Developing Soft and Parallel Programing Skills

Using Project – Based Learning

Spring 2020

8 – Bit

Akash Dansinghani

Landon Wang

Raejae Sandy

Tony Ngo

**Work Breakout Structure**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Assignee Name | Email  (studentID@student.gsu.edu) | Task | Duration (Hours) | Dependency | Due Date | Note |
| Akash Dansinghani | adansinghani1 | Gathered and assembled the reports | 2 hours on task,  5 hours on other project tasks | Everything is uploaded to Github before due time | 2/21/20 | Be ready 5 hours before due date |
| Landon Wang | lwang51 | Helped with ARM Assembling program and explaining parallel programming | 3 hours on task,  6 hours on other project tasks | None | 2/21/20 | Make sure everybody has their code running and working and knows why it works |
| Raejae Sandy | Rsandy2 | Set up GitHub branch repository | 4 hours on task, 5 hours on other project tasks | None | 2/21/20 | Send everyone the Git Hub Branch information |
| Tony Ngo | tngo23 | Group coordinator, uploaded to YouTube Channel, recorded and edited the video | 3 hours on task,  5 hours on other project tasks | None | 2/21/20 | Sent everyone the YouTube link to confirm |

**Parallel Programing Skills**

By: Akash Dansinghani

**Identifying the components on the Raspberry PI B+**

The Raspberry Pi B+ has a Quad-Core Multicore CPU with of 1GB Ram which allows the Pi to run process more efficiently and faster since it doesn’t have to share memory with other programs or functions that are performed on the Pi. Executing more processes at one time (concurrency) increases throughput of the system. The inside also holds space for an SD card where the operating system of Pi (Raspbian) is located. Externally the Pi houses two Micro USB ports where one is for power and the other is to connect it to an external display, a HDMI port to display the screen of the Pi on an external display, and lastly an auxiliary or audio port that can play any sound. On the right side of the Pi houses an Ethernet port for direct LAN connectivity in lieu of using Wi-Fi and four USB ports which we use one to connect the wireless keyboard.

**How many cores does the Raspberry Pi have?**

As previously mentioned, has a quad-core multicore CPU with 1GB of Ram which means it has 4 cores.

**Difference between ARM and x86**

The Raspberry Pi has a Quad-Core Multicore. The difference between ARM and x86 are their architectures. ARM has RISC or Reduced Instruction Set Computer architecture which allows the microprocessor to execute codes in less cycles than Complex Instruction Set Computer which is the architectures x86 uses. As the name suggests, RISC is more optimized so its instruction set is more generalized than CISC. Also, x86 uses little endian which the processor reads data in the memory right to left as opposed to Big Endian which ARM uses where it reads data left to right. Lastly, RISC uses load and store architecture which has access to memory which deals with loading and storing in the memory and registers and the ALU which deals with only registers.

**Sequential Computations vs. Parallel Computations**

Sequential computations have processes that execute in a sequence of processors while in parallel it happens concurrently which means that it can produce more processes at one time. In layman’s terms, with sequential processing, a program will run with the CPU and produce a result. After the result is produced, it is allowed to process another program. On the contrary, parallel processing is allowed to run a program that has multiple tasks and produce the result in the order the tasks are meant to run.

**Identify the basic form of data and task parallelism in computational problem**

Data parallelism deals with processing data where it is divided between processors which makes it superior to the latter task parallelism. Task parallelism instead divides the task to produce the result rather than the data between processors to produce a result (which is more efficient).

**Threads vs. Processes**

Threads run in a memory space that is shared while processes run in separate memory spaces. A process is able to run one process at a time with a single core processor, but if the CPU has more core, it can work with more processes. Threads run single a process and is broken down into smaller parts.

**OpenMP vs. OpenMP Pragmas**

OpenMP is a set of complier directives that works for programs such as C, C++, or FOTRAN and executes code in parallel and uses multithreading. OpenMP Pragmas are a set of complier directives that generate threaded code.

**Applications that benefit using Multi-Core applications**

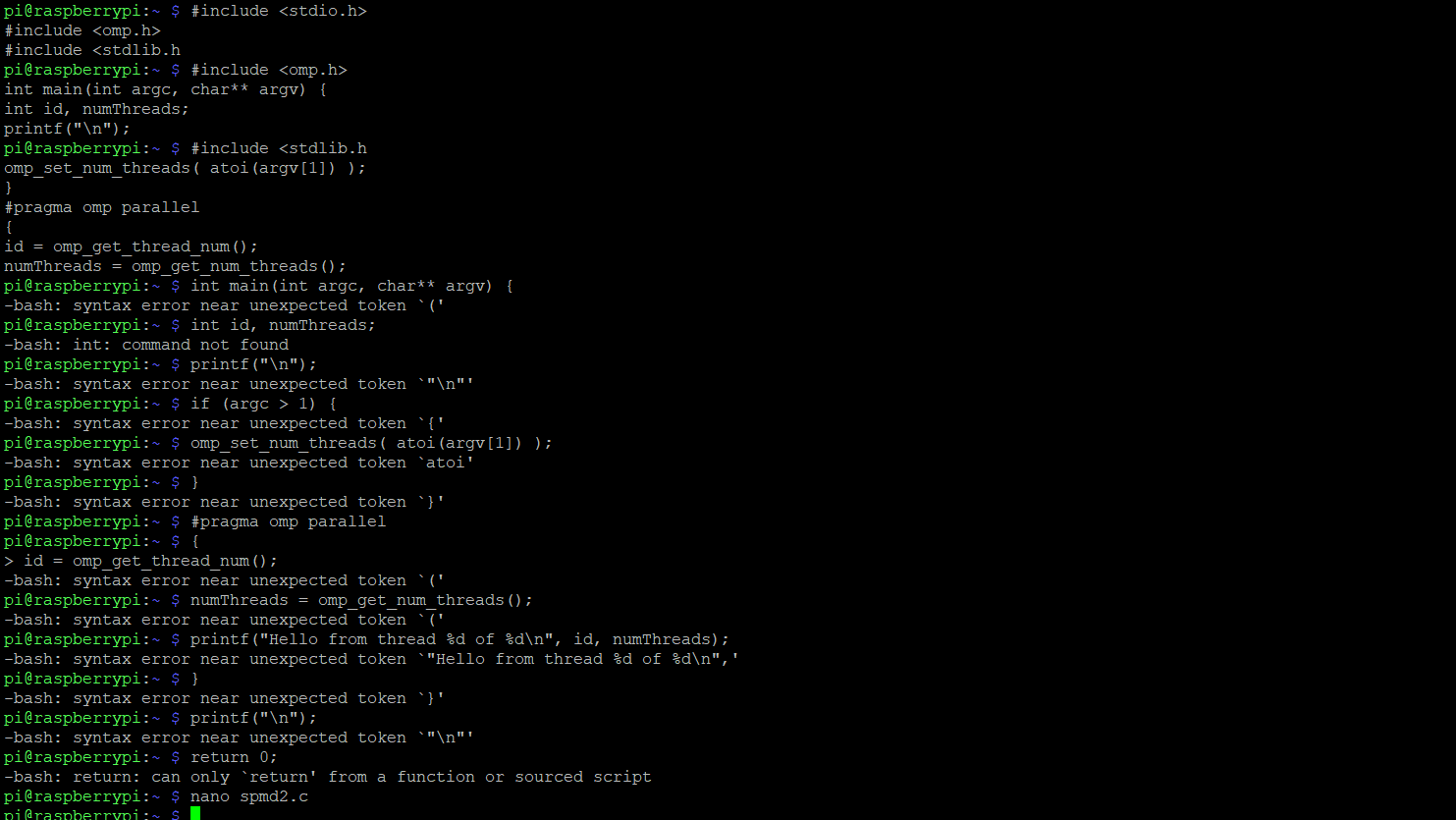
Applications that benefit from multi-core are ones that implement thread-level parallelism such as database servers (MySQL, PostgreSQL, Microsoft SQL Server, Oracle, SAP and DB2), webservers (Apache, Microsoft's Internet Information Server (IIS) and Nginx), multimedia applications (text, graphics, images, sound and audio, animation and video), and scientific applications (CAD/CAM).

**Why use Multi-Core vs. Single Core**

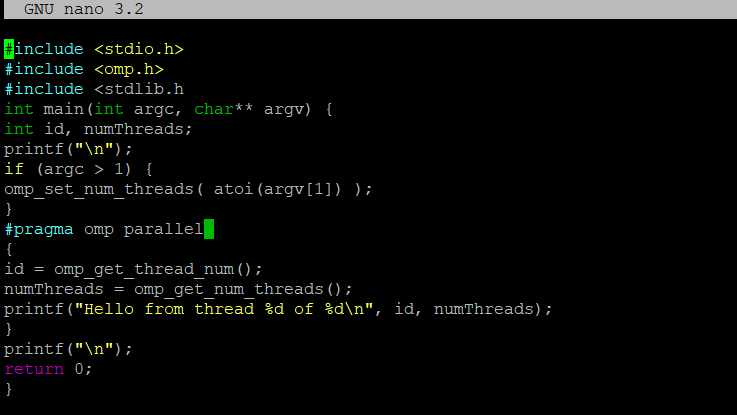
Most applications will benefit using multicore since most applications that developers use for example use thread-level parallel processing. Multicore processors can have multicores handling one process and it can work with different data at one time. Multicores is also more efficient on power. Single core has a single L1 cache as well as a L2 cache while multicores have a shared L2 cache and a L1 cache as well.

**Getting Started with Raspberry Pi and Parallel Programing**

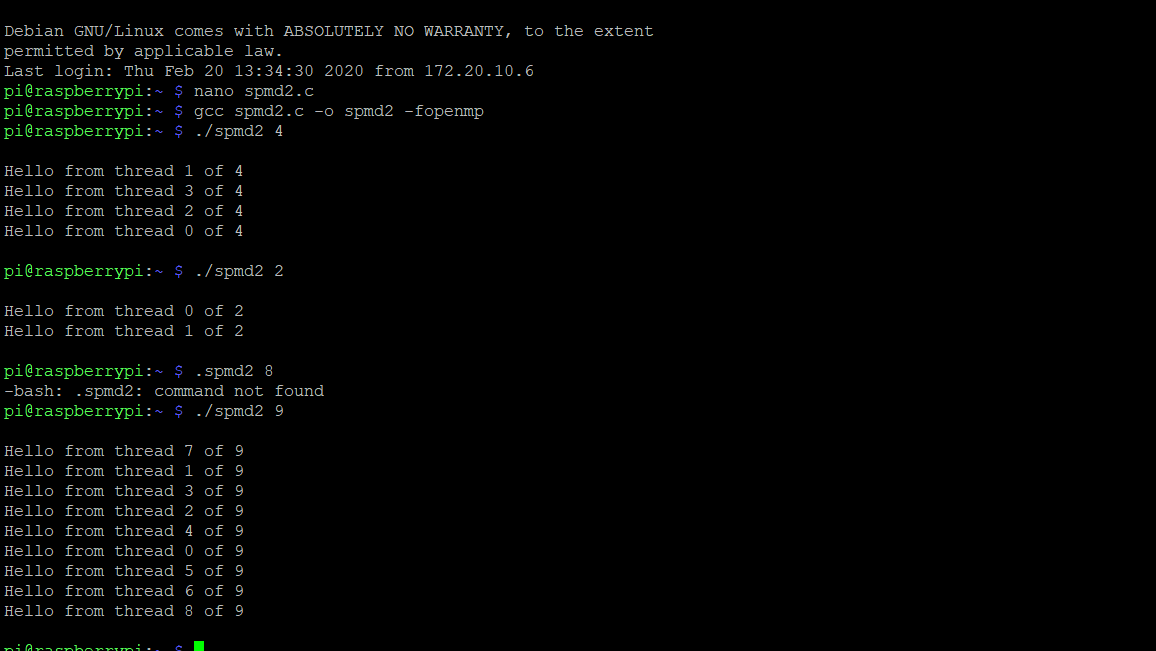
By: Akash Dansinghani



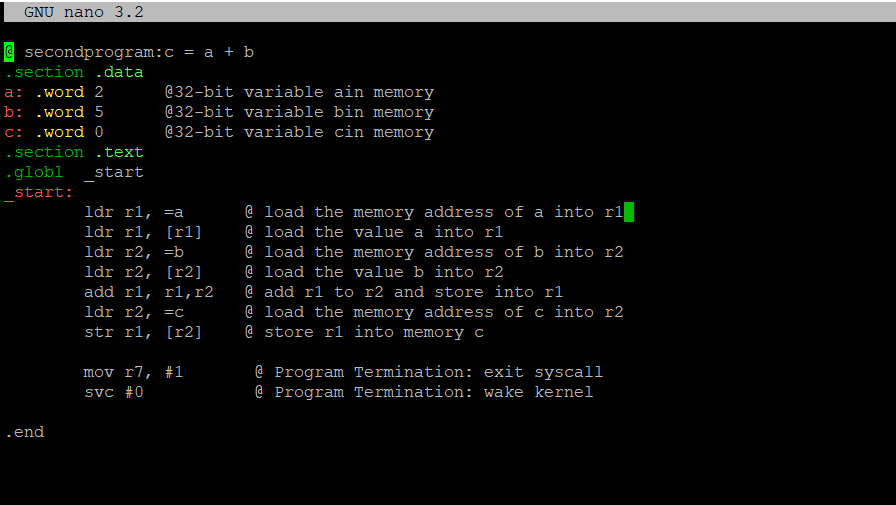
This is when I copied the code into the terminal. I proceeded to get a few errors and copied the code into the GNU.



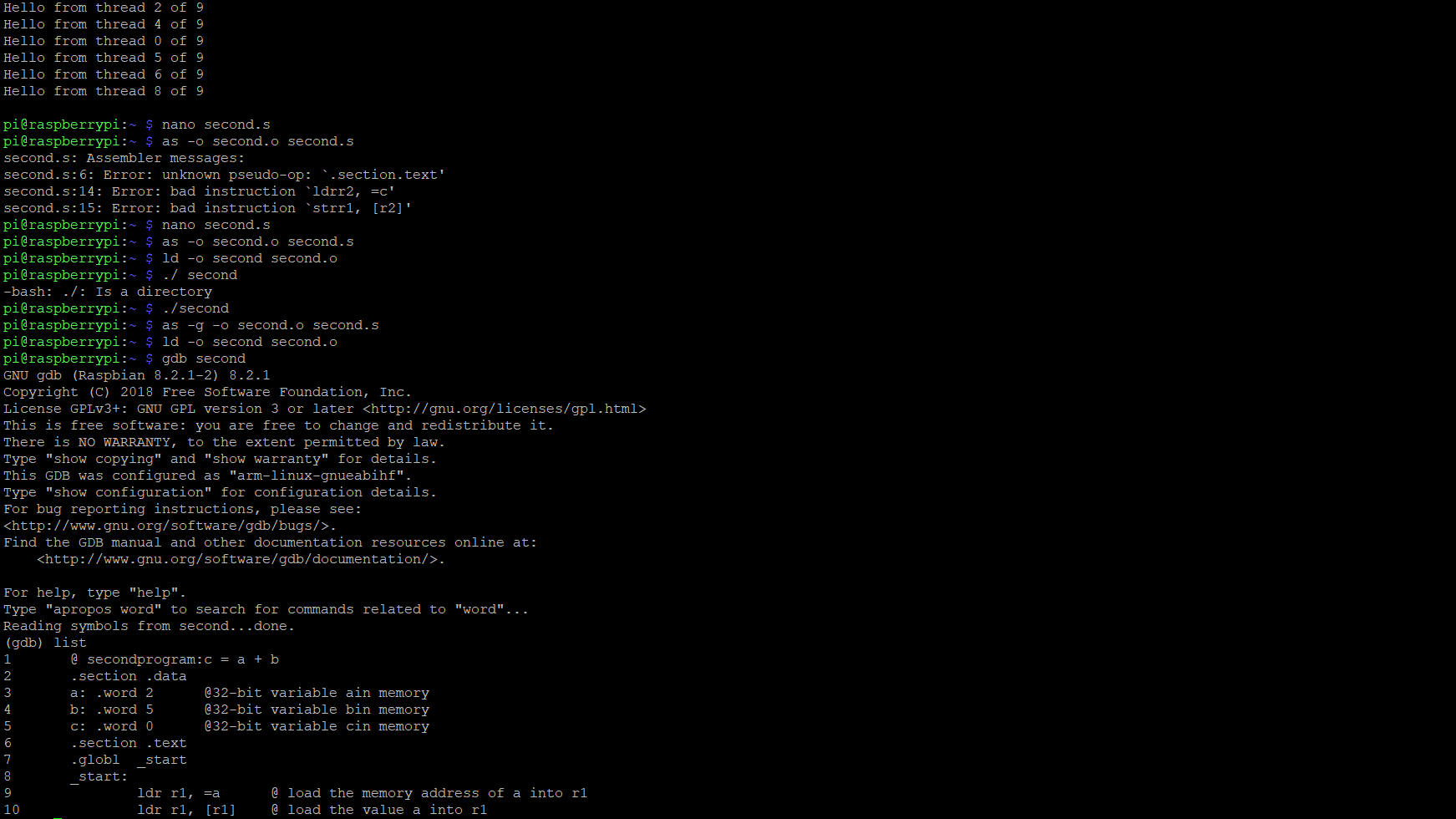
The code was put into the GNU. It was not able to run because of two things: 1. Line 3 needs a > after the “h” in the code. 2. Line 12 and 13 has id = class and numThreads = class respectively, but haven’t been initialized so putting int in front of id and numThreads will allow the code to finally run (int id = cmp\_get\_thread\_num(); and int numThreads = cmp\_get\_num\_threads();)



After the issues were fixed, I was able to run the code. After I copied to code into the nano as you can see above, I used the code “nano spmd2.c”. To be able to run the file, I used the code gcc spmd2.c -o spmd2 -fopenmp and after that, I ran the code using “./spmd2 and added 4 after that to execute 4 threads. After I ran 4 threads, I wanted to see how many tries it would take to get a sequence of threads in order with two threads and I got it in one try. I also tried to do 9 threads to see how the threads would be ordered and as you can see it was random and threads start from 0 and are out of 9 but go to 8.



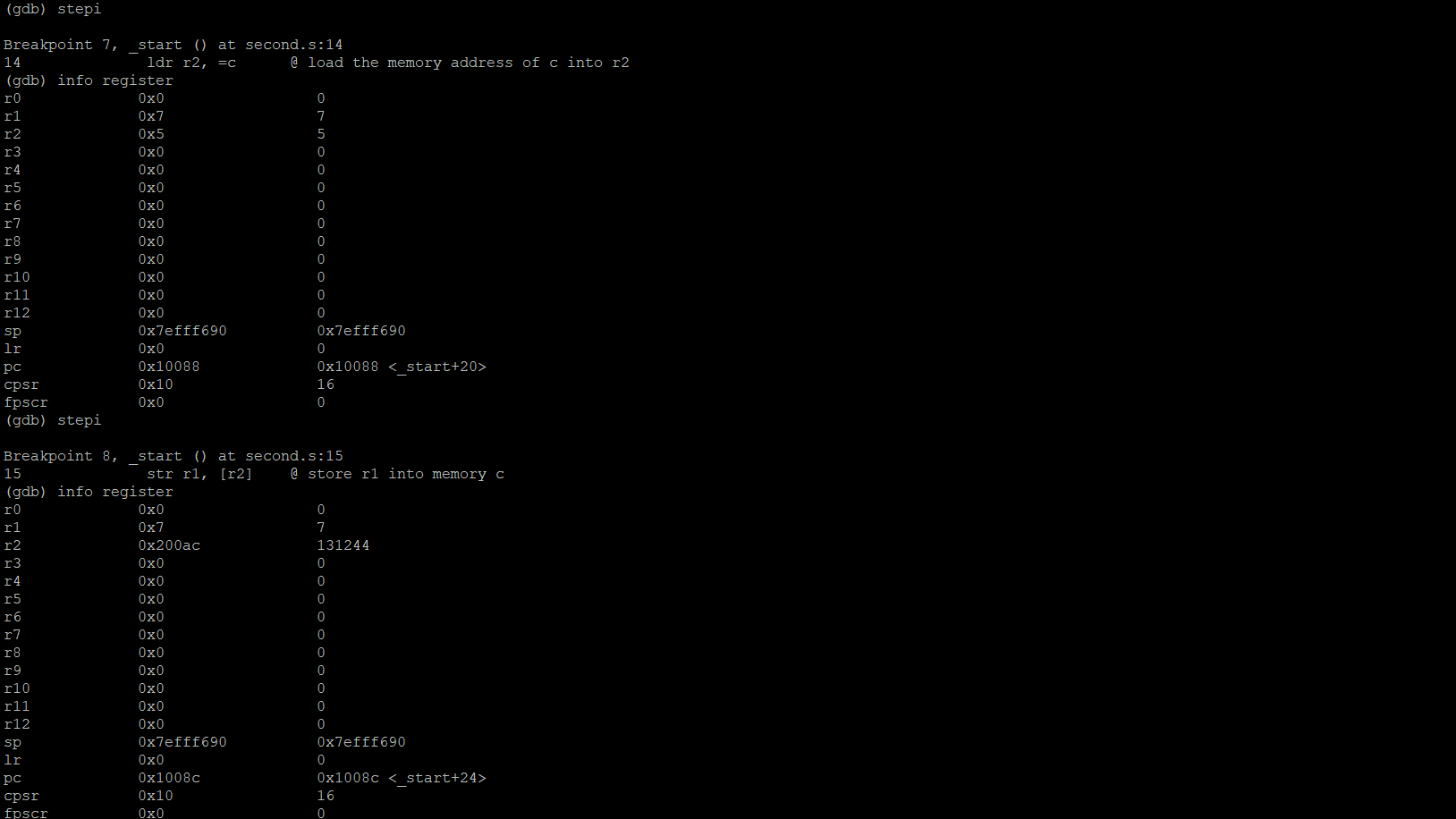
Here’s my program I called second.s and tried to add two values placed in a and b and equate it to c and store it into that memory.

