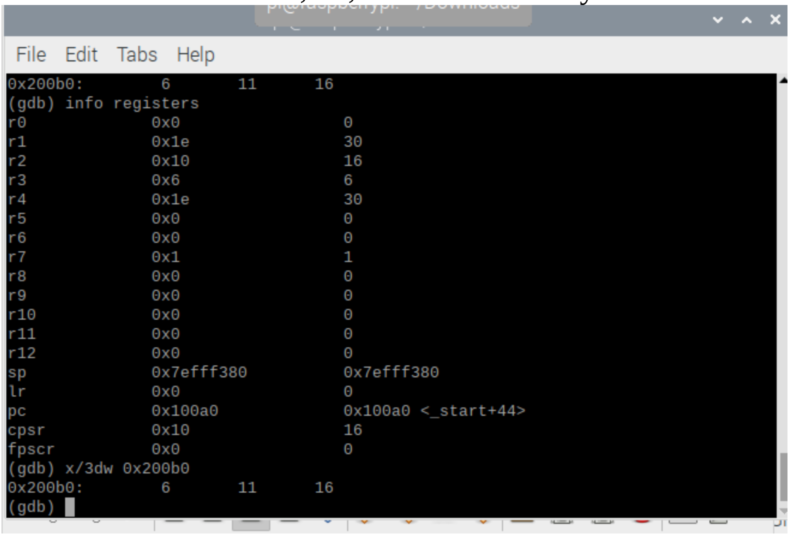
Arithmetic2_Kirby

```
pi@raspberrypi: ~/Downloads
File  Edit  Tabs  Help
@ second excercise of  second program: Register = val2 + 9 + val3 - val1
.section .data
val1: .word 6  @ 32-bit variable val1 in memory
val2: .word 11 @ 32-bit variable val2 in memory
val3: .word 16 @ 32-bit variable val3 in memory
.section .text
.globl _start
_start:
        ldr r1, =val2   @load the memory address of val2 into r1
        ldr r1, [r1]    @load the value of val2 into r1
        add r1, #9      @add 9 to val2
        ldr r2, =val3   @load the memory address of val3 into r2
        ldr r2, [r2]    @load the value val3 into r2
        add r1, r2      @add r2 to and store into r1
        ldr r3, =val1   @load memory address of val1 into r3
        ldr r3, [r3]    @load the value of val3 into r3
        sub r1, r3      @subtract r3 from and store into r1
        mov r4, r1      @store the value of r1 into r4

        mov r7, #1      @Program Termination: exit syscall
        svc #0          @ Program Termination: wake kernel
.end
arithmetic2_Kirby.s (END)
```

The Memory address of Val2 is loaded into r1, which is 0x200b4.

```
pi@raspberrypi: ~/Downloads
File  Edit  Tabs  Help

Breakpoint 1, _start () at arithmetic2_Kirby.s:10
10              ldr r1, [r1]    @load the value of val2 into r1
(gdb) info registers
r0             0x0                 0
r1             0x200b4             131252
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff380          0x7efff380
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

This would indicate that val1 starts at 0x200b0, which is confirmed when we run x/3dw 0x200b0 the values come back 6, 11, and 16 and finally we see that the final value of 30 is stored in r4.



<u>**Thu Vo**</u>

Parallel Programming Skills: Foundations

1. **Identifying the components on the raspberry PI B+?**
   - Ethernet controller, RAM, CPU, power source connector, USB ports, HDMI port, camera connector, display connector.

2. **How many cores does the Raspberry Pi's B+ CPU have?**
   - 4

3. **List three main differences between X86 (CISC) and ARM Raspberry PI (RISC).**
   - CISC has more addressing modes and operations than RISC, but RISC has more registers.
   - CISC can access the memory directly, while RISC requires registers and uses Load/Store instructions to access the memory.
   - RISC instructions take a single clock cycle to execute, while CISC has more complex instructions and takes multiple clock cycles to execute.

4. **What is the difference between sequential and parallel computation and identify the practical significance of each?**
   - In sequential computation, a problem breaks down into a series of instructions. A single processor will execute each instruction one after another.
   - In parallel computation, a problem breaks down into smaller, subset problems that can be solved concurrently, in overlapping time periods. These subsets problem can further break down into a series of instructions, which can be solved by multiple processors at the same time.

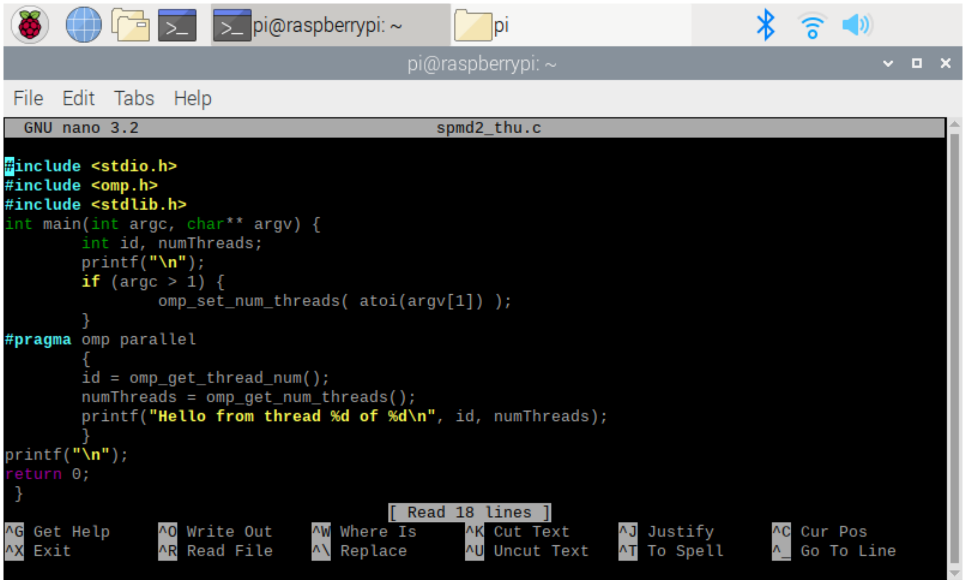
5. **Identify the basic form of data and task parallelism in computational problems.**
    - Data parallelism: Taking a large data set and dividing it up among multiple processors; the same task will be performed on the subsets of the same data. For example, a large 3D image needs to be rendered.
    - Task parallelism: A program with multiple tasks that need to be done on the same set of data, taking those tasks and dividing them among processors.

6. **Explain the differences between processes and threads.**
    - A process is the instance of a program in execution, and processes do not share a memory with each other. Thread is also known as lightweight process, and it is a segment of a process, threads share the same memory of the process they belong to.

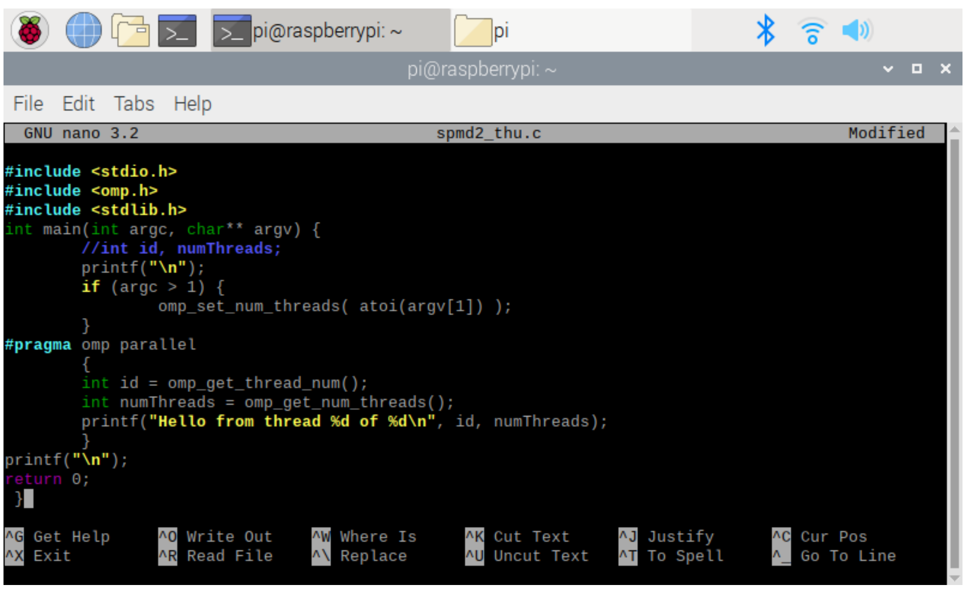
7. **What is OpenMP and what is OpenMP pragmas?**
    - OpenMP is a library that supports shared memory multiprocessing that means all threads share memory and data, and it handles tasks like thread creation and management. OpenMP pragmas is a compiler directive that enables the compiler to generate parallel execution code.

