Developing Soft and Parallel Programming Skills Using Project- Based Learning

SPRING 2019

ARM Strong

Group Members:

Ailany Icassatti, Hashim Amin, Isaiah Smith,

Sivasubramaniyan Mourougassamy,

Thang Nguyen & Toan Le

- ❑ Planning and Scheduling

Assignee Name	**Email**	**Task**	**Duration**	**Dependency**	**Due Date**	**Notes**
Ailany Icassatti	aicassatti1@student.gsu.edu	1. Task 3 Part a. Parallel Programming Question 2. Task 5 - Attach task 3 report on Google Docs 3. Task 6 - Create a video with group members	1. 4 hr 2. 10 minutes 3. 2 hrs	1. Waiting for everyone to review, to create its final version	1. 2/19/19 2. 02/19/19 3. 02/19/19	
Hashim Amin	hamin3@student.gsu.edu	Task 3/4: Helping with debugging the program and understand what was going on Task 6: Collaborate with team on making the video	2 hrs 1 hr	Programs for tasks 3&4 being written up to be able to debug them	2/12/19 2/19/19	Task3/4: was done in the presence of the whole group

Isaiah Smith (Coordinator)	ismith27@student.gsu.edu	Task 1: Responsible for setting up the chart Task 5: Compile the first draft of the Report and edit it	1. 5mins 5. 1hr to compile first draft 2-3hr to edit	1. Dependent on everyone being assigned as well as doing their task and reporting in	2/21/2019	Everyone getting their stuff in on time. I only set up the chart everyone filled theirs in.
Sivasubramaniyan Mourougassamy	smourougassamy1@student.gsu.edu	Task 4: typing up the program for task 4 part 1 and 2 in-front of the group for everyone to debug together. Task 4: Detailed lab report	1 hour - typing code 2 hour - detailed lab report	Having personal Raspberry Pi to work on	2/19/19	Note: Everyone contributed to making sure the code was debugged Note: Had to install NTFS-3g onto the pi to allow capability to write to NTFS storage devices (to transfer

						code from PI to local machine).
Thang Nguyen	tnguyen469 @student.gsu.edu	1. Task#6: Edit group video and upload the video 2. Meet with group mates to discuss program and participate in video	2.5 hours to edit .5 hour to upload	Getting everyone together to film and Video editing software to work	2/21/19	
Toan Le	tle96@student.gsu.edu	- Task 3, part b: Parallel programming basic + Observe, screen shot, explain to the team + Detailed report - Present Parallel programming basic to the group	45 minutes -1hr		02/19/2019	Note: Everyone was present when the code was being written and debugged. Everyone observed what was wrong with the code and how it was fixed.

- ❑ Parallel Programming Skills

Part A

Question #1 - Identifying the components on the raspberry PI B+

1. CPU/RAM (Processor - 64-bit Broadcom BCM2837B0, Quad Core A53(ARM v8))

2. Extended 40 pin GPIO Header

3. USB - 4x USB 2.0 Ports

4. Ethernet Controller (LAN Port)

5. Stereo Audio/Video Jack (3.5mm 4-pole Composite)

6. MIPI CSI Camera Port

7. HDMI video Output

8. MicroSD Card Slot (bottom)

9. MIPI DSI Display Port

10. Micro USB Power Input 5.1V, 2.5A

11. Bluetooth – Cypress Semiconductor CYW43455 Wi-fi, Bluetooth (BR + EDR + BLE)

Question #2 - How many cores does the Raspberry Pi's B+ CPU have?

The Raspberry Pi B+ CPU has four independent cores.

Question #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)

Reduced Instruction Set Computing (RISC)	**Complex Instruction Set Computer (CISC)**
This processor has simple instructions, making the average clock cycle per instruction (CPI) 1.5.	This processor has complex instructions with multiple clocks, making the average CPI range from 2-15.

Its performance optimization focus on software, not requiring external memory for calculations.	Its performance optimization focus on hardware, requiring external memory for calculations.
Multiple register sets used in high-end applications, like image and video processing, telecommunications, etc.	Single register set used in low-end applications, like home automation, security systems, etc.

Question #4 - What is the difference between sequential and parallel computation and identify

the practical significance of each?

In sequential or serial computing, a problem is separated in a discrete series of instructions. These instructions are executed sequentially one after another, on a single processor, and only one instruction may execute at any moment in time.

In parallel computing, a problem is separated in different parts that can be solved concurrently and then each part is broken down into a series of instructions executed simultaneously on different processors. Simultaneous use of multiple compute resources to solve a computational problem employing an overall control/coordination mechanism.

Traditionally, software has been written for serial computation, to be run on a single computer having a single Central processing Unit (CPU). A single computer resource can only do one thing at a time. Multiple compute resources can do many things simultaneously running using multiple CPUs. However, parallel processors are costly compared to a serial processor, but serial processing takes more time than parallel.

Question #5 - Identify the basic form of data and task parallelism in computational

problems.

Data parallelism is the concept of dividing up the data and doing the same work on different processors; the same computation is applied to multiple data items. This leads to enormous amount of potential parallelism, since its amount is proportional to the input size. It gives programmers flexibility in writing scalable parallel programs.

Task parallelism is the concept of dividing up the task and processing each part concurrently. As an example, task parallelism can have different processors, each look at the same data set and compute different answers; since the parallelism is organized around the functions to be performed rather than around the data. It ensures that all of the processors contribute to the result. One challenge facing task parallelism is that it is not as scalable as data parallelism. A special form of task parallelism is called pipelining. In pipelining, a problem is divided into sub-problems, and they can each be operated on independently with multiple problem instances being solved. To maximize its effectiveness, the operations (stages) must be balanced out to equal out the work, and to be completed in the same amount of time.

Question #6 - Explain the differences between processes and threads.

Each instance of a running program is a called a process, while a thread is a unit of execution within a process. A process can have multiple threads. A process has separate memory and address space, meaning it runs independently and thus cannot directly access shared data in other processes. Threads can be called lightweight processes used for small tasks, because they have their own stack and can access shared data.

Question #7 - What is OpenMP and what is OpenMP pragmas?

OpenMP is an application programming interface that uses an implicit multithreading model in which the library handles thread creation and management. OpenMP is a standard that compilers who implement it must adhere to. It is used on shared memory multiprocessor (core) computers. OpenMP pragmas are compiler directives that enable the compiler to generate threaded code.

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

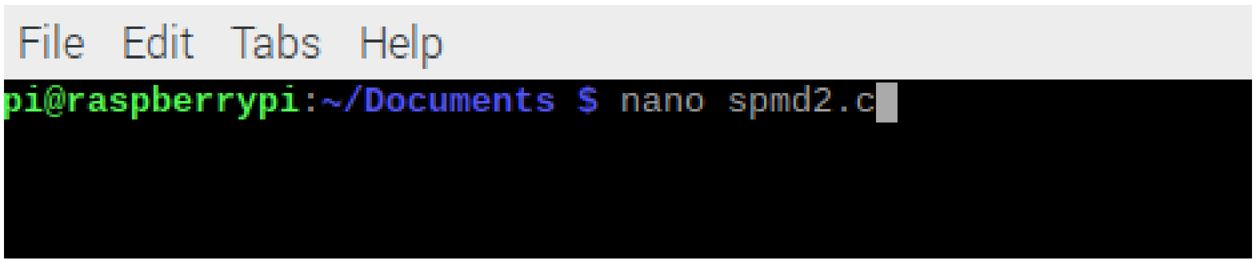
Database servers, Web servers, Compilers, and Multimedia applications.

