Introduction

In this homework you will be implementing the game of Chess for 2 players. Complete knowledge of how to play or win Chess is not required to complete the assignment. The provided game play will allow 2 users to enter chess moves in turn and determine check and checkmate. General information about Chess can be found here for reference.

The goal of this assignment is:

- Practice writing MIPS functions calls and nested function calls
- Practice following MIPS register conventions
- Experience working with the stack & memory organization & data types
- Work with 2D arrays
- Interacting with input files (system calls)
- Improve your understanding of MIPS assembly instructions and writing MIPS assembly

To implement the game we will use a basic display screen which operates similarly to VT100. The standard VT100 terminal uses ANSI escape codes to specify the foreground and background colors of each position of the terminal. In this assignment, you will use the ANSI characters with and a specially designed font to display Chess in a 8 x 8 display. To display the game, the tool uses the idea of Memory Mapped I/O (MMIO), which is described a little later. In order to complete this assignment you will have to get familiar with displaying information in the MARS MMIO Chess display, which is a 2D array data structure.

Please read the assignment completely before implementing all the functions. You **MUST** implement each of the functions in the assignment as defined. It is OK to implement additional helper functions of your own in the `hw4.asm` file.

your grade.

> ! **You MUST follow the efficient MIPS calling and register conventions. Do not brute-force save registers! You WILL lose points.**
>
> ! **Do NOT rely on changes you make in your main files! We will test your functions with our own testing mains. Functions will not be called in the same order and will be called independent of each other.**

> 🤔 If you are having difficulties implementing these functions, write out the pseudo code or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS.
>
> 🤔 When writing your program, try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly

How to test your functions

To test your functions, simply open the provided `main.asm` file which tests your functions in MARS. Next, assemble and run the main file. Mars will take the contents of the file referenced with the `.include` at the end of the file and add the contents of your `hw4_netid.asm` file before assembling it.

`main.asm` provides a wrapper for your functions to enable actual game play. You may modify this file, or create your own files to test your functions independently. This file will not be used for grading.

> ! It is highly advised to write your own main programs (new individual files) and create your own data arrays (in the main files) to test each of your functions thoroughly. Your assignment will not be graded using the examples provided!

> ! Make sure to initialize all of your values within your functions! Never assume registers or memory will hold any particular values!

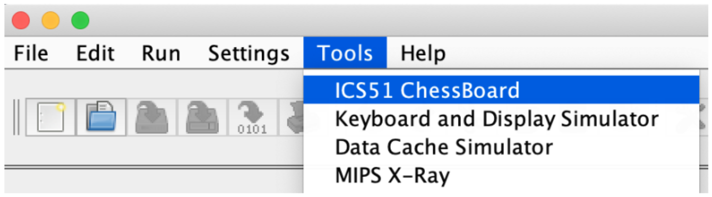
Any modifications to the main files will not be graded. You will only submit your `hw4_netid.asm` file to Gradescope. Make sure that all code required for implementing your functions are included in the `hw4_netid.asm` file!

> ! There is no need to define a `.data` section within your homework. Your functions should not refer to any labels directly.

possibly by moving labels you inadvertently defined in a main file to [illegible]

<u>The Chess Board Display in MARS</u>

For this assignment, we will use an add-on for Mars. **Download the Chess version of Mars from HW#4 Assignment on Canvas.** In the Tools menu, there is a tool called **ICS51 ChessBoard**.



To use the tool, you must connect it to your assembled program. When you are ready to test your code, assemble the `main.asm`, open the Chess tool and then press the button ``Connect to MIPS". When you run your code you will see real-time changes in the display when you change the MMIO values.

> **!** On some screen resolutions the window for the board is not wide enough and the last column of the board won't display. To resolve this issue, widen the board window.

This tool simulates displaying data on the screen similar to a terminal windows (like openlab machines hosted by ICS). A section of main memory is mapped directly to each cell in the screen. This technique is a form of [Memory Mapped I/O (MMIO)](). Each cell of the display is specified by a half-word (2 bytes) of information. The lower byte contains the ASCII character to be displayed at the cell position. The upper byte contains the background and foreground color information for the cell. The MMIO region in MARS begins at address `0xffff0000`. You can select this region in the MARS simulator from the drop down in the data segment sub-window. The simulator will treat the bytes starting at `0xffff0000` to `0xffff007f` as the values of a 8-column-by-8-row window. What this means is that the simulator will attempt to interpret the values stored at these memory addresses as ASCII characters and colors to print to the display.

Part 2: Moving Chess Pieces

Before starting to implement the movement for the Chess pieces, basic helper functions are needed to simplify the logic.

