**PROJECT REPORT**



**Fall 2022**

**CSE208L Object Oriented Programming Lab**

Shahzad Bangash (**21PWCSE1980)**

Suleman Shah (**21PWCSE1983)**

Ali Asghar (**21PWCSE2059)**

“On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work.”

Submitted to:

**Engr. Sumayyea Salahuddin**

February 6, 2023

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

**DIGITAL ELECTRONIC GAME**

## Introduction:

The **digital** **electronic** **game** is a **unity-based** **educational** game that **teaches** **players** about **digital** **logic** **circuits**. This game is **designed** to provide **players** with **hands-on experience** in **creating** and **simulating** **digital logic circuits**. The **goal** of the game is to **complete** **various** **logic** **circuit tasks** within a **given** **time** **frame**. The **game's** **focus** is on **teaching** **players** the **basic** **principles** of **digital** **electronics** and how to **apply** these **principles** to **solve** **real-world** **problems**.

## Gameplay:

This **game** **consists** of **various** **levels**, **each** with a **different** **logic** **circuit** **task**. In **each** **level**, the **player** is **presented** with a **set** of **components** and **inputs**, and **must** **use** **these** **components** to **create** a **working** **logic** **circuit** **that** **meets** the **specified** **requirements**. The **player** has a **limited** **amount** of **time** to **complete** **each** **task**, **which** **adds** an **element** of **challenge** and **excitement** to this **game**.

## Components:

This **game** includes **components**, such as **AND** **gates**, **OR** **gates**, **NOT** **gates**. **Players** are **able** to **connect** **these** **components** together to **create** **more** **complex** **circuits**. Each **component** is **animated** to **show** its **state**, which **helps** **players** **understand** how the **circuit** **works** and what is **happening** at each **stage**.

## Simulation:

The game **features** an **advanced** **simulation** **engine** that **allows** **players** to **test** their **circuits** and see the **results** in **real-time**. This **allows** **players** to **experiment** with **different** **circuit** **configurations** and see the **effect** on the **outputs**. The **simulation** **engine** is **highly** **accurate** and **provides** a **realistic** **representation** of **how** **real-world** digital **circuits** **behave**.

## Time Limits:

Each level has a specific time limit, and the player must complete the task within this time frame to progress to the next level. This adds an element of challenge and encourages players to think quickly and work efficiently. The time limits can be adjusted to suit the player's skill level and provide a greater or lesser degree of difficulty.

## Learning Outcome:

The digital electronics game provides players with an interactive and engaging way to learn about digital electronics. The game is designed to teach players the fundamental principles of digital circuits, such as Boolean algebra, gate logic, and circuit simulation. By playing the game, players are able to build their own circuits and see the results of their work, which reinforces their understanding of the subject and helps them retain the information.

## Framework Used:

**Unity Engine** and **C#.**

## Role and Usage of Oop in Project:

* Objects and Classes(Object Oriented Programming)
* Inheritance
* Abstraction & Encapsulation
* Static Classes

