KTU STUDY MATERIALS

APJ ABDUL KALAM
TECHNOLOGICAL UNIVERSITY

INTRODUCTION TO SYSTEM SOFTWARE AND MACHINE STRUCTURE

1.1 SYSTEM SOFTWARE

- System software consists of a variety of programs that support the operation of a computer.
- It is a set of programs to perform a variety of system functions as file editing, resource management, I/O management and storage management.
- The characteristic in which system software differs from application software is machine dependency.
- An application program is primarily concerned with the solution of some problem, using the computer as a tool.
- System programs on the other hand are intended to support the operation and use of the computer itself, rather than any particular application.
- For this reason, they are usually related to the architecture of the machine on which they are run.
- For example, assemblers translate mnemonic instructions into machine code. The instruction formats, addressing modes are of direct concern in assembler design.
- There are some aspects of system software that do not directly depend upon the type of computing system being supported. These are known as machine-independent features.
- For example, the general design and logic of an assembler is basically the same on most computers.System software can be broadly classified into
    - System control programs: controls the execution of program manages the storage, processing resources of the computer and performs other management and monitoring functions. Operating systems, DBMS and communication monitors are the examples of such systems.
    - **System support programs**: provide routine service functions to the other computer programs and computer users. Eg: Utilities, libraries, performance monitors and job accounting.
    - **System development programs:** programs assist in the creation of application program.

TYPES OF SYSTEM SOFTWARE:

1. Operating system
2. Language translators
    a. Compilers
    b. Interpreters
    c. Assemblers
    d. Preprocessors
3. Loaders
4. Linkers
5. Macro processors

Operating System

- It is the most important system program that act as an interface between the users and the system. It makes the computer easier to use.
- It provides an interface that is more user-friendly than the underlying hardware.
- The functions of OS are:
    1. Process management
    2. Memory management
    3. Resource management
    4. I/O operations
    5. Data management
    6. Providing security to user's job.

Language Translators

It is the program that takes an input program in one language and produces an output in another language.

Source Program → [**Language Translator**] → **Object Program**

Compilers

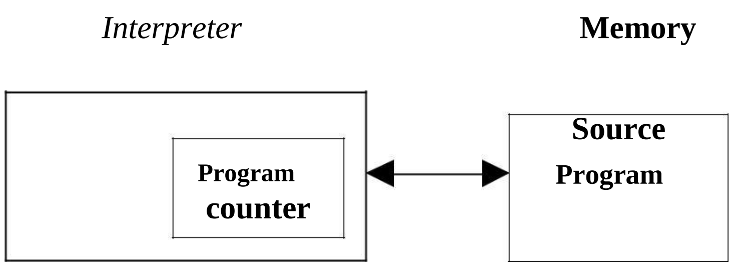
- A compiler is a language program that translates programs written in any high-level language into its equivalent machine language program.
- It bridges the semantic gap between a programming language domain and the execution domain.
- Two aspects of compilation are:
    - Generate code to increment meaning of a source program in the execution domain.
    - Provide diagnostics for violation of programming language, semantics in a source program.
- The program instructions are taken as a whole.

High level language → [**Compiler**] → **Machine language program**

Interpreters:

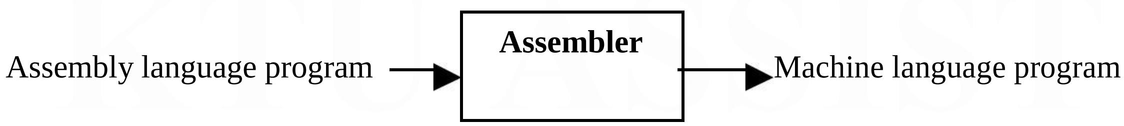
- It is a translator program that translates a statement of high-level language to machine language and executes it immediately. The program instructions are taken line by line.
- The interpreter reads the source program and stores it in memory.
- During interpretation, it takes a source statement, determines its meaning and performs actions which increments it. This includes computational and I/O actions.
- Program counter (PC) indicates which statement of the source program is to be interpreted next. This statement would be subjected to the interpretation cycle.
- The interpretation cycle consists of the following steps:
    - Fetch the statement.
    - Analyze the statement and determine its meaning.
    - Execute the meaning of the statement.
- The following are the characteristics of interpretation:

  - The source program is retained in the source form itself, no target program exists.
  - A statement is analyzed during the interpretation.



Assemblers:

- Programmers found it difficult to write or red programs in machine language. In a quest for a convenient language, they began to use a mnemonic (symbol) for each machine instructions which would subsequently be translated into machine language.
- Such a mnemonic language is called Assembly language.
- Programs known as Assemblers are written to automate the translation of assembly language into machine language.



- Fundamental functions:
  1. Translating mnemonic operation codes to their machine language equivalents.
  2. Assigning machine addresses to symbolic tables used by the programmers.

APPLICATION SOFTWARES

Application software is a computer software which is designed to help the user in performing single or multiple related tasks. In other words, application software is actually a subclass of computer software, which employs the capabilities of a computer directly to a task that the user wishes it to perform. Hence, often application software is looked upon as software as well as its implementation.

There are different types of application software, which include Enterprise Resource Planning Software, Accounting Software, Customer Relationship Management Software, Graphics Software, Media Players, etc.

Application software vs System software

Subject	Application Software	System Software
Definition	Application software is computer software designed to help the user to perform specific tasks.	System software is computer software designed to operate the computer hardware and to provide a platform for running application software.
Purpose	It is specific purpose software.	It is general-purpose software.
Classification	Package Program, Customized Program	Time Sharing, Resource Sharing, Client Server Batch Processing Operating System Real time Operating System Multi-processing Operating System Multi-programming Operating System Distributed Operating System
Environment	Application Software performs in a environment which created by System/Operating System	System Software Create his own environment to run itself and run other application.
Execution Time	It executes as and when required.	It executes all the time in computer.
Essentiality	Application is not essential for a computer.	System software is essential for a computer
Number	The number of application software is much more than system software.	The number of system software is less than application software.

THE SIMPLIFIED INSTRUCTIONAL COMPUTER (SIC):

It is similar to a typical microcomputer. It comes in two versions:

- The standard model
- XE version

SIC Machine Structure:

Memory:

- It consists of bytes (8 bits) ,words (24 bits which are consecutive 3 bytes) addressed by the location of their lowest numbered byte.
- There are totally 32,768 bytes in memory.

Registers:

There are 5 registers namely

1. **Accumulator (A):** Used Accumulator is a special purpose register used for arithmetic operations
2. **Index Register(X)** Stores and calculates addresses.

- **Linkage Register (L):** Linkage register stores the return address of the jump of subroutine instructions (JSUB).
- **Program Counter (PC):** Program counter contains the address of the current instructions being executed.
- **Status Word (SW)**: Status word contains a variety of information including the condition code.

Data formats:

- Integers are stored as 24-bit binary numbers: 2's complement representation is used for negative values characters are stored using their 8 bit ASCII codes.
- They do not support floating – point data items.

Instruction formats:

All machine instructions are of 24-bits wide

Opcode (8)	**X (1)**	**Address (15)**

X-flag bit that is used to indicate indexed-addressing mode.

Addressing modes:

- Two types of addressing are available namely,
    1. Direct addressing mode
    2. Indexed addressing mode or indirect addressing mode

Mode	**Indication**	**Target Address calculation**
Direct	X=0	TA=Address
Indexed	X=1	TA=Address + (X)

- Where(x) represents the contents of the index register(x)

Instruction set:

It includes instructions like:

1. Data movement instruction: Basic set of instructions load and store registers.
   Ex: LDA, LDX, STA, STX.
2. Arithmetic operating instructions: Arithmetic operations involve register A and a word in memory and result is left in the register.
   Ex: ADD, SUB, MUL, DIB.
3. Branching instructions Ex: JLT, JEQ, TGT.
4. Subroutine linkage instructions: These instructions are used for subroutine linkage.
   Ex: **JSUB**: jumps to the subroutine placing the return address in register L.
   RSUB: returns by jumping to the address contained in register L

Input and Output:

I/O is performed by transferring one byte at a time to or from the rightmost 8 bits of register A. Each device is assigned a unique 8-bit code.

- There are 3 I/O instructions,
  1) The Test Device (TD) instructions tests whether the addressed device is ready to send or receive a byte of data. Check condition code to see if device is ready. If CC setting is < the device is ready, if setting is = device is not ready.
  2) A program must wait until the device is ready, and then execute a Read Data (RD) or Write Data (WD).
  3) The sequence must be repeated for each byte of data to be read or written.