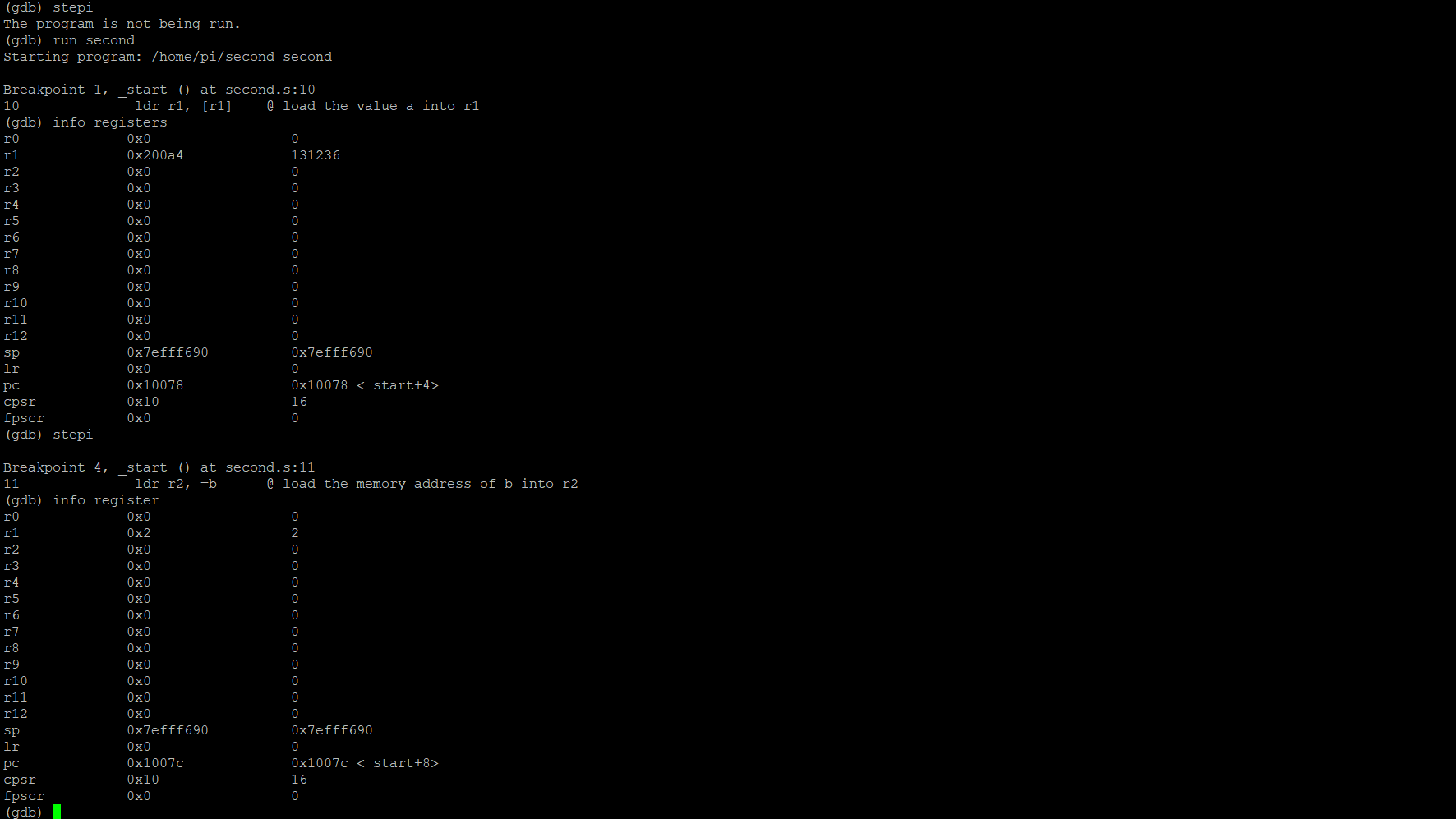


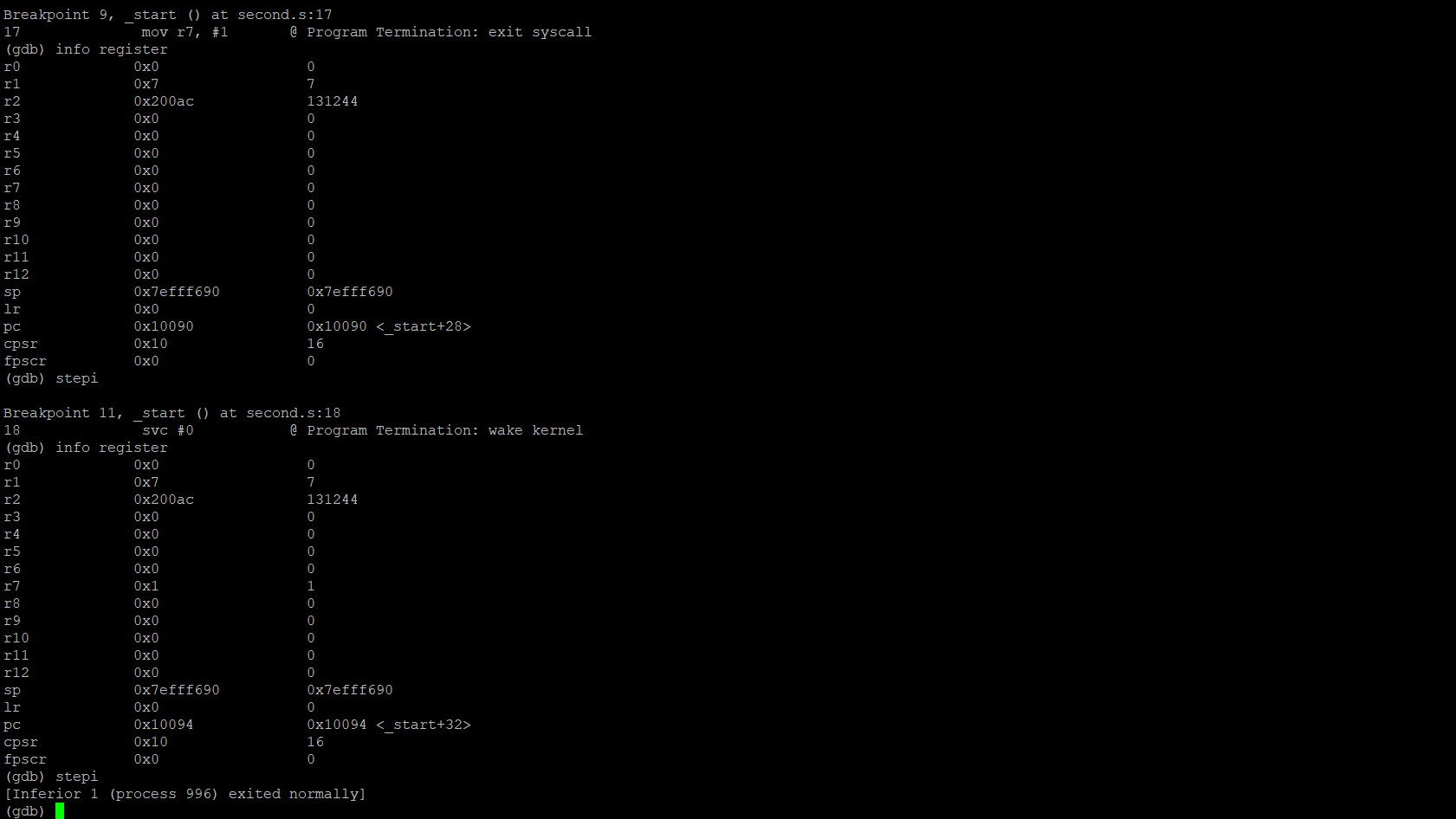
Within these two screenshots, I create second.s file using “nano second.s”. After that I assembled the file by using “as -o second.o second.s” and used “ld -o second second.o” to link the file. After this I used “./second” to run the code. Since no errors came up, I decided to run the program through the debugger to see the final result

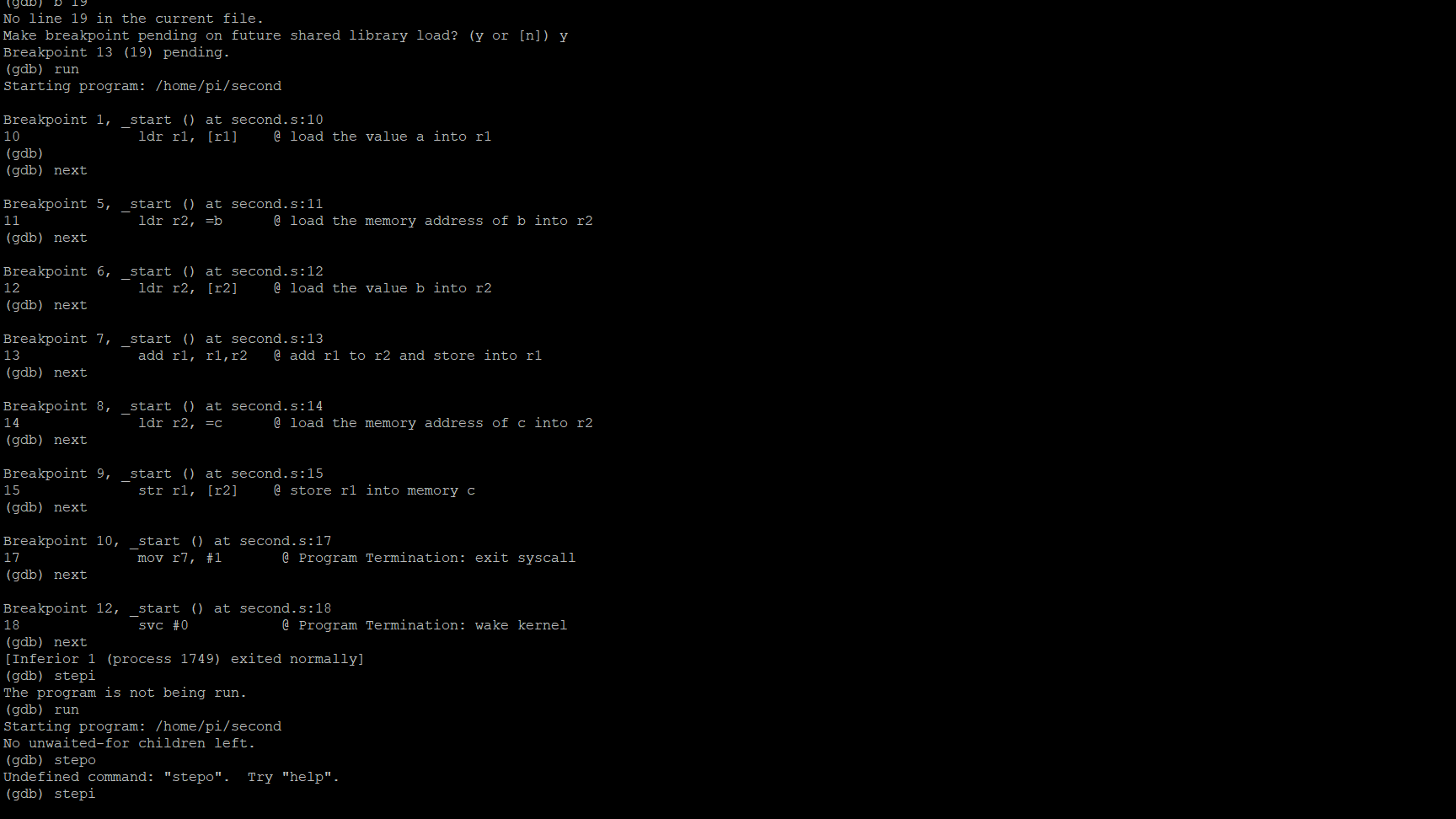


After loading into the debugger, I set the breakpoints from line 10 to line 18. To make sure all lines of code were executed, I put a breakpoint at “b 19” or line 19 to make sure I was at the end of the line.

 The debugger ran the program and stopped at the first breakpoint. From the first breakpoint at line 10 we can see the value the memory address of a (0x200a4) was loaded into Register 1.

 I checked the address to make sure the right value was put into the right address to make sure there was no fault in my code and as you can see the value of 2 in Register 1 stands true. The next breakpoint the memory address of b (0x200a8) was loaded into Register 2.

 From the next breakpoint we can see the value from Register 1 (Previously the sum of Register 1 and Register 2), which is 7 was stored in memory c. As we can see above the memory address is seen above in Register 2 (0x200ac).



Initially I ran the code and did not check the info register, I just checked if the code ran. But as you can see, I corrected it above.

**Parallel Programming Skills**

By: Landon Wang

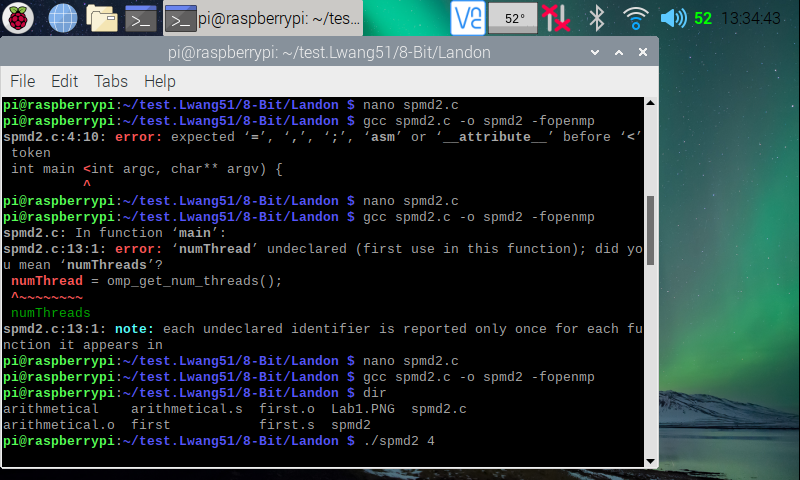
* **Identifying the components on the raspberry PI B+ (5pt).**
  + Looking at the PI with birds-eye view and the USB ports facing to the right
    - CPU/RAM at the bottom of the GPIO pins
    - Ethernet controller to the left of the USB ports
    - USB ports to the top of ethernet port
    - Ethernet port to the right of audio port
    - Audio port to the left of camera ribbon cable connector
    - Camera connector to the right of HDMI port
    - HDMI port to the right of power supply port
    - Display ribbon connector to the left of CPU/RAM and top of power port
* **How many cores does the Raspberry Pi’s B+ CPU have (5pt)?**

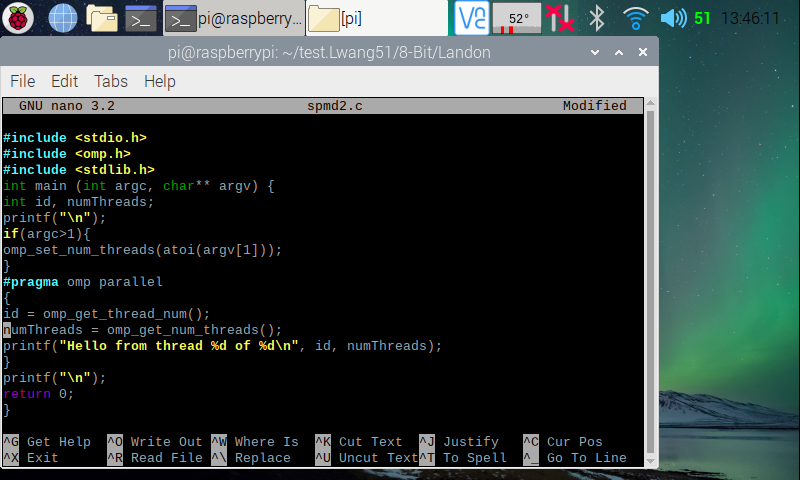
Raspberry PI 3 module B+ has a Quad-Core CPU

* **List three main differences between X86 (CISC) and ARM Raspberry PI (RISC). Justify you answer and use your own words (do not copy and paste) (8pt).**
  + X86 is a CISC (Complex Instruction Set Computing) processor while ARM is a RISC (reduced Instruction Set Computing. The biggest difference between the two is the instruction sets. CISC processors have bigger instruction sets with more features (while RISC has 100 or less instructions), and they give memory access to complex instructions. Because of this, x84 processors has more operations and addressing mode but less registers than ARM.
  + X86 processors uses the little-endian format (storing data from right to left). The old version (before v3) of ARM processors also uses little-endian, but as new versions roll out, it switched to big-endian (storing data from left to right) with a switchable endianness feature.
  + ARM processors use instructions that can only operate on registers (while X86 can use both register and memory) and uses the load/store model to access the memory. For example, if we want to increment a value on a specific memory address on ARM, we first need to load it to the register, increment it, then store it back to the memory from the register. In X86, we can directly access the memory and increment the value since instructions can both operate on registers and memory.
* **What is the difference between sequential and parallel computation and identify the practical significance of each (6pt)?**
  + Sequential computation executes programs on a single processor at a time while a parallel computation executes programs on multiple processors at a time
  + In sequential computing, programs are broken into series of instructions that will be sequentially executed one after another by the CPU with only one instruction being executed at any given time.
  + In parallel computing, programs are broken into parts of instructions that will then be broken down into series of instructions that each different processor will then execute in coordination with each other.
* **Identify the basic form of data and task parallelism in computational problems (5pt).** 
  + Data Parallelism
    - Computation is applied to multiple data items
    - Amount of parallelism is proportional to input size (causes huge amount on potential parallelism
    - Gives programmers flexibility in writing scalable (with input size) parallelism programs in a way that the program should be using all the available parallelism.
  + Task Parallelism
    - Parallelism is organized around tasks instead of data.
    - Work should be balanced, and all the work should contribute to the result in some way or form
    - Task parallelism does not scale well as data parallelism
* **Explain the differences between processes and threads (6pt).**
  + Processes:
    - Abstraction of a running program
    - Does not share memory with each other
    - Single-core CPU operates on one process at a time while multi-core CPU can operate on more processes at a time (concurrency)
  + Threads:
    - Lightweight process allowing a single executable to be decomposed to smaller independent parts
    - All thread shares common memory of the process they belong to
    - The operating system will schedule threads on separate available cores/CPU
* **What is OpenMP and what is OpenMP pragmas (3pt)?** 
  + OpenMP
    - Industry standard since the end of the 1990s with native support with GCC compilers
    - Uses implicit multithreading model where the library creates and manages threads
    - Makes programmer’s task simpler and less error-prone
    - Uses Thread Pool pattern to concurrent execution control
    - Initializes group of threads (pool of threads) for programs, and the threads will execute concurrently during portion of the code when told so by the programmer
  + OpenMP pragmas
    - An alternative to pthreads (a low-level thread package) when writing programs for shared-memory, multicore hardware
    - Compiler directives that allows the compiler to generate threaded code instead of having the programmer generating and managing it (like in pthread).
    - Uses fork/join and single program, multiple data (two primary patterns used as program structure implementation strategies.
* **What applications benefit from multi-core (list four) (4pt)?** 
  + Database and web servers
  + Compilers
  + Multimedia and scientific applications
  + Applications with thread-level parallelism
* **Why Multicore (why not single core, list four) (4pt)?**
  + Hard to increase single-core clock frequencies
  + Deeply pipelined circuits
    - Heat and speed of light problems
    - Difficult design and verification with large design team needed
    - Server farms need expensive AC units
  + Most new applications use multithreading
  + Trending in computer architecture (we are shifting towards parallelism)

