

**Developing Soft and Parallel Programming
Skills Using Project-Based Learning**

SPRING 2019

ARM Strong

Group Members:

Ailany Icassatti, Hashim Amin, Isaiah Smith,
Sivasubramaniyan Mourougassamy,
Thang Nguyen & Toan Le

❖ **Planning and Scheduling**

Work Breakdown Structure

Assignee Name	Email	Task	Duration	Dependency	Due Date	Notes
Ailany Icassatti	aicassatti1@student.gsu.edu	Task 3 B (Running Loops in Parallel code 1 & 2) Task 6 (making the group video)	8 hours 2 hours	Meeting up with group members	Mar .5 Mar . 8	
Hashim Amin (Coordinator)	hamin3@student.gsu.edu	Compile, revise, format final report	2 hours	All subparts of the final report must be completed before I can compile it	Mar . 7	
Isaiah Smith	ismith27@student.gsu.edu	Task 3 part A (Answering all of the questions for the first draft then editing after group looks over.)	First Draft: 2 hours Editing : 1-1.5 hours	Group giving me feedback to I can grade it.	Mar . 6	

Sivasubramanian Mourougassamy	smourougassamy1@student.gsu.edu	Task 4 (ARM Assembly Programming)	Creating and stepping through code with group: 1.5 hours Lab Report: 2.5 hours	GEF (GDB extended features), ARM info center	Mar .7	Note: installed GEF
Thang Nguyen	tnguyen469@student.gsu.edu	Task 3b-second part(When Loops Have Dependencies)	2.5 hrs	Depend on Pi functioning and groupmates discussing code	Mar .7	
Toan Le	tle96@student.gsu.edu	Record, edit and upload the presentation video to the team Youtube channel	1 hour	All members in the team meet up to present their tasks.	Mar .5	.

❖ Parallel Programming Skills

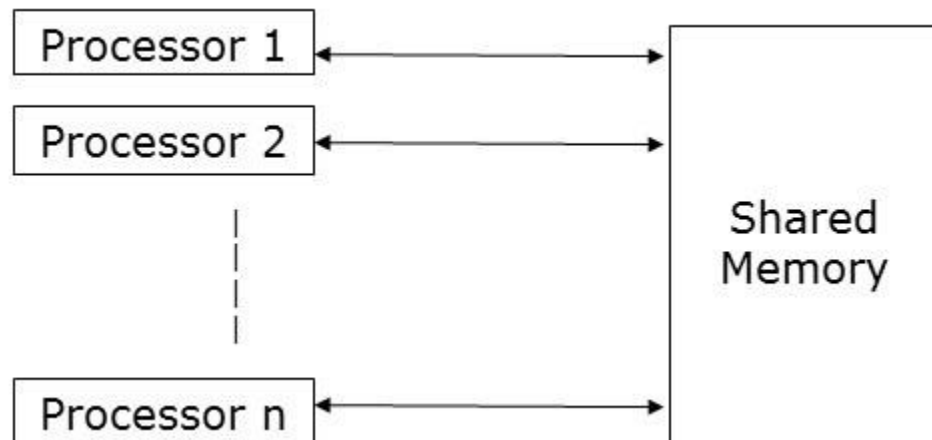
Introduction to Parallel Computing and the Raspberry Pi

1. **Define the following: Task, Pipelining, Shared Memory, Communications, Synchronization.**
 - Task: A task is a set of instructions and calculations that a computer performs
 - Pipelining: Pipelining is the process of breaking a task into smaller pieces to be performed by multiple different processors.
 - Shared Memory: From a hardware perspective, shared memory is when various processors have access to the same memory. From a programming perspective, it is where parallel tasks all have the same “picture” of memory and can access the same logical memory locations regardless of where the memory is physically.
 - Communications: Communication in terms of parallel programming is the process of parallel tasks communicating with each other through various means.
 - Synchronization: Synchronization is when parallel tasks communicate and coordinate in real time.
2. **Classify parallel computers based on Flynn’s taxonomy. Briefly describe every one of them.**
 - Based on Flynn’s Taxonomy, parallel computers are considered Multiple Instruction, Multiple Data Computers.
 - The types of parallel computers classified in Flynn’s Taxonomy include:
 - Single Instruction Stream, Single Data Stream (SISD): A serial computer that handles only one instruction stream and one data stream during a single clock cycle.
 - Single Instruction Stream Multiple Data stream (SIMD): A parallel computer that executes the same instruction at one clock cycle but each processing unit can operate on a different data element during a single clock cycle.
 - Multiple Instruction Stream Single Data Stream (MISD): A parallel computer where each processing unit operates on the data independently by using separate instruction streams, but one data stream is used as input for multiple processing units.
 - Multiple Instruction Stream, Multiple Data Stream (MIMD): A parallel computer where every processor executes a different instruction stream as well as a different data stream.
3. **What are the Parallel Programming Models**
 - Some Parallel Programming Models in use include:
 - Shared Memory (w/o Threads)
 - Threads
 - Distributed Memory / Message Passing
 - Data Parallel
 - Hybrid
 - Single Program Multiple Data (SPMD)
 - Multiple Program Multiple Data (MPMD)
4. **List and briefly describe the types of Parallel Computer Memory Architectures. What type is used by OpenMP and why?**