1.3 SIC/XE ARCHITECTURE & SYSTEM SPECIFICATION

Memory:

- 1 word = 24 bits (3 8-bit bytes)
- Total (SIC/XE) = 2^{20} (1,048,576) bytes (1Mbyte)

Registers:

- 10 x 24 bit registers

MNEMONIC	**Register**	**Purpose**
A	0	Accumulator
X	1	Index register
L	2	Linkage register (JSUB/RSUB)
B	3	Base register
S	4	General register
T	5	General register
F	6	Floating Point Accumulator (48 bits)
PC	8	Program Counter (PC)
SW	9	Status Word (includes Condition Code, CC)

Data Format:

- Integers are stored in 24 bit, 2's complement format
- Characters are stored in 8-bit ASCII format

- Floating point is stored in 48 bit signed-exponent-fraction format:

s	exponent {11}	fraction {36}

- The fraction is represented as a 36 bit number and has value between 0 and 1.
- The exponent is represented as a 11 bit unsigned binary number between 0 and 2047.
- The sign of the floating point number is indicated by s : 0=positive, 1=negative.
- Therefore, the absolute floating point number value is: $f*2^{(e-1024)}$

Instruction Format:

- There are 4 different instruction formats available:

Format 1 (1 byte):

op {8}

Format 2 (2 bytes):

op {8}	r1 {4}	r2 {4}

Format 3 (3 bytes):

op {6}	n	i	x	b	p	e	displacement {12}

Format 4 (4 bytes):

n i x b p e address {20}

op {6}	n	i	x	b	p	e	address{20}

Formats 3 & 4 introduce addressing mode flag bits:

- n=0 & i=1
  Immediate addressing - TA is used as an operand value (no memory reference)
- n=1 & i=0
  Indirect addressing - word at TA (in memory) is fetched & used as an address to fetch the operand from
- n=0 & i=0
  Simple addressing TA is the location of the operand
- n=1 & i=1
  Simple addressing same as n=0 & i=0

Flag x:

x=1 Indexed addressing adds contents of X register to TA calculation

Flag b & p (Format 3 only):

- b=0 & p=0
  Direct addressing displacement/address field contains TA (Format 4 always uses direct addressing)
- ***b=0 & p=1***
  PC relative addressing - TA=(PC)+disp (-2048<=disp<=2047)*
- b=1 & p=0
  Base relative addressing - TA=(B)+disp (0<=disp<=4095)**

Flag e:

e=0 use Format 3 e=1

use Format 4

Instructions:

SIC provides 26 instructions, SIC/XE provides an additional 33 instructions (59 total)

SIC/XE has 9 categories of instructions:

- **Load/store registers** (LDA, LDX, LDCH, STA, STX, STCH, etc.)
- **Integer arithmetic operations** (ADD, SUB, MUL, DIV) these will use register A and a word in memory, results are placed into register A
- **Compare (COMP)** compares contents of register A with a word in memory and sets CC (Condition Code) to <, >, or =
- **Conditional jumps** (JLT, JEQ, JGT) - jumps according to setting of CC
- **Subroutine linkage** (JSUB, RSUB) - jumps into/returns from subroutine using register L
- **Input & output control** (RD, WD, TD) - see next section
- **Floating point arithmetic operations** (ADDF, SUBF, MULF, DIVF)
- **Register manipulation,** operands-from-registers, and register-to-register arithmetics (RMO, RSUB, COMPR, SHIFTR, SHIFTL, ADDR, SUBR, MULR, DIVR, etc)

Input and Output (I/O):

- 2^8 (256) I/O devices may be attached, each has its own unique 8-bit address
- 1 byte of data will be transferred to/from the rightmost 8 bits of register A

Three I/O instructions are provided:

- RD Read Data from I/O device into A
- WD Write data to I/O device from A
- TD Test Device determines if addressed I/O device is ready to send/receive a byte of data. The CC (Condition Code) gets set with results from this test:
  < device is ready to send/receive
  = device isn't ready

SIC/XE Has capability for programmed I/O (I/O device may input/output data while CPU does other work) - 3 additional instructions are provided:

- SIO Start I/O
- HIO Halt I/O
- TIO Test I/O

MACROPROCESSORS

INTRODUCTION

Macro Instructions

A macro instruction (macro)

