浙江大学

课程设计报告

题目： 基于数字系统的俄罗斯方块游戏

姓名：曾奕久

学号：**3170105966**

指导教师：施青松、洪奇军

Date: 2019-1-15

基于数字系统的俄罗斯方块游戏

摘要：本次课程设计以俄罗斯方块作为课题，主要完成了基于数字系统的俄罗斯方块小游戏的实现，下文主要介绍了关于俄罗斯方块设计的背景、功能、具体实现、结果验证与展望等要素，并对课题完成过程中所遇到的困难和解决方法做了分析，最后讨论了本人对于课程学习的收获。

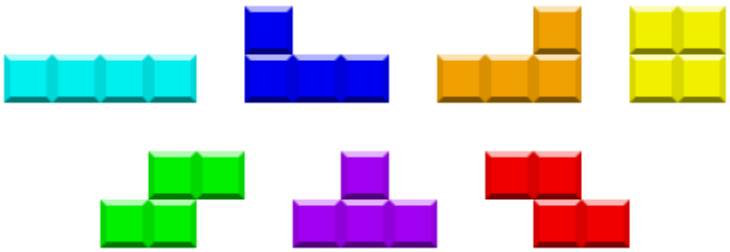
关键词：数字逻辑；俄罗斯方块；Verilog

一、绪论

1.1 游戏背景介绍

《俄罗斯方块》（Tetris）是一款由俄罗斯人阿列克谢·帕基特诺夫于 1984 年 6 月发明的休闲游戏。游戏过程中，由小方块组成的不同形状的板块陆续从屏幕上方落下来，玩家通过调整板块的位置和方向，使它们在屏幕底部拼出完整的一条或几条。这些完整的横条会随即消失，给新落下来的板块腾出空间，与此同时，玩家得到分数奖励。没有被消除掉的方块不断堆积起来，一旦堆到屏幕顶端，玩家便告输，游戏结束。

俄罗斯方块的标准大小行宽为 10，列高为 20，以每个小正方形为单位。一组由 4 个小型正方形组成的规则图形，英文称为 Tetromino，中文通称为方块，一共有 7 种类型。



图表 1 游戏基本方块类型

游戏基本规则如下：

（1）玩家可以以 90° 为单位旋转方块，以格子为单位左右移动方块，让方块加速落下。

（2）方块移到区域最下方或是着地到其他方块上无法移动时，就会固定在该处，而新的方块出现在区域上方开始落下。

（3）当区域中某一列横向格子全部由方块填满，则该列会消失并成为玩家的得分。

1.2 课题功能与接口

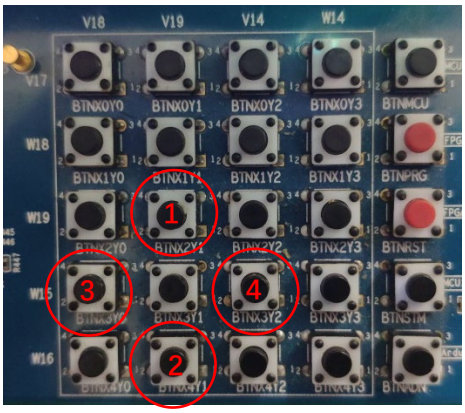
1.2.1 PS/2 键盘输入（未完全实现）

支持上下左右四个方向的输入信号，玩家可以通过按下对应的按键进行游戏，其中"←"

代表方块左移一格；"→"代表方块右移一格；"↑"表示旋转方块；“↓”表示将方块快速下落到最低可以下落的位置。

1.2.2 阵列键输入

支持阵列键其中的四个进行游戏操作，如图所示：



图表 2 阵列键使用说明

图中标注的 1 号按键可以旋转方块，2 号按键可以使方块快速下落，3 号键是方块左一一格，4 号键使方块右移一格。

1.2.3 开关输入

最右端 SW[0]控制 Vga 板是否输出图像，最左端 SW[15]控制游戏挂起状态，即 SW[15]被拨起时，游戏内数据重置且停止游戏，直到 SW[15]再被置 0，游戏再次开始。

二、课题设计

2.1 理论基础

2.1.1VGA 显示

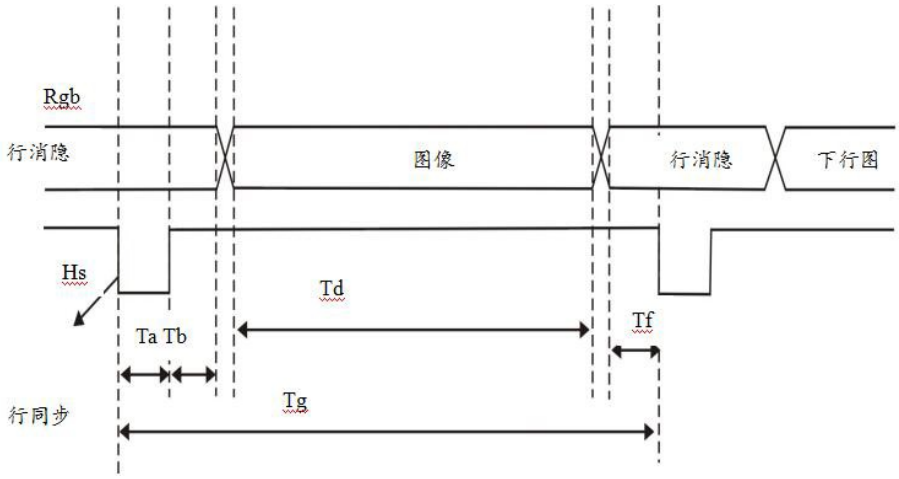
VGA(Video Graphics Array)作为一种标准的显示接口得到了广泛的应用，其信号类型为模拟类型，显示卡端的接口为 15 针母插座。VGA 在任何时刻都必须工作在某一显示模式之下，其显示模式分为字符显示模式和图形显示模式，在应用中，讨论的都是图形显示模式。工业标准的 VGA 显示模式为：640* 480* 16 色* 60Hz。

常见的彩色显示器一般都是由 CRT(阴极射线管)构成，其引出线共含 5 个信号：R、G、B(三基色信号)、HS(行同步信号)、VS(场同步信号)。每一个像素的色彩由 R(红,Red) .G(绿，Green). B(蓝，Blue)三基色构成，在实验的验证阶段可以仅利用 R、G、B 三种基色的一元化值(0 和 1)的不同组合来验证设计的正确性。

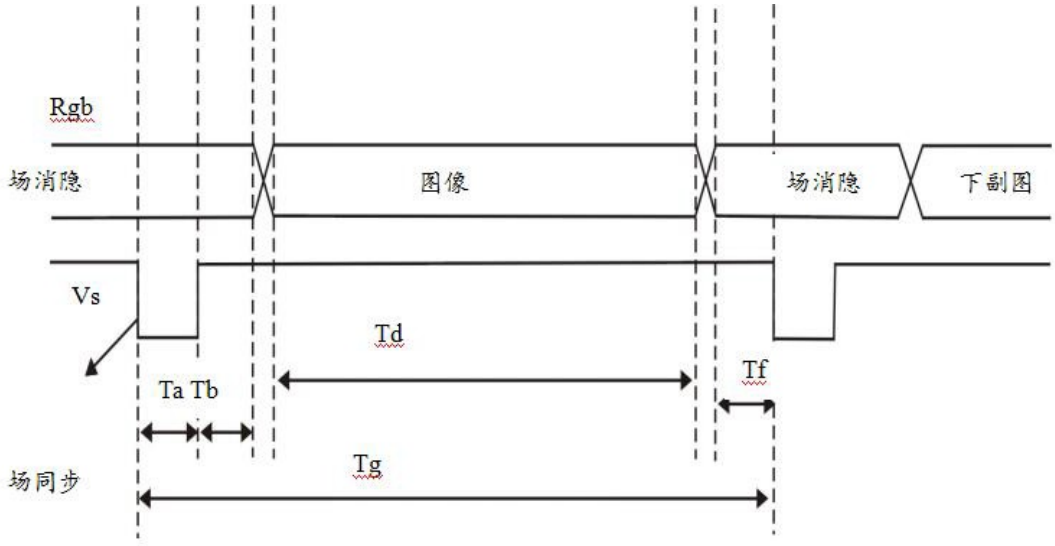
电子束扫描一幅屏幕图像上的各个点的过程称为屏幕扫描。现在显示器都是通过光栅扫描的方式来进行屏幕扫描。在光栅扫描方式下，，由显示模块产生的水平同步信号和垂直同

