

**Faculty of Engineering and Applied Science**

**SOFE 4590U Embedded Systems**

**Group 6 CRN 74020**

**Lab 1**

| **Name** | **Student #** |
| --- | --- |
| **Alexander Campbell** | **100703650** |
| **Atharshan Kennedy** | **100590243** |
| **Nicole Okeke** | **100769799** |
| **Tiwaloluwa Ojo** | **100700622** |
| **William Robinson** | **100751756** |

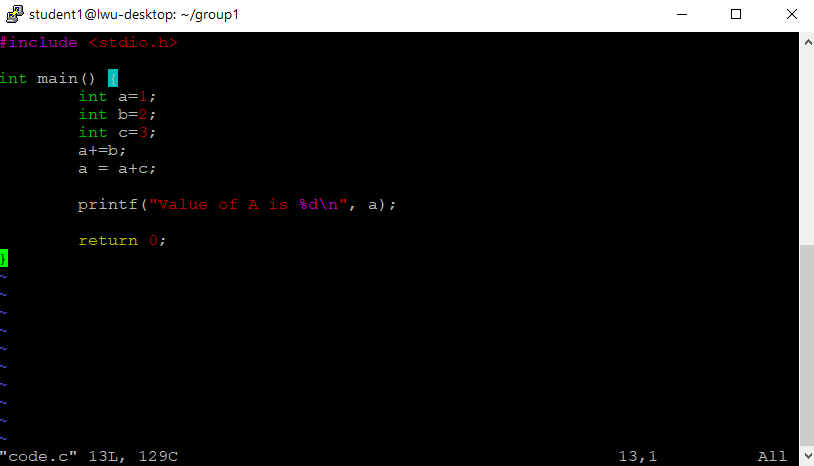
**Introduction:**

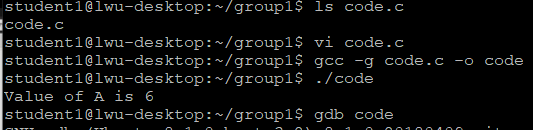
In this lab we experimented with programming C in an embedded software environment using the NVIDIA Jetson TX2 development kit. Through this experiment we gained more knowledge in both the GNU Debugger (GDB) and how memory works in C. In C, and many other languages, there is a data collection system known as the “stack”. The stack is responsible for memory tracking when the application is executing, therefore it is deleted upon completion or stopping of execution. There are only two operations that a stack will perform, a push and a pop. A push is adding an entity on top of the stack, as you cannot insert memory into a stack below the very top. A pop is removing the upper most entity on the stack.

**Lab Tasks:**

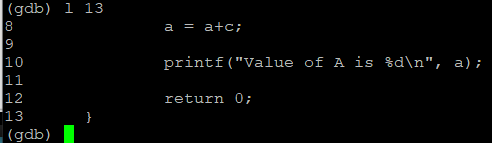
We had two methods of accessing the Jetson TX2 terminal environment. One is the traditional method utilized in the lab instructions, and the other was using Visual Studio to SSH into the board.

*Method 1:*

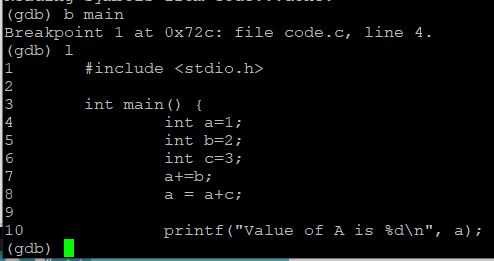
*C program code*

**

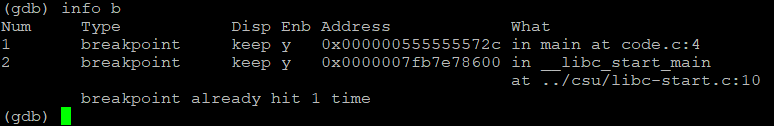
*Successfully compiling and executing the C program*



