浙江大学

数字逻辑设计

课程设计

设计思路

游戏功能

该游戏为一个 **迷宫路径** 小游戏。我们需要用**最少步数**从迷宫入口走到出口（每走一步步数 + 1）。同时，地图上的不同角落散落了一些宝箱，若我们（愿意多走几步）获取宝箱就能减少总步数（减少步数随机）。快来考验你的运气和 trade-off 能力吧！

I/O 接口

在此次小游戏中，我们使用的输入有：

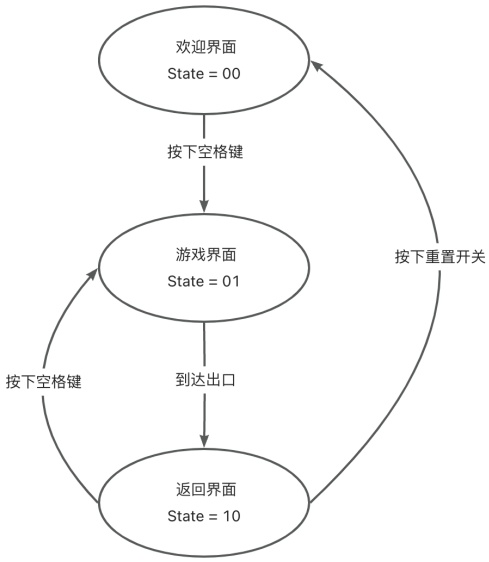
- ROM 模块，存储宝箱、人物、地图、背景等图片信息
- PS2 模块，获取键盘输入来控制界面跳转和人物移动
- SWITCH 开关，用于对游戏进行重置

我们使用的输出有：

- VGA 模块，用于显示游戏界面
- 七段数码管，用于显示当前步数或当前最短步数

时序逻辑

此次大程的时序逻辑并不复杂。在 Verilog 代码中，我们用 state 变量来记录当前状态。初始化状态下，我们会进入游戏欢迎界面，此时 state = 00。当我们按下空格键，游戏随即开始，游戏状态下 state = 01 。在游戏状态下，我们可以用 WASD 四键来操控人物的移动状态。当人物成功走到出口，我们将进入返回界面，此时 state = 10。倘若我们按下空格键，则我们将再次进入游戏；倘若我们拨下重置开关，游戏将重置并进入欢迎界面。



