

摘要

“星空漫步”在多周期 MCPU 的架构之上，通过对 **MIO-BUS** 进行扩展，利用 **RAM** 与 **VRAM** 完成存储空间译码，从而实现 **VGA 显示**、**PS2 控制**、七段码显示等功能。本项目基于实现游戏的趣味性、操作方便性以及人性化原则，为使用者提供良好的人际交互。

SWORD 实验板上可操作的丰富资源，为设计一个可使用的、多功能的和具备较强趣味性的程序需提供了硬件基础。FPGA 实验板设计对于计算机底层设计以及硬件元器件的功能和作用的理解具有重要的意义，能加深对于可编程硬件设计的理解和设计能力，VGA 显示的显示化为人机交互提供了一个很好的平台。同时 PS2 的实现能够使玩家在游戏中操作更加方便，互动性更强。

关键词： SWORD MCPU MIO-BUS VRAM VGA 显示 PS2 显示

目录

摘要	ii
第 1 章 绪论	5
1.1 “星空漫步”设计背景	5
1.2 国内外现况分析	6
1.3 主要内容和难点	7
1.3.1 “星空漫步”主要内容	7
1.3.2 设计重点、难点	7
2 章 “星空漫步”设计原理	9
2.1 “星空漫步”设计相关内容	9
2.2 “星空漫步”硬件设计	11
第 3 章 “星空漫步”设计实现	14
3.1 实现方法	14
3.1.1 实现步骤	14
3.1.2 重点难点的经验和方法	14
3.2 实现过程	16
3.2.1 总线接口 MIO_BUS 模块	17
3.2.2 RAM 模块——汇编代码	21
3.2.3 PS2 显示模块	25
3.2.4 VGA 显示模块	21
3.2.5 七段码模块	30
第 4 章 系统测试验证与结果分析	32
4.1 功能测试	32
4.2 结果分析	32
4.3 系统演示与操作说明	38
第 5 章 结论与展望	39
5.1 结论	39
5.2 展望	39
附录	40

图目录

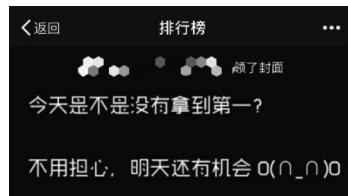
图表 1 游戏创意来源.....	5 ↴
图表 2 CRT 显示原理.....	6 ↴
图表 3 PS2 键盘扫描码.....	7 ↴
图表 4 MIO-BUS 主控器.....	9 ↴
图表 5 MIO-BUS 电路图.....	10 ↴
图表 6 VGA 扫描示意图.....	10 ↴
图表 7 顶层电路图.....	11 ↴
图表 8 TOP 模块电路图&结构图.....	12 ↴
图表 9 工程相关设置.....	16 ↴
图表 10 工程结构.....	17 ↴
图表 11 MIO-BUS 电路图.....	21 ↴
图表 12 迷宫初始化.....	23 ↴
图表 13 RAM 与 VRAM 电路图.....	26 ↴
图表 14 PS2 电路图.....	28 ↴
图表 15 VGA 电路图.....	31 ↴
图表 16 七段码电路图.....	31 ↴
图表 18 游戏 VGA 显示.....	24 ↴
图表 19 七段码数据显示距离.....	24 ↴

第 1 章 绪论

1.1 “星空漫步”设计背景

(内容要点：选择这个设计的思想背景和意义，有什么特点，希望达到什么目的等)

1. 通过课程设计一个简易小游戏，完成从硬件到简单应用程序的设计，加深对多周期 CPU 以及 MIO-BUS 总线和外部输入输出设备间交互原理的理解，并通过这次设计练习使用汇编编写程序，为后续课程做知识储备。
2. **FPGA：**在传统方面主要应用于通信设备的高速接口电路设计，利用 FPGA 处理高速接口的协议并且完成数据的收发和交换。由于通信协议需要高速的处理，并且通信协议需要经常进行修改，因此对于 FPGA 来说，可灵活编程就显得非常重要。
3. **Verilog：**本项目中所使用硬件编程软件是 Xilinx 公司设计的 ISE14.7 软件，使用的语言是 Verilog 语言。Verilog 语言和 C 语言较类似，但是实际编程的过程当中，Verilog 语言进行硬件编程设计需要更多方面的进行硬件实现的考虑。Verilog 语言编程常用自底而上的模块编程的设计方法，在本项目当中采用顶层模块实现各个模块的接口，使得各个子模块有机的结合在一起。
4. **游戏创意来源：**每天 22: 53 分，微信都会准时向我推送“微信运动排行榜”，而每每此时，我都会发现同一个 ID 以“今天是不是没有拿到第一？不用担心，明天还有机会 O(∩_∩)O”的背景图霸占我的微信运动排行榜，步数更是多到望尘莫及。于是，我决定做一个“星空漫步”游戏，其主要功能是通过 PS2 控制人物移动，实时改变 VGA 显示颜色从而记录人物移动路线，并通过七段码显示人物总移动距离，模拟微信运动。



图表 1 游戏创意来源

5. 目的：在自己实现的 SOC 上运行有意义的应用；
 - 1) 底层架构：自主完成的计算机组成课程实验十二——多周期 MCPU 的架构；
 - 2) 通过 MIO-BUS 基础功能，实现七段码对于步数的实时显示与更新，使游戏可以更加真实的模拟微信运动；
 - 3) 通过对 MIO-BUS 进行功能扩展，从而实现 VGA 文本总线接口支持 VGA640*480；
 - 4) 通过对 MIO-BUS 进行功能扩展，从而实现 PS2 键盘总线接口，摆脱单纯的通过实验板上机械按键、电平开关控制的局限，拓宽了设计的角度，提高了用户体验性；
 - 5) 利用 RAM 完成存储空间译码，从而实现汇编驱动程序运行；
 - 6) 利用 VRAM 完成 VGA 显示数据的双向输入输出功能；
 - 7) 本项目基于实现游戏的趣味性、操作方便性以及人性化原则，为使用者提供良好的人机交互。

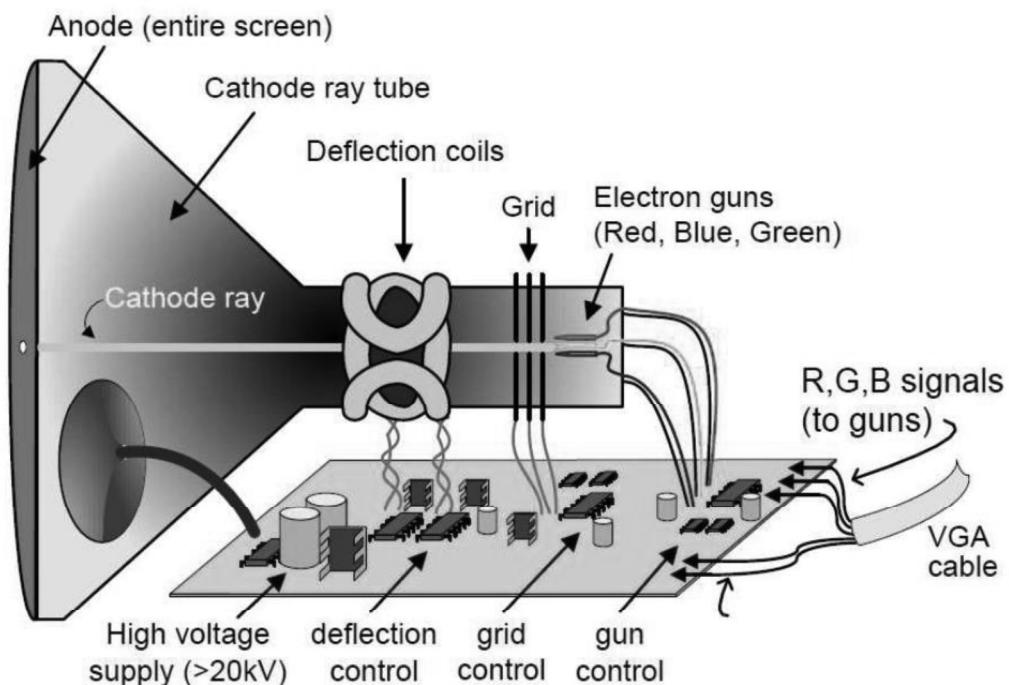
1.2 国内外现状分析

(内容要点：查阅文献和资料，国内外高校是否有相同和类似的设计，现状、特点和结果与本设计的异同等。鼓励增加国内外“应用与研究”的现状分析)。

基于 FPGA 实验板实现游戏的例子在浙江大学有许多，例如计算机兴趣小组的学长们在这个学期中的一些分享以及施老师在上课时播放的游戏视频 demo。

1. FPGA：FPGA 的开发相对于传统 PC、单片机的开发有很大不同。其以并行运算为主，以硬件描述语言来实现；相比于 PC 或单片机的顺序操作有很大区别。FPGA 开发需要从顶层设计、模块分层、逻辑实现、软硬件调试等多方面着手。

2. 在图形处理方面：现有通过滤波算法和降噪算法来提高图像的质量，通过边缘检测的算法对于图像进行处理等技术 FPGA 在通信领域有重要的作用，对于高速通信协议的发展具有重大作用。



图表 2 CRT 显示原理

1.3 主要内容和难点

(内容要点：本设计要完成的主要内容（任务）、功能、技术要求和目的，以及实现的重点或难点；)

本项目主要分为以下三个部分：一、能够在显示器上正确地显示游戏界面，并且能够进行图形的相应操作；二、能够调用 PS2 键盘控制游戏，对于 PS2 键盘的传输信号能够进行正确的判断；三、游戏中能够正确 PS2 控制信息，并通过 MCPU 进行运算后从七段码显示管将步数正确地显示出来。

1.3.1 “星空漫步”主要内容

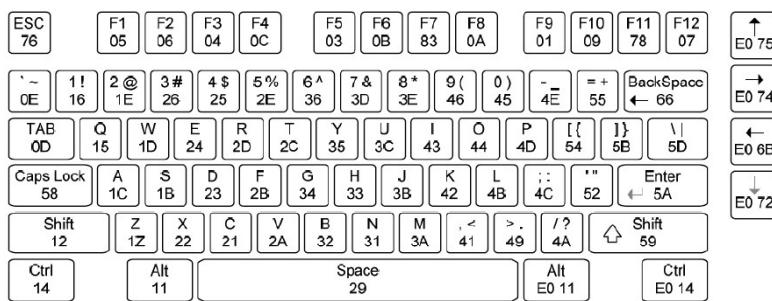
本项目中根据原创经典跑酷游戏改编，加入本组成员的创新功能组合完成。因此在设计当中会包含有以下几个基本的功能和任务：

