Space Invaders in Assembly: A Microprocessor-Based Game System

Laksh Gehani

Abstract — This project implements a version of the classic arcade game *Space Invaders* on the PIC18 microprocessor, using MPASM assembler to control graphics, input, and sound. The system features real-time gameplay on a 128×64 graphical LCD, with user input from a 4×4 keypad and audio feedback from a buzzer. Performance testing demonstrated smooth operation at 9.5 ± 0.04 frames per second, 100% collision detection accuracy, and highly efficient resource usage. Only 2.5% of program memory and 2.3% of data memory were utilised. The project showcases how modular design and efficient memory handling can be used to build interactive embedded systems.

I. Introduction

Embedded systems form the foundation of modern digital electronics, allowing for development of efficient and reliable devices. Microprocessors serve as the core processing units of these systems. From household appliances such as washing machines [1] to critical systems in industrial machinery [2], microprocessors enable highly reliable and efficient operations. Their widespread adoption is largely attributed to their compact size, cost-effectiveness and low-power consumption. These features make microprocessors highly suitable for tasks that require dedicated control logic and precise timing. Beyond industrial automation, microprocessors have also been integral to the evolution of the gaming industry, enabling the development of interactive electronic games. A notable example is Nintendo's Game Boy, a handheld gaming console released in 1989, which utilised a custom 8-bit microprocessor known as the Sharp LR35902 [3]. This allowed the console to execute game logic, manage graphics and respond to real-time user input using minimal hardware resources. Its architecture laid the foundation for portable gaming and showing that engaging gameplay could be achieved through efficient low-level programming.

Inspired by this idea, the current project seeks to replicate a simplified game environment on a modern microprocessor. The motivation for this project stems from an interest in exploring how cost-effective and portable gaming systems can be developed using microprocessors. The aim is to develop a classic 2D arcade shooter game, Space Invaders, originally released in 1978 [4]. In this game, the player controls a cannon to defend against waves of descending alien invaders. This project aims to implement Space Invaders on a PIC18 microprocessor, programmed entirely in assembly language using MPASM assembler. The system utilises a graphical LCD for the output, a keypad for user input and a buzzer for sound effects corresponding to in-game events. The project utilises a modular approach to easily implement and test new functions independently. The design principles utilised in this project can be directly applied to advanced applications such as interactive entertainment systems.

II. System Overview

The goal of Space Invaders is for the player to defend from descending alien invaders. The player controls the cannon located at the bottom of the screen and must shoot projectiles upward to destroy the enemies before they reach the player's row. The game is over either when an alien successfully collides with the cannon or reaches the player's level on the screen. The player can move left or right to align their cannon with the enemy and fire bullets. The challenge is in timing the bullets as there is a delay between firing consecutively. The software for the game is designed for real-time responsiveness and efficient resource management. The system broadly consists of the following three main components:

1) Input Interface: The input consists of a keypad which allows the user to control game actions. Specific keys are assigned for movement (left, right), shooting bullets and start/reset functionality.
2) Processor: A PIC18 microprocessor processes the inputs, executes the core game logic and produces visual output via the LCD and audio output using the buzzer.
3) Output Interface: A graphical LCD is used to display game sprites. The buzzer produces sound effects corresponding to in-game events.

III. Hardware Design

The system consists of the following key hardware components:

1) PIC18F87K22 Microprocessor (Model 30009960E): It is a powerful 8-bit microprocessor with a 64 MHz internal oscillator, delivering up to 16 million instructions per second [5]. It acts as the central processing unit that executes all game logic, including player and enemy movement, bullet dynamics, collision detection, scoring and game-over conditions. It includes 128 kB of flash program memory for storing the game's logic and lookup tables for assets, 4kB of data memory for real-time variables such as positions, timers and flags and 1 kB of electrically erasable programmable read-only memory (EEPROM) for non-volatile data storage.

