



UNIT 3

MACROS AND MACRO PROCESSORS

INTRODUCTION

su ape 6e ?= program generation facility

- Macros are used to provide a program generation facility through macro expansion. kevi ritna ape 6e
- Many languages provide build-in facilities for writing macros like PL/I, C, Ada AND C++. kai languages build-in functionality ape 6e.
- Assembly languages also provide such facilities. does assembly languages apee?= yes.
- When a language does not support build-in facilities for writing macros what is to be done?
- A programmer may achieve an equivalent effect by using generalized preprocessors or software tools like Awk of Unix.

agar jo build-in functionality na hoi to su kari sakai 6e?

= aeno j equivalent effect achieve kari sakai 6e by using generalized preprocessors or software tools like Awk of Unix.



A MACRO

- **Def: A macro** is a unit of specification for program generation through expansion.
- A macro consists of
 - a name,
 - a set of formal parameters and
 - a body of code.
- The use of a macro name with a set of actual parameters is replaced by some code generated from its body.
- This is called **macro expansion**.
- Two kinds of expansion can be identified.



ek character string ne bije character string thi replace kari sakai 6e ...
kyare ?= during the program generation

CLASSIFICATION OF MACROS: (replaced by) formal parameters -----> actual parameters

- Lexical expansion:
 - Lexical expansion implies replacement of a character string by another character string during program generation.
 - Lexical expansion is to replace occurrences of formal parameters by corresponding actual parameters.
- Semantic expansion: *generation of instructions that are tailored to the requirements of the usage.*
 - Semantic expansion implies generation of instructions tailored to the requirements of a specific usage.
 - Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and opcodes of instructions.
 - Eg: Generation of type specific instructions for manipulation of byte and word operands.

different uses of macro => lead to codes ke je differ karse in the
1.number 2.sequence 3. opcodes of instructions.



EXAMPLE

1. move => memory word -> m/c register
2. add => increment the value
3. move => m/c register -> m/m word

- The following sequence of instructions is used to increment the value in a memory word by a constant.
 - 1. Move the value from the memory word into a machine-register.
 - 2. Increment the value in the machine register.
 - 3. Move the new value into the memory word.
- Since the instruction sequence MOVE-ADD-MOVE may be used a number of times in a program, it is convenient to define a macro named INCR.
- Using Lexical expansion the macro call INCR A,B,AREG can lead to the generation of a MOVE-ADD-MOVE instruction sequence to increment A by the value of B using AREG to perform the arithmetic.
- Use of Semantic expansion can enable the instruction sequence to be adapted to the types of A and B.
- For example an INC instruction could be generated if A is a byte operand and B has the value "1".

lexical ->
code generation of the instructions

semantic -> data types



HOW DOES MACRO DIFFER FROM SUBROUTINE?

- Macros differ from subroutines in one fundamental respect.
- Use of a macro name in the mnemonic field of an assembly statement leads to its expansion,
- whereas use of subroutine name in a call instruction leads to its execution.
- So there is difference in size and execution efficiency.
 - Size
 - Execution Efficiency macros will increase the program size for the execution efficiency
- Macros can be said to trade program size for execution efficiency.
- More difference would be discussed at the time of discussion of macro expansion.



Here's a comparison of macros and subroutines in a tabular form:

Feature	Macro	Subroutine
Purpose	Define reusable code snippets	Encapsulate and modularize code
Expansion	Expanded inline wherever invoked	Called via "call" instruction
Parameters	Can accept parameters	Can accept parameters and return values
Flexibility	Offers flexibility in code generation	Provides structured parameter passing
Usage	For code expansion at compile time	For modularizing code and improving readability
Overhead	No subroutine call overhead	Incurs subroutine call overhead
Visibility	Limited to the scope of the assembly file	Can be called from any part of the program
Memory Usage	May increase code size	May consume stack space

MACRO DEFINITION AND CALL

- MACRO DEFINITION

- A macro definition is enclosed between a macro header statement and a macro end statement. *kevi ritna tame maro definaciton ne lakso.*

- Macro definitions are typically located at the start of a program. *where are they located*

- A macro definition consists of.

- A macro prototype statement
 - One or more model statements
 - Macro preprocessor statements

pmp =>

1.a macro prototype statement

2.one or model statement

3.macro preprocessor statement

- The macro prototype statement declares the name of a macro and the names and kinds of its parameters.

- It has the following syntax

`<macro name> [< formal parameter spec > [...]]`

- Where `<macro name>` appears in the mnemonic field of an assembly statement and

- `< formal parameter spec>` is of the form

`&<parameter name> [<parameter kind>]`

- See example of slide 10.



MACRO CALL

- A macro is called by writing the macro name in the mnemonic field.
- Macro call has the following syntax.

actually ma to aa pass thase inside the macro

<macro name> [<actual parameter spec>[,...]]

- Where an actual parameter resembles an operand specification in an assembly language statement.

actual parameters no matlab su 6e ke=> tame operands ne specify karso...in the assembly language statements na andar.