1. VGA 显示模块：

- 1) 利用 MCPU 计算像素点位置对应的 RGB 值；
- 2) 将该 RGB 值存入到 VRAM 中对应 VGA 数据存储的地址中去，其中每 8*8 个像素与 VRAM 中一个数据一一对应；
- 3) VGA 显示时通过扫描像素点坐标并获得对应的 RGB 值；

2. PS2 功能实现：

- (1) 人物左移：“A”
- (2) 人物右移：“D”
- (3) 人物上移：“W”
- (4) 人物下移：“X”（不是“S”！）



图表 3 PS2 键盘扫描码

3. 每次人物完成一次移动，无论方向，总距离都将加一，然后在七段数码管上显示出对应距离值。

1.3.2 设计重点、难点

1. 各种模块需要有各种参数的输入和输出，需要在顶层模块进行参数接口的设计。由于参数多并且参数复杂，因此在设计当中需要考虑代码和硬件设计的优化，使得整个项目在实验板运行的时候正常；

2. 各个模块之间的时序问题，由于最终程序将通过 SWORD 硬件实现，因此在实际操

作中遇到了较多问题都与硬件本身的时序问题有关，比如在完成一次计算后，不能立即进行数值比较，应适当延迟之后确保数据已经完整传输，再进行判断操作或其他操作；

3. MIO-BUS 对各个模块的调用以及数据的正确传输；
4. VGA 显示模块中 VRAM 数据的正确输入输出以及像素点与 RGB 数据的一一实时对应实现；
5. PS2 通过 MCPU 进行遍历每一个方向输入信号，从而判断 PS2 的具体执行操作；
6. 由于模块的参数之间都比较多，因此在测试中会出现下板验证时间长，多参数难以同时调整等硬件调试问题。

第 2 章 “星空漫步”设计原理

2.1 “星空漫步”设计相关内容

(内容要点：课程设计用到的理论要点、技术工具和方法。如果有课程外理论或技术需要另分节展开简述)

整个项目当中，使用了多周期指令扩展后 MCPU，自主设计模块为：游戏的 MIO-BUS 总线，RAM 存储指令，VRAM 存储 GRB 数据，VGA 模块实现可视化，PS2 实现键盘控制，七段码显示距离。

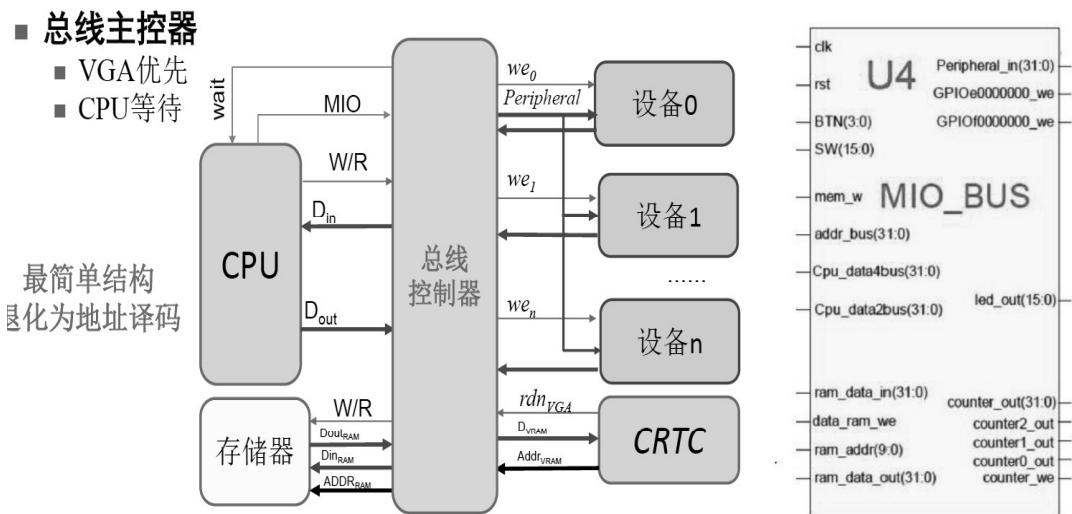
1. MIO-BUS 工作原理：是 MCPU 与外部数据交换接口模块，在本次实验中将数据交换电路合并成一个地址译码电路，从而提供 CPU 读取 SW 等与外部数据以及数据存储内容，并向外设 VGA、7-Seg 传送信号的功能，控制 CPU 与外部数据交换。

1) 基本地址分配：

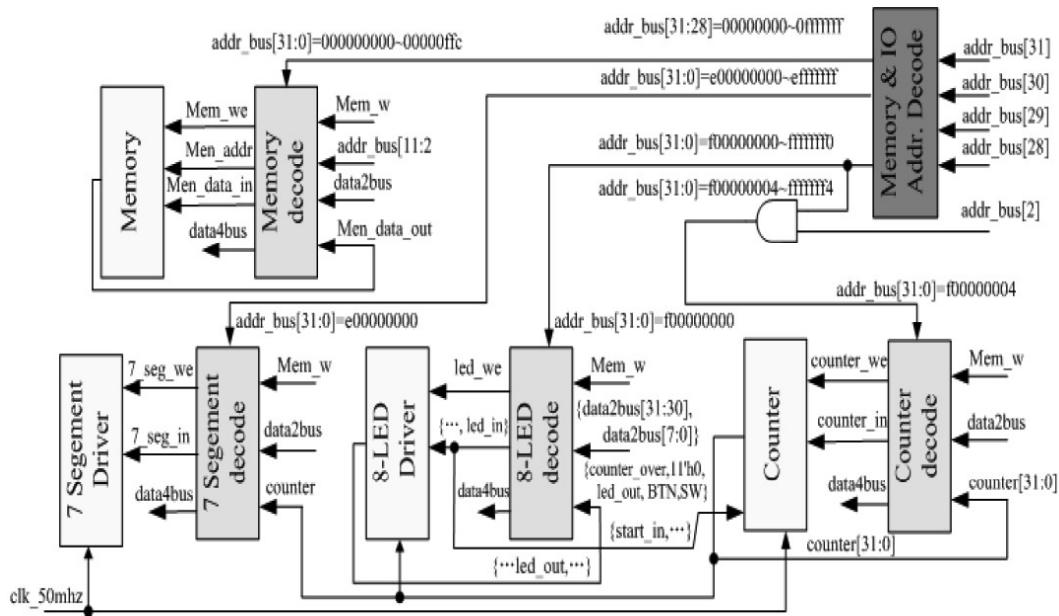
存储器空间:	0000 0000 - ...
GPIO/LED 显示地址 (写) :	F000 0000/FFFF FF00
GPIO/Swith、BTN 地址 (读) :	F000 0000/FFFF FF00
7 段现实器地址 (写) :	E000 0000/FFFF FE00
硬件计数器地址:	F000 0004/FFFF FF04

2) 扩展地址分配：

阵列式键盘地址:	F000 0008/FFFF FF08
PS2 键盘:	E000 0004
VRAM 地址:	0000 Cxxx



图表 4 MIO-BUS 主控器



图表 5 MIO—BUS 电路图

2. VGA 控制器：是一个控制视频显示的 5 个信号基的模块。这些信号为行同步信号 HS、场同步信号 VS，以及基色信号 R、G 和 B。三基色信号分别为红、绿和蓝，从而三色结合产生色彩。显示器的扫描是通过从左到右、从上到下一次逐行进行扫描显示，最终到达屏幕的右下角。第一行表示行同步信号的时序表，前 187 个计数点表示的在消影区，即还没开始进入显示区，从 188 开始进入显示区，到 987 结束，后面的 52 个计数点又在消影区。第二行表示场同步信号。同理。前 31 个计数点和后 56 个计数点表示在消影区，是不显示的。本项目中设计的是 640*480 的标准图像控制器，需要使用 25Hz 的时钟驱动该控制器。



图表 6 VGA 扫描示意图

3. PS2 键盘：是通过扫描编码来识别按键输入，扫描编码与物理按键相关联，因此在键盘中的不同的键盘具有不同的扫描编码。当按下键盘上的一个键时，**Made** 扫描编码被发送到 PS2 接口；释放按键的时候，**Break** 扫描编码被发送到 PS2 接口上。通过对发送信号的判断，判断用户的键盘输入，从而实现游戏状态变化。

