Developing Soft and Parallel Programming Skills Using Project-Based Learning

Kishan Bhakta, Shilp Patel, Sean Staley, Mindy Tran, Donald Weaver
Spring 2019, A Wad Winning Group

Planning and Scheduling

For the second project, we had an easier time assigning the tasks and set up meetings since we figured out most of the kinks from the first project. We used the same procedure from the last project to pick our tasks. Even though we started earlier on this project, we didn't finish as early as we expect due to everyone having more workload for these past two weeks. Here is the breakdown of our schedule.

Name	**Email**	**Task**	**Duration (hours)**	**Dependency**	**Due Date**	**Note**
Mindy Tran (coordinator)	mtran42@student.gsu.edu	Report	5	Slack, table	2/22/19	Remember to print the report
Sean Staley	sstaley4@student.gsu.edu	ARM assembly program	3	ARM assembly program lab report	2/22/19	Take screenshots
Shilp Patel	Spatel255@student.gsu.edu	Parallel programming	2.5	Parallel programming lab report, GitHub	2/15/19	Take screenshots
Donald Weaver	Dweaver20@student.gsu.edu	Video	.5	Uploading video to YouTube, video editing	2/22/19	Make sure video is within time frame
Kishan Bhakta	kbhakta1@student.gsu.edu	Foundation of parallel programming	1.5	Lead the meeting for answering the questions	2/15/19	Make sure everyone understands it

Parallel Programming Skills

1. Identify the components on the Raspberry Pi B+.
   - ARM Cortex A-53 CPU (Broadcom BCM2837B0 SoC)
   - Dual Core VideoCore IV (1080p60)
   - 4 x USB 2.0 Port
   - 1 x 10/100 Ethernet Port
   - RCA Video / Audio Jack
   - CSI Camera Connector
   - HDMI Video / Audio Connector
   - Micro USB Connector (to power the Raspberry Pi B+)
   - Micro CD Card Slot
   - DSI Display Connector
   - Status LEDs
   - GPIO (General Purpose Input / Output) Header

2. How many cores does the Raspberry PI's B+ CPU have?
   - 1.4 GHz 64-bit Quad Core processor (***4***)

3. List three main differences between X86(CISC) and ARM Raspberry PI(RISC). Justify your answer and use your own words.
   - RISC (Reduced Instruction Set Computer)
     - has more general-purpose registers than CISC.
     - has a more simplified instruction set (100 instructions or less) which only operate on registers and uses a Load/Store memory model for memory access.
     - Only Load/Store instructions can access memory.
     - Load, Increment, Store
     - to first load the value at an address into a register, you increment it with the register, and store it back to the memory from the register.
   - CISC (Complex Instruction Set Computer)
     - is larger and has a richer instruction set and allows many complex instructions to access memory.
     - has more operations and addressing modes.
     - use little-endian format (Least significant byte to most significant byte [right to left]).

4. What is the difference between sequential and parallel computation and identify the practical significance of each?
   - Sequential processing requires the subscribing system to process instructions in the order they are received, and each instruction execution takes a similar amount of time to process. The next process begins processing only when the previous one is completed.
   - Parallel processing requires the subscribing system to run multiple instructions simultaneously. Processing time will vary for different instructions

5. Identify the basic form of data and task parallelism in computational problems
   - Data Parallelism is the simultaneous execution on multiple cores of the same function across the elements of a dataset.
   - Task Parallelism is the simultaneous execution on multiple cores of many different functions across the same or different datasets

6. Explain the differences between processes and threads
   - A process is the abstraction of a running program and does not share a memory with each other
   - A tread is a lightweight process that allows a single executable or process to be decomposed to smaller independent parts and share the common memory of the process they belong to.
   - An Operating System will schedule threads on separate cores as they become available.

7. What is OpenMP and what are the OpenMP pragmas?
   - OpenMP is a standard that compilers who implement it must adhere to.
   - OpenMP uses an implicit multithreading model in which the library handles thread creation and management. This ability makes the programmer's task less complex and less error-prone.
   - OpenMP programs initialize a group of threads to be used by a given program(often called a pool of thread). The threads will execute at the same time of the code specified by the programmer
   - OpenMP pragmas are compiler directives that enable the compiler to generate threaded code.

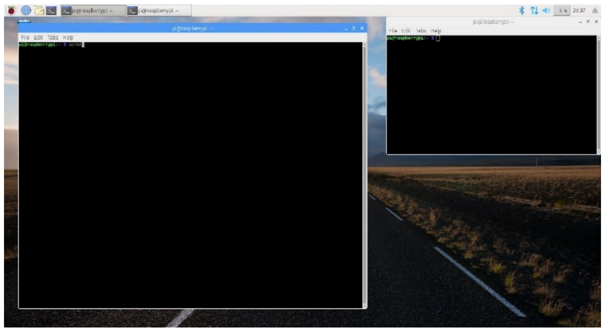
8. What application benefit from multi-core (list four)
   1. Database servers
   2. Webservers
   3. Compilers
   4. Scientific applications such as CAD/CAM

9. Why multicore? (why not single core, list four)
   - Multi-core CPU can process more processes at once, and applications with thread-level parallelism benefit from it.
   - It is difficult to make single-core clock frequencies higher.
   - Deeply pipelined circuits
     - heat problems
     - the speed of light problems
     - difficult design and verification
     - large design teams necessary
     - server farms need expensive air-conditioning
   - Many new applications are multithreaded
   - A popular trend in computer architecture leans more towards parallelism.

GETTING STARTED WITH THE RASPBERRY PI AND PARALLEL PROGRAMMING

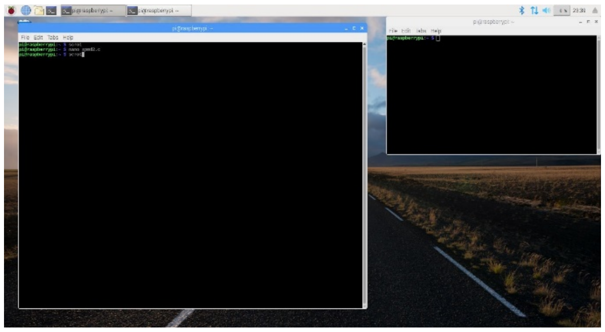
Written and executed by Shilp Patel

1.1
Open the terminal on raspberry pi:



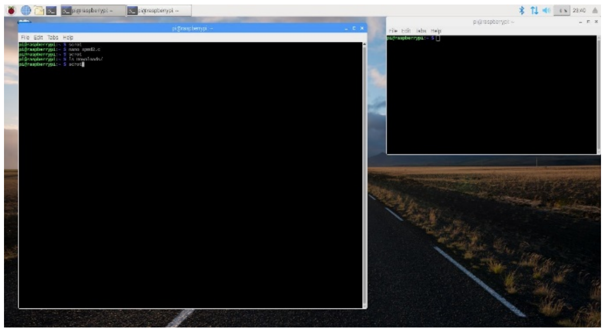
```
pi@raspberrypi:~ $ scrot
```

1.2
Open an editor that you run in the terminal called nano.



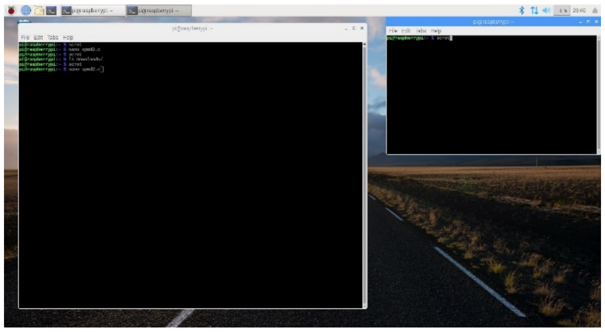
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ scrot
```

1.3.1.1
Auto-complete a command by using Tab as a shortcut key.



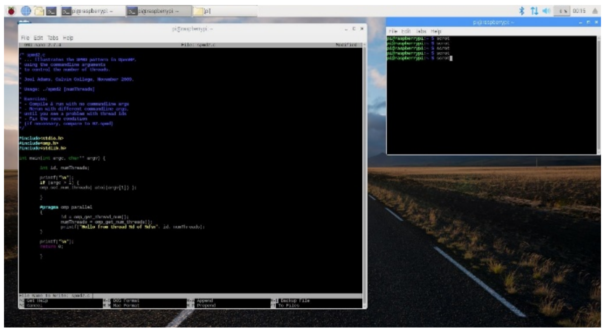
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ls Downloads/
pi@raspberrypi:~ $ scrot
```

1.3.2.2. & 2.1
Use UP arrow key goto to previously use command. Create an spmd2 file in nano editor by typing *nano spmd2.c* .



