; revised .

8086 Assembly Game Implementation

Viteri Demian[1]

[1]Universidad de Investigación de Tecnología Experimental Yachay Tech, San Miguel de Urcuquí, Imbabura, Ecuador

Corresponding author: Demian Viteri (email: demian.viteri@yachaytech.edu.ec).

ABSTRACT This project presents the implementation of a retro Space Race–inspired video game using the Intel 8086 assembly language. Assembly language programming enables direct CPU communication through registers, memory segments, and system interruptions, essential to implement graphics, input handling, and logic for the game. The game is structured into stack, data, and code segments, and it incorporates concepts the Pong video-game, particularly regarding register use, interruptions, collision detection, menu design, and point systems. Some mechanics from Pong were adapted, including player movement, projectile interaction, and scoring, while new modifications were introduced to create the Space Race video-game: spaceship reset logic, finish-line scoring, meteor behavior, race timing, and a redesigned spaceship shape. The final Space Race video-game four scenes: a main menu, the game, a restart menu, and an Easter egg screen. Despite some functionalities working, issues were encountered with array management, meteor logic and collision detection. These challenges highlight the complexity of low-level programming and the importance of proper project organization, procedure usage, and cohesive game logic design.

INDEX TERMS Assembly languages, Microprocessor programming, Computer Architecture, Video games, Debugging

I. INTRODUCTION

ASSEMBLY language is a low-level programming language dependent on the instruction set and architecture of the CPU [1]. This language is then processed by a compiler into machine code (i.e. 0's and 1's). Even though programming in assembly language involves sacrificing portability and maintenance, due to the way instructions are written, it offers a great way to access system hardware and, therefore, efficiency [1]. 8086 assembly language is used to program on the Intel's 8086 microprocessor with a 16 bit architecture, and it is used in this project to implement a variant of the retro video-game Space Race, developed by Atari and published in 1973.

A. 8086 ASSEMBLY LANGUAGE

The 8086 assembly language is dependent on the Intel's 8086 microprocessor. Due to its architecture (16 bits) it is possible to work with up to 16 bit registers (a huge improvement when compared to the 8085 microprocessor). There are fourteen registers available in the 8086 [2], those being:

- General-Purpose Registers: AX, BX, CX, DX.
- Index/Pointer Registers: SI, DI, BP, SP.
- Segment Registers: CS, DS, SS, ES.

There are also system interruptions that allow for more complex processes such as video routines (INT 10H), DOS: Disk Operating System services (INT 21H), and keyboard input (INT 16H) [2]. These interruptions were fundamental when implementing the video-game.

II. METHODOLOGY

A. PROJECT STRUCTURE

When writing a program in assembly language, it is necessary to structure the project properly by defining a Stack Segment, a Data Segment, and a Code Segment (Figure 1). As their name implies, each segment will contain certain information. The Stack Segment will be used by the Stack, the Data Segment will, essentially, store the definition of variables for the program to use, and the Code Segment will hold the code of the actual program. In order for these segments to be recognized by assembly, it is necessary to reference them in the code.

B. SPACE RACE VIDEO-GAME

In order to understand the 8086 assembly language and to implement the Space Race video-game variant, a tutorial on how to implement the Pong video-game was followed [3]. In this tutorial, basic register and interruption manipulation was learned. It was also useful to understand and implement the core logic of the Space Race video-game, menus, graphics and player input. The tools used for this project were a text editor (such as Visual Studio Code), a DOS emulator

(DOSBox) and a set of programs to compile the .asm file into a DOS executable (Figure 2).

```
;-------------------------------------------------------------;

STACK SEGMENT PARA STACK

    DB 64 DUP (' ')

STACK ENDS

DATA SEGMENT PARA 'DATA' ; DEFINE VARIABLES HERE.

; SYSTEM VARIABLES:

    VIDEO_MODE       DB 13H  ; VIDEO MODE: (320 x 200) AND 256 COLORS.
    WINDOW_WIDTH     DW 140H ; 320px.
    WINDOW_HEIGHT    DW 0C8H ; 200px.
    BORDER_FIX       DW 02H  ; TO FIX COLISSION WITH WINDOW BORDERS.
    BACKGROUND_COLOR DB 00H  ; BLACK.
    PIXEL_COLOR      DB 0FH  ; WHITE.
    TIME_AUX         DB 00H
    GAME_ACTIVE      DB 01H  ;
    EXITING_PROGRAM  DB 00H  ; BOOL.
    CURRENT_SCENE    DB 00H
    MAX_SCORE        DB 0AH ; SCORE TO WIN: 10PTS.
    WINNER_INDEX     DB 00H
```