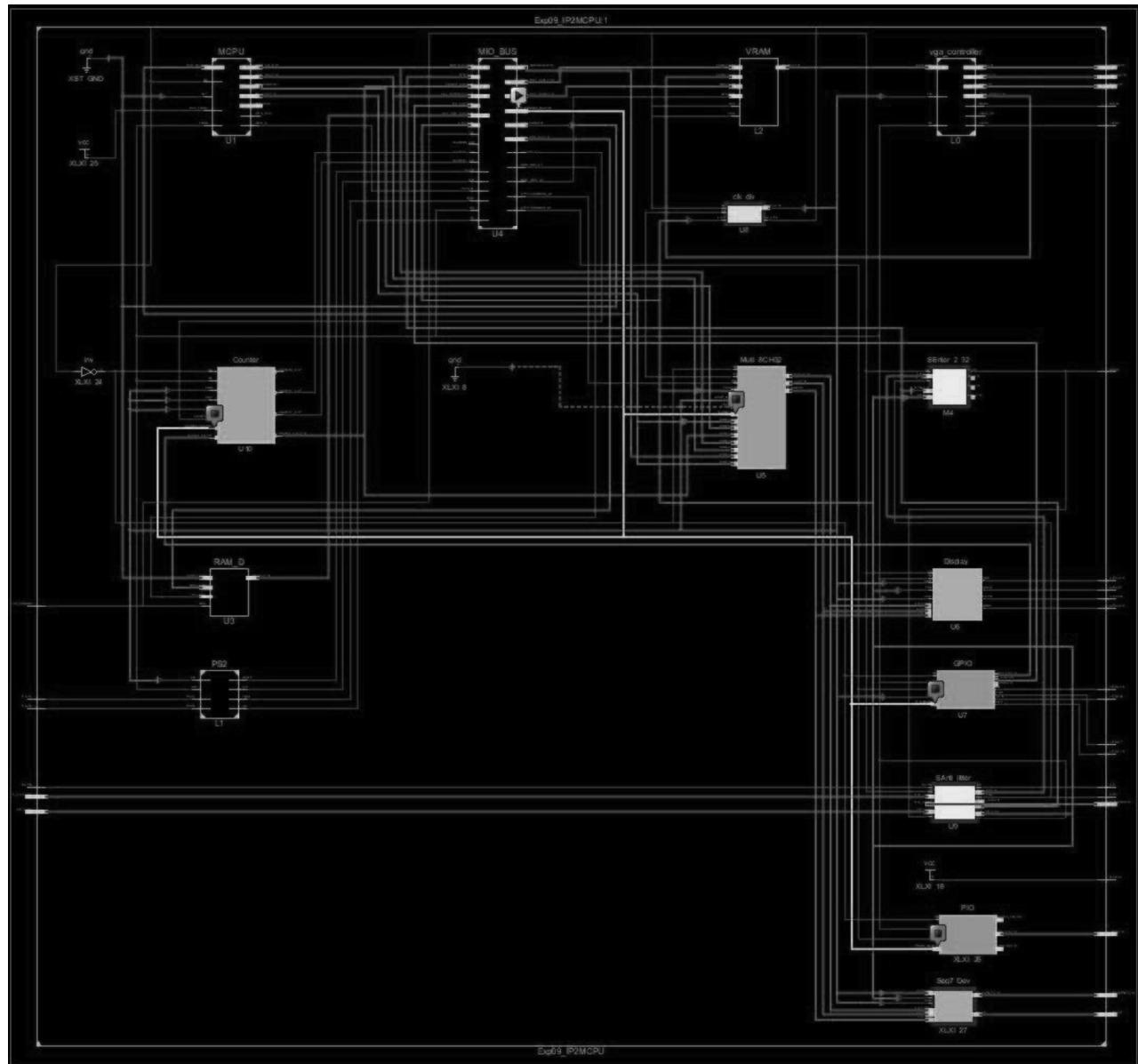
4. VRAM: Video Random Access Memory, 显存，帧存储器，是一种双端口内存，允许在同一时间被所有硬件访问。其主要功能是将显卡的视频数据输出到数模转换器中。采用双数据口设计，其中一个数据口是并行式的数据输出入口，另一个是串行式的数据输出口。在本实验中，使用 VRAM 存储 VGA 对应的 RGB 数值。在此次实验中由于要实现 VGA 显示功能，而 VGA 是时时需要读取 RGB 值完成显示扫描的，同时还需要进行数据写入，因此必须使用 VRAM 来对输入输出 RGB 的值。否则如果只有 RAM 的话，程序运行的数据输入输出将与

VGA 数据输入输出冲突，无法完成实验。

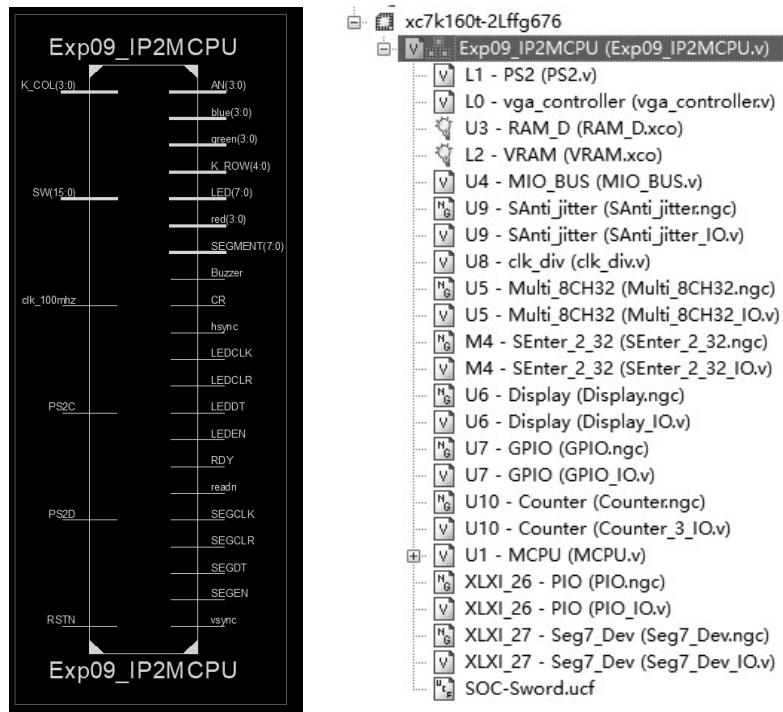
2.2 “星空漫步”硬件设计

(内容要点：详细的硬件设计分析过程，包括顶层及各电路模块结构、逻辑电路图、硬件描述代码等)

1. 项目中的顶层模块电路设计图以及工程各个模块构架，如图所示：



图表 7 顶层电路图



图表 1 TOP 模块电路图&结构图

2. 由于顶层涉及到较多模块且在课程多周期实验的基础上完成,因此在此报告中不再将顶层的所有代码附上,而是选择自主扩展或设计的模块的顶层代码:

```

wire video_on;
wire [9:0] ram_addr;
wire [31:0] ram_data_in;
wire [31:0] ram_data_out;
wire data_vram_we;
wire [12:0] vram_addr;
wire [12:0]cpu_vram_addr;
wire [11:0]cpu_vram_data;
wire [11:0]vram_data_out;

wire clk_25mhz, clk_12mhz;
assign clk_25mhz = Div[1];
assign clk_12mhz = Div[2];

wire left;
wire right;
wire up;
wire down;

PS2 L1(
    .clk(clk_25mhz),
    .clr(rst),
    .data(data_vram_we),
    .addr(ram_addr),
    .read(readin),
    .write(data_vram_we),
    .data_in(ram_data_in),
    .data_out(ram_data_out)
);

```

```

    .PS2C(PS2C),
    .PS2D(PS2D),
    .left(left),
    .right(right),
    .up(up),
    .down(down)
);

vga_controller L0 (
    .clk(clk_25mhz),
    .rst(rst),
    .color(vram_data_out),
    .hsync(hsync),
    .vsync(vsync),
    .video_on(video_on),
    .vram_addr(vram_addr),
    .red(red),
    .green(green),
    .blue(blue)
);

RAM_D U3 (
    .addr(ram_addr[9:0]),
    .wea(data_ram_we),
    .dina(ram_data_in[31:0]),
    .clka(clk_100mhz),
    .douta(ram_data_out[31:0])
);

VRAM L2 (
    .addr(cpu_vram_addr),
    .dina(cpu_vram_data),
    .wea(data_vram_we),
    .clka(clk_100mhz),
    .addrb(vram_addr),
    .clkb(clk_100mhz),
    .doutb(vram_data_out)
);

MIO_BUS U4 (
    .clk(clk_100mhz),
    .rst(rst),
    .BTN(BTN_OK[3:0]),
    .SW(SW_OK[15:0]),

```

```
.mem_w(mem_w),
.addr_bus(Addr_out[31:0]),
.Cpu_data4bus(Data_in[31:0]),
.Cpu_data2bus(Data_out[31:0]),

.ram_data_in(ram_data_in[31:0]),
.data_ram_we(data_ram_we),
.ram_addr(ram_addr[9:0]),

.ram_data_out(ram_data_out[31:0]),

.Peripheral_in(CPU2IO[31:0]),

.GPIOe0000000_we(GPIOe0000000_we),
.GPIOf0000000_we(GPIOF0),

.led_out(LED_out[15:0]),

.counter_out(Counter_out[31:0]),
.counter2_out(counter2_out),
.counter1_out(counter1_out)
.counter0_out(counter0_OUT),
.counter_we(counter_we),

.data_vram_we(data_vram_we),
.cpu_vram_addr(cpu_vram_addr),
.cpu_vram_data(cpu_vram_data),


.left(left),
.right(right),
.up(up),
.down(down)
);
```

第3章 “星空漫步”设计实现

3.1 实现方法

3.1.1 实现步骤

(内容要点：实现的具体思路和方法、重点难点在实现中的经验和方法等)

实现的具体思路和方法

1. 首先将VGA模块连接进入顶层，尝试在不经过原有的模块的情况下是否能在VGA中显示图像，如果显示了预期图像，则说明VGA模块基本调通；
2. 尝试将VGA模块与MCPU相连，编写简单的汇编程序调试，若符合预期显示，则VGA模块与CPU模块调通；
3. 加入PS2判断，使用按键控制移位，下板验证；
4. 给按键移位加VGA显示的逻辑约束，下板验证；
5. 输出人物移动距离，下板验证7段码是否现实正确并实时更新；

3.1.2 重点难点的经验和方法

1. 1) 难点：各种模块需要有各种参数的输入和输出，需要在顶层模块进行参数接口的设计；
2) 解决方法：在实验十二的多周期顶层图基础上，再将新增模块增添进去，并完成数据传输连接。
2. 1) 难点：各个模块之间的时序问题，由于最终程序将通过 SWORD 硬件实现，因此在实际操作中遇到了较多问题都与硬件本身时序问题有关；
2) 解决方法：VGA 改变输入频率，使用 $\text{clk_25mhz} = \text{Div}[1]$ ；即 25mhz 频率；PS2 应适当延迟之后确保数据已经完整传输，即通过寄存器单步数据循环操作后，再进行判断操作或其他操作；
3. 1) 难点：MIO-BUS 对各个模块的调用以及数据的正确传输；
2) 解决方法：写简单的汇编代码进行调试；
4. 1) VGA 显示模块中 VRAM 数据的正确输入输出以及像素点与 RGB 数据的一一实时对应实现；
2) 解决方法：写简单的汇编代码进行调试；
5. 1) PS2 通过 MCPU 进行遍历每一个方向输入信号，从而判断 PS2 的具体执行操作；
2) 解决方法：写简单的汇编代码进行调试，比如每次检测到一个键盘操作就让 VGA 显示一个 8*8 的指定颜色；

