浙江大学

期末 Project 报告

课程名称：数字逻辑设计

姓　　名：肖振新

学　　院：竺可桢学院

专　　业：交叉创新平台（计算机+自动化控制）

学　　号：3160104243

指导教师：蔡铭

2018 年 1 月 1 日

基于 FPGA 的小球迷宫游戏

目录

一• 简介

1• 游戏背景介绍

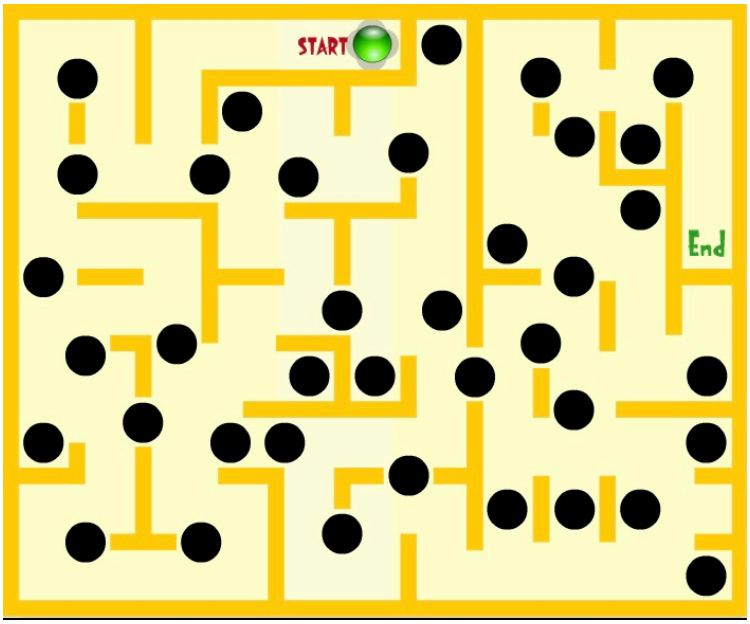
小球迷宫游戏（Labyrinth Ball）是一个简单易玩的体感游戏，它的操作逻辑很简单，就是通过上下左右移动 FPGA 板，控制小球的移动方向，躲避重重的黑洞，并绕过围墙，最终到达终点。这个游戏规则虽然简单，但要成功到达终点，仍需要不小的努力。

2• 游戏整体简介

此游戏采用 vivado2017.4（xilinx 公司的最新发行版）集成开发环境，配合 nexys4 ddr 系列的 FPGA 开发版，编程语言方面，主要采用 verilog 硬件描述语言进行开发（加速度传感器模块使用 VHDL 语言）。

I/O 方面，本游戏使用板载的 ADXL362(ANALOG DEVICES)三轴加速度传感器获取人机交互信息以控制小球的移动，同时配合七段数码管显示当前的 x、y 轴的加速度值，以方便用户把控偏移力度。同时，两个 LED 灯负责指示此时的加速度正负值（x、y 轴正方向为正，负方向为负）。为自定义游戏的难易程度（分为简单 easy、正常 normal 和困难 hard 三类，难度越高小球移动速度越快，越难以控制），本游戏采用了三个 switch 进行控制选择。另外，还使用了一个 switch 控制继续/暂停。另外，为了在游戏结束后（死亡、通关）重启游戏，游戏采用了一个 button 进行重新游戏操作。

同时，为了增强游戏体验，游戏会根据用户倾斜 FPGA 板的角度不同，加快/减慢小球的运动速度，这使得小球更加难以控制，游戏也更富趣味性。



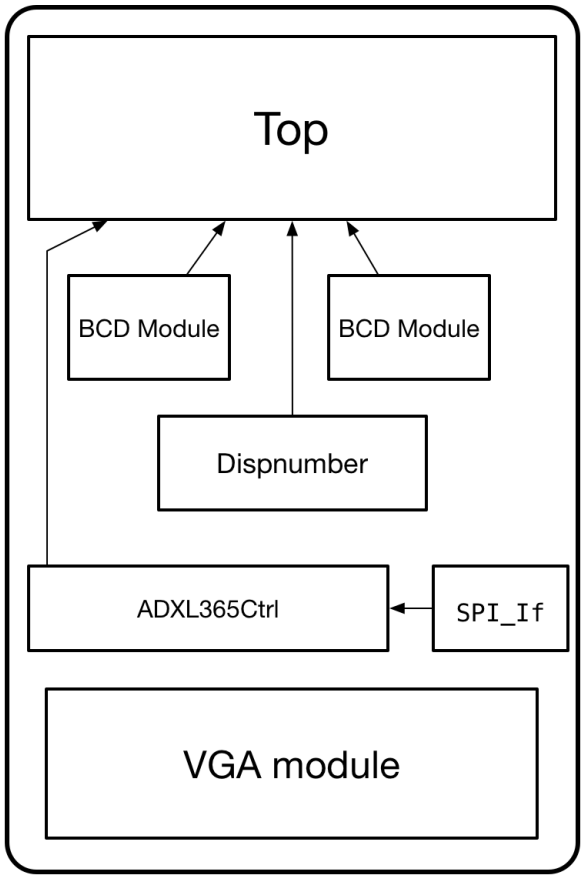
游戏界面

二• 游戏整体架构分析

1• 游戏规则说明

将 bit 文件下载到 FPGA 版后，小球默认在初始位置（start 区）。当控制开始/暂停游戏的 switch(sw[0])拨为 1 后，小球开始移动。用户可以通过上下左右倾斜 FPGA 版，来控制小球在屏幕上的移动。小球无法越过墙这一障碍物前行，同时，如果不慎掉入黑洞，将会死亡，游戏结束。将板上的 switch[1-3]（sw[1]:easy,sw[2]:normal,sw[3]:hard）向上拨，可以控制难易程度（同时只能有一个向上拨）。用户也可以实时地在屏幕右侧的提示栏看到此时的难度（当前难度对应的字符将会变为红色）。如果绕过重重障碍物到达图中标明的 end 区域，那么胜利，游戏结束。不论是在何种情况，用户都可以通过按下板上的中间按钮进行复位操作，此时游戏将会重新开始，小球也会回到对应的初始位置。

