2021-1-2

秦嘉余 & 刘永鹏

191220088 & 191220070

1348288404@qq.com & 693901492@qq.com

数字电路大实验 打字小游戏

目录

[1. 实验基本内容 2](#_Toc60585320)

[2. 模块设计与综合 6](#_Toc60585321)

[1. 模块划分 6](#_Toc60585322)

[2. 模块功能协调 6](#_Toc60585323)

[3. 内存文件(.mif/.txt) 7](#_Toc60585324)

[3. 模块详解与代码实现 8](#_Toc60585325)

[1. 键盘模块 8](#_Toc60585326)

[2. 音频模块 10](#_Toc60585327)

[音量控制 10](#_Toc60585328)

[速度控制 11](#_Toc60585329)

[循环BGM播放 12](#_Toc60585330)

[3. 随机数模块 13](#_Toc60585331)

[4. 显示器模块 14](#_Toc60585332)

[像素点阵坐标处理 14](#_Toc60585333)

[颜色信息控制 15](#_Toc60585334)

[5. 逻辑控制模块 16](#_Toc60585335)

[声明rom及ram 16](#_Toc60585336)

[开始界面及画面显示 17](#_Toc60585337)

[显存更新 19](#_Toc60585338)

[随机信息生成 22](#_Toc60585339)

[等级调整 23](#_Toc60585340)

[字符队列更新 24](#_Toc60585341)

[4. 问题与解决方案 28](#_Toc60585342)

[1.显存如何规划？ 28](#_Toc60585343)

[2.循环如何实现？ 28](#_Toc60585344)

[3.读写冲突如何避免？ 29](#_Toc60585345)

[5. 版本更迭 30](#_Toc60585346)

[1. Version 1 30](#_Toc60585347)

[2. Version 2 30](#_Toc60585348)

[3. Version 3 30](#_Toc60585349)

[4. Version 4(Final Version) 30](#_Toc60585350)

[6. 分工情况 31](#_Toc60585351)

[刘永鹏 191220070 31](#_Toc60585352)

[秦嘉余 191220088 31](#_Toc60585353)

