Aadhi Sivakumar

Alice Wang

CS/SE 2340.006

November 22, 2024

Project Report

For our term project, we replicated the “Math-match game”, in the “Multiply Easy” mode, inspired by the game on Math is Fun. This is a single-player game where the goal is to match multiplication expressions with their corresponding integer results as quickly as possible.

The original game display features a 4 x 4 grid consisting of 16 cells, each initially covered with a “?”. Above the grid, there is a counter for the number of unmatched cards and a timer that tracks the elapsed time since the start of the game. The player can then click on a card, to reveal its content. The player then selects another card. If the cards are equal to each other (e.g. 5 x 3 = 15), the cards remain flipped. If the cards are not equal to each other, both cards will flip back to the “?” side. The game continues until all cards are matched where it displays a message stating “Well Done! You finished in XX:XX”. This game also included sound effects for the card flipping and once all the cards were matched.

Our first step to programming this game in MIPS Assembly in MARS was to break the program down into multiple .asm files. Each .asm file covered a portion of the game, such as the creation of the board(board.asm), taking input from the player(input.asm), the overall game logic including the matching(game_logic.asm), the timer(timer.asm), and the main file(main.asm) where we call all these files. We first started off with creating the 4 x 4 grid which contains 16 cards, one side being a “?” and the other side being the multiplication expression/integer result in board.asm. In the .data section of the file, we defined the total number of cells(board_size) and a 64 byte space for the board. We also initialized a predefined list of all the expressions and results as well as the card_state where 0 is hidden and 1 is revealed. We used the card_state variable to track the flipping of the cards. Lastly, we defined the row separator which consisted of ascii characters to create the grid. The grid is created through a row and column loop using the row_seperator and the pipeline symbol. Each of the 16 cells in the grid are then initialized with a "?" or the actual value, depending on the card's state which also uses loops. To begin the game, they are hidden and initialized with a "?". We then generate a random index for each card with an upper bound of 16 and use the syscall for the random number to place the card in a random index. Next, we checked if the card index was already filled, and placed the card in another index if it is. We repeat this loop until all the grid indexes are filled with cards which contain an expression or a result.

Our second step to this project was taking in the user's selection for what card they want to flip in the grid. To accomplish this, we created an input.asm file. In the .data section, we had a prompt telling the user to enter an index of the card they wanted to flip, and some error messages if the card was not a valid input or if it was already revealed. We first had an input loop displaying the prompt message, taking in and reading an integer , validating the input range is between 0 and 15 using if statements, checking if the card is already revealed using its address, and restoring the saved return address from the stack if it is a valid input. If it is not a valid address, it then either goes to input_error or card_revealed_errror and displays a message.

Next, we incorporated the game/matching logic in game_logic.asm. This file consists of two main functions, check_match and is_board_complete. The check_match function compares two cards and makes sure the two indexes are valid. If it is, it then retrieves the values of the two cards from the board array in board.asm. The function then proceeds to check if either of the cards is an expression or a result by calling the is_expression helper function. If one card is an expression and the other is a result, the evaluate_expression_impl is called to compute the result of the expression, which is then compared with the integer on the other card. If they match, the function returns 1, indicating a match; otherwise, it returns 0. The is_board_complete function checks whether all cards in the card_state array are marked as complete. It iterates in a loop through the card_state array and makes sure all cards are non-zero(complete). This indicates when to stop the game.

Writing this file gave us many challenges that we had to overcome such as handling conditional logic and managing multiple checks for the card matching process. We needed to verify if the cards were valid, determine if one was an expression and the other a result, and ensure the correct matching logic was applied. This process involved numerous steps which we had to keep track of. To solve this issue, we used a series of conditional jumps, and managed the flow of control with branches. We also saved important registers on the stack to preserve state during function calls. Another big challenge we encountered was parsing strings and converting characters to integers so we could compare the multiplication expressions with the integer results. MIPS doesn’t have built-in string manipulation functions, so we had to implement a loop to process the expression, convert each character to a digit, and multiply the numbers. This process led to many overflow problems, which we managed by breaking down the expressions.

Our last and most difficult task for this project was implementing the timer in timer.asm. When we tried to create the timer to act as a stopwatch, where it updates the time each second, it printed a separate string each time. We tried using a carrier to not create a separate string each time, but it did not work. To resolve this issue, we had to settle on showing the elapsed time after each card flip, instead of a dynamic timer updating each second. To implement this timer, we stored the current system time as the starting time. Then after each card flip, we called the timer function and retrieved the current system time again. We then subtracted the stored start time to calculate the elapsed time in milliseconds. The milliseconds are then converted into seconds using the function display_time_in_seconds which uses a stack to efficiently retrieve the time. For formatting the time, I used the function print_two_digits. Another challenge of coding this timer was getting it to even run as well.

We finally pieced all these files together through main.asm. To get access to all these files in one file, we used include statements and global variables. We also implemented our counter variable here through the function: unmatched_cards_count. We broke the main.asm file into components that we wanted to display one by one. We first initialized the board and timer, then had functions to reveal the first card, second card, check if it was or was not a match, and if the board was complete. If the board's complete function was true, we exited the loop to leave the game and display the message "Well done, you finished the game in XX:XX".

Through this project, I learned how to compare strings with integers by breaking the string down to individual characters, converting each character into a digit, and multiplying the digit to get the digit representation of the string. I also learnt how to implement a running timer in MIPS, which I did not know how to do before as well as become more comfortable and understand fully how to work with Stacks on MIPS. I got to understand fully how memory allocation works and how data is stored in MIPS, either through static memory or dynamic memory.

My partner Kevin Sun contributed heavily to this project. He was responsible for the creation of the board, taking in the user input, and the counter for the unmatched cards. I was responsible for implementing the timer into our program. We were both equally responsible for implementing the game logic and writing the main file to connect all our files together.

A future suggestion for this project would be implementing the sound effects and graphics into our program. I actually wrote a python script to process a sound file, but I was unsure of how to incorporate it into MIPS.