FPGA Final Project: Arkanoid

111062134 游竣閔

111062332 朱誼學

Table Of Content:

Introduction:

Arkanoid is a game which the player controls a paddle, and uses the paddle to bounce back the ball to destroy the block above. There are many versions of Arkanoid, but the one we've made is simply controlling the paddle and destroying the block by bouncing the ball to gain score. Also, usually, Arkanoid is played vertically, but with our version, it is played horizontally.

Motivation:

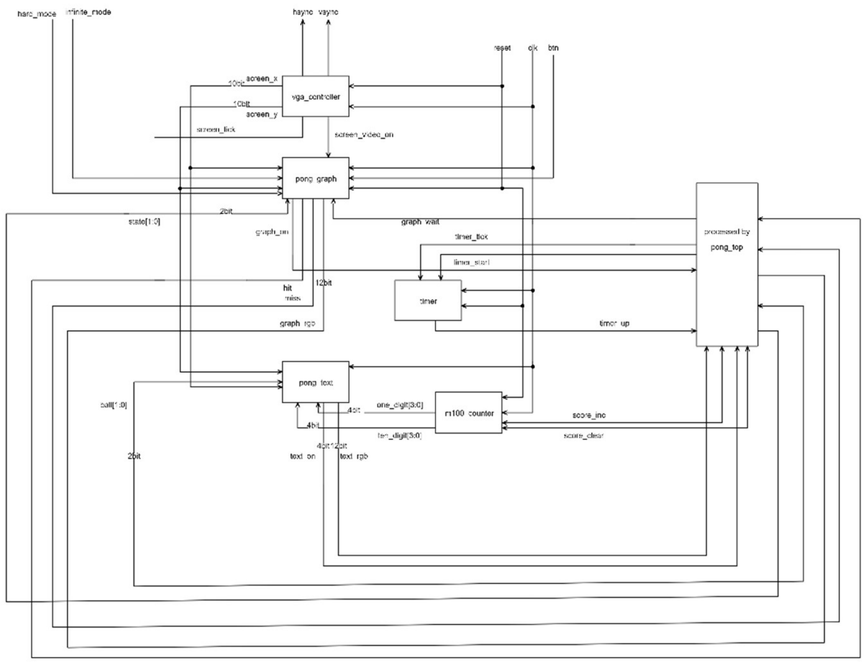
We used to play video game together last year when we're bored in the dorm, and we both love retro-styled items. We've played lots of old-styled video games like Donkey Kong, Bomberman, Pacman… etc. We even think that those simple games are way more entertaining than some of the video games made nowadays although the quality of them may not be as good as games nowadays.

When we first heard of this final project, to be honest, at first, we want to make it as simple as possible. But after thinking for a while, we decided to make an intermediate-level video game, which is not too easy, but also not too hard for us to create. After doing some research and having deeper understanding for Verilog video game structure and VGA mechanisms, this project is born and done, one of our favorite games—Arkanoid!

Implementation:

(ascii_rom.v(data for text display) and **vga_controller** are directly downloaded from internet without modifying, source is mentioned in the reference)

pong_top:



Input: clk, reset, btn[1:0], infinite_mode, hard_mode,
Output: hsync, vsync, [11:0]rgb
Wire: screen_x[9:0], screen_y[9:0], screen_video_on, screen_tick, graph_on, hit, miss, text_on[3:0], graph_rgb[11:0], text_rgb[11:0], one_digit[3:0], ten_digit[3:0], timer_tick, timer_up
Reg: [1:0]state_reg, [1:0]state_next, [11:0]rgb_reg, [11:0]rgb_next, graph_wait, score_inc, score_clear, timer_start, [1:0]ball_reg, [1:0]ball_next
Parameter: newgame = 2'b00, play = 2'b01, newball = 2'b10, over = 2'b11

<u>How every signal works?</u>

Talking about **timer_tick** signal, it will equal to 1 if both screen_x and screen_y equals to 0, else, timer_tick equals to 0, which means that it is a tick which its time interval equals to screen refresh rate.