1. 实验基本内容

经过一番争论之后，我们最终决定模仿ICS-PA nemu typing game的游戏形式进行大实验的设计。在前者的基础之上，我们添加了许多自己实现的额外功能。

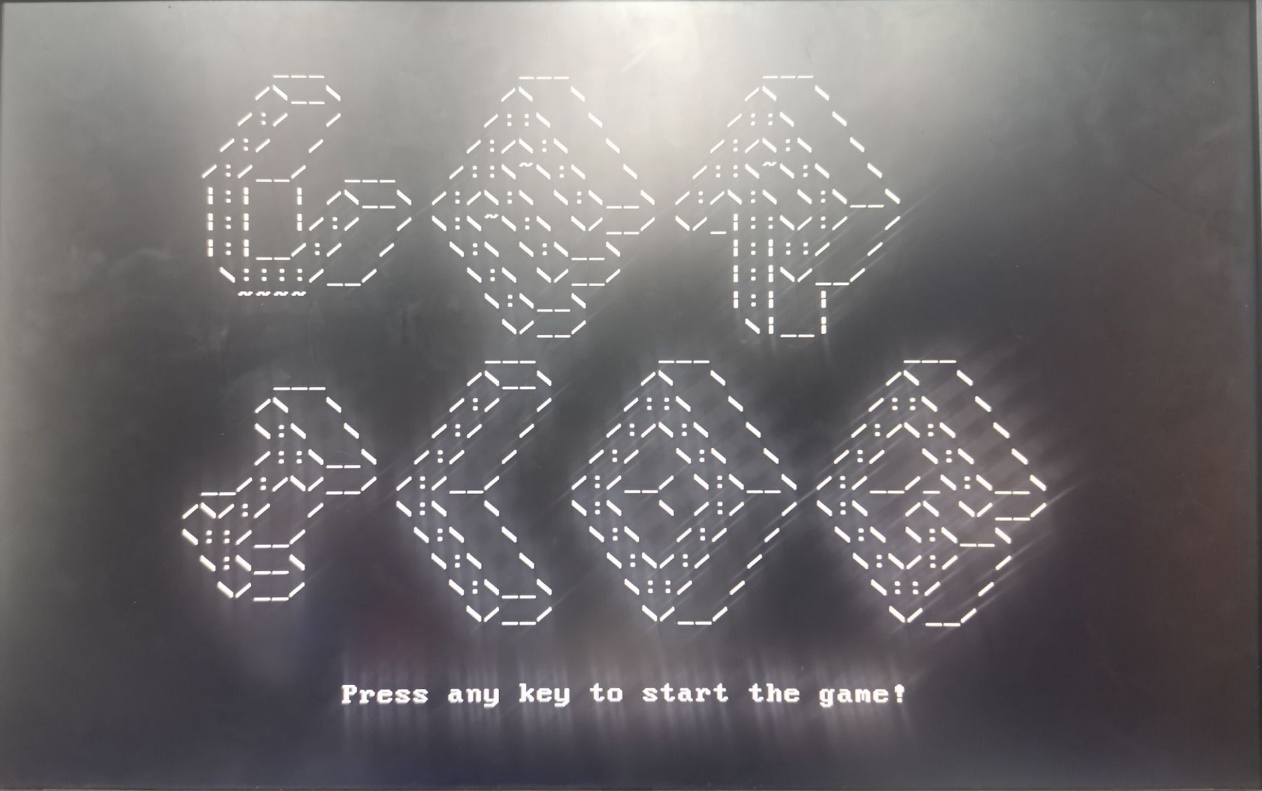
我们的独家字符小游戏可以实现如下功能：

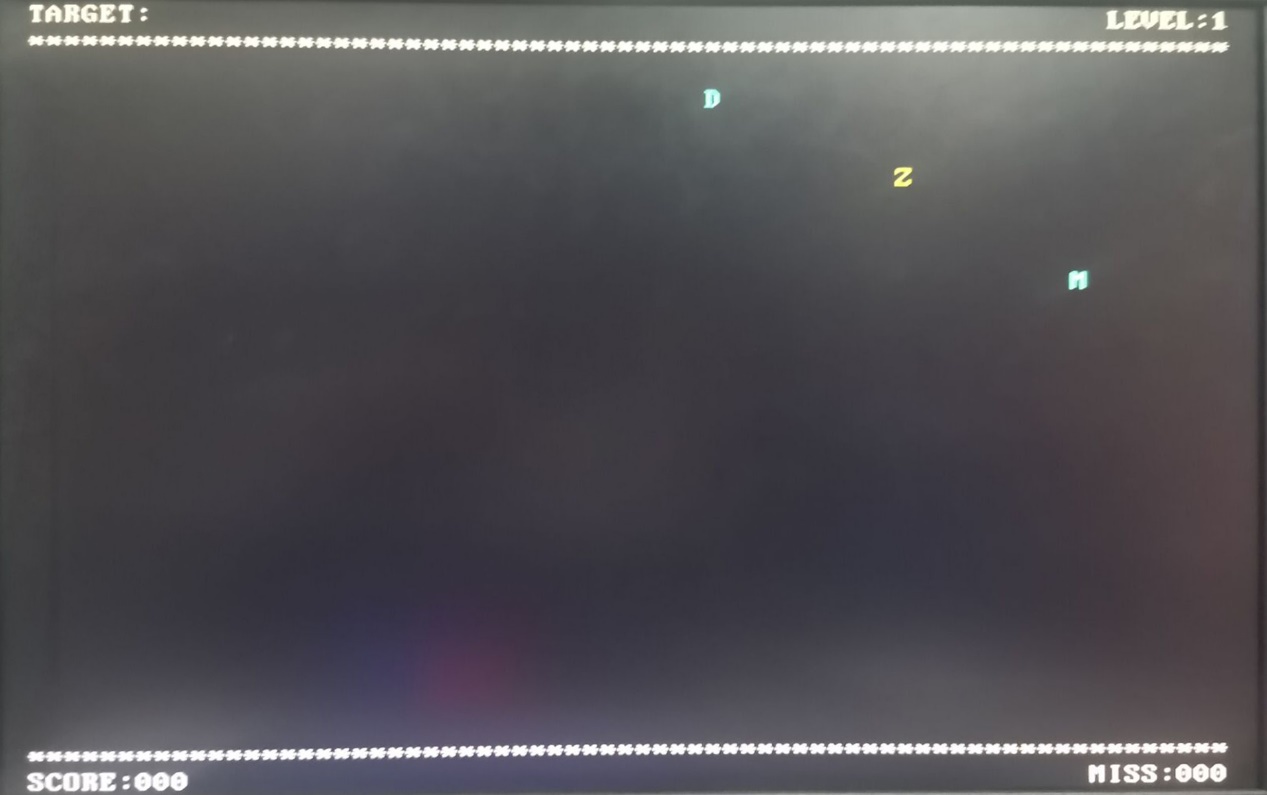
1. 首先，进入游戏前有一个初始画面，显示“Verilog”这个词的艺术字体（本想给游戏起个名字，但想象力着实匮乏）。
2. 游戏过程类似nemu-typing game：从屏幕上方随机位置随机生成A-Z的26个大写字母，玩家需要在字母掉落到屏幕底部前按相应的按键将它“消去”。
3. 掉落的字母有粉色、天蓝色、黄色三种，分别对应3、2、1三个“字符等级”。按下对应的按键，“字符等级”会对应的下降，当“字符等级”降为0时字符才算消去。（如粉色的F，在玩家第一次按下F时将变成天蓝色，第二次按下时变成黄色，第三次按下时才消去）。

