Lab H7 Report

PB21050996 徐航宇 PB21111696 闫泽轩 PB21111639 李牧龙

项目介绍

锵锵，可莉登场
~~哒哒哒~~
这是一款模仿谷歌小恐龙设计的小游戏，你将化身为可莉或其他角色，展开一场惊心动魄的大冒险！冒险的途中，注意避开史莱姆和天上飞的傻乎乎的派蒙哦。 至于为什么三个不玩原神的人凑在一起，会设计出这款游戏呢？这就不得而知了。（我们自己也不知道。）

实验目的与内容

实验目的：

综合运用实验课和理论课所学知识，自主设计和完成实验项目

实验内容：

制作类似Chrome浏览器无网络连接时的彩蛋跑酷游戏，具体要求如下：

1. 使用键盘进行操控：Enter进入游戏，数字键选择角色，w操作角色跳跃，空格键暂停游戏，Esc回到开始界面。
2. 游戏界面分为开始界面、选角色界面、游戏进行界面、暂停界面、游戏结束界面。
3. 游戏画面通过VGA接口输出到显示屏上。玩家通过控制角色跳跃躲避怪物，一旦碰到怪物则游戏结束。
4. 游戏得分由时间决定。每存活0.1秒加一分，上限为99999分。分数显示在画面右上角。
5. 在游戏的不同界面，通过3.5mm音频接口播放不同的BGM。角色跳跃时播放对应的音效。

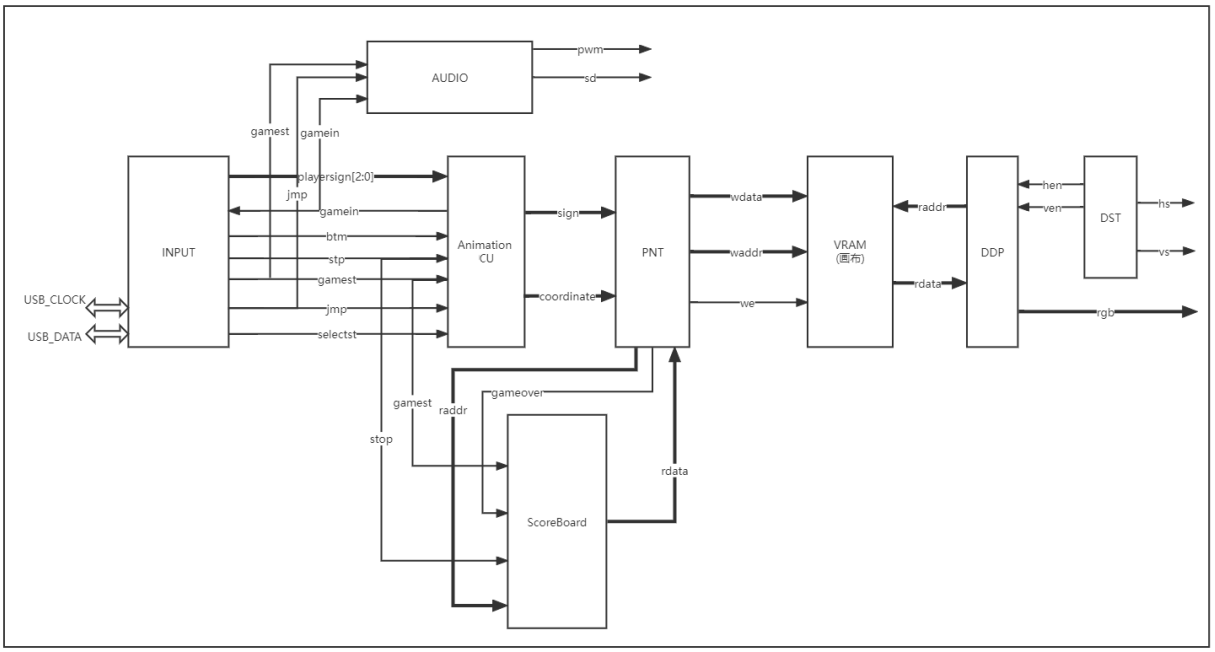
逻辑设计

顶层模块：

实现要求：

> 通过连接键盘输入模块、音频输出模块、动画模块等各种模块，实现实验内容中的具体要求。

逻辑图：



动画控制与动画显示模块

实现要求

> 接收外部有关游戏进程及按键的信号，并在显示屏上显示出对应的动画
> 具体要求完成：
> 1.在菜单界面时，接收到gamest信号后，转换到选择角色界面
> 2.在选择角色界面时，接收到gamest信号后，转换到游戏界面
> 3.在游戏界面时：
>
> - 按照选择角色界面的选择结果，显示对应人物的动画
> - 接收到jump信号后，人物跳跃
> - 人物在地面上时，播放跑动动画
> - 随机生成空中或地面的怪物
> - 怪物向左匀速运动
> - 人物和怪物的原图片透明部分显示为背景
> - 人物和怪物发生碰撞后，游戏暂停，并发出gameover信号
> - 背景向左滚动

实现原理：

由于需要实现的动画及其转换较多、较为复杂，总结出了以下的游戏设计模式，这种模式经过实践与摸索，被不断更新完善，并最终形成了一套较为完整的体系框架。这一框架受到了Unity这一著名的游戏开发引擎的启发，我们也将在后面对这一框架与Unity的各项功能建立对应，并由此说明这一框架对于使用verilog硬件级语言开发简单2D游戏的通用性。

我们首先提出如下的几个核心设计思想：

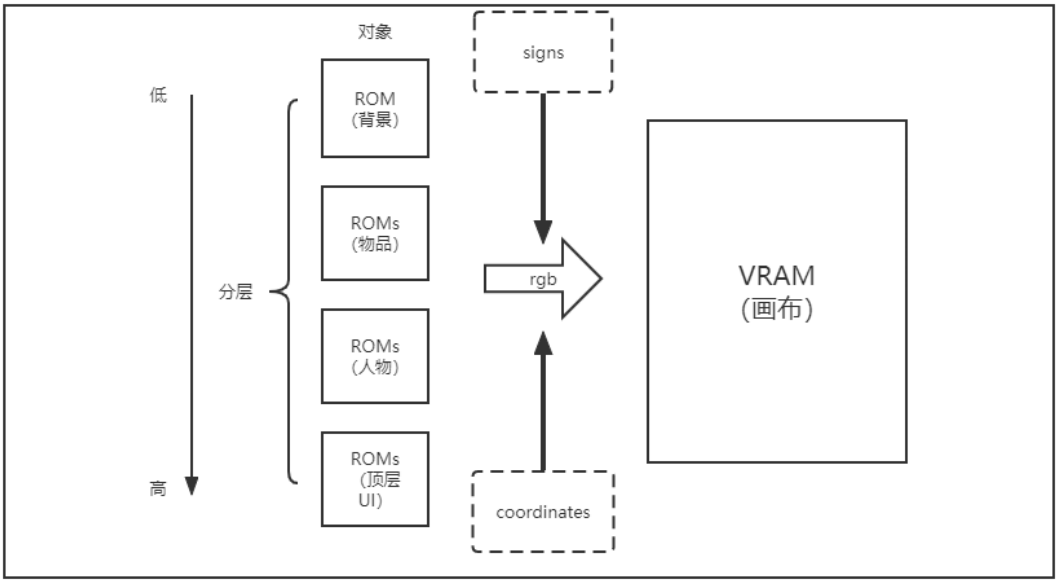
1.**共画布**：在这一设计中，所有动画显示都读取自一个VRAM，我们称之为"画布"。在游戏进行的每一"帧"，都对画布进行"刷新"——即对整个VRAM进行一次刷写操作。
这一设计本质上分离了实际的显示屏显示模块(DST、DDP等)与动画设置模块，使得可以专注于对画布的读写，而无需关心具体的硬件显示。对应到Unity，即，Scene（画布）与camera（摄像机），缺点是无法实现camera对人物等的跟踪。

2.**对象分离**：将各个需要显示的动画拆分为若干对象，并将所需的图片信息分别存储于各个ROM中。每一帧根据对象的sign和coordinate将动画信息写到画布VRAM的正确位置上。
这一设计本质上拆解了各个对象，使得可以分别对各个对象的动画进行控制。对应到Unity，即gameobject（游戏对象）的设计。

3.**对象分层**：各对象根据显示的优先级分层，可能发生冲突的对象置于不同层中，在对画布的某坐标进行写时，按层级从高到低，写入最高有效层的信息。 可在指定的两层发生冲突时，发出反馈信号，以此实现碰撞反馈等机制。
各层有分离的rdata和raddr接线，为透明度的实现提供便利。
对应到Unity，即gameobject的sorting layer（显示层级）与碰撞反馈等。

4.**面向对象**：将动画的显示，拆分为对对象的位置及是否显示等属性的修改，并交由动画控制模块进行控制（例子：跳跃、跑动）
这一设计本质上分离了动画切换与位移，是**对象分离**思想的延续。对应到Unity，即gameobject的坐标与图片等属性。

总的来说，该框架通过画布来分隔后端的动画控制与前端的硬件显示，并利用coordinate和sign等对象性质，有层次地，从存储有图片信息的ROM中选择合适的信息，写到画布上，如图：



下面通过在这里的应用，对该框架做进一步解释：

核心代码：

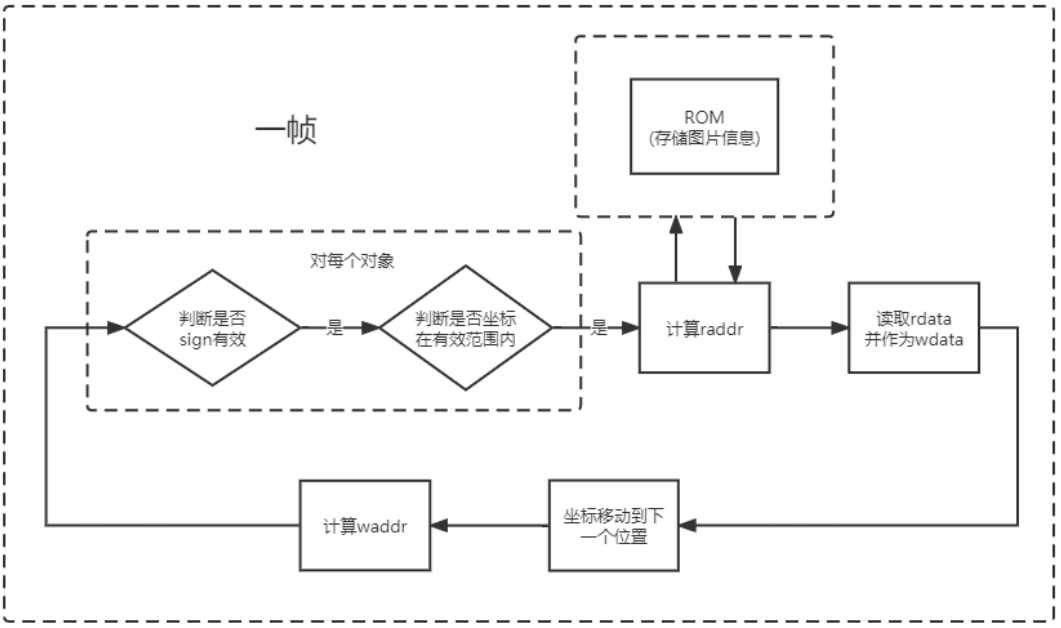
PNT_CU

绘画模块，接收来自Animation_CU的各对象coordinate与sign信息，并向画布VRAM传输waddr,wdata,we等信号，产生周期性的对画布的刷新。即直接控制着对画布的写操作的进行。 具体代码详见sources文件，下面来具体分析：

总体框架

```
//assign waddr={{7{1'b0}},x_coordinate}+H_LEN*{{7{1'b0}},y_coordinate};
//assign raddr=x_coordinate_r+6'b110010*y_coordinate_r;

//写地址的计算及结束信号
always @(posedge clk) begin
    //此处中间省略
    waddr<={{7{1'b0}},x_coordinate}-3+H_LEN*{{7{1'b0}},y_coordinate};
    //此处中间省略
end

//坐标的计算
always @(posedge clk) begin
    //此处中间省略
    if(dead_sign
        && x_coordinate>=8'd70
        && x_coordinate<=8'd129
        && y_coordinate>=8'd52
        && y_coordinate<=8'd97
        ) begin
        x_coordinate_r<=x_coordinate-70;
        y_coordinate_r<=y_coordinate-52;
    //此处中间省略
end

//读地址的计算
always @(posedge clk) begin
    //此处中间省略
    if(dead_sign
       && x_coordinate>=8'd71
       && x_coordinate<=8'd130
       && y_coordinate>=8'd52
       && y_coordinate<=8'd97
    ) begin
        raddr<=x_coordinate_r+15'd60*y_coordinate_r;
    end
    //此处中间省略
end

//wdata的确定
always @(posedge clk) begin
    //此处中间省略
    if(dead_sign
       && x_coordinate>=8'd73
       && x_coordinate<=8'd132
       && y_coordinate>=8'd52
       && y_coordinate<=8'd97
    ) begin
        wdata<=dout_dead;
    end
    //此处中间省略
end
```

1.逻辑解释： 每一帧对画布进行一次刷写，在每次刷写中，对画布的每一个位置(x,y)的rgb逐一更新。写入时需要的wdata，来自于存储着各对象的各图片的ROM，并根据按照优先级顺序，判断对应sign是否有效以及(x,y)是否在有效显示范围内，以选择合适的ROM中的信息，赋值给wdata。如此，在(x,y)扫过整个画布后，画布便完成了一次更新。 由于刷写信息依赖于sign与coordinate，对动画的控制转变为了对sign与coordinate的修改。



2.使用四个always块，分别计算写地址的计算及结束信号、计算读地址坐标、计算读地址、读取wdata。一方面，这样使得模块结构更清晰，更方便编写、调试与修改；另一方面，可以应对读写时的时序延迟，并提高并行性能，防止时序爆炸。

- 起初，使用组合逻辑描述，但由于每个时钟周期所需要进行的判断较多较复杂，时序紧张甚至爆炸，电路无法在10ns的时钟周期内完成应完成的工作。调整为时序逻辑并分块后，减少了各判断与计算间的依赖性，极大地优化了电路的时序。
- 通过调整坐标的计算，以及对不同always块中的坐标判断做恰当位移，使得图像能够准确显示，而不会因为时序的延迟产生偏差。上面的代码中以死亡动画的播放为例，展示了这一过程：
  - 由于从确定坐标到得到对应的wdata的过程共有3个时钟周期的延迟，所以在计算waddr时，对其减3，以得到三个周期前预期写入的地址。
  - 计算读地址与确定wdata时，分别对有效范围做了+1和+3的位移，以保证契合预期的坐标及显示。

3.各个always块的功能：

- 写地址的计算及结束信号：

```
//写地址的计算及结束信号
always @(posedge clk) begin
    if(!rstn) begin
        pe<=1; ce<=1; s<=0; we<=0;
    end
    else begin
        waddr<={{7{1'b0}},x_coordinate}-3+H_LEN*{{7{1'b0}},y_coordinate};

        if(q==0) begin         //开始一次刷写
            s<=1; ce<=0; we<=1;
            x_coordinate<=0;
            y_coordinate<=0;
        end
        else if(s) begin                //刷写时
            if(y_coordinate==V_LEN&&x_coordinate==8'd3) begin    //结束刷写(后延三位)
                y_coordinate<=0;

                we<=0;
                ce<=1;
                pe<=1;
                s<=0;
            end
            else if(x_coordinate+1==H_LEN) begin
                x_coordinate<=0;
                y_coordinate<=y_coordinate+1;
            end
            else begin
                x_coordinate<=x_coordinate+1;
            end
        end
        else if(!s) begin           //非刷写时
                pe<=0;
            end
    end
end
```

借助计数器，每一帧(TimePerFrame)，即每隔一段时间，都对画布进行一次刷新。 刷新过程中,s=1,并令(x,y)逐一扫过整个画布，同时根据(x,y)计算本周期将要刷写的位置(waddr)。

- 坐标的计算

```
//人物
if(x_coordinate>=p1_x_coordinate
&& x_coordinate<=p2_x_coordinate
&& y_coordinate>=p1_y_coordinate
&& y_coordinate<=p2_y_coordinate
) begin
    //碰撞
    if(sm_sign<4'd12
    && x_coordinate>=sm1_x_coordinate
    && x_coordinate<=sm2_x_coordinate
    && y_coordinate>=sm1_y_coordinate
    && y_coordinate<=sm2_y_coordinate
    ) begin
        gameover<=1;
    end
    else if(smm_sign<4'd12
    && x_coordinate>=smm1_x_coordinate
    && x_coordinate<=smm2_x_coordinate
    && y_coordinate>=smm1_y_coordinate
    && y_coordinate<=smm2_y_coordinate
    ) begin
        gameover<=1;
    end

    x_coordinate_r<=x_coordinate-p1_x_coordinate;
    y_coordinate_r<=y_coordinate-p1_y_coordinate;
end
```

根据第一个always块得到的(x,y)坐标，计算将要写入的对象图片数据所需的点在对应ROM中的坐标。以人物为例，在确定人物的sign有效且(x,y)在有效的显示范围内后，计算出x_coordinate_r与y_coordinate_r，为计算raddr做准备；并在此过程中完成碰撞的判定。代码展示如上。

- 读地址的计算

```
if(x_coordinate>=p1_x_coordinate+8'd1
    && x_coordinate<=p2_x_coordinate+8'd1
    && y_coordinate>=p1_y_coordinate
    && y_coordinate<=p2_y_coordinate
 ) begin
     raddr<=x_coordinate_r+15'd20*y_coordinate_r;
 end
```

根据计算出的坐标x_coordinate_r与y_coordinate_r，计算对应的raddr，传输给ROM，

- wdata的确定

```
if(dead_sign
    && x_coordinate>=8'd73
    && x_coordinate<=8'd132
    && y_coordinate>=8'd52
    && y_coordinate<=8'd97
 ) begin
     wdata<=dout_dead;
 end
```

将合适的ROM中得到的rdata赋值给wdata,并最终写到画布上。

动画切换

- 通过player_sign的值，选择不同人物对应的图片，从而实现了人物切换：

```
case(player_sign)
 3'd0: begin ···
 end
 3'd1: begin ···
 end
 3'd2: begin ···
 end
 3'd3: begin ···
 end
  ···
 3'd7: begin ···
 end
endcase
```

- 通过jump_sign是否有效，判断使用跳动动画还是跑动动画：

```
if(jump_sign==3'b000) begin      //没有跳起
  ···
end
else begin
  ···
end
```

- 通过run_sign的值，选择人物跑动时的不同状态的图片。

```
case (run_sign)
 2'b00: begin
     if(dout_p0_run0[3:0]==4'b000) wdata<=dout_bk;
     else wdata<=dout_p0_run0[15:4];
 end
 2'b01:begin
     if(dout_p0_run1[3:0]==4'b000) wdata<=dout_bk;
     else wdata<=dout_p0_run1[15:4];
 end
 2'b10: begin
     if(dout_p0_run2[3:0]==4'b000) wdata<=dout_bk;
     else wdata<=dout_p0_run2[15:4];
 end
 2'b11: begin
     if(dout_p0_run3[3:0]==4'b000) wdata<=dout_bk;
     else wdata<=dout_p0_run3[15:4];
 end
endcase
```

优先级显示

```
//以读地址的计算为例
   always @(posedge clk) begin
          //顶层
       if(menu_sign||select_sign) begin
       end
       else if(dead_sign···
       end
       else if(stop_sign···
       end
       else begin      //第二层
           //计分板
           if(   x_coordinate>=8'd151···
           ) begin
           end
           //人物
           else if(x_coordinate>=p1_x_coordinate+8'd1···
           ) begin
           end
           //障碍物1
           else if(sm_sign<4'd12···
           ) begin
           end
           //障碍物2
           else if(smm_sign<4'd12···
           ) begin
           end
       end
       //背景循环播放
       if(b_x_coordinate+bk_x_coordinate<=199) begin
       end
       else begin
       end
```

通过将不同对象的sign和coordinate是否有效的判定及操作，按照优先级的层次放在if-else块中。这样，在高优先级的对象被显示时，将忽略其他低优先级的对象。以读地址的计算为例，展示如上。

透明显示

```
    if(dout_p0_run0[3:0]==4'b000) wdata<=dout_bk;
    else wdata<=dout_p0_run0[15:4];
```

对于需要部分透明显示的对象，在将图片信息转化为coe文件时，额外存储其透明度信息(这里为4位)，并在获取到rdata时，判断rdata中的alpha（即rdata[3:0]）是否超过阈值（可自行设定，这里判断是否为0），如果确定该点处透明显示，则将下一层的rdata赋给wdata（这里即背景）

碰撞反馈

```
if(x_coordinate>=p1_x_coordinate
   && x_coordinate<=p2_x_coordinate
   && y_coordinate>=p1_y_coordinate
   && y_coordinate<=p2_y_coordinate
   ) begin
       //碰撞
       if(sm_sign<4'd12
       && x_coordinate>=sm1_x_coordinate
       && x_coordinate<=sm2_x_coordinate
       && y_coordinate>=sm1_y_coordinate
       && y_coordinate<=sm2_y_coordinate
       ) begin
           gameover<=1;
       end
```

