SYSTEM PROGRAMMING PROJECT REPORT

Topic (Problem -3) - **Design of a 8086 Simulator/Assembler which supports macro-processor using C language following the working principle of Two Pass Assembler**

Team Members
Group 2 (UG3 BCSE A3)

Tonmoy Biswas - 002110501133
Tuhin Saha - 002110501087
Sailik Pandey - 002110501132
Ritish Dubey - 002110501089
Santimoy Tantubay - 002110501131

Theory

Two Pass Assembler

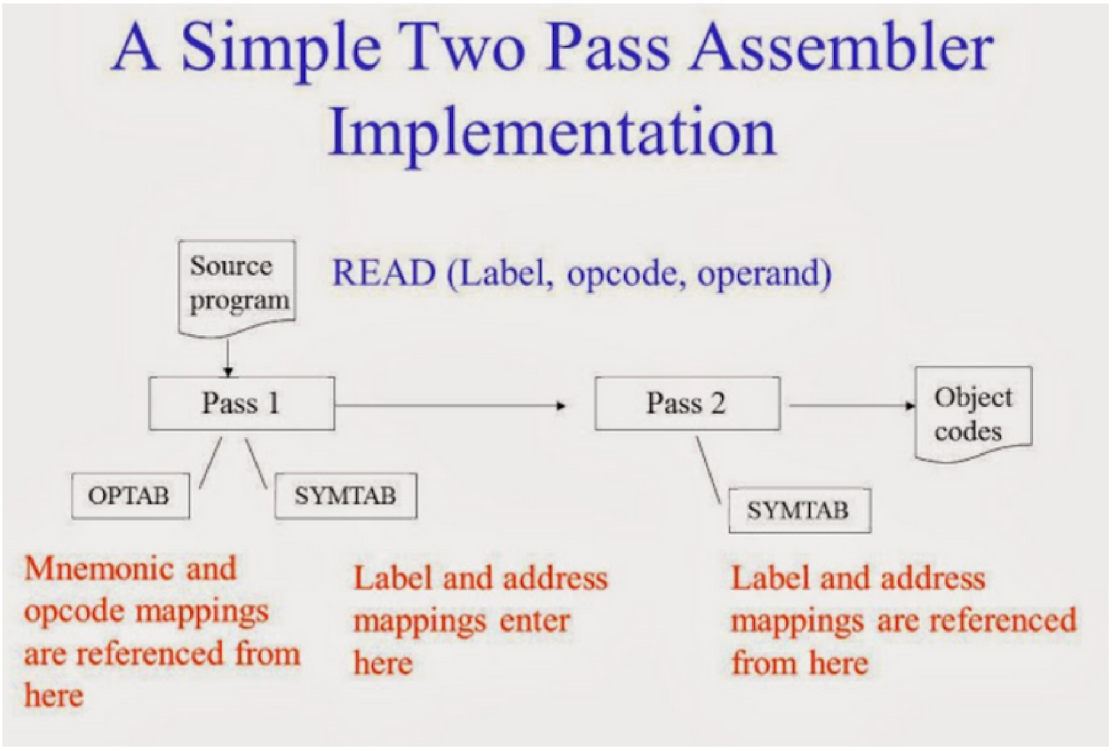
First Pass:

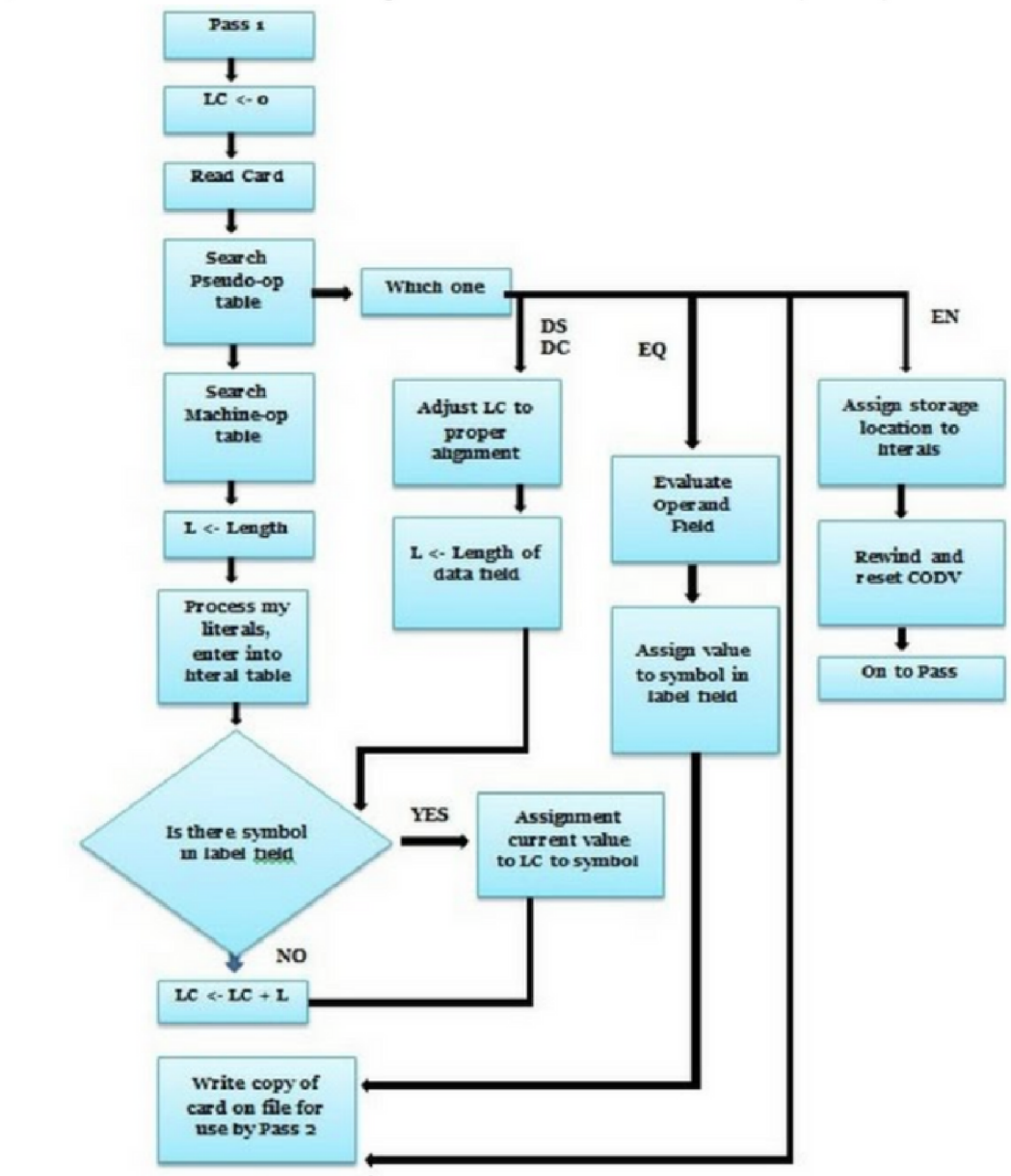
- **Lexical Analysis:**
  - Tokenize the input source code.
  - Identify keywords, symbols, and literals.
- **Symbol Table Creation:**
  - Build a symbol table to store information about labels and variables.
  - Assign addresses to labels and reserve space for variables.
- **Macro Expansion:**
  - Process and expand macros defined in the source code.
- **Intermediate Code Generation:**
  - Generate intermediate code based on the processed source code.
  - Update the symbol table with the addresses and sizes of instructions and data.

Second Pass:

- **Intermediate Code Translation:**
  - Use the symbol table from the first pass to generate the final machine code.
  - Resolve addresses and handle relocation.
- **Output Generation:**
  - Generate the final executable code or assembly listing.
  - Produce any necessary output files.

Two pass Assembler Flowchart





File Structure

- **input.asm** - Input assembly language program
- **output.txt** - Output text file
- **macro_processor.cpp -** This C++ program implements a two-pass macro assembler for processing assembly code with macro definitions. In the first pass, it reads an assembly file ("input1.asm") and identifies macro definitions, storing them in a map. In the second pass, it replaces instances of macros with their corresponding definitions, considering formal and actual parameter mapping. The final output is written to a new file ("output_1.txt"). The program includes functions for reading macro definitions, mapping parameters, and handling macro replacement. The main function orchestrates the two-pass process and demonstrates the capability to preprocess assembly code containing macros.

Fundamental Approach

First pass

We are observing the macros declared throughout the input.asm file and storing it in map (macro name , macro body) data structure.

