

# 一、实验内容

1. 本综合实验为基于 **VS1003B—MP3、COMPRQ 键盘和 VGA 显示器的 FPGA 数字华容道小游戏**。实验内容为在 VGA 显示器上的数字华容道游戏界面，其中涵盖三个游戏关卡（简单、中等、困难）和一个测试关卡，每个关卡都伴随着一首古典背景乐曲。用户可根据 COMPRQ 键盘输入按键选择将要移动的数字块，并通过 NEXYS4 开发板上的方向按键进行数字块移动方向的选择，选择的方案会显示在 VGA 显示屏上，而移动的步数会显示在 NEXYS4 开发板的七段数码管上。
2. **实验语言：** Verilog、(C++与 MATLAB 作为脚本语言)
3. **开发环境：** Vivado 2016.2、Visual Studio Code
4. **设备介绍：**

【NEXYS4 DDR 开发板】一款 Xilinx 公司开发的 Digilent 多孔现场可编程门阵列开发板。

【显示屏】VGA 接口，分辨率为 640 \* 480，刷新率为 60hz，对应时钟频率 25Mhz。

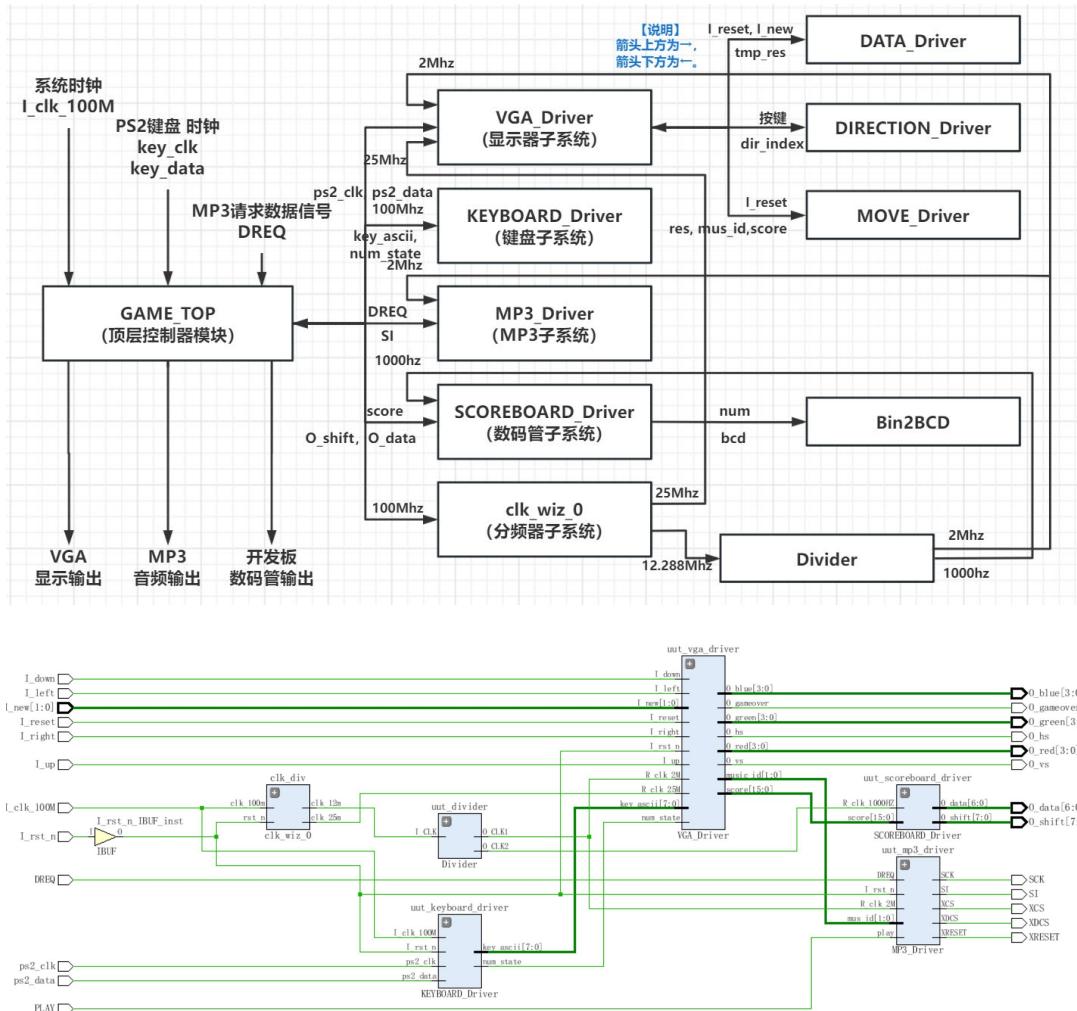
【VS1003B—MP3】具有实现 WMA 4.0/4.1/7/8/9 5-384kbps 所有流文件、MP3 和 WAV 流的音频播放功能。

【COMPRQ 键盘】康柏公司出品的普通键盘 (PS/2)，USB 接口，有线。

## 5. 游戏使用说明：

在连接完 VS1003B—MP3、COMPRQ 键盘和 VGA 显示器至 NEXYS4 开发板，并将 GAME\_TOP 的 bit 文件烧入开发板后，将 J15 引脚至 1，进入游戏。根据 H6 和 T13 引脚选择游戏关卡（难度）后，按压开发板 N17 引脚对游戏进行初始化操作。随后可在键盘的 1-8 中选择想要移动的数字方块，通过按压开发板 M18、M17、P18、P17 四个方向按键（上、右、下、左）进行数字块移动方向上的选择。若当前数字块无法移动，游戏统计步数不会增加；若当前数字块得以移动，游戏步数增加。此过程中，用户选择的数字和方向都会在 VGA 显示器上显示，用户的移动步数会在开发板的数码管上显示。若游戏闯关成功，会在 VGA 显示器右下角出现恭喜你的字样，数字华容道的外边框颜色会由红色变化为绿色，同时开发板 K15 的 LED 灯会亮起。此外，在游戏过程中可以任意的进行重置和关卡切换的操作（同样是按压 N17 按键）。将开发板 U18 引脚向上，每个关卡对应的背景音乐会随之播放，若想要重头播放音乐，可以将开发板 U18 引脚先置 0，再置 1。

## 二、数字华容道数字系统总框图



RTL 原理图

### 1. 顶层模块 (GAME\_TOP)

涉及到使用 vivado 自带的 IP 核, 将 100Mhz 的系统时钟进行分频(**clk\_wiz\_0**), 以满足各个子系统模块对不同时钟频率的要求。同时顶层模块负责对**显示器子系统**、**键盘子系统**、**MP3 子系统**和**数码管子系统**进行端口信息的交互。

### 2. 显示器子系统 (VGA\_Driver)

通过接收行同步信号、场同步信号和 RGB 颜色管脚, 以 640 \* 480 的分辨率, 主要用于实现 VGA 显示屏的图像信息输出的功能。其中包括“数字华容道”、“移动的数字是”等的文字信息, 数字华容道的游戏界面图像信息, 游戏通关后的结束信息等。同时显示器子系统与 **DATA\_Driver** (传递生成的游戏数据 **tmp\_res**)、**DIRECTION\_Driver** (传递用户选择的运动方向 **dir\_index**) 和 **MOVE\_Driver** (判断用户选择的移动方案是否可行 **res, score**) 三个子系统相互信息交互。

### **3. 键盘子系统 (KEYBOARD\_Driver)**

通过读取 PS/2 键盘的时钟脚 (ps2\_clk) 和数据脚(ps2\_data), 实现对 PS/2 键盘的键盘按键输入的读取，并将输入结果返回给 GAME\_TOP 模块。

### **4. MP3 子系统 (MP3\_Driver)**

基于 SPI 协议，在 MP3 数据请求线 (DREQ) 管脚为高电平时，将信息一位一位的传输给 SI，并通过 XCS、XDCS、SCK 端口通过 MP3 的管脚输入到 MP3 外设中，实现指令的传递和信号的输出。

### **5. 数码管子系统 (SCOREBOARD\_Driver)**

通过读取 GAME\_TOP 模块的 score 信号，通过 Bin2BCD 子模块将其转化为 BCD 码后，在开发板的数码管区域进行输出。

### **6. DATA\_Driver 子模块**

通过用户选择的关卡 (I\_new[0], I\_new[1]) 进行游戏生成数据上的选择，通过 tmp\_res 将生成数据传递给显示器子系统。

### **7. DIRECTION\_Driver 子模块**

通过用户输入按键 (I\_up, I\_down, I\_left, I\_right)，将其按键表示的方向转化为数字，通过 dir\_index 将数据传递给显示器子系统。

### **8. MOVE\_Driver 子模块**

通过来自显示器子系统的 tmp\_num\_index, tmp\_res, I\_new 等信息数据，对当前用户选择的是哪种关卡，将背景音乐转化为关卡对应的音乐，并将其以 mus\_id 的形式返回给显示器子系统（显示器子系统再向上传递至顶端模块后，顶端模块得以传输给 MP3 子系统）。同时，通过传入数据判断用户选择的操作是否可行，若可行则交换 res 的值，否则 res 不变，并通过将 res 的值返回给显示器子系统而进行数字块的变化。

### **9. BinBCD 子模块**

通过接收数码管子系统传递的步数 (num)，将其转化为 BCD 字码后，以 bcd 重新传递至数码管子系统。

### 三、系统控制器设计

MP3 系统 (其余模块并未使用到控制器):

	$Q_2$	$Q_1$	$Q_0$	$PS$	$NS$
H_RESET	0	0	1	$Q_2^n Q_1^n Q_0^n$	$Q_2^{n+1} Q_1^{n+1} Q_0^{n+1}$
S_RESET	0	1	0	$0 \ 0 \ 0$	$0 \ 0 \ 0$
SET_CLOCKF	0	1	1	$0 \ 0 \ 0$	$0 \ 0 \ 1$ $x=0$ 持续 play
SET_BASS	1	0	0	$0 \ 0 \ 1$	$0 \ 1 \ 0$
SET_VOL	1	0	1	$0 \ 1 \ 1$	$0 \ 1 \ 1$
WAIT	1	1	0	$1 \ 0 \ 0$	$1 \ 0 \ 0$
PLAY	0	0	0	$1 \ 0 \ 1$	$1 \ 1 \ 0$

状态转移图:

$$Q_2^{n+1} = \bar{Q}_2^n Q_1^n Q_0^n + Q_2^n \bar{Q}_1^n + Q_2^n Q_1^n x$$

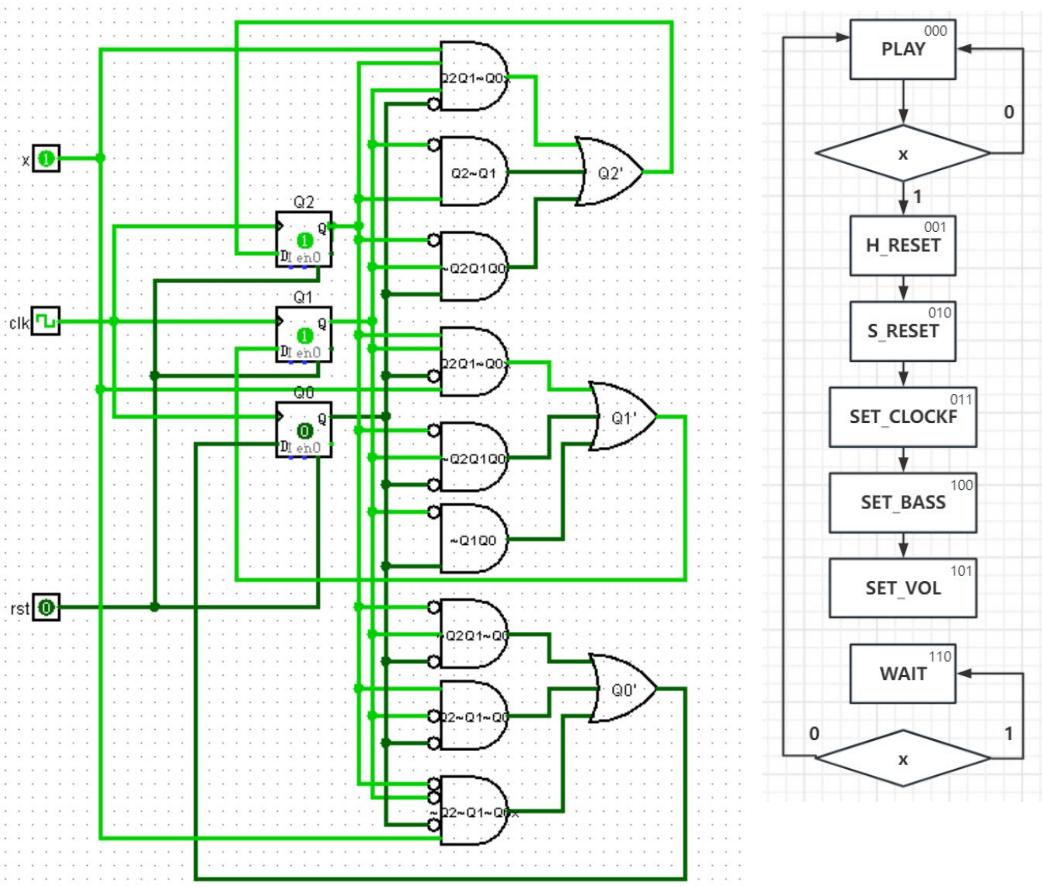
$$Q_1^{n+1} = \bar{Q}_2^n \bar{Q}_1^n Q_0^n + \bar{Q}_2^n Q_1^n \bar{Q}_0^n + \underline{\bar{Q}_2^n Q_1^n Q_0^n}$$

$$+ Q_2^n \bar{Q}_1^n \bar{Q}_0^n x$$

$$= \bar{Q}_2^n Q_0^n + Q_1^n \bar{Q}_2^n \bar{Q}_0^n x + \bar{Q}_2^n Q_1^n \bar{Q}_0^n$$

$$Q_0^{n+1} = \bar{Q}_2^n \bar{Q}_1^n \bar{Q}_0^n x + \bar{Q}_2^n Q_1^n \bar{Q}_0^n + Q_2^n \bar{Q}_1^n \bar{Q}_0^n$$

状态转移图



## 四、子系统模块建模

### 1. 顶层模块 GAME\_TOP

1) 描述：顶层模块负责调用及连接各个子模块，故定义了各个输入输出端口和数据传递的中间变量。

2) 接口信号及其定义：

接口名称	接口属性	接口描述
I_clk_100M	input	系统时钟
I_RST_N	input	复位按键（低电平有效）
I_new	input	关卡的选择
I_up	input	NEXYS4 方向按键：上
I_down	input	NEXYS4 方向按键：下
I_left	input	NEXYS4 方向按键：左
I_right	input	NEXYS4 方向按键：右
I_reset	input	NEXYS4 方向按键：游戏重置
O_red	output	VGA 红色分量
O_green	output	VGA 绿色分量
O_blue	output	VGA 蓝色分量
O_hs	output	行同步信号
O_vs	output	场同步信号
O_gameover	output	游戏是否结束
ps2_clk	input	PS/2 键盘时钟输入
ps2_data	input	PS/2 键盘数据输入
O_shift	output	第几个数码管
O_data	output	移动步数
PLAY	input	是否播放音乐
DREQ	input	MP3 数据请求线，
XCS	output	片选输入，低电平有效（SCI 传输读写指令）
XDCS	output	数据片选，字节同步（SDI 传输数据）
SCK	output	SPI 总线时钟，12.288MHZ
SI	output	声音传感器有效信号灯（传入 mp3）
XRESET	output	复位引脚（硬件复位），低电平有效

