Programming Assignment, Pgm3

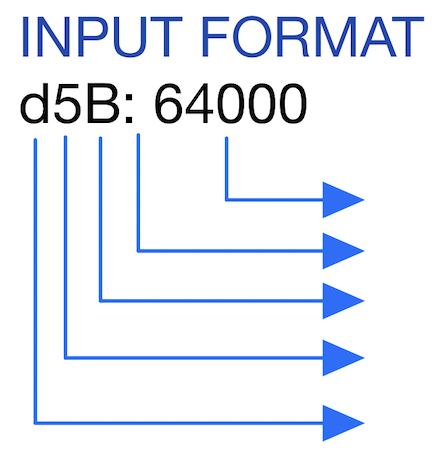
1. Problem Statement:

The third programming assignment, Pgm3 for the course Computer Science 2 (CS5330.501) is a MIPS Assembly Language program that is intended to open/read a text file, perform batch number conversion, and output the results in required format. Batch Number Conversion includes converting binary, decimal and hexadecimal values to one another. The programming assignment also includes bonus for implementing delimiters, space for each 4 characters of binary or hexadecimal numbers, and comma for each 3 characters of decimal numbers.

2. Approach:

The MIPS code for Pgm3 can be divided into three sections. The first section of the program is the *userInput* section in the main procedure. Here we get the file name from the user and call several procedures to do the remaining sections such as *inputOperations* and o*utputOperations*. The steps involved in *User Input* section are,

1. Prompt the user to enter the file name.
2. Use **openFile** procedure to remove the end of line character so the file is identified properly and then open the file using MIPS system call for reading file.
3. Once the file is ready for reading, a loop called **iterateFile** is started to read the contents of the file.



Input Value

Colon and Space

Output Type

Input length, could be 1 or 2 digits

Input Type

Figure Input Nomenclature

In the second section, inputOperations, the contents of the file are read one byte at a time. Each byte is processed by several procedures to store it in correct format based on the requirement to display content to the console later and to perform the conversion process. The several steps involved in this section are,

1. Separate variables are defined to store important values like input type, input value, output *type,* output value, sign flag for negative inputs, and the base of I/O values (Refer Figure 1 for the file content nomenclature).
2. Procedure **nextByte** is used to read the contents of the file one byte at a time.
3. The first byte read is the input type (b, d, or h), the value is then printed back to console followed by “: “(colon and space) as per program requirement.
4. The nextByte procedure is called again to read the next value, which is the input value length. The input value length could be either 1 character or 2 characters long. First character is read and if we find another number next to it (comparison by ASCII of 0 to 9), then it’s a 2-digit number, else, it’s a one-digit number. The digits are stored accordingly.
5. The byte after the input length is the output value, which is also stored for later use.
6. Next two bytes in the file are ‘:’ and ‘ ‘, which are skipped based on the suggestion from professor as they do not impact our program. Here, it is assumed that the input file has used the proper nomenclature shown in Figure 1.

If white space

Ignored and read until you get a relevant value

True

Character read

Added to stack

False

Figure 2: Flow of Operation of procedure nextValue

Return

Once we know the input type, length, and output type, we need to read the input values and do necessary cleaning. Procedure **nextValue** is used to read the next relevant value which is not a white space. White spaces like spaces, tabs, vertical tabs, new lines, carriage returns are ignored. Figure 2 shows the flow of operation for this procedure. Following are the steps after getting a relevant value from the file.

1. If the next non-white space value is minus symbol ‘-‘, then the input value is negative. Make the sign flag variable 1 to use it for later operations like 2’s complement. Read the next non-white space value and store it.
2. Procedure named **findBase** is used to calculate the integer base of input or output type. The integer bases are, b or B = 2, d or D = 10, h or H = 16.
3. Procedure name **convertInteger** is used to convert read ASCII characters to integers based on their input type for the ease of computation.
4. Once the first relevant value if read, multiply it with base to accommodate more digits from the file.
5. All these steps are repeated for the remaining non-white space values until the end of line. If end of line is reached, skip to next step.
6. The next step is to check if the input value is negative, if the sign flag has a non-zero value, it implies that the input value is negative. Negate the read input value to make it negative.

In the third section, *outputOperations*, the read input values are manipulated and converted into respective output values requested. The operations done in this section are still under the main loop iterateFile. The main aim in the section is to print binary and hexadecimal in 32 bits format, add delimiters (space or comma) for the output values based on the output type. The following are the steps involved in this section,

1. Clear the space defined for output value just in case so that it is empty.
2. Use findBase procedure, introduced earlier, to calculate the numerical base of the output type.
3. Copy address of output to the current address to work on it.
4. Check If the sign flag is non-zero, if yes, negate the output value for ease of computation.
5. Compute the number of times the **outputLoop** should run, if the output is binary it should run for 32 characters, similarly 10 for decimal and 8 for hexadecimal.

Once the necessary initializations are complete, we start the outputLoop. This is where the conversion of values and implementing 2’s complement of negative input values is done. The following steps are done in this loop until the iterator is done computing for the whole output length.

