**Project 1 – BitBoard Checker’s Game Report**

Adam Llado

Professor Chris Regan

CS 3503

Oct 13th 2024

**1. Design, Use, & Implementation of the Utility Class:**

At the heart of Bitboard checkers, only 0s and 1s; a Utility class is constructed to manipulate the numerical representation of each board using bitwise operators so that the binary can be accurately portrayed as the game state progresses. The Utility class is the brains of the game, being used within core functionalities including the methods for moving pieces, capturing pieces, and checking pawn movement integrity. Methods have been copied to satisfy input of different data types including int, long, short, and byte, but each method remains the same for each data type. An important note is the use of an array called checkersSquareMap which is a logical representation of the game board that holds a –1 to represent out of bounds squares and numbers 0-31 that act as the possible squares a piece can move to. When manipulating bits, we retrieve the value of these squares and use them as the position to manipulate bits on each bitboard, essentially acting as the binary exponents that tell the setting, clearing, and flipping bit methods which bit to manipulate. All functions and their implementations will be discussed in reference to the AdamLladoBoard.java file which is not explicitly mentioned in the explanations below. Here is a brief explanation of the design and implementation of each core method:

**1.1 flipBit**

The flipBit method allows for single bit manipulations to turn on or off bits as pawns move around the board. It uses the XOR bitwise operator and shifts a 1 onto the specified position wherever there isn’t a 1. The function is used for a fundamental variable called startBit inside the movePiece method that represents the numerical value of the startSquare when a pawn has been moved from its start position. This variable is used as a checking variable within conditional statements to ensure there is a pawn at this starting position before moving a piece, to enforce no backwards movements, and to check if a piece being moved is a king or not.

**1.2 setBit**

The setBit method is a fundamental function used throughout the logic of the game where it uses the OR bitwise operator to shift a 1 onto the specified position. This function is used in critical methods like MovePiece to set a specific bit of a bitboard to the endSquare that is determined by the checkersSquareMap array. Similar to the flipBit method, the setBit method is also used by a variable called endBit in pawn movement integrity checks to check if pawns move backward.

**1.3 binaryAND**

The binaryAND method is used in one instance to compare bits between the king bitboards and the startBit (starting position) to determine if the piece being moved is a king or not. We use this as our first conditional statement when moving a piece so that if it is a king piece, we can tell our MovePiece method to manipulate either of the king bitboards.

**1.4 binaryORed**

The binaryORed function is used in conjunction with the binaryMult function as an experiement to use binary multiplication to set bits as pieces move across the board. I have two variables used for this method, whitePawnsShiftSet and whitePawns, and after I shift the bit of the whitePawnsShiftSet variable to the endSquare the user wants to move to, I simply OR both numbers together, esentially adding them using bitwise operators.

**1.5 clearBit**

The clearBit method is the second fundamental movePiece function meant to remove or turn off the bit of the specified bitboard that is currently being played. Wherever a piece moves, there must be a method to remove it from its starting position, and clearBit achieves this by using the bitwise operators & and ~ to find the startSquare the user’s piece is on, and then remove it through ANDing the bits and shifting a 0, or ~1 onto the specified position. It is also used when capturing pieces and performs the same calculation but now uses the captureSquare variable as the position to be manipulated.

**1.6 checkBit**

The checkBit function is the catalyst to the bitboardTo2DArray method which allows the bitboards to be formatted to a 2D array so it can be visually represented on the terminal. The checkBit is used to check if the indexBit, or the bit that correlates to the current startSquare (row/col) of the for loop overlaps with any of bitboards by using conditional statements asking if they are equal. If they are, we know there should be a symbol of that bitboard in the array to represent the placement of that piece at that square.

