# Tetris (Digital Circuits Project Implemented In Proteus)

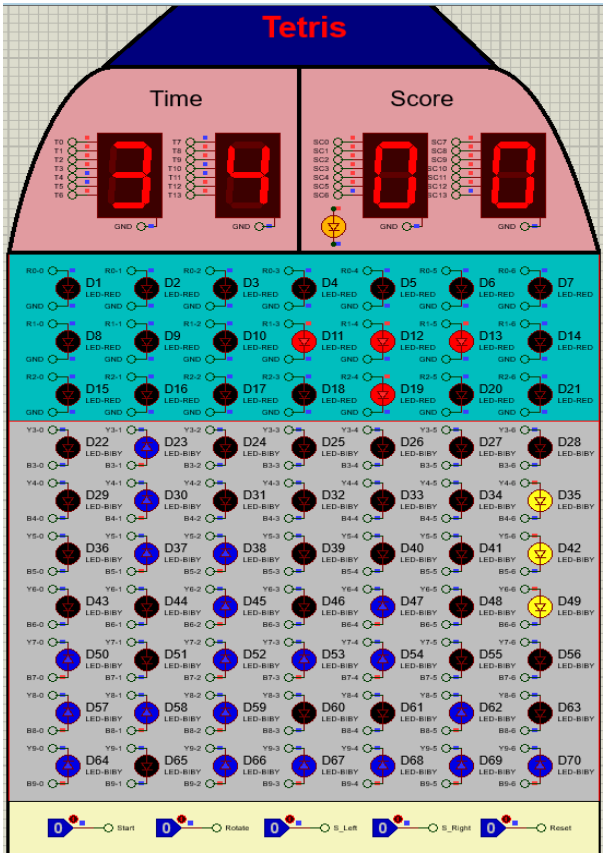
## Introduction

This game is a digital block-stacking puzzle where the player's goal is to strategically guide and align falling blocks to complete rows within a limited time. The game is played on a 10x7 matrix, with blocks randomly generated in the top three rows. Players can control the position and orientation of these blocks using shift and rotation keys.

As rows are completed, they will flash for two seconds before being cleared, causing all blocks above them to shift downward. Players earn points for each cleared row, with the aim of reaching a total of three points to win the game. However, the game ends if:

- A block collides with a fixed block in the top three rows and cannot move out of the area.
- The timer reaches 99 seconds.

> NOTE:
>
> This document explains the overall project, the idea behind its implementation, and the functionality of each section.
>
> **However if you want to jump straight to install and running the game, use the table of contents below.**
>
> You can download a sample gameplay by clicking **here**.



# Table of Contents

# Gameplay Overview

## General Rules and Interface Setup

The game interface consists of an 10x7 grid of LED , four seven segment responsible for displaying time and score and five control buttons which are Start , Reset , Rotate , S_Right and S_Left.

In the interface, each LED can either be on or off. The LEDs in the first three rows, when lit, are red, while the LEDs in rows four to ten can be either yellow or blue.

Yellow LEDs indicate a moving block, whereas blue LEDs represent a fixed block. If any part of a moving block collides with the bottom of the grid or with a fixed block, the entire moving block becomes fixed at the point of collision.

**End Condition:**

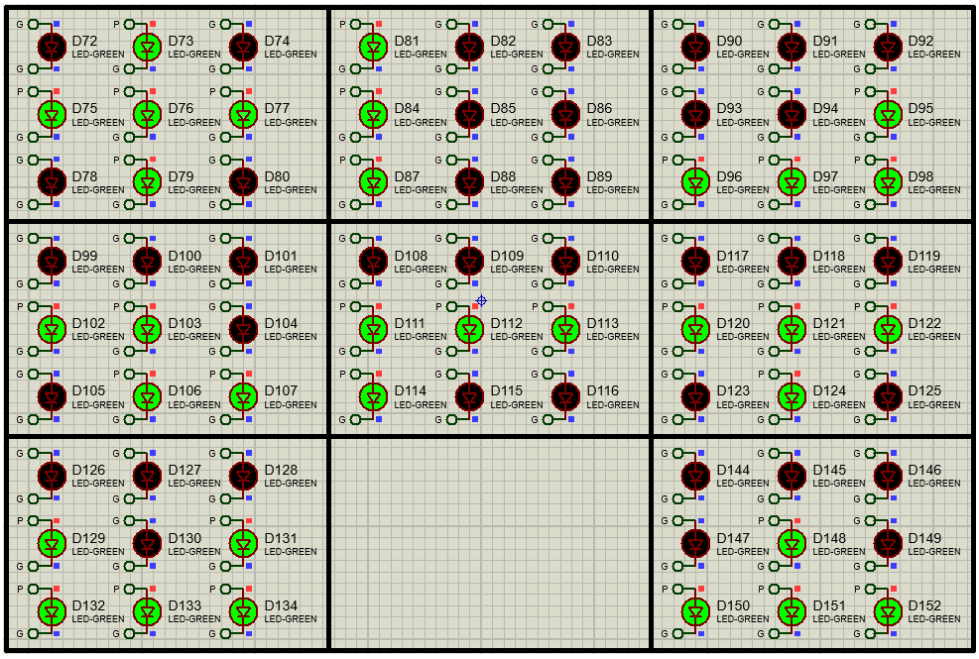
**Winning Condition:** The player wins by successfully collecting 3 points.

**Losing Conditions:**

1. The player loses if a generated block collides with a fixed block while exiting the top 3 rows and cannot fully leave the area.
2. The player loses if the game time reaches 99 seconds.

## Starting the Game

After pressing the start button, the game begins, and the time and score values on the 7-segment displays reset to zero. One of the eight predefined 3x3 blocks is randomly generated and placed in a random position within the top three rows. (All eight predefined 3x3 blocks are displayed in the picture below.)



For the first three seconds, while the block is still in the top three rows, the player can use the shift and rotate buttons to change the block's position and orientation. Once the block completely moves out of the top three rows, a new block is generated, and this process continues until the game ends.

## Player Interactions

- **Reset:** Clicking this option resets the game and user interface to a ready state.
- **Start:** Clicking this option starts the game, resets the score and time displays to zero, and initiates the random generation of blocks. This button is only active when the game has been reset and is ready to start.
- **Rotate:** When this key is pressed, the block in the top three rows rotates counterclockwise.
- **S_Right:** When the key is pressed, the block in the top three rows shifts right, if possible.
- **S_Left:** When the key is pressed, the block in the top three rows shifts left, if possible.

## Objective of the Game

The objective of the game is to arrange the blocks so they align in a single row. When a row of LEDs is completed, all the LEDs in that row will blink for two seconds before turning off. After that, all the blocks above the completed row will shift downward, and the player will receive one point added to their total score. The player wins by accumulating 3 points.

# Implementation Idea

## Before Game Starts

After running the simulation and before pressing the start button, the last two digits of the student ID numbers of the two project group members will alternately blink on the four 7-segment displays.

At the same time, the LEDs on the game matrix will illuminate in a spiral pattern from top to bottom, creating an effect similar to a light show. Please see the video below for demonstration. This adds a visually engaging prelude to the game.

> NOTE:
>
> To simplify and enhance understanding, from now on we will refer to 'LEDs' as 'lights'.

Our implementation approach for this section is clear and systematic:

### Blinking Student ID Digits on 7-Segment Displays

**BCD to 7-Segment Conversion:**

To display the digits of student IDs on four 7-segment displays, we connect a BCD-to-7-segment decoder to each display. The input to each decoder consists of the BCD representation of the corresponding digit from the student ID.

**Blinking Mechanism:**

The outputs of the decoders, which are connected to the 7-segment displays, pass through a tri-state buffer. The enable condition of the tri-state buffer is controlled by an AND gate that combines the following signals:

1. **Game Not Started:** This ensures that the blinking occurs only before the game begins.
2. **2 Hz Clock Signal:** This clock alternates the buffer's enable state every 0.5 seconds, toggling the display output on and off.

This setup creates a blinking effect on the digits of the 7-segment displays.

### Light Show (Spiral Pattern)

**Shift Registers for Light Control:**

We employ several shift registers to control the lights in a spiral pattern from top to bottom. The input to the serial shift registers is set to 1 to simulate sequentially lighting up each light.

**Operation:**

Before the game begins, the shift registers sequentially shift a '1' through the lights, creating the spiral lighting effect. Once all the lights are illuminated, the shift registers are reset, and the process repeats until the game starts.

**Condition for Operation:**

The shift registers remain active as long as the game has not started. Once the game begins, the enabling condition for the shift registers is invalidated, thus stopping the light show.

## Game State Management

To manage the state of the game, we utilize two variables: GameStartState and GameEndState. As their names suggest:

- **GameStartState = 1**: This indicates that the game has started, meaning the player has clicked the Start button.
- **GameEndState = 1**: This indicates that the game has ended, whether the player won or lost.

### Functionality of the States

**GameStartState**:

- When the Start button is pressed, a value of 1 is loaded into the register associated with the GameStartState output.
- This register is reset only when the Reset button is pressed.

**GameEndState**:

- The value of GameEndState is determined by the XOR operation applied to two variables: GameWon and GameLost.
- These variables are explained in further detail in the section on Game End Condition.

### Game State Table

| GameStartState | GameEndState | Description |
|---|---|---|
| 0 | X | The game has not started. |
| 1 | 0 | The game has started but not yet ended. |
| 1 | 1 | The game has started and has ended. |

### Controlling Circuit Activities

Using these two variables (GameStartState and GameEndState), we can effectively control the operation of the circuits for each part of the game:

- **When GameStartState = 0**, pre-start behaviors such as a blinking display and light show are active.
- **When GameStartState = 1** and GameEndState = 0, the game logic, block movement, and player interactions are active.
- **When GameEndState = 1**, post-game behaviors, such as displaying results, can be triggered.

This modular approach ensures clear control over the different stages of the game.

# Game End Condition

To track whether the player has won or lost, we define two variables: **GameWon** and **GameLost**. These variables indicate the player's win or loss status and are stored in a register. Here's how their values are managed:

## Reset Condition

Both variables are reset to 0 when the **Reset** button is pressed.

## Load Condition

- **GameWon** is set to 1 when the **WinGame** signal is activated.
- **GameLost** is set to 1 when the **LoseGame** signal is activated.

These signals are generated based on the game's win and loss conditions.

## Win Condition

The player wins if they collect 3 points.

The player's score is displayed on the interface using two 7-segment displays, representing the score as an 8-bit BCD (Binary-Coded Decimal) value.

A comparator circuit compares this 8-bit BCD value with **0000 0011** (the binary representation of 3). If the comparator output indicates "greater than or equal to 3," the **WinGame** signal is set to 1. This activates the **GameWon** variable, marking the game as won and ending it.

## Loss Conditions

A player loses the game if either of the following conditions is met:

**Collision in the Top 3 Rows (FullBoard Condition):**

If a newly generated 3x3 block collides with a fixed block and cannot completely move out of the top three rows, a variable named **FullBoard** is set to 1.

When **FullBoard** equals 1, the **LoseGame** signal is triggered, which sets **GameLost** to 1 and ends the game.

Details on how **FullBoard** is determined can be found in the Full Board Condition section.

## Timer Reaches 99 Seconds

The game timer is represented as an 8-bit BCD (Binary-Coded Decimal) value. To detect when the timer reaches 99 (**1001 1001 in BCD**) seconds, the following condition is checked:

If we let eight BCD Timer bits be T0...T7:

$$(T0 \text{ AND } \neg T1 \text{ AND } \neg T2 \text{ AND } T3) \text{ AND } (T4 \text{ AND } \neg T5 \text{ AND } \neg T6 \text{ AND } T7)$$

If this condition evaluates to true, the `LoseGame` signal is triggered, resulting in `GameLost` being set to 1 and the game ending.