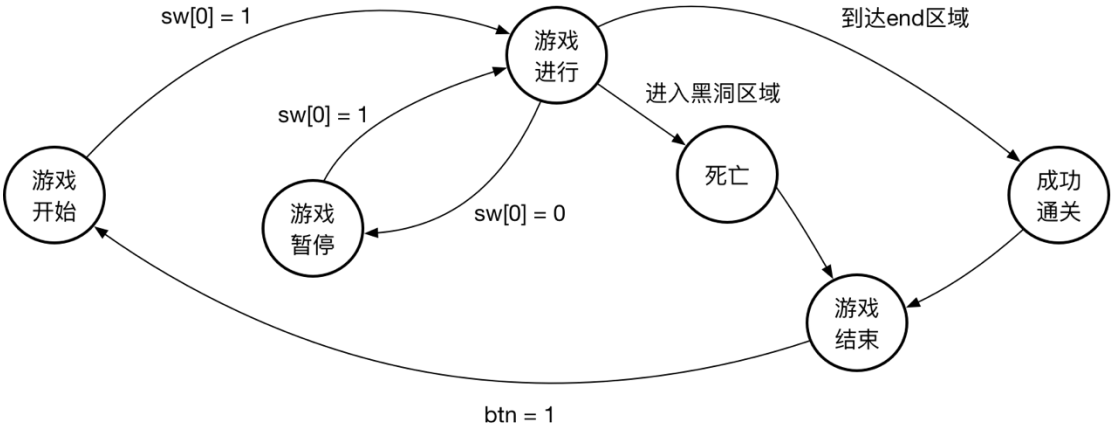
2• 游戏整体架构设计



本游戏的主体部分都设计在了 Top 主模块中,这样方便于各个游戏环节的通信，但是也牺牲了一定的层次逻辑。我在设计游戏时，发现如果设计多个模块，那么在时序电路的通信过程中将会有很多问题，比如会造成各个模块之间有着一个周期时间的落后（D 触发器的工作原理决定的）。

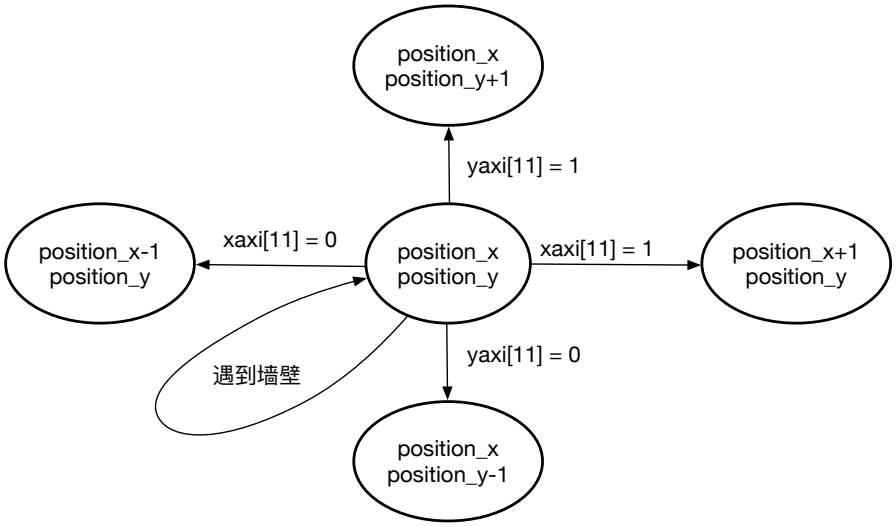
其中，Top 模块为主模块，负责各个子模块之间的连接与通信，同时完成了大部分的游戏逻辑实现。VGA 模块控制显示屏幕的输出，ADXL362Ctrl 模块为三轴加速度传感器的驱动模块，负责实时将得到的数据返回给主模块进行处理。BCD 模块将 x、y 轴的加速度值转化为十进制，并由 dispnumber 模块在七段数码管上进行输出。

3• FSM 有限状态机



如图所示，整体的游戏流程较为简单，初始时处于游戏开始位置，在将 sw[0]开关拨向 1 后，游戏启动。如果 sw[0]被拨向 0，那么游戏暂停，如果小球行进到黑洞区域（有一系列很复杂的判断，将在后续中介绍），那么游戏结束。同时，如果成功到达 end 区域，那么则成功通关，游戏亦结束。在游戏结束后，如果按下 button，那么游戏回到初始状态，小球也会到了 start 位置，从而可以进行新一局的游戏。

在“游戏进行”这一状态下，根据小球位置的不同，又有一个状态机在运行：



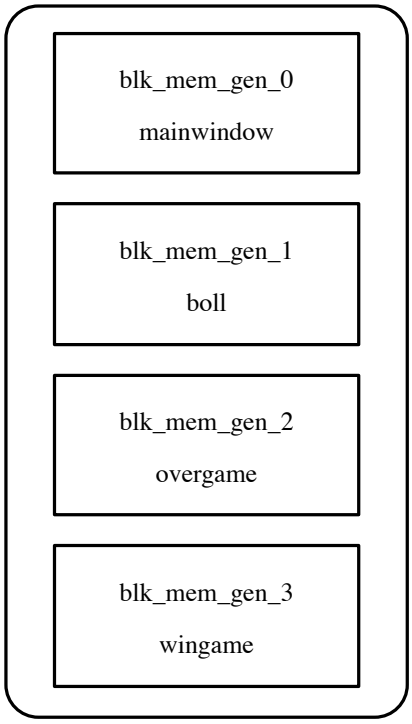
如图，position_x,position_y 为一个十位寄存器，负责存着小球当前位置坐标。在一个周期来临时，判断相应的 xaxi[11],yaxi[11]的值，然后试探相应方向是否是围墙（试探方法将在后续介绍），如果不是，那么就行进一格。如果是围墙，那么停在原地不动（position_x,position_y 的值不更新）。

