ECE 3710 - Embedded Systems

Semester Project - Gorilla

A Classic Turn-Based Game on the 8052 Microcontroller

Submitted by: Billy Corona
Brandon Arbegast

Instructor: Justin Jackson
Date Submitted: Apr 23, 2025

Table of Contents

1. Introduction

The purpose of this project is to recreate the turn-based game, Gorilla, implemented using the Silicon Labs 8052 Microcontroller. This project exists to reinforce the fundamental concepts of Embedded Systems. This includes concepts such as digital input/output, delay loops, interrupt handling, finite state machine design, and hardware interfacing.

The game is played between two players. The players take turns inputting angle and velocity values to throw a projectile at the other player. The players are represented by gorillas and the projectile being thrown is a banana. The left potentiometer controls angle, and the right controls velocity. The left button launches the banana, and the right button resets the game. Player one will always be on the left, and player two will always be on the right. Random wind values can be enabled/disabled by two dip switches. The core functionality of the game is built around a state machine, which controls the game's transitions.

This project is designed to run on the 8052 development board, using I/O peripherals for user interaction and display. The game logic was implemented using C, and is constrained to the limits of the 8052. These constraints include memory, processing power, and onboard hardware.

This project provides an opportunity to design and implement a working embedded application with real-time input, basic graphics, and finite state logic. While the game is simple by modern standards, it still provides practical experience in debugging and programming within a constrained environment.

2. Scope

This document outlines the hardware and software design of a recreation of the game Gorilla, implemented on the Silicon Labs 8052 microcontroller. It contains the structure of the main game logic, user input handling, and the turn-based projectile physics simulation, all using a finite state machine. The display and interface are implemented using a peripheral board.

This document does not contain a listing of hardware datasheets, low level register configurations, or a complete explanation of the 8052 instruction set. It is assumed that the reader has knowledge of basic microcontroller architecture and embedded systems concepts. Additionally the system does not contain more than a basic audio system. The visual design of the system is neglected in favor of functionality, the banana, gorillas, and skyline contain just enough detail to recognize these elements. Finally, there are no considerations in regards to production, mass testing, reliability analysis, or cost.

---

3. Design Overview

The Gorilla Game project is divided into two main hardware components: the Silicon Labs 8052 development board and a peripheral interface board. The connected board provides the controls and display. Gameplay is presented to the users on a 64x128 pixel graphical LCD. Inputs are taken from potentiometers, buttons, and DIP switches, and the system is powered by a 9V supply.

Game operations are handled through software run on the 8052. This includes player input handling, physics simulation, collision detection, and visual and audio output. A finite state machine manages the flow of the game across four states: IDLE, MENU, LAUNCH, and GAME OVER. Timer interrupts, ADC channels, and DAC output are used for animation, input sampling, and sound generation.

The game supports two players, each controlling a gorilla which is placed randomly on a randomly generated skyline. Each player takes turns adjusting their launch parameters using potentiometers and launching the banana via a button press. Trajectory of the banana is affected by gravity and wind which is controlled by two DIP switches. The system reacts accordingly to collisions depending on if the skyline or a gorilla was hit. The system is fully self contained, so all input and output occur locally.

3.1 Requirements

The design satisfies the following requirements:

- Operate from a 9V supply.
- Display game information using a 64x128 pixel LCD.
- Support two buttons (launch and reset).
- Support two potentiometers (for velocity and angle).
- Support two DIP switches (for wind control).
- Animate turn-based projectile launches with real-time physics.
- Render graphics for the gorillas, skyline, and banana.
- Display information relating to the banana launch including velocity, angle, wind, and the current player.
- Play a sound for launch and collision events.
- Respond appropriately to different collision types, and reset for new rounds.

The full breakdown of how each requirement is fulfilled is provided within section 4.

3.2 Dependencies

The system relies on the following hardware and software components

- C8051F020 Microcontroller
- Peripheral Interface Board
  - 64x128 LCD screen
  - Two potentiometers
  - Two push buttons
  - Two DIP switches
  - DAC audio amplifier
- 9V DC Power Supply
- Timer, ADC, and DAC located on the 8052

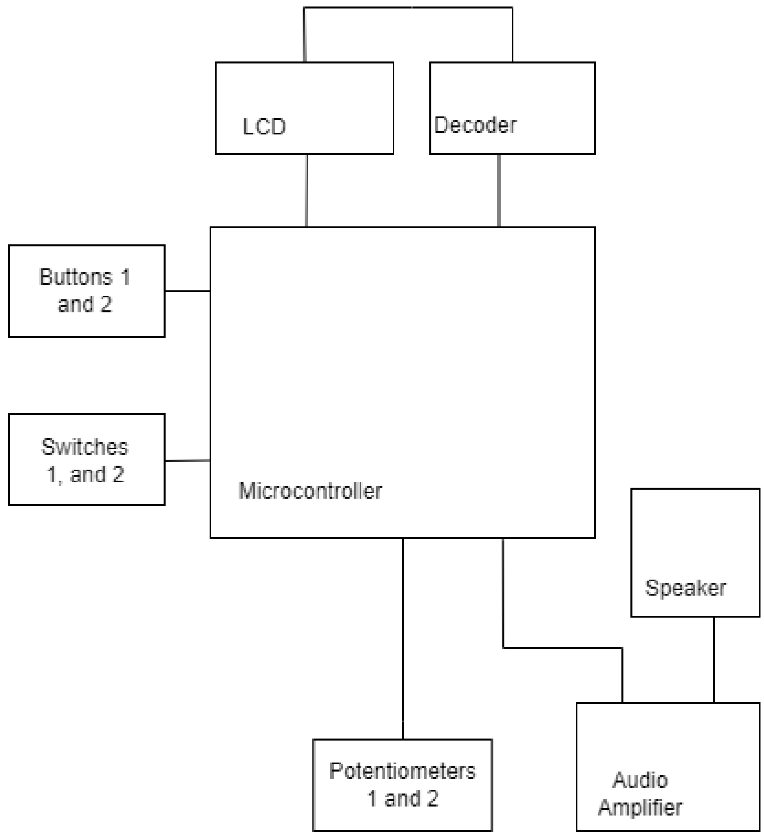
3.3 Theory of Operation

When the system is turned on, it initializes and enters the idle state, displaying a static message to await user input. When the launch button is pressed, the system transitions into game setup: random skyline heights, wind speed, and gorilla positions are generated.

The game is played in alternating turns. The active player can adjust the angle and velocity of the launch using the potentiometers which are sampled and displayed in real-time. Once the launch button is pressed, a banana is animated following the path using the input values from the potentiometers. The projectile will continue regardless of whether it is on screen, collision detection is performed to stop the banana when hitting the skyline or a gorilla.

The system either ends the round or switches turns depending on the collision type. When a hit is detected on a gorilla, the system enters the game over state, which displays a message awaiting a new round. A finite state machine manages all state transitions, and timer based interrupts ensure consistent physics timing and input handling.

The overall system architecture is illustrated in the block diagram below. It highlights the connections between the microcontroller and its peripheral components, including input controls, display, and audio output. This visual provides a high-level overview of how data flows between hardware modules during operation.



3.4 Design Alternatives

Several options were considered but ultimately not implemented. These included:

- A scoring system. While it was considered early on, it was not included in the final design as it fell outside the scope of the project requirements.

- Avoiding the use of the math.h library by implementing all necessary functions manually. However, this was not pursued, as it would have introduced unnecessary complexity and significantly increased development time without adding meaningful value to the project.

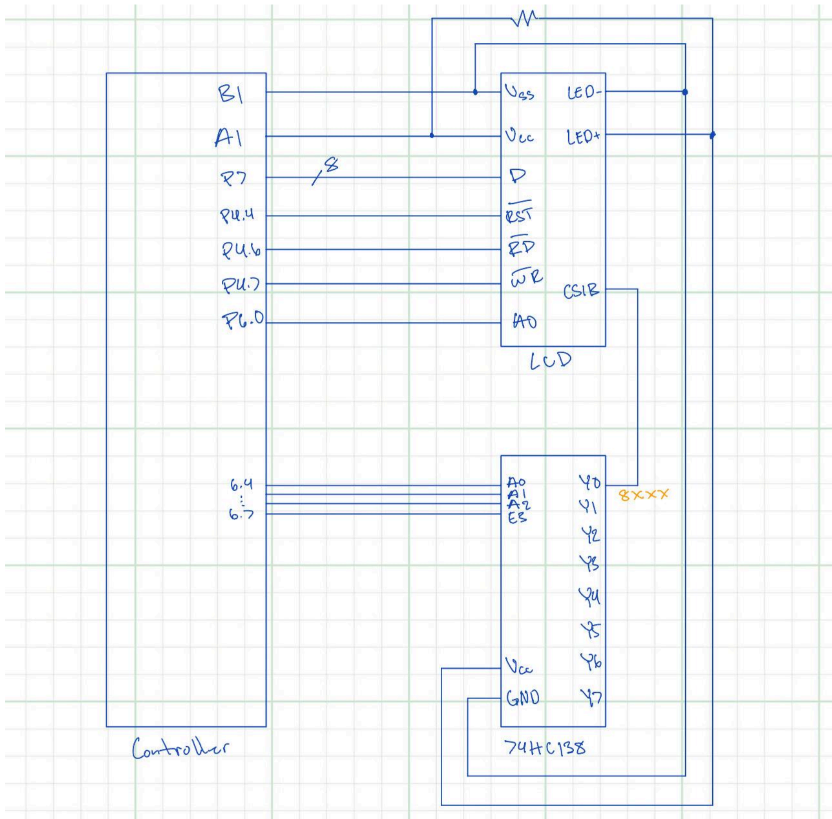
- More realistic audio effects for the launch and explosion events. This would have required more sophisticated waveform synthesis or the integration of pre-recorded samples. However, due to the project's focus on gameplay functionality, this feature was deemed out of scope. The simpler tones currently used fulfill the basic requirement of providing audio feedback without consuming excessive resources or development time

