Experiment No-6

Aim: Defining a macro with more positional arguments & label argument, expansion of macro & generating expanded source code.

Theory :

1. Macros in Assembly Language:

Macros in assembly language allow for the creation of reusable code snippets, similar to those in high-level languages like C/C++.

In assembly language, macros are typically defined using preprocessor directives or specific macro assembly language instructions.

2. Multiple Positional Arguments in Macros:

Macros can accept multiple positional arguments, enabling flexibility and reusability.

Positional arguments are placeholders within the macro definition that are replaced with actual values when the macro is invoked.

They are represented using symbols like &ARG1, &ARG2, etc., within the macro definition.

3. Label Argument in Macros:

In addition to positional arguments, macros can also include a label argument.

A label argument is a special kind of argument that represents a label or identifier within the macro definition.

It allows for dynamic labeling of instructions or code segments within the macro expansion.

4. Macro Definition Syntax:

The syntax for defining a macro with multiple positional arguments and a label argument typically follows this pattern:

MACRO

&LAB <label_arg> &ARG1, &ARG2, ...

<instruction1 using args>

<instruction2 using args>

MEND

5. Macro Expansion Process:

When a macro call is encountered in the source code, the preprocessor replaces it with the expanded instructions defined in the macro definition.

During expansion, the positional arguments in the macro definition are replaced with the actual arguments provided in the macro call.

The label argument is also replaced with the specified label or identifier.

6. Practical Implementation:

In practice, defining macros with multiple positional arguments and a label argument allows for the creation of reusable code blocks with customizable behavior.

This facilitates code reuse, improves code readability, and reduces redundancy in programming tasks.

7. Considerations and Limitations:

While macros offer flexibility and convenience, they should be used judiciously.

Excessive use of macros can lead to code bloat, reduced readability, and potential maintenance issues.

Care should be taken to ensure that macros are well-documented, properly tested, and used in appropriate contexts.

Input 1: Input Source code with Macro calls

```
MOV R
STAR: RAHUL 30, 40, 50
DCR R
AND R
NEXT: RAHUL 33, 44, 55
MUL 88
HALT
```

Input 2: Macro definition

```
MACRO
&LAB RAHUL
&ARG1, &ARG2, &ARG3
&LAB ADD   &ARG1
SUB   &ARG2
OR    &ARG3
MEND
```

Output source code after Macro expansion:

```
MOV R
STAR: ADD 30
SUB 40
OR  50
DCR R
AND R
NEXT: ADD 33
SUB  44
OR  55
MUL 88
```

HALT

Statistical output: Number of instructions in input source code (excluding Macro calls) = 5

Number of Macro calls = 2

Number of instructions defined in the Macro call = 3

Actual argument during first Macro call "RAHUL" = 30, 40, 50

Actual Label argument during first Macro call = STAR

Actual argument during second Macro call "RAHUL" = 33, 44, 55

Actual Label argument during second Macro call = NEXT

Total number of instructions in the expanded source code = 11

Conclusion:

Macros with multiple positional arguments and a label argument are valuable tools for code abstraction and reuse in assembly language programming.

Understanding their syntax, usage, and expansion process is essential for effective assembly language development and maintenance.

CODE :

```
def expand_macro(input_source, macro_definition):
    source_code_instructions = input_source.split('\n')
    macro_instructions = macro_definition.split('\n')
    macro_name = macro_instructions[1].split()[1]  # Extract macro name
    macro_args = macro_instructions[1].split()[2:]  # Extract macro arguments
    expanded_source_code = []
    macro_calls = 0
    macro_instructions_count = 0
    actual_arguments = []
    label_arguments = []
    for instruction in source_code_instructions:
        if macro_name in instruction:
            macro_calls += 1
            label_arg, *args = instruction.split()[1:]  # Extract label argument and other
arguments
            label_arguments.append(label_arg)
            actual_arguments.append(args)
            # Expand macro
            for i in range(2, len(macro_instructions) - 1):
                macro_instruction = macro_instructions[i]
                for j in range(len(macro_args)):
                    macro_instruction = macro_instruction.replace("&" + macro_args[j],
args[j])
                expanded_source_code.append(macro_instruction.replace("&LAB",
label_arg))
                macro_instructions_count += 1
        else:
            expanded_source_code.append(instruction)
    # Calculate statistics
```

```
    total_instructions = len(source_code_instructions) - macro_calls + macro_instructions_count
    # Output expanded source code
    print("Output source code after Macro expansion:")
    for instruction in expanded_source_code:
        print(instruction)
    # Output statistics
    print("\nStatistical output:")
    print("Number of instructions in input source code (excluding Macro calls) =", len(source_code_instructions) - macro_calls)
    print("Number of Macro calls =", macro_calls)
    print("Number of instructions defined in the Macro call =", macro_instructions_count)
    for i in range(macro_calls):
        print("Actual argument during Macro call \"{}\" = {}".format(macro_name, actual_arguments[i]))
        print("Actual Label argument during Macro call =", label_arguments[i])
    print("Total number of instructions in the expanded source code =", total_instructions)
# Input source code with Macro calls
input_source_code = """MOV R
STAR: RAHUL 30, 40, 50
DCR R
AND R
NEXT: RAHUL 33, 44, 55
MUL 88
HALT"""
# Macro definition
macro_definition = """MACRO
&LAB RAHUL &ARG1, &ARG2, &ARG3
```

&LAB ADD &ARG1

SUB &ARG2

OR &ARG3

MEND"""

expand_macro(input_source_code, macro_definition)

OUTPUT :

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\sikandar\OneDrive\Desktop\6TH SEMESTER\SPCC>python morepositional.py
Output source code after Macro expansion:
MOV R
RAHUL ADD &ARG1
SUB &ARG2
OR &ARG3
DCR R
AND R
RAHUL ADD &ARG1
SUB &ARG2
OR &ARG3
MUL 88
HALT

Statistical output:
Number of instructions in input source code (excluding Macro calls) = 5
Number of Macro calls = 2
Number of instructions defined in the Macro call = 6
Actual argument during Macro call "RAHUL" = ['30,', '40,', '50']
Actual Label argument during Macro call = RAHUL
Actual argument during Macro call "RAHUL" = ['33,', '44,', '55']
Actual Label argument during Macro call = RAHUL
Total number of instructions in the expanded source code = 11

C:\Users\sikandar\OneDrive\Desktop\6TH SEMESTER\SPCC>
```