Developing Soft and Parallel Programming Skills Using Project-Based Learning
Fall 2019

Group Name: The Snakes
Group Members: Nabeeha Ashfaq, Jose Diaz, Matthew Hayes, Micah Robins, Austin Yuille

Planning and Scheduling

Name	Email	Task	Duration (hours)	Dependency
Jose Diaz	jdiaz28@student.gsu.edu	Facilitator, organized report and task 3 answers	6	Google docs
Austin Yuille	ayuille1@student.gsu.edu	Typed up ARM programming,	6	Raspberry Pi
Micah Robins	mrobins1@student.gsu.edu	Created new assignment on Github and managed it	6	Github
Matt Hayes	mhayes37@student.gsu.edu	Directed and edited video presentation	6	YouTube
Nabeeha Ashfaq	nashfaq1@student.gsu.edu	Team coordinator, organized meetings, crated task sheet and report	6	Slack, Google Docs

Parallel Programming Skills

a) Foundation:

1. Identifying the components on the raspberry PI B+
    a. Quad-Core Multi-core CPI
    b. 1GB RAM
    c. Ethernet controller
    d. Ethernet port
    e. HDMI port
    f. Micro-USB port for power
    g. 4 USB ports
    h. Display connector
    i. Camera connector
    j. 2 led lights near power connector
2. How many cores does the Raspberry Pi's B+ CPU have?
    a. 4 Cores. It is a quad-core CPU.
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)?
    a. One of the main differences between x86 and ARM is the instruction set. The instruction set of x86 is much more complex and feature rich which allows for more instructions to access memory and as such is usually the choice when it comes to personal computing.
    b. ARM has more registers than x86 that are much more general purpose. Also the instruction set is a lot simpler with the max being 100 instructions.
    c. ARM machines can be executed more quickly given that having less instructions reduces the clock cycles per instruction but due to this efficiency must be taken into account given that less instructions are available to you.
    d. Finally ARM machines use a load/store memory model for memory access which means that it can not pull and act on data straight from memory. Everything must be loaded onto the registers and then the operations can take place.
4. What is the difference between sequential and parallel computation and identify the practical significance of each?
    a. In sequential computation, a problem is broken up into a discrete series of instructions and then executed one after the other.
    In parallel computation, a problem is broken up into parts that can all be done at the same time and then broken up into a series of instructions. \
    Parallel is useful when a problem can be broken down and done side-by-side, however more processors and space are used. Serial computation is slower but more useful in tasks that require organization and a certain order.
5. Identify the basic form of data and task parallelism in computational problems.
    a. Data parallelism focuses on the distribution data sets of a problem while task parallelism focuses on the tasks needed to complete the problem.
6. Explain the differences between processes and threads
    a. Processes are the representation of a running program. Processes do not share memory with each other. Threads are smaller processes that can be broken up into

separate, independent parts that share memory. Processes are used for more demanding tasks while threads are used for computing smaller pieces of a larger problem.

7. What is OpenMP and what is OpenMP pragmas?
    a. OpenMP is a programming interface that takes a thread and breaks it down, or forks it, into multiple threads that are run independently and then joined for the final result when all of them are complete. OpenMP pragmas are directives in the compiler that allow the creation of threaded code. OpenMP pragmas use a method for creating threads that doesn't require users to manually create and manage threads.
8. What applications benefit from multi-core (list four)?
    a. Database servers
    b. Web servers, like in ecommerce
    c. Compilers
    d. Multimedia applications
    e. Scientific applications, CAD/CAM
    f. Applications with thread-level parallelism
9. Why multicore? (why not single core, list four)
    a. Difficult to make single-core clock frequencies even higher.
    b. Deeply pipelined circuits:
        i. Heat problems
        ii. Speed of light problems
        iii. Difficult design and verification
        iv. Large design teams necessary
        v. Server farms need expensive air-conditioning to keep the machines cool
    c. Many new applications are multithreaded
    d. The trend in computer architecture (shift towards more and more parallelism)

b) Parallel Programming Basics

To complete all of the programming requirements for this project, our group set up a meeting in one of the GSU Library conference rooms and hooked up the Raspberry Pi to the monitor. Once the machine was booted up we launched the terminal and created the "spmd2.c" program in nano (see Appendix C, 1). We added the provided code to it, and then saved and compiled our code to create an executable file (see Appendix C, 2). Finally, we ran the program using the command "./spmd2 4" which uses the command-line argument "4" to create four threads in the fork.

We then observed the results and noticed that some threads would have duplicate IDs each time the program was run. It was a different result each time, but always ended up with more than one thread printing identical ID numbers (see Appendix C, 3). This is a result from declaring our variables outside of the block of code that runs in parallel. Each thread is referencing the same memory for the variables, hence the overlap of ID numbers.

Our next step was to open the program in nano again and alter our code so that we could make the program run correctly. By commenting out line 5, we effectively removed the code where we originally declared the “id” and “numThreads” integers. We added the declarations for those variables to lines 12 and 13, which is inside the block of code that is threaded to multiple cores. These changes will give each thread a different copy of each variable (see Appendix C, 4).

To finish up with this part of the programming assignment, we saved, compiled, and ran the “spmd2” program. And now, each time we ran the program each thread had a distinct ID number (see Appendix C, 5).

ARM Assembly Programming

Part 1:

After completing the parallel programming task and discussing our results, we moved on to the Arm Assembly Programming straight away. Using the Raspberry Pi's terminal we created the file "second.s" using nano. We added the provided code to our program, and then saved, assembled, linked, and ran the program (see Appendix D, 1 and D, 2). Since the code does not print any results, we had to launch the GDB debugger program to see the result of our program on the registers and memory. We reassembled the program, making sure to include the "-g" flag this time. Then linked it and launched GNU debugger. A breakpoint at line 15 was added which allowed us to observe the program just before the final command of storing the contents of register r1 into variable c in the memory (see Appendix D, 3).

The task of our "second" program was to calculate the equation *c=a+b,* where a=2, b=5, and initially c was set to a value of 0. By observing the registers at our breakpoint we could see that register r1 = 7, which is exactly what we wanted to see because we stored the result of *a+b* in to register r1. Additionally, we observed that the address for the memory location of c was in register r2. Next we needed to observe what was stored in the memory location of c. We used the command "x/3xw 0x200ac" which displayed the first 3 items in hex starting at location 0x200ac. We knew that we needed to look specifically at address 0x200ac because it was in register r2 (see Appendix D, 3). We noticed that the memory for our c variable had a stored value of zero. This made sense because our breakpoint stopped the code just before writing the contents of register r1 in to that part of memory, and we had initialized c to zero at the beginning of the program.

At this point we paused for a few minutes to explore how the "x/nfs address" command worked. We tried changing the number and format to see how the displayed results were affected (see Appendix D, 4). Finally, used the "stepi" command to execute our last command in the program. We then observed the registers and also the memory of variable c. The registers were unchanged, but the memory of c (0x200ac) was now storing our result of 7 (see Appendix D, 5).