### Combining Loss Conditions

The `LoseGame` signal is determined based on the following conditions:

$$LoseGame = FullBoard \text{ OR}$$
$$\Big((T0 \text{ AND } \neg T1 \text{ AND } \neg T2 \text{ AND } T3) \text{ AND } (T4 \text{ AND } \neg T5 \text{ AND } \neg T6 \text{ AND } T7)\Big)$$

This means the game will be marked as lost if either the board is full or the timer has reached 99 seconds.

To ensure these loss conditions only apply after the game has started, we finalize the `LoseGame` condition by ANDing it with the `GameStartState`:

$$LoseGame = FullBoard\ OR$$
$$\Big((T0\ AND\ \neg T1\ AND\ \neg T2\ AND\ T3)\ AND\ (T4\ AND\ \neg T5\ AND\ \neg T6\ AND\ T7)\Big)$$
$$AND\ GameStartState$$

### Summary of Logic

| **Condition** | **Signal Triggered** | **Result** |
|---|---|---|
| Score >= 3 | WinGame = 1 | GameWon = 1 (End Game) |
| Collision in top 3 rows | FullBoard = 1 | GameLost = 1 (End Game) |
| Timer reaches 99 seconds | Timer AND 1001 1001 | GameLost = 1 (End Game) |

## Timer and Score

To display the game timer and player score, we use four 7-segment displays, each driven by a Binary-Coded Decimal (BCD) value, since humans naturally count in base-10. Below is a detailed implementation:

### Game Timer

The game timer counts the elapsed seconds and is displayed as two digits:

- **Units Place (0–9):** Controlled by Counter A.
- **Tens Place (0–9):** Controlled by Counter B.

## Implementation of Timer Counters

**Counter A (Units):**

- **Clock Input:** A 1 Hz clock signal serves as the input.
- **Operation:**
  - Counter A increments from 0 to 9.
  - When it reaches 9, it resets to 0 on the next clock edge.
  - When Counter A resets, Counter B increments by 1.

**Counter B (Tens):**

- **Clock Input:** Triggered whenever Counter A resets after reaching 9.
- **Operation:**
  - Counter B increments from 0 to 9.

**Output:**

The combined output of Counter A and Counter B forms an 8-bit BCD value representing the timer:

**Timer Output = Counter B (MSB) | Counter A (LSB)**

**Connection to 7-Segment Displays:**

The 8-bit BCD output is sent to BCD-to-7-segment decoders. To control when the timer is displayed, we use tri-state buffers with the following enable condition:

$$GameStartState \text{ AND } \neg GameEndState$$

This condition ensures the timer is displayed only when the game is active (i.e., when it has started but not yet ended).

## Player Score

The player score is displayed using a similar structure to the timer, with a few key differences:

**Clock Input for Counter A (Units):**

- Instead of a 1 Hz clock, Counter A utilizes a signal known as AddScore as its clock input.

**AddScore:**

- The AddScore signal becomes active (1) whenever the player successfully clears a complete row of lights.
- Further details about this signal are provided in the Score Calculation and Row Clearing Mechanism section.

**Operation:**

- Counter A increments each time it receives an AddScore pulse.
- When Counter A reaches a value of 9, it resets to 0 and increments Counter B, similar to how the timer counters operate.

**Output and Display:**

- The combined output of Counter A and Counter B forms the 8-bit BCD (Binary-Coded Decimal) score value.
- This output is sent to the BCD-to-7-segment decoders, which are controlled by tri-state buffers. The enable condition for these buffers is the same as that for the timer:

$$GameStartState \text{ AND } \neg GameEndState$$

This ensures a clear and logical display of the game timer and player score.

# Generating and Placing a Random 3x3 Block in the Game

To create and position a random 3x3 block on the game board, we utilize a Linear-Feedback Shift Register (LFSR) to generate 6 pseudo-random bits. These bits determine both the shape and position of the block.

## Generating the Random Bits

**LFSR for 6-Bit Output:**

The LFSR continuously generates a 6-bit pseudo-random sequence. When the block generation conditions are met, the current output of the LFSR is stored in a register to ensure stability for subsequent block generation.

**Block Generation Conditions:**

A block is generated only when the following conditions are met:

1. All lights in the top 3 rows are off (indicating that last generated 3x3 block has completely left this area or simply the game just started).
2. `GameStartState = 1` (the game has started).
3. `GameEndState = 0` (the game has not ended).

These conditions ensure that blocks are created only when the game is active and there is space to place a new block.

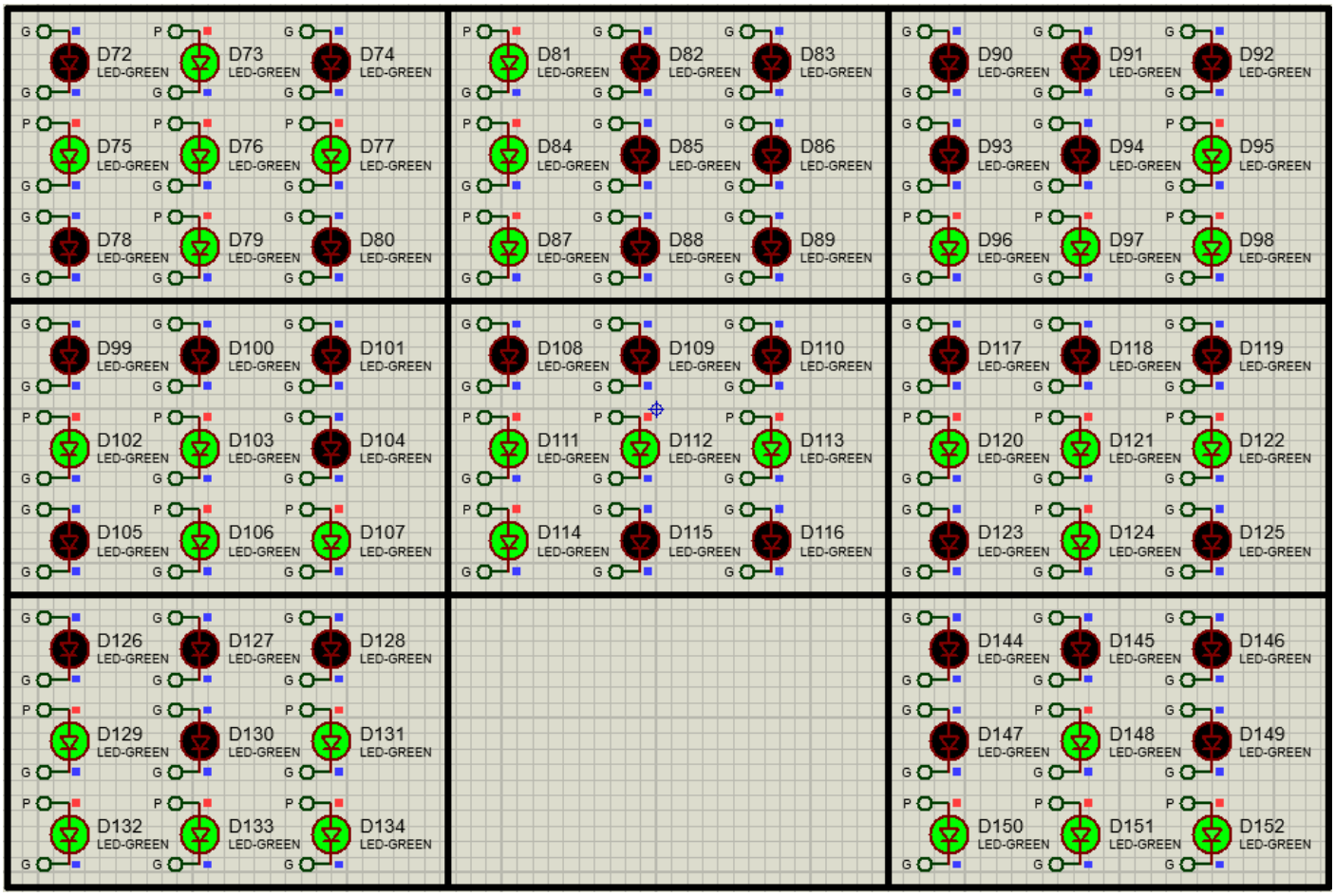
## Using Random Bits to Determine the Block Shape

**Block Shape Selection:**

The 3 most significant bits (MSBs) of the LFSR output are used as a selector for an 8-to-1 multiplexer. This multiplexer has 8 inputs, each representing one of the predefined 3x3 block shapes, as follows:

$$
\begin{aligned}
S1: &\ 010\ 111\ 010 \\
S2: &\ 100\ 100\ 100 \\
S3: &\ 000\ 001\ 111 \\
S4: &\ 000\ 110\ 011 \\
S5: &\ 000\ 111\ 100 \\
S6: &\ 000\ 111\ 010 \\
S7: &\ 000\ 101\ 111 \\
S8: &\ 000\ 010\ 111
\end{aligned}
$$

Each shape is represented by 9 bits (3 bits per row). The output of the multiplexer (9 bits) is stored in variables: `SHAPE0`, `SHAPE1`, ..., `SHAPE8`.




## Using Random Bits to Determine Block Position

**Block Positioning:**

The three least significant bits (LSBs) of the Linear Feedback Shift Register (LFSR) output are utilized as inputs to a 3-to-8 decoder. This decoder determines which three adjacent columns on the game board the block will occupy.

**Mapping Decoder Outputs to Positions:**

Given that the game board consists of seven columns, the possible positions are as follows:

- **POS0-2**: Columns 0, 1, 2
- **POS1-3**: Columns 1, 2, 3
- **POS2-4**: Columns 2, 3, 4
- **POS3-5**: Columns 3, 4, 5
- **POS4-6**: Columns 4, 5, 6

To account for the eight decoder outputs, the outputs corresponding to positions 5, 6, and 7 are merged with the first five positions using OR gates. This ensures that only valid positions are generated, increasing the likelihood of placing blocks further away from the board edges.

**Connecting to Buffers:**

For each position (POS0-2, POS1-3, ..., POS4-6), a 9-bit tri-state buffer is employed. The enable condition for each buffer corresponds to its respective POS signal. The output of the buffer is then connected to the appropriate columns on the game board.

## Process Summary

**Shape Selection:**
The top 3 bits of the LFSR output are used to select one of the predefined 8 shapes through an 8-to-1 multiplexer.

**Position Selection:**
The bottom 3 bits of the LFSR output determine the position of the block using a 3-to-8 decoder.

**Block Placement:**
The selected shape, which consists of 9 bits, is routed through the corresponding position buffer. This enables the shape to control the lights in the specified 3 columns.

# Control Generated Block

After a block is generated, the player has three seconds to control it using the following actions:

- **Shift Right (S_Right)**
- **Shift Left (S_Left)**
- **Rotate (Rotate)**

## Shifting Left and Right

Each of the top three rows is managed by a separate 7-bit shift register. When a block is created, its shape data is initially stored in temporary variables, which are then loaded into these shift registers. This setup allows for future modifications, such as rotations, without directly altering the display registers.

**To Shift the Block:**

The shift registers move their bits to the left or right.

The shift operation is clocked by the OR combination of `S_Right`, `S_Left`, `Rotate`, and `CreationCondition` signals.

The shift direction is determined by the order of temporary light variables as shift register inputs:

- **If left-to-right:** S_Right decides shift direction.
- **If right-to-left:** S_Left decides shift direction.

## Preventing Invalid Shifts

To ensure that a block does not shift out of bounds:

- **Shift Right** is permitted only if there are no active bits in last column. This is verified using a NOR gate that checks the bits in 7th column.
- **Shift Left** is permitted only if there are no active bits in first column. This is verified using a NOR gate that checks the bits in 1st column .

Both conditions are combined using an AND operation with the shift signals before execution. Additionally, shifts are allowed only while the ControlCondition signal is active, which means they can only occur within the three-second control window. So The previous output will be combined with **ControlCondition** using an AND operation.

