CAIRO UNIVERSITY FACULTY OF ENGINNERING COMPUTER ENGINNERING DEPARTMENT

TEAM 2

Mostafa Muhammed	91240765
Muslim Ahmed	91241062
Ahmed Maged	91240928
Mariam Muhammed	91240747
Mariam Sameh	91240739
Anne Omer	91241299
Alaa Tarek	91240162
Habiba Mahmoud	91240261

CMPG103-25 | MICROPROCESSORS SYSTEMS

Table of Contents

Teamwork workload

GAME	INPUT	LOGIC	DRAWING	TESTING
TIC-TAC-TOE	Mariam Muhammed Mariam Sameh	Anne Omer Alaa Tarek Habiba Mahmoud	Ahmed Maged Muslim Ahmed Mostafa Muhammed	Mariam Sameh
PONG	Anne Omer	Alaa Tarek Mariam Muhammed Mariam Sameh	Ahmed Maged Muslim Ahmed	Alaa Tarek
Swords Bound	Alaa Tarek Mariam Sameh	Ahmed Maged Muslim Ahmed Mostafa Muhammed	Anne Omer Habiba Mahmoud Mariam Muhammed	Muslim Ahmed Anne Omer

Notes

- **All images** are edited by Mostafa Muhammed, Ahmed Maged, and Muslim Ahmed
- **Playing vs Computer** in TIC-TAC-TOE is done by Habiba Mahmoud
- **Hardware assembling** is done by Ahmed Maged, Muslim Ahmed, and Mostafa Muhammed
- **Code assembling** is done by Mostafa Muhammed
- **Menu** is done by Anne Omer, and Mostafa Muhammed
- **Documentation report** is written by Habiba Mahmoud
- **Code review** is done by Mostafa Muhammed, Muslim Ahmed, Mariam Muhammed, Mariam Sameh, Anne Omer, Alaa Tarek, Ahmed Maged

Introduction to our project

As computer engineering students, and as part of the curriculum “Introduction to Microprocessors and microcontrollers”, we implemented this project which involves practical application using “**ARM assembly**” and some Hardware components.

We used **Keil**, **Proteus** for simulation, and **python** for generating data files.

Hardware Components

STM32F401RCT6, used in the **Black Pill development board**
TFT 3.5-inch ili9486 (320 x 480)
ST-link v2 (debugger)
Buttons
Wires and a board