当可能发生碰撞的两个对象，如这里的人物和障碍物，在某一点处，同时sign有效且坐标在有效范围内，即被判定为要同时显示，则令gameover信号有效。 缺点是：这一反馈信号依赖于画布的刷写，即不是脉冲信号，在未碰撞时信号始终无效，但在碰撞后信号时而有效时而无效，不够稳定。但可以通过状态机将其转换为稳定的信号（我们也正是这么做的）。

Animation_CU

动画控制模块，接收外部有关游戏进程及按键的信号，并进行分析，产生coordinate与sign等控制信息，传输给绘画模块。即实际控制着动画的变化与显示。 具体代码详见sources文件，下面来具体分析：

信号转换

```
always @(posedge clk) begin
        if(!rstn) begin
            menu_sign<=1;
            //清空信号
            select_sign<=0;
            background_sign<=0;dead_sign<=0;stop_sign<=0;gaming<=0;
            jump_rstn<=0;
            roll_rstn<=0;
        end
        else if(bk_to_menu) begin
            menu_sign<=1;
            //清空信号
            select_sign<=0;
            background_sign<=0;dead_sign<=0;stop_sign<=0;gaming<=0;
            jump_rstn<=0;
            roll_rstn<=0;
        end
        else if(select_st) begin
            select_sign<=1;
            //清空信号
            menu_sign<=0;
            background_sign<=0;dead_sign<=0;stop_sign<=0;gaming<=0;
            jump_rstn<=0;
            roll_rstn<=0;
        end
        else if(gamest) begin
            background_sign<=1;gaming<=1;roll_rstn<=0;
            //清空信号
            select_sign<=0;menu_sign<=0;stop_sign<=0;dead_sign<=0;
        end
        if(gaming) begin
            if(gameover) begin
                background_sign<=1;dead_sign<=1;
                gaming<=0;
            end
            else if(stop) begin
                stop_sign<=1;
                gaming<=0;
            end
            else if(jump) begin
                if(jump_sign==2'b00) begin
                    jump_rstn<=0;
                    readytojump<=1;
                end
            end
            else begin
                if(jumping) begin
                    readytojump<=0;
                end
                jump_rstn<=1;
                roll_rstn<=1;
            end
        end
        else begin
                jump_rstn<=1;
                roll_rstn<=1;
            end
    end
```

将外部接收到的信号转换为sign信息，供绘画模块使用。

人物跳动

```
    FreqD #(32) J_fq(
        .k          (JumpTime),
        .clk        (clk),
        .rstn       (jump_rstn),

        .y          (jump_clk)
    );

always @(posedge clk) begin
        if(!rstn) begin
            ···
            jump_rstn<=0;
        end
        else if(其他情况) begin
            ···
            jump_rstn<=0;
        end
        else if(jump) begin
            if(jump_sign==2'b00) begin
                jump_rstn<=0;
                readytojump<=1;
            end
        end
        else begin
            if(jumping) begin
                readytojump<=0;
            end
            jump_rstn<=1;
        end

always @(posedge jump_clk) begin
    if(!rstn||bk_to_menu) begin
        //初始化
        p1_x_coordinate<=p1_x_coordinate0;
        p2_x_coordinate<=p2_x_coordinate0;
        p1_y_coordinate<=p1_y_coordinate0;
        p2_y_coordinate<=p1_y_coordinate0+p_height;

        jump_sign<=3'b000;
        jumping<=0;
    end
    else if(readytojump) begin
        if(jump_sign==2'b00) begin
            jumping<=1;

            p1_y_coordinate<=p1_y_coordinate1;
            p2_y_coordinate<=p1_y_coordinate1+p_height;
        end
    end
    else if(gaming) begin
        case (jump_sign)
            3'b000: begin
                if(jumping) begin
                    jump_sign<=3'b001;

                    p1_y_coordinate<=p1_y_coordinate2;
                    p2_y_coordinate<=p1_y_coordinate2+p_height;
                end
            end
            3'b001: begin
                jump_sign<=3'b010;

                p1_y_coordinate<=p1_y_coordinate3;
                p2_y_coordinate<=p1_y_coordinate3+p_height;
            end
            ·····
            3'b110: begin
                jump_sign<=3'b000;
                jumping<=0;

                p1_y_coordinate<=p1_y_coordinate0;
                p2_y_coordinate<=p1_y_coordinate0+p_height;
            end
            default: begin
                jump_sign<=3'b000;
```

```
            end
        endcase
    end
end
```

利用分频器，在跳动过程中（即jumping有效），每隔RunTime时间，对run_sign递增，并根据run_sign对人物y方向坐标赋不同值。jumping有效时，人物处于跳动过程中；readytojump信号的设计是为了取消跳跃时坐标变化的延迟。

人物跑动

```
    FreqD #(32) RUN_fq(
        .k          (RunTime),
        .clk        (clk),
        .rstn       (rstn),

        .y          (run_clk)
    );

always @(posedge run_clk) begin
        if(!rstn) begin
            run_sign<=2'b00;
        end
        else begin
            if(gaming) begin
                run_sign<=run_sign+1;
            end
        end
    end
```

利用分频器，在游戏过程中（即gaming有效），每隔RunTime时间，对run_sign递增，从而实现对跑动动画的切换。

障碍物滚动与随机刷新

```
    //随机数生成器
       Random random(
           .clk        (clk),
           .rstn       (rstn),
           .en         (en),
           .randnum    (sm_kind)
       );
    always @(posedge roll_clk) begin
           if(!rstn||bk_to_menu) begin
               //初始化
               sm_sign<=4'b1111;
               en<=0;

               sm1_x_coordinate<=sm_x_coordinate_st;
               sm1_y_coordinate<=sm_y_coordinate_st;
               sm2_x_coordinate<=0;
               sm2_y_coordinate<=sm_y_coordinate_st+sm_height;

           end
           else if(gaming) begin
               //怪物1
               //x坐标
               if(sm2_x_coordinate==0) begin
                   en<=0;
                   sm_sign<=sm_kind;
                   sm1_x_coordinate<=sm_x_coordinate_st;
                   sm2_x_coordinate<=sm_x_coordinate_st+sm_length;
               end
               else if(sm2_x_coordinate==1) begin
                   en<=1;
                   sm1_x_coordinate<=sm1_x_coordinate-1;
                   sm2_x_coordinate<=sm1_x_coordinate+sm_length-1;
               end
               else begin
                   sm1_x_coordinate<=sm1_x_coordinate-1;
                   sm2_x_coordinate<=sm1_x_coordinate+sm_length-1;
               end
               //y坐标
               if(sm_sign<4'd8) begin
                   sm1_y_coordinate<=sm_y_coordinate_st;
                   sm2_y_coordinate<=sm_y_coordinate_st+sm_height;
               end
               else if(sm_sign<4'd12) begin
                   sm1_y_coordinate<=pm_y_coordinate_st;
                   sm2_y_coordinate<=pm_y_coordinate_st+pm_height;
               end
               else begin
                   sm1_y_coordinate<=sm_y_coordinate_st;
                   sm2_y_coordinate<=sm_y_coordinate_st+sm_height;
               end
           end
       end
```

以一个障碍物为例，代码如上。（实际上，我们放置了两个，二者的起始位置不同，使得能够错开出现）

- 每隔一段时间，x坐标递减，以实现障碍物的向左移动；x坐标为1时，意味着障碍物将要向左移动越出屏幕，令en=1,使得随机数生成器开始工作，并在x坐标为0时读取随机数结果赋给sm_sign，以随机生成下一次生成的障碍物的类型，并对x坐标重置，使得障碍物回到到屏幕最右侧。其中：sm_sign=0~~7为地面单位，sm_sign=8~~11为空中单位，sm_sign=12~15为空白（会被透明处理，且不发生碰撞）。
- 根据sm_sign，即障碍物类型，确定其y坐标。

背景滚动

```
//Animation_CU中：
   always @(posedge roll_clk) begin
       if(!rstn) begin
           bk_x_coordinate<=0;
       end
       if(gaming) begin
          if(bk_x_coordinate==8'd199) begin
              bk_x_coordinate<=0;
          end
          else begin
              bk_x_coordinate<=bk_x_coordinate+1;
          end
       end

//PNT_CU中：
   //背景循环播放
   always @(posedge clk) begin
      if(b_x_coordinate+bk_x_coordinate<=199) begin
          raddr_bk<=b_x_coordinate+bk_x_coordinate+15'd200*b_y_coordinate;
      end
      else begin
          raddr_bk<=b_x_coordinate+bk_x_coordinate-199+15'd200*b_y_coordinate;
      end
   end
```

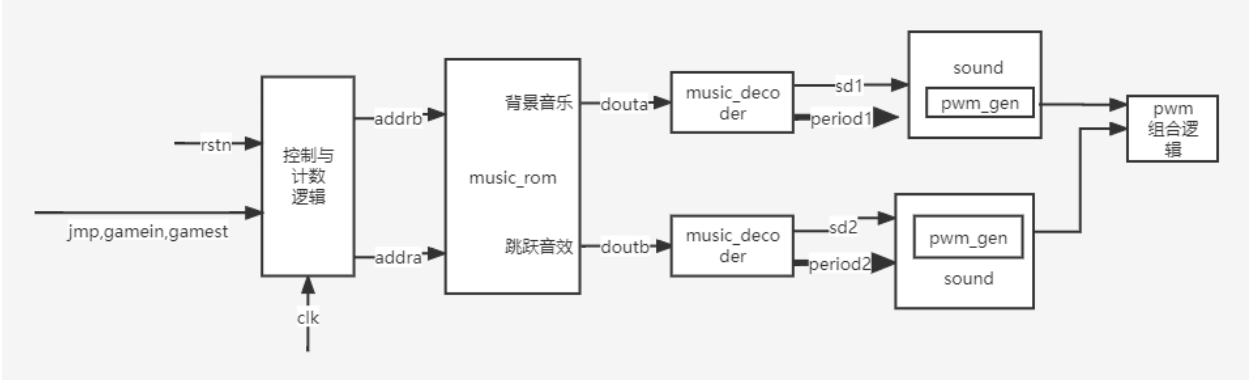
使用与障碍物移动相同的分频后时钟信号（实际可以根据需要调整），每隔一段时间，对背景的起始坐标bk_x_coordinate进行循环位移，并将其加入raddr_bk的计算中。当b_x_coordinate+bk_x_coordinate>199，意味着raddr读取到了下一行，应当循环读取，所以对raddr减200。（200为背景宽度）

