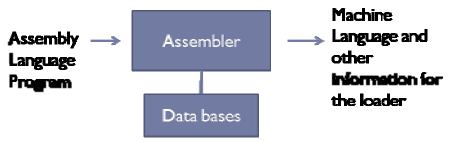
ASSEMBLER

An assembler is a program that accepts as input an assembly language program and produces its machine language equivalent along with information for the loader. Although we the main task of the assembler is to produce machine code, but also has to produce other information for the loader to use. E.g. externally defined symbols must be noted and passed on to the loader as the loader doesn't know the address (value) of these symbols and it is up to the loader to find the programs containing them, load them in core and place the values of these symbols in the calling function.


Assembly Language Program
Assembler
Machine Language and other Information for the loader
Data bases

Here we will consider the programs as "decks" that are inputs as well as outputs of the assembler. Basically decks of cards were used in olden day as secondary memory.

GENERAL DESIGN PROCEDURE

There is a general way in which all software is designed and this should be known before designing the assembler. Listed below are six steps that should be followed by the designer:

1. Specify the problem.
2. Specify data structures.
3. Define format for data structures.
4. Specify algorithm
5. Look for modularity (i.e. dividing a program into modules)
6. Repeat 1 through 5 on each module in step 5.

DESIGN OF ASSEMBLER

STATEMENT OF PROBLEM

Source program			*Relative address*	*Mnemonic instruction*		*Relative address*	*Mnemonic instruction*	
JOHN	START	0						
	USING	*,15						
	L	1,FIVE	0	L	1,_(0,15)	0	L	1,16(0,15)
	A	1,FOUR	4	A	1,_(0,15)	4	A	1,12(0,15)
	ST	1,TEMP	8	ST	1,_(0,15)	8	ST	1,20(0,15)
FOUR	DC	F'4'	12	4		12	4	
FIVE	DC	F'5'	16	5		16	5	
TEMP	DS	1F	20	-		20	-	
	END							

Let us pretend to an assembler and translate the given assembly language program into machine language. The building blocks of the assembly programs are pseudo codes, machine codes and symbols or literals. The pseudo codes as discussed give information to the assembler. The machine codes has got unique binary code that can be found out from the 360 manual and can be substituted. Next the symbols or literals that represent values stored at particular locations are substituted with these memory addresses.

As we scan through the program, we first encounter a pseudo code START that informs the assembler that the name of the program is JOHN. Next is the USING pseudo code that informs the assembler that R15 is used as the base register and during execution will contain the address of the first instruction of the program. The load instruction next is substituted for its binary values and then R1 and since location of FIVE is not known so we leave space for offset, and substitute 0 for index register since it is not used and 15 in place of base register. So the entry

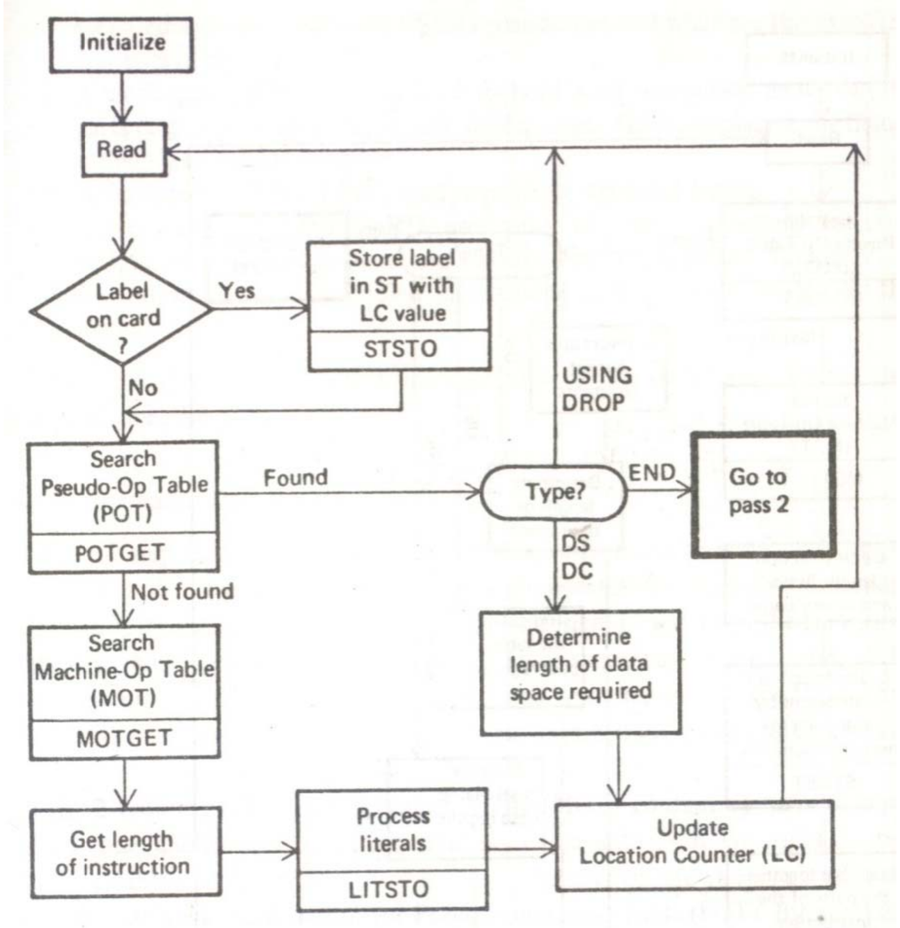

Initialize
Read
Label on card ?
Yes
Store label in ST with LC value
STSTO
No
USING DROP
Search Pseudo-Op Table (POT)
POTGET
Found
Type?
END
Go to pass 2
DS DC
Not found
Search Machine-Op Table (MOT)
MOTGET
Determine length of data space required
Get length of instruction
Process literals
LITSTO
Update Location Counter (LC)

FIGURE 3.3 Pass 1 overview: define symbols

becomes L 1,_(0, 15). The add (A) instruction and store (ST) instruction is assembled in the same way as the address of FOUR and TEMP is not known. Next we have pseudo codes DC then define symbols FOUR, FIVE in the relative location 12 and 16. The DS pseudo codes then define TEMP as 1F. And the END informs the assembler the program terminates over here. And all these codes are produced in the centre column of the figure.

As an assembler we now need to go to the starting of the program again to fill the offsets of the symbols define in the first pass, this is clearly shown in the third column of the figure. Since in assembly programs the symbols are used before they are declared so it become necessary to perform the second pass. The first pass defines the symbols and the second pass generates the instructions. It is also possible to have a single pass compiler there this kind of situation is not found. Specifically the assembler must do the following.