3) 调试说明：

在编写顶层模块时，尽可能地将一些共用的数据传递并储存至此，以得到更好的程序封装性。同时由于各个模块会涉及到不同的时钟分频，因此分频操作也在此进行实现。

4) Verilog 代码:

```

`timescale 1ns / 1ps

module GAME_TOP (
    input          I_clk_100M,      // 系统时钟
    input          I_rst_n,        // 复位按键（低电平有效）
    input [1: 0]   I_new,         // 关卡的选择

    /* VGA */
    input          I_up,           // NEXYS4 方向按键: 上
    input          I_down,          // NEXYS4 方向按键: 下
    input          I_left,          // NEXYS4 方向按键: 左
    input          I_right,         // NEXYS4 方向按键: 右
    input          I_reset,         // NEXYS4 方向按键: 游戏重置

    output [3: 0]  O_red,          // 红
    output [3: 0]  O_green,         // 绿
    output [3: 0]  O_blue,          // 蓝
    output          O_hs,            // 行同步信号
    output          O_vs,            // 场同步信号
    output          O_gameover,       // 游戏是否结束

    /* KEYBOARD */
    input          ps2_clk,         // PS2 键盘时钟输入
    input          ps2_data,         // PS2 键盘数据输入

    /* SCORE BOARD */
    output [7: 0]  O_shift,         // 第几个数码管
    output [6: 0]  O_data,          // 移动步数

    /* MP3 BOARD */
    input          PLAY,             // 是否播放音乐
    input          DREQ,             // MP3 数据请求线, 显示 VS1003 是否可以接受数据, 高
    // 电平可以传输数据
    output          XCS,              // 片选输入, 低电平有效 (SCI 传输读写指令)
    output          XDCS,             // 数据片选, 字节同步 (SDI 传输数据)
    output          SCK,              // SPI 总线时钟, 12.288MHZ
    output          SI,               // 声音传感器有效信号灯 (传入 mp3)
    output          XRESET,            // 复位引脚 (硬件复位), 低电平有效

);

wire R_clk_25M, R_clk_12M, R_clk_2M, R_clk_1000HZ;

wire [7:0] key_ascii;
wire key_state;
wire num_state;
wire [1:0] music_id;
wire [15: 0] score;

clk_wiz_0 clk_div (
    .clk_100m (I_clk_100M),
    .clk_25m (R_clk_25M),
    .clk_12m (R_clk_12M),
    .rst_n (I_rst_n)
);

```

```

Divider uut_divider (
    .I_CLK (R_clk_12M),
    .O_CLK1 (R_clk_2M),
    .O_CLK2 (R_clk_1000HZ)
);

VGA_Driver uut_vga_driver (
    .R_clk_25M (R_clk_25M),
    .R_clk_2M (R_clk_2M),
    .I_rst_n (I_rst_n),
    .I_up (I_up),
    .I_down (I_down),
    .I_left (I_left),
    .I_right (I_right),
    .I_reset (I_reset),
    .O_red (O_red),
    .O_green (O_green),
    .O_blue (O_blue),
    .O_hs (O_hs),
    .O_vs (O_vs),
    .O_gameover(O_gameover),
    .key_ascii (key_ascii),
    .num_state (num_state),
    .score (score),
    .I_new (I_new),
    .music_id (music_id)
);

KEYBOARD_Driver uut_keyboard_driver(
    .I_clk_100M (I_clk_100M),
    .I_rst_n (I_rst_n),
    .ps2_clk (ps2_clk),
    .ps2_data (ps2_data),
    .key_ascii (key_ascii),
    .num_state (num_state)
);

SCOREBOARD_Driver uut_scoreboard_driver(
    .R_clk_1000HZ (R_clk_1000HZ),
    .score (score),
    .O_shift (O_shift),
    .O_data (O_data)
);

MP3_Driver uut_mp3_driver (
    .R_clk_2M (R_clk_2M),
    .I_rst_n (I_rst_n),
    .play (PLAY),
    .mus_id (music_id),
    .DREQ (DREQ),
    .XCS (XCS),
    .XDCS (XDCS),
    .SCK (SCK),
    .SI (SI),
    .XRESET (XRESET)
);

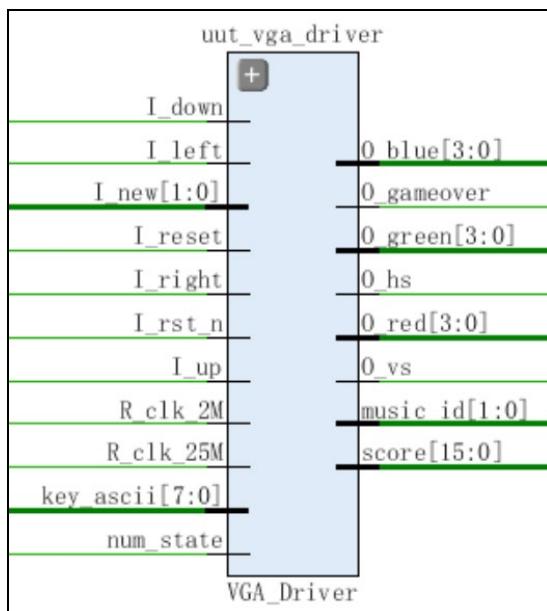
endmodule

```

## 2. 显示器子系统 VGA\_Driver

1) 描述：此综合实验主干部分，涵盖整个数字华容道游戏的处理逻辑。此系统得以实现游戏界面的输出（文字与图像信息），游戏运行的动态画面以及通过实例化 DATA\_Driver、DIRECTION\_Driver 和 MOVE\_Driver 三个模块以及顶端模块传递的 KEYBOARD\_Driver 子系统来达到生成游戏数据，读取用户选择的数字和方向以及判断是否得以移动数字块的功能。此外，显示器子系统还会向顶端子系统传输 score, mus\_id 等游戏运行信息数据。

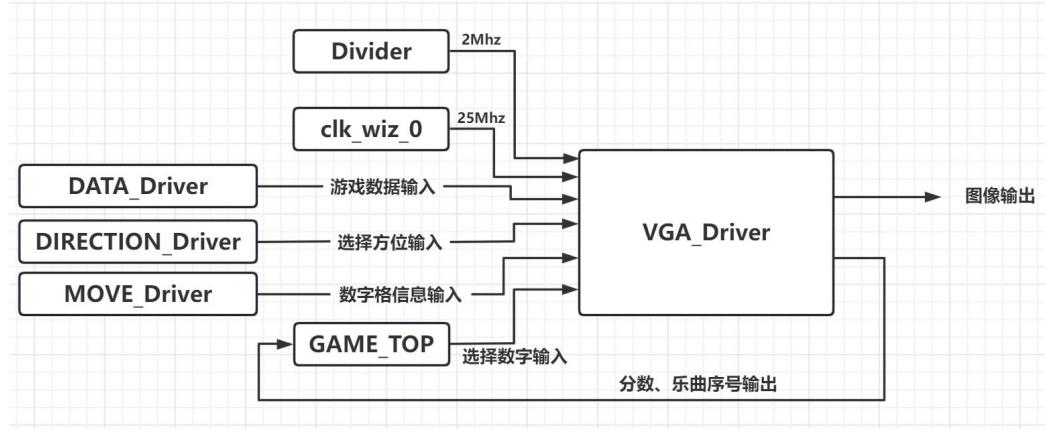
2) 接口信号及其定义：



接口名称	接口属性	接口描述
<code>R_clk_25M</code>	input	频率为 25Mhz 的时钟
<code>R_clk_2M</code>	input	频率为 2Mhz 的时钟
<code>I_rst_n</code>	input	复位按键（低电平有效）
<code>I_new</code>	input	关卡的选择
<code>I_up</code>	input	NEXYS4 方向按键：上
<code>I_down</code>	input	NEXYS4 方向按键：下
<code>I_left</code>	input	NEXYS4 方向按键：左
<code>I_right</code>	input	NEXYS4 方向按键：右
<code>I_reset</code>	input	NEXYS4 方向按键：游戏重置
<code>O_red</code>	output	VGA 红色分量
<code>O_green</code>	output	VGA 绿色分量
<code>O_blue</code>	output	VGA 蓝色分量
<code>O_hs</code>	output	行同步信号

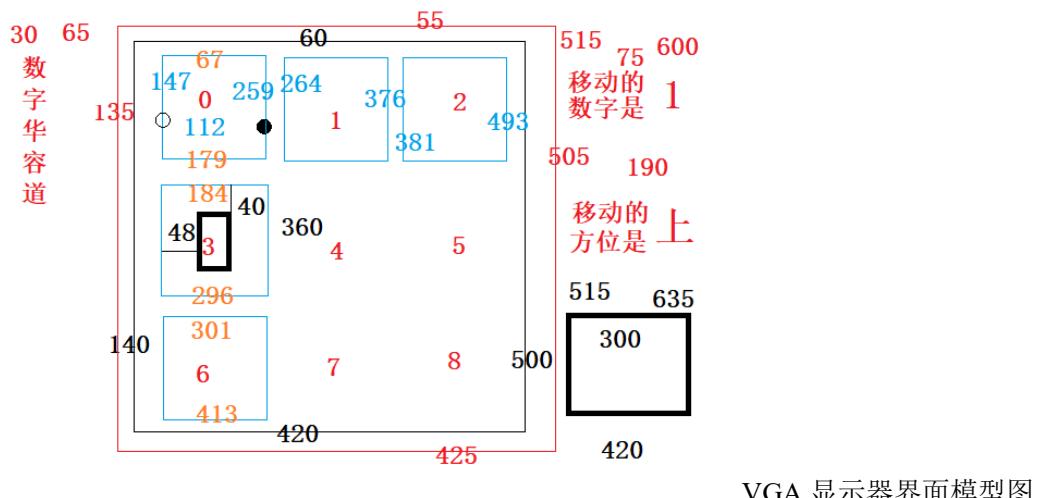
<b>O_vs</b>	output	场同步信号
<b>O_gameover</b>	output	游戏是否结束
<b>key_ascii</b>	input	键盘读取
<b>num_state</b>	input	是否有符合要求的键盘按键输入
<b>score</b>	output	游戏分数
<b>music_id</b>	output	当前关卡对应的乐曲序号

### 3) 结构框图:



### 4) 调试说明:

此子系统是整个综合实验的**主干部分**,整体实现过程是通过以整化分的方法将其不断进行拆分和化简。首先,构思的是游戏的整体布局,这里采用了**画图**软件进行文字与图像输出样式上的排版布局,在此过程中也对显示器子系统所需要实现的功能有了一个大致的规划。在草图绘制完毕后,便进行了参数的计算与处理(如下),并在代码中通过 parameter 语句对其进行统一管理,使得在后续代码的编写中,便于删改。接着在 VGA 显示器针对文字与图像初步调试成功的基础上(可见后文的 TestBench 说明),开始对文字与图像输出的绘制工作。



在绘制工作中需要注意的是 VGA 显示器扫描方式是从屏幕左上角一点开始，从左向右逐点扫描，每扫描完一行，电子束回到屏幕的左边下一行的起始位置，每行结束时，用行同步信号进行同步。当扫描完所有的行，形成一帧，用场同步信号进行场同步，并使扫描回到屏幕左上方，开始下一帧。因此，通过 if-else 语句对整个显示屏输出界面（H\_ACTIVE\_TIME 和 V\_ACTIVE\_TIME）进行区域的划分是较好的一种解决方式。同时，由于 VGA 的扫描与输出是以像素点为单位，因此在区域划分时，**某个边界能否取到**也是需要着重进行考虑的，如上图第一个数字块的左边不取，右边取，从而实现数字块的边长为 112 个像素点。

此外，在输出颜色时，由于此 VGA 支持的是 12 位深度颜色的输出（RGB 各四位），因此不可以使用普遍的 24 位 RGB 颜色表进行 VGA 颜色分量赋值。同时，在调试的过程中可以通过输出不同颜色来观察所划定的区域是否正确。在输出文字时，采用的是使用**线网型**变量来进行对文字坐标的锁定，并将 2 进制字模点阵信息存储在 reg 二维变量数组中。**在这里需要注意的是，Verilog 语法中二维数组变量名前的区间范围表示的是二维数组的列数，在变量名后的区间范围表示的是二维数组的行数**，若以 C++ 语法错误理解会在行列不相等的情况下无法完整输出文字信息。同样，在后续判断字符数组某个坐标是否为 1 时，也要将以 **char[char\_y][char\_x]** 进行元素的索取，否则出现的字符会进行旋转。**数组的大小也需要根据文字字模点阵的大小进行修改**，否则同样无法输出完整文字信息。（看似简单，但在编程过程中，多次犯了这样的错误）而在输出字符的 if-else 区域判断条件上，**对 h\_ent 的范围需要向前推一个（提前一个时钟节拍）**，因为 VGA 的颜色分量是**时序逻辑**，在对 RGB 分量进行赋值时，会滞后坐标一个时钟节拍。此外，在编写字模点阵信息时，自己编写了几个 C++ 的脚本来化简工作量。

```
string s[24];
string t;

int main()
{
    ofstream fout;
    fout.open("phrase.txt");
    for (int j = 0; j < 3; ++ j) {
        for (int i = 0; i < 24; ++ i) {
            cin >> t;
            s[i] += t;
        }
    }

    for (int i = 0; i < 24; ++ i) {
        fout << s[i] << endl;
    }
    fout.close();
    return 0;
}
```

```
string s;
int main()
{
    ofstream fout;
    fout.open("abc.txt");
    for(int i = 0; i < 32; ++ i) {
        cin >> s;
        fout << "char4[" << i << "] <= 128'h" << s << ";" << endl;
    }
    fout.close();
    return 0;
}
```

部分 C++ 脚本代码

由于文字输出采用的是将字模点阵的 2 进制信息直接存储在.v 代码模块中，在图像的输出上选择将图像的 2 进制信息存储到 ROM 的 IP 核中。在此过程中需要注意的是，需要对 24-bit 真彩色图像进行颜色数据的重组与拼接。核心思路为将通过 matlab 中 image\_array 和 reshape 处理后的 8bit 红色分量右移 4 位取出高 4 位，然后左移 8 位，作为 ROM 中 RGB 数据的第 11-bit 到第 8-bit（绿色分量和蓝色分量同样为右移 4 位取高 4 位，并分别左移 4 位和 0 位），核心代码如下：

```
for i = 1 : height * width
    rgb(i) = bitshift(bitshift(r(i), -4), 8) + bitshift(bitshift(g(i), -4), 4) + bitshift(bitshift(b(i), -4), 0);
end
```

在图像输出的过程中，需要注意对存储 ROM 的地址的变量 R\_rom\_addr 的维护，经过 TestBench 测试发现，若在不输出图像的“else”情形下，R\_rom\_addr 有出现随机值的可能性。

同时针对游戏的内部实现逻辑，考虑使用一个二维数组存储每一个方格中的数字进行维护，并将空格赋值为 0。当数组元素从 1-9，分别为 1, 2, 3, 4, 5, 6, 7, 8, 0 时，即为闯关成功。对于用户选择的操作判断，则通过 if-else 选择语句一一判断是否可行，若可行则交换（具体说明可见 MOVE\_Driver 子模块）。如此设想构建完成后，发现其中存在漏洞：先前我用着“123456708”的数据进行代码上的构思，即用每个数组元素的值来判断用户能否移动滑块，但是在游戏过程中 1 号方格存的数不可能一直是 1，而用户想移动的 1，也并不是 1 号方格，而是在数字华容道的方格中数字为 1 的那个方格，因此还需要增加一个 always 模块，用来判断用户真正想要移动的方格是哪个。

