# Macro

# Introduction

- A macro instruction (abbreviated to macro) is simply a notational convenience for the programmer.
- Represents a commonly used group of statements in the source programming language
- Expanding a macro
  - Replace each macro instruction with the corresponding group of source language statements

- We will follow SIC (Simplified Instructional Computer) architecture and then
- NASM assembler (x86 arch)

# Introduction

- SIC:
  - 24 bit registers
  - A, X, L - (0,1,2)
  - PC
  - SW (Status Word) - (8,9)

  - GPR:
    - B, S, T, F - (3,4,5,6)

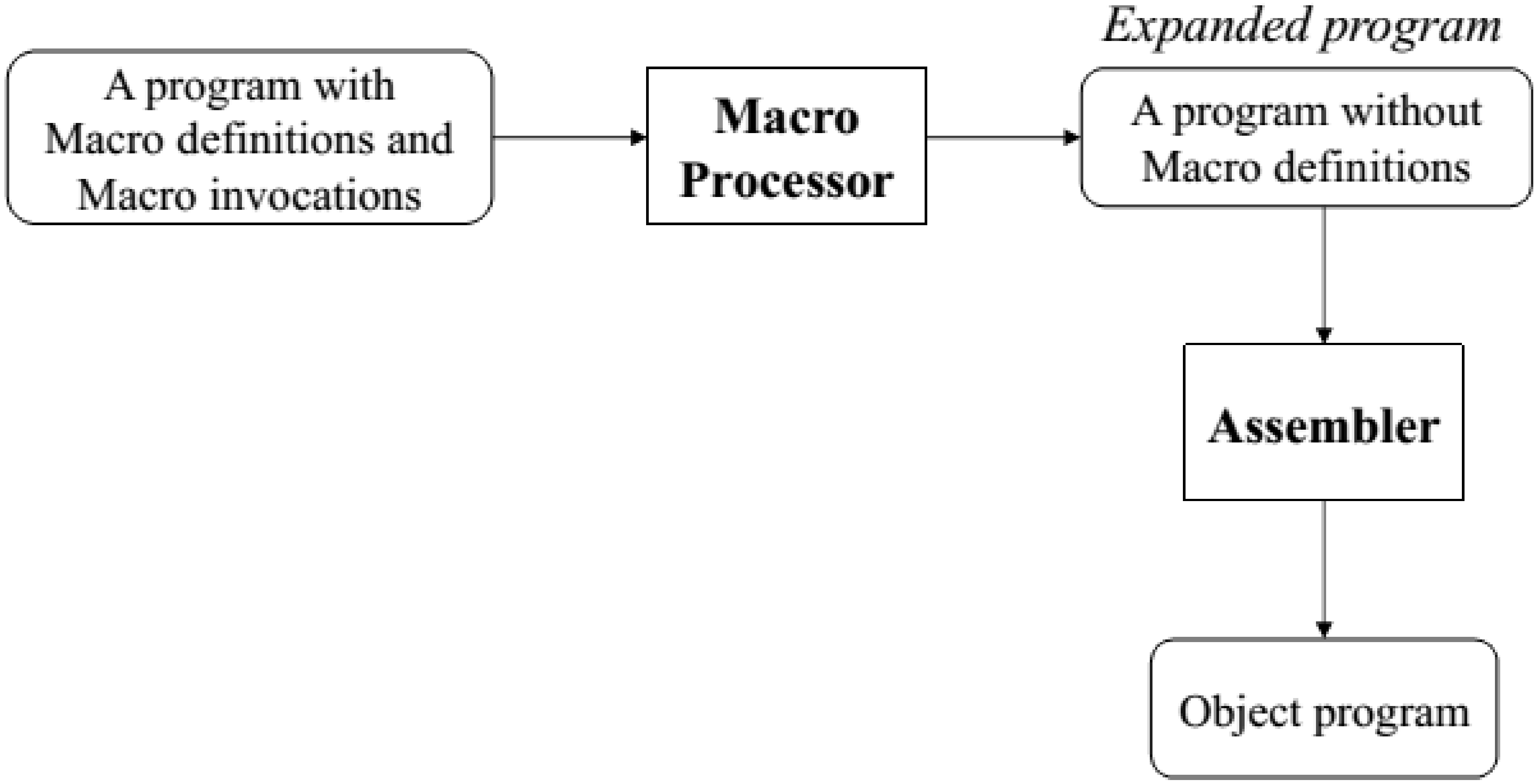
# Introduction

- For example:
- On SIC/XE (Extra Equipment), requires a sequence of seven instructions to save the contents of all registers

- A macro processor is not directly related to the architecture of the computer on which it is to run

- Macro processors can also be used with high-level programming languages, OS command languages, etc.

# Basic Macro Processor Functions



# Basic Macro Processor Functions

Macro Definition:

1. Two new assembler directives
    - **MACRO**
    - **MEND**
2. A pattern or **prototype** for the macro instruction
3. Macro name and parameters

# Basic Macro Processor Functions

- Macro invocation
  - Often referred to as a ***macro call***
  - Need:

  a) name of the macro instruction being invoked

  b) arguments to be used in expanding the macro
- Expanded program
  - No macro instruction definitions
  - Each macro invocation statement has been expanded with

  a) Macro body

  b) Arguments substituted with the parameters in the prototype

# Macro Definition

1. Copy code
2. Parameter substitution
3. Conditional macro expansion
4. Macro instruction defining macros

# 1. Copy Code

*Source*

```
STRG    MACRO
        STA     DATA1
        STB     DATA2
        STX     DATA3
        MEND
  .
STRG
  .
STRG
  .
  .
```

*Expanded source*

```
  .
  .
  .
  {  STA     DATA1
     STB     DATA2
     STX     DATA3
  .
  {  STA     DATA1
     STB     DATA2
     STX     DATA3
  .
```

# 2. Parameter Substitution

*Source*

```
STRG    MACRO   &a1, &a2, &a3
        STA     &a1
        STB     &a2
        STX     &a3
        MEND
  .
STRG    DATA1, DATA2, DATA3
  .
STRG    DATA4, DATA5, DATA6
  .
  .
```

*Expanded source*

```
  .
  .
  .
    { STA   DATA1
      STB   DATA2
      STX   DATA3
  .
    { STA   DATA4
      STB   DATA5
      STX   DATA6
  .
```

# 2. Parameter Substitution

- Dummy arguments
  - Positional argument

```
STRG    DATA1, DATA2, DATA3
GENER   ,,DIRECT,,,,,,3
```

  - Keyword argument

```
STRG    &a3=DATA1, &a2=DATA2, &a1=DATA3
GENER   TYPE=DIRECT, CHANNEL=3
```

# Example

**Line** **Source statement**

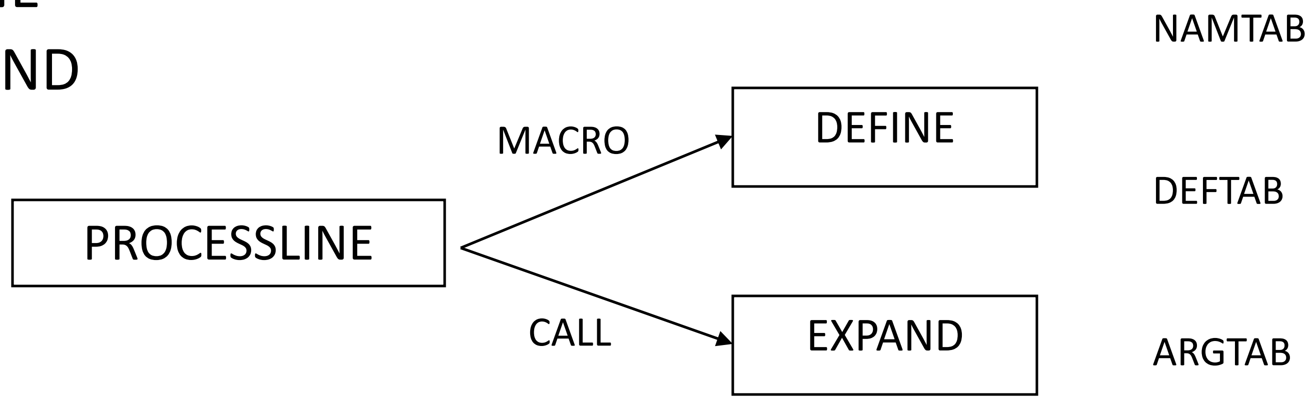
```
COPY      START   0                 COPY FILE FROM INPUT TO OUTPUT
RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
.
.         MACRO TO READ RECORD INTO BUFFER
.
          CLEAR   X                 CLEAR LOOP COUNTER
          CLEAR   A
          CLEAR   S
         +LDT     #4096             SET MAXIMUM RECORD LENGTH
          TD      =X'&INDEV'        TEST INPUT DEVICE
          JEQ     *-3               LOOP UNTIL READY
          RD      =X'&INDEV'        READ CHARACTER INTO REG A
          COMPR   A,S               TEST FOR END OF RECORD
          JEQ     *+11              EXIT LOOP IF EOR
          STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
          TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
          JLT     *-19                HAS BEEN REACHED
          STX     &RECLTH           SAVE RECORD LENGTH
          MEND
```

# Example

```
        WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
        .
        .        MACRO TO WRITE RECORD FROM BUFFER
        .
                 CLEAR    X                  CLEAR LOOP COUNTER
                 LDT      &RECLTH
                 LDCH     &BUFADR,X          GET CHARACTER FROM BUFFER
                 TD      =X'&OUTDEV'         TEST OUTPUT DEVICE
                 JEQ      *-3                LOOP UNTIL READY
                 WD      =X'&OUTDEV'         WRITE CHARACTER
                 TIXR     T                  LOOP UNTIL ALL CHARACTERS
                 JLT      *-14                 HAVE BEEN WRITTEN
                 MEND
```

