Developing Soft and Parallel Programming Skills Using Project Based Learning

Spring 2019, Team HUSTLE

Eseosasere (Eseosa) Omobude, Lucky T Phan, Hwysun (Harry) Park, Smit Patel, Tommy Lim

Task 1:

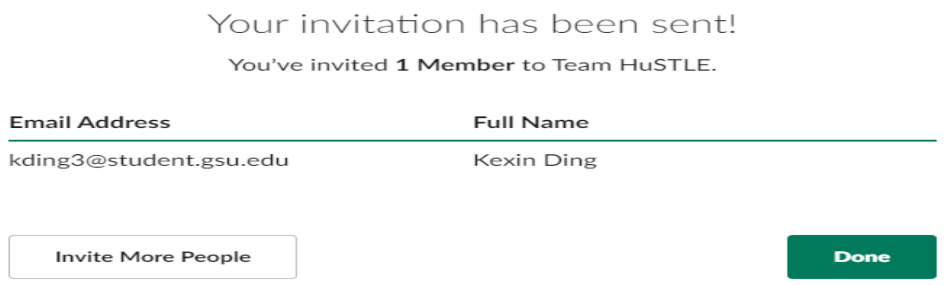
Planning and Scheduling/Work Breakdown Structure

Name	E-mail	Tasks	Duration	Note
Eseosasere Omobude (Coordinator)	eomobude1@student.gsu.edu	Put the project together into one final project. Oversaw making sure everyone got their designated part done and completed	5 hours	Slack and GitHub should be created in the first place; Put everything in Slack and GitHub. Report due Fri, 2/22/19
Smit Patel	spatel261@student.gsu.edu	Responsible for the Parallel programming Basics part	2 hours	Parallel programming Basics questions due Fri, 2/20/19
Tommy Lim	tlim3@student.gsu.edu	Responsible for the ARM Assembly Programming and the lab report	5 hours	Set up the date for the ARM Assembly Programming before Wed, 2/22/19
Lucky Phan	lphan7@student.gsu.edu	Responsible for recording and editing the video.	3 hours	Set up the date for the recording before Wed 2/20/19; Video due Fri, 2/22/19
Hwysun Park	hpark44@student.gsu.edu	Responsible for studying the parallel programming foundation and answering the questions from the assignment	3 hours	Parallel programming foundation and answered questions from the assignment due before Wed, 2/20/19

Task 2:

Collaboration

Slack Invitation

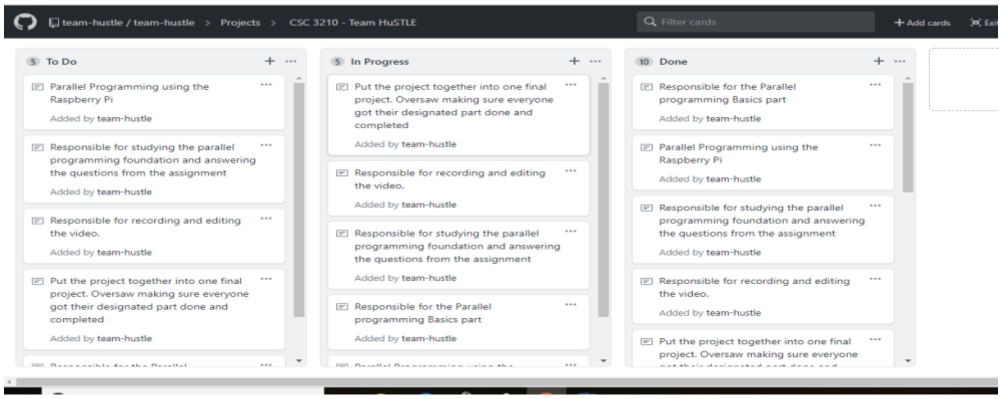


Your invitation has been sent!

You've invited **1 Member** to Team HuSTLE.

Email Address	Full Name
kding3@student.gsu.edu	Kexin Ding

GitHub


team-hustle / team-hustle > Projects > CSC 3210 - Team HuSTLE

5 To Do

- Parallel Programming using the Raspberry Pi
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Put the project together into one final project. Oversaw making sure everyone got their designated part done and completed
  Added by team-hustle

5 In Progress

- Put the project together into one final project. Oversaw making sure everyone got their designated part done and completed
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for the Parallel programming Basics part
  Added by team-hustle

10 Done

- Responsible for the Parallel programming Basics part
  Added by team-hustle
- Parallel Programming using the Raspberry Pi
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Put the project together into one final project. Oversaw making sure everyone

Task 3:

Parallel Programming Skills

a) Foundation

- Identify the components on the raspberry PI B+.
  - Micro USB Connector (power supply)
  - DSI Display Connector
  - HDMI Connector
  - CSI Camera Connector
  - RCA Video/Audio Jack
  - 10/100 Ethernet Port
  - 2 USB 2.0 ports
  - Ethernet Controller
  - CPU/RAM
- How many cores does the Raspberry Pi's B+ CPU have?
  - Quad-Core (4 Cores) CPU
- List three main differences between X86 (CISC) and ARM Raspberry PI (RISC). Justify your answer and use your own words.
  - While X86 processors can take many instructions to its memory, ARM processors can only use registers to operate instructions and have to load/store to access memory.
  - X86 processors have larger instruction set with more features, allowing for more operations with less registers. ARM processors have a reduced instruction set with more registers, allowing the instructions to be executed more quickly.
  - Unlike X86 processors, almost every instruction in ARM processors can be executed conditionally.

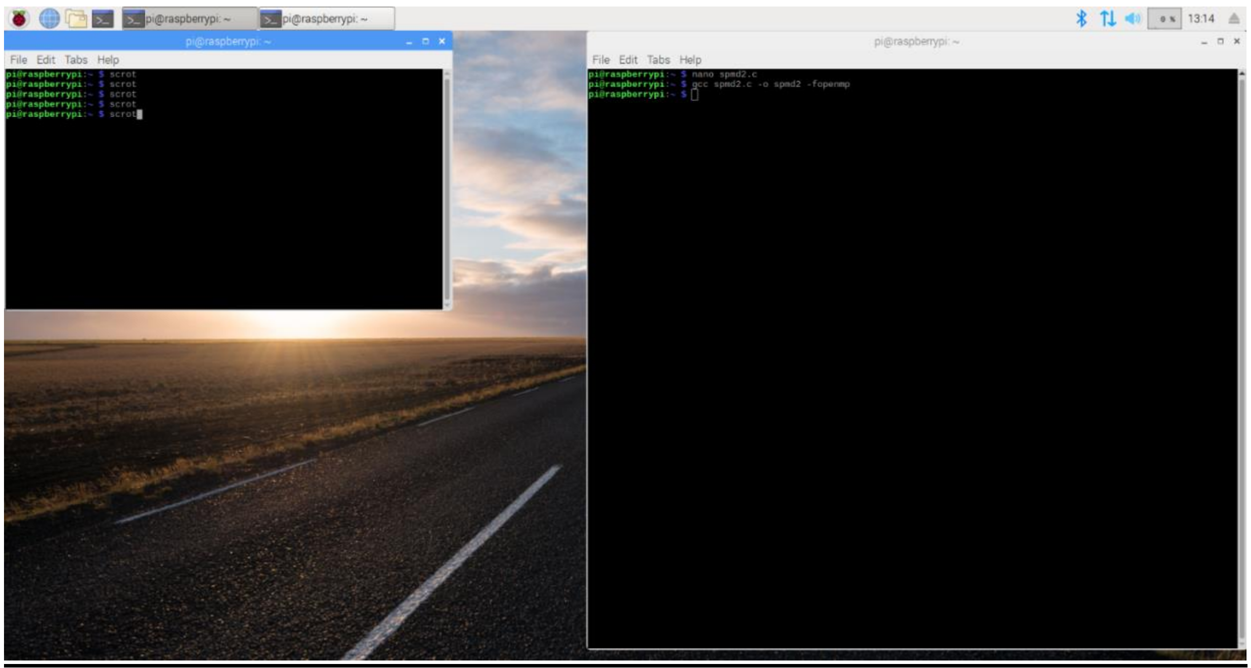
  - X86 processors use the little-endian format which addresses, sends, and stores the least significant byte first and the most significant byte last. ARM processors are known as bi-endian which can be operated in either little-endian or big-endian mode.

