DEVELOPING SOFT AND PROGRAMMING SKILLS USING PROJECT BASED LEARNING

(FALL 2019) *THE 6-PACK:*

TEAM MEMBERS:

HAZEL SANTIAGO

RACHID BODSON

JENNIFER VU

ABRAHAM MAMMEN

BRYAN RUDY GONZALES

LAUREN JAMES

Planning and Scheduling (T-1):

Work Breakdown Structure

Assignee Name	Email	Task	Duration (hours)	Dependency	Due Date	Note
Lauren James **(coordinator)**	ljames26@student.gsu.edu	-Create the work breakdown -Slack Invitation to TA sent -Create new columns and cards for project's assigned tasks on GitHub -ARM Assembly Programming participation -Parallel Programming Basics participation -Proofread and submit report electronically -Hand in hard copy of report to instructor	-1 -0.5 -0.5 -2 -2 -1	-Reading and understanding the Parallel Programming Basics instructions	-11/8/19 -11/8/19 -11/14/19 -11/19/19	
Hazel Santiago	hsantiago1@student.gsu.edu	-ARM Assembly Programming participation -Parallel Programming Basics participation -Parallel Programming Foundation	-2 -2 -2	-Answers to Parallel Foundation -Reading and understanding the Parallel Programming Basics instructions	-11/8/19 -11/8/19 -11/12/19	-Schedule and reserve meeting rooms in the library
Rachid Bodson	rbodson1@student.gsu.edu	-ARM Assembly Programming participation -Parallel Programming Basics participation -ARM Assembly Programming Lab Report	-2 -2 -3	-Reading and understanding the Parallel Programming Basics instructions -Understanding ARM Assembly Programmi	-11/8/19 -11/8/19 -11/12/19	

				ng instructions		
Bryan Gonzales	bgonzales1@student.gsu.edu	-ARM Assembly Programming participation -Parallel Programming Basics participation -Project Report	-2 -2 -3	Screenshots from Slack & GitHub, and Lab Reports needed to complete report -Reading and understanding the Parallel Programming Basics instructions	-11/8/19 -11/8/19 - 11/13/19	-Report needed 30 hours before deadline
Abraham Mammen	amammen1@student.gsu.edu	-ARM Assembly Programming participation -Parallel Programming Basics participation -Video: Shooting & Editing	-2 -2 -3	-Reading and understanding the Parallel Programming Basics instructions	-11/8/19 -11/8/19 - 11/14/19	-Video needed 24 hours before deadline
Jennifer Vu	jvu@student.gsu.edu	-ARM Assembly participation -Parallel Programming Basics participation -Parallel Programming Lab Report	-2 -2 -3	-Reading and understanding the Parallel Programming Basics instructions	-11/8/19 -11/8/19 - 11/12/19	

Parallel Programming Skills Foundation (T-3):

1. **Race Condition:**
   A. What is race condition?

   Race condition, or race hazard, is the behavior of electronics, software, or other systems where the output is dependent on the sequence/timing of uncontrollable events. In short, it

is when there's a reliance on the program to run as properly intended. They can occur in logic circuits, and in multithreaded/distributed software programs.

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

Race conditions have a nondeterministic end-result and depend on timing between interfering threads, "race." Problems can disappear in debug mode, when more logging is added, or when attaching a debugger (often referred as a "Heisenbug"). All this makes it seemingly impossible to reproduce the condition.

C. How can it be fixed? Provide an example from your Project_A3 (spmd2.c).

2. Summarize the Parallel Programming Patterns section in the "Introduction_to_Parallel_Computing_4.pdf" in your own words.

There are two main categories in parallel programming patterns are grouped: Strategies and Concurrent Execution Mechanisms. The two primary strategies are: arithmetic and implementation strategies. Arithmetic strategies involve choosing which tasks can be done concurrently by multiple processing units executing concurrently. Implementation strategies accompany arithmetic ones. Parallel uses several implementations. Some contribute to the overall structure of the program while others concentrate on how the data computed by multiple processors are structured. The two main concurrent executions are: process/thread (which dictate how the processing units of parallel execution on the hardware are controlled at run time) and coordination patterns. The latter set up how multiple concurrently running tasks coordinate to complete parallel computation; there are two. One passes messages between concurrent processes on either single multiprocessor machines or clusters of computers (MPI). The other mutually excludes concurrent executing threads on a single shared memory (OpenMP). There is also a hybrid.

