同济大学计算机系

数字逻辑课程综合实验报告

学　　号 2252201

姓　　名 胡世成

专　　业 计算机科学与技术

授课老师 张冬冬

一、实验内容

基于旋转编码器及 VGA 显示器的自制小游戏

命运石扭结

“致敬时间单行道上所有命运的相遇与意外的错过”

——基于《命运石之门》故事设定与《时空幻境》游戏理念启发的原创小游戏

1.游戏目标

让男主（凤凰院凶真）与逆行的女主（牧濑红莉栖）在同一条世界线的同一时刻相遇。

相遇亦可，错过亦可，都会成就一段不一样的故事。

2.游戏设定

① **时间之矢**——一条由过去指向未来的箭头，物理学家爱丁顿在根据热力学第二定律即熵增定律提出的概念，沿着这一个箭头的方向热力学熵会只增不减，理论上无法从未来回到过去；

② **事件奇点**——当时间线上发生一个有多个可能结果的事件时，会产生不同的未来。这些影响时间线走向的事件称为事件奇点；

③ **世界线**——基于不同的过去会产生不同的未来，位于过去的某个事件奇点的不同结果会产生不同的未来世界，这些彼此相异的世界因各种事件缠绕在一起成为世界线簇，而其中每一个不同的世界称为一条世界线。理论上可以通过改变过去某一时刻的选择而实现在不同世界线之间的跨越；

④ **世界偏移量**——由辉光管显示的世界线偏移度数，从 0.0000000%到 9.9999999%连续表示当前世界线与由当前世界线可跨越到的最远的世界线。由于一个世界线簇的世界线通过无数个事件奇点彼此纠缠，所以理论上可以根据两条世界线的相似程度来定义两条世界线之间的距离。这个距离由辉光管以数字的形式表现出来，称为世界偏移量。世界偏移量只能表示当前世界线与原世界线的相对关系，而不是绝对关系，即相同的世界偏移量会对应到不同的世界线，但原世界线（即 0.0000000%）只有一条；

⑤ **世界线收束**——一个世界线簇彼此之间高度相似，该簇中的每条世界线会因回归性原理走向同一个未来（即世界线簇呈过去收束，现在发散，未来再次收束的糖果状），只有当相似度降低到一定阈值后才能逃出这个既定的未来；

⑥ **电话微波炉**——男女主自己研发的时间机器：通过电话将当前的信息以电磁波的方式传给过去每个时刻的自己，从而使过去的某个事件奇点的结果发生改变，进而跳跃到另一个世界线的方法，但这种方法无法再返回到原来的世界线，是单行道。于是，男女主打电话时会在时间倒退的同时进行世界线单向跨越，离原世界线越远男女主的时间流逝速度越快（参考相对论中的闵可夫斯基四维空间理论）；

⑦ **α-世界线**——世界线偏移量 0.0000000%，男主（凤凰院凶真）所在的世界线，这条世界线的过去女主（牧濑红莉栖）被某神秘机构射杀，男主为了真相打算通过电话微波炉的方式跨越到β世界线寻找尚未被射杀的女主；

⑧ **β-世界线**——世界线偏移量约 1.0000000%（会有所偏差），女主（牧濑红莉栖）存活，过去未知，未来未知，世界线收束度未知；

⑨ **逆行者**——β-世界线的女主（牧濑红莉栖）在未来遭遇到****【机密信息】后为了回到α-世界线而研发的逆行技术，通过该技术，女主可以通过遗忘部分记忆的方式逆熵而行回到α-世界线的过去****【机密信息】。女主也只能通过电话微波炉退回未来来实现世界线跨越，与男主反向而行；

⑩ **命运石扭结**——凤凰院和红莉栖相遇是命运石之门的选择!男主由过去的α-世界线跨越到未来的β-世界线，而女主从未来的β-世界线跨越回过片的α-世界线，如果男女主在相邻世界线（世界偏移量差值 0.02%）的相同时间（现实时间差值 5s）相遇，即构成命运石扭结。但若两者相互错过，由于他们都无法回到原来世界线，最终只能渐行渐远，但游戏不会结束，因为他们还将在自己的路途上继续走下去，感受不一样的现实。

3.游戏规则

① 男主（凤凰院凶真）顺着时间之矢从过去走向未来，世界偏移量由α-世界线（0%）跨向β-世界线（约 1%）；女主（牧濑红莉栖）逆着时间之矢从未来走向过去世界偏移量由β-世界线（约 1%）跨向α-世界线（0%）；

② 使用电话微波炉后，男主回到过去的同时世界偏移量逐渐增加，女主回到未来的同时世界偏移量逐渐减小；

③ 游戏成功条件：如果男女主在相邻世界线（世界偏移量差值 0.02%）的相同时间（现实时间差值 5s）相遇，则两者相遇。

④ 游戏结束条件：游戏成功、男主或女主跃出屏幕、男主越过β-世界线、女主越过α-世界线达成其中之一。

4.键位设置

游戏进行时：

① 按下旋转编码器按钮：切换男女主视角，当位于其中一人视角时另外一个人会根据当前操作继续前行（不会停下），游戏是男女主并行的。

② 向左旋转：男女主由未来走向过去，即在男主视角时男主正常前行，在女主视角时女主使用电话微波炉进行世界线跨越。只会控制当前视角下的角色的行为方式，不会影响另一个角色；

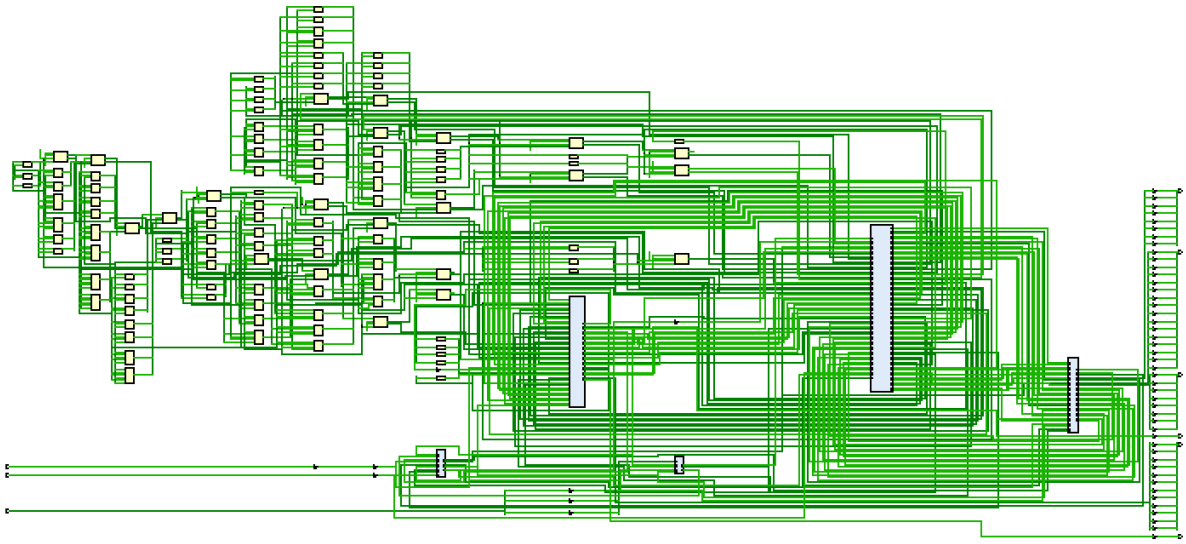
③ 向右旋转：男女主由过去走向未来，即在男主视角时男主使用电话微波炉进行世界线跨越，在女主视角时女主正常前行。只会控制当前视角下的角色的行为方式，不会影响另一个角色。

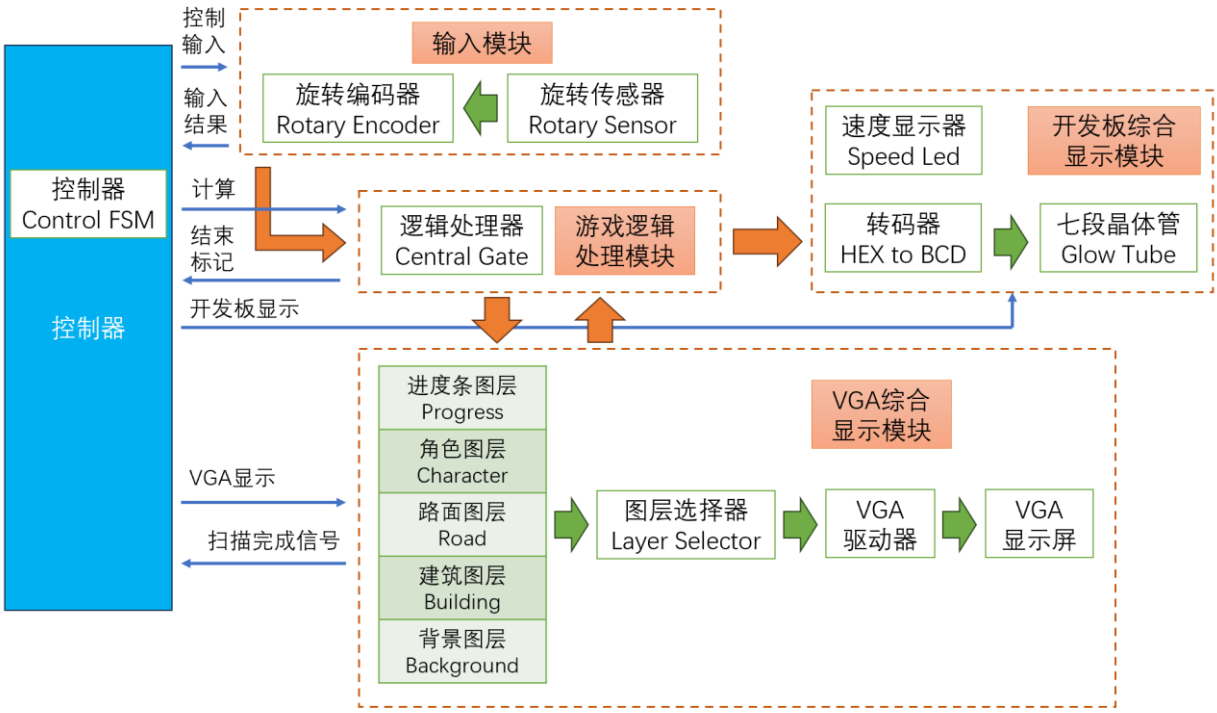
游戏结束时：

① 按下旋转编码器按钮：切换男女主视角；

② 向左/右旋转：开始下一局游戏。

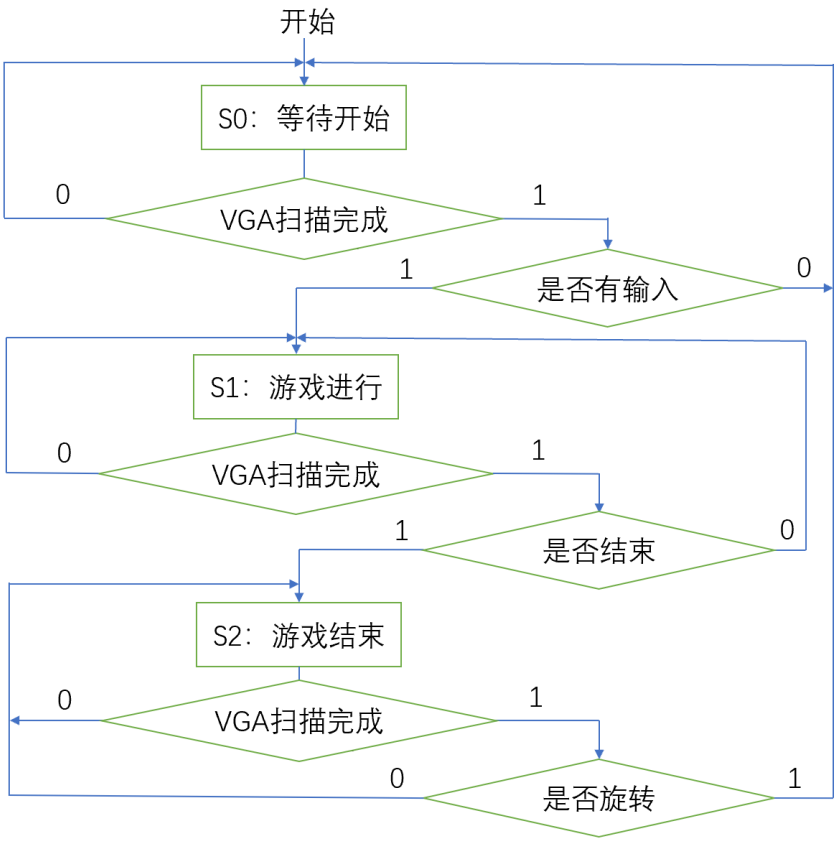
二、命运石扭结小游戏数字系统总框图





三、系统控制器设计

1.ASM 流程图：



2. 状态转移真值表：

PS		NS		转换条件C	
编 码	状态名	编 码	状态名	固定条件	特殊条件
00	Idle	01	Start	VGA扫描完一屏（V = 1）	操作（P = 1）
01	Start	11	Over		结束（O = 1）
11	Over	00	Idle		旋转（R = 1）
10	-	00	Idle	-	-

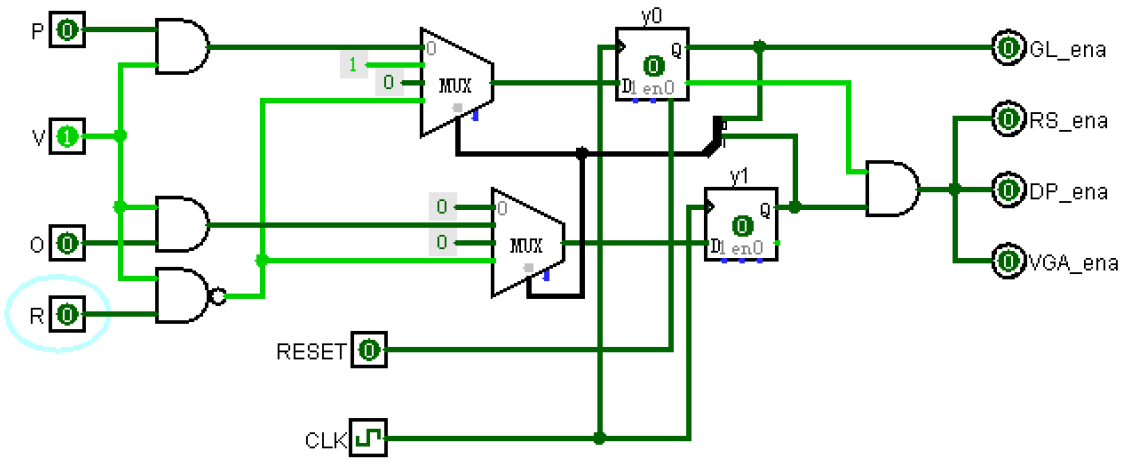
3. 次态激励函数表达式：

$$y_0^{n+1} = y_0^n(\overline{y_1^n} + \bar{V} + \bar{R}) + \overline{y_0^n y_1^n} VP \quad (1)$$

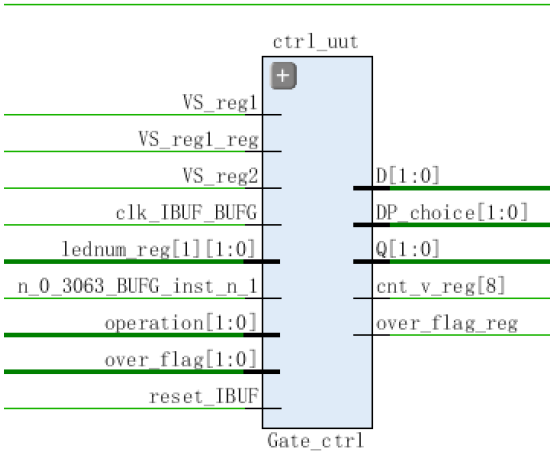
$$y_1^{n+1} = y_1^n y_0^n(\bar{V} + \bar{R}) + \overline{y_1^n} y_0^n VO \quad (2)$$

4. 控制命令逻辑表达式：
①输入模块使能：RS_en = ~y0 & y1;
②逻辑模块使能：GL_en = y0;
③开发板显示模块使能：DP_en = ~y0 & y1;
④VGA 显示模块使能：VGA_en = ~y0 & y1;

5. 系统控制器逻辑方案图：



6.模块功能框图：



四、子系统模块建模

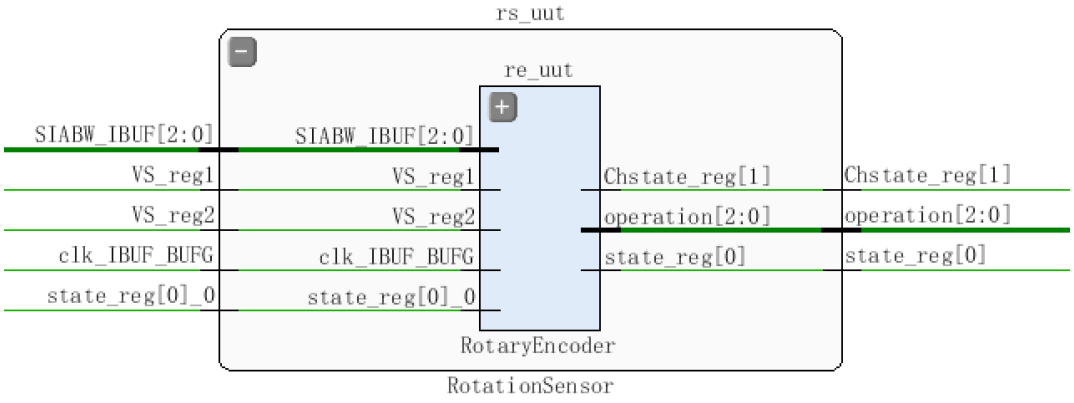
1、旋转编码器模块（RotationSensor.v）

该模块的主要功能是检测和响应旋转编码器的旋转（顺时针、逆时针）和按压动作。

模块功能：

- **RotaryEncoder**：检测旋转编码器的旋转方向和按压状态。
- **RotationSensor**：集成 **RotaryEncoder**，提供统一的接口。

功能框图：



接口信号定义：

- **RotaryEncoder 输入：**
  - **clk**：时钟信号。
  - **rst_n**：复位信号。
  - **SIA、SIB**：旋转编码器的两个相位信号输入。
  - **SW**：按压动作信号输入。
- **RotaryEncoder 输出：**
  - **CW**：顺时针旋转检测输出。
  - **CCW**：逆时针旋转检测输出。
  - **Pressed**：按压动作检测输出。

设计及实现思路：

- **消抖机制**：为了确保稳定的输入信号，采用 10ms 计数器对 SIA 和 SIB 信号进行消抖处理。
- **边沿检测**：通过对消抖后的信号进行上升沿和下降沿检测，来确定旋转方向。
- **行为描述**：根据 A 相的上升沿和下降沿以及 B 相的状态，判断旋转编码器的旋转方向。
- **保持信号**：使用计时器保持 CW 和 CCW 信号的状态，以便于检测旋转动作。

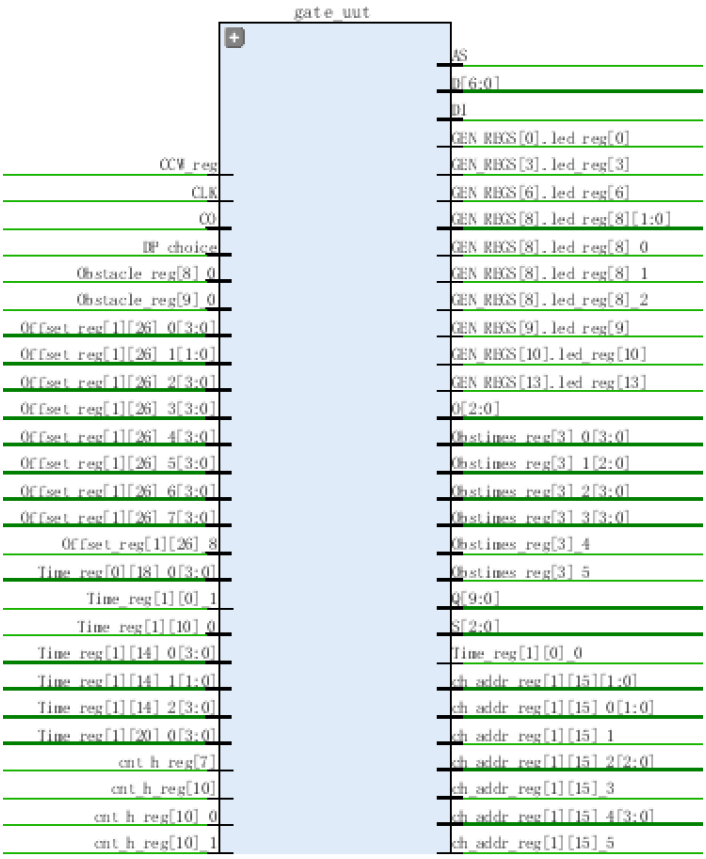
2、游戏逻辑处理模块（meet_logic.v）

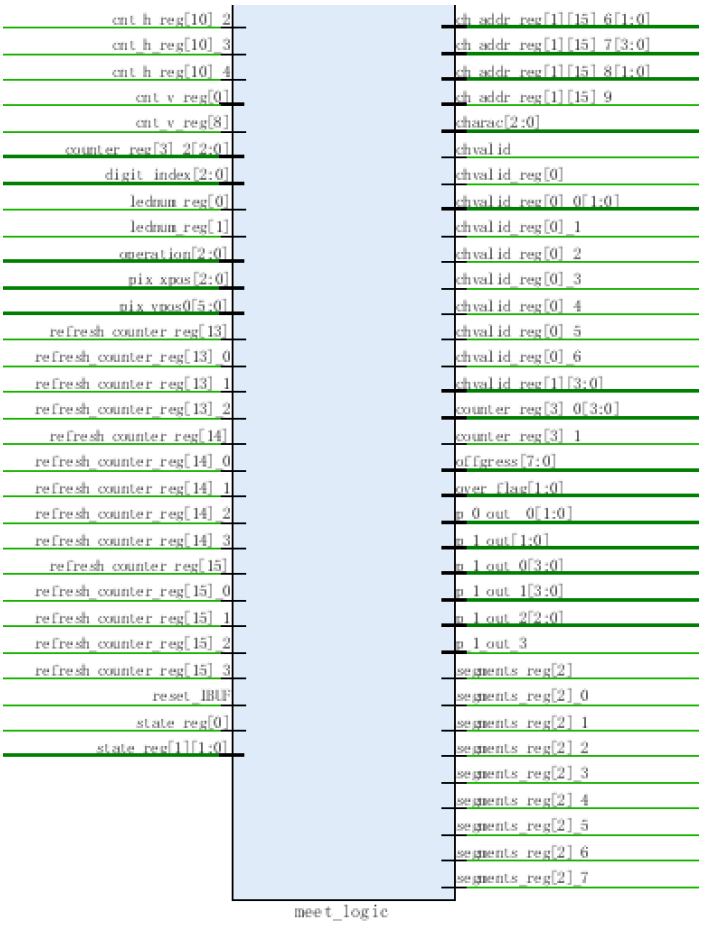
游戏逻辑处理模块是小游戏数字系统的核心部分，负责处理游戏的主要逻辑和状态更新。该模块的设计和实现确保了游戏的流畅运行和用户输入的有效响应。

