CMPT 295

Name: Junchen Li

Student ID: 301385486

Date: 2020/3/8

Assignment 5

Objectives:

- x86-64 function calls and stack
- Recursion in x86-64 assembly code
- Manipulating 2D arrays (matrices) in x86-64 assembly code

Submission:

- Submit your document called **Assignment_5.pdf**, which must include your answers to all of the questions in Assignment 5.
  - Add your full name and student number at the top of the first page of your document **Assignment_5.pdf**.
- When creating your assignment, first include the question itself and its number then include your answer, keeping the questions in its original numerical order.
- **If you hand-write your answers (as opposed to using a computer application to write them):** When putting your assignment together, do not take photos (no .jpg) of your assignment sheets! Scan them instead! Better quality -> easier to read -> easier to mark!
- Submit your assignment **Assignment_5.pdf** electronically on CourSys.
- You will need to submit your code on CourSys as well. Refer to the Submission section of Q2 and Q3 for further submission instructions.

Due:

- Thursday, March 5 at 3pm
- Late assignments will receive a grade of 0, but they will be marked in order to provide the student with feedback.

Marking scheme:

This assignment will be marked as follows:

- All questions will be marked for correctness.

The amount of marks for each question is indicated as part of the question.

A solution will be posted after the due date.

1. [5 marks] x86-64 function calls and stack

   a. Hand trace the code from our Lab 4 (`main.c`, `main.s`, `p1.c`, `p1.s`, `p2.c` and `p2.s`) using the current test case, i.e., `x = 6`, `y = 9`, `buf[40]`. As you do so, draw the corresponding Stack Diagram and Register Table for the entire program, i.e., until you reach (but have not yet executed) the `ret` instruction of the main function. Use the "Stack Diagram" sheet at the end of this assignment. Indicate the movement of %rsp by crossing its old location and rewriting "%rsp" to indicate its new location (as we have done in our lectures). Cross the content of the stack that have been popped. When the value of a stack location is changed, cross its old value and write the new value in the same stack location. Make sure you include the content of `buf` in your Stack Diagram.

   b. Modify `main.c` by reducing the size of `buf[]` from 40 to 24. Remake the code and hand trace it using the following test case, i.e., `x = 6`, `y = 9`, `buf[24]`. Do exactly has instructed in the section a. above. Make sure you include the content of `buf` in your Stack Diagram.

      What happens to the "canary value" in this situation?

Answer: Stack smashing is a protection mechanism for GCC to detect "cache overflow". When the allocated memory is insufficient, the execution continues. But the error does not appear until the program returns at the end of the program.

x= 6 , y= 9 , buf[40]

