

作业4_数字系统设计

实验目的

基于Basys 3的FPGA开发板，设计一个VGA显示适配器；在能显示图像的基础上，通过开发板上的按键进行交互，做成简单有趣的小游戏。

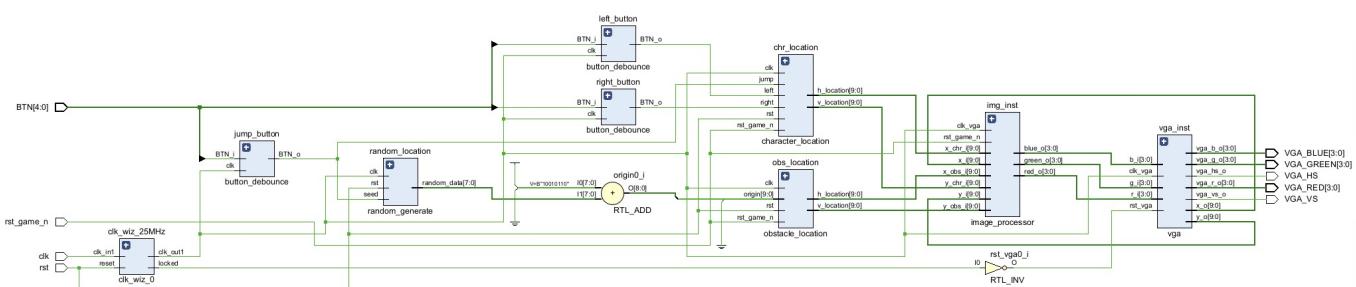
游戏简介

本实验设计了一个Flappy Bird为基础的小游戏，通过按下**BTNC**中间按键实现角色的跳跃，通过按下**BTNL**和**BTNR**左右按键实现角色的左右移动，如果没有按下跳跃键，人物将加速下落。通过最右侧开关实现整个系统的复位，通过右边第二个开关实现游戏的复位。



