

Final Project Report

EL2003-Computer Organization and Assembly Language - Lab

Semester Project

Umer Farooq(I22-0518)

Muhammed Bilal Ikram (I22-1636)

Arslan Javed (I22-1632)

Section : CY-(A)

*Submitted to: Sir Suleman Saboor*

Department of Cyber Security BS(CY)

FAST-NUCES Islamabad

**Macros**

**1. writePixel macro:**

* Service Routine: BIOS Interrupt 10h (Video Services)
* Functionality: Writes a pixel of a specified color to the screen.
* Arguments:
  + col: Color value to be written.
* Operation:
* Sets the video mode and color using BIOS interrupt 10h.
* Registers Used:
  + ah: Function number (0Ch for setting pixel color).
  + al: Color value to be written.

**2. moveCursor macro:**

* Service Routine: BIOS Interrupt 10h (Video Services)
* Functionality: Moves the cursor to a specified position on the screen.
* Arguments:
  + row: Row position.
  + col: Column position.
  + pg: Page number (for text modes).
* Operation:
  + Sets cursor position using BIOS interrupt 10h.
* Registers Used:
  + ah: Function number (02h for setting cursor position).
  + bh: Video page number.
  + dh: Row position.
  + dl: Column position.

**3. writeLine macro:**

* Service Routine: DOS Interrupt 21h (File Services)
* Functionality: Writes a line of text to a file.
* Registers Used:
  + ah: Function number (40h for writing to a file).
  + bx: File handle.
  + cx: Length of the data to be written.
  + dx: Offset of the data.

**4. initiateFile macro:**

* Service Routine: DOS Interrupt 21h (File Services)
* Functionality: Initializes a file for reading or writing.
* Arguments:
  + FileName: Name of the file to be initialized.
* Operation:
  + Sets file access mode.
  + Calls DOS interrupt 21h to initialize the file.
* Registers Used:
  + ah: Function number (3Dh for opening a file).
  + dx: Offset of the file name.
  + Al: Access mode (0 for reading, 1 for writing, 2 for both).

**5. writeData macro:**

* Service Routine: DOS Interrupt 21h (File Services)
* Functionality: Writes data to a file.
* Arguments:
  + var: Data to be written.
* Registers Used:
  + ah: Function number (40h for writing to a file).
  + bx: File handle.
  + cx: Length of the data to be written.
  + dx: Offset of the data.

**6. initializeVideoMode macro:**

* Service Routine: BIOS Interrupt 10h (Video Services), Interrupt 1Ah (System Services)
* Functionality: Initializes the video mode for graphics.
* Operation:
  + Sets video mode to 640x480 with 16 colors using BIOS interrupt 10h.
  + Gets a system timer to seed a random number generator using Interrupt 1Ah.
* Registers Used:
  + ah, al: Function numbers for setting video mode (00h and 12h respectively).

**7. setActiveDisplay macro:**

* Service Routine: BIOS Interrupt 10h (Video Services)
* Functionality: Sets the active display page.
* Arguments:
  + pageNo: Page number to be set active.
* Operation:
  + Sets the active display page using BIOS interrupt 10h.
* Registers Used:
  + ah: Function number (05h for setting active page).

**8. hideMouseCursor macro:**

* Service Routine: BIOS Interrupt 33h (Mouse Services)
* Functionality: Hides the mouse cursor.
* Operation:
  + Saves current mouse position.
  + Moves mouse out of the visible area.
* Registers Used:
  + ax: Function number (02h for hiding cursor, 03h for getting cursor position, 04h for setting cursor position).
  + cx, dx: Mouse coordinates.

**9. showMouseCursor macro:**

* Service Routine: BIOS Interrupt 33h (Mouse Services)
* Functionality: Shows the mouse cursor.
* Operation:
  + Restores mouse position.
  + Shows the mouse cursor.
* Registers Used:
  + ax: Function number (01h for showing cursor, 04h for setting cursor position).
  + cx, dx: Mouse coordinates.

**10. updateScore macro:**

* Functionality: Updates the game score based on the number of candies removed.
* Operation:
* Adds the specified candy count to the appropriate level's score.

**11. delay macro:**

* Functionality: Introduces a delay based on the provided delay factor.
* Arguments:
  + delayFactor: Factor determining the length of the delay.
* Operation:
  + Uses nested loops to introduce delay.
* Registers Used:
  + cx: Counter register for the outer loop.
  + Bx: Delay factor counter for the inner loop.

**12. getRandNum macro:**

* Functionality: Generates a random number within a specified range.
* Arguments:
  + firstNum: Lower bound of the range.
  + lastNum: Upper bound of the range.
* Operation:
  + Uses a Linear Congruential Generator (LCG) algorithm to generate pseudo-random numbers.
* Registers Used:
  + ax, dx: Accumulator registers for arithmetic operations.
  + cx: Counter register for loop.
  + bp, sp: Base and Stack pointers for managing stack variables.

**13. isEven macro:**

* Functionality: Checks if a number is even.
* Arguments:
  + num: Number to be checked.
* Operation:
  + Divides the number by 2 and checks the remainder.
* Registers Used:
  + ax, bl: Accumulator and Base Low registers for arithmetic operations.
  + bp, sp: Base and Stack pointers for managing stack variables.

**14. setMouseMinMaxPositions macro:**

* Functionality: Sets the minimum and maximum positions for the mouse cursor.
* Operation:
  + Uses BIOS interrupt 33h to set mouse position limits.
* Registers Used:
  + ax, cx, dx: Accumulator and Counter registers for setting mouse position limits.