Inside the first always block triggered by **posedge clk** or **posedge reset** , if **reset** equals to 1, then **state_reg** will be set as **new_game**'s value. Else, if **screen_tick** equals to 1, **rgb_reg** will be set as **rgb_next's** new value. **state_reg's** value will be set as **state_next's** new value, and that **ball_reg's** value will be set as **ball_next's** value.

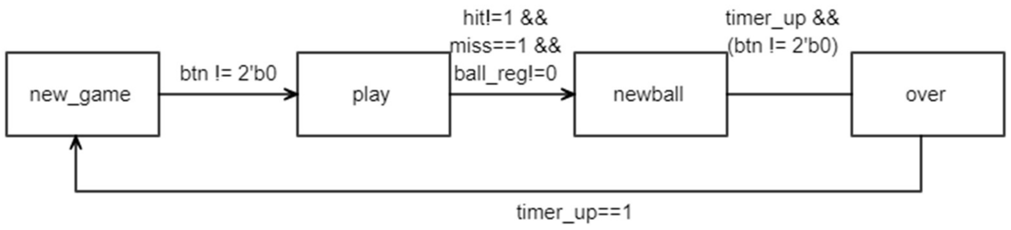
In the first combinational always block, **graph_wait, timer_start, score_inc, score_clear** will be set as 0(setting as default value to prevent error), and **state_next** will be set as **state_reg's** value, **ball_next** will be set as **ball_reg's** value before the case statement. In the case statement using **state_reg's** value to determine which block to execute:

> If **state_reg** equals to **newgame, ball_next** will be set as 2'b11(ball number equals to 3), and **score_clear** will be set as 1'b1 to clear the score. If **btn** isn't equal to 2'b0(button pressed), then **state_next** will be set as **play's** value, and that **ball_next** will be set as **ball_reg's** value minus by 1(A ball is launched).
>
> Else if **state_reg** equals to **play**, **graph_wait** will be set as 1'b0, and that if **hit** (ball touched either the paddle or a block) equals to 1, then **score_inc** will be set as 1(add one point). Else if **miss** equals to 1 (miss one ball),if **ball_reg** equals to 0, then **state_next** will be set as **over**(no more ball to launch anymore), else **state_next** equals to **newball. timer_start** will minus by 1, and that **ball_next** will equal to **ball_reg** -1.
>
> Else if **state_reg** equals to **newball**, if **timer_up** and **btn** isn't equal to 0 ,then **state_next** equals to **play.**
>
> Else if **state_reg** equals to **over**, if **timer_up** equals to 1, then **state_next** equals to **newgame**.

In the second combinational always block (which is designed for filling screen color for each area), if **screen_video_on** is equal to 0 ,then fill nothing (**rgb_next=12'h0)**. Else, if **graph_on** equals to 1(**graph_on** equals to 1 if any region on the graph except for text and background that needs specific color is visited) , **rgb_next** equals to **graph_rgb**. Else if (**text_on[3]** or **text_on[2]** equals to 1) or (**state_reg** equals to **newgame** and that **text_on[1]** equals to 1), or **state_reg** equals to **over** and that **text_on[0]** equals to 1, **rgb_next** will be set as **text_rgb**. Else, **rgb_next** will be set as **graph_rgb**.(**text_on** = {**score_on**, **PONG_on**, **rule_on**, **over_on**}, which indicates that the region being visited in the presence will be set as 1, else, set as 0).

Last but not least, assign **rgb** as **rgb_reg.**

__**State diagram for this module:**__



Due to the fact that the picture will be too crowded if I write every signal's value in each state into the picture, so I didn't write it inside the graph. I've explained every signal's value change in the above explanation section.

pong_graph:

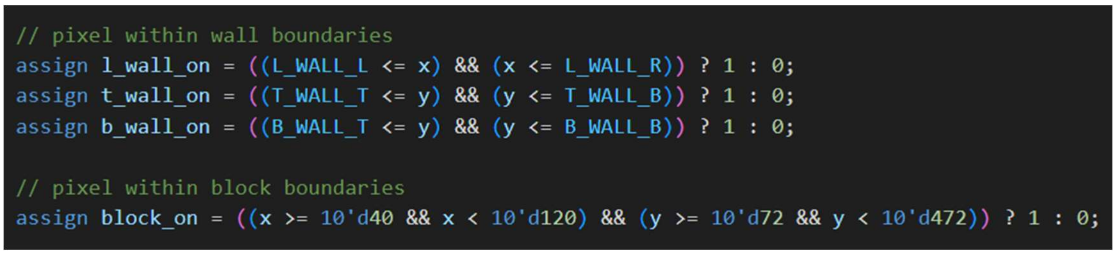
Input : clk, reset, btn[1:0], graph_wait, video_on, x[9:0], y[9:0], state[1:0]
Output : graph_on, hit, miss, graph_rgb[11:0]

__**Design explanation :**__

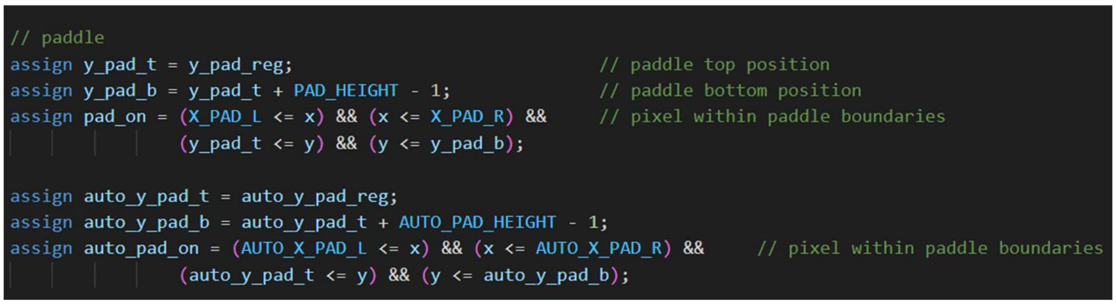
This module handles the graph part including object, animation and background. The main working strategy is to output the corresponding color depending on which 'on' signal is 1. The 'on' signal is mostly determined by the input **x[9:0]** and **y[9:0]**. Then, the multiplexer will choose the proper output with the on signal.

Take the paddle for example, when (paddle left boundary) < x < (paddle right boundary) and (paddle up boundary) < y < (paddle down boundary) the **pad_on** signal will be 1, simultaneously, the output **graph_rgb** will get the value of **pad_rgb**.

That is, for each on signal, we can set them as below.

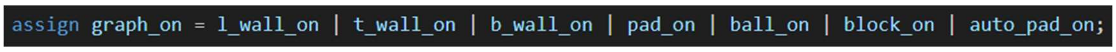


```
// pixel within wall boundaries
assign l_wall_on = ((L_WALL_L <= x) && (x <= L_WALL_R)) ? 1 : 0;
assign t_wall_on = ((T_WALL_T <= y) && (y <= T_WALL_B)) ? 1 : 0;
assign b_wall_on = ((B_WALL_T <= y) && (y <= B_WALL_B)) ? 1 : 0;

// pixel within block boundaries
assign block_on = ((x >= 10'd40 && x < 10'd120) && (y >= 10'd72 && y < 10'd472)) ? 1 : 0;
```



