**Course - System Programming and Compiler Construction (SPCC)**

|  |  |
| --- | --- |
| **UID** | 2021300126 |
| **Name** | Pranay Singhvi |
| **Class and Batch** | TE Computer Engineering - Batch C |
| **Date** | 22-04-2024 |
| **Lab #** | 9 |
| **Aim** | Design a two pass Macro Processor |
| **Objective** | Implement macros for assembly language instructions to improve code readability, efficiency, and maintainability in a simulated environment. |
| **Theory** | **MACRO PREPROCESSOR**  **Macro**   * A macro (or macro instruction)   + It is simply a notational convenience for the programmer.   + It allows the programmer to write shorthand version of a program   + It represents a commonly used group of statements in the source program. * For example:   + Suppose a program needs to add two numbers frequently. This requires a sequence of instructions. We can define and use a macro called SUM, to represent this sequence of instructions.   SUM MACRO &X,&Y  LDA &X  MOV B  LDA &Y  ADD B  MEND  **Macro Preprocessor**   * The macro pre-processor(or macro processor) is a system software which replaces each macro instruction with the corresponding group of source language statements. * This operation is called **expanding the macro.** * It does not concern the meaning of the involved statements during macro expansion. * The design of a macro processor generally is machine independent.   A diagram of a computer code  Description automatically generated  **BASIC MACRO PROCESSOR FUNCTIONS**  The fundamental functions common to all macro processors are: ( Code to remember - **DIE** )   * Macro **D**efinition * Macro **I**nvocation * Macro **E**xpansion   **Macro Definition**   * Macro definitions are typically located at the start of a program. * A macro definition is enclosed between a macro header statement(MACRO) and a macro end statement(MEND) * Format of macro definition   macroname MACRO parameters  :  body  :  MEND   * A macro definition consist of macro prototype statement and body of macro. * A macro prototype statement declares the name of a macro and its parameters. It has following format:   *macroname MACRO parameters*  where *macroname* indicates the name of macro, *MACRO* indicates the beginning of macro definition and *parameters* indicates the list of formal parameters. *parameters* is of the form &parameter1, &parameter2,…Each parameter begins with ‘&’. Whenever we use the term macro prototype it simply means the macro name along with its parameters.   * Body of macro consist of statements that will generated as the expansion of macro. * Consider the following macro definition:   SUM MACRO &X,&Y  LDA &X  MOV B  LDA &Y  ADD B  MEND  Here, the macro named SUM is used to find the sum of two variables passed to it.  **Macro Invocation(or Macro Call)**   * A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments to be used in expanding the macro. * The format of macro invocation   macroname p1, p2,...pn   * The above defined macro can be called as SUM P,Q   **Macro Expansion**   * Each macro invocation statement will be expanded into the statements that form the body of the macro. * Arguments from the macro invocation are substituted for the parameters in the macro prototype. * The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc. * Comment lines within the macro body have been deleted, but comments on individual statements have been retained. Macro invocation statement itself has been included as a comment line. * Consider the example for macro expansion on next page:   In this example, the macro named SUM is defined at the start of the program. This macro is invoked with the macro call SUM P,Q and the macro is expanded as  LDA &P  MOV B  LDA &Q  ADD B  MEND  Again the same macro is invoked with the macro call SUM M,N and the macro is expanded as  LDA &M  MOV B  LDA &N  ADD B  MEND  Figure: Example for macro expansion  A diagram of a computer code  Description automatically generated  **Two pass macro processor**   * It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass and all macro invocation statements are expanded during second pass. * Such a two pass macro processor cannot handle **nested macro definitions.** Nested macros are macros in which definition of one macro contains definition of other macros. * Consider the macro definition example given below, which is used to swap two numbers. The macro named SWAP defines another macro named STORE inside it. These type of macro are called nested macros.   A diagram of a function  Description automatically generated with medium confidence  **One pass macro processor**   * A one-pass macro processor uses only one pass for processing macro definitions and macro expansions. * It can handle nested macro definitions. * To implement one pass macro processor, the definition of a macro must appear in the source program before any statements that invoke that macro.   **Data Structures involved in the design of one pass macro processor**   * There are 3 main data structures involved in the design of one pass macro processor:   **DEFTAB**  **NAMTAB**  **ARGTAB**  **Definition table (DEFTAB)**   * All the macro definitions in the program are stored in DEFTAB, which includes macro prototype and macro body statements. * Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion. * References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.   **Name table (NAMTAB)**   * The macro names are entered into NAMTAB * NAMTAB contains pointers to beginning and end of definition in DEFTAB.   **Argument table (ARGTAB)**   * The third data structure is an argument table (ARGTAB), which is used during expansion of macro invocations. * When macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list. * As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body. * Example: Consider the following source code   SUM MACRO &X,&Y  LDA &X  MOV B  LDA &Y  ADD B  MEND  START  LDA 4500  ADD B  SUM P,Q  LDA 3000  ………….  END   * When the macro definition for SUM is encountered, the macro name SUM along with its parameters X and Y are entered into DEFTAB. Then the statements in the body of macro is also entered into DEFTAB. The positional notation is used for the parameters. The parameter &X has been converted to ?1, &Y has been converted to ?2. * The macro name SUM is entered into NAMTAB and the beginning and end pointers are also marked. * On processing the input code, opcode in each statement is compared with the NAMTAB, to check whether it is a macro call. When the macro call SUM P,Q is recognized, the arguments P and Q will entered into ARGTAB. The macro is expanded by taking the statements from DEFTAB using the beginning and end pointers of NAMTAB. * When the ?n notation is recognized in a line from DEFTAB, the corresponding argument is taken from ARGTAB   **Figure shows the different data structures used**A diagram of a number of numbers  Description automatically generated |
| **Input** | ; Define a macro to increment two values  INCR MACRO X, Y  MOVER X, Y  MEND  ; Define a macro to decrement two values  DECR MACRO X, Y  MOVEM Y, X  MEND  ; Define a macro to print three values  PRN MACRO X, Y, Z  MOVER X, Y  MOVEM Y, Z  PRINT X, Y  PRINT Z  MEND  START 100  READ N1  INCR N1, N2  DECR N1, N2  READ N2  INCR N1, N2  INCR N3, N4  DECR N3, N4  PRN A1, A2, Z9  STOP  END |
| **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** | A screenshot of a computer program  Description automatically generated  A screenshot of a computer  Description automatically generated  A screenshot of a computer program  Description automatically generated |
| **Conclusion** | In conclusion, I successfully implemented macros, enhancing code readability, efficiency, and maintainability in the assembly language program. |
| **References** | [1] CS303 System Software  <http://www.icet.ac.in/Uploads/Downloads/Module%205.pdf>  [2] ChatGPT (April 26, 2024) Two Pass Macro Processor  <https://chat.openai.com/share/cf1f7ca3-7c4c-4223-a4e2-3a0c5372af82> |