模块功能：

- 根据用户输入（操作信号）更新游戏角色的状态和位置。
- 计算游戏进度和角色之间的相遇情况。
- 判断游戏结束和游戏胜利条件。

功能框图：





接口信号定义：

- **输入：**
  - **clk**：系统时钟。
  - **rst_n**：复位信号，低电平有效。
  - **next_s**：开始运算记号，高电平激发（频率为 60Hz）。
  - **operation**：操作信号，包括左转、右转和按钮操作。
- **输出：**
  - **character**：当前角色信息及状态。
  - **offset**：世界线偏移量。
  - **speed**：角色移动速度。
  - **obstacle**：另一角色位置。
  - **obstimes**：另一角色清晰度。
  - **progress**：当前时间进度。
  - **offgress**：当前偏移进度。
  - **game_over**：游戏结束标志。
  - **game_win**：游戏胜利标志。

设计及实现思路：

- **角色状态和时间处理**：基于输入信号和内部计时器，更新角色状态和游戏时间。
- **游戏进度和偏移量**：根据角色移动和时间流逝，计算游戏进度和世界线偏

移。

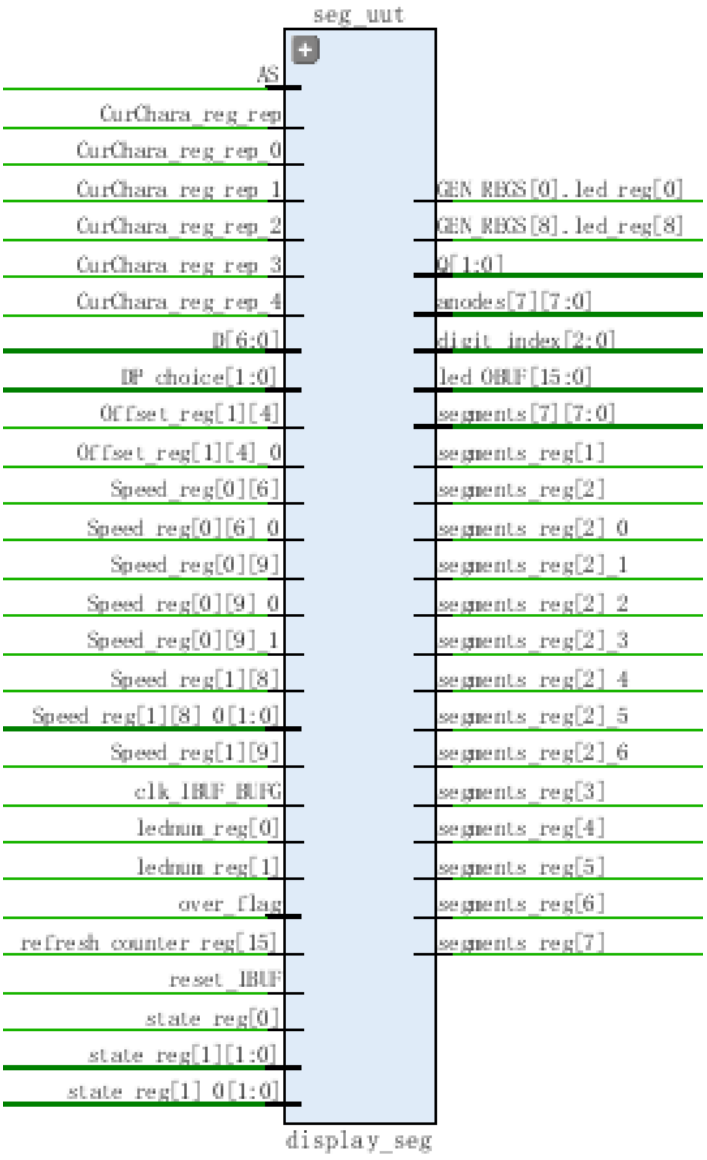
- **胜利和失败条件**：根据角色位置和时间判断游戏的胜利或失败条件。
- **速度和角色相对位置计算**：**calSpeed** 和 **anotherChara** 子模块用于计算角色速度和角色相对位置。

3、开发板综合显示模块（display_seg.v）

该模块整合了以下三个子模块：十六进制转 BCD 编码、七段晶体管显示和速度 LED 灯显示，提供一个多功能的显示界面。该模块的设计展示了如何在单一界面上有效地展示多种信息，同时确保了模块间的高效协作和数据流的正确管理。

模块功能：根据输入信号，将数字信息显示在七段显示器上，同时根据需要显示速度信息在 LED 灯上。

功能框图：



接口信号定义：

- **输入：**
  - **clk**：时钟信号。
  - **reset**：重置信号，低电平有效。
  - **choice**：显示模式选择。
  - **hex**：要显示的 27 位十六进制数字。
  - **speed**：要显示的速度值。
- **输出：**
  - **anodes**：七段显示器的阳极信号。
  - **segments**：七段显示器的段信号。
  - **led**：LED 速度显示器。

设计及实现思路：

- **模块整合**：将 **hex_to_bcd**、**seven_segment_display** 和 **speed_led_display** 模块整合到一个顶层模块中。
- **数据流处理**：**hex_to_bcd** 模块将十六进制数字转换为 BCD 格式，供 **seven_segment_display** 模块使用。
- **显示控制逻辑**：根据 **choice** 信号选择不同的显示模式。
- **速度显示**：**speed_led_display** 模块根据输入的速度值控制 LED 灯的显示。

3-1、十六进制转 bcd 编码（hex2bcd.v）

用于有效地转换数值格式，以便于将游戏逻辑处理器模块传出的十六进制信息转换为七段晶体管便于解析的 bcd 码编码形式。

模块功能：转换一个 27 位的十六进制输入为 32 位的 BCD（二进制编码的十进制）输出。

接口信号定义：

- **输入：**
  - **hex**：27 位的十六进制数。
- **输出：**
  - **bcd**：转换后的 32 位 BCD 码（8 位，每位 4-bit）

设计及实现思路：

- **初始化**：将 BCD 输出初始化为 0。
- **十六进制处理**：对每个十六进制输入位进行处理。
- **BCD 转换：**
  - 遍历 BCD 码中的每个 4-bit 数字，若大于 4，则加 3（这是因为在十进制中，每个数字最大为 9，而在 BCD 中每个 4-bit 数字能表示的最大值是 15）。
  - 将 BCD 码左移一位，将十六进制的最高位加入到 BCD 的最低位。

3-2、 七段晶体管显示（seven_segment_display.v）

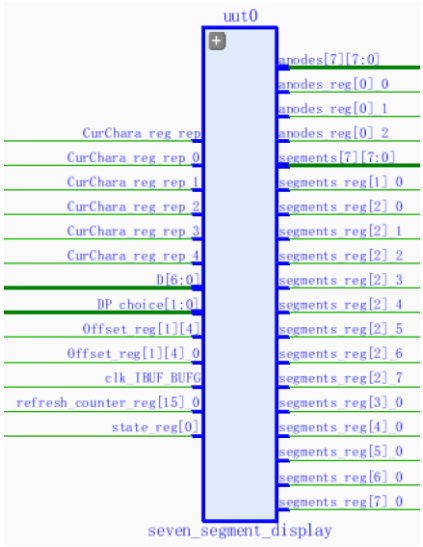
七段晶体管显示模块是小游戏数字系统中的重要组件，负责将数字和文字信息以

视觉友好的方式显示在七段显示器上。模块的设计充分考虑了显示的灵活性和动态特性，以适应不同的显示需求。

模块功能：

- 根据输入的 BCD 码和选择信号，控制七段显示器的显示内容。
- 将 BCD 码表示的 8 位数字显示在七段晶体管上
- 将特殊文字（STEN.GATE 及 GAME.OVER）显示在七段晶体管上以显示游戏状态

功能框图：



接口信号定义：

- **输入：**
  - **clk**：时钟信号。
  - **reset**：重置信号，低电平有效。
  - **choice**：选择显示内容。
  - **bcdnum**：要显示的 8 位数字（每个数字 4 位）。
- **输出：**
  - **anodes**：阳极信号（AN0..AN7）。
  - **segments**：段信号（A..G）和小数点 DP。

设计及实现思路：

- **分频计数器**：用于创建显示刷新率。
- **七段解码器**：将输入的 BCD 码转换为七段显示器的段信号。
- **显示内容控制**：根据 **choice** 信号选择显示不同的内容，包括普通的数字解码、特定文字（例如"STEN_GATE"或"GAME_OVER"）。
- **动态显示**：通过快速切换不同的显示位（anodes），实现动态显示

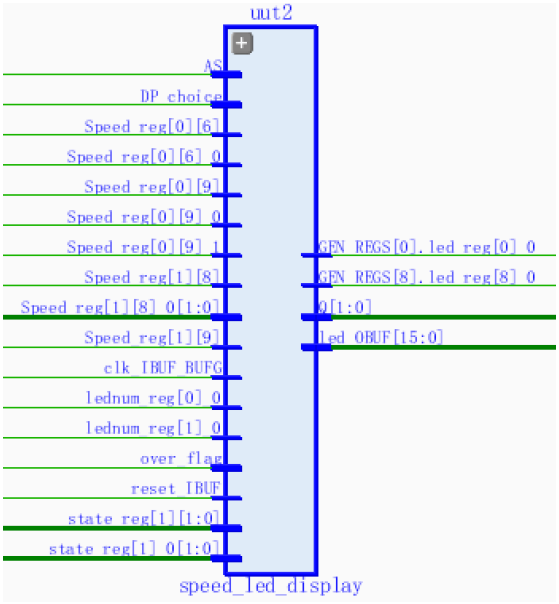
3-3、速度 LED 灯显示（speed_led_display.v）

该模块的主要功能是通过 LED 灯以可视化的方式显示当前的速度，并可实现闪烁效果。该模块的设计考虑了用户交互体验，通过灯光闪烁和数量变化直观地展示速度信息。

模块功能：

- 根据输入的速度值控制 LED 灯的点亮情况，可实现闪烁效果。
- 当显示速度时，会有相应数量的 LED 灯自左向右亮起，直观地表现当前行进速度大小，和 VGA 显示器相结合有利于玩家掌握情况。
- 当游戏开始或结束时，会依次点亮 LED 灯，呈现加载及波浪效果，使显示更加美观

功能框图：



接口信号定义：

- **输入：**
  - **clk**：时钟信号。
  - **ena**：使能信号。
  - **flicker**：闪烁效果控制信号。
  - **speed**：速度值（4 位）。
- **输出：**
  - **led**：LED 显示灯（16 位）。

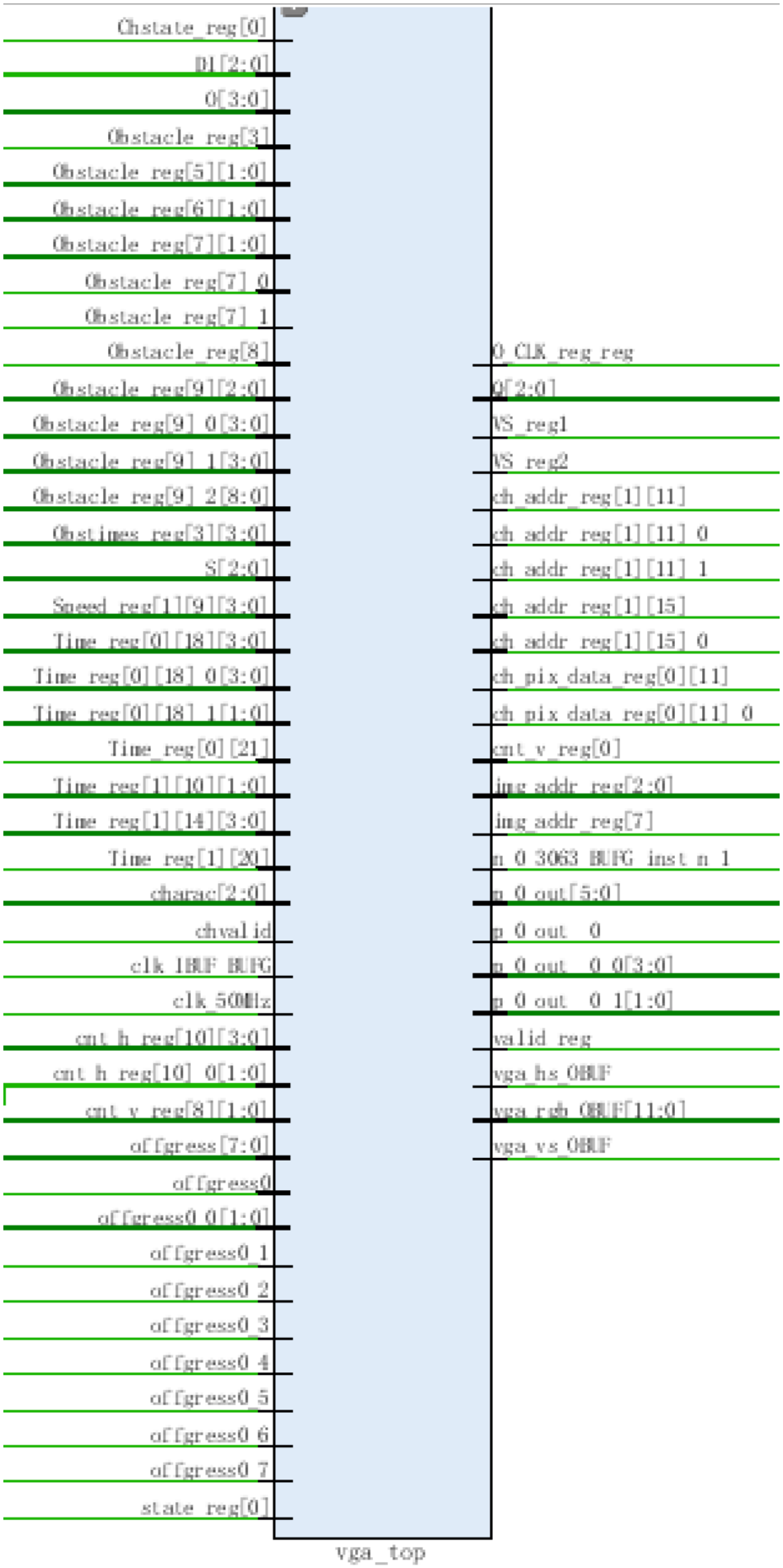
设计及实现思路：

- **时钟分频**：利用分频器产生低频时钟信号，控制 LED 灯的闪烁频率。
- **闪烁效果控制**：使用一个计数器来实现 LED 灯的闪烁效果。
- **速度显示逻辑**：根据速度值点亮对应数量的 LED 灯。当 **flicker** 为真时，仅显示闪烁的单个 LED 灯；否则根据速度值点亮相应数量的 LED 灯。
- **生成语句**：使用 Verilog 的 generate 语句创建 16 个独立控制的 LED 灯。

4、VGA 综合显示模块（vga_top.v）

该模块是 VGA 子系统的顶层模块，负责整合时钟分频、图层综合和 VGA 驱动器模块，以在 VGA 显示器上呈现游戏内容。

功能框图：


Chstate_reg[0]
D1[2:0]
O[3:0]
Obstacle_reg[3]
Obstacle_reg[5][1:0]
Obstacle_reg[6][1:0]
Obstacle_reg[7][1:0]
Obstacle_reg[7]_0
Obstacle_reg[7]_1
Obstacle_reg[8]
Obstacle_reg[9][2:0]
Obstacle_reg[9]_0[3:0]
Obstacle_reg[9]_1[3:0]
Obstacle_reg[9]_2[8:0]
Obstimes_reg[3][3:0]
S[2:0]
Speed_reg[1][9][3:0]
Time_reg[0][18][3:0]
Time_reg[0][18]_0[3:0]
Time_reg[0][18]_1[1:0]
Time_reg[0][21]
Time_reg[1][10][1:0]
Time_reg[1][14][3:0]
Time_reg[1][20]
charac[2:0]
chvalid
clk_IBUF_BUFG
clk_50Mhz
cnt_h_reg[10][3:0]
cnt_h_reg[10]_0[1:0]
cnt_v_reg[8][1:0]
offgress[7:0]
offgress0
offgress0_0[1:0]
offgress0_1
offgress0_2
offgress0_3
offgress0_4
offgress0_5
offgress0_6
offgress0_7
state_reg[0]
0_CLK_reg_reg
0[2:0]
VS_reg1
VS_reg2
ch_addr_reg[1][11]
ch_addr_reg[1][11]_0
ch_addr_reg[1][11]_1
ch_addr_reg[1][15]
ch_addr_reg[1][15]_0
ch_pix_data_reg[0][11]
ch_pix_data_reg[0][11]_0
cnt_v_reg[0]
img_addr_reg[2:0]
img_addr_reg[7]
n_0_3063_BUFG_inst_n_1
n_0_out[5:0]
n_0_out_0
n_0_out_0_0[3:0]
n_0_out_0_1[1:0]
valid_reg
vga_hs_OBUF
vga_rgb_OBUF[11:0]
vga_vs_OBUF
vga_top

接口信号定义：

- 输入：
  - **clk_100MHz**：标准时钟。
  - **rst_n**：复位信号。
  - **chara**、**speed**、**obstacle**、**obstimes**、**progress**、**offgress**：游戏相关的状态和数据。
- 输出：
  - **vga_hs**：行同步信号。
  - **vga_vs**：场同步信号。
  - **vga_rgb**：RGB 颜色输出。
  - **VS_negedge**：场同步信号的下降沿

设计及实现思路：

- **时钟分频**：使用 **divider_2** 模块将 100MHz 时钟分频为 50MHz，以符合 VGA 驱动器的时钟需求。
- **图层综合**：使用 **VGA_game_layers_syn** 模块整合游戏的不同图层，生成最终的像素数据。
- **VGA 驱动**：使用 **vga_driver_1024x600** 模块根据像素数据生成 VGA 信号，控制显示器的显示。

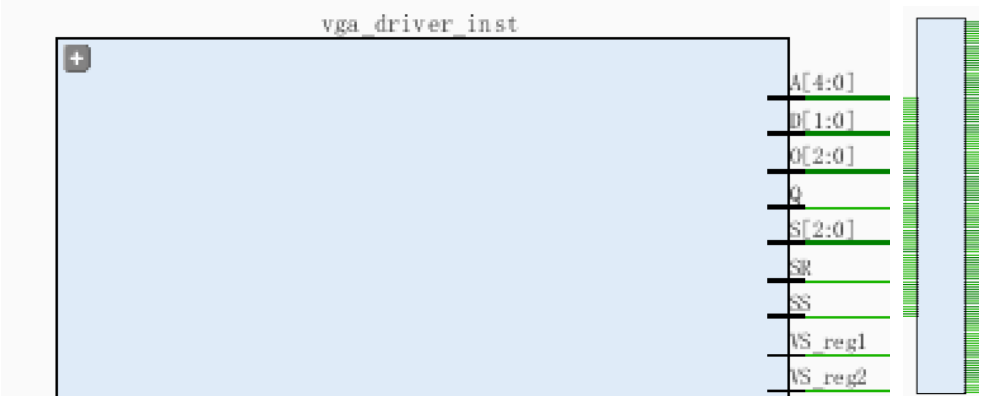
4-1、 VGA 驱动器（vga_driver_1024x600.v）

该模块负责根据输入的像素数据生成 VGA 信号，以便在显示器上渲染图像。该模块的设计考虑了显示器的时序要求，并确保了图像的正确渲染和同步显示。

模块功能：

- 根据时钟信号生成 VGA 的行同步（**vga_hs**）和场同步（**vga_vs**）信号。
- 计算并输出当前像素的坐标（**pix_xpos**, **pix_ypos**）。
- 控制 VGA 显示的像素颜色（**vga_rgb**）。
- 扫描完一页后会传出场下降沿信号（**VS_negedge**），该信号负责指挥控制器和游戏逻辑处理器开始下一步计算，以免在扫描页面过程中改变状态使画面不完整。