**Shift Registers Clock Signal:**

$$\Big(S_{\text{Left}} \text{ AND } (R_{0,0} \text{ NOR } R_{1,0} \text{ NOR } R_{2,0}) \text{ AND } ControlCondition) \text{ OR}$$
$$\Big(S_{\text{Right}} \text{ AND } (R_{0,6} \text{ NOR } R_{1,6} \text{ NOR } R_{2,6}) \text{ AND } ControlCondition) \text{ OR}$$
$$(Rotate \text{ AND } ControlCondition) \text{ OR}$$
$$CreationCondition$$

$$\text{which } [R_{0,0} - R_{1,0} - R_{2,0}] \text{ are red lights in the first column and}$$
$$[R_{0,6} - R_{1,6} - R_{2,6}] \text{ are red lights in the last column.}$$

## Rotation Implementation

Blocks can rotate counterclockwise in four states:

1. 0° (Initial State)
2. 90° Counterclockwise
3. 180° Counterclockwise
4. 270° Counterclockwise

A 2-bit counter tracks the rotation state (from 0 to 3). Each time a Rotate signal is received, the counter increments, looping back to 0 after reaching 3.

A 2-to-4 decoder maps the counter value to one of four 9-bit tri-state buffers, each storing the block's shape in the correct rotated orientation. The rotations follow this pattern:

**Initial Shape (0°):**

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

**After 90° Counterclockwise Rotation:**

$$\begin{matrix} 3 & 6 & 9 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{matrix}$$

Each further 90° rotation applies the same transformation. The new rotated shape is stored in temporary variables before being loaded into shift registers, ensuring seamless display updates.

## Center of Rotation

The block must rotate around its current column position. A shift register stores the block's current position and updates it when the block shifts left or right. This approach ensures that the rotated shapes remain within their assigned three columns. The position register:

1. Loads the initial position upon block creation.
2. Shifts left or right when the player moves the block.
3. Prevents shifting beyond the edges of the board by applying conditions to POS4-6 (for right shifts) and POS0-2 (for left shifts).

Each rotation applies the updated shape data to the relevant three columns based on the position register.

**Finalizing Block Placement**

When the 3-second control period ends, the block becomes fixed in place. The block’s shape is transferred to the game board registers, and the falling mechanism begins. Then when the last generated block completely exits the three first rows, the next block generation process starts.

# Downward Shift of the Generated Block

Once the block is finalized after the 3-second control period, it must transition from the construction phase to the game board. This section explains how the block moves downward within the game grid.

## Transferring Block Data to the Game Core

After a block is created and the 3-second movement/rotation period expires, it becomes static. This is determined when both the ControlCondition and CreationCondition signals are deactivated (set to 0).

At this point, the values stored in the shift registers, which are responsible for constructing and controlling the block, are transferred to the Game Core section, managing the main grid of the game.

To facilitate this transfer:

1. The Enabler for the construction/control registers is set to ControlCondition.
2. The Enabler for the Game Core registers is set to NOT(ControlCondition).
3. The Load signal is activated simultaneously, ensuring proper data transfer.

## Game Core Structure and Shift Register Configuration

In the Game Core section, shift registers are used to store and move blocks downward. The configuration is as follows:

- **First 3 Rows**: Each column in the top three rows has its own 3-bit shift register. Since the board has 7 columns, this results in 7 shift registers handling the 21 lights of the top three rows.
- **Rows 4 to 10**: Each light (cell) in these rows has an individual shift register that stores Blue (B) and Yellow (Y) values, representing the block's color state. As the game board contains 49 lights from row 4 onward, this requires 49 shift registers.

## Total Shift Registers

- 7 shift registers for the top 3 rows
- 49 shift registers for rows 4-10
- **Total**: 56 shift registers

## Functionality of Shift Registers in Rows 1-3

The 7 shift registers managing the first three rows have a unique function:

Each register receives three input values (one from each row) from its respective column, along with a 0 (Ground) input at DL (Data Load).

With each shift operation:

- The first bit is replaced with 0.
- The remaining bits shift downward from row 1 to row 3.
- The outputs of these registers are connected to the corresponding column lights.

These shift registers operate with a 1Hz clock, meaning the block moves down once per second.

## Load Signal Conditions

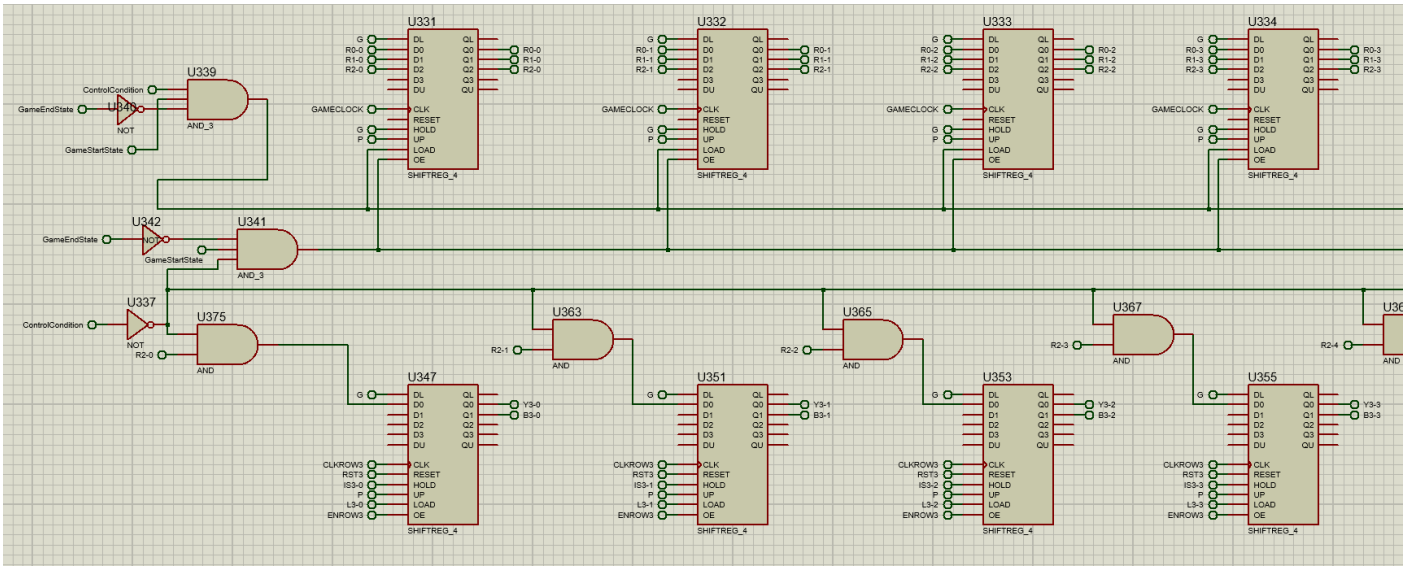
The load signal is activated when the following conditions are met:

- GameStartState = 1 **(Game is running)**
- NOT(GameEndState) = 1 **(Game is not over)**
- ControlCondition = 1 **(Block is still in the control phase)**

However, the Output Enable (OE) is set to NOT(ControlCondition), which means the transferred data remains hidden until the block is finalized.

**When ControlCondition switches to 0:**

- The Load signal is deactivated.
- The shift registers become active, and the block starts moving downward at each clock pulse.
- The third row's values transition into the fourth-row shift registers, ensuring there is no premature movement into row 4 before finalization.

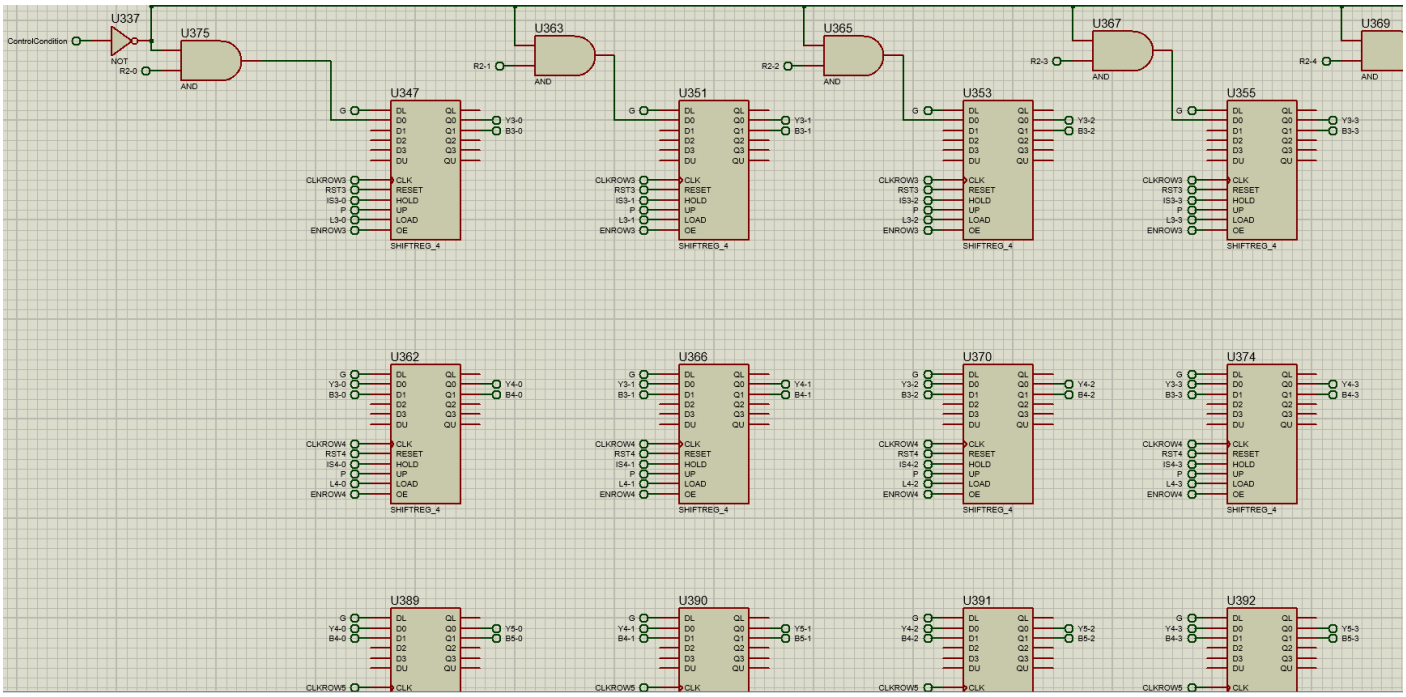


## Downward Shift in Rows 4-10

From row 4 onward, each shift register behaves independently:

- The Yellow (Y) and Blue (B) values from the current row are loaded into the shift register of the row below.
- The shift occurs at each clock pulse, effectively moving the entire block downward one row per second.

This setup allows full control over each light, mimicking the behavior of a 2D shift register system.




> NOTE:
>
> In all of the schematic images above, the variables indices begin from 0.

## Next Steps: Collision Handling & Block Locking

Now that the downward shift mechanism is in place, the next step involves handling collisions and locking blocks in place by transferring Yellow (Y) values into Blue (B). This will be covered in the next section.

# Collision Handling & Block Locking Mechanism

In the game, a **moving block** can become a **fixed block** under two conditions:

1. **It reaches the bottom** of the game grid.
2. **It collides with an existing fixed block.**

When either of these conditions is met, all parts of the block transition from **Yellow (Y) to Blue (B)**, indicating they are now part of the static game board.

---

## Step 1: Independent Behavior of Lights (Before Group Dependency)

To simplify the collision logic, we first assume that **each moving light is independent** and doesn't belong to a block group. That means each individual light will turn into a fixed block only when it **personally collides** with something below it.

