2015 fall

6.a) Define object diagram. Identify possible objects and draw the object diagram for two pass assembler.

Object diagram indicates the methods that are invoked by each ~~other~~ object. For example: source program object invokes methods create, assign_location and translate on the source line objects.

Object diagram may also ~~indicate~~ invoke the class of each object. The invocation may be numbered to indicate the sequence in which they occur and the flows of info they cause.

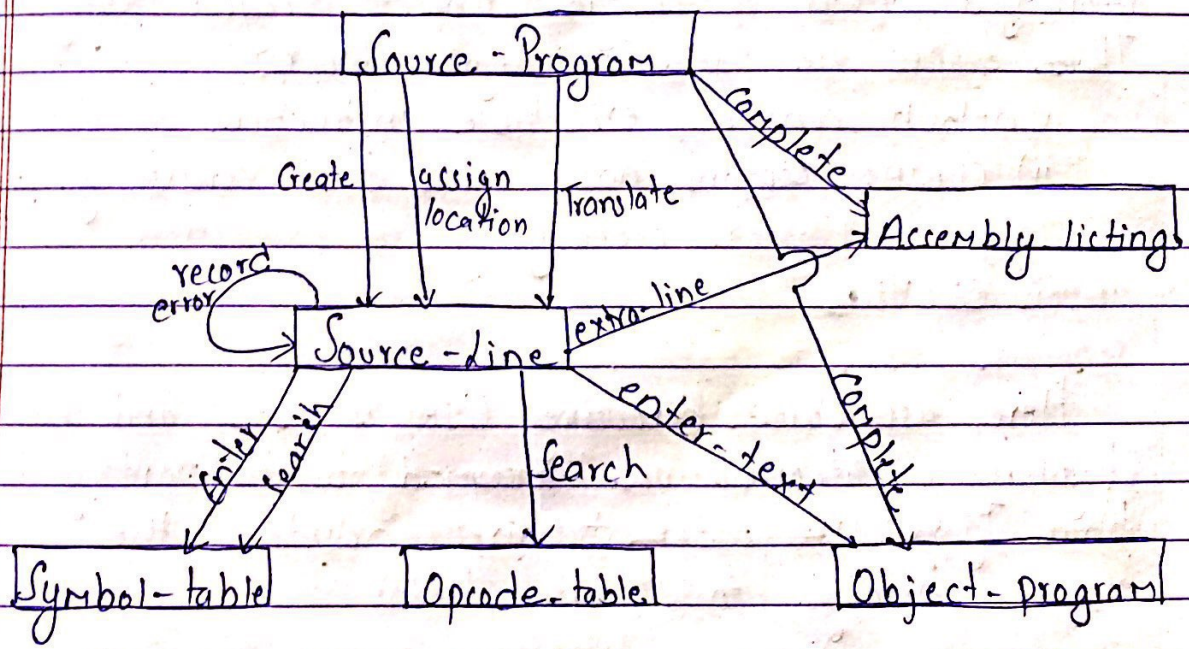


Fig:- Object diagram for Assembler.

2. Write about program blocks and control section.

Program Blocks

- The source programs logically contained main, sub-routines, data areas, etc. but assembler handle them as a single entity resulting in a single block of object code. Here, the generated machine instructions and data appeared in the same order as in source program. Some assemblers provide flexible handling by allowing the generated machine instruction and data to appear in object program in different order.

They are the segments of code that are re-arranged within a single object program unit. Three blocks are used in program blocks:

i) default: contains executable instruction.

ii) CDATA: Contains data areas that consist of larger block memory.

iii) CBLKS.

Here, assembler directive USE is used and at the begining, statements are assumed to be part of default block. If no USE statements are included, the entire program belongs to single block.

No longer need of extended formate, base register because large buffer (BUFFERS) is move to end of object program.

Program linking

Control Sections:

They are the segments of code that are translated into independent object program units.

∴ Control sections can be loaded and relocated independently of others.

→ Control sections are most often used for subroutines or other logical subdivision of program.

→ The programmer can assemble, load and manipulate each of those control sections separately. Because of this, there should be some means of linking them together.

→ Assembler directive CSECT is used

Syntax: SectionName CSECT

→ Separate location counter for each control section.

→ instructions in one control section might need to refer to instruction or data located in another section.

External Definitions:

• EXTDEF name[,name]

It names symbols that are defined in this control section and may be used by other sections.

Eg:- EXTDEF BUFFER, BUFFEREND, LENGTH.

External References:

• EXTREF name[,name]

→ It names symbols that are used in this control section and are defined elsewhere. Eg:- EXTREF, RDREC, WRREC.

→ To reference an external symbol, external format instruction is needed.

Dynamic linking:

- ~~Linkage~~ It is a linkage editor which performs linking before the program is loaded for execution. Linking loader performs linking at load time. Dynamic linking postpones the linking function until the execution time. A subroutine is loaded and linked to the rest of the program when it is first called. Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.

eg:- a single copy of standard library can be loaded into memory.

All c programs currently in execution can be linked to this one copy instead of linking a separate copy to each object program.

Dynamic linking can avoid the necessity of loading the entire library for each execution except those ~~necessity~~ necessary sub-routines.

Hence, Dynamic linking makes it possible for one object to be shared by several programs, and provides the ability to load the routines only when they are needed. It is a scheme that postpones the linking function until execution.

(#) Difference between linkage editor and linkage loader.

- A linkage loader performs all linking and relocation operations including automatic library search and loads the linked program into memory for execution.

- A linkage editor produces a linked version of the program, which is normally written to a file for later execution.

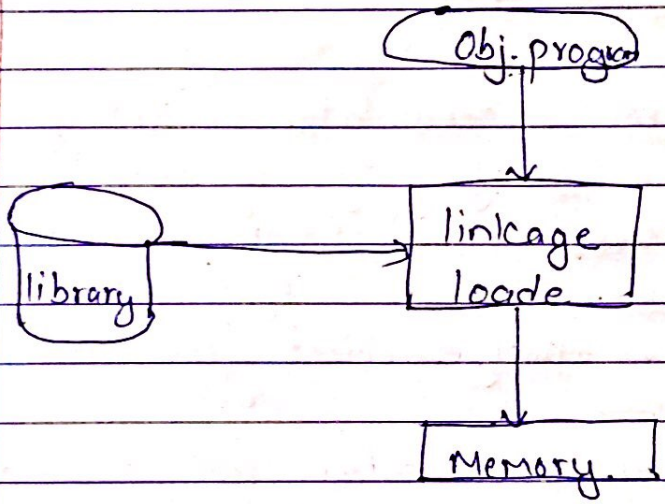


fig:- processing of object program in linkage loader

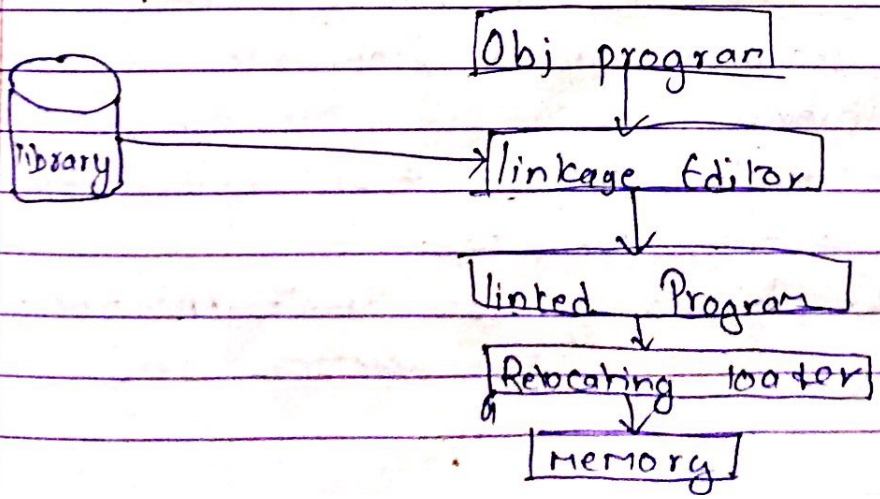


fig:- performing of object program in linkage editor.