功能框图：



接口信号定义：

- **输入：**
  - **clk_50MHz**：VGA 驱动时钟。
  - **rst_n**：复位信号。
  - **pix_data**：像素点数据。
- **输出：**
  - **vga_hs**：行同步信号。
  - **vga_vs**：场同步信号。
  - **vga_rgb**：红绿蓝输出。
  - **pix_xpos**：像素点横坐标。
  - **pix_ypos**：像素点纵坐标。
  - **disp_en**：显示有效信号。
  - **VS_negedge**：场信号下降沿。

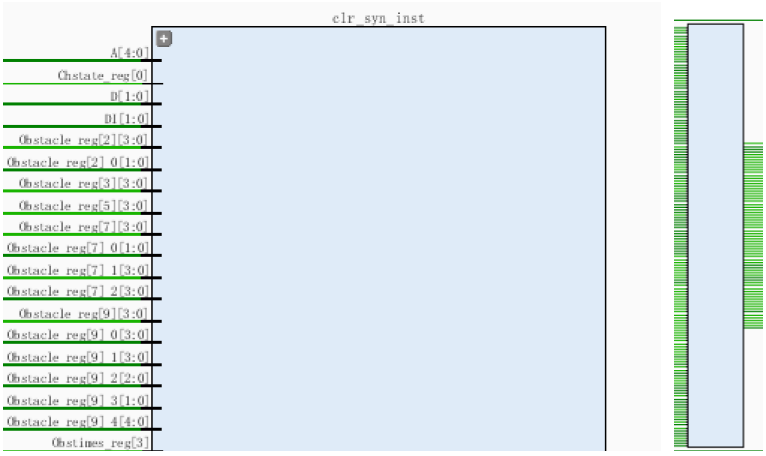
设计及实现思路：

- **时序参数**：根据具体显示器的要求设置行场同步及显示区域的参数。
- **行场同步信号**：利用时钟信号和内部计数器生成行场同步信号。
- **像素坐标计算**：在显示有效区域内计算当前像素的坐标。
- **RGB 数据输出**：当显示有效时，根据像素数据输出 RGB 颜色。
- **场信号下降沿检测**：监测场同步信号的下降沿，用于触发特定事件。

4-2、 图层综合模块（VGA_game_layers_syn.v）

该模块整合了多个图层以生成最终的 VGA 显示输出，目前包括背景、角色、路面、角色和进度条一共五个图层。该模块展示了如何在单一界面上有效地展示多种图像元素，同时确保了图层之间的正确合成和优先级处理。

功能框图：



接口信号定义：

- **输入：**
  - **vga_clk**：VGA 驱动时钟。
  - **disp_en**：显示有效信号。
  - **chara**、**speed**、**obstacle**、**obstimes**、**progress**、**offgress**：角色、速度、障碍物位置和大小、时间进度和偏移进度等游戏参数。

    - **pix_xpos**、**pix_ypos**：像素点坐标。
- **输出**：
    - **pix_data**：合成后的像素点数据。

设计及实现思路：

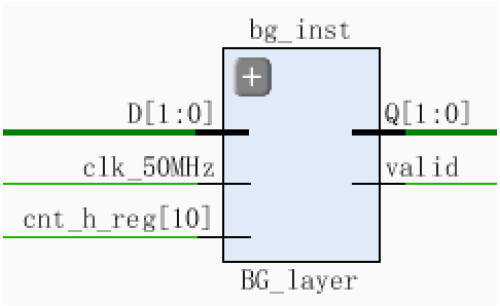
- **图层合成**：将各个图层（如背景、角色等）的像素数据根据优先级合成为最终显示的像素点。
- **有效性判断**：每个图层都有一个有效性标志，指示在当前像素点上该图层是否有数据要显示。
- **像素数据选择**：根据图层的有效性和优先级选择要显示的像素数据。
- **模块实例化**：实例化每个独立的图层模块，如背景、角色等。

各级图层（Game_Layers.v）

4-2-1、背景图层（BG_layer）

功能：负责渲染游戏的背景。

功能框图：



接口信号定义：

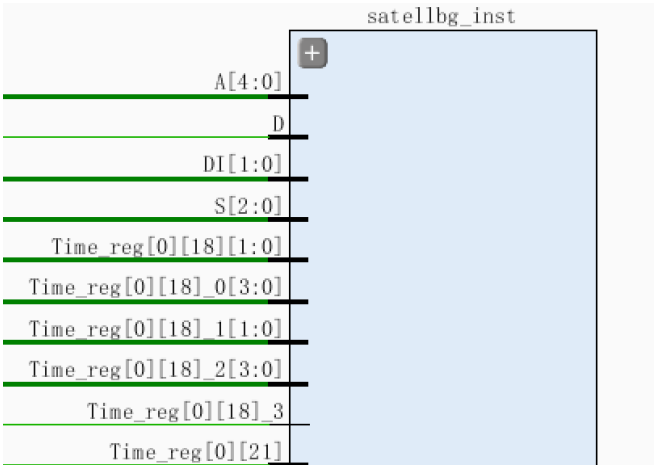
- **输入**：
    - **vga_clk**：VGA 驱动时钟。
    - **disp_en**：显示有效信号。
    - **pix_xpos**、**pix_ypos**：像素点坐标。
- **输出**：
    - **valid**：标记当前像素点是否有效。
    - **pix_data**：背景像素点数据。

实现思路：在有效显示区域内，为每个像素点分配固定颜色（例如蓝色），以渲染统一的背景色。

4-2-2、建筑物图层（Satellbg_layer）

功能：根据游戏进度动态渲染建筑物或卫星图像。从该图层开始会对图片进行透明判断，如果该图片某个位置色素为特定值（比如 12’hfff 纯白、12’h0ff 纯黄、12’h0f0 纯绿等）时该位置视为透明，这个特定值应该取整张图出现的最少的颜色，防止有效的位置被透明化处理。

功能框图：



接口信号定义：

- **输入：**
  - **vga_clk**：VGA 驱动时钟。
  - **disp_en**：显示有效信号。
  - **progress**：当前角色时间坐标，根据改坐标计算建筑物位置。
  - **pix_xpos**、**pix_ypos**：像素点坐标。
- **输出：**
  - **valid**：标记当前像素点是否有效。
  - **pix_data**：建筑物或卫星的像素点数据。

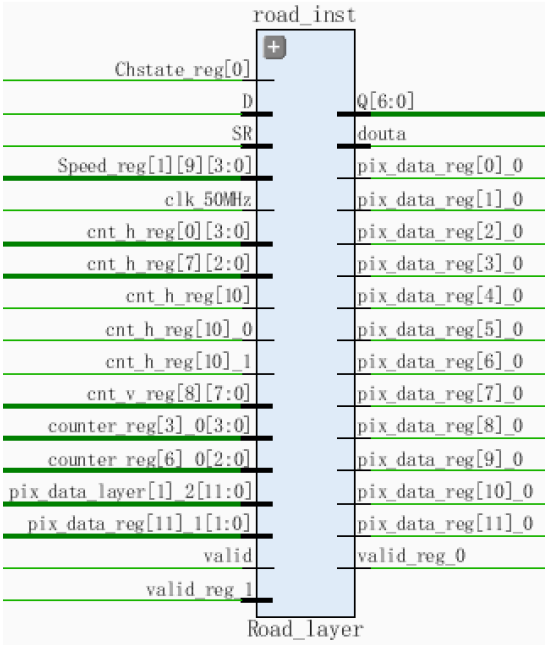
实现思路：

- 根据游戏进度选择显示建筑物或卫星图像。
- 通过读取 ROM 中的图像数据并根据像素坐标渲染相应的图像。

4-2-3、路面图层（Road_layer）

功能：动态渲染游戏中的路面，表现角色的移动。

功能框图：



接口信号定义：

- **输入：**
  - **vga_clk**：VGA 驱动时钟。
  - **disp_en**：显示有效信号。
  - **forward**：角色的移动方向。
  - **speed**：角色的移动速度。
  - **pix_xpos**、**pix_ypos**：像素点坐标。
- **输出：**
  - **valid**：标记当前像素点是否有效。
  - **pix_data**：路面的像素点数据。

实现思路：

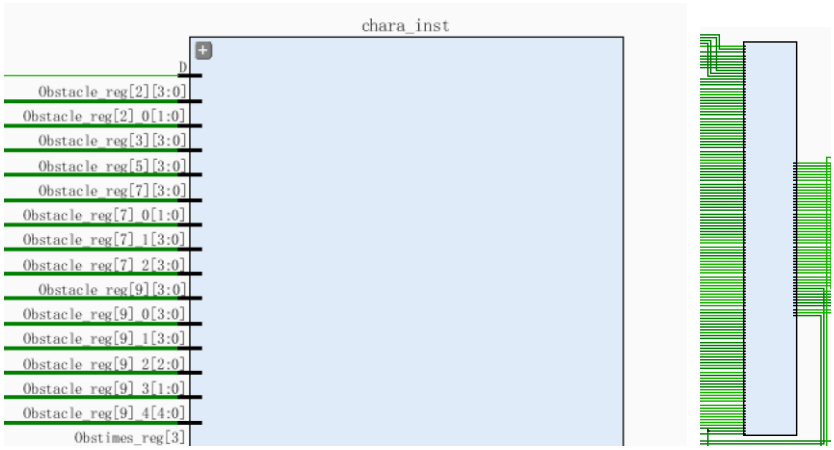
- 根据角色的速度和方向动态调整路面图像的位置，以模拟路面的移动。
- 利用时钟分频实现路面缓慢左移/右移的效果，通过路面移动来体现角色的移动。
- 从 ROM 中读取路面图像数据并根据像素坐标进行渲染。

4-2-4、角色及进度条图层（Chara_layer 游戏核心图层）

这个模块是游戏的核心图层，它不仅显示当前角色和另一个角色，而且还包括进度条的显示。其设计重点在于用尽可能少的 ROM 实例化来节省开发板的有限空间，同时有效地读取和处理四张图片的信息。这个模块的实现展示了如何在有限资源下创造丰富的游戏视觉效果，提高了游戏的整体体验和互动性。

模块功能：显示当前角色、另一个角色和进度条，以及模糊效果。

功能框图：



接口信号定义：

- **输入：**
  - **vga_clk**：VGA 驱动时钟。
  - **disp_en**：显示有效信号。
  - **chara**：选择角色。
  - **obstacle**：另一个角色的屏幕位置。
  - **obstimes**：另一个角色的清晰度。
  - **progress**：游戏进度。

    - **offgress**：偏移进度。
    - **pix_xpos**、**pix_ypos**：像素点坐标。
- **输出**：
    - **valid**：标记当前像素点是否有效。
    - **pix_data**：角色或进度条的像素点数据。

设计及实现思路：

- **内部图层处理**：内置两个图层来分别处理当前角色和另一个角色。
- **角色显示逻辑**：
    - 根据角色状态和位置动态读取和显示角色图像。
    - 使用模糊效果来表现角色的清晰度。
- **进度条显示逻辑**：
    - 根据游戏进度动态显示进度条。
    - 进度条显示位置根据角色的进度来调整。
- **资源优化**：通过智能管理 ROM 资源，减少对存储空间的需求。

五、测试模块建模

stimulation 拟真测试

1. 控制器模块测试（Gate_ctrl_tb）
    - **目标**：验证控制器模块 **Gate_ctrl** 的功能，确保它能正确响应旋转编码器的操作、游戏逻辑模块的结束信号以及 VGA 完成信号。
    - **测试逻辑**：
        - 初始化所有输入信号并模拟系统时钟。
        - 应用不同的测试场景，如游戏开始、游戏过程中的操作、游戏结束等。
        - 观察输出使能信号和显示类型信号的变化，确保控制逻辑正确。

2. 游戏逻辑模块测试（Gate_logic_tb）
    - **目标**：测试游戏逻辑模块 **Gate_logic2（该模块后来被舍弃）**的功能，以确保它能正确处理游戏的逻辑操作并生成相应的输出。
    - **测试逻辑**：
        - 模拟系统时钟，初始化所有输入信号。
        - 模拟游戏过程中的操作，如角色移动、障碍物出现等。
        - 观察游戏角色状态、游戏进度、障碍物位置等输出信号的变化。
        - 特别关注游戏结束和胜利条件的触发，以及相应的输出信号变化。

3. 旋转编码器模块测试（RotaryEncoder_tb）
    - **目标**：验证旋转编码器模块的功能，确保它能正确识别和响应旋转和按压动作。
    - **测试逻辑**：
        - 初始化输入信号并模拟时钟。

- 通过更改 SIA 和 SIB 的状态，模拟旋转编码器的顺时针和逆时针旋转。
- 模拟按压动作，并观察 Pressed 信号的变化。
- 观察 CW 和 CCW 信号以验证旋转方向的检测。

4.七段晶体管测试模块（seven_segment_display_tb）

- **目标：**测试七段晶体管显示模块的功能，确保它能正确显示不同的数字和字符。
- **测试逻辑：**
  - 生成时钟信号并初始化输入。
  - 在不同的测试用例中设置不同的数字或字符，并观察 anodes 和 segments 的输出，以确保正确显示。

5. VGA 驱动器测试（VGA_driver_tb）

- **目标：**验证 VGA 驱动器模块的功能，确保它能正确生成 VGA 信号。
- **测试逻辑：**
  - 生成时钟信号并初始化复位。
  - 观察水平同步（hsync）和垂直同步（vsync）信号，以及 RGB 输出，以验证 VGA 信号的正确生成。

test 模块下板测试

1. 综合测试模块（Gate_test）

- **目标：**在实际硬件上测试除控制器之外的组件。
- **功能：**整合旋转编码器输入、游戏逻辑、七段显示和 VGA 显示模块，以测试它们在实际硬件环境下的性能和互动。
- **测试逻辑：**
  - 通过旋转编码器模拟用户输入，检测操作反馈。
  - 观察游戏逻辑的响应，如角色状态、速度、障碍物等。
  - 在七段显示和 VGA 显示上观察输出结果，确保显示正确。

2. 开发板综合显示测试模块（test_ssd）

- **目标：**测试开发板上的七段显示和 LED 速度显示器。
- **功能：**通过不同的输入模拟各种显示场景，如不同的数字和速度值。
- **测试逻辑：**
  - 根据输入的数字和速度值，观察七段显示器和 LED 显示器的输出。
  - 检查是否所有数字和速度值都能正确显示。

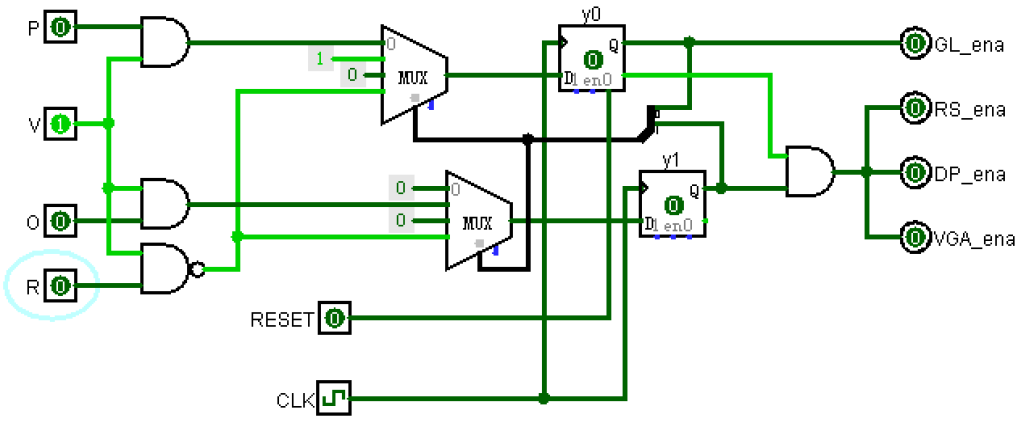
3. VGA 综合显示测试模块（vga_test）

- **目标：**在实际硬件上测试 VGA 显示功能。
- **功能：**模拟游戏中的角色移动和障碍物，检测 VGA 显示输出。
- **测试逻辑：**
  - 通过改变角色状态、速度、障碍物位置和大小，模拟游戏场景。
  - 观察 VGA 显示器上的输出，确保图像正确显示。

六、实验结果

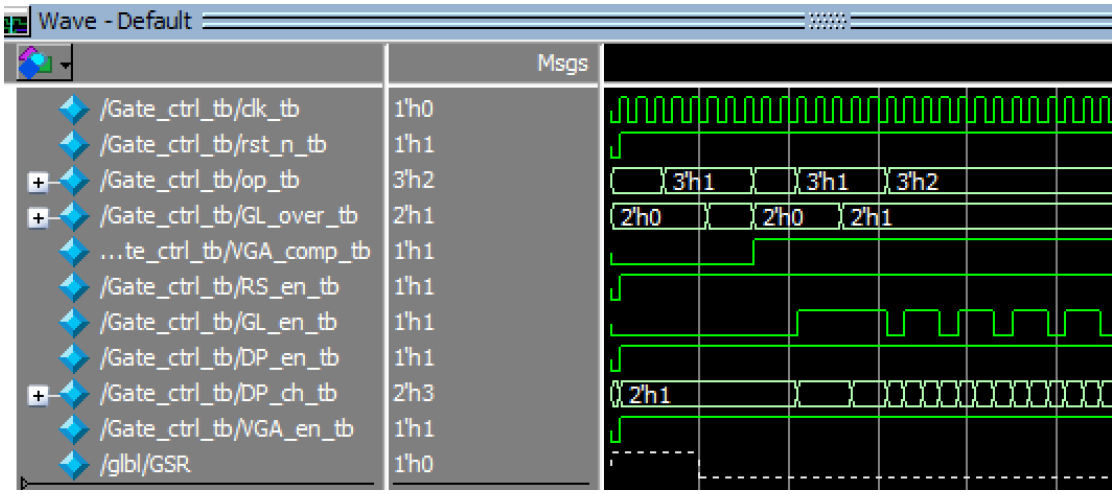
logisim 逻辑验证图

控制器状态机：



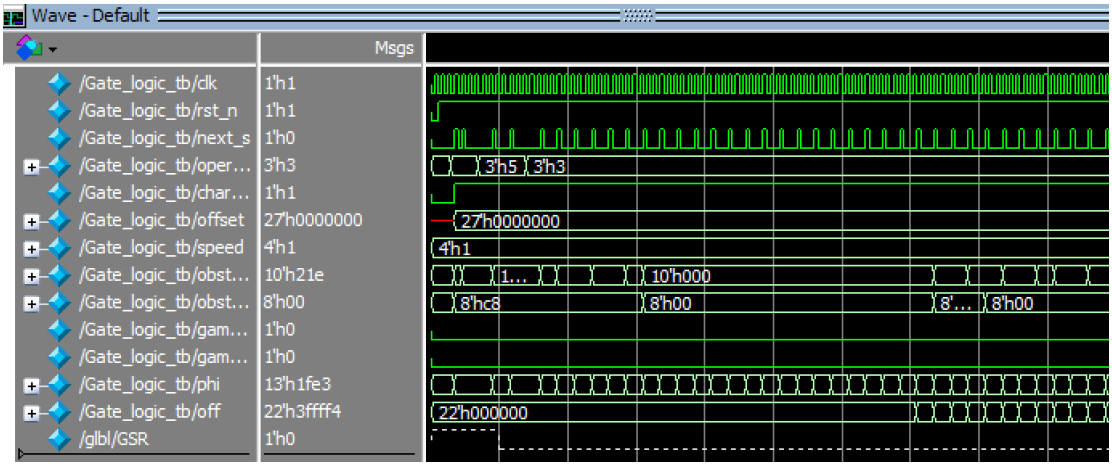
modelsim 仿真波形图

1、控制器模块测试



控制器状态转换有效，在三态之间转换。

2、游戏逻辑模块测试



与第二个游戏逻辑模型的逻辑相契合，但该模型因为操作难度过高后续被舍弃。

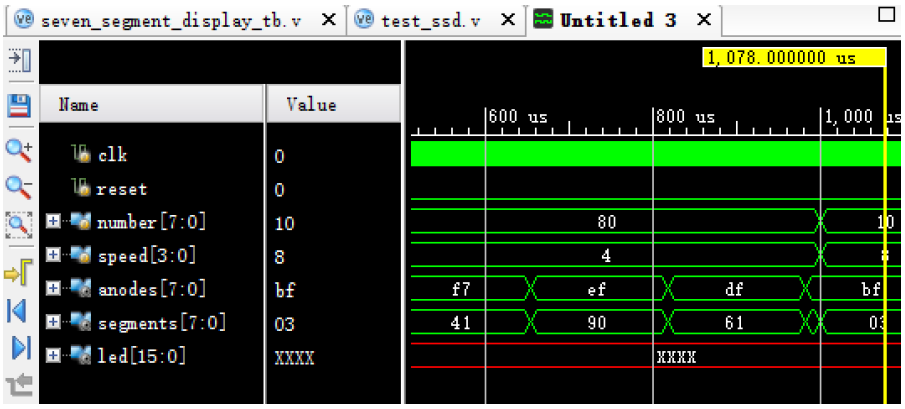
3、旋转编码器模块测试

加消抖前的测试没截图

加消抖后：

拟真频率与真实情况不符合，看不出来，但下板可行

4、七段晶体管模块测试



测试时模型与最终模型有所变更，七段晶体管显示符合逻辑，下板后可正常显示。

5、VGA 模块测试模块

该测试原始模块已遗失，故只展示最终下板效果。

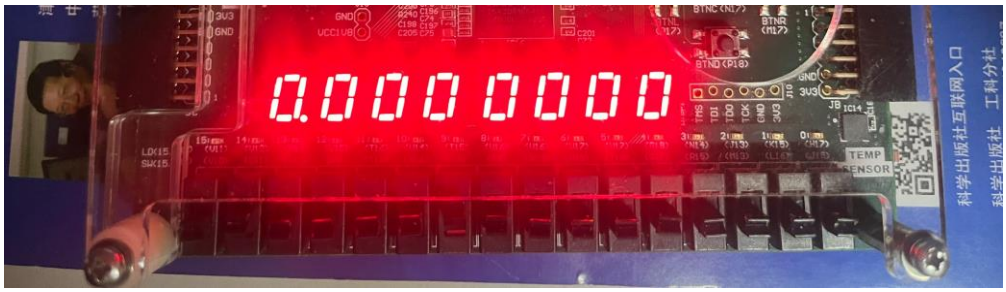
下板后的实验结果贴图

1. 综合测试模块（Gate_test.bit）——详细见视频

这是命运的相遇!!!

