**Coal Project Documentation**

1. ***Variables Used:***

**🡪 Ball-Related Variables:**

* **DotX:** Stores the x-coordinate of the ball (dot) on the screen.
* **DotY:** Stores the y-coordinate of the ball (dot) on the screen.
* **DotSize:** Represents the size (diameter) of the ball.
* **DotSpeedX:** Controls the speed of the ball's movement along the x-axis.
* **DotSpeedY:** Controls the speed of the ball's movement along the y-axis.
* **DotXCentre**: Defines the center x-coordinate where the ball resets after a goal.
* **DotYCentre:** Defines the center y-coordinate where the ball resets after a goal.

**🡪 Screen & Border Variables:**

* **Width**: Specifies the width of the screen in pixels (320 for mode 13h).
* **Height**: Specifies the height of the screen in pixels (200 for mode 13h).
* **padding**: Adds a buffer zone to prevent the ball from overlapping screen edges.
* **BorderLeftX**: X-coordinate of the left border.
* **BorderLeftY**: Y-coordinate of the left border.
* **BorderWidth**: Width of the vertical borders.
* **BorderHeight**: Height of the vertical borders.
* **BorderRightX**: X-coordinate of the right border.
* **BorderRightY**: Y-coordinate of the right border.
* **BorderWidthN**: Width of the horizontal borders.
* **BorderHeightN**: Height of the horizontal borders.
* **BorderTopY**: Y-coordinate of the top horizontal border.
* **BorderBottomY**: Y-coordinate of the bottom horizontal border.

**🡪 Paddle-Related Variables:**

* **PeddleLeftX**: Starting x-coordinate of the left paddle.
* **PeddleLeftY**: Starting y-coordinate of the left paddle.
* **PeddleLeftWidth**: Width of the left paddle.
* **PeddleLeftHeight**: Height of the left paddle.
* **PeddleRightX**: Starting x-coordinate of the right paddle.
* **PeddleRightY**: Starting y-coordinate of the right paddle.
* **PeddleRightWidth**: Width of the right paddle.
* **PeddleRightHeight**: Height of the right paddle.
* **PeddleVelocity**: Controls the speed of the paddle's movement.

**🡪 Scoring and Game State Variables:**

* **PLeftPoint**: Tracks the score of the left player.
* **PRightPoint**: Tracks the score of the right player.
* **PlayerPoint**: Stores the goal limit for winning the game.
* **GamePaused**: Flag indicating whether the game is paused (1 = paused, 0 = running).
* **WinnerIndex**: Stores the ID of the winning player (1 for Player 1, 2 for Player 2).

**🡪 Text and UI Variables:**

* **ShowPlayerOne**: String to display the score of Player 1.
* **ShowPlayerTwo**: String to display the score of Player 2.
* **TextGameOver**: Text string to display "Game Over".
* **GameWinner**: Text string indicating which player won.
* **PlayAgain**: Prompt text for restarting the game.
* **MainMenu**: Text for returning to the main menu.
* **MainMenuTitle**: Title text for the main menu.
* **StartGame**: Instruction text to start the game.
* **ExitGame**: Instruction text to exit the game.
* **Goals**: Prompt text to enter the number of goals required to win.

1. ***Functions:***

**1. main:**

**Purpose**:  
The **main function** is the starting point of the program. It initializes the game environment, handles the menu, and ensures the game runs properly.

***call inputScore***

***;Set graphics mode (Mode 13h - 320x200, 256 colors)***

***mov ax, 0x0013***

***int 0x10***

**call inputScore:** Likely calls a function to initialize or display the score at the start.

**mov ax, 0x0013:** Loads the value 0x13 into the AX register, signaling that the program will switch to graphics mode 13h (320x200 resolution with 256 colors).

**int 0x10:** Calls BIOS interrupt 0x10, which is used for video services. Setting this mode prepares the screen for drawing graphics.

***main\_loop:***

***call checkPause***

***mov al, [GamePaused]***

***cmp al,1***

***je playTheGame***

**call checkPause**: Likely checks if the game is paused, updating the value of a variable called GamePaused.

**mov al, [GamePaused]**: Loads the value of the GamePaused flag into register AL. This determines whether the game is paused.

**cmp al,1**: Compares AL to 1. If GamePaused == 1, the game is paused.

**je playTheGame**: If equal (GamePaused == 1), the program jumps to the playTheGame section to handle the paused state.

***call Border***

***call clearBall ; Clear the old ball***

***call moving ; Update position***

***call drawBall ; Draw the new ball***

***call ClearPeddl***

***call movPeddle***

***call drawPeddl ; Draw both paddles***

***call DrawScore***

**call Border**: Likely draws or updates the borders of the playing area.

**call clearBall**: Clears the ball's previous position on the screen to prevent overlapping when it's redrawn at a new position.

**call moving:** Updates the ball's position based on its direction and velocity.

**call drawBall:** Draws the ball at its new position.

**call ClearPeddl**: Clears the paddles' old positions.

**call movPeddle:** Updates the positions of the paddles based on player input or AI logic.

**call drawPeddl**: Draws the paddles at their new positions.

**call DrawScore:** Updates and displays the score on the screen.

***mov cx, 0FFFFh ; Delay loop***

***delay\_loop:***

***loop delay\_loop***

**mov cx, 0FFFFh**: Initializes the CX register with a large value (0xFFFF, which is 65,535 in decimal).

**delay\_loop**: A simple loop decrements the value in CX. This acts as a basic delay mechanism.

**loop delay\_loop**: Decrements CX and repeats the loop until CX reaches zero

***inner:***

***loop inner***

***mov cx,0xffbb***

***inner1:***

***loop inner1***

***jmp main\_loop ; Repeat the loop***

**inner and inner1**: Seem to be additional nested delay loops to further slow down the execution for smoother animation. They reduce the game update speed to ensure it runs at a human-friendly pace.

**jmp main\_loop:** Jumps back to **main\_loop**, repeating the game cycle.

***playTheGame:***

***jmp main\_loop***

**playTheGame**: If the game is paused, this label directs the flow back to main loop without performing updates.

***exit:***

***mov ax, 0x4c00***

***int 0x21***

1. **Borders Function:**

Function to draw the borders of a rectangular play area on the screen in **Mode 13h (320x200, 256 colors)**. It uses BIOS interrupt 0x10 to draw individual pixels in the specified color (cyan). The borders include **vertical borders** (left and right) and **horizontal borders** (top and bottom).