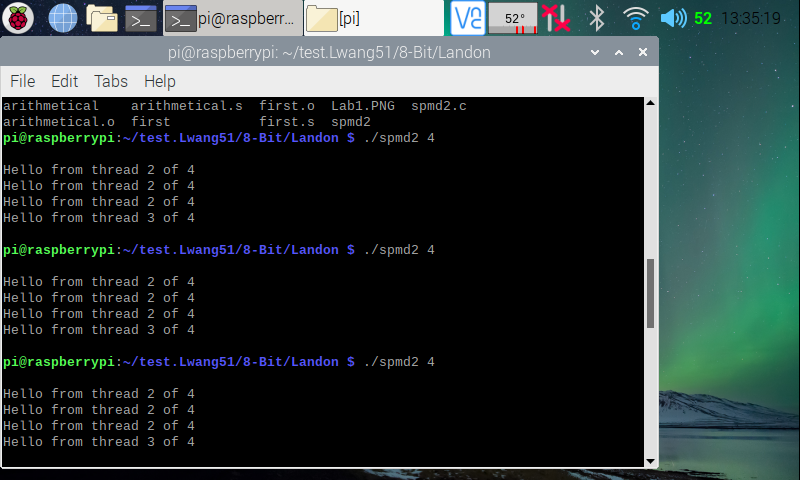
**GETTING STARTED WITH THE RASPBERRY PI AND PARALLEL PROGRAMMING**

By: Landon Wang

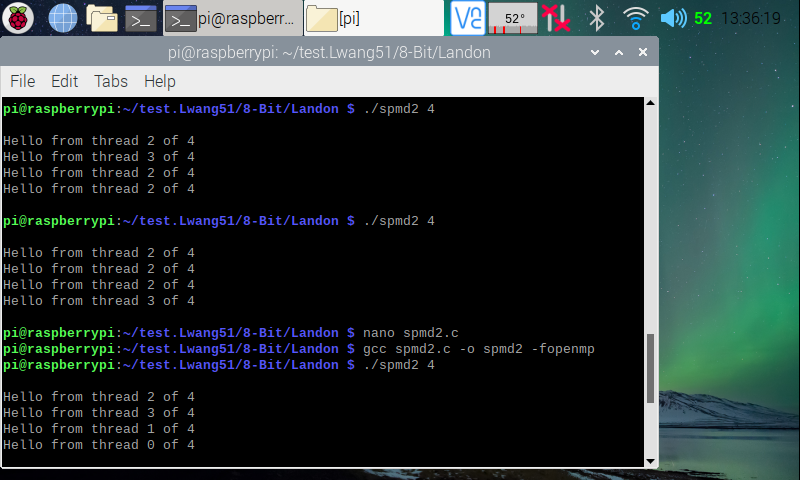


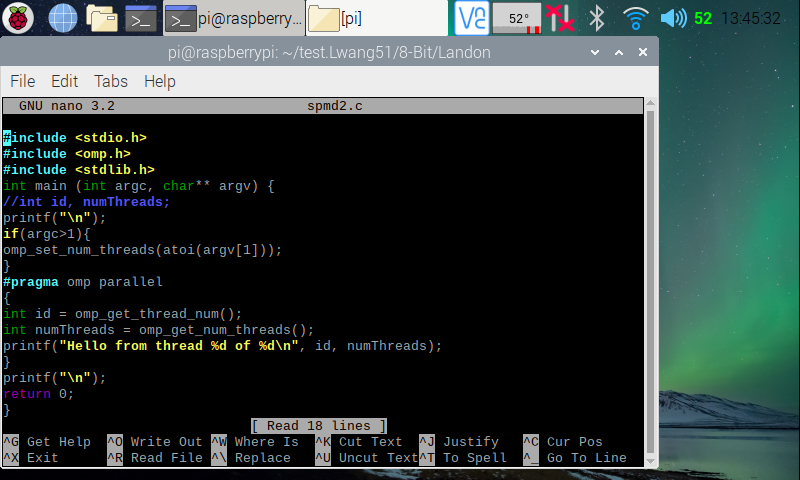


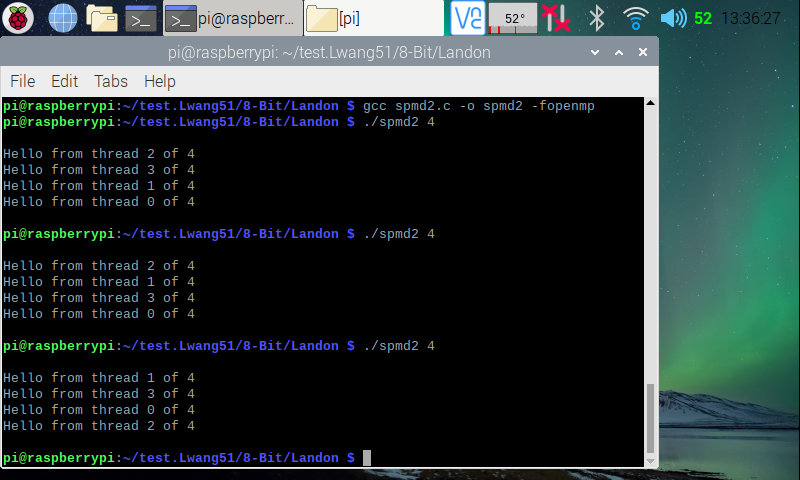
Here (in the two screenshots above), I created the spmd2 program using the instruction **nano spmd2.c** command. After writing/copying the program from the Getting Started with the Raspberry Pi and Parallel Programming file, I made an executable program file for it using the instruction **gcc spmd2.c -o spmd2 -fopenmp**. I ran into a few errors (typed a “<” instead of a “(” on line 4, 10th character and called “numThread” instead of “numThreads” on line 13) before getting the terminal to create an executable file for spmd2 program. After the executable file was created, I ran the program with 4 threads to fork using the instruction **./spmd2 4**.



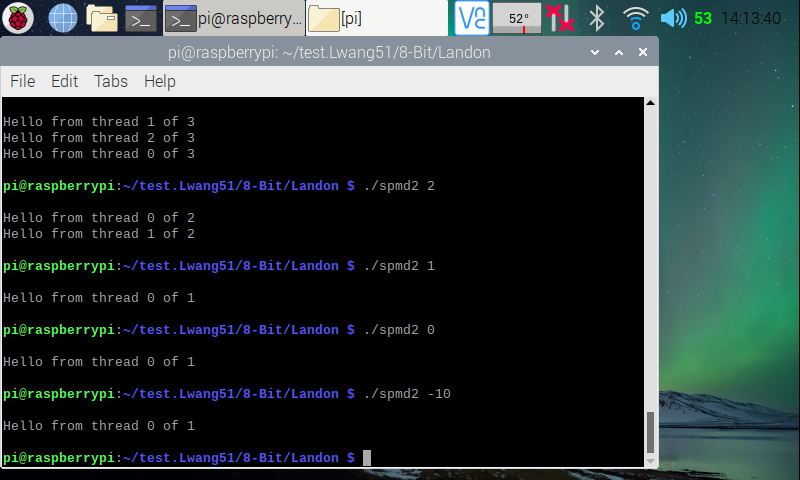
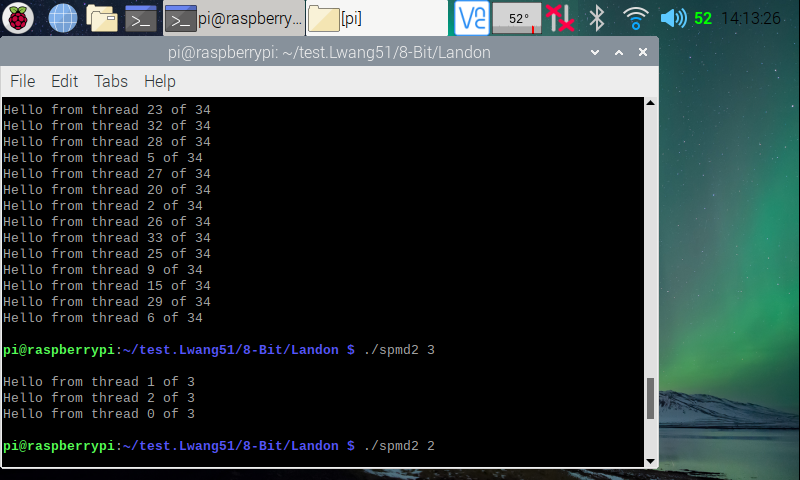
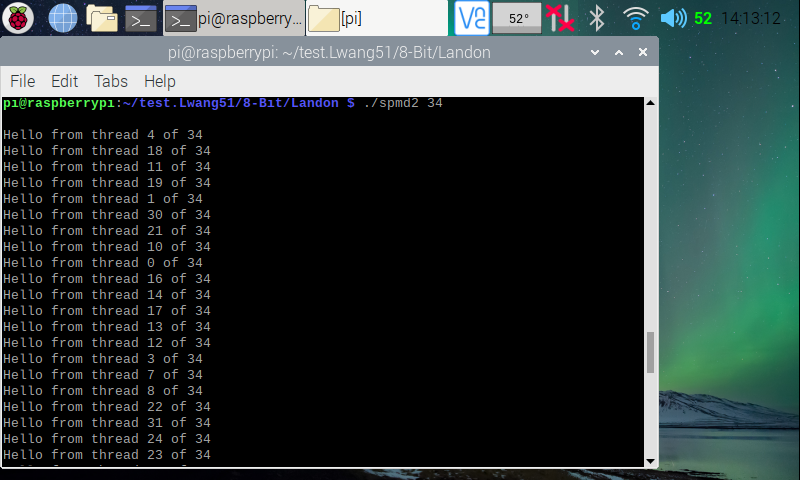
Here (in the screenshot above), I tested the spmd2 program a few times, each running with 4 threads to fork. From my observation, the program does run with four threads, but the thread id number 2 repeats many times. This is because, according to the Getting Started with the Raspberry Pi and Parallel Programming tutorial file, although the PI has multiple cores, the cores share the same memory bank in the machine. We declared variables outside of the “#pragma opm parallel” block, so all threads point to a same variable memory address.







Here (in the three screenshots above), I ran two more trails of the spmd2 program with 4 threads. Looking at the top one, we can see that the order in which the threads finished are different. This is natural, because it is never guaranteed which thread will finish first. To fix the id error, I went back in edited the code, then made a new executable file for it. In the code, I commented out “int id, numThreads” code on line 5, then made id and numThreads inside the “#pragma opm parallel” to full variable declarations. This way, when a new thread is formed, they each will have their own private copy of the two variables. I ran it again with four threads, and now, we can see that each thread has its own special id starting from zero. I then ran the program two more times, and we can see that each time it runs, the threads finish in different order.



Here (in the three screenshots above), I ran the program with different amount of threads (34; 3; 2; 1; 0; -10 threads). From my observation, running the program with n amount (greater than or equal to 1) of threads will create n amount of threads each finished in different orders. I tried with 0 threads and -10 thread (expecting an error), but it just executed with one thread. I guess that is the default.

**Parallel Programming Skills**

By: Raejae Sandy

**Identifying the components on the raspberry PI B+**

ARM CPU/GPU with video core 4 graphics  
GPIO general purpose input output connection points  
USB ports  
HDMI Ports  
Ethernet for wired network access

SD card slot

**How many cores does the Raspberry Pi’s B+ CPU have?**

-Quad Core CPU has 4 cores

**List three main differences between X86 (CISC) and ARM Raspberry PI (RISC). Justify your answer and use your own words (do not copy and paste)**

1) One difference is how X86 uses Little Ending format which means the processor reads memory from right to left  
2) Another difference is in complexity where ARM contains a reduced instruction set : RISC : and Arm contains a more complex instruction set.  
3) ARM is limited to instructions that are set to registers while x86 has more flexibility with the load store model that allows different ways to access memory and the registers.

**What is the difference between sequential and parallel computation and identify the practical**

**significance of each?**

The idea is that with sequential programming, programs are ran on a single processor while parallel runs on multiple. Since parallel allows distribution of processing power among separated inputs to source task, it is optimal in cases in which complex functions are required, while in cases with Sequential it is better for single threaded applications that only need a priority or single channel focus.

**Identify the basic form of data and task parallelism in computational problems.**

Data Parallelism allows for scaling and management of large computational issues while task has poor scaling but more efficiency in a set target.

**Explain the differences between processes and threads.**

Processes reflect on programs that are running and reflect what the cpu is running while threads are handled differently. Threads are scheduled different and more lightweight resulting in faster operating time,

**What is OpenMP and what is OpenMP pragmas?**

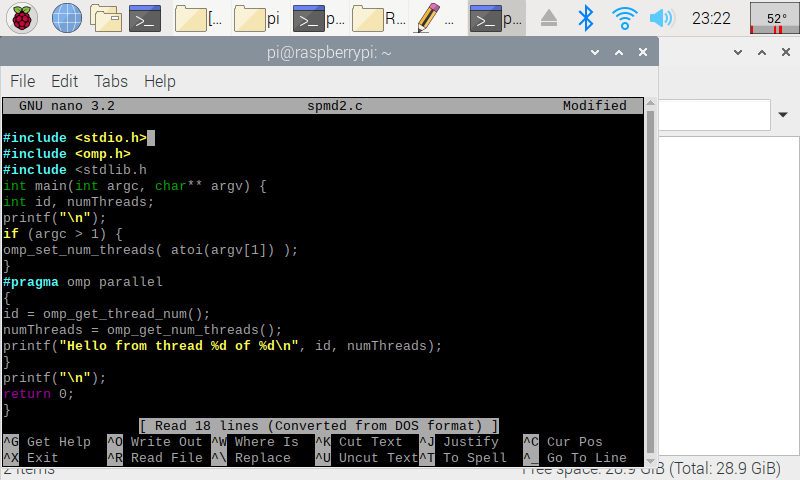
OpenMP is a library created for multithreading models and managing threads. Pragmas is a compiler created to simplify writing code for multiple cores and threaded code.

**What applications benefit from multi-core (list four)?**

Database servers  
Multimedia applications  
Web Servers  
Compilers

**Why Multicore? (why not single core, list four)**

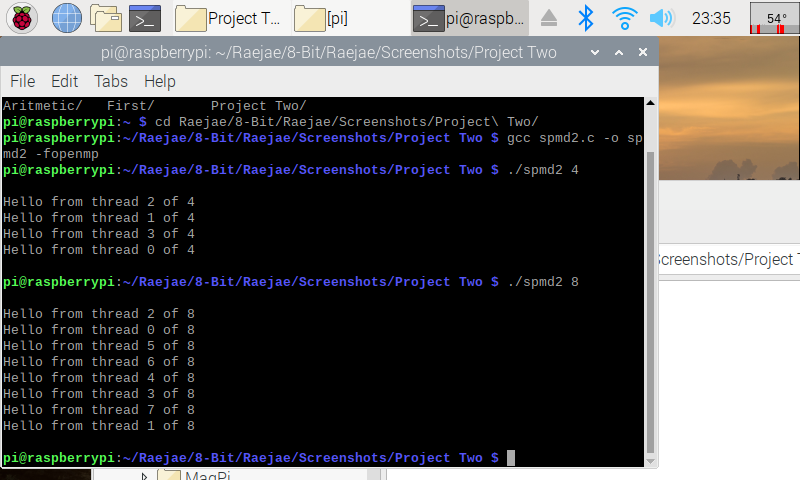
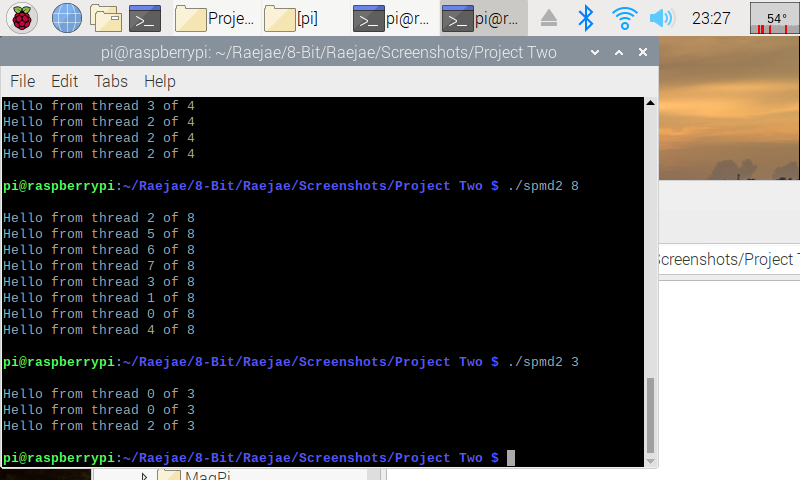
* -It is very difficult with current technology to continuously upgrade frequencies.
* -Deeply Pipelined Circuits
  + Issues with heat and speed of light
  + Having many servers require expensive air-conditioning
* Tech is moving towards parallelism as a common trend.
* Most applications are developed multi-threaded.

**Getting Started with Raspberry Pi and Parallel Programming**

By: Raejae Sandy

This code is before we dedicate ids to each thread which doesn’t result in full variable declarations. After setting this however, the threads had a separate id for each one. In the code below it is not set to numerical order as each thread completes at different intervals.

In the code above it’s interesting to note that the string formatting used reflects in higher level languages, and it’s also interesting to notice the allocation set in the beginning lines, as we introduce new libraries and new managements.

After assembling the file and running, we notice that different threads in a series of four are implemented on the same memory bank. This is bad because for optimization and doesn’t split the threads as we need it. To fix this we referenced the variables in the code to allow each thread a separate memory bank and verified as displayed.

**Parallel Programming Skills**

By: Tony Ngo

1. Identifying the Components on the Raspberry PI B+
   1. CPU/RAM, Display, Power, Ethernet, 2 USB Hubs, Ethernet Controller, HDMI
2. How many cores does the Raspberry Pi’s B+ CPU have?
   1. Four
3. List three main differences between x86 (CISC) and ARM (RISC)
   1. ARM has a simplified instruction set (Reduced) versus x86 instruction set which has a more intricate set (Complex)
   2. ARM uses only instructions which you can only use registers, while x86 can use memory-based instructions.
   3. X86 processors use the Little-Indian format, which means that the processor loads memory from right to left.
4. What is the difference between sequential and parallel computation and identify the practical significance of each?
   1. In sequential computation, instructions are executed on a single processor and only ONE instruction can be executed at a time. While in parallel computation, a problem is broken into parts that can be solved concurrently on different processors. Parallel programming would be practical in scenarios where more computing power is needed (i.e. complex mathematical equations that require number crunching, CAD programs, etc..), and sequential programming would be practical for extremely simple equations for execution.
5. Identify the basic form of data and task parallelism in computational problems
   1. Problem -> Instructions -> Processor
6. Explain the differences between processes and threads
   1. A process is a program that is in execution, while a thread is a part of the process, which can have multiple within one process (hence, multi-threading). Threads typically will take less time to terminate.
7. What is OpenMP and what is OpenMP pragmas?
   1. OpenMP is a library that supports multiprocessing in certain languages (i.e. C, C++, Fortran). OpenMP Pragmas is a compiler that enables the compiler to generate threaded code.
8. What applications benefit from multi-core (list four)?
   1. 7Zip
   2. Android Studio
   3. Flight Simulators
   4. AutoCAD
9. Why Multicore? (why not single core, list four)
   1. Multicore will benefit applications that can have multiple processes running at once, most all developer/creator programs use multithreading.
   2. It uses less power than single core processors
   3. It improves clock speeds in computers
   4. There is less travel time between signals, which means that the processer will degrade less over time.