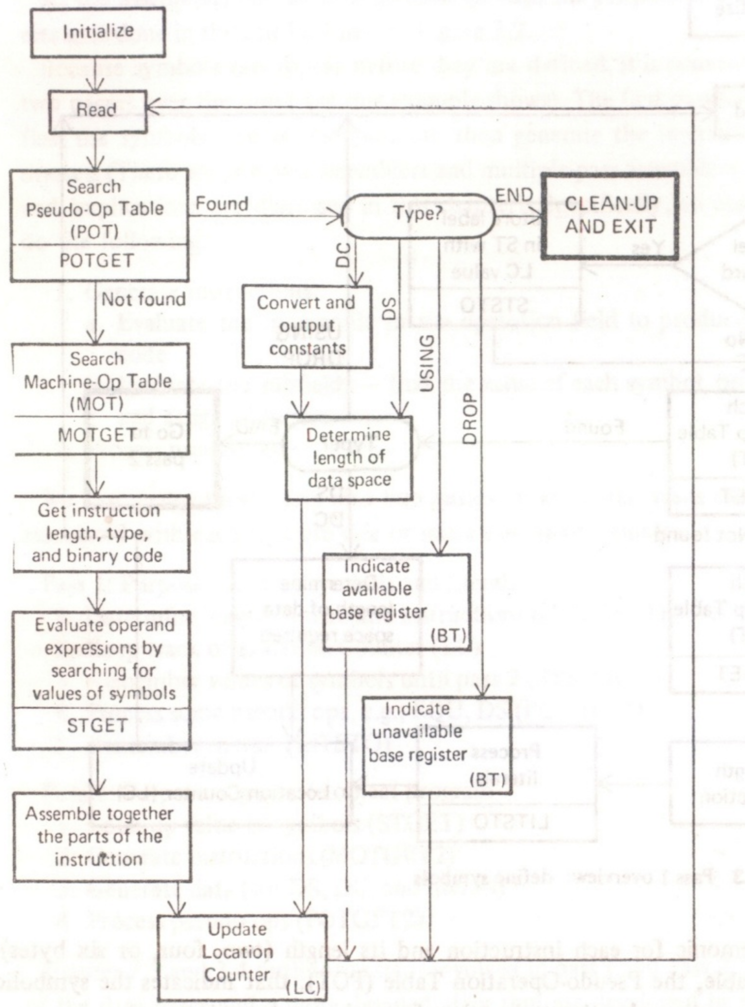

Initialize
Read
Search Pseudo-Op Table (POT)
POTGET
Found
Type?
END
CLEAN-UP AND EXIT
DC
DS
USING
DROP
Not found
Convert and output constants
Search Machine-Op Table (MOT)
MOTGET
Determine length of data space
Get instruction length, type, and binary code
Indicate available base register (BT)
Evaluate operand expressions by searching for values of symbols
STGET
Indicate unavailable base register (BT)
Assemble together the parts of the instruction
Update Location Counter (LC)

FIGURE 3.4 Pass 2 overview: evaluate fields and generate code

1. Generate instructions:
   a. Evaluate the mnemonics in the operation field to produce its machine code.
   b. Evaluate the subfields – find the value of each symbol, process literals, and assign addresses.
2. Process pseudo ops.

We can group these tasks into two passes or sequential scans over the input; associated with each task are one or more assembler modules.

Pass 1: Purpose – define symbols and literals.
1. Determine length of machine instructions (MOTGET1).
2. Keep track of Location Counter (LC).
3. Remember values of symbols until pass 2 (STSTO).
4. Process some pseudo ops, e.g. EQU, DS(POTGET1).
5. Remember literals (LITSTO).

Pass 2: Purpose – generate object programs.
1. Look up values of symbols (STGET).
2. Generate instructions (MOTGET2).
3. Generate data (for DS, DC and literals).
4. Process pseudo ops (POTGET2).

The figures represent the above steps.

DATA STRUCTURE

The second step in our design procedure is to establish the databases that we have to work with.

Pass 1: databases:
1. Input source program.
2. A Location Counter (LC), used to keep track of each instruction's location.
3. A table, the Machine-Operation Table (MOT), that indicates the symbolic mnemonic for each instruction and its length (two, four, or six bytes).
4. A table, the Pseudo-Operation Table (POT), that indicates the symbolic mnemonic and action to be taken for each pseudo-op in pass 1.
5. A table, the Symbol Table (ST), that is used to store each literal and its corresponding value.
6. A table, the Literal Table (LT), that is used to store each literal encountered and its corresponding assigned location.
7. A copy of the input to be used later by pass 2. This may be stored in a secondary storage device, such as magnetic tape, disk, or drum, or the original source deck may be read by the assembler a second time for pass 2.

Pass 2: databases:
1. Copy of source program input to pass 1.
2. Location Counter (LC).

3. A table, the Machine Operation Table (MOT), that indicates for each instruction: (a)Symbolic mnemonic , (b)Length, (c)Binary machine op code, (d)Format (e.g. RS, RX, SI)
4. A table, the Pseudo-Operation Table (POT), that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in pass 2.
5. The Symbol Table (ST), prepared by pass1, containing each label and its corresponding value.
6. A table, the Base Table (BT), that indicates which registers are currently specified as base registers by USING pseudo-ops and what are the specified contents of these registers.
7. A work-space, INST, that is used to hold each instruction as its various parts (e.g. binary op-code, register field, length field, displacement field) are being assembled together.
8. A workspace, PRINT LINE, used to produce a printable listing.
9. A workspace, PUNCH CARD, used prior to actual outputting for converting the assembled instructions into the format needed by the loader.
10. An output deck of assembled instructions in the format needed by the loader.

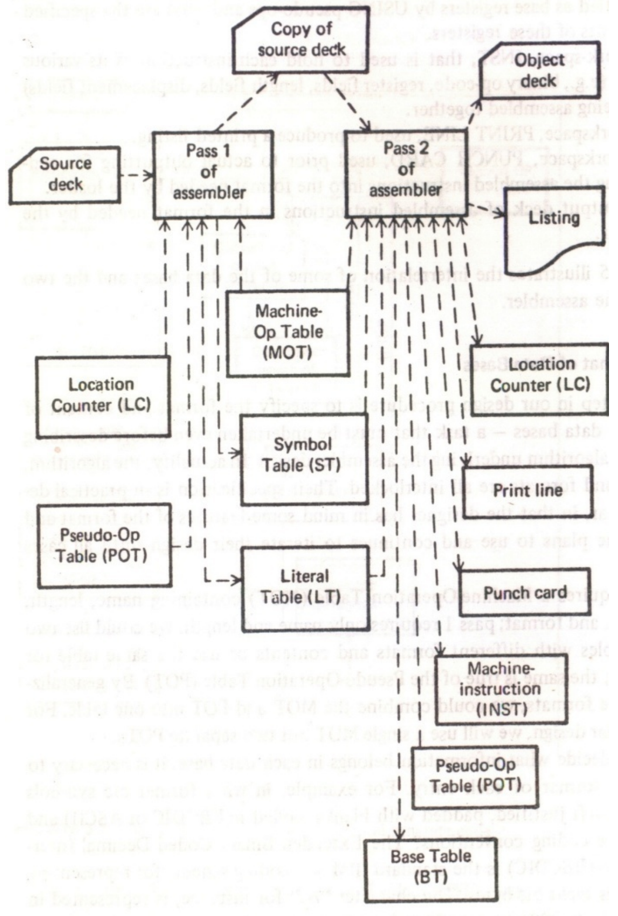


FIGURE 3.5 Use of data bases by assembler passes

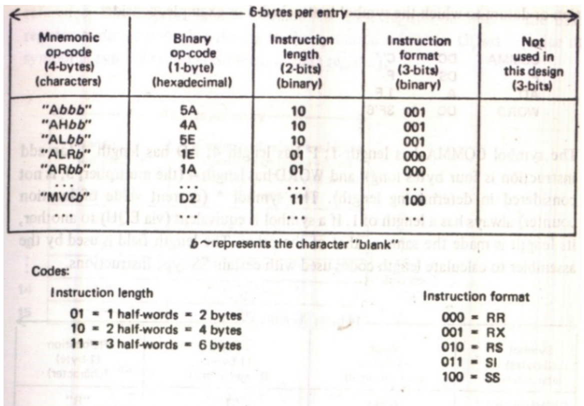
FORMAT OF DATABASES

Here we specify the format and content of each of the databases- a task that must be undertaken even before describing the specific algorithm underlying the assembler design. Actually the algorithm, databases and formats are interrelated to each other.

