Name: Minh Binh Nguyen

PantherID: 002-46-4288

**Task 4 Report**

**Part 1: fourth program**

I started with getting my Raspberry Pi connected to my computer, then I typed in the code, assembled and linked the program as instructed (Figure 1).

A screenshot of a social media post

Description automatically generated

Figure 1

As I began, I checked the memory of x and y at the addresses being loaded into r1. They are 0 as expected. After executing the cmp instruction, I went ahead and checked the flags. On the cpsr, we can see that the bits 29th and 30th (count from right to left) are turned on. They are carry flag and zero flag. This makes sense because subtracting zero from zero raises these two flags, which also means they are equal numbers. (Figure 2).

A screenshot of a cell phone

Description automatically generated

Figure 2

After that, as we expected, the program continued executing the “thenpart”, resulted in assigning 1 to y. Then, I checked the memory of y, it has the value of 1 (Figure 3).

A screenshot of a cell phone

Description automatically generated

Figure 3

**Part 2: Making fourth program efficient**

The program in part 1 is not efficient because the code contains back-to-back branches (beq followed by b). I then updated the program as below so that it only jumps when the numbers are not equal (Figure 4).

A screenshot of a social media post

Description automatically generated

Figure 4

To prove it works as it should, I debugged it using gdb. Firstly, I checked the memory of x and y before doing anything. They hold zero values as expected (Figure 5).

A screenshot of a cell phone

Description automatically generated

Figure 5

After executing the cmp instruction, the flag’s report is the same as part 1 program, and the program doesn’t jump to the “endofif” part. This results in assigning 1 to y as it’s supposed to do so (Figure 6).

A screenshot of a cell phone

Description automatically generated

Figure 6

**Part 3: Control Structure program**

This is my code for this program to calculate the given expression (Figure 7).

A screenshot of a cell phone

Description automatically generated

Figure 7

To know whether the program was written correctly or not, I went ahead and debugged it using gdb. I set the breakpoint at the very beginning to examine the memory before doing anything. Using the address that is loaded to r1, we can see that x holds the value of 1 (Figure 8).

A screenshot of a cell phone

Description automatically generated

Figure 8

After comparing x with 3, I checked the flags. The 31st bit is turned on. This is the negative flag. The negative flag is raised because 1 is less than 3, so when subtracting 3 from 1, we get a negative value. After that, the program subtracts 1 from x, making it 0. I checked the memory before exiting the program, and it shows a correct value (0) (Figure 9).

A screenshot of a cell phone

Description automatically generated

Figure 9