Assignment 6 – SOLUTION

Assignment 6 - Objectives:

- Manipulating 2D arrays (matrices) in x86-64 assembly code
- Function call convention in x86-64 assembly code

For each of our team-based assignments, you can either:

- Create a group with the same partner,
- Change your group (select a different partner), or
- Decide to work on you own.

Group of two (2):

- You can do Assignment 6 in a group of two (2) or you can work on your own.
- If you choose to work in a group of two (2):
    - Crowdmark will allow you to select your teammate (1). Both of you will only need to submit one assignment and when the TAs mark this one assignment, the marks will be distributed to both of you automatically (just like on CourSys).
    - You can always work with a student from the other section, but both of you will need to submit your assignment separately, i.e., Crowdmark will not consider the both of you as a group.
    - You will need to form the same group of 2 on CourSys.
- If you choose to work on your own:
    - You will need to form a group of 1 on CourSys. (This is the way CourSys works.)

Marking scheme:

- This assignment will be marked for correctness.
- A solution will be posted on Monday after the due date.

Deadline:

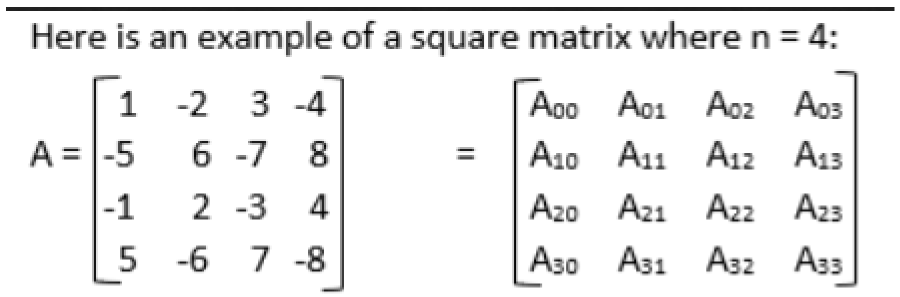
- Friday March 10 at 23:59:59 on Crowdmark and CourSys.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Enjoy!

Q1 (20 points)

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.

Click here for an example of a square matrix where N = 4.



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

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

Note the notation A_{ij} refers to the matrix entry at the ith<./super> row and the jth<./super> column of A. Each of the N rows of the matrix A resembles a one dimensional array in the programming language C, with the value of j increasing for each element.

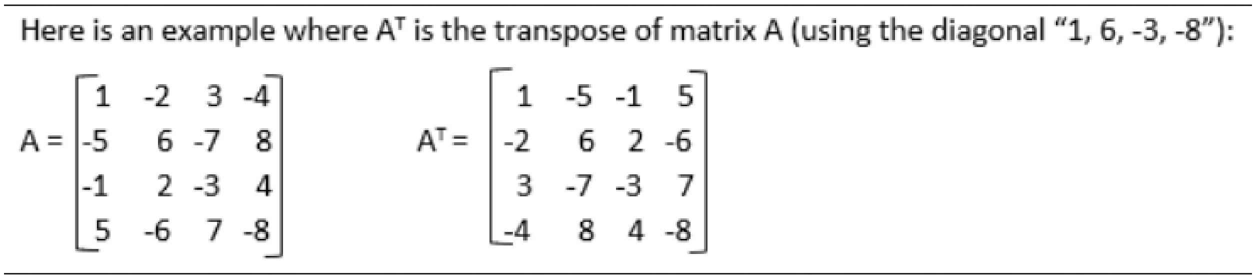
Because of this resemblance, matrices can be represented (modeled) in our C programs using two dimensional arrays. One dimensional arrays are stored in contiguous memory space, where their element 0 is followed by their element 1 which is followed by their element 2, etc... Two-dimensional arrays follow 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 N2 elements, and the base pointer A, contains the address of the (first byte of the) first element of the array, i.e., the 0th 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.

Click here to see an example where A^T is the transpose of matrix A (using the diagonal "1, 6, -3, -8").

