Playing Checkers Using Bits

Brooks, Kameron
Kennesaw State University

***Abstract*—This paper outlines my experience of creating a fully functioning checkers game using individual bits as the mechanism for storing the board state, and using bitwise operations for implementing the game logic and modifying the game state.**

I. Introduction

I was given the task of using individual bits for representing pieces on a checkers board and manipulating their state using bitwise operations. Traditionally when approaching a project such as this, I would use an object oriented approach (OOP) and represent each piece on the board as a struct that contains various properties such as, color, king status, etc. , but that was not acceptable for this project. These constraints forced me to approach the problem in a way that was novel for me, and in the process, gave me a better understanding of the possibilities for representing data using a minimal memory footprint. In the sections below, I will detail the challenges faced and the solutions I devised while working on this project as well as generally give insight into my thought process.

A. Choosing a Language

The first task of this project, was choosing a language to program it with. I am familiar with several languages/platforms and compared the ability and relative ease with which I could achieve the following:

A. Bitwise operations and bit level access
B. Graphical and UI capability
C. Required boilerplate & lines of code
D. Output & Code portability

The options were as follows:
Ranked from 1-10 based on my level of experience with applicable libraries/modules and built-in functionality

Language	A	B	C	D
Python	6	5	7	8
C++	10	5*	3	4
C#	10	8	8	8
Javascript	7	9	7	10

** Score could be significantly better with a significant increase in effort*

Since this project is about bitwise operations, Python and Javascript were excluded due to having the lowest scores in the table. C# was chosen based on my familiarity with it over C++. Additionaly, C# would require a lot less set up to get graphical and UI features and be more portable since compilation issues would not be a factor.

B. Platform

Since the language of choice was C#, the next decision was selecting a platform or framework. I considered using Unity, but ultimately settled on using windows forms for simplicity. As a result, the project is only able to run on windows machines, though other operating system compatibility is not a requirement of the project. Furthermore, the underlying application logic can easily be ported over to a more portable framework later if required.

II. Production

The first connection that I made was the relationship between the number of cells on an 8x8 checker board and the number of bits in an unsigned long (64 bits). My initial thought was that each bit could represent a cell on the board that either has (1) or does not have (0) a piece on it. The obvious problem with this naive approach, is that you would not be able to store information regarding which color the piece is. Therefore, you would need one unsigned long for each team. Since each team has its own set of pieces stored as an unsigned long, it is technically possible to end up with a state where both teams show their own piece in the same cell; I would have to make sure to handle such a scenario in the game logic itself. This solution is still incomplete, because it still does not account for piece king status (That will be addressed in a later section).

A. *Bitwise Operations Utility Class*

After determining how the game data would be represented, I set out to create the utility class that would perform the bitwise operations on the bitfields. I created several operations based on the project requirements as well as some additional operations for convenience. The class is a static class that contains only static methods, so no instantiation is required of the class itself. Each method (where applicable) is also overloaded to allow for byte (8-bit unsigned int), ushort (16-bit unsigned int), uint (32-bit unsigned int), and ulong (64-bit unsigned int). For the sake of brevity, only the uint version of the signatures is shown below.

Simple Operations

Method	Description
`FlipBit (uint val, int pos)`	Flip a bit in a position
`SetBit (uint val, int pos)`	Set the bit in the specified position to 1
`ClearBit (uint val, int pos)`	Clear the bit in the specified position
`CheckBit (uint val, int pos)`	Check if the bit at the specified position is 1
`ShiftLeft (uint val, int pos)`	Shift the bits to the left by the specified amount
`ShiftRight (uint val, int pos)`	Shift the bits to the right by the specified amount

More Complex Operations

Method	Description
`MoveBit (uint val, int from, int to)`	Move a bit from one position to another if it is 1
`CountOnes (uint b)`	Count the number of 1 bits in the input
`GetFirstBitPosition (uint bitMask)`	Get the index of the least-significant 1 bit in the input
`BitMatch (uint a, uint b)`	Returns true if any of the bits overlap
`CreateBitMask (int loc)`	Create a bitmask with a 1 at the specified location
`CombineBits (uint a, uint b)`	Bitwise Or ($A \cup B$)
`IntersectBits(uint a, uint b)`	Bitwise And ($A \cap B$)
`ToBinaryString(uint val)`	Get a binary string
`ToHexString(uint val)`	Get a hexadecimal string

Now I had a class that could perform all of the bitwise operations that I needed, so it was time to start using it to implement some game logic.