**Drawing the Left Vertical Border:**

***mov cx, [BorderLeftX]***

***mov dx, [BorderLeftY]***

***DrawLeftBorder:***

***mov ah, 0Ch***

***mov al, 0x03***

***int 0x10***

***inc dx***

***mov ax, dx***

***sub ax, [BorderLeftY]***

***cmp ax, [BorderHeight]***

***jl DrawLeftBorder***

**cx** is set to BorderLeftX, which is the x-coordinate of the left vertical border.

**dx** is set to BorderLeftY, which is the y-coordinate where the border starts.

**mov ah, 0Ch**: Sets up BIOS video interrupt 0x10 for the write pixel function in graphics mode.

**mov al, 0x03:** Specifies the color cyan (color index 3 in Mode 13h's 256-color palette).

**int 0x10:** Calls the BIOS interrupt to draw a single pixel at (cx, dx) in the specified color.

**inc dx:** Moves the pixel position down by incrementing the y-coordinate.

**mov ax, dx & sub ax, [BorderLeftY]:** Calculates the vertical distance traveled so far by subtracting the starting y-coordinate.

**cmp ax, [BorderHeight]:** Compares the traveled distance with the specified height of the border (BorderHeight).

**jl DrawLeftBorder:** If the current height is less than the required border height, repeats the loop.

**Drawing the Right Vertical Border:**

***mov cx, [BorderRightX] ; Start x-coordinate***

***mov dx, [BorderRightY] ; Start y-coordinate***

***DrawRightBorder:***

***mov ah, 0Ch***

***mov al, 0x03***

***int 0x10***

***inc dx***

***mov ax, dx***

***sub ax, [BorderRightY]***

***cmp ax, [BorderHeight]***

***jl DrawRightBorder***

**This section is similar to the left border drawing code, except:**

* The x-coordinate cx is initialized to **BorderRightX**, which is the x-coordinate of the right vertical border.
* The starting y-coordinate dx is set to **BorderRightY**.

**The process follows the same logic:**

* A vertical cyan line is drawn pixel by pixel from (**BorderRightX, BorderRightY**) downward for **BorderHeight** pixels.

**Drawing the Top Horizontal Border:**

***mov cx, 02h ; Start x-coordinate***

***mov dx, [BorderTopY] ; Top border y-coordinate***

***DrawTopBorder:***

***mov ah, 0Ch***

***mov al, 0x03***

***int 0x10***

***inc cx***

***mov ax, cx***

***sub ax, 02h***

***cmp ax, 138h***

***jl DrawTopBorder***

* **cx** is initialized to 0x02, the starting x-coordinate of the top horizontal border.
* **dx** is set to BorderTopY, the fixed y-coordinate for the top border.

**Pixel Drawing Loop (DrawTopBorder)**:

* Similar to the vertical border, int 0x10 draws a single cyan pixel at (cx, dx).
* **inc cx**: Moves the pixel position **right** by incrementing the x-coordinate.
* **mov ax, cx & sub ax, 02h**:
  + Calculates the horizontal distance traveled so far by subtracting the initial x-coordinate (0x02).
* **cmp ax, 138h**:
  + Compares the traveled width with 0x138 (320 in decimal), which represents the width of the screen in Mode 13h.
* **jl DrawTopBorder**:
  + Continues the loop if the full width hasn't been reached.

**Output**:

* Draws a **horizontal cyan line** from (0x02, BorderTopY) to the right edge of the screen.

**Drawing the** **Bottom** **Horizontal** **Border:**

***mov cx, 02h***

***mov dx, [BorderBottomY]***

***DrawBottomBorder:***

***mov ah, 0Ch***

***mov al, 0x03***

***int 0x10***

***inc cx***

***mov ax, cx***

***sub ax, 02h***

***cmp ax, 138h***

***jl DrawBottomBorder***

**This section is similar to the top border drawing code, except:**

* The y-coordinate dx is initialized to BorderBottomY, the y-coordinate for the bottom border.

**The process is identical:**

* A horizontal cyan line is drawn pixel by pixel from (0x02, BorderBottomY) to the right edge of the screen.

1. **PlaySound Function:**

The PlaySound function takes a frequency (a number in AX) and uses the computer's **Programmable Interval Timer (PIT)** to tell the PC speaker how to play that sound.

***mov bx, 11931***

***div bx***

 The PC speaker doesn’t directly understand frequency. It works with something called a divisor.

 The divisor is calculated using the formula: **Divisor=1193182/Frequency ​**

* **1193182** is the "base frequency" of the PIT. (For simplicity, in the code, this is approximated as **11931** for easier calculations.)

 The instruction **div bx** divides the number in AX (the frequency) by **bx (11931**). After this, AX contains the divisor:

* Low byte (small part) goes into **BL**.
* High byte (big part) goes into **BH**.

***mov al, 0xB6***

***out 0x43, al***

***mov al, bl***

***out 0x42, al***

***mov al, bh***

***out 0x42, al***

**** The **PIT (Programmable Interval Timer)** is like a helper who controls the timing of the PC speaker.

* You talk to it using ports (**0x43** and **0x42**).

**** First, you tell the PIT how you want to set it up:

* **0xB6** tells it: *"I’ll send you a divisor for Channel 2. Use it to make a square wave sound!"*

**** Then you send the divisor:

* The **low byte** (small part of the divisor) is sent to 0x42.
* The **high byte** (big part of the divisor) is also sent to 0x42.

***in al, 0x61***

***or al, 0x03***

***out 0x61, al***

Now it’s time to **turn on the PC speaker** so it can actually make noise.

* in al, 0x61: Reads the current status of the speaker from port 0x61.
* or al, 0x03: Turns on **bits 0 and 1** of this status. (These bits control the speaker’s on/off state.)
* out 0x61, al: Sends the updated status back to the speaker to turn it on.

***mov cx, 5000***

***SoundDelay:***

***loop SoundDelay***

This part is like a **timer** that decides how long the sound will play.

