FlaPGA Mario*

Implementing a simple video game on Basys 3 FPGA board

Liu Haohua

June, 2018

Contents

1 Introduction **2**
1.1 Background 2
1.2 Drawing the blueprint 2
1.3 System architecture 3

2 Fundamentals **4**
2.1 VGA Module 4
2.1.1 Basys 3 VGA port specs 4
2.1.2 VGA Timing 4
2.2 ROMs 6
2.2.1 Accessing the pixels 6
2.2.2 Writing data to the board 6
2.3 RAMs 6

3 Graphics Engine **7**
3.1 Displaying images 7
3.2 Sprites 7
3.3 Background Engine 7
3.4 Object Engine 8
3.5 Layers blending 8

4 Audio Engine **8**
4.1 Sine wave generator 8
4.2 Music data 10

5 Game Logic **11**
5.1 Mario 11
5.2 Pipes and coins 11
5.2.1 Pipes and coin generation 11
5.3 Collision detection 12
5.4 Scrolling 12
5.4.1 Split scrolling 12
5.4.2 Parallax scrolling effect 13

*The full code can be obtained at `https://github.com/howardlau1999/flapga-mario`

5.5 Data writing arrangement . . . 13
5.6 Game status . . . 14

6 Conclusion **14**
6.1 Screenshots . . . 14
6.2 My feelings . . . 15
6.3 Possible improvements . . . 15

7 References **16**

8 Acknowledgement **16**

9 Appendix **16**
9.1 Resources convertion . . . 16
9.1.1 Image data convertion . . . 16
9.1.2 Music data convertion . . . 16

1 Introduction

1.1 Background

Early in my childhood had I the first contact with video game consoles and I have been long fascinated by the virtual world that games create. I taught myself how to crack a game ROM to extract and modify its resources with the help of Internet. This obsession never extinguishes. As soon as I got a Basys 3 FPGA board, I immediately noticed that there was a VGA port, which left much room to possibilities.

Naturally, I came up with the idea of building a game console on this tiny board, that is, a FC on an FPGA. Nevertheless, after some simple search on the Internet, I realized that every game console needed a CPU to compute everything. Implementing a CPU is way beyond my abilities for the time being. Although there is a variety of CPU IP cores such as 6502 series used by FC, I think they deprive of much fun. What's worse, writing assembly codes is also way beyond my abilities.

1.2 Drawing the blueprint

Taking all the factors into consideration, I finally decided to implement a video game on Basys 3 FPGA board from scratch, without a CPU implementation. This put some restrictions on this video game:

(1) Due to the lack of a CPU, the game logic must not be complicated.

(2) Yet the video game must be fun enough, otherwise no one is willing to play it.

In the very first place, I regarded the classical PONG game as a good choice. But it is so simple that several people have already implemented it. On top of that, the dull white squares seemed unattractive at all. So I added some expectations of the game:

(1) The graphics should be colorful, better to have animations.

(2) It'd better not yet been implemented by anyone (which leaves room for me to explore).

After some other search on the Internet, I locked my eyes on a game named *Yoshi's Nightmare*. This game satisfies all the requirements listed above, which inspired me to a large extent. Based on this game, I initially thought of a game where Mario jumps between some floating platforms moving upwards in order not to fall to the ground. But during my exploration of the FPGA board, I found this a little complicated to implement so I finally gave up this idea.

Ultimately, I made the decision to build a game similar to *Flappy Bird*, but with the bird replaced by flying Mario – FlapPGA Mario.

Now it's time to get moving!