B. *What is Checkers Anyway?*

I had gotten this far because I remembered what a checker board looks like, but I had forgotten the details of how to play checkers so I needed to consult wikipedia and watch some videos on YouTube for a refresher. The rules can be found here [1] and a rather in-depth video explanation can be found here [2]. Now that I knew what the rules of the game are, the next step is figuring out how to implement them in the game.
In checkers, the teams move in opposite directions, so I knew that each piece would have to move differently based on the team it is on. They also move differently if they have reached the end of the board and become kings, so I knew there would

need to be additional logic to handle that feature as well.

C. *First Movement Logic Attempt*

At this point, each team's pieces were represented as a unsigned long where each bit is one of the 64 cells on a 8x8 board. The first approach I came up with involved using the bit position to come up with $[x, y]$ coordinates so that I could make calculations based on the x and y positions on the board.
For example, if a piece is at the first bit, then I know it is at $[0, 0]$ on the board. If the bit position is represented as b, then $y = floor(b/8)$ and $x = b - (8y)$.
Once x and y are known, then it is possible to determine which moves the piece can make. For example, if $x = 0$ then the piece cannot move left, if $x = 7$ then it cannot move right, and so on.
This worked, but there were a lot of conversions between bit indices and $[x, y]$ coordinates happening. This was certainly not an efficient or elegant solution. If checking to see if a piece could be captured to the right for example, it would require the selected piece to convert its bit position to $[x, y]$ coordinates, then add 1 to x and y to get x' and y', then convert those new coordinates back to a bit index by $i' = 8y' + x'$. At that point, it could be determined if there was a 1 at the bit location i'.
This resulted in very convoluted nested if statements and wasted a lot of time converting bit indices to $[x, y]$ coordinates. I knew there was a better way.

D. *We Can Do Better*

The solution was to just use the bit indices for the movement logic without converting to $[x, y]$ coordinates. To do this, I first realized that I did not need to use unsigned longs at all. The checker board is 8x8, but you can only land on 4 cells per row. This means there is no need to store the state of the non-playable cells since they will always have a value of 0. I changed the player board unsigned longs (64bits) to unsigned ints (32 bits).
Next, I precreated all the bitmasks that represent different piece conditions on the board. This way, instead of asking questions about specific x and y positions, I could just check to see if a piece bit was contained within a certain bitmask. As a simple example, If I want to know if a piece is in the last row of the board, instead of finding its y position and checking if it is 7, I can instead test the bit and see if it is contained within the specified "last row" bitmask that only contains 1s in the last row. This method is simple to understand, requires no conversions, and is very efficient.
If p = the piece bitmask (all 0s except for the index of the piece) and b is the conditional bitmask. You can determine if the state of the piece meets the conditional criteria by performing:

$$p \wedge b \neq 0$$

Utility Bitmasks:

Even Rows Mask
`(0x0F0F0F0Fu)`

▨	0	▨	0	▨	0	▨	0
1	▨	1	▨	1	▨	1	▨
▨	0	▨	0	▨	0	▨	0
1	▨	1	▨	1	▨	1	▨
▨	0	▨	0	▨	0	▨	0
1	▨	1	▨	1	▨	1	▨
▨	0	▨	0	▨	0	▨	0
1	▨	1	▨	1	▨	1	▨

Odd Rows Mask
`(0xF0F0F0F0u)`

▨	1	▨	1	▨	1	▨	1
0	▨	0	▨	0	▨	0	▨
▨	1	▨	1	▨	1	▨	1
0	▨	0	▨	0	▨	0	▨
▨	1	▨	1	▨	1	▨	1
0	▨	0	▨	0	▨	0	▨
▨	1	▨	1	▨	1	▨	1
0	▨	0	▨	0	▨	0	▨