Brief explanation to STM32

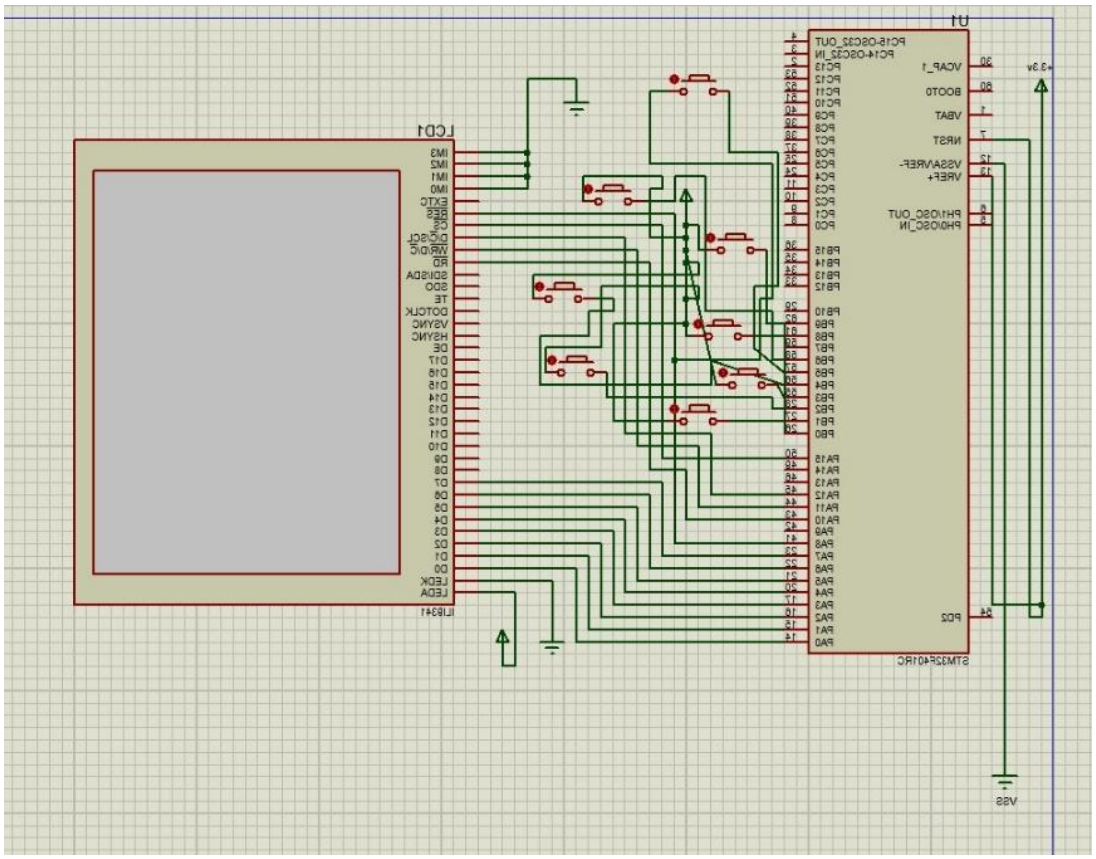
The **STM32F401RCT6**, commonly used in the **Black Pill development board**, is a powerful 32-bit microcontroller made by **STMicroelectronics**, based on the **ARM Cortex-M4** core. It’s widely used by hobbyists and professionals alike for embedded systems, robotics, IoT, and real-time control projects.

Feature	Description
Core	ARM Cortex-M4 (32-bit)
Flash memory	256 KB
Operating Voltage	3.3 V

Purpose and Goals

- Understanding Low-Level Hardware Interaction
- Master ARM Assembly Programming
- Design and Implement Simple Games
- Exploring Peripheral Features of the STM32F401RCT6
- Build a Foundation for Future Embedded Systems Projects

Hardware Connections Simulation



The STM32F401RCT6 microcontroller handles

- Logic processing
- Input reading from switches
- Driving the TFT LCD display

The ILI9341 TFT LCD display connections

- Data lines (D0–D7) are connected to STM32 GPIOs.
- Control pins include CS, RESET, WR, RD, DC.
- Uses parallel communication, 8-bit data transfer.

Multiple push buttons are connected to GPIO pins

- Each button is connected to an input pin and ground on one side, connected to VCC on the other side.
- Used as input controls for games and menu selection.

Project Description and Code Structure

We have implemented three games, the first one is **"TIC-TAC-TOE"**, the second one is **"PONG"**, and the third is**" Swords Bound**".

Initialization and Helper Functions

We have initially written the code for defining the pins and configurations for the screen and the microcontroller, along with defining the variables used in the games, then the functions responsible for initializing and drawing on the screen, followed by some functions that deal with the debouncing issue.

The constants we have used are represented by some constants specific to each game, the colors, some coordinates (used in drawing), the addresses specific to the registers and pins, and the delay intervals used.

__main Purpose in Project

This is the entry point of the program. When the microcontroller starts executing after reset or power-on, it enters this function to initialize the system and draw the initial screen.

This function sets up a black rectangle covering the full screen (from coordinates 0,0 to 320,480) by loading the dimensions and color into registers, then calling the **TFT_DrawRect** function to fill the area to effectively clear or initialize the display background.

STM_Configure_Ports

In this routine, the clocks for GPIOA and GPIOB are first enabled to activate the ports. GPIOA is then set as high-speed output to support fast data transfer to the TFT display, while GPIOB is configured as high-speed input for reading button states. Pull-down resistors are applied to GPIOB to ensure stable logic levels when buttons are unpressed. Finally, the TFT display is initialized, allowing the microcontroller to communicate with the screen and respond to user input effectively.

