浙江大学

课程设计报告

题目 ：基于数字系统的打地鼠游戏

姓名:

学号:

指导老师: 洪奇军

报告完成日期 2020 年 1 月 8 日

基于数字系统的打地鼠游戏

摘要：本次课程设计以打地鼠游戏为主，主要完成了基于数字系统的打地鼠小游戏的实现，报告将依次对游戏设计的呃背景，功能，实现方式，成果展示，以及设计时的一些思考还有感悟进行介绍，还包含对题目完成过程中遇到的困难还有解决困难的历程，以及从中获得的收获。

关键词：数字逻辑，打地鼠，Verilog

一．绪论

1.1 游戏背景介绍

打地鼠这款著名的游戏是 1976 年由 Creative Engineering 公司的 Aaron Fechter 发明的。游戏主要内容为不断打击从地鼠洞中探出头来的地鼠以获得分数，游戏比较简单，但却很容易让人快乐，也有许多新的玩法。

1.2 课题与接口功能

1.2.1 阵列键输入

设计中用到了左下方 12 个按键，分别对应对不同的地鼠洞进行打击。

1.2.2 开关输入

设计中用到了 16 个开关。

二．课题设计

2.1 理论基础

2.1.1 VGA 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。

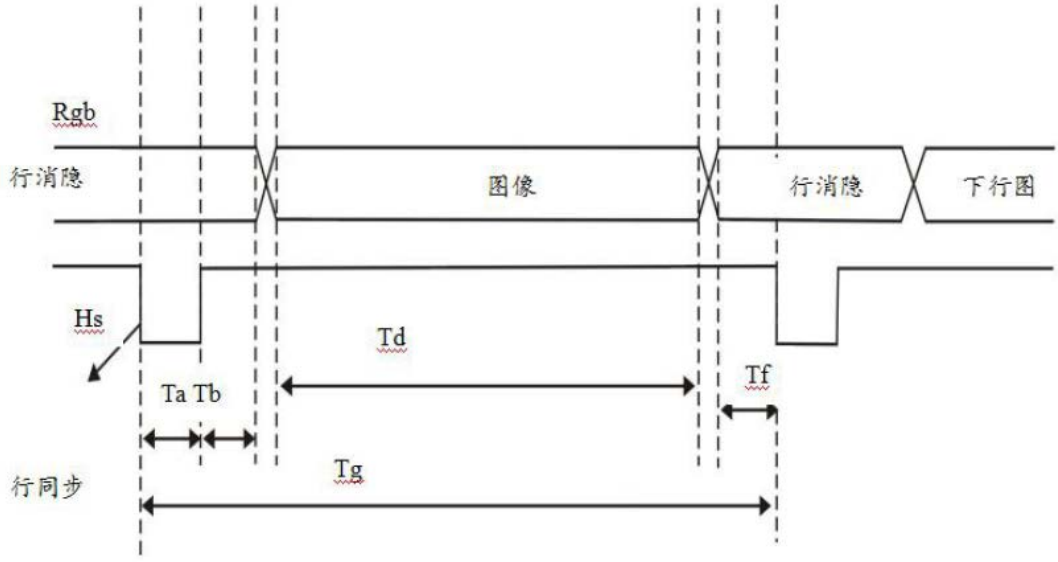
常见的彩色显示器一般都是由 CRT(阴极射线管)构成，其引出线共含 5 个信号：R、G、B(三基色信号)、HS (行同步信号)、VS (列同步信号)。每一个像素的色彩由 R(红，Red)，G(绿，Green)，B(蓝，Blue) 三基色构成，在实验的验证阶段可以仅利用 R、G、B 三种基色的一元化值(0 和 1)的不同组合来验证设计的正确性。

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，

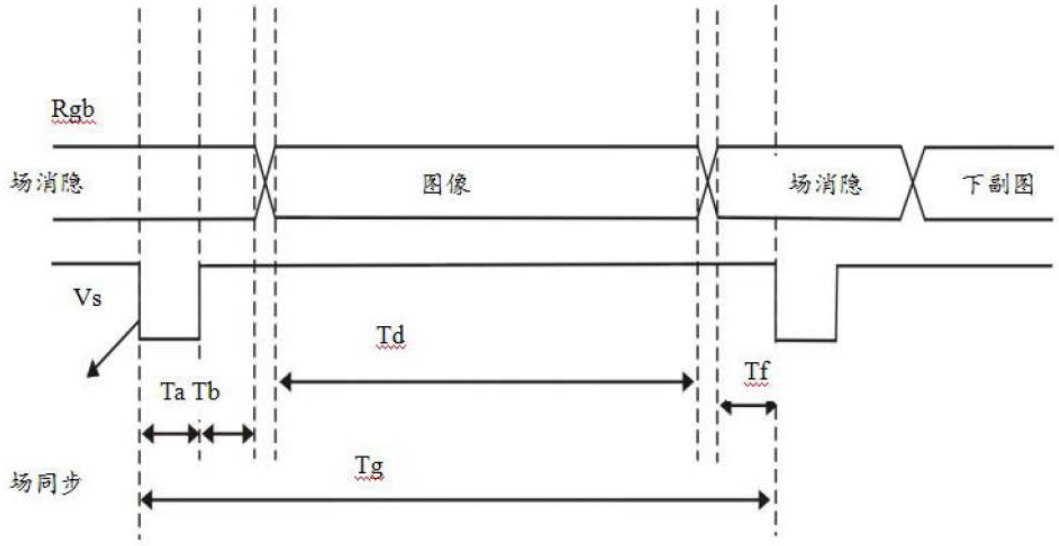
当到达屏幕的右边缘时，电子束关闭(水平消隐),并快速返回屏幕左边缘(水平回扫),然后在下一条扫描线上开始新的一次水平扫描。一旦所有的水平扫描均告完成，电子束在屏幕的右下角结束并关闭(垂直消隐)，然后迅速返回到屏幕的左上角(垂直回扫),开始下一次光栅扫描。

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。

VGA 行扫描时序图：



VGA 场扫描时序图：



2.1.2 有限状态机

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： Mealy 型和 Moore 型。Moore 型状态机的输出只是当前状态的函数。Mealy 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，Moore 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 Mealy 型状态机要多等待一个时钟周期。

2.1.3 硬件描述语言

Verilog HDL 是一种硬件描述语言（HDL: Hardware Description Language），以文本形式来描述数字系统硬件的结构和行为的语言，用它可以表示逻辑电路图、逻辑表达式，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。

2.1.4 PAL 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 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。

2.2.2 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。

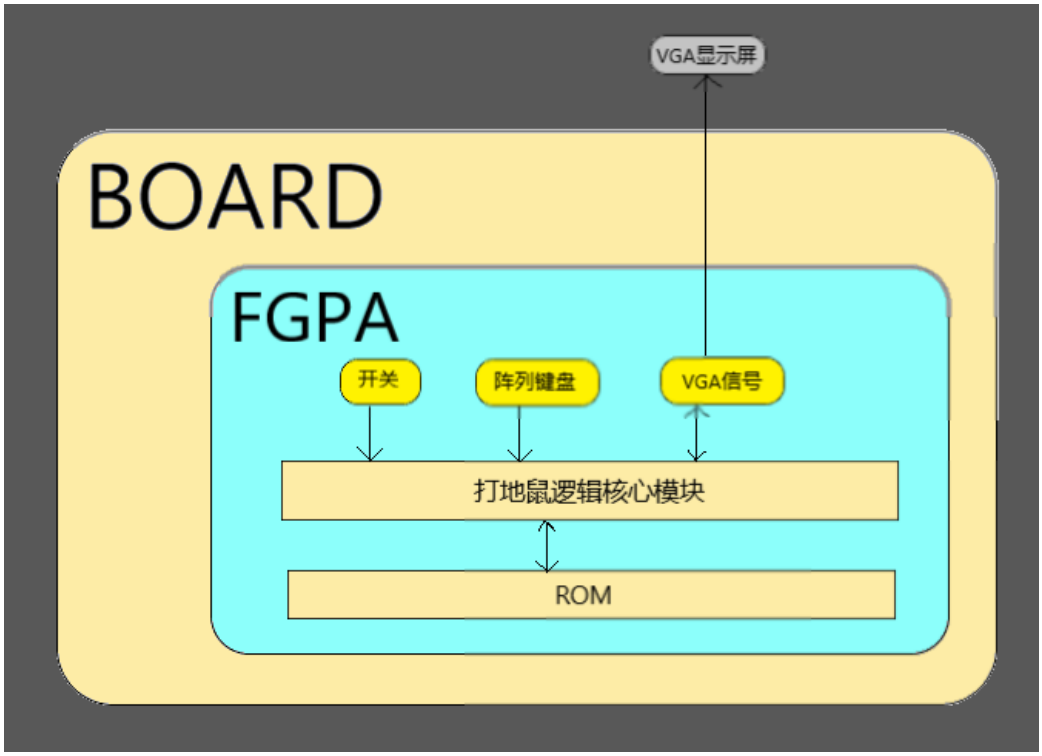
2.2.3 Photoshop 软件

Photoshop 主要处理以像素所构成的数字图像。使用其众多的编修与绘图工具，可以更有效的进行图片编辑工作。独特的历史纪录浮动视窗和可编辑的图层效果功能使用户可以方便的测试效果。对各种滤镜的支持更令使用户能够轻松创造出各种奇幻的效果。