8. **What applications benefit from multi-core (list four)?**
    - Photo editing software (Adobe Photoshop), scientific simulation software (MATLAB), database servers (MySQL), and compilers (javac).

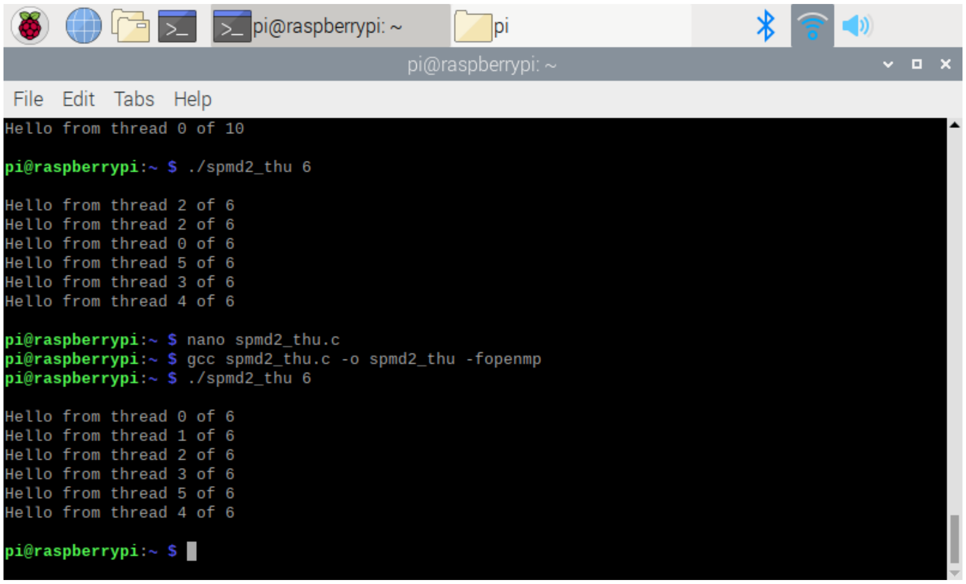
9. **Why Multicore? (why not single core, list four)**
    - It is difficult to make a single-core CPU speed to increase because we will run into problems such as increase in power consumption and heat output.
    - New applications are dealing with more complex problems; hence they are designed to support multithreading.
    - More and more computer architectures are using parallel computing.
    - A deeper pipeline has the potential to increase processor throughput, but deeply pipelined circuits generate heat problems, which is to create a solution for.



The variable id is declared before the fork, that mean all cores share the same id variable



The variable id is declared after the fork, that mean each core have a its own id variable.

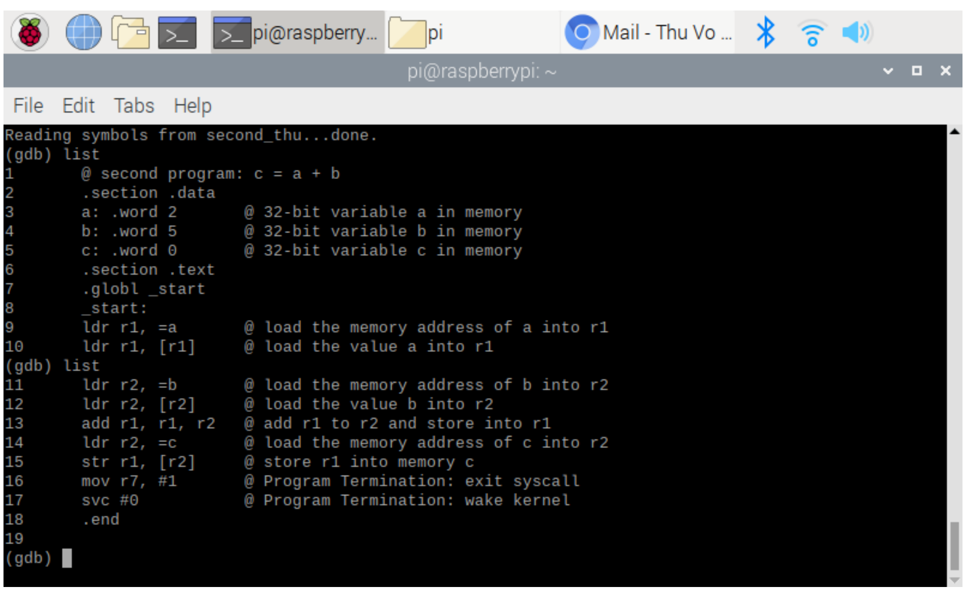


The first test is of when the cores share the same variable. Thread 1 and thread 2 try to access and update the id variable at the same time, so the update of thread 1 gets lost. The second test shows when each thread has their own local id variable, even though it is not in order, but each thread has its own unique id.