## Detecting Collision for Rows 3 to 9

A collision is detected by **AND-ing the Yellow (Y) value of the current light** with the **Blue (B) value of the light directly below it**:

$$\text{Collision Condition} = Y(i, j) \text{ AND } B(i+1, j)$$

For example, in **row 4, column 0**, the collision condition is:

$$Y(3, 0) \text{ AND } B(4, 0)$$

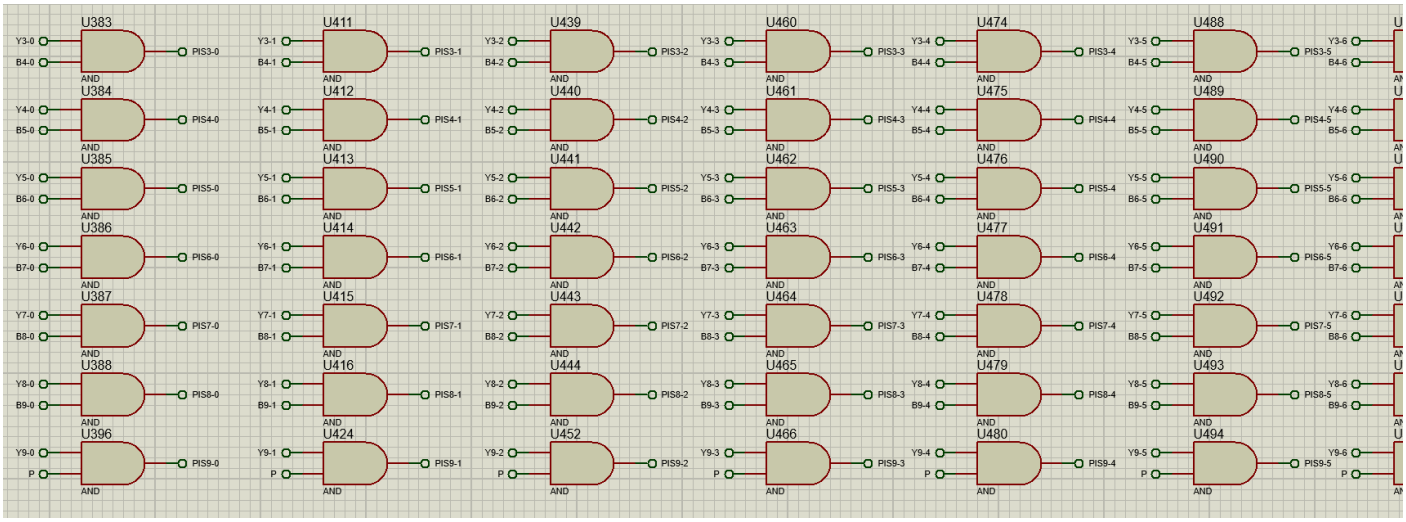
If this condition evaluates to **1**, the light must become part of the static grid.

## Handling Row 10 (Last Row)

For the bottom row (row 10), there's no row below to check. Instead, we consider a light to have **collided when it simply exists**:

$$Y(9, j) \text{ AND } 1$$

Since AND with 1 does not change the value, this means **any Yellow light in row 10 automatically triggers a collision.**




---

### Step 2: Transitioning from Y (Moving) to B (Fixed)

Once a collision is detected, we must shift the **Y value** into **B** on the next clock edge. This is done by using the **HOLD** input of the shift register.

- **HOLD = 1:** No shift occurs (light remains moving).
- **HOLD = 0:** The register shifts the **Y value into B**, making the light fixed.

To achieve this:

$$\text{HOLD}(i, j) = \neg[\text{Collision Condition}(i, j)]$$

That means:

- **If there is no collision** → HOLD remains **1**, allowing the block to keep shifting down.
- **If a collision occurs** → HOLD becomes **0**, causing Y to be shifted into B, effectively "freezing" the light.

> NOTE:
>
> **In the implementation, we utilize the collision condition directly rather than its negation. However, we still negate the output of the Collision Condition using NAND instead of AND gate.** This means that we connect the collision condition variable directly to the shift register of the light, eliminating the need to include a NOT gate.

### Step 3: Adding Group Behavior (Locking Entire Blocks Instead of Individual Lights)

So far, only individual lights become fixed when they collide. However, **entire blocks need to freeze together**. Otherwise, a falling Tetris-like piece would break into smaller parts instead of landing as a unit.

### Solution 1: Unique Block Identifiers (More General Approach)

A logical approach would be to assign **a unique 2-bit identifier** to each block at creation. This way:

- Each moving light knows which block it belongs to.
- If **any part** of the block collides, all other lights with the **same ID** will also be locked.

The **ID system uses a 2-bit counter**, allowing **up to 4 distinct moving blocks** to exist at the same time (since new blocks only appear every few seconds).

If a block's **ID matches** that of a colliding light, **all its lights must become fixed** as well.

This is the **ideal method** for a more scalable implementation.

---

### Solution 2: Simple Proximity Locking (Current Implementation)

For a quick solution before the project deadline, we use **a less precise but functional method**:

- If **any light in a row collides**, all lights in:
    - The **same row**
    - **Two rows above**
    - **Two rows below**

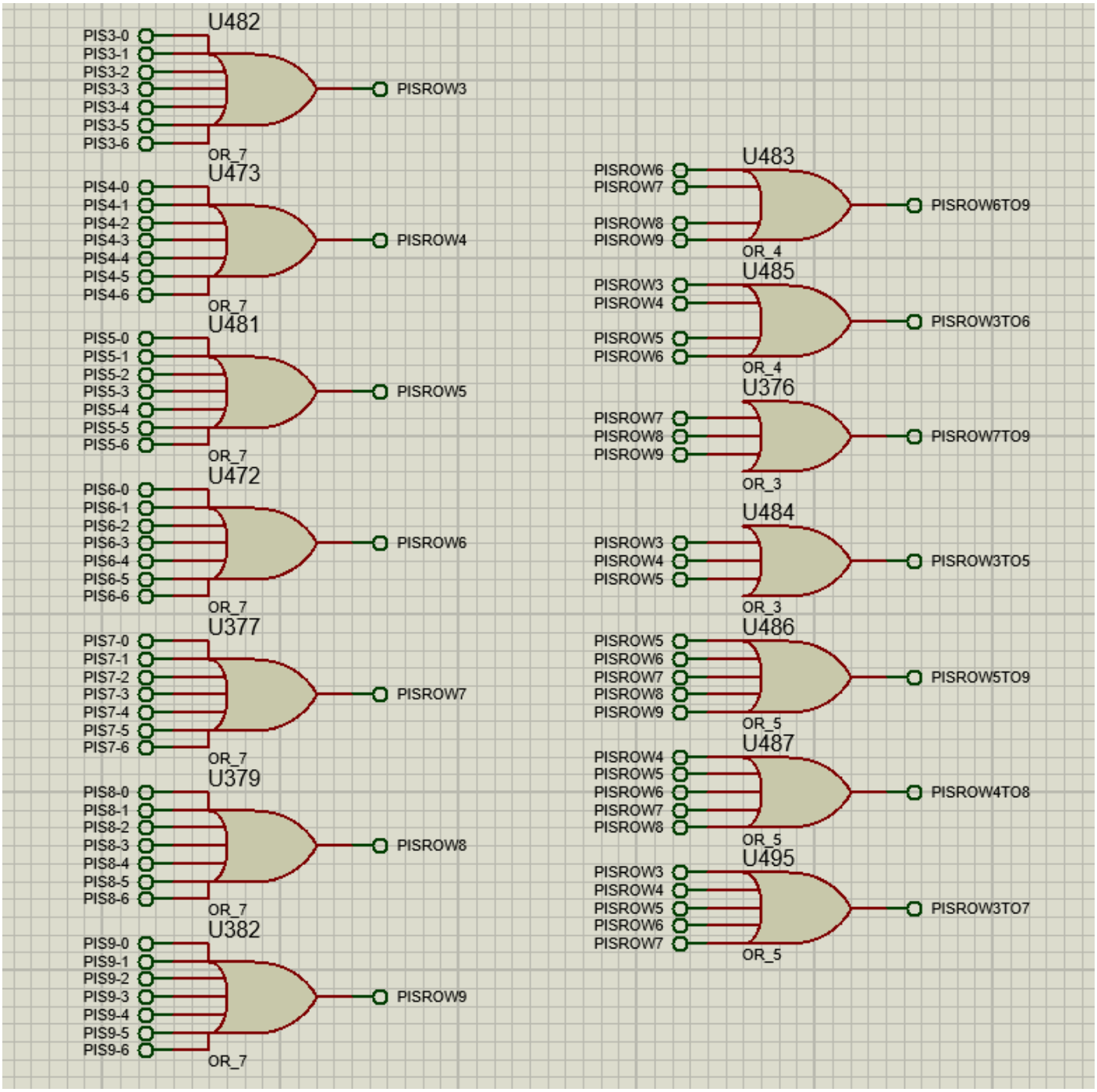
  will **also be fixed**.

This works because **all generated blocks are 3×3** in size. Given the game mechanics, this assumption is valid since no other blocks will be in this range.

**Example:**

- If a light in **row 6** collides:
    - **Rows 4, 5, 6, 7, and 8** all become fixed.

While this method **isn't as elegant as the ID-based approach**, it works **within the constraints of the project**.




---

### Step 4: Updating Collision Conditions for Block Groups

Previously, the **collision condition only checked individual lights**. Now, we extend it to apply to **entire block groups**.

For a light at **(i, j)**:

$$\text{Block Collision Condition} = \neg B(i,j) \text{ AND } [\text{Collision Occurred in Rows (i-2 to i+2)}]$$

If we use:

$$PIS(i,j)$$

to denote **the independent collision condition** at (i, j). and

$$PISROW(i-2 \text{ to } i+2)$$

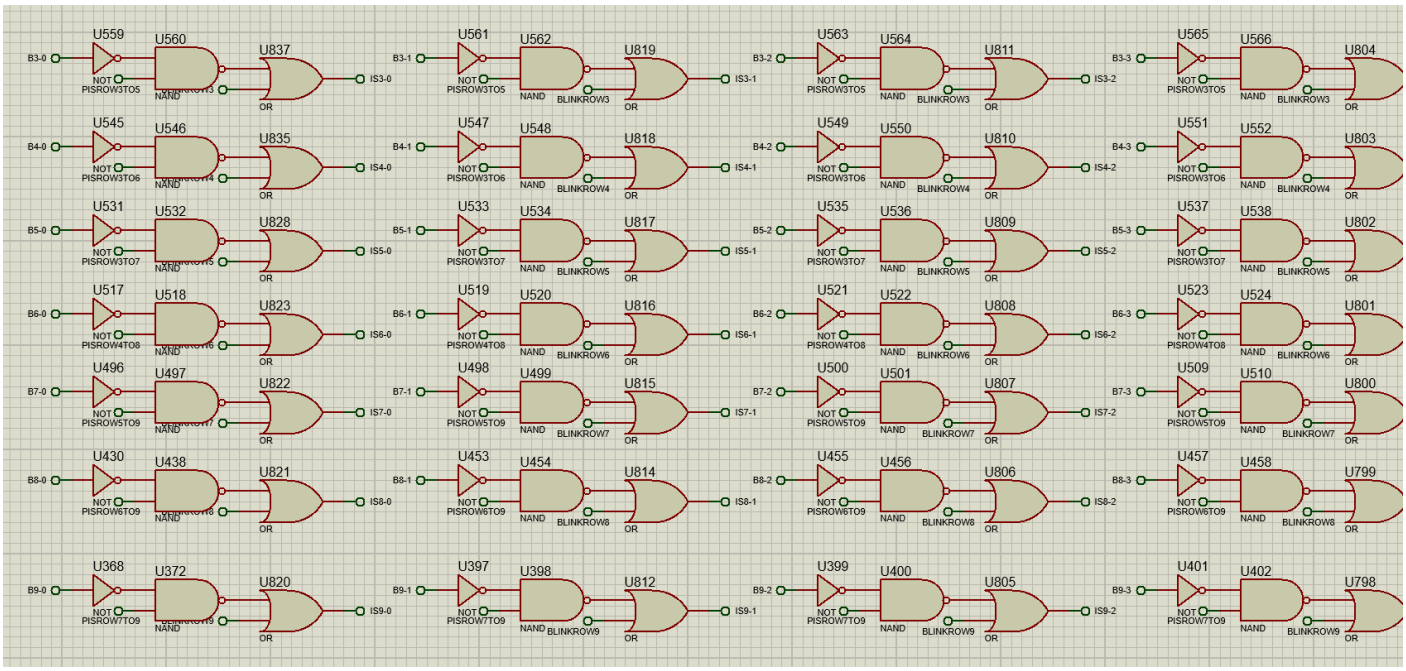
to denote **if any collision happened in rows** (i-2 to i+2).

