目录

1 项目说明

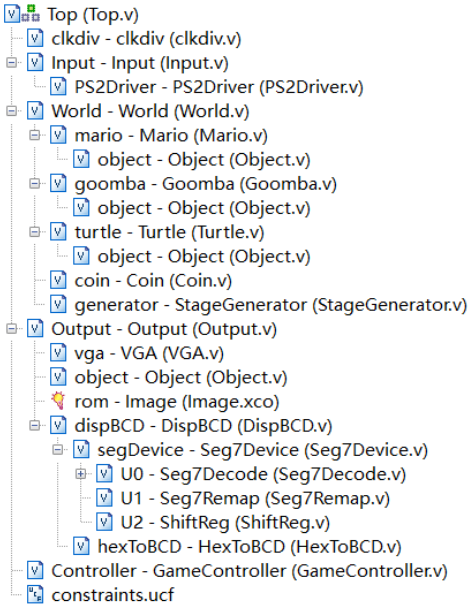
FPGA 超级马里奥游戏使用 PS2 键盘作为输入，VGA、主板七段码作为输出。其中 VGA 用于显示游戏界面，七段码显示游戏所得的分数。PS2 键盘中使用到了左方向键、右方向键、空格键和 R 键。其中方向键用于控制马里奥左右行走，空格键控制跳跃，R 键用于重新开始游戏。游戏分数中，硬币奖励 5 分，升级奖励 20 分，杀死小怪奖励 10 分。

本项目目的在于尽可能还原原版超级马里奥的游戏界面和操作逻辑，此目标已基本实现。游戏包含的要素有马里奥（Mario）、小怪（goomba）、乌龟（turtle）、砖块、空箱子（box）、未知箱子（包括金钱奖励、升级奖励）、管道（pipe）、硬币（coin）、城堡（castle）、地面（grass）等。游戏动画齐全，逻辑正确，画风还原，交互自然。

2 设计说明

2.1 项目架构

整个项目架构的结构图如下。



项目的核心原理在于将 VGA 扫描坐标信号（col_addr 和 row_addr）映射为 Rom 内存（Image.xco）中的像素点颜色，从而将图片显示在 VGA 上。

下面将从底层模块一步步说明原理，直至最顶层 top 模块。

2.2 内存模块（Image.xco、Image.v）

Image 模块为项目最底层的模块。该模块由 ISE 的 IP core 工具自动生成，类型为 Block Memory Generator。该 Rom 设计为只读模式，其中内存宽度为 12，深度为 262,253。读写深度和宽度均相同。内存中按顺序储存游戏用到的所有界面素材，包括元素的所有动画帧，总共有 36 张游戏图片。所有图片采用 12 位色储存方式（用于匹配 VGA 驱动模块的 12 位 vga_data），每个素材都对应着一个 id。

24 位图片原素材已储存在项目根目录下 converter 文件夹中。内存中每个图片素材都拥有一个内存起始位置，该位置记录在 Object 模块中。为了方便生成 Rom 模块的初始化 coe 文件，笔者使用易语言编写了一个转换工具。在将原素材转换为 12 位图片素材的同时，生成了每个素材的起始内存偏移位置，方便之后调用。详见以下的 Object 模块。

值得一提的是原素材的透明色为 F0F，这也是内存中储存的颜色。因此游戏中还需要将该透明色转换为背景蓝色。

2.3 元素管理模块（Object.v）

Object 模块将素材的 id 映射为该素材的长度（h），宽度（w）以及内存中的起始偏移量（addr），其中 h 和 w 在游戏核心模块 World 中使用，用于进行碰撞检测、逻辑处理和内存偏移计算。addr 则在输出模块 Output 中用于从内存中读出数据。Object 模块由 2.2 一节所述的转换工具自动生成，它的输入输出定义如下：

```
1. module Object(
2.     input [5:0] id, // 0 - 35 (64)
3.     output reg [10:0] h,
4.     output reg [10:0] w,
5.     output reg [18:0] addr
6. );
```

Object 模块采用 Verilog 行为描述，使用 always @*和 case 语句拟合出组合电路。该模块用于 Mario 模块、Output 模块、Turtle 模块和 Coin 模块。

2.4 输出模块（Output.v）

Output 模块的作用如下：

1. VGA 处理操作。Output 模块包含 VGA 驱动模块，输出它的扫描信号 row_addr 和 col_addr，同时接受该扫描信号对应的屏幕位置参数。该位置参数为 type、h 和 w，分别指该位置所属于的素材 ID、该位置相对于素材的横坐标、相对于素材的纵坐标。Output 模块将根据以上三个信息结合 Object 模块取得该点在内存中的地址。计算的核心代码如下：

```
wire [18:0] addr_begin;
wire [18:0] addr_offset;
wire [10:0] height;
wire [10:0] width;

Object object(
    .id(type),
    .h(height),
    .w(width),
    .addr(addr_begin)
);  // 根据接受的三个位置参数取得内存起始位置

assign addr_offset = addr_begin + h * width + w;  // 计算内存偏移量

Image rom(
    .clka(clkdiv[0]),
    .wea(1'b0),
    .addra(addr_offset),
    .dina(12'b0),
    .douta(rom_data)
);  // 取得内存的 12 位颜色数据

wire [11:0] scene_data;
assign scene_data = (rom_data != 12'hF0F ) ? rom_data : 12'h9CD;//透明色
```