- What is the difference between sequential and parallel computation and identify the practical significance of each?
  - In sequential computation, a problem is broken into a series of instructions that are executed sequentially one after another. In parallel computation, a problem is broken into discrete parts that can be solved concurrently. Then each part is broken down to a series of instructions that execute simultaneously on different processors. Traditionally, software has been written for sequential computation on a single processor. Parallel computing can solve more complex problems in less time with the simultaneous use of multiple compute resources.
- Identify the basic form of data and task parallelism in computational problems.
  - Data parallelism refers to a broad category of parallelism in which the same computation is applied to multiple data items, so the amount of available parallelism is proportional to the input size. Task parallelism applies to solutions where parallelism is organized around the functions to be performed rather than around the data.
- Explain the differences between processes and threads.
  - A process is the abstraction of a running program. A thread is a lightweight process that allows a single process to be decomposed to smaller, independent parts. Processes do not share memory with each other, whereas all threads share the common memory of the process they belong to. Even though a process can be operated on a single core at a time, threads can be scheduled on separate cores as available.
- What is OpenMP and what is OpenMP pragmas?
  - OpenMP is one of the libraries/languages to program multicore architectures. It has been the industry standard since the late 1990s. It has a native support with GCC compilers and is easier to program than POSIX threads. It uses an implicit multithreading model in which the library handles thread creation and management. OpenMP pragmas are compiler directives that enable the compiler to generate threaded code.
- What applications benefit from multi-core? (list four)
  - Database servers
  - Web servers
  - Compilers
  - Multimedia applications
  - Scientific applications
  - CAD/CAM
  - Applications with thread-level parallelism

- Why Multicore? (why not single core, list four)
  - An operating system can execute more processes at once with more CPU cores.
  - It is difficult to make single-core clock frequencies even higher.
  - Deeply pipelined circuits cause heat problems, speed of light problems, design and verification issue, etc.
  - Many new applications are multithreaded.
  - It is a general trend in computer architecture, shifting toward more parallelism.

b) <u>Team Hustle: Parallel Programming Basics</u>

<u>(1)</u>

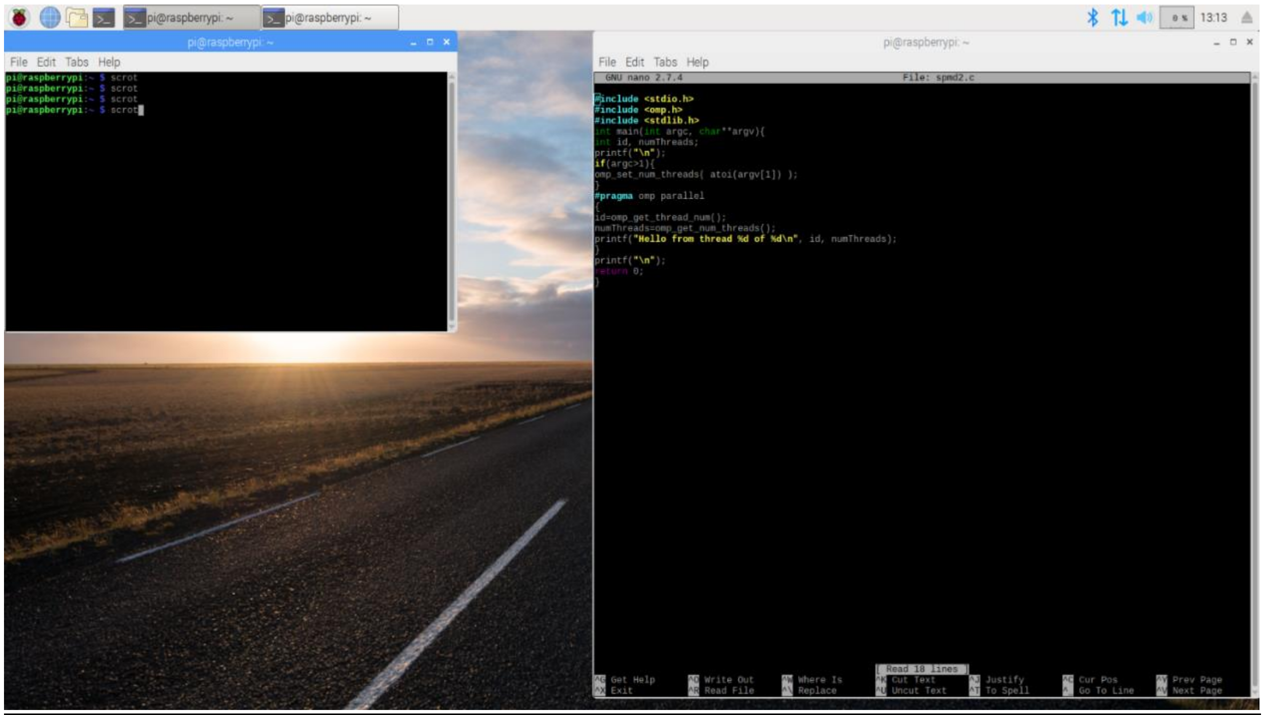


```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
```

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

-The screenshot above shows that we created a file using the nano and named it spmd2.c

- The second line creates the executable files.



