Digital Laboratory Final Report

樂高打僵屍

Student Name: 冉繼元	Student ID: 0810858
Student Name: 杜泓毅	Student ID: 0810864

Introduction

This project is about creating a video game through Nexys4 DDR board and along with other IO devices like keypad or LCD screen. The game itself is a two-player battle game in which the player who lowers the other player's health to 0 wins. The game is designed to have mechanisms like upgrade weapons or increase ability to intensify the battle. The control of the main characters is through four keypads (two keypads for each character) and the game will be displayed on the LCD screen as the output of the FPGA board to the players.

Material

Nexys4 DDR FPGA board*1
8-node keypad*4
LCD screen*1
VGA wire*1

Design and Discussion

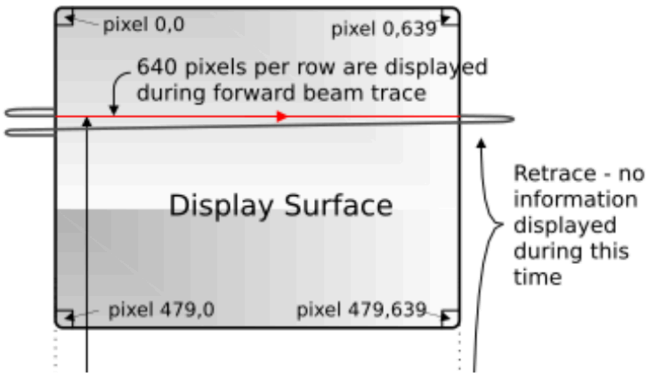
Basic modules, control design

VGA display:

Let's talk about how the display works with VGA signals first. VGA transmits 5 signals, including: R(4 bits)、G(4 bits)、B(4 bits)、Hsync、Vsync. There are two kinds of screen, one is popular recently called LCD, and the other one is traditional screens called CRT, which is the screen we're going to talk about.

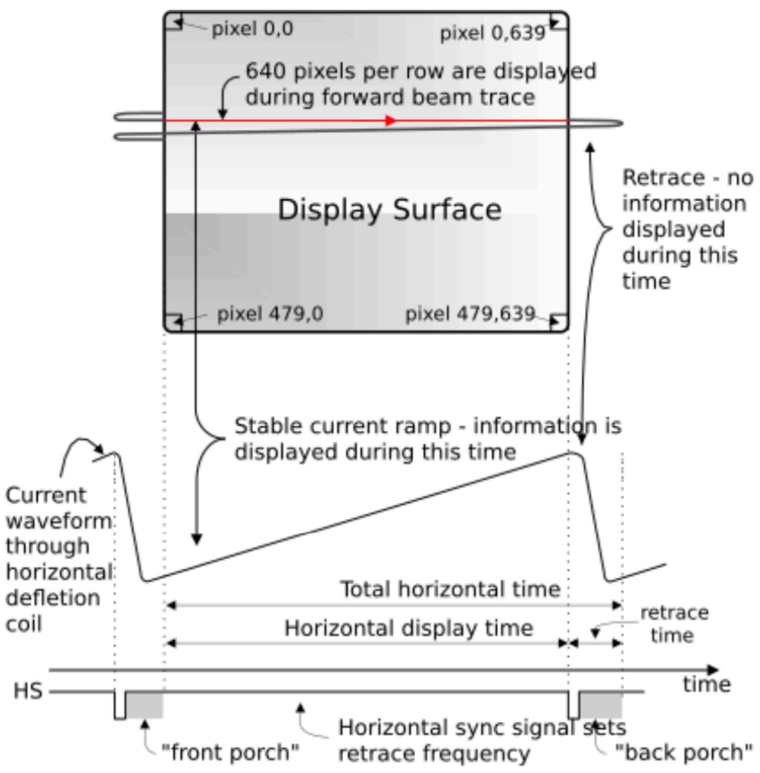
Behind the CRT screen, the heater under the picture tube heats the cathode and releases the electron. By the force of the electric field, the electron moves forward through the picture tube, and finally hits the phosphor on the screen, making the phosphor release the colored light. There are three kinds of phosphors on the screen emitting different colors of light (RGB). Though these phosphors are not on the same point, but since they are small enough, and close enough, so our human eyes will subconsciously fuse these three colors together, and thus the RGB signal can control the color of a single dot on the screen.

All the process mentioned above is the way of forming a single dot on the screen, and to image on the whole screen, the display will repeat the process on all the dots on the screen, and thus forming an 2D array of dots on the screen. The display will handle the dots on the screen from left to right and from top to bottom (as shown in Fig.1). Therefore, there are two situations where we would want to hold the electron ray.


pixel 0,0
pixel 0,639
640 pixels per row are displayed during forward beam trace
Display Surface
Retrace - no information displayed during this time
pixel 479,0
pixel 479,639