三• 各模块实现分析

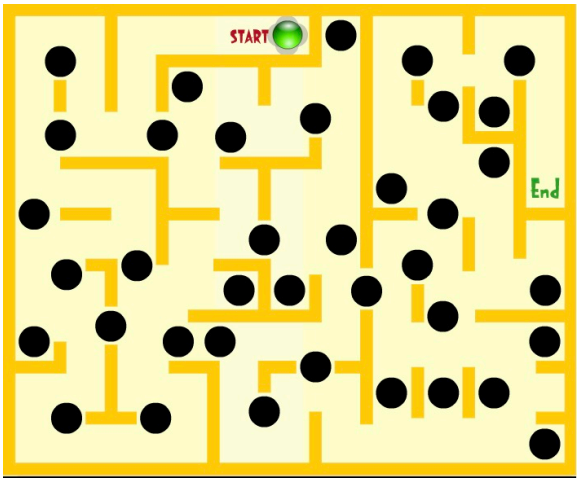
1• Top 主模块

1.1• IP 核介绍（ROM）

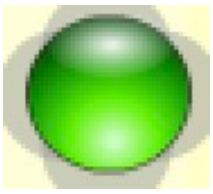
为了在 VGA 上显示图片，创建了四个 ROM 的 IP 核，如右图所示。



mainwindow 为 480*580 的主图片，如下所示：



boll 为 30*30 的小球图片，如下所示：



overgame 为 60*150 的游戏结束图片，如下所示：

wingame 为 100*100 的游戏胜利图片，如右图所示：

其中，boll、overgame 和 wingame 这三幅图片都使用 matlab 工具进行处理，将小球边缘、overgame 图片的黑色区域、wingame 的黑色区域都进行了标记，在主模块中进行识别后，选择不显示这部分的内容。具体的实现方法为：将这部分像素点标记为 0，当读取到这部分像素点时，代码进行检测，如果该位置的值为某个特定值，那么选择不读取该图片，转而读取 main_data 的值。代码实现如下：

```
always@*begin
  location_reg1 = x*10'd640+y;
  if(over == 1 && x >= 210 && x < 270 && y >= 245  && y < 395)
      data = (over_data == 12'hfff) ? 12'hfff : main_data;
  else if(win == 1 && x >= 190 && x < 290 && y >= 270 && y < 370)
      data = (win_data == 12'h000) ? main_data : win_data;
  else if((x >= position_x-15 && x < position_x+15) &&
  (y >= position_y-15 && y < position_y+15))
      data = (boll_data == 12'h000) ? main_data : boll_data;
  else if(y > 580)begin
      if(x >= 160 && x < 200)
          data = easy_data == 12'hfff ? 12'hfff : (choice == 2'b01 ? 12'hf00 : 12'h000);
      else if(x >= 220 && x < 260)
          data = normal_data == 12'hfff ? 12'hfff : (choice == 2'b10 ? 12'hf00 : 12'h000);
      else if(x >= 290 && x < 330)
          data = hard_data == 12'hfff ? 12'hfff : (choice == 2'b11 ? 12'hf00 : 12'h000);
      else
          data = main_data;
  end
  else
      data = main_data;
end
```

可以看到，我在为 data 赋值时，先判断了相应的 over_data、win_data 和 boll_data 的值是否为我所埋下的种子位，如果是，那么就显示 mian_data（背景图片），这样就实现了“透明色”的设计。

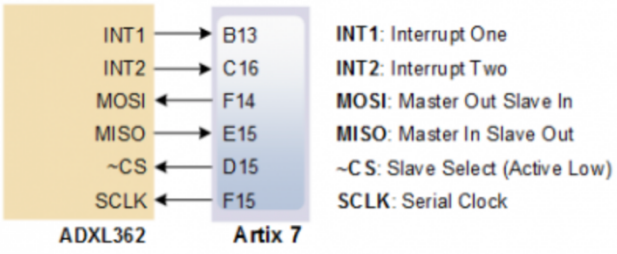
1.2 • 使用 MATLAB 进行图片向 COE 文件的转换

为了将图片转换为 coe 文件，让 rom 生成工具能够识别读取，我使用了 Matlab 进行编程，将特定图片转换为 coe 文件，源代码如下所示（此为生成主图片的代码，其他图片的代码略有不同）：

```
mat = mat(:,:,:)/16;
mat(mat>1) = mat(mat>1)-1;
mat = int16(mat);
file=fopen('main.coe','w+');
fprintf(file,'memory_initialization_radix = 10; \n');
fprintf(file,'memory_initialization_vector = \n');
for i=1:480
    for j=1:640
            fprintf(file,'%d',mat(i,j,1)*256+mat(i,j,2)*16+mat(i,j,3));
            fprintf(file,',\n');
            test(i,j) = mat(i,j,1)*256+mat(i,j,2)*16+mat(i,j,3);
    end
end
fclose(file);
```

2• ADXL362CTRL 模块

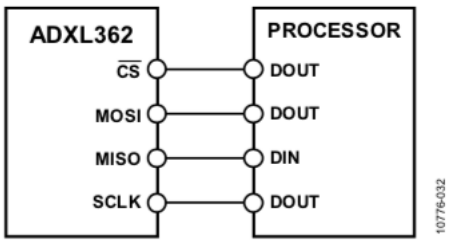
这个模块为 nexys4 ddr 板载的三轴加速度传感器的驱动模块，加速度传感器是本游戏的核心 IO，人机交互体验是否流畅都在于这个模块的设计是否合理。为了深入理解该传感器的工作原理，我在 http://www.digilentinc.com/ 官方网站上找到了我的这块板子的官方 manual：https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual 其中，第十四节简要介绍了该 Analog Device ADXL362 accelerometer 设备的接口。



如左图所示，该加速度传感器采用 SPI 通信方案。推荐的 SPI 时钟频率范围为 1MHz 至 5MHz。SPI 工作在 SPI 模式 0，CPOL = 0 且 CPHA = 0。与器件的所有通信都必须指定一个寄存器地址和一个标志，指示通信是读还是写。实际的数据传输总是遵循寄存器地址和通信标志。可以通过写入加速度计内的控制寄存器来执行器件配置。通过读取设备寄存器来访问加速计数据。