2) 128x64 Graphical LCD (Model WDG0151-TMI-V): It is responsible for displaying all visual elements of the game, including the player cannon, enemy aliens, projectiles, score and game-over messages. The GLCD is structured as a matrix of 128 columns by 64 rows. The screen is logically divided into 8 pages, each representing a horizontal band that is 8 pixels tall. Each of the 128 columns store a byte that

corresponds to 8 pixels tall. The display is internally split into two 64x64 pixel halves, each driven by a NT7108 driver [6]. Control signals (CS1 & CS2) are used to select which half of the screen to address. The GLCD communicates to the processor using an 8-bit parallel interface [7]. The GLCD communicates with the processor using PORT B and PORT D. Pins 0-5 on PORT B are used to send control signals while all pins on PORT D (0-7) are used to send data, allowing fast pixel updates.

3) 4x4 Keypad: Functions as the primary input device, with dedicated keys mapped to left, right, shoot and start-reset commands. As the keypad has no internal logic, it operates purely as a set of switches [8]. All eight pins of the keypad connect to PORT E to send input commands to the microprocessor. To detect a keypress, the keypad is scanned using an interrupt routine. This cycles through each row of the keypad and checks for signals, allowing the system to detect which key has been pressed.

4) EasyPIC PRO V7: The board includes built-in slots for the microprocessor and GLCD for easy hardware integration [9]. Ports on the board provide connections for peripherals. Also includes an onboard voltage regulation and a built-in buzzer for game sound output.

5) Buzzer: Produces audio feedback for game events such as shooting, enemy destruction and game-over conditions. It is connected to pin 6 on PORT B and can generate sounds at varying frequencies.

6) Power Supply: The system is powered by a stable 15V power source to run all components smoothly.

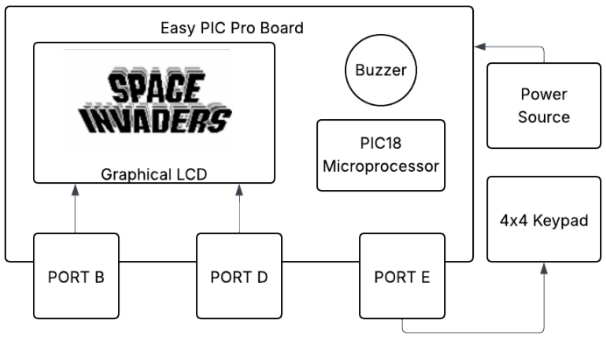

Figure. 1: Complete hardware setup of the Space Invaders system. The PIC18 microprocessor is mounted on the EasyPIC PRO board. It interfaces with a 4×4 keypad (PORT E), a 128×64 graphical LCD (PORTS B and D), and a buzzer in-built on the board. The board is powered by a 15V power supply.

IV. Software Design

A. Initialisation

Upon startup, the microprocessor configures all peripherals required for the game. The first step involves configuring the microprocessor's I/O ports, the GLCD, keypad and sound output (see Section III). The system then displays the title screen. This is rendered using a predefined bitmap image which is stored as an array in program memory. The bitmap consists of a sequence of binary values, each representing a vertical column of 8 pixels, collectively forming the words "SPACE INVADERS" in stylised block characters. These binary values are arranged to match the internal memory structure of the GLCD. Instead of manually designing each character, an online bitmap generator tool [10] is used which converts input text into stylised pixel-based characters.

B. Input Handling

After displaying the title screen, the system enters a waiting state where it continuously scans for user input. Keypress detection is handled by an interrupt-driven routine that regularly polls the keypad. When a key is pressed, the corresponding value is read from the port and compared to a set of predefined values to identify the intended action. At the title screen, the system waits specifically for the Start key to be pressed. Once detected, the system exits the idle state, clears the screen and initiates the main game loop.

During gameplay, directional keys control the movement of the player cannon. The left and right keys shift the cannon horizontally across the screen by 8 pixels. The cannon's current position is stored in data memory and updated whenever a directional key is pressed. Before each movement is applied, the system performs a boundary check to ensure that the cannon remains within the limits of the display. Y-addresses 96 through 120 are dedicated solely to the score display (see Fig. 2). No gameplay actions occur in this region. When a movement input is received, the microprocessor calculates the new position and compares it to the game boundaries. If the new position would result in the cannon exceeding the boundaries, the movement is cancelled, and the cannon remains in place.