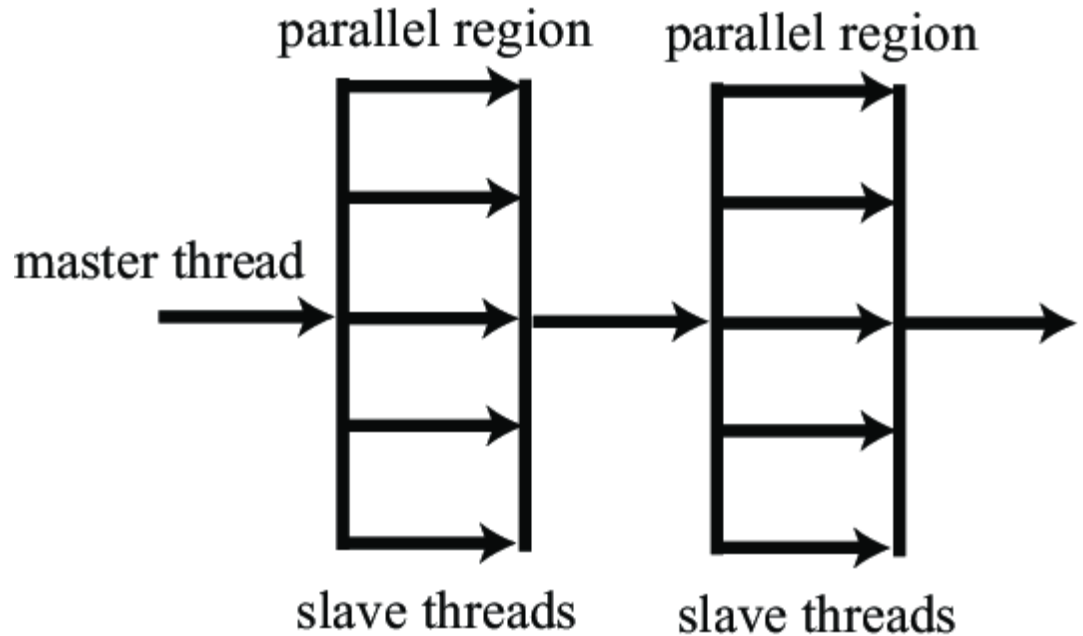
- The types of Parallel Computer Memory Architectures include:
 - Shared Memory – Uniform Memory Access: Architecture where all of the processors are identical, and all have equal access and access times to memory. Is also called Cache Coherent because if one processor updates a location in shared memory, all other processors know about it.
 - Shared Memory – Non – Uniform Memory Access: Architecture where multiple Symmetric Multiprocessor (SMP) machines are linked together. One SMP machines can access the memory of another SMP machine but not all processors have equal access time to memory
- OpenMP uses Shared memory - Uniform Memory Access because OpenMP is designed for use within one multi-core node meaning each process pulls memory from a single location.

5. Compare Shared memory Model with Threads Model? (In your own words and show pictures)

- Shared Memory Model:
 - Processes and tasks share common address space. The various processes write to this space at different times.
 - Locks and semaphores are used to limit access to the shared memory and resolve contentions and prevent issues like deadlocks.
 - Communication of data between tasks do not have to be specified, all tasks have equal access to shared memory.



- Threads Model:
 - In this model, a process can be broken down into smaller, more manageable pieces.
 - Each manageable chunk, or “thread”, shares the resources of the larger process as well as each having its own copy of memory.
 - All the threads are capable of running at any time as well as concurrently.
 - Threads communicate with each other by updating the global memory that they all have copies of one at a time.



Similarities	Differences
Each process or in Threads case, threads, can update the global memory they use. Just not at the same time.	Memory: Shared Memory Model each process accesses the same address space. Threads: Each thread has its own copy of the global memory that they use which updates.
Each process or thread can communicate with each other.	Shared Memory Model has multiple processes running at the same time while thread model has one process broken down into multiple threads.
Memory can be kept local to the process in Shared Memory and in Threads a “heavy” process provides the memory for all of the various threads.	

6. What is Parallel Programming? (In your own words)

- Parallel Programming is the practice of using multiple computer resources in order to solve a problem by breaking the problem into smaller steps, deliver instructions, and execute the solutions at the same time.

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

- A SoC is simply one silicon chip that performs the functions of various other chips such as memory, audio, and graphics. Yes, the Raspberry Pi uses the BCM2835 system on chip.

8. Explain what the advantages of having a System on a chip rather than separate CPU, GPU, and Ram components

- One advantage of a SoC is that it is only a little larger than a CPU but contains a lot more functionality along with that size. This means that computers small enough can be made for smartphones and tablets. Another advantage SoCs have is

that due to their very high level of integration along with much shorter wiring, SoCs tend to consume much less power which helps when they're being used in much smaller devices like smartphones.

Running Loops in Parallel - Part A

We compiled and ran, compared the output to the given source code, and tried with different numbers of threads 2, 3, 4, 6, 8 and 11.

File Name: parallelLoopEqualChunks.c

Source Code:

```
#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;
}
```

```
}
```

Questions:

1. How many and which iterations of the loop will each thread complete on its own?

The specifications of each case are above each picture.

2. What happens when the number of iterations (16 in this code) is not evenly divisible by the number of threads?

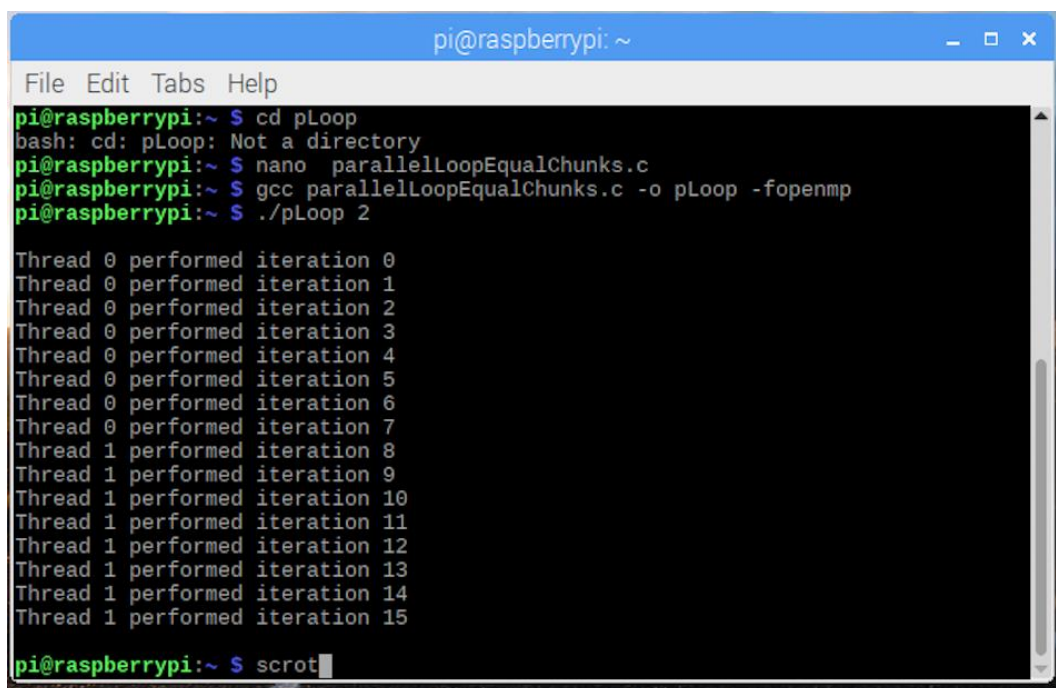
Each thread is still assigned an independent set of iterations, but the work is divided unequally, by assigning more iterations to the initial threads and less to the ending threads.