* mov cx, 5000: Sets up a counter (CX) for the delay.
* loop SoundDelay: Keeps counting down until the timer runs out.

***in al, 0x61***

***and al, 0xFC ; Clear bits 0 and 1***

***out 0x61, al***

After the delay, the speaker needs to be **turned off**:

* in al, 0x61: Reads the current status of the speaker.
* and al, 0xFC: Clears **bits 0 and 1**, turning the speaker off.
* out 0x61, al: Sends the updated status back to the speaker.

1. **InputScore Function:**

Function is a routine designed to display a message prompting for input, take a single-digit key press from the user, convert it from ASCII to a numeric value, and store it in a variable called PlayerPoint.

***mov ah, 02h***

***mov bh, 00h***

***mov dh, 12***

***mov dl, 7***

***int 10h***

Moves the cursor to the specified location on the screen. In this case, it places the cursor at row 18 and column 7.

***mov ah, 09h***

***lea dx, [Goals]***

***int 21h***

Uses DOS interrupt **21h** to display a string stored at the memory address **Goals**.

**Goals** is likely a message like **"Enter your score:"**.

***sub al, 30h***

If the user presses a key for a digit (e.g., '5'), its ASCII value is stored in AL. For example:

* The ASCII value of '5' is **53 (0x35)**.

Subtracting **30h** (48 in decimal) from the ASCII value converts it to the numeric value:

* '5' (53) → **5**.

***movzx ax, al***

***mov [PlayerPoint], ax***

**movzx ax, al:** Moves the numeric value from AL to AX, filling the rest of AX with zeros (zero extension).

**mov [PlayerPoint], ax:** Stores the value in the memory location PlayerPoint, which is likely a variable used elsewhere in the program to track the player’s score.

1. **ExitScr Function:**

***exitscr:***

***mov ah, 00h ; Set video mode***

***mov al, 02h ; Choose 80x25 text mode***

***int 10h ; Apply video mode change***

***mov ah, 0Bh ; Set background color***

***mov bh, 00h ; Page number (0)***

***mov bl, 00h ; Black background***

***int 10h ; Apply background color***

***ret ; Return from function***

**Setting Video Mode (80x25 Text Mode):**

* mov ah, 00h and mov al, 02h configure the video mode to 80 characters wide by 25 rows (standard text mode).
* int 10h is a BIOS interrupt that applies the video mode.

**Setting Background Color:**

* mov ah, 0Bh sets the background color.
* mov bh, 00h refers to page 0 (default screen).
* mov bl, 00h sets the background color to black.
* int 10h applies the background color change.

1. **CheckPause Function:**

***checkPause:***

***mov ah,01h ;check if key is pressed***

***int 16h ;interupt for keyboard***

***jz continueGame***

***mov ah, 00h ;get the key press***

***int 16h***

***cmp al, 'P'***

***je togglePause***

***cmp al, 'p'***

***je togglePause***

***continueGame:***

***ret***

***togglePause:***

***mov ah, 02h ; Set cursor position***

***mov bh, 00h ; Page 0***

***mov dh, 12 ;Row***

***mov dl, 17 ;Column***

***int 10h***

***mov ah, 09h ; Display string***

***lea dx, [Paused] ; Prompt for input***

***int 21h***

***mov al, [GamePaused]***

***xor al,1 ;Toggle the pause state (0->1, 1->0)***

***mov [GamePaused],al***

***cmp byte[GamePaused], 0***

***jne exitPaused***

***call clrscr***

***exitPaused:***

***call Border***

***call drawBall***

***call drawPeddl***

***call DrawScore***

***ret***

 **Check if a Key is Pressed:**

* The function first checks if a key has been pressed using int 16h with mov ah, 01h. If no key is pressed, it jumps to continueGame, skipping the pause handling.

**Key Detection and Pause Toggle:**

* If a key is pressed, mov ah, 00h gets the key, and it is stored in the al register.
* The code then checks if the key is 'P' or 'p'. If either is pressed, the program jumps to togglePause, which handles pausing and unpausing the game.

**Display 'Paused' Message:**

* When the game is paused, the cursor is set to row 12, column 17 on the screen, and the string "Paused" (stored at the address Paused) is displayed using int 21h.

**Toggling the Pause State:**

* The game pause state is toggled using xor al, 1, which flips the value of GamePaused between 0 and 1.
* The updated state is stored back into the GamePaused variable.

**Clearing Screen (if unpaused) and Updating the Game State:**

* If the game is unpaused (GamePaused is 0), the screen is cleared with call clrscr.
* Regardless of the pause state, the game elements like the border, ball, paddles, and score are redrawn.

1. **ClearBall Function:**

***clearBall:***

***mov cx, [DotX]***

***mov dx, [DotY]***

***ClearHorizontal:***

***mov ah, 0Ch***

***mov al, 0x00***

***int 0x10***

***inc cx***

***mov ax, cx***

***sub ax, [DotX]***

***cmp ax, [DotSize]***

***jng ClearHorizontal***

***mov cx , [DotX]***

***inc dx***

***mov ax, dx***

***sub ax, [DotY]***

***cmp ax, [DotSize]***

***jng ClearHorizontal***

***ret***

 **Initial Setup:**

* The function begins by loading the DotX and DotY coordinates (the starting position of the ball) into the cx and dx registers, respectively. These represent the ball's top-left corner.

 **Clearing the Ball Horizontally:**

* The first loop (ClearHorizontal) clears the horizontal line (width) of the ball.
* The mov ah, 0Ch and mov al, 0x00 set the pixel color to black (the background color).
* The int 0x10 BIOS interrupt writes the pixel to the screen at the current position.
* The inc cx increments the x-coordinate to move to the next pixel to the right.
* The program checks if the distance moved horizontally is less than the DotSize (ball's width). If it is, the loop repeats to clear the next pixel.
* Once the entire horizontal width of the ball is cleared, the program moves to the next step.

 **Clearing the Ball Vertically:**

* After completing one row (width), the x-coordinate (cx) is reset to the initial value (DotX), and the y-coordinate (dx) is incremented to move to the next row.
* The same process is repeated for the vertical clearing by checking the vertical size (DotSize), similar to the horizontal clearing.
* The int 0x10 interrupt writes pixels at each row, and once the entire height of the ball is cleared, the loop exits.

