Developing Soft and Parallel Programming Skills
Using Project – Based Learning

CSC 3210 Spring – 2020

8 – Bit

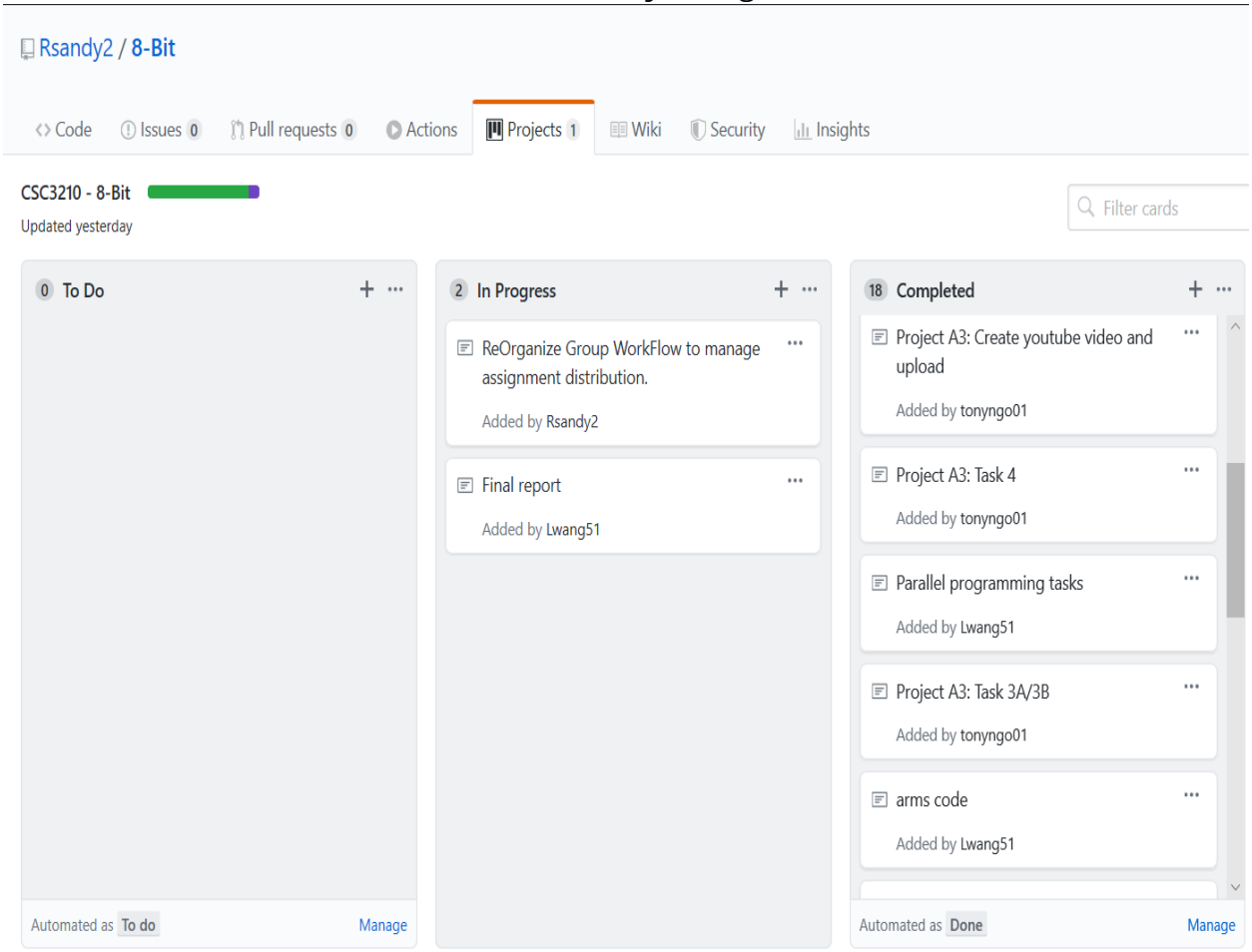
Landon Wang
Raejae Sandy
Tony Ngo

Planning and Scheduling

Work Breakdown Structure

Assignee Name	Email (studentID@student.gsu.edu)	Task	Duration (Hours)	Dependency	Due Date	Note
Akash Dansinghani	adansinghani1	WITHDRAWN FROM CLASS	--	--	--	--
Landon Wang	lwang51	Put together the final report	.5 hours on task, 6 hours on other project tasks	None	2/07/20	Make sure everything is together on the report
Raejae Sandy	Rsandy2	Help with GitHub	5 hours on task, 9 hours on other project tasks	None	2/07/20	Make sure everything is updated on GitHub
Shawn Martin	smartin93	WITHDRAWN FROM CLASS	--	--	--	--
Tony Ngo	tngo23	Made the YouTube Channel, recorded and edited the video	2 hours on task, 8 hours on other project tasks	None	2/07/20	Send the YouTube Link

GitHub Project Page



Parallel Programming Skills and Basics

Parallel Programming Skills and Basics

By: Landon Wang

Part 1: Parallel Programming Skills

- **(5p) Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)**
    - Task:
        A program (or something like a program) set of instructions that the processors execute. In a parallel program, there are multiple tasks being executed by multiple processors at a given time.
    - Pipelining:
        A type of parallel computing that break down tasks into steps that will be executed by different processors
    - Shared Memory:
        From a hardware perspective, it is a computer architecture where every processor has direct access to a common memory using the bus. From a programming perspective, it is a model where parallel tasks have the same "picture" memory with the ability to directly address and access the memory regardless of its physical location.
    - Communications:
        How parallel tasks communication with each other. Methods of communication includes network or a shared memory bus.
    - Synchronization:
        How parallel tasks cooperates with each other, mostly through a form of communication. This is mostly implemented by making a synchronization point in an application that tasks a task to wait for other task(s) to finish before proceeding. This can cause the application's wall clock execution time to increase.

- **(8p) Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.**
    - Single Instruction Single Data (SISD)
        - Serial computer
        - Only one instruction stream per one clock cycle
        - Only one input data stream per one clock cycle
        - Deterministic execution
        - Ancient computer
        - Examples includes older mainframe generations, minicomputers, workstations, and single processor PCs
    - Single Instruction Multiple Data (SIMD)
        - Parallel computer
        - All processors execute the same instruction at any given clock cycle
        - Each processor can execute different data elements
        - Most suitable for problems with high degree of regularity (like graphic/image processing)
        - Synchronous (lockstep) and deterministic execution

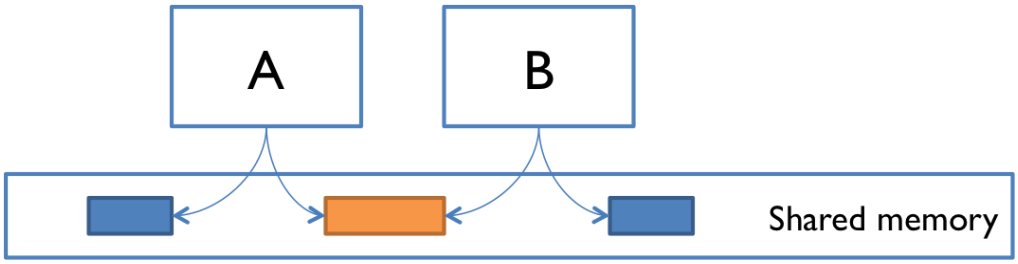
        - Two types: Processor Arrays and Vector Pipelines
        - Most modern computer (mostly those that has GPUs employ SIMD instructions and execution)
    - Multiple Instruction Single Data (MISD)
        - Parallel computer
        - Processors executes data independently by separate instruction streams
        - Single input data stream is fed to multiple processors
        - Computers rarely use this class type
        - Conceivable use may be multiple frequency filters working on a single signal stream, or multiple cryptography algorithms decrypting a code
    - Multiple Instruction Multiple Data (MIMD)
        - Parallel computer
        - Every processor can execute a different instruction stream
        - Every processor can take in different data streams
        - Most popular class of computers
        - Many MIMD also include SIMD execution sub-components

- **(7p) What are the Parallel Programming Models?**
    - Shared Memory (without threads)
    - Threads
    - Distributed Memory/message Passing
    - Data Parallel
    - Hybrid
    - Single Program Multiple Data (SPMD)
    - Multiple Program Multiple Data (MPMT)

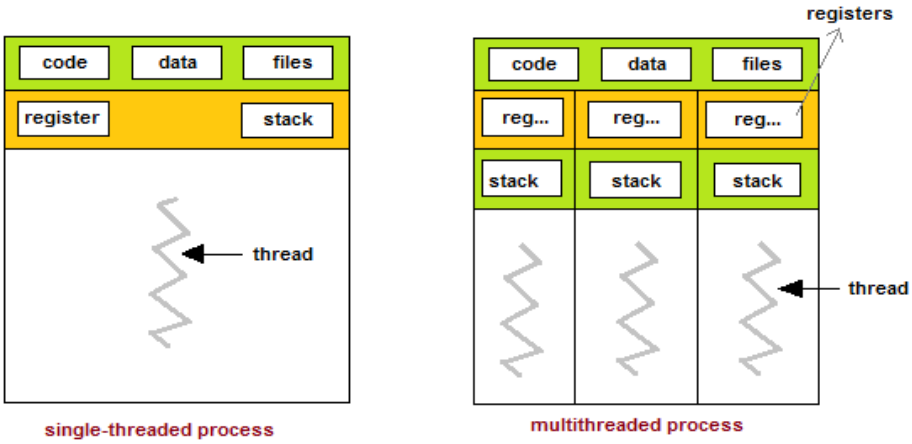
- **(12p) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**
    -

- **(10p) Compare Shared Memory Model with Threads Model? (in your own words and show pictures)**

    In the shared memory model, it does not use threads, and its tasks shares common memory address space that they read and write to asynchronously. To help prevent race conditions and dreadlocks and resolve contention, mechanisms like the locks/semaphores are used to control shared memory access. Shared memory model is the simplest parallel programming model. An advantage of using this model is that there is a lacking “ownership” of the data, which means that we do not to specify the communication of data between tasks since all processes see and have access to shared memory. However, using this model means that data locality will be harder to understand and control. The way how shared memory is implemented depends on the machines. On a stand-alone memory machine, the native operating systems, compilers, and/or hardware gives support for shared memory programming. On a distributed memory machine, the memory is made global by special hard and software.



In thread modeling, which is a type of shared memory programming, a "heavily weighted" task can have multiple "light weighted" concurrent threads. Each thread will have its own local data, but shares the entire resource of the program. This way, the program's resources does not need to be copied for each thread. The threads will also have benefits of global memory view since it shared the memory space of the program. The threads will also use synchronization constructs to communicate with each other. Implementing threads mostly comprise of a library of subroutines called within parallel source codes and a set



of compiler directives imbedded in the serial or parallel source code.

- **(5p) What is Parallel Programming? (in your own words)**

  A model of computing where program tasks can be broken down into threads that are then processed by two or more processors.

- **(5p) What is system on chip (SoC)? Does Raspberry PI use system on SoC?**

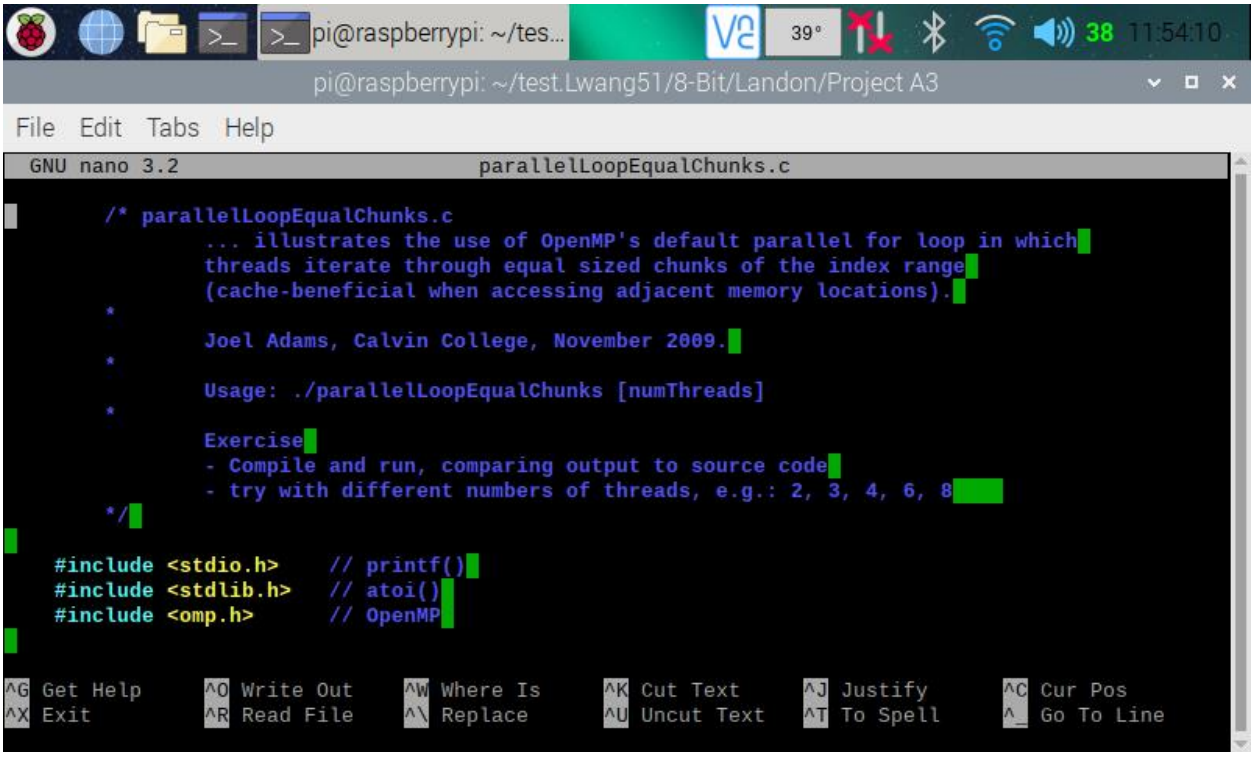
  SoC is a computer that has almost all of the components that a CPU would have integrated into a single silicon chip. An SoC usually has GPU, memory, USB controller, power management circuits, and wireless radios. Raspberry Pi computer would be an example of a System on Chip computer.

- **(5p) Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.**

  SoCs consumes less power, because it has a high level of integration, so it uses shorter wires. Furthermore, it will be cheaper to build, because far fewer physical chips are needed to.

