ASSEMBLY LANGUAGE PROJECTS 2017

Contents

Common Requirements

- Your program must be divided into **PROCs**, each of which is responsible for one and only one functionality. For example,
  - ReadArray: reads an array
  - SortArray: sorts a given array.
  - ViewImage: views an image.
- Your PROCs should be transparent for input parameters, do not forget the **USES** operator.
- Don't use any hardcoded values, instead use **constants** and operators
  - For example: Fetch array length by LENGTHOF operator… etc.
- The bonus items will not be counted unless the original project is complete.
- If the GUI is necessary for your project; **here** you are a link to a simple tutorial for how to link Assembly code with high-level language.
- Keep your code **clean**, follow a specific **convention**, and choose **meaningful** identifiers (variables and procedure names).
- All projects must be submitted with a **printed documentation**. In your documentation, draw the **flow chart** of your logic and the **procedures hierarchy** (refer to chapter 5).
- [Code like a professional]: Your code **must** be well **documented**; you should use commenting style like this:

```
;---------------------------------------------
;Calculates: Sum of an integer array
;Receives: ESI Contains the offset of the Array
;   ECX Contains the length of the Array
;Returns:  EAX contains Array Sum
;--------------------------------------------
SumArr PROC
```

1. Inline Assembly Emulator

Write an Assembly program that accepts Intel instructions as strings and executes them.

- **Details**

  Instructions:

  - Instruction format {Mnemonic {Operands}; comments}
  - Instructions and operands are case insensitive.
  - Instruction list: INC, ADD, SUB, MOV, MOVZX, MUL, DIV, call dumpregs, call writeint.
  - The operands will be the 8 general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP), including their smaller parts (AX, AL, AH… etc.) And all of the 32-bit registers should have an initial value equals 1 in the beginning before executing any instruction.
  - Immediate values should be supported as operands. ***(Only decimal numbers)***.
  - The emulator should handle any number of spaces between mnemonics and operands.
    Ex: mov   eax   ,   ebx
  - The user should be able to execute several instructions at once separated by a new line.
  - The user should be able to write a single line comment.

  Errors:

  The emulator should display an error message for the following cases of:

  1. An unsupported instruction.
  2. The instruction operands are invalid or don't have the same size.
  3. In case of MOVZX, an error should be displayed if the first operand is not bigger than the second one.

  - **You should use the error statements used in the Microsoft Visual Studio.**

  Warning:

  Your emulator should give a warning to the user when the result of an **addition** operation, **increment** operation, **multiplication** operation or **division** operation will overflow the specified destination, and ask him whether to continue with the execution or not

  Ex:

  >>ADD AL, BL   ; where AL = 100, and BL = 200

  Warning: The result of this operation won't fit in the specified destination, would you still like to continue? (Y/N)

  >>Y   // Execute the instruction and display the results

  >>N   //Don't execute the instruction and display the old registers values

- **Input**
  The user will enter the instructions

- **Output**
  The values of all registers after the execution

- **Bonus**
  - Accepting constant characters and other types of numbers (hex, binary) as operands and handling their cases.
  - Reading assembly code from a file.

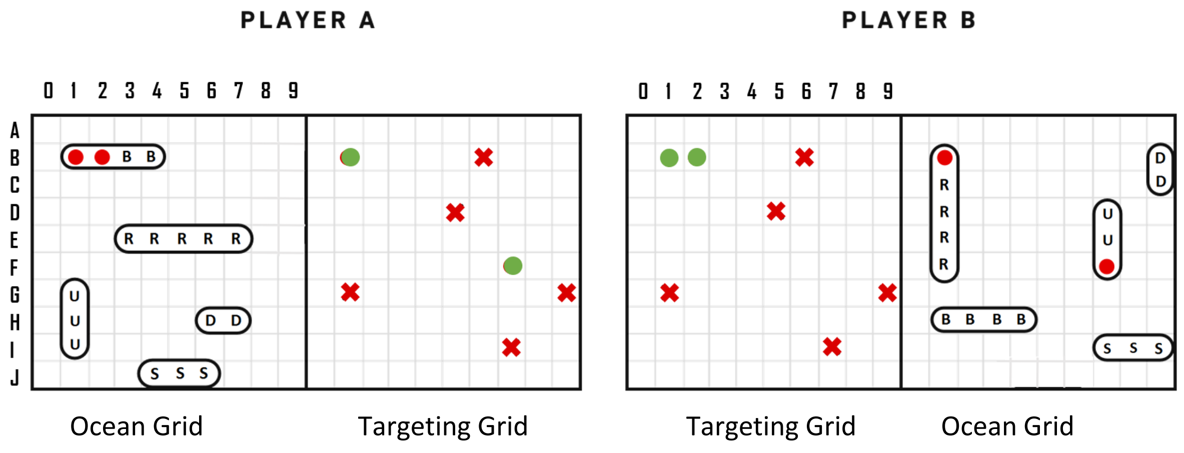
- **Team members:** 4 – 5 members.
- **Responsible TA:** Dr. Mona Sadik

