Module 3, Single-pass Assembler:

In this case the whole process of scanning, parsing, and object code conversion is done in single pass. The only problem with this method is resolving forward reference.

This is shown with an example below:

10 1000 FIRST STL RETADR 141033

--

--

--

--

95 1033 RETADR RESW

Pass-1

*Assign addresses to all the statements

*Save the addresses assigned to all labels to be used in Pass-2

*Perform some processing of assembler directives such as RESW, RESB to find the length of data areas for assigning the address values.

*Defines the symbols in the symbol table(generate the symbol table)

Pass-2

*Assemble the instructions (translating operation codes and looking up addresses).

*Generate data values defined by BYTE, WORD etc.

*Perform the processing of the assembler directives not done during pass-1.

*Write the object program and assembler listing.

Assemblers Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (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. 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.

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.

LOCCTR: Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label

The Algorithm for 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
begin
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}

Machine-Independent features:

These are the features which do not depend on the architecture of the machine. These are:
*Literals *Symbol-Defining Statements *Expressions *Program blocks *Control Section

1.Literals:

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

2.Symbol-Defining Statements:

EQU Statement: Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this EQU (Equate).
ORG Statement: This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (for origin).

3.Expressions:

Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /. Division is usually defined to produce an integer result. Individual terms may be constants, user-defined symbols, or special terms. The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location
Absolute Expressions: The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. Example: MAXLEN EQU BUFEND-BUFFER
Relative Expressions: All the relative terms except one can be paired as described in “absolute”. The remaining unpaired relative term must have a positive sign. Example: STAB EQU OPTAB + (BUFEND – BUFFER)

4.Program blocks

it 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.

5.Control Sections:

It 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. 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 syntax

secname CSECT: separate location counter for each control section
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.
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
Modification record
*Col. 1-M *Col. 2-7 Starting address of the field to be modified (hexadecimal)
*Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
*Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field
Assembler directive used is EQU.
Syntax: symbol EQU value
*Used to improve the program readability, avoid using magic numbers, make it easier to find and change constant values
*Replace +LDT #4096 with
*MAXLEN EQU 4096 +LDT #MAXLEN
*Define mnemonic names for registers.
A EQU 0 RMO A,X
XEQU 1
•Expression is allowed MAXLENEQUBUFEND-BUFFER
Assembler directive ORG
•Allow the assembler to reset the PC to values
o Syntax: ORG value
•When ORG is encountered, the assembler resets its LOCCTR to the specified value.
•ORG will affect the values of all labels defined until the next ORG.
•If the previous value of LOCCTR can be automatically remembered, we can return to the normal use of LOCCTR by simply writing
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
o Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error
ALPHA RESW 1
BETA EQU ALPHA
Not Allowed:
BETA EQU ALPHA
ALPHA RESW 1
Multi_Pass Assembler:
*For a two pass assembler, forward references in symbol definition are not allowed:
ALPHA EQU BETA
BETA EQU DELTA
DELTA RESW 1
o Symbol definition must be completed in pass 1.
*Prohibiting forward references in symbol definition is not a serious inconvenience.
o Forward references tend to create difficulty for a person reading the program.
Two-Pass Assembler
*Most assemblers
-Processing the source program intotwopasses.
-The internal tablesand subroutinesthat are used only during Pass 1.
-The SYMTAB, LITTAB, and OPTAB are used by both passes.
*The main problems to assemble a program in one pass involves forward references.

One-Pass Assemblers

*Eliminate forward references

-Data items are defined before they are referenced.

-But, forward references to labels on instructions cannot be eliminated as easily.

-Prohibitforward references to labels.

*Two types of one-pass assembler.

-One type produces object code directly in memoryfor immediate execution.

-The other type produces the usual kind of object programfor later execution.

MASM Assembler

This section describes some of the features of the Microsoft MASM assembler for Pentium and other x86 systems. Further information about MASM can be found in Barkakati (1992). The programmer of an x86 system views memory as a collection of segments. An MASM assembler language program is written as a collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST, and STACK. During program execution, segments are addressed via the x86 segment registers. In most cases, code segments are addressed using register CS, and stack segments are addressed using register SS. These segment registers are automatically set by the system loader when a program is loaded for execution.

Register CS is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is set to indicate the last stack segment processed by the loader. Data segments (including constant segments) are normally addressed using DS, ES, FS, or GS. The segment register to be used can be specified explicitly by the programmer (by writing it as part of the assembler language instruction). If the programmer does not specify a segment register, one is selected by the assembler. By default, the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME. For example, the directive - ASSUME ES : DATASEG2 tells the assembler to assume that register ES indicates the segment DATASEG2. Thus, any references to labels that are defined in DATASEG2 will be assembled using register ES. It is also possible to collect several segments into a group and use ASSUME to associate a segment register with the group.

Module 4, Pass 2 Algorithm:

*May print a load map for all control sessions and symbols, useful in program debugging

*Actually loading, relocation, and linking

*CSADDR is used as in Pass 1

*For each Text record, the object code is placed to the address (=the address in the instruction + CSADDR).

*For each Modification record, from the name, looking for ESTAB to get the address (in another control session), then add or subtract from the indicated location.

*Transfer the control to the starting address indicated by EXECADDR

Algorithm for Pass 2 of a Linking loader

1) As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

2) When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.

3) This value is then added to or subtracted from the indicated location in memory.

4) The last step performed by the loader is usually the transferring of control to the loaded program to begin execution

•The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

•If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.

•Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine.