TABLE OF CONTENTS

VTU QUESTION PAPER SOLUTION

UNIT 1: MACHINE ARCHITECTURE

DECEMBER 2010

1a) Generate the target address for the given following

The target address generated ad as follows

(i) 032600= 103000

(ii)03C300=00C303

(iii)0310C303=003030

1c) Write SIC instructions to clear 20 byte string to all blanks

```
        LDX    ZERO
CLOOP   TD     INDEV
        JEQ    CLOOP
        RD     INDEV
        STCH   RECORD, X
        TIX    B200
        JLT    CLOOP
        .
        .
INDEV   BYTE   X 'F5'
RECORD  RESB   200
ZERO    WORD   0
B200    WORD   200
```

MAY /JUNE 2010

1a)What are the Registers used in SIC machine architecture

There are five registers, each 24 bits in length. Their mnemonic, number and use are given in the following table.

Mnemonic	Number	Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB
PC	8	Program counter
SW	9	Status word, including CC

1b) Write instructions for SIC/XE to set ALPHA = GAMMA *BETA-9

```
LDA GAMMA
MUL BETA
SUB  9
STA  ALPHA
```

1C) Write a SIC program to copy character string.

```
        LDA     #5
        STA     ALPHA
        LDA     #90
        STCH    C1
          .
          .
```

```
ALPHA   RESW    1
C1      RESB    1
```

DEC 09/JAN 10

1a) Define System software.

System software consists of a variety of programs that support the operation of a computer

System software – support operation and use of computer. Application software - solution to a problem. Assembler translates mnemonic instructions into machine code. The instruction formats, addressing modes etc., are of direct concern in assembler design.

There are aspects of system software that do not directly depend upon the type of computing system, general design and logic of an assembler, general design and logic of a compiler and code optimization techniques, which are independent of target machines. Likewise, the process of linking together independently assembled subprograms does not usually depend on the computer being used.

1b) What are the Instruction format and addressing modes of SIC/XE? (DEC 08/JAN 09)

Instruction Formats:

The new set of instruction formats fro SIC/XE machine architecture are as follows. Format 1 (1 byte): contains only operation code (straight from table).

Format 2 (2 bytes): first eight bits for operation code, next four for register 1 and following four for register 2.

The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).

Format 3 (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).

Format 4 (4 bytes): same as format 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits to be represented.

Format 1 (1 byte)

8
op

Format 2 (2 bytes)

8	4	4
op	r1	r2

Formats 1 and 2 are instructions do not reference memory at all

Format 3 (3 bytes)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	disp

Format 4 (4 bytes)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	e	address

Addressing modes & Flag Bits

Five possible addressing modes plus the combinations are as follows.

- **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.

- **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)

- **Immediate**(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)

- **Indirect**(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

- **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings

e - > e = 0 means format 3, e = 1 means format 4

Bits x,b,p : Used to calculate the target address using relative, direct, and indexed addressing Modes

Bits i and n: Says, how to use the target address

b and p - both set to 0, disp field from format 3 instruction is taken to be the target address. For a x - x is set to 1, X register value is added for target address calculation

i=1, n=0 Immediate addressing, **TA**: TA is used as the operand value, no memory reference

i=0, n=1 Indirect addressing, **((TA))**: The word at the TA is fetched. Value of TA is taken as the address of the operand value

i=0, n=0 or i=1, n=1 Simple addressing, **(TA)**:TA is taken as the address of the operand value

Two new relative addressing modes are available for use with instructions assembled using format 3.

Mode	**Indication**	**Target address calculation**
Base relative	b=1,p=0	TA=(B)+ disp (0≤disp ≤4095)

Program-counter relative	b=0,p=1	TA=(PC)+ disp (-2048≤disp ≤2047)

1c) Give simple I/O operation of SIC/XE

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

Simple data and character movement operation (SIC/XE)

```
        LDA     #5
        STA     ALPHA
        LDA     #90
        STCH    C1
          .
          .
ALPHA   RESW    1
C1      RESB    1
```

DEC 07/ JAN 08

1a) What are the addressing modes of sic/xe?

Five possible addressing modes plus the combinations are as follows.

1. **Direct** (x, b, and p all set to 0): operand address goes as it is. n and i are both set to the same value, either 0 or 1. While in general that value is 1, if set to 0 for

format 3 we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to make the format compatible to the SIC format.

2. **Relative** (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1)

3. **Immediate**(i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction)

4. **Indirect**(i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

5. **Indexed** (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

   The various flag bits used in the above formats have the following meanings

e - > e = 0 means format 3, e = 1 means format 4

Bits x,b,p : Used to calculate the target address using relative, direct, and indexed addressing Modes.

Bits i and n: Says, how to use the target address

b and p - both set to 0, disp field from format 3 instruction is taken to be the target address. For a format 4 bits b and p are normally set to 0, 20 bit address is the target address

x - x is set to 1, X register value is added for target address calculation

i=1, n=0 Immediate addressing, **TA**: TA is used as the operand value, no memory reference

i=0, n=1 Indirect addressing, **((TA))**: The word at the TA is fetched. Value of TA is taken as the address of the operand value

i=0, n=0 or i=1, n=1 Simple addressing, **(TA)**:TA is taken as the address of the operand value

Two new relative addressing modes are available for use with instructions assembled using format 3.

Mode	**Indication**	**Target address calculation**
Base relative	b=1,p=0	TA=(B)+ disp (0≤disp ≤4095)
Program-counter relative	b=0,p=1	TA=(PC)+ disp (-2048≤disp ≤2047)

UNIT 2: ASSEMBLER-1

DECEMBER 2010

2a)What are the different assembler Directives? Give out the assembler functions

(DEC 09/JAN 10)

Basic assembler directives

a) START
b) END
c) BYTE
d) WORD
e) RESB
f) RESW

The assembler functions are:

1) Convert mnemonic operation codes to their machine language equivalents
2) Convert symbolic operands to their equivalent machine addresses .
3) Build the machine instructions in the proper format
4) Convert the data constants to internal machine representations
5) Write the object program and the assembly listing

A SIMPLE SIC ASSEMBLER FUNCTION :

The translation of source program to object code requires following actions :

- Convert mnemonic codes to their machine language, ex: STL to 14.
- Convert data constants to their internal machine representations, ex: EOF to 454F46.
- Convert the symbolic operands to machine addresses-ex: RETADR to 1033.
- Build the machine instructions in proper format.
- Write object program and assembly listing.

Pseudo-Instructions

g) Not translated into machine instructions

Providing information to the assembler

2b) What is need of pass 2 assembler. Give out the pass 2 algorithm.

Begin

read 1st input line

```
if OPCODE = 'START' then
 begin
   write listing line
    read next input line
  end
write Header record to object program
initialize 1st Text record
while OPCODE != 'END' do
 begin
   if this is not comment line then
    begin
      search OPTAB for OPCODE
       if found then
         begin
           if there is a symbol in OPERAND field then
              begin
                 search SYMTAB for OPERAND field then
                 if found then
                 begin
                        store symbol value as operand address
                 else
                   begin
                        store 0 as operand address
                        set error flag (undefined symbol)
                 end
```

```
                end (if symbol)
            else store 0 as operand address
                assemble the object code instruction
            else if OPCODE = 'BYTE' or 'WORD" then
                convert constant to object code
            if object code doesn't fit into current Text record then
                begin
                    Write text record to object code
                    initialize new Text record

                end
                add object code to Text record
        end {if not comment}
```

2c)Write the Pass 1 algorithm of two pass assembler.(DEC 08/JAN 09)

PASS 1:

```
Begin
 read first input line
  if OPCODE = 'START' then begin
    save #[Operand] as starting addr
    initialize LOCCTR to starting address
     write line to intermediate file
      read next line
    end( if START)
   else
```

```
        initialize LOCCTR to 0
    While OPCODE != 'END' do
begin
  if this is not a comment line then
    beg
      if there is a symbol in the LABEL field then
        begin
          search SYMTAB for LABEL
          if found then
            set error flag (duplicate symbol)
          else
            (if symbol)
        search OPTAB for OPCODE
        if found then
          add 3 (instr length) to LOCCTR
        else if OPCODE = 'WORD' then
          add 3 to LOCCTR
else if OPCODE = 'RESW' then
        add 3 * #[OPERAND] to LOCCTR
  else if OPCODE = 'RESB' then
        add #[OPERAND] to LOCCTR
   else if OPCODE = 'BYTE' then
        begin
          find length of constant in bytes
          add length to LOCCTR
```

```
        end
    else
        set error flag (invalid operation code)
  end (if not a comment)
 write line to intermediate file
    read next input line
   end { while not END}
  write last line to intermediate file
 Save (LOCCTR – starting address) as program length
End {pass 1}
```

MAY /JUNE 2010

2a) Write a note on (i) OPTAB (ii) SYMTAB

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.

- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.

- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides

fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).

- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.

- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.

- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

- Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2.

- A copy of the source program is also an input to the pass 2, which is used to retain the operations that may be performed during pass 1 (such as scanning the operation field for symbols and addressing flags), so that these need not be performed during pass 2. Similarly, pointers into OPTAB and SYMTAB is retained for each operation code and symbol used. This avoids need to repeat many of the table-searching operations.

2b) Generate the complete object program.

The object code generated are

045788h

005788h

180015h

2C5785h

984006h

00578Bh

4C0000h

DEC 09/JAN 10

2b) What are the data structures used in Assemblers?

1) **OPTAB**

- Mnemonic operation codes Machine code
- Contain instruction format and length
  - LOCCTR LOCCTR +(instruction length)
- Implementation
  - It is a static table
  - Array or hash table
    Usually use a hash table (mnemonic opcode as key)

2)SYMTAB:

- Label name label address, type, length, flag
  - To indicate error conditions (Ex: multiple define)
- It is a dynamic table
  - Insert, delete and search
  - Usually use a hash table

– The hash function should perform non-random
key (Ex: LOOP1, LOOP2, X, Y, Z)

- The major issue is for insert & search but not for delete.

3)LOCCTR:

- Initialize to be the beginning address specified in the "START" statement
- LOCCTR LOCCTR + (instruction length)
- The current value of LOCCTR gives the address to the label encountered

2c)What is program relocation ? Why is it required?

- Why Relocation
  - It is desirable to load and run several programs at the same time
  - The system must be able to load programs into memory wherever there is space
  - The exact starting address of the program is not known until load time
- Absolute Program
  - Program with starting address specified at assembly time
  - The address may be invalid if the program is loaded into somewhere else.
- Example:

```
55      101B    LDA    THREE        00102D
```

  - **Reload the program starting at 3000**

```
55      101B    LDA   THREE        00302D
```

- The only parts of the program that require modification at load time are those that specify direct addresses
- The rest of the instructions need not be modified
  - Not a memory address (immediate addressing)
  - PC-relative, Base-relative
- From the object program, it is not possible to

Distinguish the address and constant

  - The assembler must keep some information to tell the loader about this relocated address.
  - The object program that contains the modification record is called a relocatable program

Solution

- For an address label, its address is assigned relative to the start of the program (START 0)
- Produce a Modification record to store the starting location and the length of the address field to be modified.
- The command for the loader must also be a part of the object program

MODIFICATION RECORD

- One modification record for each address to be modified The length is stored in half-bytes (4 bits)
- The starting location is the location of the byte containing the leftmost bits of the address field to be modified.
- If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.
- Modification Record
  - Col. 1 M
  - Col. 2-7 Starting location of the address field to be modified,

relative to the beginning of the program.

. Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

DEC 08/JAN 09

2b) What are the different records required to obtain assembler object code?

Header

Col. 1 H

Col. 2~7 Program name

Col. 8~13 Starting address (hex)

Col. 14-19 Length of object program in bytes (hex)

Text

Col.1 T

Col.2~7 Starting address in this record (hex)

Col. 8~9 Length of object code in this record in bytes (hex)

Col. 10~69 Object code (69-10+1)/6=10 instructions

End

Col.1 E

Col.2~7 Address of first executable instruction (hex)

(END program_name)

DEC 07/ JAN 08

2a) Explain the data structures in pass two of two pass assembler.

1) **OPTAB**

- Mnemonic operation codes Machine code
- Contain instruction format and length
  - LOCCTR LOCCTR +(instruction length)
- Implementation
  - It is a static table
  - Array or hash table

    Usually use a hash table (mnemonic opcode as key)

2)SYMTAB:

- Label name label address, type, length, flag
  - To indicate error conditions (Ex: multiple define)
- It is a dynamic table
  - Insert, delete and search
  - Usually use a hash table

– The hash function should perform non-random key (Ex: LOOP1, LOOP2, X, Y, Z)

- The major issue is for insert & search but not for delete.

3)LOCCTR:

- Initialize to be the beginning address specified in the "START" statement
- LOCCTR LOCCTR + (instruction length)
- The current value of LOCCTR gives the address to the label encountered

2b) Give out the Head record, text and modification record

Header record:

Col 1	H
Col. 2-7	Program name
Col 8-13	Starting address of object program (hexadecimal)
Col 14-19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T
Col 2-7.	Starting address for object code in this record (hexadecimal)
Col 8-9	Length off object code in this record in bytes (hexadecimal)

Col 10-69 Object code, represented in hexadecimal (2 columns per byte of object code)

Modification record

Col. 1 M

Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program (Hex)

Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

2c) Explain the usage of LTORG with an example

LITERAL POOLS

- Normally literals are placed into a pool at the end of the program (after the END statement)
  - In some cases, it is desirable to place literals into a pool at some other location in the object program
- Assembler directive LTORG
  - When the assembler encounters a LTORG statement, it generates a *literal pool* (containing all literal operands used since previous LTORG)
- Reason: keep the literal operand close to the instruction Otherwise PC- relative addressing may not be allowed
- LITTAB
  The contents are
    - Literal name
    - Operand value and length
    - Address
- LITTAB is often organized as a hash table, using the literal name or value as the key

Pass 1

- Build LITTAB with literal name, operand value and length, leaving the address unassigned
- When LTORG or END statement is encountered, assign an address to each literal not yet assigned an address

- The location counter is updated to reflect the number of bytes occupied by each literal

Pass 2

- Search LITTAB for each literal operand encountered
- Generate data values using BYTE or WORD statements
- Generate Modification record for literals that represent an address in the program

UNIT 3: ASSEMBLER-2

DECEMBER 2010

3a)Define Literal and immediate operand.

A literal is identified with the prefix =, followed by a specification of the literal value

- Examples:
  LDA = C'EOF'

- Literals
  - The assembler generates the specified value as a constant at some other memory location
- Immediate Operands
  - The operand value is assembled as part of the machine instruction

55 0020 LDA #3 010003

3b) What are Control sections and program blocks?

A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references.*

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive

– assembler directive: **CSECT**

The syntax

sec name CSECT

– separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

Assembler Directive USE:

USE [blockname]

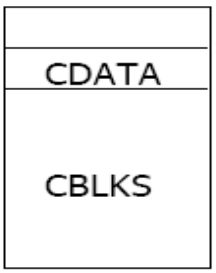
At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory



Arranging code into program blocks:

Pass 1

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

3c) Explain Multipass assembler with algorithm.

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.

- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.
    - One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
    - Load-and-go assembler generates their object code in memory for immediate execution.
    - No object program is written out, no loader is needed.
    - It is useful in a system with frequent program development and testing
    - The efficiency of the assembly process is an important consideration.
    - Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

MAY /JUNE 2010

3a) Literal operands

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

```
45    001A  ENDFIL    LDA   =C'EOF'     032010
-
-
93                    LTORG
      002D  *                 =C'EOF'     454F46
```

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value '05'.

```
215   1062  WLOOP     TD    =X'05'      E32011
```

It is important to understand the difference between a constant defined as a literal and a constant defined as an immediate operand. In case of literals the assembler generates the specified value as a constant at some other memory location In immediate mode the operand value is assembled as part of the instruction itself. Example

55 0020 LDA #03 010003

All the literal operands used in a program are gathered together into one or more *literal pool*s. This is usually placed at the end of the program. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

A literal table is created for the literals which are used in the program. The literal table contains the *literal name, operand value and length.* The literal table is usually created as a hash table on the literal name.

Implementation of Literals:

During Pass-1:

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG for literal definition. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

During Pass-2:

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or

WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITTAB

3b) What are the formats for DEFINE and REFER records?

Define record (EXTDEF)

- Col. 1 D
- Col. 2-7 Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

Refer record (EXTREF)

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

3c)Explain simple Load and go assembler with an example

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
  - The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

- Scenario for one-pass assembler:
  - Load-and-Go: generates the object for immediate execution
  - External storage for intermediate file between two passes becomes slow.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110					

DEC 09/JAN 10

3a) Define literals

- A literal is identified with the prefix =, followed by a specification of the literal value
- Examples:
  LDA = C'EOF'
- Literals
  - The assembler generates the specified value as a constant at some other memory location
- Immediate Operands
  - The operand value is assembled as part of the machine instruction

55 0020 LDA #3 010003

3b)Explain load and go assembler.

Load-and-go assembler generates their object code in memory for immediate execution.

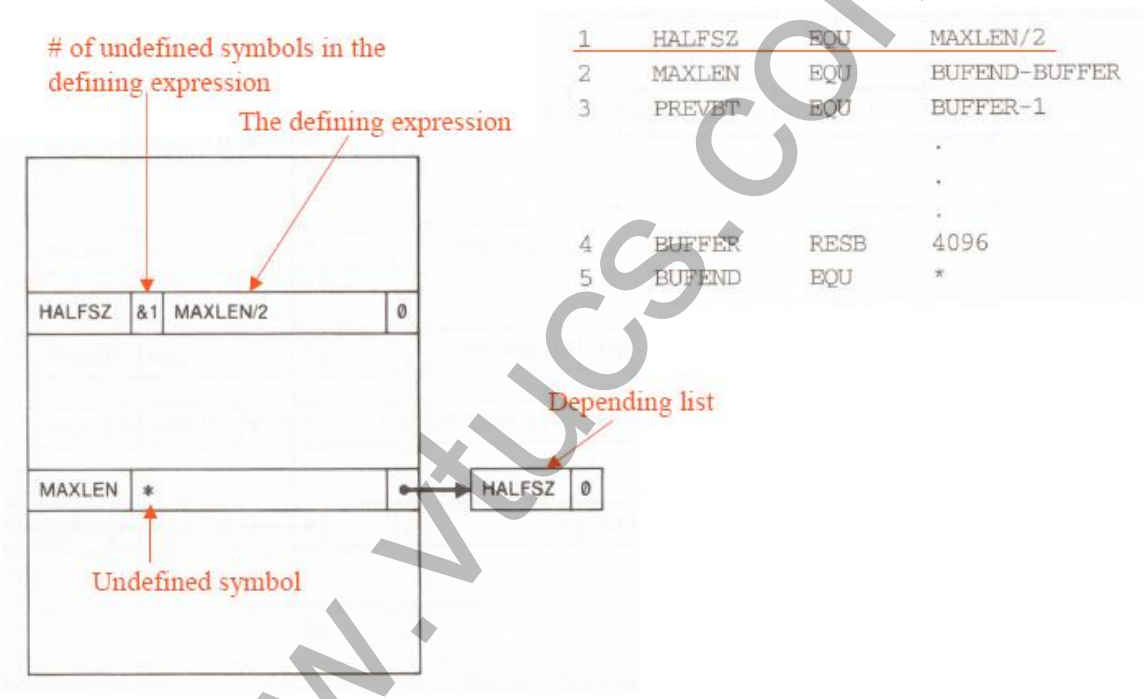
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
  - The efficiency of the assembly process is an important consideration.

- Programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration.

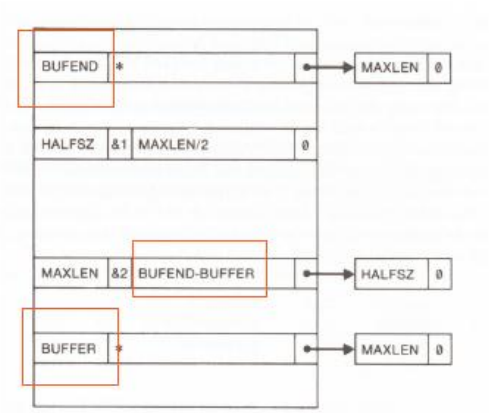
- Scenario for one-pass assembler:
  - Load-and-Go: generates the object for immediate execution

  - External storage for intermediate file between two passes becomes slow.

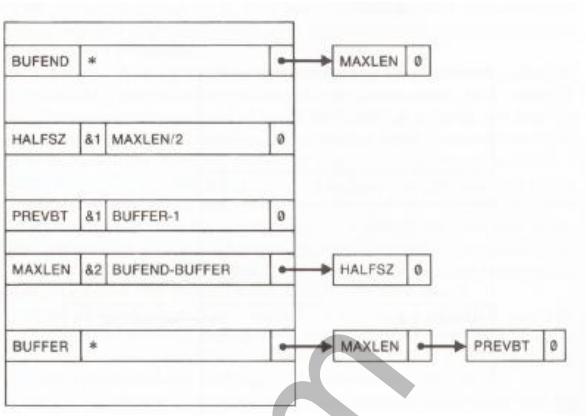
3c)Explain Multipass assembler for the given problem below from line 1 to 5.



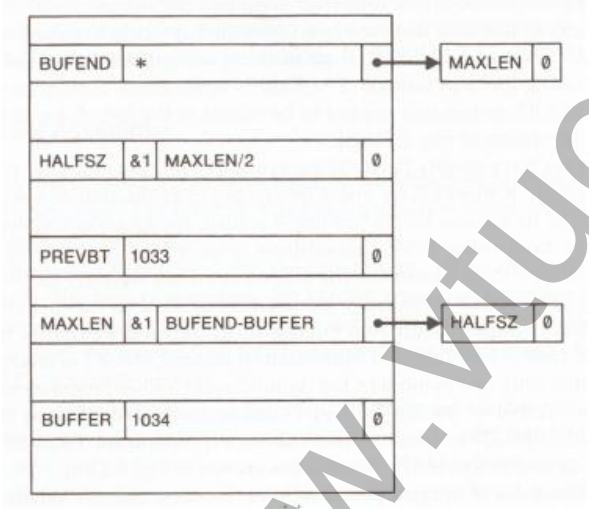
Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:



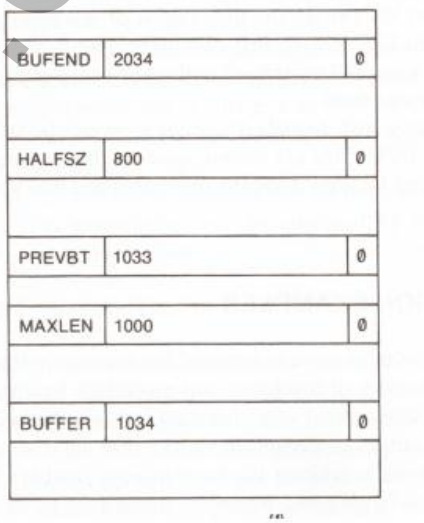
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

DEC 08/JAN 09

3a) (i) multipass assembler (ii) MASM assembler

- For a two pass assembler, forward references in symbol definition are not allowed:

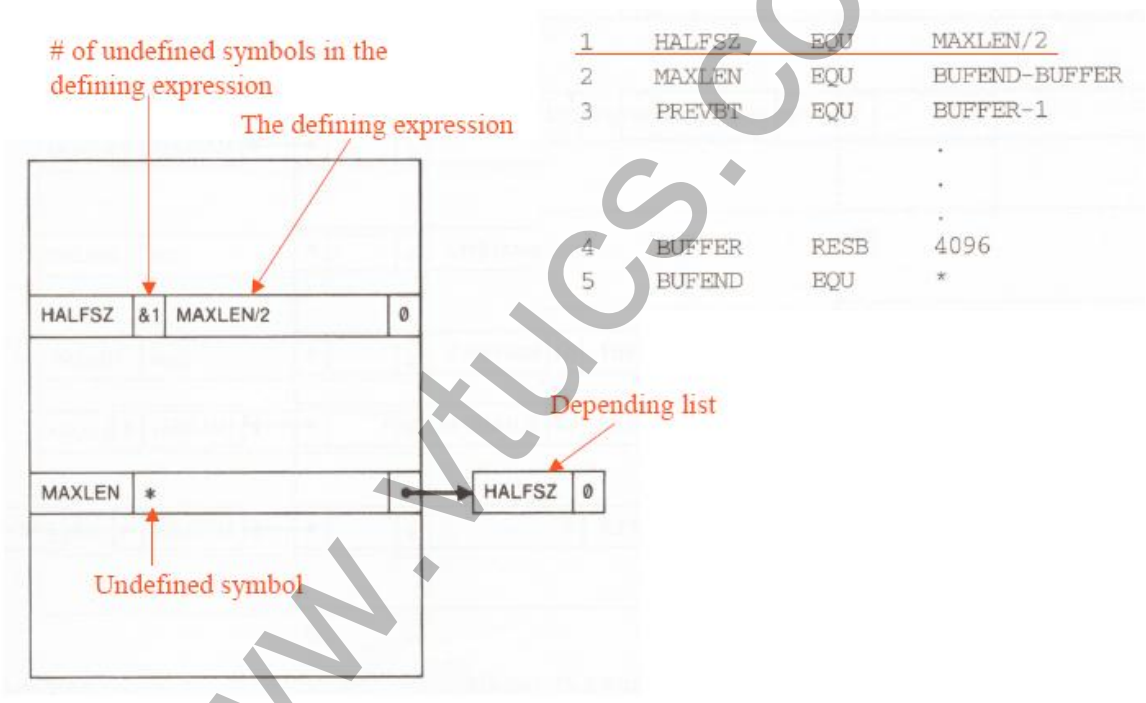
```
ALPHA    EQU  BETA

BETA     EQU  DELTA
```

DELTA RESW 1

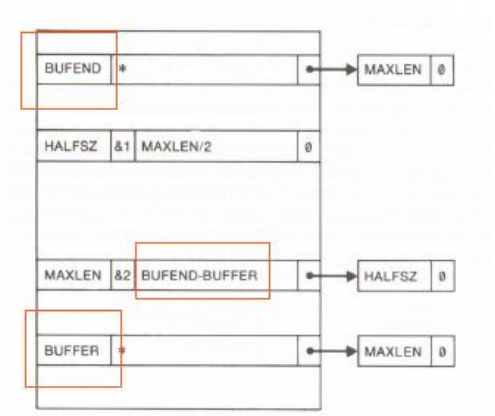
- Symbol definition must be completed in pass 1.

- Prohibiting forward references in symbol definition is not a serious inconvenience.
  - Forward references tend to create difficulty for a person reading the program.

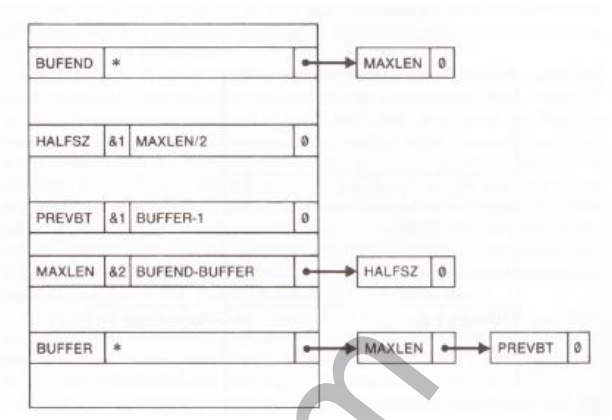
Multi-Pass Assembler Example Program



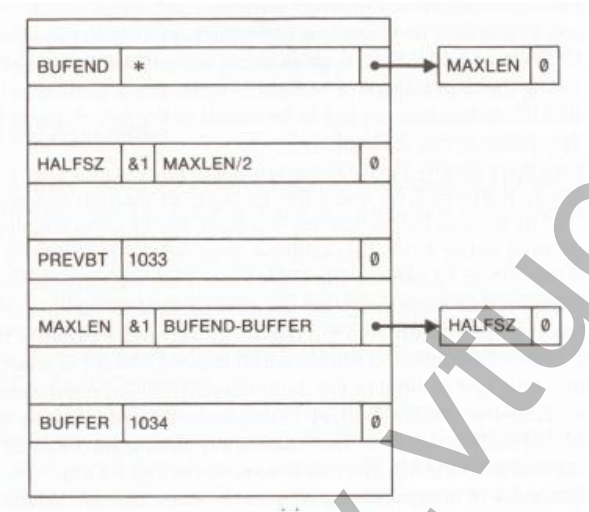
Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:



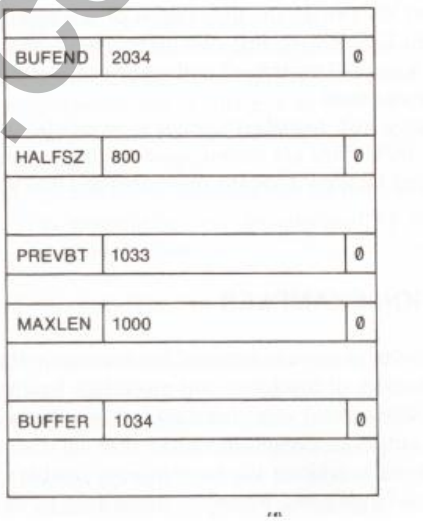
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

(ii) MASM Assembler

The **Microsoft Macro Assembler** is an X86 architecture assembler for MS-DOS and Microsoft Windows. While the name MASM has earlier usage as the Unisys OS 1100 Meta-Assembler, it is commonly understood in more recent years to refer to the Microsoft Macro Assembler. It is an archetypal MACRO assembler for the x86 PC market that is owned and maintained by a major operating system vendor and since the introduction of MASM version 6.0 in 1991 has had a powerful preprocessor that supports pseudo high level emulation of variety of high level constructions including loop code, conditional testing and has a semi-automated system of procedure creation and management available if required. Version 6.11d was 32 bit object module capable using a specialised linker available in the WinNT 3.5 SDK but with the introduction of binary patches that upgraded version 6.11d, all later versions were 32 bit Portable Executable

console mode application that produced both OMF and COFF object modules for 32 bit code.

