2D Platformer Video Game on Spartan-7 FPGA

1. Introduction:

For our final project, we designed and implemented a Super Mario Bros game with multiple levels, using the Xilinx Spartan-7 FPGA and the MicroBlaze processor. Additionally, we implemented the essential components using SystemVerilog, such as the keyboard and video display. Our goal was to demonstrate full control over the character's motion so that it reaches the end line of a platforming level. We were able to design an accurate dynamic collision detection using color mapping for multiple levels, add sound effects, display advanced graphics, manipulate text graphics, and control the character's status (win/death).

To help build the foundation of our project, we used some code from previous labs, such as the ball, color mapper, and keyboard input from lab 6 for the character and background, lab 7 to help with the text graphics, and the VGA modules from both labs for the screen display.

The game is mainly controlled through the keyboard. "A" and "D" correspond to left and right motion, respectively, and "Space" is used for the jump.

2. Description of the final project

Color Detection:

This feature is probably the most important in our project as it is considered part of the general functionality of the design and what makes our level design precise and accurate. First, when we design a level, we take the background and color over it the collisions that we want Mario to encounter during the level. For example, as can be seen in Figure 1, this is what the color map for our second level looks like. The way we detected these colors is by creating two dual-port ROMs for each level (4 in total for two levels). Each port is constantly detecting above, under, to the right, and to Mario's left at all times using an address that detects a fixed point based on Mario's position instead of normally depending on DrawX and DrawY. Based on the color Mario detects, his position and motion are adjusted accordingly. So it's like there are four hidden backgrounds that are not displayed but are working at the same time.

For example, the background in blue is used to determine if Mario is the air so that we can turn on gravity. The purple is used for general floors and walls. The red area is where Mario's status should be changed to death since there is lava or spikes there. Lastly, the green is used for two purposes: the win status and the lucky block, but they are distinguished depending on where Mario detects the color (above or under). The reason why we didn't add more colors is that we used the same GitHub picture conversion tool for the background here so we could get a matching resolution. This tool uses a power of 2 number of colors, so we wanted to use only four colors so it wouldn't use more than we need, and the detection gets confusing.

This method of color detection made our level execution very smooth but came at the price of exhausting our BRAM usage. However, putting this restriction aside or simply adding additional memory to the FPGA, we can get very creative in our level design since we have the ability to implement any level and any design with the addition of as many characters or interactive objects as we want.

