Lab 4– Checkers

CMPUT 229

Background

Checkers Game

Checkers is a two player strategy board game, the name checkers comes from the checkered board its played on

- Also known as draughts

Checkers Game – How to Play

The game is setup as shown, black moves first

Players take turns moving one of their pieces

Checkers Game – How to Play

Two types of moves:

- Slide: Move forward 1 square diagonally
  - Cannot move to an occupied square
- Jump: Move forward 2 squares diagonally
  - Must ‘jump’ over an opposing piece to an unoccupied square
  - The jumped over piece is ‘captured’ and removed from the board
  - A player may make multiple consecutive jumps with a single piece in one move

Checkers Game – How to Play

Promote ‘Man’ to ‘King’

- When a ‘man’ piece reaches the other side of the board, it becomes a ‘King’
- Kings are the same as man pieces, but they can also move backwards

Winning:

- The game is won if at any point the opponent has no legal moves
- This most commonly occurs by capturing all of the enemies pieces, leaving your opponent with no valid moves

Bitboards

Bitboards are array data structures commonly used in computer systems that handle board games

Each bit in the bitboard corresponds to a single space on the board

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

0100 1110 0010 0010

Bitboards

Bitboards can be very useful when the number of spaces we need to track fits in a word or double word

Bitwise operations, such as AND and OR can be used on words to build game states or extract information

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

0100 1110 0010 0010

Bitboards and Checkers

Notice that there are 64 squares on the board, but only 32 (the dark squares) are playable squares.

Thus we are able to store one bitboard in a single 32 bit RISC-V word

Bitboards and Checkers

Since bitboards only have binary information for each cell, we are not able to represent a game state as a single word

For this lab we will have 3 bitboards to represent the game state

`RED, BLACK, KINGS`

	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

Bitboards and Checkers

The benefits of bitboards comes from the speed and simplicity of manipulating entire game states.

For example, suppose we wanna check if cell(s) are occupied, first create a bitmask with the bits set of the cells you wish to check

```
NotOcc = ~(RED | BLACK)
(NotOcc & Cells) ? Occupied : not Occupied
```

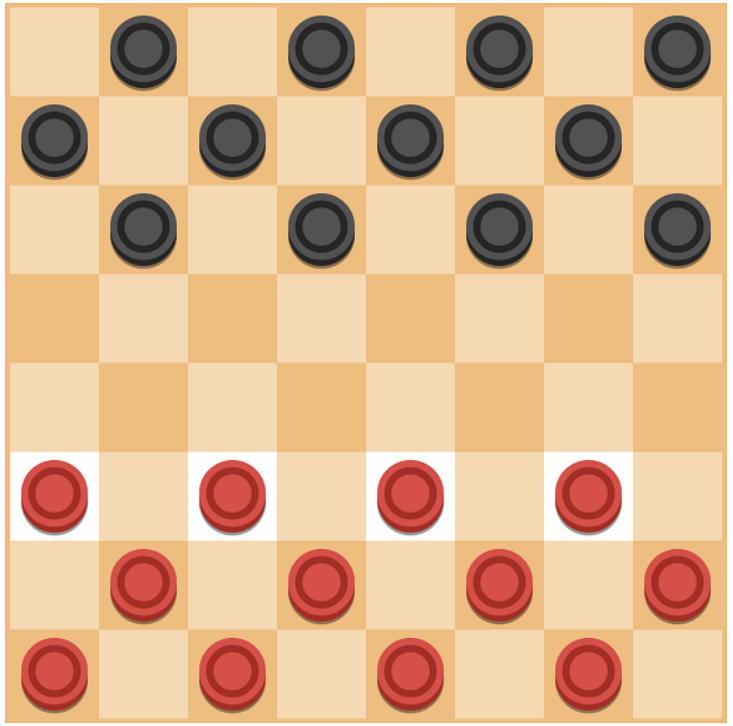
	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

Generating Moves

Generating Jump Moves

Thus we can generate jump moves for a src bb following this method:

1) Shift src by 4, get cells with enemy
2) Shift the enemy cells by 3 or 5 (using masks)
3) Get the unoccupied cells of those (add to moves)
4) Shift src by 3 or 5 (using masks)
5) Get get cells with enemies
6) Shift those by 4
7) Get the unoccupied cells of those (add to moves)



	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

Generating Slide Moves

Assume we are generating slide moves for black, (red is identical, just opposite)

If we left shift the bitboard by 4, it gives us one of the slide moves for each cell, the blue arrows show this.