音频输出模块：

实现要求：

实现音频输出模块，可以播放背景音乐和音效，选用背景音乐为Fat Rat和sound of silent

逻辑图：



实现原理：

```
rf1=[0, 0, 0, 17, 26, 26, 26, 25, 23, 23, 23, 22, 23, 23, 23, 23, 0, 0, 0, 27, 26, 26, 26, 25, 23, 23, 23, 22, 23, 23, 23, 23, 0, 0, 0, 27, 26, 26, 26,
     25, 23, 23, 23, 22,      23, 23, 23, 23,     25, 25, 25, 25,     26, 26, 26, 26,     27, 27, 27, 27]
rf2=[26, 26, 26, 26, 0, 0, 0, 27, 26, 26, 26, 25, 23, 23, 23, 22, 23, 23, 23, 23, 0, 0, 0, 27, 26, 26, 26, 25, 23, 23, 23, 22, 23, 23, 23, 23, 0, 0, 0, 23, 25, 25, 25, 26,
     27, 27, 27, 32,          33, 33, 33, 33,    32, 32, 32, 32,     33, 33, 33, 33,    35, 35, 35, 35]
rf3=[36, 36, 36, 36, 0, 0, 0, 37, 36, 36, 36, 35, 33, 33, 33, 32, 33, 33, 33, 33, 0, 0, 0, 37, 36, 36, 36, 35, 33, 33, 33, 32, 33, 33, 33, 33, 0, 0, 0,
     37, 36, 36, 36, 35, 33, 33, 33, 32, 33, 33, 33, 33,     35, 35, 35, 35,     36, 36, 36, 36,    37, 37, 37, 37]
rf4=[36, 36, 36, 36, 0, 0, 0, 37,     36, 36, 36, 35,     33, 33, 33, 32,     33, 33, 33, 33, 0, 0, 0, 37, 36, 36, 36, 35,     33, 33, 33, 32, 33, 33, 33, 33,    0, 0, 0, 33,
     35, 35, 35, 36,    37, 37, 37, 42,       43, 43, 43, 43, 42, 42, 42, 42,    43, 43, 43, 43,    45, 45, 45, 45]
rf5=[46, 46, 46, 46, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 22,
     23, 23, 23, 25,     23, 23, 23, 23, 23, 23, 23, 23,     17, 17, 17, 22,     23, 23, 23, 23,    23, 23, 23, 23,     23, 23, 23, 23,    23, 23, 23, 23, 23, 23, 23, 22]
rf6=[23, 23, 23, 33, 32, 32, 32, 32,    27, 27, 27, 27,    26, 26, 26, 26,     27, 27, 27, 27,    27, 27, 27, 27,    27, 27, 27, 27,    27, 27, 27, 26,
     27, 27, 27, 32,    27, 27, 27, 27,    26, 26, 26, 26,     25, 25, 25, 25, 26,          27, 27, 27, 27,    27, 27, 27, 27,    27, 27, 27, 27,    27, 27, 27, 26]
rf7=[27, 27, 27, 32,     27, 27, 27, 27,    26, 26, 26, 26,    25, 25, 25, 25,    23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 22,
     23, 23, 23, 25, 23, 23, 23, 23, 22, 22, 22, 22,    17, 17, 17, 22,    23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 23,    23, 23, 23, 22]
rf8=[23, 23, 23, 33, 32, 32, 32, 32,    32, 32, 32, 32, 27, 27, 27, 27, 27, 27, 26, 26, 26, 26,     27, 27, 27, 27, 27, 27, 27, 27,
     27, 27, 27, 26, 25, 25, 25, 26,     27, 27, 27, 27,     27, 27, 27, 27, 27,     27, 27, 27, 26,     25, 25, 25, 26]
rf9=[27, 27, 27, 27,  27, 27, 27, 27,  27, 27, 27, 27,  27, 27, 27, 27,  22, 22, 22, 22,  22, 22, 22, 22,  22, 22, 22, 22,  22, 22, 22, 22]
```

```
In  [1]:
dic={11:"00001",12:"00010"}
dic[13]="00011"
dic[14]="00100"
dic[15]="00101"
dic[16]="00110"
dic[17]="00111"
dic[21]="01000"
dic[22]="01001"
dic[23]="01010"
dic[22]="01001"
dic[24]="01011"
dic[25]="01100"
dic[26]="01101"
dic[27]="01110"
dic[31]="01111"
dic[32]="10000"
dic[33]="10001"
dic[34]="10010"
dic[35]="10011"
dic[36]="10100"
dic[37]="10101"
dic[38]="10110"
dic[39]="10111"
dic[41]="11000"
dic[42]="11001"
dic[43]="11010"
dic[44]="11011"
dic[45]="11100"
dic[46]="11101"
dic[47]="11110"
dic[0]="00000"
```

核心代码

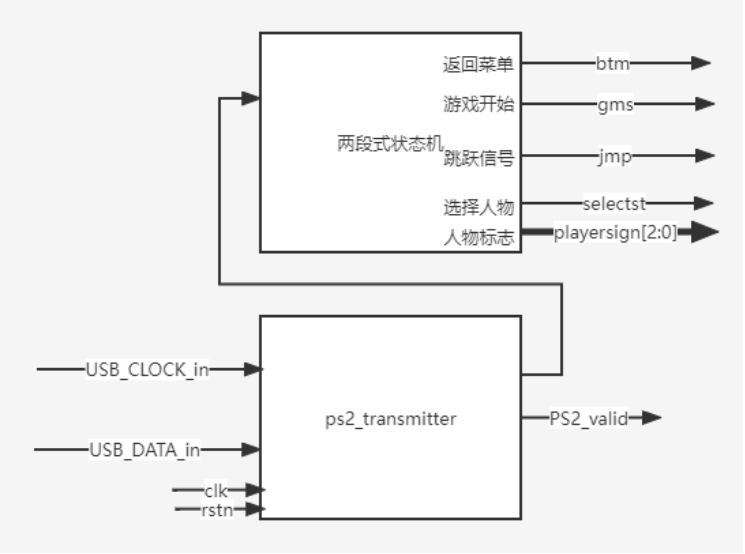
```
module pwm_gen (//占空比h_time/period,调频模块
  input [11:0] period,//512clk~Vdd
  input [11:0] h_time,
  input clk,
  output reg wave,pulse
);
   reg [11:0] cnt=0;
   always @(posedge clk) begin
       cnt<=cnt+1;
       pulse<=0;
       if(cnt>period)begin
           cnt<=0;
           pulse<=1;
       end else if(cnt<{h_time>>2})begin//减3/4音量
           wave<=1;
       end else begin
           wave<=0;
       end
   end
endmodule

module sound (//放声模块
input [11:0] period,
input clk,
output pwm
);
endmodule

module audio#(
   parameter speed = 500_0000,effect_speed=500_0000
)
(//音频模块
   input clk,jump,gamein,gamest,//gamest是脉冲信号，gamein是状态信号
   input rstn,
   output  reg pwm,
   output  sd
   // output [11:0] note
   );
endmodule
```

键盘输入模块

实现要求：

> 实现INPUT模块，接受来自键盘的输入，实现控制游戏状态的功能。

逻辑图：



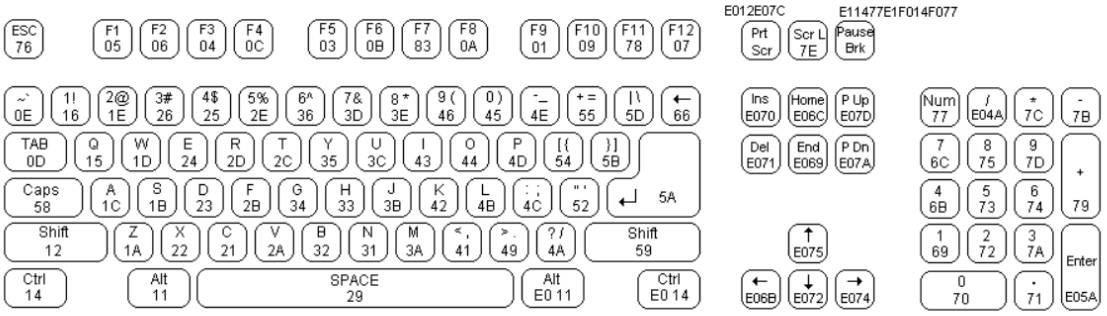
实现原理：

A. List of key codes

Graphic keys-to-keycodes map

Key names are on top, with the hexadecimal make code below.

Figure A.1. Keycode map



核心代码：

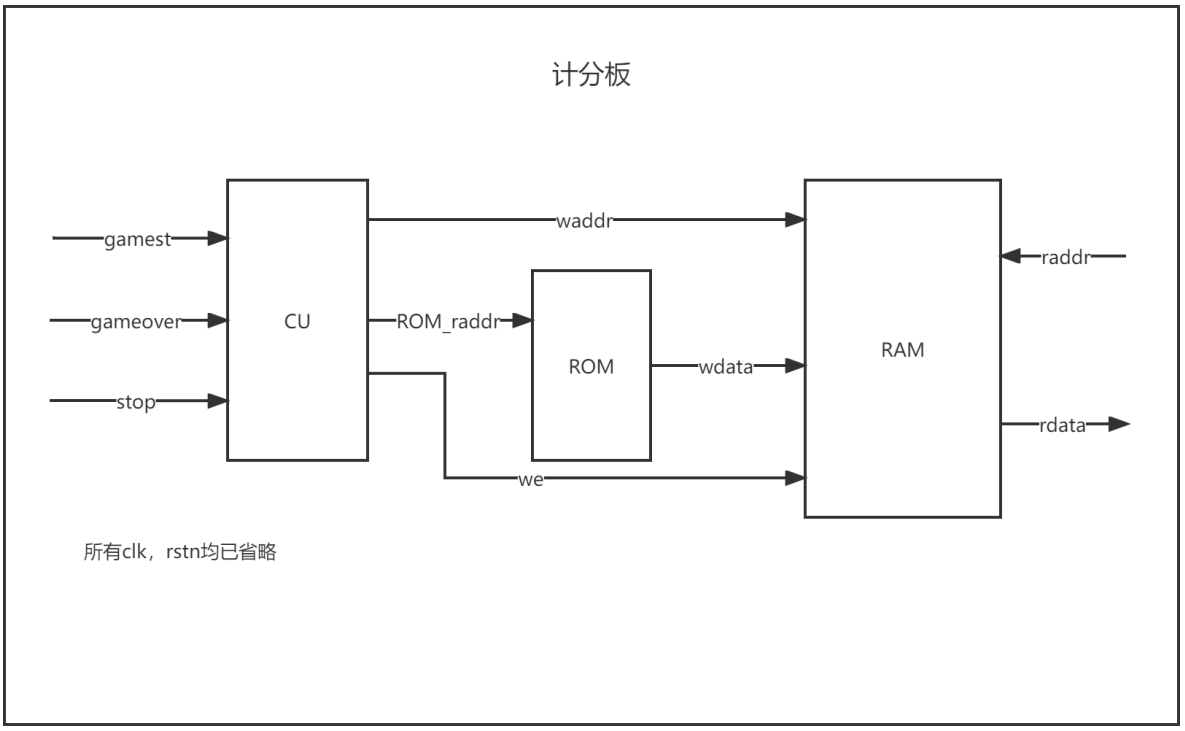
```
module INPUT(
    input          rstn,clk,gamein,
    output reg btm=0,gms=0,jmp=0,stp=0,selectst=0,
    output reg [2:0] playersign=0,
    // USB port
    inout             USB_CLOCK,
    inout             USB_DATA
);
endmodule
module ps2_transmitter (
    input            clk,
    input            rstn,

    // ports for input data
    input            clock_in,           // connected to usb clock input signal
    input            serial_data_in,     // connected to usb data input signal
    output reg [7:0] parallel_data_in,   // 8-bit input data buffer, from the USB interface
    output reg       parallel_data_valid,// indicate the input data is ready or not
    output reg       data_in_error,      // reading error when the odd parity is not matched

    // ports for output date
    output reg       clock_out,           // connected to usb clock output signal
    output reg       serial_data_out,     // connected to usb data output signal
    input      [7:0] parallel_data_out,   // 8-bit output data buffer, to the USB interface
    input            parallel_data_enable,// control signal to start a writing process
    output reg       data_out_complete,

    output reg       busy,                // indicate the transmitter is busy
    output reg       clock_output_oe,     // clock output enable
    output reg       data_output_oe       // data output enable

);
```