1.3 System architecture

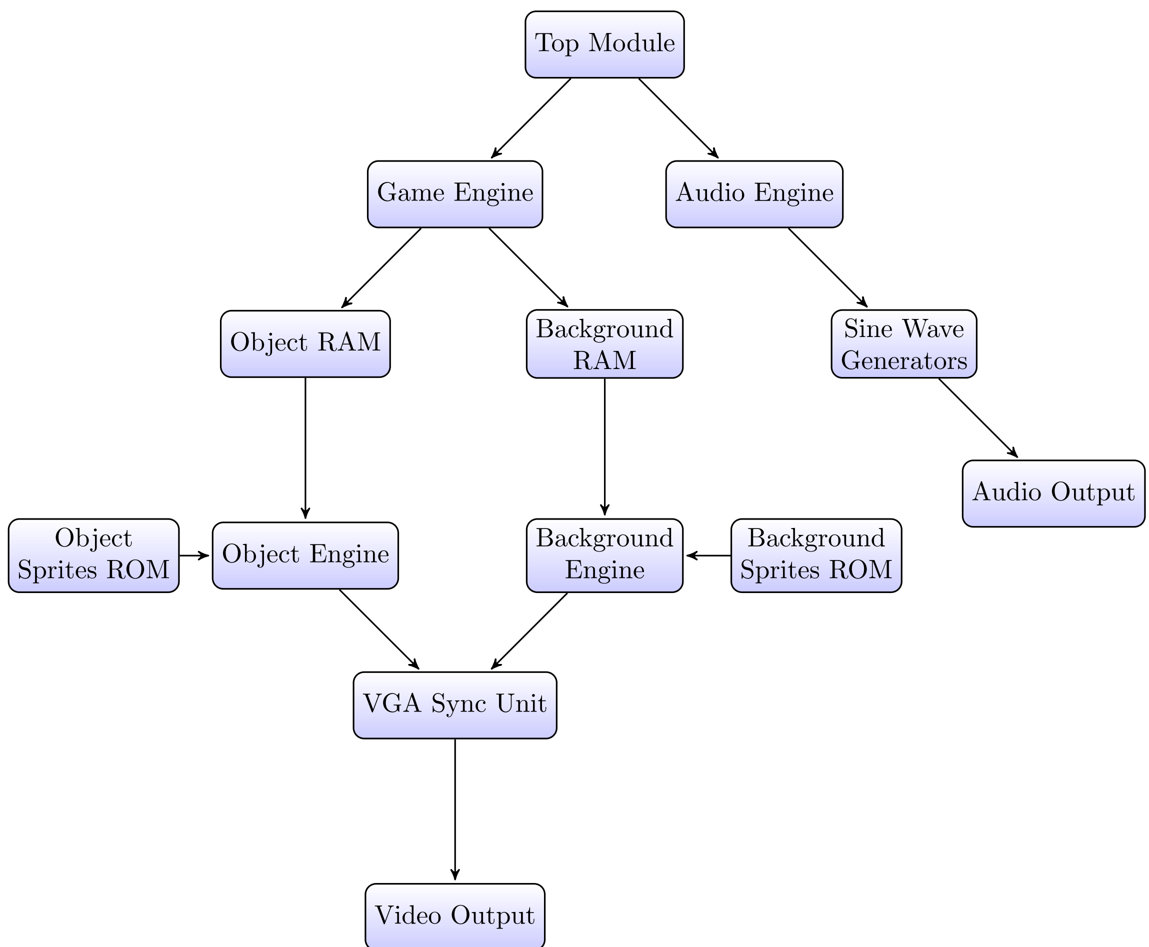


Figure 1: The whole game's hardware architecture

2 Fundamentals

2.1 VGA Module

The VGA module is the most fundamental component in the video game, which enables graphics to be displayed on a screen. Firstly, let's take a look at the tech specs of the board's VGA port.

2.1.1 Basys 3 VGA port specs

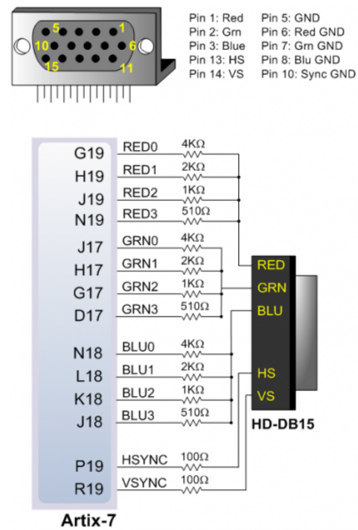


Figure 2: Basys 3 VGA port

According to the manual, this VGA port supports 12-bit depth color, 4-bit each for red, green and blue, which means we can display $16^3 = 4096$ different colors in total.

A video controller circuit must be created in the FPGA to drive the sync and color signals with the correct timing in order to produce a working display system.

2.1.2 VGA Timing

This video game runs under the resolution 640 by 480 pixels. Modern LCD monitors use an array of switches that can impose a voltage across a small amount of liquid crystal, thereby changing light permittivity through the crystal on a pixel-by-pixel basis. LCD displays have evolved to use the same signal timings as CRT displays in a scanning way.