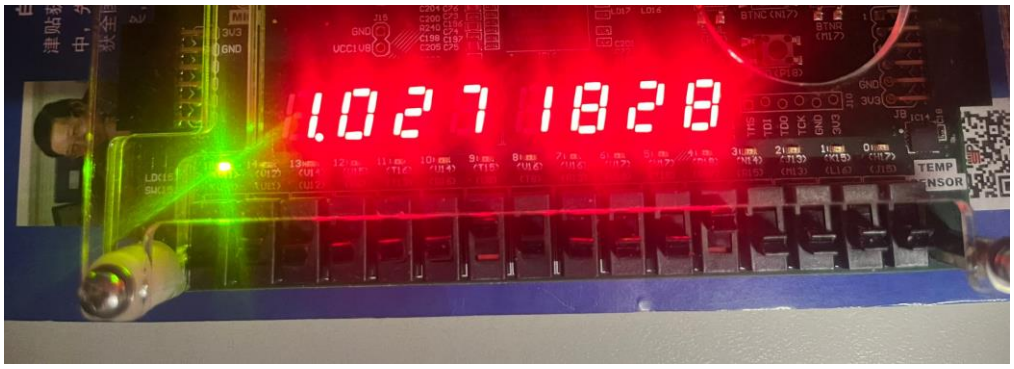
ec52bd344f9675153b1fbedd989cdbd1.mp4

2. 开发板综合显示测试模块（test_ssd.bit）



全零初始化

显示任意数字（led 灯不显示）



显示任意数字（led 灯显示当前速度）

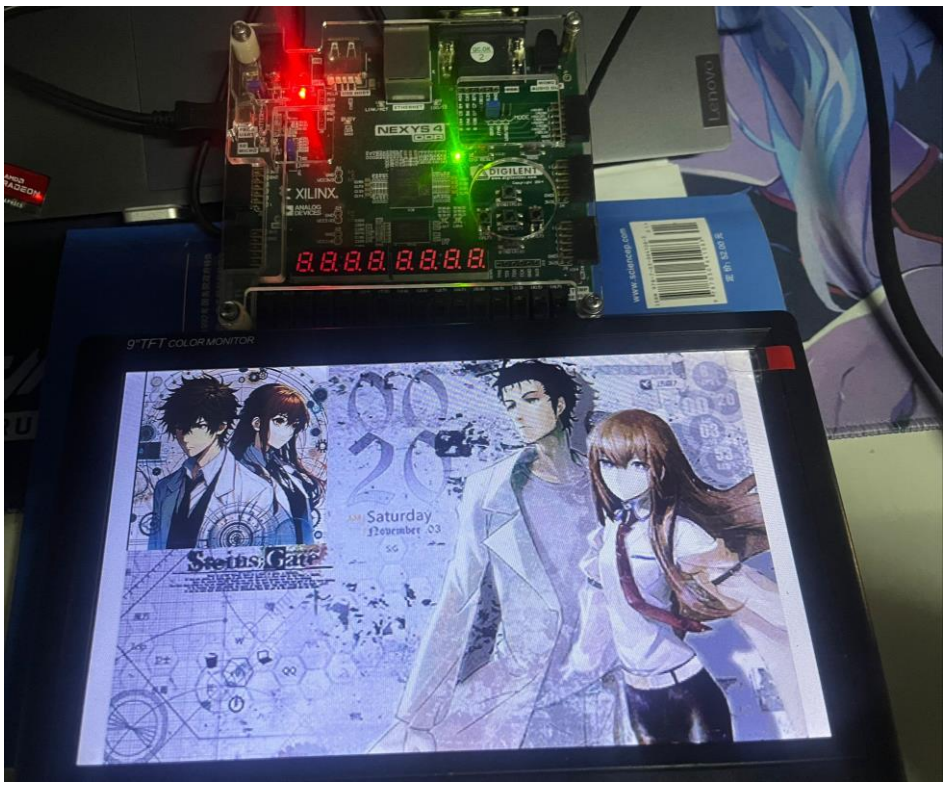
显示文字 STEN.GATE

（led 轮流显示，成流动效果）

显示文字 GAME.OVER

（led 轮流显示，成流动效果）

3. VGA 综合显示测试模块（vga_test.bit）

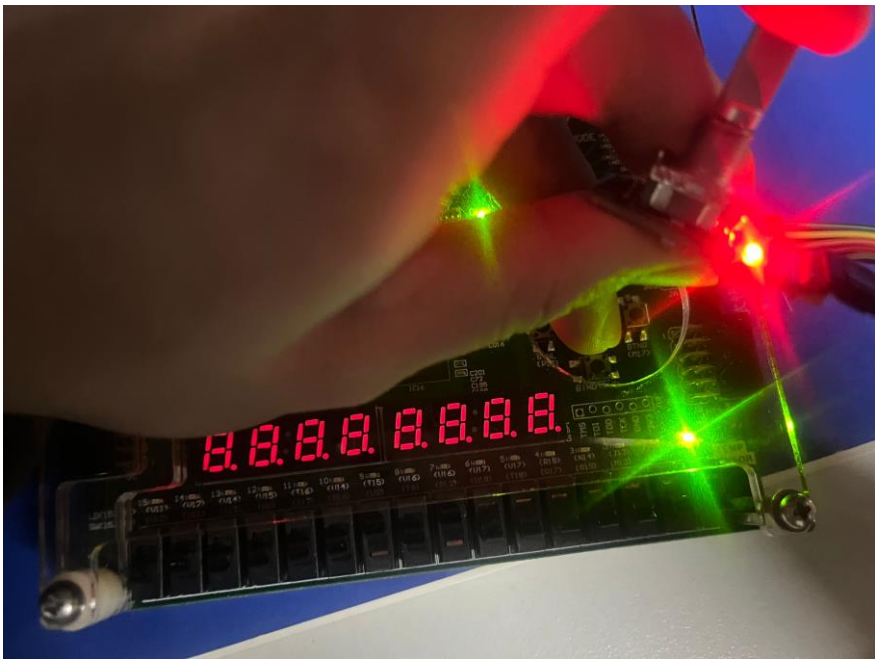


放大一倍的 480x300 图片+一个原分辨率显示的 300x300 图片（有图层效果）

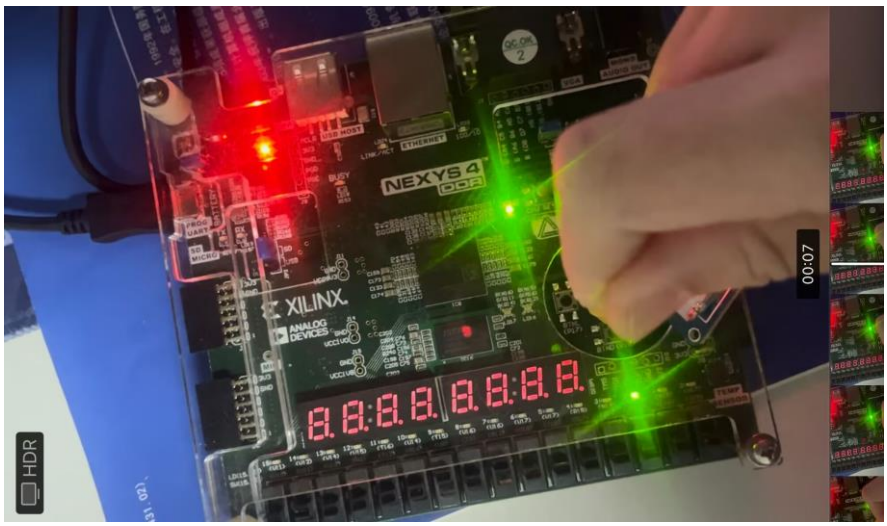
这是开发板储存空间的极致了，由于开发板空间有限，后续更换为更小的图片，这两张图片放此缅怀，如果加上 SD 卡就可以显示游戏开始画面了，但我懒得加了 qwq

4. 旋转编码器测试模块（RotationEncoder.bit）

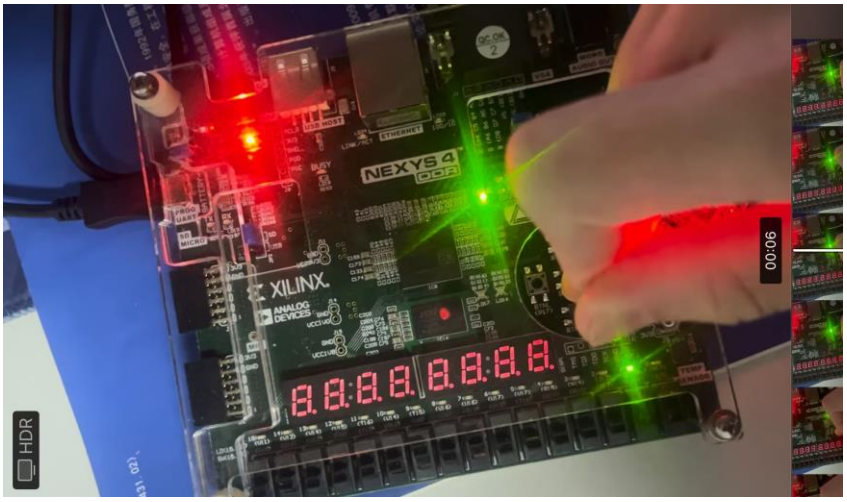
需要一手按住 rst_n 才可以感应



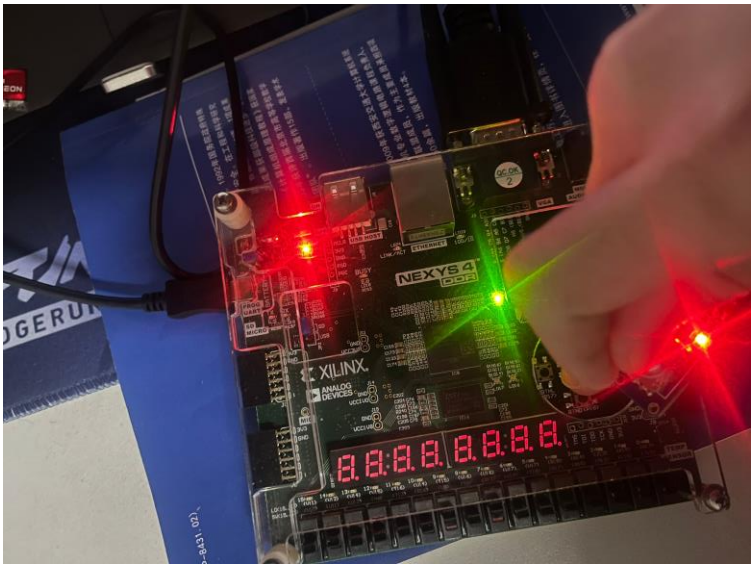
检测按压



检测左旋（由于小灯闪烁太快，只能截取视屏）

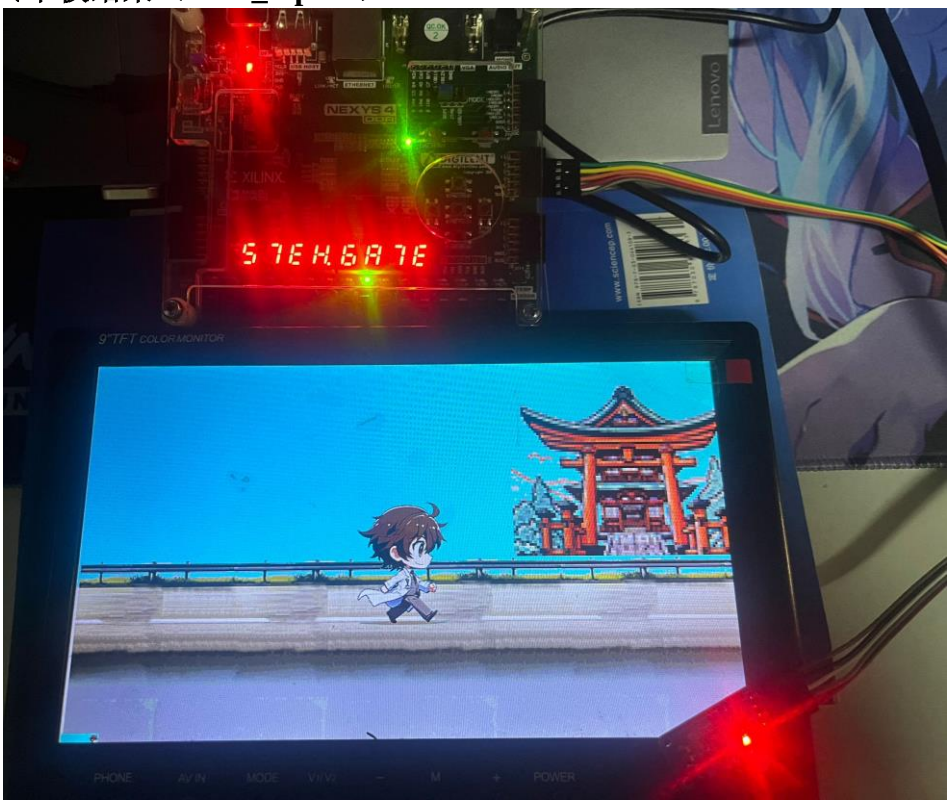


检测右旋（由于小灯闪烁太快，只能截取视频）



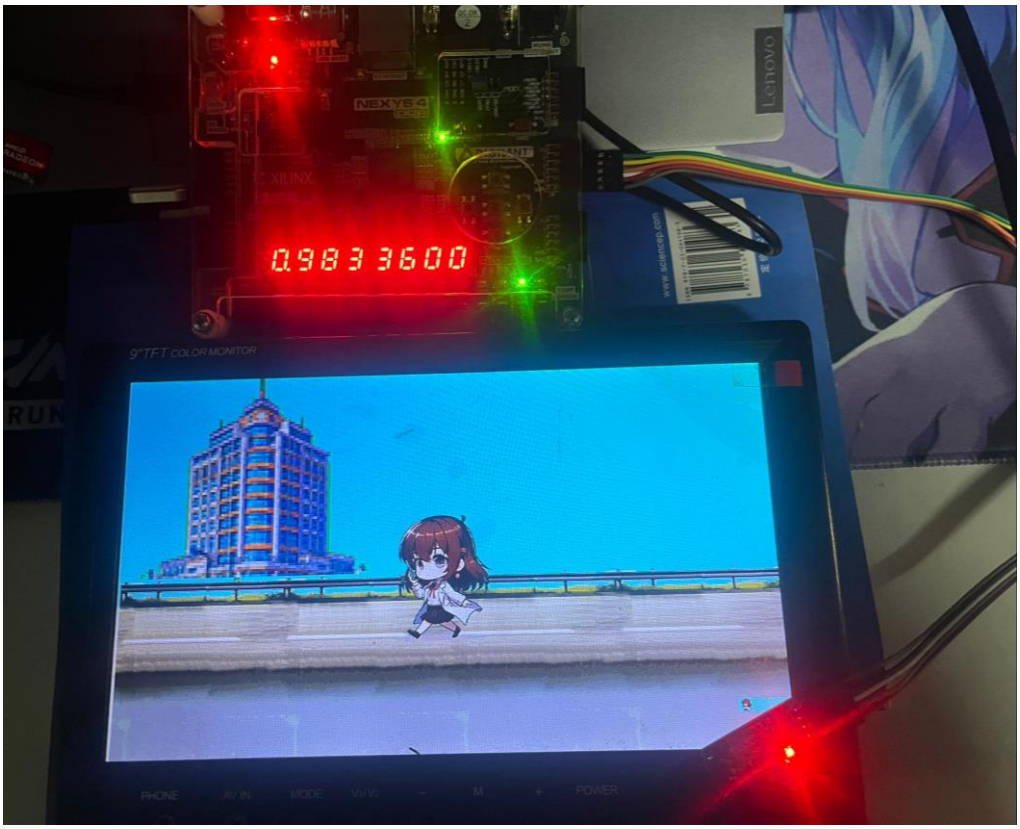
每次滚动只会闪烁 0.1s，直接拍照捕捉不到

5. 最终下板结果（Gate_top.bit）



游戏开始，男主视角

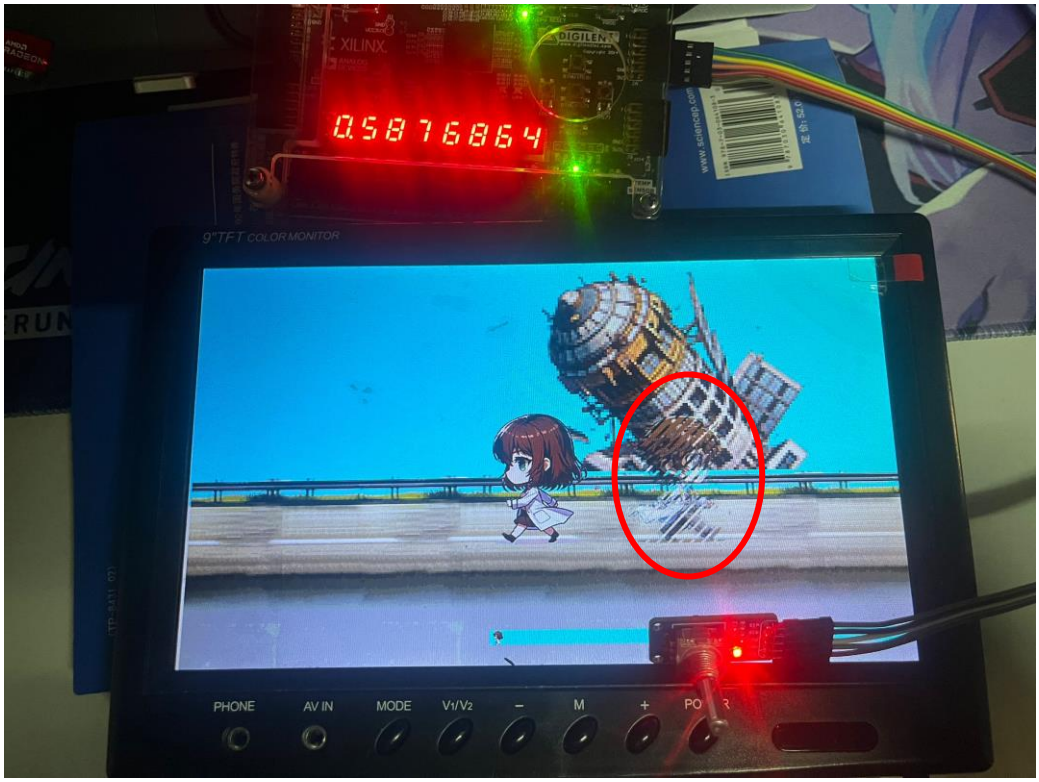
（地面静止，开发板显示 STEN.GATE）



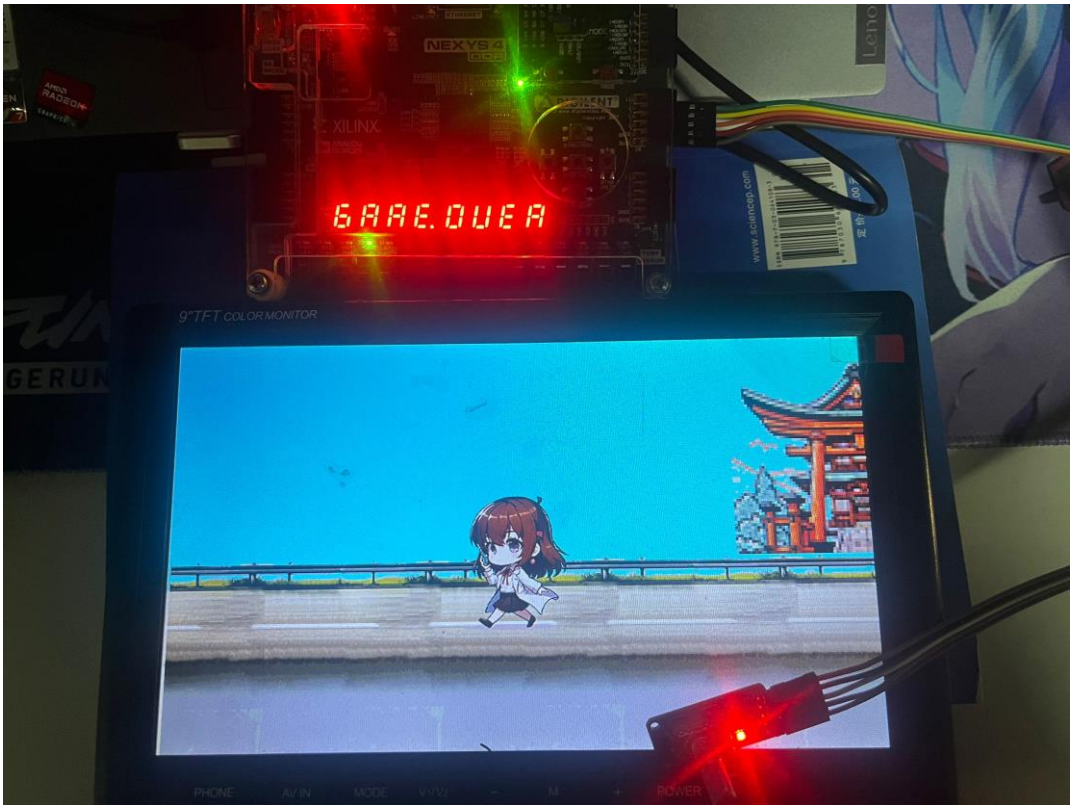
游戏开始，女主视角

世界线 0.983360%

游戏开始，男主视角
世界线 0.0171520%

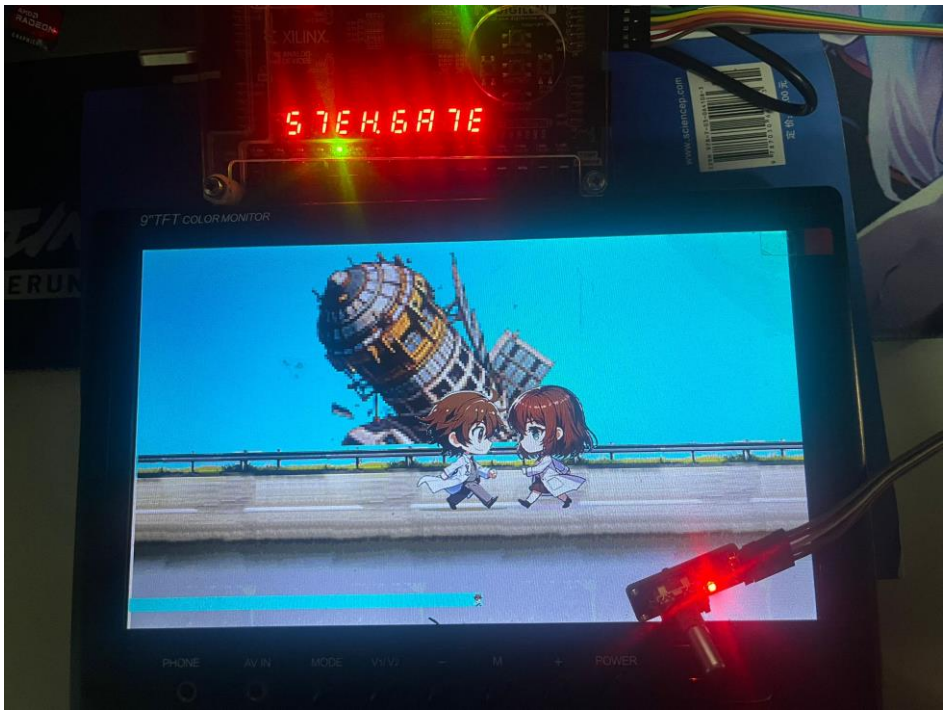


男女主出现在同一时间点
但由于两人不在一条世界线（相差约 0.2%），只能看到彼此的幻影，彼此错过
但还可以通过时间逆行进行补救



游戏结束，女主越界

游戏结束，男主未越界
但此时女主已经越界



见证命运的相遇!!!