The shoot key allows the player to fire a projectile upward towards the descending alien enemies. When the shoot key is pressed, the system checks whether a bullet is already in flight. If no active bullet exists, a new projectile is created and initialised at the top centre of the cannon's position. The upward movement of the bullet is handled in the movement module.

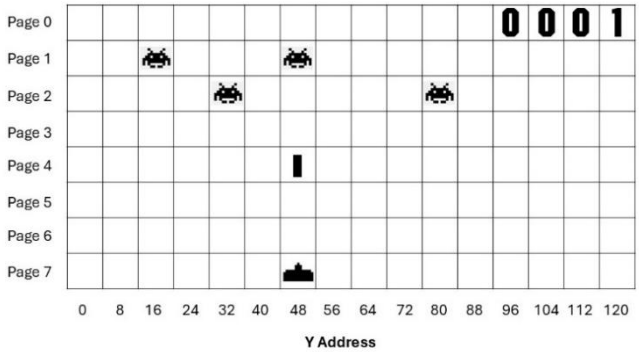

Figure 2: The 128×64 graphical LCD displays game elements across 8 vertical pages and 128 horizontal columns, with each column controlling 8 vertical pixels.

C. Main Game Loop

The game logic module is responsible for coordinating all real-time, dynamic behaviours in the game. It handles enemy generation and movement, bullet updates, collision detection, score tracking, and game-over conditions. Once the game transitions out of the title screen and into active gameplay, the system enters a main game loop, which repeatedly cycles through these operations until a game-over condition is met. The software uses a modular approach as seen in Fig. 3. The structure and sequence of the main game loop is detailed in a flowchart in Fig. 4.

The microprocessor is configured to run with a frequency of 64 MHz. As one instruction requires 4 clock cycles, the instruction execution frequency is 16 million instructions per second (MIPS). Thus, one instruction takes 62.5 ns to execute. To ensure the game runs at a smooth pace, a fixed delay of 90 ms is introduced at the end of each iteration. This corresponds to roughly 1.44 million instruction cycles. The delay is implemented using nested decrement loops, where counters stored in data memory are decremented until zero. Without this delay, the game would execute its logic far too quickly, making enemy movements near instantaneous, giving the player little to no time to react. The 90 ms delay effectively limits the frame rate of the game to approximately 11 frames per second (fps). This also standardises the timing of other gameplay mechanics such as enemy movement, bullet cooldowns and collision detection. In practice, performance may vary slightly due to interrupts and additional instruction cycles in gameplay logic.

D. Enemy Generation

Enemy generation in the game is handled dynamically to create a sense of challenge that increases over time. The horizontal positions of newly generated enemies are determined using a Linear Congruential Generator (LCG), which produces pseudo-random numbers based on the following equation:

$$X_{N+1} = (A \times X_N + C)\ mod\ M \quad (1)$$

where X_N is the current generated value, X_{N+1} is the next pseudo-random value, A is the multiplier constant, C is the increment constant and M is the modulus.

The modulus M defines the upper bound of the generated values and is set to 2^8 (256), ensuring that all generated values remain within the 8-bit range of 0 to 255. The multiplier A is chosen as 77. This is relatively prime to M (no common divisors except 1), allowing for a full-period generator that cycles through all possible values without repetition. The increment C can be set to any odd number to break symmetry in the output. In this case, it was set to 17. The expression $A \times X_N + C$ can produce a 16-bit result (up to 65535). Because M is chosen as 256, calculating the result of the expression modulo M is equivalent to simply taking the lower byte of the 16-bit result. This makes the LCG particularly quick and computationally efficient as the calculations require no division. The initial seed value X_0 is stored in data memory and can be updated within sessions to simulate randomness across games. Each new value is scaled down to map it to one of the 16 horizontal tiles across the screen. Before an enemy is placed, the system checks whether the target tile is already occupied to prevent overlapping. If the tile is occupied, the system generates a new value using the LCG and repeats the check until an unoccupied tile is found.