**15. shiftLogicalLeft macro:**

* Functionality: Performs logical left shift operation on a register.
* Arguments:
  + register: Register to be shifted.
  + count: Number of bits to shift.
* Operation:
  + Uses shl instruction to shift bits to the left.
* Registers Used:
  + cx: Counter register for loop.
  + register: Register being shifted.

**16. shiftLogicalRight macro:**

* Functionality: Performs logical right shift operation on a register.
* Arguments:
  + register: Register to be shifted.
  + count: Number of bits to shift.
* Operation:
  + Uses shr instruction to shift bits to the right.
* Registers Used:
  + cx: Counter register for loop.
  + register: Register being shifted.

**17. getBit macro:**

* Functionality: Retrieves a specific bit from a 16-bit register.
* Arguments:
  + register: Register from which the bit is to be extracted.
  + bitNumber: Position of the bit to retrieve (0-15).
* Operation:
  + Utilizes logical shift operations to isolate the desired bit.
* Registers Used:
  + register: Register containing the bits.
  + bitNumber: Position of the desired bit.

**18. findCoordinatesOfCell macro:**

* Functionality: Calculates and pushes the coordinates of the top-left corner of a selected cell onto the stack.
* Arguments:
  + cellNo: Number representing the selected cell.
* Operation:
  + Divides the cell number by the width of the game board (7 in this case) to determine the row and column numbers.
  + Calculates the x-coordinate based on the column number and pushes it onto the stack.
  + Calculates the y-coordinate based on the row number and pushes it onto the stack.
* Registers Used:
  + ax, bx, bl, bp, sp: General-purpose and base pointer registers for arithmetic operations and stack manipulation.

**19. isCellInRange macro:**

* Functionality: Checks if a cell's coordinates are within a specified range of pixel coordinates.
* Arguments:
  + xSelect, ySelect: Coordinates of the selected cell.
  + xPixel, yPixel: Coordinates of the pixel range.
* Operation:
  + Compares the cell's x and y coordinates with the pixel range.
  + If the cell's coordinates are within the pixel range, sets ax to 1, otherwise sets it to 0.
* Registers Used:
  + ax, bx, cx, dx: General-purpose registers for arithmetic and comparison operations.

**21. multiply macro:**

* Functionality: Multiplies two values and returns the result in ax.
* Arguments:
  + reg: Register containing the first value.
  + intVal: Second value.
* Operation:
  + Multiplies the values stored in reg and intVal using the mul instruction.
  + Stores the result in ax.
* Registers Used:
  + ax, bx: Accumulator and base registers for multiplication.

**22. divide macro:**

* Functionality: Divides two values and returns the quotient in al and remainder in ah.
* Arguments:
  + reg: Dividend.
  + intVal: Divisor.
* Operation:
  + Divides the values stored in reg and intVal using the div instruction.
  + Stores the quotient in al and the remainder in ah.
* Registers Used:
  + ax, bx: Accumulator and base registers for division.

**23. printSpace macro:**

* Functionality: Prints a space character.
* Operation:
  + Moves the space character into dx.
  + Calls DOS interrupt 21h with ah=02h to print the space character.
* Registers Used:
  + ah, dx: Accumulator and data register for DOS interrupt.

**24. printNum macro:**

* Functionality: Prints a number.
* Arguments:
  + Num: Number to be printed.
* Operation:
  + Extracts each digit of the number, converts it to ASCII, and prints it.
* Registers Used:
  + ax, bx, dx: General-purpose registers for arithmetic and data manipulation.

**25. makeNullArray macro:**

* Functionality: Nullifies an array with a specified null value.
* Arguments:
  + arrOffset: Offset of the array.
  + arrSize: Size of the array.
  + nullNumber: Value to be used for nullifying the array.
* Operation:
  + Initializes each element of the array with the specified null value.
* Registers Used:
  + bx, cx, si, dx: General-purpose registers for array manipulation.

**26. areCellsAdjacent macro:**

* Functionality: Checks if two cells are adjacent to each other.
* Arguments:
  + cellOne: Number representing the first cell.
  + cellTwo: Number representing the second cell.
* Operation:
  + Compares the coordinates of the two cells to determine if they are adjacent.
  + If the cells are adjacent, sets ax to 1, otherwise sets it to 0.
* Registers Used:
  + bp, sp: Base and stack pointers for accessing local variables.
  + ax: Accumulator register for storing the result.

**Variables**

**Rule Messages:**

* ruleMsg: String indicating the beginning of game rules.
* continueMsg: Message prompting the player to press Enter to continue after reading the rules.
* rule1, rule2, rule3: Detailed rules of the game.

**User Interaction Messages:**

* nameMsg, nameMsg2: Prompt for the player to enter their name.
* userName, fileUserName: Buffer to store the player's name.
* exitMsg: Message prompting the player to press ESC to exit.

**Game State and Information:**

* cursorRow, cursorCol: Row and column position of the cursor.
* levelOneMsg, levelTwoMsg, levelThreeMsg: Messages indicating the current game level.
* movesMsg: Message indicating the number of moves remaining.
* numMoves: Variable storing the number of moves allowed in the game.
* gridStatus: Status of each cell in the game grid.
* gridXCords, gridYCords: X and Y coordinates of each cell in the game grid.
* generateCandies, generateGrid: Flags indicating whether to generate candies or grid.
* levelOnePoints, levelTwoPoints, levelThreePoints: Points earned by the player in each level.
* scoreDisplayCount: Counter for displaying the player's score.
* scoreMsg: Message indicating the player's score.