3. In the section "Categorizing Patterns," compare the following:

A. Collective synchronization (barrier) with Collective communication (reduction).

Collective synchronization or barrier is used in parallel programming to ensure all threads complete a section of parallel of code before execution. When threads are generating computed data, barrier is necessary to be completed for use in another computation. Collective communication, or reduction, combines every element in a collection using an associative "combiner function." In this case, different orderings in reduction are possible.

B. Master-worker with fork join.

Master-worker is when there is one thread, called a master, execute one block of code in its fork while the rest of the threads, called workers, execute a different block of code when they fork. Fork-join allows control flow to fork into multiple parallel threads, then rejoin later. They sync which is when all threads continue.

4. Dependency:

A. Where can we find parallelism in programming?

Parallelism is a mechanism that enables programs to run faster by performing several computations at the same time by incorporating hardware with multiple CPU's. It is found in mobile phones, databases, servers, etc.

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

Dependency is when one operation depends on an earlier operation to complete and produce a result before first operation can be performed. There are four types of dependency: independent, true(flow) dependent, output dependent, and anti-dependency.

Independent example:

S1: a=1;

S2: b=1;

Statements are independent of each other.

True example:

S1: a=1;

S2: b=a;

Second is dependent on first.

Output example:

S1: a=f(x);

S2: a=b;

Second is dependent on first.

Anti-dependence:

S1: a=b;

S2: b=1;

First is dependent on second.

C. When is a statement dependent and when is it independent (provide two examples)?

A statement is dependent if its values are getting computed using previously assigned or computed values. Example:

a=10;

b=20;

c=a+b;

The statement c=a+b is dependent on the other two previous statements.

A statement is independent if its values are not dependent of previous values.

Example

a=10;

b=20;

c=45+65-90;

printf("hello")';

The last two statements are independent of the first two statements (a=10 and b=20).

D. When can two statements be executed in parallel?

Two statements can be executed in parallel if they don't share any common data item that is getting updated in two statements. If there is no dependence between both statements.

E. How can dependency be removed?

Dependency can be removed by rearranging statements or by eliminating statements.

F. How do we compute dependency for the following two loops and what type/s of dependency?

```
for (i=0; i<100; i++)
   S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
   S1: a[i] = i;
   S2: b[i] = 2*i;
}
```

The loops are DOALL loop (foreach loop). All iterations are independent of each other. All statements execute in parallel at the same time. Therefore, there is no dependency; the statements are independent in each loop.

Parallel Programming Basics

Part 1:

We created the file, trap-notworking.c, and wrote the code provided. When we compiled the program, we didn't import the math library, so there were errors.

```
pi@raspberrypi:~ $ gcc  trap-notworking.c -o trap-notworking -fopenmp
/tmp/ccwxMIhU.o: In function `f':
trap-notworking.c:(.text+0x17c): undefined reference to `sin'
collect2: error: ld returned 1 exit status
```

We searched online and realized we need to include -lm at the end of our compile command in order to use the sin function from the math library and trap-notworking compiled successfully.

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

We ran trap-notworking and it gave us an output of 1.421028. The correct output should have been 2.00.

```
pi@raspberrypi:~ $ ./trap-notworking 4
OMP defined, threadct = 4
With 1048576 trapezoids, our estimate of the integral from
0.000000 to 3.141593 is 1.421028
```

We followed the code provided and realized that integral is an accumulator variable, so we need to use "reduction: +integral". This will make sure the individual values of integral after each thread is finished will be added together. The i is declared as private so that each thread uses its own I variable, and a, n, and h were declared as shared so that all the threads could use them globally. We created a file, trap-working.c, and wrote the code like the lines in trap-notworking.c and made the necessary changes. When we compiled and ran trap-working, we got the correct output, 2.000000.

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

Part 2:

We created a file named barrier.c. Then, we created an executable program 'barrier' using gcc barrier.c -o barrier – fopenmp. We then proceed to run the program with ./barrier. We then proceeded to run the file and received the following output. We noticed that the same thread would output "before the barrier" and "after the barrier", but there was no clear divide.

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

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

We uncommented "#pragma omp barrier" at line 31. We recompiled the program, ran it, and received the following output. "#pragma omp barrier" makes sure the program waits until all the threads finished outputting "Thread # of # is BEFORE the barrier." before outputting "Thread # of # is AFTER the barrier."

