Program #5

I. Problem Statement

The assigned task was to design, implement, and test a program that requests a file address from the user, opens the file, convert the hexadecimal values and operations to decimal values, then display the values to the screen. The program is designed to be an add on to the previous program # 4 and we are called to simulate the operation of the Microsoft calculator in programmer mode with hex input and dword or 32-bit element size. The operations that the calculator supports are + (addition), - (subtraction), * (multiplication), / (division), % (modulus), = (assignment, equals), s (memory save), r (memory recall), and z (memory clear, zero). The program utilizes one memory location, and this is initialized to zero. The value to be displayed is based off the last value read or the last value computed depending on whether the previous line was a value or operation. The program's operands, operations and their results are to be outputted to the screen. Within the newLine function, the file is passed in as the argument from the user's input. In this call, the contents of the file is looped through until the end of the file is reached and is then saved in a register. The final function is the convertValues function which is used to convert the hexadecimal values and their operations within the file to decimal values and to also print them to the screen. Flags are set up so that the program stops looping through the file after a specific number of spaces and to catch new-line, non-integer, non-uppercase, and invalid values. The program has a focus on the foundations of assembly language which include opening, reading from a memory (.txt) file, converting the hexadecimal value to decimal, then printing it back out to the screen in a fixed format. The significance of this program is to show a basic understanding of converting hexadecimal values, operators, operands, and utilizing files in the MIPS programming language.

II. Approach

The decision was made to produce a program that could request a file address from the user, open the file, read the contents, convert the hexadecimal values and their operations to decimal values, and finally print out the contents to the screen. In order to test this module, it was necessary to construct three separate functions, the main function, newLine function, and the convertValues function. The purpose of the main function was to prompt the user for the file address of the file they wish to open and print the file's contents. The newline function was implemented to carry out the step of reading the file contents, which the problem statement asks for. After the initial prompt, the user inputs their file address and the filename is stored in the $a0 register with 200 bytes of allocated memory and is later moved to the $s3 register. Next, the newLine function will be called with the task of removing the extra new line character from the user's file address input, which is important to correctly open and read from the file. After this step, the file is opened and the program continues by reading through the file, storing the contents in another register, with 200 bytes of allocated memory. The contents of this register is then displayed to the screen so the user can see the data of the file address they inputted. Finally, the convertValues function is called in order to convert the saved hexadecimal values and their operations from the file depending on the corresponding decimal value and print these values out

to the screen in a fixed format. The program was written using the MARS simulator IDE in the MIPS language.

In summary, the test allows the following steps:

1. Ask the user to input their desired file's address
2. Takes in the user's file as input
3. Calls the newLine function
4. Calls the convertValues function
5. Displays the respective outputs to the screen

III. Solution

The assigned task was to design, implement, and test a program that requests a file address from the user, opens the file, reads the contents, converts the hexadecimal values and their operations to decimal values, and writes them out to the display with their respective string outputs. First, the .data directive was used to ensure the constant and variable definitions were stored in the data segment. The .asciiz string directive was used to store the user's file address prompt, as well as numerous labels used for the input and output strings in memory and null-terminated them. Also, with the .space directive it was possible to reserve 200 bytes of memory for the variables called FileName and FileContents. Next, the .text directive was called in order to open a text segment for the assembly instructions, which for this program was the main function. This includes the load address and immediate functions which were used when displaying to the screen was required. The main function also called the newLine function in order to remove the appropriate unwanted new line characters from the user's input file address. Using the move, load address, and load immediate functions it was possible to move values between memory and the registers in order to display the file contents to the user. Another set of functions which were a major part of the newLine function was the j, jr, and jal operators that were used to switch from the newLine function to the main function. Along with those, the addi function was used to increase the loop counter variable to move through the contents of the file. Next, the beg and bne functions were used to check for the new line character or when the end of the user's input is reached. Within the convertValue function the branch less than unsigned (bltu) function was also used to conditionally branch to the instruction at the label if the contents of register was less than a specific value. Another function used within the program was the shift left logical (sll) which shifts a register value left by the shift amount, in this case 4, and places the result in a third register. The last functions which were used were the store byte (sb) and load byte (lb) functions, used to move information from registers and memory to successfully execute the correct output for the program. All three types of addressing modes were utilized in this program, register, immediate, and memory, in order to specify the location of the operands. Within the convertValues function many warning labels were used to check numerous test cases. The function begins by adding the values into a register. The end label was used to call an exit when the character value is equal to null or once the end of value is reached. The getTheOperator label was used in order to determines which operation is called based on the character value after the corresponding operation is found, the program jumps to that specific operation. This label