步信号控制阴极射线管中的电子枪产生电子束，使之按照固定的路径扫过整个屏幕，如水平扫描、水平回扫、垂直扫描、垂直回扫等，在扫描过程中通过电子束的通断强弱来控制电子束所经过的每个点是否显示或显示的颜色，于显示屏上合成一个彩色像素点。光栅扫描的一般过程：电子束从屏幕左上角开始向右扫，当到达屏幕的右边缘时，电子束关闭(水平消隐)，并快速返回屏幕左边缘(水平回扫)， 然后在下一条扫描线上开始新的一次水平扫描。一旦所有的水平扫描均告完成，电子束在屏幕的右下角结束并关闭(垂直消隐)，然后迅速返回到屏幕的左上角(垂直回扫)，开始下一次光栅扫描。

VGA 时序控制模块是本设计的重要部分，最终的输出信号行、场同步信号必须严格按照 VGA 时序标准产生相应的脉冲信号。在 5 个信号时序驱动时，时序信号包括前两个，它们都有图像显示区和图像消隐区，图像消隐区又分为消隐前肩、同步脉冲区和消隐后肩。



图表 3 VGA 行扫描时序图



图表 4 VGA 场扫描的时序图

2.1.2 PS/2 键盘

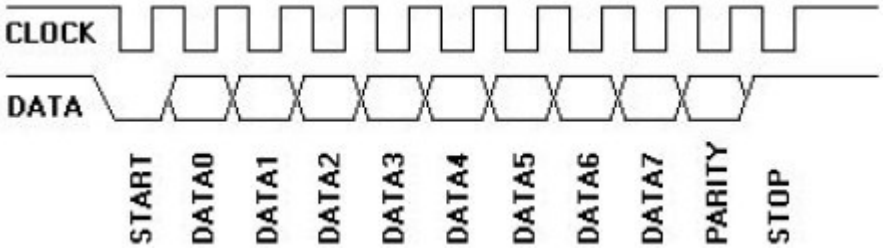
PS/2 通信协议是一种双向同步串行通信协议。通信的两端通过 CLOCK(时钟脚)同步，并通过国 DATA(数据脚)交换数据。一般两台设备间传输数据的最大时钟频率是 33kHz，大多数 PS/2 设备工作在 10--20kHz。推荐值在 15kHz 左右，也就是说，CLOCK 高、低电平

的持续时间都为 40us。每一数据帧包含 11—12 位，具体含义如下图所示。

图表 5 PS/2 通信数据帧格式

数据	含义
1个起始位	总是逻辑0
8个数据位	(LSB)低位在前
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

PS/2 到主机的通信时序如下图所示。数据在 PS/2 时钟的下降沿读取，PS/2 的时钟频率为 10—16.7kHz。对于 PS/2 设备，一般来说从时钟脉冲的上升沿到一个数据转变的时间至少要有 5us；数据变化到下降沿的时间至少要有 5us，并且不大于 25us，这个时序非常重要应该严格遵循。主机可以在第 11 个时钟脉冲停止位之前把时钟线拉低，使设备放弃发送当前字节，当然这种情况比较少见。在停止位发送后设备在发送下个包前应该至少等待 50us，给主机时间做相应的处理。不过主机处理接收到的字节时一般会抑制发送(主机在收到每个包时通常自动做这个)。在主机释放抑制后，设备至少应该在发送任何数据前等 50us。



图表 6 设备到主机的通信

2.1.3 状态机的设计

有限状态机（Finite State Machine FSM）是时序电路设计中经常采用的一种方式，尤其适合设计数字系统的控制模块，在一些需要控制高速器件的场合，用状态机进行设计是一种很好的解决问题的方案，具有速度快、结构简单、可靠性高等优点。有限状态机非常适合用 FPGA 器件实现，用 Verilog HDL 的 case 语句能很好地描述基于状态机的设计，再通过 EDA 工具软件的综合，一般可以生成性能极优的状态机电路，从而使其在执行时间、运行速度和占用资源等方面优于用 CPU 实现的方案。

有限状态机一般包括组合逻辑和寄存器逻辑两部分，寄存器逻辑用于存储状态，组合逻辑用于状态译码和产生输出信号。根据输出信号产生方法的不同，状态机可分为两类： 米里型(Mealy)和摩尔型(Moore)。摩尔型状态机的输出只是当前状态的函数。米里型状态机的输出是在输入变化后立即变化的，不依赖时钟信号的同步，摩尔型状态机的输入发生变化时还需要等待时钟的到来，必须在状态发生变化时才会导致输出的变化，因此比米里型状态机要多等待一个时钟周期。

2.1.4 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， 还可以表示数字逻辑系统所完成的逻辑功能。
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。

2.1.5 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。

2.2 实验器材

2.2.1 ISE 软件

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。

2.2.2 SWORD 板

采用开放式体系构架和 32 位存储层次结构，通用性强，适用性灵活。实验方法采用基于 FPGA 实体的虚拟实验箱和 SOC 集成技术，支持个性化开发、课程设计和创新实践。 系统资源可很好地支持数字电路、计算机组成、计算机体系结构、接口通讯、编译技术、操作系统、计算机网络、基于 IP 核的嵌入式系统和多媒体等课程的教学实践。