4. Design Details

4.1 Hardware Design

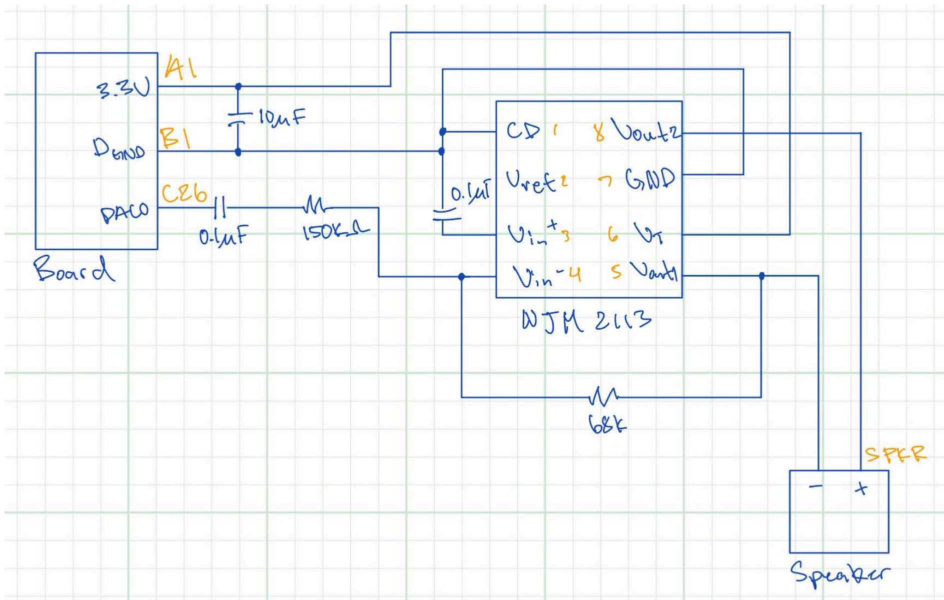
4.1.1 Bus Decoder and LCD

The figure below shows the connection between the microcontroller, LCD, and 74HC138 decoder. The microcontroller uses I/O ports P6.4–P6.7 to drive the decoder's select lines (A0–A2 and enable). When the correct combination is set (For address range 0x8XXX), the decoder activates the LCD's chip select (CS). Data and control signals (D, RD, WR, RESET) are connected directly.



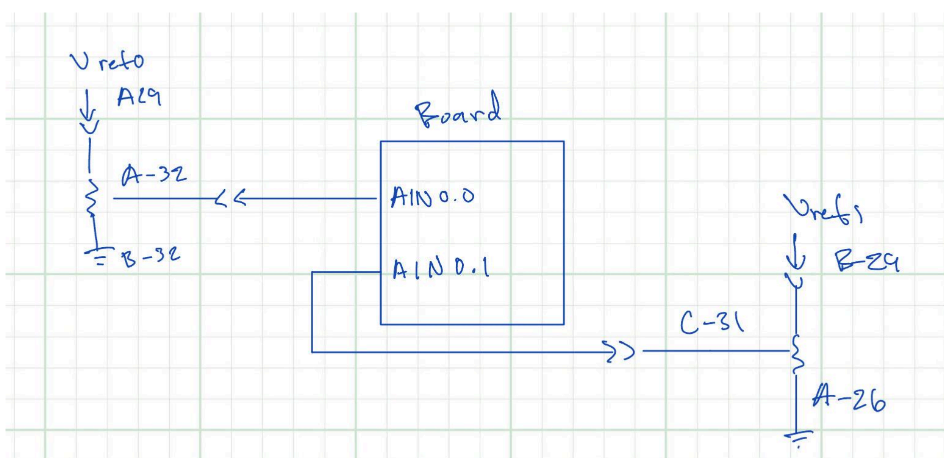
4.1.2 Audio Amplifier

The figure shows the DAC0 output connected to an NJM2113 audio amplifier. The signal is AC-coupled and filtered before entering the amplifier's Vin+ pin. The chip is powered from 3.3 V with proper decoupling, and the amplified output (Vout1) drives a speaker through a 68 kΩ resistor. This setup enables basic audio playback from the microcontroller.



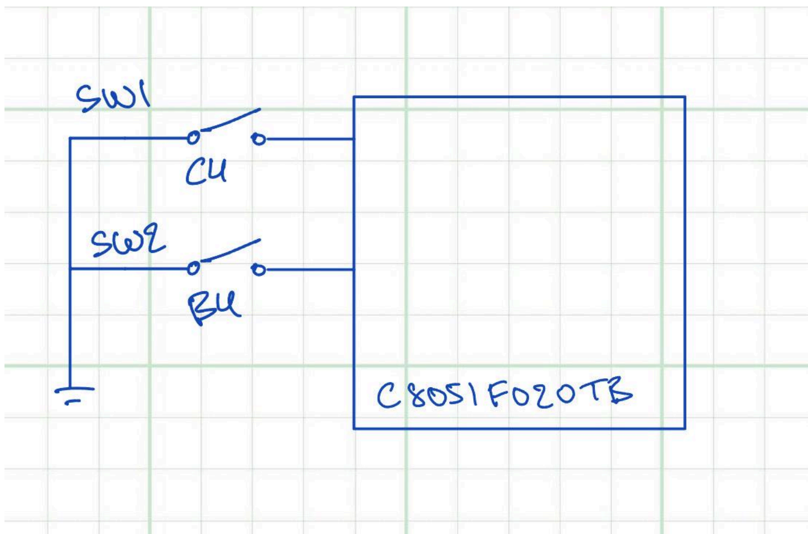
4.1.2 Potentiometer Circuit

This figure shows two potentiometers connected to AIN0.0 and AIN0.1, supplying analog input signals for ADC sampling. Pot 1 and Pot 2 are tied to Vref0 and Vref1, respectively, and provide adjustable voltages for real-time input control



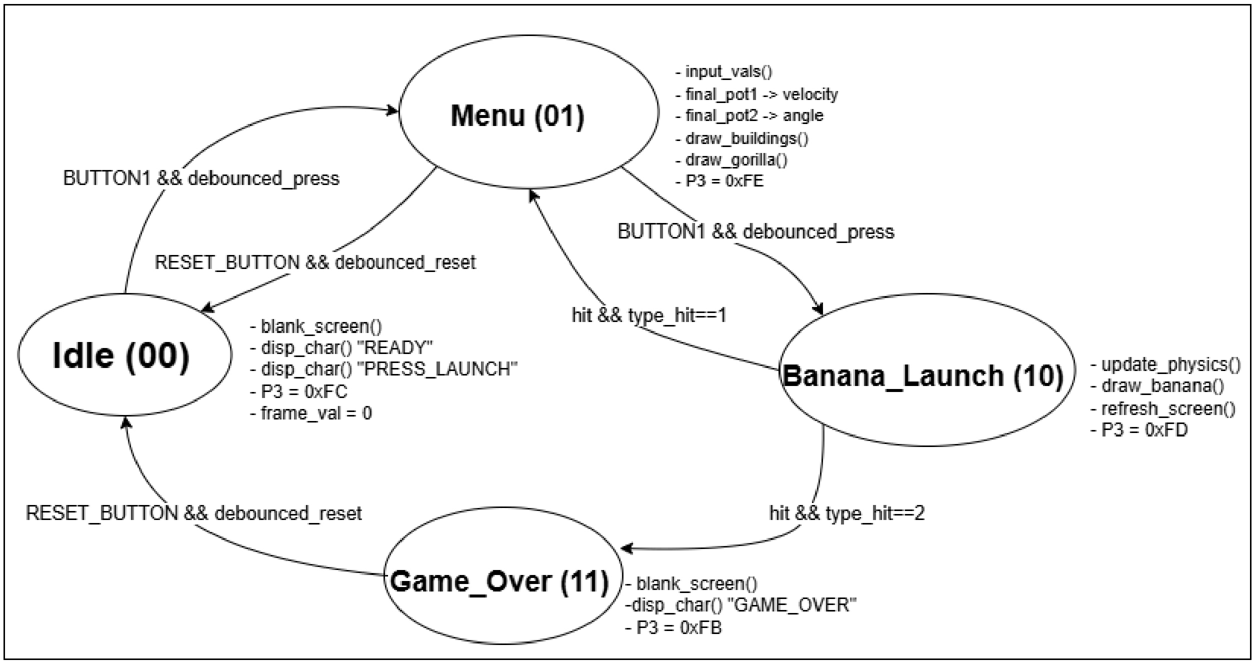
4.1.2 Dip Switch Circuit

This figure shows two switches used as wind enable controls, one for each player. Each switch connects to a digital input pin on the microcontroller. If either switch is turned off, wind is disabled for both players by forcing the wind value to zero in software.