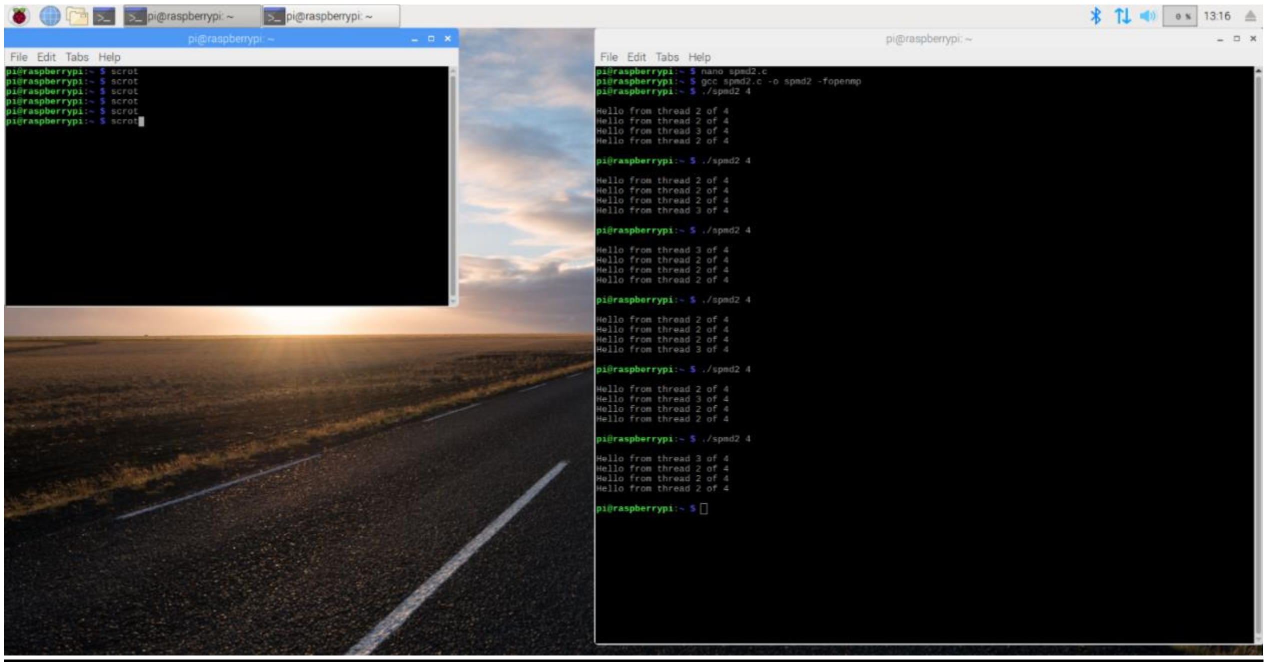
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
```

```
  GNU nano 2.7.4                                    File: 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     ^Y Prev Page
^X Exit       ^R Read File   ^\ Replace    ^U Uncut Text  ^T To Spell  ^_ Go To Line  ^V Next Page
```

- First, we typed in the program that was given in 2.1.

- Then we observed to see if everything was correct in terms of wording. The code was error free.

- Finally, we saved the file using ctrl-o.



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

```
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 3 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 2 of 4
Hello from thread 3 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:~ $ ./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 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 3 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4

pi@raspberrypi:~ $ 
```

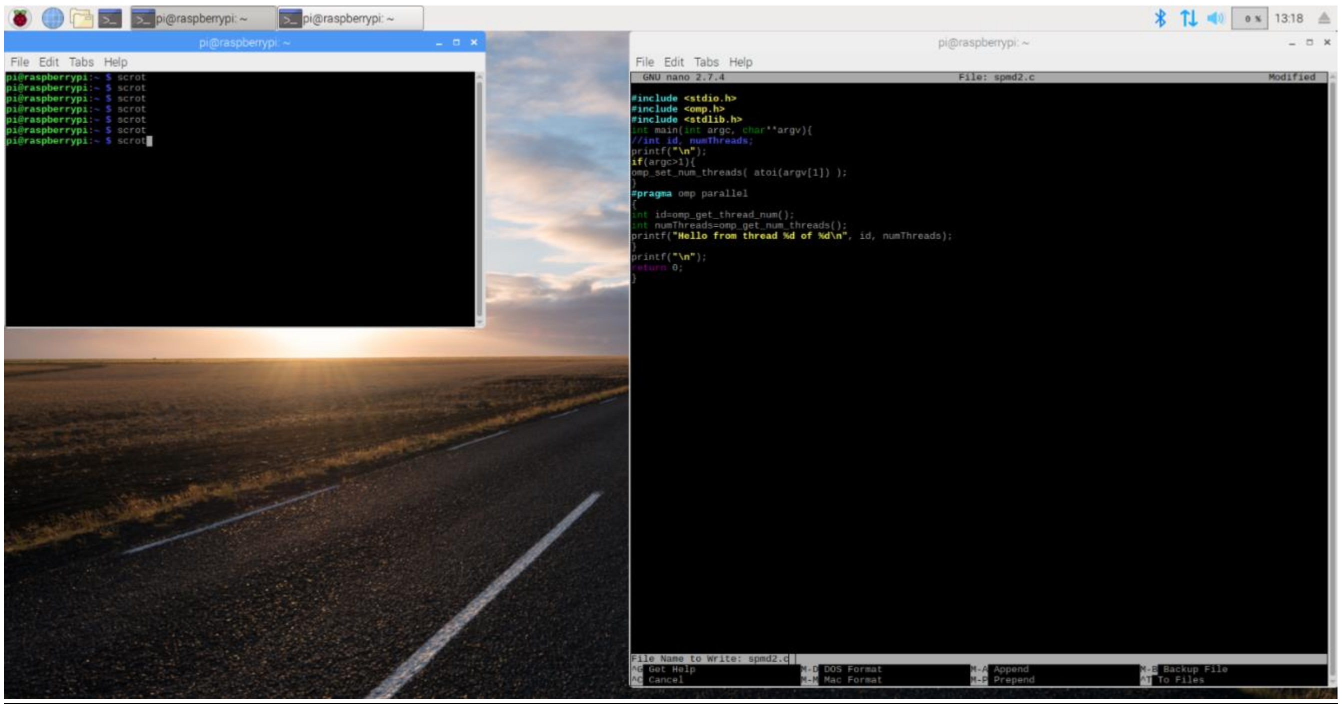
- Once you type this command (./spmd2 4), it runs the program and prints out the 4 threads that are in the file.

- We tried to use the same command more than 4 times to see if the output would be the same after a couple of runs or not.

- The patterns matched each other as the number 3's places were switched in the output.

- The pattern repeated itself most of the time.

(2)

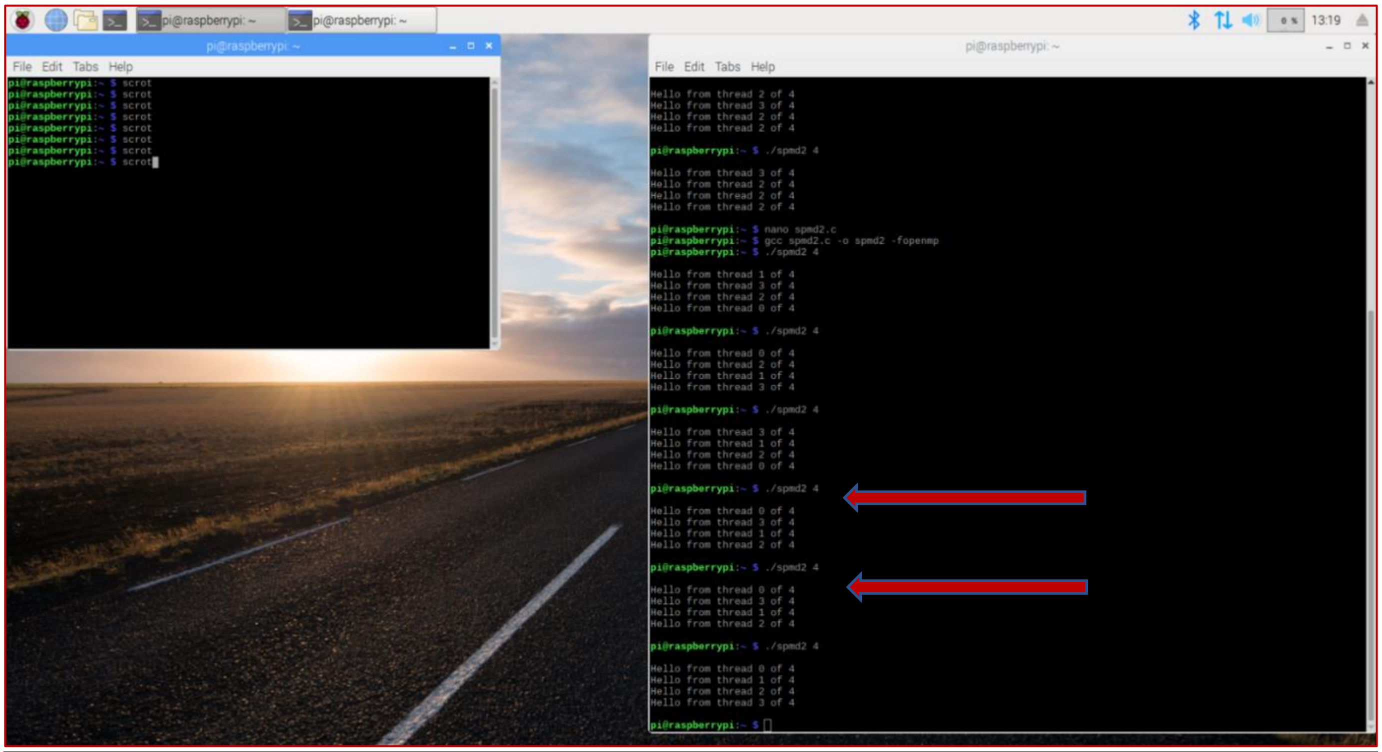


