How Does It Feel to Be Mario?

1st Ana Camba Gomes
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
anacamba@mit.edu

2nd Fabiana Gonzalez
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
fabianag@mit.edu

***Abstract*— Our project aims to create an interactive video game inspired by the mechanics of a classic game, Super Mario Bros level 1. In this game, the player controls Mario using the buttons on the FPGA to move, jump, and interact with elements in the game world, such as coins, blocks, and enemies like Goombas. We implemented logic for enemy behavior, ensuring they appear in specific locations. Depending on the player's interaction with each enemy, Mario can defeat them or be defeated. Additionally, blocks in the background, such as bricks and question mark blocks, dynamically change their format when Mario collides with them, reflecting interactions like breaking a brick or revealing a coin. Boundaries have also been added to prevent Mario from walking through certain objects in the background, such as pipes, while still allowing him to walk on top of them. This ensures realistic and immersive navigation through the game world, maintaining the classic feel of the original gameplay. The system processes player inputs and interactions in real-time, providing a responsive and interactive experience.**

I. MARIO'S LOGIC (ANA & FABIANA)

The Mario logic simulates the behavior and movement of our Mario sprite character in our 2D game environment. It handles Mario's position updates, animations, interactions with the environment (pipes, bricks), and collision detection with enemies.

For left and right movement, the Mario module checks if the corresponding left (btn[3]) or right button (btn[2]) is active on the FPGA. If so, Mario's x-coordinate (x_array_location) is incremented or decremented accordingly. At the same time, the module toggles between two Mario sprites to create a walking animation, providing a smooth visual effect. Also, if mario collides with a solid block (pipes, bricks, question blocks) when he tries to move to the left or right his x and y location will not change.

The jumping and falling logic incorporate a maximum jump height, a jump speed, and a gravity speed. When Mario is jumping (pressing btn[1] on the FPGA), his y-coordinate increases by the jump speed until reaching the maximum height or colliding with a solid block above him. If a collision occurs, the module triggers the falling flag, causing Mario to descend at the gravity speed. When falling, Mario updates his ground position to the first solid block detected below him, ensuring realistic interactions with the environment.

Collision detection with enemies is handled through a combinational logic block that iterates over the positions of all active enemies. Mario's position is compared with the boundaries of each enemy to detect overlaps. If an overlap is found and the enemy is still alive, a collision is flagged. The results of these individual checks are aggregated into a single signal to determine whether Mario has died.

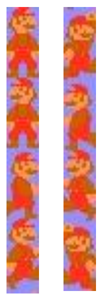


Fig. 1 Mario Sprite

If a collision with an enemy is detected, Mario transitions to a "death" state (mario_died). In this state, Mario's vertical position (y_array_location) gradually increases, simulating him being knocked off the screen until he disappears completely, signaling the end of the game.

The before_mario module calculates the image data needed to render Mario on the screen based on his center position in the game world. It takes as inputs Mario's x and y coordinates (relative to his center of mass) constantly updated by the Mario module, the current horizontal and vertical display counts, and the index of the specific image that we want to display. It also accounts for the background offset to ensure precise alignment. It computes the corresponding address data for the sprite image stored. This address is passed to the mario_sprite module, which uses it to render the correct portion of Mario's sprite (fig.1) on the display, ensuring accurate positioning and seamless interaction with the game environment.

The mario_sprite module takes the image address generated by the before_mario module, along with the current horizontal and vertical counts (hcount and vcount). It accesses BRAMs that store two files: image_mario.mem, containing concatenated sprite data for Mario, and palette_mario.mem, which defines the corresponding color palette. These files were generated using the img_to_mem.py script from Lab 5. Using the provided address, the module retrieves the pixel data and determines the appropriate color from the palette. It then outputs the pixel color to be displayed at the specified position defined by hcount and vcount, ensuring that Mario's sprite is rendered accurately on the screen

II. Display and Game Logic (Ana & Fabiana)

A. General Background (Ana & Fabiana)

The background of the world consists of a large image with dimensions of 3376 pixels (width) x 240 pixels (height). However, each displayed frame is limited to 576 pixels (width) x 240 pixels (height) (fig.2), corresponding to the visible section of the background at any given moment.

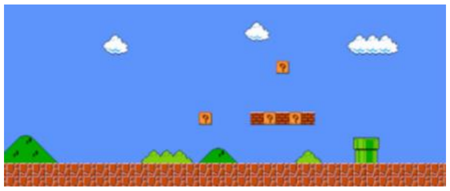


Fig. 2 Mario World 1-1

To render the background via HDMI, the project uses:

- Two main modules: before_sprite and image_sprite2.
- Three key files: guide.mem, image.mem, and palette.mem.

