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

TASK 1: Planning and Scheduling

Assignee name	Email	Task	Duration (hours)	Dependency	Due Date	Note
Ailany Icassatti	aicassatti1@student.gsu.edu	• Task 4: ARM Assembly. • Task 6: presentation	20 hours	Rasberry Pi to test the program.	03/27	
Hashim Amin	hamin3@student.gsu.edu	• Task 3 Part B: Integration. • Task 6: presentation.	2.5 hours	Rasberry Pi to observe the execution.	03/27	
Isaiah Smith	ismith27@student.gsu.edu	• Task 3 Part A: Dependencies. • Task 6: presentation.	1 hour		03/26	
Sivasubramaniyan Mourougassamy	smourougassamy1@student.gsu.edu	• Task 3 Part B: Barrier. • Task 6: presentation.	2 hours	Rasberry Pi to observe the execution.	03/26	
Thang Nguyen	tnguyen469@student.gsu.edu	• Task 3 Part A: Race Condition. • Task 6: presentation, edit and upload the video.	3 hours		03/26	
Toan Le (Coordinator)	tle96@student.gsu.edu	• Task 3 Part B: Master-worker Implementation. • Task 6: presentation. • Task 5: Finalize the report.	4 hours	Rasberry Pi to observe the execution.	03/27	

TASK 2: Collaboration (Communicate through Slack and upload materials to GitHub)

TASK 3: Parallel Programming Skill

a) Foundation

Race Condition

1. What is race condition?

Race condition is the behavior of the software or system where the output is dependent on the sequence or timing of other uncontrollable events. It is when the application relies on the order of the processes or threads for it to operate properly. When the order is not happening as intended, the race condition becomes a bug. If the processes or threads depend on some shared state, a critical race condition can occur. Operations on this shared space must be separated, if not the shared section may be corrupted.

2. Why race condition is difficult to reproduce and debug?

   Race condition is difficult to reproduce and debug because the end result is nondeterministic. Meaning the output does not depend on the input. Given the same input, the result can be different. This is because the result depends on the relative timing between interfering threads. Problems that occurred can disappear when running debugging mode, attaching a debugger, or when additional logging is added. Because of this, race conditions should be avoided instead of trying to fix them.

3. How can it be fixed?

   To fix a race condition, the part of the program where it occurs must be changed or removed. The processes are "racing" each other, and this should not be the case. To locate the offending code the programmer can use logging, printing everything out to record. Reverse debugging or debugging until a failure occurs can also be used. The programmer can work backward to find the race condition. However, it is best to avoid a race condition in first place with careful programming. In Project_A3, "reduction" program there is a race condition when using "#pragma omp parallel for" without using "reduction(+:sum)". Given the same array input, the output kept changing; it was nondeterministic. The race condition was located and fixed by adding the "reduction(+:sum)" clause.

4. Summarize the Parallel Programming Patterns section in the "Introduction to Parallel Computing_3.pdf" in your own words.

   "Strategies" is one pattern or ways of writing code. There are two considerations. First are algorithmic strategies, which is choosing which tasks can be done concurrently. Second is implementation strategies, which can contribute to the overall structure or how the data computation is structured. "Concurrent Execution Mechanisms" is the other pattern, and it is parallel code patterns and the software library used to enable parallelism. Concurrent execution patterns fall into two categories. The first is Process/Thread control which dictates how the processing units of parallel execution are controlled at run time. Second is Coordination which set up how multiple concurrently running tasks on the processing units coordinate. There are two coordination patterns. First is message passing between concurrent processes. Second is mutual exclusion between threads executing concurrently. A third emerging type is hybrid computation that uses both patterns; using a cluster of computers and often uses MPI and OpenMP together.

5. Compare the following

   - Collective synchronization (barrier) with collective communication (reduction)

     Collective synchronization is the coordination of parallel tasks by establishing a synchronization point where a task may not continue until other tasks reach the same or equivalent point. It is like a barrier that stops the tasks and makes it waits for the slowest task, often increasing the application's clock execution time. Collective communication is the communication of data from all processes. It collects data from all processes and computes the result from that data. It combines or reduces from an all-to-one process. Communication can be through a shared memory bus or over a network.

- Master-worker with fork-join.

  In master-worker the master process set up a pool of worker processes and holds a task queue. The workers execute concurrently, each repeatedly taking a task and processing it then reporting back to the master. While fork-join is when a main process forks or branches out into several threads or processes, those threads then continue in to run parallel, each doing a portion of the overall work, they are then joined when the tasks are completed. While mater-worker and fork-join are different program structure strategies, the difference is not rigid, and there is overlap. For example, master-worker can be used to implement fork-join.

Dependencies

1. Where can we find parallelism in programming?

   When processes or calculations can be performed more quickly by doing them concurrently.

2. What is dependency and what are its types (provide one example for each)?

   Dependency is when one operation requires the completion of an earlier operation with a result before this operation can be dependent. There are three types of dependency:

   - True (flow) dependence: A statement is dependent on the statement that comes before it.

     Example:

     S1: n1=5;

     S2: n2=n1;

   - Output Dependence: When the output of a statement is dependent on order of statements.