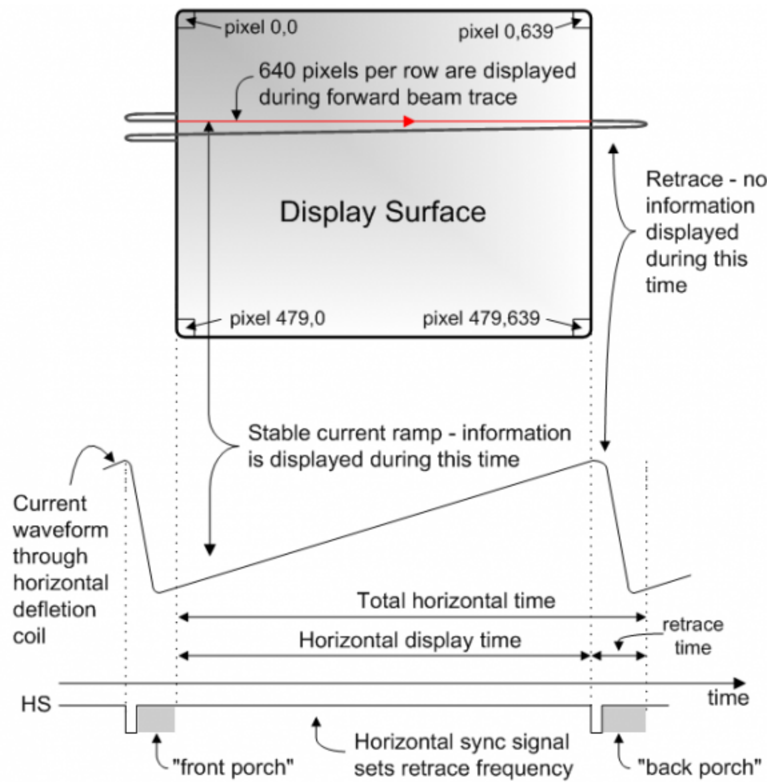
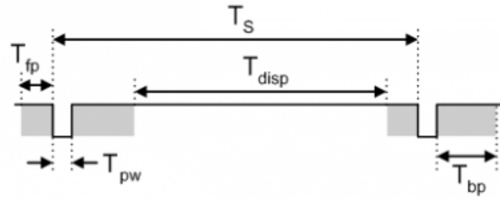


Figure 3: Pixel scanning

A VGA controller circuit must generate the HS and VS timings signals and coordinate the delivery of video data based on the pixel clock. The pixel clock defines the time available to display one pixel of information. The VS signal defines the refresh frequency of the display, or the frequency at which all information on the display is redrawn. The minimum refresh frequency is a function of the displays phosphor and electron beam intensity, with practical refresh frequencies falling in the 50Hz to 120Hz range. The number of lines to be displayed at a given refresh frequency defines the horizontal retrace frequency. For a 640-pixel by 480-row display using a 25MHz pixel clock and 60 +/-1Hz refresh, the signal timings shown in below figure can be derived. Timings for sync pulse width and front and back porch intervals (porch intervals are the pre- and post-sync pulse times during which information cannot be displayed) are based on observations taken from actual VGA displays.



Symbol	Parameter	Vertical Sync			Horiz. Sync	
		Time	Clocks	Lines	Time	Clks
T_S	Sync pulse	16.7ms	416,800	521	32 us	800
T_{disp}	Display time	15.36ms	384,000	480	25.6 us	640
T_{pw}	Pulse width	64 us	1,600	2	3.84 us	96
T_{fp}	Front porch	320 us	8,000	10	640 ns	16
T_{bp}	Back porch	928 us	23,200	29	1.92 us	48

Figure 4: VGA Timing

2.2 ROMs

2.2.1 Accessing the pixels

Now that the VGA output works well, it's time to display some images on the screen. Note that most pictures nowadays are in 24-bit true color. Thus they must be processed and converted into 12-bit color before they are stored in ROMs. This can be easily done because we only need to divide each color channel by 16.

Each pixel in a picture has a coordinate (x, y), with the top-left corner $(0, 0)$, top-right corner $(w-1, 0)$, bottom-left corner $(0, h-1)$ and bottom-right corner $(w-1, h-1)$, where w is the width of the image and h is the height of the image in pixel. Hence we can index a pixel using its coordinates. I chose to concatenate y and x to directly form an address, avoiding arithmetic operations.

2.2.2 Writing data to the board

After we have decided the way of indexing pixels, we can now use `readmemh` to load data into BRAMs. I wrote a python script to generate both initialization file and ROM file.

2.3 RAMs

Some data are dynamic during gameplay, which must be stored in a runtime-modifiable region – RAM. A dual-port RAM is good choice because they can be read and written at the same time, providing much convenience.