2. Battleship

The battleship is a two-player guessing game that dates back from World War I. It was originally a paper-and-pencil game and was later made into the well-known board game.

The objective of Battleship is to try and sink all the other player's before they sink all your ships. All the other player's ships are somewhere on his/her board. You try and hit them by calling out the coordinates of one of the squares on the board.

Each player has exactly 2 boards: (1) an ocean (primary) grid, to place their own ships, (2) a targeting (tracking) grid, to keep up with the hits and misses at the opponent. This means that there is a total of four 10x10 boards: 1 ocean and 1 targeting grid for each player.



Each player has exactly 5 specific ships and places the 5 ships somewhere on their board. **Ship placement rules**:

1. The ships can be placed either vertically or horizontally.
2. Diagonal placement is not allowed.
3. No part of a ship may extend out of the board (All ships must fit inside the board).
4. Ships may not overlap each other partially or entirely.
5. Once the game begins, the players cannot move the ships.

The 5 ships are:

1. R (occupies 5 spaces)
2. B (4)
3. U (3)
4. S (3)
5. D (2)

- **Details:**

Players take turns guessing by calling out the coordinates. The opponent responds with "hit" or "miss" as appropriate. Both players should mark their board with different marks, according to what happened. For example, if you call out F6 and your opponent does not have any ship located at F6, your opponent would respond with "miss". You record the miss F6 by placing a red X on your targeting grid at F6.

When all the squares that one of a player's ships occupies have been hit, the ship will be sunk. That player should then announce, "hit and sunk".

As soon as all of one player's ships have been sunk, the game ends.

- **What to do?**

You will create a 1-player battleship game, in which a single **player** plays against a **computer**.
<u>The game will go as follows:</u>

1) Placing ships:
   - The <u>player</u> will enter a **pair** of coordinates, for each ship. *Example: for ship "U", the player may enter an input "B3 B5". This will place ship U at coordinates B3, B4 and B5, by placing 3 Us at those coordinates.*
   - Each ship has a specific size as defined above. The program should display an **error** message and receive a new input, if:
     - The player enters coordinates with a size different from the specified size for that ship.
     - Any ship placement rule is violated for any ship.
   - For the <u>computer</u>, your program should place its ships randomly for each game, making sure to follow all the ship placement rules.
2) Starting the game:

   a. <u>The player fires:</u>
   - Read a coordinate (e.g., A5).
   - Mark this coordinate in the computer's ocean grid and identify if it is a hit or miss.
   - If the fire is a HIT, display which ship the player hits. Place a green O on the player's targeting grid at this coordinate. If all coordinates that belong to a ship have been HIT, then this ship is sunk. Display which ship was sunk.
   - If MISS, display that it was a MISS. Place a red X on the player targeting grid at this coordinate.
   - It is now the computer's turn to play.

   b. <u>Computer fires:</u>
   - Computer checks its targeting board, and <u>randomly</u> selects a coordinate to hit (that has not been hit before). The program should NOT use the places of the player's ships.
   - Whether it hits or not, follow the same logic as specified when the player fires.

   c. Continue steps (a) and (b) until the game ends.

3) The game ends when either the computer or the player hits all the places on all the opponent's ships. Display who won.

- **Design and regulations:**

- All boards are 10x10 in size.
- Only the **player's** 2 boards should be displayed on the console. The computer's boards should not be displayed.
- All the boards that will appear on the console (refer to previous point) should be numbered as shown in the first board in Figure 1. (Columns are numbered from 0 to 9. Rows are numbered from A to J.)
- After each step in the game, refresh the player's 2 boards on the console with any new changes.
- Display each of the ships in a different color.