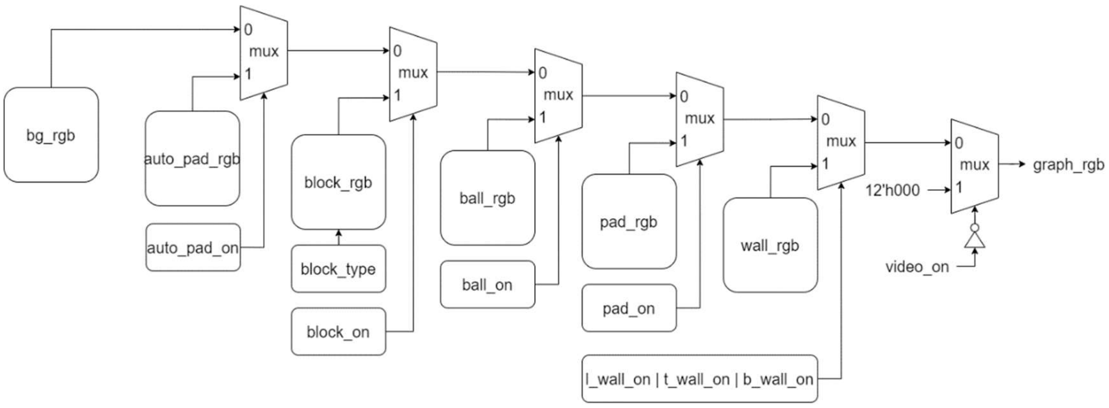
```
// paddle
assign y_pad_t = y_pad_reg;                              // paddle top position
assign y_pad_b = y_pad_t + PAD_HEIGHT - 1;               // paddle bottom position
assign pad_on = (X_PAD_L <= x) && (x <= X_PAD_R) &&      // pixel within paddle boundaries
                (y_pad_t <= y) && (y <= y_pad_b);

assign auto_y_pad_t = auto_y_pad_reg;
assign auto_y_pad_b = auto_y_pad_t + AUTO_PAD_HEIGHT - 1;
assign auto_pad_on = (AUTO_X_PAD_L <= x) && (x <= AUTO_X_PAD_R) &&      // pixel within paddle boundaries
                (auto_y_pad_t <= y) && (y <= auto_y_pad_b);
```

Besides, **graph_on** is an output to the top module, which is used to verify which part (text or graph) should be handle at the time.

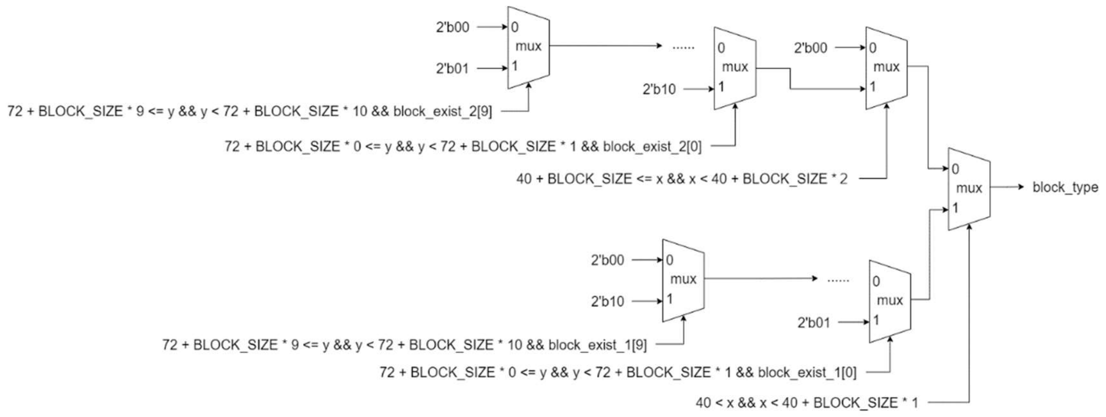


```
assign graph_on = l_wall_on | t_wall_on | b_wall_on | pad_on | ball_on | block_on | auto_pad_on;
```

The top diagram can be shown as below.

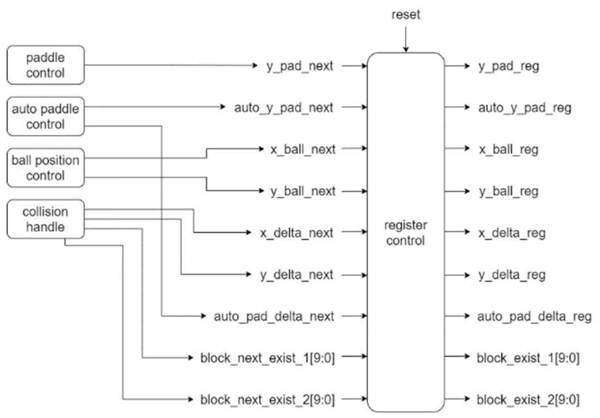


For **block_type**, we first determine which column is the current block belongs to, and next determine the row by the vertical position. Then set the value of **block_type** to 01 or 10 (represents type 1 and 2) alternately.