     Example:

     S1: n1 = 3x + 5;

     S2: n1 = n2;

   - Anti-dependence: When a statement is dependent on the outcome of a statement that comes after it.

     Example:

     S1: n1 = n2;

     S2: n2 = 300;

3. When is a statement dependent and when is it independent (Provide two examples)?

   A statement is independent if its outcome is independent of when another statement completes. I.e., the order of which they execute does not impact their outcomes. A statement is dependent when its outcome is impacted by order of execution.

   Example:

   S1: n1 = 5x +3;

   S2: n2 = 15;

   These statements are independent because their outcomes are not dependent on when they finish or are executed.

   S1: n1 = 15x -10;

   S2: n2 = n1 +50;

   These states are dependent because statement 2 requires that statement 1 finishes before it does so it can have the value of n1.

4. When can two statements be executed in parallel?

   The only time when two statements can be executed in parallel is when either statement doesn't have a form of dependency on the other.

5. How can dependency be removed?

   Depending on the type of dependence we can rearrange statements or eliminate statements.

6. How do we compute dependency for the following two loops and what types of dependency?

   To calculate dependence, we compare the IN and OUT of each node. IN is the set of memory locations or variables that are used in a statement and OUT is the variables modified by a statement. For both of the for loops, each iteration is independent because each iteration does not depend on a previous or future one because the outcome of each iteration only depends on the loop index. This means that each iteration can be done in parallel if we so desired.

   To find the dependence, we would have to "unroll" the loop and then try to find any dependence between iterations. In this case, there is no dependence between iterations.

b) Parallel Programming Basics

Integration using the trapezoidal rule

The code below demonstrates how an integration of the sine function can be implemented using parallel programming implementation and OpenMP (albeit with some errors):

trap-notworking.c

```c
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP computes trapezoidal approximation to an integral*/

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a)/n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of thread */
if(argc > 1) {
threadcnt = atoi(argv[1]);

}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadcnt = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for private(i) shared (a,n,h,integral)
for(i=1; i<n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

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

```
pi@raspberrypi:~ $ nano trap-notworking.c
```

After entering the code, the file is saved by pressing Ctrl + O, and nano is closed by pressing Ctrl + X.

The code is compiled by typing the command “gcc trap-notworking.c -o trap-notworking -fopenmp”

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

The code above does not output the correct value of 2.0 because there is an issue with line 37:

```
#pragma omp parallel for private(i) shared (a,n,h,integral)
```

The issue here is that the integral variable has been set to share. What this means is that each thread being run may read and write to the integral variable concurrently, causing errors in the integral calculations.

Here is trap-notworking.c running with threads 2-4:

```
pi@raspberrypi:~ $ ./trap-notworking 2
OMP defined, threadcnt = 2
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.782651
pi@raspberrypi:~ $ ./trap-notworking 3
OMP defined, threadcnt = 3
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.577272
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadcnt = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.397534
pi@raspberrypi:~ $
```

As can be seen here, the final value of the integral variable is never 2.0. This result is because trap-notworking.c has a race condition with regards to the shared variable integral (accumulation of all integral values calculated in the for loop). When two or more threads are being executed, the execution times of each thread can differ, and so one thread may be reading the shared integral variable while another thread may be writing to the shared integral variable. This problem causes the integral variable to omit some of the partial integral values.

NOTE: This is not the case when threadcnt is equal to one because then there is no issue of the reading and writing of the integral variable being out of order with regards to separate threads.

```
pi@raspberrypi:~ $ ./trap-notworking 1
OMP defined, threadcnt = 1
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
```

The result above is correct (2.0) because only one thread is being run, meaning there is no race condition issue.

The code below is the same as trap-notworking.c except the race condition issue has been resolved by using the reduction clause in the OpenMP pragma:

trap-working.c

```c
#include <math.h>
#include <stdio.h> // printf()
#include <stdlib.h> // atoi()
#include <omp.h> // OpenMP


/* Demo program for OpenMP computes trapezoidal approximation to an integral*/

const double pi = 3.14159265358979323846264338307 9;

int main(int argc, char** argv) {
/* Variables */
double a = 0.0, b = pi; /* limits of integration */;
int n = 1048576; /* number of subdivisions = 2^20 */
double h = (b - a)/n; /* width of subdivision */
double integral; /* accumulates answer */
int threadcnt = 1;

double f(double x);

/* parse command-line arg for number of thread */
if(argc > 1) {
threadcnt = atoi(argv[1]);

}

#ifdef _OPENMP
omp_set_num_threads( threadcnt );
printf("OMP defined, threadcnt = %d\n", threadcnt);
#else
printf("OMP not defined");
#endif

integral = (f(a) + f(b))/2.0;
int i;

#pragma omp parallel for \
private(i) shared (a,n,h,) reduction(+: integral)
for(i=1; i<n; i++) {
integral += f(a+i*h);
}

integral = integral * h;
printf("With %d trapezoids, our estimate of the integral from \n", n);
printf("%f to %f is %f\n", a,b,integral);
}

double f(double x) {
return sin(x);
}
```

The integral variable has been moved from the shared clause to a new added clause, reduction

Adding the reduction clause to the OpenMP and putting the integral variable in it resolved the race condition issue was happening in the trap-notworking.c code. This is because the reduction clause makes the integral variable private for each thread and then upon the completion of each thread the values are added to the global variable, integral.

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

