EENTON WALI GAME

COAL LAB PROJECT

ALI IBRAHIM
24F-3009
AZKA FAISAL
24F-3068

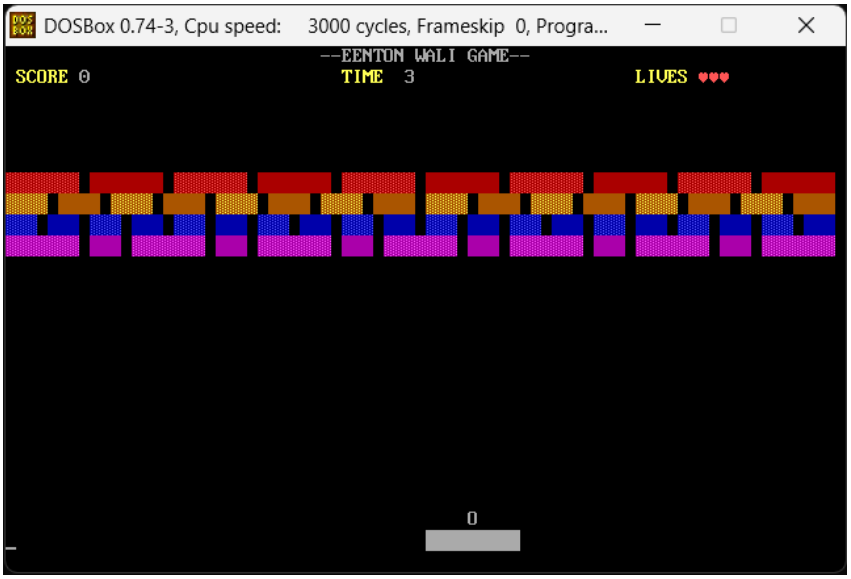
Contents

EENTON WALI GAME : Technical Documentation

