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

Fall 2019

Team CHABU

Chris Lavey, Harsh Jivani, Anggiela Yupanqui, Binh Ha Duy Le, Ugonma Nnakwe

[**1. Planning and Scheduling**](#_4ixas4gqls6d) **3**

[**2. Parallel Programming Skills**](#_7mmtsc3wz2sj) **3**

[2.1 Foundation](#_pj19uou404kp) 3

[2.2. Parallel Programming Basics](#_yr01k5kuc54p) 5

[3. ARM Assembly Programming](#_qs0tk1gdy6pg) 7

[3.1. Part 1](#_oqj6dtuldec8) 7

[3.2. Part 2](#_ywsj0q3g0rup) 9

[**4. Appendix**](#_vbm5rcidppqy) **10**

[4.1. Github](#_ornadumcexj8) 10

[4.2. Slack](#_5fylmpzc45s8) 11

[4.3. Youtube (Channel)](#_c7hgrch2k0cb) 11

[4.4. Youtube (Video)](#_ieitlfwhje8z) 11

# 1. Planning and Scheduling

**Work Breakdown**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Assignee Name | Email | Task | Duration (Hours) | Dependency | Due Date | Note |
| Harsh Jivani | hjivani1@student.gsu.edu | Slack,  Parallel Programming | 2 hours | None | 09/30/19 |  |
| Chris Lavey | clavey@student.gsu.edu | Technical writing (getting the report ready) as described in the assignment, Parallel Programming | 5 hours | Slack, Github,  Parallel Programming, ARM Assembly Programming | 10/03/19 |  |
| Anggiela Yupanqui (coordinator) | ayupanquirojas1@student.gsu.edu | GitHub,  ARM Assembly Programming | 2 hours | None | 09/30/19 |  |
| Ugonma Nnakwe | unnakwe1@student.gsu.edu | Video editing,  ARM Assembly Programming | 5 hours | Video recording | 10/03/19 |  |
| Binh Le | ble8@student.gsu.edu | Parallel Programming | 2 hours | None | 09/30/19 |  |

# 

# 

# 2. Parallel Programming Skills

## 2.1 Foundation

**2.1.1. Components of Raspberry PI B+**

**2.1.1.1. CPU**

The central processing unit acts as the brain of the unit, and is involved in

all tasks that take input, perform calculations, and produce output. The CPU is a Broadcom BCM2837B0, which is 64-bit and has a clock speed of 1.4GHz.

**2.1.1.2. GPU**

The graphics processing unit handles output of graphics to a display.

**2.1.1.3. 40-Pin Header**

These are the exposed general-purpose input/output connection pins that are used to connect external hardware components.

**2.1.1.4. 3.55mm Audio/Composite Output Jack**

This is used to connect an external sound device to the Raspberry PI, such as speakers. The Raspberry PI is not capable of receiving audio input.

**2.1.1.5. 2x2 USB 2.0 Ports**

These are used to connect peripherals such as a mouse and keyboard. If necessary, a USB hub can be used to connect more outside peripherals.

**2.1.1.6. HDMI Port**

This is used to connect to an outside display via a standard HDMI cable.

**2.1.1.7. Micro USB Port**

This is where you plug in the power cable for the PI. The PI comes with a

5V, 2.5A micro USB power connector.

**2.1.1.8. MIPI CSI Port**

This allows a small camera to be connected to the CPU.

**2.1.1.9. MIPI DSI Port**

This is used by an integrated circuit to feed graphics data to the display

**2.1.1.10. Micro SD Card Slot**

This is where a Micro SD Card is inserted. The card acts as storage for

the PI. The PI will not boot if the card doesn’t have an OS installed.

**2.1.1.11. Ethernet Port**

This is used to connect the PI to a wired network via an Ethernet cable. It is capable of having Internet speeds of up to 300Mbps.

**2.1.1.12. PoE Pin Header and PoE HAT**

This allows the system to be powered by an Ethernet cable.

**2.1.2. Number of Cores on Raspberry PI B+**

All Raspberry Pi 3’s have a quad-core CPU. The specific type and clock speed is

listed above.

**2.1.3. X86 (CISC) vs. ARM (RISC)**

**2.1.3.1. Instruction Set**

X86 processors are called Complex Instruction Set Computing processors, which means that the instruction set allows for more complex instructions to access memory. This means a X86 processor will be less dependent on registers, as it will be able to much more easily access external storage.

**2.1.3.2. Speed of Calculation**

Due to the instruction set being more complex, X86 processors generally require more clock cycles to perform calculations when compared to ARM processors.

**2.1.3.3. Writing Software**

Due to the instruction set being less complex, coding for ARM processors puts a larger strain on the programmer to write as efficiently as possible when compared to coding for X86 processors.

**2.1.4. Sequential vs. Parallel**

During parallel computation, the CPU distributes a single process to each core to

handle multiple computations at once. During sequential computation, the CPU handles each task individually and computes them sequentially. Sequential programs tend to be easier to program, and for simple programs or tasks that don’t require high efficiency it should be fine. When a program is designed to handle large amounts of data or needs to perform many calculations quickly and efficiently, parallel programming is a much better option.

**2.1.5. Data and Task Parallelism**

The general form of a program that’s designed to be run in parallel must accomplish three things. It must break apart the tasks so that they can be executed simultaneously. It must be able to multitask at any given time. Lastly, it must be proficient because it utilizes multiple computer resources rather than a single one.

**2.1.6. Processes vs. Threads**

A process is the act of running a program, while a thread is how a program is divided into smaller parts in order to be executed.

**2.1.7. OpenMP**

OpenMP is a guideline used by compilers. It is comprised of pragmas which act as directives for the compiler to generate threaded code.

**2.1.8. Applications of Multi-core**

Many applications benefit from using multi-core CPUs, including but not limited to database servers, web servers, multimedia applications, and scientific applications such as CAD and CAM.

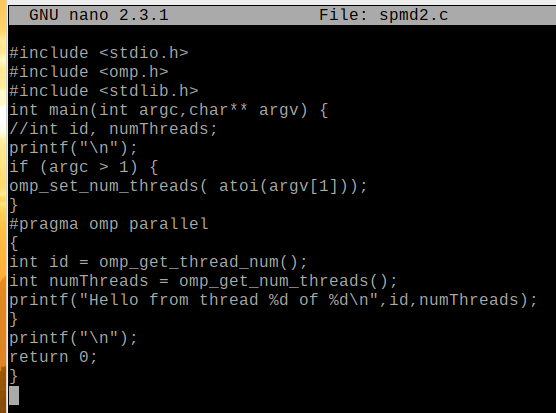
**2.1.9. Why Multi-core?**

Multi-core processors allow for much higher clock speeds compared to single-core processors. Multi-core processors have deeply pipelined circuits which are difficult to design and can have heat problems if not cooled properly. Many applications are multithreaded to increase efficiency, and there has been a general trend in computer architecture towards more parallelism.

## 2.2. Parallel Programming Basics

For the parallel programming portion of this project, we were tasked with editing

a C program that prints a string followed by a number identifying them from a group of four threads. Below is the edited code:



Prior to revision, line 5 was not commented out, and within **#pragma omp parallel** *id* and *numThreads* were not declared as **int** data types. When compiled and ran, the program printed out something similar to this:

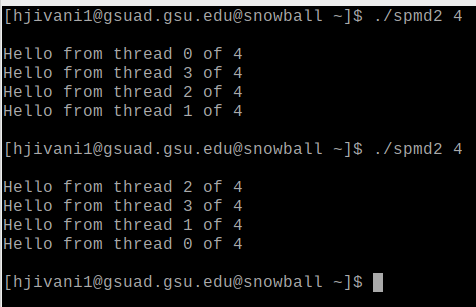
**Hello from thread** **1 of 4**

**Hello from thread 2 of 4**

**Hello from thread 1 of 4**

**Hello from thread 2 of 4**

This was a problem, as each thread ID should have been unique. The error with this program was that the variable we were wishing to access was declared outside the parallel function, meaning that all the threads shared those variables. To fix this issue, we commented out the initial declaration in line 5 and declared *id* and *numThreads* as the **int** data type. Once we saved, compiled, and ran our program, this was the resulting output:



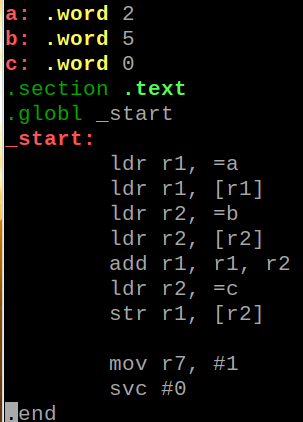
This output was what we expected from a multithreaded program. Each string was printed via a unique thread, and there were no duplicates. This is because in the revised program, the *id* and *numThreads* were declared for each thread, meaning that each thread had a unique *id*.

## 3. ARM Assembly Programming

## 3.1. Part 1

In part 1 of the ARM Assembly Programming portion of this project, we were

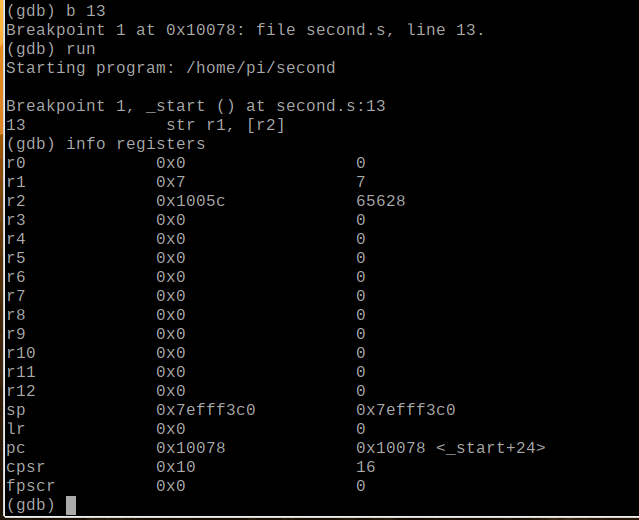
tasked with writing a program **second.s** based on a given program and examining the output. Below is a screenshot of the code:



This program begins by loading **r1** with the memory address of **a** and then loading **a** into **r1**, resulting in the value of **r1** becoming 2. It then repeats this process with **r2** and **b**, resulting in **r2** having a value of 5. It then adds **r1** to **r2** and stores the output to **r1**, meaning the value stored in **r1** was now 7. It then loads the memory address of **c** into **r2**. It finishes by storing the value of **r1**, 5, into **c**, making the value stored in **c** 7.

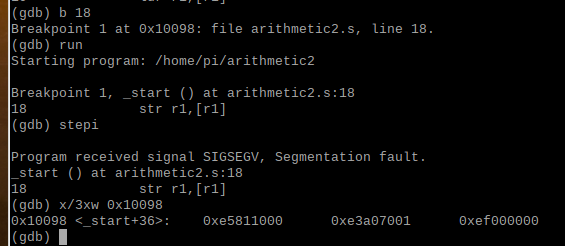
When we assemble, link, and run this program, there is no output to the terminal as there was never an instruction to print the result to the console. To see the values of the registers at the end of our program, we had to debug our program in a similar manner to our program in Project 1.

Once we had our program in the debugger, we set a breakpoint at line 13, as this was the last line before the program terminated. Below is the output of **info registers**:



From this screen, we can tell that the final value of **r1** is 7 like we intended, but

we have no way of identifying whether the final value of **c** is valid from this screen. To get information about the memory at the breakpoint, we use the command **x** followed by modifiers to determine the type and the address of the line we’re interested in, which in this case is **0x10078**.After using the command **x/3xw**0x**10078** in the debugger, this is the output:

******

By using this command, we see see the values of the first three words at that address in hexadecimal format. We are interested in **c**, so that would be the third value. By converting this number from hexadecimal to decimal, we see that the value stored in **c** is 7.

## 3.2. Part 2

For the second portion of ARM Assembly programming for this project, we were

tasked with writing a new program, based on what we learned from **second.s**, to produce the following output:

**Register = val2 + val3 +9 - val1**

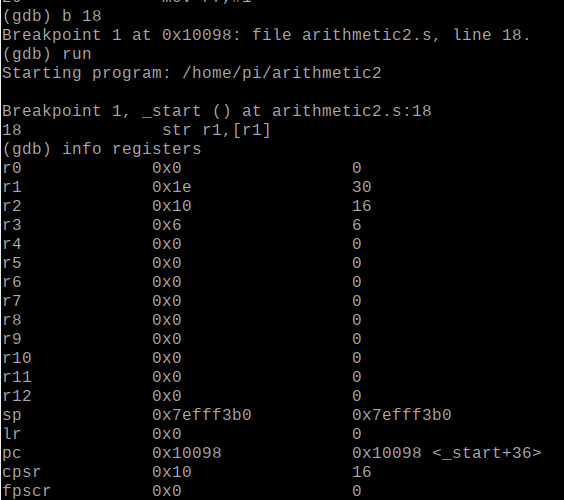
To do this, we use the following program:

# 

This program begins by declaring three variables in the **.data** section: **val1**, **val2**,

and **val3**. They are all of the **.word** data type, with **val1** having a value of 6, **val2** having a value of 11, and **val3** having a value of 16, as specified by the documents. We start the program itself by loading **val2** into **r1** in a similar manner to part 1 of this exercise. We load **r2** with **val3**, and then we add **r1** to **r2** and store the value in **r1**, making the current value of **r1** 27. We then add 9 to **r1** and store this value in **r1**, making the value of **r1** 36. We then load **r3** with the value of **val1**, then we subtract **r3** from **r1** and store the result in **r1**, making the final value of **r1** 30.

After assembling and linking the program, we ran it, and as expected there was no output to the console. We then proceeded to debug **arithmetic2.s** using the GDB debugger. Once the program was in the debugger, we set a breakpoint at line 21 and used the command **info registers** to display register values. This was the result:



This showed that the values stored within **r1**, **r2**, and **r3** were as we expected, and as such we were done with the programming portion of this assignment.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

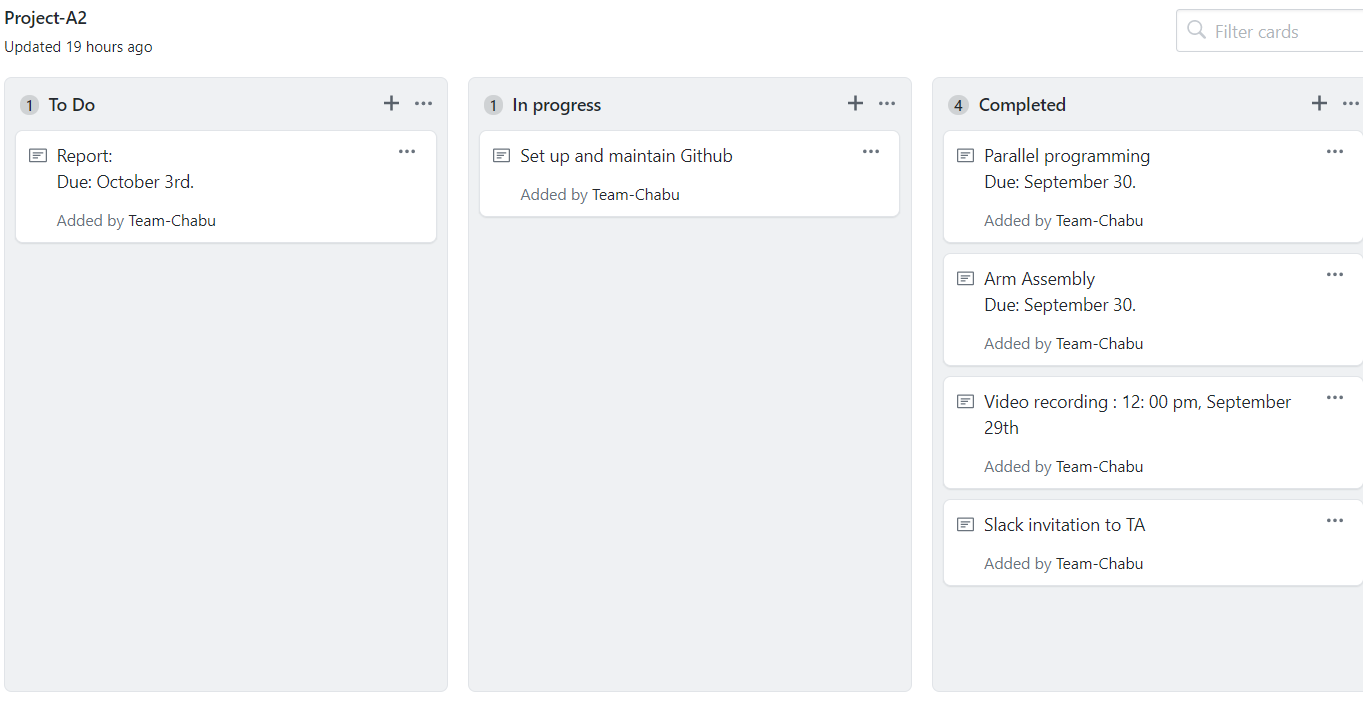
# 4. Appendix

## 4.1. Github

**4.1.1. Link:**

<https://github.com/Team-Chabu/CSC3210-CHABU>

**4.1.2. Screenshot:**

****

## 4.2. Slack

<https://join.slack.com/t/csc-3210-teamchabu/shared_invite/enQtNzc0ODEwMjkyNjU2LTU4MGQ2MzcyMDg5YTJhMjc4ZmZjZWU4OTgzNjk4NzI1M2ZjMjY5YjBiMjYyMTgyOGVmNGQzMjFhMjYzN2YxOWM>

## 4.3. Youtube (Channel)

<https://www.youtube.com/channel/UCp-0kVA08dU4F79t6slHtLQ>

## 4.4. Youtube (Video)

<https://youtu.be/rdZswfgQoh4>