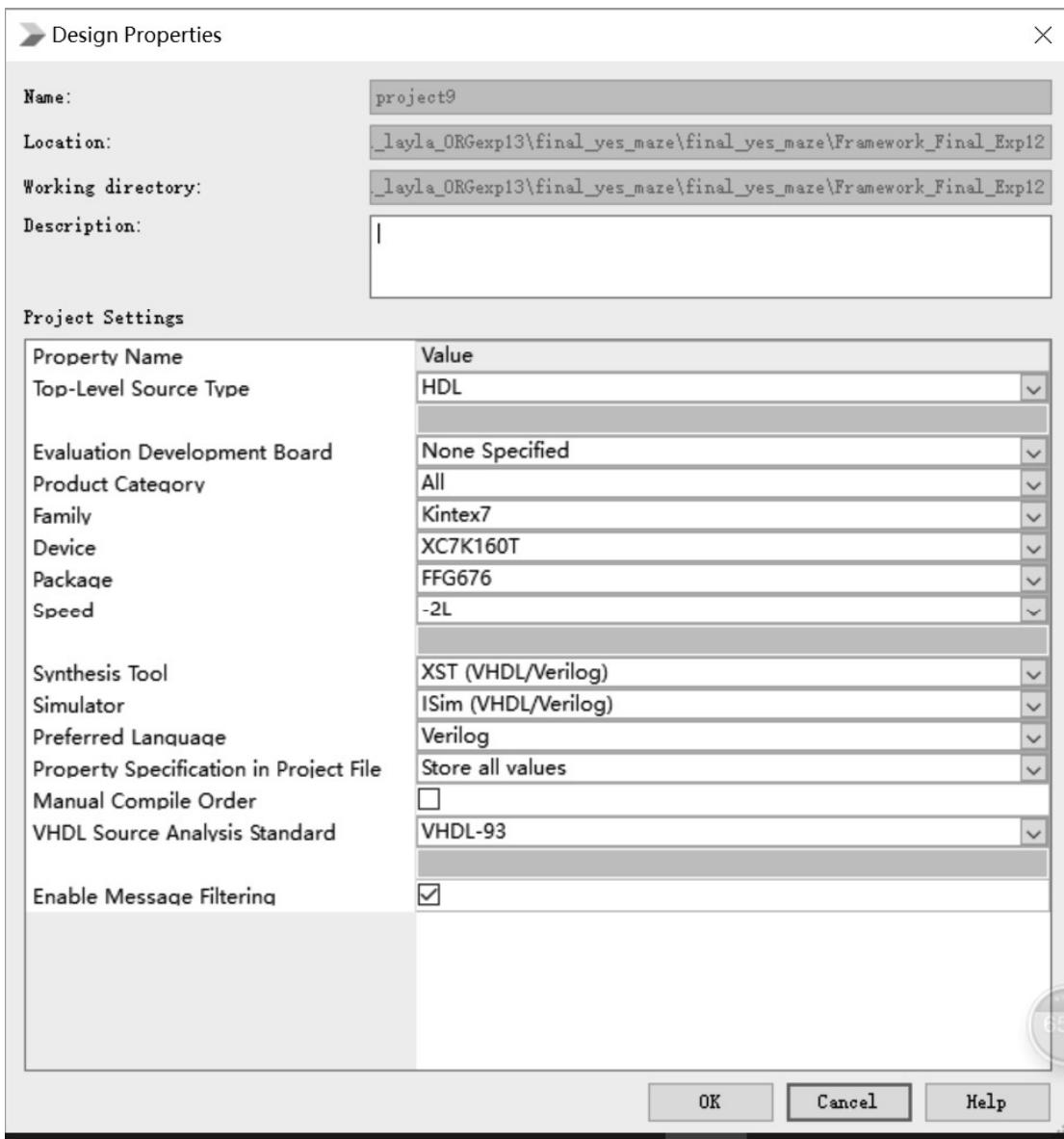
3.2 实现过程

(内容要点：详细的实现过程，包括步骤，模块层次结构，信号定义、综合后 RTL 逻辑图，那些模块需要做仿真等)

步骤：

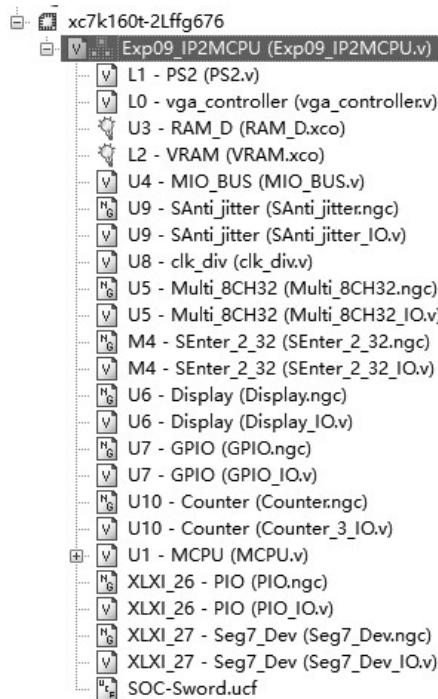
1. 由底向上完成各模块设计、调试与实现；
2. 将各模块代码封装好然后在顶层模块中将各个模块有机的结合起来；
3. 对于顶层模块进行综合，最后下板验证。

其中，SWORD 实验板的设置如下：



图表 9 工程相关设置

模块层次结构以及功能已经在上文中有所展示，因此在此不再赘述。因此下一部分将着重介绍说明各个模块的实现过程：



图表 10 工程结构

3.2.1 IO 总线接口 MIO_BUS 模块——MIO_BUS.v

3.2.1.1 功能介绍

当我了解了 MIO-BUS 的实现原理后，觉得这真的是非常惊艳的一种实现方法，其主要功能是实现存储空间的译码，为一个简单的组合电路。其本质非常简单，通过一个地址译码电路用来将 CPU 与外部数据交换，其主要判断条件为 `addr_bus` 也就是从 CPU 中获得的地址。由于每一个功能对应的数据都存放在固定区域内，因此在获得 `Cpu_data2bus` 等数据的同时，可以通过地址的译码来得到应该被使能触发的模块，同时根据每一个模块所需的不同信号，解析数据将其赋予不同的信号。

在我的 MIO-BUS 中除了原来的基本地址定义：

- 1) `addr_bus == 32'h0000xxxx : data_ram ;`
- 2) `addr_bus == 32'e0000xxxx : 7 segments LEDs;`
- 3) `addr_bus == 32'f0000xxxx : PIO;`
`addr_bus[2] == 0 : 8-LED;`
`addr_bus[2] == 1 : Counter;`

完成扩展后的 MIO-BUS 的附加地址定义：

- 1) `addr_bus == 32'h000cxxxx : data_vram ;`

```

2) addr_bus == 32'e000e0000 : left;
3) addr_bus == 32'e000e0001 : right;
4) addr_bus == 32'e000e0002 : up;
5) addr_bus == 32'e000e0003 : down;

```

3.2.1.2 实现代码

```

module MIO_BUS(    input clk,
                    input rst,
                    input [3:0]BTN,
                    input [15:0]SW,
                    input mem_w,
                    input [31:0]Cpu_data2bus,      //data from CPU
                    output reg[31:0]Cpu_data4bus, //write to CPU
                    input [31:0]addr_bus,        //addr from CPU
                    output reg[31:0]ram_data_in, //from CPU write to
Memory
                    input [31:0]ram_data_out,
                    output reg[9:0]ram_addr,      //Memory Address
signals
                    output reg data_ram_we,
                    input [15:0]led_out,
                    input [31:0]counter_out,
                    input counter0_out,
                    input counter1_out,
                    input counter2_out,
                    output reg counter_we,      //计数器
                    output reg[31:0]Peripheral_in, //送外部设备总线
                    output reg GPIOf0000000_we,   //GPIOfffff00_we
                    output reg GPIOe0000000_we,   //GPIOfffffe00_we
                    output reg data_vram_we,
                    output reg[12:0]cpu_vram_addr,
                    output reg[11:0]cpu_vram_data,
                    input left,
                    input right,

```

```

        input up,
        input down
    );

reg data_ram_rd;           //主存读信号
reg GPIOf0000000_rd;       //设备 1: PIO 写信号
reg GPIOe0000000_rd;       //计数器: Counter_x 写信号
reg counter_rd;           //计数器读信号

//RAM & IO decode signals:
always @(*)
begin
    data_ram_we=0;          //主存写信号
    data_ram_rd=0;           //主存读信号
    counter_we=0;            //计数器写信号
    counter_rd=0;           //计数器读信号
    GPIOf0000000_we=0;       //设备 1: PIO 写信号
    GPIOe0000000_we=0;       //计数器: Counter_x 写信号
    GPIOf0000000_rd=0;       //设备 3、4: SW 等读信号
    GPIOe0000000_rd=0;       //设备 2: 七段显示器写信号
    ram_addr=10'h0;          //内存物理地址: RAM_B 地址
    ram_data_in=32'h0;        //内存读数据: RAM_B 输出数据
    Peripheral_in=32'h0;       //外设总线: CPU 输出, 外设写入数据

case (addr_bus[31:28])
    4'h0:
begin
    if(addr_bus == 32'h0000xxxx) //data_ram
begin
        data_ram_we = mem_w;
        data_ram_rd = ~mem_w;
        ram_addr = addr_bus[11:2];
        ram_data_in = Cpu_data2bus;
        Cpu_data4bus = ram_data_out;
    end
    else if (addr_bus == 32'h000cxxxx) //data_vram
begin
        data_vram_we = mem_w;
        cpu_vram_addr = addr_bus[12:0];
        cpu_vram_data = Cpu_data2bus[11:0];
    end
    else if (addr_bus == 32'h000e0000) //left
begin
        Cpu_data4bus = {31'h00000000,left};
    end
end
end

```

```

    end
    else if (addr_bus == 32'h000e0001) //right
    begin
        Cpu_data4bus = {31'h00000000,right};
    end
    else if (addr_bus == 32'h000e0002) //up
    begin
        Cpu_data4bus = {31'h00000000,up};
    end
    else if (addr_bus == 32'h000e0003) //down
    begin
        Cpu_data4bus = {31'h00000000,down};
    end
end
4'he:           // 7 segments LEDs
begin
    GPIOe0000000_we = mem_w;
    GPIOe0000000_rd = ~mem_w;
    Peripheral_in = Cpu_data2bus;
    Cpu_data4bus = counter_out;
end

4'hf:           //PIO
begin
    if (addr_bus[2]) //Counter
    begin
        counter_we = mem_w;
        counter_rd = ~mem_w;
        Peripheral_in = Cpu_data2bus;
        Cpu_data4bus = counter_out;
    end
    else           //8-LED
    begin
        GPIOf0000000_we = mem_w;
        GPIOf0000000_rd = ~mem_w;
        Peripheral_in = Cpu_data2bus;
    end
end
endcase
end
endmodule

```

3.2.1.3 电路图



图表 11 MIO-BUS 电路图

3.2.2 RAM 模块——汇编代码 RAM_D.xco

3.2.2.1 功能介绍

汇编代码，通过汇编器将 MIPS 代码转换为机器码并存入 `coe` 文件中。然后将 `coe` 文件导入 RAM 中，这便是该程序逻辑控制的主要部分了。整个程序便是基于 MCPU 对该代码的执行来完成的。