Then:

$$\text{Final Collision Condition or IS}(i,j) = \neg B(i,j) \text{ AND } PISROW(i-2 \text{ to } i+2)$$

This ensures that:

- **Only moving lights (not already fixed ones) are affected**.
- **If any light in a block collides, the entire block locks**.




> NOTE:
>
> The naming of the final light collision condition as "IS" stands for "Inner Shift." This term is used because it describes how the shift register internally shifts the B and Y values, rather than employing a downward shifting mechanism on the game board. Similarly, "PIS" stands for "Partial Inner Shift."

## Step 5: Updating Load Conditions

Now that blocks are freezing properly, we must ensure **fixed blocks don't shift anymore**.

Each shift register should only load new data if:

1. **The current light isn't already fixed (B = 0).**
2. **The light in the row above isn't fixed (to prevent shifting downward).**
3. **The collision condition(IS) hasn't been triggered (to prevent overriding the freeze action).**

Mathematically:

$$\text{Load Condition}(i,j) = \neg B(i,j) \text{ AND } \neg B(i-1,j) \text{ AND } \neg IS(i,j)$$

**Example:** For **row 5, column 2 (Light4-1)**:

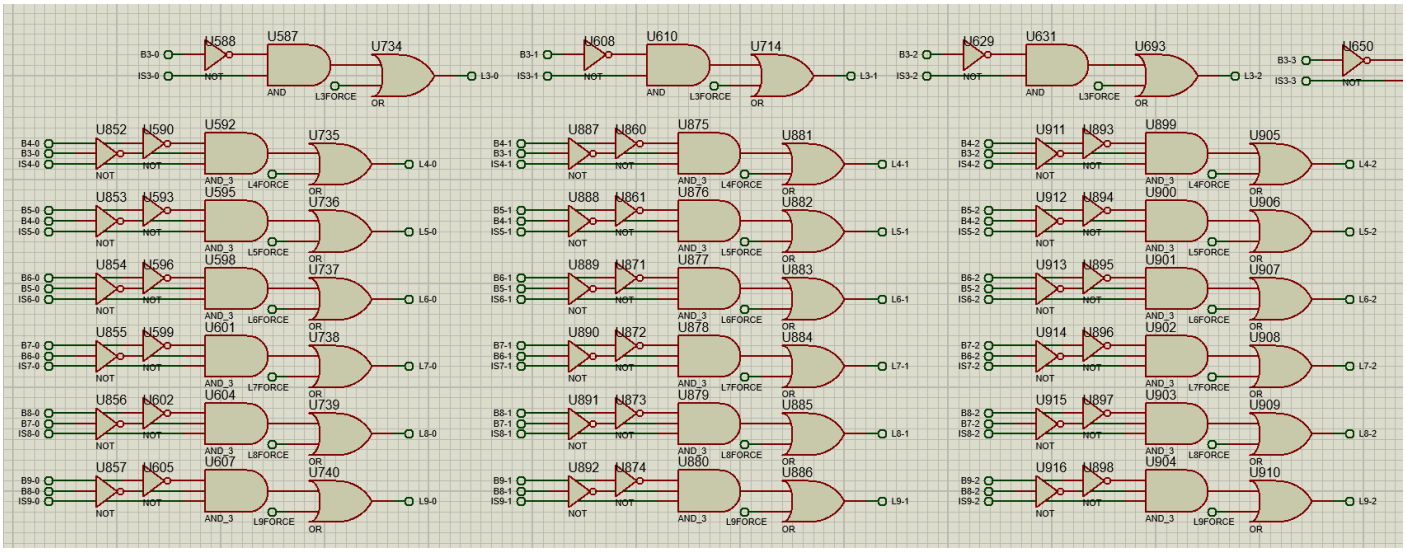
$$\neg B(4,1) \text{ AND } \neg B(3,1) \text{ AND } \neg IS(4,1)$$

For row **4**, there's no light above it that can be static, so as an example for **row 4, column 6** we simplify:

$$\neg B(3,5) \text{ AND } \neg IS(3,5)$$

This ensures:

- Fixed blocks don't shift.
- The row below doesn't keep shifting into a fixed row.
- Blocks that **should freeze** don't get overridden by a load operation.




> NOTE:
>
> Since we already negated the 'IS' signal through the NAND gate, we do not negate it again during the load condition check.

## Supplementary Information

The variables **indices all begin from 0** thats why index of a light in row 4, column 3 would be L(3,2).

LXFORCE variables will be introduced in Score Calculation and Row Clearing Mechanism section which forces load condition to be true.

While the **simple row-freezing method** was used for quick implementation, a **block ID-based method** would be better for a long-term solution. The main **downside** of the current approach is that it **relies on block size assumptions** (3×3), hence limiting flexibility.

That said, given the constraints, this method **worked well for meeting the deadline** and maintaining correct gameplay behavior.

# Score Calculation and Row Clearing Mechanism

This section explains the **scoring system and row clearing mechanism** in the game, detailing how a **full row is detected, blinked, deleted, and how upper rows shift down accordingly.** It also explains the logic behind score incrementing after a row is cleared.

## Step 1: Detecting and Blinking Full Rows Before Deletion

A row is considered **full** when all its lights are **Blue (Fixed Blocks)**. This check applies to **rows 4 to 10**.

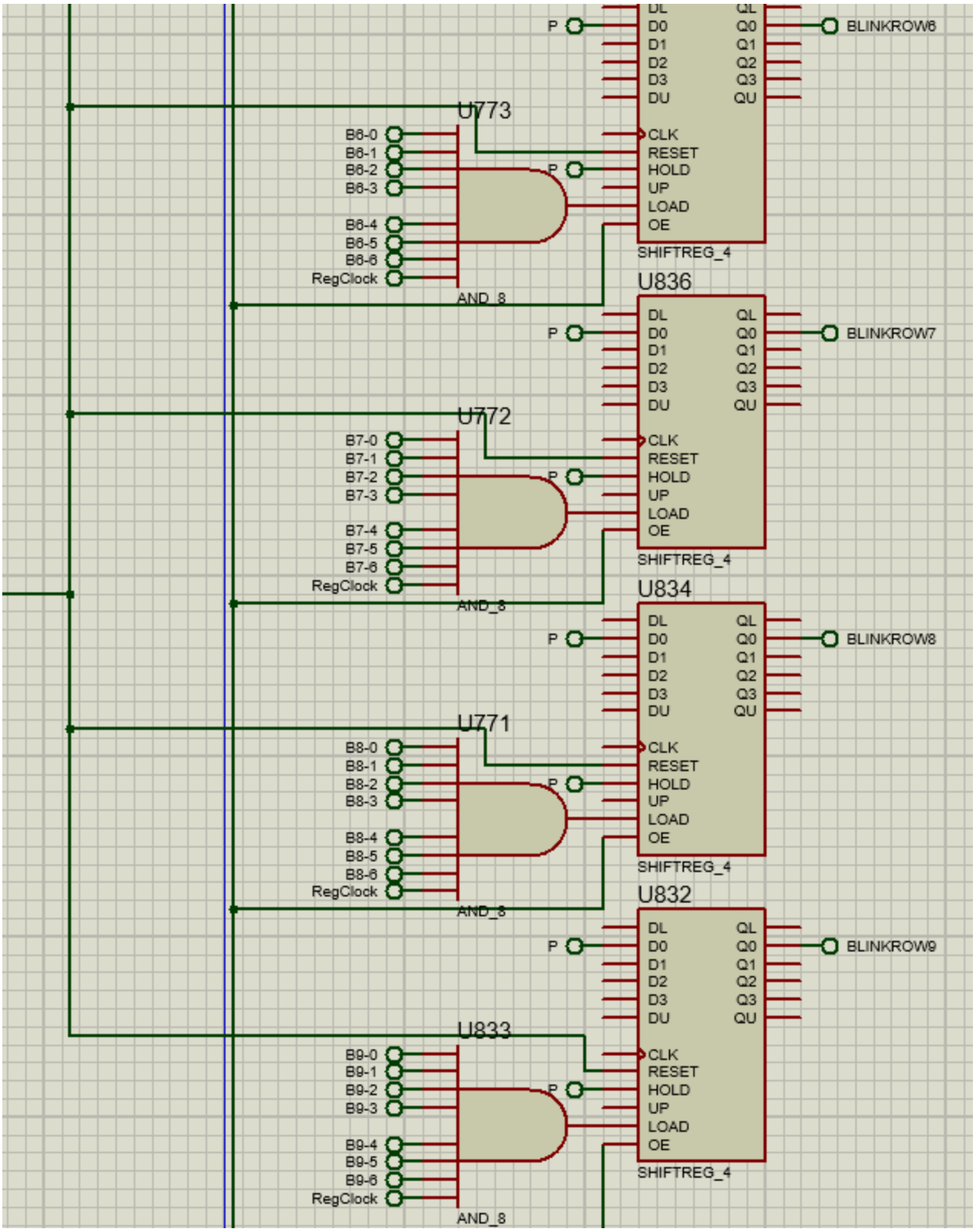
- When a row is full, it begins **blinking for 2 seconds** before being deleted.
- During this **blinking phase**, all the blocks in that row toggle between **on and off states**.
- After **2 seconds**, the entire row is removed, and **all rows above it shift down by one row**, including fixed blocks.

To define the **blinking condition** for a row, we use the following approach:

1. **Check if the row is full:**
    - **AND all 7 B values** (Fixed Blocks) of that row.
    - If the result is `1`, the row is full, and blinking starts.
2. **Toggling the Output Enable (OE) signal** of shift registers:
    - The OE input of the shift register toggles between `0` and `1`.
    - This makes the **B values of that row toggle between `1` and undefined**, creating the blinking effect.
3. **Storing the Blink Condition:**
    - Each row (4-10) requires a register to store its blinking condition.
    - The register loads the blinking condition when the row is detected as full.
    - After 2 seconds, this condition is cleared to allow row deletion.

Thus, **seven registers** (`BLINKROW3` to `BLINKROW9`) are defined, one for each row from 4 to 10.

- **Load Input of Each Register:** AND of all **B values** in that row.
- **Reset Input:** Activated **after 2 seconds of blinking**, which is explained later.


U773
B6-0
B6-1
B6-2
B6-3
B6-4
B6-5
B6-6
RegClock
AND_8
P
DL
D0
D1
D2
D3
DU
CLK
RESET
HOLD
UP
LOAD
OE
QL
Q0
Q1
Q2
Q3
QU
SHIFTREG_4
BLINKROW6
U836
BLINKROW7
U772
B7-0
B7-1
B7-2
B7-3
B7-4
B7-5
B7-6
RegClock
AND_8
U834
BLINKROW8
U771
B8-0
B8-1
B8-2
B8-3
B8-4
B8-5
B8-6
RegClock
AND_8
U832
BLINKROW9
U833
B9-0
B9-1
B9-2
B9-3
B9-4
B9-5
B9-6
RegClock
AND_8

### Step 2: Generating the Blinking Signal for Rows

The **Output Enable (OE) input** of the shift registers for each row is controlled using **ENROW3...ENROW9** signals.