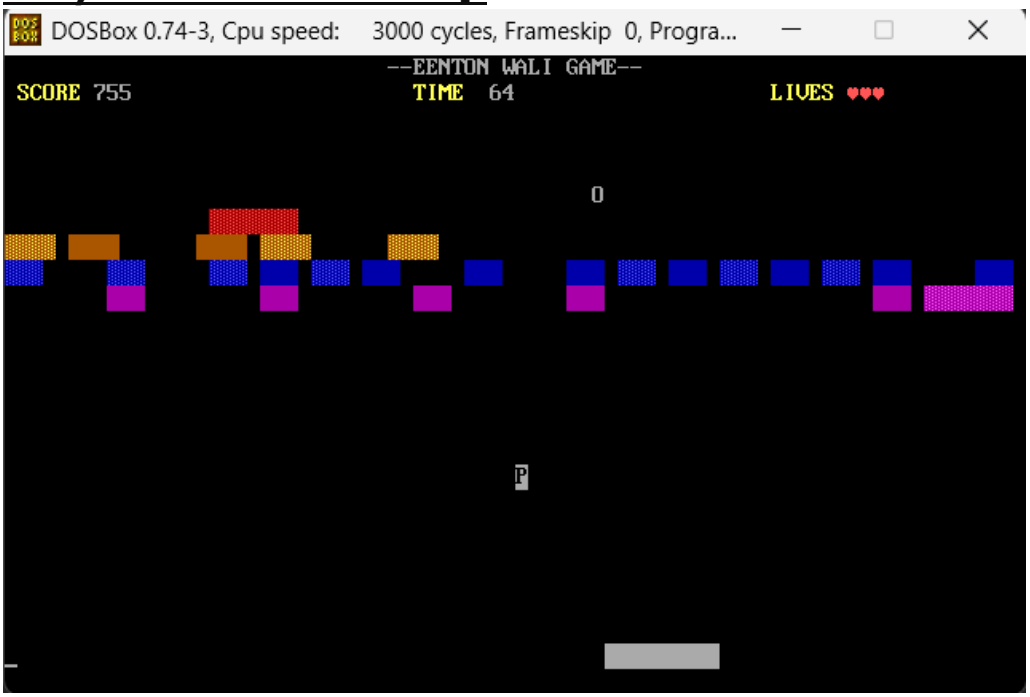
1. Gameplay Screenshots

1.1) The Main Menu

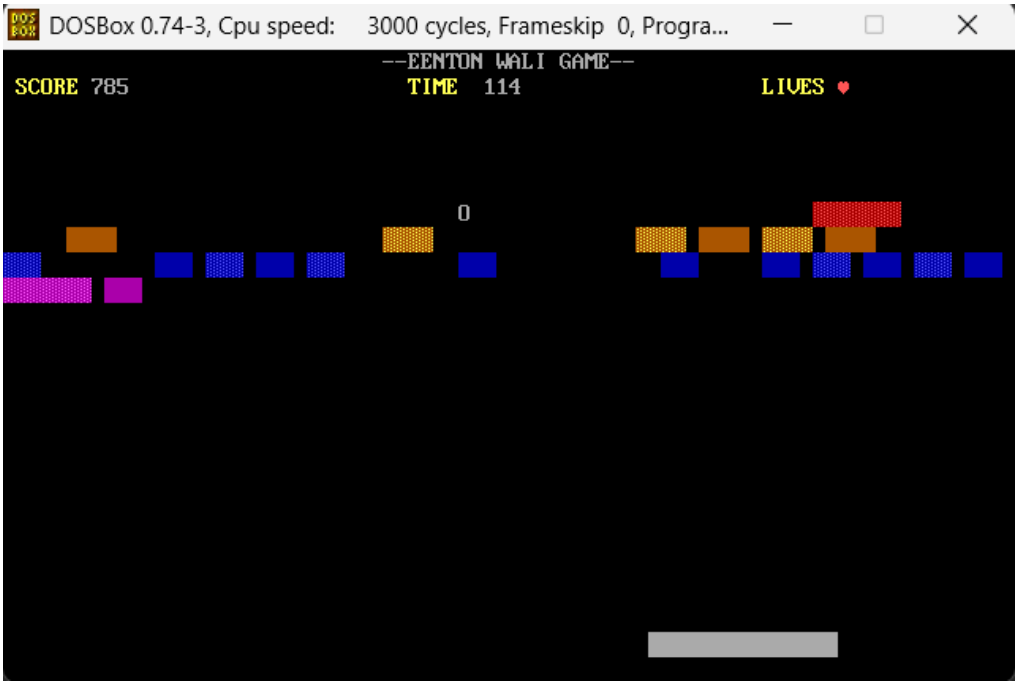
1.2) Fresh Start – Level 1



1.3) A The Power-up