这个设计是考虑到了一个时钟周期只能读 rom 内存一次的特点，因此每当一个时钟上升沿来到时，Output 模块就进行一次映射操作（素材 ID 和素材相对位置 => 内存偏移量）和内存读操作。这样就解决了无法一次性将指定素材的所有数据都读出来的不足。

2. 处理分数数据。这一部分比较简单，将输入的 num 数据输出至主板七段码驱动模块即可。

3. 处理游戏结束时的黑圈放大缩小操作。该操作主要由 2.7 节将详细说明的 GameController 模块进行。Output 模块接受其传过来的一个 12 位的 mask 参数，该参数只有两种取值：12'h000 和 12'hFFF，分别指该点为黑色、该点不变色。因此只需将 scene_data 和 mask 做一次与操作即可。代码如下：

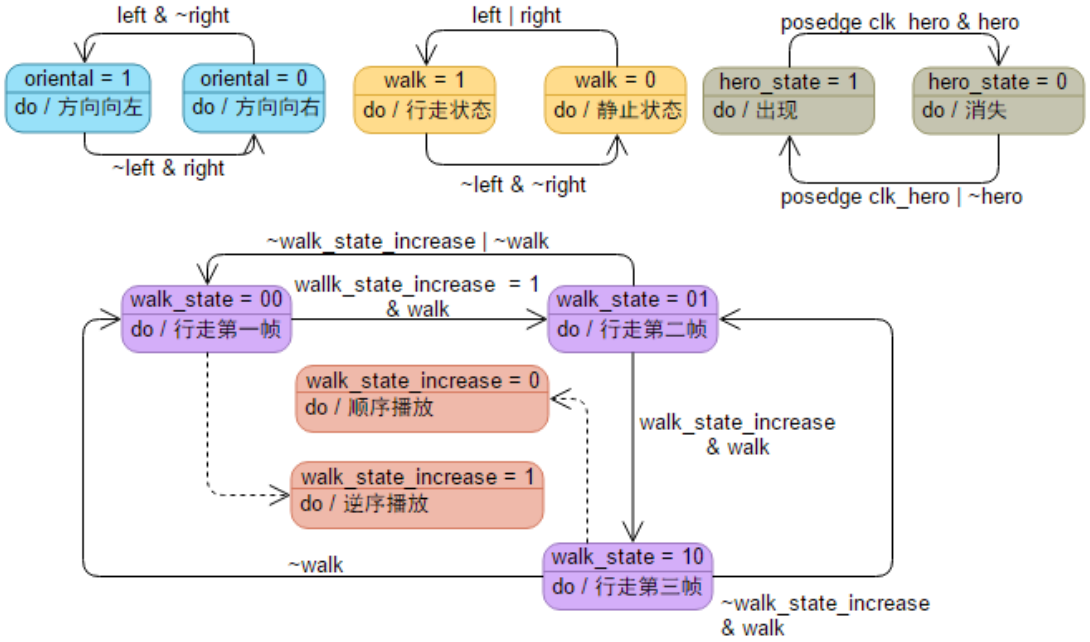
```
1. assign vga_data = mask & scene_data;
```

2.5 游戏元素

该部分将详细介绍游戏的元素。游戏核心为 World 模块以及内部包含的 Mario 模块、Goomba 模块和 Coin 模块。内部包含的三个模块仅仅用于对应游戏元素的内部操作，如逐帧加载动画、方向控制、消隐处理等，不涉及位置移动、碰撞检测等。这些部分由 World 模块进行处理。

2.5.1 Mario 模块（Mario.v）

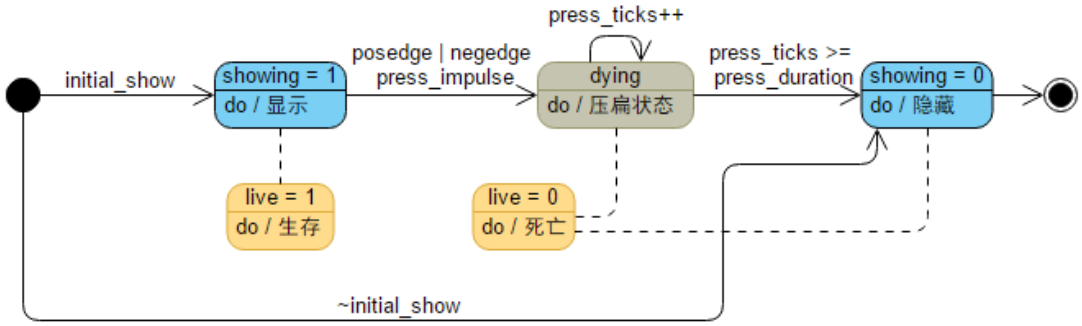
Mario 模块用于控制马里奥。它控制了马里奥的方向（oriental），无敌状态（hero 和 hero_state），等级状态（level），行走帧序号（walk_state）等。Mario 模块的状态转移图如下所示：