3b) Pass 1 and Pass 2 of linking loader

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'D' then
                    for each symbol in the record do
                        begin
                            search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while <> 'E'}
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin
        read next input record   {Header record}
        set CSLTH to control section length
        while record type <> 'E' do
            begin
                read next input record
                if record type = 'T' then
                    begin
                        {if object code is in character form, convert
                            into internal representation}
                        move object code from record to location
                            (CSADDR + specified address)
                    end {if 'T'}
                else if record type = 'M' then
                    begin
                        search ESTAB for modifying symbol name
                        if found then
                            add or subtract symbol value at location
                                (CSADDR + specified address)
                        else
                            set error flag (undefined external symbol)
                    end  {if 'M'}
            end  {while <> 'E'}
        if an address is specified (in End record) then
            set EXECADDR to (CSADDR + specified address)
        add CSLTH to CSADDR
    end  {while not EOF}
jump to location given by EXECADDR (to start execution of loaded program)
end  {Pass 2}
```

DEC 07/ JAN 08

3a)Give out the usage of Program blocks

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

Assembler Directive USE:

USE [blockname]

At the beginning, statements are assumed to be part of the *unnamed* (default) block. If no USE statements are included, the entire program belongs to this single block. Each

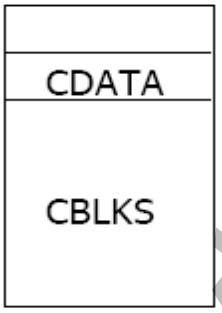
program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. *Program readability is better* if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

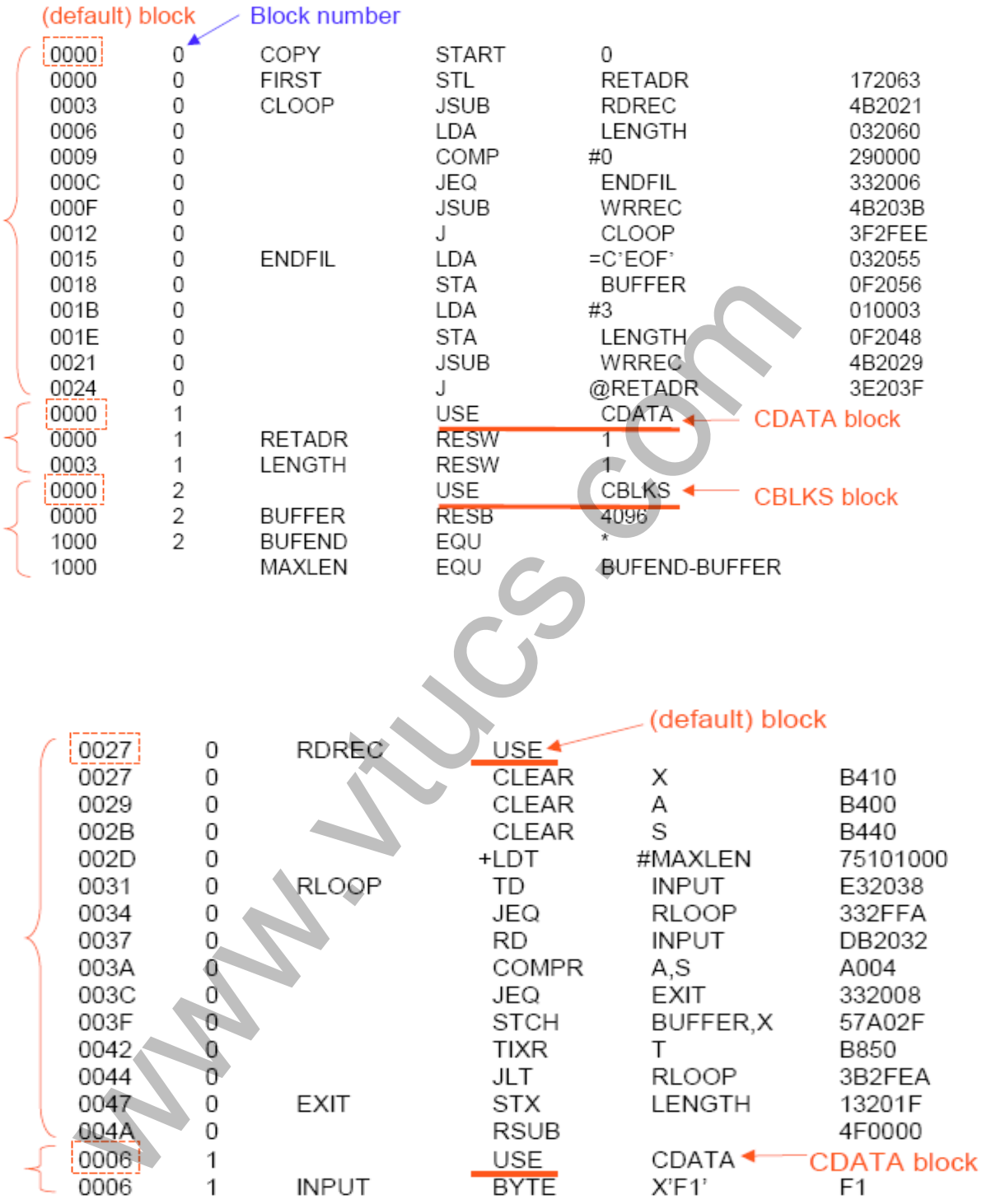
Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory



Example Code



0000	0	COPY	START	0	
0000	0	FIRST	STL	RETADR	172063
0003	0	CLOOP	JSUB	RDREC	4B2021
0006	0		LDA	LENGTH	032060
0009	0		COMP	#0	290000
000C	0		JEQ	ENDFIL	332006
000F	0		JSUB	WRREC	4B203B
0012	0		J	CLOOP	3F2FEE
0015	0	ENDFIL	LDA	=C'EOF'	032055
0018	0		STA	BUFFER	0F2056
001B	0		LDA	#3	010003
001E	0		STA	LENGTH	0F2048
0021	0		JSUB	WRREC	4B2029
0024	0		J	@RETADR	3E203F
0000	1		USE	CDATA	
0000	1	RETADR	RESW	1	
0003	1	LENGTH	RESW	1	
0000	2		USE	CBLKS	
0000	2	BUFFER	RESB	4096	
1000	2	BUFEND	EQU	*	
1000		MAXLEN	EQU	BUFEND-BUFFER	

0027	0	RDREC	USE		
0027	0		CLEAR	X	B410
0029	0		CLEAR	A	B400
002B	0		CLEAR	S	B440
002D	0		+LDT	#MAXLEN	75101000
0031	0	RLOOP	TD	INPUT	E32038
0034	0		JEQ	RLOOP	332FFA
0037	0		RD	INPUT	DB2032
003A	0		COMPR	A,S	A004
003C	0		JEQ	EXIT	332008
003F	0		STCH	BUFFER,X	57A02F
0042	0		TIXR	T	B850
0044	0		JLT	RLOOP	3B2FEA
0047	0	EXIT	STX	LENGTH	13201F
004A	0		RSUB		4F0000
0006	1		USE	CDATA	
0006	1	INPUT	BYTE	X'F1'	F1

Arranging code into program blocks:

Pass 1

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

3b) How Forward references are handled in one pass assembler?

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.
  - One that produces object code directly in memory for immediate execution (Load-and-go assemblers).
  - Load-and-go assembler generates their object code in memory for immediate execution.
  - No object program is written out, no loader is needed.

- It is useful in a system with frequent program development and testing
- The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

Forward Reference in One-Pass Assemblers:

In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler
    - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

DECEMBER 2010

4a)Explain how Relocating loaders are used using modification records?

Methods for specifying relocation:

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record

col 1: M

col 2-7: relocation address

col 8-9: length (halfbyte)

col 10: flag (+/-)

col 11-17: segment name

H^COPY ^000000 001077

T^000000 ^1D^17202D^69202D^48101036^…^4B105D^3F2FEC^032010

T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46

T^001035 ^1D^B410^B400^B440^75101000^…^332008^57C003^B850

T^001053^1D^3B2FEA^134000^4F0000^F1^..^53C003^DF2008^B850

T^00070^07^3B2FEF^4F0000^05

M^000007^05+COPY

M^000014^05+COPY

M^000027^05+COPY

E^000000

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 1: modification is needed. This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

4b)Dynamic linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call.

The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

MAY /JUNE 2010

4a) Explain Relocation and Linking operation.

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 4.1. Translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 4.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure4.3 shows the role of both loader and linker.

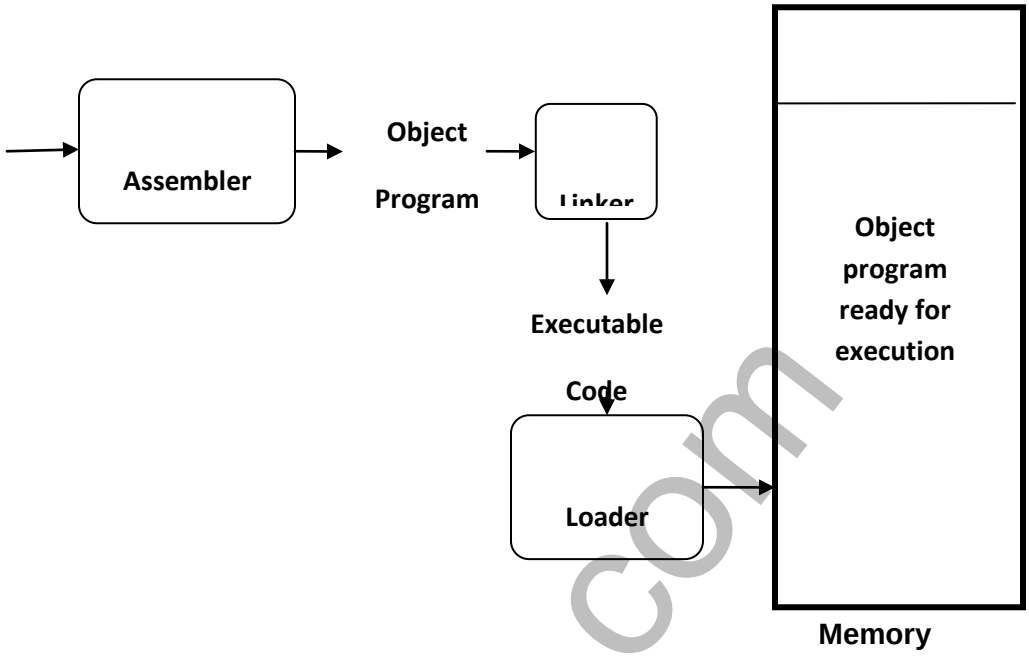
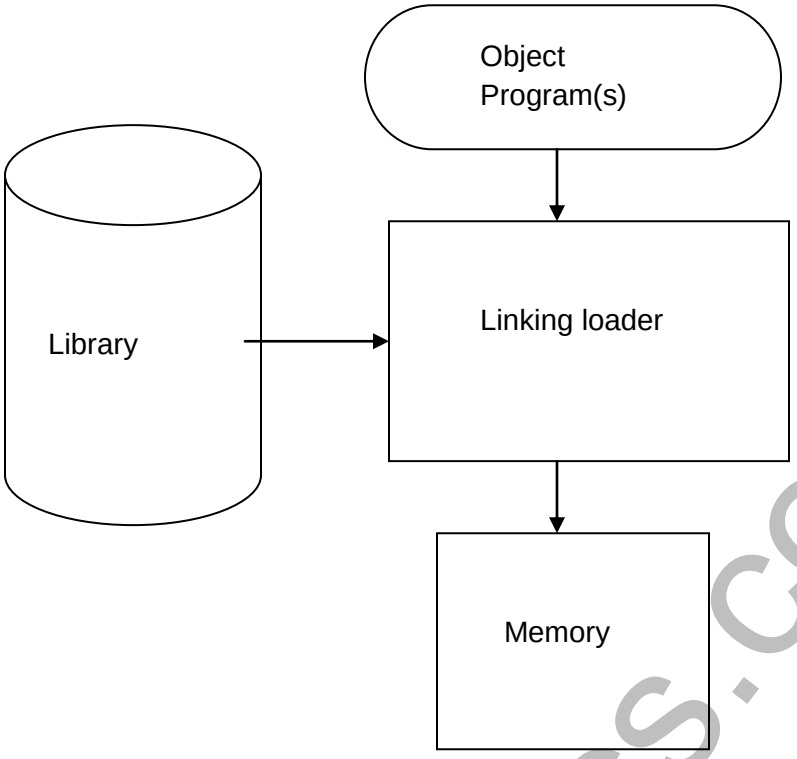


Figure : The Role of both Loader and Linker

4b) Different Loader options

There are some common alternatives for organizing the loading functions, including relocation and linking. Linking Loaders – Perform all linking and relocation at load time. The Other Alternatives are Linkage editors, which perform linking prior to load time and, dynamic linking, in which linking function is performed at execution time

Linking Loaders



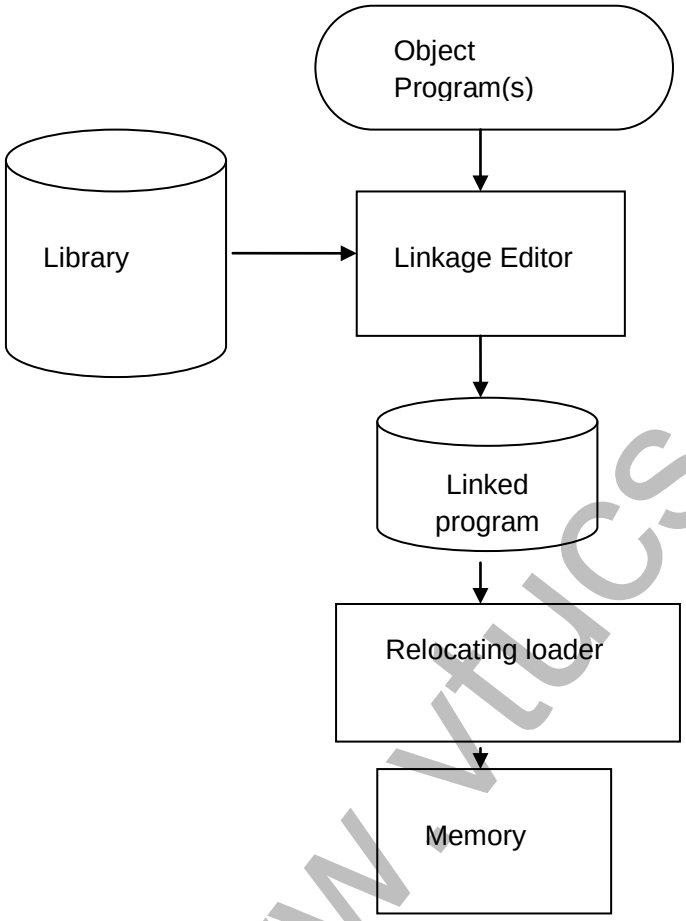
The above diagram shows the processing of an object program using Linking Loader. The source program is first assembled or compiled, producing an object program. A linking loader performs all linking and loading operations, and loads the program into memory for execution.

Linkage Editors

The figure below shows the processing of an object program using Linkage editor. A linkage editor produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution. The linked program produced is generally in a form that is suitable for processing by a relocating loader.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages

of subroutines or other control sections that are generally used together. Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space



Dynamic Linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several

programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

Bootstrap Loaders

If the question, how is the loader itself loaded into the memory? is asked, then the answer is, when computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution. The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

4c) (i) Linking Loader (ii) Dynamic Linking

Dynamic Linking

The scheme that postpones the linking functions until execution. A subroutine is loaded and linked to the rest of the program when it is first called – usually called dynamic linking, dynamic loading or load on call. The advantages of dynamic linking are, it allow several executing programs to share one copy of a subroutine or library. In an object oriented system, dynamic linking makes it possible for one object to be shared by several programs. Dynamic linking provides the ability to load the routines only when (and if) they are needed. The actual loading and linking can be accomplished using operating system service request.

Linking Loader uses two-passes logic. ESTAB (external symbol table) is the main data structure for a linking loader.

Pass 1: Assign addresses to all external symbols

Pass 2: Perform the actual loading, relocation, and linking

ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of

the control section, the symbol appearing in the control section, its address and length of the control section.

Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables & Data structures used during pass 1 are, PROGADDR (program load address) from OS, CSADDR (control section address), CSLTH (control section length) and ESTAB. The pass 1 processes the Define Record.

Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its address. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs.

4a) **bootstrap loader**

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin

X=0x80 (the address of the next memory location to be loaded

Loop

A←GETC (and convert it from the ASCII character

code to the value of the hexadecimal digit)

save the value in the high-order 4 bits of S

A←GETC

combine the value to form one byte A← (A+S)

store the value (in A) to the address in register X

X←X+1

End

It uses a subroutine GETC, which is

GETC A←read one character

if A=0x04 then jump to 0x80

if A<48 then GETC

A ← A-48 (0x30)

if A<10 then return

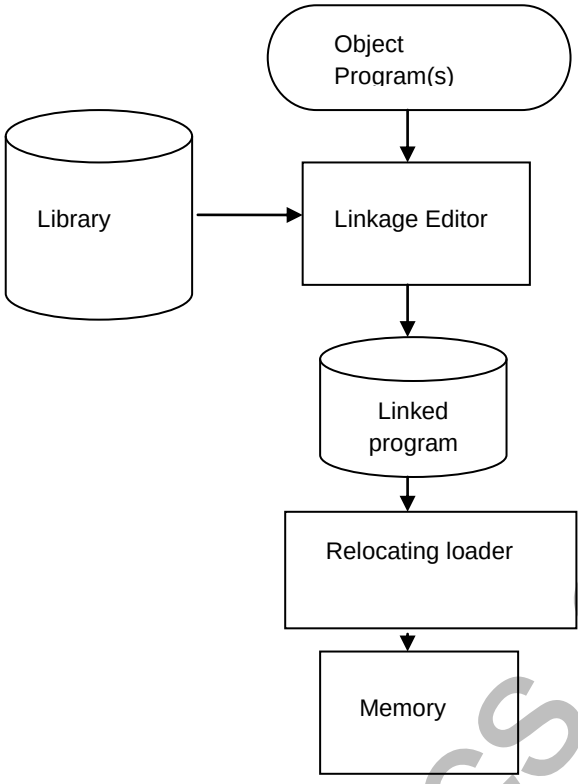
A ← A-7

return

4b) **how object program can be processed using linkage editor?**

The figure below shows the processing of an object program using Linkage editor. A linkage editor produces a linked version of the program – often called a load module or an executable image – which is written to a file or library for later execution. The linked program produced is generally in a form that is suitable for processing by a relocating loader.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space



DEC 08/JAN 09

4a) Explain Bootstrap Loader with its algorithm.

If the question, how is the loader itself loaded into the memory? is asked, then the answer is, when computer is started – with no program in memory, a program present in ROM (absolute address) can be made executed – may be OS itself or A Bootstrap loader, which in turn loads OS and prepares it for execution. The first record (or records) is generally referred to as a bootstrap loader – makes the OS to be loaded. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin

X=0x80 (the address of the next memory location to be loaded

Loop

A←GETC (and convert it from the ASCII character

code to the value of the hexadecimal digit)

save the value in the high-order 4 bits of S

A←GETC

combine the value to form one byte A← (A+S)

store the value (in A) to the address in register X

X←X+1

End

It uses a subroutine GETC, which is

GETC A←read one character

if A=0x04 then jump to 0x80

if A<48 then GETC

A ← A-48 (0x30)

if A<10 then return

A ← A-7

return

DEC 07/ JAN 08

4a)Define Loader.Give out the algorithm for bootatrap loader.

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution.

A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

Begin

X=0x80 (the address of the next memory location to be loaded

Loop

A←GETC (and convert it from the ASCII character

code to the value of the hexadecimal digit)

save the value in the high-order 4 bits of S

A←GETC

combine the value to form one byte A← (A+S)

store the value (in A) to the address in register X

X←X+1

End

It uses a subroutine GETC, which is

GETC A←read one character

if A=0x04 then jump to 0x80

if A<48 then GETC

A ← A-48 (0x30)

if A<10 then return

A ← A-7

return

4b) Explain Relocation

Use of modification record and, use of relocation bit, are the methods available for specifying relocation. In the case of modification record, a modification record M is used in the object program to specify any relocation. In the case of use of relocation bit, each instruction is associated with one relocation bit and, these relocation bits in a Text record is gathered into bit masks.

Modification records are used in complex machines and is also called Relocation and Linkage Directory (RLD) specification. The format of the modification record (M) is as follows. The object program with relocation by Modification records is also shown here.

Modification record

col 1: M

col 2-7: relocation address

col 8-9: length (halfbyte)

col 10: flag (+/-)

col 11-17: segment name

HΛCOPY Λ000000 001077

TΛ000000 Λ1DΛ17202DΛ69202DΛ48101036Λ…Λ4B105DΛ3F2FECΛ032010

TΛ00001DΛ13Λ0F2016Λ010003Λ0F200DΛ4B10105DΛ3E2003Λ454F46

TΛ001035 Λ1DΛB410ΛB400ΛB440Λ75101000Λ…Λ332008Λ57C003ΛB850

TΛ001053Λ1DΛ3B2FEAΛ134000Λ4F0000ΛF1Λ..Λ53C003ΛDF2008ΛB850

TΛ00070Λ07Λ3B2FEFΛ4F0000Λ05

MΛ000007Λ05+COPY

MΛ000014Λ05+COPY

MΛ000027Λ05+COPY

EΛ000000

The relocation bit method is used for simple machines. Relocation bit is 0: no modification is necessary, and is 1: modification is needed. This is specified in the columns 10-12 of text record (T), the format of text record, along with relocation bits is as follows.

Text record:

col 1: T

col 2-7: starting address

col 8-9: length (byte)

col 10-12: relocation bits

col 13-72: object code

Twelve-bit mask is used in each Text record (col:10-12 – relocation bits), since each text record contains less than 12 words, unused words are set to 0, and, any value that is to be modified during relocation must coincide with one of these 3-byte segments. For absolute loader, there are no relocation bits column 10-69 contains object code. The object program with relocation by bit mask is as shown below. Observe FFC - means all ten words are to be modified and, E00 - means first three records are to be modified.

H^COPY ^000000 00107A

T^000000^1E^FFC^140033^481039^000036^280030^300015^…^3C0003 ^ …

T^00001E^15^E00^0C0036^481061^080033^4C0000^…^000003^000000

T^001039^1E^FFC^040030^000030^…^30103F^D8105D^280030^...

T^001057^0A^ 800^100036^4C0000^F1^001000

4c) How Program linking is possible in loaders?

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

EXTDEF (external definition) - The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

ex: EXTDEF BUFFER, BUFFEND, LENGTH

EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC

EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record

The format of the Define record (D) along with examples is as shown here**.**

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

UNIT 5: TEXT EDITORS

DECEMBER 2010

5a) Explain document linking process in an interactive system

Most text editors have a structure similar to that shown in the following figure. That is most text editors have a structure similar to shown in the figure regardless of features and the computers

Command language Processor accepts command, uses semantic routines – performs functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions.

- Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.

- In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste paragraph etc..,.

- When editing command is issued, editing component invokes the editing filter – generates a new editing buffer – contains part of the document to be edited from current editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer being created.

- In viewing a document, the start of the area to be viewed is determined by the current viewing pointer maintained by the viewing component. Viewing component is a collection of modules responsible for determining the next view. Current viewing pointer can be set or reset as a result of previous editing operation.

5b) Give out the relationship between editing and viewing.

- Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.

- In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste paragraph etc..,.

- When editing command is issued, editing component invokes the editing filter – generates a new editing buffer – contains part of the document to be edited from current editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer being created.

- In viewing a document, the start of the area to be viewed is determined by the current viewing pointer maintained by the viewing component. Viewing component is a collection of modules responsible for determining the next view. Current viewing pointer can be set or reset as a result of previous editing operation.