## Game Flow Chart:

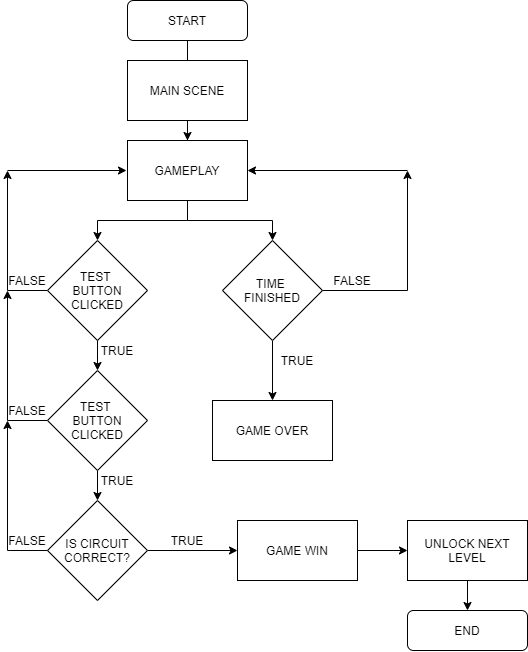


Figure 1

## Code Screenshots:

**MANAGER CLASS**

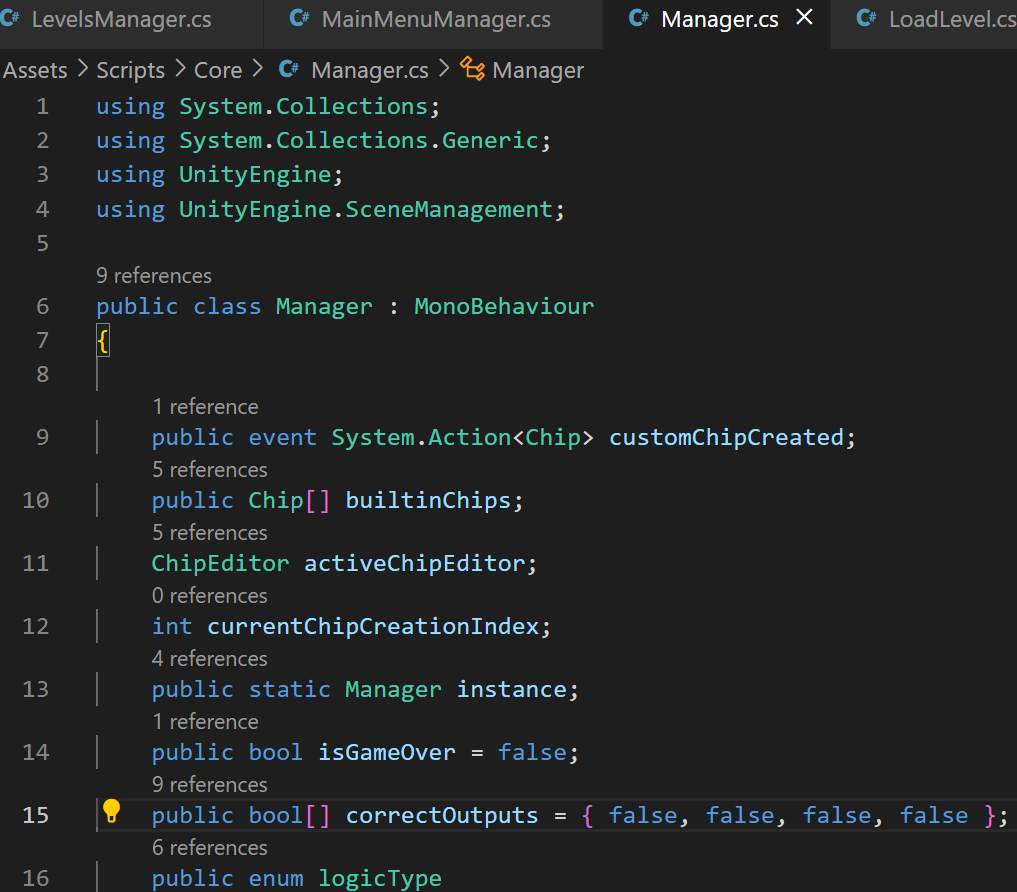
****

Figure 2

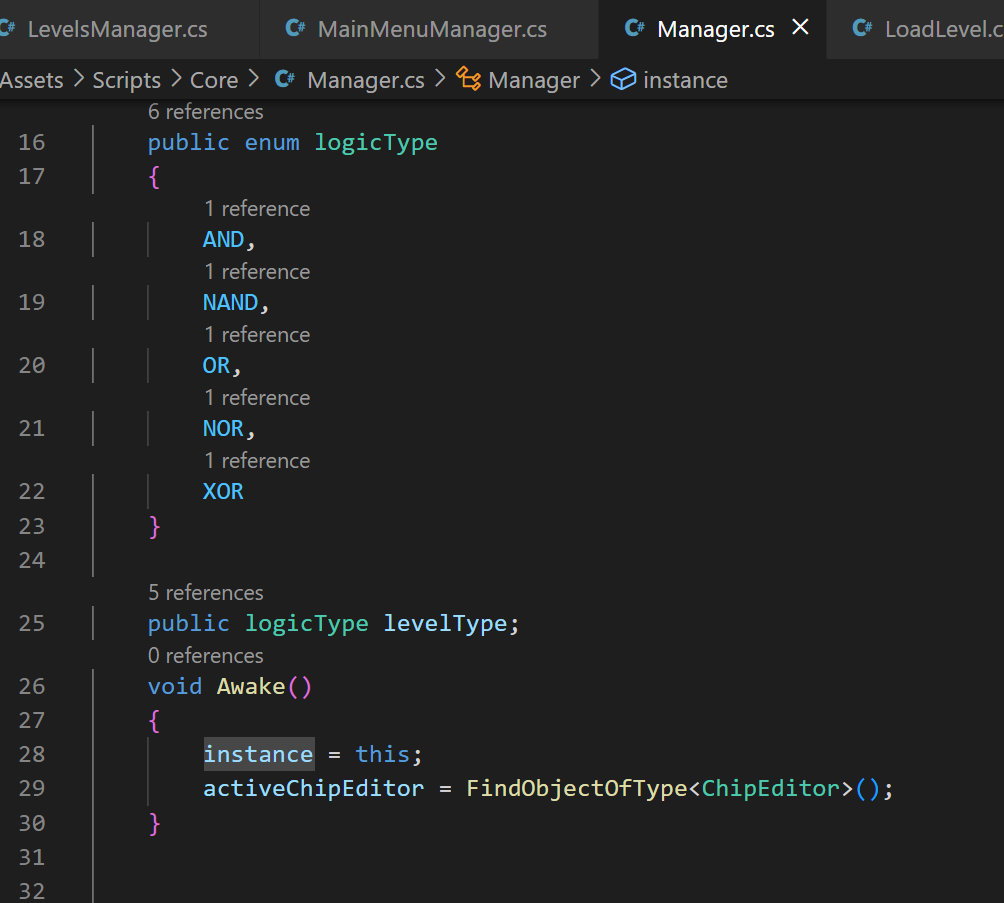
****

Figure 3

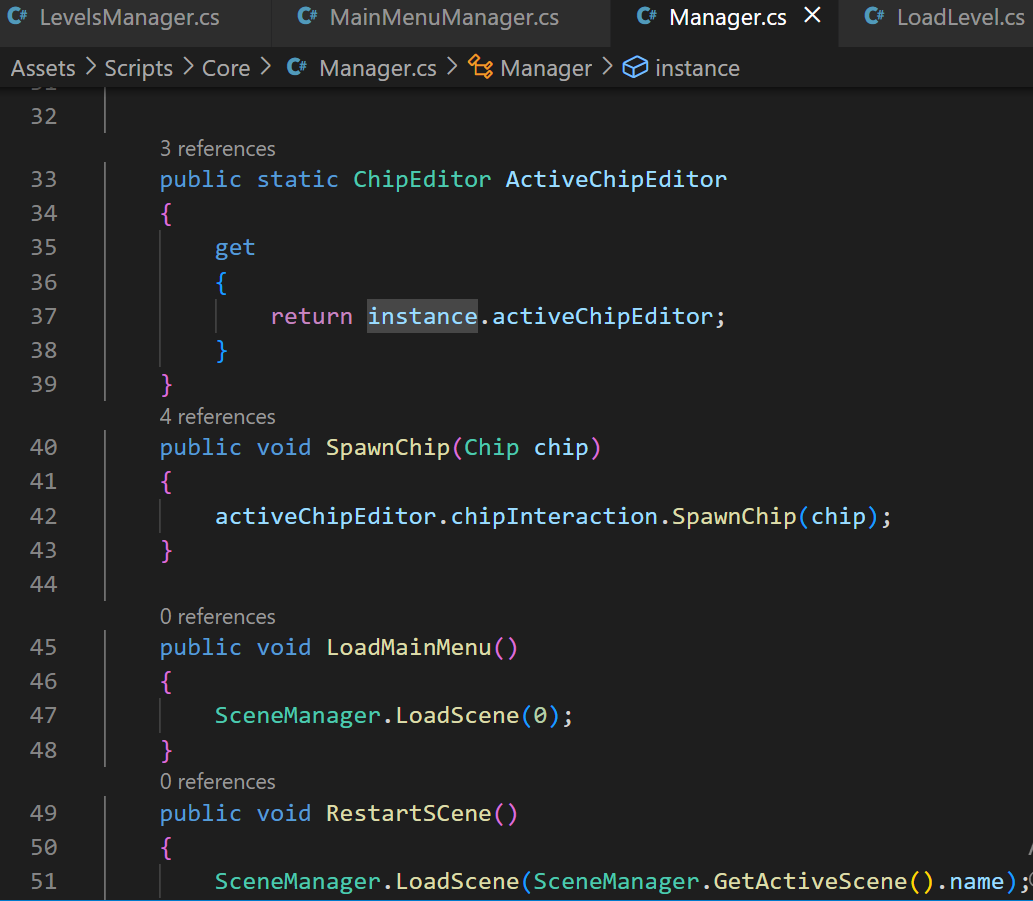
****

Figure 4

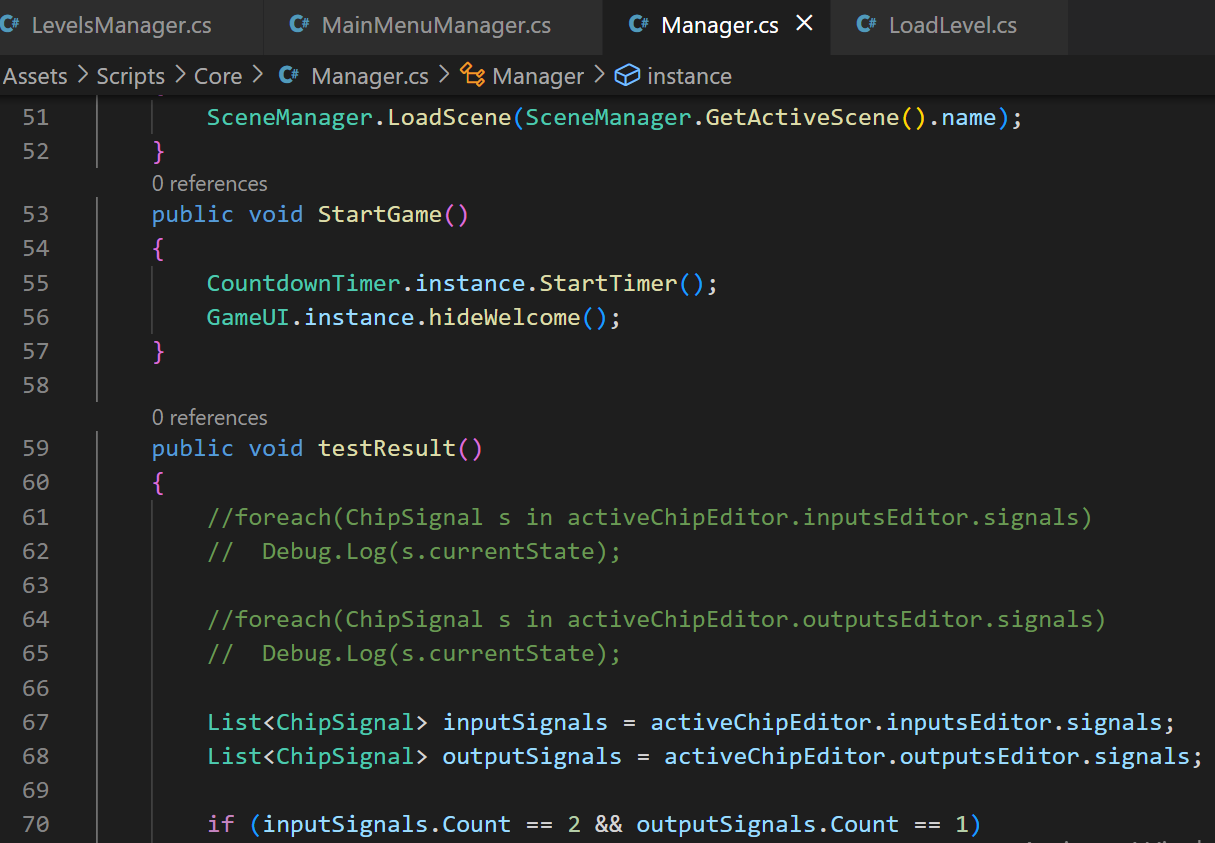
****

Figure 5

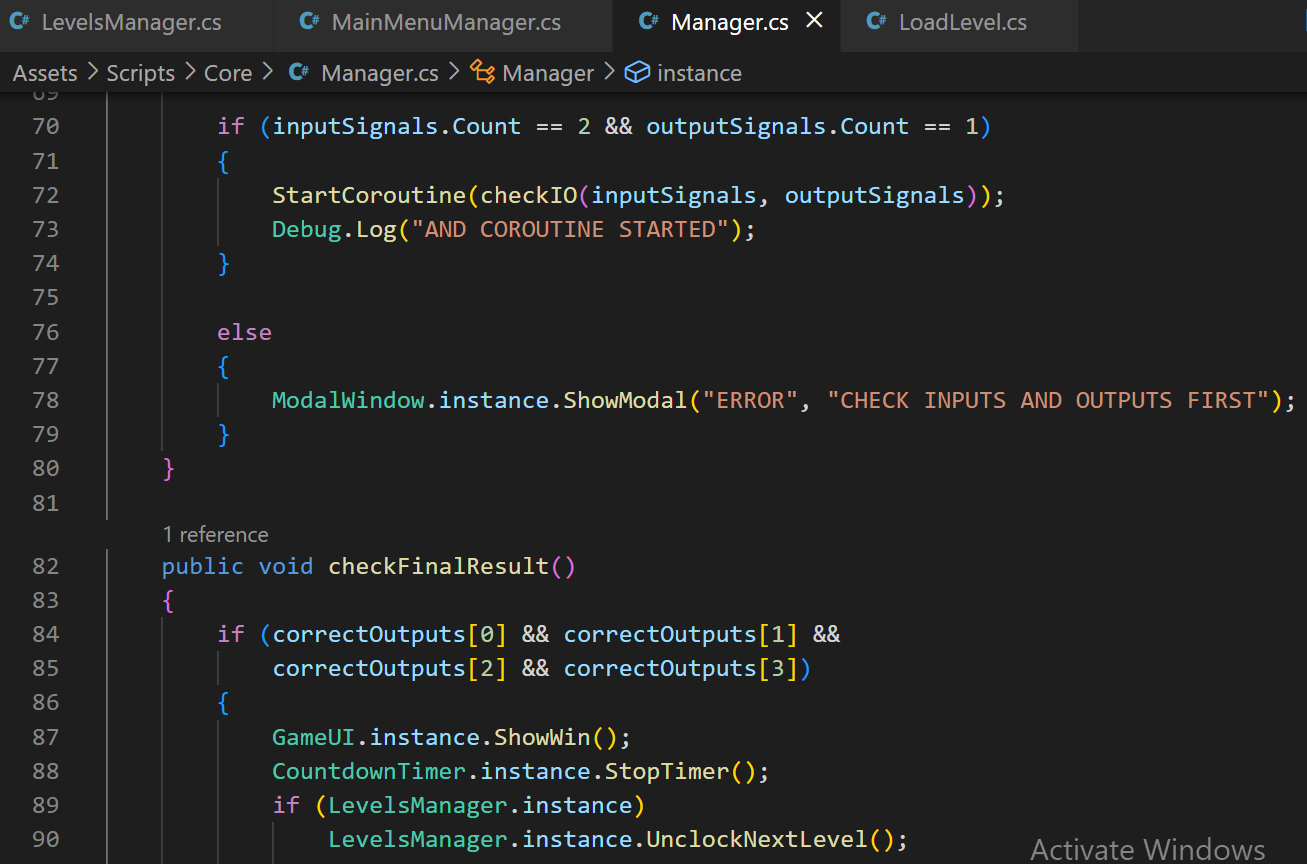
****

Figure 6

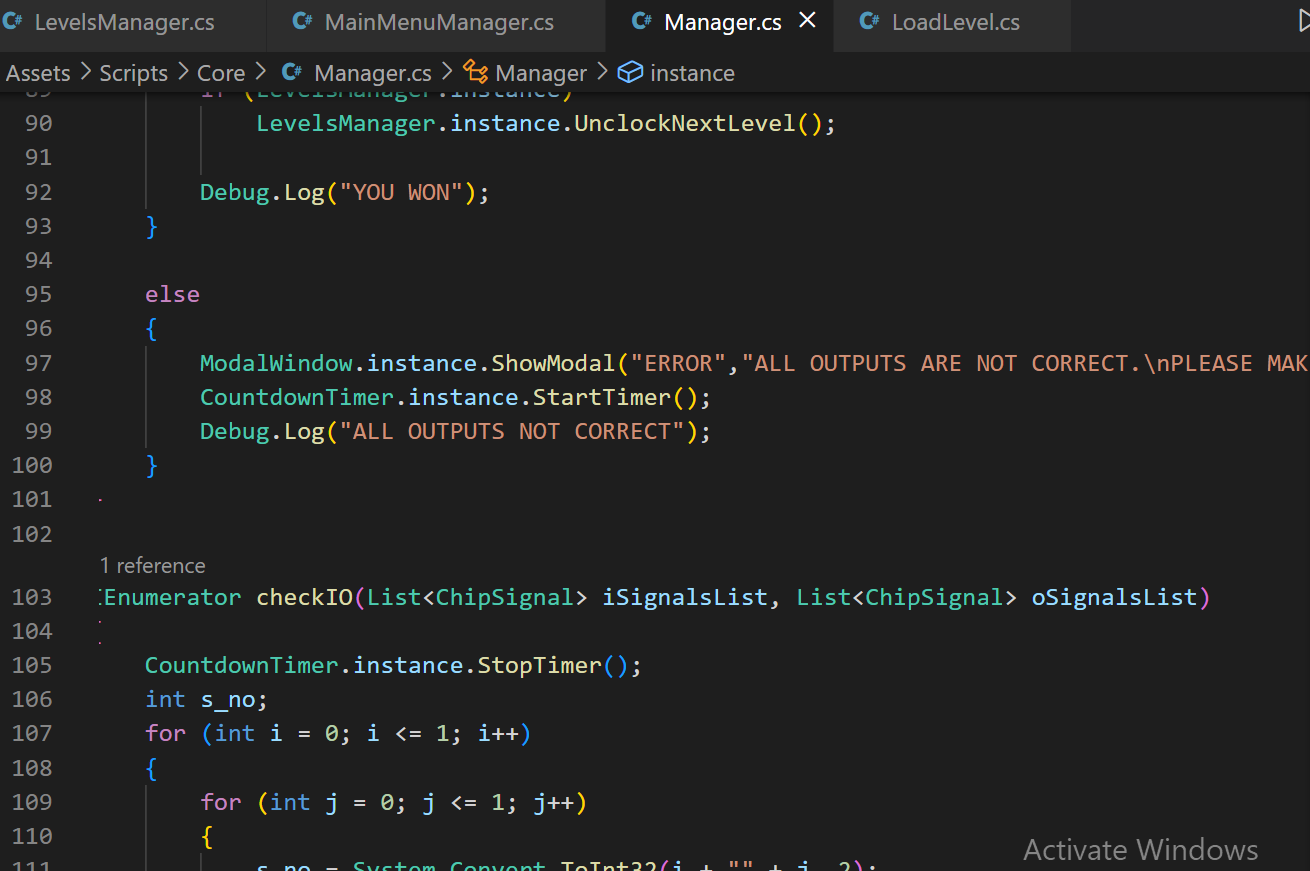
****

Figure 7

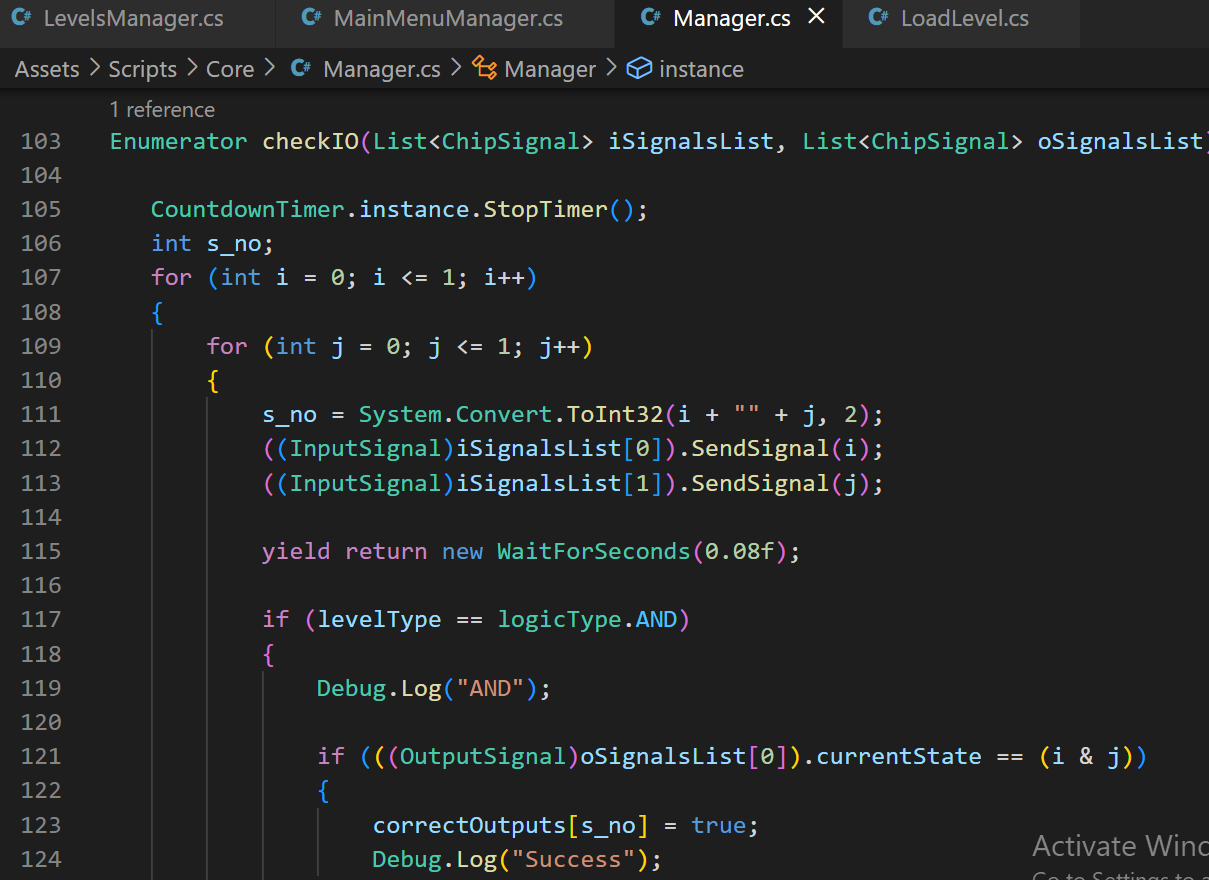
****

Figure 8

****

Figure 9

****

Figure 10

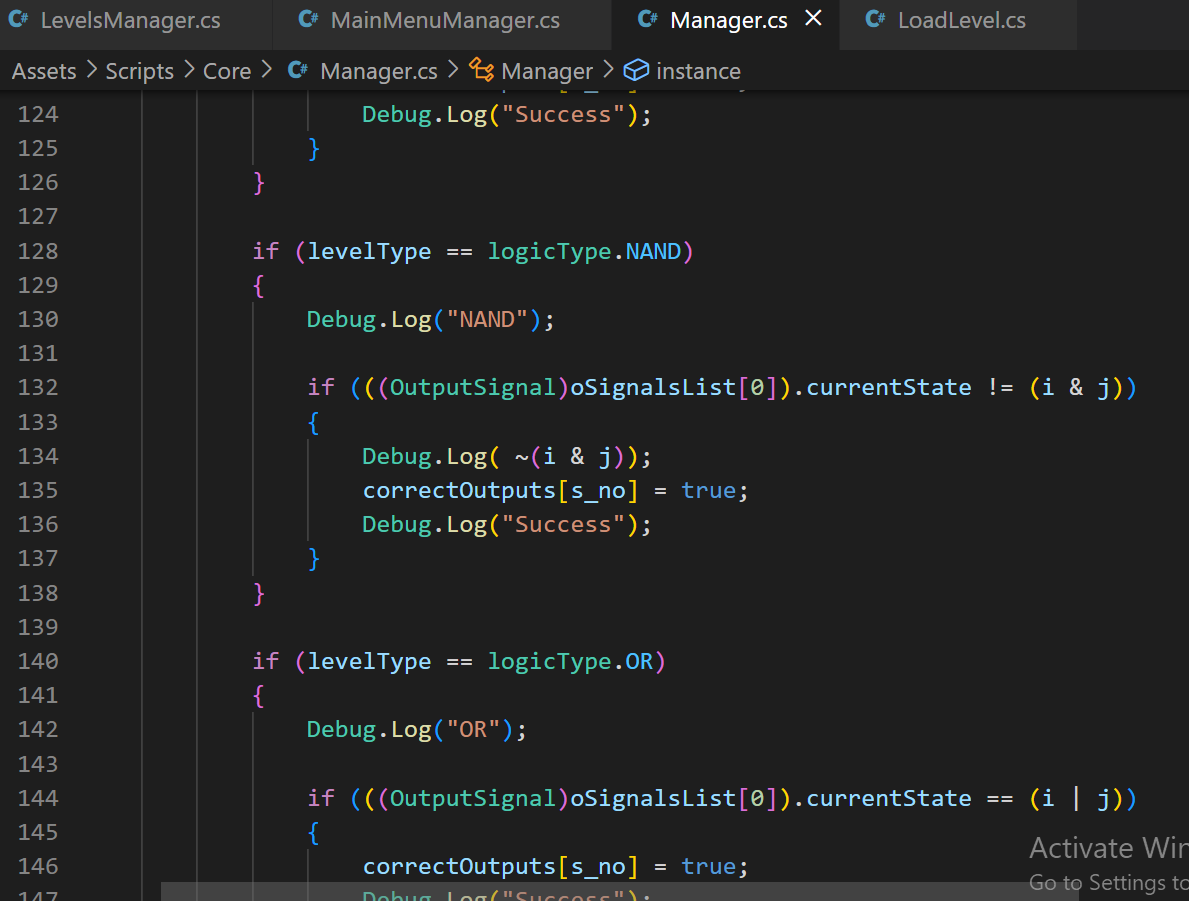
****

Figure 11

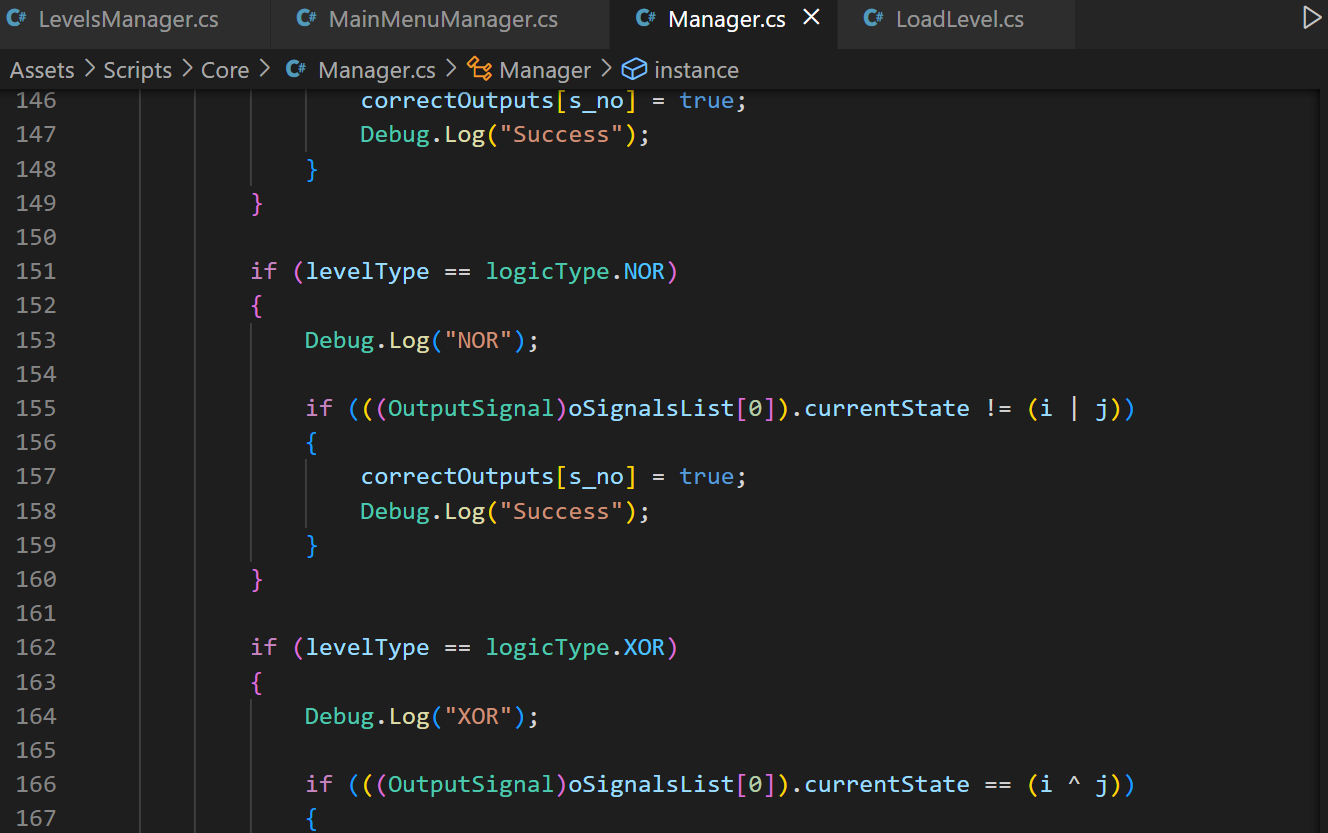
****

Figure 12

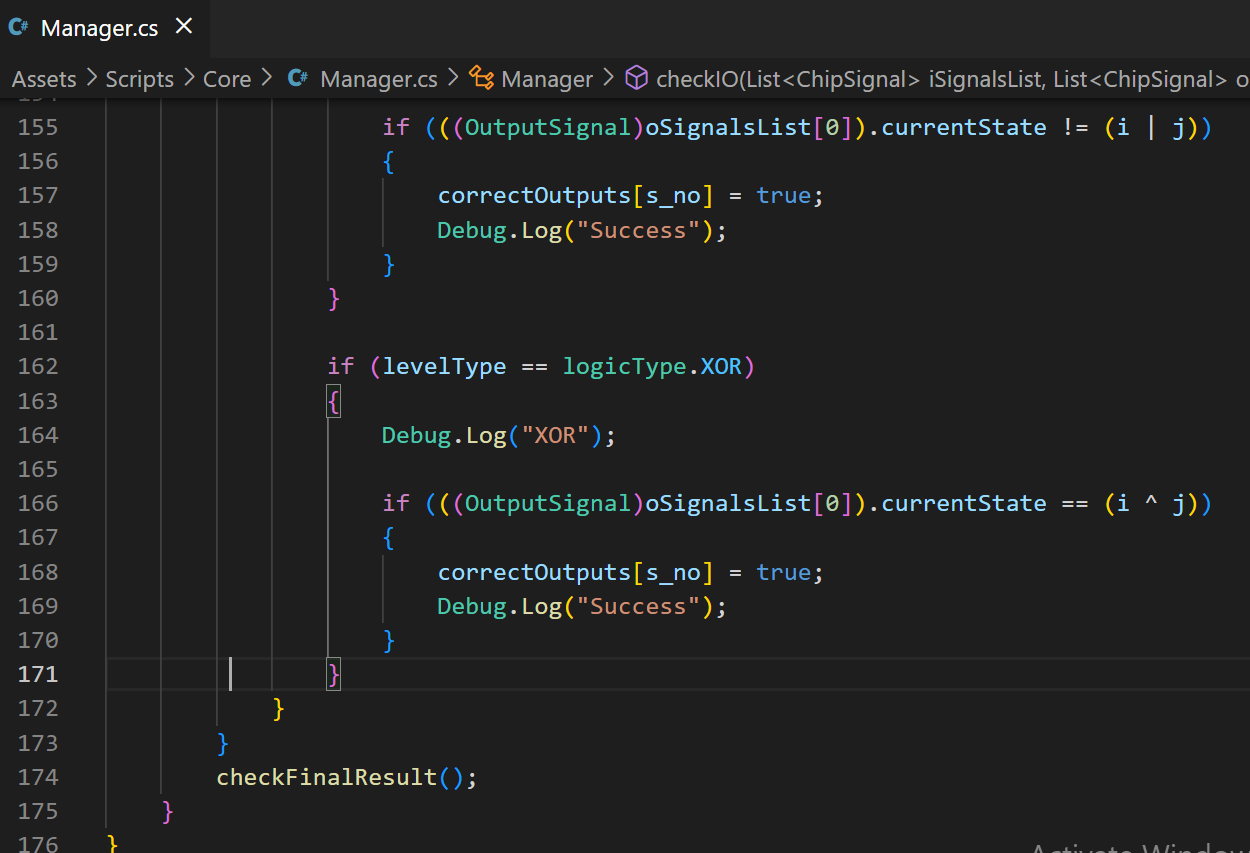
****

Figure 13

**AND GATE CLASS**

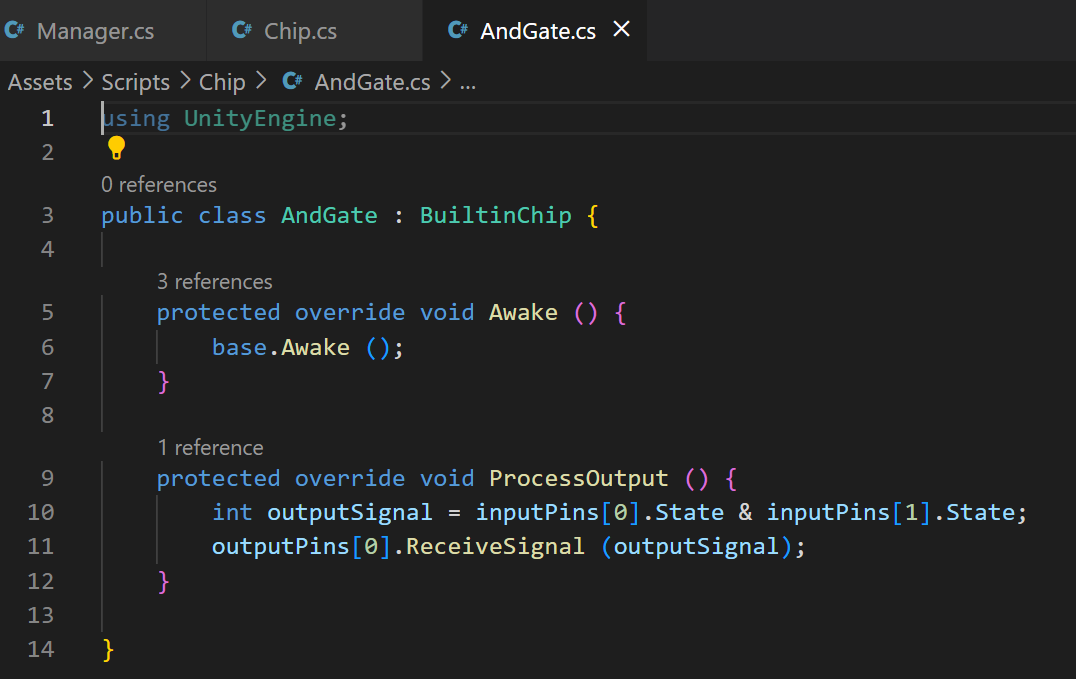
****

Figure 14

**NOT GATE CLASS**

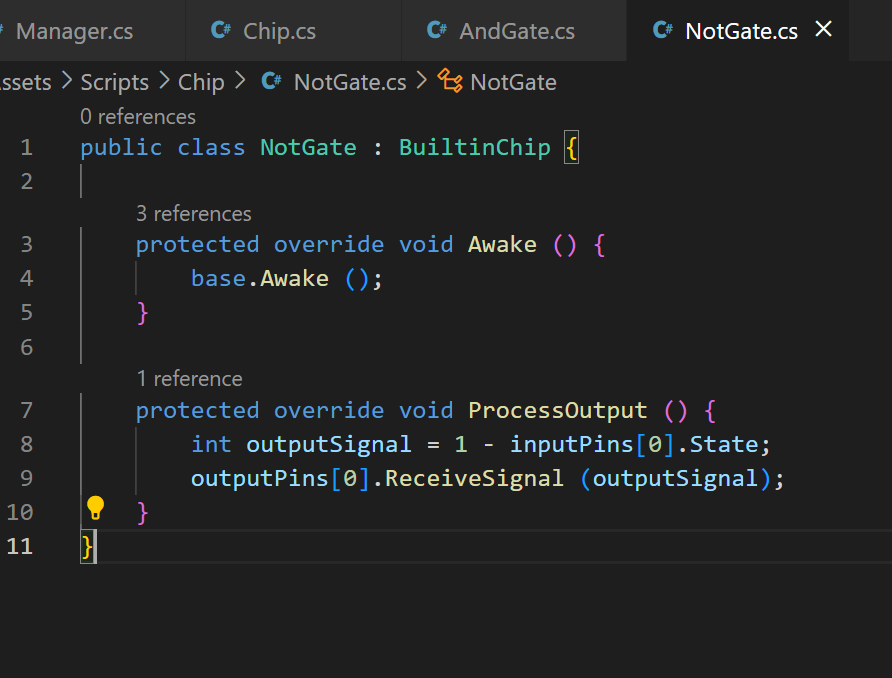
****

Figure 15

**OR GATE CLASS**

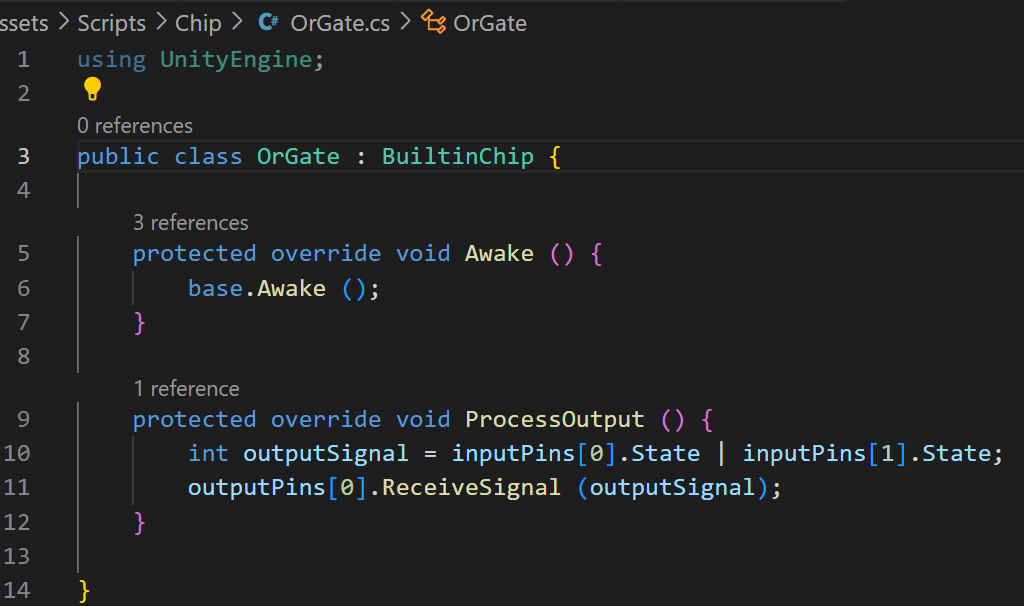
****

Figure 16

**COUNT DOWN TIMER CLASS**

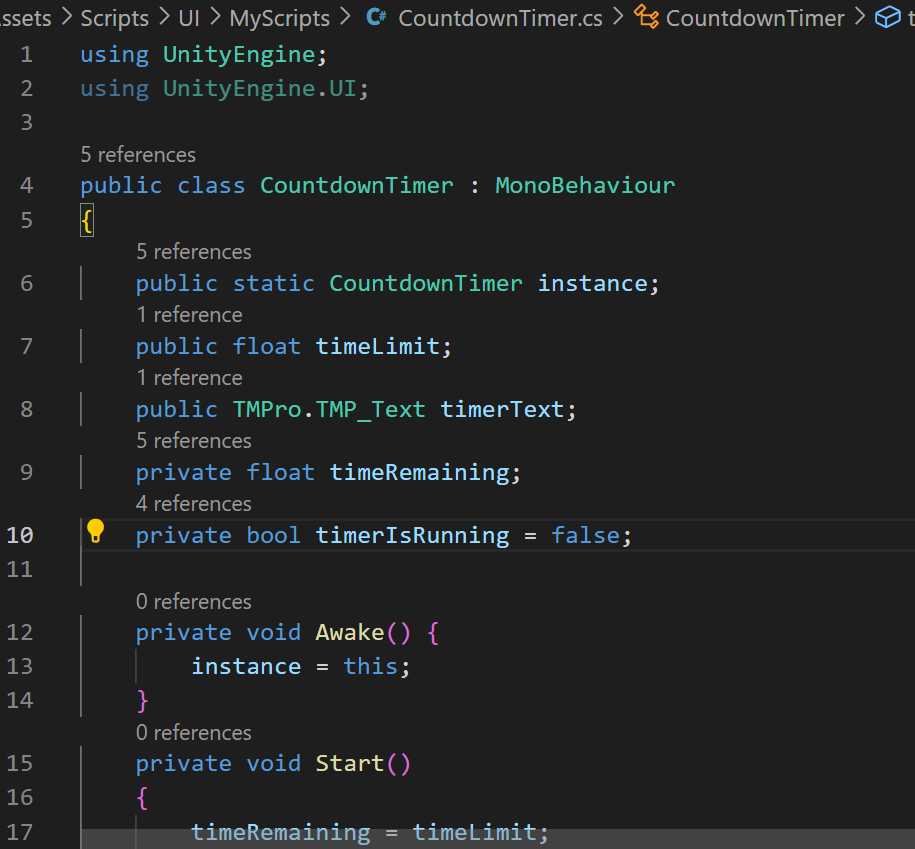
****

Figure 17

****

Figure 18

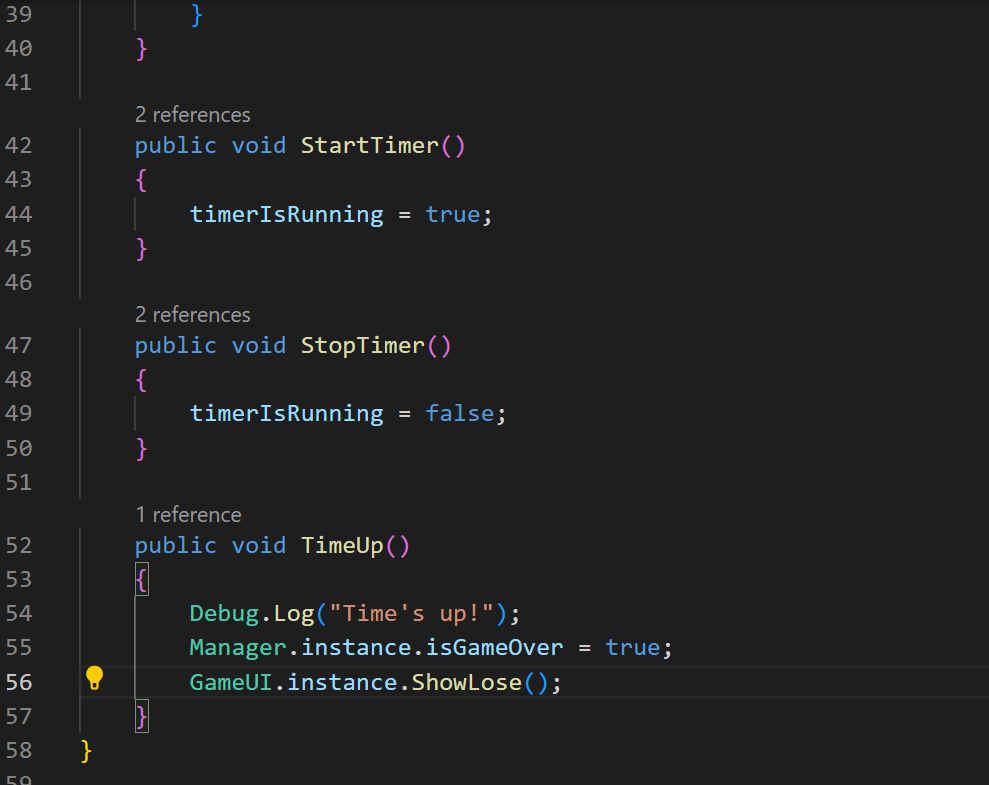
****

Figure 19

**END GAME SCREEN CLASS**

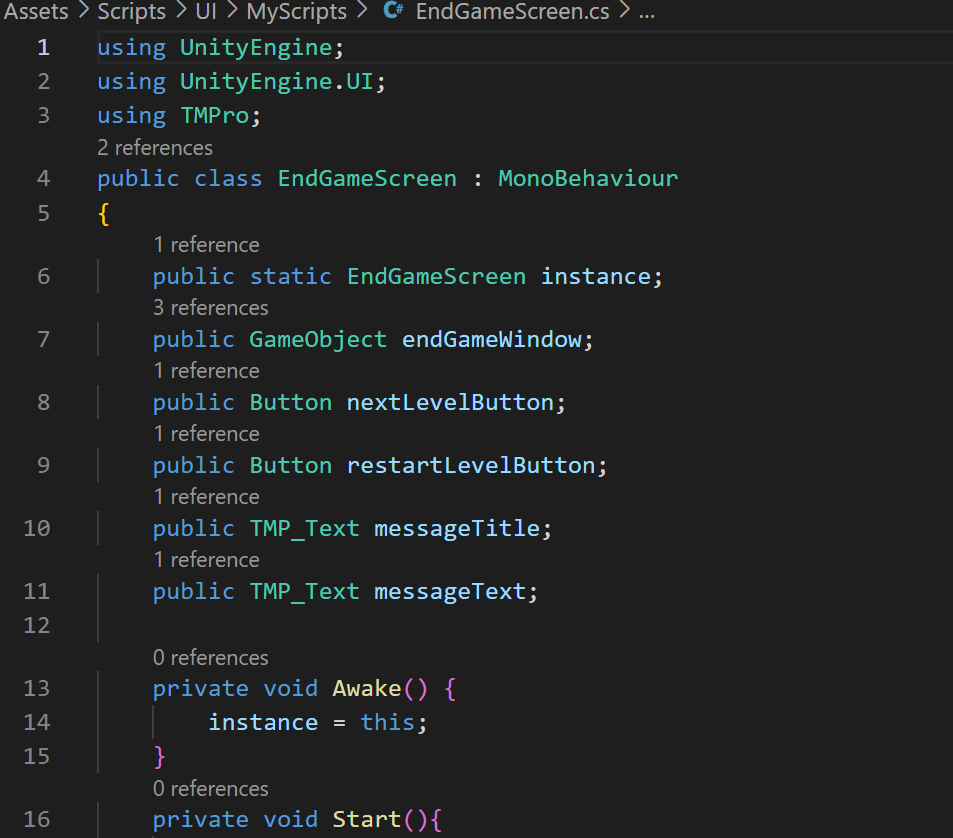
****

Figure 20

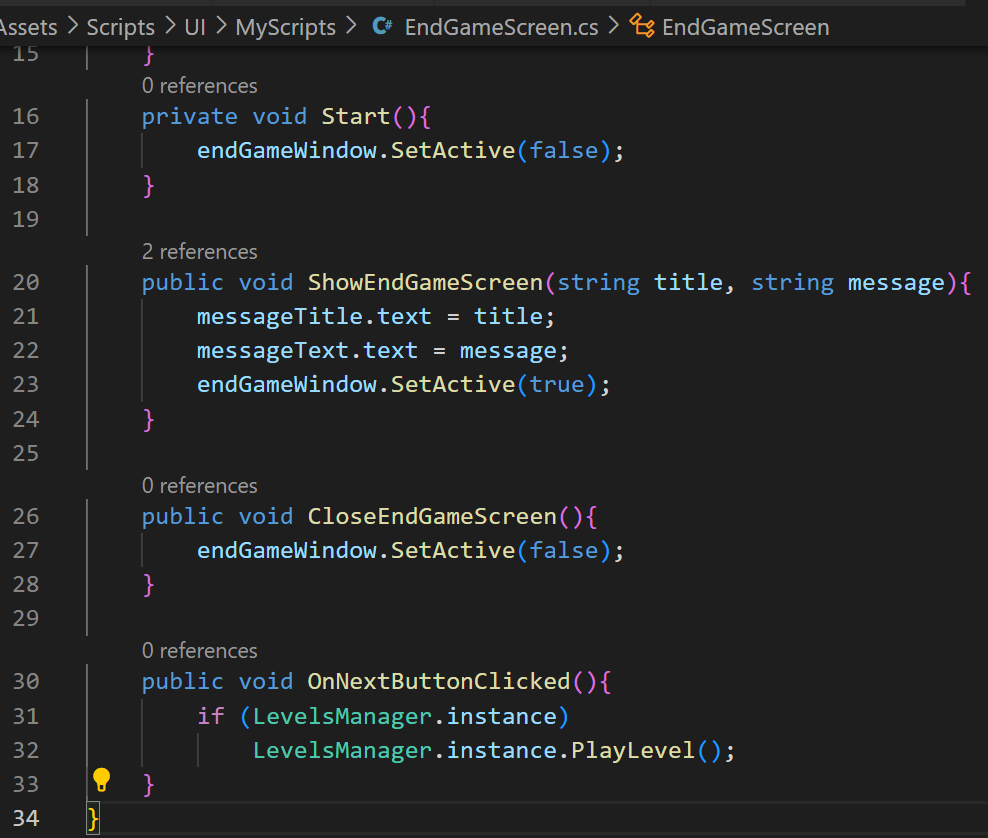
****

Figure 21

**MODAL WINDOW CLASS**

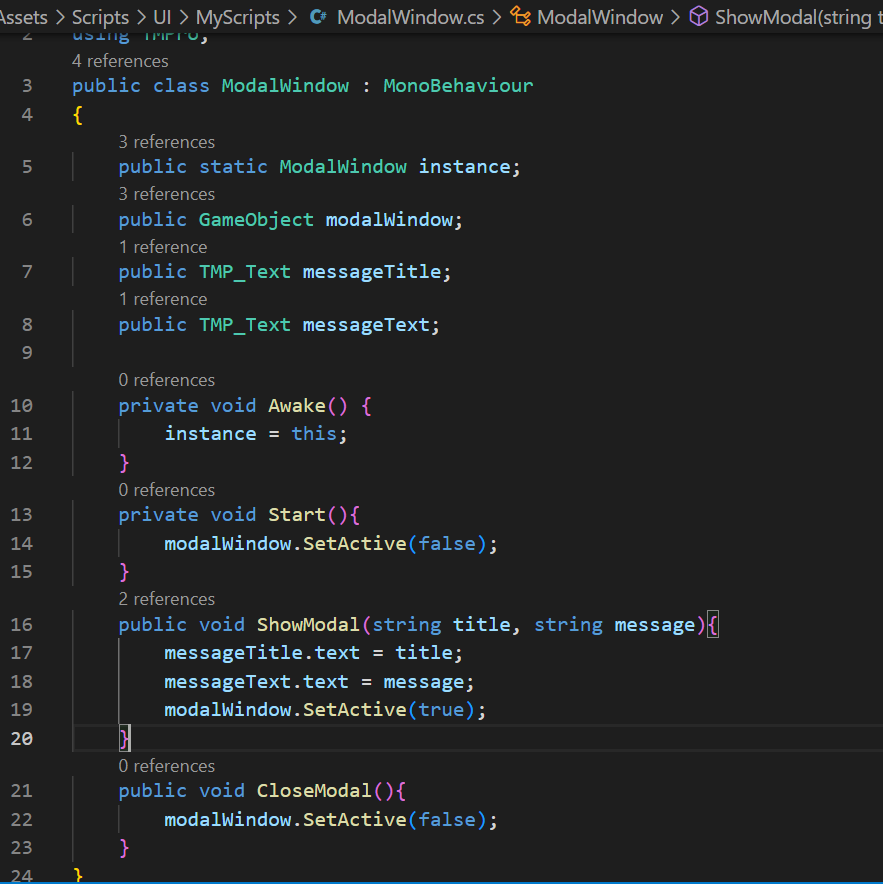