以上的 walk_state 状态转移沿时钟 clk_walk_anim 正边沿触发。其他转移若未标明触发条件，皆为 clk 正边沿触发。根据马里奥的方向、行走帧、无敌状态，即可决定此刻马里奥的素材并输出对应的 ID。输出 ID 的过程是未受时钟沿触发控制的组合电路，代码采用 always @*和 case 语句进行编写。Mario 模块同时接入了 Object 模块，输出该 ID 对应的素材宽高至主模块 World，以进行边缘碰撞处理。

2.5.2 Goomba 模块（Goomba.v）

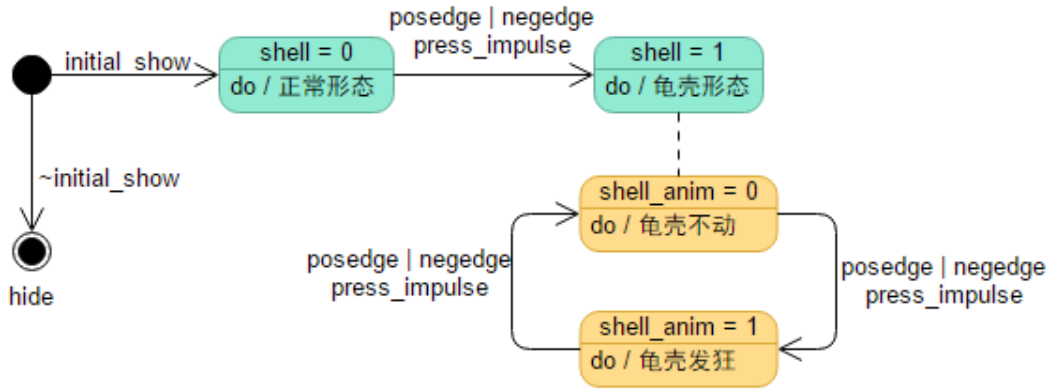
Goomba 模块用于控制小怪，内部状态包含是否显示、是否生存、是否被压扁等。它的状态转移图如下所示：



其中，在 showing 状态至 dying 状态的转移过程里，press_ticks 将初始化为 0，并在之后每一次触发中自增。直至 press_ticks 大于给定值时，小怪才进入消失状态。这就实现了对压扁的控制。live 状态的用途在于判定马里奥受伤条件。只在其 live 状态为真时，马里奥才会受到伤害。图中未标明触发条件的皆为 clk 正边沿触发。此模块同样实现了 Object 模块以输出宽高。

2.5.3 Turtle 模块（Turtle.v）

Turtle 模块用于控制乌龟。乌龟包含的状态有 shell、shell_anim、walk_state 等，其中 walk_state 用于实现乌龟行走时的动画，逻辑和马里奥行走时的动画一致，此处略去不谈。其它参数的状态转移图如下所示：



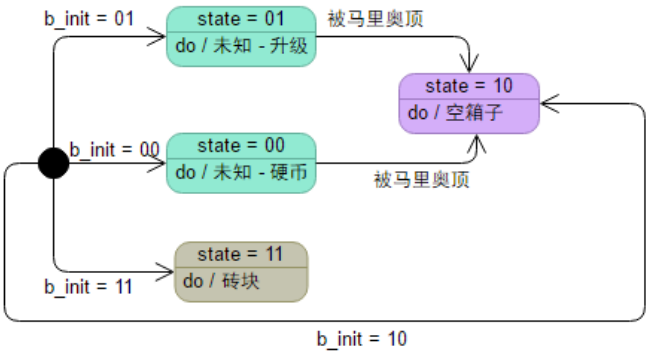
此模块同样实现了 Object 模块以输出宽高。

2.5.4 Coin 模块（Coin.v）

Turtle 模块用于控制硬币。硬币的动画逻辑与之前的相同，不再赘述。由于硬币所有素材的宽高均为 40，因此此处不需要 Object 模块来输出宽高。

2.5.5 管道（Pipe）、箱子（Box）、地面（Grass）和城堡（Castle）

这四种元素并没有特地编写模块来处理（这是因为内部状态并不复杂，并未牵涉到动画等复杂的状态转移处理），而是直接写在主模块 World 中。其中值得一提的是 Box 的状态，它的状态转移图如下所示：



其中状态 01 与状态 00 的素材相同，都是带有问号的箱子。

2.6 游戏初始化模块（StageGenerator.v）

游戏初始化用到了 StageGenerator 模块。该模块由 python 文件自动生成。游戏关卡可以在根目录下的 map/stage.txt 中编写，再运行 converter.py 即可将其编译为 Verilog 代码，名为 StageGenerator.v。在 stage.txt 中，M 表示马里奥位置，s 表示砖块，小写 c 表示带有硬币奖励的未知箱子，m 表示带有升级奖励的未知箱子，b 代表空箱子，大写 C 表示硬币，t 表示乌龟，g 表示小怪，p 表示管道，E 表示终点（城堡）。字符的行位置表示初始 x 坐标，列位置表示初始 y 坐标。