1. Divide the value by base value and store the Quotient and Remainder.
2. If the input value is negative and the output type is binary or hexadecimal, implement 2’s complement.
3. If the input value is positive or the requested output is decimal skip 2’s complement as the value can be represented with a negative sign in front. Implementation of 2’s complement is explained later in this section.
4. For hexadecimal value we need to compute both numbers and alphabets, number values from 0 to 9 are kept as it is, anything lager than that are converted to A to F.
5. If not hexadecimal or decimal, extra zeros are added to represent binary in 32 bits form.
6. If the base is 10, the decimal values are added with ‘,’ (comma) character for every 3 digits from right to left, done by dividing the value by 3 and checking for zero remainder.
7. If the base is 2, the binary values are added with ‘ ‘ (space) character for every 4 digits from right to left, done by dividing the value by 4 and checking for zero remainder.
8. Finally, if the base is 10 and the input value is negative (check sign flag), negate the output value to represent the decimal number as negative.

Implementation of 2’s complement is done similar to the way how it is calculated manually. First, check if the value is positive, if not negate it to make it positive. Subtract the positive value with 1 and then divide it by the integer base of output type (base already stored earlier). This gives the digits from least significant to most significant values as the remainder of the division operation. This value is then subtracted from the (base – 1) to get the final value.

The final step in output operation is print the computations in the required nomenclature. We have already printed the input type, colon-space, and input value in the input section of the program, now “; “(semicolon and space) is followed by that. Next, the output type is printed (B, D, or H), followed by “: “(colon and space). At last, the output value is printed followed by a new line character to mark the end of line. After the printing of each like the process program jumps back to the start of iterateFile loop to process the next line. Once the program finishes printing all the line from the file, close the file and terminate the program.

3. Solution:

The above-mentioned approach has been successfully coded and executed in MIPS assembly language. Figure 3 shows the successful completion of the assembling operation.

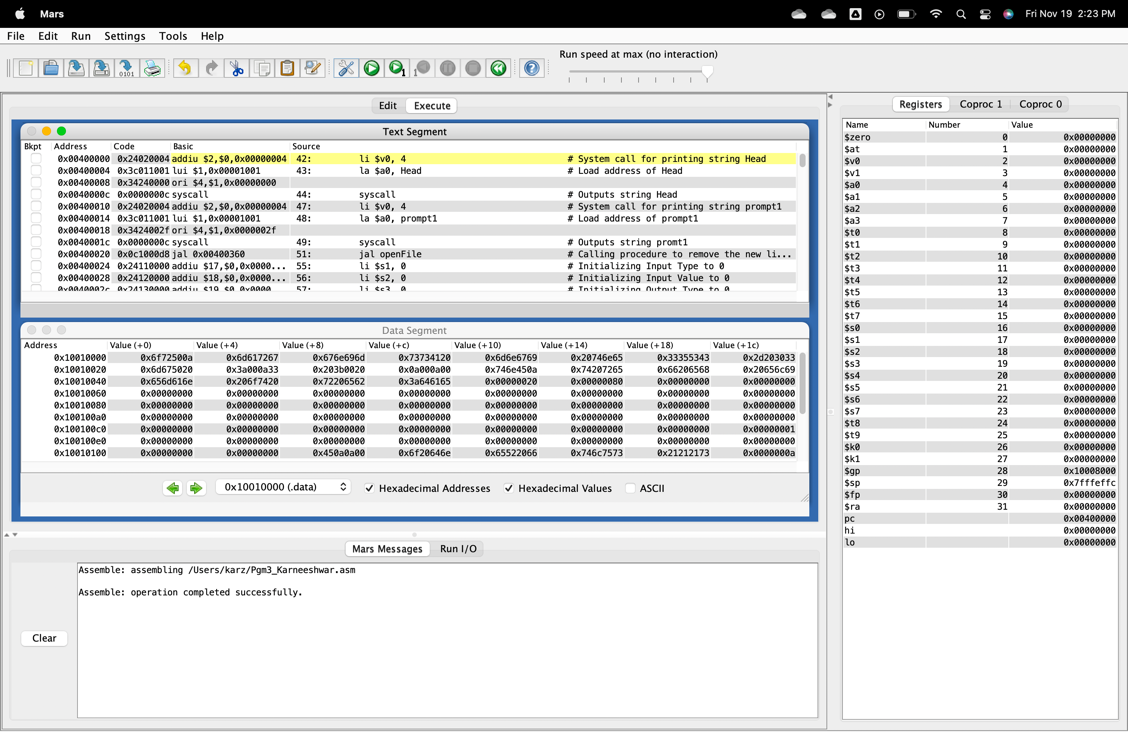


Figure 3: Screen capture of assembled code

To execute the program and check results, we need a sample text file. The file “test.txt” has been created and used for this purpose. The test file has several lines to test the program for all possible operations. The results of execution can be seen in Figure 4.

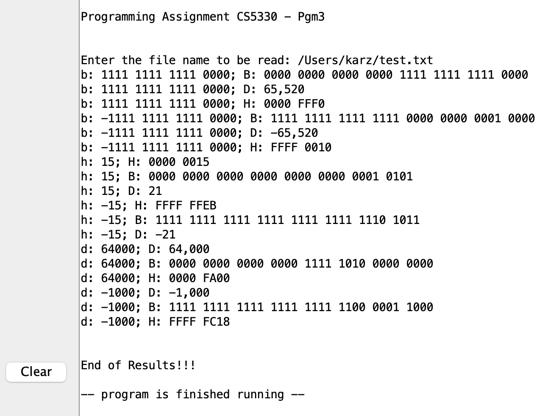


Figure 4: Results

4. Conclusion:

The given problem statement has been analyzed, and the required outputs were delivered using the MIPS programming language. The code has been created to accept a file name as user input and to perform several conditional and logical operations to calculate the required output. The results of these operations were displayed back to the user in a clear format as per the nomenclature.