**1.7 binaryString**

binaryString is the main print method used to convert the bitboards to a string of 32-bit characters to display the current state of each board. It uses a for loop with 32 iterations that simply adds the bit at each bit position to an empty binaryString by using the Integer.toString built-in function from java. It records each bit by shifting to the position i of each number and then Anding a 1 to that position, essentially checking if the position uses a 0 or 1 by Anding it. This function is used in the printBoardBinary method that outputs the whole board to the terminal and displays each bitboard in binary to the right of the physical gameboard. This allows you to actively see how moving pieces and capturing pieces affects the bitboards in real time.

**1.8 HexString**

The HexString function is used the exact same way as the binaryString function within the game, instead outputting the boards’ hex string representation. In the logic of the string, we use a java built in function called String.format that formats the decimal number as a hex string number, and we then take the length of the hexString and add leading zeros to represent the number as 8-bits which will equate to the 32-bit int data type we store the bitboards as.

**1.9 binaryAdd**

The binaryAdd function is pivotal in determining the captureSquare, or the square that pieces jump over when making a capture move. The function uses XOR and & bitwise operators that first XOR the two numbers, and then AND them until the second number equals zero. Once this occurs, the first number holds the sum of the two numbers, and we return num1. We then divide the sum by 2 to get the value of the captureSqaure held in the checkersSquareMap array. The captureSquare is used mainly in determining which square we should remove pieces in the capturePiece method, which makes it an important logical implementation using the binaryAdd function.

**1.10 binarySubtract**

BinarySubtract was used throughout the movePiece method in checking various pawn movements to ensure the integrity of the game. It acts as the brains for the pawn movement integrity checks where it determines if pawns are trying to move backwards, and if they are trying to jump empty spaces or multiple spaces that are considered illegal. The function uses XOR, ~, and & bitwise operators and works similarly to the binaryAdd function, but instead of just Anding the number after they have been XORed, it Ands the opposite of num1 (~num1) to num2 and then returns the num1 when num 2 equals zero.

**1.11 binaryMult**

To satisfy a project need to use binary multiplication in our checkers game, I decided to change the setting of bits for the whitePawns bitboard from simply setting each bit to using another variable that stores the bit shifted to the specific endSquare the user wants to move to and then ORing the two numbers together. The variable is called whitePawnsShiftSet, and I first set a bit at position 0, and then I shift the whole number using the binaryMult function to shift it to the endSquare value. After that, the two numbers, whitePawns and whitePawnsShiftSet, are ORed together. I did this for the sake of project completeness, but I did not use this method for moving other pieces around because it is far too complex and could cause some edge case errors that I am not aware of.

**1.12 binaryDiv**

Although it was hard coming up with a reason to use binaryDiv in the game as it’s not as practical as setting and clearing singular bits, I did realize that I used division in determining the captureSquare by adding the startSquare and endSquare together and then dividing by 2. Since binaryDiv is division by 2, all I needed to do was simply shift the captureSquare sum right by 1, dividing it by 2 using my binaryDiv function. So, in essence, the binaryDiv is an important, yet simple method to find the value of captureSquare so that we can accurately remove pieces once they are captured on the board.

**2. The Significance of Binary and Bitwise Operations in Low-level Computing**

The importance of using binary and bitwise operations in low-level computing and game development stems from the fact that we as developers are always trying to improve our code and make it more efficient. The speed of functions and methods are improved through using bitwise operations because we are working with the binary of each variable which is what the computer itself understands. If we choose to manipulate these bits using bitwise operators, it takes out any unnecessary conversions to binary when the machine is performing arithmetic operations. The most obvious advantage though in programming this checkers game was the memory usage and how binary can store a whole board of information with pieces in a single integer variable. Although I did use an array because of the countless chess implementations I saw converting their bitboards to an array for easy output and for pawn movement checks with the various symbols, I do see how that array compared to the 4 integer variables I used to store the bitboards is far less efficient due to the greater amount of space it uses. Thus, binary can be more useful in game development and low-level computing when memory and space is an issue.