F. `(char, int) getChessPiece(short location)`

This function obtains information about the location square of the Chess board. If the square contains a

piece, the type of piece and the associated player is returned. If the square does not contain a piece, return ('E',-1).

Function parameter and return value summary:

- `location`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- *returns* : (piece,player) corresponding to the piece on the board. Returns ('E',-1) if the location does not contain a piece.

! ~~This function must call **`mapChessMove`**.~~ 2/28/20 @8:08am Requirement removed Piazza @525

! This function does not change the state of the MMIO.

Each Chess piece on the board moves in a different way. A separate function is required to validate if a particular move is valid. In this assignment, create 2 of these functions: `validBishopMove` and `validRookMove`.

validRookMove.

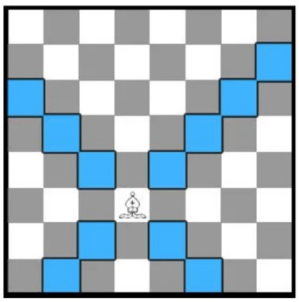
G. (int,char) validBishopMove(short from, short to, int player, short& capture)

> 🤔 This use of the **&** in the arguments is C-syntax to represent pass-by-reference. This notation means the argument is the "address-of" the datatype in memory. This is a way to pass information back to the caller function. The address passed could be on the stack (similar to the local variable space passed in the Lab #3 activity) or a global variable address.

Assuming there is a bishop at the `from` board position, this function determines if the bishop can successfully move to the `to` board position. A bishop may move to any square along a diagonal on which it stands. It can move any number of board squares in the selected direction.

The move is **invalid** if:

- the `from` and `to` positions are the same board square (no move performed)
- if any piece is obstructing the direct path between the `from` and `to` square
- if the player's own piece is in the `to` square



The move is **valid** if:

- the `to` square is empty and reachable by the piece's movement
- there are no pieces obstructing the movement
- if an opposing player's piece is in the `to` square. In this case, the piece is captured. The location of the captured piece (`to`) is stored into the memory address specified by `capture` to return the value to the calling function (since `$a` registers are not preserved over function calls).

Function parameter and return value summary:

- `from`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)

- `to`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- `player`: 1 for player 1, 2 for player 2
- `capture`: address in memory for storing the location of a captured piece (the `to` square). The 2-byte value represents the row and column for the square (see mapChessMove return type for format)

- *returns*: (status, piece if capture) where the status is:
  - (-2,'\0') for error with input arguments (invalid values for `from`, `to`, `player`)
  - (-1,'\0'), if the move is invalid
  - (0,'\0') if the move is valid and unobstructed.
  - (1, letter for piece captured), if the move is valid and a piece was captured.

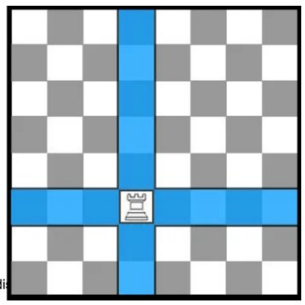
> ~~! This function must call **mapChessMove**.~~ 2/28/20 @8:08am Requirement removed Piazza @525
>
> ! This function does not change the state of the MMIO.
>
> ! This function only modifies the memory associated with `capture` if a piece is captured. Otherwise, it is unmodified.

Examples:

Case	Sample Board	Function Call	Return Value	Value in Capture
valid capture	game1	validBishopMove(0x0401,0x0603,2,0x400)	(1,'p')	0x0603
valid capture	game3	validBishopMove(0x0604,0x0200,1,0x400)	(1,'p')	0x0200
invalid movement	game1	validBishopMove(0x0401,0x0402,1,0x400) [3/2 @10:34am corrected in response to Piazza @574]	(-1,0)	unmodified
invalid movement	game1	validBishopMove(0x0401,0x0205,1,0x400)	(-1,0)	unmodified
invalid obstructed piece at to square	game1	validBishopMove(0x0002,0x0101,2,0x400)	(-1,0)	unmodified
invalid obstructed piece on path	game1	validBishopMove(0x0702,0x0500,1,0x400)	(-1,0)	unmodified
valid unobstructed	game1	validBishopMove(0x0401,0x0500,2,0x400)	(0,0)	unmodified
args error	any	validBishopMove(0x0800,0x0A00,3,0x400)	(-2,0)	unmodified

H. (int,char) validRookMove(short from, short to, int player, short& capture)

Assuming there is a rook (castle) at the `from` board position, this function determines if the rook can successfully move to the `to` board position. A rook can only move in a straight line: forward or backward (same column), or side-to-side (same row). It can move any number of board squares in the selected direction.