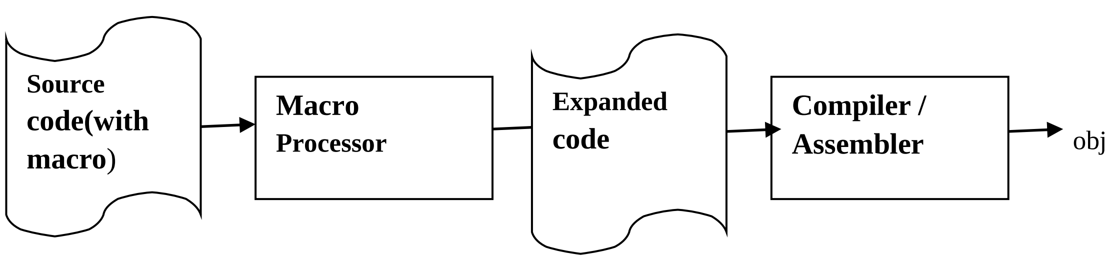
- It is simply a notational convenience for the programmer to write a shorthand version of a program(module programming).
- It represents a commonly used group of statements in the source program.
- It is replaced by the **macro processor** with the corresponding group of source language statements. This operation is called “expanding the macro”

For example:

- Suppose it is necessary to save the contents of all registers before calling a subroutine.
- This requires a sequence of instructions.
- We can define and use a macro, SAVEREGS, to represent this sequence of instructions.

Macro Processor

- A macro processor
  - Its functions essentially involve the substitution of one group of characters or lines for another.
  - Normally, it performs no analysis of the text it handles.
  - It doesn’t concern the meaning of the involved statements during macro expansion.
  - Therefore, the design of a macro processor generally is machine independent.
- Macro processors are used in
  - assembly language
  - high-level programming languages, e.g., C or C++
  - OS command languages
  - general purpose



Format of macro definition

A macro can be defined as follows

MACRO Name [List of Parameters] -MACRO pseudo-op shows start of macro definition.
Macro name with a list of formal parameters.

….
……. Sequence of assembly language instructions.

MEND MEND (MACRO-END) Pseudo shows the end of macro definition.

Example:

```
MACRO   SUM X, Y
        LDA   X
        MOV BX, X
        LDA Y
        ADD BX
MEND
```

1 BASIC MACROPROCESSOR FUNCTIONS

The fundamental functions common to all macro processors are:

1. Macro Definition
2. Macro Invocation
3. Macro Expansion

Macro Definition and Expansion

- Two new assembler directives are used in macro definition:

label	**op**	**operands**
name	**MACRO**	**parameters**
	:	
	body	
	:	
	MEND	

  - MACRO: identify the beginning of a macro definition
  - MEND: identify the end of a macro definition

- Prototype for the macro:
Each parameter begins with '&'
- Body: The statements that will be generated as the expansion of the macro.

```
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
```

It shows an example of a SIC/XE program using macro Instructions.

- This program defines and uses two macro instructions, RDBUFF and WRDUFF .
- Two Assembler directives (MACRO and MEND) are used in macro definitions.
- The first MACRO statement identifies the beginning of macro definition.
- The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field identify the parameters of macro instruction.
- In our macro language, each parameter begins with character &, which facilitates the substitution of parameters during macro expansion.
- The macro name and parameters define the pattern or prototype for the macro instruction used by the programmer. The macro instruction definition has been deleted since they have been no longer needed after macros are expanded.
- Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.
- The arguments and parameters are associated with one another according to their positions.

Macro Invocation

- A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.
- The processes of macro invocation and subroutine call are quite different.
  - Statements of the macro body are expanded each time the macro is invoked.
  - Statements of the subroutine appear only one; regardless of how many times the subroutine is called.
- The macro invocation statements treated as comments and the statements generated from macro expansion will be assembled as though they had been written by the programmer.

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.

Example of a macro expansion

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

- In expanding the macro invocation on line 190, the argument F1 is substituted for the parameter and INDEV wherever it occurs in the body of the macro.
- Similarly BUFFER is substituted for BUFADR and LENGTH is substituted for RECLTH.
- Lines 190a through 190m show the complete expansion of the macro invocation on line 190.
- 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.
- After macro processing the expanded file can be used as input to assembler.
- The macro invocation statement will be treated as comments and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

Macro Processor Algorithm and Data Structures

- 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 would not allow the body of one macro instruction to contain definitions of other macros.