实验内容

模块划分

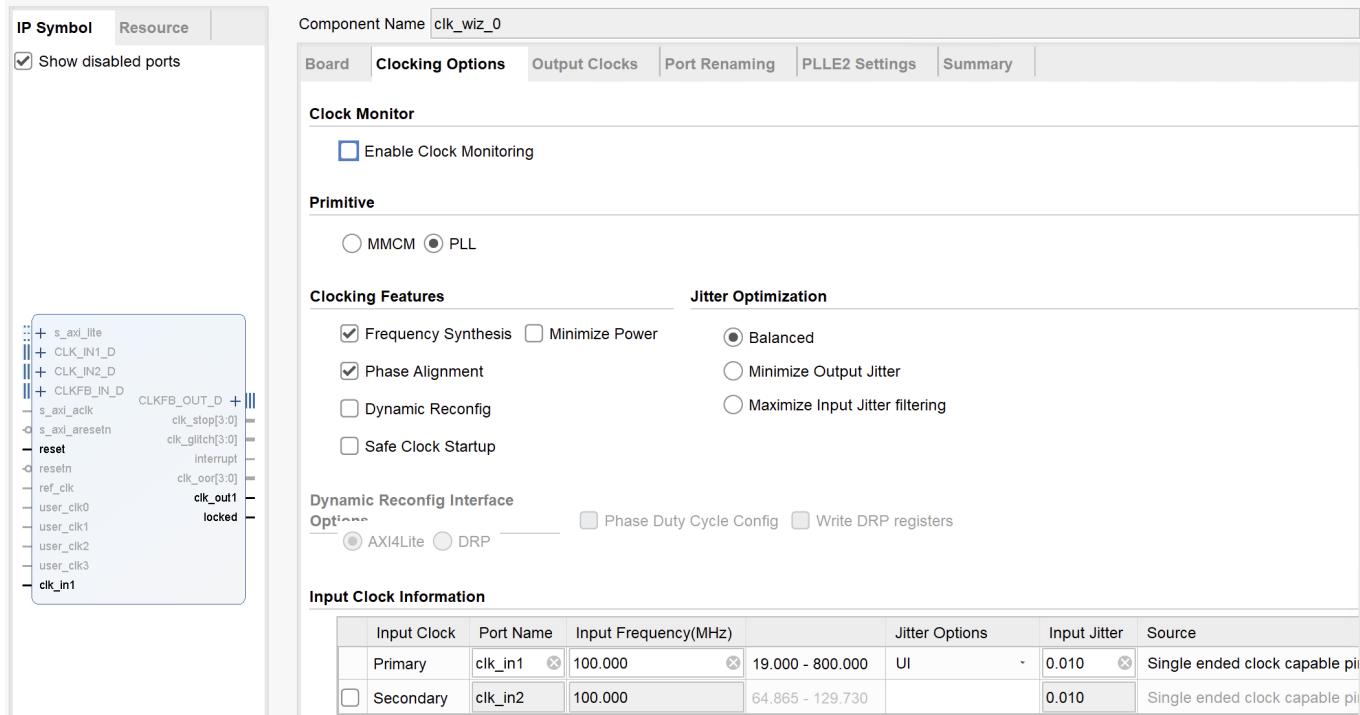


- 1、按键消抖模块 **button_debounce**: 对FPGA开发板上的按键输入进行消抖，输出稳定的按键信号。
- 2、时钟分频模块 **clk_wiz_0**: 将FPGA开发板上100MHz的时钟信号转换为25.175MHz的时钟信号。
- 3、角色位置模块 **character_location**: 输出每一个时刻角色应该处于的位置。

- 4、障碍物位置模块 `obstacle_location`: 输出每一个时刻障碍物下半部分左上角的位置。
- 5、随机数生成模块 `random_generate`: 输出一个8位二进制随机数，用于生成障碍物位置。
- 6、ROM数据导入模块 `rom_ip_cat_1`、`rom_ip_cat_2`: 储存角色的图像文件。
- 7、VGA显示数据模块 `image_processor`: 用于决定VGA屏幕显示内容。
- 8、二进制转BCD码模块 `binary_to_bcd`: 用于将二进制储存的分数转为bcd码用于显示。
- 9、VGA扫描模块 `vga`: 用于读入数据模块的图像显示内容，显示在VGA画面上。
- 10、顶层模块 `top_game`: 用于将上述模块整合在一起，与外界输入输出关联。

时钟分频模块

由于本实验计划VGA显示的分辨率为480 \times 640，在60Hz下输出，在这个参数下，水平参数为：显示像素640，前肩长16像素，同步脉冲长96像素，后肩长48像素，总计800像素；垂直参数为：显示行数为480，前肩为10行，同步脉冲2行，后肩长33行，总计525行。因此，像素时钟大小为： $800 \times 525 \times 60 = 25,200,000$ 像素/秒，因此采取25.175MHz的时钟分频。



本实验通过IP核中的PLL锁相环得到25.175MHz的时钟信号，由于精度问题，最终IP核仅能生成25.17007MHz的时钟信号，经后续观察，发现影响并不是很明显。

```
clk_wiz_0 clk_wiz_25MHz (
    .clk_in1(clk),
    .clk_out1(clk_25MHz),
    .reset(rst),
    .locked(clk_locked)
);
```

按键消抖模块

本实验采取忽略按键按下时长低于20ms的按键信号，只储存长度超过20ms的按键信号，由于前文已经采取时钟分频得到了25.175MHz的信号，粗略估计使用TIME_MIN为21'd500_000，具体代码实现如下：

```
module button_debounce #(
    parameter [20:0] TIME_MIN = 21'd500_000
) (
    input clk,
    input rst,
    input BTN_i,
    output reg BTN_o
);

reg [20:0] cnt; // A small counter to debounce
reg BTN_sync_0, BTN_sync_1; // Synchronized signal

always @(posedge clk, posedge rst) begin
    if (rst) begin
        BTN_o <= 0;
        cnt <= 0;
        BTN_sync_0 <= 0;
        BTN_sync_1 <= 0;
    end else begin