First, we verified with the command-line argument 4 that the behavior of the data decomposition pattern indicated how many threads to fork. We ran different numbers of threads to be certain how the compiler splits up the work. Results are below:

#1. Total_Threads = 2; Total_Iterations = 16

Thread[0] & Iteration[0-7]

Thread[1] & Iteration[8-15]



```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $ cd pLoop
bash: cd: pLoop: Not a directory
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp
pi@raspberrypi:~ $ ./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
Thread 1 performed iteration 15

pi@raspberrypi:~ $ scrot

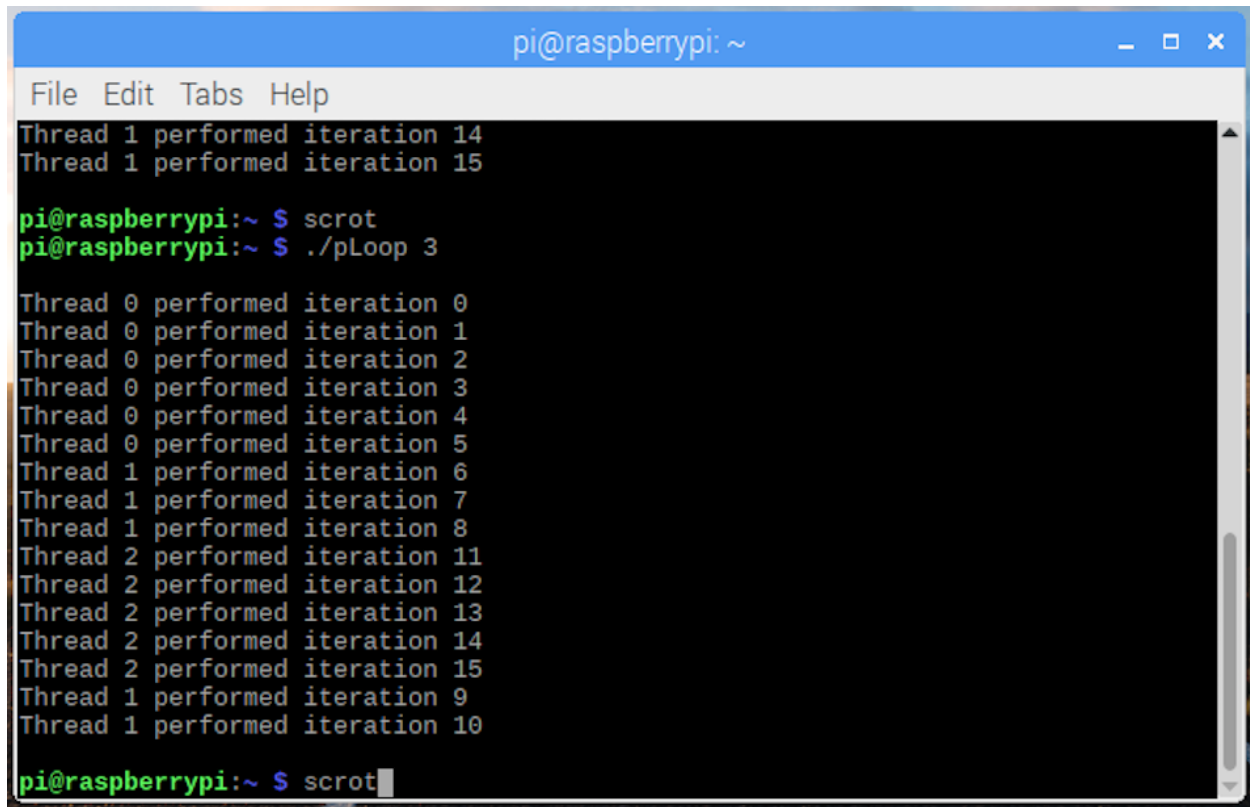
```


#2. Total_Threads = 3; Total_Iterations = 16

Thread[0] & Iteration[0-5]

Thread[1] & Iteration[6-10]

Thread[2] & Iteration[11-15]



```
pi@raspberrypi: ~  
File Edit Tabs Help  
Thread 1 performed iteration 14  
Thread 1 performed iteration 15  
pi@raspberrypi:~ $ scrot  
pi@raspberrypi:~ $ ./pLoop 3  
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 1 performed iteration 6  
Thread 1 performed iteration 7  
Thread 1 performed iteration 8  
Thread 2 performed iteration 11  
Thread 2 performed iteration 12  
Thread 2 performed iteration 13  
Thread 2 performed iteration 14  
Thread 2 performed iteration 15  
Thread 1 performed iteration 9  
Thread 1 performed iteration 10  
pi@raspberrypi:~ $ scrot
```