StageGenerator 模块的输入输出定义如下：

```
1. module StageGenerator(
2.     output [12:0] mario_x,
3.     output [12:0] mario_y,
4.     output [12:0] map_width,
```

```
    output [13*8-1:0] goomba_x, // disable when {x, y} = 00
    output [13*8-1:0] goomba_y,
    output [13*8-1:0] turtle_x,
    output [13*8-1:0] turtle_y,
    output [13*64-1:0] box_x,
    output [13*64-1:0] box_y,
    output [2*64-1:0] box_state, // 0: coin, 1: pilz, 2: box, 3: stone
    output [13*8-1:0] pipe_x,
    output [13*8-1:0] pipe_y,
    output [12:0] castle_x,
    output [12:0] castle_y,
    output [13*16-1:0] coin_x,
    output [13*16-1:0] coin_y
    );
```

模块中全部采用 assign 的连接方法，分别输出游戏初始化的各个参数至 World 模块。初始化参数包括各个元素的横纵坐标、地图大小、箱子初始状态等。若横纵坐标皆为 0，表示该物体不存在。为方便起见，所有的元素初始化参数采用并行输出的方法。

2.7 游戏核心逻辑处理（World.v）

游戏核心包括游戏关卡初始化、碰撞检测、精灵移动、界面输出等。这些部分的处理大多数都在 World.v 中进行。World 模块的输入输出定义如下：

```
module World(
    input [31:0] clkdiv,
    input rstn,
    input jump,
    input left,
    input right,
    input [8:0] row_addr, // pixel ram row address, 480 (512) lines
    input [9:0] col_addr, // pixel ram col address, 640 (1024) pixels
    output reg [5:0] type, // 63 represents background color
    output reg [10:0] h, // relative height in the object.
    output reg [10:0] w,  // relative width in the object.
    output reg [1:0] over,  // 0x: sceen; 10: gameover; 11: win
    output [9:0] cx,
    output [8:0] cy,
    output reg [31:0] score
    );
```

游戏定义了一个变量 view_x，用于储存镜头在横版游戏地图中的横坐标。因此，所有物

体在 VGA 屏幕中的横坐标即为游戏坐标减去 view_x。view_x 取决于马里奥的坐标，它的目标是尽可能将马里奥放在镜头中央。若 view_x 小于 0 或使得屏幕右侧坐标大于游戏地图宽度，则调整 view_x 的值。

2.7.1 游戏关卡初始化

在 World 模块中，StageGenerator 模块产生初始化参数。当 rst 参数为真时，游戏开始进行初始化。在初始化过程中，系统使用了循环来对各个元素种类中的所有元素进行状态处理，包括初始坐标、初始状态、存在状态等。同时也对几个时钟参数 ticks_1、ticks_2、ticks_1_5、clk_17 等进行初始化。

在该系统中，我们约定物体的坐标为最左下角点的坐标。详细参数可参见 2.6.4 中的示意图。

在创建几个精灵的子模块中，World 模块使用了 generate 语句块来快速进行批量连接，从而减少了大量代码的编写。

2.7.2 碰撞检测

World 模块只关注对精灵和障碍物（pipe、box、grass、boundary）之间的碰撞检测。碰撞检测的核心是 bound 函数，它的代码如下：

```
1.  function [6+1+1:0] bound; // box_index(7) | box(1) | bound(1)
2.      input [L-1:0] x; input [L-1:0] y; input [W-1:0] w; input [W-
    1:0] h;
3.      integer bound_j;
4.      begin
5.          bound = 9'b0;
6.          if (y >= 440) bound[0] = 1; // ground
7.          else if (x <= 0) bound[0] = 1; // left
8.          else if (x + w >= map_width - 1'b1) bound[0] = 1; // right
9.          else begin
10.             // box
11.             for (bound_j = 0; bound_j < N2; bound_j = bound_j + 1) begin
12.                 if (b_en[bound_j] & overlap(x, y, w, h, b_x[bound_j], b_
    y[bound_j], 40, 40)) begin
13.                     bound[8:2] = bound_j;
14.                     bound[1:0] = 2'b11;
15.                 end
16.             end
```

```
            // Pipe
            for (bound_j = 0; bound_j < N3; bound_j = bound_j + 1) begin
                if (p_en[bound_j] & overlap(x, y, w, h, p_x[bound_j], p_y[bound_j], 80, 80)) begin
                    bound = 9'b1;
                end
            end
        end
    end
endfunction
```

bound 函数接受某个物体的坐标和宽高，输出该物体与障碍物的碰撞状态。函数的输出有 9 位，其中，最后一位表示该物体是否与障碍物重叠，倒数第二位表示该障碍物是否为箱子。在障碍物为箱子的情况下，最左边的七位则表示了箱子障碍物的序号（index）。这个目的在于在游戏逻辑中判定是否为未知箱子，从而实现马里奥对未知箱子的“顶”操作。

bound 函数调用了 overlap 函数，它的代码如下：

```
function overlap;
    input [L-1:0] x1; input [L-1:0] y1; input [W-1:0] w1; input [W-1:0] h1;
    input [L-1:0] x2; input [L-1:0] y2; input [W-1:0] w2; input [W-1:0] h2; begin
        overlap = x1 + w1 > x2 & x2 + w2 > x1 & y1 - h1 < y2 & y2 - h2 < y1;
    end
endfunction
```

