T.C. YEDITEPE UNIVERSITY

CSE232 – Systems Programming

Term Project

Macro-processor for Assembly Language

Group 2

Mert Korkut

Muhammet Hakan Taştan

Ahmet Berke Kösretaş

Buse Karakaş

Umutcan Ağgez

5/29/2023

Spring 2023

Read

In this project, the **read()** function is responsible for retrieving macro definitions and storing them in a buffer, and also returning the number of macros detected. This buffer is a component of a data structure called **mac**, which is specified in the assignment. The mac structure comprises three elements. The first element is an array of type **'char'** with a size of 8, named **mname**, where the name of the macro is stored. The second element is a two-dimensional array of type **'char'** with 10 rows and 4 columns, named **param**. This array can hold a maximum of 10 parameters, each with a maximum character size of 3. Lastly, the mac structure includes another array of type 'char' with a size of 256, named **macro**. This array stores the macro body, which is the content between the macro definition and the "**#ENDM**" marker.

The **strtok** function, which is part of the **string.h** library, is utilized for tokenization of lines. This process involves splitting the contents of a line whenever a whitespace, tab, or new line character is encountered. The resulting tokens are then accessible using relative numbers.

Read function reads until the line with PROG keyword stored in it, which indicates that macro definitions are finished.

First, the file is opened in read mode. Then, a character named line is initialized to read the file line by line. Afterward, the presence of the given filename is checked.

The following steps are repeated.

- After tokenization, tokens are checked to see if they contain **PROG** or **NULL**. If it is PROG, we stop executing. If it is NULL, we go back to the start of the loop and takes a new line.
- If the token is not NULL, we check if first character of the token is '**#**' to determine if it represents a macro definition to continue. Else, we get new lines until we reach a definition.
- When '**#**' is found, again we do a security check to see if it is **#ENDM**. If so, program skips it and fetches a new line.

- Whenever we find a macro definition, we store the characters between '**#**' and '**:**' to **mname** variable. Then, we get the next token.
- We compare the next token with string **MACRO** to see if it is indeed a macro definition. If the comparison does not yield true, the program breaks out of the current process and resets the parameter name for further operations.
- Upon reaching the parameter part, it is assumed that the parameters are only separated by commas, with no whitespace in between. To handle this, a new token type called **commaToken** is introduced. This token is created specifically to check if the current token contains a comma and will replace the token for the following loop.
- Within a for loop, another token is initialized to tokenize the commaToken and subsequently store the individual parameters one by one into the **param** part of the "mac" structure. This process allows for the extraction and storage of each parameter separately.
- After completion of previous steps, it is time to fill body of the macro. All the lines until an '**#ENDM**' is reached will be added to the macro part of the structure. Function named **lineTrimmer()** removes any leading or trailing whitespaces to ensure easier to check character-by-character checking and guarantess there is no additional content at the end of the line. Additionally, after processing each line, a new line character is appended to differentiate between lines for future reference.
- When the previous step is finished, we increment the macro count variable and are ready to go the beginning of the loop.

When the loop is finished for good, we close the file that is read and return the macro count variable.

Parse

The parse() function in the program is responsible for analysing each line individually. It operates by parsing the current line into fields, which are then stored in a data structure called **'field'.** This structure consists of a **'char'** array with dimensions of 10 rows and 7 columns. This implies that every line can contain a maximum of 10 fields, with each field capable of accommodating up to 6 characters, with the final character being the null terminator. Each line is broken down into distinct elements, separated by whitespace or symbols. The parsing mechanism ignores any macro definitions found at the beginning of the program file and focuses solely on the lines following the 'PROG' line which indicates the start of the program. It is worth mentioning that the **'field'** data structure is defined globally, allowing other functions that require access to the **'field'** to utilize its contents. Furthermore, it is important to note that the 'field' data structure is reset to an empty state each time the parse() function is called. This deliberate clearing of the 'field' ensures that any previous data or remnants from previous parsing operations are completely removed, creating a fresh starting point for each new parsing operation. The function operates with only the filename as input and is responsible for determining the current line it is processing independently. It accomplishes this by utilizing a global variable named 'current_line'. Furthermore, a helper function called 'lineTrimmer' is employed to remove any unnecessary whitespace characters present in the lines before parsing them.

The function initiates by opening the specified file when it is executed. Subsequently, it proceeds to skip lines until it encounters the 'PROG' line, which signifies the start of the program. Once the 'PROG' line is located, the function parse commences to determine the current line. Once the current line is identified, the parsing operation begins. Initially, the line is tokenized into individual tokens based on specific delimiters. This process is accomplished by using the 'strtok' function from the string library. After tokenizing the line, the parsing operation proceeds to examine the first field to determine if it is a macro. This check is crucial for identifying and handling any macro-related components present in the line. Additionally, the parsing operation distinguishes between a normal macro and a special "if" macro during this examination. By differentiating between these types of macros, the parsing operation can appropriately process the field. After examining and processing the first field, the parsing operation proceeds to parse the remaining fields in a similar manner, ensuring that each field is handled appropriately accordingly. After concluding the parsing operation, the function completes its execution successfully by closing the file.