3.2.2.2 实现代码

1. 寄存器参数说明

```
# VGA: $t0 0x 000C 0000
# PS2: 0x 000E 0000
    # Left: $t1 +0
    # Right: $t2 +1
    # Up: $t3 +2
    # Down: $t4 +3
# player_x: $s0
# player_y: $s1
```

```
# current color: $s4
# current PS2: $s5
# current distance: $s6
# 7-Seg address: $s7
# color:
    # wall: $t5 red
    # road: $t6 white
    # player: $t7 yellow
    # exit: $t8 green
# parameter: $a0, $a1, $a2, $a3
```

2. 寄存器初始化

```
Initial:
initialize all reg
add $t0, $zero, $zero;
add $t1, $zero, $zero;
add $t2, $zero, $zero;
add $t3, $zero, $zero;
add $t4, $zero, $zero;
add $t5, $zero, $zero;
add $t6, $zero, $zero;
add $t7, $zero, $zero;
add $t8, $zero, $zero;
add $s0, $zero, $zero;
add $s1, $zero, $zero;
add $s2, $zero, $zero;
add $s3, $zero, $zero;
add $s4, $zero, $zero;
add $s5, $zero, $zero;
add $s6, $zero, $zero;
add $s7, $zero, $zero;
add $a0, $zero, $zero;
add $a1, $zero, $zero;
add $a2, $zero, $zero;
add $a3, $zero, $zero;
```

3. 寄存器参数初始赋值

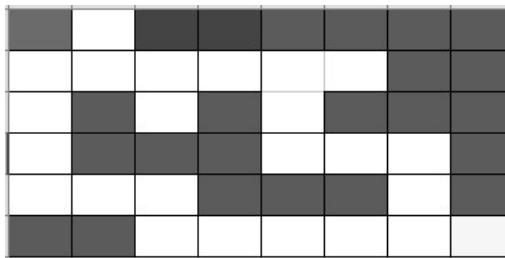
```
initialize vga
lui $t0, 0x000c; # VGA address
lui $t1, 0x000e; # left address
addi $t2, $t1, 1; # right address
addi $t3, $t1, 2; # up address
```

```

addi $t4, $t1, 3; # down address
addi $t5, $t5, 0x0011; # wall: red
addi $t6, $t6, 0x0022; # road: white
addi $t7, $t7, 0x0044; # player: yellow
addi $t8, $t8, 0x0033; # exit: green
lui $s6, 0xe000; # 7-Seg address

```

4. 初始化地图，可以通过 RAM 中 VGA 块的数据进行赋值，从而实现每一块 8*8 小方格不同颜色的效果实现地图。因为时间有限，并没有对整个屏幕的地图进行初始化，只针对左上角 8*6 的方格进行了地图初始化。



图表 12 迷宫初始图

```

Maze:
#0
sw $t7, 0($t0);
sw $t6, 1($t0);
sw $t5, 2($t0);
sw $t5, 3($t0);
sw $t5, 4($t0);
sw $t5, 5($t0);
sw $t5, 6($t0);
sw $t5, 7($t0);
#1
sw $t6, 80($t0);
sw $t6, 81($t0);
sw $t6, 82($t0);
sw $t6, 83($t0);
sw $t6, 84($t0);
sw $t6, 85($t0);
sw $t5, 86($t0);
sw $t5, 87($t0);
#2
sw $t6, 160($t0);
sw $t5, 161($t0);
sw $t6, 162($t0);
sw $t5, 163($t0);
sw $t6, 164($t0);

```

```

sw $t5, 165($t0);
sw $t5, 166($t0);
sw $t5, 167($t0);
#3
sw $t6, 240($t0);
sw $t5, 241($t0);
sw $t5, 242($t0);
sw $t5, 243($t0);
sw $t6, 244($t0);
sw $t6, 245($t0);
sw $t6, 246($t0);
sw $t5, 247($t0);
#4
sw $t6, 320($t0);
sw $t6, 321($t0);
sw $t6, 322($t0);
sw $t5, 323($t0);
sw $t5, 324($t0);
sw $t5, 325($t0);
sw $t6, 326($t0);
sw $t5, 327($t0);
#5
sw $t5, 400($t0);
sw $t5, 401($t0);
sw $t6, 402($t0);
sw $t6, 403($t0);
sw $t6, 404($t0);
sw $t6, 405($t0);
sw $t6, 406($t0);
sw $t8, 407($t0);

```

5. 按键触发判断。

1) 注意事项：在按键判断中尤其重要的一点是在每一次按键方向判断前都需要进行一次 **30000** 数据自循环来实现延迟，否则当按键触发后判断之前，数据实际上并没有完成传输，因此无法正确判断按键触发状态；而用户按键频率远远小于程序运行的频率，因此 **30000** 次的循环延迟在不会影响用户体验的同时，可以增强 PS2 判断的准确性。

2) 按键触发判断过程：

- 从左键的储存地址中获得对应数据；
- 自循环延迟；
- 判断获得的数据是否为 1，若为 1 则表示按键被触发；若为 0 则表示按键未被触发；
- 若按键被触发，则更新任务坐标，y 坐标加一，同时距离值加一然后存储至

七段码存储空间；若按键未被触发，则进入右键判断函数；

e. 串行执行上述过程，完成左右上下键的触发判断以及参数更新；

```

Game:

#left
addi $a0, $zero, 1;
lw $a1, 0($t1);

addi $a2, $zero, 0;
addi $a3, $zero, 30000;
Delay2:
addi $a2, $a2, 1;
bne $a2, $a3, Delay2;

bne $a1, $a0, right;
sub $s0, $s0, $a0;      #cur_player_x = pre_plaer_x - 1;
addi $s7, $s7, 1;
sw $s7, 0($s6);
j state;

right:
addi $a0, $zero, 1;
lw $a1, 0($t2);
bne $a1, $a0, up;
add $s0, $s0, $a0;      #cur_player_x = pre_plaer_x + 1;
addi $s7, $s7, 1;
sw $s7, 0($s6);
j state;

up:
addi $a0, $zero, 1;
lw $a1, 0($t3);
bne $a1, $a0, down;
sub $s1, $s1, $a0;      #cur_player_y = pre_plaer_x - 1;
addi $s7, $s7, 1;
sw $s7, 0($s6);
j state;

down:
addi $a0, $zero, 1;
lw $a1, 0($t4);
bne $a1, $a0, state;
add $s1, $s1, $a0;      #cur_player_y = pre_player_y + 1;
addi $s7, $s7, 1;
sw $s7, 0($s6);

```

```
j state;
```

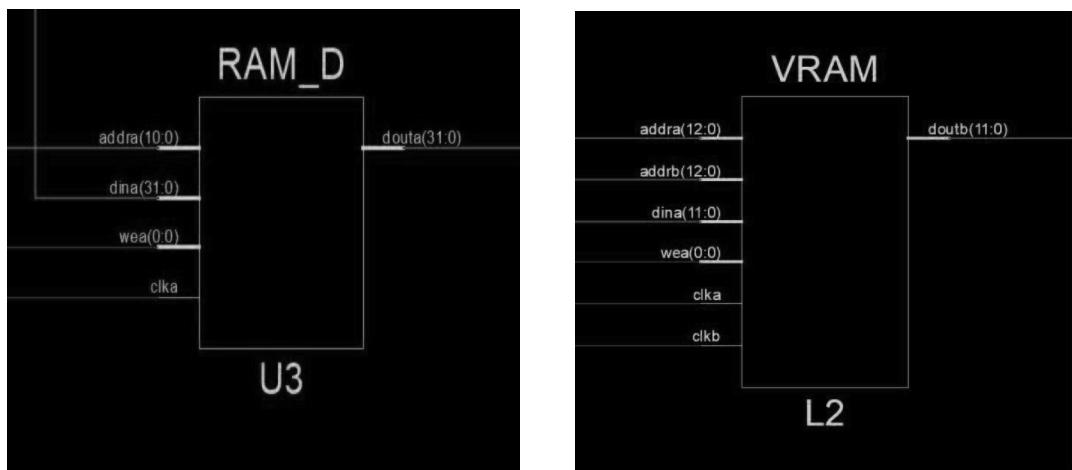
6. 绘制人物移动轨迹:

- 1) 根据更新后的人物坐标, 计算出对应坐标的 RGB 值存储地址, VGA 显示为 640*480, 而 VRAM 中存储值以 8*8 为一个单位, 即屏幕对应 80*60 个 RGB 值。因此(x,y)的 RGB 值为 $y * 80 + x$;
- 2) 更新对应坐标的 RGB 值, 存入人物轨迹的颜色, 从而实现功能;

```
state:  
add $a1, $zero, $zero;    # offset = 0;  
add $a0, $zero, $zero;    # i = 0;  
offsety2:  
beq $a0, $s1, offsetx2; # i == cur_y;  
addi $a1, $a1, 0x50;    # offset += 80;  
addi $a0, $a0, 1;       # i++;  
j offsety2;  
offsetx2:  
add $a1, $a1, $s0;      # offset += cur_x;  
add $a1, $a1, $t0;      # address of current color  
  
sw $t7, 0($a1);        # reset player color
```

```
j Game;
```

3.2.2.3 电路图



图表 13 RAM 与 VRAM 电路图

3.2.3 PS2 显示模块 PS2.v

3.2.2.1 功能介绍

扫描代码为 PS2 标准代码，参考 Verilog 书实现，如下为具体的判断代码。每当对应按键被触发时，相应的寄存器值将被改变，**0** 代表未被触发，**1** 代表该按键被触发；而该信号的改变将通过 MIO-BUS 传入到对应的 PS2 内存中，从而 RAM 在执行汇编代码时可以利用数据比较判断键盘是否被触发。

3.2.3.2 实现代码

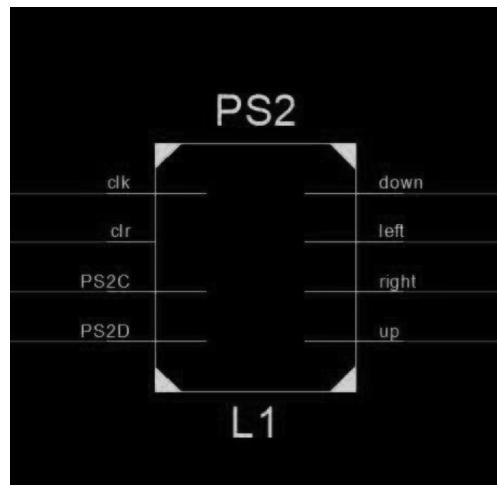
```

always @(posedge clk)
begin
    // 根据当前 PS2 读取值输出信号，信号分别为 left right return reset mode
    if (shift2[8:1] != 8'hF0 && shift1[8:1] == 8'h1C)
        left <= 1;
    else
        left <= 0;
    if (shift2[8:1] != 8'hF0 && shift1[8:1] == 8'h23)
        right <= 1;
    else
        right <= 0;
    if (shift2[8:1] != 8'hF0 && shift1[8:1] == 8'h1D)
        up <= 1;
    else
        up <= 0;
    if (shift2[8:1] != 8'hF0 && shift1[8:1] == 8'h22)
        down <= 1;
    else
        down <= 0;
end

endmodule