3 Graphics Engine

3.1 Displaying images

Normally a game console has a Video RAM where every bit of the screen stores. Since our resolution is 640 x 480 and the color depth is 12-bit, this will require $640 \times 480 \times 12 = 36864000$ bits. But only 1800000 bits of BRAM are available on Basys 3. Therefore, we must adopt the method of calculating the RGB value as soon as a pixel is hit.

The VGA synchornization unit provides the coordinates of the pixel being drawn, which are the absolute coordinates, denoted by (x_a, y_a). And the coordinates used to index a pixel in ROMs are relative coordinates, denoted by (x_r, y_r). We also want to display the image anywhere we want, so we assign another coordinates to it, which is position, denoted by (x, y).

Using these three coordinates, the RGB value at the point (x_a, y_a) is calculated in the following manner:

$$RGB(x_a, y_a) = ROM(x_a - x, y_a - y)$$

. Above are the fundamentals of our graphics engines. In this game, there are two different engines: a Background Engine and an Object Engine.

3.2 Sprites

In a game console, small images are usually concatenated into a big picture like this:

Figure 5: Sprites

Every small image is called a sprite or a tile. When we want to display one of these tiles on the screen, we just need to point out its row and column in the tile bank. And the engine will calculate the right pixel to be displayed according to them and the width and the height of a tile in this bank. Note that the pure blue means the transparent color, when the engine encounters a pure blue pixel, it should display the contents under the layers.

3.3 Background Engine

Usually the background need not much freedom of moving around and need to display a lot of things. Therefore, the background graphics information is stored in the RAM in the following manner: the address decides its position on the screen, and the data in that address decides its image.

Bit 8	Bit 7	Bit 6	Bit 5:3	Bit 2:0
Enable	Y Flipped	X Flipped	Tile Row	Tile Column

Table 1: BG data format

If bit Enable is zero, our engine will ignore the rest of data and treat the whole tile as a transparent one.

Each tile is 16 x 16 in size. If we slice the whole screen by 16 x 16 grids, we can put 40 tiles in a row and 30 tiles in a column. Therefore, the max address is 1200. However, if sprites can only move between grids, it will be strange and not smooth. To address this problem, the engine will receive a signal `x_offset` to move the whole background a little bit out of the grids.

The background engine works as follows: firstly it calculates the address to fetch data according to the coordinates of the pixel being drawn, `addr = ((x + x_offset) / TILE_WIDTH) + (y / TILE_HEIGHT) * TILE_COLS;`

Then according to the data stored in that address, we calculate which pixel we should access in the ROM, `rom_x = ‘TILE_COL * TILE_WIDTH + ((x + x_offset) % TILE_WIDTH);`

X and Y indexing are similar. If we want to flip the sprite, just access the pixels in an opposite direction.

Finally we need a signal `sprite_on` to indicate whether the background has content or it is just transparent. According to the rules, `pixel_on = (rom_data == 12’h00f | ‘ENABLE == 0);`

3.4 Object Engine

Unlike the background, objects need to move freely on the screen, and their positions are decided by their object properties rather than their addresses in the RAM.

Bit 31	Bit 27	Bit 26	Bit 25:16	Bit 15:6	Bit 5:3	Bit 2:0
Enable	Y Flipped	X Flipped	Pos Y	Pos X	Tile Row	Tile Column

Table 2: Object data format

We can only read one object data in a clock, but we must evaluate all sprites at the same time. To accomplish this, we need to build registers inside the object engine, and use a for loop to expand the logic into a parallel one. The rom indexing is similar to background engine.

3.5 Layers blending

We need to mix the signals from both engines correctly. Both engines will output a signal indicating if the current pixel is on (not transparent). The top module will check from bottom layer to the top layer to choose the correct pixel to output.

4 Audio Engine

4.1 Sine wave generator

Unfortunately, although there is PMOD interfaces, the Basys 3 does not have a DAC module. We need to manually simulate DAC using PWM.

Generating square waves is easy, but sine waves sounds better. Therefore, I implement a sine wave generator. The generator has 64 discrete samples of a sine wave. The generator will output correct amplitude according to time and frequency.