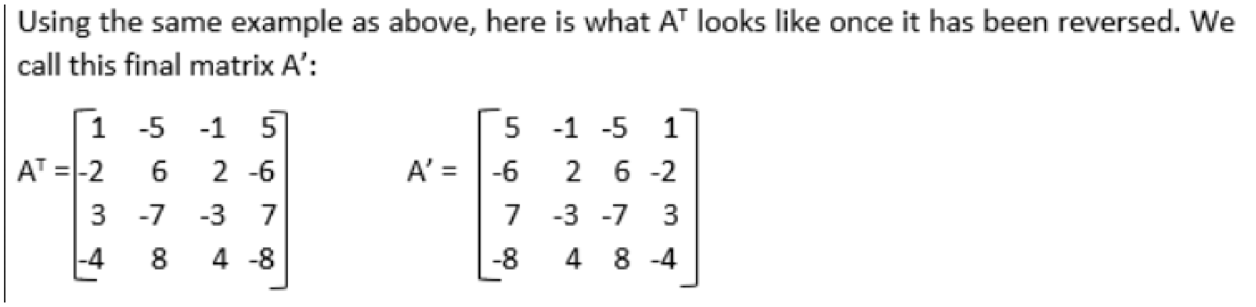


Here is an example where A^T is the transpose of matrix A (using the diagonal "1, 6, -3, -8"):

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

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, click here to see what A^T looks like once it has been reversed. We call this final matrix A'.



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 `Assn6_Files.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. During our next lecture, we shall be looking into both versions of the `copy` function found in `copy_Version_1.s` and in `copy_Version_2.s` posted on our course web site.

Finally, you may want to build the provided code using the `makefile` 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:
        - Comment of Type 1: Here is an example of a useful comment:

          ```
          cmpl  %edx, %r8d          # loop while j < N
          ```

        - Comment of Type 2: Here is an example of a **not so** useful comment:

          ```
          cmpl  %edx, %r8d          # compare %r8d to %edx
          ```

    - Make sure you write comments of Type 1. :)
- You cannot modify the prototype of any of the functions nor can you modify the code of the function `copy` provided in `matrix.s`. The reason is that these functions may be tested using a test driver built based on these function prototypes.
- You must add a header comment block to the file `matrix.s`. This header comment block must include the filename, the purpose/description of its functions, your name, your student number and the date.

- You are free to name your label(s) as you wish, however, make sure they are descriptive.
- 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 need them and make use of them in your function and their content needs to be preserved. Memory accesses are expensive!
- You must use the `makefile` provided when compiling your code. This `makefile` cannot be modified.
- Your code must compile and execute on our ***target machine*** and solve the problem, i.e., rotate an array (of size N by N) by 90 degrees clockwise.
- You cannot hard-code the value of N in your functions as we will test your code with different values of N.

