National University
of computer and emerging sciences

Final Project Report

EL2003-Computer Organization and Assembly Language - Lab

Semester Project

Mirza Humayun Masood(I22-1749)

Saif-Ur-Rehman (I22-1697)

Mustafa Zain (I22-4948)

Section : CY-(A)

Submitted to: Sir Suleman Saboor

Department of Cyber Security BS(CY)

FAST-NUCES Islamabad

In this Project of making Candy crush. We divided the code into three files, one which contains the macros, cuz we followed a youtube video in which the teacher said to always make a separate file for macros and classes.

So first we would be discussing the macros section of the Code.

Macros

clsscreen macro: This macro is defined to reset the screen. It calls a Function named mkscreen.

pushreg macro: This macro is used to push registers onto the stack. It pushes the values of di, si, ax, bx, cx, and dx registers onto the stack.

popreg macro: This macro is used to pop registers from the stack. It pops the values from the stack in reverse order compared to the pushreg macro, restoring the values of di, si, ax, bx, cx, and dx registers.

calsleep macro: This macro is used to implement a sleep function. It takes a parameter msec (milliseconds). Inside the macro, it pushes all registers onto the stack, initializes a counter (cx) to 0, then enters a loop where it compares the counter with msec. Inside the loop, it sets dx to 1, pushes dx onto the stack, calls a function named sleep, and increments the counter. This loop continues until the counter (cx) reaches the specified number of milliseconds. After the loop, it pops all registers back from the stack.

setcurs macro: This macro sets the cursor position on the screen to the specified coordinates (x, y). It pushes all registers onto the stack, sets the values of ah 02 and dl, dh (x and y coordinates) accordingly, then calls the interrupt 10h to set the cursor position. Finally, it pops all registers from the stack.

writefilestr macro: This macro writes a string to a file. It pushes all registers onto the stack, moves the offset of the string into dx, calculates the length of the string using a function StrLen, sets ah to 40h (write to file), and then makes an interrupt call (int 21h). Finally, it pops all registers from the stack.

coutnum macro: This macro is used to output a number. It pushes dx (which holds the number) onto the stack, and then calls a function named output which displays the number by dividing it with 10 and taking a reminder.

coutstr macro: This macro is used to output a string to the console. It moves the offset of the string into dx, sets ah to 09 (print string), and then makes an interrupt call (int 21h).

It was all the macros. Now we would discuss the structures we used in Code and define some Constants in assembly. I made a separate file for this as well, cuz we should keep these things in a separate file as well.

Constants

maxMoves: Represents the maximum number of moves allowed in the game, set to 15.

lvl1thresh, lvl2thresh, lvl3thresh: Thresholds for different levels of the game. These values likely represent score thresholds required to progress to the next level.

lvl1thresh: 50

lvl2thresh: 100

lvl3thresh: 150

tarueeeeee, falaseeeeeeee, Zeroo: These constants represent boolean values.

tarueeeeee: 1

falaseeeeeeee: 0

Zeroo: 0

bswidth, bsheight: Width and height of the game board in pixels, set to 50 and 20 respectively.

candywid, candyhei: Width and height of the candies in pixels, set to 30 and 12 respectively.

boardWidth, boardHeight: Width and height of the game board in terms of squares, set to 7x7.

candy_type equ <word>: This likely defines a custom data type for representing different types of candies.

candy_type_NULL, candy_type_1, ..., candy_type_Bomb: Constants representing different types of candies, ranging from 0 to 6.

candy_type_First, candy_type_Last: Constants defining the range of candy types, from candy_type_1 to candy_type_Bomb.

randSleepTime: Time interval for random sleep operations, calculated as the maximum candy type multiplied by 10.

usernameSize: Size of the username field, set to 20.

Structures

coord struct: Defines a structure for storing coordinates.

- **x**: Represents the x-coordinate.
- **y**: Represents the y-coordinate.

candy struct: Defines a structure for representing candies on the game board.

- **topLeftCoord**: Coordinates of the top-left corner of the candy.
- **shapeType**: Type of candy, using the candy_type enum.
- **rowNum, colNum**: Row and column numbers of the candy on the game board.
- **InCombo**: Indicates whether the candy is part of a combo (true/false).

boardsqr struct: Defines a structure representing a square on the game board.

- **topLeftCoord**: Coordinates of the top-left corner of the square.
- **currentCandy**: Information about the candy occupying the square.
- **IsSelected**: Indicates whether the square is currently selected (true/false).

Now let's discuss the main Code file.

Variables

Game State and Progression:

- **currlvl**: Current level number.
- **islvlfinish**: Indicator for level completion.
- **islvlpass**: Indicator for level pass.
- **movesLeft**: Number of moves left in the game.
- **rand_seed**: Seed for random number generation.

Game Board Representation:

- **boardMatrix**: 2D array representing the game board.
- **boardTopLeftCoord**: Coordinates of the top-left corner of the game board.
- **selectedCandy**: Index or identifier of the currently selected candy.

Player Data and Input:

- **username**: Player's name.
- **currscore**: Current score.
- **lvl1score, lvl2score, lvl3score**: Scores for each level.
- **highScore**: High score.

- **currscoremsg**: Message indicating the current score.
- **movesLeftMsg**: Message indicating the number of moves left.
- **enterNameMsg**: Message prompting the player to enter their name.

Game Messages and Prompts:

- **lvl1msg**, **lvl2msg, lvl3msg**: Level names.
- **lvl1pass**, **lvl1fail, lvl2pass, lvl2fail, lvl3pass, lvl3fail**: Level completion/failure messages.
- **reqscoremsg**: Message indicating the required score to pass the level.
- **crusssss, explooooo, candydropppppp**: In-game event messages.
- **gameTitleMsg**: Title of the game.
- **gameRulesMsg**: Title of the game rules section.
- **rule1Msg to rule5Msg**: Different rules of the game.
- **pressEnterMsg**: Prompt to press Enter to start the game.

File and File-related Messages:

- **scoreStr**: String for storing score data.
- **scoreFile**: Filename for storing score data.
- **lvl1filemsg, lvl2filemsg, lvl3filemsg, hsfilemsg**: Messages indicating score file data.
- **endl**: End-of-line character.

Miscellaneous:

- **emptyString**: Array used for storing empty strings.