Player Specific Bitmasks:

Initial Player 1 State Mask
`(0x00000FFFu)`

	0		0		0		0
0		0		0		0	
	0		0		0		0
0		0		0		0	
	0		0		0		0
1		1		1		1	
	1		1		1		1
1		1		1		1	

Player 1 Final Row Mask
`(0xF0000000u)`

	1		1		1		1
0		0		0		0	
	0		0		0		0
0		0		0		0	
	0		0		0		0
0		0		0		0	
	0		0		0		0
0		0		0		0	

Player 1 Left Move Mask
`(0x0EFEFEFEu)`

	0		0		0		0
0		1		1		1	
	1		1		1		1
0		1		1		1	
	1		1		1		1
0		1		1		1	
	1		1		1		1
0		1		1		1	

Player 1 Right Move Mask
`(0x0F7F7F7Fu)`

	0		0		0		0
1		1		1		1	
	1		1		1		0
1		1		1		1	
	1		1		1		0
1		1		1		1	
	1		1		1		0
1		1		1		1	

Player 1 Left Jump Mask
`(0x00EEEEEEu)`

	0		0		0		0
0		0		0		0	
	0		1		1		1
0		1		1		1	
	0		1		1		1
0		1		1		1	
	0		1		1		1
0		1		1		1	

Player 1 Right Jump Mask
`(0x00777777u)`

	0		0		0		0
0		0		0		0	
	1		1		1		0
1		1		1		0	
	1		1		1		0
1		1		1		0	
	1		1		1		0
1		1		1		0	

The same is done for all of the Player 2 masks, they are exact visual opposites of the Player 1 masks so they have been left out for the sake of brevity.
Now that I had these masks pre-calculated and saved as constants in another static utility class, I could begin the process of constructing the main gameplay loop.

E. Gameplay

Since I decided to implement this game using a Windows Forms app, the core ”loop” is event-driven. The window is listening for mouse events such as movements and clicks and then constantly repainting the game when it receives one of these events. At any time, it is the red team's turn or the black team's turn, and that determines which pieces can be selected by the user. The potential moves are calculated for a piece at the time the user selects it and these potential moves are stored as another unsigned int bitmask. The potential moves bitmask is displayed to the user so that a potential move can be selected. When a potential move is selected, the piece executes that move, increments the turn, and alerts any listeners that the board has changed via a callback. When the turn is incremented, some cleanup is done, the turn counter is incremented, and several game over conditions are checked.

This is where I ran into the next problem. The constant rendering caused the panel that was rendering the game to flicker. I was not sure how to fix this issue at first, but after some internet searching I discovered that this panel needed to be double buffered. That allows the panel to write pixel data to a second buffer while the first one is being shown and continuously swap them. To fix this issue, I created a DoubleBufferedPanel class that is derived from the Panel class. This child class performs the same function as the Panel class, but is configured to use double buffering. After replacing the panel with an instance of this new class, the issue was resolved.

F. King Me!

At this point, I still did not have a mechanism for storing the king status of pieces. I added another unsigned integer mask that is shared between both teams. This mask can be shared because there will ever only be one team's piece in any bit position, therefore there is no need for each team to have its own king state bitmask. If the king status bitmask has a 1 at a bit location, that means that piece (belonging to Player 1 or Player 2) is a king. A simple but effective solution.

G. Final Movement Calculation Logic

When the user selects a piece, the game's selection mask is updated so that all its bits are set to 0, except for the bit representing the selected piece. Based on the piece's team and king status, the appropriate left and right jump masks are chosen. The selection mask is compared to the player's left jump mask using the bitwise AND operator to determine if it is in a position where a left jump would still be within the bounds of the board. The target capture position and landing position are then calculated by bit shifting the selection mask and comparing this shifted mask with both teams' board bitmasks using the bitwise AND operator. The same is then done with the player's right jump mask and if the piece is a king, the reverse left and right jump masks as well. This jumping code is called recursively to allow for multiple captures if there is a connected chain of enemy pieces. Any possible moves are stored in the potential moves bitmask.

