浙江大学

题目(中) 基于数字系统的 AI 五子棋游戏

(英) Gobang Game with AI

组长： 沈韬立 3170102588

组员： 漆翔宇 3170104557 夏豪诚 3170102492

指导教师： 王总辉

专业： 信息安全、计算机科学与技术

所在学院： 计算机科学与技术学院

提交日期： **2019 年 1 月 10 日**

目录

一、 游戏设计背景

1、 FPGA 简介

FPGA(Field Programmable Gate Array)，即现场可编程门阵列，是用户可以在现场对可定制的数字逻辑进行编程的集成电路。使用预建的逻辑块和可重新编程布线资源，用户无需再使用电路试验板或烙铁,就能配置这些芯片来实现自定义硬件功能。 用户在软件中开发数字计算任务，并将它们编译成配置文件或比特流，其中包含元器件相互连接的信息。此外，FPGA 可完全可重配置，当用户在重新编译不同的电路配置时，能够当即呈现全新的特性。

本次设计所用的硬件编程软件为 Xilinx 公司设计的 ISE 14.7 软件，所使用的语言为 Verilog HDL 硬件描述语言。本课程设计以行为描述为主，使用 VGA，PS2 外接设备丰富游戏。

2、 游戏介绍

五子棋是世界智力运动会竞技项目之一，是一种两人对弈的纯策略型棋类游戏，是世界智力运动会竞技项目之一。通常双方分别使用黑白两色的棋子，下在棋盘直线与横线的交叉点上，先形成 5 子连线者获胜。游戏开始后执黑字的先行下棋，白子紧随其后，交替下子，每次只能下一子。当黑白两方中某一方达成“五子连珠”时该方获胜，显示屏上会将游戏结果显示，双方无法再下子，直到第二局游戏开始。玩家每下一次，五子棋 AI 会对结果进行评估并且下出 AI 认为的最优解。

3、 设计介绍

本课程设计是对 Ai 五子棋对战的一种简化，，出于表现 AI 的水平的考虑将其设为先手，人类后手的模式，另一种模式是双人对战的模式。

本课程设计采用方向控制方向，空格键下子，Ctrl 键重新开始游戏，R 键改变游戏模式的操作方式。游戏界面仅以棋盘为背景，通过 VGA 显示，开发板上的七段数码管显示的是输入的键盘值。

4、 重难点分析

- VGA 显示与 PS2 键盘输入
- AI 模块的实现
- 数据的传输与存储

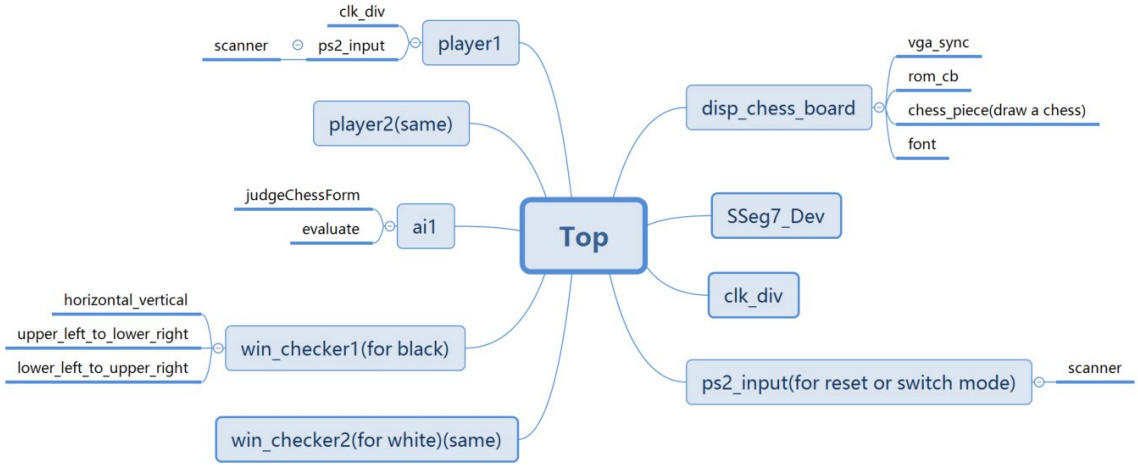
二、 整体结构

1、 输入输出

本游戏的输入全部通过 PS2 键盘输入，包括移动、下子、重启（reset）及切换模式。输出部分以 VGA 显示屏为主，显示屏上显示棋盘现状，板子上的七段数码管仅作测试用途，显示 PS2 键盘输入的值。

2、 模块层次

本课程设计通过 Top 模块综合各个模块，具体结构如下图所示。



3、 游戏过程

本课程设计采用方向控制方向，空格键下子，Ctrl 键重新开始游戏，S 键改变游戏模式的操作方式。游戏界面仅以棋盘为背景，通过 VGA 显示，开发板上的七段数码管显示的是输入的键盘值。

三、 模块介绍

1、 Top 顶层模块

Top 顶层模块主要起着统筹调用各个模块的作用，同时控制一些表示各个模块状态的寄存器使人类与人类/ai 能够异步下子。

Top 模块中直接调用的 PS2 模块主要是用于接收键盘的 Ctrl 复位信号与 R 切换信号，当 Ctrl 键被按下时，四位 whichkey 信号值变为 10，reset 信号随之改变。其他诸如 player 模块，ai 模块，display 模块，win_checker 模块将在后面详细解释。

Top 中有两个重要的寄存器 human_or_ai 和 is_player。前者用于区分先手的黑子是人类还是 ai，1 为人类，0 为 ai，后者表示黑方/白方正在下子。

human_or_ai 通过 PS2 键盘输入控制。Reset 时 human_or_ai 信号为 0，表示默认黑方为 ai。当 R 键被按下同时白方没有下子（游戏未开始），四位 whichkey 信号值变为 8，时钟上升沿触发 human_or_ai 发生翻转。

is_player 通过 player/ai 输出的 pressed 信号控制。reset 时 is_player 为 1，指黑方下子。当 pressed 信号的时钟上升沿到来时，is_player 信号发生翻转，白方可以下子。另外，当游戏处于双人模式时，由于选择的红框只显示一个，在 assign 连线时使用了三目运算符的方法进行选择。

2、 Player 模块

Player 模块调用了时钟分频模块与 PS2 键盘模块。choose_row 和 choose_col 是一个 4 位的寄存器，表示红框选择的位置，225 位的 disp 表示该方的棋盘。首先，该模块在输入的 is_player 为 1 时才会有效。异步 reset 时把棋盘清零，choose_row 和 choose_col 置为 7，即棋盘中央。当 PS2 键盘输入上下左右方向键的信号时，choose_row 和 choose_col 会发生改变，如按下左键时 choose_col 减一。当按下空格键下子时，首先确保此处没有自己的子和对手的子，下子成功后触发一个 pressed 信号，这个信号会在 top 中触发并翻转 is_player 的值，该模块失效，另一方的模块启动。由于 top 模块中检测的是 pressed 的上升沿，我们在 player 模块中手动创造出一个先上升后下降的脉冲。

Player 模块中还有一个 lastkey 寄存器用于解决 ps2 键盘输入过多的问题。在先前的版本中我们发现当键盘的上下左右被按下时，红框会立即移动到棋盘边缘，仔细研究后发现 ps2 键盘按下与弹起并不是一个信号，在按下时有一串连续的信号读入。因此我们加入 lastkey 记录上次的键盘信号，若本次的键盘信号与 lastkey 相同则无效。

3、 Win_checker 模块

Win_checker 即胜利判断模块，它拥有三个子模块 horizontal_vertical、upper_left_to_lower_right、lower_left_to_upper_right，分别是判断横竖是否已经连五，左上到右下的斜向是否已经连五，以及右上到左下是否已经连五。三个模块的行为模式类似，都是以最后下子的点为基点，检查该子所在行/列/斜向上的连子情况。由于本模块是组合逻辑模块，为简化计算我将所有可能连子的情况列出，便于电路计算，减小电路的负载。三个模块最大的区别在于行/列是可以直接计算的，斜向上比较复杂。需要列举出最后下子点可能连子的情况并对每个情况进行分析。

4、 Disp_chess_board 模块

disp_chess_board 即整个棋盘及其内容的显示模块，它调用了 vga_sync 和 chess_piece 两个子模块，以及两个 IP 核 rom_cb 和 font（类型为 memory element），用于提供 vga 限制所需的信息。vga_sync 的作用为输出 vga 显示的行扫描信号（sync_h）和帧扫描信号(sync_v)以及当前扫描位置是否属于显示区域（video_on）。chess_piece 则以一个棋盘线相交位置作为一个棋子的中心计算得到距离此中心为 15 像素及以内的像素点位置，传回 disp_chess_board 以画出棋子。若存在棋子则画出黑白棋子，若没有棋子则直接读取 RGB 值，画出存在 rom_cb 中背景图。当 human 或 AI 获胜后就进入结果显示界面，调用 font 也就是 8*16 点阵字符发生器来显示胜利信息“White Wins”或“Black Wins”。

5、 Ps2_input 模块

ps2_input 即 PS2 键盘输入模块，它调用了 ps2_scan 这个子模块。通过扫描键盘在按下不同按键时产生的不同的键盘通码来实现对按下的键盘按键的识别，并通过对这个信号的检测捕获可以给不同的按键信号赋予不同的功能。由于 PS2 键盘使用的

是串行输出方式，所以需要在 ps2_scan 中用 read_data 采用逐位接收的方式，并以 read_state 作为计数器。鉴于键盘传输的通断码的特点，我们以 8 位数据为单位进行判断，所以 read_data 仅设置为 8 位的长度。同时由于 PS2 键盘的通断码中存在不同类型的按键，第一类按键，通码为 1 字节，断码为 0xF0+通码 形式。如 A 键，其通码为 0x1C，断码为 0xF0 0x1C,第二类按键，通码为 2 字节 0xE0+0x?? 形式，断码为 0xE0+0xF0+0x?? 形式；如 right ctrl 键，其通码为 0xE0 0x14，断码为 0xE0 0xF0 0x14；第三类特殊按键有两个，print screen 键通码为 0xE0 0x12 0xE0 0x7C，断码为 0xE0 0xF0 0x7C 0xE0 0xF0 0x12; pause 键通码为 0xEl 0x14 0x77 0xEl 0xF0 0x14 0xF0 0x77，断码为空。所以在应用方向键的时候需要应用 is_e0 和 is_f0 来判断通断码。并当 is_e0 为 1，也就是接收到方向键的通码的时候在 8 位的 read_data 前加上一位 1，作为标记。在接收按键状态后，根据不同的按键赋予 whichkey 不同的值，便于在其他模块中的调用。同时为了能够通过键盘实现 reset 功能，该模块不进行 reset。

四、 调试与仿真

由于本课程设计采用 VGA，PS2 模块，有许多模块无法直接仿真，因此采用物理验证的方式（在实验结果中体现）。可以采用仿真的模块有胜利判断模块 win_checker 的三个子模块与 ai 模块。

1、 Win_checker 子模块仿真

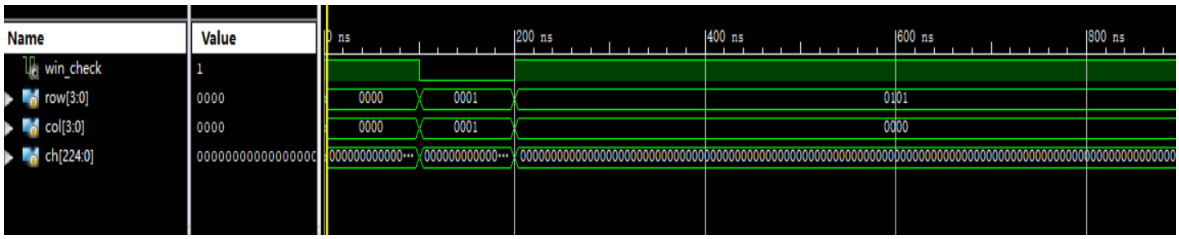
• horizontal_vertical 模块仿真

仿真代码：

```
module h_v_test;

    // Inputs
    reg [3:0] row;
    reg [3:0] col;
    reg [224:0] ch;

    // Outputs
    wire win_check;

    // Instantiate the Unit Under Test (UUT)
    horizontal_vertical uut (
        .row(row),
        .col(col),
        .ch(ch),
        .win_check(win_check)
    );

    initial begin
        row = 0;
        col = 0;
        ch = 225'b11111;
        #100;
       ch = 225'b0;
        ch[15]=1;
        ch[30]=1;
        ch[45]=1;
        ch[60]=1;
        row =1;
```

```
        col =1;
        #100;
        ch[75]=1;
        col = 0;
        row = 5;
    end

endmodule
```

仿真结果：



图表 1 horizontal_vertical **仿真结果**

由于 ch 位数过多，无法完全显示，但可以从代码中看出，当 0、1、2、3、4 位或 15、30、45、60、75 为 1 时，win_check 的值为 1。此处验证了横竖的胜利判断。