2.3 设计方案

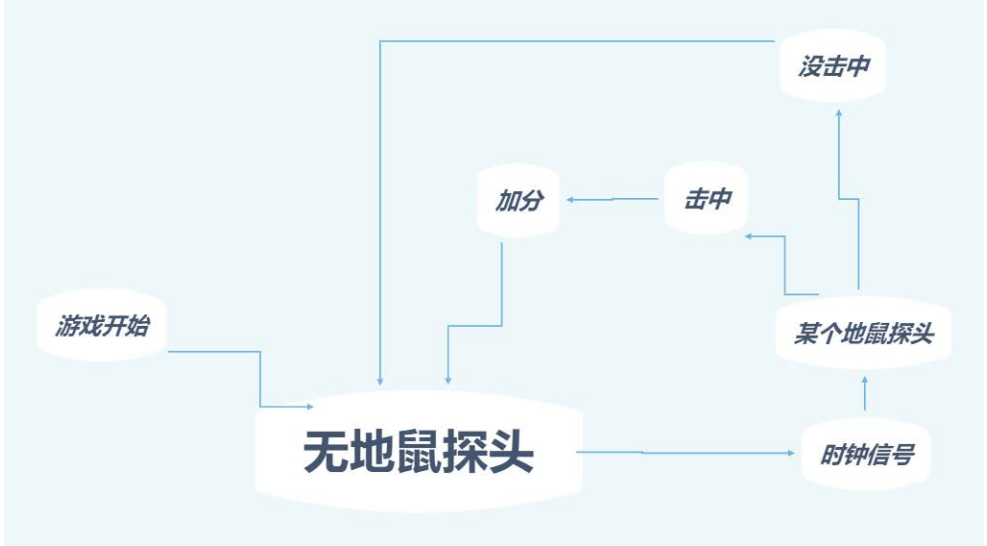
2.3.1 整体设计方案

整体硬件结构图：



开关用于控制整体电路的状态，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；打地鼠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 打地鼠逻辑核心模块设计



便于理解还有整理，整个代码被分成了三个部分：游戏逻辑部分，外设驱动部分，整合部分。在游戏逻辑部分主要有地鼠的生成，游戏计分，在外设驱动部分主要有 VGA 显示游戏主界面，七段数码管显示分数，还有阵列键盘输入打击信息，在整合部分主要将游戏逻辑部分和驱动部分整合起来。

游戏主体逻辑部分部分代码：

```
module game(clk, rst, start, hits, score, gopherNum);
    input wire clk;
    input wire rst;
    input wire start;
    input wire[11:0] hits;
    output reg[31:0] score;
    output wire[3:0] gopherNum;

    wire[31:0] tempScore;
    wire[3:0] tempGophers;

    clk1s clk1(.clk(clk), .clk_1s(clk_1s));
    clk100ms clk2(.clk(clk), .clk_100ms(clk_100ms));

    wire[11:0] gopher;
    gopherCreater gphc(.clk_1s(clk_1s), .gopher_num(gopherNum));

    initial score<=32'b0;
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            score<=32'b0;
        end else if(start) begin
            score<=tempScore;
        end else begin
            score<=32'b0;
        end
    end

    scoreCnt
sc(.clk(clk), .gopherNum(gopherNum), .start(start), .hits(hits), .sco
re_BCD(tempScore), .newGophers(tempGophers));
endmodule
```

地鼠生成板块主要代码：

```
module gopherCreater(clk_1s, gopher_num);
    input clk_1s;
    output reg[3:0] gopher_num;

    reg[3:0] num;
    always @(posedge clk_1s) begin
        num<={$random}%12;
    end
endmodule
```

计分板块主要代码：

```
module scoreCnt(clk, gopherNum, start, hits, score_BCD, newGophers);
    input clk;
    input wire[3:0] gopherNum;
    input start;
```

```
input wire[11:0] hits;
output [31:0] score_BCD;
output reg[3:0] newGophers;

reg[26:0] score=0;
binToBCDs btb(.clk(clk), .bins(score), .decs(score_BCD));

initial newGophers<=gopherNum;
assign hit=|hits;
always @(posedge hit) begin
    newGophers<=gopherNum;
    if(hits[0]) begin
        if(gopherNum==0) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[1]) begin
        if(gopherNum==1) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[2]) begin
        if(gopherNum==2) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[3]) begin
        if(gopherNum==3) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[4]) begin
        if(gopherNum==4) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[5]) begin
        if(gopherNum==5) begin
            score<=score+1'b1;
            newGophers<=4'd0;
        end else begin
            score<=score-1'b1;
        end
    end
    if(hits[6]) begin
        if(gopherNum==6) begin
```

```
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(hits[7]) begin
            if(gopherNum==7) begin
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(hits[8]) begin
            if(gopherNum==8) begin
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(hits[9]) begin
            if(gopherNum==9) begin
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(hits[10]) begin
            if(gopherNum==10) begin
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(hits[11]) begin
            if(gopherNum==11) begin
                score<=score+1'b1;
                newGophers<=4'd0;
            end else begin
                score<=score-1'b1;
            end
        end
        if(!start) begin
            score<=27'b0;
        end
        if(score>9999) begin
            score<=27'b0;
        end
    end
endmodule
```

在外设驱动部分主要分为 VGA 调用，阵列键输入，还有开关输入。

VGA 驱动部分代码：

```
module vgac (vga_clk,clrn,d_in,row_addr,col_addr,rdn,r,g,b,hs,vs); //
vgac
   input     [11:0] d_in;     // bbbb_gggg_rrrr, pixel
   input            vga_clk;  // 25MHz
   input            clrn;
   output reg [8:0] row_addr; // pixel ram row address, 480 (512)
lines
   output reg [9:0] col_addr; // pixel ram col address, 640 (1024)
pixels
   output reg [3:0] r,g,b;    // red, green, blue colors
   output reg       rdn;      // read pixel RAM (active_low)
   output reg       hs,vs;    // horizontal and vertical
synchronization
   // h_count: VGA horizontal counter (0-799)
   reg [9:0] h_count; // VGA horizontal counter (0-799): pixels
   always @ (posedge vga_clk) begin
       if (!clrn) begin
           h_count <= 10'h0;
       end else if (h_count == 10'd799) begin
           h_count <= 10'h0;
       end else begin
           h_count <= h_count + 10'h1;
       end
   end
   // v_count: VGA vertical counter (0-524)
   reg [9:0] v_count; // VGA vertical   counter (0-524): lines
   always @ (posedge vga_clk or negedge clrn) begin
       if (!clrn) begin
           v_count <= 10'h0;
       end else if (h_count == 10'd799) begin
           if (v_count == 10'd524) begin
               v_count <= 10'h0;
           end else begin
               v_count <= v_count + 10'h1;
           end
       end
   end
    // signals, will be latched for outputs
    wire  [9:0] row    =  v_count - 10'd35;     // pixel ram row addr
    wire  [9:0] 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
    // vga signals
    always @ (posedge vga_clk) begin
        row_addr <=  row[8:0]; // pixel ram row address
        col_addr <=  col;      // pixel ram col address
        rdn      <= ~read;     // read pixel (active low)
        hs       <=  h_sync;   // horizontal synchronization
        vs       <=  v_sync;   // vertical   synchronization
        r        <=  rdn ? 4'h0 : d_in[3:0]; // 3-bit red
        g        <=  rdn ? 4'h0 : d_in[7:4]; // 3-bit green
        b        <=  rdn ? 4'h0 : d_in[11:8]; // 2-bit blue
    end
endmodule
```

七段数码管显示分数部分：

```
module Seg7Device(clkIO, clkScan, clkBlink, data, point, LES, sout,
segment, anode);
    input clkIO;
    input [1:0] clkScan;
    input clkBlink;
    input [31:0] data;
    input [7:0] point;
    input [7:0] LES;
    output [3:0] sout;
    output reg [7:0] segment;
    output reg [3:0] anode;
    wire [63:0] dispData;
    wire [31:0] dispPattern;

    Seg7Decode U0(.hex(data), .point(point), .LE(LES &
{8{clkBlink}}), .pattern(dispData));

    Seg7Remap U1(.I(data), .O(dispPattern));

    ShiftReg #(.WIDTH(64))
U2(.clk(clkIO), .pdata(dispData), .sout(sout));

    always @*
        case(clkScan)
        2'b00: begin segment <= ~dispPattern[ 7: 0]; anode <=
4'b1110; end
        2'b01: begin segment <= ~dispPattern[15: 8]; anode <=
4'b1101; end
        2'b10: begin segment <= ~dispPattern[23:16]; anode <=
4'b1011; end
        2'b11: begin segment <= ~dispPattern[31:24]; anode <=
4'b0111; end
        endcase
endmodule
```

32 位数据译码部分代码：

```
module Seg7Decode(hex, point, LE, pattern);
    input [31:0] hex;
    input [7:0] point;
    input [7:0] LE;
    output [63:0] pattern;
    wire [63:0] digits;
    SegmentDecoder
        U0(.hex(hex[ 3: 0]), .segment(digits[ 6: 0])),
        U1(.hex(hex[ 7: 4]), .segment(digits[14: 8])),
        U2(.hex(hex[11: 8]), .segment(digits[22:16])),
        U3(.hex(hex[15:12]), .segment(digits[30:24])),
        U4(.hex(hex[19:16]), .segment(digits[38:32])),
        U5(.hex(hex[23:20]), .segment(digits[46:40])),
        U6(.hex(hex[27:24]), .segment(digits[54:48])),
        U7(.hex(hex[31:28]), .segment(digits[62:56]));
    assign {digits[63], digits[55], digits[47], digits[39],
digits[31], digits[23], digits[15], digits[7]} = ~point;

    assign pattern[ 7: 0] = digits[ 7: 0] | {8{LE[0]}};
    assign pattern[15: 8] = digits[15: 8] | {8{LE[1]}};
```

```
    assign pattern[23:16] = digits[23:16] | {8{LE[2]}};
    assign pattern[31:24] = digits[31:24] | {8{LE[3]}};
    assign pattern[39:32] = digits[39:32] | {8{LE[4]}};
    assign pattern[47:40] = digits[47:40] | {8{LE[5]}};
    assign pattern[55:48] = digits[55:48] | {8{LE[6]}};
    assign pattern[63:56] = digits[63:56] | {8{LE[7]}};
endmodule
```

七段数码管译码部分：

```
module SegmentDecoder(hex, segment);
    input [3:0] hex;
    output reg [6:0] segment;
    always @* begin
        case(hex)
            4'h0: segment[6:0] <= 7'b1000000;
            4'h1: segment[6:0] <= 7'b1111001;
            4'h2: segment[6:0] <= 7'b0100100;
            4'h3: segment[6:0] <= 7'b0110000;
            4'h4: segment[6:0] <= 7'b0011001;
            4'h5: segment[6:0] <= 7'b0010010;
            4'h6: segment[6:0] <= 7'b0000010;
            4'h7: segment[6:0] <= 7'b1111000;
            4'h8: segment[6:0] <= 7'b0000000;
            4'h9: segment[6:0] <= 7'b0010000;
            4'hA: segment[6:0] <= 7'b0001000;
            4'hB: segment[6:0] <= 7'b0000011;
            4'hC: segment[6:0] <= 7'b1000110;
            4'hD: segment[6:0] <= 7'b0100001;
            4'hE: segment[6:0] <= 7'b0000110;
            4'hF: segment[6:0] <= 7'b0001110;
        endcase
    end
endmodule
```

Remap 部分：

```
module Seg7Remap(I, O);
    input [31:0] I;
    output [31:0] O;
    assign O[ 7: 0] = {I[24], I[12], I[5], I[17], I[25], I[16], I[4],
I[0]};
    assign O[15: 8] = {I[26], I[13], I[7], I[19], I[27], I[18], I[6],
I[1]};
    assign O[23:16] = {I[28], I[14], I[9], I[21], I[29], I[20], I[8],
I[2]};
    assign O[31:24] = {I[30], I[15], I[11],I[23], I[31], I[22],
I[10],I[3]};
endmodule
```

移位寄存器部分：

```
module ShiftReg(clk, pdata, sout);
    parameter WIDTH = 16;
    parameter DELAY = 12;
```

```
    input clk;
    input [WIDTH - 1:0] pdata;
    output [3:0] sout;
    assign sout = {sck, sdat, oe, clrn};

    wire sck, sdat, clrn;
    reg oe;

    reg [WIDTH:0] shift;
    reg [DELAY-1:0] counter = -1;
    wire sckEn;

    assign sckEn = |shift[WIDTH - 1:0];
    assign sck = ~clk & sckEn;
    assign sdat = shift[WIDTH];
    assign clrn = 1'b1;

    always @ (posedge clk) begin
        if(sckEn) shift <= {shift[WIDTH - 1:0], 1'b0};
        else begin
            if(&counter) begin
                shift <= {pdata, 1'b1};
                oe <= 1'b0;
            end else oe <= 1'b1;
            counter <= counter + 1'b1;
        end
    end
endmodule
```

在设计过程中，发现阵列键的接口并不是一个键单独配备一个接口，而是由横，纵两个坐标一起确定一个阵列键接口，需要将两个坐标的信号进行处理，转换成简明的单键信号，这里采用了 demo 中的 KeyPad 模块，相关代码如下：

```
module Keypad(
    input clk, inout [3:0] keyX, inout [4:0] keyY,
    output reg [4:0] keyCode, output ready, output [8:0] dbg_keyLine
);

    reg state = 1'b0;
    reg [3:0] keyLineX;
    reg [4:0] keyLineY;
    assign keyX = state? 4'h0: 4'bzzzz;
    assign keyY = state? 5'bzzzzz: 5'h0;

    always @ (posedge clk) begin
        if(state)
            keyLineY <= keyY;
        else
            keyLineX <= keyX;
        state <= ~state;
    end

    assign dbg_keyLine = ~{keyLineY, keyLineX};

    wire ready_raw1 = (keyLineX == 4'b1110) | (keyLineX == 4'b1101) |
(keyLineX == 4'b1011) | (keyLineX == 4'b0111);
```

```
    wire ready_raw2 = (keyLineY == 5'b11110) | (keyLineY == 5'b11101)
| (keyLineY == 5'b11011) | (keyLineY == 5'b10111) | (keyLineY ==
5'b01111);
    wire ready_raw = ready_raw1 & ready_raw2;

    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

    AntiJitter #(4) rdyFilter(.clk(clk), .I(ready_raw), .O(ready));

Endmodule
```

整合部分（top）主要将游戏主体逻辑部分和外设接口部分连接起来，将游戏所需要的输入输出连接到外部设备，具体代码如下：

```
module top(
    input clk,
    input rstn,
    input [15:0]SW,
    inout [4:0]BTN_X,
    inout [3:0]BTN_Y,
    output buzzer,
    output SEGLED_CLK,
    output SEGLED_CLR,
    output SEGLED_DO,
    output SEGLED_PEN,
    output hs,
    output vs,
    output [3:0] r,
    output [3:0] g,
    output [3:0] b,
    output LED_start,
    output LED_rst
    );

    wire rst;
    assign rst=~rstn;
    assign LED_start=SW[15];
    assign LED_rst=rst;
    assign buzzer = 1'b1;

    wire [31:0] clkdiv;
    wire clk_1ms;
    clkdiv clk1(.clk(clk), .rst(1'b0), .clkdiv(clkdiv));
```

```
    clk1ms clk2(.clk(clk), .clk_1ms(clk_1ms));

    wire [15:0] SW_OK;
    AntiJitter #(4) a0[15:0](.clk(clkdiv[15]), .I(SW), .O(SW_OK));

    wire [4:0] keyCode;
    wire keyReady;
    Keypad k0
(.clk(clkdiv[15]), .keyX(BTN_Y), .keyY(BTN_X), .keyCode(keyCode), .re
ady(keyReady));

    wire [11:0] hits;
    assign hits[0] = keyReady&(keyCode==5'd0) ;
    assign hits[1] = keyReady&(keyCode==5'd1) ;
    assign hits[2] = keyReady&(keyCode==5'd2) ;
    assign hits[3] = keyReady&(keyCode==5'd3) ;
    assign hits[4] = keyReady&(keyCode==5'd4) ;
    assign hits[5] = keyReady&(keyCode==5'd5) ;
    assign hits[6] = keyReady&(keyCode==5'd6) ;
    assign hits[7] = keyReady&(keyCode==5'd7) ;
    assign hits[8] = keyReady&(keyCode==5'd8) ;
    assign hits[9] = keyReady&(keyCode==5'd9) ;
    assign hits[10] = keyReady&(keyCode==5'd10) ;
    assign hits[11] = keyReady&(keyCode==5'd11) ;

    wire[3:0] gopherNum;
    game
ga(.clk(clk), .rst(rst), .start(SW_OK[15]), .hits(hits), .score(score
), .gopherNum(gopherNum));

    wire[3:0] sout;
    Seg7Device
segDevice(.clkIO(clkdiv[3]), .clkScan(clkdiv[15:14]), .clkBlink(clkdi
v[25]),

    .data(score), .point(8'h0), .LES(8'h0), .sout(sout), .segment(),
.anode());
    assign SEGLED_CLK = sout[3];
    assign SEGLED_DO  = sout[2];
    assign SEGLED_PEN = sout[1];
    assign SEGLED_CLR = sout[0];

    wire [11:0] vga_data;
    wire [9:0] col_addr;
    wire [8:0] row_addr;

    draw
d(.clk(clkdiv[1]), .gopherNum(gopherNum), .hc(row_addr), .vc(col_addr
), .vga_data(vga_data));

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

endmodule
```