****

Figure 22

## Code Explanation:

**MANAGER CLASS**

**Manager Class** (shown in figure 2-13) **inherits** from the "**MonoBehaviour**" **base** **class** provided by **Unity**.

This **class** **defines** a number of **public** **variables** and **properties**, including an **array** of "**Chip**" **objects**, a **reference** to the active "**ChipEditor**" **object**, a **current** **chip** **creation** **index**, a **static** **reference** to the **instance** of the "**Manager**" class, a **boolean** indicating whether the **game** is **over**, an **array** of **correct** **output** **values**, and an **enumeration** of **logic** **types**.

The "**Awake**" **function** is called when the **object** is **created**, and **sets** the **static** "**instance**" variable to the **current** object, and **finds** the active "**ChipEditor**" **object** in the scene.

The "**SpawnChip**" **function** takes a "**Chip**" object as a **parameter** and **spawns** it in the **scene** using the "**chipInteraction**" **component** of the **active** "**ChipEditor**" object.

The "**LoadMainMenu**" function **loads** the **main menu** scene. The "**RestartScene**" function **restarts** the current scene. The "**StartGame**" function **starts** the **game** by **calling** the "**StartTimer**" function of the "**CountdownTimer**" **object** and **hiding** the **welcome** **message** in the **UI**.

