

Final Project Report

TEAM 10

111062122李彥呈

111062137陳啟綸

Table of Content

I. Introduction and Motivation

II. Difficulties in our original topic

1. Memory Configuration

2. Computation Time

III. Top-down view of our game

IV. Features

1. P2P

2. Animation

3. Power Bean

4. Others

a. Scene Control

b. Algorithm of ghost

c. Moving of pacman and ghosts

V. Experimental Results

VI. Work Distribution

VII. Epilogue

I. Introduction and Motivation

Pacman is a well-known and classic game, it has simple yet interesting rules, making people in the 80s enjoy it a lot. Our aim in this project is to remake most parts of the original Pacman and change some of the features.

Since we have done Pacman in I2P(I) course, we decided to remake it using HDL (Hardware Description Language). We think after this project we can totally understand the difference between hardware language and software language. More importantly, to know what feature or function is easier to realize in either hardware or software logic.

II. Difficulties in our original topic

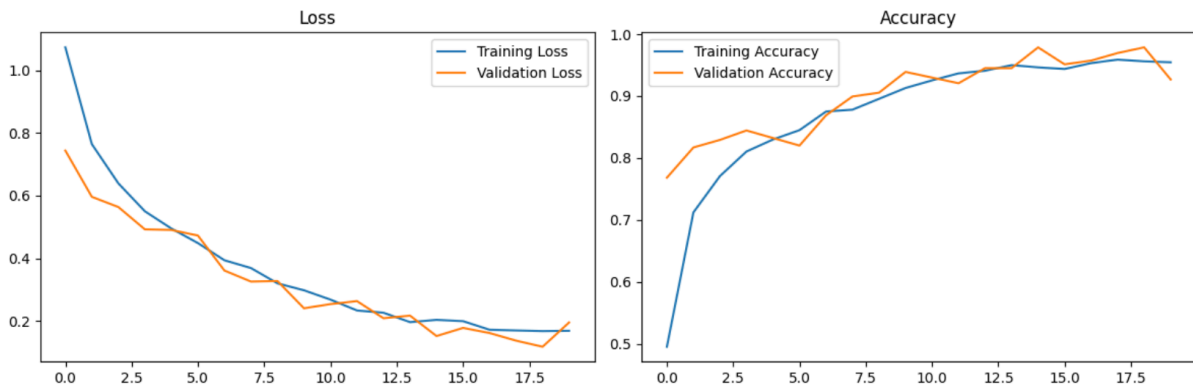
1. Memory Configuration

The first version of our model was composed of 3 CNNs and 2Dense Layers. We can reach about 98% accuracy by using this model (Fig. 1-1).

```
+ 程式碼 + Markdown | ▶ 全部執行 ↺ 重新啟動 🗑 清除所有輸出 | 📄 變數 📖 大綱 ... Python 3.10.6
Epoch 3/20
530/530 [=====] - 5s 10ms/step - loss: 0.6390 - accuracy: 0.7708 - val_loss: 0.5635 - val_accuracy: 0.8293
Epoch 4/20
530/530 [=====] - 6s 11ms/step - loss: 0.5499 - accuracy: 0.8105 - val_loss: 0.4922 - val_accuracy: 0.8445
Epoch 5/20
530/530 [=====] - 6s 10ms/step - loss: 0.4949 - accuracy: 0.8299 - val_loss: 0.4903 - val_accuracy: 0.8323
Epoch 6/20
530/530 [=====] - 6s 10ms/step - loss: 0.4484 - accuracy: 0.8450 - val_loss: 0.4725 - val_accuracy: 0.8201
Epoch 7/20
530/530 [=====] - 6s 10ms/step - loss: 0.3936 - accuracy: 0.8750 - val_loss: 0.3608 - val_accuracy: 0.8689
Epoch 8/20
530/530 [=====] - 6s 10ms/step - loss: 0.3688 - accuracy: 0.8781 - val_loss: 0.3257 - val_accuracy: 0.8994
Epoch 9/20
530/530 [=====] - 6s 11ms/step - loss: 0.3200 - accuracy: 0.8956 - val_loss: 0.3274 - val_accuracy: 0.9055
Epoch 10/20
530/530 [=====] - 6s 10ms/step - loss: 0.2978 - accuracy: 0.9131 - val_loss: 0.2404 - val_accuracy: 0.9390
Epoch 11/20
...
Epoch 19/20
530/530 [=====] - 6s 12ms/step - loss: 0.1675 - accuracy: 0.9561 - val_loss: 0.1176 - val_accuracy: 0.9787
Epoch 20/20
530/530 [=====] - 6s 11ms/step - loss: 0.1684 - accuracy: 0.9546 - val_loss: 0.1951 - val_accuracy: 0.9268
```

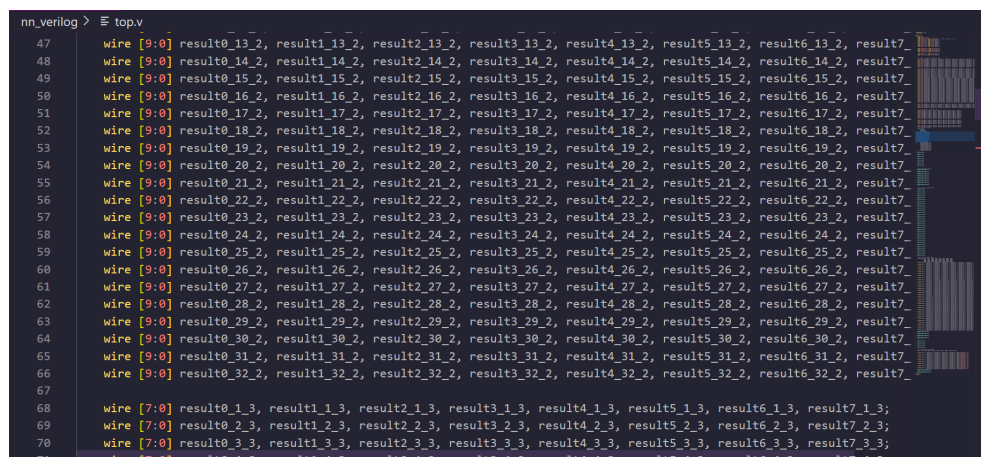
▲Fig. 1-1 The results of our first model

This model is considered effective since validation loss is continuously decreasing while validation accuracy is increasing (Fig. 1-2). Which indicates that the model is actually learning to recognize our music notes, rather than memorizing answers.



▲Fig. 1-2 The line graph of our first version model

However, after we finished implementing the first layer convolutional neural network, we realized that it is not possible to use brute force, that is, declare a new wire for every element in the matrix. In conclusion, our memory usage exceeds the limit of our Basys 3 FPGA. The following figure (Fig. 1-3) shows a small part of our original implementation (Which we think is really dumb right now :)

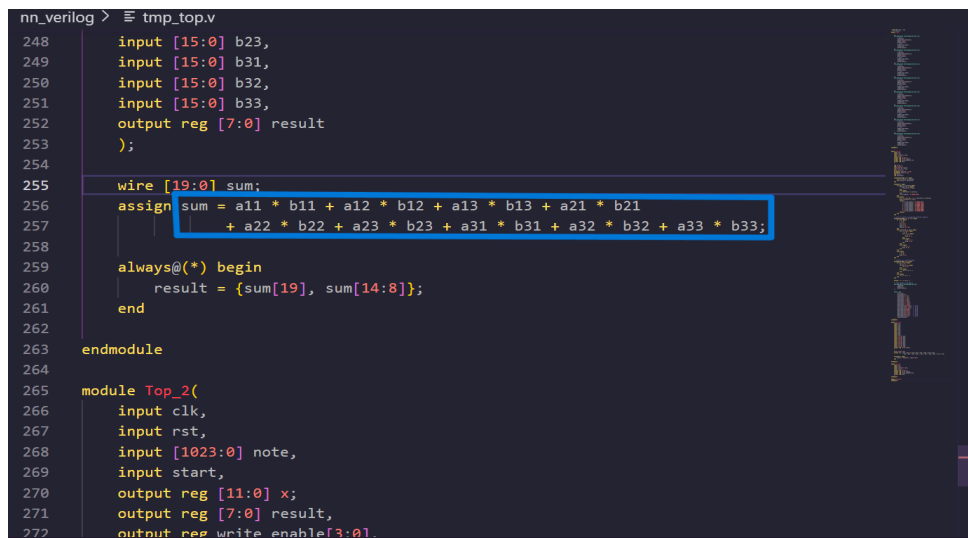


▲Fig. 1-3 A fail implementation of CNN

2. Computation Time

After we face the memory problem, we start all over and try another method. We even change the structure of our model and input size. We remove the 3-layer Convolutional Neural Network and reshape our input from 64*64 to 32*32. This time our memory is within the range. However, our code can not be synthesized since we try to compute too many operations in a single clock. An indicator in Vivado called “Time Slack” shows that we have about -2000, slack can be considered the debt of your program, instead of lacking money, it is lacking time. In other words, our wire and registers may not have the expected value since the computation is not finished yet. We face this problem and we consider it a bit too hard to solve, since the deadline is around the corner.

