Chapter 4 – Macro Processors

- A *macro* represents a commonly used group of statements in the source programming language. The macro processor replaces each macro instruction with the corresponding group of source language statements. This is called *expanding* the macros.
- Macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macro processor.
- For example, suppose that it is necessary to save the contents of all registers before calling a subprogram.

  On SIC/XE, this would require a sequence of seven instructions (STA, STB, etc.).

  Using a macro instruction, the programmer could simply write one statement like SAVEREGS. This macro instruction would be expanded into the seven assembler language instructions needed to save the register contents.
- The most common use of macro processors is in *assembler language programming*. However, macro processors can also be used with *high-level programming languages*, *operating system command languages*, etc.

4.1 Basic Macro Processor Functions

4.1.1 Macro Definition and Expansion

- Fig 4.1 shows an example of a SIC/XE program using *macro instructions*. The definitions of these macro instructions (RDBUFF and WRBUFF) appear in the source program following the START 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
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
.
.         MAIN PROGRAM
.
FIRST     STL     RETADR                 SAVE RETURN ADDRESS
CLOOP     RDBUFF  F1,BUFFER,LENGTH       READ RECORD INTO BUFFER
          LDA     LENGTH                 TEST FOR END OF FILE
          COMP    #0
          JEQ     ENDFIL                 EXIT IF EOF FOUND
          WRBUFF  05,BUFFER,LENGTH       WRITE OUTPUT RECORD
          J       CLOOP                  LOOP
ENDFIL    WRBUFF  05,EOF,THREE           INSERT EOF MARKER
          J       @RETADR
EOF       BYTE    C'EOF'
THREE     WORD    3
RETADR    RESW    1
LENGTH    RESW    1                      LENGTH OF RECORD
BUFFER    RESB    4096                   4096-BYTE BUFFER AREA
          END     FIRST
```

Figure 4.1 Use of macros in a SIC/XE program.

- Two new assembler directives (MACRO and MEND) are used in macro definitions.

  The first MACRO statement (line 10) identifies the beginning of a macro definition.

  The symbol in the label field (RDBUFF) is the name of the macro, and the entries in the operand field identify the *parameters* of the macro instruction.

- In our macro language, each *parameter* begins with the character &, which facilitates the substitution of parameters during *macro expansion*.

  The *macro name* and *parameters* define a *pattern* or *prototype* for the macro instructions used by the programmer.

  Following the MACRO directive are the statements that make up the body of the macro definition.

  The MEND assembler directive marks the end of the macro definition.

- Fig 4.2 shows the output that would be generated. Each macro invocation statement has been expanded into the statements that form the *body* of the macro, with the *arguments* from the macro invocation substituted for the *parameters* in the *macro prototype*.