MAIN LOOPS

This piece of code defines the main control flow of the project's interface and game logic. It starts by drawing the main menu screen with two icons representing the available games (XO and Pong) as **Swords Bound** isn't added to the menu due to insufficient memory and displays a menu pointer that users can move using input buttons. Based on the user's selection, it navigates to either the XO or Pong game. The XO menu allows the player to choose between a two-player mode or playing against the AI, and each mode initializes the game state, clears the screen, and enters its respective game loop (**XO_MainLoop** or **XO_AIMainLoop**). Similarly, if Pong is selected, the screen is cleared, and game variables (paddle positions and ball direction) are initialized before entering an infinite loop (**PONG_MainLoop**) to handle input, logic, and updates. Each part uses drawing routines and conditional checks to manage interactions, forming the core loop structure of the project.

MENU

This part of the code is responsible for reading button input and showing a pointer on the screen to help the user select a game. The **MENU_INPUT** section checks which button is pressed by reading from the GPIOB input pins. To avoid errors from noise or bouncing, it reads the input several times to make sure the button press is stable. Depending on which button is pressed, it updates a variable (R12) to know if the user picked the XO game, the Pong game, or wants to go back to the main menu. After confirming the button has been released, the program continues. The **DRAW_MENU_POINTER** part shows a red box around the selected game icon on the screen (XO on the left or Pong on the right) and removes the previous pointer by drawing a black box. This gives the user a clear visual of what game is currently selected.

TFT Code

- *TFT_WriteCommand*

This function sends a command to the TFT screen. It starts by lowering the chip select (CS) pin to start communication. Then it sets the data/command (DC) pin low to say, “this is a command.” It clears the data lines, adds the command from register R0, and sends it. It uses the WR (write) pin to tell the screen to accept the data. Finally, it raises the CS pin to end the command.

- *TFT_WriteData*

Like the command function, but this one sends regular data instead of a command. The main difference is it sets the DC pin **high** to say, “this is data.” Then it sends the 8-bit value in R0 using the same WR pulse technique and ends with raising the CS pin.

- *TFT_Init*

This function prepares the TFT screen to work. First, it performs a **hardware reset** by toggling the reset (RST) pin: high → low → high. Then it does a **software reset** using the command sequence. It sends several commands for contrast (VCOM), pixel format (16-bit RGB), and screen direction. It also wakes up the display from sleep mode and turns it on. At the end, it turns off color inversion to show colors normally.

- *Address_Set*

This function sets the area (rectangle) on the screen where pixels will be drawn. It takes starting and ending X and Y coordinates. It sends a command for setting the column range (X1 to X2), then sends the page (row) range (Y1 to Y2). These values are split into high and low bytes and sent as data.

TIC-TAC-TOE GAME

The program uses R9, R10, and R11 registers and variables (**XO_Player1** & **XO_Player2** & **XO_Grid**) to store the state of the game board and the players' moves. Specifically, R9 holds the grid status, R10 stores Player X's moves, and R11 stores Player O's moves. The **Current_Pointer** variable indicates the latest selected cell on the grid.

Input Functions

Starting with **XO input** function, the function starts by saving all the important registers to make sure their values don't get lost during the process. Then, it reads the current position of the pointer in the XO game board from memory and stores it in register R12. It also prepares a mask to help isolate the position part of the data (ignoring the "play" bit). After that, it reads the input from the GPIOB port, where the buttons are connected, and saves it into R1.

To make sure the button press is real (not noise or a quick glitch), the code performs **hardware debouncing**. It reads the button multiple times. If the button state stays the same for all these checks, then it accepts it as a valid press. If the state changes at any time, it resets the counter and starts checking again.

Once a stable button press is detected, the program checks to ensure that only one button is pressed at a time. It does this by counting how many bits are set (how many buttons are being pressed). If the number is not exactly one, the program ignores the input and starts over, going back to the top. If the "Back" button is pressed, the program waits for the button to be released (again using debounce logic), saves the current pointer position, and jumps back to the XO menu screen.