1. **Moving Function:**

***moving:***

***mov ax, [DotSpeedX]***

***add [DotX], ax***

***cmp word[DotX], ax***

***jl GivePointtoTwo ;jumping to give point to player tw***

***mov ax, word[Width]***

***sub ax, word[DotSize]***

***sub ax, [padding]***

***cmp word[DotX], ax***

***jg GivePointtoOne***

***jmp moveVertically***

***GivePointtoOne:***

***inc word[PLeftPoint]***

***mov ax, 440***

***call PlaySound***

***call RestartCentre***

***call updateScoreOne***

***mov ax,word[PlayerPoint]***

***cmp word[PLeftPoint],ax***

***jge GameOver***

***ret***

***GivePointtoTwo:***

***inc byte[PRightPoint]***

***mov ax, 440***

***call PlaySound***

***call RestartCentre***

***call updateScoreTwo***

***mov ax,word [PlayerPoint]***

***cmp word[PRightPoint],ax***

***jge GameOver***

***ret***

***skipMovement:***

***ret***

**Updating Ball's Position:**

* The ball's horizontal movement is controlled by the variable DotSpeedX. The value of DotSpeedX is added to DotX, the x-coordinate of the ball, to update its position on the screen.
* The program compares the ball's updated x-coordinate with the speed to check if the ball has moved off the screen.

**Checking for Ball Out of Bounds (Left or Right):**

* If the ball's x-coordinate exceeds the screen width or goes off the left side, it triggers the appropriate function (GivePointtoOne for player one or GivePointtoTwo for player two) to award a point.
* The screen width is checked by subtracting the ball's size (DotSize) and padding from the width. This ensures the ball doesn't go beyond the screen boundaries.

 **Awarding Points:**

* When the ball goes out of bounds:
  + For player one, the score (PLeftPoint) is incremented, and a sound is played to indicate the point.
  + For player two, the score (PRightPoint) is incremented, and a sound is played as well.
* After scoring, the ball is reset to the center of the screen using RestartCentre, and the respective score is updated with the updateScoreOne or updateScoreTwo functions.

 **Checking for Game Over:**

* The current score is compared with the target score (PlayerPoint) to check if either player has won. If the score of either player reaches or exceeds the target, the GameOver function is triggered.

 **Skipping Movement:**

* The skipMovement label is used to skip any further movement logic if necessary, but it does not perform any specific actions in the current function and just returns.

1. **GameOver Function:**

***GameOver:***

***mov ax, word[PlayerPoint]***

***cmp word[PLeftPoint], ax***

***jnl WinPone***

***jmp WinPTwo***

***WinPone:***

***mov word[WinnerIndex], 01h***

***jmp continueGameOver***

***WinPTwo:***

***mov word[WinnerIndex], 02h***

***jmp continueGameOver***

***continueGameOver:***

***mov word[PLeftPoint], 00h***

***mov word[PRightPoint], 00h***

***call DrawOverMenu***

***call updateScoreOne***

***call updateScoreTwo***

***jmp exit***

***ret***

 **Game Over Check:**

* The game checks if either player has reached the target score (PlayerPoint), and compares it with the scores of both players (PLeftPoint for Player 1 and PRightPoint for Player 2).
* If Player 1's score is greater than or equal to the target (PlayerPoint), the program jumps to WinPone. Otherwise, it jumps to WinPTwo for Player 2 to win.

 **Winning Logic:**

* In WinPone, if Player 1 wins, the WinnerIndex is set to 01h (indicating Player 1 is the winner).
* In WinPTwo, if Player 2 wins, the WinnerIndex is set to 02h (indicating Player 2 is the winner).

 **Resetting the Game:**

* After determining the winner, the game proceeds to continueGameOver, where both players' scores are reset to zero (PLeftPoint and PRightPoint set to 00h).
* The game over menu is drawn with the DrawOverMenu function, and the updated scores are shown using updateScoreOne and updateScoreTwo.

 **Exiting the Game:**

* Finally, the program jumps to exit, which is assumed to terminate the game or return to the main menu or previous screen.

1. **DrawOverMenu Function:**

Function displays the game over screen, showing who won, and provides options for restarting the game or returning to the main menu. It waits for the user's key press and reacts based on whether they want to restart the game or go back to the main menu

***DrawOverMenu:***

***call clrscr***

***mov ah,02h***

***mov bh,00h***

***mov dh,04h***

***mov dl,10h***

***int 10h***

***mov ah,09h***

***lea dx,[TextGameOver]***

***int 21h***

***mov ah,02h***

***mov bh,00h***

***mov dh,06h***

***mov dl,10h***

***int 10h***

***call updateWin***

***mov ah,09h***

***lea dx,[GameWinner]***

***int 21h***

***;restart message***

***mov ah,02h***

***mov bh,00h***

***mov dh,08h***

***mov dl,10h***

***int 10h***

***mov ah,09h***

***lea dx,[PlayAgain]***

***int 21h***

***;Main menu message***

***mov ah,02h***

***mov bh,00h***

***mov dh,0Ah***

***mov dl,10h***

***int 10h***

***mov ah,09h***

***lea dx,[MainMenu]***

***int 21h***

***;wait for keypress***

***mov ah,00h***

***int 16h***

***;For Restarting Game***

***cmp al, 'R'***

***je Restart***

***cmp al , 'r'***

***je Restart***

***;For Goint to Main Menu of Game***

***cmp al, 'M'***

***je DrawMainMenu***

***cmp al , 'm'***

***je DrawMainMenu***

***ret***

***;***

***updateWin:***

***mov al,byte[WinnerIndex]***

***add al,0x30***

***mov byte[GameWinner+8],al***

***ret***

***Restart:***

***mov word[PLeftPoint],00h***

***mov word[PRightPoint],00h***

***call updateScoreOne***

***call updateScoreTwo***

***call start***

***ret***

 C**lear the screen:**

* The clrscr function is called to clear the screen so the game over and restart information is displayed cleanly.

 **Setting cursor positions:**

* The program moves the cursor to specific positions on the screen multiple times to display the relevant game over, winner, and restart messages at the right places.

 **Displaying text:**

* The program displays several messages:
  + **"Game Over"** at row 4, column 16.
  + **Winner information** after determining the winner.
  + **"Play Again?"** message for the user to decide.
  + **"Main Menu"** option for navigating back to the main menu.

