Gomoku Design Document

Hongyu Sun, Yunzhou Li, Zhiwei Xie, Zixuan Fang

Introduction

Gomoku, also called Five in a Row, is an abstract strategy board game. Two players take turns placing a stone of their color on an empty intersection. Black plays first. The winner is the first player to form an unbroken line of five stones of their color horizontally, vertically, or diagonally. In this project, we plan to implement a Gomoku game on DE1-SoC. The main function will include: Players vs Players mode, Players vs AI mode. The user interface and the display section will be delicately designed, and the game will be controlled by game controllers instead of a traditional keyboard.

Hardware

Display

Basic Colors and Synchronization Signals

We use the VGA interface to transmit image data, mainly utilizing pins 1 to 3 on the VGA interface for generating basic colors, representing R, G, B respectively. Additionally, pins 13 and 14 are used for horizontal and vertical synchronization signals.

Since the chessboard is green, and the pieces are black and white, we need at least three colors. Assuming disp_rgb is a 12-bit output signal used for controlling RGB, with 4 bits per color, the parameters for each color are:

```
Unset
output[11:0] disp_rgb;

// Black
assign disp_rgb = 12'h000; // RGB: 0000 0000 0000
```

```
// White
assign disp_rgb = 12'hFFF; // RGB: 1111 1111 1111

// Green
assign disp_rgb = 12'h0F0; // RGB: 0000 1111 0000
```

VGA Controller Design

To send the correct timing signals and generate the corresponding pixel color values based on the content of the image to be displayed, a VGA controller must be designed on the FPGA. Incorrect timing design can lead to random generation of the chess pieces' colors during gameplay, which is uncontrollable.

Scanning Mechanism

This code section mainly includes the following aspects:

- Horizontal scanning (Row scanning)
  - This part of the code is responsible for the counting of row scanning.
- Vertical scanning (Frame scanning)
  - Each time a row ends, the vertical counter is incremented. When the vertical counter reaches the last row, it also resets to 0, starting a new frame.

Synchronization Signal Processing

To accurately and synchronously draw the chessboard and pieces of a Gomoku game on a VGA display according to the game's state and rules, we must generate horizontal and vertical synchronization signals based on the counters for Horizontal scanning and Vertical scanning. The former controls the screen's row scanning, ensuring each pixel row is drawn in sequence to form a complete image. The latter indicates the start of each new frame, ensuring each frame of the game is drawn and displayed at the correct moment. They provide a stable and continuous foundation for the visualization of the game state.

Unset

```
// Initialization
```

```
Set hcount to 0
Set vcount to 0

// Horizontal scanning
always@(posedge vga_clk) begin
     #On each rising edge of the VGA clock
     If hcount reaches the end of a line:
          Reset hcount to 0
     Else:
          Increment hcount by 1

// Vertical scanning
always@(posedge vga_clk) begin
     #On each rising edge of the VGA clock
     If hcount was reset (i.e., one line scan is completed):
          If vcount reaches the last line:
               Reset vcount to 0
          Else:
               Increment vcount by 1

// Synchronization Signals
assign hsync = (hcount > hsync_end);
assign vsync = (vcount > vsync_end);
```

Display Game Logic and State

Once the synchronization signals are functioning correctly, the screen can display the game state updated by the game logic in real time. For instance, if the game is not over, it combines multiple conditions with the logical AND operator to determine whether the current pixel should be displayed, such as which pixels need to be lit based on the positions of the chessboard and pieces. If the game ends, another screen is displayed, for example, showing who the winner is.

Unset
```
// Display Game Logic and State
always @(posedge vga_clk) begin
     if GAME is not over
```

```
        # Determine the chessboard area and generate corresponding
    colors
        ...
        data <= Calculate and display color blocks
    else
        data <= Display another screen
    ...
end
```

Parameters and Variable Definition

Also, the Display module will include some basic parameters and variables needed for the chessboard and pieces. This part of the code includes:

- Defining the dimensions of each square on the chessboard
- Defining the boundaries of the entire display area on the VGA screen
- Drawing the boundaries of the chessboard
- Drawing the black lines between squares
- The cursor for the current piece position

Input

Device description

The Input device we choose is a Xbox one controller (Figure 1).