ADXL362 始终提供 12 位输出分辨率。测量范围为±2 g，±4 g 和±8 g，分辨率为±2 g 范围内的 1 mg / LSB。FPGA 可以通过 SPI 接口与 ADXL362 进行通信。当 ADXL362 处于测量模式时，它会连续测量和存储 X 数据，Y 数据和 Z 数据寄存器中的加速度数据。

官网的介绍过于简单，无法深入摸清数据传输的原理，所以我到了 Analog Device 官网上去找该设备的使用手册：http://www.analog.com/media/cn/technical-documentation/data-sheets/ADXL362_cn.pdf 。



我通过阅读说明书，了解到了他的数据传输原理，如左图所示，其中 miso 为输入信号，由加速度传感器向 fpga 输入信号，其他三个为输出信号，由 fpga 向加速度传感器输出信号。工作原理为，cs 为低电平有效位，当 cs 为 0 是，我们开始读取数据，sclk 为时钟信号（因为 SPI 是串行数据通信协议，串行通信协议需要时钟信号来保障双端数据传输没有错位，正确传输）。然后 mosi 发出一个字节的命令，命令集如下：

- 0x0A:写入寄存器
- 0x0B:读取寄存器
- 0x0D:读取FIFO

因为我们只需要读取数据，所以采用 0x0B 的 instruction 信号，然后 mosi 传入 8bit 的地址，接下来，从 miso 中读取 8bit 的数据，该数据即为我们想要的 x、y 轴加速度值。因为我们需要不断读取数据，所以需要设计时钟信号 cs，让其隔一段时间就获取一次数据，以保证游戏的流畅运行。相应的信号图如下所示：

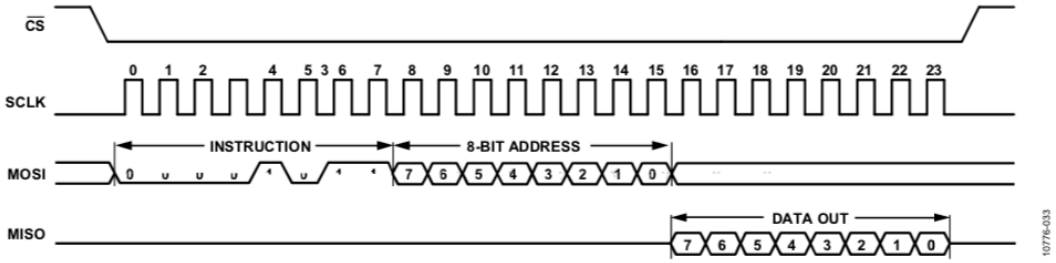


图36. 寄存器读取

该模块我使用了官网 demo 里的由 VHDL 写成的驱动模块，凭借我对 SPI 协议的理解，根据我这个游戏的需要作出了相应的个性化修改，采用 verilog 和 VHDL 混合编译的方式，在 verilog 中调用该 VHDL 模块，下面对该模块的输入输出进行一下简单的介绍：

如右图所示：

sysclk 为时钟信号，直接连接到 100mhz 的时钟即可。reset 的信号为复位信号。accel_x、accel_y、accel_z、accel_tmp 分别为 x、y、z 轴的加速度（本游戏只用到了 x、y 轴的加速度数据），和温度（ADXL362 还内置了温度传感器，不过本游戏没有用到）。下面四个 sclk、mosi、miso、ss 为 SPI 通信的四个引脚，需要在引脚约束文件中进行配置。

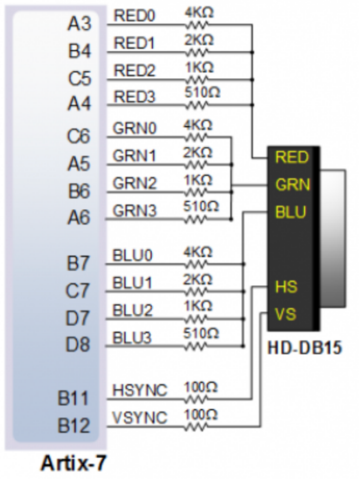
```
port
(
   SYSCLK      : in STD_LOGIC; -- System Clock
   RESET       : in STD_LOGIC;

   -- Accelerometer data signals
   ACCEL_X     : out STD_LOGIC_VECTOR (11 downto 0);
   ACCEL_Y     : out STD_LOGIC_VECTOR (11 downto 0);
   ACCEL_Z     : out STD_LOGIC_VECTOR (11 downto 0);
   ACCEL_TMP   : out STD_LOGIC_VECTOR (11 downto 0);
   Data_Ready  : out STD_LOGIC;

   --SPI Interface Signals
   SCLK        : out STD_LOGIC;
   MOSI        : out STD_LOGIC;
   MISO        : in STD_LOGIC;
   SS          : out STD_LOGIC
);
```

3• VGA 显示模块

右图为 VGA 显示的引脚约束图，其中，RED[3:0]为红色通道值（四位 bit 共 16 级分化），其他的 BLUE[3:0]、GREEN[3:0]为相应的蓝色、绿色通道值。HSYNC、VSYHC 为水平、垂直同步信号。



使用这个电路，可以显示 4096 个不同的颜色，每个像素点都有独特的 12 位模式。必须在 FPGA 中创建一个视频控制器电路，以正确的时序驱动同步信号和彩色信号，以便生成一个可用的显示系统。

视频数据通常来自存储器;一个或多个字节分配给每个像素位置（Nexys4 DDR 使用每像素 12 位）。控制器必须在光束在显示器上移动时将其索引到视频存储器中，并在电子束在给定像素上移动时精确地检索和应用视频数据到显示器。

VGA 控制器电路必须产生 HS 和 VS 定时信号，并根据像素时钟来协调视频数据的传送。像素时钟定义显示一个像素信息的时间。VS 信号定义了显示器的“刷新”频率，或显示器上所有信息重新绘制的频率。最小刷新频率是显示器荧光粉和电子束强度的函数，实际刷新频率落在 50Hz 到 120Hz 范围内。在给定刷新频率下要显示的行数定义了水平“回扫”频率。