Report of IS_MACRO() Function

Introduction:

is_macro() function is the function that checks the “**field[0]**” which is the first element of the current line that parse has completed, for a presence of macro calls or “**#if**” conditioned macro call. And if it finds a macro it calls “**expand()**” function and if it finds a “**#if**” it checks the condition and if the condition is true it calls “**expand()**”. If the condition is false, is_macro skips that line. And if the line does not contain any macro or “**#if**” then the line is printed to “**filename.asm**” as it is.

Detailed Report

In the beginning of this function. It is checked that whether or not the “**filename.asm**” is empty or not. And if it is empty, “**PROG**” is added to the first line and then “**field[0]**” is checked from “**buffer[ii].mname**” (ii is the number of macros which is added to the buffer structure). The cases are defined as “**macflag**”, “**ifflag**” .Both of them are defined as 0 in the beginning. “**macflag**” is the flag that determines the “**field[0]**” as a macro. “**ifflag**” is the flag that determines the “**field[0]**” as “**#if**” condition. And the third case which is the default case “**(macflag==0&&ifflag==0)**” determines that the line is not a macro and not a condition.

When “**macflag==1**” is encountered “**expand()**” function is called to expand the macro.

When “**ifflag==1**” is encountered, in “**field**” array the first 3 elements hold “**#if”, “$cond1” and** “**cond2**”. The condition needs to be checked. The “**$cond1**” is the number of the argument to be checked (cond1 can be i=(0, 1, 2, 3, 4, 5) which denotes argv[i])

And an if condition will check whether ot not “**(argv[cond1]==cond2)**” is true.

If the “**if**” returns false, nothing will be printed to the “**filename.asm**”. So when “**if**” is false the line is essentially skipped. And “**if**” is true we need to do some editing to the current line to send the macro to “**expand()**” function clearly.

A temporary array is created to hold the macro name and the parameters. The first 3 elements of field are overwritten using temporary array and the remaining non overwritten elements are set to “\0”. This recreates a new **“field[]”** line that the **“expand()”** function can work with. And the the **“expand()”** function is called.

If both macflag and ifflag is still “0” then there is no macro on the current line and the **“filename.asm”** file is opened with **“a+”** permission which allows only to write to the end of the file. And the current line is written there and the function is complete until it is called again in the next line.

EXPAND

The requested function called “expand()” is responsible for expanding macros in a program. In this implementation, it reaches this aim by iterating a loop, processing each line of the macro body and generating the expanded code. After that, this expanded code is written to a file.

In this implementation, first, function calls “createPT()” function to obtain a parameter table which stores information about the macros such as their names, their number of parameters and the corresponding dummy and actual parameter values. After that, it opens a file in "a+" (append and read) mode with a filename derived from the input file. This allows the function to append the expanded code to the existing file or create a new file if it doesn't exist. The function then enters a loop, iterating a maximum of ten times. In each iteration, it checks if the macro name in the first field matches any of the macro names stored in the buffer. If a match is found, the function proceeds to expand the macro. For each line of the macro body, the function processes each character individually. It builds a new string, ‘newStr’, by replacing any dummy parameters with their corresponding actual parameter values obtained from the parameter table. To accomplish this, the function searches for dummy parameter names within the line and substitutes them with the actual parameter values.

Once the expanded line is constructed, the function writes it to the file. If a tab character is encountered in the expanded line, it is replaced with a newline character to format the code properly. This ensures that each line of the expanded code appears on a new line in the file. After processing all the lines of the macro body, the loop proceeds to the next iteration, if available, to expand other macros in the buffer. Finally, the file is closed, completing the expansion process.

In summary, this function is to automate the expansion of macros by replacing their dummy parameter values with the appropriate actual parameter values. It reads the macro definitions, generates the expanded code, and writes it to a file. This enables the further processing or compilation of the expanded code.

CREATEPT

The function begins by initializing several variables and data structures that will be used throughout the execution.

The variable i is declared and initialized to 0. It will serve as an index to iterate over the macro definitions in the buffer array.

Additionally, variables j and a are declared to be used within the loops later on.

A data structure called PT (Parameter Table) is declared to store the macro information. It contains the following fields:

mname: A character array to store the name of the macro.

nparams: An integer to track the number of parameters.

dummy[ ]: A two-dimensional character array to store the names of dummy parameters.

actual[ ]: A two-dimensional character array to store the values of the actual parameters.