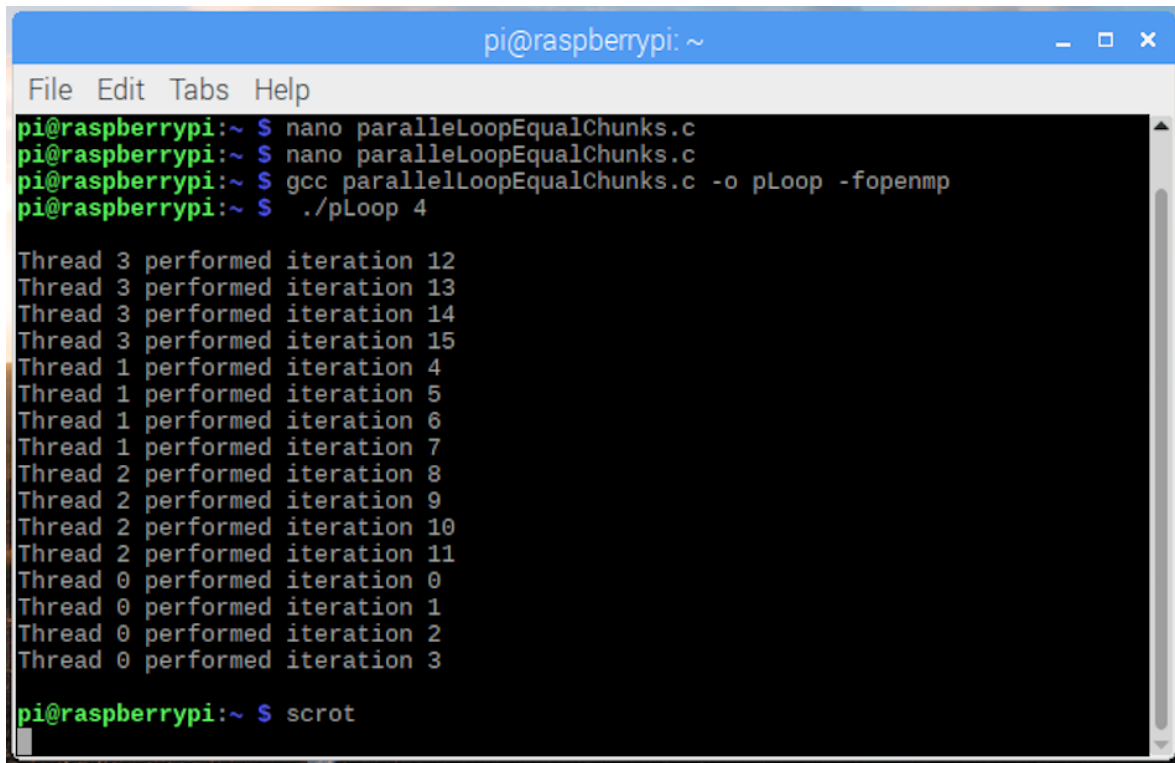
#3. Total_Threads = 4; Total_Iterations = 16

Thread[0] & Iteration[0-3]

Thread[1] & Iteration[4-7]

Thread[2] & Iteration[8-11]

Thread[3] & Iteration[12-15]



```
pi@raspberrypi: ~  
File Edit Tabs Help  
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c  
pi@raspberrypi:~ $ nano parallelLoopEqualChunks.c  
pi@raspberrypi:~ $ gcc parallelLoopEqualChunks.c -o pLoop -fopenmp  
pi@raspberrypi:~ $ ./pLoop 4  
  
Thread 3 performed iteration 12  
Thread 3 performed iteration 13  
Thread 3 performed iteration 14  
Thread 3 performed iteration 15  
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 0 performed iteration 0  
Thread 0 performed iteration 1  
Thread 0 performed iteration 2  
Thread 0 performed iteration 3  
  
pi@raspberrypi:~ $ scrot
```

#4. Total_Threads = 6; Total_Iterations = 16

Thread[0] & Iteration[0-2]

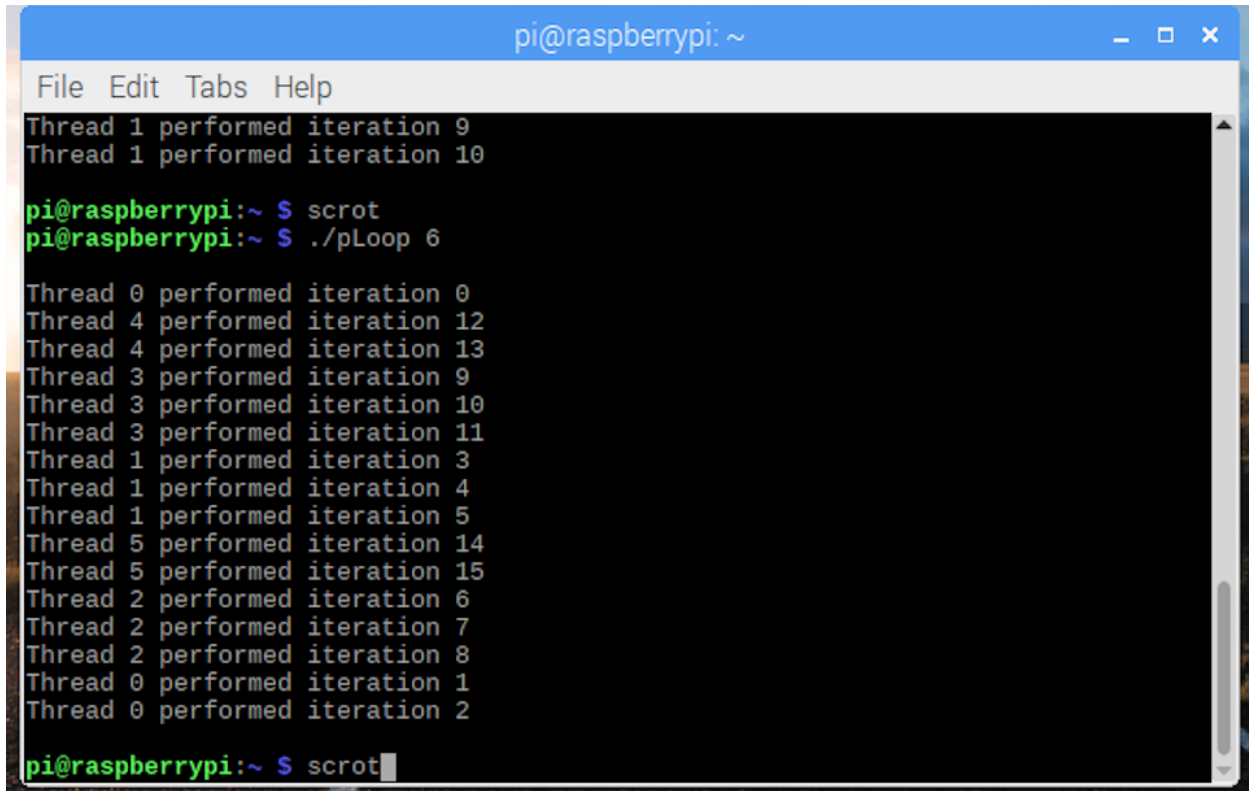
Thread[1] & Iteration[3-5]

Thread[2] & Iteration[6-8]

Thread[3] & Iteration[9-11]

Thread[4] & Iteration[12-13]