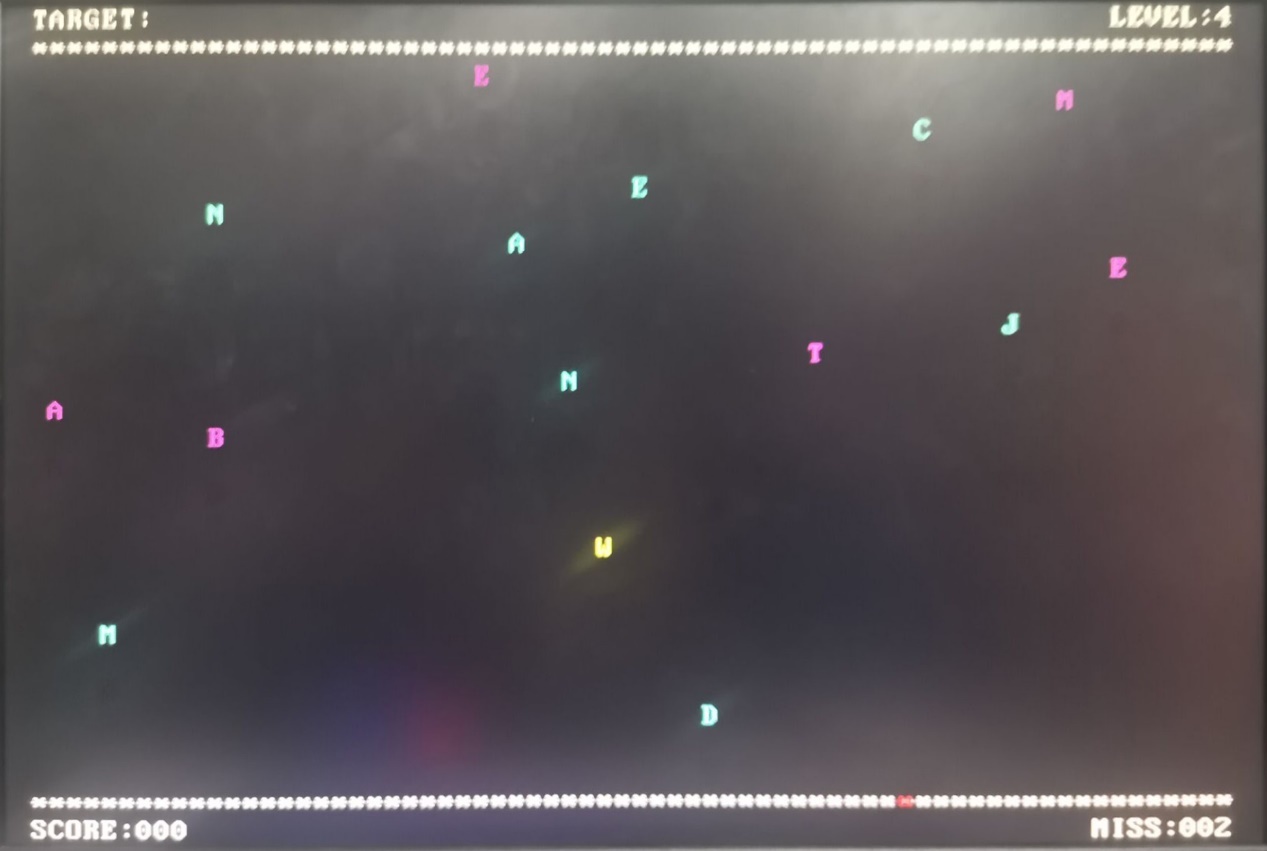
按下+抬起算作一次按下。

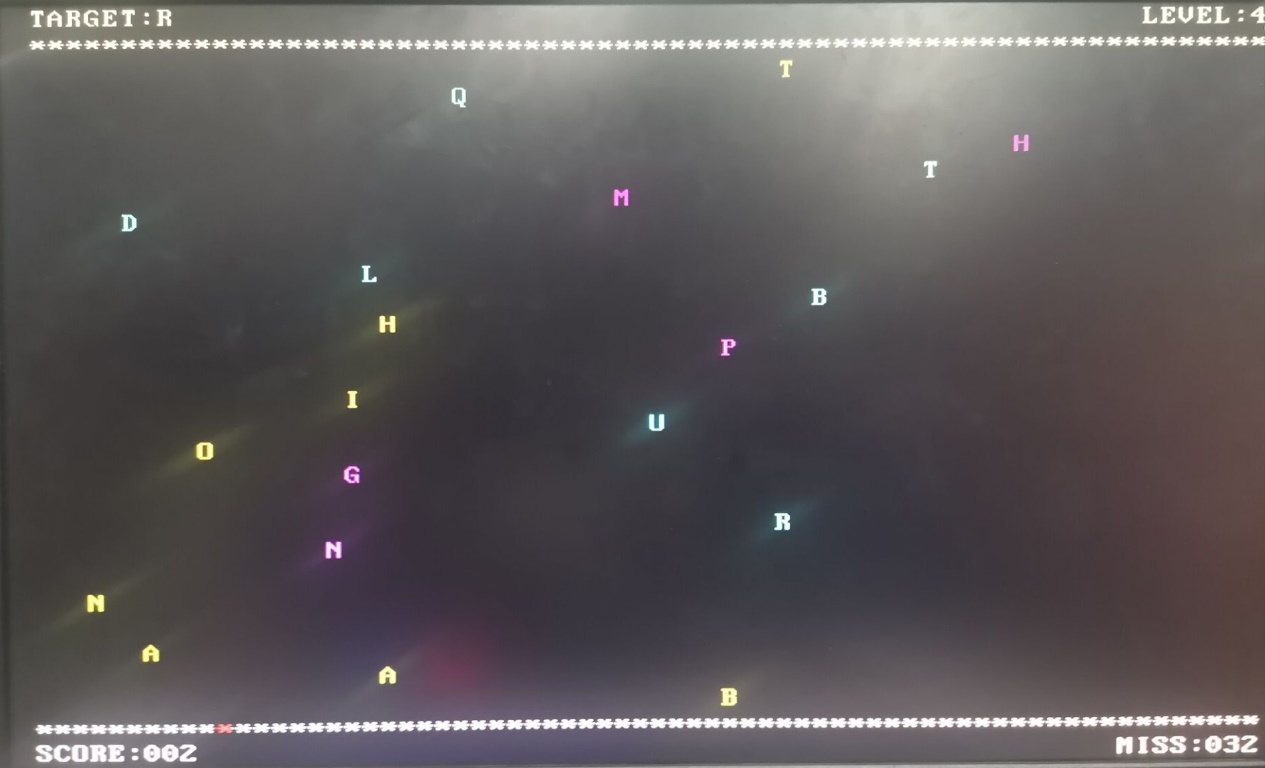
1. 当字符被消去时，会有一个颜色“渐隐”的动画效果（实现为颜色逐渐变深直至消失），当实际上持续时间比较短。
2. 游戏左下角和右下角分别记录了当前游戏的得分(Score)和漏掉的字母数(Miss)，每消去一个字符SCORE+1，字符掉落到底部则MISS+1。但我们并没有游戏胜利/失败的判定方式，我们认为这种设计并不能提高游戏的趣味性和可玩性，我们觉得我们的设计很cooooool。
3. 字符掉落后，字符对应列下方的边框将会变红一段时间，而如果成功消去，边框将会变绿。
4. 屏幕最上方将显示本次游戏已经进行的时间，你可通过时间+得分+MISS数来看看自己的手速是否达标。屏幕左上方会显示当前按下的字符的ASCII码，只有按下的字符是游戏定义有效的才会显示。
5. 不过，玩家可以通过FPGA的KEY[0]按钮手动重新启动游戏，在游戏过程中按下KEY[0]后，屏幕将会暂停游戏并弹出提示框，玩家需要再次按下KEY[0]确认重启或按下KEY[1]取消本次操作。
6. 也可以通过波拨动FPGA开关SW[0]的方式来暂停游戏，游戏暂停过程中无法进行操作，同时屏幕底部显示“THE WORLD!”字样。
7. 此外，玩家可以按键盘上的1，2，3，4数字键来改变游戏难度（由字符掉落速度和生成数量定义），数字越大，难度越大，可以在游戏过程中实时控制难度。屏幕右上角将显示当前难度水平（LEVEL 1/2/3/4）。
8. 我们还实现了游戏过程中的同步音频：当玩家按下一个有效键时，将会触发内置的BGM开始播放，默认持续1s，每次按下将会重启计时器。也就是说，如果按键的速度够快（手速够快），你可以一直听到连续的BGM。BGM设定为《碟中谍》系列电影的Theme。如果长时间（1s以上）没有按下有效键，BGM将会暂停。BGM将会循环播放。
9. 游戏难度将影响BGM的播放速度。随着难度增加，BGM播放速度也将递增，营造一种更加紧张刺激的氛围。BGM也可以通过键盘上的’-‘’+’来实时调节音量。

下面时运行时的截图，由于拍照时考虑不周，截取的图片可能无法体现出上述的所有功能，烦请见谅呀。











2. 模块设计与综合

## 1. 模块划分

经过我们的讨论，我们大致对主要模块进行了如下划分：

1. **键盘模块：**键盘模块负责接收键盘的按键信息，并将其转换为ASCII码传给主模块。键盘模块应当给出有效按键的ASCII码，对于无效按键（‘;’、‘：’等），输出0（相当于认为什么也没按下）。由于本实验中不涉及组合按键的功能需要，处于简化代码的考虑，没有实现组合按键的功能。
2. **音频模块：**音频模块接收主模块提供的信息（如根据当前难度等级调整播放速度等），经过处理后完成循环播放BGM的功能。
3. **随机数模块：**该模块负责生成随机数信息，这一模块将会有多个实例被其他模块调用，如生成随机字符的ASCII码信息，以及生成的位置信息等。
4. **显示器控制模块：**该模块负责根据显存信息刷新显示器。
5. **逻辑控制模块：**本模块是实验核心：它将负责调用随机数模块生成随机数，控制每一个字符的正常下落，决定字符的生存期（根据键盘按键的有效性和字符位置是否已到底部）并更新显存。SCORE和MISS以及时间等游戏信息都由本模块维护并更新。
6. **其他模块：**包括生成相应时钟的时钟模块，在HEX七段数码管上打印调试信息的调试模块等。

