浙江大学

# 课程设计报告

**中文题目：基于数字系统的“高空坠落”游戏设计**

**英文题目：BuildingDrop**


作者：朱理真

学号：3190101094

专业：计算机科学与技术

学院：计算机科学与技术学院

指导教师：蔡铭


**论文提交日期 2021 年 01 月 21 日**

# 基于数字系统的“高空坠落”游戏设计

摘要

本课程设计题为“高空坠落”，是基于现场可编程阵列，VGA 接口，ps2 键盘，并用 Verilog 语言编写的一款原创简单游戏。课程设计详细介绍了设计该项目所需的知识背景，以及对项目的基本介绍，包括整体架构以及状态机的状态图，将游戏分为不同的场景进行交互，使游戏的交互功能较为完善。还有各个模块的组成介绍，详尽地给出了各个模块的算法和输入输出规范。同时还包括仿真与调试过程。最后，还给出了基于设计和调试过程的体会心得和经验教训，给出了设计这一课题所得到的收获和总结。

关键词

数字逻辑，坠落物，躲避，现实场景，Verilog，IP 核，ps2 键盘

## 一、知识背景

### 1.VGA

VGA 接口，即视频图像阵列（Video Graphics Array）接口，通过显卡输出模拟信号，使液晶屏输出图像。VGA 有 15 个针孔，分成三排，如图 1.1。

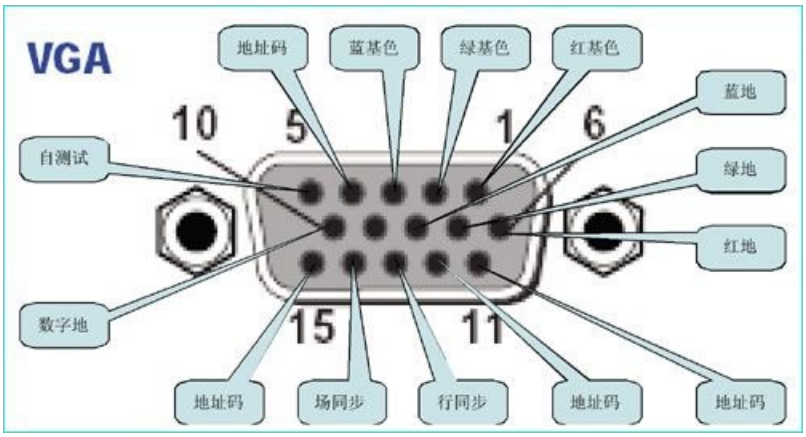


图 1.1

对于该实验，只需要红、绿、蓝三个通道的颜色信号个 4 位、以及同步信号 HS、VS 即可。VGA 采用一个阴极射线管扫描屏幕生成图像的模型，一行一行地从左向右扫描，当扫完一行后，回到下一行最左边的位置。回去的过程中电子并不显示在屏幕上，称为“行消隐”。当扫完所有行后，扫描信号就要回到左上角的开头处，此期间也不会显示图像，称为“场消隐”。每次开始行消隐或场消隐时，VGA 接口都要输出一个同步信号，分布称为行同步信号和场同步信号，用于提示扫描信号回转。了解以上基本知识，便可开始 VGA 处理模块的编写。如图 1.2.

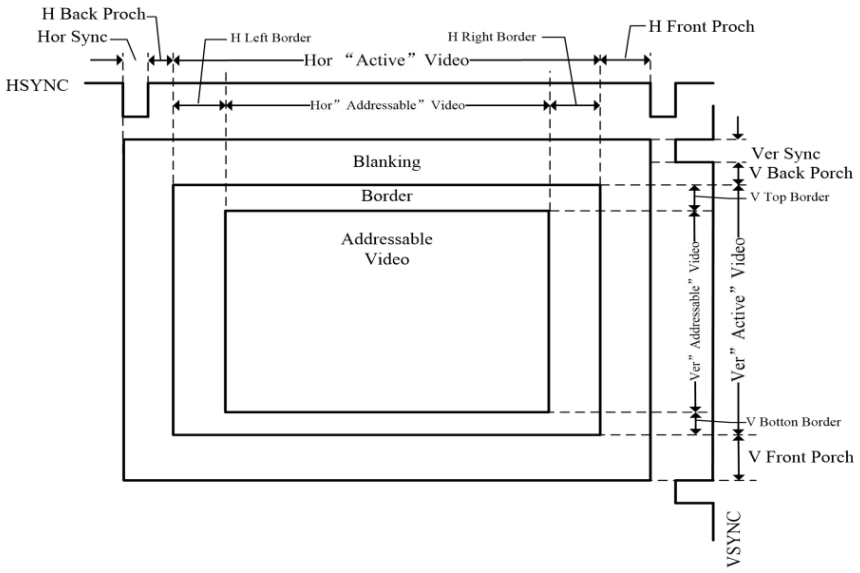


图 1.2

**2.PS2 键盘**

PS2 键盘采用 PS2 协议进行输入。PS2 键盘在有新的按键信号产生时，都会产生 ps2 时钟信号，同时输入 ps2 按键信息（一位）。当时钟的输入显示为下降沿时，即示意该数据可以读取。实际上，若时钟为上升沿，主机也可向 ps2 设备输出信息，不过此处不会用到。每次收到一个按键信号，ps2 实际至少会发出 11 次时钟信号（下降沿），传达 11 位数据，表达一个键值信息。其中第一位是示意开始，必为 0；之后的 8 位是具体的键值信息（可能是一个按键的通码，也可能表示断码或者扩展键位），接着一位是 8 位数据加上开始位的校验码，最后一位是结束信号，必为 1。这 11 位构成了一个按键信号。实际上，当一个按键变动时，最多可能发出三个按键信号，若按下/松开的按键是扩展按键，则首先收到的是扩展按键信息；排在其次的是断码信号（如果有），表示键盘动作为松开而不是按下；最后是每个按键独特的通码信号，显示什么键被按下。了解了这些相关知识，便可设计出一个 ps2 键盘处理模块。具体如表 1.

表 1 输入数据格式（12 位不用）

| 数据 | 含义 |
| --- | --- |
| 1 个起始位 | 总是逻辑 0 |
| 8 个数据位 | (LSB)地位在前 |
| 1 个奇偶校验位 | 奇校验 |
| 1 个停止位 | 总是逻辑 1 |
| 1 个应答位 | 仅用在主机对设备的通信中 |

**3.硬件描述语言**

Verilog HDL 是一种硬件描述语言（HDL:Hardware Description Language），以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，使用 Verilog 描述硬件的基本设计单元是模块（module）。构建复杂的电子电路，主要是通过模块的相互连接调用来实现的。模块被包含在关键字 module、endmodule 之内。实际的电路元件。Verilog 中的模块类似 C 语言中的函数，它能够提供输入、输出端口，可以实例调用其他模块，也可以被其他模块实例调用。
还 可以表示数字逻辑系统所完成的逻辑功能。

**4.可编程阵列逻辑**

PAL 器件由可编程的与阵列、固定的或阵列和输出反馈单元组成。不同型号 PAL 器件有不同的可编程阵列逻辑输出和反馈结构，适用于各种组合逻辑电路和时序逻辑电路的设计，是一种可程式化的装置。PLA 具有一组可程式化的 AND 阶，AND 阶之后连接一组可程式化的 OR 阶，如此可以达到：只在合乎设定条件时才允许产生逻辑讯号输出。PLA 如此的逻辑闸布局能用来规划大量的逻辑函式，这些逻辑函式必须先以积项（有时是多个积项）的原始形式进行齐一化。在 PLA 的应用中，有一种是用来控制资料路径， 在指令集内事先定义好逻辑状态，并用此来产生下一个逻辑状态（透过条件分支）。 举例来说，如果目前机器（指整个逻辑系统）处于二号状态，如果接下来的执行指令中含有一个立即值（侦测到立即值的栏位）时，机器就从第二状态转成四号状态，并且也可以进一步定义进入第四状态后的接续动作。

# 二、设计介绍

本游戏是基于 Verilog 硬件描述语言，利用 FPGA（现场可编程门阵列（field-programmable gate array）进行的实验设计。该游戏使用了 SWORD 板上的 ps/2 键盘和按钮进行输入，通过 VGA 接口在屏幕上，同时也在七段数码管上显示输出，从而实现人机交互。总体结构如图 2.1 所示。

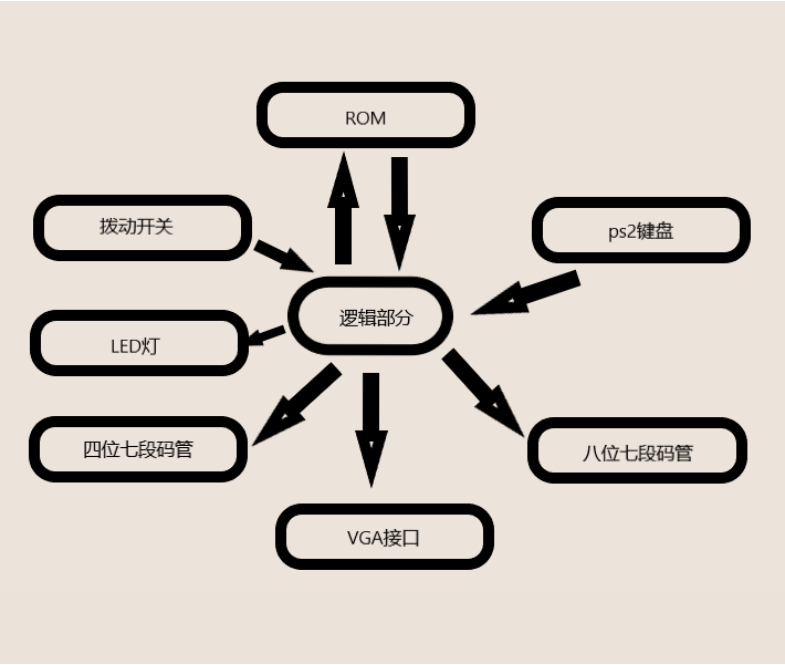


图 2.1 总体结构

本游戏设计根据游戏场景（scene），给出不同的交互响应。游戏主要有三个场景，即“开始界面”，“主界面”和“结束界面”。三个界面的相互转换形成了一种初步的状态机。如图 2.2，再非调试状态下，当处于开始界面时，按下并松开任意键可以进入主界面开始游戏；游戏人物失败后，进入失败界面；失败界面在按下并松开任意键即可进入开始界面。

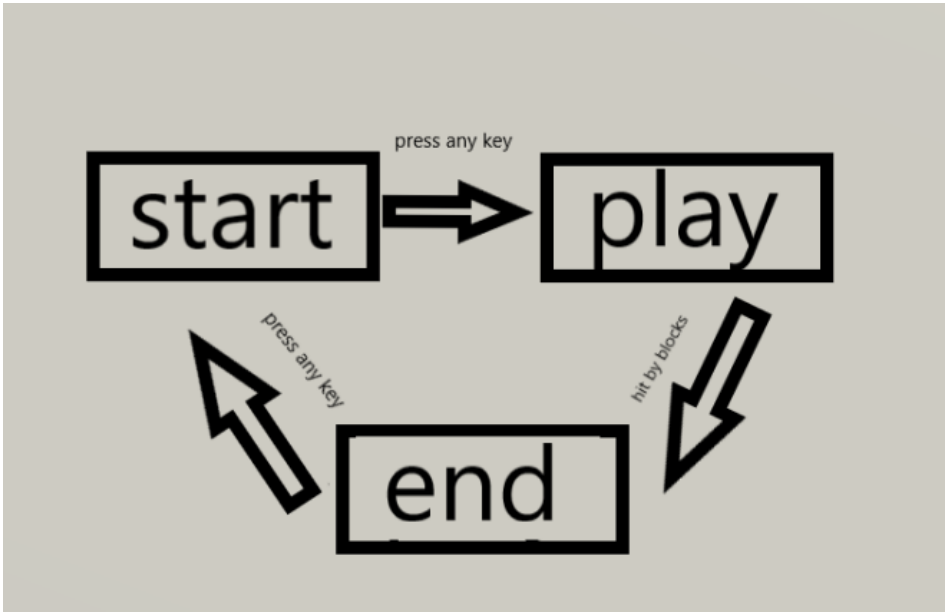


图 2.2 场景与场景之间的转换

除了场景的转换，该设计的主要工作还是集中在对游戏的操作上。游戏的主要内容是，通过移动键盘的左右键使人物移动，躲避空中落下的掉落物。值得一提的是，本游戏为增加挑战性，将左右键的操作互换，使人物需要右键才能往左，左键才能往右。另外，本游戏将人物的位置限定在给定的四个位置上，当处于左边缘或右边缘时，可以继续往左/右回到右/左边缘，实现特殊的移动。

但是，当玩家不慎被下落物击中，则游戏结束，进入结束界面。总体见图 2.3。

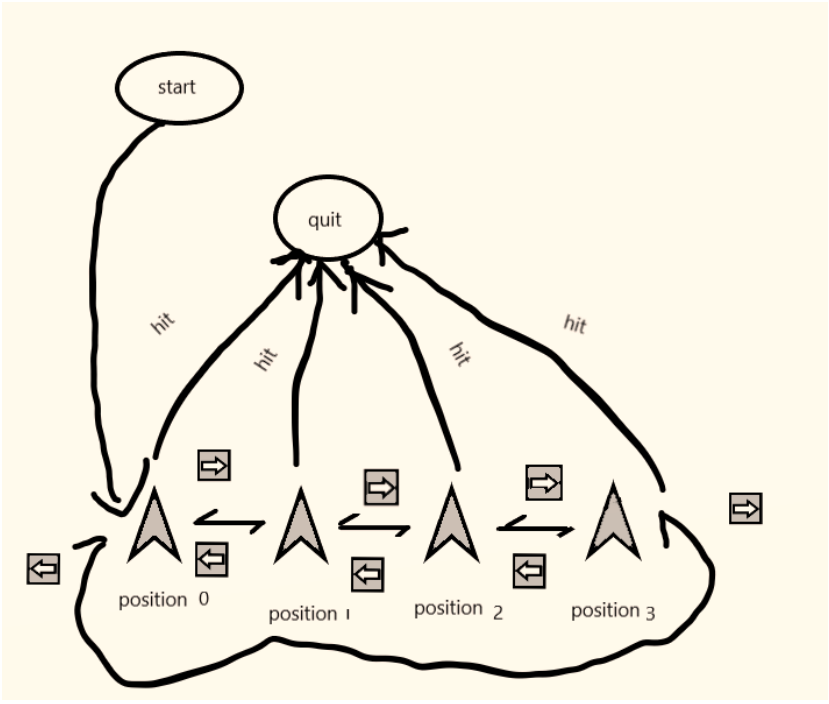


图 2.3 游戏状态机（左右颠倒+循环设计）

分数记录：

分数即为每次游戏的游戏时间，显示在 4 位七段数码管上。每次回到开始界面，分数都会清零。

调试部分：

在游戏的设计过程中，由于遇到了一些问题，进行了输出调试，设计了额外的一些输入输出功能，如键盘的禁用，场景的强制切换以及 VGA 接口的重启。同时。事实上，游戏启动时，必须让所有拨动开关置 0，以实现初始化。具体分布如图 2.4

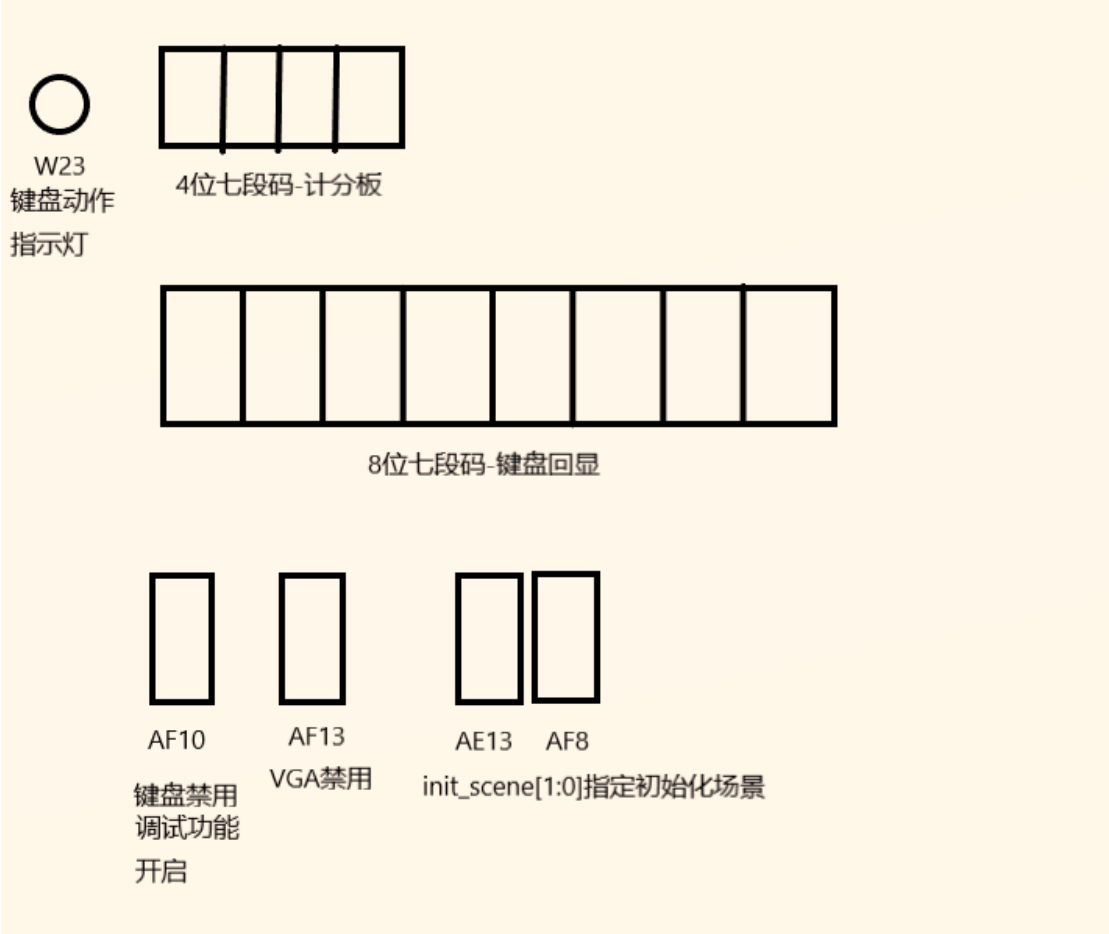


图 2.4 交互界面（除 vga 屏）各部分显示

# 三、设计实现

## 1.工具

SWORD 板

ISE14.7

ISE 是使用 XILINX 的 FPGA 的必备的设计工具。它可以完成 FPGA 开发的全部流程，包括设计输入、仿真、综合、布局布线、生成 BIT 文件、配置以及在线调试等，功能非常强大。ISE 除了功能完整，使用方便外，它的设计性能也非常好，以集成的时序收敛流程整合了增强性物理综合优化，提供最佳的时钟布局、更好的封装和时序收敛映射，从而获得更高的设计性能。