计分板模块

实现要求：

1. 实现一个47×7的长条形计分板，分数以“SCORE：xxxxx”的形式显示在计分板（RAM）上，外界输入raddr，得到对应位置的rdata（颜色）。
2. 计分板的分数更新由三个输入gamest、gameover和stop以及时间决定。复位后计分板处于停止计时状态，gamest有效后清零并开始计时，此后在正常计时状态下每0.1秒分数+1；stop有效时暂停计时，直至stop无效；gameover有效后停止计时，直至收到gamest。

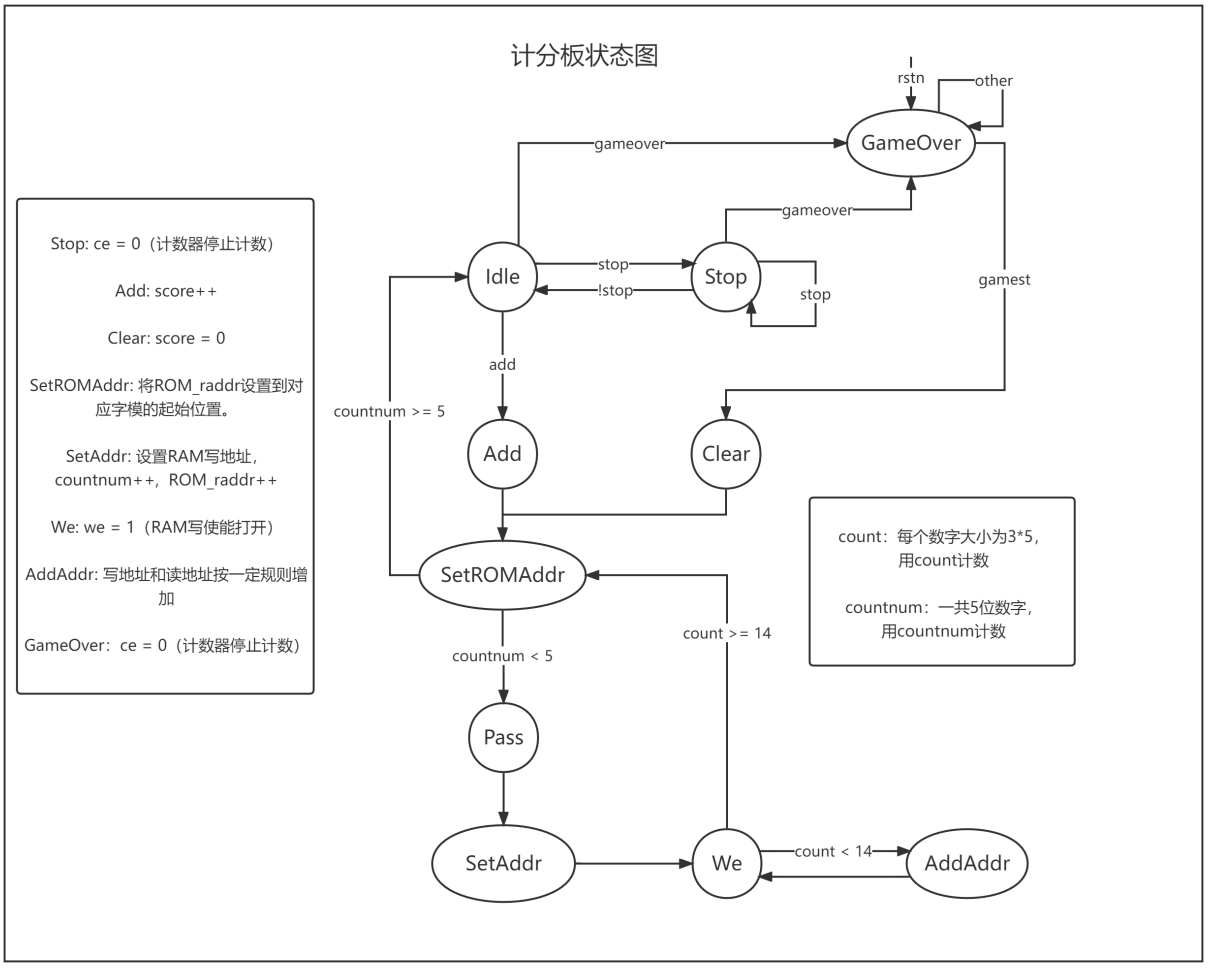
逻辑图：



实现原理：

如逻辑图所示，每个数字的字模被存储在ROM中。CU接受gamest（游戏开始）、gameover（游戏结束）和stop（游戏暂停）三种信号，根据收到的不同信号决定是否继续计分/清空当前分数，并且在每次分数发生改变时都刷新RAM，从ROM中读取对应的字模写到RAM的对应位置。

状态转换图：



每个状态的作用：

> GameOver：游戏结束状态，计分板停止计分。
> Idle：空状态，用于Add、Stop、GameOver状态间的转换。
> Stop：暂停游戏，暂停计分。
> Add：分数+1。
> Clear：清空分数。
> SetROMAddr：设置ROM的起始读地址（选择字模）。
> Pass：空状态，用于延迟一个时钟周期。
> SetAddr：设置写地址等。
> We：打开写使能。
> AddAddr：写和读地址按一定规则增加，写入同一字模的下一个位置。

注意到图中将SetROMAddr（设置ROM的读地址）状态单独设置，并且相较SetAddr提前了2个周期。这是因为控制信号的产生是同步的，因此ROM的读地址在上升沿产生，ROM在下一个上升沿才能检测到读地址的改变，再下一个上升沿才能给出有效的输出，一共延迟2个周期。因此将SetROMAddr提前2个周期，使读出的字模和写使能信号同步。

核心代码：

```
//计分板的画布大小为47*7，用大小为512的RAM存储
module ScoreBoardCU(
    input clk, rstn,
    input gamest, gameover, stop,
    input [8:0] raddr,
    output [11:0] rdata
    );

    wire [11:0] wdata;  //wdata 连接ROM的输出和RAM的输入

    reg [23:0] CNT;     //计数器，计算什么时候需要加分
    reg [19:0] score;   //分数，按BCD码存储
    wire [8:0] waddr;    //RAM的写地址
    reg [7:0] ROM_raddr;//ROM的读地址
    reg [3:0] count, tempScore;    //tempScore表示当前正在写入哪个数字
    reg [2:0] countnum;            //countnum和上面的count是在写入RAM过程中的计数变量，在状态图中已解释过。
    reg [3:0] State;    //State of the State machine
    reg we, ce, add, clear, clearCNT;    //we for RAM, ce for CNT, add for score

    reg [5:0] offset;   //横坐标的偏移量
    reg [2:0] x, y;     //x和y是RAM上面的横、纵坐标

    parameter Idle = 0, Stop = 1, Add = 2, Clear = 3, SetROMAddr = 4, Pass = 5, SetAddr = 6, We = 7, AddAdd
r = 8, GameOver = 9;
    parameter T = 10000000;

    always @(posedge clk) begin//计数器，每计到0.1秒（10000000周期）即发出add信号并重置。
        if(!rstn)   begin
            CNT <= T;
            add <= 1'b0;
        end
        else if(clearCNT)   begin
            CNT <= T;
            add <= 1'b0;
        end
        else if(ce) begin
            if(CNT == 0)    begin
                CNT <= T;
                add <= 1'b1;
            end
            else    begin
                CNT <= CNT - 1;
                add <= 1'b0;
            end
        end
        else    add <= 1'b0;
    end

    always @(*) begin//维护tempScore
        case (countnum)
            3'd0:   tempScore = score[19:16];
            3'd1:   tempScore = score[15:12];
            3'd2:   tempScore = score[11:8];
            3'd3:   tempScore = score[7:4];
            3'd4:   tempScore = score[3:0];
            default: tempScore = 4'b0;
        endcase
    end

    always @(posedge clk) begin//维护分数
        if(!rstn)   score <= 20'b0;
        else if(clear)  score <= 20'b0;
        else if(add)    begin
            //这里是按8421BCD码规则让score自增的部分，若score已达到99999则不变。过于冗长，因此省略
        end
    end

    always @(posedge clk) begin//维护状态
        //按状态转换图更新状态，过于冗长，省略
    end

    always @(posedge clk) begin//给出每个状态对应的控制信号
        we <= 1'b0; ce <= 1'b1; clear <= 1'b0; clearCNT <= 1'b0;
        case (State)
```

```
            Stop:   ce <= 1'b0;
            Add:    countnum <= 3'b0;
            Clear:  begin
                countnum <= 3'b0;
                clear <= 1'b1;
                clearCNT <= 1'b1;
            end
            SetROMAddr:    begin
                count <= 4'b0;
                case (tempScore)//选取字模
                    4'd0:   ROM_raddr <= 8'd0;
                    4'd1:   ROM_raddr <= 8'd16;
                    4'd2:   ROM_raddr <= 8'd32;
                    4'd3:   ROM_raddr <= 8'd48;
                    4'd4:   ROM_raddr <= 8'd64;
                    4'd5:   ROM_raddr <= 8'd80;
                    4'd6:   ROM_raddr <= 8'd96;
                    4'd7:   ROM_raddr <= 8'd112;
                    4'd8:   ROM_raddr <= 8'd128;
                    4'd9:   ROM_raddr <= 8'd144;
                    default:    ROM_raddr <= 8'd0;
                endcase
            end
            SetAddr:       begin//设置写地址，同时增加读地址
                x <= 3'b0;
                y <= 3'b1;
                countnum <= countnum + 3'b1;
                offset <= countnum * 3'd4 + 8'd27;
                ROM_raddr <= ROM_raddr + 8'b1;
            end
            We:            begin
                we <= 1'b1;
                count <= count + 4'b1;
            end
            AddAddr:       begin//写同一位数字中的下一个地址
                ROM_raddr <= ROM_raddr + 8'b1;
                if(x < 3'd2)    x <= x + 3'b1;
                else    begin
                    y <= y + 3'b1;
                    x <= 0;
                end
            end
            GameOver:   ce <= 1'b0;
        endcase
    end

    assign waddr = y * 6'd47 + x + offset;//根据横纵坐标和偏移量合成写地址

    ROM NUM_ROM(
        .addra (ROM_raddr),
        .clka (clk),
        .douta (wdata)
    );

    RAM ScoreBoardRAM(
        //write ports
        .addra (waddr),
        .clka (clk),
        .dina (wdata),
        .wea (we),
        //read ports
        .addrb (raddr),
        .clkb (clk),
        .doutb (rdata)
    );
endmodule
```

