BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300108
Name	Hatim Sawai
Class and Batch	TE Computer Engineering - Batch C
Date	1-04-2024
Lab #	9
Aim	Write a program to Implement a 2 pass Macro Processor
Objective	Implement Macros in Assembly language, and a 2 pass macro processor to pass the program to improve code efficiency and readability.

Theory	**Macros:**
	Macros are a fundamental concept in programming that allows code to be written in a more abstract and reusable manner. They provide a way to define and use reusable code segments within a program. Macros are defined using a macro definition, and they are invoked or called using a macro invocation.
	
	Macro Definitions:
	A macro definition specifies the name of the macro and the sequence of instructions or expressions it represents. Macro definitions are typically defined using a special syntax or keyword. Here's an example of a macro definition in assembly language:
	ADD MACRO a, b
	MOV a, R0
	ADD b, R0
	MEND
	In this example, ADD is the name of the macro, and a and b are the parameters.
	
	Macro Invocations:
	Macro invocations are instances where the macro is used or called within the code. When a macro is invoked, the macro processor replaces the macro invocation with the corresponding sequence of instructions or expressions defined in the macro definition. Here's an example of a macro invocation:
	ADD A, B
	
	Macro Expansions:
	Macro expansion is the process of replacing macro invocations with their corresponding sequence of instructions or expressions. This process is performed by a macro processor before the code is passed to the actual compiler or interpreter for execution.

Macro Processor:

A macro processor is a program or part of a compiler that performs macro expansion. It takes the input code containing macro invocations, expands these macros, and produces the output code with the expanded macros.

2 Pass Macro Processor:

A 2-pass macro processor is a type of macro processor that performs macro expansion in two passes or phases.

First Pass:

In the first pass, the macro definitions are expanded.
The macro processor scans the entire source code and expands all macro invocations, replacing them with their corresponding sequence of instructions or expressions defined in the macro definition.

Second Pass:

In the second pass, the actual code is processed.
This pass is performed by the actual compiler or interpreter.
The code generated in the first pass, which contains expanded macros, is passed to the compiler or interpreter for further processing and execution.

Advantages of 2 Pass Macro Processor:

1. Provides efficient and flexible macro expansion.
2. Reduces the complexity of code by allowing the use of macros.
3. Improves code readability and maintainability by reducing code duplication.
4. Allows for modular programming by enabling the reuse of code fragments.

Comparison:

Below is a comparison between single pass and 2 pass macro processor:

Aspect	Single Pass Macro Processor	2 Pass Macro Processor
Macro Expansion	Macro expansion is performed in a single pass	Macro expansion is performed in two passes
Efficiency	May require multiple scans of the source code, leading to slower processing times	Performs macro expansion more efficiently, leading to faster processing times
Code Optimization	Limited code optimization opportunities	Allows for more extensive code optimization
Complexity	Relatively simpler implementation	Requires more complex implementation
Memory Usage	May require more memory due to multiple passes	Typically requires less memory due to fewer passes
Maintenance	May lead to more difficult code maintenance	Easier code maintenance due to improved readability and modularity