Part 2:

For part 2, we needed to write a program to calculate the expression:

Register = val2 + 9 + val3 - val1, where val1=6, val2=11, and val3=16.

Additionally, all 3 variables needed to be stored in memory. Whereas the Register value only needed to be a general purpose register, we took it one step further and created a variable in memory named "result" and stored the answer for the equation in that location in memory. Calculating the equation above, we can see the integer 30 should be the answer we stored in result.

You can see our original code in Appendix D, 6 and D, 7. We started by declaring and initializing our four variables in the .data part of the program. We then loaded registers r1, r2, and r3 with the values of our first three variables (val1, val2, and val3 respectively). Now comes the arithmetic part of the program. Our equation has 3 operations in it. There are 2 additions and a subtraction. At this point in our code r1 is storing a value of 6, r2 contains 11, and r3 contains 16. Our next line of code adds r2 with the immediate value of 9, and stores the result in r2. Then we add r2 to r3 and store it in r2. Finally, we subtract r1 from r2 and store it in r2. So now r2 should be storing our final answer of 30. But we wanted to go a little further with our program and store that result in a memory location (the variable "result" that we declared earlier). However, if you look at Appendix D, 7 the highlighted part shows that we had our command for storing the answer in to memory incorrect (which we eventually fixed).

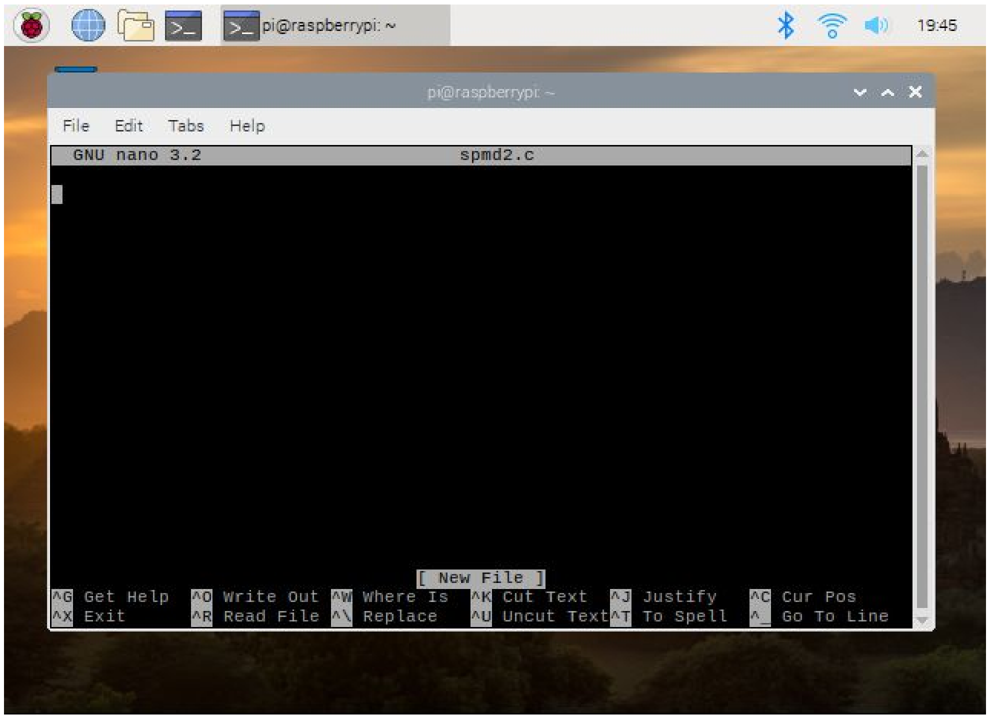
At this point we attempted to assemble our program, but ran in to a few errors in the process (Appendix D, 8 shows the errors). The first error we dealt with was "unknown preudo-op: '.section.text'." It was a simple fix because it was a typo. Just added a space between ".section" and ".text." The next error was a syntax error in the command to assemble our arithmetic2 program. It was a problem with the flags we used. Another simple fix. At this point we were able to assemble and link our program, but encountered another error when trying to run the program: "Segmentation fault." I think we all let out a collective "Uh Oh" when we saw this. But, knowing that a segmentation fault applies to accessing memory, we were able to pinpoint pretty quickly that it had something to do with our attempt to store the result from a register to our "result" memory location. We looked back at part1 of this assignment and compared it to our code. We realized that we had reversed the registers in the "str" command at the end of our program. Once we fixed this, our program assembled, linked, and ran properly (see Appendix D, 9 and D, 10 for the corrected code).

If you look at Appendix D, 11 you can see the result of our code. Register r2 (which originally was loaded with the value of val2: 11) now contains the answer to the equation: 30. Or in hex, 1E. Additionally, if you look at register r4 you can see the memory address for our variable "result" is 200C4h. At the bottom of the screen we printed the contents of that memory, which is 0000001E in hex, or 30 in decimal. We are very happy with our results , as everything appears to have worked exactly as we expected.

Appendix A:Links
Github Project: https://github.com/orgs/thesnakes-csc3210/projects/1
Github Repository:https://github.com/thesnakes-csc3210/ProjectA2
Video Presentation: https://www.youtube.com/watch?v=GHltgs3Ao34&feature=youtu.be

Appendix C: Screenshots for Parallel Programming Basics

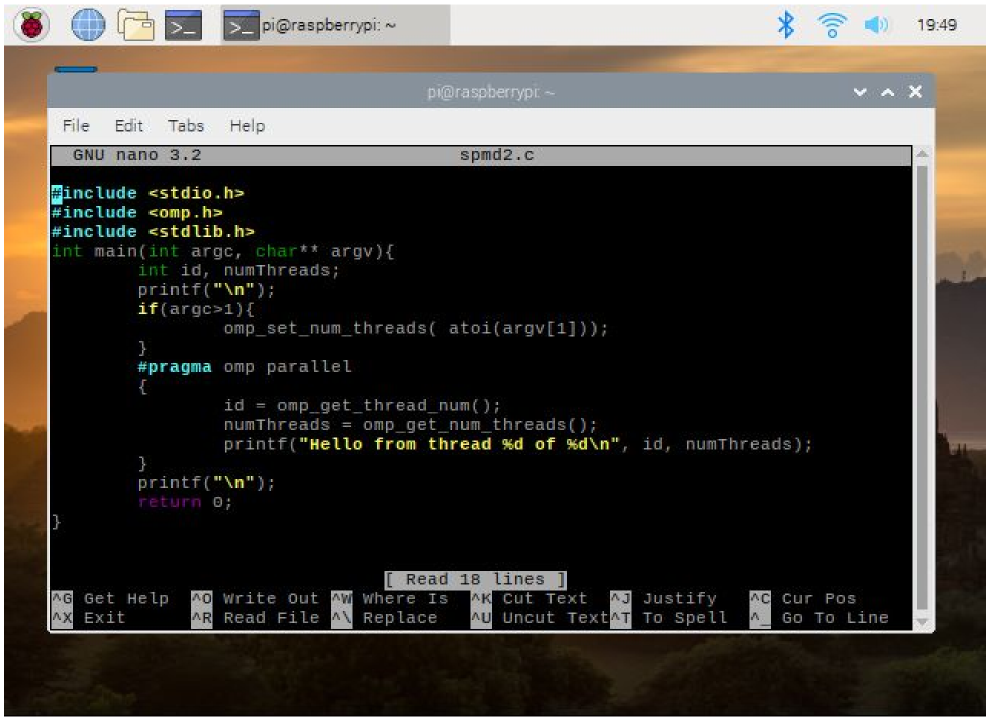
(C, 1)



```
  GNU nano 3.2                              spmd2.c



                                     [ New File ]
^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
```