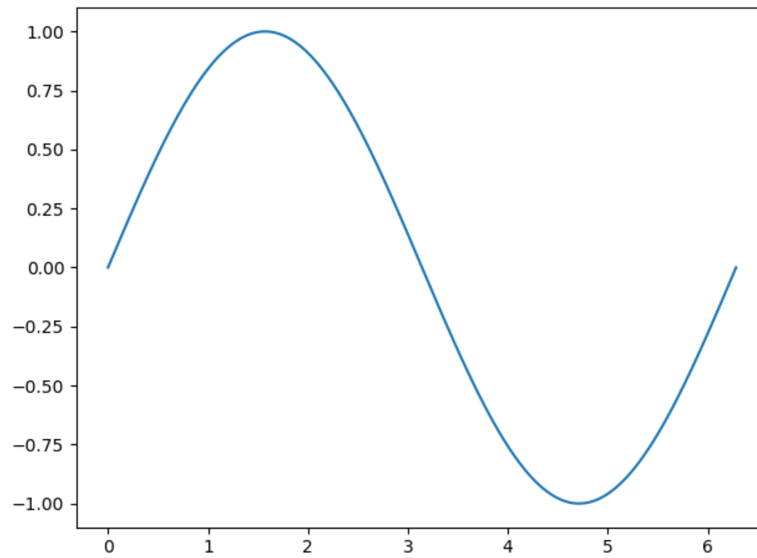


Figure 6: Continuous sine wave

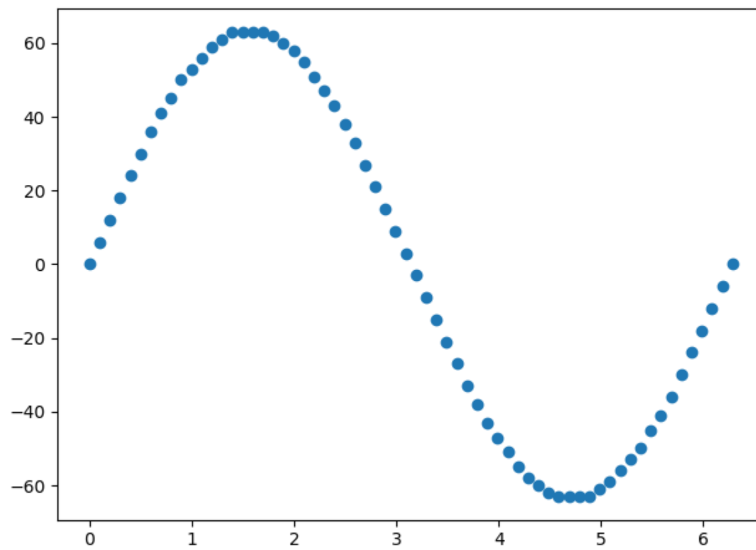


Figure 7: Discrete sampling of sine wave

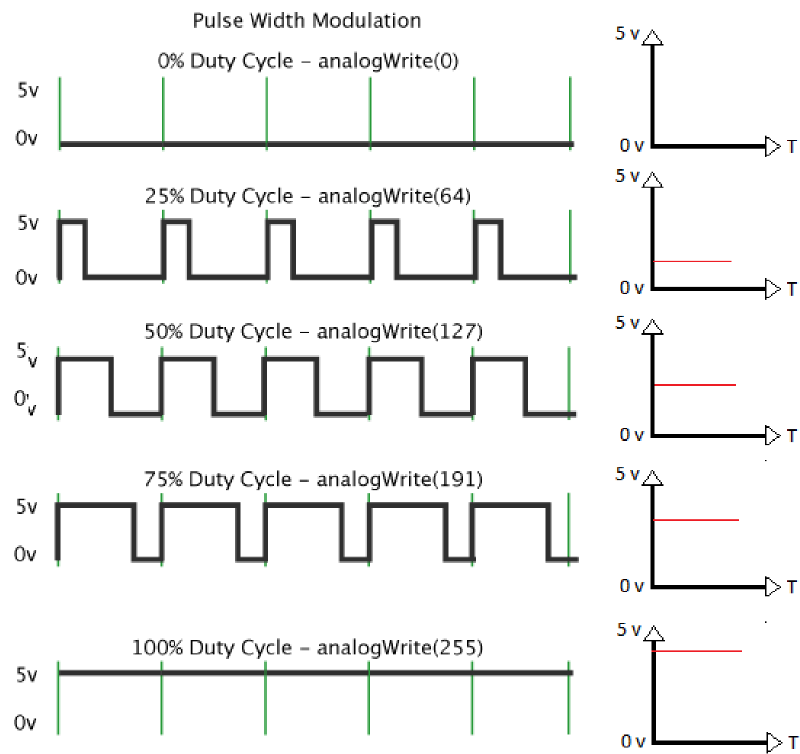


Figure 8: Pulse width modulation

4.2 Music data

With the wave generators, we can now play music by telling them what frequencies we want.