When the game begins, only one enemy is active at a time, giving the player time to adapt to the mechanics and controls. After the first three enemies are killed, the system gradually begins to spawn multiple enemies simultaneously, increasing the difficulty of the game. An internal counter keeps track of the maximum number of enemies that can be active at any given time. Each time an enemy is destroyed, a flag is set in data memory to indicate that a kill has occurred. This flag is checked during the next enemy update cycle which is every 7 game loop iterations (≈ 630 ms). If active, the enemy count is incremented by one, up to a maximum of seven enemies.

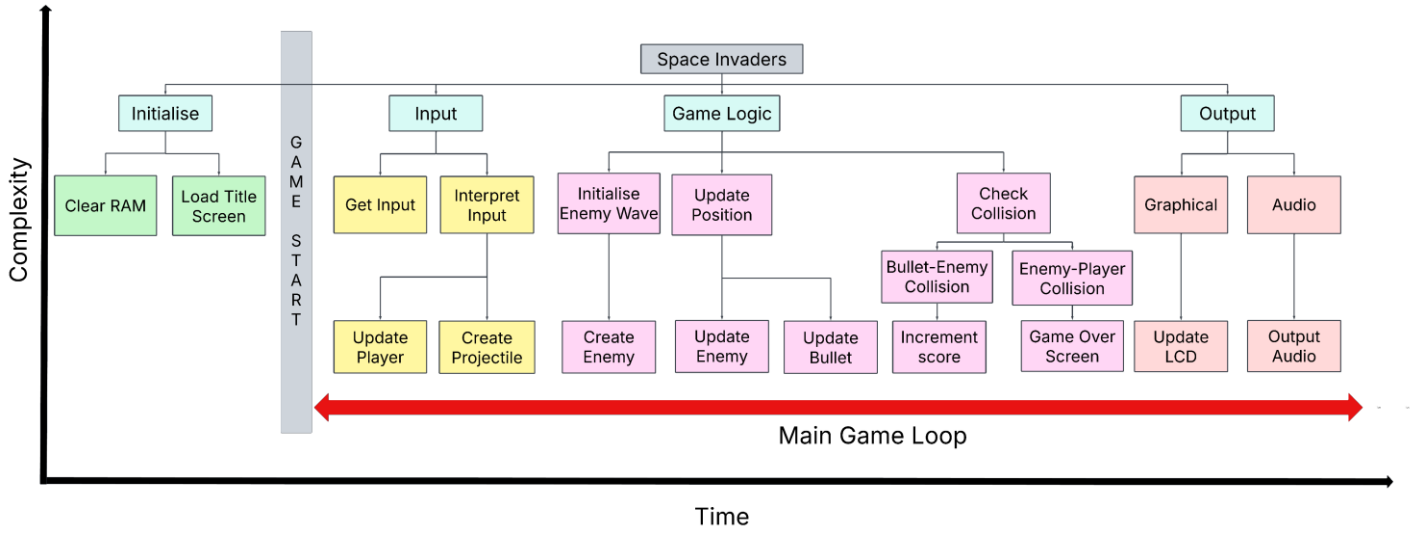

Figure 3: Modular diagram of the game's software
This diagram outlines the modular structure of the game, showing how each subsystem interacts over time within the main game loop. The game begins with initialisation, followed by continuous input handling, game logic, and output rendering. The vertical axis represents complexity, while the horizontal axis is the sequential flow of execution during gameplay.

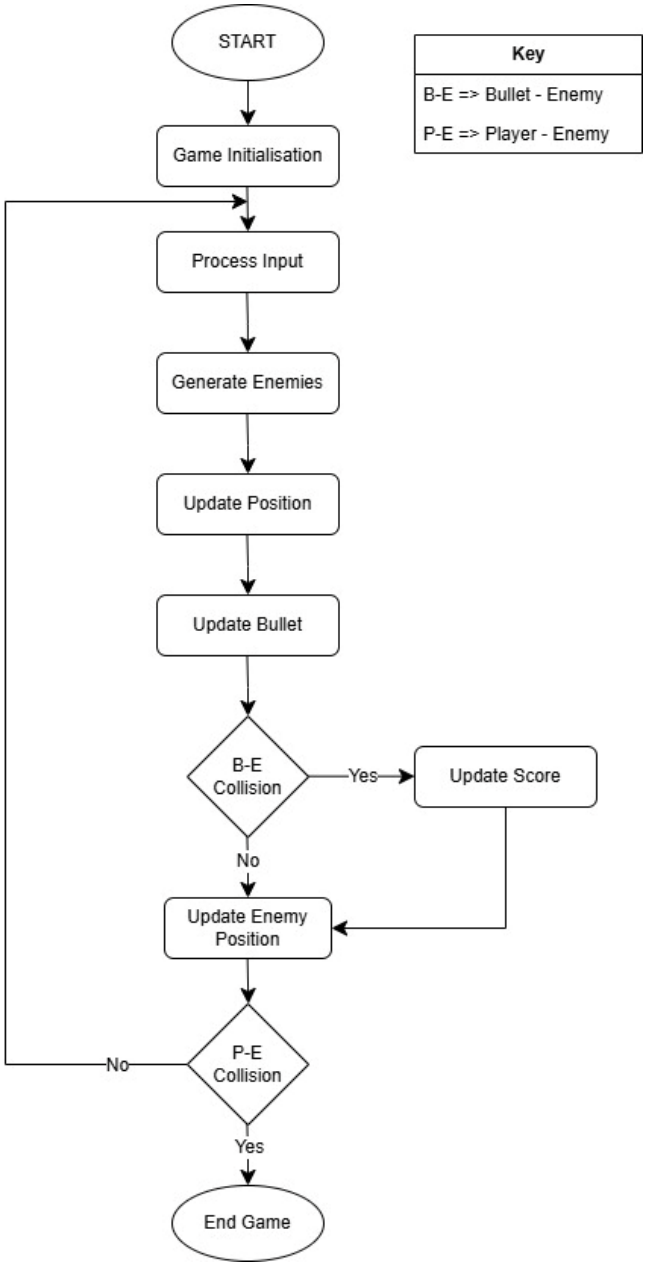

Figure 4: Game Logic Flowchart for Space Invaders This flowchart explains the sequence of operations executed within the main game loop. After game initialisation, the system processes player input, generates enemies, updates object positions, and checks collisions. Bullet-enemy (B–E) collisions results in score updates, while player-enemy (P–E) collisions trigger the end of the game.

E. Enemy & Bullet Movement

In each iteration of the main game loop, the system updates enemy and bullet positions based on timing logic. Each enemy uses a movement delay counter, which is incremented once per game loop cycle. When the counter reaches a threshold value of 7, the enemy moves down by one page (8 pixels) and the counter is reset to zero. If an enemy reaches page 7, a game over condition is triggered, terminating the game loop. In contrast, bullets move at a much faster rate. If a bullet is active, it will move upward by one page every iteration. However, bullets cannot be fired continuously. A new bullet is only created if no other bullet is currently active on the screen. This means the previous bullet has either exited the screen or successfully hit an enemy. This introduces a natural cooldown between firing bullets, preventing rapid-fire. Additionally, this built-in delay helps mitigate button debouncing effects. This reduces the chance of multiple bullets being created from a single prolonged or bouncing keypress.

F. Collision Detection

The collision detection module is responsible for identifying when a bullet hits an enemy or when an enemy collides with the player. It is executed during each iteration of the main game loop, immediately following the movement of both entities. Since enemy and bullet movement are updated at different rates, collisions are checked immediately after each movement update to ensure no overlaps are missed.

When a bullet is active, the system checks for collisions by comparing its Y address to each enemy currently on screen. If a match is found, it further checks the page address to confirm alignment. A successful hit is registered if both the addresses match. However, as enemies move downward every 7 cycles and bullets move upward every cycle, it is possible for the two objects to pass through each other in the same iteration without having the same page address. This leads to some collisions being missed during initial testing. To account for this, an additional check is performed: a collision is also registered if the enemy's page address is exactly one page above the bullet's current page. This ensures that if the two objects are one page apart but moving toward each other, the collision is still detected. Once a collision is detected, the bullet and enemy are removed from memory and the screen. The enemy's state is reset, and a new enemy is initialised at the top of the screen with a new Y address using the LCG. After a successful hit, the score is incremented by one and a sound effect is played through the buzzer to provide audio feedback.