```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ scrot
```

```
  GNU nano 2.7.4                    File: 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);
}
printf("\n");
return 0;
}

File Name to Write: spmd2.c
^G Get Help       M-D DOS Format      M-A Append       M-B Backup File
^C Cancel         M-M Mac Format      M-P Prepend      ^T To Files
```

-In this screenshot, we changed the code in line 5, 12 and 13 as mentioned in 2.4.1.

- Line 5 was commented out and in line 12 and 13 we added 'int' in the beginning of the code.



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

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

pi@raspberrypi:~ $ ./spmd2 4

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

pi@raspberrypi:~ $ ./spmd2 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 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:~ $ ./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:~ $ ./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:~ $ 
```

-After the changes were made into the spmd2.c file, we then printed the output.

- As mentioned in 2.4., we tried the ./spmd2 4 command more than 4 times to see what the output would be.

- The values were very different from the program that was used in the part 1.

- Now, the values are printed after the changes are 0,1,2,3 in the 4 threads.

- After trying to print the output several times, the pattern that appeared in this program was very different from the output on previous program (Before the edit of spmd2.c).

- The same pattern does not repeat again and again. Instead, the numbers appeared very often.

-We received the same output on the 4th and the 5th try (0,3,1,2).

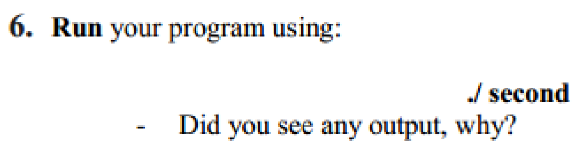
- Each time we ran the program, it printed out a unique output. It was more difficult to figure out the pattern that it was producing.

Task 4:

ARM Assembly Programming

Part1: Second Program

Question:

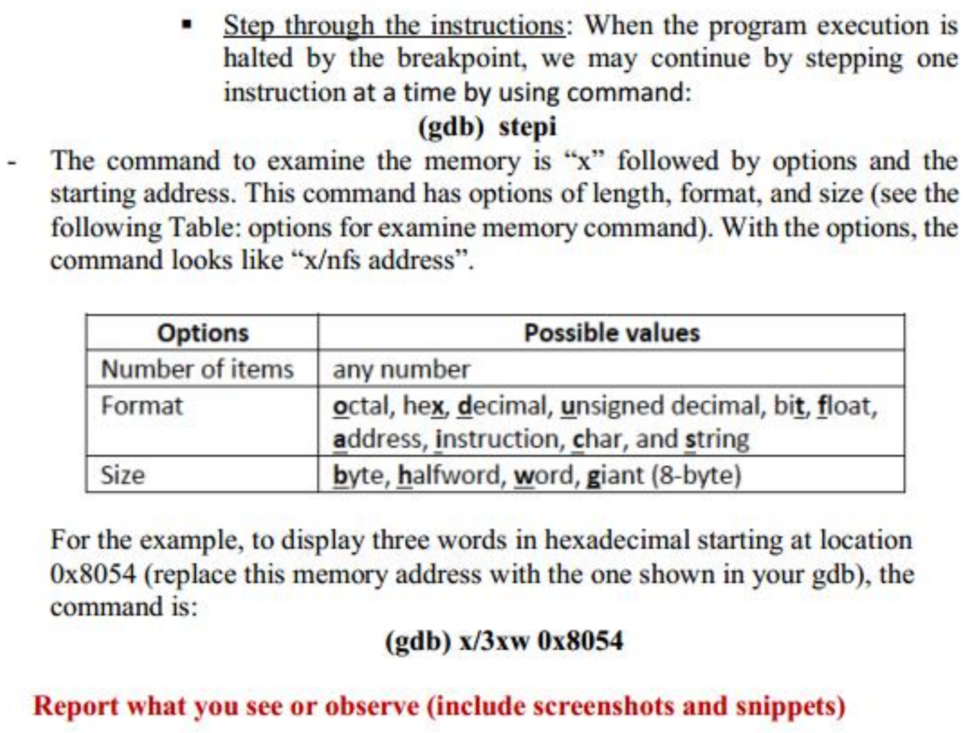


6. **Run** your program using:

./ second

- Did you see any output, why?

We did not see any output the first time we ran ‘second.’ The reason was because the program manipulates data between CPU registers and memory and does not involve any input or output. Once we used GDB to debug the program, added a breakpoint, and entered the command ‘info registers,’ we were able to display and see the output



- Step through the instructions: When the program execution is halted by the breakpoint, we may continue by stepping one instruction at a time by using command:

(gdb) stepi

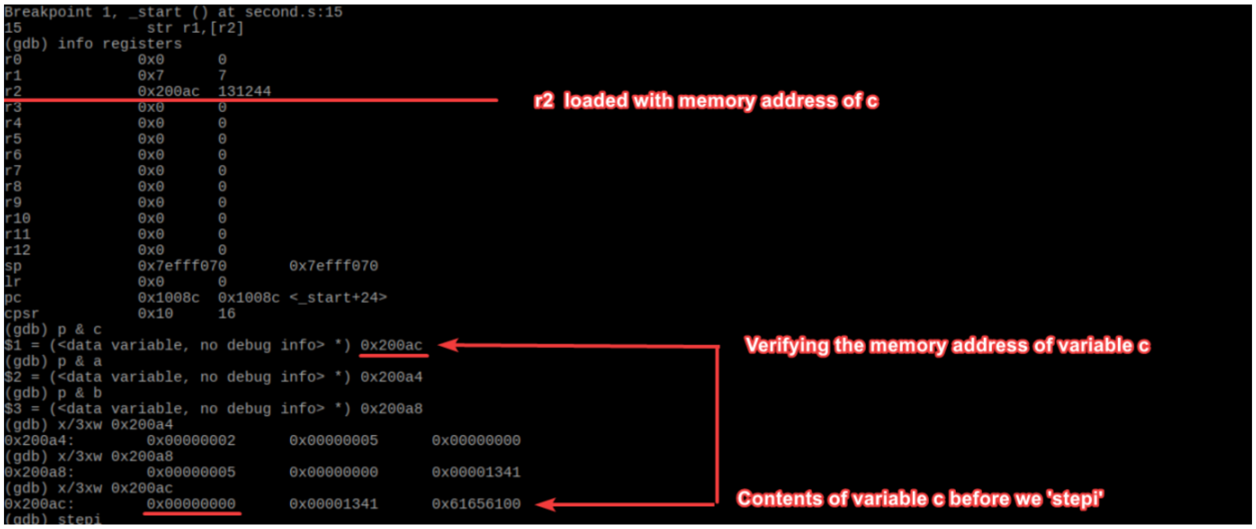
- The command to examine the memory is “x” followed by options and the starting address. This command has options of length, format, and size (see the following Table: options for examine memory command). With the options, the command looks like “x/nfs address”.

Options	Possible values
Number of items	any number
Format	octal, hex, decimal, unsigned decimal, bit, float, address, instruction, char, and string
Size	byte, halfword, word, giant (8-byte)

For the example, to display three words in hexadecimal starting at location 0x8054 (replace this memory address with the one shown in your gdb), the command is:

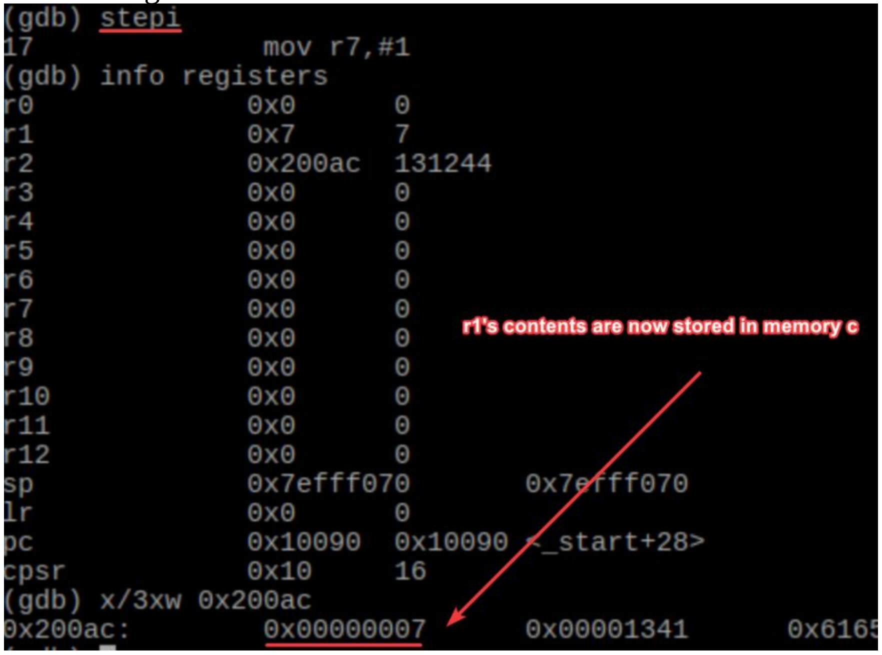
(gdb) x/3xw 0x8054

Report what you see or observe (include screenshots and snippets)


r2 loaded with memory address of c
Verifying the memory address of variable c
Contents of variable c before we 'stepi'

```
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             0x7efff070       0x7efff070
lr             0x0      0
pc             0x1008c  0x1008c <_start+24>
cpsr           0x10     16
(gdb) p & c
$1 = (<data variable, no debug info> *) 0x200ac
(gdb) p & a
$2 = (<data variable, no debug info> *) 0x200a4
(gdb) p & b
$3 = (<data variable, no debug info> *) 0x200a8
(gdb) x/3xw 0x200a4
0x200a4:        0x00000002      0x00000005      0x00000000
(gdb) x/3xw 0x200a8
0x200a8:        0x00000005      0x00000000      0x00001341
(gdb) x/3xw 0x200ac
0x200ac:        0x00000000      0x00001341      0x61656100
(gdb) stepi
```

- With the breakpoint set at line 15 of the program (str r1,[r2]), we see that register r2 contains a very large decimal value, 131244, represented in hexadecimal as 0x200ac.
- We had a hunch that the hexadecimal representation was the memory address of variable c, but we wanted to confirm that.
- Using the command **p & c** prints the address of the variable c, which displayed **0x200ac**, confirming our suspicion, and now allowing us to examine the memory using "x" command.
- **x/3xw 0x200ac** output displayed the contents of three words (32-bit data), but we were only concerned about the far left one. It has been underlined in the picture above, showing 0x00000000.



```
(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             0x7efff070       0x7efff070
lr             0x0      0
pc             0x10090  0x10090 <_start+28>
cpsr           0x10     16
(gdb) x/3xw 0x200ac
0x200ac:        0x00000007      0x00001341      0x6165
```

- We stepped through to the next instruction, which meant that the instructions at the previous breakpoint were executed (str r1, [r2]).
- Result: the content of r1,7, is now stored in memory c.
- We confirmed this by examining the memory address c once more.

PART 2

b) **Part2:** Using the second.s program as a reference, and use the same steps to edit, assemble, link, run, and debug the new program.

- Write a program that calculates the following expression:

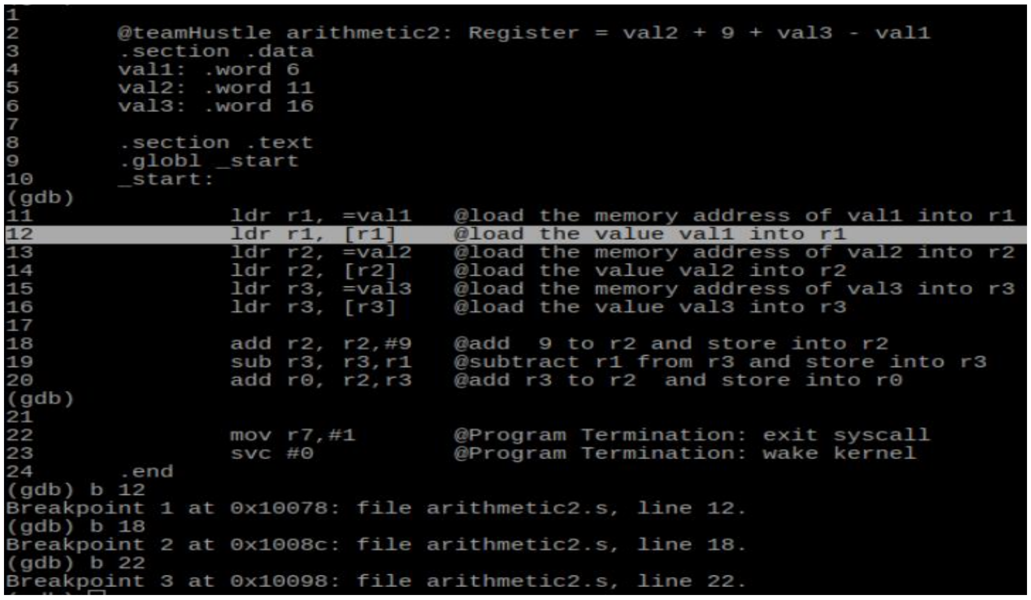
Register = val2 + 9 + val3 - val1

Assume that val1, val2, and val3 are 32-bit integer memory variables.

