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
```

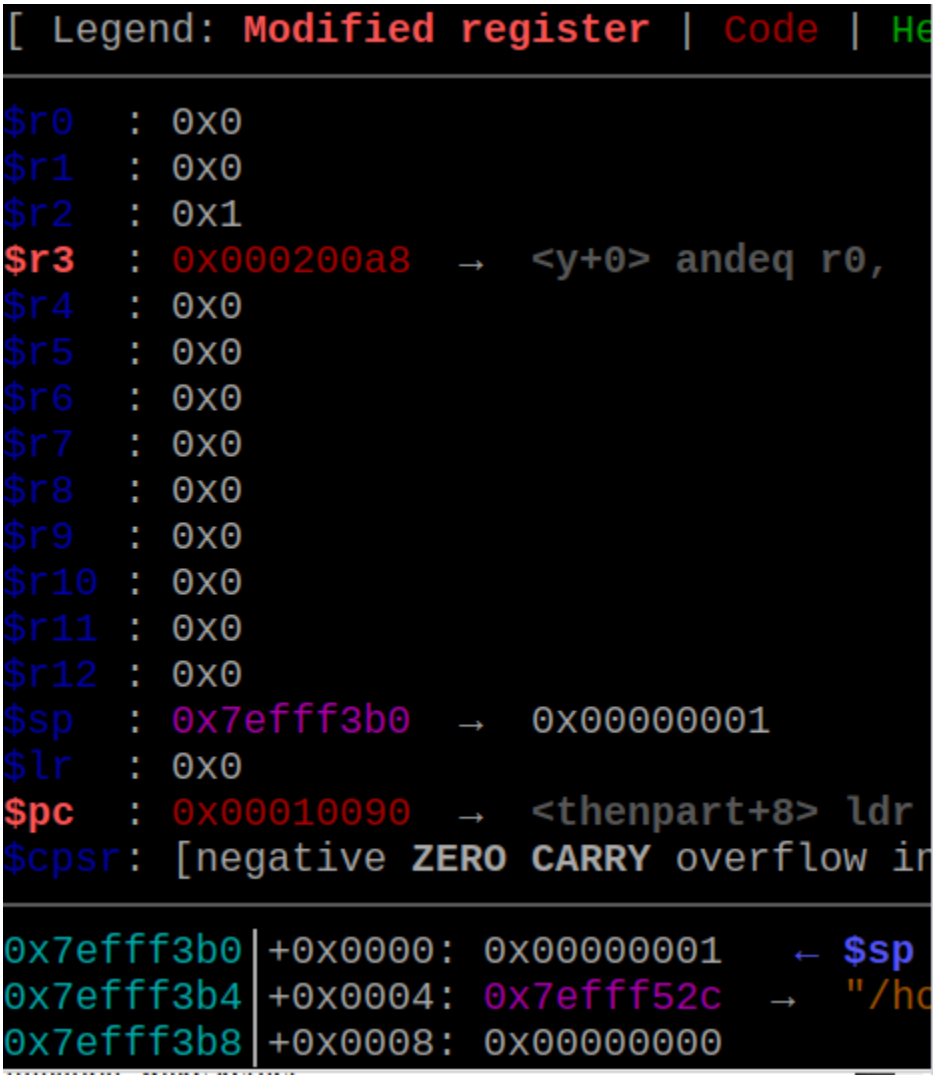
We assembled and linked the program, then launched the gdb debugger for fourth.