Pass 2 requires a Machine Operation Table (MOT) containing name, length, binary code and formats, where as pass 1 requires only name and the length. We could use two separate tables with different formats and contents or use the same table for both passes; the same is true of the Pseudo Operation Table (POT). By generalization we can combine the MOT and POT into one table. For this particular design we will use a single MOT but two separate POTs.

Once we decide what information belongs in each database, it is necessary to specify the format of each entry e.g. format of storing string may be EBCDIC or ASCII etc. IBM 360 employs EBCDIC for storing symbols.

The MOT and POTs are example of *fixed tables* i.e. the content of these tables are not filled or altered during the assembly process. The figure 3.6 depicts the possible contents and format of MOT.


6-bytes per entry

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"A*bbb*"	5A	10	001	
"AH*bb*"	4A	10	001	
"AL*bb*"	5E	10	001	
"ALR*b*"	1E	01	000	
"AR*bb*"	1A	01	000	
...	...	...	...	
"MVC*b*"	D2	11	100	
...	...	...	...	

b ~ represents the character "blank"

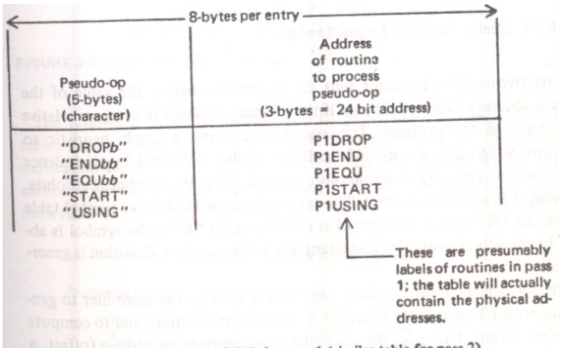
Codes:

Instruction length
01 = 1 half-words = 2 bytes
10 = 2 half-words = 4 bytes
11 = 3 half-words = 6 bytes

Instruction format
000 = RR
001 = RX
010 = RS
011 = SI
100 = SS

FIGURE 3.6 Machine-Op Table (MOT) for pass 1 and pass 2

The op code is the key and its value is the binary op code equivalent, which is stored for use in generating machine codes. The instruction length is store for use in updating the location counter; the instruction format for used in forming the machine language equivalent. Figure 3.7 depicts a possible pseudo op table. Each pseudo op is listed with an associated pointer to the assembler routine for processing the pseudo op.

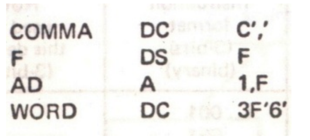

8-bytes per entry

Pseudo-op (5-bytes) (character)	Address of routina to process pseudo-op (3-bytes = 24 bit address)
"DROP*b*"	P1DROP
"END*bb*"	P1END
"EQU*bb*"	P1EQU
"START"	P1START
"USING"	P1USING

These are presumably labels of routines in pass 1; the table will actually contain the physical addresses.

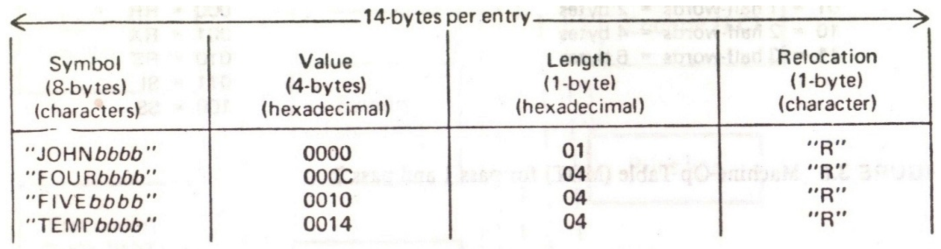
FIGURE 3.7 Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

The Symbol Table and Literal Table (figure 3.8) include for each entry not only the name and assembly time value field bit also a length field and a relative location indicator. The length filed indicates the length (in bytes) of the instruction or datum to which the symbol is attached. The symbol COMMA being a character has length 1, symbol F being a floating point has length 4, AD being an add instruction has length 4 and symbols WORD being a literal has length 4. If a symbol



COMMA	DC	C','
F	DS	F
AD	A	1,F
WORD	DC	3F'6'

equivalent to another its length is made the same as that of the other. * if of length 1 and has the address as that of current content of the LC.

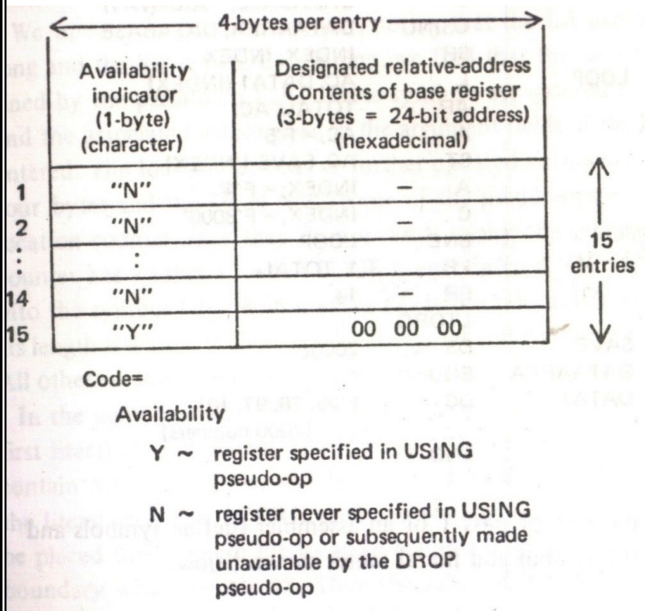


Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

14-bytes per entry

FIGURE 3.8 Symbol Table (ST) for pass 1 and pass 2

The relative location indicator tells the assembler whether the value of the symbol is absolute (does not change if the program is moved in core), or relative to the base of the program. If the symbol is defined by equivalence with a constant (e.g. 6) or an absolute symbol then it is absolute "A", otherwise it is relative "R".



	Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24-bit address) (hexadecimal)
1	"N"	—
2	"N"	—
:	:	
14	"N"	—
15	"Y"	00 00 00

4-bytes per entry
15 entries

Code= Availability

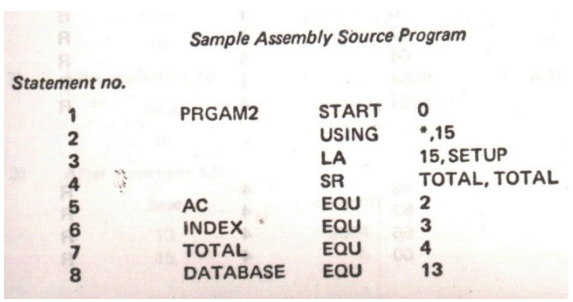
Y ~ register specified in USING pseudo-op