```
pi@raspberrypi:~ $ nano trap-working.c
```

After entering the code, the file is saved by pressing Ctrl + O, and nano is closed by pressing Ctrl + X.

The code is compiled by typing the command “gcc trap-working.c -o trap-working -fopenmp”

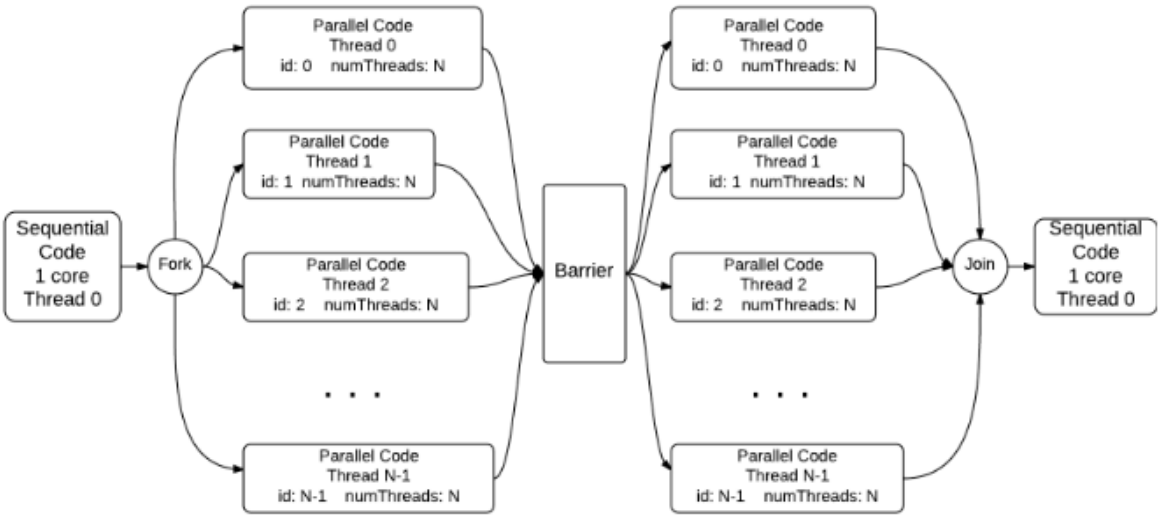
```
pi@raspberrypi:~ $ gcc trap-working.c -o trap-working -fopenmp
```

As can be seen here, trap-working.c works no matter the number of threads the for loop is being run on:

```
pi@raspberrypi:~ $ ./trap-working 1
OMP defined, threadcnt = 1
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $ ./trap-working 2
OMP defined, threadcnt = 2
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $ ./trap-working 3
OMP defined, threadcnt = 3
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $ ./trap-working 4
OMP defined, threadcnt = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 2.000000
pi@raspberrypi:~ $
```

Coordination: Synchronization with a Barrier

This task focuses on the Pragma barrier, which is a point in parallelized code which stops all threads from continuing to execute tasks until they have reached the same point in the code. Here is a visual representation of how the Pragma barrier operates within a task using multiple threads once it has been forked:



To test this functionality, we can use this code:

```c
//barrier.c

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

int main(int argc, char** argv){
printf("\n");
if(argc>1){
omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier. \n", id,numThreads);

//        #pragma omp barrier

printf("Thread %d of %d is AFTER the barrier.\n", id,numThreads);
}

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

Notice that the barrier is commented out in this version of the code, this is so that we can have a control to see how the code outputs without any barriers.

Here is the output for the code without the barrier after compiling and running:

```
pi@raspberrypi:~ $ gcc nobarrier.c -o nobarrier -fopenmp
pi@raspberrypi:~ $ ls
2019-03-26-012304_1824x984_scrot.png  2019-03-27-225345_182
pi@raspberrypi:~ $ ./nobarrier

Thread 3 of 4 is BEFORE the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 2 of 4 is AFTER the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
```

As displayed above, the output shows that the task was simply forked into multiple tasks and then joined, without any involvement of a barrier. Each of the threads may finish the first instruction and move on to the second without waiting for the other threads to finish the first instruction.

To test the Pragma barrier, we can uncomment the barrier line in the code:

```c
//barrier.c

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

int main(int argc, char** argv){
printf("\n");
if(argc>1){
omp_set_num_threads( atoi(argv[1]) );
}
#pragma omp parallel
{
int id = omp_get_thread_num();
int numThreads = omp_get_num_threads();
printf("Thread %d of %d is BEFORE the barrier. \n", id,numThreads);

        #pragma omp barrier

printf("Thread %d of %d is AFTER the barrier.\n", id,numThreads);
}

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

This puts the barrier in between the first and the second instruction, so the threads should wait for all other threads to finish the first instruction before moving to the second instruction.

As expected, after compiling the code and running, this is exactly what happens:

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

Thread 3 of 4 is BEFORE the barrier.
Thread 1 of 4 is BEFORE the barrier.
Thread 0 of 4 is BEFORE the barrier.
Thread 2 of 4 is BEFORE the barrier.
Thread 0 of 4 is AFTER the barrier.
Thread 1 of 4 is AFTER the barrier.
Thread 3 of 4 is AFTER the barrier.
Thread 2 of 4 is AFTER the barrier.
```