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

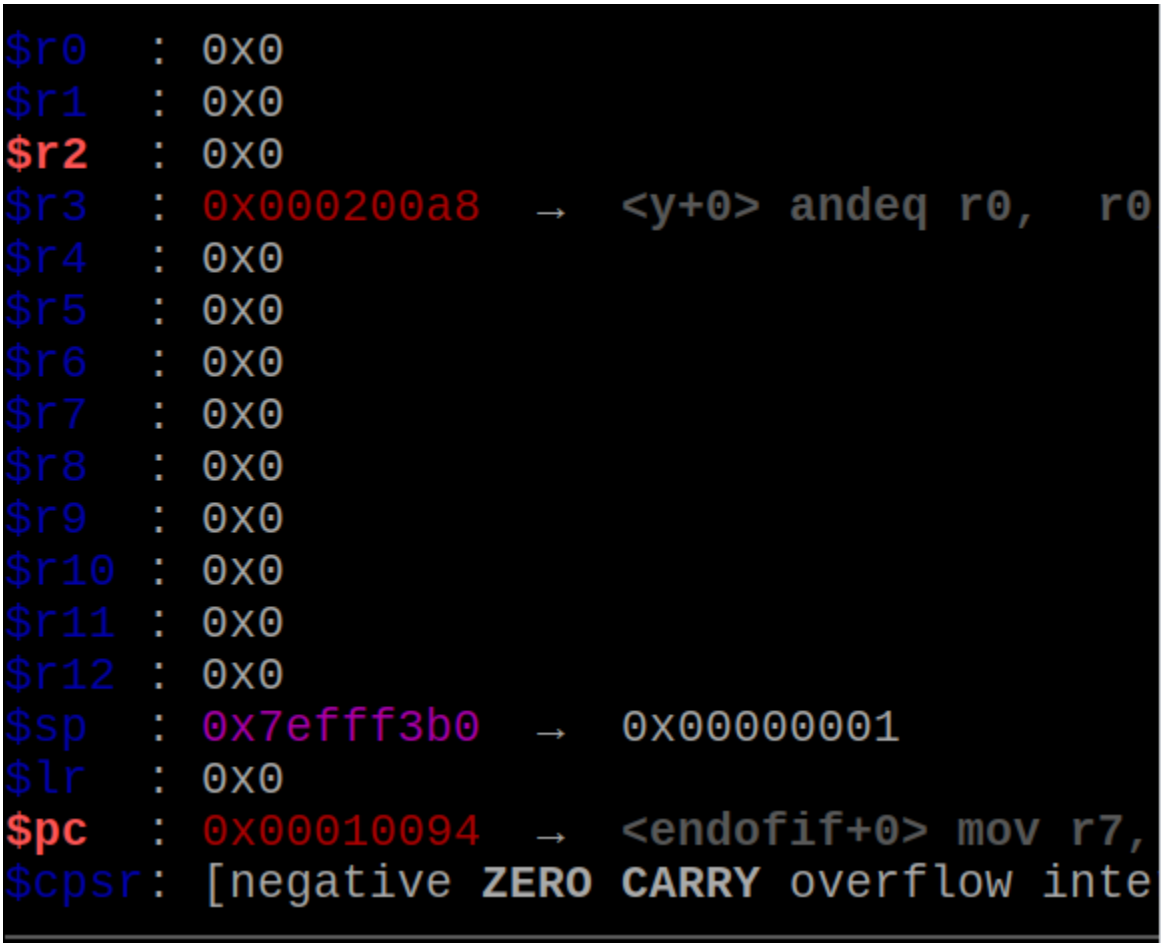
```
gef➤  list
1         .section .data
2         x: .word 0
3         y: .word 0
4         .section .text
5         .globl _start
6         _start:
7             ldr r1,=x
8             ldr r1,[r1]
9             cmp r1,#0
10            beq thenpart
gef➤
```

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).**

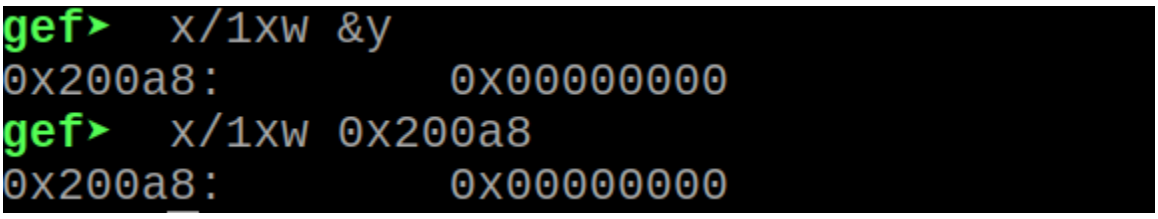
The next line executed is the **beq thenpart** which indicates that we will jump to the statement **thenpart** if the zero flag is equal to 1, in case we do jump to the thenpart because the zero flag is indeed equal to 1; therefore, the next line to be executed when we stepi is line 13 which indicated 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