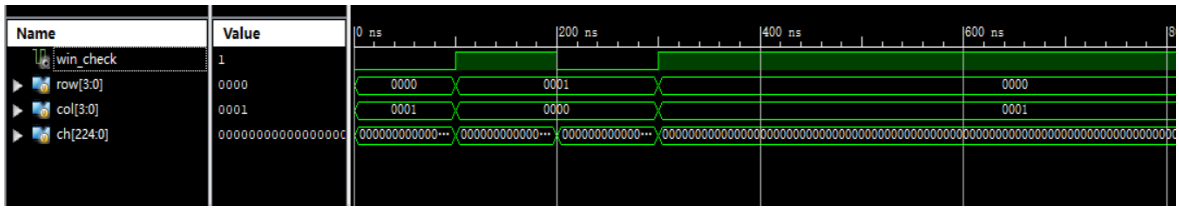
• upper_left_to_lower_right 模块仿真

仿真代码：

```
module u_l_l_r_test;

    // Inputs
    reg [3:0] row;
    reg [3:0] col;
    reg [224:0] ch;

    // Outputs
    wire win_check;

    // Instantiate the Unit Under Test (UUT)
    upper_left_to_lower_right uut (
        .row(row),
        .col(col),
        .ch(ch),
        .win_check(win_check)
    );

    initial begin
        row = 0;
        col = 1;
        ch =0;
        ch[1]=1;
        ch[31]=1;
        ch[47]=1;
        ch[63]=1;
        ch[79]=1;
        #100;
        row =1;
        col =0;
        ch[15]=1;
        #100;
        ch =0;
        ch[15]=1;
        ch[17]=1;
```

```
        ch[33]=1;
        ch[49]=1;
        ch[65]=1;
        #100;
        row =0;
        col =1;
        ch[1]=1;
    end

endmodule
```

仿真结果：



图表 2 upper_left_to_lower_right 仿真结果

可以从代码中看出，当 15、31、47、63、79 位或 1、17、33、49、65 位为 1 时，win_check 的值为 1。

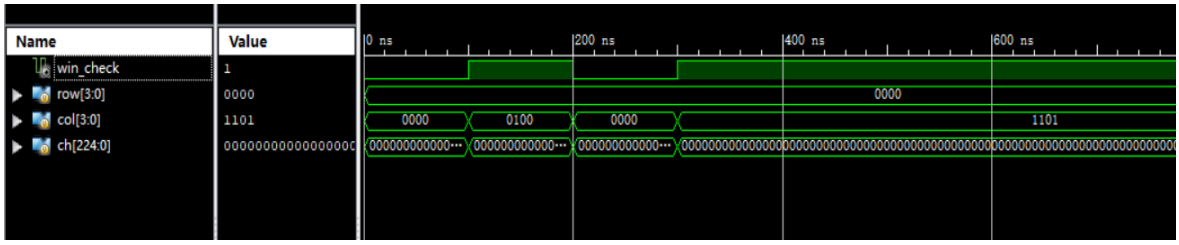
• lower_left_to_upper_right 模块仿真

仿真代码：

```
module l_l_u_r_test;

    // Inputs
    reg [3:0] row;
    reg [3:0] col;
    reg [224:0] ch;

    // Outputs
    wire win_check;

    // Instantiate the Unit Under Test (UUT)
    lower_left_to_upper_right uut (
        .row(row),
        .col(col),
        .ch(ch),
        .win_check(win_check)
    );

    initial begin
        row = 0;
        col = 0;
        ch = 0;
        ch[0]=1;
        ch[18]=1;
        ch[32]=1;
        ch[46]=1;
        ch[60]=1;
        #100;
        row =0;
        col = 4;
        ch[4]=1;
        #100;
        row = 0;
        col = 0;
```

```
        ch = 0;
        ch[0]=1;
        ch[27]=1;
        ch[41]=1;
        ch[55]=1;
        ch[69]=1;
        #100;
        row = 0;
        col = 13;
        ch[13]=1;
    end

endmodule
```