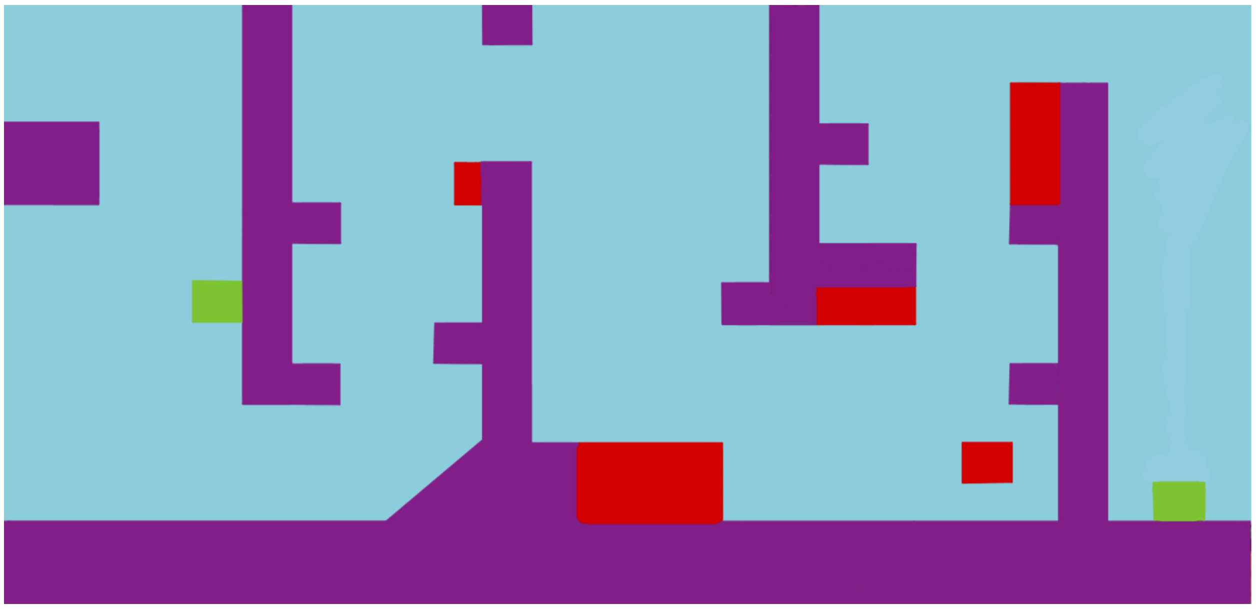

Figure 1: Color map of level 2 background used for collision detection

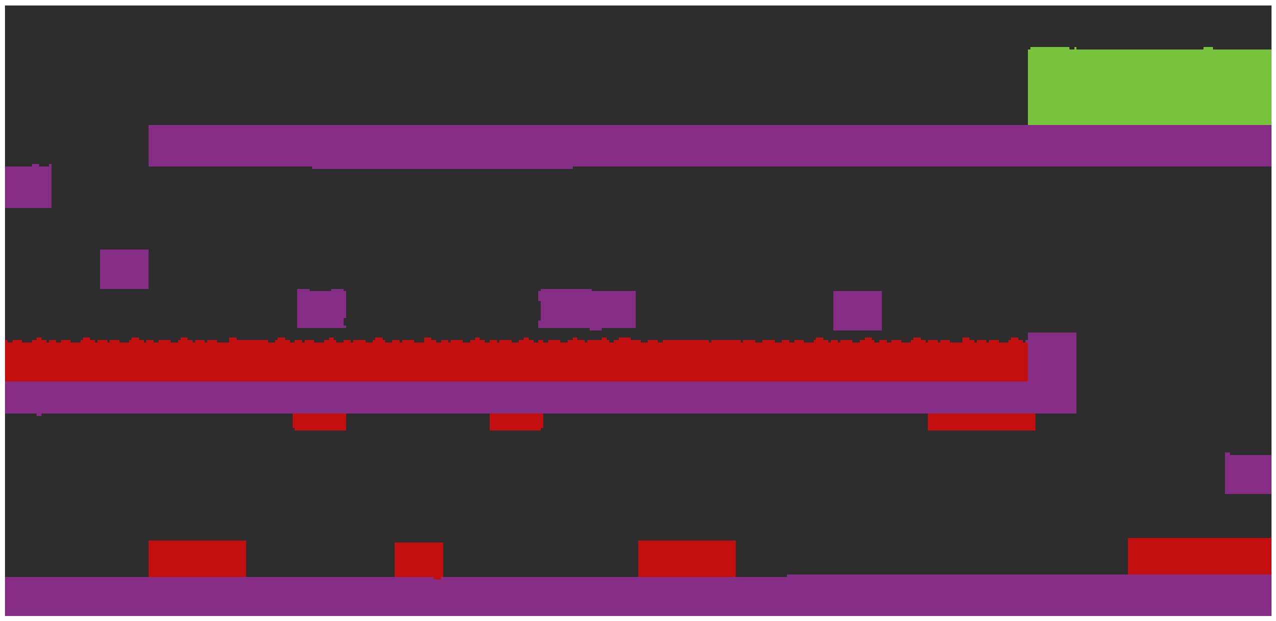

Figure 2: Color map of level 1 background used for collision detection

Display Control:

Usually, the control of transition or states in any project is done by using FSM. However, we didn't use it in a direct approach. We had control signals that controlled different statuses of the game, but we didn't create an FSM unit or states that we chose from. The first transition encountered when playing is the start screen in Figure 3, which is a simple welcoming screen that says press enter, but it was very compressed (200*100) due to lack of memory. By simply pressing enter on the keyboard, you proceed to the first level in Figure 5. Then, when Mario enters a red area, as elaborated before or as can be seen in Figure 1, the screen turns full red, indicating the character's death. Later, when Mario reaches the yellow pipe in level 1, a signal named next_screen is turned on, and the screen switches to the next level, as shown in Figure 6. Lastly, when Mario reaches the end line (when it detects green from under), the screen turns full green, indicating the player's arrival at the end line.