4.2 Software Design

The Gorilla Game is built around a finite state machine (FSM) that governs the main gameplay flow. The system uses two global state bits, main_state_bit0 and main_state_bit1, to encode the current state of the game. This allows for four distinct states:



00: IDLE

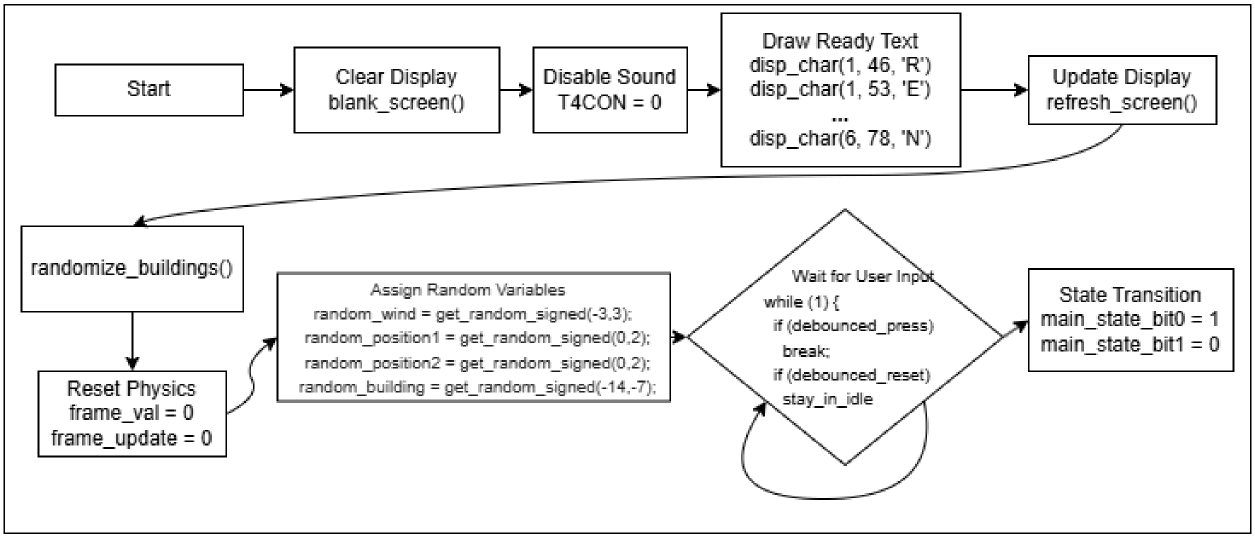
01: MENU (Player Input)

10: LAUNCH (Projectile Motion)

11: GAME OVER

This structure was chosen because each state is able to handle a specific part of the game logic, with transitions triggered by user input (debounced_press or debounced_reset) or internal game events such as collisions. The FSM structure provides a clean and predictable way to manage the game's control flow while maintaining modularity and responsiveness.

4.2.1 IDLE State (00)

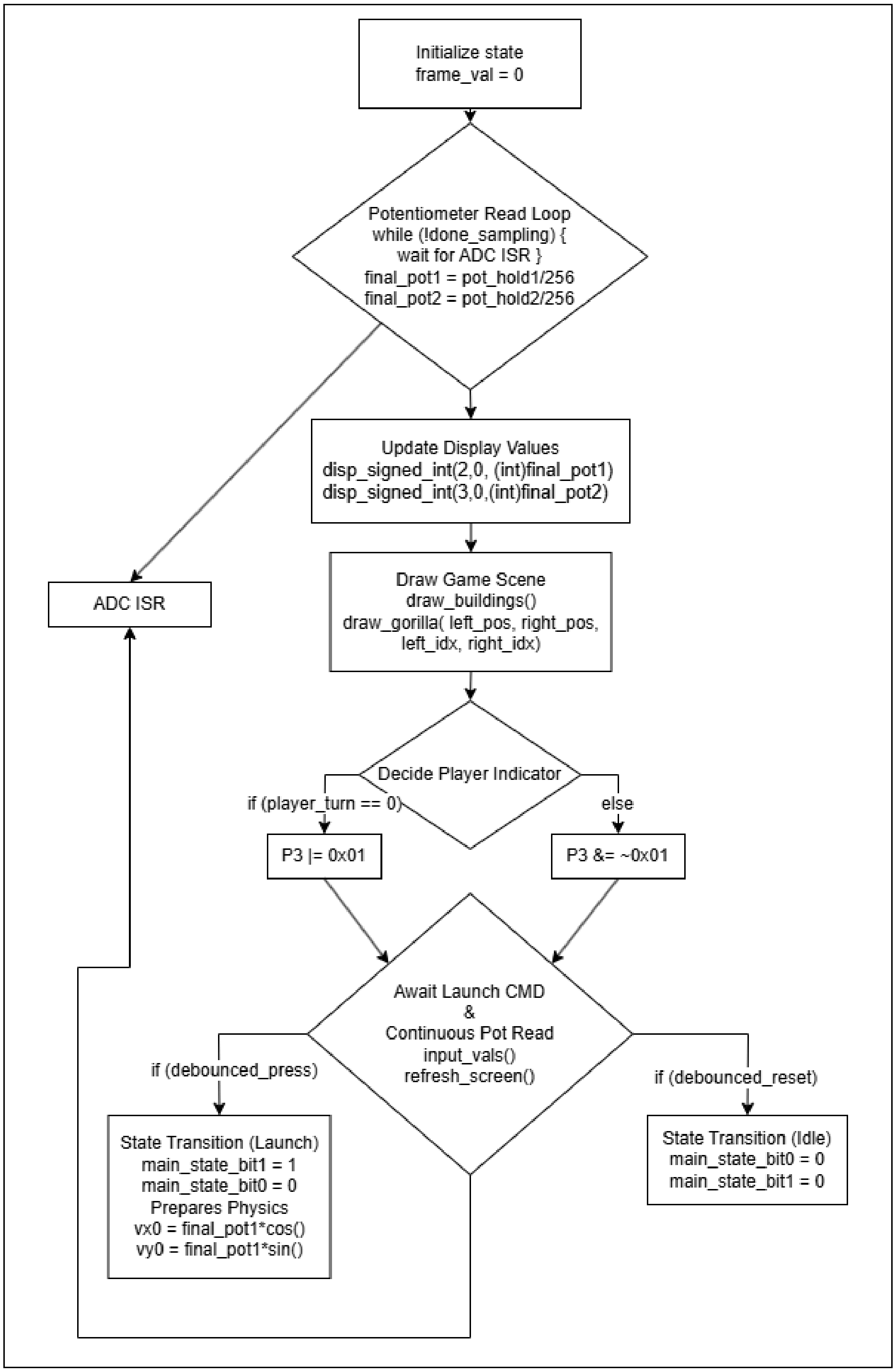


The IDLE state serves as both the game's entry point and its reset condition. When active, it calls blank_screen to clear the display and disables audio by setting T4CON to zero. It uses disp_char to print a static welcome message instructing the player to press the launch button, and then updates the screen using refresh_screen.

It prepares for the next state by calling randomize_buildings, and generates values for the variables random_wind, random_position1, random_position2, and random_building. These values are intentionally computed here so that the MENU state can immediately begin rendering without additional delay or setup.

The state transitions when either debounced_press or debounced_reset is detected. If reset is pressed, it clears the state bits and sets player_turn to zero. If launch is pressed, it sets the FSM to MENU state and resets frame_val and frame_update in preparation for physics timing.

4.2.2 MENU State (01)

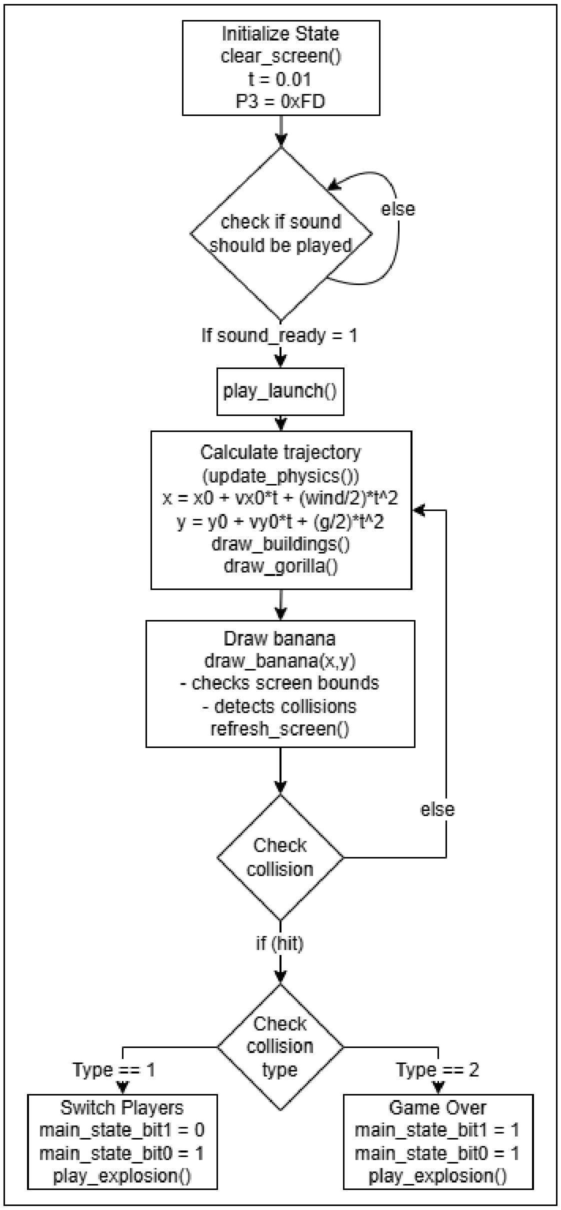


The MENU state prepares the game for a player's turn. It clears the screen using blank_screen and resets frame_val. Gorilla positions are derived from random_position1 and random_position2, which determine their drawing locations and building indices.

