FPGA Pixel Processing Unit Documentation

-James Guo

1.Introduction

The Pixel Processing Unit (PPU) was originally a microprocessor in the Nintendo Entertainment System that generates video signals from graphic data stored in memory (game cartridges). It was quite advanced at its time, being able to support multiple sprites, moving backgrounds with relatively low memory usage. This project aims to replicate the functions of the PPU using an FPGA board: reading/displaying sprites stored in memory, supporting a background and having simple motion and physics. All of the functions are rendered via hardware/logic within the FPGA board.

2.Module Overview

As shown below, Figure 1 is the top level view of the PPU. The PPU is connected to the framebuffer/video driver provided in the 271 final project lab files and the board is physically connected to a VGA display. The details of each module will be discussed in sections below.

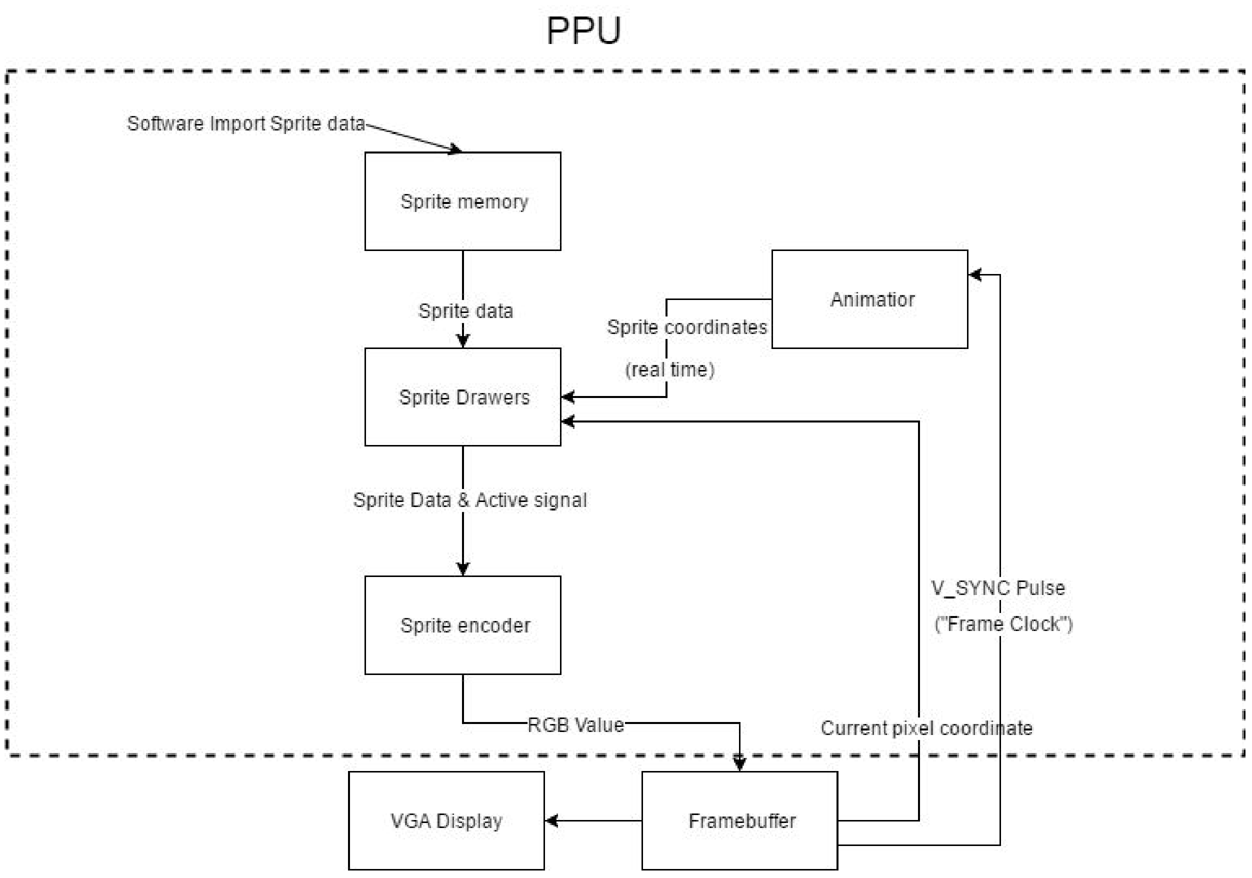


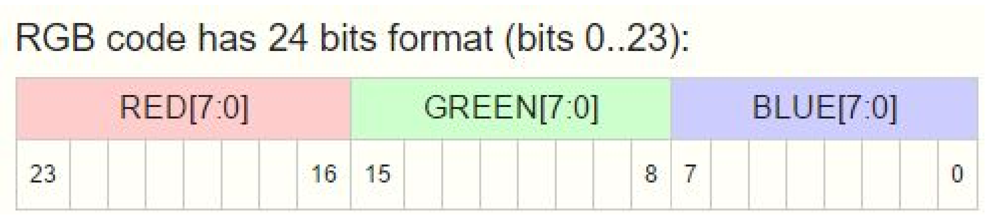
Figure 1. The top level view of the PPU.