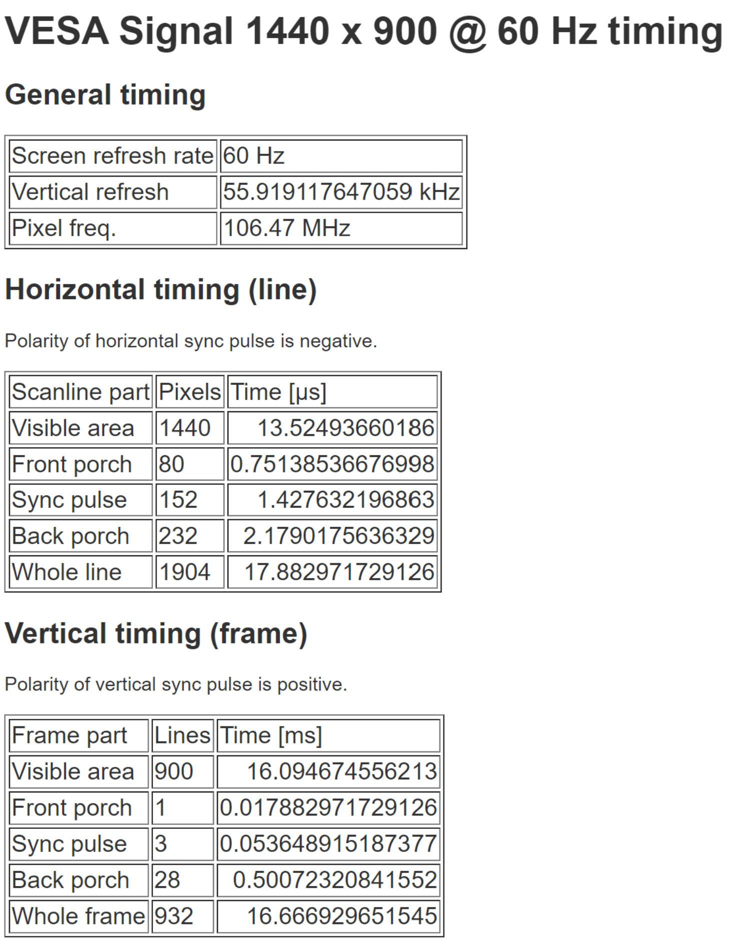
First, when the display finishes a row and is turning back to the left. Second, when the display finishes a frame and is heading back to the start point. To achieve the pause of electron ray, the VGA display code needs to stop transmitting the RGB signal (set R, G, B = 4'b0000) during the time. Besides, the display needs to know when the ray arrives the right most and to start moving forward to the left most, and so does the vertical direction (moves from button to the top), and thus the Hsync and Vsync signal is for this purpose.

As shown in Fig.2, when the ray arrives the right most, the Hsync signal pulls down, and the ray goes back to the left-most position. At the same time, during the sync, front porch, and the back porch, the RGB signal should be pulled down to 0. Otherwise, the ray will release the electron to the surface while the ray is moving to the left most position and pollutes the screen with the wrong color. There is the same pattern for the vertical situation, I'll skip it since the process is the same, while the difference is only between the timing.


pixel 0,0
pixel 0,639
640 pixels per row are displayed during forward beam trace
Display Surface
Retrace - no information displayed during this time
pixel 479,0
pixel 479,639
Stable current ramp - information is displayed during this time
Current waveform through horizontal defletion coil
Total horizontal time
Horizontal display time
retrace time
HS
time
"front porch"
Horizontal sync signal sets retrace frequency
"back porch"

(Fig.2)

There are various resolutions for screens, and different resolutions will require a unique timing. To find the timing for the resolution, one can find the data in http://tinyvga.com/vga-timing, and in our demo, the screen resolution is 1440*900. As shown in Fig.3, the list shows all the timing we need.



VESA Signal 1440 x 900 @ 60 Hz timing

General timing

Screen refresh rate	60 Hz
Vertical refresh	55.919117647059 kHz
Pixel freq.	106.47 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	1440	13.52493660186
Front porch	80	0.75138536676998
Sync pulse	152	1.427632196863
Back porch	232	2.1790175636329
Whole line	1904	17.882971729126

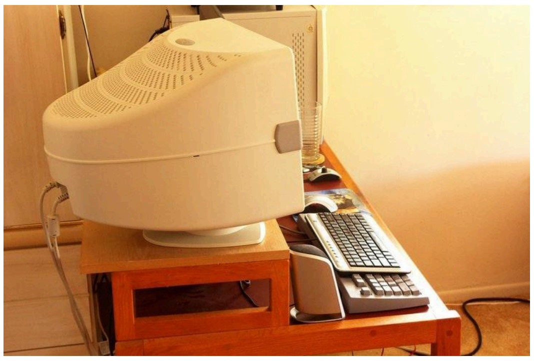
Vertical timing (frame)

Polarity of vertical sync pulse is positive.

Frame part	Lines	Time [ms]
Visible area	900	16.094674556213
Front porch	1	0.017882971729126
Sync pulse	3	0.053648915187377
Back porch	28	0.50072320841552
Whole frame	932	16.666929651545

(Fig.3)

Although our discussion is all about the CRT display, for the LCD display it requires the same VGA signal, so the way coding can remain. Note that nowadays almost all the screens we can see are LCD screens.

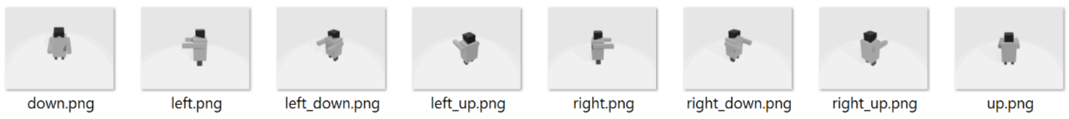


(Fig.4 CRT display)

(Fig.5 LCD display)

Character model design:

We build our character model through 3D painting, and after modeling, we extract the pictures of the character with 8 different facing direction (down, left down, left, left up, up, right up, right, right down) through rotating the angle of view in 3D painter.



(Fig.6 some picture data stored)

Image post processing:

After we have all our pictures we need, we'll need to find a way to store these pictures, and since we know that the VGA display will need RGB signals, what we'll need to store is the RGB information at every point in the pictures. Before we store the information, there is little processing needed to do. First, as shown in Fig.6, the background takes a large portion of the pictures, so I have to cut the pictures. Second, during the game, it is quite important to deal with the case of multiple characters overlapping. To handle it, we decide to make sure all our characters do not show a color of white, as we'll use the color white to determine whether we're showing a character or a background. This problem arises as our pictures are all square, which is an easy way of displaying. As a result, the gray background as shown in the pictures is not welcome (the gray background is generated by 3D painter, which is not evitable), so I have to turn all the background into white. To achieve this, we need to know all these gray colors is formed RGB > a certain threshold, so since I have avoid our character has color of white or light gray, I can simply checking every single dot and turn all the dots with (R > a certain threshold) ,(G > a certain threshold) and (B > a certain threshold) to white. Third, I have to resize the pictures to a moderate size to show on the screen. Fourth, most of the pictures nowadays is stored with R,G,B rang from 0~255, but as mentioned above the VGA takes only 4 bits, so I have to turns the data into the range from 0~15, and at the same time I'll have to store it in binary, so what I do is cut the RGB data into 16 sections (0~15,16~31,…,240~255), and turn the number into binary accordingly.