核心模块说明

ROM 模块

为了游戏的观感以及设计方便，我对人物、宝箱、界面等元素进行了预先设计并保存成图片，这样就省去了用 `verilog` 手搓像素点的痛苦。

但是，在这种情况下，有两点非常值得注意：

1. 我们需要将生成的图片转换成 `IP` 核可以读取的 `COE` 文件
2. 由于板上的存储资源有限，我们需要控制图片的总像素

在刚开始，我对 `COE` 文件的生成一头雾水，也没有获得任何的指导，后来上网查了很久之后才了解了 `COE` 文件生成的基本逻辑和方式。在生成过程中，我们

需要格外注意对像素的 rearrange 和 rgb 像素的排列方式。以下是我的 matlab 代码：

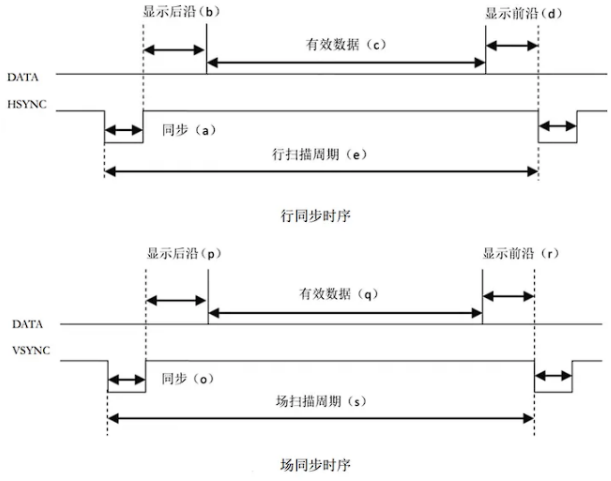
```
clear                %清理命令行窗口
clc                  %清理工作区

% 使用 imread 函数读取图片,并转化为三维矩阵
image_array = imread('...');

% 使用 size 函数计算图片矩阵三个维度的大小
% 第一维为图片的高度，第二维为图片的宽度，第三维为图片维度
[height,width,z]=size(image_array);
red   = image_array(:,:,1);
green = image_array(:,:,2);
blue  = image_array(:,:,3);
% 这样导出的 rgb 分量都是 8bit

% 使用 reshape 函数将各个分量重组成一个一维矩阵
%为了避免溢出,将 uint8 类型的数据扩大为 uint32 类型
r = uint32(reshape(red', 1 ,height*width));
g = uint32(reshape(green', 1 ,height*width));
b = uint32(reshape(blue', 1 ,height*width));

% 初始化要写入.COE 文件中的 RGB 颜色矩阵
rgb=zeros(1,height*width);   % 1 维 数据大小

% 显示模式的转换
% 将 RGB888 转换为 RGB444
for i = 1:height*width
   rgb(i) = bitshift(bitshift(r(i),-4),0)+ bitshift(bitshift(g(i),-4),
4)+ bitshift(bitshift(b(i),-4),8);
end

fid = fopen( '...', 'w+' );      % // 这里的地址需要写成绝对路径

fprintf( fid, 'MEMORY_INITIALIZATION_RADIX=16;\n');
fprintf( fid, 'MEMORY_INITIALIZATION_VECTOR=\n',height*width);

% 写入图片数据
for i = 1:height*width
    if i == height*width
        fprintf(fid,'%x;\n',rgb(i)); %最后一个数据后面加分号
    else
        fprintf(fid,'%x,\n',rgb(i));
    end
end

fclose( fid ); % 关闭文件指针
```

VGA 显示模块

VGA(Video Graphics Array) 协议是一种使用模拟信号的显示标准，我们需要提供的是数字信号（如比较重要的 RGB 三色值、扫描同步信号等），板内 DAC(Digital-to-Analog Converter, DAC/D2C) 会将其转换为 VGA 接口需要的模拟信号。

影响画面质量的因素较主要的有分辨率、刷新率以及色彩。**分辨率**指屏幕中显示的有效像素点数量，一般以 aaa × bbb 表示，前者指一行中像素点个数，后者指一列中像素点个数。**刷新率**指屏幕内容刷新速度，一般以 Hz 为单位表示一秒钟刷新多少次。**色彩**主要指色彩空间格式与“精度”，VGA 要求使用 RGB 色彩空间（即红绿蓝三色混合）且为 12 位，即一个色彩通道用 4 位表示。在我们的实验中，使用 640×480@60Hz 的显示模式，需要接入的时钟频率为 25.175MHz。

我们使用**逐行扫描**的方式来打印界面，即每次图片刷新都从左上角开始，先从左到右扫描完一行，再转到下一行的最左边开始扫描，直到扫描完最后一行。我们需要处理**行时序**与**场时序**，时序图如下：



输入信号：

- vga_clk：由原理可知，迎接入 25MHz 时钟，如果接入后无法正常显示，可以尝试用 MMCM 获得 25.175MHz 的时钟接入
- clrn：重置信号，低电平有效，有效时将扫描信号归位到 (0, 0)
- d_in：12 位 RGB 信号，格式为 bbbb_gggg_rrrr，每个色彩通道使用 4 位。需要注意，这里的 RGB 值是根据**上个**时钟周期的 row_addr, col_addr 确定的

输出信号：

- row_addr, col_addr：扫描地址
- r, g, b：三个色彩通道值，直接连接到顶层模块输出即可
- rdn：判断当前扫描到的地址是否为有效数据，低电平为有效
- hs, vs：行同步信号与场同步信号

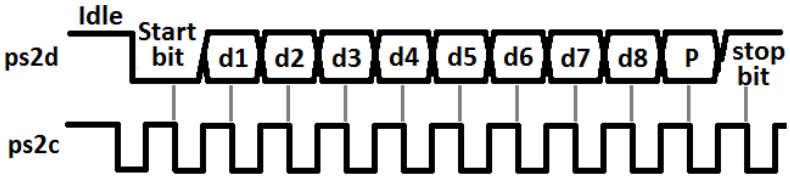
以下是我的 VGA 显示模块代码：

```
module VGA(
    input clk,  //  25MHz
    input rst,
    input [11:0] Din,   //  bbbb_gggg_rrrr, pixel
    output reg [8:0] row,
    output reg [9:0] col,
    output reg rdn,
    output reg [3:0] R, G, B,
    output reg HS, VS
);

    reg [9:0] h_count;

    always @(posedge clk) begin
        if (rst)
            h_count <= 10'h0;
        else if (h_count == 10'd799)
            h_count <= 10'h0;
        else
            h_count <= h_count + 10'h1;
    end

    reg [9:0] v_count;
    always @(posedge clk or posedge rst) begin
        if (rst)
            v_count <= 10'h0;
        else if (h_count == 10'd799) begin
            if (v_count == 10'd524)
                v_count <= 10'h0;
            else
                v_count <= v_count + 10'h1;
        end
    end

    wire [9:0] row_addr = v_count - 10'd35;
    wire [9:0] col_addr = h_count - 10'd143;
    wire       h_sync = (h_count > 10'd95);
    wire       v_sync = (v_count > 10'd1);
    wire       read = (h_count > 10'd142) &&
                      (h_count < 10'd783) &&
                      (v_count > 10'd34)  &&
                      (v_count < 10'd515);

    always @(posedge clk) begin
        row <= row_addr[8:0];
        col <= col_addr;
        rdn <= ~read;
        HS  <= h_sync;
        VS  <= v_sync;
        R   <= rdn ? 4'h0 : Din[3:0];
        G   <= rdn ? 4'h0 : Din[7:4];
        B   <= rdn ? 4'h0 : Din[11:8];
    end

endmodule
```

PS2 模块

PS/2 接口(Personal System/2)是一种 PC 电脑上的接口，可用来连接键盘和鼠标。接口共有 6 个接脚，除接地与 Vcc 外，有**时钟和一位数据**(另外两脚为保留未使用）。

PS/2 协议中，一次传输有效数据为**一字节**，每次传输（一帧）为 **11 位**，分别为开始位（1 位，一直为 0）、有效数据（8 位）、校验位（1 位）、结束位（1 位，一直为 1），在我们简单的设计里，每一帧数据中只需要关注中间的 8 位有效数据即可。



PS/2 将键盘编码分为通码(Make)与断码(Break)，通码代表“按下”，断码代表“松开”。键盘上大部分按键的通码只有一字节（比如 WASD 等字母按键），但也有特殊按键的通码为两字节（比如上下左右，其格式为 E0 开头的两字节数据）。断码是在通码的基础上添加一字节的 F0 数据，比如 W 的通码为 1D 断码为 F0 1D，上 ↑ 的通码为 E0 75 断码为 E0 F0 75。由此我们可知，通码可能需要 1~2 帧，断码可能需要 2~3 帧（如果传输内容超过 1 帧，键盘可以保证传输内容是连续的，不会被其他信号隔开）。

由于 PS 模块的频率比较低（kHz 水平），因此我们可以对时钟信号进行分频来降低它的采样频率。需要注意的是，PS2 模块的按键信号并非 ASCII 码，因此我们需要先获得每个按键的信号再进行后续的设计：

```
module ps2 (
    input clk,
    input rst,
    input ps2_clk,
    input ps2_data,
    output [9:0] data_out,
    output ready
);

    reg ps2_clk_flag0, ps2_clk_flag1, ps2_clk_flag2;
    wire negedge_ps2_clk;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            ps2_clk_flag0 <= 1'b0;
            ps2_clk_flag1 <= 1'b0;
            ps2_clk_flag2 <= 1'b0;
        end
        else begin
            ps2_clk_flag0 <= ps2_clk;
            ps2_clk_flag1 <= ps2_clk_flag0;
            ps2_clk_flag2 <= ps2_clk_flag1;
        end
```

```
end

assign negedge_ps2_clk = !ps2_clk_flag1 & ps2_clk_flag2;
reg [3:0] num;

always @(posedge clk or posedge rst) begin
    if (rst)
        num <= 4'd0;
    else if (num == 4'd11)
        num <= 4'd0;
    else if (negedge_ps2_clk)
        num <= num + 1'b1;
end

reg negedge_ps2_clk_shift;

always @(posedge clk) begin
    negedge_ps2_clk_shift <= negedge_ps2_clk;
end

reg [7:0] temp_data;
always @(posedge clk or posedge rst) begin
    if (rst)
        temp_data <= 8'd0;
    else if (negedge_ps2_clk_shift) begin
        case (num)
            4'd2 : temp_data[0] <= ps2_data;
            4'd3 : temp_data[1] <= ps2_data;
            4'd4 : temp_data[2] <= ps2_data;
            4'd5 : temp_data[3] <= ps2_data;
            4'd6 : temp_data[4] <= ps2_data;
            4'd7 : temp_data[5] <= ps2_data;
            4'd8 : temp_data[6] <= ps2_data;
            4'd9 : temp_data[7] <= ps2_data;
            default : temp_data <= temp_data;
        endcase
    end
    else begin
        temp_data <= temp_data;
    end
end

reg data_break, data_done, data_expand;
reg [9:0] data;

always @(posedge clk or posedge rst) begin
    if (rst) begin
        data_break <= 1'b0;
        data <= 10'd0;
        data_done <= 1'b0;
        data_expand <= 1'b0;
    end
    else if (num == 4'd11) begin
        if (temp_data == 8'hE0)
            data_expand <= 1'b1;
        else if (temp_data == 8'hF0)
            data_break <= 1'b1;
        else begin
            data <= {data_expand, data_break, temp_data};
            data_done <= 1'b1;
            data_expand <= 1'b0;
```

```
                data_break <= 1'b0;
            end
        end
        else begin
            data <= data;
            data_done <= 1'b0;
            data_expand <= data_expand;
            data_break <= data_break;
        end
    end

    assign data_out = data;
    assign ready = data_done;

endmodule
```

核心控制模块

首先，我们有必要对本次游戏中用到的重要变量进行说明：

3. 人物坐标。

```
// store the position of the figure
reg [9:0] col_pos;
reg [8:0] row_pos;
```

1. 宝箱信息。其中，我们有 8 个宝箱，每个宝箱有 x, y 坐标。为了节省空间和设计便利，我们以图片的左上角为坐标记录点，并采用网格坐标法（即以 32 像素为网宽划分屏幕），每个坐标点用 5 位来记录。 box_ena 来记录宝箱是否被访问（即是否应该显示）

```
// store the position of the boxes
reg [79:0] box_pos;
// store the visibility of the boxes
reg [7:0] box_ena;
```

1. 状态信息。

```
// state of the game
// 00 : welcome page
// 01 : game page
// 02 : finish page
reg [1:0] state;
```

在游戏开始时，我们先对必要信息进行初始化：

```
    initial begin
        state = 2'b00;

        col_pos = 10'd512;
        row_pos = 9'd64;

        score = {8{1'b0}};
        best_record = {8{1'b1}};

        box_ena = {8{1'b1}};
```

```
        box_pos[4:0] = 5'd3;    box_pos[9:5] = 5'd10;
        box_pos[14:10] = 5'd4;  box_pos[19:15] = 5'd4;
        box_pos[24:20] = 5'd5;  box_pos[29:25] = 5'd7;
        box_pos[34:30] = 5'd7;  box_pos[39:35] = 5'd15;
        box_pos[44:40] = 5'd8;  box_pos[49:45] = 5'd9;
        box_pos[54:50] = 5'd9;  box_pos[59:55] = 5'd4;
        box_pos[64:60] = 5'd10;  box_pos[69:65] = 5'd13;
        box_pos[74:70] = 5'd11;  box_pos[79:75] = 5'd7;
    end
```

接下来是核心的状态控制模块：

```
    always @(posedge clkdiv[22]) begin
        //  state controller
        case (state)
            2'b00: begin
                if (data[7:0] == 8'h29 && data[8] == 0) begin
                    state = 2'b01;
                end
            end

            2'b01: begin
                //  movement controller
                if (data[8] == 0) begin
                    //  left
                    if (data[7:0] == 8'h1c && (col_pos >= 128 && row_pos !
= 64)) begin
                        col_pos = col_pos - 32;
                        score = score + 1;
                    end
                    //  right
                    if (data[7:0] == 8'h23 && col_pos < 512) begin
                        col_pos = col_pos + 32;
                        score = score + 1;
                    end
                    //  up
                    if (data[7:0] == 8'h1d && row_pos >= 128) begin
                        row_pos = row_pos - 32;
                        score = score + 1;
                    end
                    //  down
                    if (data[7:0] == 8'h1b && (row_pos <= 320 || (row_pos
== 352 && col_pos == 96))) begin
                        row_pos = row_pos + 32;
                        score = score + 1;
                    end
                end

                //  win
                if (row_pos == 384 && col_pos == 96) begin
                    best_record = (score < best_record)? score : best_rec
ord;
                    state = 2'b10;
                    col_pos = 10'd512;
                    row_pos = 9'd64;
                    box_ena = {8{1'b1}};
                end

                //  box detector
                for (i = 0; i < 8; i = i + 1) begin
                    if (box_pos[(i * 10) +: 5] * 32 == row_pos && box_pos
```

```
[(i * 10 + 5) +: 5] * 32 == col_pos && box_ena[i] == 1'b1) begin
                    box_ena[i] = 1'b0;
                    score = (score - 5 < 0) ? 0 : score - 5;
                end
            end
        end

        2'b10 : begin
            score = 8'b0;
            if (data[7:0] == 8'h29 && data[8] == 0) begin
                state = 2'b01;
            end
        end

    endcase

    if (SW == 1'b1) begin
        state = 2'b00;
        col_pos = 10'd512;
        row_pos = 9'd64;
        score = {8{1'b0}};
        best_record = {8{1'b1}};
        box_ena = {8{1'b1}};
    end
end
```

- 在 `00` 状态下，我们需要探测空格输入。
- 在 `01` 状态下，我们需要探测方向键输入、宝箱检测、出口检测
- 在 `10` 状态下，我们需要探测空格输入。
- 额外地，在任何情况下，我们都需要对重置按钮进行检测。

数码管显示模块

```
    reg [15:0] num;
    always @(*) begin
        if (state == 2'b00) begin
            num = 16'b0;
        end
        else if (state == 2'b01) begin
            num[15:12] = 4'b0;  num[11:8] = 4'b0;   num[7:4] = score / 10;
  num[3:0] = score % 10;
        end
        else if (state == 2'b10) begin
            num[15:12] = 4'b0;  num[11:8] = 4'b0;   num[7:4] = best_recor
d / 10;  num[3:0] = best_record % 10;
        end
    end
```

由于在之前的实验中，我们已经大量使用过七段数码管显示的模块，因此在这里我们直接对它进行了复用。需要注意的是，我们需要合理地控制数字输入接口和状态设计。在游戏状态下，我们将显示当前步数，而在其他状态下，我们将显示历史最佳步数记录。

VGA 控制模块

在这个模块中，我们主要实现了对 ROM 模块的读入控制和定位，并按照游戏逻辑对相应元素进行控制输出。

首先，我们对每张存储在 ROM 中的图片都给予了对应的 pos（图片中的位置信息）和 pic（图片的像素信息）。接下来，我们按照图层的显示顺序对 vga_data 进行赋值。以下是我的代码：

```
module vgaRGB(
    input wire rgb_clk,
    input wire [9:0] hc,
    input wire [8:0] vc,

    input wire [1:0] state,

    input wire [9:0] col_pos,
    input wire [8:0] row_pos,

    input wire [79:0] box_pos,
    input wire [7:0] box_ena,

    output reg [11:0] vga_data
);

    reg [18:0] pos;
    wire [11:0] welcome_pic;
    wire [11:0] background_pic;
    wire [11:0] figure_pic;
    wire [11:0] box_pic;
    wire [11:0] back_pic;

    reg [10:0] figure_pos;
    reg [10:0] inbox_pos;
    reg [10:0] back_pos;

    integer i;

    always @(*) begin
        pos <= vc * 640 + hc;

        if (state == 2'b00) begin
            vga_data <= welcome_pic;
        end
        else if (state == 2'b01)  begin
            if (vc >= row_pos && vc <= row_pos + 32 && hc >= col_pos && h
c <= col_pos + 32) begin
                figure_pos <= (vc - row_pos) * 32 + hc - col_pos;
                if (figure_pic == 11'b0)
                    vga_data <= background_pic;
                else
                    vga_data <= figure_pic;
            end
            else begin
                vga_data <= background_pic;

                //  render the boxes
                for (i = 0; i < 8; i = i + 1) begin
```

```
                if (vc >= box_pos[(i * 10) +: 5] * 32 && vc < box_pos
[(i * 10) +: 5] * 32 + 32 &&
                    hc >= box_pos[(i * 10 + 5) +: 5] * 32 && hc < box_p
os[(i * 10 + 5) +: 5] * 32 + 32) begin
                        inbox_pos <= (vc - box_pos[(i * 10) +: 5] * 32)
 * 32 + hc - box_pos[(i * 10 + 5) +: 5] * 32;
                        if (box_pic == 11'b0 || box_ena[i] == 1'b0)
                            vga_data <= background_pic;
                        else
                            vga_data <= box_pic;
                end
            end
        end
    end
    else if (state == 2'b10) begin
        if (vc >= 211 && vc < 238 && hc >= 285 && hc < 356) begin
            back_pos <= (vc - 211) * 71 + hc - 285;
            vga_data <= back_pic;
        end
        else begin
            vga_data <= background_pic;
        end
    end

end


welcome m0(
    .clka(rgb_clk),
    .addra(pos),
    .douta(welcome_pic),
    .ena(1'b1)
);

background m1(
    .clka(rgb_clk),
    .addra(pos),
    .douta(background_pic),
    .ena(1'b1)
);

figure m2(
    .clka(rgb_clk),
    .addra(figure_pos),
    .douta(figure_pic),
    .ena(1'b1)
);

box m3(
    .clka(rgb_clk),
    .addra(inbox_pos),
    .douta(box_pic),
    .ena(1'b1)
);

back m4(
    .clka(rgb_clk),
    .addra(back_pos),
    .douta(back_pic),
    .ena(1'b1)
);
```

```
endmodule
```

调试过程

本次实验也并非一帆风顺，其中也遇到了许多的调试问题，接下来我对几个比较关键的调试环节进行说明：

ROM 图片的显示

在 ROM 图片的存储过程中，我一开始设计了很多张图片进行显示，但是在综合的过程中发生了 OOM 的报错，才意识到存储资源有限的问题。因此，我们要对能够复用的图片模块尽可能地减少重复存储。

另外，我在图片显示的上板调试过程中遇到了如下的显示问题：

一开始我怀疑是 ROM 出现了问题，但是当我元素化了更多的组件之后，这样的问题更加严重了，我才意识到可能是出现了时序上的扫描与显示问题。最终由于仍未解决这个 bug，我在图片显示上进行了一点妥协，好在并没有影响整体的显示效果。

七段数码管的显示

首先是进制转换。由于之前的实验中我们的显示都是十六进制的，用 4 位的位宽存储并不会产生问题，但是这里我们遵循约定采用十进制的分数显示。我们在进行分数运算和最后的 num 的过程中需要格外注意传入值和预期的进制。

另外，我在上板之后发现数码管显示出来的结果始终是 8888。之后，我对这一部分的代码进行了仿真，发现并没有问题。后来才意识到是引脚约束的问题。尽管我们对小数点的引脚并没有要求，但是在显示过程中必须要进行约束才能够显示出正确的结果。

PS2 键盘的输入

对外设的输入显示与调试还算幸运，第一次上板就成功得到了正确的反馈。但是一开始由于我对 PS2 的扫描信号的频率设置过高，导致按下按键之后人物会瞬移好多格。后来，对 clkdiv 进行了多次尝试之后我才找到合适的扫描频率，解决了这一问题。

代码实现

我在开发过程中遇到了两个关键的关于 verilog 代码的问题。

- `for loop`. 在 `verilog` 中书写循环很多时候是为了代码的可读性考虑，例如在游戏中对箱子的显示和隐藏是通过位置判断的，如果我们机械地对每个箱子都分开判断的话就会让代码变得冗长而且不易调试。后来，我发现了合适的循环写法。但是需要注意的是，我们对寄存器位宽的定位必须是定长的，例如 `box_pos[(i * 10) +: 5]` 的写法是正确的，而 `box_pos[(i * 10) : (i * 10 + 5)]` 这样的写法是不允许的。

- 另外需要注意的是，我们不能在两个 `always` 模块中同一寄存器的信息进行更改，可能会带来时序冲突的问题，因此我们需要尽量用条件判断代替寄存器存储，并善于进行合理的功能设计。

实现说明

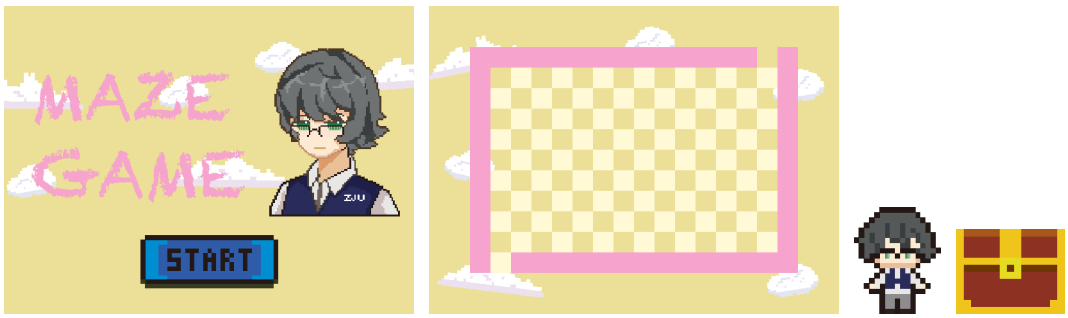
在完成上述环节之后，我们最终成功上板并实现了目标功能。

以下是我的约束文件：

```
# Main clock
set_property PACKAGE_PIN AC18 [get_ports clk]
set_property IOSTANDARD LVCMOS18 [get_ports clk]

create_clock -period 10.000 -name clk [get_ports "clk"]

# ps2
set_property PACKAGE_PIN N18 [get_ports ps2_clk]
set_property IOSTANDARD LVCMOS33 [get_ports ps2_clk]
set_property PACKAGE_PIN M19 [get_ports ps2_data]
set_property IOSTANDARD LVCMOS33 [get_ports ps2_data]

# Switches
set_property PACKAGE_PIN AA10 [get_ports {SW}]
set_property IOSTANDARD LVCMOS15 [get_ports {SW}]

# VGA
set_property PACKAGE_PIN N21 [get_ports {r[0]}]
set_property PACKAGE_PIN N22 [get_ports {r[1]}]
set_property PACKAGE_PIN R21 [get_ports {r[2]}]
set_property PACKAGE_PIN P21 [get_ports {r[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {r[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {r[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {r[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {r[3]}]
set_property PACKAGE_PIN R22 [get_ports {g[0]}]
set_property PACKAGE_PIN R23 [get_ports {g[1]}]
set_property PACKAGE_PIN T24 [get_ports {g[2]}]
set_property PACKAGE_PIN T25 [get_ports {g[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {g[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {g[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {g[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {g[3]}]
set_property PACKAGE_PIN T20 [get_ports {b[0]}]
set_property PACKAGE_PIN R20 [get_ports {b[1]}]
set_property PACKAGE_PIN T22 [get_ports {b[2]}]
set_property PACKAGE_PIN T23 [get_ports {b[3]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {b[3]}]
set_property PACKAGE_PIN M22 [get_ports hs]
set_property PACKAGE_PIN M21 [get_ports vs]
set_property IOSTANDARD LVCMOS33 [get_ports hs]
set_property IOSTANDARD LVCMOS33 [get_ports vs]

# Arduino-Segment & AN
set_property PACKAGE_PIN AD21 [get_ports {AN[0]}]
set_property PACKAGE_PIN AC21 [get_ports {AN[1]}]
set_property PACKAGE_PIN AB21 [get_ports {AN[2]}]
set_property PACKAGE_PIN AC22 [get_ports {AN[3]}]
set_property PACKAGE_PIN AB22 [get_ports {SEGMENT[0]}]
set_property PACKAGE_PIN AD24 [get_ports {SEGMENT[1]}]
set_property PACKAGE_PIN AD23 [get_ports {SEGMENT[2]}]
set_property PACKAGE_PIN Y21 [get_ports {SEGMENT[3]}]
set_property PACKAGE_PIN W20 [get_ports {SEGMENT[4]}]
set_property PACKAGE_PIN AC24 [get_ports {SEGMENT[5]}]
set_property PACKAGE_PIN AC23 [get_ports {SEGMENT[6]}]
set_property PACKAGE_PIN AA22 [get_ports {SEGMENT[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SEGMENT[7]}]

# Present output on LED of arduino
# set_property PACKAGE_PIN AF24 [get_ports {res}]
# set_property IOSTANDARD LVCMOS33 [get_ports {res}]
```

以下是游戏的一些基本界面和基本元素：



具体的实现过程将在展示视频中详细说明。

注：本实验中 VGA 和 PS2 的部分驱动代码借鉴自瓜豪学长的实验文档。

体会与心得

终于在摸爬滚打中结束了这个学期的数逻实验。从一开始对 verilog 的一脸懵逼到现在终于能写出一点像样的代码，这个学期的还是收获到了很多很多。在一开始布置大程而且是 solo 的时候完全不知道如何下手，后来一点一点看了一些 demo、自己做了很多的尝试之后才终于写出一个有一定功能的小游戏。虽然最后的大程还是有些粗糙，但自己亲手从零能够把它搭起来也还是很有成就感的。回头看来，原来看似不可能的任务其实没有想象中的那么复杂。

最后，要特别感谢各位助教老师一个学期的实验指导，感谢蔡爹的用心教学，感谢瓜豪学长的实验文档带我跌跌撞撞用 vivado 撑了下来。

那么下个学期计组再见~