**Blinking Logic:**

1. **Use a 2 Hz clock pulse** (i.e., toggles every 0.5s).
2. **NAND this clock signal with BLINKROWX** (the blinking flag for that row).
3. **AND the result with `GameStartState` & NOT(`GameEndState`).**

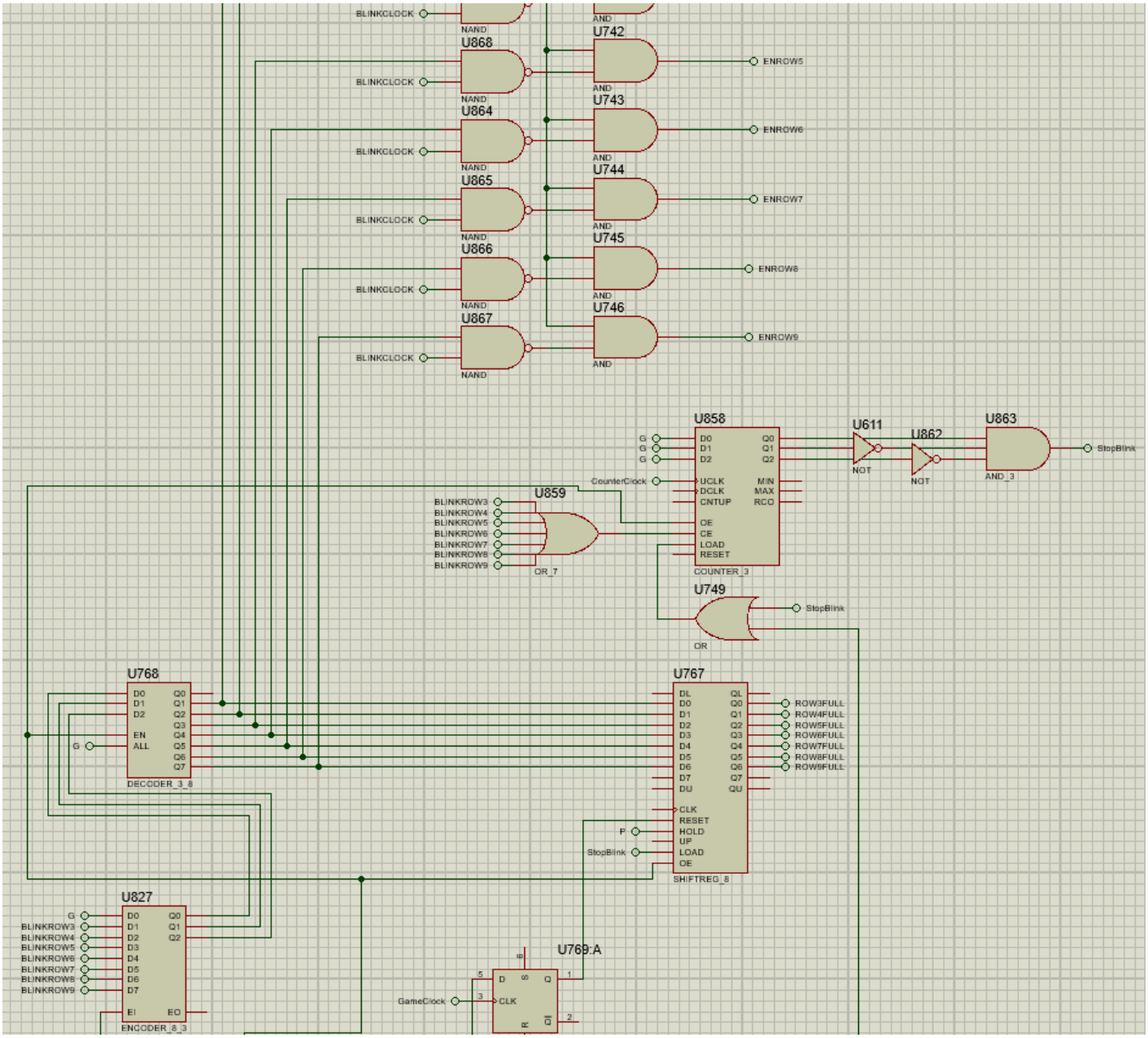
### Why NAND?

- When `BLINKROWX = 0`, the **NAND output is always `1`**, keeping the row visible.
- When `BLINKROWX = 1`, the **NAND output toggles with the clock pulse**, causing a blinking effect.

### Truth Table for ENROWX (Blinking Signal Generation)

| GameStartState | GameEndState | BLINKROWX | Clock Pulse (2Hz) | NAND Output | ENROWX |
|---|---|---|---|---|---|
| 0 | X | X | X | X | 0 |
| 1 | 0 | 0 | X | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | X | X | X | 0 |

- **When `BLINKROWX = 0`**, row is always visible.
- **When `BLINKROWX = 1`**, row toggles visibility at 2 Hz, making it blink.


BLINKCLOCK
NAND
AND
U868
U742
ENROW5
U864
U743
ENROW6
U865
U744
ENROW7
U866
U745
ENROW8
U867
U746
ENROW9
U858
U611
U862
U863
NOT
AND_3
StopBlink
CounterClock
U859
BLINKROW3
BLINKROW4
BLINKROW5
BLINKROW6
BLINKROW7
BLINKROW8
BLINKROW9
OR_7
COUNTER_3
U749
OR
U768
DECODER_3_8
U767
SHIFTREG_8
ROW3FULL
ROW4FULL
ROW5FULL
ROW6FULL
ROW7FULL
ROW8FULL
ROW9FULL
U827
ENCODER_8_3
U769:A
GameClock

### Step 3: Controlling the Blinking Duration (2 Seconds Limit)

- The **blinking should last exactly 2 seconds**.
- We use a **3-bit counter** to track this duration.
- The counter starts counting **whenever any row starts blinking**.
- The counter **resets after 2 seconds**, stopping the blinking.

### Counter Configuration

- **Clock Enable (CE) Input:**
  - **OR of all** `BLINKROW3...BLINKROW9` signals.
  - If any row is blinking, the counter starts.
- **Counter Operation:**
  - After **1 second**, a signal `StopBlink = 1`.
  - After **2 seconds**, the counter resets (`StopBlink = 0`).

### StopBlink Calculation

If `C0, C1, C2` are the **3-bit counter outputs** from least to most significant bit:

$$\text{StopBlink} = C0 \text{ AND } \neg C1 \text{ AND } \neg C2$$

- `StopBlink = 1` after **1 second** of blinking.
- `StopBlink = 0` after **2 seconds** (counter resets).

### Counter Reset Logic

- **Reset Input:** `StopBlink`
- When `StopBlink = 1`, on the next **counter clock cycle (1Hz)**, the counter loads `000`, effectively resetting.

---

### Step 4: Define And Store Row Full Condition

Once `StopBlink = 1`, full row **must be deleted**.

To store full row conditions, we define **seven new registers:**

- `ROW3FULL ... ROW9FULL`
- These store which row was **full before deletion**.

### Register Behavior

- **Load Input:** `BLINKROW3...BLINKROW9`
- **Load Condition:** `StopBlink = 1` (asynchronous)
- **Reset Condition:**
  - OR of all `ROW3FULL...ROW9FULL`
  - Reset synchronously with **game shift clock (1Hz).**

This ensures **ROWXFULL stays `1` for exactly 1 game shift cycle** before resetting.

---

## Step 5: Deleting Full Rows & Shifting Upper Rows Down

To **delete full rows** and shift down all rows **above the deleted row**, we define:

- `L3FORCE ... L9FORCE`
- These force the **loading of shift registers for affected rows**, ensuring they move down.

### Force Load Logic

- **When `ROWXFULL = 1`, all rows above `X` must shift down.**
- We define:

$$\text{L9FORCE} = ROW9FULL$$

$$L8FORCE = ROW8FULL \text{ OR } ROW9FULL = ROW8FULL \text{ OR } L9FORCE$$

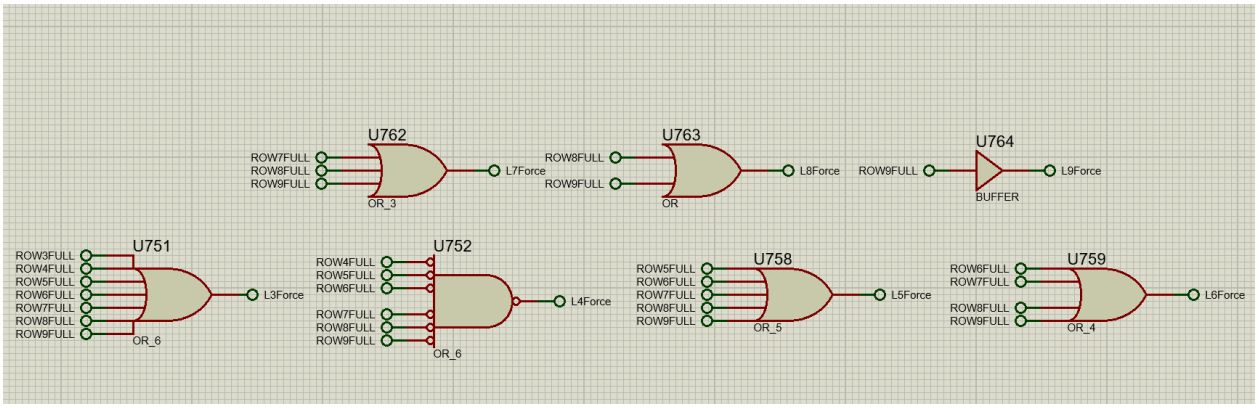
$$L7FORCE = ROW7FULL \text{ OR } \ldots \text{ OR } ROW9FULL = ROW7FULL \text{ OR } L8FORCE$$

$$L6FORCE = ROW6FULL \text{ OR } \ldots \text{ OR } ROW9FULL = ROW6FULL \text{ OR } L7FORCE$$

$$L5FORCE = ROW5FULL \text{ OR } \ldots \text{ OR } ROW9FULL = ROW5FULL \text{ OR } L6FORCE$$

$$L4FORCE = ROW4FULL \text{ OR } \ldots \text{ OR } ROW9FULL = ROW4FULL \text{ OR } L5FORCE$$

$$L3FORCE = ROW3FULL \text{ OR } \ldots \text{ OR } ROW9FULL = ROW3FULL \text{ OR } L4FORCE$$



### 5.1 Understanding the Row Deletion Process

When a row (let's say row X) is completely filled, all its lamps have **B = 1** (indicating they are active). The deletion process must:

- **Shift all rows above it downward by one row**, including both **fixed lamps** and moving ones.
- Ensure that the **previous state of row X does not remain** after shift. (**Overwrite lamps** in row X with the row above it.)

To achieve this, the **shift register loading condition** is carefully designed.

Each row in the system has a **shift register** responsible for storing its lamp states. The loading condition of each row's shift register determines **whether it updates based on the previous row or retains its current state**.

> NOTE:
>
> For additional info on how load condition is determined check out Shift Registers Load Condition.

- Normally, a row loads its new values from the row above it **only if certain conditions are met**.
- However, when a row is **completely filled and needs to be deleted**, all of its lamps have **B = 1**.
    - This causes the **shift register loading condition** of that row and the row below to be **false (0)**, meaning that row and the row below it will **not copy any values from the row above it** and will hold its current state.
    - As a result, all rows **above the full row shift down by one position**.
    - Since the **row below the full row is prevented from loading data**, it holds its current state instead of incorrectly inheriting values from the full row.
    - To finalize the deletion, **we force-load the shift registers of the full row** with the values from the row above it.
- This effectively **pushes all upper rows down** by one step.
- The full row is **overwritten and disappears**.

Thus, all shift registers for **the full row and rows above the full row** are **loaded unconditionally** on the next shift clock, **bypassing normal conditions** and ensuring a correct downward shift.

## 5.2 Handling Multiple Full Rows (Priority Encoder & Decoder Mechanism)

When multiple rows are full at the same time, the system must determine **which row to blink and delete first**. To enforce an **orderly deletion process**, we use a **priority encoder** and a **decoder**.

