Henry Nguyen

Professor Alice Wang

CS 2340

April 29, 2025

Multiplication Game Project

1. Description of the Program:

The Multiplication 4-in-a-Row Game is a one-player game implemented in MIPS. The player competes against a computer to mark numbers on a 6×6 multiplication board. The objective is to get four numbers in a row (horizontally, vertically, or diagonally).

The game uses ASCII characters to display the board and provides clear messages for each game event. Each game action (e.g., drawing the board, validating a move, checking win condition) is implemented as a separate module, in request with the assignment requirement.

**Modular File Purpose:**

**Main.asm:** Controls the game loop and coordinates player/computer turns

**Board\_data.asm**: Has board values, markings array, and game messages

**Draw\_board.asm**: Prints the current state of the 6x6 multiplication board

**Draw\_slider.asm**: Displayers the slider showing remaining/available numbers

**Player\_move.asm**: Handles user input and calls validation and marking procedures

**Validate\_move.asm**: Ensures selected moves are legal and have not already been taken

**Mark\_move.asm**: Updates the markings array to record moves

**Computer\_move.asm**: Implements a computer strategy with winning/blocking logic

**Check\_win.asm**: Scans the markings array to detect a win condition

1. **Challenges/Solutions**

**Modular Assembly Conflicts:**

In my earlier stages of coding, had a lot of problems including recursive errors and duplicate globals. I finally fixed them using the .extern and .include command properly, as well as turning on the assemble all files in assembly within Mars.

**Data Initializations (.space):**

I first used. space 41, but this led to garbage/blank values to appear when drawing the board since the memory was initialized. Fixed this my using a .word layout of pre-computed multiplications values. Much easier to see and control the board, ensuring the board positions had its correct initial values.

**Tracking marked moves:**

When I first tried to track them, it failed as I was trying to detect a win by scanning the board’s literal multiplications values. However, this led to incorrect results, because the same number can appear multiple times in different positions. The way I solved this was just to introduce a separate markings array that tracks the state of each board cell (shown as 0 = unclaimed, 1 = player, and 2 = computer). The win validation and move was moved to markings, which makes the logic code easier to work with and more reliable than my previous method.

**Winning AI Strategy:**

Although this is optional, my initial computer code was coded to just select random valid moves. However, in my effort for extra credit, I decided to take on the task to implement a winning strategy. The computer now checks each move and the current state of the board to see if it would result in 4 in a row or block the player. If there is no **immediate** thread or win exists, it results in the computer to choose a valid move.

1. **Lessons:**

**Modular Design:**

Although I had trouble with the .asm files being split and calling each other correctly, splitting the program into independent modules like draw\_board.asm, player\_move.asm, etc, helped me learn how important clean separation of concerns is. It allowed me to seriously focus on one task a time, test functionality in isolation, and learn how lower language works within multiple files.

**Working in MIPS:**

Unlike high-level languages and are other homework’s, this project really tested my knowledge on how MIPS works, requiring careful planning of memory layout, register usage, and control flow. Really taught me how to work in MIPS, breaking down complex logic like win checking or computer decision making into steps in assembly, also letting me know how in higher level languages what it requires on the lower level.

1. **Techniques/Algorithms:**

The board was made using nested iteration and array traversal.

Starting off with draw\_board, it had two arrays:

* The board array containing numbers and markings array (which holds cells).

The procedure loops across the board row by row and column by column, checking each index in the markings array for X’s and O’s. If it has an entry, it overrides board values, and if not, the board value is printed. This allows for dynamic updates without altering original multiplication board.

Some other techniques used was sequential searching and mapping across the two arrays. The algorithm that I implemented performed a linear search across the board array to locate the selected number by the user or computer. Once found, it maps the board index directly to the corresponding entry in the markings array to check for prior ownership.

Lastly, the final techniques that I believe that are used in the rest of the program (main) are linear search, index, state encoding, and conditional branching for role assignment, which can all be shown in mark move.

Linear Search: Uses another linear scan of the board to locate the product the player selected, matching the number to its exact index in the board array.

Index Translation: Once the number was found, the same index is used to access the parallel markings array, showing parallel array mapping – each cell has a value in the board and its respective owner in markings array.

State: Marks array with X for player and O for computer, storying them in the markings array with byte storage (marked in comments)

Conditional Branching: Player/Computer identity is passed as a parameter and used to decide whether to store its respective X or 0 in the array. Done by using conditional check structure using beq and bne to replicate an if else loop.