overlap 接受两个物体的坐标和宽高，返回该两个物体（两个矩形）是否有重叠。

2.7.3 精灵移动与游戏逻辑

游戏中使用 clkdiv[17]作为游戏进程时钟。在每一个周期，World 模块将判断每个精灵的移动状态（如方向状态 oriental 等），根据移动的目标进行坐标的加减。在实现对坐标增减的时候，系统首先使用 temp_x 或 temp_y 的 reg 型变量记录目的坐标，再将该目的坐标传入 2.6.2 节所述的 bound 函数进行碰撞检测。倘若不存在碰撞情况，则将精灵的坐标赋值为 temp_x 和 temp_y。否则，则判定为碰撞，系统进行碰撞处理。

具体来说，对于乌龟和小怪，发生碰撞时，系统将对对应的 collapsion 变量进行取反处理，从而向这两者对应的模块传递“碰撞”信号。两个模块在检测到碰撞信号时，将对 oriental

输出端口的值进行取反操作。这就实现了这两者的自动转向。

对于马里奥，移动的逻辑略微复杂。首先，系统将检测马里奥是否处于跳跃状态（jump_state）。若是，则根据上移或下降状态进行 y 坐标的处理。若马里奥处于上升状态（jump_state＝10）且发生了碰撞，或者是上升计数（jump_ticks）达到了给定的目标值，则将上升状态改为下降状态（jump_state＝11），之后马里奥将发生下降。同时，若马里奥处于上升状态，且与上方某个未知箱子发生了碰撞，则打开该箱子，进行一定的奖励（升级或得到金币）。

当马里奥顶到藏有蘑菇的箱子时，马里奥立即实现升级。升级时，系统将直接对马里奥的坐标进行下移处理，从而防止其发生变大后卡进箱子中无法动弹的情况。升级时系统调用了 upGrade 函数，将 level 变量置为 1。马里奥模块将捕捉到该变量的改变，从而改变素材的 ID，实现马里奥的变大操作。同时，马里奥的 hero 状态变为 1，hero_ticks 清零，开始了无敌状态。hero_ticks 在每一个时钟周期时发生自增，到了一定程度后，系统将重置 hero 变量，使马里奥失去无敌状态。马里奥模块同样可捕捉到 hero 变量的改变，从而使自身快速闪烁。升级时，系统将会进行加分操作。

当马里奥顶到藏有硬币的箱子时，马里奥拿到了金币奖励。为了增强游戏的体验，系统将对某个硬币进行上升操作。该硬币的使能开关将被打开，从而使其可视。硬币将从这个箱子出发，不断地减小 y 坐标，视觉效果看上去好像马里奥撞飞了一个硬币。当该硬币的 y 坐标为 0 时，重新隐藏该硬币。

当马里奥处于下降状态时，系统将进行下方物体的碰撞检测。若发生了碰撞，则将马里奥的 jump_state 变为 00，从而将马里奥跳跃状态归位。同时，在下降过程中，系统将判断马里奥下方是否有小怪或乌龟。若是，则将对应精灵的 press 变量取反，向对应模块发出信号。对应精灵的模块将捕捉到 press 的改变，从而进行被压操作，改变自身状态。倘若下方精灵为小怪，则进行加分操作。

对马里奥下方物体的碰撞检测是一直进行的。即使马里奥不处于下降状态，系统能会判断下方是否存在支撑物。若否，则将 jump_state 置为 11，实现马里奥的掉落处理。

对于马里奥的左右移动同样是先用 temp_x 储存预计的横坐标并进行碰撞检测。倘若不存在障碍物，则马里奥的 x 坐标发生了改变，实现左右移动。游戏中，系统会使用 overlap 函数，实时监测马里奥是否与小怪或乌龟发生重叠现象。若发生重叠且马里奥不处于无敌状

态（hero = 0），则对马里奥进行降级处理。

降级处理在 downGrade 函数中进行。该函数首先判断 level 变量是否为 1。若是，则马里奥处于变大状态，将马里奥的 level 置为 0，同时进行无敌处理（将 hero 置为 1）。若不是，则游戏结束。