- Besides, Val1, val2 and val3 are integers.
- Assign val2=11, val3=16, val1=6.
- Register could be any of the Arm general purpose registers
- Use the debugger to verify the result in the memories and the Register.
- Report the Register value in hex (as shown in the debugger)

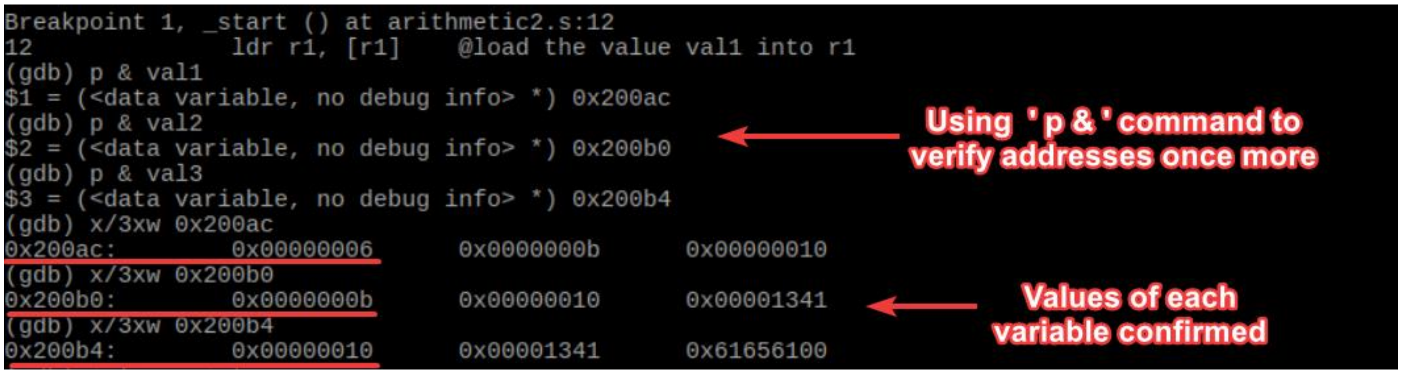
name your program a *arithmetic2.s*

- Assemble, Link, run, and debug the program



```
1
2       @teamHustle arithmetic2: Register = val2 + 9 + val3 - val1
3       .section .data
4       val1: .word 6
5       val2: .word 11
6       val3: .word 16
7
8       .section .text
9       .globl _start
10      _start:
(gdb)
11              ldr r1, =val1   @load the memory address of val1 into r1
12              ldr r1, [r1]    @load the value val1 into r1
13              ldr r2, =val2   @load the memory address of val2 into r2
14              ldr r2, [r2]    @load the value val2 into r2
15              ldr r3, =val3   @load the memory address of val3 into r3
16              ldr r3, [r3]    @load the value val3 into r3
17
18              add r2, r2,#9   @add  9 to r2 and store into r2
19              sub r3, r3,r1   @subtract r1 from r3 and store into r3
20              add r0, r2,r3   @add r3 to r2  and store into r0
(gdb)
21
22              mov r7,#1       @Program Termination: exit syscall
23              svc #0          @Program Termination: wake kernel
24      .end
(gdb) b 12
Breakpoint 1 at 0x10078: file arithmetic2.s, line 12.
(gdb) b 18
Breakpoint 2 at 0x1008c: file arithmetic2.s, line 18.
(gdb) b 22
Breakpoint 3 at 0x10098: file arithmetic2.s, line 22.
```

a. Since this is ARM architecture, we must move data values into registers before using, as we can't perform memory to memory data processing operations.
- The screenshot shows our program and the breakpoints we've set.
- To verify the contents of the memories, we repeated the process of using the '**p &** 'and '**x/nfs address**' commands for each variable.

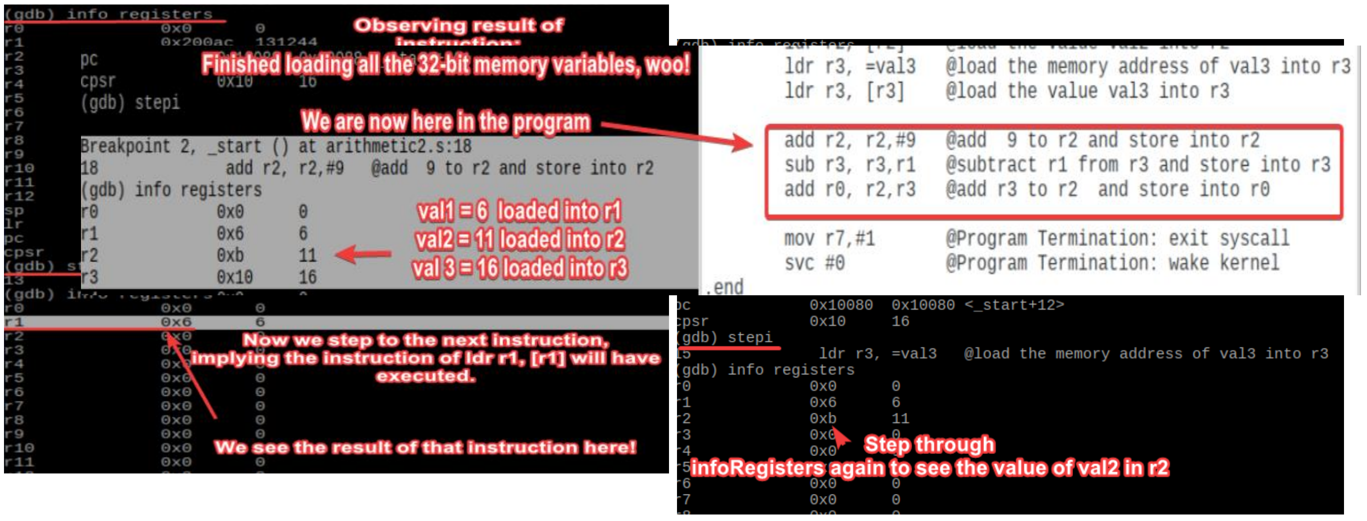


```
Breakpoint 1, _start () at arithmetic2.s:12
12              ldr r1, [r1]    @load the value val1 into r1
(gdb) p & val1
$1 = (<data variable, no debug info> *) 0x200ac
(gdb) p & val2
$2 = (<data variable, no debug info> *) 0x200b0
(gdb) p & val3
$3 = (<data variable, no debug info> *) 0x200b4
(gdb) x/3xw 0x200ac
0x200ac:        0x00000006      0x0000000b      0x00000010
(gdb) x/3xw 0x200b0
0x200b0:        0x0000000b      0x00000010      0x00001341
(gdb) x/3xw 0x200b4
0x200b4:        0x00000010      0x00001341      0x61656100
```