七、结论

本实验设计和实现了一个基于 FPGA 的小游戏数字系统。该系统的设计和开发过程涵盖了从控制器状态机的实现、旋转编码器的使用、VGA 传输协议的应用到逻辑设计和模块规划的各个方面。以下是实验的一些结论：

1、时序逻辑及控制器状态机：

本实验的其他模块在没有控制器时就已经可以正常运行（在 Gate_test.v 下测试运行）。游戏逻辑处理器实时对接旋转编码器输入模块、开发板综合显示模块及 VGA 综合显示模块，保证游戏数据的实时性。但由于 VGA 的刷屏特性，如果在 VGA 刷屏的过程中改变显示参数，可能会导致画面模糊错位等问题。于是给 VGA 驱动器专门加了一个传出的场下降沿信号，每次扫描完一屏该模块就会传出一个场下降沿信号，提示游戏逻辑处理模块可以开始更新数据，对下一个画面进行计算。这样可以解决在现实过程中改变现实参数产生的错位问题。同时，游戏逻辑处理模块中也有部分数据是实时计算的，比如速度和另一个角色的相对位置，但是只有当 VGA 模块的场下降沿信号来到时才会用实时数据更新寄存器里面的值。这样子，就算没有控制器，游戏也能在正常的逻辑下运行。

在没有控制器的状态下，只能实现游戏进行时的状态，每次想要重开需要按 rst_n 对系统进行重置，显然这不是一般游戏机的操作逻辑。于是我加了一个只有“等待开始”、“游戏进行”、“游戏结束”三个状态的状态机作为控制器内核。从而实现“点击任意位置进入游戏”以及游戏结束后通过转动操作按钮重开的功能，使之更像一般游戏机。同时，根据之前的时序逻辑，状态机的跳转也需要考虑 VGA 的显示周期，只有当 VGA 扫描完一屏之后才能进行安全的状态跳转，于是该状态机也需要先检测 VGA 场下降沿信号，再判断状态跳转。

另外，本来作为一个游戏一定要有游戏开始界面（比如 vga 测试中的那个 480x300 的概念图）和游戏结束画面的。但是局限于 FPGA 板的存储空间，只能忍痛割爱，这部分将放在空间资源管理部分详细介绍。

2、外围部件使用：

本实验主要使用了两个外围部件：旋转编码器及 VGA 显示屏。

旋转编码器：

防抖检测：在硬件项目中，输入的稳定性至关重要，尤其是对于旋转编码器这类机械式输入设备。防抖检测是通过软件算法实现的，用于消除由于机械和电气噪声导致的不稳定信号。这在旋转编码器中尤为重要，因为它确保了每次旋转或按压的信号都是准确且可靠的。在实现上，通过设置适当的时间延迟，监测输入信号的稳定性，从而有效地过滤掉由于抖动引起的误操作。

检测结果延时：为了确保其他模块稳定地接收到旋转编码器的信号，实现了信号的延时处理。这意味着一旦检测到旋转或按压动作，系统会在将会保持一段时间检测信号。这种设计允许系统有足够的时间来处理信号变化，确保其他依赖于这些输入的模块（如游戏逻辑处理模块）能够接收到正确且一致的信号。该实验中，由于 VGA 的刷屏频率是 60Hz，所以所有计算模块的频率都被调整到 60Hz，为了确保游戏逻辑处理器能够在检测到每一次旋转及按压操作，需要将旋转编码器的检测结果延迟到 1/60s，即大约 17ms 的时间。

VGA 显示屏：

VGA 传输协议：在设计中遵循了 VGA 传输协议，确保了图像数据能够以正确的格式和时序被发送到显示器。VGA 协议涉及行同步和场同步信号的生成，以及按照正确的时序发送像素数据。这些细节的处理对于生成清晰且无闪烁的图像至关重要。

本实验对比了 1024x600 和 640x480 两种分辨率下的 VGA 显示图像，发现 VGA 显示屏会自动调整信号的图像比例投放在屏幕上。由于我用的屏幕比例为 16:9，如果硬调到 4:3 会使两边多出两段蓝屏区域，很不美观，所以我用了更接近 16:9 比例的 1024x600 分辨率的信号。而之所以不用 1024x576（16:9）的原因是在 VGA 的传输协议下 1024x600 分辨率的信号频率正好约为 50MHz（50.561MHz），而我的开发板系统时钟频率是 100MHz，这样正好可以用一个 2 倍分频器来实现始终转换，简单有效。还是高分辨率的屏幕看着舒服，只是 FPGA 板的空间存不下那么大的图片。

驱动器与图层综合器分离：为了提高设计的灵活性和可维护性，VGA 驱动器与图层综合器被设计为独立的模块。这种分离的架构使得对图像内容的修改（如添加、移除或更改图层）不会影响到 VGA 驱动器的实现。同时，这也使得 VGA 驱动器可以重用于不同的项目中，而无需每次都进行大幅度的修改。从设计的角度来看，这种方法提供了更高的模块化，使得项目更易于管理和扩展。

主要是受 PS 影响专门设置了图层整合模块，这样就可以将每个图层的图片信号分开处理，方便之后的修改及阅读，大大提高了编程效率。本实验中共用了五个图层（加上一个原本被删除的图层实际上一共有六个图层），其中为了节省 FPGA 板的储存空间，又将角色图层和进度条图层进一步整合到一个模块中，这部分会在空间资源管理部分详细介绍。

3、游戏逻辑设计：

实际上，在设计这个游戏的过程中，我尝试过了三个数学模型，其中每个模型的内涵和意义都不太一样，但总体逻辑逐渐简化。这种简化不仅带来了硬件上的可操作性，也使游戏本身的操作更加直观。最终确定为这个双主角并行的游戏模型。

在硬件设计中，由于缺乏对浮点数和高级数学函数（如三角函数）的直接支持，需要采取创新的方法来实现这些运算。在本项目中，我通过使用固定点数表示法和近似算法来处理

复杂的数学运算。这种方法使得即使在硬件资源有限的情况下，也能实现需要精确计算的游戏逻辑，如物理运动的模拟和几何变换。

此外，为了防止游戏逻辑的运算对 VGA 显示造成干扰，我设计了并行计算和定时数据更新的功能。并行计算是指在组合逻辑上不同参数在不同线路中同时计算下一个时刻的状态，在时序逻辑上 VGA 刷屏的同时游戏逻辑处理器会同时进行部分较为复杂的计算（比如速度及角色相对位置）。定时数据更新指的是在 VGA 的刷屏间隙中更新需要显示的游戏参数，从而避免了在显示过程中引入图像撕裂或闪烁等问题。这种设计不仅提高了图像显示的质量，也确保了游戏的流畅性和稳定性。此外，这种方法还优化了系统的性能，因为它允许游戏逻辑和 VGA 显示之间有效地共享处理器资源，而不会互相干扰。

4、模块规划：

四大模块的划分：

该项目被划分为四个主要模块：输入（旋转编码器检测）、游戏逻辑处理、开发板显示、VGA 显示。这种划分不仅使系统的结构清晰易懂，也有助于分散和管理复杂性。例如，输入模块专注于处理用户的物理操作并将其转换为数字信号；逻辑处理模块处理游戏规则和决策；开发板显示模块负责本地显示元素，如 LED 和七段显示器；而 VGA 显示模块则处理与外部显示器相关的所有内容。

细分子模块 - 面向对象思想：

在每个主模块内部，进一步细分为更具体的子模块，运用了类似面向对象编程的思想。这种方法不仅提高了代码的可读性和可维护性，而且促进了功能间的封装和独立性。例如，逻辑处理模块可划分为游戏状态管理、分数计算等；显示模块包括七段晶体管显示、LED 灯显示、VGA 驱动器、图层整合等；。这样的细分使得每个子模块都有明确的功能和界限，便于开发和后续的调整。

模块间的协作和独立性：

虽然各模块在功能上是独立的，但它们之间的协作也是系统成功的关键。例如，输入模块的输出直接影响逻辑处理模块的决策，而逻辑处理的结果又决定了显示模块的输出。这种相互依赖和协作关系是通过明确的接口和协议来管理的，确保了数据和信号在系统内部流动的一致性和准确性。

总结：

通过这种模块化和细分的方法，项目不仅易于管理和扩展，而且能够有效地应对复杂性，减少各个部分间的相互干扰。每个模块作为一个独立的单元，可以单独开发和测试，这大大提高了开发效率和系统的可靠性。同时，这也为未来可能的升级或功能扩展奠定了坚实的基础。

5、空间资源管理：

本项目一共调用了 8 个 COE 文件和 IP 核，其中 4 张 200x200、3 张 100x100、1 张 256x141，每个像素点用 1byte 存储。各个图层模块保证每个 IP 核只被实例化一次，但在一个模块中会被重复调用，通过这种方式减少对 FPGA 板存储空间的要求。

其中角色及进度条图层（Chara_layer）的设计体现了空间资源管理。该图层实例化了 4 张 200x200 的角色图片，分别表示男女主的两个状态。该图层实际上融合了当前角色、另一个角色、进度条三个图层任务，其中进度条和当前角色使用 4 张图片里的同一张图片，而另

一个角色会使用另一张照片。所以在部署这个模块时，用了两个地址寄存器分别记录查看两个角色的 ROM 的地址，使当前角色和另一个角色的读取分开，从而实现了并行读取 ROM 的功能，而该模块会根据游戏逻辑处理器传来的信号通过选择器选择对应角色的像素点数据。这样一来，该模块中的三个图层任务（当前角色&另一个角色&进度条）可以在不干扰其他图层的情况下根据需求读取四个 ROM 里的信息。将这三个图层任务整合到一个模块中可以避免 ROM 的重复实例化导致的空间资源消耗。

八、心得体会及建议

1. 游戏设计心得

设计一个游戏一方面需要有巧思，使玩家玩完能有所收获或者感悟；另一方面要兼顾小游戏的直观性和趣味性，使玩家能理解游戏逻辑并能沉浸其中。但我个人的设计理念还是偏向灵感，需要有一个有寓意的故事，或者有深度的理论来支撑整个游戏。这个项目是受到《时空幻境》的时间变幻启发，再进一步结合《命运石之门》的世界观进行的爱的产出。不论结果何如，我享受在其中。

2. 硬件设计心得

- 模块化思维：硬件设计中的模块化思维对我帮助巨大。将复杂的系统拆分为多个模块，不仅使问题变得更易于管理，而且也提高了代码的可重用性。
- 性能与资源的平衡：在有限的硬件资源下优化性能是一项挑战。我学会了权衡不同设计选择的利弊，并尽可能优化每一部分以提高整体性能。
- 细节关注：在硬件设计中，细节至关重要。微小的错误可能导致整个系统的失败。我学会了仔细检查每一个细节，确保所有组件都能正确地协同工作。

3. 建议

- 持续学习与实践：数字系统设计是一个不断发展的领域，持续学习新技术和工具是非常重要的。结合实际项目进行实践，可以有效地提升技能和经验。
- 开放性思维：在面对设计挑战时，保持开放和创新的思维至关重要。不要害怕尝试新方法，即使它们可能与传统做法不同。

九、附录

设计文件代码

0、游戏顶层模块（Gate_top.v）

```
module Gate_top(
    input clk,                  // 时钟信号
    input reset,                // 重置信号，高电平有效
    input [2:0] SIABW,          // 旋转编码器 SIA、SIB、SW 引脚
    output [7:0] anodes,        // 阳极信号（AN0..AN7）
    output [7:0] segments,      // 段信号（A..G）和小数点 DP
    output [15:0] led,          // led 速度显示器
    output vga_hs,              // 行同步信号
    output vga_vs,
```

```
// 场同步信号
    output          [11:0]          vga_rgb
// 红绿蓝输出
    );
  // 控制器使能
  wire RS_ena;
  wire GL_ena;
  wire DP_ena;
  wire [1:0] DP_choice;
  wire VGA_ena;
  // 传递参数
  wire [2:0] operation;  // 操作标记
  wire  [26:0]  offset;      // 世界线偏移量
0-999999
  wire  VS_negedge;          // 开始运算记
号，高电平激发（频率为 60Hz）
  wire [2:0] charac;       // 角色标记
  wire [3:0] speed;        // 速度标记
  wire [9:0] obstacle;    // 障碍物位置
  wire [7:0] obstimes;    // 障碍物大小
  wire [9:0] progress;    // 进度标记
  wire [7:0] offgress;    // 进度标记
  wire [1:0] over_flag;  // 游戏结束标记
  // 系统控制器
  Gate_ctrl ctrl_uut(
      .clk(clk),                  // 系统时钟
      .rst_n(!reset),            // 重置信号
      .op(operation),            // 旋转编码器
操作信号
      .GL_over(over_flag),   // 游戏逻辑模
块的结束信号(win,over)
      .VGA_comp(VS_negedge),// VGA 完
成一屏打印信号
      .RS_en(RS_ena),            // 旋转编码
器模块使能信号
      .GL_en(GL_ena),            // 游戏逻辑
处理模块使能信号
      .DP_en(DP_ena),            // 开发板显
示模块使能信号
      .DP_ch(DP_choice),       // 开发板显
示类型
      .VGA_en(VGA_ena)          // VGA 显
示模块使能信号
  );
  // 旋转编码器输入
  RotationSensor rs_uut(
      .clk(clk),                      // 时钟信
号
      .rst_n(RS_ena),                // 复位信
号
      .SIABW(SIABW),                  // 旋
转编码器 SIA、SIB、SW 引脚
      .operation(operation)       // 顺时针、
逆时针、按压检测
  );
  // 实例化中心逻辑
  meet_logic gate_uut (
      .clk(clk),
      .rst_n(GL_ena),
      .next_s(VS_negedge),
      .operation(operation),

      .character(charac),
      .offset(offset),
      .speed(speed),
      .obstacle(obstacle),
      .obstimes(obstimes),
      .progress(progress),
      .offgress(offgress),

      .game_over(over_flag[0]),
      .game_win(over_flag[1])
    );
  // 辉光管显示 offset 值
  display_seg seg_uut(
       .clk(clk),
       .reset(DP_ena),
       .choice(DP_choice),
       .hex(offset),
       .speed(speed),
       .anodes(anodes),
       .segments(segments),
       .led(led)
   );
   // VGA 显示屏
   vga_top vga_uut(
       .clk_100MHz(clk),              // 标准
时钟
```

```
        .rst_n(VGA_ena),                // 复位信号，低电平有效

        .chara(charac),                 // 角色状态
        .speed(speed),                  // 行进速度
        .obstacle(obstacle),            // 障碍物位置
        .obstimes(obstimes),            // 障碍物大小
        .progress(progress),            // 进度条
        .offgress(offgress),            // 进度条

        .vga_hs(vga_hs),                // 行同步信号
        .vga_vs(vga_vs),                // 场同步信号
        .vga_rgb(vga_rgb),              // 红绿蓝输出
        .VS_negedge(VS_negedge)         // 下降沿信号
    );
endmodule
```

1、控制器模块（GateFSM.v）

```
module Gate_ctrl(
    input clk,                  // 系统时钟
    input rst_n,                // 重置信号
    input [2:0] op,             // 旋转编码器操作信号
    input [1:0] GL_over,// 游戏逻辑模块的结束信号(win,over)
    input VGA_comp,             // VGA 完成一屏打印信号
    output reg RS_en,           // 旋转编码器模块使能信号
    output reg GL_en,           // 游戏逻辑处理模块使能信号
    output reg DP_en,           // 开发板显示模块使能信号
    output reg [1:0] DP_ch, // 开发板显示类型
    output reg VGA_en           // VGA 显示模块使能信号
);
// 状态
reg[1:0] state;
localparam idle = 2'b00, start = 2'b01, over = 2'b10;

// 初始化
initial begin
    // 状态初始化
    state <= idle;
    // 控制端口初始化
    RS_en <= 0;
    GL_en <= 0;
    DP_en <= 0;
    DP_ch <= 0;
    VGA_en <= 0;
end

// 状态机
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= idle;
    end else begin
        case(state)
            // 等待开始状态
            idle:begin
                if (VGA_comp && op) state <= start;
                else state <= idle;
            end
            // 游戏进行状态
            start:begin
                if (VGA_comp && GL_over[0]) state <= over;
                else state <= start;
            end
            // 游戏结束状态
            over:begin
                if (VGA_comp && op[2:1]) state <= idle;
                else state <= over;
```

```
                end
                // 其他状态
                default:state <= idle;
            endcase // end of case
        end // end of rst_n
end // end of always

// 组合逻辑
always @(state or rst_n) begin
        if (!rst_n) begin
            RS_en <= 0;
            GL_en <= 0;
            DP_en <= 0;
            DP_ch <= 0;
            VGA_en <= 0;
        end else begin
             case(state)
                // 等待开始状态
                idle:begin
                    RS_en <= 1;
                    GL_en <= 0;        // 逻辑处理器清零
                    DP_en <= 1;
                    DP_ch  <=  2'b01;  // 显示 STEN_GATE
                    VGA_en <= 1;
                end
                // 游戏进行状态
                start:begin
                    RS_en <= 1;
                    GL_en <= 1;        // 逻辑处理器运行
                    DP_en <= 1;
                    DP_ch  <=  2'b00;  // 显示偏移量及速度
                    VGA_en <= 1;
                end
                // 游戏结束状态
                over:begin
                    RS_en <= 1;
                    GL_en  <=  1;          // 逻辑处理器继续运行
                    DP_en <= 1;
                    DP_ch  <=  {~GL_over[1], GL_over[0]};// 显示结果
                    VGA_en <= 1;
                end
                // 其他状态
                default:begin
                    RS_en <= 0;
                    GL_en <= 0;
                    DP_en <= 0;
                    DP_ch <= 0;
                    VGA_en <= 0;
                end
            endcase // end of case
        end // end of rst_n
end // end of always

endmodule
```