Procedures

Main:

- Setting up Data Segment:
    - mov ax, @data: Loads the segment address of the data segment into the AX register. @data is a special directive that represents the starting address of the data segment.
    - mov ds, ax: Moves the value in AX (which now holds the address of the data segment) into the DS (Data Segment) register. This sets up DS to point to the data segment, which is where variables and constants are typically stored.
- Saving Registers and Seeding Random Number Generator:

    - pushreg: This likely pushes the values of certain registers onto the stack. This is a common practice to preserve the state of registers before calling other procedures or functions.
    - call srand: Calls a function or procedure named srand, which is probably responsible for seeding the random number generator. This ensures that the sequence of random numbers generated by the program is different each time the program runs. The specific implementation of srand would be defined elsewhere in the code.
- Game Loop:
    - call GameLoop: Calls a function or procedure named GameLoop. This is where the main logic of the game is executed. It likely contains the core game loop that repeatedly checks for user input, updates game state, and renders the game.
- Exiting the Program:
    - call exit: Calls a function or procedure named exit. This is used to gracefully terminate the program.
- Restoring Registers and Returning:
    - popreg: This likely pops the previously saved register values from the stack, restoring their original state.
    - ret: Returns control to the calling function. Since this is the main procedure, returning from it effectively ends the program execution.

Startmouse:

- Saving Registers:
    - pushreg: This likely pushes the values of certain registers onto the stack. This is a common practice to preserve the state of registers before making changes or calling other procedures or functions.
- Mouse Initialization:
    - mov ax, 0: Moves the value 0 into the AX register.
    - int 033h: Generates a software interrupt 033h (hexadecimal), which is typically used to interact with mouse functions in DOS or BIOS environments. This interrupt is used to initialize the mouse.
- Restoring Registers:
    - popreg: This likely pops the previously saved register values from the stack, restoring their original state. This is done to ensure that the state of registers is not modified outside of this procedure.
- Returning:
    - ret: Returns control to the calling function or procedure. This indicates that the mouse initialization process is complete, and the program can continue executing.

Mkscreen:

- Saving Registers:
    - pushreg: This instruction is likely pushing the values of certain registers onto the stack. It's a common practice to preserve the state of registers before making changes or calling other procedures or functions.
- Setting Video Mode:
    - mov ah, 00h: Moves the value 00h (0) into the AH register. This value typically indicates that an "interrupt" or "function" call is about to be made.
    - mov al, 0eh: Moves the value 0eh (14 in decimal) into the AL register. This value represents the video mode 640 x 200 pixels with 16 colors in many BIOS interrupt services for video functions.
    - int 010h: Executes interrupt 10h with the specified function and parameters. In DOS environments, this interrupt is used to perform various video-related operations, including setting the video mode.
- Restoring Registers:
    - popreg: This instruction is likely popping the previously saved register values from the stack, restoring their original state. This is done to ensure that the state of registers is not modified outside of this procedure.
- Returning:
    - ret: Returns control to the calling function or procedure. This indicates that the screen has been set up according to the specified video mode, and the program can continue executing.

Startgame:

- Saving Registers:
    - pushreg: This instruction is likely pushing the values of certain registers onto the stack. It's a common practice to preserve the state of registers before making changes or calling other procedures or functions.
- Setting Up Screen:
    - call mkscreen: Calls the mkscreen procedure, which sets up the screen for graphics display, likely configuring it to a specific mode suitable for the game.
- Starting Mouse:
    - call startmouse: Calls the startmouse procedure, which initializes the mouse, enabling its functionality for user interaction with the game.
- Displaying Rules:
    - call displayrule: Calls a procedure (displayrule) to display the rules of the game. This likely presents information to the player about how to play the game effectively.
- Clearing Screen:

    - clsscreen: Clears the screen. This could be a custom procedure designed to clear the screen of any previous content, preparing it for the game display.
- Setting Up Board:
    - call makeboard: Calls a procedure (makeboard) to initialize and set up the game board. This likely involves creating the game board grid and populating it with initial game elements.
- Displaying Messages:
    - call displaymsgs: Calls a procedure (displaymsgs) to display game-related messages or prompts on the screen. These messages might include instructions for the player or information about the current game state.
- Restoring Registers:
    - popreg: This instruction is likely popping the previously saved register values from the stack, restoring their original state. This is done to ensure that the state of registers is not modified outside of this procedure.
- Returning:
    - ret: Returns control to the calling function or procedure. This indicates that the setup for the game has been completed, and the game is ready to be played.

Makeboard:

- Saving Registers:
    - pushreg: Saves the state of registers on the stack to preserve them during the execution of the procedure.
- Initializing Variables:
    - mov si, offset boardMatrix: Initializes the source index (si) to point to the start of the boardMatrix array, which represents the game board.
    - mov cx, boardTopLeftCoord.x and mov dx, boardTopLeftCoord.y: Initialize the coordinates (cx for x-coordinate and dx for y-coordinate) of the top-left corner of the board, using the values stored in the boardTopLeftCoord structure.
- Iterating Over Board Squares:
    - Uses nested loops to iterate over each square in the game board:
    - Outer loop (bx) iterates over the rows (boardHeight).
    - Inner loop (ax) iterates over the columns (boardWidth).
- Setting Up Square Properties:
    - For each square, sets up its properties:
    - Sets the coordinates of the square's top-left corner.
    - Calculates and sets the coordinates of the candy inside the square, positioning it at the center of the square.
    - Sets the row and column numbers of the current candy.

    - Calls drawboardsq procedure to draw the square on the screen.
- Adjusting Coordinates:
    - After iterating over a row, adjusts the cx (x-coordinate) back to the start of the row and increments dx (y-coordinate) to move to the next row.
- Hiding Mouse:
    - Calls hidemouse to hide the mouse cursor while setting up the board.
- Dropping Candies:
    - Allocates space on the stack to pass parameters to the dropcandy procedure.
    - Calls dropcandy to initiate the process of dropping candies onto the board.
- Drawing Crushes:
    - Calls drawcrush to draw any initial crushes on the board.
- Balancing Board:
    - Calls balancebord to ensure the initial board state meets the game's requirements.
- Showing Mouse:
    - Calls showmouse to make the mouse cursor visible again after the board setup is complete.
- Restoring Registers:
    - popreg: Restores the state of registers from the stack.
- Return:
    - ret: Returns control to the calling function or procedure.

GameLoop:

- Initialization:
    - Calls the startgame procedure to initialize the game environment, including setting up the screen, mouse, board, and displaying initial messages.
    - Calls showmouse to make the mouse cursor visible.
- Main Loop:
    - Enters a continuous loop that runs as long as tarueeeeee (presumably true) is true.
- Game State Check:
    - Checks if the current level is finished (islvlfinish). If it is:
        - Calls DisplayResults to show the results of the level.
        - Checks if the level is not passed (islvlpass is false) or if the current level is 3:
            - If true, calls exit to exit the game.
            - If false:
                - Increments the current level (currlvl) by one.
                - Calls calsleep to pause the execution for 1000 milliseconds.
                - Clears the screen (clsscreen).
                - Calls calsleep again to pause for 500 milliseconds.
                - Calls resetgame to reset the game state.