If there are no jumps available for a piece, then the same operation is performed using the left and right move masks (without the recursion). This ensures that a piece cannot move passively if in position to capture an enemy piece.

Potential captures are stored in an array of 32 bitmasks. Each bit in the potential moves bitmask has a corresponding potential captures bitmask that contains 1s for each piece that will be removed from the board if the selected piece moves to that position. When a piece is initially selected, all bitmasks in the array are reset to 0 to clear any previous data. During the jump calculation phase, these potential capture masks are updated accordingly.

H. Solution Analysis

This model changes the logic into a simple series of bitwise AND operations and bitwise shifts instead of a tangled mess of $[x, y]$ bounds checks and nested if statements. No time is wasted converting between bit indices and $[x, y]$ coordinates, which also makes this method more efficient. And lastly a lot of memory is saved by representing the pieces as bitfields instead of using an OOP approach. We can estimate this memory savings by comparing the memory usage of this approach to a potential straight-forward OOP approach.

1) OOP Approach: Each piece would be represented as a Piece struct that contains a few fields.

Field	**Type**
`team`	`byte (8 bits)`
`isKing`	`bool (8 bits)`
Total	**16 bits**

The game begins with 12 pieces per team (24 total), so the minumum number of bits required to store the piece objects (ignoring any struct padding) will be $24 \times 16 = 384$ bits.
The board itself would be a 8x8 grid of pointers to Piece structs where a null pointer indicates there is no piece in the cell. In a 64-bit application, each pointer is 64 bits. This results in $64 \times 64 = 4096$ bits of memory for the board.
The memory required for the board and pieces are $384 + 4096 = 4480$

2) Bitwise Approach: Each team has a 32-bit *pieces* bitmask that represents its pieces, $32 \times 2 = 64$ bits
There is a shared 32-bit *king status* bitmask that represents the king status of each piece on the board. That brings the total memory required for the board and pieces to $64 + 32 = 96$ bits
That is a 97.9% reduction in memory use.

$$\frac{4480 - 96}{4480} = 97.9\%$$

Another way of looking at it, is that it takes almost the same amount of memory to store one piece and one board cell using the OOP method as it takes to store the entire board state using the bitwise method. That is a dramatic memory savings.

III. Rendering and UI

A. Windows Forms UI

The UI is all handled using callbacks to communicate gameplay changes to other windows forms controls in the window. There are several controls on the application that are kept in sync with the game state and show various properties about the game. Properties such as, the current turn, each team's score, and the binary and hexidecimal representations of the piece bitmasks are displayed. There is also an ascii representation of the game state to the right of the graphical representation. A *Reset* button in the top right corner can be used to restart the game when it is over or give up prematurely if the user is failing miserably. The bottom of the window shows the bitmask of the cursor position on the board.

B. Rendering the Board

By making use of the custom DoubleBuffered-Panel class mentioned earlier, I leveraged the built-in .NET windows forms drawing API to render colored boxes and PNG images that represent all elements within the game. The pieces are represented using PNG images that I made with Adobe Photoshop, as are the move selection icons. The implementation of this part is all very straightforward.

IV. Conclusion

I successfully built a functional checkers game using bitwise operations as the core mechanic and I learned a lot in the process. Coming from a background in object-oriented programming I had to challenge my usual approach, and as a result, created a system that was significantly smaller and more performant. I am still amazed at how much information can be contained in just three 32-bit integers. Moving forward, I have gained a new perspective on how to approach tasks like this in the future.

References

[1] W. contributors. (2024) Checkers. *Wikipedia, The Free Encyclopedia*. Accessed: October 12, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Checkers

[2] azcheckers, "Back to the basics: how to play checkers," *YouTube*, March 30 2020, youTube video. [Online]. Available: https://www.youtube.com/watch?v=WD3NTNQElew