The program checks each direction button one by one (UP – DOWN – RIGHT – LEFT). For each of these, it checks if the movement is valid (not moving off the board), and if so, it updates the position. If the "Play" button is pressed, it sets a specific bit (bit 9, called the "A bit") in the current position data. This bit tells the system that a move has been made on the current cell. The updated position is saved back into memory. If no valid input was detected, or if the movement would go outside the game board, the system keeps the old position and starts over, waiting for the next input.

At several places in the code, there's a **Wait_Input_Finish** function. It waits until the user releases the button and adds a short delay to make sure the button is fully released, and the system is stable before accepting the next input. This prevents repeated actions from being a single press.

After all the input checking and actions are complete, the function restores all the register values saved and returns to the rest of the program.

Drawing Functions

➢ XO_DrawGrid

Draws the Tic-Tac-Toe board grid with vertical and horizontal white lines. To create the grid's vertical divisions, the function uses predefined X-axis coordinates: Box1X, Box2X, and Box3X. These represent the positions where vertical lines should be drawn to segment the screen into three columns. The implementation avoids using a loop and instead draws each vertical line individually using the **TFT_DrawRect** routine, resulting in four vertical white lines in total. Similarly, the horizontal divisions are drawn using the coordinates Box1Y, Box4Y, and Box7Y. These are the Y-axis positions that separate the grid into three horizontal rows. As with the vertical lines, four horizontal white lines are drawn using **TFT_DrawRect**.

➢ *XO_ResetGrid*

It clears the game grid area and redraws the grid & pointer. It clears the main game area and the button bar area, then it resets pointer to its default value (R12 = 0x0010, stores in **Current_Pointer**). Finally, it draws the pointer outline again.

➢ XO_ResetScreen

It resets the full screen UI after a game round. By clearing the grid and redrawing the score area.

➢ *XO_OneWins / XO_TwoWins*

Each of them shows the winning player and its score through displays the images of winning, Extracting the score from each player's register, then drawing the score.

➢ XO_DrawDraw

shows the "DRAW" image.

➢ GetBoxCoordinates

Converts the active box into X and Y coordinates by iterating through bits from 0 to 8 (from box 1 : 9).

➢ DrawBoxOutline

Draws a rectangle outline for a single box (draws 4 rectangles as borders of the box).

➢ XO_DrawX / XO_DrawO

Extracts new pointer bits from R12, after that it uses **GetBoxCoordinates** to get the center of the box, then it adjusts the offsets and draws either **Ximg** or **oimg.**

Game Logic

When a button is pressed, the game checks if the player presses the PLAY button. If there is no press, the function exits. If there is input, the program checks if any player has already won or if the game is a draw. If so, it resets the game.

Before making a move, the game verifies that the selected grid position is not already taken. If the cell is free, it proceeds based on the current player's turn, which is tracked using a specific bit in the grid register (R9).

If it's Player X's turn, their move is recorded in R10 and the grid (R9). The turn is then passed to Player O by modifying the control bit. The **XO_DrawX** function is called to display the X on screen. The game then checks if Player X has won and, if so, updates the score and calls the win-handling function.

Similarly, for Player O, the move is saved in R11 and the grid. The **XO_DrawO** function shows the O symbol, and the game checks for a win or draw.

If there is a win or draw, the Reset section clears the first 10 bits of all three registers (R9, R10, and R11) to restart the game. The **XO_ResetGrid** function is also called to clear the display.

The CHECK_WINNING subroutine checks all possible winning combinations (rows, columns, diagonals). If any combination matches a player’s moves, it returns a flag by setting a specific bit. This helps determine if a win occurred.

In the **vsComputer** routine, Player X is human, and Player O is the computer. It follows a similar flow to the multiplayer logic: detecting input, validating the move, updating the state, and drawing X.