2.2.3 计算机

2.2.4 PS 软件

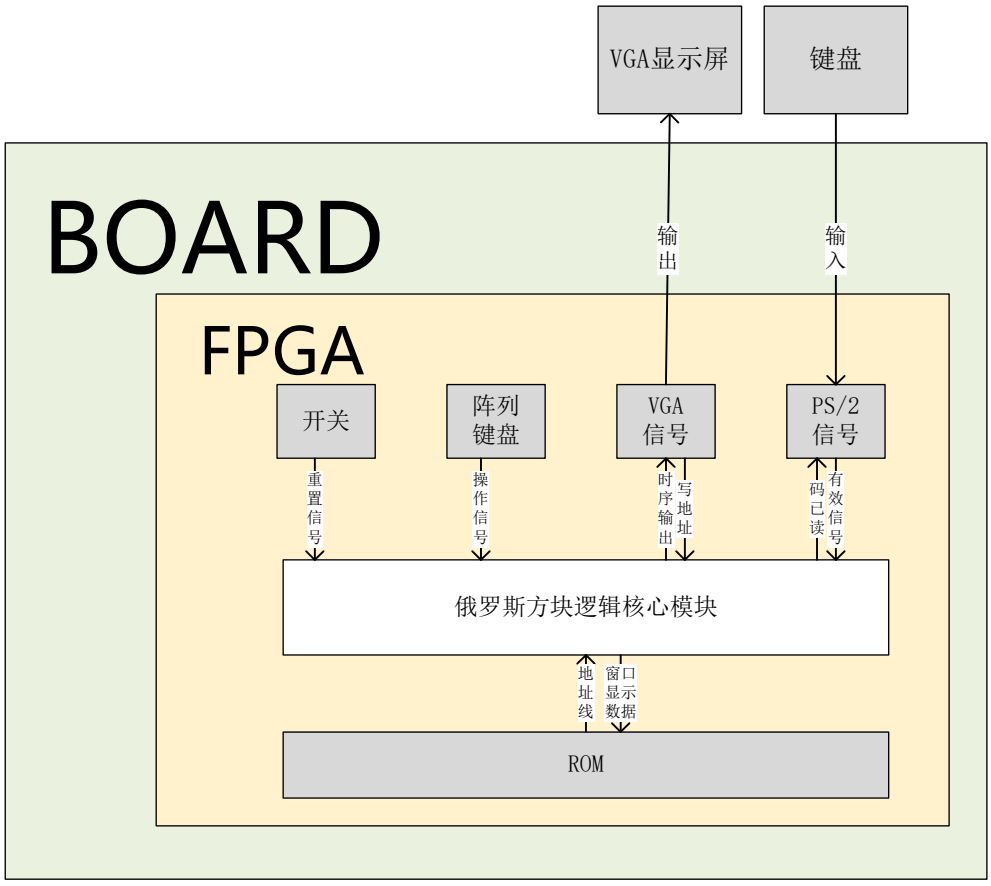
进行图片编辑工作，包括修改图片大小以及查看图片像素，或者设计心仪的图案。

2.2.5 Matlab 软件

将.jpg 格式图像文件转换位 3 位 16 进制.coe 文件，便于 IP 核调用传输图像。

2.3 设计方案

2.3.1 整体设计方案

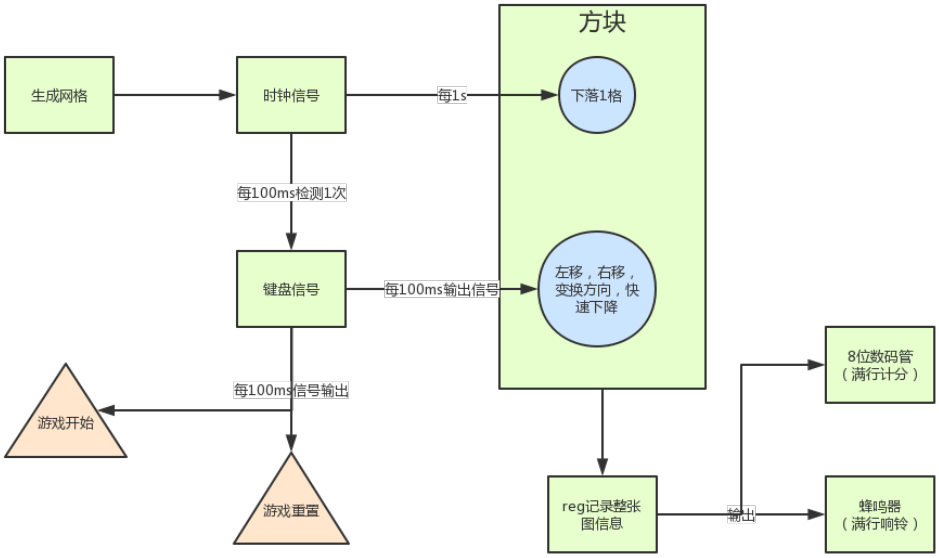


图表 7 硬件结构框图

如图所示，整个课程设计由以上部分构成，其中 VGA 接口负责 VGA 显示屏与核心模块之间的数据传输；阵列键盘输入作为控制与测试信号；ROM 存储需要显示的图片颜色信息，使画面更加精美；俄罗斯方块逻辑核心模块处理输入数据，进行主逻辑的状态机运行，并进行输出。

2.3.2 俄罗斯方块逻辑核心模块设计

为了实现俄罗斯方块游戏的基本功能，设计逻辑流程如下图：



图表 8 俄罗斯方块逻辑核心模块设计流程图

为了方便检测以及思考，我将整个逻辑模块的任务划分为三块：生成对应的形状、方块移动、消除并计算分数。对于外界操作，转向由第一个部分（生成形状）完成，左移、右移以及快速下落均由第二部分完成。整个程序都由数组储存信息（并行信号），1 代表有方块，0 代表没有方块。

对于生成形状部分，在考虑到俄罗斯方块的基本块的形状后，我使用一个 4* 4 的数组储存不同图像不同方向的信息，而且刚好可以用四位 16 进制数枚举，非常方便，同时为了便于形状是否靠边的判断，这里特地增加了一个名为 wid 的输出，用来表示当前形状最大宽度。

```
module Shape(input [2:0]type,
             input [1:0]dir,
             output reg[15:0]shape,
             output reg[1:0]wid
    );
always@(*) begin
    case(type)
        3'b000:begin shape <= 16'h00cc; wid <= 1;end
        3'b001:case(dir)
                 2'b10,
                 2'b00:begin shape <= 16'h00c6; wid <= 2;end
                 2'b11,
                 2'b01:begin shape <= 16'h04c8; wid <= 1;end
               endcase
        3'b010:case(dir)
                 2'b10,
                 2'b00:begin shape <= 16'h006c; wid <= 2;end
                 2'b11,
                 2'b01:begin shape <= 16'h08c4; wid <= 1;end
```

```
                endcase
        3'b011:case(dir)
                    2'b00:begin shape <= 16'h004e; wid <= 2;end
                    2'b01:begin shape <= 16'h08c8; wid <= 1;end
                    2'b10:begin shape <= 16'h00e4; wid <= 2;end
                    2'b11:begin shape <= 16'h04c4; wid <= 1;end
                endcase
        3'b100:case(dir)
                    2'b00:begin shape <= 16'h008e; wid <= 2;end
                    2'b01:begin shape <= 16'h0c88; wid <= 1;end
                    2'b10:begin shape <= 16'h00e2; wid <= 2;end
                    2'b11:begin shape <= 16'h088c; wid <= 1;end
                endcase
        3'b101:case(dir)
                    2'b00:begin shape <= 16'h002e; wid <= 2;end
                    2'b01:begin shape <= 16'h044c; wid <= 1;end
                    2'b10:begin shape <= 16'h00e8; wid <= 2;end
                    2'b11:begin shape <= 16'h0c44; wid <= 1;end
                endcase
        3'b110:case(dir)
                    2'b10,
                    2'b00:begin shape <= 16'h000f; wid <= 3;end
                    2'b11,
                    2'b01:begin shape <= 16'h8888; wid <= 0;end
                endcase
        default: ;
    endcase
end
endmodule
```

对于方块移动部分，我采用 PS 中“图层叠加”的思想，设置一个储存区 already 和一个缓存区 move 以及最终输出区 window，利用 x、y 记录形状在整个输出窗口中的位置。每一次如果 1 秒内无操作，则 y 自增 1，表示下落一行，同时利用 x、y 数据将当前的图形打印在缓存区对应的位置上，如果缓存区与储存区的图像有重合，则将上一次的输出储存，重置缓存区；如果没有重合，则输出缓存区与储存区进行“或”操作之后的结果，也就是将两个图层叠加。此外，如果不到一秒就有左移或右移操作，同样打印形状后判断。

```
always@(posedge clk100)begin
    if(SW[15])begin
        window = 0;
        already = 0;
        dir = 0;
        move = 0;
        start = 1;
        score <= 0;
```

```
        show = 0;
        startbuzzer = 0;
    end
    else if(start) begin
        if(right) x = x + 1;
        if(left) x = x - 1;
        if(x < 0) x = 0;
        if(x >= 10 - wid) x = 10 - wid - 1;
        if(down) begin
            y = 9;
            for(i = 9; i >= 0; i = i - 1)begin
                if((already[i * 10 + x] & shape[3])||(already[i * 10 +
x + 1] & shape[2])||(already[i * 10 + x + 2] & shape[1])||(already[i
* 10 + x + 3] & shape[0]))
                    y = i - 1;
            end
        end
        if(turn) dir = (dir + 1)%4;
        if(con == 9)begin scan = 1; con = 0;end
        else begin con = con + 1; scan = 0;end

        if(startcountbz)begin
                countbuzzer=countbuzzer+1;
                startbuzzer=1;
        end
        if(countbuzzer==6)begin
            countbuzzer=0;
            startcountbz=0;
            startbuzzer=0;
        end

        if(scan || left || right || down)begin
            if(move == 0)begin
                x = 5;
                y = 0;
                type = (type + 1)%7;
                move[8:5] = {shape[0],shape[1],shape[2],shape[3]};
            end
            else begin

                move = 0;
                for(i = 0; i <= wid; i = i + 1)begin
                    move[y * 10 + x + i] = shape[3 - i];
                    if(y > 0) move[y * 10 + x - 10 + i] = shape[7 - i];
```

```
                if(y > 1) move[y * 10 + x - 20 + i] = shape[11 -
i];
                if(y > 2) move[y * 10 + x - 30 + i] = shape[15 -
i];
            end
        end
        flag = 0;
        for(i = 0; i <= 99; i = i + 1)
        if(move[i]&already[i]) flag = 1;
        if(flag||(y > 9))begin
                if(y <= 1) start = 0;
                x = 5;
                y = 0;
                already = window;
                move = 0;
        end
        else begin
            window = already | move;
            if(scan)y = y + 1;
        end
        j = 9;
        count = 0;
        show = 0;
        for(i = 9; i >= 0; i = i - 1)begin
            flag = window[i * 10 + 9];
            for(k = 8; k >= 0; k = k - 1)
                flag = window[i * 10 + k] & flag;
            if(flag) count = count + 1;
            else begin
                for(k = 9; k >= 0; k = k - 1)
                    show[j * 10 + k] = window[i * 10 + k];
                j = j - 1;
            end
        end
    end
end
```

对于快速下落操作，则需要单独检测当前形状可以下落的最大位置，我通过循环检测储存区在当前图像 x 坐标对应的位置是否与形状对应的位置有重叠，找到可以下落的最低点，并对 y 进行赋值后输出。

```
if(down) begin
        y = 9;
        for(i = 9; i >= 0; i = i - 1)begin
            if((already[i * 10 + x] & shape[3])||(already[i * 10 +
```

```
x + 1] & shape[2])||(already[i * 10 + x + 2] & shape[1])||(already[i
* 10 + x + 3] & shape[0]))
                    y = i - 1;
            end
        end
```