3. Submodules

I.Framebuffer

The frame buffer is provided from the EE 271 final project resource archive. It directly interfaces with the board VGA port and produces the output according to the given RGB value at the time. The module takes in RGB values and gives out the x and y coordinate of the pixel it is currently drawing, with a 680*480 pixel resolution. Contrary to intuition as we perceive screens frame by frame, the framebuffer actually draws only one pixel at a time, but at a very high speed so that we would not notice.

The PPU would have to give the framebuffer the correct RGB value at the given coordinate to correctly display the desired data. RGB values are simply 24 bits of data that determines the amount of red, green and blue of one pixel that synthesizes it into one single color, as shown in the figure below.


RGB code has 24 bits format (bits 0..23):

RED[7:0]								GREEN[7:0]								BLUE[7:0]							
23							16	15							8	7							0

Figure 2. The RGB code format.

II.Sprite memory and sprite format

By definition, a sprite is a piece of computer graphic that can be moved on screen and manipulated as a single entity. An example would be the pokeball shown below. It is within a 16*16 pixel matrix. Each pixel has one RGB value and there are 16*16 = 256 pixels, thus a 24 bit array that has 256 elements would be required to store a sprite shown below. One thing to note is that one color value that is not used should be chosen as a “transparent” color to make it possible for different layers of sprites and background overlap. In the figure shown below, the grey and cyan pixels are actually “transparent”. In the project, the transparent value used is pure white, 0xFFFFFF. For actually displaying white, 0xFFFFFE was used.

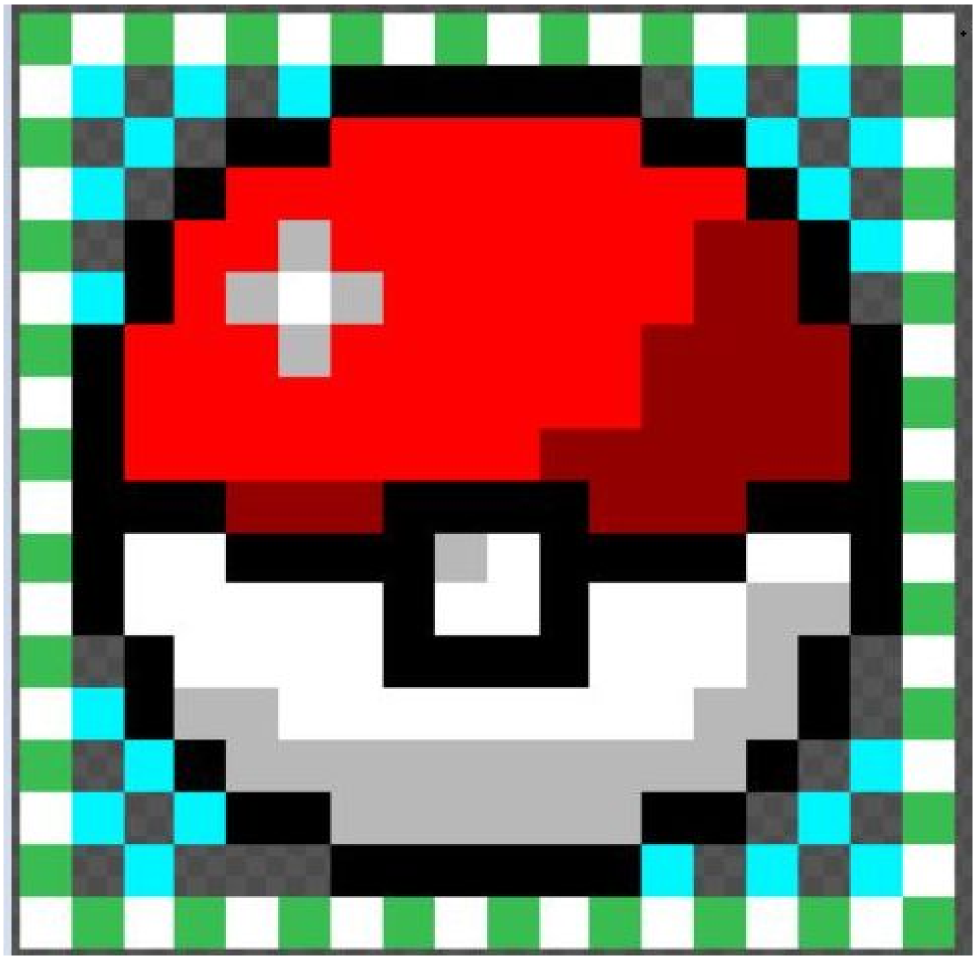


Figure 3. A sprite that is a pokeball.