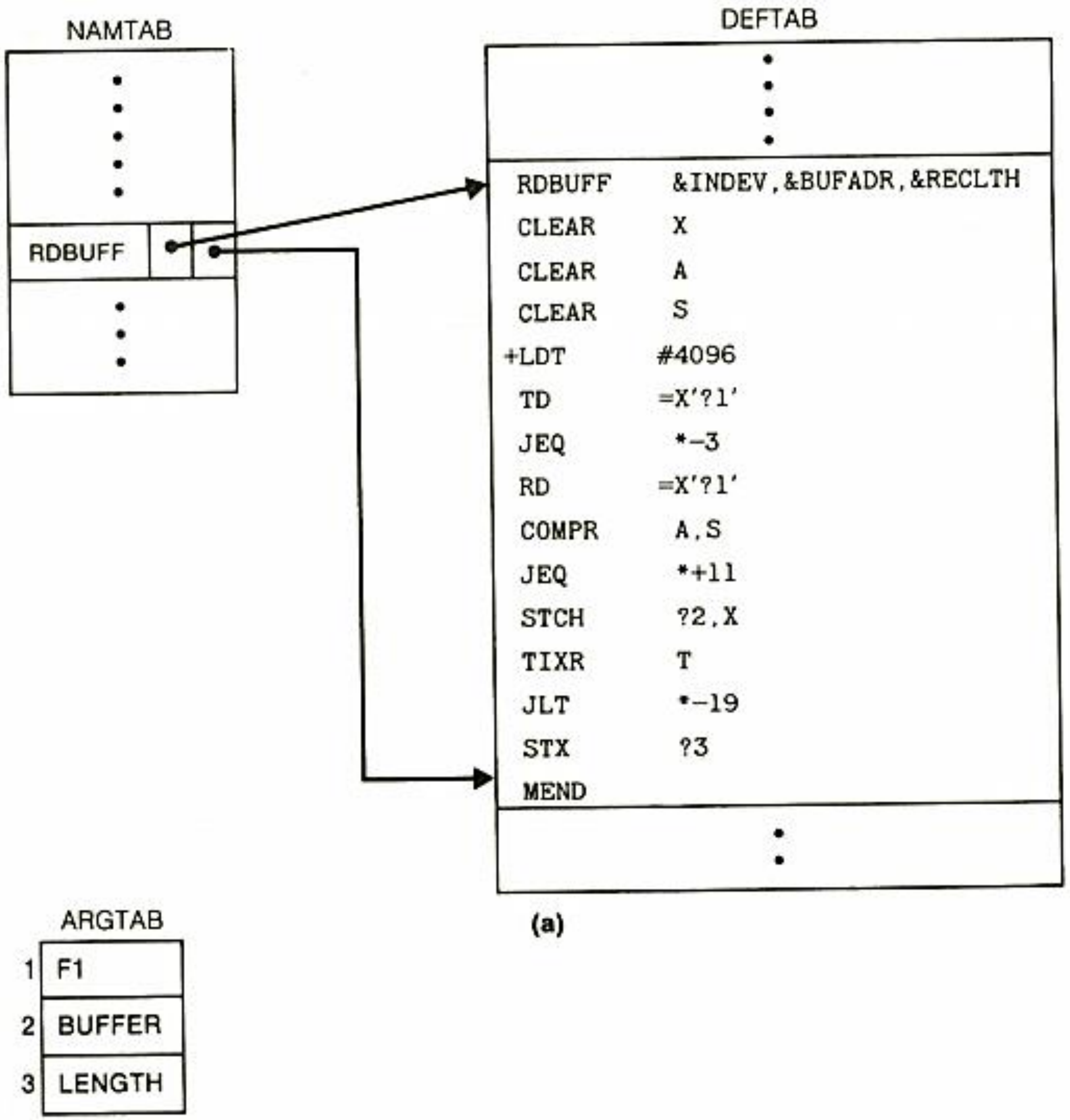
# One Pass Macro Processor

- Prerequisite
  - every macro must be defined before it is called
- Sub-procedures
  - macro definition: DEFINE
  - macro invocation: EXPAND



# Data Structures

- Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro
- Three main data structures involved in an one-pass macro processor
  - DEFTAB,
  - NAMTAB,
  - ARGTAB


NAMTAB
RDBUFF
DEFTAB
RDBUFF &INDEV,&BUFADR,&RECLTH
CLEAR X
CLEAR A
CLEAR S
+LDT #4096
TD =X'?1'
JEQ *-3
RD =X'?1'
COMPR A,S
JEQ *+11
STCH ?2,X
TIXR T
JLT *-19
STX ?3
MEND
ARGTAB
1 F1
2 BUFFER
3 LENGTH

(a)

```
begin {macro processor}
    EXPANDING := FALSE
    while OPCODE  ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}


procedure PROCESSLINE
    begin
        search NAMTAB for OPCODE
        if found then
            EXPAND
        else if OPCODE = 'MACRO' then
            DEFINE
        else write source line to expanded file
    end {PROCESSLINE}
```