**Game Logic and Control:**

* pageNum: Current page number.
* isLevelOne, isLevelTwo, isLevelThree: Flags indicating the current game level.
* pixelColor: Color value for pixels.
* randNumSeed: Seed value for generating random numbers.
* finalCandyNum, successfulSwap: Variables for managing candy swapping.
* tempFinalCandyNumIndexSI, finalCellNo: Temporary variables for managing candies.
* isSelected, selectedCellNo, checkForSwap: Variables for cell selection and swapping.
* initCandyNum, initCellNo: Initial candy and cell numbers.
* successfulCrush, tempCandyNo: Variables for managing crushed candies.
* tempCandyCounter, crushingCandiesCountRow: Counters for crushing candies.
* bombCounter, bombPositions: Counter and positions for bombs.
* tempCandyBottom, candyBottomIndex: Variables for managing candy positions.
* countOfCrushingCandies, startingCellNoRow: Count of crushing candies and starting cell numbers.
* tempCandyTop, candyTopIndex: Variables for managing candy positions.
* mouseInitXCord, mouseFinXCord, mouseInitYCord, mouseFinYCord: Mouse coordinates.
* mouseXCordSaveVar, mouseYCordSaveVar: Saved mouse coordinates.
* tempRtAddressfindCellNo, tempCandyNoForScore: Temporary variables.
* tempCellNo, tempCandyNoForSwapping, tempCellNoForSwapping: Temporary variables.
* tempForHexaCandy: Temporary variable for hexa candy.
* winMsg1, winMsg2, winMsg3, winMsg4, winMsg5, winMsgScore: Winning messages and scores.
* fileName, handle, Filetemp, fileDigitCount: File-related variables for saving game data.

**Procedures**

**Main:**

* Setting up Data Segment:
  + mov ax,@data: Load the address of the data segment.
  + mov ds,ax: Set the data segment register to the address of the data segment.
  + mov es, ax: Set the extra segment register to the address of the data segment.
* Opening File for Writing:
  + makefile fileName: Opens a file named fileName for writing.
  + mov handle, ax: Stores the file handle in the handle variable.
  + mov ah,42h: Sets up for adjusting the file pointer.
  + mov bx,handle: Specifies the file handle for which the file pointer is being adjusted.
  + xor cx,cx and xor dx,dx: Clears CX and DX registers.
  + mov al, 2: Specifies movement based on end of file.
  + INT 21H: DOS interrupt to adjust the file pointer.
* Initialization and Display:
  + makescreen: Clears the screen and sets up video mode.
  + startingcout and coutrules: Procedures to display initial messages and game rules.
  + Writes fileUserName to the file.
  + setmouse\_xy: Sets up the mouse coordinates.
  + initgrid, startcrush: Initializes game grid and starts the initial candy crush.
* Main Game Loop:
  + Checks if game conditions for level completion or failure are met.
  + Moves to the next level if conditions are met.
  + Continues displaying game data, player score, and updating grid and candies.
  + Checks for mouse input and handles candy swapping.
  + Checks for exit key press (ESC) to break out of the loop.
* Game Over:
  + If the loop breaks (game over), it writes the final score to the file and closes it.
  + Sets the cursor position and ends the program.

**Coutrules:**

* Pushing Registers and Variables:
  + pushA: Pushes all the general-purpose registers onto the stack to save their current values.
* Displaying Rule Messages:
  + Sets up the display of various rule messages:
  + ruleMsg: Displays the main heading for the rules.
  + rule1, rule2, rule3: Display specific rules.
  + continueMsg: Displays a message prompting the player to continue.
* Displaying Text at Specific Locations:
  + Each rule message is displayed at a specific position on the screen using the displayColorData subroutine.
  + The subroutine takes the message offset and length as parameters and sets the cursor position accordingly.
  + After each message is displayed, the cursor position is updated for the next message.
* Waiting for Player Input:
  + After displaying all the messages, the cursor position is set to coordinates (100, 100) which are off-screen.
  + Then, a loop waits for the player to press the Enter key (AL == 13).
  + The loop continuously checks for keyboard input using DOS interrupt int 21h until the Enter key is pressed.
* Restoring Registers and Exiting:
  + popA: Pops the previously pushed registers from the stack to restore their original values.
  + ret: Returns control to the calling routine.

**Coutlose:**

* Pushing Registers and Variables:
  + pushA: Pushes all the general-purpose registers onto the stack to save their current values.
* Displaying Messages:
  + makescreen: Clears the screen to prepare for displaying messages.
  + Sets the cursor position to display the messages at specific locations:
  + moveMsg1: Displays a message indicating failure to complete the level optimally.
  + moveMsg4: Displays the player's score.
  + Uses the displayColorData subroutine to display the messages with color attributes.
* Displaying the Score:
  + Checks which level the player is on (isLevelOne, isLevelTwo, or isLevelThree).
  + If the level points are zero, displays '0' directly.
  + If the level points are greater than zero, converts the points to ASCII characters and displays them using DOS interrupt int 21h to print each digit.
* Waiting for Player Input:
  + Moves the cursor position to a specific location to prompt the player to press Enter to exit the lose screen.
  + Enters a loop to wait for the player to press the Enter key (AL == 13) using DOS interrupt int 21h.
* Restoring Registers and Exiting:
  + Restores the previously pushed registers from the stack to their original values using popA.
  + ret: Returns control to the calling routine.