```

3.2.3.3 电路图



图表 14 PS2 电路图

3.2.4 VGA 显示模块 vga_controller.v

3.2.4.1 功能介绍

核心思想为，VGA 根据协议保持对屏幕像素点 x 、 y 值的扫描，对应像素点所需要的 RGB 值则被存储在不同于程序其他数据存储区 RAM 的 VRAM 中，原因已经在上述报告中给出。

3.2.4.2 实现代码

- 判断扫描坐标是否在现实范围之内，因为 VGA 实现的过程中实际上的 x 、 y 比显示的 x 、 y 要大，因此需要首先判断当前 VGA 扫描的坐标是否为需要被显示的坐标，如果不是则不需要从 VRAM 中读入对应数据，事实上 VRAM 也没有存储不被显示像素点所需要的 RGB 值；如果是需要被显示的坐标，则将 `video_on` 使能开启。

```
// signals, will be latched for outputs
assign row    = v_count - 10'd35;      // pixel ram row addr
assign col    = h_count - 10'd143;     // pixel ram col addr
wire   h_sync = (h_count > 10'd95);   // 96 -> 799
wire   v_sync = (v_count > 10'd1);    // 2 -> 524
wire   read   = (h_count > 10'd142) && // 143 -> 782
           (h_count < 10'd783) &&      // 640 pixels
           (v_count > 10'd34) &&       // 35 -> 514
           (v_count < 10'd515);        // 480 lines
```

2. VGA 根据扫描协议完成各参数输入之后，进行数据处理和传输。此处很好的体现了 VRAM 的双端口性，如下只讨论 `video_on == 1` 的情况，及 VGA 需要输出显示的请情况，因为当 `video_on==0` 的时候，只需要给 `color(red, green, blue)` 以及 `vram_addr` 随意赋值就可以了。

1) 当 `video_on == 1` 的时候 VGA 模块实际上需要同步实现数据的读入与输出；

RGB 颜色传输顺序：VGA 产生 x、y 值

→`vram_addr` 传该像素点对应 VRAM 中数据的地址

→在 VRAM 中获得该地址存储的 RGB 值

→将数据重新传入给 VGA 模块并对 red、green、blue 赋值

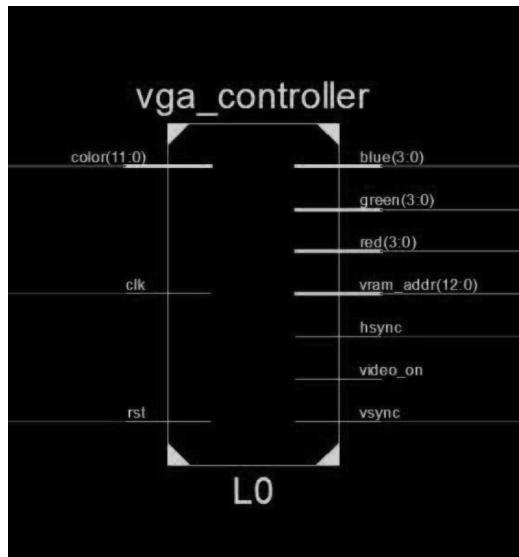
→VGA 显示；

- 2) 数据读入：VGA 的显示依赖于对颜色信号的赋值，在此次实验中使用了 red、green、blue 各 4 位的值，总共 12 位；
- 3) 数据输出：`vram_addr` 由上述报告已经知道 VGA 对应的数据为 80*60 的数组，而在实际存储过程中数据为一维数组。即若像素坐标为(x, y)则对应的数组为(x/8, y/8)，再将该数组转换为一维计算其对应地址为： $y/8*80 + x/8 = y * 10 + x/8$ ；
- 4) 在本次实验中，应该确保所有的算数计算等操作都是由 MCPU 完成的，而不是由硬件语言完成的，因此在此步骤中不能直接进行算数计算。在此采用信号线的移动传输，例如 $y * 10 = y*8 + y*2$ ；而 *8 可以通过将低位数据移高 3 位完成传输；*2 同理；因此 `assign vram_addr = video_on ? ({row[9:3], 6'h0} + {2'h0, row[9:3], 4'h0} + {6'h0, col[9:3]}) :13'h0 ;`

```
// vga signals
always @ (posedge clk) begin
    video_on <= read;      // read pixel (active low)
    hsync     <= h_sync;    // horizontal synchronization
    vsync     <= v_sync;    // vertical   synchronization
    red       <= video_on ? color[11:8]:4'hF; // 3-bit red
    green     <= video_on ? color[7:4]:4'h0; // 3-bit green
    blue      <= video_on ? color[3:0]:4'hF; // 3-bit blue
end

assign vram_addr = video_on ? ({row[9:3], 6'h0} + {2'h0, row[9:3],
4'h0} + {6'h0, col[9:3]}) :13'h0 ;
endmodule
```

3.2.4.3 电路图



图表 15 VGA 电路图

3.2.5 七段码技术显示

3.2.5.1 功能介绍

本游戏需要实现每次移动都将总距离加一并在七段码上实时显示。

3.2.5.2 实现代码

- 在汇编中每一次判断得到一次按键被触发，就将总距离进行一次自加。

```
addi $s7, $s7, 1;
sw $s7, 0($s6);
```

- 将该数据传入到七段码中，因为使用了 sw 因此 CPU2IO[31:0]即为需要输出的值，因此在通道 0 即 SW[7:5]=000 的时候即为总距离。

```
Multi_8CH32 U5 (.clk(IO_clk),
                    .rst(rst),
                    .EN(GPIOe0000000_we),
                    .Test(SW_OK[7:5]),
                    .point_in({Div[31:0], Div[31:0]}),
                    .LES({N0, N0, N0, N0, N0, N0, N0,
                           N0, N0, N0, N0, N0, N0, N0,
                           N0, N0, N0, N0, N0, N0, N0, N0});
```

```

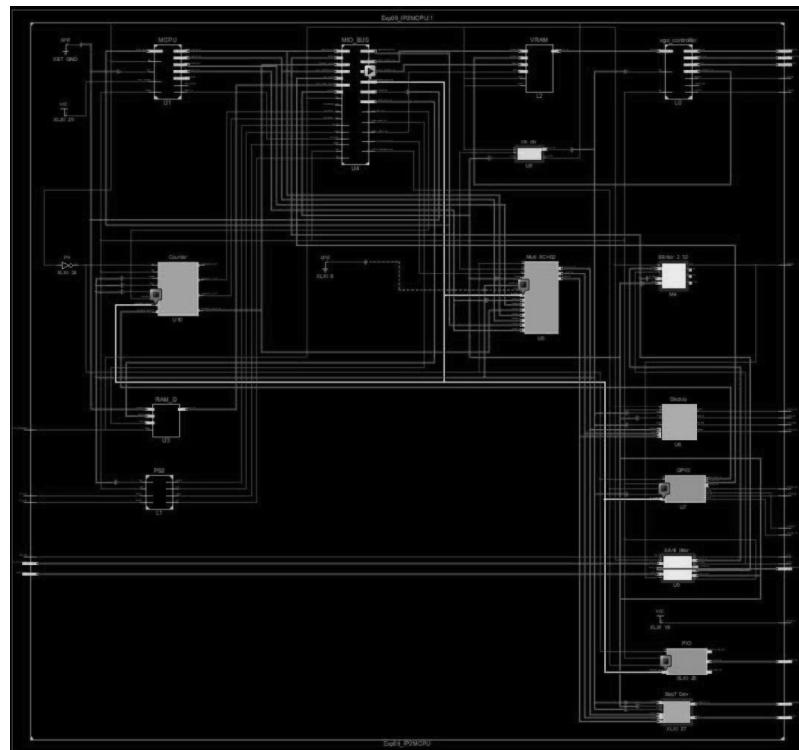
NO, NO, NO, NO, NO, NO, NO, NO,
NO}),

.Data0(CPU2IO[31:0]),
.data1({NO, NO, PC[31:2]}),
.data2(inst[31:0]),
.data3(Counter_out[31:0]),
.data4(Addr_out[31:0]),
.data5(Data_out[31:0]),
.data6(Data_in[31:0]),
.data7(PC[31:0]),

.Disp_num(Disp_num[31:0]),
.point_out(point_out[7:0]),
.LE_out(LE_out[7:0])
);

```

3.2.5.3 电路图



图表 16 Seg 电路图

第 4 章 系统测试验证与结果分析

4.1 功能测试

(内容要点：根据系统设计的功能设计测试方法，验证是否达到设计功能目标及功能正确性和完备性等)

本项目当中，功能的实现主要在四个模块：PS2 模块，即能否正确地判断 PS2 信号并且进行相应操作；VGA 模块能否正确地将人物轨迹根据 PS2 控制信号表现出来；七段码显示模块，能否正确地将统计距离实时显示；基于以上的主功能外，还需完成各个具体功能的实现，具体如下：

- (1) 正确显示游戏初始界面，左上角有一个固定的迷宫；
- (2) 按下“A”按键，人物将向左移动；
- (3) 按下“B”按键，人物将向右移动；
- (4) 按下“C”按键，人物将向上移动；
- (5) 按下“D”按键，人物将向下移动；
- (6) 每次按下按键，分数加 1，七段码可以正确现实距离。

在本次实验中的测试主要采用上述报告中的实验步骤进行，物理验证采用写单独的简单汇编代码，通过 VGA 特殊化输出进行验证。

4.2 结果分析

(内容要点：分析验证结果、存在的问题及原因、最终的成果内容等)

1. 从验证结果和仿真结果来看，整个项目都能正确地按照预定的目标进行，最后从下板的结果来看，项目能够成功运行。其中各个辅助功能之间的切换以及参数的统计都能够正确地运行。

2. 存在的问题：由于在 VGA 模块中，虽然使用了 VRAM 双端口输入输出，但实际上由于实时显示与数据传输之间仍然存在时间差，因此 VGA 背景显示中可能会出现闪烁的情况。