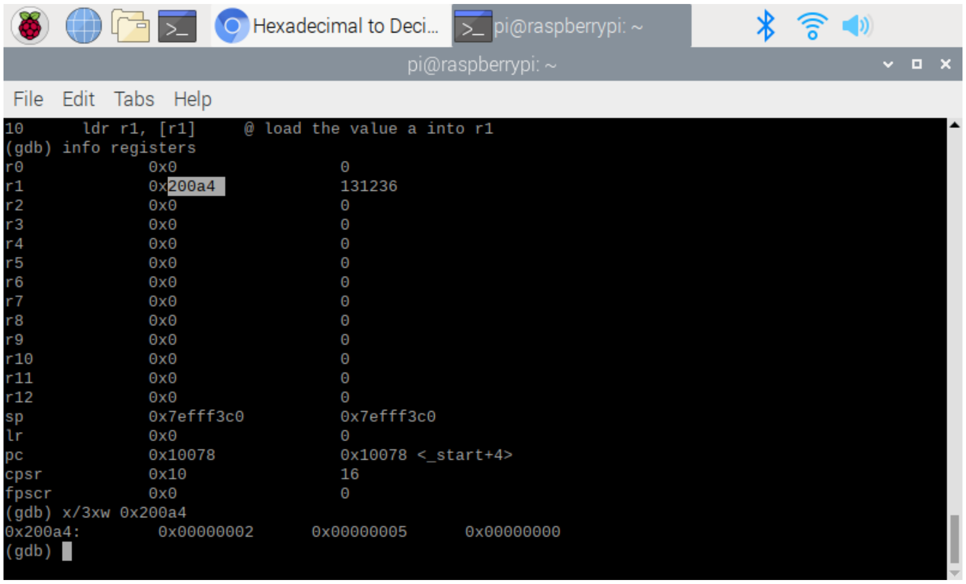
<u>Thu Vo</u>

ARM Assembly Programming

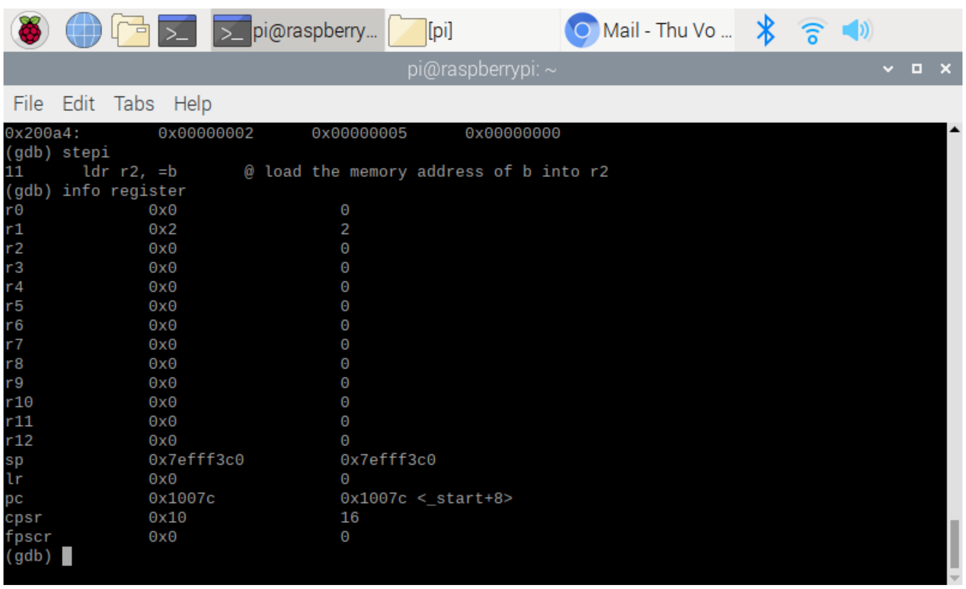
<u>Part1- second</u>



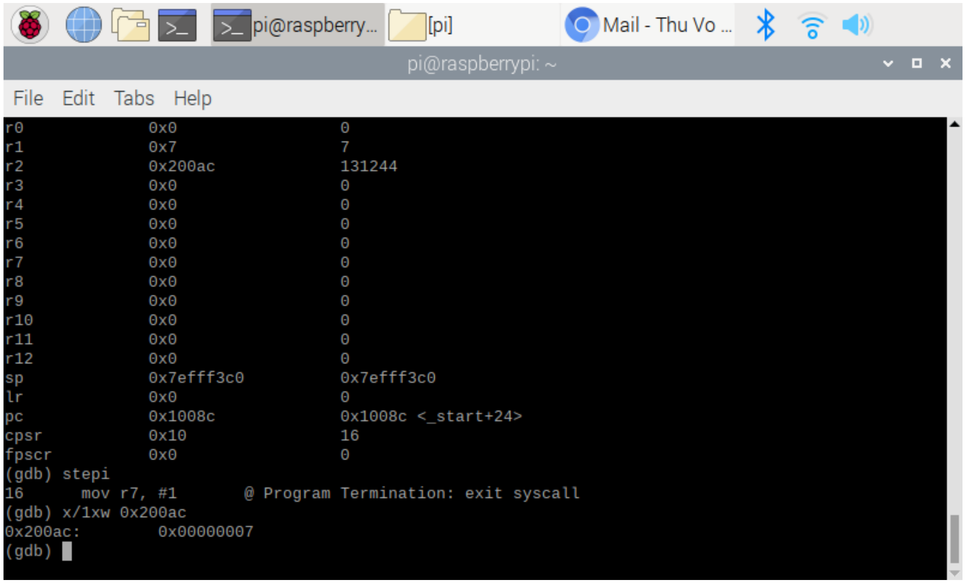
The code



The address of a is loaded to register r1. The address of a is 0x200a4, which if we look into the memory, we can see that the value of a is 2 and we can also see the value of b and c also. In ARM a word is 4 bytes, so the offset is 4. The address of a is 0x200a4, the address of b is 0x200a8, and the address of c is 0x200ac.

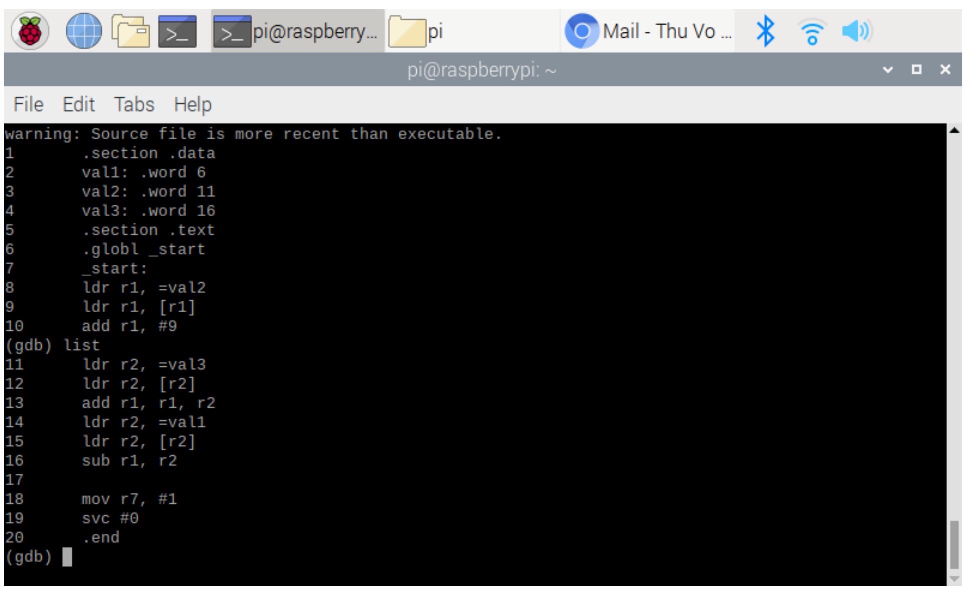


The value of a is been loaded into r1. For line 11 and 12 is the same but the program just loads the address of b to r2 then the value of b to r2.

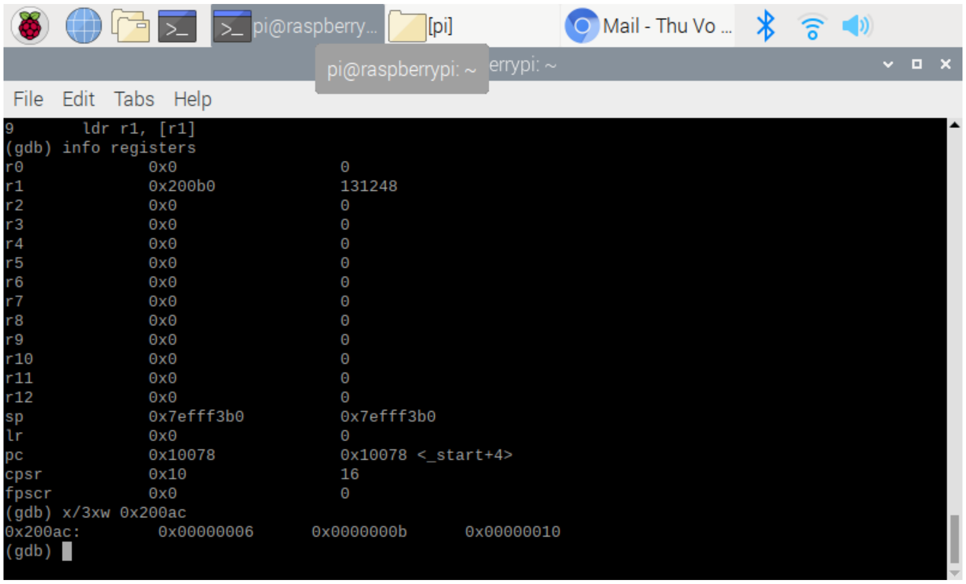


After loading both a and b to r1 and r2, the program adds r1 and r2 then stores the value in r1. Finally, the program loads the address of c into r2, then stores the value of r1 into the address in r2, which is the address of c (0x200ac). So, if we checked the memory at 0x200ac.