三．实验过程

3.1 遇到的困难及解决

3.1.1 随机数的生成

在实现设计的过程中遇到的第一个困难就是如何随机地让“地鼠”探出头来，关键在于随机，问题可以转化为如何生成一个在一定范围内的随机数，尽查阅资料后发现，可以使用 Verilog 中的内置函数 random 实现，具体代码如下：

```
module gopherCreater(clk_1s, gopher_num);
    input clk_1s;
    output reg[3:0] gopher_num;

    reg[3:0] num;
    always @(posedge clk_1s) begin
        num<={$srandom}%12;
    end
endmodule
```

在 *IEEE Standard for Verilog* 中，对 random 函数的描述如下：

> *The system function $random provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative.*

即 random 函数产生一个 32bit 的随机有符号整数。

3.1.2 分数的计算机制

在设计的时候，根据需求应该实现每当有按键按下时，如果成功击中地鼠，则分数增加，否则分数减少，同时需要将现有的分数储存，于是想到了用按键的边沿触发，还有寄存器的相关知识。相关代码如下：

```
always @(posedge hits) begin
    if(hits[i]) begin
        if(gopher[i]) begin
            score<=score+1'b0;
            gophers[i]<=1'b0;
        end else
            score<=score-1'b0;
        end
    end
end
```

其中$i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11$

3.1.3 将分数显示在 7 段数码管上

分数为 10 进制，要显示在七段数码管，需用每四位二进制数表示一位十进制数，由此想到了 BCD 码的相关知识，需要将二进制码转换为 BCD 码，相关代码如下：

```
module binToBCDs(clk, bins, decs);
    input clk;
    input [26:0] bins;
    output reg[31:0] decs;

    initial begin
        decs<=32'b0;
    end
    reg [4:0] count=27;
    reg [31:0] shiftLeft;
    always @(posedge clk) begin
        count<=count-1'b1;
    end
    always @(posedge clk) begin
        if(count>=0 && count<=26) begin
            if(shiftLeft[31:28]>4) begin
                shiftLeft[31:28]=shiftLeft[31:28]+4'b0011;
            end
            if(shiftLeft[27:24]>4) begin
                shiftLeft[27:24]=shiftLeft[27:24]+4'b0011;
            end
            if(shiftLeft[23:20]>4) begin
                shiftLeft[23:20]=shiftLeft[23:20]+4'b0011;
            end
            if(shiftLeft[19:16]>4) begin
                shiftLeft[19:16]=shiftLeft[19:16]+4'b0011;
            end
            if(shiftLeft[15:12]>4) begin
                shiftLeft[15:12]=shiftLeft[15:12]+4'b0011;
            end
            if(shiftLeft[11:8]>4) begin
                shiftLeft[11:8]=shiftLeft[11:8]+4'b0011;
            end
            if(shiftLeft[7:4]>4) begin
                shiftLeft[7:4]=shiftLeft[7:4]+4'b0011;
            end
            if(shiftLeft[3:0]>4) begin
                shiftLeft[3:0]=shiftLeft[3:0]+4'b0011;
            end
            shiftLeft=shiftLeft<<1;
            shiftLeft[0]=bins[count];
        end else if(count==27)begin
            shiftLeft=32'b0;
        end
    end
    assign finish = &count;
    always @(posedge finish) begin
        decs<=shiftLeft;
    end

    endmodule
```

然后需要将八位十进制数显示在七段数码管上，这里调用了 Seg7Device 模块，相关代码如下：

```
    wire[3:0] sout;
    Seg7Device
segDevice(.clkIO(clkdiv[3]), .clkScan(clkdiv[15:14]), .clkBlink(clkdi
v[25]),
```

```
    .data(score), .point(8'h0), .LES(8'h0), .sout(sout), .segment(),
.anode());
    assign SEGLED_CLK = sout[3];
    assign SEGLED_DO  = sout[2];
    assign SEGLED_PEN = sout[1];
    assign SEGLED_CLR = sout[0];
```