The move is **invalid** if:

- the `from` and `to` positions are the same board square (no move performed)
- if any piece is obstructing the direct path between the `from` and `to` square

- if the player's own piece is in the `to` square

The move is **valid** if:

- the `to` square is empty and reachable by the piece's movement
- there are no pieces obstructing the movement
- if an opposing player's piece is in the `to` square. In this case, the piece is captured. The location of the captured piece (`to`) is stored into the memory address specified by `capture` to return the value to the calling function (since `$a` registers are not preserved over function calls).

Function parameter and return value summary:

- `from`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- `to`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- `player`: 1 for player 1, 2 for player 2
- `capture`: address in memory for storing the location of a captured piece (the `to` square). The 2-byte value represents the row and column for the square (see mapChessMove return type for format)
- *returns*: (status, piece if capture) where the status is:
  - (-2,'\0') for error with input arguments (invalid values for from, to, player)
  - (-1,'\0'), if the move is invalid
  - (0,'\0') if the move is valid and unobstructed
  - (1, letter for piece captured), if the move is valid and a piece was captured

🤔 For those who play Chess, this is a simplified implementation. "Castling" is not considered.

❗ ~~This function must call **`mapChessMove`**.~~ 2/28/20 @8:08am Requirement removed Piazza @525

❗ This function only modifies the memory associated with `capture` if a piece is captured. Otherwise, it is unmodified.

❗ This function does not change the state of the MMIO.

Examples:

Case	Sample Board	Function Call	Return Value	Value in Capture
valid capture (assume rook at from)	game2	validRookMove(0x0702,0x0002,1,0x400)	(1,'R')	0x0002
valid capture	game3	validRookMove(0x0002,0x0502,2,0x400)	(1,'H')	0x0502
invalid movement	game1	validRookMove(0x0705,0x0604,1,0x400)	(-1,0)	unmodified
invalid obstructed piece at to square	game1	validRookMove(0x0700,0x0600,1,0x400)	(-1,0)	unmodified
invalid obstructed	game2	validRookMove(0x0700,0x0200,1,0x400)	(-1,0)	unmodified

invalid obstructed piece at to square	game1	validRookMove(0x0700,0x0600,1,0x400)	(-1,0)	unmodified
invalid obstructed piece on path	game2	validRookMove(0x0700,0x0200,1,0x400)	(-1,0)	unmodified
invalid obstructed piece on path	game1	validRookMove(0x0700,0x0400,1,0x400)	(-1,0)	unmodified
valid unobstructed	game2	validRookMove(0x0700,0x0704,1,0x400)	(0,0)	unmodified

valid unobstructed	game2	validRookMove(0x0705,0x0701,1,0x400)	(0,0)	unmodified
args error	any	validRookMove(0x0800,0x0001,-2,0x400)	(-2,0)	unmodified

Empty templates for the remaining Chess moves are provided. Set the return values of these library functions identical to those described above to test your implementation locally without having to create board states.

- `(int,char) validKingMove(short from, short to, int player, short& capture)`
- `(int,char) validQueenMove(short from, short to, int player, short& capture)`
- `(int,char) validKnightMove(short from, short to, int player, short& capture)`
- `(int,char) validPawnMove(short from, short to, int player, short& capture, char piece)`
  - if it is the first move for the pawn the piece is denoted with `'p'`. Otherwise, the piece is `'P'`.

Implement a function to tie all of these checks together and perform the player's move, if valid, and update the Chess board accordingly.

I. **`(int,char) perform_move(int player, short from, short to, byte fg, short& king_pos)`**

This function attempts to perform the requested move for a player, updating the MMIO chess board if the move is valid. If the move is invalid for the player, the function does not modify the MMIO and returns an error value.

One possible approach for the function:

Get the Chess piece at the `from` position.
- Return argument error if position is invalid.
- Return invalid if there is no piece at the `from` position.

If the piece is not for the specified `player`, return argument error.
Call `validMove` function for the piece type. Return error based on function return value.
If move is valid, then perform the move
- Set the `to` position on the board with the piece
  - NOTE: Set all pawn pieces in the board with `'P'` (moved from initial position)
- Clear the `from` position on the board
- If the king piece was moved, save the new board position into memory at `king_pos`
- Return success and the piece captured

Function parameter and return value summary:
- `player`: 1 for player 1, 2 for player 2
- `from`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- `to`: a 2-byte value which represents the row and column for the square (see mapChessMove return type for format)
- `fg`: foreground color to set the square to upon piece removal, eg `'E'`
- `king_pos`: address in memory for storing the location of the king piece. The 2-byte value represents the row and column for the square (see mapChessMove return type for format)
- *returns*: (status, piece if capture) where the status is:
  - (-2,'\0'), for error with input arguments (invalid values for `from`, `to`, `player`)
  - (-1,'\0') if the move is invalid