 **Waiting for user input:**

* The program waits for a keypress. The user can press R or r to restart the game, or M or m to go to the main menu.

 **Checking user input:**

* If the user presses R or r, the game restarts by resetting the scores and calling the start function to begin a new game.
* If the user presses M or m, the program jumps to the DrawMainMenu function to navigate to the main menu.

 **Updating winner information:**

* The updateWin function updates the winner's information in the GameWinner string, based on the WinnerIndex.

 **Restarting the game:**

* In the Restart section, the scores for both players (PLeftPoint and PRightPoint) are reset to 0, and the updateScoreOne and updateScoreTwo functions update the score displays. The game restarts by calling the start function.

1. **DrawMainMenu Function:**

Function sets up and displays the main menu options for the game, allowing the user to either start a new game or exit. The program then waits for user input to choose between starting a game (S) or quitting (E).

***DrawMainMenu:***

***call clrscr ; Clear the screen***

***; Main menu Title***

***mov ah, 02h ; Set cursor position***

***mov bh, 00h ; Page 0***

***mov dh, 06h ; Row 6***

***mov dl, 08h ; Column 8***

***int 10h ; BIOS interrupt to set cursor***

***mov ah, 09h ; Display string***

***lea dx, [MainMenuTitle] ; Load "Main Menu" title text address***

***int 21h ; Display the string***

***; SinglePlayer text***

***mov ah, 02h ; Set cursor position***

***mov bh, 00h ; Page 0***

***mov dh, 08h ; Row 8***

***mov dl, 08h ; Column 8***

***int 10h ; BIOS interrupt to set cursor***

***mov ah, 09h ; Display string***

***lea dx, [StartGame] ; Load "Single Player" text address***

***int 21h ; Display the string***

***; Exit Text***

***mov ah, 02h ; Set cursor position***

***mov bh, 00h ; Page 0***

***mov dh, 0Ah ; Row 10***

***mov dl, 08h ; Column 8***

***int 10h ; BIOS interrupt to set cursor***

***mov ah, 09h***

 **Clearing the Screen:**

* call clrscr: Clears the screen before displaying the main menu.

 **Displaying Main Menu Title:**

* The cursor is moved to row 6, column 8 using BIOS interrupt (int 10h), and the title "Main Menu" is displayed using DOS interrupt (int 21h).

 **Displaying SinglePlayer Option:**

* The cursor is then moved to row 8, column 8, and the option to start a single-player game ("Single Player") is displayed.

 **Displaying Exit Option:**

* The cursor is moved to row 10, column 8, and the "Exit Game" option is displayed.

 **Waiting for User Input:**

* The program waits for the user to press a key (int 16h), then compares the key pressed with the letters S or s (for starting the game) and E or e (for exiting).

 **Handling Input:**

* If the user presses S or s, the program jumps to the Restart section to start the game.
* If the user presses E or e, the program jumps to the QuitGame section to exit the game.
* If any other key is pressed, the program will return to the caller without taking action.

1. **RestartCenter Function:**

***RestartCentre:***

***mov ax, [DotXCentre]***

***mov [DotX], ax***

***mov ax, [DotYCentre]***

***mov [DotY], ax***

***neg word [DotSpeedX]***

***neg word [DotSpeedY]***

***ret***

 **Reset the Dot's Position:**

* mov ax, [DotXCentre]: This instruction loads the x-coordinate of the center (DotXCentre) into the AX register.
* mov [DotX], ax: The current x-coordinate of the dot (DotX) is updated to the center's x-coordinate, effectively moving the dot to the center horizontally.
* Similarly, mov ax, [DotYCentre] and mov [DotY], ax update the dot's y-coordinate to the center's y-coordinate, effectively moving it to the center vertically.

 **Reverse Dot's Direction:**

* neg word [DotSpeedX]: This negates (reverses) the horizontal speed of the dot (DotSpeedX), making it move in the opposite direction.
* neg word [DotSpeedY]: Similarly, this negates the vertical speed of the dot (DotSpeedY), reversing its vertical movement.