## 2. 模块功能协调

如上所述，本实验以逻辑控制模块作为核心，其他模块围绕该模块进行辅助提供相应的功能实现。

音频模块、显示器模块和键盘模块都是的相对独立，它们不需要调用任何其他模块，而只需要被其他模块去调用。这也是因为这些模块直接控制底层硬件和忽略实验本身的游戏逻辑实现，因此对它们进行功能封装是合理的。在主模块中，音频模块和键盘模块需要联动控制实现上述功能简述中BGM的播放控制。

随机数模块也相对独立，它采用《数字电路与数字设计》教材中的环形计数器算法提供随机数生成，它根据外部时钟每过一段时间生成一个随机数，这个随机数也将受外部参数控制，以满足我们的需求。

逻辑控制模块作为实验核心模块，可以分为两大部分：显存管理模块和字符逻辑模块。顾名思义，显存管理部分将根据逻辑模块的处理结果更新显存，维护字符的ASCII码信息与到显示器位置的映射信息，并维护相应的颜色信息以满足“多彩字符”“字符等级”的功能需要。逻辑模块则通过循环遍历字符寄存器组来对每一个字符进行相应处理。显存管理与逻辑控制的时序彼此分离，以避免显存读写带来冲突。

键盘模块、逻辑控制模块、音频模块都在主模块中并行调用，而随机数模块和显示器模块只在逻辑控制模块中调用，此外，负责时钟控制的时钟模块在各模块中均有出现。

我们以功能为单位对项目进行如下划分，封装相应的组成模块，使得在调试以及后续额外功能的添加上获得便利。

## 3. 内存文件(.mif/.txt)

需要如下预设的内存文件：

**键盘模块：**

1. Ascii.mif: 扫描码到ASCII码的转换文件

**音频模块：**

1. Note.txt: 用于音频频率控制的文件，记录了两个八度共24个音符。
2. Sintable.mif: 用于将频率转换为正弦波振幅。

**显示器模块：**

1. Vga\_font.mif: ASCII码字符显示点阵文件。
2. Videa\_ram.mif: 显存文件，双口双时钟控制，IP核生成。
3. Begin.mif: 初始化界面生成。我们使用某网页插件进行生成：

（<http://patorjk.com/software/taag>）

模块的具体关系图如下(由于图片过大，请旋转查看或者直接打开压缩包内对应文件：模块图.png查看)：



3. 模块详解与代码实现

## 键盘模块

always @ (posedge clk)

begin

    if(en && ready && nextdata\_n == 1'b1)

    begin

        nextdata\_n <= 0;

        if(flag == 1)

        begin

            q <= 8'h00;

            flag <= 0;

        end

        else if(data == 8'hf0)

            flag <= 1;

        else if(asdata >= 8'h61 && asdata <= 8'h7a)*//valid input: a-z*

        begin

            q <= asdata - 8'h20;*//to capital*

        end

        else if(asdata >= 8'h31 && asdata <= 8'h34)

        begin

            q <= asdata;

        end

        else if(asdata == 8'h3d || asdata == 8'h2d)

        begin

            q <= asdata;

        end

        else begin

            q <= 8'h00;

        end

    end

    else begin

        q <= q;

        nextdata\_n <= 1;

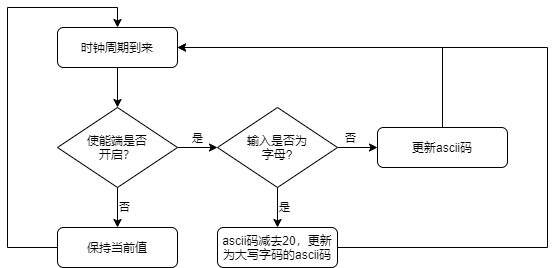
    end;

end

键盘控制模块沿用了实验八的设计，在该框架的基础上做了一定加工。首先，鉴于组合键功能在本次实验中并无用武之地，因此干脆设计时不考虑组合键功能，也就抛弃了较为复杂的状态机设计。

使用if-else语句可以很好的完成所需要的工作，当需要添加新按键时，只需要添加一个if-else分支即可。

我们仍然使用一个en 输入端口来控制整个模块的使能情况，虽然该模块在任何情况下都应当工作，也就是en输入端直接接高电平，但处于可扩展性的考虑，仍然添加了使能端。我们的所有模块中都有使能输入的设计，无论是否派上用场。该部分流程图如下：



## 音频模块

本模块共有三个always语块，下面将分别讲解。

### 音量控制：

先贴代码为敬：

always @ (posedge clk\_bgm\_vol)

begin

    if(kbdata == 61 && volume < 127)

    begin

        volume <= volume + 1;

        reset <= 1;

    end

    else if(kbdata == 45 && volume > 0)

    begin

        volume <= volume - 1;

        reset <= 1;

    end

    else begin

        volume <= volume;

        reset <= 0;

    end

end

Volume较好理解，当接收到键盘的“+”“-”信息时（事实上是‘=’‘-’信息），音频对传给audio\_config（音频控制芯片处理模块）的输出端volume进行修改。在修改音量后，应当实时获取音量修改的效果，因此对reset进行修改以便当按下修改音量对应的键时，不需要做其他操作使修改音量这一操作生效。

直接在修改volume时修改reset是我们对实验十：音频实验音量修改扩展功能的优化，效果还是比较令人满意的。

### 速度控制：

播放速度，即每个四分之一音符的持续时长。对于这个的修改，我们通过修改计数器的达标值进行实现。

always @ (posedge clk)

begin

    case(level)

    1: speedmax <= 10000000;

    2: speedmax <= 8000000;

    3: speedmax <= 6250000;