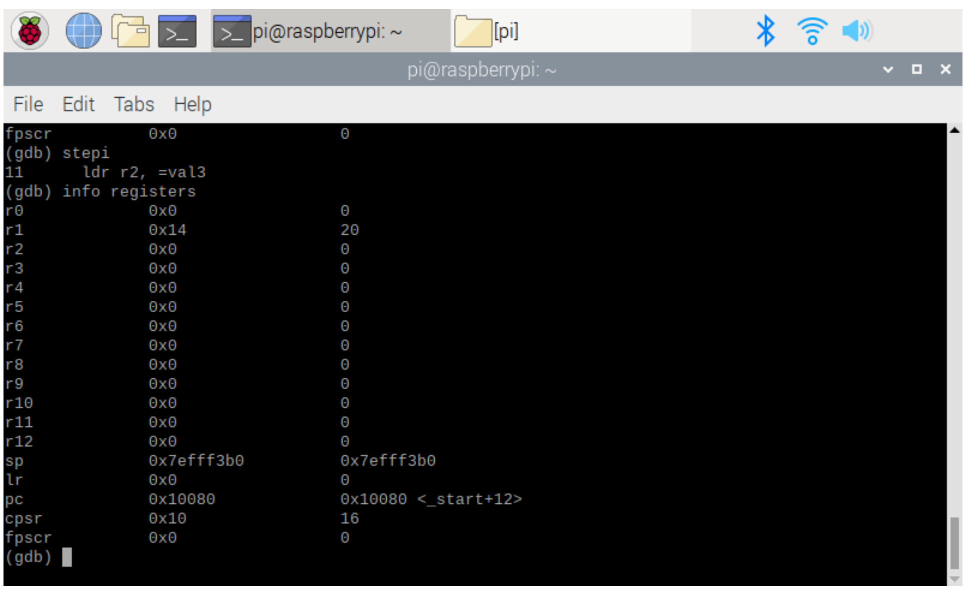
<u>Part 2 - arithmetic2</u>



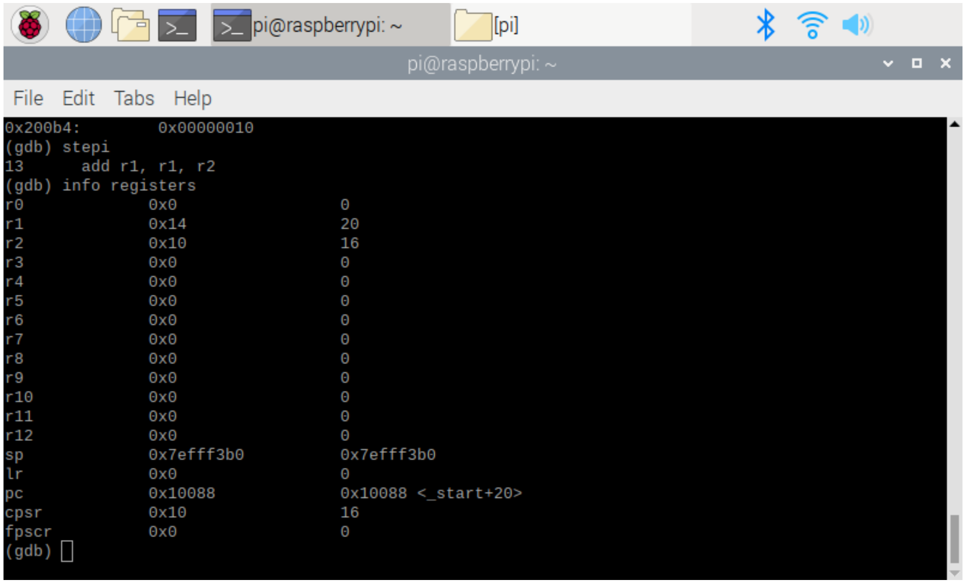
The code: Register = val2 + 9 + val3 - val1



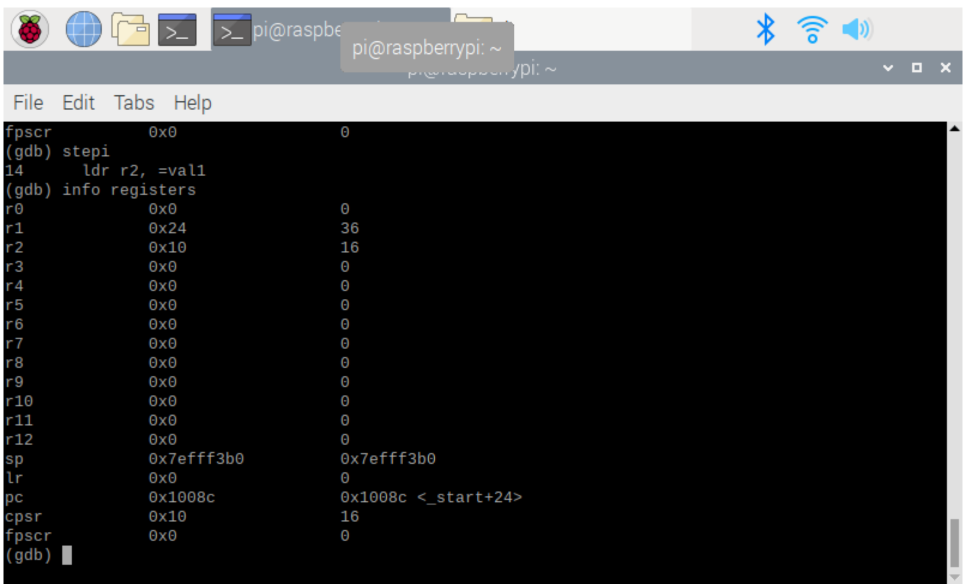
Line 8 loads val2 address in to r1. Check the memory of val1, val2, and val3. To check the memory of val1, I just subtract 4 from the address of val2 the offset of a word in ARM is 4.



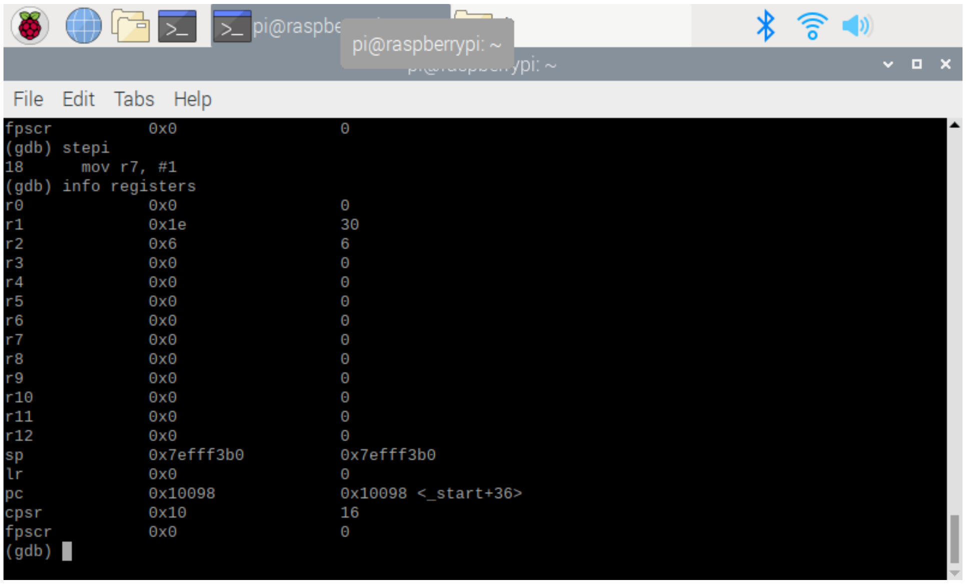
Line 9 loads the value of val 2 to r1, which makes the value of r1 is 11. Then line 10 adds 9 to r1, so value of r1 now is 20



Line 11 and 12 load the address of val3 to r2 then the value of val3 to r2.



Line 13 adds r2 to r1. Register = val2 + 9 + val3 - val1 ⇒ Register = 11 + 9 + 16 - val1 ⇒ Register = 36 - val1.



Line 14 and 15 load the address of val1 to r2 then the value of val1 to r2. Line 16 subtract r2 from r1. Register = 36 - val1⇒ Register = 36 - 6⇒ Register = 30. The final result is store in r1.

<u>**Humaira Ridi**</u>

Parallel Programming Skills: Foundation

1. **Identifying the components on the raspberry PI B+**
    - Display, Power, CPU/RAM, HDMI, Camera, Ethernet Controller, Ethernet, USB

2. **How many cores does the Raspberry Pi's B+ CPU have?**
    - 4

3. **List three main differences between X86 (CISC) and ARM Raspberry PI (RISC). Justify you answer and use your own words (do not copy and paste)**
    - X86 allows for more operations and addressing modes than ARM Raspberry PI
    - ARM contains more registers in its processor than x86
    - CISC processors use the little-endian format while ARM RISC processors have gone the BI-endian route, which supports the switching of endian.

4. **What is the difference between sequential and parallel computation and identify the practical significance of each?**
    - During sequential computing, a problem is solved after instructions are consecutively executed. In parallel computing, however, parts of a problem can be solved at the same time since instructions from each part of the problem is able to execute simultaneously on separate processors. This allows problems to be solved more quickly during parallel computing.