The file spmd2.c when first created

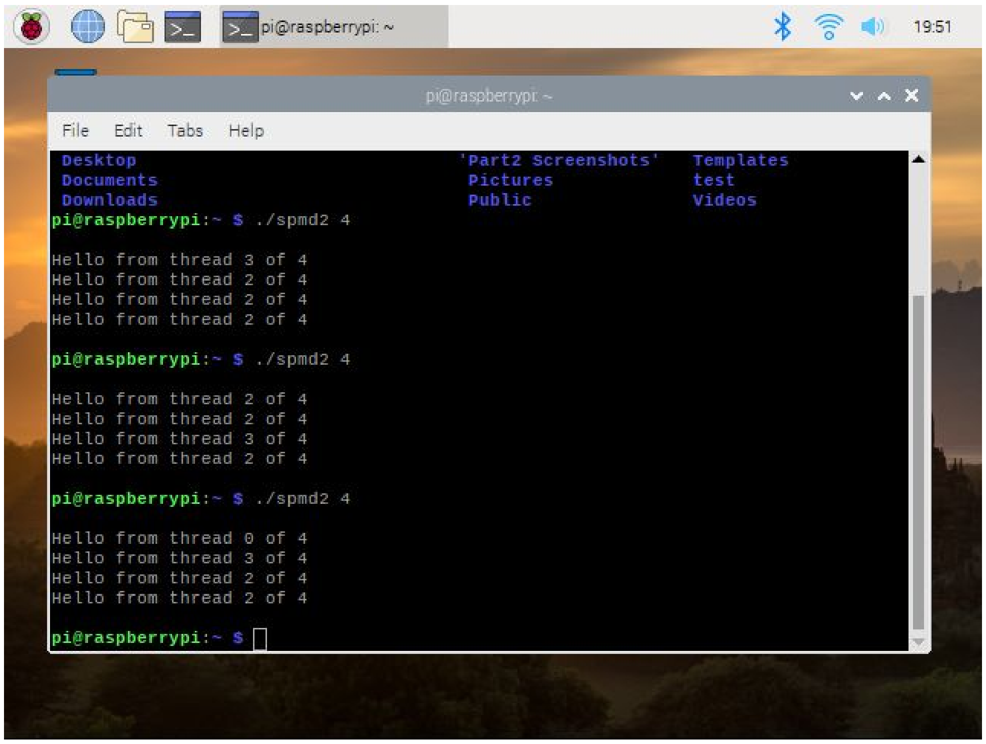
(C, 2)



```
  GNU nano 3.2                              spmd2.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
        {
                id = omp_get_thread_num();
                numThreads = omp_get_num_threads();
                printf("Hello from thread %d of %d\n", id, numThreads);
        }
        printf("\n");
        return 0;
}

                                  [ Read 18 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
```

The complete file, spmd2.c, before correcting the variable declaration issue.

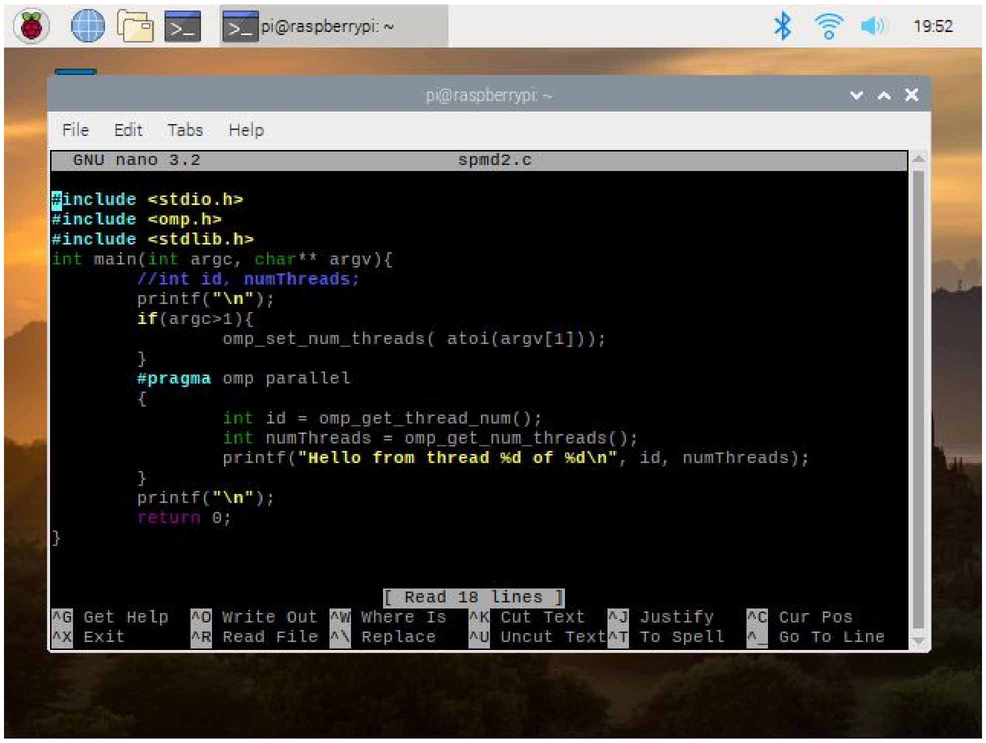
(C, 3)



```
 Desktop                                      'Part2 Screenshots'   Templates
 Documents                                     Pictures             test
 Downloads                                     Public               Videos
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 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 0 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ 
```

The flawed output of the uncorrected spmd2.c.

