

# Embedded Systems - Geometry Dash

Riju Dey, Sasha Isler, Charles Chen, Rachinta Marpaung

April 2025

## 1 Introduction

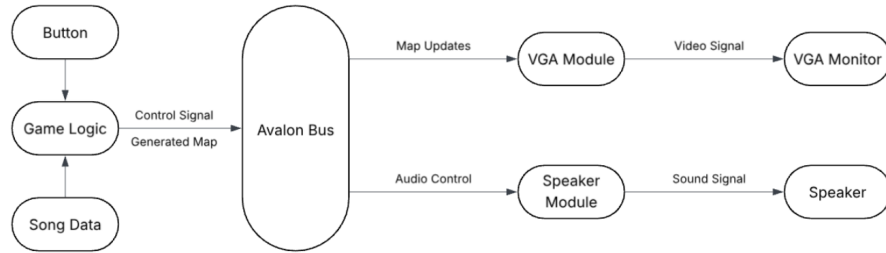


Figure 1: Block Diagram

We aim to create a version of the popular game, "Geometry Dash", on the VGA display and with the DE-1 SoC. The map is preloaded into the FPGA and its length is determined by the length of the song. Players run through the map at a set speed, avoiding obstacles through jumping. The design responds to user input through a button on the FPGA which triggers jumps/flying movement. The game logic detects failure conditions such as obstacle collisions and manages the physics of the game.

## 2 Design Details

### 2.1 Rendering

The player sprite and background will be rendered separately from the map.

#### Map Rendering

The level map will be contained in software and sent over to the hardware block by block. This gives us the flexibility to have different maps of different lengths, and they don't need to be stored entirely in hardware.

The map will be written to hardware in units of `map_block` which represent a 10x10p section of the VGA screen. Each `map_block` is one byte and contains the `object_id` of the object to be displayed in that block. This `object_id` is used to index into the `object_table` in ROM which contains the images for each object.

Each `map_block` will be stored in hardware in a 2D array of memory in RAM that we call the `display_buf`.

To determine the required length of the `display_buf`, we use:

- Screen width: 640 pixels
- Each pixel: 3 bytes (RGB)
- Each `map_block`: 10 pixels
- We double the screen width to support smooth scrolling

$$\frac{640 \times 3 \times 2}{10 \times 3} = \frac{3840}{30} = 128 \text{ blocks}$$

To determine the required height of the `display_buf`, we use:

- Screen height: 480 pixels
- Each pixel: 3 bytes (RGB)
- Each `map_block`: 10 pixels

$$\frac{480}{10} = 48 \text{ blocks}$$

Therefore, the `display_buf` will be 128x48, totaling  $128 \times 48 = 6,144$  bytes.

Listing 1: Pixel Rendering Logic

```
// Compute block_x, block_y
block_x = x / 10; // 0 to 63 (for 640px wide screen)
block_y = y / 10; // 0 to 47 (for 480px tall screen)

// Compute pixel_in_block_x, pixel_in_block_y
px = x % 10;
py = y % 10;

// Get current map_block from display_buf
map_block = display_buf[block_x][block_y];

if (map_block == 0) {
    vga_color = background_color;
} else {
    object_id = map_block;
```

```

    vga_color = object_table[object_id][py][px]; // ROM lookup
}

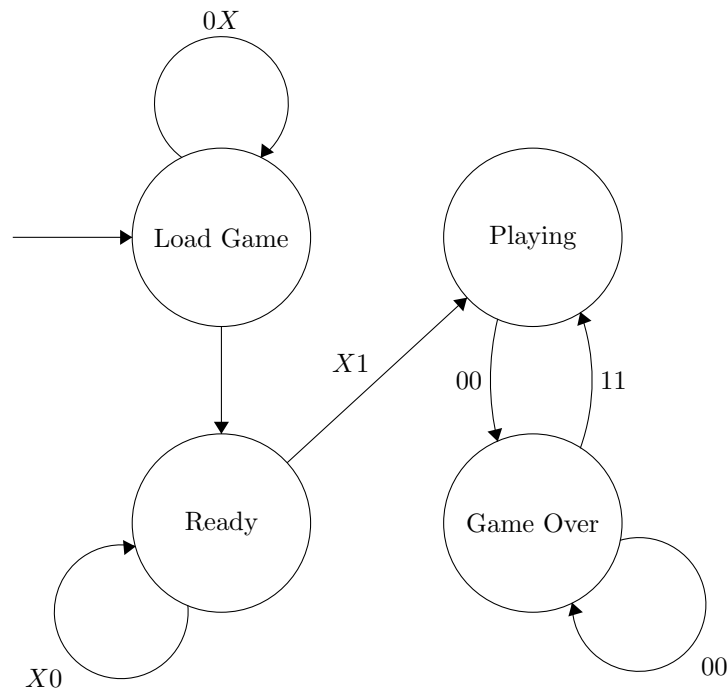
// Output to VGA

```

## Sprite and Background Rendering

The player sprite will always remain in the center of the screen and the user will control its movement on the y-axis. The rendering of the player sprite will always take priority over the rendering of the background. There will never be overlap between the player sprite and an obstacle, so there is no need to indicate priority for that scenario.