N ~ register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

FIGURE 3.9 Base Table (BT) for pass 2

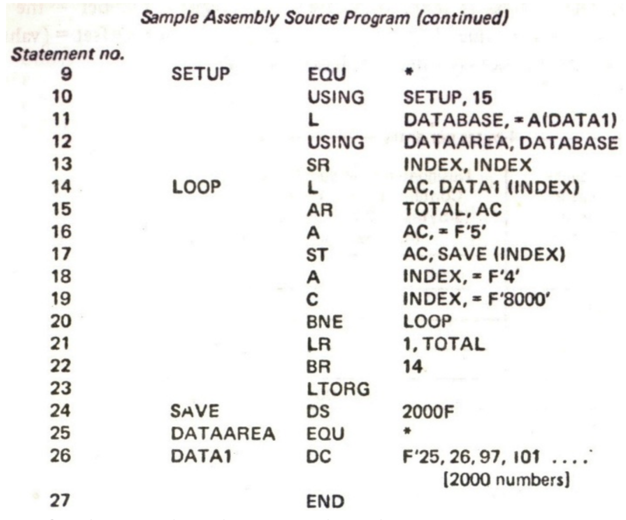
Figure 3.9 depicts a possible base table that is used by the assembler to generate the proper base register reference in machine instructions and to compute the correct offsets. The assembler must generate an address (offset, base register number, index register number) for most symbolic references. The ST contains the address of the symbol relative to the beginning of the program. While doing so the assembler use the BT to choose a base register that will contain a value closest to the symbolic reference. The address is then formulated as Base register number = the base register containing a values closest to the symbolic reference and Offset= (value of the symbol in ST) - (content of the base register) it the address is "A".

The following program is used to illustrate the use of the *variable table* (ST, LT and BT) are to demonstrate the motivation for the algorithms presented in the next section. Here we are only concerned with assembling the program and its specific function is irrelevant. In keeping with the purpose of pass 1 of an assembler (define symbols and literals), we can create the symbols and literal tables as shown in the figure for VT.



Sample Assembly Source Program

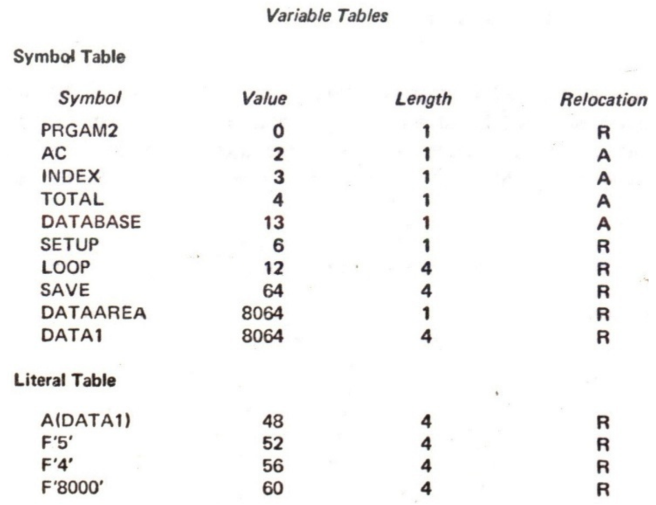
```
Statement no.
    1      PRGAM2      START   0
    2                  USING   *,15
    3                  LA      15,SETUP
    4                  SR      TOTAL, TOTAL
    5      AC          EQU     2
    6      INDEX       EQU     3
    7      TOTAL       EQU     4
    8      DATABASE    EQU     13
```



Sample Assembly Source Program (continued)

```
Statement no.
    9      SETUP       EQU     *
   10                  USING   SETUP, 15
   11                  L       DATABASE, = A(DATA1)
   12                  USING   DATAAREA, DATABASE
   13                  SR      INDEX, INDEX
   14      LOOP        L       AC, DATA1 (INDEX)
   15                  AR      TOTAL, AC
   16                  A       AC, = F'5'
   17                  ST      AC, SAVE (INDEX)
   18                  A       INDEX, = F'4'
   19                  C       INDEX, = F'8000'
   20                  BNE     LOOP
   21                  LR      1, TOTAL
   22                  BR      14
   23                  LTORG
   24      SAVE        DS      2000F
   25      DATAAREA    EQU     *
   26      DATA1       DC      F'25, 26, 97, 101 ....
                                   [2000 numbers]
   27                  END
```

For the symbol PRGAM2, its value is its relative location. By IBM convention its length is 1. We update the location counter, noting that the LA instruction of 4B long and the SR is two. Continuing, we find that the next five symbols are defined by pseudo op EQU. These symbols are entered into the ST and the associated values given in the argument fields of the EQU statement are entered. The LC is further updated noting that the L instruction is 4B and the SR is 2B. (None of the pseudo ops encounter affects the LC since they do not result in any object code). Thus location counter has a value 12 when LOOP is encountered. There for LOOP is entered into the ST with a value 12. It is a re-locatable variable and so noted and is of size 4B as it denotes an instruction of size 4B. All other symbols are entered in this manner.



Variable Tables

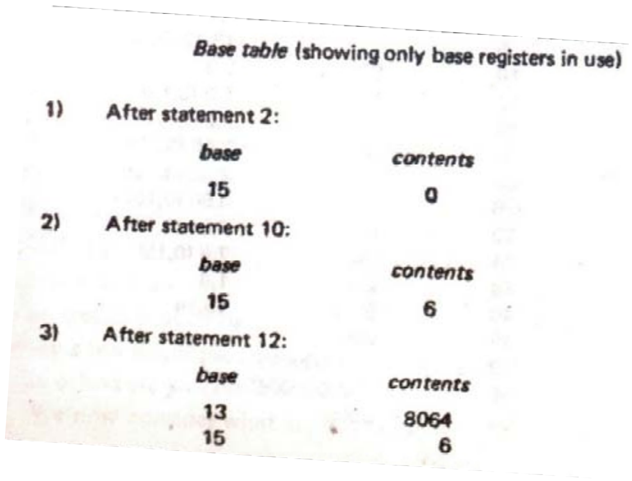
Symbol Table

Symbol	*Value*	*Length*	*Relocation*
PRGAM2	0	1	R
AC	2	1	A
INDEX	3	1	A
TOTAL	4	1	A
DATABASE	13	1	A
SETUP	6	1	R
LOOP	12	4	R
SAVE	64	4	R
DATAAREA	8064	1	R
DATA1	8064	4	R

Literal Table

A(DATA1)	48	4	R
F'5'	52	4	R
F'4'	56	4	R
F'8000'	60	4	R

In the same pass all literals are recognized and entered into literal table. The first literal is in statement 11 and its value is the address of the location that will contain the literal. Since this is the first literal, it will have the first address of the literal area. The LTORG pseudo op (statement 23) forces the literal table to be placed there, where the LC is updated to the next double word boundary which equals 48. Thus the value of '=A(DATA1)' is its address, 48. Similarly, the value of the literal F'5' is the next location in the literal table, 52 and so on.