Thread[5] & Iteration[14-15]



The screenshot shows a terminal window titled "pi@raspberrypi: ~" with a menu bar containing "File", "Edit", "Tabs", and "Help". The terminal output displays the following:

```

Thread 1 performed iteration 9
Thread 1 performed iteration 10

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop 6

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

pi@raspberrypi:~ $ scrot

```

#5. Total_Threads = 8; Total_Iterations = 16

Thread[0] & Iteration[0-1]

Thread[1] & Iteration[2-3]

Thread[2] & Iteration[4-5]

Thread[3] & Iteration[6-7]

Thread[4] & Iteration[8-9]

Thread[5] & Iteration[10-11]

Thread[6] & Iteration[12-13]

Thread[7] & Iteration[14-15]

```

pi@raspberrypi: ~
File Edit Tabs Help
Thread 0 performed iteration 1
Thread 0 performed iteration 2

pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop 8

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

pi@raspberrypi:~ $ scrot

```

#6. Total_Threads = 11; Total_Iterations = 16

Thread[0] & Iteration[0-1]

Thread[1] & Iteration[2-3]

Thread[2] & Iteration[4-5]

Thread[3] & Iteration[6-7]

Thread[4] & Iteration[8-9]

Thread[5] & Iteration[10]

Thread[6] & Iteration[11]

Thread[7] & Iteration[12]

Thread[8] & Iteration[13]

Thread[9] & Iteration[14]

Thread[10] & Iteration[15]

Thread[11] & Iteration[16]

```

pi@raspberrypi: ~
File Edit Tabs Help
pi@raspberrypi:~ $
pi@raspberrypi:~ $ scrot
pi@raspberrypi:~ $ ./pLoop 11

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

pi@raspberrypi:~ $ scrot

```

In conclusion, the work on EqualChunks' code was divided between threads into chunks with the consecutive iterations of the loop, meaning chunks of work/data per thread using a parallel for loop implementation strategy.

Running Loops in Parallel - Part B

We compiled and ran ChunksOf1, and compared its output to the output of the EqualChunks. Different than EqualChunks, the loop in ChunksOf1 is not accessing consecutive memory locations, meaning each thread does one iteration up to N threads, and loops to the first thread again repeating the process. On EqualChunks file the code inside the the loop was just splitted between forked threads.

We uncommented the commented-out given code below and compared both codes inside the ChunksOf1 file. The first loop is simpler but more restrictive with the word static added to its parameter, meaning a clause scheduling each thread to do one iteration of the loop in regular pattern. While the second code in the fChunksOf1 file is less restrictive, allowing the next available thread to simply take the next iteration, but more complex because the loop is working with *dynamic* scheduling instead of the static scheduling.

File: parallelLoopChunksOf1.c

Code:

```

#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", id,i);
        }
    }
}

```

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

Top Code:

Total_Threads = 4; Total_Iterations = 16

Thread[0] & Iteration[0,4,8,12]

Thread[1] & Iteration[1,5,9,13]

Thread[2] & Iteration[2,6,10,14]

Thread[3] & Iteration[3,7,11,15]

```
pi@raspberrypi: ~
File Edit Tabs Help
Thread 2 performed iteration 14
Thread 0 performed iteration 12
Thread 0 performed iteration 15
pi@raspberrypi:~ S scrot
pi@raspberrypi:~ S ./pLoop2 4
Thread 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
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 2 performed iteration 10
Thread 2 performed iteration 14
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
---
Thread 2 performed iteration 2
Thread 2 performed iteration 6
Thread 2 performed iteration 10
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 0 performed iteration 0
Thread 0 performed iteration 4
Thread 0 performed iteration 8
Thread 0 performed iteration 12
Thread 1 performed iteration 1
Thread 1 performed iteration 5
Thread 1 performed iteration 9
Thread 1 performed iteration 13
pi@raspberrypi:~ S scrot
```

Bottom Code:

```
Total_Threads = 4; Total_Iterations = 16
```

```
Thread[0] & Iteration[0,4,8,12]
```

```
Thread[1] & Iteration[1,5,9,13]
```

```
Thread[2] & Iteration[2,6,10,14]
```

```
Thread[3] & Iteration[3,7,11,15]
```

In conclusion, the two codes in ChunksOf1 are just different ways of assigning threads to iterations using loops, but they produced the same output. While in the EqualChunks file, the code assigned a chunk of iterations per thread in parallel, where the loop access consecutive memory locations.

When Loops Have Dependencies

Part 1: Compile and run the program as it is

The code is entered in the “nano” text editor by typing the command “nano + filename” .



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

After entering the code, the file is saved by pressing Ctrl + O, and nano is closed by Ctrl + X combination. Below is a snippet of the program.

```
#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",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;
}

```

An executable program is made by typing the command “gcc reduction.c -o reduction -fopenmp”.

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

After creating the executable file, it is ran with the command “./reduction 4”. The number 4 is the number of threads or leave off. The result is print below the run command. The parallel result is the same as the correct Sequential result. This is because line 39 is commented out. This effectively makes the Sequential Sum and Parallel Sum methods identical. They both sum sequentially; the only difference is the comment.

```
pi@raspberrypi:~ $ ./reduction 4
Sequential sum:      499562283
Parallel sum:    499562283
```

Using a different number of threads does not change the result.

```
pi@raspberrypi:~ $ ./reduction 10
Sequential sum:      499562283
Parallel sum:    499562283
```

Part 2: Uncomment #pragma in function parallelSum(), but leave its reduction clause commented out

Next, we edited the reduction program by uncommenting “#pragma omp parallel for” in parallel sum method, but still leave its reduction clause commented. We ran it and noticed that the parallel sum method produced a difference results from the sequential sum method. The parallel sum must be wrong then. The results are below.

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

Sequential sum:      499562283
Parallel sum:    154816908
```

Using a different number of threads do change the result. The closer the number of threads is to one the closer the parallel sum is to the correct sequential sum. Although higher number of threads still produces incorrect numbers. At one thread the parallel sum method is effectively doing sequential sum. Here is the program ran with 3, 2, and 1 thread.