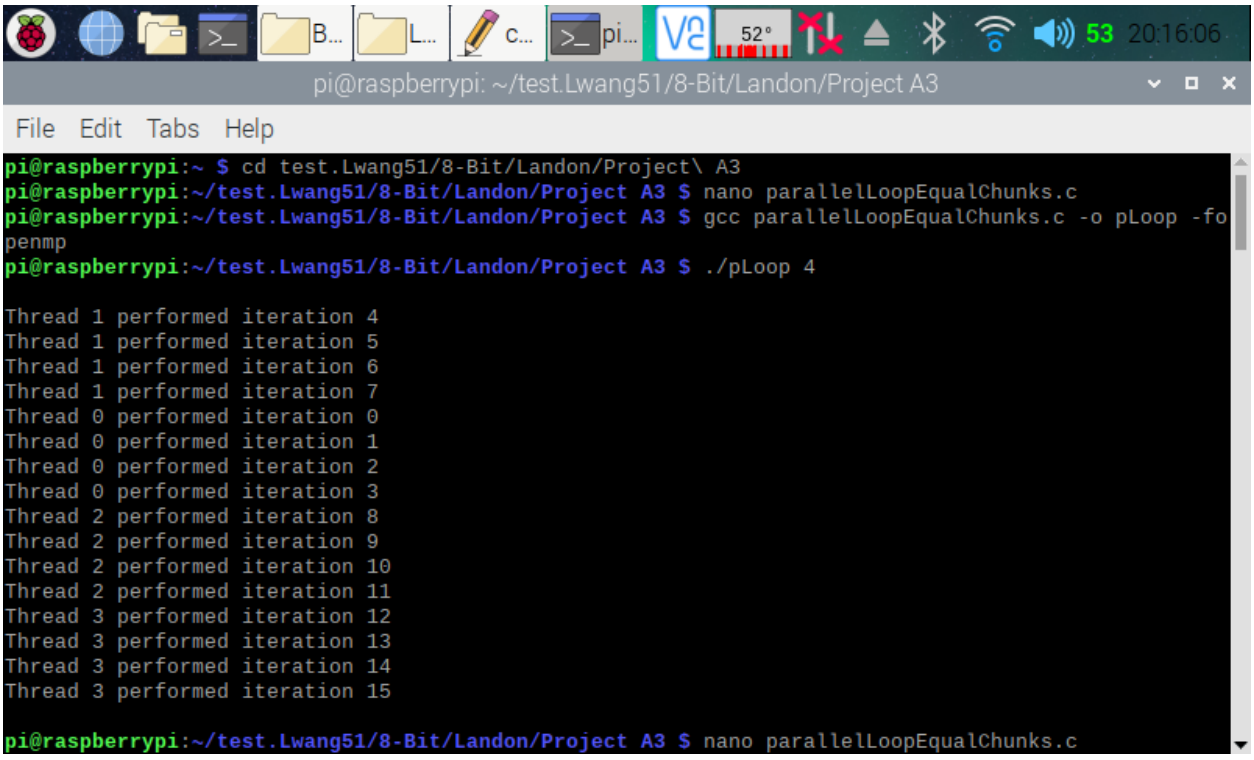
Part 2: Parallel Programming Basics

Parallel Loop Equal Chunks Program



```
int main(int argc, char** argv) {
const int REPS = 16;  //default value was 16

printf("\n");
if (argc > 1) {
   omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
  int id = omp_get_thread_num();
  printf("Thread %d performed iteration %d\n", id, i);
}

printf("\n");
return 0;
}

^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
```

Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.



```
pi@raspberrypi: ~/test.Lwang51/8-Bit/Landon/Project A3
File  Edit  Tabs  Help
pi@raspberrypi:~ $ cd test.Lwang51/8-Bit/Landon/Project\ A3
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc parallelLoopEqualChunks.c -o pLoop -fo
penmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 4

Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 3 performed iteration 15

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano parallelLoopEqualChunks.c
```

Here (in the screenshot above), I exited from nano and made an executable program by using **gcc parallelLoopEqualChunks.c -o pLoop -fopenmp** instruction (this created the pLoop executable program). I then ran pLoop with a command line argument indicating how many threads to fork (**./pLoop 4**). We can see from the output that the program has forked four thread to complete 16 iterations.

```
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc parallelLoopEqualChunks.c -o pLoop -fo
penmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 4

Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 3

Thread 0 performed iteration 0
```

Here (in the screenshot above), I went back into the code and changed the number of reps to 15 instead of 16. I then ran the program indicating 4 threads to fork. We can see that threads 1 to 2 processes four iterations while thread 3 only processed three iterations. This is what we get when we have iterations that's not evenly divisible by the number of threads.

```
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 1 performed iteration 8
Thread 1 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 2 performed iteration 12
Thread 2 performed iteration 13
Thread 2 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 15

Thread 1 performed iteration 1
Thread 9 performed iteration 9
Thread 2 performed iteration 2
Thread 13 performed iteration 13
Thread 3 performed iteration 3
Thread 0 performed iteration 0
```

Here (in the screenshot above), I ran the program indicating 3 threads to fork. We can see that threads 1 to 2 each processed five iterations. Each thread processed an even number of iterations, because the number of iterations (15) is divisible by 3.

```
Thread 0 performed iteration 0
Thread 12 performed iteration 12
Thread 4 performed iteration 4
Thread 5 performed iteration 5
Thread 7 performed iteration 7
Thread 8 performed iteration 8
Thread 10 performed iteration 10
Thread 11 performed iteration 11
Thread 14 performed iteration 14
Thread 6 performed iteration 6

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 16

Thread 4 performed iteration 4
Thread 14 performed iteration 14
Thread 3 performed iteration 3
Thread 7 performed iteration 7
Thread 2 performed iteration 2
Thread 8 performed iteration 8
Thread 1 performed iteration 1
Thread 0 performed iteration 0
Thread 9 performed iteration 9
Thread 6 performed iteration 6
Thread 10 performed iteration 10
```

```
Thread 10 performed iteration 10
Thread 11 performed iteration 11
Thread 12 performed iteration 12
Thread 13 performed iteration 13
Thread 5 performed iteration 5

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 106

Thread 1 performed iteration 1
Thread 2 performed iteration 2
Thread 4 performed iteration 4
Thread 0 performed iteration 0
Thread 7 performed iteration 7
Thread 3 performed iteration 3
Thread 6 performed iteration 6
Thread 11 performed iteration 11
Thread 5 performed iteration 5
Thread 8 performed iteration 8
Thread 13 performed iteration 13
Thread 10 performed iteration 10
Thread 9 performed iteration 9
Thread 12 performed iteration 12
Thread 14 performed iteration 14
```

Here (in the two screenshots above), I ran the program indicating 16 and 106 threads to fork. We can see that although 16 and 106 threads were forked, only 15 of them were used for the 15 iterations.

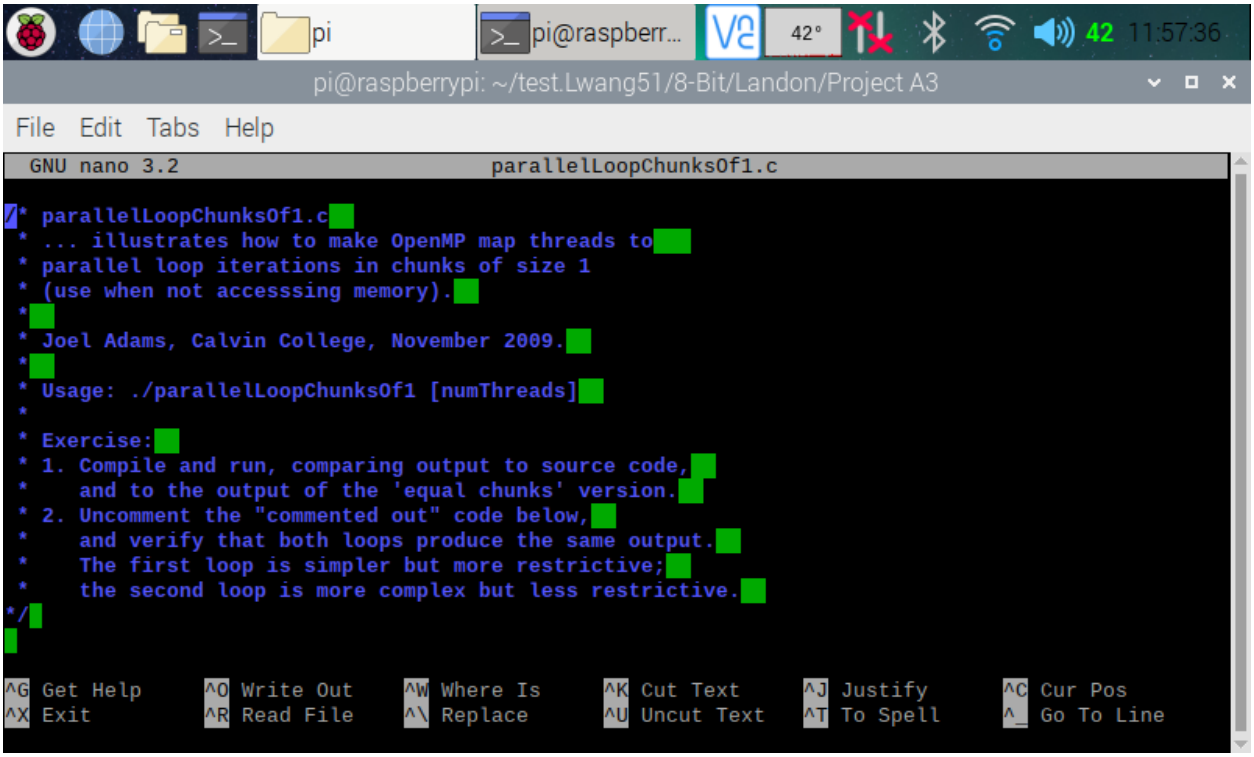
```
Thread 14 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop 2

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 0 performed iteration 4
Thread 0 performed iteration 5
Thread 0 performed iteration 6
Thread 0 performed iteration 7
Thread 1 performed iteration 8
Thread 1 performed iteration 9
Thread 1 performed iteration 10
Thread 1 performed iteration 11
Thread 1 performed iteration 12
Thread 1 performed iteration 13
Thread 1 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop

Thread 0 performed iteration 0
Thread 0 performed iteration 1
```

Here (in the screenshot above), I ran the program indicating 2 threads to fork. We can see that thread 0 processed eight iterations, and thread 1 processed seven iterations. The reason behind this is the same as when we ran the program with four threads to fork.

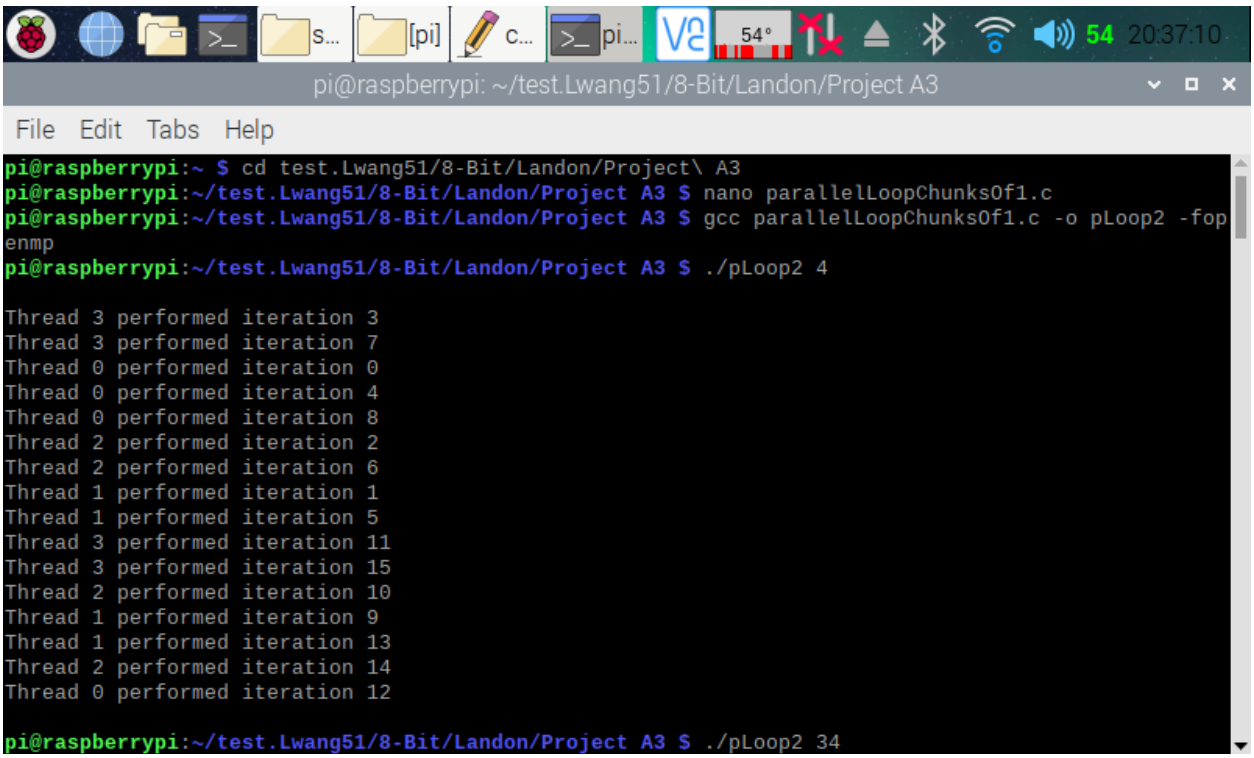
```
Thread 1 performed iteration 10
Thread 1 performed iteration 11
Thread 1 performed iteration 12
Thread 1 performed iteration 13
Thread 1 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop

Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 2 performed iteration 8
Thread 1 performed iteration 6
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 1 performed iteration 7
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 3 performed iteration 14
```

Here (in the screenshot above), I ran the program without indicating how many thread(s) to fork. The processor automatically goes with fours threads to fork since we have four processors available.

Parallel Loop Chunks of 1 Program



Here (in the three screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.



Here (in the screenshot above), I exited from nano and made an executable program by using **gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp** instruction (this created the pLoop2 executable program). I then ran pLoop2 with a command line argument indicating how many threads to fork (**./pLoop2 4**). We can see from the output that the program has forked four thread to complete 16 iterations. This type of threading is a little different from the previous program. It doles out one loop iteration to one thread, then the next iteration to the next thread and so on. As

a thread completes its iteration, it goes to the next one. The threads still preform the same amount of work, but just not in a consecutive iteration like how it did in the last program.

```
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 34

Thread 3 performed iteration 3
Thread 14 performed iteration 14
Thread 13 performed iteration 13
Thread 12 performed iteration 12
Thread 11 performed iteration 11
Thread 8 performed iteration 8
Thread 4 performed iteration 4
Thread 7 performed iteration 7
Thread 2 performed iteration 2
Thread 10 performed iteration 10
Thread 6 performed iteration 6
Thread 9 performed iteration 9
Thread 5 performed iteration 5
Thread 15 performed iteration 15
Thread 0 performed iteration 0
Thread 1 performed iteration 1

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 3

Thread 1 performed iteration 1
Thread 1 performed iteration 4
```

```
Thread 15 performed iteration 15
Thread 0 performed iteration 0
Thread 1 performed iteration 1

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 3

Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 1 performed iteration 13
Thread 0 performed iteration 0
Thread 0 performed iteration 3
Thread 0 performed iteration 6
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano parallelLoopChunksOf1.c
```

Here (in the screenshot above), I ran the program with 34 and 3 threads to fork. The concept of the amount of work each thread preformed is the same as the one explained in the previous program. How the threads preform is the same as if we ran it with 4 threads to fork.

```
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc parallelLoopChunksOf1.c -o pLoop2 -fop
enmp
parallelLoopChunksOf1.c: In function ‘main’:
parallelLoopChunksOf1.c:44:48: error: ‘a’ undeclared (first use in this function)
  printf("Thread %d performed iteration %d\n",  a. id, i);
                                                ^
parallelLoopChunksOf1.c:44:48: note: each undeclared identifier is reported only once for each fun
ction it appears in
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano parallelLoopChunksOf1.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc parallelLoopChunksOf1.c -o pLoop2 -fop
enmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 4

Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 3 performed iteration 3
Thread 1 performed iteration 9
Thread 0 performed iteration 8
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
```

```
Thread 2 performed iteration 10
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 0 performed iteration 12
Thread 1 performed iteration 13
Thread 2 performed iteration 14

---

Thread 3 performed iteration 3
Thread 3 performed iteration 7
Thread 3 performed iteration 11
Thread 3 performed iteration 15
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 0 performed iteration 0
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
```