Second pass

We are replacing the code of each macro to those places where the macro is actually being called.

In the end we re-writing the file by file stream method after replacing all the macro codes.

Detailed Explanation

Structure of macro

```
struct macro_def{
    string macro_name;
    vector<string> para_list;
    vector<string> definition;
};
```

First pass

```
//to read the macro definition
void firstpass(map<string,macro_def *> &m,string input_file){
    ifstream in(input_file);
    string line;
    // int count=0;
    while(getline(in,line)){
        // cout<<count<<endl;
        // count++;
        vector<string> words=extractWords(line);
        if(words.size()>2&&words[1]=="macro"){
            macro_def *new_macro=new macro_def;
            new_macro->macro_name=words[0];
            int n=words.size();
            //getting the macro paramenter
            for(int i=2;i<n;i++){
                new_macro->para_list.push_back(words[i]);
            }
            string def_line;
            while(getline(in,def_line)){
                vector<string> def_words=extractWords(def_line);
                if(def_words[0]=="endm"){
                    break;
                }
                else{
                    new_macro->definition.push_back(def_line);
                }
            }
            m[new_macro->macro_name]=new_macro;
        }
    }
    in.close();
}
```

The firstpass function reads through the input file, identifies and captures information about macro definitions. It checks each line for the "macro" keyword, and if found, creates a macro_def structure. The macro's name is set to the first word, and parameters are extracted from subsequent words. The function then reads lines until it encounters "endm," storing each line in the macro's definition vector. Completed macros are added to a map for later reference. This pass sets up the necessary data structures for the second pass, where actual macro substitution occurs.

Two pass

```
//to replace the macro
void secondpass(map<string,macro_def *> &m,string input_file,string output_file){
    ofstream out(output_file);
    ifstream in(input_file);
    string line;
    while(getline(in,line)){
        vector<string> words=extractWords(line);
        if(words.size()>0&&m.count(words[0])){
            map<string,string> arg_list;
            if(words.size()>1&&words[1]=="macro"){
                out<<line<<endl;
                continue;
            }
            string space;
            int str_len=line.size();
            int j=0;
            while(j<str_len&&(line[j]==' '||line[j]=='\t')){
                space.push_back(line[j++]);
            }
            int n=words.size();
            macro_def* macro=m[words[0]];
            if((n-1)!=macro->para_list.size()){
                cout<<"Parameter list doesn't match"<<endl;
                exit(1);
            }
            int i=1;
            //formal and actual parameter mapping
            for(auto it:macro->para_list){
                arg_list[it]=words[i++];
            }
            //replace the macro with macro definition
            for(auto def_line:macro->definition){
                vector<string> def_words=extractWords(def_line);
                int n=def_words.size();
                for(int i=0;i<n;i++){
                    if(arg_list.count(def_words[i])){
                        def_words[i]=arg_list[def_words[i]];
                    }
                }
                string final_line=join_str(def_words," ");
                out<<space<<final_line<<endl;
            }
        }
        else{
            out<<line<<endl;
        }
    }

    in.close();
    out.close();
}
```

The secondpass function performs the second pass through the input file, replacing macro calls with their corresponding definitions. It reads each line, extracts words, and checks if the first word (macro name) is present in the map of macro definitions (m). If a macro is found, it creates a mapping between formal and actual parameters and replaces macro calls with their definitions, accounting for indentation. The processed lines are then written to the output file. If no macro is detected, the line is copied directly to the output. The function ensures that parameter lists match and handles macro calls with or without the "macro" keyword. The processed output is saved to the specified output file.

Input / Output

Macro Definition

```
print macro msg
    push ax
    push dx
    mov ah, 09h
    lea dx, msg
    int 21h
    pop dx
    pop ax
endm
```

Input File

```
main proc
    mov ax,@data
    mov ds,ax

    start:

    print msg1


    print msg2

    exit:
    mov ah, 4ch
    int 21h

main endp
```

Output File

```
main proc
    mov ax,@data
    mov ds,ax

    start:

    push ax
    push dx
    mov ah, 09h
    lea dx, msg1
    int 21h
    pop dx
    pop ax


    push ax
    push dx
    mov ah, 09h
    lea dx, msg2
    int 21h
    pop dx
    pop ax

    exit:
    mov ah, 4ch
    int 21h

main endp
```

Clearly we can see that the macro calling line got replaced by the body of the macro