## 5) Verilog 代码：

```
`timescale 1ns / 1ps

module VGA_Driver(
    input          R_clk_25M,      // 25mhz
    input          R_clk_2M,
    input          I_rst_n,        // 系统复位

    input          I_up,
    input          I_down,
    input          I_left,
    input          I_right,
    input          I_reset,
    output reg [3: 0] O_red,       // VGA 红色分量
    output reg [3: 0] O_green,     // VGA 绿色分量
```

```

output reg [3: 0]      o_blue,           // VGA 蓝色分量
output                 o_hs,            // VGA 行同步信号
output                 o_vs,            // VGA 场同步信号
output reg              o_gameover,       // 游戏结束信号

input      [7: 0]      key_ascii,
input                  num_state,
output     [15: 0]      score,
input      [1: 0]       I_new,
output     [1: 0]       music_id
);

// 分辨率为 640*480 时行时序各个参数定义
parameter   H_SYNC_PULSE    = 96,
            H_BACK_PORCH   = 48,
            H_ACTIVE_TIME   = 640,
            H_FRONT_PORCH   = 16,
            H_LINE_PERIOD    = 800;

// 分辨率为 640*480 时场时序各个参数定义
parameter   V_SYNC_PULSE    = 2,
            V_BACK_PORCH   = 33,
            V_ACTIVE_TIME   = 480,
            V_FRONT_PORCH   = 10,
            V_FRAME_PERIOD   = 525;

// 行列时序计数器
reg [11:0]    h_cnt;          // 行
reg [11:0]    v_cnt;          // 列

wire          active_flag;    // 激活标志, 当这个信号为 1 时 RGB 的数据可以显示在屏幕上

// 内边框坐标
parameter   B_X1    = 140,
            B_X2    = 500,
            B_Y1    = 60,
            B_Y2    = 420;

// 外边框坐标
parameter   B_X_MIN   = 135,
            B_X_MAX   = 505,
            B_Y_MIN   = 55,
            B_Y_MAX   = 425;

```

```

parameter      CELL          = 112; // 边框大小

// Gameover 图像信息
parameter      C_IMAGE_WIDTH     = 120,
               C_IMAGE_HEIGHT    = 120,
               C_IMAGE_PIX_NUM   = 120 * 120;
reg [13:0]      R_rom_addr = 14'b0; // ROM 的地址
wire[11:0]      W_rom_data;        // ROM 中存储的数据

// 文字: 数字华容道
parameter      CHAR_W       = 12'd64, // 字符宽度
               CHAR_H       = 12'd336; // 字符深度
parameter      CHAR_B_H     = H_SYNC_PULSE + H_BACK_PORCH + 30,
               CHAR_B_V     = V_SYNC_PULSE + V_BACK_PORCH + 65;
wire[11:0]      char_x;        // 字符横坐标
wire[11:0]      char_y;        // 字符纵坐标

// 文字: 移动的数字是
parameter      CHAR_W1      = 12'd72,
               CHAR_H1      = 12'd50;
parameter      CHAR_B_H1    = H_SYNC_PULSE + H_BACK_PORCH + 515,
               CHAR_B_V1    = V_SYNC_PULSE + V_BACK_PORCH + 75;
wire [11:0]    char_x1;
wire [11:0]    char_y1;

// 文字: 移动的方位是
parameter      CHAR_W2      = 12'd72,
               CHAR_H2      = 12'd50;
parameter      CHAR_B_H2    = H_SYNC_PULSE + H_BACK_PORCH + 515,
               CHAR_B_V2    = V_SYNC_PULSE + V_BACK_PORCH + 190;
wire [11:0]    char_x2;
wire [11:0]    char_y2;

// 文字: 具体数字是
parameter      CHAR_W3      = 12'd16,
               CHAR_H3      = 12'd32;
parameter      CHAR_B_H3    = H_SYNC_PULSE + H_BACK_PORCH + 605,
               CHAR_B_V3    = V_SYNC_PULSE + V_BACK_PORCH + 80;
wire [11:0]    char_x3;
wire [11:0]    char_y3;

// 数字: 华容道数字
parameter      NUM_X       = 48,
               NUM_Y       = 40;

```

```

parameter      CHAR_B_H_1  = H_SYNC_PULSE + H_BACK_PORCH + NUM_X,
              CHAR_B_V_1  = V_SYNC_PULSE + V_BACK_PORCH + NUM_Y;
wire    [11:0]  char_x_1;
wire    [11:0]  char_y_1;
// ... 共 9 个, 具体可看源码
wire    [11:0]  char_x_9;
wire    [11:0]  char_y_9;
reg     [9:0]   cell_x  [8:0];
reg     [9:0]   cell_y  [8:0];

// 文字: 具体的方位
parameter      CHAR_W4      = 12'd32,
              CHAR_H4      = 12'd32;
parameter      CHAR_B_H4    = H_SYNC_PULSE + H_BACK_PORCH + 600,
              CHAR_B_V4    = V_SYNC_PULSE + V_BACK_PORCH + 202;
wire    [11:0]  char_x4;
wire    [11:0]  char_y4;

reg     [7:0]   num_index;

// 存储的字符码
reg     [63:0]  char     [335:0];
reg     [71:0]  char1    [49:0];
reg     [71:0]  char2    [49:0];
reg     [127:0] char3    [31:0];
reg     [127:0] char4    [31:0];

wire    [3:0]   res      [9:0];
wire    [3:0]   tmp_res  [9:0];

// 选择关卡
DATA_Driver uut_date_driver (
  .I_new (I_new),
  .tmp_res1 (tmp_res[1]),
  .tmp_res2 (tmp_res[2]),
  .tmp_res3 (tmp_res[3]),
  .tmp_res4 (tmp_res[4]),
  .tmp_res5 (tmp_res[5]),
  .tmp_res6 (tmp_res[6]),
  .tmp_res7 (tmp_res[7]),
  .tmp_res8 (tmp_res[8]),
  .tmp_res9 (tmp_res[9])
);

```

```

// 初始化格子坐标
always @(posedge R_clk_25M) begin
    cell_x[0] <= 10'd147;    cell_y[0] <= 10'd67;
    cell_x[1] <= 10'd264;    cell_y[1] <= 10'd67;
    cell_x[2] <= 10'd381;    cell_y[2] <= 10'd67;
    cell_x[3] <= 10'd147;    cell_y[3] <= 10'd184;
    cell_x[4] <= 10'd264;    cell_y[4] <= 10'd184;
    cell_x[5] <= 10'd381;    cell_y[5] <= 10'd184;
    cell_x[6] <= 10'd147;    cell_y[6] <= 10'd301;
    cell_x[7] <= 10'd264;    cell_y[7] <= 10'd301;
    cell_x[8] <= 10'd381;    cell_y[8] <= 10'd301;
end

// 判断游戏是否结束
always @(posedge R_clk_25M) begin
    if ((res[1] == 4'd1) && (res[2] == 4'd2)
        && (res[3] == 4'd3) && (res[4] == 4'd4)
        && (res[5] == 4'd5) && (res[6] == 4'd6)
        && (res[7] == 4'd7) && (res[8] == 4'd8)
        && (res[9] == 4'd0))
        O_gameover <= 1'b1;
    else
        O_gameover <= 1'b0;
end

reg [7:0] tmp_index;

// 读取键盘选择的按键
always @(posedge R_clk_25M) begin
    if (key_ascii) begin
        num_index <= key_ascii - 8'h30;
        tmp_index <= key_ascii - 8'h30;
    end
    if (num_state) begin
        num_index <= tmp_index;
    end
end

wire [2:0] dir_index;

// 读取方向
DIRECTION_Driver uut_direction_driver (
    .I_up (I_up),
    .I_down (I_down),

```

```

.I_left (I_left),
.I_right (I_right),
.O_gameover (O_gameover),
.dir_index (dir_index)
);

///////////////////////////////
//char_x; char_y - 数字华容道
assign char_x = (h_cnt >= CHAR_B_H) && (h_cnt < CHAR_B_H + CHAR_W)
    && (v_cnt >= CHAR_B_V) && (v_cnt < CHAR_B_V + CHAR_H)
    ? (64 - h_cnt + CHAR_B_H) : 10'h3ff;
assign char_y = (h_cnt >= CHAR_B_H) && (h_cnt < CHAR_B_H + CHAR_W)
    && (v_cnt >= CHAR_B_V) && (v_cnt < CHAR_B_V + CHAR_H)
    ? (v_cnt - CHAR_B_V) : 10'h3ff;

///////////////////////////////
//char_x1; char_y1 - 移动的数字是
assign char_x1 = (h_cnt >= CHAR_B_H1) && (h_cnt < CHAR_B_H1 + CHAR_W1)
    && (v_cnt >= CHAR_B_V1) && (v_cnt < CHAR_B_V1 + CHAR_H1)
    ? (72 - h_cnt + CHAR_B_H1) : 10'h3ff;
assign char_y1 = (h_cnt >= CHAR_B_H1) && (h_cnt < CHAR_B_H1 + CHAR_W1)
    && (v_cnt >= CHAR_B_V1) && (v_cnt < CHAR_B_V1 + CHAR_H1)
    ? (v_cnt - CHAR_B_V1) : 10'h3ff;

///////////////////////////////
//char_x2; char_y2 - 移动的方位是
assign char_x2 = (h_cnt >= CHAR_B_H2) && (h_cnt < CHAR_B_H2 + CHAR_W2)
    && (v_cnt >= CHAR_B_V2) && (v_cnt < CHAR_B_V2 + CHAR_H2)
    ? (72 - h_cnt + CHAR_B_H2) : 10'h3ff;
assign char_y2 = (h_cnt >= CHAR_B_H2) && (h_cnt < CHAR_B_H2 + CHAR_W2)
    && (v_cnt >= CHAR_B_V2) && (v_cnt < CHAR_B_V2 + CHAR_H2)
    ? (v_cnt - CHAR_B_V2) : 10'h3ff;

///////////////////////////////
//char_x3; char_y3 - 具体数字
assign char_x3 = (h_cnt >= CHAR_B_H3) && (h_cnt < CHAR_B_H3 + CHAR_W3)
    && (v_cnt >= CHAR_B_V3) && (v_cnt < CHAR_B_V3 + CHAR_H3)
    && (key_ascii) // && (num_ascii)
    ? (128 - h_cnt + CHAR_B_H3) : 10'h3ff;
assign char_y3 = (h_cnt >= CHAR_B_H3) && (h_cnt < CHAR_B_H3 + CHAR_W3)
    && (v_cnt >= CHAR_B_V3) && (v_cnt < CHAR_B_V3 + CHAR_H3)
    ? (v_cnt - CHAR_B_V3) : 10'h3ff;

/////////////////////////////

```

```

//char_x4; char_y4 - 具体方位
assign char_x4 = (h_cnt >= CHAR_B_H4) && (h_cnt < CHAR_B_H4 + CHAR_W4)
    && (v_cnt >= CHAR_B_V4) && (v_cnt < CHAR_B_V4 + CHAR_H4)
    ? (128 - h_cnt + CHAR_B_H4) : 10'h3ff;
assign char_y4 = (h_cnt >= CHAR_B_H4) && (h_cnt < CHAR_B_H4 + CHAR_W4)
    && (v_cnt >= CHAR_B_V4) && (v_cnt < CHAR_B_V4 + CHAR_H4)
    ? (v_cnt - CHAR_B_V4) : 10'h3ff;

///////////////////////////////
//char_x_1; char_y_1 -- 每个格子的数字
assign char_x_1 = (h_cnt >= CHAR_B_H_1 + cell_x[0]) && (h_cnt < CHAR_B_H_1 +
cell_x[0] + CHAR_W3)
    && (v_cnt >= CHAR_B_V_1 + cell_y[0]) && (v_cnt < CHAR_B_V_1 +
cell_y[0] + CHAR_H3)
    && (res[1])
    ? (128 - h_cnt +CHAR_B_H_1 + cell_x[0] + (9-res[1])*16) :
10'h3ff;
assign char_y_1 = (h_cnt >= CHAR_B_H_1 + cell_x[0]) && (h_cnt < CHAR_B_H_1 +
cell_x[0] + CHAR_W3)
    && (v_cnt >= CHAR_B_V_1 + cell_y[0]) && (v_cnt < CHAR_B_V_1 +
cell_y[0] + CHAR_H3)
    ? (v_cnt - (CHAR_B_V_1 + cell_y[0])) :
10'h3ff;
// ... 共 9 个, 具体可看源码
//char_x_9; char_y_9
assign char_x_9 = (h_cnt >= CHAR_B_H_1 + cell_x[8]) && (h_cnt < CHAR_B_H_1 +
cell_x[8] + CHAR_W3)
    && (v_cnt >= CHAR_B_V_1 + cell_y[8]) && (v_cnt < CHAR_B_V_1 +
cell_y[8] + CHAR_H3)
    && (res[9])
    ? (128 - h_cnt +CHAR_B_H_1 + cell_x[8] + (9-res[9])*16) :
10'h3ff;
assign char_y_9 = (h_cnt >= CHAR_B_H_1 + cell_x[8]) && (h_cnt < CHAR_B_H_1 +
cell_x[8] + CHAR_W3)
    && (v_cnt >= CHAR_B_V_1 + cell_y[8]) && (v_cnt < CHAR_B_V_1 +
cell_y[8] + CHAR_H3)
    ? (v_cnt - (CHAR_B_V_1 + cell_y[8])) : 10'h3ff;

// 判断移动的是哪个格子
reg [3:0] tmp_num_index;
integer i;
always @(posedge R_clk_25M) begin // ①
if ((num_index <=8) && (num_index >=1)) begin
    for (i = 1; i <= 9; i = i + 1) begin

```

```

        if (res[i] == num_index)
            tmp_num_index = i;
    end
end
end

// 判断能否移动
MOVE_Driver uut_move_driver(
    .R_clk_2M (R_clk_2M),
    .I_reset (I_reset),
    .I_up (I_up),
    .I_down (I_down),
    .I_left (I_left),
    .I_right (I_right),
    .tmp_num_index (tmp_num_index),
    .tmp_res1 (tmp_res[1]),
    .tmp_res2 (tmp_res[2]),
    .tmp_res3 (tmp_res[3]),
    .tmp_res4 (tmp_res[4]),
    .tmp_res5 (tmp_res[5]),
    .tmp_res6 (tmp_res[6]),
    .tmp_res7 (tmp_res[7]),
    .tmp_res8 (tmp_res[8]),
    .tmp_res9 (tmp_res[9]),
    .I_new (I_new),
    .O_gameover (O_gameover),
    .res1 (res[1]),
    .res2 (res[2]),
    .res3 (res[3]),
    .res4 (res[4]),
    .res5 (res[5]),
    .res6 (res[6]),
    .res7 (res[7]),
    .res8 (res[8]),
    .res9 (res[9]),
    .mus_id (music_id),
    .score (score)
);

// 游戏结束照片
photo gameover (
    .clka (R_clk_25M),      // input wire clka
    .addr (R_rom_addr),    // input wire [13 : 0] addr
    .douta (W_rom_data)   // output wire [11 : 0] douta