Here (in the screenshot above), I went back into the code and removed the commented section of the code as stated in the instructions. I then fixed up an error and updated the executable file. Then, I ran the program with 4 threads to fork. Now, both for loops produces the same type of output, but like stated in the instruction, the second for loop is more complex but less restrictive. Part of this program output is on the next screenshot.

```
Thread 0 performed iteration 12
Thread 2 performed iteration 10
Thread 2 performed iteration 14

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 34

Thread 2 performed iteration 2
Thread 4 performed iteration 4
Thread 6 performed iteration 6
Thread 8 performed iteration 8
Thread 13 performed iteration 13
Thread 9 performed iteration 9
Thread 10 performed iteration 10
Thread 3 performed iteration 3
Thread 11 performed iteration 11
Thread 12 performed iteration 12
Thread 1 performed iteration 1
Thread 5 performed iteration 5
Thread 7 performed iteration 7
Thread 15 performed iteration 15
Thread 0 performed iteration 0
Thread 14 performed iteration 14

---
```

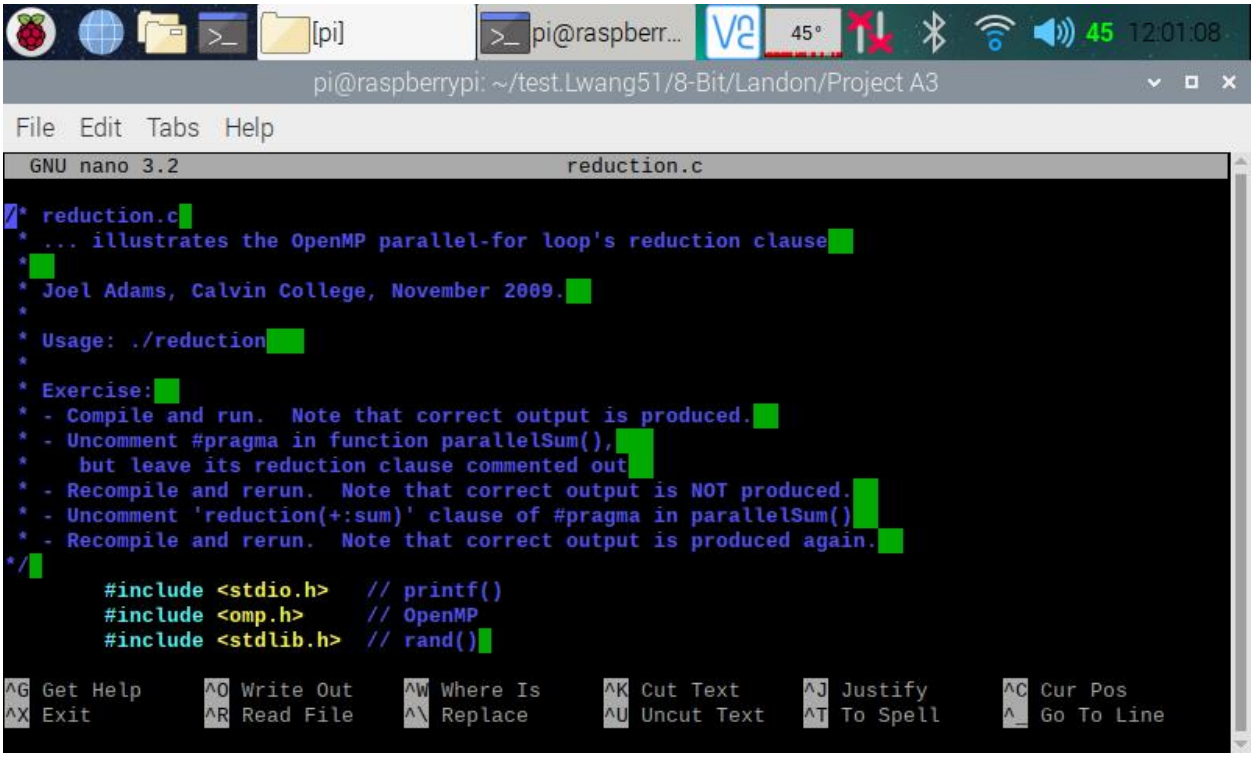
```
---

Thread 4 performed iteration 4
Thread 1 performed iteration 1
Thread 5 performed iteration 5
Thread 7 performed iteration 7
Thread 6 performed iteration 6
Thread 2 performed iteration 2
Thread 0 performed iteration 0
Thread 10 performed iteration 10
Thread 12 performed iteration 12
Thread 11 performed iteration 11
Thread 9 performed iteration 9
Thread 15 performed iteration 15
Thread 14 performed iteration 14
Thread 13 performed iteration 13
Thread 8 performed iteration 8
Thread 3 performed iteration 3

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./pLoop2 3

Thread 0 performed iteration 0
Thread 0 performed iteration 3
Thread 0 performed iteration 6
```

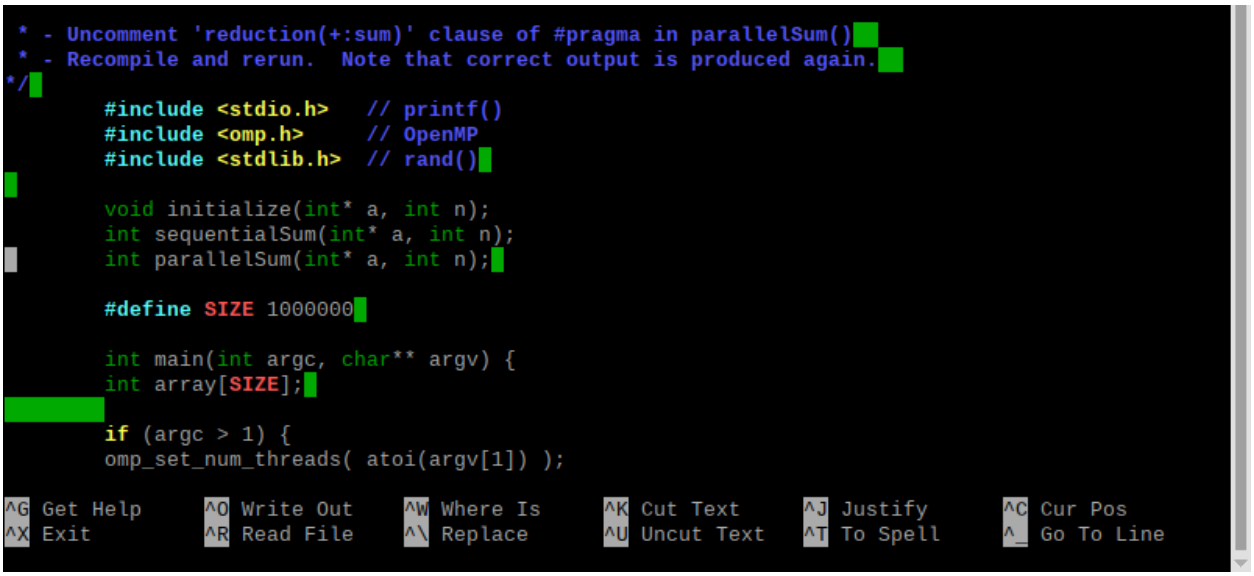
```
Thread 0 performed iteration 6
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14
Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 1 performed iteration 13
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15

---

Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 1 performed iteration 13
Thread 2 performed iteration 2
Thread 2 performed iteration 5
```

```
Thread 0 performed iteration 12
Thread 0 performed iteration 15

---

Thread 1 performed iteration 1
Thread 1 performed iteration 4
Thread 1 performed iteration 7
Thread 1 performed iteration 10
Thread 1 performed iteration 13
Thread 2 performed iteration 2
Thread 2 performed iteration 5
Thread 2 performed iteration 8
Thread 2 performed iteration 11
Thread 2 performed iteration 14
Thread 0 performed iteration 0
Thread 0 performed iteration 3
Thread 0 performed iteration 6
Thread 0 performed iteration 9
Thread 0 performed iteration 12
Thread 0 performed iteration 15

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $
```

Here (in the four screenshots above), I ran the program with 34 and 3 threads to fork, the output and the reason why each program threaded in a way is the same as in the previous examples.

Reduction Program





```
        omp_set_num_threads( atoi(argv[1]) );
        }

        initialize(array, SIZE);
        printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
        sequentialSum(array, SIZE),
        parallelSum(array, SIZE) );

        return 0;
        }

        /* fill array with random values */
        void initialize(int* a, int n) {
                int i;
                for (i = 0; i < n; i++) {
                a[i] = rand() % 1000;
                }
        }

^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
```

```
        }

        /* sum the array sequentially */
        int sequentialSum(int* a, int n) {
        int sum = 0;
        int i;
        for (i = 0; i < n; i++) {
                sum += a[i];
        }
        return sum;
        }

        /* sum the array using multiple threads */
        int parallelSum(int* a, int n) {
        int sum = 0;
        int i;
        #pragma omp parallel for reduction(+:sum)
        for (i = 0; i < n; i++) {

^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
```

```
        return sum;
        }

        /* sum the array using multiple threads */
        int parallelSum(int* a, int n) {
        int sum = 0;
        int i;
        #pragma omp parallel for reduction(+:sum)
        for (i = 0; i < n; i++) {
                sum += a[i];
        }
        return sum;
        }


^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
```

Here (in the five screenshots above), I copied and pasted the codes from the Parallel Programming Task A3 document and used the nano editor to create a program on my Raspberry PI.

```
pi@raspberrypi: ~/test.Lwang51/8-Bit/Landon/Project A3
File  Edit  Tabs  Help
reduction.c:63:7: note: each undeclared identifier is reported only once for each function it appe
ars in
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   188800115

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
^[[Api@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $
```

```
pi@raspberrypi:~ $ cd test.Lwang51/8-Bit/Landon/Project\ A3
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
reduction.c: In function 'parallelSum':
reduction.c:61:18: error: expected ';' before 'int'
  int sum = 0; 38. int i;
                 ^~~~
                 ;
reduction.c:63:7: error: 'i' undeclared (first use in this function)
  for (i = 0; i < n; i++) {
       ^
reduction.c:63:7: note: each undeclared identifier is reported only once for each function it appe
ars in
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ nano reduction.c
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ gcc reduction.c -o reduction -fopenmp
pi@raspberrypi:~/test.Lwang51/8-Bit/Landon/Project A3 $ ./reduction 4
```

Here (in the two screenshots above), I exited out of the nano, and made the executable of the program. After compiling and running the program for the first time, we can see that both of the output is the same. However, from looking at the code and reading the parallel programming task document, I concluded that the parallel sum from the first run is not actually calculated using parallel computing. I say this because the "// #pragma omp parallel for //reduction(+:sum)" line of code is commented out, and we need that to do parallel programming.

After going back and removing the first comment of the "// #pragma omp parallel for //reduction(+:sum)" code, I ran the program for the second time. After the program executed, we can see that the results from the sequential sum and the parallel sum does not match. I then went back and removed the second comment from the "#pragma omp parallel for //reduction(+:sum)" code. I then ran the program for the third time, and now, we can see that the sequential and a parallel sum both matched. Furthermore, the parallel sum is now calculated using parallel computing.

The reason why parallel for pragma without the reduction clause did not produce the correct result was because the variables used were not private, and that each thread did not have its own copy of the variable.

Parallel Programming Skills and Basics

By: Raejae Sandy

- (5p) Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)

Task – Instructions that are to be executed by processor.
Pipelining - This is dividing up task into different processing units.
Shared Memory – The location in where the processor has direct access to RAM. Also location where memory can be accessed regardless of existence.
Communications – Exchanged data between parallel task.
Synchronization – Coordination of task.

- (8p) Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.

SIMD – stands for Single Instruction, Multiple Data. It contains two types called Processor Arrays and Vector Pipelines. Single Instructions references when only one instruction is executed on a CPU cycle and Multiple is when each processing unit can be diversified.
SISD – Stands for Single Instruction Single Data. This is not parallel but rather means there is only one instruction that will execute during a CPU cycle.
MIMD – Stands for multiple instruction, multiple data. This utilizes both the benefits of different data sources and instructions. SuperComputers use this because they can operate complex functions in different allocated processors or through separate instruction streams.

- (7p) What are the Parallel Programming Models?

- Shared Memory (w/out threads)
- Threads – A type of shared memory programming. Usually used in libraries/compiler directives (i.e. OpenMP).
- Distributed Memory/Message Passing – Distributed memory means that memory is physically distributed across a network of machines through specialized hardware and software.
- Hybrid
- Data Parallel
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

- (12p) List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?

UMA stands for uniform memory access and is when machines have processors that have the same access time to memory as. It also supports parallel functionality where if one is updated for cache all are updated.
NUMA stands for Non-Uniform Memory Access. These are machines built on multiple SMP's that don't have the same access time as UMA. These are typically(mostly) slower.
OpenMP utilizes UMA due to the way it models the memory locations and the access time for each.

- (10p) Compare Shared Memory Model with Threads Model? (in your own words and

show pictures)

The main difference is within the thread model where you can control how much is allocated in a program; however, the shared memory model is an original concept where task are set in an isolated environment to be written / read from.

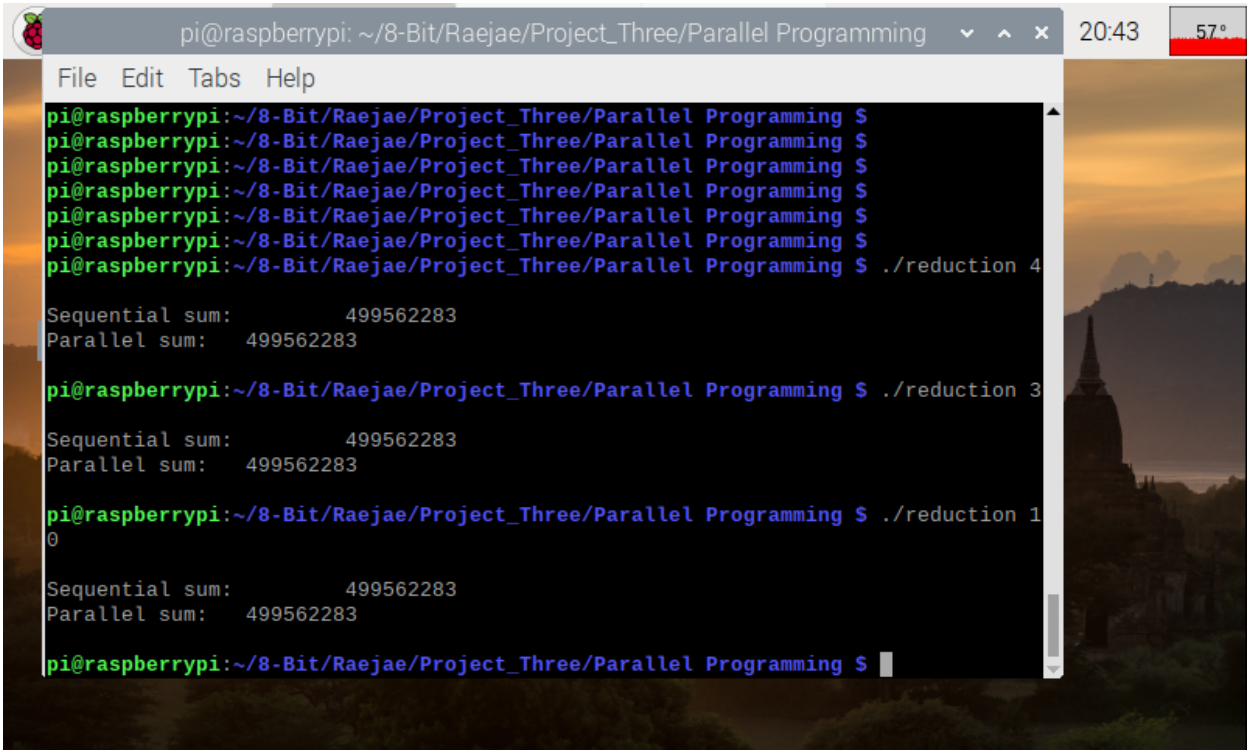
- (5p) What is Parallel Programming? (in your own words)

Parallel programming is the distribution of resources (threads / cores) (processing power) to carry out task and execute instructions in a case by case format.