```
5      COPY     START    0                  COPY FILE FROM INPUT TO OUTPUT
180    FIRST    STL      RETADR             SAVE RETURN ADDRESS
190    .CLOOP   RDBUFF   F1,BUFFER,LENGTH   READ RECORD INTO BUFFER
190a   CLOOP    CLEAR    X                  CLEAR LOOP COUNTER
190b            CLEAR    A
190c            CLEAR    S
190d           +LDT      #4096              SET MAXIMUM RECORD LENGTH
190e            TD       =X'F1'             TEST INPUT DEVICE
190f            JEQ      *-3                LOOP UNTIL READY
190g            RD       =X'F1'             READ CHARACTER INTO REG A
190h            COMPR    A,S                TEST FOR END OF RECORD
190i            JEQ      *+11               EXIT LOOP IF EOR
190j            STCH     BUFFER,X           STORE CHARACTER IN BUFFER
190k            TIXR     T                  LOOP UNLESS MAXIMUM LENGTH
190l            JLT      *-19                 HAS BEEN REACHED
190m            STX      LENGTH             SAVE RECORD LENGTH
195             LDA      LENGTH             TEST FOR END OF FILE
200             COMP     #0
205             JEQ      ENDFIL             EXIT IF EOF FOUND
210             WRBUFF   05,BUFFER,LENGTH   WRITE OUTPUT RECORD
210a            CLEAR    X                  CLEAR LOOP COUNTER
210b            LDT      LENGTH
210c            LDCH     BUFFER,X           GET CHARACTER FROM BUFFER
210d            TD       =X'05'             TEST OUTPUT DEVICE
210e            JEQ      *-3                LOOP UNTIL READY
210f            WD       =X'05'             WRITE CHARACTER
210g            TIXR     T                  LOOP UNTIL ALL CHARACTERS
210h            JLT      *-14                 HAVE BEEN WRITTEN
215             J        CLOOP              LOOP
220    .ENDFIL  WRBUFF   05,EOF,THREE       INSERT EOF MARKER
220a   ENDFIL   CLEAR    X                  CLEAR LOOP COUNTER
220b            LDT      THREE
220c            LDCH     EOF,X              GET CHARACTER FROM BUFFER
220d            TD       =X'05'             TEST OUTPUT DEVICE
220e            JEQ      *-3                LOOP UNTIL READY
220f            WD       =X'05'             WRITE CHARACTER
220g            TIXR     T                  LOOP UNTIL ALL CHARACTERS
220h            JLT      *-14                 HAVE BEEN WRITTEN
225             J        @RETADR
230    EOF      BYTE     C'EOF'
235    THREE    WORD     3
240    RETADR   RESW     1
245    LENGTH   RESW     1                  LENGTH OF RECORD
250    BUFFER   RESB     4096               4096-BYTE BUFFER AREA
255             END      FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

- For example, in expanding the macro invocation on line 190, the argument F1 is substituted for the parameter &INDEV wherever it occurs in the body of the macro.

Similarly, BUFFER is substituted for &BUFADR, and LENGTH is substituted for &RECLTH.

- The comment lines within the macro body have been deleted. Note that the *macro invocation statement* itself has been included as a comment line. This serves as documentation of the statement written by the programmer.
- The *label* on the macro invocation statement (CLOOP) has been retained as a label on the first statement generated in the macro expansion.

  This allows the programmer to use a *macro instruction* in exactly the same way as an *assembler language mnemonic*.