Potentiometer inputs are sampled via input_vals, updating final_pot1 and final_pot2, which are then used to calculate the launch vector. The HUD is drawn using disp_char and disp_signed_int, showing wind speed, player number, and the selected speed and angle. The scene is rendered with draw_buildings and draw_gorilla.

If either wind switch is off, random_wind is set to zero. Launch vector components vx0 and vy0 are computed from the input values, and x-direction is reversed if it's player two's turn.
The FSM waits for input. A launch press sets sound_ready and transitions to the LAUNCH state. A reset returns the game to IDLE and resets the player turn.

4.2.3 LAUNCH State (10)



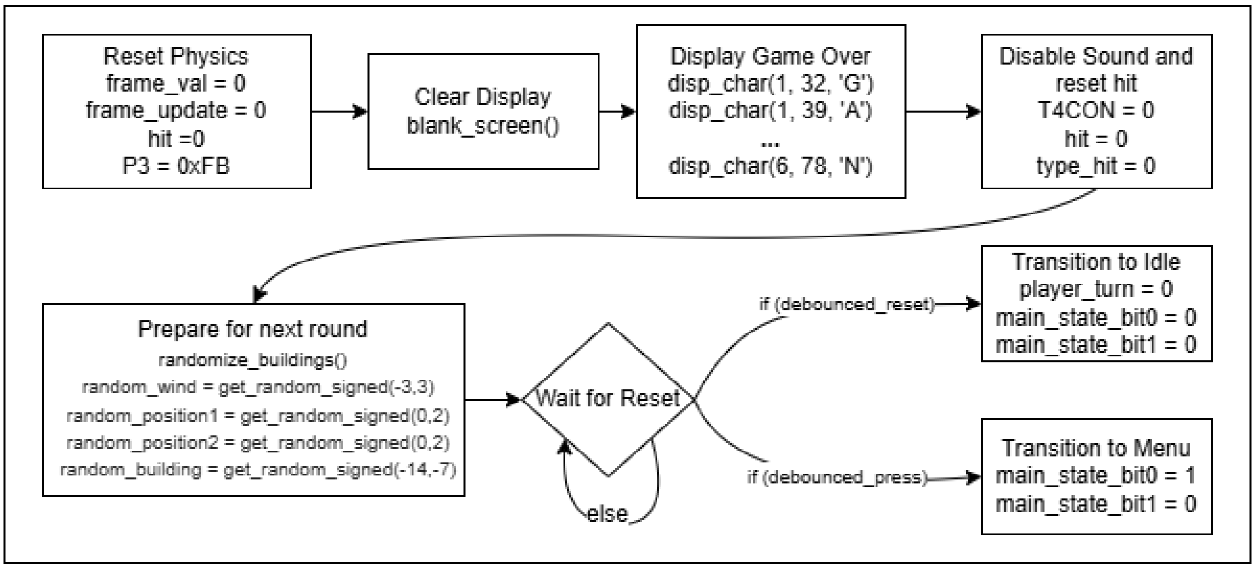
The LAUNCH state handles projectile motion and collision detection. It begins by clearing the screen with blank_screen, and if sound_ready is set, it triggers a launch sound using play_launch.

The starting column and building index are selected based on the current player's turn. These values determine the banana's launch position (current_x, current_y), set just above the player's gorilla to avoid immediate collision.

The buildings and gorillas are redrawn using draw_buildings and draw_gorilla, and the projectile's motion is updated using update_physics, which calculates the banana's new position over time.

The screen is refreshed with refresh_screen. If a collision is detected (hit is set), the system calls play_explosion. If the collision was with a gorilla (type_hit == 2), the game transitions to the GAME OVER state. Otherwise, the player is switched and the FSM returns to MENU. If reset is pressed at any time, the system reverts to IDLE and resets the player turn.

4.2.4 GAME OVER State (11)



The GAME OVER state is entered when a gorilla is hit. It clears the screen using blank_screen, resets frame tracking by setting frame_val and frame_update to zero, and displays a static “GAME OVER” message using disp_char.

All sound is disabled by clearing T4CON, and the flags hit and type_hit are reset. The system prepares for the next round by calling randomize_buildings and generating new values for random_wind, random_position1, random_position2, and random_building.

The FSM remains in this state until either the launch or reset button is pressed. A launch press moves the system to MENU to start a new round, while a reset returns it to IDLE and resets the player turn.

4.2.5 Graphics Subsystems

- **draw_buildings()** - Paints a city skyline on the LCD. It starts with a solid platform at the screen's bottom. Then for every column (0–127), it calculates which 16-column block it belongs to, uses building_heights_ram[] to determine height, and renders pixels upward. Roofs use a solid fill (0x80), and floors use alternating window patterns (0x18 or 0xFF). This was done to enhance the visuals.

```c

void draw_buildings() {
    unsigned char col, page, height;
    unsigned int addr;

    // Platform line (bottom of screen)
    for(col = 0; col < 128; col++) {
        screen[896 + col] = 0xFF;  // Page 7 (Y=56-63)
    }

    // Draw buildings with varying heights
    for(col = 0; col < 128; col++) {
        // Get height for current 16-column block
       // height = building_heights[(col/16) % 8];
  height = building_heights_ram[(col / 16) % 8];


        // Draw building with window pattern
        for(page = 0; page < height; page++) {
            addr = (7 - page) * 128 + col;

            // Solid top layer for building roof
            if(page == height - 1) {
                screen[addr] = 0x80;  // Solid top
            }
            // Window pattern for lower floors
            else {
                screen[addr] = (col % 4 == 1 || col % 4 == 2) ? 0x18 : 0xFF;
            }
        }
    }
}
```

- **draw_gorilla()** - Places two pixel-art gorillas on top of their respective buildings. It uses building_heights_ram[] to calculate how high each building is and determines which LCD page (vertical position) the gorilla sprite should occupy. If a building has at least two "floors" of height, the function renders a 13-column-wide sprite from the gorilla[] array directly above that building. It also draws an additional gorilla head segment (gorilla_head[]) on the page above to simulate a taller sprite without complex bit-shifting. Both gorillas are drawn independently using their assigned column positions and building indices. A check ensures that drawing does not exceed screen boundaries, preventing glitches if a gorilla is near the edge. This function is always called after draw_buildings() to ensure proper layering.

```c
void draw_gorilla(unsigned char LEFT_COL, unsigned char RIGHT_COL,
unsigned char BUILDING_INDEX1, unsigned char BUILDING_INDEX2) {
unsigned char height1 = building_heights_ram[BUILDING_INDEX1];
unsigned char height2 = building_heights_ram[BUILDING_INDEX2];

    unsigned char page1 = 7 - height1 + 1;  // Platform-relative position for gorilla 1
    unsigned char page2 = 7 - height2 + 1;  // Platform-relative position for gorilla 1
unsigned char col, upper_page1, upper_page2;

    // Draw first gorilla if building has at least 2 pages height
    if(height1 >= 2) {
        // Draw all 13 columns of the gorilla
        for(col = 0; col < 13; col++) {
            // Ensure we don't draw beyond screen boundaries
            if((LEFT_COL + col) < SCREEN_WIDTH) {
                screen[page1 * 128 + LEFT_COL + col] = gorilla[col];
            }

        }

        // TEST: Fill page above gorilla with solid block (head area)
        upper_page1 = page1 - 1;
        if(upper_page1 >= 0) { // Prevent underflow
            for(col = 0; col < 13; col++) {
                if((LEFT_COL + col) < SCREEN_WIDTH) {
                    screen[upper_page1 * 128 + LEFT_COL + col] = gorilla_head[col]; // All pixels on
                }
            }
        }
    }

    // Draw second gorilla
    if(height2 >= 2) {
        // Draw all 13 columns of the gorilla
        for(col = 0; col < 13; col++) {
            // Ensure we don't draw beyond screen boundaries
if((RIGHT_COL + col) < SCREEN_WIDTH) {
screen[page2 * 128 + RIGHT_COL + col] = gorilla[col];
}
        }

        // TEST: Fill page above gorilla with solid block (head area)
        upper_page2 = page2 - 1;
        if(upper_page2 >= 0) { // Prevent underflow
            for(col = 0; col < 13; col++) {
                if((RIGHT_COL + col) < SCREEN_WIDTH) {
                    screen[upper_page2 * 128 + RIGHT_COL + col] = gorilla_head[col]; // All pixels on
                }
            }
        }
    }
}
```