The following figure is an example of why we face the computation time problem (Fig. 1-4). We asked the other group who also do deep learning, they compute one addition and two multiplications in a single clock. No wonder our implementation does not work, since we want to perform nine multiplications and eight additions within a clock, which is impossible.



```

nn_verilog > tmp_top.v
248     input [15:0] b23,
249     input [15:0] b31,
250     input [15:0] b32,
251     input [15:0] b33,
252     output reg [7:0] result
253 );
254
255 wire [19:0] sum;
256 assign sum = a11 * b11 + a12 * b12 + a13 * b13 + a21 * b21
257            + a22 * b22 + a23 * b23 + a31 * b31 + a32 * b32 + a33 * b33;
258
259 always@(*) begin
260     result = {sum[19], sum[14:8]};
261 end
262
263 endmodule
264
265 module Top_2(
266     input clk,
267     input rst,
268     input [1023:0] note,
269     input start,
270     output reg [11:0] x;
271     output reg [7:0] result,
272     output reg write_enable[3:0];

```

▲ Fig. 1-4 One example of our computation time problem

III. Top-down view of our game

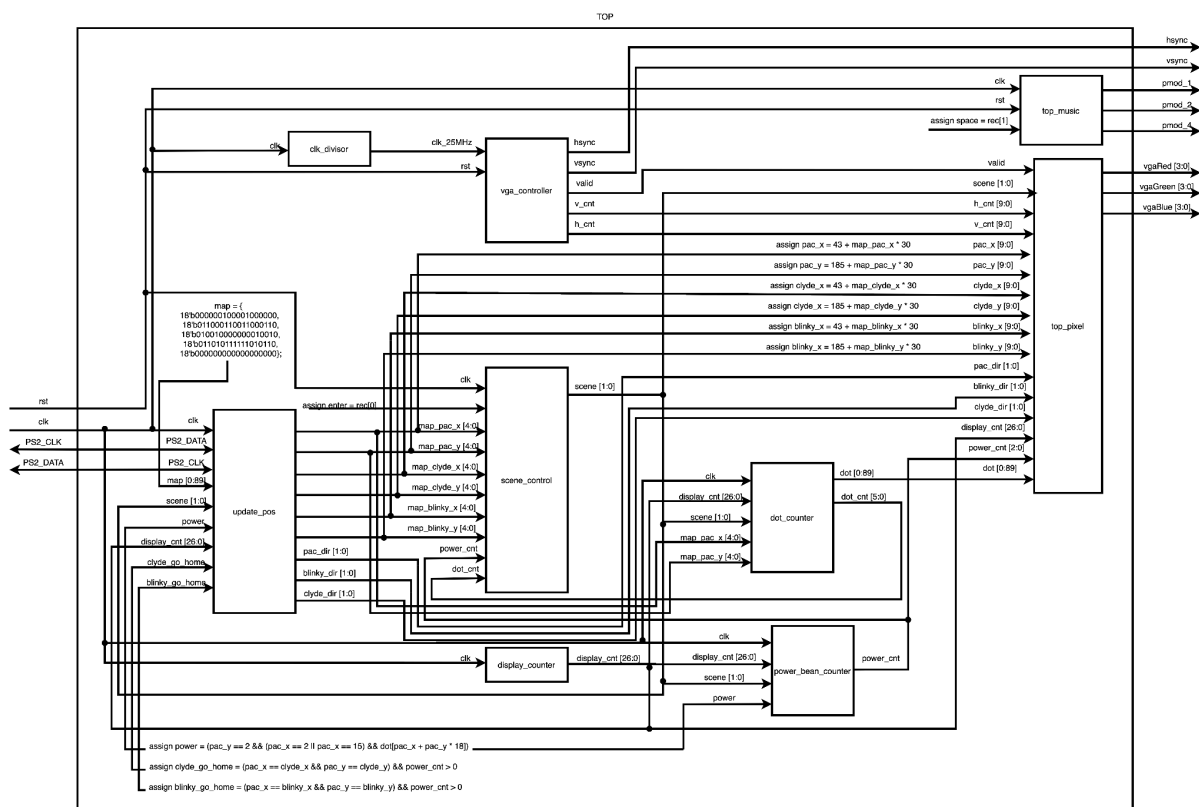
Since there are numerous modules in the **Top** module (Fig. 2-1), to improve the readability of the block diagram for the top module, we group modules responsible for controlling the directions and positions of pacman, clyde, and blinky an **update_pos** module (Fig. 2-2). We also group the memory address

generator, block memory generator, and pixel generator into a **top_pixel** module (Fig. 2-3).

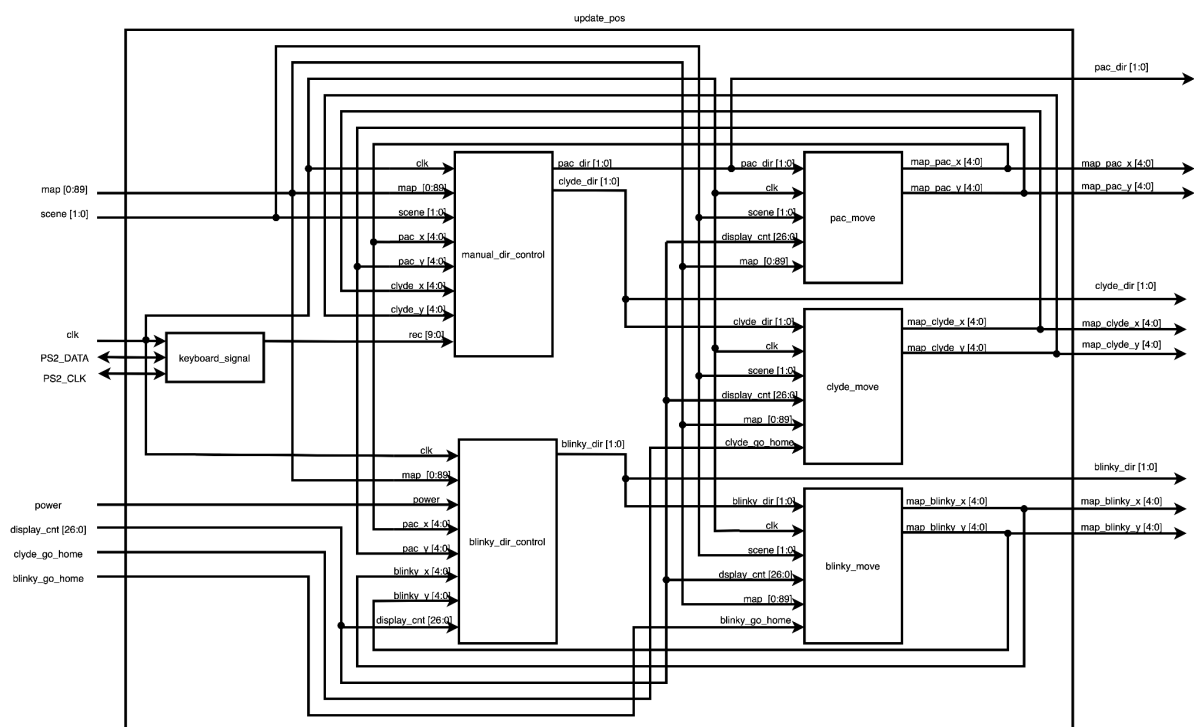
At the beginning of the game, the scene control will wait for the pressing of the "enter" key to start the game. Once the game begins, the **update_pos** module will continuously receive keyboard signals and update the directions and positions of pacman, clyde, and blinky accordingly. The updated information will be output to other modules to determine whether dots or power beans are consumed, and whether pacman or ghosts has won the game.