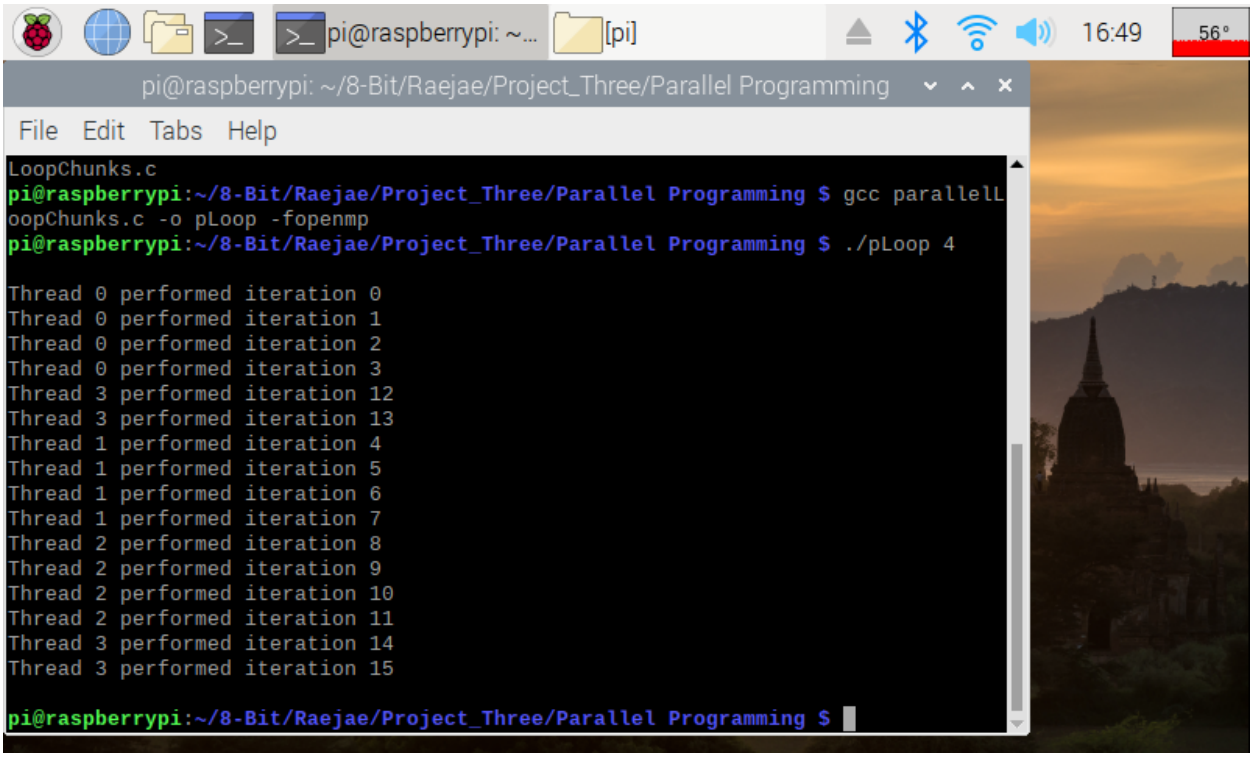
- (5p) What is system on chip (SoC)? Does Raspberry PI use system on SoC? - (5p) Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components.

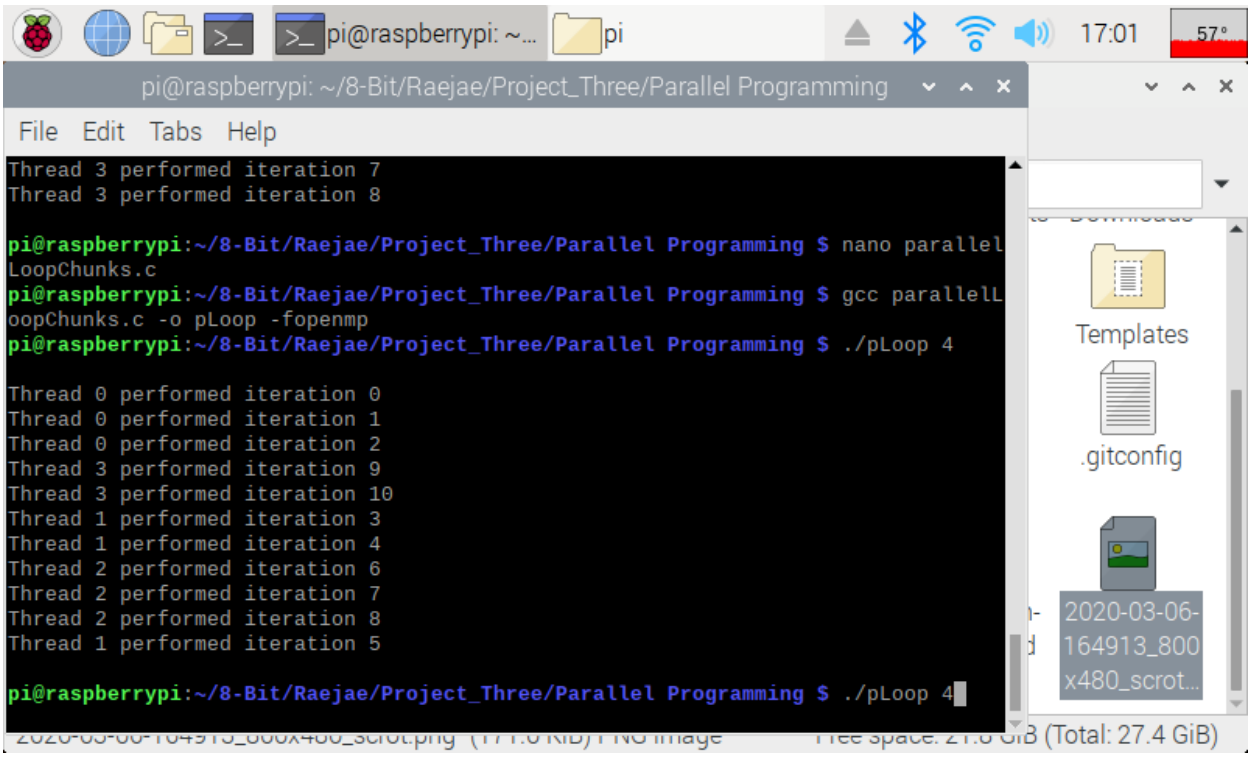
A Soc is similar to that of a shrink ray, because it really helps in minimization in the sense that you can manage output and optimize for spacing in devices. Since it emulates the components of a standard computer, you can get great functionality and open space for other components.

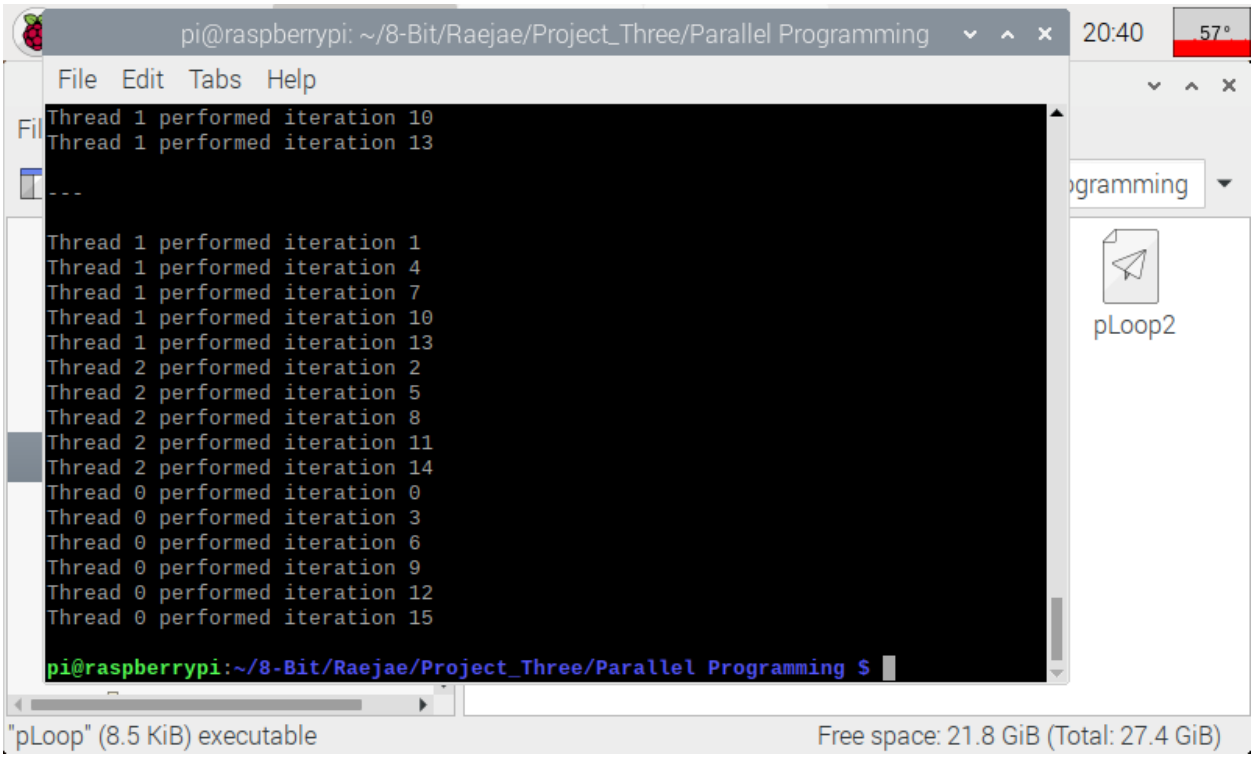
Parallel Programming Task



On the initial run of this code without removing the comments there was an issue of the values not matching. This is due to not initializing multi threading which results in a disconnect in values however after uncommenting the values match.


pi@raspberrypi: ~...
[pi]
16:49
56°
pi@raspberrypi: ~/8-Bit/Raejae/Project_Three/Parallel Programming
File Edit Tabs Help
LoopChunks.c
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ gcc parallelL
oopChunks.c -o pLoop -fopenmp
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ ./pLoop 4
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 0 performed iteration 3
Thread 3 performed iteration 12
Thread 3 performed iteration 13
Thread 1 performed iteration 4
Thread 1 performed iteration 5
Thread 1 performed iteration 6
Thread 1 performed iteration 7
Thread 2 performed iteration 8
Thread 2 performed iteration 9
Thread 2 performed iteration 10
Thread 2 performed iteration 11
Thread 3 performed iteration 14
Thread 3 performed iteration 15
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $


pi@raspberrypi: ~...
pi
17:01
57°
pi@raspberrypi: ~/8-Bit/Raejae/Project_Three/Parallel Programming
File Edit Tabs Help
Thread 3 performed iteration 7
Thread 3 performed iteration 8
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ nano parallel
LoopChunks.c
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ gcc parallelL
oopChunks.c -o pLoop -fopenmp
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ ./pLoop 4
Thread 0 performed iteration 0
Thread 0 performed iteration 1
Thread 0 performed iteration 2
Thread 3 performed iteration 9
Thread 3 performed iteration 10
Thread 1 performed iteration 3
Thread 1 performed iteration 4
Thread 2 performed iteration 6
Thread 2 performed iteration 7
Thread 2 performed iteration 8
Thread 1 performed iteration 5
pi@raspberrypi:~/8-Bit/Raejae/Project_Three/Parallel Programming $ ./pLoop 4
Templates
.gitconfig
2020-03-06-164913_800x480_scrot...
(Total: 27.4 GiB)



The fun aspect of running the codes for Loops is how the CPU will allocate resource based on its determined priority; moreover how the CPU will subdivide the processes to address the issue with running different iterations. Since they are subdivided each block will approach its handling different. The other cool thing to notice is when using the Loop2 is how the CPU manages optimization with certain threads having different precedence.

Parallel Programming Skills and Basics

By: Tony Ngo

CSC3210 Assignment A3 Task 3

Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization. (in your own words)

Task – A set of instructions that is executed by a processor.
Pipelining – Pipelining is taking a task and breaking it into separate steps performed by different processing units.
Shared Memory – Hardware-wise, shared memory shows where in the architecture where processors have direct access to physical memory (RAM). Programming-wise, it describes where parallel tasks can access the same logical memory regardless of where the memory actually exists.
Communications – Communication is considered exchanged data between parallel tasks.
Synchronization – Synchronization is the coordination of parallel tasks- usually implemented by creating a "synchronization point".

Classify parallel computers based on Flynn's taxonomy. Briefly describe every one of them.

SISD (Single Instruction, Single Data): This is the oldest type of computer that does not have parallel instruction. Single instruction means that there is only one instruction executing during a single CPU Clock Cycle. Single data means that there's only one stream of data being used as input during a single CPU Clock Cycle.

SIMD (Single Instruction, Multiple Data): This is a parallel computer that has two types (Processor Arrays & Vector Pipelines); they are specialized for issue that need intense graphics/image processing. Computers that have GPUs have SIMD instruction & execution units. Single instruction means that there is only one instruction executing during each CPU cycle. Multiple Data means that each processing unit can operate on a different data unit.

MISD (Multiple Instruction, Single Data): This is a parallel computer that is very uncommon, they are used in very specific cases (i.e. decoding cryptography algorithms). Multiple Instructions means that each processing unit operates independently via separate instruction streams. Single Data means that single data streams are fed into multiple processing units.

MIMD (Multiple Instruction, Multiple Data): This is a parallel computer that integrate both multiple instructions and multiple data; the only computers that can do this are mainly supercomputers. Usually, MIMD computers utilize SIMD components. Multiple instruction means that each processing unit operates independently via separate instruction streams. Multiple Data means that each processing unit can operate on a different data unit.

What are the Parallel Programming Models?

Parallel Programming models:

- Shared Memory (w/out threads)
- Threads – A type of shared memory programming. Usually used in libraries/compiler directives (i.e. OpenMP).
- Distributed Memory/Message Passing – Distributed memory means that memory is physically distributed across a network of machines through specialized hardware and software.
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?

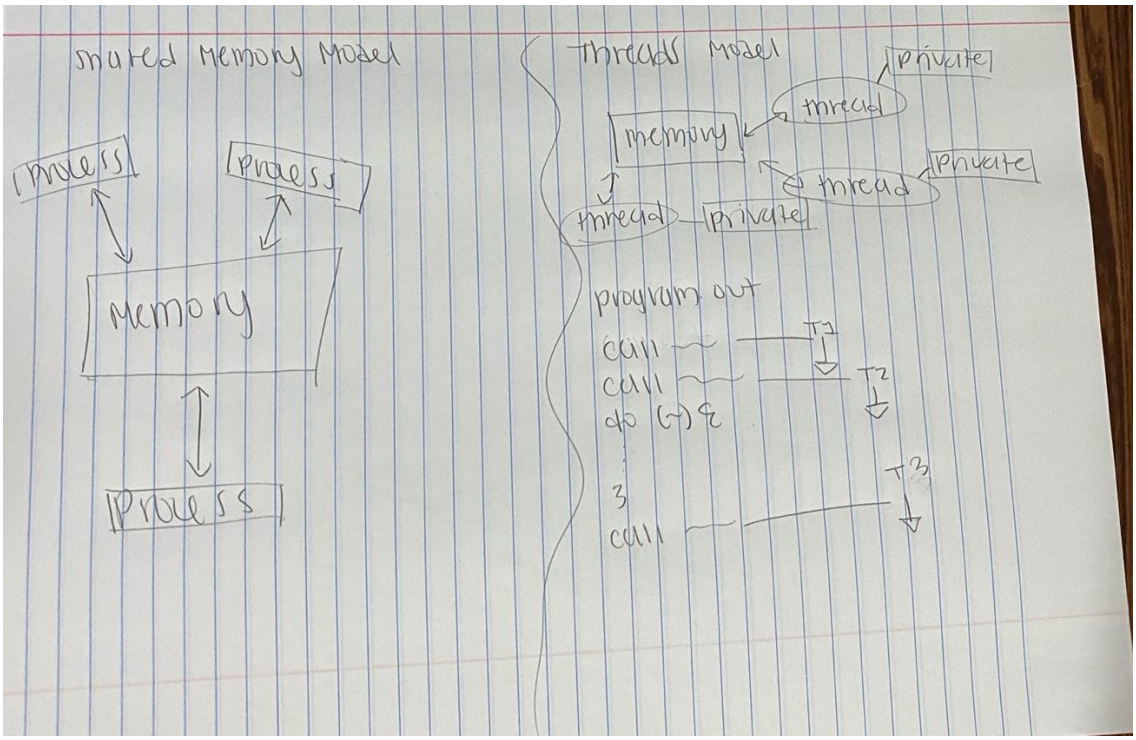
UMA (Uniform Memory Access) – Uniform Memory Access machines have identical processors which have equal access and access times to memory. It includes CC-UMAA which means that if one cache is updated, the rest are also updated. They are commonly represented by Symmetric Multiprocessor machines (SMP).