EXAMPLE

- MACRO and MEND are the macro header and macro end statements.
- The prototype statement indicates that three parameters called

- MEM_VAL,
- INCR_VAL and
- REG exists

prototype thi khbar padi ke su su avse?
1.m/m val 2.inc_val 3.reg exists

for the macro.

what is the default type of parameters ? = positional parameters

- Since parameter kind is not specified for any of the parameters, they are all of the default kind „positional parameter“.
- Statements with the operation codes MOVER, ADD and MOVEM are model statements.
- No preprocessor statements are used in this macro.

model statements na andar to opcodes avse?=>
MOVEM, MOVER , ADD

```
MACRO
INCR      &MEM_VAL, &INCR_VAL, &REG
MOVER     &REG, &MEM_VAL
ADD       &REG, &INCR_VAL
MOVEM     &REG, &MEM_VAL
MEND
```



MACRO EXPANSION

- Macro call leads to macro expansion.
- During macro expansion, the macro call statement is replaced by a sequence of assembly statements.
- How to differentiate between the original statements of a program and the statements resulting from macro expansion?
MACRO statements ne tame kevi ritna tame differ karso from the original statements?
- Ans: Each expanded statement is marked with a '+' preceding its label field.
- Two key notions concerning macro expansion are
 - A. Expansion time control flow : This determines the order in which model statements are visited during macro expansion. kaya order ma model statements ne tamare visit karvanu 6e during the macro expansion
 - B. Lexical substitution: Lexical substitution is used to generate an assembly statement from a model statement. this is to generate the assembly statements -----> model statements



agar jo preprocessor statements nahi hoie then to tmre sequentially vist karvanu 6e?=>
starting line after the macro prototype statements
ending before the line MEND

A. EXPANSION TIME CONTROL FLOW

- The default flow of control during macro expansion is sequential. default flow su e?
- In the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement.
- What can alter the flow of control during expansion?
- A preprocessor statement can alter the flow of control during expansion such that
 - Conditional Expansion: some model statements are either never visited during expansion, or
 - Expansion Time Loops: are repeatedly visited during expansion.
- The flow of control during macro expansion is implemented using a macro expansion counter (MEC)

kone alter kari sake 6e
tamara macro expansion no flow
1.conditional expansion
2.expansion time loops



ALGORITHM (MACRO EXPANSION)

- 1. $MEC :=$ statement number of first statement following the prototype stmt.
- 2. While statement pointed by MEC is not a MEND statement.
 - a. If a model statement then
 - i. Expand the statement
 - ii. $MEC := MEC + 1$;
 - b. Else (i.e. a preprocessor statement)
 - i. $MEC :=$ new value specified in the statement.
- 3. Exit from macro expansion.



B. LEXICAL SUBSTITUTION

- A model statement consists of 3 types of strings.
 - An **ordinary string**, which stands for itself. => unchanged
 - The name of a **formal parameter** which is preceded by the character „&“.
 - The name of a **preprocessor variable**, which is also preceded by the character „&“.
- During lexical expansion, strings of type 1 are retained without substitution.
- String of types 2 and 3 are replaced by the „values“ of the formal parameters or preprocessor variables.
- Rules for determining the value of a formal parameter depends on the kind of parameter:
 - Positional Parameter
 - Keyword Parameter
 - Default specification of parameters
 - Macros with mixed parameter lists
 - Other uses of parameter

pkdmo



POSITIONAL PARAMETERS

- A positional formal parameter is written as &<parameter name>,
 - e.g. &SAMPLE
 - where SAMPLE is the name of parameter.
 - <parameter kind> of syntax rule is omitted.
- The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:
 - Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
 - Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement.

EXAMPLE

- Consider the call:

INCR A,B,AREG

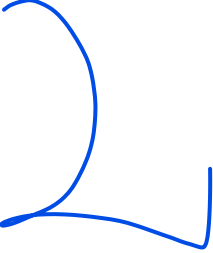
- On macro INCR, following rule of positional association, values of formal parameters are:

○	Formal parameter	value
○	MEM_VAL	A
○	INCR_VAL	B
○	REG	AREG

- Lexical expansion of the model statements now leads to the code

- + MOVER AREG,A
- + ADD AREG,B
- + MOVEM AREG,A

KEYWORD PARAMETER

- 
- For keyword parameter,
 - <parameter name> is an ordinary string and
 - <parameter kind> is the string „≐“ in syntax rule.
 - The <actual parameter spec> is written as <formal parameter name>= <ordinary string>.
 - Note that the ordinal position of the specification XYZ=ABC in the list of actual parameters is immaterial.
 - This is very useful in situations where long lists of parameters have to be used.
 - Let us see example for it.

EXAMPLE:

- Following are macro call statement:

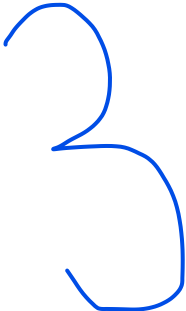
INCR_M MEM_VAL=A, INCR_VAL=B, REG=AREG

INCR_M INCR_VAL=B, REG=AREG, MEM_VAL=A

- Both are equivalent.
- Following is macro definition using keyword parameter:
- MACRO
- INCR_M &MEM_VAL=, &INCR_VAL=,®=
- MOVER ®, &MEM_VAL
- ADD ®, &INCR_VAL
- MOVEM ®,&MEM_VAL
- MEND



DEFAULT SPECIFICATIONS OF PARAMETERS

- 
- A default value is a standard assumption in the absence of an explicit specification by the programmer.
 - Default specification of parameters is useful in situations where a parameter has the same value in most calls.
 - When the desired value is different from the default value, the desired value can be specified explicitly in a macro call.
 - The syntax for formal parameter specification, as follows:
&<parameter name> [<parameter kind> [<default value>]]

EXAMPLE

- The macro can be redefined to use a default specification for the parameter REG
- INCR_D MEM_VAL=A, INCR_VAL=B
- INCR_D INCR_VAL=B, MEM_VAL=A
- INCR_D INCR_VAL=B, MEM_VAL=A, REG=BREG
- First two calls are equivalent but third call overrides the default value for REG with the value BREG in next example. Have a look.
- MACRO
- INCR_D &MEM_VAL=, &INCR_VAL=, ®=AREG
- MOVER ®, &MEM_VAL
- ADD ®, &INCR_VAL
- MOVEM ®, &MEM_VAL
- MEND