  Note that the two invocations of WRBUFF specify different arguments, so they produce different expansions.
- After macro processing, the expanded file (Fig 4.2) can be used as input to the assembler.
- In general, the statements that form the expansion of a macro are generated (and assembled) each time the macro is invoked (see Fig 4.2). Statements in a subroutine appear only once, regardless of how many times the subroutine is called (see Fig 2.5).

```
Line          Source statement

  5    COPY      START   0           COPY FILE FROM INPUT TO OUTPUT
 10    FIRST     STL     RETADR      SAVE RETURN ADDRESS
 12              LDB     #LENGTH     ESTABLISH BASE REGISTER
 13              BASE    LENGTH
 15    CLOOP    +JSUB    RDREC       READ INPUT RECORD
 20              LDA     LENGTH      TEST FOR EOF (LENGTH = 0)
 25              COMP    #0
 30              JEQ     ENDFIL      EXIT IF EOF FOUND
 35             +JSUB    WRREC       WRITE OUTPUT RECORD
 40              J       CLOOP       LOOP
 45    ENDFIL    LDA     EOF         INSERT END OF FILE MARKER
 50              STA     BUFFER
 55              LDA     #3          SET LENGTH = 3
 60              STA     LENGTH
 65             +JSUB    WRREC       WRITE EOF
 70              J       @RETADR     RETURN TO CALLER
 80    EOF       BYTE    C'EOF'
 95    RETADR    RESW    1
100    LENGTH    RESW    1           LENGTH OF RECORD
105    BUFFER    RESB    4096        4096-BYTE BUFFER AREA
110    .
115    .         SUBROUTINE TO READ RECORD INTO BUFFER
120    .
125    RDREC     CLEAR   X           CLEAR LOOP COUNTER
130              CLEAR   A           CLEAR A TO ZERO
132              CLEAR   S           CLEAR S TO ZERO
133             +LDT     #4096
135    RLOOP     TD      INPUT       TEST INPUT DEVICE
140              JEQ     RLOOP       LOOP UNTIL READY
145              RD      INPUT       READ CHARACTER INTO REGISTER A
150              COMPR   A,S         TEST FOR END OF RECORD (X'00')
155              JEQ     EXIT        EXIT LOOP IF EOR
160              STCH    BUFFER,X    STORE CHARACTER IN BUFFER
165              TIXR    T           LOOP UNLESS MAX LENGTH
170              JLT     RLOOP         HAS BEEN REACHED
175    EXIT      STX     LENGTH      SAVE RECORD LENGTH
180              RSUB                RETURN TO CALLER
185    INPUT     BYTE    X'F1'       CODE FOR INPUT DEVICE
195    .
200    .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
210    WRREC     CLEAR   X           CLEAR LOOP COUNTER
212              LDT     LENGTH
215    WLOOP     TD      OUTPUT      TEST OUTPUT DEVICE
220              JEQ     WLOOP       LOOP UNTIL READY
225              LDCH    BUFFER,X    GET CHARACTER FROM BUFFER
230              WD      OUTPUT      WRITE CHARACTER
235              TIXR    T           LOOP UNTIL ALL CHARACTERS
240              JLT     WLOOP         HAVE BEEN WRITTEN
245              RSUB                RETURN TO CALLER
250    OUTPUT    BYTE    X'05'       CODE FOR OUTPUT DEVICE
255              END     FIRST
```

Figure 2.5 Example of a SIC/XE program.

4.1.2 Macro Processor Algorithm and Data Structures

- Approach 1: 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 the *second* pass.

However, such a two-pass macro processor would not allow the body of one macro instruction to contain definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

- Approach 2: A *one-pass macro processor* that can alternate between *macro definition* and *macro expansion* is able to handle macros like those in Fig 4.3.

```
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)

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}

                          (b)
```

Figure 4.3 Example of the definition of macros within a macro body.

Because of the one-pass structure, the definition of a macro must appear in the source program *before* any statements that invoke that macro.

- There are three main data structures involved in our macro processor.

  The *macro definitions* themselves are stored in a *definition table* (DEFTAB), which contains the *macro prototype* and the statements that make up the macro body (with a few modifications). Comment lines from the macro definition are not entered into DEFTAB because they will not be part of the macro expansion.

  References to the *macro instruction parameters* are converted to a *positional notation* for efficiency in substituting arguments.

  The *macro names* are entered into NAMTAB, which serves as an index to DEFTAB. For each macro instruction defined, NAMTAB contains pointers to the *beginning* and *end* of the definition in DEFTAB.

- The third data structure is an *argument table* (ARGTAB), which is used during the expansion of macro invocations.

  When a macro invocation statement is recognized, the arguments are stored in ARGTAB according to their position in the argument list.

  As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.

- Fig 4.4 shows portions of the contents of these tables during the processing of program in Fig 4.1.

