University of Khartoum

Faculty of Engineering

Department of Electrical and Electronics Engineering

Microprocessor and Assembly language project

<Minesweeper Game>

- **Prepared By:**

- Ahmed Almamoun Awad	144012
- Maab Balla Aawedelseed	144057
- Maab Sadig Bella	144059
- Mushtaha Mutasim Mohamed	144091
- Mohamed Abdulrahman Fadul	144075

Supervisor:

- Mohanad Ahmed.

1. Objectives:

Our objective was to design our own version of the Minesweeper game played with both mouse and keyboard using assembly language assembler and a virtual machine virtual box.

Software design:

The program was a bootable bare machine program that displays the minesweeper game on the screen.

Minesweeper is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden "mines" or bombs without detonating any of them, with help from clues about the number of neighboring mines in each field.

We used Video Graphics Array (VGA) mode to display our game .The screen size was (320*200) pixels.

---

2. What did we achieve?

We were able to design the Minesweeper game that can be played using both mouse and keyboard for a limited set of maps specifies game rules.

---

3. Algorithm Description:

- **Game rules:**
  1- The grid (map) is generated after the player clicks one of the squares.
  2- The first square clicked must not include neither a mine nor a number which represents the number of mines surrounding the square from all sides.
  3- In case of open command; if the number in the square clicked is bigger than zero, then it is simply displayed.

4- If it is equal to zero a function (showAdj) is called. This function is a recursive function that checks all the elements around the square, displays all the surrounding elements bigger than zero, and calls itself if the surrounding element was zero.
5- If the number in the square clicked is equal to -1, then all the bombs are displayed except those that are flagged and the game is over.
6- In case of flag command; if the element is not shown then flagged value is converted and either flag is printed or removed.
7- When the player loses; the numbers flagged are shown as a crossed (red) bomb. The bomb flagged are shown as a green bombs. And the bomb opened is shown is a red square.
8- In the program, the player can enter 'A' if he wants to play again. Or 'G' if he gave up in any time.

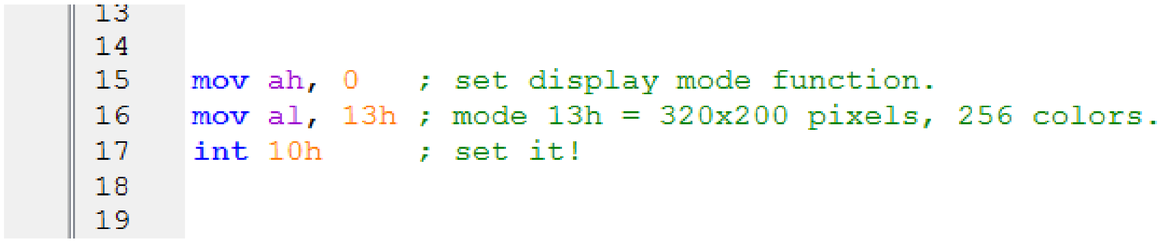
• Game algorithm:

1- Print the blank squares that represent the grid.
2- Take the location of the square that is desired to be clicked from the user (mouse/keyboard) and open it (right click/zero) or put a flag on it(left click/one).
3- If the number in the square clicked is bigger than zero, then it is simply displayed.
4- And if it is equal to zero a function (showAdj) is called. This function is a recursive function that checks all the elements around the square, displays all the surrounding elements bigger than zero, and calls itself if the surrounding element was zero.
5- If the number in the square clicked is equal to -1, then all the bombs are displayed except those that are flagged and the game is over.
6- When the player loses; the numbers flagged are shown as a crossed (red) bomb.

- **VGA algorithm:**

We used Video Mode 013h which is a standard **IBM VGA BIOS** mode number for a 256 color 320x200 resolution. It uses a 256-color palette allows access to the Video Memory as a Packed-Pixel Framebuffer. **Int 10h [Int16]** is shorthand for **BIOS** interrupt call 10 hex, the 17th interrupt vector in an x86-based computer system. BIOS calls can only be performed in Real mode or Virtual 8086 mode. The BIOS typically sets up a real mode interrupt handler at this vector that provides video services. Such services include setting the video mode, character and string output, and graphics primitives (reading and writing pixels in graphics mode).

We switched from text mode to (VGA) mode [Video graphics array] using the following code:



```
mov ah, 0    ; set display mode function.
mov al, 13h  ; mode 13h = 320x200 pixels, 256 colors.
int 10h      ; set it!
```

- After setting the mode we do the following steps:

1- load AH with the number of the desired sub function

We use AH=0Ch.

Write graphics pixel	AH=0Ch	AL = color , BH = Page Number, CX = x, DX = y

2- load other required parameters in other registers[cx=x, dx=y] [Counters bx & di].

3- Make the call.

INT 10h is fairly slow, so many programs bypass this BIOS routine and access the display hardware directly. Setting the video mode, which is done infrequently, can be accomplished by using the BIOS, while drawing graphics on the screen in a game needs to be done quickly, so direct access to video RAM is more appropriate than making a BIOS call for every pixel.

We use the VGA mode to implement all the game elements such as:

1- Coloring and partitioning the screen.
2- Drawing the game numbers [1-5].
3- Drawing the flag.
4- Drawing the shaped mines.

Using different colors and techniques, color the screen pixel by pixel to draw vertical and horizontal lines and circles.