In addition, the game also checks for collisions between enemies and the player cannon. Since the player is restricted to the bottom row of the screen, a player-enemy collision is detected if an enemy reaches page 7 and has the same Y address as the player. As enemy movement is updated every 7 game loop cycles, player-enemy collisions are evaluated only during these cycles, reducing unnecessary checks. If such a condition is met, the system triggers a game-over flag, exits the main game loop and clears the screen. However, the game is designed to detect if an enemy reaches page 7, even if not aligned with the player. This will also exit the main game loop. Even if multiple enemies reach the bottom of the screen simultaneously, a single collision is sufficient to end the game.

G. Score

The score system in the game keeps track of the number of enemies eliminated by the player. The score is updated immediately after each successful enemy hit, ensuring real-time feedback to the player. Internally, the score is stored as a 16-bit binary value in data memory. However, to convert

into human-readable format, this binary value must be converted to its 4-digit decimal representation (0–9999). Due to the microprocessor's limitation of 8x8 bit multiplication, any multiplication involving larger operands must be implemented manually. To extract the decimal digits of the score, the 16-bit binary value is first split into two 8-bit halves, high and low bytes. Manual routines simulate 16×16-bit and 24×8-bit multiplication by performing multiple 8×8 operations and combining the partial results using positional shifts and additions. This allows the program to isolate each digit (thousands, hundreds, tens, and units) based on its decimal weight. Once the four digits are extracted, each one is matched to a predefined bitmap stored in program memory. These bitmaps are 8×8-pixel patterns corresponding to digits 0 through 9. These are written sequentially to a dedicated display area on the GLCD (Y-addresses 96 to 120 on page 0). Enemies are prevented from spawning in this region, and the player cannot move into it, ensuring the score display remains clear throughout gameplay.

When a game-over condition is triggered, the main game loop is exited, and the screen is cleared. A predefined "GAME OVER" bitmap stored in program memory is displayed. This is rendered column by column to read each byte from a lookup table and write to the GLCD, similar to the title screen bitmap. In addition, the system also displays the high score achieved during the session. After each game, the player's final score is compared to the previous high score stored in data memory. If the current score is higher, it overwrites the previous high score. However, since this value is stored in data memory, it is reset when the system is powered off or manually restarted. As a result, high scores are maintained only during a single session.

H. Output

The system provides two forms of output: visual and audio. The game logic module calls dedicated subroutines to produce outputs. This avoids interrupting the main game loop while still providing immediate feedback. All visual elements are displayed on the graphical LCD. All visual assets are stored as lookup tables (bitmaps) in program memory, with each value representing a vertical 8-pixel column. Enemies and bullets are drawn at their current Y-address and page. Before updating their position, the system clears their previous location by writing blank bytes to their same coordinates to avoid graphical remnants. The player cannon is also updated similarly, redrawn only when a movement key is pressed. Its horizontal position is stored in data memory and constrained to a valid range (0–88 Y-addresses) to prevent overlap with the score area. Score digits are only redrawn when there is a successful bullet-enemy collision.

Audio feedback in the system is produced through a buzzer connected to PORT B pin 6 which connects to the microprocessor. Sound effects for key gameplay are produced by toggling the pin at specific frequencies to simulate square wave pulses. Each sound is generated by setting the pin high and low repeatedly within a loop, with delays between transitions. These delays determine the frequency of the tone. The delays are implemented using decrement loops stored in data memory, much like the delay system used in the main game loop. Different events such as shooting, scoring a hit, or game-over use distinct delay patterns to produce distinguishable sound cues.