画图(用于绘制图形背景，以及编辑图像)
Visual Studio Code + mingw-gcc（用于编写 Verilog 程序和 bmp 转 coe 文件程序）

## 2.素材

**背景图**（网上搜集+自己加工）


图 3.1 开始界面

图 3.2 主界面


图 3.3 结束界面

**道具图**（原创）

图 3.4 人物

图 3.5 掉落物

## 3.逻辑部分说明

**头部模块 MyTop：**

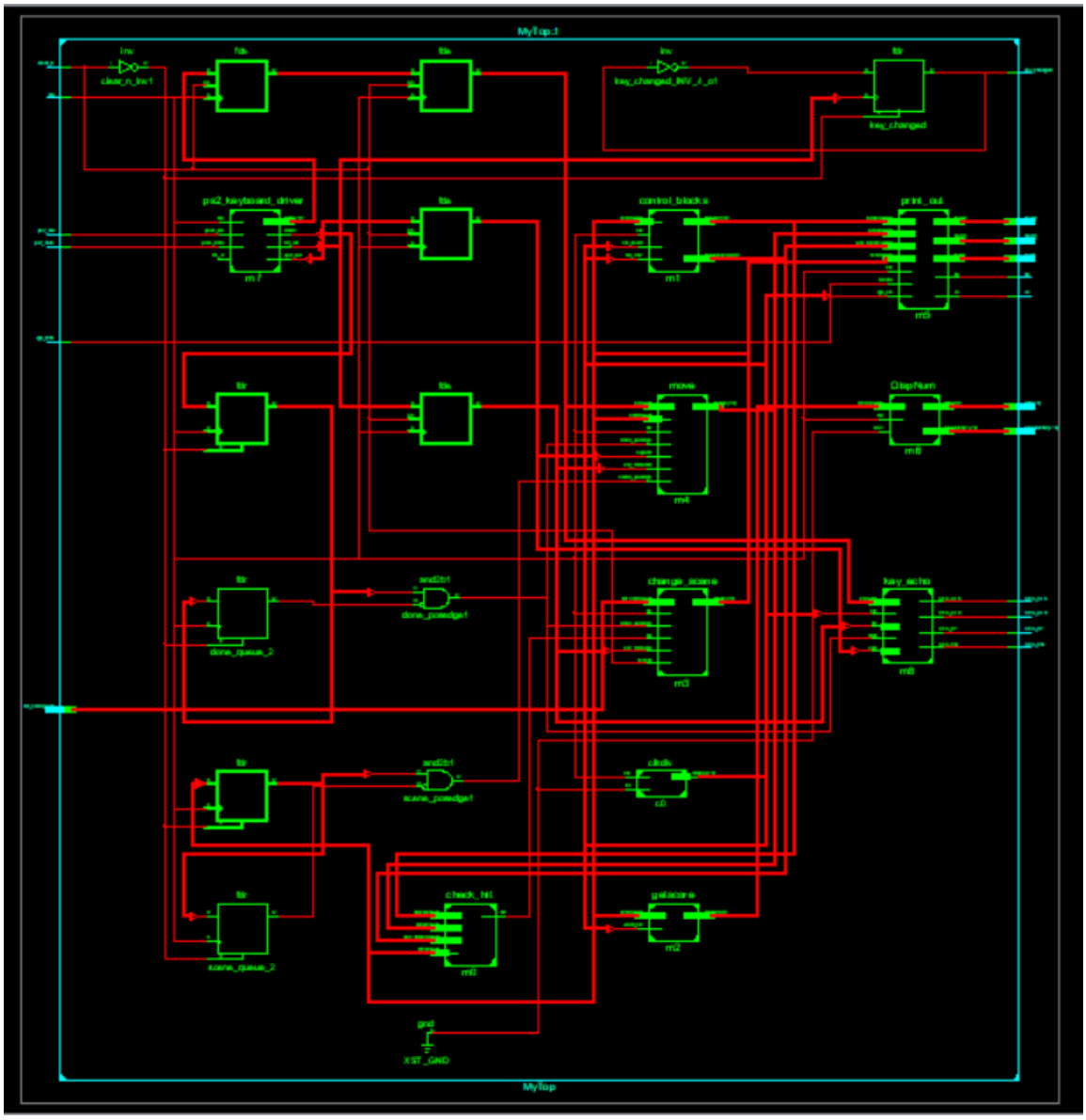


图 3.6 MyTop 模块缩略图

输入（如图 3.7 左）:

时钟信号（100mhz）
键盘清零信号
屏幕清零信号
场景初始化设置
键盘时钟信号
键盘数据信号

输出（如图 3.7 右）:

键盘改变信号
4 位、8 位七段数码管输出
vga 接口输出

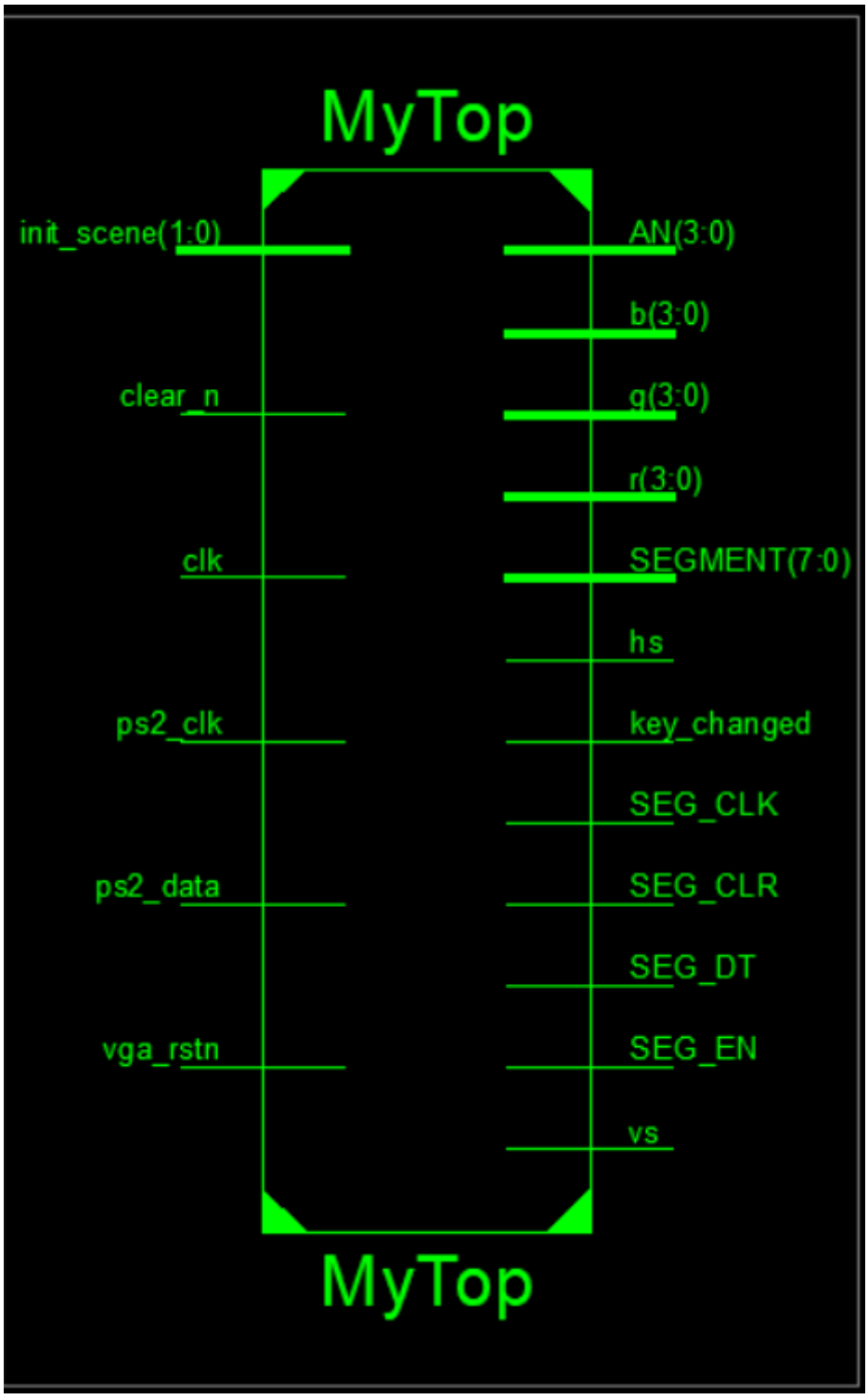


图 3.7 top 模块接口

调用的模块:

clkdiv（分频模块）

check_hit（检测碰撞）

control_blocks（坠落物运动控制）

getscore（分数记录）

change_scene（场景控制）

move（人物控制）
print_out（屏幕显示）
DispNum（分数显示）
ps2_keyboard_driver（键盘输入管理）
key_echo（键盘信息回显）

结构说明

Top 模块的主要作用是连接各个模块以及输入输出，同时对一些信号进行处理，起到统领全局的效果。

在连接输入输出方面，将 clk 时钟连接至各个需要时钟的模块（除了 check_hit，和 control_blocks）；同时将各输入输出与对应的处理模块相连接，如 print_out 模块与 r、g、b 颜色信号以及行同步、帧同步信号连接。

在连接各个模块方面，比较典型的就是场景信号 scene 的传送。scene 作为 change_scene 的输出，同时是 check_hit, control_blocks, get_score 和 move 等模块的输入，这是因为不同的游戏场景有不同的作用方式，因此该信号用途广泛。

同时，一些对信号的处理是必要的。但又由于代码行数较少，决定不封装为独立的模块。信号的处理主要指的是对脉冲信号的处理。很多信号会发生变化，如键盘输出的 done 信号，它为真代表一个键盘信号已经准备完毕，可以输出。可是，我们并不能很快使该信号重新变为非真，因为拿不准什么时候才读取该信号。而 done 信号一直为真也是令人困扰的，因为如果以 done 信号为标准接收键盘数据输出，模块就会一直收到键盘的数据，这个数据是之前，因此并不符合设计思路。因此，我仿照了 ps2 协议的方式，设计了一个 done 信号的队列，每次时钟正边沿才左移一次，以接收新的 done 信号。因此，可以比较老的 done 信号与新的 done 信号，当 done 信号由低到高上升时，就预示着新键盘输入的来到，我们就可以做出正确的反应了。

**check_hit 模块**

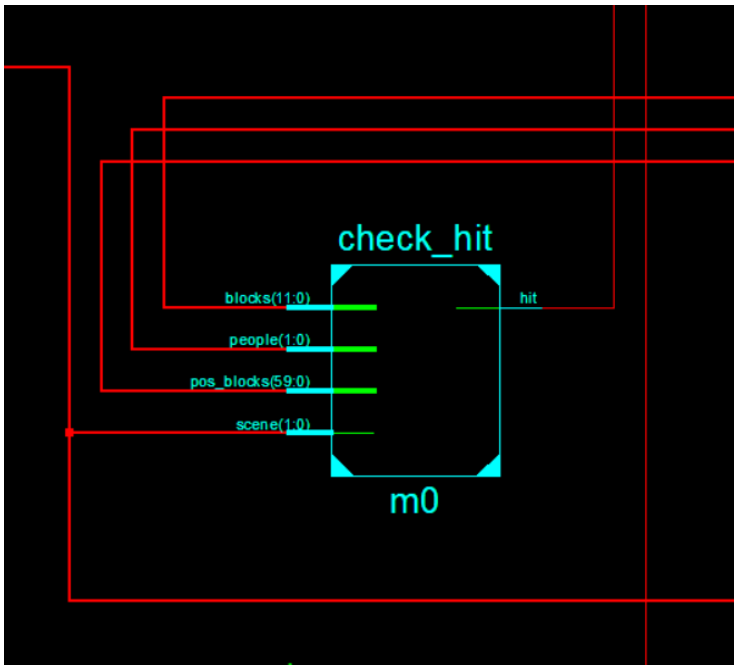


图 3.8 碰撞检测

该模块的设计是简单的，只需分别检测各个坠落物是否和人物处于同一高度，并且和人物处于同一水平坐标即可。

**control_blocks 模块**

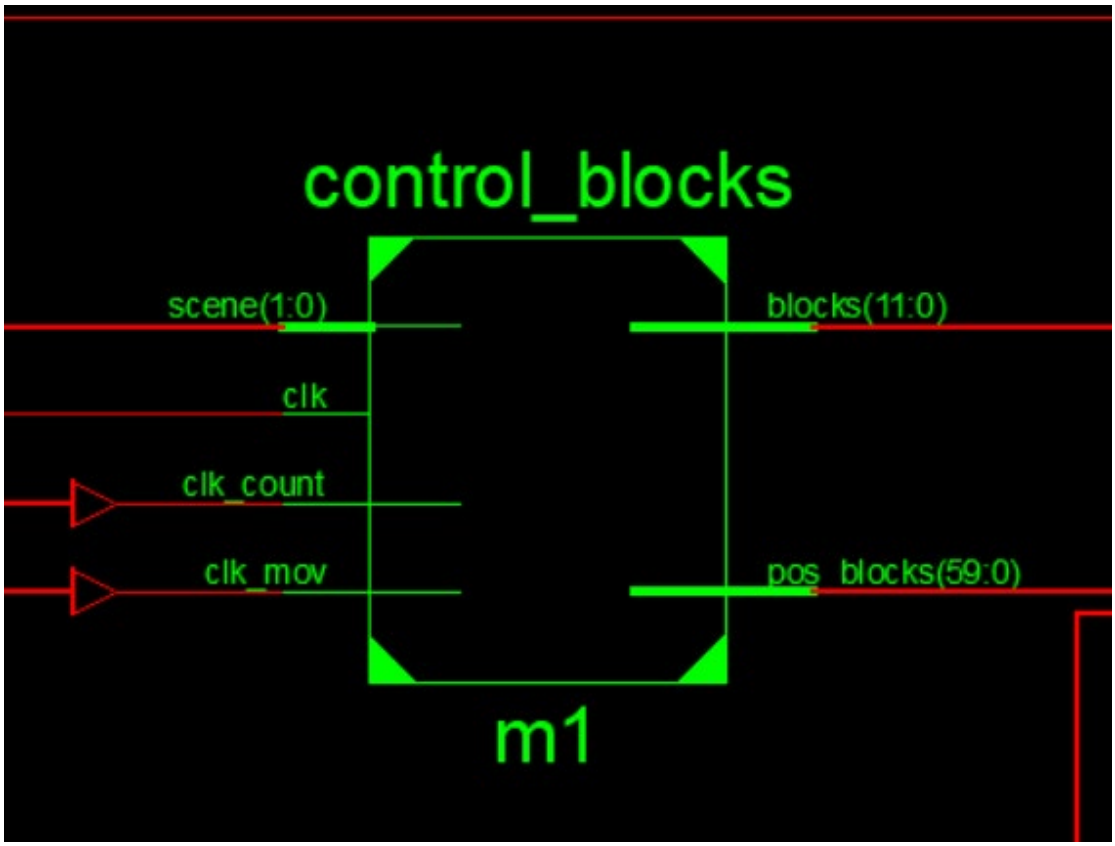


图 3.9 坠落物控制

该模块调用了随机数模块。是通过移位寄存器实现的，稍后会介绍。

该模块控制了 6 个坠落物的横纵坐标，当坠落物在下落过程中，通过接收下落时钟“clk_mov”来进行合适速度的匀速下落，直至落地或游戏场景切换（一般是由于坠落物撞倒游戏角色造成的）。

坠落物落地以后，会将坠落物坐标设置到不可见处，以实现坠落物“粉碎”效果。经过一段时间以后，该坠落物会再次从上空落下。考虑的游戏的可玩性，坠落物的从落地到再次下落的时间间隔是随机的，具体来说，使用 3 位随机数来控制，并用“clk_count”来计数。决定再次下落时，坠落物还会随机改变其横坐标，改道下落。

**随机数模块 rand_generator**

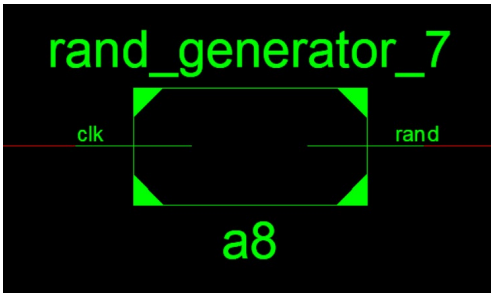


图 3.10 随机数生成接口

如上所说，该模块使用了移位寄存器进行随机数生成，具体如下（图源蔡铭老师的 ppt term review p30）

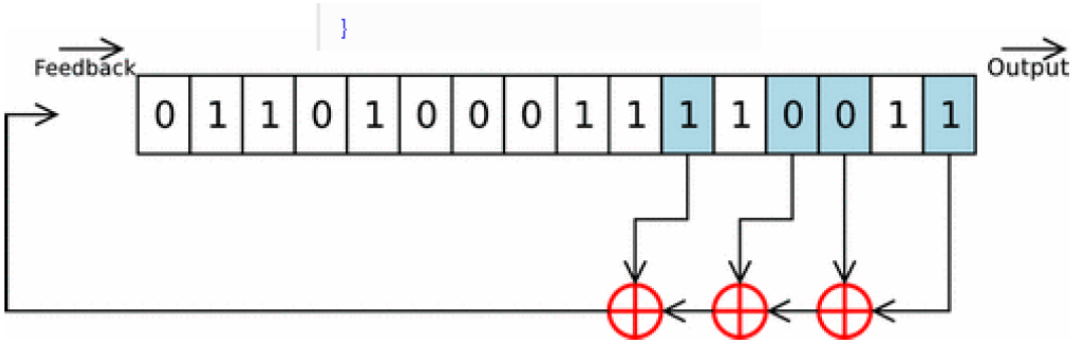


图 3.11 随机数实现原理

为保证随机数的多样性，我们选取了多个种子，创建了多个模块进行随机数的生成。

**场景控制模块 change_scene**

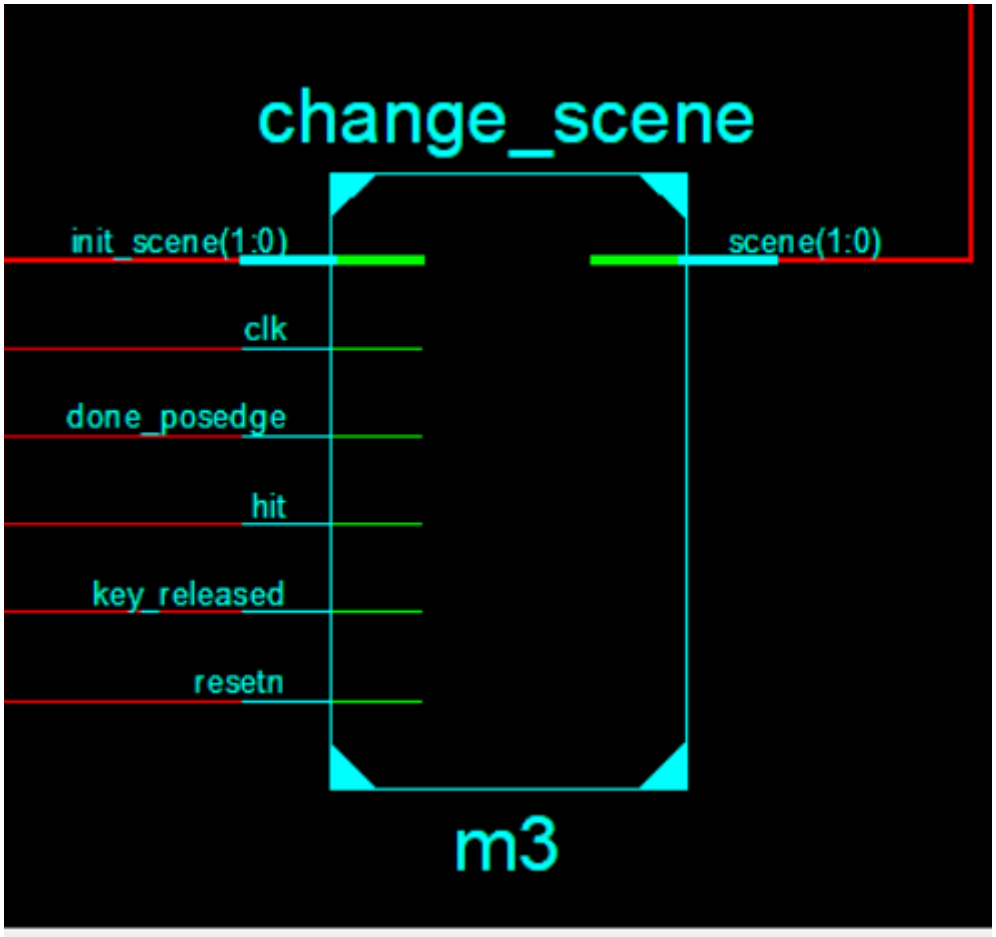


图 3.12 场景控制模块接口

当 resetn=0 时，场景处于调试状态，scene 完全由 init_scene 决定；当 resetn 信号失效时，scene 信号由键盘信号以及碰撞信号决定。当处于开始场景，收到任意松开键盘信号进入游戏场景；处于游戏场景时，若收到碰撞信号，进入失败界面；失败界面若收到键盘松开信号，进入开始场景。

**人物控制模块 move**

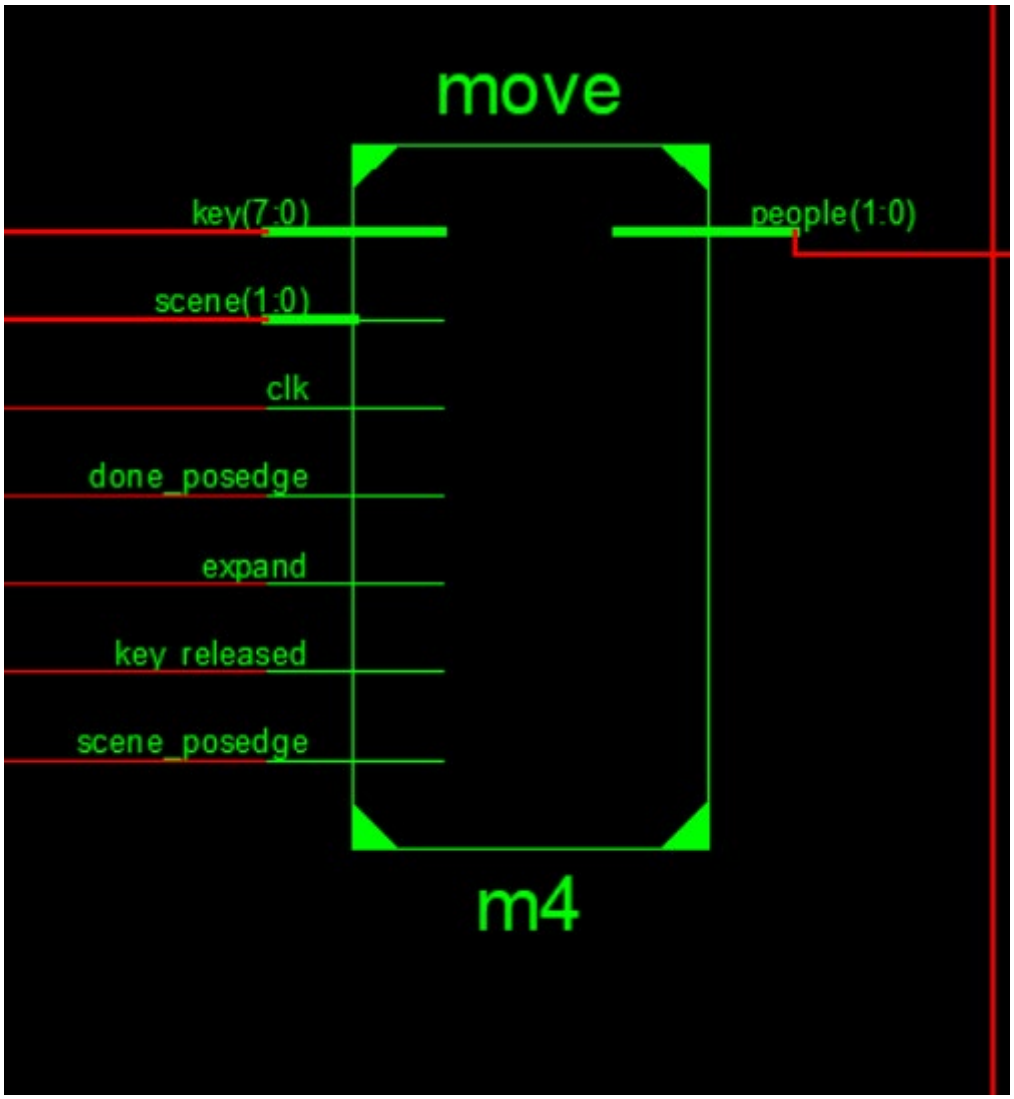


图 3.13 人物控制模块

人物模块收到各类控制信息，输出人物坐标信息。

当处于非游戏场景时，人物坐标处于初始化状态，即最左边；当处于游戏状态中，人物位置由键盘信号控制，若收到左方向键信号则向右运动，收到右方向键信号则向左运动。考虑二进制整数的有界性，人物处于最左位置再往左运动后会回到最右位置，反之亦成立。

**显示模块 print_out**

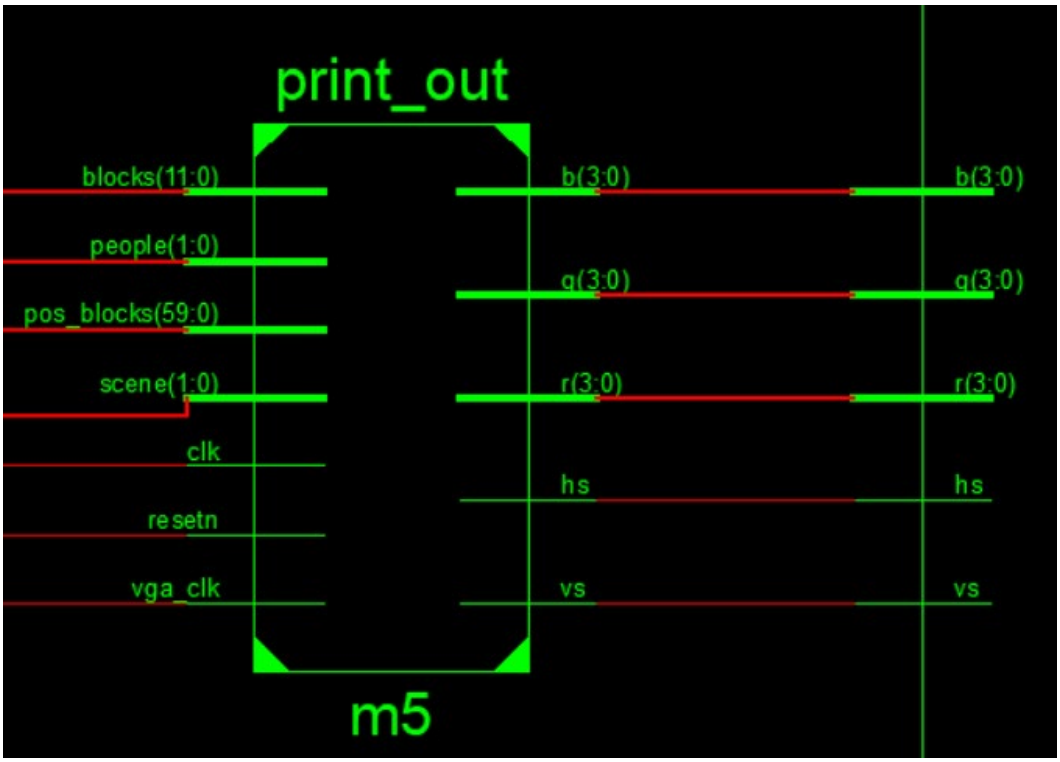


图 3.14 显示模块

显示模块调用了 vgac 模块，进行颜色信号的输入输出。由于 vgac 会给出扫描信号的位置信息，模块利用该位置信息，根据其所处的范围给予不同的输入，从而得到符合设计的图像。比如若扫描坐标再人物图形范围内，则根据其偏离人物中心的位置给出相应的图片地址，输入到 rom 中，再将 rom 输出的图片信息输入到 vga 信号中，最终显示在屏幕上。

**键盘处理模块 ps2_keyboard_driver**

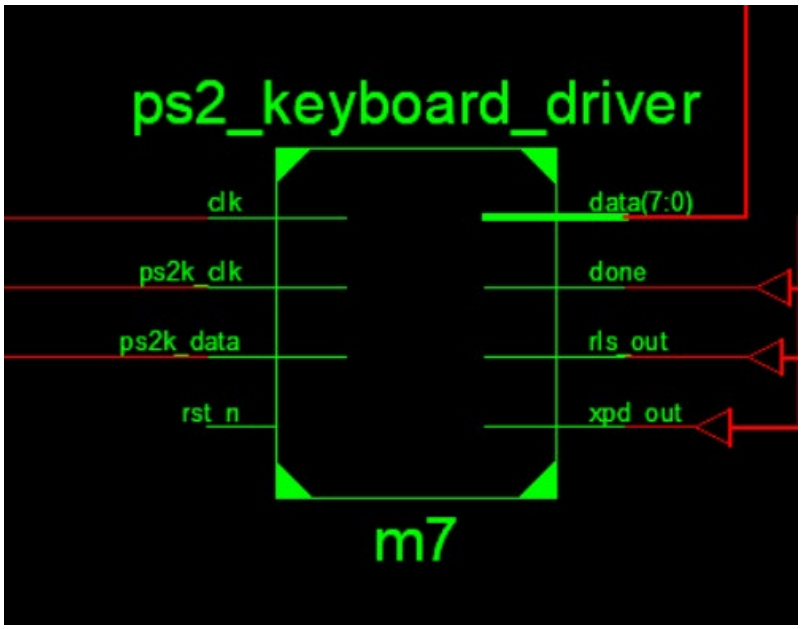


图 3.15 键盘处理模块

键盘处理模块是我参照接口文件的键盘调用模块根据自己的需求重新设计的键盘处理模块，思路与 ps2 协议基本一致，只是得到一个完整的编码字节后并不马上往外输出，而是判断其是否为扩展信号或断码信号，若是，则先记录其状态，等到有效的键盘信号出现后一起输出，同时 done 置 1，表示处理完成；而信号处理过程中，done 会置 0，为下次的完成并达到上升沿做准备。

**键盘回显模块 key_echo**

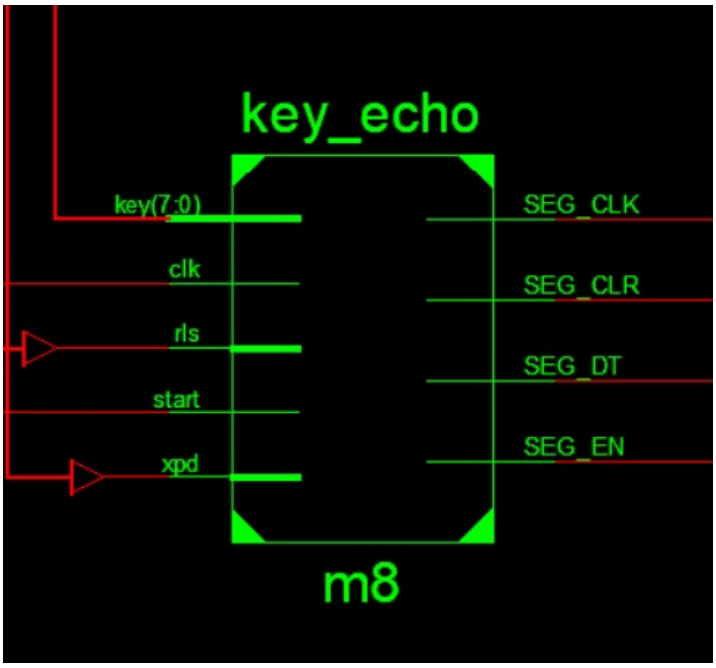


图 3.16 键盘回显模块

该模块通过简单地接收键盘信息，再调用 SEG_DRV 模块，实现键盘信息到八位七段数码管的显示。其中，扩展键盘和断码信号均用 1 位显示，按键信息用 8 位表示。

**分数记录模块 getscore**

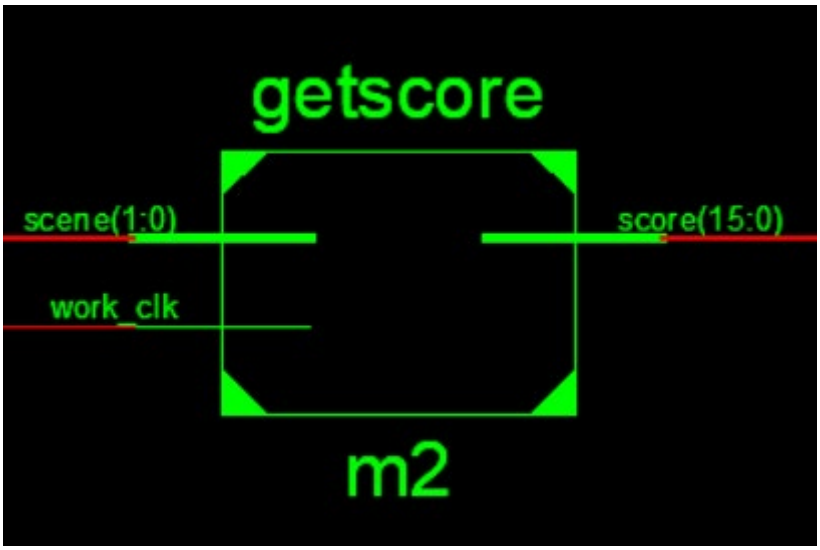


图 3.17 分数记录模块

该模块接入计时时钟进行工作。在开始场景时，分数清零；游戏场景，分数随时间的增加而增加；结束场景，分数保持，便于观察。

ROM 模块与 IP 核生成

IP 核生成：通过上述图片素材，结合设计的 C 代码，生成 coe 文件，分别生成各个 ROM 模块。ROM 模块以地址信号为输入，图片像素信号为输出，既具有储存图片功能，又实现了将控制信号转化为具体的图片信息的功能，说显示模块的重要组成部分。

生成文件的程序具体如下（该代码吸收了优秀报告的经验，并在其基础上改进，既可以接收 24 位 bmp 文件，也可以接受 32 位 bmp 文件，同时可以通过命令行参数一次处理多个图片文件生成多个 coe 文件（当然也可以合在一个 coe 文件中）：

```c
//Bmp24or32toCoe.c
#include<windows.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<assert.h>

#define TRUE 1
#define FALSE 0
#define MAXVECTORSINALINE 16
#define MAXFILENAMELEN 80

void bmp_to_coe(char bmp_name[],char coe_name[]);
int bmp_to_vector(char filename[],FILE* out_fp);
void postfix_change(char* src,char* dst,char* postfix);

int main(int argc,char* argv[]){
    int i=1;
    char coe[MAXFILENAMELEN];

    for(;i<argc;i++){
        postfix_change(argv[i],coe,"coe");
        bmp_to_coe(argv[i],coe);
    }

    system("pause");
    return 0;
}

void postfix_change(char* src,char* dst,char* postfix){
    strcpy(dst,src);
    char* p=strstr(dst,".");
    if(p) strcpy(p+1,postfix);
    else sprintf("%s.%s",dst,postfix);
}
```

```
void bmp_to_coe(char bmp_name[],char coe_name[]){
    FILE* outfp = fopen(coe_name,"w");
    assert(outfp);

    fprintf(outfp,"memory_initialization_radix=16;\nmemory_initializati
on_vector =\n");

    bmp_to_vector(bmp_name,outfp);

    fprintf(outfp,";");

    fclose(outfp);
}

int bmp_to_vector(char filename[],FILE* out_fp){
    FILE* in_fp;
    static int vectors_in_a_line=0;
    int i,j;
    char bm[3]={'\0'};

    BITMAPINFOHEADER head;

    in_fp=fopen(filename,"rb");
    if(!in_fp){
        printf("file %s not found\n",filename);
        return FALSE;
    }
    else printf("In the file %s:\n",filename);

    fread(bm,sizeof(bm),1,in_fp);

    fseek(in_fp,sizeof(BITMAPFILEHEADER),SEEK_SET);
    fread(&head,sizeof(BITMAPINFOHEADER),1,in_fp);

    int bit_count=(int)head.biBitCount;
    int picwidth = abs(head.biWidth), picheight = abs(head.biHeight);

    if(!strcmp("BM",bm) || bit_count!=32 && bit_count!=24){
        printf("This picture is not the bmp file of 32 or 24 bits\n");
        return FALSE;
    }

    printf("The sizeof picture is %d*%d\n",picwidth,picheight);
```

```
    fseek(in_fp,sizeof(BITMAPINFOHEADER)+sizeof(BITMAPFILEHEADER),SEEK_
SET);

    unsigned char r,g,b,t;

    for(i=0;i<picheight;i++){
        for(j=0;j<picwidth;j++){
            fread(&b,sizeof(unsigned char),1,in_fp);
            fread(&g,sizeof(unsigned char),1,in_fp);
            fread(&r,sizeof(unsigned char),1,in_fp);
            fprintf(out_fp,"%x%x%x",b>>4,g>>4,r>>4);

            if(i== picheight -1 && j==picwidth-1);
            else fprintf(out_fp,",");
            vectors_in_a_line++;

            if(vectors_in_a_line==MAXVECTORSINALINE)
                fprintf(out_fp,"\n"),vectors_in_a_line=0;

            if(bit_count==32) fread(&t,sizeof(unsigned char),1,in_fp);//
/alpha
        }
    }

    fclose(in_fp);
    return TRUE;
}
```

## 四、调试与仿真过程

### 仿真

**1. get_score 模块仿真**

仿真代码如下:

```
module sim_get_score;

    // Inputs
    reg [1:0] scene;
    reg work_clk;

    // Outputs
    wire [15:0] score;
```

```
    // Instantiate the Unit Under Test (UUT)
    getscore uut (
        .scene(scene),
        .work_clk(work_clk),
        .score(score)
    );

    initial begin
        // Initialize Inputs
        scene = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        scene=1;
        #500;
        scene=2;
        #200;
        scene=0;
    end

    always begin
        work_clk<=1'b0;
        #20;
        work_clk<=1'b1;
        #20;
    end

endmodule
```

仿真结果:

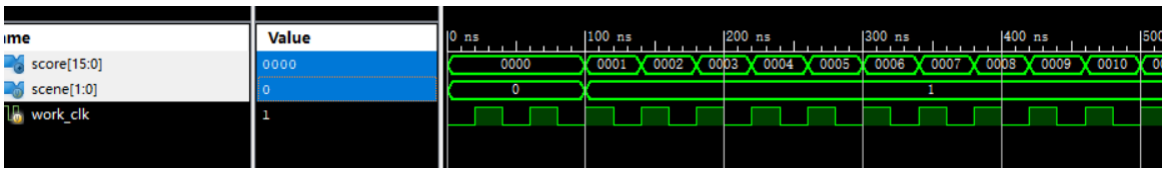


图 4.1 get_score 仿真波形

**2. move 模块仿真**

仿真代码如下

```
module sim_move;

    // Inputs
```

```
    reg clk;
    reg [1:0] scene;
    reg scene_posedge;
    reg [7:0] key;
    reg key_released;
    reg done_posedge;
    reg expand;

    // Outputs
    wire [1:0] people;

    // Instantiate the Unit Under Test (UUT)
    move uut (
        .clk(clk),
        .scene(scene),
        .scene_posedge(scene_posedge),
        .key(key),
        .key_released(key_released),
        .done_posedge(done_posedge),
        .expand(expand),
        .people(people)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        scene = 0;
        scene_posedge = 0;
        key = 0;
        key_released = 0;
        done_posedge = 0;
        expand = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        scene<=1;
        scene_posedge<=1;
        #40;
        scene_posedge<=0;
        #120;
        expand<=1;
        #40;
```

```
        key_released<=1;
        #40;
        key<=8'h6b;
        done_posedge<=1;
        #40;
        done_posedge<=0;
    end

    always begin
        clk<=0;
        #20;
        clk<=1;
        #20;
    end
endmodule
```

仿真波形:

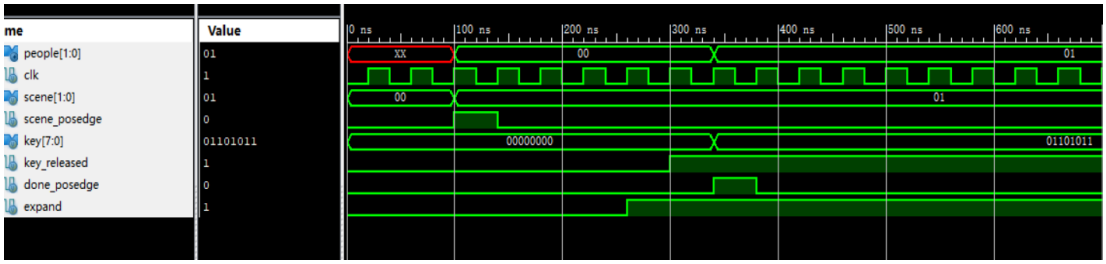


图 4.2 move 仿真波形

**3. print_out 模块仿真**

仿真代码如下

```
module sim_print;

    // Inputs
    reg clk;
    reg vga_clk;
    reg resetn;
    reg [11:0] blocks;
    reg [59:0] pos_blocks;
    reg [1:0] people;
    reg [1:0] scene;

    // Outputs
    wire hs;
    wire vs;
    wire [3:0] r;
    wire [3:0] g;
    wire [3:0] b;
```

```
    // Instantiate the Unit Under Test (UUT)
    print_out uut (
        .clk(clk),
        .vga_clk(vga_clk),
        .resetn(resetn),
        .blocks(blocks),
        .pos_blocks(pos_blocks),
        .people(people),
        .scene(scene),
        .hs(hs),
        .vs(vs),
        .r(r),
        .g(g),
        .b(b)
    );

    initial begin
        // Initialize Inputs
        resetn = 1;
        blocks = 0;
        pos_blocks = 0;
        people = 0;
        scene = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

    end
    always begin
        clk<=0;
        vga_clk<=0;
        #10;
        clk<=1;
        #10;
        clk<=0;
        vga_clk<=1;
        #10;
        clk<=1;
      #10;
    end
endmodule
```

仿真波形如图：

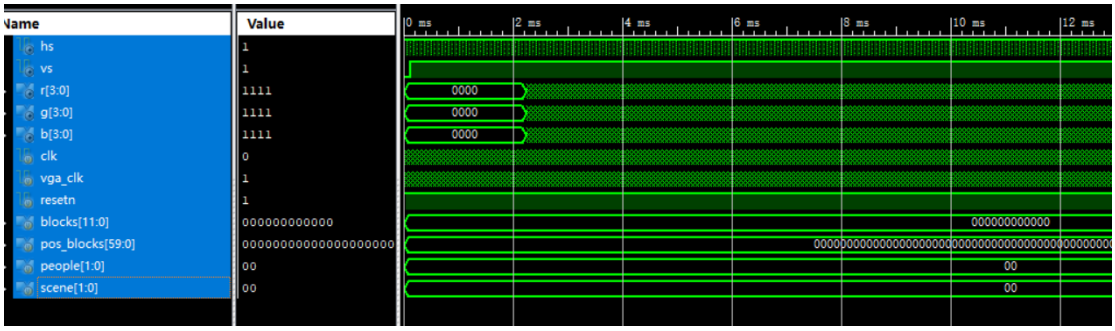


图 4.3 print_out 仿真波形

经分析得知，仿真运行正常：各个地址均正确传送，行消隐与侦消隐信号均可正确同步。

**4. ps2_keyboard_driver 模块仿真**

仿真代码如下：

```
module sim_ps2;

    // Inputs
    reg clk;
    reg rst_n;
    reg ps2k_clk;
    reg ps2k_data;

    // Outputs
    wire release_pressN;
    wire done;
    wire expand;
    wire [7:0] data;

    // Instantiate the Unit Under Test (UUT)
    ps2_keyboard_driver uut (
        .clk(clk),
        .rst_n(rst_n),
        .ps2k_clk(ps2k_clk),
        .ps2k_data(ps2k_data),
        .release_pressN(release_pressN),
        .done(done),
        .expand(expand),
        .data(data)
    );
    reg [10:0] rls;
    reg [10:0] A;
    integer cnt;
    initial begin
        // Initialize Inputs
```

```
        rst_n = 0;
        ps2k_clk = 0;
        ps2k_data = 0;
        rls={1'b1,1'b1,8'hf0,1'b0};
        A={1'b1,1'b0,8'h1c,1'b0};
        cnt=0;
        // Wait 100 ns for global reset to finish
        #100;
        rst_n<=1;
        #20;
      for(cnt=0;cnt<11;cnt=cnt+1)begin
            #80;
            ps2k_clk<=1;
            ps2k_data<=rls[cnt];
            #80;
            ps2k_clk<=0;
        end

        for(cnt=0;cnt<11;cnt=cnt+1)begin
            #80;
            ps2k_clk<=1;
            ps2k_data<=A[cnt];
            #80;
            ps2k_clk<=0;
        end
    end

    always begin
        clk<=0;
        #10;
        clk<=1;
        #10;
    end
endmodule
```

仿真波形:

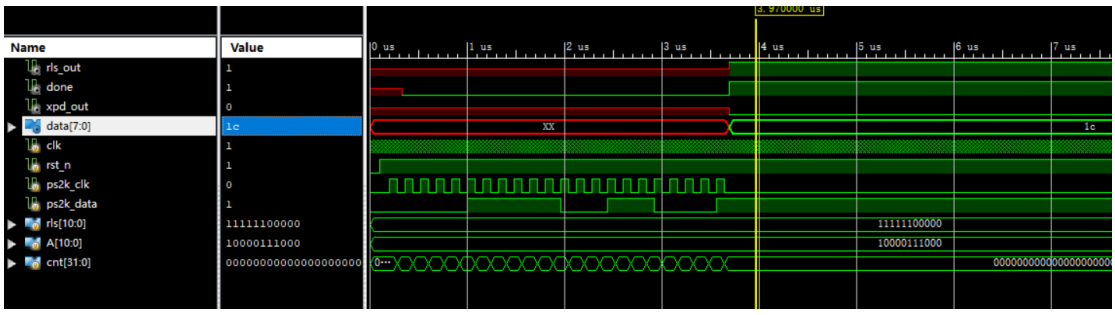


图 4.4 ps2_keyboard_driver 模块仿真波形

由仿真波形，可以看出随着断码和键值信息的接受完毕，模块成功处理并输出相应的键值信号“A”

## 调试

模块设计过程中，出现了很多的问题：
问题 1：图片乱码：
原因：在输入地址 row*640 时，将整数 640 的位宽设为 9(0-511)，导致其溢出（值为 640-512=128），不能使 rom 正确输出图片像素信息。
调试分析：

发现图像倒像，并且图片显示十分不正常。尝试解决倒像问题，即用总高度减去各个背景、人物和坠落物的纵坐标，以此算出图片地址。下板验证后，倒像问题解决，然而图片依然存在（尽管部分文字勉强可以辨认）。

怀疑是 vgac 模块出现问题，则尝试调试扫描像素的行、列坐标。
设置 vgac 输入条件，当列地址大于总列数一半时，显示黑屏。
下板结果显示左半边和右半边黑屏分割线为一条垂直直线，则列扫描信号正常。

同理，将列条件恢复正常，使行地址大于总行数一半时，显示黑屏。
结果发现两半边以水平直线分割，则行信号正常。

怀疑是图片转化为 coe 的过程中数据损坏，即自己写的转化程序不正确。
另外新建一个模块，加入新的图片的 coe 文件（也是用此程序转化而成），调试后发现该图片能正常显示。

对 print_out 模块进行仿真检查，以发现问题。仿真结果如图 24。开始，扫描行列信号处于消隐阶段，屏幕不显示，可以清楚地看到 rgb 信号均为“0”。之后，hs 和 vs 可以正常工作，rom 地址所对应的像素信息均与 rom 输出以及 vga_in 相等（差一个时钟周期）。因此除了输入的数据外无法检验正误外，还不能发现任何问题。

仔细对照新建模块代码和 print_out 模块代码，尝试发现不同。终于发现是 9‘d640 的位宽错误，改正该错误后，再次下板验证，显示正常。

问题 2：键盘失灵
调试分析：

下板验证过程中，发现游戏不能响应键盘，产生问题。对 move 进行仿真，手动生成输入，产生扩展信号、断码信号以及键值信号和场景切换信号，发现（见图 23），发现输出均能正确对应输出，说明不是应用模块出错。

对 MyTop 新增输出，一为 led 灯：每次出现键盘松开信号，led 灯就亮，并把该状态锁存。调试发现，led 灯从未亮过，说明键盘信号处理模块从来就没有发出过松开信号。

新增输出 8 位七段数码管：记录最新一次的键盘按键信息，包括扩展信号、断码信号以及键值信息。调试发现，该设计虽然能够接收键值信息，却对断码和扩展信号一概不响应（均为 0）。

对 ps2_keyboard_driver 模块进行仿真分析，手动生成输入和设计时钟信号（见仿真代码），进行调试，发现错误：扩展信号和断码没有相应。

对该现象进行分析，发现是每次收到新的 8 位键值信号之后，就会将之前扩展键盘和断码的状态置 0，导致虽然之前匹配到断码信号，每次输出时却被覆盖为 0.

对该模块代码进行改进，将扩展键盘记录和断码记录信号保存到整个键（2-3 个字节）都处理完成后再清零（同时把之前的值传给输出，输出也是寄存器）。调试后，发现 led 虽能闪烁，但是 8 位七段数码管仍然没有关于扩展键盘和断码的有效响应。

分析代码，发现由于键盘完成信号 done 采用队列形式检查上升沿，与基本时钟相比有一定的延时，而此时键值早已更新，因此之前的输入的键值被无效化，因此也对键值也采用队列形式更新，收到 done 的上升沿时，也采用对应的键值。下板验证后，键盘操作正常化。

最后，图 4.5-4.6 是调试成功后的游戏界面展示

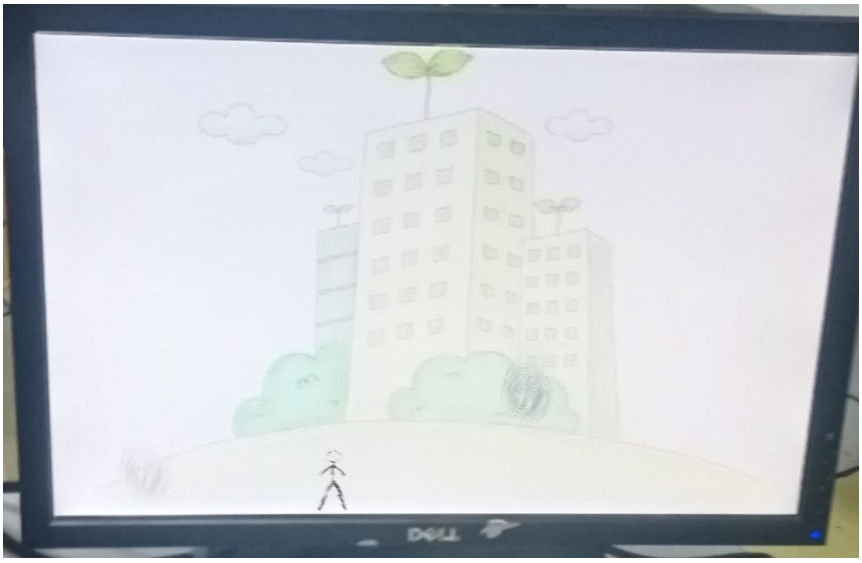

图 4.5 游戏界面展示 1

图 4.6 游戏界面展示 2

## 五、总结与体会

**1.实验体会**

作为数逻的大程，该课程设计与之前的各种实验相比，难度提高了很多，因此也锻炼了很多能力。

由于没有一个既定的目标和模板，只能靠着自己的构想去设计，这虽然让人头疼，也锻炼了我们的独立思考与设计能力。

同时，由于一些必要的知识如 VGA 接口和 ps2 接口等知识没有直接教授，需要不断地搜集资料并弄懂其原理，锻炼了我们的自学能力；

代码编写规模较大（与之前的实验相比），任务重，没有其他人能够合作，且时间紧，因此更加考验了我们的 verilog 编写能力以及程序设计思维，当然，也考察了课程中涉及的各种关于数字系统的状态机、组合电路等知识，逻辑电路设计能力有了进一步的提升。

程序编写之外，最重的任务还是调试程序。开始综合时，警告和错误数不胜数，修正了一些问题警告确源源不绝。一些警告甚至让人毫无头绪（实际上，是因为其他的警告问题所间接发生的错误）。经过不断的修正，警告数量终于达到了相对少的程度。尽管如此，屏幕、键盘依然失灵。经过大量的调试、检验、思考，最终才做出了没有太多 bug 的游戏设计。因此我认为，该数逻大程锻炼了我们的调试与解决办法的能力。

**2.经验教训**

一个教训是意识到了警告的重要性。一些难以解决的问题，其实在警告中其实已经存在，只要按着解除警告的路线走，调试过程可能会相对轻松。

要锻炼仿真代码编写能力。在本次实验中，仿真代码的编写时间较长，效率较低。我认为，既然 Verilog 在仿真过程中有这么多而全的语法，应妥善利用，而不是机械式、低效率地用基础语法来完成。这样就能提高仿真的速度，从而更快的发现并解决问题。

善用仿真的单步调试工具。仿真的波形图中只能显示外部的输入输出情况，看不到内部线路的运作。一开始我尝试将内部线路输出从而便于仿真观察，但该方法效率过低，随着大量的仿真需求，越来越没有可行性。一个行之有效的方法是在波形生成时按暂停键，即跳转至代码执行页面，可以看到每个内部变量的值。

注意劳逸结合。发现总是忽略自己的精神状态来调试，结果是在较为疲劳时，仿真的效率很低。此时倒不如趴在桌上小憩一会儿，精神复苏后的效率相对较高，总的来说能使工作速度提升。