```
pi@raspberrypi:~ $ ./reduction 3
Sequential sum:      499562283
Parallel sum:    213646271
pi@raspberrypi:~ $ ./reduction 2
Sequential sum:      499562283
Parallel sum:    303368067
pi@raspberrypi:~ $ ./reduction 1
Sequential sum:      499562283
Parallel sum:    499562283
```

To take a closer look at the program and figure out why it was incorrect, we created and edit a version of the code called “reductionEDIT”. In it we made the array size much smaller to 4 to easily see the entire array. Also a lastsum variable was added to see the arithmetic and print statements were added to see the current state of the program. The differences are highlighted.

Here it is below.

```
#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 4

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",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;
printf("Array item %d: %d\n", i, a[i]);
}
}

/*sum the array sequentially*/
int sequentialSum(int* a,int n) {
int sum = 0;
int lastsum = 0;
int i;
for(i = 0;i<n;i++) {
lastsum = sum;
sum +=a[i];
printf("Sequential- curr item:%d curr sum:%d+%d=%d\n", a[i],a[i],lastsum ,sum);
}
```

```

return sum;
}

/*sum the array using multiple threads*/
int parallelSum(int* a, int n) {
int sum = 0;
int lastsum = 0;
int i;
// #pragma omp parallel for //reduction(+:sum)
for(i = 0;i<n;i++) {
int id = omp_get_thread_num();
lastsum = sum;
sum += a[i];
printf("Parallel- curr item:%d curr sum:%d+%d=%d thread:%d\n", a[i],a[i],lastsum,sum,id);
}
return sum;
}

```

We used our “reductionEDIT” program and ran it with “#pragma omp parallel for” uncommented. The results are below.

```

pi@raspberrypi:~ $ nano reductionEDIT.c
pi@raspberrypi:~ $ gcc reductionEDIT.c -o reductionEDIT -fopenmp
pi@raspberrypi:~ $ ./reductionEDIT 4
Array item 0: 383
Array item 1: 886
Array item 2: 777
Array item 3: 915
Sequential- curr item:383 curr sum:383+0=383
Sequential- curr item:886 curr sum:886+383=1269
Sequential- curr item:777 curr sum:777+1269=2046
Sequential- curr item:915 curr sum:915+2046=2961
Parallel- curr item:383 curr sum:383+886=1269 thread:0
Parallel- curr item:777 curr sum:777+886=1663 thread:2
Parallel- curr item:886 curr sum:886+0=886 thread:1
Parallel- curr item:915 curr sum:915+1663=2578 thread:3

Sequential sum:      2961
Parallel sum:   2578

```

What seems to be happening above is thread 1 is finishing first so the current sum is 886. Thread 0 then uses the sum of 886 to compute the updated sum. After that thread 2 also uses 886 to update the sum. Then thread 3 uses the sum of 1663 from thread 2 to update the final sum which is incorrect. This is happening because the sum variable currently is not private; instead it is being shared with all the threads, producing an incorrect sum.

Part 3: Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()

Next we uncomment “reduction(+:sum)” clause in the parallel sum method. It is then built the executable file and ran it. We noticed the result is correct for parallel again. Also changing the thread number does not change the result.

```

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

Sequential sum:      499562283
Parallel sum:    499562283

```

Using our “reductionEDIT” version we ran the code with “reduction(+:sum)” clause uncommented to investigate further; we have this result.

```

pi@raspberrypi:~ $ nano reductionEDIT.c
pi@raspberrypi:~ $ gcc reductionEDIT.c -o reductionEDIT -fopenmp
pi@raspberrypi:~ $ ./reductionEDIT 4
Array item 0: 383
Array item 1: 886
Array item 2: 777
Array item 3: 915
Sequential- curr item:383  curr sum:383+0=383
Sequential- curr item:886  curr sum:886+383=1269
Sequential- curr item:777  curr sum:777+1269=2046
Sequential- curr item:915  curr sum:915+2046=2961
Parallel- curr item:383  curr sum:383+0=383  thread:0
Parallel- curr item:777  curr sum:777+0=777  thread:2
Parallel- curr item:886  curr sum:886+0=886  thread:1
Parallel- curr item:915  curr sum:915+0=915  thread:3

Sequential sum:      2961
Parallel sum:    2961

```

What seems to be happening is each thread gets forked, all doing their task with their own private sum variable. In the end they are joined and the final sum is computed from the individual sums. Threads are not sharing the sum variable; each has its own copy. This is due to the reduction clause “reduction(+:sum)” where the sum variable has a dependency on what the other threads do.

❖ ARM Assembly Programming

Part 1 - Third.s Program

code for third.s:

```
@Third prog
.section .data
a: .shword -2
.section .text
.globl _start
_start:
mov r0,#0x1 @=1
mov r1,#0xFFFFFFFF @-1 (signed)
mov r2,#0xFF @255
mov r3,#0x101 @=257
mov r4,#0x400 @=1024

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

After writing the code, we encountered some problems, as the assembler did not recognize .shword as a data type.

Because of the error assigning the data type signed half-word to variable a, half-word data type with a signed negative data value was used instead:

```
@Third prog
.section .data
a: .hword -2
.section .text
.globl _start
_start:
mov r0,#0x1 @=1
mov r1,#0xFFFFFFFF @-1 (signed)
mov r2,#0xFF @255
mov r3,#0x101 @=257
mov r4,#0x400 @=1024

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

After correcting the issue, we used the assembler to create an objective file:

```
as -g -o third.o third.s
```

Using the linker, we created the executable file from the objective file:

```
ld -o third third.o
```

With the executable file, we ran the program and debugged with gdb:

```
gdb third
```

```

pi@raspberrypi:~/ProjectPiAssignment2/A3 $ gdb third
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-gnueabi".
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"...
ligGEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 7.12.0.20161007-git using Python engine 3.5
[*] 5 commands could not be loaded, run `gef missing' to know why.
Reading symbols from third...done.
gef> list
1      @Third prog
2      .section .data
3      a: .hword -2
4      .section .text
5      .globl _start
6      _start:
7      mov r0,#0x1 @=1
8      mov r1,#0xFFFFFFFF @-1 (signed)
9      mov r2,#0xFF @255
10     mov r3,#0x101 @=257
gef>
11     mov r4,#0x400 @=1024
12
13     mov r7,#1
14     svc #0
15     .end
gef>
Line number 16 out of range; third.s has 15 lines.
gef>

```

To test the operations through the debugger, we can place a breakpoint at line 7 and step through the 5 mov instructions, checking the registers each time.