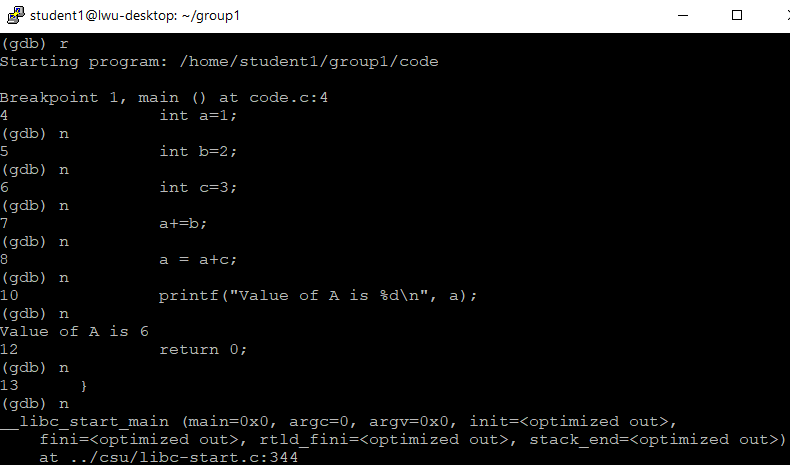
Beginning the debugging process



*Creating a breakpoint at line 1 in the main function*

**

*Viewing breakpoint*

**

*Results of stepping through the program, reruns the value of A*

*Method 2:*

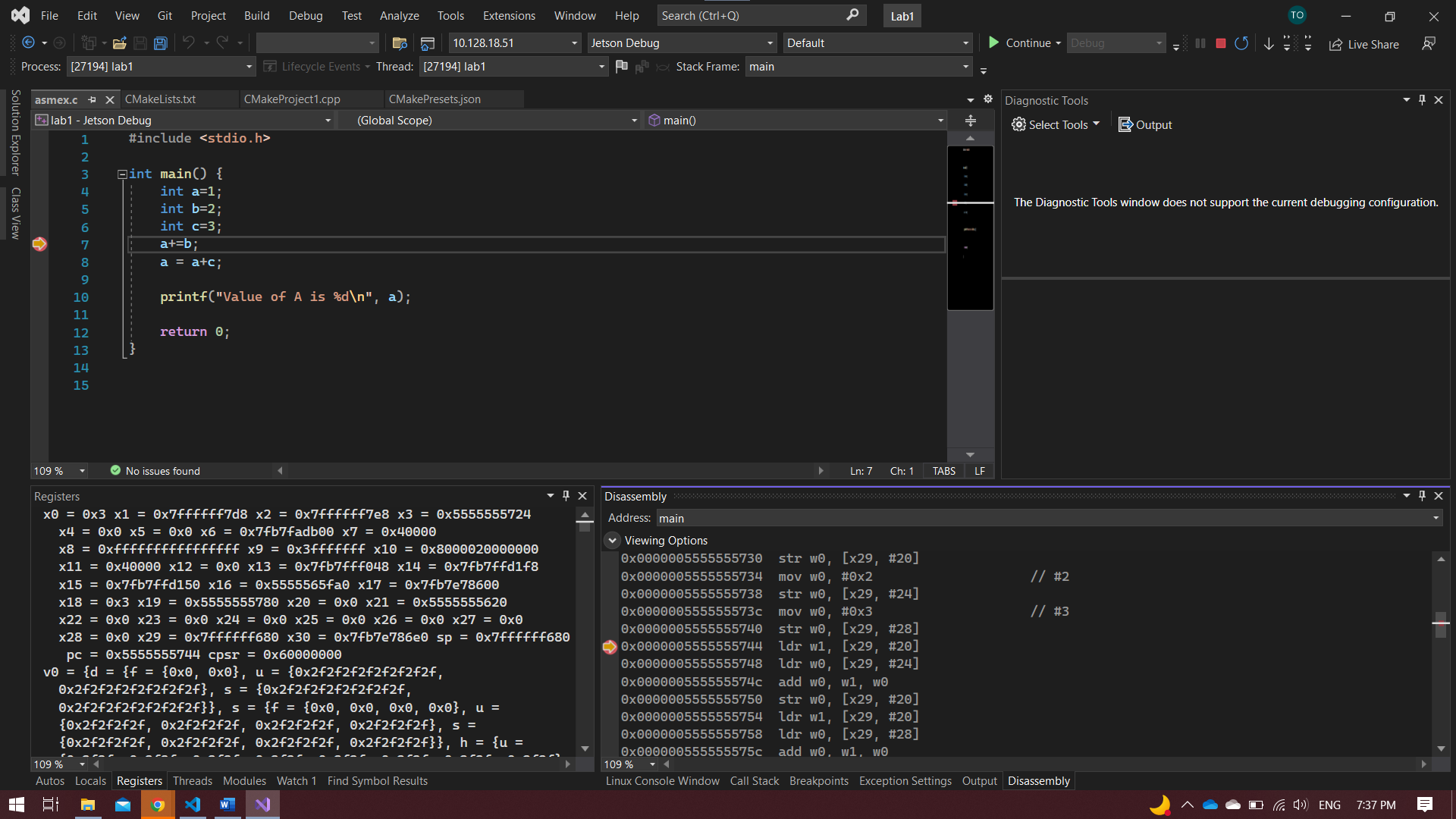
Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, text, application