The **top_pixel** module will decide the output patterns of individual pixels on the screen based on the positions of pacman, clyde, and blinky, as well as whether pacman has consumed a power bean and whether the game has started.

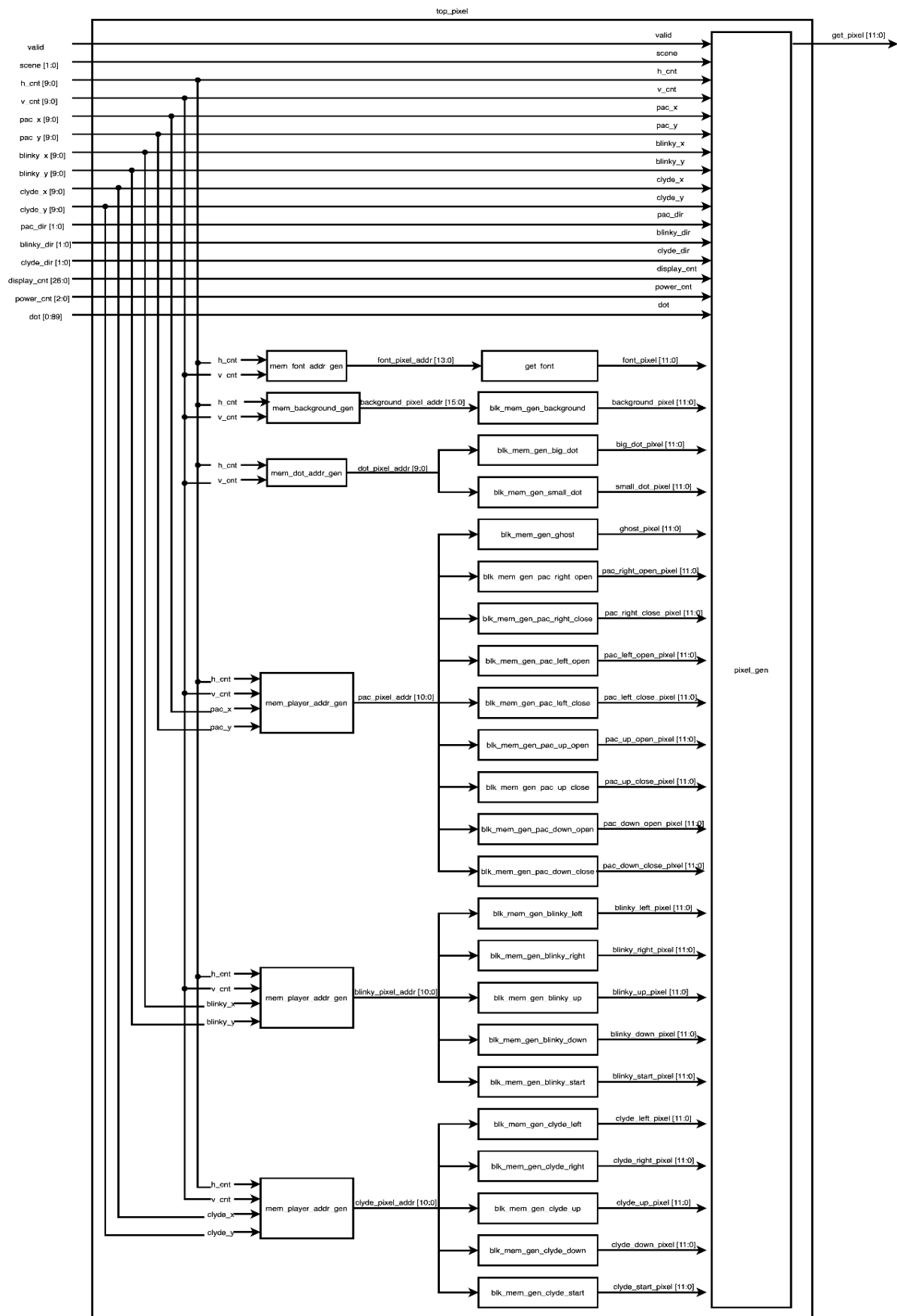
The **top_music** module will continuously output background music, and if the space key is pressed, it will either mute or unmute the audio.



▲Fig. 2-1 The block diagram of TOP



▲Fig. 2-2 The block diagram of **update_pos**



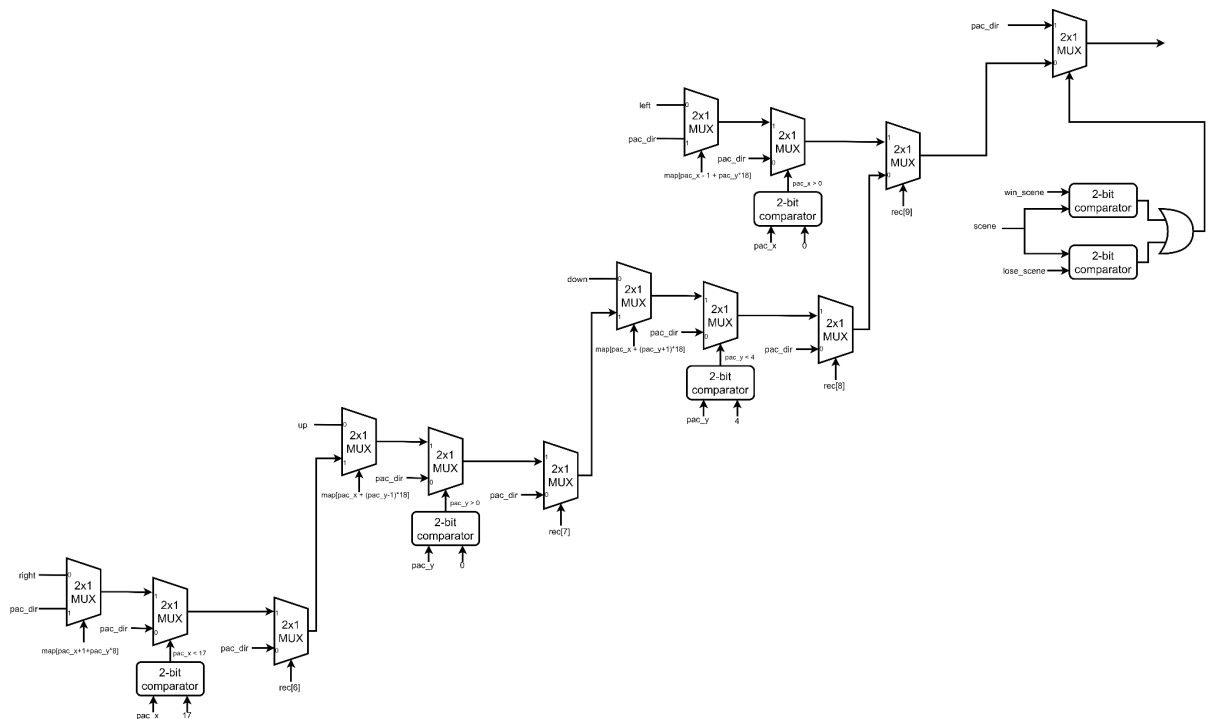
▲ Fig. 2-3 The block diagram of `top_pixel`

IV. Features

Since introducing each module without a specific order will be incomprehensible, we decide to discuss our modules by features. And we will put other non-feature modules in the last part.