1. **MoveVertically Function:**

***moveVertically:***

***mov ax, word[DotSpeedY] ; Load the vertical speed of the dot into AX***

***add word[DotY], ax ; Update the dot's Y-coordinate by adding the vertical speed***

***cmp word[DotY], 2 ; Check if the dot has reached the top (Y < 2)***

***jl reverseSpeedY ; If the dot is above the top boundary, reverse the Y speed***

***mov ax, word[Height] ; Load the screen height into AX***

***sub ax, word[DotSize] ; Subtract the dot's size to get the bottom boundary***

***sub ax, [padding] ; Subtract padding to get the precise bottom boundary***

***cmp word[DotY], ax ; Check if the dot has passed the bottom boundary***

***jg reverseSpeedY ; If the dot is below the bottom boundary, reverse the Y speed***

***mov ax, [DotX] ; Load the dot's X-coordinate into AX***

***add ax, [DotSize] ; Add the dot's size to get the right side of the dot***

***cmp ax, [PeddleRightX] ; Check if the dot has passed the right paddle's X boundary***

***jng checkwithLeft ; If the dot is left of the right paddle, check with the left paddle***

***mov ax, [PeddleRightX] ; Load the right paddle's X-coordinate into AX***

***add ax, [PeddleRightWidth] ; Add the paddle's width to get the right boundary of the right paddle***

***cmp [DotX], ax ; Check if the dot's X-coordinate is past the right paddle's right edge***

***jnl checkwithLeft ; If the dot is to the right of the right paddle, check with the left paddle***

***mov ax, [DotY] ; Load the dot's Y-coordinate into AX***

***add ax, [DotSize] ; Add the dot's size to check the bottom part of the dot***

***cmp ax, [PeddleRightY] ; Check if the dot is below the top of the right paddle***

***jng checkwithLeft ; If the dot is above the right paddle's Y boundary, check with the left paddle***

***mov ax, [PeddleRightY] ; Load the right paddle's Y-coordinate into AX***

***add ax, [PeddleRightHeight] ; Add the paddle's height to get the bottom boundary***

***cmp [DotY], ax ; Check if the dot is above the bottom of the right paddle***

***jnl checkwithLeft ; If the dot is below the right paddle's bottom, check with the left paddle***

***jmp reverseSpeedX ; Reverse the X speed (change direction)***

***checkwithLeft:***

***mov ax, [DotX] ; Load the dot's X-coordinate into AX***

***add ax, [DotSize] ; Add the dot's size to get the right side of the dot***

***cmp ax, [PeddleLeftX] ; Check if the dot has passed the left paddle's X boundary***

***jng ExitCollision ; If the dot is to the left of the left paddle, exit the collision check***

***mov ax, [PeddleLeftX] ; Load the left paddle's X-coordinate into AX***

***add ax, [PeddleLeftWidth] ; Add the paddle's width to get the right boundary of the left paddle***

***cmp [DotX], ax ; Check if the dot's X-coordinate is past the left paddle's right edge***

***jnl ExitCollision ; If the dot is to the right of the left paddle, exit the collision check***

***mov ax, [DotY] ; Load the dot's Y-coordinate into AX***

***add ax, [DotSize] ; Add the dot's size to check the bottom part of the dot***

***cmp ax, [PeddleLeftY] ; Check if the dot is below the top of the left paddle***

***jng ExitCollision ; If the dot is above the left paddle's Y boundary, exit the collision check***

***mov ax, [PeddleLeftY] ; Load the left paddle's Y-coordinate into AX***

***add ax, [PeddleLeftHeight] ; Add the paddle's height to get the bottom boundary of the left paddle***

***cmp [DotY], ax ; Check if the dot is above the bottom of the left paddle***

***jnl ExitCollision ; If the dot is below the left paddle's bottom, exit the collision check***

***jmp reverseSpeedX ; Reverse the X speed (change direction)***

***reverseSpeedY:***

***neg word [DotSpeedY] ; Negate the Y speed to reverse the vertical movement***

***ret***

***reverseSpeedX:***

***neg word [DotSpeedX] ; Negate the X speed to reverse the horizontal movement***

***ret***

***ExitCollision:***

***Ret***

 **Vertical Movement and Boundary Checking:**

* The function moves the dot vertically by updating the DotY position with the DotSpeedY.
* If the dot reaches the top boundary (Y < 2), or the bottom boundary (Y reaches the height of the screen minus the dot's size and padding), the vertical speed (DotSpeedY) is reversed by negating it.

 **Collision Detection with Paddles:**

* The function checks if the dot has collided with either the left or right paddle. This involves checking if the dot’s position intersects with the paddle’s position and dimensions (X and Y coordinates).
* If a collision is detected, the function reverses the horizontal speed (DotSpeedX) by negating it.

 **Reversing Movement Directions:**

* If the dot collides with a boundary or a paddle, its movement direction is reversed. This is achieved by negating the appropriate speed variable (DotSpeedY for vertical direction and DotSpeedX for horizontal direction).

 **Exit Collision:**

* If no collision occurs, the function exits and allows the game to continue with the dot's updated position.

 **Function Flow:**

* The reverseSpeedY and reverseSpeedX labels handle the reversal of the movement directions.
* The ExitCollision label exits the collision checking process if no collision occurs.

1. **DrawBall Function:**

 **Set Starting Coordinates:**

* The starting x (DotX) and y (DotY) coordinates are loaded into cx and dx, respectively. These represent the initial position where the ball will begin drawing.

 **Horizontal Loop:**

* DotHorizontal is a loop that draws the pixels of the ball horizontally.
* The loop uses the BIOS interrupt 0x10 with function 0x0C to draw a pixel at the current coordinates (cx, dx) where cx is the x-coordinate and dx is the y-coordinate.
* The pixel color is set to white (0x0f).
* The x-coordinate (cx) is incremented after each pixel is drawn to move the drawing position horizontally.
* The difference between the current x-coordinate and the starting x-coordinate ([DotX]) is checked. If it is less than the size of the dot ([DotSize]), it continues drawing horizontally.

 **Vertical Loop:**

* Once a full horizontal line of pixels (equal to the dot's size) is drawn, the x-coordinate (cx) is reset to the initial starting x-coordinate, and the y-coordinate (dx) is incremented to move down to the next row.
* The same logic as the horizontal loop is applied vertically. The function continues to draw horizontally and moves to the next line until the vertical size of the dot ([DotSize]) is reached.

 **Exit Condition:**

* The loop continues drawing until both horizontal and vertical drawing reaches the specified size of the dot. Once the ball is drawn, the function ends.

1. **ClearPeddle Function:**
2. **Left Paddle Erasure:**
   * The function starts by loading the x and y coordinates of the left paddle into registers cx and dx.
   * It enters the PeddlLeftHorizontal1 loop, which repeatedly draws a black pixel (0x00) at the current coordinates (cx, dx) using BIOS interrupt 0x10, function 0x0C.
   * The x-coordinate (cx) is incremented, and the distance from the starting x-coordinate of the left paddle is calculated. If the distance is less than the width of the paddle (PeddleLeftWidth), the function continues to erase the paddle's pixels horizontally.
   * Once the end of the width is reached, the x-coordinate is reset to the left paddle's starting x-coordinate, and the y-coordinate (dx) is incremented to move to the next row.
   * The function then compares the distance from the starting y-coordinate (PeddleLeftY) to the height of the paddle (PeddleLeftHeight). If the distance is less than the paddle's height, the loop continues to erase the next row of pixels.
   * This process continues until the entire left paddle is erased.
3. **Right Paddle Erasure:**
   * After clearing the left paddle, the function does the same for the right paddle.
   * It loads the x and y coordinates of the right paddle into cx and dx, then enters the PeddlRightHorizontal1 loop.
   * It follows the same procedure as the left paddle, erasing the pixels of the right paddle one by one until the whole paddle is erased.

**Purpose of the Function:**

* The purpose of ClearPeddl is to clear or erase the paddles from the screen when they move or when the game state is reset, such as at the start of a new round or after a collision.
* It ensures that the paddles are properly removed before being redrawn in their new positions.