V. Results & Discussion

To evaluate the effectiveness and responsiveness of the game system, several performance metrics were measured during gameplay on the PIC18 microprocessor. The following criteria were used to assess system performance:

1) Frame Rate: The game uses a fixed software delay to regulate frame updates, aiming to achieve a consistent frame duration of 90 ms. This corresponds to an ideal frame rate of approximately 11 frames per second. However during gameplay, measurements indicate a higher average frame duration of 105.3 ± 0.48 ms, as seen in Fig. 5. This corresponds to a practical frame rate of 9.5 ± 0.04 fps. To collect this data, frame timing was measured by toggling pin 1 on PORT F at the start and end of each iteration of the main game loop. This was captured using a digital oscilloscope connected to the port, allowing for precise measurements. Over 3500 consecutive frames were recorded during gameplay. The standard deviation of 0.48 ms in frame duration corresponds to a frame rate uncertainty of approximately 0.5%. This suggests that all dynamic events remain accurate within a single game loop cycle. The deviation from the ideal value can be attributed to several factors. Firstly, the interrupt-driven keypad routine occasionally halts the main game loop, introducing small but accumulative delays. Secondly, the GLCD is updated frequently during gameplay, especially when drawing multiple enemies. Since GLCD writes are relatively slow, this significantly contributes to higher frame times. These combined effects result in the observed drift from the intended frame rate.

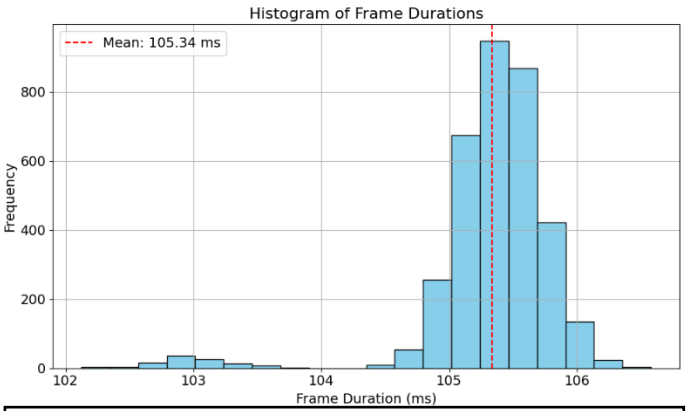

Figure 5: Histogram of frame durations measured over 3500 frames during gameplay. The majority of frames cluster around a mean of 105.3 ms, indicating a practical frame rate of approximately 9.5 fps. The spread reflects minor timing variations caused by interrupts and GLCD write delays. The dashed red line marks the average frame duration.

2) Memory Usage: Memory utilisation is a key metric when working with microprocessors with constrained resources. During gameplay, the available data memory is used to store real-time data such as current player position, active bullet and enemy coordinates. The program memory holds the main game code, bitmap assets and all logic modules. To evaluate

the memory usage, the project is compiled using MPLAB X IDE, which automatically generates a memory summary upon a successful build. This shows that the compiled game uses 2.5% of program memory and 2.3% of data memory. No EEPROM space is used during the implementation. As the overall memory usage is minimal, the system remains highly scalable, allowing further features to be added with ease.

3) Collision detection accuracy: To evaluate the effectiveness of the collision detection system, the game was tested during an extended play session lasting approximately 30 minutes. During this time, collision events between bullets and enemies, were closely monitored using internal counters. Two counters were used: one to track how often a collision condition was checked, and another to track how many of those checks led to a confirmed hit and removal of the objects from the screen. Across over 1700 bullet-enemy interactions, the vast majority of collisions were detected and responded to accurately. During testing, an edge-case bug was identified where if an enemy reached page 6 in the same column as the player, there was not enough vertical space above the cannon to draw a new bullet. As a result, the bullet would not appear and the enemy would continue descending, leading to a game-over condition being triggered even though the player attempted to shoot. To resolve this, the system was modified to allow the bullet logic and collision detection to proceed even if the bullet could not be drawn.This ensured that the enemy could still be destroyed, preserving game fairness. Additionally, a small number of missed collisions were initially observed, typically occurring when the bullet and enemy passed each other between frames without landing on the same page. However, after including the secondary check where a collision is detected when the enemy is one page above the bullet (see section III.F), the system achieved a collision detection accuracy of 100%.

