NAME: Kevin Gamage

For this assignment, I was tasked with creating several programs in assembly. The purpose of these assignments was to help me familiarize myself with the control flow of an assembly program as well as the X86 general registers and basic operations such as push, pop, add, div, and mul. I first learned how to set up the structure of a very simple assembly program that runs in Linux that had a text section and global _start. I was then able to learn about how the push and pop operations are used and how they alter the value stored in the ESP register and modify the stack of the program. I then learned how to use uninitialized variables in the .bss section of my programs to store values as well without having to explicitly place them on the stack and manipulate it myself. I was able to use these variables to store important values when my general registers were being used for other purposes. I then began to learn how to properly use loops in assembly code to automate the process of pushing and popping arbitrarily large numbers of values to and from the stack. I was able to implement a fairly smooth flow through my program despite a mildly intricate set of conditions that needed to be met for each program. I was also able to understand the process of how strings and integers are printed to the screen as I was able to write a program that printed a message to the console. Overall this project was enjoyable and helped me to understand a lot about how assembly control flow works and has certainly helped me to gain experience with following the logic in an assembly program. My only critique of this assignment would be that I struggled to learn a method to debug my programs as I went through the process of creating them. Thankfully one of my classmates was able to assist me and show me a fairly simple method of debugging that I wish we had seen an example of in class. I think it would have saved me a lot of time to have had the option to learn about some debugging tools and a short example on how to use them when creating programs.

Problem 1 Assembly code: a1p1.asm

I ran my .asm files by entering the following lines into the console with the specific name of the .asm file:

```
nasm -f elf32 prog.asm -o prog.o -g

ld -m elf_i386 prog.o -o prog

./prog
```

I was also able to debug the program by entering:

```
objdump –disassemble a1p1
```

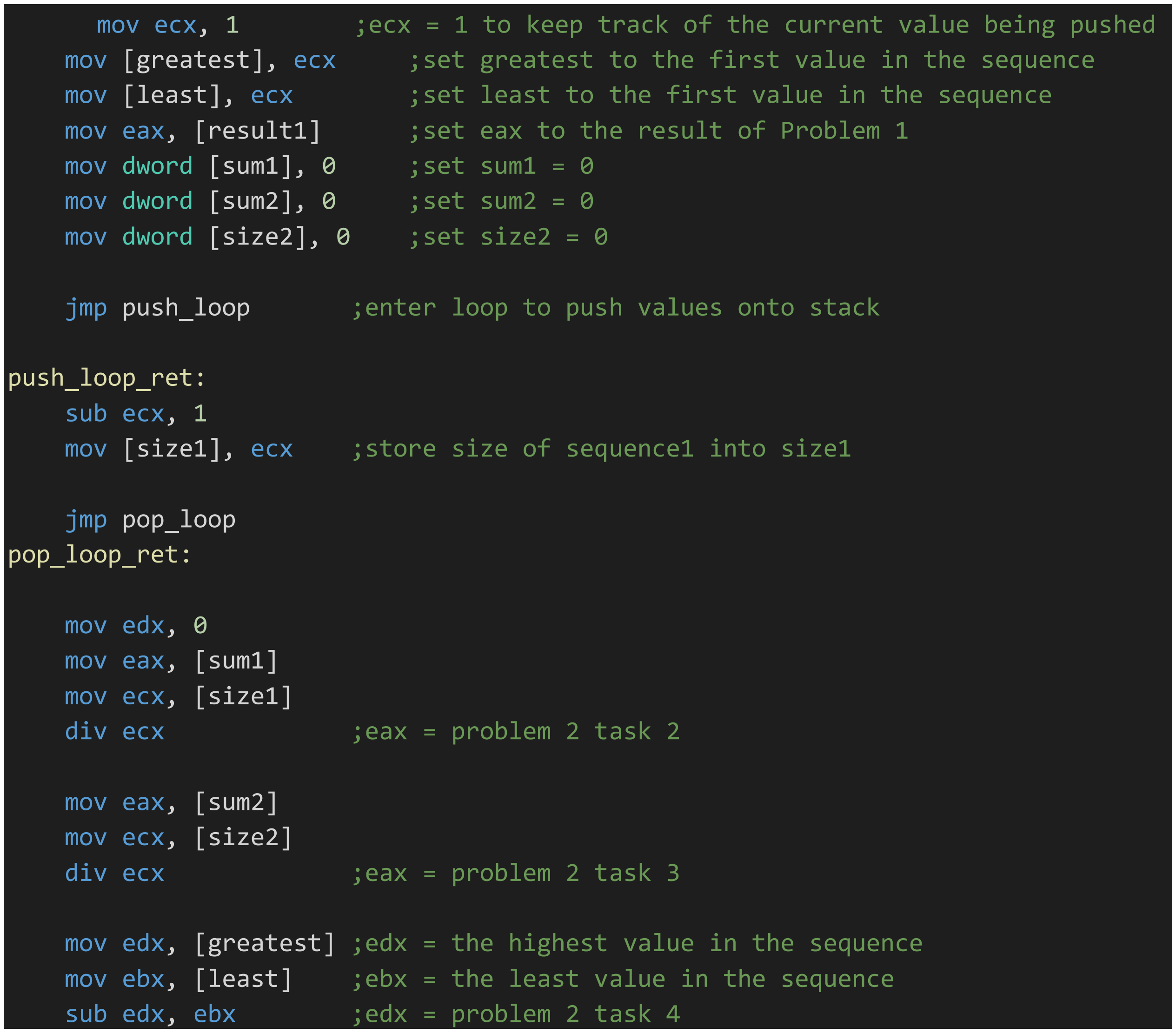
and running the program in gdb with breakpoints at the line values noted in the objdump output.

```
global _start

section .text
_start:

;Problem 1

    ;store values on the stack
    push 4
    push 77
    push 18
    push 57
    push 9

    ;add all the values and store in eax
    pop eax                 ;eax = 9
    pop ebx                 ;ebx = 57
    add eax, ebx            ;eax += 57
    pop ebx                 ;ebx = 18
    add eax, ebx            ;eax += 18
    pop ebx                 ;ebx = 77
    add eax, ebx            ;eax += 77
    pop ebx                 ;ebx = 4
    add eax, ebx            ;eax += 4

    ;divide the sum of the values by the number of values
    mov ecx, 5              ;ecx = 5
    div ecx                 ;eax = problem 1 task 2
```