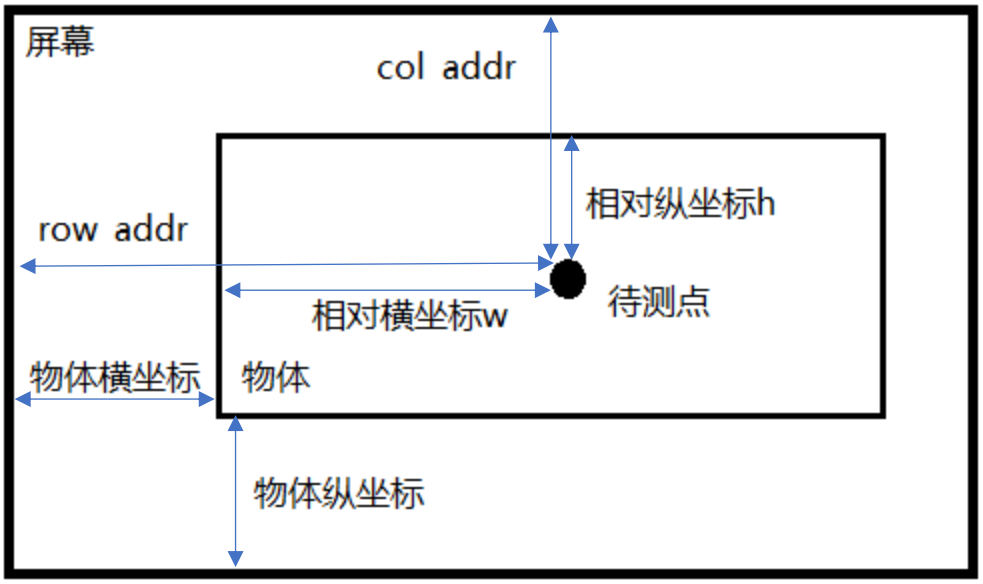
游戏结束与输出端口 over 变量有关。当 over 变量为 11 时，游戏失败。外部模块将捕捉到 over 的改变，从而进行游戏进程的处理。当 over 变量为 10 时，游戏胜利。

马里奥在游戏进程中同时会使用 overlap 函数，检测是否与硬币发生碰撞。若是，则进行加分操作并使硬币的使能开关置为 0，从而实现了硬币的隐藏。同样，游戏进程中会检测是否与城堡（castle）发生碰撞。若是，则游戏胜利，over 变量为 11。

2.7.4 界面输出操作

界面输出处理与游戏逻辑处理的触发时钟不同。界面输出采用 clkdiv[1]即可，因为这与 vga 扫描信号的时钟周期一致。

界面输出的核心是将扫描信号的坐标 row_addr 和 col_addr 映射为物体的类型，以及这一点相对于物体贴图的相对坐标。下图为相对坐标、绝对坐标的示意图。



对照上图，系统即可将扫描信号转为相对横纵坐标 h 和 w 输出至 Output 模块。Output

模块将对 w、h 计算出待测点相对起始点的内存偏移地址，再根据图片的起始偏移地址得出该点颜色在内存中的位置。具体操作见上 2.4 一节。

当每次 clkdiv[1]上升沿到来时，系统将对所有的元素进行遍历操作。倘若该点在该物体元素内，就利用该物体的横纵坐标计算出相对坐标，实现对物体的输出。

2.8 游戏控制模块（GameController.v）

Game Controller 模块用于对 World 模块进行控制，同时协助游戏的初始化和结束操作。

初始化阶段，GameController 模块将三位变量 init_ticks 归零。在下四个时钟周期内，每次 init_ticks 将发生自增直至 2'b111。在这之前，输出端口 rst 模块将保持为 1。rst 端口最终将传入 World 模块的 rst 端口，从而进行了 7 个时钟周期的初始化操作。这是因为 World 模块的初始化有多个步骤，无法在一个时钟周期内完成。

GameController 同时也控制了游戏可见画面（光圈）的变大变小操作。游戏开始时，屏幕全为黑，光圈半径为 0。此后每次在 clk_frame 时钟上升沿触发时，将光圈半径自增 1，直至其变为 MAX_RADIUS。在游戏结束时，同样在每次触发时自减光圈半径直至 0，再开始新一轮的初始化和光圈变大。

对光圈半径的操作体现在 12 位输出端口 mask 上。GameController 同样接受扫描信号 col_addr 和 row_addr，并使用勾股定理判断该扫描信号是否在光圈内。若在光圈内，则 mask 变量为 12'hFFF；否则为 12'h0。在游戏进程中，mask 将恒为 12'h0，保证整个画面的可见性。

该模块除了接受 World 传来的 over 信号外，同时也接受外部的 retry 信号。当这些信号蕴含了重新开始的信号时，则进行游戏结束操作和重新开始操作，伴随着光圈的缩放。

2.9 输入模块（Input.v）

输入模块较为简单，主要是对 PS2 驱动模块的封装。这个模块接收了 PS2 传来的空格、左方向键、右方向键和 R 键的按下和放开信号，将其转化为 left、right、jump、retry 信号的 0 和 1，并输出至外部模块。

2.10 顶级模块（Top.v）

最后介绍一下最顶层的模块。Top 模块则将游戏的所有部分连接在一起，组合成一个完

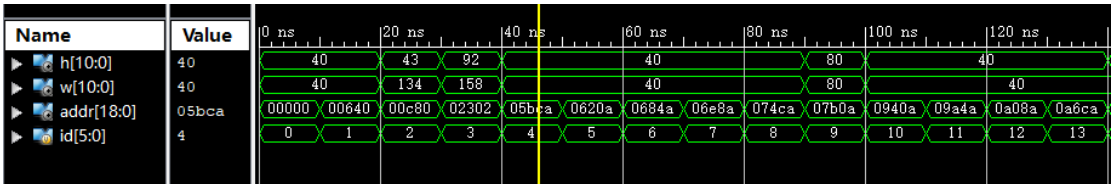
整的游戏系统。它包含了 clkdiv 分频模块、Input 输入模块、World 游戏核心模块、GameController 游戏控制模块和 Output 输出模块，并将对应的端口连接在一起。Top 模块不进行任何游戏逻辑操作，只负责进行子模块的连接，同时利用输入输出模块实现与外部硬件设备的通信。