**removeBombs:**

* Initialization:
  + mov bx,offset gridStatus: Sets the BX register to point to the beginning of the gridStatus array, which likely represents the status of each cell in the game grid.
  + mov si,0: Initializes the SI register to 0, which will be used as an index to iterate through the gridStatus array.
* Loop through the Grid:
  + .repeat: Starts a loop to iterate through each element of the gridStatus array.
  + mov dx,[bx+si]: Loads the value at the memory location pointed to by BX+SI into the DX register. This value likely represents the status of a cell in the game grid.
* Removing Bombs:
  + .if(dx==6): Checks if the value in DX is equal to 6, which likely represents a bomb.
  + getRandNum 1,5: Calls a subroutine or macro to generate a random number between 1 and 5. This random number will be used to replace the bomb.
  + pop [bx+si]: Removes the bomb from the grid by replacing it with a random candy, which is popped from the stack.
* Iteration:
  + add si,2: Moves to the next cell in the gridStatus array. Since each element in the array likely occupies 2 bytes (for example, each cell may have a status represented by a 2-byte value), SI is incremented by 2.
  + .until(si==98): Checks if the end of the grid has been reached. If not, the loop continues.
* Return:
  + ret: Returns control to the calling routine.

**makeGrid:**

* Initialization:
  + pushA: Pushes all the general-purpose registers onto the stack to preserve their values.
  + hideMouseCursor: Hides the mouse cursor from the screen to prevent it from being displayed over the grid.
* Drawing Horizontal Lines:
  + push 100 and push 180: Pushes the starting y-coordinate (100) and the starting x-coordinate (180) onto the stack.
  + mov bp, sp: Moves the stack pointer (SP) into the base pointer (BP) register to access function parameters.
  + mov cx, 8: Initializes the CX register to 8, representing the number of horizontal lines to draw.
  + horizontalLines: Label indicating the start of the loop for drawing horizontal lines.
  + Within this loop:
    - mov ah, 0ch: Sets the video mode to draw lines.
    - push cx: Pushes the loop counter onto the stack.
    - .repeat: Starts a repeat loop.
    - mov al, 00001111b: Sets the line style to draw solid lines.
    - mov bh, 0: Sets the video page.
    - mov cx, [bp] and mov dx, [bp+2]: Retrieves the x and y coordinates from the stack.
    - pushA: Pushes all the general-purpose registers onto the stack to preserve their values.
    - Checks if a cell is already selected and whether it's within the range of selected cells.
    - Draws a line using BIOS interrupt 10h if necessary.
    - inc word ptr [bp]: Increments the x-coordinate.
    - .until ax == 460: Checks if the end of the grid has been reached.
    - pop cx: Pops the loop counter from the stack.
    - mov word ptr [bp], 180 and add word ptr [bp+2], 40: Resets the x-coordinate and increments the y-coordinate.
    - dec cx and jnz horizontalLines: Decrements the loop counter and continues looping if it's not zero.
* Drawing Vertical Lines:
  + Similar to the horizontal lines, but for vertical lines. The loop iterates through the x-coordinate instead of the y-coordinate.
* Conditional Drawing:
  + If a cell is already selected, additional lines are drawn around it to highlight it.
* Finalization:
  + mov generateGrid, 0: Signals that the grid has been generated.
  + showMouseCursor: Displays the mouse cursor.
  + popA: Pops the general-purpose registers from the stack to restore their values.
  + ret: Returns control to the calling routine.

**Startcrush:**

* Initialization:
  + A loop is started using .repeat, indicating that the crushing process may need to be repeated multiple times until no more successful crushes occur.
  + mov successfulCrush, 0: Initializes a variable successfulCrush to 0. This variable will be used to determine whether any successful crushes occur during the current iteration of the loop.
* Crushing Process:
  + Inside the loop, three procedures are called sequentially:
  + call removeBombs: Removes any color bombs that may be present in the grid.
  + call dropCandies: Drops candies down to fill any empty spaces created by crushing.
  + call crushCandies: Attempts to crush candies and identify combinations.
  + These procedures are repeated until no more successful crushes occur (until(successfulCrush == 0)).
* Post-Crush Processing:
  + After the crushing process is completed:
  + mov generateCandies, 1: Signals that new candies need to be generated since some were crushed.
  + Depending on the current level (isLevelOne, isLevelTwo, or isLevelThree), the points are reset to 0 for the respective level:
  + If it's Level 1, levelOnePoints are reset to 0.
  + If it's Level 2, levelTwoPoints are reset to 0.
  + If it's Level 3, levelThreePoints are reset to 0.
* Return:
  + ret: Returns control to the calling routine.