- For both player or computer, in the ocean grid:
    - each ship is represented by an array of characters of the ship. The size of the array is given by the number of places in the ship.
    - when a place in a ship is hit, that coordinate is replaced with an “*”. The ***** is colored **red** if displayed on the console.
- For both player or computer, in the targeting grid:
    - when a player/computer hits, that coordinate is replaced with an “O”. The “**O**” is colored **green** if displayed on the console.
    - when a player/computer misses, that coordinate is replaced with a “X”. The “**X**” is colored **red** if displayed on the console.
- Refer to Section 5.4.3 in the textbook to know how to write in a specific coordinate (Gotoxy) and change the text color (SetTextColor) and other supporting Irvine functions.

- **Things to take care of:**

- No one can fire at the same coordinate twice (whether it was a HIT or MISS). If the player tries to fire at the same coordinate again, display an error message, and ask for another input. The computer should never attempt to fire at the same coordinate twice.
- No one can fire at a coordinate outside of the board.

- **Bonus:**

- Create a GUI (Windows Forms) for the game.

- **Team members:** 4 – 5 members.
- **Responsible TA:** Dr. Marwah Helaly
- **References (read for better understanding):**
    - https://www.youtube.com/watch?v=q0qpQ8doUp8
    - https://www.thespruce.com/the-basic-rules-of-battleship-411069
    - https://www.cs.nmsu.edu/~bdu/TA/487/brules.htm

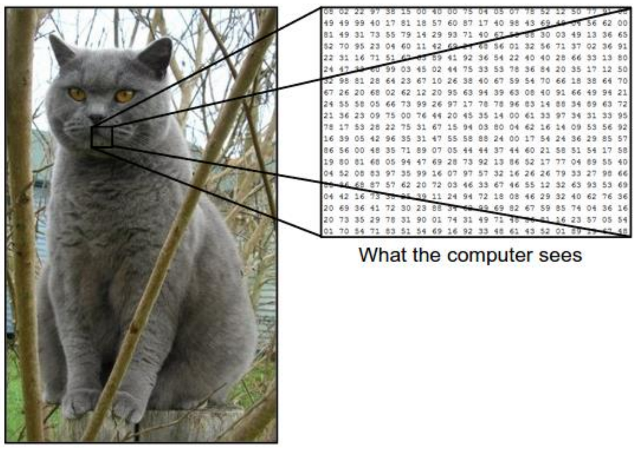
3. Image Processing Package

Our Image Processing Package includes three tasks:

1. Noise removal in Image processing using the median filter.
2. Grayscale convertor.
3. Brightness adjustment.

What is a computer image?

A computer image is a digitized version of a picture taken with a capturing device like the camera or it is the electronic visual representation of pictures which is stored in a computer. As we all know that the computer understands 1's and 0's only, so everything you see on the computer screen is mapped to binary number(s) and of course a computer image is mapped to some binary numbers. A computer image stored as a 2D array of pixels, each pixel consists of three color channels Red, Green and Blue (RGB) each color channel takes a value between 0 – 255 inclusive. Mixing decimal values of these color channels gives us the different color that we see on the screen.



A portion of an image showing what the computer sees

1. Noise removal in Image processing using the median filter

The median filter is a nonlinear digital filtering technique, often used to remove noise from an image or signal. Such noise reduction is a typical pre-processing step to improve the results of later processing (for example, edge detection on an image). Median filtering is very widely used in digital image processing because, under certain conditions, it preserves edges while removing noise.

The main idea of the median filter is to **sort** the pixel values in a neighborhood region with a certain **window size** (W_S), then choose the median of these values, and place it in the center of the window in a **new image**, see figure 1.