```

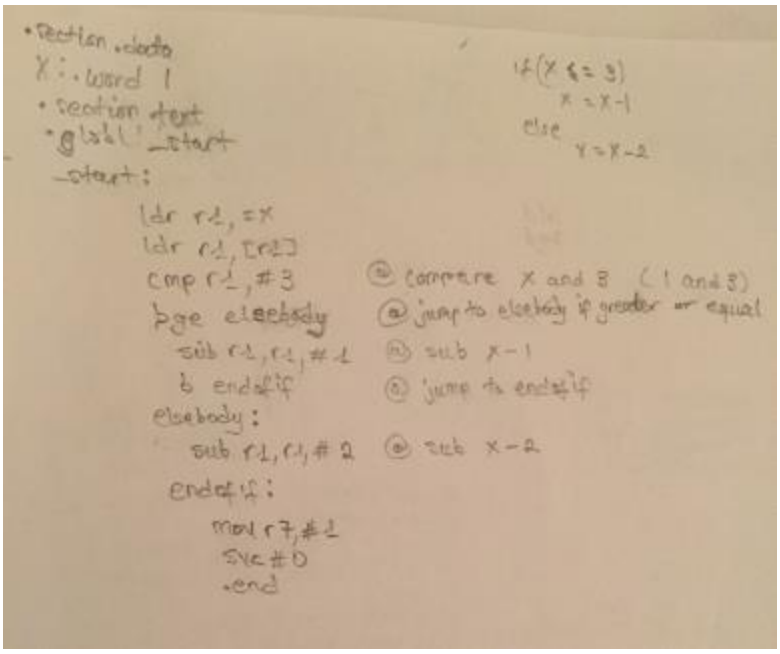
The code produced the same output as the code in part 1 because they are the same code, however, part 2's code is more efficient due to the avoidance of the back-to-back branches.

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.

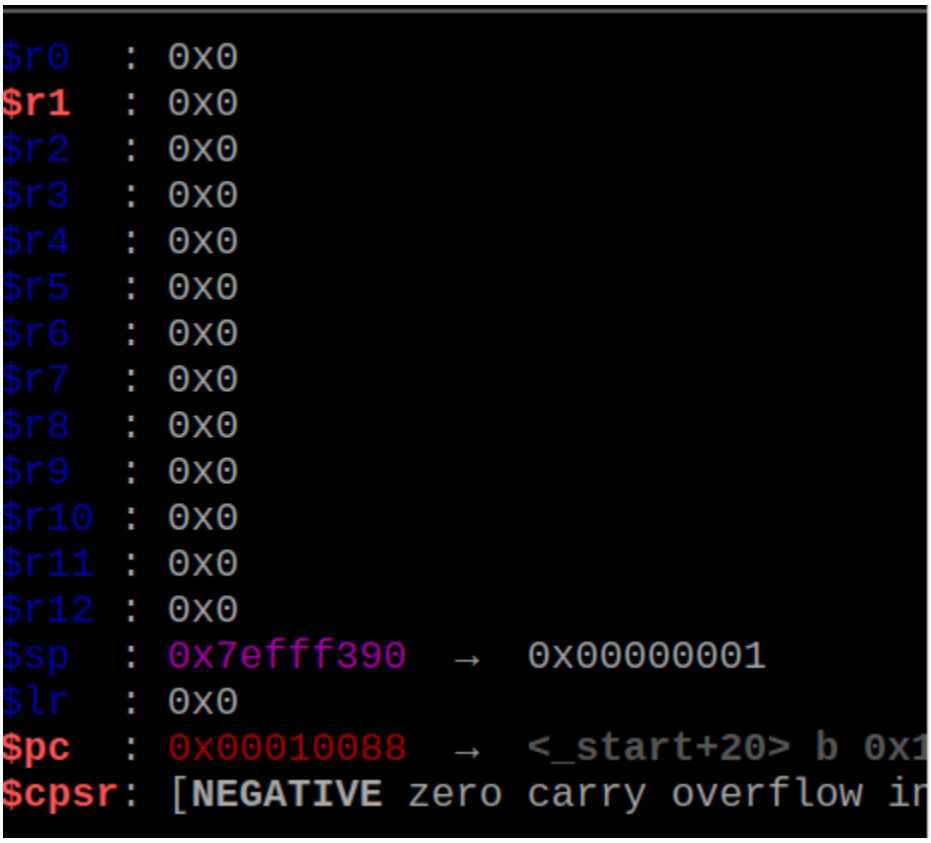


To determine the output of the program, we first opened a ControlStructure1.s file to write our program. We used the directive, **.data**, to indicate that we were going to declare some variables. We created a word memory size(32-bits) at variable location X which was initialized with 1.
To use variable **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**.