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

1. Many new applications are multithreaded
2. Difficult to make single-core clock frequencies even higher
3. Deeply pipelined circuits: heat problems, speed of light problems, large design and verification, large design teams necessary, and server farms need expensive air-conditioning.
4. General trend in computer architecture (shift towards more parallelism)

Part B

The code is edited by "nano," a built-in text editor in terminal. The command to open nano is "nano + file name."

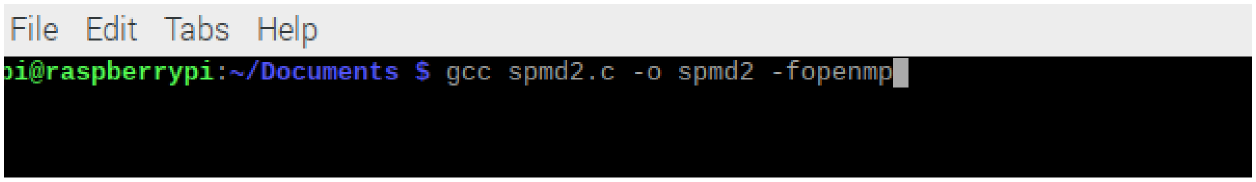


```
pi@raspberrypi:~/Documents $ nano spmd2.c
```

After entering the code, the file can be saved by pressing Ctrl + O, and nano is closed by Ctrl + X combination. An executable file is built from the source code by the command "gcc spmd2.c -o spmd2 – fopenmp". Here is the first code snippet version of this program.

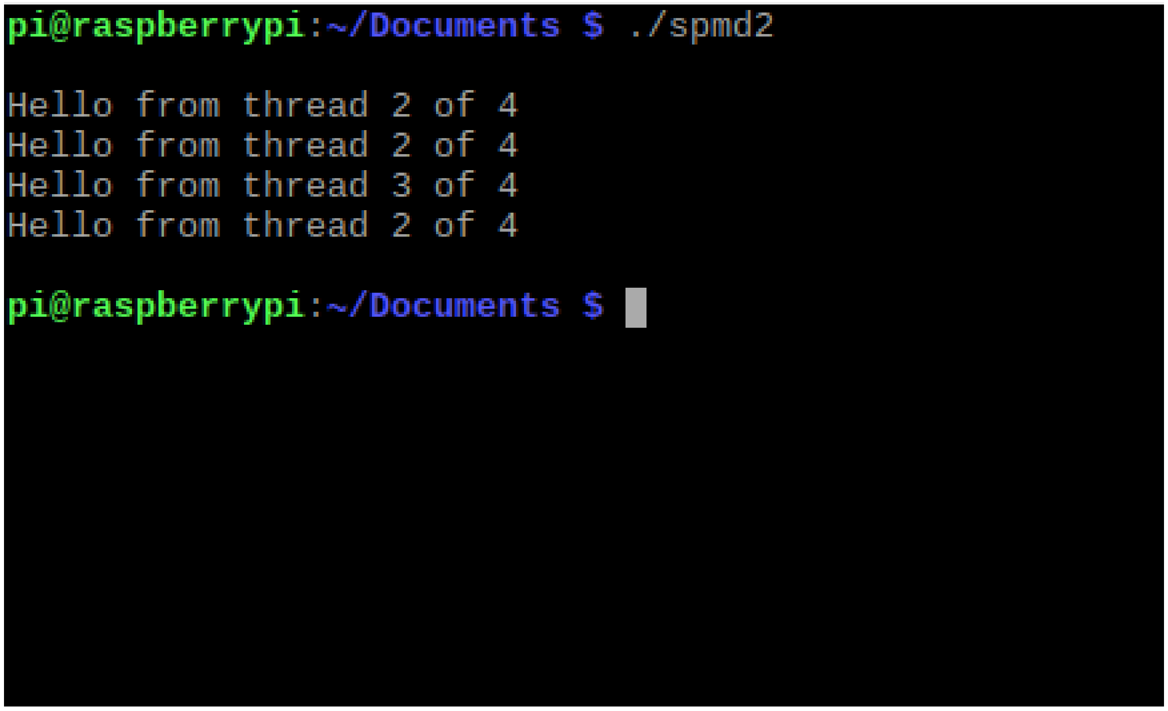
```
#include<stdio.h>
#include<omp.h>
#include<stdlib.h>
int main(int argc, char** argv)
 {
 int id, numThreads;
 printf("\n");
 if(argc > 1) {
   omp_set_num_threads( atoi(argv[1]));
 }
 #pragma omp parallel
 {
   id = omp_get_thread_num();
   numThreads = omp_get_num_threads();
   printf("Hello from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

Here is the command to create the executable file



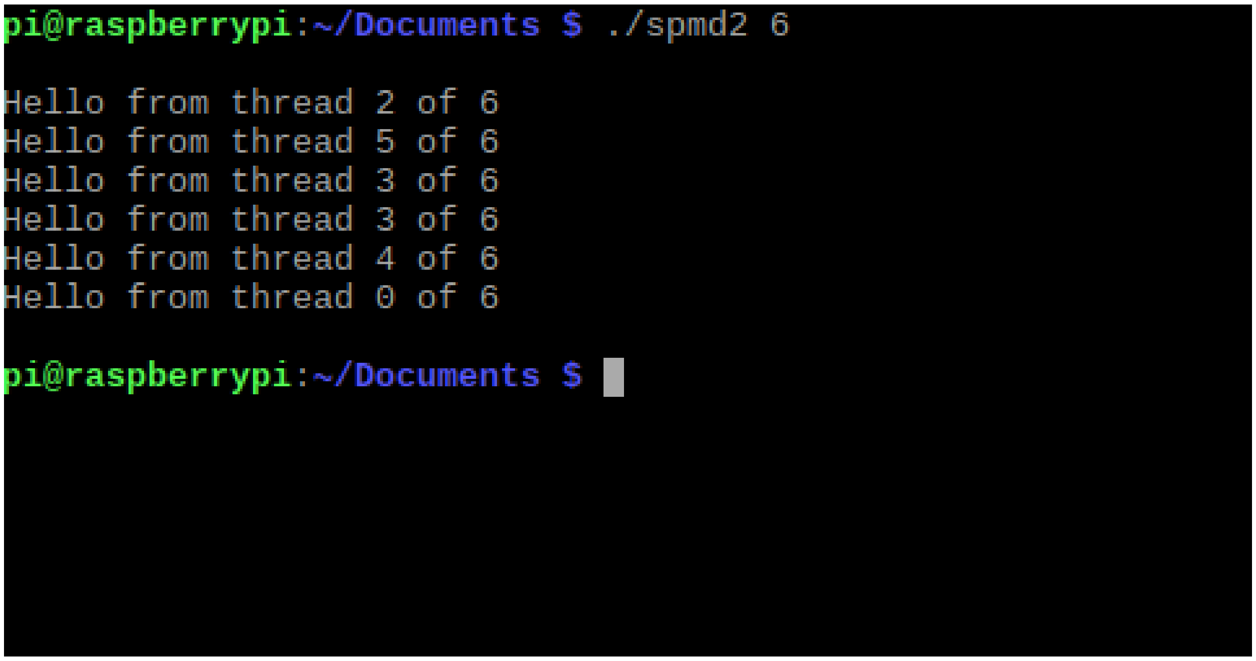
```
pi@raspberrypi:~/Documents $ gcc spmd2.c -o spmd2 -fopenmp
```

After succeeding in creating the executable file, it can be run with the command "./spmd2 4". The number 4 is the number of threads to fork, and with this code, the ID of each thread will be printed. The number of threads can be changed by entering a different value from 4.

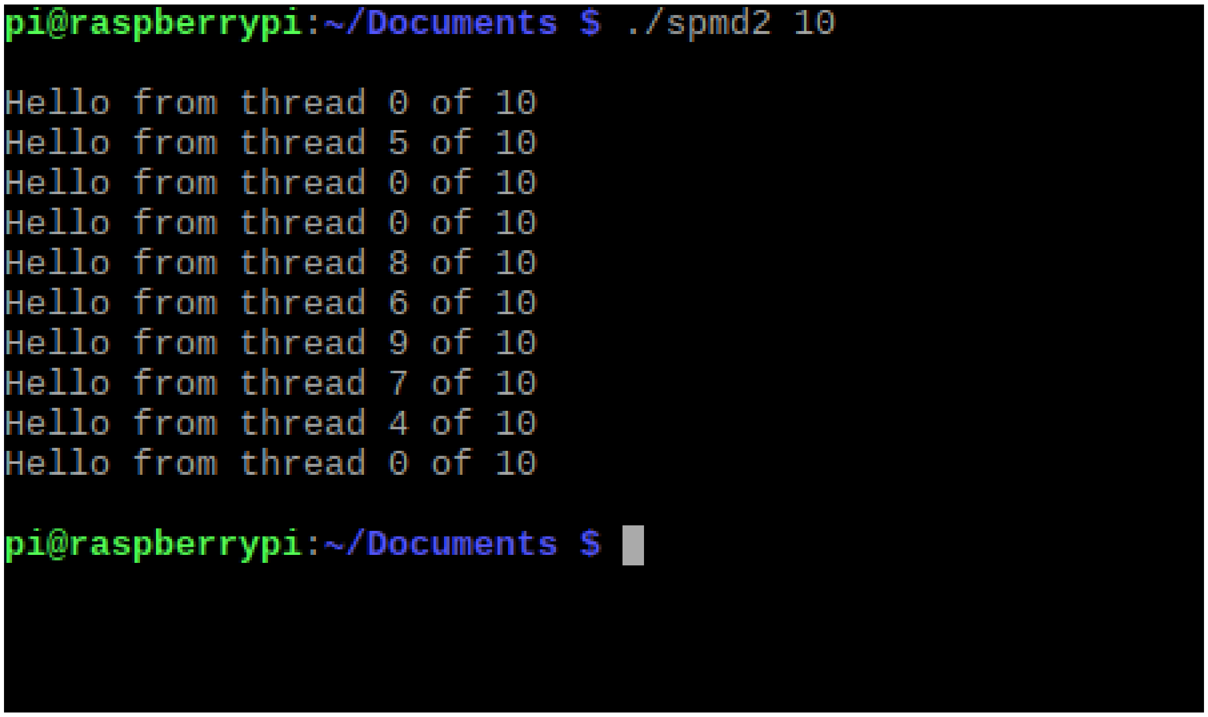