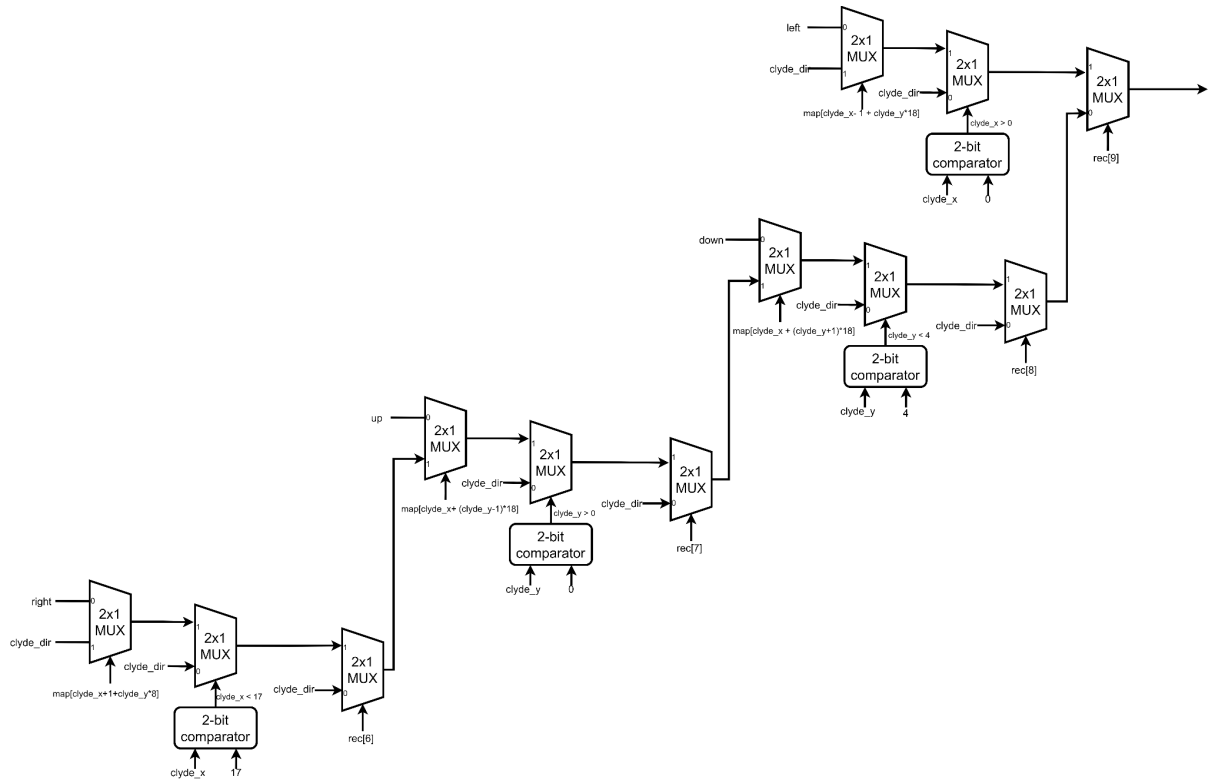
1. P2P

One of the players, assumed player 1, will control the pacman. And try every best he or she can to eat all the dots. As for ghosts, the first ghost is driven by our own-customized algorithm, the other is manually controlled by the other player, assumed player 2. The following figure shows the block diagram of the next direction of pacman (Fig. 3-1).



▲Fig. 3-1 The block diagram of **next_pac_dir**

Next, the next figure demonstrates the block diagram of the next direction of the manually-controlled ghost (Fig. 3-2).



▲ Fig. 3-2 The block diagram of **next_clyde_dir**

We update the **pac_dir** and **clyde_dir** every clock cycle, and the default direction of pacman is going upwards.

2. Animation

To show the animation on the screen, we first load the images we need using a block memory generator. We get their pixel address by using the four modules below, which are **mem_background_addr_gen**, **mem_dot_addr_gen**, **mem_font_addr_gen** and **mem_player_addr_gen**.

mem_background_addr_gen:

Since the top and bottom of the picture are both black, we reduce the size of the image to 320*170 (it will be enlarged by two times on the screen), and manually fill in the black pixels. Hence the top-left corner of the background image is (0, 30), and the right-bottom corner of the background image is (639, 369), and the formula to obtain the value of **background_pixel_addr** can be expressed as follows:

```
background_pixel_addr = ((h_cnt >> 1) + 320 * ((v_cnt - 30) >> 1)) % 54400;
```

mem_dot_addr_gen:

Since there are many dots on the map, and the dot at the top_left corner starts from (49, 191), the formula to obtain the value of dot_pixel_addr can be expressed as follows:

```
dot_pixel_addr = ((h_cnt - 49) % 30 + ((v_cnt - 191) % 30) * 30) % 900;
```

mem_font_addr_gen:

Since the top-left corner of the font image is (160, 390), and the right-bottom corner of the font image is (479, 439), the image has a size of 320 pixels in width and 50 pixels in height. Hence the formula to obtain the value of font_pixel_addr can be expressed as follows:

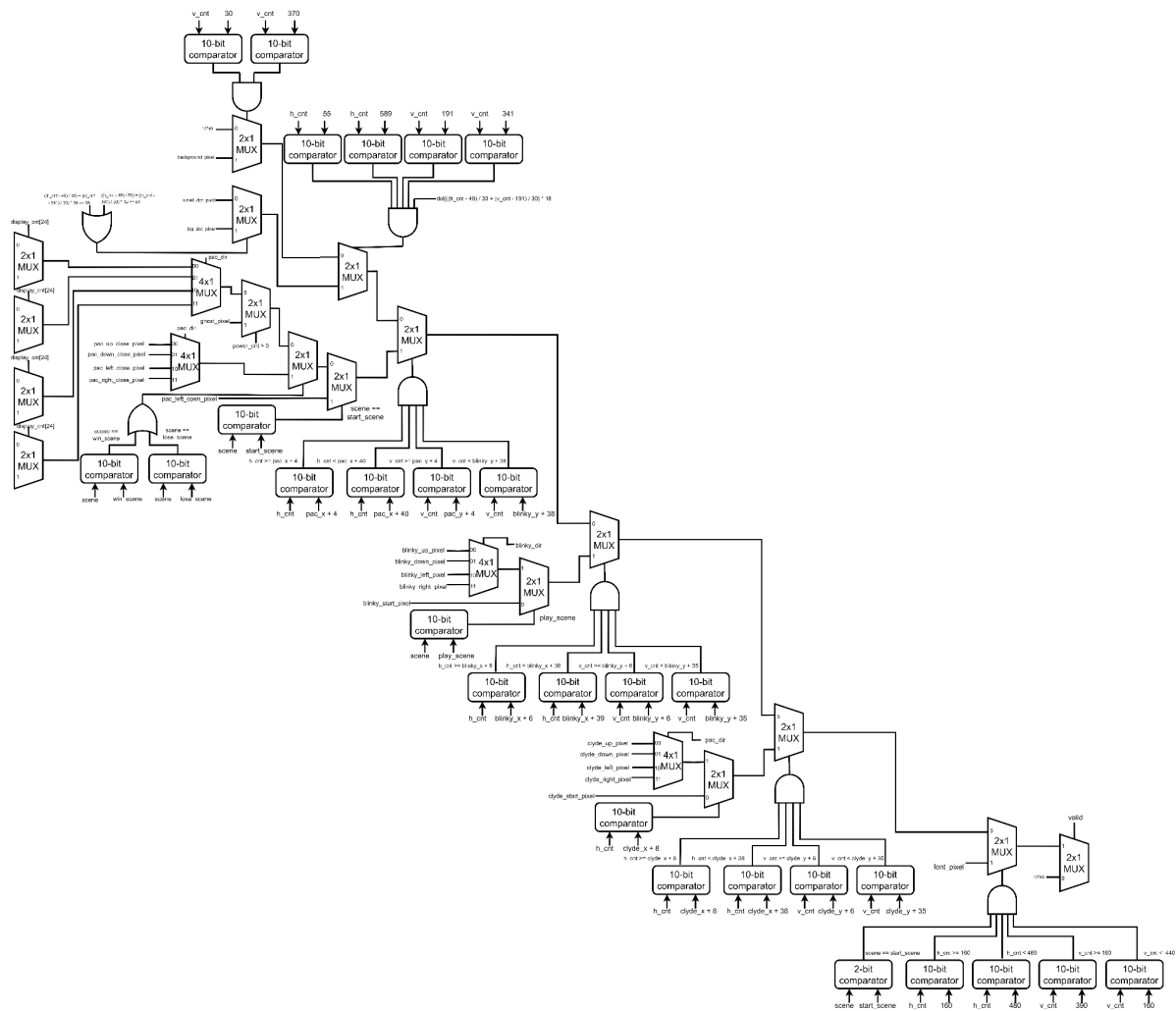
```
font_pixel_addr = ((h_cnt - 160) + (v_cnt - 390) * 320) % 16000;
```

mem_player_addr_gen:

Since pacman, clyde and blinky keep moving on the map, we need to get their pixel coordinates of their top-left corner. After we get there coordinates, and we know the size of there images are 42*42, we can obtain the value of there pixel_addr, which is named player_pixel_addr in the mem_player_addr_gen as follows:

```
player_pixel_addr = ((h_cnt - x) % 42 + ((v_cnt - y) % 42) * 42) % 1764;
```

Our pacman needs 8 images to display its animation, while each of our ghosts need 5 images. In total we need 18 images. After we get the pixel of all the images we load by the block memory generator, we use **pixel_gen** as the module to determine the pixel that we want in a specific place. The figure below shows the block diagram of **pixel_gen** (Fig. 3-3).

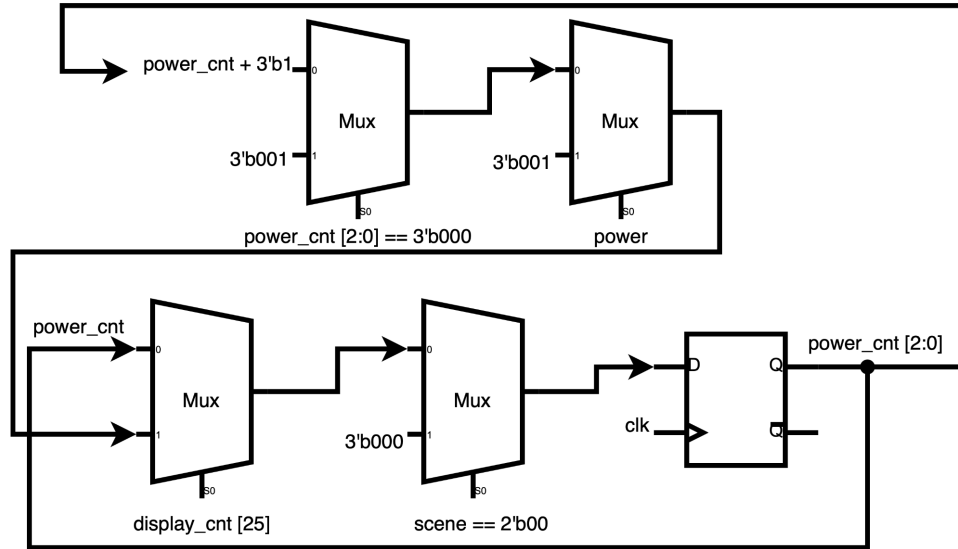


▲ Fig. 3-3 The block diagram of pixel_gen

In the above figure, we can see that **pixel_gen** is composed of a series of if-else statements. This kind of software coding style may cause problems when synthesizing, since it is not friendly to hardware. In our case, when we are about to finish our project, we indeed face some synthesis failed with no error message. We conclude that is because we use too many nested if-else. Therefore, it is better to “flatten” the if-else statements when implementing.

3. Power Bean

The power_counter module helps us to record if the special effect of the power bean still continues or not. By checking the value of power_cnt, if power_cnt is equal to 0, then the special effect has ended; otherwise, the special effect is still active (Fig. 3-4).

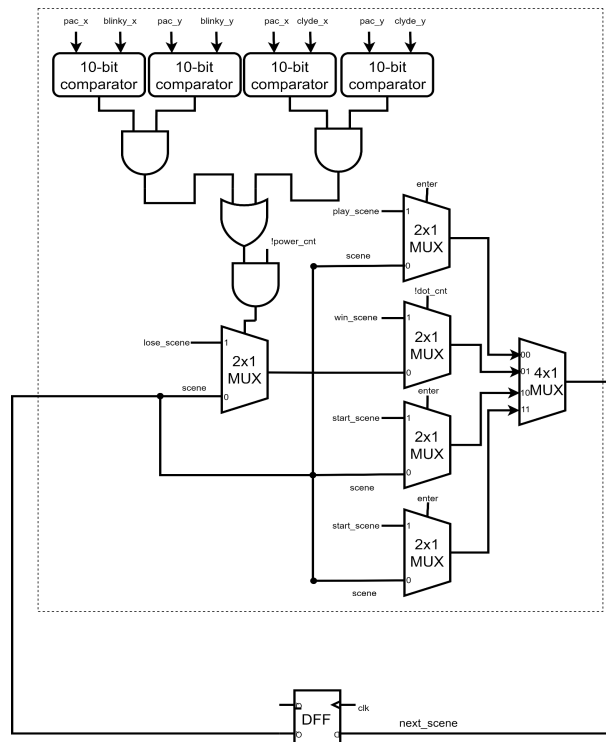


▲ Fig. 3-4 The block diagram of **power_counter**

4. Others

The following modules are considered irrelevant to the features, including scene control, the algorithm of ghost, moving of pacman and ghosts.

a. Scene Control

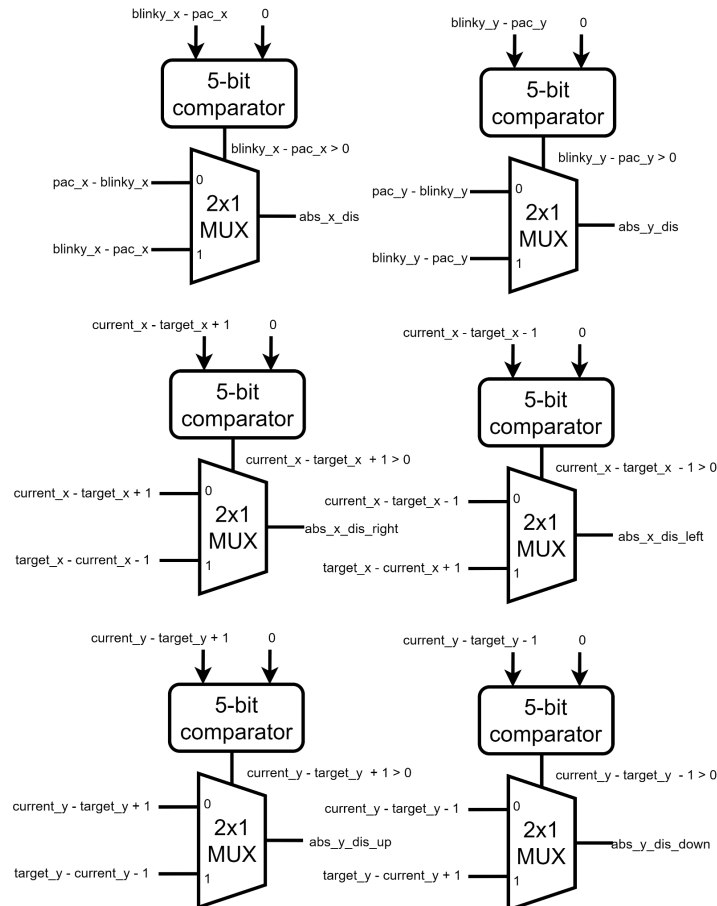


▲ Fig. 3-5 The block diagram of `scene_control`

In the above figure (Fig. 3-5), the combinational part was marked inside the dotted line. We use the number of dots, collision, and whether the enter key is pressed to check which scene should be displayed. The scene will change from start to play when enter is pressed. In the play scene, when all the dots are eaten by pacman, pacman wins the game. However, before pacman eats all the dots, if he touches the ghost without power bean's special effect, pacman loses the game, that is, the ghosts win the game. After the game ends, press enter again to go back to the start scene.