Base table (showing only base registers in use)

1) After statement 2:

base	contents
15	0

2) After statement 10:

base	contents
15	6

3) After statement 12:

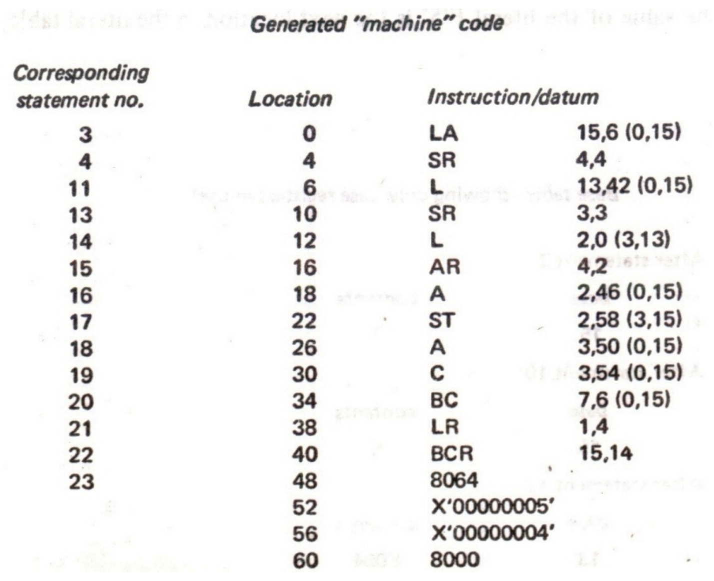
base	contents
13	8064
15	6

The LT and ST being completed, we may initiate pass, whose purpose is to evaluate arguments and generate code. To generate proper address in an instruction, we need to know the base register. The assembler of course does not know the execution time value of the base register, but it does know the value relative to the start of the program, therefore, the assembler enters as "contents" its relative value. This value is used to compare the offset. Processing the USING pseudo-ops produces the BT shown above.

For each instruction in pass 2, we create the equivalent machine language instruction as shown below. For example, for statement 3 we:

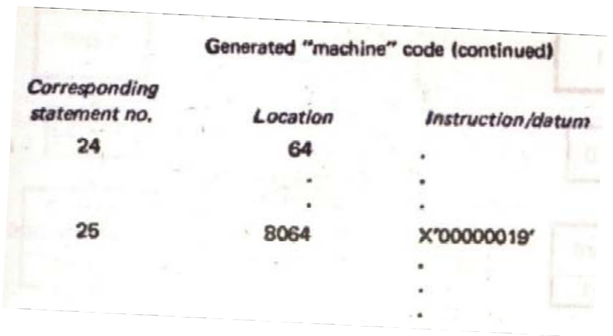
1. Look up value of SETUP in symbol table (which is 6).
2. Look up value of op code in MOT (binary op code for LA)
3. Formulate address
   a. Determine base register –pick one that has content closest to value of SETUP (register 15)
   b. Offset = value of symbol – content of base register = 6 – 0 = 6.
   c. Formulate address: offset(index register, base register) = 6(0,15).
4. Average output code in appropriate formula.

Similarly we generate instructions for the remaining code as shown below:



Generated "machine" code

Corresponding statement no.	Location	Instruction/datum	
3	0	LA	15,6 (0,15)
4	4	SR	4,4
11	6	L	13,42 (0,15)
13	10	SR	3,3
14	12	L	2,0 (3,13)
15	16	AR	4,2
16	18	A	2,46 (0,15)
17	22	ST	2,58 (3,15)
18	26	A	3,50 (0,15)
19	30	C	3,54 (0,15)
20	34	BC	7,6 (0,15)
21	38	LR	1,4
22	40	BCR	15,14
23	48	8064	
	52	X'00000005'	
	56	X'00000004'	
	60	8000	



Generated "machine" code (continued)

Corresponding statement no.	Location	Instruction/datum
24	64	.
	.	.
	.	.
25	8064	X'00000019'
		.
		.
		.

ALGORITHM

The flowchart in the figure 3.10 and 3.11 describe in some detail an algorithm for an assembler for an IBM 360 computer. These diagrams represent a simplification of the operations performed in a complex assembler but they illustrate most of the logical processes involved.

Pass 1: DEFINE SYMBOLS

The purpose of the first pass is to assign a location to each instruction and data defining pseudo-instruction, and this to define vales for symbols appearing in the label fields of the source program. Initially, the LC is set to the first location in the program (relative address 0). Then a source statement is read. The operation code field is examined to determine if it is a pseudo op; if it is not, the table of MOT is searched to find a match for the source statement's op code field. The matched MOT entry specifies the length (2,4 or 6 B) of the instruction. The operand field is scanned for the presence of a literal. If new literal is found, it is entered into the LT for later processing. The label field of the source statement is then examined for the presence of a symbol. If there is a label, the symbol is saved in ST along with the current value of the LC. Finally, the current value of the LC is incremented by the length of the instruction and a copy of the source code is saved for use by pass 2. The above sequence is then repeated for the next instruction.

The loop described is physically a small portion of pass 1 even though it is the most important function. The largest sections of pass 1 and pass 2 are devoted to the special processing needed for the various pseudo operations. For simplicity, only a few major pseudo operations are explicitly indicated in the flowchart; the others are processed in a straightforward manner.

We now consider what must be done to process a pseudo op. the simplest procedure occurs for USING and DROP. Pass 1 is only concerned with pseudo-ops that define symbols (labels) or affect the LC; USING and DROP do neither, the assembler need only save the USING and DROP cards for pass 2.

In the case of the EQU pseudo op during pass 1, we are concerned only with defining the symbol in the label field. This requires evaluating the expression in the operand field. (The symbols in the operand field of an EQU statement must have been defined previously.)

The DS and DC pseudo ops can affect both the LC and the definition of symbols in pass 1. The operand field must be examined to determine the number of bytes of storage required. Due to requirements for certain alignment conditions (e.g. full words must start on a byte whose address is a multiple of four), it may be necessary to adjust the LC before defining the symbol.

When the END pseudo op is encountered, pass 1 is terminated. Before transferring control to pass 2, there are various "housekeeping" operations that must be performed. These include assigning locations to literals that have been collected during pass 1, a procedure that is very similar to that for the DC pseudo op. Finally, conditions reinitialized for processing by pass 2.

Pass 2: GENERATE CODE

After all the symbols have been defined by pass 1, it is possible to finish the assembly by processing each card and determining values for its op code and its operand field. In addition, pass 2 must structure the generated code into the