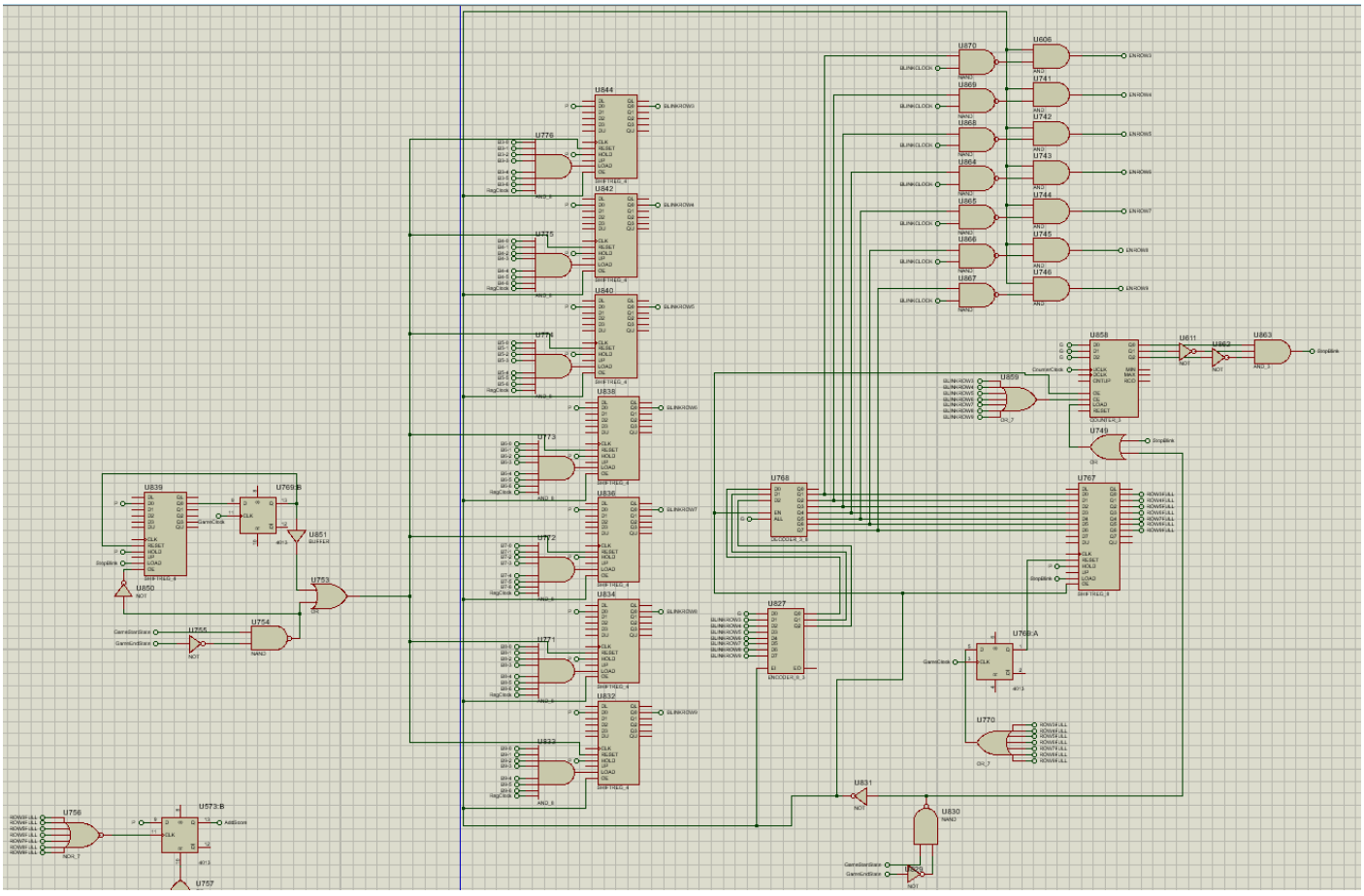
### Priority Encoder (8-to-3)

- The priority encoder **detects the highest-numbered (deepest) full row** and assigns it the highest priority.
- It takes **seven inputs** (`ROW3FULL` to `ROW9FULL`), each representing whether a row is full.
- The **output is a 3-bit code**, identifying the highest-number full row.

### Decoder (3-to-8)

- The **3-bit output** of the priority encoder is fed into a **3-to-8 decoder**.
- The decoder converts this back into **one-hot format**, activating only the corresponding row's **BLINKROW** signal (`BLINKROW3` to `BLINKROW9`).
- The **highest-numbered full row** is now the only one that blinks, ensuring orderly deletion.

> NOTE:
>
> The **first output of the decoder (least significant bit)** corresponds to **"no row is full"**. We ignore this signal. The remaining **seven outputs** control the **blinking process** for `BLINKROW3` to `BLINKROW9`.



### Step 6: Adding Score After Row Deletion

To count the score:

- A **D Flip-Flop** with **rising edge trigger** and **asynchronous reset** is used.
- **Clock Input:** `NOR(ROW3FULL...ROW9FULL)`.
- **Output:** `AddScore` (acts as clock for the score counter).

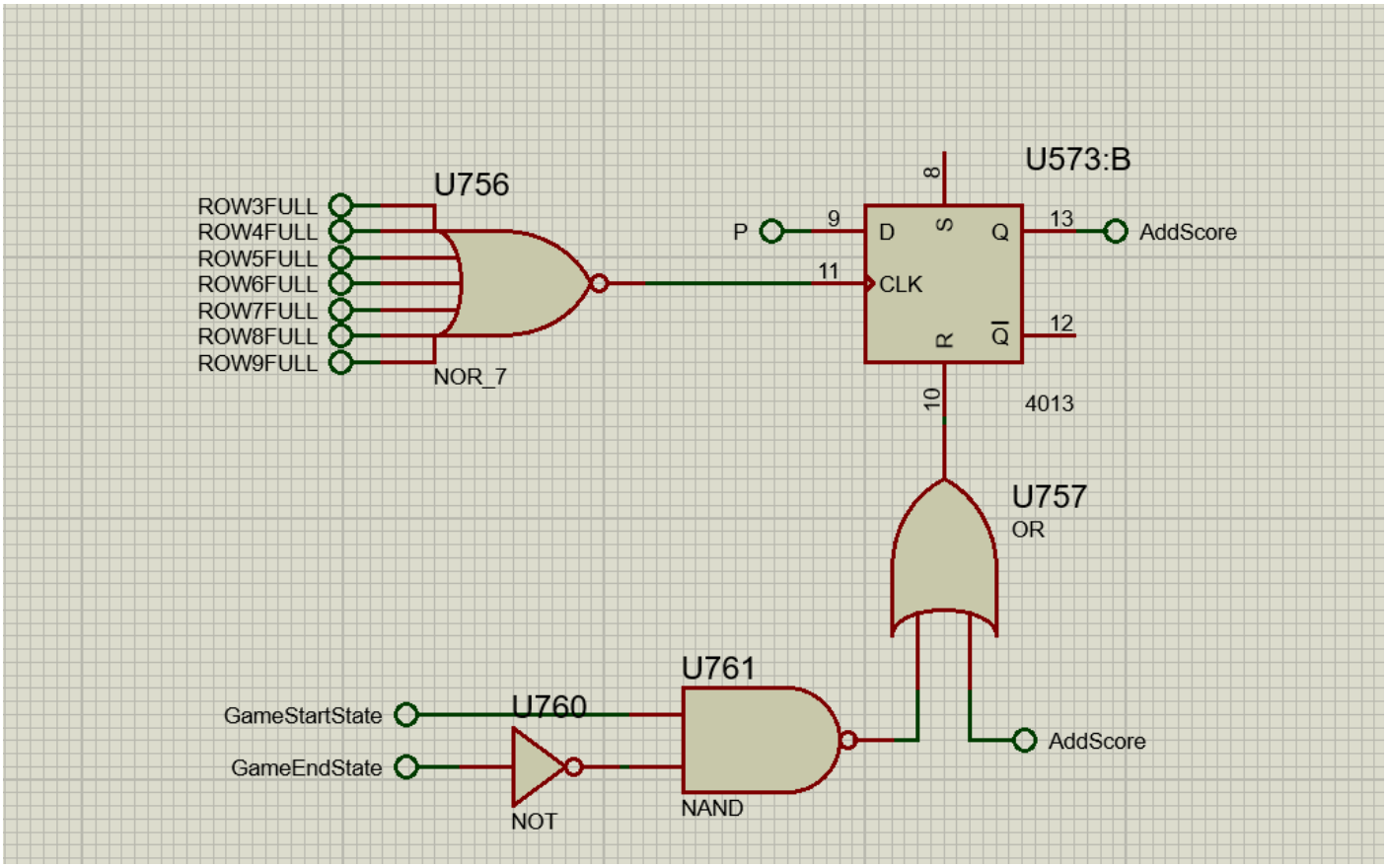
#### Reset Logic

- **Reset when:**
  - Game hasn't started (`GameStartState = 0`).
  - Game is over (`GameEndState = 1`).
  - `AddScore` is already `1`.

$$\text{DFF Reset} = AddScore \text{ OR } (GameStartState \text{ NAND } \neg GameEndState)$$

#### Score Incrementation

- **At game start, the NOR output is `1`.**
- **When a row gets deleted, `NOR(ROWXFULL) = 0` (falling edge).**
- **As soon as ROWXFULL resets (`0 → 1`), NOR becomes `1` again, creating a rising edge.**
- This **rising edge** triggers `AddScore`, **increasing the player's score**.



# Full Board Condition

The **Full Board Condition** occurs when a newly generated block collides with a fixed block in the first three rows and fails to exit them. If this condition is met, the game ends, and the player loses.

To implement this condition, a systematic approach is taken to detect collisions within the top three rows and determine whether the generated block is unable to move beyond them. The detection mechanism is based on verifying whether a collision has occurred in rows **4, 5, or 6**, which would prevent the block from fully exiting the first three rows.

---

## Step 1: Collision Detection in Row 4

To check for a collision in row 4, a column-wise detection method is utilized. Each column consists of multiple lamps, where the state of the lamps in the first three rows is represented by **R**, and the fixed blocks in row 4 are represented by **B**.

The detection process involves the following steps:

1. **Logical AND Operation**
    - For each column, the value of **R** in the first three rows is **AND-ed** with the value of **B** in row 4.
    - This operation determines whether an active lamp in the top three rows aligns with an already fixed block in row 4.
2. **Ensuring the Block is in Motion**
    - The result of the previous step is further **AND-ed** with the **NOR of CreationCondition and ControlCondition**.
    - This ensures that the block is currently in motion and is neither in the process of being generated nor under external control.
3. **Aggregating Results Across All Columns**
    - The outputs of these operations across all **seven columns** are combined using a **logical OR operation**.
    - If any of the columns produce a **true** value, a collision in row 4 is confirmed.

The resulting Boolean equation for detecting a collision in row 4 is given by:

$$\Big((R_{2,0} \text{ AND } B_{3,0}) \text{ OR } (R_{2,1} \text{ AND } B_{3,1}) \text{ OR } (R_{2,2} \text{ AND } B_{3,2}) \text{ OR } (R_{2,3} \text{ AND } B_{3,3}) \text{ OR } (R_{2,4} \text{ AND } B_{3,4}) \text{ OR } (R_{2,5} \text{ AND } B_{3,5}) \text{ OR } (R_{2,6} \text{ AND } B_{3,6})\Big) \text{ AND } (CreationCondition \text{ NOR } ControlCondition)$$

This equation ensures that any collision in row 4 is detected and flagged as part of the **Full Board Condition**.

## Step 2: Collision Detection in Rows 5 and 6

For detecting collisions in rows 5 and 6, a different approach is employed. Instead of checking direct **R-B** collisions, the **Inner Shift (IS) variables** are utilized. These variables indicate whether a collision has occurred within a **2-row radius** of a lamp.