        BTN_sync_0 <= BTN_i;
        BTN_sync_1 <= BTN_sync_0;
        if (BTN_sync_1 == BTN_o) begin
            cnt <= 0;
        end else begin
            cnt <= cnt + 1;
            if (cnt >= TIME_MIN) begin
                cnt <= 0;
                BTN_o <= BTN_sync_1;
            end
        end
    end
end
end
endmodule
```

角色位置模块

该模块通过读入按键信号和游戏复位信号来输出角色每个时刻所在的位置数据，当复位信号激活后，角色复位到画面中央；当没有信号输入时，角色做自由落体运动。重力引擎的设置方法为：设置一个固定的加速度为1，每过一段时间对下落速度velocity进行+1，每过一段时间纵向位置加1个velocity的量。当按下跳跃键时，角色的下落速度清零，重新回到无初速的下落状态，并且位置上移10像素。当按下左右移动按键时，角色向左或者向右移动10像素位，但是纵向速度不做改变，该部分实现代码如下：

```
always @(posedge clk, posedge rst) begin
    if (rst) begin
```

```

v_location <= 10'd216;
h_location <= 10'd300;
velocity <= 0;
cnt <= 0;
cnt_plus <= 0;
end else if (rst_game_n) begin
    if (jump_impulse) begin
        v_location <= v_location - 20;
        cnt <= 0;
        velocity <= 0;
        cnt_plus <= 0;
    end else if (left_impulse) begin
        h_location <= h_location - 10;
    end else if (right_impulse) begin
        h_location <= h_location + 10;
    end else begin
        if (cnt < 21'd400_000) begin
            cnt <= cnt + 1;
        end else begin
            if (cnt_plus < 21'd10) begin
                v_location <= v_location + velocity;
                cnt <= 0;
                cnt_plus <= cnt_plus + 1;
            end else begin
                velocity <= velocity + 1;
                cnt_plus <= 0;
            end
        end
    end
end else begin
    v_location <= 10'd216;
    h_location <= 10'd300;
    velocity <= 0;
    cnt <= 0;
    cnt_plus <= 0;
end

```

此外，由于输入的信号是一段长信号，而按下次应该只上移一次角色，而非一直移动，所以这里需要检测输入信号的上升沿，每次上升沿对应一次跳跃，因此通过储存当前信号 `impulse` 和上一时刻信号 `prev`，当且仅当满足 `impulse & ~prev` 为真时，进行一次操作，代码实现如下：

```

// Check the button's positive edge
assign jump_impulse = jump & ~jump_prev;
assign left_impulse = left & ~left_prev;
assign right_impulse = right & ~right_prev;

// Store the previous state
always @(posedge clk) begin
    jump_prev <= jump;
    left_prev <= left;

```

```

        right_prev <= right;
end

```

障碍物位置模块

该模块主要储存障碍物下半部分的左上角坐标来定位整个障碍物的位置，本实验为简化代码，固定了上下两个障碍物之间的间隙和障碍物的水平宽度，因此通过这个位置可以确定整个上下两部分障碍物的情况。该模块通过接受来自`random_generate`产生的随机数来确定障碍物从右边出现的位置，当障碍物产生于右边后，以一定速度向左移动，移动至左侧边界后重新再右边生成，直到游戏结束，具体代码如下：

```

always @(posedge clk, posedge rst) begin
    if (rst) begin
        cnt <= 0;
        v_location <= origin;
        h_location <= 10'd625;
    end else if (rst_game_n) begin
        if (cnt < 21'd100_000) begin
            cnt <= cnt + 1;
        end else begin
            if (h_location > 10'd0) begin
                h_location <= h_location - 1;
            end else begin
                h_location <= 625;
                v_location <= origin;
            end
            cnt <= 0;
        end
    end else begin
        cnt <= 0;
        v_location <= 290;
        h_location <= 10'd625;
    end
end

```

随机数生成模块

考虑到直接使用`$random`这一内置函数生成随机数具有不可综合的问题，因此本实验采取了伪随机的方式生成随机数。具体过程为：设置一个8位计数器能够从0到255进行计数，当检测到跳跃按键按下时进行计数，当松开按键后保持计数，由于人按键时长在以25.175MHz单位下具有随机性，因此能够产生一个伪随机的8位二进制随机数`random_data`，具体代码如下：

```

reg [7:0] cnt;

always @(posedge clk, posedge rst) begin
    if (rst) begin
        cnt <= 0;
    end else begin
        if (seed) begin

```

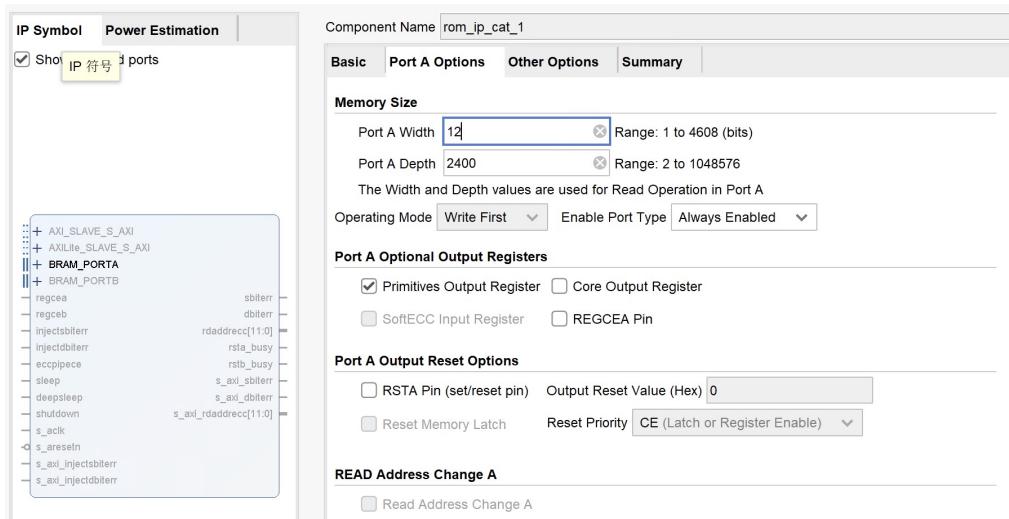
```

        cnt <= cnt + 1;
    end else begin
        cnt <= cnt;
    end
end
end

assign random_data = cnt;

```

ROM数据导入模块



通过IP核中的**Block Memory Generator**设置**ROM**的方式储存图片信息，这里由于只需要读入信息而不需要写入信息因此只使用了**Single Port Rom**而没有使用**RAM**，本实验RGB三种颜色储存均采用4位二进制数，并且图片大小为 40×51 ，因此取宽度为12，深度为2400，为方便起见选取**Always Enabled**的驱动方式。

在初始化方面，通过传入.coe文件初始化ROM，具体操作方式为通过Python中的PIL库读入图片在每个像素点生成12位的颜色数据信息，代码如下：

```

from PIL import Image
im = Image.open('1.png')
im = im.resize((45, 51))
# 检查图像模式，确保为 RGB，如果不是则进行转换
if im.mode != 'RGB':
    im = im.convert('RGB')
# 获取图像的像素数据
pixels = list(im.getdata())
# 打开或创建 COE 文件进行写入
with open('image_cat_1.coe', 'w') as f:
    # 写入 COE 文件的头部信息，指定进制（这里使用16进制）
    f.write('memory_initialization_radix=16;\n')
    f.write('memory_initialization_vector=\n')
    # 遍历所有像素，将 RGB 值转换为 12 位（每个通道 4 位）的十六进制字符串并写入文件
    for i, pixel in enumerate(pixels):
        r, g, b = pixel

```

```
# 将 8 位的 RGB 值 (0-255) 转换为 4 位的值 (0-15)
r_4bit = r >> 4
g_4bit = g >> 4
b_4bit = b >> 4
# 将 4 位的 RGB 值合成为一个 12 位的数
rgb12 = (r_4bit << 8) | (g_4bit << 4) | b_4bit
# 格式化为 3 位十六进制数
hex_str = '{:03X}'.format(rgb12)           f.write(hex_str)
# 如果不是最后一个像素，添加逗号和换行符
if i != len(pixels) - 1:
    f.write(',\n')
else:
    # 最后一个像素，用分号结束
    f.write(';\\n')
```

VGA显示数据模块

通过输入时钟、复位信号以及角色、障碍物的坐标和扫描坐标，输出每个扫描坐标下的RGB颜色数据值。从显示数据储存的角度来说，主要有两种方式：一种是通过ROM的IP核从.coe文件中读取图片信息，另一种是通过LUT的方式用assign的方式将显示数据储存在wire型变量中（由于开发板ROM储存空间较小，无法储存太多图片数据，因此用ROM储存复杂的角色图片而采用储存空间更大的LUT来储存单色数据），如下图所示：

在LUT写入图片数据时，采用Python中的PIL库对图片进行灰度处理，将白底像素和非白底像素通过0-1区分，并输出为LUT储存形式。具体操作代码如下：

```
from PIL import Image
# 打开图像并转换为灰度模式
image = Image.open('9.png').convert('L')
# 获取图像的尺寸 (假设是360x100)
width, height = image.size
```

```

# 确保图像大小为360x100
if (width, height) != (24, 30):
    raise ValueError("Image size must be 24x30")
# 创建一个列表来存储每一行的二进制字符串
binary_lines = []
# 遍历每一行，生成对应的二进制字符串
for y in range(height):
    binary_string = ''
    for x in range(width):
        # 获取像素值，255表示白色，0表示黑色
        pixel = image.getpixel((x, y))
        # 将像素值转换为二进制（白色为0，黑色为1）
        if pixel >= 235:
            binary_string += '0'
        else:
            binary_string += '1'
        # 格式化输出行
        binary_lines.append(f'    assign digit_image[9]{{29 - y}} = {24}\b{binary_string};')
    # 输出到文件
    with open('output.txt', 'w') as f:
        for line in binary_lines:
            f.write(line + '\n')
    print("输出完成，结果保存在output.txt中。")

```

在处理图片时，为了让图片显示更加清晰，主要采取了黑白的图片读入，并且为了处理掉图片中轮廓附近存在的一些颜色“毛刺”，将pixel大于235的像素全部视为白色，而非只有pixel为255的时候才视为白色，下图展示了处理前后的效果对比，左图为处理前的结果，右图为处理后的结果：



在读取数据和输出方面，通过设置`in_area`的`wire`型变量检测扫描坐标是否进入对应图片范围中，比如`assign in_area = (x_i >= x_lc) && (x_i < x_lc + width) && (y_i >= y_lc) && (y_i < y_lc + width);`来检测，通过建立`img_x`、`img_y`变量来储存进入图片范围后在图片的定位坐标，比如`assign img_x = x_i - x_lc;`来记录图片中坐标，通过这样的方式来读取图片信息。

两种不同的数据储存方式具有不同的读取方式，对于LUT储存的数据，通过判断`in_area`和定位坐标处数据是否为1来进行赋值，如果条件为真，则为其`red_o`、`green_o`、`blue_o`值进行赋值；对于ROM储存的数据，通过IP核模块输出的12位`pixel`数据进行RGB值赋值，其中`pixel[11:8]`对应Red值，`pixel[7:4]`对应Green值，`pixel[3:0]`对应Blue值，具体`pixel`数据输出代码如下：

```

rom_ip_cat_1 rom_cat1_inst (
    .clka(clk_vga),
    .addr(img_y * IMG_WIDTH + img_x),
    .douta(rom_pixel_data_cat_1)
);

```

为避免游戏中出现"穿模"的bug，在设计代码时需要确定图像显示的优先级，因此采用if-else语句来确定优先级，判断语句靠前的图像拥有更高的优先级，当显示该位置的颜色后不再显示其他图像，在else中输出背景底色，具体代码如下：

```

if (~end_game && in_image_area && crash_flag) begin
    end_game <= 1;
end else if (~end_game && in_image_area) begin
    red_o <= clk_cat ? rom_pixel_data_cat_1[11:8] :
rom_pixel_data_cat_2[11:8];
    green_o <= clk_cat ? rom_pixel_data_cat_1[7:4] :
rom_pixel_data_cat_2[7:4];
    blue_o <= clk_cat ? rom_pixel_data_cat_1[3:0] :
rom_pixel_data_cat_2[3:0];
end else if (in_score_area && score_image[SCORE_HEIGHT - 1 -
score_y][SCORE_WIDTH - score_x]) begin
    red_o <= 4'h0;
    green_o <= 4'h0;
    blue_o <= 4'h8;
end else if (in_digitHundreds && digitHundreds[DIGIT_HEIGHT -
1 - digitHundreds_y][DIGIT_WIDTH - digitHundreds_x]) begin
    red_o <= 4'h0;
    green_o <= 4'h0;
    blue_o <= 4'h8;
end else if (in_digitTens && digitTens[DIGIT_HEIGHT - 1 -
digitTens_y][DIGIT_WIDTH - digitTens_x]) begin
    red_o <= 4'h0;
    green_o <= 4'h0;
    blue_o <= 4'h8;
end else if (in_digitOnes && digitOnes[DIGIT_HEIGHT - 1 -
digitOnes_y][DIGIT_WIDTH - digitOnes_x]) begin
    red_o <= 4'h0;
    green_o <= 4'h0;
    blue_o <= 4'h8;
end else if (in_obstacle_area && ~end_game) begin
    red_o <= 4'h0;
    green_o <= 4'h8;
    blue_o <= 4'h8;
end else if (end_game && in_end_area &&
end_game_image[END_HEIGHT - 1 - end_y][END_WIDTH - end_x]) begin
    red_o <= 4'hf;
    green_o <= 4'h0;
    blue_o <= 4'h0;
end else begin
    red_o <= 4'hf;

```

```
green_o <= 4'hf;  
blue_o <= 4'hf;  
end
```

由于本实验中角色采用动图的形式进行展示，具体体现方法为储存多张图片进行循环播放达到动图的效果，类似于聊天平台中发送的GIF表情包，因此额外进行了动图显示分频时钟`clk_cat`，当时钟信号改变时改变输出的图片。在本实验中主要采取两种图片来回变换来实现Arc猫奔跑的动图感：

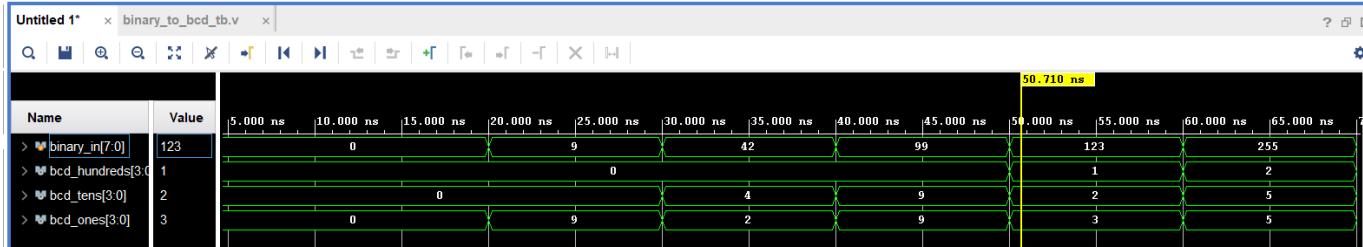


在检测碰撞时，采用颜色识别的方式进行检测，而非矩形碰撞箱的方式检测，这样方式能够更加精确地去判断游戏是否结束。具体而言，当扫描坐标进入角色图片范围内，检测当前像素点是否同时满足角色像素颜色数据不为白且存在障碍物的颜色，如果满足则发生碰撞而引起游戏结束，具体实现为`assign crash_flag = (red_o == 4'd0 && green_o == 4'd8 && blue_o == 4'd8) && rom_pixel_data_cat_1 && rom_pixel_data_cat_2;`，当游戏结束后，进入游戏结束页面如下：



二进制码转BCD码模块

该模块采用`Double Dabble`算法的方式进行转换，将二进制输入数扩展到足够宽的寄存器，以容纳最终的BCD结果。本实验通过`Simulation`验证了该算法的准确性，具体结果如下：



在本实验中，使用了一个 20 位的移位寄存器 shift_reg，其中高 12 位用于存储 BCD 的各个位（百位、十位、个位），低 8 位存储输入的二进制数。对二进制数的每一位，从高位到低位，逐一进行处理。在每次处理前，检查每个 BCD 位的值：如果某个 BCD 位的值大于或等于 5，则加上 3。这是为了在后续的移位操作中避免 BCD 位溢出，确保每个位始终保持在 0-9 之间。将整个寄存器内容左移一位，同时将下一个二进制位填入最低位。过所有位的处理后，寄存器中的高 12 位即为转换后的 BCD 值，分别对应百位、十位和个位。这里为了保证代码是可综合的，同时为了保证电路是组合设计的，因此采用多个 `wire` 型变量 `shift_reg` 来储存每一步的计算过程，具体代码如下：

```

wire [19:0] shift_reg [0:8];
genvar i;
assign shift_reg[0] = {12'd0, binary_in};
// Use double dabble method
generate
    for (i = 0; i < 8; i = i + 1) begin : dd_logic
        wire [3:0] hundreds = shift_reg[i][19:16];
        wire [3:0] tens     = shift_reg[i][15:12];
        wire [3:0] ones     = shift_reg[i][11:8];
        wire [19:0] shifted;

        wire [3:0] hundreds_adj = (hundreds >= 5) ? hundreds + 4'd3 :
hundreds;
        wire [3:0] tens_adj      = (tens      >= 5) ? tens      + 4'd3 :
tens;
        wire [3:0] ones_adj      = (ones      >= 5) ? ones      + 4'd3 :
ones;

        assign shifted = {hundreds_adj, tens_adj, ones_adj,
shift_reg[i][7:0]} << 1;

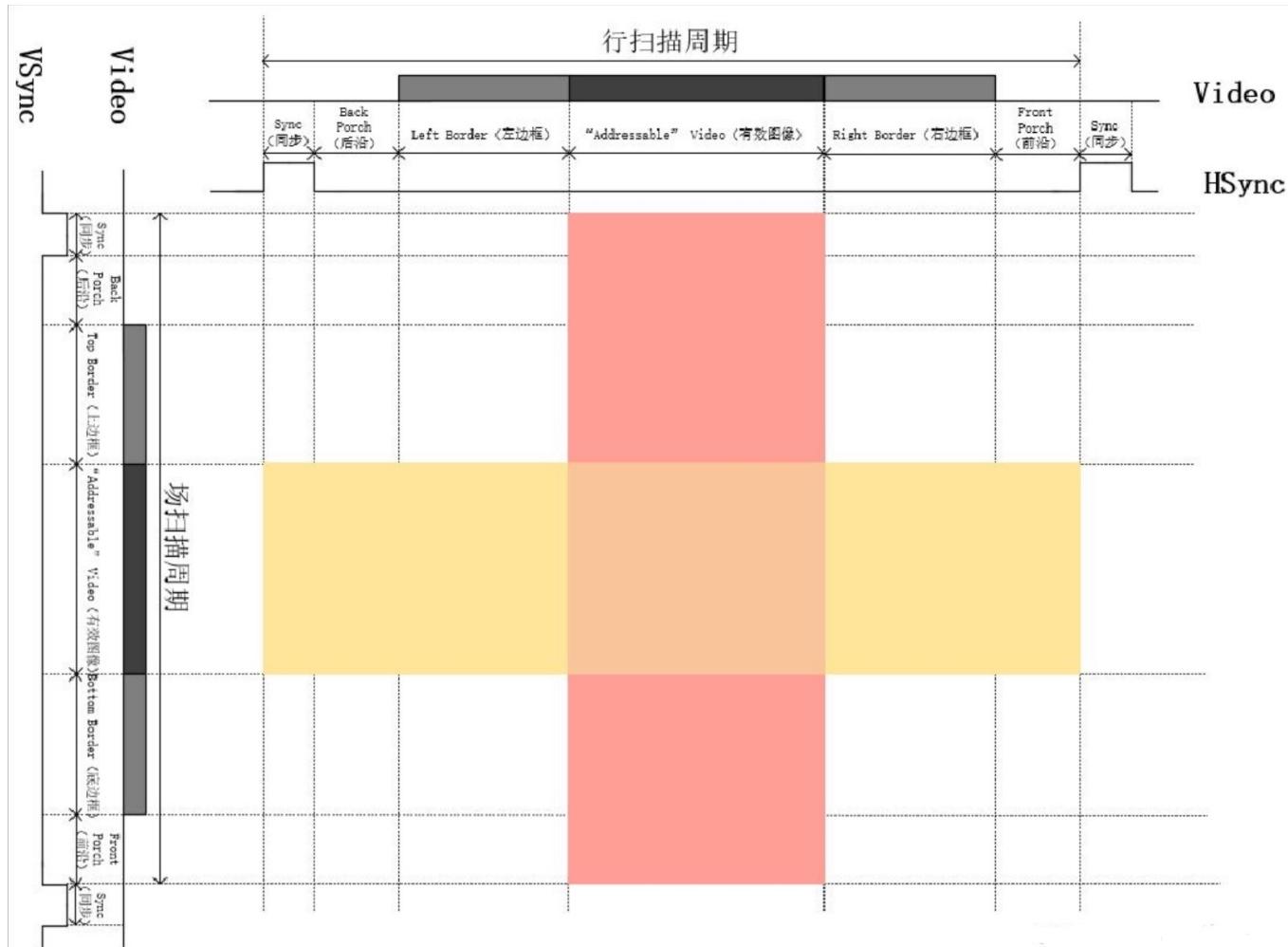
        assign shift_reg[i+1] = shifted;
    end
endgenerate

assign bcdHundreds = shift_reg[8][19:16];
assign bcdTens     = shift_reg[8][15:12];
assign bcdOnes     = shift_reg[8][11:8];

```

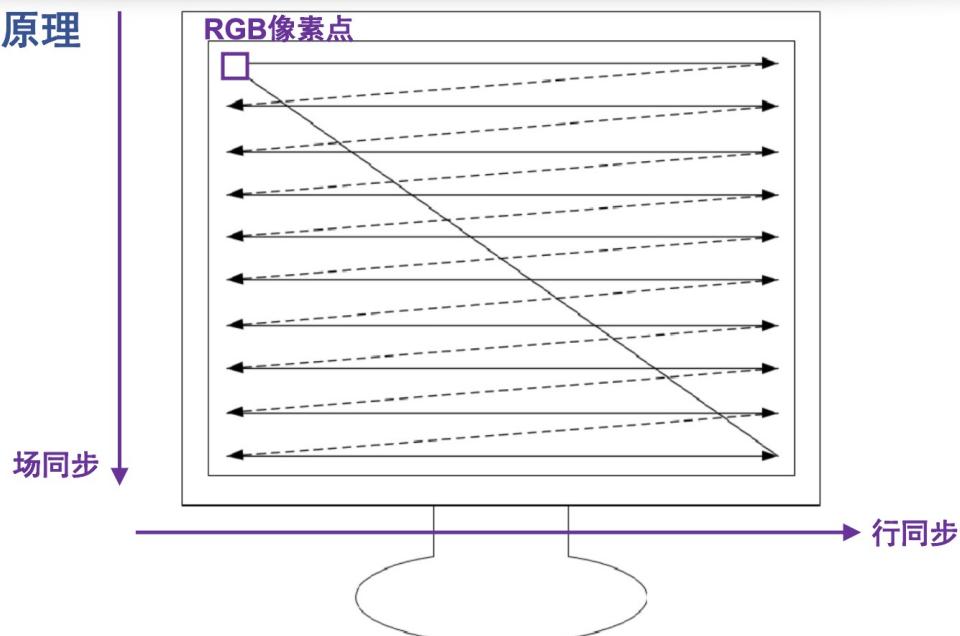
VGA扫描模块

按照查阅得到的数据设置行同步扫描参数和场同步扫描参数，扫描过程为从左至右，从上到下，具体扫描方法如下图所示：



实际上，扫描过程中除显示图像部分外，行扫描周期和场扫描周期外还存在前沿部分、后沿部分、同步部分，且在扫描过程中同步信号高有效，其他部分低有效，具体展示如下：

VGA显示原理



代码实现如下：

```
// Horizontal Timing Parameters
localparam H_SYNC_PW = 96; // Horizontal Sync Pulse Width
```

```
localparam H_L_PORCH = 48; // Horizontal Left Porch
localparam H_VISIBLE = 640; // Visible Area
localparam H_R_PORCH = 16; // Horizontal Right Porch
localparam H_TOTAL_W = 800;

// Vertical Timing Parameters
localparam V_SYNC_PW = 2; // Vertical Sync Pulse Width
localparam V_U_PORCH = 33; // Vertical Up Porch
localparam V_VISIBLE = 480; // Visible Area
localparam V_D_PORCH = 10; // Vertical Down Porch
localparam V_TOTAL_H = 525;

reg [9:0] h_cnt, v_cnt; // Horizontal Counter and Vertical Counter
reg rgb_en;

// Scan the horizontal and vertical line
always @(posedge clk_vga) begin
    if (rst_vga) begin
        v_cnt <= 0;
        h_cnt <= 0;
    end else begin
        if (h_cnt == H_TOTAL_W - 1 && v_cnt == V_TOTAL_H - 1) begin
            h_cnt <= 0;
            v_cnt <= 0;
        end else if (h_cnt == H_TOTAL_W - 1) begin
            v_cnt <= v_cnt + 1;
            h_cnt <= 0;
        end else begin
            v_cnt <= v_cnt;
            h_cnt <= h_cnt + 1;
        end
    end
end

// Decide whether the pixel is lightened or not. Only in the visible
area, the pixel is lightened.
always @(*) begin
    if (rst_vga) begin
        x_o = 0;
        y_o = 0;
        rgb_en = 0;
    end else if (v_cnt >= V_SYNC_PW + V_U_PORCH && v_cnt < V_SYNC_PW +
V_U_PORCH + V_VISIBLE
                && h_cnt >= H_SYNC_PW + H_L_PORCH && h_cnt < H_SYNC_PW +
H_L_PORCH + H_VISIBLE) begin
        x_o = h_cnt - (H_SYNC_PW + H_L_PORCH);
        y_o = v_cnt - (V_SYNC_PW + V_U_PORCH);
        rgb_en = 1;
    end else begin
        x_o = x_o;
        y_o = y_o;
        rgb_en = 0;
    end
end
end
```

```
assign vga_vs_o = v_cnt < V_SYNC_PW;
assign vga_hs_o = h_cnt < H_SYNC_PW;

assign vga_r_o = rgb_en ? r_i : 4'd0;
assign vga_b_o = rgb_en ? b_i : 4'd0;
assign vga_g_o = rgb_en ? g_i : 4'd0;
```

实验结果

视频展示可以见 <https://www.bilibili.com/video/BV1zRmiYZEqc>

实验总结与心得

这次数字系统设计的作业在设计上比以往几次更为复杂，单一的模块复杂度可能已经能匹配前几次的作业了。此外，也得益于前几次作业设计积累的经验，这次设计才能顺利完成，比如：在自动售货机作业中对按键的消抖模块用在了这次设计的输入中，数码管频闪显示的方式用在了本次角色动图的插入中。

在本次实验中也学会了从简到难的设计方法，由于代码量的庞大和思路的复杂性，这次实验不能像往常一样直接按照一个思路从头写到底进行调试，而是在实验过程中一点点加入新的内容调试。比如在这次实验中，我首先通过给每个像素赋值为白色来测试自己的VGA扫描模块和VGA渲染数据模块正确性。在确认单一颜色覆盖成功后，尝试通过ROM的方式读入coe文件静态图片显示，在这一过程中发现ROM的储存空间有限，不能容纳较大的图片，在下板数据中发现LUT利用率远远低于BRAM的利用率，因此尝试了小而复杂的图使用ROM的IP核方式读取，而大而简单的图采用LUT的方式储存在wire型总线中。在图片显示成功后，加入障碍物的移动、角色的移动再到角色的重力引擎测试，一点点循循渐进改进项目。当游戏轮廓勾画出来后，再加入随机数的生成，最后添加记分板完成项目，整个过程并非一气呵成，而是一点点调试，这样利于发现中间出现的各种各样的bug，避免了项目结束后难以寻找代码问题的结果。

由于时间限制，本实验也存在许多可以改进的地方，比如添加游戏暂停功能，增加难度功能（随着分数提高，障碍物移动速度更快），设置多样方向的障碍物（本来写过了角色的左右移动，打算增加垂直方向的障碍物，但是没来得及完成这个想法）。