**Description of the Program:**

This Lexathon game is an adaptation of the Android word game Lexathon. The goal of the game is to find as many words as possible given the letters that are provided in the 3x3 grid. The user will have 60 seconds at the start of the game, and they will be given an additional 10 seconds for every correct word they find.

**Challenges along the way:**

We as a team had a lot of challenges programming this project. A problem we had to overcome was the synchronizing of the code. There was a challenge finding the correct order of where the files and code had to sync. We also faced many challenges individually, such as getting the border to print correctly, the time counter and the overall flow of the game. This was a great learning experience for MIPS and I’m glad to be a part of this group.

**What I’ve learned from this project:**

There is a tremendous amount of knowledge that I have learned from this project. When it comes to MIPS, I learned how to use functions and calling correctly with jal and jump. I also learned how the time and scoring works in MIPS. I also learned about how stacks work by the use of printing out the grid.

**The algorithm and techniques used in the program:**

**IMPORTING.ASM:**

Random file name:

First, a random integer is produced from 0 to 8 (through syscall 42 command). The random integer will then be added with 65. 65 is the asci value for ‘A’. So 1 + 65 will equate to B, and etc. This new integer is concatenated with a string with value “.txt” The integer and string is concatenated by using a buffer to hold the words and concatenate accordingly. The concatenated string is stored in the variable “concat”. (i.e. “concat” can contain “A.txt, B.txt, ,,,”

Import File Function:

This will use the “concat” string to choose the file to download into a space named fileContents. All the files that are available to use start with a capital letter.

NEXT STEP in Importing:

After the importing of file, we want to be able to create a word list that starts with a nine character word and contains the corresponding possible combinations with that nine letter. It will first choose a random integer from 0 to 200000. This integer will be used as a random pointer that will point to a random character in the downloaded space.

Look For Word Function:

This function will look for the word following the random character. It will use the random character pointer and go through the file until it finds a newline character. With how our text files are set up, this means that the next character will be the start of a new word.

Look For Nine Letter Word:

This will look for the first nine letter word following the start of the new word we found in the Look For Word function. We will continue searching until we found an asterisk. This means that we have found a nine letter word. Every nine letter word starts with an asterisk. So we will move the pointer to the next character which will be the start of the nine letter word.

Make Valid Word List:

This function will create a word list of all of the characters until the next nine letter word or an asterisk. The functions works by saving all of the characters until an asterisk is found. Essentially we are starting the download from the start of the new nine character word to the next asterisk that is found.

**VALIDATE.ASM:**

This validate file is responsible for getting the user input for the game and checking to make sure it is a valid word, contains at least 4 letters, contains the middle character, and is not a duplicate.

Check Middle Function:

The key character is stored in a saved register. The function will loop through the user Input and check character by character until the key character is found. If it is not found, it will prompt the user that it not four characters and ask the user for a new word

Check Valid Word Length Function:

This will count the number of characters in the word that the user enters through a loop. It will have a counter that will count each time a character is loaded. It will branch at the end of the user input and check the counter with the valid length number. If it is less than the valid length number, it will prompt the user that it is not a valid length and ask the user for a new word.

Check Word For Validity and Duplicate Function:

This function utilizes two saved registers ($s6 and $s7) that will hold the position of the spaces checkingWords and validWords. The function will first load bytes from the start of validWords and the start of the user input all done in a loop. We know the end of the first valid word if we load a value that is less than 65. If that is the case, then this means that all of the characters of current valid word and user Input match up. Then we need to branch for further checking. The further check will read the last loaded byte from the user input. If the loaded byte is also a value less than 65, then this means this is the end of the user input. Thus, we will conclude that the user input word is a valid word in the list. We will then move to the duplicate function to check before adding the word to the list of correct words found by user (checkingWords).

Duplicate Function:

The duplicate function will check the user input with a space that contains the correct words the user has entered previously. It will start a loop that will go through the user input and the first byte of checkingWords, which is the space that contains the previously correct words. If there are any differences, it will jump and check the next word in checkingWords. If there are no more words, the user input will be determined not a duplicate and sent to be saved to checkingWords. However, at any time, if the user input and one of the words match character for character, it will jump and tell the user that a duplicate has been entered. The word will not be saved and then prompt the user for a new word.

Save Function:

After passing through the validity and duplicate function, this function will save the user input into the checkingWords space. It will add a newline at the end and then increment to point to next empty byte space for new possible word.