    4: speedmax <= 5000000;

    default: speedmax <= 10000000;

    endcase

end

Speedmax用来控制速度，而在bgm播放模块，则有如下判断：

    if(clk\_bgm\_cnt <= speedmax)

    begin

        clk\_bgm\_cnt <= clk\_bgm\_cnt + 1;

        freq <= freq;

    end

如果clk\_bgm\_cnt计数器没有达到speedmax的定义值，那么就将计数器+1, 不进行任何其他操作。因此，每一个speedmax到达时，才会播放一个新的音符

由于输入clk就是开发板定义的25000000Hz时钟，因此从Level 1到Level 4，每个音符的切换速度分别是0.4s, 0.32s, 0.25s, 0.2s，从而达到一个随游戏难度增加，BGM也越来越“紧张”的效果。

### 循环BGM播放：

由上所述，当cnt达到speedmax后，进入播放BGM的环节。由于本BGM的长度较短，因此直接采用case语句来记录要播放的音符信息。

begin

begin

        case (cnt)

           0: freq <= note[9];

            1: freq <= 0;

            2: freq <= 0;

            3: freq <= note[9];

            68: freq <= 16'h0496;

  ……

            default: freq <= 0;

        endcase

        if(cnt >= 87)

        begin

            cnt <= 0;

            clk\_bgm\_cnt <= 0;

        end

        else begin

            cnt <= cnt + 1;

            clk\_bgm\_cnt <= 0;

        end

    end

end

else begin

    cnt <= 0;

    clk\_bgm\_cnt <= 0;

    freq <= 0;

end

这里为了节省空间，只截取了case“乐谱”的首尾部分。每播放一个音符（即修改freq频率信息，这一信息储存在note寄存器组中，在初始化时从note.txt文件中读取）,除了控制speedmax播放clk\_bgm\_cnt需要递增，计数器cnt也要递增，从而在下一次进入case语句时，正确播放下一个音符的效果。当完成一次循环后，cnt置0，即重新循环这段旋律。

## 随机数模块

    always@(posedge en or negedge rst\_n or posedge srand)

    begin

        if(!rst\_n)

            num<=0;

        else if(srand)

            num<=seed;

        else if(en)

        begin

*//num<={num[7],num[0],num[1],num[2],num[3]^num[7],num[4]^num[7],num[5]^num[7],num[6]};*

            num<={num[4]^num[3]^num[2]^num[0],num[7:1]};

            out\_num<=num%(max-min+1)+min;

        end

    end

    parameter seed=100;

    parameter min=0;

    parameter max=255;

上图时random.v随机数模块的主要内容。我们使用《数字电路与数字设计》教科书中LSPR环形计数器的原理设计随机数生成器，经测验随机程度完全达标。

随机数生成器的种子以及上下界（见小图）则需要在调用该模块的时候显式指出，这一工作有上层模块完成。

## 显示器模块

显示器模块需要是实现的部分不多，我们只需要在逻辑控制模块完成对内存的修改，正确的将信息反馈给显示器即可。

### 像素点阵坐标处理：

    always @(vga\_clk)

    begin

        v\_address\_b<=(h\_addr/9)+(v\_addr>>4)\*70;

        if(begin\_en)

            line<=v\_addr[3:0]+(v\_q[7:0]<<4);

        else

            line<=v\_addr[3:0]+(begin\_q[7:0]<<4);

    end

    always @(h\_addr or v\_addr)

    begin

        blank<=font[(h\_addr+7)%9];

end

### 颜色信息控制：

*// 000 ffffff 白色*

*// 001 eefe25 黄色*

*// 010 10feee 天蓝色*

*// 011 e864d5 粉色*

*// 100 38f722 绿色*

*// 101 ee2d1b 红色*

    always @(h\_addr or v\_addr)

    begin

        if(blank && (h\_addr<630))

        begin

            case (color)

                0: vga\_data<=24'hffffff;

                1: vga\_data<=24'heefe25;

                2: vga\_data<=24'h10feee;

                3: vga\_data<=24'he864d5;

                4: vga\_data<=24'h38f722;

                5: vga\_data<=24'hee2d1b;

                default: vga\_data<=24'hffffff;

            endcase

        end

        else

            vga\_data<=24'h000000;

    end

## 逻辑控制模块

### 声明rom及ram：

    video\_ram vram(

    .address\_a(v\_address\_a),

    .address\_b(v\_address\_b),

    .clock(clk),

    .data\_a(v\_data),

    .wren\_a(1'b1),

    .wren\_b(1'b0),

    .q\_b(v\_q)

    );

    font\_rom from(

    .address(line),

    .clock(clk),

    .q(font)

    );

    begin\_rom brom(

    .address(v\_address\_b),

    .clock(clk),

    .q(begin\_q)

    );

在这个部分我们需要一系列rom即ram，上述代码声明了三个rom和ram，其功能与含义如下表所示：

|  |  |  |
| --- | --- | --- |
| 显存 | 对应文件 | 功能 |
| vram | video\_ram.mif | 保存显存信息 |
| from | vga\_font.mif | 保存字符信息 |
| brom | begin.mif | 保存开始界面信息 |

### 开始界面及画面显示：

这段代码中，通过一系列的显存索引，得到某一个像素点的明暗信息，保存在blank变量里，同时，为了显示颜色，我们使用color变量来表示颜色，最后通过如下的always语句块来赋值颜色，完成输出：

    always @(posedge clk)

    begin

        if(kbdata!=0)

        begin

            begin\_en=1'b1;

        end

        if(begin\_en)

        begin

            color<=v\_q[10:8];

        end

        else

        begin

            color<=0;

        end

    end

    always @(vga\_clk)

    begin

        v\_address\_b<=(h\_addr/9)+(v\_addr>>4)\*70;

        if(begin\_en)

            line<=v\_addr[3:0]+(v\_q[7:0]<<4);

        else

            line<=v\_addr[3:0]+(begin\_q[7:0]<<4);

    end

    always @(h\_addr or v\_addr)

    begin

        blank<=font[(h\_addr+7)%9];

    end

在此处，由于需要展示开始界面，所以我们使用一个begin\_en的使能信号，表示是否开始，当按下任意键时，开始游戏，可以通过上述代码发现，当使能信号有效时，表示为开始界面，我们读取的是开始界面显存的信息，对应的ascii码信息变量为begin\_q，而其无效时，我们读取的时标准显存，对应的ascii码信息变量为v\_q。

*// 000 ffffff 白色*

*// 001 eefe25 黄色*

*// 010 10feee 天蓝色*

*// 011 e864d5 粉色*

*// 100 38f722 绿色*

*// 101 ee2d1b 红色*

    always @(h\_addr or v\_addr)

    begin

        if(blank && (h\_addr<630))

        begin

            case (color)