NUMA (Non-Uniform Memory Access) – Non-Uniform Memory Access machines consist of two or more SMPs and do not have equal access time to the memory, which means the processes are usually slower.

OpenMP uses UMA because it has equal access times to all memory locations, which is a characteristic of UMA.

Compare Shared Memory Model with Threads Model? (in your own words and show pictures)

The shared memory model is the most basic form of parallel programming which shows that processes/tasks are written into a singular space where they can read/write from. In the thread model, you can choose how much processing power is put into a single part of a program, so it lets you focus more threads on heavier tasks.



What is Parallel Programming? (in your own words)

Parallel programming is the utilization of multiple computer threads or cores to execute a single instruction, thus making it faster and more efficient to solve complex issues.

What is system on chip (SoC)? Does Raspberry PI use system on SoC?

A SoC (System on Chip) integrates a CPU, GPU, memory, and IO devices together onto a single chip. The Raspberry PI does use a System on Chip.

Explain what the advantages are of having a System on a Chip rather than separate CPU, GPU and RAM components

A System on Chip system allows you to decrease the size of a product significantly, which is extremely beneficial in smartphone development since you have more space to put a battery; also

it utilizes less power since there is shorter wiring that is associated with it. However, there is no room for expandability, since the SoC is integrated directly onto the chip.

Parallel Programming Task

CSC3210 Assignment 3 Task 3B – Tony Ngo

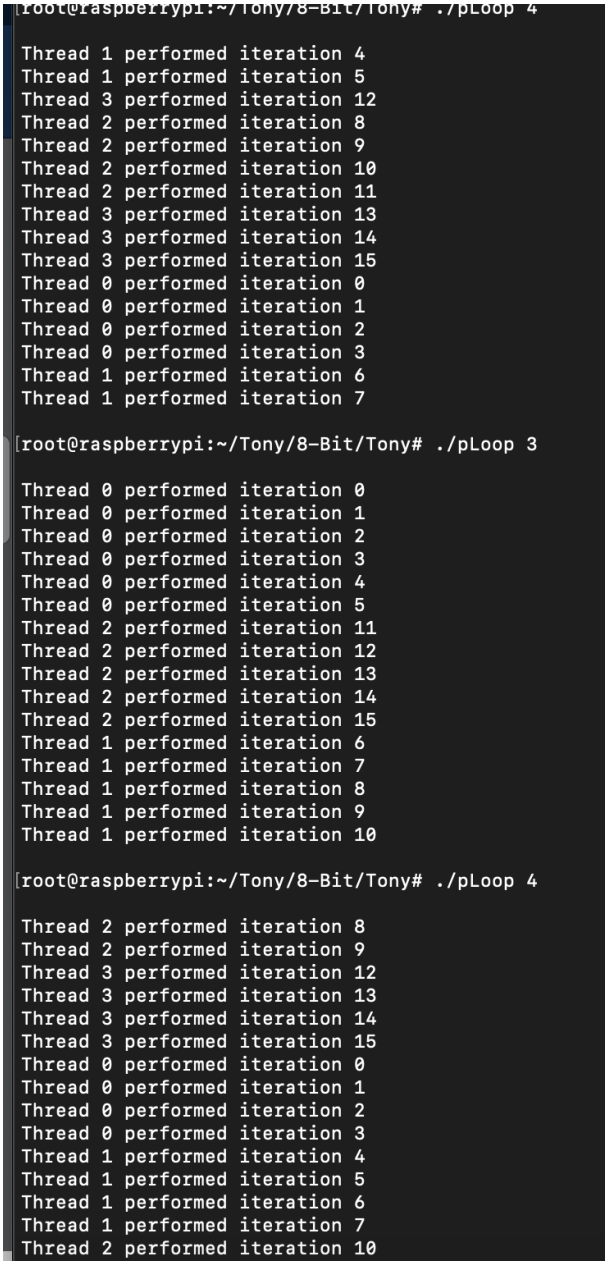
2.2 – parallelLoopEqualChunks.c

```
  GNU nano 3.2                        parallelLoopEqualChunks.c

/* parallelLoopEqualChunks.c
... illustrates the use of OpenMP's default parallel for loop in which
threads iterate through equal sized chunks of the index range
(cache-beneficial when accessing adjacent memory locations).
*
Joel Adams, Calvin College, November 2009.
*
Usage: ./parallelLoopEqualChunks [numThreads]
*
Exercise
- Compile and run, comparing output to source code
- try with different numbers of threads, e.g.: 2, 3, 4, 6, 8
*/
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for
for (int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}

printf("\n");
return 0;
}
```

This is the code for parallelLoopEqualChunks.c shown in the nano text editor.

```
root@raspberrypi:~/Tony/8-Bit/Tony# nano parallelLoopEqualChunks.c
root@raspberrypi:~/Tony/8-Bit/Tony# ls
arithmetic1    arithmetic2.o            first                      second    spmd2.c
arithmetic1.o  arithmetic2.s            first.o                    second.o
arithmetic1.s  CSC3210_A1T3.docx        first.s                    second.s
arithmetic2    CSC3210A1T4_TonyNgo.docx  parallelLoopEqualChunks.c  spmd2
root@raspberrypi:~/Tony/8-Bit/Tony# gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
parallelLoopEqualChunks.c:14:1: error: expected identifier or '(' before numeric constant
 1. #include <stdio.h> // printf()
 ^~
parallelLoopEqualChunks.c:14:4: error: stray '#' in program
 1. #include <stdio.h> // printf()
    ^
parallelLoopEqualChunks.c:15:4: error: stray '#' in program
 2. #include <stdlib.h> // atoi()
    ^
parallelLoopEqualChunks.c:16:4: error: stray '#' in program
 3. #include <omp.h> // OpenMP
    ^
parallelLoopEqualChunks.c:26:5: error: stray '#' in program
 10. #pragma omp parallel for
     ^
root@raspberrypi:~/Tony/8-Bit/Tony# nano parallelLoopEqualChunks.c
root@raspberrypi:~/Tony/8-Bit/Tony# gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
root@raspberrypi:~/Tony/8-Bit/Tony# ./pLoop 4
```

This is my creating and creating the executable of the parallelLoopEqualChunks.c program. When I originally compiled it, I forgot to remove the numbers beforehand from the copy and paste which did not allow me to compile it, creating those errors. Once I removed it I was able to create the executable using “gcc parallelLoopEqualChunks.c -o pLoop -fopenmp”



This is the execution of pLoop with multiple threads. The threads themselves did not run in order, but they were executed in the correct iteration. This is because the threads will perform in the order that the CPU believes is most efficient.

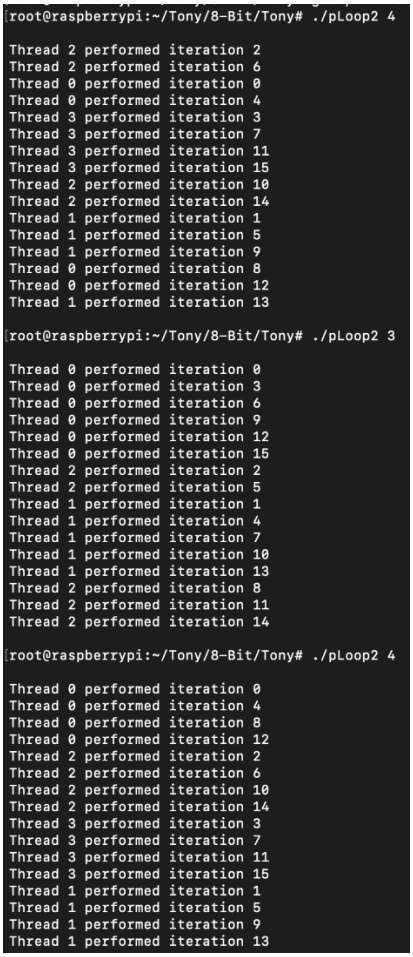
2.2 – parallelLoopChunksOf1.c

```c
/* parallelLoopChunksOf1.c
* ... illustrates how to make OpenMP map threads to
* parallel loop iterations in chunks of size 1
* (use when not accesssing memory).
*
* Joel Adams, Calvin College, November 2009.
*
* Usage: ./parallelLoopChunksOf1 [numThreads]
*
* Exercise:
* 1. Compile and run, comparing output to source code,
* and to the output of the 'equal chunks' version.
* 2. Uncomment the "commented out" code below,
* and verify that both loops produce the same output.
* The first loop is simpler but more restrictive;
* the second loop is more complex but less restrictive.
*/

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
const int REPS = 16;

printf("\n");
if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
}

#pragma omp parallel for schedule(static,1)
for (int i = 0; i < REPS; i++) {
        int id = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n", id, i);
}
/*
printf("\n---\n\n");
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
for (int i = id; i < REPS; i += numThreads) {
printf("Thread %d performed iteration %d\n", a. id, i);
}
}
*/
printf("\n");
return 0;
}
```

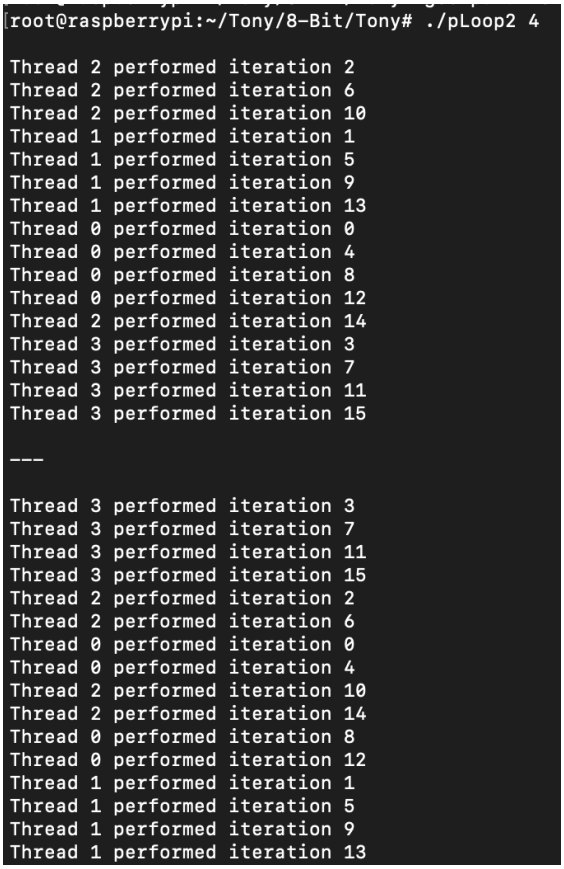
This is the code for parallelLoopChunksOf1.c shown in the nano text editor.

```
[root@raspberrypi:~/Tony/8-Bit/Tony# nano parallelLoopChunksOf1.c
[root@raspberrypi:~/Tony/8-Bit/Tony# gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp
[root@raspberrypi:~/Tony/8-Bit/Tony# ./pLoop2 4
```

This is the creation of the parallelLoopChunksOf1.c file. I created the “pLoop2” executable using “gcc parallelLoopChunksOf1.c -o pLoop2 -fopenmp”.



This is the execution of pLoop2 with different values. The way that splitting it into chunks works is that the CPU will use each thread as needed for each operation, which is seen in this code. Because for each iteration, a different thread is being used in order (i.e. with four cores being used, Thread 0 performed iteration 0 -> Thread 1 performed iteration 1 -> Thread 2 performed iteration 2 -> Thread 3 performed iteration 3 …).



This is the pLoop2 output with the uncommented variation. The output is the same for both.

4.2 - reduction

```c
/* reduction.c
* ... illustrates the OpenMP parallel-for loop's reduction clause
*
* Joel Adams, Calvin College, November 2009.
*
* Usage: ./reduction
*
* Exercise:
* - Compile and run. Note that correct output is produced.
* - Uncomment #pragma in function parallelSum(),
* but leave its reduction clause commented out
* - Recompile and rerun. Note that correct output is NOT produced.
* - Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()
* - Recompile and rerun. Note that correct output is produced again.
*/
#include <stdio.h> // printf()
#include <omp.h> // OpenMP
#include <stdlib.h> // rand()

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
int array[SIZE];
if (argc > 1) {
       omp_set_num_threads( atoi(argv[1]) );
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE) );

return 0;
}

/* fill array with random values */
void initialize(int* a, int n) {
int i;
for (i = 0; i < n; i++) {
       a[i] = rand() % 1000;
}
}

/* sum the array sequentially */
int sequentialSum(int* a, int n) {
int sum = 0;
```

```c
int main(int argc, char** argv) {
int array[SIZE];
if (argc > 1) {
       omp_set_num_threads( atoi(argv[1]) );
}

initialize(array, SIZE);
printf("\nSequential sum: \t%d\nParallel sum: \t%d\n\n",
sequentialSum(array, SIZE),
parallelSum(array, SIZE) );

return 0;
}

/* fill array with random values */
void initialize(int* a, int n) {
int i;
for (i = 0; i < n; i++) {
       a[i] = rand() % 1000;
}
}

/* sum the array sequentially */
int sequentialSum(int* a, int n) {
int sum = 0;
int i;
for (i = 0; i < n; i++) {
       sum += a[i];
}
return sum;
}

/* sum the array using multiple threads */
int parallelSum(int* a, int n) {
int sum = 0;
int i;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++) {
       sum += a[i];
}
return sum;
}
```

Here is the code for reduction.c shown in the nano text editor

```
root@raspberrypi:~/Tony/8-Bit/Tony# nano reduction.c
root@raspberrypi:~/Tony/8-Bit/Tony# gcc reduction.c -o reduction -fopenmp
root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 4
```

This is the creation of the parallelLoopChunksOf1.c file. I created the “reduction” executable using “gcc reduction.c -o reduction -fopenmp”.