Program Structure: The Master-Worker Implementation Strategy

The code below demonstrates the master-worker implementation in the most basic way.

```
/*masterWorker.c
*illustrates the master-worker pattern in OpenMP
*
*Joel Adams, Calvin College, November 2009
*
*Usage:./masterWorker
*
*Exercise:
*-Compile and run as is.
*-Uncomment #pragma directive, re-compile and re-run
*-Compare and trace the different executions
*/

#include<stdio.h> //printf()
#include<stdlib.h> //atoi()
#include<omp.h> //OpenMP

int main(int argc,char** argv) {
   printf("\n");
   if(argc>1) {
      omp_set_num_threads(atoi(argv[1]));
   }

   //#pragma omp parallel
   {
      int id = omp_get_thread_num();
      int numThreads = omp_get_num_threads();

      if(id==0) {  //thread with ID 0 is master
         printf("Greetings from the master, #%d of %d threads\n",id,numThreads);
      } else { //threads with IDs>0 are workers
         printf("Greetings from a worker, #%d of %d threads\n",id,numThreads);
      }
   }

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

Line #24, “pragma omp parallel” is used to run the code in the block below it run in parallel to take the advantage of the multi-core processor. To show that, we run this program twice, one with this line commented out and one without.

Without line 24, only one core of the processor is used, numThreads is 1 and the id is always 0. Thus, no matter how many times we run this code and what core is used, the result will be all the same. The only core that is used to run the code is declared as the master with its id as 0.

```
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ gcc masterWorker.c -o masterWorker -fopenmp
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ ./masterWorker

Greetings from the master, #0 of 1 threads

pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $
```

When line 24 comes to play, the program will have all cores of the processor to work. Each core is summoned to run randomly, specifically from 0 to 3 in our case because we have a four-core processor. This time, we expect to have results from both the master and workers. When the program notifies that it uses the core labeled 0, it will run the master code block, and the same applies to other cores, except the worker code block will be run instead.

Here is the second version of the code with line 24 included.

```
/*masterWorker.c
*illustrates the master-worker pattern in OpenMP
*
*Joel Adams, Calvin College, November 2009
*
*Usage:./masterWorker
*
*Exercise:
*-Compile and run as is.
*-Uncomment #pragma directive, re-compile and re-run
*-Compare and trace the different executions
*/

#include<stdio.h> //printf()
#include<stdlib.h> //atoi()
#include<omp.h> //OpenMP

int main(int argc,char** argv) {
   printf("\n");
   if(argc>1) {
      omp_set_num_threads(atoi(argv[1]));
   }

   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      int numThreads = omp_get_num_threads();

      if(id==0) {  //thread with ID 0 is master
         printf("Greetings from the master, #%d of %d threads\n",id,numThreads);
      } else { //threads with IDs>0 are workers
         printf("Greetings from a worker, #%d of %d threads\n",id,numThreads);
      }
   }

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

Below is the expected output when we have pragma OpenMP enabled.

```
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ gcc masterWorker_ver2.c -o masterWorker_ver2 -fopenmp
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ ./masterWorker_ver2

Greetings from the master, #0 of 4 threads
Greetings from a worker, #3 of 4 threads
Greetings from a worker, #1 of 4 threads
Greetings from a worker, #2 of 4 threads

pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $
```

Since the program does not call four cores in order, it is normal to have the master code block executed in the middle of the process, or at the end of the process. Here are some examples:

```
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ gcc masterWorker_ver2.c -o masterWorker_ver2 -fopenmp
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ ./masterWorker_ver2

Greetings from a worker, #1 of 4 threads
Greetings from a worker, #2 of 4 threads
Greetings from the master, #0 of 4 threads
Greetings from a worker, #3 of 4 threads

pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $
```

```
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ gcc masterWorker_ver2.c -o masterWorker_ver2 -fopenmp
pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $ ./masterWorker_ver2

Greetings from a worker, #1 of 4 threads
Greetings from a worker, #2 of 4 threads
Greetings from a worker, #3 of 4 threads
Greetings from the master, #0 of 4 threads

pi@raspberrypi:~/Documents/CSC3210_Project/Assignment_4 $
```

TASK 4: ARM Assembly Programming

a) Part 1: Fourth Program

Error is encountered on the given code on the "thenpart: mov r2,1", we fix it by adding at #1 instead of 1 to the code and change from ldr r2,[r3] to str r2,[r3] to store the value of r2 to memory.