Pseudo Code:

After initializing the map after the first input and enabling mouse and VGA mode:

```
Int game=1;

while (game){

        -   user input index(i,j) and open/flag command
        -   if(flag[i][j]==1 && k == 0){
                ▪   print();
                ▪   continue;                    //Check for another input
        -   }
        -   if(open command){
                ▪   show[i][j]= 1;
                ▪   if(map[i][j]== -1){
                        •   game = 0;                    // user lost
                        •   break;
                ▪   }
                ▪   if( map[i][j] == 0 )
                        •   showAdj(i,j);
                ▪   }else{
                ▪   if(k==1){
                        •   flag[i][j] ^= 1;
                ▪   }       }



    //// This is to know whether game is going on

        -   game=1;
        -   for(int p=0; p< m;p++)
                ▪   for(int q=0; q<n;q++)
                        •   if (map[p][q]==-1 && flag[p][q]==0){
```

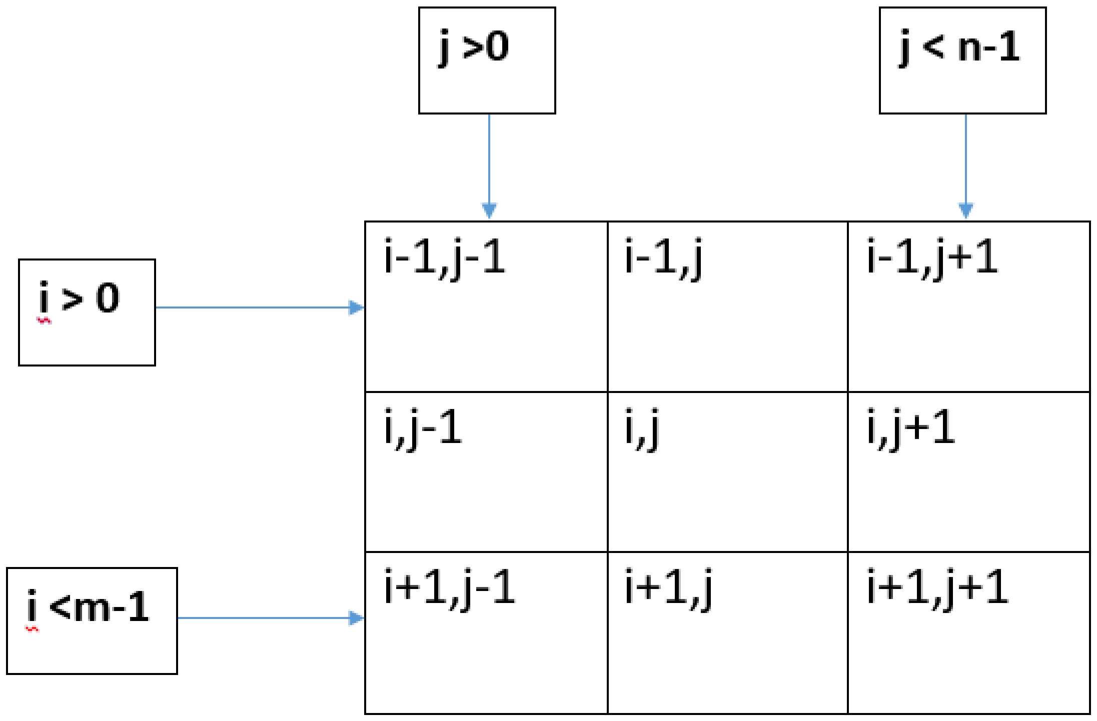
- game = 0;
- }

- if(game==1)
- break; // user wins.. taking value game=1

}

- **ShowAdj**:

As specified in game algorithm, current block with index i&j which has valid adjacent blocks available as follows:



//Repeating this test for all 8 surrounding indexes:

```
If(adjacent block is valid){
    If(block value = 0){
        showElement
        Call showAdj for this block
    }else{
        showElement        // as it can't be -1
    }
}
Call Print.
```

- **Print:**

  Traverse map array and displays shown & flagged elements in the screen

```
For i = 1 -> m
    For j = 1 -> n
        If(show[i][j] == 0)
            If(flag[i][j]==0)
                Continue;
            else
                Print flag
        else
            Print Element
```

4. What did we learn?

1- Dealing with relatively big project with many details to be considered while developing the game.
2. Practicing of breaking big idea down into relatively small parts to be handled then integrating the whole program. This includes clearly specifying inputs and outputs, considering no part will affect another one in bad order, i.e. making save needed registers values as caller's or callee's responsibility.

3. Understanding how I/O devices deals with the OS, i.e. reading input from keyboard and mouse from port 0x60 after reading the status byte in port 0x64 and differentiate whether device it request for its input.

4. Understanding basic idea of virtual machines, how to deal with secondary storages and how do bootloader work. Furthermore how to search for bugs and uses oracle virtual box debugger to some extent.

5. Learning more about 16-bits mode as we were dealing with 32-bits mode before.

6. Generating computer graphics.

7. Dealing with some interrupts and how they work, such as:

- Set video mode - INT 10h / AH = 0h.
- Set cursor position - INT 10h / AH = 2.
- Change color for a single pixel - INT 10h / AH = 0Ch.
- Get color of a single pixel - INT 10h / AH = 0Dh.
- Read disk sectors into memory - INT 13h / AH = 02h.

8. Learning about assembly language helped us to fill the gap between our understanding of high level programming language and how computer hardware performs these operations.

9. Learning a lot of assembly potentials as it gives programmer total control of processor and who may need to write assembly code for a certain application.

10. Working under some pressure in specified frame of time to accomplish project goals and setting priorities.