仿真结果与分析

动画控制与动画显示模块

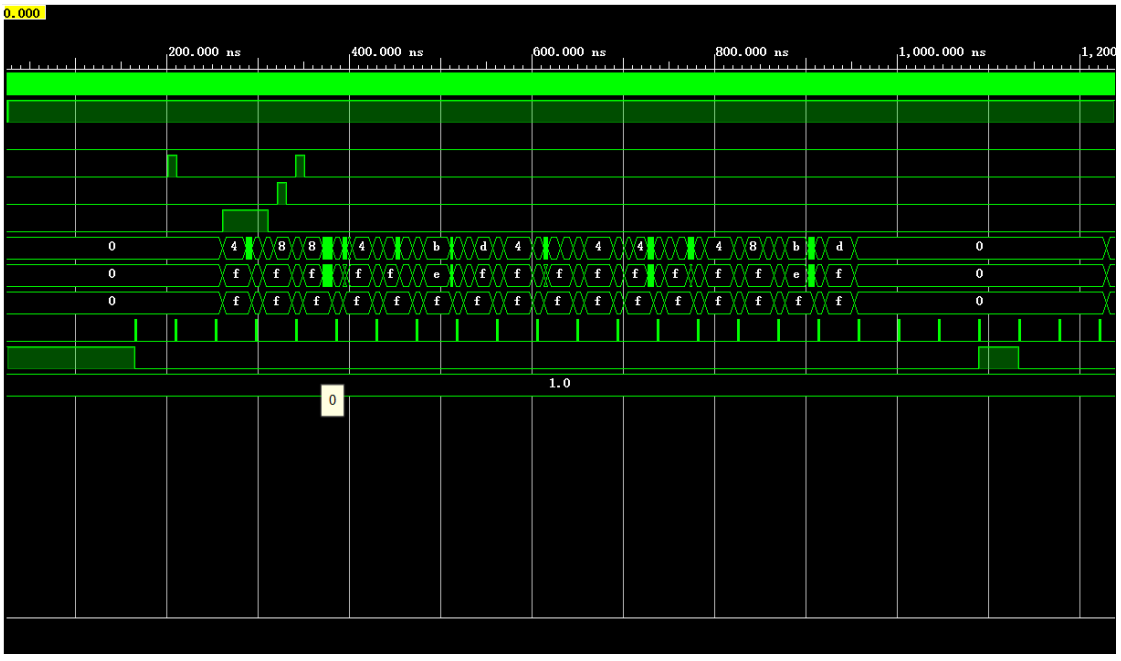
仿真文件代码

```
module PNT_tb;
    //参数
    parameter Period = 1.0;

    parameter TimePerFrame = 5;
    parameter JumpTime = 4;
    parameter RollTime = 2;

    parameter H_LEN = 4;
    parameter V_LEN = 4;

    parameter HSW_t = 0;
    parameter HBP_t = 1;
    parameter HEN_t = 15;
    parameter HFP_t = 2;

    parameter VSW_t = 0;
    parameter VBP_t = 1;
    parameter VEN_t = 15;
    parameter VFP_t = 2;

    //变化信号
    reg clk;
    reg rstn;
    reg bk_to_menu;
    reg gamest;
    reg stop;
    reg jump;

    //输出信号
    wire [3:0] red;
    wire [3:0] green;
    wire [3:0] blue;

    wire hs;
    wire vs;


PNT #(
    .TimePerFrame       (TimePerFrame),
    .JumpTime           (JumpTime),
    .RollTime           (RollTime),

    .H_LEN              (H_LEN),
    .V_LEN              (V_LEN),

    .HSW_t              (HSW_t),
    .HBP_t              (HBP_t),
    .HEN_t              (HEN_t),
    .HFP_t              (HFP_t),

    .VSW_t              (VSW_t),
    .VBP_t              (VBP_t),
    .VEN_t              (VEN_t),
    .VFP_t              (VFP_t)
)pnt (
    .clk            (clk),
    .rstn           (rstn),
    .bk_to_menu     (bk_to_menu),
    .gamest         (gamest),
    .stop           (stop),
    .jump           (jump),

    .red            (red),
    .green          (green),
    .blue           (blue),

    .hs             (hs),
    .vs             (vs)
);

    initial begin
        clk=1;
        forever begin
```

```
            #(Period/2) clk=~clk;
        end
    end

    initial begin
        rstn=0;
        bk_to_menu=0;gamest=0;stop=0;jump=0;
        #(Period);
        rstn=1;
        #(Period*125);
        gamest=1;
        #(Period*10);
        gamest=0;
        #(Period*50);
        jump=1;
        #(Period*50);
        jump=0;
        #(Period*10);
        stop=1;
        #(Period*10);
        stop=0;
        #(Period*10);
        gamest=1;
        #(Period*10);
        gamest=0;

    end
endmodule
```

仿真结果截图

利用参数化的方式，模拟16*16的分辨率屏幕下，对各种模拟信号的变化。
仿真结果如下图：



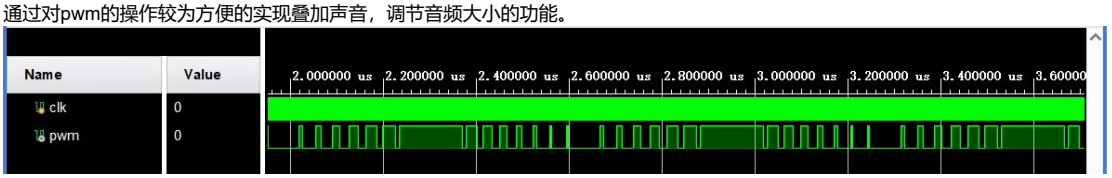
音频输出模块：

仿真文件代码：

```
`timescale 1ns / 1ps

module audio_tb(
            );
    reg clk=0;
    always @(*) begin
        forever begin
            #2;
            clk=~clk;
        end
    end
    sound sd(
        /*
        input [11:0] period,
        input clk,
        output pwm
        */
        .period(7),
        .clk(clk),
        .pwm(pwm)
    );
endmodule
```

仿真结果截图：

示例period=7,pwm占空比越高，表示当前音频输出电压越高，sound模块通过调节period来产生不同频率的正弦波，这些正弦波可以通过对pwm的操作较为方便的实现叠加声音，调节音频大小的功能。