```
pi@raspberrypi:~/Documents $ ./spmd2

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:~/Documents $
```

Here are some more results with a different number of threads

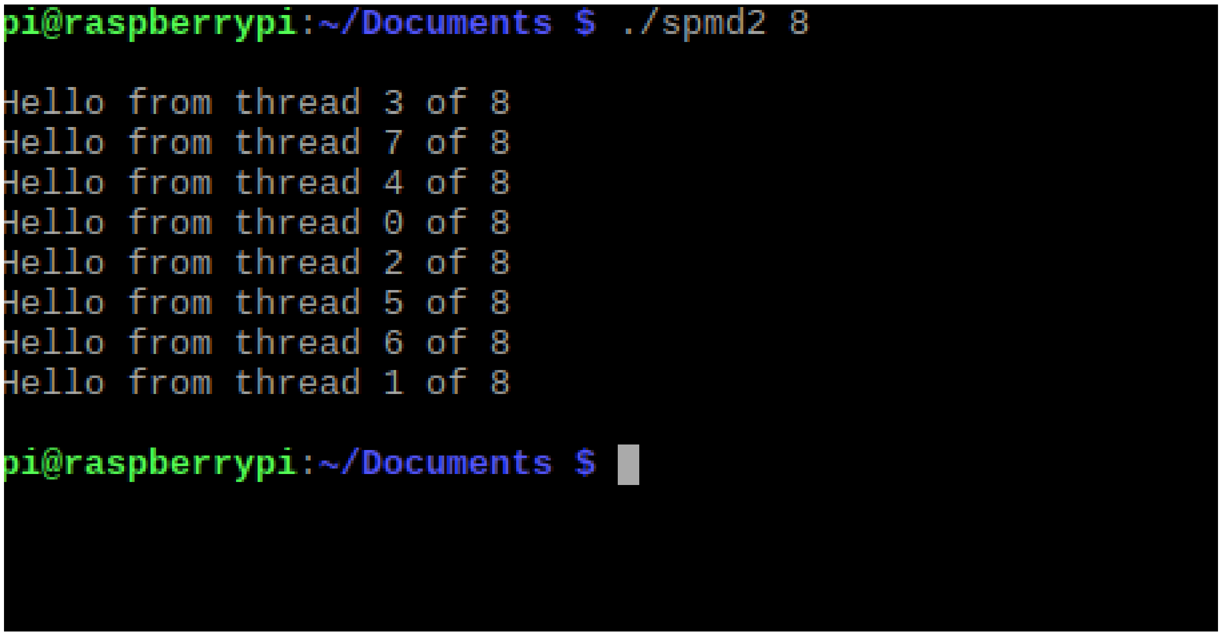