After the human plays, the computer takes its turn using **MOVE_COMPUTER**. It checks if it can win immediately; if not, it tries to block the player’s winning move. If neither is possible, it chooses the center, a corner, or any remaining spot based on availability. The selected move is displayed, and the game checks if the computer won.

The computer’s move is saved in R11, and the turn is passed back to the player. If the computer wins, the score is updated, and the corresponding win function is called.

PONG GAME

Input Function

The function starts by saving the current values of some registers to protect them during this operation.
Then it loads the current positions of paddle 1 and paddle 2 from memory into registers R8 and R7.
It reads the values from the input port (GPIOB), which contains the states of the buttons. These values are used to detect if the player has pressed a button.
To avoid reacting to quick, noisy button presses (which can happen due to the mechanical nature of buttons), the code checks the input value three times in a row. If the input is stable for all three checks, it continues. If it's different, it resets the check and starts over.
The code checks if a specific button is pressed (bit 5). If it is, the player wants to return to the main menu, so it ends the function and jumps to the **MAIN_MENU.**
Next, it checks if the reset button (bit 4) is pressed. If so, it sets a flag to indicate the game should be reset and ends the function early.

The function checks bits 0 and 1 to see if player 1 wants to move their paddle. If bit 0 is set, it moves the paddle up by adding 20 pixels. If it's already at the top, it stops at the top edge. If bit 1 is set, it moves the paddle down by subtracting 20 pixels but doesn't let it go past the bottom edge. The paddle's new position is drawn afterward.
Similarly, the function checks bits 6 and 7 to move player 2's paddle. Bit 6 moves it up, and bit 7 moves it down, both by 20 pixels. It also checks the top and bottom boundaries to keep the paddle inside the screen.
Once everything is done, the function stores the updated paddle positions back in memory. It calls a small delay (so the loop doesn't run too fast), then restores the saved registers and ends the function.

Logic Function

This function manages the core logic of the PONG game, including Ball Movement, Collision with paddles, Wall boundaries and Win conditions. After saving important registers in the stack and load game states like 1's paddle position, 2's paddle position, ball direction, X position of the ball, and Y position of the ball. If either player's "win bit" is set, jump to CHECK_R12 to handle game reset. If the direction is 0, the ball is moving up-right, toward Player 2's paddle. It calculates the ball's new position, checks if it's in range of the paddle, if it misses, Player 2 loses, If it hits, bounce ball and update direction to up left. If ball is moving up-left toward Player 1, It is similar logic to up-right, but for P1, If P1 misses, P2 scores. If the ball hits vertically in a direction, it prevents the ball from going out of bounds on top. If the ball is moving down-right toward Player 2, the same check on collision as before. Same in moving down--left.

After these checks, the function updates the ball position and draws it on screen. Then checking winning. If P1's score reaches 5, set the win bit and call **PONG_OneWins,** if player 2 wins, call **PONG_TwoWins.**

The **RESET_FINAL** section initializes registers and calls a new game function. The instruction BL **PONG_NewGame** calls a subroutine to start a new game, and the program then branches to **PONG_EXIT**, terminating the current execution flow. The **PONG_Reset** section resets registers R8 and R7 and prepares the game for a new round or reset state. The AND operation is used on R8 and R7 to clear bits in the range of 0x00000F00. The ORR operation then sets those cleared bits to 0x00000078, reconfiguring these registers. R11 is set to 160, and R10 is set to 26. The EOR instruction toggles a bit in R9. The CMP instruction compares the value in R3 to 1, and if the condition is not met (BNE), it branches to **P2_RESET** to reset Player 2. The BL **PONG_NewRound** calls a subroutine that starts a new round of the game. The program then branches to **PONG_EXIT** to exit the subroutine. For Player 2's reset, The BL **PONG_NewRound** calls the **PONG_NewRound** subroutine again, resetting the game or game state for Player 2, and then branches to **PONG_EXIT** to exit the subroutine. The CHECK_R12 section checks the value of register R12 to determine the next step. The CMP instruction compares the value of R12 to 0x00000001. If R12 equals 1, the program branches to **RESET_FINAL** to initiate the final reset process. Otherwise, it simply branches to **PONG_EXIT**, terminating the operation without resetting.