(4) Principles of Object Oriented programming:

i) Object:
- It is a basic unit of OOP.
- It is a component of a program ~~which~~ that knows as how to perform certain actions and how to interact with other elements of program.
- Contains some data and defines a set of operation on that data that can be invoked by other parts of program.

ii) Class:
- It is a blueprint or tempelate or set of instruction to build a specific type of object.
- defines the instances variables and methods of an object.
- An instance is a specific object from specific class.
- Many objects can be created from some class.

iii) Encapsulation:
- means that the internal representation of an object is generally hidden from view outside of object definition.
~~- is the implementation of an object that contains some essential properties.~~
- is the hiding of data implementation by restricting access to accessors and mutators.

iv) Abstraction:

- is a model, a view or some other function representation for an actual item.
- is the implementation of an object that contains some essential properties and actions we can find in the original object we are representing.

v) Inheritance:

- is a way to reuse code of existing object or to establish a subtype from an existing object.
- the relationship of classes through inheritance gives rise to a hierarchy.

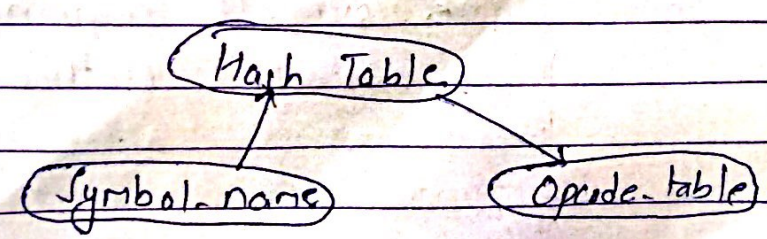
i) Subclass:

ii) Superclass:

vi) Polymorphism:

The same message sent to different objects can result in different behaviour. Polymorphism is one of the most powerful features of the object-oriented methodology as it manifests itself by having multiple methods all with same name, but slightly different functionality.

Source Program



Source line

(a) has-a relationship

(b) is-a relationship.

(#) Object Oriented design of an assembler:

According to Booch (1994), there are two different development process:

i) Micro.
ii) Macro.

i) Booch's macro process represents the overall activities of the development team on a long-range scale. It includes following activities:

C 1. Establish the requirements for the software (Conceptualization)
A 2. Develop an overall model of system's behaviour (Analysis)
D (design) 3. Create an architecture for the implementation (design)
E 4. Develop the implementation through successive refinement (evolution)
M 5. Manage the continued evolution of a delivered system (Maintainance)

This Macro process repeats itself after each release of software product. The overall sequence of events is similar to waterfall method.

ii) Booch's micro process essentially represent the daily activity of system developers. It consists of following activities:

1. Identify the classes and objects of system
2. Establish the behaviour and other attributes of the classes and objects.
3. Analyze the relationship among the classes and objects.
4. Specify the implementation of the classes and objects.
5. These activities may be repeated as needed with increasing level of detailed.