对于消除以及计算分数部分，为了降低模块间的粘连性，便于调试排查问题，我并未在第二个模块中设置关联触底后才积分，而是使用时序电路，每一秒检测一次，通过循环将前一部分输出的数组 window 的每一行复制到“真输出”数组 show 上，如果行中全为 1，则该行不复制，并且计数。最后根据计数结果累计相应分数，并且更新储存区和缓存区，蜂鸣器响铃 0.5s。

```
if(j >= 0)begin already = show; startcountbz = 1; window = show; move
= 0;end
            case(count)
                1:begin score <= score + 3; count = 0;end
                2:begin score <= score + 7; count = 0;end
                3:begin score <= score + 12; count = 0;end
                4:begin score <= score + 20; count = 0;end
                default: count = 0;
            endcase
            if(score[3:0] > 4'b1001)begin
                score[7:4] <= score[7:4] + 1;
                score[3:0] <= score[3:0] - 10;
            end
            if(score[7:4] > 4'b1001)begin
                score[11:8] <= score[11:8] + 1;
                score[7:4] <= score[7:4] - 10;
            end
            if(score[11:8] > 4'b1001)begin
                score[15:12] <= score[15:12] + 1;
                score[11:8] <= 0;
            end
            if(score[15:12] > 4'b1001)begin
                score[19:16] <= score[19:16] + 1;
                score[15:12] <= 0;
            end
            if(score[19:16] > 4'b1001)begin
                score[23:20] <= score[23:20] + 1;
                score[19:16] <= 0;
            end
            if(score[23:20] > 4'b1001)begin
                score[27:24] <= score[27:24] + 1;
                score[23:20] <= 0;
            end
```

```
            if(score[27:24] > 4'b1001)begin
                score[31:28] <= score[31:28] + 1;
                score[27:24] <= 0;
            end
            if(score[31:28] > 4'b1001)begin
                score[31:28] <= 0;
            end
        end
    end
end

    integer m;
    always@(*)begin
        address=350 * row_addr+col_addr;
        if(!start)begin
            vga_data <= ipcolor;
        end
        else begin
        if(((col_addr>=10'd119&&col_addr<=10'd121)
                ||(col_addr>=10'd159&&col_addr<=10'd161)
                ||(col_addr>=10'd199&&col_addr<=10'd201)
                ||(col_addr>=10'd239&&col_addr<=10'd241)
                ||(col_addr>=10'd279&&col_addr<=10'd281)
                ||(col_addr>=10'd319&&col_addr<=10'd321)
                ||(col_addr>=10'd359&&col_addr<=10'd361)
                ||(col_addr>=10'd399&&col_addr<=10'd401)
                ||(col_addr>=10'd439&&col_addr<=10'd441)
                ||(col_addr>=10'd479&&col_addr<=10'd481)
                ||(col_addr>=10'd519&&col_addr<=10'd521)
                ||(row_addr>=9'd39&&row_addr<=9'd41)
                ||(row_addr>=9'd79&&row_addr<=9'd81)
                ||(row_addr>=9'd119&&row_addr<=9'd121)
                ||(row_addr>=9'd159&&row_addr<=9'd161)
                ||(row_addr>=9'd199&&row_addr<=9'd201)
                ||(row_addr>=9'd239&&row_addr<=9'd241)
                ||(row_addr>=9'd279&&row_addr<=9'd281)
                ||(row_addr>=9'd319&&row_addr<=9'd321)
                ||(row_addr>=9'd359&&row_addr<=9'd361)
                ||(row_addr>=9'd399&&row_addr<=9'd401)
                ||(row_addr>=9'd439&&row_addr<=9'd441))
                &&(col_addr>=10'd119)
                &&(col_addr<=10'd521)
                &&(row_addr>=9'd39)
                &&(row_addr<=9'd441))
```

```
            vga_data <= 12'h000;
        else vga_data <= 12'hfff;

        for(m=0;m<=99;m=m+1)begin
            if(show[m] == 1)begin

    if(row_addr<79+40*(m/10)&&row_addr>41+40*(m/10)&&col_addr<159+4
0*(m%10)&&col_addr>121+40*(m%10))begin
                    vga_data<=12'h008;
                end
            end
        end
        end
```

2.3.3 开关接口设计

开关接口属于硬件接口，同时基于以前已经做过的实验，本课程设计中同样使用了开关防抖动机制，再将“过滤”后的开关状态输入，作为逻辑模块判断游戏是否开始的信号以及是否开始输出 VGA 信号。

开关防抖动机制即设置一个合适的时长，检测到状态变化后进行计时，如果一定时间内（较短）状态无变化，说明可以稳定输出变化信号。

```
module AntiJitter(
    input clk, input I, output reg O
);
    parameter WIDTH = 20;
    reg [WIDTH-1:0] cnt = 0;

    always @ (posedge clk)
    begin
        if(I)
        begin
            if(&cnt)
                O <= 1'b1;
            else
                cnt <= cnt + 1'b1;
        end
        else
        begin
            if(|cnt)
                cnt <= cnt - 1'b1;
            else
                O <= 1'b0;
        end
```

```
    end

endmodule
```

2.3.4 阵列键盘接口设计

SWORD 板上有由 4*5 个按键组成的阵列键盘，在本实验中用 9 位 2 进制数表示按下按键的行列信息，通过主时钟 clk（25MHz 脉冲）进行扫描，从而确定哪个键被按下，在代码中体现如下：

```
//输入输出
inout [3:0] keyX,
inout [4:0] keyY,

reg [3:0] keyLineX;//X 按键信息
reg [4:0] keyLineY;//Y 按键信息

//在不同状态下存储 X,Y 按键信息
always @ (posedge clk)
begin
    if(state)
        keyLineY <= keyY;
    else
        keyLineX <= keyX;
        state <= ~state;
end

//当 X,Y 信息发生改变时，将变化记录到输出 keyCode 中，不同按键对应不同数值
always @*
begin
case(keyLineX)
4'b1110: keyCode[1:0] <= 2'h0;
4'b1101: keyCode[1:0] <= 2'h1;
4'b1011: keyCode[1:0] <= 2'h2;
default: keyCode[1:0] <= 2'h3;
endcase
case(keyLineY)
5'b11110: keyCode[4:2] <= 3'h0;
5'b11101: keyCode[4:2] <= 3'h1;
5'b11011: keyCode[4:2] <= 3'h2;
5'b10111: keyCode[4:2] <= 3'h3;
5'b01111: keyCode[4:2] <= 3'h4;
default: keyCode[4:2] <= 3'h7;
endcase
```

```
end
```

通过 AntiJitter 模块进行防抖动操作后，可以运用到本实验中。

三、实现与验证

3.1 遇到的困难与解决方案

3.1.1 VGA 模块理解与实现

3.1.1.1 VGA 原理

要实现俄罗斯方块必须先搞懂 VGA 的实现原理，VGA 显示器的输入和对应功能如下表所示：

信号线	定义
HS	列同步信号（3.3V 电平）
VS	行同步信号（3.3V 电平）
R	红基色（0-0.714V 模拟信号）
G	绿基色（0-0.714V 模拟信号）
B	蓝基色（0-0.714V 模拟信号）

图表 9 VGA 标准五输入对应功能

由表可知，HS 用于控制列扫描的频率，VS 用于控制行扫描的频率。R，G，B 的组合控制最终每个像素点颜色的显示。在代码中体现如下：

```
//行扫描
always @ (posedge vga_clk) begin
       if (!clrn) begin  //使能控制
           h_count <= 10'h0;
       end else if (h_count == 10'd799) begin//799 次 clk 上升沿一个周期
           h_count <= 10'h0;
       end else begin
           h_count <= h_count + 10'h1;
       end
   end
//列扫描
always @ (posedge vga_clk or negedge clrn) begin
       if (!clrn) begin //使能控制
           v_count <= 10'h0;
       end else if (h_count == 10'd799) begin //行扫描到 799 时，列扫描
           if (v_count == 10'd524) begin //524 次 clk 上升沿一个周期
               v_count <= 10'h0;
           end else begin
```

```
                v_count <= v_count + 10'h1;
            end
        end
    end

always @ (posedge vga_clk) begin
        row_addr <=  row[8:0]; // 像素点行地址存储
        col_addr <=  col;      // 像素点列地址存储
        rdn      <= ~read;
        hs       <=  h_sync;   // 水平同步
        vs       <=  v_sync;   // 垂直同步
        r        <=  rdn ? 4'h0 : d_in[3:0]; //红绿蓝共 12 位
        g        <=  rdn ? 4'h0 : d_in[7:4];
        b        <=  rdn ? 4'h0 : d_in[11:8];
    end
```