2、旋转编码器模块（RotationSensor.v）

```
module RotaryEncoder (
    input clk,                  // 时钟信号
    input rst_n,                // 复位信号
    input  SIA,                 // 旋转编码器 SIA 引脚
    input  SIB,                 // 旋转编码器 SIB 引脚
    input  SW,                  // 旋转编码器 SW 引脚
    output reg  CW,             // 顺时针旋转检测输出
    output reg  CCW,            // 逆时针旋转检测输出
    output reg Pressed      // 按压动作检测输出
);

parameter HOLD_TIME = 1000000;  // 保持信号的时间
//10ms 计数器，用于消抖。
reg ok_10ms;
reg [31:0]cnt0;
always@(posedge clk,negedge rst_n)
begin
```

```
    if(!rst_n)begin
        cnt0 <= 0;
        ok_10ms <= 1'b0;
    end
    else begin
        if(cnt0 < 32'd49_9999)begin//10ms 消抖 //我的编码器 此值设置为 4999 可用!!!!!!!!!!!!
            cnt0 <= cnt0 + 1'b1;
            ok_10ms <= 1'b0;
        end
        else begin
            cnt0 <= 0;
            ok_10ms <= 1'b1;
        end
    end
end


//同步/消抖 A、B
reg A_reg,A_reg0;
reg B_reg,B_reg0;
wire A_Debounce;
wire B_Debounce;
always@(posedge clk,negedge rst_n)begin
    if(!rst_n)begin
        A_reg <= 1'b1;
        A_reg0 <= 1'b1;
        B_reg <= 1'b1;
        B_reg0 <= 1'b1;
    end
    else begin
        if(ok_10ms)begin
            A_reg <= SIA;
            A_reg0 <= A_reg;
            B_reg <= SIB;
            B_reg0 <= B_reg;
        end
    end
end

assign A_Debounce = A_reg0 && A_reg && SIA;
assign B_Debounce = B_reg0 && B_reg && SIB;


//对消抖后的 A 进行上升沿，下降沿检测。
reg A_Debounce_reg;
wire A_posedge,A_negedge;
always@(posedge clk,negedge rst_n)begin
    if(!rst_n)begin
        A_Debounce_reg <= 1'b1;
    end
    else begin
        A_Debounce_reg <= A_Debounce;
    end
end
assign A_posedge = !A_Debounce_reg && A_Debounce;
assign A_negedge = A_Debounce_reg && !A_Debounce;


//对 AB 相编码器的行为进行描述
reg rotary_right;
reg rotary_left;
always@(posedge clk,negedge rst_n)begin
    if(!rst_n)begin
        rotary_right <= 1'b1;
        rotary_left <= 1'b1;
    end
    else begin
        //A 的上升沿时候如果 B 为低电平，则旋转编码器向右转
        if(A_posedge && !B_Debounce)begin
            rotary_right <= 1'b1;
        end
        //A 上升沿时候如果 B 为高电平，则旋转编码器向左转
        else if(A_posedge && B_Debounce)begin
            rotary_left <= 1'b1;
        end
        //A 的下降沿 B 为高电平，则一次右转结束
        else if(A_negedge &&
```

```
B_Debounce)begin
                rotary_right <= 1'b0;
            end
            //A 的下降沿 B 为低电平，则一次左转结束
            else                    if(A_negedge && !B_Debounce)begin
                rotary_left <= 1'b0;
            end
        end
end


//通过上面的描述，可以发现，
//"rotary_right"为上升沿的时候标志着一次右转
//"rotary_left"  为上升沿的时候标志着一次左转
//以下代码是对其进行上升沿检测
reg rotary_right_reg,rotary_left_reg;
wire rotary_right_pos,rotary_left_pos;
always@(posedge clk,negedge rst_n)begin
    if(!rst_n)begin
        rotary_right_reg <= 1'b1;
        rotary_left_reg <= 1'b1;
    end
    else begin
        rotary_right_reg <= rotary_right;
        rotary_left_reg <= rotary_left;
    end
end

assign rotary_right_pos = !rotary_right_reg && rotary_right;
assign rotary_left_pos = !rotary_left_reg && rotary_left;

// 保持信号
reg [23:0] cw_timer = 0, ccw_timer = 0;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        cw_timer <= 0;
        ccw_timer <= 0;
    end else begin
        if (cw_timer > 0) cw_timer <= cw_timer - 1;
        else if (rotary_right_pos) cw_timer <= HOLD_TIME;

        if (ccw_timer > 0) ccw_timer <= ccw_timer - 1;
        else if (rotary_left_pos) ccw_timer <= HOLD_TIME;
    end
end

// 设置输出
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        CW <= 1'b0;
        CCW <= 1'b0;
        Pressed <= 1'b0;
    end else begin
        CW <= cw_timer != 0;
        CCW <= ccw_timer != 0;
        Pressed <= !SW;
    end
end

endmodule

module RotationSensor (
    input clk,                  // 时钟信号
    input rst_n,                // 复位信号
    input   [2:0] SIABW,                // 旋转编码器 SIA、SIB、SW 引脚
    output [2:0] operation          // 顺时针、逆时针、按压检测
);
//旋转编码器实例化
RotaryEncoder re_uut(
    .clk(clk),
    .rst_n(rst_n),
    .SIA(SIABW[2]),
    .SIB(SIABW[1]),
    .SW(SIABW[0]),
    .CW(operation[2]),
    .CCW(operation[1]),
```

```
    .Pressed(operation[0])
);
```

```
endmodule
```

3、游戏逻辑处理模块（三代模型 meet_gate.v）

```
module meet_logic(
    input  clk,                          // 系统时钟
    input rst_n,                         // 置零，低电平有效
    input next_s,                        // 开始运算记号，高电平激发（频率为 60Hz）
    input  [2:0] operation,              // 操作信号：左转、右转、按钮，高电平有效

    output [2:0] character,              // 当前角色信息及角色状态：另一个角色状态、当前角色状态、当前角色
    output [26:0] offset,                // 世界线偏移量 0-999999
    output [3:0] speed,                  // 速度：15 梯度，0 为静止
    output [9:0] obstacle,               // 另一个角色位置：0 则表示无
    output  [3:0]  obstimes,             // 另一个角色清晰度（0 最清晰）
    output [9:0] progress,               // 当前时间进度 0-1024
    output [8:0] offgress,               // 当前偏移进度

    output  game_over,                   // 游戏结束，高电平有效
    output  game_win                     // 游戏成功（1：成功，0：失败）
    );
parameter _MAX_OFFSET = 3355_4432; // 最大世界线偏移量
parameter _BAS_OFFSET = 1000_0000; // 基础世界线偏移量
parameter _PER_OFFSET = 0000_0512; // 偏移基量
parameter _DIAMETER = 21600;         // 世界直径 6min，15bit
//游戏参数（频率为 60Hz）
parameter _START_P = 1000;       // 初始时间点 16s
parameter _MEET_DIST = 300;      // 相遇时间
parameter _MEET_OFF = 0020_0000;      // 相遇世界线

parameter _BAS_SPEED = 64;       // 速度单位量
parameter _BAS_TIME = 64;        // 时间流逝基准值
parameter _SPEED_LEVEL = 16; // 速度有 16 个等级
parameter _BACK_TIMES = 3;       // 世界线收束速率（未使用）

parameter _SCREEN_WIDTH = 1024; // 屏幕宽度
parameter _SCREEN_TIME = 2048;   // 一屏代表的时间
parameter _SCREEN_OFFSET = 0050_0000; // 相遇时间线差基值
parameter _CHARA_XPOS = 512; // 角色水平位置
parameter _CHARA_WIDTH = 200; // 角色水平宽度
//状态标记量
reg over_flag;          // 游戏结束标记
reg win_flag;           // 游戏胜利标记
//游戏变量
reg CurChara;                    // 当前角色
reg Chstate[1:0];                // 角色状态
reg [21:0] Time[1:0];          // 时间 t（最大值 32768 * 64）
reg [26:0] Offset[1:0];       // 角色偏移量 s
reg  [9:0]  Speed[1:0];          // 角色行进速度 v（基准级：64）
reg [9:0] Obstacle;              // 另一个人的位置
reg  [3:0]  Obstimes;            // 另一个角色
```

```
清晰度（0 最清晰）
// 反应计时器
parameter _HOLD_TIME = 20;//检测间隔 0.3s
reg [6:0] counter;
// 初始化
initial begin
    CurChara <= 0;
    Chstate[0] <= 0;
    Chstate[1] <= 0;
    Time[0] <= _START_P * _BAS_TIME;
    Time[1] <= (_DIAMETER - _START_P) * _BAS_TIME;
    Offset[0] <= 0;
    Offset[1] <= _BAS_OFFSET;
    Speed[0] <= 0;//_BAS_SPEED;
    Speed[1] <= 0;//_BAS_SPEED;
    Obstacle <= 0;
    Obstimes <= 0;
    win_flag <= 0;
    over_flag <= 0;
    counter <= 0;
end

//更新另一个角色相对位置
wire [9:0] realObsta;
wire [3:0] realObstm;
anotherChara #(_BAS_TIME, _SCREEN_WIDTH, _SCREEN_TIME, _SCREEN_OFFSET, _CHARA_XPOS, _CHARA_WIDTH)
        anch_uut(Time[CurChara], Time[~CurChara], Offset[CurChara], Offset[~CurChara], realObsta, realObstm);
// 切换人物操作（独立判断）
always @(posedge next_s or negedge rst_n) begin
    if (!rst_n) begin // 初始化
        CurChara <= 0;
        counter <= 0;
        Obstacle <= 0;
        Obstimes <= 0;
    end // end of rst_n
    else begin
        if (next_s) begin
            // 更新另一角色的显示
            Obstacle <= realObsta;
            Obstimes <= realObstm;
            // 切换人物
            if (counter != 0) begin
                counter <= counter - 1;
            end else begin
                if (operation[0]) begin
                    CurChara <= !CurChara;
                    counter <= _HOLD_TIME;
                end // end of operation
            end // end of counter
        end // end of next_s
    end // end of rst_n
end // end of always

//更新角色速度
wire [9:0] realSpeed[1:0];
calSpeed #(_MAX_OFFSET, 0, _BAS_SPEED, _SPEED_LEVEL) chara_speed(Offset[0], realSpeed[0]);
calSpeed #(_MAX_OFFSET, _BAS_OFFSET, _BAS_SPEED, _SPEED_LEVEL) chris_speed(Offset[1], realSpeed[1]);
// 单次运算
always @(posedge next_s or negedge rst_n) begin
    if (!rst_n) begin // 初始化
        Chstate[0] <= 0;
        Chstate[1] <= 0;
        Time[0] <= _START_P * _BAS_TIME;
        Time[1] <= (_DIAMETER - _START_P) * _BAS_TIME;
        Offset[0] <= 0;
        Offset[1] <= _BAS_OFFSET;
        Speed[0] <= 0;//_BAS_SPEED;
        Speed[1] <= 0;//_BAS_SPEED;
        win_flag <= 0;
        over_flag <= 0;
    end // end of rst_n
```

```
    else begin
        if (next_s && !over_flag) begin // 计算操作
            // 处理操作
            casex(operation[2:1])
                // 向右走
                2'bx1: begin
                    if (!CurChara) Chstate[0] <= 1'b0;  //伦太郎正行
                    else Chstate[1] <= 1'b1;            //红莉栖倒行
                end
                // 向左走
                2'b10: begin
                    if (!CurChara) Chstate[0] <= 1'b1;  //伦太郎倒行
                    else Chstate[1] <= 1'b0;            //红莉栖正行
                end
            endcase

            // 同步计算双主角参数
            // 更新时间
            if (Chstate[0]) begin // 伦太郎
                if (Time[0] > Speed[0]) Time[0] <= Time[0] - Speed[0];
                else Time[0] <= _DIAMETER * _BAS_TIME + Time[0] - Speed[0];
            end else begin
                if (Time[0] < _DIAMETER * _BAS_TIME) Time[0] <= Time[0] + Speed[0];
                else Time[0] <= Speed[0];
            end
            if (!Chstate[1]) begin // 红莉栖
                if (Time[1] > Speed[1]) Time[1] <= Time[1] - Speed[1];
                else Time[1] <= _DIAMETER * _BAS_TIME;
            end else begin
                if (Time[1] < _DIAMETER * _BAS_TIME) Time[1] <= Time[1] + Speed[1];
                else Time[1] <= Speed[1];
            end

            // 更新偏移量
            if (Chstate[0]) begin // 伦太郎
                if (Offset[0] < _MAX_OFFSET) Offset[0] <= Offset[0] + _PER_OFFSET * Speed[0] / _BAS_SPEED;
                else Offset[0] <= 0;
            end else begin
                Offset[0] <= Offset[0]; // 保持不变
            end
            if (Chstate[1]) begin // 红莉栖
                if (Offset[1] > _PER_OFFSET) Offset[1] <= Offset[1] - _PER_OFFSET * Speed[1] / _BAS_SPEED;
                else Offset[1] <= _MAX_OFFSET;
            end else begin
                Offset[1] <= Offset[1]; // 保持不变
            end

            // 更新角色速度
            Speed[0] <= realSpeed[0];
            Speed[1] <= realSpeed[1];

            // 判断游戏结束
            // 两人相遇，游戏胜利
            if (Time[0] <= Time[1] && Time[1] - Time[0] <= _MEET_DIST * _BAS_TIME &&
                (Offset[0] >= Offset[1] ? Offset[0] - Offset[1] : Offset[1] - Offset[0]) <= _MEET_OFF)
                begin Chstate[0] <= 0; Chstate[1] <= 0;  //角色状态置零
                    Speed[0] <= 0; Speed[1] <= 0;        //角色速度置零
                    win_flag <= 1; over_flag <= 1; end //胜利标记
```

```
                // 彼此错过，游戏失败
                if (Time[0] <= Speed[0] || Time[1] >= _DIAMETER * _BAS_TIME || Offset[1] > _BAS_OFFSET * _BAS_TIME || Offset[2] > _BAS_OFFSET * _BAS_TIME)
                    begin Chstate[0] = 1; Chstate[1] = 1;       //角色状态置一
                        Speed[0] <= 0; Speed[1] <= 0;           //角色速度置零
                        win_flag <= 0; over_flag <= 1; end    //失败标记
            end // end of next_s
        end // end of ena
end // end of always

//输出赋值
assign character = {Chstate[~CurChara], Chstate[CurChara], CurChara};          // 角色信息
assign offset = Offset[CurChara];  // 世界线偏移量 0-99999999
assign speed = Speed[CurChara] / _BAS_SPEED;                // 速度：16 梯度 0-15
assign obstacle = Obstacle;  // 障碍物位置
assign obstimes = Obstimes;  // 障碍物大小
assign progress = Time[CurChara] / _BAS_TIME * _SCREEN_WIDTH / _DIAMETER;    //游戏进度
assign offgress = (Offset[CurChara] >> 12) * 60 / (_BAS_OFFSET >> 12);        //偏移进度
assign game_over = over_flag;                // 游戏结束，高电平有效
assign game_win = win_flag;  // 游戏成功，高电平有效

endmodule

module calSpeed(
    input[26:0] offset,
    output[9:0] speed
    );
parameter _MAX_OFFSET = 3355_4432; // 最大世界线偏移量
parameter _BAS_OFFSET = 1000_0000; // 基础世界线偏移量
parameter _BAS_SPEED = 64;    // 速度单位量
parameter _SPEED_LEVEL = 16; // 速度有 16 个等级

wire [26:0] off = (offset > _BAS_OFFSET) ? (offset - _BAS_OFFSET) % (_MAX_OFFSET >> 1) : (_BAS_OFFSET - offset) % (_MAX_OFFSET >> 1);
wire [5:0] ori_level = (off >> 8) * 2 * _SPEED_LEVEL / (_MAX_OFFSET >> 9); // 32 个层级
assign speed = _BAS_SPEED * ((ori_level < _SPEED_LEVEL / 2) ? (1 + ori_level) :
                                (ori_level < _SPEED_LEVEL) ? (5 + ori_level / 2) :
                                (7 + ori_level / 4));

endmodule

module anotherChara(
    input [21:0] thisTime,
    input [21:0] thatTime,
    input [26:0] thisOffset,
    input [26:0] thatOffset,
    output [9:0] obsta,
    output [3:0] obstm
    );
parameter _BAS_TIME = 64;     // 时间流逝基准值
parameter _SCREEN_WIDTH = 1024; // 屏幕宽度
parameter _SCREEN_TIME = 2048;  // 一屏代表的时间
parameter _SCREEN_OFFSET = 0050_0000; // 相遇时间线差基值
parameter _CHARA_XPOS = 512; // 角色水
```

```
平位置
parameter _CHARA_WIDTH = 200; // 角色水平宽度

wire [15:0] thisT = thisTime[21:6];
wire [15:0] thatT = thatTime[21:6];
wire exist = (thisT >= thatT ? thisT - thatT < _SCREEN_TIME >> 1 : thatT - thisT < _SCREEN_TIME >> 1);
wire [5:0] times = (thisOffset >= thatOffset ? (thisOffset - thatOffset) >> 5 : (thatOffset - thisOffset) >> 5) / (_SCREEN_OFFSET >> 5);
assign obsta = !exist ? 10'b0 :
                    thisT >= thatT ? _CHARA_XPOS - (thisT - thatT) * _SCREEN_WIDTH / _SCREEN_TIME :

_CHARA_XPOS + (thatT - thisT) * _SCREEN_WIDTH / _SCREEN_TIME;
assign obstm = times > 15 ? 4'b0 : 15 - times;
endmodule
```

4、开发板综合显示模块（display_seg.v）

```
module display_seg(
    input clk,                  // 时钟信号
    input reset,                // 重置信号，低电平有效
    input [1:0] choice,         // 显示模式：00:只显示数字;10:显示数字和速度;01:游戏开始;11:游戏结束;
    input [26:0] hex,           // 要显示的27bit 十六进制数字
    input [3:0] speed,          // 要显示的速度值
    output [7:0] anodes,        // 阳极信号（AN0..AN7）
    output [7:0] segments, // 段信号（A..G）和小数点 DP
    output [15:0] led           // led 速度显示器
    );
    wire [31:0]bcd; //中转 bcd 码
    // 实例化 bcd 码转换
    hex_to_bcd uut1(
        .hex(hex),
        .bcd(bcd)
        );
    // 实例化七段显示模块
    seven_segment_display uut0 (
        .clk(clk),
        .reset(reset),
        .choice(choice),
        .bcdnum(bcd),
        .anodes(anodes),
        .segments(segments)
        );
    // 实例化速度 led 显示器：只有当choice=10 时才会显示
    speed_led_display uut2(
        .clk(clk),
        .ena(reset && choice != 2'b00),
        .flicker(choice[0]),
        .speed(speed),
        .led(led)
        );
endmodule
```

4-1、十六进制转 BCD 码转码器（hex_to_bcd.v）

```
module hex_to_bcd(
    input [26:0] hex,   // 27-bit 十六进制数
    output reg [31:0] bcd // 32-bit BCD 码（8位，每位 4-bit）
);

integer i, j;

always @(hex) begin
    // 初始化 BCD 为 0
    bcd = 0;

    // 对每一个输入位进行处理
    for (i = 26; i >= 0; i = i - 1) begin
```

```
        // 检查每个 BCD 数字是否大于 4
        for (j = 0; j < 8; j = j + 1) begin
            if (bcd[4*j +: 4] > 4)
                bcd[4*j +: 4] = bcd[4*j +: 4] + 3;
        end

        // 左移 BCD 码
        bcd = bcd << 1;

        // 将十六进制的最高位添加到 BCD 的最低位
        bcd[0] = hex[i];
    end
end

endmodule
```

4-2、七段晶体管显示模块（seven_segment_display.v）

```
module seven_segment_display (
    input clk,                  // 时钟信号
    input reset,                // 重置信号，低电平有效
    input [1:0] choice,     // 选择显示内容
    input [31:0] bcdnum,    // 要显示的 8 位数字（每个数字 4 位）
    output reg [7:0] anodes,    // 阳极信号（AN0..AN7）
    output reg [7:0] segments   // 段信号（A..G）和小数点 DP
);

// 分频计数器，用于创建显示刷新率
reg [15:0] refresh_counter;
always @(posedge clk or negedge reset) begin
    if (!reset)
        refresh_counter <= 0;
    else
        refresh_counter <= refresh_counter + 1;
end

// 当前活跃的显示位
wire [2:0] active_digit = refresh_counter[15:13]; // 更改这个以调整刷新率

// 七段解码器
function [7:0] decode;
    input [3:0] digit;
    input [2:0] digit_index;
    begin
        case (digit)
            4'h0: decode[7:1] = 7'b0000001; // 0
            4'h1: decode[7:1] = 7'b1001111; // 1
            4'h2: decode[7:1] = 7'b0010010; // 2
            4'h3: decode[7:1] = 7'b0000110; // 3
            4'h4: decode[7:1] = 7'b1001100; // 4
            4'h5: decode[7:1] = 7'b0100100; // 5
            4'h6: decode[7:1] = 7'b0100000; // 6
            4'h7: decode[7:1] = 7'b0001111; // 7
            4'h8: decode[7:1] = 7'b0000000; // 8
            4'h9: decode[7:1] = 7'b0000100; // 9
            default: decode = 7'b1111111; // 空白
        endcase
        // 最高位小数点永远点亮
        decode = {decode[7:1], digit_index != 3'b111};
        //if (digit_index == 3'b110) decode = 8'b1111_1110;
    end
endfunction

// 七段解码器（STEN_GATE）
```

```
function [7:0] STEN_GATE;
    input [2:0] digit_index;
    begin
        case (digit_index)
            4'h7:  STEN_GATE[7:1]  = ~7'b1011011; // S
            4'h6:  STEN_GATE[7:1]  = ~7'b1110000; // T
            4'h5:  STEN_GATE[7:1]  = ~7'b1001111; // E
            4'h4:  STEN_GATE[7:1]  = ~7'b0110111; // N
            4'h3:  STEN_GATE[7:1]  = ~7'b1011111; // G
            4'h2:  STEN_GATE[7:1]  = ~7'b1110111; // A
            4'h1:  STEN_GATE[7:1]  = ~7'b1110000; // T
            4'h0:  STEN_GATE[7:1]  = ~7'b1001111; // E
            default:  STEN_GATE  = 7'b1111111; //  空白
        endcase
        //  中间的小数点永远点亮
        STEN_GATE = {STEN_GATE[7:1], digit_index != 3'b100};
    end
endfunction

//  七段解码器（GAME_OVER）
function [7:0] GAME_OVER;
    input [2:0] digit_index;
    begin
        case (digit_index)
            4'h7:  GAME_OVER[7:1]  = ~7'b1011111; // G
            4'h6:  GAME_OVER[7:1]  = ~7'b1110111; // A
            4'h5:  GAME_OVER[7:1]  = ~7'b1110111; // M
            4'h4:  GAME_OVER[7:1]  = ~7'b1001111; // E
            4'h3:  GAME_OVER[7:1]  = ~7'b1111110; // O
            4'h2:  GAME_OVER[7:1]  = ~7'b0111110; // V
            4'h1:  GAME_OVER[7:1]  = ~7'b1001111; // E
            4'h0:  GAME_OVER[7:1]  = ~7'b1110111; // R
            default:  GAME_OVER  = 7'b1111111; //  空白
        endcase
        //  中间的小数点永远点亮
        GAME_OVER  = {GAME_OVER[7:1], digit_index != 3'b100};
    end
endfunction

//  更新显示内容
always @(posedge clk or negedge reset) begin
    if (!reset) begin
        anodes <= 8'b11111111;
        segments <= 8'b11111111;
    end else begin
        anodes <= ~(1'b1 << active_digit); //  选择当前活跃的数字
        casex(choice)
            2'bx0:  segments  <= decode(bcdnum[4*active_digit  +:  4], active_digit); //  解码当前数字并处理小数点
            2'b01:  segments  <= STEN_GATE(active_digit);
            2'b11:  segments  <= GAME_OVER(active_digit);
        endcase
    end
end

endmodule
```