仿真结果：



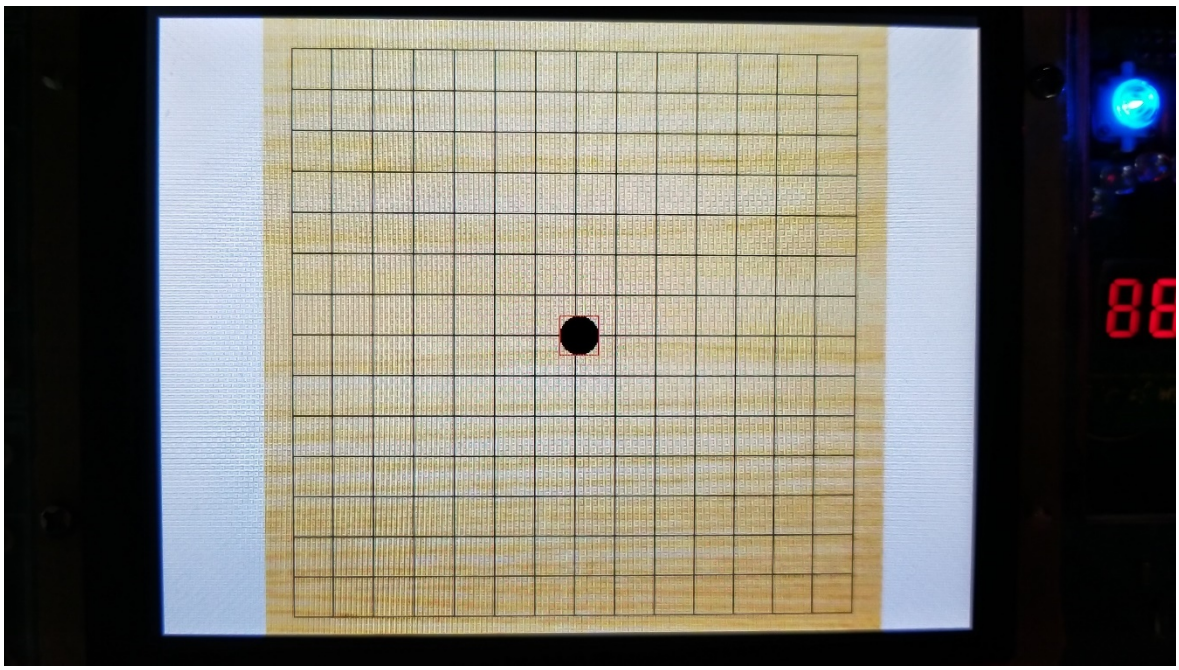
图表 3 lower_left_to_upper_right **仿真结果**

可以从代码中看出，当 4、18、32、46、60 位或 13、27、41、55、69 位为 1 时，win_chek 的值为 1。

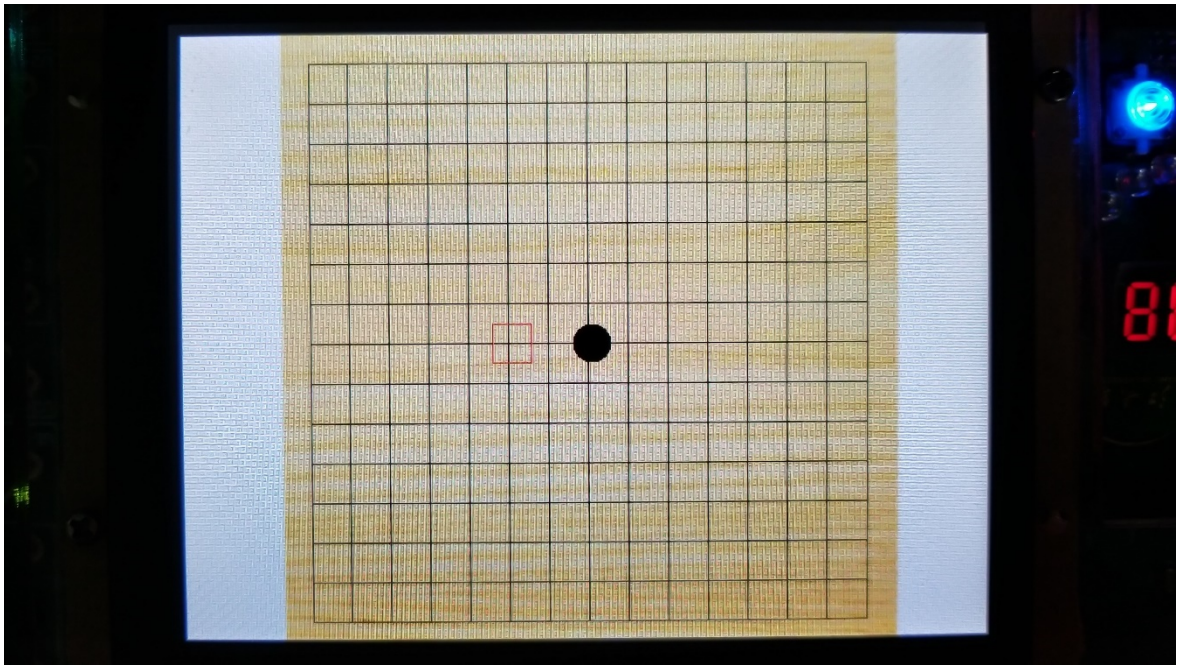
五、 游戏演示

1. 基本操作

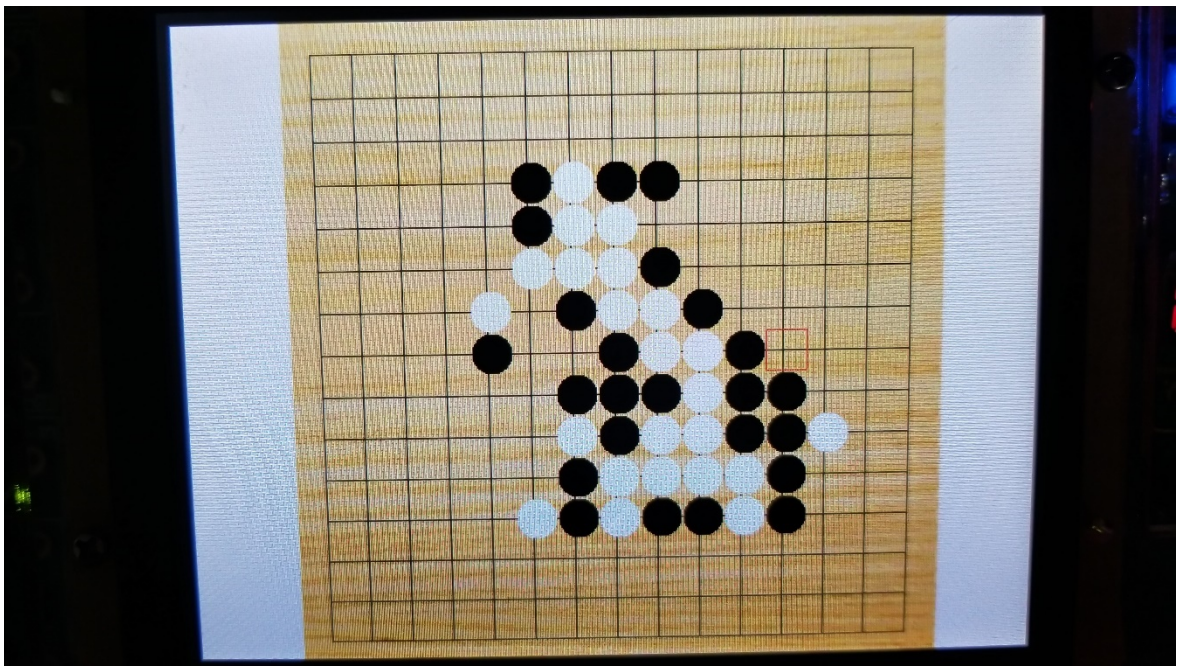
1. 将工程文件生成 bit 文件烧录至 SWORD 板由此进入开始界面，可以看到，AI 执黑先行落在棋盘中心，红色矩形表示当前选中位置。



2. 通过键盘的方向键可以控制表示选中位置的红色矩形的移动。

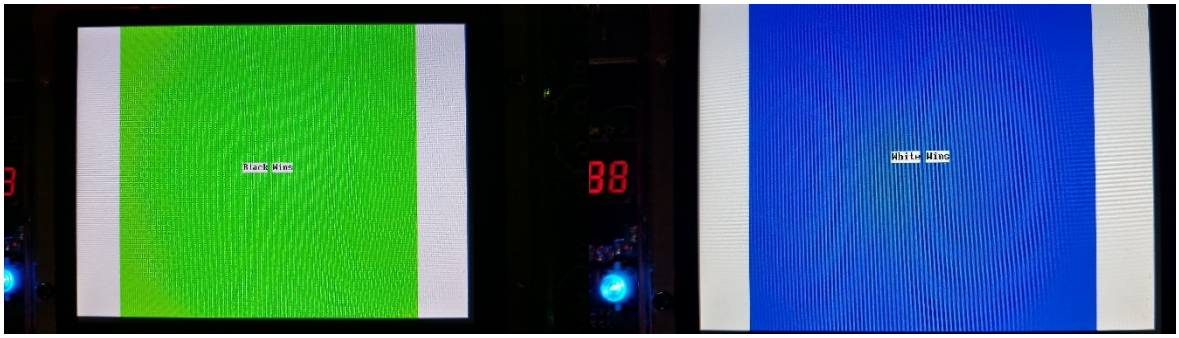


3. 按下空格键落白子，AI 迅速落黑，循环交替，直至一方获胜。



2. 胜利结算界面

规则为只要黑白方中任意一方的棋子有五子连在一起（即五子连珠），可以为横连、纵连、斜连，则该方获胜，游戏结束。当执黑的 AI 获胜时，会自动进入显示“Black Wins”的界面，反之，当执白的人类获胜时会显示“White Wins”。而此时按下“Ctrl”键即可重启游戏。