3.1.1.2 VGA 功能实现

首先通过观察 VGA 实现的小球移动和实例代码，我发现不同位置的像素点显示不同颜色主要是通过以下模块和语句实现的。

```
//计算 x2,y2,r2
assign x_sqr = (x - col_addr) * (x - col_addr);
assign y_sqr = (y - row_addr) * (y - row_addr);
assign r_sqr = radius * radius;
//赋给像素点颜色
always@(*) begin
        if ((x_sqr + y_sqr < r_sqr)) //当 x2+y2<r2 时
            vga_data <= SW[12:1]; //由 12 位开关控制颜色，形成小球
        else vga_data <= 12'hfff; //其他位置的像素点为白色
end
```

示例文件 TOP 中 VGA 模块的连接如下：

```
vgac v0 (
    .vga_clk(clkdiv[1]), .clrn(SW_OK[0]), .d_in(vga_data), .row_add
r(row_addr), .col_addr(col_addr), .r(r), .g(g), .b(b), .hs(hs), .vs
(vs)
    );
```

这里理解 x,y 与 col_addr, row_addr 的关系时我产生了一些困惑，因为 col_addr,row_addr 没有找到确定的值，仅仅与 x,y,r 有关。后来我把它们类比为 x,y 坐标中的坐标，用于确定每一个像素点。

从把表示颜色的 12 位值赋给 vga_data 来看这个输入是控制颜色的，并且可以看出，不同位置的像素点 vga_data 不同。一开始我把它理解为存储所有像素点颜色信息的容器，后来在添加 IP 核的时候，发现我可以通过在不同时机改变对 vga_data 的赋值，从而改变对

d_in 的输入。这时我才理解到 vga_data 并不是一下子存储了所有点的颜色信息，而是在扫描到这个像素点的时候，对 vga_data 赋这个点应该显示的颜色的值。

因此之后我并没有改动示例文件中的 VGA 模块，直接把它应用到我的大程序中，通过设置不同时机，不同位置像素点的不同颜色来实现图像。

3.1.2 俄罗斯方块实现，从像素点控制变成数组控制

3.1.2.1 如何访问每个方块

在我画出 10*10 的网格之后，就是怎么对每一个格子赋值颜色的问题了。一开始我打算设置两个整数，一个表示行数，一个表示列数，共同确定方块处于第几行第几列。记录行，列与颜色的对应关系。但是后来发现这样难以找到方块与方块之间对应的关系，即不能从一个方块的位置的到我想要的方块的位置。因此我设置了数组，即 reg 类型参数。最初设置的是 20*20=400 大小的 reg 参数，但是由于后续程序较为复杂之后跑起来太慢，改为 10*10=100 大小了。这样既不影响总体功能的实现还能提高效率。

在设置了数组之后，我就可以通过下标访问某个特定的方块并对其颜色进行赋值，比如第 i 行第 j 列的方块它对应的数组下标为[10*i+j]，还是较为方便的。

3.1.2.2 如何控制每个方块的颜色

在这里我犯了个很大的错误，我完全把 reg 当成数组来理解了，最初的时候我以为 reg 每一位都可以表示任意整数，并打算对每一位 reg 赋值不同的整数来代表这一位对应的方块显示不同的颜色。后来我发现 n 条 reg 最多只能表示 0-2^n 个整数，也就是说，我如果要表示 100 个格子，每个格子可能的颜色为 4 种，那么我要设置 100*4=400 条 reg。这就意味着我每一个方块对应的下标不再是[10*i+j]了，要更加复杂。设计起来代码过于冗长，且一旦发现错误需要很长时间来调整。

为了不改变下标与方块较为简单的对应关系，我决定多设几个 10*10=100 大小的数组。不同数组代表不同的颜色，另设置两个数组，一个负责记录正在下落的方块组合的位置，另一个负责记录所有已经落下的方块组合的位置。通过这几个数组合并起来显示最终的游戏图像。

3.1.3 IP 核功能实现

3.1.3.1 生成 IP 核

虽然 PPT 上已经给出了 IP 核设置的方法，但是在实际生成过程中还是遇到了一些问题，比如上传的图片格式必须是 coe 格式，图片的大小要与 VGA 屏幕大小相同，即 640*480 像素，另外，在设置大小的窗口不是 PPT 上给出的“12 和 1024”，而应该是“12 和 640*480（VGA 屏幕大小）”。

3.1.3.2 调用 IP 核生成图像

PPT 上并没有给出具体的调用 IP 核的方法，通过“View HDL Instantiation Template”查看调用结构可以看出 IP 核的输入输出，通过询问已经做出 IP 核的同学，我了解到 IP 核

中的四个输入，一个输出中，我需要用到并设置的只有”clka,addra 和 douta”它们的功能如下：

输入/输出	功能
clka	扫描频率
addra	像素点地址输入
douta	像素点颜色输出

图表 10 IP 核输入输出功能对应

在给 addra 赋值的时候我遇到了一些小问题，我以为 addra 仅仅等于 row_addr 和 col_addr 简单相加，并且不确定是 row_addr 在前还是 col_addr 在前。通过询问同学，得知 addra 应该等于“640*row_addr+col_addr”。

另外要注意，在 IP 核中作为输出的 ipcolor 要做为输入在特定的时机输入给 vga。

调用 IP 核相关语句：

```
RAM_B(.clka(clk),.addra(address),.douta(ipcolor));
address=640*row_addr+col_addr;
vga_data=ipcolor;
```

3.1.3 键盘驱动实现，如何读取 PS/2 信号

成功驱动 PS/2 键盘主要是要构造出读取信息的“轮子”,根据了解到的 PS/2 协议内容，检测上升沿与下降沿，当在时间范围内测出一个连续的上升—下降电平变化后，说明有数据传入，此时可以连续读取对应的信号，由信号判断是否为对应的上下左右按键，同时加入一个简易的防抖动板块，“过滤”得到最终的键盘输入。

```
module ps2_scan(
    input wire clk,              // Clock
    input wire rst,              // Reset
    input wire ps2_clk,          // PS2 clock
    input wire ps2_data,         // PS2 data

    output reg [8:0] crt_data    // Input data of the keyboard
    );



    reg [1:0] ps2_clk_state;    // PS2 clock recorder
    wire ps2_clk_neg;            // True at the negedge of the PS2
clock
    assign ps2_clk_neg = ~ps2_clk_state[0] & ps2_clk_state[1];

    // Registers for data reading
    reg [3:0] read_state;
    reg [7:0] read_data;

    // Registers for special signals
    reg is_f0, is_e0;
```

```
// Record the PS2 clock
always @ (posedge clk or negedge rst)
    if (!rst)
        ps2_clk_state <= 2'b0;
    else
        ps2_clk_state <= {ps2_clk_state[0], ps2_clk};

always @ (posedge clk or negedge rst) begin
    if (!rst) begin
        read_state <= 4'b0;
        read_data <= 8'b0;

        is_f0 <= 1'b0;
        is_e0 <= 1'b0;
        crt_data <= 9'b0;
    end
    else if (ps2_clk_neg) begin
        // Reads in the data
        if (read_state > 4'b1001)
            read_state <= 4'b0;
        else begin
            if (read_state > 4'b0 && read_state < 4'b1001)
                read_data[read_state - 1] <= ps2_data;
            read_state <= read_state + 1'b1;
        end
    end
    else if (read_state == 4'b1010 && |read_data) begin
        if (read_data == 8'hf0)
            is_f0 <= 1'b1;
        else if (read_data == 8'he0)
            is_e0 <= 1'b1;
        else
            if (is_f0) begin
                // A key is released
                is_f0 <= 1'b0;
                is_e0 <= 1'b0;
                crt_data <= 9'b0;
            end
            else if (is_e0) begin
                is_e0 <= 1'b0;
                crt_data <= {1'b1, read_data};
            end
            else
```

```
                    crt_data <= {1'b0, read_data};

            read_data <= 8'b0;
        end
    end

endmodule
```