```

```

);

// 数字华容道
always @(posedge R_clk_25M) begin
    char[0]  <= 64'h0000000000000000;
    char[1]  <= 64'h0000000000000000;
    // ... 略 具体可看源码
    char[62]  <= 64'h0000000000000000;
    char[63]  <= 64'h0000000000000000; // 数
    // ... 略 具体可看源码
    char[130]  <= 64'h0000000000000000;
    char[131]  <= 64'h0000000000000000; // 字
    // ... 略 具体可看源码
    char[198]  <= 64'h0000000000000000;
    char[199]  <= 64'h0000000000000000; // 华
    // ... 略 具体可看源码
    char[266]  <= 64'h0000000000000000;
    char[267]  <= 64'h0000000000000000; // 容
    // ... 略 具体可看源码
    char[334]  <= 64'h0000000000000000;
    char[335]  <= 64'h0000000000000000; // 道
end
// 移动的数字是
always @(posedge R_clk_25M) begin
    char1[0]  <= 72'h0000000000000000;
    char1[1]  <= 72'h00000000000020100;
    // ... 略 具体可看源码
    char1[49]  <= 72'h0000000000000000;
end
// 移动的方位是
always @(posedge R_clk_25M) begin
    char2[0]  <= 72'h0000000000000000;
    char2[1]  <= 72'h00000000000020100;
    // ... 略 具体可看源码
    char2[49]  <= 72'h0000000000000000;
end
// 数字库
always @(posedge R_clk_25M) begin
    char3[0]  <= 128'h000000000000000000000000000000000000000000000000000000000000000;
    char3[1]  <= 128'h000000000000000000000000000000000000000000000000000000000000000;
    // ... 略 具体可看源码
    char3[31]  <= 128'h000000000000000000000000000000000000000000000000000000000000000;
end
// 方位库

```

```

always @(posedge R_clk_25M) begin
    char4[0]  <= 128'h00000000000000000000000000000000;
    char4[1]  <= 128'h00000000000000000000000000000000;
    // ... 略 具体可看源码
    char4[31] <= 128'h00000000000000000000000000000000;
end

// 功能: 产生行时序
always @(posedge R_clk_25M or negedge I_rst_n) begin
    if (!I_rst_n)
        h_cnt <= 12'd0;
    else if (h_cnt == H_LINE_PERIOD - 1'b1)
        h_cnt <= 12'd0;
    else
        h_cnt <= h_cnt + 1'b1;
end
assign O_hs = (h_cnt < H_SYNC_PULSE) ? 1'b0 : 1'b1;

// 功能: 产生场时序
always @(posedge R_clk_25M or negedge I_rst_n) begin
    if (!I_rst_n)
        v_cnt <= 12'd0 ;
    else if (v_cnt == V_FRAME_PERIOD - 1'b1)
        v_cnt <= 12'd0 ;
    else if (h_cnt == H_LINE_PERIOD - 1'b1)
        v_cnt <= v_cnt + 1'b1 ;
    else
        v_cnt <= v_cnt ;
end
assign O_vs = (v_cnt < V_SYNC_PULSE) ? 1'b0 : 1'b1;

// 功能: 输出场同步和行同步
assign active_flag = (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH))
&& (h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH +
H_ACTIVE_TIME))
&& (v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH))
&& (v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH +
V_ACTIVE_TIME));

// 功能: 输出数字华容道格子
always @(posedge R_clk_25M or negedge I_rst_n) begin
    if (!I_rst_n) begin
        O_red      <= 4'b0000;
        O_green    <= 4'b0000;

```

```

    O_blue      <= 4'b0000;
    R_rom_addr <= 14'd0 ;
end
else if (active_flag) begin
    // 红色外边框
    if (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH + B_X_MIN)
        && h_cnt < (H_SYNC_PULSE + H_BACK_PORCH + B_X1)
        && v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MIN)
        && v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MAX)) begin
        if (!O_gameover) begin
            O_red    <= 4'b1111;
            O_green  <= 4'b0000;
            O_blue   <= 4'b0000;
        end
        else begin
            O_red    <= 4'b0000;
            O_green  <= 4'b1111;
            O_blue   <= 4'b0000;
        end
    end
    else if (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH + B_X1)
        && h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH + B_X2)
        && v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MIN)
        && v_cnt < (V_SYNC_PULSE + V_BACK_PORCH + B_Y1)) begin
        if (!O_gameover) begin
            O_red    <= 4'b1111;
            O_green  <= 4'b0000;
            O_blue   <= 4'b0000;
        end
        else begin
            O_red    <= 4'b0000;
            O_green  <= 4'b1111;
            O_blue   <= 4'b0000;
        end
    end
    else if (h_cnt > (H_SYNC_PULSE + H_BACK_PORCH + B_X2)
        && h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH + B_X_MAX)
        && v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MIN)
        && v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MAX)) begin
        if (!O_gameover) begin
            O_red    <= 4'b1111;
            O_green  <= 4'b0000;
            O_blue   <= 4'b0000;
        end
    end

```

```

        else begin
            O_red    <= 4'b0000;
            O_green <= 4'b1111;
            O_blue   <= 4'b0000;
        end
    end
    else if (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH + B_X1)
        && h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH + B_X2)
        && v_cnt > (V_SYNC_PULSE + V_BACK_PORCH + B_Y2)
        && v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH + B_Y_MAX)) begin
        if (!O_gameover) begin
            O_red    <= 4'b1111;
            O_green <= 4'b0000;
            O_blue   <= 4'b0000;
        end
        else begin
            O_red    <= 4'b0000;
            O_green <= 4'b1111;
            O_blue   <= 4'b0000;
        end
    end
    // 内边框
    else if (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH + B_X1)
        && h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH + B_X2)
        && v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH + B_Y1)
        && v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH + B_Y2)) begin
        if ((h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + cell_x[0])
            || (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + cell_x[1])
            && (h_cnt > H_SYNC_PULSE + H_BACK_PORCH + cell_x[0] + CELL))
            || (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + cell_x[2])
            && (h_cnt > H_SYNC_PULSE + H_BACK_PORCH + cell_x[1] + CELL))
            || (h_cnt > H_SYNC_PULSE + H_BACK_PORCH + cell_x[2] + CELL)
            || (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + cell_y[0])
            || (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + cell_y[3])
            && (v_cnt > V_SYNC_PULSE + V_BACK_PORCH + cell_y[0] +
CELL))
            || (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + cell_y[6])
            && (v_cnt > V_SYNC_PULSE + V_BACK_PORCH + cell_y[3] + CELL))
            || (v_cnt > V_SYNC_PULSE + V_BACK_PORCH + cell_y[6] + CELL))
        begin
            O_red    <= 4'b0000;
            O_green <= 4'b0000;
            O_blue   <= 4'b1111;
        end
    end

```

```

else if ((h_cnt > H_SYNC_PULSE + H_BACK_PORCH + cell_x[0] + 45)
  && (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + cell_x[0] + 65)
  && (v_cnt > V_SYNC_PULSE + V_BACK_PORCH + cell_y[0] + 35)
  && (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + cell_y[0] + 70))
begin
  if ((h_cnt > CHAR_B_H_1 + cell_x[0] - 1)
    && (h_cnt <= CHAR_B_H_1 + cell_x[0] + CHAR_W3 - 1)
    && (v_cnt >= CHAR_B_V_1 + cell_y[0]))
    && (v_cnt < CHAR_B_V_1 + cell_y[0] + CHAR_H3)
    && (char3[char_y_1][char_x_1] == 1'b1)) begin // 第一个空格
      O_red      <= 4'b0000;
      O_green    <= 4'b0000;
      O_blue     <= 4'b0000;
    end
  else begin // 米褐色背景
    O_red      <= 4'b1111;
    O_green    <= 4'b0110;
    O_blue     <= 4'b0101;
  end
end
// ... 共 9 个, 具体可看源码
else if ((h_cnt > H_SYNC_PULSE + H_BACK_PORCH + cell_x[8] + 45)
  && (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + cell_x[8] + 65)
  && (v_cnt > V_SYNC_PULSE + V_BACK_PORCH + cell_y[8] + 35)
  && (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + cell_y[8] + 70))
begin
  if ((h_cnt > CHAR_B_H_1 + cell_x[8] - 1)
    && (h_cnt <= CHAR_B_H_1 + cell_x[8] + CHAR_W3 - 1)
    && (v_cnt >= CHAR_B_V_1 + cell_y[8]))
    && (v_cnt < CHAR_B_V_1 + cell_y[8] + CHAR_H3)
    && (char3[char_y_9][char_x_9] == 1'b1)) begin // 第九个空格
      O_red      <= 4'b0000;
      O_green    <= 4'b0000;
      O_blue     <= 4'b0000;
    end
  else begin
    O_red      <= 4'b1111;
    O_green    <= 4'b0110;
    O_blue     <= 4'b0101;
  end
end
else begin // 米褐色
  O_red      <= 4'b1111;
  O_green    <= 4'b0110;

```

```

        O_blue  <=  4'b0101;
    end
end
else if ((h_cnt >= H_SYNC_PULSE + H_BACK_PORCH + 30)
    &&  (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + 95)
    &&  (v_cnt >= V_SYNC_PULSE + V_BACK_PORCH + 65)
    &&  (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + 420)) begin
if  ((h_cnt >= CHAR_B_H - 1) && (h_cnt < CHAR_B_H + CHAR_W - 1)
&&  (v_cnt >= CHAR_B_V)      && (v_cnt < CHAR_B_V + CHAR_H)
&&  (char[char_y][char_x] == 1'b1)) begin // 数字华容道
    O_red   <=  4'b0000; // 青色
    O_green <=  4'b1111;
    O_blue  <=  4'b1111;
end
else begin // 黑色
    O_red   <=  4'b0000;
    O_green <=  4'b0000;
    O_blue  <=  4'b0000;
end
end
else if ((h_cnt >= H_SYNC_PULSE + H_BACK_PORCH + 510)
    &&  (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + 585)
    &&  (v_cnt >= V_SYNC_PULSE + V_BACK_PORCH + 75)
    &&  (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + 125)) begin
if  ((h_cnt >= CHAR_B_H1 - 1) && (h_cnt < CHAR_B_H1 + CHAR_W1 - 1)
&&  (v_cnt >= CHAR_B_V1)      && (v_cnt < CHAR_B_V1 + CHAR_H1)
&&  (char1[char_y1][char_x1] == 1'b1)) begin // 移动的数字是
    O_red   <=  4'b1111; // 白色
    O_green <=  4'b1111;
    O_blue  <=  4'b1111;
end
else begin // 黑色
    O_red   <=  4'b0000;
    O_green <=  4'b0000;
    O_blue  <=  4'b0000;
end
end
else if ((h_cnt >= H_SYNC_PULSE + H_BACK_PORCH + 600)
    &&  (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + 630)
    &&  (v_cnt >= V_SYNC_PULSE + V_BACK_PORCH + 80)
    &&  (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + 115)) begin
if  ((h_cnt >= CHAR_B_H3 - 1) && (h_cnt < CHAR_B_H3 + CHAR_W3 - 1)
&&  (v_cnt >= CHAR_B_V3)      && (v_cnt < CHAR_B_V3 + CHAR_H3)
//  && (num_ascii >= 1) && (num_ascii <= 8)

```

```

    && (char3[char_y3][char_x3 + 16 * (9 - num_index)] == 1'b1))

begin // 具体数字为
    O_red   <= 4'b1111; // 白色
    O_green <= 4'b1111;
    O_blue  <= 4'b1111;
end
else begin // 灰色
    O_red   <= 4'b0011;
    O_green <= 4'b0011;
    O_blue  <= 4'b0011;
end
end

else if ((h_cnt >= H_SYNC_PULSE + H_BACK_PORCH + 510)
    && (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + 585)
    && (v_cnt >= V_SYNC_PULSE + V_BACK_PORCH + 190)
    && (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + 245)) begin
if ((h_cnt >= CHAR_B_H2 - 1) && (h_cnt < CHAR_B_H2 + CHAR_W2 - 1)
    && (v_cnt >= CHAR_B_V2)      && (v_cnt < CHAR_B_V2 + CHAR_H2)
    && (char2[char_y2][char_x2] == 1'b1)) begin // 移动的方位是
    O_red   <= 4'b1111; // 白色
    O_green <= 4'b1111;
    O_blue  <= 4'b1111;
end
else begin // 黑色
    O_red   <= 4'b0000;
    O_green <= 4'b0000;
    O_blue  <= 4'b0000;
end
end

else if ((h_cnt >= H_SYNC_PULSE + H_BACK_PORCH + 600)
    && (h_cnt <= H_SYNC_PULSE + H_BACK_PORCH + 630)
    && (v_cnt >= V_SYNC_PULSE + V_BACK_PORCH + 202)
    && (v_cnt <= V_SYNC_PULSE + V_BACK_PORCH + 237)) begin
if ((h_cnt >= CHAR_B_H4 - 1) && (h_cnt < CHAR_B_H4 + CHAR_W4 - 1)
    && (v_cnt >= CHAR_B_V4)      && (v_cnt < CHAR_B_V4 + CHAR_H4)
    && (dir_index)
    && (char4[char_y4][char_x4 + 32 * (5 - dir_index)] == 1'b1))

begin // 具体方位为
    O_red   <= 4'b1111; // 白色
    O_green <= 4'b1111;
    O_blue  <= 4'b1111;
end
else begin // 灰色
    O_red   <= 4'b0011;

```

```

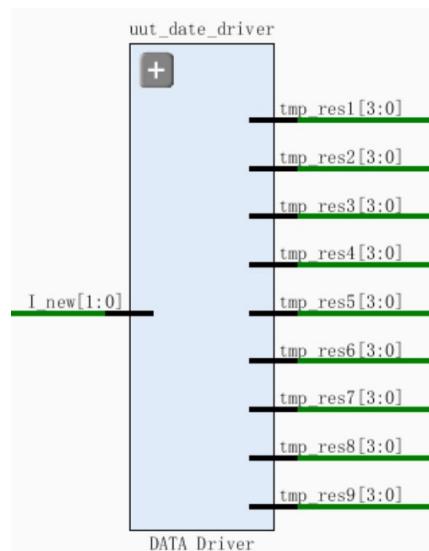
        O_green <= 4'b0011;
        O_blue  <= 4'b0011;
    end
end
else if (h_cnt >= (H_SYNC_PULSE + H_BACK_PORCH + 515)
&&     h_cnt <= (H_SYNC_PULSE + H_BACK_PORCH + 515 + C_IMAGE_WIDTH
- 1'b1)
&&     v_cnt >= (V_SYNC_PULSE + V_BACK_PORCH + 300)
&&     v_cnt <= (V_SYNC_PULSE + V_BACK_PORCH + 300 + C_IMAGE_HEIGHT
- 1'b1)) begin
    if (O_gameover) begin // 闯关成功图片
        O_red      <= W_rom_data[11:8]; // Rom 中的数据
        O_green    <= W_rom_data[7:4];
        O_blue     <= W_rom_data[3:0];
    end
    else begin
        O_red      <= 4'b0000;
        O_green    <= 4'b0000;
        O_blue     <= 4'b0000;
        //R_rom_addr <= 14'd0;
    end
    if (R_rom_addr == C_IMAGE_PIX_NUM - 1'b1)
        R_rom_addr <= 14'd0; // Rom 地址
    else
        R_rom_addr <= R_rom_addr + 1'b1;
    end
    else begin
        O_red      <= 4'b0000;
        O_green    <= 4'b0000;
        O_blue     <= 4'b0000;
        R_rom_addr <= R_rom_addr; // 不可以给 0
    end
end
else begin
    O_red      <= 4'b0000;
    O_green    <= 4'b0000;
    O_blue     <= 4'b0000;
    R_rom_addr <= R_rom_addr; // 不可以给 0
end
end
endmodule