Figure 9: A small piece of music

Technically a music note has four basic properties: volume, pitch, time and timbre. To simplify the circuit, we only use time and pitch to represent a music note.

Bit 31:16	Bit 15:0
Frequency	Duration (ms)

Table 3: Music note data format

What the music engine does is quite simple: it fetches a note in a music sequence, plays its frequency until the duration is satisfied, and fetches the next note.

Usually a song consists of multiple tracks. We just need to play them at the same time and mix their waves together.

5 Game Logic

With the help of these engines, we now need not to deal with the details of graphics. In the game engine, we just need to calculate the correct data to put into RAMs and the graphics engines will do the rest for us.

5.1 Mario

In this game, Mario will only go up or down, so the logic is quite simple. To create the effect of gravity, several counters are used. When we press the up button, we give Mario a upgoing velocity by setting the movement counter to a small value and Mario's status to jumping up. When the counter decreases to zero, we move Mario up one pixel and assign a greater value to the movement counter until we have reached the maximum counter value. After Mario reaches its peak, we set the status to jumping down and assign a big value to the movement counter. Similarly, when the counter reaches zero, we move Mario down one pixel and assign a smaller value to the counter, until we reached the minimum value. Then Mario will keep falling down at the same speed until the up button is pressed again.

During movement, different figures of Mario will be displayed. So we will store the animation frame information as well.

I extract this logic into a module named `mario`. It will communicate with the game engine using several signals.

5.2 Pipes and coins

Pipes can be described using several data: its X position, where its top ends and where the gap ends, all of which are the grid number not pixel coordinates.

The coin is also described by several properties: grid X, grid Y and current animating frame.

The game engine then uses these information to write the correct properties into background RAM in the corresponding addresses.

5.2.1 Pipes and coin generation

When a pipe or a coin reaches the left edge of the screen, we immediately move it to the right. However, it will be boring if they always appear at the same y location. Due to lack of random number generator, we use a simple oscillator to achieve the goal of randomly putting things around.

The oscillators keep adding some numbers again and again every clock. The clock runs so fast the we cannot predict what the next number is. This enable us to get random numbers.

5.3 Collision detection

The game engine needs to do something when Mario hits either the coin or the pipe. Because all the things in the game are square, the collision detection is in fact detecting rectangle intersections, which is quite simple.

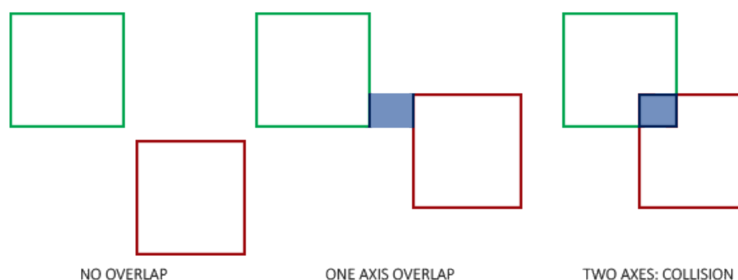


Figure 10: Collision detection

5.4 Scrolling

Now everything is ready and we just need to get things moving.

Moving things needs some cooperation. Firstly we slowly increase the `x_offset` of the background. When it reaches 15, it means we have to move things a grid left and reset the counter to 0. Before we move things, we need to clear the background first, then we write the new data into the background RAM.

5.4.1 Split scrolling

As we can see, the top area is the place where we put the score display. It will be funny if the score moves along with the pipes. So here we use a technique called split scrolling. When drawing the status bar, we do not set any x offset. This can be done by checking the y coordinate of the drawing pixel.