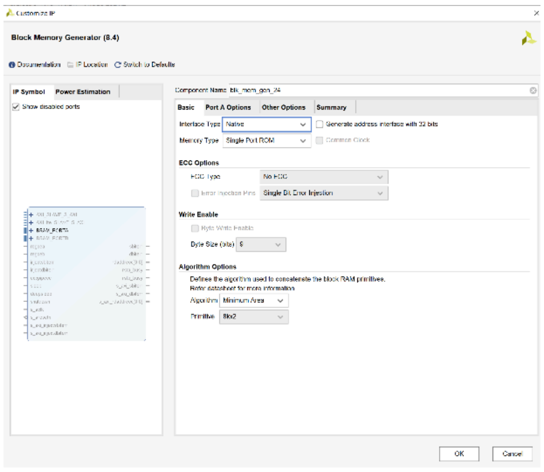
Above is the image post processing, and I do all of these with python code, as the OpenCV library has provided a bunch of useful functions for doing this. However, there is still one thing left to decide, the data type. To know which data type will be used, we shall first explain our plan about storing pictures. Under our plan, there are two ways of storing data. First, use a huge array of flip-flops to store all the pictures. This is doable as my python code can turn the data into Verilog type. Although under this way, we can conveniently and rapidly retrieve data, yet the plan is rejected as the runtime for the simulation is abnormally long (I run the RTL for three days and the result hasn't come out yet). The second plan is utilizing the IP core built in the vivado: Block memory generator. The IP core provides multiple choice of memory to choose, we'll talk about it later. The thing is that the IP requires a special file: .coe for initializing the data. Hence, the output of my python code is the coe file which has to follow the requirement of coe file. The coe file is actually not complicated. It is like a normal txt file with some few restrictions. Here is the restriction:

```
memory_initialization_radix = 2; # telling what radix the data stores
memory_initialization_vector = … #list the data
```

Above is mainly a coe file containing. Every colored picture needs to store 3 coe files, including the value of R, G, B in each pixel.

Block memory generator:

The IP provides a couple of useful storing units, including single port RAM, dual port RAM, single port ROM, and dual port ROM. In our design, the single port ROM is sufficient. In fact, we have tried to use a big dual port RAM storing the whole pixels in the RAM and modify it every frame. However, the plan is shut down as the block memory is not sufficient for these many pixels, so the plan is only achievable through lowering down the resolution. But under my calculation, the resolution will be too low which is not desirable. Therefore, another plan is adopted. The whole design will be discussed later. Now, our design requires us to store every RGB coe file, respectively. As a result, one picture will require 3 ROM to store, and as introduced above, a character will require 8 different pictures representing different directions, so a character will need 24 ROM to store data. Below, I'll show how block memory generator work.


Customize IP

Block Memory Generator (8.4)

Component Name: blk_mem_gen_24

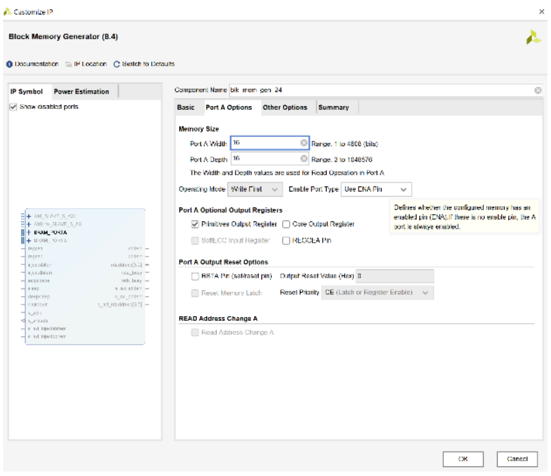
Basic | Port A Options | Other Options | Summary

Interface Type: Native
☐ Generate address interface with 32 bits
Memory Type: Single Port ROM
☐ Common Clock

ECC Options
ECC Type: No ECC
☐ Error Injection Pins: Single Bit Error Injection

Write Enable
☐ Byte Write Enable
Byte Size (bits): 9

Algorithm Options
Defines the algorithm used to concatenate the block RAM primitives.
Refer datasheet for more information.
Algorithm: Minimum Area
Primitive: 8kx2


Customize IP

Block Memory Generator (8.4)

Component Name: blk_mem_gen_24

Basic | Port A Options | Other Options | Summary

Memory Size
Port A Width: 16 Range: 1 to 4608 (bits)
Port A Depth: 16 Range: 2 to 1048576
The Width and Depth values are used for Read Operation in Port A
Operating Mode: Write First
Enable Port Type: Use ENA Pin

Defines whether the configured memory has an enabled pin (ENA). If there is no enable pin, the A port is always enabled.

Port A Optional Output Registers
☑ Primitives Output Register
☐ Core Output Register
☐ SoftECC Input Register
☐ REGCEA Pin

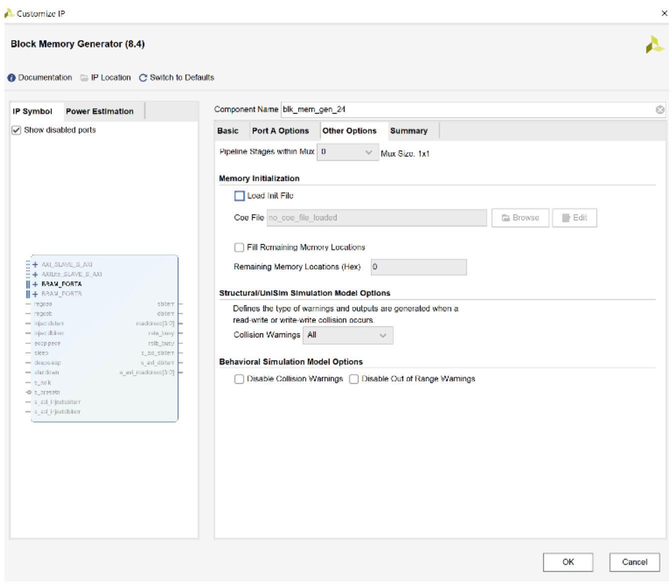
Port A Output Reset Options
☐ RSTA Pin (set/reset pin)
Output Reset Value (Hex): 0
☐ Reset Memory Latch
Reset Priority: CE (Latch or Register Enable)

READ Address Change A
☐ Read Address Change A

(Fig.7) (Fig.8)

First of all, after opening the block memory generator, the UI will be like Fig.7, here, the type of store unit can be selected, different types of units may lead to a different choice for the IP. To see other store units you can read the reference4. As shown in Fig.8 the UI requires the user to specify the data length stored in the store units, and the amount of data required.