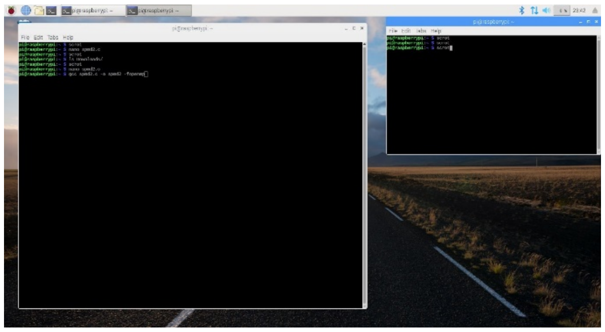
```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ls Downloads/
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
```

2.1 Conti.
Write the code provided by the instructor in the project on spmd2 file in nano editor. Use Control + W to write the code, Ctrl + O to save and Ctrl + X to exit the editor.



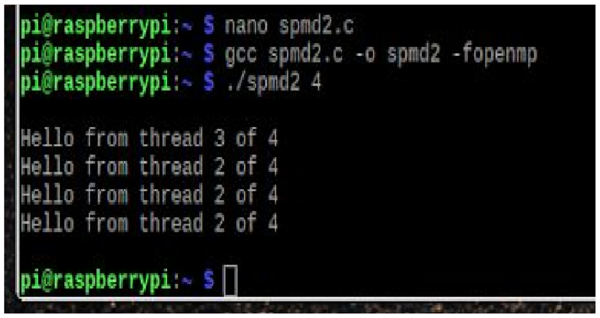
```
/* spmd2.c
 * ... illustrates the SPMD pattern in OpenMP,
 * using the commandline arguments
 * to control the number of threads.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./spmd2 [numThreads]
 *
 * Exercise:
 * - Compile & run with no commandline args
 * - Rerun with different commandline args,
 * until you see a problem with thread ids
 * - Fix the race condition
 * (if necessary, compare to 02.spmd)
 */

#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;

        }
```

2.2
Make the file an executable program by typing: *gcc spmd2.c -o spmd2 -fopenmp*



```
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ls Downloads/
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ nano spmd2.c
pi@raspberrypi:~ $ gcc spmd2.c -o spmd2 -fopenmp
```

2.2 Conti.
To run the spmd2 program, I used the following command:
./spmd2 4



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

pi@raspberrypi:~ $
```

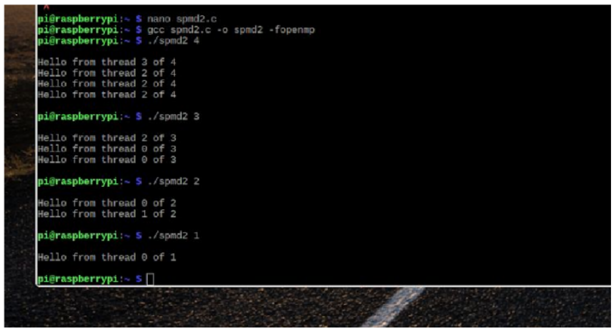
2.2 Conti.

In this part I ran the program using different number threads such as 3, 2 and 1, I used the following command:

./spmd2 3

./spmd2 2

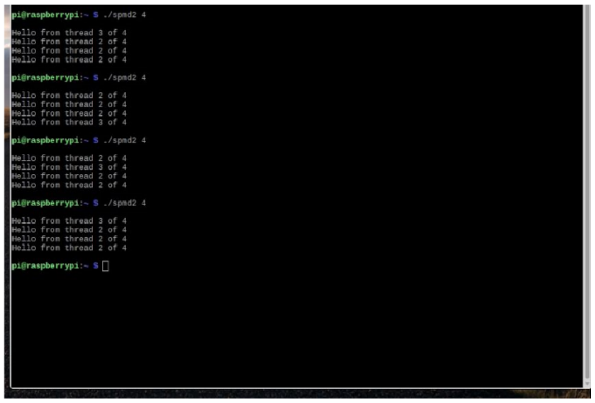
./spmd2 1



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

pi@raspberrypi:~ $ ./spmd2 3

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

pi@raspberrypi:~ $ ./spmd2 2

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

pi@raspberrypi:~ $ ./spmd2 1

Hello from thread 0 of 1

pi@raspberrypi:~ $
```

2.3

Here I ran the program 3 times more using 4 threads. As a result, I observed that the pattern repeats after executing the program three times.

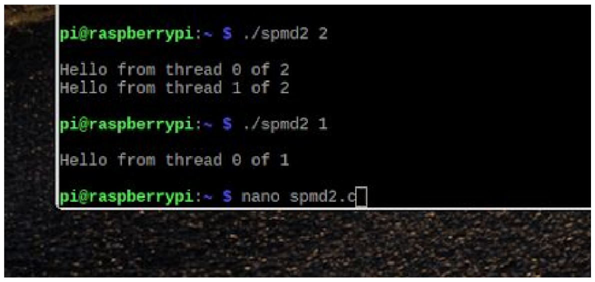


```
pi@raspberrypi:~ $ ./spmd2 4

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:~ $
```

2.4

Open the file in nano editor, I used the following command:

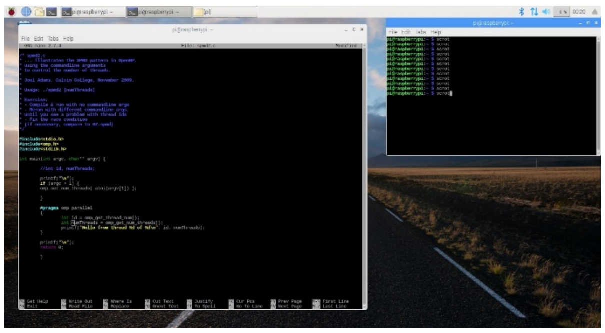
nano spmd2.c



```
pi@raspberrypi:~ $ ./spmd2 2

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

pi@raspberrypi:~ $ ./spmd2 1

Hello from thread 0 of 1

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

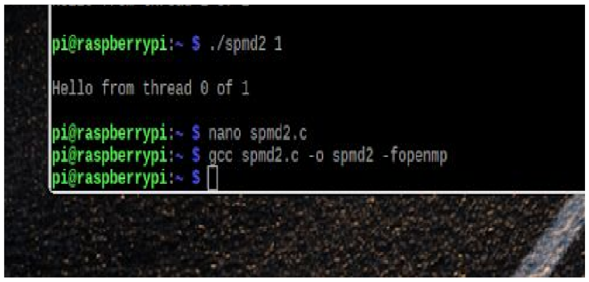
2.4.1

I fix the code by adding comment // line in front of line 5 and adding *int* in front of line 12 and 13.



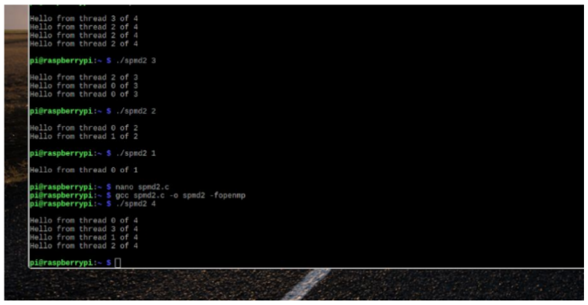
```
GNU nano 2.7.4                    File: spmd2.c                    Modified