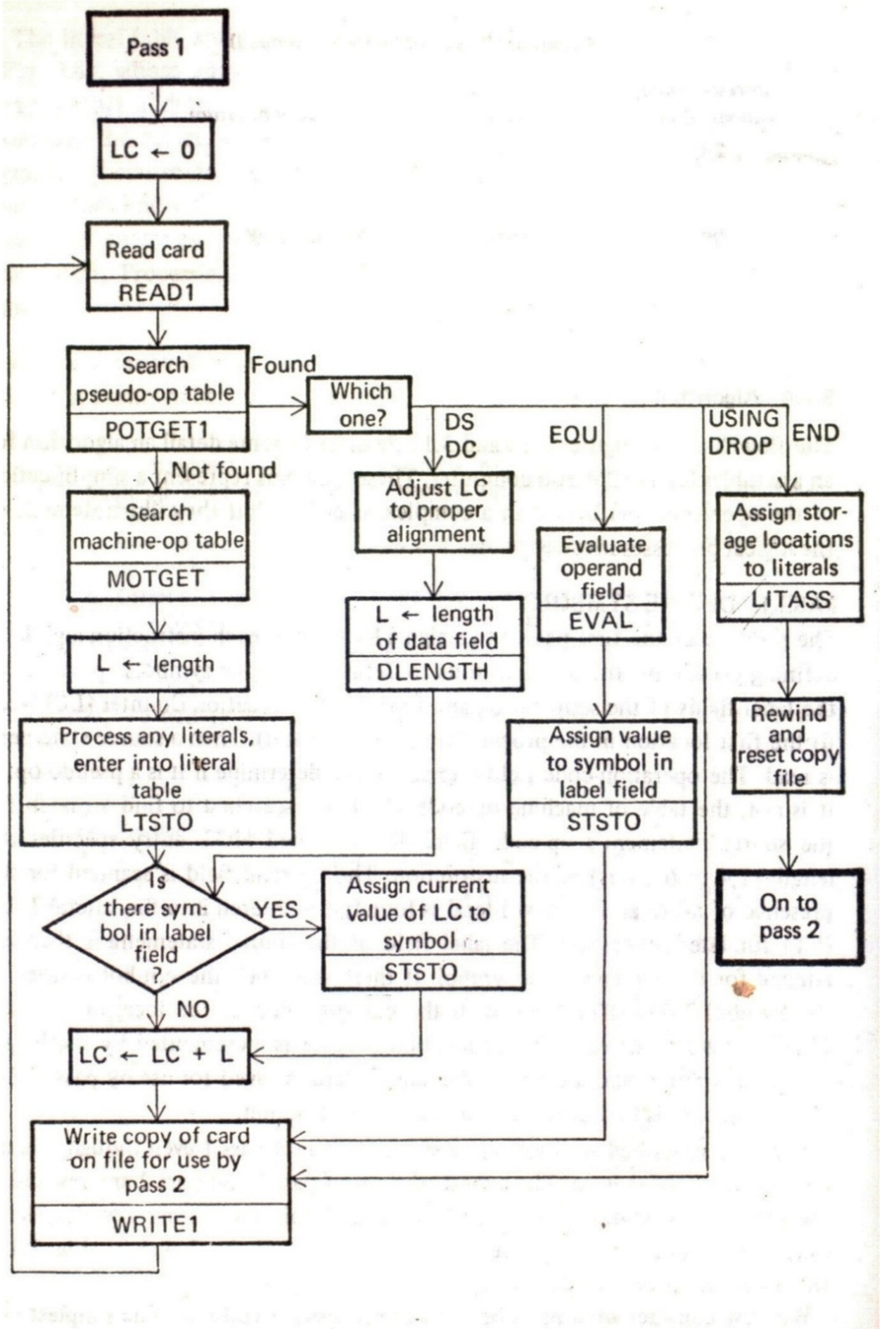

Pass 1
LC ← 0
Read card
READ1
Search pseudo-op table
POTGET1
Found
Which one?
Not found
DS DC
EQU
USING DROP
END
Search machine-op table
MOTGET
Adjust LC to proper alignment
L ← length of data field
DLENGTH
Evaluate operand field
EVAL
Assign storage locations to literals
LITASS
L ← length
Process any literals, enter into literal table
LTSTO
Assign value to symbol in label field
STSTO
Rewind and reset copy file
Is there symbol in label field ?
YES
NO
Assign current value of LC to symbol
STSTO
On to pass 2
LC ← LC + L
Write copy of card on file for use by pass 2
WRITE1

FIGURE 3.10 Detailed pass 1 flowchart

appropriate format for later processing by the loader, and print an assembly listing containing the original source and the hexadecimal equivalent of the bytes generated. The LC is initialized as in pass 1, and the processing continues as follows.

A card is read form the source file left by pass 1. As in pass 1, the op code field is examined to determine if it is a pseudo op; if it is not, the MOT is searched to find a match for the card's op code field. The matching MOT entry specifies the length, binary op code, and the format type of the instruction. The operand fields of the different instruction format types require somewhat different processing.

For the RR- format instructions, each of the two register specification field is evaluated. This evaluation may be very simple, as in: AR 2,3 or more complex, as in : MR EVEN, EVEN+1

The two fields are inserted into their respective four bit fields in the second byte of the RR instruction,

For RX format instruction, the register and index fields are evaluated and processed in the same way as the register specifications for RR format instructions, the storage address operand is evaluated to generate an Effective Address (EA). Then the BT must be examined to find a suitable base register (B) such that D=EA-c(B)<4096. The corresponding displacement can be determined. The 4 bit base register specification and 12 bit displacement field are then assembled into the third and fourth bytes of the instruction. Only the RR and RX instruction type are explicitly shown in the flowchart, the other instruction formats are handled similarly.

After the instruction has been assembled, it is put into the necessary format for later processing by the loader. Typically, several instructions are placed on single card. A listing line containing a copy of the source card, its assigned storage location, and its hexadecimal representation is then printed. Finally, the LC is incremented and processing is continued with the next card.

As in pass 1, each of the pseudo ops call for special processing. The EQU pseudo op requires very little processing in pass 2, because symbol definition was completed in pass 1. It is necessary only to print the EQU card as [art of the printed listing.

The USING and DROP pseudo ops, which were largely ignored in pass 1, require additional processing in pass 2. The operand fields of the pseudo ops are evaluated then the corresponding BT entry is either marked as available, if USING, or unavailable, if DROP. The BT is used extensively in pass 2 to compute the base and displacement fields for machine instructions with storage operands.

The DS and DC pseudo ops are processed essentially as in pass 1. In pass 2, however, actual code must be generated for the DC pseudo op. depending upon the data types specified, this involves various conversions (e.g. floating point character to binary representation) and symbol evaluations (e.g. address constants).

The END pseudo ops indicate the end of the source program and terminates the assembly. Various "housekeeping" tasks must be generated for any literal remaining on the LT.