The value of these setups require the user to calculate before using the IP, and I do it with the python code mentioned before, it'll tell the size while processing the data. Additionally, the IP also provides an option to use enable pin and reset pin. As we're using ROM, the enable pin and reset pin is not needed. Note that the reset is not clearing all the data to 0 but turning all the data back to the initial state. I saw it in the discussion of Xilinx Discussion. Last, the UI allows the user to import an initial data, which needs to be aligned with the setting of data length and data number. That is, if the data inside the coe file is more than the setup, the coe file is not allowed.


Customize IP

Block Memory Generator (8.4)

Documentation | IP Location | Switch to Defaults

IP Symbol | Power Estimation

[✓] Show disabled ports

Component Name: blk_mem_gen_24

Basic | Port A Options | Other Options | Summary

Pipeline Stages within Mux: 0 — Mux Size: 1x1

Memory Initialization

[ ] Load Init File

Coe File: no_coe_file_loaded — Browse — Edit

[ ] Fill Remaining Memory Locations

Remaining Memory Locations (Hex): 0

Structural/UniSim Simulation Model Options

Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.

Collision Warnings: All

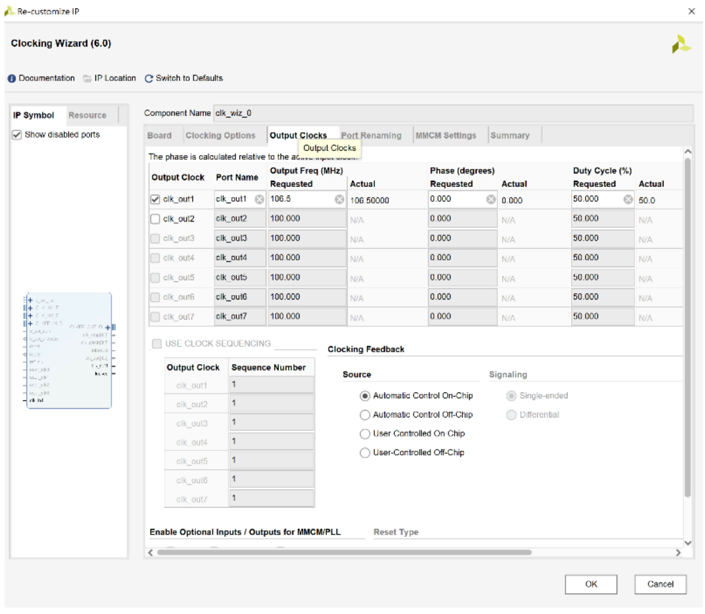
Behavioral Simulation Model Options

[ ] Disable Collision Warnings [ ] Disable Out of Range Warnings

OK | Cancel

(Fig.9)

As for the utilization, the IP has input port: clk and addr, and output port: data, which is pretty clear how to use it. Send the addr into the IP, and it will return the value after 2 clock cycles. (It is the way it works in single port ROM and has no additional delay cycles. If not, the way it works may be slightly different.)


Re-customize IP

Clocking Wizard (6.0)

Documentation | IP Location | Switch to Defaults

IP Symbol | Resource

[✓] Show disabled ports

Component Name: clk_wiz_0

Board | Clocking Options | Output Clocks | Port Renaming | MMCM Settings | Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz) Requested	Actual	Phase (degrees) Requested	Actual	Duty Cycle (%) Requested	Actual
[✓] clk_out1	clk_out1	106.5	106.50000	0.000	0.000	50.000	50.0
[ ] clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000	N/A
[ ] clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A
[ ] clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A
[ ] clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A
[ ] clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A
[ ] clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A

[ ] USE CLOCK SEQUENCING

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Clocking Feedback

Source: [✓ Automatic Control On-Chip] Automatic Control Off-Chip / User Controlled On Chip / User-Controlled Off-Chip

Signaling: [✓ Single-ended] Differential

Enable Optional Inputs / Outputs for MMCM/PLL | Reset Type

OK | Cancel

(Fig.10 CLOCK WIZARD)

Clock wizard:

As we say that the resolution is not the same for different types of screens, but the frame fate is normally the same, at 60 HZ. This implies that the clock rate used has to be different. As our screen is 1440*900, the clock rate needed is 106.5MHZ, but the frequency produced by NEXYS 4 DDR is only 100MHZ, which is insufficient and cannot be achieved by the frequency divider taught before. Hence, I use an IP core provided by Xilinx called clock wizard, which is in fact widely used in different designs. The clock wizard is convenient in that the IP can output a various range of clock rates according to the input clock. With the IP, I can easily generate the 106.5MHZ clock rate.

The Horizontal counter and Vertical counter:

As mentioned above, the screen is imaged pixel by pixel. Therefore, we need a horizontal counter and a vertical counter to tell the entire circuit which position the electron ray is shooting. The horizontal counter is a counter working with the 106.5MHZ clock rate and output the H_cnt_calue. The vertical counter will sense the H_cnt_value, and count for the V_cnt_value. The H_cnt_value and V_cnt_value are mostly used in the display circuit, but they are also used to perform like 60 Hz clock rate, since it is 60 Hz when H_cnt_value and V_cnt_value equal zero.

The H_sync and V_sync signal:

Both signals are generated by sensing the H_cnt_value and V_cnt_value. If H_cnt_value is smaller than the sync pulse which is listed on Fig.3 or searched by the website I pasted, the H_sync signal is 1. Otherwise, it is 0. Additionally, the V_cnt_value works in the same way.