/* spmd2.c
 * ... illustrates the SPMD pattern in OpenMP,
 * using the commandline arguments
 * to control the number of threads.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./spmd2 [numThreads]
 *
 * Exercise:
 * - Compile & run with no commandline args
 * - Rerun with different commandline args,
 * until you see a problem with thread ids
 * - Fix the race condition
 * (if necessary, compare to 02.spmd)
 */

#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;

        }
```

2.4.1 Conti.
Made the file an executable program by typing:
gcc spmd2.c -o spmd2 -fopenmp



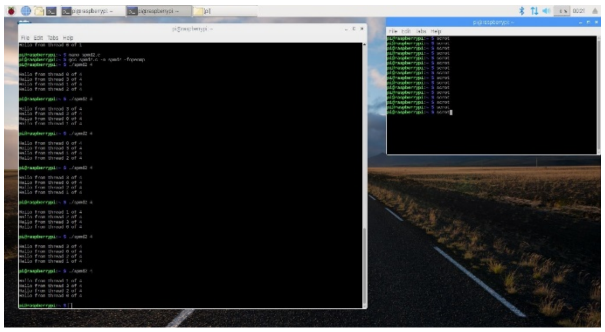
```
pi@raspberrypi:~ $ ./spmd2 1

Hello from thread 0 of 1

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

2.4.1 Conti.
To run the spmd2 program I used the following command again:
./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 3

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

pi@raspberrypi:~ $ ./spmd2 2

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

pi@raspberrypi:~ $ ./spmd2 1

Hello from thread 0 of 1

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

2.4.1 Conti.
Here I ran the program multiple times more using 4 threads. As a result, I observed there is a different pattern every time I execute the code and it rarely repeats itself.



```
Hello from thread 0 of 1

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

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

pi@raspberrypi:~ $ 
```

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

CODES USED IN THIS PROJECT:

2.2 BEFORE FIX

```
/* spmd2.c
* … illustrates the SPMD pattern in OpenMP,
* using the commandline arguments
* to control the number of threads.
*
* Joel Adams, Calvin College, November
2009.
* Usage: ./.spmd2 [numThreads]
*
* Exercise:
* - Compile & run with no commandline args
* - Rerun with different commandline args,
* until you see a problem with thread is
* - Fix the race condition
* (if necessary, compare to 02.spmd)
*/

#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();
                int numThreads =
omp_get_num_threads();
                printf("Hello from thread %d
of %d\n", id, numThreads);
        }

        printf("\n");
        return 0;

        }
```

2.4 AFTER FIX

```
/* spmd2.c
* … illustrates the SPMD pattern in OpenMP,
* using the commandline arguments
* to control the number of threads.
*
* Joel Adams, Calvin College, November
2009.
* Usage: ./.spmd2 [numThreads]
*
* Exercise:
* - Compile & run with no commandline args
* - Rerun with different commandline args,
* until you see a problem with thread is
* - Fix the race condition
* (if necessary, compare to 02.spmd)
*/

#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();
                int numThreads =
omp_get_num_threads();
                printf("Hello from thread %d
of %d\n", id, numThreads);
        }

        printf("\n");
        return 0;

        }
```

ARM Assembly Programming

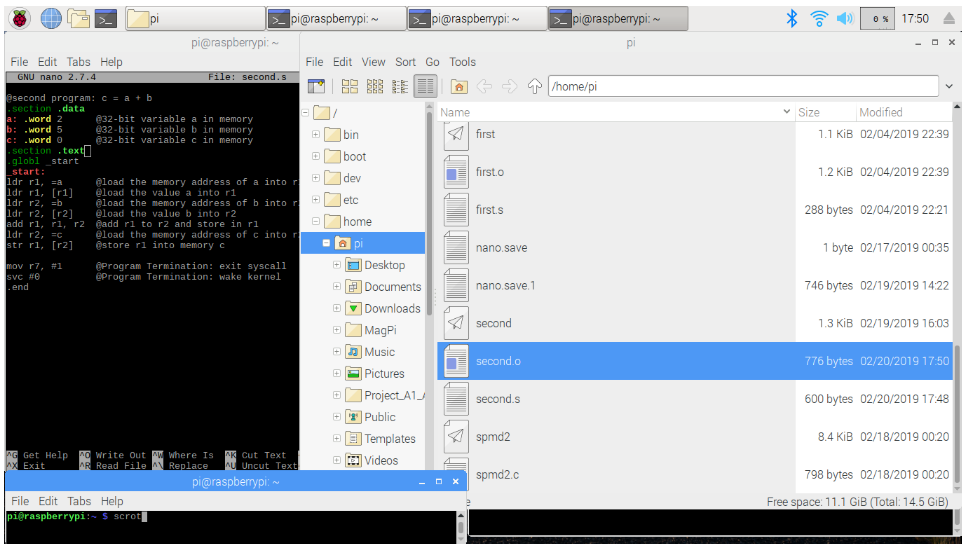
Executed by Sean Staley

In this assignment I wrote two arithmetic programs that performed operations and stored the values of these operations into memory variables and registers. The first program is shown below.

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

This program initializes three unsigned word variables of 32-bit length in the .data section of the code. The variables a and b are added together and stored in the memory location of variable c

Once the program has been built and saved Typing the command as **-o second.o second.s** into the terminal creates an objective file, a machine language translation of the ASCII text file.

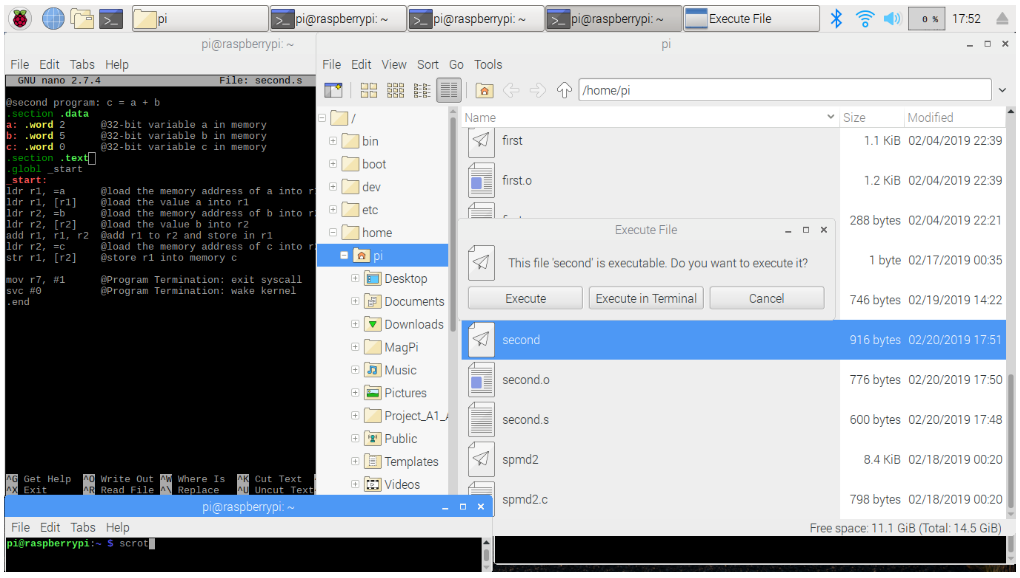


```
  GNU nano 2.7.4                      File: second.s

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

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