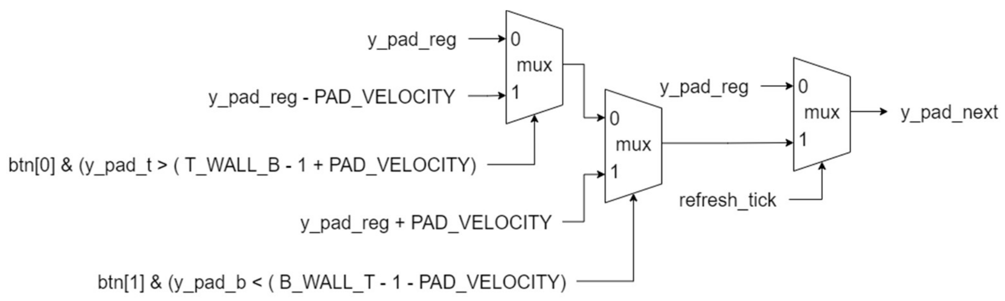


2'b00
2'b01
mux
0
1
72 + BLOCK_SIZE * 9 <= y && y < 72 + BLOCK_SIZE * 10 && block_exist_2[9]
......
2'b10
72 + BLOCK_SIZE * 0 <= y && y < 72 + BLOCK_SIZE * 1 && block_exist_2[0]
2'b00
40 + BLOCK_SIZE <= x && x < 40 + BLOCK_SIZE * 2
block_type
2'b00
2'b10
72 + BLOCK_SIZE * 9 <= y && y < 72 + BLOCK_SIZE * 10 && block_exist_1[9]
2'b01
72 + BLOCK_SIZE * 0 <= y && y < 72 + BLOCK_SIZE * 1 && block_exist_1[0]
40 < x && x < 40 + BLOCK_SIZE * 1

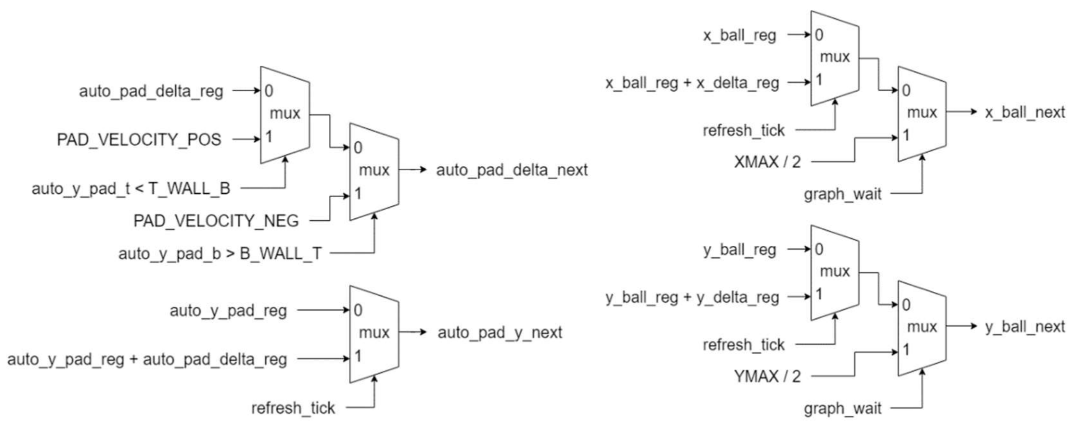
For the register signals used in on signal's assignment, we update them with next signals, and the value will be updated by **clk**.



For auto paddle in the middle of the screen, we have a block called **auto paddle controller**, covers the paddle's y register and delta value, so as normal paddle and the ball. The difference is that auto paddle changes position with refresh tick automatically, and paddle changes position when the buttons are pressed, ball changes its direction when a collision occurs. Following are the block diagrams.



y_pad_reg
y_pad_reg - PAD_VELOCITY
btn[0] & (y_pad_t > (T_WALL_B - 1 + PAD_VELOCITY)
mux
0
1
y_pad_reg + PAD_VELOCITY
btn[1] & (y_pad_b < (B_WALL_T - 1 - PAD_VELOCITY)
y_pad_reg
refresh_tick
y_pad_next



pong_text

Input : clk, ball[1:0], one_digit[3:0], ten_digit[3:0], x[9:0], y[9:0]
Output : text_on[3:0], text_rgb[11:0]