Example 1:

```
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}
```

Example 2:

```
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}
```

- Defining MACROS or MACROX does not define RDBUFF and the other macro instructions. These definitions are processed only when an invocation of MACROS or MACROX is expanded.
- A one pass macroprocessor that can alternate between macro definition and macro expansion is able to handle macros like these.
- There are 3 main data structures involved in our macro processor.

Definition table (DEFTAB)

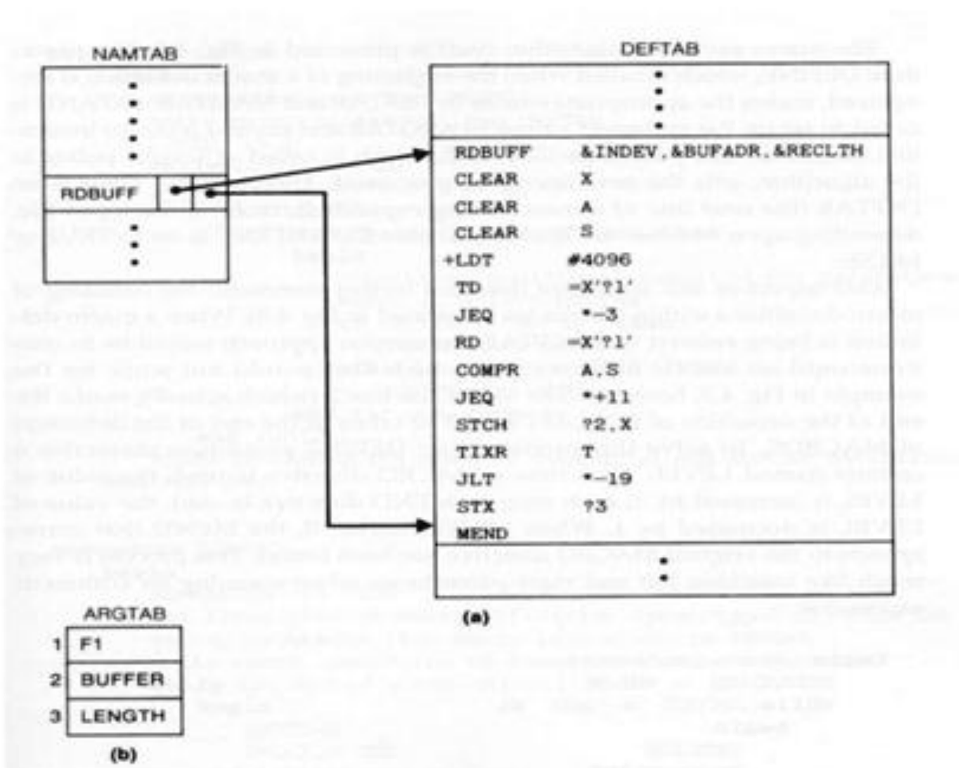
1. The macro definition themselves are stored in definition table (DEFTAB), which contains the macro prototype and statements that make up the macro body.
2. Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion.

Name table (NAMTAB)

1. References to macro instruction parameters are converted to a positional entered into NAMTAB, which serves the index to DEFTAB.
2. For each macro instruction defined, NAMTAB contains pointers to beginning and end of definition in DEFTAB.

Argument table (ARGTAB)

1. The third Data Structure in an argument table (ARGTAB), which is used during expansion of macro invocations.
2. When macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.
3. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.



- The position notation is used for the parameters. The parameter &INDEV has been converted to ?1, &BUFADR has been converted to ?2.
- When the ?n notation is recognized in a line from DEFTAB, a simple indexing operation supplies the property argument from ARGTAB.