```

### 6) DATA\_Driver 子模块:

1) 描述: 根据 I\_new 引脚, 用户所选择的关卡, 进行游戏数据的生成, 将结果通过 tmp\_res 返回给 VGA 子系统。

### 2) 接口信号及其定义:



接口名称	接口属性	接口描述
I_new	input	游戏关卡的选择
tmp_res1	output	数字格 1 内的数字
tmp_res2	output	数字格 2 内的数字
tmp_res3	output	数字格 3 内的数字
tmp_res4	output	数字格 4 内的数字
tmp_res5	output	数字格 5 内的数字
tmp_res6	output	数字格 6 内的数字
tmp_res7	output	数字格 7 内的数字
tmp_res8	output	数字格 8 内的数字
tmp_res9	output	数字格 9 内的数字

### 3) 调试说明:

在 Verilog 语言中, 任何模块的输入输出端口的变量都不可以是二维数组的类型, 需要将一维数组一一列举。同时由于这里的 tmp\_res 是用来每次的初始化或游戏中的重置游戏数据, 所以在显示器的子系统中, tmp\_res 传输给的变量需要定义为 wire 类型, 即相当于物理连线。同时在显示器子系统中用来真正表示数字格数据的变量 res 需要定义为 reg 类型, 因为其需要在判断能否移动的 always 模块中进行数据变化操作。

#### 4) Verilog 代码:

```
`timescale 1ns / 1ps

module DATA_Driver(
    input      [1: 0]  I_new,
    output reg [3: 0] tmp_res1,
    output reg [3: 0] tmp_res2,
    output reg [3: 0] tmp_res3,
    output reg [3: 0] tmp_res4,
    output reg [3: 0] tmp_res5,
    output reg [3: 0] tmp_res6,
    output reg [3: 0] tmp_res7,
    output reg [3: 0] tmp_res8,
    output reg [3: 0] tmp_res9
);

always @ (I_new[0] or I_new[1]) begin // 也可以加 pin
    if (I_new[0] == 0 && I_new[1] == 0) begin // case 1
        tmp_res1  <= 4'd1;
        tmp_res2  <= 4'd2;
        tmp_res3  <= 4'd3;
        tmp_res4  <= 4'd4;
        tmp_res5  <= 4'd0;
        tmp_res6  <= 4'd6;
        tmp_res7  <= 4'd7;
        tmp_res8  <= 4'd5;
        tmp_res9  <= 4'd8;
    end
    else if (I_new[0] == 1 && I_new[1] == 0) begin // case 2
        tmp_res1  <= 4'd0;
        tmp_res2  <= 4'd3;
        tmp_res3  <= 4'd6;
        tmp_res4  <= 4'd2;
        tmp_res5  <= 4'd5;
        tmp_res6  <= 4'd8;
        tmp_res7  <= 4'd1;
        tmp_res8  <= 4'd4;
        tmp_res9  <= 4'd7;
    end
    else if (I_new[0] == 0 && I_new[1] == 1) begin // case 3
        tmp_res1  <= 4'd7;
        tmp_res2  <= 4'd3;
        tmp_res3  <= 4'd6;
        tmp_res4  <= 4'd2;
    end
end
```

```

    tmp_res5  <= 4'd4;
    tmp_res6  <= 4'd1;
    tmp_res7  <= 4'd5;
    tmp_res8  <= 4'd8;
    tmp_res9  <= 4'd0;
end
else begin // 测试
    tmp_res1  <= 4'd1;
    tmp_res2  <= 4'd2;
    tmp_res3  <= 4'd3;
    tmp_res4  <= 4'd4;
    tmp_res5  <= 4'd5;
    tmp_res6  <= 4'd6;
    tmp_res7  <= 4'd7;
    tmp_res8  <= 4'd0;
    tmp_res9  <= 4'd8;
end
end

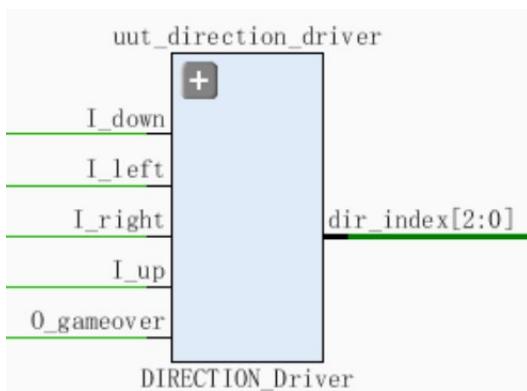
endmodule

```

### 7) DIRECTION\_Driver 子模块:

1) 描述: 在 O\_gameover 不是 1 的情况下, 根据用户在 NEXYS4 开发板上选择的方向按键, 将其转化为方向键对应的序号, 返回给显示器子系统。

### 2) 接口信号及其定义:



接口名称	接口属性	接口描述
I_up	input	NEXYS4 方向按键: 上
I_down	input	NEXYS4 方向按键: 下
I_left	input	NEXYS4 方向按键: 左
I_right	input	NEXYS4 方向按键: 右

<b>O_gameover</b>	input	是否游戏结束，高电平为游戏结束
<b>dir_index</b>	output	方向键对应的序号

### 3) 调试说明:

由于用户选择数字输出，采用的是持续输出，因此这里对方向输出考虑采用间断输出，即仅在用户按方向键时，才会输出用户选择的按键。

### 4) Verilog 代码:

```

`timescale 1ns / 1ps

module DIRECTION_Driver(
    input          I_up,
    input          I_down,
    input          I_left,
    input          I_right,
    input          O_gameover,
    output reg [2:0] dir_index
);

    always @ (I_up, I_down, I_left, I_right, O_gameover) begin //added
        if (!O_gameover) begin
            if (I_up) begin // 1
                dir_index <= 1;
            end
            else if (I_down) begin // 2
                dir_index <= 2;
            end
            else if (I_left) begin // 3
                dir_index <= 3;
            end
            else if (I_right) begin // 4
                dir_index <= 4;
            end
            else
                dir_index <= 0;
        end
    end

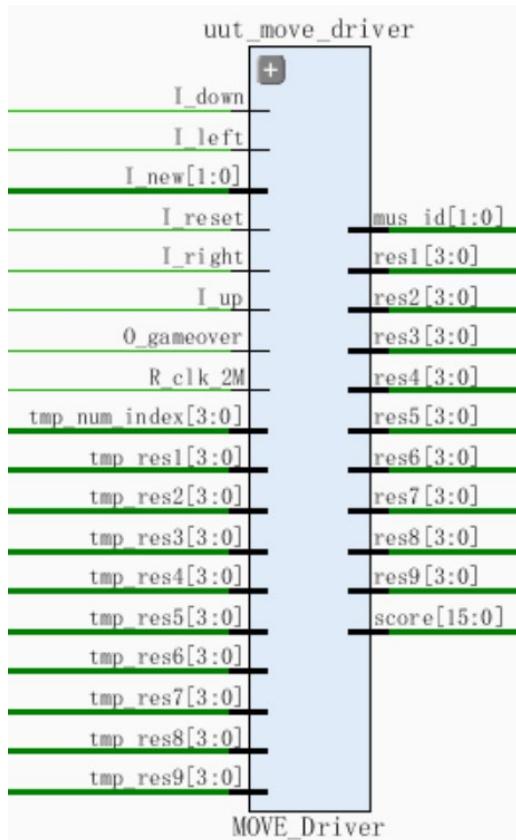
endmodule

```

## 8) MOVE\_Driver 子模块:

1) 描述: 若用户按下重置键 (I\_reset), 则对游戏数据根据用户所选择的关卡 (I\_new) 进行重新赋值。在其余情形下, 根据用户选择的移动方案, 判断该移动是否可行, 若可行, 则交换两个 res 的值, 并令分数 (步骤) +1。将改变后的 res 的值, mus\_id, score 全部传递回给显示器系统。

## 2) 接口信号及其定义:



接口名称	接口属性	接口描述
R_clk_2M	input	频率为 2mhz 的时钟
I_reset	input	是否对游戏进行重置
I_up	input	NEXYS4 方向按键: 上
I_down	input	NEXYS4 方向按键: 下
I_left	input	NEXYS4 方向按键: 左
I_right	input	NEXYS4 方向按键: 右
tmp_num_index	input	用户选择的数字格
tmp_res1...9 (以 1 为例)	input	当前数字格中存储的数字
I_new	input	用户想要选择的关卡
O_gameover	input	是否游戏结束, 高电平为游戏结束

<b>res1...9 (以 1 为例)</b>	output	移动后数字格将要存储的数字
<b>mus_id</b>	output	当前关卡对应的乐曲序号
<b>score</b>	output	当前游戏已执行的步数（分数）

### 3) 调试说明:

首先需要注意的是这里的时钟频率，一定要小于 VGA 显示屏输出图像的时钟频率，否则将有可能导致两个数字格出现同一个数字。其次，这里的 tmp\_res 无需使用 reg 类型，因为仅仅需要读取该数组内容即可。而 res 需要使用 reg 类型，因为在后续判断可否移动的过程中，可能会对 res 进行重新的赋值。而对于可交换的情形，例如我需要对 res1 和 res2 交换，可以由两种实现方式。其一为 {res1, res2} <= {res2, res1}，其二为 begin res1 <= res2; res2 <= res1; end。这里就可以看出时序电路的好处，在 begin-end 块中的所有数据都是同时进行的，而无先后关系，因此不需要定义一个临时变量来辅助实现两个数据的交换。

### 4) Verilog 代码:

```

`timescale 1ns / 1ps

module MOVE_Driver(
    input          R_clk_2M,
    input          I_reset,
    input          I_up,
    input          I_down,
    input          I_left,
    input          I_right,
    input [3: 0]   tmp_num_index,
    input [3: 0]   tmp_res1,
    input [3: 0]   tmp_res2,
    input [3: 0]   tmp_res3,
    input [3: 0]   tmp_res4,
    input [3: 0]   tmp_res5,
    input [3: 0]   tmp_res6,
    input [3: 0]   tmp_res7,
    input [3: 0]   tmp_res8,
    input [3: 0]   tmp_res9,
    input [1: 0]   I_new,
    input          O_gameover,
    output reg [3: 0] res1,
);

```

```

        output reg [3: 0] res2,
        output reg [3: 0] res3,
        output reg [3: 0] res4,
        output reg [3: 0] res5,
        output reg [3: 0] res6,
        output reg [3: 0] res7,
        output reg [3: 0] res8,
        output reg [3: 0] res9,
        output reg [1: 0] mus_id,
        output reg [15: 0] score
    );

always @(posedge R_clk_2M) begin // ①
    if (I_reset) begin
        res1 <= tmp_res1;
        res2 <= tmp_res2;
        res3 <= tmp_res3;
        res4 <= tmp_res4;
        res5 <= tmp_res5;
        res6 <= tmp_res6;
        res7 <= tmp_res7;
        res8 <= tmp_res8;
        res9 <= tmp_res9;
        score <= 0;
    case ({I_new[0], I_new[1]})
        2'b00: mus_id <= 0;
        2'b10: mus_id <= 1;
        2'b01: mus_id <= 2;
        default: ;
    endcase
    end
    else begin // 判断能否移动
        if (!O_gameover) begin
            if (tmp_num_index == 1) begin // 1-②
                if (I_right) begin
                    if (res2 == 0) begin {res1,res2} <= {res2,res1}; score <=
score + 1; end
                end
                else if (I_down) begin
                    if (res4 == 0) begin {res1,res4} <= {res4,res1}; score <=
score + 1; end
                end
            end // ②
        end
    end

```

```

else if (tmp_num_index == 2) begin // 2-②
    if (I_right) begin
        if (res3 == 0) begin {res2,res3} <= {res3,res2}; score <=
score + 1; end
    end
    else if (I_left) begin
        if (res1 == 0) begin {res2,res1} <= {res1,res2}; score <=
score + 1; end
    end
    else if (I_down) begin
        if (res5 == 0) begin {res2,res5} <= {res5,res2}; score <=
score + 1; end
    end
end

else if (tmp_num_index == 3) begin // 3-②
    if (I_left) begin
        if (res2 == 0) begin {res3,res2} <= {res2,res3}; score <=
score + 1; end
    end
    else if (I_down) begin
        if (res6 == 0) begin {res3,res6} <= {res6,res3}; score <=
score + 1; end
    end
end

else if (tmp_num_index == 4) begin // 4-②
    if (I_up) begin
        if (res1 == 0) begin {res4,res1} <= {res1,res4}; score <=
score + 1; end
    end
    else if (I_right) begin
        if (res5 == 0) begin {res4,res5} <= {res5,res4}; score <=
score + 1; end
    end
    else if (I_down) begin
        if (res7 == 0) begin {res4,res7} <= {res7,res4}; score <=
score + 1; end
    end
end

else if (tmp_num_index == 5) begin // 5-②
    if (I_left) begin

```

```

        if (res4 == 0) begin {res5,res4} <= {res4,res5}; score <=
score + 1; end
        end
        else if (I_right) begin
            if (res6 == 0) begin {res5,res6} <= {res6,res5}; score <=
score + 1; end
            end
        else if (I_up) begin
            if (res2 == 0) begin {res5,res2} <= {res2,res5}; score <=
score + 1; end
            end
        else if (I_down) begin
            if (res8 == 0) begin {res5,res8} <= {res8,res5}; score <=
score + 1; end
            end
        end

else if (tmp_num_index == 6) begin // 6-②
    if (I_left) begin
        if (res5 == 0) begin {res6,res5} <= {res5,res6}; score <=
score + 1; end
        end
    else if (I_up) begin
        if (res3 == 0) begin {res6,res3} <= {res3,res6}; score <=
score + 1; end
        end
    else if (I_down) begin
        if (res9 == 0) begin {res6,res9} <= {res9,res6}; score <=
score + 1; end
        end
    end
end

else if (tmp_num_index == 7) begin // 7-②
    if (I_right) begin
        if (res8 == 0) begin {res7,res8} <= {res8,res7}; score <=
score + 1; end
        end
    else if (I_up) begin
        if (res4 == 0) begin {res7,res4} <= {res4,res7}; score <=
score + 1; end
        end
    end
end

else if (tmp_num_index == 8) begin // 8-②

```

```

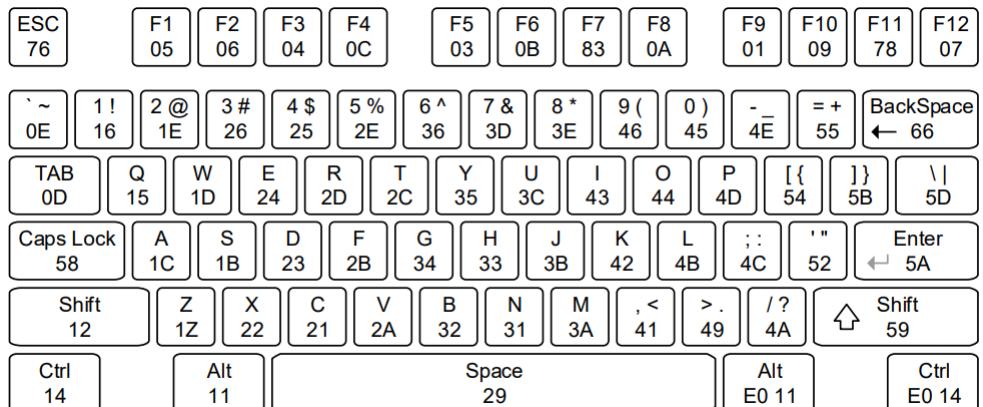
        if (I_right) begin
            if (res9 == 0) begin {res8,res9} <= {res9,res8}; score <=
score + 1; end
            end
        else if (I_left) begin
            if (res7 == 0) begin {res8,res7} <= {res7,res8}; score <=
score + 1; end
            end
        else if (I_up) begin
            if (res5 == 0) begin {res8,res5} <= {res5,res8}; score <=
score + 1; end
            end
        end // ②

else if (tmp_num_index == 9) begin // 5-②
    if (I_left) begin
        if (res8 == 0) begin res8 <= res9; res9 <= res8; score <=
score + 1; end
        end
    else if (I_up) begin
        if (res6 == 0) begin {res9,res6} <= {res6,res9}; score <=
score + 1; end
        end
    end
end
end
end
end // ①
endmodule