```
Name            Size        Modified
first           1.1 KiB     02/04/2019 22:39
first.o         1.2 KiB     02/04/2019 22:39
first.s         288 bytes   02/04/2019 22:21
nano.save       1 byte      02/17/2019 00:35
nano.save.1     746 bytes   02/19/2019 14:22
second          1.3 KiB     02/19/2019 16:03
second.o        776 bytes   02/20/2019 17:50
second.s        600 bytes   02/20/2019 17:48
spmd2           8.4 KiB     02/18/2019 00:20
spmd2.c         798 bytes   02/18/2019 00:20
Free space: 11.1 GiB (Total: 14.5 GiB)
```

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

The command "**ld -o second second.o**" creates an executable file. The linker reads the objective file shown above and translates it into this executable.



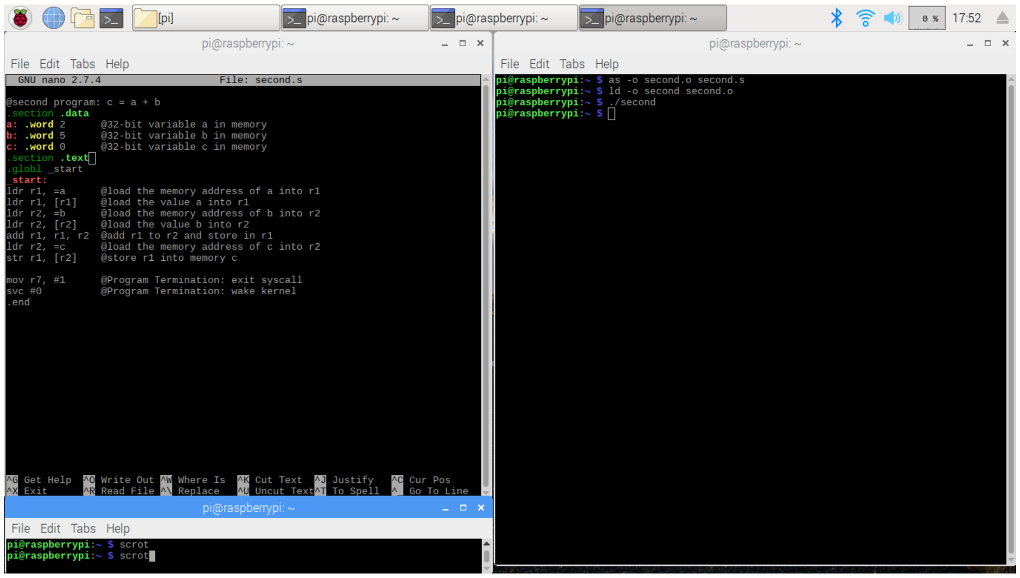
```
  GNU nano 2.7.4                      File: second.s

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

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

This file 'second' is executable. Do you want to execute it?

```
Name            Size        Modified
first           1.1 KiB     02/04/2019 22:39
first.o         1.2 KiB     02/04/2019 22:39
                288 bytes   02/04/2019 22:21
                1 byte      02/17/2019 00:35
                746 bytes   02/19/2019 14:22
second          916 bytes   02/20/2019 17:51
second.o        776 bytes   02/20/2019 17:50
second.s        600 bytes   02/20/2019 17:48
spmd2           8.4 KiB     02/18/2019 00:20
spmd2.c         798 bytes   02/18/2019 00:20
Free space: 11.1 GiB (Total: 14.5 GiB)
```

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

The command "**./second**" runs the program. The operating systems loader reads the executable file into the memory and branches the CPU to the file's starting address. The executable file does not output any information to the screen however, because we have not commanded it to do so. It simply does our calculation and stores in in the memory.



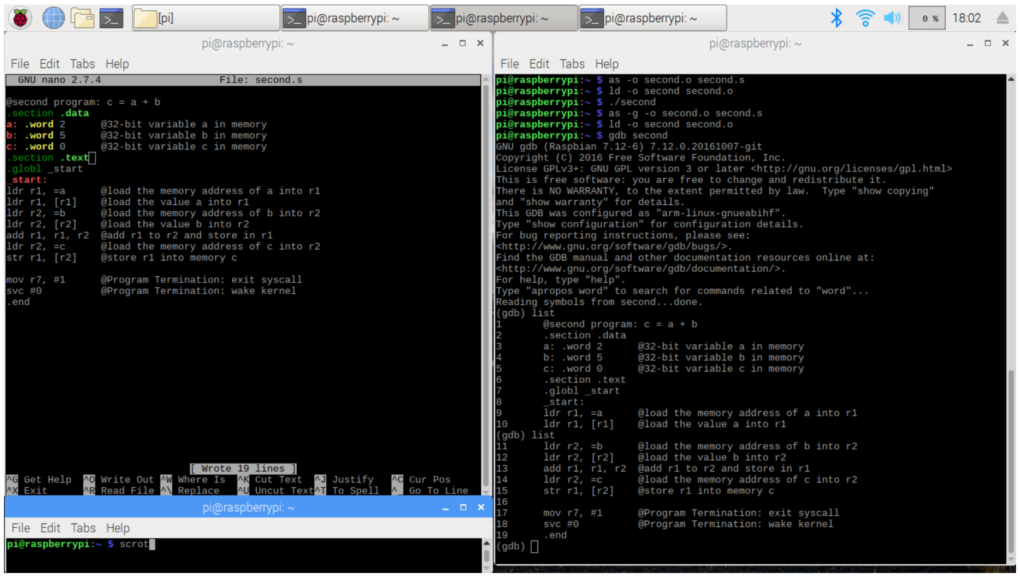
```
GNU nano 2.7.4                    File: second.s

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

```
pi@raspberrypi:~ $ as -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ ./second
pi@raspberrypi:~ $
```

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

[pi]
pi@raspberrypi: ~
17:52
File Edit Tabs Help
^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

We can go into the GDB debugger to see what is going on in the program using the command "**as -g -o second.o second.s**" shows us the number lines of the code when we debug. The command "**gdb second**" launches us into the debugger, and the "list" command shows us ten lines of code.

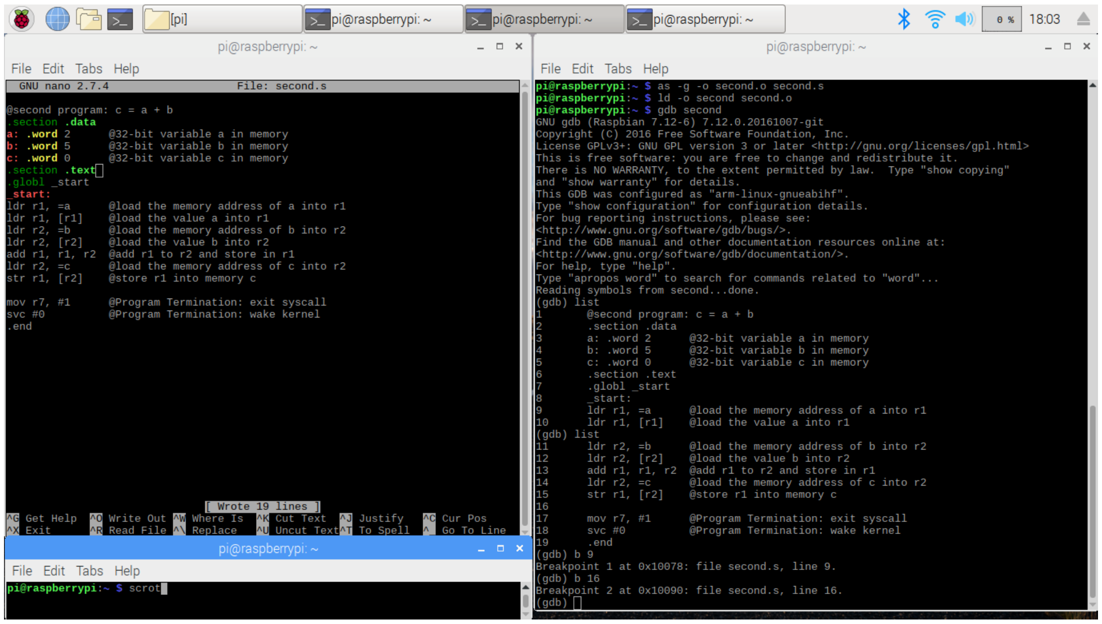