**Coutfilescore:**

* Initialization:
  + fileDigitCount, 0: Initializes a temporary variable to store the length of the score.
  + mov ax, bx: Copies the score from bx to ax.
  + mov bx, 10: Sets bx to 10 for later division.
* Push Data onto Stack:
  + pushData:: Label for the loop.
  + mov dx, 0: Clears dx.
  + div bx: Divides ax by 10, storing the quotient in ax and the remainder in dx.
  + push dx: Pushes the remainder (which represents a digit of the score) onto the stack.
  + inc fileDigitCount: Increments fileDigitCount to keep track of the number of digits pushed onto the stack.
  + cmp ax, 0: Compares ax with 0 to determine if there are more digits to process. If ax is not 0, the loop continues.
* Write to File:
  + writeToFile:: Label for the loop to write data to the file.
  + cmp fileDigitCount, 0: Compares fileDigitCount with 0 to check if all digits have been processed.
* je closeFile: If all digits have been processed, the loop exits to closeFile.
  + dec fileDigitCount: Decrements fileDigitCount to get the last digit from the stack.
  + pop bx: Pops the last digit from the stack into bx.
  + add bx, 48: Converts the digit to its ASCII representation by adding 48 ('0').
  + mov Filetemp, bx: Moves the ASCII digit to Filetemp.
  + Writes the ASCII digit to the file using interrupt 21h, function 40h (WRITE TO FILE).
  + The loop continues until all digits have been written to the file.
* Close File and Return:
  + closeFile:: Label for closing the file.
  + ret: Returns control to the calling routine.

**makeCandies:**

* Initialization:
  + pushA: Pushes all general-purpose registers onto the stack to preserve their values.
  + hideMouseCursor: Hides the mouse cursor to prevent it from being displayed during the generation of candies.
* Loop through Grid Cells:
  + mov si, 0: Initializes the source index si to 0, representing the beginning of the grid status array.
  + mov cx, 0: Initializes the loop counter cx to 0.
  + .repeat: Marks the beginning of the loop.
  + pushA: Pushes the general-purpose registers onto the stack to preserve their values for each iteration of the loop.
* Push Grid Data onto Stack:
  + Retrieves information about the candy at the current grid position (candy number, Y coordinate, and X coordinate).
  + Pushes the candy number, Y coordinate, and X coordinate onto the stack in reverse order (from right to left).
  + Calls the drawcandy subroutine to display the candy on the screen based on the retrieved information.
* Update Loop Variables:
  + popA: Restores the values of the general-purpose registers from the stack.
  + add si, 2: Moves to the next candy in the grid status array (each candy occupies 2 bytes).
  + inc cx: Increments the loop counter to keep track of the number of candies processed.
* Loop Termination:
  + .until(cx == 49): Terminates the loop when 49 candies (representing the entire grid) have been processed.
* Finalization:
  + mov generateCandies, 0: Indicates that candies have been generated and displayed, so the generation process is now complete.
  + showMouseCursor: Restores the visibility of the mouse cursor.
  + popA: Restores the values of the general-purpose registers from the stack.
  + ret: Returns control to the calling routine.

**crushCandies:**

* Initialization:
  + pushA: Pushes all general-purpose registers onto the stack to preserve their values.
  + Initializes various counters and variables used throughout the procedure.
* Horizontal Crush Check:
  + The procedure starts by checking for horizontal candy combinations within each row of the grid.
  + It iterates through each candy in the grid, counting consecutive candies of the same type (tempCandyCounter).
  + If it finds three or more consecutive candies of the same type, it marks the combination for crushing.
  + The starting and ending cell numbers of the combination are stored in arrays (startingCellNoRow and endingCellNoRow).
* Vertical Crush Check:
  + Next, the procedure checks for vertical candy combinations within each column of the grid.
  + Similar to the horizontal check, it iterates through each candy in the grid, counting consecutive candies of the same type.
  + If it finds three or more consecutive candies of the same type, it marks the combination for crushing.
  + The starting and ending cell numbers of the combination are stored in arrays (startingCellNoCol and endingCellNoCol).
* Crushing Process:
  + Once all combinations are identified, the procedure crushes the candies.
  + It loops through the rows and columns where combinations were found.
  + For each combination, it updates the score, removes the candies from the grid status, and visually removes them from the GUI.
  + If a combination of three or more candies is found, it either drops a color bomb or removes the candies.
  + The crushing process is performed separately for rows and columns.
* Loop Termination and Cleanup:
  + The procedure terminates the loops once all combinations have been processed.
  + It restores the values of registers, pops values from the stack, and returns control to the calling routine.

**removeCandy:**

* Initialization:
  + Registers are preserved using pushA.
  + The cell number to remove candies from is retrieved from the stack (tempCellNo).
  + The coordinates of the cell are obtained using findCoordinatesOfCell and stored in ax and dx.
* Candy Removal Loop:
  + The procedure enters a nested loop to iterate over the pixels within the cell.
  + The outer loop iterates over the vertical pixels (cx) within the cell.
  + The inner loop iterates over the horizontal pixels within the cell.
  + Within the inner loop, each pixel's color is set to black (00000000b) using BIOS interrupt 10h, function 0Ch, to visually remove the candy.
* Loop Termination and Cleanup:
  + Once all pixels within the cell are processed, the loops terminate.
  + The procedure restores the values of registers and variables from the stack.
  + Mouse cursor is shown again using showMouseCursor.
  + Finally, the procedure returns control to the calling routine after cleaning up the stack and destroying local variables.