Using ' p & ' command to verify addresses once more
Values of each variable confirmed

- Memory address of 32-bit integer variable val1 = 0x200ac assigned a value of 6
- Memory address of 32-bit integer variable val2 = 0x200b0 assigned a value of 11 (0x0000000b where b is 11 in hexadecimal).
- Memory address of 32-bit integer variable val3 = 0x200b4 assigned a value of 16 (0x00000010 in hexadecimal after conversion)

b. A bulk of the program's instructions involved loading a memory address of a variable into a register and then loading the value of that variable into the register.
- We observed this process using **info registers** and the **stepi command** each time.
- We continued to do this until we reached the arithmetic instructions.



```
(gdb) info registers
r0             0x0      0
r1             0x200ac  131244
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
sp
lr
pc
cpsr
(gdb) st
13
(gdb) info registers
r0             0x0      0
r1             0x6      6
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
```

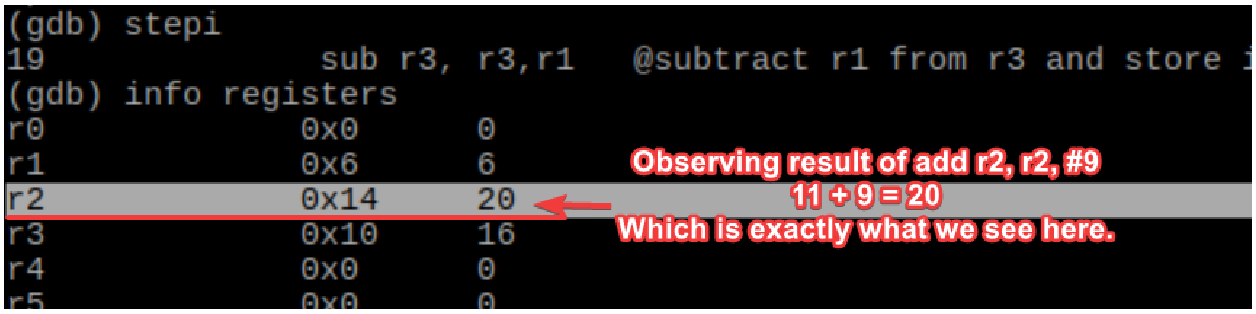
```
pc
cpsr           0x10     16
(gdb) stepi

Breakpoint 2, _start () at arithmetic2.s:18
18              add r2, r2,#9   @add  9 to r2 and store into r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0xb      11
r3             0x10     16
```

```
        ldr r3, =val3   @load the memory address of val3 into r3
        ldr r3, [r3]    @load the value val3 into r3

        add r2, r2,#9   @add  9 to r2 and store into r2
        sub r3, r3,r1   @subtract r1 from r3 and store into r3
        add r0, r2,r3   @add r3 to r2  and store into r0

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

```
pc             0x10080  0x10080 <_start+12>
cpsr           0x10     16
(gdb) stepi
15              ldr r3, =val3   @load the memory address of val3 into r3
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0xb      11
r3             0x0      0
r4             0x0      0
r5
r6             0x0      0
r7             0x0      0
```

Observing result of instruction:
Finished loading all the 32-bit memory variables, woo!
We are now here in the program
val1 = 6 loaded into r1
val2 = 11 loaded into r2
val 3 = 16 loaded into r3
Now we step to the next instruction, implying the instruction of ldr r1, [r1] will have executed.
We see the result of that instruction here!
Step through
infoRegisters again to see the value of val2 in r2

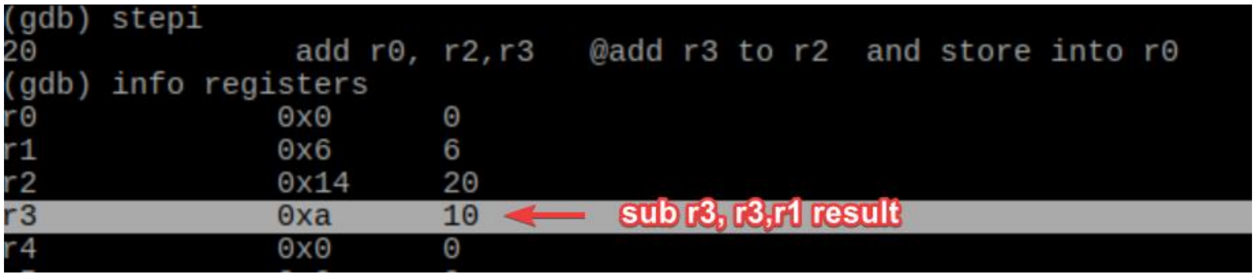
c. Breakpoint 2 is placed on line 18 of the program with the instruction "add r2, r2, #9", so the next issued **stepi** command should show that **r2**'s content to be **20**, **0x14** in hexadecimal.



```
(gdb) stepi
19              sub r3, r3,r1   @subtract r1 from r3 and store i
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x14     20
r3             0x10     16
r4             0x0      0
r5             0x0      0
```

Observing result of add r2, r2, #9
11 + 9 = 20
Which is exactly what we see here.

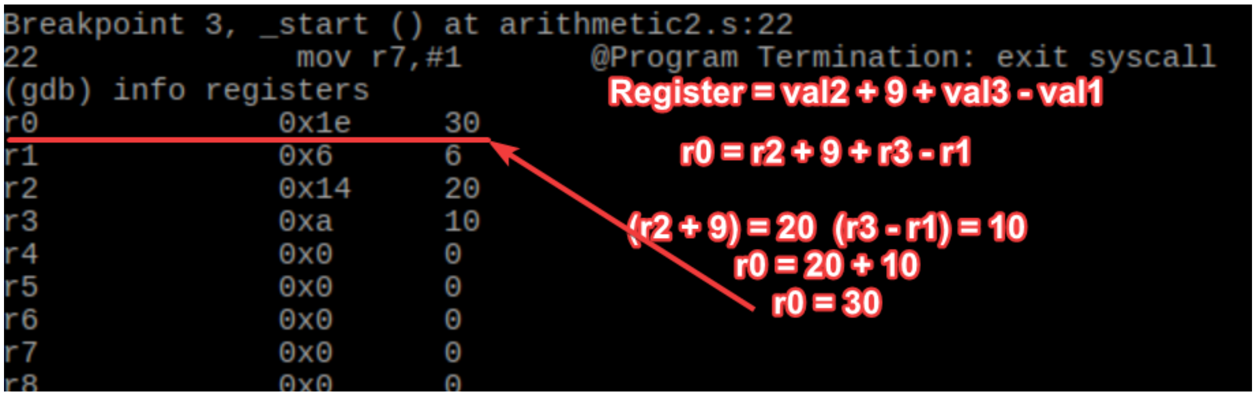
d. Stepping through, we see that r2 shows exactly what we were expecting.
   - The next **stepi** should show the result of **sub r3, r3,r1** or 16 – 6 and store that into r3.