Lastly, once your code compiles, executes and solves the problem on our ***target machine***, submit your `matrix.s` on **CourSys** and write the word **Done!** in the answer space below so that Crowdmark does not think you haven't completed this question of Assignment 6. Do not upload your file in the answer space below.

```
Solution:
     .globl    copy
# ***** Version 2 *****
copy:
# A in rdi, C in rsi, N in edx

# This function is not a "caller" i.e., it does not call functions
(it is a leaf function),
# hence it does not have the responsibility of saving "caller-
saved" registers
# such as rax, rdi, rsi, rdx, rcx, r8 and r9.
# This signifies that it can use these registers without first
saving their content.

# Set up registers
     xorl %eax, %eax                # set eax to 0
     xorl %ecx, %ecx                # i = 0 (row index i is in ecx)
```

```
# For each row
rowLoop:
     xorl %r8d, %r8d               # j = 0 (column index j in r8d)
     cmpl %edx, %ecx               # while i < N (i - N < 0)
     jge doneWithRows

# For each cell of this row
colLoop:
     cmpl %edx, %r8d               # while j < N (j - N < 0)
     jge doneWithCells

# Copy the element A points to to C (in same cell location)
     movb (%rdi), %r9b       # temp = element A points to
     movb %r9b, (%rsi)       # cell location C points to = temp

# Update A and C so they point to their next element/cell location
     incq %rdi
     incq %rsi

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

# Go to next row
doneWithCells:
     incl %ecx                # i++ (row index in ecx)
     jmp rowLoop                   # go to next row

doneWithRows:                 # bye! bye!
     ret

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

# # ***** Version 1 - Original version *****
#    .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:
# void transpose(void *, int ); transpose(C, N);
# C in rdi, N in esi

# This function is not a "caller" i.e., it does not call functions
(it is a leaf function),
# hence it does not have the responsibility of saving "caller-
saved" registers
# such as rax, rdi, rsi, rdx, rcx, r8, r9, r10 and r11.
# This signifies that it can use these registers without first
saving their content.

# This function is a "callee" hence it does have the responsibility
of saving callee-saved" registers
# such as r12 before using it and the responsibility of restoring
it before returning to "caller" (main).
     pushq %r12

# Set up registers
     xorl %eax, %eax               # set eax to 0
     xorl %ecx, %ecx               # i = 0 (row index i is in ecx)

     movl %esi, %edx               # save N (into %edx)
     decl %edx                # N-1 now in %edx

# For each row
rowLoopT:
     cmpl %edx, %ecx               # while i < N-1 (i - N-1 < 0)
     jge doneWithRowsT

     leal 1(%ecx), %r8d       # j = i+1 (column index j in r8d)

# For each cell of this row
colLoopT:
     cmpl %esi, %r8d               # while j < N (j - N < 0)
     jge doneWithCellsT

# Transpose (swap) cell C + L(j + i*N) with  C + L(i + j*N)
# Compute the address of cell 1 -> C + L(j + i*N)
     movl %esi, %r9d         # temp (r9d) = N
     imull %ecx, %r9d        # temp (r9d) = i*N
     addl %r8d, %r9d         # temp (r9d) = j + i*N
#    imull $1, %r9d          # temp (r9d) = L(j + i*N) - no need to
do this since L=1 (char=1B)
     addq %rdi, %r9          # temp (r9) = C + L(j + i*N)

# Save cell 1 to temp (r10b)
     movb (%r9), %r10b

# Compute the address of cell 2 (the transpose) -> C + L(i + j*N)
     movl %esi, %r11d         # temp (r11d) = N
     imull %r8d, %r11d         # temp (r11d) = j*N
     addl %ecx, %r11d         # temp (r11d) = i + j*N
#    imull $1, %r11d          # temp (r11d) = L(i + j*N) - no need
to do this since L=1 -> char = 1B
```

```
     addq %rdi, %r11                # temp (r11) = C + L(i + j*N)

# Save cell 2 to temp (r12b)
     movb (%r11), %r12b

# Copy temp (r10b) to cell 2 (the transpose) -> C + L(i + j*N)
     movb %r10b, (%r11)

# Copy temp (r12b) to cell 1 -> C + L(j + i*N)
     movb %r12b, (%r9)

     incl %r8d                 # j++ (column index in r8d)
     jmp colLoopT              # go to next cell

# Go to next row
doneWithCellsT:
     incl %ecx                 # i++ (row index in ecx)
     jmp rowLoopT              # go to next row

doneWithRowsT:
# Stack clean up
     popq %r12                # restore "callee-saved" register
before returning to "caller" (main)
     ret


#####################
     .globl    reverseColumns
# ***** Version 2 *****
reverseColumns:
# void reverseColumns(void *, int n); reverseColumns(C, N);
# C in rdi, N in esi

# This function is not a "caller" i.e., it does not call functions
(it is a leaf function),
# hence it does not have the responsibility of saving "caller-saved"
registers
# such as rax, rdi, rsi, rdx, rcx, r8, r9, r10 and r11.
# This signifies that it can use these registers without first
saving their content.

# This function is a "callee" hence it does have the responsibility
of saving callee-saved" registers