**Figure 4.5** Algorithm for a one-pass macro processor.

# 4. Nested Macro

```
1   MACROS    MACRO     {Defines SIC standard version macros}
2   RDBUFF    MACRO     &INDEV,&BUFADR,&RECLTH
                .
                .       {SIC  standard version}
                .
3             MEND      {End of RDBUFF}
4   WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
                .
                .       {SIC standard version}
                .
5             MEND      {End of WRBUFF}
                .
                .
                .
6             MEND      {End of MACROS}
```

(a)

# Nested Macro

```
1  MACROX   MACRO     {Defines  SIC/XE macros}
2  RDBUFF   MACRO     &INDEV,&BUFADR,&RECLTH
            .
            .         {SIC/XE version}
            .
3           MEND      {End of RDBUFF}
4  WRBUFF   MACRO     &OUTDEV,&BUFADR,&RECLTH
            .
            .         {SIC/XE version}
            .
5           MEND      {End of WRBUFF}
            .
            .
            .
6           MEND      {End of MACROX}
```

# One Pass Macro Processor:
# *for nested macro*

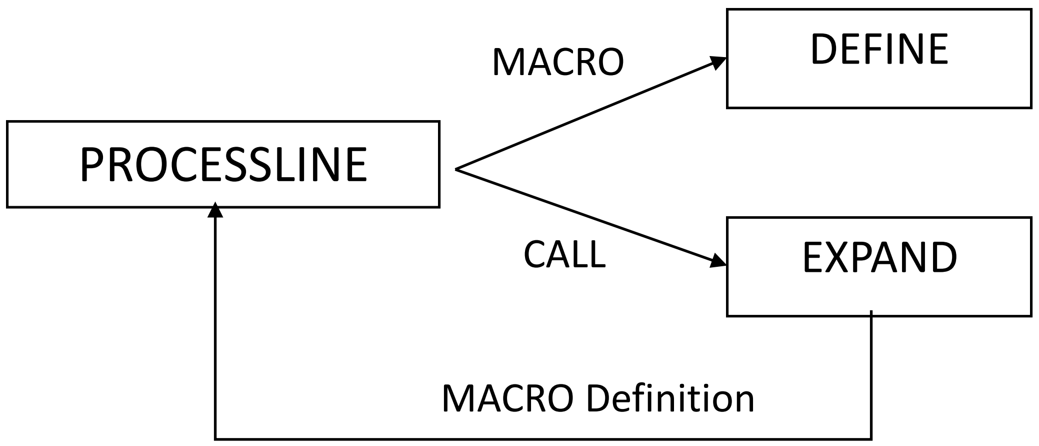
- Sub-procedures
  - macro definition: DEFINE
  - macro invocation: EXPAND
- EXPAND may invoke DEFINE when encounter macro definition

NAMTAB

DEFTAB

ARGTAB

Expanding



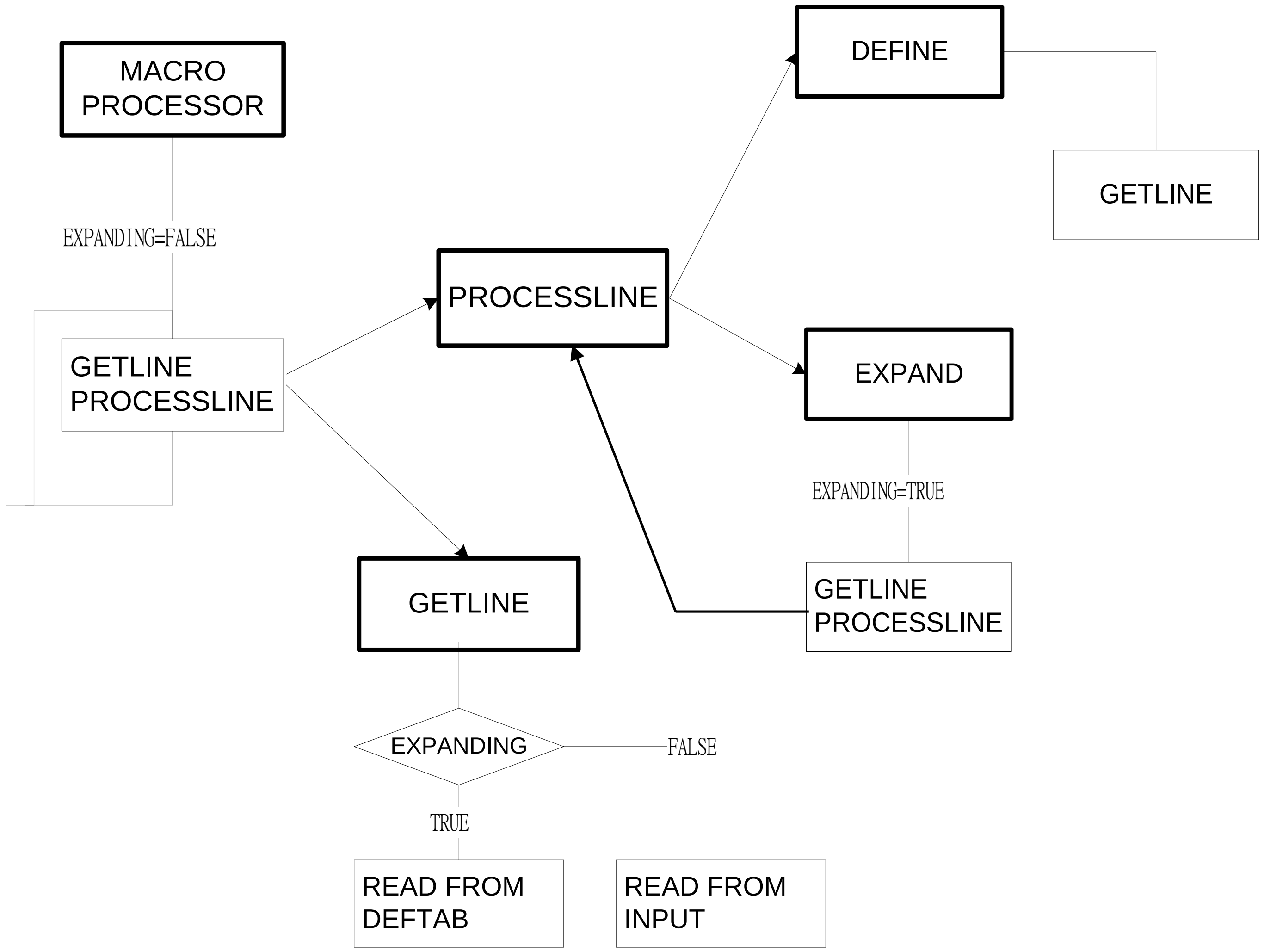
```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL  :- 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE - 'MEND' then
                            LEVEL  := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}
```

```
procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}
```

```
procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

**Figure 4.5** *(cont'd)*

# One Pass Macro Processor



# Algo.