防抖动：

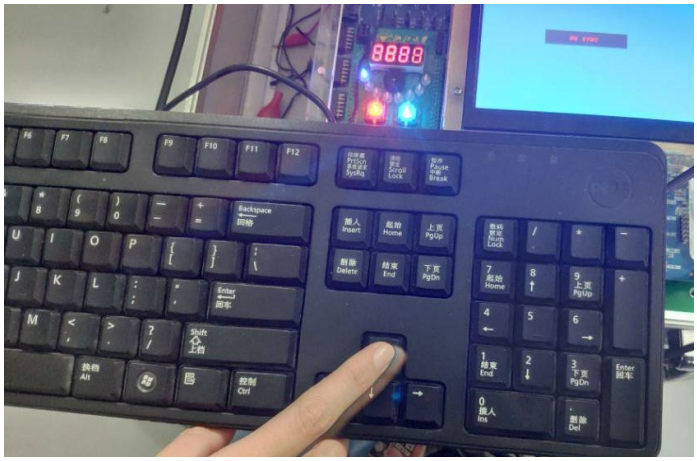
```
ps2_scan
        scanner(
            .clk(clk_fast),
            .rst(rst),
            .ps2_clk(ps2_clk),
            .ps2_data(ps2_data),
            .crt_data(crt_data)
        );

    always @ (posedge clk_slow or negedge rst)
        if (!rst) begin
            key_up_state <= 2'b0;
            key_down_state <= 2'b0;
            key_left_state <= 2'b0;
            key_right_state <= 2'b0;
//            key_ok_state <= 2'b0;
//            key_switch_state <= 2'b0;
          end
                else begin
            // Record the key state
            key_up_state <= {key_up_state[0], crt_data == 9'h175};
            key_down_state  <=  {key_down_state[0],  crt_data  ==
9'h172};
            key_left_state  <=  {key_left_state[0],  crt_data  ==
9'h16b};
            key_right_state  <=  {key_right_state[0],  crt_data  ==
9'h174};
            key_ok_state <= {key_ok_state[0], crt_data == 9'h029};
 //            key_switch_state <= {key_switch_staendte[0], crt_data
== 9'h014};
        end
```

后进行测试，将上、下、左、右四键接到 LED 等上，分别对应前四个灯：

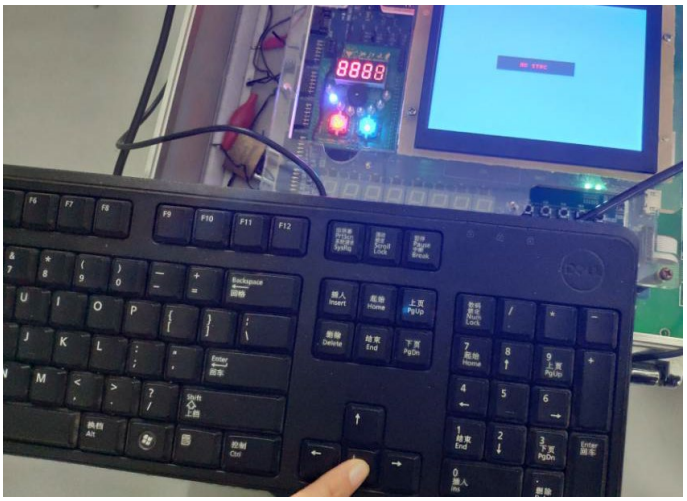
```
always @(*)begin
       if(kup)begin LED[0] <= 1;end
```

```
    else if(kdown)begin LED[1] <= 1;end
    else if(kleft) begin LED[2] <=1;end
    else if(kright)begin LED[3] <= 1;end
    else LED <= 0;
end
```

测试成功如下图：

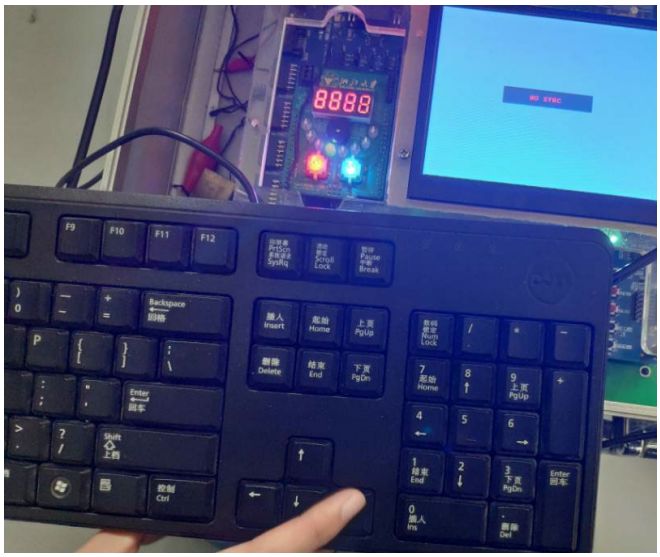


图表 11 键盘上键对应第一个 LED 灯



图表 12 键盘下键对应第二个 LED 灯

图表 12 键盘左键对应第三个 LED 灯



图表 12 键盘右键对应第四个 LED 灯

3.2 模块仿真测试

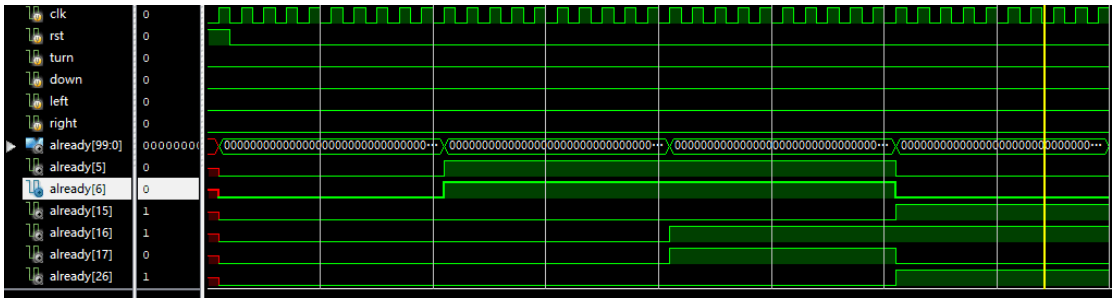
为了验证俄罗斯方块主要逻辑的正确性，我对其中的主要功能也就是方块移动部分进行了仿真激励，其代码如下：

```
module MyVga_test;

	// Inputs
	reg clk;
	reg rst;
	reg turn;
	reg down;
	reg left;
	reg right;
	wire start;
	// Outputs
	wire [99:0] window;
	wire [15:0]shape;
	wire [99:0]move;
	wire scan;
	wire [3:0]con;

	// Instantiate the Unit Under Test (UUT)
	MyVga uut (
		.clk(clk),
		.rst(rst),
		.turn(turn),
		.down(down),
		.left(left),
```

```
            .right(right),
            .window(window),
            .start(start),
            .shape(shape),
            .move(move),
            .scan(scan),
            .con(con)
        );
integer i = 0;
        initial begin
            // Initialize Inputs
            clk = 0;
            rst = 1;
            turn = 0;
            down = 0;
            left = 0;
            right = 0;

            // Wait 100 ns for global reset to finish
            #100;
            rst = 0;
            // Add stimulus here
            for(; i < 20; i = i + 1)
                #100;
            left = 1;
            #100;
            left = 0;
            #100;
            right = 1;
            #100;
            right = 0;
            #100;
            turn = 1;
            #100;
            turn = 0;
            #100;
            down = 1;
            #100;
            down = 0;
            for(i = 0; i < 20; i = i + 1)
                #100;
        end
       always begin
            clk = 0;
```

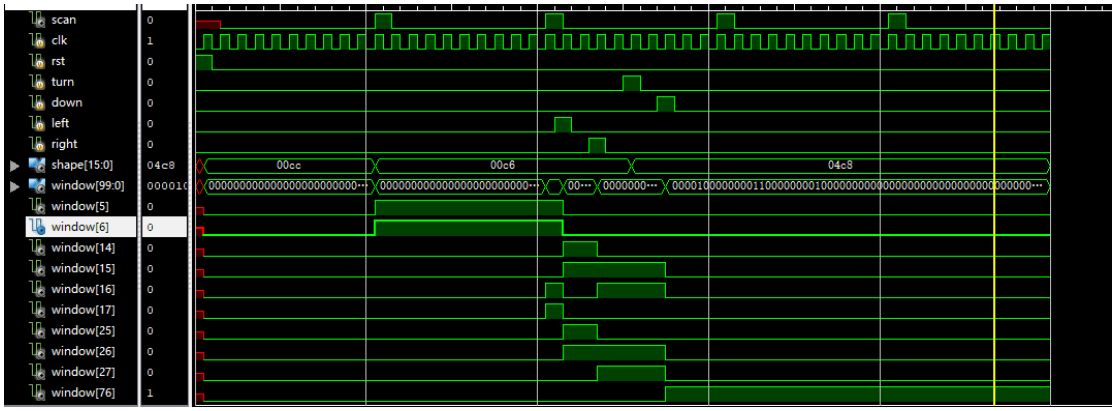
```
            #50;
            clk = 1;
            #50;
        end
endmodule
```