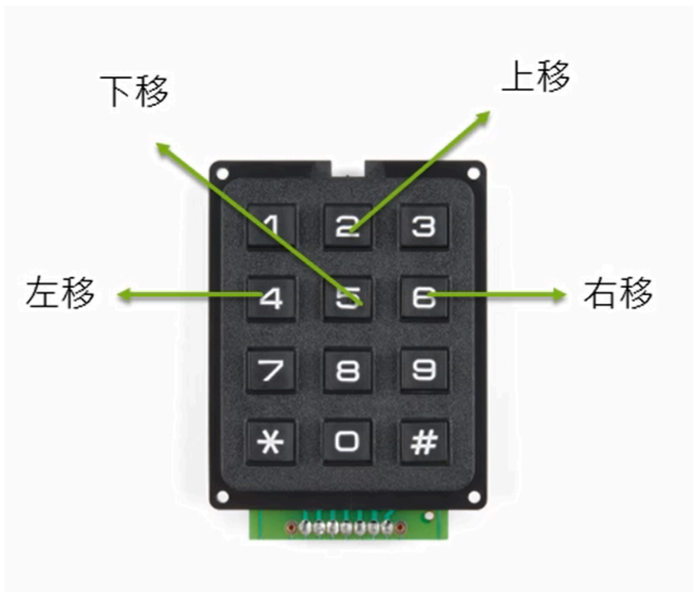
Game mechanism design

The start mechanism & the ending:

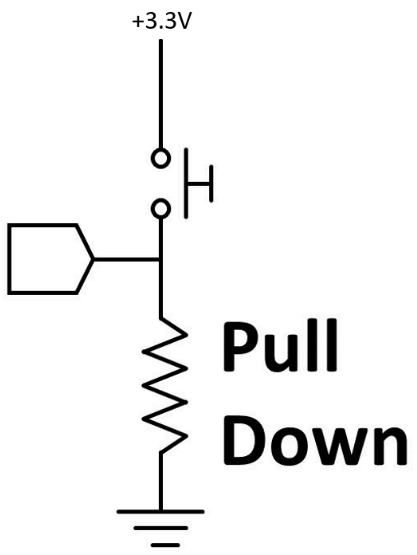
Before pushing the start button, we hope the game won't start, and all the time-related flip-flops cannot start between this duration, so I set a signal stored by flip-flop which will be pulled up when the start button is pushed. Otherwise, it will sense the HP of the both characters, if one of HP is 0, the start will pull down. Beside the situations mentioned, the start signal will hold the previous state. With such a signal, I place a multiplexer before almost every flip-flop, and they will work normally only if the start is 1. It is like the mechanism of reset. If it is not reset, it works normally. If start is 0, there will be 3 conditions. First, no one wins. It's the case that the game just starts, under this condition, the screen will show a simple start picture I designed, and with a single bit added. I make it blink with a certain frequency, since I think it is better. If the human side wins, the screen will show a human win picture. Otherwise, if the zombie side wins, the screen will show a zombie win picture.

Moving signals generating circuit:

The way to control the movement of two characters in the game is using the keypad. In many PC games, we often see the games control the direction (of either characters or other main elements in the games) through either WASD buttons (for up, left, down and right) or the direction buttons on the keyboard. For players to get used to the game faster, I decide to use the similar relative position for the characters control and design the circuit to decode the signal from the keypad to realize the instruction as follow:



First, we need to decide how to detect the activity of the buttons on the keypad. In the project, the button activities on the keypad are passed using a logic HIGH COM node and its connection to each wire when the buttons are pressed. For the COM node of the keypad, it will be connected to the 3.3V output from the FPGA board. At the same time, all the other wires (A~G) are pulled down by connecting the nodes to ground through a resistor (one resistor for a wire). The structure is similar to the graph:



When a button, let’s say 2, is pressed, the COM node will directly connect to wire A and wire F, making these two nodes change to HIGH. By measuring the voltages of the wire A~G, we can locate the buttons that are being pressed. Of course, every button is designed with the functionality of debouncing. This method, however, will cause a disadvantage for the character movement designed in this project. We will talk about that later.

In this game, the characters are designed to be able to move in eight directions: up, down, left, right, up-right, up-left, down-left and down-right. The simplest way to achieve this is to generate a relative signal when a corresponding button is pressed and control the characters according to these signals. For example, when button 2 is pressed, meaning that the character should be moving upward, the signal Up will be assigned to HIGH. When button 6 is pressed, meaning that the character should be moving right, the signal Right will be assigned to HIGH. Through the combination of these signals, we can control the characters to move in each of the eight directions. The basic situations are as follow:

1. When there is only one signal among Up, Down, Left and Right is HIGH, the characters move straightly toward the direction.
2. When either Up or Down is HIGH along with either Left or Right, the characters move diagonally toward the combined direction. (for instance, when Down and Left are HIGH at the same time, the characters move toward down-left of the screen)
3. When both Up and Down or both Left and Right are HIGH at the same time, the character will stand still.

Also, because the graphs of the characters change as the characters move in different directions, we need to have signals (zombie_facing or main_facing) to store the direction the characters are going. These signals are decoded as 0 for facing-down and increase clockwise. This signal is stored with DFF so that the direction of the characters facing will stay the same when the characters are not moving.

However, this design has a problem with the method for detecting the activity of the buttons we have mentioned above. When both button 2 and button 4 are pressed, wire A, wire B, wire E and wire F will all be charged to HIGH. Although we only press two buttons, the result of the detection will indicate that button 2, button 4 and button 5 are being pressed at the moment. The reason for the false detection of button 5 is because both wire B and wire F are HIGH due to the pressed button 4 and button 2 respectively. Thus, when button 2 and button 4 are pressed, Up, Down and Left signals will all be HIGH, and the characters will incorrectly move toward left but not up-left. The situation is similar when button 2 and button 6 are pressed at the same time. To solve this problem, we make an assumption here. We assume that only two buttons are pressed in these two situations. Thus, when button 2 and button 4 are pressed, the logic circuit we designed will not pull up the Down signal but only to make Up and Left HIGH. When button 2 and button 6 are pressed, only Up and Right will be HIGH. This way, we can control the characters to move in diagonal directions (notice that when moving toward down-left (pressing button 4, button 5) and down-right (pressing button 5, button 6), the situation of false detection will not happen since wire A is still LOW).