```
pi@raspberrypi:~/Documents $ ./spmd2 6

Hello from thread 2 of 6
Hello from thread 5 of 6
Hello from thread 3 of 6
Hello from thread 3 of 6
Hello from thread 4 of 6
Hello from thread 0 of 6

pi@raspberrypi:~/Documents $
```



```
pi@raspberrypi:~/Documents $ ./spmd2 10

Hello from thread 0 of 10
Hello from thread 5 of 10
Hello from thread 0 of 10
Hello from thread 0 of 10
Hello from thread 8 of 10
Hello from thread 6 of 10
Hello from thread 9 of 10
Hello from thread 7 of 10
Hello from thread 4 of 10
Hello from thread 0 of 10

pi@raspberrypi:~/Documents $
```

In this code, the two variables are declared outside of the block that is forked and runs in parallel. Thus, all the threads use the same memory address. This problem happens occasionally, but there is a chance that duplicate thread ID may appear. Here is one run that all the thread IDs are unique.



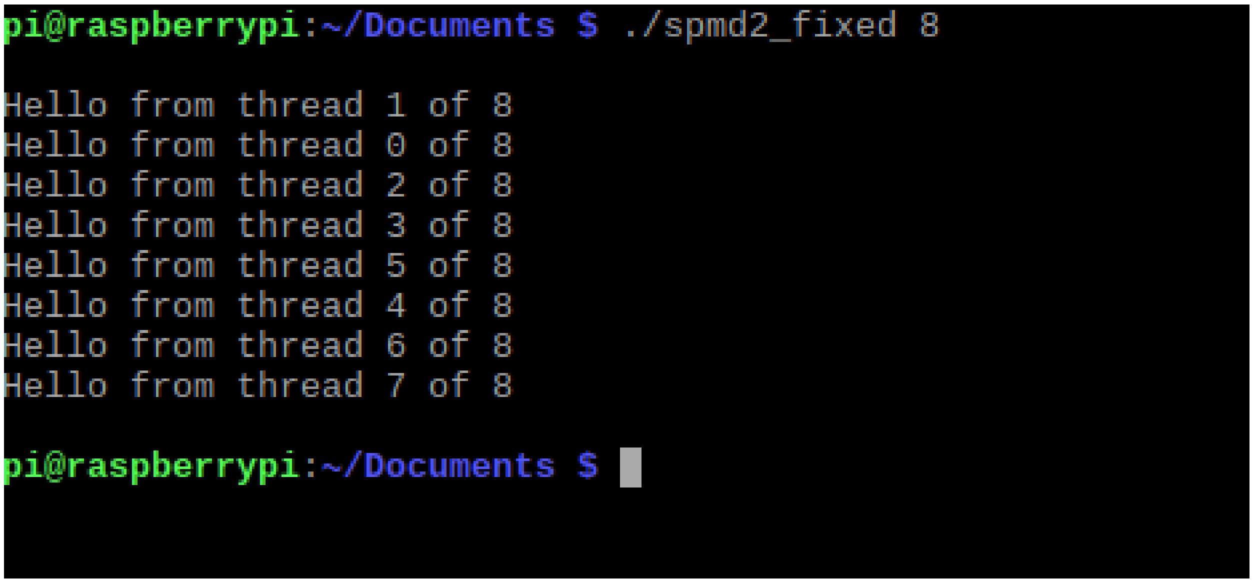
```
pi@raspberrypi:~/Documents $ ./spmd2 8

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

pi@raspberrypi:~/Documents $
```

To resolve this issue, the two variables just need to be declared inside of the block that will be forked and run. Here is the code snippet of the fixed version.

```
#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;
             }
```

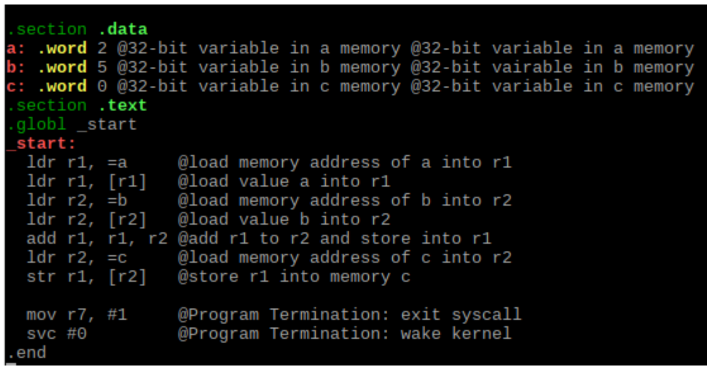
With this fix, all threads have their unique ID even though they are still randomly generated.



```
pi@raspberrypi:~/Documents $ ./spmd2_fixed 8

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

pi@raspberrypi:~/Documents $
```

❏ ARM Assembly Programming

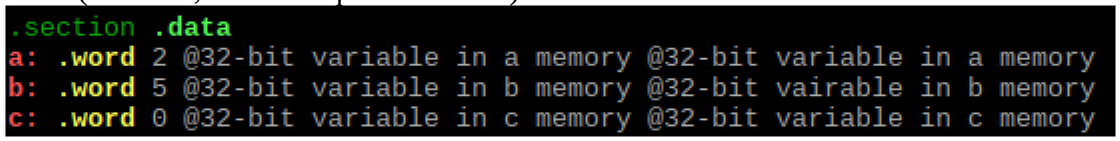
ARM Assembly Programing - Project A2 - Task 4 - Lab Report
Creating Second.s



```
.section .data
a: .word 2 @32-bit variable in a memory @32-bit variable in a memory
b: .word 5 @32-bit variable in b memory @32-bit vairable in b memory
c: .word 0 @32-bit variable in c memory @32-bit variable in c memory
.section .text
.globl _start
_start:
  ldr r1, =a     @load memory address of a into r1
  ldr r1, [r1]   @load value a into r1
  ldr r2, =b     @load memory address of b into r2
  ldr r2, [r2]   @load value b into r2
  add r1, r1, r2 @add r1 to r2 and store into r1
  ldr r2, =c     @load 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
```

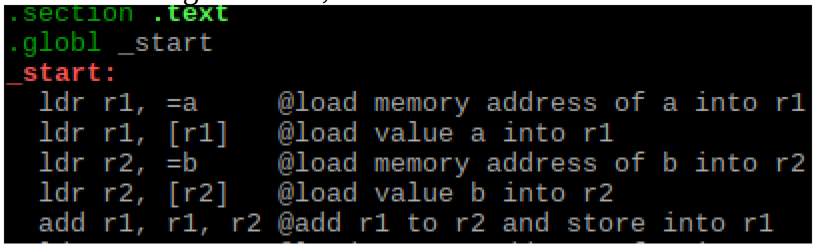
Code after typing up and second.s and adding comments

In the .section .data section of the code, 3 variables in memory are created of data spec word (in ARM, word is equal to 32 bits):



```
.section .data
a: .word 2 @32-bit variable in a memory @32-bit variable in a memory
b: .word 5 @32-bit variable in b memory @32-bit vairable in b memory
c: .word 0 @32-bit variable in c memory @32-bit variable in c memory
```

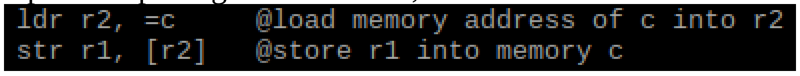
After declaring variables, we move on to the arithmetic:



```
.section .text
.globl _start
_start:
  ldr r1, =a     @load memory address of a into r1
  ldr r1, [r1]   @load value a into r1
  ldr r2, =b     @load memory address of b into r2
  ldr r2, [r2]   @load value b into r2
  add r1, r1, r2 @add r1 to r2 and store into r1
```

This arithmetic simply loads the variables into the registers and performs an addition of the value in variable a (which is in r1 register), and the value in variable b (which is in r2 register), and stores the result into r1 register.

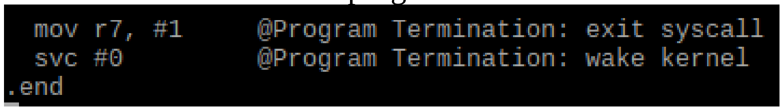
Upon completing the arithmetic, we must store the result value into variable in c memory:



```
ldr r2, =c     @load memory address of c into r2
str r1, [r2]   @store r1 into memory c
```

Because the memory address for variable c is stored in r2 register, we can use the str instruction to store the value of r1 into the memory address which is stored in r2 (in this case, memory c)

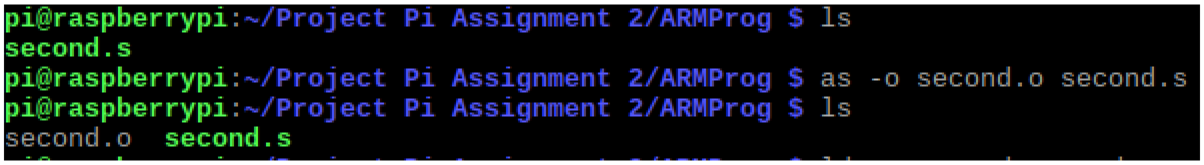
Now we can terminate the program:



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

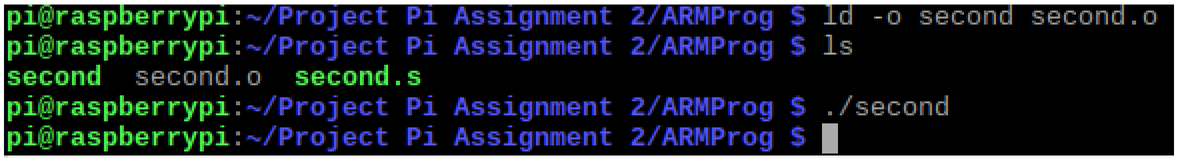
<u>Assembling, Linking, and Running Second.s</u>

Now we must assemble the second.s file to create the second.o file (objective file):



```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ as -o second.o second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
second.o  second.s
```

Once we have the objective file, we link the objective file to get an executable file which we can run:

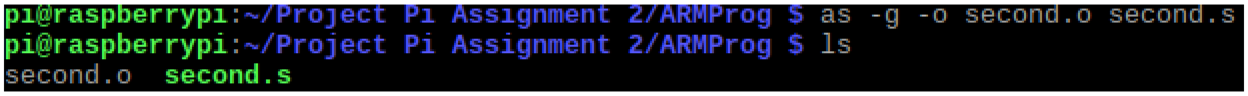


```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ld -o second second.o
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
second  second.o  second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ./second
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $
```

Although the executable file was created and run, there is no output. This is because there is no utilization of I/O (Input/Output)

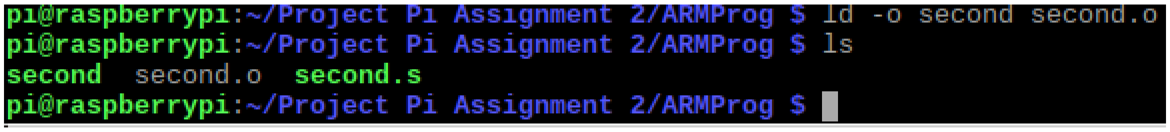
<u>Debugging Second.s</u>

In order to see the results of the arithmetic performed in second.s, we can debug it using GNU Debugger (gdb). We can begin this process by typing the -g flag when assembling the Second.s file:



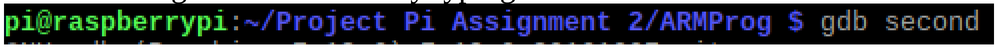
```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ as -g -o second.o second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
second.o  second.s
```

After creating the objective file, we can use the same linker command as before to create an executable file:



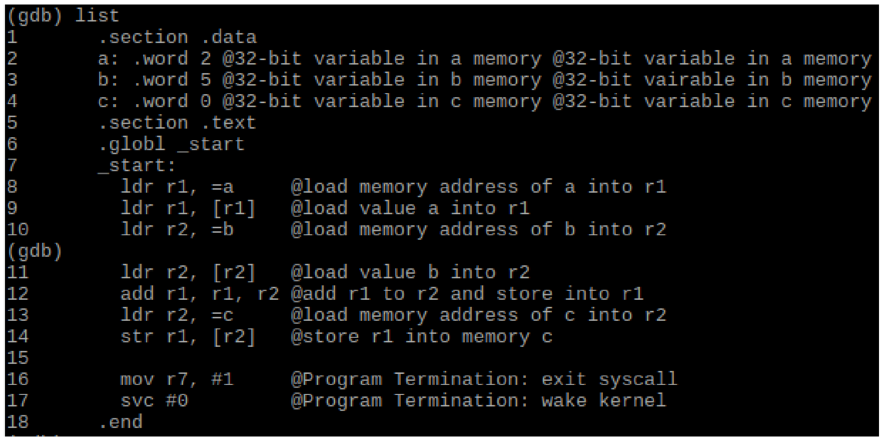
```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ld -o second second.o
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
second  second.o  second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $
```

We can debug this executable by typing this command:



```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ gdb second
```

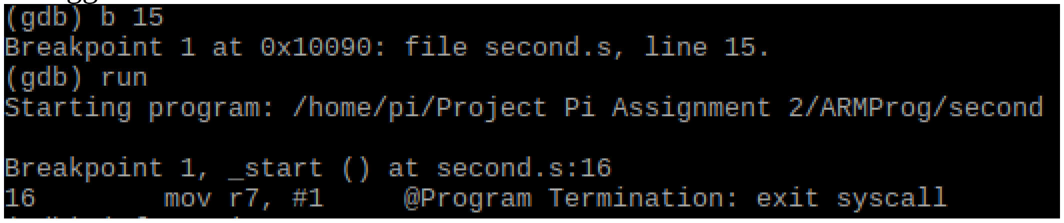
We use list to view the code with line numbers included:



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

Just as before, we can see that the variables will be declared, and the arithmetic will be performed.

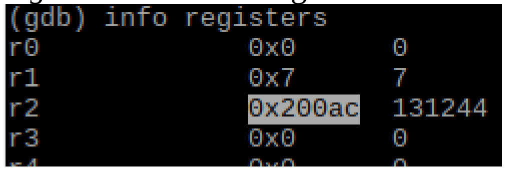
We can put a breakpoint at line 15 to check the results of the arithmetic, and run the debugger:



```
(gdb) b 15
Breakpoint 1 at 0x10090: file second.s, line 15.
(gdb) run
Starting program: /home/pi/Project Pi Assignment 2/ARMProg/second

Breakpoint 1, _start () at second.s:16
16          mov r7, #1     @Program Termination: exit syscall
```

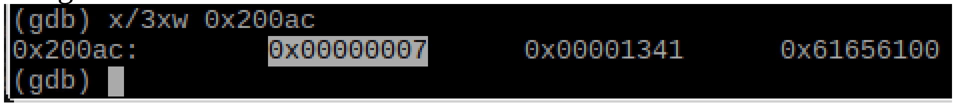
This will run the program but stop at line 16 (However, it is important to know that line 16 hasn't been executed yet. To execute this line and stop at the next, the step command is necessary)

Because the memory address for variable c is stored in register r2, we can type the *info registers* command to get the address:



```
(gdb) info registers
r0             0x0      0
r1             0x7      7
r2             0x200ac  131244
r3             0x0      0
```

Using the address above, we examine the memory at the address as a hexadecimal value using the *x/3xw 0x200ac* command:

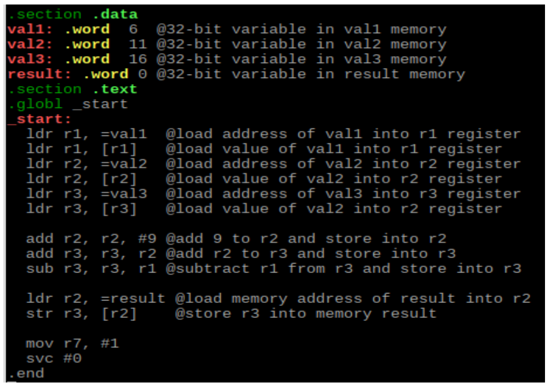


```
(gdb) x/3xw 0x200ac
0x200ac:        0x00000007      0x00001341      0x61656100
(gdb) 
```

The value in memory c is 0x00000007h (which is equal to 7 in decimal). This is indeed the correct answer.

<u>**Arithmetic2.s**</u>

Code for Arithmetic2.s:

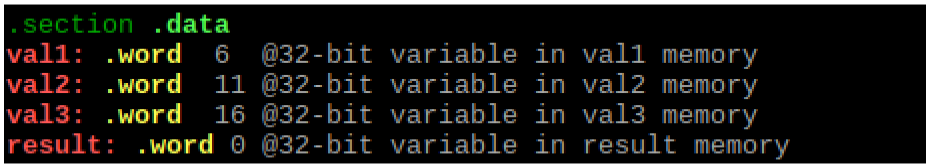


```
.section .data
val1: .word  6  @32-bit variable in val1 memory
val2: .word  11 @32-bit variable in val2 memory
val3: .word  16 @32-bit variable in val3 memory
result: .word 0 @32-bit variable in result memory
.section .text
.globl _start
_start:
  ldr r1, =val1  @load address of val1 into r1 register
  ldr r1, [r1]   @load value of val1 into r1 register
  ldr r2, =val2  @load address of val2 into r2 register
  ldr r2, [r2]   @load value of val2 into r2 register
  ldr r3, =val3  @load address of val3 into r3 register
  ldr r3, [r3]   @load value of val2 into r2 register

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

  ldr r2, =result @load memory address of result into r2
  str r3, [r2]    @store r3 into memory result

  mov r7, #1
  svc #0
.end
```

Arithmetic2 is a program designed to calculate the following expression:
Register = val2 + 9 + val3 - val1
Using precedence, we can also write the expression as such:
Register = ((val2 + 9) + val3) - val1

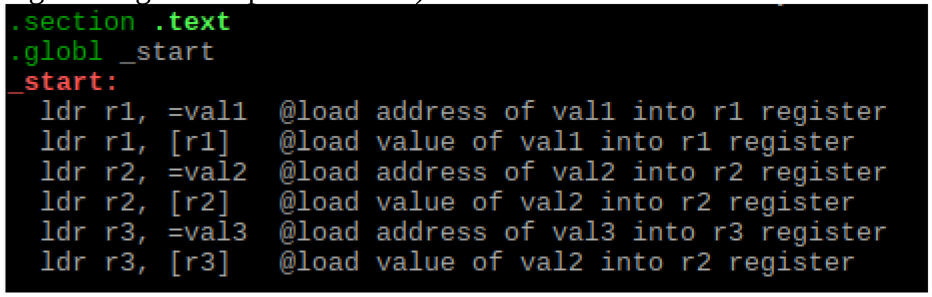
In this case, val1, val2, and val3 are all 32-bit integer memory variables. Assigning val1 = 6,
val2 = 11, val3 = 16, will give us the following expression:
Register = ((11 + 9) + 16) - 6
Register = ((20) + 16) - 6
Register = (36) - 6
Register = 30
As demonstrated above, performing the arithmetic of this expression would yield the result 30.

To do this arithmetic through the arithmetic2.s program, the variables should first be declared:



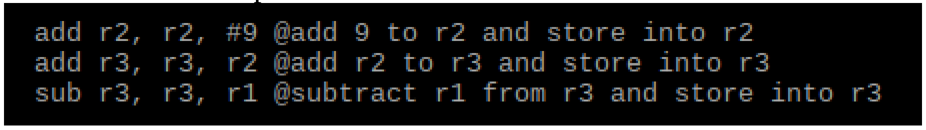
```
.section .data
val1: .word  6  @32-bit variable in val1 memory
val2: .word  11 @32-bit variable in val2 memory
val3: .word  16 @32-bit variable in val3 memory
result: .word 0 @32-bit variable in result memory
```

Then, we store the values from memory into registers (because we're dealing with a register-register implementation):



```
.section .text
.globl _start
_start:
  ldr r1, =val1  @load address of val1 into r1 register
  ldr r1, [r1]   @load value of val1 into r1 register
  ldr r2, =val2  @load address of val2 into r2 register
  ldr r2, [r2]   @load value of val2 into r2 register
  ldr r3, =val3  @load address of val3 into r3 register
  ldr r3, [r3]   @load value of val2 into r2 register
```

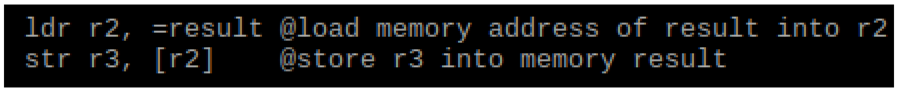
After the values are stored inside the registers, the registers can be used to perform the arithmetic of the expression:



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

The order of the arithmetic operations in the code is based off of the expression modified for precedence displayed above.

The result value is stored into r3 register, so the program is successful in its task. But additionally, it is possible to store this result into a memory (variable result) as we did in second.s:

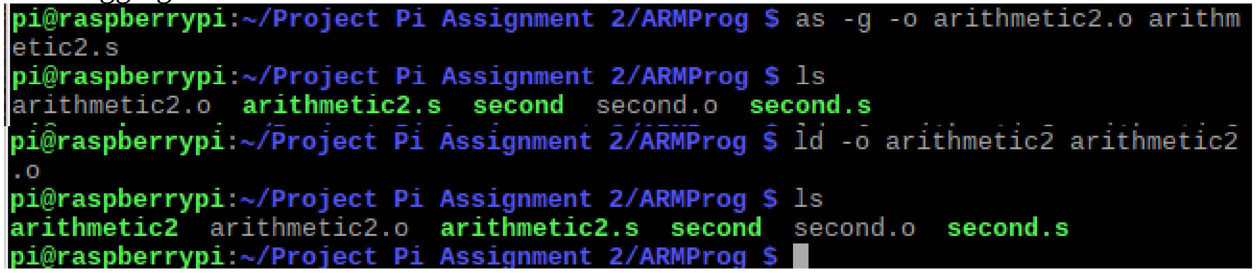