- **draw_banana() -** Renders a banana sprite in real-time using projectile coordinates generated by the update_physics() function. It calculates pixel position and shift based on Y, builds a mask, and draws to the LCD buffer. If a pixel collision is detected with a building or gorilla sprite, hit is set and type_hit is updated.

```
int draw_banana (int x, int y) {
int mask;
int page = y >> 3;
int shift = y & 7;
int i = page * 128 + x;
char k;
hit = 0;

for (k = 0; k < 3; k++) {
if (x + k < 0 || x + k > 127) {
hit =1;
break;
}
if (y + k < 0 || y + k > 64) {
hit = 0;
return hit;
}

mask = banana[k] << shift;

// Check if drawing this pixel will overwrite a non-zero screen pixel (collision)
if ((screen[i + k] & mask) != 0) {
collided_byte = screen[i + k];

// Building patterns
if (collided_byte == 0x18 || collided_byte == 0xFF || collided_byte == 0x80) {
type_hit = 1; // Building
hit = 1; // hit building
} else {
// Gorilla patterns
for (gorilla_check = 0; gorilla_check < 13; gorilla_check++) {
if (collided_byte == gorilla[gorilla_check] || collided_byte == gorilla_head[gorilla_check]) {
type_hit = 2; // Gorilla
break;
}
}
hit = 1; // hit gorila
}
break;
}

if (y > 0 && y <= 63) {
screen[i + k] = mask;
}
if (y > -8 && y <= 55) {
screen[i + k + 128] |= mask >> 8;
}

hit = 0;
}
//hit = 0;

return hit;
}
```

4.2.6 Timing and Input

- **Timer0_ISR() -** Handles debounce logic for the launch and reset buttons. Also updates frame_val and frame_update every 10ms to synchronize physics updates. frame_val serves as a global time accumulator updated at a fixed 10ms interval. This enables consistent physics simulation using elapsed time rather than per-frame iteration, improving accuracy and simplifying the update_physics logic..

```
// Timer0 ISR - handles debouncing
void Timer0_ISR(void) interrupt 1 {
    static bit last_button_state = 1;
static bit last_reset_state = 1;
    TH0 = 0xF1;    // Reload timer high byte
    TL0 = 0x9A;    // Reload timer low byte
//frame_update = 0;

    if (frame_counter < 5) {
        frame_counter += 1;

    } else {
        frame_update = 1;
        frame_counter = 0;
frame_val += 1;
    }

    // --- Debounce logic here ---
    if (BUTTON1 != last_button_state) {
        if (++debounce_counter >= DEBOUNCE_TIME) {
            last_button_state = BUTTON1;
            if (!BUTTON1)
                debounced_press = 1;
            debounce_counter = 0;
        }
    } else {
        debounce_counter = 0;
    }

```

4.2.7 Potentiometer Input (ADC)

- **adc_isr() -** Alternates between two potentiometer channels. Samples each 256 times to reduce noise. Sets done_sampling when complete.

```
void adc_isr(void) interrupt 15 {
    AD0INT = 0;                                // Reset ADC interrupt flag
    adc_val = (ADC0H << 8) | ADC0L;            // Combine ADC high and low registers

    if (AMX0SL == 0x00) {
        // pot1 reading
        ADC0CF = 0x41;    // Configure gain as needed for pot1
        pot_hold1 += adc_val;
        AMX0SL = 0x01;    // Switch to pot2 for next conversion
    } else if (AMX0SL == 0x01) {
ADC0CF = 0x40;
pot_hold2 += adc_val;
AMX0SL = 0x00;
}

    count++;  // One ADC conversion completed

if (count == 512) { // Now 512 (256 pairs)
        pot_sample1 = pot_hold1 / 256;  // correctly average the ADC readings

    pot_sample2 = pot_hold2 / 256;  // if you need a second channel
    pot_hold1 = pot_hold2 = 0;
    count = 0; // reset count
    done_sampling = 1; // signal that samples are ready
}



}
```

- **input_vals()** - Scales averaged ADC values to match game parameters:
    - final_pot1: Speed (10–41)
    - final_pot2: Angle (0–90°)

```
void input_vals(void){
if (done_sampling == 1) {
        done_sampling = 0; //resets the sampling signal

final_pot1 = 10L + ((pot_sample1 * 31L) / 4096);// for range 20-40

//final_temp = final_temp*(9L/5L) + 32;

final_pot2 = ((pot_sample2*90L)/4096);



    }
}
```

4.2.8 Physics Engine

- **update_physics()** - Computes the projectile's X and Y using kinematic formulas based on initial velocity, wind, gravity, and elapsed time (frame_val). Then calls draw_banana() to render the new banana position. This was chosen so that we can use the timer0 frame_val to keep track of time. A time-based update approach was chosen instead of frame-based stepping to allow consistent physics calculations regardless of frame rendering or input lag. This method also improves portability to systems with different clock rates.

```
void update_physics () {


// first calculate the current velocity amplitudes
    xdata float dt = 0.01; // account for the 10 ms scaling
    xdata float time_elapsed = frame_val * dt;


x_position = current_x + vx0 * time_elapsed + ((float)random_wind / 2) * time_elapsed * time_elapsed;
y_position = current_y + vy0 * time_elapsed + ((float)g / 2) * time_elapsed * time_elapsed;
        draw_banana(x_position, y_position);
refresh_screen();


}
```

4.2.9 Audio System

- **play_launch() & play_explosion() -** Set playback frequencies and initialize waveform parameters to simulate sound effects. These functions enable Timer4 and load values into the DAC. Simple tone generation was chosen due to limited DAC resolution and to conserve MCU cycles for gameplay logic. More complex audio synthesis would have interfered with real-time responsiveness or required external hardware.

- **Timer4_ISR() -** Uses a sine table and amplitude envelope to generate a decaying tone. Sends output to DAC0 for audio playback. Each frame reduces volume until the sound fades out.

```
// Timer4 overflow ISR:
void Timer4_ISR(void) interrupt 16 {
    T4CON &= ~0x80; // Clear Timer4 interrupt flag

    if (duration == 0) {
        DAC0H = 128;  // Set DAC output to silence (midpoint)
        return;
    }

    DAC0H = (sine[phase] * amplitude) >> 8;

    if (phase < sizeof(sine) - 1) {
        phase++;
    } else {
        phase = 0;
        duration--;
        amplitude = (amplitude * 251) >> 8;
        if (amplitude > 0) {
            amplitude--;
        }
    }
}
```

5. Testing

5.1 LCD Display

- Test Procedure: Observe LCD during gameplay and after a reset to look for a display of correct static and dynamic elements.
- Expected Result: “Ready” and “Press Launch to Start” should be displayed after reset. During gameplay, skyline, gorillas, wind, angle, velocity, player indicator, banana, “Game over” and “Press Launch Button” should all correctly be displayed at their respective times.
- Actual Observation: All expected information is correctly displayed at the appropriate time. Banana trajectory and final Game Over screen render as expected.

5.2 Button Functionality

- Test Procedure: Press reset and launch buttons in each game state.
- Expected Result: Reset should bring the game to the idle state, and launch should either begin the game or launch the banana.

- Actual Observation: Button presses activate the correct state transitions, and the debounce logic functions well.

5.3 Potentiometer Input

- Test Procedure: Rotate each potentiometer along its entire range and monitor the LCD display for velocity and angle changes.
- Expected Result: Displayed values should cover the desired ranges and adjust smoothly.
- Actual Observation: The potentiometers give correct input ranges. Sampling noise is sufficiently reduced.

5.4 Wind Control

- Test Procedure: Toggle the DIP switches for each player and observe the effect on the wind speed.
- Expected Result: If the switch is off, the wind effect should be zero for that player's turn. If the switch is on, the wind effect should be a random value.
- Actual Observation: Wind is correctly enabled/disabled and is randomly generated when it is enabled.

5.5 Game Initialization

- Test Procedure: Press launch button from idle state multiple times to observe the gorilla and skyline positions.
- Expected Result: The skyline, and gorilla positions should be randomized each time.
- Actual Observation: The positions vary as expected.

5.6 Player Turn Logic and Projectile Trajectory

- Test Procedure: Observe gameplay to confirm player turn transitions, and vary angle/velocity values.
- Expected Result: Player one starts, and the turn alternates after every launch. Trajectory shape should reflect which value(s) were changed.
- Actual Observation: Player turn logic works as expected. The trajectory seems natural and is affected by the wind.

5.7 Sound Output

- Test Procedure: Listen for audio output during launch and collision.
- Expected Result: A sound should play on launch, and a separate sound should play on collision.
- Actual Observation: The sounds are played at the desirable times, and are distinguishable

---

5.8 Collision and Game Over Logic

- Test Procedure: Intentionally trigger a collision and observe the result.

- Expected Result: Depending on the collision type, the game should either end or switch players.
- Actual Observation: Collisions are triggered reliably and when hitting a gorilla the game ends, and when hitting the skyline it becomes the other player's turn.