VI. Constraints & Improvements

Despite the successful implementation of a playable Space Invaders clone on the microprocessor, the project encountered several constraints most of which stemmed from the hardware limitations. Programming in low-level assembly provided full control over execution but increased the development time and complexity. The microprocessor lacks the capacity for multiplication of large integers which made arithmetic-heavy tasks like score conversion more difficult. Furthermore, the microprocessor has no operating system or real-time scheduler, requiring all timing and concurrency to be managed manually. Additionally, the GLCD used a relatively slow parallel interface with manual byte-wise data transmission. Frequent updates, especially when multiple enemies or bullets were redrawn introduced a larger delay than intended.

Several enhancements could be added to further expand the functionality and gameplay experience of the system. One such improvement would be the introduction of a lives system, where the player is allowed multiple attempts before the game ends. This would reduce the harshness of a single mistake leading to immediate game over, providing a more engaging experience. The number of lives could be tracked using a simple counter in data memory and displayed alongside the score using small icons.

Another improvement would be level-based progression. Currently, the difficulty increases gradually by spawning more enemies over time. However, by dividing the gameplay into levels, the game can escalate challenges. For example, higher levels could reduce the delay between enemy movements, increase the number of enemies spawned, or introduce new enemy movement patterns like diagonal motion. These changes would require relatively minor additions to the game logic but would significantly improve replayability.

Finally, the current implementation loses the high score are on reset, as they are stored in data memory. This can be resolved by utilising the EEPROM of the microprocessor which offers 1kB of non-volatile memory. By writing the high score to EEPROM at the end of each session and reading it back on startup, the game can store the player progression. This change would only require a few additional EEPROM read/write instructions, making it an efficient yet impactful improvement.

VII. Conclusion

This project successfully demonstrates the implementation of a simplified game of Space Invaders on a PIC18 microprocessor, using assembly language. Despite operating withing tight hardware constraints, the system delivers a smooth and responsive game through careful software design and efficient use of memory. Key features include pseudo-random enemy placement, real-time collision detection, score tracking and audio-visual feedback.

Performance evaluations show consistent frame rate of 9.5 ± 0.04 fps, 100% accurate collision handling, 2.5% of program memory and 2.3% of data memory usage. The modular structure of the codebase allows for future enhancements such as the addition of a multi-life system, level-based difficulty progression, and persistent high score storage using EEPROM. These improvements can be implemented with minimal overhead due to the efficient resource allocation of the current system. Overall, the project highlights the feasibility of developing interactive games on resource-constrained microprocessors. It shows that engaging gameplay can be delivered on smaller systems with careful low-level optimisations. The success of this project also highlights the potential for further advancement in the field of game development, where lightweight and low-cost platforms can continue to deliver increasingly interactive experiences.

APPENDIX

Cycle 1 feedback: There was good background research done on energy costs and use, providing strong motivation for the work. An area that can be improved is to ensure that your conclusions are quantifiable (where possible), very clearly. “Power can’t sustain a house”. Does that mean “solar energy is a good alternative” or not.

Cycle 2 feedback: Include official datasheet references for all hardware components used. Clearly define target performance goals. State the memory capacities of the microprocessor and how much is used by a typical game.

REFERENCES

[1] Desai, R. (2013) The role of MCUs and PSoCs in making home appliances smarter, Embedded.com.

[2] Rutronik. (2024) Microcontroller for high-end industrial equipment, robotics, and industrial drives. rutronik.com

[3] Abhinav the Builder. (12 October 2020) *Gameboy – Everything under the hood*, Dev.to.

[4] Britannica, Space Invaders, Video game series, Britannica.com

[5] Microchip Technology Inc. (2024). PIC18F87K22 Family Data Sheet

[6] LCD Driver Datasheet. Sitronix Technology Corporation

[7] Winstar Display Corporation. (2009). WDG0151-TMI-V#N00 128x64 Graphic LCD Module Datasheet.

[8] Farnell, 4x4 Matrix Keypad – Technical Datasheet.

[9] MikroElektronika. (2013). EasyPIC PRO v7 User Manual

[10] FontBolt. Space Invaders Font Generator.