3. 其他显示

在游戏中一旦按下对应的键盘按键即会在 SWORD 板下方的 LED 数码管显示该按键在模块中对应的 16 进制数值。下方为按下空格键（对应值为 5）时的显示，没有任何键被按下时的显示为 0。

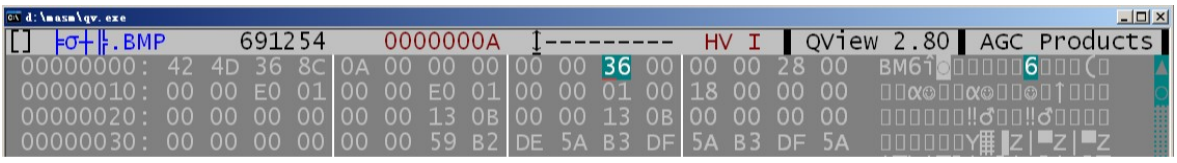
附：背景图制作过程

我们通过 Photoshop 绘制背景棋盘，导出 JPG 图片，其格式如下：

棋盘整体规格	**480像素*480像素**
行列数	**15**
线宽	**1像素**
方格大小（不计线宽）	**30像素**
边界线外像素数	**上23；下22；左22；右23**

采用这样的格式可以便于显示所需要的棋子中心位置的寻找。

通过格式工厂转换为 bmp 文件，以二进制文件格式打开,文件偏移+0AH 处即为图像数据的文件偏移为 36H；



编写 c 语言程序 transfer.c 生成 transfer.exe 实现功能，二进制格式读入图像文件，读取每个 24 位图的像素点颜色数据，转换为 12 位的颜色数据，写入到文件 "cdb.txt"。

将得到的数据再次处理后，添加文件头，并将后缀修改为 coe，得到:

```
//文件头
memory_initialization_radix = 16;

memory_initialization_vector =
```

transfer.c 的代码如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int OutImport(FILE *in,FILE *out);
int main(int argc,char **argv)
{
    if(argc > 1 && strcmp(argv[2],"--help") == 0)
    {
        printf("\nusage: linkrules_god  inputfile outputfile\n");
```

```
            printf("    encrypt inputfile to outputfile.\n");
            exit(0);
        }
        if( argc !=3)
        {
            fprintf(stderr,"%s\n","please using --help go get help.");
            exit(-1);
        }
         //attention: open in binary
        FILE *in = fopen(argv[1],"rb");
        FILE *out = fopen(argv[2],"w+");
        if(in == NULL || out == NULL)
        {
            fprintf(stderr,"%s\n","open file error!");
            exit(1);
        }
         OutImport(in,out);
        fclose(in);
        fclose(out);
        return 0;
    }
    int OutImport(FILE *in,FILE *out){
        int F0,F1;
        unsigned char hex[4];
        int count=0;
        fseek(in,0L,0);
        fread(&hex[0],1,1,in); F0 = hex[0];

        head = F0 ;

        fseek(in,(long)head,0);

while(fread(&hex[0],1,1,in)&&fread(&hex[1],1,1,tmp)&&fread(&hex[2],1,1,in
)){
                hex[0] = hex[0]/16;
                hex[1] = hex[1]/16;
                hex[2] = hex[2]/16;
               fprintf(out,"%1x%1x%1x",hex[2],hex[1],hex[0]);
                fprintf(out,"\n");
        }
        return 0;