Before we talk about the mechanism for moving the characters, we need to first explain how we display the characters. For each main object (zombie, shooter and supply box), it will have two x-positions and two y-positions to indicate overall position of the objects. When the detection logic find that the VGA scanning process is now entering the position of an object, it will use the current H_cnt_value and V_cnt_value to calculate the address in order to acquire the graph information (R,G,B) from ROMs and display the pixel according to these information.

So, to make the characters appear to be moving, when one of the moving signals (Up, Down, Left and Right) is activated and is not canceled by other moving signal, the x-position and y-position of these objects will be adjusted according to the signals (x-position increase toward the left of the screen and y-position increase toward the bottom of the screen). These position movements are realized only at the beginning of each VGA scanning process (H_cnt_value = 0 and V_cnt_value = 0) so that the positions can be updated at a low rate of 60Hz instead of the clock rate of 106.5MHz to realize a visible moving. While the shooter has a simpler moving logic (move whenever there are moving signals activated and not canceled by each other), since zombie will have skills that instantly changes its position or increase the moving speed and cannot move its position when attacking, its control logic is much more complicated than the shooter and will be summarized along with the skill explanation.

Zombie skill & attack sensing:

The zombie is able to use the skill, which includes speedup and flash (move forward a certain distance instantly) once it collects enough supply boxes. As we use an independent keypad to cast skill and attack, a keypad module will handle the sensing, including counting the cooldown time and output the skill signal. Additionally, since we want to show the cooldown time on the screen, the module will also output the cooldown time signal.

The CD bar and icon display:

The cd bar is formed by using the signal of cooldown time. Through dividing it into a moderate value and using this value to decide the length of the CD bar. With this design, the cooldown time can be shown with a simple CD bar, and so the player can visually see whether the skill is cooldown or not. As for the human, there is a signal called Weapon deciding which weapon is being used. This signal is mainly controlling the bullet. Yet through a multiplexer sensing the signal, we can choose the icon to be shown, and the player can see which weapon is being used.

The zombie's position and HP calculation:

The skill signal is transmitted to a module, which handles the movement of the zombie, since all the skill and attack signals are related to the movement. Besides, there is another keypad that controls the movement of the zombie, so this module will sense both the movement signal and the skill signal at the same time and calculate the correct movement. All this is done by correctly assigning the multiplexer, which is quite complicated, and I will briefly explain. We hope the movement update follows the frame rate which is approximately 60 HZ, and thus we take the H_cnt_value and V_cnt_value into consideration. In our plane, the attack signal will make the zombie jump a short distance and hold the position in a short moment, and additionally it is handled in the first priority. The task of holding position is simply done by just a counter with some multiplexer. On the next priority, the module handles the skill flash signal. Receiving this signal, the module senses the facing direction signal and adds a value onto the zombie's position signal. One thing that is worth saying is that the module has to also consider the boundary of the map, or the zombie will rush outside the map and cause error. If there is no attack signal and skill flash signal, the zombie simply moves with a speed signal. The speed is decided by whether the skill speedup is working or not. When sensing skill speedup signal, the speed signal will be a higher value, and the zombie will then move faster.

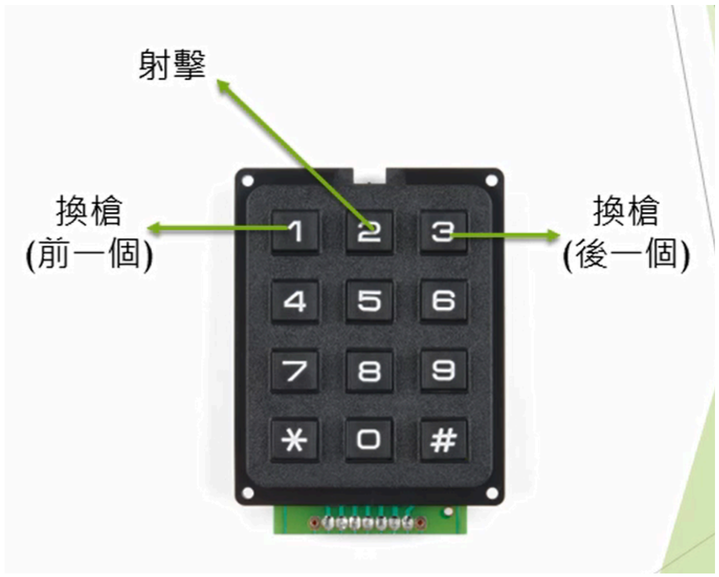
Surely, under the moving, the module still needs to consider the boundary of the map. Besides, there is another signal handling the zombie’s facing direction. The signal is mainly controlled by the input moving signal and holds if there is an attack or flash signal. This module will also handle the HP signal of the zombie. After taking a hit signal, the hp signal will then be decreased with a certain value. At the same time, another signal called blood_steal will be pull-up if a zombie's attack hits the human, and the module will sense the signal, adding the hp signal. Last but not least, this circuit will output the horizontal, vertical positions, facing direction, and the hp signal. The position and direction signals are used in the display circuit and the circuit checking if the bullet hit or the zombie’s is validated or not. The hp signal is used for checking if the game is ending or not and displaying the zombie's health bar.

Zombie’s attack validation:

There is needed a circuit judging the zombie’s hit is validate or not. The judgment will be done by taking consideration of both the zombie's and human’s position. We calculate the position of the zombie's center, and if the center is located at the human’s box, the attack will be judged as validate. At the same time, trigger the blood_steal signal. Otherwise, the attack is not validated, and of course the zombie can’t be healed. To complete this task, the main quest is preventing the multiple validation being judged in one attack. We achieve this by using H_cnt_vaule, V_cnt_value, and one other counter signal, which count the zombie’s attack position holding time. With these, we can ensure that the attack will only be judge once under one attack.

Shooting and bullet generation:

For the shooter, the way for him to defeat the zombie is by his weapons. Besides the initial pistol, machine gun, shotgun and mine can be acquired as the game process (by collecting the supply box). Shooting activity is also generated by the keypad. The control is as follow:



The control for shooting and bullets is separated into two parts. One is to detect the action of the keypad and generate the corresponding signals. The other is to control the bullets, including bullet generation, moving and their interaction with boundary and zombie, according to signals from other modules. Let's first talk about the first part, keypad detection.