The "**testResult**" function **checks** the **input** and **output** **signals** of the **active** "**ChipEditor**" **object**, and if there are **two** **input** **signals** and **one** **output** **signal**, it **starts** a **coroutine** called "**checkIO**" with the **input** and **output** **signals** as **parameters**.

The "**checkFinalResult**" function **checks** **whether** the **correct** **outputs** **array** is equal to the **expected** **values**, and **shows** the **win** **screen** and **unlocks** the **next** **level** if all **outputs** are **correct**.

The "**checkIO**" **coroutine** **loops** through all **possible** **combinations** of **input** **signals**, **sends** the **input** **signals** to the **active** **chip**, **waits** for a short **period** **of** **time**, **checks** the **output** **signal** **against** the **expected** **value** for the **level** **type**, and **sets** the **appropriate** **value** in the **correct** **outputs** **array**.

The "**levelType**" **enum** is **used** to **specify** the **type** of **logic** for the **current** **level**, and is **used** in the "**checkIO**" **coroutine** to **determine** the **expected** **output** **value** for each **input** **combination**.

**AND, NOT AND OR GATE CLASSES**

**And, Not and Or Gate Classes** (shown in figure 14,15 & 16 respectively) are used for **making built-in logic chips** in this **game**.

All **three** **classes** **inherit** from the **BuiltinChip** **class**, which provides common **functionality** for all **built-in chips** in the game. The **Awake**() **method** is **overridden** from the **base** **class** and it calls the **Awake** **method** of **base** **class**(**BuiltinChip**). The **ProcessOutput**() **method** is called when the chip **needs** to **produce** an **output** **signal** based on the **input** **signals**. In case of **AndGate** class, the **ProcessOutput**() **method** **calculates** the **output** **signal** by **performing** a **bitwise** **AND** **operation** on the **two** **input** **signals**. For **OrGate**, it **performs** a bitwise **OR** **operation** on the **two** **inputs**, while for **NotGate**, **inverse** of the **input** **signal** is **calculated** by **subtracting** it from **1**, which **results** in a **value** of **0** if the **input** **signal** is **1**, and a **value** of **1** if the **input** **signal** is **0**. These are then stored in **inputPins[0]** and **inputPins[1].** The **result** of the **bitwise** **operations** is **stored** in the **outputSignal** **variable**. The **output** **signal** is then sent to the **outputPins[0]** **pin** using the **ReceiveSignal**() **method**.