# such as r12 and r13 before using it and the responsibility of
restoring it before returning to "caller" (main).
     pushq %r12
```

```
	pushq %r13

# Set up registers
	xorl %eax, %eax			# set eax to 0

	movl %esi, %edx			# save N into %edx
	decl %edx			# N-1 now in %edx
	movl %esi, %r13d		# save N into %r13d
	shrl $1, %r13d			# N/2 now in %r13d

	xorl %r8d, %r8d			# j = 0 (column index j in r8d)

# For each column
nextColumnPairLoop:
	cmpl %r13d, %r8d		# while j < floor(N/2) (i - floor(N/2)
< 0)
	jge doneReversing

# Set up loop variable i
	xorl %ecx, %ecx			# i = 0 (row index i is in ecx)

# Reverse the cells of this row
# Compute the address of cell 1 -> C + j
	movl %r8d, %r9d		# temp (r9d) = j
	addq %rdi, %r9		# temp (r9) = C + L(j + i*N), but L=1
and i=0

# Compute the address of cell 2 (the reverse) -> C + (N-1-j)
	movl %edx, %r11d		# temp (r11d) = N-1
	subl %r8d, %r11d		# temp (r11d) = N-1-j
	addq %rdi, %r11			# temp (r11) = C + L((i*N) + (N-
1-j)), but L=1 and i=0

# Swap each cell in these 2 columns
nextElementPairLoop:
	cmpl %esi, %ecx			# while i < N (i - N < 0)
	jge doneWithColumnPair

# Save cell 1 to temp (r10b)
	movb (%r9), %r10b
# Save cell 2 to temp (r12b)
	movb (%r11), %r12b
# Copy temp (r10b) to C+ L((i*N) + (N-1-j))
	movb %r10b, (%r11)
# Copy temp (r12b) to C + L(j + i*N)
	movb %r12b, (%r9)
```

```
# Go to next element in both columns ...
    leaq (%rsi, %r9), %r9   # ... by adding N to address of current
element of column
    leaq (%rsi, %r11), %r11 # ... by adding N to address of current
element of column

     incl %ecx                # i++
     jmp nextElementPairLoop  # go to next element

# Go to next row
doneWithColumnPair:
     incl %r8d                # j++
     jmp nextColumnPairLoop   # go to next row

doneReversing:
# Stack clean up - reverse order
     popq %r13               # restore "callee-saved" register
before returning to "caller" (main)
     popq %r12
     ret

# #####################
#    .globl    reverseColumns
# # ***** Version 1 *****
# reverseColumns:
# # void reverseColumns(void *, int n); reverseColumns(C, N);
# # C in rdi, N in esi

# # This function is not a "caller" i.e., it does not call functions
(it is a leaf function),
# # hence it does not have the responsibility of saving "caller-
saved" registers
# # such as rax, rdi, rsi, rdx, rcx, r8, r9, r10 and r11.
# # This signifies that it can use these registers without first
saving their content.

# # This function is a "callee" hence it does have the
responsibility of saving callee-saved" registers
# # such as r12 and r13 before using it and the responsibility of
restoring it before returning to "caller" (main).
#    pushq %r12
#    pushq %r13

# # Set up registers
#    xorl %eax, %eax              # set eax to 0
#    xorl %ecx, %ecx                # i = 0 (row index i is in ecx)

#    movl %esi, %edx                # save N into %edx
#    decl %edx                 # N-1 now in %edx
#    movl %esi, %r13d          # save N into %r13d
#    shrl $1, %r13d            # N/2 now in %r13d

# # For each row
# rowLoopR:
#    cmpl %esi, %ecx                # while i < N (i - N < 0)
#    jge doneWithRowsR

#    xorl %r8d, %r8d                # j = 0 (column index j in r8d)

# # For each cell of this row
# colLoopR:
#    cmpl %r13d, %r8d          # while j < floor(N/2) (i - floor(N/2)
< 0)
#    jge doneWithCellsR

# # Reverse the cells of this row
# # Compute the address of cell 1 -> C + L(j + i*N)
#    movl %esi, %r9d          # temp (r9d) = N
#    imull %ecx, %r9d         # temp (r9d) = i*N
#    addl %r8d, %r9d          # temp (r9d) = j + i*N
# #  imull $1, %r9d           # temp (r9d) = L(j + i*N) - no need to
do this since L=1 (char=1B)
#    addq %rdi, %r9           # temp (r9) = C + L(j + i*N)

# # Save cell 1 to temp (r10b)
#    movb (%r9), %r10b

# # Compute the address of cell 2 (the reverse) -> C + L((i*N) + (N-
1-j))
#    movl %esi, %r11d          # temp (r11d) = N
#    imull %ecx, %r11d          # temp (r11d) = i*N
#    movl %edx, %r12d           # temp (r12d) = N-1
#    subl %r8d, %r12d           # temp (r12d) = N-1-j
#    addl %r12d, %r11d          # temp (r11d) = (i*N) + (N-1-j)
# #  imull $1, %r11d           # temp (r11d) = L(i + j*N) - no need to
do this since L=1 -> char = 1B
#    addq %rdi, %r11                # temp (r11) = C + L((i*N) + (N-
1-j))

# # Save cell 2 to temp (r12b)
#    movb (%r11), %r12b
# # Copy temp (r10b) to C+ L((i*N) + (N-1-j))
#    movb %r10b, (%r11)

# # Copy temp (r12b) to C + L(j + i*N)
#    movb %r12b, (%r9)

#    incl %r8d                    # j++ (column index in r8d)
#    jmp colLoopR                 # go to next cell

# # Go to next row
# doneWithCellsR:
#    incl %ecx                    # i++ (row index in ecx)
#    jmp rowLoopR                 # go to next row

# doneWithRowsR:
# # Stack clean up - reverse order
#    popq %r13                   # restore "callee-saved" register
before returning to "caller" (main)
#    popq %r12
#    ret
```