FIGURE 1. Some of the Project Structure - Space Race video-game

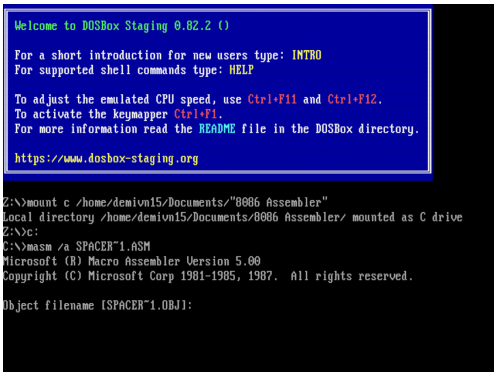

FIGURE 2. DOSBox Emulator

FIGURE 3. Main Menu - Space Race video-game

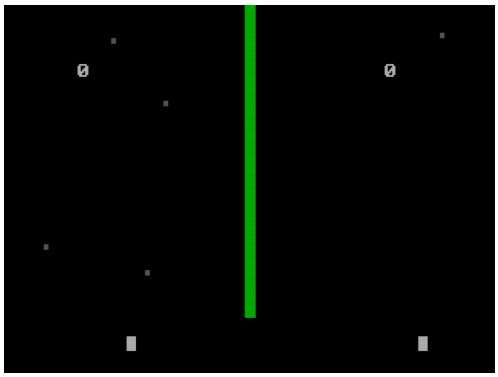

FIGURE 4. Video-game - Space Race video-game

1) Useful mechanics from Pong video-game

The Space Race and the Pong video-games have different mechanics in common such as player movement, projectiles, collision, and points system. In Pong, the player controls the paddles up or down, same as the spaceships controls in Space Race. In Pong, there is a ball that collisions with the paddles, while in Space Race, there are meteors that collide with the spaceships. Finally, in Pong players gain points when scoring goals into the opponent side. Meanwhile in Space Race, players sum point when reaching the finish line.

2) Changes made to implement the Space Race video-game

In order to implement the Space Race video-game some changes were made to the base code from the Pong video-game. Spaceships were coded to reset back to their initial position if they reached the finish line, while also adding points to the respective player. There were also changes in the ball collision and spawning logic, as it was intended to be used as the base for the meteors. Race time was also implemented. Finally, the shape of the paddles was changed to a more spaceship-like one. In general, this changes preserved the code structure from the Pong video-game.

III. RESULTS

The video-game is made up of four scenes, those being the game itself, the main menu, a restart menu and an Easter egg screen. The main menu (Figure 3) provides the user options to start the game in two different modes: singleplayer (pressing S key) and multiplayer (pressing A key). When choosing the singleplayer option, one of the spaceships is controlled by the machine, and what it does is move the spaceship up at constant speed all the time. When selecting the multiplayer option, two players can now play the video-game by controlling the spaceships with W/S and O/L keys. The main menu also has an option to exit the game (pressing M key), which basically terminates the program.

The restart menu (Figure 5) appears when the time is over and prints the winner (the player who made more points) and it has the options to restart the game (pressing R key) and return to main menu ((pressing M key). Finally, there is an Easter egg screen (Figure 6) that is shown whenever one of the players reaches 10 points.

FIGURE 5. Restart Menu - Space Race video-game

FIGURE 6. Ester Egg - Space Race video-game

This implementation of the Space Race video-game variant in 8086 assembly had some issues in regard to its core logic (Figure 4). The meteors do not spawn properly and the direction they have is not correct either. For this reason, there is no collisions between spaceships and meteors. Changing the shape of the spaceships was not possible.

A. Challenges faced

The problems mentioned above are related to array management issues as this structure were found to be difficult to implement. Both the meteors and the spaceship shape problems could be solved with arrays. Workarounds were tried, such as generating each meteor one by one, but this was much more time consuming and, in fact, produced more execution errors. As there was no proper meteorite system, the collisions were not implemented properly. There was also a problem with the input for one of the ships as a the O/L keys were still being detected after a press due to keyboard errors. The video-game was tested in other machines and the error did not occurred.

IV. CONCLUSION

This implementation of a video-game in 8086 assembly allows the understanding of low-level programming and video-game development:

- The use of registers and system interruptions for different tasks in 8086 assembly: by constantly working with different registers sizes and system interruptions to execute complex system operations.
- The importance of properly structuring a project and using procedures: as it is important to organize code in order to improve readability and future maintenance.
- Basic video-game design and logic: when implementing the different elements of the video-game in a cohesive way.

REFERENCES

[1] Dandamudi, S. P., Introduction to Assembly Language Programming: From 8086 to Pentium Processors. New York, NY, USA: Springer Science & Business Media, 1998.

[2] M. A. Mazidi and J. G. Mazidi, The 8086/8088 Microprocessor: Architecture, Programming, and Design, Upper Saddle River, NJ, USA: Prentice Hall, 1995.

[3] Programming Dimension, "8086 Assembly Tutorial – Getting Started," YouTube, video no. dyANHsj2UOw. [Online]. Available: https://www.youtube.com/watch?v=dyANHsj2UOw .

HOW TO COMPILE

It is important to have the .asm file and the assembler in the same folder. Here are the instruction on how to compile:

```
mount c [path to 8086 assembler & video-game]
c:
masm /a spacer\~1.asm
link spacer\~1.obj
spacer\~1.exe
```