5. **Identify the basic form of data and task parallelism in computational problems.**
    - Data parallelism is when the amount of parallelism corresponds with the input size because multiple data items can be traced back to the same computation. Parallelism, on the other hand, is designed to execute around the functions, instead of the data.

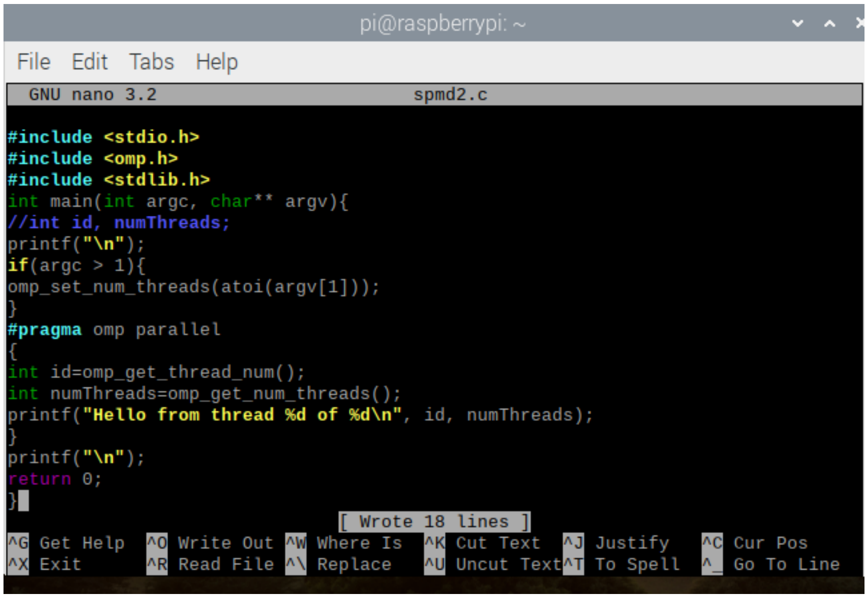
6. **Explain the differences between processes and threads.**
    - Threads are able to share their memory of whichever process they are associated with while processes, themselves, do not share memory at all.

7. **What is OpenMP and what is OpenMP pragmas?**
    - OpenMP uses implicit multithreading modeling that allows the library to handle the management and formation of thread. OpenMP pragmas, on the other hand, uses an explicit multithreading model to ensure that the programmer is the creator and manager of threads.

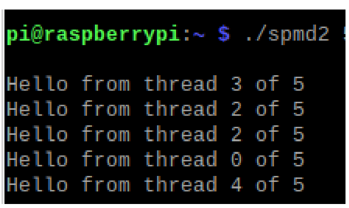
8. **What applications benefit from multi-core (list four)?**
    - Compilers, Multimedia applications, Scientific applications, Web servers

9. **Why Multicore? (why not single core, list four)**
    - Parallelism is becoming the norm in computer architecture
    - New applications are designed with multithreading models
    - Pipelined circuits cause many problems
    - Single-core clock frequencies take more time to design

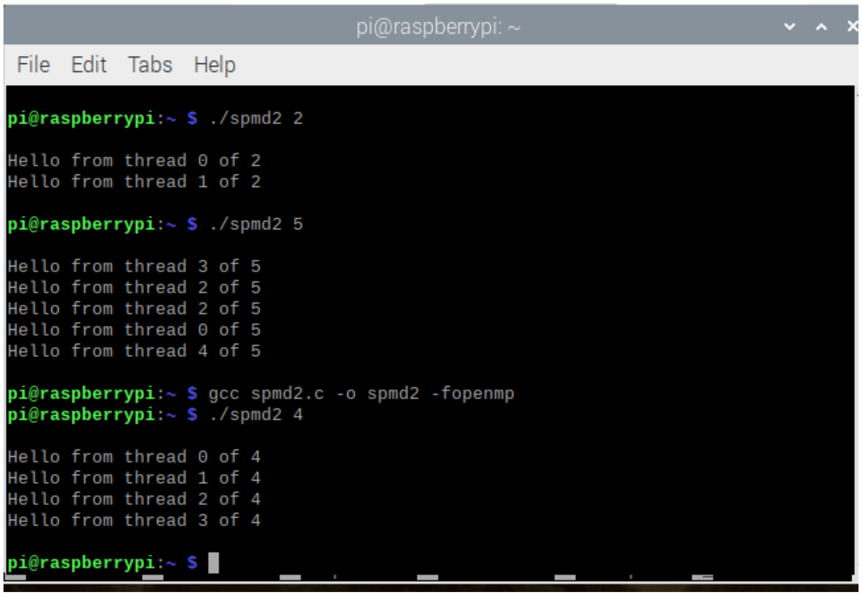
Parallel Programming Basics



In the original given code, int id, numThreads was not commented out. Because of this, it attempted to declare 2 variables that were not located within the parameters that are going to be forked.

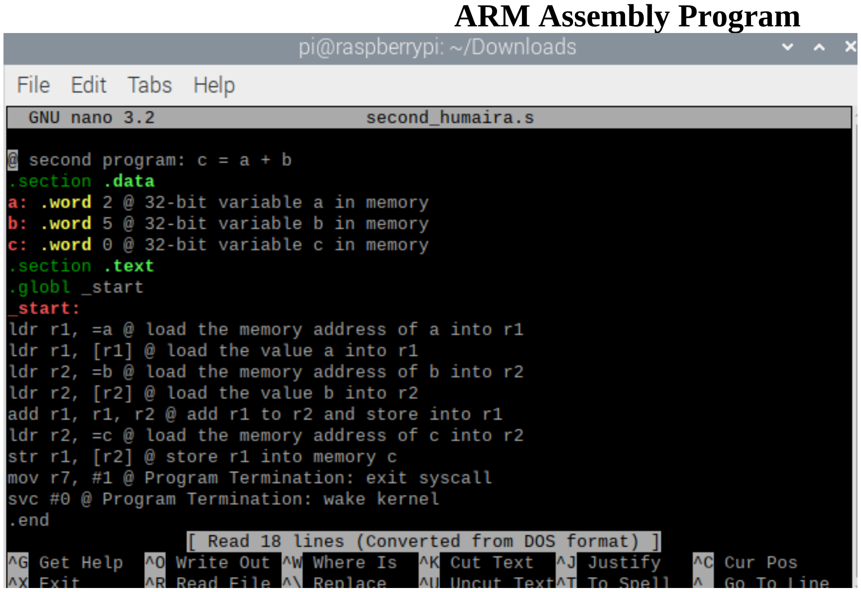


That caused the threads to share the memory location of the variables, hence the repeating of threads, as shown above.



After the code was fixed and int id, numThreads was commented out, each thread could now access its own id separately with their own copy of variables id and numThreads, as seen in the lower half of the above screenshot.

<u>Humaira Ridi</u>
ARM Assembly Program



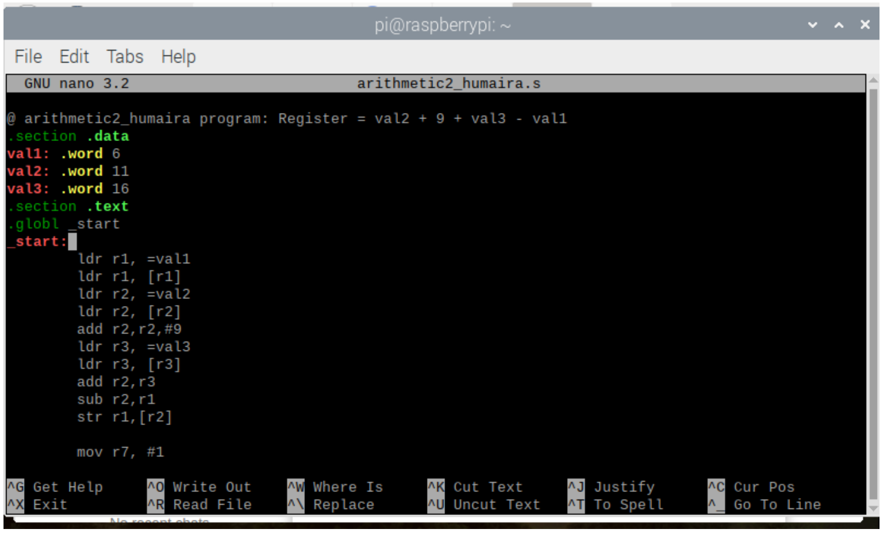
The program "second_humaira.s" solves the expression "c = a + b", where a=2, b=5, and c=0. We can see that, initially, the address of "a" is being loaded into register r1 using "ldr." To load the actual value of a into r1, brackets must be used (ldr r1, [r1]). The same condition applies for variable b and register r2. Then the value of r2 is being added to r1 which is then being stored into r1. After the memory address of c is loaded into r2, r2 is stored inside r1.