---

6. Conclusion

The project was successfully completed, with all primary functional interface requirements implemented and verified through the testing procedures outlined above. The game behaves as intended—responding accurately to player input, delivering smooth gameplay, and providing an intuitive user interface.

System performance has been stable with no detected issues. Both the graphical and audio outputs met expectations, and the input systems performed reliably. The finite state machine proved highly effective in managing game flow, while timers ensured consistent timing for physics and input handling.

The main area for future improvement lies in the code structure. Some sections could benefit from cleaner formatting and the elimination of redundancy. However, these adjustments are primarily stylistic and do not impact gameplay or system performance. The current implementation meets all functional and timing requirements within the constraints of the 8052 platform.

Appendix

Schematics are not included in this section as they are already covered in section 4 of this document.

```
#include <C8051F020.h>
#include <stdlib.h>
#include "lcd.h"
#include <math.h>

#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define DEBOUNCE_TIME 5  // 5ms debounce time
#define g 10 // rounded from 9.8m/s/s
#define PI 3.14159265
#define F800  0x9400  // 800 Hz “whoosh” (launch)
#define F400  0x2883  // 400 Hz “boom”   (hit/explosion)
#define F200  0x5115  // 200 Hz “signal” (game over)
#define F120  0xD2C0    // 120 Hz explosion sound
```

```
// CORRECTED FREQUENCIES (22.1184MHz clock)
#define F800 0x9400  // 800Hz: Verified
#define F635 0x7C22  // 635Hz: CORRECTED (was 0x7C22)
code unsigned char sine[] = {176,217,244,254,244,217,176,128,80,39,12,2,12,39,80,128};

xdata unsigned char phase = sizeof(sine) -1;
xdata unsigned int duration = 0;
xdata unsigned int amplitude = 255;
xdata float angle, env;
xdata long dac;

sfr TMR3CN  = 0x91;  // Timer 3 Control SFR from the F020 datasheet
sfr TMR3RLL = 0x92;  // reload low byte
sfr TMR3RLH = 0x93;  // reload high byte

#define TR3_BIT 0x04  // bit2 = run control
#define TF3_BIT 0x80  // bit7 = overflow flag

sbit BUTTON1 = P2^6;
sbit wind_enable1 = P1^0;
sbit wind_enable2 = P1^1;
bit wind_enable = 0;
int a;

unsigned int lfsr = 0xACE1u;  // Initial seed (can be any non-zero value)
unsigned int range;
unsigned lsb;
unsigned int building_index;
unsigned int start_col;
unsigned int height;

bit hit = 0;
bit player_turn = 0;

// kinematics equations
long current_vx, current_vy, next_vx, next_vy;
long current_x, current_y, old_x, old_y;
float vx, vy, vx0, vy0;
float t = 0.01;

volatile bit sound_ready = 0;

// Global state variables
bit main_state_bit0 = 0;
bit main_state_bit1 = 0;
volatile bit debounced_press = 0;
volatile unsigned char debounce_counter = 0;

sbit RESET_BUTTON = P2^5;
xdata unsigned char reset_debounce_counter = 0;
volatile bit debounced_reset;

// Signed to match the random generator
signed char random_wind;
signed char random_position1;
signed char random_position2;

signed char random_building;

xdata float pi = 3.14159;

unsigned int left_position, right_position;
```

```
unsigned int left_index, right_index;
xdata unsigned int x_position, y_position;

unsigned int banana_height,left_height, right_height;

volatile unsigned int frame_counter = 0;
volatile bit frame_update = 0;

xdata volatile unsigned int frame_val = 0;

static bit banana_initialized;

xdata float input_speed, input_angle;
xdata float angle_radians;

xdata unsigned char collided_byte, gorilla_check, type_hit;

xdata unsigned long pot_sum1,pot_sum2    = 0;
xdata unsigned char sample_count = 0;
// Replace 'bit' with unsigned char for xdata storage
xdata unsigned char data_ready = 0;

xdata unsigned pot_sample1, pot_sample2;
xdata long pot_hold1, pot_hold2 = 0;  // hold pot values for average
xdata int adc_val;
xdata int count = 0;
// Similarly, replace 'bit' with unsigned char for done_sampling
xdata unsigned char done_sampling = 0;

xdata long final_pot1,final_pot2;

xdata volatile unsigned char building_heights_ram[16];

code unsigned char building_heights[16] = {
    3, 4, 3, 4,  // First 4 buildings (columns 0-31)
    4, 2, 5, 3,   // Next 4 buildings (columns 32-63)
    4, 5, 2, 5,   // Columns 64-95
    3, 4, 5, 1    // Columns 96-127
};

code unsigned char gorilla[] = {
    0x04,  // 00000100
    0x0A,  // 00001010
    0xD1,  // 11010001
    0xB5,  // 10110101
    0x99,  // 10011001
    0xC9,  // 11001001  ? Column 5: Added 2 top bits (11000000 | original 00001001)
    0x25,  // 00100101
    0x49,  // 01001001
    0x99,  // 10011001
    0xB5,  // 10110101  ? Column 9: Added 2 top bits (10100000 | original 00110101)
    0xD1,  // 11010001
    0x0A,  // 00001010
    0x04   // 00000100
};

code unsigned char gorilla_head[] = {
0x00,
0x00,
0x00,
0x00,
0xc0,
```

```
0x20,
0x20,
0x20,
0xc0,
0x00,
0x00,
0x00,
0x00
};

code unsigned int gorilla_positions[2][3] = {
    {0, 16, 32},   // Left buildings 1-3 (columns 0, 16, 32)
    {80, 96, 112}  // Right buildings 6-8 (columns 80, 96, 112)
};

// Call this whenever you need a random number 1-10
// Generates signed numbers between min and max (inclusive)
signed int get_random_signed(signed int min, signed int max) {
    // Swap if min > max
    if(min > max) {
        signed char temp = min;
        min = max;
        max = temp;
    }

    // Calculate range size
    range = (unsigned int)(max - min) + 1;

    // Advance LFSR (same as before)
    lsb = lfsr & 1;
    lfsr >>= 1;
    if(lsb) lfsr ^= 0xB400u;

    // Generate value in range
    return (signed int)((lfsr % range) + min);
}

void draw_buildings() {
    unsigned char col, page, height;
    unsigned int addr;

    // Platform line (bottom of screen)
    for(col = 0; col < 128; col++) {
        screen[896 + col] = 0xFF;  // Page 7 (Y=56-63)
    }

    // Draw buildings with varying heights
    for(col = 0; col < 128; col++) {
        // Get height for current 16-column block
       // height = building_heights[(col/16) % 8];
  height = building_heights_ram[(col / 16) % 8];

        // Draw building with window pattern
        for(page = 0; page < height; page++) {
            addr = (7 - page) * 128 + col;

            // Solid top layer for building roof
            if(page == height - 1) {
                screen[addr] = 0x80;  // Solid top
            }
            // Window pattern for lower floors
            else {
```

```
                screen[addr] = (col % 4 == 1 || col % 4 == 2) ? 0x18 : 0xFF;
            }
        }
    }
}

void randomize_buildings() {
int i;
    for (i = 0; i < 16; i++) {
        // You can tweak min/max height values here to ensure valid drawings (e.g., between 1 and 5)
        building_heights_ram[i] = get_random_signed(2, 5);
    }
}

void draw_gorilla(unsigned char LEFT_COL, unsigned char RIGHT_COL,
unsigned char BUILDING_INDEX1, unsigned char BUILDING_INDEX2) {

 /*
    unsigned char height1 = building_heights[BUILDING_INDEX1];
    unsigned char height2 = building_heights[BUILDING_INDEX2];*/

unsigned char height1 = building_heights_ram[BUILDING_INDEX1];
unsigned char height2 = building_heights_ram[BUILDING_INDEX2];

    unsigned char page1 = 7 - height1 + 1;  // Platform-relative position for gorilla 1
    unsigned char page2 = 7 - height2 + 1;  // Platform-relative position for gorilla 1

unsigned char col, upper_page1, upper_page2;

    // Draw first gorilla if building has at least 2 pages height
    if(height1 >= 2) {
        // Draw all 13 columns of the gorilla
        for(col = 0; col < 13; col++) {
            // Ensure we don't draw beyond screen boundaries
            if((LEFT_COL + col) < SCREEN_WIDTH) {
                screen[page1 * 128 + LEFT_COL + col] = gorilla[col];
            }
        }
        // TEST: Fill page above gorilla with solid block (head area)
        upper_page1 = page1 - 1;
        if(upper_page1 >= 0) { // Prevent underflow
            for(col = 0; col < 13; col++) {