**Getting Started with Raspberry Pi and Parallel Programing**

By: Tony Ngo

A screenshot of a cell phone

Description automatically generated Picture 1: Code before fixA screenshot of a cell phone

Description automatically generatedPicture 2: Threads before fix

A screenshot of a cell phone

Description automatically generatedPicture 3: Code post-fix

A screenshot of a cell phone

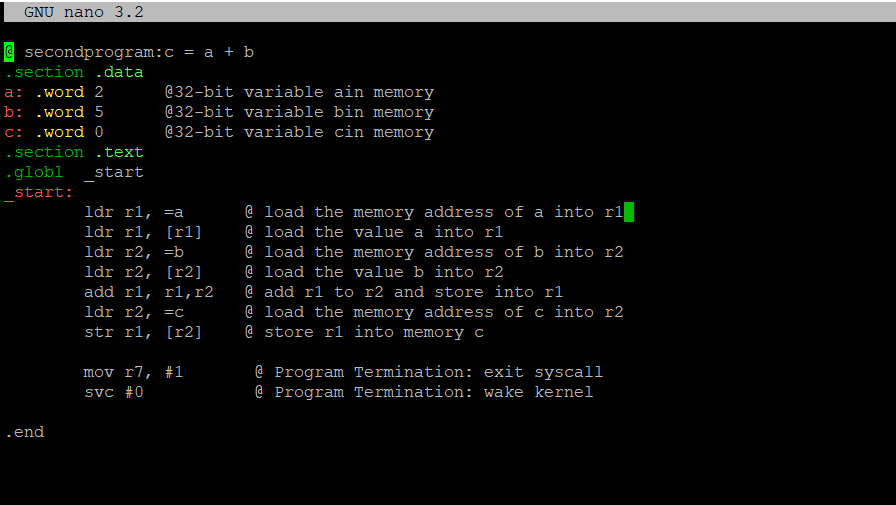
Description automatically generatedPicture 4: Threads post-fix

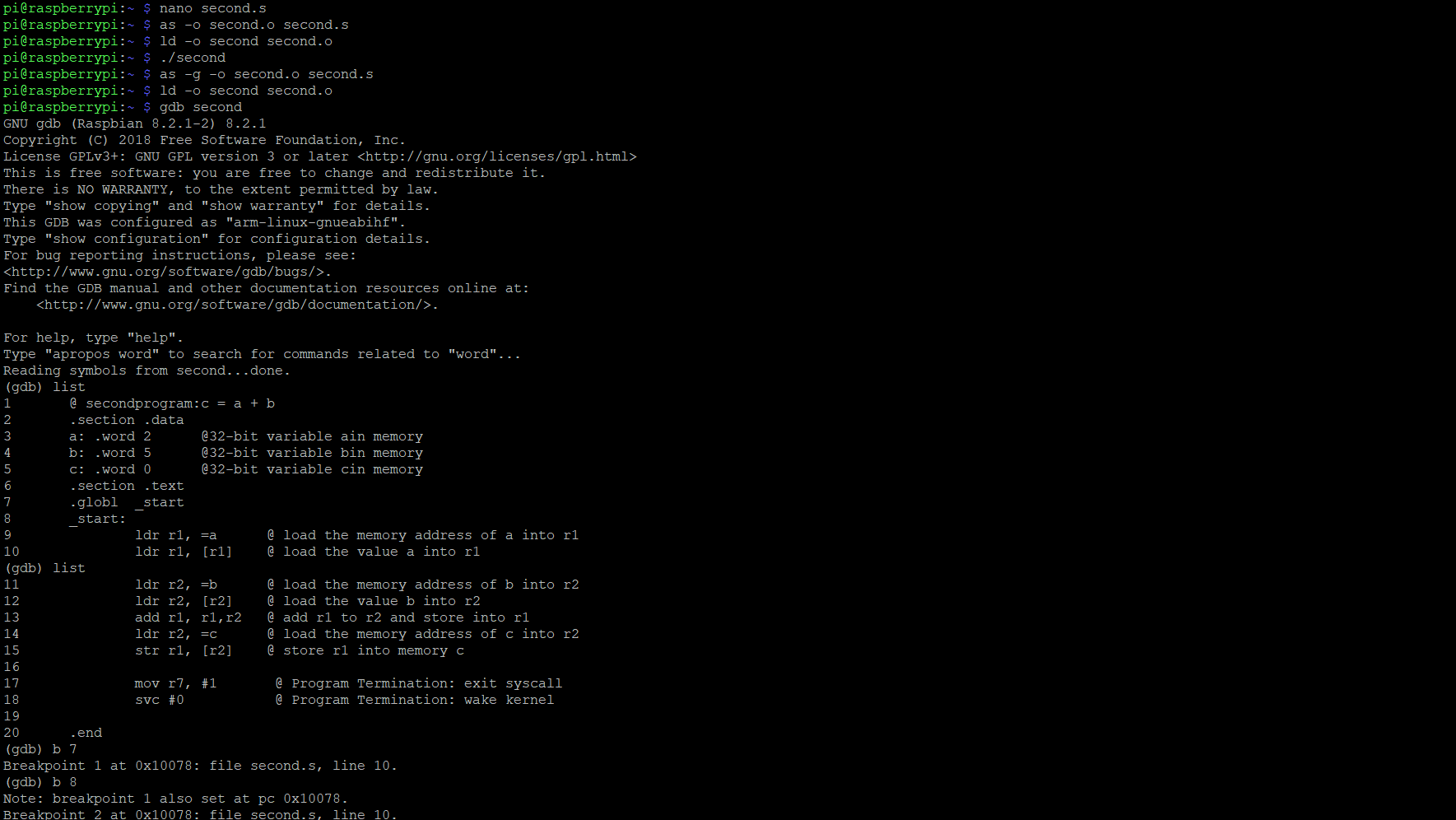
I originally created the program using the nano text editor on the terminal. I copied the spmd2 program and saved and exited the program and made the executable using “***gcc spmd2.c -o -fopenmp***” command, which created an executable called smpd2, when I originally ran the program using the command “***./spmd2 4***”, the thread IDs repeated, which was not supposed to happen. So I went back into the code and changed lines 12/13 to initialize the ints within the function rather than outside. I remade my executable using the “***gcc spmd2.c -o spmd2 -fopenmp***” command and reran it using the command “***./spmd2 4***”, and I did not have repeating threads as seen from the fourth picture.

**ARM Assembly Programing**

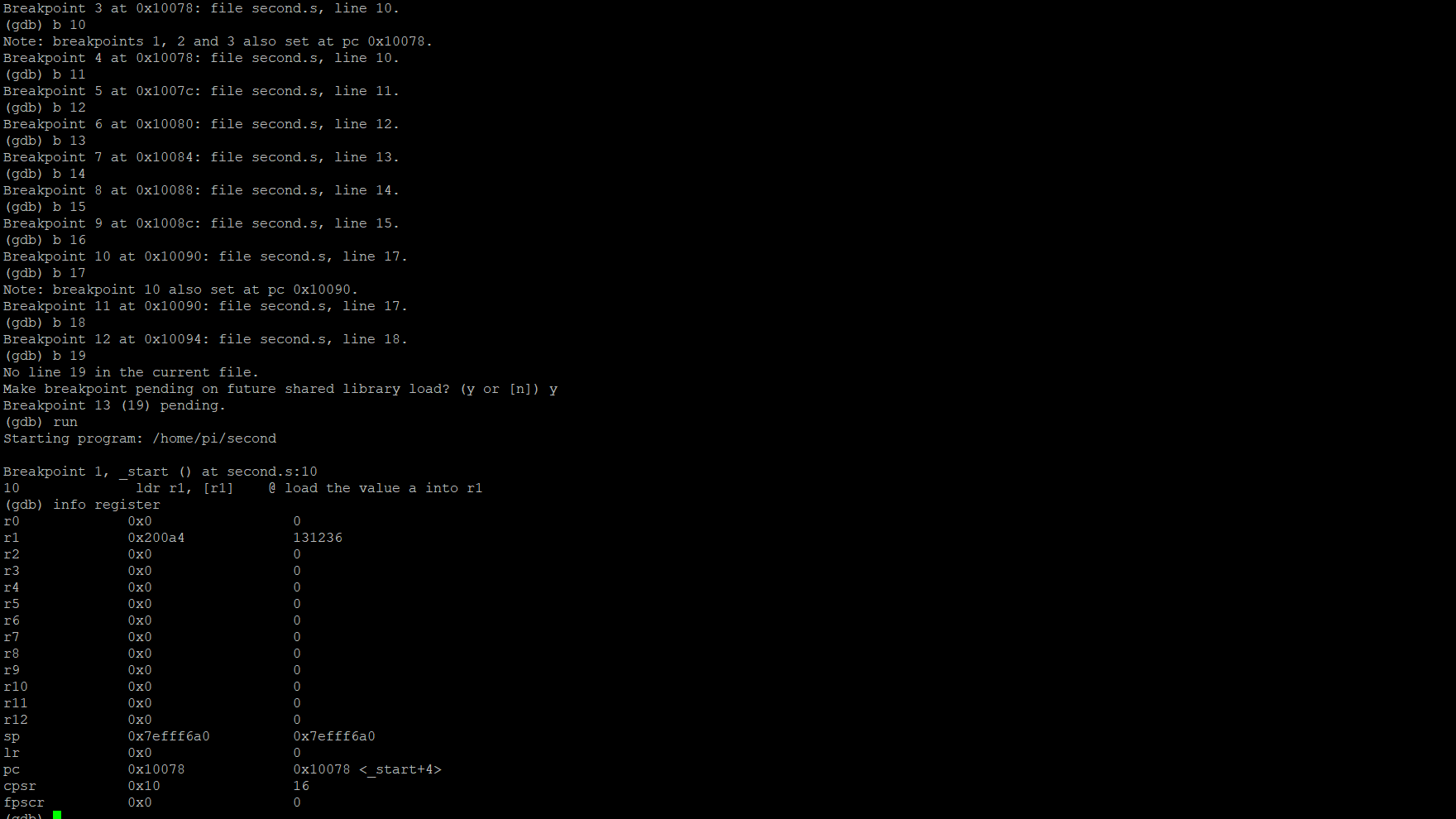
By: Akash Dansinghani

**Second Program**

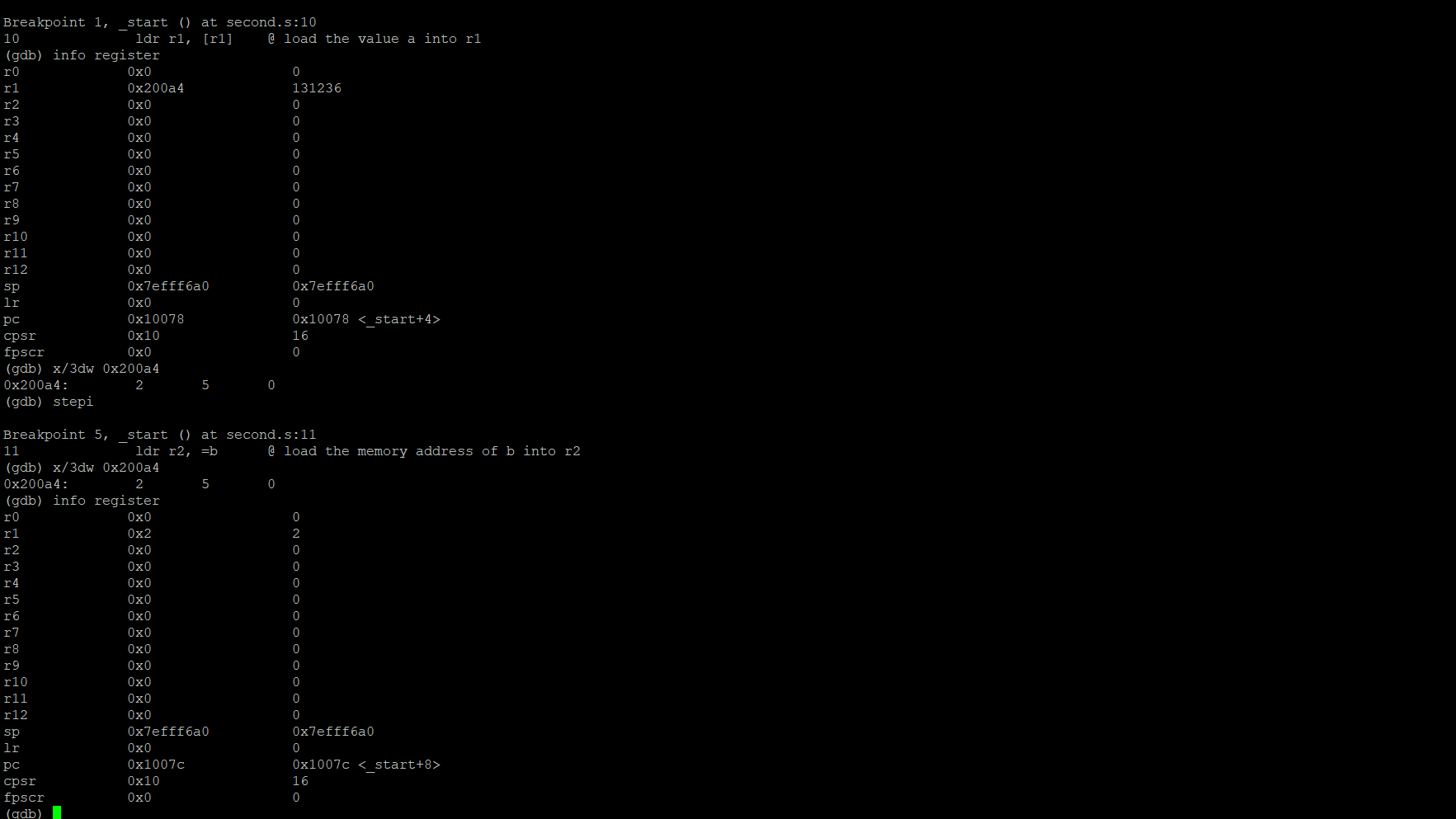




Within these two screenshots, I create second.s file using “nano second.s”. After that I assembled the file by using “as -o second.o second.s” and used “ld -o second second.o” to link the file. After this I used “./second” to run the code. Since no errors came up, I decided to run the program through the debugger to see the final result



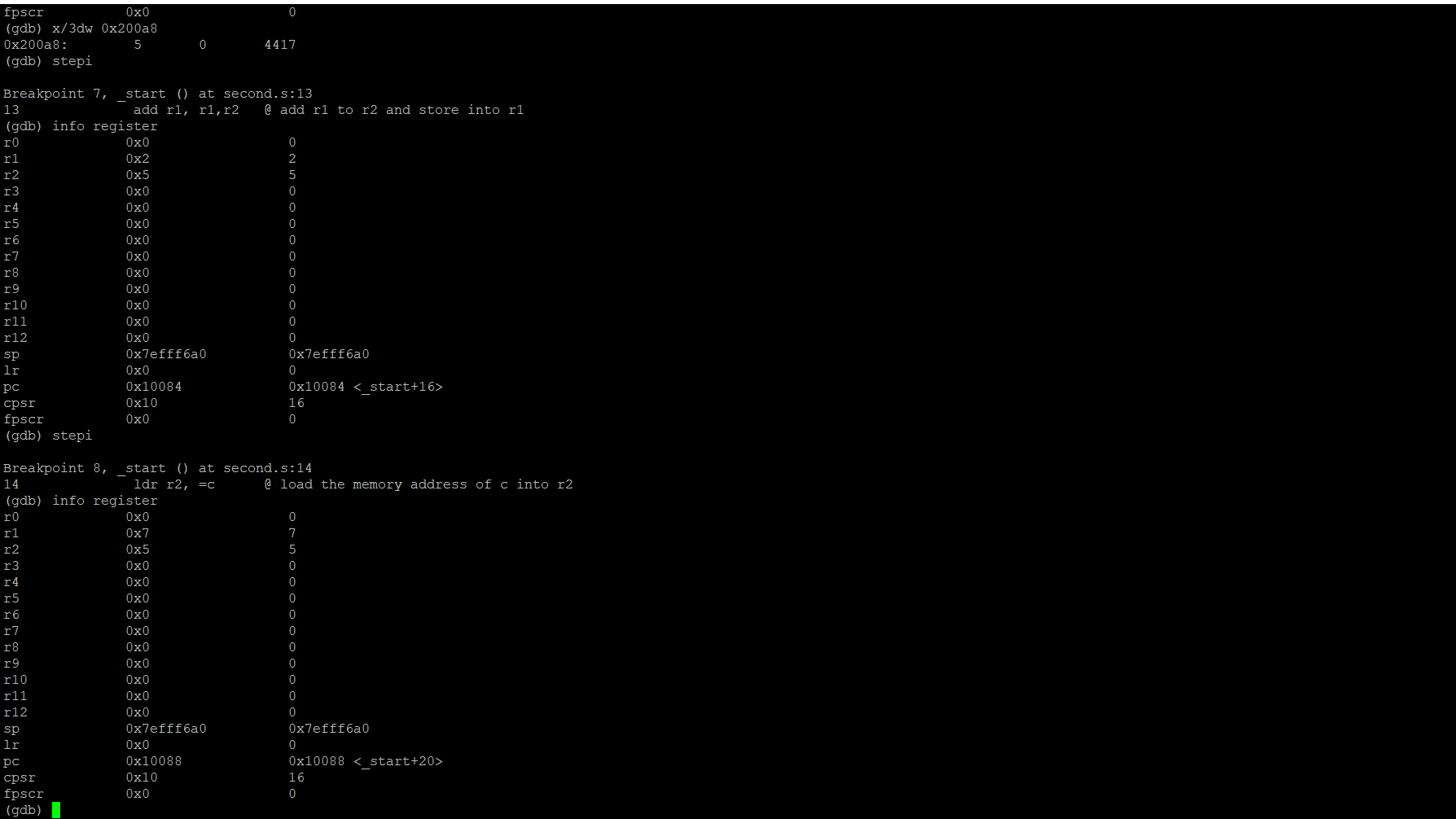
After loading into the debugger, I set the breakpoints from line 10 to line 18. To make sure all lines of code were executed, I put a breakpoint at “b 19” or line 19 to make sure I was at the end of the line.

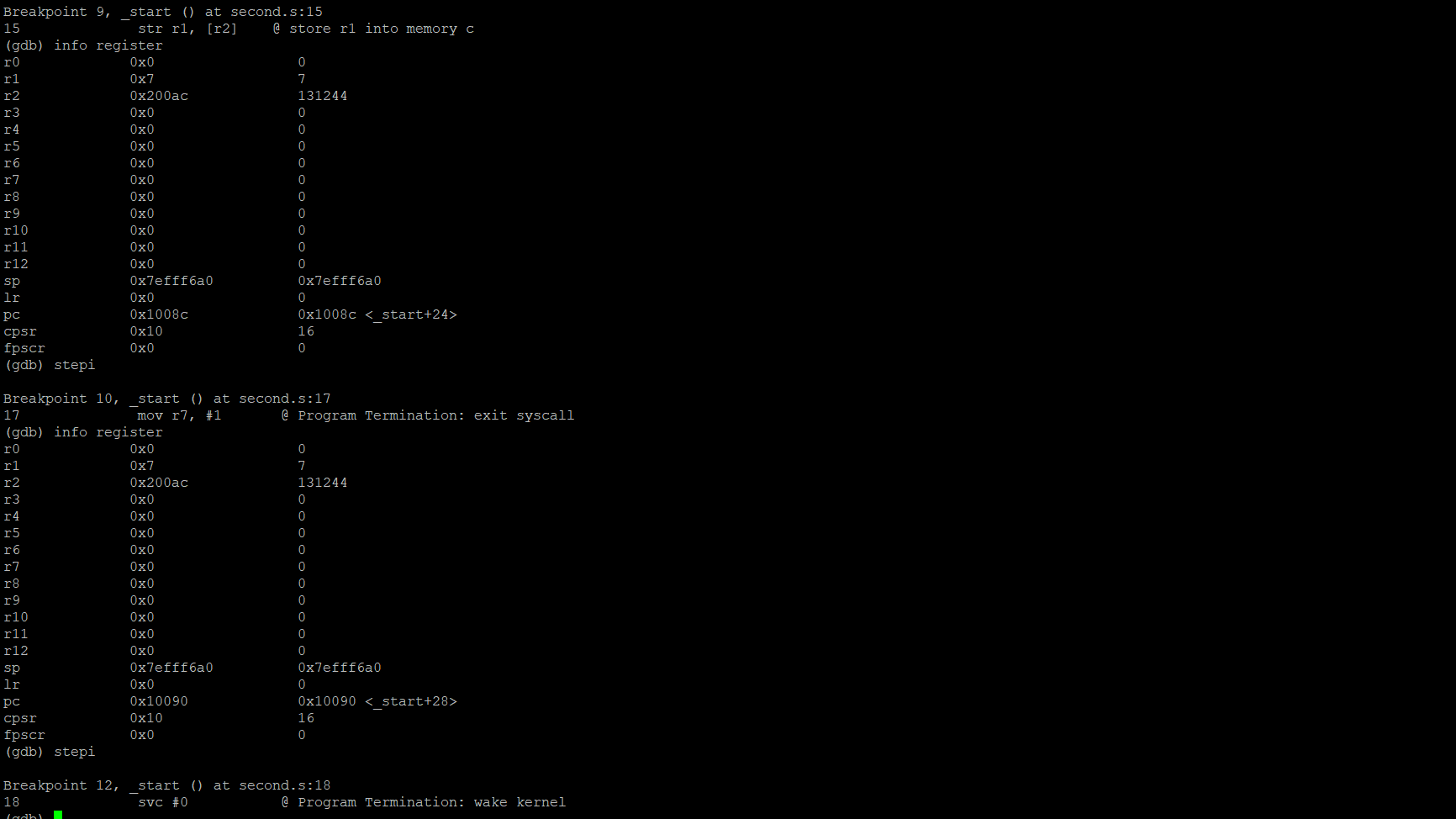


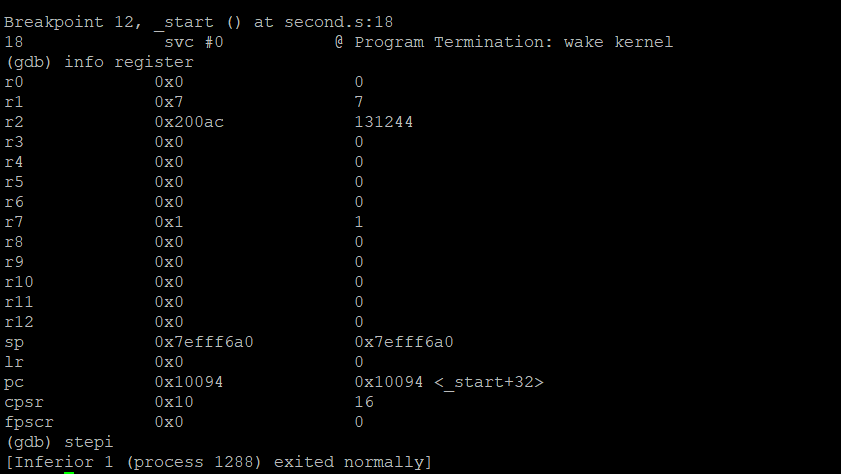
The debugger ran the program and stopped at the first breakpoint. From the first breakpoint at line 10 we can see the value the memory address of a (0x200a4) was loaded into Register 1.