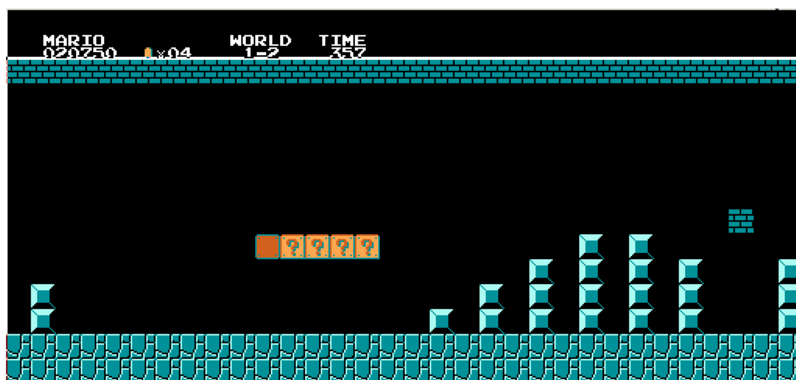


Figure 11: No scrolling when drawing status bar

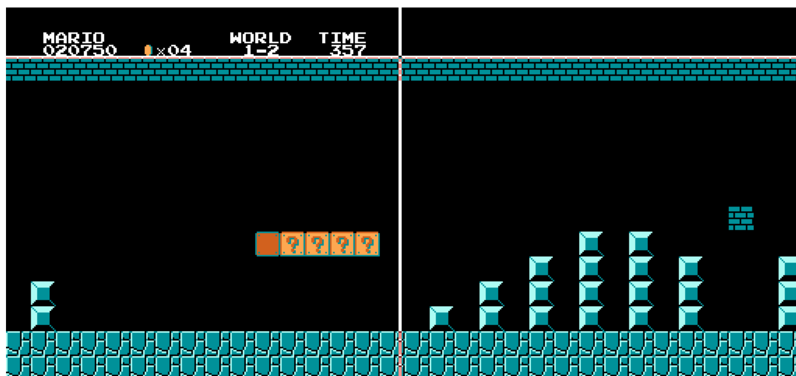


Figure 12: With scrolling when drawing game area

The horizontal white line illustrates the current scanline, and the vertical white line on the left (barely seen) illustrates the `x_offset`. From the figures, we can have an intuition of how split scrolling works.

5.4.2 Parallax scrolling effect

It will be boring if the background is pure black. To add more fun, I decided to lie a back-background behind the background layer. It will be strange if the pipes and the background at the same speed. To make the back-background look far away from the pipes, we need to let it scroll at a slower speed. This is achieved by a clock divider and called the parallax scrolling effect.

5.5 Data writing arrangement

There is only one port for data input in the RAM, but multiple components need to manipulate the data. Since we don't have a CPU, we need to manually switch between the data sources using a counter.

5.6 Game status

The game engine also need to store how much points the player has got, whether the game has been already over.

The score is updated when a coin is eaten or a pipe is passed. We use another module to display score on the top of the screen. When the game is over, a message will also be displayed.