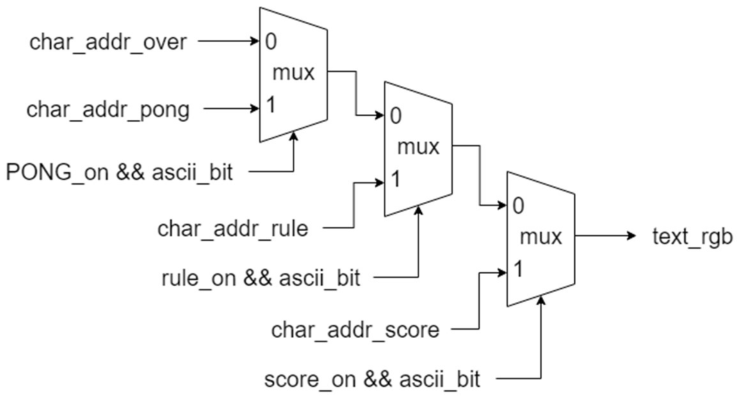
Design explanation :

In this module, what we have to do is similar to the things we've done in **pong_graph**, the difference is that we don't need to handle animation in this module, the only this we do is to set the corresponding value into registers, so when x and y value reach the position where the text should be shown, we can output the color properly. For the pattern of the words, we get the value from **ascii_rom** module, which encodes all the 128 ascii characters, each of them are 8 * 16 bits.

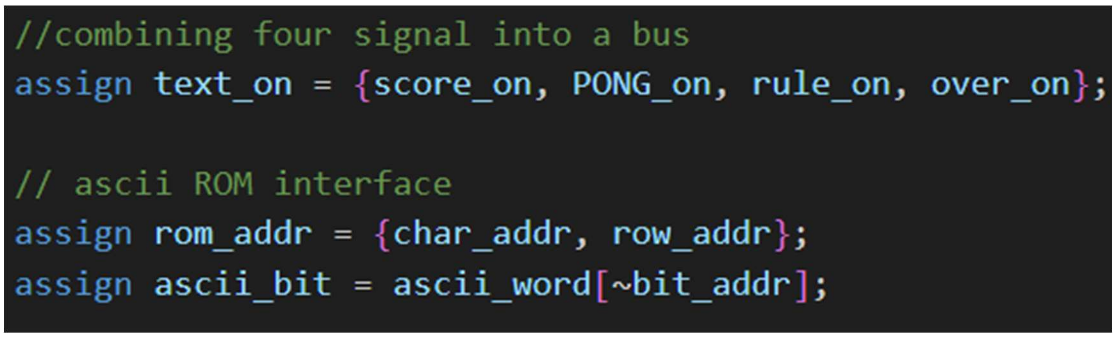
As for the top diagram, a multiplexer will choose which **rgb** signal should be outputted depending on the on signals as well. The on signals are defined as below.

assign PONG_on = (y[9:7] == 2) && (3 <= x[9:6]) && (x[9:6] <= 6);
assign score_on = (y >= 32) && (y < 64) && (x[9:0] >= 48) && (x[9:0] < 255 + 48);
assign rule_on = (x[9:7] == 2) && (y[9:6] == 2);
assign over_on = (y[9:6] == 3) && (5 <= x[9:5]) && (x[9:5] <= 13);

And this is the top diagram.