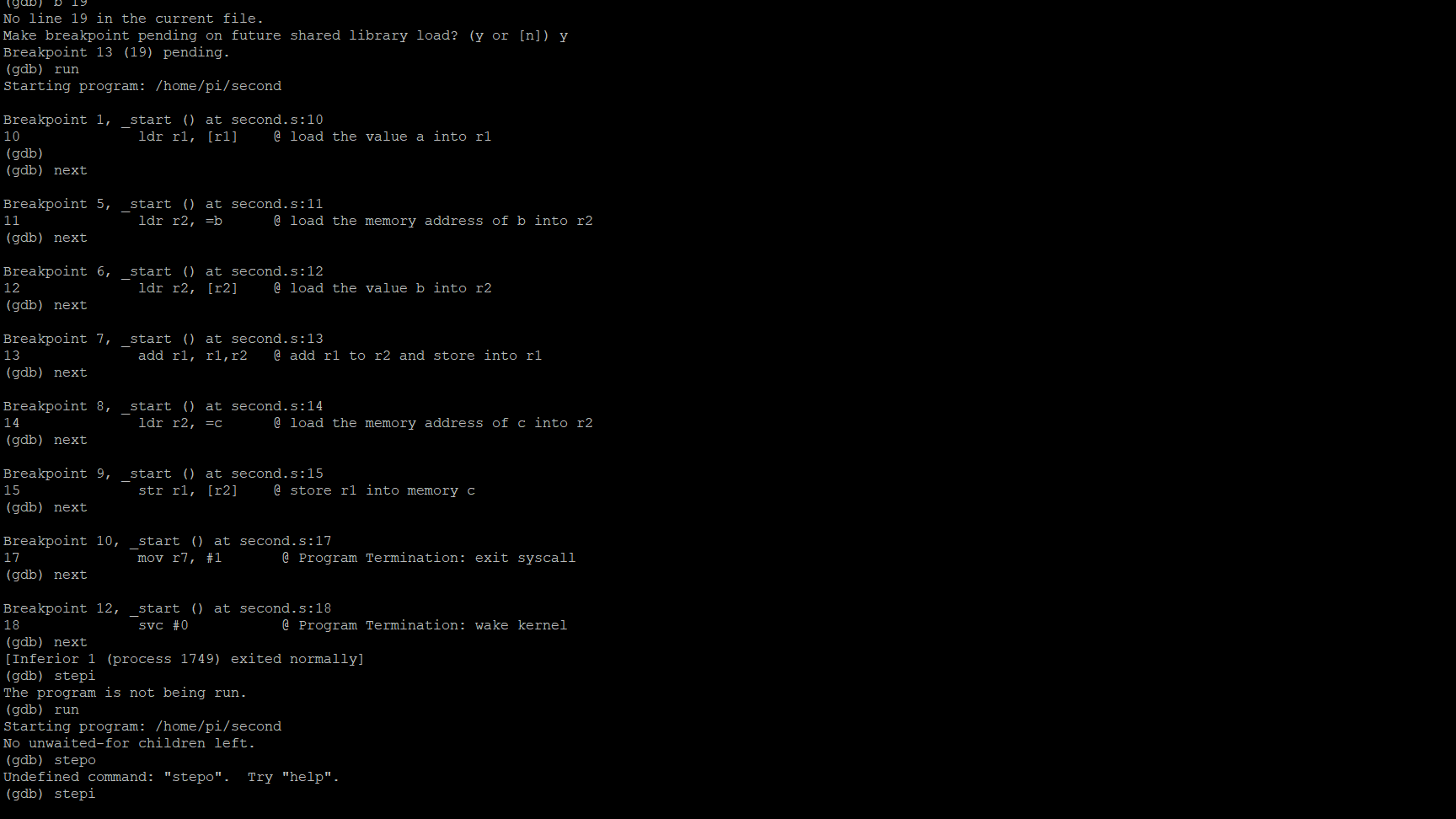
I checked the address to make sure the right value was put into the right address to make sure there was no fault in my code and as you can see the value of 2 in Register 1 stands true. The next breakpoint the memory address of b (0x200a8) was loaded into Register 2.

We can see the value of 5 was place in Register 2. In Breakpoint 8 on Line 14 we can see the value of the Register 1 and the value of Register 2 were added together and placed back into Register 1.



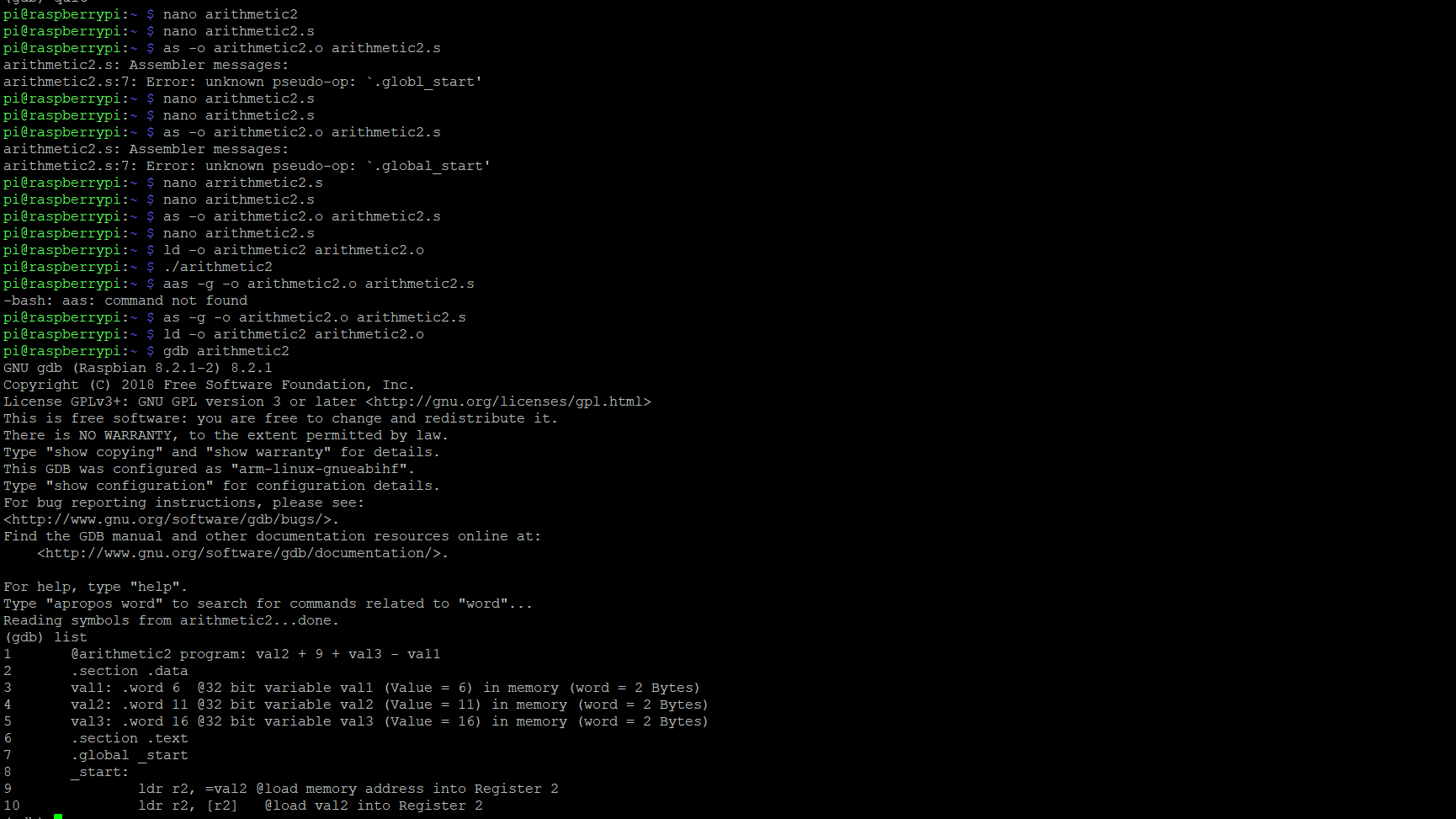
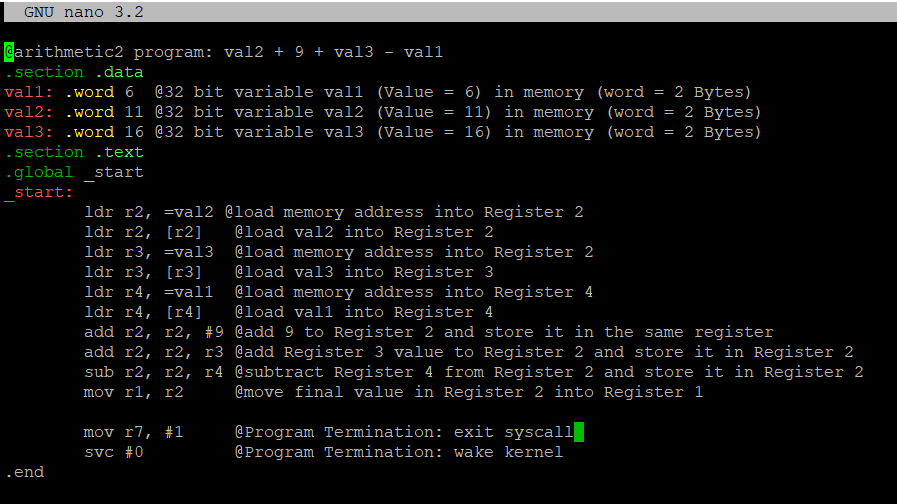
From the next breakpoint we can see the value from Register 1 (Previously the sum of Register 1 and Register 2), which is 7 was stored in memory c. As we can see above the memory address is seen above in Register 2 (0x200ac). 

Lastly, we can see the value of one (coded as #1) was placed in Register 7.

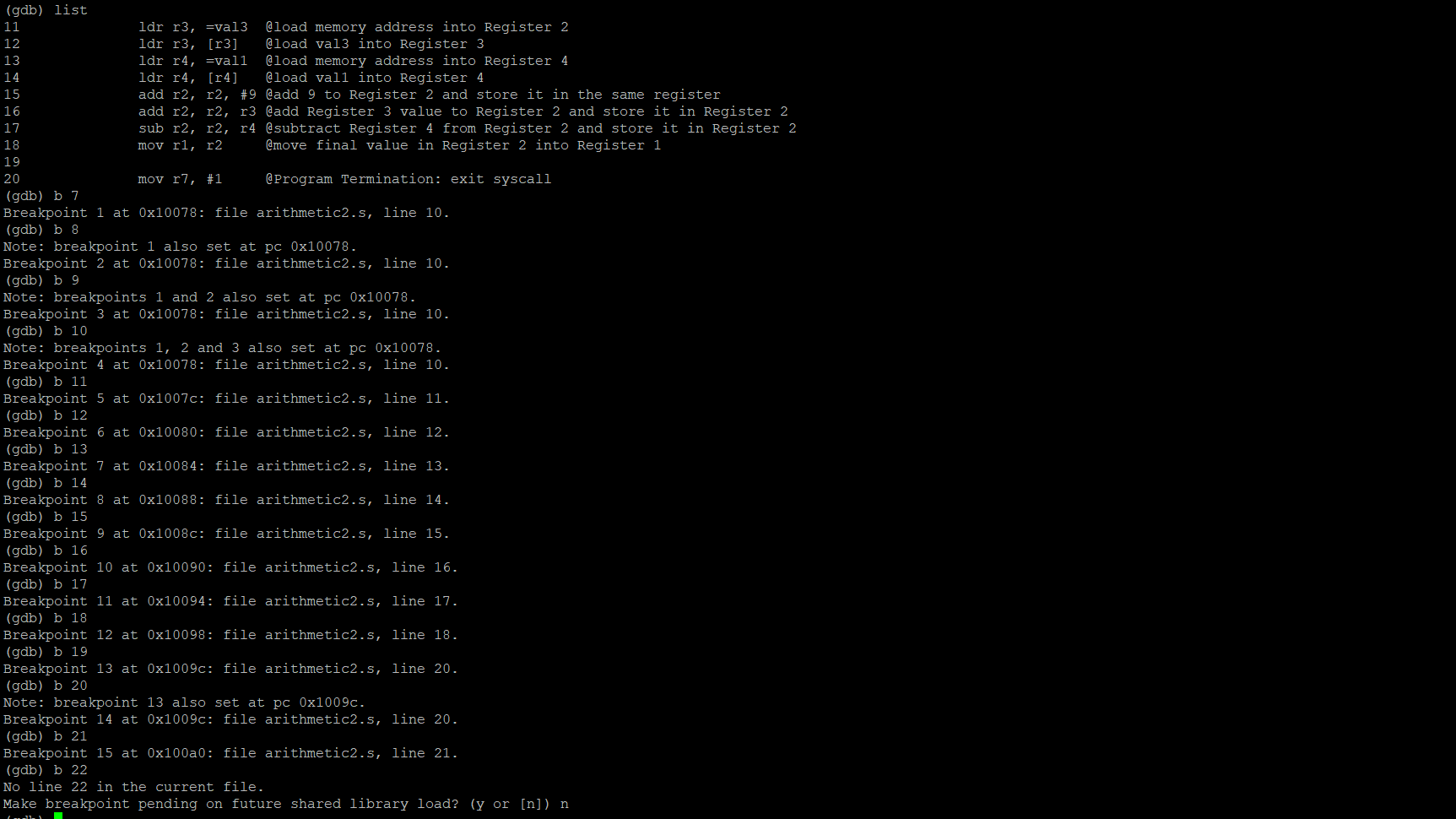


Initially I ran the code and did not check the info register, I just checked if the code ran. But as you can see, I corrected it above.

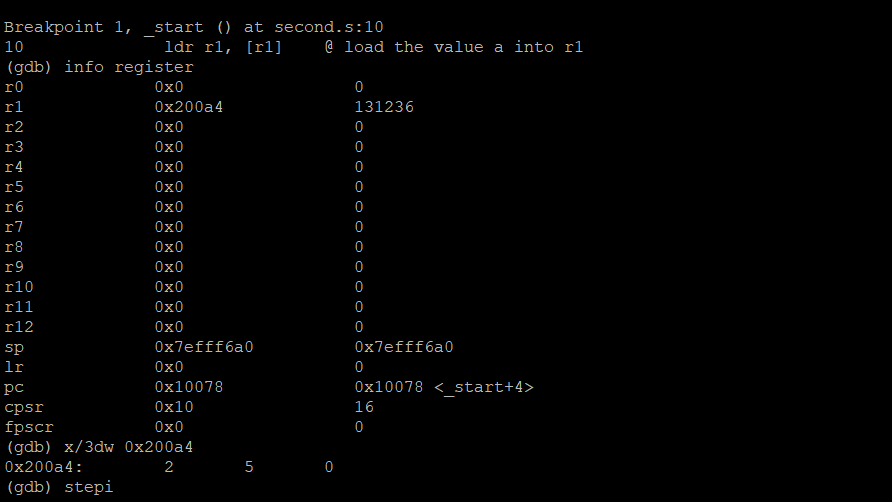
**Arithmetic2 Program**



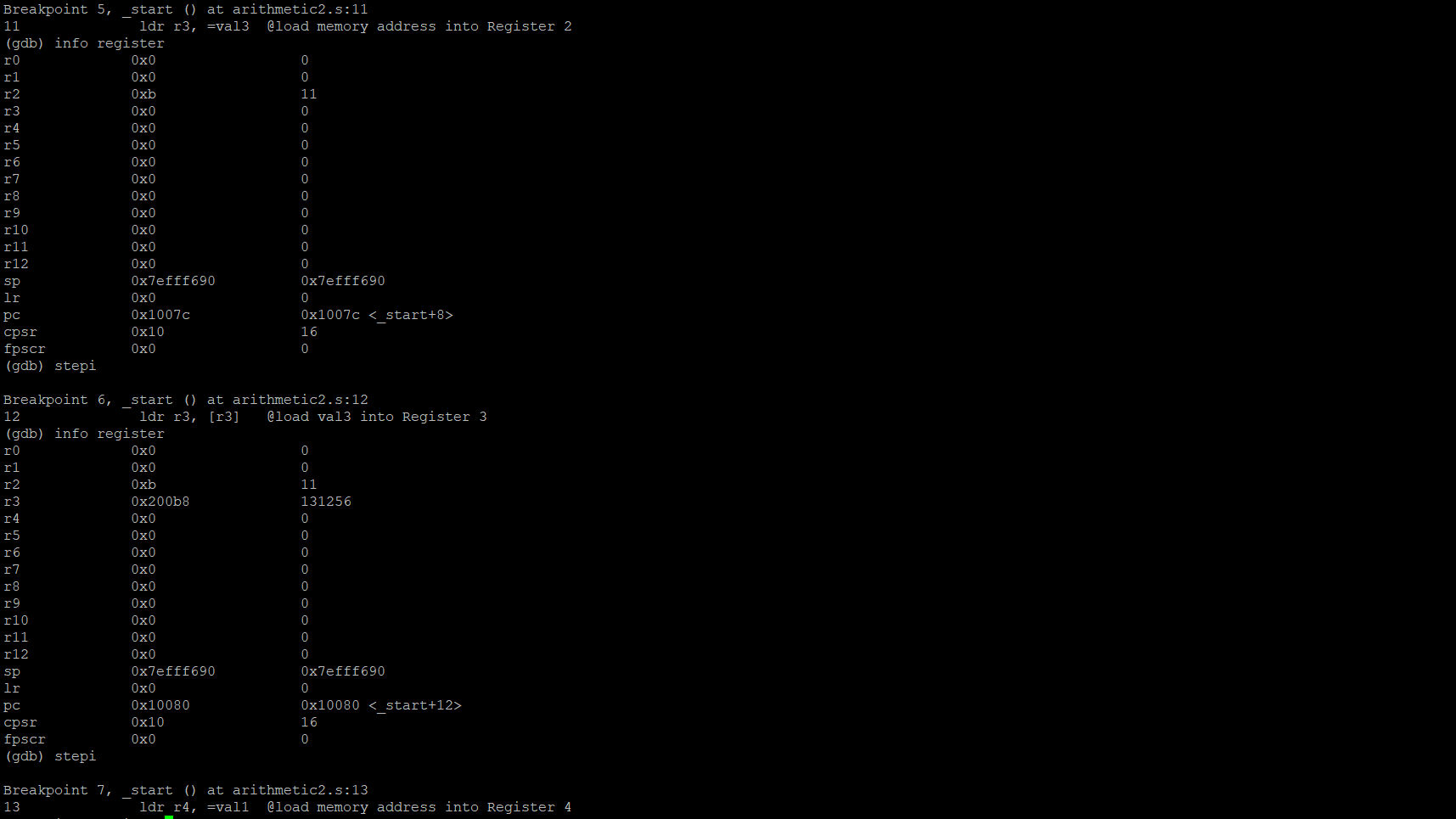
I created the file arithmetic2.s file using nano. After that I assembled the file using “as -o arithmetic2.o arithmetic2.s”. I initially got an error with the code because I did not put a space between .globl [and] \_start, but when I fixed that the file was assembled. After that, I linked arithmetic2 file by typing in “ld -o arithmetic2 arithmetic2.o”. Since no errors came up, I ran the program. The program did not run as you can see above so I prepared the program to go through the debugger using “as -g -o arithmetic2.o arithmetic2.s and ld -o arithmetic2 arithmetic2.o to assemble and link the program to run in the debugger respectively and ran the program through the debugger.