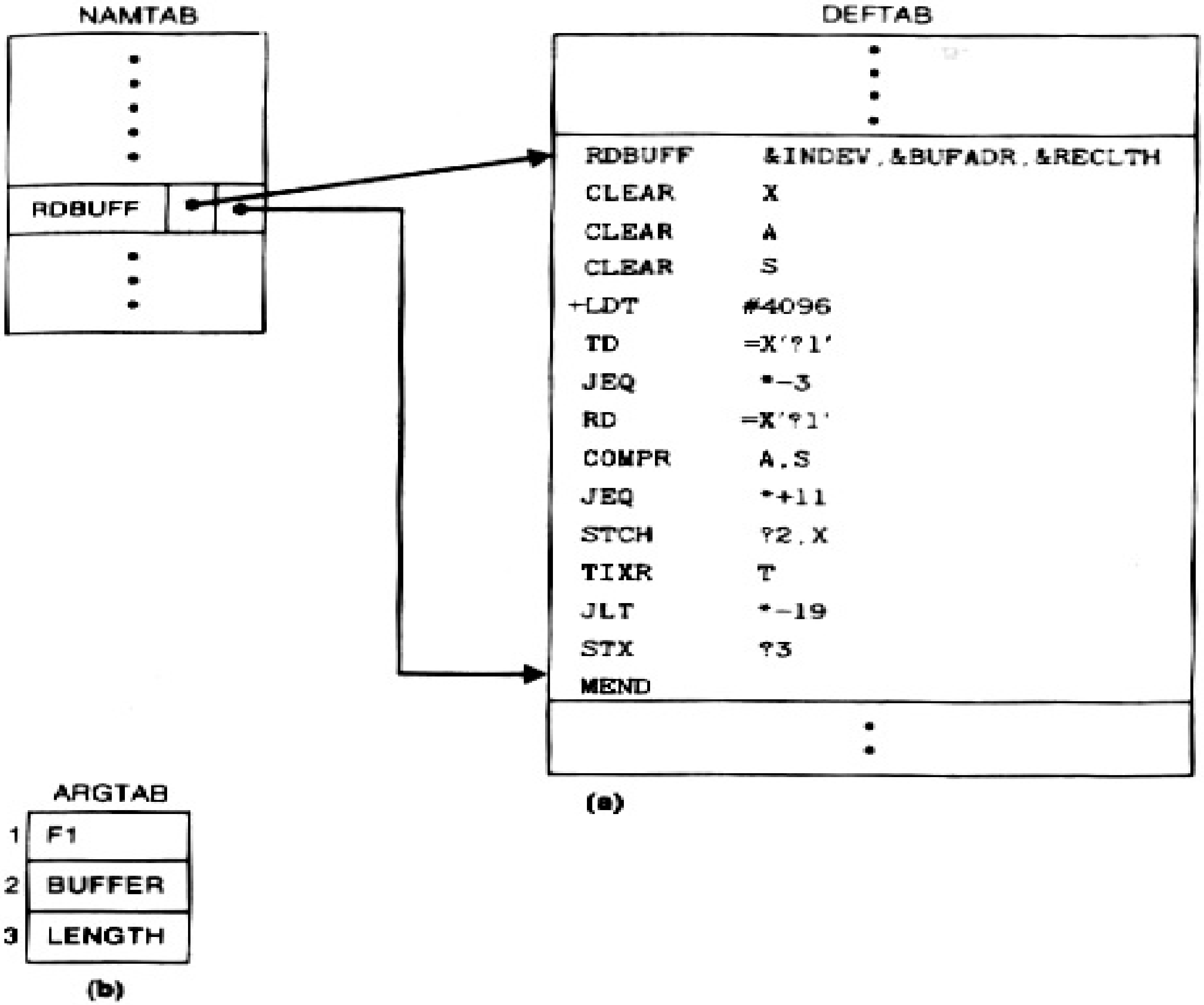

Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF. (b) entries in ARGTAB for invocation of RDBUFF on line 190.

Fig 4.4(a) shows the definition of RDBUFF stored in DEFTAB, with an entry in NAMTAB identifying the beginning and end of the definition.

Note the *positional notation* that has been used for the *parameters*: &INDEV → ?1 (indicating the first parameter in the prototype), &BUFADR → ?2, etc.

Fig 4.4(b) shows ARGTAB as it would appear during expansion of the RDBUFF statement on line 190. In this case (this invocation), the first argument is F1, the second is BUFFER, etc.