                0: vga\_data<=24'hffffff;

                1: vga\_data<=24'heefe25;

                2: vga\_data<=24'h10feee;

                3: vga\_data<=24'he864d5;

                4: vga\_data<=24'h38f722;

                5: vga\_data<=24'hee2d1b;

                default: vga\_data<=24'hffffff;

            endcase

        end

        else

            vga\_data<=24'h000000;

    end

### 显存更新：

    always @(posedge update\_clk2)

    begin

        if(update\_signal2==0)

        begin...

        end

        else

        begin

            if(update\_signal3)

            begin

                if(update\_cnt2<128)

                begin...

                end

                else if(update\_cnt2==128)

                begin...

                end

            end

            else

            begin

                if((update\_cnt>=71&&update\_cnt<=138)||(update\_cnt>=1961&&update\_cnt<=2028))

                begin...

                end

                else if(update\_cnt>=1&&update\_cnt<=68)

                begin...

                end

                else if(update\_cnt>=2031&&update\_cnt<=2099)

                begin...

                end

                else if(update\_cnt<2100)

                begin...

                end

                else if(update\_cnt==2100)

                begin...

                end

            end

        end

    end

上述代码为控制模块中显存更新的主要逻辑代码，主要变量意义如下：

|  |  |  |
| --- | --- | --- |
| 变量名 | 含义 | 主要功能 |
| update\_cnt | 显存坐标 | 标识当前遍历的显存的位置范围为0~2099 |
| update\_cnt2 | 字符遍历坐标 | 标识字符队列遍历的位置，范围为0~127 |
| update\_signal2 | 更新信号 | 打开后进行显存的一次循环更新 |
| update\_signal3 | 遍历信号 | 打开后遍历字符队列，由此更新显存某一位置的信息 |

因为此处由于显存更新的复杂性，我们采用了类似于“信号传输”的方式来进行显存的更新，一次更新循环次数大概为2100\*128，在这种方式下，每次循环都只占用一个时钟周期，保证这种单一操作都可以在一个时钟周期内完成，在我们显存更新的规则主要如下：

（1）根据字母队列中每个有效字母的位置信息更新对于显存位置的信息

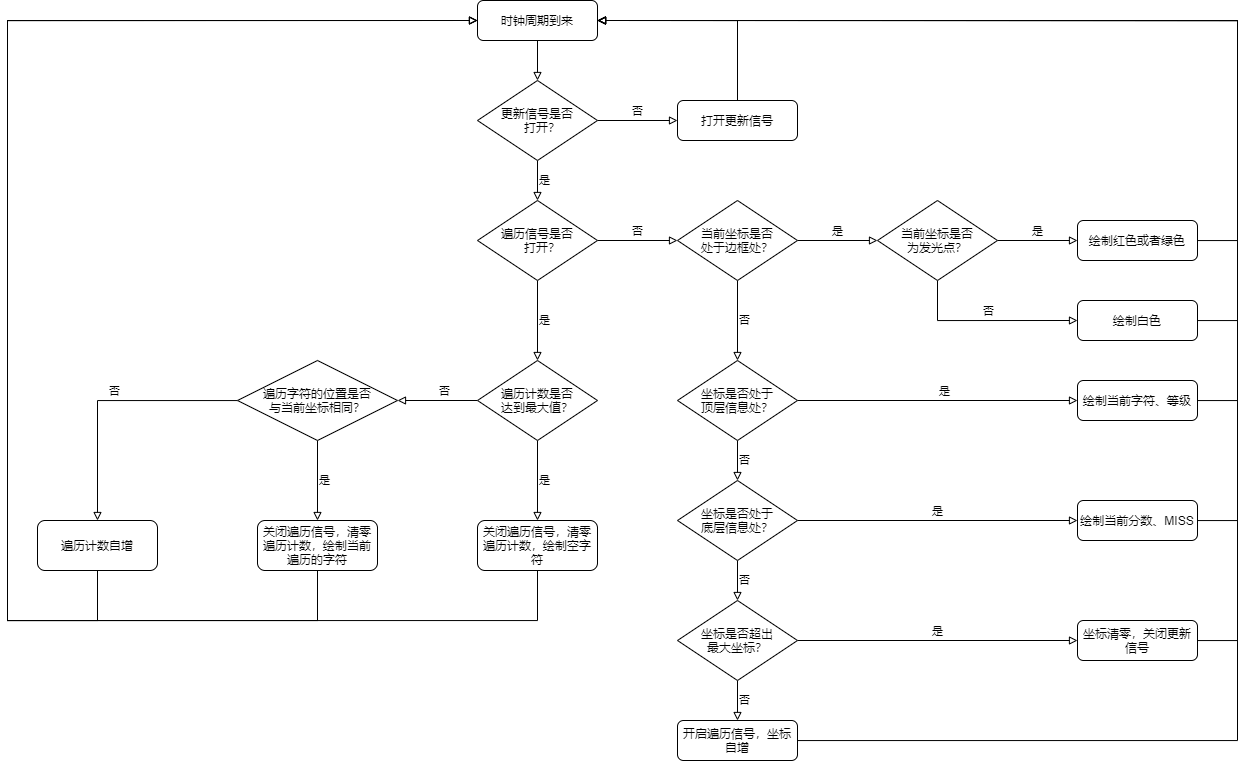
（2）在屏幕第二行和倒数第二行输出游戏区域的边框，同时此处需要判断此处的边框点是

否处于屏幕发光提示点的位置，如若是，则根据发光点颜色信息输出对应颜色

（3）在屏幕第一行输出提示信息，包括：当前键入的字符、游戏等级

（4）在屏幕最后一行输出提示信息，包括：当前的得分SCORE、当前的失分MISS、暂停信 息”THE WORLD!”