4.3

```
root@raspberrypi:~/Tony/8-Bit/Tony# nano reduction.c
root@raspberrypi:~/Tony/8-Bit/Tony# gcc reduction.c -o reduction -fopenmp
root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 4

Sequential sum:         499562283
Parallel sum:   150209378

root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 3

Sequential sum:         499562283
Parallel sum:   211361203

root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 2

Sequential sum:         499562283
Parallel sum:   297854298
```

This is the code ran with the first // comment removed at line 39. The parallel sum is NOT the same as sequential sum. I noticed that the number would be bigger with the less amount of CPU cores that were used in the calculation.

```
root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 4

Sequential sum:         499562283
Parallel sum:   499562283

root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 3

Sequential sum:         499562283
Parallel sum:   499562283

root@raspberrypi:~/Tony/8-Bit/Tony# ./reduction 2

Sequential sum:         499562283
Parallel sum:   499562283
```

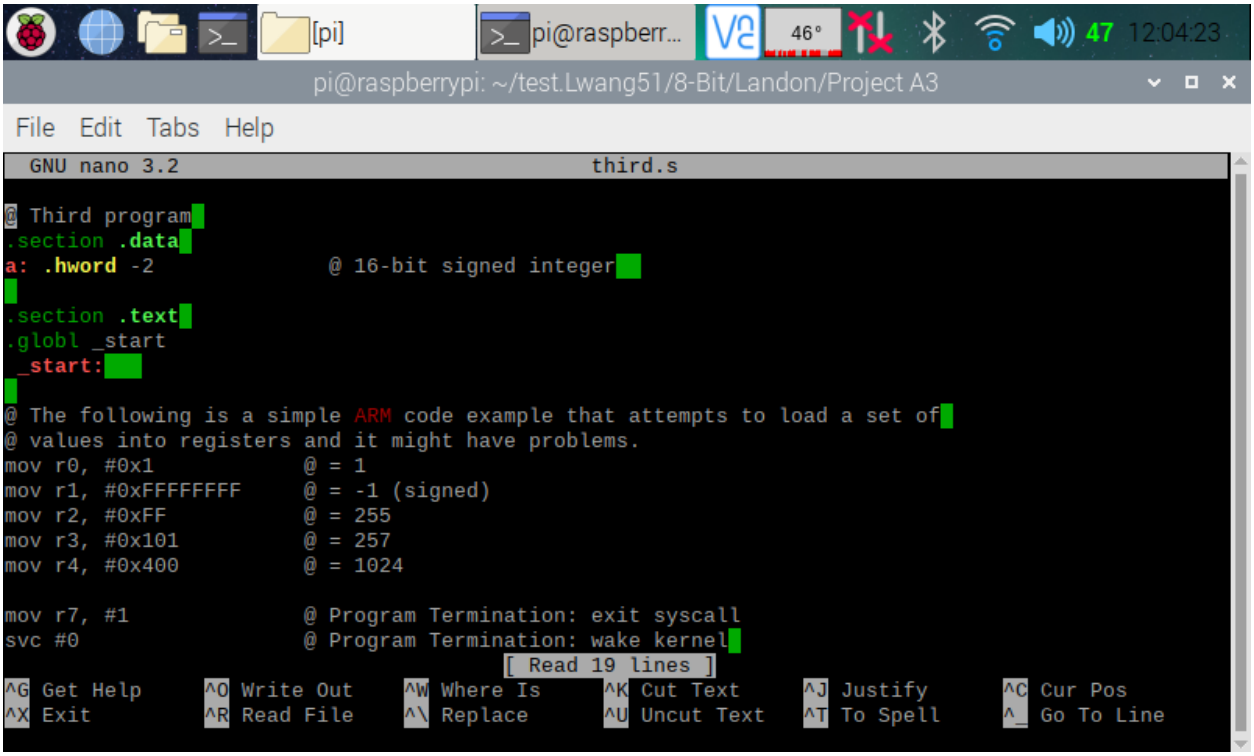
This is the output ran with the second // comment removed also at line 39. The parallel sum is the same as the sequential sum in this case. I believe that this is because the initialization of the pragma openMP needs to be utilized or multithreading would not be enabled, which was seen in the first execution with just the first // comment removed.

ARM Assembly Programming

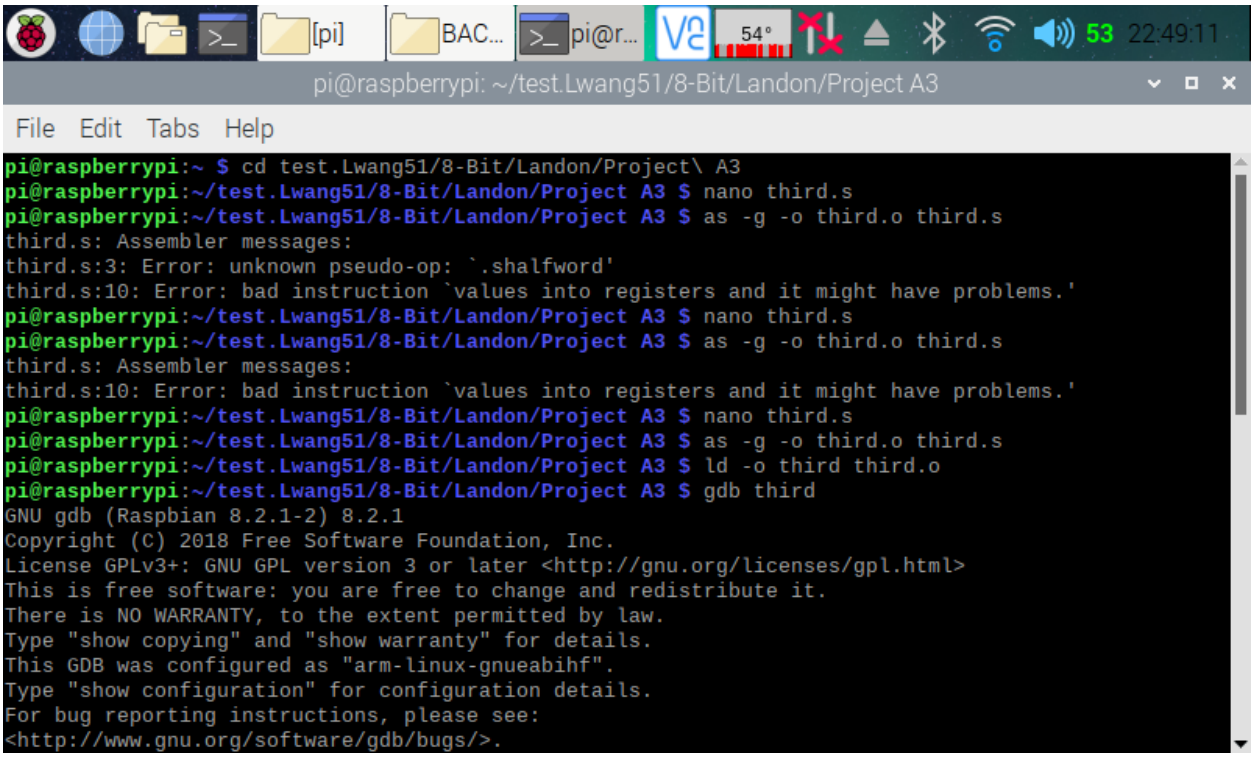
ARM Assembly Programming

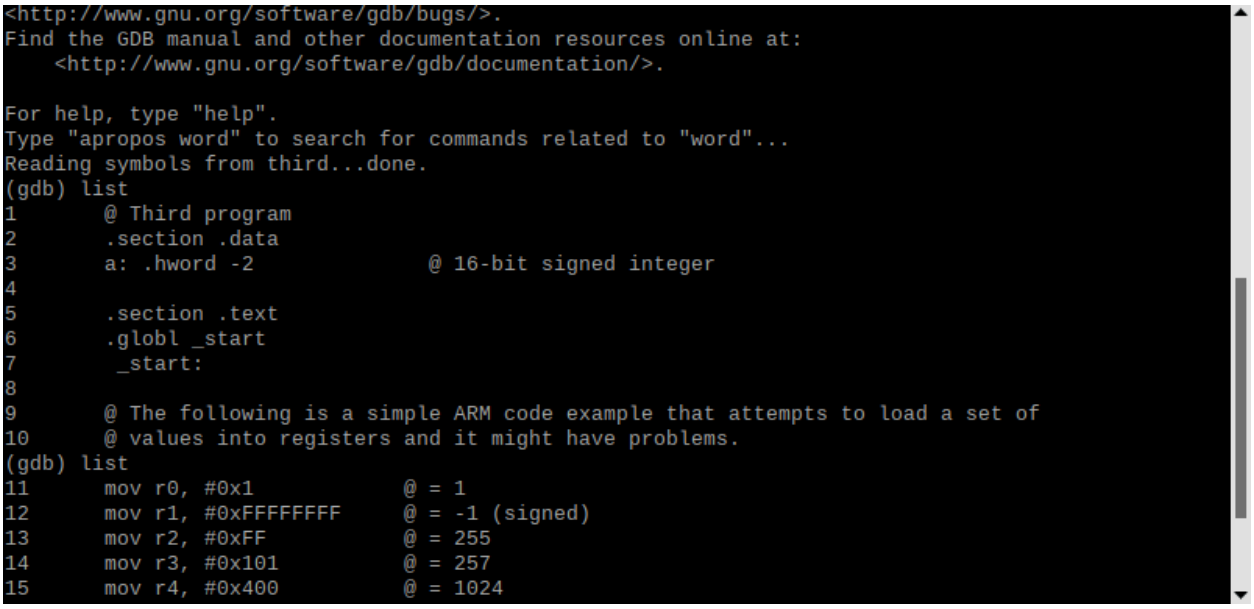
By: Landon Wang

Part 1: Third Program

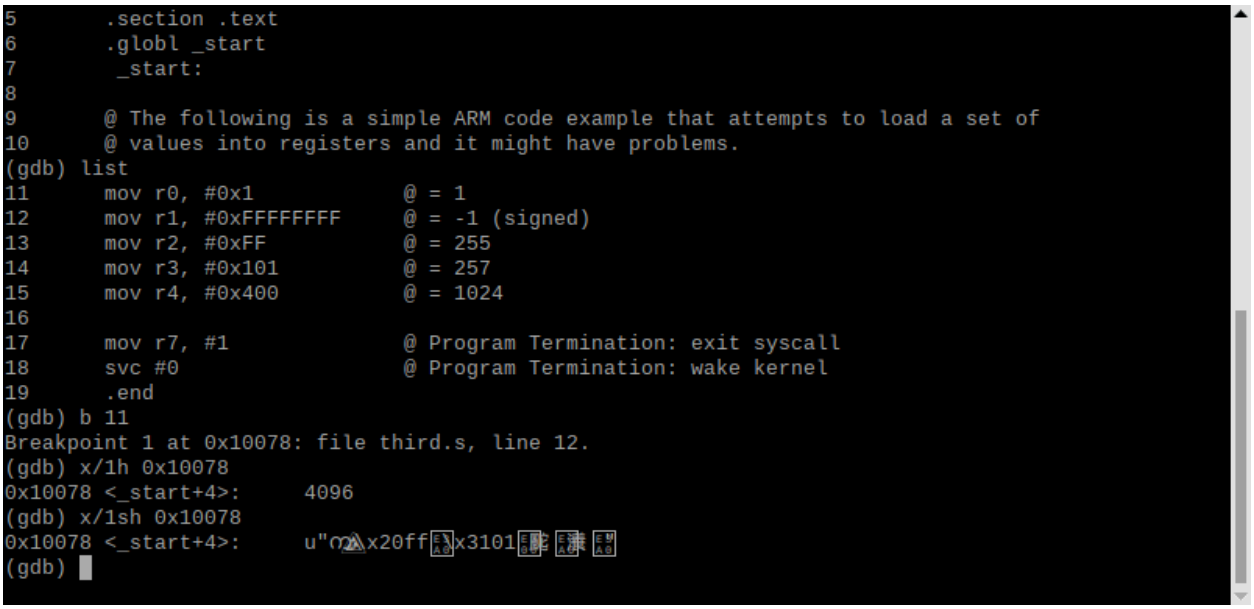


Here (in the two screenshots above), I copied and pasted the Third program from the ARM Assembly Programming A3 document and used the nano editor to create a program on my Raspberry PI.



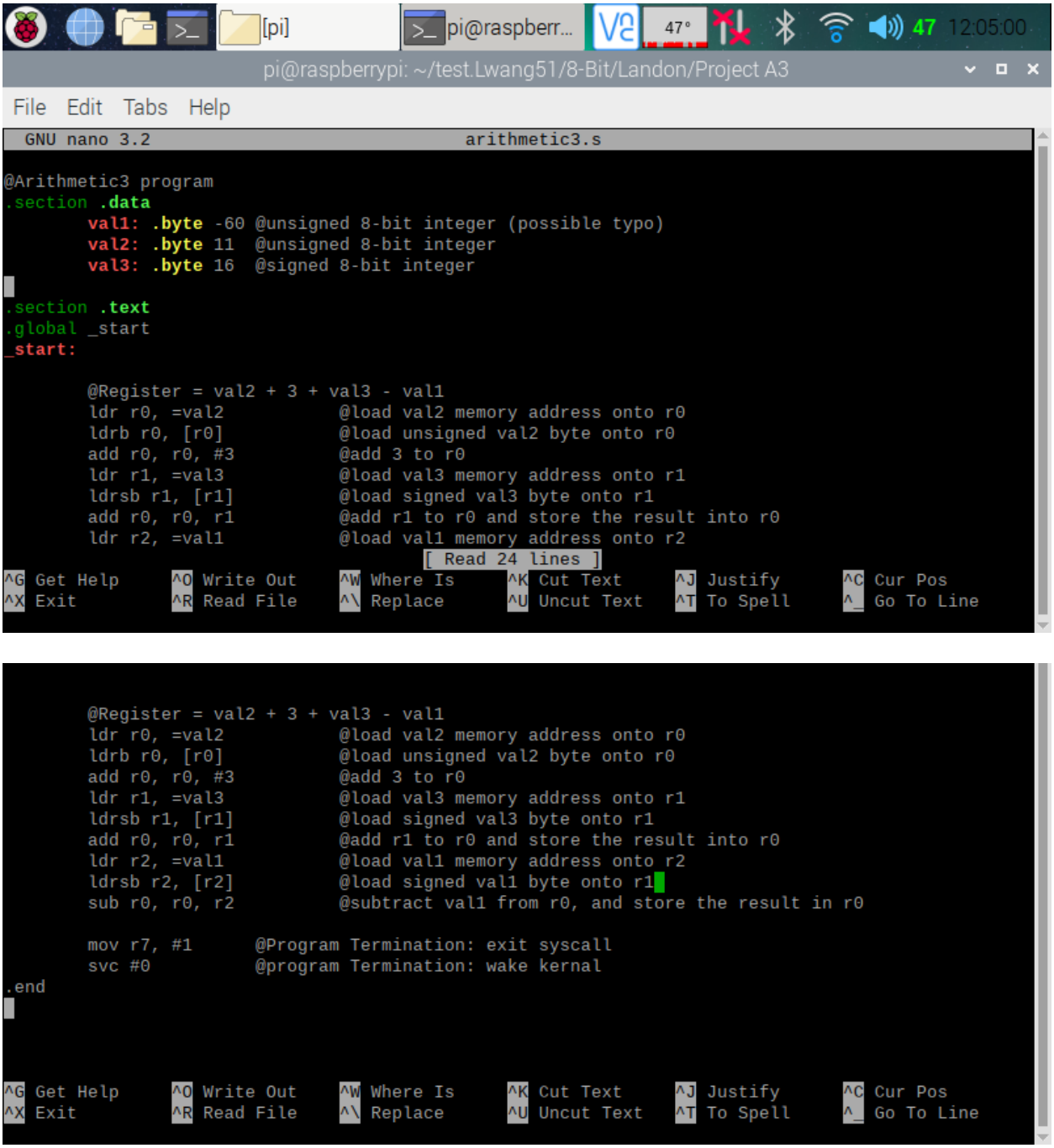


Here (in the two screenshots above), I exited out from the nano editor and assembled and linked the Third program. I got two error while attempt to assemble the program. My first error was on line three of my program. I got this error, because there was no such thing as .shalfword in ARM. Since ARM supports operation on different datatypes, there was no need to add in the ‘s’ in halfword. Along with this, to declare a halfword in ARM, we simply just type hword. To fix this error, we simply just change “a: .shalfword -2” to “a: .hword -2”. The second error was cause by a missing comment sign. I just simply added in a ‘@’ to resolve the error. After assembling and linking the program, I entered the GDB debugger.