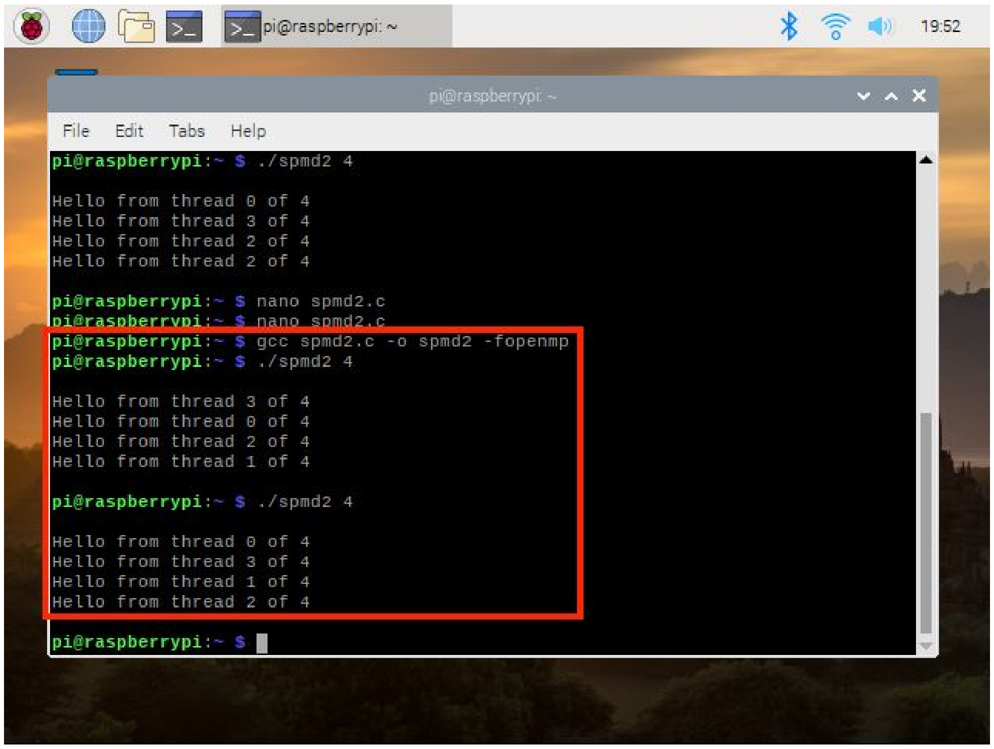
(C, 4)



```
  GNU nano 3.2                                spmd2.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;
}

                                [ Read 18 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
```

The corrected version of spmd2.c, where the variables are now declared in the pragma.

(C, 5)

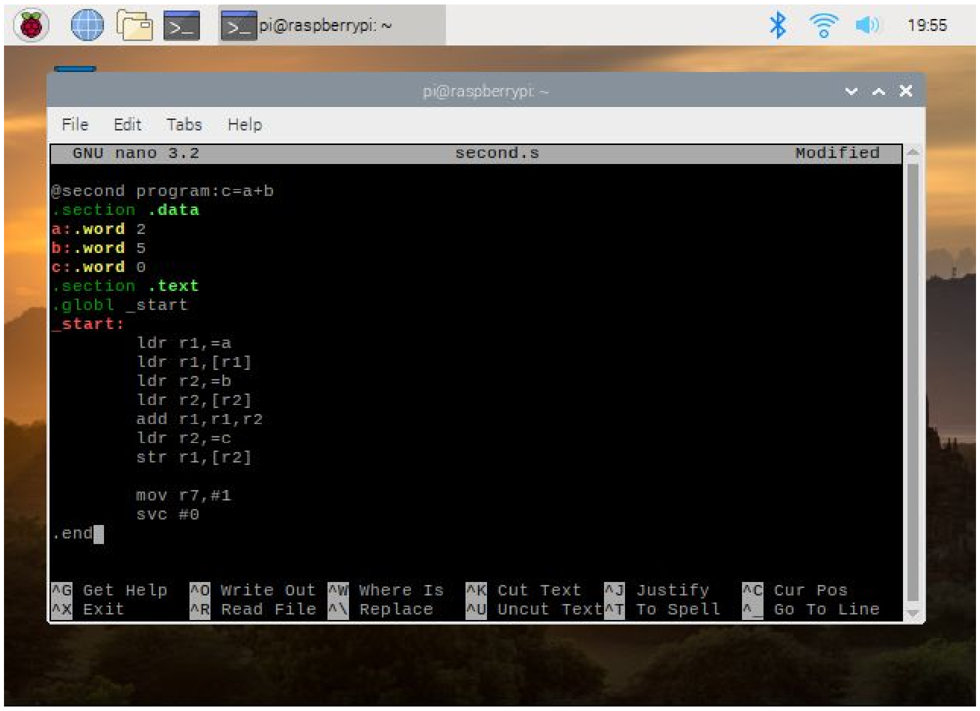


```
pi@raspberrypi:~ $ ./spmd2 4

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

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

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

pi@raspberrypi:~ $ ./spmd2 4

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

pi@raspberrypi:~ $
```

The new, expected output of the file spmd2.c.

Appendix D: Code Screenshots for ARM Assembly Programming

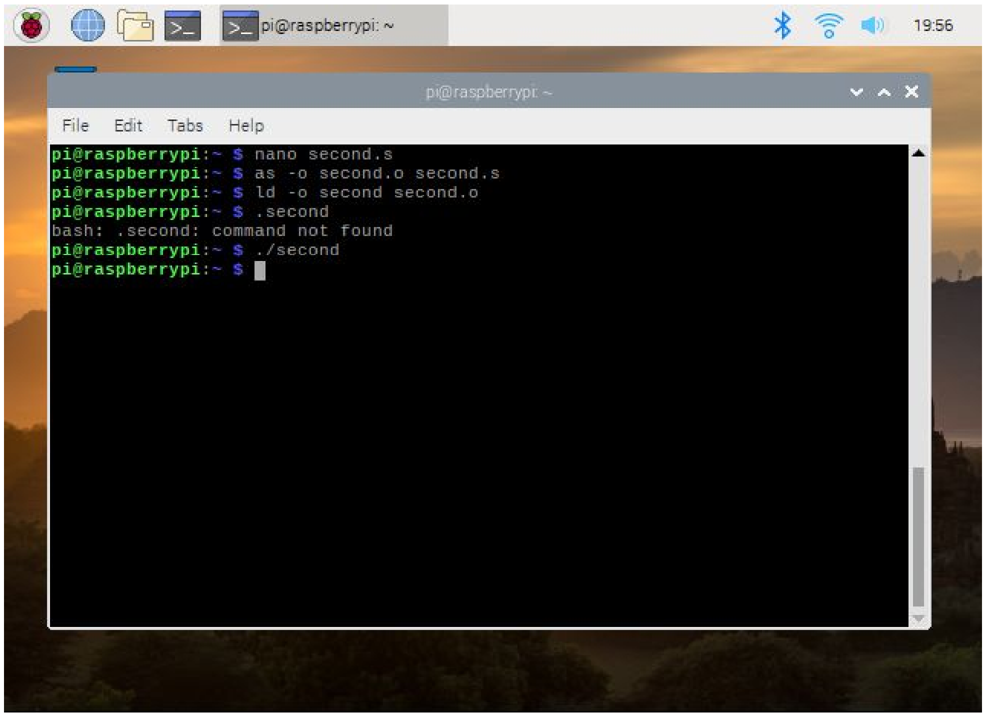
(D, 1)



```
  GNU nano 3.2                          second.s                         Modified

@second program:c=a+b
.section .data
a:.word 2
b:.word 5
c:.word 0
.section .text
.globl _start
_start:
        ldr r1,=a
        ldr r1,[r1]
        ldr r2,=b
        ldr r2,[r2]
        add r1,r1,r2
        ldr r2,=c
        str r1,[r2]

        mov r7,#1
        svc #0
.end

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text^T To Spell     ^_ Go To Line
```

The file second.s.

