Assignment-7

Submitted by:
Name: Mohammed Junaid Mulla
Batch: D2
Roll No.:324042
PRN No.: 22111045
Division: TY D

Aim: Design suitable data structures & implement pass-I of a two-pass Macro processor

Objective:

1. Understand Macro Processing: Develop a clear understanding of macro processors, specifically focusing on how they facilitate code reuse and simplification in assembly language programming by allowing programmers to define reusable code blocks.

2. Implement Pass I of a Macro Processor: Successfully implement the first pass of a two-pass macro processor, which involves scanning the source code to identify macro definitions and construct the Macro Name Table (MNT) and Macro Definition Table (MDT).

3. Data Structure Design: Design appropriate data structures to store macro definitions efficiently, ensuring that macro invocations can be resolved quickly and accurately in the second pass.

4. Error Handling and Validation: Implement robust error handling and input validation to ensure that the macro processor can gracefully handle syntactical and logical errors in the macro definitions and other parts of the assembly code.

Theory:

A macro processor is a software utility that enables the expansion of shorthand notations (macros) into set sequences of instructions or data structures. Primarily used in assembly language programming, macro processors facilitate code reuse and manageability by allowing programmers to define repetitive code once and use it multiple times throughout a program. The macro processor translates these macros into their underlying instructions before the code undergoes further compilation or assembly. This preprocessing phase helps in reducing coding errors and enhances productivity by abstracting complex code sequences into simple, readable identifiers.

Definition and Purpose

A macro in the context of assembly language programming is a named block of code that can be defined once and expanded in multiple places within a program. Macros can be parameterized, allowing for versatile adaptations of the block depending on the input arguments. The macro processor handles the definition, expansion, and integration of these macros into the main body of code.

Components of a Macro Processor

A macro processor generally operates in two phases:

Pass I (Definition Phase):

Identification: It scans the source code to identify macro definitions. Each macro starts with a keyword like MACRO and ends with MEND.

Macro Name Table (MNT): It records each macro's name along with additional metadata such as its starting and ending points in the Macro Definition Table (MDT).

Macro Definition Table (MDT): It stores the actual code of the macro body. Each entry corresponds to a line of macro definition, preserving the sequence for accurate expansion.

Argument Processing: Any parameters within the macro are identified and placeholders are created for them, to be replaced with actual arguments during the expansion phase.

Pass II (Expansion Phase):

Macro Invocation: During the second pass, each invocation of a macro in the source code is replaced by the macro's actual code from the MDT. Parameters are substituted with actual values provided in the macro call.

Integration: The expanded code is integrated into the main code stream, replacing the macro calls. Importance of Macro Processors

Macro processors are vital in systems programming, embedded systems, and applications where hardware-level control and optimization are crucial. They allow developers to write more maintainable and error-free low-level code by abstracting repetitive patterns into single-line macro calls. Moreover, macro processors can significantly reduce the size of source code files and simplify complex programming constructs, making them easier to understand and manage.

Code:

```
#include <iostream>
#include <fstream>
#include <string>
// #include<unordered_map>
#include<vector>

using namespace std;


vector<string> MDT;

class MNT_class{
    public:
    string macro_name;
    int arg_num;
    int start_index;

    MNT_class(string macro_name,int arg_num, int start_index){
        this->macro_name = macro_name;
        this->arg_num = arg_num;
        this->start_index = start_index;
    }
};
vector<MNT_class> MNT;


int main(int argc, char *argv[]) {
    if (argc != 2) {
        cout << "Usage: " << argv[0] << " <input_file>" << endl;
        return 1;
    }


    ifstream inputFile(argv[1]);


    if (!inputFile) {
        cout << "Error opening file: " << argv[1] << endl;
        return 1;
    }


    string line;
    bool macro = false;
    int macro_line_count=0;
    int arg=0;
    string macro_name;
```

```
    while (getline(inputFile, line)) {
        string word;
        // cout<<line<<endl;
        for(int i=0;i<=line.length();i++){
            if(line[i]==' ' || line[i]==',' || line[i]=='\0'){
                if(word=="MACRO"){
                    macro=true;
                    word.erase();
                    continue;
                }
                if(macro==true && macro_line_count==0 && word[0]=='&'){
                    arg++;
                    if(line[i]=='\0'){
                        macro_line_count++;
                        // cout<<"in"<<endl;
                        MNT.push_back(MNT_class(macro_name,arg,MDT.size()));
                    }
                }
                if(macro==true && macro_line_count==0 && word[0]!='&' &&
word!=""){
                    macro_name = word;
                    // cout<<"Macro name:"<<macro_name<<endl;
                }

                if(macro==true && macro_line_count>=1){
                    do{
                        getline(inputFile,line);
                        MDT.push_back(line);
                        // cout<<"Macro Next Line: "<<line<<endl;
                        macro_line_count++;
                    }while(line!="MEND");
                    macro=false;
                    arg=0;
                    macro_line_count=0;
                    macro_name.erase();
                    word="MEND";
                }
                // cout<<word<<" ";
                word.erase();
            }else{
                word += line[i];
            }
        }
        // cout<<endl;
    }

    cout<<endl<<"MNT:"<<endl;
    for(auto itr=MNT.begin();itr!=MNT.end();itr++){
```

```
        cout<<"Name:"<<itr->macro_name<<" Arguments = "<<itr->arg_num<<" Start
= "<<itr->start_index<<endl;
    }

    cout<<endl<<"MDT:"<<endl;
    for(auto itr=MDT.begin();itr!=MDT.end();itr++){
        cout<<*itr<<endl;
    }

    inputFile.close();

    return 0;
}
```

User Input: Text file containing the following assembler code.

LOAD A

STORE B

MACRO

ADD1 &ARG

LOAD X

STORE &ARG

MEND

MACRO

ADDS &A1, &A2, &A3

LOAD C

STORE D

ADD1 5

ADD1 &A1

LOAD &A2

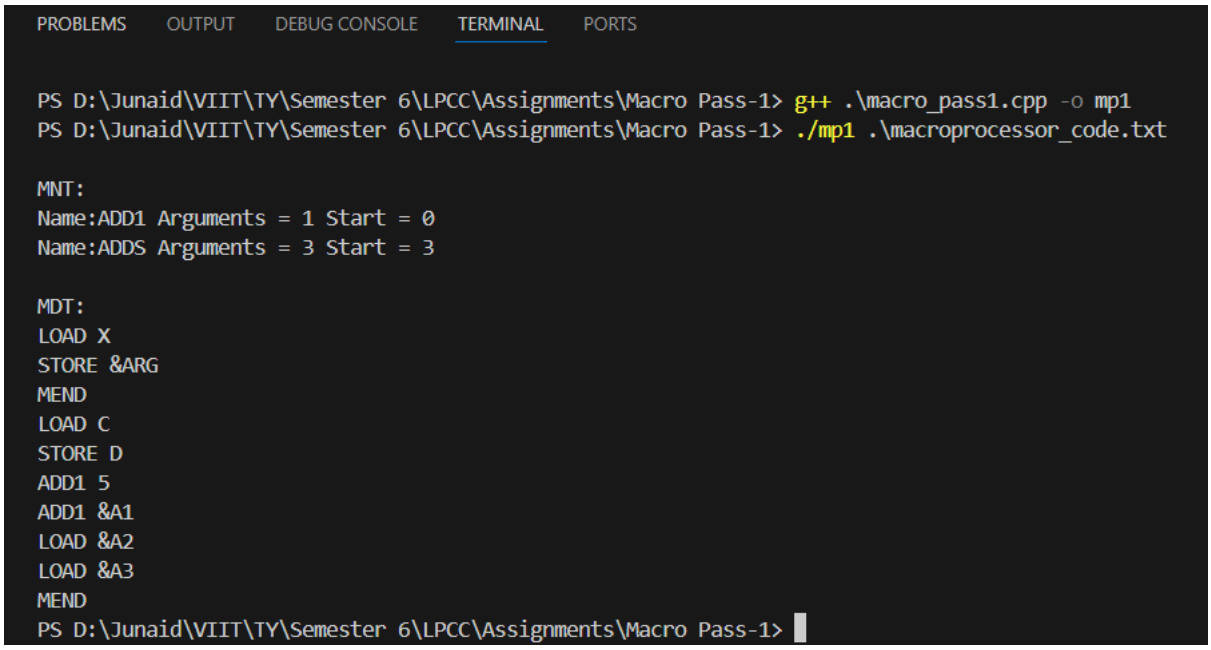
LOAD &A3

MEND

ADDS D1, D2, D3

END

Output:



Conclusion:

Completing this assignment on designing and implementing Pass I of a two-pass macro processor has provided valuable insights into the mechanics of macro processing in assembly language programming. It reinforced the importance of abstraction and code reuse, which are crucial for efficient programming and maintenance. Through the practical application of designing data structures such as the Macro Name Table (MNT) and Macro Definition Table (MDT), this assignment has enhanced understanding of how compilers and preprocessors handle repetitive code, optimizing programming efforts and execution efficiency. Furthermore, it has developed skills in problem-solving and software design, essential for tackling complex programming challenges in future projects. This experience has laid a solid foundation for further exploration into compiler design and other aspects of system software development.