```
pi@raspberrypi: ~
File  Edit  Tabs  Help

Breakpoint 1, start () at second_humaira.s:15
15      mov r7, #1
(gdb) info register
r0             0x0                 0
r1             0x7                 7
r2             0x200ac             131244
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10090             0x10090 <start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb)
```

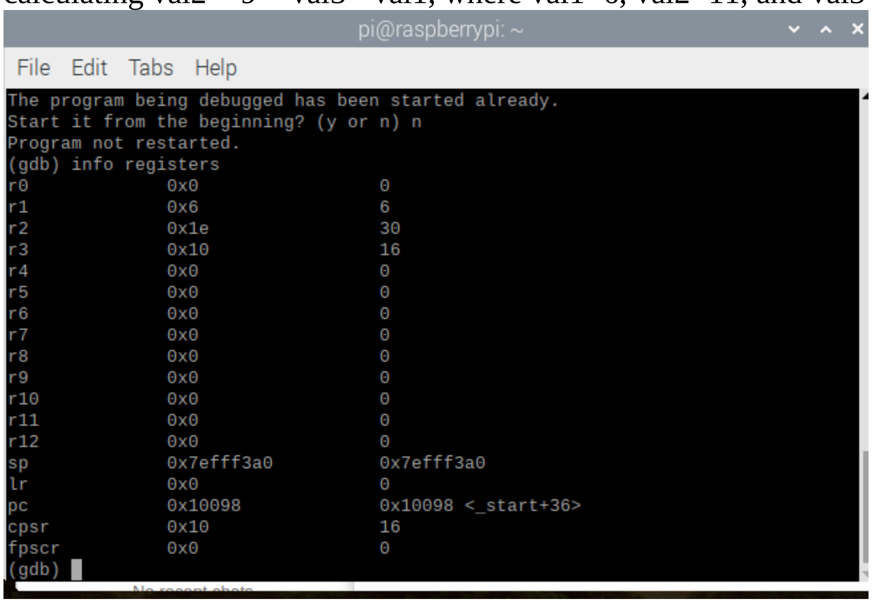
After registers are displayed, the location of r2 can be seen as “0x200ac”

```
pi@raspberrypi: ~
File  Edit  Tabs  Help
r0             0x0                 0
r1             0x7                 7
r2             0x200ac             131244
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3b0          0x7efff3b0
lr             0x0                 0
pc             0x10090             0x10090 <start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3xw 0x8054
0x8054: Cannot access memory at address 0x8054
(gdb) x/3xw 0x200ac
0x200ac:        0x00000007      0x00001141      0x61656100
(gdb)
```

Part 2:



In the program “arithmetic2_humaira.s,” we were asked to find the value of a register after calculating val2 + 9 + val3 - val1, where val1=6, val2=11, and val3=16.



After the registers were displayed, we can see the result of the expression, the decimal value 30, is stored in register r2. Register r3 has value 16 since it was not changed. Register r1 has value 6 since it was also not changed.

<u>Dhananjay Khazanchi</u>

<u>Part 1Parallel Programming Skills:</u>

1. **Identifying the components on the raspberry PI B+**
   - Display, CPU/RAM, power(x2), HDMI, camera, ethernet controller, USB(x2), and ethernet

2. **How many cores does the Raspberry Pi’s B+ CPU have?**
   - The raspberry pi has 4 cores.

3. **List three main differences between X86 (CISC) and ARM Raspberry PI (RISC). Justify your answer and use your own words (do not copy and paste)**
    - Instruction set – x86 has a complex instruction set which contains a far larger number of instructions. ARM has a very small instruction set compared to x86 with less than 100 instructions.
    - Architecture – x86 utilizes a register/memory architecture, which means that operations can occur between registers and the memory directly. ARM uses a register/register architecture, so to use the memory, the memory value must first be loaded onto a register.
    - Registers – Due to x86 having a more complex instruction set, fewer registers are present in the CPU than a device, which uses ARM Assembly.

4. **What is the difference between sequential and parallel computation and identify the practical significance of each?**
    - In sequential computation, a program is broken down into a series of instructions that can only be executed one at a time, whereas in parallel computation, it is broken down into parts that can be solved at the same time.

5. **Identify the basic form of data and task parallelism in computational problems.**
    - Data parallelism is where the same computation is applied to multiple pieces of data. The amount of possible parallelism increases proportionally with the number of data items.
    - Task parallelism is where the data is separated into different categories that have different functions performed on them at the same time. Task parallelism is much more useful when the data has more context, but it does not scale as much as data parallelism.

6. **Explain the differences between processes and threads.**
    - A process is an abstraction of a running program. A single-core can only operate on one process at a time.
    - Threads are a lightweight process that share the memory of the process they belong to. Threads allows processes to be broken down into smaller pieces.

7. **What is OpenMP and what is OpenMP pragmas?**
    - OpenMP pragmas are compiler directives that enable the compiler to generate threaded code
    - OpenMP is an interface that allows for implicit multithreading creation, meaning that the library will handle thread creation and thread management.

8. **What applications benefit from multi-core (list four)?**
    - Database servers, web servers, multimedia applications, and compilers

9. **Why Multicore? (why not single core, list four)**
    - Many stand-alone computers today are parallel from a hardware perspective
    - A multicore CPU is necessary for more complex tasks because a single core CPU cannot run fast enough to keep up with a multicore CPU.

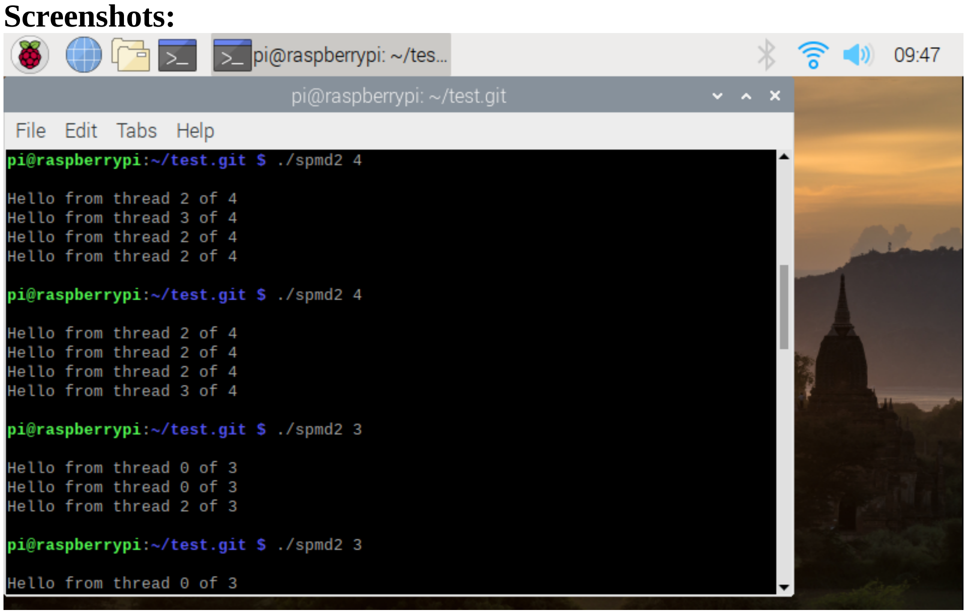
- Most new applications are multi-threaded, so they would not be able to run on a single core
- Single cores run into heat, speed of light, design, and verification problems

Dhananjay Khazanchi
Parallel Programming Basics

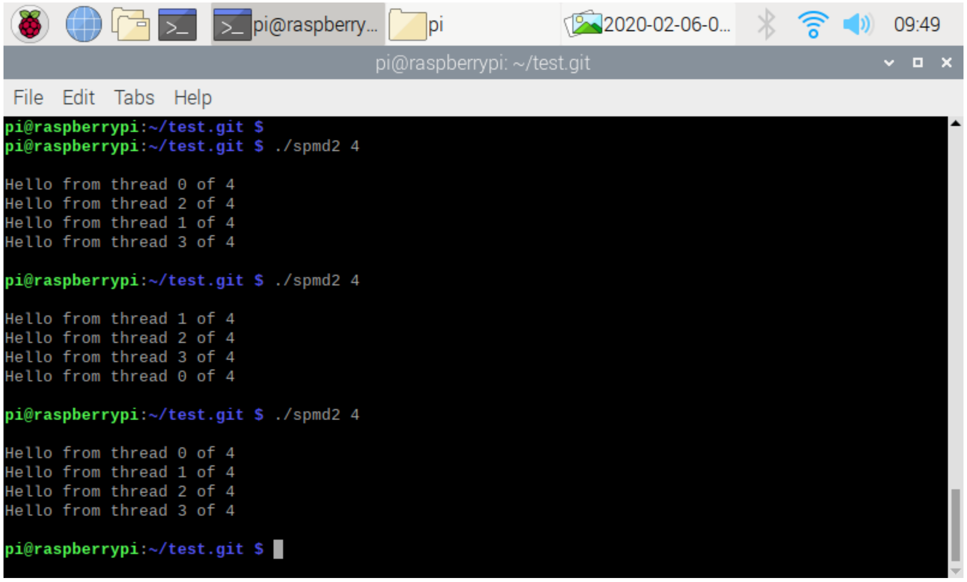
Code:

```c
#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;
}
```

Screenshots:



In the initial version of this program, the id and numThreads was being declared in the main method of the code. This was causing the same thread to do multiple operations, which is not parallel.



