Arlan Prado
Professor Chenhansa
CS-118-02

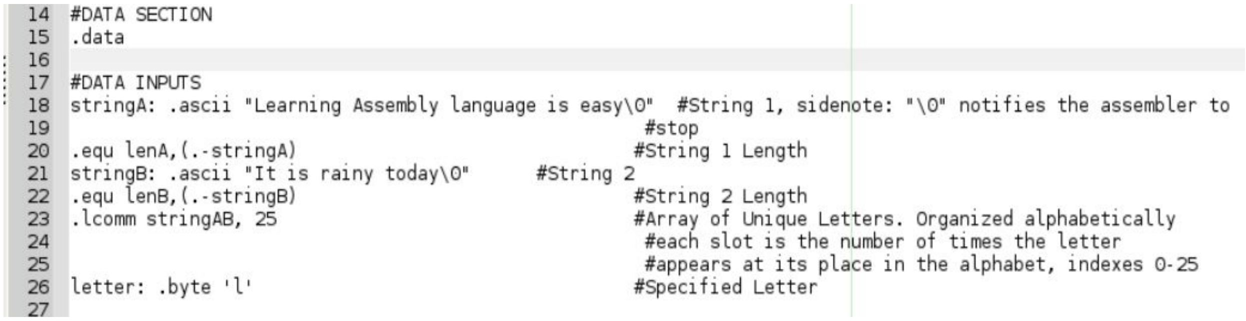
Lab 4

Purpose:

The purpose of this assignment is to enforce the use of Bitwise Boolean, using labels, using unconditional/conditional jumps, using arrays, and using many registers. It also helps student plan out their code more and trying to make it the least complex it could be while also not making it too long.

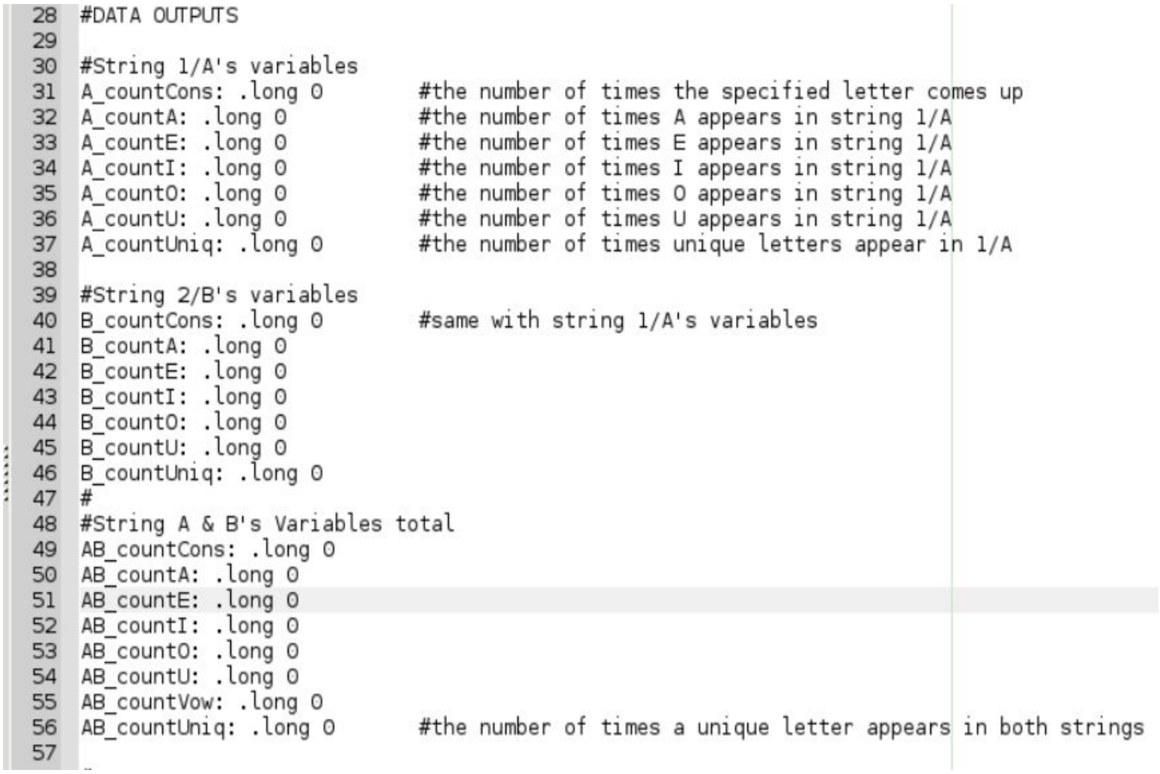
Process:

The first step is to create the variables. The Input variables are the two strings(A&B), the length of the strings, the uninitialized array to hold the count of unique letters between A&B, and the specified letter to be identified from each array. The uninitialized array represents the alphabet and each index would be the number of times the letter appears in both stringA and stringB.



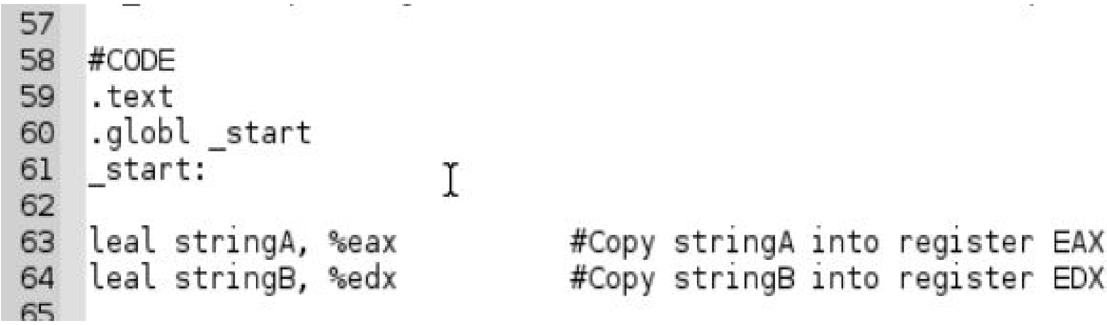
```
#DATA SECTION
.data

#DATA INPUTS
stringA: .ascii "Learning Assembly language is easy\0"  #String 1, sidenote: "\0" notifies the assembler to
                                                  #stop
.equ lenA,(.-stringA)                            #String 1 Length
stringB: .ascii "It is rainy today\0"     #String 2
.equ lenB,(.-stringB)                            #String 2 Length
.lcomm stringAB, 25                              #Array of Unique Letters. Organized alphabetically
                                                  #each slot is the number of times the letter
                                                  #appears at its place in the alphabet, indexes 0-25
letter: .byte 'l'                                #Specified Letter
```

The second variables are for stringA's counters, stringB's counters, and the counters for both stringA and stringB. Both StringA and stringB need a counter for the identified consonant (_countCons), a counter for all the vowels (_countA), and a counter for all unique letters (_countUniq). AB_countUniq gets the number of unique letters from both stringA and stringB, meaning that A_countUniq and B_countUniq can't be added to get the value of unique letters between both strings.



```
#DATA OUTPUTS

#String 1/A's variables
A_countCons: .long 0        #the number of times the specified letter comes up
A_countA: .long 0           #the number of times A appears in string 1/A
A_countE: .long 0           #the number of times E appears in string 1/A
A_countI: .long 0           #the number of times I appears in string 1/A
A_countO: .long 0           #the number of times O appears in string 1/A
A_countU: .long 0           #the number of times U appears in string 1/A
A_countUniq: .long 0        #the number of times unique letters appear in 1/A

#String 2/B's variables
B_countCons: .long 0        #same with string 1/A's variables
B_countA: .long 0
B_countE: .long 0
B_countI: .long 0
B_countO: .long 0
B_countU: .long 0
B_countUniq: .long 0
#
#String A & B's Variables total
AB_countCons: .long 0
AB_countA: .long 0
AB_countE: .long 0
AB_countI: .long 0
AB_countO: .long 0
AB_countU: .long 0
AB_countVow: .long 0
AB_countUniq: .long 0       #the number of times a unique letter appears in both strings

```

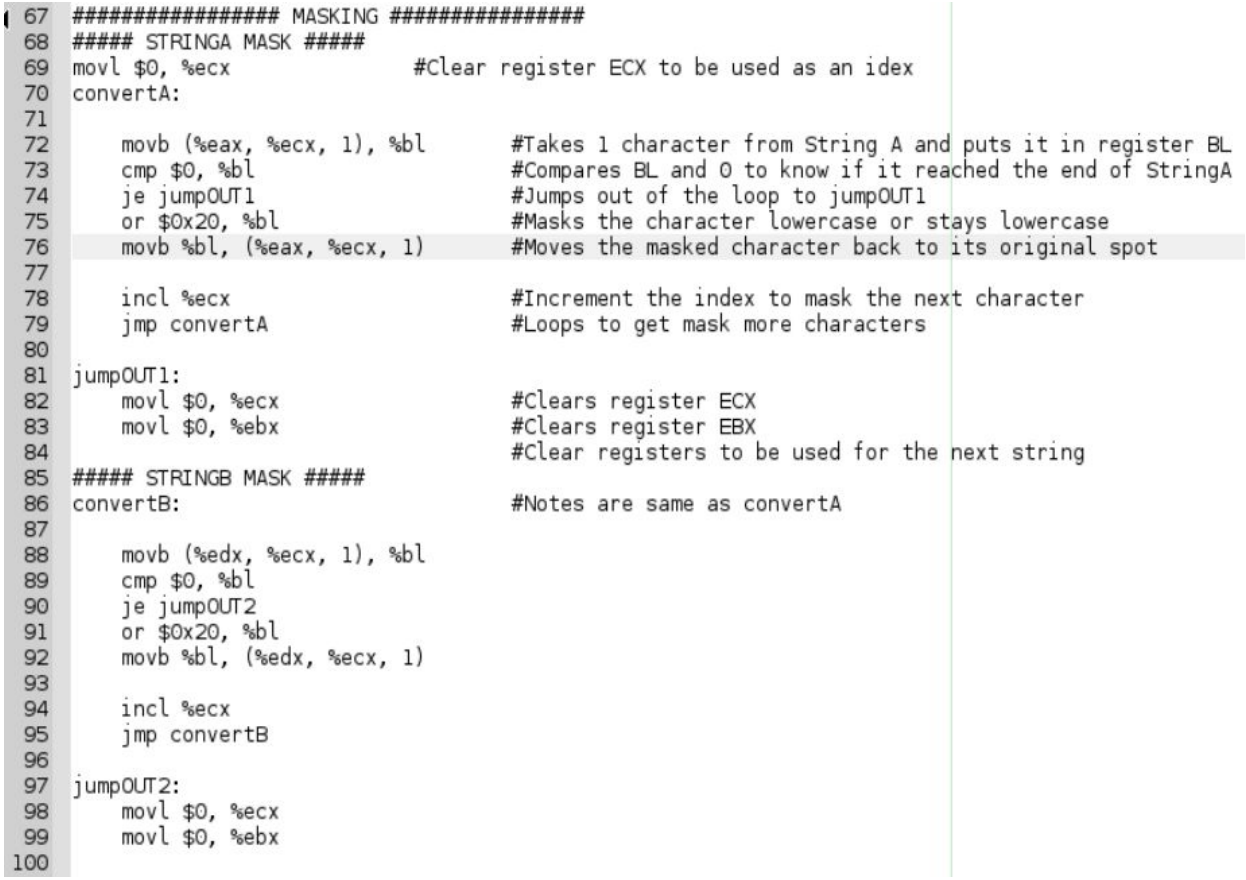
The first lines of code are copying the strings into registers EAX and EDX.



```
#CODE
.text
.globl _start
_start:

leal stringA, %eax          #Copy stringA into register EAX
leal stringB, %edx          #Copy stringB into register EDX
```

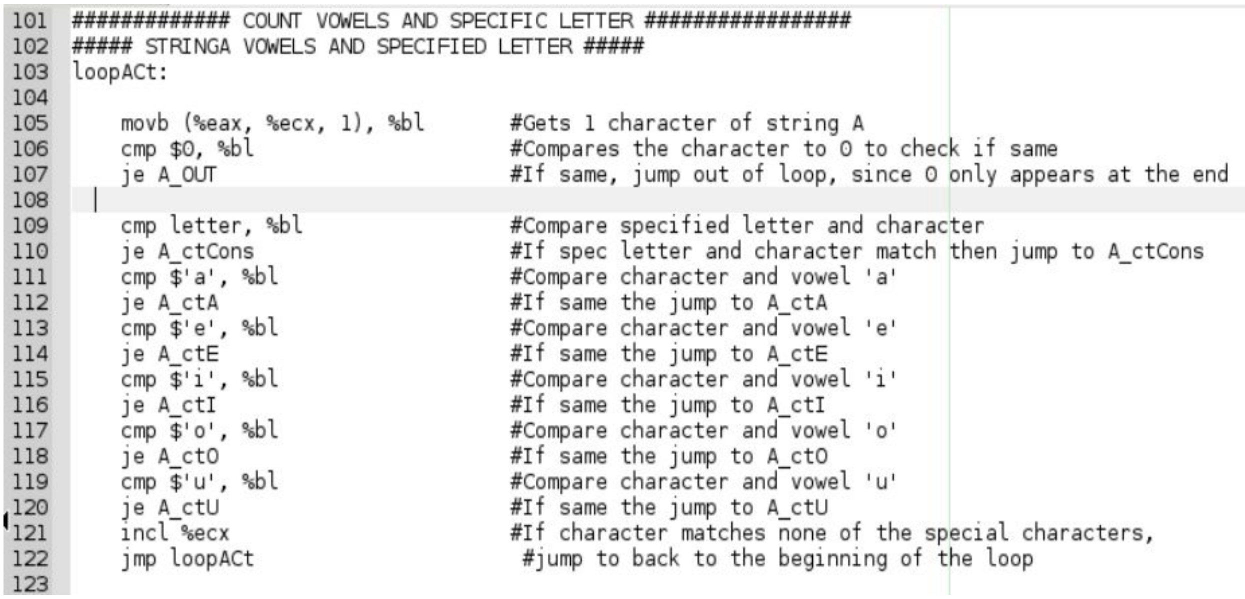
Before counting any characters in strings, all the letters need to be in lowercase so I would not need to take into consideration the uppercase letters and their ascii code. The easiest way to change the characters of the strings to lowercase is by masking all the characters of the string by 32 (in this case 20 in hex) since in ascii, the difference between uppercase and lowercase is 32.

So the process would be to get the character of the string into an 8 bit register (in this case it is BL) by using movb. Then compare 0 to BL since if BL was 0 that would mean the whole string has been parsed and the loop would end. If BL is not 0, mask the character with $0x20 and replace the original character with the new masked character. After that, increase the register holding the index to move onto the next letter and go back to the loop. After the whole string has been successfully masked, jump out of the loop and reset the registers holding the counter and the character to 0.



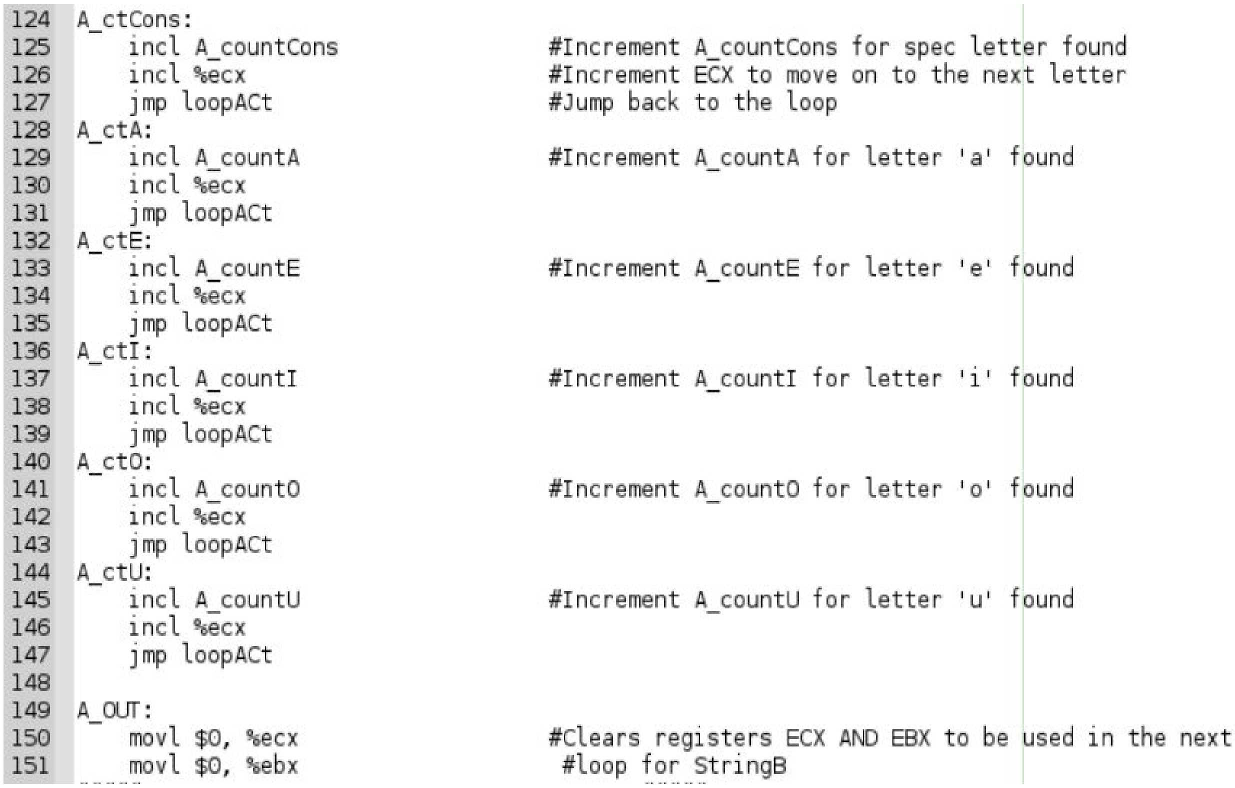
```
################# MASKING ################
##### STRINGA MASK #####
movl $0, %ecx               #Clear register ECX to be used as an idex
convertA:

    movb (%eax, %ecx, 1), %bl       #Takes 1 character from String A and puts it in register BL
    cmp $0, %bl                     #Compares BL and 0 to know if it reached the end of StringA
    je jumpOUT1                     #Jumps out of the loop to jumpOUT1
    or $0x20, %bl                   #Masks the character lowercase or stays lowercase
    movb %bl, (%eax, %ecx, 1)       #Moves the masked character back to its original spot

    incl %ecx                       #Increment the index to mask the next character
    jmp convertA                    #Loops to get mask more characters

jumpOUT1:
    movl $0, %ecx                   #Clears register ECX
    movl $0, %ebx                   #Clears register EBX
                                    #Clear registers to be used for the next string
##### STRINGB MASK #####
convertB:                           #Notes are same as convertA

    movb (%edx, %ecx, 1), %bl
    cmp $0, %bl
    je jumpOUT2
    or $0x20, %bl
    movb %bl, (%edx, %ecx, 1)

    incl %ecx
    jmp convertB

jumpOUT2:
    movl $0, %ecx
    movl $0, %ebx
```

After making all the characters in both strings lowercase, then the vowels and specified consonant have to be counted. To do this, the same method of getting each character will be used from the masking procedure (using movb (array,index,size), 8bitregister). The 8 bit register would then be compared to 0 to see if the whole string has been parsed through. If the register IS 0, jump out of the loop since it is done. But for when it parses through the string, the character is compared to the specified character and vowels. If the character is matched then jump into its label where it would increment the its respective count variable. If the character is not matched, then the index is increased and it is jumped back to the loop.