1.3) B The Big Paddle State

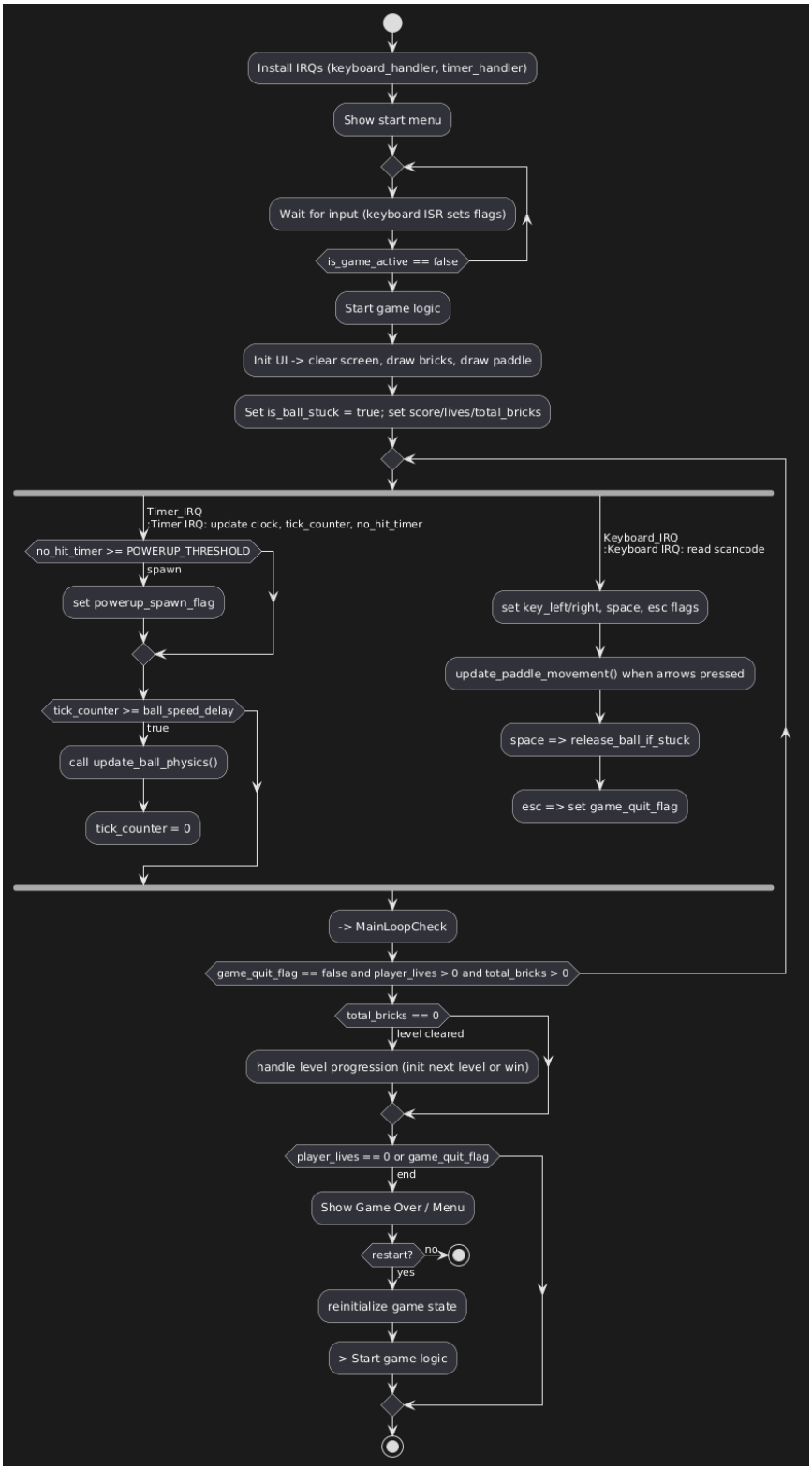


2. Flowcharts of Logic

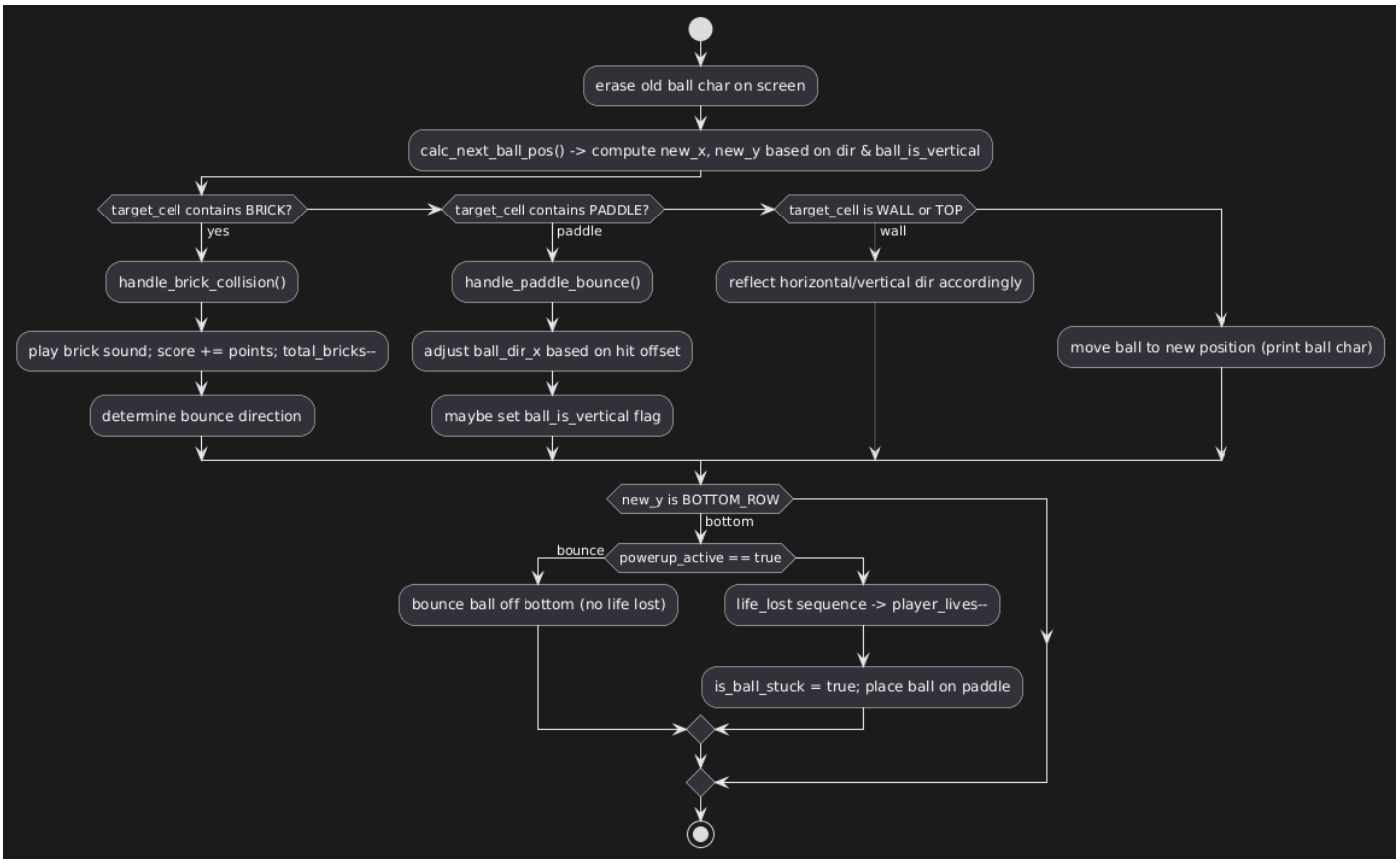
The game operates on a **Game Loop** architecture. Unlike linear programs, this system continuously cycles through input detection, logic updates, and rendering until a termination condition (Win, Loss, or Quit) is met.