- The *macro processor algorithm* is presented in Fig 4.5.

```
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}

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 Algorithm for a one-pass macro processor.

The procedure DEFINE, which is called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB.

EXPAND is called to set up the argument values in ARGTAB and expand a macro invocation statement.

The procedure GETLINE, which is called at several points in the algorithm, gets the next line to be processed. This line may come from DEFTAB (the next line of a macro begin expanded), or from the input file, depending on whether the Boolean variable EXPANDING is set to TRUE or FALSE.

- One aspect of this algorithm deserves further comment: *the handling of macro definitions within macros* (as illustrated in Fig 4.3).

  The DEFINE procedure maintains a counter named LEVEL. Each time a MACRO directive is read, the value of LEVEL is increased by 1.

  Each time an MEND directive is read, the value of LEVEL is decreased by 1.

  When LEVEL reaches 0, the MEND that corresponds to the original MACRO directive has been found.

- The above process is very much like matching *left* and *right parentheses* when scanning an arithmetic expression.

4.2 Machine-Independent Macro Processor Features

4.2.1 Concatenation of Macro Parameters

- Suppose that a program contains one series of variables named by the symbols XA1, XA2, XA3, …, another series named by XB1, XB2, XB3, …, etc.

If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction.

The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.).

- Most macro processors deal with this problem by providing a special *concatenation operator*.

  This operator is the character →.
  For example, the statement LDA **X&ID→1**
  so that the end of the parameter &ID is clearly identified.

  The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so → will not appear in the macro expansion.

- Fig 4.6(a) shows a macro definition that uses the concatenation operator as previously described. Fig 4.6(b) and (c) shows macro invocation statements and the corresponding macro expansions.

```
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
```

(a)

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

(b)

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

(c)

Figure 4.6 Concatenation of macro parameters.

4.2.2 Generation of Unique Labels

- Consider the definition of WRBUFF in Fig 4.1. If a label were placed on the TD instruction on line 135, this label would be defined twice – once for each invocation of WRBUFF.

  This duplicate definition would prevent correct assembly of the resulting expanded program.

- Many macro processors avoid these problems by allowing the creation of special types of labels within macro instructions. Fig 4.7 illustrates one technique for generating unique labels within a macro expansion.

```
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
```

(a)

```
.         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
```

(b)

Figure 4.7 Generation of unique labels within macro expansion.

- Fig 4.7(a) shows a definition of the RDBUFF macro. Labels used within the macro body begin with the special character $.

  Fig 4.7(b) shows a macro invocation statement and the resulting macro expansion. Each symbol beginning with $ has been modified by replacing $ with $AA.

More generally, the character $ will be replaced by $xx, where xx is *a two-character alphanumeric counter* of the number of macro instructions expanded.

For the first macro expansion in a program, xx will have the value AA. For succeeding macro expansions, xx will be set to AB, AC, etc.

4.2.3 Conditional Macro Expansion

- Most macro processors can also modify the sequence of statements generated for a macro expansion, depending on the *arguments* supplied in the macro invocation. This is called conditional macro expansion.
- Fig 4.8 shows the use of one type of conditional macro expansion statement.

```
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
```

(a)

```
         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
```

(b)

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

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

(c)

```
                RDBUFF   F1,BUFF,RLENG,04

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

(d)

Figure 4.8 Use of macro-time conditional statements.

Fig 4.8(a) shows a definition of a macro RDBUFF, the logic and functions of which are similar to those previously discussed.

Two additional parameters are defined in RDBUFF: <u>&EOR</u>, which specifies a hexadecimal character code that marks the end of a record, and <u>&MAXLTH</u>, which specifies the maximum length record that can be read.

- 1st illustration: The statements on <u>lines 44 through 48</u> of this definition illustrate a simple macro-time conditional structure.

  The <u>IF statement evaluates a *Boolean expression*</u> that is its operand (In this case, it is [&MAXLTH EQ ‘ ‘].). If TRUE, the statements following the IF are generated until an

ELSE is encountered (Line 45 is generated.).