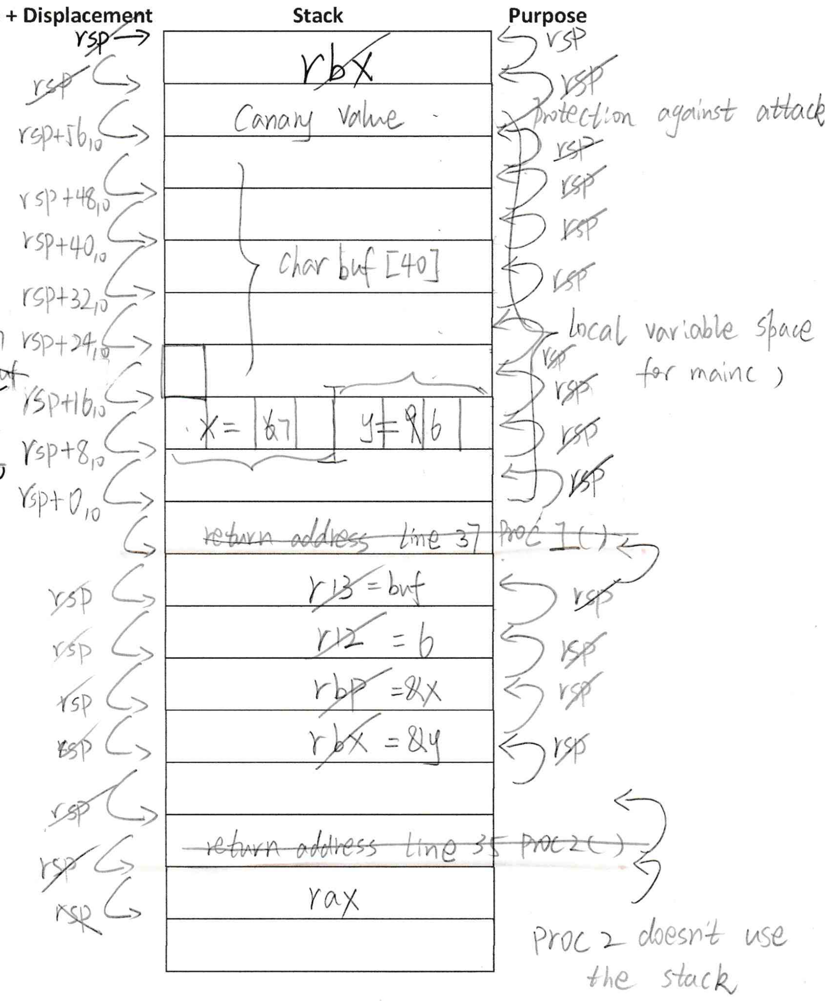
CMPT 295 – Spring 2020

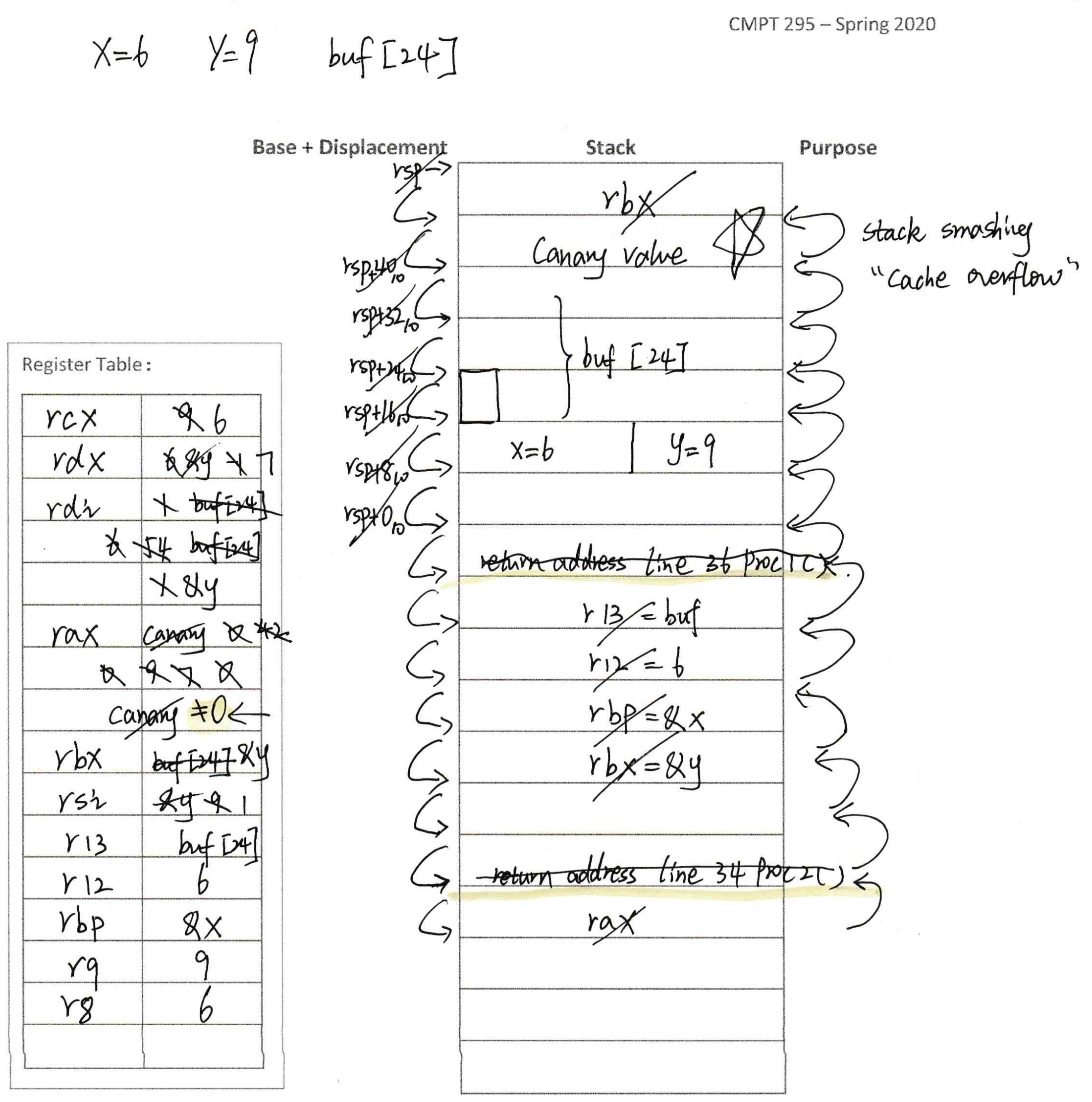
Register Table :