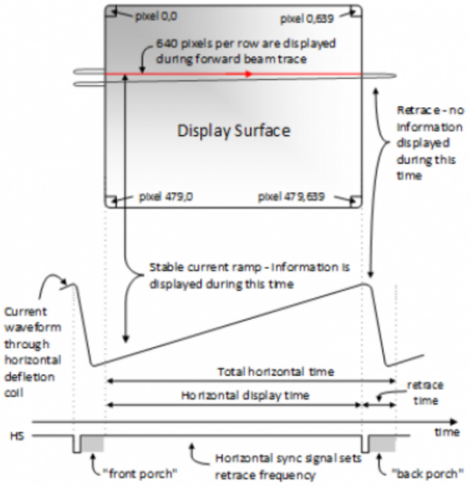


Figure 13. VGA horizontal synchronization.

具体工作机理如右图所示：

4• BCD 转换模块

该模块十分简单，主要工作就是把传入的数据（二进制）进行处理，转换为十进制输出，该模块的作用是将加速度传感器输出的值转换为十进制，然后送给七段数码管模块进行显示。

代码如下图所示，输入数据为 12-bit 的二进制数值，输出为 16-bit 的值，分为四个 4-bit 值，每个部分范围都为 0-9，作为转换后的四位 10 进制数值输出。

```
module bcd(
    input [11:0] in,
    output [15:0] out
    );

    wire [11:0]split1 =in;
    wire [11:0]split2 = in/10;
    wire [11:0]split3 = in/100;
    wire [11:0]split4 = in/1000;

    assign out[3:0] = (split1[3:0] > 12'd9 )? 9: split1[3:0];
    assign out[7:4] = (split2[3:0] > 12'd9 )? 9: split2[3:0];
    assign out[11:8] = (split3[3:0] > 12'd9 )? 9: split3[3:0];
    assign out[15:12] = (split4[3:0] > 12'd9 )? 9: split4[3:0];

endmodule
```

5• DISPNUMBER 七段数码管显示模块

该模块读取 32-bit 的值，然后在七段数码管的八个位置上输出数值，代码做到了稳定输出、无 bug。

四• 调试过程分析

代码写的越复杂，bug 也就越多，在游戏设计的过程中，我遇到了非常之多的 bug 和问题。下面一一讲起。

1• VGA 显示

在用 Matlab 处理图片时，一个比较坑的地方就是，vga 显示模块支持 16 种不同色彩（因为有四位 bit）但是现代网络上找到的图片基本上都为 256 种色彩的（即 int8），所以，我在相应代码（详情见三.1.2 节中的代码）的第一行将所有的矩阵值都除了 16，把它们从 256 种转换为了 16 种，然后因为这样的除法有些值会变成 16，这是不被允许的，因为 verilog 的代码值是 0-15，并没有 16 这个数字，所以我将所有大于 1 的数字都减 1，这样才保证了没有 16 这样的数字出现。

2• 时序电路电路遇到的不同步问题

在时序逻辑设计是，遇到了很麻烦又难以解决的错位问题，这个问题困扰了我很久没想明白，后来仔细分析代码才想通了这个问题，并想出了解决方案，下面详细介绍我遇到的困难和解决方法。

这次实验也告诉了我，在设计电路时能用组合逻辑的一定要用组合逻辑，只有迫不得已时才用时序逻辑。因为时序逻辑的时延，可能会造成很多难以想象的问题，需要很细心谨慎的操作才能避免。

先且看这段代码：

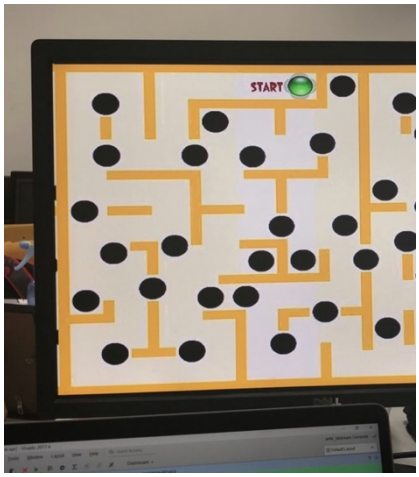
```
vgac main(.vga_clk(clkdiv[1]),.clrn(rstn_i),.d_in(data),
.row_addr(x),.col_addr(y),.rdn(enable),.r(vga_red_o),
.g(vga_green_o),.b(vga_blue_o),.hs(vga_hs_o),.vs(vga_vs_o));

blk_mem_gen_0 mainwindow(.clka(clkdiv[1]),.ena(1'b1),
.addra(location_reg1),.douta(main_data));

always@(posedge clkdiv[1])begin
  location_reg1 = x*10'd640+y;
end
```

vga 模块输出 x、y 轴的坐标数据，代表了此时扫描到了哪一个像素点，然后在 always 中，计算出相应的 location 值，然后根据这个 location 值，到 rom 中去读取数据，rom 输出该地址的数据，然后将该数据传回 vga 模块进行显示输出。

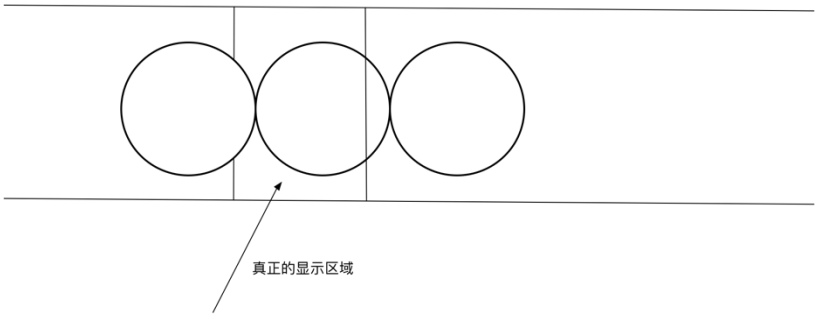
乍一看似乎没有任何问题，但是这样在显示时，比如说显示主图片，会变成这样：



仔细一看，左边似乎没有顶到头，而是输出了最右边的像素点，相应的，最右边的像素点也消失没显示了。似乎是整个图片整体向右偏移了一段距离，然后最右边的跑到了最左边去了。这是为什么呢？