An orientation for the RGB data stored is also important. In this project, the start of the data would be the top left pixel, and then going to the right and downwards, so the order of the data aligns with the actual picture. The generation of the actual data in HEX RGB format was done via python and the python image class. Clever programs can be written within this class to do image conversions for the project.

With the sprite data loaded into the memory, the other submodules will be able to load and manipulate the individual sprites as a single entity.

III. Sprite drawer

The purpose of the sprite drawer is to produce an RGB signal when the framebuffer is drawing part of the sprite on the screen. This requires the data of the sprite itself, imported from the sprite memory module mentioned above and the coordinates of the origin of the sprite, given by the animator module. One thing to note that the animator module will give changing coordinates over time, thus the sprite drawer is required to have the ability to constantly. Thus a Finite State Machine is required.

The FSM within the sprite drawer would be able to determine whether the sprite loaded is currently being drawn by the framebuffer and return the corresponding RGB values and a “sprite active” signal to the sprite encoder which then encodes all sprite values and backgrounds, resolving collisions and then sending the signal to the framebuffer.

The FSM should be able to determine the corresponding pixel of the sprite array given and find the index of the array that holds the data. The equation needed to decode the needed x and y coordinate of the sprite and to find the corresponding index in the data array is very straightforward: Index = x coordinate + y coordinate * width of matrix.

IV.Animator

The purpose of the animator is to keep track of the coordinates of individual sprites and animate them accordingly via an FSM powered by a “frame clock” that is much slower than the clock driving the framebuffer. Since the framebuffer operates using the CLOCK_50 on the De1-Soc board, a much slower “frame clock” is required. The framebuffer has v-sync module that prevents display cutting off due to timing issues. This means that the framebuffer provides a vsync pulse that changes every time a new frame is drawn. However, in the case of the project, the vsync pulse is too fast for slower, recognizable animations, so a slow clock CLOCK_50 divided 15 times was used to update the origin coordinates of the sprites.

The nature of FSMs allows us to slice up the time into discrete pieces, thus simple physics animations are possible to be rendered via the animator. In this project, a simple projectile motion was implemented by constantly updating the x and y coordinates and velocity over time (the “frame clock” powering the animation FSM).

V.Sprite Encoder

The purpose of the sprite encoder is to encode all the data from the various sprites and put them in the desired order and then output the overall RGB value of the current pixel to the framebuffer.

Another main issue that the sprite encode solves is the collision/layers of the sprites. When sprites collide and share pixels, the encoder will be able to determine the layers/priority of the sprites and let the front sprite take over the pixels, resolving the conflict.

In this project, the sprite encoder takes both the RGB values and “active” values from multiple sprite drawers. The RGB value given by the sprite drawers will only be valid both when the active signal is high and the RGB value is not the background color.

The encoder was done by implementing a complex FSM with nested “if-else” statements, checking the active signals of the sprites, from higher priority to lower and then checking if the RGB value received was the background color (in this project, 0xFFFFFF), and then outputs the corresponding desired RGB value of the pixel to the framebuffer, whether it is from a sprite drawer or the background.

4.Potential improvements

Though the core functions are implemented, there is room for improvement of the project, to make it more functional and straightforward to use.

I.External modifiable ROM for sprite data

The sprite data memory is implemented via allocating memory from the FPGA and thus reading/storing the data requires a recompile every time the data is modified. The De1-SoC board supports SD card reading, thus implementing a system that allows the PPU to read data directly from external memory would make using and programming for the PPU more straightforward and also save the precious FPGA logic elements for more functions.

II.”Software” Support

Though is is entirely possible to create an interactive game just using verilog logic, it is highly undesirable to do so. In this project, the PPU had a projectile motion demo using the FSM, but if more functions were to be implemented, the FSM would have to be more complicated and more states would need to be introduced. Thus a “software” system powered by a CPU that keeps track of the state of the “game” and the sprites would be ideal both for simplifying the design and for further applications of the PPU.

III.Peripheral Devices

The De1-SoC FPGA only contains several buttons and switches. These are not the best devices for an interactive game/system, thus implementing a simple controller or other peripheral devices would be ideal to improve the user experience and for wider applications.

5. Conclusion

In conclusion, the FPGA PPU project successfully implemented 24-bit RGB sprites that supports multiple layers, a background and simple physics to animate the sprites. Its function is comparable to the actual PPU unit in the NES system, though being more simple due to the nature of this project being a demonstration for the functions instead of an applied ASIC circuit that was mass produced. Though the core functions are implemented, there is room for improvement of the project, to make it closer to a modern design.