x&y

0

rcx	~~9~~ 6
rdx	~~6~~ ~~&y~~ 17
rdi	~~x~~ ~~buf~~ ~~x~~ ~~17~~ buf
rax	~~canary~~ ~~0~~ ~~42~~
	~~0~~ ~~9~~ ~~7~~ ~~0~~ ~~canary~~
rbx	~~buf~~ &y
rsi	~~&x~~ ~~9~~ 1
r13	buf
r12	~~buf~~ 6
rbp	&x
r9	9
r8	6





X=6 Y=9 buf[24]

Base + Displacement	Stack	Purpose
~~rsp~~ →	~~rbx~~	
	Canary value	stack smashing
~~rsp+40~~$_{10}$		"Cache overflow"
~~rsp+32~~$_{10}$	buf[24]	
~~rsp+24~~$_{10}$		
~~rsp+16~~$_{10}$	x=6 | y=9	
~~rsp+8~~$_{10}$		
~~rsp+0~~$_{10}$	~~return address line 36 Proc1()~~	
	~~r13~~ = buf	
	~~r12~~ = 6	
	~~rbp~~ = &x	
	~~rbx~~ = &y	
	~~return address line 34 Proc2()~~	
	~~rax~~	

Register Table :

rcx	~~9~~ 6
rdx	~~x~~ ~~&y~~ ~~x~~ 7
rdi	~~x~~ ~~buf[24]~~
	~~x~~ ~~54~~ ~~buf[24]~~
	~~x~~ &y
rax	~~canary~~ ~~x~~ ~~*2~~
	~~x~~ ~~9~~ ~~x~~ ~~x~~
	~~canary~~ ≠0 ←
rbx	~~buf[24]~~ &y
rsi	~~&y~~ ~~9~~ 1
r13	buf[24]
r12	6
rbp	&x
r9	9
r8	6

2. [5 marks] Recursion in x86-64 assembly code

In this problem, you are asked to rewrite the `mul` function you wrote in Assisgnment 4. This time, instead of using a loop, you are to use recursion. You must use the stack, either by pushing/popping or by getting a section of it (e.g., `subq $24, %rsp`) and releasing it (e.g., `addq $24, %rsp`) at the end of your program.

Use your files from Assignment 4: `main.c`, `makefile` and `calculator.s`. Then, copy the following and paste it over (replace) your entire `mul` function in `calculator.s`:

```
mul: # performs integer multiplication - when both operands
are non-negative!
# x in edi, y in esi
# Requirements:
# - cannot use imul* instruction
# - you must use recursion (no loop) and the stack
```

Then implement this recursive version of `mul`. While doing so, you must satisfy its new requirements. You must also satisfy the requirements below.

Requirements:

- Your code must be commented such that others (i.e., TA's) can read your code and understand what each instruction does.
  - About comments:
    - Here is an example of a useful comment:
      `cmpl   %edx, %r8d  # loop while j < N`
    - Here is an example of a **not** so useful comment:
      `cmpl   %edx, %r8d  # compare %edx with %r8d`

    Do you see the difference? Make sure you write comments of type 1.
- Make sure you update the header comment block in `calculator.s`.
- You must use the makefile provided in Assignment 4 when compiling your code. This makefile cannot be modified.
- You cannot modify the prototype of the function **`mul`**. The reason is that your code may be tested using a test driver built based on this function prototype.
- Your code **must** compile and successfully execute on the computers in CSIL (using the Linux platform).
- You must follow the x86-64 function call and register saving conventions described in class and in the textbook.

- Do not push/pop registers unless you make use of them in your function. Memory accesses are expensive!