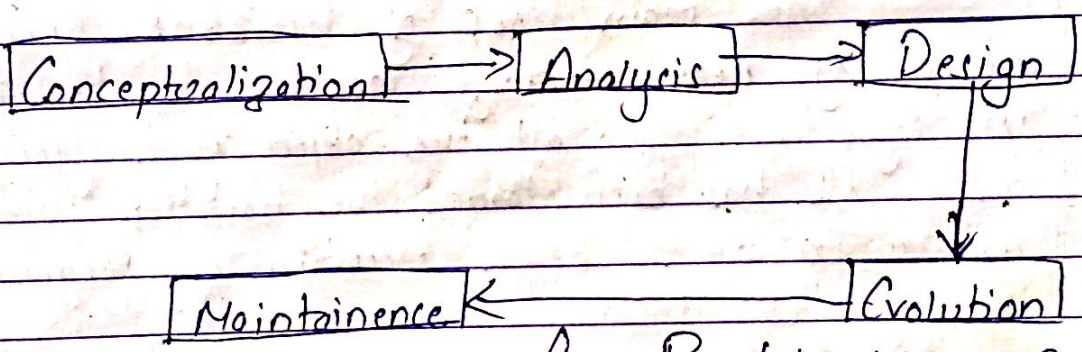


Fig: Booch's Macro process

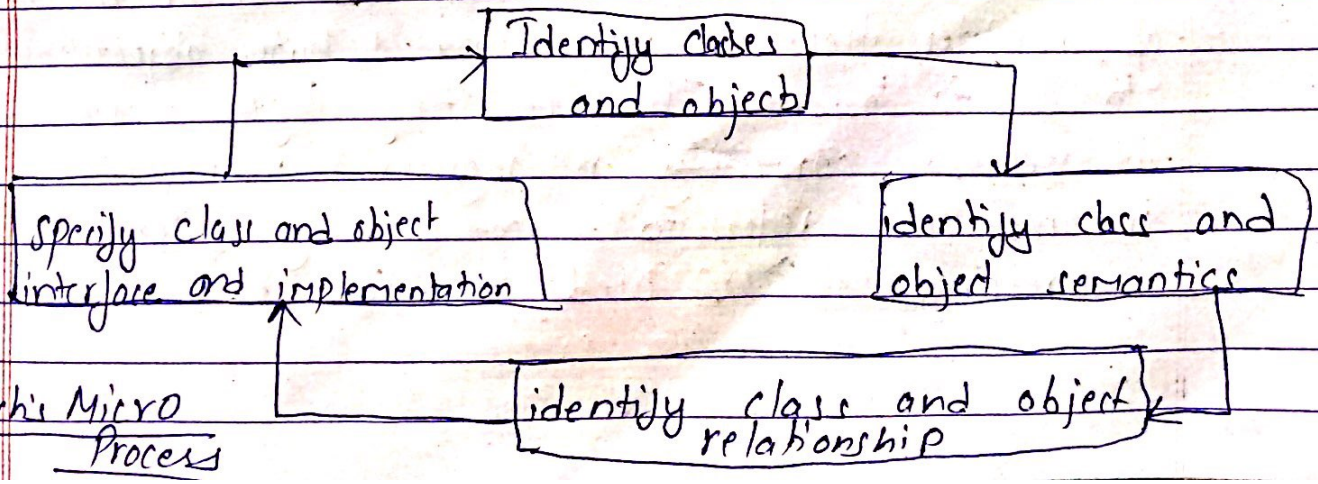


Fig: Booch's Micro Process

(#) A simple Bootstrap loader:

It is a special type of absolute loader. This bootstrap loads the first program to be run by the computer, usually an operating system. It begins at address 0 in memory of machine. It loads the operating system starting at address 80.

Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits just as it is in a Text record of a SIC object program. The object code from device F1 is always loade into consecutive byte of Memory, starting at address 80.

The main loop of bootstrap keeps the address of next memory location to be held in register X. After all the object code from the device F1 has been loaded the bootstrap jums to address 80 which began execution of program that was loaded.

Much of the work is done by subroutine GETC. GETC is used to read and convert a pair of charecter from F1 representing 1 byte object code to be loaded. The resulting byte is stored at address currently in register X using STCH instance that refers to location 0 using indexed addressing. TIXR is used to add 1 to value in X.