**timetrack.asm**

The time function starts with having timeleft, the variable that holds the time remaining for the seconds left for the player. The next step begins before a user input is called. The timeSeconds method is called, which gets the current system time and stores it in the label startTIme.

timeLeft:

This function makes a call to the timelapse function which will calculate the time difference between the system time when called and the start time. The result is then added in to $t1 with $0 in order to store. Then the LeftTIme is stored into $t2 and subtracted from $t1 to obtain the total time remaining. This value is stored back into leftTime. If this time becomes equal to 0 it will branch to the endGamePrompt. Then Time remaining is then printed. it will then jump back to validation

addTime:

When a correct word is validated a jump will happen to addTime which will load the timeLeft and add 10 seconds. and then update timeleft by 10 additional seconds. It will then jump back to validation.

**score.asm**

The score function begins by storing a score of zero into the score label.

mainScore:

when a correct word is validated a jump will happen to mainScore. MainScore will then load userInput, the validated user input, to $a0 and this will in turn be used to call the addScore function and jump back to validation

addScore:

this function initializes a count to 0 to find a count of the word length. Then it enter a loop which reads an individual char of the input word starting at index 0. It will update the count by one and the string pointer by one until it has found a null charcter at which point it will exit the loop. Once exited from the loop the previous score will be loaded in $t1 the count which remains in $t0 will be multiplied by 2 to receive the total points acquired. Then $t0 and $t1 are added to get the total number of points and then stored back into score and exit back to validation

**LexathonGridFinal.ASM:**

setKey Function:

The Set Key function begins by getting a random integer from 0 to 8 by use of the syscall 42 in MIPS. The random integer is then tested through a series of branch equal statements similar to a switch statement. The random integer then becomes the index for the character within the 9 letter word stored in nineCharacters. Then nineCharacters is loaded into register $t0, and depending on the random integer, the corresponding character within nineCharacters is loaded into $s0 which holds the key character for the grid. For example, if the random integer generated is 6, then the character in index 6 of nineCharacters would be loaded into $s0, which corresponds to the 7th character in the word stored in nineCharacters.

fillCharArray Function:

After setting the key for the grid, this function contains a loop that to traverse the nineCharacters array that holds the nine letter word. The exit condition for the loop is when the counter reaches 8. Within the loop there is a check to see if the current character in nineCharacters is equal to the key character. If they are equal, then the loop skips storing the character into a different array, otherwise the current character in nineCharacters is stored into toShuffleChars. toShuffleChars is the array holding the 8 characters besides the key character in nineCharacters that will be shuffled before being output to the grid.

printGrid Function:

This function prints out the grid by using asciiz strings and outputting them while using the stack pointer. Before each print statement, room is made on the stack to store the return address then sw is used to store the return address. Then using smaller functions defined within printGrid to print the borders and characters, we use jal to jump and link to the smaller functions such as printOuterBorder which is used to print the top and bottom borders of the grid. After returning from the function call, we then restore the stack pointer by use of lw. After restoring the stack pointer, we increment the stack pointer. This process is used for the entire printing of the grid.

shuffleCharactersInGrid Function:

A random integer is generated from 0 to 8 through the use of syscall 42. Save register $s1 holds the last random integer generated using the shuffleCharactersInGrid. This is to prevent the shuffle function from using the same shuffle twice in a row. There are 9 different shuffling orders that the shuffleCharactersInGrid function can perform. The way the order is determined is by branching to the matching shuffling order after checking to see what random integer was generated. For example, if the random integer was 0, then shuffle1 is called, if the integer is 1, then shuffle2 is called. The shuffle works by keeping having two registers ($t0, $t1) representing the index of toShuffleChars, which is the array filled by the fillCharArray function and the second register holds the index of shuffledChars, which is the array that the printGrid function uses to fill the grid. The registers $t0 and $t1 are initially set to 0. Depending on which shuffle is selected, an immediate ranging from 0 to 7 is added to $t0 and the corresponding character in toShuffleChars is loaded into $t2. The character in $t2 is then stored into shuffleChars and $t1 (the index of shuffleChars) is then incremented by 1. After storing into shuffleChars, $t0 is then reset to 0 again, and the process of adding an immediate is done again. This process is done 8 times before the shuffle jumps to the end and returns to where shuffleCharactersInGrid was called.

**Contributions of each member:**

**Aditya Buvanendiran – score and timer**

**Changho Lee – importing and validation**

**Kenny Hoang – printing and shuffling of the grid**

**Nick Ramos – input files**