uses the branch on equal instruction reference which checks if the two registers are of an equal value. The isAdd operation is used to set the value of the register to 1, then prints that value to the screen and jumps to the operatorIsFound operation. The isSub operation is used to set the value of the register to 2, then prints that value to the screen and jumps to the operatorIsFound operation. The isMult operation is used to set the value of the register to 3, then prints that value to the screen and jumps to the operatorIsFound operation. The isDiv operation is used to set the value of the register to 4, then prints that value to the screen and jumps to the operatorIsFound operation. The isMod operation is used to set the value of the register to 5, then prints that value to the screen and jumps to the operatorIsFound operation. The isEquals operation is used to set the value of the register to 6, then increments the index value until a newline or null is found, then prompts an exit. Once the newline is found the program jumps to the equals operation. The operatorIsFound operation prints a newline, increments the index, then checks for the next character within the program, then when the character value is equal to null, exits out. The operandSearch label checks the character value by comparing it to values of null, which would call the end label, newline, which would call the operandFound label, and r, which calls the notR label. The label continues to read through and prints the readPrompt and increments the index value before jumping to the printValues operation. The notR label considers a few test cases. The notIntegerValue label, which checks if the value is an integer before continuing through the program. It uses the invalidValue label, which checks if the character is an invalid value, and the notUpperCase label to determine whether to continue or not. Next, the function utilizes the addValue label to add the values to the register total and continues to loop back through. Finally, the end label is used to call the $s6 register which contains the specific value and prints it to the display in a fixed format. The operandFound label prints the hexadecimal value in decimal and goes to the getTheOperator operation. The printValues label prints the result which is stored within the $s5 register, then calls the corresponding function based on their precedence. The operandFoundContinued label calls the equalSign label to print the statement saying the value is equal to, prints a new line, then loops through again. The addition label sets the total result equal to the result, prints the entire statement, then jumps to the operandFoundContinued label. The subtraction label sets the result equal to the result minus the operand, prints the entire statement, then jumps to the operandFoundContinued label. The multiplication label multiplies the operand, prints the entire statement, then jumps to the continueEquals label. The division label divides the result by the operand, prints the entire statement, then jumps to the operandFoundContinued label. The modulo operation divides the result by the operand, prints the entire statement, then jumps to the operandFoundContinued label. The equals label increment the index value, checks if this value should be saved before clearing it, then calls the continueEquals label. The continueEquals label resets the register values and restarts the process to get the first operand. The firstNotInteger label check if the value is a capital letter between ‘A’ and ‘F’ and checks if it is the firstInvalidValue and the firstNotUpperCase, then jumps to the firstAddValue label. The firstNotUpperCase label check if the character value is a lowercase letter between ‘a’ and ‘f’ then goes to the firstAddValue label. The firstAddValue label is used to add the value to the total number and goes to the getFirstOperand operation. The saveValue label saves the values to the $s6 register and prints the “Save: ”string followed by the contents of the saved value. The clearMemory label is used to clear the memory and to print the “Result: r” string followed by a

new line and to jump to the continuesEquals label. Finally, the firstOperandFound label which searches for the operand and prints the $s5 register based on the respective input. System call codes were also implemented at different times to ensure the program followed the specific guidelines established in the problem statement. System code 1 to print integer values, System code 4 to print string values, system code 8 to read the string values, system code 13 to open the file, system code 14 to read the file, and system code 16 to close the file when done using it. All these functions and variables within the .data and .text directives were all fundamental to complete the five steps asked to be accomplished in the problem statement.