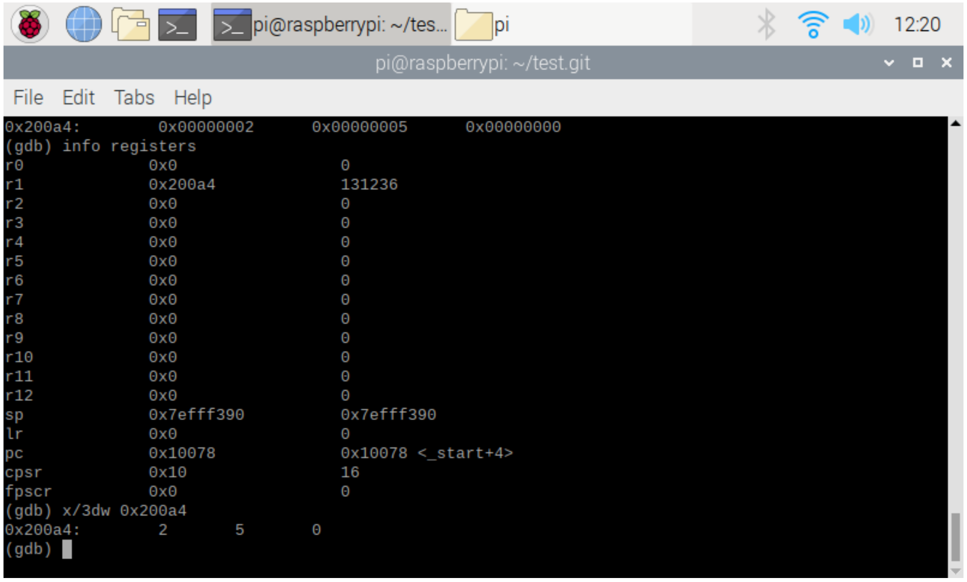
After changing id and numThreads to being declared in the omp parallel body, different threads were being used for the individual lines. As can be seen, no threads were used more than once. The threads are not in order from low to high, but that is irrelevant to the functionality.

<u>Dhananjay Khazanchi</u>
Arm Assembly Programming

Part 1 Code:

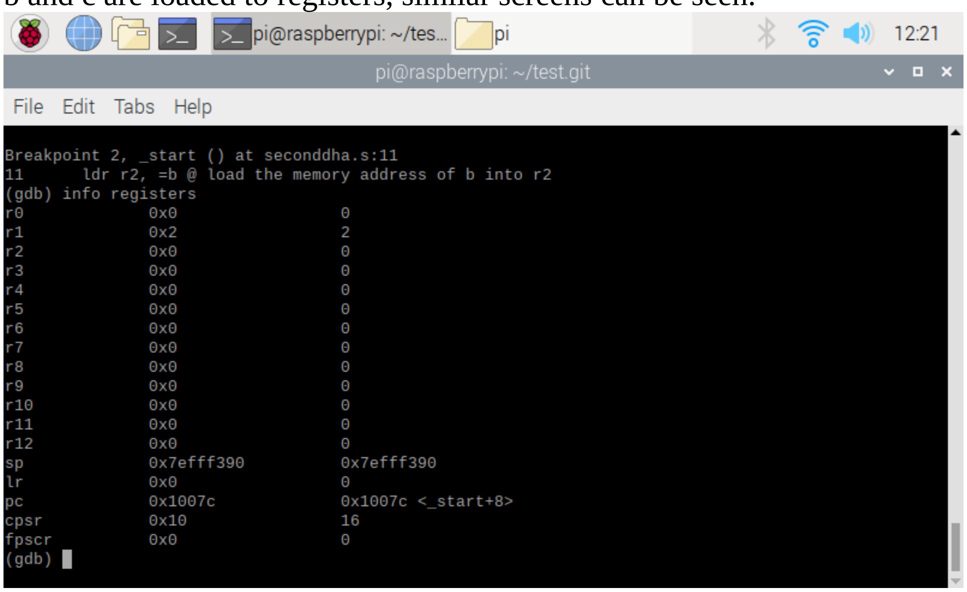
```
@ second program:  = a + b
.section .data
a: .word 2 @ 32-bit variable a in memory
b: .word 5 @ 32-bit variable b in memory
c: .word 0 @ 32-bit variable c in memory
.section .text
.globl _start
_start:
ldr r1, =a @ load the memory address of a into r1
ldr r1, [r1] @ load the value a into r1
ldr r2, =b @ load the memory address of b into r2
ldr r2, [r2] @ load the value b 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] @ store r1 into memory c
mov r7, #1 @ Program Termination: exit syscall
svc #0 @ Program Termination: wake kernel
.end
```

Part 1 Screenshots:



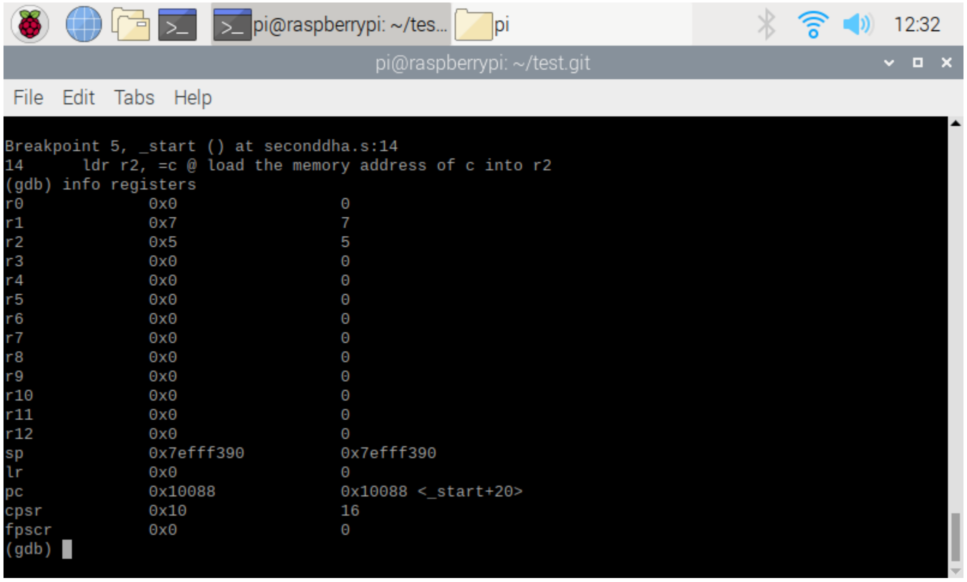
Step 1: ldr r1 = a

This step loads the memory address into r1. The middle column is the memory address in hexadecimal, and the last column is the value in decimal. At the bottom of the screen, the values stored in the memory address beginning at 0x200a4 can be seen. When the memory addresses of b and c are loaded to registers, similar screens can be seen.