Drawing Functions

The **paddle drawing functions**, **PONG_DrawPaddle1** and **PONG_DrawPaddle2**, each begin by erasing the paddle's previous position by drawing a black rectangle over the old location. They use Y-coordinates for paddle height and rely on registers (R8 for Paddle 1 and R7 for Paddle 2) to get the current X-position. After clearing the previous paddle, they draw a white rectangle at the new position to represent the paddle.

The **ball drawing routine**, **PONG_DrawBall**, consists of two subroutines: **ClearOldBall** and **DrawNewBall**. These functions manage the animation of the ball by erasing the old ball location and drawing it at a new position. The ball's X and Y positions are encoded in registers R11 and R10, respectively, with the old and new coordinates packed into the upper and lower bits. **ClearOldBall** first extracts the old coordinates using logical shift and masking, then draws multiple overlapping black rectangles both vertically and horizontally to erase the ball from its previous location. **DrawNewBall** does the same but with white rectangles to render the ball in its new position. This layered rectangle technique creates a visually thicker and more rounded representation of the ball.

The **winning functions**, **PONG_OneWins** and **PONG_TwoWins**, are simple display routines that render a pre-designed victory image when player wins. They use the **TFT_DrawImage** function to display these graphics on the screen.

The **boundary drawing function**, **PONG_DrawBoundaries**, outlines the playable area of the screen. It draws white rectangles along the top, bottom, left, and right edges of the screen to form a clear border. This visual frame defines the limits within which the paddles and ball can move and interact.

The **score drawing routines**, **PONG_P1DrawScore** and **PONG_P2DrawScore**, are responsible for visually updating each player's score. The scores are encoded in the upper bits of registers R8 and R7. The functions extract the number of points, convert them into pixel positions, and render a point graphic (**Pong_Point**) under the respective player's label ("P1" or "P2") on the scoreboard area. This allows multiple point images to be drawn to represent the accumulated score of each player.

The **PONG_NewRound** function resets the playing field for a new round. It clears the game area, excluding the scoreboard, by drawing a large black rectangle over the play region. It then redraws the paddles and balls in their current positions using the appropriate drawing functions.

Finally, the **PONG_NewGame** function is responsible for fully resetting the game screen. It clears the entire screen, including both the game area and the scoreboard, and then redraws all essential elements: the boundaries, paddles, ball, and the scoreboard background. It also displays the score label image (**Pong_Score**) to frame where the score points will be drawn.

Swords Bound

We would like to note that since the memory was not enough to add the game along with the previous games, we have implemented it outside the menu.

Drawing Functions

- DRAW_STARTING_SCREEN

This function initializes and draws the starting visuals of the game. First, it fills the entire screen with a sky-blue background using a rectangular fill function (**TFT_DrawRect**). It then enters a loop to draw alternating segments of the sky using two image resources (**Sky_Top** and **Sky_Top2**), laying them horizontally from left to right. Afterwards, it completes the sky border by drawing two more sky tiles near the right edge.

Next, the function draws the ground by placing the **Ground_Left** and **Ground_Right** images at the top and bottom corners of the screen and then enters another loop to draw the general ground pattern in the middle using **Ground_General**, creating a complete terrain layout.

Following the background and terrain, the two warriors (players) are drawn at their initial positions using **Left_One** and **Right_One** sprite. Then, each player's full health bar is displayed at the top corners using the **Health100** and **Health100_2** images. After that, both warriors' lives are represented visually by three heart icons for each player.

- DRAW_DEC_WARRIOR1_HEALTH_BAR / DRAW_DEC_WARRIOR2_HEALTH_BAR

These two functions are responsible for updating the warriors' health bars visually as their health points decrease during the game. They read the health value from specific registers (R7 for Warrior 1 and R10 for Warrior 2) and check which range the health falls under. According to the health value, they select and draw the appropriate image at fixed positions to visually represent the current health level.