- When display needs to be updated, viewing component invokes the viewing filter – generates a new viewing buffer – contains part of the document to be viewed from current viewing pointer. In case of line editors – viewing buffer may contain the current line, Screen editors - viewing buffer contains a rectangular cutout of the quarter plane of the text.

- Viewing buffer is then passed to the display component of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen –

called a window. Identical – user edits the text directly on the screen. Disjoint – Find and Replace (For example, there are 150 lines of text, user is in 100th line, decides to change all occurrences of 'text editor' with 'editor').

- The editing and viewing buffers can also be partially overlapped, or one may be completely contained in the other. Windows typically cover entire screen or a rectangular portion of it. May show different portions of the same file or portions of different file. Inter-file editing operations are possible.

5c)What are the features of interactive debugging system?

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as tracing and traceback .

- Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on…
- Traceback can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

Program-Display capabilities

A debugger should have good program-display capabilities.

- Program being debugged should be displayed completely with statement numbers.
- The program may be displayed as originally written or with macro expansion.
- Keeping track of any changes made to the programs during the debugging session. Support for symbolically displaying or modifying the contents of any of the variables and constants in the program. Resume execution – after these changes.

MAY /JUNE 2010

5a) What are the four important tasks of text editor?

Document-editing process in an interactive user-computer dialogue has four tasks:

- Select the part of the target document to be viewed and manipulated
- Determine how to format this view on-line and how to display it
- Specify and execute operations that modify the target document
- Update the view appropriately

5b) What are the three basic types of computing environment for editors?

Editors function in three basic types of computing environments:

1. Time sharing
2. Stand-alone
3. Distributed.

Each type of environment imposes some constraints on the design of an editor.

- In time sharing environment, editor must function swiftly within the context of the load on the computer's processor, memory and I/O devices.
- In stand-alone environment, editors on stand-alone system are built with all the functions to carry out editing and viewing operations – The help of the OS may also be taken to carry out some tasks like demand paging.
- In distributed environment, editor has both functions of stand-alone editor; to run independently on each user's machine and like a time sharing editor, contend for shared resources such as files.

5c)Explain traceback in debugging systems?

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging

commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as tracing and trace back.

- Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on…
- Trace back can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

5d)What are the user interface criteria in text editor?

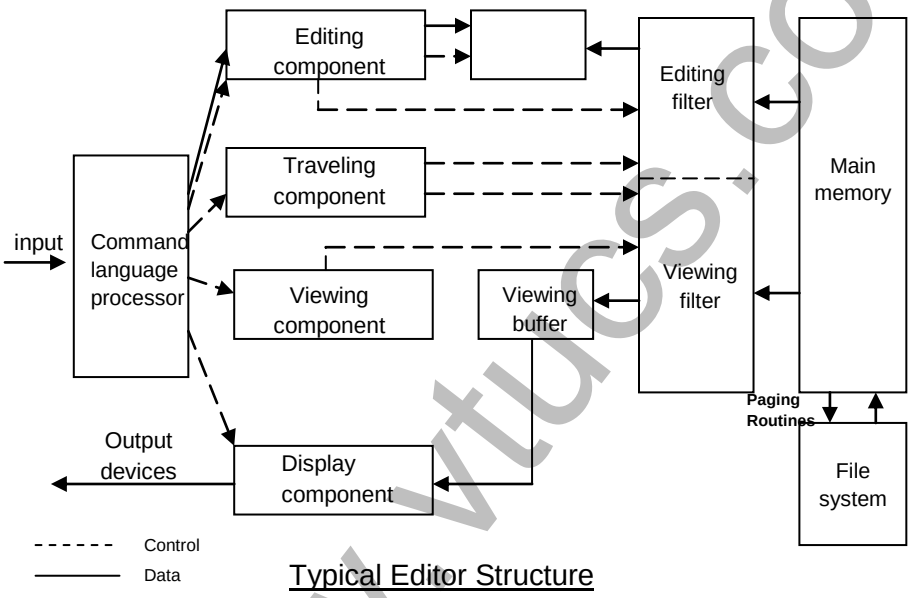
- Debugging systems should be simple in its organization and familiar in its language, closely reflect common user tasks.
- The simple organization contribute greatly to ease of training and ease of use.
- The user interaction should make use of full-screen displays and windowing-systems as much as possible.
- With menus and full-screen editors, the user has far less information to enter and remember. There should be complete functional equivalence between commands and menus – user where unable to use full-screen IDSs may use commands.
- The command language should have a clear, logical and simple syntax.
- command formats should be as flexible as possible.
- Any good IDSs should have an on-line HELP facility. HELP should be accessible from any state of the debugging session.

DEC 09/JAN 10

5a)Explain the structure of interactive editor in detail.(DEC 08/JAN 09)

Most text editors have a structure similar to that shown in the following figure. That is most text editors have a structure similar to shown in the figure regardless of features and the computers

Command language Processor accepts command, uses semantic routines – performs functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions.



Typical Editor Structure

- Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.

- In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste paragraph etc..,.

- When editing command is issued, editing component invokes the editing filter – generates a new editing buffer – contains part of the document to be edited from

current editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer being created.

- In viewing a document, the start of the area to be viewed is determined by the current viewing pointer maintained by the viewing component. Viewing component is a collection of modules responsible for determining the next view. Current viewing pointer can be set or reset as a result of previous editing operation.

5b) What are the debugging functions and capabilities? (DEC 08/JAN 09)

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as tracing and traceback .

- Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on…
- Traceback can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

Program-Display capabilities

A debugger should have good program-display capabilities.

- Program being debugged should be displayed completely with statement numbers.
- The program may be displayed as originally written or with macro expansion.
- Keeping track of any changes made to the programs during the debugging session. Support for symbolically displaying or modifying the contents of any of the variables and constants in the program. Resume execution – after these changes.

DEC 07/ JAN 08

5a)Explain Macro Definition and Expansion.

Figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction.

M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

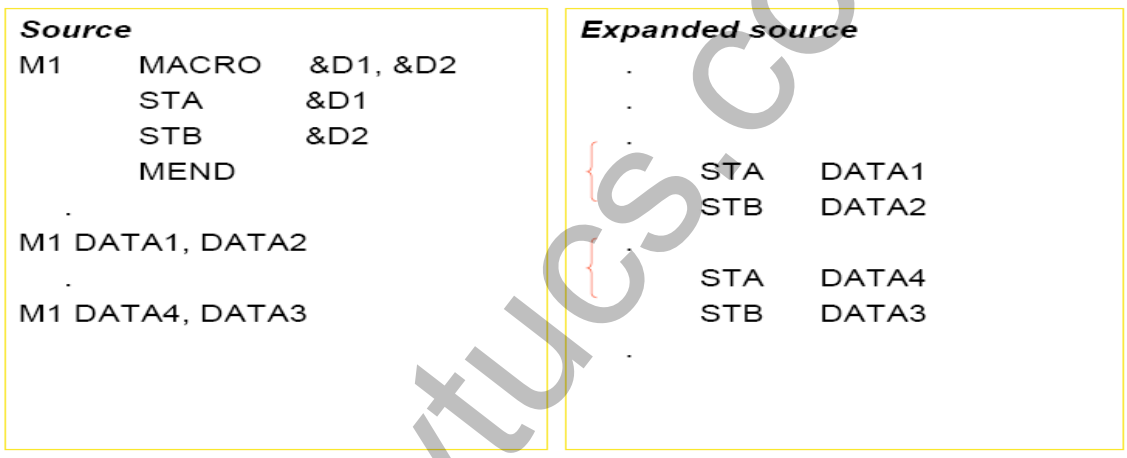


Fig 6.1: macro call

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

Macro Expansion:

The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed.

The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on.

After *macro processing* the expanded file can become the input for the *Assembler*. The *Macro Invocation* statement is considered as comments and the statement generated from expansion is treated exactly as though they had been written directly by the programmer.

he difference between *Macros* and *Subroutines* is that the statement s from the body of the Macro is expanded the number of times the macro invocation is encountered, whereas the statement of the subroutine appears only once no matter how many times the subroutine is called. Macro instructions will be written so that the body of the macro contains no labels.

- Problem of the label in the body of macro:
  - If the same macro is expanded multiple times at different places in the program …
  - There will be *duplicate labels*, which will be treated as errors by the assembler.
- Solutions:
  - Do not use labels in the body of macro.
  - Explicitly use PC-relative addressing instead.
- Ex, in RDBUFF and WRBUFF macros,
  - JEQ *+11
  - JLT *-14
- It is inconvenient and error-prone.

5b) How are Labels used in Macros?

It is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler. This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation

and expansion. During macro expansion each $ will be replaced with $XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion.

For example,

XX = AA, AB, AC…

This allows 1296 macro expansions in a single program.

The following program shows the macro definition with labels to the instruction.