```

7      mov r0,#0x1 @=1
→ 8      mov r1,#0xFFFFFFFF @-1 (signed)
9      mov r2,#0xFF @255
10     mov r3,#0x101 @=257
11     mov r4,#0x400 @=1024
12
13     mov r7,#1

[#0] Id 1, Name: "third", stopped, reason:
[#0] 0x10078 → _start()

Breakpoint 1, _start () at third.s:8
8      mov r1,#0xFFFFFFFF @-1 (signed)
gef> info registers
r0          0x1      0x1
r1          0x0      0x0

```

After performing the instruction `mov r0,#0x1`, the value `0x1` is moved into `r0`

We can also check the flags register with the debugger (In the case of ARM assembly, the flags register is known as CPSR, as opposed to EFLAGS in intel x86 assembly).

```

$sp : 0x7efff000 - 0x00000001
$lr : 0x0
$pc : 0x00010088 - <_start+20> mov r7, #1
$cpsr: [thumb fast interrupt overflow carry zero negative]

```

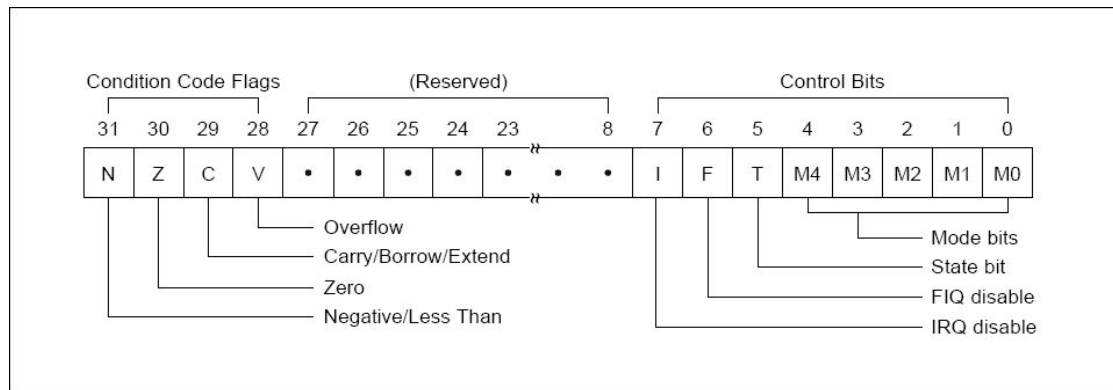
Using GEF (GDB extended features), we are able to see the thumb, fast, interrupt, overflow, carry, zero, and negative flags.

```

pc      0x10088 0x10088
cpsr    0x10 0x10

```

But this is also possible by using the `info registers` command with GDB, which will display the value in the CPSR register.



This diagram displays what each bit in the CPSR register represents. Using this, we can determine if any flags are triggered by examining the value of CPSR.

Because the value in the 32-bit CPSR register is 0x10, we know that none of the condition code flags are triggered after the `mov r0,#0x1` instruction. This is because the move operation does not access the ALU of the processor (which is reserved for arithmetic and logic operations).


```

      8  mov r1,#0xFFFFFFFF @-1 (signed)
→     9  mov r2,#0xFF @255
     10  mov r3,#0x101 @=257
     11  mov r4,#0x400 @=1024
     12
     13  mov r7,#1
     14  svc #0

[#0] Id 1, Name: "third", stopped, reason:
[#0] 0x1007c → _start()

9      mov r2,#0xFF @255
gef> info registers
r0      0x1      0x1
r1      0xffffffff 0xffffffff
r2      0x0      0x0

```

After performing the `mov r1,#0xFFFFFFFF` operation, we can see that the value is stored into `r1` as expected. Once again, none of the condition flags for are triggered because this is a move operation.

After performing all 5 `mov` operations, we can see that the values are properly stored into the general purpose registers `r0-r4`:

```

gef> info registers
r0      0x1      0x1
r1      0xffffffff 0xffffffff
r2      0xff      0xff
r3      0x101     0x101
r4      0x400     0x400
r5      0x0      0x0
r6      0x0      0x0
r7      0x0      0x0
r8      0x0      0x0
r9      0x0      0x0
r10     0x0      0x0
r11     0x0      0x0
r12     0x0      0x0
sp      0x7efff000 0x7efff000
lr      0x0      0x0
pc      0x10088   0x10088 <_start+20>
cpsr    0x10     0x10

```

We can also see that none of the condition flags were triggered in any of the operations.

Part 2 - Arithmetic3.s Program

Arithmetic3 presents us with the task of calculating this expression:

Register = val2 + 3 + val3 - val1

Performing the operations from right to left, the expression can also be written as such:

Register = (val2 + (3 + (val3 - val1)))

Where the values are:

val1 = -60 (unsigned)

val2 = 11 (unsigned)

val3 = 16 (signed)

Before we load the values in memories val1-val3 into registers r1-r3, we must take into account that we are moving signed and unsigned byte variables. Simply loading them with the ldr will yield results such as:

r1	0x41100bc4	0x41100bc4
r2	0x1341100b	0x1341100b
r3	0x134110	0x134110

To fix this, we must use the suffixes for the ldr instruction to deal with signed and unsigned byte variables:

```
ldr = Load Word
ldrh = Load unsigned Half Word
ldrsh = Load signed Half Word
ldrb = Load unsigned Byte
ldrsb = Load signed Bytes
```

Here is the code with the correct load operations:

```
@arithmetic 3
.section .data
val1: .byte 60 @unsigned
val2: .byte 11 @unsigned
val3: .byte 16 @signed

.section .text
.globl _start
_start:

ldr r1, =val1
ldrb r1,[r1] @load unsigned byte

ldr r2, =val2
ldrb r2,[r2] @load unsigned byte

ldr r3, =val3
ldrsb r3,[r3] @load signed byte

sub r3,r3,r1 @subtract r1 from r3 and store into r3
add r3,r3,#0x3 @add immediate 3 to r3
add r2,r2,r3 @add r3 to r2 and store in r2

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

There is still one more thing to take care of. If we debug the code as is, the condition flags in the CPSR register (flags) will not change. We must add the `s` suffix to our arithmetic operations to allow the condition flags in the CPSR register to change.