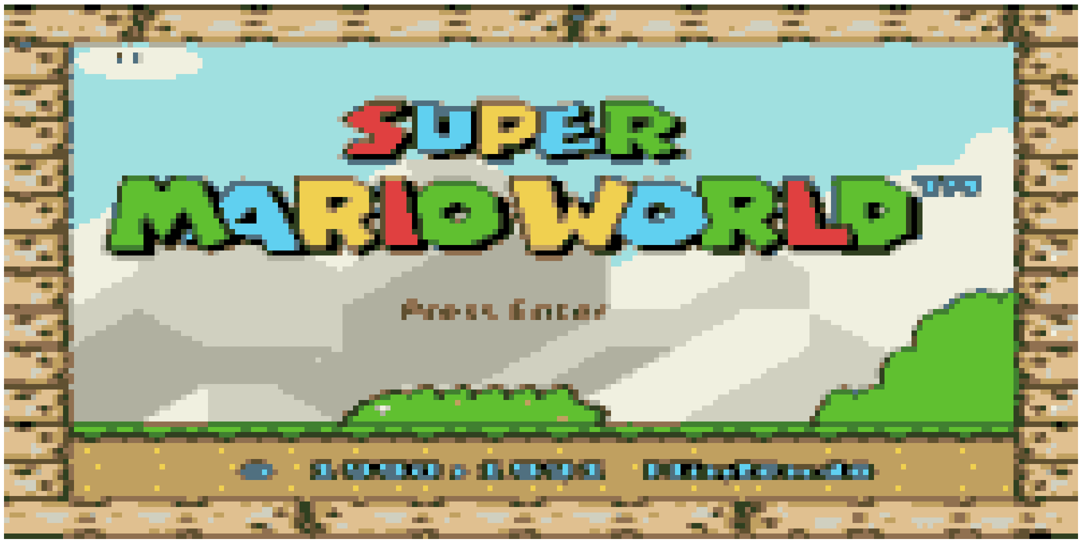

Figure 3: Final result after compressing the background of the welcome screen

Screen Compression:

Since we implemented two levels, we needed a lot of memory space. Especially because the color detection algorithm takes a lot of space due to each level using two Dual Port ROMs, meaning that the color detection alone uses 4 BRAMs, each with 130000 read depth, worth of memory per level. Because of that, we needed to be as efficient as possible in order to manage fitting two levels with the limited Spartan-7 sources. What ended up working the best for us was that instead of using the full 640*480 size for the level backgrounds, we thought about compressing it to much smaller dimensions and then stretching it back to fit into the whole screen. The drawback of this decision was that it had a bit of a blurry resolution due to compression and stretching back; however, we thought that it added to the authenticity of the game.

After trying various resolutions, we decided that 520*250 is the best resolution in terms of being as efficient as possible while also providing good background resolution. Using 520*250 means that we need to configure the read depth for the ROM to be 130000, while on the other hand, using 640*480 means that we need to configure the read depth for the ROM to be 307200, almost three times as much. We used the Image_to_COE GitHub to generate COE files and corresponding palettes for our backgrounds. In that tool, we can specify the resolution of the image, so we set it to 520 horizontal by 250 vertical and then stretch the image to the whole screen. Figure 4 shows the final image produced for level 1.

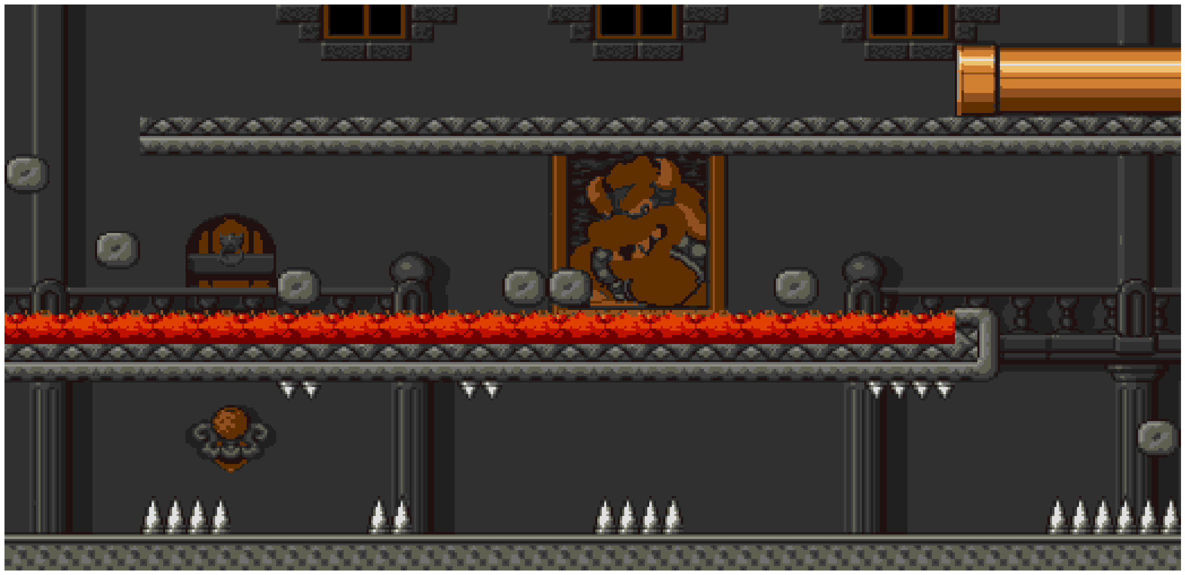
Figure 4: Final result after compressing the background of level 1 and stretching it back