1. **DrawPeddle Function:**
2. **Drawing the Left Paddle:**
   * The function begins by setting up the starting coordinates (PeddleLeftX and PeddleLeftY) for the left paddle.
   * The loop PeddlLeftHorizontal draws one pixel at a time horizontally across the paddle.
   * It uses BIOS interrupt 0x10 (function 0x0C) to plot the pixels at the current coordinates (cx, dx). The color of the left paddle is set to 0x05 (which could correspond to a specific color in the palette, like light green or white).
   * The x-coordinate is incremented (inc cx) to move to the next pixel horizontally. Once the width of the paddle (PeddleLeftWidth) is reached, it moves to the next row by incrementing the y-coordinate (inc dx).
   * This process continues until the entire height (PeddleLeftHeight) of the paddle is drawn.
3. **Drawing the Right Paddle:**
   * After finishing the left paddle, the function does the same for the right paddle.
   * The right paddle's starting coordinates (PeddleRightX and PeddleRightY) are loaded into cx and dx.
   * The color for the right paddle is set to 0x06 (which could correspond to another shade, such as light gray or another color).
   * The loop PeddlRightHorizontal works the same way as for the left paddle, drawing each pixel horizontally and then moving to the next row until the entire paddle is drawn.

**Important Notes:**

* **BIOS Interrupt (0x10):** The function uses BIOS interrupt 0x10 to interact with the screen. Specifically, function 0x0C is used to write a single pixel to the screen, which is why we set ah = 0x0C and al to the desired color.
* **Coordinates Management:**
  + The code uses cx for the x-coordinate and dx for the y-coordinate of the paddle.
  + It uses PeddleLeftX, PeddleLeftY, PeddleRightX, and PeddleRightY to store the coordinates of the left and right paddles, respectively.
* **Drawing Logic:**
  + The horizontal loop (PeddlLeftHorizontal and PeddlRightHorizontal) increments the x-coordinate (cx), checks if it has reached the paddle's width, and then moves to the next row by incrementing the y-coordinate (dx).
  + This process ensures that the paddle is drawn as a rectangular shape.

1. **MovPeddle Function:**

The movPeddle function handles the movement of the left and right paddles in response to key presses. The function checks for specific key presses and adjusts the paddles' positions accordingly, while also ensuring that the paddles stay within the game screen boundaries.

Here's a breakdown of the logic:

**Left Paddle Movement:**

* The function first checks if any key is pressed using BIOS interrupt 0x16, function 0x01. If no key is pressed, it jumps to handle the right paddle's movement (RightPeddleMovement).
* **Checking for w or W:**
  + If the w key (ASCII value 77h) or W key (ASCII value 57h) is pressed, the left paddle moves **up**. The PeddleVelocity value is subtracted from the PeddleLeftY coordinate (i.e., moving the paddle up).
  + The position is then checked to ensure the paddle doesn’t go above the screen boundary, and if it does, it is fixed to the top boundary.
* **Checking for s or S:**
  + If the s key (ASCII value 73h) or S key (ASCII value 53h) is pressed, the left paddle moves **down**. The PeddleVelocity value is added to the PeddleLeftY coordinate (i.e., moving the paddle down).
  + The position is checked to ensure the paddle doesn’t go below the bottom boundary. If it does, it is fixed to the bottom boundary.

**Right Paddle Movement:**

* After handling the left paddle, the function moves to check for the right paddle’s movement.
* **Checking for Arrow Keys (up or down):**
  + The function checks the ah register for key codes 48h (up arrow key) and 50h (down arrow key) to move the right paddle.
  + The PeddleVelocity value is used to adjust the y-coordinate (PeddleRightY) of the right paddle, moving it either up or down.
  + Similarly, boundary checks are done to ensure the right paddle stays within the screen's vertical limits.

**Movement and Boundary Checks:**

* **Up Movement:**
  + The y-coordinate is decremented to move the paddle up. It is then compared with the padding value to ensure the paddle doesn’t go past the top boundary.
* **Down Movement:**
  + The y-coordinate is incremented to move the paddle down. It is then compared with the screen’s height (Height) minus the paddle's height (PeddleRightHeight or PeddleLeftHeight) to ensure the paddle doesn’t go past the bottom boundary.

**Edge Handling:**

* If the paddle moves beyond the boundaries (either top or bottom), the code uses the fixLeftPeddleTop and fixLeftPeddleBottom (for the left paddle) or fixRightPeddleTop and fixRightPeddleBottom (for the right paddle) subroutines to reset the paddle's position within the screen limits.

**Exit Condition:**

* If no movement keys are pressed for either paddle, the function exits, ending the movement logic for both paddles.

**Key Logic Summary:**

* **Left Paddle Movement**:
  + w or W: Move up (subtract from PeddleLeftY).
  + s or S: Move down (add to PeddleLeftY).
* **Right Paddle Movement**:
  + Arrow Up (48h): Move up (subtract from PeddleRightY).
  + Arrow Down (50h): Move down (add to PeddleRightY).
* Boundary checks are applied to both paddles to ensure they stay within the screen limits

1. **DrawScore Function:**

The DrawScore function in your assembly code is responsible for drawing the score of the left and right paddles on the screen. The process involves positioning the cursor at a specific location on the screen and printing the score of each player (left and right paddles). Below is an explanation of the steps:

**Step-by-Step Breakdown:**

**1. Draw Left Paddle Score:**

* **Position the cursor:**
  + mov ah,02h: Set the cursor position using BIOS interrupt 0x10, function 0x02.
  + mov bh,00h: Specifies the page number (0 for the primary screen).
  + mov dh,02h: The row (Y-coordinate) where the cursor will be positioned (2nd row).
  + mov dl,06h: The column (X-coordinate) where the cursor will be positioned (6th column).
  + int 10h: Call the BIOS interrupt to set the cursor position.
* **Display Player One's Score:**
  + mov ah,09h: Function 0x09 of BIOS interrupt 0x21 to print a string.
  + lea dx,[ShowPlayerOne]: Load the address of the string (score for player one) into the dx register.
  + int 21h: Call DOS interrupt 0x21 to print the score of player one at the specified cursor location.