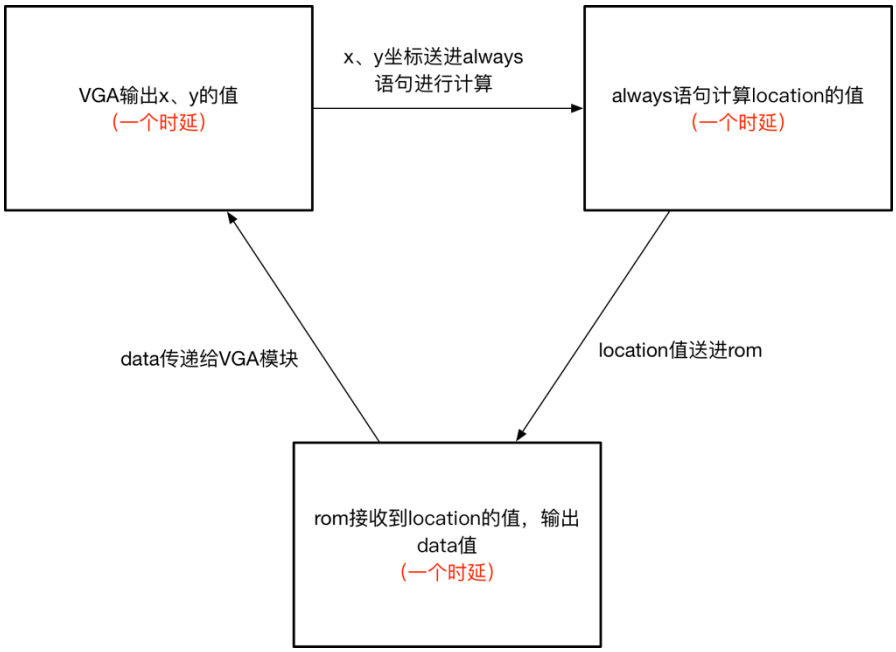
原因在于，我 vga 模块输出的 x、y 的坐标，在 always 语句中不会立刻计算这个 location 值，他会等到下一个时钟（clkdiv[1]）来的时候才更新 location 的值，同时，location 的值变化后，rom 的 data 输出也不会立刻改变，而是等到下一个时钟来的时候，data 才会发生变化，同样的，vga 也要等到再下一个时钟来的时候才使用这个 data 值。

所以，以小球为例显示就会变成这样：



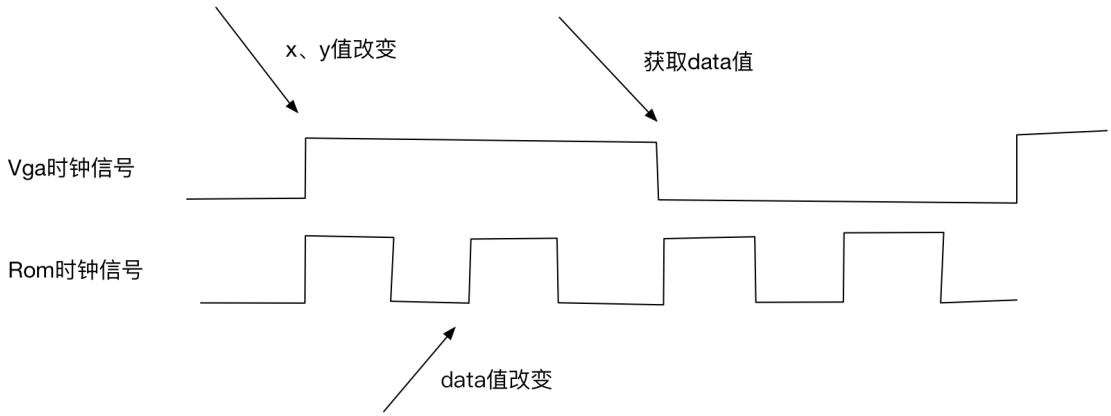
即我们因为是循环读取，所以越界的坐标值被自动取模，相当于是回到了最开始点，所以显示的是如图的区域范围，从而很好的解释了为什么显示屏输出会是那个样子。

为了更好的理解这个问题，状态图如下图所示：



那么如何解决呢？首先，将 always 模块的触发条件改为 always（*），这样就将时序电路转换为了组合逻辑，此模块现在逻辑无时延（当然物理电路电压改变一定会有时延）。但是 rom 是没有办法改成组合逻辑的，所以不可以用这种方法解决。故，我采用了 vga 与 rom 模块不同的时钟信号来解决这个问题，让 rom 模块的时钟比 vga 快四倍（即，vga 将 100mhz 的时钟进行两次分频处理，变成 25mhz），然后把 vga 模块里面读取 data 时序逻辑的 posedge 上升沿触发改为 negedge 下降沿触发。这样，当 vga 在上升沿传输出 x、y 时，不立刻读取 data 的值，在此时 rom 经过四分之一个 vga 的时钟周期（因为 rom 的时钟比 vga 快四倍，所以可以快四倍的改变数据），data 值改变，准备好了给 vga 进行读取。然后在 vga 时钟的下降沿时，vga 读取 data 的值，进行显示，这样就做到了 x、y 对应的 data 值准确无差。