- DRAW_DEC_WARRIOR1_LIVES / DRAW_DEC_WARRIOR2_LIVES

These functions manage the visualization of the remaining lives for both warriors. They use bit manipulation to extract the lives count from R7 (Warrior 1) and R10 (Warrior 2), specifically bits 8 and 9. According to the lives left (3, 2, 1, or 0), they draw a sky-blue rectangle (**TFT_DrawRect**) over the heart icons to "erase" one at a time, giving the visual effect of life decrement.

- RESET_FIGHT

This function clears a portion of the screen to reset the battle area and redraws the warriors, their full health bars, and all three heart icons for both players. It uses the same positioning and image resources as the starting screen to reinitialize the fight zone, effectively preparing the game for a new round or restart after a win/loss.

- DRAW_WARRIOR1_MOVING / DRAW_WARRIOR2_MOVING

These functions are responsible for animating the movement of the warriors across the screen. They first erase the warrior's previous image by drawing a sky-blue rectangle over it. Then they draw the warrior sprite (**Left_One** or **Right_One**) at the new position. The positions are calculated using bitwise operations on the player position data stored in registers (R6 for Warrior 1 and R9 for Warrior 2), isolating the relevant 9 bits using a mask (0x1FF).

- DRAW_WARRIOR1_ATTACKING / DRAW_WARRIOR2_ATTACKING

These functions animate the attack sequence for both warriors. Like the movement functions, they first erase the previous warrior spirit. Then, they display a sequence of three different images **(Left_Three → Left_Two → Left_One**, or right equivalents) with delays in between to simulate an attacking animation. This creates a dynamic fighting scene.

- DRAW_WARRIOR1_DIZZY / DRAW_WARRIOR2_DIZZY

These functions show a dizzy animation, likely used when a warrior is hit or stunned. The current warrior image is cleared, and a dizzy sprite (**Left_Four** or **Right_Four**) is displayed temporarily before restoring the original sprite. This is also done with a brief delay to simulate a reaction or dazed state.

- DRAW_WARRIOR_ONE_WINS / DRAW_WARRIOR_TWO_WINS

This function visually represents a victory scene when a Warrior wins. It begins by clearing both warriors' current positions. Then, it draws the losing warrior in a dizzy pose and the winning warrior \ in an attacking pose. Finally, it draws a **GAMEOVER** image in the center of the screen, declaring the end of the game.

Input Functions

- FIGHT_INPUT

This is the main input handler function. It starts by saving important registers to preserve the state of the processor during input processing. It reads the GPIO input data register to determine which buttons are currently pressed. It then enters a debounce routine to verify that the input is stable and not just a quick fluctuation. After confirmation, it processes input for Warrior 1 (Player 1) and Warrior 2 (Player 2), handling their movement and attacks based on the button states. After all actions are handled, it waits for the input to finish and restores all saved registers before returning.

- CHECK_P1_ATTACK

This section checks if the attack button for Player 1 is pressed. If so, it checks a cooldown timer stored in R8. If the cooldown is over, it enables attack by setting the attack bit in R6. If there's a remaining cooldown, it reduces it instead of allowing an attack.

- FINISH_P1

After processing movement and attack for Player 1, this block finalizes the update by combining the new position data with the main register. It checks whether the position changed, and if so, it calls **DRAW_WARRIOR1_MOVING** to update the sprite on the screen.

- CHECK_P2

This block is the equivalent of the Player 1 movement section but for Player 2. It reads the direction input (right or left), calculates the new position, ensures the player does not move off the screen, and ensures Player 2 doesn't overlap with Player 1. It applies movement restrictions similarly to the first player.

- CHECK_P2_ATTACK

Like Player 1's attack check, this section evaluates whether Player 2 has initiated an attack. It checks the cooldown value in R11, and if it is zero, it enables attack by setting the appropriate bit in R9. Otherwise, it decrements the cooldown counter.