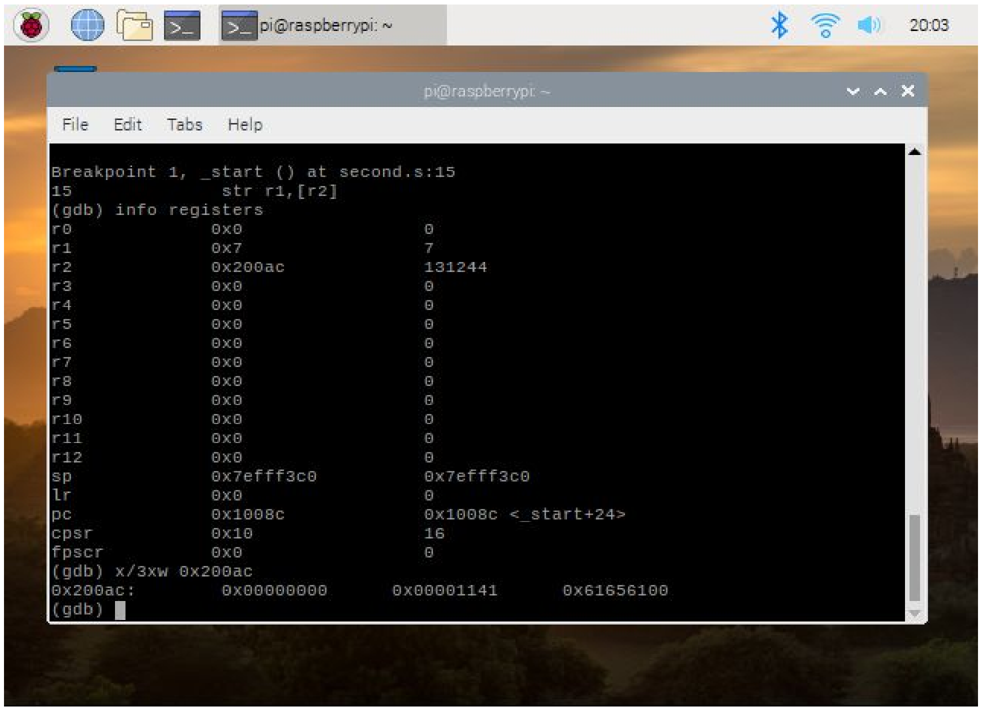
(D, 2)



```
pi@raspberrypi:~ $ nano second.s
pi@raspberrypi:~ $ as -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ .second
bash: .second: command not found
pi@raspberrypi:~ $ ./second
pi@raspberrypi:~ $ 
```

After running second.s there was no output.

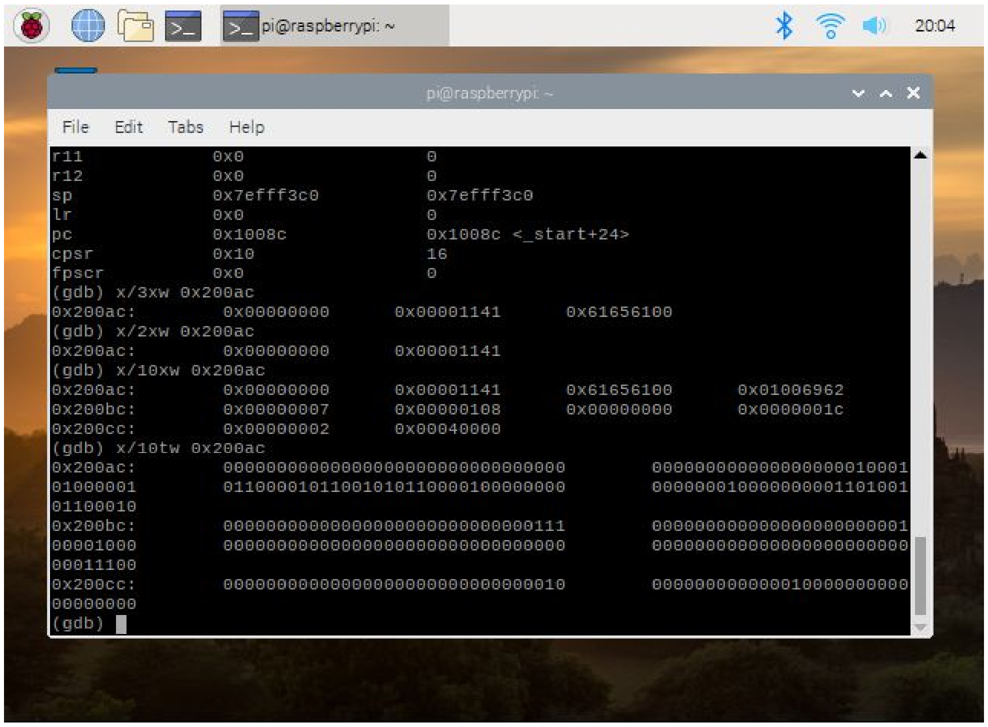
(D, 3)



```
Breakpoint 1, _start () at second.s:15
15              str r1,[r2]
(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             0x7efff3c0          0x7efff3c0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3xw 0x200ac
0x200ac:        0x00000000      0x00001141      0x61656100
(gdb) 
```

The register values of second.s after the execution of line 14. Register r2 contains the address 0x220ac.