```
25      RDBUFF  MACRO   &INDEV, &BUFADR, &RECLTH
30              CLEAR   X               CLEAR LOOP COUNTER
35              CLEAR   A
40              CLEAR   S
45              +LDT    #4096           SET MAXIMUM RECORD LENGTH
50      $LOOP   TD      =X'&INDEV'      TEST INPUT DEVICE
55              JEQ     $LOOP           LOOP UNTIL READY
60              RD      =X'&INDEV'      READ CHARACTER INTI REG A
65              COMPR   A, S            TEST FOR END OF RECORD
70              JEQ     $EXIT           EXIT LOOP IF EOR
75              STCH    &BUFADR, X      STORE CHARACTER IN BUFFER
80              TIXR    $LOOP           HAS BEEN REACHED
90      $EXIT   STX     &RECLTH         SAVE RECORD LENGTH
                MEND
```

UNIT 6: MACRO PROCESSORS

DECEMBER 2010

6a) what are the independent macro features? Explain any two.

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor. These features are:

- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

1. Concatenation of unique labels:

- Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as 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.).
- Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like
    - LDA X&ID1

```
TOTAL  MACRO  &ID
       LAD    X&ID1
       ADD    X&ID2
       STA    X&ID3
       MEND
```

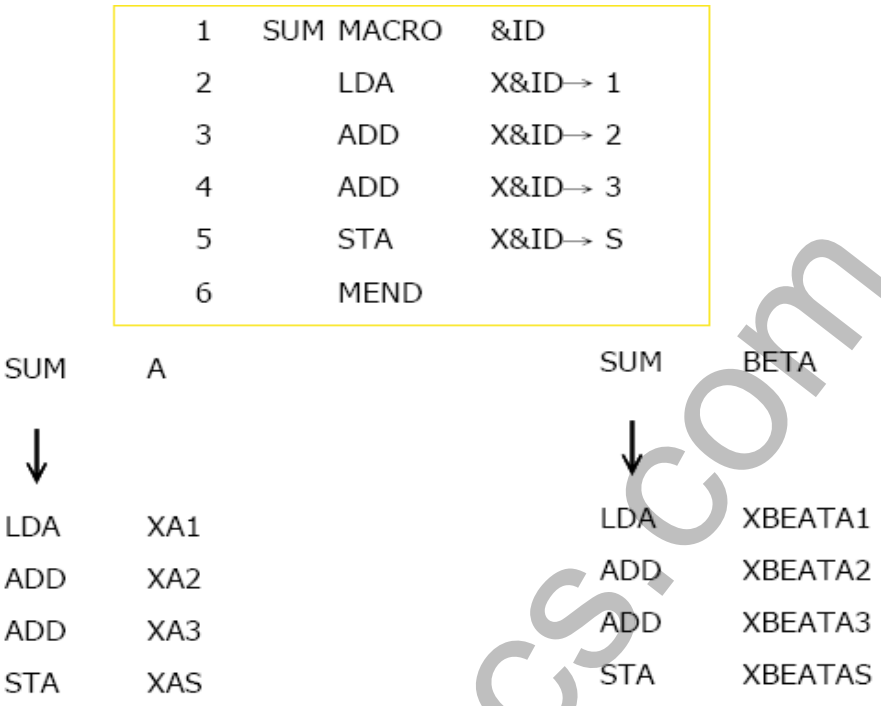
```
TOTAL  A
```

```
LAD    XA1
ADD    XA2
STA    XA3
```

& is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended.

- If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.
- Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as

  LDA X&ID→



The above figure shows a macro definition that uses the concatenation operator as previously described. The statement SUM A and SUM BETA shows the invocation statements and the corresponding macro expansion.

2. Generation of Unique Labels

- it is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler.
- This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion.
- During macro expansion each $ will be replaced with $XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion.

For example,

XX = AA, AB, AC…

This allows 1296 macro expansions in a single program.

The following program shows the macro definition with labels to the instruction.

```
25        RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
30                 CLEAR    X                    CLEAR LOOP COUNTER
35                 CLEAR    A
40                 CLEAR    S
45                +LDT      #4096                SET MAXIMUM RECORD LENGTH
50        $LOOP    TD       =X'&INDEV'           TEST INPUT DEVICE
55                 JEQ      $LOOP                LOOP UNTIL READY
60                 RD       =X'&INDEV'           READ CHARACTER INTI REG A
65                 COMPR    A, S                 TEST FOR END OF RECORD
70                 JEQ      $EXIT                EXIT LOOP IF EOR
75                 STCH     &BUFADR, X           STORE CHARACTER IN BUFFER
80                 TIXR     $LOOP                HAS BEEN REACHED
90        $EXIT    STX      &RECLTH              SAVE RECORD LENGTH
                   MEND
```

The following figure shows the macro invocation and expansion first time.

```
          .        RDBUFF   F1, BUFFER, LENGTH


30                 CLEAR    X              CLEAR LOOP COUNTER
35                 CLEAR    A
40                 CLEAR    S
45                +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 INTI 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
```

If the macro is invoked second time the labels may be expanded as $ABLOOP $ABEXIT.

6b)Explain the macro processing feature of MASM Macro processor.

We have seen an example of the *definition* of one macro instruction by another. But we have not dealt with the *invocation* of one macro by another. The following example shows the invocation of one macro by another macro.

```
10        RDBUFF   MACRO    &BUFADR, &RECLTH, &INDEV
15        .
20        .        MACRO TO READ RECORD INTO BUFFER
25        .
30                 CLEAR    X              CLEAR LOOP COUNTER
35                 CLEAR    A
40                 CLEAR    S
45                +LDT      #4096          SET MAXIMUN RECORD LENGTH
50        $LOOP    RDCHAR   &INDEV         READ CHARACTER INTO REG A
65                 COMPR    A, S           TEST FOR END OF RECORD
70                 JEQ      &EXIT          EXIT LOOP IF EOR
75                 STCH     &BUFADR, X     STORE CHARACTER IN BUFFER
80                 TIXR     T              LOOP UNLESS MAXIMUN LENGTH
85                 JLT      $LOOP          HAS BEEN REACHED
90        $EXIT    STX      &RECLTH        SAVE RECORD LENGTH
95                 MEND
```

```
5    RDCHAR      MACRO   &IN
10   .
15   .     MACROTO READ CHARACTER INTO REGISTER A
20   .
25                 TD      =X'&IN'                 TEST INPUT DEVICE
30                 JEQ       *-3                   LOOP UNTIL READY
35                 RD      =X'&IN'                 READ CHARACTER
40                 MEND
```

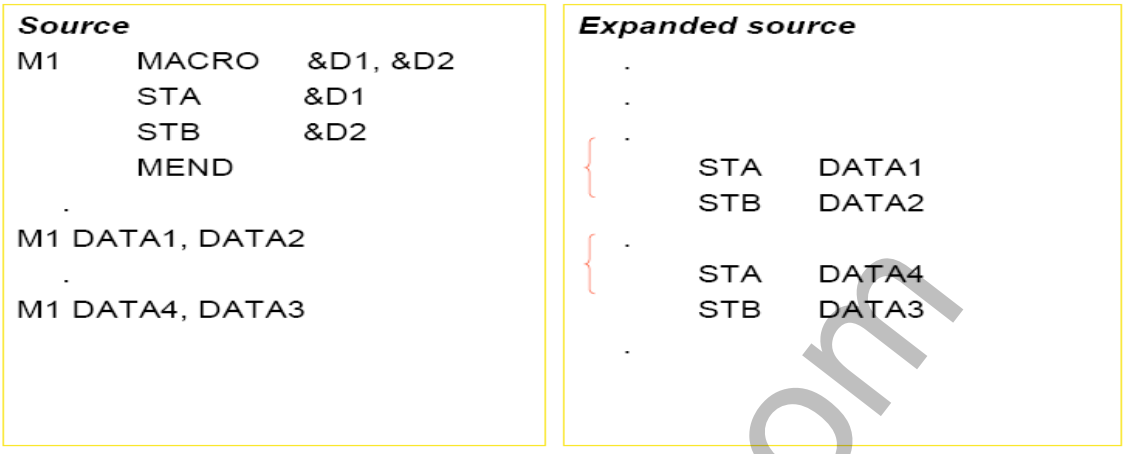
MAY /JUNE 2010

6a)Give out the algorithm for one pass macro processor (DEC 08/JAN 09)

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


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

```
Procedure DEFINE
        begin
            enter macro name into NAMTAB
            enter macro prototype into DEFTAB
            LEVEL   :- 1
            while LEVEL > 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
        while 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}
```

6b) Explain macro expansion.



DEC 09/JAN 10

6a) What are the data structures of macroprocessors.

- DEFTAB (definition table)
  - Stores the macro definition including macro prototype and macro body
  - Comment lines are omitted.
  - References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.
- NAMTAB
  - Stores macro names
  - Serves as an index to DEFTAB
  - Pointers to the beginning and the end of the macro definition (DEFTAB)
- ARGTAB
  - Stores the arguments of macro invocation according to their positions in the argument list
  - As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.

6b)What are the advantages and disadvantages of general purpose macroprocessors?

Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages

- **Pros**

- Programmers do not need to learn many macro languages.
- Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

- **Cons**
  - Large number of details must be dealt with in a real programming language
    - Situations in which normal macro parameter substitution should not occur, e.g., comments.
    - Facilities for grouping together terms, expressions, or statements
    - Tokens, e.g., identifiers, constants, operators, keywords
    - Syntax had better be consistent with the source programming language

UNIT 7: LEX AND YACC-1

DECEMBER 2010

7a) Give out the Structure of lex program (DEC 09/JAN 10)

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); to look for the string integer in the input stream and print the message "found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right

side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

7b) Explain Regular expression in detail. (MAY /JUNE 2010)(DEC 09/JAN 10)

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >"

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

[a-z0-9<>_]

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the

computer character set. Thus

[^abc]

Optional expressions: The operator ? Indicates an optional element of an expression. Thus

ab?c

matches either ac or abc.

Repetitions of classes are indicated by the operators * and +.

a*

is any number of consecutive a characters, including no character; while

a+

Character	Meaning
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches *any* character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.
|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.

( )	Groups a series of regular expressions.

7c) Write a Lex program to count the number of words. (MAY /JUNE 2010)

```
        %{
    int wordCount = 0;
    %}
    chars [A-za-z\_\'\.\"]
    numbers ([0-9])+
    delim [" "\n\t]
    whitespace {delim}+
    words {chars}+
    %%

{words} { wordCount++; /*
    increase the word count by one*/ }
    {whitespace} { /* do
    nothing*/ }
    {numbers} { /* one may
    want to add some processing here*/ }
    %%

    void main()
    {
        yylex(); /* start the analysis*/
        printf(" No of words:
        %d\n", wordCount);
    }
    int yywrap()
    {
        return 1;
    }
```

MAY /JUNE 2010

7a)Write a note on ANSI C macro language

In the ANSI C language, definitions and invocation of macros are handled by a preprocessor. This preprocessor is not integrated with the compiler.

ANSI C Macro definitions:

Macros are used to define symbolic names.

Example:

```
# define PI 3.14
# define NULL 0.
```

wherever PI and NULL occurs in the program, they are replaced by 3.14 and 0 respectively.

The macro

define EQ = = can be used to reduce the common errors which always occurs while writing programs in C. The programmer is allowed to write Statement like

```
while (I EQ O)
        {
        .
        .
        .