If FALSE, these statements are skipped, and the statements following the ELSE are generated (Line 47 is generated.).

The ENDIF statement terminates the conditional expression that was begun by the IF statement.

- 2nd illustration: On line 26 through 28, line 27 is another *macro processor directive* (SET). This SET statement assigns the value 1 to &EORCK.

  The symbol &EORCK is a *macro time variable*, which can be used to store working values during the macro expansion. Note any symbol that begins with the character & and that is not a macro instruction parameter is assumed to be a *macro-time variable*. All such variables are initialized to a value of 0.

- Other illustrations: On line 38 through 43 and line 63 through 73.

- Fig 4.8 (b-d) shows the expansion of 3 different macro invocation statements that illustrate the operation of the IF statements in Fig 4.8(a).

- Note that the macro processor must maintain a *symbol table* that contains the values of all macro-time variables used.

  Entries in this table are made or modified when SET statements are processed. The table is used to look up the current value of a macro-time variable whenever it is required.

- Syntax 1 – IF (Boolean Exp.) (statements) ELSE (statements) ENDIF: If IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.

If TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement. If an ELSE is found, the macro processor then skips lines in DEFTAB until the next ENDIF. Upon reaching the ENDIF, it resumes expanding the macro in the usual way.

If FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion.

- The implementation outlined above does not allow for nested IF structures.
- It is extremely important to understand that the testing of Boolean expressions in IF statements occurs at the time macros are expanded.

  By the time the program is assembled, all such decisions (must) have been made.

  The conditional macro expansion directives (must) have been removed. The same applies to the assignment of values to macro-time variables, and to the other conditional macro expansion directives.
- Fig 4.9 shows the use of *macro-time loop statements*. The definition in Fig 4.9(a) uses a macro-time loop statement WHILE.

```
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
```

(a)

```
.         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
```

(b)

Figure 4.9 Use of macro-time looping statements.

The WHILE statement specifies that the following lines, until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. Note that all the generation is done at the macro expansion time. The conditions to be tested involve macro-time variables and arguments, not run-time data values.

- The use of the WHILE-ENDW structure is illustrated on lines 63 through 73 of Fig 4.9(a). The macro-time variables &EORCT has previously been set (line 27) to the value %NITEMS(&EOR).

  %NITEMS is a *macro processor function* that returns as its value the number of members in an argument list. For example, if the argument corresponding to &EOR is (00, 03, 04), then %NITEMS(&EOR) has the value 3.

  The macro-time variable &CTR is used to count the number of times the lines following the WHILE statement have been generated. The value of &CTR is initialized to 1 (line 63), and incremented by 1 each time through the loop (line 71).

  Fig 4.9(b) shows the expansion of a macro invocation statement using the definition in Fig 4.9(a).

- Syntax 2 – WHILE (Boolean Exp.) (statements) ENDW: When a WHILE statement is encountered during macro expansion, the specified Boolean expression is evaluated.

  If the value of this expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ENDW statement, and then resumes normal macro expansion.

  If TRUE, the macro processor continues to process lines from DEFTAB in the usual way until the next ENDW statement. When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value of this expression as previously described.

- Note that no nested WHILE structures are allowed.

4.2.4 Keyword Macro Parameters

- All the macro instruction definitions we have seen thus far used *positional parameters*. That is, *parameters* and *arguments* were associated with each other according to their positions in the macro prototype and the macro invocation statement.

- With *positional parameters*, the programmer must be careful to specify the arguments in the proper order. If an argument is to be omitted, the macro invocation statement must contain a null argument (two consecutive commas) to maintain the correct argument positions.

  For example, a certain macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only 3rd and 9th parameters are to be specified. Then, the macro invocation might look like GENER , , DIRECT, , , , , , 3.