2.1) MAIN FLOWCHART

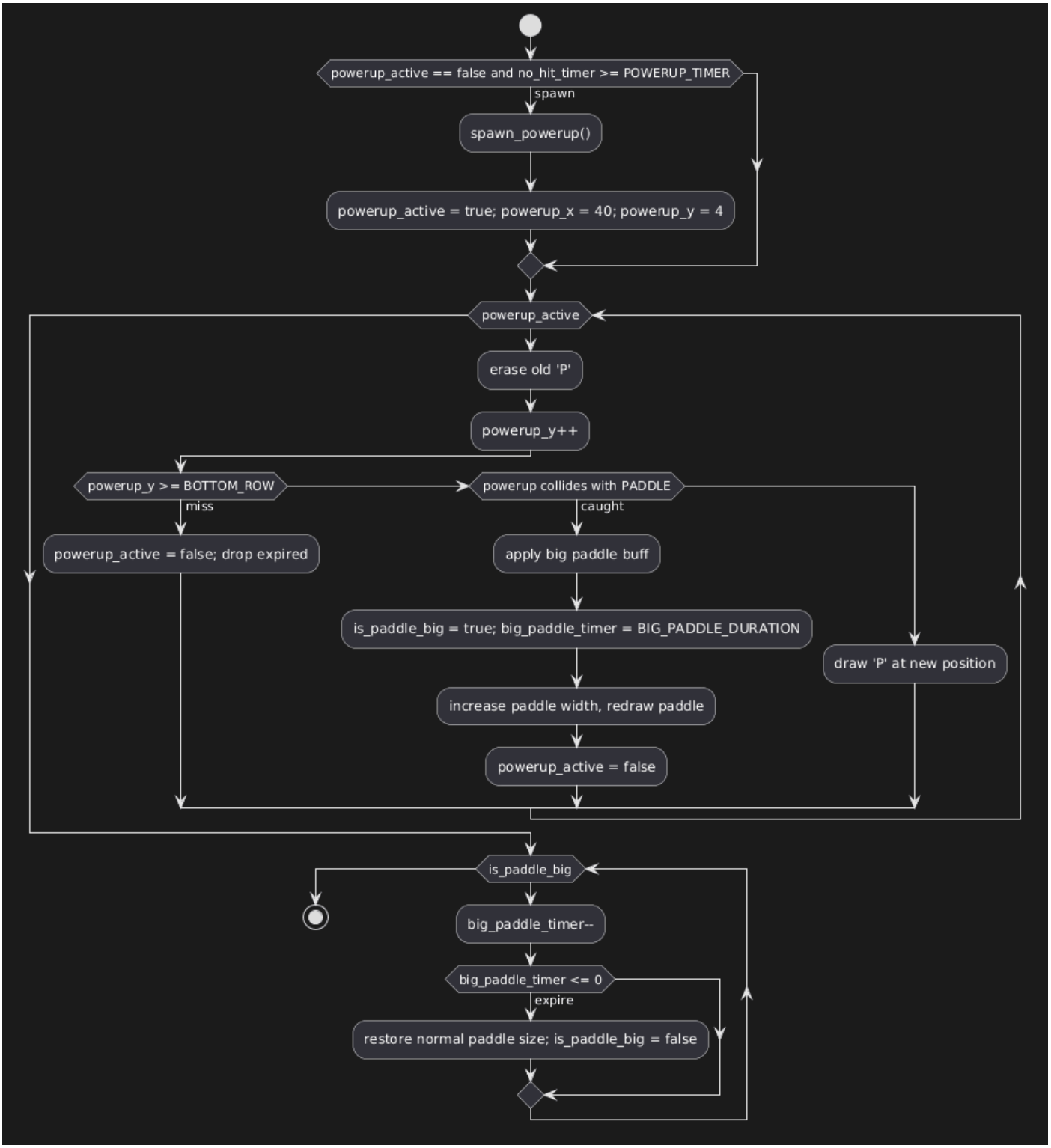
This is the main flow chart of EENTON WALI GAME showing the broad overview of the label/function calls and logic conditions.



2.2) BALL PHYSICS LOGIC FLOW:



2.3) Power-up Logic Flow



2.4Text-Based Flow Representation:

Initialization (main)

1. Save original Interrupt Vectors (IVT) for INT 9h and INT 8h.
2. Hook custom ISRs (Interrupt Service Routines).
3. Initialize Video Memory (0xB800).
4. Display **Start Menu**.

Menu Loop

1. Wait for Input.
2. **ESC:** Jump to Exit.
3. **ENTER:** Jump to start_game_logic.

Game Loop (Main)

1. **Check State:** Is Game Over flag set? Is Player Lives == 0?
2. **Check Win:** Is total_bricks == 0?
   If Yes: Trigger Level 2 or Win Screen.
3. **Input Handling:**
   - Read flags set by Keyboard ISR.
   - Move Paddle Left/Right (Update Memory Pointer).
4. **Physics (Throttled by Timer):**
   - Update Ball Position.
   - Check Wall Collisions.
   - Check Paddle/Brick Collisions.
5. **Rendering:**
   - Draw Paddle.
   - Draw Ball.
   - Update UI (Score, Lives, Time).

Termination (exit_to_dos)

1. Restore original IVT entries (Crucial to prevent DOS crash).
2. Return control to OS (INT 21h, 4Ch).

3. Interrupt Usage

This project utilizes Hardware Interrupts to handle asynchronous events (Input and Timing) without freezing the game logic.