## 2.2 Control Path



Using the flags register, the software will manage a FSM with two flags: ready and set. Initially, while the map and music are being loaded, the software sets both flags to zero. When loading finishes, it sets the ready flag and transitions into the Ready state. In this state, upon user input, the game will transition to the Playing state. When the physics engine detects collisions between any obstacles in this state, both flags will be turned off and the game enters the Game Over state. If the user presses the button again, they reenter the Playing state, starting over from the beginning.

---

**Algorithm 1** FSM Using Flags Register

---

```
ready  $\leftarrow$  0
set  $\leftarrow$  0
currentState  $\leftarrow$  LOADING
while true do
  if currentState = LOADING then
    loadMapAndMusic()
    ready  $\leftarrow$  1
    currentState  $\leftarrow$  READY
  else if currentState = READY then
    if userInput() then
      set  $\leftarrow$  1
      currentState  $\leftarrow$  PLAYING
    end if
  else if currentState = PLAYING then
    runGamePhysics()
    if collisionDetected() then
      ready  $\leftarrow$  0
      set  $\leftarrow$  0
      resetX()
      currentState  $\leftarrow$  GAME_OVER
    end if
  else if currentState = GAME_OVER then
    if userInput() then
      set  $\leftarrow$  1
      currentState  $\leftarrow$  PLAYING
    end if
  end if
end while
```

---

## 2.3 Hardware Software Interface

The register map for `GameRenderEngine` is shown below:

Table 1: Register Map for GameRenderEngine

Address	Register Name	Description
0	player_y_pos	Player y position
2	player_x_pos	Player x position
4	background_r	Red background color
6	background_g	Green background color
8	background_b	Blue background color
10	map_block	A section of map containing an obstacle id
12	flags	Start, Acknowledgment
14	output	Output flags

The Engine will use 16 bit addressing. The red, green, and blue registers determine the background color. During loading, the software loads twice the screen length of the map array,  $L$ , word by word into the RAM through the map\_block register. Upon receiving and successfully storing a word, the hardware sets the ACK flag and the software begins loading the next. Once the map is fully loaded, the software sends the start signal to begin the map scroll.

For each clock cycle, the hardware reads the player\_x\_pos to determine the player sprite's x position.

## 2.4 Physics








The player sprite will implement the following attributes to simulate movement. The physics logic will be implemented in software, to enable more efficient testing.

Variable	Description
gravity	controls fall of player sprite
jump_velocity	controls jump of player sprite

The software will control scrolling and collisions. The game logic will maintain the player x position, which will be communicated to the hardware, which will be used to determine what section of the level is being rendered. The software will also use the variable  $bps$  to calculate the speed at which the block moves through the level. To track collisions, the software will look at a small range of blocks around the player square and check if there are any collisions. Hitboxes will be maintained in software.

### 3 Resource Budget

Object images are stored in preloaded ROM which we will call the `object_table`:

Components	Graphic
Player Sprite	
Regular Spike	
Short Spike	
Platform	
Square	
Razor	
Portal	

Each 1 byte `map_block` is stored in the `display_buf` which is maintained in RAM and has dimensions 128x48 and size  $128 \times 48 = 6,144$  bytes.

#### 3.1 Music

We start with a **MIDI file**, which contains a sequence of time-stamped note events. Each event tells us when to *start* or *stop* playing a note. It does not include audio, just timing and pitch data.

To generate sound, we use **oscillators** that produce waveforms like square waves. Although basic, this gives us a foundation. To go further, we adopt a **sampling synthesizer** approach using **sound fonts**.

- **Attack:** When the note is first pressed.
- **Sustain:** Looped waveform while the note is held.
- **Release:** The fading sound when the note is released.

Since MIDI files do not contain actual audio, we use a SoundFont2 (.sf2) file to provide the corresponding instrument sounds. These sound fonts contain pre-recorded waveforms organized by pitch and instrument type, enabling sample-based audio generation.

We load this file into memory and access it like a wavetable synthesizer. The MIDI file provides the **when**, and the SoundFont gives us the **what**.

To output the audio, we send the synthesized samples to a **codec**, which expects data at a fixed sampling rate—**48 kHz**. Our system must ensure that we feed the codec numbers at exactly this rate.

Software parses the MIDI file and schedules note events. Hardware performs real-time synthesis using the SoundFont data and streams audio samples to the codec.

### 3.1.1 System Implementation

The MIDI file is preloaded and parsed in software. A hardware synthesizer component loads and stores the SF2 file in memory. Then it retrieves sample data and handles interpolation as necessary. Then, it streams audio samples at a fixed sampling rate to the audio codec.

## 4 Timeline/Milestones

We plan to split up the work into several sections with milestones outlined below.

1. A hardware component that
  - (a) renders the player sprite
  - (b) renders obstacles
  - (c) renders a background
2. A music component that
  - (a) plays music upon receiving a start signal.
3. A game logic component that
  - (a) detects collisions between the player and obstacles
  - (b) implements physics and jumping mechanics
  - (c) keeps track of the game state (eg. playing, game over)
4. A user input component that
  - (a) detects button presses to trigger jumps
  - (b) debounces input signals for reliability
5. Stretch goals (if time allows)
  - (a) create a simple menu or restart feature
  - (b) add extra 'coin' object for the player to collect on the map