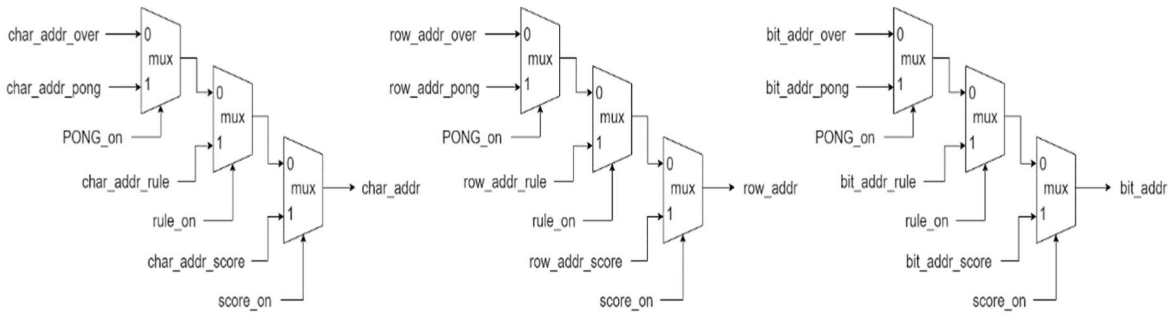


Moreover, the below code tell that **text_on** is outputted to the top module. As for **rom_addr, char_addr** will determine the address of a character, **row_addr** and **bit_addr** shows which row and column should be handle now. (Each character has 16 rows and 8 column of data).

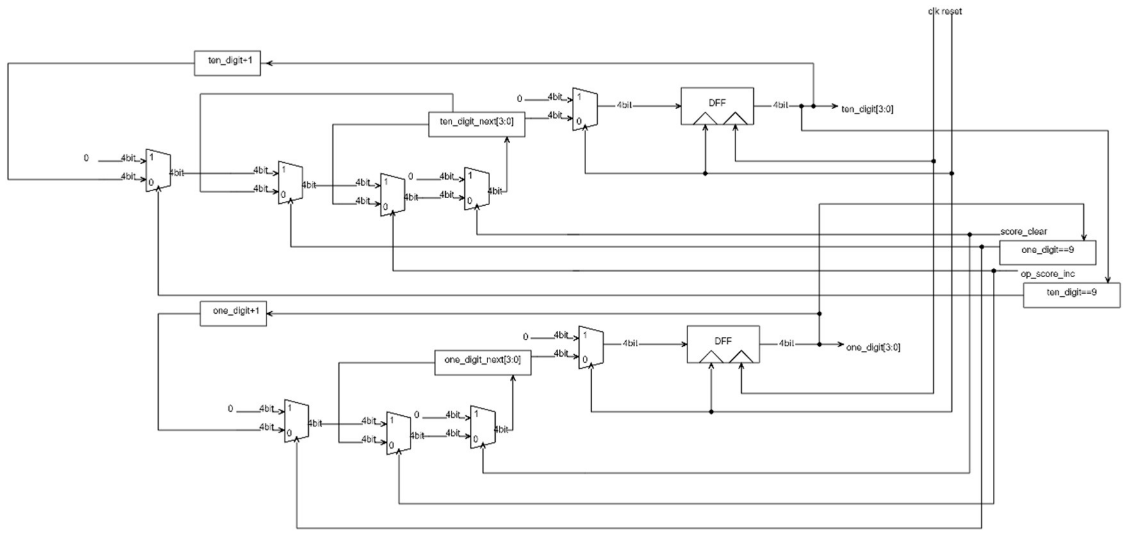


```
//combining four signal into a bus
assign text_on = {score_on, PONG_on, rule_on, over_on};

// ascii ROM interface
assign rom_addr = {char_addr, row_addr};
assign ascii_bit = ascii_word[~bit_addr];
```

At last, this block diagram can tell that the value of the 3 addr signal will be decided by multiplexer depending on on signals.



m100_counter:



input: clk, reset, score_inc, score_clear
output: one_digit[3:0], ten_digit[3:0]
wire: op_score_inc
reg: one_digit[3:0], ten_digit[3:0], one_digit_next[3:0], ten_digit_next[3:0]

How every signal works?