**3. The Process of Converting Data Between Different Formats**

The process of converting the bitboards to binary and hexadecimal required that the numbers be translated to strings to allow for easy output to the terminal. The binaryString function simply iterated over the binary of each number and recorded whether it was a 0 or 1 by ANDing a 1 to the current position in the for loop. I did not have to worry about padding the string with leading zeros because my for loop has 32 iterations starting at 31 and records each bit from 0-31. However, with the HexString function, I did have to pad zeros because the String.format built in function I use to format the bitboards to hex does remove leading zeros. Once it’s formatted to a hexString, I check the length of that string and compare it to the value 8. If it is less than 8, then I add zeros until it equals 8. I think these functions are simple enough for people to understand and utilize simple for loops to pad zeros when necessary or iterate over the binary and record each bit. Once the functions return their values, I simply added the output to the right of certain columns within my printBoardBinary and printBoardHex methods so that users can see the values change in real time to their moves on the board.

**4. The Challenges Faced During Implementation and How They Were Overcome**

**4.1 Conversion from Binary to 2D Array**

One of the toughest challenges I faced was understanding how to convert my bitboards to a 2D array that I wanted to implement because of all the chess implementations that I followed which used this feature. However, many of their implementations used pre-calculated board positions found on Wikipedia to instantiate the starting board and piece positions, but I knew I had to convert mine using bitwise operations and my Utility Class. Many of the chess implementations used 64 item arrays as well to convert the boards, when checkers, although 64 squares, only uses 32 of them. So, I kept the 64 item, 2D array but used a –1 to represent out of bounds on the board. This allowed me to use the values 0-31 and store them in my checkersSquareMap array to reference when moving pieces, clearing pieces, and checking if the position of that bit is equal to a position for each bitboard when converting it to a 2D array. So, for instance, I made four different int checkBit variables (whitePawnsCheckBit, blackPawnsCheckBit, etc.), and then in my 8x8 double for loop, I would place a dot at every position equal to –1 in the checkersSquareMap array, and then use a variable called indexBit that set a bit at the squarePosition we were currently on to compare it to the checkBits of each bitboard and determine which bitboard had a bit set at that squarePosition. For instance, if the double for loop is currently on row 1, column 2, this would mean that the value of squarePosition is 5, so I would set the 5th bit of the indexBit variable and then use if statements to check which bitboard had a bit set in the 5th position of their binary integer value. This made the logic and conversion of bitboard to a 2D array simple, even though it was the toughest thing to conceive.

**4.2 Pawn Movement Integrity Checks**

The conditional statements used to check every pawn movement and ensure pawns were moved legally was an exhaustive and time-consuming endeavor. Not only did you have to create each conditional statement, but the order of these staements was important as well because certain checks come before others when moving a checkers piece logically. However, I utilized a variable called isWhite to represent the current turn and passed it into my movePiece method where it was consistently used to differentiate between white and black pawn checks within my conditional statements. Another key function was the use of binarySubtract as many of these conditional statements required comparing the difference of the values from the starting and ending positions or comparing it to the expected number of spaces the piece would have shifted to get to the ending position. For example, to check if pieces were trying to jump empty spaces when not making adjacent moves, I would take the difference of the startSquare and endSquare using binarySubtract and then determine if these values were greater than 3,4, or 5, and if the jumped square was empty, this would mean they were making an illegal jump by jumping an empty square.

**4.3 Checking for Double Jumps**

When I first started implementing my double jump check method, I had it inside one method. However, after a while, I realized I was using a lot of redundant code because the checks depended on where each piece was on the board and the double jump checks in the middle were resuing code from the left and right checks. Because of this, I split the double jump check function into smaller functions depending on if I was checking whitePawns to the left, to the right, in the middle, or blackPawns and kings in that similar fashion. This way, it would reduce my use of redundant code and make my programming a little easier to read and understand. The logic of each method though is straightforward, checking the upper right and left or lower right and left squares depending on if it was a black or white piece and then checking if the square being jumped over had a piece to capture. When I finally coded the isKingDoubleJump method, all I had to do was call the previous methods I had already implemented, saving me a lot of time and space in checking for double jumps.