```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL  := 1
        while LEVEL > 0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL  := LEVEL - 1
                    end {if not comment}
            end {while}
        store in NAMTAB pointers to beginning and end of definition
    end {DEFINE}
```

# Algo.

```
procedure EXPAND
    begin
        EXPANDING  := TRUE
        get first line of macro definition {prototype} from DEFTAB
        set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded file as a comment
        while not end of macro definition do
            begin
                GETLINE
                PROCESSLINE
            end {while}
        EXPANDING := FALSE
    end {EXPAND}
```

```
procedure GETLINE
    begin
        if EXPANDING then
            begin
                get next line of macro definition from DEFTAB
                substitute arguments from ARGTAB for positional notation
            end {if}
        else
            read next line from input file
    end {GETLINE}
```

# Macro Expansion Types

- **Lexical Substitution:**
  - Replacement of a character string by another character string during program generation
  - Replacement of Formal Parameters with Actual Parameters
  - Formal Parameter => Macro name and/or parameter list
    - E.g. for the macro **STRG** MACRO **&a1, &a2, &a3**
    - The call **STRG** DATA1, DATA2, DATA3
  - Actual Parameter => Macro body that replaces the formal parameters
    - E.g. After replacement, the macro body
    - **STA DATA1**
    - **STB DATA2**
    - **STX DATA3**

# Macro Expansion Types

- **Semantic Expansion:**
  - Generation of instructions tailored to the requirements of a specific usage.
- ***Characteristics***:
  - 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.

# Example

- The following sequence of instructions is used to increment the value in a memory word by a constant.

1. Move the value (A) 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.

# Example

- Using **Lexical expansion** the macro call **INCR A, B, AREG** can lead to the generation of a **MOVE-ADD-MOVE** instruction sequence
- Increments **A** by the value of **B** using **AREG** (register) 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 (in Intel **8088**) could be generated if **A** is a **byte** operand and **B** has the value **"1".**

# Macro vs. Subroutine

- 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**
  - **Execution Efficiency**
- Macros can be said to trade program size for execution efficiency.

# Machine-Independent Macro Processor Feature

1. Concatenation of Macro Parameters

2. Generation of Unique Labels

3. Conditional Macro Expansion

4. Keyword Macro Parameters

# 1. Concatenation of Macro Parameters

- Most macro processors allow parameters to be concatenated with other character strings
- The need of a special catenation operator
    - LDA **X&ID1**
    - LDA **X&ID**
- The catenation operator
    - LDA **X&ID -> 1**

# 1. Example

```
1  SUM    MACRO  &ID
2         LDA    X&ID→1
3         ADD    X&ID→2
4         ADD    X&ID→3
5         STA    X&ID→S
6         MEND
```

```
SUM    A
   ↓
LDA    XA1
ADD    XA2
ADD    XA3
STA    XAS
```

```
SUM    BETA
   ↓
LDA    XBETA1
ADD    XBETA2
ADD    XBETA3
STA    XBETAS
```

# 2. Generation of Unique Labels

- It is, in general, **not possible** for the body of a macro instruction to contain labels of the usual kind

- Leads to the use of relative addressing at the source statement level
  - Only be acceptable for short jumps

- Solution:
  – Allowing the creation of special types of labels within macro instructions

# 2. Generation of Unique Labels

Solution:

- Allowing the creation of special types of labels within macro instructions

- Labels used within he macro body begin with the special character **$**

- Programmers are instructed **no to use $** in their source programs

# 2. Example