```
(gdb) stepi
20              add r0, r2,r3   @add r3 to r2  and store into r0
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x14     20
r3             0xa      10
r4             0x0      0
```

e. **r3** accurately shows the result of **sub r3, r3, r1**.
   - The final step through should show the result of **add r0, r2,r3**.

   - with r2 = 20 r3 = 10 then **Register 0 (r0)** should contain the value of 30 after the instruction is executed.

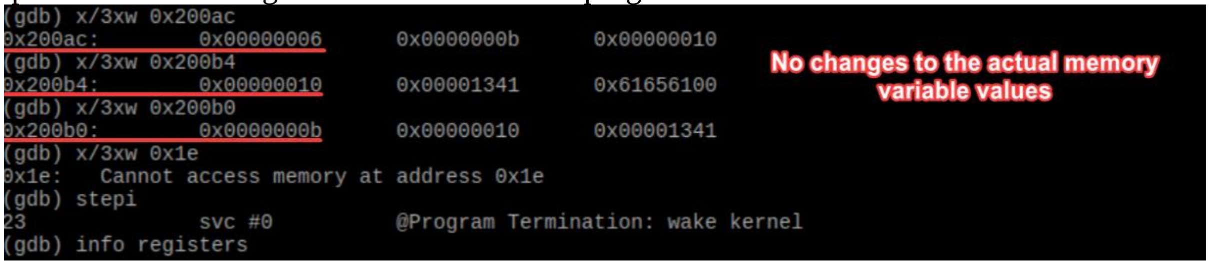


```
Breakpoint 3, _start () at arithmetic2.s:22
22              mov r7,#1       @Program Termination: exit syscall
(gdb) info registers
r0             0x1e     30
r1             0x6      6
r2             0x14     20
r3             0xa      10
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
```

Register = val2 + 9 + val3 - val1
r0 = r2 + 9 + r3 - r1
(r2 + 9) = 20 (r3 - r1) = 10
r0 = 20 + 10
r0 = 30

f. r0 contains decimal value 30, 0x1e in hexadecimal, which is exactly what we were expecting to see.
   - Register r0 is equal to the expression **val2 + 9 + val3 – val1**

g. We were curious to see if the actual memory variable values changed, so we inspected their respective addresses right before the end of the program.



```
(gdb) x/3xw 0x200ac
0x200ac:        0x00000006      0x0000000b      0x00000010
(gdb) x/3xw 0x200b4
0x200b4:        0x00000010      0x00001341      0x61656100
(gdb) x/3xw 0x200b0
0x200b0:        0x0000000b      0x00000010      0x00001341
(gdb) x/3xw 0x1e
0x1e:   Cannot access memory at address 0x1e
(gdb) stepi
23              svc #0          @Program Termination: wake kernel
(gdb) info registers
```

- There were no changes necessary!

Task 6:

Presentation

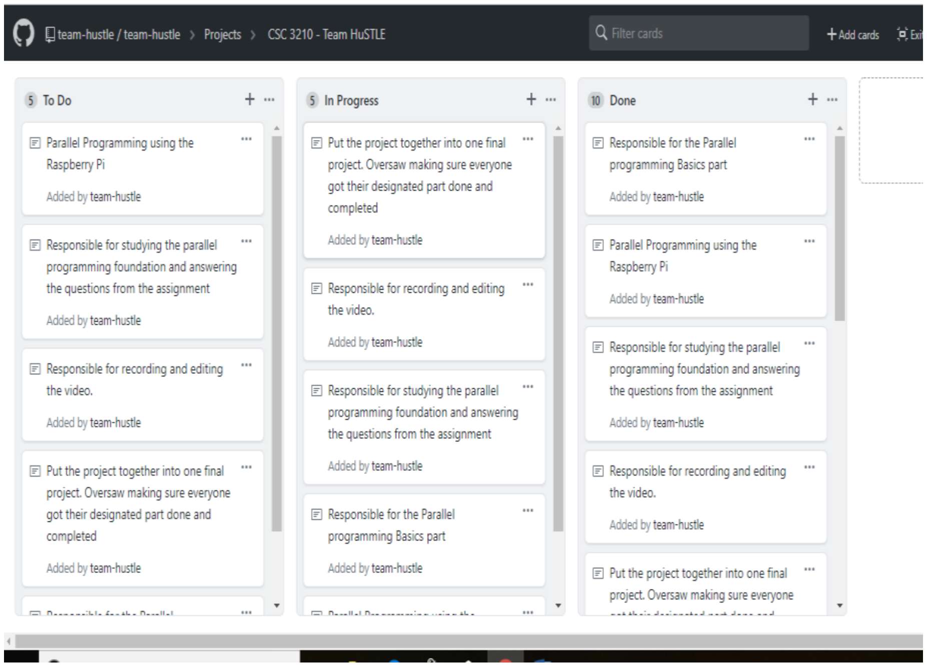
Link to Video: YouTube – https://www.youtube.com/channel/UCfW_InzE5vaBrMkOPuKUd7g

Appendix

Slack: https://teamhustle-gsu.slack.com/

GitHub: https://github.com/team-hustle

Screenshots for Slack and Github:



team-hustle / team-hustle › Projects › CSC 3210 - Team HuSTLE

Filter cards

+ Add cards

5 To Do

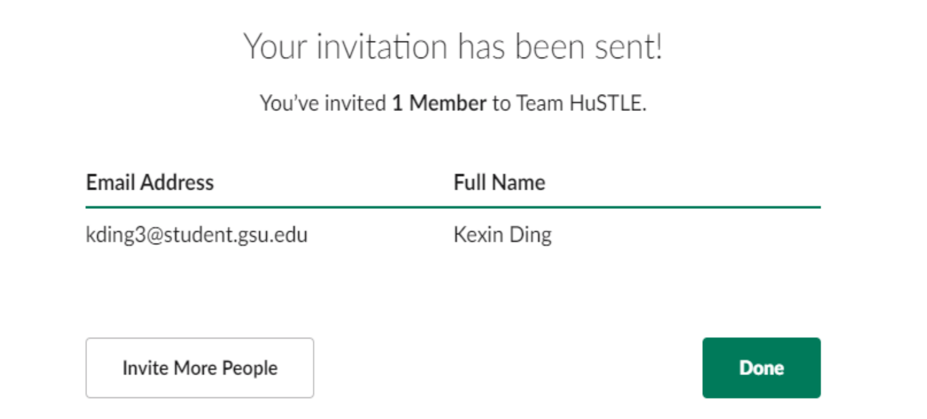
- Parallel Programming using the Raspberry Pi
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Put the project together into one final project. Oversaw making sure everyone got their designated part done and completed
  Added by team-hustle

5 In Progress

- Put the project together into one final project. Oversaw making sure everyone got their designated part done and completed
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for the Parallel programming Basics part
  Added by team-hustle

10 Done

- Responsible for the Parallel programming Basics part
  Added by team-hustle
- Parallel Programming using the Raspberry Pi
  Added by team-hustle
- Responsible for studying the parallel programming foundation and answering the questions from the assignment
  Added by team-hustle
- Responsible for recording and editing the video.
  Added by team-hustle
- Put the project together into one final project. Oversaw making sure everyone



Your invitation has been sent!

You've invited **1 Member** to Team HuSTLE.

Email Address	Full Name
kding3@student.gsu.edu	Kexin Ding

YouTube – https://www.youtube.com/channel/UCfW_InzE5vaBrMkOPuKUd7g