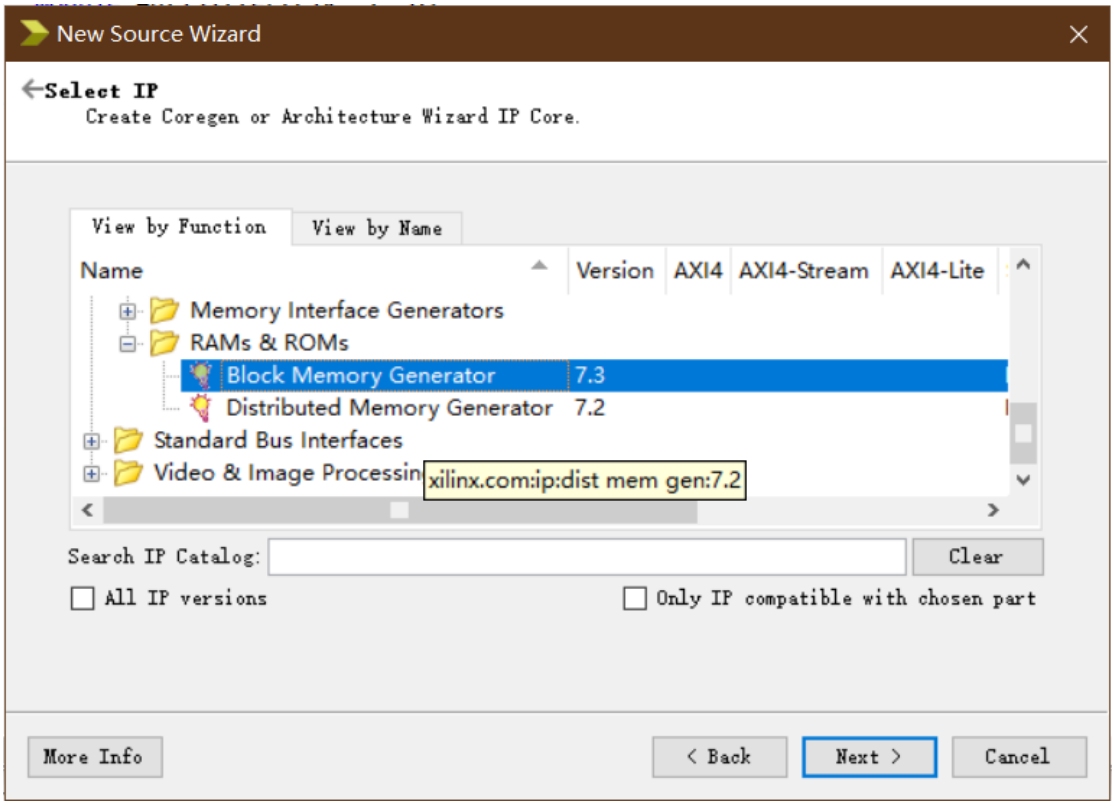
3.1.4 将游戏信息显示在 VGA 上

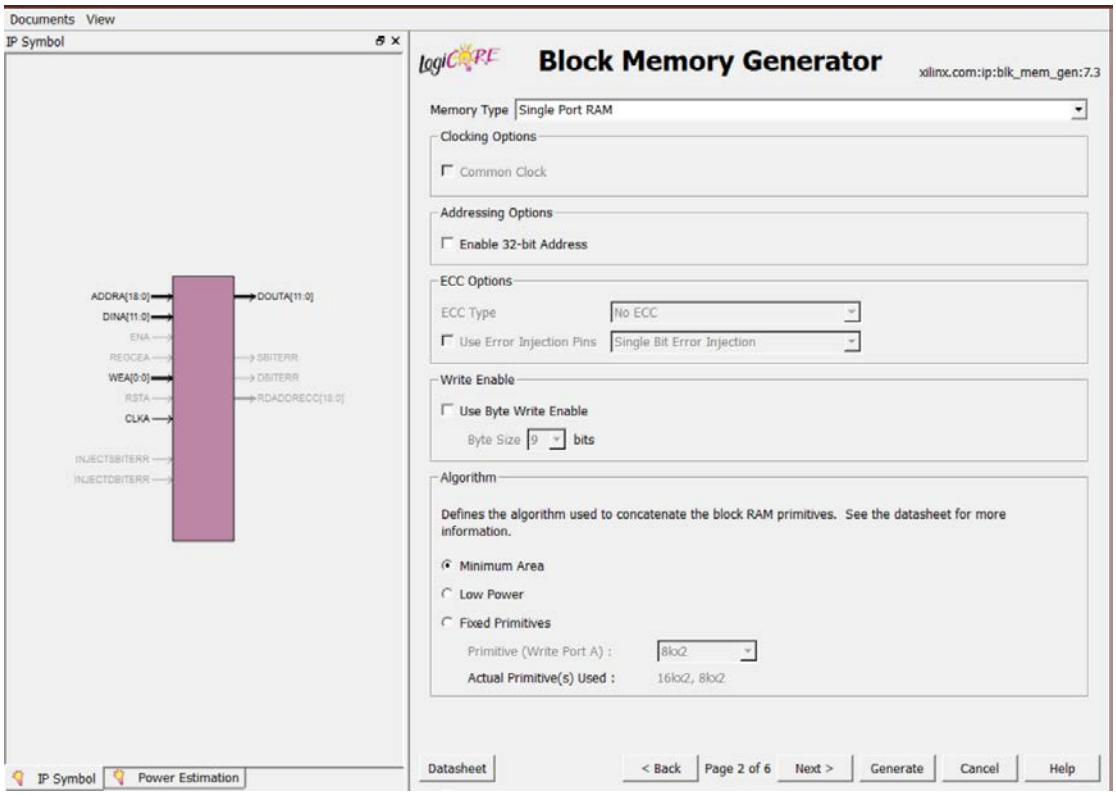
第一次接触 VGA，感觉很棘手，不知到该从哪里下手，反复研究老师给的 VGAdemo 还有相关 PPT 之后有了一些了解，发现关键在于将扫描地址与像素信息对应起来，可以将图像导入 IP 内核然后调用，根据当前扫描地址就可以直接得到对应的像素 RGB 信息，方便起见，设计中采用了将图像信息导入 IP 内核，然后根据游戏状态切换调用的 IP 内核的方式，来实现显示不同的图像，相关代码如下：

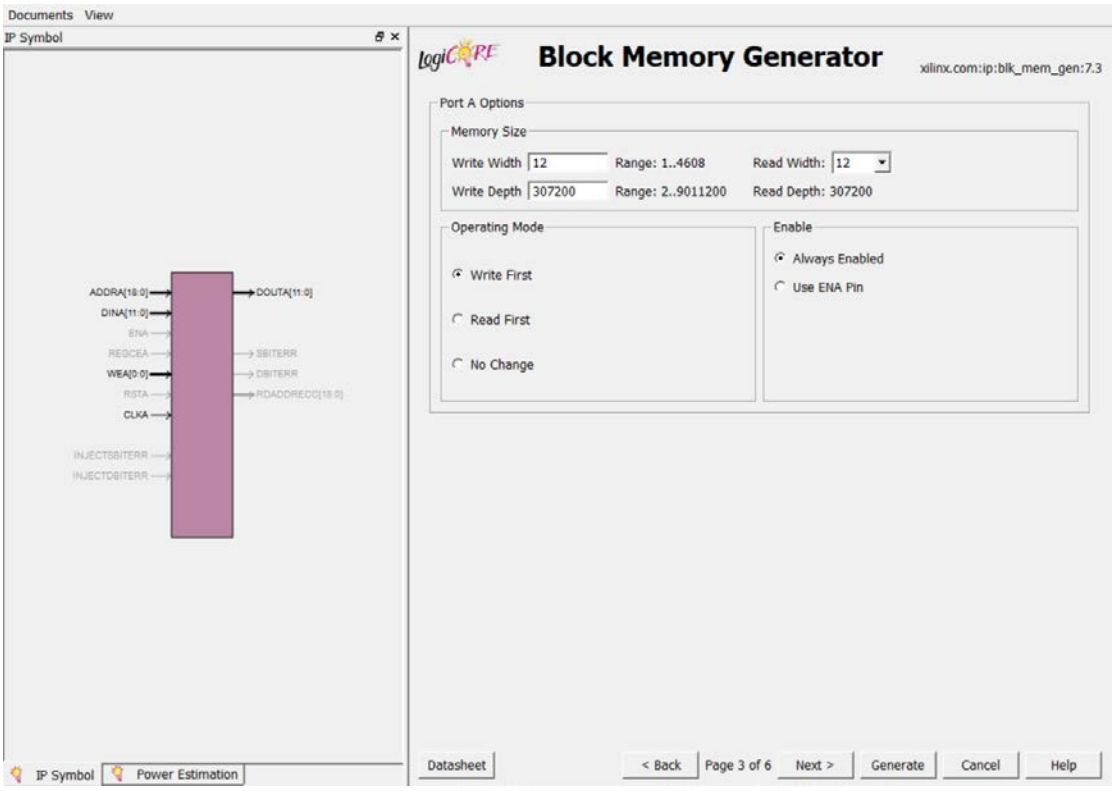
```
module draw(clk, gophers, hc, vc, vga_data);
    input clk;
    input [11:0] gophers;
    input [8:0] hc;
    input [9:0] vc;
    output reg[11:0] vga_data;

    wire[11:0] bg;
    wire[11:0] gp0,gp1,gp2,gp3,gp4,gp5,gp6,gp7,gp8,gp9,gp10,gp11;
    reg [18:0] address;

    background m0(.clka(clk), .addra(address), .douta(bg));
    gopher0  g0(.clka(clk), .addra(address), .douta(gp0));
    gopher1  g1(.clka(clk), .addra(address), .douta(gp1));
    gopher2  g2(.clka(clk), .addra(address), .douta(gp2));
    gopher3  g3(.clka(clk), .addra(address), .douta(gp3));
    gopher4  g4(.clka(clk), .addra(address), .douta(gp4));
    gopher5  g5(.clka(clk), .addra(address), .douta(gp5));
    gopher6  g6(.clka(clk), .addra(address), .douta(gp6));
    gopher7  g7(.clka(clk), .addra(address), .douta(gp7));
    gopher8  g8(.clka(clk), .addra(address), .douta(gp8));
    gopher9  g9(.clka(clk), .addra(address), .douta(gp9));
    gopher10 g10(.clka(clk), .addra(address), .douta(gp10));
    gopher11 g11(.clka(clk), .addra(address), .douta(gp11));

    always @(*) begin
        address<=hc*640+vc;
        if(gophers[0])begin
            vga_data<=gp0;
        end else if(gophers[1])begin
            vga_data<=gp1;
        end else if(gophers[2])begin
            vga_data<=gp2;
        end else if(gophers[3])begin
            vga_data<=gp3;
        end else if(gophers[4])begin
            vga_data<=gp4;
        end else if(gophers[5])begin
            vga_data<=gp5;
        end else if(gophers[6])begin
```