```

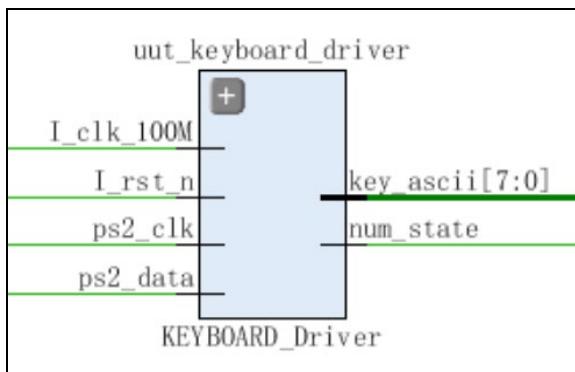
### 3. 键盘子系统 KEYBOARD\_Driver

1) 描述: ps/2 设备的 ps2\_clk 和 ps2\_data 在正常情形为高电平, 当 ps/2 设备等待发送数据时, 先对键盘时钟数据信号进行延时锁存 (看 ps2\_clk 是否为高电平), 再根据键盘的时钟信号的下降沿读取数据, 包括起始位、校验位和结束位, 每 11bit 一读, 将数据位 (8 个 bit) 进行暂存。接着, 由于 ps/2 设备在按下一个键后, 键盘会输出相应的扫描码 (Make Code), 松开按键后, 又会输出 F0 (Break Code) 和扫描码, 因此需要检测到的是松开后读取到的最后一个键码, 得到键码后将其转化为对应的 ascii 码后, 返回给顶端系统。



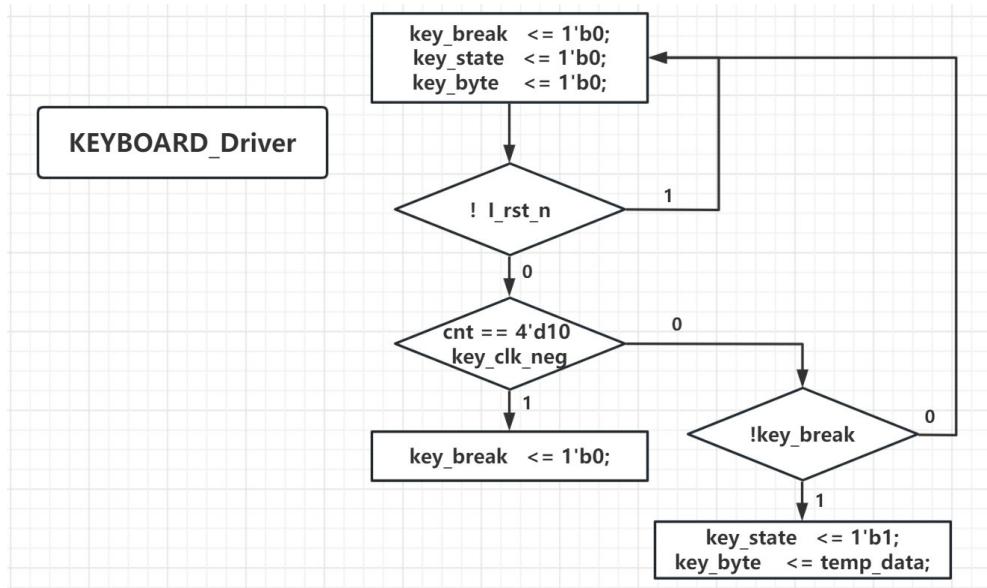
### 按键对应扫描码示意图

### 2) 接口信号及其定义:



接口名称	接口属性	接口描述
I_clk_100M	input	系统时钟
I_rst_n	input	系统复位，低有效
ps2_clk	input	PS2 键盘时钟输入
ps2_data	input	PS2 键盘数据输入
key_ascii	output	按键键值对应的 ASCII 编码
num_state	output	当前是否有按键

### 3) 结构框图:



### 4) 调试说明:

根据描述中所说，尤其需要注意 PS/2 键盘有断码的特性，针对持续按压的情况，只需要返回一个值即可。同时，**在这里的 if-else 判断使用到非阻塞赋值，时序逻辑的特性**，在第一次收到断码时，进行 key\_break 是 1 的赋值，不要进行读入键值的赋值，因为此时的键值是 F0，直接退出当前一次的 always 模块即可。在下一个上升沿来临时，直接进入 key\_break 等于 1 的分支，读入键值进行赋值。接着在将键盘返回的有效键值转换为按键字母对应的 ASCII 码的 always 模块中，对按键为数字的情形进行转化，否则赋为 0，便于后续通过 TOP\_GAME 模块传入 VGA 模块中使得数字得以持续输出。

### 5) Verilog 代码

```

`timescale 1ns / 1ps

module KEYBOARD_Driver(
    input          I_clk_100M,      // 系统时钟
    input          I_rst_n,        // 系统复位，低有效
    input          ps2_clk,        // PS2 键盘时钟输入
    input          ps2_data,        // PS2 键盘数据输入
    output reg [7:0] key_ascii,    // 按键键值对应的 ASCII 编码
    output reg      num_state
);

reg      ps2_clk_r0  = 1'b1, ps2_clk_r1  = 1'b1;
reg      ps2_data_r0 = 1'b1, ps2_data_r1 = 1'b1;

```

```

initial num_state <= 0;

// 对键盘时钟数据信号进行延时锁存
always @ (posedge I_clk_100M or negedge I_rst_n) begin
    if (!I_rst_n) begin
        ps2_clk_r0 <= 1'b1;
        ps2_clk_r1 <= 1'b1;
        ps2_data_r0 <= 1'b1;
        ps2_data_r1 <= 1'b1;
    end
    else begin
        ps2_clk_r0 <= ps2_clk;           // PS2 键盘时钟输入
        ps2_clk_r1 <= ps2_clk_r0;
        ps2_data_r0 <= ps2_data;
        ps2_data_r1 <= ps2_data_r0;      // PS2 键盘时钟输入
    end
end

// 键盘时钟信号下降沿检测
wire key_clk_neg = ps2_clk_r1 & (~ps2_clk_r0);

reg [3:0] cnt;
reg [7:0] temp_data;

//根据键盘的时钟信号的下降沿读取数据
always @ (posedge I_clk_100M or negedge I_rst_n) begin
    if (!I_rst_n) begin
        cnt <= 4'd0;
        temp_data <= 8'd0;
    end
    else if (key_clk_neg) begin
        if (cnt >= 4'd10) cnt <= 4'd0;
        else cnt <= cnt + 1'b1;
        case (cnt)
            4'd0: ; // 起始位
            4'd1: temp_data[0] <= ps2_data_r1;
            4'd2: temp_data[1] <= ps2_data_r1;
            4'd3: temp_data[2] <= ps2_data_r1;
            4'd4: temp_data[3] <= ps2_data_r1;
            4'd5: temp_data[4] <= ps2_data_r1;
            4'd6: temp_data[5] <= ps2_data_r1;
            4'd7: temp_data[6] <= ps2_data_r1;
            4'd8: temp_data[7] <= ps2_data_r1;
        end
    end
end

```

```

        4'd9; // 校验位
        4'd10;; // 结束位
        default: ;
    endcase
end
end

reg      key_break = 1'b0;
reg      key_state = 1'b0;
reg [7:0] key_byte = 1'b0;

// 根据通码和断码判定按键的当前是按下还是松开，松开是 0
always @ (posedge I_clk_100M or negedge I_rst_n) begin
    if (!I_rst_n) begin
        key_break <= 1'b0;
        key_state <= 1'b0;
        key_byte <= 1'b0;
    end
    else if (cnt == 4'd10 && key_clk_neg) begin
        if (temp_data == 8'hf0)      // 收到断码 (8'hf0)：按键松开，下一个数据为
断码，设置断码标示为 1
            key_break <= 1'b1;    // 不能赋值，现在是断码
        else if (!key_break) begin // 断码标示 0：当前数据为按下数据，输出键值，
并设置按下标示为 1
            key_state <= 1'b1;
            key_byte <= temp_data;
        end
        else begin // 断码标示 1：当前数据为松开数据，断码标示和按下标示清零
            key_state <= 1'b0;
            key_break <= 1'b0;
        end
    end
end
end

reg num_state2 = 1'b0;

// 将键盘返回的有效键值转换为按键字母对应的 ASCII 码
always @ (key_state) begin // 一有新的就会读入
    if (num_state2 == 1'b0) begin
        case (key_byte) //translate key_byte to key_ascii
            8'h16: begin key_ascii <= 8'h31; end //1
            8'h1E: begin key_ascii <= 8'h32; end //2
            8'h26: begin key_ascii <= 8'h33; end //3
            8'h25: begin key_ascii <= 8'h34; end //4

```

```

    8'h2E: begin key_ascii <= 8'h35; end //5
    8'h36: begin key_ascii <= 8'h36; end //6
    8'h3D: begin key_ascii <= 8'h37; end //7
    8'h3E: begin key_ascii <= 8'h38; end //8
    default: key_ascii <= 0; // 保持原状
endcase
end
end

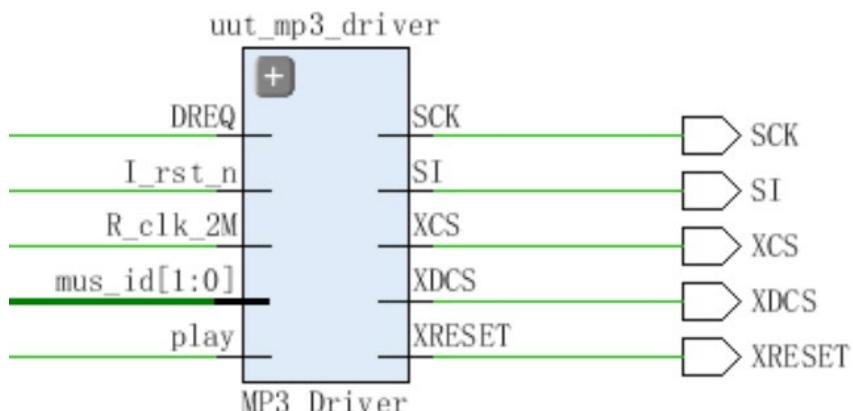
endmodule

```

#### 4. MP3 子系统 MP3\_Driver

1) 描述：通过参数定义的硬复位、软复位、等待、播放等不同状态，通过输入参数 play 和 DREQ，以及寄存器 sci\_w\_cnt 存储的 SCI 指令地址位数计数来判断当前处于的状态，由于是时序电路，每次只会执行一种状态，因此在现态，需要针对次态的条件进行复制。同时，由于每个寄存器被赋值后，都需要等待时间，因此需要做延时处理 (delay)，只有 DREQ 在寄存器被赋值拉低后，重新拉高，才可以进入下一个 SCI。若重置 play 后得以使乐曲重新播放，若改变 mus\_id 后同样得以令乐曲转换至该 id 对应曲目后播放。

2) 接口信号及其定义：



接口名称	接口属性	接口描述
R_clk_2M	input	12.288/6MHZ 时钟
I_rst_n	input	系统复位，低有效
play	input	开始播放请求
mus_id	input	PS2 键盘数据输入

<b>DREQ</b>	input	MP3 数据请求线
<b>XCS</b>	output	SCI 传输读写指令, 低电平有效
<b>XDCS</b>	output	SDI 传输数据, 字节同步
<b>SCK</b>	output	SPI 总线时钟, 12.288MHZ
<b>SI</b>	output	声音传感器有效信号灯 (传入 mp3)
<b>XRESET</b>	output	复位引脚 (硬件复位), 低电平有效

### 3) 调试说明:

根据 MP3 文档说明看懂 MP3 运行原理后, 参考 MP3 运行代码, 对 MP3 模块中用到的状态机进行分析, 并在读入曲调的代码块中, 加入四选一选择器, 以便不同关卡对应的不同歌曲的播放。同时在绘制 ASM 流程图, 列写状态转移真值表和绘制与调试控制器逻辑方案图的过程中, 可以很好的认识到 MP3 外设下几个状态处理的变化。而在模块实现中, 个人认为最复杂的是歌曲到 COE 的转化, 由于开发板 ROM 内存的有限性, 无法直接存放内存较大的 MP3 文件, 因此我考虑的是 MID 文件, 但是 MID 文件中也只有单声道的可以播放 (可以使用 Audacity 软件来查看), 是单声道 MID 文件后, 先将其转化为 mif 文件, 再编写 matlab 脚本, 将 mif 文件转化为 coe 文件。

### 4) Verilog 代码:

```

`timescale 1ns / 1ps

module MP3_Driver(
    input          R_clk_2M,      // 12.288/6MHZ 时钟
    input          I_RST_N,
    input          play,         // 开始播放请求
    input [1: 0]   MUS_ID,
    input          DREQ,         // MP3 数据请求线, 显示 VS1003 是否可以接受数据,
                                // 高电平可以传输数据
    output reg     XCS,          // 片选输入, 低电平有效 (SCI 传输读写指令)
    output reg     XDCS,         // 数据片选, 字节同步 (SDI 传输数据)
    output         SCK,          // SPI 总线时钟, 12.288MHZ
    output reg     SI,           // 声音传感器有效信号灯 (传入 mp3)
    output reg     XRESET        // 复位引脚 (硬件复位), 低电平有效
);

parameter H_RESET = 4'd0,          // 硬复位
          S_RESET = 4'd1,          // 软复位
          SET_CLOCKF = 4'd2,        // 设置时钟寄存器
          SET_BASS = 4'd3,          // 设置音调寄存器

```

```

        SET_VOL      = 4'd4,          // 设置音量
        WAIT         = 4'd5,          // 等待
        PLAY         = 4'd6;          // 播放

reg [3:0]      state       = WAIT;          // 状态
reg [31:0]     delay        = 32'd0;          // 延时
reg [31:0]     sci_w       = 32'd0;          // 指令与地址 写
reg [7:0]      sci_w_cnt   = 8'd32;          // SCI 指令地址位数计数
reg [31:0]     music_data  = 32'd0;          // 音乐数据
reg [31:0]     sdi_cnt     = 32'd32;          // SDI 当前 4 字节已传送 BIT 数

reg [11:0]     addra       = 14'd0;          // ROM 中的地址
wire[31:0]    douta;                  // ROM 传出
wire[31:0]    doutb;
wire[31:0]    doutc;
reg [1:0]      pre_id      = 0;

reg           ena        = 0;

assign SCK = (R_clk_2M & ena);

always @ (negedge R_clk_2M) begin
    if (!I_rst_n || pre_id != mus_id || !play) begin    // 初始化
        pre_id      <= mus_id;
        XDCS        <= 1'b1;
        ena         <= 0;
        SI          <= 1'b0;
        XCS         <= 1'b1;
        XRESET      <= 1'b1;
        state       <= WAIT;
        addra       <= 14'd0;
        sdi_cnt     <= 32'd32;
        music_data  <= 32'd0;
    end
    else begin
        case (state)
            /*-----等待-----*/
            WAIT: begin
                if (play) begin
                    if (delay > 0)
                        delay  <= delay - 1'b1;
                    else begin                      // 转到硬复位
                        delay  <= 32'd1000;
                    end
                end
            end
        endcase
    end
end