由于该情况的存在，本游戏的最初设计“走迷宫”会出现一定的 bug。

1) “走迷宫”的最初设计原理是，初始化一张固定的迷宫图，即初始化对应 RGB 值，分为人物颜色，墙的颜色，路的颜色以及出口颜色四种不同颜色；

2) PS2 进行按键判断，然后更新人物坐标；

3) 判断当前坐标颜色，若为墙的颜色，则将人物坐标还原为变换前的值；若为路的颜色，则将当前坐标的 RGB 值改为人物的颜色，将变换前坐标还原为路的颜色；若为出口的颜色，则停止游戏，7 段码显示 WIN。

在“走迷宫”中判断条件为像素点的颜色，因此由于上述所述原因，无法保证人物移动的正确性。

解决方法：将判断条件由颜色改为坐标值，即存储迷宫中强的坐标，从而完成判断。

由于临近期末，5 个大程多门期末考试压身，因此在明白原理的基础，未再对该大程进行完善，可以考后对解决方法进行验证。现附上“走迷宫”的汇编代码，硬件架构可以直接

使用“星空漫步”的整个框架：

```

# VGA: $t0 0x 000C 0000
# PS2: 0x 000E 0000
# Left: $t1 +0
# Right: $t2 +1
# Up: $t3 +2
# Down: $t4 +3
# pre_player_x: $s0
# pre_player_y: $s1
# cur_player_x: $s2
# cur_player_y: $s3
# current color: $s4
# current PS2: $s5
# color:
# wall: $t5 red
# road: $t6 white
# player: $t7 yellow
# exit: $t8 green
# state:
# initial: draw maze
    # move: renew the color of player's position
    # fail: collide the wall, color is brown $t0
    # win: arrive exit, color is yellow $t4
# parameter: $a0, $a1, $a2, $a3

Initial:
#initialize all reg
add $t0, $zero, $zero;
add $t1, $zero, $zero;
add $t2, $zero, $zero;
add $t3, $zero, $zero;
add $t4, $zero, $zero;
add $t5, $zero, $zero;
add $t6, $zero, $zero;
add $t7, $zero, $zero;
add $t8, $zero, $zero;
add $s0, $zero, $zero;    # pre_player_x
add $s1, $zero, $zero;    # pre_player_y
add $s2, $zero, $zero;    # cur_player_x
add $s3, $zero, $zero;    # cur_player_y
add $s4, $zero, $zero;
add $s5, $zero, $zero;
add $a0, $zero, $zero;
add $a1, $zero, $zero;
```

```

add $a2, $zero, $zero;
add $a3, $zero, $zero;

#initialize vga
lui $t0, 0x000c; # VGA
lui $t1, 0x000e; # left
addi $t2, $t1, 1; # right
addi $t3, $t1, 2; # up
addi $t4, $t1, 3; # down
addi $t5, $t5, 0x0011; # wall: red
addi $t6, $t6, 0x0022; # road: white
addi $t7, $t7, 0x0033; # player: yellow
addi $t8, $t8, 0x0044; # exit: green

Maze:
#0
sw $t7, 0($t0);
sw $t6, 1($t0);
sw $t5, 2($t0);
sw $t5, 3($t0);
sw $t5, 4($t0);
sw $t5, 5($t0);
sw $t5, 6($t0);
sw $t5, 7($t0);
#1
sw $t6, 80($t0);
sw $t6, 81($t0);
sw $t6, 82($t0);
sw $t6, 83($t0);
sw $t6, 84($t0);
sw $t6, 85($t0);
sw $t5, 86($t0);
sw $t5, 87($t0);
#2
sw $t6, 160($t0);
sw $t5, 161($t0);
sw $t6, 162($t0);
sw $t5, 163($t0);
sw $t6, 164($t0);
sw $t5, 165($t0);
sw $t5, 166($t0);
sw $t5, 167($t0);
#3
sw $t6, 240($t0);

```

```

sw $t5, 241($t0);
sw $t5, 242($t0);
sw $t5, 243($t0);
sw $t6, 244($t0);
sw $t6, 245($t0);
sw $t6, 246($t0);
sw $t5, 247($t0);
#4
sw $t6, 320($t0);
sw $t6, 321($t0);
sw $t6, 322($t0);
sw $t5, 323($t0);
sw $t5, 324($t0);
sw $t5, 325($t0);
sw $t6, 326($t0);
sw $t5, 327($t0);
#5
sw $t5, 400($t0);
sw $t5, 401($t0);
sw $t6, 402($t0);
sw $t6, 403($t0);
sw $t6, 404($t0);
sw $t6, 405($t0);
sw $t6, 406($t0);
sw $t8, 407($t0);

Game:
#renew pre_position
add $s0, $s2, $zero;
add $s1, $s3, $zero;

left:
addi $a0, $zero, 1;
lw $a1, 0($t1);
addi $a2, $zero, 0;
addi $a3, $zero, 30000;
Delay2:
addi $a2, $a2, 1;
bne $a2, $a3, Delay2;
bne $a1, $a0, right;
sub $s2, $s0, $a0;    #cur_player_x = pre_plaer_x - 1;
j state;

right:

```

```

addi $a0, $zero, 1;
lw $a1, 0($t2);
bne $a1, $a0, up;
add $s2, $s0, $a0;    #cur_player_x = pre_plaer_x + 1;
j state;

up:
addi $a0, $zero, 1;
lw $a1, 0($t3);
bne $a1, $a0, down;
sub $s3, $s1, $a0;    #cur_player_y = pre_plaer_x - 1;
j state;

down:
addi $a0, $zero, 1;
lw $a1, 0($t4);
bne $a1, $a0, state;
add $s3, $s1, $a0;    #cur_player_y = pre_player_y + 1;
j state;

state:
# offset = y * 80 + x
add $a1, $zero, $zero;    # offset = 0;
add $a0, $zero, $zero;    # i = 0;
offsety0:
beq $a0, $s3, offsetx0; # i == cur_y;
addi $a1, $a1, 0x50; # offset += 80;
addi $a0, $a0, 1;      # i++;
j offsety0;
offsetx0:
add $a1, $a1, $s2;      # offset += cur_x;
add $a1, $a1, $t0;      # address of current color

lw $s4, 0($a1);         # $s4 = current color of new position

addi $a2, $zero, 0;
addi $a3, $zero, 30000;
Delay3:
addi $a2, $a2, 1;
bne $a2, $a3, Delay3;

beq $s4, $t5, statewall; # if new position is wall, stay
beq $s4, $t6, stateroad; # if new position is road, move
#beq $s4, $t8, stateexit; # if new position is exit, win

```

```

statewall:
    add $s2, $s0, $zero; # player will not move
    add $s3, $s1, $zero;
    j drawmaze;

stateroad:
#pre_position = road color
    add $a1, $zero, $zero; # offset = 0;
    add $a0, $zero, $zero; # i = 0;
    offsety1:
        beq $a0, $s1, offsetx1; # i == pre_y;
        addi $a1, $a1, 0x50; # offset += 80;
        addi $a0, $a0, 1; # i++;
        j offsety1;
    offsetx1:
        add $a1, $a1, $s0; # offset += pre_x;
        add $a1, $a1, $t0; # address of current color
        sw $t6, 0($a1); # reset road color
        j drawmaze;

# stateexit

drawmaze:
#cur_position = player color
    add $a1, $zero, $zero; # offset = 0;
    add $a0, $zero, $zero; # i = 0;
    offsety2:
        beq $a0, $s3, offsetx2; # i == cur_y;
        addi $a1, $a1, 0x50; # offset += 80;
        addi $a0, $a0, 1; # i++;
        j offsety2;
    offsetx2:
        add $a1, $a1, $s2; # offset += cur_x;
        add $a1, $a1, $t0; # address of current color
        sw $t7, 0($a1); # reset player color

        addi $a0, $zero, 0;
        addi $a1, $zero, 30000;
    Delay1:
        addi $a0, $a0, 1;
        bne $a1, $a1, Delay1;

    j Game;

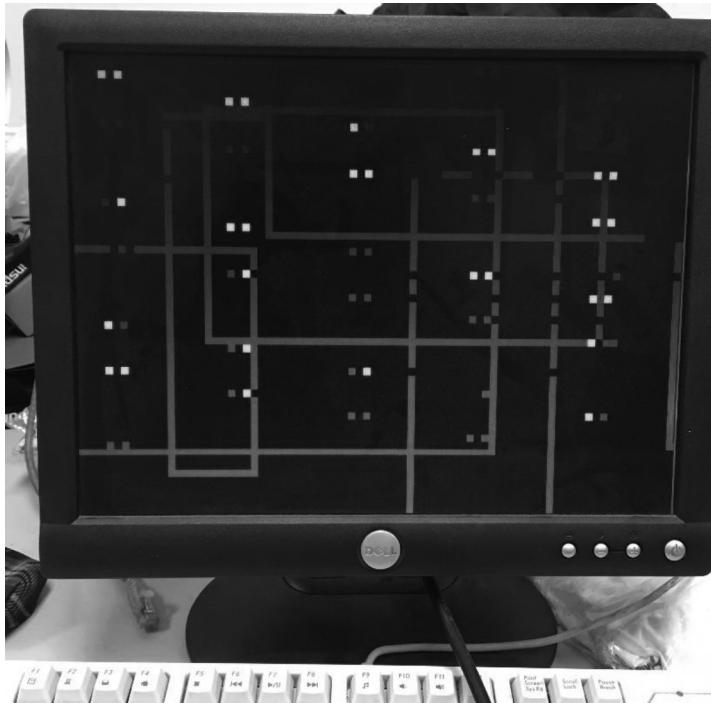
```

4.3 系统演示与操作说明

(内容要点：分析验证结果、存在的问题及原因、最终的成果内容；系统操作说明（使用说明），演示的主要结果截图（要有说明）等）

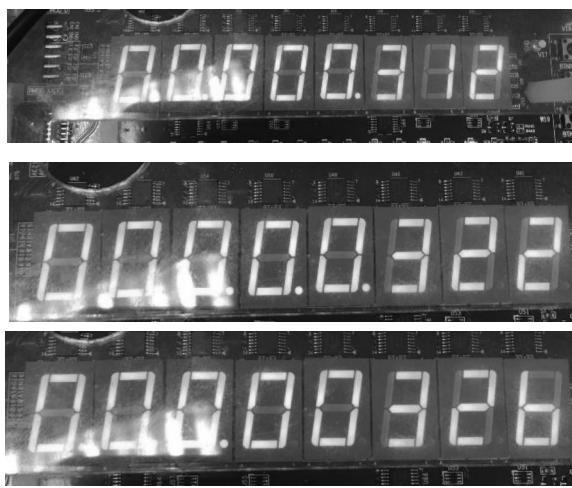
下板实践当中，初始化界面能够正常进行。如下图所示，首先界面会有一个游戏界面人物初始化于屏幕左上角。