A. Keyboard Interrupt (INT 9h)

We hook the hardware keyboard interrupt to achieve smooth, non-blocking movement. Standard DOS interrupts (INT 16h) pause the CPU while waiting for a key; our custom handler allows the game to keep running while keys are held down.

Technical Implementation:

- **Port 0x60:** The ISR reads the raw Scancode from the Keyboard Controller at I/O Port 0x60.
- **Scancode Logic:**

1. 0x4B (Left Arrow) & 0x4D (Right Arrow): These set memory flags key_left_pressed and key_right_pressed.
2. 0x01 (ESC): Sets game_quit_flag.
3. 0x1C (Enter): Sets is_game_active.
4. 0x26 (L): Trigger Cheat (Clear Level).

- **The "Flag" Pattern:** The ISR **does not** move the paddle directly. It only sets a flag (1 or 0). The Main Loop reads this flag to update the paddle position. This decouples the interrupt rate from the game's frame rate.
- **EOI:** We send 0x20 to Port 0x20 (PIC) to signal the End of Interrupt.

B. Timer Interrupt (INT 8h)

The Programmable Interval Timer (PIT) fires IRQ0 approximately **18.2 times per second**. We use this as the heartbeat of the game physics.

- **Technical Implementation:**
  1. **Global Clock:** Increments clock_ticks. Every 18 ticks, clock_seconds is incremented for the UI timer.
  2. **Physics Throttling:** To control game speed, we use a tick_counter.
     - ball_speed_delay is set to **2** (Normal) or **1** (Fast/Level 2).
     - The physics engine (update_ball_physics) is only called when tick_counter matches ball_speed_delay.
  3. **Powerup Logic:** The timer increments no_hit_timer. If this reaches 180 (approx 10 seconds without a paddle hit), it triggers the spawn_powerup routine.

4. Scoring and Gameplay Mechanics

A. Reflection Logic (Bitwise Physics)

The ball's direction is stored in ball_dir_x and ball_dir_y (0 or 1).

Standard Reflection:

When a collision occurs (e.g., hitting the ceiling or a brick), we simply **flip** the direction bit using the XOR instruction.

```
xor byte[ball_dir_y], 1  ; If 0 (Up), becomes 1 (Down). If 1, becomes 0.
```

This is extremely efficient (1 CPU cycle) compared to arithmetic negation.

The "Sweet Spot" (90-Degree Bounce):

To add skill, the game calculates where the ball hit the paddle.
Math: ABS(Ball_Next_Pos - Paddle_Center_Mem).
Logic: If the difference is less than **4 bytes** (2 characters), it means the ball hit the exact center.
Effect: The ball_is_vertical flag is set. In the movement calculation, if this flag is active, the X-axis update is skipped entirely, causing the ball to shoot straight up.

B. Brick Scoring System

The scoring engine determines points based on the **Memory Offset (SI)** of the brick in the bricks_start_loc array. Instead of storing a separate "value" for each brick, we infer value from its position in the array.

Rows & Points:

- Indices 0-19 (Row 1): Topmost row. 50 Points.
- Indices 20-51 (Row 2): High row. 20 Points.
- Indices 52-91 (Row 3): Mid row. 10 Points.
- Indices 92+ (Row 4): Low row. 5 Points.

C. Paddle Mechanics (Pointer Arithmetic)

The paddle is not an object, but a range of Video Memory addresses painted Grey (0x70).

Movement:

- Right: add word[paddle_center_mem], 8 (Moves 4 characters right).
- Left: sub word[paddle_center_mem], 8 (Moves 4 characters left).

Wall Clamping:

Before moving, the code compares the current address against wall_left_limit (3680) and wall_right_limit (3822). If the move would exceed these bounds, the address is snapped to the limit. This prevents the paddle from wrapping around the screen memory buffer.

Dynamic Expansion (Powerup):

When the powerup is caught:

- paddle_width_chars changes from 9 to 15.
- wall_right_limit is decreased to compensate for the wider paddle (preventing wall clipping).
- A timer (big_paddle_timer) counts down 270 ticks (15 seconds) before reverting these values.