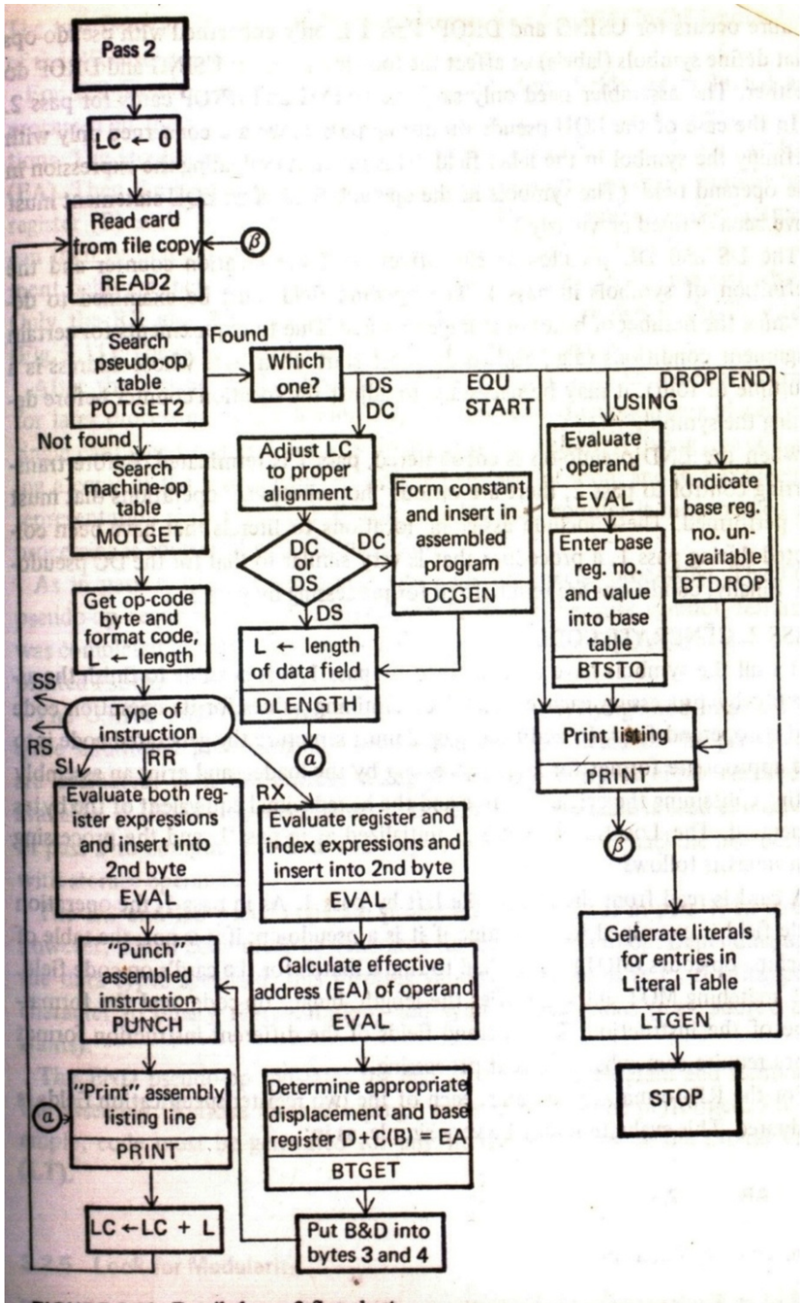

Pass 2
LC ← 0
Read card from file copy
READ2
Search pseudo-op table
POTGET2
Found
Which one?
DS DC
EQU START
USING
DROP
END
Not found
Search machine-op table
MOTGET
Adjust LC to proper alignment
DC or DS
DC
DS
Form constant and insert in assembled program
DCGEN
L ← length of data field
DLENGTH
Get op-code byte and format code, L ← length
Type of instruction
SS
RS
SI
RR
RX
Evaluate operand
EVAL
Enter base reg. no. and value into base table
BTSTO
Indicate base reg. no. unavailable
BTDROP
Print listing
PRINT
Evaluate both register expressions and insert into 2nd byte
EVAL
Evaluate register and index expressions and insert into 2nd byte
EVAL
"Punch" assembled instruction
PUNCH
Calculate effective address (EA) of operand
EVAL
Generate literals for entries in Literal Table
LTGEN
STOP
"Print" assembly listing line
PRINT
Determine appropriate displacement and base register D+C(B) = EA
BTGET
LC ← LC + L
Put B&D into bytes 3 and 4
α
β

FIGURE 3.11 Detailed pass 2 flowchart

LOOK FOR MODULARITY

We now review our design, looking for functions that can be isolated. Typically, such functions fall into two categories 1. Multi user and, 2. Unique.

In the flowchart for pass 1 and pass 2, we examine each step as a candidate for logical separation. Likely choices are identified in the flowcharts by the shapes where "name" is the name assigned to the function (e.g. MOTGET, EVAL, PRINT).

Listed below are some of the functions that may be isolated in the two passes

PASS 1

1.	READ1	--	Read the next assembly source card.
2.	POTGET1	--	Search the pass 1 POT for a match with the operation field of the current source card.
3.	MOTGET1	--	Search the MOT for a match with the operation of the current source card.
4.	STSTO	--	Store a label and its associated value into the ST. if the symbol is already in the table, return error indication (multiply defined symbols)
5.	LTSTO	--	Store the literal into the LT; donot store the same literal twice.
6.	WRITE1	--	Write a copy of the assembly source card on a storage device for use by pass 2.
7.	DLENGTH	--	Scan operand fields of DC or DC pseudo op to determine the amount of storage required.
8.	EVAL	--	Evaluate an arithmetic expression consisting of constants and symbols (e.g. ALPHA, 6 , GAMMA etc)
9	STGET	--	Search the ST for the entry corresponding to a specific symbol (used by STSTO and EVAL).
10	LITASS	--	Assign storage location to each literal in the literal table (may use DLENGTH).

PASS 2

1.	READ2	--	Reads the next assembly source card form the file copy.
2.	POTGET2	--	Similar to POTGET1 (search POT)
3.	MOTGET2	--	Same as in pass 1 (Search MOT)
4.	EVAL	--	Same as in pass 1 (evaluate expressions)
5.	PUNCH	--	Convert generated instruction to card format; punch card when it is filled with data
6.	PRINT	--	Converts relative location and generate code to character format: print the line along with copy of the same source card.
7.	DCGEN	--	Process the field of the DC pseudo to generate object code (uses EVAL and PUNCH)
8.	DLENGTH	--	Same as pass 1
9	BTSTO	--	Insert data into appropriate entry of BT
10	BTDROP	--	Insert "unavailable" indicator into appropriate entry of BT
11.	BTGET	--	Converts effective address into base and displacement by searching BT for available registers.
12.	LTGEN	--	Generate code for the literal (use DCGEN)

TABLE PROCESSING: SEARCHING AND SORTING

As it is clear from the above discussion, in the process of assembling a code written in assembly language involves a frequent use of a number of tables. A huge amount of table processing activities are involved in an assembler that basically deals with searching and sorting. Hence we should be well aware with the techniques that are employed for searching and sorting.

SEARCHING A TABLE

The symbols or data needed to be searched by the assembler whenever they are referred. Generally a keyword or *key* is given and the search for the appropriate item is done in the respective table. Basically *linear* and *binary search* are employed for searching.

LINEAR SEARCH

Linear search is basically used when the table is *unsorted*. Here the key is compared with entries of the table from the first to the nth item. Linear search is also called as *brute search*. Let T_L be the time required to find the item in the last location in the table, then the average time to search an item it $T_L/2$. Hence its complexity is of O(n).

BINARY SEARCH

Binary search is basically employed to a table with sorted key values. Here in each comparison the list is reduced to half. A item to be searched is compared with the middle item of the table. This lead to the following outcome:

- Item equal to the middle element: Here the item is found and the corresponding record is returned.
- Item more than the middle element: Here the item is searched in the lower half of the table.
- Item less than the middle element: Here the item is searched in the upper half of the table.

This process continues till the item search succeeds or fails. The searched list gets reduced each and every time so the average time taken in this case to search an item in the table is O(*log(n))*. Hence this search is much faster than the linear search but subject to constraint that the input list or table is a sorted one.

SORTING A TABLE

Basically the tables i.e. MOT, POT, etc that are generated in the assembling process are not sorted. So when we use think of applying binary search to these tables they needs to be sorted first. For this we have a number of techniques as follows:

INTERCHANGE SORT

Interchange sort is otherwise called as *bubble sort* or *sinking sort* or *shifting sort*. The sorting process involves an interchange of adjacent pair of elements and put them in order. Let us take an example and explain the sorting techniques. Let the list of number (keys) be 19, 13, 05, 27, 01, 26, 31, 16, 02, 09, 11 and 21. Then these are sorted as follows:

Unsorted List	1st pass	2nd pass	3rd pass	4th pass	5th pass	6th pass	7th pass
19	13	05	05	01	01	01	01
13	05	13	01	05	05	05	02
05	19	01	13	13	13	02	05
27	01	19	19	16	02	09	09
01	26	26	16	02	09	11	11
26	27	16	02	09	11	13	13
31	16	02	09	11	16	16	16
16	02	09	11	19	19	19	19
02	09	11	21	21	21	21	21
09	11	21	26	26	26	26	26
11	21	27	27	27	27	27	27
21	31	31	31	31	31	31	31

In each of these pass, elements are fixed to the last of the list. From the example 31, 27 , 26, 21, 19, 16 respectively are fixed in pass 1, 2, 3,4 and 5. Such a sort requires N*(N-1)/2 comparisons and thus would take time roughly proportional to N^2. We would like to have sorting algorithms that has time better than this. Basically, sorting can be divided into three categories that are as follows:

1. *Distributive:* Here the keys are checked digit wise.
2. *Comparative:* Here the two keys are compared.
3. *Address calculation:* Here the key is transformed to an address, where the item is closed to end up.