                if((LEFT_COL + col) < SCREEN_WIDTH) {
                    screen[upper_page1 * 128 + LEFT_COL + col] = gorilla_head[col]; // All pixels on
                }
            }
        }
    }

    // Draw second gorilla
    if(height2 >= 2) {
        // Draw all 13 columns of the gorilla
        for(col = 0; col < 13; col++) {
            // Ensure we don't draw beyond screen boundaries
if((RIGHT_COL + col) < SCREEN_WIDTH) {
screen[page2 * 128 + RIGHT_COL + col] = gorilla[col];
}
        }

        // TEST: Fill page above gorilla with solid block (head area)
        upper_page2 = page2 - 1;
        if(upper_page2 >= 0) { // Prevent underflow
```

```
            for(col = 0; col < 13; col++) {
                if((RIGHT_COL + col) < SCREEN_WIDTH) {
                    screen[upper_page2 * 128 + RIGHT_COL + col] = gorilla_head[col]; // All pixels on
                }
            }
        }
    }
}

code int banana[4] = {0x06, 0xf, 0x9};

int n,m;

int draw_banana (int x, int y) {
int mask;
int page = y >> 3;
int shift = y & 7;
int i = page * 128 + x;
char k;
hit = 0;

for (k = 0; k < 3; k++) {
if (x + k < 0 || x + k > 127) {
hit =1;
break;
}
if (y + k < 0 || y + k > 64) {
hit = 0;
return hit;
}

//if (y >

mask = banana[k] << shift;

// Check if drawing this pixel will overwrite a non-zero screen pixel (collision)
if ((screen[i + k] & mask) != 0) {
collided_byte = screen[i + k];

// Building patterns
if (collided_byte == 0x18 || collided_byte == 0xFF || collided_byte == 0x80) {
type_hit = 1; // Building
hit = 1; // hit building
} else {
// Gorilla patterns
for (gorilla_check = 0; gorilla_check < 13; gorilla_check++) {
if (collided_byte == gorilla[gorilla_check] || collided_byte == gorilla_head[gorilla_check]) {
type_hit = 2; // Gorilla
break;
}
}
hit = 1; // hit gorila
}
//hit = 1;
break;
// return hit;  // Stop drawing on hit
}
//break;

// Draw to screen if no collision
if (y > 0 && y <= 63) {
screen[i + k] = mask;
```

```
}
if (y > -8 && y <= 55) {
screen[i + k + 128] |= mask >> 8;
}

hit = 0;
}
//hit = 0;

return hit;
}

// Key: Use cummulative sum of area under curver for integration
// to iteratively get the position
// a numerical integration approximation

// Using Brandon's idea for Euler's approximation for
// numerical integration.
void update_physics () {

// first calculate the current velocity amplitudes
  xdata float dt = 0.01; // account for the 10 ms scaling
  xdata float time_elapsed = frame_val * dt;
//vx = vx0 + w*t;

/*if (player_turn == 0) {
//current_x = old_x + w*t;
    x_position = current_x + vx0 * time_elapsed + ((float)random_wind / 2) * time_elapsed * time_elapsed;
    // Note: Adjust the sign of g as necessary (here it's added as in your original logic)
    //y_position = current_y + vy0 * time_elapsed + ((float)g / 2) * time_elapsed * time_elapsed;

} else {
x_position = current_x + vx0 * time_elapsed - ((float)random_wind / 2) * time_elapsed * time_elapsed;
}*/

x_position = current_x + vx0 * time_elapsed + ((float)random_wind / 2) * time_elapsed * time_elapsed;
y_position = current_y + vy0 * time_elapsed + ((float)g / 2) * time_elapsed * time_elapsed;
    draw_banana(x_position, y_position);
refresh_screen();
// draw_banana(current_x, 0);
// refresh_screen();

// then calculate the new position using Euler's

}

// LCD character display function (with bit reversal)
void disp_char(unsigned char row, unsigned char col, char ch) {
  int i = 128 * row + col;
  int j = (ch - 0x20) * 5;
  char k;

  for(k = 0; k < 5; k++) {
    screen[i + k] = font5x8[j + k]; // Fix font orientation
  }
}



void disp_signed_int(unsigned char row, unsigned char col, signed int x) {
```

```
    if(x < 0) {
        disp_char(row, col, '-');
        x = -x; // Convert to positive
      //  col += 8;
    } else {
        disp_char(row, col, '+');  // Explicit positive sign
    }
    // Move past sign (5 columns for char + 3 spacing)
    col += 8;

    // Display digits
    disp_char(row, col, (x/10) + '0');  // Tens place
    disp_char(row, col + 8, (x%10) + '0'); // Units place
}

// Timer0 Initialization
void Init_Timer0(void) {
    TMOD &= 0xF0;       // Clear Timer0 config
    TMOD |= 0x01;       // Mode 1: 16-bit timer
    TH0 = 0xF1;         // New values for 2ms @11.0592MHz
    TL0 = 0x9A;
    ET0 = 1;            // Enable Timer0 interrupt
    TR0 = 1;            // Start timer
    EA = 1;             // Global interrupts
}

// Timer0 ISR - handles debouncing
void Timer0_ISR(void) interrupt 1 {
    static bit last_button_state = 1;
static bit last_reset_state = 1;
    TH0 = 0xF1;    // Reload timer high byte
    TL0 = 0x9A;    // Reload timer low byte
//frame_update = 0;

    if (frame_counter < 5) {
        frame_counter += 1;

    } else {
        frame_update = 1;
        frame_counter = 0;
frame_val += 1;
    }

    // --- Debounce logic here ---
    if (BUTTON1 != last_button_state) {
        if (++debounce_counter >= DEBOUNCE_TIME) {
            last_button_state = BUTTON1;
            if (!BUTTON1)
                debounced_press = 1;
            debounce_counter = 0;
        }
    } else {
        debounce_counter = 0;
    }

if (RESET_BUTTON != last_reset_state) {
if (++reset_debounce_counter >= DEBOUNCE_TIME) {
last_reset_state = RESET_BUTTON;
if (!RESET_BUTTON) {
debounced_reset = 1;
reset_debounce_counter = 0;
}
```

```
}
} else {
reset_debounce_counter = 0;
}
}

// ADC ISR: Alternates between Temperature sensor and Potentiometer (AN1)
//   - AMX0SL == 0x01 selects AN1 (Potentiometer)
void adc_isr(void) interrupt 15 {
   AD0INT = 0;                          // Reset ADC interrupt flag
   adc_val = (ADC0H << 8) | ADC0L;         // Combine ADC high and low registers

   if (AMX0SL == 0x00) {
      // pot1 reading
      ADC0CF = 0x41;    // Configure gain as needed for pot1
      pot_hold1 += adc_val;
      AMX0SL = 0x01;    // Switch to pot2 for next conversion
   } else if (AMX0SL == 0x01) {
ADC0CF = 0x40;
pot_hold2 += adc_val;
AMX0SL = 0x00;
}

   count++;  // One ADC conversion completed

  //count++; // Incremented every ISR call
// since it alternates on every conversion, there is a
// count of 512 to sample each input 256 times each
if (count == 512) { // Now 512 (256 pairs)
      pot_sample1 = pot_hold1 / 256;  // correctly average the ADC readings

   pot_sample2 = pot_hold2 / 256;  // if you need a second channel
   pot_hold1 = pot_hold2 = 0;
   count = 0; // reset count
   done_sampling = 1; // signal that samples are ready
}
}
void input_vals(void){
if (done_sampling == 1) {
      done_sampling = 0; //resets the sampling signal

     // blank_screen(); //clears

     // final_pot1 = 55L + ((pot_sample1*31L)/4096);  // Using original 4096 denominator
      //final_temp = ((temp_sample*3406L) >> 14) - 241;
//final_pot1 = pot_hold1;
//final_temp = temp_sample;
final_pot1 = 10L + ((pot_sample1 * 31L) / 4096);// for range 20-40

//final_temp = final_temp*(9L/5L) + 32;

final_pot2 = ((pot_sample2*90L)/4096);
   }
}

void disp_int(unsigned char row, unsigned char col, int x){
disp_char(row, col, (x/10)+'0'); // get tens place
disp_char(row, col+8, (x%10)+'0'); // units place
}

void set_frequency(unsigned int freq) {
   RCAP4H = (freq >> 8) & 0xFF;
```

```
    RCAP4L = freq & 0xFF;
}
// Timer4 overflow ISR:
void Timer4_ISR(void) interrupt 16 {
    T4CON &= ~0x80; // Clear Timer4 interrupt flag

    if (duration == 0) {
        DAC0H = 128;  // Set DAC output to silence (midpoint)
        return;
    }

    DAC0H = (sine[phase] * amplitude) >> 8;

    if (phase < sizeof(sine) - 1) {
        phase++;
    } else {
        phase = 0;
        duration--;
        amplitude = (amplitude * 251) >> 8;
        if (amplitude > 0) {
            amplitude--;
        }
    }
}
void play_launch() {
// T4CON = 0x04;
// EIE2 = 0x06;
    set_frequency(F800);
    //duration = 2;  // 150ms "whoosh"
phase = 0;

    duration = 1;      // disable the normal envelope
    // plop out one quick tick and silence:
    //DAC0H = (sine[0] * 255) >> 8;