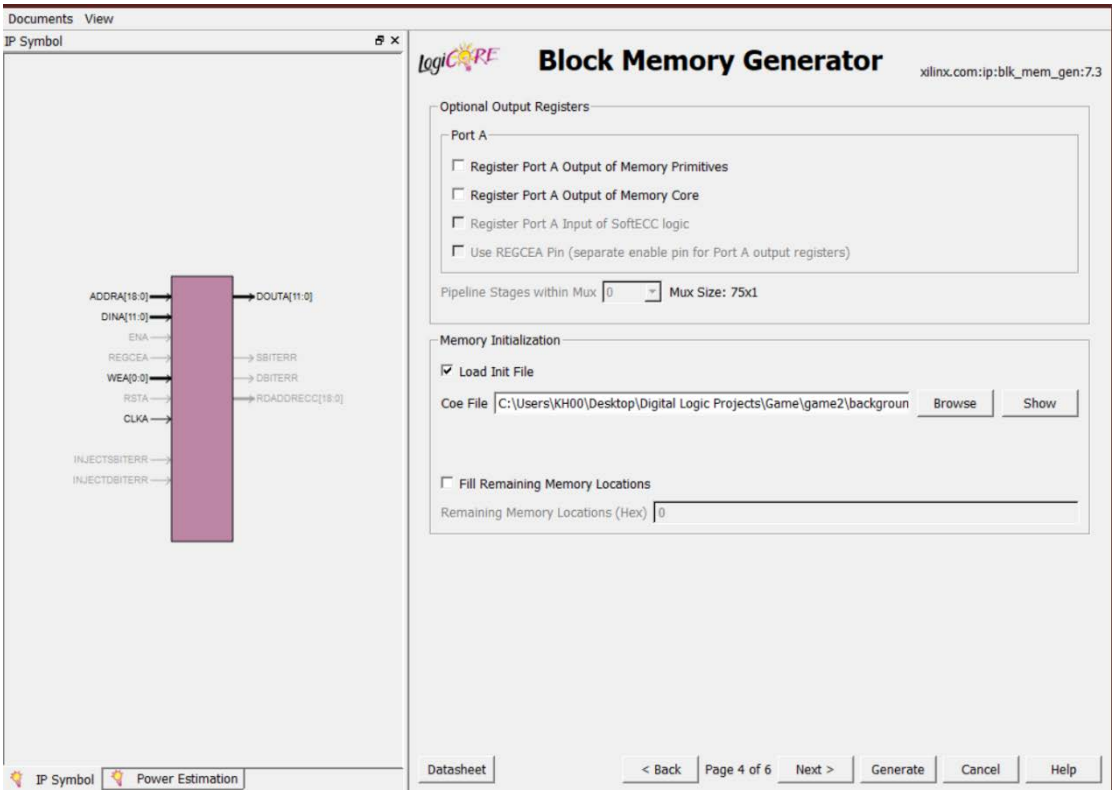
```
            vga_data<=gp6;
        end else if(gophers[7])begin
            vga_data<=gp7;
        end else if(gophers[8])begin
            vga_data<=gp8;
        end else if(gophers[9])begin
            vga_data<=gp9;
        end else if(gophers[10])begin
            vga_data<=gp10;
        end else if(gophers[11])begin
            vga_data<=gp11;
        end else begin
            vga_data<=bg;
        end
    end
endmodule
```

其中对 IP 内核进行调用时，由于不进行写的操作，所以忽略了接口 WEA 还有 DINA，IP 内核生成时，需依次选择 single port，将 read width 设置位 12，read depth 设置为 307200 (480*640)，然后预先导入 coe 文件。




Documents View
IP Symbol
ADDRA[18:0]
DINA[11:0]
ENA
REGCEA
WEA[0:0]
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
DOUTA[11:0]
SBITERR
DBITERR
RDADDRECC[18:0]
IP Symbol
Power Estimation
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
Memory Type
Single Port RAM
Clocking Options
Common Clock
Addressing Options
Enable 32-bit Address
ECC Options
ECC Type
No ECC
Use Error Injection Pins
Single Bit Error Injection
Write Enable
Use Byte Write Enable
Byte Size
9
bits
Algorithm
Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.
Minimum Area
Low Power
Fixed Primitives
Primitive (Write Port A) :
8kx2
Actual Primitive(s) Used :
16kx2, 8kx2
Datasheet
< Back
Page 2 of 6
Next >
Generate
Cancel
Help


Documents View
IP Symbol
ADDRA[18:0]
DINA[11:0]
ENA
REGCEA
WEA[0:0]
RSTA
CLKA
INJECTSBITERR
INJECTDBITERR
DOUTA[11:0]
SBITERR
DBITERR
RDADDRECC[18:0]
IP Symbol
Power Estimation
Block Memory Generator
xilinx.com:ip:blk_mem_gen:7.3
Port A Options
Memory Size
Write Width
12
Range: 1..4608
Read Width:
12
Write Depth
307200
Range: 2..9011200
Read Depth: 307200
Operating Mode
Write First
Read First
No Change
Enable
Always Enabled
Use ENA Pin
Datasheet
< Back
Page 3 of 6
Next >
Generate
Cancel
Help



3.1.5 将 bmp 位图转换为 coe 文件

这部分需要用到图像信息处理的一些知识，查阅资料后，用 C 语言写了一个小程序，用来把 bmp 位图转换为 IP 内核所需的 coe 文件。

相关代码如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAXN 500
const int INF=0x7FFFFFFF;

int map[MAXN][MAXN],cost[MAXN];
int visited[MAXN];
int N,M;

void minTree(int v){
    int dist[MAXN];
    memset(visited,0,(N+1)*sizeof(int));
    int r=N-2;
    int s=(v==N)?1:v+1;
    for(int i=1;i<=N;++i){
        dist[i]=map[s][i];
    }
    visited[v]=visited[s]=1;
    int next;
    while(r--){
        int min=INF;
```

```
        for(int i=1;i<=N;++i){
            if(!visited[i] && min>dist[i]){
                min=dist[i];
                next=i;
            }
        }
        if(min==INF){
            cost[v]=INF;
            break;
        }
        cost[v]+=min;
        visited[next]=1;
        for(int i=1;i<=N;++i){
            if(!visited[i] && map[next][i]<dist[i]){
                dist[i]=map[next][i];
            }
        }
    }
}
int main(){
    scanf("%d%d",&N,&M);
    for(int i=1;i<=N;++i){
        for(int j=1;j<=N;++j){
            map[i][j]=INF;
        }
    }
    for(int i=0;i<M;++i){
        int city1,city2,c,status;
        scanf("%d%d%d%d",&city1,&city2,&c,&status);
        if(status)
             map[city1][city2]=map[city2][city1]=0;
        else
             map[city1][city2]=map[city2][city1]=c;
    }
    int max_cost=0;
    for(int i=1;i<=N;++i){
        cost[i]=0;
        minTree(i);
        if(cost[i]>max_cost)
            max_cost=cost[i];
    }
    if(max_cost==0)
        printf("0\n");
    else{
        int first=1;
        for(int i=1;i<=N;++i){
            if(cost[i]==max_cost){
                printf("%s%d", first?"":" ",i);
                first=0;
            }
        }
        printf("\n");
    }

    system("pause");
    return 0;
}
```

3.1.6 开关/按键防抖动

阵列键还有开关输入都是人工行为，可能会存在不稳定或者抖动，造成输入信号的不稳定，通过限制信号脉冲时间，可以滤掉那些不稳定的输入，相关代码如下：

```
module AntiJitter(clk, I, O);
    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
```

3.2 仿真模拟

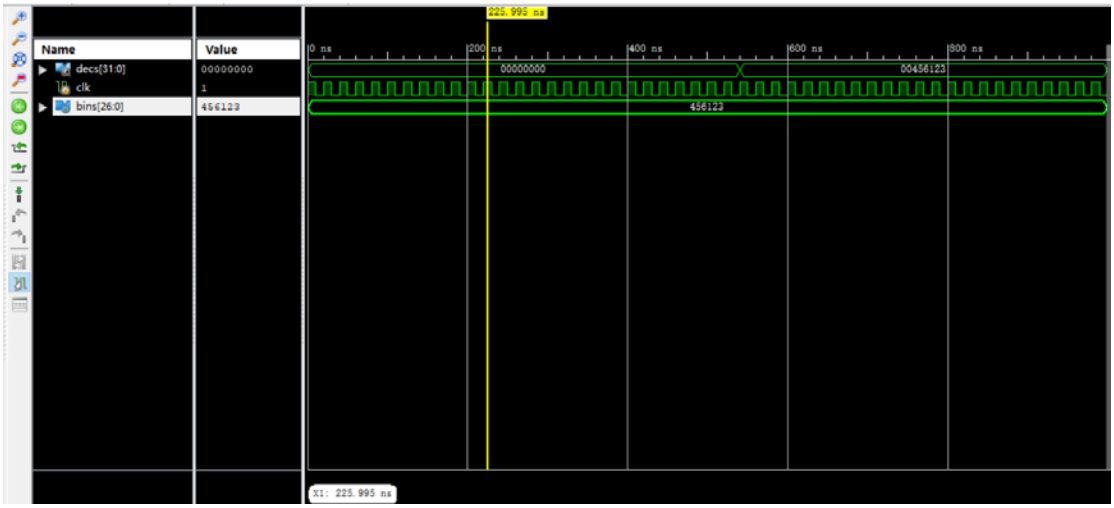
设计中为防止出现重大错误，对几个关键模块进行了仿真模拟，分别为 binToBCDs 模块，numToBits 模块，GopherCreater 模块，scoreCnt 模块

3.2.1 binToBCDs 模块

激励代码：

```
    initial begin
        clk = 0;
        bins = 456123;
        #100;
    end