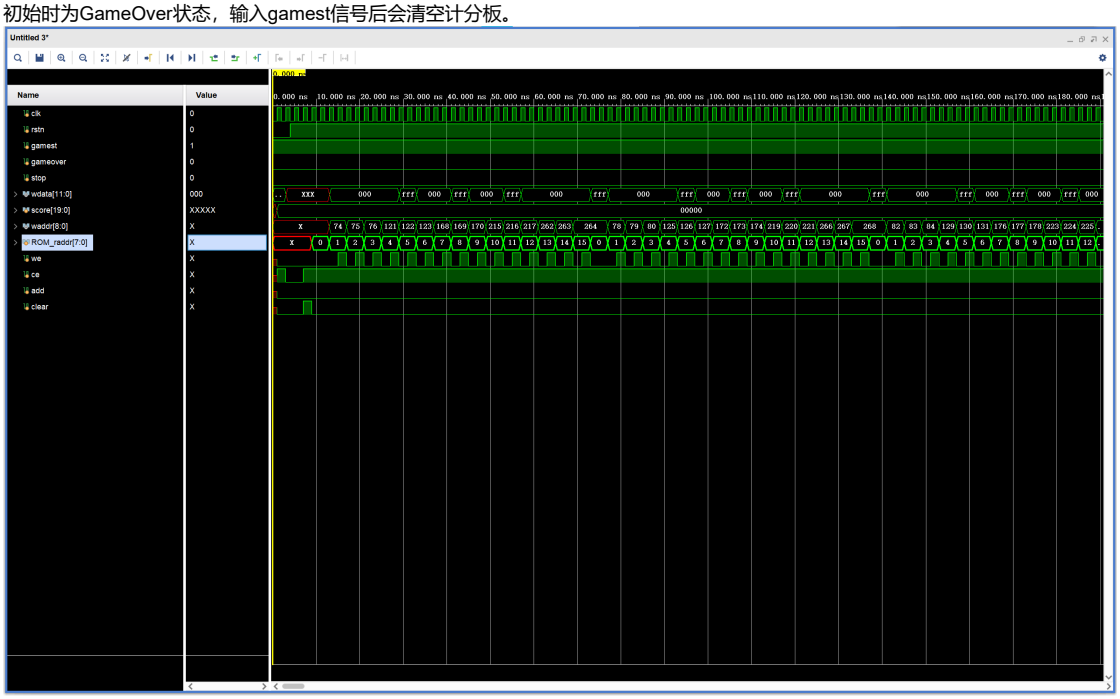
计分板模块：

仿真文件代码：

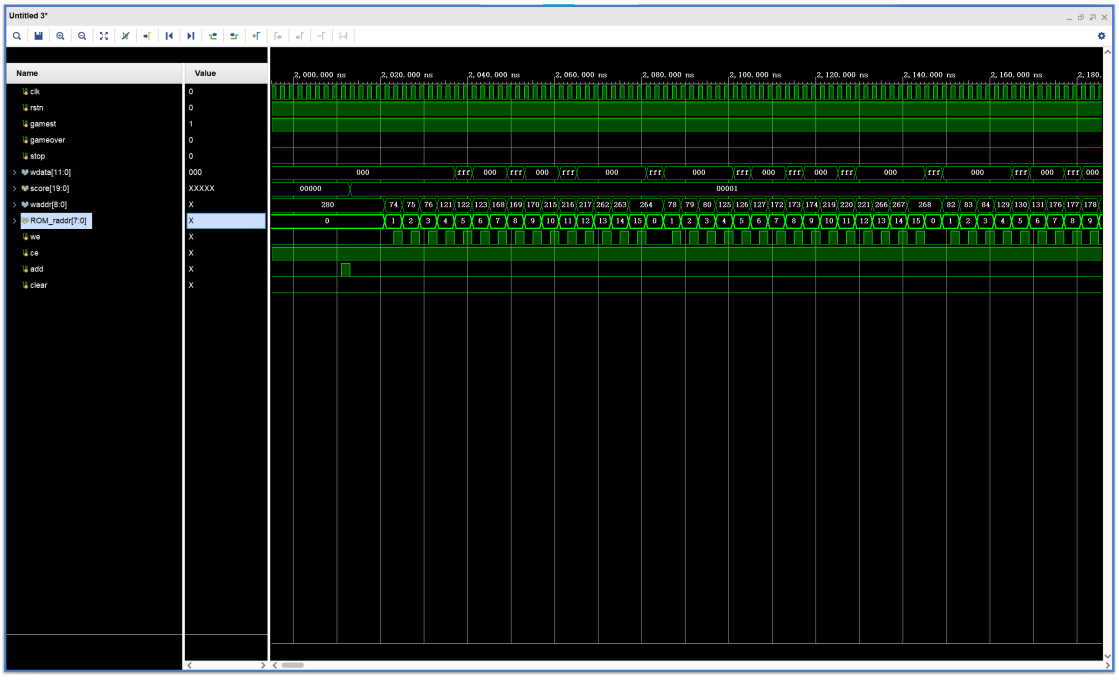
```
module CU_tb();
reg clk, rstn, gamest, gameover, stop;
ScoreBoardCU ScoreBoardCU_dut (
  .clk (clk),
  .rstn (rstn),
  .gamest (gamest),
  .gameover (gameover),
  .stop (stop)
);

initial begin
    gamest = 1;
    gameover = 0;
    stop = 0;
    clk = 0;
    rstn = 0;
    #4 rstn = 1;
end

always #1 clk = ~clk;
endmodule
```