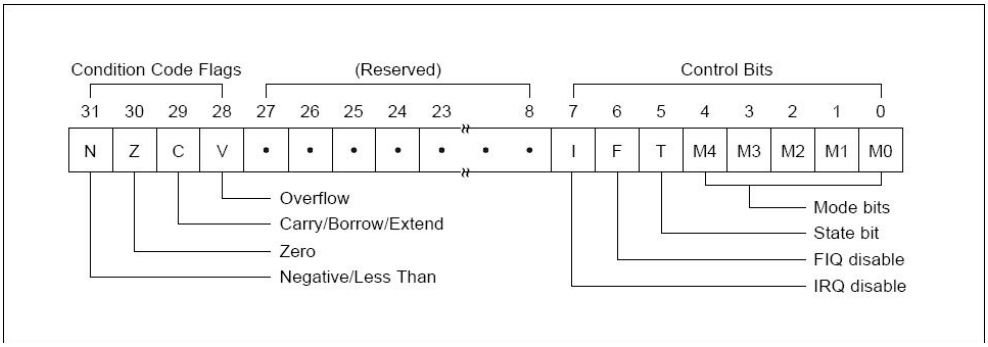
fourth.s

```
.section .text
.globl _start
_start:
          ldr r1,=x          @load the memory address of x into r1
          ldr r1,[r1]         @load the value x i

          cmp r1,#0           @
          beq thenpart        @branch((jump) if true (Z==1) to the thenpart
          b endofif           @branch (jump) if false to the of if statement body (branch always)
thenpart: mov r2,#1
          ldr r3, =y          @load the memory address of y into r3
          str r2,[r3]         @store r2 register value into y memory address

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

After correcting the issue, we create the objective file using the assembler. Then the objective file is used to make the executable file. Finally, we use gdb to debug the code

```
pi@raspberrypi:~ $ nano fourth.s
pi@raspberrypi:~ $ as -g -o fourth.o fourth.s
fourth.s: Assembler messages:
fourth.s:15: Error: bad instruction `ldrs r1,=x'
fourth.s:22: Error: bad instruction `ldrs r3,=y'
pi@raspberrypi:~ $ nano fourth.s
pi@raspberrypi:~ $
pi@raspberrypi:~ $ as -g -o fourth.o fourth.s
pi@raspberrypi:~ $ ld -o fourth fourth.o
pi@raspberrypi:~ $ gdb fourth
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from fourth...done.
(gdb)
```

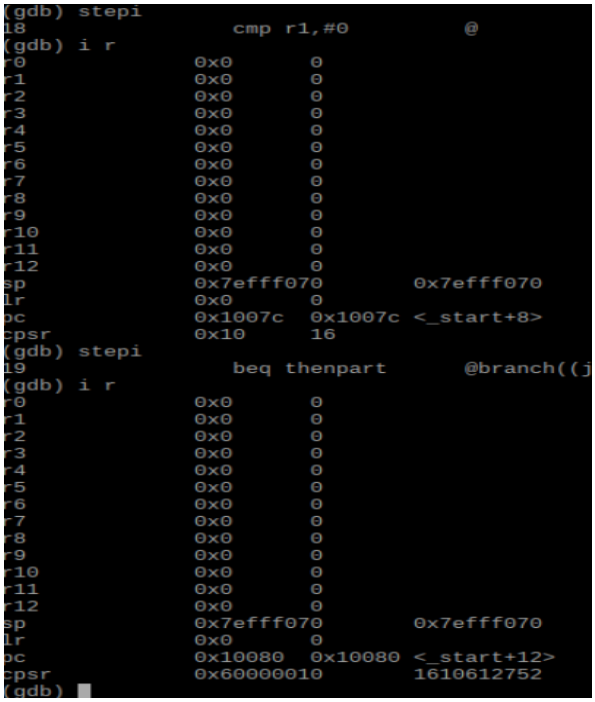
The diagram below displays what each bit in the CPSR register represents. Using this, we can determine if any flags are triggered by examining the value of CPSR.



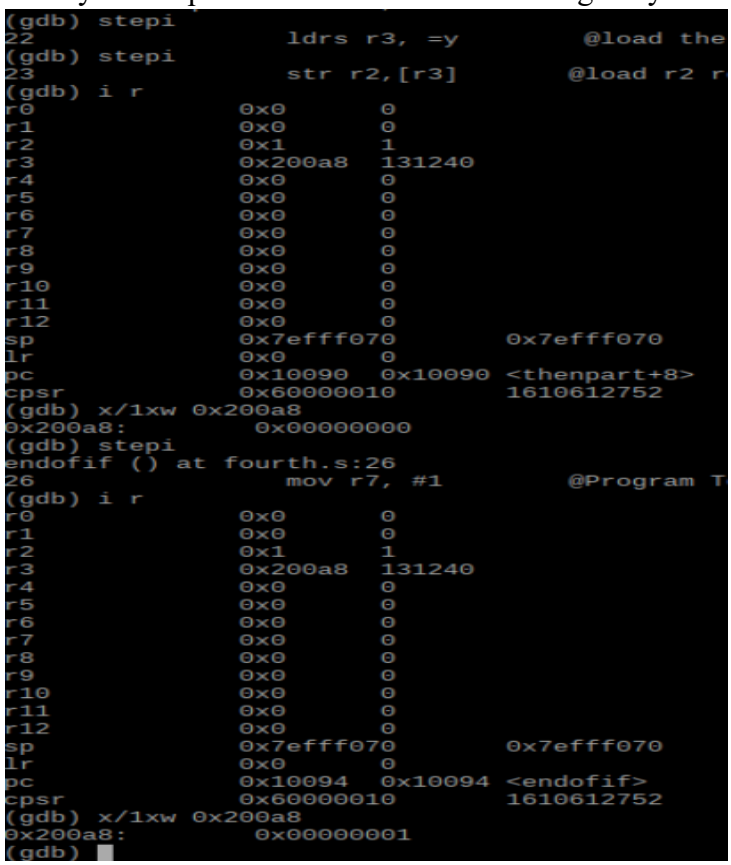
Because the value in the 32-bit CPSR register is 0x10, we know that none of the condition code flags are triggered after the ldr r1, [r1] instruction.