Description automatically generated

*Utilizing Visual Studio’s built-in SSH connection to connect to the board*

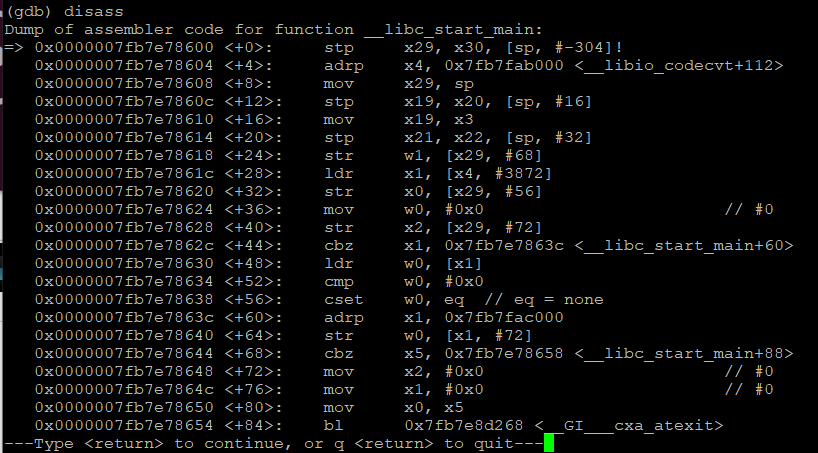


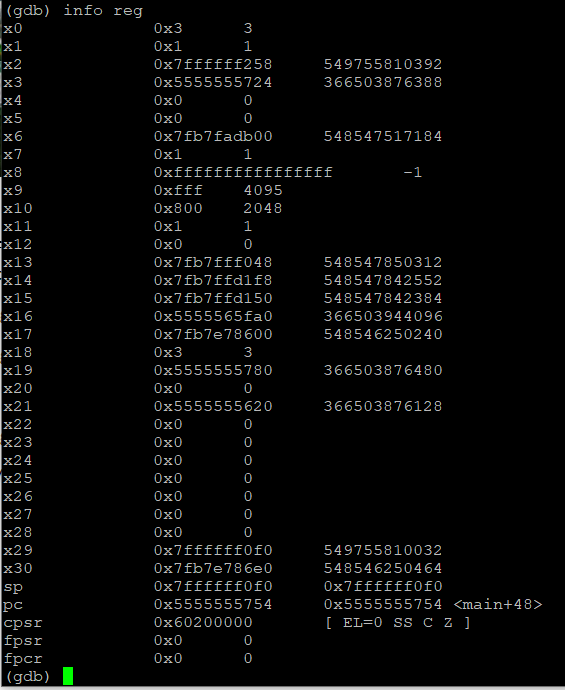
*Successfully entering the TX2 environment and using Visual Studio to create C files*

*Visual Studio downloads CMake onto the board & syncs the source files*

After successfully connecting to the TX2 environment using either method, we can run the files using GDB and disassemble them in either environment. We can also view registers after the program has run.