    - Calls makeboard to recreate the game board.
    - Calls displaymsgs to display messages related to the game state.
    - Calls showmouse to make the mouse cursor visible.
- Mouse Handling:
    - Within the loop, calls mousehandling to handle mouse input and interactions.
- End of Loop:
    - Ends the loop (endw) when the game state no longer meets the condition.
- Restoration of Registers:
    - popreg: Restores the state of registers from the stack.
- Return:
    - ret: Returns control to the calling function or procedure.

Sleep:

- Parameter Passing:
    - Upon entry, the base pointer bp is pushed onto the stack to save its previous value.
    - The base pointer is then moved to the stack pointer sp, establishing a new stack frame.
    - The registers are saved using pushreg, preserving their state.
- Delay Calculation:
    - The value passed to the sleep procedure, located at [bp + 4], is assumed to represent the duration of the delay in milliseconds.
    - This value is multiplied by 1000 (converted to microseconds) to prepare it for the interrupt call.
    - The result is split into the cx and dx registers to accommodate the 16-bit division required by the int 15h interrupt.
- Interrupt Call:
    - The ah register is set to 86h, indicating an interrupt call for "Wait/Release Time" functionality.
    - The int 15h interrupt is invoked, which causes the program execution to pause for the specified duration.
- Register Restoration:
    - After the delay, the registers are restored to their previous state using popreg.
- Stack Cleanup and Return:
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

Drawblankcandy:

- Parameter Passing:
    - The base pointer bp is pushed onto the stack to save its previous value.
    - Then, the base pointer is moved to the stack pointer sp, establishing a new stack frame.
    - The registers are saved using pushreg, preserving their state.
- Preparing for Drawing:
    - The parameters passed to the procedure are accessed from the stack. The pointer to the Candy structure is located at [bp + 4], indicating the candy to be drawn.
    - The topLeftCoord of the candy is retrieved to determine the starting position for drawing.
- Drawing Process:
    - The procedure uses BIOS interrupt int 10h with function 0Ch to plot pixels directly to the screen.
    - It iterates through each row and column of the candy, starting from its top-left corner.
    - Within the nested loops:
        - The bx register is used to iterate through the width of the candy.
        - Inside the inner loop, pixels are plotted one by one horizontally.
        - After completing a row, the starting position is reset to the beginning of the row, and dx is incremented to move to the next row.
- Register Restoration:
    - After completing the drawing process, the registers are restored to their previous state using popreg.
- Stack Cleanup and Return:
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

Drawcandy1-5:

All functions have there each colors

- Parameter Passing:
    - The base pointer bp is pushed onto the stack to save its previous value.
    - Then, the base pointer is moved to the stack pointer sp, establishing a new stack frame.
    - The registers are saved using pushreg, preserving their state.
- Preparing for Drawing:
    - The parameters passed to the procedure are accessed from the stack. The pointer to the Candy structure is located at [bp + 4], indicating the candy to be drawn.

    - The topLeftCoord of the candy is retrieved to determine the starting position for drawing.
- Drawing Process:
    - The procedure uses BIOS interrupt int 10h with function 0Ch to plot pixels directly to the screen.
    - It iterates through each row and column of the candy, starting from its top-left corner.
    - Within the nested loops:
        - The bx register is used to iterate through the width of the candy.
        - Inside the inner loop, pixels are plotted one by one horizontally.
        - After completing a row, the starting position is reset to the beginning of the row, and dx is incremented to move to the next row.
- Register Restoration:
    - After completing the drawing process, the registers are restored to their previous state using popreg.
- Stack Cleanup and Return:
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

Drawbomb:

- Parameter Passing:
    - The base pointer bp is pushed onto the stack to save its previous value.
    - The base pointer is moved to the stack pointer sp, creating a new stack frame.
    - The registers are saved using pushreg to preserve their state.
- Preparing for Drawing:
    - Space is reserved on the stack to store temporary variables using sub sp, 4.
    - The parameters passed to the procedure are accessed from the stack. The pointer to the candy structure is located at [bp + 4], indicating the bomb candy to be drawn.
    - The topLeftCoord of the bomb candy is retrieved to determine the starting position for drawing. The x-coordinate is stored in [bp - 2], and half the candy width is added to calculate the center. The left and right boundaries of the bomb's explosion are stored in [bp - 2] and [bp - 4], respectively.
- Drawing Process:
    - The procedure iterates through each row and column of the bomb candy.
    - The outer loop iterates over the height of the candy, decrementing bx until it reaches 0.
    - Inside the nested loops:

        - The inner loop iterates over the width of the candy, decrementing bx until it reaches 0.
        - Within the inner loop, the procedure checks if the current x-coordinate is within the boundaries of the bomb explosion. If so, it plots a pixel using BIOS interrupt int 10h.
        - The x-coordinate (cx) is incremented after each pixel is plotted, moving horizontally across the screen.
- Register Restoration:
    - After completing the drawing process, the registers are restored to their previous state using popreg.
- Stack Cleanup and Return:
    - The space reserved on the stack is released using add sp, 4.
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

DrawCandy:

- Parameter Passing:
    - The base pointer bp is pushed onto the stack to save its previous value.
    - The base pointer is moved to the stack pointer sp, creating a new stack frame.
    - The registers are saved using pushreg to preserve their state.
- Accessing Candy Information:
    - The pointer to the Candy structure is accessed from the stack at [bp + 4].
    - The shape type of the candy is retrieved from the Candy structure and stored in the ax register.
- Drawing the Candy:
    - Based on the shape type of the candy, a specific drawing subroutine is called.
    - If the shape type is candy_type_1, the subroutine drawcandy1 is called to draw the candy.
    - Similarly, for other shape types (candy_type_2 to candy_type_Bomb), corresponding drawing subroutines (drawcandy2 to drawbomb) are called.
- Drawing Subroutines:
    - Each drawing subroutine is responsible for drawing a specific type of candy shape on the screen.
- Register and Stack Cleanup:
    - After drawing the candy, the registers are restored to their previous state using popreg.
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

Drawboardsq:

- Parameter Passing:
    - The base pointer bp is pushed onto the stack to save its previous value.
    - The base pointer is moved to the stack pointer sp, creating a new stack frame.
    - The registers are saved using pushreg to preserve their state.
- Accessing Board Square Information:
    - The pointer to the boardsqr structure is accessed from the stack at [bp + 4].
    - Information about the square, such as whether it's selected (IsSelected) and its top-left coordinates, is retrieved from the boardsqr structure.
- Drawing the Square:
    - The color attribute (al) is determined based on whether the square is selected (IsSelected). If it is selected, the color attribute is set to 06h (yellow); otherwise, it's set to 0fh (white).
    - Using interrupt 10h function 0ch, the square is drawn on the screen pixel by pixel. This function sets the color attribute of the pixel at the specified coordinates.
- Drawing the Border:
    - The border of the square is drawn by moving to the right (cx incremented) and then down (dx incremented), drawing each line of the border.
    - After completing the bottom and right borders, the position is adjusted to draw the left and top borders.
- Register and Stack Cleanup:
    - After drawing the square and its borders, the registers are restored to their previous state using popreg.
    - The base pointer bp is restored by popping its value from the stack.
    - The procedure returns control to the calling function or procedure, taking into account the 2 bytes of parameters passed to it.