    }
```

ISE 载入 coe 文件 生成 IP 核。

六、 分工说明

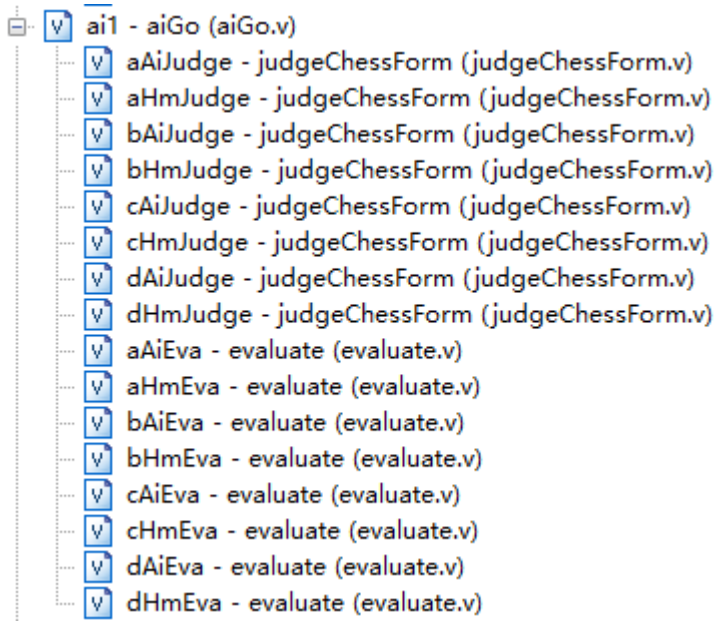
沈韬立（组长）：负责游戏基础逻辑、top 模块的编程及相关文档的撰写。（1/3）

漆翔宇：负责 ai 模块的编程及相关文档的撰写。（1/3）

夏豪诚：负责 VGA、PS2 模块的编程及相关文档的撰写。（1/3）

附：AI 模块设计介绍

一． 模块基本组成



如图，整个 AI 部分总共有三个源文件，分别为 aiGo.v，judgeChessForm.v，evaluate.v。aiGo.v 实现了整个棋盘枚举和最优化筛选的逻辑。judgeChessForm.v 实现了棋形的判断。evaluate.v 实现了对输入棋形编号返回估值的功能。

ai 模块下棋形判断器和估值器分别有八个。之所以是八个，是因为对于每一个落子点，有四个方向需要判断棋形并估值，并且要分别对人类和 AI 在这里落子都估值。

aiGo 模块由时序驱动，在时钟上升沿转移状态，完成对棋盘的枚举和最优化筛选。棋形判断和估值模块都是组合电路。aiGO 模块在枚举的时候，会实时的将枚举位置四周的棋子信息更新到判断器和估值器的输入信号中。aiGo 模块在最优化判断的时候，从估值器的输出信号中获得此处的估值。

二． 贪心算法实现五子棋 AI

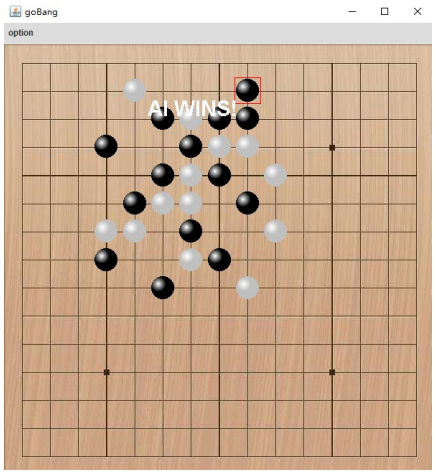
1. 评估函数：

为当前盘面每一个落子点的收益进行评估。对于每一个落子点，我们分别对其四个方向上形成的棋形进行判定。我们预先为每种棋形设定了预置的估值。评估函数就是对四个方向做棋形判定，查表获得估值，然后将四个方向的估值线性和作为这个落子点的估值。

对于棋形判断来说，棋形的种类之于一个方向上落子点两侧的四个棋子有关。所以对于每个方向上我们只需要考虑 9 个位置的棋子。（与落子点距离超过 4 的棋子不考虑）

显然这是一个贪心策略，只关心当前这一步的最优，甚至只关心周围很小范围的一块棋形。我们只需要按着棋形那里描述的几个状态做决策就行了。这是一个非常粗糙的最优化思路。棋类游戏的精髓，其一为大局观，其二为远见性。这个算法既没有

大局观，也没有远见性。然而由于五子棋的规则非常简单，所以这样的算法在对弈上的表现有着较强的智能性和实力，常常能将我们的进攻精确的化解，还能出其不意的战胜我们。除非我们提前做好巧妙的布局，否则战胜它有一定的难度。



图为在 Java 平台该算法的实现，AI 战胜了人类。可以看到虽然它并没有很强的大局观和远见性，但是却总是能最好的做到防御中进攻，进攻中防御，这是因为我们算法中有这样的一个机制，具体会在下面的贪心中策略中介绍。

2. 贪心策略：

(1) 枚举下棋点，找出使我方势力值最大的点和使地方势力值最大的点。寻找我方势力值最大点时，如果出现多个点都可以使得我方势力最大，则选择那个可以使得对方势力最大的点。（一个点，如果自己能获得最大的效益，对手如果在这个点落子也能获得最大效益，那我在这里落子显然是占优的）

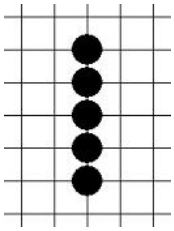
(2) 在第一步中求得了我方和对方最大势力点。在我方最大势力点落子可以让我方势力值不减（大概率下都是会增加的），对方势力值不增（可能会减，比如说我下的点恰好堵住了对方的某个棋形）。在对方最大势力点落子可以让对方势力值不增（大概率下都是下降的，至少减少了对方在未来的增加），可以让自己的势力值不减（可能恰好也让自己有了连珠，增加了自己的势力值）。在经验上来看，下我方最大势力点对应进攻，下对方最大势力点对应防守。进攻还是防守？我们简单的根据这两个最大势力点对应的势力值来判断，如果我下了我的最大势力点，对方下了他的最大势力点，我的势力值比对方高，那我选择进攻，否则我选择防守。（如果对方的杀招比较厉害，我先防守，如果我的杀招比较厉害，我则攻）

3. 棋形与估值

我们在网上查阅资料获得了五子棋棋形的基本棋谱，并基于将这份棋形移植到了我们的算法实现中。我们总共整理了八种棋形，分别编号为 0~7，并为每一种棋形都赋予了估值。

（1）连五

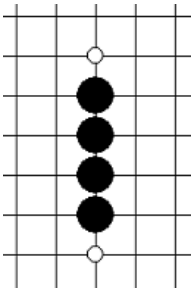
五子连珠，游戏胜利。棋形编号 7，估值2^{24}。



（2）活四

活四被定义为四子连珠，且有两个落子点可以让它变为连五。

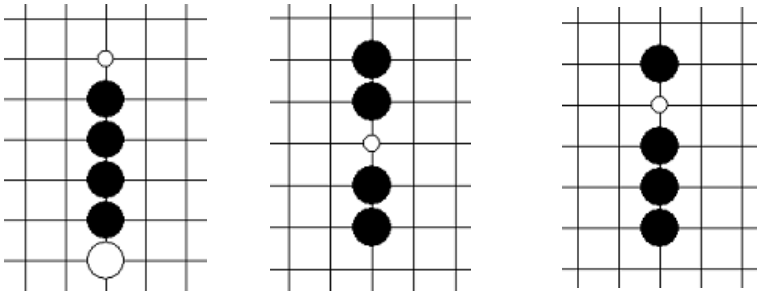
棋形编号 6，估值2^{19}。



（3）冲四

冲四定义为四子连珠，只有一个落子点可以让它变为连五。

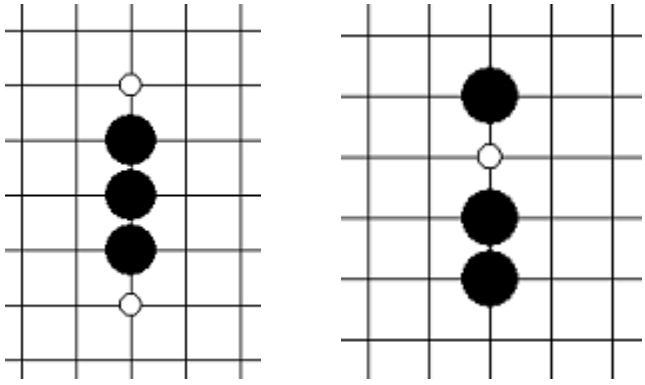
棋形编号 5，估值2^{16}



（4）活三

活三定义为三子连珠，并且存在落子点让它变为活四。

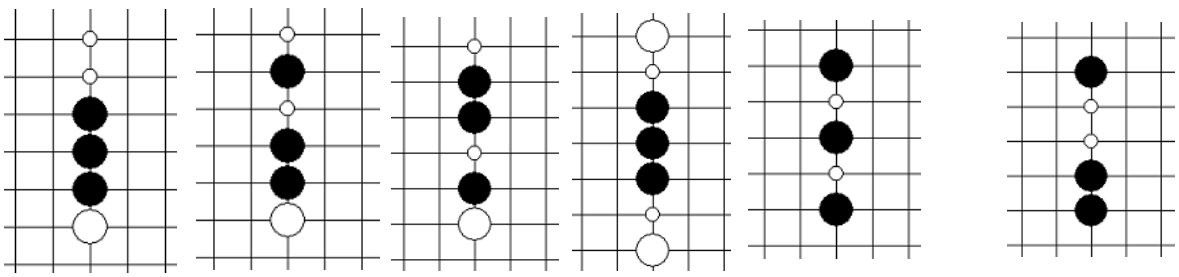
棋形编号 4，估值2^{15}



（5）眠三

眠三定义为三子连珠，并且存在落子点让它变为冲四，但不存在落子点让它变换活四。

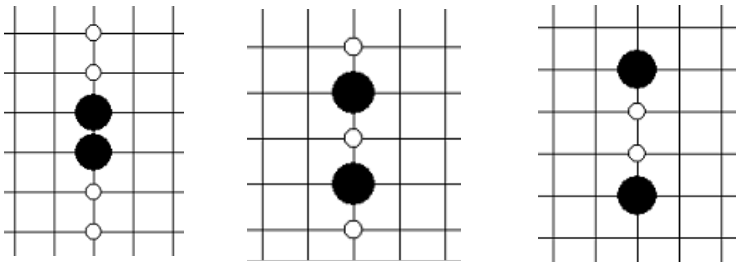
棋形编号 3，估值2^{14}



（6）活二

活二定义为二子连珠，并且存在落子点让它变为活三。

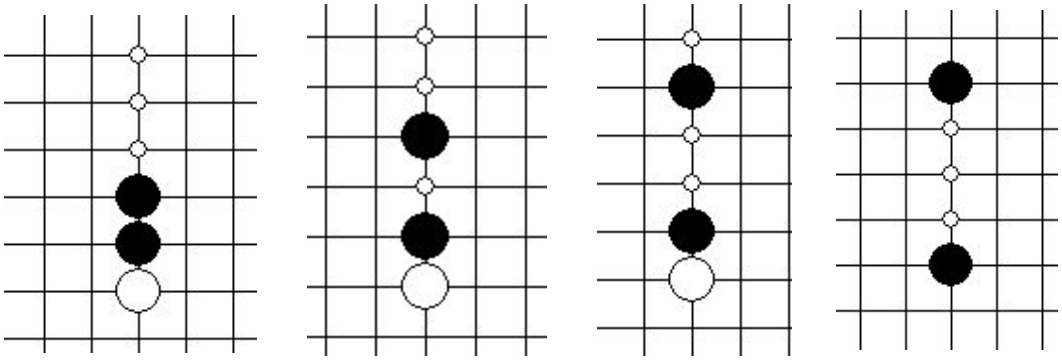
棋形编号 2，估值2^{12}



（7）眠二

眠二定义为二子连珠，并且存在落子点变为眠三，但不存在落子点变为活三。

棋形编号为 1，估值2^{11}



（8）其他

棋形编号为 0，估值 0

棋形分类和估值的设计中有两个值得注意的地方。

其一是像连五，活四这样的杀招的估值需要被设得足够大。可以看到，连五是活四估值的 32 倍，活四是冲四的 8 倍。这样的设计是避免了四个方向都是活四，叠加起来居然连 5 大这样的情况。杀招一定要有绝对的优势，否则不仅容易错失胜机，也有可能会贪图小利而被对方击败。而对于非杀招，我们一般相邻两个等级就只有两倍的差距，这样可以鼓励多创造小机会，不要吊死在一棵树上。

其二是，可以看到整个棋形的定义是一个递归定义。所以棋形的判断很适合设计成状态机或者递归的形式。我们在 FPGA 平台上实现之前，在 Java 平台上的预实现也确实是用递归的方法实现的棋形判定。但是我们最后在 FPGA 上采用了枚举的方式做判断，而不是用递归来简化，原因会在下一个部分提到。

三． FPGA 上实现

1. 顶层 AI 模块的功能和行为