```
pi@raspberrypi:~ $ as -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ ./second
pi@raspberrypi:~ $ as -g -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ gdb second
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from second...done.
(gdb) 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 in r1
14      ldr r2, =c      @load the memory address of c into r2
15      str r1, [r2]    @store r1 into memory c
16
17      mov r7, #1      @Program Termination: exit syscall
18      svc #0          @Program Termination: wake kernel
19      .end
(gdb)
```

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

[pi]
pi@raspberrypi: ~
18:02
File Edit Tabs Help
GNU nano 2.7.4 File: second.s
[Wrote 19 lines]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line

Setting a breakpoint will stop the debugger from running at the line before that segment of code. Using the **"stepi"** command allows us to step into each line individually in the debugger.

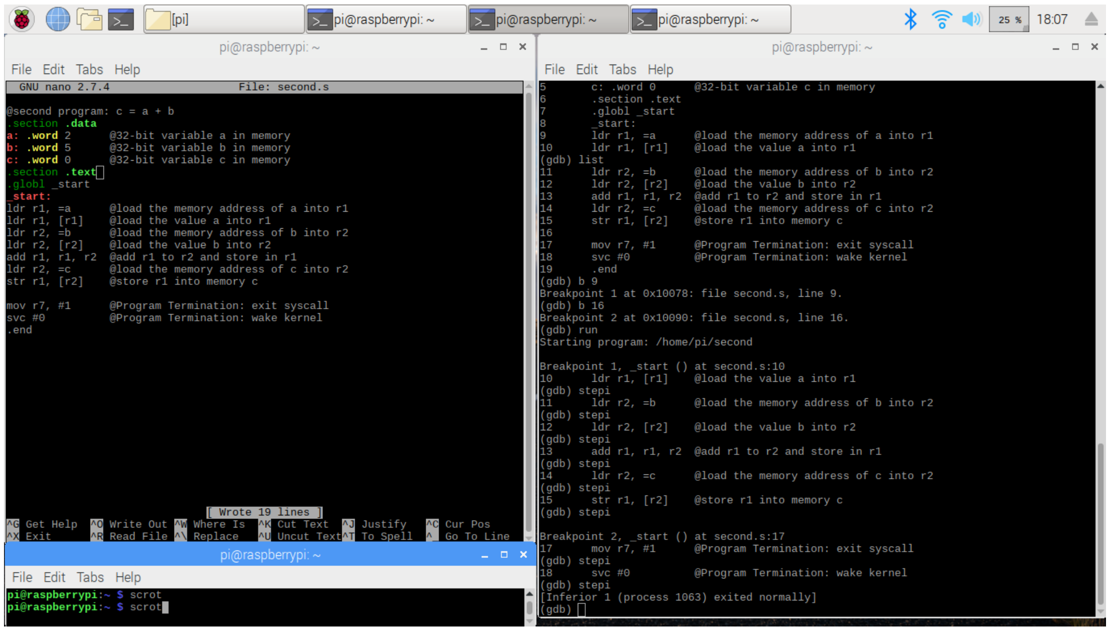


```
  GNU nano 2.7.4                    File: second.s

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

                              [ Wrote 19 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

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

```
pi@raspberrypi:~ $ as -g -o second.o second.s
pi@raspberrypi:~ $ ld -o second second.o
pi@raspberrypi:~ $ gdb second
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from second...done.
(gdb) 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 in r1
14      ldr r2, =c      @load the memory address of c into r2
15      str r1, [r2]    @store r1 into memory c
16
17      mov r7, #1      @Program Termination: exit syscall
18      svc #0          @Program Termination: wake kernel
19      .end
(gdb) b 9
Breakpoint 1 at 0x10078: file second.s, line 9.
(gdb) b 16
Breakpoint 2 at 0x10090: file second.s, line 16.
(gdb)
```

The screenshot below shows each line of code individually in the debugger as we use the command, **"stepi."**



```
  GNU nano 2.7.4                    File: second.s

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

                              [ Wrote 19 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line
```

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

```
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 in r1
14      ldr r2, =c      @load the memory address of c into r2
15      str r1, [r2]    @store r1 into memory c
16
17      mov r7, #1      @Program Termination: exit syscall
18      svc #0          @Program Termination: wake kernel
19      .end
(gdb) b 9
Breakpoint 1 at 0x10078: file second.s, line 9.
(gdb) b 16
Breakpoint 2 at 0x10090: file second.s, line 16.
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:10
10      ldr r1, [r1]    @load the value a into r1
(gdb) stepi
11      ldr r2, =b      @load the memory address of b into r2
(gdb) stepi
12      ldr r2, [r2]    @load the value b into r2
(gdb) stepi
13      add r1, r1, r2  @add r1 to r2 and store in r1
(gdb) stepi
14      ldr r2, =c      @load the memory address of c into r2
(gdb) stepi
15      str r1, [r2]    @store r1 into memory c
(gdb) stepi

Breakpoint 2, _start () at second.s:17
17      mov r7, #1      @Program Termination: exit syscall
(gdb) stepi
18      svc #0          @Program Termination: wake kernel
(gdb) stepi
[Inferior 1 (process 1063) exited normally]
(gdb)
```

Using the command x/3w at the memory address allows us to see 3 hexadecimal word values at the specified address. Breakpoint 1 is set at line 10 at the memory address 0x10078. The hexadecimal word values at this address before the addition has taken place are

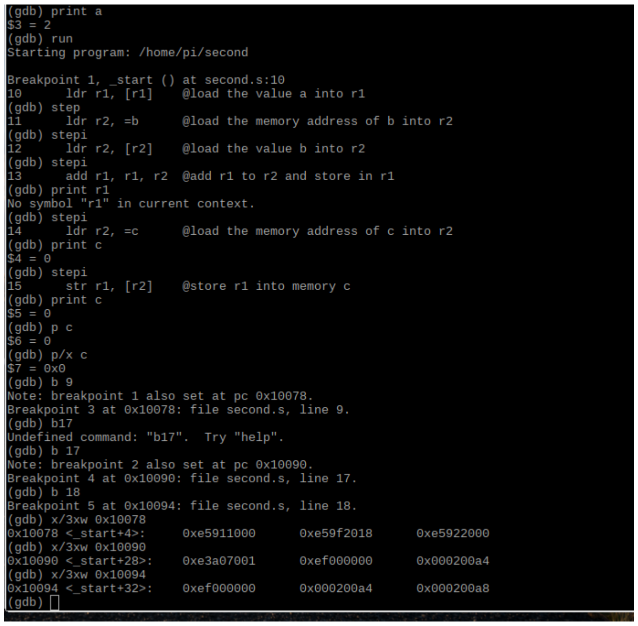
0xe5911000 0xe59f2018 0xe5922000

The second address I investigated was at line 17 just before the final calculation.