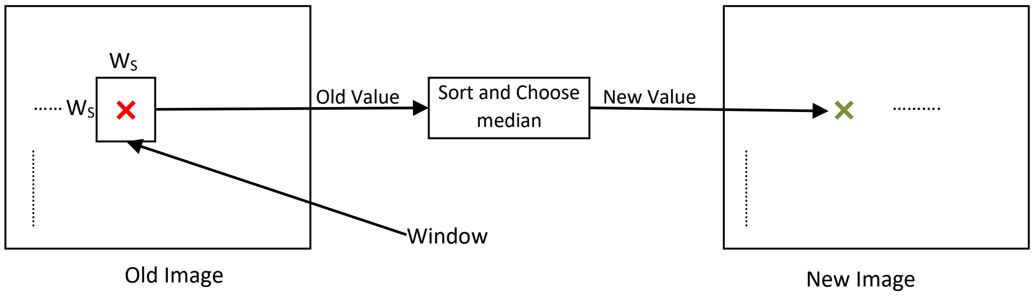


Figure 1: Main idea of median filter

As the window size increases, the effect of the filter is increased, as shown in figure 2.

1.1 Implementation steps of Median filter on Image

The idea is to **calculate** the median of neighboring pixel values. This can be done by repeating the following steps for **each** pixel (i, j) in the image:

1. Store the values of the neighboring pixels in an array. This array is called the **window**.
2. Sort this array using any sorting algorithm.
3. Pick the **median** from the window (array) as the **new** pixel value and place it in the center of the window in the new image, see figure 1.
4. Repeat steps 1 – 3 for the next pixel (i, j+1). If all pixels in row i are processed, move to the next row i+1 and so on for all pixels in the image.

Notes:

- Window size would be either 3 x 3 or 5 x 5.
- For a boundaries pixel, you have to pad it with the original pixel's value to fill the window.
- The median is the middle value in the window array after sorting.

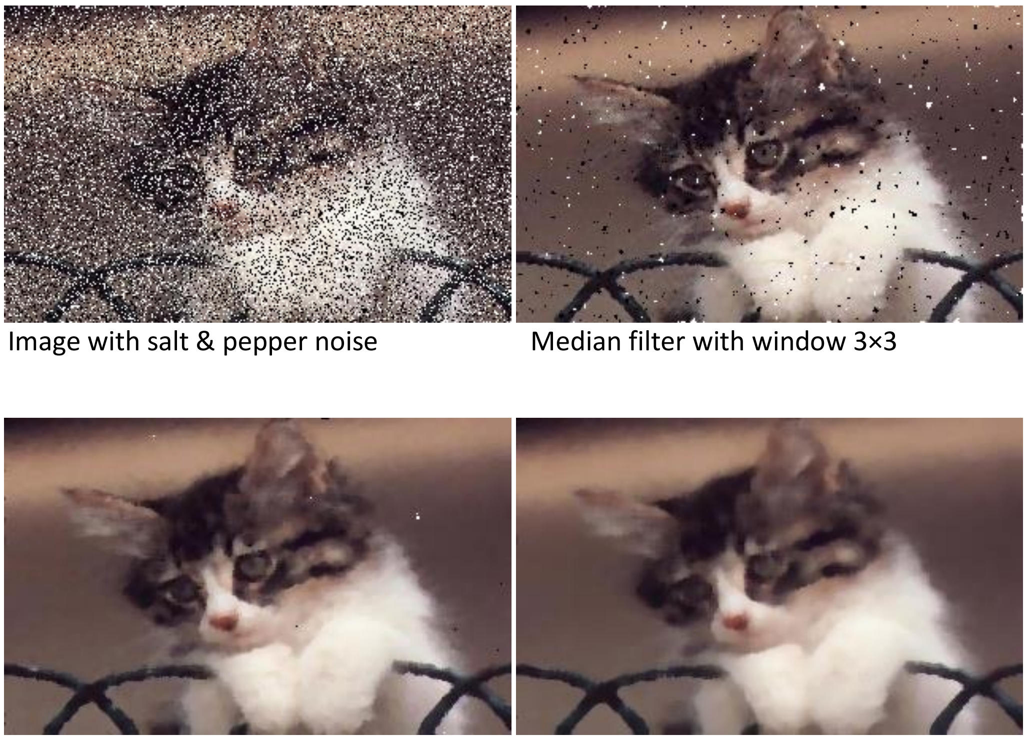


Image with salt & pepper noise
Median filter with window 3×3
Median filter with window 5×5
Median filter with window 7×7

Figure 2: Effect of the median filter with different window size

2. Grayscale convertor

Grayscale is a range of shades of gray without apparent color. The darkest possible shade is black, which is the total absence of transmitted or reflected light. The lightest possible shade is white, the total transmission or reflection of light at all visible wavelengths. Intermediate shades of gray are represented by equal brightness levels of the three primary colors (red, green and blue) for transmitted light, or equal amounts of the three primary pigments (cyan, magenta and yellow) for reflected light.