**explodeBomb:**

* Initialization:
  + Registers are preserved using pushA to ensure they remain unchanged throughout the procedure.
  + The bx register is set to point to the gridStatus array, which holds the status of candies on the game grid.
  + The loop index si is initialized to 0 to iterate through the gridStatus array.
* Bomb Explosion:
  + The procedure enters a loop to iterate through each cell on the game grid.
  + For each cell, it checks if the candy in the current cell (dx) matches the candy that was swapped with the bomb (stored in [bp]).
  + If a match is found:
  + The candy number is stored in tempCandyNoForScore.
  + The updateScore procedure is called to update the game score based on the exploded candy.
  + The candy in the current cell is replaced with a "removed" status (represented by 9), indicating that it has been exploded.
  + The removeCandy procedure is called to visually remove the exploded candy from the game grid.
* Loop Termination:
  + The loop continues until all cells in the grid have been processed.
  + The loop terminates when the index si reaches 98, which represents the end of the grid (48\*2 = 96, plus 2 for the last cell).
* Cleanup:
  + After processing all cells, registers are restored to their original values using popA.
  + The procedure returns control to the calling routine after cleaning up the stack and destroying the local variable storing the candy number.

**Swapready:**

* Initialization:
  + Registers are preserved using pushA to ensure they remain unchanged throughout the procedure.
* Check for Swap:
  + It checks if the checkForSwap flag is set to 1, indicating that a swap operation has been requested by the player.
* Main Swap Logic:
  + It iterates through the grid to find the initial and final cell numbers selected by the player for swapping.
  + Once both initial and final cell numbers are identified, it checks if the cells are adjacent using the areCellsAdjacent procedure.
  + If the cells are adjacent and meet certain conditions based on the game level:
  + It removes candies from both cells using the removeCandy procedure.
  + It swaps the candies' positions in the gridStatus array.
  + It updates the game grid using the makeGrid and makeCandies procedures.
  + If one of the swapped candies is a color bomb, it triggers an explosion and removes the bomb.
  + It then checks for successful crushes and continues crushing candies until there are no more successful crushes.
  + If the swap operation results in no successful crushes, it swaps the candies back to their original positions and regenerates candies.
  + It resets various flags and variables related to the swap operation.
* Cleanup and Return:
  + After completing the swap operation, the procedure restores the registers using popA and returns control to the calling routine.

**displayPlayerScore:**

* Initialization:
  + Registers are preserved using pushA to ensure they remain unchanged throughout the procedure.
* Display Score Message:
  + It prepares the message to display the score using the displayColorData procedure, which likely handles the color and positioning of text on the screen.
  + It sets the cursor position using setcurs to specify where the score will be displayed on the screen.
* Level-Specific Score Display:
  + It checks which level the player is currently in (isLevelOne, isLevelTwo, or isLevelThree flags).
  + Depending on the level, it checks if the score is zero. If it is, it directly displays '0' as the score.
  + If the score is non-zero, it converts the score to a string for display:
  + It initializes variables (dx for storing remainders, bx for the divisor, and ax for the score value).
  + It iterates through the score value using a loop:
  + It divides the score by 10 (div bx) to separate each digit.
  + It pushes the remainder onto the stack.
  + It increments scoreDisplayCount to keep track of the number of digits in the score.
  + After separating the digits, it pops them from the stack and displays each digit by adding 48 (ASCII value for '0') and using interrupt 21h (DOS service) with function 02h (display character).
  + It decrements scoreDisplayCount until all digits are displayed.
* Cleanup and Return:
  + After displaying the score, it restores the registers using popA and returns control to the calling routine.

**displayGameData:**

* Initialization:
  + Registers are preserved using pushA to ensure they remain unchanged throughout the procedure.
* Display Level Message:
  + It checks which level the player is currently in (isLevelOne, isLevelTwo, or isLevelThree flags).
  + Depending on the level, it prepares the corresponding level message (levelOneMsg, levelTwoMsg, or levelThreeMsg) for display using the displayColorData procedure. It specifies the position of the message on the screen.
* Display Player Name:
  + It prepares the player's name message (nameTextMsg) for display using the displayColorData procedure. It specifies the position of the message on the screen.
  + It sets the cursor position to display the player's actual name obtained from the userName variable using interrupt 21h function 09h (display string).
* Display Exit Message:
  + It prepares the exit message (exitMsg) for display using the displayColorData procedure. It specifies the position of the message on the screen.
* Display Remaining Moves:
  + It prepares the moves message (movesMsg) for display using the displayColorData procedure. It specifies the position of the message on the screen.
  + It checks the number of remaining moves (numMoves):
  + If the number of moves is zero, it directly displays '0' at the specified position on the screen.
  + If the number of moves is less than or equal to 9, it adjusts the cursor position to ensure proper alignment, then converts the number of moves to a character and displays it on the screen.
  + If the number of moves is greater than 9, it converts the number of moves to a string and displays each digit individually:
    - It initializes variables (dx for storing remainders, bx for the divisor, and ax for the number of moves).
    - It iterates through the number of moves using a loop:
      * It divides the number of moves by 10 (div bx) to separate each digit.
      * It pushes the remainder onto the stack.
      * It increments scoreDisplayCount to keep track of the number of digits in the number of moves.
    - After separating the digits, it pops them from the stack and displays each digit by adding 48 (ASCII value for '0') and using interrupt 21h function 02h (display character).
* Cleanup and Return:
  + After displaying all game-related information, it restores the registers using popA and returns control to the calling routine.