0xe3a07001 0xef000000 0xe000200a4

The third was at line 18 after all the code had been executed. The three word values here were,

0xef000000 0xe000200a4 0xe000200a8



```
(gdb) print a
$3 = 2
(gdb) run
Starting program: /home/pi/second

Breakpoint 1, _start () at second.s:10
10      ldr r1, [r1]    @load the value a into r1
(gdb) step
11      ldr r2, =b      @load the memory address of b into r2
(gdb) stepi
12      ldr r2, [r2]    @load the value b into r2
(gdb) stepi
13      add r1, r1, r2  @add r1 to r2 and store in r1
(gdb) print r1
No symbol "r1" in current context.
(gdb) stepi
14      ldr r2, =c      @load the memory address of c into r2
(gdb) print c
$4 = 0
(gdb) stepi
15      str r1, [r2]    @store r1 into memory c
(gdb) print c
$5 = 0
(gdb) p c
$6 = 0
(gdb) p/x c
$7 = 0x0
(gdb) b 9
Note: breakpoint 1 also set at pc 0x10078.
Breakpoint 3 at 0x10078: file second.s, line 9.
(gdb) b17
Undefined command: "b17".  Try "help".
(gdb) b 17
Note: breakpoint 2 also set at pc 0x10090.
Breakpoint 4 at 0x10090: file second.s, line 17.
(gdb) b 18
Breakpoint 5 at 0x10094: file second.s, line 18.
(gdb) x/3xw 0x10078
0x10078 <_start+4>:     0xe5911000      0xe59f2018      0xe5922000
(gdb) x/3xw 0x10090
0x10090 <_start+28>:    0xe3a07001      0xef000000      0x000200a4
(gdb) x/3xw 0x10094
0x10094 <_start+32>:    0xef000000      0x000200a4      0x000200a8
(gdb) 
```

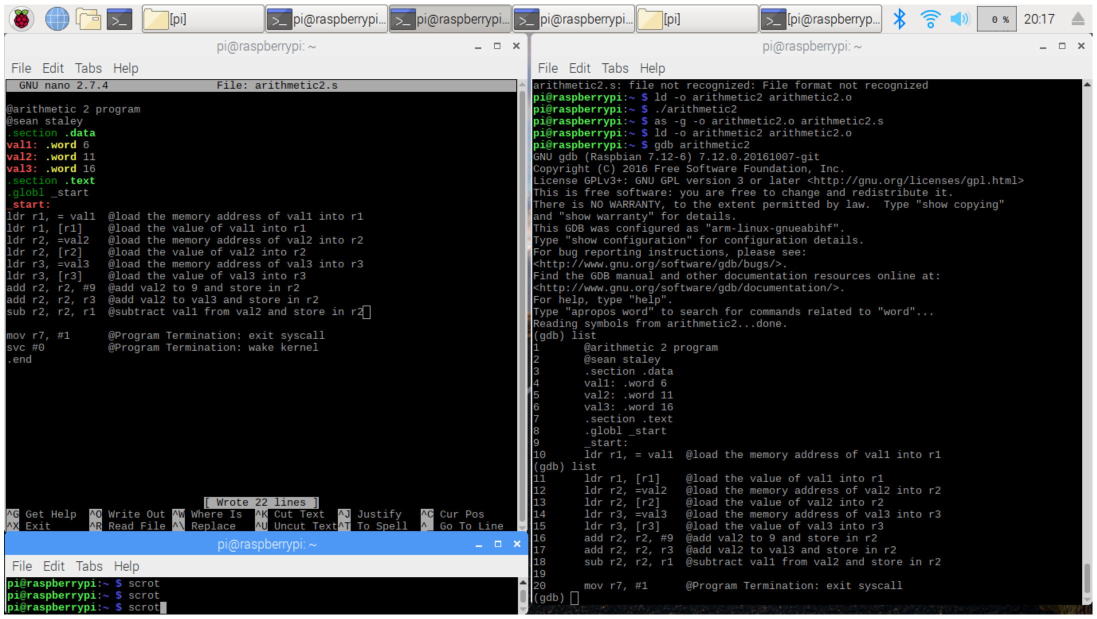
Arm Assembly Programming Part Two.

Written and executed by Sean Staley

In this assignment the goal was to write an arighmetic program that calculated, **Register = val2 + 9 + val3 - val1**, with the values in the memory being val2=11, val3=16, val1=6.

The code is as follows.

```
@arithmetic 2 program
@sean staley
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
ldr r1, = val1   @load the memory address of val1 into r1
ldr r1, [r1]     @load the value of val1 into r1
ldr r2, =val2    @load the memory address of val2 into r2
ldr r2, [r2]     @load the value of val2 into r2
ldr r3, =val3    @load the memory address of val3 into r3
ldr r3, [r3]     @load the value of val3 into r3
add r2, r2, #9   @add val2 to 9 and store in r2
add r2, r2, r3   @add val2 to val3 and store in r2
sub r2, r2, r1   @subtract val1 from val2 and store in r2

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

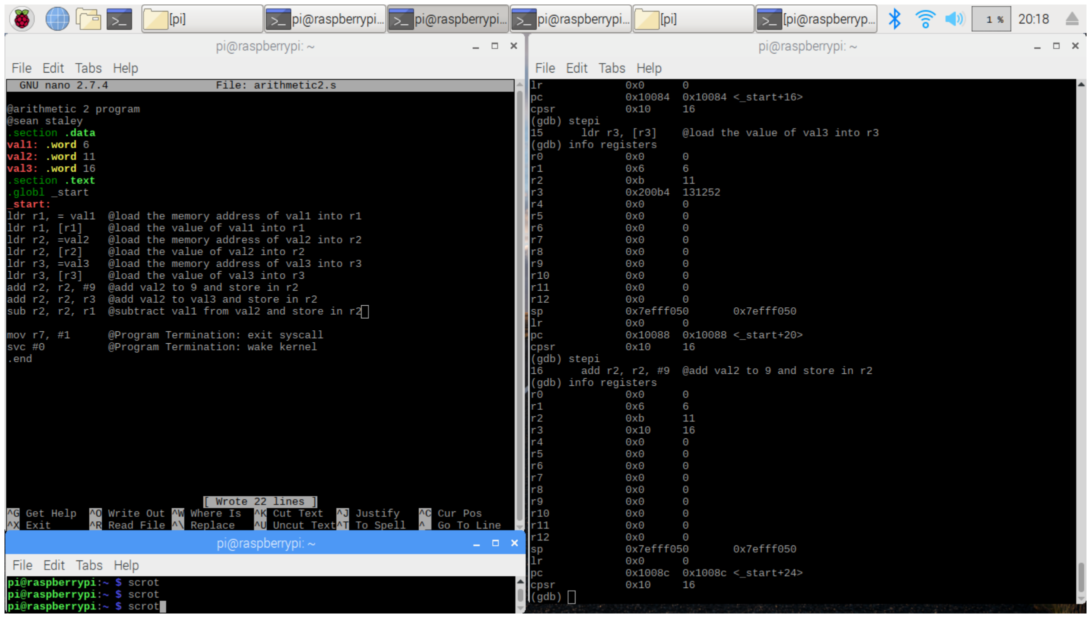


```
GNU nano 2.7.4                File: arithmetic2.s

@arithmetic 2 program
@sean staley
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
ldr r1, = val1  @load the memory address of val1 into r1
ldr r1, [r1]    @load the value of val1 into r1
ldr r2, =val2   @load the memory address of val2 into r2
ldr r2, [r2]    @load the value of val2 into r2
ldr r3, =val3   @load the memory address of val3 into r3
ldr r3, [r3]    @load the value of val3 into r3
add r2, r2, #9  @add val2 to 9 and store in r2
add r2, r2, r3  @add val2 to val3 and store in r2
sub r2, r2, r1  @subtract val1 from val2 and store in r2