T4CON = 0x04;
    amplitude = 255;
}

void play_explosion() {
    set_frequency(F120);
    duration = 1;  // 200ms "boom"
    //amplitude = 127;
    //set_frequency(F800);
    //duration = 2;  // 150ms "whoosh"
phase = 0;

    //duration = 0;      // disable the normal envelope
    // plop out one quick tick and silence:
    //DAC0H = (sine[0] * 255) << 8;
    T4CON = 0x04;  // ? make sure Timer4 is enabled

    amplitude = 255;
}

void main() {
WDTCN = 0xde; // disable watchdog
WDTCN = 0xad;
XBR2 = 0x40; // enable port output
XBR0 = 4; // enable uart 0
OSCXCN = 0x67; // turn on external crystal
TMOD = 0x20; // wait 1ms using T1 mode 2
TH1 = -167; // 2MHz clock, 167 counts - 1ms
```

```
TR1 = 1;
while ( TF1 == 0 ) { } // wait 1ms
while ( !(OSCXCN & 0x80) ) { } // wait till oscillator stable
OSCICN = 8; // switch over to 22.1184MHz

// DAC stuff below

    // Timer 2 configuration (400ms update period)
    RCAP2H = -1843 >> 8;  // High byte of 64228
    RCAP2L = -1843;  // Low byte of 64228
    TR2 = 1;

    ADC0CN = 0x8C;  // ADC enabled, timer 2 overflow
    REF0CN = 0x03;  // Temp sensor, VREF enabled, Fig. 9.2
    AMX0CF = 0xc0;  // Single-ended inputs, and write (don't care)
    AMX0SL = 0x00;  // read and AIN0 (Fig. 5.6)

duration = 0;

// timer 4 stuff
RCAP4H = 0;
RCAP4L = 0;

T4CON = 0x04;

DAC0CN = 0x94;

// ADC0CF = 0x40;  // Default: Gain = 1, SAR clock = SYSCLK / 8
    IE = 0x80;      // Global interrupts
   // EIE2 = 0x02;    // ADC0 interrupt
EIE2 = 0x06;

RCAP4H = -1;
RCAP4L = -144;
    P3 = 0xff;  // Initialize LEDs

    Init_Timer0();

player_turn = 0;

wind_enable = 0;

amplitude = 0;
phase = 0;


init_lcd();


while(1) {
        // State machine
        //------------------------------------------------
        // IDLE State (00) - P3 = 11111100
        if (!main_state_bit1 && !main_state_bit0) {

  blank_screen();

T4CON = 0x00;
           P3 = 0xff;
           // Display code here
    // Draw "PRESS START" text
```

```
disp_char(1,46, 'R');
disp_char(1,53, 'E');
disp_char(1,60, 'A');
disp_char(1,67, 'D');
disp_char(1,74, 'Y');

disp_char(5,22, 'P');
disp_char(5,29, 'R');
disp_char(5,36, 'E');
disp_char(5,43, 'S');
disp_char(5,50, 'S');
disp_char(5,57, ' ');
disp_char(5,64, 'L');
disp_char(5,71, 'A');
disp_char(5,78, 'U');
disp_char(5,85, 'N');
disp_char(5,92, 'C');
disp_char(5,99, 'H');


disp_char(6,43, 'B');
disp_char(6,50, 'U');
disp_char(6,57, 'T');
disp_char(6,64, 'T');
disp_char(6,71, 'O');
disp_char(6,78, 'N');


          refresh_screen();
randomize_buildings();

  // get random values before moving on
  random_wind = get_random_signed(-3,3);
  random_position1 = get_random_signed(0,2);
  random_position2 = get_random_signed(0,2);
  random_building = get_random_signed(-14,-7);

 // current_wind();
          if (debounced_press) {
              main_state_bit0 = 1;
              debounced_press = 0;

    frame_val    = 0;
    frame_update = 0;
          } else if (debounced_reset) {
               main_state_bit1 = 0;
main_state_bit0 = 0;
               debounced_reset = 0;
player_turn = 0;
}
        }

        //-----------------------------------------------
        // PLAYER1 State (01) - P3 = 11111110
        else if (!main_state_bit1 && main_state_bit0) {
            P3 = 0xFE;
            blank_screen();
// duration = 0;

frame_val = 0;
left_position = gorilla_positions[0][random_position1];
right_position = gorilla_positions[1][random_position2];
```

```
left_index = random_position1; // corresponds to random position
right_index = 5 + random_position2; // 5 offset to only get far right 3 columns

disp_signed_int(0, 0, random_wind);

disp_char(0,23,'m');
disp_char(0,30,'/');
disp_char(0,37,'s');

input_vals();

disp_char(0, 100, 'P');
if (player_turn == 0) {
  disp_char(0, 108, '1');
} else {
  disp_char(0, 108, '2');
}

// disp_signed_int(0,0,random_position1);

// get pot val before too
// disp_signed_int(1,0,random_position2);

// get pot val before too
  // Corrected drawing order: buildings first, then gorillas
  draw_buildings();
  draw_gorilla(left_position, right_position,
left_index, right_index);

if (!wind_enable1 && !wind_enable2) {
random_wind = 0;
}

input_speed = 27;//30 feels like a good max
input_angle = 15;

angle_radians = final_pot2 * (PI / 180.0);

if (player_turn == 0) {
vx0 = final_pot1* cos(angle_radians);
} else {
vx0 = -final_pot1* cos(angle_radians);
random_wind = -random_wind;
}
vy0 = - final_pot1*sin(angle_radians);

hit = 0;
type_hit = 0;
disp_char(2, 45, 'S');
disp_signed_int(2, 62, final_pot1);

disp_char(3, 45, 'A');
disp_signed_int(3, 62, final_pot2);

refresh_screen();
// vx0 = 10;
// vy0 = -10;

frame_update = 0;

      if (debounced_press) {
        main_state_bit1 = 1;
```

```
main_state_bit0 = 0;
            debounced_press = 0;
T4CON = 0x04;
//play_launch();
sound_ready = 1;
         } else if (debounced_reset) {
            main_state_bit1 = 0;
main_state_bit0 = 0;
            debounced_reset = 0;
player_turn = 0;
}
      }

      //------------------------------------------------
      // LAUNCH State (10) - P3 = 11111101
      else if (main_state_bit1 && !main_state_bit0) {
         P3 = 0xFD;
blank_screen();

if (sound_ready) {
play_launch();
sound_ready = 0;
}

/* for (m=-2; m<128;m+=4){
for (n=12;n<64;n+=5){
draw_banana(m,n);

}
}*/

if (player_turn == 0) {
   start_col = left_position;
   building_index = left_index;
} else {
   start_col = right_position;
   building_index = right_index;
}

height = building_heights_ram[building_index];
current_x = start_col+5;  // center column of gorilla
current_y = (7 - height + 1) * 8 - 7;  // offset by 7 to not trigger hit right away

// disp_signed_int(0, 0, x_position);
// disp_signed_int(1, 0, y_position);
//play_launch();


// disp_signed_int(2, 0, frame_val);

   draw_buildings();
   draw_gorilla(left_position, right_position,
left_index, right_index);

// update the physics
update_physics();

/* if (hit) {
update_physics();
} else {
            main_state_bit1 = 1;
            main_state_bit0 = 1;
```

```
}
*/

//draw_banana(0, 1);


refresh_screen();

if (hit) {

play_explosion();
while (duration > 0);
   if (type_hit == 2) {
      // Gorilla hit ? Game Over
      main_state_bit1 = 1;
      main_state_bit0 = 1;
player_turn = !player_turn;
   } else {
amplitude = 0;

      player_turn = !player_turn;
      main_state_bit1 = 0;
      main_state_bit0 = 1;
   }
   debounced_press = 0;  // consume the button press if needed
}
else if (debounced_reset) {
   main_state_bit1 = 0;
   main_state_bit0 = 0;
   debounced_reset = 0;
player_turn = 0;
}
      }
      // Error state (11) - reset to idle
      else if (main_state_bit1 && main_state_bit0) {
P3 = 0xfc;
blank_screen(); //clear when game over

frame_update = 0;
frame_val = 0;

disp_char(1,32, 'G');
disp_char(1,39, 'A');
disp_char(1,46, 'M');
disp_char(1,53, 'E');
disp_char(1,60, ' ');
disp_char(1,67, 'O');
disp_char(1,74, 'V');
disp_char(1,81, 'E');
disp_char(1,88, 'R');

disp_char(5,22, 'P');
disp_char(5,29, 'R');
disp_char(5,36, 'E');
disp_char(5,43, 'S');
disp_char(5,50, 'S');
disp_char(5,57, ' ');
disp_char(5,64, 'L');
disp_char(5,71, 'A');
disp_char(5,78, 'U');
disp_char(5,85, 'N');
disp_char(5,92, 'C');
```

```
disp_char(5,99, 'H');

disp_char(6,43, 'B');
disp_char(6,50, 'U');
disp_char(6,57, 'T');
disp_char(6,64, 'T');
disp_char(6,71, 'O');
disp_char(6,78, 'N');
        refresh_screen();


T4CON = 0x00;

hit = 0;
type_hit = 0;

randomize_buildings();

  // get random values before moving on
  random_wind = get_random_signed(-3,3);
  random_position1 = get_random_signed(0,2);
  random_position2 = get_random_signed(0,2);
  random_building = get_random_signed(-14,-7);

if (debounced_press) {
            main_state_bit1 = 0;
            main_state_bit0 = 1;
            debounced_press = 0;
// player_turn = !player_turn;
        } else if (debounced_reset) {
            main_state_bit1 = 0;
main_state_bit0 = 0;
            debounced_reset = 0;
player_turn = 0;
}
      }
    }

}
```