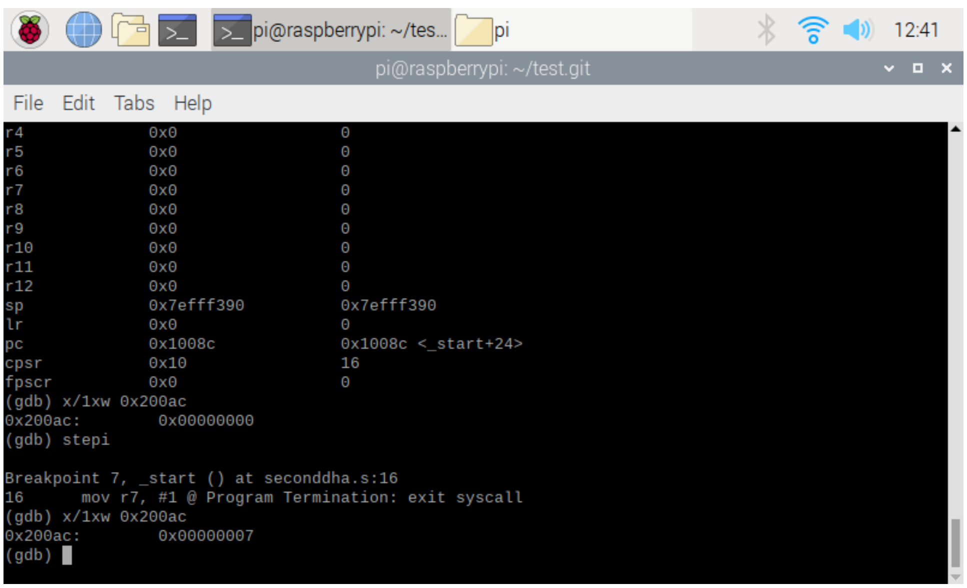
Step 2: ldr r1, [r1]

This step loads the value that was located at the memory address 0x200a4 into r1. This has to be done because ARM is based on a register/register architecture, so the memory address and then the value is loaded to r1 to get around it.



Step 3: add r1, r1, r2

In this step, the value in r2 was added to r1 and was then stored in r1.



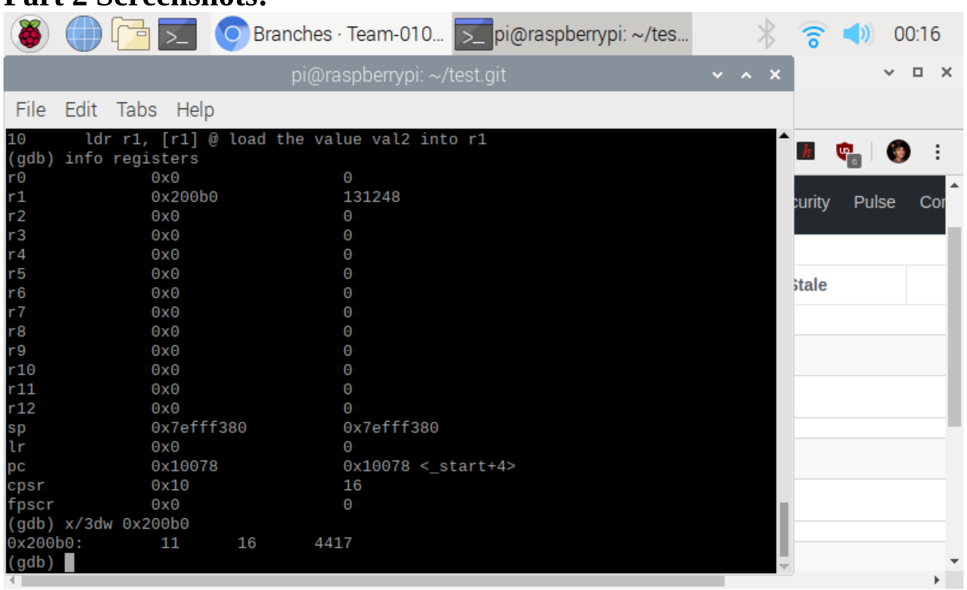
Step 4: str r1, [r2]

This step stores the contents of r1 to the memory address of c, which was previously loaded onto r2.

Part 2 Code:

```
@ Register = val2 + 9 + val3 - val1
.section .data
val1: .word 6 @ 32-bit variable val1 in memory
val2: .word 11 @ 32-bit variable val2 in memory
val3: .word 16 @ 32-bit variable val3 in memory
.section .text
.globl _start
_start:
ldr r1, =val2 @ load the memory address of val2 into r1
ldr r1, [r1] @ load the value val2 into r1
add r1, #9 @add 9 to the value in r1
ldr r2, =val3 @ load the memory address of val3 into r2
ldr r2, [r2] @ load the value val3 into r2
add r3, r1, r2 @ add r1 to r2 and store into r1
ldr r4, =val1 @ load the memory address of val1 into r4
ldr r4, [r4] @ load the value val1 into r4
sub r5, r3, r4 @subtract the value in r4 from the value in r3 and store in r5
mov r7, #1 @ Program Termination: exit syscall
svc #0 @ Program Termination: wake kernel
.end
```

Part 2 Screenshots:

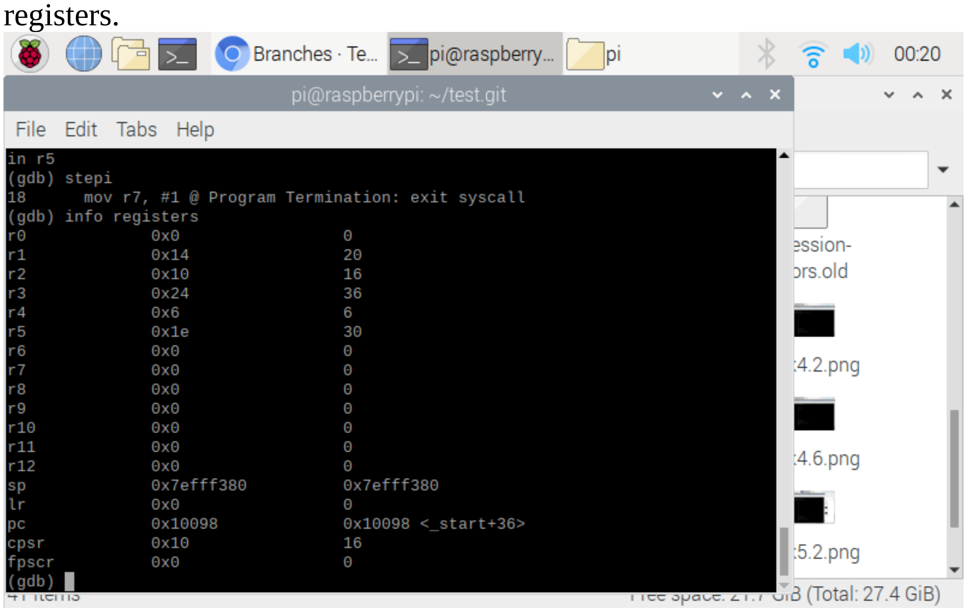


Step 1: ldr r1, =val2

This step involves loading the memory address of val2 into r1. Val2 and Val3 can be seen at the bottom of the screen as the first two elements of the memory values. Similar screens can be seen when loading the memory addresses of val1 and val3.

Step 2: ldr r1, [r1]

This step involves loading the value at the address in r1 into r1. Similar screens can be seen when the values of val2 and val3 are loaded from their respective addresses into their respective registers.



This screenshot displays the final step in the program, which was subtracting r4 from r3 and storing it in r5. This leads to the final decimal value of 30, after all arithmetic has been performed.

<u>Task 5:</u>

GitHub: https://github.com/Team-0101/CSC3210-Team0101

Slack: https://app.slack.com/client/TT7TS0KN2/CTWU0DS8P

Video: **YouTube Link:** https://youtu.be/awn5TcLhN1s