**COUNT DOWN TIMER CLASS**

**Count Down Timer Class** (shown in figure 17, 18 & 19) is **responsible** for **implementing** a **countdown** **timer** in this game. The **CountdownTimer** class contains a **static** **instance** of itself that can be **accessed** from other **scripts**. The **timeLimit** **public** **float** **variable** is used to **set** the **total** **time** **limit** for the **countdown** **timer**. The **timerText** **public** **TMP\_Text** **variable** is **used** to **display** the **remaining** **time** on the **screen** using the **Text Mesh Pro component.** The **timeRemaining** **private** **float** **variable** **keeps** **track** of the **remaining** **time** during the **countdown**. The **timerIsRunning** **private** **bool** **variable** is **used** to **control** **whether** the **timer** is **running** or **not**. The **Awake**() **method** **sets** the **instance** of the **script** to **itself**, allowing **other** **scripts** to **access** **it**. The **Start**() **method** **initializes** the **timeRemaining** **variable** with the **value** of **timeLimit**. The **Update**() **method** is called **every** **frame** and is **responsible** for **updating** the **remaining** **time** and **displaying** it on the **screen**. If the **remaining** **time** is **zero** or **less**, the **timer** **stops** and **calls** the **TimeUp**() **method**. The **StartTimer**() **method** is a **public** **method** that can be **called** to **start** the **countdown** **timer**. The **StopTimer**() **method** is a **public** **method** that can be **called** to **stop** the **countdown** **timer**. The **TimeUp**() **method** is a **public** **method** that **handles** the **event** when the **time** is **up**. In this case, it sets the **isGameOver** **flag** in the **Manager** **script** to **true** and **displays** the "**lose**" **screen** **using** the **GameUI** **script**. The **TimeUp**() **method** also **logs** a **message** to the **console** **indicating** that the **time** is **up**. The **remaining** **time** is **displayed** in **minutes** and **seconds** **format** **using** the **string.Format**() **method**.

**END GAME SCREEN CLASS**

**End Game Screen Class** (shown in figure 20 & 21) is used to **create** an **end** **game** **screen** in this **project**.

This script **contains** a **public** **GameObject** **variable** named "**endGameWindow**" that **represents** the **end** **game** **screen** in the **scene**.