- Using a different form of parameter specification, called *keyword parameters*, each argument value is written with a keyword that names the corresponding parameter.

  Arguments may appear in any order.

  For example, if 3rd parameter in the previous example is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation statement would be
  GENER TYPE=DIRECT, CHANNEL=3.

- Fig 4.10(a) shows a version of the RDBUFF macro definition using keyword parameters.

```
RDBUFF  MACRO  &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
        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
       +LDT    #&MAXLTH        SET MAXIMUM RECORD LENGTH
$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
```

(a)

Figure 4.10 Use of keyword parameters in macro instructions.

In the macro prototype, each parameter name is followed by an *equal sign* (=), which identifies a keyword parameter.

After = sign, a default value is specified for some of the parameters. The parameter is assumed to have this default value if its name does not appear in the macro invocation statement.

Default values can simplify the macro definition in many cases.

4.3 Macro Processor Design Options

4.3.1 Recursive Macro Expansion

- Fig 4.11 shows an example of macro invocations within macro definitions.

```
RDBUFF   MACRO    &BUFADR,&RECLTH,&INDEV
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR    X                   CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096               SET MAXIMUM RECORD LENGTH
$LOOP    RDCHAR   &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
```

(a)

```
RDCHAR   MACRO    &IN
.
.        MACRO TO READ CHARACTER INTO REGISTER A
.
         TD       =X'&IN'             TEST INPUT DEVICE
         JEQ      *-3                 LOOP UNTIL READY
         RD       =X'&IN'             READ CHARACTER
         MEND
```

(b)

```
         RDBUFF   BUFFER,LENGTH,F1
```

(c)

Figure 4.11 Example of nested macro invocation.

Fig 4.11(a) shows the definition of RDBUFF. In this case, a macro invocation (RDCHAR) is invocated in the body of RDBUFF and a related macro instruction already exists.

The definition of RDCHAR appears in Fig 4.11(b).

- Unfortunately, the macro processor design we have discussed previously cannot handle such *invocations of macros within macros*.

  Fig 4.11(c) shows a macro invocation statement of RDBUFF. According to the algorithm in Fig 4.5, the procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as shown in page 201.

The processing would proceed normally until line 50, which contains a statement invoking RDCHAR. At that point, PROCESSLINE would call EXPAND again. This time, ARGTAB would look like as shown in page 201.

The expansion of RDCHAR would also proceed normally. At the end of this expansion, however, a *problem* would appear. When the end of the definition of RDCHAR was recognized, EXPANDING would be set to FALSE. Thus, the macro processor would "*forget*" that it had been *in the middle of expanding a macro* when it encountered the RDCHAR statement.

In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the values in ARGTAB were overwritten with the arguments from the invocation of RDCHAR.

- This cause of these difficulties is the *recursive call* of the procedure EXPAND.

  When the RDBUFF macro invocation is encountered, EXPAND is called. Later, it calls PROCESSLINE for line 50, which results in another call to EXPAND before a return is made from the original call.

  A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.

- These problems are not difficult to solve if the macro processor is being written in a programming language that allows recursive calls.

- If a programming language that supports recursion is not available, the programmer must take care of handling such items as *return addresses* and *values of local variables* (that is, handling by looping structure and data values being saved on a stack).

4.3.2 General-Purpose Macro Processors

- The most common use of macro processors is as an aid to assembler language programming. Macro processors have also been developed for some high-level programming languages.

  These *special-purpose macro processors* are similar in general function and approach. However, the details differ from language to language.

- The *general-purpose macro processors* are not dependent on any particular programming language, but can be used with a variety of different languages.

- There are relatively few general-purpose macro processors. The major reason is the large number of details that must be dealt within a real programming language. That is to say, a general-purpose facility must provide some way for a user to define the specific set of rules to be followed. Therefore, there are some difficulties in some way.

- Case 1: Comments are usually ignored by a macro processor (at least in scanning for parameters). However, each programming language has its own methods for identifying comments.