Finding Macro Call:

The function enters a while loop that continues until the index i is less than the number of macro definitions (m_count) and the mname field of the current buffer entry is not empty ('\0').

Within the loop, it compares the first field (field[0]) with the mname of the current buffer entry using the strcmp() function.

If a matching macro call is found (i.e., strcmp() returns 0), the function proceeds to fill the Parameter Table (PT) with the relevant information.

If the macro has parameters, it enters another loop with a variable j initialized to 0.

Inside this loop, it copies the j-th parameter of the current buffer entry to the corresponding position in PT.dummy[ ] using strcpy().

The function also increments PT.nparams to keep track of the number of parameters.

Filling Actual Parameters:

After handling the dummy parameters, the function moves on to fill the actual parameters from the parsed fields.

It declares another variable j and initializes it to 1 to start from the second field (field[1]) since the first field contains the macro name.

It enters a loop that continues until the current field is not empty (field[j][0] != '\0') and j is less than 10 (the maximum number of parameters).

Inside the loop, it copies the j-1-th field (field[j]) to the corresponding position in PT.actual[] using strcpy().

This process ensures that the actual parameter values are stored in the Parameter Table.

Printing the Parameter Table (PT):

Once the Parameter Table (PT) is filled, the function proceeds to print its contents for debugging purposes.

It starts by printing the macro name (PT.mname), the number of parameters (PT.nparams), and the dummy parameters (PT.dummy[]).

It then enters a loop to print the actual parameters (PT.actual[]).

Inside this loop, it declares a variable a and initializes it to 0 to iterate over the characters of the current actual parameter.

It enters another while loop that continues until the current character (PT.actual[j][a]) is not '\0'.

Inside the loop, it prints the character (PT.actual[j][a]) using printf().

It increments a to move to the next character.

If a reaches the maximum parameter size of 4, it means that four characters have been printed, and it resets a to 0 and breaks out of the inner while loop.

After printing the current actual parameter, it prints a space using printf() to separate the parameters.

This process continues until all actual parameters are printed.

In the Main:

The createPT() function is called within the code block. This function is responsible for creating the Parameter Table (PT) based on the current macro call and the parsed fields.

ADD MACRO:

The macro takes three parameters: DEST, ADR1, and ADR2.

DEST: Represents the destination address where the result of the addition operation will be stored.

ADR1: Represents the first address to be added.

ADR2: Represents the second address to be added.

LDA ADR1:

This instruction loads the value from the address specified by ADR1 into the accumulator.

ADDA ADR2:

This instruction adds the value from the address specified by ADR2 to the value in the accumulator.

STA DEST:

This instruction stores the value in the accumulator into the address specified by DEST, effectively storing the result of the addition operation.

Overall:

When the ADD macro is used, it performs the following steps:

Loads the value from the address specified by ADR1 into the accumulator.

Adds the value from the address specified by ADR2 to the value in the accumulator.

Stores the result of the addition operation into the address specified by DEST.

SUB MACRO

The asked macro is a simple macro that performs subtraction of two numbers and stores the result. This macro has three parameters are, ‘DEST’ which stands for the memory location of the result number, ‘SRC1’ which stands for the memory location of the first number, and ‘SRC2’ which stands for the memory location of the second number.

In the second line of the macro, I loaded the first number to the accumulator by ‘LDAA’ command. Next, I subtracted the number in memory location ‘SRC2’ from the accumulator which is holding the ‘SRC1’ value currently. After this subtraction, the result number is stored in accumulator A and I stored that result number in the memory location of ‘DEST’ by typing the command ‘STAA DEST’.

To sum up, the SUB macro simplifies the process of performing subtraction by encapsulating the necessary instructions within a reusable macro definition.

MULT

This macro is structured to execute multiplication functions. It requires three inputs: D, A1, and P. The MULT macro accomplishes its task by multiplying a value present at a memory location given by the second parameter (A1) by 2 to the power indicated by the third parameter (P). The resultant product is subsequently stored in the memory spot indicated by the first parameter (D). This is accomplished by performing an arithmetic left shift on the value in A1, P times.

DIV

The macro is designed to perform division operations. It takes three arguments: D, A1, and P. The DIV macro divides a number stored in a memory location specified by the second parameter by a power of 2 specified in the third parameter. The resulting quotient is then stored in the memory location specified by the first parameter. This achieved by performing arithmetic shift right on the value stored in A1 by P times.

REM

Performing modulo operations using powers of two is equivalent to performing simple bit-wise operations. By subtracting one from the power of two and applying a bit-wise **AND** operation with the resulting value and the target value, you can obtain the modulo of the target value.

In the macro, this is achieved by a loop in which the value one is arithmetically shifted left to get the power of two specified by the parameter **P**. Then, this value is decremented by one and applied as the operand in a bit-wise AND operation with the target value **A1**. Lastly, we store the result in the destination determined by **D** parameter.