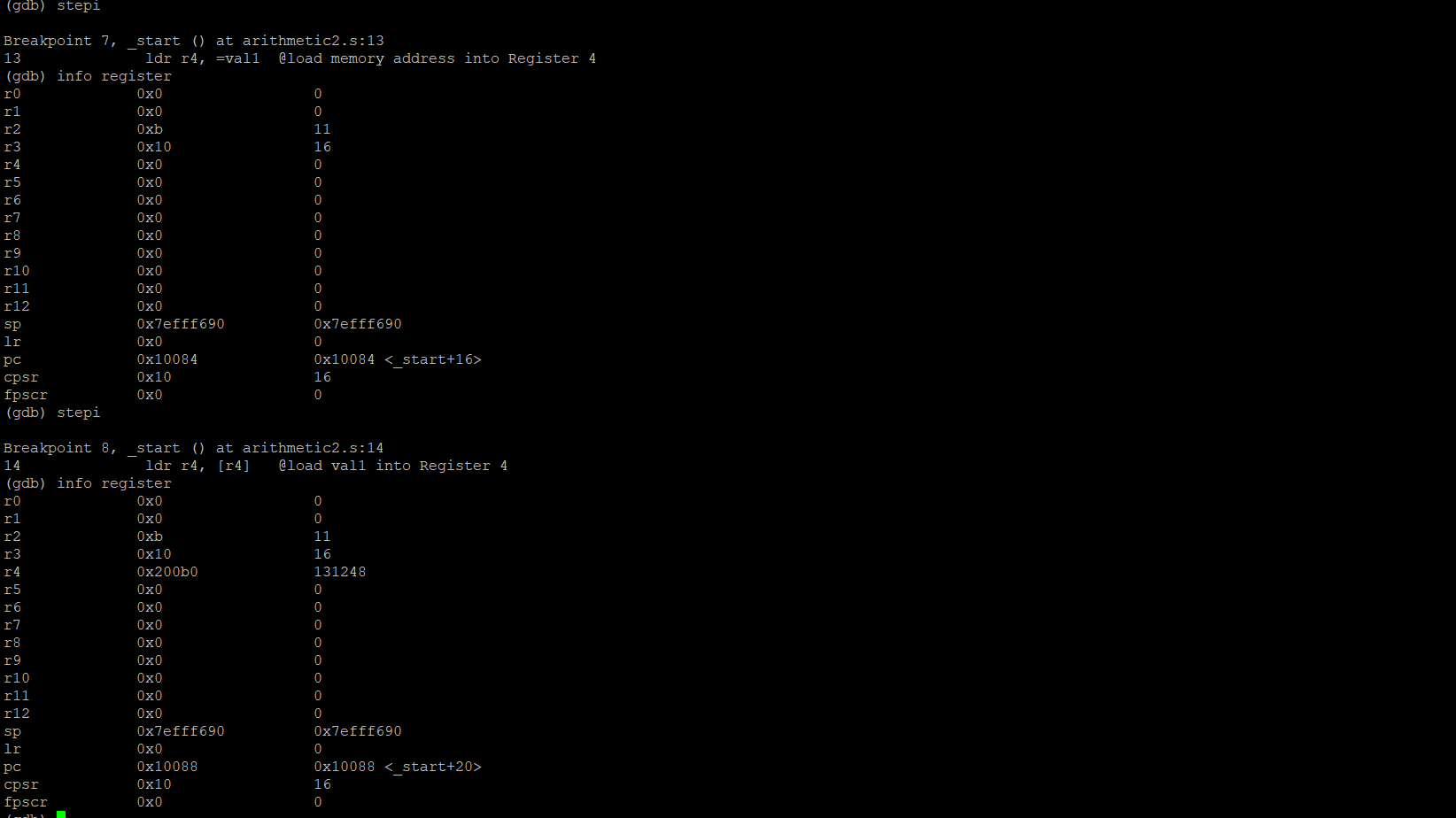
I placed breakpoints from Line 10 through Line 21 using “b 10” through “b 21”. As you can see, I started my breakpoints before 10 and after 21 to make sure I did not miss any code through the program.



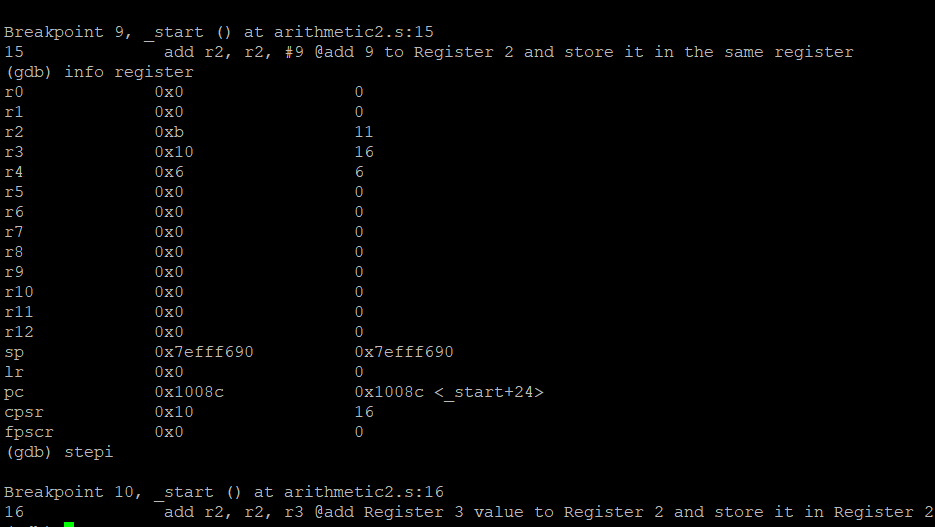
After running the program, we can see the memory address (0x200a4) of “a” is loaded into Register 1.



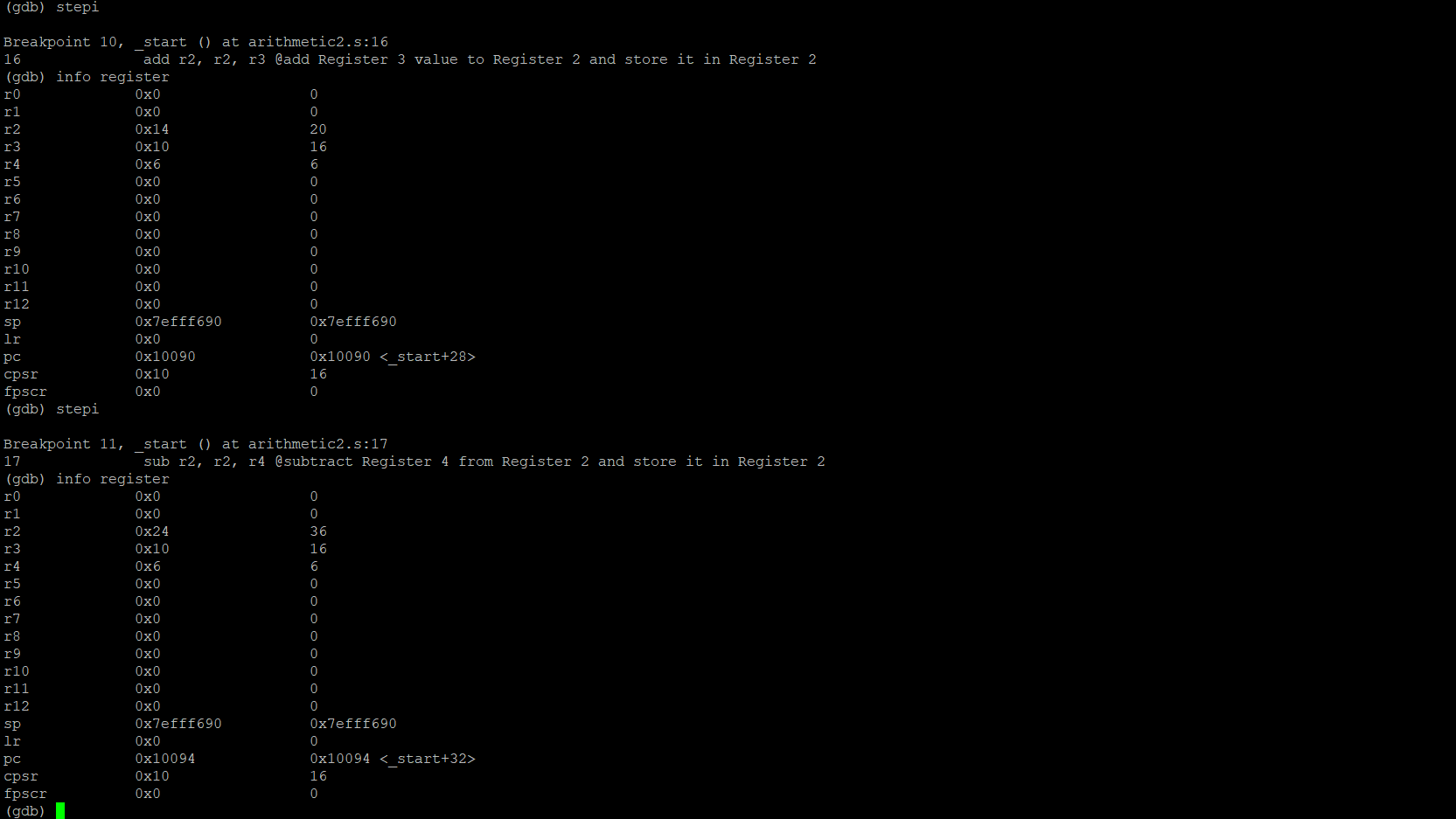
After we loaded the address of “a” into Register 1, we place the value of 11 into Register 1

After that, we load the memory address of val3 into Register 3

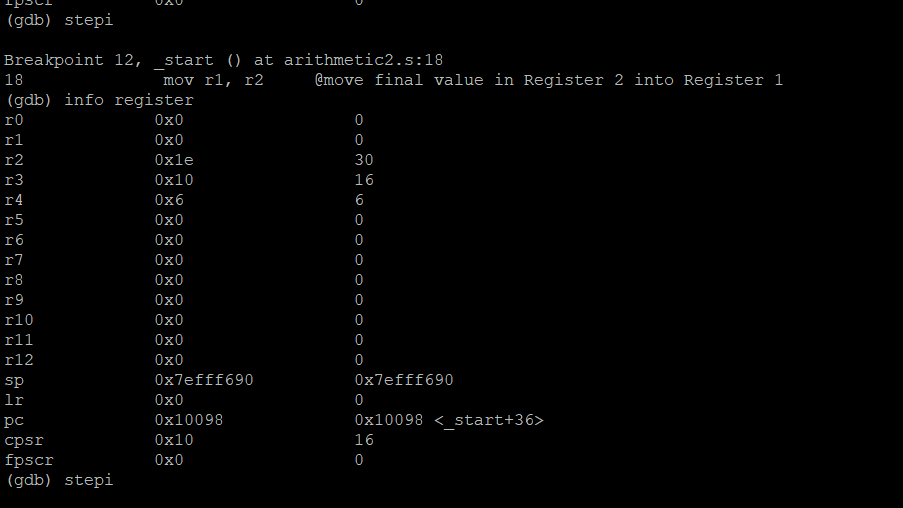
Once we run the step to the next breakpoint, we get the value of 16 placed in Register 3. During that breakpoint, we see the memory address of val1 is loaded into Register 4.



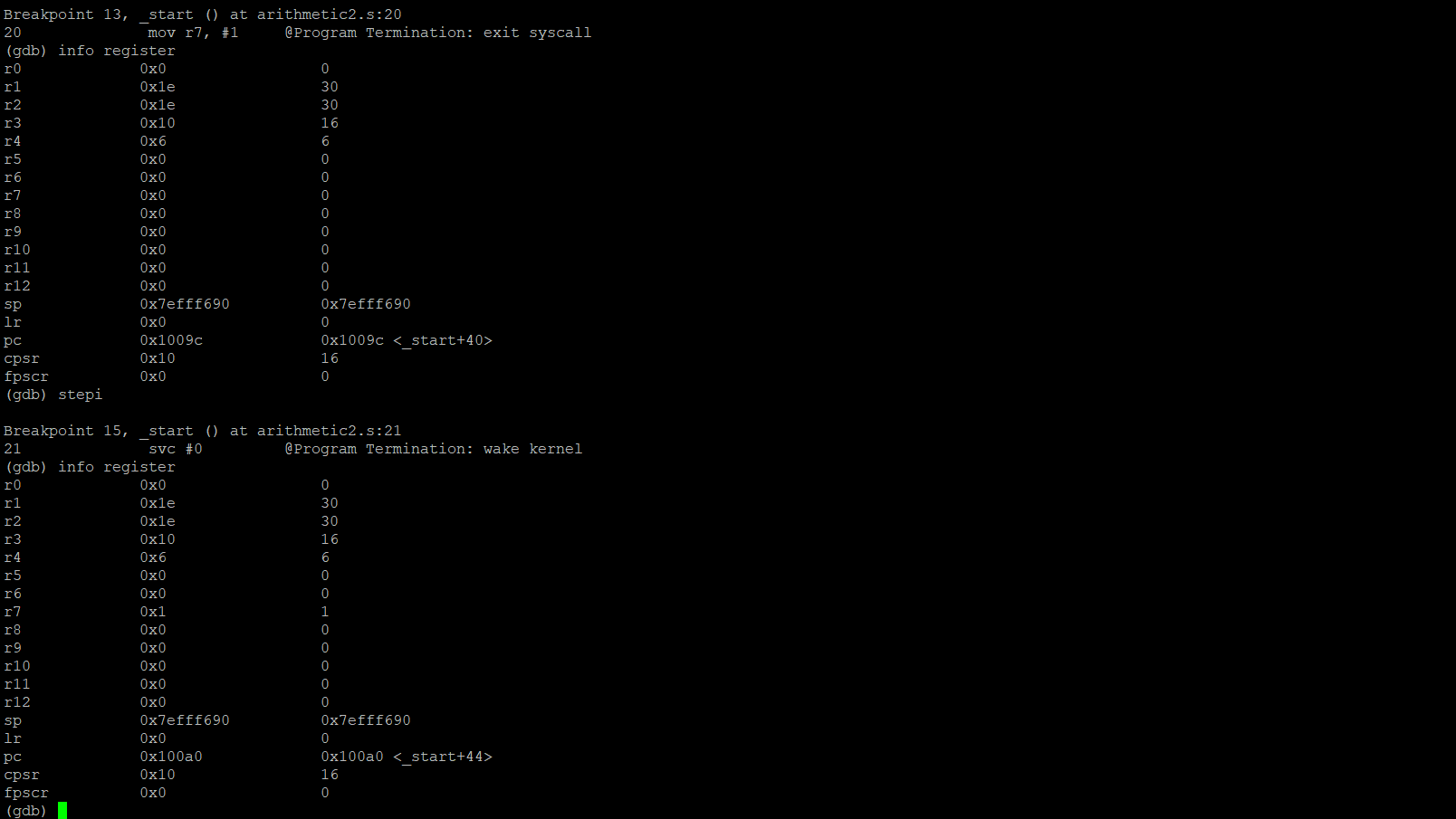
We can see the content of val1 is placed in Register 4 which is 6 as we can see above.



Next, we add the value Register 3 which is 16 and the value of Register 2 which is 20 and place it back to Register 2, overriding the previous value of 20.



Now we move the final value which is stored in Register 2 and move (copy) it into Register 1.

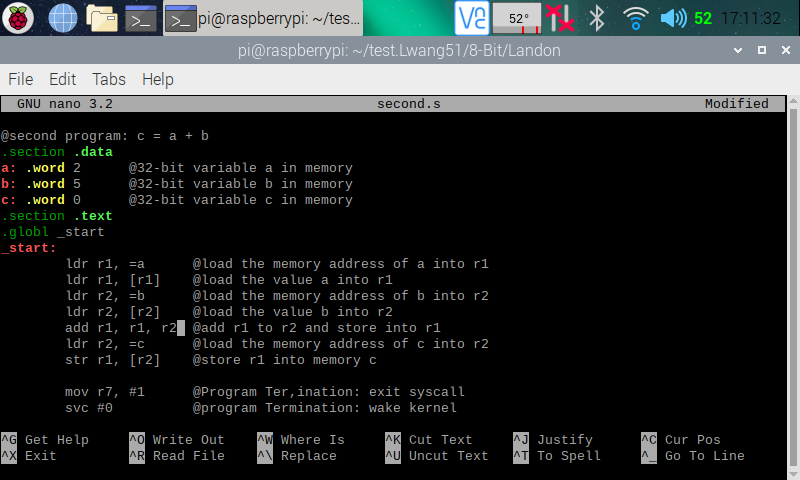


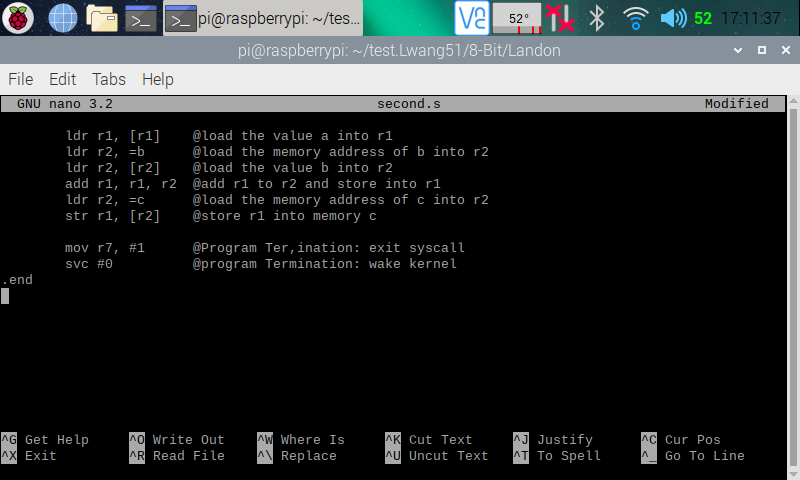
We can see the result of 30 placed in Register 1. The last code to execute is to move number (#) 1 into Register 7. I tried to use “stepi” to see if we can execute an additional breakpoint but the program ended so I exited the debugger by typing in (gdb) “quit”.