4-3、LED 灯显示模块（speed_led_display.v）

```
module speed_led_display(
    input clk,          //时钟信号
    input ena,          //使能信号
    input flicker,      //闪烁效果
```

```
    input [3:0] speed,          //速度
    output reg [15:0] led    //led 显示灯
    );

// 时钟分频
wire clk_5Hz;
divider_5M divider_uut(clk, ena, clk_5Hz);
//闪烁效果
reg [3:0] lednum = 4'b1111;
always @(posedge clk_5Hz or negedge ena)
begin
    if (!ena || !flicker)
        lednum = 4'b1111;
    else if (lednum == 4'b0000)
        lednum = 4'b1111;
    else
        lednum = lednum - 1;
end
// 用 led 显示当前速度 or 闪烁
genvar i;
generate
    for (i = 0; i < 16; i = i + 1) begin :
GEN_REGS
        always @(posedge clk_5Hz) begin
            if (!ena) begin
                led[i] <= 1'b0;
            end else begin
                if (!flicker) begin
                    if (i >= 15 - speed)
led[i] <= 1'b1;
                    else
led[i] <= 1'b0;
                end else begin
                    if (i == lednum)
led[i] <= 1'b1;
                    else
led[i] <= 1'b0;
                end
            end
        end
    end
endgenerate

endmodule
```

5、VGA 综合显示模块（vga_top.v）

```
module vga_top(
    input     clk_100MHz,       // 标准时钟
    input     rst_n,            // 复位信号，
高电平有效（方便测试）

    input [2:0]       chara,       // 角色状态
    input [3:0]       speed,       // 行进速度
    input [9:0]       obstacle, // 障碍物位置
    input [3:0]       obstimes,// 障碍物大小
    input [9:0]       progress,    // 时间进度
    input [8:0]       offgress,    // 偏移进度

    output            vga_hs,    // 行同步信号
    output            vga_vs,    // 场同步信号
    output   [11:0] vga_rgb, // 红绿蓝输出
    output        VS_negedge // 下降沿信号
    );
wire clk_25MHz;
wire disp_en;
wire [11:0] pix_data;
wire [11:0] pix_xpos;
wire [11:0] pix_ypos;
//时钟分频
divider_2       d2(clk_100MHz,       rst_n,
clk_50MHz);
//图层综合
VGA_game_layers_syn clr_syn_inst(
    .vga_clk(clk_50MHz), // VGA 驱动时钟
    .disp_en(disp_en),      // 显示有效信号

    .chara(chara),          // 选择角色
    .speed(speed),          // 前进速度
    .obstacle(obstacle),    // 障碍物位置
    .obstimes(obstimes),    // 障碍物大小
    .progress(progress),    // 时间进度
    .offgress(offgress),    // 偏移进度

    .pix_xpos(pix_xpos),    // 像素点横坐标
    .pix_ypos(pix_ypos),    // 像素点纵坐标
    .pix_data(pix_data)     // 像素点数据
```

```
    );
//VGA 驱动器
vga_driver_1024x600 vga_driver_inst(
    .clk_50MHz(clk_50MHz),// VGA 驱动时钟
    .rst_n(rst_n),              // 复位信号，低电平有效

    .disp_en(disp_en),      //显示有效信号
    .VS_negedge(VS_negedge),
//输出场信号下降沿

    .vga_hs(vga_hs),        // 行同步信号
    .vga_vs(vga_vs),        // 场同步信号
    .vga_rgb(vga_rgb),      // 红绿蓝输出

    .pix_data(pix_data),    // 像素点数据
    .pix_xpos(pix_xpos),    // 像素点横坐标
    .pix_ypos(pix_ypos)     // 像素点纵坐标
    );
endmodule
```

5-1、VGA 驱动器模块（vga_driver_1024x600.v）

```
module vga_driver_1024x600(
    input           clk_50MHz,      // VGA 驱动时钟
    input           rst_n,          // 复位信号

    output          vga_hs,         // 行同步信号
    output          vga_vs,         // 场同步信号
    output  [11:0]  vga_rgb,        // 红绿蓝输出

    input   [11:0]  pix_data,       // 像素点数据
    output  [11:0]  pix_xpos,       // 像素点横坐标
    output  [11:0]  pix_ypos,       // 像素点纵坐标

    output disp_en,                 //显示有效信号
    output  VS_negedge              //输出场信号下降沿
    );

/************************************************
参数、寄存器、线网定义(分辨率: 1024*600 时钟频率: 50.561mhz 位数：12 位 )
************************************************/
// 以下时序参数需根据您的显示器进行调整
parameter H_SYNC  = 136;    // 行同步
parameter H_BACK  = 160;    // 行显示后沿
parameter H_DISP  = 1024;   // 行有效数据
parameter H_FRONT = 24;     // 行显示前沿
parameter H_TOTAL = 1344;   // 行扫描周期

parameter V_SYNC  = 3;      // 场同步
parameter V_BACK  = 23;     // 场显示后沿
parameter V_DISP  = 600;    // 场有效数据
parameter V_FRONT = 1;      // 场显示前沿
parameter V_TOTAL = 627;    // 场扫描周期

reg [11:0] cnt_h;           // 行时序计数器
reg [11:0] cnt_v;           // 场时序计数器

reg  VS_reg1, VS_reg2;      //场信号由
```

1->0 时，即下降沿时才进行状态转移。每一帧的下降沿，图片看起来是一帧一帧的运动，进来连续运动形成动画

```
/*************************************
************
VGA 时钟：50M
*************************************
************/
wire vga_clk;
assign vga_clk = clk_50MHz;

/*************************************
************
VGA 显示
*************************************
************/
// VGA 请求数据标志
wire pix_data_req_flag;
assign pix_data_req_flag = ((cnt_h >=
(H_SYNC + H_BACK )) &&
                           (cnt_h <=
(H_SYNC + H_BACK + H_DISP )) &&
                           (cnt_v >=
(V_SYNC + V_BACK )) &&
                           (cnt_v <=
(V_SYNC + V_BACK + V_DISP )))
                           ? 1'b1 :
1'b0;

// 输出的像素点坐标
assign pix_xpos = pix_data_req_flag ? (cnt_h
- (H_SYNC + H_BACK) + 1'b1) : 12'd0;
assign pix_ypos = pix_data_req_flag ? (cnt_v
- (V_SYNC + V_BACK) + 1'b1) : 12'd0;

// VGA 行场同步信号
assign vga_hs  = (cnt_h < H_SYNC ) ? 1'b0 :
1'b1;
assign vga_vs  = (cnt_v < V_SYNC ) ? 1'b0 :
1'b1;

// 行计数器计数
always @(posedge vga_clk or negedge rst_n)
begin
    if (!rst_n)
        cnt_h <= 12'd0;
    else begin
        if(cnt_h < H_TOTAL - 1'b1)
            cnt_h <= cnt_h + 1'b1;
        else
            cnt_h <= 12'd0;
    end
end

// 场计数计数
always @(posedge vga_clk or negedge rst_n)
begin
    if(!rst_n)
        cnt_v <=  12'd0  ;
    else if(cnt_v == V_TOTAL - 1'b1)
        cnt_v <=  12'd0  ;
    else if(cnt_h == H_TOTAL - 1'b1)
        cnt_v <=  cnt_v + 1'b1  ;
    else
        cnt_v <=  cnt_v ;
end

//计算是否为场时序下降沿
always @ (posedge vga_clk or negedge rst_n)
begin
    if(!rst_n) begin
        VS_reg1 <= 0;
        VS_reg2 <= 0;
    end else begin
        VS_reg1 <= vga_vs;
        VS_reg2  <=  VS_reg1;      //非
阻塞赋值，此刻 reg1 的值是当前 clk 上升沿
的 VS，reg2 为上一 clk 上升沿的 VS
    end
end

 assign VS_negedge = ~VS_reg1 & VS_reg2;
//优先级~高于&，VS 由 1 变为 0 时，最终
值取 1，作为图片移动状态机转移的有效信
号（来自网上）

// RGB 显示的使能信号
```

```
assign disp_en    = pix_data_req_flag;

// VGA 显示像素数据
assign vga_rgb = disp_en ? pix_data : 12'd0;

endmodule
```

5-2、VGA 游戏图层综合模块（VGA_game_layers_syn.v）

```
module VGA_game_layers_syn(
    input    vga_clk,                    // VGA 驱动时钟
    input    disp_en,                    // 显示有效信号

    input    [3:0]  chara,               // 当前角色
    input    [3:0]  speed,               // 前进速度
    input    [9:0]  obstacle,            // 另一个角色位置
    input    [3:0] obstimes,             // 另一个角色清晰度
    input    [9:0]  progress,            // 时间进度
    input    [8:0] offgress,             // 偏移进度

    input    [11:0]    pix_xpos,         // 像素点横坐标
    input    [11:0]    pix_ypos,         // 像素点纵坐标
    output reg    [11:0]    pix_data     // 像素点数据
    );
parameter    H_DISP    =    1024;    // 水平像素点数
parameter    V_DISP    =    600;     // 竖直像素点数
wire [4:0] vaild_layer;               // 判断各图层是否有效
wire [11:0] pix_data_layer [4:0]; // 储存各图层像素点信息

always @(posedge vga_clk) begin
    if (disp_en &&
        pix_xpos >= 0   && pix_xpos < H_DISP &&
        pix_ypos >= 0   && pix_ypos < V_DISP) begin
    // 按图层优先级显示图片
        casex(vaild_layer)
            5'b00001: begin pix_data <= pix_data_layer[0]; end
            5'b0001x: begin pix_data <= pix_data_layer[1]; end
            5'b001xx: begin pix_data <= pix_data_layer[2]; end
            5'b01xxx: begin pix_data <= pix_data_layer[3]; end
            //5'b1xxxx: begin pix_data <= pix_data_layer[4]; end
            default: begin pix_data <= 12'hfff; end//默认白色
        endcase
    end else begin
        pix_data <= 12'h000;
    end
end

// 实例化图层
BG_layer bg_inst(
    .vga_clk(vga_clk),                // VGA 驱动时钟
    .disp_en(disp_en),                // 显示有效信号
    .pix_xpos(pix_xpos),              // 像素点横坐标
    .pix_ypos(pix_ypos),              // 像素点纵坐标
    .valid({vaild_layer[0]}),         // 该点是否有像素
    .pix_data(pix_data_layer[0])      // 像素点数据
    );
Satellbg_layer satellbg_inst(
    .vga_clk(vga_clk),                // VGA 驱
```

```
动时钟
        .disp_en(disp_en),                  // 显示有效信号
        .progress(progress),                // 进度
        .pix_xpos(pix_xpos),                // 像素点横坐标
        .pix_ypos(pix_ypos),                // 像素点纵坐标
        .valid({vaild_layer[1]}),           // 该点是否有像素
        .pix_data(pix_data_layer[1])        // 像素点数据
        );
Road_layer road_inst(
        .vga_clk(vga_clk),                  // VGA驱动时钟
        .disp_en(disp_en),                  // 显示有效信号
        .forward(!(chara[0]^chara[1])),     // 控制行进方向
        .speed(speed),                      // 路面行进速度
        .pix_xpos(pix_xpos),                // 像素点横坐标
        .pix_ypos(pix_ypos),                // 像素点纵坐标
        .valid({vaild_layer[2]}),           // 该点是否有像素
        .pix_data(pix_data_layer[2])        // 像素点数据
        );
Chara_layer chara_inst(
        .vga_clk(vga_clk),                  // VGA驱动时钟
        .disp_en(disp_en),                  // 显示有效信号
        .chara(chara),                      // 选择当前角色
        .obstacle(obstacle),                // 另一个角色位置
        .obstimes(obstimes),                // 另一个角色清晰度
        .progress(progress),                // 时间进度
        .offgress(offgress),                // 偏移进度
        .pix_xpos(pix_xpos),                // 像素点横坐标
        .pix_ypos(pix_ypos),                // 像素点纵坐标
        .valid({vaild_layer[3]}),           // 该点是否有像素
        .pix_data(pix_data_layer[3])        // 像素点数据
        );

endmodule
```

5-2-0、各级图层模块（Game_Layers.v）

5-2-1、背景图层模块（BG_layer）

```
module BG_layer(
    input   vga_clk,                    // VGA 驱动时钟
    input   disp_en,                    // 显示有效信号
    input   [11:0]  pix_xpos,           // 像素点横坐标
    input   [11:0]  pix_ypos,           // 像素点纵坐标
    output  reg   valid,                // 该点是否有像素
    output  reg   [11:0]  pix_data      // 像素点数据
    );
    parameter   H_DISP    =   1024;     // 水平像素点数
    parameter   V_DISP    =   600;      // 竖直像素点数

    always @(posedge vga_clk) begin
        if (disp_en) begin
            if(pix_xpos   >=   0    &&
```

```
pix_xpos < H_DISP && pix_ypos >= 0    &&
pix_ypos < V_DISP)
                        begin
                            valid <= 1'b1;//该
图层有效数据
                            pix_data        <=
12'h0ff;
                        end
                    else begin
                            valid  <=  1'b0;//该图
层透明像素点
                            pix_data <= 12'hfff;
                        end
                end else begin
                        valid <= 1'b0;
                        pix_data <= 12'h000;
                end
        end

endmodule
```

5-2-2、背景建筑物图层模块（Satellbg_layer）

```
module Satellbg_layer(
    input    vga_clk,                    //
VGA 驱动时钟
    input    disp_en,                    //
显示有效信号
    input    [9:0]  progress,            // 进
度条
    input    [11:0]    pix_xpos,         // 像
素点横坐标
    input    [11:0]    pix_ypos,         // 像
素点纵坐标
    output reg    valid,                 // 该
点是否有像素
    output reg    [11:0]    pix_data     // 像
素点数据
    );
    parameter    H_DISP    =    1024;    //
水平像素点数
    parameter    V_DISP    =    600;     //
竖直像素点数
    parameter    IMG_WIDTH = 100;
    parameter    IMG_HEIGHT = 100;
    parameter    BASEY = 400;    // 纵坐标
    parameter    START_PRO = 341; // 三分
之一点
    parameter    END_PRO  =  682;    // 三
分之二点
    parameter    TIMES = 4;    // 放大 4 倍
    reg [12:0] img_addr;
    wire [15:0] img_data;
    wire    [1:0]    choice    =    progress    /
START_PRO;
    wire    [9:0]    basex    =    (START_PRO    -
progress % START_PRO) * 3;
    always @(posedge vga_clk) begin
        if (disp_en) begin
            img_addr <= 12'b0; //文件指
针归零
            if(basex    &&    pix_xpos    >=
basex - IMG_WIDTH / 2 * TIMES    &&
pix_xpos < basex + IMG_WIDTH / 2 *
TIMES    &&
                pix_ypos >= BASEY -
IMG_HEIGHT * TIMES && pix_ypos <
BASEY)
                    begin
                        //building2 ： fff
为默认透明的区域
                        //satellite：0ff 为
默认透明的区域
                        //building1 ： 0f0
为默认透明的区域
                        if    (choice    ==
2'b00    &&    img_data[11:8]    >=    4'hd    &&
img_data[7:4] >= 4'hd && img_data[3:0] >=
4'hd ||
                            choice    ==
2'b01    &&    img_data[11:8]    <=    4'h2    &&
img_data[7:4] >= 4'hd && img_data[3:0] >=
4'hd ||
                            choice    ==
2'b10    &&    img_data[11:8]    <=    4'h2    &&
img_data[7:4] >= 4'hd && img_data[3:0] <=
4'h2 ||
```

```
                choice == 2'b11)
                valid <= 1'b0;//该图层透明像素点
            else
                valid <= 1'b1;//该图层有效数据
            pix_data <= img_data[11:0];
            img_addr <= (IMG_HEIGHT - (BASEY - pix_ypos) / TIMES) * IMG_WIDTH +

(pix_xpos + IMG_WIDTH / 2 * TIMES - basex) / TIMES;//比色素早一步
                end
            else begin
                valid <= 1'b0;//该图层透明像素点
                pix_data <= 12'hfff;
            end
        end else begin
            valid <= 1'b0;
            pix_data <= 12'h000;
        end
    end

    wire [15:0] img_sate;
    wire [15:0] img_build1;
    wire [15:0] img_build2;

    assign img_data = choice == 2'b00 ? img_build2 :
                      choice == 2'b01 ? img_sate :
                      choice == 2'b10 ? img_build1 :
                      12'hfff;
    satellite1_rom ROM_inst (
        .clka(vga_clk),
        .addra(img_addr),
        .douta(img_sate),
        .ena(disp_en) // 连接 ROM 输出数据到您的模块中
        );
    building1_rom ROM1_inst (
        .clka(vga_clk),
        .addra(img_addr),
        .douta(img_build1),
        .ena(disp_en) // 连接 ROM 输出数据到您的模块中
        );
    building2_rom ROM2_inst (
        .clka(vga_clk),
        .addra(img_addr),
        .douta(img_build2),
        .ena(disp_en) // 连接 ROM 输出数据到您的模块中
        );

endmodule
```

5-2-3、路面移动图层模块（Road_layer）

```
module Road_layer(
    input   vga_clk,                    // VGA 驱动时钟
    input   disp_en,                    // 显示有效信号
    input   forward,                    // 根据角色状态前进/后退
    input   [3:0] speed,                // 前进速度
    input   [11:0]  pix_xpos,           // 像素点横坐标
    input   [11:0]  pix_ypos,           // 像素点纵坐标
    output  reg  valid,                 // 该点是否有像素
    output  reg  [11:0]  pix_data       // 像素点数据
    );
    parameter   H_DISP    = 1024;       // 水平像素点数
    parameter   V_DISP    = 600;        // 竖直像素点数
    parameter   IMG_WIDTH = 256;
    parameter   IMG_HEIGHT = 141;
```

```
    reg [15:0] img_addr;
    wire [15:0] img_data;
    //马路移动
    wire clk_50Hz;
    reg [7:0] counter = 0;
    divider_1M d1M(vga_clk, 1'b1, clk_50Hz);
    always @(posedge clk_50Hz) begin
        if (forward)
            counter <= counter - speed;//时间顺行
        else
            counter <= counter + speed;//时间回溯
    end
    //显示马路
    always @(posedge vga_clk) begin
        if (disp_en) begin
            img_addr <= 12'b0; //文件指针归零
            if(pix_xpos >= 0 && pix_xpos < H_DISP && pix_ypos >= V_DISP - IMG_HEIGHT * 2 && pix_ypos < V_DISP)
                begin
                    //fff 为默认透明的区域
                    if (pix_ypos <= V_DISP - IMG_HEIGHT * 5 / 3 && img_data[11:8] >= 4'he && img_data[7:4] >= 4'he && img_data[3:0] >= 4'he)
                        valid <= 1'b0;//该图层透明像素点
                    else
                        valid <= 1'b1;//该图层有效数据
                    pix_data <= img_data[11:0];
                    img_addr <= (pix_ypos - V_DISP + IMG_HEIGHT * 2) / 2 * IMG_WIDTH + ((pix_xpos - counter) % IMG_WIDTH);//比色素早一步
                end
            else begin
                valid <= 1'b0;//该图层透明像素点
                pix_data <= 12'hfff;
            end
        end else begin
            valid <= 1'b0;
            pix_data <= 12'h000;
        end
    end

    road1_rom ROM_inst (
        .clka(vga_clk),
        .addra(img_addr),
        .douta(img_data),
        .ena(disp_en), // 连接 ROM 输出数据到您的模块中
        .wea()
        );

endmodule
```