仿真激励结果如下：



图表 13 仿真测试结果 1

这个是在未对逻辑模块进行左移、右移、下降、转向的操作时的仿真，首先重置模块状态，拉起 rst（重置），输出窗口 window 清零。因为设置的是每 1s 扫描一次，所以要十个时钟信号后才会有一次扫描。第一次扫描调用生成形状的模块，window 中出现方块的一行，这里可以看见 window[5]、window[6]变为 1（其他部分均为 0，受空间所限这里没有列出），再过 10 个 clk 上沿信号，第二次扫描，方块应该下降一行，此时可以看到 window[15]、window[16]也变成了 1，说明方块下降一行，完整图像已经在 window 中体现，同理，第三次扫描时仍然下移一行。

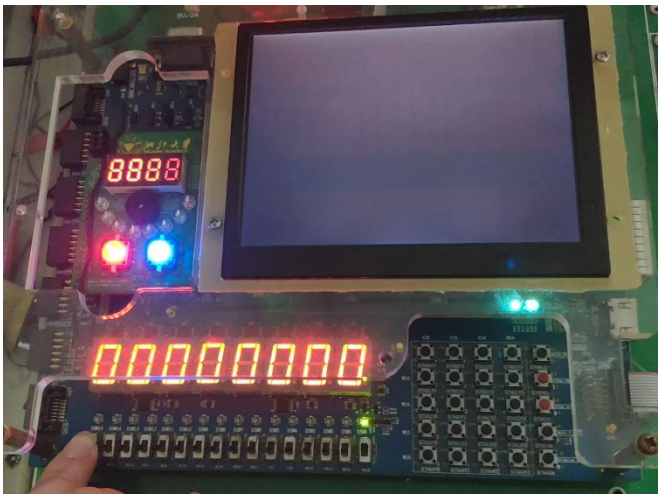


图表 14 仿真测试结果 2

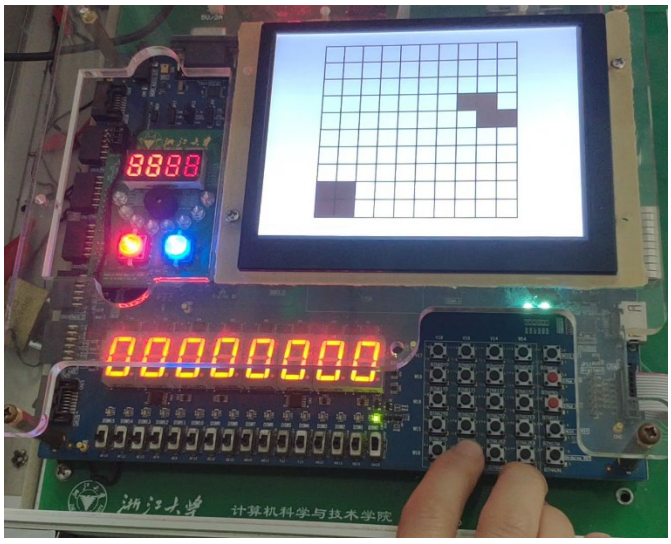
这个是测试该模块对于左移、右移、下降、转向的操作的反应，为了便于观察，我将 scan 信号（扫描）和 shape 信号（方块形状）也放在了图像上，重置后在第十个 clk 信号处 scan 信号输出高电平，说明开始扫描，在下一个 scan 之后，left 被拉高，图像左移，可以看见数据从 window[17:16]移至 window[16:15]，之后 right 拉高又回到 window[17:16]，接下来 turn 被拉高，因为此时没有 scan，所以只是形状发生了改变，可以看见 shape 从 16'h00c6 变为 16'h04c8，之后拉高 down，发现 window 数值下沉，信号传递至 window[76]、window[87:86]和 window[97]即形状触底。以上说明模块功能已成功实现。

3.3 完整课题测试

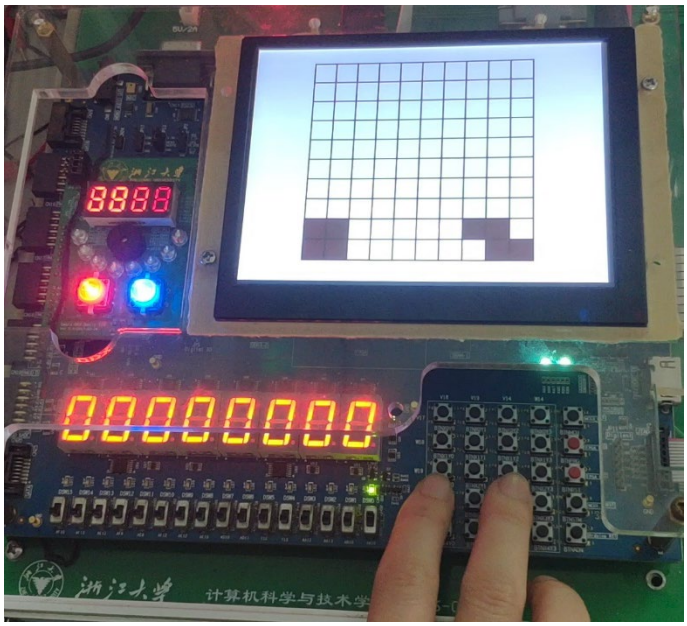
游戏开始，打开 SW[0]，VGA 信号输出，显示 IP 核中的图像（这里因为图片的分辨率不高导致图片较暗难以辨别），记分板上的分数重置为 0，上拨 SW[15]后再下拉即可开始游戏。游戏过程中可操作阵列键盘对方块进行对应操作，当一行满格了之后会消除该行，蜂鸣器鸣叫，记分板更新，最终效果展示如下：