```
//开启八个棋形判断器，分别判断AI和人类四个方向的棋形
judgeChessForm aAiJudge(aAi,aHmt,aTypeAi);
judgeChessForm aHmJudge(aHm,aAit,aTypeHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
judgeChessForm bAiJudge(bAi,bHmt,bTypeAi);
judgeChessForm bHmJudge(bHm,bAit,bTypeHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
judgeChessForm cAiJudge(cAi,cHmt,cTypeAi);
judgeChessForm cHmJudge(cHm,cAit,cTypeHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
judgeChessForm dAiJudge(dAi,dHmt,dTypeAi);
judgeChessForm dHmJudge(dHm,dAit,dTypeHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//开启八个估值器，接受AI/人类四个方向的棋形，给出估值
evaluate aAiEva(aTypeAi,aValueAi);
evaluate aHmEva(aTypeHm,aValueHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
evaluate bAiEva(bTypeAi,bValueAi);
evaluate bHmEva(bTypeHm,bValueHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
evaluate cAiEva(cTypeAi,cValueAi);
evaluate cHmEva(cTypeHm,cValueHm);
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
evaluate dAiEva(dTypeAi,dValueAi);
evaluate dHmEva(dTypeHm,dValueHm);
```

如图为顶层的 aiGo 模块驱动的 16 个子模块。分别是 8 个棋形判断器和 8 个估值器。顶层的 aiGo 模块向判断器输入棋盘信息，判断器输出的棋形编号信号接入估值器的输入。最后 aiGo 从估值器的输出获得最终得估值。

顶层的 AI 模块就干两件事情。在时序的驱动下，改变状态，不断的切换枚举的落子点，同时在输出状态和输入状态之间切换。每一次切换落子点后，将落子点四个方向的棋子信息传入棋形判断器。在这之后，顶层模块在下一次时钟上升沿进入输入模式，这一模式下，从估值器取得最新的估值，这对应于新的枚举点处的估值。顶层模块会计算四个方向估值和，并与最大值比较，做最优化筛选。然后又进入输出状态，移动枚举点，并且又改变对棋形判断器的输入。

```
//AI模块由时序同步触发或者rest信号异步触发
/*
    AI模块会枚举棋盘上15*15=255个位置。
    我们把整个枚举过程分解到若干个时钟周期钟去做。
    可以理解为一个状态机。
    整个状态机总共有510个状态。
    我们用target和state共同表示。
    其中target用来枚举255个位置。
    state=0的时候，修改输入到棋形判断器的棋子信息。
    state=1的时候，从估值器中取出估值，做最优化筛选。
*/
```

整个状态的切换如上所述。

当枚举完成，AI 顶层模块会通过分析人类和 AI 分别取得的最大值，决定是进攻还是防守，最后选择自己的落子点。落子的行为具体到底层就是把记录 AI 棋盘信息的寄存器组对应的位修改为 1。AI 顶层会实时的输出棋盘信息的信号。VGA 显示模块会实时的将寄存器组信息反应为图像。