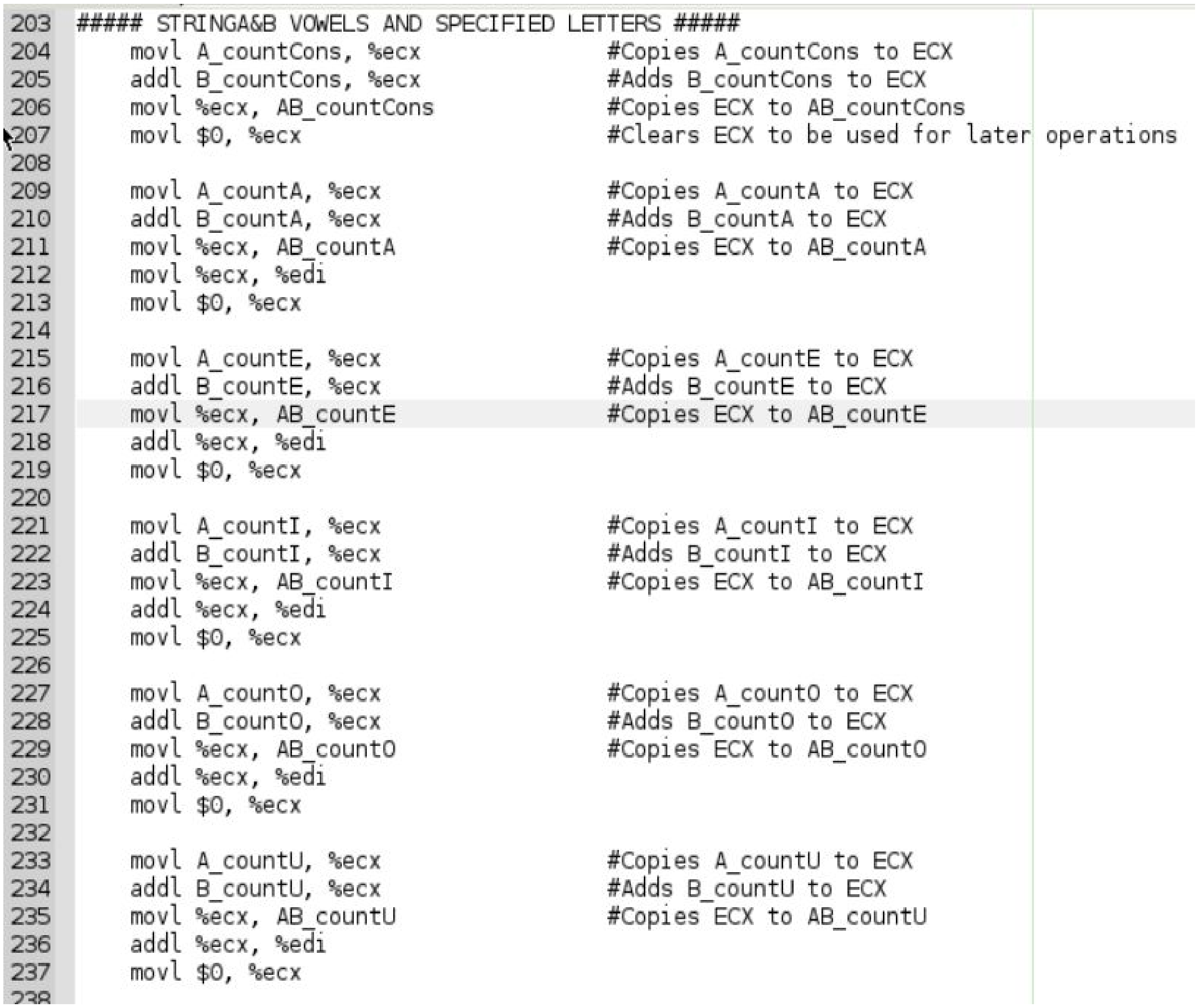
```
############# COUNT VOWELS AND SPECIFIC LETTER #################
##### STRINGA VOWELS AND SPECIFIED LETTER #####
loopACt:

    movb (%eax, %ecx, 1), %bl       #Gets 1 character of string A
    cmp $0, %bl                     #Compares the character to 0 to check if same
    je A_OUT                        #If same, jump out of loop, since 0 only appears at the end

    cmp letter, %bl                 #Compare specified letter and character
    je A_ctCons                     #If spec letter and character match then jump to A_ctCons
    cmp $'a', %bl                   #Compare character and vowel 'a'
    je A_ctA                        #If same the jump to A_ctA
    cmp $'e', %bl                   #Compare character and vowel 'e'
    je A_ctE                        #If same the jump to A_ctE
    cmp $'i', %bl                   #Compare character and vowel 'i'
    je A_ctI                        #If same the jump to A_ctI
    cmp $'o', %bl                   #Compare character and vowel 'o'
    je A_ctO                        #If same the jump to A_ctO
    cmp $'u', %bl                   #Compare character and vowel 'u'
    je A_ctU                        #If same the jump to A_ctU
    incl %ecx                       #If character matches none of the special characters,
    jmp loopACt                      #jump to back to the beginning of the loop

```

Here is the labels for the previous picture. Each label (except the last) would increase the vowel count or the consonant, the index, and then jump back to the loop. The A_OUT label, clears the registries and exits the loop.