Implementation / Code

```
from prettytable import PrettyTable
class DefinitionTable:
    def __init__(self):
        self.index = None
        self.definition = None
        self.arg = [None, None]
        self.next = None

class ArgumentListArray:
    def __init__(self):
        self.index = None
        self.arg = None
        self.next = None

class NameTable:
    def __init__(self):
        self.index = None
        self.name = None
        self.dt_index = None
        self.next = None

def find_arg_index(arg, al_head):
    temp = al_head
    while temp is not None:
        if temp.arg == arg:
            return temp
        temp = temp.next
    return None

def find_name(name, nt_head):
    temp = nt_head
    while temp is not None:
        if temp.name == name:
            return temp.dt_index
        temp = temp.next
    return None
```

```
def pass1(fp):
    global MDTC, MNTC
    MDTC = MNTC = 1
    dt_head = None
    nt_head = None
    al_head = None
    al_index = 1

    while True:
        line = fp.readline()
        if not line:
            break

        if "MACRO" in line:
            tokens = line.split()
            print(f"\nMACRO {tokens[0]} Detected...\n")

            if nt_head is None:
                nt_head = NameTable()
                nt_temp = nt_head
            else:
                nt_temp.next = NameTable()
                nt_temp = nt_temp.next

            nt_temp.index = MNTC
            MNTC += 1
            nt_temp.name = tokens[0]
            print(f"\n{tokens[0]} added into Name Table")

            for token in tokens[1:]:
                if token != "MACRO" and token != "\n":
                    if al_head is None:
                        al_head = ArgumentListArray()
                        al_temp = al_head
                    else:
```

```
                        al_temp.next = ArgumentListArray()
                        al_temp = al_temp.next

                    al_temp.index = al_index
                    al_index += 1
                    al_temp.arg = token
                    print(f"\nArgument {al_temp.arg} added into
argument list array")

            if dt_head is None:
                dt_head = DefinitionTable()
                dt_temp = dt_head
            else:
                dt_temp.next = DefinitionTable()
                dt_temp = dt_temp.next

            dt_temp.definition = nt_temp.name
            print(f"\nDefinition table entry created for
{nt_temp.name}")
            nt_temp.dt_index = dt_temp

            while True:
                line = fp.readline()
                if line.strip() == "MEND":
                    break

                tokens = line.split()
                is_arg = 0
                index = 0

                for token in tokens:
                    if is_arg == 0:
                        if dt_head is None:
                            dt_head = DefinitionTable()
                            dt_temp = dt_head
                        else:
```

```
                            dt_temp.next = DefinitionTable()
                            dt_temp = dt_temp.next

                        dt_temp.index = MDTC
                        MDTC += 1
                        dt_temp.definition = token
                        print(f"\nEntry appended for
{dt_temp.definition} at index {dt_temp.index}")
                        is_arg = 1
                    else:
                        if find_arg_index(token, al_head) is None:
                            if al_head is None:
                                al_head = ArgumentListArray()
                                al_temp = al_head
                            else:
                                al_temp.next = ArgumentListArray()
                                al_temp = al_temp.next

                            al_temp.index = al_index
                            al_index += 1
                            al_temp.arg = token
                            dt_temp.arg[index] = al_temp
                        else:
                            dt_temp.arg[index] = find_arg_index(token,
al_head)
                        index += 1

        # print("\nAll three tables are updated. Pass 1 Complete!\n")
    # Assuming nt_head, dt_head, and al_head are initialized in the
main function
    print_name_table(nt_head)
    print_definition_table(dt_head)
    print_argument_list_array(al_head)

def pass2(fp):
    line = fp.readline()
```

```
    while line:
        print(line)
        temp = find_name(line, nt_head)
        if temp is not None:
            while temp.definition != "MEND":
                print("-", temp.definition, temp.arg[0], temp.arg[1])
                temp = temp.next
        line = fp.readline()

    print("\nOutput file updated with expanded code. Pass 2
Complete!\n")


def print_name_table(nt_head):
    table = PrettyTable(["Index", "Name", "Definition Table Index"])
    temp = nt_head
    while temp:
        table.add_row([temp.index, temp.name, temp.dt_index.index])
        temp = temp.next
    print("Name Table:")
    print(table)

def print_definition_table(dt_head):
    table = PrettyTable(["Index", "Definition", "Arguments", "Next"])
    temp = dt_head
    while temp:
        arg_list = [arg.arg for arg in temp.arg if arg]
        table.add_row([temp.index, temp.definition, arg_list,
temp.next])
        temp = temp.next
    print("\nDefinition Table:")
    print(table)

def print_argument_list_array(al_head):
    table = PrettyTable(["Index", "Argument", "Next"])
    temp = al_head
```

```
    while temp:
        table.add_row([temp.index, temp.arg, temp.next])
        temp = temp.next
    print("\nArgument List Array:")
    print(table)

def main():
    global nt_head, al_head
    nt_head = None
    al_head = None

    try:
        with open("input.asm", "r") as fp:
            print("\nPass 1 in progress\n")
            pass1(fp)

        with open("input.asm", "r") as fp:
            print("\nPass 2 in progress\n")
            pass2(fp)


    except IOError:
        print("\nFailed to open the assembly file!")

if __name__ == "__main__":
    main()
```

Output

```
Hitstar53 at …\SPCC Practicals on main (⚠✓) via
→ python -u "d:\SEM_6\SPCC Practicals\Exp9\exp9.py"

Pass 1 in progress


MACRO INCR Detected...


INCR added into Name Table

Argument X added into argument list array

Definition table entry created for INCR

Entry appended for MOV at index 1

Entry appended for ADD at index 2

All three tables are updated. Pass 1 Complete!
```

```
Pass 2 in progress


INCR MACRO X


MOV R 1


ADD X R


MEND
.CODE


MOV AX, 5


INCR AX


CLC


END


Output file updated with expanded code. Pass 2 Complete!
```

Conclusion

A 2 pass macro processor offers several advantages over a single pass macro processor, including improved efficiency, code optimization, and ease of maintenance. By performing

	macro expansion in two passes, it enables more efficient and flexible code processing, leading to faster processing times and improved code quality.
References	[1] Chatgpt, https://chat.openai.com/share/2f45bddf-4a87-49ef-b4c0-c0d2fa2e4fb7