**2. Draw Right Paddle Score:**

* **Position the cursor:**
  + mov ah,02h: Set the cursor position using BIOS interrupt 0x10, function 0x02.
  + mov bh,00h: Specifies the page number (0 for the primary screen).
  + mov dh,02h: The row (Y-coordinate) where the cursor will be positioned (2nd row).
  + mov dl,1Fh: The column (X-coordinate) where the cursor will be positioned (31st column).
  + int 10h: Call the BIOS interrupt to set the cursor position.
* **Display Player Two's Score:**
  + mov ah,09h: Function 0x09 of BIOS interrupt 0x21 to print a string.
  + lea dx,[ShowPlayerTwo]: Load the address of the string (score for player two) into the dx register.
  + int 21h: Call DOS interrupt 0x21 to print the score of player two at the specified cursor location.

1. **UpdateScore Function:**

The updateScoreOne and updateScoreTwo subroutines in your assembly code are responsible for updating the scores of Player 1 and Player 2, respectively.

***updateScoreOne:***

***xor ax, ax ; Clear the AX register (set it to 0)***

***mov al, [PLeftPoint] ; Move the value of Player 1's score (PLeftPoint) into AL (lower byte of AX)***

***add al, 30h ; Convert the numeric score into its ASCII character equivalent by adding 30h ('0' in ASCII)***

***mov [ShowPlayerOne], al ; Store the updated score (as an ASCII character) in the memory location ShowPlayerOne***

***ret ; Return from the subroutine***

 The xor ax, ax instruction clears the AX register, ensuring it starts from 0.

 mov al, [PLeftPoint] loads the score of Player 1 from the memory address of PLeftPoint into the AL register.

 The add al, 30h instruction converts the score into its ASCII equivalent (since ASCII codes for digits start from 30h).

 mov [ShowPlayerOne], al stores the updated score as an ASCII character in the memory location ShowPlayerOne.

 Finally, ret returns from the subroutine.

***updateScoreTwo:***

***xor ax, ax ; Clear the AX register (set it to 0)***

***mov al, [PRightPoint] ; Move the value of Player 2's score (PRightPoint) into AL***

***add al, 30h ; Convert the numeric score into its ASCII character equivalent by adding 30h ('0' in ASCII)***

***mov [ShowPlayerTwo], al ; Store the updated score (as an ASCII character) in the memory location ShowPlayerTwo***

***ret ; Return from the subroutine***

 Similar to updateScoreOne, xor ax, ax clears the AX register.

 mov al, [PRightPoint] loads the score of Player 2 from the memory address of PRightPoint into the AL register.

 The add al, 30h instruction converts the score into its ASCII character equivalent.

 mov [ShowPlayerTwo], al stores the updated score as an ASCII character in the memory location ShowPlayerTwo.

 Finally, ret returns from the subroutine

**Interrupts Used:**

**1. BIOS Interrupt 0x10 (Video Services)**

This interrupt provides various video functions, such as setting video modes, displaying characters, and drawing pixels. It is used extensively in the code for screen operations.

* **Function 0x00**: Set video mode
  + Used to set the display mode of the screen. In the code, int 0x10 with ah=0x00 sets the screen to a specific mode (al=0x13), which is VGA 320x200 with 256 colors.
* **Function 0x0B**: Set background and text colors
  + Used to set the color attributes of the video page. In the code, this is used to set the background and text color.
* **Function 0x02**: Set cursor position
  + Moves the cursor to a specific row and column on the screen. It is used to position the cursor before printing text messages or drawing on the screen.
* **Function 0x0C**: Write a pixel to the screen
  + Draws a pixel at a specific (x, y) coordinate. This function is used to draw the ball, paddles, borders, and text on the screen. The color of the pixel is specified by the al register, and the coordinates are passed through the cx and dx registers.

**2. DOS Interrupt 0x21 (DOS Services)**

DOS interrupts provide a wide range of services for file management, input/output, and system control. The relevant functions used in this code are related to keyboard input and displaying strings.

* **Function 0x09**: Display a string
  + This function is used to print a null-terminated string to the screen. In the code, it is used to display prompts like "Press S to start the game" and "Game Over."
* **Function 0x01**: Keyboard input (single character)
  + This function waits for a key press and returns the character typed by the user in the al register. It is used to get input from the user (e.g., to check if the user presses 'P' to pause the game).
* **Function 0x00**: Program termination
  + Used to terminate the program. It is called when the game ends or when exiting to DOS.

**3. BIOS Interrupt 0x16 (Keyboard Services)**

This interrupt provides keyboard input functions.

* **Function 0x01**: Check for key press
  + This function is used to check whether a key has been pressed. It does not wait for input but immediately returns. In the code, it is used in the checkPause function to determine if the user has pressed a key (for pausing the game).
* **Function 0x00**: Read a key press
  + This function waits for the user to press a key and returns the ASCII value of the key in the al register. It is used in the code to read the key pressed by the player (e.g., to pause the game when the 'P' key is pressed).

**4. Port I/O Instructions for Sound (via PIT)**

* **out 0x43 and out 0x42**: These instructions send data to the Programmable Interval Timer (PIT), which controls the timing of the sound output through the PC speaker.
  + The code uses the PIT to generate a square wave sound at a specific frequency. The divisor is calculated from the input frequency (using the value 11931 for the base PIT frequency). The frequency is then divided by this base frequency to generate the correct output.
  + **out 0x43, al**: Sends the command to the PIT to select channel 2 for sound generation.
  + **out 0x42, al**: Sends the low byte of the divisor to the PIT.
  + **out 0x42, ah**: Sends the high byte of the divisor to the PIT.
* **in al, 0x61 and out 0x61**: These instructions interact with the PC speaker control register.
  + **in al, 0x61**: Reads the current state of the PC speaker.
  + **or al, 0x03**: Sets the speaker enable bits.
  + **out 0x61, al**: Writes the updated state to the PC speaker register, enabling the speaker.

**5. Interrupt 0x13 (Disk Services)**

Although not explicitly used in the code, BIOS interrupt 0x13 is typically used for disk services like reading and writing to the disk. However, this interrupt does not appear to be utilized in this code snippet.

**Summary of Interrupt Usage:**

* **0x10** (Video services): Used for screen operations (setting video mode, drawing pixels, displaying text).
* **0x21** (DOS services): Used for displaying strings and reading keyboard input.
* **0x16** (Keyboard services): Used for checking and reading key presses.
* **Port I/O (0x43, 0x42, 0x61)**: Used for generating sound via the PC speaker.