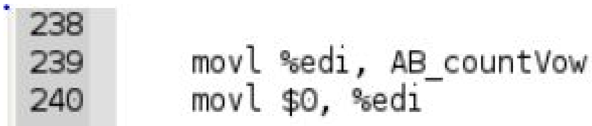
```
A_ctCons:
    incl A_countCons                #Increment A_countCons for spec letter found
    incl %ecx                       #Increment ECX to move on to the next letter
    jmp loopACt                     #Jump back to the loop
A_ctA:
    incl A_countA                   #Increment A_countA for letter 'a' found
    incl %ecx
    jmp loopACt
A_ctE:
    incl A_countE                   #Increment A_countE for letter 'e' found
    incl %ecx
    jmp loopACt
A_ctI:
    incl A_countI                   #Increment A_countI for letter 'i' found
    incl %ecx
    jmp loopACt
A_ctO:
    incl A_countO                   #Increment A_countO for letter 'o' found
    incl %ecx
    jmp loopACt
A_ctU:
    incl A_countU                   #Increment A_countU for letter 'u' found
    incl %ecx
    jmp loopACt

A_OUT:
    movl $0, %ecx                   #Clears registers ECX AND EBX to be used in the next
    movl $0, %ebx                    #loop for StringB
```

After getting the counts for each stringA and stringB, add each count from the strings into the AB_count variables. The EDI register counts all the variables while the ECX register is only used for immediate addition for the AB_count(Vowel) variables.



```
##### STRINGA&B VOWELS AND SPECIFIED LETTERS #####
    movl A_countCons, %ecx              #Copies A_countCons to ECX
    addl B_countCons, %ecx              #Adds B_countCons to ECX
    movl %ecx, AB_countCons             #Copies ECX to AB_countCons
    movl $0, %ecx                       #Clears ECX to be used for later operations

    movl A_countA, %ecx                 #Copies A_countA to ECX
    addl B_countA, %ecx                 #Adds B_countA to ECX
    movl %ecx, AB_countA                #Copies ECX to AB_countA
    movl %ecx, %edi
    movl $0, %ecx

    movl A_countE, %ecx                 #Copies A_countE to ECX
    addl B_countE, %ecx                 #Adds B_countE to ECX
    movl %ecx, AB_countE                #Copies ECX to AB_countE
    addl %ecx, %edi
    movl $0, %ecx

    movl A_countI, %ecx                 #Copies A_countI to ECX
    addl B_countI, %ecx                 #Adds B_countI to ECX
    movl %ecx, AB_countI                #Copies ECX to AB_countI
    addl %ecx, %edi
    movl $0, %ecx

    movl A_countO, %ecx                 #Copies A_countO to ECX
    addl B_countO, %ecx                 #Adds B_countO to ECX
    movl %ecx, AB_countO                #Copies ECX to AB_countO
    addl %ecx, %edi
    movl $0, %ecx

    movl A_countU, %ecx                 #Copies A_countU to ECX
    addl B_countU, %ecx                 #Adds B_countU to ECX
    movl %ecx, AB_countU                #Copies ECX to AB_countU
    addl %ecx, %edi
    movl $0, %ecx
```

(AB_countVow, takes the total number of vowels between each string but is not needed for the assignment)



```

    movl %edi, AB_countVow
    movl $0, %edi
```