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

仿真波形：

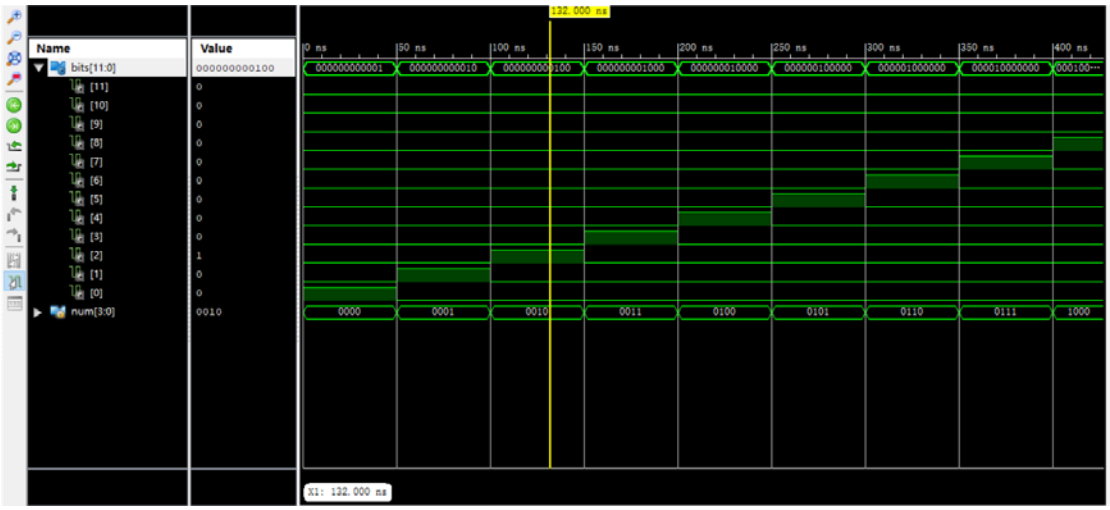


3.2.2 numToBits 模块

激励代码：

```
initial begin
    num = 0; #50;
    num = 1; #50;
    num = 2; #50;
    num = 3; #50;
    num = 4; #50;
    num = 5; #50;
    num = 6; #50;
    num = 7; #50;
    num = 8; #50;
    num = 9; #50;
    num = 10; #50;
    num = 11; #50;
end
```

仿真波形：

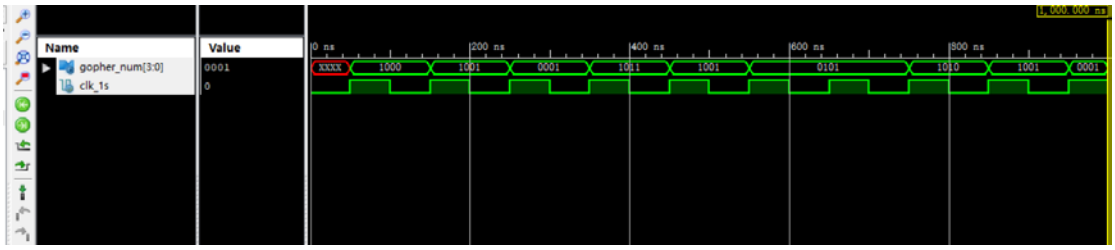


3.2.3 gopherCreater 模块

激励代码:

```
initial begin
    // Initialize Inputs
    clk_1s = 0;
    #100;
end

always begin
    clk_1s=0;
    #50;
    clk_1s=1;
    #50;
end
```

仿真波形;



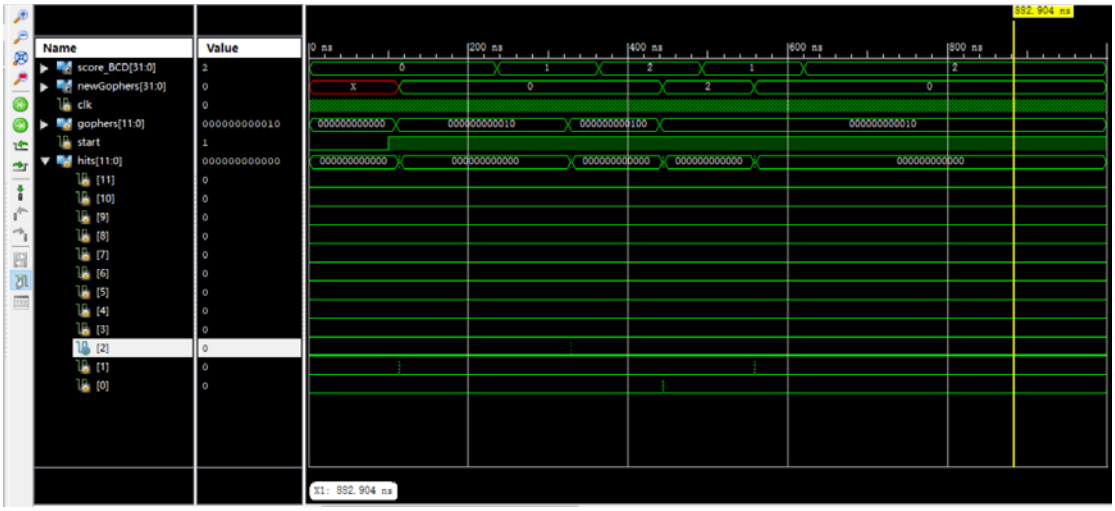
3.2.4 scoreCnt 模块

激励代码:

```
initial begin
    // Initialize Inputs
    clk = 0;
    gophers = 0;
    start = 0;
    hits = 0;
    #100;

    start=1;
    #10
    gophers=2;
    #3;
    hits=2;
    #2;
    hits=0;
    #10;
    #200;

    gophers=4;
    #3;
    hits=4;
    #2;
    hits=0;
    #10;
```

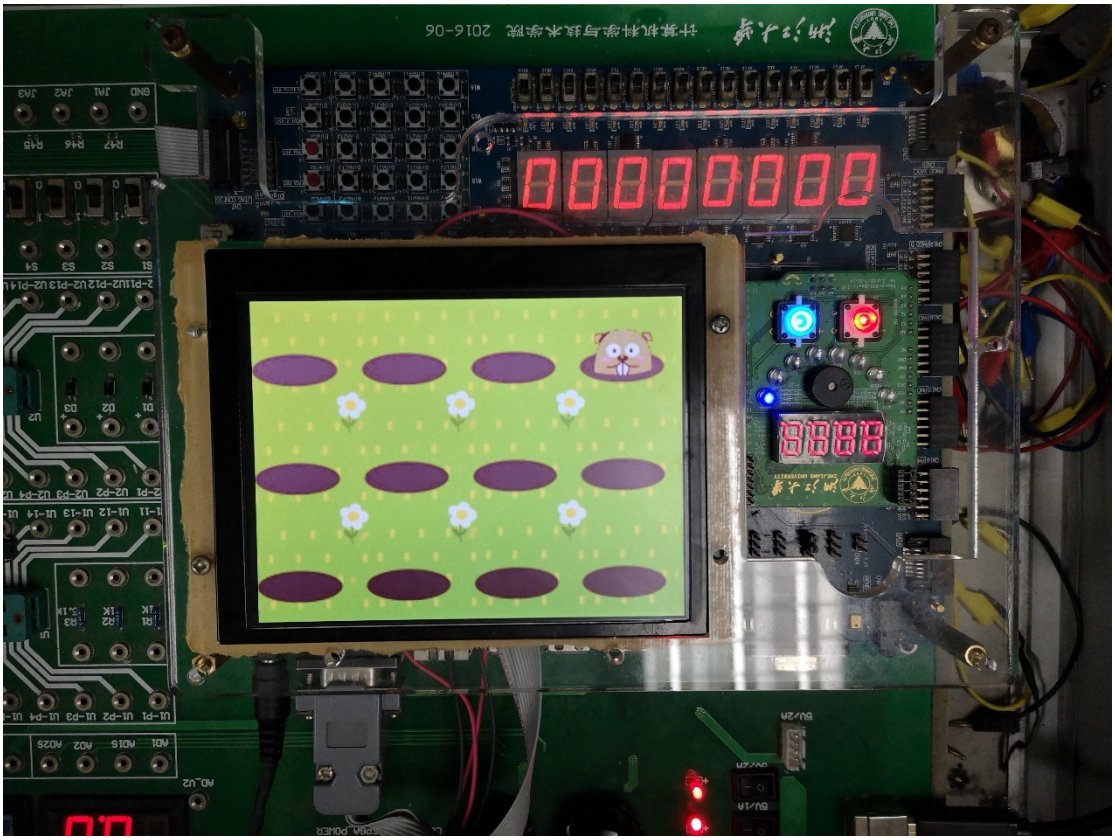
```
        #100;

        gophers=2;
        #3;
        hits=1;
        #2;
        hits=0;
        #10;
        #100;

        gophers=2;
        #3;
        hits=2;
        #2;
        hits=0;
        #10;
    end

    always begin
        clk=1;
        #2;
        clk=0;
        #2;
    End
```