- Case 2: Another difference between programming languages is related to their facilities for grouping together *terms*, *expressions*, or *statements*. A general-purpose macro processor may need to take these groupings into account in scanning the source statements.

- Case 3: Languages differ substantially in their restrictions on the length of *identifiers* and the rules for the formation of constants (i.e. the *tokens* of the programming language – for example, identifiers, constants, operators, and keywords).

- Case 4: 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.

4.3.3 Macro Processing within Language Translators

- The macro processors might be called *preprocessors*. Consider an alternative: combining the macro processing functions with the language translator itself.

- The simplest method of achieving this sort of combination is a *line-by-line* macro processor.

  Using this approach, the macro processor reads the source program statements and performs all of its functions as previously described.

  The output lines are then passed to the language translator as they are generated (one at a time), instead of being written to an expanded source file.

  Thus, the macro processor operates as a sort of *input routine* for the assembler or compiler.

- Although a line-by-line macro processor may use some of the same utility routines as the language translator, the functions of macro processing and program translation are still relatively independent.

- There exists even closer cooperation between the macro processor and the assembler or compiler. Such a scheme can be thought of as a language translator with an *integrated* macro processor.

An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

For example, at a relatively simple level of cooperation, the macro processor may use the results of such translator operations as scanning for symbols, constants, etc. The macro processor can simply use the results without being involved in such details as multiple-character operators, continuation lines, and the rules for token formation.

- There are disadvantages to integrated and line-by-line macro processors.

  They must be specially designed and written to work with a particular implementation of an assembler or compiler.

  The costs of macro processor development must be added to the cost of the language translator, resulting in a more expensive piece of software.

  The size may be a problem if the translator is to run on a computer with limited memory.

4.4 Implementation Examples

4.4.1 (Skip)

4.4.2 ANSI C Macro Language

- In the ANSI C language, definitions and invocations of macros are handled by a preprocessor. This preprocessor is generally not integrated with the rest of compiler. Its operation is similar to the macro processor we discussed before.
- Two simple (and commonly used) examples of ANSI C macro definitions:

```
#define    NULL    0
#define    EOF     (-1)
```

After these definitions, every occurrence of NULL will be replaced by 0, and every occurrence of EOF will be replaced by (-1).

- It is also possible to use macros like this to make limited changes in the syntax of the language. For example, after defining the macro

  #define EQ ==.

  A programmer could write while (I EQ 0)…
  The macro processor would convert this into while (I == 0) …

- ANSI C macros can also be defined with parameters. Consider, for example, the macro definition

  #define ABSDIFF(X,Y) ((X) > (Y)) **?** (X) – (Y) **:** (Y) – (X))

  For example, ABSDIFF (I+1, J-5) would be converted by the macro processor into

  ((I+1) > (J-5) **?** (I+1) – (J-5) **:** (J-5) – (I+1)).

  The macro version can also be used with different types of data. For example, we could invoke the macro as ABSDIFF(I, 3.14159) or ABSDIFF('D', 'A').

- It is necessary to be very careful in writing macro definitions with parameters. The macro processor simply makes string substitutions, without considering the syntax of the C language.

  For example, if we had written the definition of ABSDIFF as

  #define ABSDIFF(X, Y) X>Y **?** X-Y **:** Y-X. The macro invocation ABSDIFF(3+1, 10-8) would be expanded into

3+1 > 10-8 **?** 3+1-10-8 **:** 10-8–3+1.

- The ANSI C preprocessor also provides *conditional compilation statements*. For example, in the sequence

  ```
  #ifndef   BUFFER_SIZE
  #define   BUFFER_SIZE    1024
  #endif
  ```

  the #define will be processed only if BUFFER_SIZE has not already been defined.

- *Conditionals* are also often used to control the inclusion of debugging statements in a program. (See page 213 for example.)

4.4.3 (Skip)