        }
```

and the macro processor would convert this to

```
        while ( I = = 0)
        {
        .
        .
        }
```

Use of Macros instead of functions:

Macro is more efficient than functions as the amount of computation required Is less than the overhead of calling a function.

Example:

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

and the macro invocation statement

```
        ABSDIFF (I+1, J-5)
```

would result in macro expansion statement

```
    ((I+1) > (J – 5) ? (I+1) – (J – 5) : (J – 5) – (I + 1))
```

In this macro definition, the macro processor simply makes string substitutions, without considering the syntax of the C language.

As a result, the macro Invocation statement

ABSDIFF (4 + 1, 8 – 6) would result in

4 + 1 > 8 – 6 ? 4 + 1– 8 – 6 : 8 – 6 – 4 + 1

which gives a unexpected result.

DEC 08/JAN 09

7b)Give out Lex structure. Write a Lex Program to count number of vowels and consonants

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); to look for the string integer in the input stream and print the message “found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If

the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces

```
%{
/*  to find vowels and consonents*/
  int vowels = 0;
  int consonents = 0;
%}
%%
[ \t\n]+
[aeiouAEIOU]   vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] consonents++;
.
%%
main()
{
  yylex();
  printf(" The number of vowels = %d\n", vowels);
  printf(" number of consonents = %d \n", consonents);
return(0);
}
```

DEC 07/ JAN 08

7a) structure of Lex and Yacc

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains actions, program fragments to be executed when the expressions are recognized.

Thus an individual rule might appear integer printf ("("found keyword INT"); to look for the string integer in the input stream and print the message "found keyword INT" whenever it appears.

The host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

7 b)What are conflicts in Yacc?

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr '-' expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

(expr - expr) - expr

or as

expr - (expr - expr)

(The first is called **left association**, the second **right association**).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

- expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr - expr

UNIT 8: LEX AND YACC-2

DECEMBER 2010

8a)Explain shift reducing parsing.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off the stack, and in the lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide

whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34. Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule).

In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

8b) Write a Yacc program to recognize the given arithematic expression

(MAY /JUNE 2010)

```
        #include<stdio.h>
        #include <string.h>
        #include <stdlib.h>
%}
% token num name
% left '+' '-'
% left '*' '/'
% left unaryminus
%%

st      :  name '=' expn
        | expn   { printf ("%d\n" $1);  }
        ;
expn    :  num   { $$ = $1 ; }
        | expn '+' num  { $$ = $1 + $3; }
        | expn '-'  num  { $$ = $1 - $3; }
        | expn '*' num  { $$ = $1 * $3; }
        | expn '/'  num  { if (num == 0)
                  { printf ("div by zero \n");
                    exit (0);
                  }
                  else
                   { $$ = $1 / $3; }
        | '(' expn ')'  { $$ = $2; }
```

```
        ;
%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}
```

8c) what are ambiguous grammar? Explain.(DEC 09/JAN 10)

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

expr : expr '-' expr

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

expr - expr - expr

the rule allows this input to be structured as either

(expr - expr) - expr

or as

expr - (expr - expr)

(The first is called **left association**, the second **right association**).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has seen:

expr - expr

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule; after applying the rule; the input is reduced to expr (the left side of the rule). The parser would then read the final part of the input:

- expr

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

expr - expr

MAY /JUNE 2010

8a) What are Yacc tools. What are the two types of conflicts that arises during parsing?

Yacc invokes two **disambiguating** rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ``if-then-else" construction:

stat : IF '(' cond ')' stat

| IF '(' cond ')' stat ELSE stat

;

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements. The first rule will be called the simple-if rule, and the second the if-else rule.

These two rules form an ambiguous construction, since input of the form

EXAMPLE:

IF (C1) IF (C2) S1 ELSE S2

can be structured according to these rules in two ways:

```
IF ( C1 ) {
    IF ( C2 ) S1
    }
ELSE S2
```

or

```
IF ( C1 ) {
    IF ( C2 ) S1
    ELSE S2
    }
```

- The second interpretation is the one given in most programming languages having this construct. Each ELSE is associated with the last preceding ``un-ELSE'd'' IF. In this example, consider the situation where the parser has seen

IF (C1) IF (C2) S1

and is looking at the ELSE. It can immediately reduce by the simple-if rule to get

IF (C1) stat

and then read the remaining input,

ELSE S2

and reduce

IF (C1) stat ELSE S2

by the if-else rule. This leads to the first of the above groupings of the input.

- On the other hand, the ELSE may be shifted, S2 read, and then the right hand portion of

IF (C1) IF (C2) S1 ELSE S2

can be reduced by the if-else rule to get

IF (C1) stat

which can be reduced by the simple-if rule.

- Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.
- This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

IF (C1) IF (C2) S1

- In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

stat : IF '(' cond ')' stat

- Once again, notice that the numbers following ``shift" commands refer to other states, while the numbers following ``reduce" commands refer to grammar rule

numbers. In the y.output file, the rule numbers are printed after those rules which can be reduced.

8c) What are the usage of YYparse()?

The yacc program gets the tokens from the lex program. Hence a lex program has be written to pass the tokens to the yacc. That means we have to follow different procedure to get the executable file.

i. The lex program <lexfile.l> is fist compiled using lex compiler to get **lex.yy.c.**

ii. The yacc program <yaccfile.y> is compiled using yacc compiler to get **y.tab.c.**

iii. Using c compiler b+oth the lex and yacc intermediate files are compiled with the lex library function. **cc y.tab.c lex.yy.c –ll.**

iv. If necessary out file name can be included during compiling with –o option.

DEC 07/ JAN 08

8a)Write a note on Editors

- An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.
- A text editor allows you to edit a text file (create, modify etc…). For example the Interactive text editors on Windows OS - Notepad, WordPad, Microsoft Word, and text editors on UNIX OS - vi, emacs , jed, pico.
- Normally, the common editing features associated with text editors are, Moving the cursor, Deleting, Replacing, Pasting, Searching, Searching and replacing, Saving and loading, and, Miscellaneous(e.g. quitting).

An interactive editor is a computer program that allows a user to create and revise a target document. Document includes objects such as computer diagrams, text, equations tables,

diagrams, line art, and photographs. In text editors, character strings are the primary elements of the target text.

Document-editing process in an interactive user-computer dialogue has four tasks:

- Select the part of the target document to be viewed and manipulated
- Determine how to format this view on-line and how to display it
- Specify and execute operations that modify the target document
- Update the view appropriately

The above task involves traveling, filtering and formatting. Editing phase involves – insert, delete, replace, move, copy, cut, paste, etc…

- Traveling – locate the area of interest
- Filtering - extracting the relevant subset
- Formatting – visible representation on a display screen

There are two types of editors. Manuscript-oriented editor and program oriented editors. Manuscript-oriented editor is associated with characters, words, lines, sentences and paragraphs. Program-oriented editors are associated with identifiers, keywords, statements. User wish – what he wants – formatted.

8b) Write a note on conditional Macro expansion

There are applications of macro processors that are not related to assemblers or assembler programming.

Conditional assembly depends on parameters provides

```
MACRO &COND
……..
```

```
IF (&COND NE ‘’)
        part I
ELSE
        part II
ENDIF
.....
ENDM
```

Part I is expanded if condition part is true, otherwise part II is expanded. Compare operators: NE, EQ, LE, GT.

Macro-Time Variables:

Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion. Any symbol that begins with symbol & and not a macro instruction parameter is considered as *macro-time variable.* All such variables are initialized to zero.

```
25        RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH
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 COUNTER
42                 RMO     A, S
43                 ENDIF
44                 IF      (&MAXLTH EQ ' ')
45                +LDT     #4096                SET MAX LENGTH = 4096
46                 ELSE
47                +LDT     #&MAXLTH             SET MAXIMUM RECORD LENGTH
48                 ENDIF
50        $LOOP    TD      =X'&INDEV'           TEST INPUT DEVICE
55                 JEQ      $LOOP               LOOP UNTIL READY
60                 RD      =X'&INDEV'           READ CHARACTER INTI 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 MAXIMUN LENGTH
85                 JLT     $LOOP                HAS BEEN REACHED
90        $EXIT    STX     &RECLTH              SAVE RECORD LENGTH
95                 MEND
```

Macro-time variable

Figure gives the definition of the macro RDBUFF with the parameters &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH. According to the above program if &EOR has any value, then &EORCK is set to 1 by using the directive SET, otherwise it retains its default value 0.

```
          .        RDBUFF   F31 BUF, RECL, 04, 2048

30                 CLEAR    X                   CLEAR LOOP COUNTER
35                 CLEAR    A
40                 LDCH    =X'04'               SET EOR CHARACTER
42                 RMO      A, S
47                +LDT     #2048                SET MAXIMUM RECORD LENGTH
50        $AALOOP  TD      =X'F3'               TEST INPUT DEVICE
55                 JEQ      $AALOOP             LOOP UNTIL READY
60                 RD      =X'F3'               READ CHARACTER INTI REG A
65                 COMPR    A, S                TEST FOR END OF RECORD
70                 JEQ      $AAEXIT             EXIT LOOP IF EOR
75                 STCH     BUF, X              STORE CHARACTE IN BUFFER
80                 TIXR     T                   LOOP UNLESS MAXIMUM LENGTH
85                 JLT      $AALOOP             HAS BEEN REACHED
90        $AAEXIT  STX      RECL                SAVE RECORD LENGTH
```

Fig : Use of Macro-Time Variable with EOF being NOT NULL

The above program show the expansion of Macro invocation statements with different values for the time variables. In figure the &EOF value is NULL. When the macro invocation is done, IF statement is executed, if it is true EORCK is set to 1, otherwise normal execution of the other part of the program is continued.

The macro processor must maintain a symbol table that contains the value of all macro-time variables used. Entries in this table are modified when SET statements are processed. The table is used to look up the current value of the macro-time variable whenever it is required.

When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.

If the value of this expression TRUE,

- The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement.
- If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF.
- Once it reaches ENDIF, it resumes expanding the macro in the usual way.

If the value of the expression is FALSE,

- The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement.
- The macro processor then resumes normal macro expansion.

The *macro-time* IF-ELSE-ENDIF structure provides a mechanism for either generating(once) or skipping selected statements in the macro body. There is another construct WHILE statement which specifies that the following line until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. The testing of this condition, and the looping are done during the macro is under expansion. The example shown below shows the usage of Macro-Time Looping statement.

WHILE-ENDW structure

- When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- TRUE
    - The macro processor continues to process lines from DEFTAB until it encounters 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.
- FALSE
    - The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.