```
begin
   if(state==0)//state==0,修改输入估值器的棋子信息
      begin
         if(target==225)//枚举结束,进入终止状态
            begin
               if(maxAi>maxHm)//判断进攻还是防守，选择落子点
                  begin
                     AI[aiPos]<=1;
                     x<=aiPos/15;
                     y<=aiPos%15;
                  end
               else
                  begin
                     AI[humanPos]<=1;
                     x<=humanPos/15;
                     y<=humanPos%15;
                  end
                isFinished<=1;
                target<=0;
                //结束AI's turn,重置枚举量,发出结束信号
            end
```

如图，枚举结束后，表征枚举位置的 target 寄存器被赋值为 0，回到原点。同时表征枚举结束的 isFinished 寄存器被置为 1。外部的顶层逻辑获得这个信号后，会停止 AI 模块状态的跳变，进入人类行为状态。

2. 修改对棋形判断器的输入

```
/*
   下面开始更新四个方向的棋子信息
*/
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  从左到右方向的棋子信息
   for(i=-4;i<=4;i=i+1)
   if(col+i<0||col+i>14)
      begin
         aHmt[i+4]<=1;
         aAit[i+4]<=1;
         aHm[i+4]<=0;
         aAi[i+4]<=0;
      end
   else
      begin
         if(i==0)
            begin
               aHm[i+4]<=1;
               aAi[i+4]<=1;
               aHmt[i+4]<=0;
               aAit[i+4]<=0;
            end
         else
            begin
               aHm[i+4]<=humanIn[target+i];
               aHmt[i+4]<=humanIn[target+i];
               aAi[i+4]<=AI[target+i];
               aAit[i+4]<=AI[target+i];
            end
      end
```

上图展示了左右水平方向上输入棋形判断器的棋盘状态的修改。

3. 最优化筛选

```
else    //state==1
   begin//AA
      if(empty==1) //如果枚举点为空,获取估值信息,更新最优点
            begin
               sumAi=aValueAi+bValueAi+cValueAi+dValueAi;
               sumHm=aValueHm+bValueHm+cValueHm+dValueHm;
               //分别将四个方向的估值相加,这里必须要用阻塞赋值
               if(firstValid) //如果是第一次找到有效的枚举点,最优值就是当前枚举点
                  begin
                     maxAi<=sumAi;
                     maxAiForHm<=sumAi;
                     maxHm<=sumHm;
                     maxHmForAi<=sumHm;
                     aiPos<=target-1;
                     humanPos<=target-1;
                     firstValid<=0;
                  end
               else  //否则和之前的最优值比较,更新
                  begin
                     if(sumAi>maxAi||(sumAi==maxAi&&sumHm>maxHmForAi))
                        begin
                           maxAi<=sumAi;
                           maxHmForAi<=sumHm;
                           aiPos<=target-1;
                        end
                     if(sumHm>maxHm||(sumHm==maxHm&&sumAi>maxAiForHm))
                        begin
                           maxHm<=sumHm;
                           maxAiForHm<=sumAi;
                           humanPos<=target-1;
                        end
                  end
            end

            state<=~state;
   end//AA
```

上图展示了与最大值比较做最优化筛选的逻辑。

4. FPGA 上 AI 顶层模块行为的特殊性

（1）调用估值函数与接受返回值的分离

前面提到了，在 FPGA 平台上实现和在常规的程序语言平台上实现有很大的不同。在 Java 平台上实现，对于落子点枚举，只需要用一个 for 循环语句遍历每一个位置，对每个位置做估值，然后立刻与最优值比较。整个循环结束后，最优值也出现了。

verilog 虽然提供了 for 循环，但这里我们不能这样做。原因是 Java 中调用估值函数模块时，顶层的循环逻辑是被阻塞的。抽象表现上，串行 Java 程序实际上是在执行一个命令序列，任意时刻，只有一个命令在被执行。但是 FPGA 上对应的却是一堆电路块，我们调用估值不是在调用一个函数或者说指令序列的插入，而仅仅是改变了对一个电路模块的信号输入。要从输出接收到响应，会有延迟存在。所以我们必须把调用估值模块和接收估值模块返回分开到不同的时钟周期里。所以整个 AI 顶层模块是一个状态不断跳变的状态机。

也正是因为这样的设计，我们直接把棋形判断和估值全部设计为组合逻辑。

（2）判断器与估值器的组合逻辑化

驱动我们的 AI 顶层模块的时钟信号来自于顶层结构中分频器提供的一个频率信号。而棋形判断和估值器是被 AI 顶层模块驱动的。

如果要用时序驱动它们，显然时钟信号应该比驱动 AI 顶层模块的时钟信号快，这就意味着，我们还要再放一个更快的时钟输入 AI 顶层模块，这样非常的不优雅，显得很蠢。同时还要考虑为判断器和估值器设定一个合适的时钟频率，既不快到一个周期内不能完成计算任务，也不慢到不能及时把结果输出到顶层模块。

这样的设计完全是自找麻烦。我们直接把棋形判断和估值设计成组合逻辑，就可以把上面的所有问题全部规避掉。然后只需要给顶层 AI 模块设计一个合适的时钟频率，让它慢到一个周期内组合逻辑能完成响应即可。简单而有效。

（3）考虑赋值的阻塞

```
if(empty==1) //如果枚举点为空,获取估值信息,更新最优点
      begin
         sumAi=aValueAi+bValueAi+cValueAi+dValueAi;
         sumHm=aValueHm+bValueHm+cValueHm+dValueHm;
         //分别将四个方向的估值相加,这里必须要用阻塞赋值
```

这是上面设计最优化筛选逻辑中的一个例子。这里对寄存器的赋值必须用“ = ”而不是“ <= ”。如果这里不阻塞，那么对着两个寄存器的改变和下面的最大最小的判断在电路的逻辑实现上就会是同一层的。这个最优化判定就无法进行。我们一开始这里的赋值上没有注意阻塞问题，导致了 AI 的表现非常的奇怪。一度怀疑是更加复杂的棋形判断模块出了问题，花了很多时间在错误的方向上。

最后我们一个周期一个周期的观察了波形图，最后终于在波形图的异常表现中发现了最优化筛选模块总是未能正常的运行，最后在几百行的 AI 模块代码中锁定了这两行。

5. 棋形判断与估值

我们在第二部分说过，我们的棋形判断是用枚举的方法实现的，原因在上面谈 FPGA 平台的特殊性与设计分析时已经谈过了，我们的棋形判断模块设计为组合逻辑。这注定了我们无法通过实现状态机来模拟递归的栈过程。

这个枚举，完全是基于第二部分展示的棋谱展开的。由于枚举过程非常的繁琐，所以这里不再详细展开介绍。

估值模块非常简单，获得棋形判断传来的棋形编号后，对输出的估值寄存器进行赋值维护就行了，赋值量完全是基于第二部分我们谈到的预设估值量进行的。