Srand:

- Parameter Passing:
    - The procedure doesn't take any parameters.
- Saving Register State:
    - The pushreg macro is used to save the state of the registers.
- Seeding the Random Number Generator:
    - Interrupt 01Ah (INT 1A - Time of Day) is invoked with function 00h to get the current system time.
    - The random seed is set to the value stored in the dx register after the interrupt call. This value is part of the current time information obtained from the system.
- Restoring Register State:

    - The popreg macro is used to restore the state of the registers to their previous values.
- Return:
    - The procedure returns control to the caller.

Rand:

- Parameter Passing:
    - The procedure expects three parameters on the stack:
    - The first parameter ([bp + 4]) represents the upper bound of the range.
    - The second parameter ([bp + 6]) represents the lower bound of the range.
    - The third parameter ([bp + 8]) is where the generated random number will be stored.
- Saving Register State:
    - The procedure begins by pushing the base pointer (bp) onto the stack and then moving bp to sp, effectively creating a new stack frame.
    - The pushreg macro is used to save the state of the registers.
- Generating Random Number:
    - The current value of rand_seed (initially set by srand) is multiplied by a constant (25173) and then increased by another constant (13849).
    - The result is stored back into rand_seed.
    - The value of rand_seed is right-shifted 1 bit (shr ax, 1) five times to enhance randomness.
    - The difference between the upper and lower bounds of the range is calculated and stored in bx.
    - The pseudo-random number is obtained by dividing rand_seed by the range (the difference between upper and lower bounds) and adding the lower bound.
    - The result is stored in [bp + 8], which represents the output parameter.
- Restoring Register State:
    - The popreg macro is used to restore the state of the registers to their previous values.
- Restoring Stack Frame:
    - The base pointer bp is restored to its original value by popping it from the stack (pop bp).
    - The procedure returns by executing ret 4, which removes the parameters from the stack and returns control to the caller.

Balancebord:

- Saving Register State:
    - The procedure starts by saving the state of the registers using the pushreg macro.
- Repeat Loop:

    - The procedure enters a loop with .repeat and continues until there are no more candy combinations found.
- Find All Combos:
    - Within each iteration of the loop, the procedure calls the findallcombos subroutine to search for all possible candy combinations on the board. This subroutine likely identifies matches of three or more candies of the same type.
    - The result of the subroutine is stored in the ax register.
- Checking for Combos:
    - If the result stored in ax is not zero (indicating that candy combinations were found), the procedure proceeds to crush those candies by calling the drawcrush subroutine.
- Loop Termination:
    - The loop continues until no more candy combinations are found (ax == 0).
- Restoring Register State:
    - After the loop completes, the procedure restores the state of the registers using the popreg macro.
- Return:
    - Finally, the procedure returns control to the caller.

Drawcrush:

- Saving Register State:
    - The procedure starts by saving the state of the registers using the pushreg macro.
- Pop Combos:
    - The procedure calls the popcombos subroutine to remove all candy combinations from the game board. This likely involves updating the state of the affected candies to reflect their removal.
- Displaying Crushing Message:
    - It sets the cursor position to row 55 and column 17 using the setcurs subroutine.
    - Then, it prints the crushing message "CRUSHING!!!" using the coutstr subroutine to inform the player about the crushing effect.
- Pushing Down Candies:
    - The procedure calls the pushdown subroutine to adjust the position of candies above the removed ones, simulating the effect of candies falling down to fill the empty spaces.
- Dropping Candies:
    - It calls the dropcandy subroutine to drop new candies into the empty spaces created by the removed combinations.
- Level Block Check:

    - The procedure calls the lvlblock subroutine to check if any blocks have been completed. This likely involves checking if the player has reached the required score to pass a level or if there are any other level-specific conditions.
- Restoring Register State:
    - After completing its tasks, the procedure restores the state of the registers using the popreg macro.
- Return:
    - Finally, the procedure returns control to the caller.

Mousehandling:

- Saving Register State:
    - The procedure starts by saving the state of the registers using the pushreg macro.
- Checking Mouse Input:
    - It uses interrupt 0x33 with function 0x03 to check the status of the mouse.
    - If the left mouse button is pressed (bx & 001b), it calls the leftmousepressed subroutine to handle the action associated with a left mouse click.
    - If the right mouse button is pressed (bx & 010b), it checks if a candy is currently selected (selectedCandy != 0). If so:
        - It clears the selection by updating the IsSelected flag of the corresponding candy to falaseeeeeeee.
        - Calls hidemouse to hide the mouse cursor.
        - Redraws the unselected candy on the game board.
        - Calls showmouse to display the mouse cursor again.
        - Resets the selectedCandy variable to Zeroo.
    - If the middle mouse button is pressed (bx & 100b), it exits the game by calling the exit function.
- Restoring Register State:
    - After handling the mouse input, the procedure restores the state of the registers using the popreg macro.
- Return:
    - Finally, the procedure returns control to the caller.

Showmouse:

- Save Register State:
    - The procedure starts by saving the state of the registers using the pushreg macro.
- Display Mouse Cursor:
    - It sets the AX register to 1, indicating the "show cursor" function.
    - Then, it invokes interrupt 0x33 to perform the action, which displays the mouse cursor on the screen.
- Restore Register State:

    - After displaying the mouse cursor, the procedure restores the state of the registers using the popreg macro.
- Return:
    - Finally, it returns control to the caller.

Hidemouse:

- Save Register State:
    - The procedure begins by saving the state of the registers using the pushreg macro.
- Hide Mouse Cursor:
    - It sets the AX register to 2, indicating the "hide cursor" function.
    - Then, it invokes interrupt 0x33 to perform the action, which hides the mouse cursor from the screen.
- Restore Register State:
    - After hiding the mouse cursor, the procedure restores the state of the registers using the popreg macro.
- Return:
    - Finally, it returns control to the caller.

Leftmousepressed:

- Save Register State and Retrieve Parameters:
    - It begins by saving the base pointer and the state of registers using the push bp and pushreg macros, respectively.
    - Then, it retrieves parameters from the stack.
- Selecting Candy:
    - It calls the selectingcandy subroutine to handle the selection of candies based on mouse input.
    - If a candy is selected, it updates the selectedCandy variable and marks the corresponding candy on the game board as selected.
- Candy Swap:
    - If two candies are selected (ax == true), it swaps their positions on the game board.
    - It checks for possible combinations after the swap and updates the display accordingly.
    - If combinations are found, it updates the display to show the effects of crushing candies and balancing the board.
    - It also decrements the number of moves left and checks if the level is finished or failed based on the remaining moves and score.
- Update Display and Return:

    - After processing the left mouse button press, it updates the display to reflect any changes and returns control to the caller.
    - It also includes a brief sleep to ensure smooth gameplay.
- Restore Register State and Return:
    - Finally, it restores the register state and base pointer using the popreg and pop bp macros, respectively, and returns control to the caller.