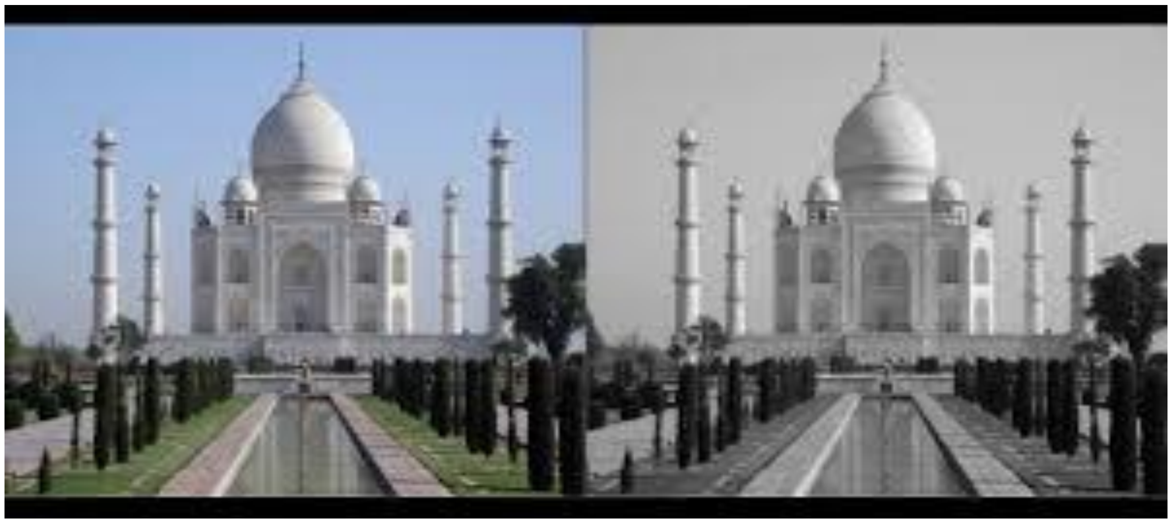


Figure 3: Effect of the grayscale

In the case of transmitted light (for example, the image on a computer display), the brightness levels of the red (R), green (G) and blue (B) components are each represented as a number from decimal 0 to 255, or binary 00000000b to 11111111b. For every pixel in a red-green-blue (RGB) grayscale image, R = G = B. The lightness of the gray is directly proportional to the number representing the brightness levels of the primary colors. Black is represented by R = G = B = 0 or R = G = B = 00000000b, and white is represented by R = G = B = 255 or R = G = B = 11111111b. Because there are 8-bit s in the binary representation of the gray level, this imaging method is called 8-bit grayscale.

When converting an RGB image to grayscale, we have to take the RGB values for each pixel and make as output a single value reflecting the brightness of that pixel. One such approach is to take the average of the contribution from each channel:

$\boxed{(R+B+C)/3.}$

However, since the perceived brightness is often dominated by the green component, a different, more "human-oriented", method is to take a weighted average,