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

                          [ Wrote 22 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

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

```
arithmetic2.s: file not recognized: File format not recognized
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ ./arithmetic2
pi@raspberrypi:~ $ as -g -o arithmetic2.o arithmetic2.s
pi@raspberrypi:~ $ ld -o arithmetic2 arithmetic2.o
pi@raspberrypi:~ $ gdb arithmetic2
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from arithmetic2...done.
(gdb) list
1       @arithmetic 2 program
2       @sean staley
3       .section .data
4       val1: .word 6
5       val2: .word 11
6       val3: .word 16
7       .section .text
8       .globl _start
9       _start:
10      ldr r1, = val1  @load the memory address of val1 into r1
(gdb) list
11      ldr r1, [r1]    @load the value of val1 into r1
12      ldr r2, =val2   @load the memory address of val2 into r2
13      ldr r2, [r2]    @load the value of val2 into r2
14      ldr r3, =val3   @load the memory address of val3 into r3
15      ldr r3, [r3]    @load the value of val3 into r3
16      add r2, r2, #9  @add val2 to 9 and store in r2
17      add r2, r2, r3  @add val2 to val3 and store in r2
18      sub r2, r2, r1  @subtract val1 from val2 and store in r2
19
20      mov r7, #1      @Program Termination: exit syscall
(gdb)
```

Below we see the GDB debugger. Using the code "info registers" shows us the values of the registers. As expected, after line 16, the values of r1, r2, and r3 are 6, 11, and 16 respectively.

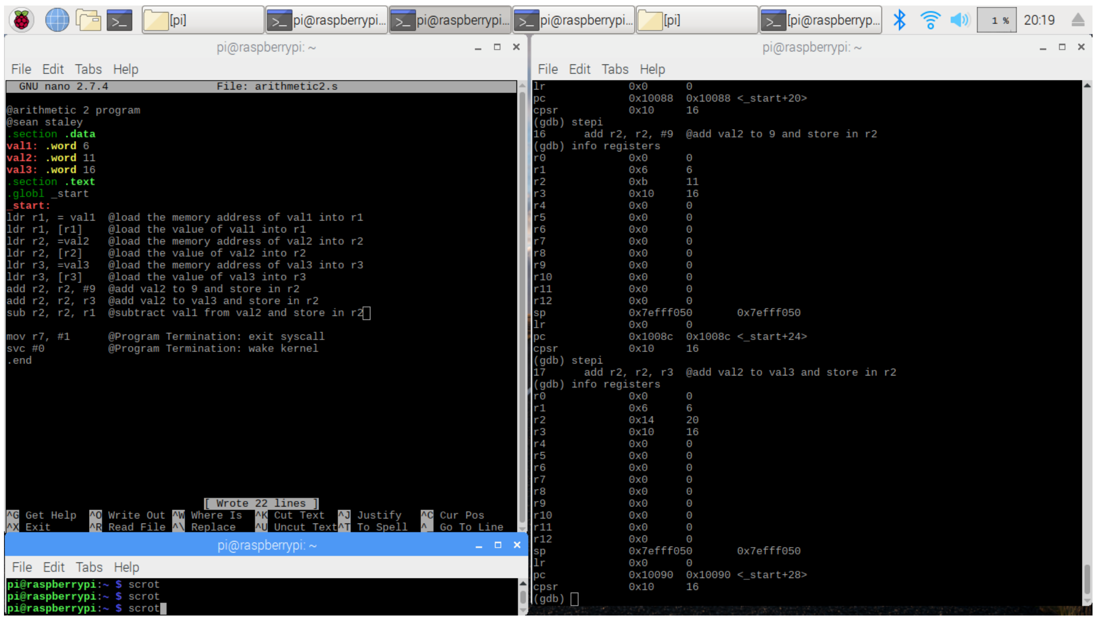


```
GNU nano 2.7.4                File: arithmetic2.s

@arithmetic 2 program
@sean staley
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
ldr r1, = val1  @load the memory address of val1 into r1
ldr r1, [r1]    @load the value of val1 into r1
ldr r2, =val2   @load the memory address of val2 into r2
ldr r2, [r2]    @load the value of val2 into r2
ldr r3, =val3   @load the memory address of val3 into r3
ldr r3, [r3]    @load the value of val3 into r3
add r2, r2, #9  @add val2 to 9 and store in r2
add r2, r2, r3  @add val2 to val3 and store in r2
sub r2, r2, r1  @subtract val1 from val2 and store in r2

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

                          [ Wrote 22 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

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

```
lr             0x0      0
pc             0x10084  0x10084 <_start+16>
cpsr           0x10     16
(gdb) stepi
15      ldr r3, [r3]    @load the value of val3 into r3
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0xb      11
r3             0x200b4  131252
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             0x7efff050       0x7efff050
lr             0x0      0
pc             0x10088  0x10088 <_start+20>
cpsr           0x10     16
(gdb) stepi
16      add r2, r2, #9  @add val2 to 9 and store in r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0xb      11
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x1008c  0x1008c <_start+24>
cpsr           0x10     16
(gdb)
```

After adding the decimal value 9 to the value in r2 and storing the result in the r2 register, we see that the value in the register is now 20! Just as we hoped.

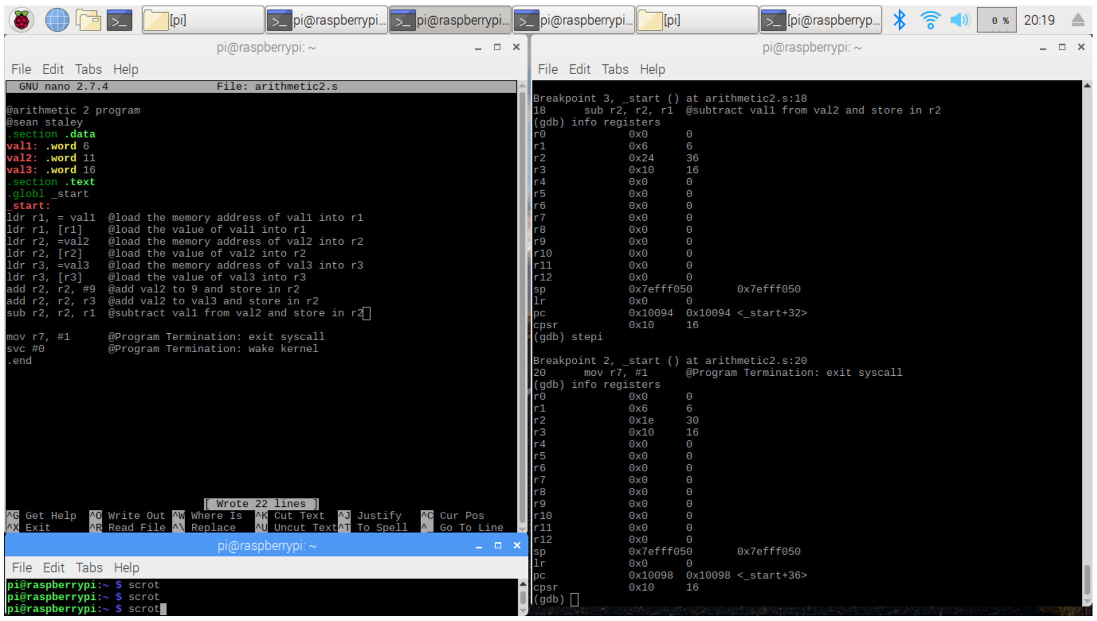


```
GNU nano 2.7.4                    File: arithmetic2.s

@arithmetic 2 program
@sean staley
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
ldr r1, = val1  @load the memory address of val1 into r1
ldr r1, [r1]    @load the value of val1 into r1
ldr r2, =val2   @load the memory address of val2 into r2
ldr r2, [r2]    @load the value of val2 into r2
ldr r3, =val3   @load the memory address of val3 into r3
ldr r3, [r3]    @load the value of val3 into r3
add r2, r2, #9  @add val2 to 9 and store in r2
add r2, r2, r3  @add val2 to val3 and store in r2
sub r2, r2, r1  @subtract val1 from val2 and store in r2