Here is the code with the corrected arithmetic operations:

```
@arithmetic 3
.section .data
val1: .byte -60 @unsigned
val2: .byte 11 @unsigned
val3: .byte 16 @signed

.section .text
.globl _start
_start:

ldr r1, =val1
ldrb r1,[r1] @load unsigned byte

ldr r2, =val2
ldrb r2,[r2] @load unsigned byte

ldr r3, =val3
ldrsb r3,[r3] @load signed byte

subs r3,r3,r1 @subtract r1 from r3 and store into r3
adds r3,r3,#0x3 @add immediate 3 to r3
adds r2,r2,r3 @add r3 to r2 and store in r2

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

Now we can create our debuggable executable to test the program.

```
pi@raspberrypi:~/ProjectPiAssignment2/A3 $ as -g -o arithmetic3.o arithmetic3.s
pi@raspberrypi:~/ProjectPiAssignment2/A3 $ ld -o arithmetic3 arithmetic3.o
pi@raspberrypi:~/ProjectPiAssignment2/A3 $ gdb arithmetic3
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
```

Assembling and linking the file did not give us any issues.

```
20      sub r3,r3,r1 @subtract
gef> info registers
r0          0x0      0x0
r1          0xc4     0xc4
r2          0xb      0xb
r3          0x10     0x10
```

We can see that the values from memory have been loaded into the registers properly with the use of the correct suffixes for the `ldr` instruction.

After performing the `subs r3,r3,r1` operation, which adds the `r1` to `r3` and stores the result in `r3`, we can see that the value in `r3` has changed:

```

21      adds r3,r3,#0x3 @add immediate 3 to
gef> info registers
r0      0x0      0x0
r1      0xc4     0xc4
r2      0xb      0xb
r3      0xffffffff4c  0xffffffff4c

```

We can also take a look at the flags, in which the negative flag should be triggered:

```
cpsr      0x80000010  0x80000010
```

According to the diagram above, this indeed indicates that the negative flag was triggered. We can double check this using GEF:

```
$cpsr: [thumb fast interrupt overflow carry zero NEGATIVE]
```

After performing the `adds r3,r3,#0x3` operation, which adds the immediate value 3 to r3 and stores the result in r3, we can see that the value in r3 has changed:

```

22      adds r2,r2,r3 @add r3 to r2 and sto
gef> info registers
r0      0x0      0x0
r1      0xc4     0xc4
r2      0xb      0xb
r3      0xffffffff4f  0xffffffff4f

```

We can also take a look at the flags, in which the negative flag should still be triggered:

```
cpsr      0x80000010  0x80000010
```

We can double check this using GEF:

```
$cpsr: [thumb fast interrupt overflow carry zero NEGATIVE]
```

After performing the `adds r2,r2,r3` operation, which adds r3 to r2 and stores the result in r2, we can see that the value in r2 has changed:

```

24      mov r7,#1
gef> info registers
r0      0x0      0x0
r1      0xc4     0xc4
r2      0xffffffff5a  0xffffffff5a
r3      0xffffffff4f  0xffffffff4f

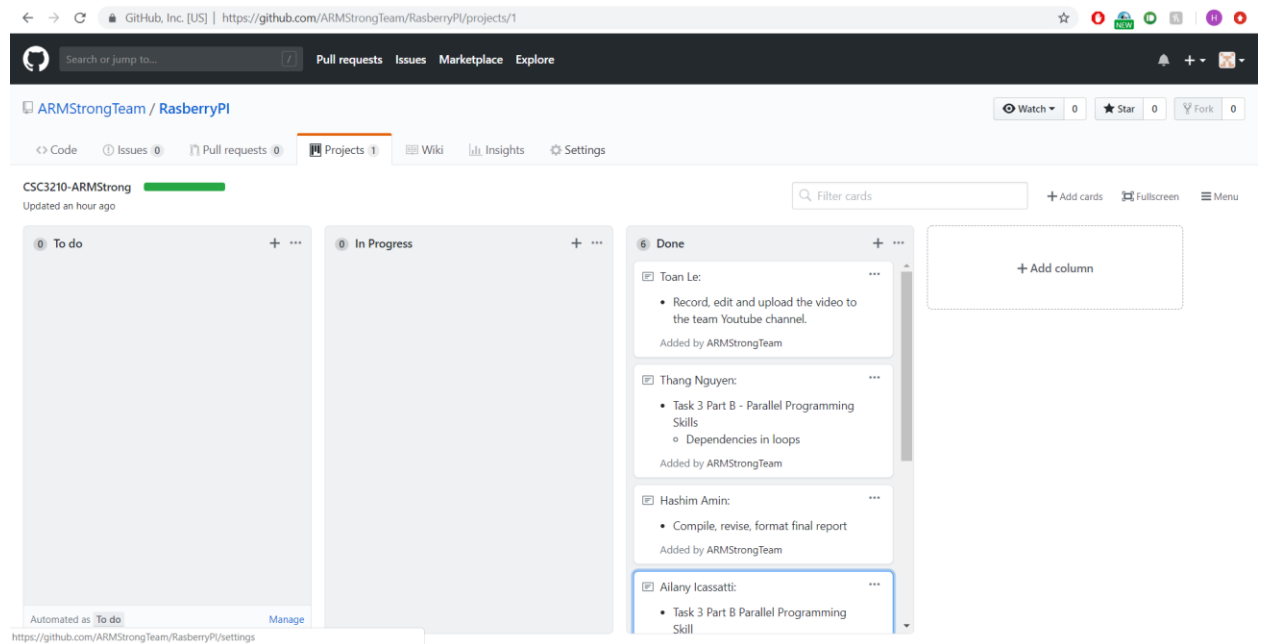
```

CPSR should still indicate that the negative flag is triggered:

```
cpsr      0x80000010  0x80000010
$cpsr: [thumb fast interrupt overflow carry zero NEGATIVE]
```

❖ Appendix

Github Tasks Screenshot:



Youtube Video Link:

https://www.youtube.com/watch?v=TePE4M0zhJU&list=PLTLekPDMOIn2A1IHlf5pOO_jOZVoXyoeP&index=3

Slack Link:

<https://armstrongprojectteam.slack.com/>