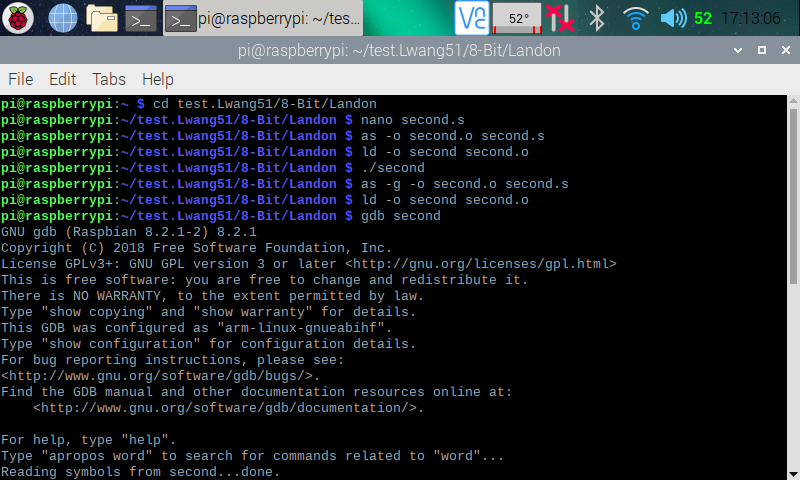
**ARM Assembly Programing**

By: Landon Wang

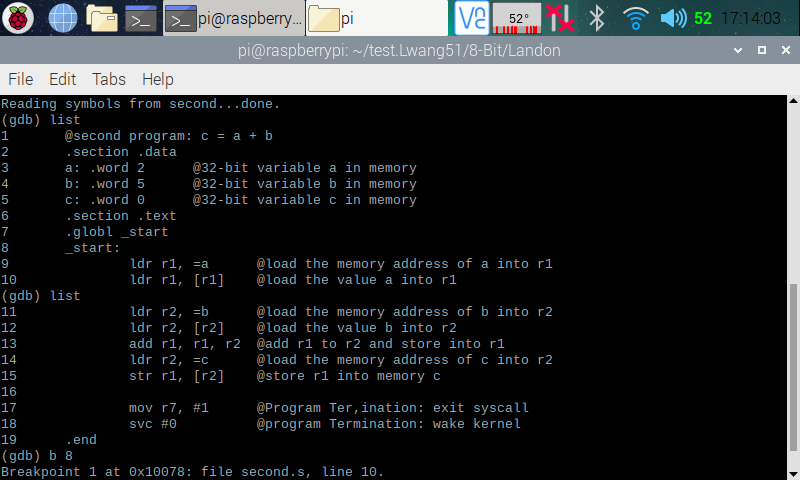
**Part One: Second Program:**



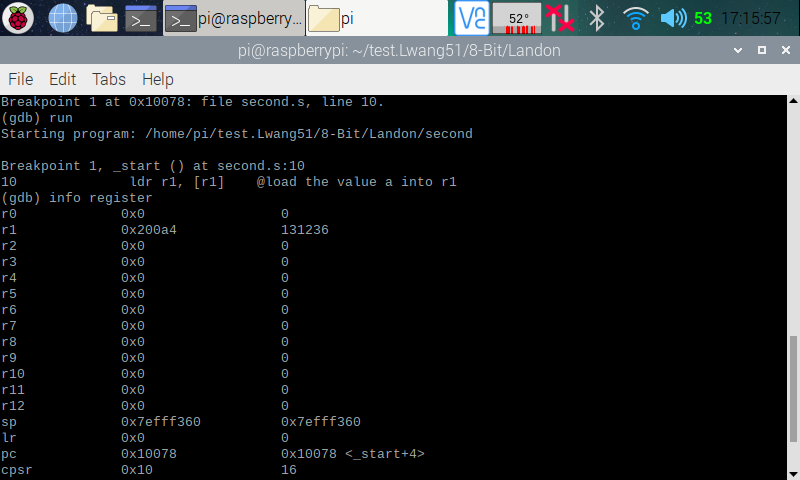




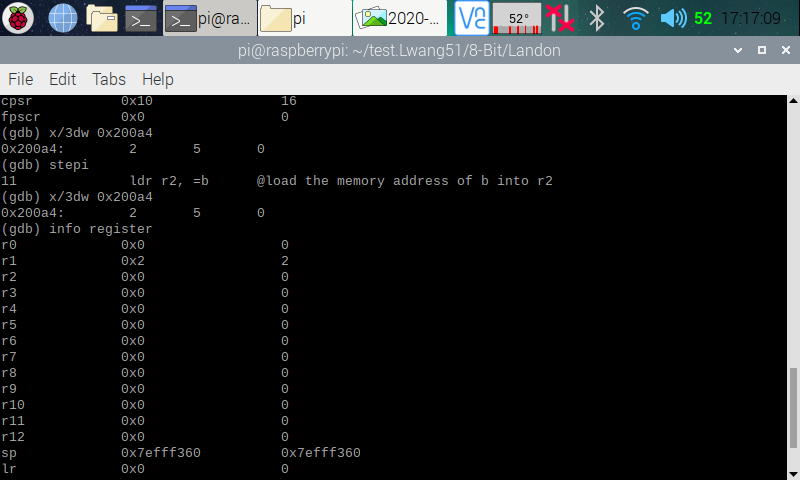
Here (in the three screenshots above), I created the Second program using the nano editor. I then assembled and linked the programs using the instructions **as -o second.o second.s** and **ld -o second second.o** respectively. After that, I ran the program using the instruction **./second,** but nothing is shown, because data was only manipulated between the CPU registers and memory. To see if my program is running correctly, I went into the debugger using the instruction **gdb second**.



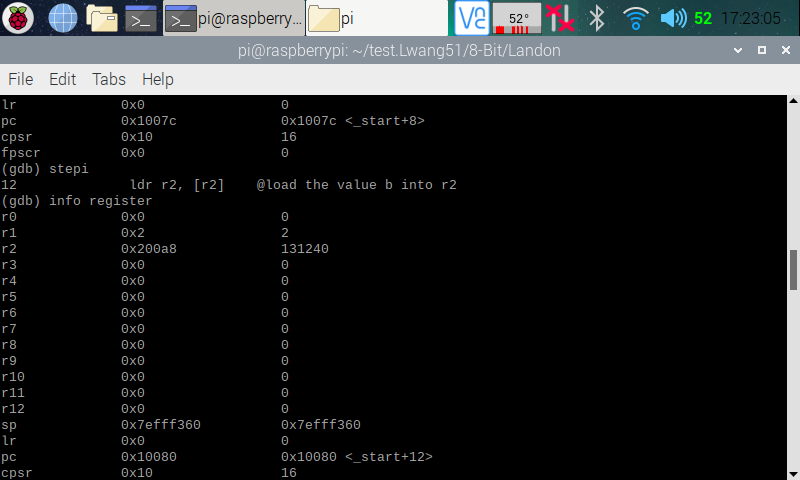
Here (in the screenshot above), I used **list** to show my program instructions, then set a breakpoint at line 8 using **b 8** (which automatically moved the breakpoint to line 10).



Here (in the screenshot above), I ran the debugger using **run**, and it stopped at line 10 where the breakpoint is set. I then pulled up the register using **info register**, and in the register, we can see the line 9 of the program executed correct, because a memory address (0x200a4) was loaded onto register 1.



Here (in the screenshot above), I went into the memory to double check if it was pointing to the right memory address using **x/3dw 0x200a4**. The memory pulled up contained 2 5 0, which was the numbers we initialized a, b, and c with. We now know that the right memory address was loaded. I then stepped over to the next line (line 11) of code using **stepi**, so that line 10 (which loaded value a into r1) will run. I pulled up the register information, and we can see that 2 have been loaded into r1.



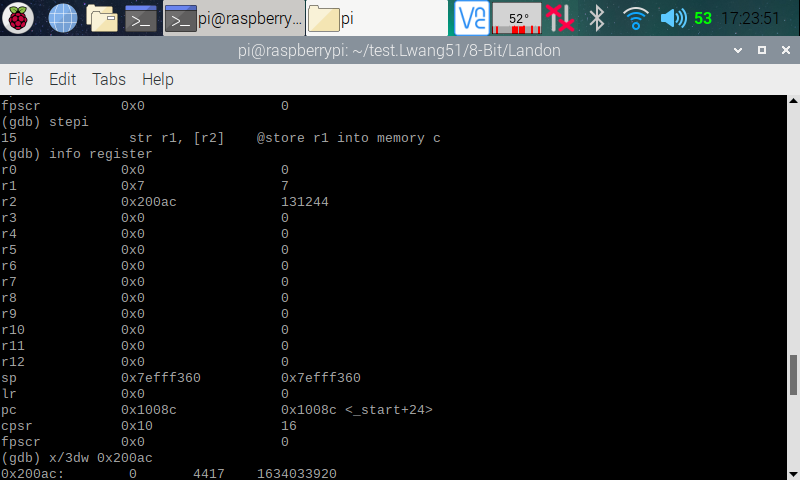
Here (in the screenshot above), I stepped over to the next line (line 12), so that line 11 (which loaded memory address of b into r2) will run. I pulled up the register information, and we can see that a memory address (0x200a8) has indeed been added into r2.



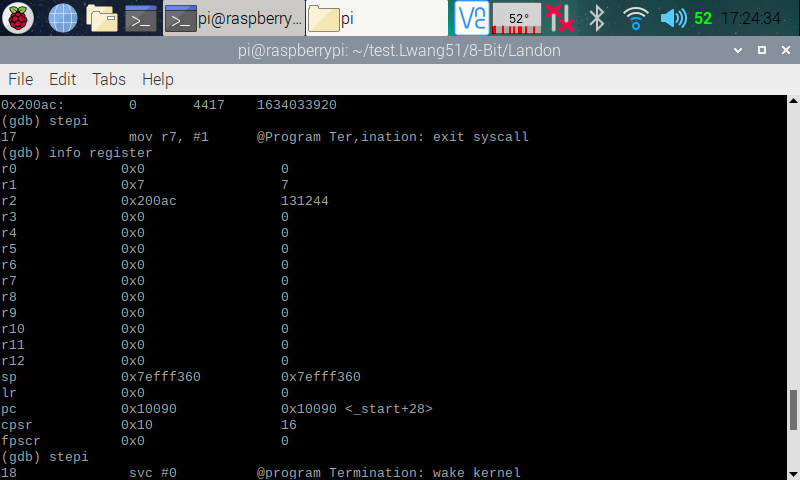
Here (in the screenshot above), I pulled up the memory that was just loaded onto r2 using the memory address (0x200a8), and we can see that it is pointing to the right part of the memory with 5 0 4417 in it. I then stepped over to the next line (line 13) of the code, so that line 12 (which loads b’s value into r2) will run. I pulled up the register information, and we can see that r2 is now loaded with 5.



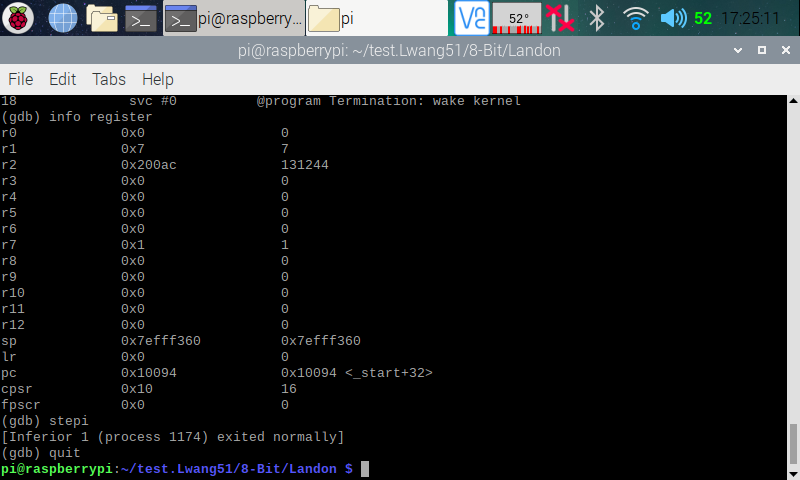
Here (in the screenshot above), I stepped over to the next line (line 14), so that line 13 (which adds r2 to r1) will run. I pulled up the register information, and we can see that 7 (from 2 + 5) is now stored in r1.



Here (in the screenshot above), I stepped over to the next line (line 15), so that line 14 (which loaded memory address of c into r2) will run. I pulled up the register information, and we can see that a memory address (0x200ac) has indeed been added into r2.



Here (in the screenshot above), I stepped over to the next line (line 17 [line 16 was blank]), so that line 15 (which stores r1 into memory c) will run.

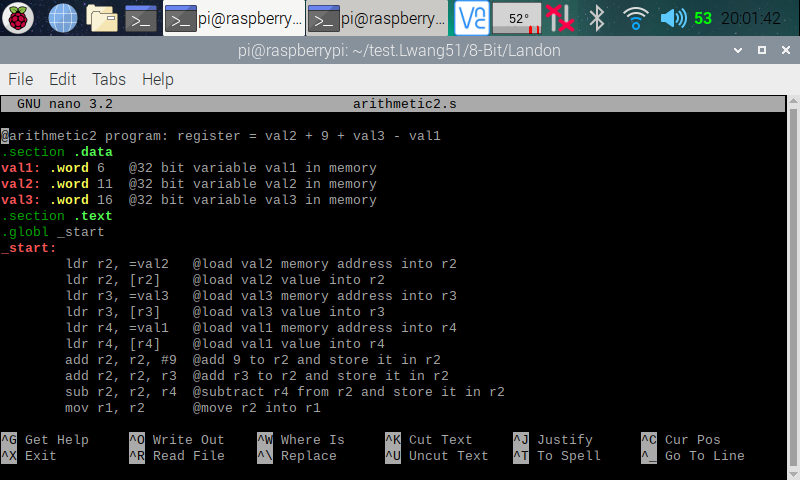


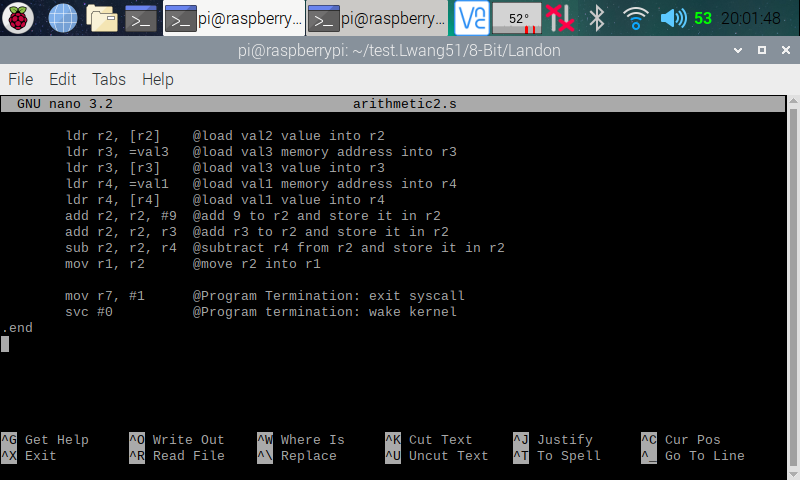
Here (in the screenshot above), I went through the rest of the debugger and exited after it went through the code.

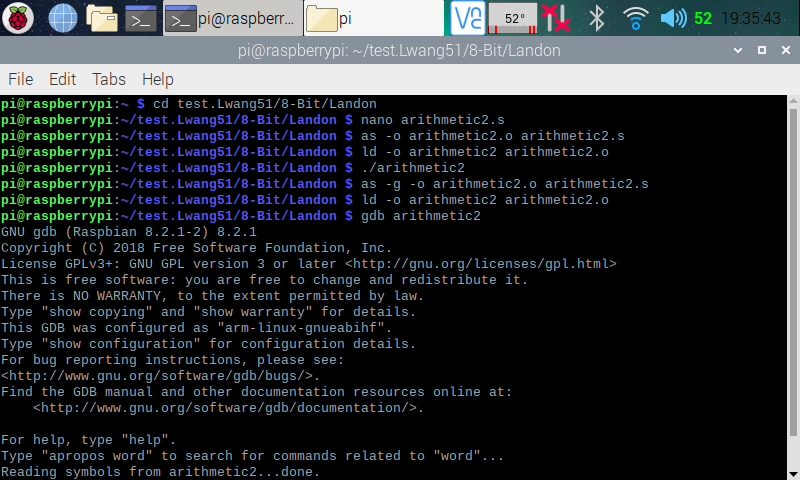


Here (in the screenshot above), I went back to the debugger and went through it again. At the end, I used the memory address of c (0x200ac) to pull it up to check if r1 was stored into memory c. In memory c, we can see that we do indeed have a 7 stored in it. We now know that our code is running correctly.

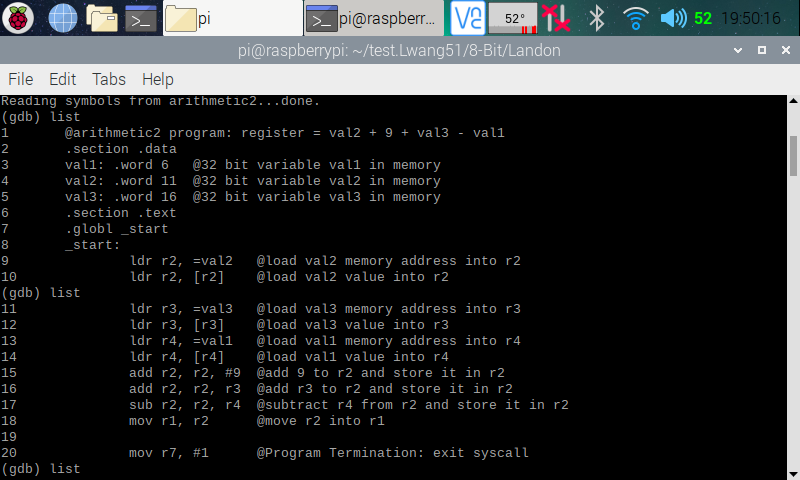
**Part Two: Arithmetic2 Program:**

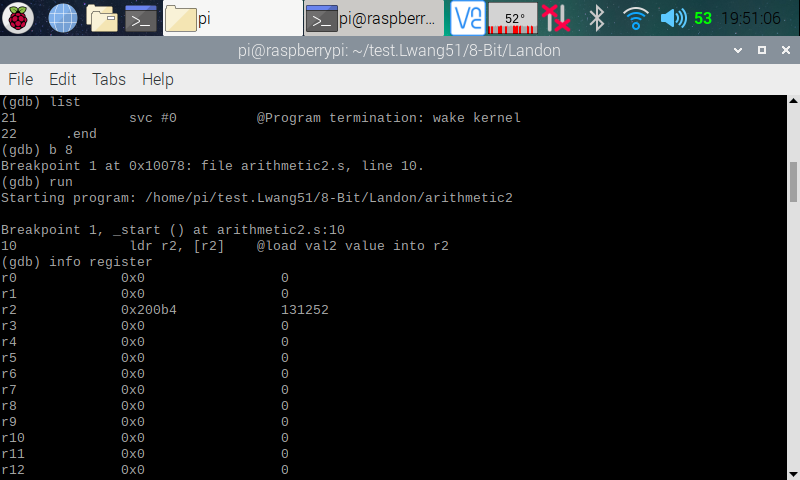




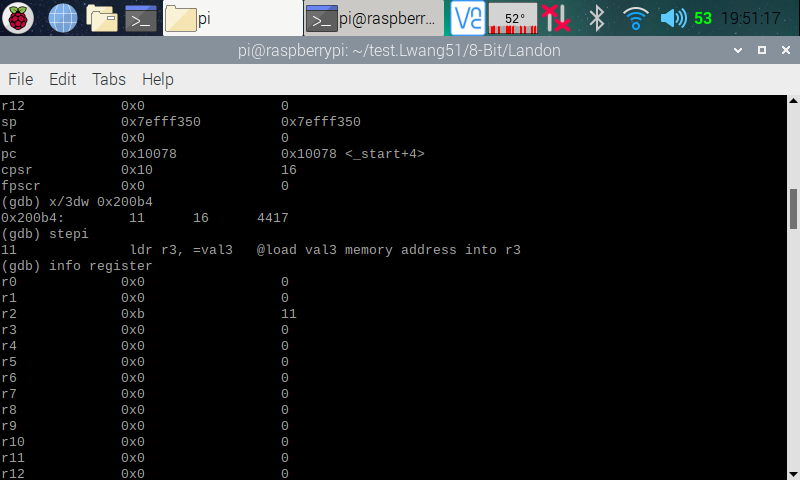


Here (in the three screenshots above), I created the Arithmetic program using the nano editor. I then assembled and linked the programs using the instructions **as -o arithmetic.o arithmetic.s** and **ld -o arithmetic arithmetic.o** respectively. After that, I ran the program using the instruction **./ arithmetic,** but nothing is shown, because data was only manipulated between the CPU registers and memory. To see if my program is running correctly, I went into the debugger using the instruction **gdb arithmetic**.