SHELL SORT

This is kind of comparative sort that compare two numbers *d* distance apart, where *d* is the length of the list. In pass the distance *d* is updated as (*d+1)/2* and the comparison and exchange process continues till a sorted list is obtained then value of *d* turns to be 1. This sorting take *log_2d* passes and since in each step it reduces the *d* to half, so it has a complexity equal to *$N*(log_2N)^2$*. The process can be clearly understood from the example given below.

Unsorted List	1st pass (d_1=6)	2nd pass (d_2=3)	3rd pass (d_2=2)	4th pass (d_2=1)
19	19	*09	*02	01
13	13	*01	01	02
05	*02	02	*09	05
27	*09	*19	*05	09
01	01	**11	11	11
26	*21	*05	**13	13
31	31	*27	**16	16
16	16	**13	*19	19
02	*05	*21	***21	21
09	*27	*31	*26	26
11	11	*16	*27	27
21	*26	26	*31	31

BUCKET SORT

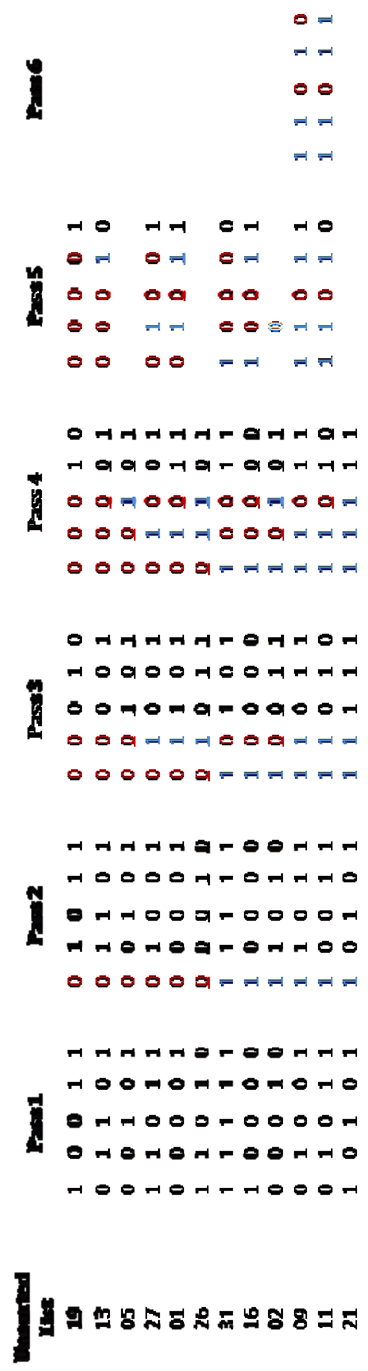
One of the simplest type of distributed sort is *radix sort* or *bucket sort*. It involves examining of the least significant digit of the keys first, and the same is assigned to as *bucket* uniquely dependent on the value of the digit. After the items have been distributed into buckets, they are merged in order and then the process is repeated until no more digits are left. A number with a base requires p buckets. Although this sorting is faster but suffers from serious disadvantages that are as follows:

1. It involves two processes, a separation and a merge.
2. It requires a lot of extra storage for the buckets. This can be overcome chaining records within a logical record rather than predefining maximum size to bucket.

The process can be clear from the example given below.

Unsorted List	Pass 1		Pass 2	
	Distribution	Merge	Distribution	Merge
19		01		01
13	0)	31	0)01,02,05,09	02
05	1)01,31,11,21	11	1)11,13,16,19	05
27	2)02	21	2)21,26,27	09
01	3)13	02	3)31	11
26	4)	13	4)	13
31	5)05	05	5)	16
16	6)26,16	26	6)	19
02	7)27	16	7)	21
09	8)	27	8)	26
11	9)19,09	19	9)	27
21		09		31

RADIX EXCHANGE SORT



Radix exchange sort can only be applied to table that key values in binary. The ordering is done by taking groups with M common bits and ordering that group with respect to M+1 bits. The process of sorting actually involves comparing the leftmost bit in the list and then dividing the list into two by performing a number of exchanges of 1 that is near the top and a zero that is near the bottom. And after this division the two sub lists are sorted in a similar manner till all the keys are ordered. The number of passed required to finish the sorting process is equal to the number of digits in the keys. Here for the given example the number of passes is 5, since the number of digits in the keys is 5.

The complexity of the radix exchange sort is N*log(N) as compared to bucket sort that is $N*\log_p(N)$.

ADDRESS CALCULATION SORT

This sorting is very fast if there is enough storage space. To sort by address calculation we require a space that is sufficiently more than the size of the list to be sorted. Here we need to find a *factor* that will be helping us in calculation the address. This *factor=ceiling(Max element/number of item in the list)*. In our example we have *factor=ceiling(31/12)=6.* The table where the number is to stored is labeled form *0* to *n-1*. Then the address is calculated by dividing the input number by the factor and the item is inserted in the found place if it empty. If it is not empty then elements in the table next to address are pushed down the table to a vacant location there by creating space for the new item.

Data number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Data	**19**	**13**	**05**	**27**	**01**	**26**	**31**	**16**	**02**	**09**	**11**	**21**
Calculated address	**6**	**4**	**1**	**9**	**0**	**8**	**10**	**5**	**0**	**3**	**3**	**7**
0					01	01	01	01	01	01	01	01
1			05	05	05	05	05	05	02	02	02	02
2									05	05	05	05
3										09	09	09
4		13	13	13	13	13	13	13	13	13	11	11
5								16	16	16	13	13
6	19	19	19	19	19	19	19	19	19	19	16	16
7											19	19
8						26	26	26	26	26	26	21
9				27	27	27	27	27	27	27	27	26
10							31	31	31	31	31	27
11												31

HASH OR RANDOM ENTRY SEARCHING

It not always that for faster searching we need to sort the list so that we can apply binary search. Here the data or keys are also packed (no spaces between the data).

We can also apply hash search to an unordered and unpacked list, still we get faster results. Here we define the size of the table. In our example it is 17. The item to be searched is divided by 17 and the remainder gives the location where the item is to be searched or stored. It the address generated is not empty then the item is place in the next place that may be immediate or after some location. Now when we think of searching an item in this table, we call this as a *probe*. If the item is stored at the calculated address then the probe is 1 else it is equals to the number of spaces below its position where it was stored. Similarly the probes not to find an element are number of probes for finding an empty space. We have

Positions	items	Probes to find	Probes to find not
0			1
1	01	1	6
2	19,02	1	5
3	02	2	4
4	21	1	3
5	05	1	2
6			1
7			1
8			1
9	26,09	1	7
10	27,09	1	6
11	09,11	3	5
12	11	2	4
13	13	1	3
14	31	1	2
15			1
16	16	1	1
		16	54

$T_P=1-(\rho/2)(1/(1-\rho))$ (Probes to search)

$T_P=(1/(1-\rho))$ (Probes to not to find) Also T_p = Probes to store

COMPARISON OF SORTS

Type	Average time (approx)	Extra storage (wasted space)
Interchange	$A*N^2$	None
Shell	$B*N*(\log_2(N))^2$	None
Radix	$C*N*\log_p(N)$	N*p
Radix exchange	$D*N*\log_2(N)$	k+1
Address calculation	E*N	2.2*N