```

```

        state    <= H_RESET;
    end
end
else
    delay    <= 32'd16700;
end
/*-----硬复位-----*/
H_RESET: begin
    if (delay > 0)
        delay    <= delay - 1'b1;
    else begin
        XCS      <= 1'b1;           //传输读、写指令
        XRESET   <= 1'b0;           //硬件复位，低电平有效
        delay    <= 32'd16700;       //复位后延时一段时间
        state    <= S_RESET;         //转移到软复位
        sci_w    <= 32'h02_00_0804; //软复位指令；使 MODE
的值为 0804；即第 2 位和第 11 位置 1；软件复位和本地模式
        sci_w_cnt <= 8'd32;          //指令、地址、数据总长
度
    end
end
/*-----软复位-----*/
S_RESET: begin
    if (delay > 0) begin
        XRESET    <= (delay < 32'd16650);
        delay     <= delay - 1'b1;
    end
    else if (sci_w_cnt == 0) begin           //软复位结束
        delay     <= 32'd16600;

        state    <= SET_VOL;           //转移到设置 VOL
        sci_w    <= 32'h02_0b_0000;
        sci_w_cnt <= 8'd32;

        XCS      <= 1'b1;           //拉高 XCS
        ena      <= 1'b0;           //关闭输入时钟
        SI       <= 1'b0;
    end
    else if (DREQ) begin                   //当 DREQ 有效时开始软
复位；高电平传输数据
        XCS      <= 1'b0;
        ena      <= 1'b1;
        SI       <= sci_w[sci_w_cnt - 1];
        sci_w_cnt <= sci_w_cnt - 1'b1;

```

```

        end
    else begin                                //DREQ 无效时继续等待;
        XCS      <= 1'b1;                      //片选 SCI 传输读、写
    end

指令
    ena      <= 1'b0;
    SI       <= 1'b0;
end

/*
-----播放音乐-----
PLAY: begin
    if (delay > 0)
        delay      <= delay - 1'b1;
    else if (play) begin
        XDCS      <= 1'b0;
        ena       <= 1'b1;
        if (sdi_cnt == 0) begin                //传输完 4 字节 (32 位)
            XDCS      <= 1'b1;                  //拉高 XDCS
            ena       <= 1'b0;
            SI        <= 1'b0;
            sdi_cnt   <= 32'd32;
            if (mus_id == 2'b00)
                music_data <= douta;
            else if (mus_id == 2'b01)
                music_data <= doutb;
            else if (mus_id == 2'b10)
                music_data <= doutc;
            else;
            addra <= addra + 1'b1;
        end
    else begin
        //当 DREQ 有效 或当前字节尚未发送完毕 则继续传输
        if (DREQ || (sdi_cnt != 32 && sdi_cnt != 24 && sdi_cnt != 16 && sdi_cnt != 8)) begin
            SI      <= music_data[sdi_cnt - 1];
            sdi_cnt <= sdi_cnt - 1'b1;
            ena     <= 1;
            XDCS    <= 1'b0;
        end
        else begin      //DREQ 拉低, 停止传输
            ena     <= 1'b0;
            XDCS    <= 1'b1;
            SI      <= 1'b0;
        end
    end

```

```

        end
    end
    else begin
        XCS      <= 1'b1;
        ena      <= 1'b0;
        SI       <= 1'b0;
    end
end

/*-----寄存器配置-----*/
default: begin
    if (delay > 0)
        delay <= delay - 1'b1;
    else if (sci_w_cnt == 0) begin           //结束一次 SCI 写入
        if (state == SET_CLOCKF) begin
            delay      <= 32'd11000;          //他是 mp3speed
            state      <= PLAY;
        end
        else if (state == SET_BASS) begin
            delay      <= 32'd2100;
            sci_w      <= 32'h02_03_7000;   // 7000 = 0111 0000
0000 0000
            state      <= SET_CLOCKF;      // 设置时钟寄存器
        end
    else begin
        delay      <= 32'd2100;
        sci_w      <= 32'h02_02_0000;
        state      <= SET_BASS;          // 设置音调寄存器
    end
    sci_w_cnt      <= 8'd32;
    XCS      <= 1'b1;
    ena      <= 1'b0;
    SI       <= 1'b0;
end
else if (DREQ) begin           // 写入 SCI 指令、地址、
数据
    XCS      <= 1'b0;
    ena      <= 1'b1;
    SI       <= sci_w[sci_w_cnt - 1];
    sci_w_cnt  <= sci_w_cnt - 1'b1;
end
else begin                      // DREQ 拉低，等待
    XCS      <= 1'b1;
    ena      <= 1'b0;

```

```

        SI           <= 1'b0;
    end
end
endcase
end
end

music_light m1 (
    .clka(R_clk_2M),      // input wire clka
    .addr(addr),   // input wire [11: 0] addr
    .douta(douta) // output wire [31 : 0] douta
);

music_pipa m2 (
    .clka(R_clk_2M),      // input wire clka
    .addr(addr),   // input wire [11 : 0] addr
    .douta(doutb) // output wire [31 : 0] douta
);

music_fly m3 (
    .clka(R_clk_2M),      // input wire clka
    .addr(addr),   // input wire [11 : 0] addr
    .douta(doutc) // output wire [31 : 0] douta
);

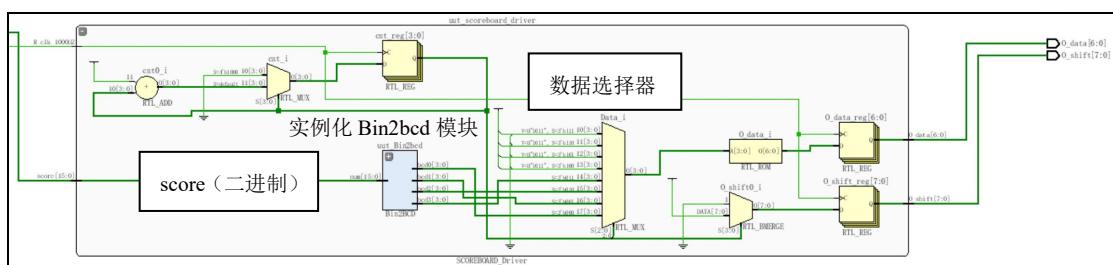
Endmodule

```

## 5. 数码管子系统 SCOREBOARD\_Driver

1) 描述：基于数字逻辑小实验 display7 模块的基础上，对传入的数据 score 进行二进制到 BCD 码的转化，使得其可以以 10 进制的形式在 NEXYS4 开发板的数码管上进行游戏步数的显示。

2) 接口信号及其定义：



接口名称	接口属性	接口描述
<b>R_clk_1000HZ</b>	input	频率为 1000hz 的时钟
<b>score</b>	input	需要输出的二进制数据
<b>O_shift</b>	output	数据输出对应第几个数码管
<b>O_data</b>	output	数码管输出的字码

3) 调试说明:

由于数字华容道步骤的有限性，没有必要进行过大的步数的输出，而 NEXYS4 开发板的七段数码管是同时开启的，因此考虑将左四个数码管的各段全部赋值为 1。同时，由于模块内定义的 Data 数组是 wire 类型，因此需要用 assign 赋值，而非在 always 模块内通过非阻塞赋值。

4) Verilog 代码:

```

`timescale 1ns / 1ps

module SCOREBOARD_Driver(
    input          R_clk_1000HZ,
    input      [15: 0] score,
    output reg [7: 0] O_shift,
    output reg [6: 0] O_data
);

    wire    [3:0]  Data[7:0];
    reg     [3:0]  cnt = 0; //计数器

    assign Data[4] = 4'd11;
    assign Data[5] = 4'd11;
    assign Data[6] = 4'd11;
    assign Data[7] = 4'd11;

    Bin2BCD uut_Bin2bcd(
        .num(score),
        .bcd0(Data[0]),
        .bcd1(Data[1]),
        .bcd2(Data[2]),
        .bcd3(Data[3])
    );

    //片选输出
    always @ (posedge R_clk_1000HZ) begin
        if (cnt == 4'd8)

```

```

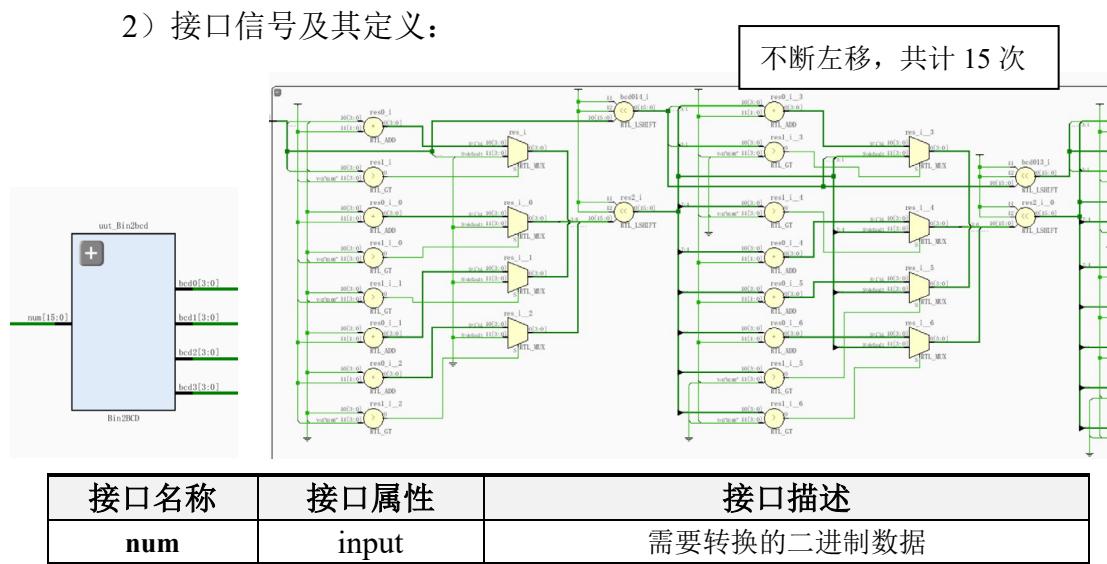
        cnt <= 0;
    else
        cnt <= cnt + 1;
    0_shift <= 8'b1111_1111;
    0_shift[cnt] <= 0;//选择一个数码管进行输出
    case (Data[cnt])
        4'b0000: 0_data <= 7'b100_0000; // 0
        4'b0001: 0_data <= 7'b111_1001; // 1
        4'b0010: 0_data <= 7'b010_0100; // 2
        4'b0011: 0_data <= 7'b011_0000; // 3
        4'b0100: 0_data <= 7'b001_1001; // 4
        4'b0101: 0_data <= 7'b001_0010; // 5
        4'b0110: 0_data <= 7'b000_0010; // 6
        4'b0111: 0_data <= 7'b111_1000; // 7
        4'b1000: 0_data <= 7'b000_0000; // 8
        4'b1001: 0_data <= 7'b001_0000; // 9
        default: 0_data <= 7'b111_1111;
    endcase
end
endmodule

```

## 5) Bin2BCD 子模块

1) 描述：将 num 二进制数转化为 BCD8421 码。先将 num 用临时寄存器存储，每次向其传递最高位，令寄存器左移，若寄存器每 4 位的值大于 4，就将这一段数据加 3，否则不变，如此重复 15 次即可。在操作过程中需要注意的是模块输入输出端口的变量不可以使用二维数组的形式，因此必须要一一列举。

2) 接口信号及其定义：



<b>bcd0</b>	output	最低位
<b>bcd1</b>	output	次低位
<b>bcd2</b>	output	次高位
<b>bcd3</b>	output	最高位

3) Verilog 代码:

```

`timescale 1ns / 1ps

module Bin2BCD(
    input  [15: 0]      num,
    output [3: 0]      bcd0,
    output [3: 0]      bcd1,
    output [3: 0]      bcd2,
    output [3: 0]      bcd3
);

    reg [15: 0]  bin;
    reg [15: 0]  res;
    reg [15: 0]  bcd;

    always @ (num) begin
        bin = num[15: 0];
        res = 16'd0;
        repeat (15) begin
            res[0] = bin[15];
            if (res[3:0] > 4)
                res[3:0] = res[3:0] + 4'd3;
            else
                res[3:0] = res[3:0];
            if (res[7:4] > 4)
                res[7:4] = res[7:4] + 4'd3;
            else
                res[7:4] = res[7:4];
            if (res[11:8] > 4)
                res[11:8] = res[11:8] + 4'd3;
            else
                res[11:8] = res[11:8];
            if (res[15:12] > 4)
                res[15:12] = res[15:12] + 4'd3;
            else
                res[15:12] = res[15:12];
            res = res << 1;
            bin = bin << 1;
        end
    end
endmodule

```

```

        end
        res[0] = bin[15];
        bcd = res;
    end

    assign bcd0 = bcd[3:0];
    assign bcd1 = bcd[7:4];
    assign bcd2 = bcd[11:8];
    assign bcd3 = bcd[15:12];

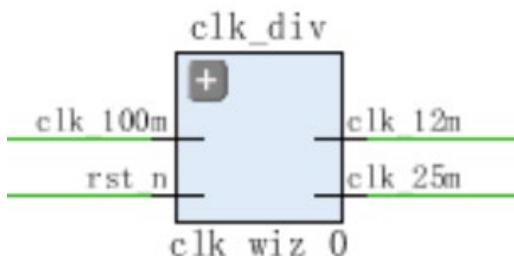
endmodule

```

## 6. clk\_wiz\_0 分频子模块

1) 描述: 对时钟进行分频, 100Mhz 分频得到 12.288Mhz 和 25Mhz。

2) 接口信号及其定义:

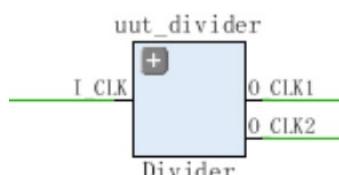


接口名称	接口属性	接口描述
<b>clk_100m</b>	input	系统时钟
<b>rst_n</b>	input	复位信号, 低有效
<b>clk_12m</b>	output	频率为 12mhz 的时钟
<b>clk_25m</b>	output	频率为 25mhz 的时钟

3) Divider 子模块

1) 描述: 对分频器进行实例化, 将 12.288Mhz 分频得到 2Mhz 和 1000hz。

2) 接口信号及其定义:



接口名称	接口属性	接口描述
<b>I_clk</b>	input	需要分频的时钟
<b>O_clk1</b>	output	分频得到的时钟 1
<b>O_clk2</b>	output	分频得到的时钟 2

3) Verilog 代码:

```
`timescale 1ns / 1ps

module Divider(
    input      I_CLK,
    output reg O_CLK1 = 0,
    output reg O_CLK2 = 0
);

parameter MOD1 = 32'd6;
parameter MOD2 = 32'd12288;

integer cnt1 = 32'd0;
integer cnt2 = 32'd0;

always @(posedge I_CLK) begin
    if (cnt1 < MOD1 / 2 - 1)
        cnt1 <= cnt1 + 1'b1;
    else begin
        cnt1 <= 32'd0;
        O_CLK1 <= ~O_CLK1;
    end

    if (cnt2 < MOD2 / 2 - 1)
        cnt2 <= cnt2 + 1'b1;
    else begin
        cnt2 <= 32'd0;
        O_CLK2 <= ~O_CLK2;
    end
end

endmodule
```