Figure 1

This controller has many buttons which will greatly satisfy our demands, we will use the direction buttons on the left and ABXY buttons as “confirm/back” buttons on the right of the controller.

To use this controller in our soc, we first need to install a Linux Kernel Driver, in this project we choose Xpad, an open source Driver.

We can also use the controller as part of the output. The controller will vibrate when:

- It is the player's turn to place pieces.
- The game finishes

Input Steps

The input steps are as follows:

1. Driver waits for controller inputs, program waits for device events
2. Controller sends inputs, driver receives and converts them to events
3. Program receives and parses the events to different commands
4. Program calls relevant functions according to commands, e.g.: moving current cursor, placing pieces, etc.

Input Commands

Input-command conversions are described in the following table.

Device input	Command

LJOYSTICK_UP / ARROW_UP_PRESS	Cursor move up
LJOYSTICK_DOWN / ARROW_DOWN_PRESS	Cursor move down
LJOYSTICK_LEFT / ARROW_LEFT_PRESS	Cursor move left
LJOYSTICK_RIGHT / ARROW_RIGHT_PRESS	Cursor move right
BUTTON_A_PRESS	Place piece at current cursor location / Confirm
BUTTON_B_PRESS	Cancel / Go back

We may add more commands during implementation.

Software

Game logic Implementation

The basic game logic and AI algorithm is written in C++. In this section, some of the key classes which are important for implementing the game and the AI are introduced.

class Gomoku

The Gomoku class is the backbone of the game logic in this Gomoku implementation, which encapsulates the essential components and operations needed by the game. This includes (but not limited to):

Board Validity Checks: Methods like on_board and valid_move that checks the validity of a move based on the rule of Gomoku.

Game Progression: The make_move method updates the game state with each new move made by the player (or the AI).

Win Conditions: check_win is a critical aspect of the class, which verifies if either player has met the winning condition—forming a contiguous line of five stones.

Game Status Updates: Functions such as is_draw and switchPlayers manage the game's progress, determining draw conditions and alternating turns between players.

Board Display: The displayBoard is for visualization, called when simulating an actual game between player and computer or between players.

Directional Check: This is a template, with its x_step and y_step parameters, and it allows for checking the board in various directions for a winning line, showcasing the game's attention to the intricate winning conditions beyond simple horizontal or vertical alignments. This template is called by check_win.

class gomokuAI

The GomokuAI encapsulates some of the key components of implementing the AI algorithms. Some of them are introduced as follows.

Game Link: GomokuAI holds the instance of Gomoku class, which is important when implementing a real game.

Strategic Knowledge: a table, shapeTable, is set with every valid shape and pattern and their corresponding scores. Through this table, the AI can effectively evaluate positions and make decisions.

Move Generation: The getLegalMoves methods provide the AI with a list of all valid moves on the board. It calls getLegalMoves

Evaluation Functions: The evaluate functions allow the AI to assess the current board state from the perspective of either player, which is essential for Minimax decision-making.

Position Rating: With ratePos, the AI can determine the strategic value of placing a stone at a particular position, taking into account the potential for creating winning shapes or blocking the opponent.

Move Making: makeMove and undoMove enable the AI to simulate moves on the board, which is used for exploring different game states during the Minimax process.

Best Move Calculation: findBestMove employs the Minimax algorithm with Alpha-Beta pruning to identify the optimal move based on the current game state.

AI Algorithm

In this section, the detailed AI algorithm is introduced.

Introduction to Decision Tree

A *decision tree* is a graphical representation of decision-making processes, illustrating various outcomes based on a series of choices. Technically, it is a tree-like model of decisions and their possible consequences, used to create a plan to reach a goal or make a decision. In the context of computing and AI, decision trees are structured as nodes and branches: nodes represent the points where decisions are made, and branches represent the possible choices leading to different outcomes.

