**Semester Project Report :**

**Project :**

Snake game in assembly language.

**Working of game :**

The game starts with a menu containing three options:

1. Start game
2. Instructions
3. Exit

Buttons used are to I – Up, J – left, K – down, L- Right, and S – select from menu.

**Start Game :**

We can use I, K, J , L to move snake Up , Down , left and right respectively. Food in form of ‘X’ will be generated randomly on the screen. Which can be picked by snake once the snake picks the food. It’s Speed and Score will increase. And if the snake collides with it’s body then game Ascii pattern for game exit will be displayed and as user press any key the console will exit.

**Instructions :**

Simply print the Instructions for Gameplay.

**Exit:**

Exits the game.

**Working of code :**

We have created different procedures for almost every-thing in code to maintain modularity of the code and make it easier to understand. When the main procedure starts working start menu is printed using the “menu” procedure Whenever the control is returned in the main procedure. It means that user has selected start- game option from menu otherwise user would have selected instructions or exit. Then we have cleared the Screen using “clear\_screen” procedure. Then score is displayed using “score\_display” procedure and After that boundary is drawn for our game using “draw\_boundary procedure”. Then another procedure “ start\_snake” is called to draw and start motion of snake in the game.

**Working of procedures :**

Although working of Each and every procedure is already been described through the comments in the code, but a brief description of working of procedure is also described here.

**start\_snake :**

It first calls the generate\_food procedure to randomly generate the food within the boundaries of game, and then store the current position coordinates of snake in temp variable. And then uses Readkey if there is any input then compares with I, J ,K , L. If input = “L” then increment the x- coordinate of snake so . If input = j then decrement x coordinate. If input = ‘K’ then increment the y- coordinate of snake. If input = ‘I’ then decrement the y- coordinate of snake. But If the user is inputs the previous input again then nothing is changed within the coordinates of snake. After the input the control jumps to update label, where the previous position of snake is moved to ‘al’ and ‘ah’. And then boundary collision is checked using check\_coundary\_collisoin (if there is a boundary collision then snake is wrapped around accordingly in the check\_coundary\_collisoin procedure), After that the snake is redrawn using draw\_snake and update\_snake procedure. After that check\_food procedure is called (to check whether the previously generated food is eaten or not if the food is eaten then new food is generated). At the end check\_snake\_collision is called which will mov 1 to death variable if there is collision. Now in start\_snake procedure if death variable is 1, we have called gameover\_display procedure (to display the “game end” ascii pattern and also end the game in gameover\_display procedure). If death is not 1 we simply move to safe label give some delay to whole console and loop again from where we take input from the user.

**Menu :**

It is quite simple procedure. At the start it moves cursor to (50,10) position and it print ‘Start game’ and prints <-- , then move cursor to (50,11) then prints ‘Instructions’ and then move cursor to (50,12) then prints “exit”. Now there are three labels in the code namely (case 1, case2 and case3). when user input ‘I’ it goes to another case3 where it print in the same way but now arrow is printed adjacent to ‘Instruction’ option. If input were ‘k’ then it would jump to Case3 where arrow would be printed adjacent to exit option. Similar concept for inputs is used in any of the other 2 labels. then after analyzing the input the control jumps to particular label out of 3 labels. And if the input is S inside any of the labels then procedure required corresponding to the option with arrow in the label is called. (This procedure is clearly explained in code using comments)

**display\_instructions :**

This procedure simply clears the screen using clear\_screen procedure and then draw the boundaries, then move cursor to (32,10) prints the first instruction , then move cursor to (32,11) prints the second instruction, and then move cursor to (32,12) and then print the third instruction.

**gameover\_display :**

This procedure simply moves the cursor at positions within the boundaries of game like in display\_instructions and print all the corresponding lines of ascii pattern.

**check\_boundary\_collision:**

This procedure moves the x and y coordinates of snake in al, ah and the coordinates of boundaries all four boundaries one by one in ‘bl’, then for left wall collision checking it moves left\_wall coordinates to the bl, and compares bl with al, if both are same. It jumps to label for wrapping around. For wrapping from right it changes the x coordinates of snake with the coordinates of right wall. We have used similar concept for other checking collision with other walls as well.(It is further explained in code using comments)

**check\_snake\_collision :**

This procedure moves the coordinates of snake’s head in al and ah, and a loop is used which runs for the length of snack, and coordinates of other segments of snack are moved to dl and dh one by one in turns with loop, and then coordinates of snake’s head and other segments are compared if any of the coordinates are matched then 1 is move death variable to show collision.

**draw\_boundary :**

This procedure first moves cursor to upper left corner of boundary i.e. (8,2) and then an loop is run for 100 times to print character corresponding to (219) to draw upper wall. Then the cursor is again shifted to (8,2), Now another loop runs for 20 times to draw left and right boundary. In this loop first y coordinate of cursor is incremented (using INC dh) and wall charactered is printed and then x coordinated of cursor is incremented by 99, and wall character is printed and then cursor is moved to (left\_wall i.e. 8, dh). After the loop is completed Left\_wall i.e. 8 is moved to dl. Here cursor points to the left-lower corner of boundary wall element is again printed 100 times using another loop.

**draw\_snake**

**update\_snake**

**clear\_screen :**

We have defined clear\_screen procedure instead of using clsr procedure because clsr clears only some portion of screen, Where as in our procedure we have defined a byte array of 110 (total x coordinates used in program,) and initialized it with “ “, and then used a loop with runs 24 times (total number y-coordinate used in program), and in loop we printed this array 24 times followed by an new line. To clear the whole screen.

**score\_display :**

This procedure moves the cursor to the ( 16 , 0), where 16 = (left\_wall + 8 [other characters of “score : ” string]), and there simply prints the score.

**generate\_food :**

This procedure simply generates the food inside the boundaries of program. In y coordinate we need food in y coordinate from 4 to 19. So we use a randomize procedure to generate a number between 4 and 19 using eax = 15 and then adding 4 and move it to ‘dh’, and for x coordinate we need food from 9 to 107, so using eax = 90 and adding 14 to eax, and move it to ‘dl’, then set cursor to given dl and dh, and print the food there in form of ‘X’ and also moves these coordinates to temp variables.

**check\_food :**

We have created this procedure to check whether the food is being eaten by snack or not, to do so we compare the coordinates of snack with the coordinates of food stored in temp variable in generate\_food procedure, if both the coordinates are same we say the food is eaten by snack and increment the score and speed. And also update the score, and then generate the food again using generate\_food procedure.