6 Conclusion

6.1 Screenshots

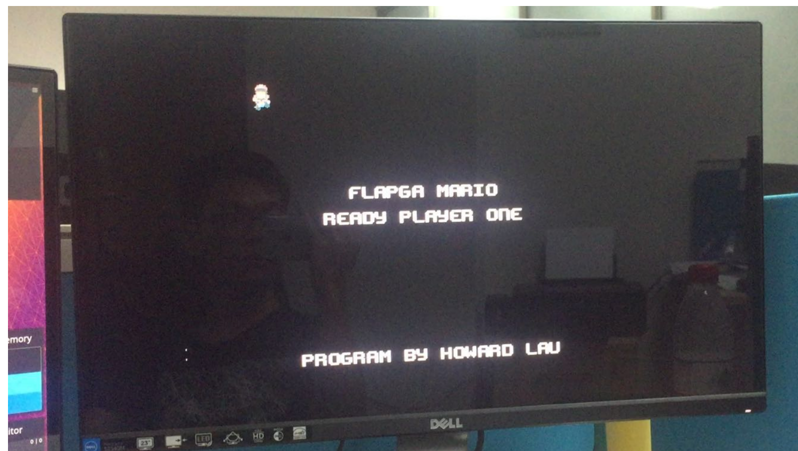


Figure 13: Title screen

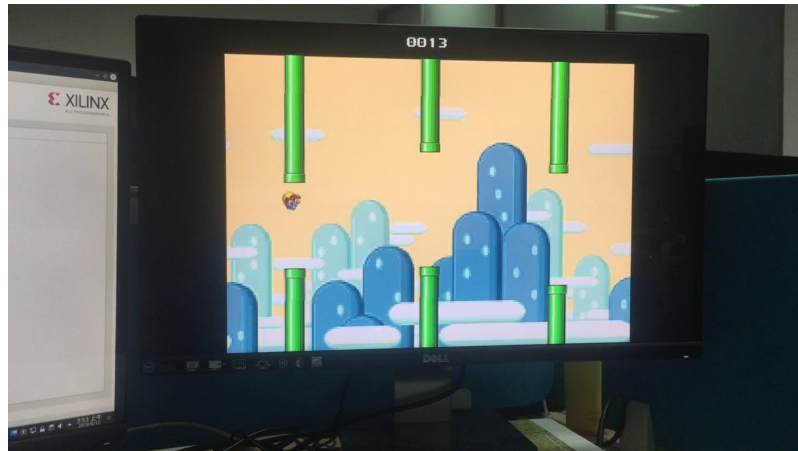


Figure 14: Gameplay

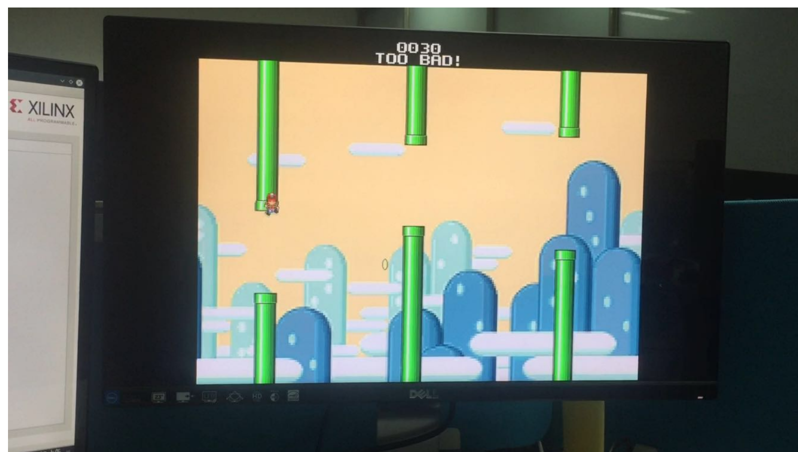


Figure 15: Game over!

6.2 My feelings

Building a video game from scratch is challenging yet interesting. It needs programmer's ability to organize different components well. What's more, it requires comprehensive knowledge of graphics and music. Most importantly, preseverance and patience are the decisive factors of success in building such a video game.

It is difficult to debug due to lack of tools. The only way to debug is read the code carefully again and again and observe the behaviour of the board.

In fact, during the early stages, I encountered countless failures. And sometimes I got stuck without any progress. Because no similar project was done by others before, I had to refer to some hardware specifications of the old-school game consoles. Thanks to Nintendo, its FC inspired me to a great extent. The current architecture adopted is somehow related to FC's PPU. Yet with these

engines, we can do more than a flappy-bird-like game. But as a school project, this seems quite amazing.

Lastly, during coding, I referred to lots of information on the Internet and tried to exploit them in my project. This is a great chance to learn and I think I have learned a lot!

6.3 Possible improvements

1. The game is far from fully exploiting the potential of the FPGA board. If a CPU is implemented, we can build any game we want theoretically.

2. The audio engine can only play the same music again and again and is unable to switch between music. There is no sound effects either. These can be improved but I don't have enough time.

3. The object engine can only deal with 8 objects at the same time. Even FC's object engine is able to process 64 objects in a frame. It uses a technique called Object Evaluation to save hardware resources by reducing number of objects in one scanline to 8 in the clock cycles of HBlank.

7 References

1. **Basys 3 Reference**

   `https://reference.digilentinc.com/basys3/refmanual`

2. **Yoshi's Nightmare (an FPGA based game)**

   `https://embeddedthoughts.com/2016/12/09/yoshis-nightmare-fpga-based-video-game/`

3. **NES (FC) Picture Processing Unit (PPU) hardware behaviours**

   `https://wiki.nesdev.com/w/index.php/PPU`

4. **Split Scrolling**

   `https://retrocomputing.stackexchange.com/questions/1898/how-can-i-create-a-split-scroll-effect-in-an-nes-game`

5. **Audio output**

   `http://www.instructables.com/id/Basys3-FPGA-Digital-Audio-Synthesizer/`

6. **MIDI timing**

   `http://sites.uci.edu/camp2014/2014/05/19/timing-in-midi-files/`

8 Acknowledgement

The SMW sprites resources are provided by YochiThMaster333.
The water level background music is from
`https://www.youtube.com/watch?v=sdb2yQwGjUo` by JesseRoxII.

9 Appendix

9.1 Resources convertion

All resources convertions are done by python scripts, which can also be found on the github repo. Here are the main ideas but not the details.

9.1.1 Image data convertion

Image convertion is done by using python library `PIL`. The program just needs to read in a picture, divide its RGB value, and write data in a format accepted by Vivado.

9.1.2 Music data convertion

Music convertion is more complicated. Its source files are MIDI files. The program needs to find out the correct timing and the correct frequency of a note. This requires some music theory knowledge and many convertions between datas. The details can be found in the source code.