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

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

Part 3:

We created a file named masterWorker.c. Then, we created an executable program 'masterWorker using gcc masterWorker.c -o masterWorker – fopenmp. We then proceed to run the program with ./ masterWorker. We then proceeded to run the file and received the following output.

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

Greetings from  the master, # 0 of 1 threads
```

We uncommented "#pragma omp parallel" at line 24. We recompiled the program, ran it, and received the following output. The master prints "Greetings from the master, #0 of 4 threads" and the rest of the worker threads executes the worker message.

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

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

ARM Assembly Programming:

Part 1:

The purpose of this ARM programming is to evaluate and understand how **compare** and **branch(jump)** conditions work in ARM assembly using conditional statement such as **if else**. To achieve this, we started the assembly programming by opening a terminal, then opened a fourth.s file to write our program. We used the directive, .data, to indicate that we were going to declare some variables. We created a 2-word memory sizes at location **x** initialized with 0 and at location **y** initialized with 0. We then loaded the registers with memory variables x and y.

```
.section .data
x: .word 0
y: .word 0
.section .text
.globl _start
_start:
    ldr r1,=x
    ldr r1,[r1]
    cmp r1,#0
    beq thenpart
    b endofif
thenpart:
    mov r2,#1
    ldr r3,=y
    ldr r2,[r3]
endofif:
    mov r7,#1
svc #0
.end
```

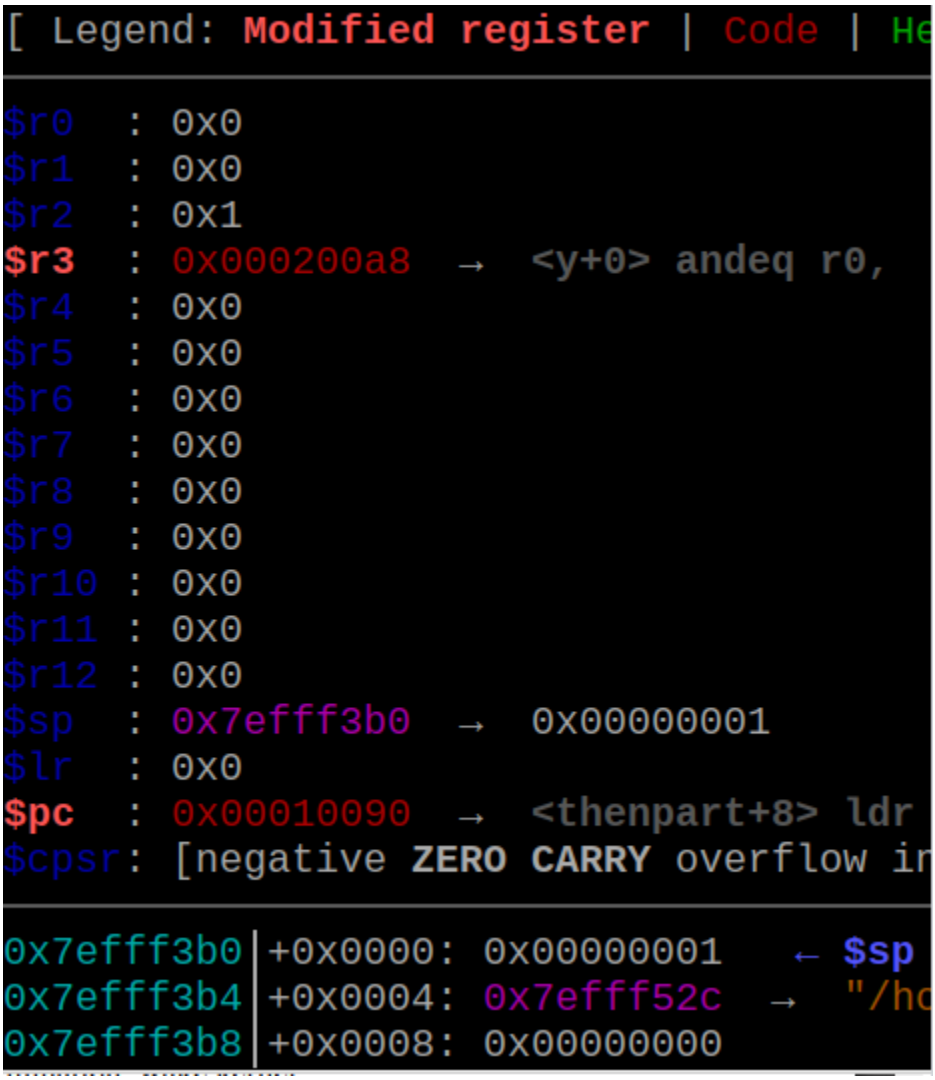
We assembled and linked the program, then launched the gdb debugger for fourth.

We ran the **list** command to make sure our program was correct.

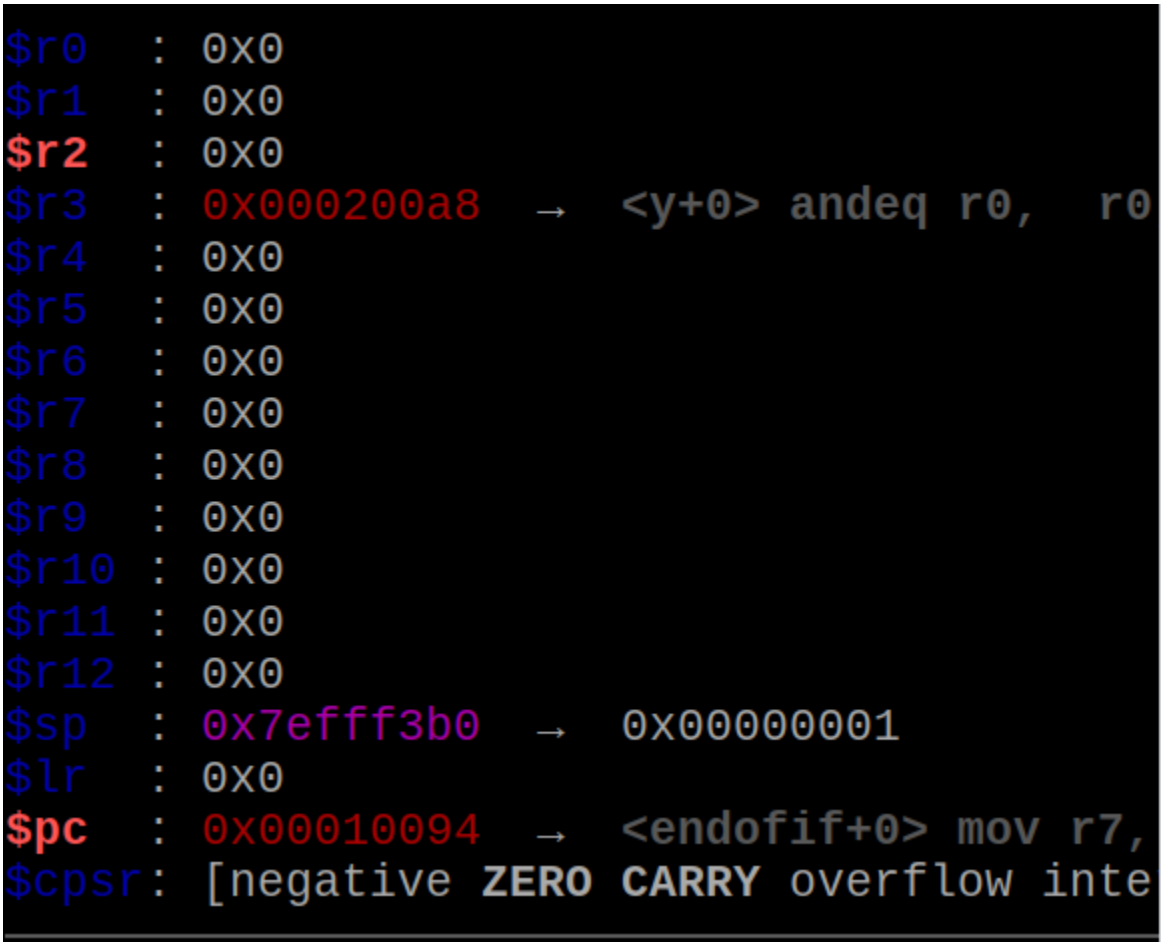
```
gef➤  list
1         .section .data
2        x: .word 0
3        y: .word 0
4        .section .text
5        .globl _start
6        _start:
7            ldr r1,=x
8            ldr r1,[r1]
9            cmp r1,#0
10           beq thenpart
gef➤
```

We set a breakpoint at line 7 using the syntex **b 7** and ran the program using **run**. After the program at line 7 was successfully executed, we started stepping over ever line to execute them using **stepi.** Since our goal is to understand how the compare and the jump conditions work, at each stepi we go in the register to evaluate them and check on the flags. After the execution of lines 7 and 8, we automatically knew what the code would do at the level; however, after the executon of line 