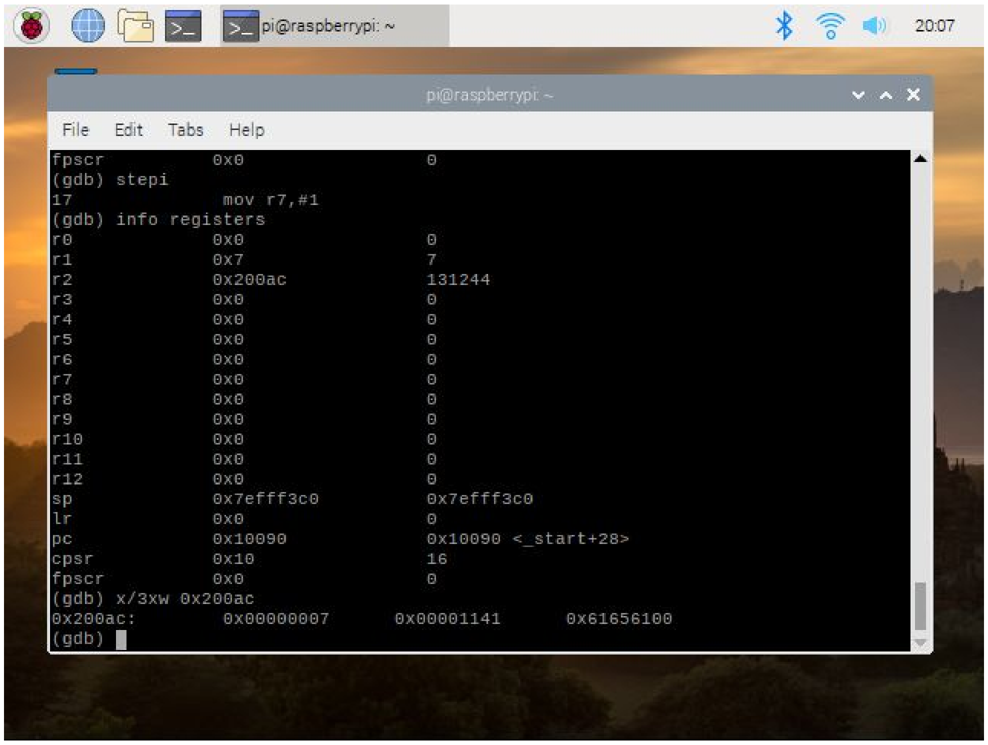
(D, 4)



```
r11            0x0                 0
r12            0x0                 0
sp             0x7efff3c0          0x7efff3c0
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3xw 0x200ac
0x200ac:        0x00000000      0x00001141      0x61656100
(gdb) x/2xw 0x200ac
0x200ac:        0x00000000      0x00001141
(gdb) x/10xw 0x200ac
0x200ac:        0x00000000      0x00001141      0x61656100      0x01006962
0x200bc:        0x00000007      0x00000108      0x00000000      0x0000001c
0x200cc:        0x00000002      0x00040000
(gdb) x/10tw 0x200ac
0x200ac:        00000000000000000000000000000000        000000000000000000010001
01000001        01100001011001010110000100000000        000000010000000001101001
01100010
0x200bc:        00000000000000000000000000000111        000000000000000000000001
00001000        00000000000000000000000000000000        000000000000000000000000
00011100
0x200cc:        00000000000000000000000000000010        000000000000010000000000
00000000
(gdb) 
```

Shows different memory values starting at 0x200ac. We changed the values to see the different outputs and understand how this command worked.

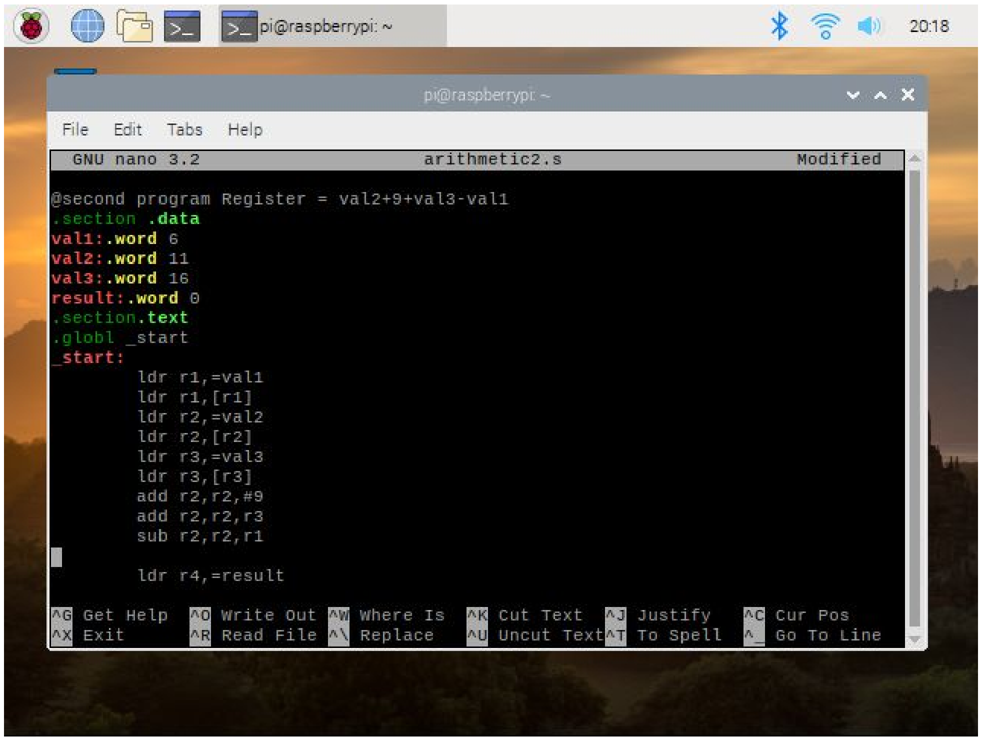
(D, 5)



```
fpscr          0x0                 0
(gdb) stepi
17              mov r7,#1
(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             0x7efff3c0          0x7efff3c0
lr             0x0                 0
pc             0x10090             0x10090 <_start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3xw 0x200ac
0x200ac:        0x00000007      0x00001141      0x61656100
(gdb) 
```

The value of r1, 7, is now stored in the memory at address 0x200ac.

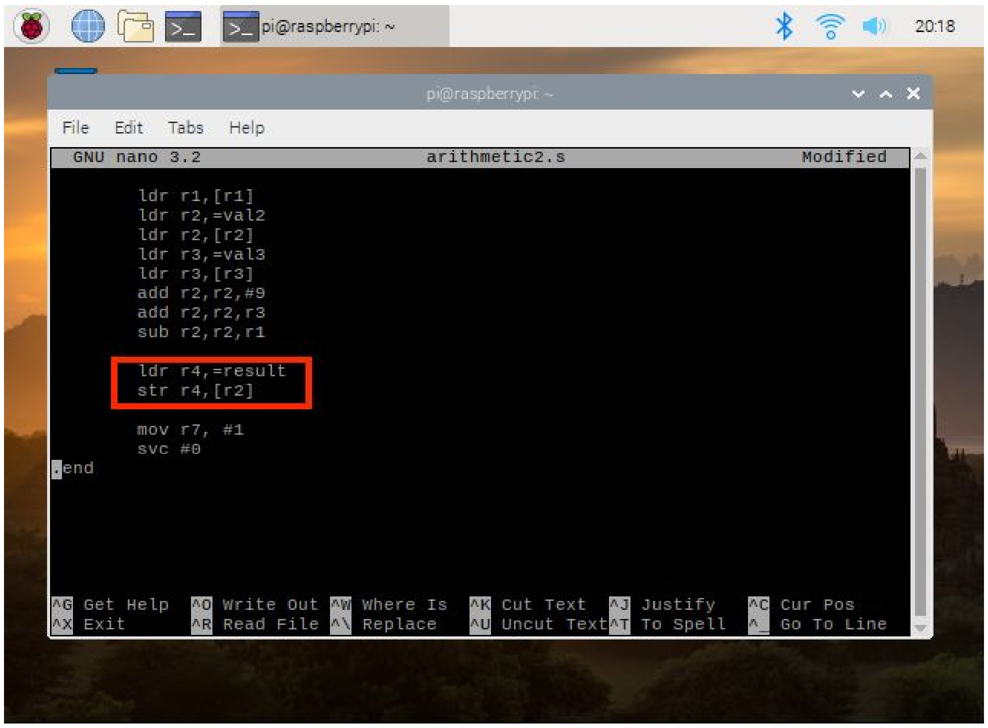
(D, 6)



```
  GNU nano 3.2                    arithmetic2.s                      Modified

@second program Register = val2+9+val3-val1
.section .data
val1:.word 6
val2:.word 11
val3:.word 16
result:.word 0
.section.text
.globl _start
_start:
        ldr r1,=val1
        ldr r1,[r1]
        ldr r2,=val2
        ldr r2,[r2]
        ldr r3,=val3
        ldr r3,[r3]
        add r2,r2,#9
        add r2,r2,r3
        sub r2,r2,r1

        ldr r4,=result

^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
```