To account for the other slide moves, we need to left shift some squares of the bitboard by 3 (green arrows) or 5 (red arrows) depending on the position.

	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

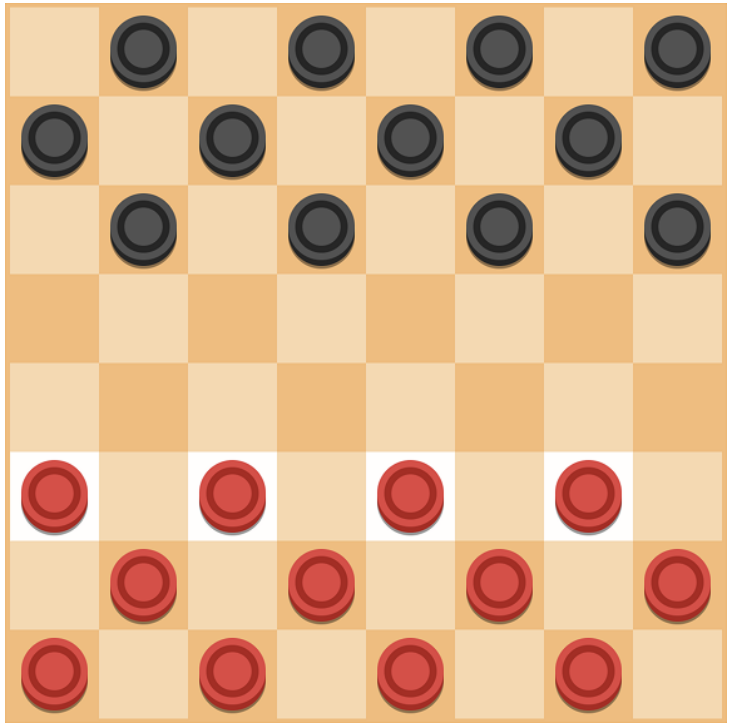
Generating Slide Moves

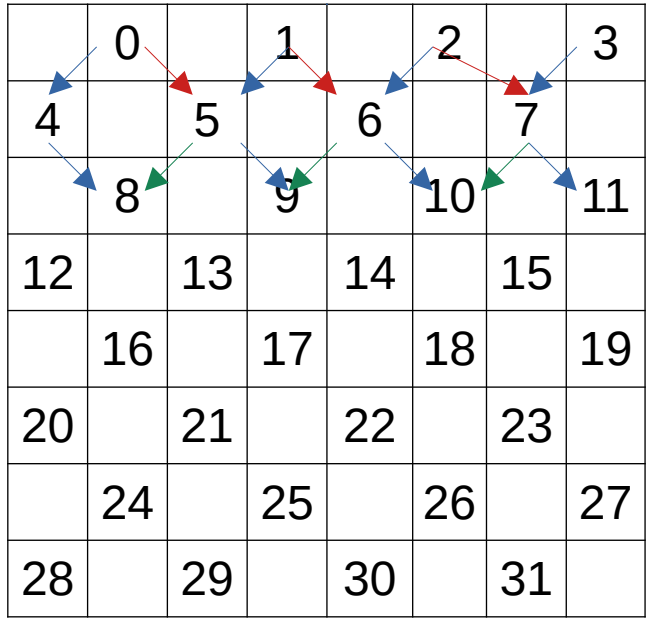
The bitboard mask representing these squares is given:

THREE_D = 0xE0E0E0E0 = FIVE_U

FIVE_D = 0x07070707 = THREE_U

Notice that when going the up the board instead of down the board, the masks for the cells that need to be shifted three and five are swapped in comparison with the masks for shifting down.