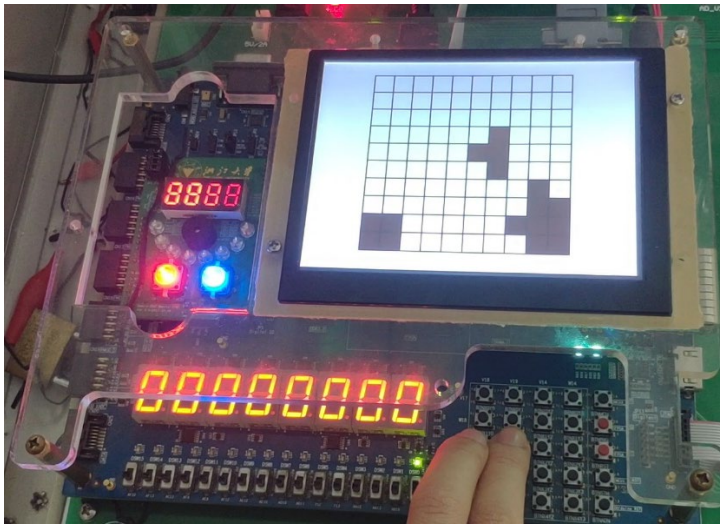
图表 15 开始界面



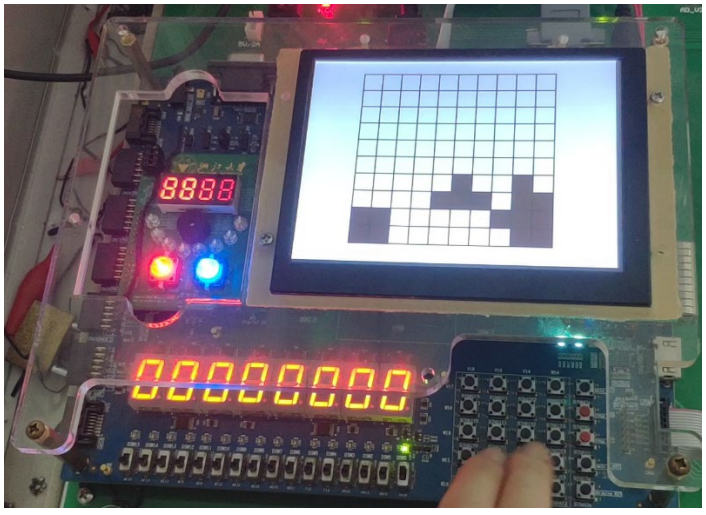
图表 16 按下快速下落键之前



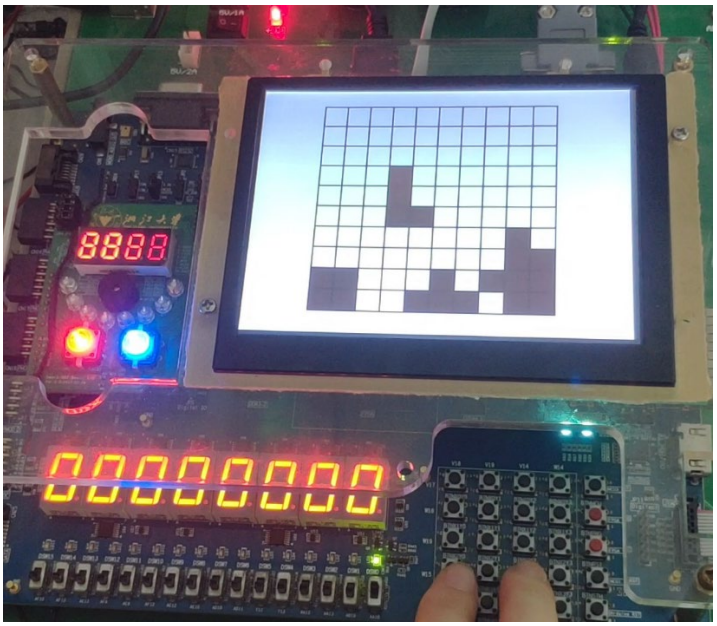
图表 17 按下快速下落键之后



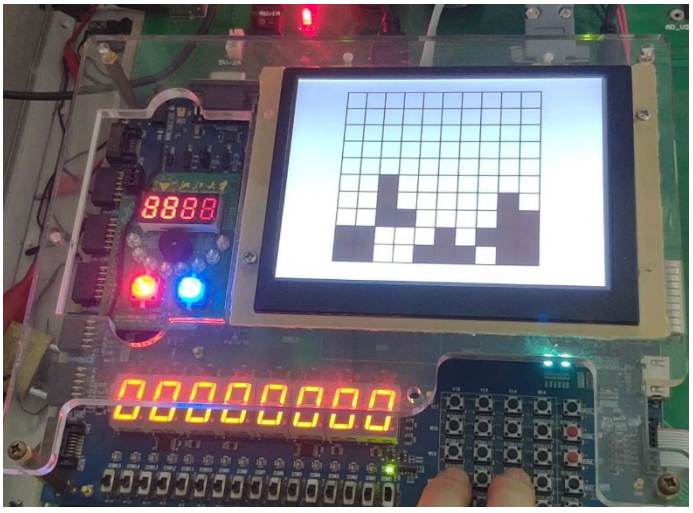
图表 18 改变方向前



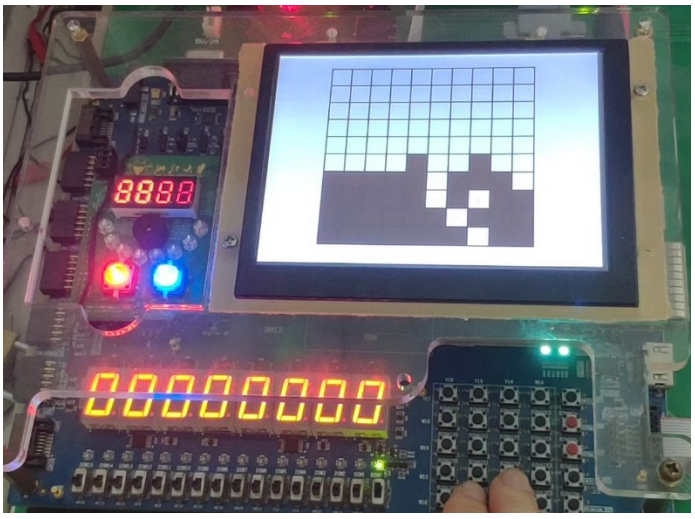
图表 19 按下改变方向后



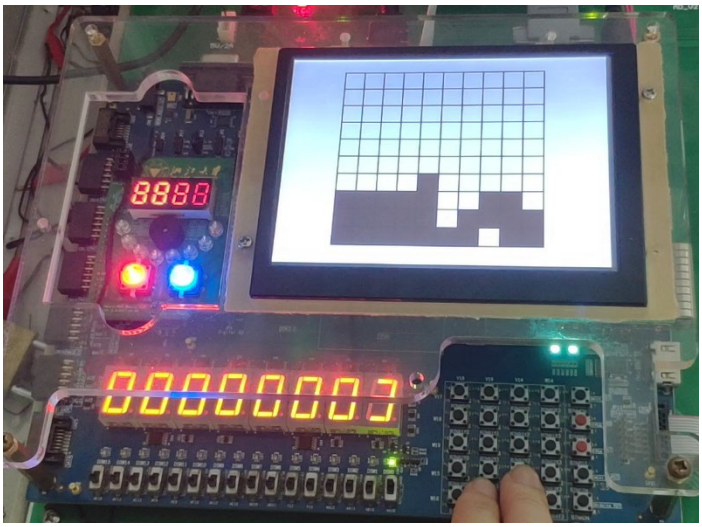
图表 20 按下左移键前



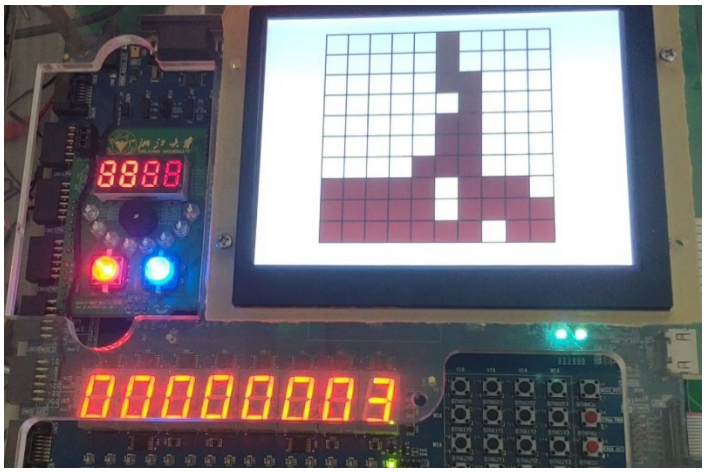
图表 21 按下左移键之后



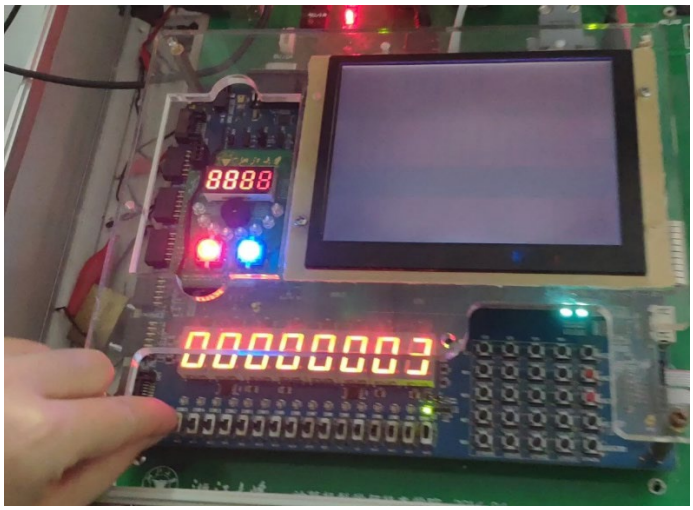
图表 22 方块落下前（即将满行）



图表 23 方块落下后，满行消除，且记分板记 3 分



图表 24 当方块触到顶端游戏结束



图表 25 游戏结束界面

游戏失败，返回初始界面，此时记分板未清零，手动拨起 SW[15]可重置，再次放下后可进行新一轮的游戏

四、总结

本次课题研究可以说是真正的自己写程序了，以往的硬件实验课都是一边看老师给出的代码一边“抄“，遇到最大的问题也就是抄错了导致实验失败。而本次课题研究连相关协议的资料都要自行查阅，更遑论遇到问题难以求助，只能自己一点一点摸索。而且因为整个程序涉及的数据较大，仿真测试难以直观体现结果，很多特殊情况也不是一眼能看出的，导致物理验证时会出现各种意料之外的情况，却无法准确了解原因，只能一步一步排查，而庞大的程序也注定这个排查会耗费很长的时间。

但是所幸时间没有白费，除了最后完整的课题成果，我对于本课程中的一些概念以及要点有了更深刻更全面的认识，尤其是实验方面，显著提升了我对 Verilog 语言的掌握程度，同时还增强了我关于学好硬件课的自信心。

当然本次课题还有一些遗憾，比如选图失败导致 IP 核上的图案非常模糊，但是因为时间限制不能找到合适的图像，所以只能将就使用那张黑色“幕布“，这样想倒也不失为一种神秘感了。还有尝试很多次最后以失败告终的 PS/2 键盘，虽然键盘没能成功加载上去，但是在测试中证明对于键盘的驱动文件是没有问题的。