The first part of the file arithmetic2.s .

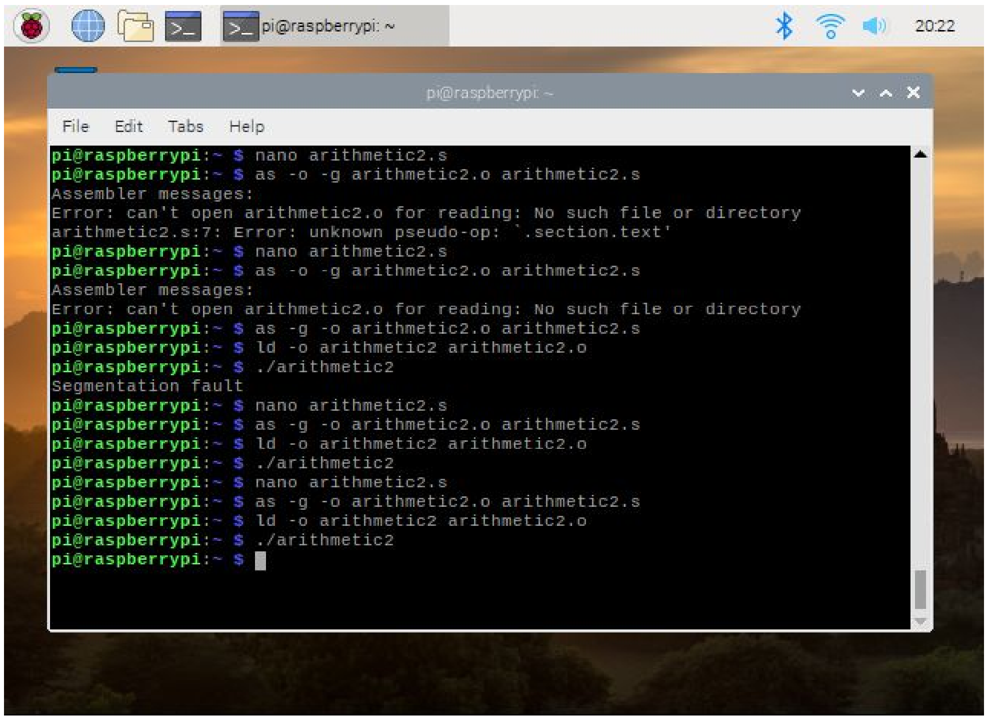
(D, 7)



```
 GNU nano 3.2                  arithmetic2.s                    Modified

        ldr r1,[r1]
        ldr r2,=val2
        ldr r2,[r2]
        ldr r3,=val3
        ldr r3,[r3]
        add r2,r2,#9
        add r2,r2,r3
        sub r2,r2,r1

        ldr r4,=result
        str r4,[r2]

        mov r7, #1
        svc #0
.end

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

Continuation of the file arithmetic2.s. The highlighted part is incorrect in this image as we had the registers reversed.

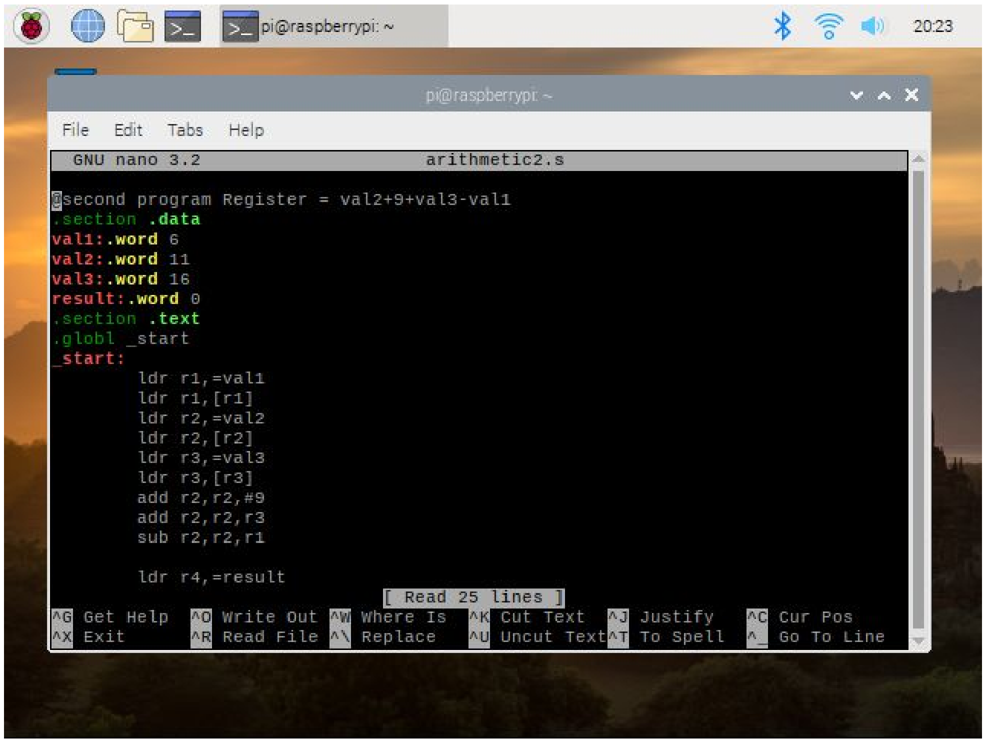
(D, 8)