9 which is comparing the value in r1 to 0, we noticed that the **zero flag(ZF=1)** was set. According to our understanding of comparison and flags, the zero flag is always set when the destination is equal to the source, which in this case **r1** is the destination and the immediate **0** is the source (**0=0).**

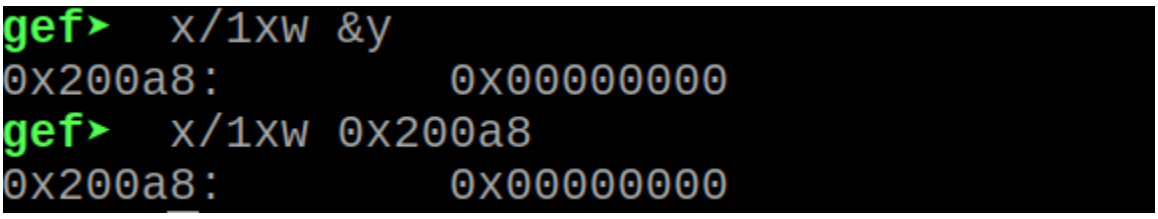
The next line executed is the **beq thenpart** which indicates that we will jump to the statement **thenpart** if the zero flag is equal to 1, in case we do jump to the thenpart because the zero flag is indeed equal to 1; therefore, the next line to be executed when we stepi is line 13 which indicated moving 1 into r2, followed by loading memory address of y into r3.



The next instruction loads r2 value into y memory address, then the code exits out after the execution of **endofif.**



We then examined the content of y memory location using **x/1xw 0x000200a8**



Part 2:

The objective of part 2 is to make the program in part 1 more efficient. We achieved this by eliminating the back-to-back branches (beq followed by b). We replaced **beq** with **bne (**branch on not equal (Z==0)) then we deleted the b instruction from the code as such that we jump to the endofif to terminate the code if zero flag is not equal to 1; else, we execute the immediate next instruction.