由于屏幕被我们分成了70\*30的单元格单位，所以可以得到对应的位置信息，分别指向对应操作。此处的具体操作逻辑见下方的流程图(由于图片过大，请旋转查看或者直接打开压缩包内对应文件：显存更新.png查看)：



### 随机信息生成：

    clkgen #(25000000/6250000) gen\_randomclk(clk,1'b0,1'b1,random\_clk);

    decode\_hex h2(testascii[3:0],HEX2);

    decode\_hex h3(testascii[7:4],HEX3);

    Random #(200,65,90) gen\_randchar(

    .en(random\_clk),

    .rst\_n(1'b1),

    .srand(1'b0),

    .clk(clk),

    .out\_num(rand\_char)

    );

    Random #(100,72,137) gen\_randloc(

    .en(random\_clk),

    .rst\_n(1'b1),

    .srand(1'b0),

    .clk(clk),

    .out\_num(rand\_loc[7:0])

    );

    Random #(50,1,3) gen\_randcolor(

    .en(random\_clk),

    .rst\_n(1'b1),

    .srand(1'b0),

    .clk(clk),

    .out\_num(rand\_color)

    );

在游戏中我们需要一些随机信息，例如：

（1）随机的ASCII字符

（2）随机的字符位置

（3）随机的颜色即字符等级

我们在先前实现了随机数生成模块，这个模块提供了3个参数，分别为：随机数播种数，随机数的最小值，随机数的最大值，通过改变这个三个参数，我们可以得到我们需要的任意大小的随机数，除此之外，需要改变播种值和随机数时钟，来保证不会产生同一时间内字符ASCII码和位置相关联的现象。

### 等级调整：

    always @(level)

    begin

        case (level)

            1:

            begin

                GEN\_FREQ<=40000000;

                FALL\_FREQ<=12500000;

            end

            2:

            begin

                GEN\_FREQ<=25000000;

                FALL\_FREQ<=10000000;

            end

            3:

            begin

                GEN\_FREQ<=12500000;

                FALL\_FREQ<=6250000;

            end

            4:

            begin

                GEN\_FREQ<=6250000;

                FALL\_FREQ<=5000000;

            end

            default:

            begin

                GEN\_FREQ<=40000000;

                FALL\_FREQ<=12500000;

            end

        endcase

    end

游戏中我们设置了不同的游戏难度，这些难度是由一个变量level来标识的，当level改变时，在这个always语句块内，我们修改了GEN\_FREQ和FALL\_FREQ变量，这两个变量分别代表字符的产生频率和掉落频率，通过改变这两个参数，可以使得游戏不同等级难度产生字符和掉落字符的速度不同，从而达到不同的游戏效果。

### 字符队列更新：

    always @(posedge en\_clk)

    begin

        if(modify\_cnt<MAX)...

        else if(modify\_cnt==MAX)...

        if(modify\_cnt%GEN\_FREQ==0)

        begin...

        end

        if(modify\_cnt%KB\_FREQ==0)

        begin...

        end

        else if(kb\_signal)

        begin

            if(kb\_cnt<128)

            begin...

            end

            else if(kb\_cnt==128)

            begin...

            end

        end

        if(modify\_cnt%FALL\_FREQ==0)

        begin...

        end

        else if(fall\_signal)

        begin

            if(fall\_cnt<128)

            begin...

            end

            else if(fall\_cnt==128)

            begin...

            end

        end

        if(light\_idx!=0)

        begin

            if(light\_cnt<12500000)...

            else

            begin...

            end

        end

    end

上述代码为控制模块中显存更新的主要逻辑代码，主要变量意义如下：

|  |  |  |
| --- | --- | --- |
| 变量名 | 含义 | 主要功能 |
| modify\_cnt | always模块总计数 | 每次时钟周期到来计数，为其他if语句块做准备 |
| GEN\_FREQ | 字符生成频率 | 总计数达到该值时生成一个字符，可以自由调整 |
| KB\_FREQ | 键盘响应频率 | 总计数达到该值时响应一次键盘，可以自由调整 |
| kb\_signal | 键盘信号 | 当键盘信号打开时，读取当前键盘的键码并更新当前ASCII码 |
| FALL\_FREQ | 字符掉落频率 | 总计数达到该值时字符整体掉落一次，可以自由调整 |
| fall\_signal | 掉落信号 | 当掉落信号打开时，将当前位置的字符位置信息向下调一行，若触底则有效位置零 |
| fall\_cnt | 掉落计数 | 即为当前需要调整坐标的字符的索引 |
| light\_idx | 发光提示位置 | 为游戏边框需要发光的显存对应位置 |
| light\_cnt | 发光计数 | 纪录12500000即0.5秒保证只每次只发光0.5秒 |

此处需要完成的功能为字符队列的更新，游戏更新的内容较多，且互不影响，所以在这个always语句块内我们编写了多个if-else语句块，分别完成不同的操作，由于每种功能的时钟长度不同，所以我们采取了传统生成不同频率时钟的方式，声明一个总计数不断增长，当计数模对应频率为0时，即可开启相应功能的if-else块。在这个always里我们需要完成的主要功能如下：

