Group Project Report:

Video link: https://youtu.be/QudWS01zBhA

a. A program that runs a dot and boxes game against a predeveloped CPU. This game first generates a board for the user to see and use as they play. The board consists of several ASCII characters: -, |, u (user completed square), c (CPU completed square), numbers (1 - 6 for rows and 1 - 8 for columns), and space. The user is then prompted to make a move. This prompt first asks the user to enter the row number, then the column number, and finally the line direction (top, left, bottom, and right). After the user has made their move, the board is updated with the move that the user has made. Then, the CPU makes its move based on the first open position on the board. After every move, the program calculates to see if there are any closed squares in the board. If any exist, that center element of that square is changed from an empty space to the character that describes who closed that square. For example, if the user made the final move to close the square, the center of it should read "U". If the CPU closed it, the center should read "C". Also after every move, the program calculates to see if the user or the CPU has won more than half of the total available squares. If one of them has done so, the game is finished. The program prompts whether the user or CPU won the game.

b. Individual challenges: I hard a hard time making sure the files worked with each other properly and managing my time for working on this project, since I have exams to study for as well. Other than those two problems, I believe working on the project went pretty smoothly.
Team challenges:
   - Early on we had trouble dividing up the project into different areas that we can each cover
   - Managing time to spend on this project every week for the past 2 months
      - Coordination between moving parts in shared files that everyone is collectively working on. For example, one member may make changes that another member already added
   - Modular development
      - Organization of the code
      - It took a long meeting between the group just to come up with a good architecture and code design before we began actually writing the files
   - Error handling
      - Each member had to implement error handling in their asm files and since it's something we all learned relatively recently, we all found it a little challenging

c. I learned how to work with multiple files at once and how to manage files that link to each other and depend on each other. For example, if one of the files wasn't working correctly, the rest of the project couldn't work/assemble correctly; so I believe I have learned how to deal with programming in these situations.

d. Main:
    - Integers storing number of boxes each player has
    - Loop calling needed functions until win condition is met
        - Print the board
        - Get a player move
            - Validate, if not valid ask for another
        - Print Board
        - Get CPU move

This program:

- Initializes game variables, including the number of completed cells, player and CPU scores, and current player.
- Prints the initial game board.
- Enters a game loop that ends when all boxes on the board are filled.
- In each iteration of the loop, checks whose turn it is (player or CPU) and calls the corresponding function to make a move.
- If there was an error in the previous move, the current player gets another turn.
- If a box was completed on the previous turn, updates the score and gives the current player another turn.
- If a box was not completed on the previous turn, switches to the other player's turn.
- When the game is over, prints the final game board and scores, as well as a game over message.

Board:

- Print: loop through array of line and box status and print out

The program contains three functions:

1. print_board: This function prints the current state of the game board on the console. It iterates through each row and column of the game board and prints either a horizontal line, a vertical line, or a space character depending on the current state of the line.
2. check_line_state: This function checks the state of a given line on the game board. It calculates the index of the selected line in the horizontal or vertical lines array based on the row, column, and direction values, and then loads and returns the state of the selected line.
3. check_cell_completion: This function checks whether a move made by the player completes a cell and updates the player's score. It also checks whether the game has ended by counting the number of completed cells on the game board and comparing it to the total number of cells (48).

The program uses several algorithms, including:

1. Array indexing: The program uses arrays to represent the game board, and it calculates the index of a line in the horizontal or vertical lines array based on its row, column, and direction values.

2. Looping: The program uses loops to iterate through the rows and columns of the game board, to check the completion of each cell, and to count the number of completed cells.
3. Branching: The program uses conditional branching to check the direction of a selected line and to determine whether a cell has been completed or not.
4. Multiplication and addition: The program uses multiplication and addition to calculate the index of a line in the horizontal or vertical lines array based on its row and column values.
5. Load and store: The program uses load and store instructions to read and write the state of a line in the horizontal or vertical lines array.

Move:

- Prompt player for move
- Check if move is valid

This program implements the game loop for a game that involves drawing lines between dots to form a grid of cells. The program prompts the user for input to make a move in the game, and performs the following steps:

1. Prompts the user to enter the row and column of the cell where the line will be drawn, and the direction of the line.
2. Calls the validate_move function from the validation file to check if the move is valid.
3. If the move is valid, calls the update_board function from the board file to update the game board with the new line.
4. Checks if the move completes a cell and updates the player's score if necessary.
5. Returns from the function.

The program uses several algorithms, including:

1. System call: The program uses the MIPS syscall 4 and 5 instructions to print prompts on the console and to read user input.
2. Register manipulation: The program uses register manipulation to move data between registers and to set the arguments for the validate_move and update_board functions.
3. Branching: The program uses conditional branching to check whether a move is valid or not, and to determine the next action to take based on the result.
4. Subroutine: The program calls the validate_move, update_board, and check_cell_completion subroutines from the validation and board files.
5. Looping: The program implements the game loop by repeatedly calling the make_move function until the game is over.

Validation:

- Check if player move is valid. If so update board, else get player move again
- Check after each move if a box has been completed.
    - If box completed, update box array and increment player score

This program validates a move made by the player in the game of drawing lines between dots to form a grid of cells. The function validate_move checks whether a given move is valid or not, by performing the following checks:

1. Checking if the row, column, and direction values are integers.
2. Checking if the row value is between 0 and 5 inclusive.
3. Checking if the column value is between 0 and 7 inclusive.
4. Checking if the direction value is between 0 and 3 inclusive.
5. Checking if the selected line is already marked as true on the game board.

If any of the above checks fail, the program prints an error message and returns 0 to indicate that the move is invalid. If all checks pass, the program returns 1 to indicate that the move is valid. The program uses several algorithms, including:

1. Branching: The program uses conditional branching to check whether the row, column, and direction values are within the valid ranges, and to check whether the selected line is already marked as true on the game board.
2. Subroutine: The program calls the check_line_state subroutine to check the state of the selected line on the game board.
3. Load and store: The program uses load and store instructions to read and write the state of a line on the game board.
4. System call: The program uses the MIPS syscall 4 instruction to print the error message on the console.
5. Register manipulation: The program uses register manipulation to move data between registers and to set the return value of the function.

CPU:

- This program has a function computer_turn that selects a move for the computer player.
- The program includes move.asm, which likely contains functions to update the game state and validate moves.
- The computer_turn function begins by setting the row, column, and direction to 0, then enters a loop that iterates through all possible moves on the game board. For each move, it calls the validate_move function to check if the move is valid. If the move is valid, it returns from computer_turn with the selected row, column, and direction.
- The program uses three nested loops to iterate through all possible moves on the game board. The outer loop iterates through each row, the middle loop iterates through each column, and the inner loop iterates through each direction (top, right, bottom, left). This algorithm ensures that the computer player considers every possible move on the game board.
- Overall, this program implements a simple strategy for the computer player: it selects the first valid move it finds by iterating through all possible moves on the game board. A more advanced strategy might use heuristics or game tree search algorithms to select more optimal moves.

e. I believe each member equally worked on the project. Me (Kevin Pulikkottil), Joe Linden, Ben Willingham, and Yousuf Ahmad all did out part and helped each other out with everything. If I were to grade my team members out of 10 for this project, I would give them all a 10/10.
f. The only suggestion I would have is an easier game to create using MIPS assembly language, since this project was pretty hard.