In game scenarios like Gomoku(Figure 2), decision trees simulate the game board's state space. Each node(M0 to M5) is a possible *state*, which represents a board configuration, and each edge denotes a player's move,which leads to a different state. The root of the tree reflects the *current* state of the game, and the branches represent potential future moves. The leaves of the tree correspond to *end states* of the game, which could be a win, loss, or draw, depending on the sequence of moves represented by the path from root to leaf.

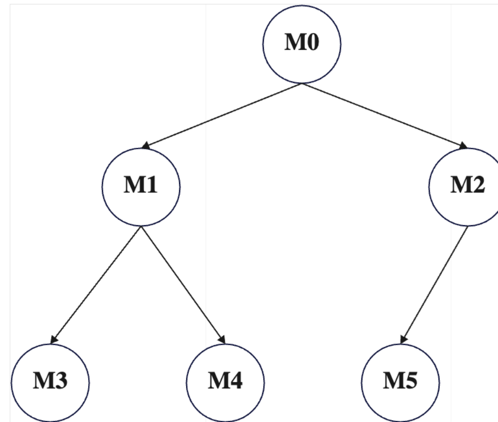


Figure 2

Min Max

Minimax is a typical decision tree algorithm for games like Gomoku, where each node, as described above, represents a possible state in the game. At its core, Minimax seeks to maximize the potential scores for the player (the maximizer) while minimizing the gains for the opponent (the minimizer).

The algorithm switches between maximizing and minimizing phases at each level of the tree. Assuming you are the maximizing player, at your turn, you will explore all possible moves (child nodes) and choose the one that leads to the highest score. The score is determined by a utility function evaluating the game state, and it is based on factors like the number of aligned pieces or the potential to create a chain of five.

After the maximizer's move, the algorithm moves to the next layer (the opponent's turn), switching to minimizing mode. Here, it assumes the opponent will play optimally and thus chooses the move that results in the lowest score for the maximizer. This process of switching between maximizing and minimizing continues recursively, until an end state occurs or the max depth has been reached.

It is worth noting that there is no need to build the whole decision tree when implementing MiniMax. The tree can be traversed implicitly by recursive function calls.

Here is the detailed workflow of MiniMax (Assume you are playing black, and now it's your turn)

1. Look at all the places where you can make a legal move, make a temporary move at each possible location, and begin simulation for that move.

2. For each move, think ahead about possible responses by the white player (opponent). Each of these responses creates further branches in the tree. Specifically, in the maximum layer, you are making moves which lead to the highest score for you, while in the minimum layer, your opponent does the very opposite thing, making moves that are worst for you, minimizing your score.
3. When the simulation for the move(mentioned in 1) is done, undo the move, move to the next possible position, and begin simulation.
4. Choose the move with the highest score.

Here is the pseudo code for MiniMax:

Unset

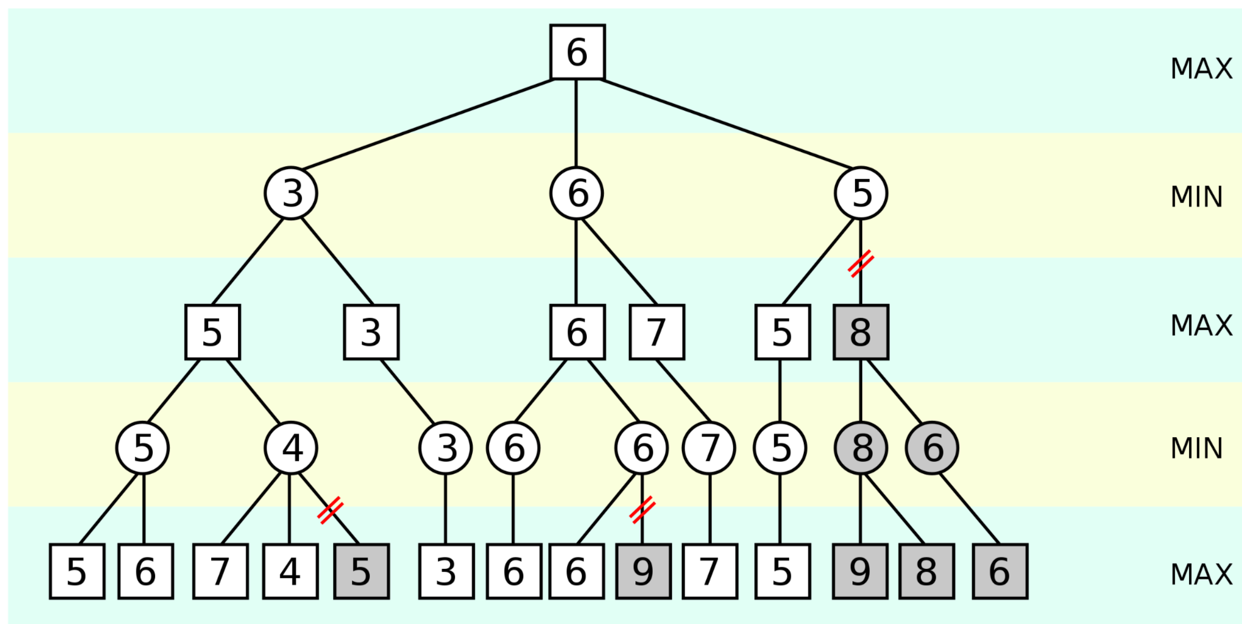
```
function  minimax( node, depth, maximizingPlayer ) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max( value, minimax( child, depth - 1, FALSE ) )
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min( value, minimax( child, depth - 1, TRUE ) )
        return value
```

Alpha-Beta Pruning

MiniMax is a *brute force* solution. Its time complexity can be exponential, which is unacceptable in real life application. As Gomoku is played on a 15x15 board, the decision tree for the game can become extremely large, leading to a computational bottleneck. Alpha-beta pruning is a common method to reduce the number of nodes evaluated in the decision tree, speeding up the search process.

The essence of alpha-beta pruning is to ignore parts of the decision tree that don't *affect* the final decision. It is based on the principle that if you have already found a "good enough" path, there's no need to explore options that are doomed to be worse. For the maximizing player, any branch that leads to a value lower than the current best (which is alpha) is irrelevant. Similarly, for the minimizing player, branches leading to a value higher than the current worst (which is beta) won't impact the results. This selective process allows the algorithm to dramatically reduce the number of branches it needs to consider, leading to efficiency to a great extent.

Here is a description of how it works.



Starting Point: The root of the tree represents the current game state with the MAX player (you) to move, aiming for the highest score.

Initial Exploration: The MAX player looks at all possible moves. The first move considered has a value of 5, which becomes the initial alpha value.

First Pruning Opportunity: When evaluating the MIN player's options, the first move results in a score of 3, and the second move shows a 6. Since the MAX player already has an option that leads to a 5, the branch with a 6 is explored further, but when a subsequent move by MAX in that branch leads to a 7, it means the MIN player would never allow this, as it's worse than the 3 previously seen. Therefore, the rest of the branches following the 6 can be pruned.

Continuing Exploration and Pruning: As the tree evaluation continues, the MAX player discovers an 8, setting a new alpha value. This means any subsequent branches that lead to a value less than 8 for the MIN player can be pruned immediately, as shown on the far right of the tree with the red slashes.

Outcome: The process continues, with each layer alternating between maximizing and minimizing, and Alpha-Beta pruning optimizes this by cutting branches that won't affect the overall decision. This is why certain branches are marked with red slashes—these are pruned, not requiring further evaluation because they can't possibly influence the final decision.

Here is the pseudo code for Minimax with alpha beta pruning.
(https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)

Unset
```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, alphabeta(child, depth - 1, α, β,
FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth - 1, α, β,
TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
```

```
        return value
```

Benefits and Limitations

Alpha-beta pruning significantly reduces the number of nodes that need to be evaluated in the decision tree. For now, the AI has reached a fairly good level of performance, capable of demonstrating basic tactics and strategies. However, it is far from flawless; there are limitations inherent to the algorithm's design. Despite its calculated approach, the AI may not always anticipate complex long-term strategies.