Submission:

- Electronically submit your file `calculator.s` via CourSys.
- Also include a hardcopy of your `calculator.s` in this assignment. Make sure it is the version of `calculator.s` that compiles and executes on a CSIL Linux machine.

```
# Name: calculator.s

# Junchen Li

# 2020.3.6

# 301385486

# add a header comment block

        .globl  lt

        .globl  plus

        .globl  minus

        .globl  mul


# x in edi, y in esi


lt: # add a description to this function

        xorl    %eax, %eax

        cmpl    %esi, %edi

        setl    %al

        ret


plus:  # performs integer addition
```

```
# Requirement:
# - you cannot use add* instruction
   leal (%edi,%esi), %eax   # eax <- x + y
   ret

minus: # performs integer subtraction
# Requirement:
# - you cannot use sub* instruction
   movl %esi, %eax
   negl %eax
   leal (%edi,%eax), %eax   # eax <- x + (- y)
   ret

mul: # performs integer multiplication - when both operands are non-negative!
 # x in edi, y in esi
 # Requirements:
# - cannot use imul* instruction
# - you must use recursion (no loop) and the stack
mul:
   xorl    %eax, %eax      # we set the return value to zero
   movl    %esi, %eax      # we save the value y into the return value for counting purpose
   testl   %esi, %esi      # test if the x equal to zero  if (x==0)
   je      done            # if it equal to zero then jump to return
   pushq   %rbx            # push the callee saved value to the stack
   movl    %edi, %ebx      # let callee saved value save x value x+mul(x,y-1)
   pushl   %edi
   pushl   %esi
```

```
    subl    $1,   %esi      # change the value of y to y-1
    call    mul             # call the mul function
    addl    %ebx, %eax      # add the function returnd value with the x. x+mul(x,y-1)
    popq    %rbx            # pop the rsp value
    popl    %edi
    popl    %esi
done:
    ret
```

---

3. [10 marks] Manipulating 2D arrays (matrices) in x86-64 assembly code

In linear algebra, a matrix is a rectangular grid of numbers. Its dimensions are specified by its number of rows and of columns. This question focuses on the representation and manipulation of square matrices, i.e., where the number of rows and the number of columns both equal **n**.

Here is an example of a square matrix where n = 4:

$$A = \begin{bmatrix} 1 & -2 & 3 & -4 \\ -5 & 6 & -7 & 8 \\ -1 & 2 & -3 & 4 \\ 5 & -6 & 7 & -8 \end{bmatrix} \quad = \quad \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Note the notation A_{ij} refers to the matrix entry at the i^{th} row and the j^{th} column of A. Each row of the matrix A resembles a one dimensional array in the programming language C, with the value of j increasing for each element. The matrix A has i such rows.

Because of this resemblance, matrices can be represented (modeled) in our C programs using two dimensional arrays. One dimensional arrays are stored in contiguous memory space, where their element 0 is followed by their element 1 which is followed by their element 2, etc… Two-dimensional arrays follow a similar pattern when stored in memory: the one row following the other. In other words, the elements from row 0 are followed by the elements from row 1, which are followed by elements of row 2, and so on. Thus, a two dimensional array, representing a n x n matrix, has n^2 elements, and the base pointer A, contains the address of the first element of the array, i.e., the 0^{th} element of row 0.

Because of this regular pattern, accessing a two dimensional array element can be done in a random fashion, where the address of A_{ij} = A + L (i * n + j), where L is the size (in bytes) of each array element. For example, when L = 1, as it is for this assignment, then the element A_{32} can be found at address A + 1 (3 * 4 + 2) = A + 14.

In this question, you are asked to rotate a matrix 90 degrees clockwise. One way to do this is to first transpose the matrix then to reverse its columns.

Wikipedia says that, in linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal, i.e., it switches the row and column indices of the matrix by producing another matrix denoted as A^T. Thank you, Wikipedia.

Here is an example where A^T is the transpose of matrix A (using the diagonal "1, 6, -3, -8"):

$$A = \begin{bmatrix} 1 & -2 & 3 & -4 \\ -5 & 6 & -7 & 8 \\ -1 & 2 & -3 & 4 \\ 5 & -6 & 7 & -8 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & -5 & -1 & 5 \\ -2 & 6 & 2 & -6 \\ 3 & -7 & -3 & 7 \\ -4 & 8 & 4 & -8 \end{bmatrix}$$

We reverse the columns of the transpose matrix A^T, by swapping the last column with the first column, the penultimate column with the second column, etc...