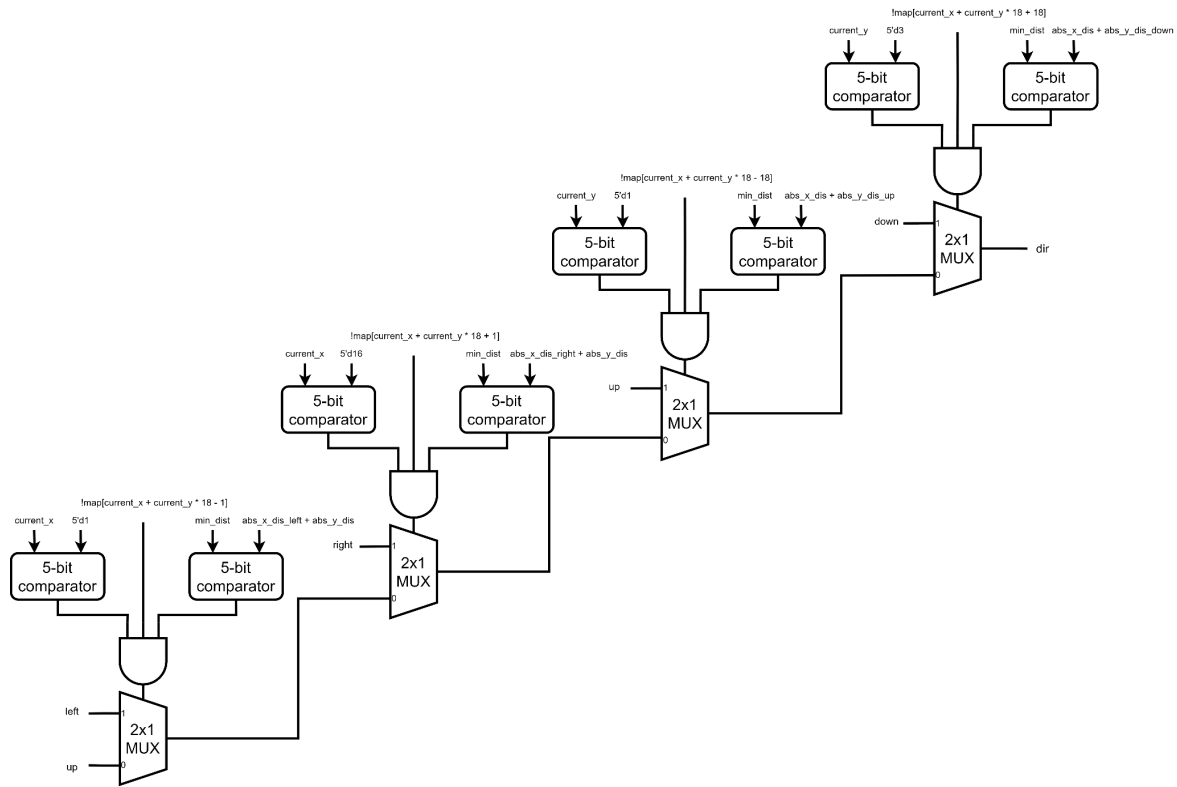
b. Algorithm of ghost

The following figures (Fig. 3-6, Fig. 3-7) first show how we get the shortest distance, and then will show how we design the algorithm of the ghost. One of our ghosts, blinky, will chase pacman if its distance with pacman is larger than 4. Or else it will hide in its home, and then resume to chase pacman.



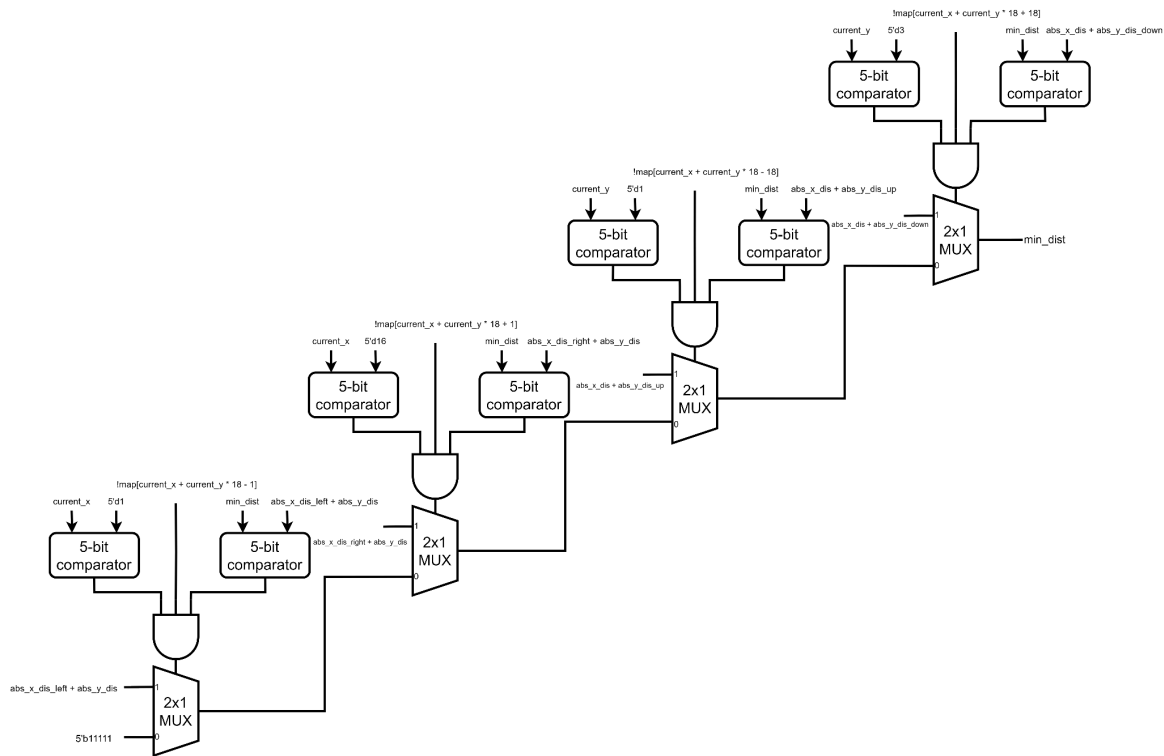
▲ Fig. 3-6 The block diagram of wire required in **shortest_path**

The above figure (Fig. 3-6) shows how to derive the distance (always ≥ 0), we now use them to determine the next direction, which is closest to the target (Fig. 3-7).



▲ Fig. 3-7 The block diagram of **dir** in module **shortest_path**

Then, we determine the minimum distance (**min_dist**) which is closest to the target (Fig. 3-8).

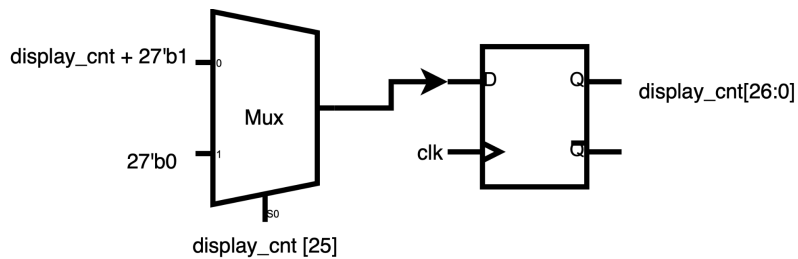


▲ Fig. 3-8 The block diagram of **min_dist** in module **shortest_path**

In the above figures, we use a series of if-else statements to update the direction and the minimum distance, since we have not thought of any other ways to implement this part.