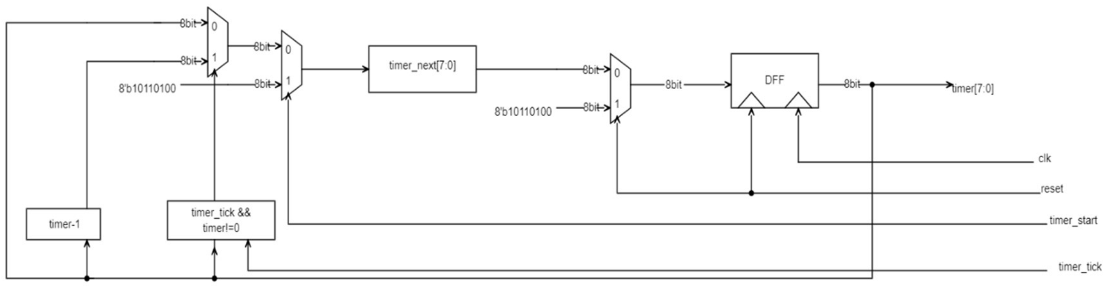
Speaking from the first sequential block triggered by **posedge clk** or **posedge reset**, if **reset** equals to 1, then both **ten_digit** and **one_digit** will be set to 0, else, **ten_digit** and **one_digit** will be set to **ten_digit_next** and **one_digit_next** respectively.

The second always block is a combinational block. First, it initialize **one_digit_next's** and **ten_digit_next's** value to **one_digit** and **ten_digit** respectively. If **score_clear** equals to 1, then both **one_digit_next** and **ten_digit_next** will be set as 0. Else if **op_score_inc**(one-pulse processed **score_inc** signal to prevent adding multiple points during one **hit**), if **one_digit** equals to 9, then **one_digit_next** shall be set as 0(carry), and that if **ten_digit** equals to 9, then **ten_digit_next** equals to 0(maximum score:99), else, **ten_digit_next** will be equal to **ten_digit**. Else if **one_digit** isn't equal to 9, then **one_digit_next** will be equal to **one_digit** adding by 1.

Below the always block is a module called **onepulse** to make

score_inc signal into one pulse to prevent adding multiple points in a single hit.

timer:



Input: timer_start, timer_tick, reset, clk
Output: timer_up
Reg: timer[7:0], timer_next[7:0]

How every signal works?

Basically, this module is separated into three parts, sequential always block, combinational always block, and an assign statement.

In the first always block(sequential block tiggered by **posedge reset** or **posedge clk**), if **reset** equals to 1, then **timer** should be set to 180(8'b10110100), else, **timer** should be equal to **timer_next**.

In the second always block(combinational block), if **timer_start** is equal to 1, then **timer_next** should be set as 180. Else if **timer_tick** equals to 1 and that **timer** isn't equal to 0, **timer_next** should be equal to **timer** -1. Else, **timer_next** should be equal to **timer.**

The last assign statement is to assign **timer_up** as 1 whenever **timer** equals to 0, which is similar to a count-down counter.

The reason why we use 180 as initial value is because **timer_tick** will be raised to 1 every 1/60 second, and that we want **timer_up** signal to be raised after three second(3*60=180).

Conclusion:

While realizing functions or implementing our ideas, we've encounter some issues that we've never think of. Most of the time spent on making this game isn't creating functionalities, but solving ridiculous issues that we'd never thought of. Also, coding games in Verilog is pretty challenging for Verilog-learning beginners like us, who couldn't even tell how VGA works exactly. But it was pretty fun though despite we've met many issues throughout the process.

After watching other groups' demo, we were amazed by their final results, and we've come to a conclusion: although coding games using Verilog is a bit inconvenient and inefficient, it forced us to understand specific topics in Verilog thoroughly to finish this project successfully. It definitely make us better at coding Verilog!

Reference:

1. Pong Game, author: fpga4fun
   Address: https://www.fpga4fun.com/PongGame.html

2. VGA Project Pong pt2 Complete Game Verilog Basys 3 FPGA Xilinx Vivado
   Address: https://www.youtube.com/watch?v=tELTeQb-Dc4
   Author: David J. Marion

3. An Arkanoid-like game using Verilog.
   Author: shaform
   Address: https://github.com/shaform/ArkanoidOnVerilog

4. Nandland Go Board Project 10 - Pong! (On your VGA Monitor)
   Author: nandland
   Address: https://www.youtube.com/watch?v=sFgNpK4yQwQ

5. Program Your Own FPGA Video Game
   Author: elements14 presents
   Address: https://www.youtube.com/watch?v=inrfigeLJeM&t=370s

Division for work:

111062134	游竣閔	50%
111062332	朱誼學	50%