```
.section .data
x: .word 0
y: .word 0
.section .text
.globl _start
_start:
    ldr r1,=x
    ldr r1,[r1]
    cmp r1,#0
    bne endofif
    mov r2,#1
    ldr r3,=y
    ldr r2,[r3]
endofif:
    mov r7,#1
svc #0
.end
```

We assembled, linked, ran and evaluated the zero flag. The ZF is still set

```
$cpsr: [negative ZERO CARRY overflow ir
```

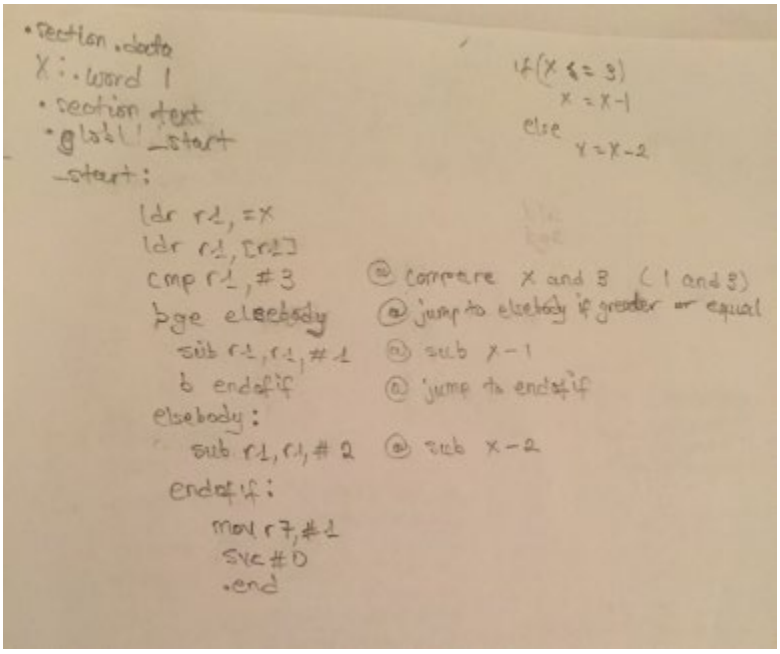
The code produced the same output as the code in part 1 because they are the same code, however, part 2's code is more efficient due to the avoidance of the back-to-back branches.

Part 3:

Using part 1's program as a guidance, we created a new program called ControlStructure1.*s*. We first planned what our program should do using the high-level programming code given as

if X <= 3
X = X – 1
else
X = X – 2 with X initialized with 1. The interpretation of the above code according to our understanding is that "if the variable x which is initialized with 1 is less than or equal to 3, then the next statement x = x – 1 will be executed, if not then x = x – 2 will be executed.

On a piece of paper, we illustrated what the ARM assembly version of the program should look like.

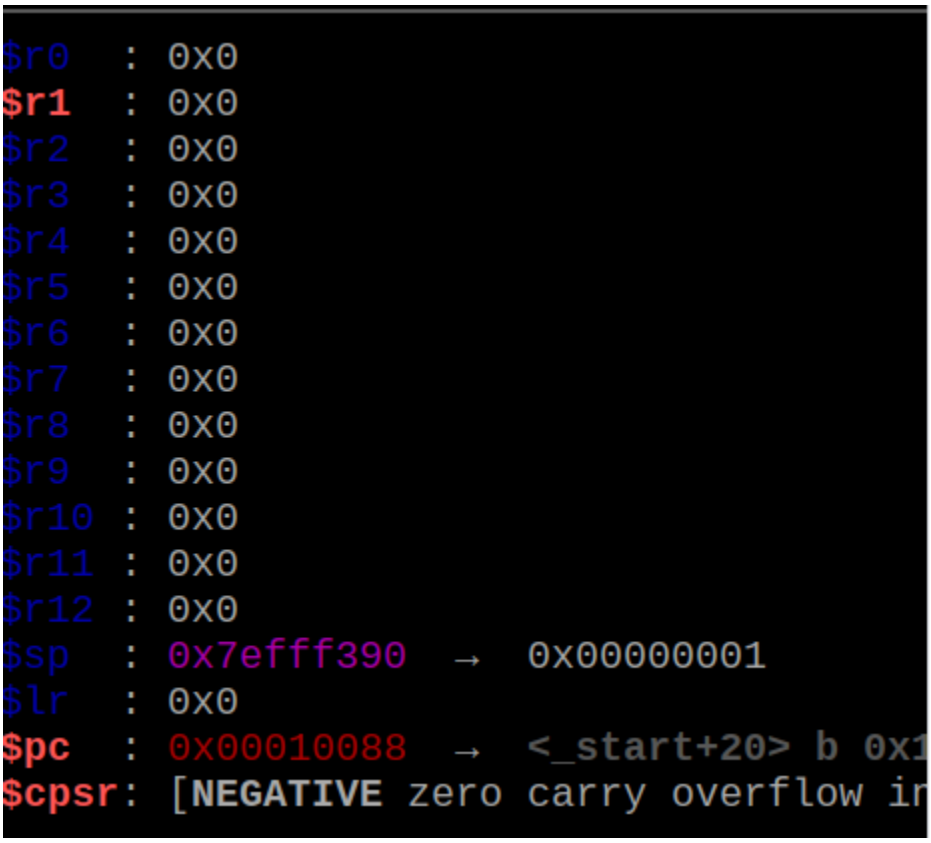


To determine the output of the program, we first opened a ControlStructure1.s file to write our program. We used the directive, **.data**, to indicate that we were going to declare some variables. We created a word memory size(32-bits) at variable location X which was initialized with 1.
To use variable **X**, we had to load (**ldr**) the memory address of **X** into register r1. Then, we had to load the value of **X** (**[r1]**) into register r1. After we loaded the variable, we compared **3** to the value of **r1**. From here forward, the output of the whole program is based(dependent) on the outcome of the comparison of r1 and 3, which also generates the position of the zero flag.