Algorithm:

- 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 gets the next line to be processed
- This line may come from DEFTAB or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

```
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 *(cont'd)*

Generation of Unique Labels

- Labels in the macro body may cause "duplicate labels" problem if the macro is invocated and expanded multiple times.
- Use of relative addressing at the source statement level is very inconvenient, error-prone, and difficult to read.
- It is highly desirable to

1. Let the programmer use label in the macro body
    - Labels used within the macro body begin with $.
2. Let the macro processor generate unique labels for each macro invocation and expansion.
    - During macro expansion, the $ will be replaced with $xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded.
    - XX=AA, AB, AC …….

RDBUFF definition

```
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
```

Labels within the macro body begin with the special character $.

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

Macro expansion

```
          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
```

- Unique labels are generated within macro expansion.
- Each symbol beginning with $ has been modified by replacing $ with $AA.
- 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.

Keyword Macro Parameters

- **Positional parameters**
  - Parameters and arguments are associated according to their positions in the macro prototype and invocation. The programmer must specify the arguments in proper order.
  - If an argument is to be omitted, a null argument should be used to maintain the proper order in macro invocation statement.
  - For example: Suppose a macro instruction GENER has 10 possible Parameters, but in a particular invocation of the macro only the 3rd and 9th parameters are to be specified.
  - The statement is GENER ,,DIRECT,,,,,,3.
  - It is not suitable if a macro has a large number of parameters, and only a few of these are given values in a typical invocation.
- **Keyword parameters**
  - Each argument value is written with a keyword that names the corresponding parameter.
  - Arguments may appear in any order.
  - Null arguments no longer need to be used.
  - If the 3rd parameter is named &TYPE and 9th parameter is named &CHANNEL, the macro invocation would be
    GENER TYPE=DIRECT, CHANNEL=3.
  - It is easier to read and much less error-prone than the positional method.

Consider the example

- Here each parameter name is followed by equal sign, which identifies a keyword parameter and a default value is specified for some of the parameters.

```
25   RDBUFF   MACRO   &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
26            IF      (&EOR NE '')
27   &EORCK   SET     1
28            ENDIF
30            CLEAR   X               CLEAR LOOP COUNTER
35            CLEAR   A
38            IF      (&EORCK EQ 1)
40            LDCH    =X'&EOR'        SET EOR CHARACTER
42            RMO     A,S
43            ENDIF
47           +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
50   $LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
55            JEQ     $LOOP           LOOP UNTIL READY
60            RD      =X'&INDEV'      READ CHARACTER INTO REG A
63            IF      (&EORCK EQ 1)
65            COMPR   A,S             TEST FOR END OF RECORD
70            JEQ     $EXIT           EXIT LOOP IF EOR
73            ENDIF
75            STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
80            TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85            JLT     $LOOP             HAS BEEN REACHED
90   $EXIT    STX     &RECLTH         SAVE RECORD LENGTH
95            MEND
```

```
     .        RDBUFF   BUFADR=BUFFER,RECLTH=LENGTH
```

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

Here the value if &INDEV is specified as F3 and the value of &EOR is specified as null.

Praseeda K Gopinadhan, CSE, SNGCE

MACROPROCESSOR DESIGN OPTIONS

Recursive Macro Expansion

```
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
```

```
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
```

- RDCHAR:
  - read one character from a specified device into register A
  - should be defined beforehand (i.e., before RDBUFF)

Praseeda K Gopinadhan, CSE, SNGCE

Implementation of Recursive Macro Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion, e.g., RDBUFF BUFFER, LENGTH, F1
- Reasons:
  1) The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.
  2) The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, that is, the macro process would forget that it had been in the middle of expanding an "outer" macro.
  3) A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.
- Solutions:
  1) Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  2) Use a stack to take care of pushing and popping local variables and return addresses.
- Another problem can a macro invoke itself recursively?

<u>Algorithm</u>

```
procedure EXPAND
 level=0; SP=-1
     begin
         set S(SP+N+2) =SP
         set SP=SP+N+2
         set S(SP+1) = DEFTAB index from NAMTAB
         set up macro call argument list array in S(SP+2).......S(SP+N+1) where N= number of
        arguments
         while not end of macro definition and level !=0 do
          begin
                GETLINE
                PROCESSLINE
        end {while}
         set N=SP-S(SP)-2
         set SP=S(SP)
     End {EXPAND}


procedure GETLINE
begin if sp!=-1 then
   begin
```

```
        increment DEFTAB pointer to next entry
        Set S(SP+1) = S(SP+1) + 1
        get the line form DEFTAB with the pointer S(SP+1)
        substitute arguments from macro call
             S(SP+2)…………..S(SP+N+1)
      end
 else
      read next line from input file
end {GETLINE}
```

END