The entire background is composed of 36 unique images representing various elements such as bricks, clouds, pipes, and bushes (fig. 3). A Python script was used to identify and extract these unique images from the full background. These images were concatenated into a single image and synthesized into the image.mem and palette.mem files.

The background is divided into 16x16 pixel blocks, each represented by one of the 36 unique images. Another Python script was used to process the full background and map each 16x16 block to its corresponding unique image, resulting in a guide.mem file. This file contains:

- 211 blocks in width (16x16 blocks) by 15 blocks in height, totaling 3165 blocks.

The before_sprite module reads the current horizontal and vertical values and uses the guide.mem file stored in BRAM to determine which unique image should be drawn at that specific position.

It outputs a unique_image_index, which corresponds to one of the 36 unique images.

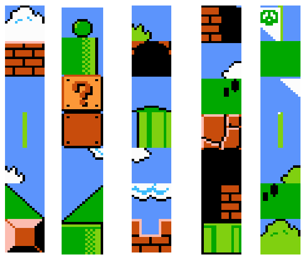


Fig. 3 unique images from background

The image_sprite2 module receives the unique_image_index from the before_sprite module, along with the current horizontal and vertical count.

This module accesses BRAMs containing the image.mem and palette.mem files, which were generated using the concatenated unique images and processed with the "img_to_mem.py" script from Lab 5. These files contain the pixel data and color palette for the unique images.

Based on the unique_image_index and information from the image.mem and palette.mem files, the module determines the appropriate pixel color and outputs it to be displayed at the specified horizontal and vertical location.

B. Collisions (Ana & Fabiana)

The display_logic module will determine which object to draw at each horizontal count and vertical count by using the before_sprite module. If the object at the given horizontal count and vertical count is a brick, a question mark or a block and a range of Mario's pixels (identified by Mario's center of mass location) overlaps with these objects, the module will output a collision signal equal to 1.

The collision_check module will include a BRAM initialized to zero (based on the size of guide.mem, as no collisions are present at the start of the game. When the display_logic detects a collision, it updates the corresponding entry in the BRAM to 1 at the specific horizontal count and vertical count of the collision. The output of the collision_check module will reflect whether a collision has occurred, returning 1 if the BRAM contains a collision entry or if the display_logic reports a new collision.

The before_sprite module uses this collision information from the collision_check to adjust the displayed sprite. If a collision is detected:

- For bricks, the sprite will be replaced with the sky.
- For question mark blocks, the sprite will be replaced with an "off" block.

If no collision is detected, the original object (brick or question mark block) is drawn as usual. This mechanism ensures the display dynamically updates based on gameplay, integrating seamlessly with the other modules responsible for background and sprite management.

C. Enemies (Ana & Fabiana)

The starting locations (x and y) of all enemies are stored in an array. The enemies_logic module functions similarly to the image_sprite module, drawing each enemy based on its initial coordinates from the array. Enemies are free to move within specified boundaries as long as their coordinates remain within the display's visible area. Each enemy is restricted to a predefined movement range, typically moving left and right within its designated area.

To enhance animation, the enemies use a sprite set consisting of three images (fig.4). When an enemy is walking, the images alternate between the first two frames, creating the illusion of a walking Goomba. The third image is reserved for the "squashed" version of the Goomba after a collision. To render enemies on the screen, an alpha channel is utilized. This method enables enemy pixels to overlay background pixels seamlessly. If a pixel from the enemy sprite matches a specific alpha value (#9290FF), the background will be drawn at that position, creating a transparency effect. Otherwise, the pixel from the enemy sprite is rendered as usual.

The module continuously updates each Goomba's location, allowing it to be compared with Mario's position in real-time. If Mario's feet touch the Goomba's top y-coordinate and mario is within the ranges of the Goomba's x coordinate, the Goomba will be replaced by its "squashed" version. After being squashed, the Goomba disappears, leaving only the background displayed at that location, and it will not reappear. This dynamic interaction ensures realistic gameplay and adds a satisfying level of interactivity.

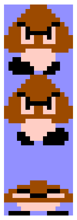


Fig. 4 Image sprite for Goombas

D. Coins (Ana & Fabiana)

The starting locations (x and y) of all coins are stored in an array. The coin's logic operates similarly to the enemies logic, drawing each coin based on its initial coordinates from the array. When Mario collides with a brick located at the same position as a coin's initial location, a flag is activated, triggering the coin to be drawn. The coin sprite consists of 4 unique 16x16 pixel images (fig.5). Each image is displayed in sequence, creating an animation effect that simulates the coin rotating. This enhances the visual dynamics of the game.