Selectingcandy:

- Initialize Variables:
    - It initializes the base pointer and stack pointer using the push bp and mov bp, sp instructions, respectively.
    - The pushreg macro saves the register state.
- Iterate Through Game Board:
    - Using a loop, it iterates through each square on the game board.
    - For each square, it checks if the mouse coordinates fall within the boundaries of the square.
- Check Mouse Coordinates:
    - It compares the mouse coordinates (bp + 6, bp + 4) with the coordinates of the current square's top-left corner.
    - If the mouse coordinates are within the boundaries of the square and there is a candy present, it selects that candy.
- Return Selected Candy:
    - If a candy is found, it returns the address of that candy ([bp + 8]).
    - If no candy is found at the mouse coordinates, it returns Zeroo to indicate no candy selection.
- Restore Register State and Return:
    - Finally, it restores the register state and base pointer using the popreg and pop bp instructions, respectively, and returns control to the caller.

Candyswap:

- Initialization:
    - It sets the flag [bp + 8] to falaseeeeeeee, indicating that no successful swap has been made yet.
    - It retrieves the addresses of the candies to be swapped from the stack and stores them in di and si.
- Check Validity of Swap:
    - It checks if both di and si are not null (Zeroo) and if they are not the same candy. If these conditions are met, it proceeds with the swap.
- Check Candy Types:
    - It retrieves the candy types of the candies at addresses di and si.

    - If the candy types are different, indicating a valid swap, it proceeds to check if the candies are adjacent using the chkcandyneighbour procedure.
- Perform Swap:
    - If the candies are adjacent, it swaps their positions by exchanging their candy types.
    - It then hides the mouse cursor, redraws the candies at their new positions on the game board, and shows the mouse cursor again.
    - It sets the flag selectedCandy to Zeroo to indicate that no candy is currently selected.
    - Finally, it sets the flag [bp + 8] to tarueeeeee to indicate that a successful swap has been made.
- Return:
    - It restores the register state and base pointer and returns control to the caller, passing [bp + 8] to indicate the success of the swap.

Chkcandyneighbour:

- Initialization:
    - It retrieves the addresses of the two candies to be checked (di and si) from the stack.
- Check for Null Pointers:
    - It checks if either di or si is a null pointer (Zeroo). If either of them is null, it sets the flag [bp + 8] to falaseeeeeeee and returns.
- Calculate Horizontal and Vertical Differences:
    - It calculates the horizontal difference between the x-coordinates of the candies (ax) and the vertical difference between the y-coordinates of the candies (cx).
    - It subtracts the x-coordinates of the candies and stores the result in ax, and does the same for the y-coordinates, storing the result in cx.
- Check Neighbourhood Relationship:
    - It checks if the absolute value of the horizontal difference (ax) is equal to the width of a board square (bswidth), or if it is zero, indicating that the candies are adjacent horizontally.
    - Similarly, it checks if the absolute value of the vertical difference (cx) is equal to the height of a board square (bsheight), or if it is zero, indicating that the candies are adjacent vertically.
    - If either of these conditions is met, it sets the flag [bp + 8] to tarueeeeee, indicating that the candies are adjacent.
- Return:
    - It restores the register state and base pointer and returns control to the caller, passing [bp + 8] to indicate the result of the check.

Rowcomboo:

- Initialization:
    - It pushes the base pointer onto the stack and sets up the stack frame.
    - It initializes the combo count to 1 (indicating the current candy in the row being checked) and retrieves the address of the candy to be checked (si) from the stack.
    - It retrieves the shape type of the candy (ax) to be used as a reference for comparison.
- Check to the Right:
    - It starts by checking candies to the right of the current candy (si). It moves to the next column (colNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy to the right. This process continues until a non-matching shape type is encountered or the end of the row (boardWidth) is reached.
- Check to the Left:
    - If the combo count is greater than 1 (indicating that at least two matching candies were found to the right), the procedure proceeds to check candies to the left of the current candy (si). It moves to the previous column (colNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy to the left. This process continues until a non-matching shape type is encountered or the beginning of the row (column index 0) is reached.
- Mark Combos:
    - If the combo count is greater than 2 (indicating that at least three matching candies were found either to the right or to the left), it marks all the candies in the combo as part of the combo. It sets the InCombo flag of each candy to tarueeeeee, indicating that it is part of a combo.
    - If the combo count is not greater than 2, it resets the combo count to 0.
- Cleanup and Return:
    - It restores the register state and base pointer, then returns to the caller, passing the combo count ([bp + 6]) as the result of the row combo detection process.

Colcomboo:

- Initialization:
    - It pushes the base pointer onto the stack and sets up the stack frame.
    - It initializes the combo count to 1 (indicating the current candy in the column being checked) and retrieves the address of the candy to be checked (si) from the stack.

    - It retrieves the shape type of the candy (ax) to be used as a reference for comparison.
- Check Downwards:
    - It starts by checking candies downwards from the current candy (si). It moves to the next row (rowNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy below. This process continues until a non-matching shape type is encountered or the bottom of the column (boardHeight) is reached.
- Check Upwards:
    - If the combo count is greater than 1 (indicating that at least two matching candies were found downwards), the procedure proceeds to check candies upwards from the current candy (si). It moves to the previous row (rowNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy above. This process continues until a non-matching shape type is encountered or the top of the column (row index 0) is reached.
- Mark Combos:
    - If the combo count is greater than 2 (indicating that at least three matching candies were found either downwards or upwards), it marks all the candies in the combo as part of the combo. It sets the InCombo flag of each candy to tarueeeeee, indicating that it is part of a combo.
    - If the combo count is not greater than 2, it resets the combo count to 0.
- Cleanup and Return:
    - It restores the register state and base pointer, then returns to the caller, passing the combo count ([bp + 6]) as the result of the column combo detection process.

Ldcomboo:

- Initialization:
    - It pushes the base pointer onto the stack and sets up the stack frame.
    - It initializes the combo count to 1 (indicating the current candy in the diagonal being checked) and retrieves the address of the candy to be checked (si) from the stack.
    - It retrieves the shape type of the candy (ax) to be used as a reference for comparison.
- Check Down-Right:
    - It starts by checking candies in the down-right diagonal from the current candy (si). It moves to the next row (rowNum) and next column (colNum) and checks the shape type of the candy (bx).