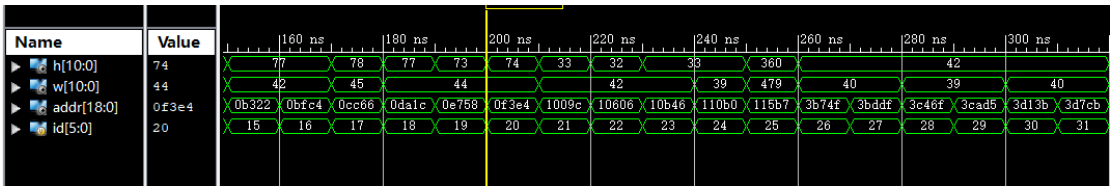
3 调试过程分析

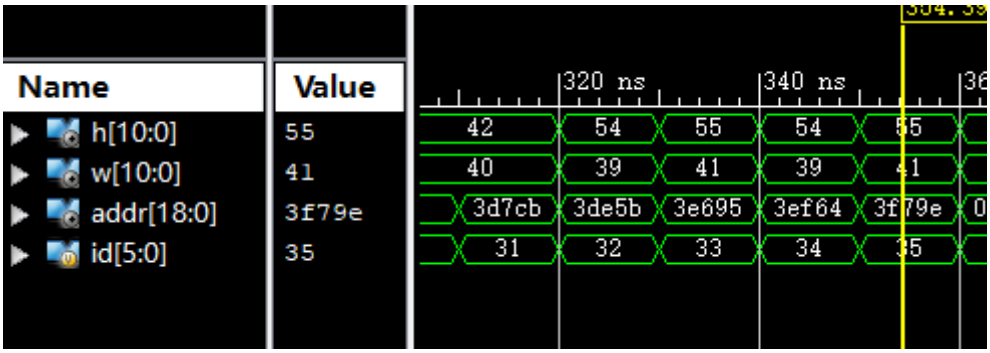
调试过程见所附带的视频。由视频的调试过程可见，该马里奥游戏与原版马里奥游戏的操作逻辑和交互结果近似相符，可以作为一个完整的游戏。

4 核心模块仿真

4.1 Object 模块测试

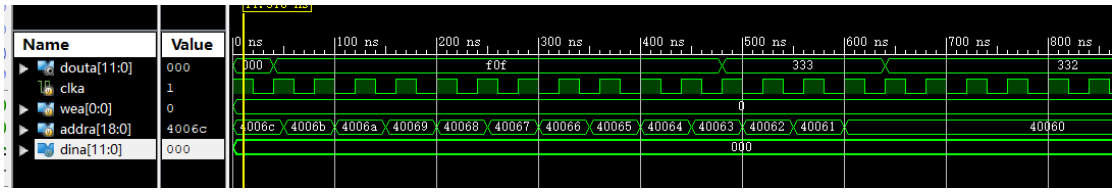






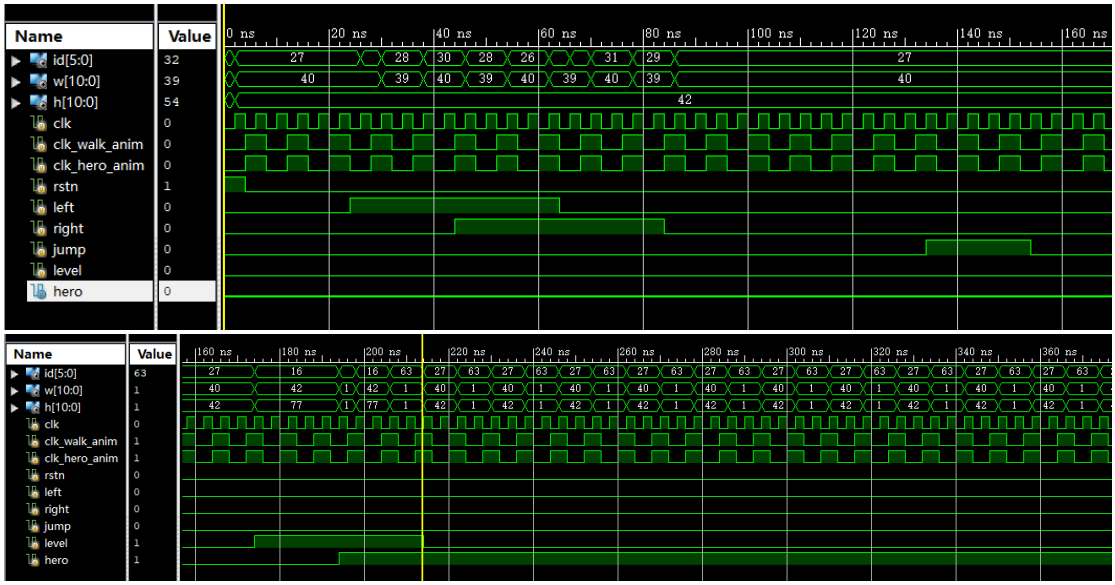
由以上测试情况可见，Object 模块工作正常，能正确地将素材的 ID 映射为内存地址，以及该素材的宽和高。该模块的正确性为游戏逻辑和交互界面提供了坚实的保障。