We then use these modules to implement the controller of blinky, the block diagram of module **blinky_dir_control** will be mentioned in the next part.

c. Moving of pacman and ghosts

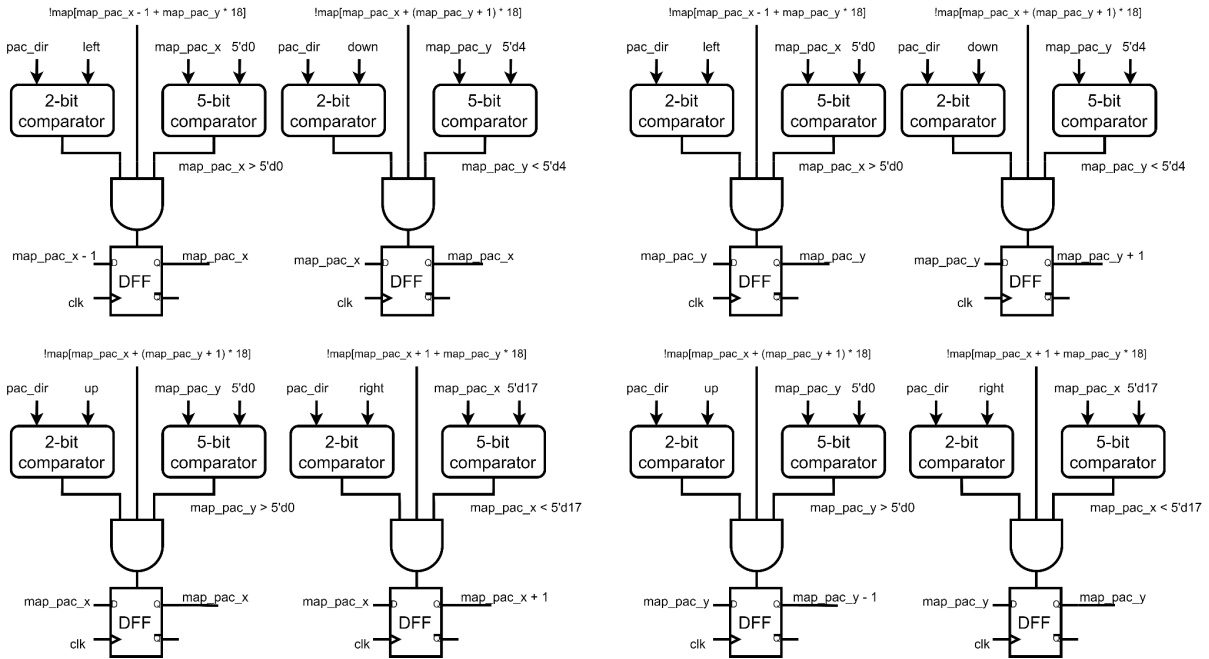


▲ Fig. 3-9 The block diagram of **display_counter**

We first introduce the **display_counter** (Fig. 3-9) module, which helps us to record the time for Pacman, Clyde, and Blinky to move

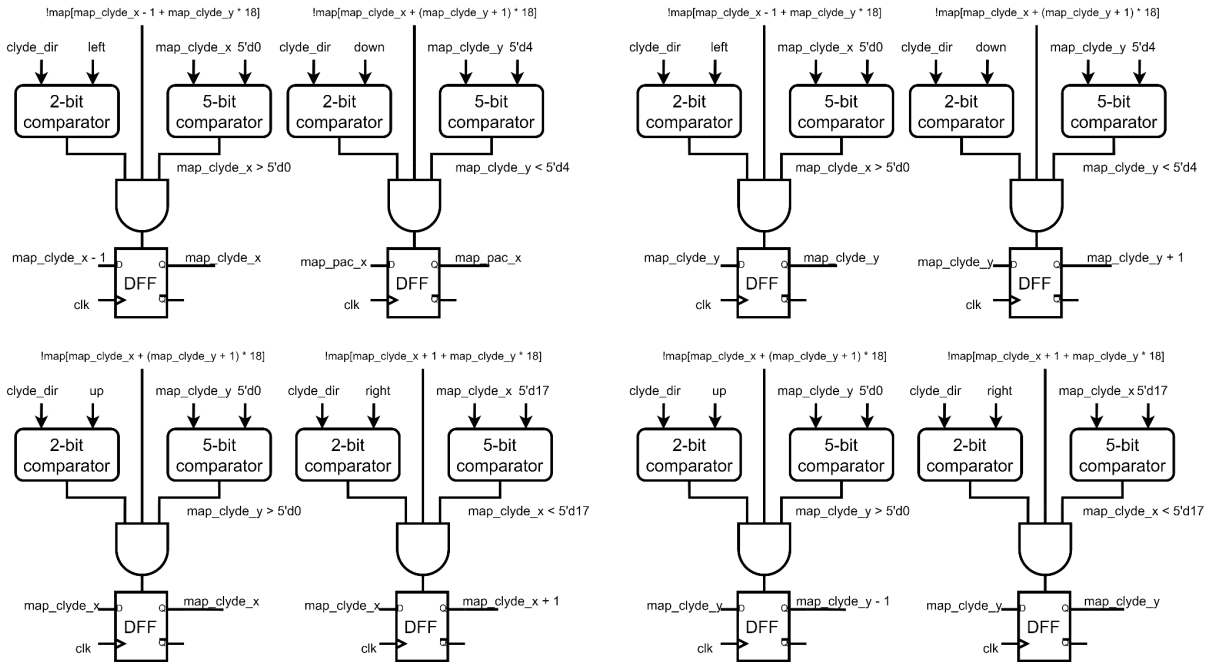
to the next position. It also keeps track of whether Pacman's mouth should be open or closed. By recording the time for the next movement, the output of **display_counter**, which is named **display_cnt**, can be input into the **power_bean_counter** module, helping us record the remaining time of the power bean's special effect after Pacman consumes it.

With the **display_counter**, we can update the pacman's location, which is determined by its direction and the accessibility of the next block it is going to. The following figure illustrates the block diagram of the moving of pacman when **play_scene** (Fig. 3-10). We use the D Flip-flop with an enable signal to present the block diagram.

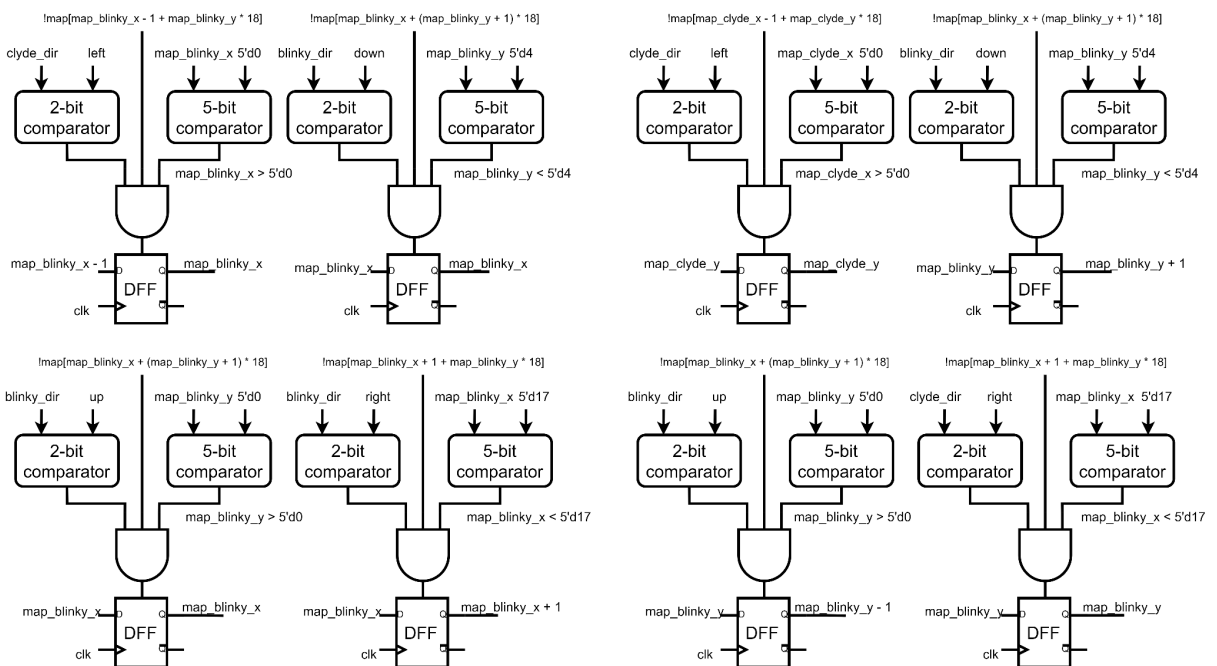


▲ Fig. 3-10 The block diagram of the location of pacman

Next, we are going to talk about the moving of two ghosts. The structure of the block diagram looks exactly the same with Fig. 3-10, except for the name of the register and the initial location (Fig. 3-11, Fig. 3-12).