    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy in the down-right diagonal. This process continues until a non-matching shape type is encountered, the bottom or right edge of the board is reached.
- Check Up-Left:
    - If the combo count is greater than 1 (indicating that at least two matching candies were found in the down-right diagonal), the procedure proceeds to check candies in the up-left diagonal from the current candy (si). It moves to the previous row (rowNum) and previous column (colNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy in the up-left diagonal. This process continues until a non-matching shape type is encountered, or the top or left edge of the board is reached.
- Mark Combos:
    - If the combo count is greater than 2 (indicating that at least three matching candies were found diagonally), it marks all the candies in the combo as part of the combo. It sets the InCombo flag of each candy to tarueeeeee, indicating that it is part of a combo.
    - If the combo count is not greater than 2, it resets the combo count to 0.
- Cleanup and Return:
    - It restores the register state and base pointer, then returns to the caller, passing the combo count ([bp + 6]) as the result of the diagonal combo detection process.

Rdcomboo:

- Initialization:
    - It pushes the base pointer onto the stack and sets up the stack frame.
    - It initializes the combo count to 1 (indicating the current candy in the diagonal being checked) and retrieves the address of the candy to be checked (si) from the stack.
    - It retrieves the shape type of the candy (ax) to be used as a reference for comparison.
- Check Up-Right:
    - It starts by checking candies in the up-right diagonal from the current candy (si). It moves to the previous row (rowNum) and next column (colNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy in the up-right diagonal. This process continues until a non-matching shape type is encountered, or the top or right edge of the board is reached.

- Check Down-Left:
    - If the combo count is greater than 1 (indicating that at least two matching candies were found in the up-right diagonal), the procedure proceeds to check candies in the down-left diagonal from the current candy (si). It moves to the next row (rowNum) and previous column (colNum) and checks the shape type of the candy (bx).
    - If the shape type matches the reference shape type (ax), it increments the combo count and moves to the next candy in the down-left diagonal. This process continues until a non-matching shape type is encountered, or the bottom or left edge of the board is reached.
- Mark Combos:
    - If the combo count is greater than 2 (indicating that at least three matching candies were found diagonally), it marks all the candies in the combo as part of the combo. It sets the InCombo flag of each candy to tarueeeeee, indicating that it is part of a combo.
    - If the combo count is not greater than 2, it resets the combo count to 0.
- Cleanup and Return:
    - It restores the register state and base pointer, then returns to the caller, passing the combo count ([bp + 6]) as the result of the diagonal combo detection process.

Findcomboo:

- Initialization:
    - It pushes the base pointer onto the stack and sets up the stack frame.
    - It initializes the combo count to 0 ([bp + 6]) and retrieves the address of the candy to be checked (si) from the stack.
    - It retrieves the shape type of the candy (ax) to be used as a reference for comparison.
- Check for Null Candy:
    - If the shape type of the current candy is candy_type_NULL, indicating an empty space on the board, it jumps to findcomboo_Return, bypassing combo detection for this candy.
- Row Combo Detection:
    - It calls the rowcomboo procedure to detect combos in the row where the current candy is located. The result of this detection (the number of candies in a combo) is added to the combo count.
    - This procedure checks for consecutive candies of the same shape type in the row, both to the left and right of the current candy.
- Column Combo Detection:

    - It calls the colcomboo procedure to detect combos in the column where the current candy is located. The result of this detection (the number of candies in a combo) is added to the combo count.
    - This procedure checks for consecutive candies of the same shape type in the column, both above and below the current candy.
- Return:
    - It pops any registers pushed onto the stack, restores the stack and base pointer, then returns to the caller, passing the combo count ([bp + 6]) as the result of the combo detection process.

Pushdowncol:

- Initialization:
    - It sets up the stack frame and saves the base pointer.
    - The parameter passed on the stack ([bp + 4]) represents the column index where the candies need to be pushed down. This value is multiplied by the size of a single board square to find the starting position of the column in the boardMatrix.
    - The loop counter ch is initialized to 0.
- Iterating Through Rows:
    - It iterates through each row of the column specified by the column index.
    - For each candy in the current row (si), it checks if the candy is of type candy_type_NULL, indicating an empty space.
- Handling Empty Spaces:
    - If an empty space is found and the current row is not the top row (ch > 0), it checks the candy above it.
        - If the candy above is not empty (candy_type_NULL), it increments the combo count ([bp + 6]). This indicates that a candy has been pushed down.
    - Then, it iterates upwards from the current empty space to shift candies downwards.
        - It swaps the shape types of the current candy with the one above it, simulating the pushing down effect.
        - It redraws both candies to reflect the new positions after the swap.
        - If both candies are empty, it continues iterating upwards until a non-empty candy is found to swap with.
- Moving to the Next Row:
    - After completing the column operation, it moves to the next row by adding the width of a board square to the current position.
    - The loop counter ch is incremented to proceed to the next row.
- Cleanup and Return:

    - Once all rows in the column have been processed, it restores the stack frame and returns to the caller, passing the combo count ([bp + 6]) as the result.

Pushdown:

- Initialization:
    - It sets up the stack frame and saves the base pointer.
    - The loop counter cx is initialized to 0.
    - si is set to point to the beginning of the game board matrix, specifically to the current candy within each square.
- Iterating Through Columns:
    - It iterates through each column of the game board.
    - For each column, it checks the shape type of the candy in the current square (si).
- Calling pushdowncol:
    - It calls the pushdowncol procedure to push down candies in the current column where empty spaces are present.
    - The return value from pushdowncol, which represents the number of combos created by pushing down candies in the column, is added to the combo count ([bp + 6]).
- Moving to the Next Column:
    - After completing the pushdown operation for the current column, it moves to the next column by advancing the pointer si to the next column in the game board matrix.
    - The loop counter cx is incremented to proceed to the next column.
- Cleanup and Return:
    - Once all columns have been processed, it restores the stack frame and returns to the caller, passing the total combo count ([bp + 6]) as the result.

Lvlblock:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
- Level Checking:
    - It checks the value of the current level (currlvl).
    - If the current level is 2, it calls the lvlblock2 procedure to set up the blocks for level 2.
    - If the current level is 3, it calls the lvlblock3 procedure to set up the blocks for level 3.
    - If the current level is not 2 or 3, no action is taken.
- Cleanup and Return:
    - After setting up the blocks for the current level, it restores the register context and returns to the caller.

Lvlblock2:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
- Block Setup:
    - It iterates through each cell on the game board.
    - For each cell, it checks if it matches specific conditions:
    - If the cell is at one of the four corners of the board.
    - If the cell is at one of the positions on the second row or second-to-last row.
    - If the cell is at the middle of the top or bottom row, or at the middle of the left or right column.
    - If any of these conditions are met, it clears the candy from that cell.
    - After clearing the candy, it proceeds to the next cell.
- Cleanup and Return:
    - After setting up the block configuration, it restores the register context and returns to the caller.