1. **Using Inner Shift (IS) for Collision Detection**
    - The **IS** variables provide a mechanism to detect potential collisions that extend beyond row 4.
    - The values of **IS** for row 4 and row 5 are examined to identify if the generated block is encountering an obstruction.
2. **Preventing False Positives from Row 7 Collisions**
    - If a collision occurs in row 7, it can incorrectly set the **IS** values for rows 5 and 6, leading to a false detection.
    - To avoid this issue, it is necessary to verify that the generated block is still within the **first three rows**.
    - Additionally, the **Control Condition** must be **0** to ensure that the block is actively moving.
3. **Ensuring Row 6 Detection is Valid**
    - An additional requirement is introduced for row 6: at least **one lamp in row 4 must be active** to confirm a valid collision.
    - This prevents a scenario where a new block has formed in row 8, but an old block remains in row 5, leading to incorrect detection.

The Boolean equation for collision detection in rows 5 and 6 is structured as follows:

$$
\begin{gathered}
(IS_{3,0} \text{ NAND } IS_{3,1} \text{ NAND } IS_{3,2} \text{ NAND } IS_{3,3} \text{ NAND} \\
IS_{3,4} \text{ NAND } IS_{3,5} \text{ NAND } IS_{3,6} \text{ NAND } IS_{4,0} \text{ NAND} \\
IS_{4,1} \text{ NAND } IS_{4,2} \text{ NAND } IS_{4,3} \text{ NAND } IS_{4,4} \text{ NAND} \\
IS_{4,5} \text{ NAND } IS_{4,6}) \text{ AND } (Y_{3,0} \text{ OR } Y_{3,1} \text{ OR} \\
Y_{3,2} \text{ OR } Y_{3,3} \text{ OR } Y_{3,4} \text{ OR } Y_{3,5} \text{ OR } Y_{3,6})
\end{gathered}
$$

This equation ensures that a valid collision is detected in rows 5 and 6 only when the generated block remains in the first three rows and is actively moving.

### Step 3: Combining the Conditions and Game Termination Logic

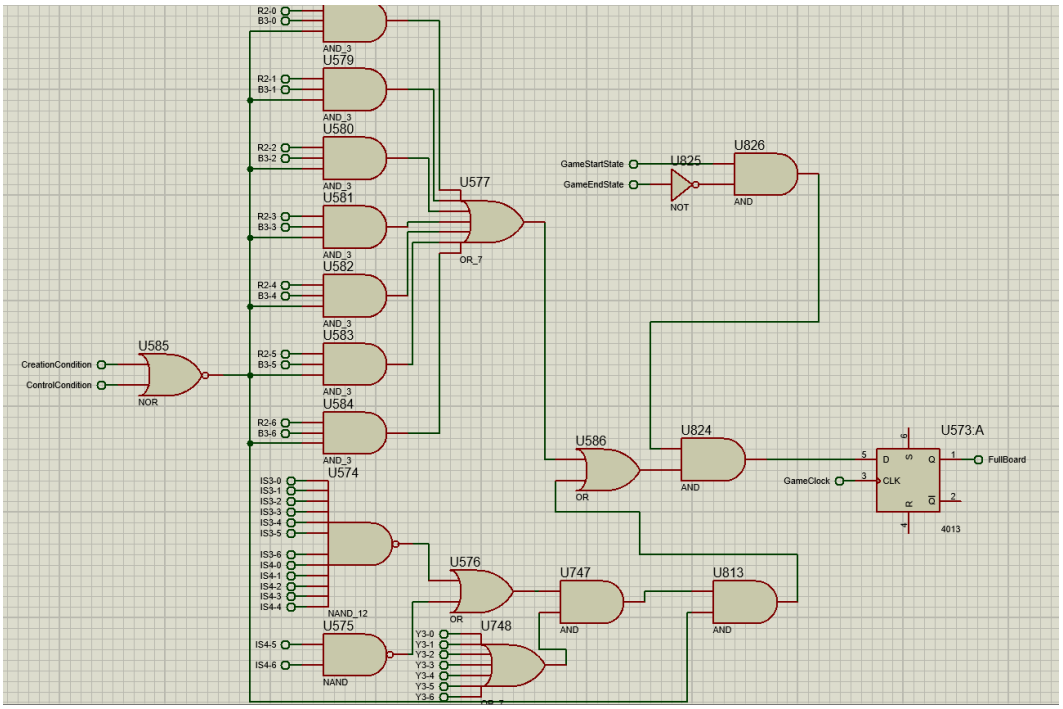
To finalize the Full Board Condition, the results from both collision detection mechanisms are combined and integrated into the game's logic:

1. **Combining the Collision Conditions**
    - The results from **Step 1 (row 4 collision)** and **Step 2 (row 5 and 6 collisions)** are combined using a **logical OR operation**.
    - This ensures that a collision in **either case** is sufficient to trigger the game-over condition.
2. **Ensuring the Game is Active**
    - The combined result is **AND-ed** with **GameStartState** and **NOT(GameEndState)**.
    - This ensures that the game-over condition is triggered only when the game is running and has not already ended.
3. **Storing the Result in a D Flip-Flop (DFF)**
    - The final **Full Board Condition** is stored in a **D Flip-Flop**, which is **clocked by the game's shift clock**.
    - This ensures that the condition is evaluated **synchronously** with the game's progression.

The final Boolean equation for the Full Board Condition is expressed as:

$$FullBoard = \Big((\text{Collision in Row 4}) \text{ OR } (\text{Collision in Rows 5 \& 6})\Big) \text{ AND } (GameStartState \text{ AND } \neg GameEndState)$$

When this condition evaluates to **1**, the game-over state is activated in the next clock cycle, ensuring that the player loses the game as soon as an irreversible collision occurs in the top three rows.



In the implementation, the **Inner Shift (IS) variable** is directly used as the **HOLD input** of each lamp's shift register. This means that when **IS = 0**, the **Inner Shift operation occurs**, which transitions zero to Y and Y to B value.

Since **IS is negated by design**, its logic must be carefully considered when detecting collisions in rows **5 and 6**. Instead of using a **logical OR operation** to aggregate collision results across lamps, a **NAND operation** is used.

## Reason for Using NAND Instead of OR

1. **Default State of IS Variables**
    - The **default value** of all IS variables is **1** (meaning no collision is detected).
    - When a collision is detected at a particular lamp, its IS variable transitions to **0**.
2. **Effect on OR and NAND Operations**
    - If OR were used to aggregate the collision detection results, it would produce **0** only when **all lamps are in collision**, which is not the desired behavior.
    - Instead, using **NAND ensures that a single detected collision (IS = 0 for any lamp) results in a final output of 1**, which correctly signals a collision.
3. **Consistency with IS Inversion in Shift Registers**
    - Since IS is already **inverted in the hardware implementation**, using NAND instead of OR maintains logical consistency.
    - This approach prevents unnecessary NOT gates in the circuit, simplifying the design and ensuring efficient collision detection.

This ensures that **if any IS value becomes 0 (collision detected), the entire NAND operation produces 1**, correctly signaling a collision while maintaining compatibility with the hardware's use of IS in shift registers.

# Notes

## Timing Considerations in the Schematic

Throughout the schematic design of the project, **RegClock** has been used wherever a **short delay** is required. This clock is connected to a **20 Hz signal**, which introduces a delay ranging from a **minimum of approximately 0.001 seconds to a maximum of 0.05 seconds**.

The purpose of using RegClock is to ensure that sequential operations have sufficient timing margins to execute correctly, preventing **glitches or race conditions** caused by immediate state transitions.

## Rotation Issue

### Problem Overview

For shapes that do not fully occupy a **3×3 grid**, unintended rotations can occur when shifting the shape to the **edge of the board**.

### Example Scenario

Consider the following **vertical 3×1 shape**:

```
1 0 0
1 0 0
1 0 0
```

If this shape is positioned in the **first column (left edge)** and rotated **twice counterclockwise**, it will end up in the **third column (right edge)** as shown below:

```
0 0 1
0 0 1
0 0 1
```

Now, if the shape is shifted **two positions to the left** (back to the first column) and rotated once more, it results in:

```
1 1 1
0 0 0
0 0 0
```

which is **not a proper counterclockwise rotation**.

**Cause of the Issue**

- Rotation logic assumes that the **shape remains within a fixed 3×3 grid**.
- When a shape is **pushed against a wall** and rotated, the board may attempt to place blocks in **invalid positions**.
- Since the shift operation **moves the shape before rotation**, the resulting transformation may **violate expected rotational symmetry**.

**Impact on Gameplay**

- This issue **does not cause the game to crash** or become unplayable.
- However, it **introduces unexpected rotation behavior** when pieces are placed at the **left or right edges**.

# Installation

1. **Clone or Download the Project:**
    - Clone the repository using Git:

      ```
      git clone https://github.com/Ohtears/Tetris-w-Digital-Logic.git
      ```

      or download the ZIP file and extract it.
2. **Prerequisites:**
    - **Proteus Design Suite:** Ensure you have Proteus (version 8.15 or later) installed on your computer.
3. **Setup:**
    - Open Proteus and load the main schematic file `Tetris_Digital_Project.pdsprj`.
    - **(Optional)** Configure simulation parameters such as clock frequency according to your personal preferences.
4. **Compilation/Simulation:**
    - In Proteus, run the simulation to check that the circuit behaves as expected.
    - Make any necessary adjustments based on the simulation feedback.
5. **Additional Notes:**
    - Refer to the detailed documentation in this document for troubleshooting and further configuration details.

# Running the Game

1. **Start the Simulation:**
    - With the project loaded in Proteus, click the "Run" button or press F12 shortcut to begin the simulation.
2. **Game Start-Up:**
    - Press the **Start** button on the game interface to initialize the game.
    - The 7-segment displays will reset the timer and score, and a random 3×3 block will be generated in the top three rows.
3. **Gameplay Controls:**
    - **Start:** Begins the game.
    - **Reset:** Resets the game and returns the circuit to its initial state.
    - **Rotate:** Rotates the current block counterclockwise.
    - **S_Left/S_Right:** Shifts the block left or right within the top three rows.
    - During the initial **three-second control window**, you can adjust the block's position and orientation **(There is an orange LED indicator for this control window placed below the 7-segment display for scores.)**. Once the block leaves the top rows, the game continues with the next block generation.
4. **Game End Conditions:**
    - The game ends if:
        - A new block **collides** with fixed blocks in the top three rows (Full Board Condition).
        - The game timer reaches **99 seconds**.
    - A win is achieved when the player clears enough rows to **reach 3 points**.
    - Use the Reset button to start a new session after the game ends.

> NOTE:
>
> You can download a sample gameplay by clicking **here**.

# Resources

### Tools and Software Used

- **Proteus Design Suite** – Used for circuit simulation and game logic implementation.

### References and Academic Materials

- **Digital Design Principles & Practices** – By John F. Wakerly.
- **Digital Design** – By M. Morris Mano.
- **Course Materials from Guilan University** – Lecture notes and assignments from the *Digital Logic Design* course provided by **Professor Mahdi Aminian**.

# Acknowledgments

We would like to express our sincere gratitude to:

- **Professor Mahdi Aminian** and their **honorable Teaching Assistants** – For their invaluable guidance and support throughout the project.
- Our dear friend **Arash Parsa** – For providing valuable insights and feedback.

---

Team Members

- **MohammadHossein Keyvanfar** – Implementation, Documentation & Optimization
- **Ashkan Marali** – Implementation & Debugging

# Contact

For any questions or feedback, please feel free to reach out:

- **Name:** MohammadHossein Keyvanfar
- **Email:** Mohammadhossein.kv@gmail.com
- **GitHub:** https://github.com/MohammadHosseinKv

---

- **Name:** Ashkan Marali
- **Email:** AshkanMarali@gmail.com
- **GitHub:** https://github.com/Ohtears