```
.section .data
x: .word 1
.section .text
.globl _start
_start:
    ldr r1,=x
    ldr r1,[r1]
    cmp r1,#3
    bge elsebody
    sub r1,r1,#1
    b endofif
elsebody:
    sub r1,r1,#2
endofif:
    mov r7,#1
    svc #0
.end
```

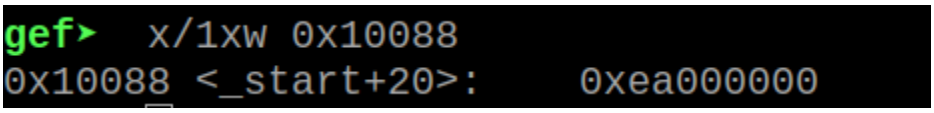
After the code, we assembled, linked and debugged using **(gdb ControlStructure1)**. We listed the program to make sure that everything was correct, then we set a breakpoint at 6 **(b 6)** and ran the program. We then started stepping over to execute the code line by line using **stepi.** In our program, we used De Morgan's law to determine how to translate the if statement from the high-level code to assembly code. So in our case, we jump to the else statement if the source(3) is greater than or equal to the destination(r1) **bge elsebody.** After the execution of the comparison statement, the next statement executed was the subtration of 1 from r1 **(sub r1,r1,#1)**. The is because the 1 is less than 3 so the program did not jump to else statement. After the execution of r1 – 1 **(1 – 1 = 0)**, we jumped to the end of the if statement **b endofif** to exit the program because we didn't need to execute the else statement.