Lvlblock3:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
- Block Setup:
    - It iterates through each cell on the game board.
    - For each cell, it checks if the cell is either on the middle row or the middle column.
    - If the cell meets either of these conditions, it clears the candy from that cell.
    - After clearing the candy, it proceeds to the next cell.
- Cleanup and Return:
    - After setting up the block configuration, it restores the register context and returns to the caller.

Findallcombos:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
    - It initializes a counter to keep track of the total number of combos found.
- Iterating Through the Game Board:
    - It loops through each cell on the game board.
    - For each cell, it calls the findcomboo procedure to find combos that include candies adjacent to the current cell.
    - The result of findcomboo (the number of combos found) is added to the counter.
- Cleanup and Return:

    - After iterating through all cells, it restores the register context and returns to the caller, providing the total number of combos found during the process.

Findbombcombos:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
    - It initializes a counter to keep track of the number of candies that match the bomb shape type.
- Iterating Through the Game Board:
    - It loops through each cell on the game board.
    - For each cell, it checks if the candy's shape type matches the shape type of the bomb candy.
    - If a match is found, it marks the candy as part of a combo and increments the counter.
- Cleanup and Return:
    - After iterating through all cells, it restores the register context and returns to the caller, providing the total number of candies that match the bomb shape type.

Popcombos:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
    - It initializes a loop counter cx.
- Iterating Through the Game Board:
    - It loops through each cell on the game board.
    - For each cell, it checks if the candy is marked as part of a combo (InCombo flag).
    - If a candy is part of a combo, it:
        - Retrieves the candy's shape type.
        - Sets the candy's shape type to candy_type_NULL to remove it from the board.
        - Resets the InCombo flag to false to indicate that it's no longer part of a combo.
        - Pauses briefly (sleeps) to create a visual effect.
        - Calls DrawCandy to visually update the game board with the removed candy.
        - Increments the current score.
        - Calls displaymsgs to update any relevant messages or displays in the game.
- Cleanup and Return:
    - After iterating through all cells, it restores the register context and returns to the caller.

Dropcandycol:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
    - It initializes variables, including the loop counter cx.
- Iterating Through the Column:
    - It starts by locating the starting position of the column on the game board.
    - It loops through each row in the column.
    - For each cell in the column:
        - It checks if the cell is empty (candy_type_NULL).
        - If the cell is empty, it:
            - Pauses briefly to create a visual effect.
            - Generates a random candy type to drop into the cell based on the current level.
            - If it's not the top row:
                - Swaps the candy types with the cell above to simulate the candy dropping down.
                - Calls DrawCandy to visually update the game board with the dropped candies.
            - If it's the top row:
                - Just draws the candy without swapping.
            - Increments the count of dropped candies in the column.
- Cleanup and Return:
    - After iterating through all rows in the column, it restores the register context and returns to the caller.

Dropcandy:

- Initialization:
    - It saves the register context by pushing the registers onto the stack.
    - Initializes variables, including the loop counter cx.
- Iterating Through Each Column:
    - It loops through each column on the game board.
    - For each column:
        - It checks each cell in the column to see if it's empty (candy_type_NULL).
        - If an empty cell is found:
            - It calls the dropcandycol procedure to drop candies into the empty cells of that column.
            - It accumulates the count of dropped candies.
- Cleanup and Return:

- After iterating through all columns, it restores the register context and returns to the caller.

Resetgame:

- Initialization:
  - It saves the register context by pushing the registers onto the stack.
  - Initializes variables including the loop counters si and cx.
- Resetting the Board:
  - It iterates through each cell of the game board.
  - For each cell:
    - It sets the content to zero, effectively clearing the cell.
- Resetting Game State Variables:
  - It resets various game state variables:
    - selectedCandy is set to zero, indicating no candy is currently selected.
    - currscore is set to zero, resetting the current score to zero.
    - movesLeft is set to maxMoves, restoring the maximum number of moves allowed.
    - islvlfinish is set to false, indicating the level is not finished.
    - islvlpass is set to false, indicating the level is not yet passed.
- Cleanup and Return:
  - After resetting all variables, it restores the register context and returns to the caller.

StrLen:

- Initialization:
  - It starts by pushing the base pointer onto the stack and setting up its own base pointer.
  - It saves the register context by pushing the registers onto the stack.
  - It initializes si to point to the beginning of the string passed as an argument and sets the length counter to zero.
- Looping Through Characters:
  - It enters a loop that continues until it encounters the null character ('$') which marks the end of the string.
  - Within the loop:
    - It loads the byte at the memory location pointed to by si into the al register.
    - It checks if al contains the null character.
    - If the null character is found, it jumps to the return label.
    - Otherwise, it increments the length counter and moves to the next character by incrementing si.

- Returning the Length:
    - When the null character is encountered, it reaches the return label.
    - It pops the register context from the stack.
    - It pops the base pointer from the stack.
    - Finally, it returns the length of the string (in bytes) in the second argument space on the stack.

Findhighscore:

- Initialization:
    - It pushes the register context onto the stack to save the current state of registers.
    - It loads the scores of each level (lvl1score, lvl2score, lvl3score) into the ax, bx, and cx registers, respectively.
- Comparing Scores:
    - It compares the scores of adjacent levels to find the highest one.
    - It starts by comparing the scores of levels 1 and 2:
        - If the score of level 1 (ax) is greater than that of level 2 (bx), it copies the score of level 1 into the bx register.
    - Then, it compares the updated score of level 2 (bx) with the score of level 3 (cx):
        - If the score of level 2 (bx) is greater than that of level 3 (cx), it copies the score of level 2 into the cx register.
- Storing the Highest Score:
    - After the comparisons, the highest score among the three levels is stored in the highScore variable.
- Restoring Register Context and Returning:
    - It pops the register context from the stack to restore the original state of the registers.
    - Finally, it returns from the procedure.

IntToStr:

- Initialization:
    - It starts by pushing the register context onto the stack to save the current state of registers.
    - It loads the integer value to be converted ([bp + 6]) and the destination address for the string ([bp + 4]) into registers.
- Conversion Loop:
    - The procedure enters a loop (IntToStr_PushLoop) to convert the integer to a string. Inside the loop:
    - It divides the integer by 10, storing the quotient in ax and the remainder in dx.
    - The remainder, which represents a digit of the integer, is pushed onto the stack.
    - It increments the count cx to keep track of the number of digits.

- String Construction:
    - After converting the integer into individual digits and pushing them onto the stack, the procedure enters another loop (IntToStr_PopLoop) to pop the digits from the stack and construct the string.
    - Inside this loop:
        - It pops a digit from the stack and adds the ASCII value of '0' to convert it to its corresponding character representation.
        - It stores this character in the memory location pointed to by si, which represents the destination string.
        - It continues this process until all digits are processed, using the loop instruction to decrement the loop counter cx.
- Null Termination:
    - Once all digits are processed and the string is constructed, it adds a null terminator ('$') to the end of the string to signify its termination.
- Restoration and Return:
    - After completing the conversion, it restores the register context by popping the register values from the stack.
    - Finally, it returns from the procedure.