$\boxed{0.3R + 0.59G + 0.11B}$

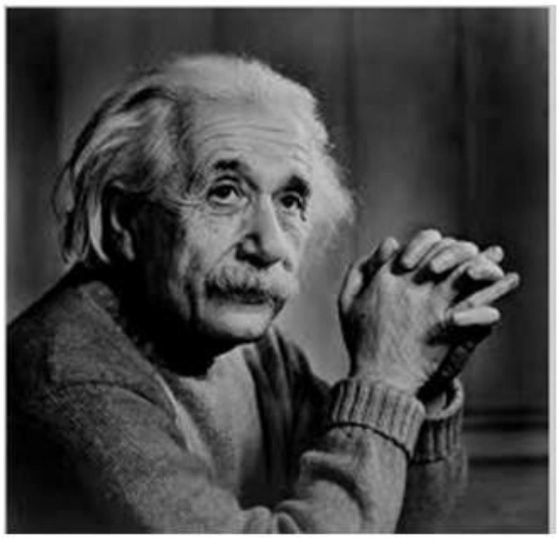
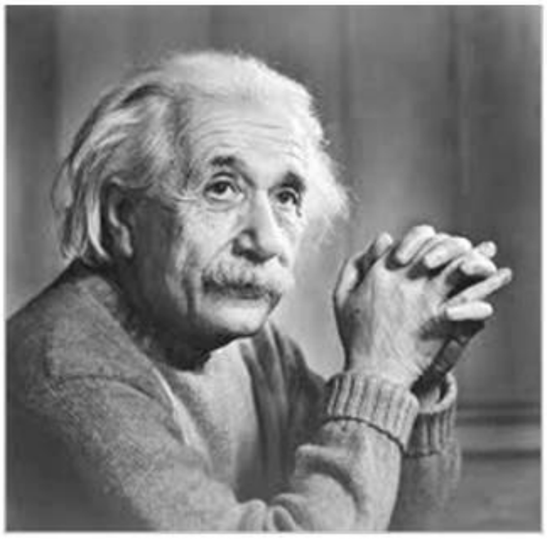
There is another simpler method, in which each of the Red, Green and Blue channels are set to the same value of any channel value.

Example: Suppose we have a pixel with the following values: Red = 128, Green = 20 and Blue = 200. The new pixel value after converting it to grayscale:

Red = 128, Green = 128 and Blue = 128 (note that we changed all of the channels values to the value of the Red channel.

3. Brightness Adjustment

Just have a look at both of these images, and compare which one is brighter.

We can easily see, that the image on the right side is brighter as compared to the image on the left.

But if the image on the right is made darker than the first one, then we can say that the image on the left is brighter than the left.

How to make an image brighter?

Brightness can be simply increased or decreased by a simple addition or subtraction, to the pixel values.

Pseudo Code:

```
foreach (pixel in Image)
{
        if (pixel + brightnessValue  < 255)
            pixel = pixel + brightnessValue;
        else
            pixel = 255;
}
```

Implementation Notes

Unfortunately, **Irvine** library doesn't support images, so we have to find a method to read pixel values from an image. We will integrate our assembly project's DLL in a C# project. The C# project will read an image and dump it into a decimal array (1D Array), then it will pass the array to the assembly functions to manipulate these pixel values.

You'll deliver two modules (C# & Assembly Project). The first module is a GUI desktop application written in C#, which has the following functionalities:

1- Open a bitmap image from the disk.
2- Save a bitmap to disk.
3- Convert an array of pixel values of an image to bitmap then view it.
4- Convert a bitmap image into an integer array.
5- The required GUI components to convert the image into grayscale, then display it.
6- The required GUI components to apply median filter to an image, then display it.
7- The required GUI components to adjust image brightness using a slider.

The second module is the assembly library (DLL) with the following functionalities:

1- A function that takes a 1D integer array of pixel values and a window size, then returns the same array after filtering it using median filter using the required window size.
2- A function that takes a 1D integer array of pixel values and returns the array after converting its values into grayscale.

3- A function that takes a 1D integer array of pixel values and a brightness value (+ve or –ve) then adds it to each pixel of the array to adjust its brightness.
4- Integrate that module with the C# desktop application project and use the required functionalities by calling the procedures from the DLL class library.

Bonus:

1- Make the whole system using **ASSEMBLY LANGUAGE ONLY** (search for other libraries).

Team Members: 4 – 6 members

Responsible TA: Dr. Donia Gamal and Dr. Ibrahim Amer

References

1- Project template will be made available

4. Mini Database

Write an assembly program that handles the basic tasks of a student database. The database should be handled through files. (i.e., student's data and their grades will be stored in files. Each record will be stored in a separate line and columns are separated by a delimiter.)
Your DB will support only one grade for each student and all students are registered in one course.

Functionality:

You're asked to implement the following functions in assembly:

Function	**Parameters**
EnrollStudent	Student ID, Student Name
UpdateGrade	Student ID, Integer Grade
DeleteStudent	Student ID
DisplayStudentData	Student ID
SaveDatabase[1]	File Name, DB Key
OpenDatabase[2]	File Name, DB Key
GenerateFullReport[3]	File Name, Sortby

[1] When saving a database, each data item should be written encrypted with a user key (a single byte). Use XOR encryption which xor each byte of the data item with the user key.
[2] When opening a database, the user can work on data previously saved and do all other functions on them. The user should enter the database key to be able to view its content.
[3] The full report will create a file with this header (Student ID, Student Name, Numeric Grade and Alphabetic Grade), and list all students data with their grades. The report should be **sorted** either by StudentID or by their numeric grade.
Alphabetic grade is calculated based on the following rules:
A 100 – 90, B 89 – 80, C 79 – 70, D 69 – 60, F Below 60

Input
The application should display a menu asking the user for the operation she wants to perform, asks for data then call the corresponding function in the assembly dll. In case of error it should notify the user.