5-2-4、角色及进度条图层模块（Chara_layer）

```
module Chara_layer(
    input   vga_clk,                    // VGA 驱动时钟
    input   disp_en,                    // 显示有效信号
    input   [3:0]  chara,               // 选择角色
    input   [9:0]  obstacle,            // 另一个角色
    input   [3:0]  obstimes,            // 另一个角色的清晰度
    input   [9:0]  progress,            // 游戏进度
    input   [8:0] offgress,             // 偏移进度
    input   [11:0]   pix_xpos,          // 像
```

```
素点横坐标
    input [11:0] pix_ypos,          // 像素点纵坐标
    output valid,                   // 该点是否有像素
    output [11:0] pix_data      // 像素点数据
    );
    parameter H_DISP = 1024;    // 水平像素点数
    parameter V_DISP = 600;     // 竖直像素点数
    parameter IMG_WIDTH = 200;
    parameter IMG_HEIGHT = 200;
    parameter SHRINK = 10;      // 线性缩小倍数
    parameter POSX = 512;           // 当前角色横坐标
    parameter POSY = 230;           // 当前角色纵坐标
    parameter SHARP = 16;           // 清晰度
    //读取图片信息
    wire [15:0] img_addr [1:0];     // 分别读两个角色的地址，0：伦太郎，1：红莉栖
    reg [15:0] ch_addr [1:0];       // 分别读两个角色的地址，0：当前角色，1：另一个角色红莉栖
    assign img_addr[0] = ch_addr[chara[0]];
    assign img_addr[1] = ch_addr[~chara[0]];
    wire [15:0] ch_data [1:0];      // 分别读两个角色的数据，0：当前角色，1：另一个角色
    //内部图层：0 为当前角色，1 为另一个角色
    reg [1:0] chvalid;
    reg [11:0] ch_pix_data [1:0];
    assign valid = chvalid[1] || chvalid[0];
    assign pix_data = chvalid[0] ? ch_pix_data[0] :
                      chvalid[1] ? ch_pix_data[1] :
                      12'hfff;//默认白色
    // 当前角色图层（包括进度条）
    always @(posedge vga_clk) begin
        if (disp_en) begin
            ch_addr[0] <= 12'b0; //文件指针归零
            // 显示进度条
            if(progress &&
                (!chara[0] && pix_xpos < progress + IMG_WIDTH / 2 / SHRINK ||
                chara[0] && pix_xpos > progress - IMG_WIDTH / 2 / SHRINK) &&
                pix_ypos >= V_DISP - offgress - IMG_HEIGHT / SHRINK && pix_ypos < V_DISP - offgress)
                begin
                if (!chara[0] && pix_xpos < progress - IMG_WIDTH / 2 / SHRINK ||
                    chara[0] && pix_xpos > progress + IMG_WIDTH / 2 / SHRINK) begin
                    chvalid[0] <= 1'b1;
                    ch_pix_data[0] <= 12'h0ff;//蓝条
                end else begin
                    //0ff 亦显示
                    chvalid[0] <= 1'b1;//该图层有效数据
                    ch_pix_data[0] <= ch_data[0][11:0];
                    ch_addr[0] <= (IMG_HEIGHT - (V_DISP - offgress - pix_ypos) * SHRINK) * IMG_WIDTH +
                    (pix_xpos + IMG_WIDTH / 2 / SHRINK - progress) * SHRINK;
                end
            end
            else begin
                // 显示当前角色
                if(pix_xpos >= POSX -
```

```
IMG_WIDTH / 2  && pix_xpos < POSX + IMG_WIDTH / 2  && pix_ypos >= POSY && pix_ypos < POSY + IMG_HEIGHT)
                                begin
                                    //0ff 为默认透明的区域
                                    if (ch_data[0][11:8] <= 4'h2 && ch_data[0][7:4] >= 4'hd && ch_data[0][3:0] >= 4'hd)

chvalid[0] <= 1'b0;//该图层透明像素点
                                    else

chvalid[0] <= 1'b1;//该图层有效数据

ch_pix_data[0] <= ch_data[0][11:0];
                                    ch_addr[0] <= (pix_ypos - POSY) * IMG_WIDTH + (pix_xpos - POSX + IMG_WIDTH / 2);
                                end
                            else begin
                                chvalid[0]  <= 1'b0;
                                ch_pix_data[0] <= 12'hfff;
                            end // end of this character
                        end // end of progress
                    end else begin
                        chvalid[0] <= 1'b0;
                        ch_pix_data[0] <= 12'h000;
                    end // end of disp_en
                end // end of always

                // 另一个角色图层
                always @(posedge vga_clk) begin
                    if (disp_en) begin
                        ch_addr[1] <= 12'b0; //文件指针归零
                        // 显示另一个角色
                        if (obstacle && obstimes && pix_xpos >= obstacle - IMG_WIDTH / 2  && pix_xpos < obstacle + IMG_WIDTH / 2  && pix_ypos >= POSY && pix_ypos < POSY + IMG_HEIGHT)
                            begin
                                //0ff 为默认透明的区域
                                if (ch_data[1][11:8] <= 4'h2 && ch_data[1][7:4] >= 4'hd && ch_data[1][3:0] >= 4'hd ||
                                    (pix_xpos + pix_ypos) % SHARP > obstimes)//模糊效果
                                    chvalid[1]  <= 1'b0;//该图层透明像素点
                                else
                                    chvalid[1]  <= 1'b1;//该图层有效数据
                                ch_pix_data[1]  <= ch_data[1][11:0];
                                ch_addr[1]  <= (pix_ypos - POSY) * IMG_WIDTH + (pix_xpos - obstacle + IMG_WIDTH / 2);
                            end
                        else begin
                            // 透明像素点
                            chvalid[1] <= 1'b0;//该图层透明像素点
                            ch_pix_data[1]  <= 12'hfff;;
                        end // end of other character
                    end else begin
                        chvalid[1] <= 1'b0;
                        ch_pix_data[1] <= 12'h000;
                    end // end of disp_en
                end // end of always

                wire [15:0] img_chara1;
                wire [15:0] img_chara2;
                wire [15:0] img_chris1;
                wire [15:0] img_chris2;
                //当前角色数据
                assign ch_data[0] = (!chara[0] ? (chara[1] ? img_chara2 : img_chara1) : //当前角色状态
                                    (chara[1] ? img_chris2 : img_chris1)); //当前角色状态
```

```
    //另一个角色数据
    assign ch_data[1] = (chara[0] ?
                                (chara[2]      ?
img_chara2 : img_chara1) : //另一角色状态
                                (chara[2]      ?
img_chris2 : img_chris1)); //另一角色状态
    //实例化 ROM
    chara1_rom ROM1_inst (
        .clka(vga_clk),
        .addra(img_addr[0]),
        .douta(img_chara1),
        .ena(disp_en), // 连接 ROM 输出
数据到您的模块中
        .wea()
        );
    chara2_rom ROM2_inst (
        .clka(vga_clk),
        .addra(img_addr[0]),
        .douta(img_chara2),
        .ena(disp_en)   // 连接 ROM 输
出数据到您的模块中
        );
    chris1_rom ROM3_inst (
        .clka(vga_clk),
        .addra(img_addr[1]),
        .douta(img_chris1),
        .ena(disp_en)   // 连接 ROM 输
出数据到您的模块中
        );
    chris2_rom ROM4_inst (
        .clka(vga_clk),
        .addra(img_addr[1]),
        .douta(img_chris2),
        .ena(disp_en)   // 连接 ROM 输
出数据到您的模块中
        );

endmodule
```

6、分频器模块（driver.v）

6-1、led 显示分频器（display_divider.v）

```
//分频器 5M
module divider_5M (
    input I_CLK, //输入时钟信号，上升沿有
效
    input rst,    //复位信号，低电平有效
    output O_CLK //输出时钟
);
// 默认分频倍数
parameter DIV_FACTOR = 5000000;
reg [$clog2(DIV_FACTOR) - 1:0] count; //
计数器
reg O_CLK_reg; // 内部时钟输出
initial O_CLK_reg <= 1'b0;//初始化
  // 计数器逻辑 同步复位
  always @(posedge I_CLK) begin
  if (!rst) begin
      count <= 0;
      O_CLK_reg <= 0;
  end else begin
      if (count == DIV_FACTOR - 1) begin
          count <= 0;
          O_CLK_reg <= ~O_CLK_reg; //
产生分频后的时钟信号
      end else begin
          count <= count + 1;
      end
  end
  end

  // 输出
  assign O_CLK = O_CLK_reg;
endmodule
```

6-2、vga 分频器（vga_driver.v）

```
//分频器 2
module divider_2 (
    input I_CLK,      //输入时钟信号，上
升沿有效
    input I_rst_n,    //复位信号，低电平有
效
    output O_CLK      //输出时钟
);
```

```
// 2 分频
reg O_CLK_reg; // 内部时钟输出
always @(posedge I_CLK or negedge I_rst_n)
begin
    if(!I_rst_n)
        O_CLK_reg    <=   1'b0          ;
    else
        O_CLK_reg                       <=
~O_CLK_reg   ;
end

  // 输出
  assign O_CLK = O_CLK_reg;
endmodule
```

Stimulation 测试文件代码

1. 控制器模块测试（Gate_ctrl_tb.v）

```
module Gate_ctrl_tb();

    reg clk_tb;              // 系统时钟
    reg rst_n_tb;            // 重置信号
    reg  [2:0]  op_tb;       // 旋转编码器操作信号
    reg  [1:0]  GL_over_tb;// 游戏逻辑模块的结束信号(win,over)
    reg VGA_comp_tb;         // VGA 完成一屏打印信号
    wire  RS_en_tb;          // 旋转编码器模块使能信号
    wire  GL_en_tb;          // 游戏逻辑处理模块使能信号
    wire  DP_en_tb;          // 开发板显示模块使能信号
    wire  [1:0]  DP_ch_tb; // 开发板显示类型
    wire  VGA_en_tb;         // VGA 显示模块使能信号

    // 实例化被测试模块
    Gate_ctrl uut(
        .clk(clk_tb),
        .rst_n(rst_n_tb),
        .op(op_tb),
        .GL_over(GL_over_tb),
        .VGA_comp(VGA_comp_tb),
        .RS_en(RS_en_tb),
        .GL_en(GL_en_tb),
        .DP_en(DP_en_tb),
        .DP_ch(DP_ch_tb),
        .VGA_en(VGA_en_tb)
    );

    // 时钟信号生成
    initial begin
        clk_tb = 0;
        forever  #10  clk_tb  =  ~clk_tb;   //
生成 50MHz 时钟
    end

    // 测试序列
    initial begin
        // 初始化输入
        rst_n_tb = 0;
        op_tb = 0;
        GL_over_tb = 0;
        VGA_comp_tb = 0;

        // 重置
        #10 rst_n_tb = 1;

        // 设置测试条件
        // 例如：开始游戏、游戏过程、游戏结束等
        #50 op_tb = 3'b001;
        #50 GL_over_tb = 1;
        #50 op_tb = 3'b010;

        VGA_comp_tb = 1;
        op_tb = 0;
        GL_over_tb = 0;
        #50 op_tb = 3'b001;
        #50 GL_over_tb = 1;
        #50 op_tb = 3'b010;
```

```
        // 结束测试
        #1000;
        $finish;
    end

endmodule
```

2. 游戏逻辑模块测试（Gate_logic_tb.v）

```
module Gate_logic_tb;

    // 定义时钟周期
    reg clk;
    // 为模块定义信号
    reg rst_n, next_s;
    reg [2:0] operation;
    wire character;
    wire [26:0] offset;
    wire [3:0] speed;
    wire [9:0] obstacle;
    wire [7:0] obstimes;
    wire game_over, game_win;

    //测试
    wire [12:0] phi;
    wire [21:0] off;
    // 实例化被测模块
    Gate_logic2 uut (
        .clk(clk),
        .rst_n(rst_n),
        .next_s(next_s),
        .operation(operation),
        .character(character),
        .offset(offset),
        .speed(speed),
        .obstacle(obstacle),
        .obstimes(obstimes),
        .game_over(game_over),
        .game_win(game_win),
        //测试
        .phi(phi),
        .off(off)
    );

    // 定义初始时钟周期
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 定义一个 50%占空比的时钟信号，周期为 10 个时间单位
    end

    // 定义测试过程
    initial begin
        // 初始化所有输入信号
        rst_n = 0;
        next_s = 0;
        operation = 0;

        // 开始测试用例
        // 等待一段时间，然后激活复位信号
        #10 rst_n = 1;

        // 在此添加更多的测试步骤，根据需要设置输入信号
        #20 operation = 3'b001;
        repeat(4) begin #5 next_s = 1; #5 next_s = 0; end // n 次运算

        #20 operation = 3'b101;
        repeat(4) begin #5 next_s = 1; #5 next_s = 0; end // n 次运算

        #20 operation = 3'b011;
        repeat(100) begin #5 next_s = 1; #5 next_s = 0; end // n 次运算

        // 等待一段时间，观察输出信号
        #100 $display("Character: %b, Offset: %b, Speed: %b, Obstacle: %b, Game Over: %b, Game Win: %b", character, offset, speed, obstacle, game_over, game_win);

        // 在此添加更多的测试步骤，根据需要设置输入信号

        // 结束仿真
    end

endmodule
```

3．旋转编码器模块测试（RotaryEncoder_tb.v）

```
module RotaryEncoder_tb;
    // Inputs
    reg clk;
    reg reset;
    reg SIA;
```

```
reg SIB;
reg SW;

// Outputs
wire CW;
wire CCW;
wire Pressed;

// 实例化被测试模块
RotaryEncoder uut (
    .clk(clk),
    .reset(reset),
    .SIA(SIA),
    .SIB(SIB),
    .SW(SW),
    .CW(CW),
    .CCW(CCW),
    .Pressed(Pressed)
);

initial begin
    // 初始化信号
    clk = 0;
    reset = 1;
    SIA = 0;
    SIB = 0;
    SW = 0;

    // 复位
    #20;
    reset = 0;

    // 模拟顺时针旋转
    #20; SIA = 1; SIB = 0;
    #20; SIA = 1; SIB = 1;
    #200; SIA = 0; SIB = 1;
    #20; SIA = 0; SIB = 0;

    // 模拟逆时针旋转
    #80; SIA = 0; SIB = 1;
    #20; SIA = 1; SIB = 1;
    #200; SIA = 1; SIB = 0;
    #20; SIA = 0; SIB = 0;

    // 模拟按压动作
    #80; SW = 1;
    #200; SW = 0;

    // 结束模拟
    #100;
end

// 时钟信号生成
always #10 clk = ~clk; // 生成 50MHz 的时钟

endmodule
```

4.七段晶体管测试模块（seven_segment_display_tb.v）

```
module testbench;

    reg clk;
    reg reset;
    reg [7:0] choice;
    wire [7:0] anodes;
    wire [7:0] segments;

    // 实例化七段显示模块
    test_ssd uut (
        .clk(clk),
        .reset(reset),
        .choice(choice),
        .anodes(anodes),
        .segments(segments)
    );

    // 时钟信号生成
    initial begin
        clk = 0;
        forever #10 clk = ~clk; // 生成一个周期为 20ns 的时钟信号
    end

    // 测试用例
    initial begin
        // 初始化
        reset = 1;
        number = 0;
        #100; // 等待一段时间以稳定初始状态

        // 释放重置信号
        reset = 0;
        number = 32'h12345678; // 设置一个测试数字
        #1000; // 等待一段时间以观察输出

        // 更改数字
        number = 32'h87654321;
        #1000; // 继续观察输出

        // 重置并结束测试
        reset = 1;
        #100;
        $finish;
    end

endmodule
```

5. VGA 驱动器测试（VGA_driver_tb.v）

```
module VGA_driver_tb;
    reg clk;          // 输入时钟信号
    reg rst_n;        // 复位信号（低有效）
    wire hsync;       // 水
```

```
平同步信号
    wire vsync;        // 垂直同步信号
    wire [3:0] red; // 红色分量
    wire [3:0] green; // 绿色分量
    wire [3:0] blue;    // 蓝色分量

    // 定义时钟参数
    parameter CLK_PERIOD = 2; // 时钟周期（单位：ns）

    // 生成时钟信号
    always #((CLK_PERIOD / 2)) clk = ~clk;

    // 产生复位信号
    initial begin
        clk = 0;
        rst_n = 0;
        #10; // 等待一段时间后释放复位
        rst_n = 1;
    end

    // 实例化待测试的 VGA_driver 模块
    VGA_driver dut (
        .clk(clk),
        .rst_n(rst_n),
        .hsync(hsync),
        .vsync(vsync),
        .red(red),
        .green(green),
        .blue(blue)
    );

    // 添加其他测试逻辑和断言（Assertions）...

    // 模拟时长
    initial #5000 $finish; // 模拟 5000 个时钟周期后停止仿真

endmodule
```

Test 下板测试文件代码

1. 综合测试模块（Gate_test.v）

```
module Gate_test(
    input clk,                  // 时钟信号
    input reset,                // 重置信号，高电平有效
    input [2:0] SIABW,          // 旋转编码器 SIA、SIB、SW 引脚
    output [7:0] anodes,        // 阳极信号（AN0..AN7）
    output [7:0] segments, // 段信号（A..G）和小数点 DP
    output [15:0] led,          // led 速度显示器
    output                      vga_hs,     // 行同步信号
    output                      vga_vs,     // 场同步信号
    output          [11:0]      vga_rgb     // 红绿蓝输出
    );
  wire [2:0] operation;   // 操作标记
  wire [26:0] offset;       // 世界线偏移量 0-999999
  wire VS_negedge;          // 开始运算记号，高电平激发（频率为 60Hz）
  wire [2:0] charac;        // 角色标记
  wire [3:0] speed;         // 速度标记
  wire [9:0] obstacle;    // 障碍物位置
  wire [7:0] obstimes;    // 障碍物大小
  wire [9:0] progress;    // 进度标记
  wire [7:0] offgress;    // 进度标记
  wire game_over;           // 游戏结束标记
  wire game_win;            // 游戏胜利标记
  // 旋转编码器输入
  RotationSensor rs_uut(
      .clk(clk),                    // 时钟信号
      .rst_n(!reset),               // 复位信号
      .SIABW(SIABW),                // 旋转编码器 SIA、SIB、SW 引脚
      .operation(operation)         // 顺时针、逆时针、按压检测
  );
  // 实例化中心逻辑
  meet_logic gate_uut (
      .clk(clk),
```

```
        .rst_n(!reset),
        .next_s(VS_negedge),
        .operation(operation),

        .character(charac),
        .offset(offset),
        .speed(speed),
        .obstacle(obstacle),
        .obstimes(obstimes),
        .progress(progress),
        .offgress(offgress),

        .game_over(game_over),
        .game_win(game_win)
    );
  // 辉光管显示 offset 值
  display_seg seg_uut(
         .clk(clk),
         .reset(!reset),
         .choice({!game_win, game_over}),
         .hex(offset),
         .speed(speed),
         .anodes(anodes),
         .segments(segments),
         .led(led)
   );
   // VGA 显示屏
    vga_top vga_uut(
         .clk_100MHz(clk),            // 标准时钟
         .rst_n(!reset),              // 复位信号，高电平有效（方便测试）

         .chara(charac),              // 角色状态
         .speed(speed),               // 行进速度
         .obstacle(obstacle),         // 障碍物位置
         .obstimes(obstimes),         // 障碍物大小
         .progress(progress),         // 进度条
         .offgress(offgress),         // 进度条

         .vga_hs(vga_hs),             // 行同步信号
         .vga_vs(vga_vs),             // 场同步信号
         .vga_rgb(vga_rgb),           // 红绿蓝输出
         .VS_negedge(VS_negedge)      // 下降沿信号
    );
endmodule
```

2. 开发板综合显示测试模块（test_ssd.v）

```
module test_ssd(
    input clk,                  // 时钟信号
    input reset,                // 重置信号，高电平有效
    input  [7:0]  number,       // 要显示的数字（一共八个数字供测试）
    input [3:0] speed,          // 要显示的速度值
    output  [7:0]  anodes,      // 阳极信号（AN0..AN7）
    output [7:0] segments, // 段信号（A..G）和小数点 DP
    output  [15:0]  led         // led 速度显示器
    );
    reg [1:0]ch = 0;
    reg [26:0]hex;
    // 实例化开发板显示模块
    display_seg uut(
        .clk(clk),
        .reset(!reset),
        .choice(ch),
        .hex(hex),
        .speed(speed),
        .anodes(anodes),
        .segments(segments),
        .led(led)
        );
    // 测试数据
    always @(number) begin
```

```
        casex(number)
            8'b1xxxxxxx: begin ch <= 2'b01; end
            8'b01xxxxxx: begin ch <= 2'b11; end
            8'b001xxxxx: begin hex <= 01010101; ch <= 2'b10; end
            8'b0001xxxx: begin hex <= 10271828; ch <= 2'b10; end
            8'b00001xxx: begin hex <= 99999999; ch <= 2'b00; end
            8'b000001xx: begin hex <= 98765432; ch <= 2'b00; end
            8'b0000001x: begin hex <= 55555555; ch <= 2'b00; end
            8'b00000001: begin hex <= 123; ch <= 2'b00; end
            default: begin hex <= 00000000; ch <= 2'b00; end
        endcase
    end
endmodule
```

3. VGA 综合显示测试模块（vga_test.v）

```
module vga_test(
    input                clk_100MHz,
// 标准时钟
    input                rst_n,
// 复位信号，高电平有效（方便测试）
    output               vga_hs,
// 行同步信号
    output               vga_vs,
// 场同步信号
    output      [11:0]   vga_rgb
// 红绿蓝输出
    );
wire VS_negedge;
// 前行速度
reg chara = 0;
reg [3:0] speed = 0;
reg [9:0] obstacle = 512;
reg [7:0] obstimes = 100;
wire clk_1Hz;
divider_100M d100M(clk_100MHz, !rst_n, clk_1Hz);
always @(posedge clk_1Hz) begin
    speed <= speed + 1;
    if (speed == 0)
        chara <= ~chara;
    obstacle <= 512;
    obstimes <= 100;
    //if (obstimes == 0)obstimes <= 50;
    //else obstimes <= obstimes + 1;
end
//VGA 顶层模块
vga_top vga_inst(
    .clk_100MHz(clk_100MHz),        // 标准时钟
    .rst_n(!rst_n),                 // 复位信号，高电平有效（方便测试）

    .chara(chara),              // 角色状态
    .speed(speed),              // 行进速度
    .obstacle(obstacle),        // 障碍物位置
    .obstimes(obstimes),        // 障碍物大小

    .vga_hs(vga_hs),                // 行同步信号
    .vga_vs(vga_vs),                // 场同步信号
    .vga_rgb(vga_rgb),              // 红绿蓝输出
    .VS_negedge(VS_negedge)         // 下降沿信号
    );
endmodule
```