```
(gdb) list
1       @Fourth program
2       @This program compute the following if statement construct:
3               @intx;
4               @inty;
5               @if(x==0)
6               @ y = 1;
7
8       .section .data
9       x: .word 0  @32-bit signed int
10      y: .word 0  @32-bit signed int
(gdb) b 7
Breakpoint 1 at 0x10078: file fourth.s, line 7.
(gdb) run
Starting program: /home/pi/fourth

Breakpoint 1, _start () at fourth.s:16
16              ldr r1,[r1]       @load the value x i
(gdb) stepi
18              cmp r1,#0         @
(gdb) i r
r0             0x0      0
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             0x7efff070       0x7efff070
lr             0x0      0
pc             0x1007c  0x1007c <_start+8>
cpsr           0x10     16
(gdb)
```

Since CMP instruction can set - Z, O, C, and S. The condition flags zero and carry are triggered after the CMP r1,#0 instruction. The zero flag is set whenever the result of the subtraction is equal to zero. This situation only occurs when the operands are equal. For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produces a borrow (that is, an unsigned underflow), and to 1 otherwise. Then, beq is true and the thenpart instruction is executed.



By comparing x/1xw 0x200a8 before and after the str r2,[r3] instruction, we can see the value stored into r3 at 0x200a8 is as expected. The value 1 from r2 was loaded in r3 or into y memory address since the thenpart was excuted. Again, the condition flags zero and carry were triggered by the cmp instruction and weren't changed by the end.



Conclusion: Is the program in part one efficient?

The answer is no because the code contains back-to-back branches (**beq** followed by **b**). We want to avoid this, since branches may cause a delay slot.

b) Part 2

For this part we replace **beq** with **bnq (**branch on not equal (Z==0)) and remove **b** instruction from the code, as shown in the screenshot below.

fourth1.s

```
@Fourth program
@This program compute the following if statement construct:
        @intx;
        @inty;
        @if(x==0)
        @ y = 1;

.section .data
x: .word 0  @32-bit signed int
y: .word 0  @32-bit signed int

.section .text
.globl _start
_start:
          ldr r1,=x         @load the memory address of x into r1
          ldr r1,[r1]        @load the value x i

          cmp r1,#0          @
          bne endofif       @branch((jump) if not equal (Z==0) to the  endofif
          mov r2,#1
          ldr r3, =y         @load the memory address of y into r3
          str r2,[r3]        @str r2 register value into y memory address
endofif:
          mov r7, #1         @Program Termination: exit syscall
          svc #0             @Program Termination: wake kernel
          .end
```

To run the program, we create objective file from the source, then use that to form the executable file. Finnaly, gdb is used to debug.

Because the value in the 32-bit CPSR register is 0x10, we know that none of the condition code flags are triggered after the ldr r1, [r1] instruction.

```
(gdb) list
1       @Fourth program
2       @This program compute the following if statement construct:
3               @intx;
4               @inty;
5               @if(x==0)
6               @ y = 1;
7
8       .section .data
9       x: .word 0  @32-bit signed int
10      y: .word 0  @32-bit signed int
(gdb) b 7
Breakpoint 1 at 0x10078: file fourth1.s, line 7.
(gdb) run
Starting program: /home/pi/fourth1

Breakpoint 1, _start () at fourth1.s:16
16              ldr r1,[r1]       @load the value x i
(gdb) stepi
18              cmp r1,#0         @
(gdb) i r
r0             0x0      0
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             0x7efff070       0x7efff070
lr             0x0      0
pc             0x1007c  0x1007c <_start+8>
cpsr           0x10     16
(gdb)
```

When we replace beq with bnq (branch on not equal (Z==0)) and remove b instruction from the code, and one extra instruction, we jump when the condition is false which is different from what we do in the high-level languages. In Other words, we reversed the Boolean expression using DE Morgan Law (< is >=, > is <=, == is !=, <= is > and >= is <). Because the if-statement is false, no branch is executed, and the code keeps going just like part 1.

Again, like part 1, the condition flags zero and carry are triggered by the cmp instruction and are not changed by the end.

```
Breakpoint 1, _start () at fourth1.s:16
16              ldr r1,[r1]       @load the value x i
(gdb) stepi
18              cmp r1,#0         @
(gdb) stepi
19              bne endofif      @branch((jump) if not equal (Z==0) to the  endofif
(gdb) stepi
20              mov r2,#1
(gdb) i r
r0             0x0      0
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             0x7efff070       0x7efff070
lr             0x0      0
pc             0x10084  0x10084 <_start+16>
cpsr           0x60000010       1610612752
(gdb)
```

By comparing x/1xw 0x200a8 before and after the str r2,[r3] instruction, we can see the value stored into r3 at 0x200a8 as expected. The value 1 from r2 is loaded in r3 or into y memory address since the thenpart is excuted.