```
RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT     #4096           SET MAXIMUM RECORD LENGTH
$LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
         JEQ      $LOOP          LOOP UNTIL READY
         RD      =X'&INDEV'      READ CHARACTER INTO REG A
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $EXIT          EXIT LOOP IF EOR
         STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP            HAS BEEN REACHED
$EXIT    STX      &RECLTH        SAVE RECORD LENGTH
         MEND
```

# 2. Example

```
        .           RDBUFF    F1,BUFFER,LENGTH

                    CLEAR     X                 CLEAR LOOP COUNTER
                    CLEAR     A
                    CLEAR     S
                   +LDT      #4096              SET MAXIMUM RECORD LENGTH
        $AALOOP     TD       =X'F1'             TEST INPUT DEVICE
                    JEQ       $AALOOP           LOOP UNTIL READY
                    RD       =X'F1'             READ CHARACTER INTO REG A
                    COMPR     A,S               TEST FOR END OF RECORD
                    JEQ       $AAEXIT           EXIT LOOP IF EOR
                    STCH      BUFFER,X          STORE CHARACTER IN BUFFER
                    TIXR      T                 LOOP UNLESS MAXIMUM LENGTH
                    JLT       $AALOOP             HAS BEEN REACHED
        $AAEXIT     STX       LENGTH            SAVE RECORD LENGTH
```

# 3. Conditional Macro Expansion

Most macro processors can:

- Modify the sequence of statements generated for a macro expansion
- Depending on the arguments supplied in the macro invocation

- Macro processor directive
  - IF, ELSE, ENDIF
  - SET
  – Macro-time variable (set symbol)

  - WHILE-ENDW

# 3. Example for **IF-ELSE-ENDIF**

```
RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
         IF      (&EOR NE '')
&EORCK   SET     1
         ENDIF
         CLEAR   X                 CLEAR LOOP COUNTER
         CLEAR   A
         IF      (&EORCK EQ 1)
         LDCH    =X'&EOR'          SET EOR CHARACTER
         RMO     A,S
         ENDIF
         IF      (&MAXLTH EQ '')
        +LDT     #4096             SET MAX LENGTH = 4096
         ELSE
        +LDT     #&MAXLTH          SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD      =X'&INDEV'        TEST INPUT DEVICE
         JEQ     $LOOP             LOOP UNTIL READY
         RD      =X'&INDEV'        READ CHARACTER INTO REG A
         IF      (&EORCK EQ 1)
         COMPR   A,S               TEST FOR END OF RECORD
         JEQ     $EXIT             EXIT LOOP IF EOR
         ENDIF
         STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
         TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP               HAS BEEN REACHED
$EXIT    STX     &RECLTH           SAVE RECORD LENGTH
         MEND
```

```
.                 RDBUFF    F3,BUF,RECL,04,2048

                  CLEAR     X             CLEAR LOOP COUNTER
                  CLEAR     A
                  LDCH      =X'04'        SET EOR CHARACTER
                  RMO       A,S
                 +LDT       #2048         SET MAXIMUM RECORD LENGTH
        $AALOOP   TD        =X'F3'        TEST INPUT DEVICE
                  JEQ       $AALOOP       LOOP UNTIL READY
                  RD        =X'F3'        READ CHARACTER INTO REG A
                  COMPR     A,S           TEST FOR END OF RECORD
                  JEQ       $AAEXIT       EXIT LOOP IF EOR
                  STCH      BUF,X         STORE CHARACTER IN BUFFER
                  TIXR      T             LOOP UNLESS MAXIMUM LENGTH
                  JLT       $AALOOP         HAS BEEN REACHED
        $AAEXIT   STX       RECL          SAVE RECORD LENGTH
```

```
.           RDBUFF    0E,BUFFER,LENGTH,,80

                      CLEAR     X                CLEAR LOOP COUNTER
                      CLEAR     A
                     +LDT       #80              SET MAXIMUM RECORD LENGTH
            $ABLOOP   TD        =X'0E'           TEST INPUT DEVICE
                      JEQ       $ABLOOP          LOOP UNTIL READY
                      RD        =X'0E'           READ CHARACTER INTO REG A
                      STCH      BUFFER,X         STORE CHARACTER IN BUFFER
                      TIXR      T                LOOP UNLESS MAXIMUM LENGTH
                      JLT       $ABLOOP            HAS BEEN REACHED
            $ABEXIT   STX       LENGTH           SAVE RECORD LENGTH
```

```
.            RDBUFF   F1,BUFF,RLENG,04

                      CLEAR    X                  CLEAR LOOP COUNTER
                      CLEAR    A
                      LDCH     =X'04'             SET EOR CHARACTER
                      RMO      A,S
                     +LDT      #4096              SET MAX LENGTH = 4096
             $ACLOOP  TD       =X'F1'             TEST INPUT DEVICE
                      JEQ      $ACLOOP            LOOP UNTIL READY
                      RD       =X'F1'             READ CHARACTER INTO REG A
                      COMPR    A,S                TEST FOR END OF RECORD
                      JEQ      $ACEXIT            EXIT LOOP IF EOR
                      STCH     BUFF,X             STORE CHARACTER IN BUFFER
                      TIXR     T                  LOOP UNLESS MAXIMUM LENGTH
                      JLT      $ACLOOP              HAS BEEN REACHED
             $ACEXIT  STX      RLENG              SAVE RECORD LENGTH
```

# 3. Example for **WHILE-ENDW**