- FINISH_P2

This block completes the position and attack updates for Player 2. It merges the new position data into the main register and compares it to the previous value. If the position has changed, it calls **DRAW_WARRIOR2_MOVING** to animate Player 2's new position on the screen.

- FINISH

This label marks the end of the input processing. It calls the **Wait_Input_Finish** function to introduce a small delay before accepting new input and then restores all saved registers and exits the function.

Logic Functions

- SWORDS_BOUND _LOGIC

This is the main control function for the shadow fight logic. It begins by preserving the current register states using PUSH and proceeds to evaluate whether any player has won by checking if either player's life bits are zero. If so, it waits for an attack button press to reset the game. If both players are still in the game, it compares health values to determine attack priority, allowing the player with the lower health to attack first. It then calls the appropriate attack logic routines and exits by restoring register states.

- CHECK_P1_ATTACK_LOGIC

This function handles Player 1’s attack logic. It first checks whether P1 has initiated an attack and whether the delay counter is zero. If these conditions are met, it examines the hit counter to ensure P1 is still capable of attacking. If so, the hit counter is decremented, and an attack animation is triggered. It checks whether P2 is in range and, if so, pushes P2 back and reduces P2’s health. If P2’s health falls below the damage threshold, it triggers a life decrement, resets positions and counters, and updates the health and life bars accordingly.

- CHECK_P2_ATTACK_LOGIC

This is the mirror function of **CHECK_P1_ATTACK_LOGIC**, but for Player 2. It verifies if P2 has issued an attack, ensuring both the delay and hit counters allow for it. If the attack is valid, P2’s hit counter is decreased, and an attack animation is shown. If P1 is in range, they are pushed back, and their health is reduced. In case P1’s health drops to or below zero, a life is deducted, and their state is reset for the next round. The visuals for health bar and dizzy animation are also updated accordingly.

- P1_LOSES_LIFE

This function executes when Player 1’s health is depleted. It resets their health to zero visually, draws a dizzy effect, and decrements their lives. It checks if any lives remain; if not, Player 2 is declared the winner. If lives remain, both players’ positions and counters are reset for the next round, and the appropriate visual updates are drawn.

- P2_LOSES_LIFE

Like **P1_LOSES_LIFE**, this function handles the case when Player 2's health reaches zero. It reduces P2's life count, updates health visuals, and if P2 has no lives remaining, Player 1 is declared the winner. Otherwise, both players' positions, health, and counters are reset, and corresponding UI elements are updated to reflect the new round.

- RESET_P1_COUNTERS

This function resets Player 1's counters (delay and hit) to their default values and clears their attack bit, which effectively ends their current attack phase. It is used when P1's hit counter reaches zero.

- RESET_P2_COUNTERS

Acts just like **RESET_P1_COUNTERS**, but for Player 2. It restores the default delay and hits counter values and clears the attack bit to stop the ongoing attack logic if P2's hit counter has expired.

- P1_FINAL_WIN

This function is called when Player 2 runs out of life, meaning Player 1 wins the game. It resets both players' counters and calls the drawing routine to display that Player 1 has won, then exits cleanly.

- P2_FINAL_WIN

This function mirrors **P1_FINAL_WIN**. It is triggered when Player 1 loses all lives, resulting in Player 2's victory. It resets game counters and invokes the function to draw the victory screen for Player 2.

- RESET_FINAL

This function performs a full reset of the game's state. It resets both players' positions, health values (to full), lives (to 3), and counters to default. It then calls the function responsible for redrawing the entire game screen, setting it up for a fresh start.

- SWORDS_BOUND _EXIT

This is a utility function used at the end of all other major routines. It restores all previously saved registers and returns control to the caller using BX LR.

Future Enhancements

The following enhancements have been scoped for future releases due to current time and memory resource limitations.

1. Connection of an external memory
2. Using" touch" to interact with the screen
3. Adding multiple themes for each game so the user can switch between them
4. Connection of a buzzer so that it makes a sound for winning and for unacceptable input.
5. Connection of a joystick