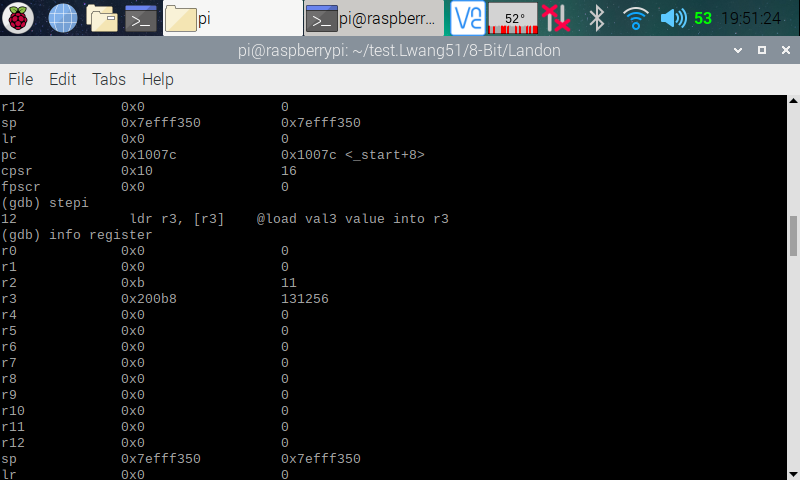




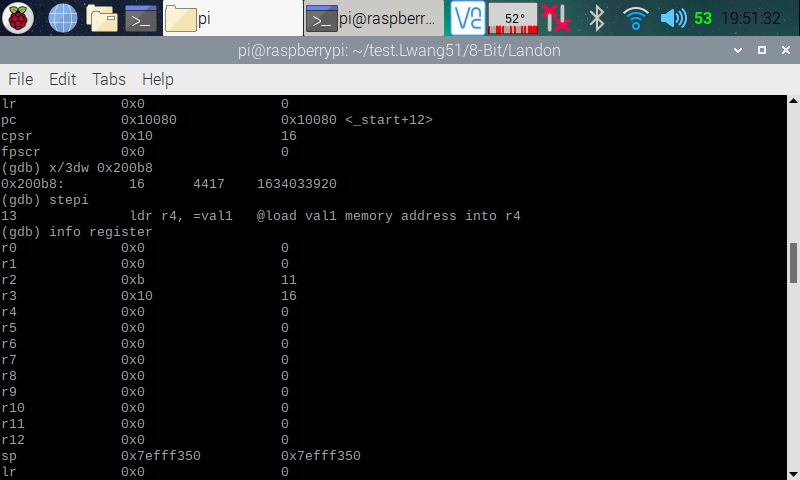
Here (in the two screenshots above), I used **list** to show my program instructions, then set a breakpoint at line 8 using **b 8** (which automatically moved the breakpoint to line 10). I then ran the debugger using **run**, and it stopped at line 10 where the breakpoint is set. I then pulled up the register using **info register**, and in the register, we can see the line 9 of the program executed correct, because a memory address (0x200b4) was loaded onto register 2.



Here (in the screenshot above), I pulled up the memory (0x200b4) using the its address, and we can see that in the memory, we have the values 11, 16, and 4417 which are values of val2 and val3. We now know that we have the right memory address. I then stepped over to the next line (line 11), so that line 10 (which loaded value of val2 into r2) will run. I pulled up the register information, and we can see that 11 is now loaded onto r2.



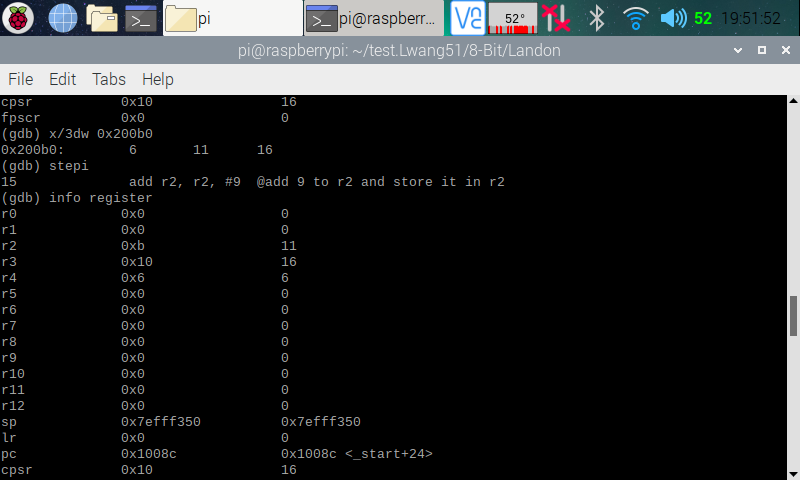
Here (in the screenshot above), I stepped over to the next line (line 12), so that line 11 (which loaded memory address of val3 into r3) will run. I pulled up the register information, and we can see that a memory address (0x200b8) has indeed been added into r3.



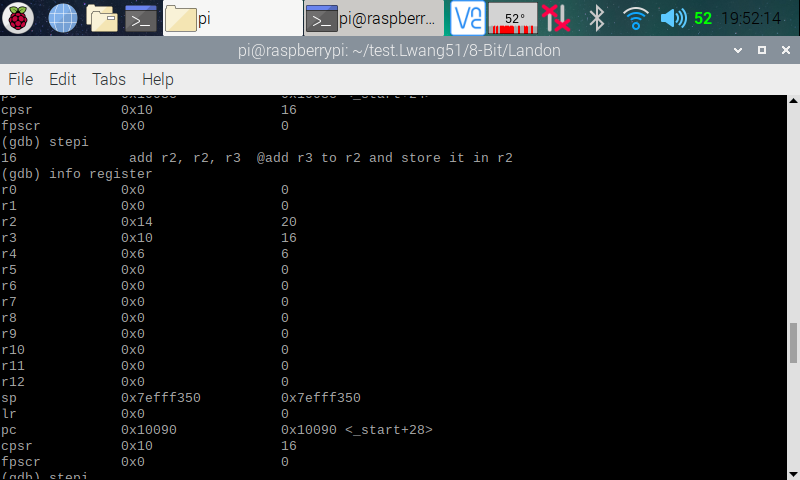
Here (in the screenshot above), I pulled up the memory (0x200b8) using the its address, and we can see that in the memory, we have the values 16, 4417, and 1634033920 which are values of val3. We now know that we have the right memory address. I then stepped over to the next line (line 13), so that line 12 (which loaded value of val3 into r3) will run. I pulled up the register information, and we can see that 16 is now loaded onto r3.



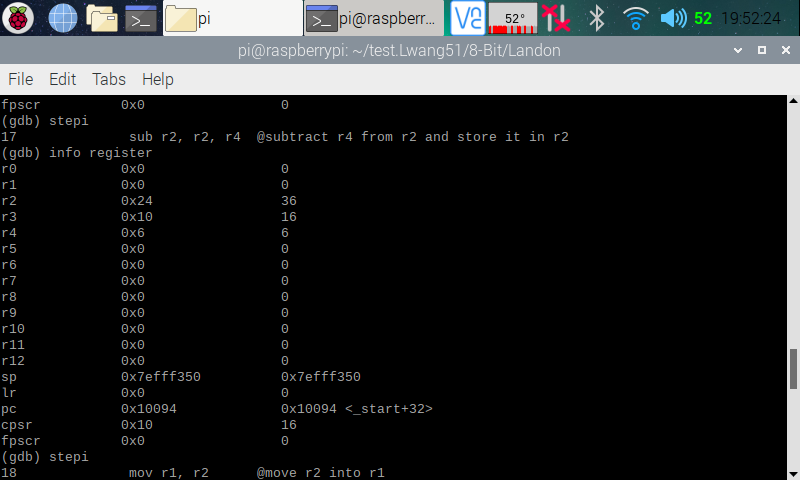
Here (in the screenshot above), I stepped over to the next line (line 14), so that line 13 (which loaded memory address of val1 into r4) will run. I pulled up the register information, and we can see that a memory address (0x200b0) has indeed been added into r4.



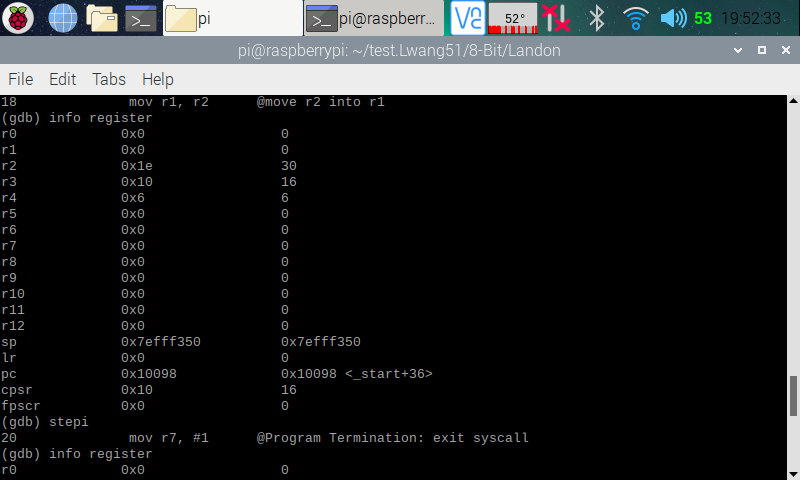
Here (in the screenshot above), I pulled up the memory (0x200b0) using the its address, and we can see that in the memory, we have the values 6, 11, 16 which are values of val1, val2, and val3. We now know that we have the right memory address. I then stepped over to the next line (line 15), so that line 14 (which loaded value of val1 into r4) will run. I pulled up the register information, and we can see that 6 is now loaded onto r4.



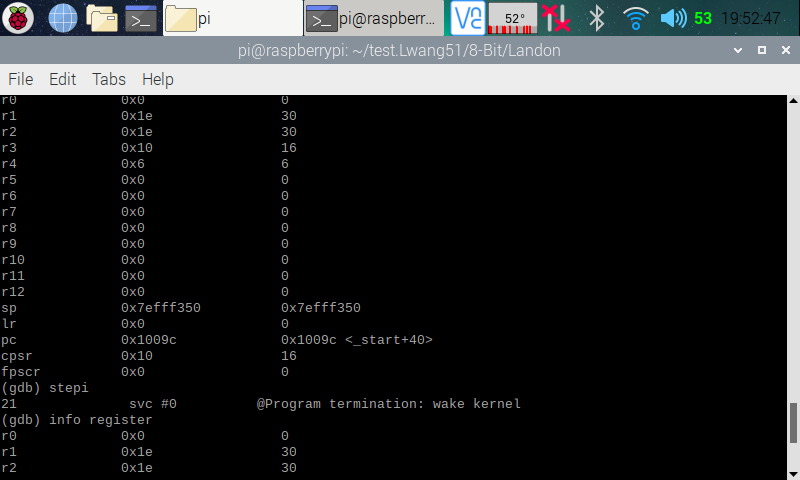
Here (in the screenshot above), I stepped over to the next line (line 16), so that line 15 (which added 9 to r2) will run. I pulled up the register information, and we can see that 20 (11 + 9) is now stored in r2.



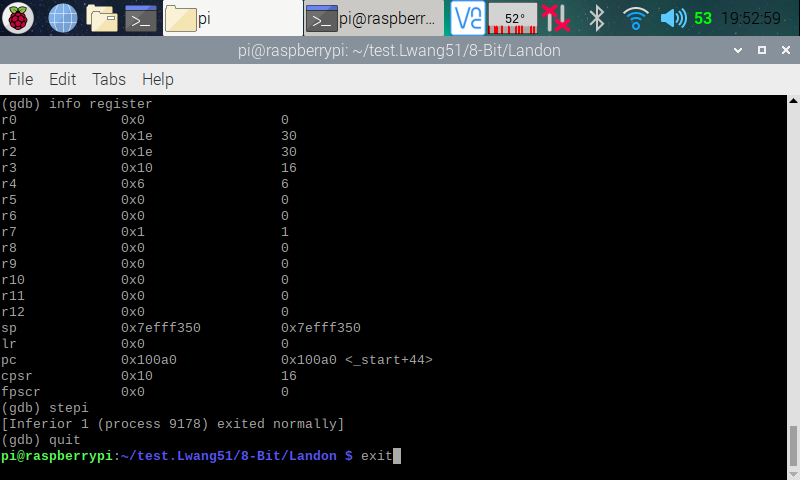
Here (in the screenshot above), I stepped over to the next line (line 17), so that line 16 (which added r3 to r2) will run. I pulled up the register information, and we can see that 36 (20 + 16) is now stored in r2. I then stepped over to the next line (line 18), so that line 17 (which subtracted r4 from r2) will run.



Here (in the screenshot above), I pulled up the register information, and we can see that 30 (36 - 6) is now stored in r2. I then stepped over to the next line (line 20 [line 19 was blank]), so that line 18 (moving r2 to r1) will run.



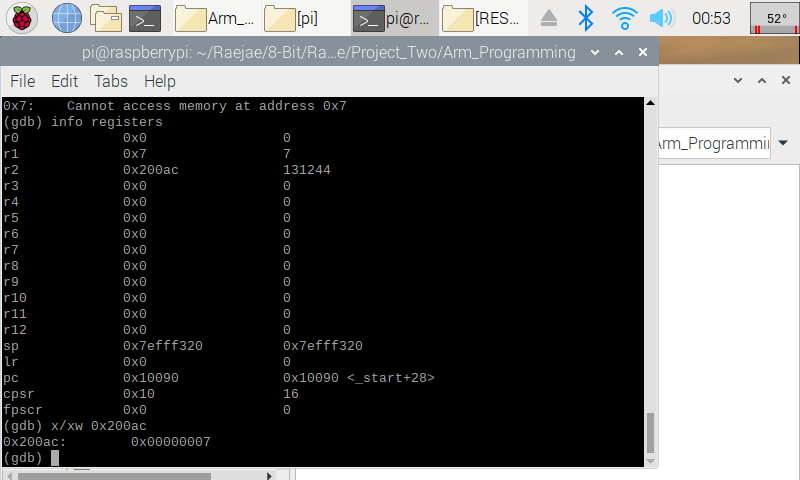
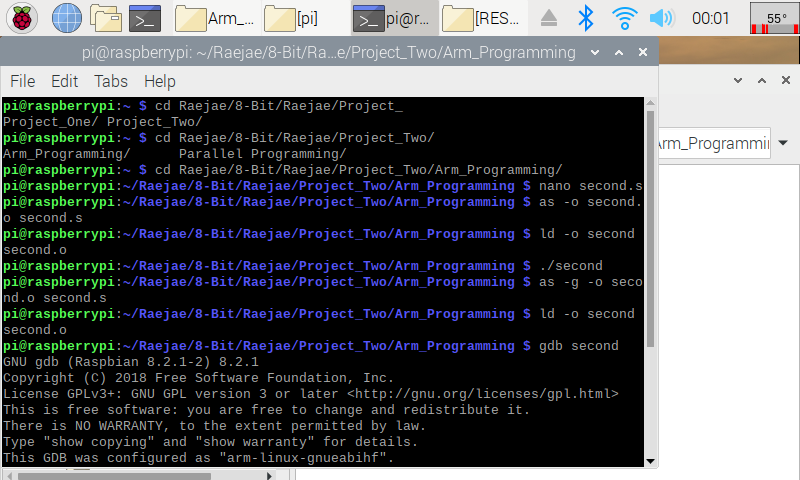
Here (in the screenshot above), I pulled up the register information, and we can see that 30 is now stored in r1. I then continued to step over to the next line until I reached the end of the program.



Here (in the screenshot above), I went through the rest of the debugger and exited it after it went through the code. I then exited the terminal.

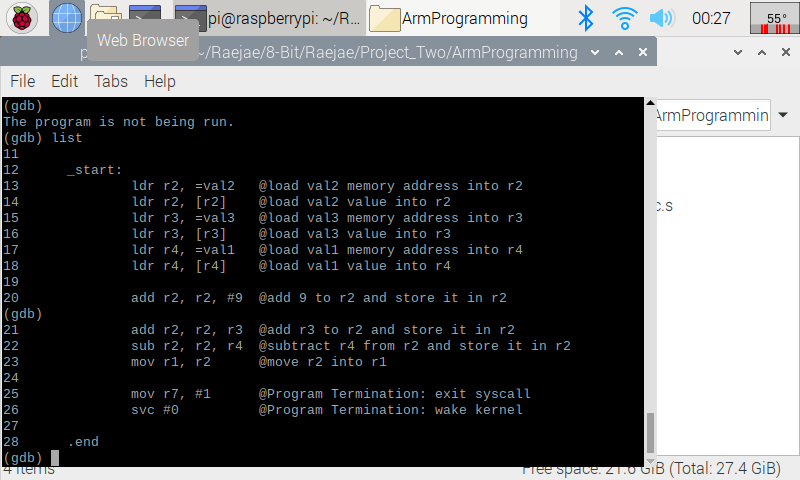
**ARM Assembly Programming**

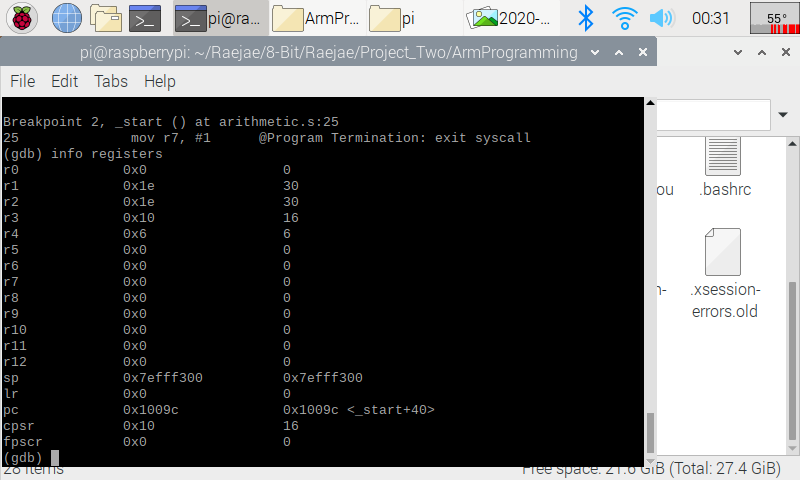
By: Raejae Sandy



The setup for this code was very similar and its explanation is straightforward as we use register manipulation and addresses to display a word with hexadecimal. We introduced new concepts this time by stepping into breakpoints, and using address searches to locate numbers. In the code after running through the functions we run x/xw to tell our code to locate the hexadecimal in address 0x200ac. The address was found by searching (info registers) and then we confirmed that the value is indeed 7.

**Aritmetic2 Program**





In this code we implemented the same functions, but the interesting concepts is how we loaded the variables into the registers to carry out our operations. By stepping into the final breakpoint, we can see the output sets to 30 placing it into the first register after storing in each and placing the operations into registers we finalize our code by setting the result in r1.

**ARM Assembly Programming**

By: Tony Ngo

Tony Ngo – Project A2 Task 4

A close up of text on a black background

Description automatically generated Picture 1: Compiling, assembly, execution, and creating the debugger of “second” program

A close up of text on a black background

Description automatically generated Picture 2: Debugging of “second” program

The “second” program is formatted very similar like the “first” program, except for the fact that we are storing memory values instead of loading values into just registers. To write the program itself, I used the nano text editor within the terminal. To assemble the program, I used “***as -o second.o second.s***”, which creates an objective file named “second.o”. To link the program, I used “***ld -o second second.o***”, which creates an executable called “second”, however you do not see output when “***./second***” is ran because all the values are loaded into the registers, which we will need to use the GDB debugger to see the values. To assemble a program that can be used by GDB, you will need to add the “-g” flag, which makes the assembly of the program now “***as -g -o second.o second.s***”. The linking will be the same, but the executable can now be ran into GDB as “***gdb second***”. In the debugger, I first typed “list” to show all of the code that I wrote, then I added a flag at 15, by entering “***b 15***”. To look into the actual memory at the breakpoint, I typed “***x/3xw 0x1008c***”, which is the memory value where my breakpoint is located. “***x/3xw***” indicates that it is going to display three words in hexadecimal, “***0x1008c***” is the location where my breakpoint is located. These values served as a reference so we knew where our code was stored to.

While writing the program, I did not run into any complications in terms of errors since I did most of my syntax-based errors during Project 1. However, I forgot to do “#9” instead of “9” because of the syntax in x86 vs ARM and it gave me compiler errors.

A close up of text on a black background

Description automatically generated Picture 1: Compiling, assembly, and execution of Arithmetic2

\*insert pic of debugging\*

A close up of text on a black background

Description automatically generatedPicture 2: Debugging of Arithmetic2

In this program, I stored values into “val1”, “val2”, and “val3” and accessed them by loading them into registers r1, r2, and r3 and computed the values using the registers and stored the final value into r4. I wrote the program using nano in my terminal. To assemble this program, I used the command “***as -o arithmetic2.o arithmetic2.s***”, which creates an objective file called “arithmetic2.o”. I linked my program using the command “***ld -o arithmetic2 arithmetic2.o***”, which creates an executable called “arithmetic2”, but to be able to debug it using the GDB debugger, we need to reassemble using “***as -g -o arithmetic2.o arithmetic2.s***”, and we have to relink it using “***ld -o arithmetic2 arithmetic2.o***”. In the debugger, I used “list” to show all the code that I wrote. To look into the actual memory at the breakpoint, I typed “***x/3xw 0x1008c***”, which is the memory value where my breakpoint is located. “***x/3xw***” indicates that it is going to display three words in hexadecimal, “***0x1008c***” is the location where my breakpoint is located. These values served as a reference so we knew where our code was stored to. I added a flag at 26 using “b 26”, this was to check the registers after my code executed to see if I did my arithmetic right, which I did because r4 has 30 loaded into it and the arithmetic equation would be “11 + 9 + 16 – 6”, which is equal to 30.

While writing the program, I did not run into any complications in terms of errors since I did most of my syntax-based errors during Project 1.

Appendix

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

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

Presentation on YouTube: <https://youtube.com/watch?feature=youtu.be&v=H4_dVGlscE0>

Screenshot of Slack Chat: 