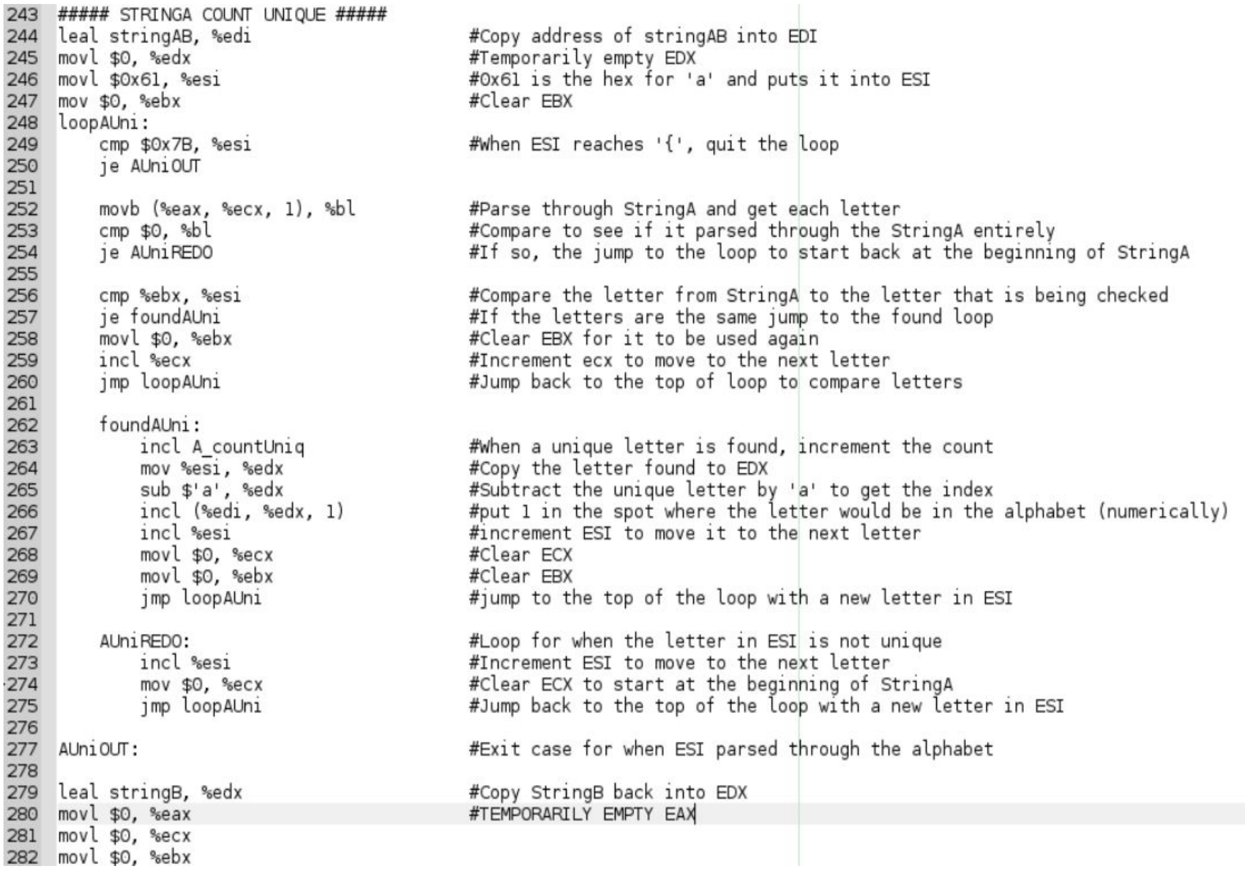
To count the unique characters of the string (for stringA), I first had to copy stringAB (the unique count for both strings) into register EDI, copy the ascii value of ‘a’ into ESI, and empty registers I will be using for the unique count loop.

The first thing the loop does is compare ESI to the ascii value of ‘{‘ because it is after ‘z’. If ESI is ‘{‘, then jump out of the loop. If ESI is not ‘{‘ then get the character from the string (put into BL) and compare it to 0.

If BL is 0 then it means the letter of ESI is not in the string so jump to the “UniREDO” label and ESI is incremented to go to the next letter and the index of the string (ECX) is reset to 0. If BL is not 0, compare EBX (since it contains BL) and ESI.

If they are the same letter, jump to the “foundUni” label and increment the unique letter count. Copy ESI into EDX (which is empty, to be used), then subtract ‘a’ from EDX to get the index for the uninitialized array in EDI. When the index is given, increment at the index of EDI to show that the unique letter is counted. After that, increment ESI and clear ECX and EBX. Then jump back to the loop.

After jumping out of the loop because ESI reached to ‘z’, copy stringB back into EDX, and empty EAX, EBX, and ECX (For stringB’s portion).