**dropCandies:**

* Initialization:
  + Registers are preserved using pushA to ensure they remain unchanged throughout the procedure.
  + The base address of the game grid (gridStatus) is loaded into the bx register.
  + The variable si is initialized to the index of the bottom row of the game grid (96, corresponding to the last row in a 7x7 grid, where each cell occupies two bytes in the array).
* Dropping Candies:
  + The procedure enters a loop to iterate through each cell of the game grid from bottom to top.
  + Within the loop:
  + It checks if the current cell contains a vacant space (candy number 9).
  + If a vacant space is found:
    - It saves the index of the bottom vacant cell (candyBottomIndex).
    - It searches for the nearest non-vacant candy cell above the vacant space.
    - If there are no candies above (reached the top of the grid) or the cell above is empty (for level 2), it generates a new random candy for the current vacant cell using the getRandNum procedure.
    - Otherwise, it swaps the candy from the cell above with the vacant space, effectively dropping the candy down.
    - After dropping the candy, it exits the loop for the current column.
  + The loop continues until it reaches the top row of the game grid (index -2).
* Cleanup and Return:
  + After processing all cells in the grid, it restores the registers using popA and returns control to the calling routine.

**findCellNo:**

* Initialization:
  + The procedure starts by popping the return address from the stack and pushing it back after storing it in the tempRtAddressfindCellNo variable. This ensures that the return address is preserved during the execution of the procedure.
  + Registers are preserved using pushA to prevent their values from being altered.
* Finding X Coordinate:
  + It initializes bx with the offset of gridXCords and si with 0, preparing to iterate over the X coordinates of the grid.
  + It enters a loop that iterates over the X coordinates of the grid cells (from left to right).
  + Within the loop:
    - It retrieves the X coordinate of the current cell from memory.
    - It adds 40 to the X coordinate to account for any offset.
    - If the modified X coordinate is greater than or equal to the X coordinate passed to the procedure (word ptr[bp], representing the X coordinate clicked by the user), it updates the X coordinate stored at [bp] and exits the loop.
    - Otherwise, it increments si to move to the next X coordinate.
  + After finding the appropriate X coordinate, it exits the loop.
* Finding Y Coordinate:
  + Similar to the X coordinate search, this part of the procedure iterates over the Y coordinates of the grid cells (from top to bottom).
  + It initializes bx with the offset of gridYCords and si with 0.
  + It enters a loop that iterates over the Y coordinates of the grid cells.
  + Within the loop:
    - It retrieves the Y coordinate of the current cell from memory.
    - It adds 40 to the Y coordinate to account for any offset.
    - If the modified Y coordinate is greater than or equal to the Y coordinate passed to the procedure (word ptr[bp+2], representing the Y coordinate clicked by the user), it updates the Y coordinate stored at [bp+2] and exits the loop.
    - Otherwise, it increments si to move to the next row of Y coordinates.
  + After finding the appropriate Y coordinate, it exits the loop.
* Calculating Cell Number:
  + It calls the calculateCellNo procedure to calculate the cell number based on the X and Y coordinates obtained from the previous steps.
  + The return address is then pushed back onto the stack, and the procedure returns.

**Chkmouse:**

* Initialization:
  + It begins by setting ax to 01 and invoking interrupt int 33h to display the mouse cursor.
  + Then, it sets ax to 5 and bx to 0 before calling interrupt int 33h to check if the left mouse button (LMB) is being pressed.
* Checking Mouse Input:
  + If isSelected is 0 (indicating no candy is currently selected), it checks if the left mouse button is pressed (bx != 0).
    - If the left mouse button is pressed:
      * It enters a loop to continuously check if the left mouse button is held down.
      * Within the loop:
        + It sets cx and dx to 0 and invokes interrupt int 33h with ax set to 3 to get the current mouse cursor coordinates.
        + It pushes necessary values onto the stack and calls the findCellNo procedure to determine the cell number corresponding to the mouse cursor position.
        + It pops the initial cell number (initCellNo) from the stack and sets selectedCellNo to this value.
        + It sets generateGrid to 1, indicating that the game grid should be updated.
      * The loop continues until the left mouse button is released.
  + If isSelected is 1 (indicating a candy is already selected), it performs a similar process to check if the left mouse button is pressed.
    - If the left mouse button is pressed:
      * It enters a loop to continuously check if the left mouse button is held down.
      * Within the loop:
        + It sets checkForSwap to 1, indicating that the game should check if a candy swap is possible.
        + It resets isSelected to 0 and selectedCellNo to 100, indicating no candy is currently selected.
        + It gets the current mouse cursor coordinates.
        + It determines the cell number corresponding to the mouse cursor position using the findCellNo procedure.
        + It sets generateGrid to 1 to update the game grid.
      * The loop continues until the left mouse button is released.
* Returning Control:
  + Finally, the procedure returns.

**Initgrid:**

* Initializing Candy Numbers:
  + The procedure begins by setting up a loop to populate the gridStatus array with random candy numbers.
  + Depending on the current game level (isLevelOne, isLevelTwo, or isLevelThree), it generates random candy numbers for each cell of the grid.
  + For Level 2 and Level 3, certain cells are initialized with specific candy numbers (0 for Level 2 and 7 for Level 3) as defined in the game logic.
* Initializing X Coordinates:
  + After initializing the candy numbers, it sets up another loop to populate the gridXCords array with x coordinates for each cell.
  + It starts by pushing the value 180 onto the stack, which represents the initial x coordinate from the left side of the screen to the first grid box.
  + Within a nested loop, it sets each x coordinate for the grid cells, incrementing by 40 pixels for each cell horizontally.
* Initializing Y Coordinates:
  + Similar to the process for x coordinates, it sets up a loop to populate the gridYCords array with y coordinates for each cell.
  + It starts by pushing the value 100 onto the stack, which represents the initial y coordinate from the top side of the screen to the first grid box.
  + Within a nested loop, it sets each y coordinate for the grid cells, incrementing by 40 pixels for each cell vertically.
* Cleanup:
  + After completing the initialization, it cleans up by popping the local variables from the stack.
* Return:
  + Finally, it returns from the procedure.