It also contains **several** **public** **Button** **variables** named "**nextLevelButton**" and "**restartLevelButton**" that **represent** the **buttons** in the **end** **game** **screen**.

This script **contains** **two** **public** **TMP\_Text variables** named "**messageTitle**" and "**messageText**" that represent the **title** and **message** **text** in the **end game screen**.

It has a **static** **instance** **variable** named "**instance**" that allows the **end game screen** to be **accessed** from **other** **scripts** in the **project**.

In the **Awake** **method**, the **instance** variable is **set** to "**this**", which **allows** the **script** to be used as a **singleton**.

In the **Start** **method**, the **endGameWindow** **object** is initially **set** to be **inactive**, so it does not **show** **up** on the **screen** when the **scene** **starts**.

The **ShowEndGameScreen** **method** is a **public** **method** that can be **called** from **other** **scripts** in the project. It takes **two** **string** **parameters**, "**title**" and "**message**", which represent the **title** and **message** **text** to be **displayed** in the **end** **game** **screen**.

**Inside** the **ShowEndGameScreen** **method**, the **messageTitle** and **messageText** **variables** are **set** to the corresponding **string** **parameters** **passed** into the **method**. **Finally**, the **endGameWindow** **object** is **set** to be **active**, which causes it to be **displayed** on the **screen** with the **updated** **title** and **message** **text**.

The **CloseEndGameScreen** **method** is also a **public** **method** that can be **called** from **other** **scripts** in the **project**. It simply **sets** the **endGameWindow** **object** to be **inactive**, which hides the **end** **game** **screen** from the **screen**.

The **OnNextButtonClicked** **method** is a **public** **method** that is **called** when the "**next** **level**" **button** is **clicked** in the **end** **game** **screen**. It **checks** if the **LevelsManager** **singleton** **instance** **exists** and if it does, **calls** the **PlayLevel** **method**, which **starts** the **next** **level** of the **game**.

Similarly, the **OnRestartButtonClicked** **method** is a **public** **method** that is **called** when the "**restart** **level**" **button** is **clicked** in the **end game screen**. It also **checks** if the **LevelsManager** **singleton** **instance** **exists** and if it **does**, calls the **RestartLevel** **method**, which **restarts** the **current level** of the **game**.

**MODAL WINDOW CLASS**

**Modal Window Class** (shown in figure 22) is used to create a **modal window** in the Unity engine.

The script contains a public **GameObject** **variable** named "**modalWindow**" that represents the **modal** **window** in the scene.

The script also contains two **public** **TMP\_Text** **variables** named "**messageTitle**" and "**messageText**" that represent the **title** and **message** **text** in the **modal** **window**.

This script has a **static** **instance** variable named "**instance**" that allows the **modal** **window** to be accessed from **other scripts** in the project. In the **Awake method**, the **instance** variable is set to "**this**", which allows the script to be used as a **singleton**.

In the **Start method**, the **modalWindow** object is initially set to be **inactive**, so it does not show up on the screen when the **scene** **starts**.

The **ShowModal** **method** is a **public** method that can be called from **other scripts** in the project. It takes **two string parameters**, "**title**" and "**message**", which represent the **title** and **message** **text** to be **displayed** in the **modal window**. **Inside** the **ShowModal method**, the **messageTitle** and **messageText** **variables** are set to the corresponding **string** **parameters** **passed** into the **method**.

Finally, the **modalWindow** **object** is set to be **active**, which **causes** it to be **displayed** on the **screen** with the **updated** **title** and **message** text.

The **CloseModal** **method** is also a **public** **method** that can be **called** from **other** **scripts** in the project. It simply **sets** the **modalWindow** **object** to be **inactive**, which **hides** the **modal** **window** from the **screen**.

## Game Screenshots:

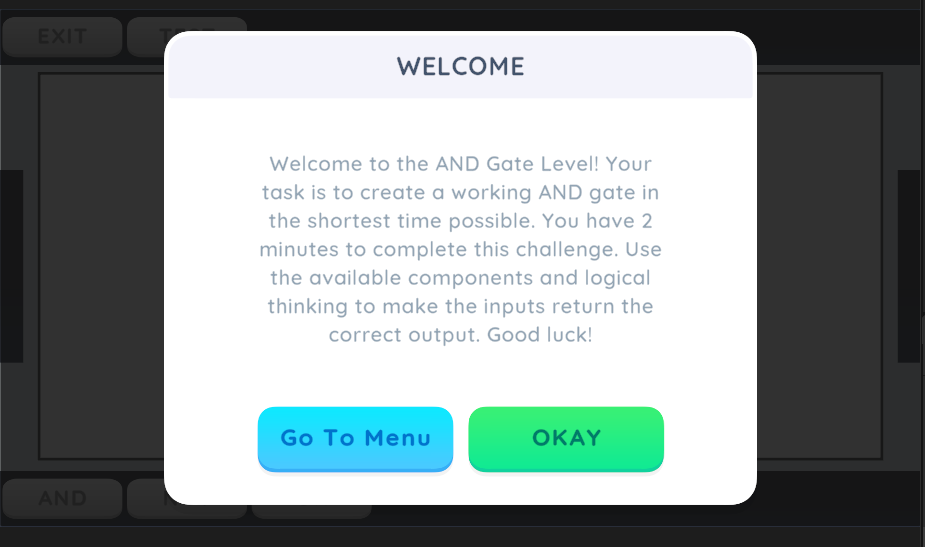


Figure 23

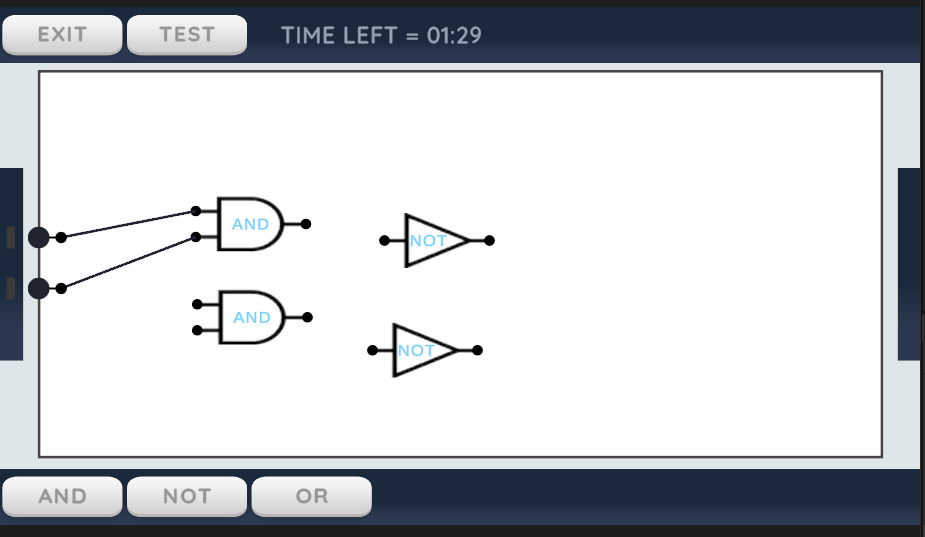


Figure 24

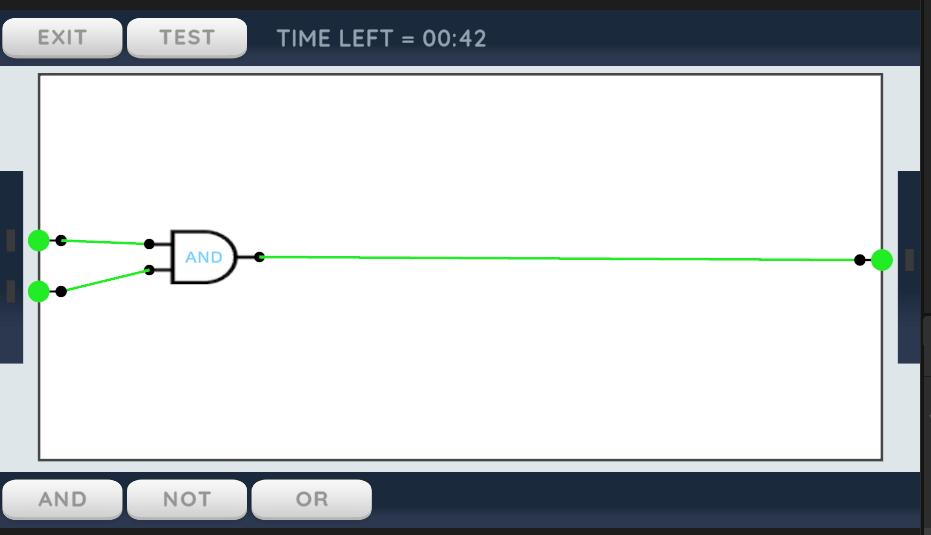


Figure 25

## Conclusion:

The digital electronics game is a fun and educational tool that can help players learn about digital electronics in a fun and interactive way. The game is built using the unity platform, which provides a rich and immersive experience for players. The combination of hands-on learning, simulation, and time limits makes the game an ideal tool for teaching digital electronics and providing a fun and engaging way for players to learn.