逻辑如下图所示：

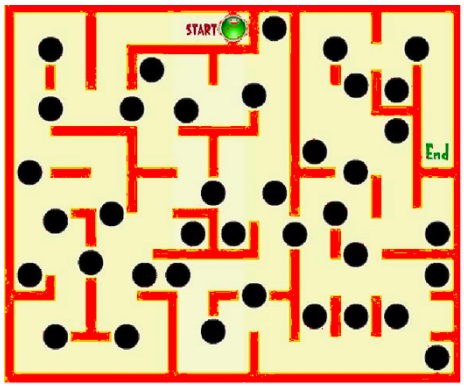


自此，该问题完美解决。

3• 存储空间有限带来的同步读取冲突问题

在我的 nexys4 ddr 板上，由于内存有限，创建的 rom 只能存一张 640*480 的图片，不能存两张。我一开始用创建了两张 640*480 的图片，结果在综合的时候报错了，提示我板上的内存空间不够。所以我一开始的打算用一张图片做主图片（640*480），另一张图同样大小的图片来记录黑洞和墙壁位置信息的计划就泡汤了。那么这也就说明了我必须在主图片中，去记录墙壁和黑洞的信息，与此同时还不能改变这个图片的颜色，以免显示效果不佳。于是我想到了将某一通道的值（我最后采用的是 blue 通道），变成一个特定的值，作为埋藏在这个图片之中的信息，标记了黑洞和墙壁的信息。黑洞的颜色很单一，就是黑色，对应的 blue 通道值也为 0，所以我就将其区域内值全部改为了 0。墙壁的值有点复杂，通过查看图片的像素矩阵，我发现了墙壁的 blue 通道值大部分都介于 190-200 之间，于是，我使用 Matlab 编程，遍历矩阵的每个值，将 blue 通道像素值在 190-200 之间的点的 blue 通道值改为了 200，因为相差无几，所以图片看起来和原来没有什么区别。为了说明处理效果，我写了一段测试代码，把我识别出来的墙的位置标记为红色，如下图所示：

```
for i = 1:480
    for j = 1:580
        if(mat(i,j,2) > 190 && mat(i,j,2) < 200)
            mat(i,j,:) = [ 255 0 0];
        end
    end
end
```



可见，这种处理的效果很好，大部分的墙壁都已经被准确识别并标记了。

但是，这又带来了另外一个问题，我使用的是单口 rom，单口 rom 在同一时间只能输入一个地址，输出一个相应的数据。但是，我的 vga 显示模块需要持续地读取 rom 的值，以便不断地扫描输出。然而，我每隔一段周期，测试让小球移动的时候，仍然需要读取 rom 的值，重要的是，我读取的地址和 vga 需要读取的地址并不相同（因为小球能否往左走，是看向左半径距离的位置是不是墙壁）。这就会造成冲突，就是 rom 到底为谁服务的问题。

但是我注意到，小球的移动其实很慢，换句话说，vga 所需要的 25mhz 的时钟对于小球移动来说太快了。小球大概 0.1 秒移动一次，就已经足够了。这就给我了设计的空间。我先将 clk 进行分频处理，得到差不多 0.1 秒数量级的时钟周期，此 status 信号每隔一段很长的时间变为 1 一次，然后在下个时钟周期变为 0。代码如右图所示。

```
always@(posedge clkdiv[1])begin
    if(count < 100000)begin
        count <= count+1;
        status <= 0;
    end
    if(count >= 100000)begin
        count <= 0;
        status <= 1;
    end
end
```

然后使用了一个新的地址寄存器来计算需要的值（在代码中，该地址被命名为 location_reg2），rom 读入的地址变为了 status ? location_reg2 : location_reg1 。这样当 status 为 1 时，rom 读入 location_reg2，因为周期相比于 vga 很长，所以 vga 显示造成的误差几乎可以忽略不计。

这里的 always 模块，同样采用了 negedge 触发（注意到，我的 status 生成模块使用的是 clkdiv[1]触发，这样可保证 status 为 1 的时间恰好是 rom 传输周期的两倍），根据上一节的分析，可以保证得到的 rom 输出的 data 数据无任何延迟与错误，对于有无墙壁的判断，这是十分十分关键的，因为这一时刻我可能去判断小球上面有无墙壁，下一时刻去判断小球左边有无墙壁，如果误差一个周期，那将导致数据完全错误，无法达到所需功能。

```
always@(negedge status)begin
```

关键一步

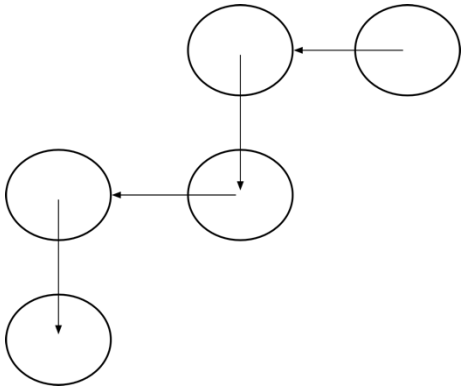
4• 引入难易程度规则和多阶段加速度造成的移动锯齿

在这节中，我为了流畅游戏体验，牺牲了一部分代码的间接性，使得电路设计变得更为复杂，做了一个 tradeoff。

在游戏主要功能设计完成后，为了丰富游戏体验，我设计了三档难度，随着难度的增加，小球移动的速度越快。同时，我还引入了在倾斜不同角度 fpga 板时，小球移动速度不同的特性，这也符合了正常的操作逻辑，使得游戏更有趣味。让小球更加难以控制，必须小心翼翼才能不让小球运动的太快。

我这样设计后，初步的方案是，小球的速度为难度+倾斜程度，然后每个时钟周期来的时候，让小球一次性移动该速度个像素点。这样的设计的确达到了我所需要的设计需求，小球在难易程度不同的情况下会有不同的移动速度，但是也带来了新的问题，这是我在下板子测试游戏的时候发现的，也就是小球的移动有着明显的锯齿。

一个很重要的限制在于：小球不可能在同一个周期内向左上、右下、左下、右上这四个方向移动，这是因为在一个时间周期内，只能读取一次 rom 的值（这在前面已经详细介绍过了，rom 要等到时钟来的时候才送出 data 值）。那么为了向左上移动，我必须先向左移动，在向上移动。这就需要判断两次（先判断左边有没有墙壁，在判断上面有没有墙壁），这在一个时间周期内是不可能实现的。所以小球的移动就变成了如下图这样：