The before_coin module calculates the address data where the coin should be drawn on the screen based on its current position. Once triggered, the coin module updates the coin's y-coordinate, making it move upward and then downward within a defined range, simulating a bouncing effect. This movement continues until the animation is complete. At that point, the coin disappears, and the background is redrawn in its place.

The coin_sprite module takes the calculated address and retrieves the specific image frame to render it on the screen. Like the enemies, the coins use an alpha channel for seamless integration with the background. Pixels outside the sprite boundaries or with a specific alpha value (purple, #9290FF) will display the background instead. This ensures that the coins appear and interact seamlessly with the game environment while maintaining a clean and visually appealing design.

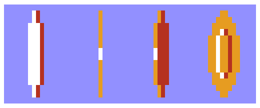


Fig. 5 Image sprite for Coins

E. Game Logic (Ana & Fabiana)

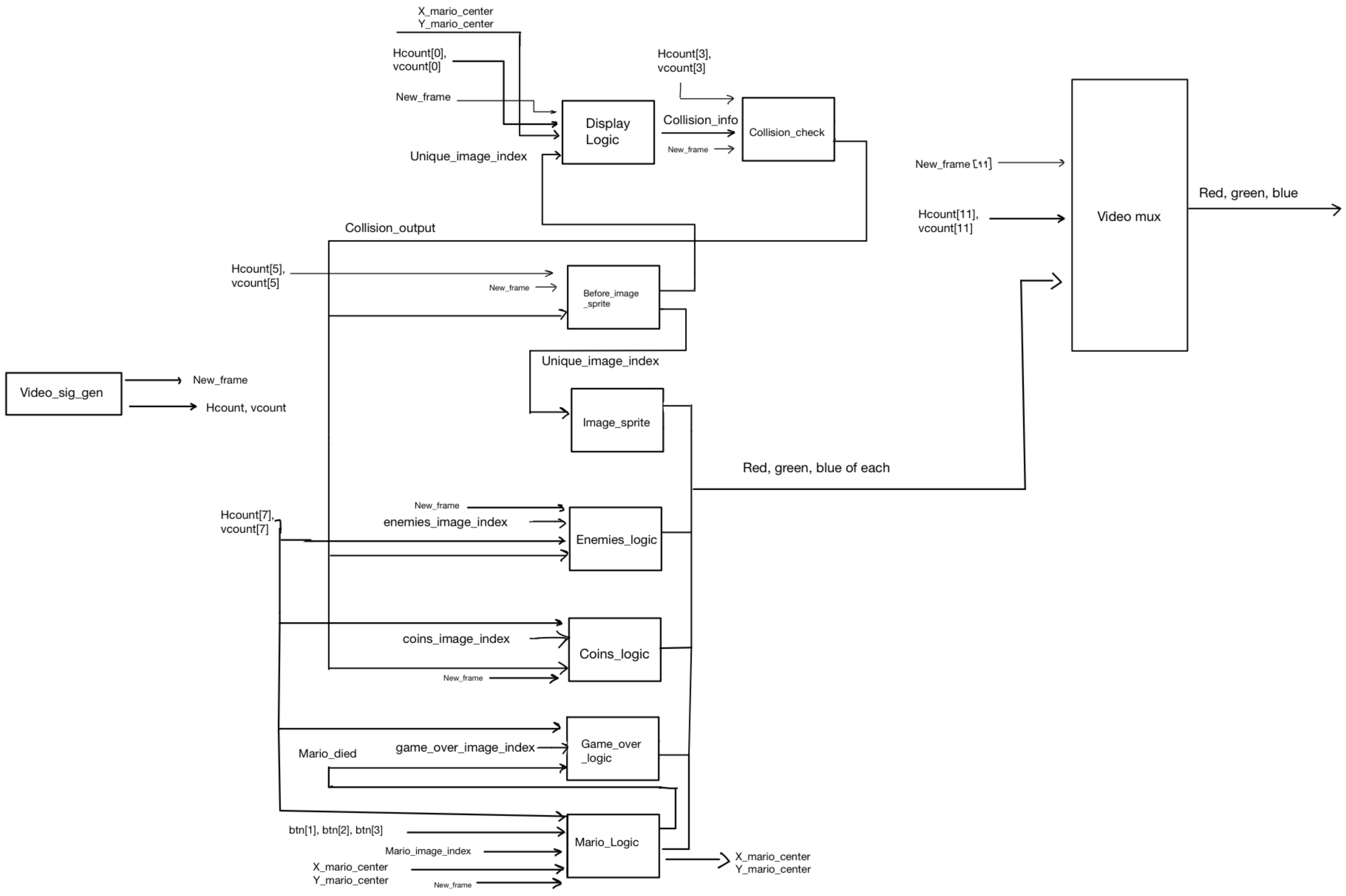
The Game Logic is found in different modules of our project. When the player dies due to a Goomba collision the game over state will begin. These actions are handled by the Mario module and the game over module. Lastly, the reset of the game happens in each individual module when the system reset flag is triggered by btn[0].