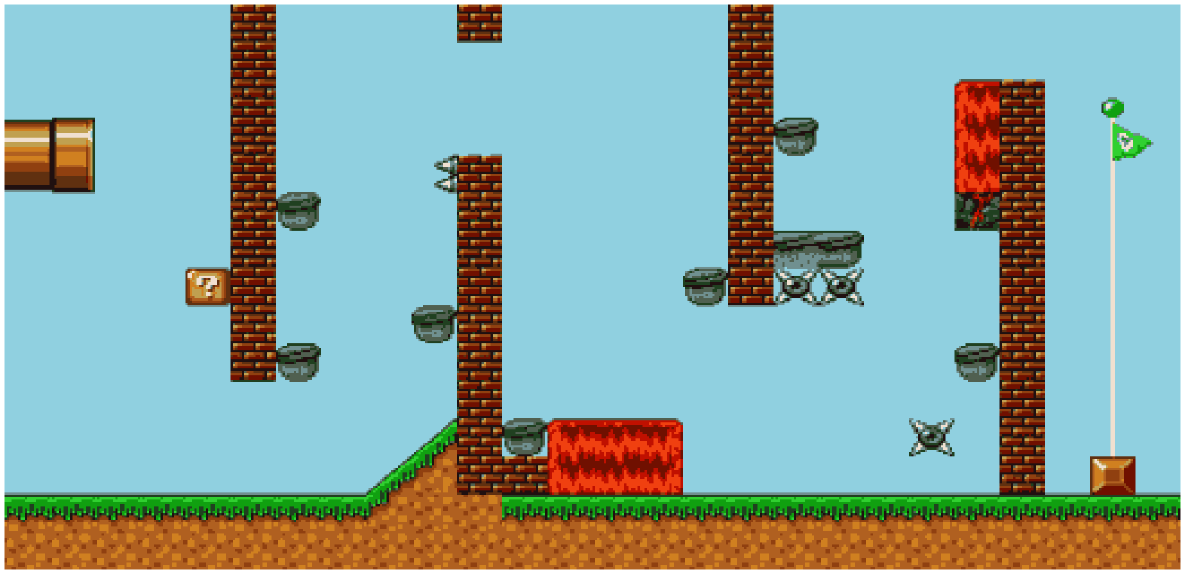
Figure 5: Final result after compressing the background of level 2 and stretching it back

Score Counter:

We implemented a counter that is displayed on the top left of the screen to keep track of Mario's score. The score increments when Mario goes to the next level (through the yellow pipe shown in Figure 4) or when Mario hits the bottom of the lucky block in level 2. Since we were limited on memory, we couldn't import the whole IP we created for lab 7, so we only used the font_rom.sv module with some modifications to draw the score. The way we implemented the score counter first was to check if DrawX and DrawY were inside a rectangle, and based on where they were on that rectangle, we drew the corresponding character. So the leftmost position of this rectangle draws S, the position right after that draws C, and so on. Then, after printing the string "SCORE:" we print whatever is on the 1's digit counter, 10's digit counter, and 100's digit counter.

Sprite Display (Mario):

Just like the backgrounds, Mario itself had its own ROM, which stored its colors. Just like the ball from lab 6, Mario is just a box that is 36 pixels wide and 48 pixels tall, with its center as a parameter that is updated depending on Mario's motion and collisions. Whenever Mario's first pixel is displayed on the screen (which is an offset calculated based on the center of Mario), a counter starts counting until 36, which is used as an address for the ROM, as shown in Figure 4. Another counter is incremented whenever the first counter reaches 36 until this counter gets to 48 to complete Mario's box. One touch that we added is that when Mario's box is not displaying Mario but instead its background, we made that pixel to display the level's background, not Mario's. So, in the game, Mario doesn't look like a box because only the important pixels are displayed.

Sound:

We thought of adding this feature as a bonus after we finished the whole design of the project. We created a module that generates a square wave that outputs a beeping sound. We added this effect when the character dies in the game to enhance the player's experience. In order to hear the sound, a headphone or a speaker should be attached to the FPGA.