(4) Conditional Macro Expansion:

When a macro instruction is invoked the same sequence of statements are used to expand macro. Here, depending on the arguments supplied in the macro invocation, the sequence of statements generated for macro expansion can be modified. This adds greately to power and flexibility of macro language.

Macro time variable is a variable that begins with '&' and that is not a macro instruction parameter. It can be initialized to a value of 0. It can be set by macroprocessor directive SET and can be used to store working values during expansion. It stores the evaluation result of Boolean expression. It controls macro time conditional structure.

→ Macro time conditional structure.

(a) IF-ELSE-ENDIF

(b) While - ENDW.

Implementation of Conditional Macro Expansion:

(a) IF - ELSE - ENDIF

- firstly, a symbol table is maintained by macroprocessor, that contains the value of all macro time variables used.
- Entries in this table are made or modified when SET statements are processed.
- this table is used to look up the current

value of 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 value is TRUE
  * the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
  * If ELSE is encountered, then skipped to ENDIF.

- if value is false:
  * the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement.

(b) WHILE - ENDW:

- When a while statement is encountered during the expansion of a macro, the specified boolean expression is evaluated.

- If value is TRUE:
  * the macro processor continues to process lines from ~~deftab~~ DEFTAB until it encounters next ENDW statement.
  * When ENDW is encountered the macro processor returns to the preceeding WHILE, revaluates BOOLEAN expression and takes action again.

- if value is false:
  * the Macro processor skips ahead in DEFTAB until it finds the next ENDM statement and then resumes normal Macro expansion.

(4) Concatenation of Macro parameters.

⇒ Most Macro processors allow parameters to be concatenated with other character string. Say a program contains one series of variables named by the symbols XA1, XA2, XA3... and another is named by XB1, XB2, XB3.... etc.

- The body of macro definition might contain statement like:

```
SUM     MACRO   &ID
        LDA     X&ID1
        ADD     X&ID2
        ADD     X&ID3
        STA     X&ID4
        MEND
```

- Here, the begining of macro parameter is identified by starting symbol '&', however the end of parameter is not marked.

The problem is that the end of the parameter is not marked. Thus X&ID2 may mean 'X'+ID+1' or 'X'+ID2. To avoid this ambuigity a special concatenation operator '→' is used

Now new form become &ID->1.
'->' will not appear in macro expansion.

Example:-

```
SUM     MACRO   &ID
        LDA     X&ID->1
        ADD     X&ID->2
        ADD     X&ID->3
        STA     X&ID->4
        MEND
```

Now,

(i) SUM A

↓

```
LDA XA1
ADD XA2
ADD XA3
STA XA4
```

(ii) SUM BETA

↓

```
LDA XBETA1
ADD XBETA2
ADD XBETA3
STA XBETA4.
```

(11) MS-DOS Linker:

MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program. This executable program has the file name extension .EXE. LINK can also combine the translated programs with other modules from object code libraries.

1. MS-DOS compilers and assembler produce object modules (.OBJ).
2. Each .OBJ contains a binary image of the translated instructions and data of the program.
3. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program (.exe).
4. MS-DOS object module

Record types	Description
THEADR →	Translator header
TYPDEF, PUBDEF, EXTDEF →	External symbols and references
LNAMES, SEGDEF, GRPDEF →	Segment definition and grouping.

LEDATA, LIDATA	Translated instruction and Data
FIXUPP	Relocation and linking information
MODEND	End of object Module

Pass 1:

i) Segments are placed in same order as given in SEGDEF records.

ii) Segments from different object modules that have the same segment name and class are combined.

iii) Segments with same class, but different names are concatenated.

iv) Segments starting address is updated as these combinations and concatenations are performed.

Pass 2:

i) Process each LEDATA and LIDDATA record along with FIXUPP

ii) Relocation process that involves the starting address of the segment are added to a segment of FIXUPPS

iii) Build an image of executable program in memory.

iv) Write it to the executable file (.EXE) file.