We then examined the content of the register (r1) and the zero flag to make sure that the outputs are correct.



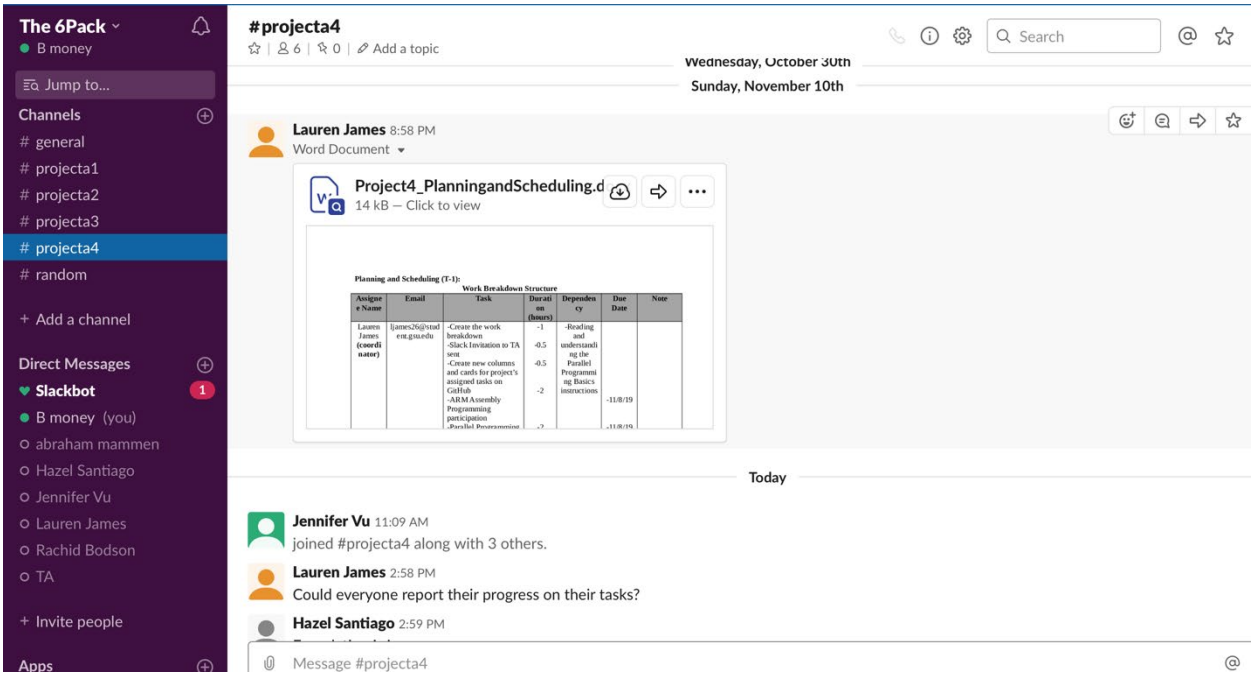
We noticed that the zero flag is not set. This is because according to our understanding of comparison, the ZF is only set when the destination is equal to the source; however, in this case the destination is less than the source. More so, we noticed that the negative flag (sign flag) is set. To our knowledge, to compare is to subtract; therefore when we compared r1 to 3, it meant that we subtracted 3 from 1**(1-3)** which resulted as a negative 2 (**-2)**, thus the set of the negative flag.

We also verified the result in the memories using the memory location at **x/1xw 0x10088**.

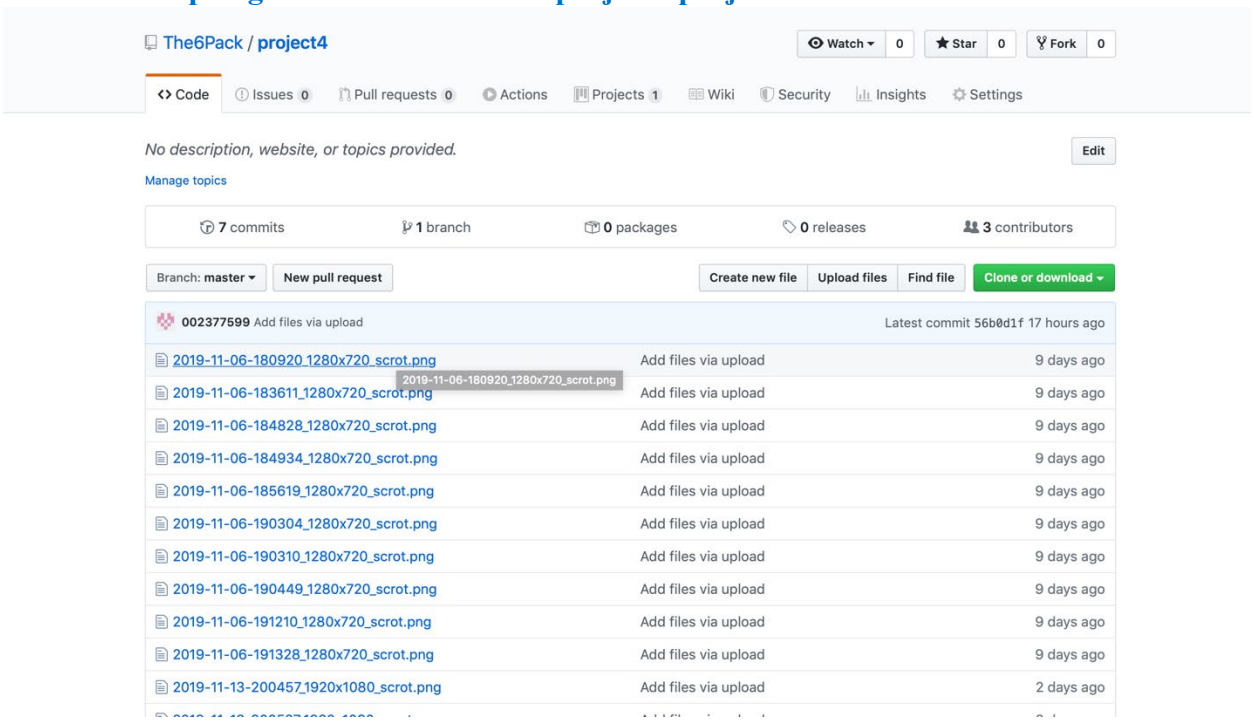


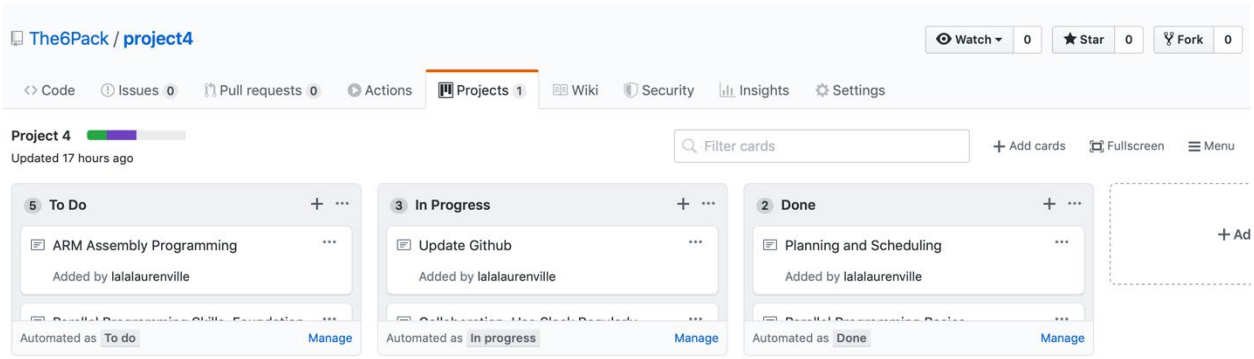
Appendix:

SLACK: https://app.slack.com/client/TN9D41Q1K/CPN499CFM



GITHUB: https://github.com/The6Pack/project4/projects/1





Youtube: https://www.youtube.com/channel/UC7dMmLnGrmjZv3d8c-39qJg?view_as=subscriber