For the keypad detection, the method we used is similar to the button activities detection method used in characters moving. Unlike moving, the shoot control will not lead to false detection of the buttons as long as the player presses only button 1, 2, 3. The circuit is designed to be as follow:

1. When button 2 is pressed, the shooter either shoots bullets or places mines according to the current weapon. There will be a cooldown time between successive shoot action and the cooldown time varies from weapons to weapons.
2. When button 1 or button 3 is pressed, the weapon shooter is currently holding changes among the available weapons at the time.

To realize these functionalities, some signals need to be created to indicate the activity:
Shoot: Indicate whether the shoot operation is required by pressing button 2.
Weapon: Indicate the current weapon.
Cooldown: Indicate how long it will take to be able to activate the next shoot operation.
Level: Indicate what weapons are available now. Increase when the shooter collects the supply box.

When button 2 is pressed, the circuit will test if Cooldown is 0. If Cooldown is 0, Shoot will be assigned to HIGH and stay at its value (since it is stored in a DFF) until it is reset at the beginning of each VGA scanning. (The reason for it is to not affect the VGA display process by only generating bullets at a specific moment). Also, Cooldown will be assigned to a value depending on the Weapon. Cooldown will decrease its value every clock cycle as long as its value is not 0. If Cooldown is not 0 when button 2 is pressed, the circuit will ignore the shoot requirement and keep Shoot at LOW. Through this, we can generate a timing gap between each shooting operation to realize the cooldown for shooting. For button 2, its detection circuit can be designed without the functionality of debouncing since Cooldown will do the work to ignore the bouncing of the button at the beginning of being pressed.

When button 1 or button 3 is pressed, the value of Weapon is either decreased or increased by 1 to indicate the weapon swapping. However, the value of Weapon should be kept in a range indicated by Level to show that some weapon may not be available if the shooter has not yet got enough supply boxes. For example, when Level = 2, Weapon can only be in the range of 0~2. When the increment or decrement of Weapon makes it out of range, its value will be changed to the beginning (= 0) or the end (= 2 in the example) of the value range. The value of Level will increase by one every time the shooter gets a supply box, which is indicated by a signal Get, until it reaches 3. This way, we can control the current holding weapon based on the available weapons at that time.

For the bullet control part, besides receiving Weapon and Shoot signals from the keypad detection part, it will also require the position of the shooter to determine the initial position of the bullets and the position of zombie to detect the hit activity of the bullets to the zombie. Whenever each of the scanning processes of the VGA display comes to the end, the bullet control circuit will detect if the Shoot signal is HIGH. If it is HIGH, it will push the information of bullets into the bullet array to generate new bullets. This information includes a valid bit, position of the bullet and the direction it is going. The initial position of the bullets will be in the middle of the shooter and the direction the bullet is going is determined by the direction the shooter is facing when the shooting operation is activated. When a bullet is generated, the information will be put at the first location of the bullet array while other existing information is moved to their next location (for the shotgun, three bullets will be generated each time the shooting operation is activated). The maximum number of existing bullets (including mines) is set to be 50. When the number of bullets reaches the limit, the oldest existing bullet will be deleted. This is to prevent the unlimited mines appearing on the map. Also, the bullet control circuit will detect the position relationship of bullets with boundary or zombie. Whenever a bullet reaches the boundary of the map or hits a zombie, the bullet will be deleted by changing its valid bit to LOW and setting its other information to 0. Furthermore, when bullets hit the zombie, a zombie_hit signal will be generated and the health of the zombie will be decreased by a value depending on zombie_hit. The value of zombie_hit is depending on how many bullets hit the zombie at the moment so that each of the bullets from the shotgun can deal its own damage to the zombie. Both the generating and deleting operation happen at the end of each VGA scanning to not disrupt the scanning display of VGA.

Mine:

The mine is just like a bullet that will not move, so we continue to use the bullet system to represent the mine. What makes a difference is that we add another signal to judging whether the bullet is mine or not. The signal is a 1-bit signal called mine. Besides, at the system checking if the bullet is hit or not, we will also check the mine signal. If it is a mine, the circuit will output an explosion signal and take the position of the bullet as an explosion position, outputting it. The explosion signal and its position is then transmitted to the display system, and the system will then display an explosion picture at the position for a few frames. The task of holding the explosion pictures for a few frames is just what we do in holding the zombie's position while attacking, done by a counter and a few multiplexers. In addition, the explode signal will also transmit to the module which calculates the zombie's position and HP. With the signal, the zombie's position will be held for a few frames. This way, we can represent the effect of zombies being stunned by the explosion.

Supply box generation:

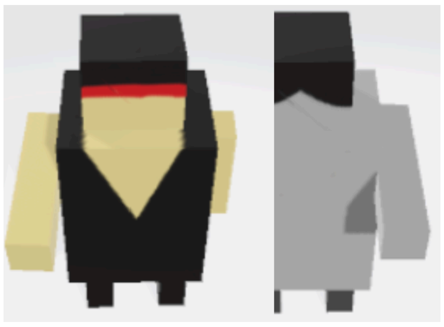
There is one more thing that is important in this game, that is, supply boxes. Through collecting the supply box, both zombie and shooter can get their ability improved and have a better advantage over each other. The control of the supply box is packed into one module. This module includes box generating and box-collecting detection. The information about the supply box is similar to that of bullets. Supply box has position signals to indicate where on the map the supply box should be displayed and a valid bit to indicate whether the box is currently on the map or not. First, let's start with box generating.

At the beginning of the game, the supply box will not appear on the map initially. Instead, it will wait for a period of time for the box to emerge. This period of time can be simply realized by a counting-down counter. Whenever the game starts or the box is collected, the counter will be assigned to a number and decrease its value over time. When the counter reaches zero, the supply box will be generated and displayed on the map. The position where the box will appear depends on the position of both shooter and zombie. By evaluating the position of shooter and zombie when the counter reaches zero, the position of the box will be set to one of the eight possible locations. Since the position of shooter and zombie is constantly changing, generating in this way will make the generation of the supply box seem to be random.