```
(gdb) stepi
23              str r2,[r3]       @str r2 register value into y memory address
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x1      1
r3             0x200a4  131236
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff070       0x7efff070
lr             0x0      0
pc             0x1008c  0x1008c <thenpart+8>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a4
0x200a4:        0x00000000
(gdb) stepi
endofif () at fourth1.s:25
25              mov r7, #1        @Program Termination: exit syscall
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x1      1
r3             0x200a4  131236
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff070       0x7efff070
lr             0x0      0
pc             0x10090  0x10090 <endofif>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a4
0x200a4:        0x00000001
(gdb)
```

Conclusion: The program in ‘Part 2’ is more efficient than ‘Part 1’ because we avoid back-to-back branches that could cause a delay slot.

c) Part 3

We follow the same steps from Part 1 and 2.

Expression used

Write a program that calculates the following expression:

```
if  X <= 3
      X = X – 1
else
      X = X – 2
```

file code ControlStructure1.s

```
.section .data
x: .word 1  @32-bit signed int


.section .text
.globl _start
_start:
        ldr r4,=x           @load the memory address of x into r4
        ldr r1,[r4]         @load the value x i

        cmp   r1,#3         @compare x =1 to number 3
        bgt   thenpart      @branch((jump) if greater to the thenpart
        subs  r1,r1,#1      @x=x-1
        b     endofif
thenpart: subs  r1,r1,#1      @x=x-2

endofif:
        str   r1,[r4]      @store value in x memory
        mov   r7,#1        @Program Termination: exit syscall
        svc   #0            @Program Termination: wake kernel
        .end
```

To debug this code, we use as and ld to build objective file and executable file respectively. Then we enter debugging mode by gdb.

```
pi@raspberrypi:~ $ as -g -o ControlStructure1.o ControlStructure1.s
pi@raspberrypi:~ $ ld -o ControlStructure1 ControlStructure1.o
pi@raspberrypi:~ $ gdb
GNU gdb (Raspbian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Because the value in the 32-bit CPSR register is 0x10, we know that none of the condition code flags are triggered after the ldr r1, [r1] instruction. When the code runs, r4 takes the x value 1 before the first ‘stepi’ and after it r4 value is loaded in r1.

```
(gdb) b 7
Breakpoint 1 at 0x10078: file ControlStructure1.s, line 7.
(gdb) run
Starting program: /home/pi/ControlStructure1

Breakpoint 1, _start () at ControlStructure1.s:16
16              ldr r1,[r4]         @load the value x i
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10078  0x10078 <_start+4>
cpsr           0x10     16
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
(gdb) stepi
18              cmp   r1,#3         @compare x =1 to number 3
(gdb) i r
r0             0x0      0
r1             0x1      1
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x1007c  0x1007c <_start+8>
cpsr           0x10     16
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
```

Again, like Part 1 and Part 2, the condition flags zero and carry are triggered by the CMP instruction, but now the sign flag is triggered as well. In CMP if an operand is subtracted from another of equal value, the Zero flag is set. The Sign flag indicates that an operation produces a negative result. If the most significant bit of the destination operand is set, the Sign flag is set. Eflag value is set to CPSR = 0x80000010. Also, it does not (bgt) branch to the thenpart because x is not greater than 3.

```
(gdb) stepi
19                bgt   thenpart      @branch((jump) if greater to the thenpart
(gdb) i r
r0             0x0      0
r1             0x1      1
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10080  0x10080 <_start+12>
cpsr           0x80000010       -2147483632
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
(gdb) stepi
20                subs  r1,r1,#1      @x=x-1
(gdb) i r
r0             0x0      0
r1             0x1      1
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10084  0x10084 <_start+16>
cpsr           0x80000010       -2147483632
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
```

Here the value 1 is subtracted from r1 and stored in r1, making r1 = 0x0 and changing the Eflag value to CPSR = 0x60000010, untriggers the sign flag.

```
0x200a0:        0x00000001
(gdb) stepi
21                b     endofif
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10088  0x10088 <_start+20>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
(gdb) stepi
endofif () at ControlStructure1.s:25
```

In the picture below, the ‘b endofif’ branches to endofif, and skips the ‘thenpart.’ Also, the value of r1 is stored in r4 which is connected to the x memory.

```
(gdb) stepi
endofif () at ControlStructure1.s:25
25                str   r1,[r4]      @store value in x memory
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10090  0x10090 <endofif>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a0
0x200a0:        0x00000001
(gdb) stepi
26                mov   r7,#1        @Program Termination: exit syscall
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10094  0x10094 <endofif+4>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a0
0x200a0:        0x00000000
(gdb) stepi
27                svc   #0            @Program Termination: wake kernel
(gdb)
```