Figure 1 File Contents

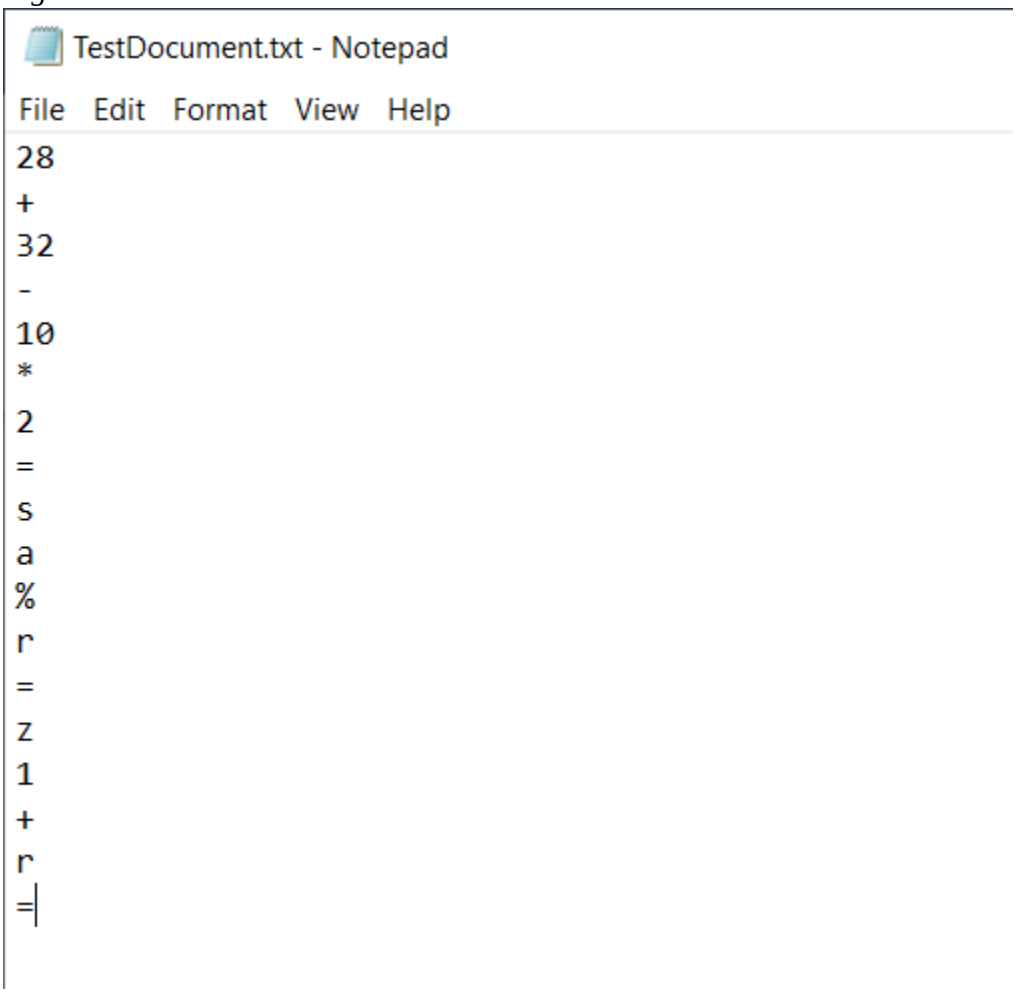

Figure 1 demonstrates the user's text file contents, which is the hexadecimal values and their operations to be converted and printed out to the display.

Figure 2 Synthesis and Simulation

```
Please enter the file name: C:/Users/Titus Varghese/Desktop/TestDocument.txt
Input: 40
Input: +
Input: 50
Result: 40+50 = 90
Input: -
Input: 16
Result: 90-16 = 74
Input: *
Input: 2
Result: 74*2 = 148
Save: 148
Input: 10
Input: %
Input: r
Read: 148
Result: 10%148 = 10
Input: z
Input: 1
Input: +
Input: r
Read: 0
Result: 1+0 = 1

-- program is finished running --
```

Figure 2 demonstrates how the user's file address is synthesized and simulated. Special flags were used during synthesis or simulation in order to ensure enough space was allocated for specific variables and to ensure only valid values were passed in to be converted. The output of the simulation shows that:

1. The program begins by displaying the prompt for the user to enter their file address
2. The program then gets the user's file and stores it into the $s3 register
3. During this step the program goes into the newLine function
4. Here the program goes into the convertValues function to convert the hexadecimal values and their operations and tests them against specific test cases
5. Displays the respective outputs for the hexadecimal values and their operations from the file to the screen in a fixed format