CS 3503: Computer Organization and Architecture Checkers Game Report

# **Design and Implementation of Utility Class - Including Bitwise Operations and Binary Arithmetic Usage Explanations**

The utility class was designed to use a 64 - bit board. Each player’s pieces are represented by a 64 bit integer (U64), corresponding to a square on the 8x8 board. 1’s represent positions occupied, 0 represents empty spaces. It creates 4 boards in total at the beginning of the class. Two for the standard pieces, one for the red player (PLAYER1), the other for the black player (PLAYER2). The other two boards are used for king pieces, one for the red king pieces (PLAYER1 KINGS) and the other for black king pieces (PLAYER2 KINGS). Afterwards, it is designed to keep track of the player based on 1 and -1, 1 meaning the player moving down, and -1 for the player moving up.

It then implements the ***SetBit*** method and a ***ClearBit*** method, both take in a 64bit value and an int called position, and both return a 64 bit value. Inside both these functions is a 1ULL << position which creates a bitmask with a 1 in the correct position and | combines it with its current value, setting the bit. The bitwise OR operator (|) combines this bitmask with the current value, ensuring that the bit at position is set to 1.The ***ClearBit*** does a similar function but rather creates a 0 in the position.The NOT operator (~) inverts this bitmask, resulting in 0 at the desired position and 1s elsewhere. The bitwise AND operator (&) is then used to clear the bit at position, resulting in that bit being set to 0.

The next function added was the ***InitializeBoard*** method, which returns void, and takes in a 64 bit pointing at the board and a 64 bit pointing at the kings, it then initializes player1 and player 2’s 4 bitboards with 0’s (0ULL). It then set’s the board for player1 and player2, calling SetBit. I then decided to add an extra method which would allow me to customize my board to have rows be called by numbers and columns by numbers. Afterwards, the method ***PrintBoard*** was implemented to print the rows and columns in a 8x8 grid design with rows labeled 0-7 and the columns A-H, and print out R for red player and B for black player in correct formatting. It was also implemented to check to see if that player is in the Kings bitboard, if it is then it will print out with a special symbol to show it as a king.

***IsLegalMove*** is implemented to return an integer, and takes in a U64 bit pointing at the board, a U64 pointing at the kings, an integer player, an integer start, and an integer end. It then checks for basic move validation, ensuring in bounds, continuing to check if starting position has a pieces, check if the end position is empty, and now check to see if the move is a valid move (diagonal), but it then had to be implemented to allow for king movement (diagonal + forwards and backwards). The bitwise AND operator (&) checks if the bit at the start position is set.

If the result is 0, it indicates there is no piece at that position. Lastly, it checks to ensure capturing is valid or not when a move is made. Bitwise operations and binary arithmetic came into play extremely in this method. This method used integer division and modulus to ensure diagonal moves:

// Calculate row and column differences

int rowDiff = (end / 8) - (start / 8);

int colDiff = (end % 8) - (start % 8);

***UpdateGameState*** is the next method in the Utility class. This method is a void method, taking in only the current U64 pointed board, with an integer pointing at the current player. I found a built in method that helped me count pieces for both players, then I used boolean algebra to check if player 1 had 0 pieces or player 2 had 0 pieces, if either, it prints out the winner and exits the game. At the end of the method, if the exit is not executed (if there are still pieces for both players), then it switches turns. ***CapturePiece*** removed a piece from the opponent’s bitboard when captured, so it took in the U64 pointer board, an integer player (stating which player's piece had been captured at what position) and an integer position. The middle position is calculated by averaging the start and end positions. The code checks if the middle position has a piece (either player) by using the bitwise AND operator. If a piece occupies the middle square, the move is considered a capture.

***MovePiece*** is the second to last method implemented. Its goal was to move a piece from one piece to another, my implementation required an integer return value, and took in U64 board, U64 kings, integer player, const char pointer from, const char pointer to. It’d then first call my method that converted the input (since I created a custom format) into an int index. Then check for invalid input, if the move is legal (calling IsLegalMove), move the pieces, check for capturing, and lastly check for capturing, indicating success with a 1 returned. **Middle Calculation**: The average of fromIndex and toIndex gives the middle position where an opponent's piece might be. **Bitwise AND**: Checks if the middle position is occupied by an opponent's piece by creating a mask with 1ULL << middle and using the AND operator to verify if that bit is set.

The last method implemented is the void ***PromoteToKing*** which takes in a U64 board, the U64 pointing to kings, an integer player, and an integer position. It checks rows 0-7 for Player 1, and Row 56-63 for Player 2. If either is found it calls ***SetBit*** for the king bitboard for the player and passes the position.

Boolean Algebra for Player 1:

if (player == PLAYER1 && position >= 56) { // Rows 0-7 for Player 1

Boolean Algebra for Player 2:

} else if (player == PLAYER2 && position < 8) { // Rows 56-63 for Player 2

## **Significance of Operations in Low-Level Computing, Converting Data, Challenges Faced, and Solution’s**

The significance of these operations in low level computing and game development is crucial. It is the foundation of basic movements and basic logic in most, if not all the game’s being created since automating bitwise operations so it is working in the back end of the computer without us needing to go so low level. The computer reads in 0’s and 1’s and therefore these skills are very powerful, C itself, if used right with these operations can be a very powerful tool. Low-level computing allows for direct manipulation of the binary data, it allows for efficient methods and functions to directly shift or easily check piece positions and move validating.

Converting the data between different formats using bitwise manipulation was crucial for compatibility. I had to use bitwise conversions such as binary to decimal and decimal to binary. The challenges I faced with developing and implementing the utility class and bitboard was starting off with a 32 bit board. I quickly realized I was making a mistake due to the board not being 64 when watching a chess game backend creation video on youtube. 8x8 = 64 bits. The other challenge was the king. I was unsure and attempted to have him with the 2 boards of player 1 and player 2. To overcome this, creating 4 separate boards made it so much more simpler and efficient. Especially when checking for a king piece and setting that.

DOCUMENTATION INCLUDED ON README

<https://github.com/rami-elmostafa/CheckersGame.git>