```
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -o -g arithmetic2.o arithmetic2.s
Assembler messages:
Error: can't open arithmetic2.o for reading: No such file or directory
arithmetic2.s:7: Error: unknown pseudo-op: `.section.text'
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -o -g arithmetic2.o arithmetic2.s
Assembler messages:
Error: can't open arithmetic2.o for reading: No such file or directory
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
Segmentation fault
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
pi@raspberrypi:~ $ nano arithmetic2.s
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
pi@raspberrypi:~ $ 
```

Shows the errors encountered while attempting to run the program. First, “unknown pseudo-op: ‘.section.text’. Second, “no such file or directory found”. Third, “segmentation fault”.

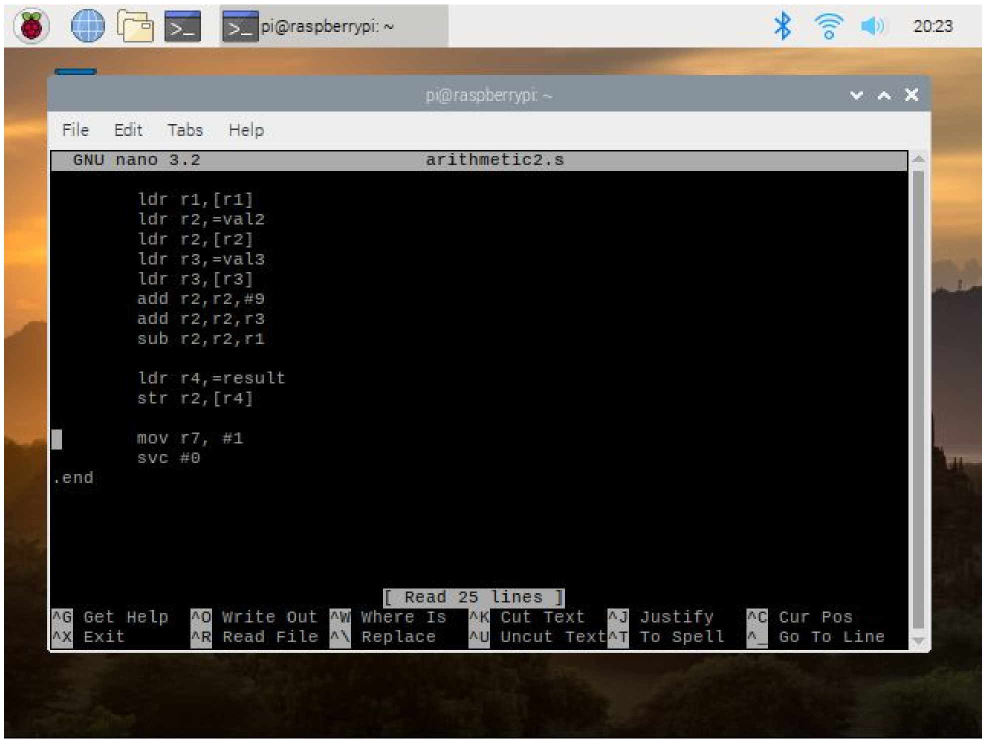
(D, 9)



```
  GNU nano 3.2                       arithmetic2.s

@second program Register = val2+9+val3-val1
.section .data
val1:.word 6
val2:.word 11
val3:.word 16
result:.word 0
.section .text
.globl _start
_start:
        ldr r1,=val1
        ldr r1,[r1]
        ldr r2,=val2
        ldr r2,[r2]
        ldr r3,=val3
        ldr r3,[r3]
        add r2,r2,#9
        add r2,r2,r3
        sub r2,r2,r1

        ldr r4,=result
                             [ Read 25 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
```

The first part of the corrected arithmetic2.s.

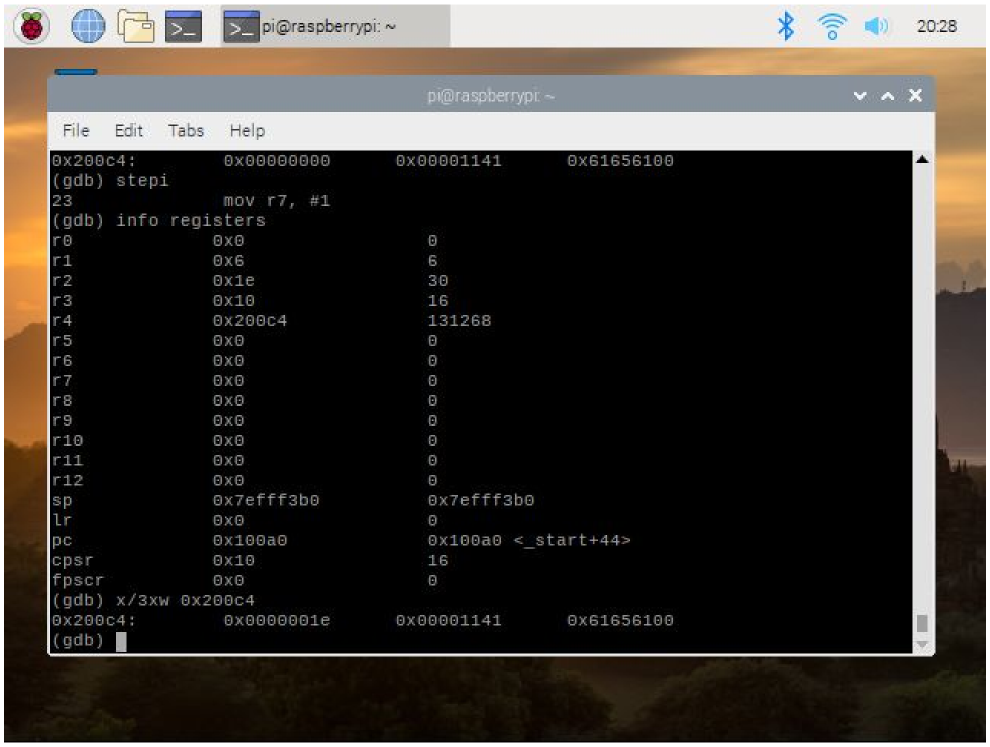
(D, 10)



```
  GNU nano 3.2                       arithmetic2.s

        ldr r1,[r1]
        ldr r2,=val2
        ldr r2,[r2]
        ldr r3,=val3
        ldr r3,[r3]
        add r2,r2,#9
        add r2,r2,r3
        sub r2,r2,r1

        ldr r4,=result
        str r2,[r4]

        mov r7, #1
        svc #0
.end



                             [ Read 25 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
```

The second part of the corrected arithmetic2.s.

(D, 11)



```
0x200c4:        0x00000000      0x00001141      0x61656100
(gdb) stepi
23              mov r7, #1
(gdb) info registers
r0             0x0                 0
r1             0x6                 6
r2             0x1e                30
r3             0x10                16
r4             0x200c4             131268
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             0x100a0             0x100a0 <_start+44>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/3xw 0x200c4
0x200c4:        0x0000001e      0x00001141      0x61656100
(gdb) 
```

The values stored in the registers and memory after execution of the file.