3. Block Diagram:

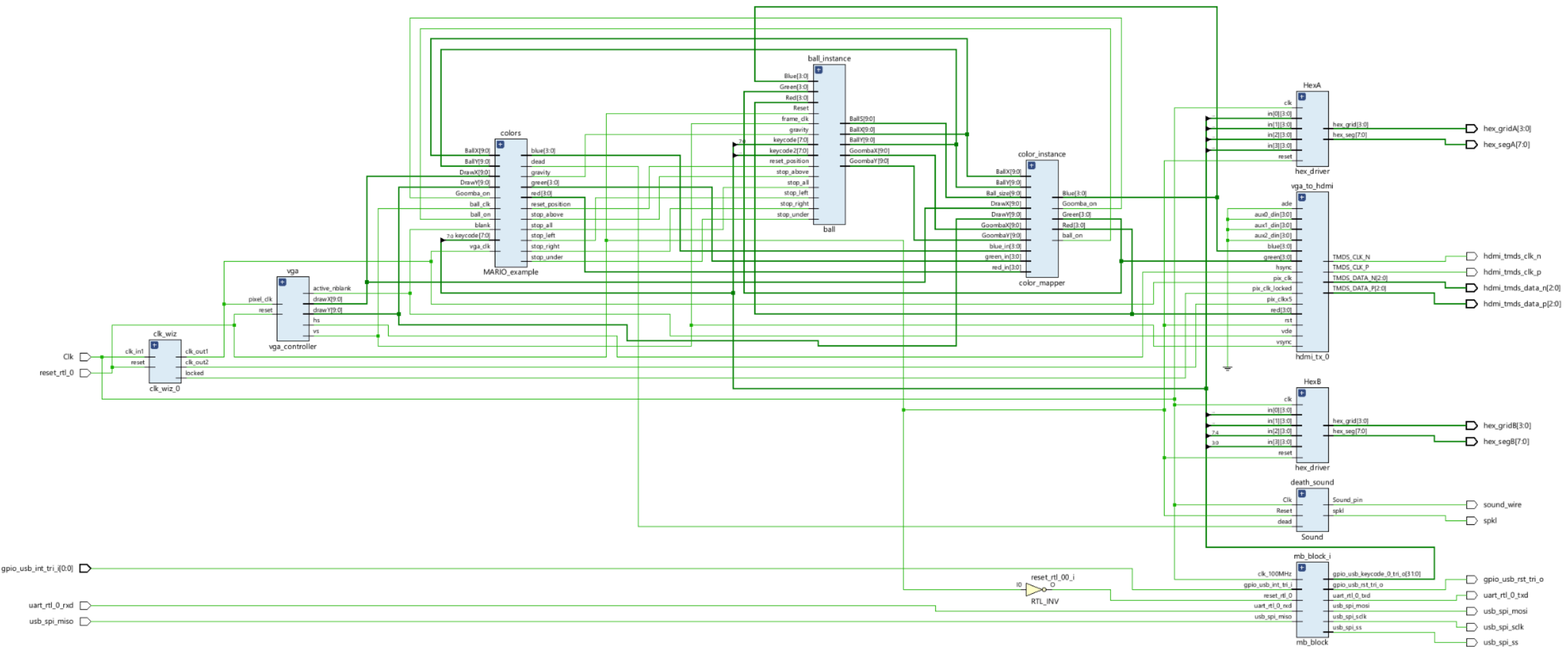

Figure 6: Block diagram of all the project's modules

4. Performance:

Look Up Tables (LUT)	4449
DSP	16
Memory (BRAM)	75
Flip-Flop	2777
Latches	0
Frequency	132.40MHz
Static Power	79mW
Dynamic Power	461mW
Total Power	540mW

Table 1: Design Statistics Table for Final Project

5. Conclusion:

In conclusion, our project of creating a Super Mario Bros minigame on the Xilinx Spartan-7 FPGA has been a comprehensive and enlightening experience. We successfully achieved our objective of constructing a fully functional game with advanced features such as dynamic collision detection, multiple-level designs, sound effects, and enhanced graphics using SystemVerilog. Our innovative approach to color detection and memory management allowed us to overcome hardware limitations, showcasing our ability to optimize resources effectively.

Throughout the project, we faced challenges, including managing memory constraints and implementing efficient collision detection algorithms. However, these obstacles provided valuable learning experiences in FPGA programming, game design, and the practical application of embedded system principles.

Our project not only met the basic functionality we set in the proposal, such as basic control over Mario's movement but also included several features that made it more impressive. Some of these features are color detection, implementation of sounds, and a score counter displayed on the screen.

Potential improvements as we proceed include improving the game's visual elements, introducing more complex levels, and implementing additional features such as high-score tracking and multiplayer capabilities. With further development, this minigame could become an even more engaging application of FPGA technology.