Savescores:

- File Opening:
    - It starts by pushing the register context onto the stack to save the current state of registers.
    - It loads the offset of the file name (offset scoreFile) into dx and sets al to 02h (for opening an existing file) and ah to 03dh (for opening a file).
    - It then invokes interrupt 21h (int 021h) to open the file. Upon return, the file handle is stored in bx.
- Writing User Information:
    - It writes the username to the file using the writefilestr procedure.
    - It writes an end-of-line marker (endl) to the file.
- Writing Level 1 Score:
    - It writes a message indicating the level (e.g., lvl1filemsg) to the file.
    - It pushes the level 1 score (lvl1score) onto the stack.
    - It converts the score to a string using the IntToStr procedure.
    - It writes the converted score to the file.
    - It writes an end-of-line marker to the file.
- Writing Level 2 Score:
    - Similar to the previous step, it writes the level 2 score to the file.
- Writing Level 3 Score:
    - Similar to the previous steps, it writes the level 3 score to the file.

- Writing High Score:
    - It writes a message indicating the high score (e.g., hsfilemsg) to the file.
    - It calls the findhighscore procedure to find the highest score among the three levels.
    - It pushes the high score onto the stack.
    - It converts the high score to a string using the IntToStr procedure.
    - It writes the converted high score to the file.
    - It writes an end-of-line marker to the file.
- File Closing:
    - After writing all the necessary information, it closes the file by setting ah to 03eh (for closing a file) and invoking interrupt 21h.
- Restoration and Return:
    - Finally, it restores the register context by popping the register values from the stack.
    - It returns from the procedure.

nameInp:

- Initialization:
    - It starts by pushing the register context onto the stack to preserve the current register values.
    - It sets si to point to the beginning of the username buffer and initializes cx to 0, which will count the number of characters entered.
- Input Loop:
    - It enters a loop (NameInput_Loop) where it compares cx (character count) with usernameSize to ensure the input doesn't exceed the buffer size.
    - It uses interrupt 21h with ah set to 01h to read a character from the keyboard (int 021h).
    - If the input character is Enter (ASCII 13) or Backspace (ASCII 8), it jumps to NameInput_Return to terminate input.
    - Otherwise, it stores the input character (al) into the buffer at the current position ([si]), increments si and cx, and continues the loop.
- Return:
    - When input is terminated, it checks if the input string is empty (i.e., the first character is $). If it is, it fills the username buffer with a default name ("Player").
    - If the input string is not empty, it adds the null terminator $ at the end of the input string.
    - It then pops the register context from the stack to restore the original register values.
    - Finally, it returns from the procedure.

Displayrule:

- Displaying Game Title and Rules:
    - It sets the cursor position to (35, 2) and prints the game title using the message stored at the memory address pointed by gameTitleMsg.
    - It then sets the cursor position to (10, 5) and prints the game rules using the message stored at the memory address pointed by gameRulesMsg.
    - Following that, it sets the cursor position to various locations and prints each rule of the game using the messages stored at the memory addresses pointed by rule1Msg, rule2Msg, rule3Msg, rule4Msg, and rule5Msg.
- Prompting for Name Input:
    - After displaying the game rules, it sets the cursor position to (5, 21) and prints the message prompting the user to enter their name using the message stored at the memory address pointed by enterNameMsg.
    - It then calls the nameInp procedure to capture the user's name input.
- Returning from the Procedure:
    - After the user inputs their name, it restores the register context and returns from the procedure.

Displaymsgs:

- Setting Cursor Positions and Printing Messages:
    - It sets the cursor position to (55, 5) and prints the message specific to the current level based on the value stored in currlvl.
    - It then sets the cursor position to (55, 9) and prints the username retrieved from the username variable.
    - After that, it sets the cursor position to (55, 11) and prints the current score using the message stored at the memory address pointed by currscoremsg along with the current score stored in currscore.
    - Next, it sets the cursor position to (55, 13) and prints the required score to pass the level using the message stored at the memory address pointed by reqscoremsg along with the required score threshold specific to the current level (lvl1thresh, lvl2thresh, or lvl3thresh).
    - It then sets the cursor position to (55, 15) and prints the number of moves left using the message stored at the memory address pointed by movesLeftMsg along with the value stored in movesLeft.
    - Finally, it sets the cursor position to (55, 17) and prints the message indicating candy dropping, which is stored in the candydroppppppp variable.
- Returning from the Procedure:
    - After printing all the messages, it restores the register context and returns from the procedure.

DisplayResults:

- Setting Cursor Position:
    - It sets the cursor position to (13, 2) to specify the location where the result message will be displayed.
- Displaying Level Results:
    - Based on the current level (currlvl), it checks if the level was passed (islvlpass is true) or failed.
    - If the level was passed, it prints a success message specific to the current level (e.g., "Level 1 Passed", "Level 2 Passed", or "Level 3 Passed").
    - If the level was failed, it prints a failure message specific to the current level (e.g., "Level 1 Failed", "Level 2 Failed", or "Level 3 Failed").
- Returning from the Procedure:
    - After displaying the result message, it restores the register context and returns from the procedure.

Output:

- Procedure Setup:
    - It sets up the procedure by pushing the base pointer (bp) onto the stack and moving the stack pointer (sp) into the base pointer (bp).
    - Registers are preserved by pushing them onto the stack.
- Conversion to String:
    - It starts by initializing a counter cx to 0.
    - Then, it enters a loop (output_PushLoop) to convert the integer into a string:
    - It divides the integer ([bp + 4]) by 10 to extract the least significant digit.
    - The remainder is pushed onto the stack.
    - The loop counter cx is incremented.
    - If the integer is not yet fully converted (i.e., it's not equal to 0), the loop continues.
    - After the loop, it enters another loop (output_PopLoop) to output each digit character:
    - It pops each digit from the stack.
    - Converts the digit into its ASCII representation by adding '0'.
    - Uses DOS interrupt 21h (int 21h) with function 02h (mov ah, 02h) to output the character.
    - The loop continues until all digits are output.
- Procedure Exit:
    - After all digits are output, it returns from the procedure, restoring the register context and popping the base pointer (bp) from the stack.

Exit:

- Procedure Setup:
    - It begins by pushing the registers onto the stack to preserve their values during execution.
- Saving Scores:
    - It calls the savescores procedure, presumably to save the game scores before exiting.
- Setting Cursor and Exiting:
    - It sets the cursor position to the bottom left corner of the screen, likely to provide a clean output interface.
    - Then, it uses DOS interrupt 21h (int 21h) with function 4Ch (mov ah, 04Ch) to terminate the program.
- Procedure Cleanup:
    - After the program termination, it pops the registers from the stack to restore their original values.
- Return:
    - Finally, it returns from the procedure.