Graphical user interface, text

Description automatically generated



**Disassembled Code Explanation:**

A screenshot of a computer

Description automatically generated with medium confidence

***push %rbp:*** Pushes the rbp, which is a pointer that points to the base of the stack, onto the call stack. **PUSH OPERATION**

***mov %rsp, %rbp:*** Copies the value of the rsp, which is the pointer that points to the top of the stack, into the base register. This makes sense as the stack is currently empty so the base pointer and top pointer both point to the same place.

**sub $0x10, %rsp:** Subtracts the value of the %rsp from the register 0x10. A substituted instruction can happen sometimes, as stated in the ARM documentation.

***movl $0x1, -0xc(%rbp):***Copies the value of 1 into the register that is at the address of %rbp offset by -0xc. This register holds the value of **int a**.

***movl $0x2, -0x8(%rbp):***Copies the value of 2 into the register that is at the address of %rbp offset by -0x8. This register holds the value of **int b**.

***movl $0x3, -0x4(%rbp):***Copies the value of 3 into the register that is at the address of %rbp offset by -0x4. This register holds the value of **int c**.

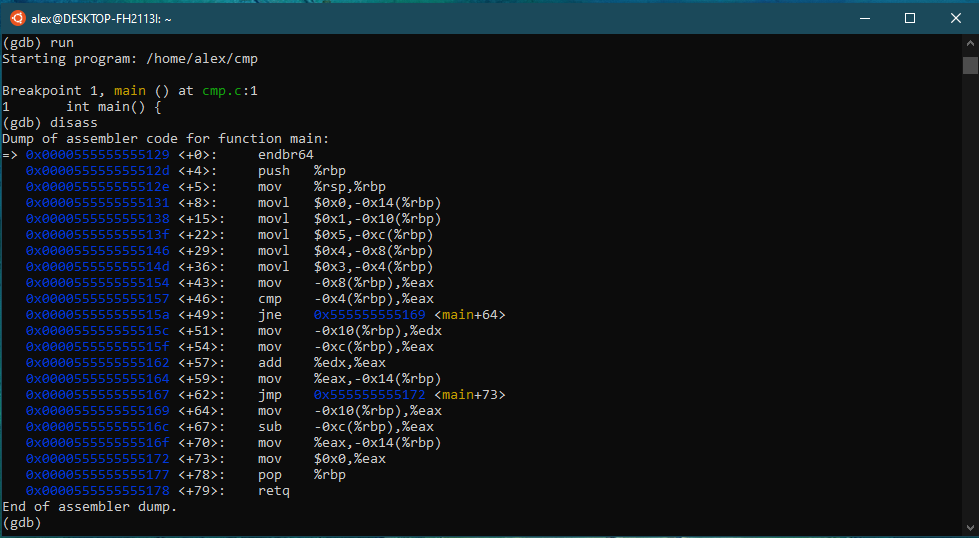
***mov -0x8(%rbp), %eax:*** Copies the value stored at -0x8(%rbp), which is int b, into the %eax register, which is a general purpose register.

***add %eax, -0xc(%rbp):*** Adds the value stored at %eax to -0xc(%rbp), which is adding of the value of the general purpose register to int a. This is **a += b**, the value is then stored in the next instruction.

***mov -0x4(%rbp), %eax:*** Moves the value of int c into the general purpose register.

***add %eax, -0xc(%rbp):*** Adds the value of the general purpose register onto the value stored at -0xc(%rbp), which is holding the value of a. This is equivalent to the **a = a+c** part of the code.

The rest of the assembly code is used to print the value that has been located with the string that was stored in the buffer at compile time.

**cmp.c File Disassembly Comparison**

A screenshot of a computer

Description automatically generated with medium confidence

In this side-by-side comparison of the cmp.c file being disassembled in Ubuntu (left image) and in the Jetson TX2 environment (right image) there are a few immediate differences. In Ubuntu, the memory addresses are all directly initialized one by one using the movl command at the beginning of the program before moving onto using the general purpose register to do most of the arithmetic. In the Jetson TX2, we see mainly the using of 2 registers w0 and w1 to hold the immediate values, which are then stored into memory addresses after. One similarity between the two in this is that to store at a specific address, they always use the stack pointer offset by some immediate value. In Ubuntu, this immediate value offset is a negative hexadecimal while in the Jetson TX2 this immediate value is a positive integer.