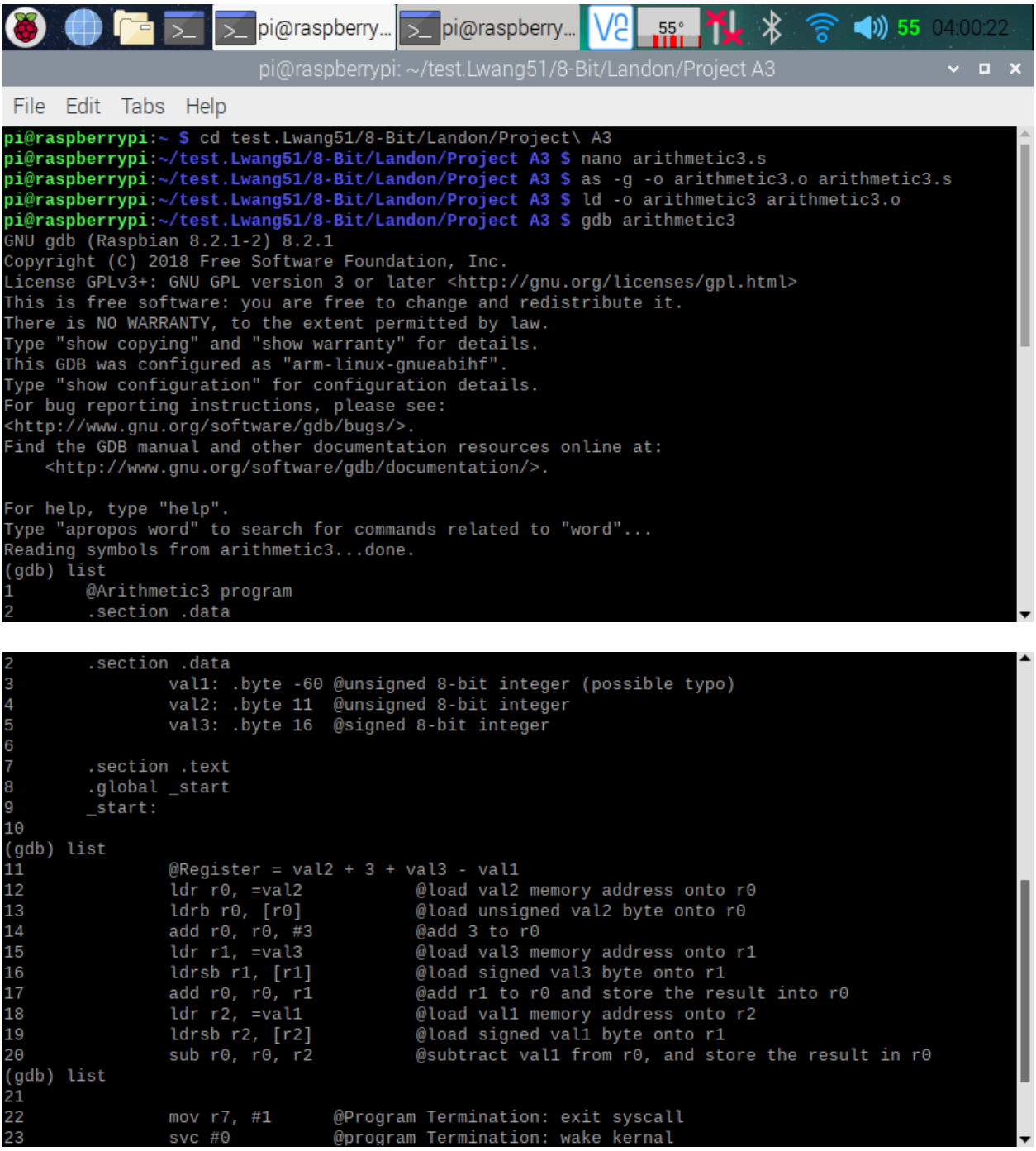


Here (in the screenshot above), I displayed my code, then placed a breakpoint at line 11 (which the debugger automatically moved it to line 12 since line 11 was an invalid point). I then ran the program and entered the memory using the memory address given to me when I set the breakpoint. When I used “x/1h 0x10078” to access the memory, I got a number (4096), and when I used “x/1sh 0x10078” to access the memory, I got many weird symbols.

Part 2: Arithmetic3 Program



Here (in the two screenshots above), is my code for the Arithmetic3 program.



Here (in the two screenshots above), I exited the nano editor, then assembled and linked my program. I then entered the debugger. In the debugger I used **List** to display my codes for reference.

```
23              svc #0          @program Termination: wake kernal
24      .end
(gdb) b 13
Breakpoint 1 at 0x10078: file arithmetic3.s, line 13.
(gdb) run
Starting program: /home/pi/test.Lwang51/8-Bit/Landon/Project A3/arithmetic3

Breakpoint 1, _start () at arithmetic3.s:13
13              ldrb r0, [r0]           @load unsigned val2 byte onto r0
(gdb) info register
r0             0x200ad            131245
r1             0x0                0
r2             0x0                0
r3             0x0                0
r4             0x0                0
r5             0x0                0
r6             0x0                0
r7             0x0                0
r8             0x0                0
r9             0x0                0
r10            0x0                0
r11            0x0                0
r12            0x0                0
sp             0x7efff330         0x7efff330
```

Here (in the screenshot above), I set a breakpoint at line 13, then ran the program. I then opened the register information, and we can see that a memory address has been placed into r0.

```
sp             0x7efff330         0x7efff330
lr             0x0                0
pc             0x10078            0x10078 <_start+4>
cpsr           0x10               16
fpscr          0x0                0
(gdb) x/1db 0x200ad
0x200ad:        11
(gdb) stepi
14              add r0, r0, #3          @add 3 to r0
(gdb) info register
r0             0xb                11
r1             0x0                0
r2             0x0                0
r3             0x0                0
r4             0x0                0
r5             0x0                0
r6             0x0                0
r7             0x0                0
r8             0x0                0
r9             0x0                0
r10            0x0                0
r11            0x0                0
r12            0x0                0
sp             0x7efff330         0x7efff330
```

Here (in the screenshot above), I used the memory address placed into r0 to check if it is pointing at the right memory place. The memory that I pulled up shows 11, which is what I was expecting. I then stepped over to the next line so that line 13 can execute. I then pulled up the register, and we can see that number 11 has been added to r0, which is what I wanted.

```
sp             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x1007c             0x1007c <_start+8>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
15              ldr r1, =val3           @load val3 memory address onto r1
(gdb) info register
r0             0xe                 14
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x10080             0x10080 <_start+12>
```

Here (in the screenshot above), I stepped over to the next line so that line 14 can execute. I then opened the register information, and we can see that r0 now holds 14 (11 + 3), and that is what I was expecting.

```
pc             0x10080             0x10080 <_start+12>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
16              ldrsb r1, [r1]          @load signed val3 byte onto r1
(gdb) info register
r0             0xe                 14
r1             0x200ae             131246
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x10084             0x10084 <_start+16>
cpsr           0x10                16
fpscr          0x0                 0
```

Here (in the screenshot above), I stepped over to the next line so that line 15 can execute. I then opened the register information, and we can see that a memory address has been placed into r1.

```
fpscr          0x0                 0
(gdb) x/1db 0x200ae
0x200ae:        16
(gdb) stepi
17              add r0, r0, r1          @add r1 to r0 and store the result into r0
(gdb) info register
r0             0xe                 14
r1             0x10                16
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x10088             0x10088 <_start+20>
cpsr           0x10                16
fpscr          0x0                 0
```

Here (in the screenshot above), I pulled up the memory, and we can see that it is pointing to 16 as expected. I then stepped over to the next line so that line 16 can execute. I then opened the register information, and we can see that 16 has been loaded into r1.

```
fpscr          0x0                 0
(gdb) stepi
18              ldr r2, =val1           @load val1 memory address onto r2
(gdb) info register
r0             0x1e                30
r1             0x10                16
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x1008c             0x1008c <_start+24>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
19              ldrsb r2, [r2]          @load signed val1 byte onto r1
```

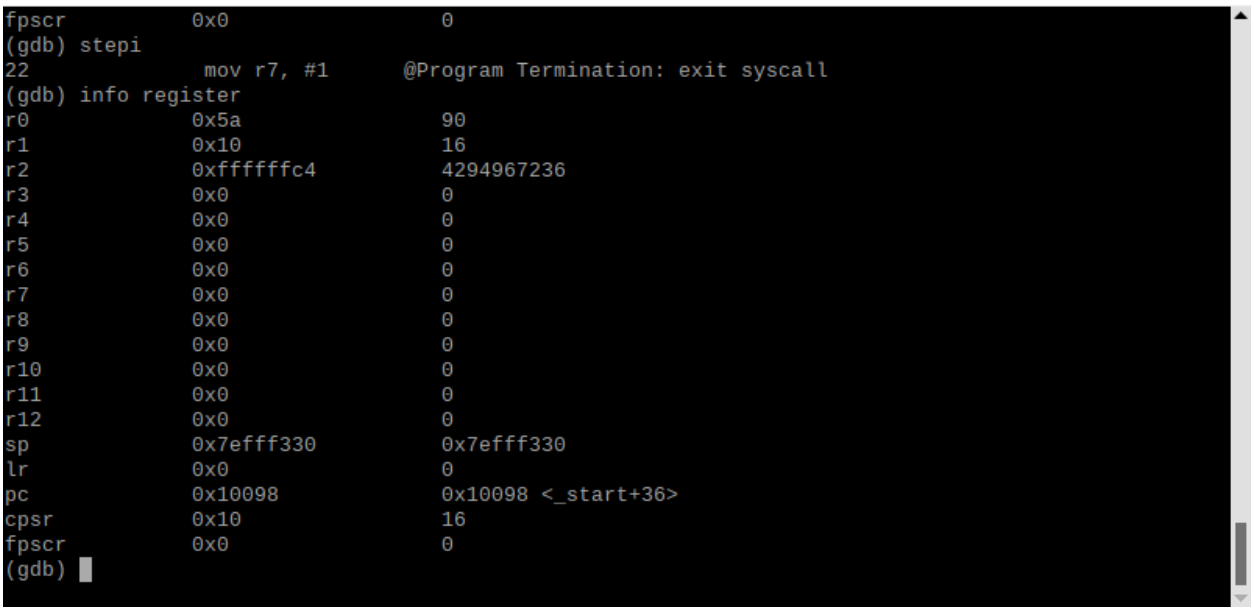
Here (in the screenshot above), I stepped over to the next line so that line 17 can execute. I then opened the register information, and we can see that r0 now stores a 30 (14 + 16), which was expected. I then stepped over to the next line so that line 18 can execute.

```
19              ldrsb r2, [r2]          @load signed val1 byte onto r1
(gdb) info register
r0             0x1e                30
r1             0x10                16
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             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x10090             0x10090 <_start+28>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) x/1db 0x200ac
0x200ac:        -60
(gdb) stepi
20              sub r0, r0, r2          @subtract val1 from r0, and store the result in r0
```

Here (in the screenshot above), I opened the register information, and we can see that a memory address has been loaded onto r2. I then used that memory address to pull up the memory, and we can see that it is pointing at -60 as expected. I then stepped over to the next line so that line 19 can execute.

```
20              sub r0, r0, r2          @subtract val1 from r0, and store the result in r0
(gdb) info register
r0             0x1e                30
r1             0x10                16
r2             0xffffffc4          4294967236
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             0x7efff330          0x7efff330
lr             0x0                 0
pc             0x10094             0x10094 <_start+32>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) stepi
22              mov r7, #1      @Program Termination: exit syscall
(gdb) info register
r0             0x5a                90
```

Here (in the screenshot above), I pulled up the register information, and we can see that FFFFFFC4 or 4294967236 (two's complement of -60) is loaded onto r2 as expected. I then stepped over to the next line so that line 20 can execute.

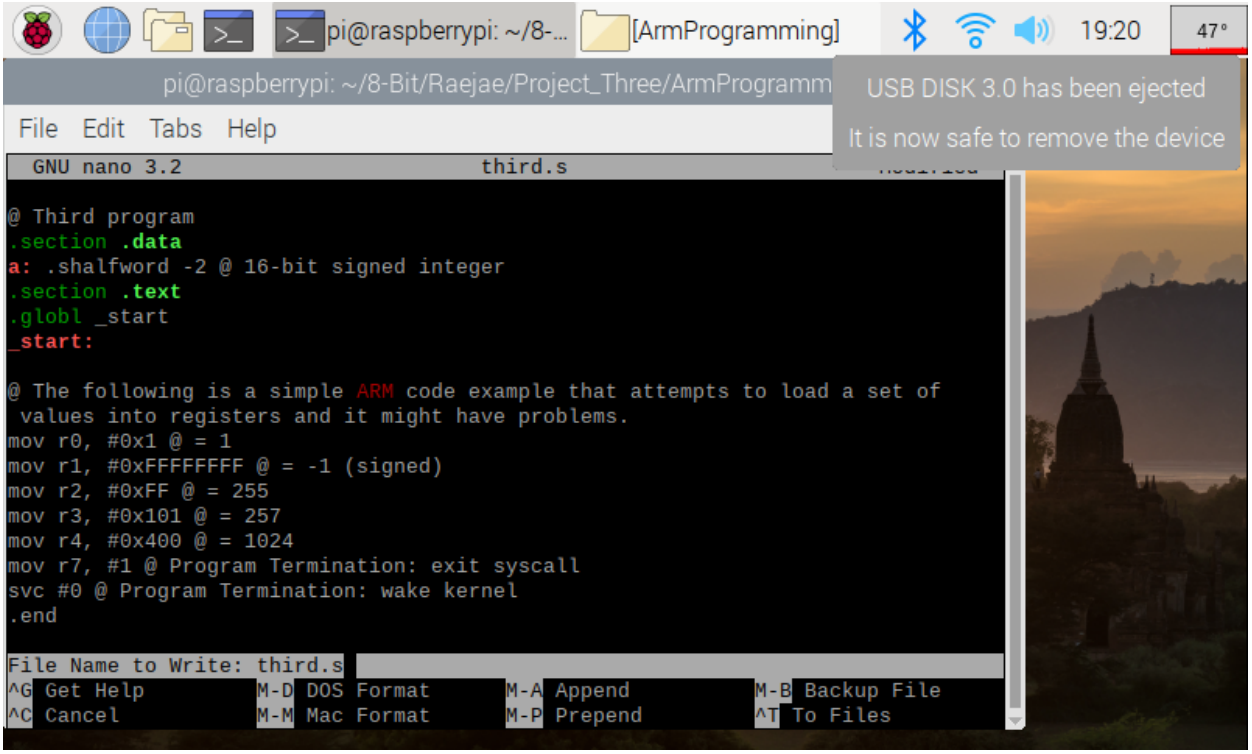


Here (in the screenshot above), I opened the register information, and we can see that 90 (30 - -60) is now stored in r0, which is what I wanted. I now know that I have the right program.