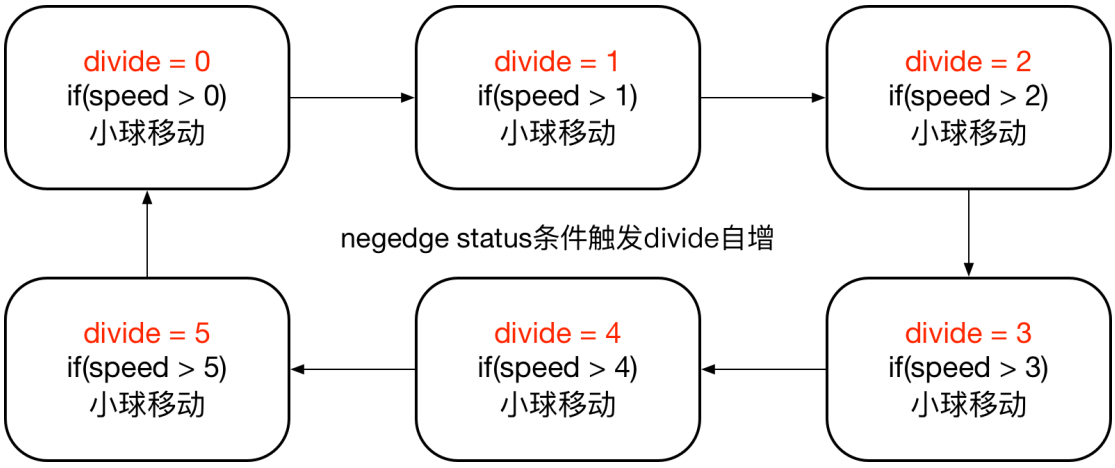
这样，如果每次一次性移动多格，在玩游戏的时候就会带来很不好的游戏体验。因为会观测到小球没有斜着移动而是多格多格的跳动。为了解决这个问题，首先我就把每次移动的距离强行改成了 1，直接确保小球移动的顺滑性。

那么，速度怎么体现呢？

于是，我先将 status 的速度调快三倍，即每次变为 1 的间隔时间变为原来的三分之一，然后在设计了一个 0-5 不断循环的计数器 divide：

```
if(divide == 3'b101)
    divide <= 0;
else
    divide <= divide+1'b1;
```

然后设计了一个 FSM 有限状态机。在每次 status 时钟周期来的时候，判断 divide 的值，进行不同的响应。如下图所示（speed 的值为 1-6）：



这样就完美控制了速度，原理在于，不同 speed 的情况下，完成 divide 从 0-5 这六个 status 周期，小球移动的次数是不同的，当然，这种复杂的设计将电路设计变得臃肿，综合速度也会变慢，做了一个 tradeoff。但是更新后，游戏体验十分流畅，如丝般顺滑。

五• 核心模块模拟仿真

由于我采用了三轴加速度传感器，这个模块的表现很难仿真（时时刻刻加速度的大小和方向都在改变），必须下板子查看不同的用户交互会带来什么样的后果，所以我没有对 top 主模块进行仿真。一切的调试都下板子进行。由于我采用了 vivado2017.4 集成开发环境，开启了多核心综合，使得综合速度相比 ISE 大大变快。最终完整版的工程，从综合到最后的生成 bit 文件总用时，也在十分钟左右，比起 ISE 动不动一两个小时，速度快了不止一点半点。

六• 实验体会

这次实验前前后后做了两周的时间，遇到的问题和 bug 层出不穷似乎没有尽头。往往都是拆了东墙补西墙，补上这个 bug 就又会出现其他的 bug。其实归根结底，是自己对数字逻辑设计和 verilog 语言的理解不够深刻。但是，通过这个实验，我对数字逻辑设计的理解增加了许多，可以说，如果没有这个实验，在第四章第二小节我所讲到的时序电路不同步的问题，我可能永远也不会想到。因为不会遇到，所以也不会去思考。所以，这个实验大大增强了我数字逻辑设计的能力，提高了我写 verilog 代码的能力，增强了设计工程的经验。我个人认为，付出这么多的精力是值得的。

七• 经验教训与总结

1• 经验教训

主要的经验教训，我都在第四章遇到 bug 分析的部分都已经讲到了。在这里无非提以下几个比较关键或者零散的点：

•电路设计能够使用组合逻辑的，尽量采用组合电路实现。因为时序电路因为时延的存在（需要等待下一个周期才会更新值），会带来各种意想不到的后果，并且在后期很难 debug 并解决。

•同一个 reg 变量，不能在多个 always 语句中赋值，否则一定在 route 布线的时候报错，但是在同一个 always 语句中，可以采用 if 语句在多个 if 块中进行赋值，最终的值为最后一个条件成立的 if 语句对应的值，因此，建议采用 else if 语句避免意想不到的后果。原因在于，always 语句中的代码是顺序执行的，所以同一个 always 块中对 reg 变量多次赋值，无非是不断刷新 reg 的值罢了，结果就是最后一个语句刷新的值。但是多个不同的 always 语句之间是并行执行的，如果在多个 always 里面对同一个变量进行赋值，那么到底谁先谁后无法判断，这是严禁的。

•一个 module 模块的输出 output，在他的顶层调用模块里面调用他时，绑定到 output 的变量必须为 wire 而不能是 reg。这是 verilog 语法所规定的，也符合正常的操作与设计逻辑。

•如果电路有形如 always@(btn[0])这样的语句，即 btn 上升沿触发，是不被允许的，综合时会提示 btn 不像 clk 有防抖动模块，所以不能作为 always 的触发条件，可以使用防抖动模块解决这一问题。

2• 总结

本次实验采用 vivado2017.4（xilinx 公司的最新发行版）进行开发，使用 verilog 和 VHDL 混合编译的方式，充分利用了 nexys4 ddr 板载的三轴加速度传感器进行人机交互，并通过一系列设计使得游戏体验尽可能的流畅。游戏前前后后大概使用了两周的时间进行设计，期间解决了许许多多的 bug，加强了自己对数字逻辑设计的理解，增加了自己的经验。