当玩家按下相应的运动键以后，人物开始根据运动方向进行移动，在显示器上就会显示相应的运动轨迹。



图表 17 游戏 VGA 显示

与显示器对应的是，七段数码管也会显示相应的距离读数，如下图所示：



图表 2 七段数码管显示距离

第 5 章 结论与展望

(内容要点：简要讨论并叙述 Project 过程中的感受，以及其他的问题和自己的感想。)

本次 Project 的设计当中，需要充分地将游戏和 FPGA 实验板结合起来。虽然在数字逻辑中已经接触过 VGA 和 PS2 两个主要模块，但不通过硬件逻辑，完全通过 MCPU 完成整个实验还是具有一定挑战性。由于时序的问题，本次实验最终实现仍存在着一些闪屏的问题，但是通过对模块原理的不断学习以及模块测试，已经找到了问题的来源并理解，也让我认识到硬件在实现过程中不仅需要保证代码的准确性，也要考虑硬件实现过程中时序等问题的现实约束。

总体来说，本次实验过程大大加深了我对 MIO-BUS 的理解以及整个程序运行原理的理解，在此之前并不了解每次我们的跑马灯是怎样真正运行起来的，可以说是对整个框架有了总体的认知。因为在本次实验中使用了 VRAM 来存储，觉得数据传输非常精妙，并且在实验后真正的理解了数据在 RAM 以及 VRAM 的输入输出，非常有趣。也正是因为周末在实验室苦肝了三天完成了此次实验，在实验考试中才可以充分理解原理，完成七段码的学号现实。

最后作为施老师计算机兴趣小组的成员，这一年米几乎每周五老师和学长们的教学和分享，在实验室中和小伙伴们一起自学理论知识、调试程序，分享实验过程中遇到的问题和富有创意的想法，都让我收获很多。除了完成计组的相关内容外，平时也有很多时间在 512 实验室学习专业课程，觉得实验室的氛围特别好。

5.1 展望

本项目在 FPGA 实验板上能够顺利实现的“星空漫步”基本功能和其他辅助功能。在 FPGA 实现 VGA 显示，可以通过调节时钟频率或其他方式减少由于同步性而存在的闪烁问题；同时可以在该游戏的基础上，通过坐标值的逻辑判断而不是 RGB 值的判断来完成游戏的最初构想“走迷宫”。

附录 .ucf

```

#系统时钟
NET "clk_100mhz"      LOC = AC18      | IOSTANDARD = LVCMOS18 ;
NET "RSTN"              LOC = W13       | IOSTANDARD = LVCMOS18 ;
NET "clk_100mhz"      TNM_NET = TM_CLK ;
TIMESPEC TS_CLK_100M = PERIOD "TM_CLK" 10 ns HIGH 50%;

#LED 串行接口
NET "LEDCLK"           LOC = N26      | IOSTANDARD = LVCMOS33 ;
NET "LEDCLR"            LOC = N24      | IOSTANDARD = LVCMOS33 ;
NET "LEDDT"              LOC = M26      | IOSTANDARD = LVCMOS33 ;
NET "LEDEN"              LOC = P18      | IOSTANDARD = LVCMOS33 ;

#七段码串行接口
NET "SEGCLK"            LOC = M24      | IOSTANDARD = LVCMOS33 ;
NET "SEGCLR"            LOC = M20      | IOSTANDARD = LVCMOS33 ;
NET "SEGDT"              LOC = L24      | IOSTANDARD = LVCMOS33 ;
NET "SEGEN"              LOC = R18      | IOSTANDARD = LVCMOS33 ;

#三色信号灯: Tri_LED
NET "RDY"                LOC = U21      | IOSTANDARD = LVCMOS33 ; #LED_nR0
NET "readn"               LOC = U22      | IOSTANDARD = LVCMOS33 ; #LED_nG0
NET "CR"                  LOC = V22      | IOSTANDARD = LVCMOS33 ; #LED_nB0
#NET "LED_nR1"            LOC = U24      | IOSTANDARD = LVCMOS18 ;
#NET "LED_nG1"            LOC = U25      | IOSTANDARD = LVCMOS18 ;
#NET "LED_nB1"            LOC = V23      | IOSTANDARD = LVCMOS18 ;

#阵列式按键
NET "K_ROW[0]"           LOC = V17      | IOSTANDARD = LVCMOS18 ; #ROW0
NET "K_ROW[1]"           LOC = W18      | IOSTANDARD = LVCMOS18 ; #ROW1
NET "K_ROW[2]"           LOC = W19      | IOSTANDARD = LVCMOS18 ; #ROW2
NET "K_ROW[3]"           LOC = W15      | IOSTANDARD = LVCMOS18 ; #ROW3
NET "K_ROW[4]"           LOC = W16      | IOSTANDARD = LVCMOS18 ; #ROW4
NET "K_COL[0]"            LOC = V18      | IOSTANDARD = LVCMOS18 ; #COL0
NET "K_COL[1]"            LOC = V19      | IOSTANDARD = LVCMOS18 ; #COL1
NET "K_COL[2]"            LOC = V14      | IOSTANDARD = LVCMOS18 ; #COL2
NET "K_COL[3]"            LOC = W14      | IOSTANDARD = LVCMOS18 ; #COL3

#switch
NET "SW[0]"              LOC = AA10     | IOSTANDARD = LVCMOS15 ;
NET "SW[1]"              LOC = AB10     | IOSTANDARD = LVCMOS15 ;
NET "SW[2]"              LOC = AA13     | IOSTANDARD = LVCMOS15 ;
NET "SW[3]"              LOC = AA12     | IOSTANDARD = LVCMOS15 ;
NET "SW[4]"              LOC = Y13      | IOSTANDARD = LVCMOS15 ;
NET "SW[5]"              LOC = Y12      | IOSTANDARD = LVCMOS15 ;
NET "SW[6]"              LOC = AD11     | IOSTANDARD = LVCMOS15 ;

```

```

NET "SW[7]"           LOC = AD10 | IOSTANDARD = LVCMOS15 ;
NET "SW[8]"           LOC = AE10 | IOSTANDARD = LVCMOS15 ;
NET "SW[9]"           LOC = AE12 | IOSTANDARD = LVCMOS15 ;
NET "SW[10]"          LOC = AF12 | IOSTANDARD = LVCMOS15 ;
NET "SW[11]"          LOC = AE8  | IOSTANDARD = LVCMOS15 ;
NET "SW[12]"          LOC = AF8  | IOSTANDARD = LVCMOS15 ;
NET "SW[13]"          LOC = AE13 | IOSTANDARD = LVCMOS15 ;
NET "SW[14]"          LOC = AF13 | IOSTANDARD = LVCMOS15 ;
NET "SW[15]"          LOC = AF10 | IOSTANDARD = LVCMOS15 ;

#ArDUNIO-Sword-002-Basic IO
NET "Buzzer"         LOC = AF24 | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[0]"      LOC = AB22 | IOSTANDARD = LVCMOS33 ;#a
NET "SEGMENT[1]"      LOC = AD24 | IOSTANDARD = LVCMOS33 ;#b
NET "SEGMENT[2]"      LOC = AD23 | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[3]"      LOC = Y21  | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[4]"      LOC = W20  | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[5]"      LOC = AC24 | IOSTANDARD = LVCMOS33 ;
NET "SEGMENT[6]"      LOC = AC23 | IOSTANDARD = LVCMOS33 ;#g
NET "SEGMENT[7]"      LOC = AA22 | IOSTANDARD = LVCMOS33 ;#point

NET "AN[0]"           LOC = AD21 | IOSTANDARD = LVCMOS33 ;
NET "AN[1]"           LOC = AC21 | IOSTANDARD = LVCMOS33 ;
NET "AN[2]"           LOC = AB21 | IOSTANDARD = LVCMOS33 ;
NET "AN[3]"           LOC = AC22 | IOSTANDARD = LVCMOS33 ;

NET "LED[0]"          LOC = AB26 | IOSTANDARD = LVCMOS33 ;
NET "LED[1]"          LOC = W24  | IOSTANDARD = LVCMOS33 ;
NET "LED[2]"          LOC = W23  | IOSTANDARD = LVCMOS33 ;
NET "LED[3]"          LOC = AB25 | IOSTANDARD = LVCMOS33 ;
NET "LED[4]"          LOC = AA25 | IOSTANDARD = LVCMOS33 ;
NET "LED[5]"          LOC = W21  | IOSTANDARD = LVCMOS33 ;
NET "LED[6]"          LOC = V21  | IOSTANDARD = LVCMOS33 ;
NET "LED[7]"          LOC = W26  | IOSTANDARD = LVCMOS33 ;

NET "PS2C"             LOC = N18  | IOSTANDARD = LVCMOS33 ;
NET "PS2D"             LOC = M19  | IOSTANDARD = LVCMOS33 ;

#NET "PS2_clk"         LOC = N18  | IOSTANDARD = LVCMOS33 | SLEW
= FAST | PULLUP ;
#NET "PS2_data"        LOC = M19  | IOSTANDARD = LVCMOS33 | SLEW
= FAST | PULLUP ;

NET "blue[0]"          LOC = T20  | IOSTANDARD = LVCMOS33 | SLEW

```

```
= FAST ;
    NET "blue[1]"          LOC = R20 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "blue[2]"          LOC = T22 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "blue[3]"          LOC = T23 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "green[0]"         LOC = R22 | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
    NET "green[1]"         LOC = R23 | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
    NET "green[2]"         LOC = T24 | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
    NET "green[3]"         LOC = T25 | IOSTANDARD = LVCMOS33 | SLEW =
FAST ;
    NET "red[0]"           LOC = N21 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "red[1]"           LOC = N22 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "red[2]"           LOC = R21 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "red[3]"           LOC = P21 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "hsync"             LOC = M22 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
    NET "vsync"             LOC = M21 | IOSTANDARD = LVCMOS33 | SLEW
= FAST ;
```

```
state[3:0]
0000 game stop/game not start
0001 game lose
0010 stand back
0011 walk back
0100 stand front
0101 walk front
0110 up back
0111 up front
1000 only up
1001 marry's state doesn't change
1010 pause
```

btn<3:0>:

按键	按键在实验板编号	状态
btn<0>	B8	游戏开始\重置
btn<1>	C4	左移动
btn<2>	D9	右移动
btn<3>	A8	上跳
btn<4>	C9	暂停

键盘上键的对应功能:

space 暂停
Enter 游戏开始\重置

多路选择器关于 pos[2:0]状态

pos[2:0]

000	stand back
001	walk back
010	stand front
011	walk front
100	not appear