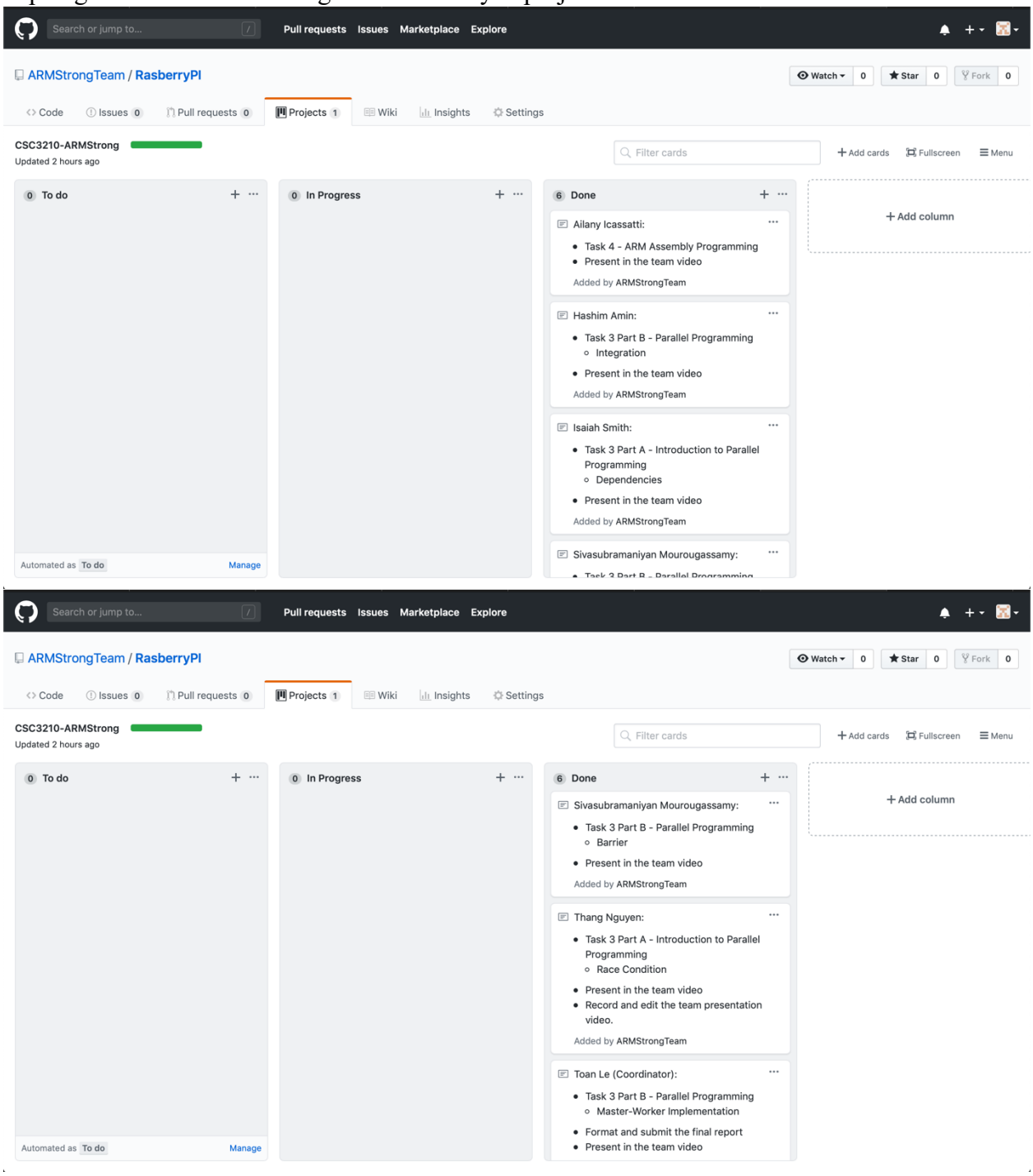
The program ends

```
(gdb) stepi
26                mov   r7,#1        @Program Termination: exit syscall
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x200a0  131232
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             0x7efff060       0x7efff060
lr             0x0      0
pc             0x10094  0x10094 <endofif+4>
cpsr           0x60000010       1610612752
(gdb) x/1xw 0x200a0
0x200a0:        0x00000000
(gdb) stepi
27                svc   #0            @Program Termination: wake kernel
(gdb)
```

In conclusion: We use two branches to run our code, and we reverse the Boolean expression using DE Morgan Law (<= is >). The x memory changes from 1 to 0 according to our code instructions. Since CMP instruction can set - Z, O, C, and S. The condition flags zero, carry, and sign are triggered after the CMP r1,#3 instruction. The zero flag is set whenever the result of the subtraction is equal to zero. This case only occurs when the operands are equal. For subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produces a borrow (that is, an unsigned underflow), and to 1 otherwise. In CMP if an operand is subtracted from another of equal value, the Zero flag is set. The Sign flag indicates that an operation produces a negative result. If the most significant bit of the destination operand is set, the Sign flag is set. Eflag value is set to CPSR = 0x80000010. But once the value 1 is subtracted from r1 and stored in r1, r1 = 0x0 and the Eflag value changes to CPSR = 0x60000010 and does not trigger the sign flag.

APPENDIX:

- Github link and screenshot:
  https://github.com/ARMStrongTeam/RasberryPI/projects/1



- Slack link:
  https://armstrongprojecteam.slack.com
- Youtube video link:
  https://www.youtube.com/watch?v=tAT0AO1Ez8c