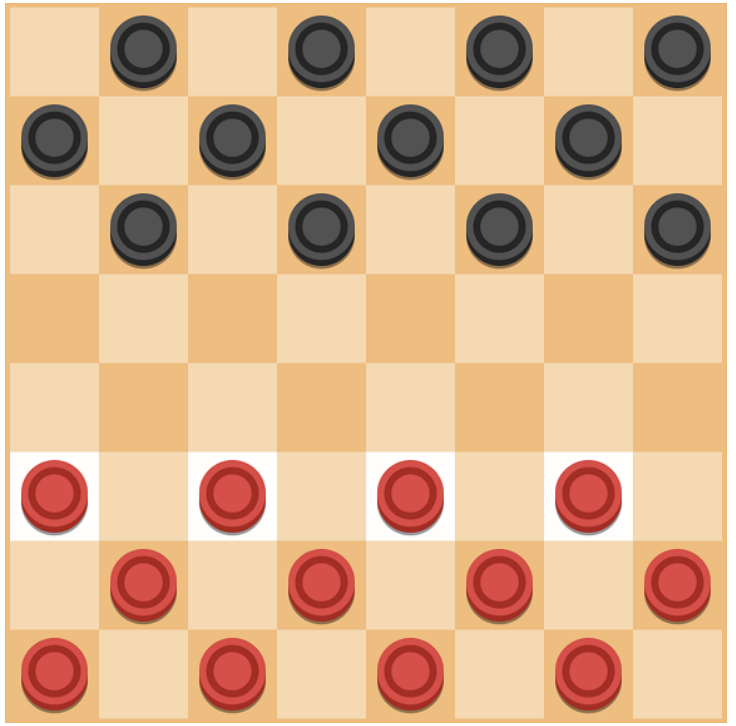


Generating Jump Moves

Along any top-left -> bottom-right diagonal, values alternate increasing by 4 and 5

Along any top-right -> bottom-left diagonal, values alternate increasing by 3 and 4

Thus every jump must consist of a shift of 4 and a shift of 3 or 5.



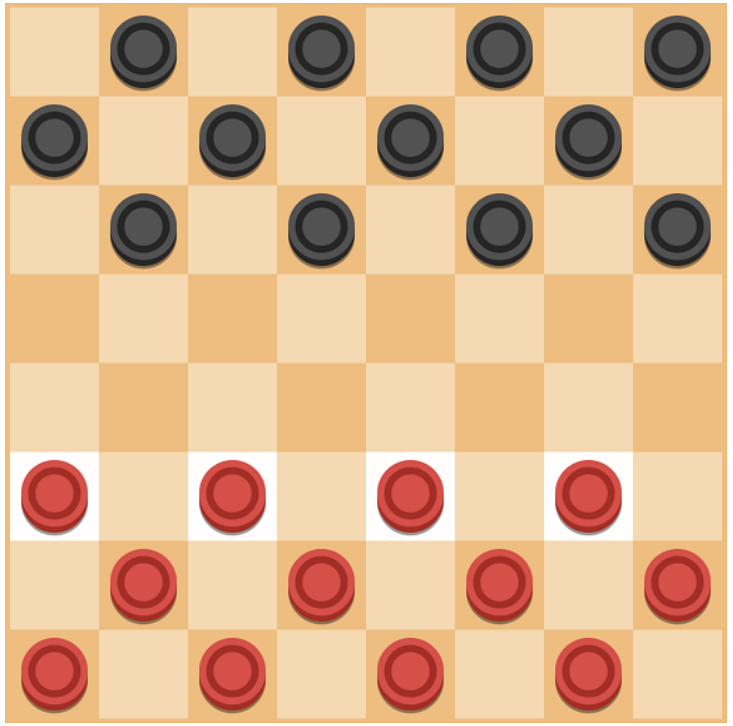
	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

What about Kings?

You can extract a color's kings by:

COLOR_BB & KINGS.

Then to generate moves for kings, simply generate moves in the opposite direction.



	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

Assignment

checkers Function

This function is the entry point for the game, it should initialize the starting position, starting turn, and have the main game loop.

displayGame Function

This function takes no arguments, and has no returns, prints the game to the rars MMIO interface.

This function should print the game spaces onto the board template displayed by the ‘displayBoard’ helper function.

genSlideMoves Function

Takes a bitboard of ‘source’ squares as an argument, returns a bitboard of all slide moves from those source squares.

The function will return a 0-bitboard if any of the source squares are invalid.

genJumpMoves Function

Takes a bitboard of ‘source’ squares as an argument, returns a bitboard of all jump moves from those source squares.

The function will return a 0-bitboard if any of the source squares are invalid.

makeMove Function

Takes a source bitboard (a0) and destination bitboard (a1).

This function performs a move and modifies bitboards without validation.

In the case of a jump, remember to remove ‘capture’ the opposing piece jumped over.

doTurn Function

Takes a pointer to the head of moves array terminated with a sentinel of (-1).

Checks if all the moves in moves array are valid moves, executing all the moves if they are valid, or doing nothing if the moves are not valid.

handler Function

The handler processes interrupts and exceptions. The handler must preserve all registers, including ‘t’ registers.

The End