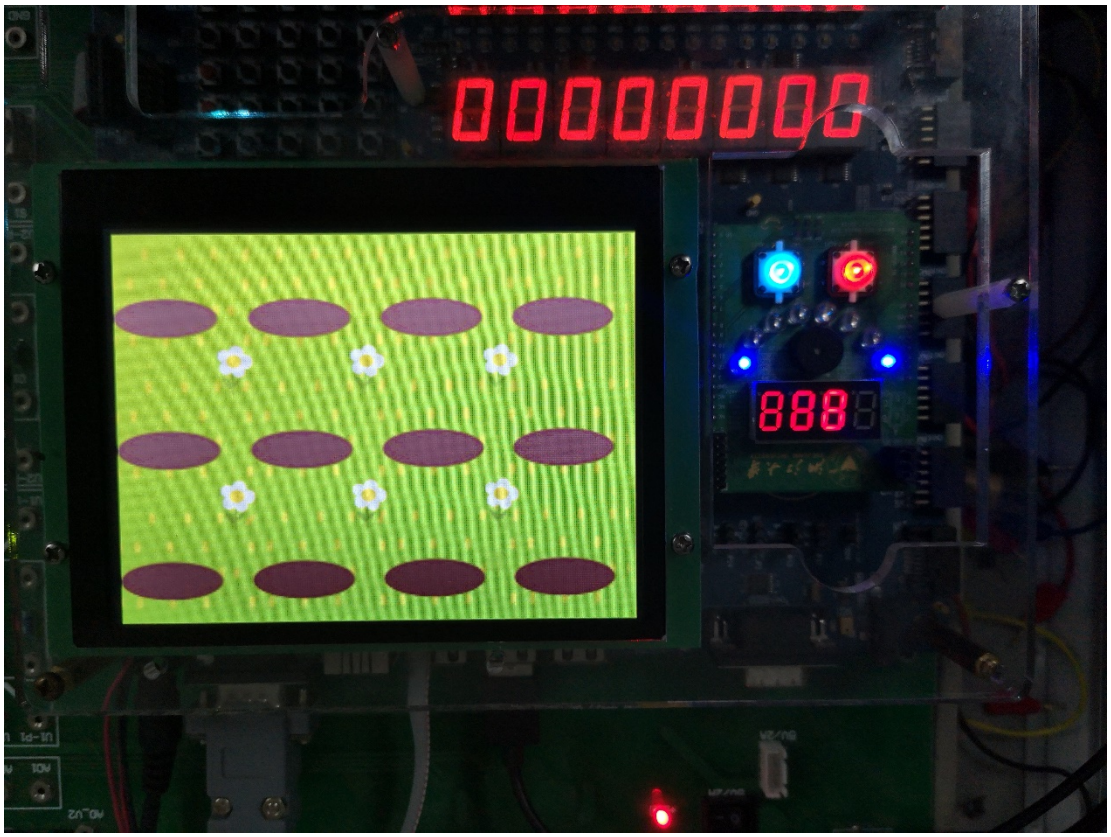
仿真波形:

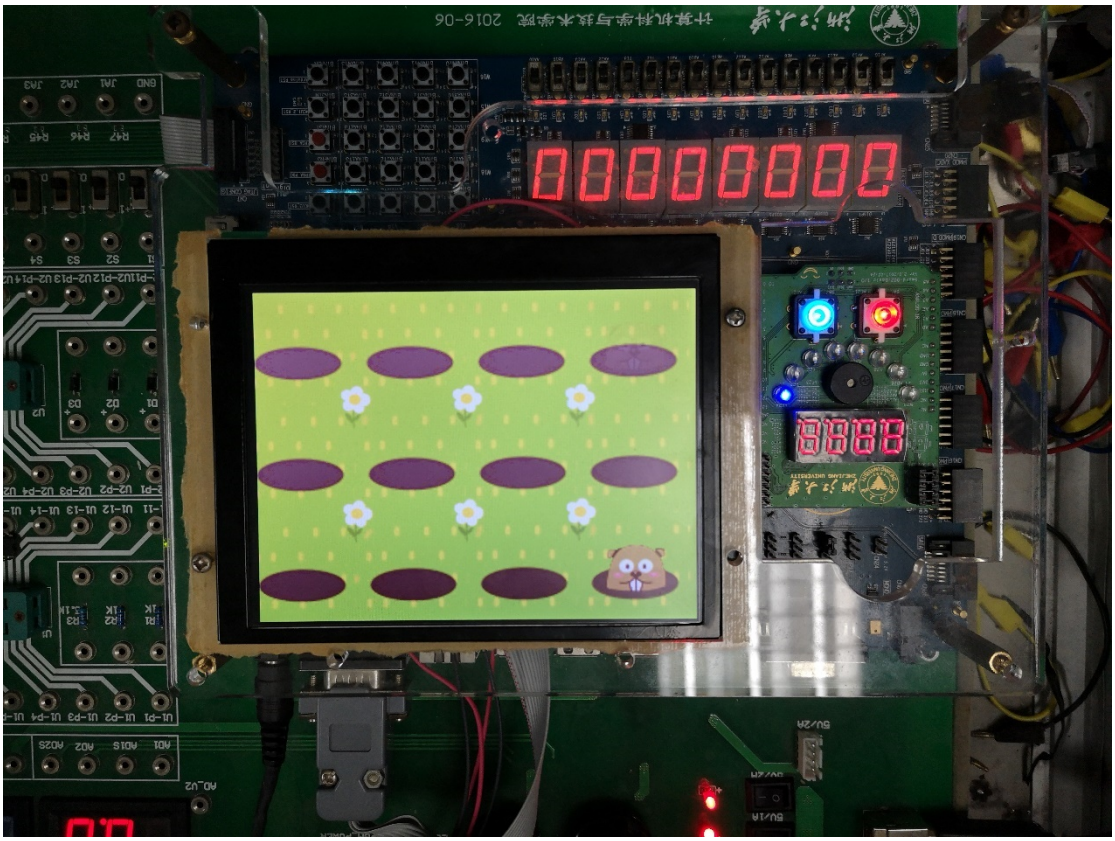


四．结果展示

下面是游戏在 sword 板上运行时的一些照片：







五．可以改进的地方

由于劳动力问题还有时间问题，游戏设计的比较粗糙，勉强算完成了，其实说是一个半成品或者残次品更加合适，因此还有很多可以改进的地方，首先是游戏的画面质量，可以通过用 Photoshop 软件进行加工，让画质在分辨率有限的情况下更好一点，还有游戏画面的设计问题，可以设计的更美观一些，改善游戏体验。

由于直接采用调用不同 ip 内核的方式来显示不同的图片的方式，画面的切换略显生硬，可以改进为动态读写 vga_data，使画面更有动画感，比如可以将地鼠探出头来的过程与时钟联系起来，使地鼠探头的过程更加流畅，或者根据阵列键输入，添加打击特效，而不只是生硬的图片切换。

还可以使游戏的内容更加丰富，比如增加游戏模式，通过开关选择游戏模式，或者游戏难度，通过不同的时钟分频控制地鼠的探头频率，从而控制游戏的难度，或者增加双人甚至多人游戏模式，通过引入 PS2 键盘外设，来增加可能的游戏输入，使游戏的可玩性更强。

六．感悟与心得

做打地鼠这个游戏主要是看到之前的优秀报告中有一份也是做的打地鼠（未参考其任何代码），但没有用 VGA 显示，觉得自己可以做的更好一些，实际上自己在做的过程中也遇到了许多困难。

整个游戏的从构思到设计，再到实现，调试，得到最后的结果，经过了很长的一段时间，由于是单独完成，期间遇到的问题有很多都没有得到有效的解决，从而做了很多无用功，但是感觉真的学到了很多东西，虽然游戏很普通，逻辑关系也很简单，但在硬件底层层面，涉及到了许许多多学过的知识，在完成大程序的过程中，理论课的许多知识也得到了复习和巩固。

在写代码的过程中，有很多问题都需要查阅资料，阅读官方文档，或者别人的代码，感觉自己在这方面的能力得到了很大的提升，还有遇到问题时的心态，也得到了锻炼。

有点遗憾由于各方面因素的限制没能把游戏做的很令人满意，但这次写大程序的经历确实不可多得，感觉学到了很多，而不只是完成一次作业。