```
RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH,&EOR
&EORCT   SET      %NITEMS(&EOR)
         CLEAR    X                   CLEAR LOOP COUNTER
         CLEAR    A
        +LDT      #4096               SET MAX LENGTH = 4096
$LOOP    TD       =X'&INDEV'          TEST INPUT DEVICE
         JEQ      $LOOP               LOOP UNTIL READY
         RD       =X'&INDEV'          READ CHARACTER INTO REG A
&CTR     SET      1
         WHILE    (&CTR LE &EORCT)
         COMP     =X'0000&EOR[&CTR]'
         JEQ      $EXIT
&CTR     SET      &CTR+1
         ENDW
         STCH     &BUFADR,X           STORE CHARACTER IN BUFFER
         TIXR     T                   LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP                 HAS BEEN REACHED
$EXIT    STX      &RECLTH             SAVE RECORD LENGTH
         MEND
```

```
.              RDBUFF    F2,BUFFER,LENGTH,(00,03,04)


               CLEAR     X                     CLEAR LOOP COUNTER
               CLEAR     A
              +LDT       #4096                 SET MAX LENGTH = 4096
     $AALOOP   TD        =X'F2'                TEST INPUT DEVICE
               JEQ        $AALOOP              LOOP UNTIL READY
               RD        =X'F2'                READ CHARACTER INTO REG A
               COMP      =X'000000'
               JEQ        $AAEXIT
               COMP      =X'000003'
               JEQ        $AAEXIT
               COMP      =X'000004'
               JEQ        $AAEXIT
               STCH       BUFFER,X             STORE CHARACTER IN BUFFER
               TIXR       T                    LOOP UNLESS MAXIMUM LENGTH
               JLT        $AALOOP                HAS BEEN REACHED
     $AAEXIT   STX        LENGTH               SAVE RECORD LENGTH
```

# 4. Keyword Macro Parameters

- Positional Parameters
- Keyword Parameters
  - As discussed earlier

# Previous Design Approach

# Type of Macro processors

- Macro Preprocessors
  - Independent Macro processor.
  - To be invoked separately before the Assembler

- Macro Assemblers
  - Part of the Assembler
  - Assembler runs this module by default before assembling

# Pass Structure

- Two-pass macro processors
  - Goes through the program twice
  - Once to find the definitions
  - Second pass to expand the Macros

- One-pass macro processors
  - Single scan on the source program
  - Defines and Expands macro in single go
  - We have discussed this type

# Two-pass Macro Processors

- Assumptions:
  - Functionally independent of the Assembler
  - Nested Macros are not permitted
  - Macro call occurs within a macro definition

# Pass I – Create Database

1. Check each input line for Keyword MACRO
2. If found,
   a) Copy the entire code block, including corresponding MEND
   b) Insert this code block into next available location in DEFTAB
   c) Insert Macro name into next available location in NAMTAB
   d) Insert Start and End address from DEFTAB into corresponding NAMTAB entry
   e) Remove the entire the entire code block from the source program
3. Stop if no more lines in the source program

# Pass II – Process Macro Call

1. Check each WORD in the source program and match with NAMTAB entries
2. If found
   a) From DEFTAB, find the number of parameters required
   b) Search for the Arguments (if any) from the source code
   c) Put the Arguments in the corresponding location of the ARGTAB
   d) Remove the Formal Parameters from Source program
   e) Take next line of Macro body from DEFTAB
   f) Replace Formal Parameters with Actual Parameters using ARGTAB
   g) Put line in source program's current location
   h) Go back to **Step (e)** till MEND is encountered
3. Stop if no more lines in the source program

# Advantages

- Simple, well defined algorithm
- Modest space requirement
  - Pass 1 requires memory for NAMTAB and DEFTAB
- Forward references are permitted
  - Macro call can precede the associated Macro definition
- A Macro can be defined anywhere in the program

# Disadvantages

- Source program is read twice
  - Execution overhead
- A macro can not be redefined
- Macro conditional statements are not permitted
- Intermediate file need to be stored between Pass1 and Pass2

# One-pass Macro Properties

- Advantages
  - Single-pass: Less overhead
  - Nested Macro permitted
  - Conditional statements permitted
  - Macro can be redefined
  - No intermediate file required
- Disadvantages
  - A Macro definition must precede any call to that macro
  - Primary Memory requirement is more

# General-Purpose Macro Processors: Issues

- In spite of the advantages noted, there are still relatively few general-purpose macro processors.

- In a typical programming language, there are several situations in which normal macro parameter substitution should not occur
  - E.g. comments should usually be ignored by a macro processor

# General-Purpose Macro Processors: Issues

- Another difference between programming languages is related to their facilities for grouping together terms, expressions, or statements
  - E.g. Some languages use keywords such as begin and end for grouping statements. Others use special characters such as { and }

# General-Purpose Macro Processors: Issues

- A more general problem involves the tokens of the programming language
- – E.g. identifiers, constants, operators, and keywords
- – E.g. blanks

# General-Purpose Macro Processors: Issues

- Another potential problem with general purpose macro processors involves the **syntax** used for **macro definitions** and **macro invocation** statements.
- With most special purpose macro processors, macro invocations are very similar in form to statements in the source programming language

# Hygienic macro

- If a Macro expansion is guaranteed not to cause the accidental capture of Identifiers

- This problem is well known in LISP

- A Macro Specific identifier may overshadow a source program’s identifier

# Example