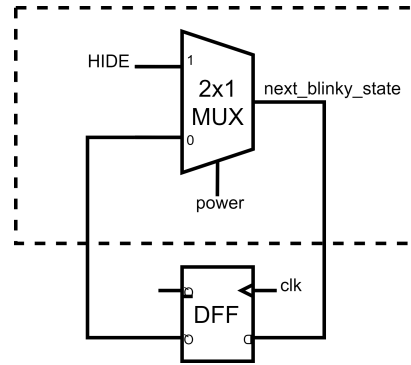


▲Fig. 3-11 The block diagram of the location of Clyde

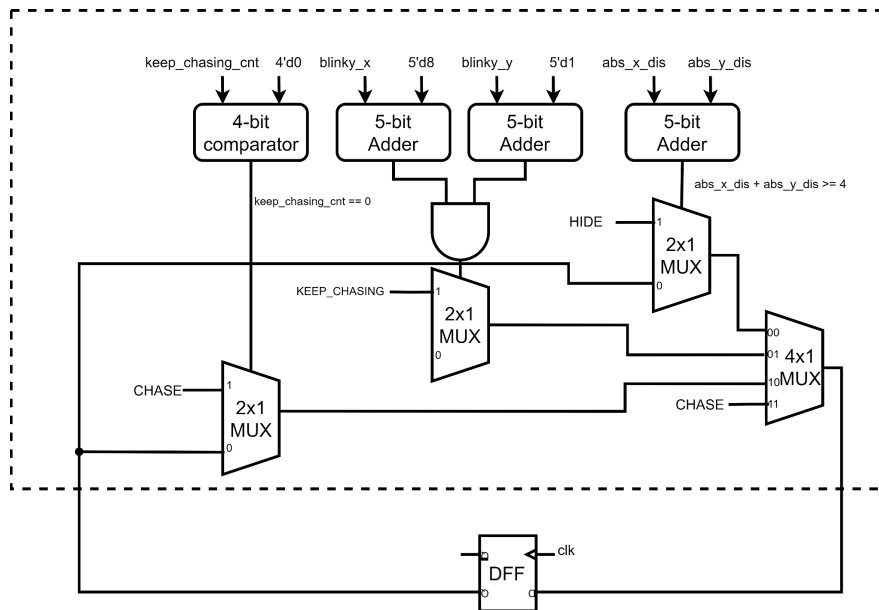


▲Fig. 3-12 The block diagram of the location of Blinky

Let us discuss how we control the Blinky, we first show we decide the state of Blinky, it is shown in Fig. 3-13 and Fig. 3-14.

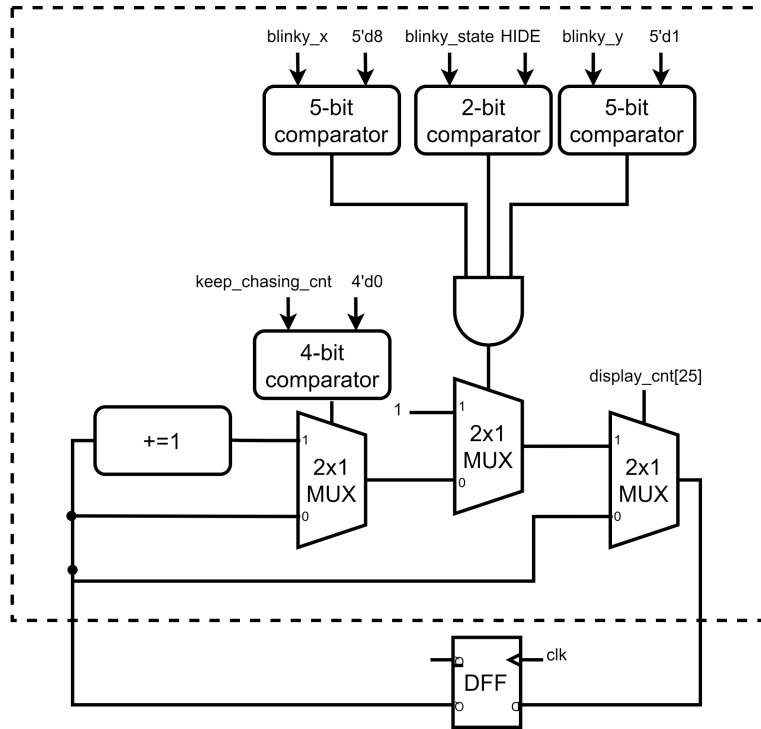


▲ Fig. 3-13 The block diagram of **blinky_state**



▲ Fig. 3-14 The block diagram of **next_blinky_state**

Also, we add an additional counter **keep_chasing_cnt**(Fig. 3-15) for Blinky, so she will keep chasing pacman in the next 16 moves, no matter how close the distance between pacman and she.



▲ Fig. 3-15 The block diagram for **keep_chasing_cnt**

V. Experimental Results

The method we implemented involves testing each time a new feature is added or one or two images are included. We started by adding the background, then introduced Pacman, allowing Pacman to freely move within the maze using keyboard inputs, ensuring it doesn't pass through walls or go beyond the maze boundaries. Next, we added Clyde and Blinky, implementing a simple module that generates random directions to ensure Clyde and Blinky also stay within the maze.

Following that, we introduced dots and implemented transitions, testing whether the game ends when Pacman consumes all the dots or encounters a ghost. We then added background music and introduced power beans, testing the duration of the power beans' effects and ensuring that Pacman's encounter with a ghost doesn't end the game but instead resets the ghost.

Throughout the process, we continuously modified the algorithm governing the ghost's pursuit of Pacman. Finally, we developed an algorithm that doesn't exceed board resource limits and meets our expectations, completing our code.

VI. Work Distribution

111062122 李彥呈:

Code:

Algorithm of the ghost, Music

Report:

blinky_dir_control.v, blinky_move.v, clyde_move.v,
manual_dir_control.v, pac_move.v, pixel_gen.v,
scene_control.v, shortest_path.v

111062137 陳啟綸:

Code:

Animation, Everything about block memory and Display

Report:

Top.v, mem_background_addr_gen.v, mem_dot_addr_gen.v,
mem_font_addr_gen.v, mem_player_addr_gen.v,
power_bean_counter.v, dot_counter.v, display_counter.v

VII. Epilogue

In a nutshell, we learned a lot in this project. Although we failed to finish our original topic, we made a new one and we consider the result to be kind of acceptable. However, our project still can be improved in various ways. For example, the number of Ips can be lessened. In our

project, we use more than 20 Ips and there's only $\frac{1}{4}$ left in block memory. My friend from another team suggests that we can put similar images (e.g. images which involve pacman and ghost animation) together as one Ip to save some space and for better management.

As for the original project that we deserted, we think that after we learned some advanced courses related to deep learning or hardware. Maybe we will have a better idea on how to deal with the difficulties that we have encountered right now. We sincerely hope that we can restart and finish the project someday!