仿真结果截图：

初始时为GameOver状态，输入gamest信号后会清空计分板。



当add信号到来时，score会改变，并且刷新RAM。

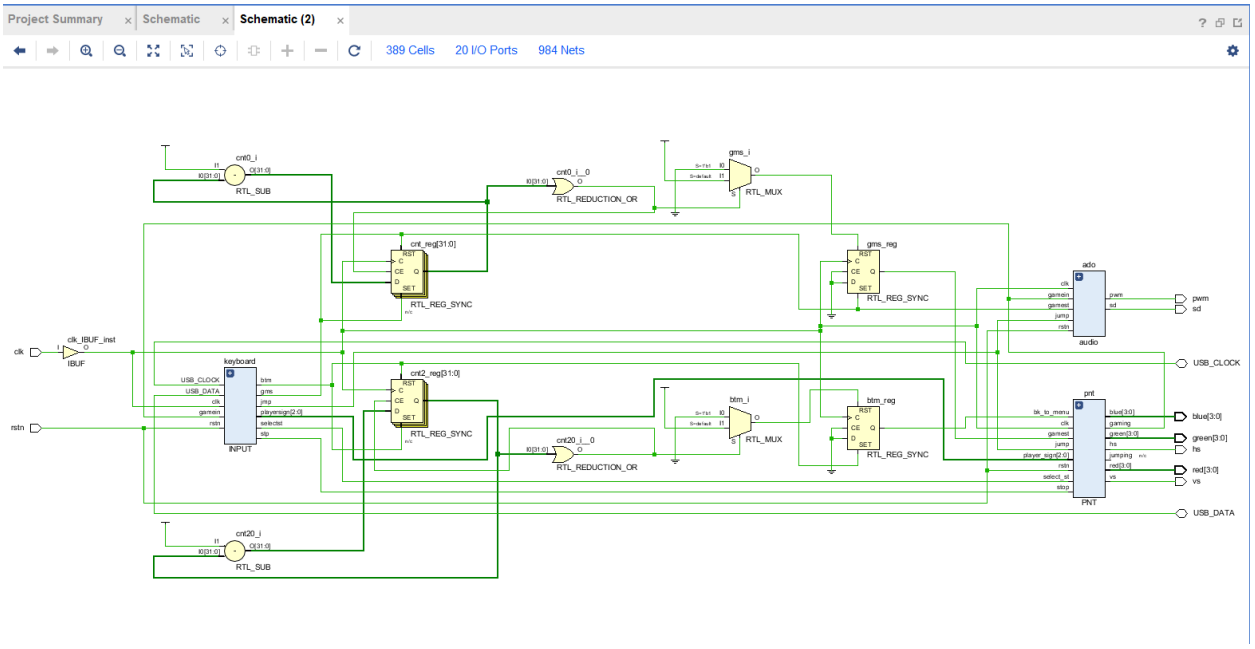


电路设计与分析

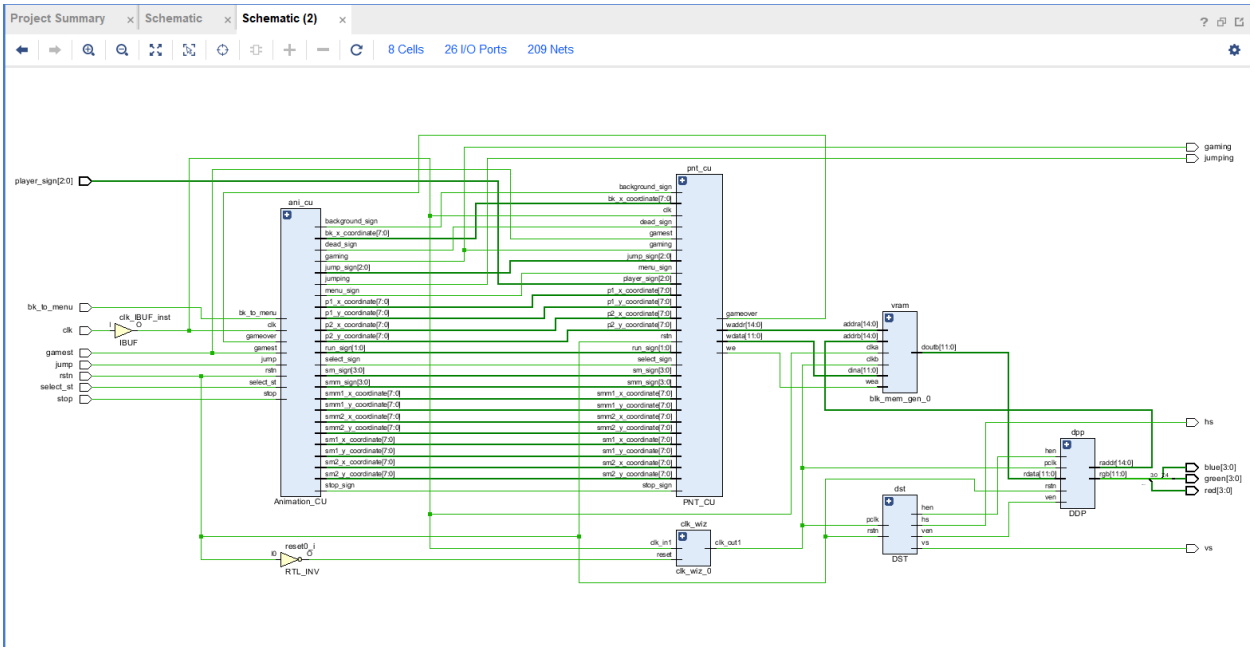
RTL电路图：

> *看看就好*

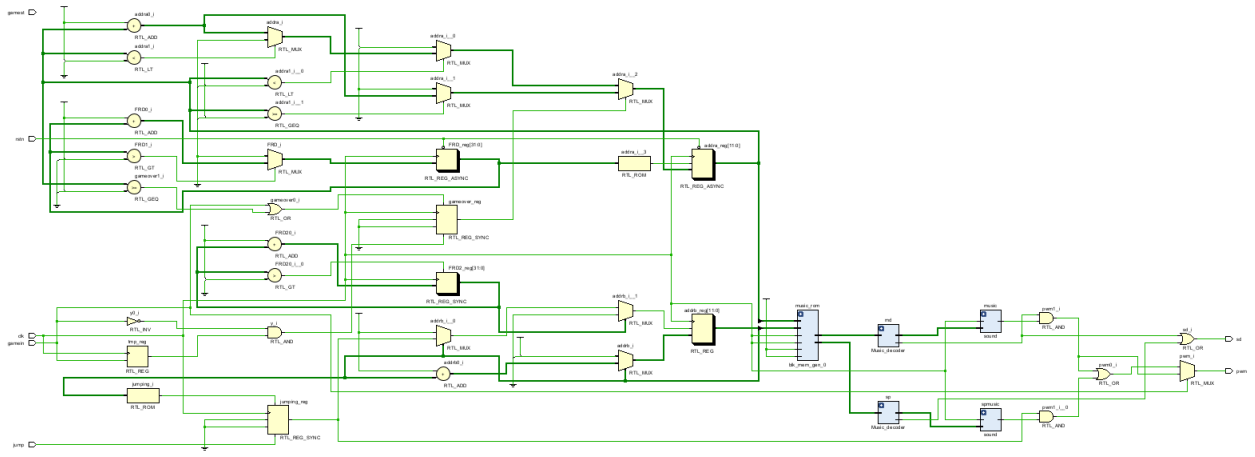
顶层模块



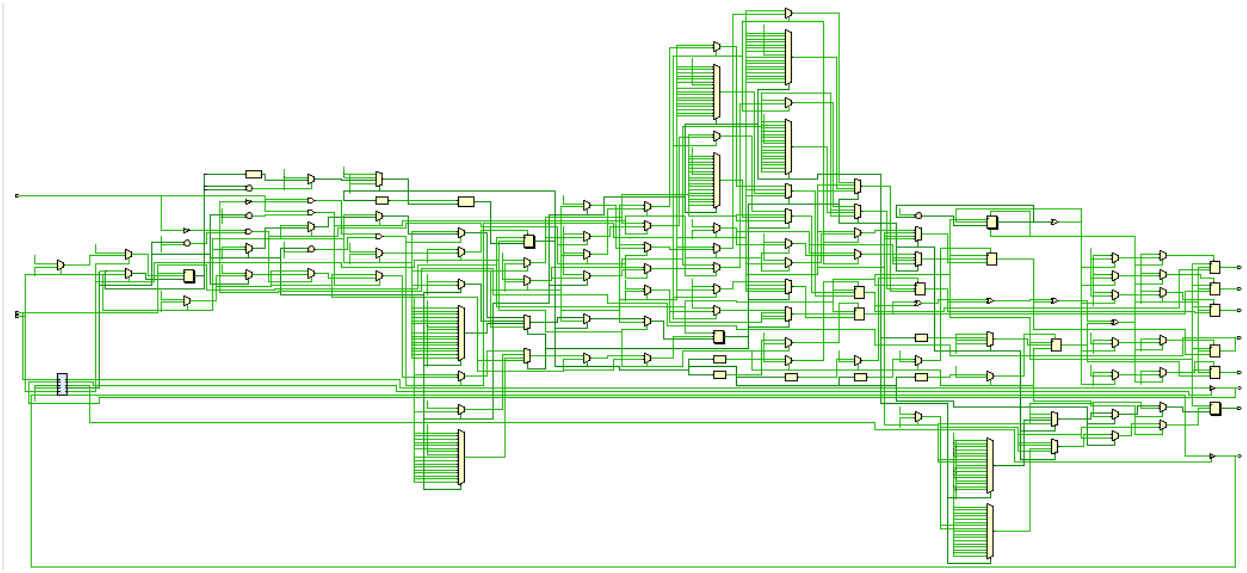
动画控制与动画显示模块



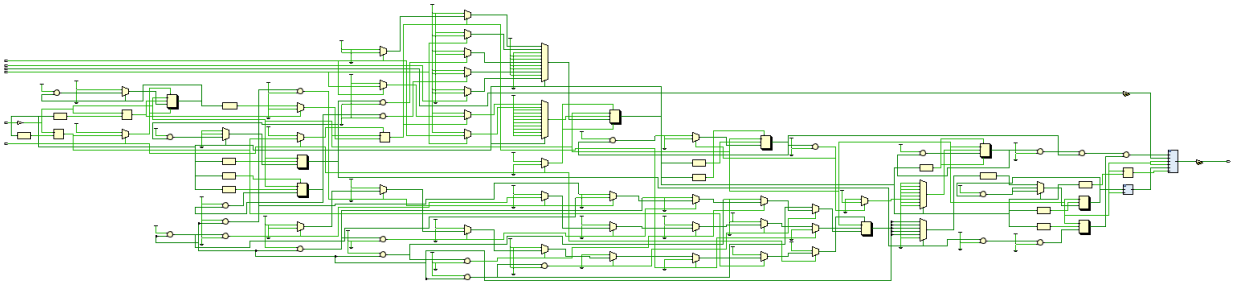
音频输出模块



键盘输入模块



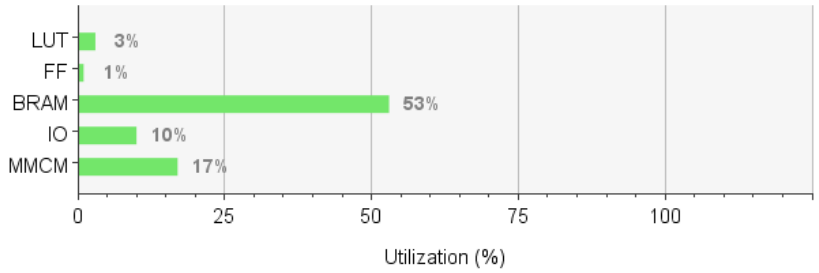
计分板模块



资源使用情况：

资源使用量

Resource	Utilization	Available	Utilization %
LUT	2157	63400	3.40
FF	896	126800	0.71
BRAM	71	135	52.59
IO	20	210	9.52
MMCM	1	6	16.67



时序

时钟为100MHz

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.616 ns	Worst Hold Slack (WHS):	0.098 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2976	Total Number of Endpoints:	2976	Total Number of Endpoints:	997

All user specified timing constraints are met.

测试结果与分析

> *在答辩时已经放过展示视频，就不再贴图片了*

动画控制与动画显示模块

经测试，可以完成显示和碰撞反馈等各种功能。中途曾出现过怪物贴图显示异常等问题，在对实现进行优化，改进了竞争-冒险问题后得到解决。

音频输出模块：

播放永恒的东风，可以听出是电音风味，这是因为输出波形为方波，故在边缘有突变且不连续。但是反用电音风味实现本来就是电音的Fat Rat音乐就可以感到毫无违和感。

键盘输入模块：

INPUT模块需要注意传出的信号需要延续几个周期，便于后续模块读取，若只传送一个周期的信号会造成异常。同时控制游戏状态采用状态机便于排查异常，是程序更加易读。

计分板模块：

单独接上显示模块进行测试，根据不同的输入可以给出对应的响应（如gamest后从0开始计分，stop时暂停，gameover之后停止计分等），满足要求。

感想

徐航宇：

> 不知不觉又一学期过去了，虽然不得不说，进阶班的任务很重很累，但是从我的收获来看，还是很值得的。我对课内知识的理解更深刻了，硬件设计思想和方法更成熟了，也对计算机有了一个更深刻的认识。而我最大的收获，莫过于下面这个：
>
> 一直以来，我都极度偏爱理论的、抽象的东西，相应的，对硬件或是偏底层的东西总是有着很强烈的抵触情绪。我总觉得它们太过繁琐而无用、公式化而无创造性，缺乏美感。但是后来，抱着想要更全面地了解计算机与计算机科学的心态，我尝试选择了进阶班。在与fpga打了一个学期的交道后，我渐渐地改变了对硬件的看法。硬件不完全是繁琐而无用的，它也可以很有趣很美丽。并行和流水线让我体会到，在硬件层次，除了优化复杂度以外，还可以有别的优化方法；一些看起来无法实现的东西，比如游戏等，也可以通过精巧的设计，得到完美的实现；仿真综合无数遍也死活找不出来、发现问题关键时又如柳暗花明又一村的bug们也让人痛并快乐着（雾）……尤其是最后的综合实验，当我们用看似笨拙的硬件级语言，一步步搭建、抽象出了一套通用的2D游戏设计方案，那种理论与实际相结合的感觉，真的令我无比陶醉。总之，如今，虽然远远不能说比起理论更喜欢硬件，但至少我已不再排斥，已经有了一点兴趣了，也对以后的课程有了更多的期待。这也许就是对我来说最大的收获了。

闫泽轩：

> 提高班的教学在一开始总是给我一种强烈的"fly!bitch"的感受——即什么都不知道，或知道的很少，最后要做出来一个对于没有做过的人是很amazing的组合/时序逻辑所对应的FPGA编程，是非常困难的。我还记得第一次课的加法器，在张老师讲完后，我看到PPT里有一段代码，心里十分得意：不就是把这段代码封装一下，上板跑个流程嘛，于是没有过多长时间就从教室润了。然后去图书馆自信半个小时就可以上板+写报告了，但是在图书馆枯坐了两个小时我才刚弄明白我要干什么，这个东西的难点在哪里——弄清楚一次Lab的困难也是比较困难的。
>
> 但是还好，第一次实验有两周。但是很快后面就有新的感触了。
>
> 做到加法器的时候，竟然有同学将每一种加法器的实现都实现并进行了比较，当做到寄存器堆的时候，我目睹马助教在我旁边“顺手”就写好并且给出了.BIT文件，原来人与人之间是有差距的。更进一步的，在VGA的实验时，坐在我旁边的同学在一下午的时间内就迅速的完成了——我吃晚饭前他还在调白色屏幕，吃完饭回来他已经彩色屏幕也调好了，在我调该死的绿屏闪烁等等问题时——他已经写完了。
>
> 世界上有以一当十的将军，也有以一当千的程序员。
>
> Linux之父林纳斯，一己之力开发并维护了如今最伟大的操作系统，在使用Linux系统之前我觉得windows系统作为巨无霸大概没有什么对手了，但是在使用Linux和逐渐了解林纳斯后，我才逐渐了解这样一个伟大的巨人，真正的是凭一己之力维护了开源世界的标杆，开发了版本管理的标准工具，乃至更多！
>
> 科大培养的优秀校友中，有不少以一当千的院士，但是显然我不是其中的任何一个。
>
> 不过从另一个方面来说，被智力因素打败，或者承认技不如人，智不如人而落后于竞争者，目睹卷王卷到最后应有尽有，对我来说要比被金钱，家境或者国籍，年龄因素打败要公平得多，即使在这场竞争中我不是胜者，而且这样的相对令我心服口服的竞争只有在科大才能参与。
>
> 在提高班中，每一个Lab都伴随着痛苦，随后才是喜悦。在调试代码的过程中，不断一遍一遍地否定自己，不断一遍一遍的遗忘自己的思路，一遍一遍的重构，反复前后寻找才是常态，但是在这样的过程中，锻炼自己的意志力对我来说已经是收获中的小量，真正的收获也不是一遍一遍的突破自己——相反，我是在这个过程中一遍一遍的触碰到了自己的边界，有的边界只要走过去就可以超越，有的边界就像一堵墙，如果我由于这种边界而遇到了困难，最终我解决了这个困难也不是因为我翻过了这堵墙，而是我走过了其他的边界，这堵墙只是变得更加棱角分明。因此直接的说，我的收获主要是更加清晰的认识了自己，认识自己的不足，认识自己的边界，也认知了自己更多的未认知之处。
>
> 但是我从不后悔加入了这场痛又有一点快乐的旅程中，我要登上自己人生宇宙的月球，not because it is easy,but because it is hard.

李牧龙：

> 这学期是我第一次接触硬件电路。此前编写的程序都是使用C语言或Python等高级语言实现的小程序，且几乎都是单线程。而Verilog写出来的程序却是并行的，与高级语言截然不同。与一条条指令顺序（有时会有循环和分支）执行的C语言相比，Verilog的上手难度显然大了许多。好不容易能体会硬件的并行思想后，时序又接踵而至……在提高班的这大半个学期，几乎每个项目（或许除了LabH3）都是一项巨大的挑战。从如何高效实现一个乘法器到“玄学”调试VGA显示信号，每个星期都历经磨难，却又收获颇丰。除了掌握基本的硬件电路设计能力外，更重要的是，数字电路的实验让我对计算机的底层有了更深刻的理解。尽管目前还无法看到这些知识的实际用处，但脑内的充实感和完成一门课程后的成就感是切实存在的。我也相信，这些看似无用的底层知识，是我们将来迈向尖端必需的踏板。

总结

> 1. 本次实验是Verilog的第七次设计，小组实验的过程中锻炼了团队协作能力。
> 2. 对助教和老师的精心指导表示诚挚的感谢！
> 3. xhy就是神。