Using the same example as above, here is what A^T looks like once it has been reversed. We call this final matrix A':

$$A^T = \begin{bmatrix} 1 & -5 & -1 & 5 \\ -2 & 6 & 2 & -6 \\ 3 & -7 & -3 & 7 \\ -4 & 8 & 4 & -8 \end{bmatrix} \qquad A' = \begin{bmatrix} 5 & -1 & -5 & 1 \\ -6 & 2 & 6 & -2 \\ 7 & -3 & -7 & 3 \\ -8 & 4 & 8 & -4 \end{bmatrix}$$

As you can see, A' is the rotated version of A (A has been rotated by 90 degrees clockwise).

Your task is to implement these two functions in x86-64 assembly code:

```
void transpose(void *, int );
void reverseColumns(void *, int n);
```

When they are called in this order, using a two dimensional array as their first argument, the effect will be to rotate this array by 90 degrees clockwise.

Download `Assn5-Files_Q3.zip`, expand it and open the files (`makefile`, `main.c` and an incomplete `matrix.s.`). Have a look at `main.c` and notice its content. Have a look at `matrix.s`. It contains functions manipulating matrices such as `copy`, `transpose` and `reverseColumns`. You need to complete the implementation of the functions `transpose` and `reverseColumns`. The function `copy` has already been implemented for you. You may find hand tracing its code useful. You may also want to "make" the given code and see what it does.

Requirements:

- Your code must be commented such that others (i.e., TA's) can read your code and understand what each instruction does.
    - About comments:
        - Here is an example of a useful comment:
          ```
          cmpl   %edx, %r8d     # loop while j < N
          ```
        - Here is an example of a **not** so useful comment:
          ```
          cmpl   %edx, %r8d     # compare %edx with %r8d
          ```
          Do you see the difference? Make sure you write comments of type 1.
- You must add a header comment block to the file `matrix.s`. This header comment block must include he filename, the purpose/description of its functions, your name and the date.
- You must use the makefile provided when compiling your code. This makefile cannot be modified.
- You cannot modify the code that has been supplied to you in the zip file. This signifies that, amongst other things, you must not change the prototype of the functions given. The reason is that these functions may be tested using a test driver built based on these function prototypes.
- Your code **must** compile and successfully execute on the computers in CSIL (using the Linux platform).
- You must follow the x86-64 function call and register saving conventions described in class and in the textbook.
- Do not push/pop registers unless you make use of them in your function. Memory accesses are expensive!

Submission:

- Electronically submit your file `matrix.s` via CourSys.
- Also include a hardcopy of your `matrix.s` in this assignment. Make sure it is the version of `matrix.s` that compiles and executes on a CSIL Linux machine.

```
# Name: matrix.s

# Junchen Li
```

```
# 2020.3.6
# 301385486
.globl    copy
   copy:
   # A in rdi, C in rsi, N in edx
      xorl %eax, %eax           # set eax to 0
   # since this function is a leaf function, no need to save caller-saved registers rcx and r8
      xorl %ecx, %ecx           # row number i is in ecx -> i = 0

   # For each row
   rowLoop:
      movl $0, %r8d           # column number j in r8d -> j = 0
      cmpl %edx, %ecx           # loop as long as i - N < 0
      jge doneWithRows

   # For each cell of this row
   colLoop:
      cmpl %edx, %r8d           # loop as long as j - N < 0
      jge doneWithCells

   # Compute the address of current cell that is copied from A to C
   # since this function is a leaf function, no need to save caller-saved registers r10 and r11
      movl %edx, %r10d       # r10d = N
         imull %ecx, %r10d    # r10d = i*N
      addl %r8d, %r10d       # j + i*N
      imull $1, %r10d           # r10 = L * (j + i*N) -> L is char (1Byte)
      movq %r10, %r11           # r11 = L * (j + i*N)
      addq %rdi, %r10          # r10 = A + L * (j + i*N)
      addq %rsi, %r11          # r11 = C + L * (j + i*N)

   # Copy A[L * (j + i*N)] to C[L * (j + i*N)]
      movb (%r10), %r9b        # temp = A[L * (j + i*N)]
      movb %r9b, (%r11)        # C[L * (j + i*N)] = temp

      incl %r8d                # column number j++ (in r8d)
      jmp colLoop              # go to next cell

   # Go to next row
   doneWithCells:
      incl %ecx                # row number i++ (in ecx)
      jmp rowLoop              # Play it again, Sam!

   doneWithRows:               # bye! bye!
      ret

   ####################

      .globl    transpose
   transpose:
   # A in rdi, C in rsi, N in edx, i (roll)in rcx, j (column) in r8
      xorl %eax, %eax          # set eax to 0
      xorl %ecx, %ecx          # set j to 0

   columnLoops:
     movl $0, %ecx          # Row number i in rcx change to zero
     cmpl %edx, %r8d        # go to the loop when j<n
     jge finishWithColumns    # if already fill-in all elements then jump to ret


   rowLoops:
     cmpl %edx, %ecx        # go to the loop if i < N
     jge finishWithRows      # finished the row fill-in then back to outside loop
   # compute the address for every cell and swap them with opposite order of i with j
   # ex  C[1][2]->C[2][1]  C[2][1]->C[1][1]
     movl %edx, %r10d       # r10d = N
     movq %r10, %r11        # r11 = N
     imull %ecx, %r10d      # r10 = N * i
     addl %r8d, %r10d       # r10 = N * i + j
     imull $1, %r10d        # r10 = L * (N * i + j )
     addq %rsi, %r10        # r10 = C + L * (N * i + j)  ex. C[1][2]
     imull %r8d, %r11d      # r11 = N * j
     addl %ecx, %r11d       # r11 = N * j + i
     imull $1, %r11d        # r11 = L * (N * i + j )
     addq %rsi, %r11        # r11 = C + L * (N * j + i)  ex. C[2][1]

   # Copy the r10 into the temp value which sotres the value. Then swap two value
   # C [ L * (N * i + j) ] = C [ L * (N * j + i) ]


     movb (%r10), %r9b      # temp1 = r10 = C [ L * (N * i + j) ]
     movb (%r11), %r15b     # temp2 = r11 = C [ L * (N * j + i) ]
     movb %r15b, (%r10)     # C [ L * (N * i + j) ] = C [ L * (N * i + j) ]
     movb %r9b, (%r11)      # C [ L * (N * j + i) ] = C [ L * (N * j + i) ]
```

```
   incl %ecx              # increase the number of i (row)
   jmp rowLoops           # go to the loop of row

finishWithRows:
   incl %r8d              # increase the number if j (column)
   jmp columnLoops        # go to the loop for col

finishWithColumns:
   ret                    # finish all tasks and return



#####################
   .globl    reverseColumns
reverseColumns:
# A in rdi, C in rsi, N in edx, i (colunm)in rcx, j (roll) in r8
   xorl %eax, %eax        # set the return value to zero
   xorl %ecx, %ecx        # set the ecx value to zero
   movl %edx, %ebx        # move the value N into the stack
   subl $2, %ebx          # Decrease the value N to N - 2

.L3:
   movl $0, %r8d          # set r8 to the value of zero (the value of j )
   cmpl %ebx, %ecx        # go to the loop as long as i < (N - 2)
   jge .L4                # if it is greater or equal then jump tp L4

.L1:
   cmpl %ebx, %r8d          # go to the loop as long as j < (N - 2 )
   jge .L2                  # go to the loop L2 if greater or equal



   movl %edx, %r10d         # r10 = N
   imull %ecx, %r10d        # r10 = N * i
   movq %r10, %r11          # r11 = i * N
   addl %r8d, %r10d         # r10 = N * i + j
   imull $1, %r10d          # r10 = 1 * (j + i * N)
   addq %rsi, %r10          # r10 = C + 1 * (j + i * N)
   addl %edx, %r11d         # r11 = i * N + N
   subl %ecx, %r11d         # r11 = i * N + N - i
   subl $1, %r11d           # r11 = i * N + N - i - 1
   imull $1, %r11d          # r11 = 1 * ( i * N + N - i - 1 )
   addq %rsi, %r11          # r11 = C + 1 * ( i * N + N - i - 1 )

# Copy the C + 1 * (j + i * N) = C + 1 * ( i * N + N - i - 1 )
   movb (%r10), %r9b        # temp 1 = C + 1 * (j + i * N)
   movb (%r11), %r13b       # temp 2 = C + 1 * ( i * N + N - i - 1 )
   movb %r13b, (%r10)       # C + 1 * (j + i * N) = C + 1 * ( i * N + N - i - 1 )=temp2
   movb %r9b, (%r11)        # C + 1 * ( i * N + N - i - 1 ) = temp1
   incl %r8d                # increase the j++
   jmp .L1                  # jump to the L1

.L2:
   incl %ecx                # increase the value of i, i++
   jmp .L3                  # jump to the L3
```

```
.L4:
   ret                    # finish all task then return
```