Problem 2 Assembly code: a1p2.asm

This program was significantly harder to write than the previous one and required me to implement at first. I first began by instantiating variables for the greatest and least values in the sequence. I then instantiated the variables for the number of elements in the two sequences as well as the sums of the two sequences. I then created a loop in the program that pushed all the appropriate values onto the stack. The program then loops through the elements in the sequence and pops them and stores the sum and size in the appropriate variables. Finally, I divided the sums of the two sequences with their respective size and obtained the correct result.

This snippet shows the sequence of pushing and then popping values off the stack and computing the final result. I include the full .asm file in my submission.



```
      mov ecx, 1        ;ecx = 1 to keep track of the current value being pushed
    mov [greatest], ecx     ;set greatest to the first value in the sequence
    mov [least], ecx        ;set least to the first value in the sequence
    mov eax, [result1]      ;set eax to the result of Problem 1
    mov dword [sum1], 0     ;set sum1 = 0
    mov dword [sum2], 0     ;set sum2 = 0
    mov dword [size2], 0    ;set size2 = 0

    jmp push_loop       ;enter loop to push values onto stack

push_loop_ret:
    sub ecx, 1
    mov [size1], ecx    ;store size of sequence1 into size1

    jmp pop_loop
pop_loop_ret:

    mov edx, 0
    mov eax, [sum1]
    mov ecx, [size1]
    div ecx             ;eax = problem 2 task 2

    mov eax, [sum2]
    mov ecx, [size2]
    div ecx             ;eax = problem 2 task 3

    mov edx, [greatest] ;edx = the highest value in the sequence
    mov ebx, [least]    ;ebx = the least value in the sequence
    sub edx, ebx        ;edx = problem 2 task 4
```

Problem 3 Assembly code: a1p3.asm

After the second problem in this assignment I was well prepared to handle this one. I instantiated a single variable to hold the sum of the sequence. Similar to problem 2, I then pushed all the appropriate values in the sequence onto the stack in a loop. I then looped through the values on the stack popping them and adding or multiplying the values as necessary and storing the results in the appropriate registers.

The below snippet shows the general outline I describe above. I include the full .asm file in my submission.

```
    mov ecx, 0          ;ecx = 0 to keep track of the current value being pushed
    mov edx, 0     ;edx = 0 to keep track of the number of iterations in the loop
    mov dword [sum], 0  ;[sum] = 0 to keep track of the total sum of the sequence

push_seq_ret:

    cmp ecx, 100        ;if ecx <= 100
    jle push_seq        ;jump to push_seq

    mov eax, edx        ;eax = Problem 3 task 2
    mov ecx, eax        ;edx register resets from overloaded mul operation so I u
sed ecx
    mov eax, 1          ;reset eax value to use for next task

    jmp pop_seq

pop_seq_ret:

 ;eax holds the product of the multiplications at this point for Problem 3 task 3
    mov ebx, [sum]      ;store the sum in ebx

    ;Terminate the program
    mov eax, 1
    mov ebx, 0
    int 0x80
```

Problem 4 Assembly code: a1p4.asm

For this program I simply created an assembly program that is able to print the string “CSCE 451 is fun!” to the screen. I was able to see how a program like this looks from an example shown in our class. For this program to run, the EAX register must be set to 4 in order to for the system call to call the write function. The EBX register must be set to 1 in order to direct the write to stdout. The ECX register must store the pointer to the string which is going to be displayed. Interestingly, it is more complicated to print numbers to the screen as each digit in the number must be converted to its appropriate ASCII value before being printed to the screen. The EDX register will contain the number of bytes that the system will write to the screen. The system kernel is then called on the register values using the int 0x80 line. As in all my programs, I then exit the program by setting the EAX register to 1 and EBX register to 0 to signal the system to terminate the program with the exit status 0.

I set the registers to the appropriate values to print stdout with the message and then exited the program.

```
global _start

section .data
    msg db "CSCE 451 is fun!", 0x0a

section .text
_start:
    mov eax, 4      ;eax value for write()
    mov ebx, 1      ;file descriptor for stdout
    mov ecx, msg    ;message to send to stdout
    mov edx, 17     ;length of message
    int 0x80        ;system call

    mov eax, 1      ;exit program
    mov ebx, 0
    int 0x80        ;system call
```