4.2 Image 模块测试



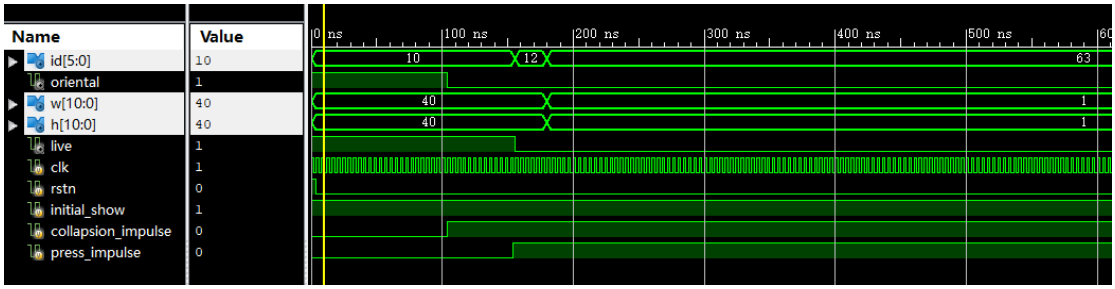
此仿真主要是为了测试 ISE 生成的 IP core 是否工作正确。此处节选了最后几个内存地址进行测试。对照 Image.coe 文件，可见该模块工作正常。

4.3 Mario 模块测试



以上仿真模拟了马里奥的各个操作，如左走、右走、跳跃，以及 World 模块内处理的升级、降级、无敌状态等。经测试，马里奥的输出素材 ID 与预期情况相符。

4.4 Goomba 模块测试



以上仿真模拟了小怪的碰撞操作、被踩操作。经测试，Goomba 的输出素材 ID 与预期情况相符，能正确地被踩、死亡。

4.5 Turtle 模块测试

以上仿真模拟了乌龟的碰撞操作、被踩操作。经测试，Turtle 的输出素材 ID 与预期情况相符，能正确地被踩、发狂、转向。

5 实验体会

本次大作业从十二月底开始做，直至 1 月 10 日左右完工，历经了近三周时间才完成了这一浩大的工程，其中遇到的 bug 更是千奇百怪，匪夷所思。最终居然能都成功解决，简直是一个奇迹。

实验中，我体会最深的一点是硬件描述语言与高级语言之间的巨大差异。硬件描述语言植根于底层的硬件，整个编程思路和高级语言差异巨大，因此大多数高级语言的编程理念都不再成立了，如 OOP、FP，甚至最基本的面向过程编程也基本失效。这次大作业也加深了我对计算机底层的理解，对计算机体系结构的了解有了一个全新的视角。

6 经验教训

这边列举几个编程过程中遇到的困扰我一段时间的问题。

1. 访问 Generate 语句块中的变量报错。之前，我是把 Generate 当做一个对象使用的，在里面新定义了一堆 reg 用于储存对象成员。但之后对 Generate 语句中的 reg 都报错了。最终我使用了不是很优雅的二维数组（最终映射为内存块）来储存数据。按理来说，Generate 语句块可以重复创建语句块中的对象，这与直接编写应该并无二致，出现的这个问题也令我困扰了一段时间。

2. 不同的 always 语句块修改同一个 reg 型变量时报错。这个原因和 always 语句的底层实现有关。不同的 always 语句块在综合时应该变成了不同的模块，因此驱动同一个 reg 变量时会出现错误。为了解决这一问题，我只好将大多数代码写进同一个时钟沿触发语句块中。这样造成的后果就是无法对块实现沿不同时钟触发的效果。最终我采取的办法是使用最快的时钟，在其中手动检测慢时钟是否处于上升沿或下降沿状态。

3. 时钟不同步问题。在输出 vga_data 时，我参照了 VGA demo 里的方法，使用 always@*来进行 vga 数据的输出。结果在下载验证时，屏幕总是出现一些花点，且花点数量随着时间的推移越来越多，最后甚至整个屏幕有一半是花的，视觉体验极差。这很明显是内存地址映射错误。最终我将 always@*修改为 always@(posedge clkdiv[1])，最终解决了这一问题。个人猜测的原因是 ISE 的优化策略导致了组合电路与时钟触发电路不同步，导致二者对应错误。

4. ISE 的赋值优化策略。以下是 turtle 模块中的一部分代码:

```
1. if (pre_collapsion != collapsion_impulse) begin
2.     oriental = ~oriental;
3. end
4.
5. pre_collapsion = collapsion_impulse;
```

这部分代码由时钟 clk 触发。在综合时，第五行的代码执行次序有几率在 if 的上方，这造成了 if 语句块中的内容无法被执行。最令人匪夷所思的是，倘若将第五行加在第二行后面，有一定几率使得 if 语句块内容在短时间内执行了两次。若加在第二行前面，问题得到解决。个人猜测原因应该和 ISE 的赋值优化策略有关，导致取反操作用时过长，pre_collapsion 无法及时赋值。