Bonus:
- GUI implementation instead of using console application.
- Handling multiple courses with a separate grade for each.

Team Members: 4 – 5 members

Responsible TA: Dr. Sarah Osama

5. Matrix Calculator

Write an assembly program that calculates the basic matrix operations (addition, subtraction, multiplication, transpose and determinant).

Details:

Your program should contain a menu of matrix operations that the user can choose from.

A - Matrix Addition:

Adding two matrices and producing a new one, the two matrices should have the same dimension (M*N) where M is number of rows and N number of columns so that the user should enter the dimension first then elements of two matrices for example:

Input:

Enter the dimension of the matrix:

2

3

Enter the elements of the first matrix:

$$\begin{matrix} 4 & 5 & 2 \\ 1 & 4 & 9 \end{matrix}$$

Enter the elements of the Second matrix:

$$\begin{matrix} 2 & 49 & 1 \\ 0 & -9 & 3 \end{matrix}$$

Output:

$$\begin{matrix} 6 & 54 & 3 \\ 1 & -5 & 12 \end{matrix}$$

B - Matrix Subtraction

Subtracting two matrices and producing a new one, the two matrices should have the same dimension (M*N) where M is number of rows and N number of columns so that the user should enter the dimension first then elements of two matrices for example:

Input:

Enter the dimension of the matrix:

2

3

Enter the elements of the first matrix:

```
4   5   2
1   4   9
```

Enter the elements of the second matrix:

```
2   −3   7
4   −5   1
```

Output:

```
 2    8   −5
−3    9    8
```

C - Multiplying a Matrix by a Constant

Multiplying a Matrix by a Constant and producing a new one, user should enter the matrix dimension first, then the constant and elements of the matrix, for example:

Input:

Enter the dimension of the matrix:

3

3

Enter the constant:

4

Enter the elements of the matrix:

```
3    6    12
4    5     2
0   16     9
```

Output:

```
12   24   48
16   20    8
 0   16   36
```

D - Multiplying Matrices

Multiplying two matrices and producing a new one, the two matrices should have the same inner dimension (M*N) so if Matrix A is of dimension N*M and B is of dimension M*P, then the dimension of their product AB is N*P, for example:

Input:

Enter the dimension of the matrix A:

2

3

Enter the element of Matrix A:

$$\begin{matrix} 2 & 8 & -5 \\ -3 & 9 & 8 \end{matrix}$$

Enter the dimension of the matrix B:

3

1

Enter the element of Matrix B:

$$\begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

Output

$$\begin{matrix} 3 \\ 39 \end{matrix}$$

E - Transposing a Matrix

The matrix transpose is formed by turning all the rows of a given matrix into columns and vice-versa. The transpose of matrix A is written A^T.

Enter the dimension of the matrix B:

3

1

Enter the element of Matrix B:

$$\begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

Output

$1 \quad 2 \quad 3$

F - Determinant of Matrix:

The determinant of a matrix is a **special number** that can be calculated from a **square** matrix.

Here, you are required to calculate the 2*2, 3*3 and 4*4 determinant of a square matrix.

Enter the dimension of the matrix B:

3

3

Enter the element of Matrix B:

$$\begin{matrix} 6 & 1 & 1 \\ 4 & -2 & 5 \\ 2 & 8 & 7 \end{matrix}$$

Output

-306

Note:

- Using a procedure is must (for every operation at least one procedure)
- You must allocate memory for matrices dynamically using Heap. Refer to Section 11.3 in the textbook to know how to implement that.

Bonus:

- **Find the determinant of n*n matrix $0 \leq n \leq 10$**

Team Members: 4 – 5 members

Responsible TA: Dr. Ahmed Saeed

Reference:

- Section 11.3 heap allocation (assembly language x86 processors 7th edition)