```
#define INCI(i) do { int a=0; ++i; }  while(0)

int main(void)
{
        int a = 4, b = 8;
        INCI(a);
        INCI(b);
        printf("a is now %d, b is now %d\n", a, b);
        return 0;
}
```

```
int main(void)
{
        int a = 4, b = 8;
        do { int a=0; ++a; }  while(0);
        do { int a=0; ++b; }  while(0);
        printf("a is now %d, b is now %d\n", a, b);
        return 0;
}
```

# NASM preprocessor

- Preprocessor directives all begin with a **%** sign.

- The preprocessor collapses all lines which end with a backslash **(\)** character into a single line. Thus:

```
%define THIS_VERY_LONG_MACRO_IS_DEFINED_TO \
THIS_VALUE
```

- will work like a single-line macro without the backslash-newline sequence.

# Single-line macros

- **%define** – defines single-line macro (c-style).
  - `%define ctrl 0x1F &`
    `%define param (a, b) ((a)+(a)*(b))`

```
    mov byte [param(2,ebx)], ctrl 'D'
```

expands to

```
    mov byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

- When the expansion of a single-line macro contains tokens which **invoke another macro**, the expansion is performed at **invocation time**, not at definition time.

```
%define a(x) 1+b(x)
%define b(x) 2*x
mov ax,a(8)
```

- will evaluate in the expected way to

```
mov ax,1+2*8
```

- Macros defined with **%define** are **case sensitive**.
- You can use **%idefine** to define **all the case variants** of a macro at once.

- There is a mechanism which detects when a macro call has occurred as a result of a previous expansion of the same macro, to guard against circular references and infinite loops.

- You can overload single-line macros:

```
%define foo(x) 1+x
%define foo(x,y) 1+x*y
```

- The preprocessor will be able to handle both types of macro call, by counting the parameters you pass.

- **%undef**– undefines defined single-line macro

```
%define foo go
%undef  foo
mov ax, foo
```

- will expand to the instruction **`mov eax, foo`**
  - since after **%undef** the macro foo is no longer defined.

- **`%assign`**
- – used to define single-line macros which **take no parameters** and have a **numeric value**.
- The value can be specified in the form of an expression, and it will be evaluated once, when the %assign directive is processed.
- Like %define, macros defined using %assign can be **re-defined** later, so you can do things like:

```
%assign i i+1
```

# multiple-line macros

- Works with %macro … %endmacro mechanism.

```
%macro prologue 1
    push ebp
    mov ebp,esp
    sub esp,%1
%endmacro
```

this macro gets only one parameter

means: the first parameter of the macro

```
my_func:  prologue 12
```

```
my_func:
    push ebp
    mov ebp,esp
    sub esp,12
```

- With a macro taking more than one parameter, subsequent parameters would be referred to as %2, %3 and so on .

- Multi-line macros, like single-line macros, are case-sensitive, unless you define them using the alternative directive %imacro.
- If you need to pass a comma as part of a parameter to a multi-line macro, you can do that by enclosing the entire parameter in braces.

```
%macro silly 2
      %2: db %1
%endmacro

silly 'a', letter_a
silly 'ab', string_ab
silly {13,10}, crlf
```

- As with single-line macros, multi-line macros can be overloaded by defining the same macro name several times with different numbers of parameters.

# Conditional Assembly

- Similarly to the C preprocessor, NASM allows sections of a source file to be assembled only if certain conditions are met.
- General syntax:

**%if<condition>**
; some code which only appears if <condition> is met
**%elif<condition2>**
; only appears if <condition> is not met but <condition2> is
**%else**
; this appears if neither <condition> nor <condition2> was met %endif

- The **%else** clause is **optional**, as is the **%elif** clause.
- You can have **more than one %elif** clause as well.

# Preprocessor Loops

- **%rep** and %**endrep** enclose a chunk of code, which is then replicated as many times as specified by the preprocessor
- Example:

```
%assign i 0
%rep 64
    inc word [table+2*i]
    %assign i i+1
%endrep
```

- To break out of a repeat loop part way along, you can use the **`%exitrep`** directive to terminate the loop

Example:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
    %if  j > 65535
        %exitrep
    %endif
    dw j
    %assign k j+i
    %assign i j
    %assign j k
%endrep
```