The before_game_over module calculates the image data needed to render the game over sprite (fig.6) on the screen based on the center position of the display. It takes as inputs the display's x and y coordinates (relative to the center of our 576 by 240 display), the current horizontal and vertical display counts, and the index of the specific image that we want to display (in this case is only one image of 73x9 pixels). It also accounts for the background offset to ensure precise alignment. It computes the corresponding memory address for the sprite image stored. This address is passed to the game_over_sprite module, which uses it to render the correct portion of the game over sprite on the display.

The game_over_sprite module takes the image address generated by the before_game_over module, along with the current horizontal and vertical counts (hcount and vcount). It accesses BRAMs that store two files: image_gameover.mem, containing the sprite corresponding to the game over image, and palette_gameover.mem, which defines the corresponding color palette. These files were generated using the img_to_mem.py script from Lab 5. Using the provided address, the module retrieves the pixel data and determines the appropriate color from the palette. It then outputs the pixel color to be displayed at the specified position defined by horizontal count and vertical count.

Fig. 6 Image sprite for Game Over

III. Block Diagram (Ana & Fabiana)



IV. Evaluation

We utilized a total of 11 BRAMs in our design. Of these, 10 were allocated to store the palette and image memory for different game elements, including Mario, enemies, background, coins, and game-over sprites. The remaining BRAM was dedicated to collision detection. Although we instantiated 11 BRAMs in our design, the FPGA synthesized them into 12.5 RAMB36 blocks, utilizing 16% of the 75 available blocks on the FPGA.

We created the BRAMs incrementally, starting with the background BRAM and adding more as we integrated Mario, enemies, and coins into the game. This approach proved to be an effective decision since having separate BRAMs for each element allowed simultaneous access and rendering of game components, ensuring efficient drawing of all elements independently.

Another key decision was to store only the unique images for each game element in the BRAMs. This avoided redundancy and significantly reduced memory usage compared to storing repeated images within the sprites. This optimization helped conserve memory and maintain an efficient design.

Our design implemented an 11-stage pipeline to ensure smooth operation. Each pipeline stage was carefully configured with the minimum number of cycles required for the modules to complete their tasks. This ensured that data was delivered on time while allowing additional cycles for modules that needed more processing time.

The design successfully met the timing requirements. In the Post-Routing Timing analysis, we achieved a Worst Negative Slack (WNS) of 0.510 ns, a positive value confirming that the design operated within the required timing constraints. The FPGA executed the design in Phase 12, demonstrating that the implementation was relatively straightforward and efficient for the hardware.

V. Results

For the results, we initially planned to use a camera to represent Mario, were instead of drawing Mario's sprite, he would be depicted through the video feed from the camera. However, we decided to change our plans for two main reasons: the camera logic was very similar to what we implemented in Lab 5, and having Mario represented by a video would give him too much freedom on the screen, making it difficult to impose boundaries and structure.

Instead, we chose to draw Mario as a sprite with defined movements and interactions, incorporating boundaries for solid objects like pipes and bricks. This allowed us to challenge ourselves by implementing complex logic for Mario’s interactions, such as determining whether Mario defeats an enemy by jumping on it or is defeated by touching the enemy.

We successfully achieved our initial goals of implementing the logic for enemies and coins and went beyond by adding Mario’s sprite, which was not originally planned. We also accomplished the goal of dynamically changing the background and objects based on Mario's interactions and position. Additionally, we created the game logic for losing, where a "Game Over" screen is displayed when Mario is defeated, fulfilling the expected gameplay mechanics.

If we had more time, we would have loved to include a scoring system that tracks and draws the number of how many coins Mario collects during the game, adding an extra layer of challenge and engagement.

Acknowledgment

We would like to thank Professor Joe Steinmeyer, Stephen Kandeh, Kiran Vuksanaj and Kailas Kahler for their support and guidance throughout this semester and with the project.

References

[1] “Super Mario Bros. - World 1-1.” *The Spriters Resource,* www.spriters-resource.com/nes/supermariobros/sheet/20592/. Accessed 26 Nov. 2024.