ARM Assembly Programming

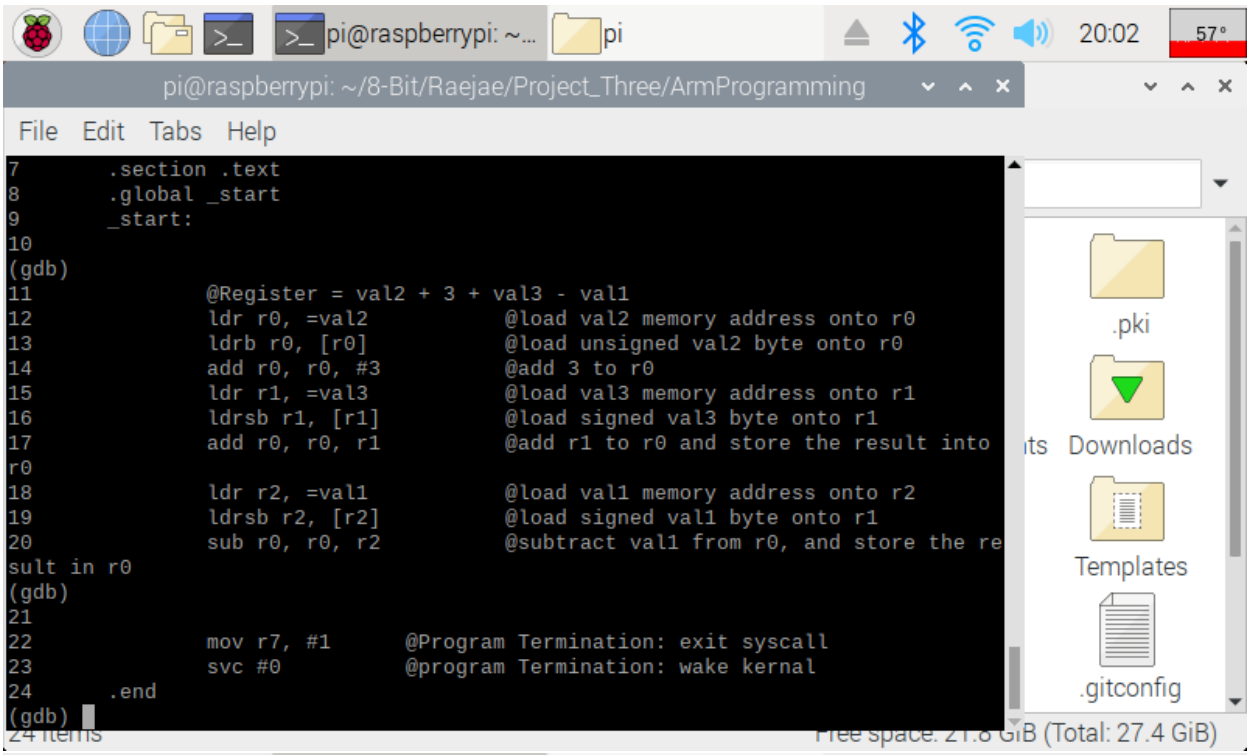
By: Raejae Sandy

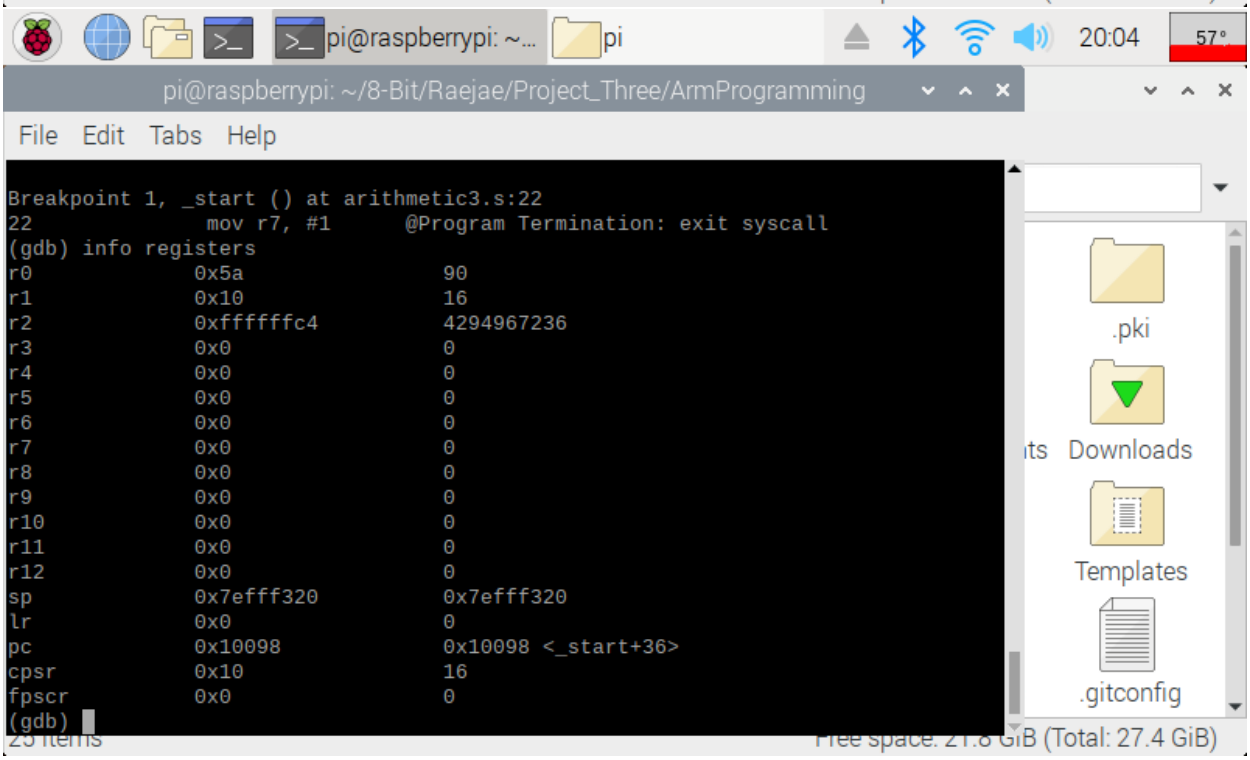
For the third ARM programming assignment there were new concepts applied. To start off here is the code written in Nano.



While programming I did come across a bug which was along the lines of Pseudo-op: '.shalfword'. This was fixed after swapping to .hword.
I found out that this is because in ARM we automatically assign types to signed.

This code was fun to write and explore because you get to see the effect of working with signed numbers. For this one after opening, I used info registers to find the memory address. At B 7 the address is 0x10078. Using “x/1xh 0x10078” returned 0x1000 however on the call of x/1sh we get a new string format which is because of the signed word.





This is the code after running Aritmethic Three and the interesting concept is when we use signed values instead to get our result of 90. If we pay attention we see we get ff5a which gives a weird number; however, taking the complement gives us our true intended value.

ARM Assembly Programming

By: Tony Ngo

CSC3210 Assignment 3 Task 4

```
@ Third program

.section .data
a: .shalfword -2 @ 16-bit signed integer

.section .text
.globl _start
_start:
@ The following is a simple ARM code example that atte

mov r0, #0x1 @ = 1
mov r1, #0xFFFFFFFF @ = -1 (signed)
mov r2, #0xFF @ = 255
mov r3, #0x101 @ = 257
mov r4, #0x400 @ = 1024

mov r7, #1 @ Program Termination: exit syscall

svc #0 @ Program Termination: wake kernel
.end
```

This is code that I wrote for the third program in nano.

```
root@raspberrypi:~/Tony/8-Bit/Tony# as -g -o third.o third.s
third.s: Assembler messages:
third.s:4: Error: unknown pseudo-op: `.shalfword'
```

This is the error that I got in the code while trying to compile; I got rid of the ".shalfword" and changed it into .hword because in ARM, the type is automatically assigned to signed. You just need to use the "-sh" at the end of the movements to move signed halfword values.

```
GNU gdb (Raspbian 8.2.1-2) 8.2.1
Copyright (C) 2018 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 third...done.
(gdb) list
1       @ Third program
2
3       .section .data
4       a: .hword -2 @ 16-bit signed integer
5
6       .section .text
7       .globl _start
8       _start:
9               @ The following is a simple ARM code example that attempts to load a set of values
into registers and it might have problems.
10
(gdb)
11              mov r0, #0x1 @ = 1
12              mov r1, #0xFFFFFFFF @ = -1 (signed)
13              mov r2, #0xFF @ = 255
14              mov r3, #0x101 @ = 257
15              mov r4, #0x400 @ = 1024
16              mov r7, #1 @ Program Termination: exit syscall
17              svc #0 @ Program Termination: wake kernel
18      .end
(gdb) b 7
Breakpoint 1 at 0x10078: file third.s, line 12.
(gdb) x/1xh 0x10078
0x10078 <_start+4>:     0x1000
(gdb) x/1xsh 0x10078
0x10078 <_start+4>:     u"α?x20ff?x3101?? ??
```

I used the command “gdb third” to open the file in the GDB debugger. I then used “list” to show all of the lines in the code. Then I set a breakpoint at 7 using “b 7”, which created a breakpoint at memory 0x10078. I used that to display the memory at the location 0x10078 using “x/1xh 0x10078” first and got 0x1000. I then used “x/1sh 0x10078” because we are using a signed word in this register and got random characters, that is because that is what the signed value is in hex.

```
@ Arithmetic 3

.section .data
        val1: .byte -60 @ 8-bit unsigned integer
        val2: .byte 11  @ 8-bit unsigned integer
        val3: .byte 16  @ 8-bit signed integer

.section .text
.globl _start
_start:
        @ The following ARM program does:
        @ Register = val2 + 3 + val3 - val1

        ldr r1, = val1          @ load the memory address of val1 into r1
        ldrb r1, [r1]           @ load the value of val1 into r1

        ldr r2, = val2          @ load the memory address of val2 into r2
        ldrb r2, [r2]           @ load the value of val2 into r2

        ldr r3, = val3          @ load the signed memory address of val3 into r3
        ldrsb r3, [r3]          @ load the signed memory address of val3 into r3

        add r2, r2, #3          @ val2 + 3 /  11+3 = 15

        sub r3, r1, r3          @ val3 - val1 / 16 - -60

        add r4, r2, r3          @ result of val2 + 3 + result of val3 - val1 stored into r4

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

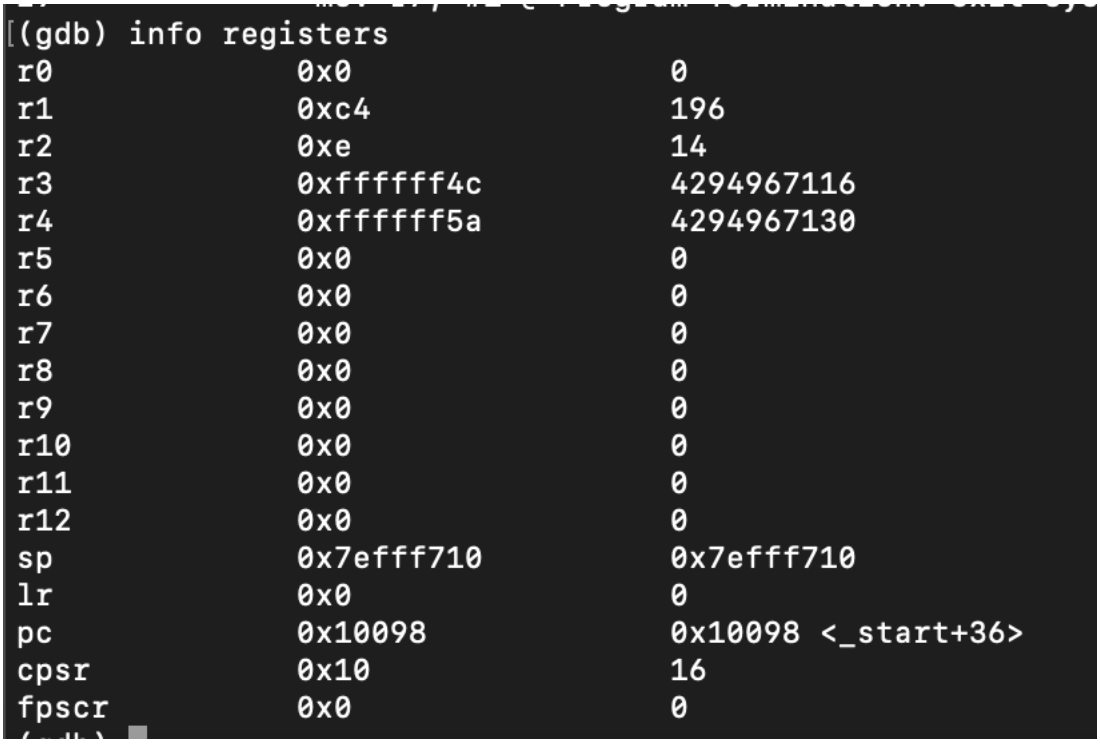
This is code that I wrote for the Arithmetic 3 program using the Nano text editor.

```
[root@raspberrypi:~/Tony/8-Bit/Tony# nano arithmetic3.s
[root@raspberrypi:~/Tony/8-Bit/Tony# as -g -o arithmetic3.o arithmetic3.s
[root@raspberrypi:~/Tony/8-Bit/Tony# ld -o arithmetic3 arithmetic3.o
[root@raspberrypi:~/Tony/8-Bit/Tony# gdb arithmetic3
GNU gdb (Raspbian 8.2.1-2) 8.2.1
```

I used “as -g -o arithmetic3.o arithmetic3.s” which links the program with the GDB debugger “-g”, I then used “ld -o arithmetic3 arithmetic3.o” to create the executable for the file which works for the GDB debugger. I started the GDB debugger using “gdb arithmetic3”.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from arithmetic3...done.
(gdb) list
1       @ Arithmetic 3
2
3       .section .data
4               val1: .byte -60 @ 8-bit unsigned integer
5               val2: .byte 11  @ 8-bit unsigned integer
6               val3: .byte 16  @ 8-bit signed integer
7
8       .section .text
9       .globl _start
10      _start:
(gdb)
11              @ The following ARM program does:
12              @ Register = val2 + 3 + val3 - val1
13
14              ldr r1, = val1          @ load the memory address of val1 into r1
15              ldrb r1, [r1]           @ load the value of val1 into r1
16
17              ldr r2, = val2          @ load the memory address of val2 into r2
18              ldrb r2, [r2]           @ load the value of val2 into r2
19
20              ldr r3, = val3          @ load the signed memory address of val3 into r3
(gdb)
21              ldrsb r3, [r3]          @ load the signed memory address of val3 into r3
22
23              add r2, r2, #3          @ val2 + 3 /  11+3 = 15
24
25              sub r3, r1, r3          @ val3 - val1 / 16 - -60
26
27              add r4, r2, r3          @ result of val2 + 3 + result of val3 - val1 stored into r4
28
29              mov r7, #1 @ Program Termination: exit syscall
30              svc #0 @ Program Termination: wake kernel
(gdb)
31      .end
(gdb) b 21
Breakpoint 1 at 0x10088: file arithmetic3.s, line 21.
(gdb) run
Starting program: /root/Tony/8-Bit/Tony/arithmetic3

Breakpoint 1, _start () at arithmetic3.s:21
21              ldrsb r3, [r3]          @ load the signed memory address of val3 into r3
(gdb) stepi
23              add r2, r2, #3          @ val2 + 3 /  11+3 = 15
```

This is what happens when you run the debugger. I used list to show my code, and then created a breakpoint at b 21 because I wanted to see what each arithmetic operation did because I was running into issues with my code.



Here are what my registers are after the code. The answer is 90 which is shown in r4 (since I used r4 to store my final value) as “0xffffff5a”, but if you take two’s complement the actual result is 90.

Appendix

Slack: https://app.slack.com/client/TTK19C222/CTGQG7H7A/user_profile/UTGB5U71S

GitHub: https://github.com/Rsandy2/8-Bit

Presentation on YouTube: https://www.youtube.com/watch?v=ggZFmFTanaE

Screenshot of Introductions on Slack: