



# 《数字逻辑设计》课程设计报告

中文题目： 基于 FPGA 的弹幕游戏

小组成员 1 \_\_\_\_\_ \*\*\*

学号 \_\_\_\_\_ 322010\*\*\*

专业 \_\_\_\_\_ 计算机科学与技术

学院 \_\_\_\_\_ 计算机科学与技术学院

小组成员 2 \_\_\_\_\_ \*\*\*

学号 \_\_\_\_\_ 322010\*\*\*

专业 \_\_\_\_\_ 计算机科学与技术

学院 \_\_\_\_\_ 计算机科学与技术学院

论文提交日期 2024 年 1 月 17 日

## 目录

<b>1 絮论</b>	<b>1</b>
1.1 基于 FPGA 的游戏设计背景 . . . . .	1
1.2 主要内容和难点 . . . . .	1
1.2.1 主要内容 . . . . .	1
1.2.2 技术要求 . . . . .	2
1.2.3 目的 . . . . .	2
1.2.4 实现重点与难点 . . . . .	2
<b>2 FPGA STG 弹幕游戏设计思路与原理</b>	<b>2</b>
2.1 基于 FPGA 的游戏设计相关内容 . . . . .	2
2.1.1 VGA 显示 . . . . .	2
2.1.2 PS/2 键盘输入 . . . . .	3
2.1.3 DDR3 内存 . . . . .	4
2.1.4 FIFO . . . . .	4
2.1.5 音频输出 . . . . .	4
2.1.6 技术工具 . . . . .	6
2.1.7 课程设计方法 . . . . .	6
2.2 设计方案 . . . . .	6
2.2.1 技术需求 . . . . .	6
2.2.2 整体设计方案 . . . . .	7
2.2.3 游戏流程与操作 . . . . .	8
<b>3 FPGA STG 弹幕游戏设计实现</b>	<b>9</b>
3.1 实现过程 . . . . .	9
3.1.1 顶层模块 . . . . .	9
3.1.2 游戏模块 . . . . .	12
3.1.3 内存管理模块 . . . . .	25
3.2 调试过程 . . . . .	31
3.2.1 仿真调试 . . . . .	31
3.2.2 使用 ILA 逻辑分析仪进行信号抓取 . . . . .	31
<b>4 结果分析与用户反馈</b>	<b>34</b>
4.1 结果分析 . . . . .	34
4.1.1 游戏开始 . . . . .	34
4.1.2 游戏运行过程 . . . . .	34
4.1.3 游戏结束 . . . . .	35
4.2 用户体验与反馈 . . . . .	36

<b>5 总结与致谢</b>	<b>36</b>
5.1 总结	36
5.2 致谢	37

# 1 緒论

## 1.1 基于 FPGA 的游戏设计背景

### FPGA 现场可编程逻辑门阵列

现场可编程逻辑门阵列 (Field Programmable Gate Array, FPGA)，它以 PAL、GAL、CPLD 等可编程逻辑器件为技术基础发展而成。作为特殊应用集成电路中的一种半定制电路，它既弥补全定制电路不足，又克服原有可编程逻辑控制器逻辑门数有限的缺点。

### Verilog 硬件描述语言

Verilog 是一种用于描述、设计电子系统（特别是数字电路）的硬件描述语言，主要用于在集成电路设计，特别是超大规模集成电路的计算机辅助设计。

Verilog 能够在多种抽象级别对数字逻辑系统进行描述：既可以在晶体管级、逻辑门级进行描述，也可以在寄存器传输级对电路信号在寄存器之间的传输情况进行描述。除了对电路的逻辑功能进行描述，Verilog 代码还能够被用于逻辑仿真、逻辑综合，其中后者可以把寄存器传输级的 Verilog 代码转换为逻辑门级的网表，从而方便在 FPGA 上实现硬件电路。

### STG 弹幕游戏

弹幕游戏 (STG) 是一种特殊的 2D 平面射击游戏（包括卷轴射击游戏，和自由移动的平面射击游戏），因其中敌我双方的子弹在屏幕上停留的时间较长且密度很高形成“幕状”而得名。弹幕系射击游戏结合了“射击”和“闪避”两大射击游戏的要素，玩家要“在敌人放出的大量子弹（弹幕）的细小空隙间闪避”，能给玩家闪避弹幕的时候的快感。

### 课程设计选题背景

基于 Verilog 的编程逻辑以及 FPGA 的可编程硬件，我们可以在硬件层级上实现很多软件编程难以实现的功能。比如一些出名的开源项目致力于用 FPGA 硬件模拟经典的游戏机，由于 FPGA 不仅可以模拟游戏机内部的电路，还可以模拟一些独属于某款游戏的定制芯片，因此效果与性能要远远优于软件模拟器。

弹幕游戏由于子弹数量多、分布密，并且游戏内部各种机制与渲染帧数强关联。因此，弹幕游戏对软件及硬件的渲染性能要求较高，需要高速输出稳定的画面，而 FPGA 不仅能够从硬件层面优化计算，还具有丰富的 ROM、RAM 硬件资源，可以使整个设备工作在较高的频率之下。所以 FPGA 正适合弹幕游戏的实现。

## 1.2 主要内容和难点

### 1.2.1 主要内容

FPGA STG 是一款运行在 FPGA 上的精简版弹幕游戏，玩家通过键盘上的方向键控制屏幕上的人物移动，以躲避敌机发射的密集且华丽的弹幕。

### 1.2.2 技术要求

熟练掌握 Verilog 语言设计; 熟练使用 Block RAM, FIFO 以及 DDR3 内存的 IP 核; 掌握信号跨时钟域的处理办法; 掌握 SWORD 键盘通过显示器、键盘、蜂鸣器与耳机接口进行输入输出的原理; 学会使用状态机进行模块化设计

### 1.2.3 目的

加深对模块化编程的理解; 加强 Verilog 的编程能力; 加强硬件调试与仿真能力

### 1.2.4 实现重点与难点

DDR3 内存读写控制; 渲染坐标 (极坐标与直角坐标) 转换的计算; 2D 游戏引擎的设计与编写;

## 2 FPGA STG 弹幕游戏设计思路与原理

### 2.1 基于 FPGA 的游戏设计相关内容

#### 2.1.1 VGA 显示

VGA 显示是我们课程设计的重要基础, 但是我们不希望将其介绍的太过复杂。

我们可以用 CRT 显示器来理解 VGA 的时序要求。CRT 显示器有一个电子枪, 从上至下、从左至右发射电子, 打到屏幕的材料上并激发出对应颜色的光。通过快速地逐行扫描, 显示器便可以利用视觉暂留效果, 在人眼中显示一幅完整的画面。

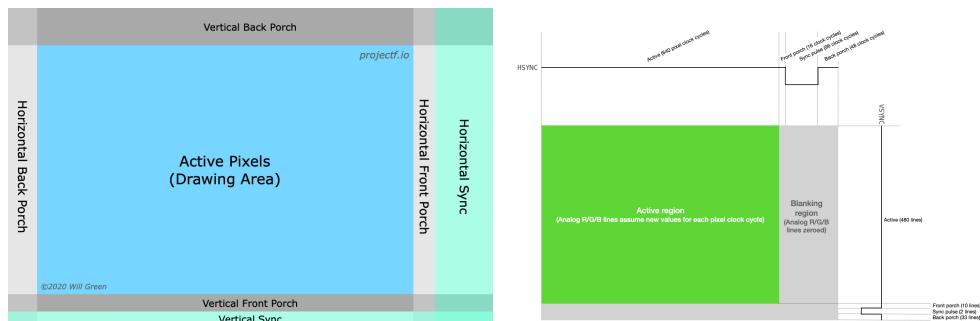


图 1 VGA 信号时序示意图

我们需要在 VGA 模块电子枪扫描到屏幕对应像素点的时候, 及时地将颜色数据并行输出到 r[3:0], g[3:0], b[3:0] 这 12 个条数据线中。当电子枪从一行的末尾回到下一行的开头, 以及从屏幕的最底下回到最顶上时, 需要一定的时间以及一定的调整, 这时就会依次进入同步 (Sync)、消隐 (Porch)、前肩 (Front Porch) 这几个状态, 随后才会进入视频有效 (Active Video), 在视频有效状态时的颜色数据才会显示在屏幕上。

对于我们编程来说，只需要知道对应分辨率与刷新率下的时序参数，并在视频有效阶段给出正确的颜色数据，以及给出正确的同步信号 (vsync 和 hsync)，就可以正常显示画面了。

### 2.1.2 PS/2 键盘输入

主要参考<https://nju-projectn.github.io/dlco-lecture-note/exp/07.html>

#### PS/2 接口的工作时序

PS/2 接口使用两根信号线，一根信号线传输时钟 PS2\_CLK，另一根传输数据 PS2\_DATA。时钟信号主要用于指示数据线上的比特位在什么时候是有效的。键盘和主机间可以进行数据双向传送，这里只讨论键盘向主机传送数据的情况。当 PS2\_DATA 和 PS2\_CLK 信号线都为高电平（空闲）时，键盘才可以给主机发送信号。如果主机将 PS2\_CLK 信号置低，键盘将准备接受主机发来的命令。在我们的实验中，主机不需要发命令，只需将这两根信号线做为输入即可。

当用户按键或松开时，键盘以每帧 11 位的格式串行传送数据给主机，同时在 PS2\_CLK 时钟信号上传输对应的时钟（一般为 10.0–16.7kHz）。第一位是开始位（逻辑 0），后面跟 8 位数据位（低位在前），一个奇偶校验位（奇校验）和一位停止位（逻辑 1）。每位都在时钟的下降沿有效，图 Fig. 55 显示了键盘传送一字节数据的时序。在下降沿有效的主要原因是下降沿正好在数据位的中间，因此可以让数据位从开始变化到接收采样时能有一段信号建立时间。

#### PS/2 键盘的扫描码

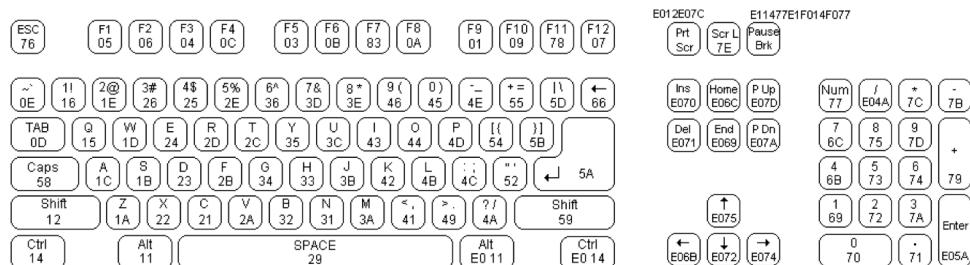


图 2 PS/2 键盘码位对应图

键盘通过 PS2\_DATA 引脚发送的信息称为扫描码，每个扫描码可以由单个数据帧或连续多个数据帧构成。当按键被按下时送出的扫描码被称为通码 (Make Code)，当按键被释放时送出的扫描码称为断码 (Break Code)。

以 W 键为例，W 键的通码是 1Dh，如果 W 键被按下，则 PS2\_DATA 引脚将输出一帧数据，其中的 8 位数据位为 1Dh，如果 W 键一直没有释放，则不断输出扫描码 1Dh 1Dh … 1Dh，直到有其他键按下或者 W 键被放开。某按键的断码是 F0h 加此按键的通码，如释放 W 键时输出的断码为 F0h 1Dh，分两帧传输。

多个键被同时按下时，将逐个输出扫描码，如：先按左 Shift 键（扫描码为 12h）、再按 W 键、放开 W 键、再放开左 Shift 键，则此过程送出的全部扫描码为：12h 1Dh

F0h 1Dh F0h 12h。

要设计一个可以处理多按键的模块，就需要类似理论课上讲的序列识别状态机，识别通码与断码，并且跳转到合理的状态，对检测键盘按下的寄存器进行赋值。

### 2.1.3 DDR3 内存

SWORD 板上有丰富的内存资源，比如 SDRAM 等等，但是经过资料和 datasheet 的查询，我发现 DDR3 的芯片是受 Vivado Memory Interface Generator 这个 IP 核支持的，并且我们历经千辛万苦终于找到了对应的引脚约束。因此我们最终选用了 DDR3 内存进行缓冲。

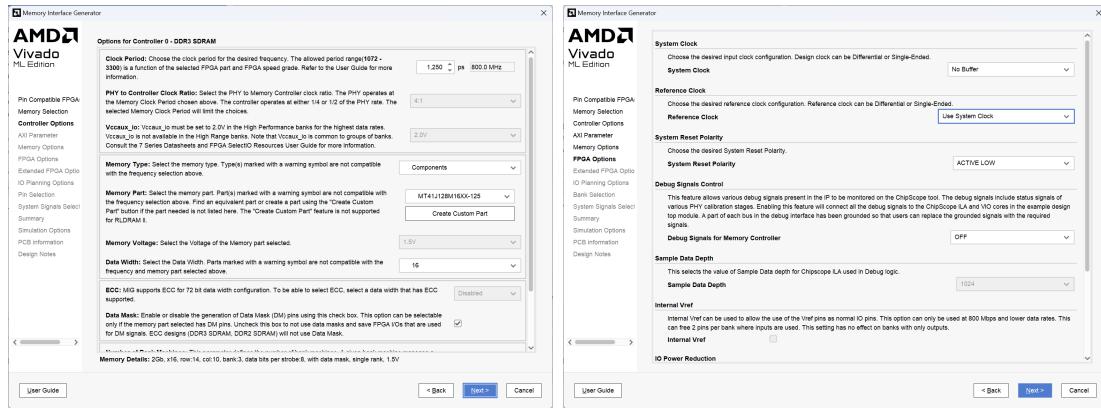


图 3 MIG IP 核设置

配置这个 IP 核看似选项很多，但是我们实际用到的没有多少。需要注意的是输入时钟需要选择 200Mhz，DDR3 芯片型号与 SWORD 板对应，位宽为 16。后面的参考时钟需要选择 No Buffer，因为我们将 100 Mhz 板载时钟升频至 200 Mhz 时，已经经过一次 buffer 了。

### 2.1.4 FIFO

在完成课程设计的过程中，我们发现 FIFO 是一个非常好用的 IP 核。

它实际上是使用了 Block RAM 维护了一个先进先出的队列，因此我们不需要在意地址以及数据的数量，只要按顺序往里写，就一定能按照原顺序读出来。在此之外，FIFO 还支持读写处于不同时钟下，以及读写位宽不同的处理，对于我们完成课程设计，尤其是 DDR3 内存管理模块有很大的帮助。

FIFO 可以被设置为 First Word Fall Through 模式，即在拉高读使能之前，队列头的数据就已经给出来了，没有 1 时钟周期的延时，这可以简化很多逻辑。

### 2.1.5 音频输出

声音来自于震动，我们可以通过调整信号输出的占空比，即一个周期内高电平和低电平时长的比值来实现“震动”。

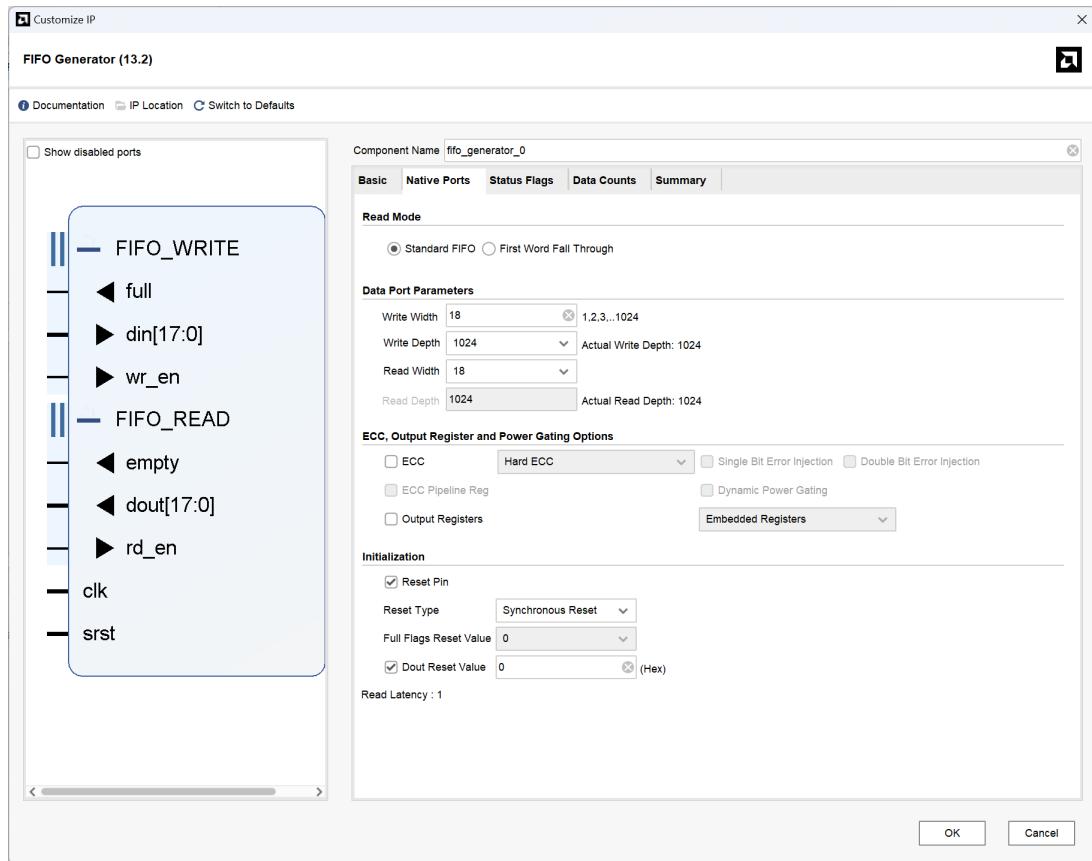


图 4 FIFO IP 核设置

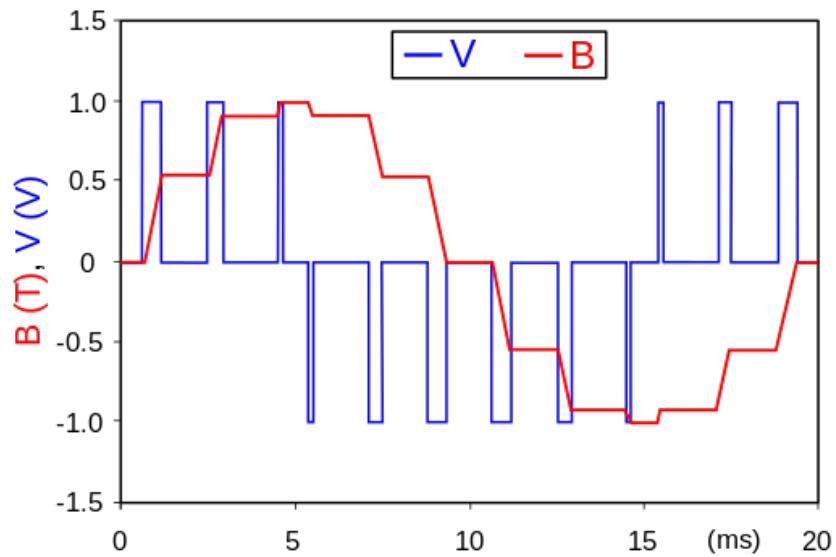


图 5 PWM 脉冲宽度调制

比如我们给蜂鸣器一个高频信号，占空比越大，可以理解为蜂鸣器振幅越大，占空比越小，蜂鸣器振幅就越小。这样，我们可以根据音频波形，根据其幅值赋予对应的占空比，使蜂鸣器发出我们想要的音乐。我们可以使用 Python 将.wav 音频文件转化为 PWM 波形占空比的形式。

在这个过程中，有两个频率，一个使蜂鸣器的振动频率，我们使用了板载时钟 100 Mhz，另一个是占空比切换的频率，要与.wav 的采样率（比如 6.4 kHz 或者 12.8 kHz）一一对应。

### 2.1.6 技术工具

- Vivado 2023.2

我们从 ISE 14.7 转为使用最新版 Vivado 作为开发 IDE，以防止 Windows 11 的一些兼容性问题。在开发过程中，我们发现 Vivado 的项目文件目录结构更加清晰，IP 核设置界面更加友好，在掌握了课程全部实验的经验后，迁移成本并不是很高。

- SWORD 开发板

东四-509 的开发板，内置一颗型号为 xc7k160tffg676-2L 的 FPGA。

- Python 与 C 语言

使用 Python 脚本与 C 语言程序进行 .coe 文件的生成和转换。

- GIMP 等图像处理软件

使用 GIMP 等图像处理软件（以及《图像信息处理》这门课上手工编写的工具）对图像素材进行抖色处理，防止因降低图像量化精度（24 位 RGB 转为 12 位 RGB）导致的色彩断层现象。

### 2.1.7 课程设计方法

利用 Verilog 硬件描述语言设计游戏。通过 Python 将游戏素材转化为.coe 文件，并制作对应的 IP 核，提供给游戏模块。使用游戏模块根据脚本实时渲染游戏画面，并由 DDR3 内存管理模块进行双缓冲，最终交给 VGA 模块进行显示，实现可交互的弹幕游戏。

## 2.2 设计方案

### 2.2.1 技术需求

在游戏开始前显示欢迎界面，并播放游戏主题音乐；进入游戏之后，弹幕可以正确生成并显示，并且可以通过键盘控制人物移动，7 段数码管显示实时得分，Arduino 板 LED 显示残机（剩余生命）数量；游戏结束时，可以显示结束画面并播放结束音乐。

## 2.2.2 整体设计方案

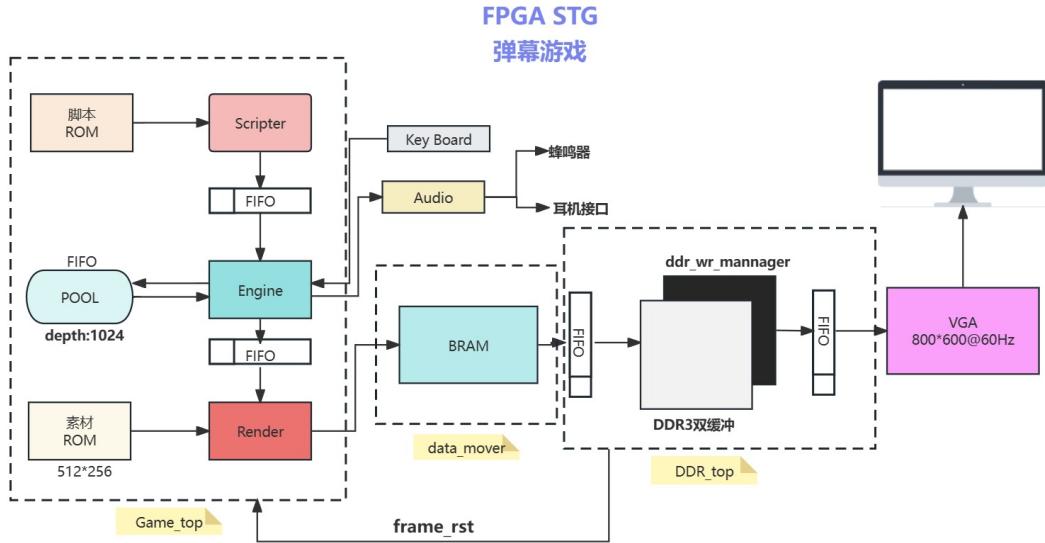


图 6 FPGA STG 弹幕游戏程序整体框架

游戏主要分为两大模块：游戏模块、内存管理模块。

### 内存管理模块

在掌握了 VGA 的时序要求之后，我们决定使用更高的  $800 \times 600 @ 60Hz$  分辨率（相较于  $640 \times 480 @ 60Hz$ ）进行输出，以提供更好的游戏体验。由于弹幕游戏对画面渲染的要求较高，我们使用常用的双缓冲技术来避免画面的撕裂问题，同时稳定输出内容。但是 SWORD 板载的 Block RAM 不足以支持两帧及以上的画面同时存储，因此我们使用了 SWORD 板上容量高达 2GB 的 DDR3 内存作为缓冲存储。

DDR3 内存在同一时刻只能进行读或者写一种操作，并且必须以 8 byte (128 bit) 为一个单位进行操作，直接操作内存不太方便。而且，DDR3 待写入数据的来源来自板载 100 Mhz 时钟，DDR3 IP 核的工作时钟为 200 Mhz，而读取 DDR3 的时钟为 VGA 时钟。为了处理字节宽度不一致，以及跨时钟域的问题，我们在 DDR3 读和写两个方向上都添加了 FIFO 队列，进行缓冲。

### 游戏模块

游戏模块主要分为三个部分：脚本管理 (Script)、游戏引擎 (Engine) 以及渲染模块 (Render)。

脚本管理负责处理我们预先写入 ROM 中的子弹控制脚本，其由 Python 预先将复杂的发射逻辑转化为初始坐标 + 初始速度的形式，方便模块进行处理。同时，我们的游戏是“可编程”的，只改变脚本而不改变其他模块的设计，可以实现不同的弹幕效果。

游戏引擎负责处理当前帧需要绘制的所有游戏元素。该模块维护了一个对象池，其中存储了所有需要绘制的弹幕状态，同时还连接了玩家控制子模块 (Player) 以管理人物当前的状态。游戏引擎会在每次帧刷新信号到来时，更新弹幕的速度和方向，并转化

为直角坐标，放入渲染模块的请求队列 FIFO 当中。

渲染模块会在帧刷新信号到来时，先绘制背景，再读取渲染请求队列中的数据，并按顺序将其绘制在位于 Block RAM 的缓冲当中。

### 其他模块

Data Mover 模块负责在游戏渲染模块渲染完成之后，将数据从 Block RAM 搬运进 DDR3 内存。

剩下的模块为比较通用的输入输出交互模块，在老师给出的参考项目上，我们手写了大部分代码，以提供一个更加通用的框架。我们在提交的文件中附带了下面这些模块单独的代码或 Demo 项目工程文件。

VGA 模块基于老师给的 vgac 模块，我们在重新编写的过程中，使用注释强调了各个信号的功能以及时序要求，并且使用 generate 语句块将分辨率变成了参数，实现了不同分辨率的自适应，从而避免直接修改已经计算好的参数导致的时序问题。

Keyboard 模块基于老师提供的 PS/2 键盘驱动模块设计了状态机，以检测通码与断码的到来，并维护键盘按下的状态。这样一来，就可以支持多按键按下，以及键盘小键盘区域（通码较其他按键有所区别）按键的识别。

Audio 模块基于 PWM 原理将音频信号输出给 Arduino 板上的蜂鸣器与 SWORD 板自带的 3.5mm 音频输出接口，实现音乐的播放。

## 2.2.3 游戏流程与操作

**1 游戏控制** SWORD 板最右侧有三个开关：右侧第一个控制显示器的开启与关闭，第二个开关控制蜂鸣器使能，第三个开关控制耳机接口的使能。

七段数码管初始状态显示分数为 0，剩余生命数预设为 8 个，同时通过 Arduino LED 亮起的个数代表剩余生命数。

### 2 键盘输入

通过上下左右键可以控制人物的移动。

通过 Shift 键可以进入低速模式，并显示与子弹碰撞的判定范围（判定点）。

通过 Enter 键可以从开始界面进入游戏。

### 3 游戏流程

按下 Enter 键开始游戏，通过上下左右键操作人物移动并躲避不断生成的密集的弹幕。

如果碰到弹幕，并且剩余生命数为 0，则判定为游戏失败，进入失败界面。

如果成功躲过了所有的弹幕，则会判定为游戏胜利，显示游戏胜利的信息。

这一部分详情请见第四章“结果分析”部分。

### 3 FPGA STG 弹幕游戏设计实现

#### 3.1 实现过程

##### 3.1.1 顶层模块

整个项目的顶层模块没有过多的处理逻辑，主要是例化子模块并将其进行正确的连接。

top.v

```

1 module Top(
2     input clk,
3     input rstn,
4     input [15:0] SW,
5     input ps2_clk,
6     input ps2_data,
7     output hs,
8     output vs,
9     output [3:0] r,
10    output [3:0] g,
11    output [3:0] b,
12
13    output SEGLED_CLK,
14    output SEGLED_CLR,
15    output SEGLED_D0,
16    output SEGLED_PEN,
17
18    output [7:0] LED,
19    output buzzer,
20    output AUD_PWM,
21    output AUD_SD,
22
23    inout [15:0] ddr3_dq,
24    inout [1:0] ddr3_dqs_n,
25    inout [1:0] ddr3_dqs_p,
26    output [13:0] ddr3_addr,
27    output [2:0] ddr3_ba,
28    output ddr3_ras_n,
29    output ddr3_cas_n,
30    output ddr3_we_n,
31    output ddr3_reset_n,
32    output [0:0] ddr3_ck_p,
33    output [0:0] ddr3_ck_n,
34    output [0:0] ddr3_cke,
35    output [0:0] ddr3_cs_n,
36    output [1:0] ddr3_dm,
37    output [0:0] ddr3_odt
38 );
39
40     wire sys_clk, vga_clock, ddr_clock, pwm_clk;
41     clock_wiz clock(.clk_in1(clk),
42                     .sys_clk(sys_clk),
43                     .ddr_clk(ddr_clock),
44                     .vga_clk(vga_clock),

```

```

45     .pwm_clk(pwm_clk));
46
47     wire [10:0] row_addr;
48     wire [10:0] col_addr;
49     wire [15:0] render_data;
50     wire [15:0] vga_data;
51
52     // DDR 内存读使能
53     wire rd_en;
54     assign rd_en = (col_addr <= 11'd799 && row_addr <= 11'd599) ? 1'b1 : 1'b0;
55
56     // 帧刷新信号
57     wire frame_rst;
58     assign frame_rst = (col_addr == 11'd799 && row_addr == 11'd599) ? 1'b1 : 1'b0;
59
60     // 处理跨时钟域问题
61     reg [1:0] frame_rst_reg;
62     wire frame_rst_sync;
63     assign frame_rst_sync = (frame_rst_reg == 2'b01);
64     always @ (posedge sys_clk) begin
65         frame_rst_reg <= {frame_rst_reg[0], frame_rst};
66     end
67
68     wire ddr_top_init_complete;
69     wire init_calib_complete;
70     wire [15:0] frame_cnt;
71
72     // 时钟分频
73     reg [31:0] clkdiv;
74     always @ (posedge sys_clk) begin
75         clkdiv <= clkdiv + 1'b1;
76     end
77
78     // 八位七段数码管，显示分数与残机
79     wire crashed;
80     wire [3:0] sout;
81     wire [3:0] life_cnt;
82     wire [31:0] seg_data = crashed ? {4'hD, 4'hE, 4'hA, 4'hD, frame_cnt} : (frame_cnt >=
83         16'h0FO0) ? {32'h6666_6666} : {4'h5, frame_cnt, 4'hc, 4'h0, life_cnt};
84     Seg7Device segDevice(.clkIO(clkdiv[3]), .clkScan(clkdiv[15:14]), .clkBlink(clkdiv[25]),
85         .data(seg_data), .point(8'h0), .LES(8'h0), .sout(sout));
86
87     assign SEGLED_CLK = sout[3];
88     assign SEGLED_D0 = sout[2];
89     assign SEGLED_PEN = sout[1];
90     assign SEGLED_CLR = sout[0];
91
92     wire [18:0] buf_addr;
93     wire [11:0] buf_data;
94     wire ready;
95     wire wr_en;
96
97     // DDR3 读写控制模块
98     ddr_top d0 (
99         .ddr_clk(ddr_clock),
100        .wr_clk(sys_clk),
101        .buf_addr(buf_addr),
102        .buf_data(buf_data),
103        .ready(ready),
104        .wr_en(wr_en)
105    );

```

```
100     .wr_en(wr_en),
101     .wr_din({buf_data, 4'h0}),
102     .rd_clk(vga_clock),
103     .rd_en(rd_en),
104     .rd_dout(vga_data),
105     .frame_rst(frame_rst_sync),
106     .rst_n(rstn),
107     .ddr_top_init_complete(ddr_top_init_complete),
108     .init_calib_complete(init_calib_complete),
109     .ddr3_dq(ddr3_dq),
110     .ddr3_dqs_n(ddr3_dqs_n),
111     .ddr3_dqs_p(ddr3_dqs_p),
112     .ddr3_addr(ddr3_addr),
113     .ddr3_ba(ddr3_ba),
114     .ddr3_ras_n(ddr3_ras_n),
115     .ddr3_cas_n(ddr3_cas_n),
116     .ddr3_we_n(ddr3_we_n),
117     .ddr3_reset_n(ddr3_reset_n),
118     .ddr3_ck_p(ddr3_ck_p),
119     .ddr3_ck_n(ddr3_ck_n),
120     .ddr3_cke(ddr3_cke),
121     .ddr3_cs_n(ddr3_cs_n),
122     .ddr3_dm(ddr3_dm),
123     .ddr3_odt(ddr3_odt)
124 );
125
126 // 游戏模块
127 wire [7:0] key_data;
128 wire crash, started;
129 game_module game (
130     .clk(sys_clk),
131     .rstn(rstn),
132     .frame_rst(frame_rst_sync),
133     .keyboard_data(key_data),
134     .buf_addr(buf_addr),
135     .buf_data(buf_data),
136     .frame_cnt(frame_cnt),
137     .game_rdy(ready),
138     .life_cnt(life_cnt),
139     .crash(crash),
140     .crashed(crashed),
141     .started(started)
142 );
143
144 // 数据搬运模块
145 data_mover mover(
146     .clk(sys_clk),
147     .buf_addr(buf_addr),
148     .frame_rst(frame_rst_sync),
149     .game_rdy(ready),
150     .wr_en(wr_en)
151 );
152
153 // VGA 模块
154 vgac #(800, 600) v0 (
155     .vga_clk(vga_clock),
```

```

156     .clr_n(SW[0]),
157     .d_in(vga_data[15:4]),
158     .row_addr(row_addr),
159     .col_addr(col_addr),
160     .r(r), .g(g), .b(b),
161     .hs(hs), .vs(vs)
162 );
163
164 // 键盘模块
165 Keyboard k0 (
166     .clk(sys_clk),
167     .rstn(rstn),
168     .ps2_clk(ps2_clk),
169     .ps2_data(ps2_data),
170     .status(key_data)
171 );
172
173 // Arduino LED 显示残机数量
174 LED_decoder led_disp (
175     .data(life_cnt),
176     .out(LED)
177 );
178
179 // 音频播放
180 Audio audio_inst (
181     .sys_clk(sys_clk),
182     .pwm_clk(pwm_clk),
183     .rstn(rstn),
184     .buzzer(buzzer),
185     .buzzer_en(SW[1]),
186     .AUD_EN(SW[2]),
187     .AUD_SD(AUD_SD),
188     .AUD_PWM(AUD_PWM),
189
190     .started(started),
191     .crash(crash),
192     .crashed(crashed)
193 );
194
195 endmodule

```

### 3.1.2 游戏模块

游戏模块将游戏的子模块：脚本模块 Scripter，游戏处理引擎 Engine，渲染模块 Render 例化并连接对应的 FIFO 队列缓冲。

在游戏模块中，我们通过 2 位寄存器将来自 VGA 时钟的 frame\_rst 信号转化为系统时钟下的信号，以方便下面的模块使用。同时还对键盘数据和游戏状态进行了一定的预处理，维护了 game\_rdy 信号作为渲染完成的信号，供 Data Mover 使用。

game\_top.v

```

1 module game_module (
2     input wire clk,

```

```

3     input wire rstn,
4     input wire frame_rst,
5     input wire [18:0] buf_addr,
6     input wire [7:0] keyboard_data,
7     output wire [15:0] frame_cnt,
8     output wire [11:0] buf_data,
9     output wire started,
10    output wire crash,
11    output wire crashed,
12    output wire game_rdy,
13    output wire [3:0] life_cnt
14 );
15
16 localparam MAX_LIFE = 8;
17 localparam KEY_ENTER = 3'd7;
18
19 reg [15:0] frame_cnt_reg = 16'd0;
20 reg ready = 1'b0;
21 reg [3:0] life_cnt_reg = MAX_LIFE;
22 reg [1:0] crash_reg = 2'b00;
23
24 wire render_ready;
25 assign game_rdy = ready;
26 assign frame_cnt = frame_cnt_reg;
27
28 // 游戏开始判断
29 reg started_reg = 1'b0;
30 assign started = started_reg;
31 always @ (posedge clk or negedge rstn) begin
32   if (!rstn) begin
33     started_reg <= 1'b0;
34   end else if (keyboard_data[KEY_ENTER] == 1'b1) begin // 按回车键开始
35     started_reg <= 1'b1;
36   end else begin
37     started_reg <= started_reg;
38   end
39 end
40
41 // 游戏结束判断
42 assign life_cnt = life_cnt_reg;
43 assign crashed = (life_cnt_reg == 4'd0) ? 1'b1 : 1'b0;
44
45 // 帧计数
46 always @ (posedge clk or negedge rstn) begin
47   if (!rstn) begin
48     frame_cnt_reg <= 16'd0;
49   end else if (frame_rst && started && !crashed) begin
50     frame_cnt_reg <= frame_cnt_reg + 1'd1;
51   end else begin
52     frame_cnt_reg <= frame_cnt_reg;
53   end
54 end
55
56 // 游戏模块初始化判断
57 always @ (posedge clk or negedge rstn) begin
58   if (~rstn) begin

```

```

59         ready <= 1'b0;
60     end else if (frame_rst) begin
61         ready <= 1'b0;
62     end else if (render_ready) begin
63         ready <= 1'b1;
64     end else begin
65         ready <= ready;
66     end
67 end
68
69 // 碰撞判断，减少残机数量
70 always @ (posedge clk or negedge rstn) begin
71     if (!rstn) begin
72         crash_reg <= 2'b00;
73     end else begin
74         crash_reg <= {crash_reg[0], crash};
75     end
76 end
77 always @ (posedge clk or negedge rstn) begin
78     if (!rstn) begin
79         life_cnt_reg <= MAX_LIFE;
80     end else if (crash_reg == 2'b01 && life_cnt_reg > 4'd0) begin // 只检测 crash 变化
81         life_cnt_reg <= life_cnt_reg - 1'd1;
82     end else begin
83         life_cnt_reg <= life_cnt_reg;
84     end
85 end
86
87 wire [87:0] script_data, object_data;
88 wire script_wren;
89 wire object_rden, object_empty;
90 script scripter(
91     .clk(clk),
92     .rstn(rstn),
93     .frame_cnt(frame_cnt_reg),
94     .obj_req_wren(script_wren),
95     .obj_request(script_data)
96 );
97 object_requests requests(
98     .clk(clk),
99     .din(script_data),
100    .wr_en(script_wren),
101    .full(),
102    .almost_full(),
103    .dout(object_data),
104    .rd_en(object_rden),
105    .empty(object_empty)
106 );
107
108 // 玩家逻辑处理
109 wire [39:0] player_data;
110 wire [15:0] player_x = player_data[31 -: 16];
111 wire [15:0] player_y = player_data[15 -: 16];
112 wire player_mode; // 是否为低速模式
113 wire [7:0] masked_data = (!crashed ? keyboard_data : 8'd0);
114 player Reimu (

```

```
115     .clk(clk),
116     .rstn(rstn),
117     .frame_rst(frame_rst),
118     .ps2_data(masked_data),
119     .render_data(player_data),
120     .low_mode(player_mode)
121 );
122
123 wire render_full, render_rden, render_empty;
124 wire engine_wren;
125 wire [39:0] engine_data, render_data;
126 engine game_engine(
127     .clk(clk),
128     .rstn(rstn),
129     .frame_rst(frame_rst),
130     .ps2_data(keyboard_data),
131     .started(started),
132     .req_empty(object_empty),
133     .req_data(object_data),
134     .req_rden(object_rden),
135     .render_full(render_full),
136     .render_data(engine_data),
137     .render_wren(engine_wren),
138     .crashed(crashed),
139     .player_data(player_data),
140     .player_mode(player_mode)
141 );
142 render_requests render_req(
143     .clk(clk),
144     .wr_en(engine_wren),
145     .din(engine_data),
146     .full(),
147     .almost_full(render_full),
148     .rd_en(render_rden),
149     .dout(render_data),
150     .empty(render_empty)
151 );
152
153 wire [11:0] buffer_data;
154 wire [18:0] buffer_addr;
155 wire buffer_wea;
156
157 render renderer(
158     .clk(clk),
159     .rstn(rstn),
160     .frame_rst(frame_rst),
161     .req_data(render_data),
162     .req_empty(render_empty),
163     .req_rden(render_rden),
164     .buffer_addr(buffer_addr),
165     .buffer_wea(buffer_wea),
166     .buffer_data(buffer_data),
167     .ready(render_ready),
168     .crash(crash),
169     .player_x(player_x),
170     .player_y(player_y)
```

```

171 );
172 frame_buffer buffer(
173     .addrA(buffer_addr),
174     .clkA(clk),
175     .dina(buffer_data),
176     .wea(buffer_wea),
177     .addrB(buf_addr),
178     .clkB(clk),
179     .doutB(buf_data)
180 );
181
182 endmodule

```

脚本模块负责处理 ROM 中的脚本数据，并在合适的帧数将其放入游戏引擎待加入物体的 FIFO 队列中。

script.v

```

1 module script(
2     input wire clk,
3     input wire rstn,
4     input wire [15:0] frame_cnt,
5     output wire obj_req_wren,
6     output wire [87:0] obj_request
7 );
8
9 // 定义状态机状态编码
10 localparam RESET = 1'b0;
11 localparam WORK = 1'b1;
12 reg state = RESET;
13
14 // 与 ROM 交互
15 reg [11:0] addr = 12'd0;
16 wire [103:0] item;
17 wire [15:0] cur_frame;
18 reg [15:0] frame = 16'd0, base_x = 16'd0, base_y = 16'd0, deg = 16'd0, rho = 16'd0;
19 reg [7:0] vdeg = 8'd0, vrho = 8'd0, res_id = 8'd0;
20 script_rom scripts(
21     .a(addr),
22     .spo(item)
23 );
24
25 // ROM 输出
26 assign cur_frame = item[103 -: 16]; // 异步读取 - 当前帧
27 always @ (posedge clk) begin // 打一拍转为同步
28     {frame, base_x, base_y, deg, rho, vdeg, vrho, res_id} <= item;
29 end
30
31 // FIFO 输入与使能
32 assign obj_request = {base_x, base_y, deg, rho, vdeg, vrho, res_id};
33 assign obj_req_wren = (frame == frame_cnt) ? 1'b1 : 1'b0;
34
35 // 状态机
36 always @ (posedge clk or negedge rstn) begin
37     if (!rstn) begin
38         state <= RESET;

```

```

39      end else begin
40        case (state)
41          RESET: begin
42            state <= WORK;
43            addr <= 12'd0;
44          end
45          WORK: begin
46            if (cur_frame == frame_cnt)
47              addr <= addr + 12'd1;
48            state <= state;
49          end
50          default: begin
51            state <= RESET;
52          end
53        endcase
54      end
55    end
56 endmodule

```

游戏引擎是一个比较复杂的状态机，比较关键的状态是 UPDATE 和 ADD 状态。

在 UPDATE 状态下，模块取出物体池中的的物体，计算其下一帧的状态，并将其坐标转换为直角坐标放入渲染模块的队列中，再将新状态的物体重新存入物体池。

在 ADD 状态下，模块读取来自 Script 模块的待加入物体队列，并将其初始状态放入口物体池。

在本模块的实际综合结果中，Vivado 使用了 DSP 芯片进行高速的乘法和加法，可以在 1 个时钟周期内得到正确的结果，是本次课程设计可以实现成功的基石。

### engine.v

```

1 module engine(
2   input wire clk,
3   input wire rstn,
4   input wire frame_rst,
5   input wire [7:0] ps2_data,
6   input wire started,
7
8   input wire req_empty,
9   input wire [87:0] req_data,
10  output wire req_rden,
11
12  input wire render_full,
13  output reg [39:0] render_data,
14  output reg render_wren,
15
16  input wire crashed,
17  input wire [39:0] player_data,
18  input wire player_mode
19 );
20
21 localparam RESET  = 3'b000;
22 localparam IDLE   = 3'b001;
23 localparam READY   = 3'b010;
24 localparam PLAYER  = 3'b011;
25 localparam UPDATE  = 3'b100;

```

```

26     localparam ADD      = 3'b101;
27     localparam START    = 3'b110;
28     localparam CRASH   = 3'b111;
29
30     localparam [7:0] LOW_RES_ID = 8'd24; // 低速模式判定点素材 ID
31     localparam [7:0] START_RES_ID = 8'd103; // 开始界面素材 ID
32     localparam [7:0] CRASH_RES_ID = 8'd104; // 结束界面素材 ID
33
34     reg [2:0] state = RESET;
35
36     // 游戏物体队列 FIFO
37     reg pool_wren = 1'b0;
38     reg [88:0] pool_din = 89'd0;
39     wire pool_rden, pool_empty;
40     wire [88:0] pool_dout;
41     object_pool pool(
42         .clk(clk),
43         .full(),
44         .din(pool_din),
45         .wr_en(pool_wren),
46         .empty(pool_empty),
47         .dout(pool_dout),
48         .rd_en(pool_rden)
49     );
50
51     // 物体参数连接到 FIFO 输出 (FWFT)
52     wire [15:0] base_x, base_y;
53     wire signed [15:0] rho, deg;
54     wire [7:0] vdeg, vrho, res_id;
55     wire id;
56     assign {id, base_x, base_y, rho, deg, vrho, vdeg, res_id} = pool_dout;
57
58     // 三角函数计算
59     wire signed [15:0] sin_val, cos_val;
60     sin_value sin(.a(deg), .spo(sin_val));
61     cos_value cos(.a(deg), .spo(cos_val));
62
63     // 下一次坐标计算
64     wire signed [31:0] prod_x, prod_y;
65     wire [16:0] result_x, result_y;
66     wire [15:0] pos_x, pos_y;
67     assign prod_x = rho * cos_val;
68     assign prod_y = rho * sin_val;
69     assign result_x = {1'b0, base_x} + prod_x[31 -: 17];
70     assign result_y = {1'b0, base_y} + prod_y[31 -: 17];
71     assign pos_x = result_x[15 -: 16];
72     assign pos_y = result_y[15 -: 16];
73
74     // 处理批次编号，每处理一帧改变一次
75     reg cur_id = 1'b1;
76
77     // FIFO 使能
78     assign req_rden = (state == ADD) ? 1'b1 : 1'b0;
79     assign pool_rden = (state == UPDATE && id != cur_id) ? 1'b1 : 1'b0;
80
81     // 角度计算，取模

```

```

82     wire [15:0] deg_sum;
83     wire [15:0] new_deg;
84     assign deg_sum = deg + vdeg;
85     assign new_deg = (deg_sum >= 16'd360) ? deg_sum - 16'd360 : deg_sum;
86
87     reg req_empty0 = 1'b0;
88     reg player_render_state = 1'b0;
89
90     always @ (posedge clk or negedge rstn) begin
91         if (!rstn) begin
92             state <= RESET;
93         end else begin
94             case (state)
95                 RESET: begin
96                     state <= IDLE;
97                     render_data <= 40'd0;
98                     render_wren <= 1'b0;
99                     cur_id <= 1'b1;
100                    req_empty0 <= 1'b0;
101                    player_render_state <= 1'b0;
102                end
103                IDLE: begin // 空闲状态
104                    if (frame_rst) begin // 接收到新一帧信号，进入准备状态
105                        state <= READY;
106                    end else begin
107                        state <= IDLE;
108                    end
109                    render_wren <= 1'b0;
110                    pool_wren <= 1'b0;
111                end
112                READY: begin
113                    if (!started) begin // 游戏开始判断
114                        state <= START;
115                    end else if (!pool_empty) begin // 优先处理队列中的物体
116                        state <= PLAYER;
117                        cur_id <= ~cur_id;
118                        player_render_state <= 1'b0;
119                    end else if (!req_empty) begin // 添加新物体 - 在游戏结束后不再添加
120                        if (!crashed) begin
121                            state <= ADD;
122                        end else begin
123                            state <= CRASH;
124                        end
125                    end else if (crashed) begin // 结束画面
126                        state <= CRASH;
127                    end else begin
128                        state <= state;
129                    end
130                    render_wren <= 1'b0;
131                    pool_wren <= 1'b0;
132                end
133                PLAYER: begin
134                    if (player_render_state == 1'b0) begin
135                        render_data <= player_data;
136                        render_wren <= 1;
137                        state <= state;

```

```

138          player_render_state <= 1'b1;
139      end else begin
140          // 低速模式显示判定点
141          render_data <= {LOW_RES_ID, player_data[31 -: 16] - 16'd16,
142                          player_data[15 -: 16] - 16'd8};
143          render_wren <= player_mode;
144          state <= UPDATE;
145          player_render_state <= 1'b0;
146      end
147  end
148 UPDATE: begin
149     if (id == cur_id) begin
150         if (crashed) begin
151             state <= CRASH;
152         end else if (!req_empty) begin
153             state <= ADD;
154         end else begin
155             state <= IDLE;
156         end
157         render_wren <= 1'b0;
158         render_data <= 40'd0;
159
160         pool_wren <= 1'b0;
161         pool_din <= 89'd0;
162     end else begin
163         // 排除出界物体
164         if (pos_x >= 16'd800 || pos_y >= 16'd600) begin
165             render_wren <= 1'b0;
166             pool_wren <= 1'b0;
167         end else begin
168             render_wren <= 1'b1;
169             pool_wren <= 1'b1;
170         end
171
172         // 更新物体数据
173         if (!crashed) begin
174             pool_din <= {cur_id, base_x, base_y, rho + vrho, new_deg, vrho,
175                             vdeg, res_id};
176         end else begin
177             pool_din <= {cur_id, base_x, base_y, rho, deg, 8'd0, 8'd0, res_id}
178                         ;
179         end
180
181     end
182 ADD: begin
183     if (req_empty0) begin
184         state <= IDLE;
185         pool_wren <= 1'b0;
186         pool_din <= 89'd0;
187     end else begin
188         pool_wren <= 1'b1;
189         pool_din <= {cur_id, req_data};
190     end

```

```

191         req_empty0 <= req_empty;
192     end
193     // 游戏开始与结束画面
194     START: begin
195         state <= IDLE;
196         render_data <= {START_RES_ID, 16'd272, 16'd208};
197         render_wren <= 1'b1;
198     end
199     CRASH: begin
200         state <= IDLE;
201         render_data <= {CRASH_RES_ID, 16'd328, 16'd264};
202         render_wren <= 1'b1;
203     end
204     default: begin
205         state <= RESET;
206     end
207   endcase
208 end
209
210 endmodule

```

交给 Render 模块的数据非常简单，就是素材 id，以及素材的直角坐标，方便渲染模块将素材及时地绘制在正确的位置上。由于 Block RAM 资源比较珍贵，我们没有办法多存 Alpha 通道代表透明度，因此我们用全黑 (#000) 代替透明像素，并使用 Python 脚本将原图片中所有的 (#000) 预先转换为 #111，以尽可能减少图片的变化。这样一来，就可以使用像素是否全黑作为 Block RAM 的写使能信号，实现透明像素的渲染。在实现 Render 时，我使用了一个小的“流水线”设计，以在 Block RAM 有延迟的情况下高效地完成渲染。

render.v

```

1 module render (
2   input wire clk,
3   input wire rstn,
4   input wire frame_RST,
5
6   // 处理请求 FIFO
7   input wire [39:0] req_data,
8   input wire req_empty,
9   output reg req_rden,
10
11  // 处理帧缓冲
12  output reg [18:0] buffer_addr,
13  output wire buffer_wea,
14  output wire [11:0] buffer_data,
15  output wire ready,
16
17  // 处理碰撞逻辑
18  output wire crash,
19  input wire [15:0] player_x,
20  input wire [15:0] player_y
21 );
22
23  localparam RESET    = 6'b000001;

```

```

24     localparam IDLE      = 6'b000010;
25     localparam CLEAR     = 6'b000100;
26     localparam REQUEST   = 6'b001000;
27     localparam RENDER    = 6'b010000;
28     localparam WRITE     = 6'b100000;
29
30     localparam WIDTH     = 17'd800;
31     localparam HEIGHT    = 17'd600;
32     localparam RES_WIDTH = 17'd512;
33
34     reg [5:0] state = RESET;
35     reg rendered = 1'b0;
36     assign ready = rendered;
37
38 // 游戏资源 ROM
39 reg [16:0] res_addr = 17'd0;
40 wire [11:0] res_data;
41 resource_rom resources(
42     .clka(clk),
43     .addr(res_addr),
44     .douta(res_data)
45 );
46
47 // 游戏资源描述 ROM
48 reg [7:0] res_id = 8'd0;
49 wire [63:0] res_desc;
50 wire [15:0] res_x, res_y, res_w, res_h;
51 resource_desc descriptions(
52     .a(res_id),
53     .spo(res_desc)
54 );
55 assign {res_x, res_y, res_w, res_h} = res_desc;
56
57 // 背景素材 ROM
58 reg [11:0] col = 12'd0;
59 reg [11:0] row = 12'd0;
60 wire [14:0] bg_addr = row[11:2] * 15'd200 + col[11:2];
61 wire [11:0] bg_color;
62 background_rom background (
63     .a(bg_addr),
64     .spo(bg_color)
65 );
66
67 // BRAM 交互
68 reg [15:0] pos_x = 16'd0, pos_y = 16'd0;
69 reg [15:0] cur_x0 = 16'd0, cur_x1 = 16'd0, cur_x2 = 16'd0;
70 reg [15:0] cur_y0 = 16'd0, cur_y1 = 16'd0, cur_y2 = 16'd0;
71
72 reg [16:0] offsetA = 17'd0;
73 reg [18:0] offsetB = 19'd0;
74
75 // RGB -> BGR
76 assign buffer_data = (state == CLEAR) ? bg_color : res_data;
77
78 localparam [18:0] MAX_ADDR = 19'd800 * 19'd600 - 19'd1;
79

```

```

80     localparam CRASH_DIST = 16; // r <= 4px
81
82     // 碰撞检测
83     wire [15:0] center_xp, center_yp;
84     wire [15:0] center_xo, center_yo;
85     // 玩家中心
86     assign center_xp = player_x + 16'd16;
87     assign center_yp = player_y + 16'd24;
88     // 子弹中心
89     assign center_xo = pos_x + res_w[15:1];
90     assign center_yo = pos_y + res_h[15:1];
91
92     reg crash_reg = 1'b0;
93     wire [31:0] dist;
94     assign crash = crash_reg;
95     assign dist = (center_xp - center_xo) * (center_xp - center_xo) + (center_yp - center_yo)
96         * (center_yp - center_yo);
97
97     always @ (posedge clk or negedge rstn) begin
98         if (!rstn) begin
99             state <= RESET;
100            crash_reg <= 1'b0;
101        end else begin
102            case (state)
103                RESET: begin
104                    state <= IDLE;
105                    cur_x0 <= 16'd0;
106                    cur_x1 <= 16'd0;
107                    cur_x2 <= 16'd0;
108                    cur_y0 <= 16'd0;
109                    cur_y1 <= 16'd0;
110                    cur_y2 <= 16'd0;
111                    pos_x <= 16'd0;
112                    pos_y <= 16'd0;
113                    res_addr <= 17'd0;
114                    res_id <= 8'd0;
115                    buffer_addr <= 19'd0;
116                    // buffer_wea <= 1'b0;
117                    req_rden <= 1'b0;
118                    rendered <= 1'b0;
119                    offsetA <= 17'd0;
120                    offsetB <= 19'd0;
121                    crash_reg <= 1'b0;
122                end
123                IDLE: begin
124                    if (frame_rst) begin
125                        // state <= REQUEST;
126                        state <= CLEAR;
127                        col <= 12'd0;
128                        row <= 12'd0;
129                        buffer_addr <= 19'd0;
130                    end else begin
131                        state <= state;
132                    end
133                    rendered <= 1'b0;
134                    crash_reg <= 1'b0;

```

```

135
136         end
137     CLEAR: begin
138         if (col == 12'd799) begin
139             if (row == 12'd599) begin
140                 buffer_addr <= 19'd0;
141                 state <= REQUEST;
142                 col <= 12'd0;
143                 row <= 12'd0;
144             end else begin
145                 buffer_addr <= buffer_addr + 19'd1;
146                 state <= state;
147                 col <= 12'd0;
148                 row <= row + 12'd1;
149             end
150         end else begin
151             buffer_addr <= buffer_addr + 19'd1;
152             state <= state;
153             col <= col + 12'd1;
154             row <= row;
155         end
156
157         crash_reg <= 1'b0;
158     end
159 REQUEST: begin
160     if (!req_empty) begin
161         {res_id, pos_x, pos_y} <= req_data;
162         req_rden <= 1'b1;
163         cur_x0 <= 16'd0;
164         cur_x1 <= 16'd0;
165         cur_x2 <= 16'd0;
166         cur_y0 <= 16'd0;
167         cur_y1 <= 16'd0;
168         cur_y2 <= 16'd0;
169         state <= RENDER;
170
171         buffer_addr <= 19'd0;
172     end else begin
173         state <= IDLE;
174         rendered <= 1'b1;
175     end
176
177     crash_reg <= 1'b0;
178 end
179 RENDER: begin
180     req_rden <= 1'b0;
181     // 同步的，x y 坐标打两拍
182     if (cur_x0 == res_w - 16'd1 || cur_x0 == 16'hFFF) begin
183         if (cur_y0 == res_h - 16'd1 || cur_y0 == 16'hFFF) begin
184             cur_x0 <= 16'hFFF;
185             cur_y0 <= 16'hFFF;
186         end else begin
187             cur_x0 <= 16'd0;
188             cur_y0 <= cur_y0 + 16'd1;
189         end
190     end else begin
191         cur_x0 <= cur_x0 + 16'd1;

```

```

191           cur_y0 <= cur_y0;
192       end
193
194           // 移位寄存
195           cur_x1 <= cur_x0;
196           cur_x2 <= cur_x1;
197           cur_y1 <= cur_y0;
198           cur_y2 <= cur_y1;
199           offsetA <= (res_y + cur_y0) * RES_WIDTH + (res_x + cur_x0);
200           offsetB <= (pos_y + cur_y1) *      WIDTH + (pos_x + cur_x1);
201
202           // 慢一个周期
203           res_addr <= offsetA;
204
205           // 慢两个周期
206           buffer_addr <= offsetB;
207
208           if (cur_x2 == 16'hFFF && cur_y2 == 16'hFFF) begin
209               state <= REQUEST;
210               // buffer_wea <= 1'b0;
211           end else begin
212               state <= state;
213               // buffer_wea <= buffer_wea;
214           end
215
216               crash_reg <= (dist > 0 && dist <= CRASH_DIST) ? 1'b1 : 1'b0;
217
218           default: begin
219               state <= RESET;
220           end
221           endcase
222       end
223   end
224 endmodule

```

### 3.1.3 内存管理模块

内存管理顶层模块，例化了 MIG IP 核，并连接了两端的 FIFO，而读写由内存读写管理模块进行处理。

ddr\_top.v

```

1 module ddr_top(
2     input wire ddr_clk,
3     input wire wr_clk,
4     input wire wr_en,
5     input wire [15:0] wr_din,
6     input wire rd_clk,
7     input wire rd_en,
8     input wire rst_n,
9     output wire [15:0] rd_dout,
10    input wire frame_rst,
11    output wire ddr_top_init_complete,
12    output wire init_calib_complete,
13    inout [15:0] ddr3_dq,

```

```

14     inout [1:0]    ddr3_dqs_n,
15     inout [1:0]    ddr3_dqs_p,
16     output [13:0]  ddr3_addr,
17     output [2:0]   ddr3_ba,
18     output         ddr3_ras_n,
19     output         ddr3_cas_n,
20     output         ddr3_we_n,
21     output         ddr3_reset_n,
22     output [0:0]   ddr3_ck_p,
23     output [0:0]   ddr3_ck_n,
24     output [0:0]   ddr3_cke,
25     output [0:0]   ddr3_cs_n,
26     output [1:0]   ddr3_dm,
27     output [0:0]   ddr3_odt
28 );
29
30     wire [15:0] wbuf_din;
31     wire [127:0] wbuf_dout;
32     wire [127:0] rbuf_din;
33     wire [15:0] rbuf_dout;
34     assign wbuf_din = wr_din;
35     assign rd_dout = rbuf_dout;
36
37     wire wbuf_wren = wr_en;
38     wire rbuf_rd_en = rd_en;
39     wire wfull, rfull;
40
41 // DDR3 用户接口
42     wire [27:0] app_addr;
43     wire [2:0] app_cmd;
44     wire [127:0] app_wdf_data;
45     wire [15:0] app_wdf_mask;
46     wire [127:0] app_rd_data;
47     wire [11:0] device_temp;
48 // wire init_calib_complete;
49
50     wire [9:0] wbuf_wcount;
51     wire [6:0] wbuf_rcount;
52
53     wire ui_clk;
54
55     wire wbuf_wrst_busy, wbuf_rrst_busy;
56     wire rbuf_wrst_busy, rbuf_rrst_busy;
57
58     wire wbuf_RST, rbuf_RST;
59     reg [15:0] wbuf_RST_SREG;
60     reg [15:0] rbuf_RST_SREG;
61
62     assign ddr_top_init_complete = (wbuf_wrst_busy == 1'b0 && wbuf_rrst_busy == 1'b0)
63                                     && (rbuf_wrst_busy == 1'b0 && rbuf_rrst_busy == 1'b0)
64                                     && !wbuf_RST && !rbuf_RST
65                                     && init_calib_complete;
66
67 // 延时，提供 FIFO 同步重置信号
68     assign wbuf_RST = ~wbuf_RST_SREG;
69     always @ (posedge ui_clk or negedge rst_n) begin

```

```

70      if (!rst_n) begin
71          wbuf_rst_sreg <= 16'h0000;
72      end else begin
73          wbuf_rst_sreg <= {wbuf_rst_sreg[14:0], frame_rst};
74      end
75  end
76
77  assign rbuf_rst = |rbuf_rst_sreg;
78  always @ (posedge ui_clk or negedge rst_n) begin
79      if (!rst_n) begin
80          rbuf_rst_sreg <= 16'h0000;
81      end else begin
82          rbuf_rst_sreg <= {rbuf_rst_sreg[14:0], frame_rst};
83      end
84  end
85
86  fifo_generator_w wbuf(
87      .rst(wbuf_rst),
88      .wr_clk(wr_clk),
89      .rd_clk(ui_clk),
90
91      // FIFO_WRITE
92      .full(),
93      .din(wbuf_din),
94      .wr_en(wbuf_wren),
95
96      // FIFO_READ
97      .empty(),
98      .dout(app_wdf_data),
99      .rd_en(app_wdf_wren),
100
101     .prog_full(wfull),
102     .wr_rst_busy(wbuf_wrst_busy),
103     .rd_rst_busy(wbuf_rrst_busy)
104 );
105
106  fifo_generator_r rbuf(
107      .rst(rbuf_rst),
108      .wr_clk(ui_clk),
109      .rd_clk(rd_clk),
110
111      // FIFO_WRITE
112      .full(),
113      .din({app_rd_data[111 -: 48], app_rd_data[127 -: 16], app_rd_data[47 -: 48],
114          app_rd_data[63 -: 16]}),
115      .wr_en(app_rd_data_valid),
116
117      // FIFO_READ
118      .empty(),
119      .dout(rbuf_dout),
120      .rd_en(rbuf_rden),
121
122      .prog_empty(rempty),
123      .wr_rst_busy(rbuf_wrst_busy),
124      .rd_rst_busy(rbuf_rrst_busy)
125 );

```

```

125
126     wire app_wdf_wren, app_rd_data_valid, rempty;
127     wire app_en, app_rdy, app_wdf_end, app_wdf_rdy, app_rd_data_end;
128     wire app_ref_ack, app_zq_ack, app_sr_active, ui_clk_sync_rst;
129     mig_7series_0 ddr_inst(
130       .ddr3_dq(ddr3_dq), .ddr3_dqs_n(ddr3_dqs_n), .ddr3_dqs_p(ddr3_dqs_p),
131       .ddr3_addr(ddr3_addr), .ddr3_ba(ddr3_ba), .ddr3_ras_n(ddr3_ras_n),
132       .ddr3_cas_n(ddr3_cas_n), .ddr3_we_n(ddr3_we_n), .ddr3_reset_n(ddr3_reset_n),
133       .ddr3_ck_p(ddr3_ck_p), .ddr3_ck_n(ddr3_ck_n), .ddr3_cke(ddr3_cke),
134       .ddr3_cs_n(ddr3_cs_n), .ddr3_dm(ddr3_dm), .ddr3_odt(ddr3_odt),
135
136       .init_calib_complete(init_calib_complete), // IP 核初始化信号
137       .sys_clk_i(ddr_clk),
138
139   // 命令相关 API
140   .app_addr(app_addr), // 地址 {bank, row, col}
141   .app_cmd(app_cmd), // 读 / 写
142   .app_en(app_en), // 使能
143   .app_rdy(app_rdy), // 就绪
144
145   // 写入数据相关 API
146   .app_wdf_data(app_wdf_data), // 写入的数据
147   .app_wdf_end(app_wdf_end), // 写入结束标志
148   .app_wdf_mask(app_wdf_mask), // 字节掩码
149   .app_wdf_wren(app_wdf_wren), // 写入使能
150   .app_wdf_rdy(app_wdf_rdy), // 写入就绪
151
152   // 读取数据相关 API
153   .app_rd_data(app_rd_data),
154   .app_rd_data_end(app_rd_data_end),
155   .app_rd_data_valid(app_rd_data_valid), // 读数据有效
156
157   .app_ref_req(1'b0), // 刷新请求
158   .app_ref_ack(app_ref_ack), // 刷新相应
159   .app_zq_req(1'b0), // ZQ 校准请求
160   .app_zq_ack(app_zq_ack), // ZQ 校准相应
161   .app_sr_req(1'b0),
162   .app_sr_active(app_sr_active),
163
164   .ui_clk(ui_clk), // 用户时钟
165   .ui_clk_sync_rst(ui_clk_sync_rst),
166   .device_temp(device_temp),
167   .sys_rst(rst_n)
168 );
169
170 assign app_wdf_mask = 16'h0000;
171 ddr_wr_manager m0(
172   .clk(ui_clk),
173   .init_calib_complete(init_calib_complete),
174   .ddr_top_init_complete(ddr_top_init_complete),
175   .frame_rst(frame_rst),
176   .app_rdy(app_rdy),
177   .app_wdf_rdy(app_wdf_rdy),
178   .wfull(wfull),
179   .rempty(rempty),
180   .app_en(app_en),

```

```

181     .app_addr(app_addr),
182     .app_cmd(app_cmd),
183     .app_wdf_wren(app_wdf_wren),
184     .app_wdf_end(app_wdf_end)
185   );
186 endmodule

```

DDR3 内存读写管理模块，通过状态机检测 FIFO 的状态并进行对应的操作。在写 FIFO 将满时，统一将其中的数据连续写入 DDR3 内存中，同样地，在读 FIFO 将空时，统一将 DDR3 内存中的数据读出并放入其中。通过 DDR 地址线的高位切换进行双缓冲的管理。由于数据写入比数据读出更重要，因此在进行读写仲裁时，我设计为写优先模式。

ddr\_wr\_manager.v

```

1 module ddr_wr_manager (
2   input wire clk,
3   input wire init_calib_complete,
4   input wire ddr_top_init_complete,
5   input wire app_rdy,
6   input wire app_wdf_rdy,
7   input wire frame_rst,
8
9   input wire wfull,
10  input wire rempty,
11
12  output wire app_en,
13  output reg [27:0] app_addr,
14  output wire [2:0] app_cmd,
15  output wire app_wdf_wren,
16  output wire app_wdf_end
17 );
18
19 localparam RESET = 4'b0001;
20 localparam IDLE = 4'b0010;
21 localparam READ = 4'b0100;
22 localparam WRITE = 4'b1000;
23
24 localparam wr_burst_len = 8'd96;
25 localparam rd_burst_len = 8'd96;
26
27 reg [27:0] app_addr_wr;
28 reg [27:0] app_addr_rd;
29 reg [7:0] wr_cnt;
30 reg [7:0] rd_cnt;
31 reg app_page_wr, app_page_rd;
32
33 reg [3:0] state = RESET;
34 reg wrote;
35
36 assign app_en = ((state == WRITE) & app_rdy & app_wdf_rdy) | (state == READ && app_rdy);
37 assign app_wdf_wren = (state == WRITE & app_rdy & app_wdf_rdy);
38 assign app_wdf_end = app_wdf_wren; // 突发读写结束，4:1 时不用考虑
39 assign app_cmd = (state == READ) ? 3'b1 : 3'b0;
40

```

```

41      always @(*) begin
42          if (state == READ) begin
43              app_addr <= {2'b0, app_page_rd, app_addr_rd[24:0]};
44          end else if (state == WRITE) begin
45              app_addr <= {2'b0, app_page_wr, app_addr_wr[24:0]};
46          end else begin
47              app_addr <= 28'd0;
48          end
49      end
50
51      always @ (posedge clk) begin
52          case (state)
53              RESET: begin
54                  if (ddr_top_init_complete) begin
55                      state <= IDLE;
56                  end else begin
57                      state <= RESET;
58                  end
59
60                  wr_cnt <= 8'd0;
61                  rd_cnt <= 8'd0;
62                  wrote <= 1'b0;
63                  app_addr_wr <= 28'd0;
64                  app_addr_rd <= 28'd0;
65                  app_page_wr <= 1'b0;
66                  app_page_rd <= 1'b0;
67              end
68              IDLE: begin
69                  if (frame_rst) begin
70                      wr_cnt <= 8'd0;
71                      rd_cnt <= 8'd0;
72                      wrote <= 1'b0;
73                      app_addr_wr <= 28'd0;
74                      app_addr_rd <= 28'd1;
75                      // 切换缓冲页
76                      app_page_wr <= ~app_page_wr;
77                      app_page_rd <= ~app_page_rd;
78                  end else if (!ddr_top_init_complete) begin
79                      state <= IDLE;
80                  end else if (wfull) begin
81                      state <= WRITE;
82                  end else if (rempty) begin // & wrote
83                      state <= READ;
84                  end else begin
85                      state <= IDLE;
86                  end
87              end
88              READ: begin
89                  if ((rd_cnt == rd_burst_len - 1) && app_rdy) begin
90                      state <= IDLE;
91                      rd_cnt <= 8'd0;
92                      app_addr_rd <= app_addr_rd + 28'd8;
93                  end else if (app_rdy) begin
94                      rd_cnt <= rd_cnt + 8'd1;
95                      app_addr_rd <= app_addr_rd + 28'd8;
96                  end else begin

```

```

97         rd_cnt <= rd_cnt;
98         app_addr_rd <= app_addr_rd;
99     end
100    end
101   WRITE: begin
102     wrote <= 1'b1;
103     if ((wr_cnt == wr_burst_len - 1) && (app_rdy && app_wdf_rdy)) begin
104       state <= IDLE;
105       wr_cnt <= 8'd0;
106       app_addr_wr <= app_addr_wr + 28'd8;
107     end else if (app_rdy && app_wdf_rdy) begin
108       wr_cnt <= wr_cnt + 8'd1;
109       app_addr_wr <= app_addr_wr + 28'd8;
110     end else begin
111       wr_cnt <= wr_cnt;
112       app_addr_wr <= app_addr_wr;
113     end
114   end
115   default: begin
116     state <= RESET;
117   end
118 endcase
119 end
120
121 endmodule

```

## 3.2 调试过程

### 3.2.1 仿真调试

#### Vivado 与 ModelSim 联合仿真

在加入了 DDR3 内存之后，Vivado 自带的仿真程序运行速度极慢。经过网络搜索，我们发现可以使用 ModelSim 专业软件加速 Vivado 的仿真。

ModelSim 对于仿真激励文件的要求比 Vivado 更加严格，比如不允许使用没有提前声明的 wire 变量。如果在打开仿真时卡住，可以观察 Tcl Console 输出，可能是 ModelSim 已经错误退出了，但 Vivado 没有注意到。

仿真 DDR 内存时，需要从 IP 核官方示例里面拷贝一个 DDR3\_model.sv 文件到项目目录中，否则 init\_calib\_complete 信号将始终不会被拉高。

同时，通过仿真可以对 Block RAM 进行调试，此时需要特别注意时序问题，需要保证地址给的是正确的，且使能信号在正确的时间被拉起。

### 3.2.2 使用 ILA 逻辑分析仪进行信号抓取

在开发后期，仿真调试主要有两个困难。第一是游戏模块之间关联紧密，难以对单一模块进行仿真，而整体仿真又要涉及 DDR3 内存、VGA 和更多的信号，速度非常慢；第二是有些奇怪的问题只会在实机上出现，而仿真调试是正常的。

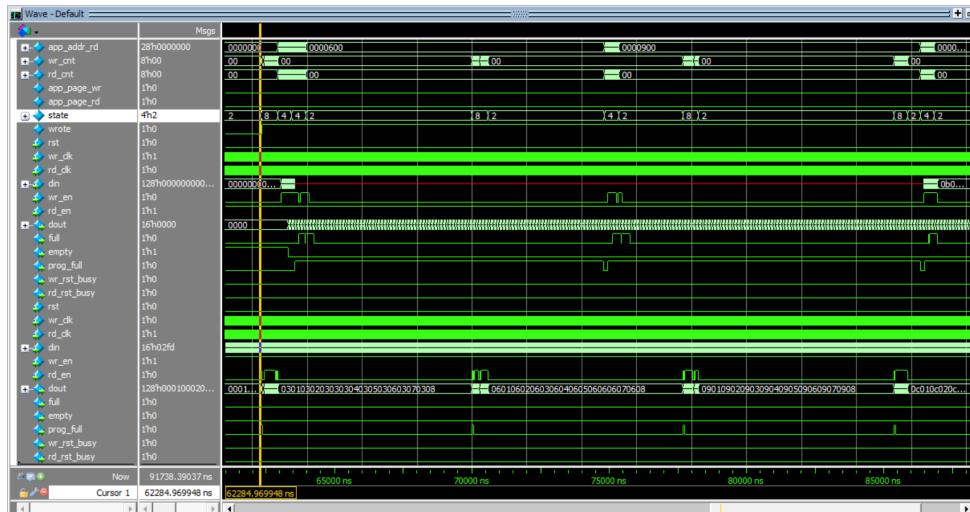


图 7 使用 ModelSim 调试 DDR 内存管理模块

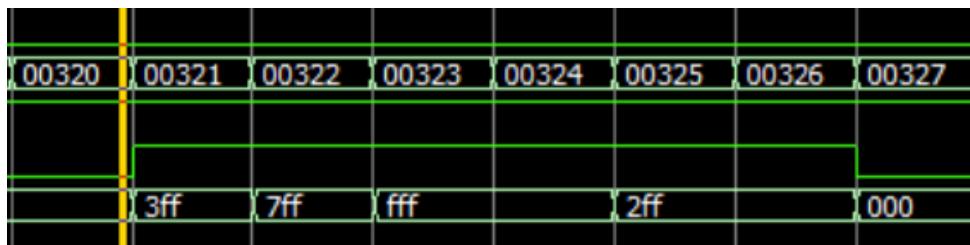


图 8 调试透明像素渲染

在 Vivado 中可以创建 ILA 逻辑分析仪的 IP 核，在实机上电运行时，通过 JTAG 与电脑进行通信，抓取我们想要捕获的信号，很好地解决了上面的两个痛点。

创建 ILA IP 核的方法很简单，网上的教程都很详细，我们只需要最基本地设置需要抓取信号的位宽，以及抓取信号的深度即可。创建好 IP 核之后，可以在顶层模块下例化 ILA IP 核，接入对应的信号，在综合之后可以得到对应的 Debug Core，与 top.bit 同时刷入开发板即可在 Vivado 界面中进行调试与抓取。

ILA 抓取信号可以设置比较简单的比较条件，比如当 buf\_addr 信号为 0001 时抓取当前的 buf\_data 等等。

通过 ILA，我成功地找到了游戏渲染画面在 VGA 显示时错乱的规律，即原先以 0, 1, 2, 3, 4, 5, 6, 7 排序的像素意外变成了 3, 0, 1, 2, 7, 4, 5, 6 的顺序。经过很长时间的调试，我并没有找到这个错位的本质原因。但是通过变通思维，只要我们调整一下写入数据的顺序，就可以将意外错乱的顺序修正回来，这也是一种可行的方法。

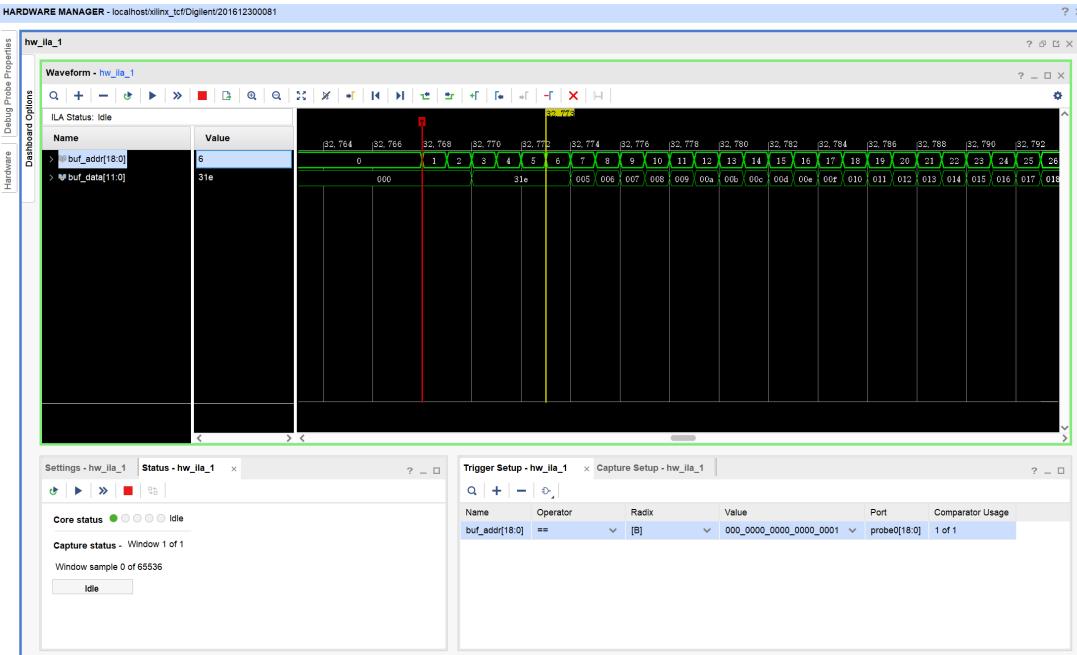


图 9 使用 ILA 设置条件进行信号抓取

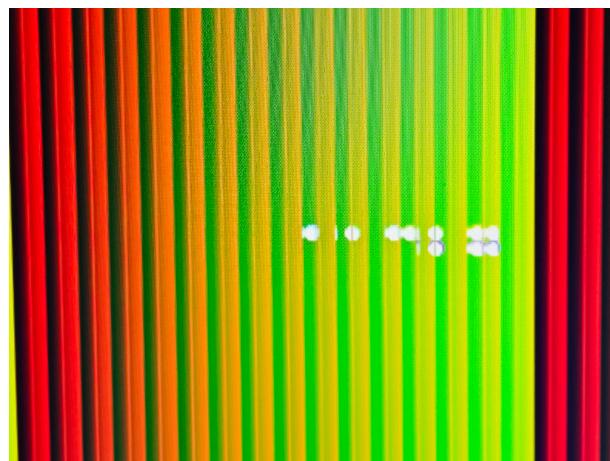


图 10 像素错位问题

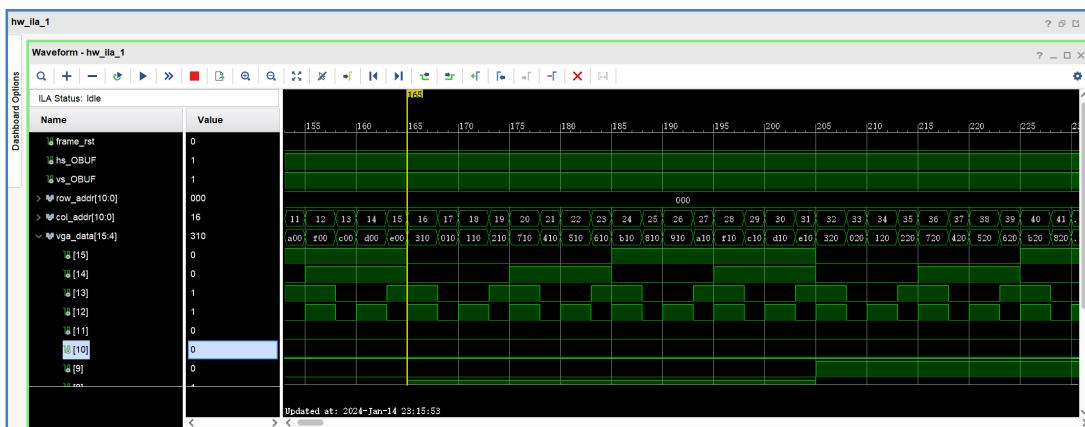


图 11 调试并抓取关键信号

## 4 结果分析与用户反馈

### 4.1 结果分析

#### 4.1.1 游戏开始



图 12 游戏开始画面

当板子上电或按下“FPGA RST”按钮后，游戏可以直接进入开始状态（所有模块都编写了对异步重置的处理），显示效果如图。

屏幕上展示游戏名称为“FPGA STG”，并提示按下“回车 ENTER”键开始游戏。此时如果打开蜂鸣器或者耳机接口的使能开关，就可以听到开始界面的主题音乐。

当按下“回车 ENTER”键后进入游戏阶段。

#### 4.1.2 游戏运行过程

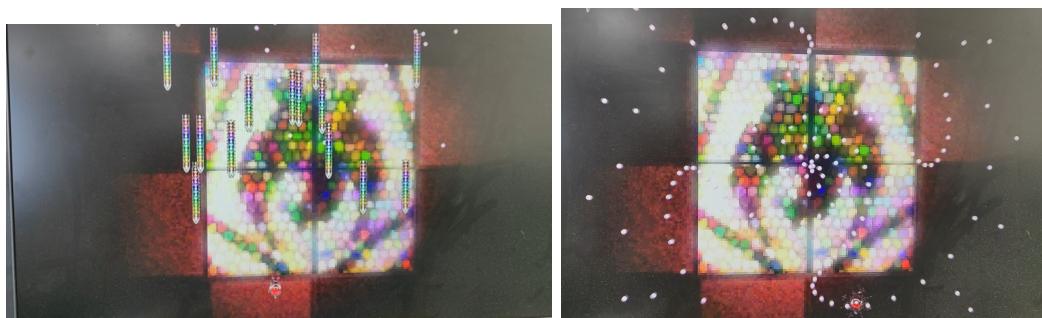


图 13 游戏运行画面

当游戏进入运行状态后，玩家会首先出现再屏幕中下方，如图

屏幕上会以不同的形式出现样式不一的弹幕，轨迹也各不相同，玩家可以通过键盘上的“上下左右”键来控制人物移动，人物移动支持多键同按，即可以向非水平竖直的方向移动，非常灵活。同时人物可以跨过显示器边缘到达另外一边，不会出界。

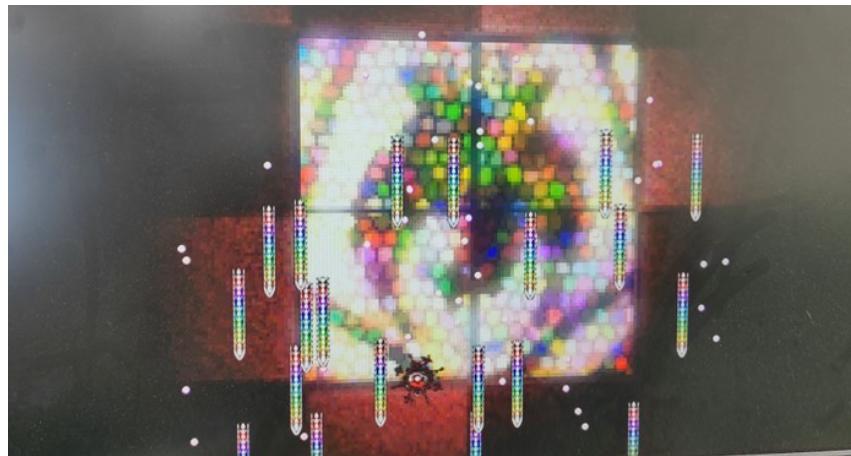


图 14 低速模式展示

当玩家按下“Shift”键时，人物会进入低速模式，便于操作，同时，人物会显示出其碰撞判定半径，有利于精细化操作。

如图，白色圆点范围内代表人物的碰撞判定范围，当这部分与子弹相遇时，玩家会失去一点生命值。当失去生命值时，蜂鸣器会发出清脆短暂的响声提醒玩家。

游戏运行过程中，七段数码管会显示存活时间以及剩余生命值（残机量），初始生命值为 8，每碰撞一次就会减少一点生命值。

#### 4.1.3 游戏结束



图 15 游戏失败显示

当生命值归零时即游戏失败，七段数码管前四位显示“DEAD”，后四位显示游戏分数。

此时屏幕也会显示对应的”Game Over”提示，并播放游戏结束音乐。



图 16 游戏胜利显示

若玩家分数达到一定阈值（我们预设为 F00），则判定为游戏胜利，七段数码管展示“6666666”为胜利庆祝。

## 4.2 用户体验与反馈

经过多位同学的游戏测试，本游戏具有一定的上手难度，但适应熟练后即可流畅操作人物移动，完成游戏胜利，趣味性与挑战性兼备。

有一位同学在观看我们调试的过程时，认出了我们复刻的东方弹幕符卡“波与粒的境界”，说明我们游戏的效果比较出色，画面类似于真正的弹幕游戏。



图 17 《东方地灵殿》中相同的符卡

## 5 总结与致谢

### 5.1 总结

本次课程大作业从十二月初开始准备，从分别编写 VGA 模块，键盘模块，音频模块，到编写 DDR 内存和游戏模块，再到它们之间的联调，总共花费了一个多月的时间。

我们的构思非常有挑战性，在无数次的自我怀疑与绝望过后，我们最终完成了这份大作业。

在完成课程设计的过程中，我们的 Verilog 编写能力相较于之前做课程实验来说有了明显的提升，仿真调试等技能也慢慢地熟练了起来。我们切实地学到了课外的很多硬件知识。

## 5.2 致谢

作为组长，我感谢我的队友对我的支持和理解。在提出最初的构想时，我自己都没有一点底气说自己能做出来这么复杂的课程设计，在交流的过程中，我们把设计的架构改了又改，画满了一张又一张的 A4 草稿纸。然而这是我们第一次接触硬件课，FPGA 的处理能力到底如何，能否在 VGA 的刷新之间那么短的时间间隔渲染出新的一帧，我们没有查到相关的资料。这也就是说，只有做出成果，才能证明这个设想是可行的——很庆幸我们真的做出来了。但如果我没有做出来，那很有可能我们拿不出任何成果。

在完成课程设计的过程中，我们参考了很多作品，他们是：

- ZUN, 《东方地灵殿》《东方风神录》
- RogerDTZ, EGO1-Piano <https://github.com/RogerDTZ/EGO1-Piano>
- Gralerfics, FmcPGA <https://github.com/Gralerfics/FmcPGA>
- CY0807, BRAM\_DDR3\_HDMI [https://github.com/CY0807/BRAM\\_DDR3\\_HDMI](https://github.com/CY0807/BRAM_DDR3_HDMI)
- 锦块, 波与粒的境界为什么长这样?这个视频会告诉你一切<https://www.bilibili.com/video/BV1B7411m7tz/>

在此对它们的作者表示感谢。