**calculateCellNo:**

* Argument Passing:
  + The procedure receives the x and y coordinates through the base pointer [bp]. The x coordinate is located at [bp] and the y coordinate is located at [bp+2].
* Calculating Column Number:
  + The x coordinate is subtracted by 180 to adjust it relative to the leftmost position of the grid.
  + Then, it divides the adjusted x coordinate by 40 (the width of each grid cell) to determine the column number.
  + The quotient (result of the division) is saved back into [bp], representing the column number.
* Calculating Row Number:
  + Similar to the column calculation, the y coordinate is adjusted by subtracting 100 to align it with the topmost position of the grid.
  + Then, it divides the adjusted y coordinate by 40 (the height of each grid cell) to determine the row number.
  + The quotient is saved back into [bp+2], representing the row number.
* Calculating Cell Number:
  + The row number is multiplied by 7 (the number of columns in the grid) to get the starting index of the row.
  + Then, it adds the column number to calculate the cell number.
  + The resulting cell number is stored in the local variable [bp+4].
* Cleanup and Return:
  + After calculation, the registers are restored, and the procedure returns, destroying the x and y coordinates from the stack while keeping the calculated cell number in the local variable.

**Drawcandy:**

* Checking Candy Number:
  + The procedure first checks the candy number passed via the base pointer [bp+4] to determine which candy to draw.
* Drawing the Candy:
  + Depending on the candy number:
    - For candy number 1: It draws a green box.
    - For candy number 2: It draws a cyan diamond.
    - For candy number 3: It draws a magenta triangle.
    - For candy number 4: It draws a toffee.
    - For candy number 5: It draws a hexa candy.
    - For candy number 6: It draws a color bomb.
* Drawing Shapes:
  + The shapes are drawn pixel by pixel using BIOS video interrupt int 10h.
  + Each shape is drawn within a loop, where the loop iterates over the width and height of the shape, drawing pixels accordingly.
  + Different shapes are drawn using different combinations of bits for the pixel color.
* Return:
  + After drawing the candy, the procedure returns, destroying the candy number, x-coordinate, and y-coordinate from the stack.

**Startingcout:**

* Displaying Prompt:
  + It first displays the first part of the name prompt stored in the nameMsg variable.
  + Then, it displays the second part of the name prompt stored in the nameMsg2 variable.
  + Setting Cursor Position:
  + It sets the cursor position to the location where the user is expected to input their name.
* Input Handling:
  + It initializes a loop labeled input to continuously read characters from the keyboard input using interrupt 21h, function 01h (int 21h, ah=01h).
  + It stores each character inputted by the user into consecutive memory locations starting from the address pointed to by the si register (which points to the userName variable).
  + The loop continues until the Enter key (carriage return) is pressed, as indicated by the ASCII code 13.
  + Once the Enter key is pressed, the loop terminates, and a null-terminated string is formed by appending a '$' character at the end of the entered name.
* Return:
  + Finally, it returns, allowing the program flow to proceed after capturing the user's name.

**displayColorData:**

* Initialization:
  + It first moves the stack pointer (sp) into the base pointer (bp) to access the parameters passed to the procedure.
  + The parameters passed on the stack are the length of the data to be displayed (dx) and the offset of the data (si).
* Setting Up Display Parameters:
  + It moves the length of the data from the stack into the dx register, which stores the number of characters to be displayed.
  + The last character in the data is usually a null terminator ('$'), so it decrements the length (cx) by one to ignore this character.
* Setting Display Attributes:
  + It sets up the display attributes required for displaying the text. In this case:
    - ah = 13h: Function 13h is used for writing a string with attributes.
    - al = 01h: This value indicates that the cursor should move after the call.
    - bh = pageNum: This specifies the display page.
    - bl = 0fh: This value represents the color attribute for the text. Here, it's set to bright white text on a black background.
* Displaying Text:
  + It sets the cursor position to the row and column specified by cursorRow and cursorCol.
  + It calls interrupt 10h (video services) to display the string with the specified attributes.
* Return:
  + Finally, it returns, allowing the program flow to proceed after displaying the text.

**Coutwin:**

* Initializing the Screen:
  + It first calls the makescreen procedure to initialize the screen.
* Displaying the Win Message:
  + It sets the cursor position to display the first part of the win message (winMsg1) at row 5 and column 24.
  + Calls displayColorData to display the first part of the win message.
  + Similar steps are repeated for displaying other parts of the win message (winMsg2, winMsg3, winMsg4, and winMsg5) at different cursor positions.
* Displaying Level Scores:
  + It sets the cursor position to display the scores for each level.
  + It calculates the scores for each level by dividing the respective score variables (levelOnePoints, levelTwoPoints, levelThreePoints) by 10 and displaying the quotient as a string.
* Waiting for User Input:
  + It sets the cursor position to row 50 and column 50 to wait for user input.
  + It waits for the user to press the Enter key (al == 13) to continue.
* Return:
  + Finally, it returns, allowing the program flow to proceed after displaying the win message and scores.