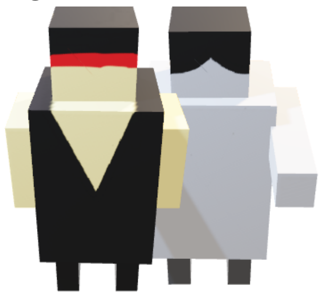
As for the collection of the box, the control circuit will compare the middle points of shooter and zombie to the position of the supply box. Whenever one of which overlaps with the position of the box, the valid bit will be pulled down to LOW, the counter will start counting and the signal that indicates the collection of the box (one signal for each character) will be raised to HIGH. According to these signals, the shooter and zombie can unlock weapons or have a new ability to intensify the battle. Also, just like the bullets, the generation and deletion of the supply box happen only at the end of VGA scanning to not affect the scanning display.

Object display of shooter, zombie and supply box:

One of the most difficult parts to design is to realize the object display of the shooter, zombie and supply box. The reason for this is due to the source graph of these objects and the way we display these graphs. Because of the white background of these graphs, the objects situated at smaller y-position may be blocked by the white background of the objects situated at larger y-position just like the situation in the picture.

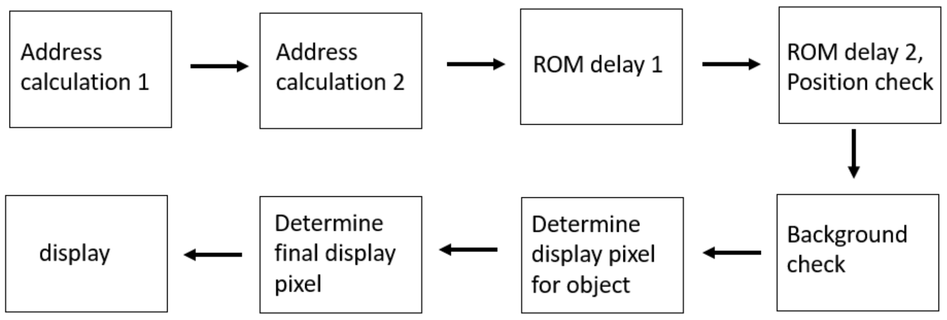


The solution for this is to determine the priority of objects and indicate the white part of these objects. Prior of the objects indicates what object is on top of another object and is determined by the higher y-position (position_y[0]) of each object. The object which is on top of the other will have a higher priority in display. However, when the pixel of the upper object is white, the display pixel will be changed to depend on the pixel of the lower object. This way, when the currently displayed pixel of the upper object is the white background, it will display the content of the object behind it. Here, we assume that the entire graph of shooter, zombie and supply box don't have a single white pixel except for the background. The resulting effect is as follow:



(The graph only shows the simulation the resulting effect of the logic mentioned)

But to realize this solution, we need to be able to evaluate the RGB of the graph that should be displayed. This forces us to acquire the display RGB from ROMs a few cycles before it is sent to the screen. Also, the complexity of the logic (calculating the ROM address through the position of the objects, sending it to ROM, determining the priority etc.) and the short clock period require the task to be pipelined to complete. The pipelined task is as follows:


Address calculation 1
Address calculation 2
ROM delay 1
ROM delay 2, Position check
Background check
Determine display pixel for object
Determine final display pixel
display

Notice that after determining the display pixel for objects, it will take another cycle to determine the final display pixel. This cycle is for bullet display. We set the bullets to be prior to any other objects and the reason is that

1. Since the bullets originate from the shooter, going away from the shooter (except for the mine) and there is no way for the shooter to outrun the bullets, the overlap situation is only a thing in the case of mine.
2. When a bullet overlaps with the zombie, it will be instantly deleted and disappear on the map, making the overlap of bullets and zombie less serious.
3. The color of the supply box and the bullets are similar, so the display is not so weird when the bullets overlap with the box.
4. If we want to realize the correct overlap display between the bullets and the object, the logic will be too complicated (compare the position of the objects and at most 50 bullets).

Due to these reasons, we then decided to make the bullets to be on top of any other objects on the map (shooter, zombie and supply box).

Health bar display:

The display of the health bar is relatively simple since it is outside the boundary of the map and will not interact with other displays. For the health bar, we first decided its total display region. Next, we will decide which part of this region is black (indicating the health lost) and which part is green (indicating the health left) based on the health of the shooter and zombie. The green part of the health bar is designed to be moving away from the center of the screen as the health decreases. The difficulty of this display is to decide at which H_cnt_value and V_cnt_value should the health bars be displayed and how the health bars change as the health of the shooter and zombie change. But it didn't take long to design, test and get a correct result.

Conclusion

This project includes many techniques for digital system design such as IP usage, pipelining etc. and the knowledge learned throughout the lab such as keypad detection and counters. Also, this project stands as the biggest project we have made so far. Such a large project will require systematic design and integration that can ease the debugging and testing process. Through this project, we may learn about systematic work distribution and cooperation which may be a good simulation for the future projects and how to use the past experience to solve the problems in each work assignment.

Reference

Fig.1 :
https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual
Fig.2 :
https://digilent.com/reference/learn/programmable-logic/tutorials/vga-display-congroller/start
Fig.3 : http://tinyvga.com/vga-timing/1440x900@60Hz
Fig.4 : https://www.inside.com.tw/article/11766-crt

Fig.5 :
https://www.google.com/url?sa=i&url=https%3A%2F%2Fnews.xfastest.com%2Freview%2F1544%2F1544%2F2%2F&psig=AOvVaw2ZoEGx1KNVgunK5Xf5T412&ust=1654246815028000&source=images&cd=vfe&ved=0CAwQjRxqFwoTCIjv1520jvgCFQAAAAAdAAAAABAD

1. VGA display: https://digilent.com/reference/learn/programmable-logic/tutorials/vga-display-congroller/start
2. https://chien0902.pixnet.net/blog/post/7000053
3. Coe file: https://www.796t.com/content/1549091012.html
4. Block memory generator: https://blog.csdn.net/wordwarwordwar/article/details/82721889
5. Clock wizard: https://www.youtube.com/watch?v=ngkpvMaNapA