## 五、测试模块建模

### 1. VGA——TestBench

```
module vga_testbench;
    reg      I_clk_100M = 0;
    reg      I_rst_n;
    wire     [3: 0]  O_r;
    wire     [3: 0]  O_g;
    wire     [3: 0]  O_b;
    wire     O_hs;
    wire     O_vs;
    wire     R_clk_25M, R_clk_12M;

    initial begin
        I_rst_n = 1;
        # 2 I_rst_n = 0;
        # 2 I_rst_n = 1;
    end

    always begin
        #1 I_clk_100M = ~I_clk_100M ;
    end

    clk_VGA clkdiv (
        .clk (I_clk_100M),
        .clk_25m (R_clk_25M),
        .clk_12m (R_clk_12M),
        .rst_n (1));

    vga_driver uut_vga (
        .R_clk_25M(R_clk_25M),
        .I_rst_n(I_rst_n),
        .O_red (O_r),
        .O_green(O_g),
        .O_blue(O_b),
        .O_hs (O_hs),
        .O_vs (O_vs));

endmodule
```

### 2. Data——TestBench

```
`timescale 1ns / 1ps
module data_tb;
```

```

reg      [1: 0]  I_new;
wire     [3: 0]  tmp_res1;
wire     [3: 0]  tmp_res2;
wire     [3: 0]  tmp_res3;
wire     [3: 0]  tmp_res4;
wire     [3: 0]  tmp_res5;
wire     [3: 0]  tmp_res6;
wire     [3: 0]  tmp_res7;
wire     [3: 0]  tmp_res8;
wire     [3: 0]  tmp_res9;

initial begin
    I_new = 2'b00;
    # 10 I_new = 2'b01;
    # 10 I_new = 2'b10;
    # 10 I_new = 2'b11;
end

DATA_Driver data_driver_tb (
    .I_new (I_new),
    .tmp_res1 (tmp_res1),
    .tmp_res2 (tmp_res2),
    .tmp_res3 (tmp_res3),
    .tmp_res4 (tmp_res4),
    .tmp_res5 (tmp_res5),
    .tmp_res6 (tmp_res6),
    .tmp_res7 (tmp_res7),
    .tmp_res8 (tmp_res8),
    .tmp_res9 (tmp_res9)
);

endmodule

```

### 3. Direction——TestBench

```

module Direction_tb;
    reg      I_up;
    reg      I_down;
    reg      I_left;
    reg      I_right;
    reg      O_gameover;
    wire   [2:0] dir_index;

initial begin
    O_gameover = 0;

```

```

    I_up = 0; I_down = 0; I_left = 0; I_right = 0;
    I_up = 1;
    # 10 I_up = 0;      I_down = 1;
    # 10 I_down = 0;    I_right = 1;
    # 10 I_right = 0;   I_left = 1;
    # 10 O_gameover = 1;
end

DIRECTION_Driver dir_tb (
    .I_up (I_up),
    .I_down (I_down),
    .I_left (I_left),
    .I_right (I_right),
    .O_gameover (O_gameover),
    .dir_index (dir_index)
);

endmodule

```

#### 4. MP3——TestBench

```

`timescale 1ns / 1ps

module mp3_tb;

reg          R_clk_2M = 0;
reg          I_rst_n;
reg          play;
reg [1:0]    mus_id;
reg          DREQ;
wire         XCS;
wire         XDCS;
wire         SCK;
wire         SI;
wire         XRESET;

always #10 R_clk_2M = ~R_clk_2M;
always #10 DREQ = ~DREQ;

initial begin
    play = 0;
    I_rst_n = 0;
    DREQ   = 0;
    mus_id = 2'b01;
    #10 I_rst_n = 1;    play = 1;

```

```

#10 DREQ = 1;
#10 mus_id = 2'b10;
#10 mus_id = 2'b11;
#10 play = 0;
#10 play = 1;
end

MP3_Driver mp3_driver_tb (
.R_clk_2M (R_clk_2M),
.I_rst_n (I_rst_n),
.play (play),
.mus_id (mus_id),
.DREQ (DREQ),
.XCS (XCS),
.XDCS (XDCS),
.SCK (SCK),
.SI (SI),
.XRESET (XRESET)
);

endmodule

```

## 5. Scoreboard——TestBench

```

`timescale 1ns / 1ps

module scoreboard_tb;
reg R_clk_1000HZ = 0;
reg [15: 0] score;
wire [7: 0] O_shift;
wire [6: 0] O_data;

always #10 R_clk_1000HZ = ~R_clk_1000HZ;

initial begin
    score = 15'd0;
    #20 score = 16'd5;
    #20 score = 16'd60;
    #20 score = 16'd198;
    #20 score = 16'd2378;
end

SCOREBOARD_Driver scoreboard_driver_tb(
.R_clk_1000HZ (R_clk_1000HZ),
.score (score),

```

```

    .O_shift (O_shift),
    .O_data (O_data)
);

endmodule

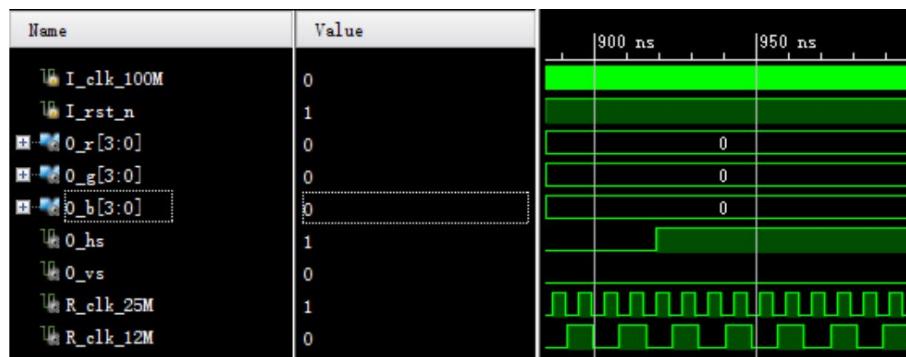
```

【说明】其余模块通过上板观察实验结果进行测试。

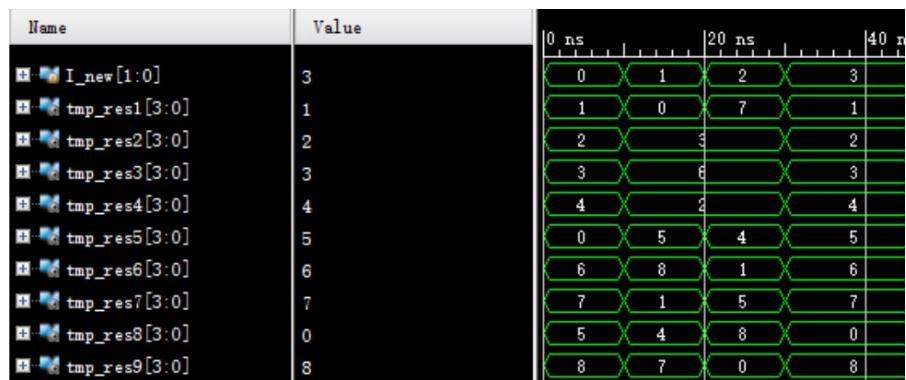
## 六、实验结果

### 1. TestBench 仿真图

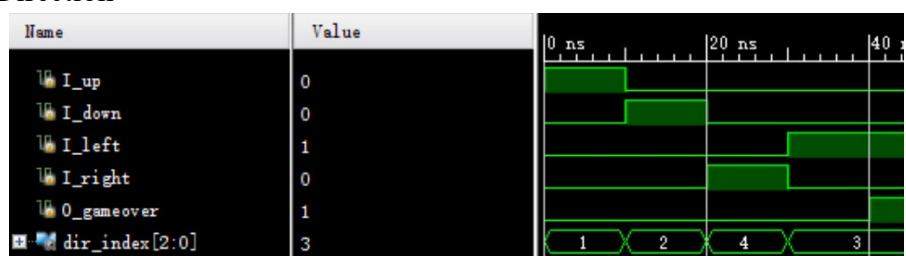
#### 1) VGA



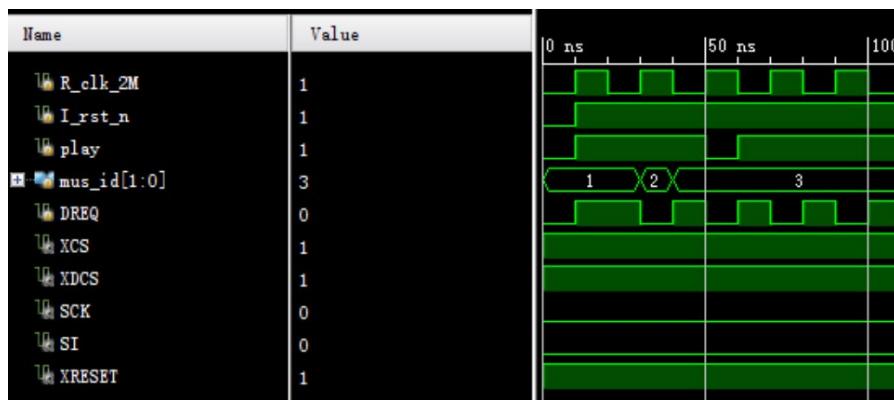
#### 2) Data



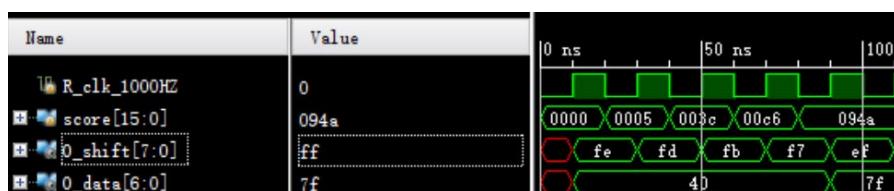
#### 3) Direction



#### 4) MP3

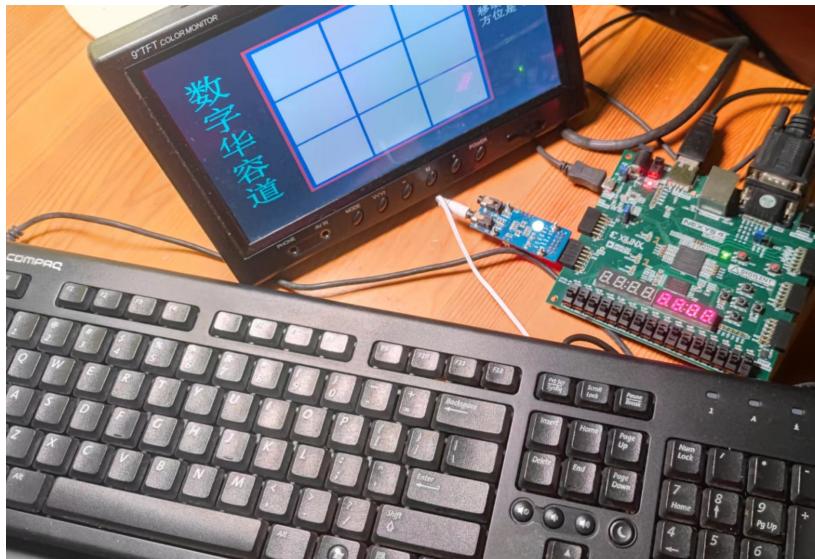


#### 5) Scoreboard

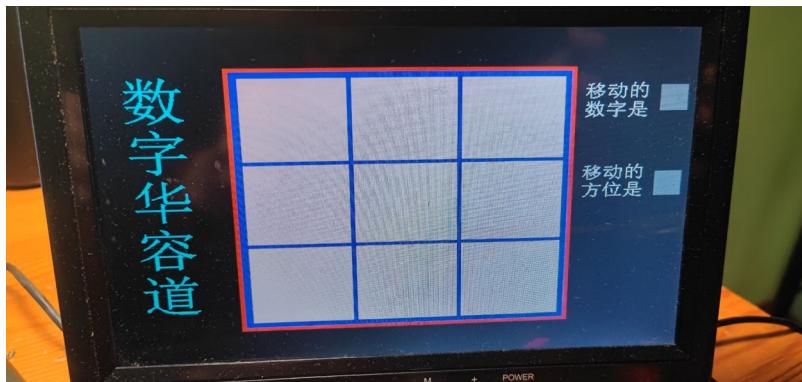


## 2. 上板观察实验结果

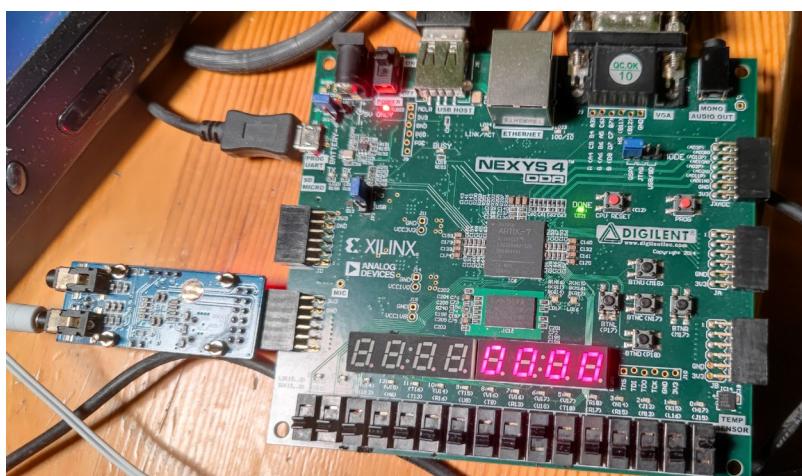
### 1) 连接设备，并烧入 bit 文件



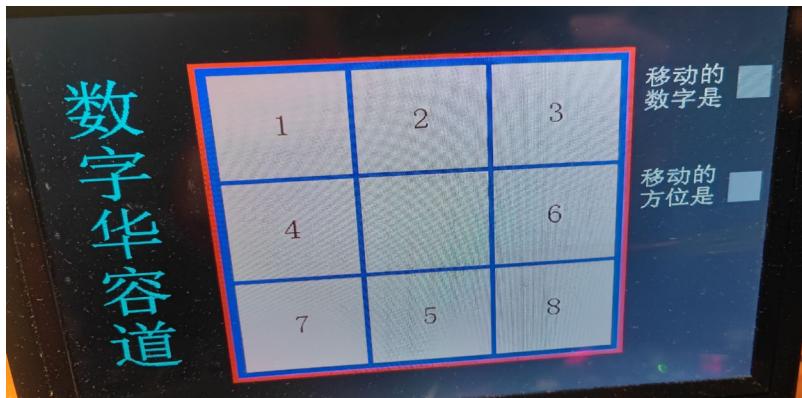
2) VGA 显示屏初始界面



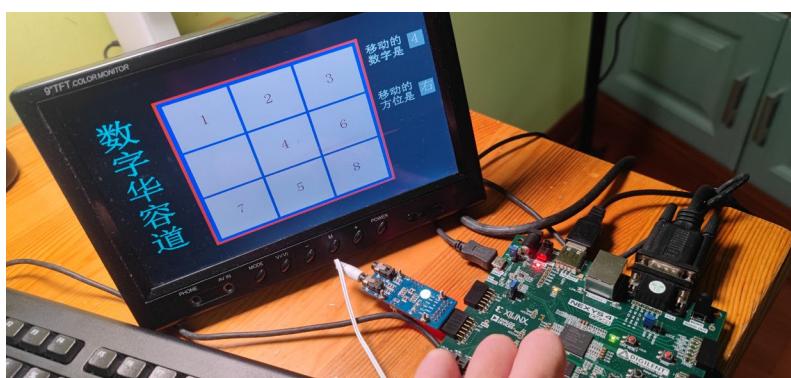
3) NEXYS4 DDR 开发板初始数据



4) 选择关卡后，按压 N17，开始游戏



5) 数字块移动的选择



6) 游戏闯关成功界面