```
 ldr r2, =result @load memory address of result into r2
 str r3, [r2]    @store r3 into memory result
```

Theoretically, the arithmetic in the code is correct, but debugging the program can ensure that the code runs as intended.

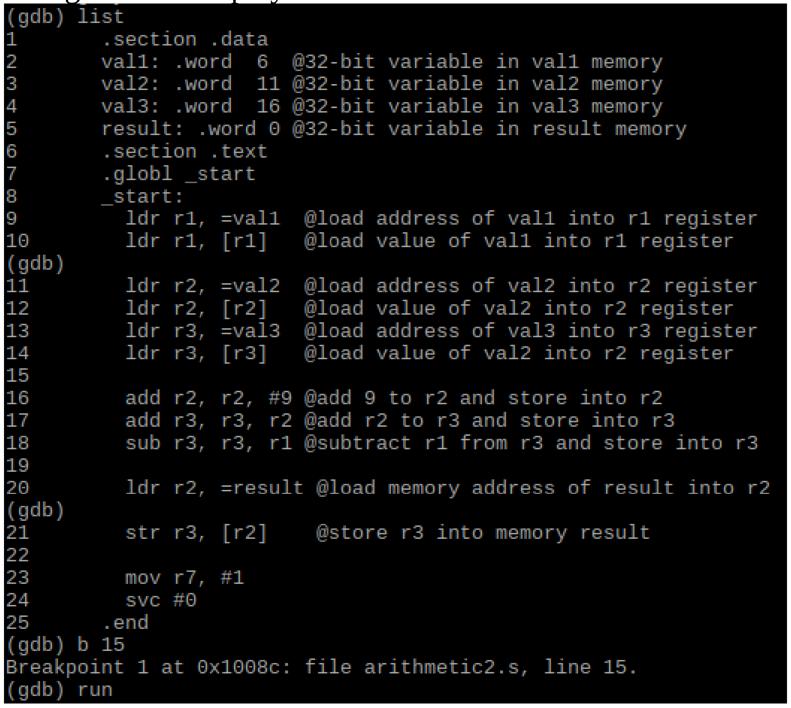
<u>Debugging Arithmetic.s</u>

To debug Arithmetic.s through the GNU debugger, performing the same steps as specified in debugging second.s should be sufficient to create the executable file:



```
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ as -g -o arithmetic2.o arithm
etic2.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
arithmetic2.o  arithmetic2.s  second  second.o  second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ld -o arithmetic2 arithmetic2
.o
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ ls
arithmetic2  arithmetic2.o  arithmetic2.s  second  second.o  second.s
pi@raspberrypi:~/Project Pi Assignment 2/ARMProg $ 
```

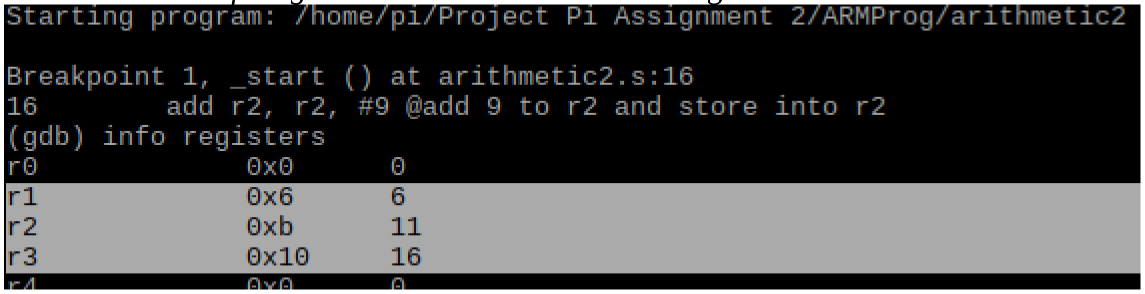
Using list will display the code with line numbers:



```
(gdb) list
1       .section .data
2       val1: .word  6  @32-bit variable in val1 memory
3       val2: .word  11 @32-bit variable in val2 memory
4       val3: .word  16 @32-bit variable in val3 memory
5       result: .word 0 @32-bit variable in result memory
6       .section .text
7       .globl _start
8       _start:
9         ldr r1, =val1  @load address of val1 into r1 register
10        ldr r1, [r1]   @load value of val1 into r1 register
(gdb)
11        ldr r2, =val2  @load address of val2 into r2 register
12        ldr r2, [r2]   @load value of val2 into r2 register
13        ldr r3, =val3  @load address of val3 into r3 register
14        ldr r3, [r3]   @load value of val2 into r2 register
15
16        add r2, r2, #9 @add 9 to r2 and store into r2
17        add r3, r3, r2 @add r2 to r3 and store into r3
18        sub r3, r3, r1 @subtract r1 from r3 and store into r3
19
20        ldr r2, =result @load memory address of result into r2
(gdb)
21        str r3, [r2]    @store r3 into memory result
22
23        mov r7, #1
24        svc #0
25      .end
(gdb) b 15
Breakpoint 1 at 0x1008c: file arithmetic2.s, line 15.
(gdb) run
```

Putting a breakpoint at line 15 will allow us to verify that the values have been properly loaded from the memory to registers, and to step through the arithmetic.

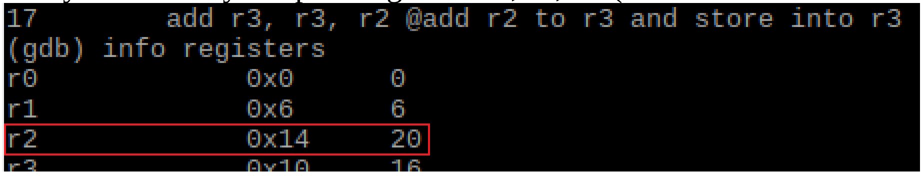
We can use the *info registers* command to check the registers:



```
Starting program: /home/pi/Project Pi Assignment 2/ARMProg/arithmetic2

Breakpoint 1, _start () at arithmetic2.s:16
16        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
```

As shown above, the values have been successfully loaded from memory into the registers.

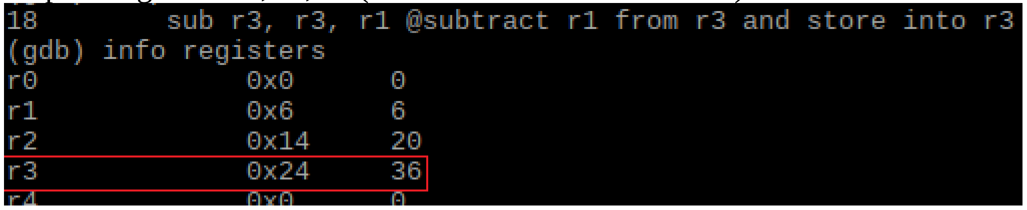
Now that the values have been loaded into registers, we can step through the arithmetic to verify its accuracy. Step through *add r2, r2, #9* (Add the immediate variable 9 to r2):



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

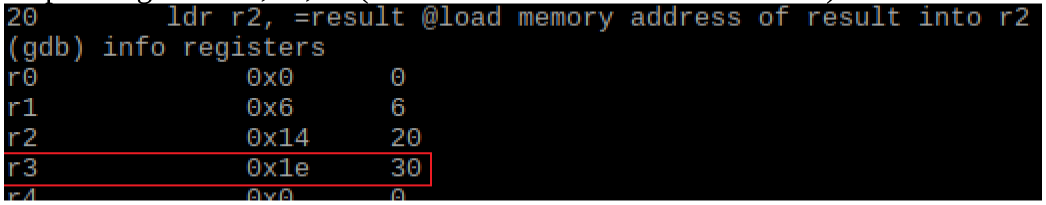
The line and line number shown while stepping does not correspond to the register values displayed adjacent because the line has not been executed yet

Step through *add r3, r3, r2* (add r2 to r3 and store into r3):



```
18        sub r3, r3, r1 @subtract r1 from r3 and store into r3
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x14     20
r3             0x24     36
```

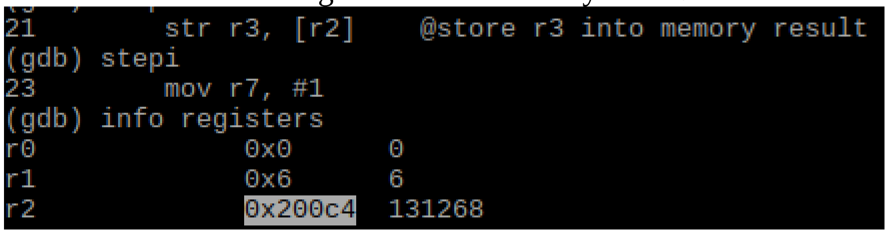
Step through *Sub r3, r3, r1* (subtract r1 from r3 and store into r3):



```
20        ldr r2, =result @load memory address of result into r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x14     20
r3             0x1e     30
```

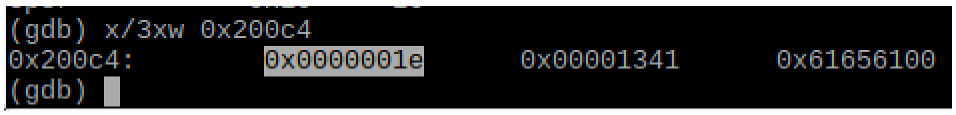
The value stored in r3 is the final result value, which is 30. This is correct according the expression.

Additionally, although not necessary, it is possible to view this result from memory because we stored the value in register *r3* into memory *result*:



```
21        str r3, [r2]    @store r3 into memory result
(gdb) stepi
23        mov r7, #1
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x200c4  131268
```

r2 contains memory address for result

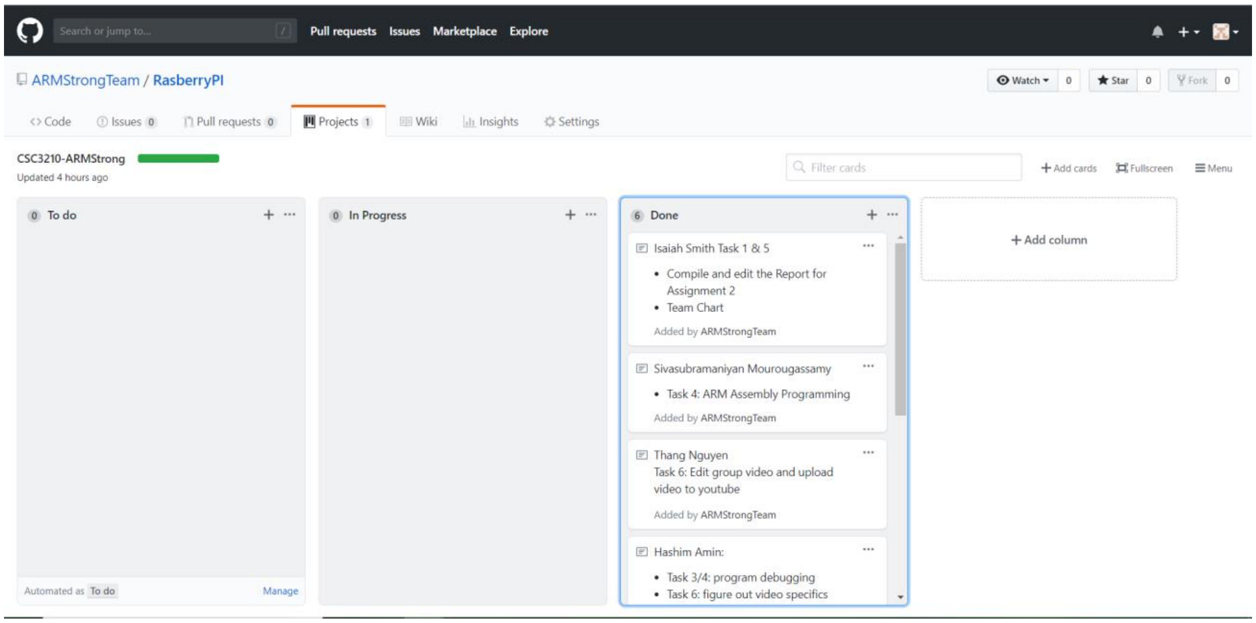


```
(gdb) x/3xw 0x200c4
0x200c4:        0x0000001e      0x00001341      0x61656100
(gdb) 
```

The value in result memory is 0x0000001eh, which is equal to 30 in decimal. This is the correct answer according to the expression.

- ❑ Appendix

Picture of Github project tab:



CSC3210-ARMStrong
Updated 4 hours ago

0 To do

0 In Progress

6 Done

Isaiah Smith Task 1 & 5
- Compile and edit the Report for Assignment 2
- Team Chart

Added by ARMStrongTeam

Sivasubramaniyan Mourougassamy
- Task 4: ARM Assembly Programming

Added by ARMStrongTeam

Thang Nguyen
Task 6: Edit group video and upload video to youtube

Added by ARMStrongTeam

Hashim Amin:
- Task 3/4: program debugging
- Task 6: figure out video specifics

Link to team slack:
- https://armstrongprojecteam.slack.com/messages/CFUT6BUJ1/

Link to YouTube video:
- https://www.youtube.com/watch?v=_WPQUGrHMW0&t=6s