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

                          [ Wrote 22 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

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

```
lr             0x0      0
pc             0x10088  0x10088 <_start+20>
cpsr           0x10     16
(gdb) stepi
16        add r2, r2, #9  @add val2 to 9 and store in r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0xb      11
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x1008c  0x1008c <_start+24>
cpsr           0x10     16
(gdb) stepi
17        add r2, r2, r3  @add val2 to val3 and store in r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x14     20
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x10090  0x10090 <_start+28>
cpsr           0x10     16
(gdb) 
```

Below we see the final two results of our operation. Adding the value in r3 to the value in r2, which is now 16 + 20, results in 36. Or in hexadecimal as, 0x24. The final operation subtracts 6 from this resulting in 30, or 0x1e. Which coincides with our intuitive thoughts on what the result should be.



```
GNU nano 2.7.4                    File: arithmetic2.s

@arithmetic 2 program
@sean staley
.section .data
val1: .word 6
val2: .word 11
val3: .word 16
.section .text
.globl _start
_start:
ldr r1, = val1  @load the memory address of val1 into r1
ldr r1, [r1]    @load the value of val1 into r1
ldr r2, =val2   @load the memory address of val2 into r2
ldr r2, [r2]    @load the value of val2 into r2
ldr r3, =val3   @load the memory address of val3 into r3
ldr r3, [r3]    @load the value of val3 into r3
add r2, r2, #9  @add val2 to 9 and store in r2
add r2, r2, r3  @add val2 to val3 and store in r2
sub r2, r2, r1  @subtract val1 from val2 and store in r2

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

                          [ Wrote 22 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

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

```
Breakpoint 3, _start () at arithmetic2.s:18
18        sub r2, r2, r1  @subtract val1 from val2 and store in r2
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x24     36
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x10094  0x10094 <_start+32>
cpsr           0x10     16
(gdb) stepi

Breakpoint 2, _start () at arithmetic2.s:20
20        mov r7, #1      @Program Termination: exit syscall
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x1e     30
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x10098  0x10098 <_start+36>
cpsr           0x10     16
(gdb) 
```

Looking into the memory at the line of memory before the operations have been executed shows the unsigned word values in hexadecimal as follows:

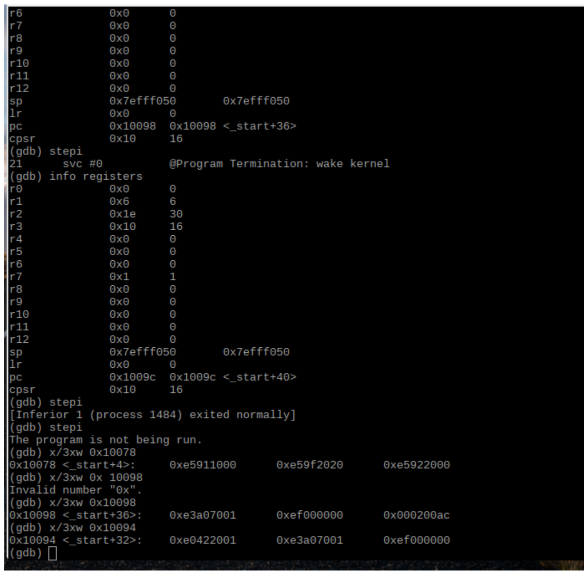
0xe501100, 0xe59f2020, 0xe592200.

The values just before the final command in line 18 are,

0xe0422001, 0xe3a7001, 0xef000000.

After the final command the word values in the memory are,

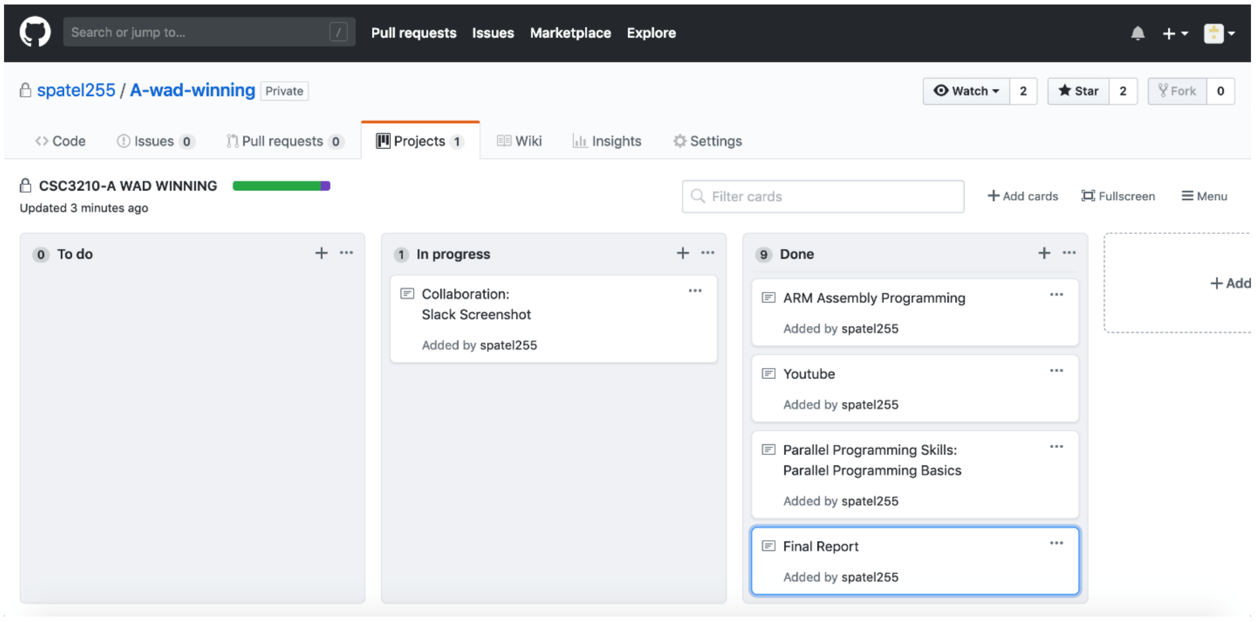
0xe3a07001, 0xe3af000000, 0xe5922000.



```
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x10098  0x10098 <_start+36>
cpsr           0x10     16
(gdb) stepi
21      svc #0          @Program Termination: wake kernel
(gdb) info registers
r0             0x0      0
r1             0x6      6
r2             0x1e     30
r3             0x10     16
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x1      1
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff050       0x7efff050
lr             0x0      0
pc             0x1009c  0x1009c <_start+40>
cpsr           0x10     16
(gdb) stepi
[Inferior 1 (process 1484) exited normally]
(gdb) stepi
The program is not being run.
(gdb) x/3xw 0x10078
0x10078 <_start+4>:     0xe5911000      0xe59f2020      0xe5922000
(gdb) x/3xw 0x 10098
Invalid number "0x".
(gdb) x/3xw 0x10098
0x10098 <_start+36>:    0xe3a07001      0xef000000      0x000200ac
(gdb) x/3xw 0x10094
0x10094 <_start+32>:    0xe0422001      0xe3a07001      0xef000000
(gdb) 
```

Appendix

GitHub:



CSC3210-A WAD WINNING
Updated 3 minutes ago

0 To do

1 In progress
- Collaboration: Slack Screenshot — Added by spatel255

9 Done
- ARM Assembly Programming — Added by spatel255
- Youtube — Added by spatel255
- Parallel Programming Skills: Parallel Programming Basics — Added by spatel255
- Final Report — Added by spatel255

YouTube Channel:

https://www.youtube.com/channel/UCVEvyRC9_1g_8306r68504g