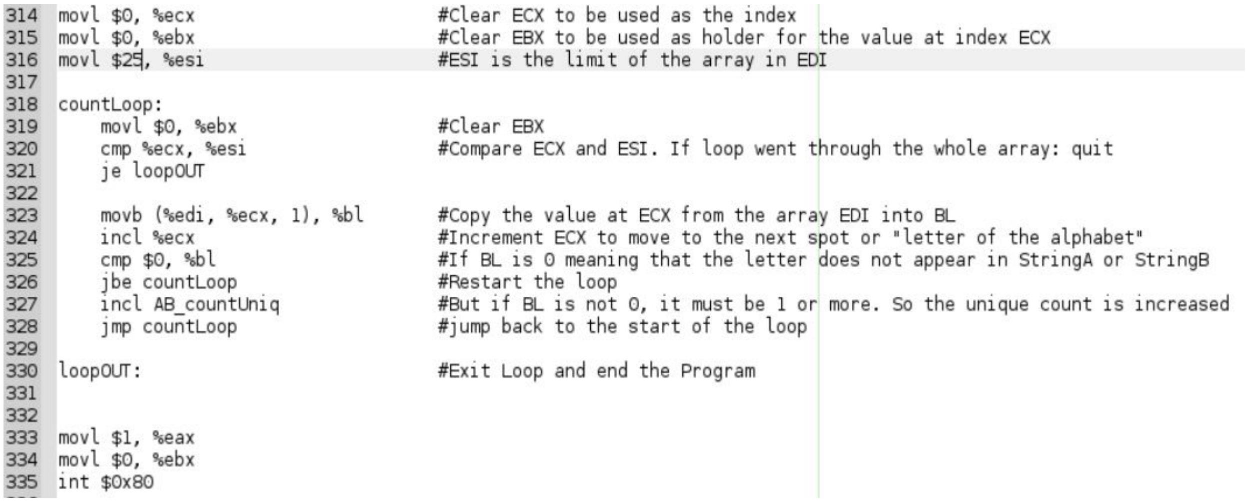
```
##### STRINGA COUNT UNIQUE #####
leal stringAB, %edi                     #Copy address of stringAB into EDI
movl $0, %edx                           #Temporarily empty EDX
movl $0x61, %esi                        #0x61 is the hex for 'a' and puts it into ESI
mov $0, %ebx                            #Clear EBX
loopAUni:
    cmp $0x7B, %esi                     #When ESI reaches '{', quit the loop
    je AUniOUT

    movb (%eax, %ecx, 1), %bl           #Parse through StringA and get each letter
    cmp $0, %bl                         #Compare to see if it parsed through the StringA entirely
    je AUniREDO                         #If so, the jump to the loop to start back at the beginning of StringA

    cmp %ebx, %esi                      #Compare the letter from StringA to the letter that is being checked
    je foundAUni                        #If the letters are the same jump to the found loop
    movl $0, %ebx                       #Clear EBX for it to be used again
    incl %ecx                           #Increment ecx to move to the next letter
    jmp loopAUni                        #Jump back to the top of loop to compare letters

    foundAUni:
        incl A_countUniq                #When a unique letter is found, increment the count
        mov %esi, %edx                  #Copy the letter found to EDX
        sub $'a', %edx                  #Subtract the unique letter by 'a' to get the index
        incl (%edi, %edx, 1)            #put 1 in the spot where the letter would be in the alphabet (numerically)
        incl %esi                       #increment ESI to move it to the next letter
        movl $0, %ecx                   #Clear ECX
        movl $0, %ebx                   #Clear EBX
        jmp loopAUni                    #jump to the top of the loop with a new letter in ESI

    AUniREDO:                           #Loop for when the letter in ESI is not unique
        incl %esi                       #Increment ESI to move to the next letter
        mov $0, %ecx                    #Clear ECX to start at the beginning of StringA
        jmp loopAUni                    #Jump back to the top of the loop with a new letter in ESI

AUniOUT:                                #Exit case for when ESI parsed through the alphabet

leal stringB, %edx                      #Copy StringB back into EDX
movl $0, %eax                           #TEMPORARILY EMPTY EAX
movl $0, %ecx
movl $0, %ebx
```

So after counting the unique characters of stringA and stringB, the unique characters between them both have to be counted.

The process for this begins with clearing EBX and ECX. Then enter the limit of the stringAB array (25) to ESI.

The loop would compare ECX (the counter/index) and the limit ESI; if they are the same, it would jump out of the loop because it is done. If not, then the value at index ECX of array stringAB/EDI would be put into BL. ECX would increment for it to move to the next value and BL would be compared to 0. If BL is equal to 0, then jump back to the top of the loop since the letter at index ECX is not counted for as a unique letter. If BL is above 0, it means that the unique letter is counted for for at least 1 string. Then it will ignore the jump and increment AB_countUniq and jump back to the top of the loop.

Once ESI and ECX are the same, it will quit the loop and the program.



```
movl $0, %ecx                   #Clear ECX to be used as the index
movl $0, %ebx                   #Clear EBX to be used as holder for the value at index ECX
movl $25, %esi                  #ESI is the limit of the array in EDI

countLoop:
    movl $0, %ebx               #Clear EBX
    cmp %ecx, %esi              #Compare ECX and ESI. If loop went through the whole array: quit
    je loopOUT

    movb (%edi, %ecx, 1), %bl   #Copy the value at ECX from the array EDI into BL
    incl %ecx                   #Increment ECX to move to the next spot or "letter of the alphabet"
    cmp $0, %bl                 #If BL is 0 meaning that the letter does not appear in StringA or StringB
    jbe countLoop               #Restart the loop
    incl AB_countUniq           #But if BL is not 0, it must be 1 or more. So the unique count is increased
    jmp countLoop               #jump back to the start of the loop

loopOUT:                        #Exit Loop and end the Program


movl $1, %eax
movl $0, %ebx
int $0x80
```

Pitfalls:

The biggest pitfall was starting off to write the code. I knew I had to mask the string with 32 to make lowercase, but I did not understand how to do that. I initially thought I could just do “or $0x20, %eax” and I could not understand why my string was being ruined. I tried to understand why it was happening by searching through google and looking through my book, but it was only until I talked to my friends that I started to understand how to do it and make it my own.

Another problem I had was trying to access one character from a string, and with my friends help, I was able to understand that movb (%eax, %ecx, 1), %bl would mean to get the ECX index at EAX array with the bit size of 1 and copy it in BL. I really needed to understand how to do that because no part of the program would run if I did not know how to get the character

The last major problem was when I needed to come up with a solution to the unique count between both strings. I came up with the idea of getting the characters of each solution and putting it into an expanding string, but I asked the tutor Jaures to help and he said to make an uninitialized array. The uninitialized array would hold the numbers of what unique character shows up and each position in the array would be corresponding to their alphabetical counterpart. It was a great idea, and I decided to implement it into the code.

Possible Improvements:

A possible improvement could be to have the vowels and specific consonant counters in the unique counter as well. It would make my code a lot shorter and more efficient but it would also make my code look clustered and it might cause me to be very confused when I would have to try to debug my program.