（1）总计数不段自增

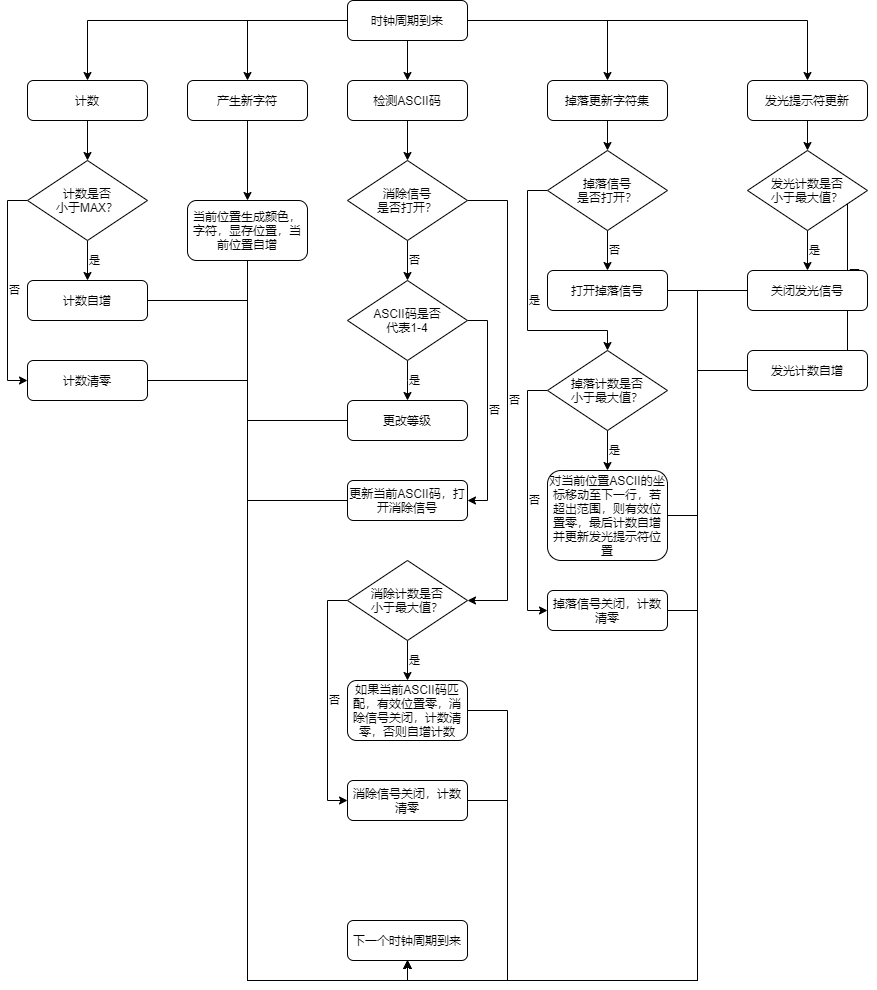
（2）生成新的字符和位置

（3）检测键盘传入的ASCII码，并消除字符，即有效位置零，同时更新得分

（4）每隔一段时间将所有字符的坐标下一行，若触底，则有效位置零，同时更新失分

（5）处理发光提示位

此处的具体操作逻辑见下方的流程图(由于图片过大，请直接打开压缩包内对应文件：字符队列更新.png查看)：



4. 问题与解决方案

## 1.显存如何规划？

由于FPGA内存有限，合理规划显存就成为了一个重要的问题。此外，合理的显存规划也有利于我们的逻辑处理和内存读写。

显然，屏幕上出现的一个字符应当被视为一个整体考虑，对它进行的操作都应当是针对该对象本身的操作，这也符合高级语言编程中，面向对象程序设计的基本理念。此外，这种方法封装了字符本身的移动细节，也更能简化对其的逻辑操作。

基于以上考虑，我们选择沿用实验11的显存规划方案，采用30\*70的屏幕划分，整个显存也就被划分成为一个30\*70的二维数组，每个数组成员为一个大小为11位的寄存器组。之所以采用十一位的寄存器组，我们使用其低八位记录屏幕的ASCII码信息，高三位记录颜色信息（我们内置了八种不同颜色）。

如此一来，除了游戏中下落的字符，诸如SCORE、MISS信息，数字以及游戏边框也都可以方便的控制其显示以及更改颜色信息。

## 2.循环如何实现？

由于Verilog硬件语言的特性，我们为了避免时序上的冲突，选择使用**信号触发**的方式完成循环代码块的设计。

我们定义了若干用于计数的信号，它们将承担类似C语言中for循环计数器i的作用。每检测到一个时钟周期上升沿，我们进行一次循环，然后将计数信号加一。这样，如果对一个长度位128的寄存器组进行遍历，我们就需要128个时钟周期上升沿来完成。只需要一个if-else语句判断即可，当计数器达到限定值，将其置零（即重新开始循环），否则计数器加一。通过这种方法，我们也可以实现多重循环嵌套，只需要多个计数器即可。这种方法可以完全避免时序冲突带来的问题。

不过，实验代码当然并非只有循环构成。在我们的逻辑控制模块中，我们需要对一个128长度的用来记录字符信息的寄存器组进行遍历，每当完成遍历，我们需要对显存进行更新。那么，当循环结束时，就不能简单地将计数器置零，而需要额外设定一个**显存更新使能位。**这一个使能信号在循环结束时被置1，通知显存读写模块可以开始进行写显存。写显存完成后，显存读写模块做好循环重新开始的准备工作，并将使能信号置0，在下一个时钟周期到来时，循环即重新开始。这样处理，也可以避免对显存的读写冲突问题。

值得一提的是，我们采用的双口ram应当可以避免上述提到的读写冲突，但多重保险总是好的。

**替代方案？**

我们曾考虑将显存读写模块和字符逻辑模块分在两个always语句中甚至分在两个文件中，以进一步封装逻辑功能，降低代码耦合性。然而，分文件意味着两个Module的IO端口都将异常庞大，这可能会带来一些问题。分always语块确实可行，但意味着更多使能控制信号的添加，这会降低代码的可读性，不利于功能扩展。在经过利弊抉择之后，我们最终选择牺牲部分代码耦合度限制，后续的功能添加进展证明了我们的选择是正确的。

## 3.读写冲突如何避免？

在上面的循环实现介绍中，我们也提到了通过开启、关闭相应的使能信号来保证显存在同一时刻只有一个模块在进行操作。我们曾经考虑使用单独的显存控制文件，进一步抽象显存读写功能，但处于代码简洁，以及避免使用过大的模块IO端口的考虑，还是将显存读写和逻辑控制放在一个文件中。

通过使能信号+双口RAM的双保险策略，我们可以完全避免显存读写冲突的发生。

5. 版本更迭

## 1. Version 1

我们首先完成了最基础的字符掉落与消去功能，已测试我们花费大量时间讨论的基础架构是否可行。十分幸运的是，这一架构在首次上机实验便宣告成功。

Version 1中，你只能通过按键消去相应的字符，制作了游戏边框，此时没有初始化界面，也没有SCORE、MISS的记录，显存还不支持添加颜色。

## 2. Version 2

在第二版中，我们修改了显存的表示。具体而言，是添加了3位的颜色位，使得游戏支持更多的色彩，同时，记录字符信息的字符寄存器组原先只需要维护一个valid位，用来判断该字符是否以被消去，现在，我们需要维护字符的三个等级。此外，我们还添加了SCORE信息显示和MISS信息显示，以及左上角显示当前按键的Target，并且新增了暂停功能。

## 3. Version 3

这一版中，我们添加四种难度等级的选择，并添加了消去字符时的“渐隐”动画效果，并添加了“重新开始”功能。

## 4. Version 4(Final Version)

最终版，我们添加了播放BGM的全套功能，得益与合理的模块划分，这些功能的添加都相对比较方便。此外，我们新增了初始化界面（Verilog 艺术字），进一步增加了游戏的颜值。

至此，整个大实验宣告完工！我们实现了设计阶段构想的所有功能！

6. 分工情况

事实上，很多代码都是我们两人在寝室或机房一边交流一边编写的（在机房当然很小声的！）。因此，很难给出一个十分清晰的分工方案，下面简单地给出了大致的分工情况：

## 刘永鹏 191220070

编写键盘、音频模块，在逻辑控制模块中主要负责字符逻辑控制部分的编写。

## 秦嘉余 191220088

编写显示器控制模块以及随机数模块，在逻辑控制模块中主要负责显存读写处理的编写。

从开始计划大实验到最终完成验收历时22天，因为时间比较零散所以不太好计算总用时，我们在12月28日完成了大实验的验收。